OUSMANE THIONGANE

SAPIENZA
UNIVERSITÀ DI ROMA

# Reinforcement Learning

## Assignment 1 – Multi-Armed Bandits and Dynamic Programming

Matricula: 2184503

October 19th, 2024

# ASSIGNMENT 1 –MULTI-ARMED BANDITS AND DYNAMIC PROGRAMMING

## I – Theory

1. Let's consider a Multi-Armed Bandit with 3 arms. We suppose that, after $t = 10$ iterations of the UCB algorithm, we have the following parameters:

$$\hat{\mu}_1^t = 0.2,\ \hat{\mu}_2^t = 0.05,\ \hat{\mu}_3^t = 0.45,\ N_1^t = 5,\ \ N_2^t = 3,\ N_3^t = 2,\ \delta = 0.15,\ T = 100.$$

(a) Let's compute the next arm to be pulled by following the UCB algorithm:

As we have seen in class, the UCB algorithm is defined as follows:

– For the first $K$ iterations, pull each arm once
– For $T = K, \dots, T$ :

Pick the action with the highest upper confidence bound:

$$I_t = \arg\max_{i \in [K]} \left( \hat{\mu}_t(i) + \sqrt{\frac{\ln\left(\frac{KT}{\delta}\right)}{N_t(i)}} \right)$$

Update statistics

By using the $I_t$ formula, we got:

$$I_1 = \hat{\mu}_1^t + \sqrt{\frac{\ln\left(\frac{KT}{\delta}\right)}{N_1^t}} = 0.2 + \sqrt{\frac{\ln\left(\frac{3 \times 100}{0.15}\right)}{5}} = 1.43$$

$$I_2 = 0.05 + \sqrt{\frac{\ln\left(\frac{3 \times 100}{0.15}\right)}{3}} = 1.64$$

$$I_3 = 0.45 + \sqrt{\frac{\ln\left(\frac{3 \times 100}{0.15}\right)}{2}} = 2.40$$

In this case, $I_3$ is the highest upper confidence bound, we can then conclude that **the next arm to be pulled by following the UCB algorithm will be the 3rd arm.**

(b) Let's now compute the updated statistics:

All the statistics from the not chosen arms will remain the same:

$$\hat{\mu}_1^{t+1} = 0.2,\ \hat{\mu}_2^{t+1} = 0.05,\ N_1^{t+1} = 5,\ \ N_2^{t+1} = 3,\ \delta = 0.15.$$

Since the reward, after the 3rd arm has been pulled is $r = 1$, we can compute the updated statistics:

$$N_3^t = 3 \text{ and } \hat{\mu}_3^{t+1} = \frac{0.45 \times N_3^t + 1}{N_3^{t+1}} = \frac{0.45 \times 2 + 1}{3} = 0{,}633$$

2. Given the following environment settings:

- States: $\{s_i : i \in \{1, \dots, 6\}\}$
- Reward:

$$r(s, a, s') \begin{cases} 1 & if\ s = s_2 \\ -10 & if\ s = s_6 \\ 0 & otherwise \end{cases}$$

- Dynamics: $p(s_5|s_5, a_1) = 0.3$, $p(s_6|s_5, a_1) = 0.7$
- Policy: $\pi(s) = a_1 \ \ \forall s \in S$

And the value function at the iteration $k = 1$:

$$v_k = [0, 1, 0, 0, 0, -10]$$

With $\gamma = 0.8$.

Let's compute $V_{k+1}(s_5)$ by following the Value Iteration algorithm.

The Value Iteration algorithm can be described as follows:

– Start with $V_0^*(s) = 0, \forall s$.

– For $i \in \{1, \dots, H\}$

For all states in $S$:

$$V_{i+1}^*(s) \leftarrow \max_{a \in A} \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_i^*(s')\right]$$

$$\pi_{i+1}^*(s) \leftarrow \arg\max_{a \in A} \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_i^*(s')\right]$$

With:

$V_i^*(s) \equiv$ The expected sum of rewards accumulated starting from state $s$, acting optimally for $i$ steps.

