SAE

Rapport de la SAE

Réalisé par Evan Nunes, Rostaing Damien, Mattéo T, Mattéo B, Hicham F, Paulo MP

Tuteur / Client : Marin Bougeret & Victor Poupet

Introduction

Dans le cadre de notre formation en BUT Informatique, nous avons choisi de nous consacrer à un projet ambitieux et stimulant : l'apprentissage par renforcement ("Reinforcement Learning"). Ce domaine de l'intelligence artificielle, en pleine expansion, permet à un agent d'apprendre à prendre des décisions optimales en interagissant avec un environnement dynamique. L'agent observe son environnement, agit, reçoit une récompense, et ajuste ses actions en conséquence, créant ainsi une boucle d'apprentissage autonome.

Notre projet, intitulé **« Apprentissage par renforcement, application à l'écriture d'IA jouant à des jeux d'arcade »**, a pour objectif d'explorer les mécanismes fondamentaux du RL. Nous nous sommes fixés comme défi de concevoir et d'entraîner des agents capables de jouer à des jeux variés, allant de jeux simplistes — où le nombre d'états est limité — à des jeux plus complexes, comme des classiques d'arcade tels qu'Arkanoid ou Tetris, voire des jeux combinatoires. Ce travail est encadré par Marin Bougeret et Victor Poupet, et s'inscrit dans une démarche à la fois théorique et pratique.

Objectifs du projet

Ce projet vise plusieurs objectifs majeurs :

- Comprendre et formaliser les algorithmes classiques d'apprentissage par renforcement : Nous avons étudié en profondeur les principes sous-jacents de ces algorithmes, leurs forces, leurs limites, et leurs propriétés mathématiques.
- Découvrir les bonnes pratiques pour entraîner ces algorithmes : Cela inclut la définition des entrées adaptées pour l'agent, la conception de systèmes de récompenses efficaces, et l'optimisation des paramètres pour garantir une convergence rapide et robuste.
- Explorer les arbres de recherche de Monte Carlo (MCTS) : Nous aborderons également les méthodes hybrides, comme le Monte Carlo Tree Search (MCTS), utilisé avec succès dans des applications.

Outils et méthodologie

Pour mener à bien ce projet, nous avons utilisé le langage **Python** et la bibliothèque **Gymnasium**. Cette librairie, spécialement conçue pour faciliter l'implémentation d'algorithmes de RL, nous a fourni les outils nécessaires pour modéliser des

environnements, simuler des interactions, et évaluer les performances de nos agents. Bien que la maîtrise préalable de Python ne fût pas obligatoire, ce projet a été l'occasion d'approfondir nos compétences en programmation et en algorithmique.

Motivations

Ce sujet nous a particulièrement attirés pour son aspect pluridisciplinaire, mêlant algorithmes avancés, mathématiques appliquées, et expérimentation pratique. Il offre une opportunité unique de comprendre comment les machines peuvent apprendre à résoudre des tâches complexes de manière autonome, tout en développant des compétences techniques et analytiques essentielles dans le domaine de l'IA.

Structure du rapport

Ce rapport retrace notre démarche, depuis l'étude théorique des algorithmes de RL jusqu'à leur implémentation concrète. Nous y détaillons :

1. Les concepts clés de l'apprentissage par renforcement.
2. Les choix méthodologiques et techniques que nous avons effectués.
3. Les résultats obtenus, ainsi que les défis rencontrés et les solutions apportées.
4. Les perspectives d'amélioration et les pistes pour de futurs travaux.

Mode dev md-book

Note à savoir

le rapport md-book est disponible en ligne au lien donné

Docker (dev)

Dockerfile

```
FROM debian:trixie

WORKDIR /workspace

RUN apt update && apt install -y curl build-essential


RUN curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
ENV PATH="/root/.cargo/bin:${PATH}"


RUN cargo install mdbook mdbook-katex

EXPOSE 8180

ENTRYPOINT ["/bin/bash", "-c", "mdbook serve -n 0.0.0.0 -p 8180"]
```

Build de la Dockerfile

```
docker build -t mdbook-live .
```

Création du containeur

```
docker run -it --rm -p 8180:8180-v lien/vers/git/sae:/workspace mdbook-live
```

lien si rien changé en local local

SI vous ne pouvez pas utiliser docker ou que vous avez envie de le faire manuellement

Linux:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env
```

la deuxième ligne permet de réactualiser l'env bash/zsh ou on est pour faire en sorte de ne pas avoir a redémarre le terminal

OU

je suppose (j'ai pas essayé mais ca doit marcher)

Debian based:

```
sudo apt install rustc cargo
```

Fedora based:

```
sudo dnf install rust cargo
```

Arch based:

```
sudo pacman -S rust
```

Après installation de rust

on va installer mdbook et un plugin pour prerender du Latex

après en allant dans le dossier de la SAE puis en rentrant dans le dossier mdbook

on fait

```
cargo install mdbook
```

puis

```
cargo install mdbook-katex
```

et enfin pour faire une serveur local pour le dev

```
mdbook serve --open
```

pour build:

```
mdbook build
```

Windows:

installer sur le site de rust le .exe en x64 et installer rust

ensuite installer le .exe mdbook ici installer **mdbook-v0.4.52-x86_64-pc-windows-msvc.zip**

glisser le .exe dans l'archive directement dans le dossier mdbook de la SAE

ensuite on installe le plugin ici il faut développer le v0.9.3-binaries et télécharger le **mdbook-katex-v0.9.3-x86_64-pc-windows-gnu.zip** et extraire le .exe au meme endroit que celui de mdbook

normalement si tout c'est bien fait les commandes sont les même que linux sauf que faut faire

```
.path\to\mdbook.exe ma_commande
```

Processus de Markov

théorique

Explication "global" du MP

Un processus de Markov est un processus stochastique dans lequel la prédiction du futur ne dépend que de l'état présent. Autrement dit, connaître les états passés ne rend pas la prédiction du futur plus précise : seul l'état actuel compte.

MP est défini par une fonction :

$$MP = (S, P)$$

- S : Un ensemble d'état fini
- P : Une matrice de transition des probabilités

P : S^2 -> [0, 1]

∀ s, ∑ P(s,s') = 1

S : Un nombre fini d'états St : Un état défini par un temps t

pour simplifier, on a une fonction qui prend comme arguments **s** et **s'** et donne la probabilité de passer de l'état **s** à l'état **s'** on l'apelle Transition Probability Function.

La matrice de transition :

Si l'ensemble des états $(S = s_1, s_2, ..., s_n)$ étant fini, on peut regrouper toutes les probabilités de transition dans une **matrice de transition** (P) :

$$P = \begin{pmatrix} \mathbb{P}(s_1 \to s_1) & \mathbb{P}(s_1 \to s_2) & \cdots & \mathbb{P}(s_1 \to s_n) \\ \mathbb{P}(s_2 \to s_1) & \mathbb{P}(s_2 \to s_2) & \cdots & \mathbb{P}(s_2 \to s_n) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbb{P}(s_n \to s_1) & \mathbb{P}(s_n \to s_2) & \cdots & \mathbb{P}(s_n \to s_n) \end{pmatrix}$$

Explication "détaillée" du MP

Une chaîne de Markov $X_i, i \geq 0$ sur un $MP(S, P)$ est une suite de variables aléatoires à valeurs dans S telle que pour tout $i \geq 0$:

$$P(X_{i+1} = s' \mid X_i = s) = P(s, s')$$

Remarque : le seul degré de liberté d'une chaîne est donc dans sa loi initiale $\mu_0(i) = P(X_0 = i)$.

1. Propriété de Markov

source: wikipédia (attention hyper complexe)

$\forall$ t $\geq$ 0, $P(S_{t+1} = s'|S_t = s) = P(s, s')$

Avant les explications, il est à noter qu'il y a plusieurs types de propriétés de Markov, chacune dans des ensembles différents ou avec des conditions différentes. Nous allons nous intéresser à la propriété de Markov dite "faible" (temps discret, espace discret) car c'est celle qui est la plus compréhensible et la plus utilisée.

On dit qu'un processus $(S_t)_{t\in T}$ vérifie la propriété de Markov si :

$$\mathbb{P}(S_{t+1} = s' \mid S_t = s, S_{t-1} = s_{t-1}, \ldots) = \mathbb{P}(S_{t+1} = s' \mid S_t = s)$$

En **GROS** le futur ne dépend que du présent, les états dans lesquels nous avons été sont **inutiles** et n'affectent rien.

---

2. Exemple

Par exemple, la météo :

$$S = \{\text{Soleil}, \text{Pluie}\,\text{Orage}\,\text{Nuage}\}$$

Supposons :

- S'il fait soleil aujourd'hui, il y a 50% de chances que demain il fasse encore soleil, 15% qu'il pleuve et 35% qu'il fasse nuageux.
- S'il pleut, il y a 60% de chances que demain il pleuve encore, 25% qu'il y ait des orages et 15% qu'il y ait des nuages.
- S'il il y a orages aujourd'hui, il y a 10% de chances que demain il fasse ensoleillé, 40% qu'il pleuve, 20% qu'il fasse encore orageux et 30% qu'il fasse nuageux.
- S'il fait nuageux aujourd'hui, il y a 10% de chances que demain il fasse ensoleillé, 10% qu'il pleuve et 80% qu'il fasse encore nuageux.

Alors :

$$P = \begin{pmatrix} 0.5 & 0.15 & 0.0 & 0.35 \\ 0.0 & 0.6 & 0.25 & 0.15 \\ 0.1 & 0.4 & 0.2 & 0.3 \\ 0.1 & 0.1 & 0.0 & 0.8 \end{pmatrix}$$

les états passés sont inutiles, seul ceux du présent comptent pour connaître les états futurs

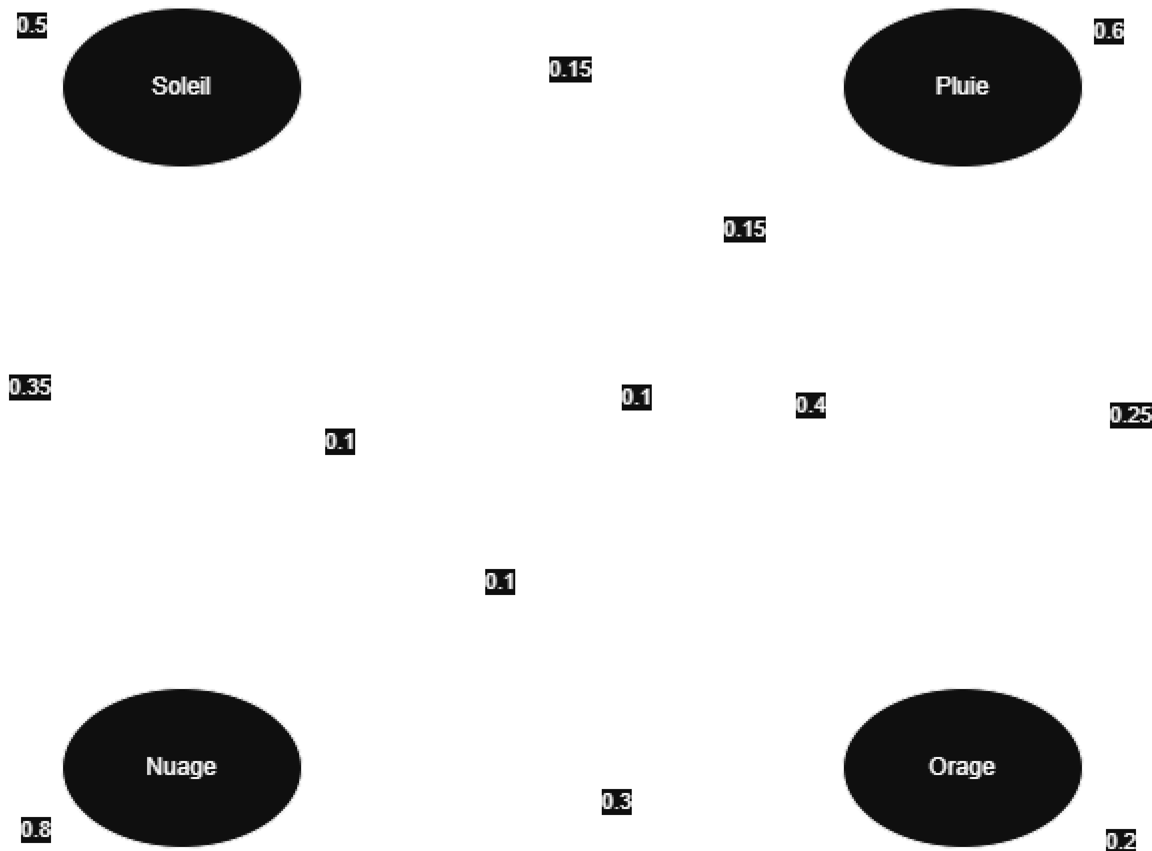
---

3. Représentation par un graphe orienté

On peut représenter la chaîne sous forme de graphe où :

- les nœuds = les états
- les arcs = les transitions
- les poids = probabilités de transition

L'exemple précedent donnerais alors :



Un MP est irréductible si le graphe de P est fortement connexe. Il est apériodique si sa période est égale à 1.

4. État stationnaire

Un processus de Markov converge parfois vers une **distribution stationnaire** π = $\pi(1)$, ..., $\pi(s)$ est telle que :

$$\pi P = \pi \quad \text{et} \quad \sum_i \pi_i = 1$$

Une distribution stationnaire est une distribution de probabilités sur les états qui reste inchangée au fil du temps.

Informellement, au départ l'état dépend de là où on démarre, **mais** à force de répétition les probabilités s'affinent et arrivent à un état de distribution **stable**

Conclusion

Un processus de Markov est donc un modèle où l’évolution ne dépend que de l'état actuel. Les probabilités de transition peuvent être regroupées dans une matrice et permettent d’étudier le comportement du système à long terme.

Markov Reward Process (MRP)

Un **Markov Reward Process (MRP)** est un processus de Markov avec une fonction qui indique combien on gagne (ou perd) à chaque transition.

Définition

Un MRP est défini par le quadruplet :

$$(S, P, R, \gamma)$$

où (S, P) est un processus de markov (MP)

- $R(s, s')$: récompense associée à l'état ou à la transition
- $\gamma \in [0, 1]$: facteur d'actualisation (plus γ est proche de 1, plus les récompenses futures comptent)

Autrement dit :

> Un MRP = Processus de Markov + Récompenses + Facteur d'actualisation.

Récompense

$R(s)$ est le gain obtenu en étant dans l'état s à l'instant t

$G(s)$ est la somme de toutes les récompenses récoltées au cours du temps.

$$G(s) = R(X_0) + \gamma R(X_1) + \gamma^2 R(X_2) + \cdots + \gamma^i R(X_i)$$

$$G(s) = \sum_{i=0}^{\infty} \gamma^i R(X_i)$$

où X_i est une chaîne de markov avec X_0 = s

Remarque : G(s) est une variable aléatoire (correspondant au gain que l'on va observer en partant de s et suivant un chemin en fonction des probabilités indiquées sur les arcs)

Valeur d’un état

Comme $G(s)$ est une variable aléatoire, on ne peut pas se baser sur cette dernière pour définir une stratégie. Dans ce cas, on va définir $V(s)$ qui représente le gain moyen observé en partant de l'état s

$$V(s) = \mathbb{E}[G(s)]$$

La valeur d’un état mesure la récompense totale attendue à long terme, en partant de cet état :

Équation de Bellman

La fonction V vérifie l'équation de Bellman :

$$V(s) = R(s) + \gamma \sum_{s'} P(s, s')V(s')$$

Cette équation exprime que :

- on reçoit immédiatement $R(s)$
- puis on se déplace vers un état futur s'
- avec probabilité $p(s, s')$
- tout en tenant compte de la valeur de cet état futur

Exemple

On a deux états :

$$S = \{\text{Normal}, \text{Bonus}\}$$

Transitions :

- Depuis **Normal** :
  - $P(\text{Normal} \rightarrow \text{Bonus}) = 0.3$
  - $P(\text{Normal} \rightarrow \text{Normal}) = 0.7$
- Depuis **Bonus** :
  - $P(\text{Bonus} \rightarrow \text{Normal}) = 1$

Récompenses :

- $R(\text{Normal}) = 0$
- $R(\text{Bonus}) = +5$

Facteur d'actualisation :

$$\gamma = 0.9$$

Interprétation

- L'état Bonus rapporte +5, mais ne dure pas.
- Depuis Normal, on peut atteindre Bonus, mais ce n'est pas garanti.
- La valeur $V(s)$ permet de savoir si ça "vaut le coup" d'être dans un état.

Calcul de la valeur d'un MRP

Valeurs des variables sur les exemples :

```
P = np.array([[0.7, 0.3],
              [0.4, 0.6]])
R = np.array([5, 10])
gamma = 0.9
```

FORME VECTORIELLE

```
Fonction Bellman_Vector_Forme(P, R, gamma):

    n ← nombre de lignes de P

    I ← matrice identité de taille n × n

    A ← I - gamma × P

    A_inv ← inverse de A

    V ← A_inv × R

    Retourner V
```

Données :

$$P = \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix} \quad R = \begin{pmatrix} 5 \\ 10 \end{pmatrix} \quad \gamma = 0.9$$

Méthode vectorielle :

$$V = (I - \gamma P)^{-1} R$$

I une matrice d'identité donc :

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

donc :

$$[I - \gamma P]^{-1} = \left[\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - 0.9 \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix} \right]^{-1}$$

$$= \left[\begin{pmatrix} 1 - 0.63 & -0.27 \\ -0.36 & 1 - 0.54 \end{pmatrix} \right]^{-1} = \begin{pmatrix} 0.37 & -0.27 \\ -0.36 & 0.46 \end{pmatrix}^{-1}$$

Il faut maintenant calculer l’inverse de cette matrice, pour résoudre cela on applique cette formule :

$$A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

$$\det(I - \gamma P) = 0.37 \times 0.46 - (-0.27) \times (-0.36)$$

$$= 0.1702 - 0.0972 = 0.073$$

$$(I - \gamma P)^{-1} = \frac{1}{0.073} \begin{pmatrix} 0.46 & 0.27 \\ 0.36 & 0.37 \end{pmatrix} = \begin{pmatrix} 6.3013 & 3.6986 \\ 4.9315 & 5.0685 \end{pmatrix}$$

$$V = \begin{pmatrix} 6.3013 & 3.6986 \\ 4.9315 & 5.0685 \end{pmatrix} \cdot R = \begin{pmatrix} 6.3013 & 3.6986 \\ 4.9315 & 5.0685 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 10 \end{pmatrix}$$

$$= \begin{pmatrix} 6.3013 \times 5 + 3.6986 \times 10 \\ 4.9315 \times 5 + 5.0685 \times 10 \end{pmatrix} = \begin{pmatrix} 68.4925 \\ 75.3425 \end{pmatrix}$$

$$V = \begin{pmatrix} 68.4925 \\ 75.3425 \end{pmatrix}$$

FORME RECURSIVE

```
Fonction Bellman_Forme_Récursive(P, R, gamma):

    V ← vecteur nul de même taille que R
    seuil ← 0.0000001
    max_iterations ← 10000

    Pour k allant de 1 à max_iterations faire:

        V_nouveau ← R + gamma × P × V

        SI max(|V_nouveau - V|) < seuil alors:
            Sortir de la boucle
        FIN SI

