MDP Tutorial

Dimitri Klimenko

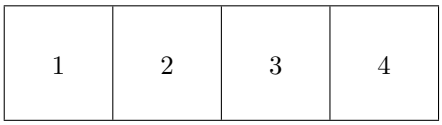
23rd October 2015

Contents

1 Problem description

Suppose you need to solve a simple robot navigation problem. The robot operates in a grid environment, as shown in Figure 1. The robot starts in cell 1 and the goal is to move to cell 4. It has two actions: Left and Right. Given a left (or right) control action, the robot will correctly move to the cell at the left (or right) of its current cell with probability 0.8. It will remain in its current cell with probability 0.2. If the action causes the robot to move outside of the boundary, then the robot will remain in its current cell.

Figure 1: The environment



2 MDP model

The description above is sufficient to define the states, actions and transitions for our MDP. However, in order to properly specify the robot's goal, we will also need a reward function; this function must encode the idea that the robot's goal is to move to cell 4.

The simplest way of doing this would be to reward the robot simply for being in cell 4, but then this reward would be received many times over after reaching cell 4. Instead, we reward the robot only upon its initial entry into cell 4, and we also change the problem so that cell 4 becomes a *terminal state*. In addition to rewarding the robot for reaching cell 4, we would like to reward it for doing this quickly. This can be achieved in two different ways—using a discount factor (which means reaching the goal earlier is worth more), and using a constant cost of movement for every step. To demonstrate how this works, we will use both of these approaches.

In particular, we set our reward function to $+10$ for reaching the goal with a cost of -1 for every move made or attempted, as well as using a discount factor of $\gamma = 0.95$. Overall, our MDP model is $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$, where

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the action space.
- T is the transition function.
- R is the reward function.
- $\gamma = 0.95$ is the discount factor.

2.1 State space

Since the environment is static, we only need one state variable to fully describe the problem state—the position of the robot. As such, the state space for this

problem can be represented as

$$\mathcal{S} = \{s_1, s_2, s_3, s_4\}.$$

We have also decided that s_4 should be a *terminal state*, i.e. the MDP stops after reaching it. One way to represent this is to require that all actions from s_4 always transition back to s_4 again and give zero reward; since no rewards occur after reaching it, this is essentially equivalent to the MDP terminating. Similarly, since we know that the agent starts in s_1, we can call this state the *initial state*—this would be the root of the tree if we were using Monte-Carlo Tree Search to solve this problem.

2.2 Action space

The robot can only go left or right, so the action space for the problem is simply

$$\mathcal{A} = \{\mathit{Left}, \mathit{Right}\}.$$

2.3 Transition function

The transition function $T(s, a, s')$ describes the probabilities for the possible next states s', conditional on the robot being in state s and taking action a, i.e.

$$T(s, a, s') = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a),$$

where S_t is the state at time t, and A_t is the action at time t. For this particular MDP model, we have

$$T(s_4, \mathit{Left}, s_4) = T(s_4, \mathit{Right}, s_4) = 1$$

since we've stated that the game ends upon reaching s_4, so taking either action from s_4 means that you stay there. We also have

$$T(s_1, \mathit{Left}, s_1) = 1$$

since moving left from s_1 causes us to stay in s_1. Because the robot has an 80% chance of going where it intended to go, we have

$$T(s_2, \mathit{Left}, s_1) = T(s_3, \mathit{Left}, s_2) = T(s_1, \mathit{Right}, s_2) = T(s_2, \mathit{Right}, s_3) = T(s_3, \mathit{Right}, s_4) = 0.8.$$

Additionally, we have

$$T(s_2, \mathit{Left}, s_2) = T(s_3, \mathit{Left}, s_3) = T(s_1, \mathit{Right}, s_1) = T(s_2, \mathit{Right}, s_2) = T(s_3, \mathit{Right}, s_3) = 0.2,$$

as the robot has a 20% chance of staying in the same square when it tries to move to a different one. For all other transitions, we have

$$T(*, *, *) = 0,$$

because, for example, the robot can never end up in s_3 immediately after being in s_1.

Table 1: Transition matrix representation of T.

(a) Transition matrix P_{Left}

	s_1	s_2	s_3	s_4
s_1	1	0	0	0
s_2	0.8	0.2	0	0
s_3	0	0.8	0.2	0
s_4	0	0	0	1

(b) Transition matrix P_{Right}

	s_1	s_2	s_3	s_4
s_1	0.2	0.8	0	0
s_2	0	0.2	0.8	0
s_3	0	0	0.2	0.8
s_4	0	0	0	1

An alternative way to represent this information is with *transition matrices*, one for *Left*, and one for *Right*; those matrices can be seen in Table 1. The rows of the transition matrix represent the *current state* s, and the columns represent the *next state* s'. For example, by looking at the cell in the second row and third column in Table 1b, we can see that the probability of reaching s_3 when going *Right* from s_2 is 0.8, i.e. $T(s_2, \mathit{Right}, s_3) = 0.8$.

2.4 Reward function

Since we have decided that s_4 is a *terminal state*, we are representing this by having the robot get "stuck" in s_4 after reaching it. This means that we must set the reward for taking actions from s_4 to always be zero, or the robot could simply continue to collect rewards in s_4 forever, as we're using an infinite-horizon MDP. As such, we must instead give the robot a reward whenever it reaches s_4. We can't simply give the robot a reward for going *Right* from s_3, as sometimes the robot stays in s_3. This means that our reward function R needs to take the form

$$R(s, a, s'),$$

which represents the reward for reaching s' after taking action a from state s.

