Reinforcement learning

In this chapter, we discuss the technique called reinforcement learning. First, we examine the basic principles. Then we look at how it is applied when a model of the environment is present. Next, we examine what to do when a model of the environment is missing. Finally, we look at how we can apply reinforcement learning to control problems.

1 Basics of reinforcement learning

1.1 Definitions in reinforcement learning

In **reinforcement learning** (RL), there is an **agent** and an **environment**. The agent has a certain **state** $s_k \in S$. During every step, the agent needs to choose one of the possible **actions** $a_k \in A$. He then reaches a new state s_{k+1}. By doing this, he gets an **immediate reward** $r_k \in \mathbb{R}$ from the environment.

The goal of the agent now is to maximize the **total reward** R_k. This total reward is a function of all future rewards. Often, the sum is used. So, $R_k = r_{k+1} + r_{k+2} + \ldots$. Another often-used function is

$$R_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \ldots = \sum_{n=0}^{\infty} \gamma^n r_{k+n+1}. \tag{1.1}$$

The parameter γ, which satisfies $0 \leq \gamma \leq 1$, is called the **discount rate**. We will use the latter total reward function in the remainder of this chapter.

The whole point of reinforcement learning is to find the optimal **policy**. A policy is a mapping: for every state s, it maps which action a is chosen by the agent in that state. If we can write $a = \pi(s)$, then we deal with a **deterministic policy**: for every state s, always the same action a is chosen. However, we can also deal with a **stochastic policy**. In this case, $\Pi(s, a)$ denotes the probability that in state s action a is chosen by the agent.

1.2 The environment

Let's suppose that the agent is in some state s_k and chooses action a_k. Also, all the previous states and actions $s_{k-1}, a_{k-1}, s_{k-2}, a_{k-2}, \ldots$ are known. In a **stochastic environment**, it is uncertain in which state s_{k+1} the agent winds up in. The probability that the agent reaches state s_{k+1} with reward r_{k+1} is denoted by

$$P\left(s_{k+1}, r_{k+1} | s_k, a_k, s_{k-1}, a_{k-1}, s_{k-2}, a_{k-2}, \ldots\right). \tag{1.2}$$

However, usually we assume that the system has the **Markov property**. This means that the state and reward at time $k+1$ only depends on the state and action at time k. Thus, the above probability is simply written as $P\left(s_{k+1}, r_{k+1} | s_k, a_k\right)$. An RL task which satisfies this property is called a **Markov decision process** (MDP).

Let's discuss some more notations. We denote the chance that the agent winds up in state s', given that he now is in state s and chooses action a, by

$$\mathcal{P}_{ss'}^a = P\left(s_{k+1} = s' | s_k = s, a_k = a\right). \tag{1.3}$$

This function is called the **state transition probability function**. Similarly, we can define the **expected reward** as

$$\mathcal{R}_{ss'}^a = E\left\{r_{k+1} | s_k = s, a_k = a, s_{k+1} = s'\right\}. \tag{1.4}$$

Here, we do have assumed that the agent always knows the state which he is in. If the agent can't always observe the state which he is in, then we are dealing with a **partially observable MDP** (POMDP). We won't deal with POMDP problems though.

1.3 The value function

Let's suppose that we have an agent that is in some state s. This agent also has a policy π. The **value function** $V^\pi(s)$ now is the expected total reward R_k when the policy π is used. So,

$$V^\pi(s) = E^\pi\left\{R_k | s_k = s\right\} = E^\pi\left\{\sum_{n=0}^{\infty} \gamma^n r_{k+n+1} | s_k = s\right\}. \tag{1.5}$$

By the way, E^π is the expectation operator, given that the agent follows the policy π. In a similar way, we can define the **action-value function** $Q^\pi(s,a)$ as the expected total reward R_k when an agent chooses action a in state s and follows policy π afterwards. So,

$$Q^\pi(s,a) = E^\pi\left\{R_k | s_k = s, a_k = a\right\} = E^\pi\left\{\sum_{n=0}^{\infty} \gamma^n r_{k+n+1} | s_k = s, a_k = a\right\}. \tag{1.6}$$

When applying RL, we always use either V or Q, never both. However, sometimes V is convenient to use and sometimes Q. So, in this summary, we will treat them both.

The goal of reinforcement learning is to find an **optimal policy** π^*. This optimal policy π^* is the policy π which maximizes the value function V^π or, alternatively, Q^π. How this policy can be found depends on the type of problem.

2 Model based RL

2.1 The Bellman optimality equation

Sometimes we have an exact model of the environment. Solution techniques to find the optimal policy are now known as **dynamic programming**.