        V ← V_nouveau
    FIN Pour

    Retourner V
```

Pour (R, P) et (γ), on prendra les mêmes données que la forme vectorielle.

Méthode récursive :

$$V(s) = R(s) + \gamma \sum P(s, s')\, V(s')$$

On commence la 1re itération avec un vecteur nul : $V_0(s) = (0, 0)$

1re itération

$$V_1 = \begin{pmatrix} 5 \\ 10 \end{pmatrix} + 0.9 \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$V_1 = \begin{pmatrix} 5 \\ 10 \end{pmatrix}$$

2e itération

$$V_2 = \begin{pmatrix} 5 \\ 10 \end{pmatrix} + 0.9 \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix} \begin{pmatrix} 5 \\ 10 \end{pmatrix}$$

$$V_2 = \begin{pmatrix} 5 \\ 10 \end{pmatrix} + 0.9 \begin{pmatrix} 6.5 \\ 8 \end{pmatrix} = \begin{pmatrix} 10.85 \\ 17.2 \end{pmatrix}$$

3e itération

$$V_3 = \begin{pmatrix} 5 \\ 10 \end{pmatrix} + 0.9 \begin{pmatrix} 0.7 \times 10.85 + 0.3 \times 17.2 \\ 0.4 \times 10.85 + 0.6 \times 17.2 \end{pmatrix}$$

$$V_3 = \begin{pmatrix} 5 \\ 10 \end{pmatrix} + 0.9 \begin{pmatrix} 12.755 \\ 14.66 \end{pmatrix} = \begin{pmatrix} 16.4795 \\ 23.194 \end{pmatrix}$$

On répète ces itérations jusqu'à atteindre un point de convergence lorsque la différence entre deux itérations devient minime.

Dans notre exemple, la convergence se fera à l'itération 172.

Itération 171 :

$$V_{171} = \begin{pmatrix} 68.49374392 \\ 75.34264868 \end{pmatrix}$$

Itération 172 :

$$V_{172} = \begin{pmatrix} 68.49374972 \\ 75.34265479 \end{pmatrix}$$

Donc :

$$V(s_1) = 68.49374972 \quad \text{et} \quad V(s_2) = 75.34265479$$

Traces écrites :

bellman vector

bellman recursive

Conclusion

Un Markov Reward Process est :

- une chaîne de Markov
- avec une récompense

- et une notion de gain à long terme via la valeur $V(s)$

C'est une étape essentielle entre :

1. Chaînes de Markov (pas de choix)
2. MDP (prise de décision et recherche de meilleure politique)

Markov Decision Process (MDP)

Un **Markov Decision Process (MDP)** est une extension du processus de Markov dans lequel un agent peut choisir une action à chaque étape.
Le passage d'un état à un autre ne dépend donc plus uniquement de l'état actuel, mais aussi de l'**action** effectuée.

L'idée global par rapport a MP :

> À chaque état, l'agent choisit une action.
> Cette action influence la probabilité des états futurs et la récompense qu'il reçoit.

Définition

Un MDP est défini par ces 5 variables :

$$(S, A, P, R, \gamma)$$

- $P(s, a, s') = \mathbb{P}(S_{t+1} = s' \mid S_t = s, A_t = a)$: probabilité de transition
- $R(s, a)$: récompense immédiate reçue en faisant l'action a dans l'état s
- $\gamma \in [0, 1]$: facteur d'actualisation (favorise les récompenses immédiates si γ proche de 0, futures si proche de 1, en gros si on découvre ou si on joue avec nos connaissance)

Fonction de transition

La probabilité de transition dépend maintenant de l'action :

$$P(s, a, s') = \mathbb{P}(S_{t+1} = s' \mid S_t = s, A_t = a)$$

Pas comme le MP de base le MDP dépend de l'état et de la nouvelle variable l'action

---

Récompense

La récompense sert a savoir si une action est correcte ou non.

Elle peut être positive (gain) ou négative (coût) ou neutre si elle n'a aucun importance.

Bien choisir ses récompenses fait change tout, si elles sont mal choisis cela peut mener a un mauvaise apprentissage (pour plus tard)

Politique

Une politique ou policy est une une fonction qui pour un état **s** nous dis quel action **a** effectuer :

$$\pi(s) = a$$

ou (si stochastique) :

$$\pi(a \mid s) = \mathbb{P}(A_t = a \mid S_t = s)$$

MDP + politique

Il est possible d'obtenir un Markov Reward Process (MRP) à partir d'un MDP, en lui ajoutant une politique.

Etant donné un MDP (S, A, P, R, gamma) et une politique Pi , on définit le MRP

$$M^\pi = (S, P^\pi, R^\pi, gamma)$$

avec

$$P^\pi(s'|s) = \sum_{a \in A} \pi(a|s)\, P(s'|s, a)$$

$$R^\pi(s) = \sum_{a \in A} \pi(a|s)\, R(s, a)$$

comme M^π est un MRP, on a donc la définition $V_{M^\pi}(s)$ qui est définie (comme $\mathbb{E}(G(s))$, cf. avant).

Valeur d'une politique

La valeur d'un état sous une politique π est l'espérance des récompenses cumulées :

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \cdot R(S_t, \pi(S_t)) \mid S_0 = s\right]$$

Etant donné un

$$MDP(S, A, P, R, gamma)$$

, on définit

$$V^\pi(s) = V_{M^\pi}(s)$$

Cette formule mesure si dans longtemps nos actions seront bien ou mal, (si le chemin est bon ou mauvais a long termes).

Objectif de l'agent

Trouver la meilleure politique $\pi^\star$ telle que :

$$\pi^\star = \arg\max_\pi V^\pi(s)$$

Autrement dit : choisir les actions permettant d'obtenir le maximum de récompenses cumulées.

Exemple

Le joueur peut être dans deux états :

$$S = \{\text{Début}, \text{Victoire}\}$$

À chaque tours, il a deux action possibles :

$$A = \{\text{Tenter}, \text{Abandonner}\}$$

- Tenter : on lance une pièce.
  - Si c'est face -> on gagne -> on va dans l'état Victoire
  - Si c'est pile -> on reste dans l'état Début
- Abandonner : on tente rien, sans gagner -> on reste dans Début (et on gagne jamais)

Probabilités de transition

- Depuis Début :

- $P(\text{Début}, \text{Tenter}, \text{Victoire}) = 0.5$
- $P(\text{Début}, \text{Tenter}, \text{Début}) = 0.5$
- $P(\text{Début}, \text{Abandonner}, \text{Début}) = 1$

- Depuis Victoire :
  - La partie est terminée (on reste en victoire ou on stop le jeu)

Récompenses

On gagne 1 point seulement quand on atteint Victoire :

$$R(\text{Début}, \text{Tenter}) = \ 1/2$$

Et :

$$R(\text{Début}, \text{Abandonner}) = 0$$

La meilleure politique $\pi^\star$ est donc (je crois) :

$$\pi^\star(\text{Début}) = \text{Tenter}$$

car c'est la seul qui permet d'obtenir une récompense.

Exemple

Données

$$P(s'|s, a_0) = \begin{pmatrix} 0.8 & 0.2 \\ 0.4 & 0.6 \end{pmatrix} \qquad P(s'|s, a_1) = \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix}$$

$$R(s, a_0) = (5,\ 10) \qquad R(s, a_1) = (2,\ 8)$$

$$\pi(a|s_0) = (0.6,\ 0.4) \qquad \pi(a|s_1) = (0.3,\ 0.7)$$

Calcul de P^π

Pour s_0

$$P(0,0) = \pi(a_0|s_0) \times P(s'_0|s_0,a_0) + \pi(a_1|s_0) \times P(s'_0|s_0,a_1)$$
$$= (0.6 \times 0.8) + (0.4 \times 0.9) = 0.84$$
$$P(0,1) = \pi(a_0|s_0) \times P(s'_1|s_0,a_0) + \pi(a_1|s_0) \times P(s'_1|s_0,a_1)$$
$$= (0.6 \times 0.2) + (0.4 \times 0.1) = 0.16$$

Pour s_1

$$P(1,0) = \pi(a_0|s_1) \times P(s'_0|s_1,a_0) + \pi(a_1|s_1) \times P(s'_0|s_1,a_1)$$
$$= (0.3 \times 0.4) + (0.7 \times 0.5) = 0.47$$
$$P(1,1) = \pi(a_0|s_1) \times P(s'_1|s_1,a_0) + \pi(a_1|s_1) \times P(s'_1|s_1,a_1)$$
$$= (0.3 \times 0.6) + (0.7 \times 0.5) = 0.53$$

Résultat final

$$P^\pi = \begin{pmatrix} 0.84 & 0.16 \\ 0.47 & 0.53 \end{pmatrix}$$

Calcul de R^π

$$R^\pi(s_0) = \pi(a_0|s_0) \times R(s_0,a_0) + \pi(a_1|s_0) \times R(s_0,a_1)$$
$$= (0.6 \times 5) + (0.4 \times 10) = 7$$
$$R^\pi(s_1) = \pi(a_0|s_1) \times R(s_1,a_0) + \pi(a_1|s_1) \times R(s_1,a_1)$$
$$= (0.3 \times 2) + (0.7 \times 8) = 6.2$$

Résultat final

$$R^\pi = \begin{pmatrix} 7 \\ 6.2 \end{pmatrix}$$

En code

```
Fonction Construire_MRP_Depuis_MDP(P, R, Politique):

    n_etats ← nombre d'états dans P
    n_actions ← nombre d'actions dans P

    P_MRP ← matrice n_etats × n_etats remplie de 0
    R_MRP ← vecteur de taille n_etats rempli de 0

    Pour chaque état s de 0 à n_etats-1 faire:
        Pour chaque action a de 0 à n_actions-1 faire:

            P_MRP[s] ← P_MRP[s] + Politique[s][a] × P[s][a]

            R_MRP[s] ← R_MRP[s] + Politique[s][a] × R[s][a]
        FIN Pour
    FIN Pour
    Retourner P_MRP, R_MRP
```

Conclusion

Un MDP est donc un modèle qui fait une prise de décision étape par étape.
en gros il ajoute au MP de base :

- Le **choix d'actions**
- La **récompense**
- La notion de **politique optimale**

Traces écrites :

MRP à partir d'un MDP

Algorithmes

Dans les pages qui suivent, nous allons présenter les principaux algorithmes utilisés pour développer la stratégie d'apprentissage de nos IAs.

Policy Improvement

Définition de l'environnement

Valeurs des variables sur l'exemple :

```
states = ["North", "South", "East", "West"]
actions = ["clock", "anti-clock", "stay"]
gamma = 0.9
horizon = 10  # noté H dans les diapositives
```

États et actions possibles

On considère un **cercle** découpé en 4 directions principales :
Nord, Est, Sud, Ouest, avec 3 actions possibles :

- `clock` (sens horaire)
- `anti-clock` (sens antihoraire)
- `stay` (rester sur place)

Chaque action entraîne une transition déterministe vers un nouvel état :

$$P(s,a) \to s'$$

Probabilités de transition

$$P = \begin{cases} P(North, clock) = East \\ P(North, \textit{anti-clock}) = West \\ P(North, stay) = North \\ P(East, clock) = South \\ P(East, \textit{anti-clock}) = North \\ P(East, stay) = East \\ P(South, clock) = West \\ P(South, \textit{anti-clock}) = East \\ P(South, stay) = South \\ P(West, clock) = North \\ P(West, \textit{anti-clock}) = South \\ P(West, stay) = West \end{cases}$$

Récompenses associées (R(s,a))

$$R = \begin{cases} R(North, clock) = 2, \quad R(North, anti\text{-}clock) = 2, \quad R(North, stay) = -2 \\ R(East, clock) = 3, \quad R(East, anti\text{-}clock) = 10, \quad R(East, stay) = 1 \\ R(South, clock) = 1, \quad R(South, anti\text{-}clock) = 3, \quad R(South, stay) = 1 \\ R(West, clock) = 10, \quad R(West, anti\text{-}clock) = -1, \quad R(West, stay) = 1 \end{cases}$$

Étape 1 — Politique initiale

On commence avec une politique arbitraire (non optimale) :

$$\pi_0 = \begin{cases} \pi(North) = stay \\ \pi(South) = clock \\ \pi(East) = clock \\ \pi(West) = anti\text{-}clock \end{cases}$$

Étape 2 — Évaluation de la politique

On estime la **valeur des états** sous la politique courante :

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s))\, V^\pi(s')$$

L'algorithme procède par itérations successives (approche approchée sur un horizon (H = 10)) :

```
V = {s: 0 for s in states}
for _ in range(horizon):
    new_V = V.copy()
    for s in states:
        a = policy[s]
        new_V[s] = R[(s,a)] + gamma * sum(P[(s,a)][s2] * V[s2] for s2 in
P[(s,a)])
    V = new_V
```

À la fin de cette boucle, on obtient une **estimation stable** de (V^π).

Étape 3 — Amélioration de la politique

Pour chaque état (s), on calcule la valeur d'action $(Q(s,a))$:

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

et on met à jour la politique en choisissant l'action optimale :

$$\pi'(s) = \arg\max_a Q(s,a)$$

Code correspondant :

```
for s in states:
    Q_values = {}
    for a in actions:
        Q_values[a] = R[(s,a)] + gamma * sum(P[(s,a)][s2] * V[s2] for s2 in
P[(s,a)])
    policy[s] = max(Q_values, key=Q_values.get)
```

Étape 4 — Vérification de la stabilité

Si la politique ne change plus $(\pi' = \pi)$, on a atteint une **politique optimale** :

$$\pi^* = \pi'$$

Sinon, on retourne à l'étape d'évaluation et on répète jusqu'à convergence.

Résultat final

Après convergence :

$$\pi^* = \begin{cases} \pi^*(North) = clock \\ \pi^*(South) = clock \\ \pi^*(East) = anti\text{-}clock \\ \pi^*(West) = clock \end{cases}$$

et les valeurs d'état associées (V^*) maximisent les retours attendus sous cette politique optimale.

Formule générale de la Policy Iteration

Évaluation :

$$V^\pi(s) = R(s,\pi(s)) + \gamma \sum_{s'} P(s'|s,\pi(s)) V^\pi(s')$$

Amélioration :

$$\pi'(s) = \arg\max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^\pi(s') \right]$$

On alterne ces deux étapes jusqu'à convergence.

Value Improvement

Définition de l'environnement

On considère le même environnement que pour les autres méthodes, afin de permettre une comparaison cohérente :

```
states = ["North", "South", "East", "West"]
actions = ["clock", "anti-clock", "stay"]
gamma = 0.9
horizon = 10
```

Définition de Value Improvement

Value Improvement est un algorithme utilisé pour calculer la meilleure policy dans un MDP. Il sert à déterminer quelle est la meilleure action à faire dans chaque état pour maximiser la récompense cumulée.

Pour cela Value Improvement va appliquer de manière répétée l'équation de Bellman optimale :

$$V_{k+1}(s) = \underset{\alpha}{max} \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s, \alpha) V_k(s') \right]$$

Une fois les valeurs optimales trouvées, on en déduit la politique optimale :

$$\pi_{k+1}(s) = arg \ \underset{\alpha}{max} \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s, \alpha) V^*(s') \right]$$

Étape 1 — Initialisation

On commence avec une valeur d'état nulle pour tous les états :

$$V_0(s) = 0 \quad \forall s$$

```
Pour chaque état s :
    V[s] ← 0
```

Étape 2 — Itération de la valeur

Pour chaque état, on applique Bellman Optimality :

$$V_{k+1}(s) = \underset{\alpha}{max}(R(s,a) + \gamma \cdot V_k(P(s,a)))$$

(Note : transitions déterministes → un seul s' possible)

```
Pour iteration = 1 à H :
    Pour chaque état s :
        valeurs_actions = liste vide

        Pour chaque action a :
            s' = P(s, a)
            q = R(s,a) + gamma * V[s']
            ajouter q à valeurs_actions

        V_temp[s] = max(valeurs_actions)

    V = V_temp
```

À la fin, on obtient une estimation de $V^*(s)$.

Étape 3 — Extraction de la politique optimale

Une fois que les valeurs des états sont stabilisées, on choisit pour chaque état l'action qui maximise la valeur attendue :

$$\pi^*(s) = arg\ \underset{\alpha}{max}\left[R(s,a) + \gamma \cdot V(P(s,a))\right]$$

```
Pour chaque état s :
    meilleur_score = -∞
    meilleure_action = null

    Pour chaque action a :
        s' = P(s, a)
        q = R(s,a) + gamma * V[s']

        Si q > meilleur_score :
            meilleur_score = q
            meilleure_action = a

    policy[s] = meilleure_action
```

Résultat final attendu

Après convergence, Value Improvement retourne :

- La politique optimale $\pi^*(s)$
- Les valeurs optimales d'état $V^*(s)$

Différence avec Policy Evaluation / Policy Improvement

Méthode	Objectif	Dépend d'une politique ?	Type
Policy Evaluation	Évaluer V_π	Oui	Estimation
Policy Improvement	Améliorer π	Oui	Optimisation locale

Policy Evaluation

Qu’est-ce que la policy evaluation ?

La policy evaluation consiste à calculer la valeur d’une politique donnée.

Informellement : si l’agent suit cette politique, quelle récompense moyenne peut-il espérer ?

On cherche donc à calculer la fonction de valeur :

- π : la policy/politique
- $V\pi(s)$: la valeur d’un état s en suivant la politique π.
- Optionnellement : $Q\pi(s, a)$: la valeur d’état-action.

Pourquoi évaluer une politique ?

Avant de vouloir améliorer une politique (policy improvement), il faut savoir si elle est bonne.

La policy evaluation permet :

- d’estimer combien la politique rapporte en moyenne ;
- d’identifier les états bons/mauvais ;

Équations de Bellman

La valeur d’un état sous la politique π est définie par :

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

Ce que ça veut dire :

- On suit ce que la politique dit de faire (π).
- On regarde les transitions possibles.
- On ajoute la récompense immédiate + la valeur future.

Méthodes pour faire la policy evaluation

1 Approche exacte : Iterative Policy Evaluation

On applique l'équation de Bellman en boucle jusqu'à convergence.

Pseudo-code :

```
initialiser V(s) = 0 pour tout s

répéter
    pour chaque état s :
        V_new(s) = somme_a π(a|s) * somme_s' P(s'|s,a) * [R(s,a,s') + γ V(s')]
    remplacer V par V_new
jusqu'à convergence
```

---

2 Monte-Carlo Policy Evaluation

On fait plusieurs épisodes :

- on suit la politique π,
- on regarde le retour observé,
- on moyenne les retours pour estimer $V\pi$.

Adapté lorsque le modèle (transition, récompenses) n'est pas connu.

Résumé simple

Terme	Signification
Politique (π)	Règles de décision de l'agent
Policy Evaluation	Calculer combien rapporte cette politique
Méthodes	Bellman, Monte-Carlo

Terme	Signification
Utilité	Base de la policy iteration et de l'amélioration des politiques

Exemple

1) Rappel (formule utilisée — horizon fini)

On utilise la version horizon fini (avec discount) : si $V^{(h)}(s)$ est la valeur avec h étapes restantes et $V^{(0)}(s) = 0$,

$$V^{(h)}(s) = \sum_a \pi(a|s)\big(R(s,a) + \gamma V^{(h-1)}(s')\big)$$

où s' est l'état déterministe résultant de (s, a).

Paramètres utilisés :

- états = {North, East, South, West}
- actions = {clock, anti-clock, stay}
- γ = 0.9
- horizon (H = 10)
- politique $\pi(a|s) = 1/3$ (uniforme)

2) Exemple de calcul manuel (pour h=1, état North)

Pour vérifier la formule :

- transitions depuis North :
  - `clock` → East, (R=2)
  - `anti-clock` → West, (R=2)
  - `stay` → North, (R=-2)

Avec $V^{(0)}(\cdot) = 0$:

$$V^{(1)}(North) = \tfrac{1}{3}\big(2 + 0.9 \cdot 0\big) + \tfrac{1}{3}\big(2 + 0.9 \cdot 0\big) + \tfrac{1}{3}\big(-2 + 0.9 \cdot 0\big) = \tfrac{2+2-2}{3} = \tfrac{2}{3} \approx 0.66$$

(Ça colle avec les calculs programmés ci-dessous.)

3) Résultats (valeurs $V^{(h)}(s)$ pour h = 0..10)

h	North	East	South	West
0	0.0000	0.0000	0.0000	0.0000
1	0.6667	4.6667	1.6667	3.3333
2	3.2667	6.7667	4.5667	5.0333
3	5.1867	9.0467	6.5767	7.1933
4	7.0947	10.9097	8.5117	9.0203
5	8.7741	12.6215	10.1992	10.7213
6	10.2368	14.1881	11.6479	12.2011
7	11.5032	15.6288	12.8769	13.4774
8	12.9090	16.7549	14.3375	14.8503
9	14.0209	17.8671	15.4495	15.9624
10	15.0218	18.8679	16.4504	16.9632

La colonne h donne le nombre d'étapes restantes. La dernière ligne est donc $V^{(10)}(s)$ (valeur attendue en suivant π uniforme pendant 10 étapes).

4) Interprétation rapide

- East a la valeur la plus élevée (≈ 18.868 à l'horizon 10) : c'est logique car depuis East il y a une action avec grosse récompense (anti-clock = 10) et en moyenne la politique explore cette action 1/3 du temps.
- North a une valeur plus faible au départ (récompense de stay négative), mais en plusieurs étapes la valeur augmente car les transitions mènent à états plus rémunérateurs.
- Les valeurs augmentent avec h (plus d'étapes = plus de récompenses accumulées, malgré le discount).

Monte Carlo Online Control on Policy improvement (MCOC on P-imp)

Qu'est ce que le Monte Carlo Online Control (MCOC) ?

Le Monte Carlo online control est un algorithme, qui à partir des résultats récoltés, améliore la politique de jeu actuelle, dans le but de trouver la meilleure possible.

Pourquoi utiliser le MCOC ?

Cet algorithme est très utile pour apprendre une politique optimale à partir des intéractions avec l'environnement, car il se base sur son expérience. Cet algorithme est simple et possède un équilibre entre exploration et exploitation, il explore la plus part du temps l'action la plus optimate, mais explore de temps en temps d'autres actions selon ϵ.

Réalisation de l'Algorithme

Pour réaliser cet algorithme, nous commençons par initialiser les données suivants :

$$Q(s,a) = 0, N(S,a) = 0, \epsilon = 1, k = 1, \pi_i = \epsilon - greedy(Q)$$

ou

- s correspond à l'état
- a correspond à l'action
- Q (s, a) correspond à la valeur d'un état en effectuant une action, et en suivant la politique π_i
- N (s, a) correspond au nombre de visite
- ϵ correspond aux taux d'explorations dans la politique ϵ-greedy
- π_i correspond à la politique utilisée
- k correspond au compteur d'épisode

Utilisation de l'algorithme sur CliffWalking-v1

Nous avons utilisé cet algorithme sur l'environnement gymnasium CliffWalking afin de trouver le chemin le plus optimale, et voici le résultat obtenu :

Code de l'algorithme

```
import gymnasium as gym
import numpy as np
from scipy.signal import lfilter
import math
from collections import defaultdict


class MCOC:
    def __init__(self, env, gamma, epsilon):
        self.env = env
        self.gamma = gamma
        self.epsilon = epsilon

        self.q_values = defaultdict(float)
        self.n_visits = defaultdict(int)
        self.policy = {}
        self.num_actions = env.action_space.n
        self.num_states = env.observation_space.n
        self.episodes = []
        self.p = []

    def get_q(self, s, a): return self.q_values.get((s, a), 0.0)
    def get_n(self, s, a): return self.n_visits.get((s, a), 0.0)
    def get_policy(self, s, a): return self.policy.get((s, a), 0.0)

    def choose_action(self, p):
        return np.random.choice(range(self.num_actions), p=p)

    def best_action(self, state):
        return max(range(self.num_actions), key=lambda a:self.get_q(state, a))

    def evaluate(self, state, action, i, G):
        self.n_visits[(state, action)] = self.get_n(state, action) + 1
        #self.q_values[(state, action)] = self.get_q(state, action) +
(math.sqrt(1 / self.get_n(state, action)) * (G[i] - self.get_q(state, action)))
        self.q_values[(state, action)] = self.get_q(state, action) + (1 /
self.get_n(state, action) * (G[i] - self.get_q(state, action)))

    def epsilon_greedy(self, state):
        best_action = self.best_action(state)

        p = np.full(self.num_actions, self.epsilon / self.num_actions,
dtype=float)
        p[best_action] += 1.0 - self.epsilon

        for action in range(self.num_actions):
            self.policy[(state, action)] = p[action]

        return p


    def update_visits(self, episodes):
        visited = set()
        for t, (s, action, reward) in enumerate(episodes):
            # if first visit

            if (s, action) not in visited:
```

```
                agent.evaluate(s, action, t, G)
                visited.add((s, action))

        return visited

    def upgrade_policy(self):
        for state in range(self.num_states):
                agent.epsilon_greedy(state)



env = gym.make("CliffWalking-v1")
epsilon = 1
k = 1
gamma = 0.9
num_episodes = 150000
agent = MCOC(env=env, gamma=gamma, epsilon=epsilon)
visited = set()