As we've said that the robot eternally receives no rewards after s_4 is reached, this means that

$$R(s_4, \mathit{Left}, s_4) = R(s_4, \mathit{Right}, s_4) = 0.$$

The robot is given a reward of 10 reaching s_4; we can also see by inspection of the transition matrices that the only possible way to reach s_4 is by going *Right* from s_3. Since it costs the robot 1 utilon every time it moves, we have

$$R(s_3, \mathit{Right}, s_4) = 9,$$

which incorporates the reward for reaching s_4 as well as the move cost. All other movements of the robot don't reach the goal but still cost one utilon, so finally we have

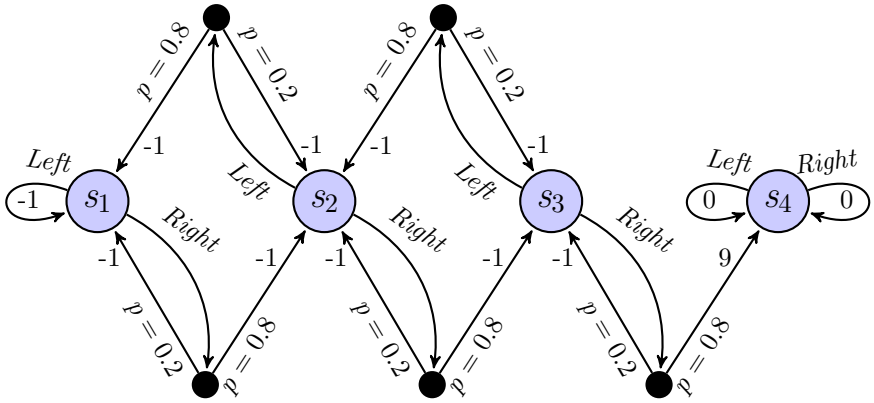
$$R(*, *, *) = -1$$

for all other transitions.

2.5 Graphical representation

This MDP can also be represented via a graph, which can be seen in Figure 2. The four blue nodes in the graph represent the various states, s_1, s_2, s_3 and s_4, while the arrows out of those nodes represent the actions (*Left* and *Right*)

Figure 2: A graphical representation of the MDP model



that can be taken from those states. The filled black nodes are "chance nodes", representing different possible outcomes due to a random event, with labels like $p = 0.2$ representing the probability $T(s, a, s')$ of different state transitions. The integer labels on the ends of the arrows represent the reward function $R(s, a, s')$, which specifies the reward for reaching state s' after taking action a from state s.

3 Policies and value functions

3.1 Policies

A *policy* $\pi : \mathcal{S} \to \mathcal{A}$ is a mapping from states to actions. This is the way we describe how the agent in an MDP will act; in the context of a game we might also call this a *strategy*.

If we associate the states with a specific order, we can also represent a policy as a *tuple*, an ordered list of actions that lists one action for each state. In this case, the obvious ordering is (s_1, s_2, s_3, s_4), and so an example policy is

$$\pi_L = (\mathit{Left}, \mathit{Left}, \mathit{Left}, \mathit{Left}).$$

This is clearly not a very good policy because the robot's goal is to get to the rightmost square, but it's a policy nonetheless! A policy that's clearly better is

$$\pi_R = (\mathit{Right}, \mathit{Right}, \mathit{Right}, \mathit{Right});$$

indeed, this is in fact one of two optimal policies in this MDP, along with

$$\pi_2 = (\mathit{Right}, \mathit{Right}, \mathit{Right}, \mathit{Left});$$

3.2 Value functions

A *value function* $V : \mathcal{S} \to \mathbb{R}$ is a function that associates each state with a *value*, which is a real number that estimates, for each state, how valuable it is for us to be in that state. For example, a relatively silly value function would be

$$V_0(\cdot) \equiv 0$$

which assumes that all states are equally worthless.

3.3 The value of a policy

For a given policy π we can calculate the value function for that policy V_π using equations that look like this:

$$V_\pi(s) = \sum_{s'} T(s, \pi(s), s')\Big[R(s, \pi(s), s') + \gamma\, V_\pi(s')\Big], \tag{1}$$

where s is the current state, $\pi(s)$ is the action that the policy chooses for that state, and the values s' represent the possible next states. It's important to note that the value function on the left hand side and right hand side of the equation is the same one. Since there many states, this actually forms a *system of linear equations* that can be solved to calculate the value of a given policy. Since there is one equation for each state, and each state has a value, this is a system of n equations in n unknowns, where n is the number of states.

Equation 1 simply represents a calculation of the expected value over different possibilities for the next state. In other words, the expected value of a given state is the sum of two terms; the expected value of the immediate reward,

$$\mathbb{E}\left[R(s, a, \cdot)\right] = \sum_{s'} T(s, \pi(s), s') R(s, \pi(s), s'), \tag{2}$$

and the discounted expected value of the next state,

$$\gamma \mathbb{E}\left[V_\pi(\text{next state})\right] = \gamma \sum_{s'} T(s, \pi(s), s')\, V_\pi(s'). \tag{3}$$

