Rapport exercices lecture2 Reinforcement Learning : Finite and Infinite horizon MDP

Lahmouz Zakaria Working with Oussama Karmaoui

October 2023

1 Finite Horizon MDP : Exercice 1 :

En exploitant l'équation d'optimalité sur un processus marcovien fini , l'expression de la fonction valeur maximale (ou optimale) à l'étape T pour un état s s'écrit :

$$V_T(s) = \max_a (r(s,a) + \sum_{s'} p(s'|s,a) V_{T-1}(s')])$$

sous les conditions initiales :

$$V_0(s) = 0 \, \text{pour} \;\; \text{tout} \quad s \; etat$$

$$V_T(0) = 0 \, \text{pour} \;\; \text{tout} \;\; T \; temps$$

Ainsi:

$$V_T(s) = \max_a (0.5+0.1{\cdot}V_{T-1}(s-1)+0.9{\cdot}V_{T-1}(s), 0.8+0.8{\cdot}V_{T-1}(s-1)+0.2{\cdot}V_{T-1}(s))$$

1.1 Implémentation du code de Vt(s) avec la décision optimale à l'instant t (en python) :

```python
def v_p(t, s, memo={}):
    if (t, s) in memo:
        return memo[(t, s)]

    if t == 0 or s == 0:
        return (0, 2)
    else:
        tup = (0.5 + 0.1 * v_p(t - 1, s - 1, memo)[0] + 0.9 * v_p(t - 1, s, memo)[0],
               0.8 + 0.8 * v_p(t - 1, s - 1, memo)[0] + 0.2 * v_p(t - 1, s, memo)[0])
        maxvalue = max(tup)
        action = tup.index(maxvalue) + 1
        memo[(t, s)] = (maxvalue, action)
        return memo[(t, s)]
```

Figure 1: Code de calcul de Vt(s) avec la décision optimale a_t

Application numérique (avec π_t la politique optimale à l'instant t) : :

$$V_3(20) = 2.4\, et\, \pi_3(2|s = 20) = 1$$

$$V_20(7) = 11.99\, et\, \pi_20(1|s = 7) = 1$$

Affichage de la politique optimale :
On souhaite représenter la politique optimale en fonction de l'état s et de l'instant t (le nombre de sièges varie de 0 à 20 et le temps restant est compris entre 0 et 50) sous forme de régions colorées en utilisant la fonction imshow de la librairie Matplotlib. Ainsi on obtient la figure 3 :

```
# Créer une grille de temps et d'états
temps = np.arange(51)  # Par exemple, 51 instants de temps (0 à 5)
etats = np.arange(21)  # Par exemple, 21 états (0 à 20)

plt.figure()

policy_matrix = np.zeros((21, 51))
for t in temps:
    for s in etats:
        policy_matrix[s,t] = v_p(t,s)[1]


# Définissez un colormap pour les actions (à adapter à vos besoins)
cmap = plt.get_cmap( name: "coolwarm",  lut: 2)  # Utilisez un colormap avec 4 couleurs (une pour chaque action)

# Affichez la matrice de politique avec imshow
plt.imshow(policy_matrix, cmap=cmap, aspect='auto', origin='lower')

# Ajoutez une légende pour expliquer les actions associées aux couleurs
actions = ['p1=5','p2=1']
cbar = plt.colorbar(ticks=[1,2])
cbar.set_ticklabels(actions)

# Affichez le graphe
plt.title("Politique optimale")
plt.xlabel("Instant t")
plt.ylabel("État s")
plt.show()
```

Figure 2: Code d'affichage de politique optimale

Et on obtient la figure suivant :

