



# Reinforcement Learning

Realisé par:

Mohamed EL YESSEFI

Yahya YOUNES

*2A-R*

2022-2023

# Contents

1	Finite horizon MDP	3
2	Grid World	4
3	Lecture 4 : Exercise 1 : Learning version of problem from TD2	7
4	A brief feedback of what we understood from this course	14

# 1 Finite horizon MDP

The problem can be modeled as a finite-horizon Markov Decision Process (MDP) with a total reward criterion. The total reward is given by the function  $V(s)$ , which represents the total reward in state  $s$  with  $T$  days remaining. The principle of optimality can be applied to this problem, yielding :

$$V_T(s) = \max(q1 * (5 + V_{T-1}(s - 1)) + q2 * (1 + V_{T-1}(s)))$$

$$r(s, 1) = 5 * 0.1 = 0.5$$

$$r(s, 2) = 1 * 0.8 = 0.8$$

$$V_r(s) = \max(r(s, 1) + 0.1V_{T-1}(s - 1) + 0.9V_{T-1}(s), r(s, 2) + 0.8V_{T-1}(s - 1) + 0.2V_{T-1}(s))$$

$$V_0(s) = 0$$

$$V_1(s) = \max(r(s, 1), r(s, 2)) = 0.8$$

```
def calc_V_T(T, s, p1, q1, p2, q2, V_prev) :  
    if T == 0:  
        return 0  
    else :  
        r1 = p1 * q1  
        r2 = p2 * q2  
  
        V_T_s_1 = r1 + q1 * V_prev[s - 1] + (1 - q1) * V_prev[s]  
        V_T_s_2 = r2 + q2 * V_prev[s - 1] + (1 - q2) * V_prev[s]  
        return max(V_T_s_1, V_T_s_2)  
  
if __name__ == '__main__':  
    V_prev = [0] * 21 # Valeurs de V {T-1} pour tous les s possibles (0 à 20)  
    V_T = []  
    for t in range(0, 51):  
        V_T.append ( [calc_V_T(t, s, 5, 0.1, 1, 0.8, V_prev) for s in range(21)] )  
    V_prev = V_T[-1] # Mise à jour de V prev pour le prochain tour de boucle  
    print (V_T) # Affiche les valeurs de V_T pour tous les T possibles (0 à 50)
```

Figure 1: A screenshot of our Python code

## 2 Grid World

Here we calculate  $V_0(s_{13})$  and  $V_0(s_{23})$  using the complete formula

$V_0(s)$  :

	1	2	3	4
1	0	0	0	<del>1</del>
2	0	<del>X</del>	0	-1
3	0	0	0	<del>0</del>

$$V_1(s_{13}) = -0,04 + 0,999 \times \max \left[ \begin{array}{l} 0,8 \times (+1) + 0,1 \times (0) + 0,1 \times (0) \quad (\text{right}) \\ 0,8 \times (0) + 0,1 \times (+1) + 0,1 \times (0) \quad (\text{up}) \\ 0,8 \times (0) + 0,1 \times (0) + 0,1 \times (0) \quad (\text{left}) \\ 0,8 \times (0) + 0,1 \times (-1) + 0,1 \times (0) \quad (\text{down}) \end{array} \right]$$

Donc  $V_1(s_{13}) = 0,7592$ .

$$V_1(s_{23}) = -0,04 + 0,999 \times \max \left[ \begin{array}{l} 0,8 \times (0) + 0,1 \times (-1) + 0,1 \times (0) \quad (\text{up}) \\ 0,8 \times (0) + 0,1 \times (0) + 0,1 \times (0) \quad (\text{left}) \\ 0,8 \times (-1) + 0,1 \times (0) + 0,1 \times (0) \quad (\text{right}) \\ 0,8 \times (0) + 0,1 \times (-1) + 0,1 \times (0) \quad (\text{down}) \end{array} \right]$$

$V_1(s_{23}) = -0,04$

Figure 2: Le calcul de  $V_0(s_{13})$  et  $V_0(s_{23})$

V1(s)	1	2	3	4
1	-0.04	-0.04	0.7592	1
2	-0.04	X	-0.04	-1
3	-0.04	-0.04	-0.04	-0.04

