Apprentissage par Renforcement

Notes de Cours

Master M2 — 2025–2026

Yaë Ulrich Gaba

« Une machine peut-elle penser ? C'est une question aussi pertinente que : un sous-marin peut-il nager ? »
— Edsger Dijkstra

25 mars 2026

Table des matières

Préface

Objectifs de cet ouvrage

L'apprentissage par renforcement profond (*Deep Reinforcement Learning*, DRL) combine l'apprentissage par renforcement classique et l'apprentissage profond. Au cours de la dernière décennie, cette discipline a permis des avancées spectaculaires : vaincre des champions humains aux jeux de Go et d'Atari, contrôler des robots manipulateurs avec une dextérité remarquable, optimiser des stratégies financières et concevoir de nouvelles molécules thérapeutiques.

Cet ouvrage s'adresse aux étudiants de Master et de Doctorat en informatique, mathématiques appliquées ou sciences de l'ingénieur, ainsi qu'aux chercheurs et praticiens souhaitant acquérir une compréhension rigoureuse des fondements théoriques et des méthodes algorithmiques du DRL.

Organisation du cours

Le cours est structuré en onze chapitres, organisés selon une progression pédagogique allant des fondements mathématiques aux applications de pointe :

1. **Processus de décision markoviens (MDP)** : le cadre mathématique fondamental qui formalise l'interaction agent-environnement.
2. **Programmation dynamique** : les algorithmes d'itération sur la valeur et d'itération sur la politique, fondements de toute méthode de RL.
3. **Méthodes de Monte Carlo et différences temporelles** : les approches sans modèle fondées sur l'échantillonnage.
4. **Q-Learning et SARSA** : les algorithmes tabulaires fondamentaux d'apprentissage hors-politique et en-politique.
5. **Réseaux Q profonds (DQN)** : la révolution de l'approximation de fonction par réseaux de neurones appliquée au RL.
6. **Gradient de politique — REINFORCE** : l'optimisation directe de la politique par montée de gradient.
7. **Méthodes acteur-critique (A3C, PPO)** : la combinaison de l'estimation de la valeur et de l'optimisation de la politique.
8. **État de l'art (SAC, TD3, DDPG)** : les algorithmes modernes pour les espaces d'action continus.

9. **Apprentissage par renforcement multi-agents** : la généralisation aux systèmes à agents multiples.

10. **RL sous contraintes et sûreté** : la prise en compte de contraintes de sécurité dans l'apprentissage.

11. **Applications** : robotique, jeux, finance et au-delà.

Prérequis

Le lecteur devra posséder des connaissances solides en :

- **Probabilités et statistiques** : variables aléatoires, espérance conditionnelle, chaînes de Markov, convergence stochastique.

- **Optimisation** : descente de gradient, convexité, multiplicateurs de Lagrange, méthodes numériques.

- **Algèbre linéaire** : espaces vectoriels, décompositions matricielles, valeurs propres.

- **Apprentissage automatique** : réseaux de neurones, rétropropagation, régularisation, architectures profondes (CNN, RNN).

- **Programmation Python** : maîtrise de NumPy, PyTorch et des environnements Gymnasium (anciennement OpenAI Gym).

Notations

Nous adoptons les conventions suivantes tout au long de l'ouvrage :

Symbole	Signification
$\mathcal{S}$	Ensemble des états
$\mathcal{A}$	Ensemble des actions
$\mathcal{R}$	Ensemble des récompenses
$\gamma \in [0, 1)$	Facteur d'actualisation
$\pi(a \mid s)$	Politique stochastique
$V^\pi(s)$	Fonction de valeur d'état sous π
$Q^\pi(s, a)$	Fonction de valeur d'action sous π
$A^\pi(s, a)$	Fonction avantage sous π
$P(s' \mid s, a)$	Probabilité de transition
$R(s, a, s')$	Fonction de récompense
$\mathbb{E}[\cdot]$	Espérance mathématique
$\mathbb{P}(\cdot)$	Probabilité
∇_θ	Gradient par rapport à θ
θ	Paramètres de la politique ou du réseau
α	Taux d'apprentissage
ϵ	Paramètre d'exploration (ϵ-greedy)
τ	Coefficient de mise à jour douce
$\mathcal{D}$	Mémoire de rejeu (*replay buffer*)

Conventions typographiques

- Les **définitions** sont présentées dans des encadrés bleus.
- Les **théorèmes** et **propositions** apparaissent dans des encadrés formels avec preuves.
- Les **algorithmes** sont présentés en pseudo-code dans des encadrés dédiés.
- Les **implémentations** PyTorch/Gymnasium sont données dans des blocs de code syntaxiquement colorés.
- Les **formules clés** sont mises en évidence dans des encadrés spéciaux.
- Les points nécessitant une **attention particulière** sont signalés dans des encadrés d'avertissement.

Remerciements

Cet ouvrage est le fruit de plusieurs années d'enseignement et de recherche en apprentissage par renforcement. Nous remercions les étudiants qui, par leurs questions et leur curiosité, ont contribué à améliorer la présentation de ces sujets. Nous exprimons

également notre gratitude envers la communauté open source, en particulier les développeurs de PyTorch, Gymnasium, Stable-Baselines3 et CleanRL, dont les outils rendent l'apprentissage par renforcement accessible à tous.

Comment utiliser ce livre

Intuition

Chaque chapitre suit une structure cohérente :

1. **Motivation et intuition** — Pourquoi cette méthode ? Quel problème résout-elle ?
2. **Fondements théoriques** — Définitions, théorèmes, preuves de convergence.
3. **Algorithmes** — Pseudo-code détaillé avec analyse de complexité.
4. **Implémentation** — Code PyTorch fonctionnel et reproductible.
5. **Exercices** — Problèmes théoriques et pratiques de difficulté croissante.

Bref historique

L'apprentissage par renforcement plonge ses racines dans les travaux de Bellman (programmation dynamique, années 1950) et de la psychologie comportementale (conditionnement opérant de Skinner). Les jalons majeurs incluent :

- **1989** : Watkins introduit le Q-Learning, premier algorithme hors-politique convergent pour les MDP.
- **1992** : TD-Gammon de Tesauro apprend le backgammon par auto-jeu avec différences temporelles.
- **2013** : Mnih et al. (DeepMind) combinent DQN avec des réseaux convolutifs pour jouer aux jeux Atari à partir de pixels.
- **2015** : Publication de DQN dans *Nature* ; Silver et al. développent AlphaGo.
- **2016** : AlphaGo bat Lee Sedol, champion mondial de Go.
- **2017** : Schulman et al. introduisent PPO ; Haarnoja et al. proposent SAC.
- **2019** : OpenAI Five bat les champions du monde de Dota 2.
- **2020–2024** : RL appliqué à la découverte de médicaments, à la fusion nucléaire (DeepMind/EPFL), à l'alignement de LLM (RLHF).

Ressources complémentaires

Pour approfondir les sujets abordés dans cet ouvrage, nous recommandons :

- **Sutton & Barto** — *Reinforcement Learning : An Introduction* (2e édition, 2018). L'ouvrage de référence en RL classique.
- **Bertsekas** — *Dynamic Programming and Optimal Control* (4e édition). Traitement rigoureux de la programmation dynamique.
- **Szepesvári** — *Algorithms for Reinforcement Learning*. Synthèse concise des algorithmes fondamentaux.
- **Agarwal et al.** — *Reinforcement Learning : Theory and Algorithms* (2022). Traitement théorique moderne.
- **Cours en ligne** : David Silver (UCL), Sergey Levine (UC Berkeley), Emma Brunskill (Stanford).

[Author Name]
Mars 2026

Chapitre 1

Processus de Décision Markovien

Imaginez un robot qui apprend à marcher. À chaque instant, il observe sa posture (l'*état*), choisit une force à appliquer à ses articulations (l'*action*), et reçoit un signal — est-il resté debout ou est-il tombé ? (la *récompense*). Son objectif : trouver une stratégie d'actions qui maximise les récompenses accumulées au fil du temps. Ce schéma — état, action, récompense, nouvel état — est universel : il décrit aussi bien un joueur d'échecs, un véhicule autonome qu'un algorithme de trading. La formalisation mathématique de ce schéma porte un nom : le *processus de décision markovien* (MDP), formalisé par Richard Bellman dans les années 1950.

Intuition

Un processus de décision markovien (MDP) est le modèle mathématique central de l'apprentissage par renforcement. Il formalise la manière dont un agent interagit avec un environnement en prenant des décisions séquentielles pour maximiser une récompense cumulée à long terme.

1.1 Introduction

L'apprentissage par renforcement repose sur une idée simple : un agent apprend à agir en interagissant avec un environnement. À chaque pas de temps t, l'agent observe un état $s_t \in \mathcal{S}$, choisit une action $a_t \in \mathcal{A}$, reçoit une récompense $r_{t+1} \in \mathbb{R}$ et transite vers un nouvel état s_{t+1}. Le formalisme mathématique qui capture cette interaction est le *processus de décision markovien.*

1.2 Définition formelle

Définition 1.1 (Processus de décision markovien)**.** Un MDP est un quintuplet $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$ où :

- $\mathcal{S}$ est un ensemble fini ou dénombrable d'**états**,
- $\mathcal{A}$ est un ensemble fini ou dénombrable d'**actions**,
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ est la **fonction de transition** : $P(s' \mid s, a) = \mathbb{P}(S_{t+1} = s' \mid S_t = s, A_t = a)$,

- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ est la **fonction de récompense** : $R(s, a, s')$ est la récompense reçue lors de la transition de s à s' sous l'action a,
- $\gamma \in [0, 1)$ est le **facteur d'actualisation**.

Propriété de Markov

La propriété de Markov stipule que l'état futur ne dépend que de l'état présent et de l'action courante :

$$\mathbb{P}(S_{t+1} = s' \mid S_0, A_0, S_1, A_1, \ldots, S_t, A_t) = \mathbb{P}(S_{t+1} = s' \mid S_t, A_t) = P(s' \mid S_t, A_t).$$

1.3 Politique

Définition 1.2 (Politique)**.** Une **politique** π est une règle de décision qui prescrit le comportement de l'agent.

- **Politique déterministe** : $\pi : \mathcal{S} \to \mathcal{A}$, où $a = \pi(s)$.
- **Politique stochastique** : $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, où $\pi(a \mid s) = \mathbb{P}(A_t = a \mid S_t = s)$ avec $\sum_{a \in \mathcal{A}} \pi(a \mid s) = 1$ pour tout s.

Définition 1.3 (Politique stationnaire)**.** Une politique π est dite **stationnaire** si elle ne dépend pas du temps : $\pi_t = \pi$ pour tout $t \geq 0$. Sauf mention contraire, toutes les politiques considérées dans ce cours sont stationnaires.

1.4 Fonctions de valeur

Définition 1.4 (Retour actualisé)**.** Le **retour actualisé** à partir du temps t est :

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

Le facteur $\gamma < 1$ assure la convergence de cette somme lorsque les récompenses sont bornées.

Définition 1.5 (Fonction de valeur d'état)**.** La **fonction de valeur d'état** sous la politique π est :

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right].$$

Définition 1.6 (Fonction de valeur d'action)**.** La **fonction de valeur d'action** (ou fonction Q) sous la politique π est :

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right].$$

Remarque 1.7. La relation entre V^π et Q^π est :

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s)\, Q^\pi(s, a).$$

1.5 Équations de Bellman

Théorème 1.8 (Équation de Bellman pour V^π)**.** *Pour toute politique π et tout état $s \in \mathcal{S}$:*

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

Démonstration. Par définition du retour actualisé et de la propriété de Markov :

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s'] \right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, V^\pi(s') \right]. \end{aligned}$$

□

Théorème 1.9 (Équation de Bellman pour Q^π)**.** *Pour toute politique π, tout état s et toute action a :*

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

Démonstration. On développe de manière similaire :

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s'] \right] \\ &= \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \sum_{a'} \pi(a' \mid s') Q^\pi(s', a') \right]. \end{aligned}$$

□

Équations de Bellman — Résumé

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V^\pi(s')] \tag{1.1}$$

$$Q^\pi(s, a) = \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s', a')] \tag{1.2}$$

1.6 Politique optimale et équations de Bellman d'optimalité

Définition 1.10 (Politique optimale)**.** Une politique π^* est **optimale** si pour tout état $s \in \mathcal{S}$:

$$V^{\pi^*}(s) \geq V^\pi(s) \quad \text{pour toute politique } \pi.$$

On note $V^*(s) = V^{\pi^*}(s)$ et $Q^*(s, a) = Q^{\pi^*}(s, a)$.

Théorème 1.11 (Équations de Bellman d'optimalité)**.** *Les fonctions de valeur optimales satisfont :*

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^*(s') \right] \tag{1.3}$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right] \tag{1.4}$$

Démonstration. Pour l'équation (1.3), on utilise le fait que la politique optimale choisit l'action maximisant la valeur :

$$\begin{aligned} V^*(s) &= \max_{\pi} V^{\pi}(s) \\ &= \max_{a} Q^*(s,a) \\ &= \max_{a} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]. \end{aligned}$$

Pour l'équation (1.4), on injecte $V^*(s') = \max_{a'} Q^*(s',a')$ dans l'équation de Bellman pour Q^*. □

Théorème 1.12 (Existence d'une politique optimale déterministe)**.** *Pour tout MDP fini, il existe au moins une politique optimale déterministe $\pi^* : \mathcal{S} \to \mathcal{A}$ définie par :*

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a).$$

1.7 Opérateur de Bellman et contraction

Définition 1.13 (Opérateur de Bellman)**.** L'**opérateur de Bellman $\mathcal{T}^\pi$** pour une politique π est défini par :

$$(\mathcal{T}^\pi V)(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')].$$

L'**opérateur de Bellman d'optimalité** est :

$$(\mathcal{T}^* V)(s) = \max_{a} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')].$$

Théorème 1.14 (Contraction de l'opérateur de Bellman)**.** *Les opérateurs $\mathcal{T}^\pi$ et $\mathcal{T}^*$ sont des contractions en norme infinie avec facteur γ :*

$$\left\| \mathcal{T}^\pi V - \mathcal{T}^\pi U \right\|_\infty \leq \gamma \left\| V - U \right\|_\infty .$$

Par le théorème du point fixe de Banach, chacun possède un unique point fixe : V^π pour $\mathcal{T}^\pi$ et V^* pour $\mathcal{T}^*$.

Démonstration. Pour tout $s \in \mathcal{S}$:

$$\begin{aligned} |(\mathcal{T}^* V)(s) - (\mathcal{T}^* U)(s)| &= \left| \max_a \sum_{s'} P(s'|s,a)[R + \gamma V(s')] - \max_a \sum_{s'} P(s'|s,a)[R + \gamma U(s')] \right| \\ &\leq \max_a \sum_{s'} P(s'|s,a) \gamma \left| V(s') - U(s') \right| \\ &\leq \gamma \left\| V - U \right\|_\infty . \end{aligned}$$

Le passage de la deuxième à la troisième ligne utilise l'inégalité $|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|$. □

1.8 Fonction avantage

Définition 1.15 (Fonction avantage)**.** La **fonction avantage** sous la politique π est :

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

Elle mesure l'avantage relatif de prendre l'action a dans l'état s par rapport au comportement moyen sous π.

Proposition 1.16. Pour toute politique π et tout état s :

$$\sum_{a \in \mathcal{A}} \pi(a \mid s)\, A^\pi(s,a) = 0.$$

Démonstration. $\sum_a \pi(a|s) A^\pi(s,a) = \sum_a \pi(a|s)[Q^\pi(s,a) - V^\pi(s)] = V^\pi(s) - V^\pi(s) = 0.$ □

1.9 Exemples de MDP

Exemple 1.17 (Gridworld)**.** Considérons une grille 4×4 où un agent se déplace dans les quatre directions cardinales. L'état terminal est la case $(4,4)$ avec récompense $+1$. Chaque déplacement coûte -0.01. Les mouvements contre les murs laissent l'agent sur place. Ce problème se formalise comme un MDP avec :

- $\mathcal{S} = \{(i,j) : 1 \le i,j \le 4\}$, $|\mathcal{S}| = 16$,
- $\mathcal{A} = \{\text{haut, bas, gauche, droite}\}$,
- Transitions déterministes (sauf aux bords),
- $\gamma = 0.99$.

Exemple 1.18 (Bandit à K bras)**.** Un bandit à K bras est un MDP dégénéré avec $|\mathcal{S}| = 1$ (un seul état). L'agent choisit à chaque pas un bras $a \in \{1, \dots, K\}$ et reçoit une récompense aléatoire $R_a \sim \mathcal{D}_a$. Ce cas particulier permet d'étudier le dilemme exploration-exploitation de manière isolée.

