Q-learning vs SARSA

September 18, 2017

After studding Sutton's book I found out that I haven't really understood the difference between Q-learning and SARSA. Since those are the fundamental TD-based algorithm, to clarify my doubts I decided to implement them with some toy environments.

Maze Environment

To run my experiments I decided to implement a simple 2-D Maze environment formed by a 8×4 matrix, where the agent start in position $[0, 0]$ and the goal is to reach the position $[7, 0]$. To make the environment more challenging; the cells of the first row, which separate the agent form the goal, are filled with holes. Scope of the agent is to learn how to correctly avoid the holes to reach the goal's position. A graphical representation of the environment is given in Fig. 1.

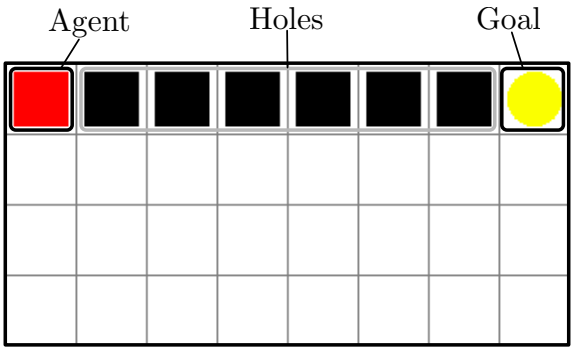


Figure 1: Maze environments

Q-learning

First of all I will do a brief recap of the theory. The Q-function is an alternative notation of the Value function (V) which assign to each state s a corresponding

value to all the actions a that we can perform. This value represent the expected cumulative return that the agent would obtain form s following a policy π. Formally, the Q-function w.r.t. a stochastic policy $\pi(s) = P(a_t|s_t = s)$ is defined as an instance of the Bellman equation:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')[\sum_{a' \in A} \pi(s')Q^{\pi}(s',a')] \tag{1}$$

where S is the set of states, A is the set of possible actions, γ is a discount factor and $T(s,a,s')$ is the state transition probability (or dynamic of the system).

The objective of learning a Q-function (or V-function) is to derive a optimal policy $\pi*$. Supposing that we have learned an optimal Q-function Q^* we can easily derive $\pi*$ as:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s,a) \tag{2}$$

Respect to a Value iterative methods Q-learning doesn't assume to know in advance neither the reward function nor the transition probability, but it is able of derive them form the data. This is partially achieve trough the temporal difference learning (TD-learning), which is similar, but more efficient than a Monte Carlo method. In Monte Carlo we need to wait until the end of a simulation to update the expected return of a state, while in TD-learning at each step we make a prediction of the expected return obtained at the current state, then we adjust this prediction with the estimate of the next state's expected return. Formally, the TD-error is defined as:

$$TD(s,a,s',a') = R_{t+1} + \gamma Q(s',a') - Q(s,a) \tag{3}$$

Q-learning exploit equation Eq. 3 to learn an optimal Q-function Q^*. We can describe the Q-learning algorithm with the following procedure:

1. Assuming that the agent is in a state s, we follow an ϵ-greedy policy π_ϵ defined as:

$$\begin{aligned} &\text{with probability } \epsilon : \text{choose an action at random} \\ &\text{with probability } 1-\epsilon : a = \arg\max_a Q(s,-) \end{aligned} \tag{4}$$

2. Obtained an action $a = \pi_\epsilon(s)$; we can observe the next state s' and the reward $R(s,a)$, just performing a step in the environment.

$$s', R(s,a) = env.step(s,a) \tag{5}$$

3. The agent can now compute the TD-error. It can update the predicted expected return ($Q(s,a)$) using the expected return obtained at the next step ($Q(s',a')$). It as to be noted that $Q(s',a')$ is estimated using a different policy form the π_ϵ. Q-learning assume that the policy followed at each TD-learning step is optimal (π^*). Formally we can define the updating rules as:

$$\pi^*(s) = max_{a \in A} Q(s,-) \tag{6}$$

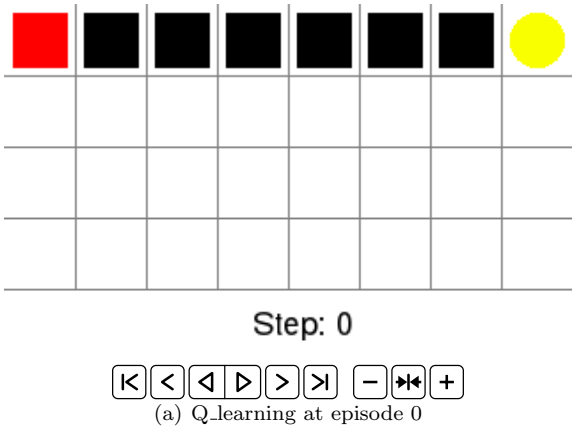
$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max_{a' \in A} Q(s',a') - Q(s,a)] \tag{7}$$

where α is the learning rate used to update the error.