Let's suppose that we are in a state s and choose an action a. If we do this, then there is a chance $\mathcal{P}_{ss'}^a$ that we wind up in state s'. In this state, our expected reward will be the sum of our immediate reward $\mathcal{R}_{ss'}^a$, and of the expected total reward of future states $\gamma V^*(s')$. (Note that a discount rate has to be added.) Based on this, we can find the expected total reward of choosing action a. Of course, we want to choose the action a which maximizes the expected total reward. This logic results in the recursively defined **Bellman optimality equation**

$$V^*(s) = \max_a \sum_{s'} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma V^*(s')\right). \tag{2.1}$$

A similar equation can be derived for Q. We then get

$$Q^*(s,a) = \sum_{s'} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma \max_{a'} Q^*(s',a')\right). \tag{2.2}$$

Solving for the value function can be quite difficult though. So we'll treat that in the next paragraph separately.

You may wonder, when we have the value function V^* (or Q^*), how do we find the optimal policy? Well, in this case the optimal policy is the so-called **greedy policy**. We simply take the action a which maximizes the value function. So,

$$\pi(s) = \arg\max_{a \in A} \sum_{s'} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma V^\pi(s)\right) \qquad \text{or} \qquad \pi(s) = \arg\max_{a \in A} Q^*(s,a). \tag{2.3}$$

2.2 Finding the optimal value function

There are two often-used methods to find the optimal value function. One of them is **policy iteration**. We start with a certain initialization $V_0(s)$ of the value function and with a certain policy π. We then simply iterate.

During every step, there is a **policy evaluation** and a **policy improvement** step. In the policy evaluation step, we use the policy to update the value function. This is done according to

$$V_{n+1}(s) = E^{\pi}\left\{r_{k+1} + \gamma V_n(s_{k+1})\right\} = \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V_n(s')\right), \qquad \text{with} \qquad a = \pi(s). \tag{2.4}$$

In the policy improvement step, we improve our policy. In fact, as policy the greedy policy π is used, corresponding to the value function $V_{n+1}(s)$. These steps are then iterated until a stopping criterion is met. For example, the policy π hasn't changed for several consecutive iterations, or the difference in the value function $V(s)$ is below a certain threshold ϵ.

A similar method is the **value iteration** method. In this method, no policy is computed anymore. Instead, the value function is updated directly using

$$V_{n+1}(s) = \max_a E\left\{r_{k+1} + \gamma V_n(s_{k+1}) | s_k = s, a_k = a\right\} = \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V_n(s')\right). \tag{2.5}$$

3 Model free RL

3.1 Temporal difference methods

It may occur that we don't have any model of our environment. In this case, the agent simply needs to explore it. There are several ways to do this. But most of the methods do use a value function. Among these methods are the **temporal difference** (TD) methods.

Let's suppose that we are in some state s_k. We then go to a state s_{k+1} in which we receive a reward r_{k+1}. We use this reward to update $V(s_k)$. This kind of makes sense: if r_{k+1} is big, then $V(s_k)$ should have been big as well, while if r_{k+1} is small, then $V(s_k)$ should have been small as well. The equation that is used is

$$V(s_k) \leftarrow (1-\alpha_k)V(s_k) + \alpha_k\left(r_{k+1} + \gamma V(s_{k+1})\right) = V(s_k) + \alpha_k\left(r_{k+1} + \gamma V(s_{k+1}) - V(s_k)\right) = V(s_k) + \alpha_k \delta_k. \tag{3.1}$$

In the above equation, α_k is the **learning rate** at time k. Also, $\delta_k = r_{k+1} + \gamma V(s_{k+1}) - V(s_k)$ is the **TD-error**.

You might be wondering, why do we use r_{k+1} to only update s_k. Can't we use r_{k+1} to update $s_{k-1}, s_{k-2}, \ldots$ as well? Well, we can. The question just is: how much should we update them? For this, we define the **eligibility trace** $e_k(s)$. This eligibility trace can be seen as the 'strength' of the relation between the reward r_{k+1} and the state s. If, for example, $s = s_{k-1}$, then there is a relatively strong relation between s and r_{k+1}. So, $e_k(s)$ should be big. On the other hand, if $s = s_{k-20}$, then $e_k(s)$ should be small. So, we can define $e_k(s)$ as

$$e_k(s) = \begin{cases} \gamma\lambda e_{k-1}(s) & \text{if } s \neq s_k, \\ 1 & \text{if } s = s_k. \end{cases} \tag{3.2}$$

The parameter λ is called the **trace-decay parameter**. (γ is still the discount rate.) Based on this eligibility trace, we can update $V(s)$. The change in $V(s)$ (denoted as $\Delta V(s)$) is now given by

$$\Delta V(s) = \alpha \delta_k e_k(s). \tag{3.3}$$

3.2 Q-learning and SARSA

Another model free RL method is Q-learning. It is a so-called **off-policy** method: it doesn't use a policy while learning. Instead, it simply uses the action-value function $Q(s,a)$ to learn. To start, we give the function $Q(s,a)$ initial values. We then update it using

$$Q(s_k,a_k) \to Q(s_k,a_k) + \alpha\left(r_{k+1} + \gamma \max_a Q(s_{k+1},a) - Q(s_k,a_k)\right). \tag{3.4}$$

