Introduction to Reinforcement Learning II

ML on quantum and classical data

Gorka Muñoz-Gil (**ICFO**)

Master in Photonics (UPC-UAB-UB-ICFO), 2018-2019

Recap

Reinforcement Learning: interaction between agent and environment. Agent does action, receives states and rewards from the environment.

Our goal is to find a policy that allows the agent to perfom a task over the environment.

The policies were based on maximizing value functions (expected reward over long run). We had value functions for states ($V(s)$ in tic-tac-toe) and for actions ($Q(a)$ in MAB).

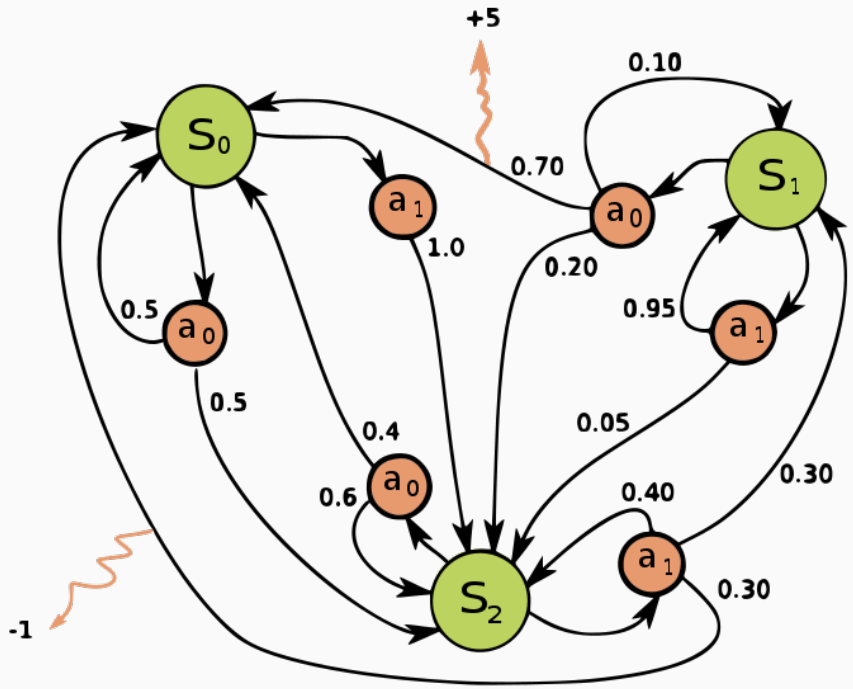
See *Reinforcement Learning: An Introduction*, A. Barto and R. S. Sutton (last edition 2017), *http://incompleteideas.net/book/bookdraft2017nov5.pdf*.

Markov decision process

MAB: the action was done over the same *invariant* system. What happens when our actions affect the system?

$\rightarrow$ We will need to learn how to choose different actions in different situations! Namely: $q_*(a) \rightarrow q_*(s, a)$

Before that, what is a **MDP**?



Markov decision process

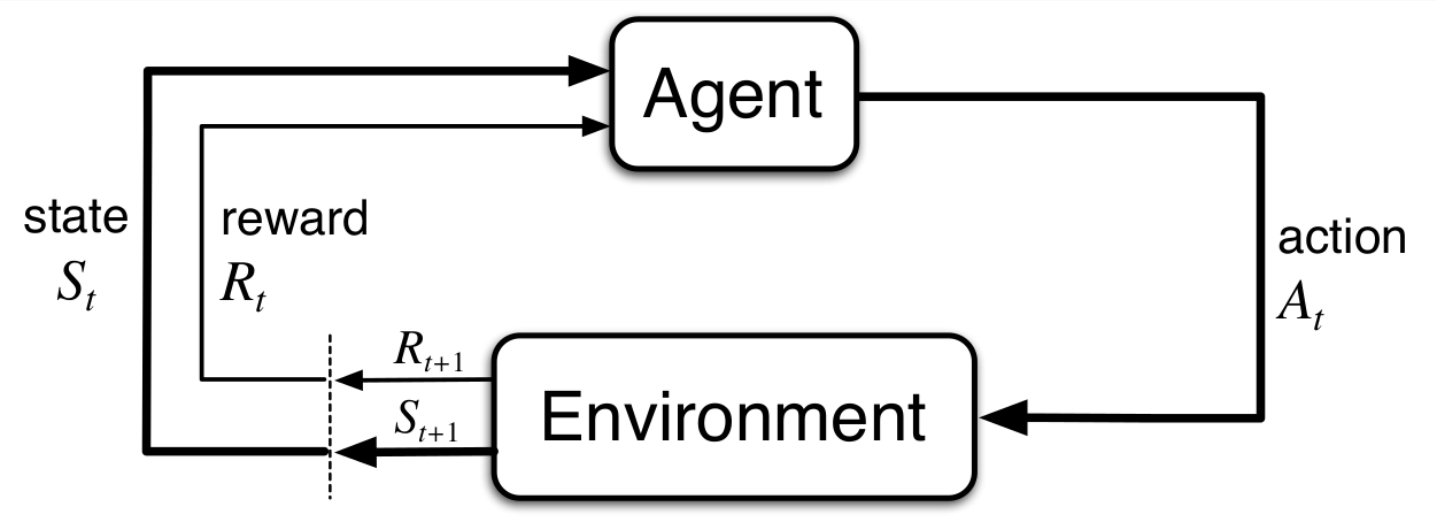
Elements of MPD

Formally, an MDP is an Markovian process ('no memory') define by a 5 elements:

- Finite set of states S.
- Finite set of actions A.
- Set of probabilities $P_a(s, s')$
- Set of rewards $R_a(s, s')$
- γ discount factor, which represents the difference in importance between future and past.

Markov decision process

In terms of RL, we treat with systems like



To keep track of learning, we will use the tuple $(S_t, A_t, S_{t+1}, R_{t+1})$.

We will consider *finite* MDP: finite set of states, actions and rewards.

Markov decision process

The MDP is completely characterized by the probability

$$p(s', r|s, a) = \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

See that $\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1$. From here one can set interesting probabilistic relations such as the *expected reward for state-action pairs*,

$$r(s, a) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

In general, MDPs allow for a simple and elegant representation of our real world problem. The only requirement is to draw the correct agent-environment system.

Goals and Rewards

In MDPs, the goal is to maximize the expected value of the cumulative sum of received rewards.

Rewards can be either positive, negative (usually referred as penalization) or zero.

Example

Positive reward: when making a robot learn to walk, reward is proportional to robot's forward motion.

Penalization: when making a robot learn to escape a maze as faster as possible, each step in maze give reward -1.

Returns and Episodes

As in the MAB, we want to maximize the *expected return*

$$G_t = R_{t+1} + R_{t+2} + ... + R_T$$

We consider the agent-environment interaction broken into *episodes* of lenght T. Normally they all start in same initial state.

Discounting: we may want to choose A_t to maximize the expected *discounted return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = R_{t+1} + \gamma G_{t+1} = \sum_{k=0}^{T-1} \gamma^k R_{t+k+1}$$

Markov decision process

Policies and value functions

Policy $\pi(a|s)$: mapping from states to probabilities of selecting each possible action. Is the 'brain' of the agent.

Value function $v_\pi(s)$: is the expected return when starting in state s and follow policy π,

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

The value of taking action a in state s, the *action-value function*, under policy π is defined as

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a]$$

Optimal policies and value functions

Policy π is better than policy π' (also written as $\pi > \pi'$) if and only if $v_\pi(s) > v_{\pi'}(s)\ \forall\ s$. We denote the optimal policy as π_*. We then have

$$\boxed{q_* = \max_\pi q_\pi(s, a)} \qquad \boxed{v_* = \max_\pi v_\pi(s)}$$

Both functions follow the *Bellman optimality equation*:

$$v_*(s) = \max_a \sum_{s', r} p(s', r|s, a)[r + \gamma v_*(s')]$$

$$q_*(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')]$$

Q-learning (Watkins, 1989)

It is one of the easiest ways for estimating $\pi \approx \pi_*$. It creates a action-value function Q that directly approximates q_* without directly following any policy.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

It follows the algorithm

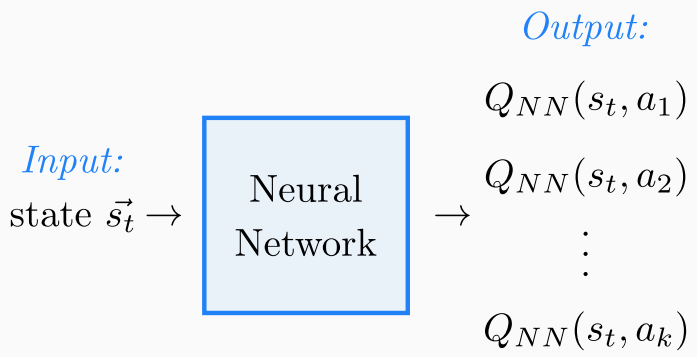
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Let's take a closer look to this algorithm. See colab for the rest.

Q-learning: Q-networks

When the environment is too big or even continuous, the Q-table may be too big to train efficiently. Then, we use *neural networks* instead of tables. The network will work as follows:



Then, we want Q_{NN} to approximate q_*. To do so we will use the following loss function:

$$\text{loss} = (\underbrace{r_t + \gamma \max_{a'} Q_{NN}(s_{t+1}, a'_{t+1})}_{\textit{target}} - \underbrace{Q_{NN}(s_t, a_t)}_{\textit{prediction}})$$

See that if the discount factor $\gamma = 0$, we exactly train the NN to output the expected reward r_t for each of the actions $a_1, a_2, ..., a_k$.