V2(s)	1	2	3	4
1	-0.07996	-0.55	0.7592	1
2	-0.07996	X	-0.151	-1
3	-0.07996	-0.07996	-0.07996	-0.07996

V3(s)	1	2	3	4
1	-0.11	0.45	0.819	1
2	-0.11	X	0.451	-1
3	-0.11	-0.11	-0.11	-0.11

V4(s)	1	2	3	4
1	0.29	0.70	0.88	1
2	-0.159	X	0.56	-1
3	-0.159	-0.159	0.29	-0.159

Figure 3: Le calcul de  $V_i(s)$  jusqu'à sa convergence

V5(s)	1	2	3	4
1	0.538	0.809	0.90	1
2	0.167	X	0.62	-1
3	-0.199	0.16	0.37	0.081

V6(s)	1	2	3	4
1	0.677	0.84	0.91	1
2	0.423	X	0.64	-1
3	0.09	0.29	0.48	0.168

V7(s)	1	2	3	4
1	0.74	0.85	0.91	1
2	0.58	X	0.65	-1
3	0.33	0.40	0.52	0.26

V8(s)	1	2	3	4
1	0.77	0.85	0.91	1
2	0.66	X	0.65	-1
3	0.49	0.45	0.54	0.37

Figure 4: Le calcul de  $V_i(s)$  jusqu'à sa convergence

```
def calculate(up,right,left,down) :

    return -0.04 + 0.999 * max(0.8 * up + 0.1 * left + 0.1 * right, 0.8 *
        right + 0.1 * down + 0.1 * up, 0.8 * down + 0.1 * left + 0.1 * right
        , 0.8 * left + 0.1 * up + 0.1 * down)
```

Figure 5: Un code simple utilisé sur python

### 3 Lecture 4 : Exercise 1 : Learning version of problem from TD2

#### Evaluation part :

The first part of Planning we calculate  $V(1)$  and  $V(2)$  :

Exercise 1 : Planning : Calculate  $V(1)$  and  $V(2)$

+ First let's define the transition probability matrix :

$$P = \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{pmatrix}$$

+ For the matrix  $R$  we'll set it to the mean of rewards of each state

$$R = \begin{pmatrix} \frac{1+10}{2} \\ \frac{0-15}{2} \end{pmatrix} \rightarrow R = \begin{pmatrix} 5,5 \\ -7,5 \end{pmatrix}$$

+ let  $\gamma = 0,9$

+ Using the equation in 18/43 in Lecture 2

we get :  $V = (I - \gamma P)^{-1} \cdot R$

$$\Rightarrow V = \begin{pmatrix} 5,5 & 4,5 \\ 4,5 & 5,5 \end{pmatrix} \begin{pmatrix} 5,5 \\ -7,5 \end{pmatrix}$$
$$\Rightarrow \boxed{V = \begin{pmatrix} -3,5 \\ -16,5 \end{pmatrix}}$$

So  $V(1) = -3,5$  and  $V(2) = -16,5$

Figure 6: Planning part of Exercise 1

In the following part of learning we implemented a Python code that implements the algorithm TD(0), and we verify that the algorithm learns the correct values of the value function

Now we test the function and we calculate the mean value of all the  $V(1)$  and  $V(2)$  values we get :