#loop
for episode in range(num_episodes):
    if episode % 1000 == 0:
        print("Episode:", episode)
    state, _ = env.reset()
    finish = False
    visited = set()

    #sample k-th episode given pi_k
    while not finish:
        p = agent.epsilon_greedy(state)

        action = agent.choose_action(p)

        next_state, reward, terminated, truncated, _ = env.step(action)

        agent.episodes.append((state, action, reward))

        state = next_state

        finish = terminated or truncated

    #Gkt
    T = len(agent.episodes)
    rewards = np.array([r for (_, _, r) in agent.episodes])
    G = lfilter([1], [1, -gamma], rewards[::-1])[::-1]

    #for t= 1..T do
    visited = agent.update_visits(agent.episodes)

    agent.episodes = []
    k += 1
    agent.epsilon = max(1 / k, 0.05)
    #agent.epsilon = 1 / k

    #pi_k = epsilon-greedy(Q)
    agent.upgrade_policy()
```

Données utilisées :

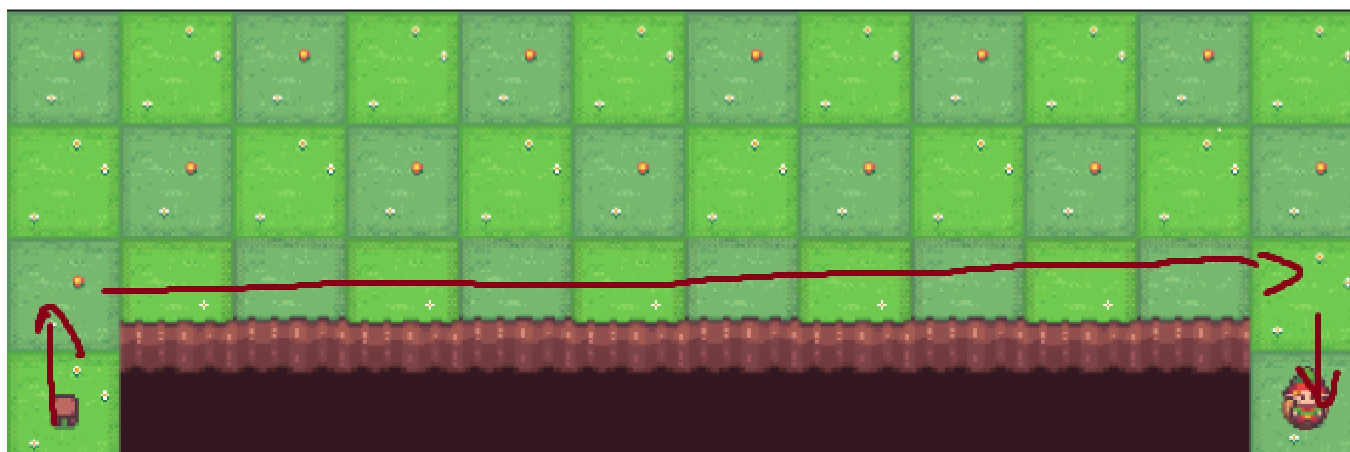
$$\gamma = 0.9, \ \epsilon = 1$$

Les essais

Nous avons réalisé des essais, et en faisant entre 150 000 et 400 000 episodes, nous obtenons toujours le même rewards, qui est égale à -17.

Conclusion

Il faut effectuer encore des millions d'episodes afin que l'agent puisse tout explorer et trouver la meilleur politique possible, qui donnerait le schéma suivant :



Utilisation de l'algorithme sur Cartpole-v1

Pour utiliser l'algorithme sur cartpole-v1, Nous avons du discretiser l'espace des etats de l'environnement avec une intervalle de 12 car Cartpole possède des états continus, alors que le MCOC nécessite des états discret.

Cartpole-v1 nous fournit les informations suivant :

- Position du chariot : de -4.8 à 4.8
- Vitesse du chariot : -∞ à +∞

- Angle du pôle : −0.418 rad à +0.418 rad
- Vitesse angulaire du pôle : −∞ à +∞

Code de l'algorithme :

```
def create_bins(num_bins=6):
    return [
        np.linspace(-4.8, 4.8, num_bins),          # position
        np.linspace(-5, 5, num_bins),              # vitesse
        np.linspace(-0.418, 0.418, num_bins),      # angle
        np.linspace(-5, 5, num_bins)               # vitesse angulaire
    ]

bins = create_bins()


def discretize(state):
    return tuple(int(np.digitize(s, b)) for s, b in zip(state, bins))


def MCOC(gamma, num_episodes):
    env = gym.make("CartPole-v1")

    # 6 bins par dimension → 6^4 = 1296 états
    num_states = 6 ** 4
    num_actions = env.action_space.n

    k = 1
    epsilon = 1

    Q_values = np.zeros((num_states, num_actions))
    N_visits = np.zeros((num_states, num_actions))
    policy = np.ones((num_states, num_actions)) / num_actions

    # --- MAIN LOOP ---
    for g in range(1, num_episodes + 1):
        episodes = []
        rewardepisode = 0

        state_cont, info = env.reset()
        state = get_index(discretize(state_cont))

        while True:
            # best action
            best_action = np.argmax(Q_values[state, :])

            # remplir proba
            for a in range(num_actions):
                if a == best_action:
                    policy[state, a] = 1 - epsilon + (epsilon / num_actions)
                else:
                    policy[state, a] = epsilon / num_actions

            p = policy[state]
            p = p / p.sum()

            action = np.random.choice(range(num_actions), p=p)
            next_state_cont, reward, terminated, truncated, _ =
env.step(action)
            rewardepisode += reward
            next_state = get_index(discretize(next_state_cont))
            episodes.append((state, action, reward))
```

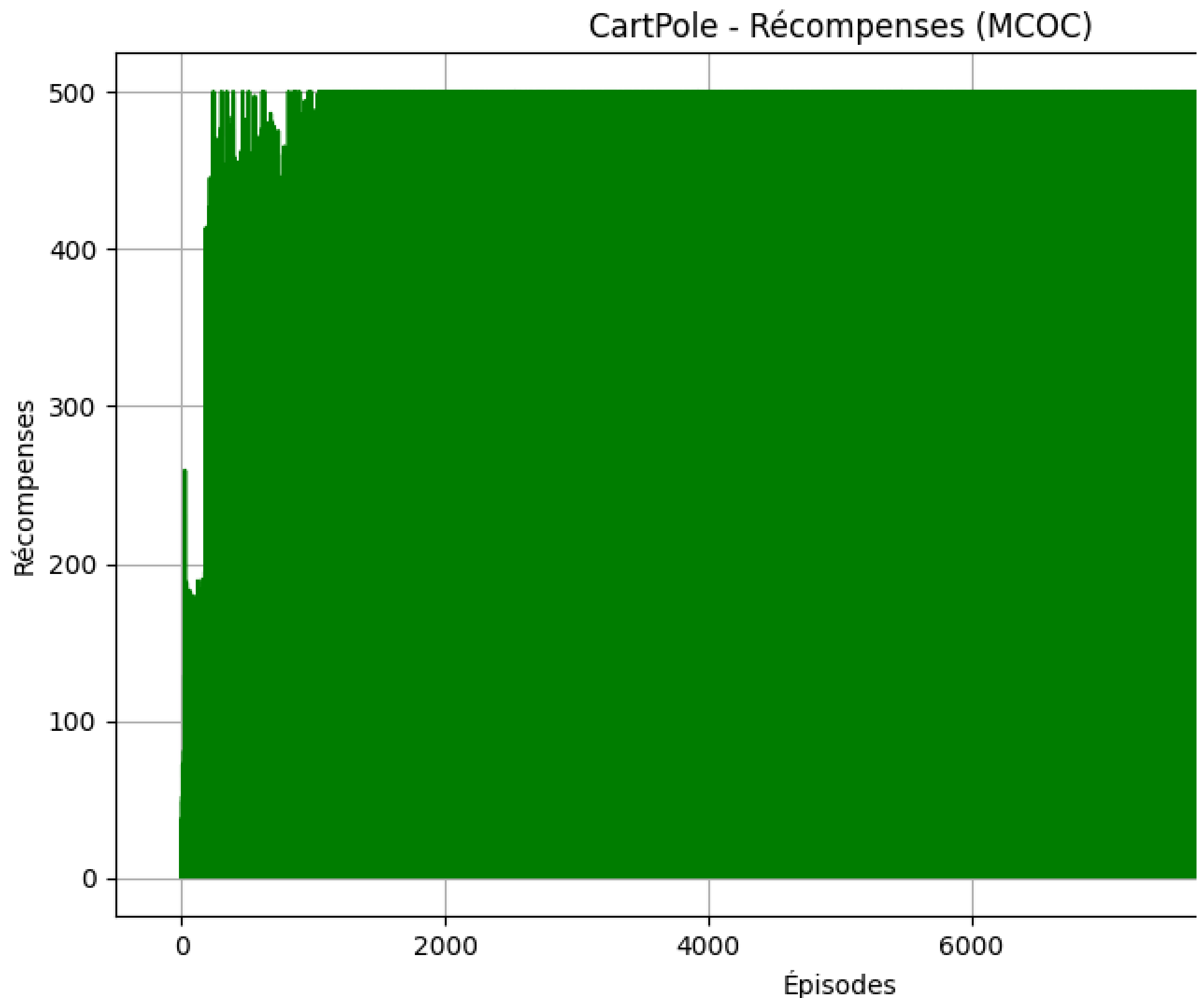
```
            if terminated or truncated:
                break

            state = next_state

        # --- Monte Carlo return ---
        G = {}
        T = len(episodes)
        G[T - 1] = episodes[T - 1][2]

        for t in range(T - 2, -1, -1):
            G[t] = episodes[t][2] + gamma * G[t + 1]

        # --- First Visit MC ---
        visited = set()
        for t, (state, action, reward) in enumerate(episodes):
            if (state, action) not in visited:
                visited.add((state, action))
                N_visits[state, action] += 1
                alpha = math.sqrt(1 / N_visits[state, action])
                Q_values[state, action] += alpha * (G[t] - Q_values[state,
action])

        # --- Policy Improvement ---
        k += 1
        epsilon = 1 / k

        for s in range(num_states):
            best_action = np.argmax(Q_values[s, :])
            for a in range(num_actions):
                if a == best_action:
                    policy[s, a] = 1 - epsilon + (epsilon / num_actions)
                else:
                    policy[s, a] = epsilon / num_actions

        print(g , rewardepisode)


    return Q_values, policy


# --- Convertit un tuple d'indices (ex: (3,2,1,4)) en un index unique 0..1295 -
--
def get_index(indices, num_bins=6):
    i0, i1, i2, i3 = indices
    return i0 * num_bins**3 + i1 * num_bins**2 + i2 * num_bins + i3
```

Données utilisées :

$$\gamma = 0.99,\ bins = 10,\ \epsilon = 1,\ k = 1$$

Essai et Résultat

Nous avons effectué un test de 10 000 episodes sur cartpole avec cet algorithme, et cela fût un front succès. L'algorithme a atteint le reward maximal à atteindre (500), comme on peut le voir dans le graphique en dessous, l'algorithme a été constant, du 1000 ème episode jusqu'à la fin.



SARSA (State, Action, Reward, next-State, next-Action)

Qu'est ce que le SARSA ?

SARSA est un algorithme d'apprentissage par renforcement **on-policy**, c'est-à-dire qu'il met à jour sa fonction de state-value $Q(s, a)$ en suivant exactement la politique utilisée pour agir. Contrairement au Q-Learning (off-policy), SARSA apprend en évaluant **les actions réellement choisies**, même lorsqu'il s'agit d'un choix aléatoire (exploration).

L'acronyme signifie :

- **s** : état actuel
- **a** : action choisie
- **r** : récompense reçue
- **s'** : état suivant
- **a'** : action suivante choisie par la même politique

La mise à jour est définie ainsi :

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma Q(s',a') - Q(s,a)\big]$$

Pourquoi utiliser SARSA ?

SARSA est particulièrement intéressant pour :

- apprendre à partir de l'expérience réelle dans l'environnement,
- gérer naturellement l'exploration via ϵ-greedy,
- conserver un comportement plus conservateur que Q-Learning (utile dans des environnements où des actions dangereuses mènent à des chutes rapides).

Comme il s'agit d'un algorithme **on-policy**, il évolue progressivement vers une politique sûre et performante tout en tenant compte des actions exploratoires.

Réalisation de l'algorithme

Dans cette implémentation, nous utilisons SARSA sur l'environnement **CartPole-v1**. Comme l'espace d'états est continu, nous devons le **discrétiser** avant l'apprentissage.

Variables initialisées :

- $Q(s,a) = 0$ pour tout couple état-action
- $N(s,a) = 0$ nombre de visites des couples
- $t = 0$ time-step gobal (multi-épisodique)
- $\alpha = 0.1$ alpha
- $\epsilon = \frac{1}{t}$ apprentissage on-policy
- π : ϵ-greedy sur Q

Structure générale du SARSA :

1. Observer s
2. Choisir a selon ϵ-greedy
3. Exécuter a, recevoir r et observer s'
4. Choisir a' selon la **même politique**
5. Mettre à jour :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a)\right]$$

6. Passer à $s \leftarrow s'$, $a \leftarrow a'$
7. Boucler sur les étapes 3 à 6 jusqu'à la fin de l'épisode

Implémentation

- Discrétisation en *n_bins* pour chaque variable de l'état
- Mise à jour SARSA pas à pas
- $\gamma = 0.9$
- Exploration décroissante proportionnelle à t
- $\epsilon\ minimal = 0.05$ (pas obligatoire pour l'algorithme)

Code utilisé

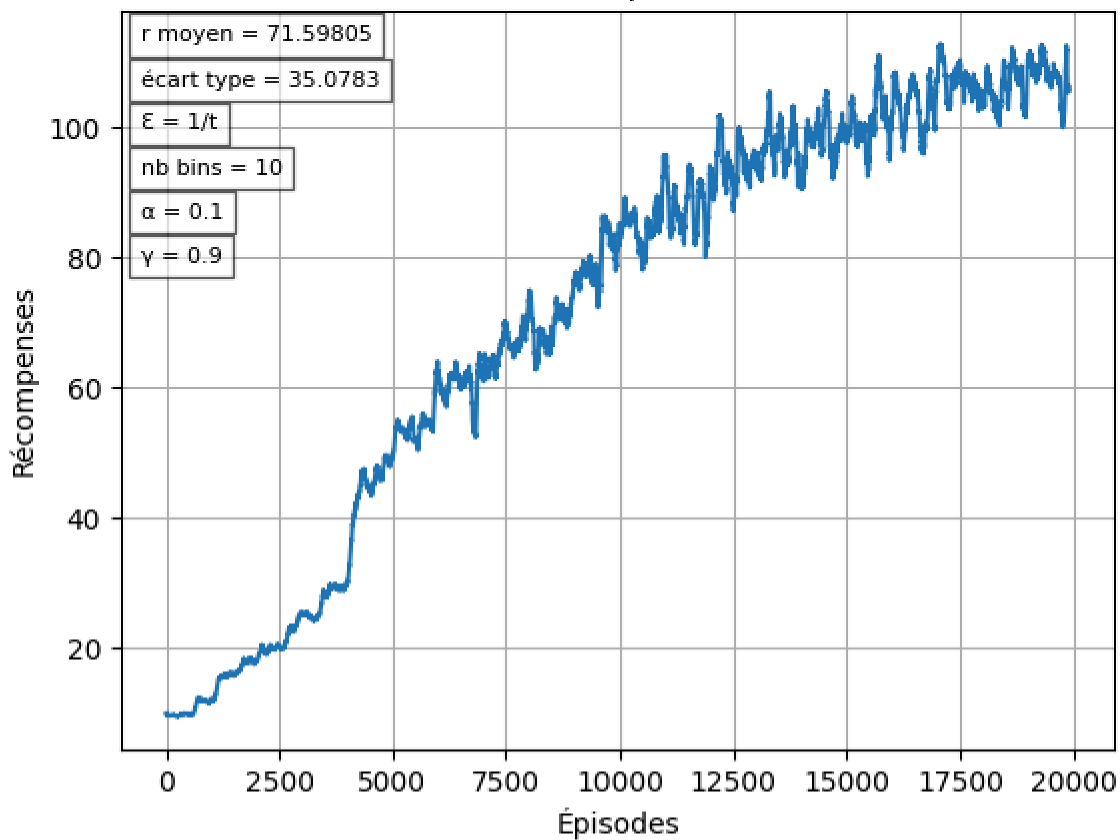
```
import gymnasium as gym
import random
import numpy as np

class SARSA:
    def __init__(self, alpha=0.1, gamma=0.9, n_bins=10):
        self.q = {}
        self.alpha = alpha
        self.gamma = gamma
        self.bins = {
            'cart_pos': np.linspace(-2.4, 2.4, n_bins),
            'cart_vel': np.linspace(-3.0, 3.0, n_bins),
            'pole_angle': np.linspace(-0.21, 0.21, n_bins),
            'pole_vel': np.linspace(-3.5, 3.5, n_bins)
        }

    def get_q(self, s, a):
        return self.q.get((s, a), 0.0)

    def choose_action(self, state, epsilon):
        if random.random() < epsilon:
            return random.randint(0, 1)
        state_disc = self.discretize(state)
        return np.argmax([self.get_q(state_disc, a) for a in [0, 1]])

    def discretize(self, state):
        state = np.clip(state, [-2.4, -3.0, -0.21, -3.5], [2.4, 3.0, 0.21, 
3.5])
        b = [
            np.digitize(state[0], self.bins['cart_pos']),
            np.digitize(state[1], self.bins['cart_vel']),
            np.digitize(state[2], self.bins['pole_angle']),
            np.digitize(state[3], self.bins['pole_vel'])
        ]
        return tuple(b)

    def learn(self, s, a, r, s_next, a_next):
        s = self.discretize(s)
        s_next = self.discretize(s_next)
        q_next = self.get_q(s_next, a_next)
        old_q = self.get_q(s, a)
        self.q[(s, a)] = old_q + self.alpha * (r + self.gamma * q_next - old_q)


env = gym.make("CartPole-v1")

alpha = 0.1
gamma = 0.9
nb_bins = 10
agent = SARSA(alpha=alpha, gamma=gamma, n_bins=nb_bins)
epsilon = 1.0
epsilon_min = 0.05
t = 0

for episode in range(20000):
```

```
        state, _ = env.reset()
        action = agent.choose_action(state, epsilon)
        total = 0
        done = False

        while not done:
            t += 1

            if epsilon > epsilon_min:
                epsilon = max(1 / t, epsilon_min)

