TIPE : De la programmation dynamique à l'apprentissage par renforcement

Abel Verley

2021-2022

1 Motivations

On appelle régime de traitements dynamiques (DTR) une séquence d'instructions qui détermine les étapes d'un traitement personnalisé tenant compte de l'historique médical d'un patient. Ce paradigme de médecine personnalisée peut s'avérer utile pour le traitement de maladies chroniques (Diabète, Asthme, Parkinson...), caractérisées par leur implantation dans la durée ainsi que leur propension à engendrer des complications et des invalidités.

Déterminer un DTR constitue un problème de décision séquentiel dans l'incertain :
- Séquentiel : Chaque étape du traitement correspond à une décision à prendre
- Incertain : Les réactions du patient sont incertaines, il s'agit d'un environnement aléatoire

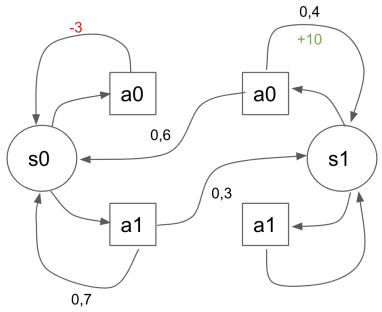
2 Formalisation : Processus de décision Markovien

Pour étudier le problème posé, on introduit la structure suivante

Définition 2.1. On appelle processus de décision markovien (MDP) un quadruplet (S, A, p, r) où :
- S est un ensemble fini d'états
- A est un ensemble fini d'actions
- p est une fonction de transition stochastique : $\forall s, s' \in S\ a \in A, p(s'|a, s)$ représente la probabilité d'être dans l'état s' en ayant pris l'action a depuis l'état s
- r est la fonction de récompense : $\forall s, s' \in S\ a \in A, r(s, a, s')$ représente une récompense générée lors de la transition de s à s' passant par l'action a

Remarques :
- Si $r(s, a, s') < 0$ on parle de pénalité
- $\forall s \in S, \sum_{s' \in S} p(s'|a, s) = 1$

Exemple 2.1. MDP à deux états et deux actions



Pour l'implémentation informatique notamment, peut résumer les informations des fonctions p et r dans des matrices :

— $\forall a \in A, [P_a]_{s,s'} = p(s'|a,s)$:

— $\forall a \in A, [R_a]_{s,s'} = r(s,a,s')$

Définition 2.2. Une politique est une application $\pi : S \longrightarrow A$. On note $\mathcal{D}$ l'ensemble des politiques.

Les politiques correspondent donc aux décisions prises au cours du traitement.

Définition 2.3. Soit π une politique. On définit la matrice $P_\pi \in \mathcal{M}_{|S|}(\mathbb{R})$ telle que $\forall s, s' \in S,\ [P_\pi]_{s,s'} = p(s'|\pi(s), s)$. De même on construit la matrice r_π telle que $[r_\pi]_s = \sum_{s' \in S} p(s'|s, \pi(s)) r(s, \pi(s), s')$

On remarque que cette matrice P_π constitue la matrice d'adjacence d'une chaîne de Markov.

On a besoin d'un moyen d'évaluer les politiques pour les comparer.

Définition 2.4. Soit $\gamma \in]0,1[,\ \pi$ une politique. On définit la fonction de valeur :

$$V_\gamma^\pi : s \in S \mapsto \mathbb{E}^\pi[\sum_{t=0}^{+\infty} \gamma^t r(s_t, a_t, s_{t+1}) | s_0 = s]$$

Il s'agit de la somme des récompenses qu'on peut espérer en suivant la politique π depuis un état initial s.

Le facteur γ a pour intérêt, r étant bornée, de faire converger la somme, mais aussi de modifier l'importance accordée aux récompenses à long terme.

L'objectif est donc de trouver une politique qui maximise la fonction valeur qui lui est associée.

Définition 2.5. Une politique, usuellement notée π^* est dite optimale si

$$\forall \pi \in \mathcal{D}, V^* = V_\gamma^{\pi^*} \geq V_\gamma^\pi$$

On souhaite calculer algorithmiquement π^* et V^*.

