Reinforcement Learning for Maze Solving

Matteo Guida[†]

[†] *Physics of Data M.D. - Dipartimento di Fisica e Astronomia G. Galilei, Università degli Studi di Padova, Via Marzolo 8, I-35131 Padova, Italy.*

January 10, 2021

1 Introduction

In the present work a basic reinforcement learning algorithm is implemented in "pure" Python to solve a 2D maze defined on a grid of size 10 x 10 in order to reach a goal cell avoiding borders and walls and choosing to move on sand, when it's convenient. The algorithm involves the interaction of an agent with an environment in discrete time steps, which in this case can be seen as moves of the game, for a fixed amount of steps, which can be seen as the duration of the game. At each time t, the agent receives the current state s_t. It then chooses an action a_t at from the set of available actions $\mathcal{A}_t$, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_t associated with the transition (s_t, a_t, s_{t+1}) is determined. The task of the algorithm is to learn the transition probabilities of the "state-action" space associated to the problem, over which a Markov decision process is defined. Those probabilities are stored not normalized in a matrix called **Q-table** whose rows are associated with all possible states while the columns with all possible actions.

2 Program Description

The assignment is solved through 2 python script (Agent.py and Environment.py) and a Jupyter notebook EX5_Matteo_Guida.ipynb. The script training.py does not train the agent but only load the obtained results to meet the request present in the assignment. Please look at the jupyter notebook in order to have a clearer view of the operations carried out.

2.1 Environment : the Maze

In the script Environment.py the same implementation presented in the laboratory program is conserved with just few important modifications. There are four types of states:

- **Wall**: cannot be crossed and its reward is - 1.
- **Sand**: can be crossed and its reward is -0.6.
- **Normal Cell**: can be crossed and its reward is 0.
- **Goal**: is a single cell in the maze and its reward is +1.

In figure 1 the designed maze structure chosen purely based on individual taste is shown.

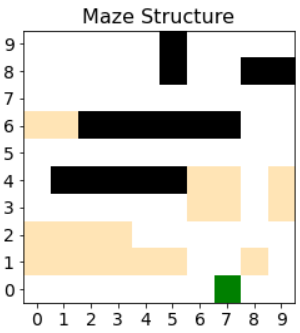

Figure 1: Maze structure fixed for all the work.

2.2 Agent

At each time step the agent needs to select an action by the behavior policy. For a given state it can move using 5 standard actions: stay, move up, move down, move right and move left. Two different algorithm are implemented: Q-learning and SARSA. In general, given a state s_t, to choose the action a_t the row of the Q-table corresponding to s_t is selected. After that the greedy approach involves picking the greedy action, i.e. the one with maximum value in the row: $\text{argmax}_a Q(s_t|a)$. This approach can present some troubles because the greedy policy is deterministic, thus it can prevents a full exploration of the possible strategies not finding the best one. Another approach entails to select the action from a probability distribution, which has a maximum for $\text{argmax}_a Q(s_t|a)$, but that with a certain probability can pick all the other possible actions. Two possible **stochastic behavior policies** $\pi(a_t|s_t)$ are tried: ϵ-greedy and softmax. In case of ϵ-greedy, the action corresponding to the element with maximum value in the row is selected with probability 1-ϵ, while the others with probability ϵ. In case of the softmax the action is selected according to the function:

$$\pi(a_t|s_t) = \Pr(\text{choose } A = a_t | S = s_t) = \frac{\exp Q(s,a)/\epsilon}{\sum_{a^*} \exp Q(s,a^*)/\epsilon} .$$

From one side ϵ-greedy fixes equal probability of an exploration between all actions which are not $\text{argmax}_a Q(s_t|a)$, on the other side the softmax takes into account the Q-values of all the actions to assign them their probabilities of choice. The difference between SARSA and Q-learning can be appreciate in the Q-update,which we write in general as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \Delta Q(s_t, a_t).$$

If we consider r_t, the reward received when moving from the state s_t to the state s_{t+1}, γ, the discount factor, and α the learning rate, in Q-learning algorithm we have:

$$\Delta Q(s_t, a_t) \leftarrow \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)].$$

We can see that to compute the long-term reward with bootstrap method we consider the greedy update policy $\max_a Q(s_{t+1}, a)$, which is different from the stochastic behavior policy $\pi(a_t|s_t)$ used to select the action a_t. This difference determines the name **off-policy** algorithm.

In the SARSA algorithm we have the following Q-update rule:

$$\Delta Q(s_t, a_t) \leftarrow \alpha\,[r_t + \gamma\, Q(s_{t+1}, a_{\pi(s_{t+1})}) - Q(s_t, a_t)].$$

We can see that the policy used to select a_t and a_{t+1} is the same, i.e. the update policy and the behavior policy are equal. For this reason the algorithm is called **on-policy**.