$\pi_i^*(s) \equiv$ The optimal action when in state $s$ and getting to act for $i$ steps.

$T(s, a, s') = p(s'|s, a)$.

According to the statement of the exercise, since for all states, the policy is equal to $a_1$, the formula becomes for the state $s_5$:

$$V_{k+1}^{*}(s_5) = \sum_{s'} p(s'|s_5, a)\left[R(s_5, a, s') + \gamma V_i^{*}(s')\right]$$

$$\begin{aligned}\Rightarrow V_{k+1}^{*}(s_5) = {} & 0 \times [0 + \gamma \times 0] \\ & +0 \times [1 + \gamma \times 1] \\ & +0 \times [0 + \gamma \times 0] \\ & +0 \times [0 + \gamma \times 0] \\ & + p(s_5|s_5, a) \times [0 + \gamma \times 0] \\ & + p(s_6|s_5, a) \times [0 + \gamma \times -10]\end{aligned}$$

$$\Rightarrow V_{k+1}^{*}(s_5) = 0 + 0 + 0 + 0 + 0.3 \times [0 + 0.8 \times 0] + 0.7 \times [0 + 0.8 \times -10]$$

$$\Rightarrow V_{k+1}^{*}(s_5) = 0 + 0.7 \times [0 + 0.8 \times -10]$$

$$\boxed{\Rightarrow V_{k+1}^{*}(s_5) = -5{,}6}$$

## II – Practice

1. Let's implement the transitions probabilities and the reward function of a modified version of the FrozenLake Gymnasium environment.

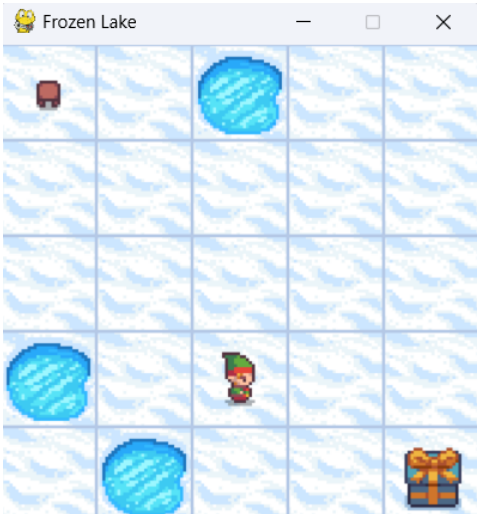


Figure 1 – FrozenLake environment display

**Reward function:**

The reward function must return 0 everywhere but 1 for the goal cell. The goal state is always located at the bottom-right corner of the grid. Since the grid is modeled by a matrix array from size env_size × env_size, with python we can access the bottom-right corner by specifying the last line and the last column of the array with [env_size-1, env_size-1].

**Transitions probabilities:**

We are using the same method than in the value iteration practical. $s'$ corresponds to the state the agent will find itself in if it chooses the direction defined by the action. $s''$ correspond to the state where the agent moves in the wrong direction regarding the chosen action ($a-1$ instead of $a$). The modulo operator allows us to return to the last action in the list (here 3) when $a = 0$.

Finally, we check whether $s$ and $s'$ are valid states (i.e. inside the environment or not in a hole) and then increase the transition probabilities of these states by $\frac{1}{2}$.

The total reward will be 1 if the agent reaches the final state and 0 if the agent fells on a hole, since we do the operation 3 times, the final mean reward will be either $0, \frac{1}{3}, \frac{2}{3}$, or 1.

2. Let's implement the *explore_and_commit* and the *epsilon_greedy.* The environment is a Contextual Multi-Armed Bandit with 3 states and 3 arms.

