Introduction to Reinforcement Learning

Jae Yun JUN KIM*

October 10, 2020

Reinforcement learning models the world using the **Markov Decision Process (MDP)** formalism: $MDP(S, A, \{P_{sa}\}, \gamma, R)$, where

- S: set of states
- A: set of actions
- $\{P_{sa}\}$: state transition distributions with

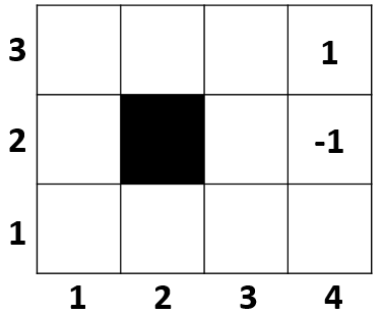
$$\sum_{s'} P_{sa}(s') = 1, \qquad P_{sa}(s') \geq 0$$

  P_{sa} gives the probability distribution of ending up at state s' when the actiona a is performed from the state s:

$$s \xrightarrow{a} s'$$

- γ: discount factor, where $0 \leq \gamma < 1$
- R: reward function $R : s \to \mathbb{R}$

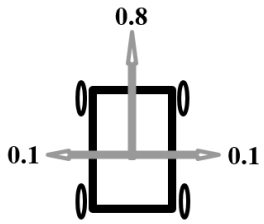
Example:



- 11 states (11 cells)
- 4 actions: $A = \{N, S, E, W\}$
- Because the system is noisy, the action is modeled as This is a very crude acion model. That is, for example

$$P_{(3,1),N}((3,2)) = 0.8$$
$$P_{(3,1),N}((4,1)) = 0.1$$
$$P_{(3,1),N}((2,1)) = 0.1$$
$$P_{(3,1),N}((3,3)) = 0.0$$
$$\vdots$$

*ECE Paris Graduate School of Engineering, 37 quai de Grenelle 75015 Paris, France; jae-yun.jun-kim@ece.fr



- Rewards:

$$R = \begin{cases} 1, & \text{cell} = (4,3) \\ -1, & \text{cell} = (4,2) \\ -0.02, & \text{otherwise} \end{cases}$$

  The reward values corresponding to any cell different from (4,3) and (4,2) can be interpreted as battery (or time) consumption before arriving to the goal cell.

- Stop condition: finish the algorithm when the robot hits either the cell of $R = 1$ or that of $R = -1$.

How does the MDP work?
At state s_0, choose a_0. Then get to $s_1 \sim P_{s_0 a_0}$. Afterwards, choose a_1, then get to $s_2 \sim P_{s1a_1}$. And, so on.
To evaluate how well the robot did by visiting the states s_0, s_1, s_2, etc.:

- define the reward function
- apply it to the sequence of states
- add up the sum of the rewards obtained along the sequence of the states that the robot visits. But, we do this in a discounted manner.

Define the cumulative discounted reward or the total pay-off:

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \tag{1}$$

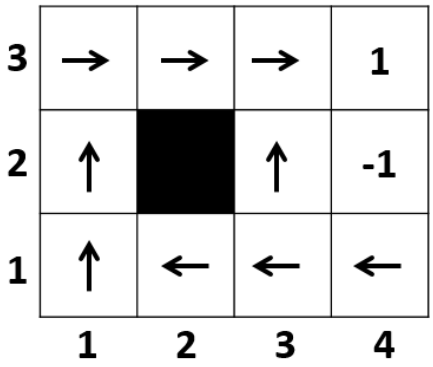
Since $0 \leq \gamma < 1$, the reward that we get at time t_1 is slightly smaller than the reward obtained at time t_0. And, the reward that we get at time t_2 is weighted such that it is even smaller than that of t_1. And so on.
If this is an economic application, then the reward is either the earning or the loss. Then, γ has a natural interpretation of time-valued money. That is, a dollar of today is slightly more worthy that a dollar of tomorrow. Because of the added bank interest, a dollar in the bank adds a little bit of interest. Conversely, having to pay out a dollar tomorrow is better than having to pay out a dollar today.
In other words, the effect of the discount factor tends to weight wins and losses in the future less than to weight immediate wins and losses.
The goal of the reinforcement learning is

- to choose actions over time $(a_0, a_1, \cdots)$
- to maximize the expected value of this total pay-off (i.e., the cumulative discounted reward):

$$\mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots].$$

More concretely, we want our reinforcement learning algorithm to compute a **policy**: $\pi : S \longrightarrow A$. A **policy** is a function that maps the state space to action space. That is, a policy is a function that recommends what action one should take for a given state.

