

# Reinforcement learning I

In supervised learning, we had a training set  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$  for which we had the “right answer”  $y^{(i)}$  for every instance  $x^{(i)}$ . For example, supervised learning algorithms would learn to drive by predicting the actions of an expert human driver. In this set of notes, we’ll talk about a different setting called **reinforcement learning**, where we don’t know the right answers ahead of time; we only know a “reward function” which tells us the goodness of particular states. (For instance, we might believe that a state where the helicopter is in the air is better than one where it is lying in bits and pieces on the ground.)

In detail, we specify a **reward function** mapping states of the world to real numbers. The algorithm’s goal is to find a sequence of actions which maximizes this reward function over time. This temporal component means the algorithm does not simply have to make a one shot decision. For instance, the helicopter must choose actions which not only allow it to stay in the air at this exact moment, but which also keep it stable enough that it can remain in the air continually.

If the world were completely deterministic, it would be easy to maximize such a reward function using techniques from our lectures on search. However, in many robotics systems, the dynamics are **stochastic**, in that the same action doesn’t always lead to the exact same result every time. For instance, telling a robot to move one meter forward could typically result in it moving anywhere between 95 and 105 cm forward, due to factors such as slippage of the wheels. Even a small amount of randomness would cause big problems for the deterministic search algorithms we covered earlier in the course.

We will take an approach where we “reward” the robot for desired outcomes and “punish” it for undesired ones. One challenge faced by reinforcement learning is the **credit assignment problem**. Upon reaching a

position with negative reward, it may not be obvious which previous action had caused that negative reward. For instance, suppose you are driving and you crash your car. Chances are, you slammed on your breaks shortly before the crash. You wouldn't want to conclude from this that it's a bad idea to ever step on the breaks again. Rather, the crash was probably due to an action you chose much earlier, such as your decision to go 90 MPH down the highway.

In these notes, we present the standard reinforcement learning formalism, known as the **Markov decision process (MDP)**. MDPs allow us to model the (stochastic) dynamics of the world as well as the desired outcomes. As we will see, within this formalism, we can tractably compute the optimal behaviors which maximize the reward function over time.

## 1 Markov Decision Processes (MDPs)

An MDP is a 5-tuple  $(S, A, \{P_{sa}\}, \gamma, R)$ , where

- $S$  is the set of states
- $A$  is the set of actions
- $P_{sa}$  gives the **state transition probabilities** for state  $s$  and action  $a$ . If we are in state  $s$ , then for any state  $s'$ ,  $P_{sa}(s')$  gives the probability that taking action  $a$  will cause us to transition to state  $s'$ . Since  $P_{sa}$  is a probability distribution,  $\sum_{s'} P_{sa}(s') = 1$  and  $P_{sa}(s') \geq 0$  for all  $s \in S$  and  $a \in A$ .
- $\gamma$  is a real-valued **discount factor** in the interval  $0 \leq \gamma < 1$  telling us how much we value rewards right now relative to rewards in the future.
- $R : S \mapsto \mathbb{R}$  is the reward function, which measures the desirability of being in each state.

Consider, for instance, the following example from our class textbook, shown in Figure 1 (a). The world is a  $4 \times 3$  grid, with one obstacle, giving a total of 11 states. There are four possible actions the robot can take, corresponding to moving in each of the four compass directions, labeled  $\{N, S, E, W\}$ . However, the motion dynamics are noisy, and so if the robot tries to move in a particular direction, there is a 10% chance of it instead moving in the direction  $90^\circ$  to the left, and a 10% chance of it moving in the direction  $90^\circ$  to the right. If the robot's direction of motion causes it to move

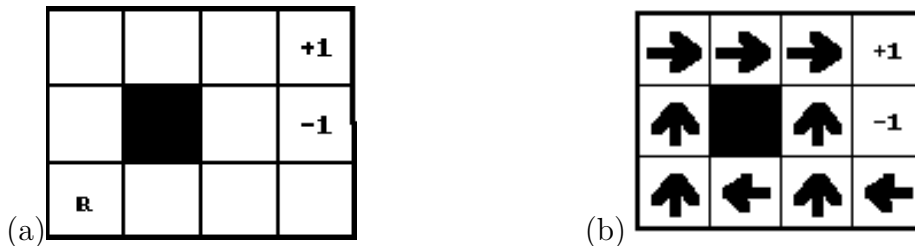


Figure 1: (a) An example MDP, taken from Russell & Norvig. The initial location of the robot is given by R. (b) The optimal policy for the MDP, for  $\gamma = 0.99$ .

into a wall, it simply bounces off and stays put in its current square. For instance, if the robot is in square below the obstacle, trying to move down will result in a 10% chance of moving left, a 10% chance of moving right, and an 80% chance of staying put.