            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            next_action = agent.choose_action(next_state, epsilon)

            agent.learn(state, action, reward, next_state, next_action)

            state, action = next_state, next_action
            total += reward

        print(f"Episode {episode+1}, score: {total}, epsilon={epsilon:.2f}")

    env.close()
```

Itérations avec plusieurs paramètres

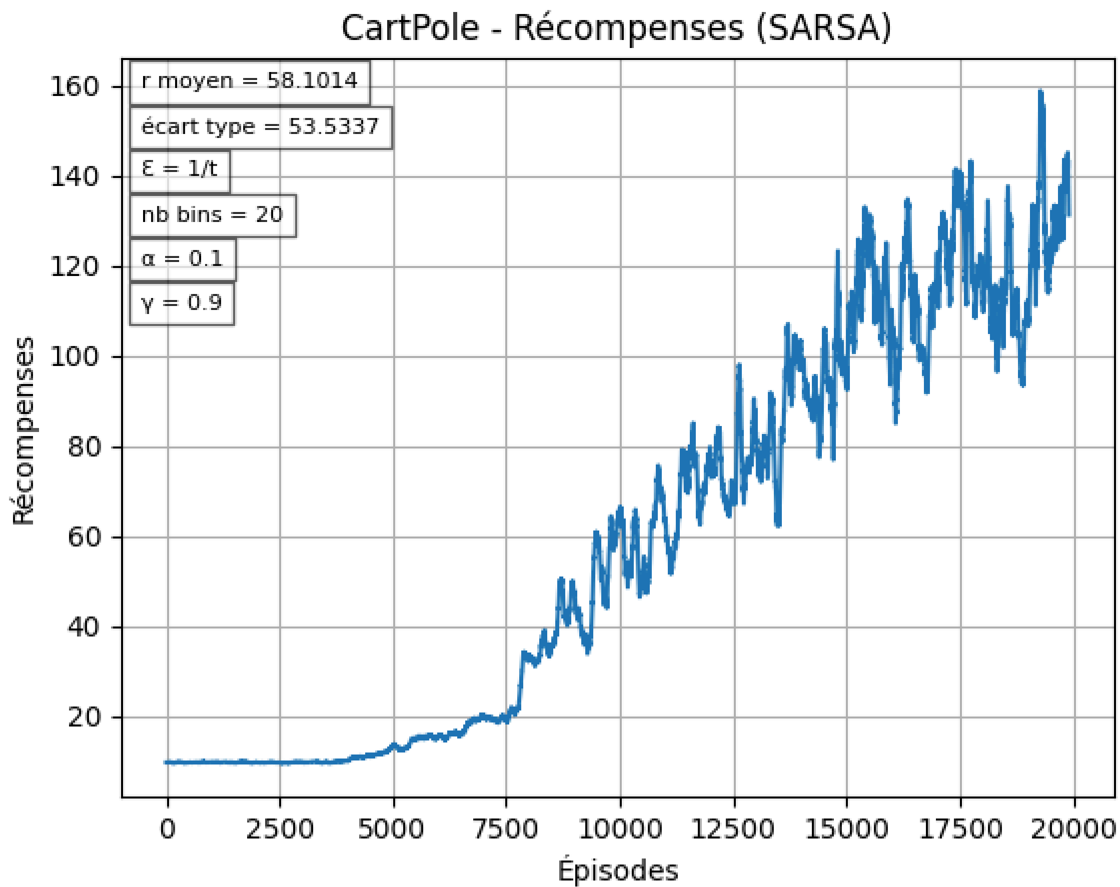
Avec ϵ = 1/t et α constant

Avec les paramètres "par défaut", les résultats obtenus sont assez faibles comparés au reward maximal de 500. C'est notamment dû au fait que ϵ diminue très vite.



Remarque sur la discrétisation

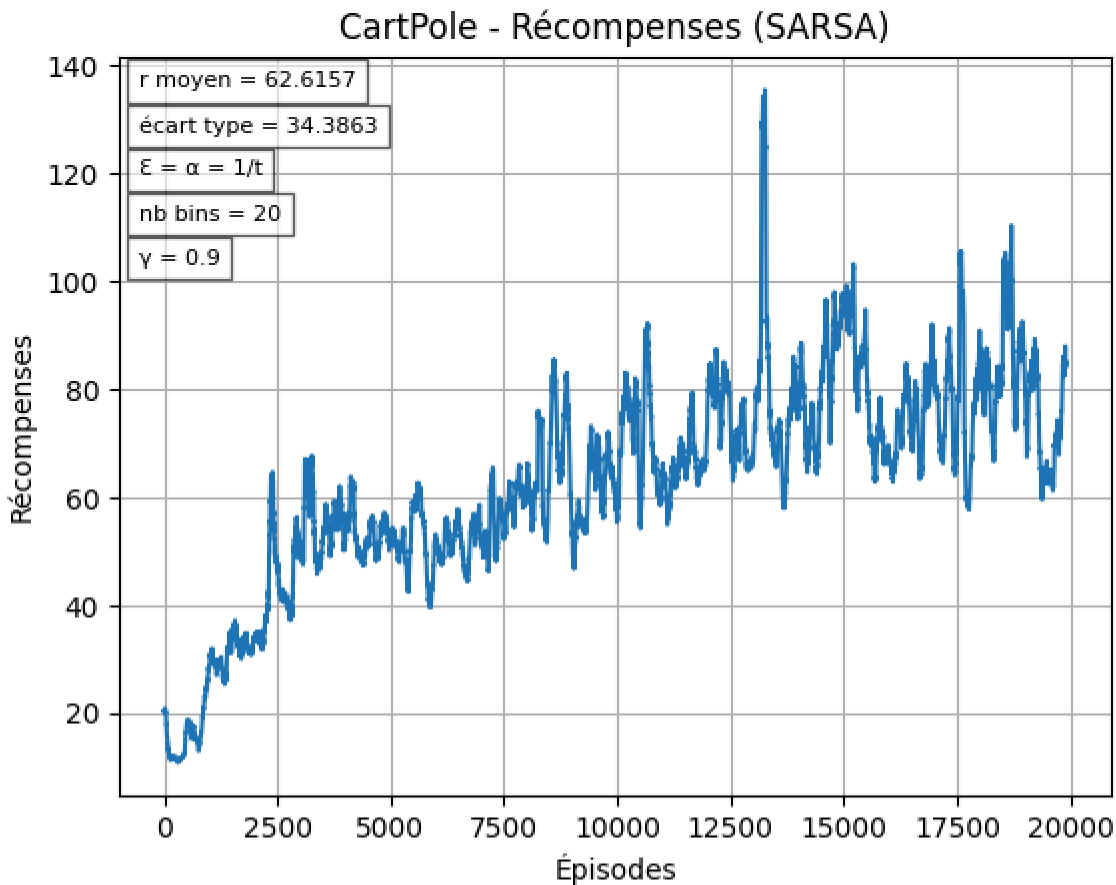
Pour cet environnement, la "résolution" (le nombre de bins) idéal pour les variables d'état semble être entre 15 et 25. On utilisera donc 20 bins pour les prochains tests:



Alpha Dynamique

Ensuite, on veut que le taux d'apprentissage général (α) évolue au fil des épisodes pour donner plus d'importance à la politique apprise au bout d'un certains temps (sinon on risque de stagner / être ralentit à cause des choix exploratoires).

$$\alpha = \frac{1}{t}$$



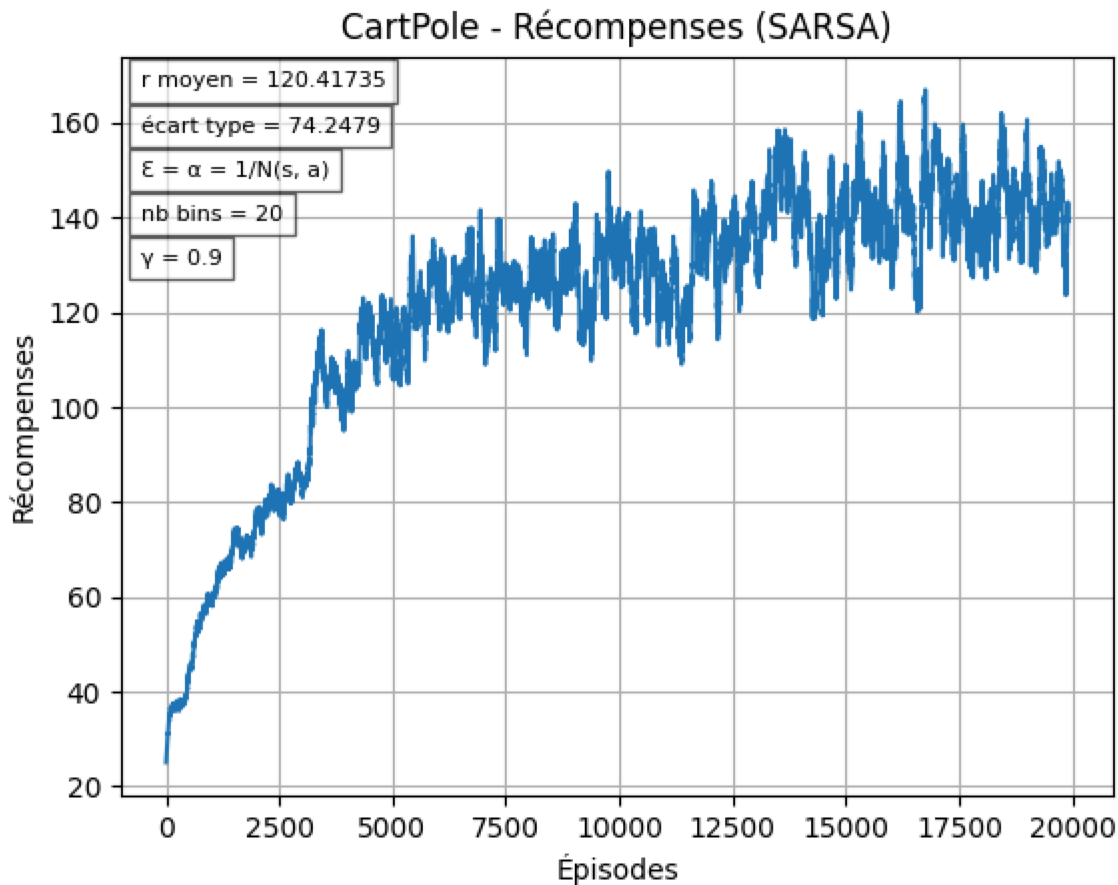
On peut observer ici que l'agent arrive un peu plus vite à des rewards "stables".

En utilisant N(s, a)

Maintenant, on a une phase majoritairement d'exploration, et une autre pour l'exploitation, mais on continue de diminuer nos chances d'explorer sans regarder si on a déjà une "bonne" politique pour l'état. Pour remédier à ça, on va modifier nos chances selon le nombre de visites de la paire (s, a):

$$\epsilon = \frac{1}{N(s,a)} \quad \alpha = \frac{1}{N(s,a)}$$

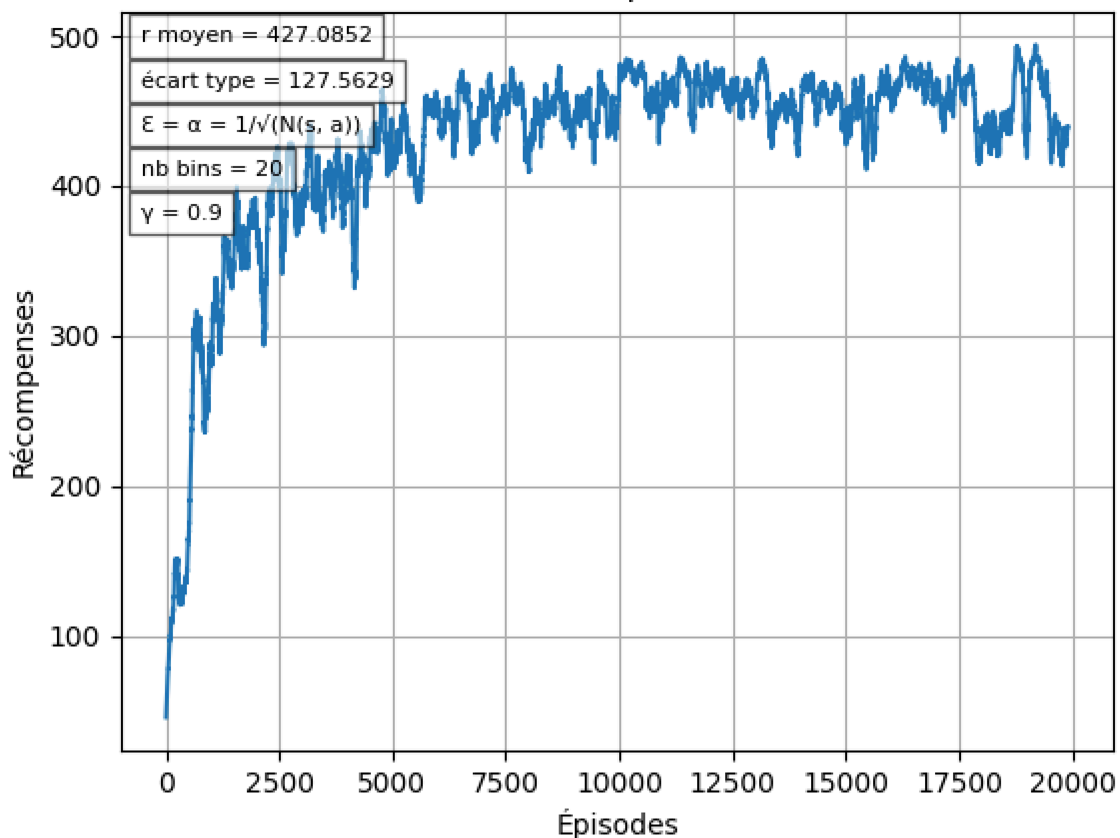
Cette solution permet d'avoir une décroissance de l'exploration qui est spécifique à chaque paire (s, a).



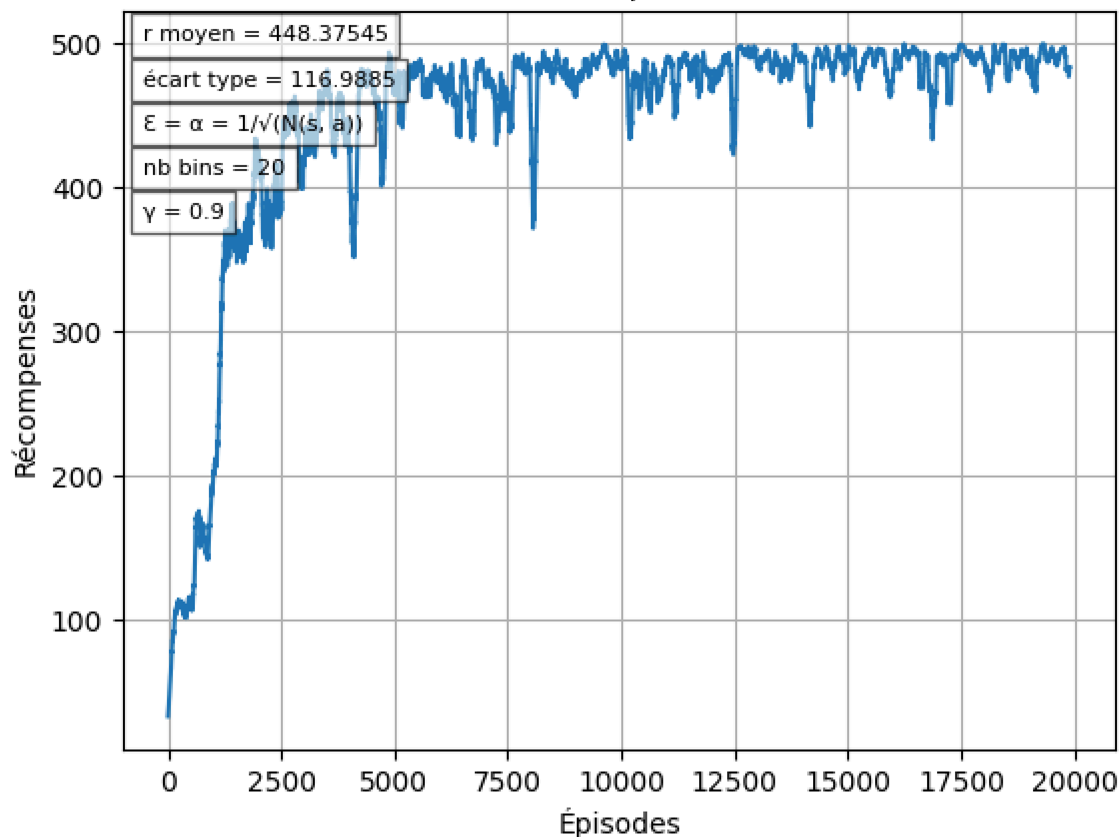
En utilisant la racine carrée de N(s, a)

Avec $\sqrt{N(s,a)}$ on permet à l'agent d'explorer un peu plus, ce qui affecte **grandement** les résulats. Comme on peut le voir ci-dessous, il est commun que l'agent s'approche de la politique optimale, mais cela dépend encore beaucoup de l'aléatoire durant les 2000 premiers épisodes.

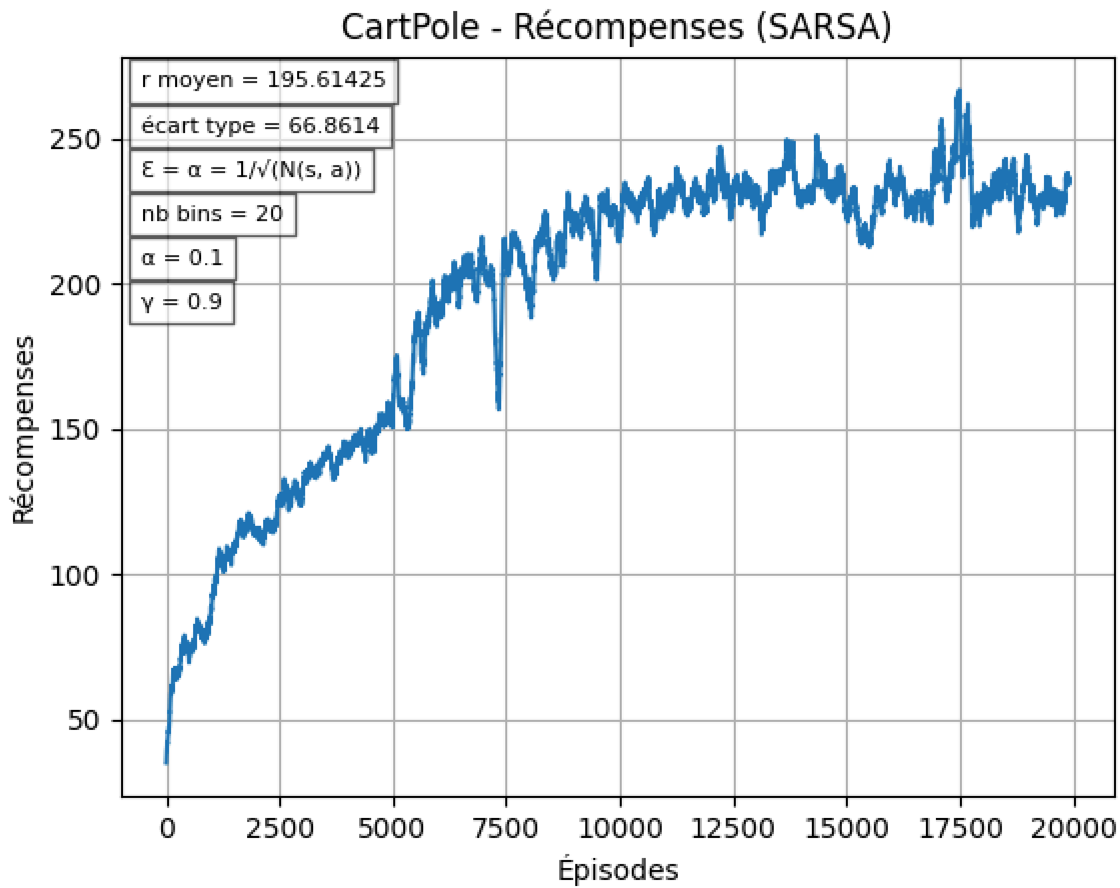
$$\epsilon = \frac{1}{\sqrt{N(s,a)}} \quad \alpha = \frac{1}{\sqrt{N(s,a)}}$$



Meilleur essai sur 20 tentatives:



Contre-exemple:



Conclusion

En résumé, SARSA est un algorithme robuste mais fortement sensible à l'initialisation **stochastique** : les premiers épisodes jouent un rôle déterminant, et un bon mécanisme de gestion de l'exploration *(comme une décroissance dépendant des visites)* peut faire toute la différence entre un apprentissage lent et la convergence vers une politique optimale.

Élagage Alpha-Beta

Qu'est ce que l'élagage Alpha-Beta ?

L'**élagage Alpha-Beta** c'est une amélioration de l'algorithme **Minimax**. Le principe de base c'est de parcourir un arbre de jeu pour trouver le meilleur coup possible, mais en évitant d'explorer les branches qui ne peuvent de toute façon pas changer la décision finale. Ça permet de gagner beaucoup de temps de calcul sans changer le résultat.

Pourquoi utiliser Alpha-Beta ?

Le problème avec Minimax c'est que sa complexité est **exponentielle** : avec un facteur de branchement b et une profondeur d, on doit évaluer $O(b^d)$ nœuds. Ça devient vite ingérable sur des jeux un peu complexes.

Alpha-Beta est utile pour :

- les jeux à **deux joueurs** comme les Échecs ou les Dames,
- les jeux à **information parfaite** où chaque joueur voit tout l'état du jeu,
- dès qu'on a une **fonction d'évaluation** pour noter les états non terminaux.

Avec un bon élagage, la complexité descend à $O(b^{d/2})$, ce qui revient à pouvoir chercher deux fois plus profond pour le même coût.

Principe du Minimax

Minimax repose sur deux rôles qui alternent à chaque niveau de l'arbre :

- **MAX** : le joueur veut **maximiser** son score.
- **MIN** : l'adversaire veut **minimiser** le score de MAX.

La valeur d'un nœud est calculée récursivement :

$$V(s) = \begin{cases} \text{eval}(s) & \text{si } s \text{ est terminal} \\ \max_a V(\text{succ}(s,a)) & \text{si c'est le tour de MAX} \\ \min_a V(\text{succ}(s,a)) & \text{si c'est le tour de MIN} \end{cases}$$

Pseudocode Minimax

```
function minimax(nœud, profondeur, joueur_max) :
    si profondeur == 0 ou nœud terminal :
        retourner eval(nœud)
    si joueur_max :
        valeur = -∞
        pour chaque enfant de nœud :
            valeur = max(valeur, minimax(enfant, profondeur-1, False))
        retourner valeur
    sinon :
        valeur = +∞
        pour chaque enfant de nœud :
            valeur = min(valeur, minimax(enfant, profondeur-1, True))
        retourner valeur
```

Élagage α-β

L'idée c'est de garder deux bornes pendant toute la recherche :

- **α** : le meilleur score que MAX a trouvé jusqu'ici (on commence à $-\infty$).
- **β** : le meilleur score que MIN a trouvé jusqu'ici (on commence à $+\infty$).

Dès qu'on a $\alpha \geq \beta$ sur un nœud, ça sert à rien de continuer à explorer ses enfants : le résultat ne changera pas. On coupe la branche.

Pseudocode Alpha-Beta

