# Chapter 17
# Reinforcement Learning

The fundamental problem addressed by this book is how to induce a classifier capable of determining the class of an object. We have seen quite a few techniques that have been developed with this in mind. In *reinforcement learning*, though, the task is different. Instead of induction from a set of pre-classified examples, the agent "experiments" with a system, and the system responds to this experimentation with rewards or punishments. The agent then optimizes its behavior, its goal being to maximize the rewards and to minimize the punishments.

This alternative paradigm differs from the classifier-induction task to such an extent that a critic might suggest that reinforcement learning should perhaps be relegated to a different book, perhaps a sequel to this one. The wealth of available material would certainly merit such decision. And yet, the author feels that this textbook would be incomplete without at least a cursory introduction of the basic ideas. Hence this last chapter.

## 17.1 How to Choose the Most Rewarding Action

To establish the terminology, and to convey some early understanding of what reinforcement learning is all about, let us begin with a simplified version of the task at hand.

**N-Armed Bandit** Figure 17.1 shows five slot machines. Each gives a different average return, but we do not know how big these average returns are. If we want to maximize our gains, we need to find out what these average returns are, and then stick with the most promising machine. This is the essence of what machine learning calls the problem of an *N-armed bandit*, alluding to the notorious tendency of the slot machines to rob you of your money.



M. Kubat, *An Introduction to Machine Learning*,
DOI 10.1007/978-3-319-63913-0_17

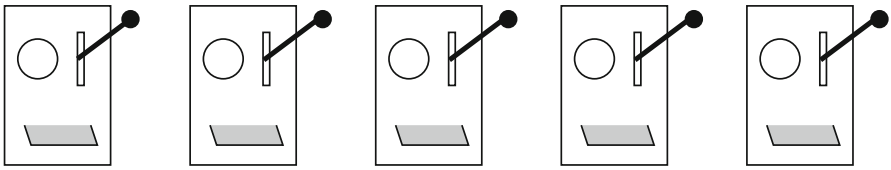

**Fig. 17.1** The generic problem: which of the slot machines offers the highest average return?

In theory, this should be easy. Why not simply try each machine many times, observe the returns, and then choose the one where these returns have been highest? In reality, though, this is not a good idea. Too many coins may have to be wasted before a reliable decision about the best machine can be made.

**A Simple Strategy** Mindful of the incurred costs, the practically minded engineer will limit the experimentation, and make an initial choice based on just a few trials. Knowing that this early decision is unreliable, she will not be dogmatic. She will occasionally experiment with the other machines: what if some of them might indeed be better? If yes, it will be quite reasonable to replace the "previously best" with this new one. The strategy is quite natural. One does not have to be machine-learning scientist to come up with something of this kind.

This then is the behavior that the *reinforcement learning* paradigm seeks to emulate. In the specific case from Fig. 17.1, there are five actions to choose from. The principle described above combines *exploitation* of the machine currently believed to be the best, and the *exploration* of alternatives. Exploitation dominates; exploration is rare. In the simplest implementation, the frequency of the exploration steps is controlled by a user-specified parameter, $\epsilon$. For instance, $\epsilon = 0.1$ means that the "best" machine (the one that appears best in view of previous trials) is chosen 90% of the time; in the remaining 10% cases, a chance is given to a randomly selected other machine.

**Keeping a Tally of the Rewards** The "best action" is defined as the one that has led to the highest average return.[1] For each action, the learner keeps a tally of the previous returns; and the average of these returns is regarded as this action's *quality*. For instance, let us refer to the machines in Fig. 17.1 by integers, $1, 2, 3, 4$, and 5. Action $a_i$ then represents the choice of the $i$-th machine. Suppose the leftmost machine was chosen three times, and these choices resulted in the following returns $r_1 = 0, r_2 = 9$, and $r_3 = 3$. The quality of this particular choice is then $Q(a_1) = (r_1 + r_2 + r_3)/3 = (0 + 9 + 3)/3 = 4$.

To avoid the necessity to store the rewards of all previously taken actions, the engineer implementing the procedure can take advantage of the following formula where $Q_k(a)$ is the quality of action $a$ as calculated from $k$ rewards, and $r_{k+1}$ is the $(k + 1)$st reward.

[1] At this point, let us remark that the returns can be negative—"punishments," rather.