+ Code + Texte

```
✓ 0 s 1 # I define the neighbors of each states : In this case we have four cases :
2
3 # If we are in 0 and we stay so it will be 0 # If we are in 1 and we stay so it will be 1
4 # If we are in 0 and we go so it will be 1 # If we are in 1 and we go so it will be 0
5
6 neighbors = {0:{'stay':0, 'go':1},
7             1:{'stay':1, 'go':0}}
8
9
10 # In this policy i defined it randomly using rd.uniform function
11 policy = {0:{'stay':rd.uniform(0,1), 'go':rd.uniform(0,1)},
12          1:{'stay':rd.uniform(0,1), 'go':rd.uniform(0,1)}}
13
14
15 # I define the rewards in each case using the draw in the Lecture
16 reward = {0:{'stay':1., 'go':10.},
17           1:{'stay':0., 'go':-15.}}
18
19 # This function, given a state "s" and an action "stay or go" it gives you the reward R and the next state
20 def state_transition(s, action, neighbors = neighbors, rewards = reward):
21     nextState = neighbors[s][action]
22     reward = rewards[s][action]
23     return (nextState,reward)
24
25 # This function uses the TD(0) algorithm to and when running it we find close values to the previous ones
26 def td_0_state_transition(policy, n_samps, gamma = 0.9):
27     ## Initialize the state list and state values
28     states = list(policy.keys())
29     v = [0]*len(list(policy.keys()))
30     action_index = list(range(len(list(policy[0].keys()))))
31     nbVisits = 0
32     alpha = 0.2
33     for _ in range(n_samps):
34         nbVisits += 1
35         s = rd.choice(states, size =1)[0]
36         probs = list(policy[s].values())
37         a = list(policy[s].keys())[rd.choice(action_index, size = 1)[0]]
38         transistion = state_transition(s, a)
39         v[s] = v[s] + alpha * (transistion[1] + gamma * v[transistion[0]] - v[s])
40     return(v)
```

Figure 7: Learning part of Exercise 1



✓  
22 s



```
1 # We test the function and print the mean values of V
2 x = []
3 y = []
4
5 for i in range (500):
6     v = td_0_state_transition(policy, n_samps = 1000)
7     x.append(v[0])
8     y.append(v[1])
9
10 # The mean values of V[0] and V[1]
11 m_1 = np.mean(x)
12 m_2 = np.mean(y)
13
14
15 print(m_1,m_2)
16
```

-3.525919596393737 -16.552116165848428

Figure 8: Test of the function

We note that the mean value of  $V(1)$  and  $V(2)$  is equal to the analytic values we calculated previously.

### Control part :

In this part we implement Q-learning in order to learn the optimal control policy :

+ Code + Texte

### Q-Learning algorithm using uniform and epsilon-greedy policies

```
✓ 0 s ▶ 1 # Let's set epsilon for the e-greedy policy
2 epsilon = 0.9
3
4
5 # In this policy we set it to be e-greedy
6
7 policy = {0: {'stay': 1-epsilon, 'go': epsilon},
8            1: {'stay': epsilon, 'go': 1-epsilon}}
9
10
11 def state_transition(next_state, policy, action_index, greedy):
12
13     # If we use the greedy policy greedy is set to True and we can tune the parameter epsilon
14     if greedy == True:
15         probs = list(policy[next_state].values())
16         next_action_index = rd.choice(action_index, size = 1, p = probs)[0]
17         next_action = list(policy[next_state].keys())[next_action_index]
18
19     # Otherwise we use the uniform policy
20     else:
21         next_action_index = rd.choice(action_index, size = 1)[0]
22         next_action = list(policy[next_state].keys())[next_action_index]
23     return(next_action_index, next_action)
24
25
```

+ Code + Texte

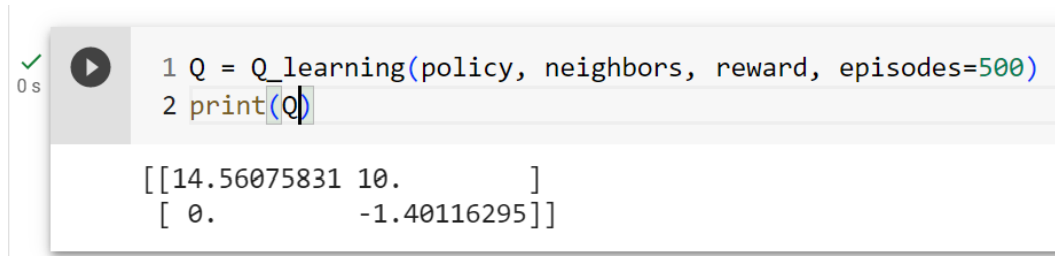
```

28
29 def Q_learning(policy, neighbors, rewards, episodes, gamma = 0.94):
30     alpha = 0.2
31
32     # Initialize the state S
33     states = list(policy.keys())
34     n_states = len(states)
35
36
37     # Initialize possible actions and the action values
38     possible_actions = list(rewards[0].keys())
39     action_index = list(range(len(list(policy[0].keys()))))
40
41     # Initialize Q(S,A)
42     Q = np.zeros((len(states), len(possible_actions)))
43
44
45     # Depends if its uniform or epsilon-greedy policy
46     current_policy = policy.copy()
47
48     # Repeat (for each episode) :
49     for _ in range(episodes):
50
51         #Initialize S
52         s = rd.choice(states, size = 1)[0]
53
54         # Now choose action following policy
55         a_index, a = state_transion(s, current_policy, action_index, True)
56
57         # Increment state
58         next_state = n_states + 1
59
60         # Get next_state given s and a
61         next_state = neighbors[s][a]
62
63         # Break any tie with multiple max values by random selection
64         action_values = Q[next_state,:]
65         next_action_index = rd.choice(np.where(action_values == max(action_values))[0], size = 1)[0]
66
67         # Take action
68         next_action = possible_actions[next_action_index]
69
70         # Observe R
71         reward = rewards[s][a]
72
73         # Update the action values
74         Q[s, a_index] = Q[s, a_index] + alpha * (reward + gamma * Q[next_state, next_action_index] - Q[s, a_index])
75
76         # Set action and state for next iteration
77         a = next_action
78         a_index = next_action_index
79         s = next_state
80
81     return(Q)