3 Un algorithme de programmation dynamique

Pour introduire une première méthode algorithmique, on s'intéresse à certaines propriétés.

Posons $E = \mathbb{R}^S$. $(E, ||.||_\infty)$ est un espace vectoriel normé. On associe naturellement à $V \in E$ un vecteur, et on identifie la fonction et le vecteur.

Définition 3.1. Soit π une politique, $0 < \gamma < 1$. Dans E on pose L_π l'opérateur tel que $\forall V \in E, L_\pi.V = r_\pi + \gamma P_\pi V$

Lemme 3.1. *Soit π une politique, $0 < \gamma < 1$. V_γ^π est l'unique solution de l'équation $V = L_\pi.V$*

On en déduit le théorème suivant qui permet de déduire π^* de V^* :

Théorème 3.2. *Si π est une politique telle que $\pi \in argmax_{\pi' \in \mathcal{D}} r_{\pi'} + \gamma P_{\pi'} V^*$, alors π est optimale.*

Par ailleurs on dispose d'une caractérisation de la fonction V^* :

Définition 3.2. On définit sur E l'opérateur de programmation dynamique $L : \forall V \in E, L.V = max_{\pi \in \mathcal{D}}(r_\pi + \gamma P_\pi V)$

Théorème 3.3 (Équation de Bellman)**.** *Soit $\gamma \in]0, 1[$ alors , V^* est l'unique solution de l'équation $V = L.V$ ie*

$$\forall s \in S,\ V(s) = max_{a \in A} \sum_{s' \in S} p(s'|a, s)(r(s, a, s') + \gamma V(s'))$$

On en déduit une méthode itérative de calculer V^* :

$$\forall s \in S.\ V_{n+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} p(s'|a, s)(r(s, a, s') + \gamma V_n(s'))$$

D'où l'algorthme d'itération des valeurs. :

Algorithm 1 Algorithme d'itération des valeurs

```
n ← 0
V_n ← 0
repeat
    V_{n+1} ← V_n
    for s ∈ S do
        V_{n+1}(s) ← max_{a∈A} Σ_{s'∈S} p(s'|a, s)(r(s, a, s') + γV_n(s'))
    end for
    n ← n + 1
until ||V_{n+1} − V_n||_∞ < ε
for s ∈ S do
    π(s) ∈ argmax_{a∈A} Σ_{s'∈S} p(s'|a, s)(r(s, a, s') + γV_n(s'))
end for
return V, π
```

Pour étudier la convergence de l'algorithme on s'interesse à une preuve partielle de l'équation de Bellman :

Théorème 3.4 (Théorème de point fixe de Banach)**.** *Soit E un espace de Banach, f λ-lipschitzienne avec $0 \leq \lambda < 1$ alors il existe une unique solution u de l'équation $f(x) = x$ et de plus, les suites définies par $u_0 \in E$ et $\forall n \in \mathbb{N},\ u_{n+1} = f(u_n)$ convergent vers u.*

Démonstration. On construit une suite (u_n) comme ci-dessus. (u_n) converge si et seulement si $\sum(u_{n+1}-u_n)$ converge. Or $\forall n \geq 1,\ |u_{n+1}-u_n| = |f(u_n)-f(u_{n-1})| \leq \lambda|u_n-u_{n-1}|$ on montre alors par récurrence que $\forall n \in \mathbb{N}, |u_{n+1}-u_n| \leq \lambda^n|u_1-u_0|$ qui est le terme général d'une série convergente. Donc $\sum(u_{n+1}-u_n)$ converge absolument ce qui implique, par completude, la convergence de la série et donc de la suite (u_n) vers un vecteur u. Par ailleurs, f est lipschitzienne donc continue et par passage à la limite dans la définition récurrente de la suite : $u = f(u)$
Supposons avoir deux points fixes u, u' alors $|u-u'| = |f(u)-f(u')| \leq \lambda|u-u'|$. Et comme $\lambda < 1$ ceci n'est possible que si $u = u'$. D'où l'unicité. □

De plus :

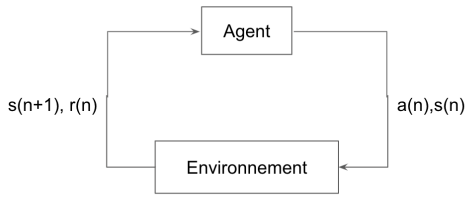
Théorème 3.5. *L'opérateur L est une contraction sur E*

On a ainsi la convergence de la méthode itérative. On admet que sa limite est V^*.