1.10 Implémentation en Python

Définition d'un MDP simple avec Gymnasium

```python
import numpy as np
import gymnasium as gym
from gymnasium import spaces

class GridWorldEnv(gym.Env):
    """Environnement Gridworld 4x4 personnalise."""
    metadata = {"render_modes": ["human"]}

    def __init__(self, size=4, gamma=0.99):
```

```
        super().__init__()
        self.size = size
        self.gamma = gamma
        self.observation_space = spaces.Discrete(size * size)
        self.action_space = spaces.Discrete(4)  # haut, bas, gauche,
        ↪  droite
        self.goal = (size - 1, size - 1)
        self._action_to_dir = {
            0: np.array([-1, 0]),  # haut
            1: np.array([1, 0]),   # bas
            2: np.array([0, -1]), # gauche
            3: np.array([0, 1]),  # droite
        }
        self.reset()

    def _pos_to_state(self, pos):
        return pos[0] * self.size + pos[1]

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.agent_pos = np.array([0, 0])
        return self._pos_to_state(self.agent_pos), {}

    def step(self, action):
        direction = self._action_to_dir[action]
        new_pos = np.clip(self.agent_pos + direction, 0, self.size - 1)
        self.agent_pos = new_pos

        terminated = tuple(self.agent_pos) == self.goal
        reward = 1.0 if terminated else -0.01
        return self._pos_to_state(self.agent_pos), reward, terminated,
        ↪  False, {}
```

Construction de la matrice de transition

```
def build_transition_matrix(env):
    """Construit P[s, a, s'] et R[s, a, s'] pour un MDP fini."""
    nS = env.observation_space.n
    nA = env.action_space.n
    P = np.zeros((nS, nA, nS))
    R = np.zeros((nS, nA, nS))

    for s in range(nS):
        for a in range(nA):
            # Simuler la transition depuis l'etat s avec l'action a
            env.agent_pos = np.array([s // env.size, s % env.size])
            s_next, reward, _, _, _ = env.step(a)
            P[s, a, s_next] = 1.0  # transition deterministe
            R[s, a, s_next] = reward
    return P, R
```

```python
env = GridWorldEnv(size=4)
P, R = build_transition_matrix(env)
print(f"Forme de P: {P.shape}")  # (16, 4, 16)
print(f"Forme de R: {R.shape}")  # (16, 4, 16)
```

1.11 Résolution algébrique directe

Proposition 1.19 (Résolution matricielle de l'équation de Bellman)**.** Pour un MDP fini avec $|\mathcal{S}| = n$, l'équation de Bellman pour V^π peut s'écrire sous forme matricielle :

$$\mathbf{v}^\pi = \mathbf{r}^\pi + \gamma\, \mathbf{P}^\pi \mathbf{v}^\pi$$

où $\mathbf{P}^\pi_{s,s'} = \sum_a \pi(a|s) P(s'|s,a)$ et $\mathbf{r}^\pi_s = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) R(s,a,s')$. La solution est :

$$\mathbf{v}^\pi = (I - \gamma \mathbf{P}^\pi)^{-1} \mathbf{r}^\pi.$$

Complexité de la résolution directe

L'inversion matricielle a une complexité $O(n^3)$, ce qui la rend impraticable pour des MDP avec un grand nombre d'états. C'est pourquoi on préfère les méthodes itératives (Chapitre 2).

1.12 Extensions du modèle MDP

Définition 1.20 (POMDP)**.** Un **processus de décision markovien partiellement observable** (POMDP) est un MDP augmenté d'un ensemble d'observations $\mathcal{O}$ et d'une fonction d'observation $O : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{O})$. L'agent n'observe pas directement l'état s_t mais une observation $o_t \sim O(\cdot|s_t, a_{t-1})$.

Définition 1.21 (MDP à horizon fini)**.** Un MDP à **horizon fini** H est un MDP où l'interaction se termine après exactement H pas de temps. Le retour est :

$$G_t = \sum_{k=0}^{H-t-1} \gamma^k R_{t+k+1}.$$

Définition 1.22 (MDP continu)**.** Lorsque $\mathcal{S} \subseteq \mathbb{R}^n$ et/ou $\mathcal{A} \subseteq \mathbb{R}^m$, on parle de MDP à espace d'états ou d'actions continu. Les sommes sont remplacées par des intégrales dans les équations de Bellman.

1.13 Exercices

Exercice 1.1 (Équation de Bellman)**.** Considérons un MDP à 3 états $\{s_1, s_2, s_3\}$ avec $\gamma = 0.9$ et la politique uniforme $\pi(a|s) = 1/2$ pour deux actions. Les transitions et récompenses sont :

- De s_1, action a_1 : vers s_2 avec $R = 1$; action a_2 : vers s_3 avec $R = 0$.

- De s_2, toute action : vers s_1 avec $R = 2$.
- De s_3, toute action : vers s_3 avec $R = 0$ (état absorbant).

Écrivez le système d'équations de Bellman pour V^π et résolvez-le.

Exercice 1.2 (Opérateur de contraction)**.** Montrez que si $\gamma = 0$, l'opérateur de Bellman d'optimalité converge en une seule itération. Quelle est l'interprétation de $\gamma = 0$?

Exercice 1.3 (Implémentation)**.** Implémentez un environnement `FrozenLake` personnalisé avec Gymnasium. Construisez la matrice de transition et calculez V^π par résolution matricielle pour la politique uniforme. Vérifiez votre résultat en comparant avec l'environnement `FrozenLake-v1` de Gymnasium.

Exercice 1.4 (Fonction avantage)**.** Montrez que pour une politique optimale déterministe π^* :

$$A^{\pi^*}(s, \pi^*(s)) = 0 \quad \text{et} \quad A^{\pi^*}(s, a) \leq 0 \quad \forall a \neq \pi^*(s).$$

Chapitre 2

Programmation Dynamique

Richard Bellman, dans les années 1950, a eu l'une de ces idées qui semblent évidentes *après coup* : pour résoudre un problème de décision séquentiel, on peut le découper en sous-problèmes plus simples et remonter de la fin vers le début. Il a appelé cette approche « programmation dynamique » — un nom choisi, dit-il, pour impressionner ses supérieurs au département de la Défense. L'idée est devenue le fondement théorique de tout l'apprentissage par renforcement.

Intuition

La programmation dynamique (DP) résout les MDP lorsque le modèle de transition $P(s'|s,a)$ et la fonction de récompense $R(s,a,s')$ sont parfaitement connus. Elle constitue le socle théorique sur lequel reposent toutes les méthodes d'apprentissage par renforcement.

2.1 Principe de la programmation dynamique

La programmation dynamique, introduite par Richard Bellman dans les années 1950, exploite la structure récursive des équations de Bellman pour calculer itérativement les fonctions de valeur. Les deux algorithmes fondamentaux sont :

1. **L'évaluation de politique** (*policy evaluation*) : calculer V^π pour une politique donnée π.
2. **L'amélioration de politique** (*policy improvement*) : construire une meilleure politique à partir de V^π.

2.2 Évaluation de politique

Définition 2.1 (Évaluation de politique itérative)**.** L'évaluation de politique itérative calcule V^π en appliquant répétitivement l'opérateur de Bellman $\mathcal{T}^\pi$:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')] \quad \forall s \in \mathcal{S}.$$

Théorème 2.2 (Convergence de l'évaluation de politique)**.** *Pour toute initialisation V_0, la suite $(V_k)_{k\geq 0}$ définie par $V_{k+1} = \mathcal{T}^\pi V_k$ converge vers V^π quand $k \to \infty$. De plus :*

$$\|V_k - V^\pi\|_\infty \leq \gamma^k \|V_0 - V^\pi\|_\infty .$$

Démonstration. C'est une conséquence directe du théorème 1.14. L'opérateur $\mathcal{T}^\pi$ est une γ-contraction en norme infinie, donc par le théorème du point fixe de Banach, la suite converge géométriquement vers l'unique point fixe V^π. □

Évaluation de politique itérative

1. Initialiser $V(s) \leftarrow 0$ pour tout $s \in \mathcal{S}$
2. **Répéter** jusqu'à convergence ($\Delta < \theta$) :
   (a) $\Delta \leftarrow 0$
   (b) Pour chaque $s \in \mathcal{S}$:
      i. $v \leftarrow V(s)$
      ii. $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$
      iii. $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
3. Retourner $V \approx V^\pi$

2.3 Amélioration de politique

Théorème 2.3 (Théorème d'amélioration de politique)**.** *Soit π une politique et π' la politique gloutonne par rapport à V^π :*

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

Alors $V^{\pi'}(s) \geq V^\pi(s)$ pour tout $s \in \mathcal{S}$, avec égalité si et seulement si π est déjà optimale.

Démonstration. Pour tout état s :

$$\begin{aligned}
V^\pi(s) &\leq \max_a Q^\pi(s,a) = Q^\pi(s, \pi'(s)) \\
&= \sum_{s'} P(s'|s,\pi'(s))[R(s,\pi'(s),s') + \gamma V^\pi(s')] \\
&\leq \sum_{s'} P(s'|s,\pi'(s))[R(s,\pi'(s),s') + \gamma Q^\pi(s',\pi'(s'))] \\
&= \sum_{s'} P(s'|s,\pi'(s)) \Big[R + \gamma \sum_{s''} P(s''|s',\pi'(s'))[R' + \gamma V^\pi(s'')]\Big] \\
&\leq \cdots \leq V^{\pi'}(s).
\end{aligned}$$

On obtient le résultat par application récursive de l'inégalité. □

2.4 Itération sur la politique

Itération sur la politique (*Policy Iteration*)

1. Initialiser $\pi(s)$ arbitrairement pour tout s
2. **Répéter** :
   (a) **Évaluation** : calculer V^π par évaluation itérative
   (b) **Amélioration** : pour chaque s :
   $$\pi'(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^\pi(s')]$$
   (c) Si $\pi' = \pi$, **arrêter** (politique optimale trouvée)
   (d) $\pi \leftarrow \pi'$
3. Retourner $\pi^* = \pi$

Théorème 2.4 (Convergence de l'itération sur la politique)**.** *L'itération sur la politique converge vers la politique optimale π^* en un nombre fini d'itérations (au plus $|\mathcal{A}|^{|\mathcal{S}|}$ itérations, car le nombre de politiques déterministes est fini et chaque itération améliore strictement la politique).*

2.5 Itération sur la valeur

Définition 2.5 (Itération sur la valeur)**.** L'itération sur la valeur (*value iteration*) combine évaluation et amélioration en une seule mise à jour :

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')] \quad \forall s \in \mathcal{S}.$$

Théorème 2.6 (Convergence de l'itération sur la valeur)**.** *La suite (V_k) définie par l'itération sur la valeur converge vers V^* :*

$$\|V_k - V^*\|_\infty \leq \gamma^k \|V_0 - V^*\|_\infty .$$

Le nombre d'itérations pour atteindre une précision ϵ est :

$$k \geq \frac{\log(\|V_0 - V^*\|_\infty / \epsilon)}{\log(1/\gamma)}.$$

Itération sur la valeur (*Value Iteration*)

1. Initialiser $V(s) \leftarrow 0$ pour tout $s \in \mathcal{S}$
2. **Répéter** jusqu'à convergence ($\Delta < \theta$) :
   (a) $\Delta \leftarrow 0$
   (b) Pour chaque $s \in \mathcal{S}$:

i. $v \leftarrow V(s)$
ii. $V(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$
iii. $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

3. Extraire la politique : $\pi^*(s) = \arg\max_a \sum_{s'} P(s'|s,a)[R + \gamma V(s')]$
4. Retourner V^*, π^*

2.6 Comparaison des méthodes

	Itération politique	**Itération valeur**
Coût par itération	$O(\|\mathcal{S}\|^3 + \|\mathcal{S}\|^2\|\mathcal{A}\|)$	$O(\|\mathcal{S}\|^2\|\mathcal{A}\|)$
Nombre d'itérations	Peu (souvent < 10)	Plus ($O(1/\log(1/\gamma))$)
Mémoire	$V + \pi$	V seul
Convergence	Finie, exacte	Asymptotique

Remarque 2.7. En pratique, l'itération sur la politique converge souvent en très peu d'itérations (5–10), même pour de grands MDP. L'itération sur la valeur nécessite plus d'itérations mais chacune est moins coûteuse car elle évite la résolution complète de l'évaluation de politique.

2.7 Itération sur la politique généralisée (GPI)

Définition 2.8 (GPI — Generalized Policy Iteration)**.** L'**itération sur la politique généralisée** désigne toute alternance entre évaluation (partielle ou complète) de politique et amélioration de politique. Presque tous les algorithmes de RL peuvent être vus comme des instances de GPI.

Remarque 2.9. L'itération sur la valeur est un cas limite de GPI où l'évaluation ne fait qu'une seule itération de Bellman avant l'amélioration.

2.8 Implémentation en Python

Évaluation de politique

```python
import numpy as np

def policy_evaluation(P, R, pi, gamma=0.99, theta=1e-8):
    """
    Evaluation iterative de politique.
    P: matrice de transition (nS, nA, nS)
    R: matrice de recompense (nS, nA, nS)
    pi: politique deterministe (nS,) -> indices d'action
    """
```

```
    nS = P.shape[0]
    V = np.zeros(nS)

    while True:
        delta = 0.0
        for s in range(nS):
            a = pi[s]
            v = V[s]
            V[s] = np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    return V
```

Itération sur la politique

```
def policy_iteration(P, R, gamma=0.99, theta=1e-8):
    """Iteration sur la politique complete."""
    nS, nA = P.shape[0], P.shape[1]
    pi = np.zeros(nS, dtype=int)  # politique initiale

    while True:
        # Evaluation
        V = policy_evaluation(P, R, pi, gamma, theta)

        # Amelioration
        stable = True
        for s in range(nS):
            old_action = pi[s]
            Q_s = np.array([
                np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
                for a in range(nA)
            ])
            pi[s] = np.argmax(Q_s)
            if old_action != pi[s]:
                stable = False

        if stable:
            break

    return V, pi
```

Itération sur la valeur

```
def value_iteration(P, R, gamma=0.99, theta=1e-8):
    """Iteration sur la valeur."""
    nS, nA = P.shape[0], P.shape[1]
    V = np.zeros(nS)
```

```
    while True:
        delta = 0.0
        for s in range(nS):
            v = V[s]
            Q_s = np.array([
                np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
                for a in range(nA)
            ])
            V[s] = np.max(Q_s)
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break

    # Extraire la politique
    pi = np.zeros(nS, dtype=int)
    for s in range(nS):
        Q_s = np.array([
            np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
            for a in range(nA)
        ])
        pi[s] = np.argmax(Q_s)

    return V, pi
```

Application au FrozenLake

```
import gymnasium as gym

# Construire le modele de transition depuis FrozenLake
env = gym.make("FrozenLake-v1", is_slippery=True)
nS = env.observation_space.n  # 16
nA = env.action_space.n       # 4
P_mat = np.zeros((nS, nA, nS))
R_mat = np.zeros((nS, nA, nS))

for s in range(nS):
    for a in range(nA):
        for prob, next_s, reward, done in env.unwrapped.P[s][a]:
            P_mat[s, a, next_s] += prob
            R_mat[s, a, next_s] = reward

V_star, pi_star = value_iteration(P_mat, R_mat, gamma=0.99)
print("Politique optimale (grille 4x4):")
actions = ['<', 'v', '>', '^']
for i in range(4):
    print([actions[pi_star[4*i + j]] for j in range(4)])
```

2.9 Exercices

Exercice 2.1 (Convergence de l'itération sur la valeur)**.** Montrez que pour le gridworld 4×4 du Chapitre 1 avec $\gamma = 0.9$ et $V_0 = 0$, l'itération sur la valeur atteint une précision de $\epsilon = 10^{-6}$ en au plus k itérations. Calculez k.

Exercice 2.2 (Itération sur la politique modifiée)**.** Implémentez une version de l'itération sur la politique où l'évaluation effectue exactement m balayages au lieu de converger. Étudiez l'effet de m sur la vitesse de convergence globale pour $m \in \{1, 3, 10, 100\}$.

Exercice 2.3 (Balayage asynchrone)**.** Implémentez une version asynchrone de l'itération sur la valeur où les états sont mis à jour dans un ordre aléatoire (un seul état par itération). Comparez la convergence avec la version synchrone standard.

Exercice 2.4 (Résolution matricielle)**.** Pour un MDP à 3 états avec la politique uniforme, calculez V^π par inversion matricielle $(I - \gamma P^\pi)^{-1} r^\pi$ et vérifiez que le résultat coïncide avec l'évaluation itérative.

Chapitre 3

Méthodes Monte Carlo et Différences Temporelles

La programmation dynamique suppose une connaissance parfaite de l'environnement : les probabilités de transition, les récompenses, tout est donné. Mais dans le monde réel, un robot qui apprend à marcher ne connaît pas les équations de la physique — il n'a que son *expérience*, les trajectoires qu'il a effectivement parcourues. Comment apprendre à partir de cette seule expérience ?

Deux réponses fondamentales émergent. Les méthodes *Monte Carlo*, héritières des travaux de Stanislaw Ulam et John von Neumann au laboratoire de Los Alamos dans les années 1940, estiment les valeurs en moyennant les retours observés sur des épisodes complets. Les méthodes de *différences temporelles* (TD), introduites par Richard Sutton en 1988, font quelque chose de plus audacieux : elles mettent à jour les estimations à chaque pas de temps, en utilisant leurs propres prédictions comme cibles — un mécanisme appelé *bootstrap*. Cette idée, intellectuellement vertigineuse (on apprend à partir de ce qu'on ne connaît pas encore), est la clé de voûte de l'apprentissage par renforcement moderne.

Intuition

Les méthodes Monte Carlo et les différences temporelles (TD) permettent d'apprendre les fonctions de valeur directement à partir de l'expérience, sans connaître le modèle de transition. Monte Carlo attend la fin d'un épisode pour mettre à jour ; TD met à jour à chaque pas de temps en utilisant une estimation bootstrap.

3.1 Apprentissage sans modèle

Contrairement à la programmation dynamique qui requiert la connaissance complète du modèle (P, R), les méthodes sans modèle (*model-free*) apprennent à partir d'épisodes générés par interaction directe avec l'environnement. Un épisode est une séquence :

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$$

où S_T est un état terminal.

3.2 Méthodes Monte Carlo

3.2.1 Estimation MC de la valeur

Définition 3.1 (Évaluation MC première visite)**.** L'estimation MC **première visite** de $V^\pi(s)$ est la moyenne des retours G_t observés à la première occurrence de s dans chaque épisode :

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

où $N(s)$ est le nombre d'épisodes où s a été visité.