```

Figure 9: Python code for Q-learning

The results we get for uniform policy by setting the parameter greedy to False :



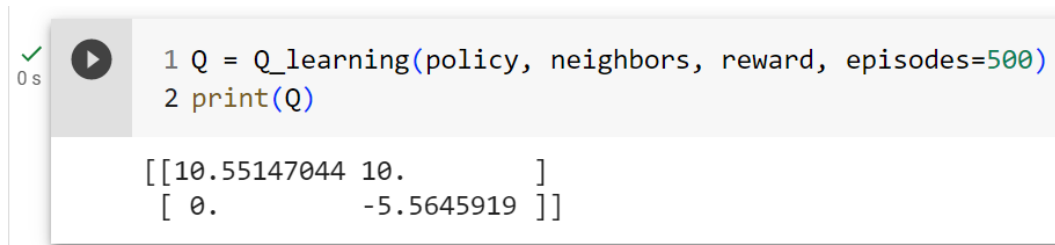
A Jupyter Notebook cell showing the execution of a Q-learning function. The cell has a green checkmark and a play button icon. The code consists of two lines: `1 Q = Q_learning(policy, neighbors, reward, episodes=500)` and `2 print(Q)`. The output is a 2x2 matrix displayed as a list of lists: `[[14.56075831 10.], [0. -1.40116295]]`.

```
1 Q = Q_learning(policy, neighbors, reward, episodes=500)
2 print(Q)
```

```
[[14.56075831 10.]
 [ 0.         -1.40116295]]
```

Figure 10: Results for uniform policy

The results we get for uniform policy by setting the parameter greedy to True and setting epsilon to 0.9 :



A Jupyter Notebook cell showing the execution of a Q-learning function. The cell has a green checkmark and a play button icon. The code consists of two lines: `1 Q = Q_learning(policy, neighbors, reward, episodes=500)` and `2 print(Q)`. The output is a 2x2 matrix displayed as a list of lists: `[[10.55147044 10.], [0. -5.5645919 ]]`.

```
1 Q = Q_learning(policy, neighbors, reward, episodes=500)
2 print(Q)
```

```
[[10.55147044 10.]
 [ 0.         -5.5645919 ]]
```

Figure 11: Results for greedy policy

With which policy (uniform or  $\epsilon$ -greedy) the learning is fastest

We use the function time from Python to compare which one is the fastest :

```
1 import time
2
3 # Get the current time before executing the function
4 start_time_uniform = time.time()
5
6 # We call the function
7 We call Q learning by setting greedy to False so it will be uniform policy
8
9 # Get the current time after executing the function
10 end_time_uniform = time.time()
11
12 # Calculate the time taken
13 execution_time_uniform = end_time_uniform - start_time_uniform
14
15
16 #-----
17
18 |
19 # Get the current time before executing the function
20 start_time_greedy = time.time()
21
22 # We call the function
23 We call Q learning by setting greedy to True so it will be greedy policy
24
25 # Get the current time after executing the function
26 end_time_greedy = time.time()
27
28 # Calculate the time taken
29 execution_time_greedy = end_time_greedy - start_time_greedy
30
31
32 print("Execution time using greedy policy :", execution_time_greedy,"seconds")
33 print("Execution time using uniform policy :", execution_time_uniform,"seconds")
```

