Exercises Lecture 2
Introduction to reinforcement learning

Léo Meissner et Héloïse Lafargue

Département Sciences du Numérique - Troisième année
Image & multimédia
2023-2024

1 Exercise : Finite horizon MDP

Revenue management : Littlewood's model

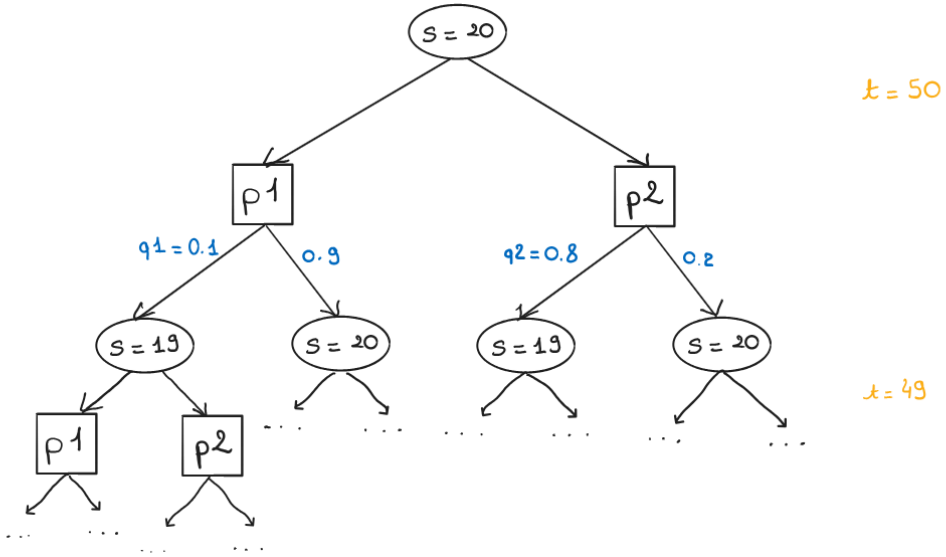
Problem : An airplane has 20 seats available, and the sell closes in 50 days. At every time epoch, the airplane decides the selling price : Either $p1 = 5$, and then it will sell a seat with probability $q1 = 0.1$, or $p2 = 1$, and then it will sell a seat with probability $q2 = 0.8$.

- Model the problem as a Finite Horizon MDP with total reward criterion, and write the optimality equation

- What is the optimal selling strategy ? which qualitative conclusion you can draw from the solution ?

Help : Let s denote the remaining seats available, and $V_T(s)$ denote the total reward in state s and with T days left. Use the principle of optimality of slide 54/61.

— Modèle et équation d'optimalité :



- $S = \{0, 1, 2, ...20\}$, ensemble des états (nombre de sièges libres).

- $A = \{p1, p2\}$, ensemble des actions (augmenter ou baisser le prix).

- R, fonction récompense $r(s, a)$:

$$r(s, 1) = p1 * q1 + 0 = 0.5$$
$$r(s, 2) = p21 * q2 + 0 = 0.8$$

- P, matrice de probabilité de transition d'état $p(s'|s, a)$:

$$p(s - 1|s, 1) = q1 = 0.1$$
$$p(s - 1|s, 2) = q2 = 0.8$$
$$p(s|s, 1) = 1 - q1 = 0.9$$
$$p(s|s, 2) = 1 - q2 = 0.2$$

- Equation d'optimalité

$$V_T(s) = max\{r(s, 1) + p(s - 1|s, 1)V_{t-1}(s - 1) + p(s|s, 1)V_{t-1}(s);$$
$$r(s, 2) + p(s - 1|s, 2)V_{t-1}(s - 1) + p(s|s, 2)V_{t-1}(s)\}$$
$$= max\{0.5 + 0.1V_{t-1}(s - 1) + 0.9V_{t-1}(s); 0.8 + 0.8V_{t-1}(s - 1) + 0.2V_{t-1}(s)\}$$