Example: Policy



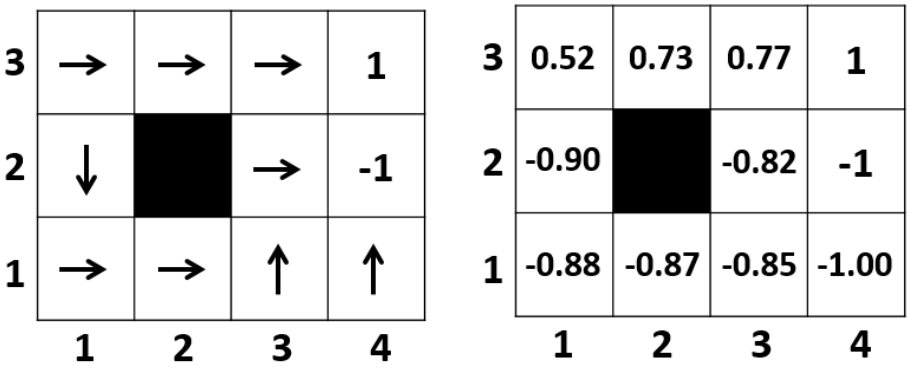
1 Value function

For any π, define **value function** $v^\pi : S \to \mathbb{R}$ such that $V^\pi(s)$ is the expected total pay-off starting in state s, and execute π.

$$V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \cdots | \pi, s_0 = s] \tag{2}$$

This is an expression loosely written because π is not actually a random variable, but this expression is commonly used.

Example: Policy and value function



(a) Policy (b) Value function

In this example, we see that the actions corresponding to the bottom two rows are bad because there is a high chance that the robot will end up in the cell with -1 reward. We can recognize this fact by looking at the negative values of the value function corresponding to the bottom two rows. This means that the expected value of the total pay-off is negative if we are at any state in these rows.

1.1 The Bellman's equations

The *value function* can also be expressed recursively as follows:

$$V^\pi(s) = \mathbb{E}[R(s_0) + \gamma(R(s_1) + \gamma R(s_2) + \cdots) | \pi, s_0 = s] \tag{3}$$

By mapping $s_0 \to s$ and $s_1 \to s'$,

$$V^\pi(s) = R(s) + \gamma \sum_{s'} P_{s\pi(s)}(s') V^\pi(s') \tag{4}$$

This equation is known as the **Bellman's equations**.
It urns out that the Bellman's equations give a way to solve the value function for a given policy in closed form.
Now the question is, for a given policy π, how do I find its value function $V^{\pi}(s)$?
By looking at the Bellman's equations, we realize that if I want to solve the value function for a given policy π, then see see that the Bellman's equations impose constraints on the value function.
The value function depends on some constant $R(s)$ and a linear function of some other values. So for any state in the MDP, we can write such an equation, and this imposes a set of linear constraints on what the value function could be. And, by solving this system of linear functions, one can finally solve the value function $V^{\pi}(s)$.

Example: Value function computation. If $\pi((3,1))=N$,

$$V^{\pi}((3,1)) = R((3,1)) + \gamma\left[0.8V^{\pi}((3,2)) + 0.1V^{\pi}((4,1)) + 0.1V^{\pi}((2,1))\right]$$

From the above example, we identify the unknown value function. We know that we have 11 unknowns (one value-function value for each state) and 11 equations (one equation for each state). Hence, we can solve uniquely the value function.

2 Optimal value function and optimal policy function

2.1 Optimal value function

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \tag{5}$$

For any given state s, the optimal value function says that when I suppose that I take the maximum over all possible policies, what is the best expected value of the total pay-off (i.e., the sum of the discounted reward).
The version of the Bellman's equations for $V^{*}(s)$ is

$$V^{*}(s) = R(s) + \max_{a} \gamma \sum_{s'} P_{sa}(s')V^{*}(s') \tag{6}$$

This equation tells us that the optimal expected pay-off is the immediate reward plus the future optimal expected pay-off by taking the best actions.

2.2 Optimal policy function

$$\pi^{*}(s) = \arg\max_{a} \sum_{s'} P_{sa}(s')V^{*}(s') \tag{7}$$

Hence, the optimal policy is the action that maximizes the future expected pay-off.
Then, how can we find π^{*}?
The strategy will be finding first $V^{*}(s)$ and then using the definition of the optimal policy, find $\pi^{*}(s)$. But, the definition of $V^{*}(s)$ does not lead to a nice algorithm for computing it. On the other hand, I know how to compute $V^{\pi}(s)$ for any state for a given π by solving a linear system of equations. But, there is an exponentially large number of policies. If 11 states and 4 actions are considered, then there can be $4^{1}1$ policies!. Hence, I can not use the brute force method to find the policy that maximizes the value function. Therefore, we need an efficient intelligent algorithm to find V^{*} and compute π^{*} afterwards.