Complexité : Chaque itération de l'algorithme s'effectue en $O(|S|^2|A|)$. Si p et r sont codés sur b bits, il est établi qu'il suffit d'un nombre d'itération polynomial en $|S|, |A|, b, \frac{1}{(1-\gamma)\log(1/(1-\gamma))}$ pour avoir une politique optimale en sortie.

4 Apprentissage par renforcement

On souhaite trouver une méthode s'appliquant des les cas où p et r sont inconnues. Ces cas sont en réalité plus applicables.
On dispose d'un agent capable de prendre des actions dans un environnement et d'observer les états et récompenses engendrés par sa prise d'action.



Initialement, l'agent opère de manière aléatoire. On cherche à estimer la fonction de valeur optimale à l'aide des récompenses observées. A partir de cette estimation, l'agent adaptera son comportement pour optimiser la récompense.

L'approximation de la fonction valeur s'effectue sur une séquence d'épisode de longueur finie. On définit un état s_0 initial à chaque épisode et un état s_f déterminant la fin d'un épisode.

Le théorème 3.2 permet de déduire π^* de V^* sous réserve de connaître p et r. On introduit alors la fonction de valeur Q suivante.

Définition 4.1. Pour une politique π, on définit la fonction Q^π telle que :

$$\forall s \in S, a \in A, Q^\pi(s,a) = \mathbb{E}^\pi[\sum_{t=0}^{+\infty} \gamma^t r(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a]$$

Il s'agit de la somme des récompenses qu'on peut espérer en suivant la politique π depuis un état initial s, en ayant pris une action initiale a.

Théorème 4.1. *En notant Q^* pour la politique optimale, on a $\forall s \in S$:*
- $V^*(s) = max_{a \in A} Q^*(s,a)$
- $\pi^*(s) \in \ argmax_{a \in A} Q^*(s,a)$

A chaque itération, le choix de l'action est orienté par un dilemme entre exploiter ou explorer :
- Exploiter : C'est choisir l'action qui, selon l'approximation de la fonction valeur, maximise la récompense. Toujours suivre cette stratégie conduit à tomber sur un maximum local.
- Explorer : C'est découvrir aléatoirement des paires de $S \times A$ pour augmenter la connaissance de l'environnement quitte à engendrer des pénalités inutiles.

Pour répondre à ce dilemme :
Stratégie ϵ-greedy :
- Avec une probabilité ϵ : l'action est choisie au hasard
- Avec une probabilité $1-\epsilon$: l'action est choisie de manière "gloutonne" : $a \in \text{argmax}_{a \in A} Q(s,a)$

Une fois l'action a_n choisie, l'agent observe s_{n+1} et r_n, on cherche alors à améliorer l'approximation des fonctions de valeur.

On remarque que les fonctions de valeurs vérifient :

$$V(s_n) = \overline{r}(s_n, a_n, s_{n+1}) + \gamma V(s_{n+1})$$
$$Q(s_n, a_n) = \overline{r}(s_n, a_n, s_{n+1}) + \gamma Q(s_{n+1}, a_{n+1})$$

On pose alors :

$$\delta_n^V = r_n + \gamma V(s_{n+1}) - V(s_n) \approx 0$$
$$\delta_n^Q = r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n) \approx 0$$

Ces termes δ permettent d'estimer l'erreur faite sur l'approximation des fonctions valeurs. Pour $\alpha_n \in]0,1[$, un taux d'apprentissage, on actualise localement l'estimation des fonctions valeur des manières suivantes, pour trois algorithmes similaires .
Algorithme TD(0) :

$$V(s_n) \leftarrow V(s_n) + \alpha_n \delta_n^V$$

Algorithme SARSA :

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha_n \delta_n^Q$$

Algorithme Q-learning :

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha_n [r_t + \gamma \text{max}_{a \in A} Q(s_{n+1}, a) - Q(s_n, a_n)]$$

Dans l'algorithme de Q-learning, l'actualisation de l'estimation de la fonction Q est rendue indépendantes de la politique d'exploration.