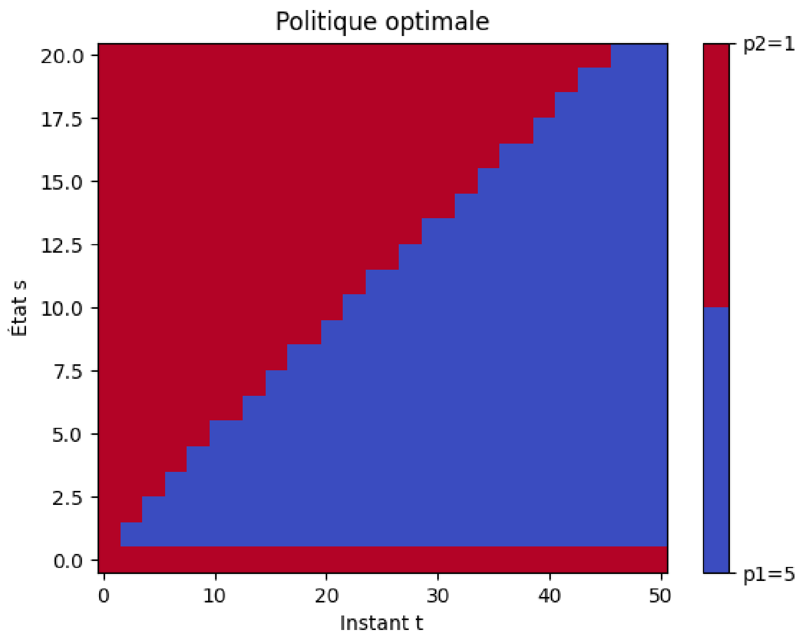

Figure 3: Représentation de la politique optimale en 2 régions colorées en fonction de t et s

Analyse :

1. Pour t fixé , si s diminue ,alors la décision favorable pour augmenter la récomponse tend vers l'action 1 (prix p1=5) , ce qui est tout à fait normal parceque'on voudrait augmenter le prix des sièges si le nombre de sièges restant diminue en une durée de temps t.

2. En fixant l'état s , d'après le graphe , la décision optimale tend vers la décision 1 (p1=5) avec l'augmentation de t (si t représente les jours ou heures, unité de temps..). Ce qui explique par exemple la croissance des prix des vols ou siège à l'approche du temps de vol.

2 Exercice2:Infinite Horizon MDP

Pour un processus marcovien infini l"équation d'optimalité de Bellman est représentée :

$$V^*(s) = \max_{a \in A} \left(r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right) \tag{1}$$

Pour résoudre cette équation , on proposera 2 méthodes :
Value Iteration : En prenant une valeur arbitraire de V* pour tous les états s (état 0 et état 1) , l'algorithme de value iteration sera :

```
eps = 0.05
3 usages
def v_iteration(gamma,v,iter) :
    max_i1, max_v1  = max(enumerate((1 + gamma*v[0], 10 + gamma*v[1])), key=lambda x: x[1])
    max_i2, max_v2  = max(enumerate((gamma * v[1], -15 + gamma * v[0])), key=lambda x: x[1])
    nv_v = np.array([max_v1,max_v2])
    nv_a = [max_i1+1,max_i2+1]
    if np.linalg.norm(v - nv_v) <= eps or iter == 1000 :
        #print(np.linalg.norm(v - nv_v))
        print(iter)
        return nv_a
    else:
        return v_iteration(gamma, nv_v, iter+1)


print(v_iteration( gamma: 0.99,v0, iter: 0)) #réponse : [1,2]
```

Figure 4: Code de l'algorithme value iteration , avec epsilon = 0.05 et V*0=[0,0]

La figure 5 représente la politique optimale en fonction du parametre de discount gamma (compris entre 0 et 1) et de s (0 ou 1) l'état initial, la représentation était sous forme de points colorés en utilisant plt.plot(gamma, s, 'yo') pour l'action 1 et plt.plot(g, s, 'ko') pour l'action 2 (jaune pour décision '<' et noir pour '≪') .

