ARTIFICIAL INTELLIGENCE

FOUNDATIONS OF COMPUTATIONAL AGENTS

11.3 Reinforcement Learning

Imagine a robot that can act in a world, receiving rewards and punishments and determining from these what it should do. This is the problem of reinforcement learning. This chapter only considers fully observable, single-agent reinforcement learning [although Section 10.4.2 considered a simple form of multiagent reinforcement learning].

We can formalize reinforcement learning in terms of Markov decision processes, but in which the agent, initially, only knows the set of possible states and the set of possible actions. Thus, the dynamics, $P(s'|a,s)$, and the reward function, $R(s,a,s')$, are initially unknown. An agent can act in a world and, after each step, it can observe the state of the world and observe what reward it obtained. Assume the agent acts to achieve the optimal discounted reward with a discount factor γ.

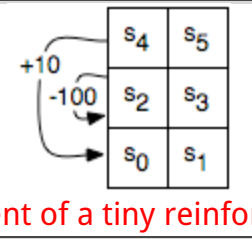

Figure 11.8: The environment of a tiny reinforcement learning problem

Example 11.7: Consider the tiny reinforcement learning problem shown in Figure 11.8. There are six states the agent could be in, labeled as $s_0,\ldots,s_5$. The agent has four actions: *UpC*, *Up*, *Left*, *Right*. That is all the agent knows before it starts. It does not know how the states are configured, what the actions do, or how rewards are earned.

Figure 11.8 shows the configuration of the six states. Suppose the actions work as follows:

upC
: (for "up carefully") The agent goes up, except in states s_4 and s_5, where the agent stays still, and has a reward of *-1*.

right
: The agent moves to the right in states s_0,s_2,s_4 with a reward of 0 and stays still in the other states, with a reward of *-1*.

left
: The agent moves one state to the left in states s_1,s_3,s_5. In state s_0, it stays in state s_0 and has a reward of *-1*. In state s_2, it has a reward of *-100* and stays in state s_2. In state s_4, it gets a reward of *10* and moves to state s_0.

up
: With a probability of *0.8* it acts like *upC*, except the reward is 0. With probability *0.1* it acts as a *left*, and with probability *0.1* it acts as *right*.

Suppose there is a discounted reward with a discount of 0.9. This can be translated as having a 0.1 chance of the agent leaving the game at any step, or as a way to encode that the agent prefers immediate rewards over

future rewards.

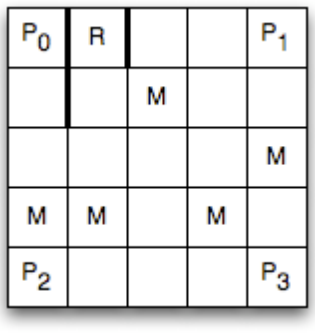

Figure 11.9: The environment of a grid game

Example 11.8: Figure 11.9 shows the domain of a more complex game. There are 25 grid locations the agent could be in. A prize could be on one of the corners, or there could be no prize. When the agent lands on a prize, it receives a reward of *10* and the prize disappears. When there is no prize, for each time step there is a probability that a prize appears on one of the corners. Monsters can appear at any time on one of the locations marked *M*. The agent gets damaged if a monster appears on the square the agent is on. If the agent is already damaged, it receives a reward of *-10*. The agent can get repaired (i.e., so it is no longer damaged) by visiting the repair station marked *R*.

In this example, the state consists of four components: $\langle X,Y,P,D\rangle$, where X is the X-coordinate of the agent, Y is the Y-coordinate of the agent, P is the position of the prize ($P=0$ if there is a prize on P_0, $P=1$ if there is a prize on P_1, similarly for 2 and 3, and $P=4$ if there is no prize), and D is Boolean and is true when the agent is damaged. Because the monsters are transient, it is not necessary to include them as part of the state. There are thus $5\times5\times5\times2 = 250$ states. The environment is fully observable, so the agent knows what state it is in. But the agent does not know the meaning of the states; it has no idea initially about being damaged or what a prize is.

The agent has four actions: *up*, *down*, *left*, and *right*. These move the agent one step - usually one step in the direction indicated by the name, but sometimes in one of the other directions. If the agent crashes into an outside wall or one of the interior walls (the thick lines near the location *R*), it remains where is was and receives a reward of *-1*.

The agent does not know any of the story given here. It just knows there are 250 states and 4 actions, which state it is in at every time, and what reward was received each time.

This game is simple, but it is surprisingly difficult to write a good controller for it. There is a Java applet available on the book web site that you can play with and modify. Try to write a controller by hand for it; it is possible to write a controller that averages about 500 rewards for each 1,000 steps. This game is also difficult to learn, because visiting *R* is seemingly innocuous until the agent has determined that being damaged is bad, and that visiting *R* makes it not damaged. It must stumble on this while trying to collect the prizes. The states where there is no prize available do not last very long. Moreover, it has to learn this without being given the concept of *damaged*; all it knows, initially, is that there are 250 states and 4 actions.

Reinforcement learning is difficult for a number of reasons:

- The blame attribution problem is the problem of determining which action was responsible for a reward or punishment. The responsible action may have occurred a long time before the reward was received. Moreover, not a single action but rather a combination of actions carried out in the appropriate circumstances may be responsible for the reward. For example, you could teach an agent to play a game by rewarding it when it wins or loses; it must determine the brilliant moves that were needed to win. You may try to train a dog by saying "bad dog" when you come home and find a mess. The dog has to determine, out of all of the actions it did, which of them were the actions that were responsible for the reprimand.
- Even if the dynamics of the world does not change, the effect of an action of the agent depends on what the agent will do in the future. What may initially seem like a bad thing for the agent to do may end up being an optimal action because of what the agent does in the future. This is common among planning problems, but it is complicated in the reinforcement learning context because the agent does not know, a priori, the effects of its actions.
- The explore-exploit dilemma: if the agent has worked out a good course of actions, should it continue to follow these actions (exploiting what it has determined) or should it explore to find better actions? An agent that never explores may act forever in a way that could have been much better if it had explored earlier. An

agent that always explores will never use what it has learned. This dilemma is discussed further in Section 11.3.4.