Algorithm 2 Algorithme de Q-Learning

for épisodes 1 à N **do**
  $n \leftarrow 0$
  $s_n \leftarrow s_0$
  while $s_n \neq s_f$ **do**
    $\tilde{Q} \leftarrow Q$
    if random[0,1] $< \epsilon$ **then**
      a_n est choisi aléatoirement
    else
      $a_n \leftarrow \text{argmax}_{a \in A} Q(s, a)$
    end if
    observer r_n et s_{n+1} d'après s_n et a_n
    $\tilde{Q}(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha[r_n + \gamma \text{max}_{a \in A} Q(s_{n+1}, a) - Q(s_n, a_n)]$
    $n \leftarrow n + 1$
  end while
end for
Return Q

On admet le théorème de correction du Q-learning :

Théorème 4.2. *La convergence de l'algorithme de Q-learning vers Q^* est presque sûre si*

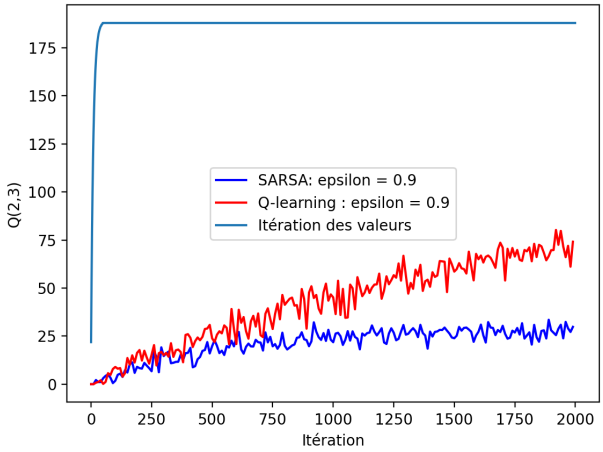
— $$\sum_{i=0}^{+\infty} \alpha_n = +\infty$$

— $$\sum_{i=0}^{+\infty} \alpha_n^2 < +\infty$$

En pratique on choisit souvent α_n constant, ce qui donne souvent une solution presque optimale. Dans l'annexe 1 on constate une amélioration par la méthode softmax.

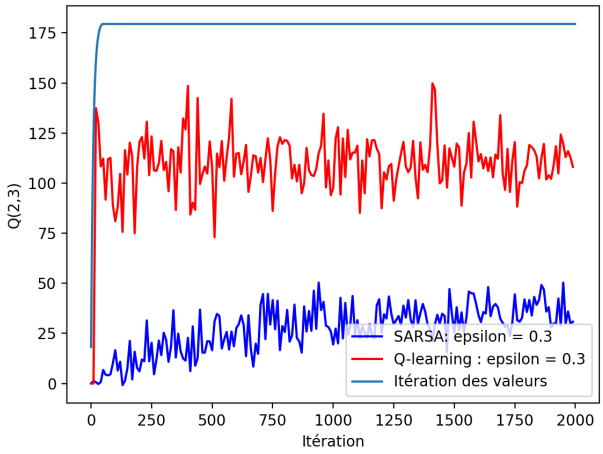
5 Annexe 1 : Résultats graphiques

On implémente les différents algorithmes sur un environnements modélisé par un MDP à 10 états et 5 actions généré aléatoirement. L'évolution de Q(2,3) en fonction du nombre d'itération donne la courbe suivante :