How does this work? Well, let's suppose that we want to update $Q(s_k,a_k)$. We then start in state s_k, choose action a_k, which brings us in state s_{k+1} with immediate reward r_{k+1}. The new value $Q(s_k,a_k)$ then depends on the old value, the immediate reward r_{k+1} which we received and the maximum expected future reward $Q(s_{k+1},a)$ which we expect to be able to get. However, to make sure that the algorithm converges, we do have to visit all state-action pairs (s_k,a_k) continually.

If also eligibility traces are used, then the above equation turns into

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_k e_k(s,a), \qquad \text{where} \quad \delta_k = r_{k+1} + \gamma \max_{a'} Q(s_{k+1},a') - Q(s_k,a_k). \tag{3.5}$$

Another method, which is somewhat similar to Q-learning, is the **SARSA** method. But contrary to Q-learning, SARSA is an **on-policy** method. That is, it does require a policy π or Π. This time, we update $Q(s_k,a_k)$ using

$$Q(s_k,a_k) \to Q(s_k,a_k) + \alpha\left(r_{k+1} + \gamma Q(s_{k+1},a_{k+1}) - Q(s_k,a_k)\right). \tag{3.6}$$

The action a_{k+1} follows from the policy. So, $a_{k+1} = \pi(s_{k+1})$ or, alternatively, the chance that an action a is chosen to be a_{k+1} is $\Pi(s,a)$.

3.3 Exploration

Previously, we saw that, to apply Q-learning, we need to examine all possible state-action combinations (s_k,a_k). But what do we do if the agent can't choose which state he is in? (That is, if he can only just 'walk' around?) In this case, it would be bad to stick to our policy. Instead, we need to **explore**. And although most of the times an **explorative action** gives a lower reward than the action we would otherwise choose, sometimes it may give a higher reward. And this will result in a better eventual outcome.

There are several ways to explore. However, we will only consider one group of methods, called **undirected exploration**. It simply means that there is a chance that you select a random action. For example, when following an **ϵ-greedy policy**, there is a chance ϵ that you select a random action. In the other cases, you simply follow a normal greedy policy and thus choose the action with the highest $Q(s,a)$ value.

Another type of undirected exploration is **Max-Boltzmann exploration** (also called **soft-max exploration**). Now, the chance that we choose an action a is given by

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}. \tag{3.7}$$

The parameter τ is a variable that determines how much you explore. If $\tau = \infty$, you select actions fully randomly (as if $\epsilon = 1$). But if $\tau = 0$, you are back to the greedy policy.

We could also use **optimistic initial values**. What this means is that we initialize $Q(s,a)$ (or alternatively, $V(s)$) with very high values. We then follow a greedy policy with a normal updating method for Q. So, when you try an action a, the $Q(s,a)$ value will very likely decrease. So the next time you arrive at state s, you will choose a different action. Only when a $Q(s,a)$ value stops to decrease, will you continue to follow the same action. And of course, the first action a for which $Q(s,a)$ stops to decrease is quite likely the best action.

A very interesting question to ask is: how much should you explore? This is called the **exploration vs. exploitation dilemma**. Initially, you should explore quite a bit. But as time progresses, and you are bound to have found some good sets of actions, you should exploit. Thus, when applying an ϵ-greedy policy or a Max-Boltzmann policy, the value of ϵ or τ should decrease over time.

4 Application of RL to control systems

Let's suppose that we have some system which we want to control. How can we use RL for this? The first problem which we run into is that in RL, all states and actions are discrete. But in most control problems, the states $\mathbf{x}$ are continuous. The first step in applying RL is thus the **quantization of state variables**.

The second step which you need to do is define the **action set** A. (That is, the set of all possible actions.) An example of an action might be 'a_1 = apply maximum negative input' and 'a_2 = apply maximum positive input'. But more difficult control laws can also be used, like 'a_1 = use fuzzy controller number 1' and 'a_2 = use fuzzy controller number 2' or something similar.

The third step is to **define the reward function**. What states do we want to reach? (Give these a high reward.) And what states do we definitely want to avoid? (Give those a low reward.) Important when defining the reward function is the rule: 'you should only tell the agent what it should do, and not how.' If you do this, then the RL algorithm might just come up with a very surprising but very effective solution.

Finally, the **results** of the algorithm should be examined. Does the resulted policy control the system sufficiently? If not, what went wrong? Can you fix it by doing the previous steps in a different way?