If the reward was independent of the next state, we could represent our reward with a function $R(s, a)$ instead, with $R(s, a, s') \equiv R(s, a)$. Consequently, the expected reward over all possible next states would simply be

$$\mathbb{E}\left[R(s, \pi(s), \cdot)\right] \equiv R(s, \pi(s))$$

and Equation 1 would simplify to give

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s')\, V_\pi(s'). \tag{4}$$

3.3.1 An example

As an example, we can calculate the value of our not-very-good policy

$$\pi_L = (\mathit{Left}, \mathit{Left}, \mathit{Left}, \mathit{Left}).$$

For convenience, we can turn Equation 1 into a matrix form. As you may recall, the transition function can represented by *transition matrices* (one for each action). Similarly, the state transitions given a fixed policy can also be described by a single transition matrix. Since π_L says we should always go left, the appropriate transition matrix is simply $P_L = P_{\mathit{Left}}$ from Table 1a, i.e.

$$P_L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.8 & 0.2 & 0 & 0 \\ 0 & 0.8 & 0.2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since moving left always has a cost of 1 utilon, we know that the expected immediate reward will always be -1, except at s_4 where the reward is always zero. Consequently, we can represent the expected immediate reward by the vector

$$\mathbf{r} = \begin{pmatrix} -1 \\ -1 \\ -1 \\ 0 \end{pmatrix}.$$

Using this, we can transform Equation 1 to the matrix form

$$\mathbf{v}_L = \mathbf{r} + \gamma P_L \mathbf{v}_L. \tag{5}$$

I've switched to the notation $\mathbf{v}_L$ for the value function to emphasize that it is a *vector* in this equation. Some basic matrix algebra gives

$$(I - \gamma P_L)\mathbf{v}_L = \mathbf{r} \tag{6}$$

which is simply a system of linear equations in matrix form! If we substitute the numbers in, we have

$$\left[\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} - 0.95 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.8 & 0.2 & 0 & 0 \\ 0 & 0.8 & 0.2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \mathbf{v}_L = \begin{pmatrix} -1 \\ -1 \\ -1 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 0.05 & 0 & 0 & 0 \\ -0.76 & 0.81 & 0 & 0 \\ 0 & -0.76 & 0.81 & 0 \\ 0 & 0 & 0 & 0.05 \end{pmatrix} \begin{pmatrix} V_L(s_1) \\ V_L(s_2) \\ V_L(s_3) \\ V_L(s_4) \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ -1 \\ 0 \end{pmatrix}.$$

If we solve the equations, we get

$$V_L(s_4) = 0, \text{ and}$$
$$V_L(s_1) = V_L(s_2) = V_L(s_3) = -20.$$

This would mean that, from our initial state of s_1, the expected value of our policy π_L is -20. Clearly not a very good policy...

3.4 The one-step best policy for a value function

Just as we calculated the value function for a given policy, we can start with a value function and work out what the best policy would be if those values were actually correct. This is a little weird, because if start with wrong values and work out a policy for those values that would actually change what the values would be. The way we make sense of this idea is that we specifically look only *one step ahead* in working out which action to take from every state - this is a *one-step best policy.*

To make this simpler, we introduce the notion of a *Q-value.* The optimal Q-value $Q^*(s, a)$ is the value of taking action a from state s, and following an optimal policy therewith. However, for a one-step best policy we don't necessarily know that our value function correctly represents the value of an optimal policy, and we don't necessarily know what the optimal policy is to begin with. Further

detail on the true or optimal Q-value is given in Section 3.5. Instead, for our purposes we use a *one-step Q-value estimate* $\widetilde{Q}_V$, which represents the value of taking action a from state s under the assumption that the given value function V correctly represents the values of the subsequent next states. This estimate is, similarly to Equation 1, described by

$$\widetilde{Q}_V(s,a) = \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma V(s')\Big]. \tag{7}$$

Note that, just as we did for Equation 4, we can simplify this equation in the case where our reward is completely independent of the next state; this would result in

$$\widetilde{Q}_V(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s')\, V(s'). \tag{8}$$

Now, in order to work out a one-step best policy for a given value function, we look one step ahead for every state and choose the action that has the best one-step value for each state. Consequently, the one-step best policy can be defined like this:

$$\widetilde{\pi}_V(s) = \arg\max_a \widetilde{Q}_V(s,a) = \arg\max_a \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma V(s')\Big], \tag{9}$$

which simply states that our policy always takes the action that has the largest expected value when we look one step ahead with our given value function V.

3.4.1 An example

As an example, let's start with a value function V_σ (σ for silly), which we will represent by the vector

$$\mathbf{v}_\sigma = (1,0,0,0),$$

which means that we're assuming that being in cell 1 is worth 1 utilon, whereas being in any of the other cells is worth 0 utilons.

In order to find the one-step best policy, we first calculate the values of all the actions from looking one step ahead, which gives us the following:

$$\begin{gathered}
\widetilde{Q}_\sigma(s_1, \mathit{Left}) = -1 + 0.95 \cdot 1 = -0.05 \\
\widetilde{Q}_\sigma(s_1, \mathit{Right}) = -1 + 0.95(0.8 \cdot 0 + 0.2 \cdot 1) = -0.81 \\
\widetilde{Q}_\sigma(s_2, \mathit{Left}) = -1 + 0.95(0.8 \cdot 1 + 0.2 \cdot 0) = -0.24 \\
\widetilde{Q}_\sigma(s_2, \mathit{Right}) = -1 + 0.95(0.8 \cdot 0 + 0.2 \cdot 0) = -1 \\
\widetilde{Q}_\sigma(s_3, \mathit{Left}) = -1 + 0.95(0.8 \cdot 0 + 0.2 \cdot 0) = -1 \\
\widetilde{Q}_\sigma(s_3, \mathit{Right}) = 0.8 \cdot (9 + 0.95 \cdot 0) + 0.2 \cdot (-1 + 0.95 \cdot 0) = 7 \\
\widetilde{Q}_\sigma(s_4, \mathit{Left}) = 0 + 0.95 \cdot 0 = 0 \\
\widetilde{Q}_\sigma(s_4, \mathit{Right}) = 0 + 0.95 \cdot 0 = 0.
\end{gathered}$$

Note that since the reward is always independent of the next state (i.e. either -1 or 0) except when we're in cell 3 and attempt to move right, we've used the simplified form of Equation 8 to calculate these q-values.