Définition 3.2 (Évaluation MC toutes visites)**.** L'estimation MC **toutes visites** utilise toutes les occurrences de s dans tous les épisodes, pas seulement la première.

Théorème 3.3 (Convergence MC première visite)**.** *L'estimation MC première visite converge vers $V^\pi(s)$ lorsque $N(s) \to \infty$, par la loi forte des grands nombres, car les retours $G_t^{(i)}$ sont des variables aléatoires i.i.d. d'espérance $V^\pi(s)$.*

Évaluation MC première visite

1. Initialiser $V(s) \leftarrow 0$, $N(s) \leftarrow 0$ pour tout s
2. Pour chaque épisode :
   (a) Générer l'épisode en suivant π : $S_0, A_0, R_1, \ldots, S_T$
   (b) $G \leftarrow 0$
   (c) Pour $t = T-1, T-2, \ldots, 0$:
      i. $G \leftarrow \gamma G + R_{t+1}$
      ii. Si S_t n'apparaît pas dans $S_0, \ldots, S_{t-1}$:
         A. $N(S_t) \leftarrow N(S_t) + 1$
         B. $V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G - V(S_t))$

3.2.2 Estimation MC de la fonction Q

Pour l'amélioration de politique sans modèle, il est nécessaire d'estimer $Q^\pi(s,a)$ plutôt que $V^\pi(s)$:

$$Q^\pi(s,a) \approx \frac{1}{N(s,a)} \sum_{i=1}^{N(s,a)} G_t^{(i)}.$$

Problème d'exploration

Si π est déterministe, certaines paires (s,a) ne seront jamais visitées. Solution : utiliser des **départs exploratoires** (*exploring starts*) ou une politique ϵ-gloutonne.

3.2.3 MC avec politique ϵ-gloutonne

Définition 3.4 (Politique ϵ-gloutonne)**.** La politique ϵ-gloutonne dérivée de Q est :

$$\pi_\epsilon(a \mid s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{si } a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{sinon} \end{cases}$$

Théorème 3.5 (Amélioration ϵ-gloutonne)**.** *La politique ϵ-gloutonne π' par rapport à Q^{π_ϵ} satisfait $V^{\pi'}(s) \geq V^{\pi_\epsilon}(s)$ pour tout s.*

3.3 MC hors-politique avec échantillonnage d'importance

Définition 3.6 (Rapport d'importance)**.** Soit b la politique de comportement et π la politique cible. Le **rapport d'importance** pour une trajectoire de t à $T-1$ est :

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}.$$

Estimateur MC hors-politique

L'estimation hors-politique de V^π est :

$$V^\pi(s) \approx \frac{\sum_i \rho_{t_i:T_i-1} G_{t_i}^{(i)}}{\sum_i \rho_{t_i:T_i-1}} \quad \text{(échantillonnage d'importance pondéré).}$$

Proposition 3.7 (Biais et variance)**.** L'estimateur ordinaire $\hat{V} = \frac{1}{n}\sum_i \rho_i G_i$ est non biaisé mais a une variance potentiellement infinie. L'estimateur pondéré $\hat{V}_w = \frac{\sum_i \rho_i G_i}{\sum_i \rho_i}$ est biaisé mais à variance bornée. Le biais tend vers zéro quand $n \to \infty$.

3.4 Différences temporelles — TD(0)

Définition 3.8 (Mise à jour TD(0))**.** La méthode TD(0) met à jour V à chaque pas de temps en utilisant la **cible TD** :

$$V(S_t) \leftarrow V(S_t) + \alpha\big[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{cible TD}} - V(S_t)\big].$$

Le terme $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ est appelé **erreur TD**.

Erreur de différence temporelle

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

L'erreur TD est un estimateur non biaisé de l'avantage $A^\pi(S_t, A_t)$ en espérance.

Théorème 3.9 (Convergence de TD(0))**.** *Sous les conditions de Robbins-Monro sur le taux d'apprentissage :*

$$\sum_{t=0}^{\infty} \alpha_t = \infty, \qquad \sum_{t=0}^{\infty} \alpha_t^2 < \infty,$$

l'algorithme TD(0) converge presque sûrement vers V^π*.*

TD(0) pour l'évaluation de politique

1. Initialiser $V(s)$ arbitrairement pour tout s, fixer α
2. Pour chaque épisode :
   (a) Initialiser S_0
   (b) Pour chaque pas $t = 0, 1, 2, \ldots$ jusqu'à terminaison :
      i. $A_t \sim \pi(\cdot|S_t)$
      ii. Observer R_{t+1}, S_{t+1}
      iii. $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$

3.5 Comparaison MC vs TD

	Monte Carlo	**TD(0)**
Nécessite épisodes complets	Oui	Non
Bootstrap	Non	Oui
Biais	Aucun	Biais (diminue)
Variance	Élevée	Faible
Convergence	$1/\sqrt{N}$	Plus rapide en pratique
Sensibilité à α	Faible	Élevée

Remarque 3.10. Le **dilemme biais-variance** est central : MC a un biais nul mais une variance élevée (car le retour G_t intègre toutes les récompenses futures). TD a un biais dû au bootstrap ($V(S_{t+1})$ est une estimation) mais une variance plus faible.

3.6 TD(λ) et traces d'éligibilité

Définition 3.11 (Retour λ)**.** Le **retour** λ est une moyenne pondérée des retours à n pas :

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

où $G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n V(S_{t+n})$ est le retour à n pas.

Définition 3.12 (Trace d'éligibilité)**.** La **trace d'éligibilité** pour l'état s au temps t est :

$$e_t(s) = \begin{cases} \gamma\lambda\, e_{t-1}(s) + 1 & \text{si } S_t = s \\ \gamma\lambda\, e_{t-1}(s) & \text{sinon} \end{cases}$$

avec $e_0(s) = \mathbb{1}_{S_0=s}$.

TD(λ) avec traces d'éligibilité

1. Initialiser $V(s)$ pour tout s
2. Pour chaque épisode :
   (a) $e(s) \leftarrow 0$ pour tout s
   (b) Pour chaque pas t :
      i. Observer R_{t+1}, S_{t+1}
      ii. $\delta_t \leftarrow R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
      iii. $e(S_t) \leftarrow e(S_t) + 1$
      iv. Pour tout s : $V(s) \leftarrow V(s) + \alpha \delta_t e(s)$
      v. Pour tout s : $e(s) \leftarrow \gamma\lambda\, e(s)$

Remarque 3.13. $\lambda = 0$ donne TD(0) ; $\lambda = 1$ donne (essentiellement) MC. Les valeurs intermédiaires $\lambda \in (0, 1)$ offrent un compromis biais-variance optimal en pratique.

3.7 Implémentation en Python

Évaluation Monte Carlo première visite

```python
import numpy as np
import gymnasium as gym

def mc_first_visit(env, pi, n_episodes=10000, gamma=0.99):
    """Evaluation MC premiere visite de V^pi."""
    nS = env.observation_space.n
    V = np.zeros(nS)
    N = np.zeros(nS)

    for _ in range(n_episodes):
        episode = []
        s, _ = env.reset()
        done = False
        while not done:
            a = pi(s)
            s_next, r, terminated, truncated, _ = env.step(a)
            episode.append((s, a, r))
            s = s_next
            done = terminated or truncated

        G = 0.0
        visited = set()
        for s, a, r in reversed(episode):
            G = gamma * G + r
            if s not in visited:
```

```
                visited.add(s)
                N[s] += 1
                V[s] += (G - V[s]) / N[s]
    return V
```

TD(0) pour l'évaluation de politique

```
def td0_evaluation(env, pi, n_episodes=10000, alpha=0.01, gamma=0.99):
    """Evaluation TD(0) de V^pi."""
    nS = env.observation_space.n
    V = np.zeros(nS)

    for _ in range(n_episodes):
        s, _ = env.reset()
        done = False
        while not done:
            a = pi(s)
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            V_next = 0.0 if terminated else V[s_next]
            V[s] += alpha * (r + gamma * V_next - V[s])
            s = s_next
    return V
```

TD(λ) avec traces d'éligibilité

```
def td_lambda(env, pi, n_episodes=10000, alpha=0.01,
              gamma=0.99, lam=0.8):
    """TD(lambda) avec traces d'eligibilite."""
    nS = env.observation_space.n
    V = np.zeros(nS)

    for _ in range(n_episodes):
        e = np.zeros(nS)  # traces d'eligibilite
        s, _ = env.reset()
        done = False
        while not done:
            a = pi(s)
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            V_next = 0.0 if terminated else V[s_next]
            delta = r + gamma * V_next - V[s]
            e[s] += 1.0  # trace accumulee
            V += alpha * delta * e
            e *= gamma * lam
            s = s_next
    return V
```

3.8 Exercices

Exercice 3.1 (Variance MC vs TD)**.** Générez 1000 épisodes dans un MDP à 5 états avec une politique fixe. Estimez V^π par MC première visite et TD(0). Tracez les courbes d'erreur $\|V_n - V^\pi\|_2$ en fonction du nombre d'épisodes n.

Exercice 3.2 (Retour à n pas)**.** Implémentez le retour à n pas $G_t^{(n)}$ et comparez les performances pour $n \in \{1, 2, 4, 8, \infty\}$ sur l'environnement `FrozenLake-v1`.

Exercice 3.3 (Preuve de convergence TD)**.** Démontrez que l'erreur TD δ_t satisfait $\mathbb{E}_\pi[\delta_t | S_t = s] = 0$ lorsque $V = V^\pi$. Qu'en déduit-on sur le point fixe de TD(0) ?

Exercice 3.4 (Échantillonnage d'importance)**.** Implémentez l'estimation MC hors-politique avec échantillonnage d'importance pondéré pour estimer V^π à partir de trajectoires générées par une politique uniforme b. Comparez avec l'estimateur ordinaire.

Chapitre 4

Q-Learning et SARSA

En 1989, Christopher Watkins, dans sa thèse de doctorat à Cambridge, propose un algorithme d'une simplicité trompeuse : à chaque transition, mettre à jour la valeur d'une paire état-action en utilisant le maximum sur les actions suivantes. C'est le *Q-learning*, le premier algorithme d'apprentissage par renforcement dont on a pu prouver la convergence vers la politique optimale *sans* connaître le modèle de l'environnement. Presque simultanément, Rummery et Niranjan proposent *SARSA*, une variante *on-policy*. La distinction entre ces deux approches — *off-policy* et *on-policy* — est l'une des lignes de fracture fondamentales de l'apprentissage par renforcement.

Intuition

Q-Learning et SARSA sont les algorithmes fondamentaux de contrôle sans modèle. Ils apprennent directement la fonction $Q(s, a)$ à partir de l'expérience. SARSA est *on-policy* (en-politique) : il apprend la valeur de la politique qu'il suit. Q-Learning est *off-policy* (hors-politique) : il apprend la valeur de la politique optimale indépendamment de la politique d'exploration.

4.1 Contrôle TD — Cadre général

Le problème de **contrôle** consiste à trouver une politique optimale π^* sans connaître le modèle. On combine l'estimation de Q^π par méthodes TD avec l'amélioration ϵ-gloutonne, dans l'esprit de GPI.

4.2 SARSA — Contrôle en-politique

Définition 4.1 (SARSA)**.** L'algorithme SARSA met à jour $Q(S_t, A_t)$ en utilisant le quintuplet $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big].$$

Mise à jour SARSA

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big]$$

où $A_{t+1} \sim \pi_\epsilon(\cdot|S_{t+1})$ est l'action effectivement choisie par la politique ϵ-gloutonne.

SARSA

1. Initialiser $Q(s,a)$ arbitrairement, $Q(\text{terminal},\cdot)=0$
2. Pour chaque épisode :
   (a) Initialiser S_0 ; choisir A_0 selon π_ϵ dérivée de Q
   (b) Pour chaque pas $t=0,1,\ldots$ jusqu'à terminaison :
      i. Exécuter A_t, observer R_{t+1}, S_{t+1}
      ii. Choisir A_{t+1} selon π_ϵ dérivée de Q
      iii. $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)]$

Théorème 4.2 (Convergence de SARSA)**.** *Sous les conditions de Robbins-Monro et si tous les couples (s,a) sont visités infiniment souvent (condition GLIE : Greedy in the Limit with Infinite Exploration), SARSA converge vers Q^* presque sûrement.*

4.3 Q-Learning — Contrôle hors-politique

Définition 4.3 (Q-Learning)**.** L'algorithme Q-Learning de Watkins (1989) met à jour :

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\big[R_{t+1} + \gamma \max_{a'} Q(S_{t+1},a') - Q(S_t,A_t)\big].$$

Mise à jour Q-Learning

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big[R_{t+1} + \gamma \max_{a'\in\mathcal{A}} Q(S_{t+1},a') - Q(S_t,A_t)\Big]$$

La différence clé avec SARSA est l'utilisation du max au lieu de $Q(S_{t+1},A_{t+1})$. Q-Learning apprend Q^* directement, quelle que soit la politique de comportement (tant que tous les couples (s,a) sont visités).

Q-Learning

1. Initialiser $Q(s,a)$ arbitrairement, $Q(\text{terminal},\cdot)=0$
2. Pour chaque épisode :
   (a) Initialiser S_0
   (b) Pour chaque pas $t=0,1,\ldots$ jusqu'à terminaison :
      i. Choisir A_t selon π_ϵ dérivée de Q
      ii. Exécuter A_t, observer R_{t+1}, S_{t+1}
      iii. $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(S_{t+1},a') - Q(S_t,A_t)]$

Théorème 4.4 (Convergence de Q-Learning)**.** *Si chaque couple (s,a) est visité infiniment souvent et si les taux d'apprentissage $\alpha_t(s,a)$ satisfont les conditions de Robbins-Monro :*

$$\sum_t \alpha_t(s,a) = \infty, \qquad \sum_t \alpha_t^2(s,a) < \infty,$$

alors $Q(s,a) \to Q^*(s,a)$ presque sûrement pour tout (s,a).

Idée de la preuve. La preuve repose sur la théorie des approximations stochastiques. On écrit la mise à jour sous la forme :

$$Q_{t+1}(s,a) = (1-\alpha_t)Q_t(s,a) + \alpha_t[R_{t+1} + \gamma \max_{a'} Q_t(S_{t+1}, a')].$$

C'est un processus d'approximation stochastique de l'opérateur de Bellman $\mathcal{T}^*$. La contraction de $\mathcal{T}^*$ en norme infinie garantit la convergence sous les conditions de Robbins-Monro (théorème de Jaakkola et al., 1994). □

4.4 SARSA vs Q-Learning : l'exemple de la falaise

Exemple 4.5 (Cliff Walking)**.** L'environnement `CliffWalking-v0` illustre la différence :

- **Q-Learning** apprend la politique optimale (le long de la falaise), mais la politique d'exploration peut faire tomber l'agent.
- **SARSA** apprend une politique plus sûre (loin de la falaise) car il tient compte du comportement ϵ-glouton.

Q-Learning optimise le retour de la politique *cible* (gloutonne), tandis que SARSA optimise le retour de la politique *effective* (ϵ-gloutonne).

4.5 Expected SARSA

Définition 4.6 (Expected SARSA)**. Expected SARSA** utilise l'espérance sur les actions futures au lieu d'une action échantillonnée :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t)\Big].$$

Remarque 4.7. Expected SARSA généralise à la fois SARSA et Q-Learning :

- Si π est la politique ϵ-gloutonne courante, c'est une version à variance réduite de SARSA.
- Si π est la politique gloutonne, c'est exactement Q-Learning.

4.6 Double Q-Learning

Biais de maximisation

Q-Learning souffre d'un **biais de maximisation** : $\mathbb{E}[\max_a Q(s,a)] \geq \max_a \mathbb{E}[Q(s,a)]$ (inégalité de Jensen). Ce biais conduit à surestimer systématiquement les valeurs Q.

Définition 4.8 (Double Q-Learning)**. Double Q-Learning** (Hasselt, 2010) maintient deux tables Q^A et Q^B. À chaque pas, avec probabilité 0.5 :

$$Q^A(S_t, A_t) \leftarrow Q^A(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q^B\big(S_{t+1}, \arg\max_{a'} Q^A(S_{t+1}, a')\big) - Q^A(S_t, A_t)\Big]$$

ou on met à jour Q^B symétriquement. L'action est sélectionnée par une table et évaluée par l'autre, éliminant le biais.

4.7 SARSA(λ) et Q(λ)

Définition 4.9 (SARSA(λ))**.** SARSA(λ) combine SARSA avec les traces d'éligibilité :

$$e_t(s,a) = \gamma\lambda\, e_{t-1}(s,a) + \mathbb{1}_{S_t=s, A_t=a}$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\, \delta_t\, e_t(s,a)$$

où $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$.