```
function alpha_beta(nœud, profondeur, α, β, joueur_max) :
    si profondeur == 0 ou nœud terminal :
        retourner eval(nœud)
    si joueur_max :
        valeur = -∞
        pour chaque enfant de nœud :
            valeur = max(valeur, alpha_beta(enfant, profondeur-1, α, β, False))
            α = max(α, valeur)
            si α ≥ β :
                break  # coupure β
        retourner valeur
    sinon :
        valeur = +∞
        pour chaque enfant de nœud :
            valeur = min(valeur, alpha_beta(enfant, profondeur-1, α, β, True))
            β = min(β, valeur)
            si α ≥ β :
                break  # coupure α
        retourner valeur
```

Complexité

Cas	Complexité
Minimax sans élagage	$O(b^d)$
Alpha-Beta (meilleur cas, ordre optimal)	$O(b^{d/2})$
Alpha-Beta (cas moyen, ordre aléatoire)	$O(b^{3d/4})$

L'ordre dans lequel on explore les coups a beaucoup d'importance : si on regarde les meilleurs coups en premier, on génère plus de coupures et on va plus vite.

Limites

Même avec l'élagage, Alpha-Beta reste bloqué sur les jeux avec un très grand espace d'états. Au Go par exemple, le facteur de branchement est d'environ 250 avec des parties de 150 coups. Même $O(b^{d/2})$ c'est bien trop lourd.

C'est pour ça qu'on s'intéresse à des algorithmes comme **MCTS**, qui n'essayent pas d'explorer tout l'arbre mais utilisent des simulations aléatoires pour estimer quelles branches valent la peine d'être creusées.

Démonstration Alpha-Beta

Ce que l'on veut montrer

L'algorithme Alpha-Beta retourne, pour tout nœud N et toute fenêtre $[\alpha, \beta]$ avec $\alpha \leq \beta$, une valeur $\alpha_\beta(N, \alpha, \beta)$ qui vérifie les trois cas suivants :

Cas	Condition	Résultat
Cas 1	$\alpha \leq \mathrm{val}(N) \leq \beta$	$\alpha_\beta(N, \alpha, \beta) = \mathrm{val}(N)$
Cas 2	$\mathrm{val}(N) < \alpha$	$\alpha_\beta(N, \alpha, \beta) \leq \alpha$
Cas 3	$\beta < \mathrm{val}(N)$	$\beta \leq \alpha_\beta(N, \alpha, \beta)$

La démonstration se fait par récurrence sur la profondeur de l'arbre de jeu.

Cas de base — N est une feuille

L'algorithme retourne directement $\mathrm{val}(N)$. Les trois cas sont trivialement vérifiés :

- **Cas 1 :** $\alpha \leq \mathrm{val}(N) \leq \beta$ → on retourne $\mathrm{val}(N)$. ✓
- **Cas 2 :** $\mathrm{val}(N) < \alpha$ → on retourne $\mathrm{val}(N) < \alpha$, donc $\alpha_\beta(N, \alpha, \beta) \leq \alpha$.
- **Cas 3 :** $\mathrm{val}(N) > \beta$ → on retourne $\mathrm{val}(N) > \beta$, donc $\beta \leq \alpha_\beta(N, \alpha, \beta)$.

Hérédité — N est un nœud interne

Hypothèse de récurrence (HR) : pour tout fils N_i de N, $\alpha_\beta(N_i, \alpha, \beta)$ vérifie les 3 cas.

On note $\alpha_i = \max(\alpha,\ \mathrm{val}(N_1),\ \ldots,\ \mathrm{val}(N_{i-1}))$ la valeur courante de α au moment d'explorer N_i (elle ne fait que croître au fil de l'exploration).

On traite le cas N nœud Max. Le cas Min est entièrement symétrique : il suffit d'échanger $\alpha \leftrightarrow \beta$, $\max \leftrightarrow \min$ et $\geq \leftrightarrow \leq$.

Cas 1 — $\alpha \leq \mathrm{val}(N) \leq \beta$

Puisque N est Max, $\mathrm{val}(N) = \max_i\ \mathrm{val}(N_i)$. Il existe donc un fils N^* tel que $\mathrm{val}(N^*) = \mathrm{val}(N)$.

N^* est bien atteint. Les fils $N_1, \ldots, N_{k-1}$ précédant N^* vérifient tous $\mathrm{val}(N_i) \leq \mathrm{val}(N) \leq \beta$. Par HR (cas 1 ou 2 selon leur valeur), aucun d'eux ne fait dépasser β à la variable `val` — aucune coupure β ne se déclenche avant N^*.

N^* retourne la bonne valeur. À l'arrivée sur N^*, on a $\alpha_k \leq \mathrm{val}(N^*) \leq \beta$, donc par HR cas 1 :

$$\alpha_\beta(N^*, \alpha_k, \beta) = \mathrm{val}(N^*) = \mathrm{val}(N)$$

La variable `val` vaut donc au moins $\mathrm{val}(N)$. Par ailleurs, tous les autres fils ont $\mathrm{val}(N_i) \leq \mathrm{val}(N)$, donc `val` ne dépasse pas $\mathrm{val}(N)$.

$$\alpha_\beta(N, \alpha, \beta) = \mathrm{val}(N)$$

Cas 2 — $\mathrm{val}(N) < \alpha$

Puisque N est Max, tous ses fils vérifient $\mathrm{val}(N_i) \leq \mathrm{val}(N) < \alpha \leq \alpha_i$.

Par HR cas 2 appliquée à chaque N_i :

$$\alpha_\beta(N_i, \alpha_i, \beta) \leq \alpha_i$$

La variable `val` ne remonte jamais au-dessus de α tout au long de la boucle. Aucune coupure β ne peut se déclencher (elle nécessiterait `val` $\geq \beta \geq \alpha$, impossible ici).

$$\alpha_\beta(N, \alpha, \beta) \leq \alpha$$

Cas 3 — $\mathrm{val}(N) > \beta$

Puisque N est Max, il existe un fils N^* tel que $\mathrm{val}(N^*) \geq \mathrm{val}(N) > \beta$.

Soit N^* le premier fils dans l'ordre d'exploration (d'indice k). Les fils $N_1, \ldots, N_{k-1}$ ont tous $\mathrm{val}(N_i) \leq \beta$, donc aucune coupure ne se déclenche avant d'atteindre N^*.

À l'arrivée sur N^*, on a $\mathrm{val}(N^*) > \beta$, donc par HR cas 3 :

$$\alpha_\beta(N^*, \alpha_k, \beta) \geq \beta$$

L'algorithme met alors `val` $= \max(\texttt{val},\ \alpha_\beta(N^*, \alpha_k, \beta)) \geq \beta$, ce qui déclenche la coupure β. On sort de la boucle et on retourne :

$$\alpha_\beta(N, \alpha, \beta) \geq \beta$$

Conclusion

Par récurrence sur la profondeur, la propriété est vraie pour tout nœud N de l'arbre.

À la racine R, on appelle $\alpha_\beta(R,\ -\infty,\ +\infty)$. Comme $-\infty \leq \text{val}(R) \leq +\infty$ est toujours vrai, le cas 1 s'applique :

$$\alpha_\beta(R,\ -\infty,\ +\infty) = \text{val}(R) = \text{minimax}(R)$$

Alpha-Beta est donc correct : il retourne exactement la même valeur que Minimax, tout en éliminant les branches dont on peut prouver — via les cas 2 et 3 — qu'elles ne peuvent pas influencer le résultat final.

Algorithme NegaMax

Qu'est-ce que NegaMax ?

L’algorithme **NegaMax** est une variante simplifiée de **Minimax** utilisée dans les jeux à deux joueurs à **somme nulle** (comme les Échecs ou les Dames).

L’idée principale est que ce qui est bon pour un joueur est directement mauvais pour l’autre. On peut donc éviter de distinguer les cas **MAX** et **MIN**, et utiliser une seule fonction récursive.

Pourquoi utiliser NegaMax ?

Dans Minimax, on doit gérer deux cas :

- maximiser pour un joueur,
- minimiser pour l’autre.

NegaMax simplifie ça en utilisant une propriété mathématique :

$$\max(a, b) = -\min(-a, -b)$$

Grâce à ça :

- le code est **plus court**,
- il est **plus facile à comprendre**,

Principe de NegaMax

Au lieu de distinguer MAX et MIN, on considère toujours qu’on cherche à **maximiser le score du joueur courant**.

Lorsqu’on change de joueur, on change simplement le signe du score.

La valeur d’un état est donnée par :

$$V(s) = \begin{cases} \text{eval}(s) & \text{si } s \text{ est terminal} \\ \max_a \left(-V(\text{succ}(s, a))\right) & \text{sinon} \end{cases}$$

Le **-** est essentiel : il permet de représenter le fait que l'adversaire joue contre nous.

Variable importante : le facteur de joueur

On introduit souvent une variable $color \in \{+1, -1\}$

Elle permet de savoir pour quel joueur on calcule le score :

- $+1$ → joueur courant (MAX)
- -1 → adversaire

Ainsi, l'évaluation devient :

$$color \times \text{eval}(s)$$

Pseudo-code NegaMax

```
function negamax(nœud, profondeur, color) :
    si profondeur == 0 ou nœud terminal :
        retourner color * eval(nœud)

    valeur = -∞

    pour chaque enfant de nœud :
        score = -negamax(enfant, profondeur-1, -color)
        valeur = max(valeur, score)

    retourner valeur
```

Version avec Alpha-Beta

NegaMax peut être combiné avec l'élagage Alpha-Beta pour améliorer les performances.

Pseudocode NegaMax avec Alpha-Beta

```
function negamax_alpha_beta(nœud, profondeur, α, β, color) :
    si profondeur == 0 ou nœud terminal :
        retourner color * eval(nœud)

    valeur = -∞

    pour chaque enfant de nœud :
        score = -negamax_alpha_beta(enfant, profondeur-1, -β, -α, -color)
        valeur = max(valeur, score)
        α = max(α, valeur)

        si α ≥ β :
            break  # coupure

    retourner valeur
```

Complexité

Comme Minimax, la complexité dépend de :

- b : facteur de branchement
- d : profondeur

Algorithme	Complexité
NegaMax (sans élagage)	$O(b^d)$
NegaMax + Alpha-Beta (meilleur cas)	$O(b^{d/2})$

Limites

Comme Minimax, NegaMax :

- explore un grand nombre de nœuds,
- devient vite coûteux pour des jeux complexes,
- dépend fortement de la **fonction d'évaluation**.

C'est pourquoi on utilise souvent :

- des heuristiques pour limiter la profondeur,
- ou des algorithmes comme **MCTS** pour les jeux très complexes.

À retenir

- NegaMax = Minimax simplifié
- On utilise une seule fonction + un changement de signe
- Très pratique en combinaison avec Alpha-Beta

Quiescence Search

Qu'est-ce que le Quiescence Search ?

Le **Quiescence Search** est une amélioration de NegaMax (ou Minimax) typiquement utilisée pour les échecs et le Go, qui permet d'éviter un problème classique appelé **effet d'horizon**.

L'idée est simple :
au lieu de s'arrêter brutalement à une profondeur donnée, on continue d'explorer **uniquement les coups "instables"** (captures, échecs, etc.) jusqu'à atteindre une position dite **calme**.

Pourquoi utiliser le Quiescence Search ?

Dans un algorithme classique, on coupe la recherche à une profondeur fixe :

- on évalue la position avec une fonction heuristique,
- mais cette position peut être **trompeuse**.

Exemple : Une pièce peut être attaquée mais pas encore capturée, donc la fonction d'évaluation ne voit pas la perte imminente.

C'est ce qu'on appelle l'**effet d'horizon** :
l'algorithme ne voit pas un événement important juste après la profondeur limite.

Le Quiescence Search permet de :

- éviter les évaluations instables,
- obtenir des scores plus fiables,
- améliorer la qualité des décisions.

Principe du Quiescence Search

Au lieu de retourner directement $\text{eval}(s)$ à profondeur 0 :

1. On calcule une valeur de base appelée **stand pat** :

$$stand_pat = eval(s)$$

2. On regarde uniquement certains coups particuliers (dit "tactiques") :
   - captures,
   - échecs,
   - promotions (selon le jeu)
3. On continue la recherche uniquement avec ces coups.

Condition d'arrêt

Une position est dite **calme** si aucun coup "violent" n'est possible.

Dans ce cas :

- on retourne simplement l'évaluation,
- sans explorer davantage.

Pseudo-code Quiescence Search

```
function quiescence(nœud, α, β, color) :
    stand_pat = color * eval(nœud)

    si stand_pat ≥ β :
        retourner β

    α = max(α, stand_pat)

    pour chaque coup tactique (captures, etc.) :
        score = -quiescence(enfant, -β, -α, -color)

        si score ≥ β :
            retourner β

        α = max(α, score)

    retourner α
```

Intégration avec NegaMax

Le Quiescence Search remplace le cas de base de NegaMax.

Au lieu de faire :

```
si profondeur == 0 :
    retourner color * eval(nœud)
```

On fait :

```
si profondeur == 0 :
    retourner quiescence(nœud, α, β, color)
```

Complexité

Le Quiescence Search :

- augmente légèrement le temps de calcul,
- mais reste limité car il explore peu de coups.

En pratique :

- il est **beaucoup moins coûteux** qu'augmenter la profondeur globale,
- tout en donnant de bien meilleurs résultats.

Limites

- Peut devenir coûteux si trop de coups tactiques sont possibles
- Nécessite un bon filtrage des coups (sinon explosion du nombre de nœuds)
- Ne résout pas tous les problèmes d'évaluation

À retenir

- Évite l'effet d'horizon
- Continue la recherche uniquement sur les positions instables
- Améliore fortement la qualité de l'évaluation
- S'utilise en complément de NegaMax + Alpha-Beta

UCB / UCT

Qu'est ce que UCB ?

UCB (*Upper Confidence Bound*) est une méthode pour résoudre le problème du **bandit manchot** (*multi-armed bandit*). L'idée du problème : on a plusieurs actions possibles (les "bras"), et on ne sait pas à l'avance quelle récompense chacune donne. On doit choisir quelle action jouer à chaque tour, en essayant de maximiser le total de récompenses sur le long terme.

La formule **UCB1** pour une action i est :

$$\text{UCB1}(i) = \frac{Q(i)}{N(i)} + C \cdot \sqrt{\frac{\ln N}{N(i)}}$$

avec :

- $Q(i)$: total des récompenses obtenues avec l'action i,
- $N(i)$: nombre de fois qu'on a choisi l'action i,
- N : nombre total d'actions jouées,
- C : coefficient qui règle l'exploration.

Pourquoi UCB ?

Le problème classique en apprentissage par renforcement c'est de trouver le bon équilibre entre **explorer** (tester des actions inconnues) et **exploiter** (jouer l'action qu'on sait être bonne).

UCB règle ça proprement :

- $\frac{Q(i)}{N(i)}$ est la récompense moyenne de l'action i → ça pousse à **exploiter** les bonnes actions.
- $C \cdot \sqrt{\frac{\ln N}{N(i)}}$ est un bonus qui augmente quand une action a été peu jouée → ça pousse à **explorer** les actions mal connues.

On choisit toujours l'action avec le score UCB1 le plus haut. Du coup, les actions peu testées finissent par avoir un bonus si grand qu'on finit par les essayer. C'est ce qui rend UCB intéressant : on a une garantie théorique que le regret cumulé croît en $O(\ln N)$, ce qui est optimal pour ce type de problème.

Paramètre C

C contrôle l'équilibre exploration/exploitation :

- C **grand** → l'agent explore beaucoup, au risque de ne pas assez exploiter ce qu'il connaît déjà.
- C **petit** → l'agent reste sur les actions connues, au risque de passer à côté d'une meilleure option.

La valeur théorique standard est $C = \sqrt{2}$, mais en pratique on l'ajuste selon le problème.

UCT — UCB Applied to Trees

UCT (*Upper Confidence bounds applied to Trees*) est simplement UCB1 appliqué aux nœuds d'un arbre de recherche MCTS. Pour chaque nœud enfant i, on calcule :

$$\mathrm{UCT}(i) = \frac{Q_i}{N_i} + C \cdot \sqrt{\frac{\ln N_{\text{parent}}}{N_i}}$$

avec :

- Q_i : somme des récompenses des simulations passées par ce nœud,
- N_i : nombre de fois qu'on a visité ce nœud,
- N_{parent} : nombre de visites du nœud parent.

Lors de la phase de **sélection** dans MCTS, on descend dans l'arbre en choisissant toujours l'enfant avec le meilleur score UCT. Les nœuds peu visités ont un bonus élevé, donc on finit toujours par les explorer, sans pour autant négliger les branches prometteuses.

> *"The UCT algorithm applies the UCB1 algorithm to the nodes of a Monte-Carlo search tree."*
> — Gelly & Silver, *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go*, Artificial Intelligence, vol. 175, n°11, 2011.

Monte Carlo Tree Search (MCTS)

Qu'est ce que MCTS ?

MCTS c'est un algorithme de recherche qui utilise des **simulations aléatoires** pour estimer quels coups valent la peine d'être explorés. Plutôt que de parcourir tout l'arbre de jeu comme Alpha-Beta, MCTS construit progressivement un arbre en jouant des parties aléatoires et en mémorisant les statistiques de chaque nœud. Plus on lui donne de temps, plus ses estimations sont précises.

C'est notamment grâce à MCTS que les programmes de jeu de Go ont fait un bond énorme, car Alpha-Beta est inutilisable sur ce jeu vu la taille de l'espace d'états.

Les 4 phases

MCTS répète les mêmes 4 étapes en boucle jusqu'à épuisement du temps de calcul :

1. Sélection

On part de la racine et on descend dans l'arbre en choisissant à chaque niveau le nœud enfant avec le meilleur score **UCT** :

$$\text{UCT}(i) = \frac{Q_i}{N_i} + C \cdot \sqrt{\frac{\ln N_{\text{parent}}}{N_i}}$$

On s'arrête quand on tombe sur un nœud qui n'a pas encore tous ses enfants développés, ou sur un état terminal.

2. Expansion

On ajoute un nouvel enfant au nœud sélectionné, correspondant à un coup qui n'a pas encore été exploré. C'est un état qu'on n'a jamais visité.

3. Simulation (rollout)

Depuis ce nouveau nœud, on joue une **partie entièrement aléatoire** jusqu'à la fin. Cela donne un résultat $z \in \{-1, 0, +1\}$ selon si on a perdu, fait nulle ou gagné.

4. Rétropropagation

On remonte le résultat z le long du chemin du nœud feuille jusqu'à la racine, en mettant à jour les compteurs de chaque nœud traversé :

$$N(s) \leftarrow N(s) + 1$$

$$Q(s, a) \leftarrow Q(s, a) + z$$

Ces mises à jour servent à améliorer les scores UCT pour les prochaines itérations.

Comparaison avec Alpha-Beta

Critère	Alpha-Beta	MCTS
Type de recherche	Exhaustive avec élagage	Guidée par statistiques
Évaluation des états	Heuristique explicite requise	Simulations aléatoires
Scalabilité	Limitée ($O(b^{d/2})$)	S'améliore avec le temps de calcul
Jeux ciblés	Échecs, Dames	Go, jeux à grand espace d'états

La différence principale c'est qu'Alpha-Beta a besoin d'une bonne fonction d'évaluation pour noter les états intermédiaires. MCTS n'en a pas besoin : il estime la valeur d'un état juste en jouant des parties aléatoires depuis cet état. C'est ce qui lui permet de s'adapter à des jeux comme le Go où écrire une bonne heuristique est très difficile.

RAVE / MC-RAVE

RAVE (*Rapid Action Value Estimation*) est une amélioration de MCTS qui permet d'apprendre plus vite. L'idée vient de l'hypothèse **AMAF** (*All Moves As First*) : si on a joué l'action a à un moment quelconque dans une simulation, on suppose que ce résultat est aussi utile pour estimer la valeur de a depuis l'état courant.

Concrètement, on combine deux estimations :

- $Q_{\text{MC}}(i)$: la valeur estimée par les simulations Monte Carlo classiques (lente mais fiable),

- $Q_{\text{RAVE}}(i)$: la valeur AMAF estimée sur toutes les simulations où le coup i a été joué (rapide mais biaisée).

La combinaison **MC-RAVE** est :

$$Q_{\text{MC-RAVE}}(i) = \beta \cdot Q_{\text{RAVE}}(i) + (1 - \beta) \cdot Q_{\text{MC}}(i)$$

β diminue au fur et à mesure que le nœud est visité : au début on fait plus confiance à RAVE (on a peu de données MC), et progressivement on bascule vers l'estimation MC. La formule proposée dans l'article est :

$$\beta = \sqrt{\frac{k}{3N_i + k}}$$

avec k un hyperparamètre qui contrôle à quelle vitesse on passe de RAVE à MC.

> *"The RAVE algorithm provides a rapid but biased estimate of action value; the Monte-Carlo algorithm provides a slower but unbiased estimate. MC-RAVE combines both estimates."* — Gelly & Silver, *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go*, Artificial Intelligence, vol. 175, n°11, 2011.

Pseudocode MCTS complet

```
import math
import random

class Noeud:
    def __init__(self, etat, parent=None):
        self.etat = etat
        self.parent = parent
        self.enfants = []
        self.N = 0        # nombre de visites
        self.Q = 0.0      # somme des récompenses

    def est_entierement_developpe(self, coups_legaux):
        coups_explores = [e.etat.dernier_coup for e in self.enfants]
        return all(c in coups_explores for c in coups_legaux)

    def score_uct(self, C=1.41):
        if self.N == 0:
            return float('inf')
        return self.Q / self.N + C * math.sqrt(math.log(self.parent.N) /
self.N)


def mcts(etat_racine, n_iterations):
    racine = Noeud(etat_racine)

    for _ in range(n_iterations):

        # --- 1. SÉLECTION ---
        noeud = racine
        etat = etat_racine.copie()

        while not etat.est_terminal() and
noeud.est_entierement_developpe(etat.coups_legaux()):
            noeud = max(noeud.enfants, key=lambda e: e.score_uct())
            etat.appliquer(noeud.etat.dernier_coup)

        # --- 2. EXPANSION ---
        if not etat.est_terminal():
            coups_explores = [e.etat.dernier_coup for e in noeud.enfants]
            coups_inexplores = [c for c in etat.coups_legaux() if c not in
coups_explores]
            coup = random.choice(coups_inexplores)
            etat.appliquer(coup)
            enfant = Noeud(etat.copie(), parent=noeud)
            noeud.enfants.append(enfant)
            noeud = enfant

        # --- 3. SIMULATION (rollout) ---
        etat_sim = etat.copie()
        while not etat_sim.est_terminal():
            coup = random.choice(etat_sim.coups_legaux())
            etat_sim.appliquer(coup)
        z = etat_sim.resultat()  # +1, 0, ou -1
```

```
        # --- 4. RÉTROPROPAGATION ---
        while noeud is not None:
            noeud.N += 1
            noeud.Q += z
            noeud = noeud.parent