Its goal is to wind up in the upper right corner, so that square is assigned a reward of +1. We don't want it to land in the square below, so that square is assigned a reward of -1. As soon as the robot enters one of these two squares, the MDP is over and no more moves are made.<sup>1</sup> We also don't want the robot to dawdle too long, and so all other states are assigned a small negative reward of -0.04 (perhaps corresponding to fuel or battery consumption). Assigning a small negative reward to all states is a common method for preventing a robot from sitting idle.

Let's now define a little more formally how an MDP works. We begin in some state  $s_0$ . At each (discrete) time  $t$ , we are in some state  $s_t$ , we choose some action  $a_t$ , and a successor  $s_{t+1}$  is drawn according to the transition probabilities, i.e.  $s_{t+1} \sim P_{s_t a_t}$ . This process generates an infinite sequence of states  $s_0, s_1, s_2, \dots$ . The **total payoff** is defined as a weighted sum of the rewards for the states in this sequence:

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

The contribution of each state to the total payoff is weighted by  $\gamma^t$ , which decreases exponentially quickly with  $t$ . Such a weighting is known as **discounting**. In a financial application, the discount factor  $\gamma$  has a natural interpretation as the time value of money. A dollar today is worth more

---

<sup>1</sup>To treat this more formally, you can imagine that both of these states transition with probability 1 to a "zero-cost absorbing state." This is a state which transitions to itself with probability 1 and has no associated reward.

than a dollar a year from now, because of the interest rate—a dollar placed in a bank will earn you back slightly more than a dollar in a year’s time. Small values of  $\gamma$  imply a very fast decay of the value of the rewards per unit time, and hence mean we will be shortsighted. Large values of  $\gamma$  (e.g. very close to 1) mean we will try to maximize our expected payoff long into the future.

We said above that we have to choose an action at each time instant. More specifically, our goal is to find a **policy**  $\pi$  which assigns an action to every state, i.e.  $\pi : S \mapsto A$ . At each time  $t$ , we will choose the action which  $\pi$  assigns the current state,  $a_t = \pi(s_t)$ . If we take actions in an MDP following the actions specified by a policy  $\pi$ , then we say that we are **executing** policy  $\pi$  in the MDP.

We are interested in computing the optimal policy of the MDP. Before we define the optimal policy, we need some preliminary definitions. First, define the **value function** of a policy  $\pi$  to be a function which takes a state  $s$  and returns the expected total payoff if we start at  $s$ . More formally,  $V^\pi : S \mapsto \mathbb{R}$ , where

$$V^\pi(s) = \mathbb{E} [R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots \mid \pi, s_0 = s].$$

We can’t compute  $V^\pi(s)$  directly from this definition, however, because it is a sum of an infinite number of terms. What we will do instead is to define  $V^\pi(s)$  recursively in terms of  $V^\pi(s')$  for all states  $s'$ . This will give a system of linear equations which can be solved. This is called **Bellman’s equation for  $V^\pi$** .

$$\begin{aligned} V^\pi(s) &= \mathbb{E} [R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots \mid \pi, s_0 = s] \\ &= \mathbb{E} [R(s_0) + \gamma (R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + \dots) \mid \pi, s_0 = s] \\ &= R(s) + \gamma \mathbb{E} [R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + \dots \mid \pi, s_0 = s] \\ &= R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') \mathbb{E} [R(s_1) + \gamma R(s_2) + \gamma R(s_3) + \dots \mid \pi, s_1 = s'] \\ &= R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^\pi(s') \end{aligned} \tag{1}$$

This is a system of linear equations, where the variables are the values of  $V^\pi(s)$  for the different states  $s$ . If  $n$  is the number of states, there are  $n$  equations with  $n$  unknowns. Such a system can be solved in closed form using standard techniques. In the equation above,  $R(s)$  is sometimes also called the **immediate reward**.