4.8 Implémentation en Python

Q-Learning tabulaire

```python
import numpy as np
import gymnasium as gym

def q_learning(env, n_episodes=5000, alpha=0.1, gamma=0.99,
               epsilon=0.1):
    """Q-Learning tabulaire avec politique epsilon-gloutonne."""
    nS = env.observation_space.n
    nA = env.action_space.n
    Q = np.zeros((nS, nA))

    def epsilon_greedy(s):
        if np.random.random() < epsilon:
            return env.action_space.sample()
        return np.argmax(Q[s])

    rewards_per_episode = []
    for ep in range(n_episodes):
        s, _ = env.reset()
        total_reward = 0
        done = False
        while not done:
            a = epsilon_greedy(s)
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            # Mise a jour Q-Learning
            target = r + gamma * np.max(Q[s_next]) * (not terminated)
            Q[s, a] += alpha * (target - Q[s, a])
            s = s_next
            total_reward += r
        rewards_per_episode.append(total_reward)

    pi = np.argmax(Q, axis=1)
    return Q, pi, rewards_per_episode
```

SARSA tabulaire

```python
def sarsa(env, n_episodes=5000, alpha=0.1, gamma=0.99,
          epsilon=0.1):
    """SARSA tabulaire avec politique epsilon-gloutonne."""
    nS = env.observation_space.n
    nA = env.action_space.n
    Q = np.zeros((nS, nA))

    def epsilon_greedy(s):
        if np.random.random() < epsilon:
            return env.action_space.sample()
        return np.argmax(Q[s])

    rewards_per_episode = []
    for ep in range(n_episodes):
        s, _ = env.reset()
        a = epsilon_greedy(s)
        total_reward = 0
        done = False
        while not done:
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            a_next = epsilon_greedy(s_next) if not done else 0
            # Mise a jour SARSA
            target = r + gamma * Q[s_next, a_next] * (not terminated)
            Q[s, a] += alpha * (target - Q[s, a])
            s, a = s_next, a_next
            total_reward += r
        rewards_per_episode.append(total_reward)

    pi = np.argmax(Q, axis=1)
    return Q, pi, rewards_per_episode
```

Comparaison sur CliffWalking

```python
import matplotlib.pyplot as plt

env = gym.make("CliffWalking-v0")

Q_ql, pi_ql, rewards_ql = q_learning(env, n_episodes=5000)
Q_sa, pi_sa, rewards_sa = sarsa(env, n_episodes=5000)

# Lissage par moyenne glissante
def smooth(x, window=100):
    return np.convolve(x, np.ones(window)/window, mode='valid')

plt.figure(figsize=(10, 5))
plt.plot(smooth(rewards_ql), label='Q-Learning')
plt.plot(smooth(rewards_sa), label='SARSA')
```

```
plt.xlabel('Episode')
plt.ylabel('Recompense cumulee (lissee)')
plt.legend()
plt.title('CliffWalking: Q-Learning vs SARSA')
plt.show()
```

Double Q-Learning

```
def double_q_learning(env, n_episodes=5000, alpha=0.1,
                      gamma=0.99, epsilon=0.1):
    """Double Q-Learning tabulaire."""
    nS = env.observation_space.n
    nA = env.action_space.n
    QA = np.zeros((nS, nA))
    QB = np.zeros((nS, nA))

    def epsilon_greedy(s):
        if np.random.random() < epsilon:
            return env.action_space.sample()
        return np.argmax(QA[s] + QB[s])

    for ep in range(n_episodes):
        s, _ = env.reset()
        done = False
        while not done:
            a = epsilon_greedy(s)
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            if np.random.random() < 0.5:
                a_star = np.argmax(QA[s_next])
                target = r + gamma * QB[s_next, a_star] * (not
                ↪  terminated)
                QA[s, a] += alpha * (target - QA[s, a])
            else:
                a_star = np.argmax(QB[s_next])
                target = r + gamma * QA[s_next, a_star] * (not
                ↪  terminated)
                QB[s, a] += alpha * (target - QB[s, a])
            s = s_next

    Q = (QA + QB) / 2
    pi = np.argmax(Q, axis=1)
    return Q, pi
```

4.9 Exercices

Exercice 4.1 (Biais de maximisation)**.** Construisez un MDP à 2 états où Q-Learning surestime systématiquement les valeurs. Exécutez 1000 répétitions et comparez les esti-

mations de Q-Learning, Double Q-Learning et la vraie valeur Q^*.

Exercice 4.2 (Taux d'apprentissage)**.** Étudiez l'effet de α sur la convergence de Q-Learning pour $\alpha \in \{0.01, 0.05, 0.1, 0.5, 1.0\}$ sur `Taxi-v3`. Tracez les courbes de convergence.

Exercice 4.3 (Expected SARSA)**.** Implémentez Expected SARSA et montrez qu'il réduit la variance par rapport à SARSA sur `CliffWalking-v0`. Comparez les écarts-types des récompenses cumulatives sur 100 exécutions.

Exercice 4.4 (Preuve formelle)**.** Montrez que Q-Learning est un cas spécial d'Expected SARSA lorsque la politique cible est la politique gloutonne. Écrivez la preuve formelle.

Chapitre 5

Deep Q-Network et Variantes

En février 2015, une équipe de DeepMind publie dans *Nature* un résultat qui fait sensation : un algorithme, le *Deep Q-Network* (DQN), apprend à jouer à 49 jeux Atari directement à partir des pixels de l'écran, atteignant un niveau surhumain dans plusieurs d'entre eux. La clé : combiner le Q-learning tabulé du chapitre précédent avec un réseau de neurones profond qui approxime la fonction Q. Deux innovations techniques — la *mémoire de rejeu* (stocker les transitions et les rééchantillonner aléatoirement) et le *réseau cible* (mettre à jour les cibles plus lentement) — stabilisent un apprentissage qui, sans elles, diverge. DQN a ouvert l'ère du *deep reinforcement learning*.

Intuition

Le Deep Q-Network (DQN) de Mnih et al. (2013, 2015) a révolutionné le RL en combinant Q-Learning avec des réseaux de neurones profonds. Deux innovations clés — la mémoire de rejeu et le réseau cible — stabilisent l'apprentissage, permettant d'apprendre directement à partir de pixels.

5.1 De Q tabulaire à l'approximation de fonction

Le Q-Learning tabulaire ne peut pas gérer les espaces d'états de grande dimension ou continus. L'idée est d'approximer $Q^*(s, a)$ par un réseau de neurones paramétré par θ :

$$Q(s, a; \theta) \approx Q^*(s, a).$$

Instabilité naïve

L'application directe de Q-Learning avec un réseau de neurones diverge à cause de trois problèmes :

1. **Corrélation temporelle** : les échantillons consécutifs $(s_t, a_t, r_{t+1}, s_{t+1})$ sont fortement corrélés.
2. **Cible non stationnaire** : la cible $r + \gamma \max_{a'} Q(s', a'; \theta)$ dépend des paramètres θ qui changent à chaque mise à jour.
3. **Approximation de fonction** : la convergence de Q-Learning n'est plus garantie avec approximation non linéaire.

5.2 Architecture DQN

Définition 5.1 (DQN)**.** Le **Deep Q-Network** utilise deux mécanismes de stabilisation :

1. **Mémoire de rejeu** (*experience replay*) : les transitions (s, a, r, s') sont stockées dans un buffer $\mathcal{D}$ et échantillonnées aléatoirement pour les mises à jour.

2. **Réseau cible** (*target network*) : un réseau $Q(s, a; \theta^-)$ avec des paramètres figés θ^- est utilisé pour calculer la cible. Les paramètres θ^- sont copiés de θ tous les C pas.

Fonction de perte DQN

La perte est l'erreur quadratique moyenne sur un mini-lot échantillonné de $\mathcal{D}$:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right].$$

DQN

1. Initialiser le réseau $Q(\cdot; \theta)$ et le réseau cible $Q(\cdot; \theta^-)$
2. Initialiser la mémoire de rejeu $\mathcal{D}$ (capacité N)
3. Pour chaque épisode :
   (a) Initialiser s_0
   (b) Pour chaque pas t :
      i. Choisir a_t par ϵ-greedy sur $Q(s_t, \cdot; \theta)$
      ii. Exécuter a_t, observer r_{t+1}, s_{t+1}
      iii. Stocker $(s_t, a_t, r_{t+1}, s_{t+1})$ dans $\mathcal{D}$
      iv. Échantillonner un mini-lot de $\mathcal{D}$
      v. Calculer les cibles : $y_i = r_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-)$
      vi. Descente de gradient sur $\frac{1}{|\text{batch}|}\sum_i (y_i - Q(s_i, a_i; \theta))^2$
      vii. Tous les C pas : $\theta^- \leftarrow \theta$

5.3 Améliorations de DQN

5.3.1 Double DQN

Définition 5.2 (Double DQN)**. Double DQN** (van Hasselt et al., 2016) réduit le biais de maximisation en découplant la sélection de l'action et son évaluation :

$$y = r + \gamma\, Q\left(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-\right).$$

L'action est sélectionnée par le réseau courant θ mais évaluée par le réseau cible θ^-.

5.3.2 Dueling DQN

Définition 5.3 (Architecture Dueling)**.** L'architecture **Dueling** (Wang et al., 2016) décompose Q en une fonction de valeur et une fonction avantage :

$$Q(s,a;\theta) = V(s;\theta_V) + A(s,a;\theta_A) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta_A).$$

La soustraction de la moyenne assure l'identifiabilité de la décomposition.

5.3.3 Prioritized Experience Replay

Définition 5.4 (Rejeu prioritisé)**.** Le **rejeu prioritisé** (Schaul et al., 2016) échantillonne les transitions proportionnellement à leur erreur TD :

$$p_i = |\delta_i| + \epsilon_{\text{prio}}, \qquad P(i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha}.$$

Pour corriger le biais, on utilise des poids d'importance : $w_i = (N \cdot P(i))^{-\beta}$ avec $\beta \to 1$ au cours de l'entraînement.

5.3.4 Noisy Networks

Définition 5.5 (NoisyNet)**. NoisyNet** (Fortunato et al., 2018) remplace l'exploration ϵ-greedy par du bruit paramétrique :

$$y = (b + Wx) + (b_{\text{noisy}} \odot \epsilon^b + (W_{\text{noisy}} \odot \epsilon^W)x)$$

où ϵ^b, ϵ^W sont des bruits aléatoires. L'agent apprend quand explorer.

5.3.5 Rainbow

Remarque 5.6. **Rainbow** (Hessel et al., 2018) combine six améliorations : Double DQN, Dueling, Prioritized Replay, NoisyNet, C51 (distributional), et n-step returns. Il constitue l'état de l'art pour le RL discret.

5.4 DQN distributional — C51

Définition 5.7 (RL distributionnel)**.** Au lieu d'estimer $Q(s,a) = \mathbb{E}[G_t]$, le RL distributionnel apprend la **distribution complète** du retour $Z(s,a)$ telle que $Q(s,a) = \mathbb{E}[Z(s,a)]$.

L'algorithme **C51** (Bellemare et al., 2017) représente la distribution par un histogramme discret sur $N_{\text{atoms}} = 51$ atomes $\{z_i\}_{i=0}^{50}$ uniformément espacés dans $[V_{\min}, V_{\max}]$:

$$Z(s,a) = \sum_{i=0}^{N_{\text{atoms}}-1} p_i(s,a)\,\delta_{z_i}.$$

5.5 Implémentation PyTorch

Réseau DQN

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque
import random

class DQN(nn.Module):
    def __init__(self, obs_dim, n_actions, hidden=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, n_actions)
        )

    def forward(self, x):
        return self.net(x)
```

Mémoire de rejeu

```python
class ReplayBuffer:
    def __init__(self, capacity=100000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        return (np.array(states), np.array(actions),
                np.array(rewards, dtype=np.float32),
                np.array(next_states),
                np.array(dones, dtype=np.float32))

    def __len__(self):
        return len(self.buffer)
```

Boucle d'entraînement DQN

```python
import gymnasium as gym

def train_dqn(env_name="CartPole-v1", n_episodes=500,
              gamma=0.99, lr=1e-3, batch_size=64,
```

```
            buffer_size=100000, target_update=10,
            epsilon_start=1.0, epsilon_end=0.01,
            epsilon_decay=0.995):
    env = gym.make(env_name)
    obs_dim = env.observation_space.shape[0]
    n_actions = env.action_space.n

    q_net = DQN(obs_dim, n_actions)
    target_net = DQN(obs_dim, n_actions)
    target_net.load_state_dict(q_net.state_dict())
    optimizer = optim.Adam(q_net.parameters(), lr=lr)
    buffer = ReplayBuffer(buffer_size)
    epsilon = epsilon_start

    for ep in range(n_episodes):
        state, _ = env.reset()
        total_reward = 0
        done = False

        while not done:
            # Epsilon-greedy
            if random.random() < epsilon:
                action = env.action_space.sample()
            else:
                with torch.no_grad():
                    q_vals = q_net(torch.FloatTensor(state))
                    action = q_vals.argmax().item()

            next_state, reward, terminated, truncated, _ =
            ↪  env.step(action)
            done = terminated or truncated
            buffer.push(state, action, reward, next_state, terminated)
            state = next_state
            total_reward += reward

            # Entrainement
            if len(buffer) >= batch_size:
                s, a, r, s2, d = buffer.sample(batch_size)
                s_t = torch.FloatTensor(s)
                a_t = torch.LongTensor(a)
                r_t = torch.FloatTensor(r)
                s2_t = torch.FloatTensor(s2)
                d_t = torch.FloatTensor(d)

                q_values = q_net(s_t).gather(1,
                ↪  a_t.unsqueeze(1)).squeeze()
                with torch.no_grad():
                    next_q = target_net(s2_t).max(1)[0]
                    targets = r_t + gamma * next_q * (1 - d_t)

                loss = nn.MSELoss()(q_values, targets)
```

```
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        epsilon = max(epsilon_end, epsilon * epsilon_decay)
        if ep % target_update == 0:
            target_net.load_state_dict(q_net.state_dict())

        if ep % 50 == 0:
            print(f"Episode {ep}, Reward: {total_reward:.0f}, "
                  f"Epsilon: {epsilon:.3f}")
    return q_net

q_net = train_dqn()
```

Architecture Dueling DQN

```
class DuelingDQN(nn.Module):
    def __init__(self, obs_dim, n_actions, hidden=128):
        super().__init__()
        self.feature = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.ReLU()
        )
        self.value_stream = nn.Sequential(
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, 1)
        )
        self.advantage_stream = nn.Sequential(
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, n_actions)
        )

