Introduction to Intelligent Systems / Lecture 6

Introduction to Reinforcement Learning

Jae Yun JUN KIM*

July 14, 2019

References: Sutton and Barto's Reinforcement Learning book (2nd edition), Wikipedia

Reinforcement learning (RL) is an area of machine learning concerned with how software agents should take actions in an environment so as to maximize some notion of cumulative reward. The problem addressed by reinforcement learning is also studied in many other disciplines such as game theory, control theory, operations research, information theory, multi-agent systems, swarm intelligence, statistics and genetic algorithms. In operations research and control literature, reinforcement learning is called *approximate dynamic programming* or *neuro-dynamic programming*. In machine learning, the environment is typically formulated as a Markov Decision Process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP, and they target large MDPs where exact methods become infeasible.

Reinforcement learning is considered as one of three machine learning paradigms, alongside supervised learning and unsupervised learning. It differs from supervised learning in that correct input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead, the focus is on performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

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

1 Comparison between different reinforcement learning algorithms

Algorithm	Description	Model	Policy	Action Space	State Space	Operator
Monte Carlo	Every visit to Monte Carlo	Model-Free	Off-policy	Discrete	Discrete	Sample-means
Q-learning	State–action–reward–state	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA	State–action–reward–state–action	Model-Free	On-policy	Discrete	Discrete	Q-value
Q-learning - Lambda	State–action–reward–state with eligibility traces	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA - Lambda	State–action–reward–state–action with eligibility traces	Model-Free	On-policy	Discrete	Discrete	Q-value
DQN	Deep Q Network	Model-Free	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	Model-Free	Off-policy	Continuous	Continuous	Q-value
NAF	Q-Learning with Normalized Advantage Functions	Model-Free	Off-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage

Figure 1: Source: Wikipedia

2 Markov Decision Process (MDP)

Basic reinforcement learning is modeled as a Markov Decision Process (MDP):

- 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 $R : s \to \mathbb{R}$

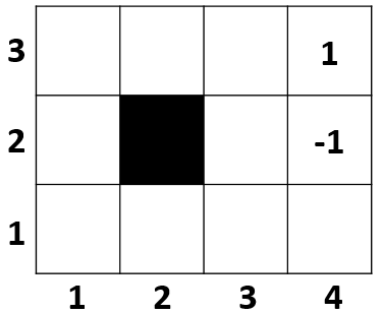
The rules are often stochastic. The observation typically involves the scalar, immediate reward associated with the last transition. The agent is often assumed to observe the current environmental state (full observability). If not, the agent has partial observability. Sometimes the set of actions available to the agent is restricted.

A reinforcement learning agent interacts with its environment in discrete time steps. At each time t, the agent receives an observation o_t, which typically includes reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined. The goal of a reinforcement learning is to collect as much reward as possible. The agent can (possibly randomly) choose any action as a function of the history.

When the agent's performance is compared to that of an agent that acts optimally, the difference in performance gives rise to the notion of regret. In order to act near optimally, the agent must reason about the long term consequences of its actions (i.e., maximize future income), although the immediate reward associated with this might be negative. Thus, reinforcement learning is particularly well-suited to problems that include a long-term versus short-term reward trade-off.

Two elements make reinforcement learning powerful: the use of samples to optimize performance and the use of function approximation to deal with large environments.

2.1 Example



- 11 states (11 cells)
- 4 actions: $A = \{N, S, E, W\}$
- Because the system is noisy, the action is modeled as

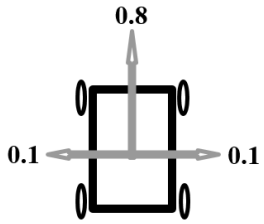


Figure 2: Action model

This is a very crude action model. That is, for example

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

- 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 $R = -1$.

3 The goal of reinforcement learning

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 $s_2 \sim P_{s_1 a_1}$. 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:

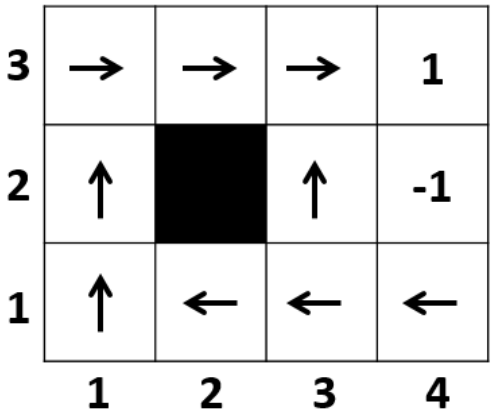


Figure 3: Example 2

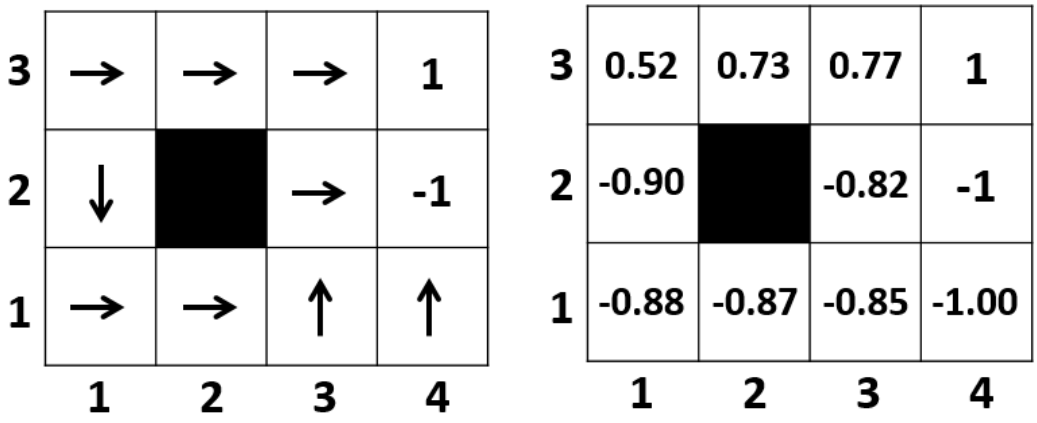
4 (State-)Value function

For any π, define **(state-)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:



(a) Policy (b) Value function

Figure 4: Policy and 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.