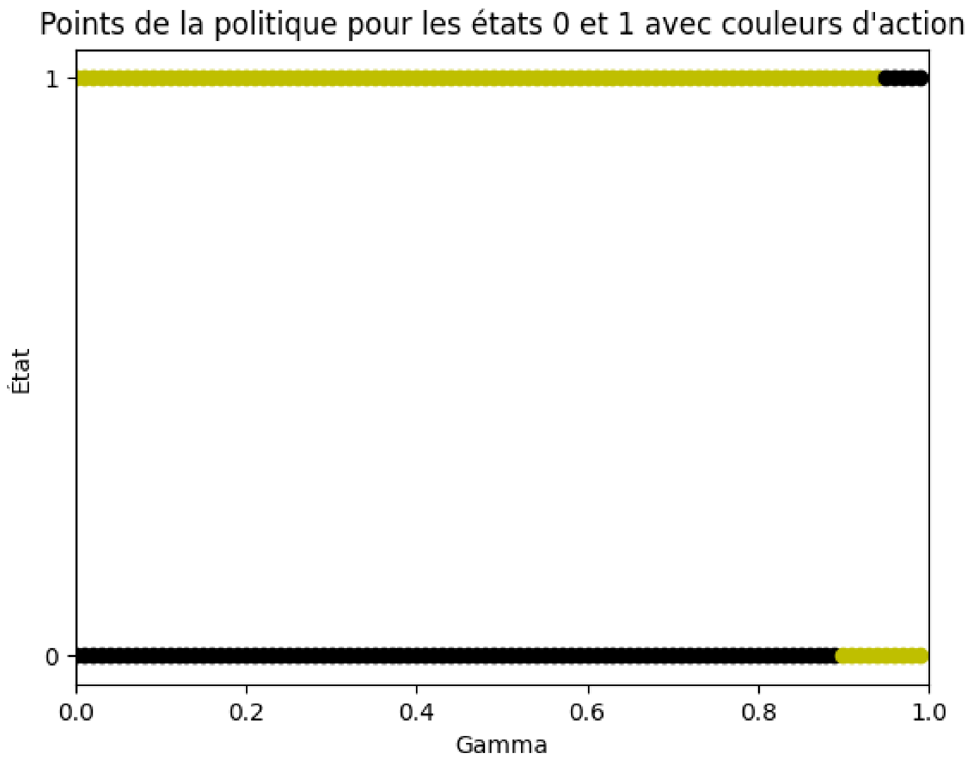

Figure 5: Figure représenant la politique optimal en fonction de gamma (entre 0 et 1) et l'état s (0 ou 1) , le jaune représente la décision 1 '<' or le noir est pour la décision 2 '≪'

Observation: Pour $\gamma < 0.9$, la politique reste la même ($\pi_*(a = 2|s = 0) = 1$ et $\pi_*(a = 1|s = 1) = 1$). Pour γ compris entre 0.9 et 0.93, la décision optimale est $a = 1$ ('<') pour les deux états. Pour γ compris entre 0.93 et 1, la décision optimale est ('<') pour $s = 0$ et '≪' pour $s = 1$.

Policy Iteration : En prenant une valeur arbitraire de $\pi_0 = [1, 2]$ (où $p(A = 1|s = 0) = 1$ et $p(A = 2|s = 1) = 1$) pour tous les états s (état 0 et état 1), on adapte en chaque itération la politique $\pi_{k+1} = \text{greedy}(V_{\pi k})$ obtenue en calculant d'abord la fonction valeur de la politique π_k (en utilisant la relation matricielle du slide 23/35 en cours3) , l'algorithme de Policy Iteration se présente comme suit :

```
eps = 0.05
poli0 = [1, 2]
1 usage
def p_iteration(gamma,poli,max_iterations) :
    poli = poli0
    for iteration in range(max_iterations):
        V_pi = np.dot(np.linalg.inv(np.eye(2) - gamma*transition(poli)),reward(poli))
        max_i1, max_v1 = max(enumerate((1 + gamma * V_pi[0], 10 + gamma * V_pi[1])), key=lambda x: x[1])
        max_i2, max_v2 = max(enumerate((gamma * V_pi[1], -15 + gamma * V_pi[0])), key=lambda x: x[1])
        nv_poli=[max_i1 +1,max_i2 +1]
        nv_V = np.array([max_v1,max_v2])
        if np.linalg.norm(V_pi - nv_V) <= eps or iteration == max_iterations - 1:
            print(np.linalg.norm(V_pi - nv_V))
            #print(iteration)
            return nv_poli
        else:
            poli = nv_poli

print(p_iteration( gamma: 0.978,poli0, max_iterations: 10000)) #réponse : [1,2]
```