    def forward(self, x):
        feat = self.feature(x)
        value = self.value_stream(feat)
        advantage = self.advantage_stream(feat)
        # Soustraction de la moyenne pour identifiabilite
        q = value + advantage - advantage.mean(dim=1, keepdim=True)
        return q
```

5.6 Analyse théorique

Théorème 5.8 (Erreur d'approximation de DQN). *Soit $\hat{Q}$ la solution de DQN et Q^* la vraie fonction de valeur optimale. L'erreur de performance de la politique gloutonne $\hat{\pi}(s) = \arg\max_a \hat{Q}(s,a)$ est bornée par :*

$$\left\|V^* - V^{\hat{\pi}}\right\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \left\|Q^* - \hat{Q}\right\|_\infty .$$

Remarque 5.9. Cette borne montre que l'erreur d'approximation est amplifiée par un facteur $O(1/(1-\gamma)^2)$, ce qui explique pourquoi DQN peut être instable avec des valeurs de γ proches de 1.

5.7 Exercices

Exercice 5.1 (Double DQN)**.** Modifiez le code de DQN pour implémenter Double DQN. Comparez les performances sur `CartPole-v1` et `LunarLander-v3`. Mesurez le biais de maximisation en traçant $\max_a Q(s_0, a)$ au cours de l'entraînement.

Exercice 5.2 (Rejeu prioritisé)**.** Implémentez une mémoire de rejeu prioritisée utilisant un arbre de somme (sum-tree). Comparez la vitesse de convergence avec le rejeu uniforme sur `CartPole-v1`.

Exercice 5.3 (DQN sur Atari)**.** Implémentez un DQN convolutif pour jouer à `Breakout` (Atari). Utilisez un empilement de 4 frames, un prétraitement en niveaux de gris 84×84 et l'architecture convolutive originale de Mnih et al.

Exercice 5.4 (Analyse de la mémoire)**.** Étudiez l'effet de la taille du replay buffer $N \in \{1000, 10000, 100000, 1000000\}$ sur les performances de DQN. Expliquez le compromis mémoire/performance.

Chapitre 6

Gradients de Politique — REINFORCE

Jusqu'ici, nous avons appris des *fonctions de valeur* et déduit la politique indirectement. Les méthodes de gradient de politique inversent cette logique : elles paramétrisent directement la politique π_θ et optimisent ses paramètres par montée de gradient. L'algorithme REINFORCE, proposé par Ronald Williams en 1992, utilise le *théorème du gradient de politique* — un résultat élégant qui exprime le gradient de la performance espérée en termes d'espérance sous la politique, sans nécessiter de différencier à travers la dynamique de l'environnement. Cette approche ouvre la voie aux espaces d'actions continus et aux politiques stochastiques expressives.

Intuition

Les méthodes de gradient de politique optimisent directement les paramètres θ d'une politique π_θ en montant le gradient de la performance $J(\theta) = \mathbb{E}_{\pi_\theta}[G_0]$. Contrairement aux méthodes fondées sur la valeur, elles peuvent naturellement gérer les espaces d'action continus et les politiques stochastiques.

6.1 Motivation

Les méthodes de type DQN présentent des limitations :

- Elles ne s'appliquent qu'aux espaces d'action **discrets** et de petite taille (le max sur les actions est explicite).
- La politique est implicite (gloutonne par rapport à Q) et ne peut représenter que des politiques déterministes.
- De petits changements dans Q peuvent causer de grands changements dans la politique, rendant l'apprentissage instable.

L'idée du gradient de politique est de paramétrer directement π_θ et d'optimiser θ par montée de gradient.

6.2 Objectif et théorème du gradient de politique

Définition 6.1 (Objectif de performance)**.** L'objectif de performance est le retour espéré sous la politique π_θ :

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t R_{t+1}\right] = \mathbb{E}_{s_0 \sim \rho_0}\left[V^{\pi_\theta}(s_0)\right]$$

où ρ_0 est la distribution initiale des états et $\tau = (s_0, a_0, r_1, s_1, \ldots)$ est une trajectoire.

Théorème 6.2 (Théorème du gradient de politique)**.** *Le gradient de $J(\theta)$ est :*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t \mid S_t)\, Q^{\pi_\theta}(S_t, A_t)\right].$$

Démonstration. On commence par le cas à un pas. Soit $J(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$ où $d^{\pi_\theta}(s)$ est la distribution stationnaire. Alors :

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_s d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \\ &\quad + \sum_s \nabla_\theta d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) Q^{\pi_\theta}(s,a).\end{aligned}$$

Le résultat remarquable (Sutton et al., 2000) est que l'on peut ignorer le second terme (dépendance de d^{π_θ} en θ). En utilisant l'identité du log-gradient $\nabla_\theta \pi_\theta = \pi_\theta \nabla_\theta \log \pi_\theta$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(A_t|S_t)\, Q^{\pi_\theta}(S_t, A_t)\right].$$

Pour l'horizon fini, on somme sur $t = 0, \ldots, T-1$. □

Théorème du gradient de politique

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t|S_t)\, Q^{\pi_\theta}(S_t, A_t)\right]$$

6.3 Algorithme REINFORCE

Définition 6.3 (REINFORCE)**.** L'algorithme **REINFORCE** (Williams, 1992) estime le gradient de politique en remplaçant $Q^{\pi_\theta}(S_t, A_t)$ par le retour Monte Carlo G_t :

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T_i - 1} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})\, G_t^{(i)}.$$

REINFORCE

1. Initialiser les paramètres θ de π_θ
2. Pour chaque épisode :

(a) Générer une trajectoire $\tau = (s_0, a_0, r_1, \ldots, s_T)$ en suivant π_θ
(b) Pour $t = T-1, T-2, \ldots, 0$: calculer $G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}$
(c) $\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t)\, G_t$

6.4 Réduction de variance : la ligne de base

Théorème 6.4 (Ligne de base)**.** *Pour toute fonction $b(s)$ ne dépendant pas de a :*

$$\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\, b(s)\right] = 0.$$

On peut donc soustraire $b(s)$ du retour sans introduire de biais :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\,(G_t - b(s_t))\right].$$

Démonstration.

$$\begin{aligned}
\mathbb{E}_\pi[\nabla_\theta \log \pi_\theta(a|s)\, b(s)] &= \sum_a \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} b(s) \\
&= b(s) \sum_a \nabla_\theta \pi_\theta(a|s) \\
&= b(s) \nabla_\theta \underbrace{\sum_a \pi_\theta(a|s)}_{=1} = 0.
\end{aligned}$$

□

Remarque 6.5. Le choix optimal de la ligne de base pour minimiser la variance est :

$$b^*(s) = \frac{\mathbb{E}[\|\nabla_\theta \log \pi_\theta\|^2\, G_t \mid S_t = s]}{\mathbb{E}[\|\nabla_\theta \log \pi_\theta\|^2 \mid S_t = s]}.$$

En pratique, on utilise $b(s) \approx V^{\pi_\theta}(s)$, ce qui donne le gradient à base d'avantage :

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\, \hat{A}_t\right]$$

où $\hat{A}_t = G_t - V(s_t;\phi)$ est l'estimateur de l'avantage. C'est la base des méthodes acteur-critique (Chapitre 7).

6.5 Politique gaussienne pour actions continues

Définition 6.6 (Politique gaussienne)**.** Pour un espace d'action continu $\mathcal{A} \subseteq \mathbb{R}^d$, on paramètre :

$$\pi_\theta(a|s) = \mathcal{N}\big(a; \mu_\theta(s), \sigma_\theta^2(s) I\big)$$

où $\mu_\theta(s)$ et $\sigma_\theta(s)$ sont des sorties du réseau de neurones. Le log-gradient est :

$$\nabla_\theta \log \pi_\theta(a|s) = \frac{(a - \mu_\theta(s))}{\sigma_\theta^2(s)} \nabla_\theta \mu_\theta(s) + \left(\frac{(a-\mu_\theta(s))^2}{\sigma_\theta^3(s)} - \frac{1}{\sigma_\theta(s)} \right) \nabla_\theta \sigma_\theta(s).$$

6.6 Politique softmax pour actions discrètes

Définition 6.7 (Politique softmax)**.** Pour un espace d'action discret $\mathcal{A} = \{1, \ldots, K\}$:

$$\pi_\theta(a|s) = \frac{\exp(h_\theta(s,a))}{\sum_{a'} \exp(h_\theta(s,a'))}$$

où $h_\theta(s,a)$ sont les logits produits par le réseau de neurones. Le log-gradient est :

$$\nabla_\theta \log \pi_\theta(a|s) = \nabla_\theta h_\theta(s,a) - \sum_{a'} \pi_\theta(a'|s) \nabla_\theta h_\theta(s,a').$$

6.7 Causalité et réduction de variance supplémentaire

Proposition 6.8 (Causalité dans le gradient de politique)**.** Les récompenses passées ne dépendent pas des actions futures. On peut donc remplacer G_t par le **retour futur** :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{k=t}^{T-1} \gamma^{k-t} r_{k+1}\right)\right].$$

Cela réduit la variance sans introduire de biais.

6.8 Implémentation PyTorch

REINFORCE avec ligne de base

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical
import gymnasium as gym
import numpy as np

class PolicyNetwork(nn.Module):
    def __init__(self, obs_dim, n_actions, hidden=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, n_actions)
        )

    def forward(self, x):
        logits = self.net(x)
        return Categorical(logits=logits)

class ValueNetwork(nn.Module):
    def __init__(self, obs_dim, hidden=128):
```

```
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, 1)
        )

    def forward(self, x):
        return self.net(x).squeeze(-1)

def reinforce_baseline(env_name="CartPole-v1", n_episodes=1000,
                       gamma=0.99, lr_pi=1e-3, lr_v=1e-3):
    env = gym.make(env_name)
    obs_dim = env.observation_space.shape[0]
    n_actions = env.action_space.n

    policy = PolicyNetwork(obs_dim, n_actions)
    value_fn = ValueNetwork(obs_dim)
    opt_pi = optim.Adam(policy.parameters(), lr=lr_pi)
    opt_v = optim.Adam(value_fn.parameters(), lr=lr_v)

    for ep in range(n_episodes):
        states, actions, rewards = [], [], []
        s, _ = env.reset()
        done = False

        while not done:
            s_t = torch.FloatTensor(s)
            dist = policy(s_t)
            a = dist.sample()
            s_next, r, terminated, truncated, _ = env.step(a.item())
            states.append(s)
            actions.append(a.item())
            rewards.append(r)
            s = s_next
            done = terminated or truncated

        # Calcul des retours
        T = len(rewards)
        returns = np.zeros(T)
        G = 0
        for t in reversed(range(T)):
            G = rewards[t] + gamma * G
            returns[t] = G

        states_t = torch.FloatTensor(np.array(states))
        actions_t = torch.LongTensor(actions)
        returns_t = torch.FloatTensor(returns)

        # Mise a jour de la ligne de base (value function)
        values = value_fn(states_t)
```

```
        value_loss = nn.MSELoss()(values, returns_t)
        opt_v.zero_grad()
        value_loss.backward()
        opt_v.step()

        # Mise a jour de la politique
        with torch.no_grad():
            advantages = returns_t - value_fn(states_t)

        dist = policy(states_t)
        log_probs = dist.log_prob(actions_t)
        policy_loss = -(log_probs * advantages).mean()
        opt_pi.zero_grad()
        policy_loss.backward()
        opt_pi.step()

        if ep % 100 == 0:
            print(f"Episode {ep}, Return: {returns[0]:.1f}")

    return policy

policy = reinforce_baseline()
```

REINFORCE pour actions continues

```
from torch.distributions import Normal

class GaussianPolicy(nn.Module):
    def __init__(self, obs_dim, act_dim, hidden=64):
        super().__init__()
        self.shared = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.Tanh(),
            nn.Linear(hidden, hidden), nn.Tanh()
        )
        self.mu_head = nn.Linear(hidden, act_dim)
        self.log_std = nn.Parameter(torch.zeros(act_dim))

    def forward(self, x):
        h = self.shared(x)
        mu = self.mu_head(h)
        std = self.log_std.exp()
        return Normal(mu, std)
```

6.9 Analyse de la variance

Théorème 6.9 (Variance de REINFORCE). *La variance de l'estimateur REINFORCE croît avec :*

1. *La longueur de l'horizon T :* $\mathrm{Var}[\hat{g}] = O(T)$.

2. *La dimensionnalité de θ : chaque composante a sa propre variance.*

3. *La stochasticité de la politique et de l'environnement.*

La ligne de base réduit la variance d'un facteur qui dépend de la corrélation entre $\nabla \log \pi$ et $G_t - b(s)$.

Limitations de REINFORCE

REINFORCE souffre de :

- Haute variance $\Rightarrow$ convergence lente.
- Nécessite des épisodes complets (pas de bootstrap).
- Utilisation inefficace des données (on-policy, pas de rejeu).

Ces limitations motivent les méthodes acteur-critique (Chapitre 7).

6.10 Exercices

Exercice 6.1 (Effet de la ligne de base)**.** Implémentez REINFORCE avec et sans ligne de base sur `CartPole-v1`. Tracez la variance du gradient estimé au cours de l'entraînement pour les deux versions. Comparez la vitesse de convergence.

Exercice 6.2 (Actions continues)**.** Appliquez REINFORCE avec politique gaussienne à l'environnement `Pendulum-v1` (action continue). Utilisez un réseau pour μ et un paramètre apprenable pour $\log \sigma$.

Exercice 6.3 (Preuve du théorème)**.** Démontrez le théorème du gradient de politique pour le cas à horizon infini en utilisant la distribution stationnaire d^{π_θ}.

Exercice 6.4 (Ligne de base optimale)**.** Dérivez analytiquement la ligne de base $b^*(s)$ qui minimise la variance de l'estimateur du gradient et montrez que $V^{\pi_\theta}(s)$ en est une bonne approximation.

Chapitre 7

Méthodes Acteur-Critique

Les méthodes de gradient de politique (REINFORCE) souffrent d'une variance élevée ; les méthodes fondées sur la valeur (DQN) peinent dans les espaces d'actions continus. Au début des années 2000, Richard Sutton, David McAllester, Satinder Singh et Yishay Mansour formalisent une idée qui réconcilie ces deux familles : le *théorème du gradient de politique* montre que l'on peut estimer le gradient à l'aide d'une fonction de valeur apprise séparément. Naît alors l'architecture *acteur-critique* : un **acteur** qui propose les actions et un **critique** qui évalue leur qualité. Le critique fournit une ligne de base (baseline) qui réduit drastiquement la variance des estimations de gradient, tandis que l'acteur conserve la capacité de représenter des politiques stochastiques sur des espaces continus. De A2C à PPO en passant par SAC, les méthodes acteur-critique dominent aujourd'hui l'apprentissage par renforcement profond.

Intuition

Les méthodes acteur-critique combinent le meilleur des deux mondes : un **acteur** (la politique π_θ) et un **critique** (la fonction de valeur V_ϕ ou Q_ϕ). Le critique fournit un estimateur à faible variance pour guider la mise à jour de l'acteur, éliminant le besoin d'attendre la fin d'un épisode.

7.1 Cadre acteur-critique

Définition 7.1 (Acteur-Critique)**.** Un algorithme **acteur-critique** maintient deux réseaux :

- **Acteur** : $\pi_\theta(a|s)$, paramétré par θ.
- **Critique** : $V_\phi(s)$ ou $Q_\phi(s,a)$, paramétré par ϕ.

Le critique est mis à jour par méthodes TD et l'acteur par gradient de politique utilisant le critique comme ligne de base.

Mise à jour acteur-critique de base

Critique :

$$\phi \leftarrow \phi - \alpha_\phi \nabla_\phi \left(R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t)\right)^2$$

Acteur :

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(A_t|S_t) \underbrace{(R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t))}_{\hat{A}_t \text{ (avantage estimé)}}$$

7.2 Estimateurs de l'avantage

Définition 7.2 (Generalized Advantage Estimation (GAE))**.** L'estimateur GAE (Schulman et al., 2016) est une moyenne exponentiellement pondérée des estimateurs à n pas :

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

où $\delta_t = R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t)$ est l'erreur TD.

Proposition 7.3 (Cas limites de GAE)**.**

- $\lambda = 0$: $\hat{A}_t = \delta_t$ (erreur TD à un pas, faible variance, biais élevé).
- $\lambda = 1$: $\hat{A}_t = G_t - V_\phi(S_t)$ (retour MC, variance élevée, pas de biais).

Démonstration. Pour $\lambda = 1$:

$$\begin{aligned}
\hat{A}_t^{\text{GAE}(\gamma,1)} &= \sum_{l=0}^{T-t-1} \gamma^l \delta_{t+l} \\
&= \sum_{l=0}^{T-t-1} \gamma^l (R_{t+l+1} + \gamma V(S_{t+l+1}) - V(S_{t+l})) \\
&= -V(S_t) + \sum_{l=0}^{T-t-1} \gamma^l R_{t+l+1} + \gamma^{T-t} V(S_T) \\
&= G_t - V(S_t) \quad (\text{si } V(S_T) = 0).
\end{aligned}$$

□

7.3 A2C — Advantage Actor-Critic

A2C (Advantage Actor-Critic synchrone)

1. Initialiser θ (acteur) et ϕ (critique)
2. Lancer N environnements en parallèle
3. Pour chaque itération :
   (a) Collecter T pas dans chaque environnement en suivant π_θ
   (b) Calculer les avantages $\hat{A}_t$ via GAE
   (c) Calculer les retours cibles $\hat{R}_t = \hat{A}_t + V_\phi(S_t)$

(d) Mettre à jour le critique : $\phi \leftarrow \phi - \alpha_\phi \nabla_\phi \frac{1}{NT} \sum_{i,t} (V_\phi(s_t^i) - \hat{R}_t^i)^2$

(e) Mettre à jour l'acteur : $\theta \leftarrow \theta + \alpha_\theta \frac{1}{NT} \sum_{i,t} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \hat{A}_t^i$

(f) Ajouter un bonus d'entropie : $+\beta H(\pi_\theta(\cdot|s_t))$

7.4 A3C — Asynchronous Advantage Actor-Critic

Définition 7.4 (A3C)**. A3C** (Mnih et al., 2016) lance N *workers* asynchrones, chacun interagissant avec sa propre copie de l'environnement. Chaque worker :

1. Copie les paramètres globaux θ, ϕ.
2. Collecte T pas de transitions.
3. Calcule les gradients locaux.
4. Met à jour les paramètres globaux de manière asynchrone.

L'asynchronisme remplace la mémoire de rejeu en décorrélant les données.

Remarque 7.5. En pratique, A2C (synchrone) est souvent préféré à A3C car il est plus simple à implémenter, offre une utilisation plus efficace du GPU et atteint des performances similaires.

7.5 PPO — Proximal Policy Optimization

Définition 7.6 (Objectif de substitution)**.** L'objectif de substitution (*surrogate objective*) utilisé par PPO est :

$$L^{\text{CPI}}(\theta) = \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] = \mathbb{E}_t \left[r_t(\theta) \hat{A}_t \right]$$

où $r_t(\theta) = \pi_\theta(a_t|s_t) / \pi_{\theta_{\text{old}}}(a_t|s_t)$ est le rapport de probabilité.

Théorème 7.7 (PPO-Clip)**.** *L'objectif PPO-Clip (Schulman et al., 2017) est :*

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) \hat{A}_t,\ \mathit{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \hat{A}_t \right) \right]$$

avec $\epsilon = 0.2$ typiquement. Cette formulation empêche les mises à jour trop importantes sans nécessiter de contrainte KL explicite.

Objectif complet de PPO

$$L(\theta, \phi) = L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\phi) + c_2 H[\pi_\theta]$$

où $L^{\text{VF}} = (V_\phi(s_t) - \hat{R}_t)^2$ est la perte du critique, $H[\pi_\theta]$ est le bonus d'entropie, et c_1, c_2 sont des hyperparamettres.

PPO

1. Pour chaque itération :
   (a) Collecter T pas dans N environnements parallèles avec $\pi_{\theta_{\text{old}}}$
   (b) Calculer $\hat{A}_t$ via GAE(γ, λ)
   (c) Pour K époques sur le mini-lot :
      i. Calculer $r_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$
      ii. $L^{\text{CLIP}} = \mathbb{E}_t[\min(r_t\hat{A}_t, \text{clip}(r_t, 1\pm\epsilon)\hat{A}_t)]$
      iii. Mettre à jour θ et ϕ par gradient sur $L(\theta, \phi)$
   (d) $\theta_{\text{old}} \leftarrow \theta$

7.6 TRPO — Trust Region Policy Optimization

Définition 7.8 (TRPO)**. TRPO** (Schulman et al., 2015) maximise l'objectif de substitution sous une contrainte de région de confiance :

$$\max_\theta \; L^{\text{CPI}}(\theta) \quad \text{s.c.} \quad \mathbb{E}_t\left[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t) \,\|\, \pi_\theta(\cdot|s_t)]\right] \leq \delta.$$

Théorème 7.9 (Garantie d'amélioration monotone)**.** *TRPO garantit une amélioration monotone de la politique :*

$$J(\theta_{new}) \geq L^{CPI}(\theta_{new}) - \frac{2\epsilon\gamma}{(1-\gamma)^2} D_{KL}^{\max}(\theta_{old}, \theta_{new})$$

où $\epsilon = \max_{s,a} |A^{\pi_{old}}(s,a)|$.