Now, our one-step best policy is simply a policy that takes the action with the highest estimated value for each state. From the above calculations, it's pretty clear that the one-step best action for s_1 and s_2 is to go *Left*. This seems pretty stupid since the robot's goal was supposed to be reaching s_4, but remember that this is the result of choosing a rather silly value function that assumed *a priori* that being in s_1 was more valuable than being in s_2. On the other hand, it is quite clear from the above that the one-step best action for s_3 is to go *Right* for a reward of 7, rather than *Left* for a reward of -1. This is because in looking one step ahead from s_3 we've seen the immediate value of potentially reaching the goal. On the other hand, when we're at s_4, we seem to be completely indifferent between the *Left* and *Right* actions, because both of them have an estimated value of 0. In that situation it doesn't really matter which one we pick, so let's pick *Right*. Consequently, we end up with a one-step best policy

$$\widetilde{\pi}_\sigma = (\mathit{Left}, \mathit{Left}, \mathit{Right}, \mathit{Right}).$$

3.5 Optimal policies and optimal value functions

Now that we've covered all of the above notions, it's actually rather easy to define a notion of what it means to have an optimal policy or an optimal value function. These two ideas are closely tied together, and they go hand-in-hand with the notion of a Q-value as we mentioned in the previous section. The underlying idea is a rather simple one: for our policy to be *optimal*, it would have to get the absolute maximum possible expected utility out of every single state we could end up in, taking into account the fact that it will also continue to maximize the expected utility from all future states. Since an MDP involves uncertainty we're necessarily working with expected utility here—it's not possible to design an agent that always gets the maximum possible reward every time, because it might get unlucky. However, although we can't necessarily maximize our utility, we *can* ensure that we maximize the *expected utility*.

Fortunately, by a theorem we know that there actually exists an *optimal value function* V^*, which correctly describes the maximum expected utility of being in any given state. Additionally, just as we defined a *Q-value estimate* in Equation 7 using a non-optimal value function, we can define the true or optimal Q-value Q^* by using the optimal value function:

$$Q^*(s, a) = \sum_{s'} T(s, a, s')\Big[R(s, a, s') + \gamma\, V^*(s')\Big]. \tag{10}$$

As we stated before, the Q-value is the expected value of being in state s, taking action a, and then following an optimal policy from that point onward.

If we really knew the optimal value function V^*, it would be very easy for us to work out the optimal policy. Since the Q-value is the expected value of taking a given action and behaving optimally thereafter, this means that all we need to do in order to behave optimally for the entirety of an MDP is to always select the action with the highest Q-value from every state!

This is all well and good, but what if we don't know the optimal value function? Well, using the points we've made above, it's actually rather straightforward to define an equation—the Bellman optimality equation—that describes the optimal value function. Basically, since the action with the highest Q-value from

any state must be optimal, it follows that the value of any state must be equal to the maximum Q-value! As such, the Bellman equation looks like this:

$$V^*(s) = \max_a Q^*(s,a) = \max_a \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma V^*(s')\Big]. \quad (11)$$

As you can see, this is just like our very first equation, Equation 1, except that we've now assumed we're always going to take the action with the best value instead of following a fixed policy. Crucially, a value function is optimal *if and only if it satisfies the Bellman equation*. Additionally, just as we defined a one-step best policy in Equation 9, we can define the *optimal policy* as a policy that is a one-step best policy for the optimal value function! The equation is simply

$$\pi^*(s) = \arg\max_a Q^*(s,a) = \arg\max_a \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma V^*(s')\Big], \quad (12)$$

which is just a different way of saying what we said before—an optimal policy always selects the action with the highest Q-value from every state.

The interesting thing about the Bellman equation is that it's defined in terms of only looking *one step ahead*, which might seem unusual at first. After all, how can we discover an optimal value function by only looking at a single step? Well, the key to understanding the Bellman equation is that it is *recursive*. The value function is optimal if, after looking one step ahead, you end up with the same value function again! Because the definition is recursive, it's a simple matter of mathematical induction to show that so will looking two steps ahead, three steps ahead, four steps ahead, and so on to infinity!

In summary, our goal in solving an MDP is to find an *optimal policy*, and we have some nice equations that such a policy satisfies. However, we're still left with a major question—how do we *calculate* an optimal policy? This is where we bring in our two key methods: *value iteration* and *policy iteration*.

4 Value iteration

4.1 Explanation

The basic idea behind value iteration is a rather simple one—let's just use the Bellman equation directly. Unfortunately, since it's recursive, it would be pretty hard for us to do this directly. Instead, we transform it into an iterative method, in which we continually apply the Bellman equation to an initial value function. Each of these steps is called a *Bellman update*. Moreover, it is guaranteed that as we continue to run our value iteration, the value function will get closer to and eventually converge to the optimal value function! The value iteration algorithm is described by the equation

$$V^{t+1}(s) = \max_a \widetilde{Q}^t(s,a) = \max_a \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma V^t(s')\Big], \quad (13)$$

where the superscript t or $t+1$ indicates the current iteration of the algorithm. This is basically the Bellman equation, except with the difference that the left hand side of the equation has V^{t+1} whereas the right-hand side has $\widetilde{Q}^t$ and V^t.