Figure 12: The logic we used to compare the time each policy needs

Those are the results we get :

```
125
126 print("Execution time using greedy policy :", execution_time_greedy,"seconds")
127 print("Execution time using uniform policy :", execution_time_uniform,"seconds")

Execution time using greedy policy : 0.0002727508544921875 seconds
Execution time using uniform policy : 0.00019311904907226562 seconds
```

Figure 13: Results for greedy policy

In terms of computational performance, the uniform strategy is faster than the  $\epsilon$ -greedy strategy in learning Q.

This is because the actions of the uniform strategy are chosen randomly with equal probability, which requires the least amount of computation. It does not require any

estimation or comparison of Q values. Therefore, the decision process is simple and computationally efficient.

On the other hand, the e-greedy policy in many cases requires estimation and comparison of Q values to determine the activity with the highest expected total reward. This requires additional calculations to determine the maximum value of Q and to make a decision accordingly. Choosing a random action with low probability during the testing phase also increases the computational cost.

## 4 A brief feedback of what we understood from this course

In this part we tried to illustrate briefly and in our words what we learned from this course and what we discovered in the domain of Reinforcement learning

**Reinforcement learning** : is like teaching a robot or a computer how to make good decisions by giving it rewards when he does good and punishing him when he does bad so he will keep looking all the time to maximize the good rewards, exactly like a dog listening to you to get some sweets !.

Imagine you have a little robot friend who wants to learn how to play a game.

In the game, show the robot different situations or states, such as being in a certain place or seeing certain things. The robot tries different actions to see what will happen. If it does something good, you get a thumbs up or a reward, and if it does something bad, you get a thumbs down or a punishment.

The robot learns from these rewards and punishments. It begins to understand for which actions it gets more rewards and for which it gets less. Over time, it tries different actions and becomes better at making decisions.

The goal is for the robot to learn how best to play the game and earn as many rewards as possible. The robot learns from its mistakes and gets better at the game by trying different actions and seeing what works.

A quick link to a fun video to understand reinforcement learning : <https://youtu.be/spfpBrBjntg>

### Markov

The Markov property is about looking at what's happening now to understand and predict what will happen next, without worrying too much about what happened in the past. It helps us simplify things and make good guesses about the future based on the current situation.

## Markov Reward Process

Imagine you have a special toy that you earn points with as you play. Every time you spin or do something, the toy rewards you with points for what you did.

The Markov reward process is similar to using this special toy, but slightly different. Instead of just earning points, it is important to understand how the points relate to the action you took.

## Bellman equation

The Bellman equation is used to compute a value function representing the expected total reward an agent will receive from a given state and policy. By solving the Bellman equation iteratively, algorithms can estimate and improve the value function, leading to better decision-making strategies.

## Q-Learning : Uniform policy vs $\epsilon$ -greedy policy

The main difference between the uniform strategy and the  $\epsilon$ -greedy strategy lies in the way they make decisions about the choice of actions. A uniform strategy assigns the same probability to all available actions and chooses actions randomly, regardless of their quality. For example, if there are four possible actions, a uniform strategy would assign a probability of 0.25 to each action.

The  $\epsilon$ -greedy strategy, on the other hand, balances prospecting and exploration, usually choosing the action with the highest expected return, but occasionally choosing a random action with a low  $\epsilon$ -probability.

To illustrate that I give this example we learned during the course :

The uniform policy : When you want to eat and you always choose the restaurant that you know it's good, or you always choose to play the game the same way you always did and made you win, briefly you play it safe !

The greedy policy : When you want to eat you choose by a epsilon probability the restaurant you always know it's good and where you ate all the time, but with a probability of  $1-\epsilon$  you can pick a random restaurant to discover it and add it to your list either the good ones or bad ones. In the case of the game you choose to play the way you always did by a probability epsilon, but with a probability  $1-\epsilon$  you try a new trick and see if it works or not. In brief you take the risk to balance between discovering but not losing !