2.3 Training

In the script Agent.py but outside the class Agent the training function can be found. Fixed the number of episodes, i.e. in our case the number of matches, for each of them we set the environment in a starting state, which is random, as long as it's not the goal or a wall cell. Each game has a fixed number of steps and at the end the mean reward per step for each match is computed and saved in a list, which is the output of the function. Has to be remembered that the mean reward per step depends both from the learning (mainly) and from the "distance" which is necessary to be traveled in the optimal path to reach the goal. Because this last cause depends on the starting point, which is set randomly, for each algorithm and for each stochastic behavior policy we run 50 iterations averaging the obtained mean rewards.

2.4 Results

As mentioned above, we tried two algorithms, i.e. Q-learning and SARSA, with two possible stochastic behavior policies: ϵ-greedy and softmax. We also tried different schedules for ϵ parameter in order to try to manage the exploration-exploitation trade-off.

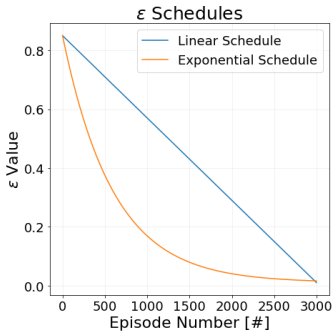

Figure 2: Linear and exponential ϵ schedules behaviour.

For both the schedules the starting ϵ value in the first episode is $\epsilon_0 = 0.85$ and the final one, used in the last episode, is $\epsilon_T = 0.01$, where we called T the number of episodes. The first tried schedule is a vanilla linear decrease, while the second one is an exponential decrease:

$$\epsilon(t) = \epsilon_T + (\epsilon_0 - \epsilon_T) \exp\left[-\lambda \frac{t}{T}\right]$$

where the temperature parameter λ controls the "speed" of the decay and after some trials is fixed to $\hat{\lambda} = 5$. In the following figures 3 and 4 the obtained results for all the final configurations are shown.

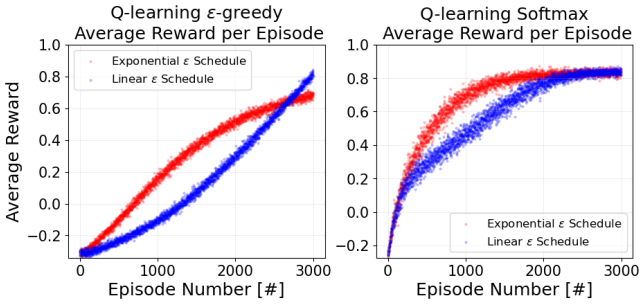

Figure 3: Q-learning algorithm results for different stochastic behavior policies and ϵ schedules.

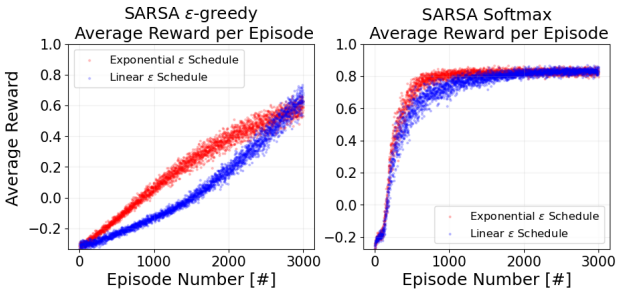

Figure 4: SARSA algorithm results for different stochastic behavior policies and ϵ schedules.

Looking at the plots can be noticed that Q-learning algorithm performs better in the case of ϵ-greedy behavior policy w.r.t. SARSA with an higher average rewards in the last episodes. In general SARSA algorithm with softmax behavior policy and exponential ϵ schedule seems the best tried solution. Moreover can

be noticed that both softmax behavior policy and exponential ϵ schedules increase in all the cases the speed of convergence. Both those behaviours are in a certain way expected: softmax behavior policy allows to choose the "explorative" actions on a probabilistic basis and not just randomly as in the case of ϵ-greedy. Also the exponential ϵ schedules, reducing before the probability of picking an "explorative" action, allows the training to converge faster. Since the problem is quite simple, limiting the exploration has not an impact on the quality of the found solutions. In figures 5 a random obtained solution is shown just for the purpose of visualization. It is not important the configuration which led to this solution because it was verified that, after the termination of the training and set the starting point, all of them find the same path.

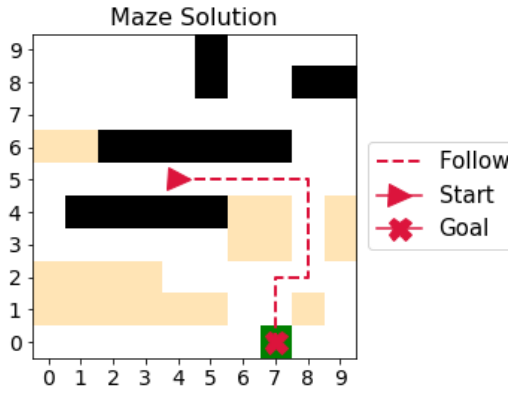

Figure 5: Obtained solution after training for a random starting point.