7.7 Implémentation PyTorch de PPO

PPO — Réseau acteur-critique

```python
import torch
import torch.nn as nn
from torch.distributions import Categorical

class ActorCritic(nn.Module):
    def __init__(self, obs_dim, n_actions, hidden=64):
        super().__init__()
        self.shared = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.Tanh(),
            nn.Linear(hidden, hidden), nn.Tanh()
        )
        self.actor = nn.Linear(hidden, n_actions)
        self.critic = nn.Linear(hidden, 1)

    def forward(self, x):
        h = self.shared(x)
```

```
        return Categorical(logits=self.actor(h)),
        ↪  self.critic(h).squeeze(-1)

    def get_action(self, obs):
        dist, value = self(obs)
        action = dist.sample()
        return action, dist.log_prob(action), value
```

PPO — Boucle d'entraînement

```
import gymnasium as gym
import numpy as np

def compute_gae(rewards, values, dones, gamma=0.99, lam=0.95):
    """Calcul du GAE (Generalized Advantage Estimation)."""
    T = len(rewards)
    advantages = np.zeros(T)
    gae = 0
    for t in reversed(range(T)):
        next_val = values[t + 1] if t + 1 < len(values) else 0
        delta = rewards[t] + gamma * next_val * (1 - dones[t]) -
        ↪  values[t]
        gae = delta + gamma * lam * (1 - dones[t]) * gae
        advantages[t] = gae
    returns = advantages + values[:T]
    return advantages, returns

def train_ppo(env_name="CartPole-v1", total_steps=200000,
              n_steps=128, n_epochs=4, batch_size=64,
              gamma=0.99, lam=0.95, clip_eps=0.2,
              lr=3e-4, ent_coef=0.01, vf_coef=0.5):
    env = gym.make(env_name)
    obs_dim = env.observation_space.shape[0]
    n_actions = env.action_space.n

    model = ActorCritic(obs_dim, n_actions)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    obs, _ = env.reset()
    step = 0

    while step < total_steps:
        # Collecte
        states, actions, rewards, dones = [], [], [], []
        log_probs_old, values_list = [], []

        for _ in range(n_steps):
            s_t = torch.FloatTensor(obs)
            with torch.no_grad():
                action, log_prob, value = model.get_action(s_t)
```

```
            next_obs, reward, terminated, truncated, _ =
            ↪  env.step(action.item())
            states.append(obs)
            actions.append(action.item())
            rewards.append(reward)
            dones.append(float(terminated))
            log_probs_old.append(log_prob.item())
            values_list.append(value.item())
            obs = next_obs
            step += 1
            if terminated or truncated:
                obs, _ = env.reset()

        # Valeur bootstrap
        with torch.no_grad():
            _, last_val = model(torch.FloatTensor(obs))
        values_list.append(last_val.item())

        advantages, returns = compute_gae(
            rewards, values_list, dones, gamma, lam)
        advantages = (advantages - advantages.mean()) / (advantages.std()
        ↪  + 1e-8)

        # Tenseurs
        s_t = torch.FloatTensor(np.array(states))
        a_t = torch.LongTensor(actions)
        adv_t = torch.FloatTensor(advantages)
        ret_t = torch.FloatTensor(returns)
        old_lp = torch.FloatTensor(log_probs_old)

        # Optimisation sur K epoques
        for _ in range(n_epochs):
            idx = np.random.permutation(n_steps)
            for start in range(0, n_steps, batch_size):
                mb = idx[start:start + batch_size]
                dist, vals = model(s_t[mb])
                new_lp = dist.log_prob(a_t[mb])
                ratio = (new_lp - old_lp[mb]).exp()
                surr1 = ratio * adv_t[mb]
                surr2 = ratio.clamp(1 - clip_eps, 1 + clip_eps) *
                ↪  adv_t[mb]
                policy_loss = -torch.min(surr1, surr2).mean()
                value_loss = (vals - ret_t[mb]).pow(2).mean()
                entropy = dist.entropy().mean()

                loss = policy_loss + vf_coef * value_loss - ent_coef *
                ↪  entropy
                optimizer.zero_grad()
                loss.backward()
                nn.utils.clip_grad_norm_(model.parameters(), 0.5)
                optimizer.step()
```

```
        if step % 10000 < n_steps:
            print(f"Step {step}, Recent rewards: {sum(rewards):.0f}")
    return model
```

7.8 Exercices

Exercice 7.1 (GAE)**.** Implémentez GAE avec différentes valeurs de $\lambda \in \{0, 0.5, 0.9, 0.95, 1.0\}$ dans PPO. Tracez les courbes d'apprentissage sur `CartPole-v1` et `LunarLander-v3`. Discutez du compromis biais-variance.

Exercice 7.2 (PPO vs TRPO)**.** Implémentez TRPO en utilisant l'optimisation sous contrainte KL avec la méthode du gradient conjugué. Comparez avec PPO-Clip en termes de performance et de temps de calcul sur `HalfCheetah-v5`.

Exercice 7.3 (Entropie)**.** Étudiez l'effet du coefficient d'entropie $c_2 \in \{0, 0.001, 0.01, 0.1\}$ sur l'exploration et la performance finale. Montrez que sans régularisation d'entropie, la politique peut converger prématurément.

Exercice 7.4 (A3C asynchrone)**.** Implémentez A3C avec le module `multiprocessing` de Python. Lancez 4 workers et comparez la vitesse d'entraînement (en temps horloge) avec A2C.

Chapitre 8

Algorithmes d'État de l'Art

En 2015, Timothy Lillicrap et ses collègues de DeepMind publient DDPG (Deep Deterministic Policy Gradient), le premier algorithme capable d'apprendre des politiques dans des espaces d'actions continus directement à partir de pixels. L'idée : combiner l'architecture acteur-critique avec les techniques de stabilisation de DQN (replay buffer, réseau cible) et une politique déterministe. Deux ans plus tard, Scott Fujimoto propose TD3, qui corrige les biais de surestimation de DDPG par le "twin" critics et le lissage de la cible. Puis, en 2018, Tuomas Haarnoja introduit SAC (Soft Actor-Critic), qui intègre un terme d'entropie dans l'objectif, encourageant l'exploration tout en maximisant le rendement. Ces trois algorithmes forment aujourd'hui le trio de référence pour le contrôle continu en apprentissage par renforcement profond.

Intuition

Ce chapitre présente les algorithmes modernes pour les espaces d'action continus : DDPG, TD3 et SAC. Ces méthodes acteur-critique hors-politique combinent des idées de DQN (replay buffer, réseau cible) avec l'optimisation directe de la politique, atteignant des performances remarquables en contrôle robotique.

8.1 DDPG — Deep Deterministic Policy Gradient

Définition 8.1 (Théorème du gradient de politique déterministe)**.** Pour une politique déterministe $\mu_\theta : \mathcal{S} \to \mathcal{A}$, le gradient de la performance est (Silver et al., 2014) :

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d^{\mu_\theta}} \left[\nabla_\theta \mu_\theta(s)\, \nabla_a Q^{\mu_\theta}(s,a)\big|_{a=\mu_\theta(s)} \right].$$

Définition 8.2 (DDPG)**. DDPG** (Lillicrap et al., 2016) est un algorithme acteur-critique hors-politique qui applique le gradient de politique déterministe avec :

- Un **acteur** $\mu_\theta(s)$ (politique déterministe).
- Un **critique** $Q_\phi(s,a)$.
- Une **mémoire de rejeu** $\mathcal{D}$.
- Des **réseaux cibles** μ_{θ^-} et Q_{ϕ^-} avec mise à jour douce : $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$.
- Du **bruit d'exploration** : $a = \mu_\theta(s) + \mathcal{N}(0, \sigma^2)$.

Mises à jour DDPG

Critique : minimiser la perte de Bellman :

$$L(\phi) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma Q_{\phi^-}(s', \mu_{\theta^-}(s')) - Q_\phi(s,a)\right)^2\right]$$

Acteur : maximiser Q :

$$\nabla_\theta J = \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_a Q_\phi(s,a)\big|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s)\right]$$

Cibles douces : $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$, $\phi^- \leftarrow \tau\phi + (1-\tau)\phi^-$

DDPG

1. Initialiser μ_θ, Q_ϕ, copier vers μ_{θ^-}, Q_{ϕ^-}
2. Initialiser $\mathcal{D}$
3. Pour chaque épisode :
   (a) Observer s_0
   (b) Pour chaque pas t :
      i. $a_t = \mu_\theta(s_t) + \epsilon_t$, $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$
      ii. Observer r_{t+1}, s_{t+1}
      iii. Stocker $(s_t, a_t, r_{t+1}, s_{t+1})$ dans $\mathcal{D}$
      iv. Échantillonner un mini-lot de $\mathcal{D}$
      v. Mettre à jour ϕ (critique) et θ (acteur)
      vi. Mise à jour douce : $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$

8.2 TD3 — Twin Delayed DDPG

Problèmes de DDPG

DDPG souffre de surestimation du critique (biais de maximisation) et d'instabilité due aux mises à jour couplées acteur-critique.

Définition 8.3 (TD3)**. TD3** (Fujimoto et al., 2018) corrige DDPG avec trois innovations :

1. **Double critique** : deux réseaux Q_{ϕ_1} et Q_{ϕ_2} ; on utilise le minimum pour la cible :
$$y = r + \gamma \min_{i=1,2} Q_{\phi_i^-}(s', \tilde{a}').$$
2. **Bruit régularisé sur la cible** :
$$\tilde{a}' = \mu_{\theta^-}(s') + \text{clip}(\epsilon, -c, c), \quad \epsilon \sim \mathcal{N}(0, \sigma^2).$$
3. **Mise à jour retardée de l'acteur** : l'acteur et les réseaux cibles ne sont mis à jour que tous les d pas du critique.

Cible TD3

$$y = r + \gamma \min_{i=1,2} Q_{\phi_i^-}(s',\ \text{clip}(\mu_{\theta^-}(s') + \text{clip}(\epsilon, -c, c),\ a_{\min}, a_{\max}))$$

8.3 SAC — Soft Actor-Critic

Définition 8.4 (RL à entropie maximale)**.** Le cadre à **entropie maximale** (*maximum entropy RL*) ajoute un bonus d'entropie à l'objectif :

$$J(\pi) = \sum_{t=0}^{\infty} \gamma^t \mathbb{E}_\pi \left[R_{t+1} + \alpha_{\text{temp}} H[\pi(\cdot|S_t)] \right]$$

où α_{temp} est le coefficient de température et $H[\pi(\cdot|s)] = -\sum_a \pi(a|s) \log \pi(a|s)$.

Définition 8.5 (Équation de Bellman douce)**.** L'équation de Bellman **douce** (*soft*) est :

$$Q^*(s,a) = R(s,a) + \gamma \mathbb{E}_{s'} \left[V^*(s') \right]$$

avec $V^*(s) = \mathbb{E}_{a \sim \pi^*} \left[Q^*(s,a) - \alpha_{\text{temp}} \log \pi^*(a|s) \right]$.

Définition 8.6 (SAC)**. SAC** (Haarnoja et al., 2018) est un algorithme acteur-critique hors-politique à entropie maximale :

- **Politique stochastique** : $a \sim \pi_\theta(\cdot|s)$ paramétrée par une distribution gaussienne avec *reparameterization trick* : $a = \tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi)$, $\xi \sim \mathcal{N}(0, I)$.
- **Double critique** : Q_{ϕ_1}, Q_{ϕ_2} (comme TD3).
- **Ajustement automatique de** α : minimisation de : $J(\alpha) = \mathbb{E}_{a \sim \pi} \left[-\alpha \log \pi(a|s) - \alpha \bar{H} \right]$ où $\bar{H}$ est l'entropie cible.

Mises à jour SAC

Critique :

$$y = r + \gamma \left(\min_{i=1,2} Q_{\phi_i^-}(s', \tilde{a}') - \alpha_{\text{temp}} \log \pi_\theta(\tilde{a}'|s') \right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

Acteur :

$$\theta \leftarrow \theta - \alpha_\theta \nabla_\theta \mathbb{E}_{a \sim \pi_\theta} \left[\alpha_{\text{temp}} \log \pi_\theta(a|s) - \min_{i=1,2} Q_{\phi_i}(s,a) \right]$$

8.4 Comparaison des algorithmes

	DDPG	**TD3**	**SAC**
Politique	Déterministe	Déterministe	Stochastique
Nombre de critiques	1	2	2
Bruit d'exploration	Explicite	Explicite + cible	Entropie
Ajustement de température	—	—	Automatique
Robustesse aux hyperp.	Faible	Moyenne	Élevée

8.5 Implémentation PyTorch de SAC

SAC — Réseaux

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.distributions import Normal
import numpy as np

class SoftQNetwork(nn.Module):
    def __init__(self, obs_dim, act_dim, hidden=256):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_dim + act_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, 1)
        )

    def forward(self, state, action):
        return self.net(torch.cat([state, action], dim=-1)).squeeze(-1)

LOG_STD_MIN, LOG_STD_MAX = -20, 2

class GaussianActor(nn.Module):
    def __init__(self, obs_dim, act_dim, hidden=256):
        super().__init__()
        self.shared = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU()
        )
        self.mu = nn.Linear(hidden, act_dim)
        self.log_std = nn.Linear(hidden, act_dim)

    def forward(self, state):
        h = self.shared(state)
        mu = self.mu(h)
        log_std = self.log_std(h).clamp(LOG_STD_MIN, LOG_STD_MAX)
        return mu, log_std

    def sample(self, state):
        mu, log_std = self(state)
        std = log_std.exp()
        dist = Normal(mu, std)
        x = dist.rsample()  # reparameterization trick
        action = torch.tanh(x)
        # Correction log-prob pour tanh
        log_prob = dist.log_prob(x) - torch.log(1 - action.pow(2) + 1e-6)
        log_prob = log_prob.sum(dim=-1)
        return action, log_prob
```

SAC — Entraînement

```python
def train_sac(env_name="Pendulum-v1", total_steps=100000,
              gamma=0.99, tau=0.005, lr=3e-4, batch_size=256,
              buffer_size=1000000, start_steps=1000):
    import gymnasium as gym
    from collections import deque
    import random

    env = gym.make(env_name)
    obs_dim = env.observation_space.shape[0]
    act_dim = env.action_space.shape[0]
    act_scale = torch.FloatTensor(env.action_space.high)

    actor = GaussianActor(obs_dim, act_dim)
    q1 = SoftQNetwork(obs_dim, act_dim)
    q2 = SoftQNetwork(obs_dim, act_dim)
    q1_target = SoftQNetwork(obs_dim, act_dim)
    q2_target = SoftQNetwork(obs_dim, act_dim)
    q1_target.load_state_dict(q1.state_dict())
    q2_target.load_state_dict(q2.state_dict())

    # Temperature automatique
    target_entropy = -act_dim
    log_alpha = torch.zeros(1, requires_grad=True)
    alpha = log_alpha.exp().item()

    opt_actor = torch.optim.Adam(actor.parameters(), lr=lr)
    opt_q = torch.optim.Adam(
        list(q1.parameters()) + list(q2.parameters()), lr=lr)
    opt_alpha = torch.optim.Adam([log_alpha], lr=lr)

    buffer = deque(maxlen=buffer_size)
    obs, _ = env.reset()

    for step in range(total_steps):
        if step < start_steps:
            action = env.action_space.sample()
        else:
            with torch.no_grad():
                a, _ = actor.sample(torch.FloatTensor(obs))
                action = (a * act_scale).numpy()

        next_obs, reward, terminated, truncated, _ = env.step(action)
        buffer.append((obs, action, reward, next_obs, float(terminated)))
        obs = next_obs if not (terminated or truncated) else
        ↪  env.reset()[0]

        if len(buffer) < batch_size:
            continue
```

```
        batch = random.sample(list(buffer), batch_size)
        s, a, r, s2, d = [torch.FloatTensor(np.array(x)) for x in
        ↪  zip(*batch)]

        # Mise a jour du critique
        with torch.no_grad():
            a2, lp2 = actor.sample(s2)
            q_target = torch.min(q1_target(s2, a2), q2_target(s2, a2))
            y = r + gamma * (1 - d) * (q_target - alpha * lp2)

        q1_loss = F.mse_loss(q1(s, a), y)
        q2_loss = F.mse_loss(q2(s, a), y)
        opt_q.zero_grad()
        (q1_loss + q2_loss).backward()
        opt_q.step()

        # Mise a jour de l'acteur
        a_new, lp_new = actor.sample(s)
        q_new = torch.min(q1(s, a_new), q2(s, a_new))
        actor_loss = (alpha * lp_new - q_new).mean()
        opt_actor.zero_grad()
        actor_loss.backward()
        opt_actor.step()

        # Mise a jour de alpha
        alpha_loss = -(log_alpha.exp() * (lp_new.detach() +
        ↪  target_entropy)).mean()
        opt_alpha.zero_grad()
        alpha_loss.backward()
        opt_alpha.step()
        alpha = log_alpha.exp().item()