In other words, the value function for the next iteration is always received by using a one-step look-ahead relative to the previous iteration. That's all there is to it. Note also that $\widetilde{Q}^t$ is simply the one-step Q-value estimate for the value function V^t, just like in Equation 7.

4.2 Applying it to our robot problem

So, let's go ahead and try it! Initially, we will start with

$$V^0(s_1) = V^0(s_2) = V^0(s_3) = -1$$
$$V^0(s_4) = 0,$$

where the superscript 0 on the value function indicates that this is our initial value function, for the 0^{th} iteration.

Now, let's work out the one-step Q-value estimates, just like we did in Section 3.4. Putting everything into Equation 7 gives

$$\widetilde{Q}^0(s_1, \textit{Left}) = -1 + 0.95 \cdot -1 = -1.95$$
$$\widetilde{Q}^0(s_1, \textit{Right}) = -1 + 0.95(0.8 \cdot -1 + 0.2 \cdot -1) = -1.95$$
$$\widetilde{Q}^0(s_2, \textit{Left}) = -1 + 0.95(0.8 \cdot -1 + 0.2 \cdot -1) = -1.95$$
$$\widetilde{Q}^0(s_2, \textit{Right}) = -1 + 0.95(0.8 \cdot -1 + 0.2 \cdot -1) = -1.95$$
$$\widetilde{Q}^0(s_3, \textit{Left}) = -1 + 0.95(0.8 \cdot -1 + 0.2 \cdot -1) = -1.95$$
$$\widetilde{Q}^0(s_3, \textit{Right}) = 0.8 \cdot (9 + 0.95 \cdot 0) + 0.2 \cdot (-1 + 0.95 \cdot -1) = 6.81$$
$$\widetilde{Q}^0(s_4, \textit{Left}) = 0 + 0.95 \cdot 0 = 0$$
$$\widetilde{Q}^0(s_4, \textit{Right}) = 0 + 0.95 \cdot 0 = 0.$$

From here, value iteration is pretty straightforward, because we know from Equation 13 that

$$V^{t+1}(s) = \max_a \widetilde{Q}^t(s, a).$$

Consequently, the result is

$$V^1(s_1) = -1.95$$
$$V^1(s_2) = -1.95$$
$$V^1(s_3) = 6.81$$
$$V^1(s_4) = 0.$$

That's one step of value iteration finished! Now onto the next one:

$$\widetilde{Q}^0(s_1, \textit{Left}) = -1 + 0.95 \cdot -1.95 = -2.8525$$
$$\widetilde{Q}^0(s_1, \textit{Right}) = -1 + 0.95(0.8 \cdot -1.95 + 0.2 \cdot -1.95) = -2.8525$$
$$\widetilde{Q}^0(s_2, \textit{Left}) = -1 + 0.95(0.8 \cdot -1.95 + 0.2 \cdot -1.95) = -2.8525$$
$$\widetilde{Q}^0(s_2, \textit{Right}) = -1 + 0.95(0.8 \cdot 6.81 + 0.2 \cdot -1.95) = 3.8051$$
$$\widetilde{Q}^0(s_3, \textit{Left}) = -1 + 0.95(0.8 \cdot -1.95 + 0.2 \cdot 6.81) = -1.1881$$
$$\widetilde{Q}^0(s_3, \textit{Right}) = 0.8 \cdot (9 + 0.95 \cdot 0) + 0.2 \cdot (-1 + 0.95 \cdot 6.81) = 8.2939$$

$$\widetilde{Q}^0(s_4, \mathit{Left}) = 0 + 0.95 \cdot 0 = 0$$
$$\widetilde{Q}^0(s_4, \mathit{Right}) = 0 + 0.95 \cdot 0 = 0,$$

and

$$V^2(s_1) = -2.8525$$
$$V^2(s_2) = 3.8051$$
$$V^2(s_3) = 8.2939$$
$$V^2(s_4) = 0.$$

One more for good measure:

$$V^3(s_1) = 1.349901$$
$$V^3(s_2) = 6.026333$$
$$V^3(s_3) = 8.575841$$
$$V^3(s_4) = 0.$$

4.3 What do we do after we stop iterating?

OK, so what happens if we stop now, at step 3? What policy would we tell the robot to use, based on what we've worked out so far? Well, we can use the same approach we used in Section 3.4—always recommend the action with the highest one-step Q-value. If our value iteration has converged to the true optimal values, or if it's at least close enough, then the one-step best policy will actually be a genuine optimal policy! If we calculated the Q-values for V^3, we would find that the recommended policy would now be

$$\pi^3 = (\mathit{Right}, \mathit{Right}, \mathit{Right}, *),$$

which is already optimal! Note that it doesn't matter whether we go *Left* or *Right* from s_4, because the rewards are always 0 regardless. In fact, we could have stopped with V^2 and we would already have gotten this result. Unfortunately, we didn't know this at the time. Indeed, based on the value iteration algorithm, we don't really have a way of knowing *for sure* when we've reached a sufficiently accurate value function to give us an optimal policy, so we typically just keep iterating until the value function only changes by a negligible amount from step to step. Additionally, in the real world, it might take too long for us to reach a genuine optimal policy, so often we just stop doing value iteration when we run out of time.