    # on retourne le coup le plus visité (le plus fiable statistiquement)
    meilleur = max(racine.enfants, key=lambda e: e.N)
    return meilleur.etat.dernier_coup
```

Référence

Sylvain Gelly, David Silver. *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go.* Artificial Intelligence, vol. 175, n°11, juillet 2011, pp. 1856–1876.

Nous avons utilisé en premier lieu une bibliothèque appelée **Gymnasium** pour développer et tester nos algorithmes d’apprentissage par renforcement. Gymnasium est une bibliothèque open-source spécialement conçue pour fournir des environnements standardisés permettant d’entraîner et d’évaluer des agents d’IA.

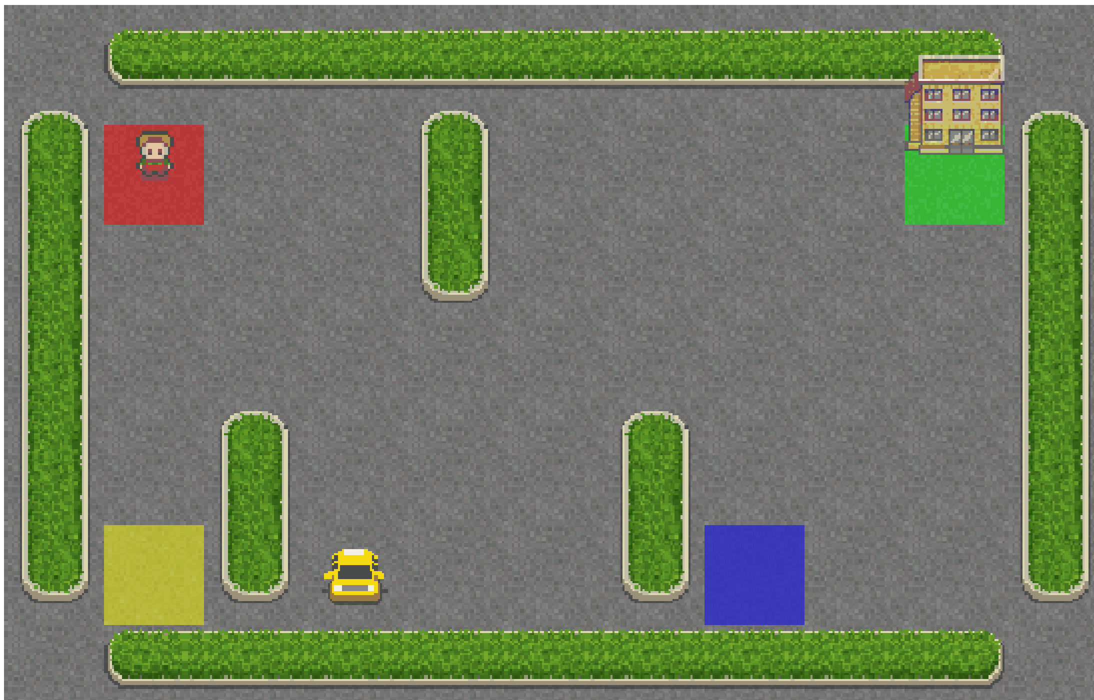
Avec Gymnasium, il est possible de :

- Créer et manipuler des environnements variés, allant de jeux simples à des simulations plus complexes.
- Tester facilement nos algorithmes grâce à une interface simple et unifiée.
- Visualiser les résultats en temps réel, ce qui facilite le débogage et l’amélioration des modèles.

Cette bibliothèque nous a permis de nous concentrer sur la logique de nos agents, sans avoir à recoder les environnements de zéro. Elle est largement adoptée dans la communauté du Reinforcement Learning pour sa flexibilité et sa simplicité d’utilisation.

Réalisation d'une IA jouant au jeu Taxi avec Q-Learning

Présentation du jeu Taxi



Le jeu Taxi est un environnement de type toy text disponible dans Gymnasium. Le but est simple :

- Un taxi doit récupérer un passager à un emplacement donné, puis le déposer à une destination précise dans une grille.
- Le taxi peut se déplacer dans quatre directions (nord, sud, est, ouest) et a la capacité de prendre ou déposer un passager.
- L'agent reçoit des récompenses négatives à chaque action (pour encourager la rapidité) et une récompense positive lorsqu'il dépose le passager à la bonne destination.

Cet environnement est idéal pour débuter avec le Q-Learning, car il est suffisamment simple pour être compris, mais assez complexe pour illustrer les défis du Reinforcement Learning.

Principe du Q-Learning

Le Q-Learning est un algorithme d'apprentissage par renforcement qui permet à un agent d'apprendre une politique optimale en interagissant avec son environnement. L'idée est d'apprendre une table de valeurs Q(s,a), où :

- Q(s,a) représente la valeur d'une action a dans un état s.
- L'agent met à jour cette table en utilisant la récompense immédiate et une estimation de la valeur future des états.

La formule de mise à jour de Q est la suivante :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

- α : taux d'apprentissage (learning rate)
- γ : facteur d'escompte (discount factor)
- r : récompense reçue après l'action
- s' : nouvel état après l'action

Étapes pour réaliser l'IA

Initialisation de l'environnement et des paramètres

On commence par importer les bibliothèques nécessaires et initialiser deux environnements :

- Un pour l'entraînement (sans rendu visuel).
- Un autre pour l'évaluation (avec rendu visuel).

```
import gymnasium as gym
import numpy as np

# Initialisation des environnements
env = gym.make("Taxi-v3")
eval_env = gym.make("Taxi-v3", render_mode="human")  # Pour visualiser l'agent
en action

# Nombre d'états et d'actions possibles
state_space = env.observation_space.n  # 500 états possibles
action_space = env.action_space.n      # 6 actions possibles

# Initialisation de la table Q avec des zéros
Q_table = np.zeros((state_space, action_space))

# Hyperparamètres
alpha = 0.1          # Taux d'apprentissage
gamma = 0.5          # Facteur d'escompte (importance des récompenses futures)
epsilon = 1.0        # Probabilité d'exploration (100% au début)
epsilon_dim = 0.999 # Décroissance de epsilon à chaque épisode
epsilon_min = 0.001 # Valeur minimale de epsilon
```

Boucle d'entraînement

Pour chaque épisode, l'agent interagit avec l'environnement en choisissant des actions selon une stratégie ϵ -gloutonne :

- Exploration : Avec une probabilité ϵ, l'agent choisit une action aléatoire.
- Exploitation : Sinon, il choisit l'action avec la plus grande valeur Q pour l'état courant.

À chaque étape, la table Q est mise à jour selon la formule du Q-Learning.

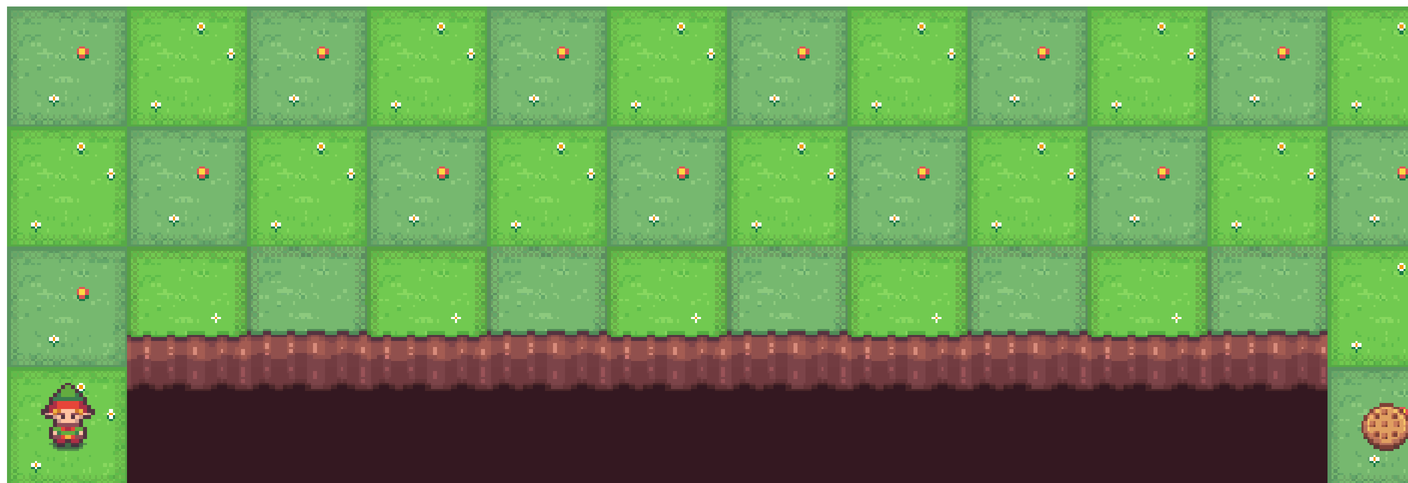
```
for ep in range(10000):
    terminated = False
    truncated = False

    # Utilisation de l'environnement d'évaluation pour les 10 derniers épisodes
    if ep >= 9990:
        observation, info = eval_env.reset()
        current_env = eval_env
    else:
        observation, info = env.reset()
        current_env = env

    while not (terminated or truncated):
        # Stratégie epsilon-gloutonne
        if np.random.uniform(0, 1) < epsilon:
            # Exploration : action aléatoire
            action = current_env.action_space.sample()
        else:
            # Exploitation : action avec la meilleure valeur Q
            valid_actions = np.where(info["action_mask"] == 1)[0]  # Actions
valides
            q_values = Q_table[observation, valid_actions]
            action = valid_actions[np.argmax(q_values)]

        # Exécution de l'action
        next_state, reward, terminated, truncated, info =
current_env.step(action)
        print("Récompense :", reward)

        # Mise à jour de la table Q
        Q_table[observation, action] = (1 - alpha) * Q_table[observation,
action] + alpha * (reward + gamma * np.max(Q_table[next_state]))

        # Passage à l'état suivant
        observation = next_state

    # Décroissance de epsilon
    epsilon = max(epsilon_min, epsilon * epsilon_dim)
```

Points à retenir

- **Exploration** vs **Exploitation** : Le paramètre ϵ permet de trouver un équilibre entre découvrir de nouvelles stratégies et exploiter les connaissances acquises.

- Décroissance de ϵ : ϵ diminue exponentiellement pour favoriser l'exploitation en fin d'entraînement.
- Actions valides : Le masque d'actions (`action_mask`) évite de choisir des actions invalides dans certains états.

Présentation de CliffWalking



Le jeu CliffWalking est un environnement classique de Gymnasium.

Le but du jeu est simple, la plateau de jeu est en 4x12, un agent doit partir d’un point de départ (case [3, 0])et atteindre l'arrivée (case [3, 11]) le plus rapidement possible tout en évitant de tomber de la falaise.

Objectifs :

Apprendre à atteindre l'arrivée le plus rapidement possible.

Éviter les cases du qui représentent la falaise car sinon l'agent recevra une récompense négative à la hauteur -100 et devra recommencer depuis son point de départ.

Chaque déplacement de l'agent coûte une pénalité de -1.

Principe du Q-Learning

Voir la partie sur Taxi

Structure du code

Pour le code, on commence avec l'importation des bibliothèques.

```python
from collections import defaultdict
import gymnasium as gym
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
```

Ensuite on créée une classe qui implémente le comportement de l'agent Q-Learning. Elle gère différentes choses :

- la table Q qui enregistre les valeurs Q pour chaque état et action
- le epsilon-greedy qui équilibre exploration et exploitation.
- La mise à jour de la table Q selon la formule du Q-Learning.
- la diminution progressive de epsilon, pour réduire l'exploration au fil du temps.

```python
class CliffAgent:

    def __init__(self,
                 env: gym.Env,
                 learning_rate: float,
                 initial_epsilon: float,
                 epsilon_decay: float,
                 final_epsilon: float,
                 discount_factor: float):
        self.env = env

        # Table Q initialisée à zéro pour chaque état/action
        self.q_values = defaultdict(lambda: np.zeros(env.action_space.n))
        self.lr = learning_rate
        self.discount_factor = discount_factor

        # Paramètres d'exploration
        self.epsilon = initial_epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon

        # Historique de l'erreur de mise à jour
        self.training_error = []
```

L'agent choisit soit une action aléatoire avec la probabilité de epsilon, on parle alors d'exploration sinon on parle d'exploitation c'est à dire que l'agent choisiral'action la plus optimisée selon sa table Q.

```python
    def get_action(self, state) -> int:
        if np.random.random() < self.epsilon:
            return self.env.action_space.sample()  # Exploration
        else:
            return int(np.argmax(self.q_values[state]))  # Exploitation
```

À chaque interaction avec l'environnement, la valeur Q(s,a) est mise à jour en fonction de la récompense reçue et de la valeur estimée du prochain état.

```
    def update(self, state, action, reward, next_state, terminated):
        # Estimation de la meilleure valeur future
        future_q = (not terminated) * np.max(self.q_values[next_state])

        # Cible de mise à jour (target)
        target = reward + self.discount_factor * future_q

        # Différence temporelle (TD error)
        temporal_difference = target - self.q_values[state][action]

        # Mise à jour de Q(s,a)
        self.q_values[state][action] += self.lr * temporal_difference

        # Sauvegarde de l'erreur pour analyse
        self.training_error.append(temporal_difference)
```

Au fil des épisodes, le epsilon va diminuer pour que l'agent se fie davantage à ce qu'il a appris.

```
    def decay_epsilon(self):
        self.epsilon = max(self.final_epsilon, self.epsilon -
self.epsilon_decay)
```

On initialise les derniers paramètres et on entraîne l'agent sur 6000 épisodes.

```
learning_rate = 0.01
n_episodes = 6000
start_epsilon = 1.0
epsilon_decay = start_epsilon / (n_episodes / 2)
final_epsilon = 0.1
discount_factor = 0.95

env = gym.make("CliffWalking-v1", render_mode="ansi")
agent = CliffAgent(
    env=env,
    learning_rate=learning_rate,
    initial_epsilon=start_epsilon,
    epsilon_decay=epsilon_decay,
    final_epsilon=final_epsilon,
    discount_factor=discount_factor
)
```

L'agent joue chaque épisode, met à jour ses valeurs Q, puis diminue epsilon. Chaque épisode correspond à une tentative complète pour atteindre la case d'arrivée.

```
rewards_per_episode = []
for episode in tqdm(range(n_episodes)):
    state, info = env.reset()
    done = False
    total_reward = 0

    while not done:
        # Choix de l’action selon epsilon-greedy
        action = agent.get_action(state)

        # Exécution dans l’environnement
        next_state, reward, terminated, truncated, info = env.step(action)

        # Mise à jour des valeurs Q
        agent.update(state, action, reward, next_state, terminated)

        total_reward += reward
        done = terminated or truncated
        state = next_state

    # Décroissance de epsilon à chaque épisode
    agent.decay_epsilon()
    rewards_per_episode.append(total_reward)

env.close()
```

Enfin pour évaluer la progression au cours des épisodes, on peut générer un graphique.

```
window = 100
moving_avg = np.convolve(rewards_per_episode, np.ones(window)/window,
mode="valid")

plt.plot(moving_avg)
plt.title("CliffWalking - Moyenne des récompenses")
plt.xlabel("Épisodes")
plt.ylabel("Récompense moyenne")
plt.show()
```

Voici le graphique à la fin des épisodes qu'on a fixé :

res cliffWalking

On remarque bien l'évolution où l'agent va tomber énormément de fois de la falaise et faire des mouvements inutiles au début des épisodes car on a des récompenses de plus -5000, pour au final d'une manière exponentielle, l'agent tendra vers les -14 après au bout de 1000 épisodes.

Réalisation d'une IA jouant au jeu Acrobot avec MCTS

Présentation du jeu Acrobot

Le jeu Acrobot est un environnement de type contrôle classique disponible dans Gymnasium. Le but du jeu est simple :

Un système à deux segments articulés (un pendule double) doit amener l'extrémité du second segment au dessus d'une certaine hauteur. L'agent peut appliquer un couple moteur limité sur l'articulation centrale, ce qui influence le mouvement des deux segments. L'agent reçoit des récompenses négatives à chaque étape, ce qui a pour but d'encourager l'agent à trouver la solution le plus rapidement possible. La réussite est atteinte lorsque l'extrémité du pendule dépasse la hauteur cible (la ligne noire).

Cet environnement est idéal pour expérimenter avec des algorithmes de planification comme le MCTS, car il combine une dynamique continue avec des actions discrètes simples, tout en offrant un défi non trivial de contrôle et de planification à court et long terme.

Monte Carlo Tree Search (MCTS)

Le **Monte Carlo Tree Search (MCTS)** est un algorithme de planification et d'apprentissage par simulation qui permet à un agent de choisir des actions en explorant un arbre de décisions. L'idée est d'estimer la valeur des actions en simulant des parties complètes depuis l'état courant et en utilisant ces simulations pour guider le choix :

- Chaque nœud de l'arbre représente un **état** de l'environnement, et chaque branche correspond à une **action possible**.
- L'agent sélectionne les actions à explorer en équilibrant **exploitation** (choisir les actions avec de bonnes valeurs estimées) et **exploration** (essayer des actions moins connues) via une formule comme UCB1 :

$$\text{score} = \frac{V_i}{N_i} + c \cdot \sqrt{\frac{\ln N_p}{N_i}}$$

où :

- (Vi) : valeur cumulée du nœud (i)
- (Ni) : nombre de visites du nœud (i)
- (Np) : nombre de visites du parent
- (c) : coefficient d'exploration

(Algorithme UCB1, réalisé en python, permettant de choisir le meilleur noeud parmis plusieurs)

```
def ucb1(children, c=1.0):
    parent_visits = sum(child.visits for child in children.values()) + 1e-5
    best_score = -float('inf')
    best_child = None
    for action, child in children.items():
        if child.visits == 0:
            score = float('inf')
        else:
            exploit = child.value / child.visits
            explore = math.sqrt(math.log(parent_visits) / child.visits)
            score = exploit + c * explore
        if score > best_score:
            best_score = score
            best_child = child
    return best_child
```

- Après chaque simulation, **la récompense finale est propagée vers le haut** de l'arbre (backpropagation) pour mettre à jour les valeurs des nœuds traversés.
- À la fin des itérations, l'action choisie est celle qui maximise la valeur moyenne estimée au nœud racine.

Ainsi, MCTS permet à l'agent de planifier plusieurs coups à l'avance sans connaître explicitement la dynamique de l'environnement, en s'appuyant sur des simulations pour guider ses décisions.

Étapes pour réaliser l'IA

Réalisation des fonctions nécessaires :

Importation des bibliothèques nécessaires et création de l'objet "Node" (noeud)

L'objet Node (noeud) possède :

- Un état
- un parent
- un enfant
- le nombre de visites
- une valeur

```
import gymnasium as gym
import math
import random
import copy

env = gym.make("Acrobot-v1")
obs, info = env.reset()

# Crtion d'un objet représentant le noeud d'un arbre
class Node:
    def __init__(self, state, parent):
        self.parent = parent
        self.state = state
        self.visits = 0
        self.value = 0.0
        self.children = {}
```

La fonction de simulation

Cette fonction permet, à partir d'un noeud, de faire X simulations à partir du noeud de la fin du jeu.

```python
def simulate(Inode: Node, max_steps):
    total_reward = 0
    env_simu = copy.deepcopy(env.unwrapped)
    env_simu.state = Inode.state
    steps = 0
    done = False

    #Tant que le programme n'a pas réussi, ou que le max_steps n'a pas été
atteint, on réalise des simulations
    while not done and steps < max_steps:
        steps += 1
        action = env_simu.action_space.sample()
        obs, reward, terminated, truncated, _ = env_simu.step(action)
        total_reward += reward

        if terminated:
            done = True

    return env_simu.state, done, total_reward
```

La fonction "Backpropagate"

La méthode backpropagate joue un rôle central dans le MCTS en permettant de remonter la récompense obtenue après une simulation à travers l’arbre de décision

```python
def backpropagate(node: Node, reward):
    while node is not None:
        node.visits += 1
        node.value += reward
        node = node.parent
```

La fonction de réalisation de cycle

La fonction cycle représente une itération complète de MCTS à partir d’un nœud donné. Elle commence par créer les enfants du nœud si nécessaire, puis sélectionne un noeud au hasard parmi les enfants nouvellement créés. À partir de ce nœud, plusieurs simulations sont effectuées, chaque simulation produisant une récompense cumulée qui est ensuite propagée vers le haut de l’arbre grâce à backpropagate

```
def cycle(noeud, env):
    simulate_node = noeud
    all_actions = list(range(env.action_space.n))
    if noeud.children == {}:
        for action in all_actions:
            noeud.children[action] = Node(state=copy.deepcopy(noeud.state),
parent=noeud)
        simulate_node = random.choice(list(noeud.children.values()))
    for _ in range(random.randint(50, 100)):
        final_state, done, epdone, total_reward = simulate(simulate_node,
max_steps=300)
        simulate_node.state = final_state
        backpropagate(simulate_node, reward=total_reward)
    return final_state, done, epdone, total_reward
```

La fonction "MCTS"

Dans cette fonction, on rassemble tout ce que l'on a codé auparavant. On commence par initialisé le noeud racine ainsi que certaines variables. Ensuite, dans la boucle "while", tant que le jeu n'est pas terminé, ou que les episodes ne se sont pas écoulés et que le compteur est inférieur aux nombres d'itérations choisis, on effectue des cycles.