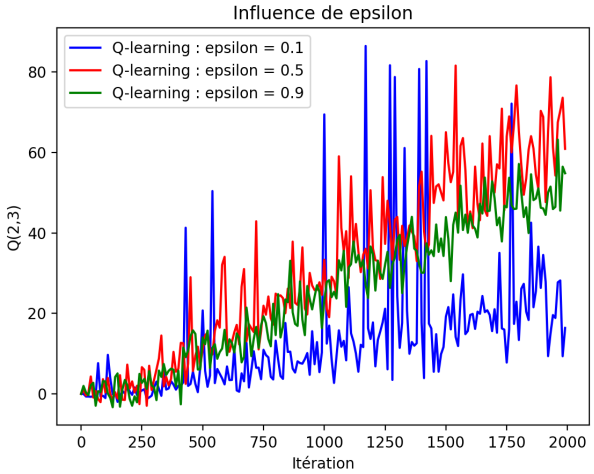


On constate que la différence entre les valeurs de convergence des algorithmes d'apprentissage par renforcement et la valeur optimale demeure importante.

L'implémentation de la méthode softmax qui consiste à choisir un taux d'apprentissage $\alpha_n = \frac{1}{n_{a,s}}$ où $n_{a,s}$ représente le nombre de fois où l'action a a été prise depuis l'état s.



On voit que l'implémentation de la méthode softmax sur l'algorithme de Q-learning permet d'améliorer la valeur de convergence.
Par ailleurs on constate que le choix de ϵ dans la stratégie ϵ-greedy a aussi son importance :



En effet, augmenter la probabilité d'exploration permet de ne pas tomber sur un maximum local.

6 Annexe 2 : Démonstrations

Lemme 3.1. Soit π une politique, $0 < \gamma < 1$. V_γ^π est l'unique solution de l'équation $V = L_\pi.V$

Démonstration. Soit V une solution de l'équation. On a $(I - \gamma P_\pi)V = r_\pi$. La matrice P_π étant stochastique, on montre que $|||P_\pi||| \leq 1$ et on en déduit par sous-multiplicité de $|||.|||$ que $\sum(\gamma^k P_\pi^k)$ converge absolument. De plus par télescopage, $(I - \gamma P_\pi)(\sum_{k=0}^{+\infty} \gamma^k P_\pi^k) = I$ donc $(I - \gamma P_\pi)^{-1} = \sum_{k=0}^{+\infty} \gamma^k P_\pi^k$. Ainsi $V = \sum_{k=0}^{+\infty} \gamma^k P_\pi^k r_\pi$ Par ailleurs, pour $s \in S$

$$\begin{aligned} V_\gamma^\pi(s) &= \sum_{t=0}^{+\infty} \gamma^t \mathbb{E}^\pi[r(s_t, a_t, s_{t+1})|s_0 = s] \\ &= \sum_{t=0}^{+\infty} \gamma^t \sum_{s' \in S} \mathbb{P}^\pi(s_t = s'|s_0 = s) \sum_{s'' \in S} p(s''|s', \pi(s'))r(s', \pi(s), s'') \end{aligned}$$

La dernière ligne provient de la définition de l'espérance conditionnelle ainsi que de la formule de transfert. Or P_π s'interprète comme la matrice d'adjacence d'une chaîne de Markov, on montre alors que $\mathbb{P}^\pi(s_t = s'|s_0 = s) = P^t_{\pi,s,s'}$ et donc pour $s \in S$

$$\begin{aligned} V_\gamma^\pi(s) &= \sum_{t=0}^{+\infty} \gamma^t \sum_{s' \in S} P^t_{\pi,s,s'} r_\pi(s') \\ &= \sum_{k=0}^{+\infty} \gamma^k P_\pi^k r_\pi(s) \end{aligned}$$

Et donc $V = V_\gamma^\pi$. L'unicité est immédiate. □

Théorème 3.2. Si π est une politique telle que $\pi \in \text{argmax}_{\pi' \in \mathcal{D}} r_{\pi'} + \gamma P_{\pi'} V^*$, alors π est optimale.