        # Mise a jour douce des cibles
        for p, pt in zip(q1.parameters(), q1_target.parameters()):
            pt.data.copy_(tau * p.data + (1 - tau) * pt.data)
        for p, pt in zip(q2.parameters(), q2_target.parameters()):
            pt.data.copy_(tau * p.data + (1 - tau) * pt.data)

    return actor
```

8.6 Exercices

Exercice 8.1 (TD3 vs DDPG)**.** Implémentez TD3 et DDPG et comparez-les sur `Pendulum-v1` et `HalfCheetah-v5`. Mesurez l'effet de chacune des trois innovations de TD3 par ablation.

Exercice 8.2 (Ajustement de température)**.** Comparez SAC avec α fixe ($\alpha \in \{0.01, 0.1, 0.2, 1.0\}$) et SAC avec ajustement automatique. Tracez l'évolution de α et de l'entropie au cours de l'entraînement.

Exercice 8.3 (Mise à jour douce vs dure)**.** Comparez la mise à jour douce ($\tau = 0.005$) et

la mise à jour dure (copie tous les C pas) dans DDPG. Étudiez l'effet de τ sur la stabilité.

Exercice 8.4 (Analyse théorique)**.** Démontrez que l'opérateur de Bellman doux (*soft Bellman operator*) $\mathcal{T}^{\pi}_{\text{soft}}$ est une γ-contraction et admet donc un unique point fixe.

Chapitre 9

RL Multi-Agents

Que se passe-t-il quand plusieurs agents apprennent simultanément dans le même environnement ? La question est fondamentale : le monde réel est peuplé d'agents en interaction — joueurs d'échecs, robots dans un entrepôt, véhicules autonomes sur une route, algorithmes de trading sur un marché. Le passage du MDP (un seul agent) au *jeu stochastique* (plusieurs agents) introduit des défis redoutables : l'environnement devient *non-stationnaire* du point de vue de chaque agent (puisque les autres agents apprennent aussi), et l'espace d'action conjoint explose combinatoirement. Les travaux de Littman (1994) sur les jeux de Markov, puis de Lowe et al. (MADDPG, 2017) et de Rashid et al. (QMIX, 2018), ont ouvert la voie à des méthodes qui équilibrent centralisation et décentralisation.

Intuition

L'apprentissage par renforcement multi-agents (MARL) étend le cadre MDP à des systèmes où plusieurs agents interagissent. Les agents peuvent être coopératifs, compétitifs ou mixtes. La non-stationnarité de l'environnement (du point de vue de chaque agent) et la dimensionnalité croissante de l'espace d'action conjoint posent des défis fondamentaux.

9.1 Jeux stochastiques et cadres formels

Définition 9.1 (Jeu de Markov (*Markov Game*))**.** Un **jeu de Markov** à N joueurs est un tuple $(\mathcal{S}, \{\mathcal{A}_i\}_{i=1}^N, P, \{R_i\}_{i=1}^N, \gamma)$ où :

- $\mathcal{S}$ est l'espace d'états global,
- $\mathcal{A}_i$ est l'espace d'actions de l'agent i,
- $P(s'|s, a_1, \ldots, a_N)$ est la transition jointe,
- $R_i(s, a_1, \ldots, a_N)$ est la récompense de l'agent i,
- γ est le facteur d'actualisation commun.

Définition 9.2 (Types de jeux)**.**

- **Coopératif (équipe)** : $R_1 = R_2 = \cdots = R_N$.
- **Compétitif (somme nulle)** : $\sum_{i=1}^N R_i = 0$.

- **Mixte (somme générale)** : pas de contrainte sur les R_i.

Définition 9.3 (Équilibre de Nash)**.** Un profil de politiques $(\pi_1^*, \ldots, \pi_N^*)$ est un **équilibre de Nash** si pour tout agent i et toute politique alternative π_i :

$$V_i^{\pi_1^*,\ldots,\pi_i^*,\ldots,\pi_N^*}(s) \geq V_i^{\pi_1^*,\ldots,\pi_i,\ldots,\pi_N^*}(s) \quad \forall s \in \mathcal{S}.$$

Aucun agent ne peut améliorer unilatéralement sa performance.

9.2 Défis du MARL

Défis fondamentaux

1. **Non-stationnarité** : du point de vue de l'agent i, l'environnement inclut les autres agents qui apprennent simultanément $\Rightarrow$ les garanties de convergence du RL à un agent ne s'appliquent plus.
2. **Explosion combinatoire** : l'espace d'actions conjoint est $\mathcal{A}_1 \times \cdots \times \mathcal{A}_N$, de taille exponentielle en N.
3. **Attribution du crédit** : dans un cadre coopératif, il est difficile d'attribuer la récompense d'équipe aux contributions individuelles.
4. **Observabilité partielle** : chaque agent n'observe souvent qu'une partie de l'état global.

9.3 Paradigmes d'entraînement

9.3.1 Apprentissage indépendant (IQL)

Définition 9.4 (Independent Q-Learning)**.** Chaque agent i exécute son propre algorithme de Q-Learning en traitant les autres agents comme faisant partie de l'environnement :

$$Q_i(s, a_i) \leftarrow Q_i(s, a_i) + \alpha \left[r_i + \gamma \max_{a_i'} Q_i(s', a_i') - Q_i(s, a_i) \right].$$

Simple mais ne converge pas en général à cause de la non-stationnarité.

9.3.2 CTDE — Centralized Training, Decentralized Execution

Définition 9.5 (CTDE)**.** Le paradigme **CTDE** autorise l'accès à l'information globale pendant l'entraînement, mais chaque agent n'utilise que ses observations locales lors de l'exécution :

- **Entraînement centralisé** : les critiques ont accès à l'état global s et/ou aux actions jointes $(a_1, \ldots, a_N)$.
- **Exécution décentralisée** : chaque acteur π_i ne dépend que de l'observation locale o_i.

9.4 MADDPG

Définition 9.6 (MADDPG)**. MADDPG** (Lowe et al., 2017) applique le paradigme CTDE avec :

- Un acteur décentralisé par agent : $\mu_{\theta_i}(o_i)$.
- Un critique centralisé par agent : $Q_{\phi_i}(s, a_1, \ldots, a_N)$.

La mise à jour du critique de l'agent i est :

$$L(\phi_i) = \mathbb{E}\left[\left(r_i + \gamma Q_{\phi_i^-}(s', a'_1, \ldots, a'_N) - Q_{\phi_i}(s, a_1, \ldots, a_N)\right)^2\right]$$

où $a'_j = \mu_{\theta_j^-}(o'_j)$ pour tout j.

9.5 QMIX et décomposition de la valeur

Définition 9.7 (QMIX)**. QMIX** (Rashid et al., 2018) apprend une fonction de valeur jointe Q_{tot} décomposable de manière monotone :

$$Q_{\text{tot}}(s, \mathbf{a}) = f_\theta(Q_1(o_1, a_1), \ldots, Q_N(o_N, a_N), s)$$

où f_θ est un réseau de mélange (*mixing network*) avec des poids positifs (garantissant la monotonie) :

$$\frac{\partial Q_{\text{tot}}}{\partial Q_i} \geq 0 \quad \forall i.$$

Cela permet l'exécution décentralisée : $a_i^* = \arg\max_{a_i} Q_i(o_i, a_i)$.

Théorème 9.8 (Condition de factorisation IGM)**.** *La condition IGM (*Individual-Global-Max*) :*

$$\arg\max_{\mathbf{a}} Q_{tot}(s, \mathbf{a}) = \left(\arg\max_{a_1} Q_1(o_1, a_1), \ldots, \arg\max_{a_N} Q_N(o_N, a_N)\right)$$

est satisfaite si Q_{tot} est monotone en chaque Q_i.

9.6 MAPPO

Définition 9.9 (MAPPO)**. MAPPO** (Yu et al., 2022) applique PPO au cadre multi-agents avec partage de paramètres :

- Acteurs partageant les mêmes paramètres θ (avec un identifiant d'agent en entrée).
- Critique centralisé $V_\phi(s)$ (ou $V_\phi(o_i, s)$).
- GAE calculé indépendamment pour chaque agent.

Malgré sa simplicité, MAPPO atteint des performances compétitives sur de nombreux benchmarks coopératifs.

9.7 Communication apprise

Définition 9.10 (CommNet)**. CommNet** (Sukhbaatar et al., 2016) permet aux agents de communiquer par un canal continu. À chaque couche l :

$$h_i^{l+1} = \sigma\left(W^l h_i^l + C^l \frac{1}{N-1}\sum_{j \neq i} h_j^l\right)$$

où h_i^l est la représentation cachée de l'agent i et C^l est la matrice de communication.

9.8 Self-Play

Définition 9.11 (Auto-jeu (*Self-Play*))**.** Dans les jeux compétitifs, l'**auto-jeu** entraîne un agent contre des copies de lui-même (éventuellement passées). Cela génère un curriculum naturel : à mesure que l'agent s'améliore, ses adversaires deviennent plus forts.

Remarque 9.12. AlphaGo Zero et AlphaZero utilisent l'auto-jeu combiné avec MCTS (*Monte Carlo Tree Search*) et un réseau de neurones profond pour apprendre sans aucune donnée humaine.

9.9 Implémentation — MARL simple

Independent Q-Learning multi-agents

```python
import numpy as np

class IndependentQLearner:
    def __init__(self, n_agents, obs_sizes, n_actions,
                 alpha=0.1, gamma=0.99, epsilon=0.1):
        self.n_agents = n_agents
        self.Q = [np.zeros((obs_sizes[i], n_actions))
                  for i in range(n_agents)]
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon

    def select_actions(self, observations):
        actions = []
        for i, obs in enumerate(observations):
            if np.random.random() < self.epsilon:
                actions.append(np.random.randint(self.Q[i].shape[1]))
            else:
                actions.append(np.argmax(self.Q[i][obs]))
        return actions

    def update(self, obs, actions, rewards, next_obs, dones):
        for i in range(self.n_agents):
            s, a, r, s2, d = obs[i], actions[i], rewards[i], next_obs[i],
            ↪  dones[i]
```

```
            target = r + self.gamma * np.max(self.Q[i][s2]) * (1 - d)
            self.Q[i][s, a] += self.alpha * (target - self.Q[i][s, a])
```

QMIX — Réseau de mélange

```
import torch
import torch.nn as nn

class QMIXMixer(nn.Module):
    """Mixing network de QMIX avec poids positifs."""
    def __init__(self, n_agents, state_dim, embed_dim=32):
        super().__init__()
        self.n_agents = n_agents
        # Hyper-reseaux generant les poids du mixer
        self.hyper_w1 = nn.Sequential(
            nn.Linear(state_dim, embed_dim),
            nn.ReLU(),
            nn.Linear(embed_dim, n_agents * embed_dim)
        )
        self.hyper_w2 = nn.Sequential(
            nn.Linear(state_dim, embed_dim),
            nn.ReLU(),
            nn.Linear(embed_dim, embed_dim)
        )
        self.hyper_b1 = nn.Linear(state_dim, embed_dim)
        self.hyper_b2 = nn.Sequential(
            nn.Linear(state_dim, embed_dim),
            nn.ReLU(),
            nn.Linear(embed_dim, 1)
        )

    def forward(self, agent_qs, state):
        """
        agent_qs: (batch, n_agents)
        state: (batch, state_dim)
        """
        bs = agent_qs.shape[0]
        q = agent_qs.unsqueeze(1)  # (batch, 1, n_agents)

        w1 = torch.abs(self.hyper_w1(state)).view(
            bs, self.n_agents, -1)  # poids positifs
        b1 = self.hyper_b1(state).unsqueeze(1)
        h = torch.relu(torch.bmm(q, w1) + b1)

        w2 = torch.abs(self.hyper_w2(state)).view(bs, -1, 1)
        b2 = self.hyper_b2(state).unsqueeze(1)
        q_tot = (torch.bmm(h, w2) + b2).squeeze(-1).squeeze(-1)
        return q_tot
```

9.10 Exercices

Exercice 9.1 (Non-stationnarité)**.** Implémentez IQL pour un jeu de coordination à 2 joueurs (type "matching pennies"). Montrez que les valeurs Q oscillent et ne convergent pas. Comparez avec un algorithme centralisé.

Exercice 9.2 (MADDPG)**.** Implémentez MADDPG pour l'environnement `simple_spread` de PettingZoo. Comparez avec DDPG indépendant en termes de récompense d'équipe.

Exercice 9.3 (QMIX vs VDN)**.** Implémentez VDN (*Value Decomposition Network*) où $Q_{\text{tot}} = \sum_i Q_i$ et comparez avec QMIX sur un scénario coopératif. QMIX est-il toujours supérieur ? Discutez.

Exercice 9.4 (Self-Play)**.** Implémentez l'auto-jeu avec PPO pour le jeu de Tic-Tac-Toe. L'agent apprend-il la stratégie optimale (jeu parfait) ?

Chapitre 10

RL avec Contraintes et Sécurité

Un robot qui apprend à marcher peut tomber. Un véhicule autonome qui apprend à conduire peut provoquer un accident. Un algorithme de trading qui apprend à investir peut faire faillite. Dans ces applications, maximiser la récompense ne suffit pas : il faut *garantir la sécurité*, y compris pendant la phase d'apprentissage. Le RL contraint (Constrained MDP, Altman 1999) formalise ce problème en ajoutant des contraintes sur les coûts cumulatifs ; le RL sûr (Safe RL) vise des garanties plus fortes, à chaque pas de temps. Les méthodes lagrangiennes, les méthodes à barrières, et les approches fondées sur les fonctions de Lyapunov (Chow et al., 2018) forment l'arsenal de ce domaine en pleine expansion.

Intuition

Dans de nombreuses applications réelles (robotique, véhicules autonomes, santé), maximiser la récompense ne suffit pas : il faut également respecter des contraintes de sécurité. Le RL contraint (*Constrained RL*) formalise ce problème en ajoutant des contraintes sur les coûts cumulatifs. Le RL sûr (*Safe RL*) vise à garantir la sécurité à chaque instant, y compris pendant l'apprentissage.

10.1 MDP contraint (CMDP)

Définition 10.1 (CMDP)**.** Un **MDP contraint** est un MDP augmenté de K fonctions de coût $c_k : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ et de seuils d_k :

$$\max_{\pi} J_R(\pi) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1}\right] \quad \text{s.c.} \quad J_{c_k}(\pi) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t c_k(S_t, A_t)\right] \leq d_k \quad \forall k.$$

Théorème 10.2 (Existence d'une politique optimale pour CMDP)**.** *Pour un CMDP fini avec K contraintes, il existe une politique optimale qui est **mélange** d'au plus $K+1$ politiques déterministes stationnaires. Si l'ensemble réalisable a un intérieur non vide (condition de Slater), le problème peut être résolu par la méthode du Lagrangien.*

10.2 Approche lagrangienne

Définition 10.3 (Lagrangien du CMDP)**.** Le **Lagrangien** du CMDP est :

$$\mathcal{L}(\pi, \boldsymbol{\lambda}) = J_R(\pi) - \sum_{k=1}^{K} \lambda_k \left(J_{c_k}(\pi) - d_k \right)$$

où $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_K) \geq 0$ sont les multiplicateurs de Lagrange.

Théorème 10.4 (Dualité forte)**.** *Sous la condition de Slater (il existe π_0 tel que $J_{c_k}(\pi_0) < d_k$ pour tout k), la dualité forte s'applique :*

$$\max_{\pi} \min_{\boldsymbol{\lambda} \geq 0} \mathcal{L}(\pi, \boldsymbol{\lambda}) = \min_{\boldsymbol{\lambda} \geq 0} \max_{\pi} \mathcal{L}(\pi, \boldsymbol{\lambda}).$$

Le problème dual est convexe en $\boldsymbol{\lambda}$.

PPO-Lagrangien

1. Initialiser θ (politique), ϕ (critique de récompense), ψ (critique de coût), $\lambda \geq 0$
2. Pour chaque itération :
   (a) Collecter des trajectoires avec π_θ
   (b) Estimer les avantages de récompense $\hat{A}_t^R$ et de coût $\hat{A}_t^C$ via GAE
   (c) Mettre à jour θ par PPO-Clip sur l'objectif lagrangien :
   $$L(\theta) = L_R^{\text{CLIP}}(\theta) - \lambda \, L_C^{\text{CLIP}}(\theta)$$
   (d) Mettre à jour λ : $\lambda \leftarrow \max(0, \lambda + \eta_\lambda (J_C(\pi_\theta) - d))$
   (e) Mettre à jour les critiques ϕ et ψ

10.3 CPO — Constrained Policy Optimization

Définition 10.5 (CPO)**. CPO** (Achiam et al., 2017) étend TRPO au cadre contraint en résolvant :

$$\max_{\theta} \ \mathbb{E}_t[r_t(\theta) \hat{A}_t^R] \quad \text{s.c.} \quad \begin{cases} J_{c_k}(\pi_{\theta_{\text{old}}}) + \mathbb{E}_t[r_t(\theta) \hat{A}_t^{c_k}] \leq d_k & \forall k \\ \text{KL}[\pi_{\theta_{\text{old}}} \| \pi_\theta] \leq \delta \end{cases}$$

Ce problème quadratique sous contraintes linéaires et quadratiques se résout analytiquement.

Théorème 10.6 (Garantie de CPO)**.** *Sous certaines conditions (linéarisation valide), CPO garantit la satisfaction des contraintes de coût à chaque mise à jour de politique, pas seulement à convergence.*

10.4 Approches par barrière et pénalité

Définition 10.7 (Méthode de barrière logarithmique)**.** On remplace la contrainte par une pénalité barrière :

$$\max_{\pi} J_R(\pi) - \frac{1}{t} \sum_{k} \log(d_k - J_{c_k}(\pi))$$

où $t > 0$ contrôle la précision de l'approximation. Quand $t \to \infty$, la solution approche celle du problème contraint.

Définition 10.8 (Pénalité quadratique augmentée)**.** La méthode du **Lagrangien augmenté** combine multiplicateurs et pénalité :

$$\mathcal{L}_\rho(\pi, \lambda) = J_R(\pi) - \lambda(J_C(\pi) - d) - \frac{\rho}{2}(J_C(\pi) - d)_+^2$$

où $(x)_+ = \max(0, x)$ et $\rho > 0$ est le paramètre de pénalité.

10.5 Safe RL — Sécurité pendant l'apprentissage

Définition 10.9 (Sécurité au sens de Lyapunov)**.** Un état s est **sûr** s'il existe une politique π qui maintient le système dans un ensemble sûr $\mathcal{S}_{\text{safe}} \subseteq \mathcal{S}$ pour tout temps futur. La **fonction de Lyapunov** $L(s) \geq 0$ satisfait :

$$\mathbb{E}[L(S_{t+1}) \mid S_t = s, A_t = a] - L(s) \leq -\epsilon \quad \forall s \notin \mathcal{S}_{\text{goal}}.$$

Définition 10.10 (Bouclier de sécurité (*Safety Shield*))**.** Un **bouclier de sécurité** est un module qui filtre les actions proposées par l'agent RL :

$$a_{\text{safe}} = \begin{cases} a & \text{si } a \in \mathcal{A}_{\text{safe}}(s) \\ \arg\min_{a' \in \mathcal{A}_{\text{safe}}(s)} \|a' - a\| & \text{sinon} \end{cases}$$

où $\mathcal{A}_{\text{safe}}(s) \subseteq \mathcal{A}$ est l'ensemble des actions sûres dans l'état s.

10.6 Fonctions barrière de contrôle (CBF)

Définition 10.11 (Control Barrier Function)**.** Une **CBF** $h : \mathcal{S} \to \mathbb{R}$ définit l'ensemble sûr $\mathcal{C} = \{s : h(s) \geq 0\}$. Pour un système $s_{t+1} = f(s_t, a_t)$, la condition de sécurité est :

$$h(f(s, a)) - h(s) \geq -\kappa\, h(s), \quad \kappa \in (0, 1).$$

Si cette condition est satisfaite, $h(s_t) \geq 0$ pour tout t si $h(s_0) \geq 0$.

RL avec filtre CBF

1. L'agent RL propose une action a_{rl}
2. Résoudre le QP (programme quadratique) :
$$a^* = \arg\min_{a} \|a - a_{\text{rl}}\|^2 \quad \text{s.c.} \quad h(f(s, a)) - h(s) \geq -\kappa\, h(s)$$
3. Exécuter a^* dans l'environnement

10.7 Implémentation

PPO-Lagrangien simplifié