**Table 17.1** The algorithm for the $\epsilon$-*greedy* reinforcement learning strategy

*Input*: user-specified parameter $\epsilon$, e.g., $\epsilon = 0.1$;
a set of actions, $a_i$, and their initial value-estimates, $Q_0(a_i)$;
for each action, $a_i$, let $k_i = 0$ (the number of times the action has been taken);

1. Generate a random number, $p \in (0, 1)$, from the uniform distribution.
2. If $p \geq \epsilon$, choose the action with the highest value (*exploitation*).
   Otherwise, choose a randomly selected other action (*exploration*).
3. Denote the action chosen in the previous step by $a_i$.
   Observe the reward, $r_i$.
4. Update the value of $a_i$ using the following formula:

$$Q(a_i) = Q(a_i) + \frac{1}{k_i + 1}[r_i - Q(a_i)]$$

5. Set $k_i = k_i + 1$ and return to 1.

$$Q_{k+1}(a) = Q_k(a) + \frac{1}{k+1}[r_{k+1} - Q_k(a)] \tag{17.1}$$

Thanks to this formula, it is enough to "remember" for each action only the values of $k$ and $Q_k(a)$—these are all that is needed, together with the latest reward, to update the action's value at the $(k + 1)$st step.

The procedure just described is sometimes called the $\epsilon$-*greedy* strategy. For the user's convenience, Table 17.1 summarizes the algorithm in a pseudocode.

**Initializing the Process** To be able to use Formula (17.1), we need to start somewhere: we need to set for each action its initial value, $Q_0(a_i)$. An elegant possibility is to choose a value well above any realistic single return. For instance, if all returns are known to come from the interval $[0, 10]$, the following will be reasonable initial values: $Q_0(a_i) = 50$.

At each moment, the system chooses, with $(1-\epsilon)$ probability, the action with the highest value, breaking ties randomly. At the beginning, all actions have the same chance of being taken. Suppose that $a_i$ is picked. In consequence of the received reward, this action's quality is then reduced using Formula (17.1). Therefore, when the next action is to be selected, it will (if *exploitation* is to be used) have to be some other action—whose value will then get reduced, too. Long story short, the reader can see that initialization of all action values to the same big number makes sure that, in the early stages of the game, all actions will be systematically experimented with.

## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Describe the $\epsilon$-*greedy* strategy to be used when searching for the best machine in the $N$-armed bandit problem. Explain the meaning of actions and their values. What is meant by *exploitation* and *exploration*?
- Describe the simple mechanism for maintaining the average rewards. How does this mechanism update the action's values?
- Why did this section recommend that the initial values, $Q_0(a_i)$, of all actions should be set to a multiple of the typical reward?

## 17.2 States and Actions in a Game

The example with slot machines is a simplification that has made it easy to explain the basic terminology. Its main limitation is the existence of only one *state* in which an appropriate action is to be selected.

In reality, the situation is more complicated than that. Usually, there are many states, each with several actions to choose from. The essence can be illustrated on the tic-tac-toe game.

**The Tic-Tac-Toe Game** The principle is shown in Fig. 17.2 for the elementary case where the size of the playing board is three by three squares. Two players are taking turns, one placing crosses on the board, the other one circles. The goal is to achieve a line of three crosses or circles—either in a column or in a row or diagonally. Who succeeds first, wins. If, in the situation on the left, it is the turn of the player that plays with crosses, he wins by putting his cross in the bottom left corner. If, conversely, it were his opponent's turn, the opponent would prevent this by putting there a circle.

**States and Actions** Each board-position represents a *state*. At each state, the player is to choose a concrete *action*. Thus in the state depicted on the left, there are three empty squares, and thus three actions to choose from (one of them winning). The whole situation can be represented by a look-up table in which each state-action


**Fig. 17.2** In tic-tac-toe, two players took turns at placing their *crosses* and *circles*. The winner is the one who obtains a triplet in a line (vertical, horizontal, or diagonal)

pair has a certain value, $Q(s, a)$. Based on these values, the $\epsilon$-*greedy* policy decides which action should be taken in the particular state. The action results in a reward, $r$, and this reward is then used to update the value of the state-action pair by means of Formula (17.1).

The most typical way of implementing the learning scenario is to let the program play a long series of games with itself, starting with ad hoc choices for actions (based on only the initial values of $Q(s, a)$), then gradually improving them until it achieves very high playing strength.