Démonstration. Soit π telle que $\pi \in \text{argmax}_{\pi' \in \mathcal{D}} r_{\pi'} + \gamma P_{\pi'} V^*$. On a alors

$$\begin{aligned} L_\pi V^* &= r_\pi + \gamma P_\pi V^* \\ &= \max_{\pi' \in \mathcal{D}} r_{\pi'} + \gamma P_{\pi'} V^* \\ &= LV^* \\ &= V^* \end{aligned}$$

cf.Équation de Bellman. On conclut par le lemme 3.1. □

Théorème 3.5 L'opérateur L est une contraction sur E

Démonstration. Soient $U, V \in E$ et $s \in S$. Posons $a_s^* \in \text{argmax}_{a \in A} \sum_{s' \in S} p(s'|a, s)(r(s, a, s' +$

$\gamma V(s')$). Par symétrie supposons, $L.V(s) \geq L.U(s)$. Alors

$$\begin{aligned}
|L.V(s) - L.U(s)| &= L.V(s) - L.U(s) \\
&\leq \sum_{s' \in S} p(s'|a_s^*, s)(r(s, a_s^*, s') + \gamma V(s')) - \sum_{s' \in S} p(s'|a_s^*, s)(r(s, a_s^*, s' + \gamma U(s')) \\
&\leq \gamma \sum_{s' \in S} p(s'|a_s^*, s')(V(s') - U(s')) \\
&\leq \gamma \sum_{s' \in S} p(s'|a_s^*, s')||V - U|| \\
&\leq \gamma ||U - V||
\end{aligned}$$

□

7 Annexe 3 : code

7.1 MDP_ens.py

```python
import numpy as np
import matplotlib.pyplot as plt
import random

#Implementation des processus de decision markovien (MDP) stationnaire

class Proba():
    """Les probabilites du MDP sont representees par la liste des |A| matrices P_a
    """

    def __init__(self, list_proba )->None:
        self.P= list_proba

class Reward ():
    """Les recompenses sont representees par la liste des |A| matrices r_a"""

    def __init__(self,list_rewards)->None:
        self.R = list_rewards


class MDP ():
    """Les ensembles S (resp.A) sont caracterises par les entiers |S| (resp.|A|) """

    def __init__(self,s:int,a:int,p:Proba,r:Reward) -> None:

        assert len(p.P)== a and len(r.R)==a
        self.S = s
        self.A = a
        self.P = p.P
        self.R = r.R


    def value_iteration(self,gamma:float, eps:float)->tuple:
        """Algorithme d'iteration sur les valeurs. Les fonction valeur et les politique sont
        representees par des vecteurs de taille |S|"""

        V= np.array([0]*self.S)
        while True:
            newV = np.copy(V)
            for s in range(self.S):
                newV[s]=max([sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for s2 in range(self.S)]) for a in range(self.A)])
            if np.linalg.norm(newV-V)<=eps:
                break
            V = newV
        pi = np.array([0]*self.S)
        for s in range(self.S):
            f = lambda a :  sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for s2 in range(self.S)])
            pi[s]=argmax(f,self.A)