Figure 6: Code de l'algorithme Policy Iteration, avec $\epsilon = 0.05$ et $\pi_0 = [1, 2]$

La figure suivante représente la politique optimal en fction de gamma (de 0 à 0.99) et de s (0 ou 1) , la représentation était sous forme de points colorés (jaune pour décision '$<$' et noir pour '$\ll$')

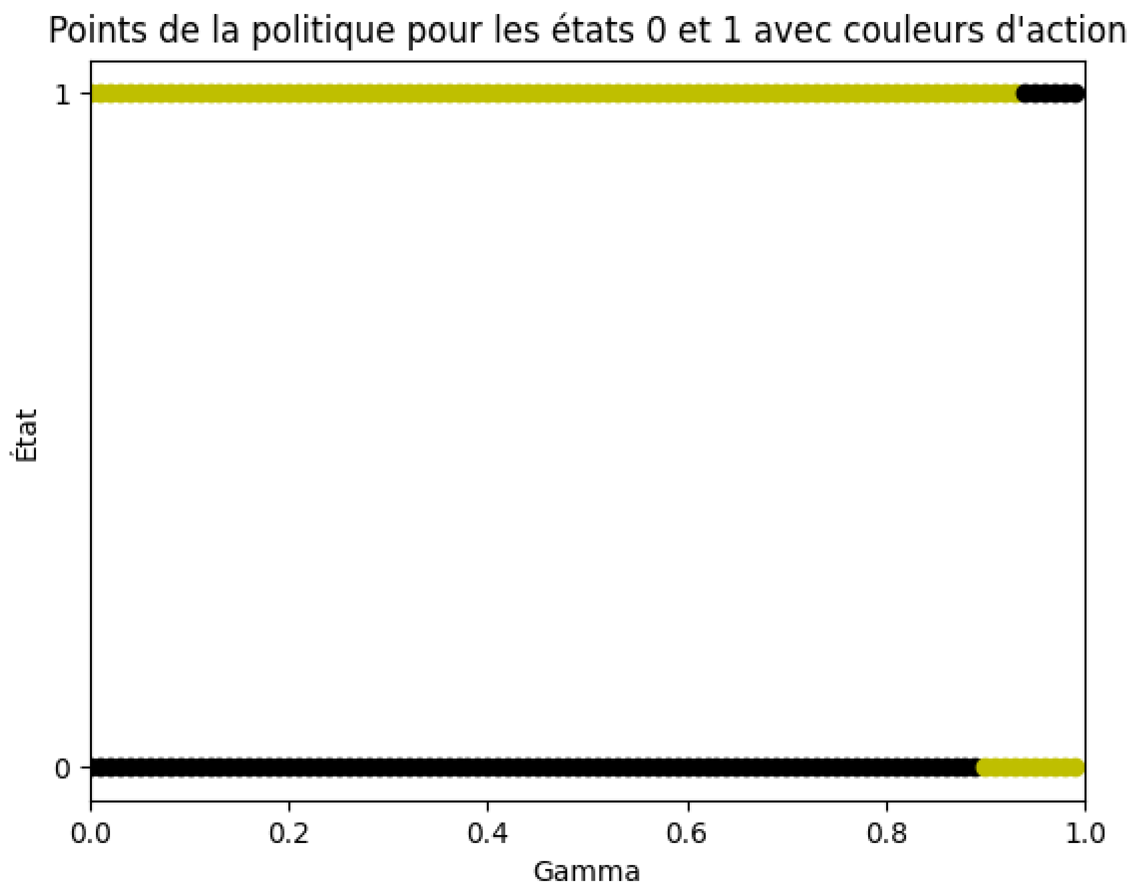

Figure 7: Figure représenant la politique optimal en fonction de gamma (entre 0 et 1) et l'état s (0 ou 1) , le jaune représente la décision 1 '$<$' or le noir est pour la décision 2 '$\ll$'

Observation : On observe que la figure est similaire à celle au cas de value iteration , une différence seulement en performance et précision de calcul pour la policy iteration , ce qui valide bien les résulats obtenus par 2 méthodes différentes.