5 Policy iteration

5.1 Explanation

The basic idea of policy iteration is pretty similar to value iteration, except that instead of trying to converge a value function towards the optimal value function, we instead attempt to converge a policy towards the optimal policy. For policy iteration, we start with an *initial policy* π^0, and we solve Equation 1 to find the

value function for that policy; using the superscript notation for the time step it looks like this:

$$V^t(s) = \sum_{s'} T(s, \pi^t(s), s')\Big[R(s, \pi^t(s), s') + \gamma V^t(s')\Big]. \tag{14}$$

Just like in Section 3.3, for a given fixed policy π^t this is actually a system of n equations in n unknowns, where n is the number of states. The unknowns in question are simply the value of each state, e.g. $V^t(s_1)$, $V^t(s_2)$, $V^t(s_3)$, $V^t(s_4)$ in our example.

So, once we've worked out the value V^t of our policy π^t, what do we do next? Well, we improve the policy by looking one step ahead, just like we improved our value function by looking one step ahead in Section 3.4. Using the superscript notation in Equation 9 gives us this:

$$\begin{aligned}\pi^{t+1}(s) &= \arg\max_a \widetilde{Q}^t(s, a) \\ &= \arg\max_a \sum_{s'} T(s, a, s')\Big[R(s, a, s') + \gamma V^t(s')\Big].\end{aligned} \tag{15}$$

Basically, then, a single iteration of policy iteration consists of repeatedly applying Equation 14 and Equation 15. In essence, the key to policy iteration is that the policy we use on the next iteration is the policy we get by looking one step ahead using the value of the policy from the previous iteration.

One advantage of policy iteration is that it has a rather straightforward stopping criterion, which follows from the Bellman equation. In short, if a policy is the one-step best policy for its own value function, it must be an optimal policy! This means that we can stop policy iteration once our policy stays the same from one iteration to the next.

5.2 Applying it to our robot problem

Now, let's get down to business, and apply policy iteration to our robot problem. We begin with an initial policy

$$\pi^0 = (\mathit{Left}, \mathit{Left}, \mathit{Left}, \mathit{Left}).$$

which says that the robot goes *Left* in every state. Now, we apply Equation 14 to solve for V^0. In fact, we've already done the work for this in Section 3.3.1, so we can just re-use our result, which was

$$\begin{gathered}V^0(s_4) = 0, \text{ and} \\ V^0(s_1) = V^0(s_2) = V^0(s_3) = -20.\end{gathered}$$

We can also represent this in vector form,

$$\mathbf{v}^0 = \begin{pmatrix} -20 \\ -20 \\ -20 \\ 0 \end{pmatrix}.$$

Now we need to figure out our policy for the next step, π^1, using Equation 9. In order to do this, we just have to calculate the Q-value estimates for each action

from each state, and pick the action with the highest value. This is just like when we calculated the Q-values for value iteration, but this time we're trying to work out the one-step *best action*

$$\arg\max_a \widetilde{Q}^t(s, a)$$

rather than the one-step *highest value*

$$\max_a \widetilde{Q}^t(s, a).$$

So, assuming I haven't made an error, the Q-value estimates look like this:

$$\begin{aligned}
\widetilde{Q}^0(s_1, \mathit{Left}) &= -1 + 0.95 \cdot -20 = -20 \\
\widetilde{Q}^0(s_1, \mathit{Right}) &= -1 + 0.95(0.8 \cdot -20 + 0.2 \cdot -20) = -20 \\
\widetilde{Q}^0(s_2, \mathit{Left}) &= -1 + 0.95(0.8 \cdot -20 + 0.2 \cdot -20) = -20 \\
\widetilde{Q}^0(s_2, \mathit{Right}) &= -1 + 0.95(0.8 \cdot -20 + 0.2 \cdot -20) = -20 \\
\widetilde{Q}^0(s_3, \mathit{Left}) &= -1 + 0.95(0.8 \cdot -20 + 0.2 \cdot -20) = -20 \\
\widetilde{Q}^0(s_3, \mathit{Right}) &= 0.8 \cdot (9 + 0.95 \cdot 0) + 0.2 \cdot (-1 + 0.95 \cdot -20) = 3.2 \\
\widetilde{Q}^0(s_4, \mathit{Left}) &= 0 + 0.95 \cdot 0 = 0 \\
\widetilde{Q}^0(s_4, \mathit{Right}) &= 0 + 0.95 \cdot 0 = 0.
\end{aligned}$$

Consequently, assuming that we stick to the old policy whenever our old action is tied for best action with another one, our next policy for policy iteration is

$$\begin{aligned}
\pi^1(s_1) &= \mathit{Left} \\
\pi^1(s_2) &= \mathit{Left} \\
\pi^1(s_3) &= \mathit{Right} \\
\pi^1(s_4) &= \mathit{Left}.
\end{aligned}$$

As you can see, the policy has gotten a little bit smarter; we've realized that based on what we know so far going right from s_3 is better than going left, and so we've changed our policy to reflect this.

5.3 Another step

Now, let's do another step of policy iteration. First, we have to work out the value of our new policy, π^1, which means solving the system of equations represented by Equation 14. Following our steps from Section 3.3.1, we start by writing down the transition matrix for our particular policy. First, recall our matrices P_{Left} and P_{Right} from Table 1. The rows of the matrices represent the transition probabilities resulting from taking actions *Left* and *Right* respectively, from each of the states. As such, we can construct our transition matrix for the policy π^1 by using the relevant row from P_{Left} or P_{Right} for each of the rows of the matrix P_1, based on the action that π^1 takes for each state. We use subscript notation for the transition matrix in order to avoid confusion with matrix multiplication. In particular,

- the first row of P_1 is the first row of P_{Left},

- the second row of P_1 is the second row of P_{Left},
- the third row of P_1 is the third row of P_{Right}, and
- the last row of P_1 is the last row of P_{Left}.