```

```
        return V, pi



    def graph_Q(self,s:int,a:int,gamma:float, N)->None:
        """Pour tracer l'evolution de Q(s,a) au fil des it rations """

        V= np.array([0]*self.S)
        Q = []

        for i in range(N):
            newV = np.copy(V)
            for s in range(self.S):
                newV[s]=max([sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for
    s2 in range(self.S)]) for a in range(self.A)])
            V = newV
            Q.append(sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for s2 in
    range(self.S)]))

        T = np.array(range(N))
        Q = np.array(Q)


        plt.plot(T,Q,label = 'It ration des valeurs')



    def policy_iteration(self,gamma:float, eps:float)->tuple:
        #Algotithme d'iteration sur politiques

        pi = np.array([0]*self.S)
        V=np.array([0]*self.S)

        while True:
            newpi = np.copy(pi)

            #prediction:

            while True:
                newV = np.copy(V)
                for s in range(self.S):
                    newV[s]= sum([self.P[pi[s]][s,s2]*(self.R[pi[s]][s,s2]+gamma*V[
    s2])  for s2 in range(self.S)])
                if np.linalg.norm(newV-V)<=eps:
                    break
                V = newV

            #controle:

            for s in range(self.S):
                f = lambda a :  sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2])
    for s2 in range(self.S)])
                pi[s]=argmax(f,self.A)
            if np.linalg.norm(newpi-pi)==0:
                break
            pi = newpi
```

```
104
105         return V , pi
106
107
108
109 def argmax(f, A:int)->int:
110     """D termine un    lment    de argmax f"""
111
112     a = 0
113     arg = f(a)
114     for a2 in range(A):
115         arg2 = f(a2)
116         if arg2>arg:
117             a = a2
118             arg = arg2
119     return a
120
121 def make_random_MDP(S:int,A:int,Rmax:float)-> MDP:
122     """Genere un MDP aleatoire"""
123
124     list_probas = []
125     list_rewards = []
126     for a in range (A):
127         Pa = np.random.rand(S, S)
128         Pa = Pa/Pa.sum(axis=1)[:,None]
129         list_probas.append(Pa)
130         Ra=Rmax*(np.random.rand(S,S))
131         for s in range(S):
132             for s2 in range(S):
133                 if random.random()<0.5:
134                     Ra[s][s2]= -Ra[s][s2]
135         list_rewards.append(Ra)
136     P = Proba(list_probas)
137     R = Reward(list_rewards)
138     return MDP(S,A,P,R)
```