Now, for a given state  $s$ , we can define the **optimal value** of  $s$ , denoted  $V^*(s)$ , as the largest expected total payoff starting from  $s$  which can be

achieved by any policy:

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

We don't want to compute  $V^*(s)$  directly from this definition (i.e., by enumerating all policies). Instead, we will make use of a version of Bellman's equation for  $V^*$ :

$$V^*(s) = R(s) + \max_a \gamma \sum_{s'} P_{sa}(s') V^*(s'). \quad (2)$$

You should convince yourself that Bellman's equation must hold true for the optimal values  $V^*$ . The converse is more subtle. You'll have a chance to prove, in Problem Set 3, that  $V^*$  is uniquely defined by Bellman's equation.

Now, rather than define the optimal policy directly, we define a particular policy  $\pi^*$ , which will turn out to be the optimal policy. Define  $\pi^*$  to be the policy which looks the best according to  $V^*$ , i.e.

$$\pi^*(s) = \arg \max_a \sum_{s'} P_{sa}(s') V^*(s').$$

It's a fact, which we won't prove, that

$$V^*(s) = V^{\pi^*}(s).$$

(Make sure you understand all the notation used here.) In other words,  $\pi^*$  is the **optimal policy**, or the one which achieves the highest expected total payoff for each state.

Figure 1 (b) shows the optimal policy for the grid world.

## 2 Solving MDPs

We have just defined our goal as finding the optimal policy  $\pi^*$ , and now we introduce algorithms to do that.

### 2.1 Value iteration

We gave a formula for computing the optimal policy  $\pi^*$  in terms of the optimal value function  $V^*$ , so one way to proceed is to compute the optimal value  $V^*(s)$  for each state  $s$ , and then use  $V^*$  to get  $\pi^*$ . **Value iteration** is an iterative algorithm which essentially changes the equality in Bellman's equation (Equation 2) into an update rule.

Specifically, value iteration works as follows:

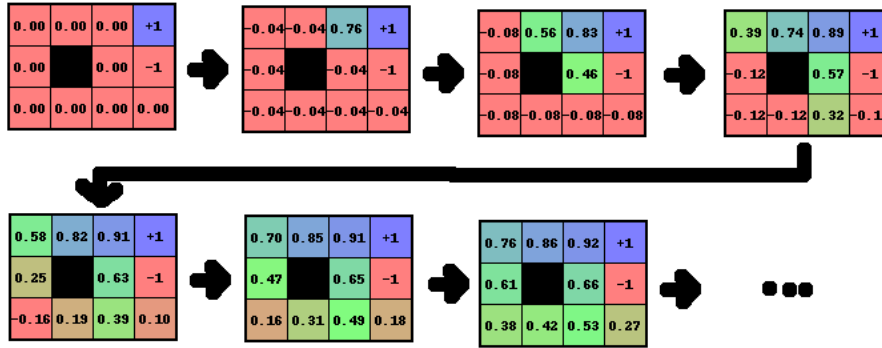


Figure 2: An example of value iteration applied to our MDP, for  $\gamma = 0.999$ .

Initialize  $V(s) = 0$  for all states  $s$ .

Repeat until convergence:

For every state  $s \in S$ , update

$$V(s) := R(s) + \max_a \gamma \sum_{s' \in S} P_{sa}(s') V(s')$$

This procedure will gradually cause  $V(s)$  to converge to the optimal value function  $V^*(s)$ . An example is shown in Figure 2.

Notice that our above definition is ambiguous. Suppose it's time to update a state  $s$  on the  $i^{\text{th}}$  pass through all of the states, and for some other state  $s'$ ,  $V(s')$  has already been updated on the  $i^{\text{th}}$  pass. Which value of  $V(s')$  do we use: the one from the  $i - 1^{\text{st}}$  pass, or the one which was newly assigned on the  $i^{\text{th}}$  pass? It turns out that both versions give a correct algorithm. Using the value from the  $i - 1^{\text{st}}$  pass is known as **synchronous** updates, while using the value from the  $i^{\text{th}}$  pass is known as **asynchronous** updates.