The result is

$$P_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.8 & 0.2 & 0 & 0 \\ 0 & 0 & 0.2 & 0.8 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Now we work out the expected immediate reward for each of our states, $\mathbb{E}\left[R(s, \pi^1(s), \cdot)\right]$. For s_1 and s_2 this is clearly -1, and for s_4 this is 0. However, for s_3 the expected immediate reward will now be

$$\mathbb{E}\left[R(s, Right, \cdot)\right] = 0.8 \cdot 9 + 0.2 \cdot -1 = 7,$$

and so we have the reward vector

$$\mathbf{r}^1 = \begin{pmatrix} -1 \\ -1 \\ 7 \\ 0 \end{pmatrix}.$$

Now we can use Equation 6 to represent our equations in matrix form, i.e.

$$(I - \gamma P_1)\mathbf{v}^1 = \mathbf{r}^1. \tag{16}$$

Substituting the values in,

$$\left[\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} - 0.95 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.8 & 0.2 & 0 & 0 \\ 0 & 0 & 0.2 & 0.8 \\ 0 & 0 & 0 & 1 \end{pmatrix}\right] \mathbf{v}^1 = \begin{pmatrix} -1 \\ -1 \\ 7 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 0.05 & 0 & 0 & 0 \\ -0.76 & 0.81 & 0 & 0 \\ 0 & 0 & 0.81 & 0.76 \\ 0 & 0 & 0 & 0.05 \end{pmatrix} \mathbf{v}^1 = \begin{pmatrix} -1 \\ -1 \\ 7 \\ 0 \end{pmatrix},$$

which we can solve to give

$$\mathbf{v}^1 = \begin{pmatrix} -20 \\ -20 \\ 8.6420 \\ 0 \end{pmatrix}.$$

Just to make sure we did it correctly, we can use Python to check our working; see Listing 1 below for what this looks like!

```
>>> import numpy as np
>>> P1 = np.matrix([[1, 0, 0, 0], [0.8, 0.2, 0, 0],
[0, 0, 0.2, 0.8], [0, 0, 0, 1]])
>>> P1
matrix([[ 1. ,  0. ,  0. ,  0. ],
        [ 0.8,  0.2,  0. ,  0. ],
        [ 0. ,  0. ,  0.2,  0.8],
        [ 0. ,  0. ,  0. ,  1. ]])
>>> r = np.matrix([-1, -1, 7, 0]).transpose()
>>> r
matrix([[-1],
        [-1],
        [ 7],
        [ 0]])
>>> np.linalg.solve(np.identity(4) - 0.95 * P1, r)
matrix([[-20.        ],
        [-20.        ],
        [  8.64197531],
        [  0.        ]]) .
```

Listing 1: Python console session, to check our working.

Now we calculate Q-values once again, resulting in

$$\widetilde{Q}^1(s_1, \mathit{Left}) = -1 + 0.95 \cdot -20 = -20$$
$$\widetilde{Q}^1(s_1, \mathit{Right}) = -1 + 0.95(0.8 \cdot -20 + 0.2 \cdot -20) = -20$$
$$\widetilde{Q}^1(s_2, \mathit{Left}) = -1 + 0.95(0.8 \cdot -20 + 0.2 \cdot -20) = -20$$
$$\widetilde{Q}^1(s_2, \mathit{Right}) = -1 + 0.95(0.8 \cdot 8.642 + 0.2 \cdot -20) = 1.768$$
$$\widetilde{Q}^1(s_3, \mathit{Left}) = -1 + 0.95(0.8 \cdot -20 + 0.2 \cdot 8.642) = -14.558$$
$$\widetilde{Q}^1(s_3, \mathit{Right}) = 0.8 \cdot (9 + 0.95 \cdot 0) + 0.2 \cdot (-1 + 0.95 \cdot 8.642) = 8.642$$
$$\widetilde{Q}^1(s_4, \mathit{Left}) = 0 + 0.95 \cdot 0 = 0$$
$$\widetilde{Q}^1(s_4, \mathit{Right}) = 0 + 0.95 \cdot 0 = 0.$$

Finally, to finish the step, we use these Q-value estimates to derive a new one-step best policy, which will be

$$\pi^2(s_1) = \mathit{Left}$$
$$\pi^2(s_2) = \mathit{Right}$$
$$\pi^2(s_3) = \mathit{Right}$$
$$\pi^2(s_4) = \mathit{Left}.$$

5.4 Yet another step

Now that we know how to do it, we might as well keep going. This time around, we get

$$P_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.2 & 0.8 & 0 \\ 0 & 0 & 0.2 & 0.8 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