Experiments

Implementing this algorithm is quite trivial. I had setup the environment to give as reward 1 if we obtain the goal, -1 if we end up in a holes, 0 otherwise. Moreover, the agent use $\alpha = 0.1$, $\epsilon = 0.1$ and $\gamma = 0.9$.

Fig. 2 shows how the agent perform before and after training. Initially the algorithm does not know anything, so it is exploring the environment randomly and eventually fall in a hole (Fig. 2(a)). After few minutes of training on my laptop, the agent found out which is the optimal path to follow (Fig. 2(b)), but due to the greedy policy it still can fall in the hole when it is randomly exploring (Fig. 2(c)). Overall, the behavior of the agent is exactly as expected, validating my code and my understanding.



(a) Q_learning at episode 0

(b) Q_learning trained

(c) Q_learning trained, but faulty due to the π_ϵ random choice

Figure 2: Q_learning experiments

SARSA

SARSA, like Q-learning, is an other TD-based learning algorithm. The only difference w.r.t. Q-learning is that SARSA follow the same π_ϵ policy in both cases: the action selection ($a = \pi_\epsilon(s)$) and the computation of expected reward at the next step ($Q(s', a')$).

In my understanding, this small difference means that with probability ϵ, the agent can choose a random $Q(s', a')$ instead of the optimal one. At this point I'm wandering why we should do that and in the experiment section something surprisingly is happening.

Experiments

I keep the same settings of Q-learing's experiments. Like before the initial episode is quite disappointing. In Fig. 3 the agent end directly in a holes.

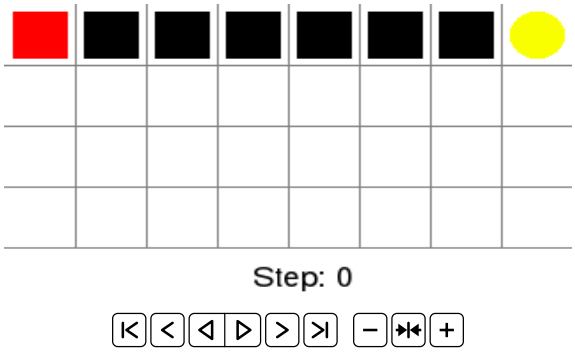


Figure 3: SARSA at episode 0

After some minutes of training I was surprised form the results. I expected to obtain a behaviour similar to the Q-learning once. Instead Fig. 4 shows something interesting, the agent learned that it could fail in a hole due to its ϵ-greedy policy; so it took a longer but safer path.

In an extreme case, if we set $\epsilon = 0$, SARSA and Q-learning should be really similar due to the following derivation:

$$\begin{aligned} \pi_\epsilon(s) &= \arg\max_{a \in A} Q(s, -) \\ Q(s, a) &= Q(s, a) + \alpha[R(s, a) + \gamma Q(s', \pi_\epsilon(s')) - Q(s, a)] \\ &= Q(s, a) + \alpha[R(s, a) + \gamma \max_{a' \in A} Q(s', a') - Q(s, a)] \end{aligned} \tag{8}$$

and I'm wandering how a small difference in the policies can cause such a huge difference in the behaviours and potentially this rais the question on why we need off-policy algorithm at all. In the end, SARSA achieve the same goal of Q-learning, but it is much more robust. Not only it is more robust, but, as

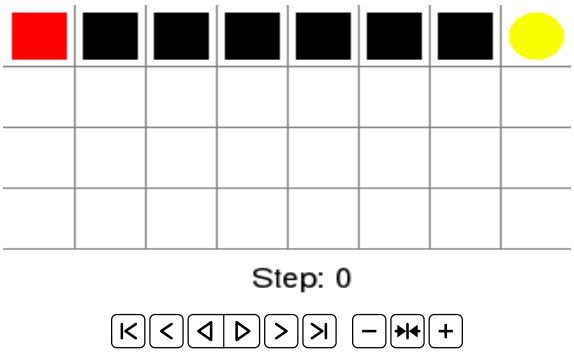
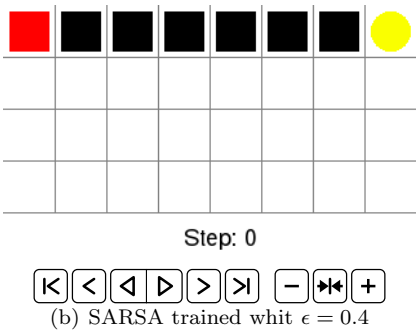


Figure 4: SARSA trained

shown in Fig. 5(a), it also learned that it is unlikely to take two random action one after the other. This hypothesis is also supported by the experiment in Fig. 5(b) where the algorithm is trained with $\epsilon = 0.4$. In this case, due to the high probability of taking a random choice, the agent learned to take the longest and safer path.

(a) SARSA trained, robust to errors



(b) SARSA trained whit $\epsilon = 0.4$