```python
import torch
import torch.nn as nn
import numpy as np

class LagrangianPPO:
    def __init__(self, actor_critic, cost_critic, cost_limit=25.0,
                 lambda_lr=0.01, clip_eps=0.2, gamma=0.99):
        self.ac = actor_critic
        self.cost_critic = cost_critic
        self.cost_limit = cost_limit
        self.log_lambda = torch.tensor(0.0, requires_grad=True)
        self.lambda_opt = torch.optim.Adam([self.log_lambda],
        ↪  lr=lambda_lr)
        self.clip_eps = clip_eps
        self.gamma = gamma

    @property
    def lam(self):
        return self.log_lambda.exp().item()

    def compute_policy_loss(self, states, actions, old_log_probs,
                            reward_advantages, cost_advantages):
        dist, _ = self.ac(states)
        new_log_probs = dist.log_prob(actions)
        ratio = (new_log_probs - old_log_probs).exp()

        # Objectif PPO-Clip pour la recompense
        surr1 = ratio * reward_advantages
        surr2 = ratio.clamp(1 - self.clip_eps, 1 + self.clip_eps) *
        ↪  reward_advantages
        reward_loss = -torch.min(surr1, surr2).mean()

        # Objectif PPO pour le cout (on veut le minimiser)
        cost_loss = (ratio * cost_advantages).mean()

        # Objectif lagrangien
        total_loss = reward_loss + self.lam * cost_loss
        return total_loss, dist.entropy().mean()

    def update_lambda(self, mean_episode_cost):
        """Mise a jour du multiplicateur de Lagrange."""
        lambda_loss = -self.log_lambda.exp() * (mean_episode_cost -
        ↪  self.cost_limit)
        self.lambda_opt.zero_grad()
        lambda_loss.backward()
        self.lambda_opt.step()
```

Filtre de sécurité simple

```python
import numpy as np
from scipy.optimize import minimize

def safety_filter(action_rl, state, barrier_fn, dynamics_fn,
                  kappa=0.1):
    """
    Filtre l'action pour satisfaire la contrainte CBF.
    barrier_fn: h(s) -> float (positif = sur)
    dynamics_fn: f(s, a) -> s_next
    """
    h_current = barrier_fn(state)

    def constraint(a):
        s_next = dynamics_fn(state, a)
        h_next = barrier_fn(s_next)
        return h_next - (1 - kappa) * h_current

    result = minimize(
        fun=lambda a: np.sum((a - action_rl) ** 2),
        x0=action_rl,
        method='SLSQP',
        constraints={'type': 'ineq', 'fun': constraint},
        bounds=[(-1, 1)] * len(action_rl)
    )
    return result.x if result.success else action_rl
```

10.8 Exercices

Exercice 10.1 (CMDP lagrangien)**.** Formulez le problème de navigation sûre (un robot doit atteindre un objectif en évitant des zones dangereuses) comme un CMDP. Implémentez PPO-Lagrangien et vérifiez que le coût cumulé reste sous le seuil après convergence.

Exercice 10.2 (CPO)**.** Implémentez CPO et comparez-le avec PPO-Lagrangien sur l'environnement `SafetyPointGoal-v0` de Safety Gymnasium. Lequel respecte mieux les contraintes pendant l'entraînement ?

Exercice 10.3 (CBF apprise)**.** Entraînez un réseau de neurones pour apprendre la fonction barrière $h_\psi(s)$ à partir de données étiquetées (états sûrs vs dangereux). Utilisez cette CBF comme filtre de sécurité et évaluez le taux de violations de contraintes.

Exercice 10.4 (Théorie)**.** Démontrez que sous la condition de Slater, la dualité forte s'applique au CMDP et que le problème dual est convexe. Donnez la forme explicite du sous-gradient de la fonction duale.

Chapitre 11

Applications

Intuition

Ce chapitre présente les applications majeures du RL profond dans trois domaines : la robotique, les jeux et la finance. Pour chaque domaine, nous discutons les défis spécifiques, les architectures utilisées et les résultats marquants, avec des exemples concrets d'implémentation.

11.1 Robotique

11.1.1 Défis du RL en robotique

Défis spécifiques à la robotique

1. **Sécurité** : un robot physique peut se détruire ou blesser des personnes pendant l'exploration.
2. **Efficacité d'échantillonnage** : les interactions réelles sont lentes et coûteuses.
3. **Sim-to-real gap** : les politiques apprises en simulation ne se transfèrent pas directement au monde réel.
4. **Espaces continus de haute dimension** : un robot humanoïde a typiquement 20–30 degrés de liberté.
5. **Observations bruyantes et partielles** : capteurs imparfaits, latence.

11.1.2 Sim-to-Real et randomisation de domaine

Définition 11.1 (Randomisation de domaine)**.** La **randomisation de domaine** (*domain randomization*) varie aléatoirement les paramètres de la simulation (masse, friction, délais, textures) pendant l'entraînement, forçant la politique à être robuste aux variations :

$$\max_{\theta} \mathbb{E}_{\xi \sim p(\xi)} \left[J(\theta; \xi) \right]$$

où ξ représente les paramètres de simulation randomisés.

Exemple 11.2 (OpenAI — Rubik's Cube)**.** En 2019, OpenAI a entraîné une main robotique Dexterous à résoudre un Rubik's Cube en simulation avec randomisation massive (ADR — Automatic Domain Randomization). La politique, entraînée avec PPO sur des milliers de GPUs, se transfère directement au robot physique.

11.1.3 Contrôle locomoteur

Exemple 11.3 (Locomotion avec SAC)**.** L'entraînement d'un agent quadrupède dans MuJoCo avec SAC :

- **Observation** : positions et vitesses articulaires, orientation du tronc (dim = 48).
- **Action** : couples articulaires (dim = 12).
- **Récompense** : vitesse vers l'avant $-c_1 \|a\|^2 - c_2 \|\text{contact}\|$.
- Résultat : démarches naturelles émergent après $\sim 5 \times 10^6$ pas.

Contrôle locomoteur avec SAC (Stable-Baselines3)

```python
from stable_baselines3 import SAC
from stable_baselines3.common.vec_env import SubprocVecEnv
import gymnasium as gym

def make_env(env_id):
    def _init():
        return gym.make(env_id)
    return _init

# Environnements paralleles
n_envs = 4
env = SubprocVecEnv([make_env("Ant-v5") for _ in range(n_envs)])

model = SAC(
    "MlpPolicy", env,
    learning_rate=3e-4,
    buffer_size=1_000_000,
    batch_size=256,
    tau=0.005,
    gamma=0.99,
    ent_coef="auto",
    verbose=1
)
model.learn(total_timesteps=3_000_000)
model.save("sac_ant")
```

11.1.4 Manipulation robotique

Définition 11.4 (Hindsight Experience Replay (HER))**. HER** (Andrychowicz et al., 2017) résout le problème des récompenses creuses en manipulation. Après un épisode

échoué, on rejoue les transitions en remplaçant l'objectif original par un objectif atteint :

$$(s_t, a_t, r_t, s_{t+1}, g) \to (s_t, a_t, r'_t, s_{t+1}, g')$$

où $g' = s_T$ (l'état final atteint) et $r'_t = R(s_t, a_t, g')$.

11.2 Jeux

11.2.1 Atari et DQN

Remarque 11.5. DQN (Mnih et al., 2015) a démontré que le RL profond pouvait atteindre des performances surhumaines sur 49 jeux Atari 2600 à partir de pixels bruts ($84 \times 84 \times 4$ frames empilées). L'architecture convolutive :

- Conv 8×8, stride 4, 32 filtres + ReLU
- Conv 4×4, stride 2, 64 filtres + ReLU
- Conv 3×3, stride 1, 64 filtres + ReLU
- FC 512 + ReLU
- FC $|\mathcal{A}|$

11.2.2 Go — AlphaGo et AlphaZero

Définition 11.6 (AlphaZero)**. AlphaZero** (Silver et al., 2018) combine :

- Un réseau $f_\theta(s) = (p, v)$ qui prédit simultanément la politique $p(a|s)$ et la valeur $v(s)$.
- Une recherche arborescente Monte Carlo (MCTS) guidée par le réseau.
- Un entraînement par auto-jeu : la cible pour p est la distribution MCTS $\boldsymbol{\pi}$; la cible pour v est le résultat $z \in \{-1, +1\}$.

Perte :

$$\ell(\theta) = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c\|\theta\|^2.$$

11.2.3 Stratégie en temps réel — StarCraft

Exemple 11.7 (AlphaStar)**. AlphaStar** (Vinyals et al., 2019) a atteint le niveau Grandmaster à StarCraft II en combinant :

- Entraînement supervisé sur des replays humains.
- RL multi-agents dans une ligue de joueurs (population-based training).
- Architecture transformer pour gérer les séquences d'actions longues.
- Espace d'action combinatoire (type d'action + cible + timing).

11.3 Finance

11.3.1 Gestion de portefeuille

Définition 11.8 (MDP de gestion de portefeuille)**.** Le problème de gestion de portefeuille se formalise comme :

- **État** : prix historiques, indicateurs techniques, positions courantes, solde.
- **Action** : vecteur d'allocation $\mathbf{w} \in \Delta^K$ où w_k est la fraction allouée à l'actif k.
- **Récompense** : rendement logarithmique ajusté du risque :

$$R_t = \log\left(\sum_k w_k \frac{p_k^{t+1}}{p_k^t}\right) - c_{\text{transaction}} - \lambda_{\text{risk}} \cdot \text{CVaR}_\alpha$$

Environnement de trading simplifié

```
import numpy as np
import gymnasium as gym
from gymnasium import spaces

class TradingEnv(gym.Env):
    def __init__(self, prices, window=20, initial_balance=10000):
        super().__init__()
        self.prices = prices  # (T, n_assets)
        self.window = window
        self.n_assets = prices.shape[1]
        self.initial_balance = initial_balance
        self.observation_space = spaces.Box(
            low=-np.inf, high=np.inf,
            shape=(window, self.n_assets + 1))  # prix + position
        self.action_space = spaces.Box(
            low=0, high=1, shape=(self.n_assets,))
        self.reset()

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.t = self.window
        self.balance = self.initial_balance
        self.holdings = np.zeros(self.n_assets)
        return self._get_obs(), {}

    def _get_obs(self):
        price_window = self.prices[self.t - self.window:self.t]
        # Rendements normalises
        returns = np.diff(price_window, axis=0) / price_window[:-1]
        position = np.tile(self.holdings / (self.balance + 1e-8),
                           (self.window, 1))
        return np.concatenate([returns, position[:returns.shape[0]]],
        ↪  axis=1)
```

```
    def step(self, action):
        weights = action / (action.sum() + 1e-8)
        # Rendements des actifs
        current_prices = self.prices[self.t]
        next_prices = self.prices[self.t + 1]
        asset_returns = next_prices / current_prices - 1

        portfolio_return = np.dot(weights, asset_returns)
        transaction_cost = 0.001 * np.sum(
            np.abs(weights - self.holdings / (self.balance + 1e-8)))

        self.balance *= (1 + portfolio_return - transaction_cost)
        self.holdings = weights * self.balance
        self.t += 1

        reward = np.log(1 + portfolio_return - transaction_cost)
        terminated = self.t >= len(self.prices) - 1
        return self._get_obs(), reward, terminated, False, {
            "balance": self.balance}
```

11.3.2 Exécution optimale d'ordres

Définition 11.9 (Problème d'exécution optimale)**.** Un trader doit exécuter un ordre de Q parts en T périodes en minimisant l'impact de marché :

- **État** : $s_t = (q_t, t, \text{features de marché})$ où q_t est la quantité restante.
- **Action** : a_t quantité à exécuter au temps t.
- **Contrainte** : $\sum_t a_t = Q$ et $a_t \geq 0$.
- **Récompense** : $-a_t \cdot \text{slippage}(a_t)$.

11.4 Autres applications

11.4.1 RLHF — Alignement de modèles de langage

Définition 11.10 (RLHF)**.** Le **Reinforcement Learning from Human Feedback** (Ouyang et al., 2022) aligne les LLM avec les préférences humaines :

1. Entraîner un modèle de récompense $R_\psi(x, y)$ sur des préférences humaines $(y_w \succ y_l | x)$.
2. Optimiser le LLM π_θ par PPO :

$$\max_\theta \mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta} \left[R_\psi(x, y) - \beta \, \mathrm{KL}[\pi_\theta \| \pi_{\text{ref}}] \right].$$

11.4.2 Découverte de médicaments

Exemple 11.11 (Génération moléculaire)**.** Le RL est utilisé pour générer des molécules optimisant des propriétés pharmacologiques (solubilité, affinité, toxicité). L'agent construit une molécule atome par atome ou fragment par fragment :

- **État** : graphe moléculaire partiel.
- **Action** : ajouter un atome/liaison.
- **Récompense** : score de *docking*, QED, SA score.

11.4.3 Contrôle de plasma de fusion

Exemple 11.12 (DeepMind & EPFL (2022))**.** Le RL a été utilisé pour contrôler la forme du plasma dans le tokamak TCV :

- **État** : mesures magnétiques (200+ capteurs).
- **Action** : tensions des bobines magnétiques (19 actionneurs).
- **Récompense** : écart à la forme cible du plasma.
- L'agent PPO contrôle avec succès des configurations de plasma jamais tentées auparavant.

11.5 Bonnes pratiques

Guide de choix d'algorithme

- **Actions discrètes, observation simple** : DQN, Rainbow, PPO.
- **Actions continues** : SAC, TD3, PPO.
- **Efficacité d'échantillonnage critique** : SAC, TD3 (off-policy).
- **Stabilité et simplicité** : PPO.
- **Multi-agents coopératifs** : MAPPO, QMIX.
- **Contraintes de sécurité** : PPO-Lagrangien, CPO.

Pièges courants

1. Ne pas normaliser les observations et les récompenses.
2. Utiliser un taux d'apprentissage trop élevé.
3. Replay buffer trop petit pour les méthodes off-policy.
4. Ignorer les graines aléatoires : toujours rapporter les résultats sur 5–10 graines avec intervalles de confiance.
5. Ne pas vérifier l'implémentation sur un problème simple d'abord.

11.6 Exercices

Exercice 11.1 (Sim-to-real)**.** Entraînez un agent PPO sur `Pendulum-v1` avec randomisation de la masse et de la friction. Évaluez la robustesse en testant sur des paramètres non vus pendant l'entraînement.

Exercice 11.2 (Trading)**.** Implémentez l'environnement de trading ci-dessus avec des données historiques réelles (par ex. Yahoo Finance). Entraînez SAC et comparez avec une stratégie buy-and-hold et une stratégie de moyenne mobile.

Exercice 11.3 (MCTS + RL)**.** Implémentez une version simplifiée d'AlphaZero pour le jeu de Connect Four. Combinez un réseau politique-valeur avec MCTS et entraînez par auto-jeu.

Exercice 11.4 (HER)**.** Implémentez HER pour un environnement de manipulation `FetchReach-v3`. Comparez les courbes d'apprentissage avec et sans HER.

Bibliographie

[1] Sutton, R.S. and Barto, A.G., *Reinforcement Learning : An Introduction*, 2nd ed., MIT Press, 2018.

[2] Bertsekas, D.P., *Dynamic Programming and Optimal Control*, 4th ed., Athena Scientific, 2017.

[3] Szepesvári, C., *Algorithms for Reinforcement Learning*, Morgan & Claypool, 2010.