and our reward vector is still the same,

$$\mathbf{r}^2 = \begin{pmatrix} -1 \\ -1 \\ 7 \\ 0 \end{pmatrix}.$$

Solving for the value function gives

$$\mathbf{v}^2 = \begin{pmatrix} -20 \\ 6.8740 \\ 8.6420 \\ 0 \end{pmatrix},$$

and the resulting Q-value estimates are

$$\widetilde{Q}^2(s_1, \textit{Left}) = -1 + 0.95 \cdot -20 = -20$$
$$\widetilde{Q}^2(s_1, \textit{Right}) = -1 + 0.95(0.8 \cdot 6.874 + 0.2 \cdot -20) = 0.424$$
$$\widetilde{Q}^2(s_2, \textit{Left}) = -1 + 0.95(0.8 \cdot -20 + 0.2 \cdot 6.874) = -14.894$$
$$\widetilde{Q}^2(s_2, \textit{Right}) = -1 + 0.95(0.8 \cdot 8.642 + 0.2 \cdot 6.874) = 6.874$$
$$\widetilde{Q}^2(s_3, \textit{Left}) = -1 + 0.95(0.8 \cdot 6.874 + 0.2 \cdot 8.642) = 5.866$$
$$\widetilde{Q}^2(s_3, \textit{Right}) = 0.8 \cdot (9 + 0.95 \cdot 0) + 0.2 \cdot (-1 + 0.95 \cdot 8.642) = 8.642$$
$$\widetilde{Q}^2(s_4, \textit{Left}) = 0 + 0.95 \cdot 0 = 0$$
$$\widetilde{Q}^2(s_4, \textit{Right}) = 0 + 0.95 \cdot 0 = 0.$$

The policy for the next step therefore becomes

$$\pi^3(s_1) = \textit{Right}$$
$$\pi^3(s_2) = \textit{Right}$$
$$\pi^3(s_3) = \textit{Right}$$
$$\pi^3(s_4) = \textit{Left}.$$

5.5 One more time?

We said before that the policy $(\textit{Right}, \textit{Right}, \textit{Right}, \textit{Left})$ was optimal, but we haven't really proved that for sure. So, if we don't know it's optimal and keep running policy iteration, what happens?

Let's find out. The new transition matrix is

$$P_3 = \begin{pmatrix} 0.2 & 0.8 & 0 & 0 \\ 0 & 0.2 & 0.8 & 0 \\ 0 & 0 & 0.2 & 0.8 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

and our reward vector is the same once again,

$$\mathbf{r}^3 = \begin{pmatrix} -1 \\ -1 \\ 7 \\ 0 \end{pmatrix}.$$

Solving for the value function gives

$$\mathbf{v}^3 = \begin{pmatrix} 5.2151 \\ 6.8740 \\ 8.6420 \\ 0 \end{pmatrix},$$

and the resulting Q-value estimates are

$$\begin{aligned}
\widetilde{Q}^3(s_1, \textit{Left}) &= -1 + 0.95 \cdot 5.215 = 3.954 \\
\widetilde{Q}^3(s_1, \textit{Right}) &= -1 + 0.95(0.8 \cdot 6.874 + 0.2 \cdot 5.215) = 5.215 \\
\widetilde{Q}^3(s_2, \textit{Left}) &= -1 + 0.95(0.8 \cdot 5.215 + 0.2 \cdot 6.874) = 4.269 \\
\widetilde{Q}^3(s_2, \textit{Right}) &= -1 + 0.95(0.8 \cdot 8.642 + 0.2 \cdot 6.874) = 6.874 \\
\widetilde{Q}^3(s_3, \textit{Left}) &= -1 + 0.95(0.8 \cdot 6.874 + 0.2 \cdot 8.642) = 5.866 \\
\widetilde{Q}^3(s_3, \textit{Right}) &= 0.8 \cdot (9 + 0.95 \cdot 0) + 0.2 \cdot (-1 + 0.95 \cdot 8.642) = 8.642 \\
\widetilde{Q}^3(s_4, \textit{Left}) &= 0 + 0.95 \cdot 0 = 0 \\
\widetilde{Q}^3(s_4, \textit{Right}) &= 0 + 0.95 \cdot 0 = 0.
\end{aligned}$$

The policy for the next step therefore becomes

$$\begin{aligned}
\pi^4(s_1) &= \textit{Right} \\
\pi^4(s_2) &= \textit{Right} \\
\pi^4(s_3) &= \textit{Right} \\
\pi^4(s_4) &= \textit{Left},
\end{aligned}$$

which is the same policy we had before. Policy iteration has converged, so we're definitely finished now. This means that the policy π^4 is optimal.

6 The optimal policy!

If you look closely at the Q-value estimates $\widetilde{Q}^3$ from policy iteration, you'll notice that something very important has happened. The Q-values satisfy the relationship

$$V^3(s) = \max_a \widetilde{Q}^3(s, a).$$

For example, the value of state 0 is 5.215, and the maximum Q-value from state 0 is also 5.215. This means that our value function V^3 satisfies the Bellman optimality equation (Equation 11), which said that

$$V^*(s) = \max_a Q^*(s, a) = \max_a \sum_{s'} T(s, a, s') \Big[R(s, a, s') + \gamma V^*(s') \Big].$$

In other words, we now know *for sure* that we've found an optimal policy and the optimal value function! In summary, we have

$$\begin{aligned}\pi^*(s_1) &= \textit{Right}\\ \pi^*(s_2) &= \textit{Right}\\ \pi^*(s_3) &= \textit{Right}\\ \pi^*(s_4) &= \textit{Left},\end{aligned}$$

and

$$\begin{aligned}V^*(s_1) &= 5.2151\\ V^*(s_2) &= 6.8740\\ V^*(s_3) &= 8.6420\\ V^*(s_4) &= 0.\end{aligned}$$

Since the value for s_1 is 5.2151, we can now confidently say that the maximum expected utility for the robot is 5.2151 when it starts at state s_1, and that this is precisely the expected utility for following our optimal policy π^*. It's important to note, however, that this is not a unique optimal policy, as the policy $(\textit{Right}, \textit{Right}, \textit{Right}, \textit{Right})$ would also be optimal.