$$V_T(0) = 0, V_{50}(s) = 0$$

— Résolution

La stratégie de vente optimale implique de déterminer, pour chaque état s et chaque étape de temps t, s'il faut fixer le prix de vente à p1 ou p1 afin de maximiser la récompense totale attendue.

```python
import numpy as np
import random

# Paramètres du problème
nombre_de_places = 20
jours = 50
prix1, q1 = 5, 0.1
prix2, q2 = 1, 0.8

# Création d'une matrice pour stocker les valeurs optimales
V = np.zeros((nombre_de_places + 1, jours + 1))

# Matrices pour stocker la politique optimale (0 pour p1, 1 pour p2)
policy = np.zeros((nombre_de_places + 1, jours + 1), dtype=int)

# Remplissage de la matrice de valeur (programmation dynamique)
for t in range(jours, -1, -1):
    for s in range(nombre_de_places + 1):
        if t == jours:
            V[s, t] = 0
        else:
            R1 = prix1 * q1
            R2 = prix2 * q2

            V1 = R1 + (1 - q1) * V[s, t + 1] + q1 * V[s-1, t+1] if s > 0 else R1
            V2 = R2 + (1 - q2) * V[s, t + 1] + q2 * V[s-1, t+1] if s > 0 else R2

            V[s, t] = max(V1, V2)
            policy[s, t] = 0 if V1 > V2 else 1


for t in range(jours, -1, -1):
  print(policy[:,t])

# Extraction de la stratégie optimale
strategie_optimale = np.zeros(jours, dtype=int)
places_restantes = nombre_de_places
revenu_total = 0
place_vendu5 = 0
for t in range(jours):
    strategie_optimale[t] = policy[places_restantes, t]
    if places_restantes > 0:
      if strategie_optimale[t] == 0:
        if random.random() < q1:
          places_restantes -= 1
          revenu_total += 5
          place_vendu5 += 1
      if strategie_optimale[t] == 1:
        if random.random() < q2:
          places_restantes -= 1
          revenu_total += 1
print(places_restantes)
print(place_vendu5)
# Affichage de la stratégie optimale
print("Stratégie optimale (0 pour p1, 1 pour p2) :")
print(strategie_optimale)


print("Revenu total avec la stratégie optimale : $", revenu_total)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
0
1
Stratégie optimale (0 pour p1, 1 pour p2) :
[0 0 0 0 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0
 0 0 0 1 1 1 0 1 1 1 0 1 1]
Revenu total avec la stratégie optimale : $ 24
```

Commencer le dernier jour ($t = T$) :
Pour chaque état s, calculer la récompense totale attendue pour chaque action :

$$RécompenseAttendue_a(s) = r(s, a) + p(s - 1|s, a)V_{t-1}(s - 1) + p(s|s, a)V_{t-1}(s)$$

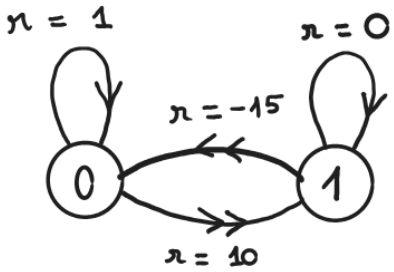
Mettre à jour l'action optimale pour chaque état :

$$ActionOptimale_t(s) = argmax_{a \in A}(RécompenseAttendue_a(s))$$

Remonter dans le temps ($t = T - 1, T - 2, \ldots, 0$) :
Pour chaque état s, calculer la récompense totale attendue pour chaque action et mettre à jour l'action optimale en fonction de la récompense attendue.

La conclusion qualitative de la solution implique de comprendre comment la stratégie de vente optimale évolue dans le temps, en considérant les compromis entre la vente à des prix plus élevés avec des probabilités plus faibles par rapport à des prix plus bas avec des probabilités plus élevées. Cela donne également des indications sur la maximisation du revenu compte tenu de la disponibilité des sièges et de la date de fin de vente.

2 Exercise : Infinite Horizon MDP



What is the optimal policy (for total discounted reward) for various values of γ ?

- Solve the optimality equation it by value iteration

- Characterize the optimal policy using policy iteration

Help : You might use Value Iteration (slide 15/35), or policy iteration (slide 23/35)

— Modèle :

- $S = \{0, 1\}$, ensemble des états.

- $A = \{0, 1\}$, ensemble des actions (aller en 0, aller en 1).

- R, fonction récompense $r(s, a)$:

$$r(0,0) = 1, r(1,0) = -15$$
$$r(0,1) = 10, r(1,1) = 0$$

- Equation d'optimalité de Bellman

$$V_*(0) = max\{1 + \gamma V(0); 10 + \gamma V_*(1)\}$$
$$V_*(1) = max\{-15 + \gamma V(0); 0 + \gamma V_*(1)\}$$

— Résolution via Value iteration, opérateur non linéaire.

Initialisation :
Fixer $V_0(s) = 0$ pour tous les s dans S.