4.1 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:

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

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.

5 Optimal value function and optimal policy function

5.1 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.

5.2 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 4^11 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.

6 Value iteration algorithm

Algorithm 1: Value iteration

1 Initialize $V(s) = 0 \qquad \forall s$;
2 **while** *until convergence* **do**
3 For every s, update $V(s) \leftarrow R(s) + \max_a \gamma \sum_{s'} P_{sa}(s')V(s')$;
4 **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.

6.1 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.

6.2 Asynchronous update

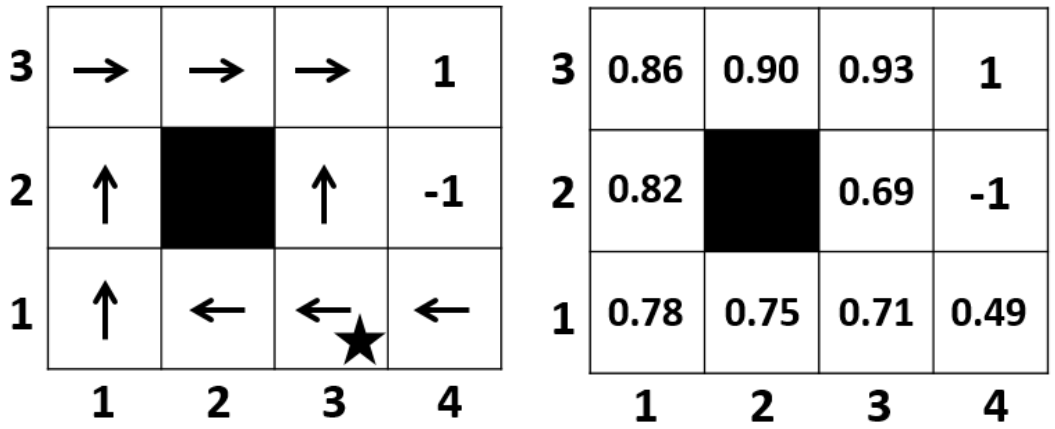
Update $V(s)$ one by one. Hence, once one value function is updated for a state, this new value is used to the value functions that depend on it, in the same optimization iteration.

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

Example:
At ★,

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

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



(a) Policy (b) Value function

Figure 5: Optimal policy and optimal value function

7 Policy iteration algorithm

Algorithm 2: Value iteration

```
Initialize π randomly ;
while until convergence do
    Let V ← V^π (i.e., solve the Bellman's equations) ;
    Let π(s) ← arg max_a Σ_{s'} P_sa(s')V(s');
return V(s), π(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.

8 Exploration

Reinforcement learning requires clever exploration mechanisms. Randomly selecting actions, without reference to an estimated probability distribution, shows poor performance. The case of (small) finite MDP is relatively well understood. However, due to the lack of algorithms that properly scale well with the number of states (or scale to problems with infinite state spaces), simple exploration methods are the most practical.

One such method is **ϵ-greedy**, when the agent chooses the action that it believes has the best long-term effect with probability $1 - \epsilon$. If no action which satisfies this condition is found, the agent chooses an action uniformly at random. Here, $0 < \epsilon < 1$ is a tuning parameter, which is sometimes changed, either according to a fixed schedule (making the agent explore progressively less), or adaptively based on heuristics.

9 Q-learning

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation of "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

For any finite MDP (FMDP), Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state.

9.1 Algorithm

The algorithm calculates the quality of a state-action combination through the (state-action)value function called **Q function**:

$$Q : S \times A \longrightarrow \mathbb{R}.$$

Before learning begins, the Q function is initialized to arbitrary or fixed values. Then, at each time t, the agent selects an action a_t (either following a policy function (π) or chosen randomly), observes a reward r_t, enters a new state s_{t+1} (that may depend on both the previous state s_t and the selected action). Afterwards, Q is updated. The core of the algorithm is a simple **value iteration update**, using the weighted average of the old value and the new information:

$$Q\left(s_t, a_t\right) \longleftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a)\right),$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning rate ($0 < \alpha \leq 1$).

An episode of the algorithm ends when state s_{t+1} is a final or *terminal state*. However, Q-learning can also learn in non-episodic tasks. If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

For all final states s_f, $Q(s_f, a)$ is never updated, but is set to the reward value r observed for state s_f. In most cases, $Q(s_f, a)$ can be taken to equal zero.

Algorithm 3: (state-action-)Value iteration

```
Initialize Q(s, a) = 0        ∀s,  ∀a;
while until convergence do
    For every s_t and a_t, update
      Q(s_t, a_t) ⟵ (1−α) · Q(s_t, a_t) + α (r_t + γ max_a Q(s_{t+1}, a));
return Q;
```

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