We complete the explore and commit algorithm the same way we did in practicals. As it can be found online, the Click-Through-Rate is defined as the number of clicks that our website/ads/… receives divided by the number of times our website/ad/… is shown. In our context it gives:

$$CTR = \frac{clicks}{views}$$

This is the value we are going to put into our Q-table.

*Note: We have seen in class that the correct version epsilon_greedy should normally continue to update the statistics (the mistake was carried over past years), so we were given some alternatives to continue the assignment. I choose to modify the code to have the epsilon-geedy works as it should be.*

I added the following strategy to choose the action:

$$\begin{cases} \text{The agent will randomly pull an action} & \text{with probability } \epsilon \\ \text{The agent will pull the current best action} & \text{with probability 1-}\epsilon \end{cases}$$

The rest is relatively similar to the previous method.

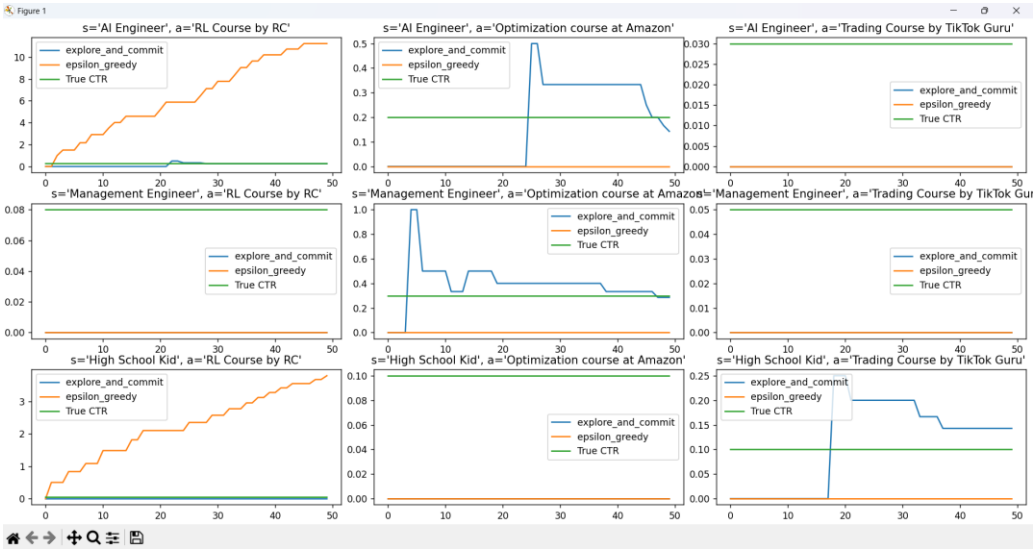


Figure 2 – CMABS Results

We can see that as the steps progress, the *explore_and_commit* method seems to converge towards the true CTR value, while the *epsilon_greedy* method seems to overestimate it. In some cases, where the true CTR is quite low, both methods seem not to have chosen to explore this action, which explains the zero estimated CTR.

## III – Resources

1. My notes from the lecture "Exploration: Regret and Multi-Armed Bandits" (UCB algorithm exercise).

2. The repository of the Value iteration practical:
https://github.com/KRLGroup/RL_2024/blob/main/p02_value_iteration/Value_iteration_sol.ipynb

3. FrozenLake – Gymnasium environment:
https://gymnasium.farama.org/environments/toy_text/frozen_lake/

4. Click-Through-Rate (CTR) definition: https://support.google.com/google-ads/answer/2615875?hl=en#:~:text=CTR%20is%20the%20number%20of,see%20listed%20in%20your%20account.

5. The repository of the Multi-Armed-Bandits (MAB) practical:
https://github.com/KRLGroup/RL_2024/blob/main/p01_multi_armed_bandits/MABs_sol.ipynb

6. The repository of the Q-Learning practical (for the epsilon-greedy strategy):
https://github.com/KRLGroup/RL_2024/blob/main/p01_multi_armed_bandits/MABs_sol.ipynb