Mise à jour de l'itération de la valeur :
Pour chaque état s danse S, mettre à jour $V_{k+1}(s)$ en utilisant l'équation de Bellman pour $V_*(s)$:

$$V_{k+1}(s) = \max_{a \in A}\{r(s,a) + \gamma \sum_{s'} P(s'|s,a)V_k(s)\}$$

$$V_{k+1}(0) = \max\{1 + \gamma V_k(0); 10 + \gamma V_k(1)\}$$

$$V_{k+1}(1) = \max\{15 + (0); 0 + \gamma V_k(1)\}$$

Répéter l'itération jusqu'à ce que $V_{k+1}(s)$ converge pour tout s.

```python
import numpy as np

# Paramètres du modèle
S = [0, 1]  # ensemble des états
A = [0, 1]  # ensemble des actions
gamma = 0.1  # facteur de réduction

# Fonction de récompense r(s, a)
reward = np.array([[1, 10], [-15, 0]])

# Algorithme d'itération de la valeur
def value_iteration():
    V = np.zeros(len(S))
    epsilon = 1e-6

    while True:
        delta = 0

        for s in S:
            max_value = float('-inf')
            for a in A:
                new_value = reward[s, a] + gamma * sum([p * V[next_s] for p, next_s in zip([0.5, 0.5], S)])
                max_value = max(max_value, new_value)

            delta = max(delta, np.abs(V[s] - max_value))
            V[s] = max_value

        if delta < epsilon:
            break

    return V

optimal_values = value_iteration()
print("Valeurs optimales pour chaque état:", optimal_values)
```

```
Valeurs optimales pour chaque état: [10.55555553  0.55555555]
```

— Résolution via Policy iteration opérateur linéaire.


```python
import numpy as np

# Paramètres du modèle
S = [0, 1]  # ensemble des états
A = [0, 1]  # ensemble des actions
gamma = 0.1  # facteur de réduction

# Fonction de récompense r(s, a)
reward = np.array([[1, 10], [-15, 0]])

# Algorithme d'itération de la valeur
def value_iteration():
    V = np.zeros(len(S))
    epsilon = 1e-6

    while True:
        delta = 0

        for s in S:
            max_value = float('-inf')
            for a in A:
                new_value = reward[s, a] + gamma * sum([p * V[next_s] for p, next_s in zip([0.5, 0.5], S)])
                max_value = max(max_value, new_value)

            delta = max(delta, np.abs(V[s] - max_value))
            V[s] = max_value

        if delta < epsilon:
            break

    return V

optimal_values = value_iteration()
print("Valeurs optimales pour chaque état:", optimal_values)
```

```
Valeurs optimales pour chaque état: [10.55555553  0.55555555]
```


$$\Pi(a1|0) = 1, \Pi(a2|1) = 1$$

$$V_{\Pi_0} = r + \gamma P_{\Pi_0} V_{\Pi_0}$$

$$(I - \gamma P_{\Pi_0}) V_{\Pi_0} = r$$

$$V_{\Pi_0} = (I - \gamma P_{\Pi_0})^{-1}$$

$$\Pi_1 = argmax_a \{r(s,a) + \gamma p^{\Pi}(s,a)\}$$

Initialiser la politique :
Commencer par une politique initiale Π où chaque état s est associé à une action a (par

exemple, $\Pi(s) = a$).

Évaluation de la politique :
Étant donné la politique actuelle Π, calculer la fonction de valeur $V_\Pi(s)$ en utilisant le système linéaire suivant :

$$V_\Pi(s) = r(s, \Pi(s)) + \gamma \sum_{s'} P(s'|s, (s))V_\Pi(s')V_\Pi(s')$$

Résoudre ce système pour $V_\Pi(s)$ en utilisant des techniques appropriées d'algèbre linéaire.

Amélioration de la politique :
Mettre à jour la politique en utilisant une stratégie gloutonne :

$$\Pi'(s) = argmax_{a \in A}\{r(s, a) + \gamma \sum_{s'} P(s'|s, (s))V_\Pi(s')V_\Pi(s')\}$$

Si Π' n'a pas changé par rapport à la politique précédente, s'arrêter et retourner la politique optimale. Sinon, définir Π' et retourner à l'étape 2.

En suivant ces étapes pour l'itération de la politique, nous avons caractériser la politique optimale pour le MDP donné pour différentes valeurs de γ.

Nous observons que si gamma est proche de 0, le poids accordé aux prochaines itérations est faible. En commençant à l'état 0, on choisira donc de se diriger en 1 grâce au reward de 10 puis on sera bloqué à l'état 1 à cause du reward -15 que l'algorithme ne veut pas emprunté. Au contraire avec un gamma proche de 1, l'algorithme choisira de rester en 0 car il voit que même avec un reward de 10 dans l'immédiat, prendre cette branche ne sera pas rentable sur le long terme.