7.2 ApprentissageRenforcement_ens.py

```
1 import numpy as np
2 from MDP_ens import *
3 import random
4 import matplotlib.pyplot as plt
5
6 class Agent():
7
8     def __init__(self,s0:int, env:MDP)-> None:
9
10         self.state = s0
11         self.env = env
12
13
14     def SARSA (self, s0:int, sf:int, eps:float, gamma:float, N:int, lr = 0.01)->np.
       ndarray:
15         """s0 est l'etat initial, sf est l' tat  final absorbant, epsilon induit la
       probabilit  d'exploration
16         (eps-greedy), N est le nombre d'episodes """
17
18
```

```
        Q = np.array([[0]*self.env.A]*self.env.S, dtype= np.float32)

        for i in range(N):
            self.state = s0
            if random.random() < eps:
                a = random.randint(0,self.env.A-1)
            else:
                a = 0
                Qa = Q[self.state][0]
                for b in range (self.env.A):
                    Q2 = Q[self.state][b]
                    if Q2>Qa:
                        a = b
                        Qa = Q2

            while self.state != sf:
                s2 = etat_suivant(self.state,a,self.env)
                if random.random() < eps:
                    a2 = random.randint(0,self.env.A-1)
                else:
                    a2 = 0
                    Qa = Q[self.state][0]
                    for b in range (self.env.A):
                        Q2 = Q[self.state][b]
                        if Q2>Qa:
                            a2 = b
                            Qa=Q2
                r = self.env.R[a][self.state,s2]

                newQ = np.copy(Q)
                newQ[self.state,a]=Q[self.state,a]+lr*(r+gamma*Q[s2,a2] -Q[self.
    state,a])

                a = a2
                self.state = s2
                Q = newQ


        return Q


    def Q_learning (self, s0:int, sf:int, eps:float, gamma:float, N:int, lr= 0.01)->
    np.ndarray:
        """s0 est l'etat initial, sf est l'etat final absorbant, epsilon induit la
    probabilit  d'exploration
        (eps-greedy), N est le nombre d'episodes """

        self.state = s0
        Q = np.array([[0]*self.env.A]*self.env.S, dtype=np.float32)

        # En commentaire: implementation softmax

        #A = np.array([[0]*self.env.A]*self.env.S)
        #A stocke les taux d'apprentissage selon la methode uncertainty estimation
        for i in range(N):
            self.state = s0
            while self.state != sf:
                if random.random() < eps:
```

```
                    a = random.randint(0,self.env.A-1)
                else:
                    a = 0
                    Qa = Q[self.state][0]
                    for a2 in range (self.env.A):
                        Q2 = Q[self.state][a2]
                        if Q2>Qa:
                            a = a2
                            Qa=Q2
                #A[self.state,a]+=1
                s2 = etat_suivant(self.state,a,self.env)
                r= self.env.R[a][self.state,s2]
                newQ = np.copy(Q)
                newQ[self.state,a]=Q[self.state,a]+(lr)*(r+gamma*max([Q[s2][a2] for a2 in range(self.env.A)])-Q[self.state][a])
                #newQ[self.state,a]=Q[self.state,a]+(1/A[self.state,a])*(r+gamma*max([Q[s2][a2] for a2 in range(self.env.A)])-Q[self.state][a])
                self.state = s2

                Q = newQ

        return Q


    def graph_Q(self, s0:int, sf:int, eps:float, gamma:float,s2: int, a2:int, M:int, pas = 10)->None:
        """Trace Q(a,s) en fonction du nombre d'episodes"""

        #il faut que l'etat final soit associ     une recompense nulle
        for a in range(self.env.A):
            for s in range(self.env.S):
                (self.env.R)[a][s,sf]=0

        Q_sa = []
        for i in range(0,M,pas):
            Q = self.Q_learning(s0,sf,eps,gamma,i)
            Q_sa.append(Q[s2,a2])
            print(i)

        T = np.array(range(0,M,pas))
        Q_sa = np.array(Q_sa)
        plt.plot(T,Q_sa, color ='red', label = 'Q-learning : epsilon = '+ str(eps))


    def graph_SARSA(self, s0:int, sf:int, eps:float, gamma:float,s2: int, a2:int, M:int, pas=10)->None:
        """Trace Q(a,s) en fonction du nombre d'episodes"""

        #il faut que l' tat  final soit associe    une recompense nulle
        for a in range(self.env.A):
            for s in range(self.env.S):
                (self.env.R)[a][s,sf]=0

        Q_sa = []
        for i in range(0,M,pas):
            Q = self.SARSA(s0,sf,eps,gamma,i)
            Q_sa.append(Q[s2,a2])
            print(i)
```

```
        T = np.array(range(0,M,pas))
        Q_sa = np.array(Q_sa)
        plt.plot(T,Q_sa, color = 'blue', label = 'SARSA: epsilon = '+ str(eps))




def etat_suivant(s:int,a:int, env:MDP)->int:
    """Determine l'etat suivant lorsqu'on prend l'action a depuis l'etat s"""

    p = random.random()
    list_probas = [env.P[a][s,s2] for s2 in range(env.S)]
    for i in range(env.S):
        if p<list_probas[i]:
            return i
        else:
            p-=list_probas[i]



if __name__ == '__main__':

    S = 10
    A = 5
    Rmax = 100
    G = make_random_MDP(S,A,Rmax)
    agent = Agent(0,G)

    si = 0
    sf = 9
    eps = 0.3
    gamma = 0.9
    s = 2
    a = 3
    iter  = 2000


    agent.graph_SARSA(si,sf,eps,gamma,s,a,iter)
    agent.graph_Q(si,sf,eps,gamma,s,a,iter )
    agent.env.graph_Q(s,a,gamma,iter)
    plt.xlabel('It ration')
    plt.ylabel('Q(2,3)')
    plt.legend()

    #agent.graph_Q(si,sf,0.1,gamma,s,a,iter)
    #agent.graph_Q(si,sf,0.5,gamma,s,a,iter)
    #agent.graph_Q(si,sf,0.9,gamma,s,a,iter)
    #plt.xlabel('It ration')
    #plt.ylabel('Q(2,3)')
    #plt.legend()
    #plt.title(label = 'Influence de epsilon')
    plt.show()
```

8 Bibliographie

1. Groupe PDMIA, Processus Décisionnels de Markov en Intelligence Artificielle (2008) (Chapitres 1 et 2) [ISBN : 9782746220577]
2. Chris Watkins : Q-learning : Machine Learning, 8 (1992), 279-292
3. DeepMind x UCL Deep learning lecture Series 2021 (épisodes 1,2,3) (https ://www.youtube.com/playlist ?list=PLqYmG7hTraZDVH599EItlEWsUOsJbAodm)