Sometimes it's convenient to think of  $V$  a vector, where the  $j^{\text{th}}$  component of  $V$  corresponds to  $V(s_j)$ . In the synchronous updates version of value iteration, we sometimes refer to one pass through all of the states as the **Bellman operator**  $B$ . In this notation, each pass through the states can be written as  $V := B(V)$ .

## 2.2 Policy iteration

**Policy iteration** is another algorithm based on a similar intuition. Policy iteration also uses the Bellman equations as update rules in an iterative

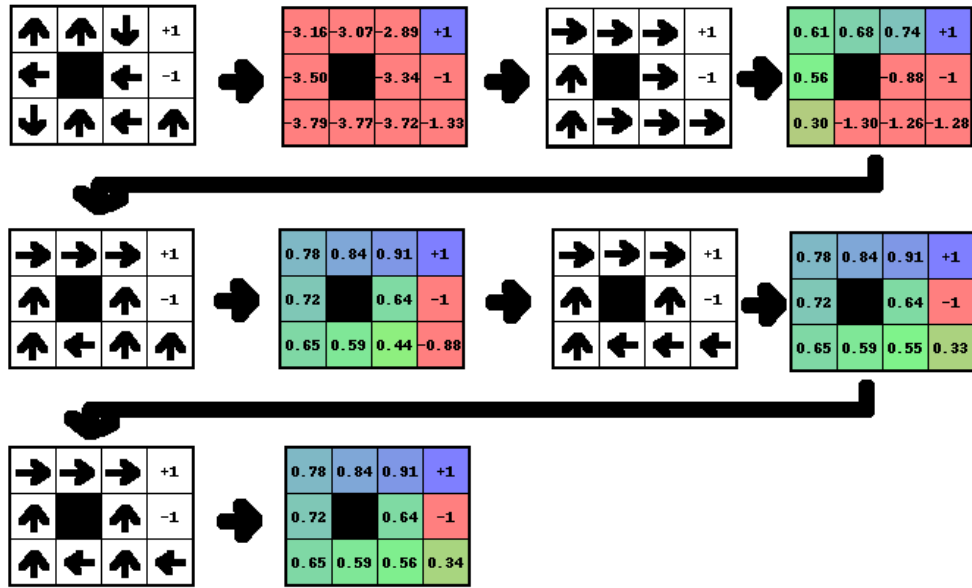


Figure 3: An example of policy iteration applied to our MDP, for  $\gamma = 0.99$ .

algorithm, but does so in a slightly different way. In policy iteration, we alternate between computing the expected total payoff function  $V^\pi$  for a given policy and choosing a new policy  $\pi$  based on our current estimate of the value of each state. Our estimate  $V(s')$  is taken to be the *actual* expected total payoff for state  $s'$  under the policy  $\pi$ , as computed by solving the system of linear equations.

Policy iteration can be written as follows:

Initialize  $\pi$  randomly.

Repeat until convergence:

Let  $V := V^\pi$ . (In other words, compute  $V^\pi$  from (1).)

Let  $\pi(s) := \arg \max_a \sum_{s' \in S} P_{sa}(s')V(s')$

In a finite number of iterations,  $V$  will converge to  $V^*$ , and  $\pi$  will converge to  $\pi^*$ . An example is shown in Figure 3.

What are the advantages and disadvantages of policy iteration relative to value iteration? The advantage is that, rather than using its previous

(possibly very inaccurate) estimate of the value of a particular state  $s$ , it actually computes the exact value of the state relative to some policy  $\pi$ . Hence, if the current policy is somewhat similar to the optimal one, then  $V$  should be very accurate. In practice, policy iteration takes many fewer iterations to converge than value iteration.<sup>2</sup> On the other hand, it's much more expensive to compute one iteration of policy iteration than one iteration of value iteration. For each update, value iteration only requires taking a maximum of a set of numbers, while policy iteration requires solving a set of linear equations. Therefore, policy iteration is perhaps most effective in domains with a small number of states, and value iteration for larger problems.

---

<sup>2</sup>It is still an open problem to find good bounds for the number of iterations required for policy iteration to converge. The best known bound is exponential in the number of states, but there are no known examples which actually take exponentially long.