```
def mcts(env, iterations=100):
    racine = Node(state=copy.deepcopy(env.unwrapped.state), parent=None)
    done = False
    compteur = 0
    while not done and compteur < iterations:
        final_state, done, epdone, total_reward = cycle(noeud=racine, env=env)
        compteur += 1
        print('episode :', compteur)
        print('recompense actuel :', total_reward)

    if done:
        print('Agent a reussi')
    else:
        print('Agent a pas réussi')

    print("Nb d'itérations :", compteur)
    print("Nombre de visites racine :", racine.visits)
    print("Nombre d’enfants :", len(racine.children))
    return racine
```

Observations

- Après réflexion, la réalisation d’un MCTS sur un jeu tel que Acrobot n’est pas très idéale car l’environnement possède un espace d’état continu et infini avec une dynamique physique complexe, or, le MCTS est efficace pour des espaces discret et limitées,

comme le jeu de plateau Taxi. Pour ce type d’environnement, le Q-Learning est plus adapté pour apprendre une politique optimale.

Réalisation d'une IA jouant au blackjack avec MCTS

Présentation du blackjack

Principe du jeu

Le Blackjack est un jeu de cartes où le but est de battre le croupier (dealer) en obtenant une main dont la valeur est la plus proche possible de 21, sans jamais la dépasser.

Règles principales

- Les cartes numérotées valent leur valeur (ex. 7 = 7 points).
- Les **figures** (Valet, Dame, Roi) valent 10 points.
- L'**As** vaut 1 ou 11 points, selon ce qui avantage le joueur.
- Le joueur commence avec 2 cartes, tout comme le croupier (une visible, une cachée).

- Le joueur peut :
  - **Hit** : tirer une nouvelle carte.
  - **Stand** : s'arrêter et conserver sa main actuelle.
- Le croupier joue ensuite selon des règles fixes :
  - Il **tire** tant que sa main est ≤ 16.
  - Il **reste** à 17 ou plus.

Objectif

- Gagner si :
  - Votre total est supérieur à celui du croupier sans dépasser 21.
  - Le croupier dépasse 21 .
- Perdre si vous dépassez 21 ou si le croupier a une meilleure main.

Particularités de la version Gymnasium

- Le jeu se joue dans un environnement simulé (`Blackjack-v1`) de la bibliothèque Gymnasium.
- Les différents états possibles sont indépendants les uns les autres car il y a remise dans la pioche à chaque tirage.

Principe du Monte Carlo Tree Search (MCTS)

Le Monte Carlo Tree Search (MCTS) est un algorithme d'apprentissage par renforcement basé sur la simulation.
Il permet à un agent de prendre des décisions optimales en explorant un arbre de recherche et en simulant de nombreuses parties à partir des états possibles.

L'idée est d'équilibrer l'exploration (essayer de nouvelles actions) et l'exploitation (choisir les actions prometteuses déjà connues) pour estimer la meilleure décision à chaque étape.

---

1. Sélection (Selection)

À partir de la racine (état actuel), l'algorithme descend dans l'arbre en sélectionnant les nœuds selon une politique d'équilibre entre exploration et exploitation, à l'aide de la formule UCB1 (Upper Confidence Bound) :

$$UCB1 = \frac{w_i}{n_i} + \sqrt{\frac{c \cdot \ln N}{n_i}}$$

- w_i : somme des récompenses du nœud i

- n_i : nombre de visites du nœud i
- N : nombre total de simulations depuis le nœud parent
- c : constante d'exploration (généralement 2)

2. Expansion (Expansion)

Si le nœud sélectionné n'est pas terminal, on ajoute un ou plusieurs nouveaux nœuds enfants correspondant aux actions possibles depuis cet état.

3. Simulation (Rollout)

Une simulation aléatoire est exécutée à partir du nouvel état jusqu'à la fin de la partie (ou un horizon fixé).
Le résultat (victoire, défaite, score) fournit une évaluation de cet état.

4. Rétropropagation (Backpropagation)

Le résultat de la simulation est propagé en remontant l'arbre : chaque nœud met à jour son nombre de visites et son score moyen.

À force de répétitions, le MCTS améliore ses estimations pour chaque action.
L'action optimale depuis l'état initial est généralement celle avec le plus grand nombre de visites ou la meilleure moyenne de récompense.

Structure de notre IA

L'intelligence artificielle repose sur deux classes principales.

La classe `State`

Cette classe représente un état du jeu, c'est-à-dire la situation actuelle dans une partie de Blackjack : les cartes du joueur, la carte visible du croupier, et la présence éventuelle d'un as compté comme 11.

Elle contient plusieurs attributs essentiels.
L'attribut `state` correspond à l'état du jeu tel qu'il est fourni par l'environnement Gymnasium.
`visits` indique combien de fois cet état a déjà été rencontré au cours des simulations.
`actions` regroupe les deux choix possibles dans une partie de Blackjack : « stand » (`Action(0)`) ou « hit » (`Action(1)`).

Enfin, `__utc_c` est une constante d'exploration utilisée dans le calcul de la valeur UCB, qui permet de gérer le compromis entre exploration et exploitation.

La méthode `UCB_value(self, action)` calcule la valeur UCB (Upper Confidence Bound) pour une action donnée. C'est à cet endroit que se décide l'équilibre entre tester de nouvelles actions et exploiter celles qui semblent déjà prometteuses.

La méthode interne `__select_action(self)` choisit l'action à jouer selon les valeurs UCB ou, si l'état n'a encore jamais été visité, de manière aléatoire. Concrètement, si `self.visits` vaut 0, on n'a encore aucune donnée, donc l'action est tirée au hasard. Sinon, on compare les deux valeurs UCB (pour les actions 0 et 1) et on garde celle qui a la meilleure valeur. En cas d'égalité parfaite, une action aléatoire est choisie afin de conserver un minimum d'exploration.

La méthode `select_learning_action(self)` sert d'interface publique pour la sélection d'une action pendant la phase d'apprentissage. Elle se contente d'appeler `__select_action()` et applique donc la logique du MCTS avec exploration activée.

La méthode `select_operating_action(self)` est utilisée quand l'IA joue réellement, c'est-à-dire lorsqu'elle doit exploiter pleinement ce qu'elle a appris sans explorer. Pour cela, la constante d'exploration `__utc_c` est temporairement désactivée, puis l'action est choisie en fonction des valeurs moyennes observées. Une fois la sélection faite, la valeur d'origine de `__utc_c` est rétablie. Ce procédé permet à l'IA de prendre la meilleure décision possible à partir de son expérience accumulée.

Enfin, la méthode `backtraking(self)` incrémente simplement le compteur de visites de l'état (`self.visits += 1`). Elle est appelée à la fin de chaque simulation afin de mettre à jour l'arbre de recherche.

Code source :

```python
import gymnasium as gym
import math
import random

# specifly design for gymnasium blackjack , previous step doesn't influence
the outcom of current
# so we redesigned some part of classical MCST to be optimized to this
challenge !
class State:
    __utc_c = 1
    def __init__(self, state):
        self.state = state
        self.visits = 0
        self.actions = [Action(0),Action(1)]


    # pré-requis : self.visits > 0
    def UCB_value(self,action):
        act_visits = max(1,action.get_visits()) # pour éviter la division par 0
        act_value = action.get_value()
        act_avg = act_value/act_visits

        exploration = math.sqrt(self.__utc_c*math.log(self.visits)/act_visits)

        return act_avg + exploration

    def choose_random_action(self):
        if random.uniform(0,1) >= 0.5:
            return self.actions[0]
        else:
            return self.actions[1]

    def __select_action(self):
        if self.visits == 0:
            return self.choose_random_action()

        act0_UCB = self.UCB_value(self.actions[0])
        act1_UCB = self.UCB_value(self.actions[1])

        if act0_UCB > act1_UCB :
            return self.actions[0]
        elif act0_UCB == act1_UCB:
            return self.choose_random_action()
        else:
            return self.actions[1]

    def select_learning_action(self):
        return self.__select_action()

    def select_operating_action(self):
        temp = self.__utc_c
        self.__utc_c = 0
        a =  self.__select_action()
        self.__utc_c = temp
        return a
```

```python
    def backtraking(self):
        self.visits = self.visits + 1
```

La classe `Action`

La classe `Action` représente une action possible dans une partie de Blackjack. Elle correspond concrètement au choix que l'agent peut faire à un instant donné : tirer une nouvelle carte (*hit*) ou s'arrêter (*stand*). Chaque objet `Action` garde en mémoire ses statistiques individuelles au fil des simulations.

Lors de son initialisation, la classe reçoit un paramètre `order` qui identifie l'action à effectuer dans l'environnement (`0` pour *stand*, `1` pour *hit*). Trois attributs sont ensuite définis :

- `order` , qui indique le type d'action,
- `visits` , le nombre de fois où cette action a été jouée depuis cet état,
- `value` , la somme totale des récompenses obtenues lorsque cette action a été choisie.

Plusieurs méthodes permettent d'accéder à ces informations.
`get_order()` renvoie simplement le numéro associé à l'action.
`get_visits()` retourne le nombre de fois où cette action a été essayée.
`get_value()` donne la valeur totale accumulée grâce aux récompenses obtenues.

La méthode `backtracking(self, reward)` est appelée après une simulation, lorsque le résultat (ou la récompense) de l'action est connu. Elle met à jour les statistiques associées en incrémentant le compteur de visites et en ajoutant la récompense reçue à la valeur totale. C'est grâce à cette mise à jour que l'algorithme peut progressivement estimer la qualité réelle de chaque action et améliorer sa prise de décision au fil des parties.

Code source :

```python
class Action:
    def __init__(self,order):
        self.order = order   # la valeur de l'action dans l'environnement
        self.visits = 0
        self.value = 0.0

    def get_order(self):
        return self.order

    def get_visits(self):
        return self.visits

    def get_value(self):
        return self.value

    def backtracking(self,reward):
        self.visits = self.visits + 1
        self.value += reward
```

Dans l'ensemble l'IA est un set de l'ensemble des states que l'on crée au cours du temps

---

Fonctionnalités développées

Nous avons développé 4 grandes fonctionnalités :

Entraînement de l'IA

Tout commence avec la fonction `ai_training()` . Celle-ci crée d'abord un environnement de jeu Blackjack grâce à la bibliothèque Gymnasium. L'utilisateur choisit ensuite combien de parties l'IA doit jouer pour s'entraîner. Ce nombre d'épisodes détermine la durée et la qualité de l'apprentissage : plus il est élevé, plus l'agent aura exploré de scénarios possibles.

Au début de chaque partie, on réinitialise le jeu et on crée un dictionnaire `dic_node_game` qui va garder en mémoire les états rencontrés et les actions associées pendant cette partie. Ce dictionnaire est temporaire et sera utilisé plus tard pour mettre à jour les statistiques.

Pendant le déroulement du jeu, l'IA observe l'état actuel (c'est-à-dire les cartes du joueur, la carte visible du croupier et la présence d'un as utilisable). Si cet état n'a jamais été rencontré auparavant, on le crée et on l'ajoute à la liste des états découverts.
Ensuite, l'agent choisit une action à l'aide de la méthode `select_learning_action()` de la classe `State` .

L'environnement exécute alors cette action grâce à la fonction `env.step()` , qui renvoie le nouvel état, la récompense obtenue et des indicateurs indiquant si la partie est terminée. Si le jeu n'est pas encore fini, l'IA continue à jouer jusqu'à ce que la manche se conclue.

Quand la partie se termine, le programme affiche la récompense finale (+1.5 pour un blackjack,+1 pour une victoire, -1 pour une défaite, 0 pour une égalité). Ensuite, la phase de mise à jour commence : pour chaque couple (état, action) rencontré pendant la partie, on appelle les méthodes `backtraking()` sur l'état et `backtracking(reward)` sur l'action. Ces deux appels servent à incrémenter les compteurs de visites et à mettre à jour les valeurs de récompense moyenne. C'est ce mécanisme qui permet à l'IA d'apprendre quelles décisions mènent le plus souvent à la victoire.

Une fois toutes les parties d'entraînement terminées, l'ensemble des états et des actions explorés est sauvegardé grâce à la classe `AI_saver`. Cela permet de réutiliser plus tard le modèle entraîné sans devoir recommencer tout le processus.

Extrait du code source :

```
def ai_training():
    descovered_states = {}

    # Utilisation de Gymnasium pour l'environnement de jeu
    env = gym.make("Blackjack-v1")

    n_episodes_train = ask_number("Please choose the number of game that the ai
should train on")
    reward = 0
    for episode in range(n_episodes_train):
        print("##################################################")
        print(f"nouvelle épisode {episode}")
        state, info = env.reset() # Nouvelle partie
        reward = 0
        done = False

        dic_node_game = {} # Un dictionnaire State -> Action qui retient le
déroulement de la partie

        while not done:
            if state not in descovered_states:
                descovered_states[state] = State(state)

            node_state = descovered_states[state]

            action = node_state.select_learning_action()

            dic_node_game[node_state] = action


            # On joue l'action proposer par l'IA et récupère le retour de
l'environnement
            next_state, reward, terminated, truncated, info =
env.step(action.get_order())
            done = terminated or truncated

            state = next_state

        # en dehors du while, la partie est finie
        print(f"Episode finished! Total reward: {reward}")
        # On rétropropage le reward de la partie
        for n_state,n_action in dic_node_game.items():
            n_state.backtraking()
            n_action.backtracking(reward)
    ai_saver.save(descovered_states,game_name,n_episodes_train)
    env.close()
```

Gestion des IA et sauvegarde des modèles

Pour éviter de devoir réentraîner une intelligence artificielle à chaque exécution, une partie essentielle du projet consiste à pouvoir sauvegarder et recharger une IA déjà entraînée.

C’est exactement le rôle de la classe `AI_saver`, qui s’appuie sur le module Pickle de Python. L’objectif est de rendre la gestion des IA simple, pratique et flexible, tout en permettant de conserver plusieurs versions d’un même modèle.

Le fonctionnement général est le suivant : lorsqu’une IA termine sa phase d’apprentissage, l’utilisateur peut la sauvegarder dans un dossier spécial nommé `.ai_saves`. À ce moment-là, le programme lui demande de choisir un nom pour la retrouver plus facilement par la suite. Le fichier est ensuite automatiquement complété par la date et l’heure de la sauvegarde, ainsi que par le nombre d’épisodes d’entraînement effectués. Cela donne des noms de fichiers à la fois uniques et informatifs, comme par exemple `blackjack_ai_test_02-11-2025--17:30_trained_5000_times.plk`.

Si le dossier n’existe pas encore, il est créé automatiquement grâce à `os.makedirs`. Une fois le nom final prêt, l’objet représentant l’IA est sauvegardé dans un fichier binaire grâce à la fonction `pickle.dump()`.

Pickle est un module standard de Python qui permet de sérialiser et désérialiser des objets. La sérialisation, c’est le processus qui consiste à transformer un objet Python (par exemple une instance de classe contenant des dictionnaires, des listes et même d’autres objets) en une suite d’octets. Ces octets peuvent ensuite être enregistrés dans un fichier ou envoyés à travers un réseau. La désérialisation, c’est l’opération inverse : à partir du fichier binaire, Pickle reconstitue exactement l’objet d’origine, avec toutes ses valeurs, sa structure et son état interne.

Concrètement, quand on écrit `pickle.dump(ai, f)`, Pickle parcourt récursivement tous les attributs de l’objet `ai`, traduit leur contenu en une forme binaire, et les écrit dans le fichier ouvert en mode `wb` (write binary). Lors du chargement avec `ai = pickle.load(f)`, le module lit ces mêmes octets et reconstruit un objet Python identique à celui qui avait été sauvegardé, en restaurant toutes les références et les valeurs. C’est ce qui permet de retrouver une IA exactement dans l’état où elle était après son apprentissage, comme si elle n’avait jamais été arrêtée.

Il faut cependant noter que Pickle est spécifique à Python. Les fichiers produits ne sont pas lisibles directement par d’autres langages, et il ne faut pas charger un fichier Pickle provenant d’une source non fiable, car le processus de désérialisation peut exécuter du code malveillant.

La méthode `load()` de la classe `AI_saver` permet de restaurer une IA précédemment sauvegardée. Elle commence par lister tous les fichiers disponibles dans le dossier `.ai_saves`, éventuellement filtrés selon le nom du jeu, comme "blackjack". Cette recherche se fait grâce à la fonction `list_files()`, qui utilise le module `glob` pour parcourir les fichiers avec l’extension `.plk`.

Le programme affiche ensuite la liste des fichiers trouvés et demande à l’utilisateur de choisir celui qu’il souhaite recharger. Une fois la sélection faite, le fichier est ouvert en lecture binaire et passé à `pickle.load()`, qui reconstruit l’objet complet en mémoire. L’IA ainsi rechargée est ensuite renvoyée pour être utilisée dans les autres fonctions du programme, comme la visualisation ou la comparaison de stratégies.

Ce système de sauvegarde et de restauration rend la gestion des intelligences artificielles bien plus simple. Il devient possible d’entraîner plusieurs modèles différents, de les comparer entre eux ou de reprendre l’apprentissage là où on s’était arrêté, sans perte de données.

Code source :

```
import pickle
from datetime import datetime
import glob
import os
import sys

# retourne tous les fichiers dans le directory qui contiennent substring ou
tous si None
def list_files(directory, substring=None):
    all_files = glob.glob(os.path.join(directory, "*.plk"))
    all_files = [f for f in all_files if os.path.isfile(f)]
    if substring:
        all_files = [f for f in all_files if substring in os.path.basename(f)]
    return all_files

def ask_number_in_list(msg, values_list):
    while True:
        try:
            val = int(input(f"{msg} {values_list} : "))
            if val in values_list:
                return val
            else:
                print(f"Wrong input. Please choose beetween {values_list}.")
        except ValueError:
            print("Invalid input. Please choose a number.")


class AI_saver:
    folder_name = ".ai_saves"
    def __init__(self):
        pass

    def save(self,ai,game_name="no_specified",nb_training=None):

        # construction du nom du fichier
        now = datetime.now()
        result = input("Choose a name to find more easely your ai :")
        name = self.folder_name+"/"+game_name+"_ai_"+result # nom de base du
fichier
        name = name + "_" + now.strftime("%d-%m-%Y--%H:%M") # on ajoute la date

        # ajout du nombre d'entrainement eventuellement ainsi que l'extension
du fichier
        if nb_training is None :
            name += "_no_training_given.plk"
        else:
            name += "_trained_"+str(nb_training)+"_times.plk"

        # Création du dossier si nécessaire
        os.makedirs(os.path.dirname(name), exist_ok=True)

        with open(name, "wb") as f:
            pickle.dump(ai,f)

        print(f"the ai as been save under the name {name}")
```

```
    def load(self,game_name=None):
        availables_files = list_files(self.folder_name,game_name)

        if len(availables_files) == 0:
            print("files not found : exiting")
            sys.exit()

        choices = []
        print("here is the list of file found with their number choice:")
        for i in range(len(availables_files)):
            choices.append(i)
            print(f"choice {i} : {availables_files[i]}")

        choice = ask_number_in_list("select the ai you want to restore :
",choices)

        print(f"You have been select the file {availables_files[choice]}")

        with open(availables_files[choice], "rb") as f:
            ai_loaded = pickle.load(f)

        return ai_loaded
```

Comparaison de l'efficacité de l'IA

Une fois l'intelligence artificielle entraînée et sauvegardée, il est important de pouvoir évaluer ses performances et de les comparer à d'autres stratégies. C'est exactement le rôle de la fonction `methods_comparaison()` . Elle permet de mesurer, sur un grand nombre de parties, à quel point notre IA joue mieux qu'un joueur aléatoire ou qu'une stratégie classique de croupier.

Le principe est simple : on recharge d'abord une IA déjà entraînée puis on fait jouer trois « joueurs » différents sur le même environnement de Blackjack. Le premier est l'IA, le deuxième représente le comportement typique d'un croupier (tirer une carte quand la main est de 16 ou moins, et s'arrêter sinon), et le troisième joue complètement au hasard, en choisissant aléatoirement entre « hit » et « stand ».

Pour chacun de ces joueurs, le programme lance une série de parties, dont le nombre est défini par l'utilisateur. À chaque tour, il enregistre les résultats dans un dictionnaire qui compte le nombre de victoires, de défaites, d'égalités et la somme totale des récompenses. Ces statistiques sont mises à jour grâce à la fonction `count_stats()` , qui ajoute +1 dans la catégorie correspondante selon le résultat de la partie et cumule le score total pour calculer une moyenne à la fin.

Pendant les tests, l'IA joue de manière purement déterministe grâce à la méthode `select_operating_action()` de la classe `State` , c'est-à-dire qu'elle choisit toujours l'action qu'elle estime la meilleure, sans aucune exploration. Cette approche permet d'évaluer objectivement la qualité de son apprentissage, sans l'influence du hasard. En revanche, le joueur aléatoire sert de référence minimale, et la stratégie du croupier donne une idée de la performance moyenne d'un joueur humain raisonnable.

Une fois toutes les parties jouées, le programme affiche un récapitulatif complet des résultats pour les trois stratégies. Il calcule les pourcentages de victoires, de défaites et d'égalités, ainsi que la récompense moyenne obtenue. Ce dernier indicateur est particulièrement important, car il reflète le gain moyen attendu à long terme.

Grâce à cette comparaison, on peut évaluer objectivement l'efficacité de l'IA. Si elle dépasse largement les deux autres méthodes, cela signifie que son apprentissage a été efficace et qu'elle a bien intégré les stratégies gagnantes du Blackjack. Ce type de test permet aussi d'ajuster le nombre d'épisodes d'entraînement nécessaires ou de vérifier si le modèle commence à surapprendre (c'est-à-dire à trop s'adapter à des cas spécifiques).

Extrait du code source :

```
def count_stats(dic,value):
    if value>0:
        dic["win"] += 1
    elif value<0:
        dic["lose"] += 1
    else:
        dic["draw"] += 1

    dic["total"] += value


def methods_comparaison():
    descovered_states = ai_saver.load(game_name)
    env = gym.make("Blackjack-v1")

    n_episodes_train = ask_number("Please choose the number of game that the ai
and methods should play")

    # plein de partie de l'ia
    ai_stats = {"win":0,"draw":0,"lose":0,"total":0}
    for _ in range(n_episodes_train):
        state, info = env.reset()
        done = False

        while not done:
            if state not in descovered_states:
                descovered_states[state] = State(state)

            node_state = descovered_states[state]

            action = node_state.select_operating_action()

            next_state, reward, terminated, truncated, info =
env.step(action.get_order())
            done = terminated or truncated

            state = next_state

        count_stats(ai_stats,reward)
    print("ai's games finished")

    # Stratégie du dealer (« hit » si <= 16 « stand » sinon)
    dealer_stats = {"win":0,"draw":0,"lose":0,"total":0}
    for _ in range(n_episodes_train):
        state, info = env.reset()
        done = False

        while not done:
            hand,a,b = state
            if hand>16:
                action = 0
            else:
                action = 1
```

```
            next_state, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated

            state = next_state

        count_stats(dealer_stats,reward)
    print("dealer's strategie games finished")