3 Value iteration algorithm

Algorithm 1: Value iteration

Initialize $V(s) = 0 \qquad \forall s$;
while *until convergence* **do**
  For every s, update $V(s) \leftarrow R(s) + \max_a \gamma \sum_{s'} P_{sa}(s')V(s')$;
return $V(s)$;

This will make $V(s) \longrightarrow V^*(s)$. And, using the definition of $\pi^*(s)$, one can compute the optimal policy from $V^*(s)$. There are two ways of implementing this algorithm.

3.1 Synchronous update

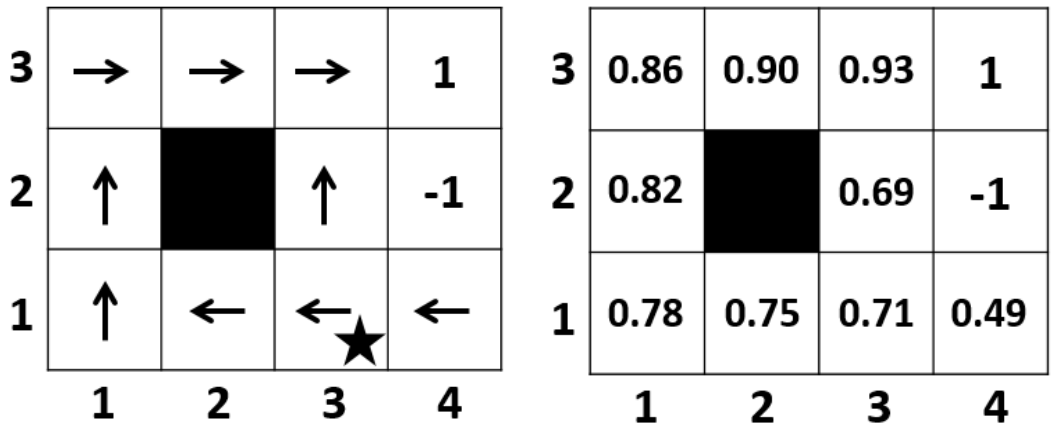
$$\forall s, \quad V(s) \leftarrow B(V(s))$$

where $B()$ is the Bellman back-up operator. One calculates the RHS values for all states. And update the value function all at once.

3.2 Asynchronous update

Update $V(s)$ one by one. Hence, once one value function is updated for a state, this new value is used to the value functions that depend on it, in the same optimization iteration.

Note: Asynchronous update is usually a bit faster than the synchronous update. But, it is easier to analyze with the synchronous update approach.
Example:



(c) Policy (d) Value function

At ★,

$$\begin{aligned} \text{W} : \sum_{s'} P_{sa}(s')V^*(s') &= 0.8 \times 0.75 + 0.1 \times 0.69 + 0.1 \times 0.71 = 0.740 \\ \text{N} : \sum_{s'} P_{sa}(s')V^*(s') &= 0.8 \times 0.69 + 0.1 \times 0.75 + 0.1 \times 0.49 = 0.676 \end{aligned} \tag{8}$$

Hence, it is recommendable to go to W rather than to N.

4 Policy iteration algorithm

Algorithm 2: Value iteration

Initialize π randomly ;
while *until convergence* **do**
    Let $V \leftarrow V^{\pi}$ (i.e., solve the Bellman's equations) ;
    Let $\pi(s) \leftarrow \arg\max_a \sum_{s'} P_{sa}(s')V(s')$;
return $V(s), \pi(s)$;

Then, it turns out that $V \longrightarrow V^*$ and $\pi \longrightarrow \pi^*$.
In the *policy iteration algorithm*, the time consuming step is solving the Bellman's equations.

Note:
If the number of states is less than 1000, then the PI is preferred, but if the number of states is more than this value, then the VI is preferred.

5 Miscellaneous

What if you do not know P_{sa}?

- S, A: you almost always know
- γ: you choose it
- R: You usually know
- P_{sa}: sometimes hard to be found/computed

We can estimate $P_{sa}(s')$ as follows:

$$P_{sa}(s') = \frac{\text{number of times took action “a” in state s and got to s'}}{\text{number of times took action “a” in state s}}$$

If $P_{sa}(s') = \frac{0}{0}$ and assign it to be $\frac{1}{|S|}$.

6 Summary

Algorithm 3: Reinforcement learning

Initialize π randomly;
while *until convergence* **do**
    Take action using π to get experience in MDP ;
    Update estimates of P_{sa};
    Solve the Bellman's equations using VI to get V;
    Update $\pi(s) \leftarrow \arg\max_a \sum_{s'} P_{sa}(s')V(s')$
return $V(s), \pi(s)$;