The main problem is how to determine the rewards of the concrete actions. In principle, three alternatives can be considered.

**Episodic Formulation** This is perhaps the simplest way of dealing with the reward-assignment problem. A whole game is played. If it is won, then all state-actions pairs encountered throughout the game by the learning agent are treated as if they received reward 1. If the game is lost, they are treated as if they all received reward $-1$.

The main weakness of this method is that it ignores the circumstance that not all actions taken in a game have equally contributed to the final outcome. A player may have lost only because of a single blunder that followed a long series of excellent moves. In this case, it would of course be unfair, even impractical, to punish the good moves. The same goes for the opposite: weak actions might actually receive the reward only because the game happened to be eventually won thanks to the opponent's unexpected blunder. One can argue, however, that, in the long run, these little "injustices" get averaged out because, most of the time, the winner's actions will be good.

The advantage of the episodic formulation is its simplicity.

**Continuing Formulation** The aforementioned problem with the episodic formulation (the fact that it may punish a series of good moves on account of a single blunder) might be removed under the assumption that we know how to determine the reward right after each action. This is indeed sometimes possible; and even in domain where this is *not* possible, one can often at least make an estimate.

Most of the time, however, an attempt to determine the reward for a given action before the game ends is speculative—and thus misleading. This is why this approach is rarely used.

**Compromise: Discounted Returns** This is essentially an episodic formulation improved in a way that determines the rewards based on the length of the game. For instance, the longer it has taken to win a tic-tac-toe game, the smaller the reward should be. There is some logic, in this approach: stronger moves are likely to win sooner. The way to implement this strategy is to discount the final reward by the number of steps taken before the victory.

Here is how to formulate the idea more technically: Let $r_k$ denote the reward obtained at the $k$-th trial and let $\gamma \in (0, 1)$ is a user-set discounting constant. The *discounted return* $R$ is then calculated as follows:

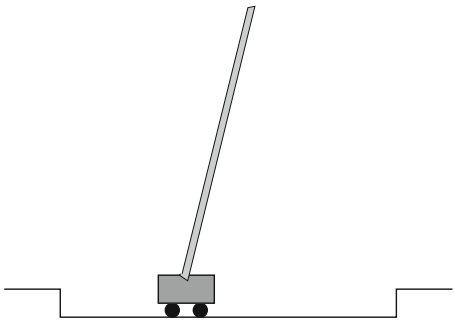

**Fig. 17.3** The task: keep the pole upright by moving the cart left or right

$$R = \Sigma_{k=1}^{\infty} \gamma^k r_k \tag{17.2}$$

Note how the growing value of $k$ decreases the coefficient by which $r_k$ is multiplied. If the ultimate reward comes at the 10th step, and if "1" is the reward for the winning game, then the discounted reward for $\gamma = 0.9$ is $R = 0.9^{10} \cdot 1 = 0.35$.

**Illustration: Pole Balancing** A good illustration of when the discounted return may be a good idea is the pole-balancing problem shown in Fig. 17.3. Here, each *state* is defined by such attributes as the cart location, the cart's velocity, the pole's angle, and the velocity of the change in the pole's angle. There are essentially two *actions* to choose from: (1) apply force in the left-right direction or (2) apply force in the right-left direction. However, a different amount of force may be used. The simplest version of this task assumes that the actions can only be taken at regular intervals, say, 0.2 s.

In this game, the longer the time that has elapsed before the pole falls, the greater the perceived success, and this is why longer games should be rewarded more than short games. A simple way to implement this circumstance is to reward each state during the game with a 0, and the final fall with, say, $r = -10$. The discounted return will then be $R = -10\gamma^N$ where $N$ is the number of steps before the pole has fallen.

## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the difference between *states* and *actions*. What is the meaning of the "value of the state-action pair"?
- When it comes to reward-assignment, what is the difference between the episodic formulation and the continuing formulation?
- Discuss the motivation behind the idea of *discounted returns*. Give the precise formula, and illustrate its use on the pole-balancing game.

## 17.3 The SARSA Approach

The previous two sections introduced only a very simplified mechanism to deal with the reinforcement-learning problem. Without going into details, let us describe here a more popular approach that is known under the name of SARSA. The pseudocode summarizing the algorithm is provided in Table 17.2.