    # Choix aléatoire entre « hit » et « stand » à chaque fois
    random_stats = {"win":0,"draw":0,"lose":0,"total":0}
    for _ in range(n_episodes_train):
        state, info = env.reset()
        done = False

        while not done:

            if random.uniform(0,1) >= 0.5:
                action = 0
            else:
                action = 1

            next_state, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated

            state = next_state

        count_stats(random_stats,reward)

    # affichage des résultats et des différentes statistiques
    print(f"Based on the {n_episodes_train} games here's our results :")
    print(f"  Ai    : {round(ai_stats["win"]*100/n_episodes_train,2)} % of win; 
{round(ai_stats["draw"]*100/n_episodes_train,2)} % of draw; 
{round(ai_stats["lose"]*100/n_episodes_train,2)} % of lose; for a reward 
average of {round(ai_stats["total"]/n_episodes_train,5)}")
    print(f"Dealer : {round(dealer_stats["win"]*100/n_episodes_train,2)} % of 
win; {round(dealer_stats["draw"]*100/n_episodes_train,2)} % of draw; 
{round(dealer_stats["lose"]*100/n_episodes_train,2)} % of lose; for a reward 
average of {round(dealer_stats["total"]/n_episodes_train,5)}")
    print(f"Random : {round(random_stats["win"]*100/n_episodes_train,2)} % of 
win; {round(random_stats["draw"]*100/n_episodes_train,2)} % of draw; 
{round(random_stats["lose"]*100/n_episodes_train,2)} % of lose; for a reward 
average of {round(random_stats["total"]/n_episodes_train,5)}")

    env.close()
```

Visualisation et interaction avec l’IA

Une fois l’intelligence artificielle entraînée, il est essentiel de pouvoir observer son comportement en situation réelle et d’interagir directement avec elle pour comprendre la logique de ses décisions. C’est tout l’intérêt des parties du programme dédiées à la visualisation et à l’interprétation des choix de l’IA. L’idée est de permettre à l’utilisateur non seulement de voir comment l’IA applique ce qu’elle a appris, mais aussi de lui poser des questions comme à un véritable joueur expérimenté.

Lorsqu’on lance la phase de visualisation, l’IA précédemment sauvegardée est rechargée depuis le disque à l’aide du module de gestion des modèles. Le programme crée alors un environnement de jeu Blackjack grâce à la bibliothèque Gymnasium, configuré en mode graphique afin que le déroulement de chaque partie soit affiché à l’écran. L’utilisateur choisit combien de parties il souhaite observer, et l’IA se met à jouer automatiquement en prenant ses décisions selon la stratégie qu’elle a apprise. Chaque état rencontré pendant la partie — c’est-à-dire la somme des cartes du joueur, la carte visible du croupier et la présence éventuelle d’un as compté comme 11 — est analysé par le modèle. L’IA choisit ensuite la meilleure action possible selon ses connaissances, qu’il s’agisse de rester ou de tirer une nouvelle carte. Un léger délai entre chaque action rend la progression plus fluide et permet de suivre clairement la logique du jeu.

Ce système offre une fenêtre graphique sur le fonctionnement interne de l’intelligence artificielle. Plutôt que de se limiter à des statistiques, on peut visualiser ses choix, constater son comportement face à des situations critiques et voir comment elle adapte sa stratégie en fonction des circonstances. Il devient alors possible d’évaluer intuitivement la cohérence de son apprentissage : si elle prend les bonnes décisions, si elle évite les risques inutiles ou au contraire s’aventure trop souvent à tirer une carte supplémentaire.

En parallèle, le programme permet aussi une interaction directe avec l’IA à travers une interface textuelle où l’utilisateur peut lui demander conseil. On indique simplement les informations de la main en cours — la somme totale de ses cartes, la carte visible du croupier et la présence ou non d’un as utilisable — et l’IA répond en proposant l’action qu’elle estime la plus judicieuse. Derrière cette simplicité, elle exploite exactement la même logique que lorsqu’elle joue seule : elle identifie l’état correspondant dans sa mémoire et applique la politique optimale qu’elle a construite durant son apprentissage. Si la situation demandée n’a jamais été rencontrée, l’IA prévient qu’elle ne connaît pas la réponse, ce qui traduit bien la limite naturelle de tout apprentissage basé sur l’expérience.

Extrait du code source :

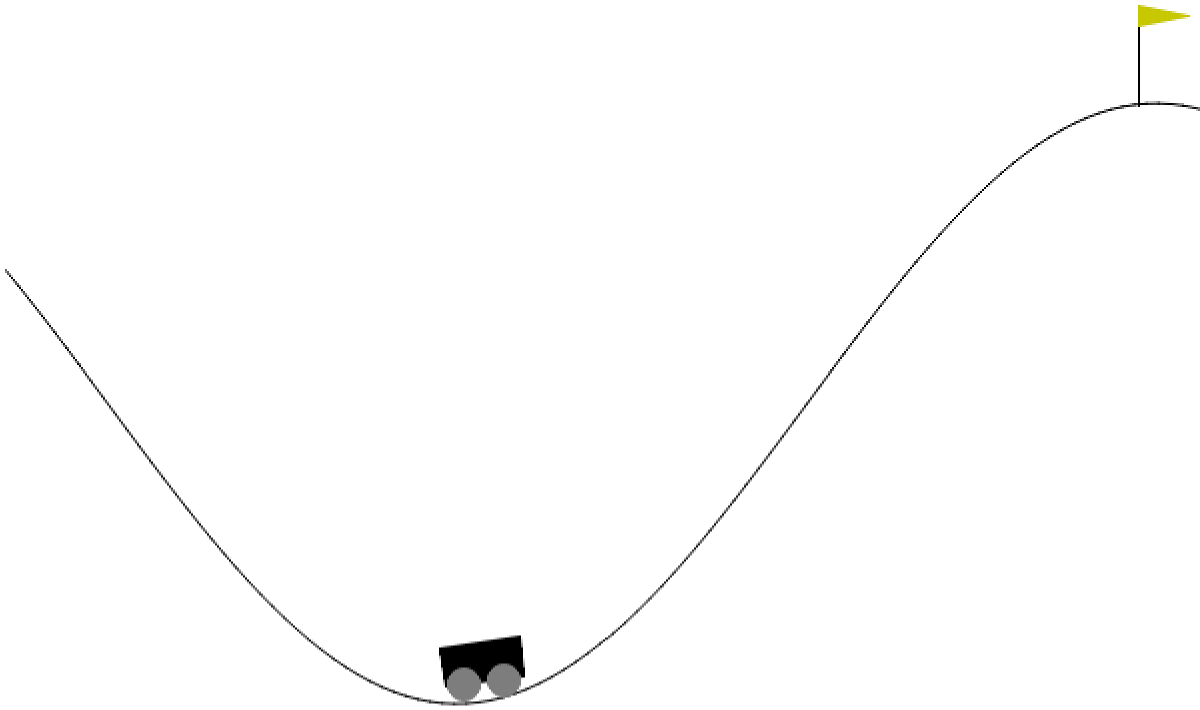
```
def blackjack_visual():
    descovered_states = ai_saver.load(game_name)
    dt = 0.7 # Délai entre deux actions pour faciliter l'observation de l'IA
(en secondes)
    env = gym.make("Blackjack-v1",render_mode="human")

    n_episodes_train = ask_number("Please choose the number of game that the ai
should play")
    for _ in range(n_episodes_train):
        state, info = env.reset()
        done = False

        while not done:
            if state not in descovered_states:
                descovered_states[state] = State(state)

            node_state = descovered_states[state]

            action = node_state.select_operating_action()

            next_state, reward, terminated, truncated, info =
env.step(action.get_order())
            done = terminated or truncated

            state = next_state
            time.sleep(dt)

    env.close()

def ai_suggestion():
    descovered_states = ai_saver.load(game_name)
    done = False
    while not done:
        choice = ask_number_in_list("select 0 to get the ai response or 1 to
exit",[0,1])
        if choice == 1:
            done = True
        else:
            player = ask_number("please enter the sum of your cards")
            dealer = ask_number("please enter dealer's cards")
            ace = ask_number("please enter 1 if you have a playable ace (count
as 11) or 0 otherwise")
            state = player,dealer,ace,
            if state not in descovered_states:
                print("The ai don't know the solution for this case")
            else:
                node_state = descovered_states[state]
                action = node_state.select_operating_action()
                print(f"Knowing that a 0 means a stay and a 1 a hit, the ai
recommands : {action.get_order()}")
```

Points clés à retenir

- **MCTS + UCB** : L'IA utilise la recherche Monte Carlo Tree Search (MCTS) avec un score UCB pour équilibrer exploration et exploitation.
- **Suivi et mise à jour des états** : Les résultats des simulations servent à mettre à jour l'arbre de recherche, guidant ainsi les décisions futures.
- **Paramètres de l'IA** : Le nombre d'itérations et le facteur UCB influencent directement la précision et le style de jeu de l'IA.
- **Sauvegarde** : L'état de l'IA peut être sauvegardé et rechargé via `pickle`, évitant ainsi de réentraîner l'IA à chaque utilisation.

Réalisation d'une IA jouant au jeu Mountain Car avec Q-Learning

Présentation du jeu Mountain Car



Le jeu Mountain Car est un environnement de type classic control disponible dans Gymnasium. Le but est simple :

- Une voiture doit atteindre le drapeau en haut de la colline droite (en prenant d'abord de l'élan sur la colline gauche).
- La voiture possède 3 choix de déplacement: accélérer vers la gauche, la droite ou ne pas accélérer.
- L’agent reçoit une récompense de -1 à chaque timestep (il n'y a aucune récompense positive).

Principe du Q-Learning

Le Q-Learning est un algorithme d’apprentissage par renforcement qui permet à un agent d’apprendre une politique optimale en interagissant avec son environnement. L’idée est d’apprendre une table de valeurs Q(s,a), où :

- Q(s,a) représente la valeur d’une action a dans un état s.
- L’agent met à jour cette table en utilisant la récompense immédiate et une estimation de la valeur future des états.

La formule de mise à jour de Q est la suivante :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

- α : taux d’apprentissage (learning rate)
- γ : facteur d’escompte (discount factor)
- r : récompense reçue après l’action
- s' : nouvel état après l’action

Étapes pour réaliser l’IA

Initialisation de l’environnement et des paramètres

On commence par importer les bibliothèques nécessaires et initialiser l'environnement :

```
from collections import defaultdict
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt

# Initialisation de l'environnement
env = gym.make("MountainCar-v0")

learning_rate = 0.1 # Taux d'apprentissage (alpha)
n_episodes = 10000 # Nombre d'épisodes
start_epsilon = 1.0 # Probabilité d'exploration (100% au début)
epsilon_decay = start_epsilon / (n_episodes / 2) # Décroissance de epsilon à chaque épisode
final_epsilon = 0.1 # Valeur minimale d'epsilon
discount_factor = 0.99 # Facteur d'escompte (gamma)
```

Discrétisation

Le problème de Mountain Car est que ses états sont continus (position et vitesse réelles). Mais le Q-Learning repose sur une table Q discrète : il faut donc convertir les valeurs continues en indices entiers.

Pour cela, on divise chaque dimension de l’espace des états en un certain nombre de bacs (bins).

Par exemple :

Si `n_bins = (20, 16)` , cela signifie :

- La **position** est découpée en 20 intervalles
- La **vitesse** est découpée en 16 intervalles
- Cela donne une table Q de taille 20 * 16 * 3 (car 3 actions possibles)

Plus les intervalles sont nombreux :

- plus l’approximation de l’espace est fine, et donc on aura une meilleure précision,
- mais la table Q devient plus grande donc l'apprentissage prend plus de temps

```
def discretize_state(self, state):
        ratios = (state - self.obs_low) / self.bin_width
        new_state = np.clip((ratios).astype(int), 0, np.array(self.n_bins) - 1)
        return tuple(new_state)
```

Boucle d’entraînement

Pour chaque épisode, l’agent interagit avec l’environnement en choisissant des actions selon une stratégie ϵ -gloutonne. À chaque étape, la table Q est mise à jour selon la formule du Q-Learning.

```
rewards_per_episode = [] # tableau pour stocker le reward de chaque épsiode
(uniquement visuel)

for episode in range(n_episodes):
    state, info = env.reset()
    state = agent.discretize_state(state) # Discrétisation de l'état initial
    done = False
    total_reward = 0

    while not done:
        action = agent.get_action(state) # Stratégie epsilon-gloutonne
        next_state, reward, terminated, truncated, info = env.step(action) #
Exécution de l'action
        next_state = agent.discretize_state(next_state) # Discrétisation

        agent.update(state, action, reward, next_state, terminated) # Mise à
jour de la table Q
        state = next_state # Passage à l'état suivant
        total_reward += reward
        done = terminated or truncated


    agent.decay_epsilon() # Décroissance de epsilon
    rewards_per_episode.append(total_reward) # Ajout du reward de cet épisode

env.close()

# Visualisation de l'apprentissage sous forme de graphe:
window = 100
moving_avg = np.convolve(rewards_per_episode, np.ones(window) / window,
mode="valid")

plt.plot(moving_avg)
plt.title("MountainCar - Moyenne des récompenses")
plt.xlabel("Épisodes")
plt.ylabel("Récompense moyenne")
plt.show()
```

Méthodes de l'agent

```
class MountainCarAgent:
    def __init__(self, env: gym.Env,
                 learning_rate: float,
                 initial_epsilon: float,
                 epsilon_decay: float,
                 final_epsilon: float,
                 discount_factor: float,
                 n_bins=(20, 16)):  # nombre d'intervalles de discrétisation
        self.env = env

        # Paramètres du Q-learning
        self.lr = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = initial_epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon
        self.training_error = []

        # Discrétisation
        self.n_bins = n_bins
        self.obs_low = env.observation_space.low # vitesse et position minimale
        self.obs_high = env.observation_space.high # vitesse et position
maximale
        self.bin_width = (self.obs_high - self.obs_low) / np.array(self.n_bins)

        # Q-table stockée sous forme de dictionnaire
        self.q_values = defaultdict(lambda: np.zeros(env.action_space.n))

    def get_action(self, state) -> int: # Stratégie epsilon-gloutonne
        if np.random.random() < self.epsilon:
            return self.env.action_space.sample() # Exploration : action
aléatoire
        else:
            return int(np.argmax(self.q_values[state])) # Exploitation : action
avec la meilleure valeur Q

    def update(self, state, action, reward, next_state, terminated): # Mise à
jour de la table Q
        future_q = (not terminated) * np.max(self.q_values[next_state])
        target = reward + self.discount_factor * future_q
        td_error = target - self.q_values[state][action]
        self.q_values[state][action] += self.lr * td_error
        self.training_error.append(td_error)

    def decay_epsilon(self): # Décroissance de epsilon
        self.epsilon = max(self.final_epsilon, self.epsilon -
self.epsilon_decay)
```

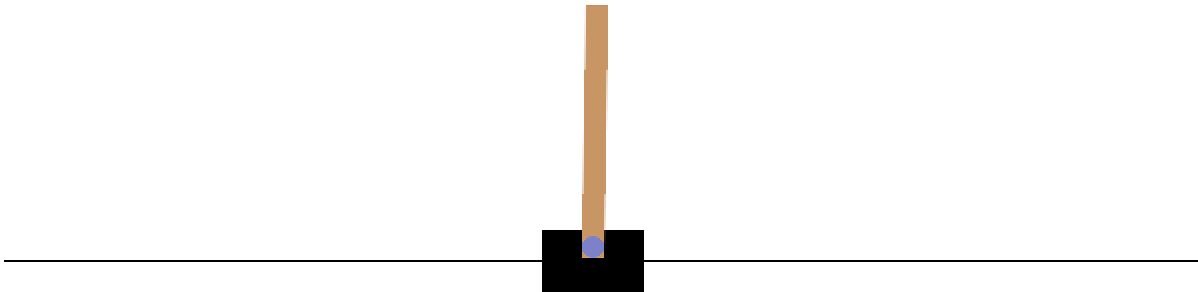
Points à retenir

- Discrétisation : on découpe l'espace continu des états pour l'adapter au Q-Learning, le paramètre `n_bins` : définit la résolution de cette discrétisation.

- Exploration/Exploitation : au début, l'agent explore beaucoup (ϵ grand), puis il exploite ce qu'il a appris.
- Récompenses : elles sont toujours négatives ce qui fait que l'agent apprend à atteindre le drapeau le plus vite possible.

Réalisation d'une IA jouant au jeu CartPole avec Q-Learning

Présentation du jeu CartPole



CartPole est un environnement de contrôle classique disponible dans **Gymnasium**.

Le but est simple :

- Un chariot peut se déplacer horizontalement.
- Un poteau est placé sur le chariot.
- L'agent doit maintenir le poteau **en équilibre** le plus longtemps possible.
- À chaque pas où le poteau tient, l'agent reçoit **+1**.
- L'épisode se termine si le poteau dépasse un certain angle ou si la limite de temps est atteinte (500 de reward).

L'objectif de l'agent est donc maximiser la durée de survie du poteau.

Principe du Q-Learning

On cherche à apprendre une fonction $Q(s, a)$ qui donne la valeur d'une action a dans un état s.

La mise à jour de la table Q est :

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

- α : taux d'apprentissage
- γ : facteur d'escompte
- r : récompense immédiate
- s' : nouvel état atteint après l'action

L'agent doit aussi résoudre le compromis **exploration/exploitation** :

- Avec probabilité ϵ, il choisit une action **aléatoire** (exploration).
- Sinon, il choisit l'action qui maximise $Q(s, a)$ (exploitation).

Discrétisation de l'espace d'état

L'environnement CartPole renvoie un état composé de **4 variables continues** :

Variable	Signification	Intervalle approximatif
cart_pos	Position du chariot	[-2.4, 2.4]
cart_vel	Vitesse du chariot	[-3.0, 3.0]
pole_angle	Angle du poteau	[-0.21, 0.21]
pole_vel	Vitesse angulaire du poteau	[-3.5, 3.5]

Comme une table Q nécessite des états **discrets**, on découpe chaque dimension en *bins* :

```
self.bins = {
    'cart_pos': np.linspace(-2.4, 2.4, 40),
    'cart_vel': np.linspace(-3.0, 3.0, 40),
    'pole_angle': np.linspace(-0.21, 0.21, 40),
    'pole_vel': np.linspace(-3.5, 3.5, 40)
}
```

La fonction de discrétisation transforme l'état continu en tuple d'indices :

```python
def discretize(self, state):
    state = np.clip(state, [-2.4, -3.0, -0.21, -3.5], [2.4, 3.0, 0.21, 3.5])
    cart_pos, cart_vel, pole_angle, pole_vel = state
    b0 = np.digitize(cart_pos, self.bins['cart_pos'])
    b1 = np.digitize(cart_vel, self.bins['cart_vel'])
    b2 = np.digitize(pole_angle, self.bins['pole_angle'])
    b3 = np.digitize(pole_vel, self.bins['pole_vel'])
    return (b0, b1, b2, b3)
```

Stratégie ϵ-gloutonne

```python
def choose_action(self, state, epsilon):
    if random.uniform(0,1) < epsilon:
        return random.randint(0, 1) # action aléatoire
    else:
        state_disc = self.discretize(state)
        return np.argmax([self.get_q(state_disc, a) for a in [0,1]])
```

Mise à jour de la Q-table

```python
def learn(self, s, a, r, s_next):
    s = self.discretize(s)
    s_next = self.discretize(s_next)
    max_q_next = max([self.get_q(s_next, a_next) for a_next in [0,1]],
default=0.0)
    old_q = self.get_q(s,a)
    new_q = old_q + self.alpha * (r + self.gamma * max_q_next - old_q)
    self.q[(s,a)] = new_q
```

Code complet d’entraînement

```
import gymnasium as gym
import random, numpy as np

class Q_learning:
    def __init__(self, alpha=0.1, gamma=0.9):
        self.q = {}
        self.alpha = alpha
        self.gamma = gamma
        self.bins = {
            'cart_pos': np.linspace(-2.4, 2.4, 40),
            'cart_vel': np.linspace(-3.0, 3.0, 40),
            'pole_angle': np.linspace(-0.21, 0.21, 40),
            'pole_vel': np.linspace(-3.5, 3.5, 40)
        }

    def get_q(self, s, a):
        return self.q.get((s,a), 0.0)

    def discretize(self, state):
        state = np.clip(state, [-2.4, -3.0, -0.21, -3.5], [2.4, 3.0, 0.21,
3.5])
        cart_pos, cart_vel, pole_angle, pole_vel = state
        return (
            np.digitize(cart_pos, self.bins['cart_pos']),
            np.digitize(cart_vel, self.bins['cart_vel']),
            np.digitize(pole_angle, self.bins['pole_angle']),
            np.digitize(pole_vel, self.bins['pole_vel'])
        )

    def choose_action(self, state, epsilon):
        if random.uniform(0,1) < epsilon:
            return random.randint(0,1)
        state_disc = self.discretize(state)
        return np.argmax([self.get_q(state_disc, a) for a in [0,1]])

    def learn(self, s, a, r, s_next):
        s = self.discretize(s)
        s_next = self.discretize(s_next)
        max_q_next = max([self.get_q(s_next, a_next) for a_next in [0,1]])
        old_q = self.get_q(s,a)
        self.q[(s,a)] = old_q + self.alpha * (r + self.gamma * max_q_next -
old_q)


env = gym.make("CartPole-v1")
AI = Q_learning()

epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.9995
n_episodes = 10000

for episode in range(n_episodes):
    state, info = env.reset()
```

```
        done = False
        total_reward = 0

        while not done:
            action = AI.choose_action(state, epsilon)
            next_state, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated
            AI.learn(state, action, reward, next_state)
            state = next_state
            total_reward += reward

        if epsilon > epsilon_min:
            epsilon *= epsilon_decay
```

Points à retenir

- Les **états continus sont discrétisés** pour permettre une table Q.
- ϵ commence élevé pour **explorer**, puis diminue pour **exploiter**.
- La récompense étant positive, l'agent apprend à **maintenir l'équilibre le plus longtemps possible**.