Essentially, the episodic formulation with discounting is used. The episode begins with selecting an initial state, $s$ (in some domains, this initial state is randomly generated). In a series of successive steps, actions are taken according to the $\epsilon$-*greedy* policy. Each such action results in a new state, $s'$, being reached, and reward, $r$, being received. The same $\epsilon$-*greedy* policy is then used to choose the next action, $a'$ (to be taken in state $s'$). After this, the quality, $Q(s, a)$, of the given state-action pair is updated by the following formula:

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \tag{17.3}$$

Here, $\alpha$ is a user-set constant and $\gamma$ is the discounting factor.

Note that the update of the state-action pair's quality is based on the quintuple $(s, a, r, s', a')$. This is how the technique got its name.

## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Describe the principle of the SARSA approach to reinforcement learning. Where did this name come from?

**Table 17.2** The SARSA algorithm—using the $\epsilon$-*greedy* strategy and the episodic formulation of the task

*Input*: user-specified parameters $\epsilon, \alpha, \gamma$
Initialized values of all action-value pairs, $Q_0(s_i, a_j)$;
for each state-action pair, $s_i, a_j$, initialize $k_{ij} = 0$;

1. Choose an initial state, $s$.
2. Choose action $a$ using the $\epsilon$-*greedy* strategy from Table 17.1.
3. Take action $a$. This results in a new state, $s'$, and reward, $r$.
4. In state $s'$, choose action $a'$ using the $\epsilon$-*greedy* strategy.
   Update $Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
5. Let $s = s'$ and $a = a'$.
   If $s$ is a terminal state, start a new episode by going to 1; otherwise, go to 3.

## 17.4 Summary and Historical Remarks

- Unlike the classifier-induction problems from the previous chapters, reinforcement learning assumes that an agent learns from direct experimentation with a system it is trying to control.
- In the greatly simplified formalism of the $N$-armed bandit, the agent seeks to identify the most promising action—the one that offers the highest average returns. The simplest practical implementation relies on the so-called $\epsilon$-*greedy* policy.
- More realistic implementations of the task assume the existence of a set of states. For each state, the agent is to choose from a set of alternative actions. The choice can be made by the $\epsilon$-*greedy* policy that relies on the qualities of the state-action pairs, $Q(s, a)$.
- The problem of assigning the rewards to the state-action pairs can be addressed by its episodic formulation, by continuing formulation, or by episodic formulation with discounting.
- Of the more advanced approaches to reinforcement learning, the chapter briefly mentioned the SARSA method.

**Historical Remarks** One of the first systematic treatments of the "bandit" problem was offered by Bellman [3] who, in turn, was building on some earlier work still. Importantly, the same author later developed the principle of *dynamic programming* that can be regarded as a direct precursor to reinforcement learning [4]. The basic principles of reinforcement learning probably owe most for their development to Sutton [87].

## 17.5 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

## Exercises

1. Calculate the number of state-action pairs in the tic-tac-toe example from Fig. 17.2.

## Give It Some Thought

1. This chapter is all built around the idea of using the $\epsilon$-*greedy* policy. What do you think are the limitations of this policy? Can you suggest how to overcome them?
2. The principles of reinforcement learning have been explained using some very simple toy domains. Can you think of an interesting real-world application? The main difficulty will be how to cast the concrete problem into the reinforcement-learning formalism.
3. How many episodes might be needed to solve the simple version of the tic-tac-toe game shown in Fig. 17.2?

## Computer Assignments

1. Write a computer program that implements the $N$-armed bandit as described in Sect. 17.1.
2. Consider the maze-problem illustrated in Fig. 17.4. The task is to find the shortest path from the starting point, $S$, to the goal, $G$. A computer can use the principles of reinforcement learning to learn this shortest path based on great many training runs.

   Suggest the data structures to capture the states and actions of this game. Write a computer program that relies on the episodic formulation and the $\epsilon$-*greedy* policy when addressing this task.

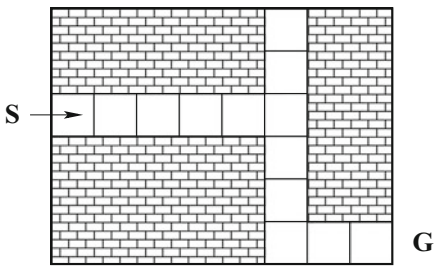


**Fig. 17.4** The agent starts at $S$; the task is to find the shortest path to $G$