CMPUT 652: Reinforcement Learning with Robots **Fall 2019**

Lecture 8: September 26

Instructor: Rupam Mahmood *Scribe: Kris De Asis*

8.1 Bellman Equation in Matrix Form

Recall the Bellman equation for v_π:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big(r + \gamma v_\pi(s')\big) \tag{8.1}$$

We can separate the immediate reward from the remainder of the return:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)r + \gamma \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) v_\pi(s')$$

Denoting $r(s,a) = \sum_{s',r} p(s',r|s,a)r$ and $p(s'|s,a) = \sum_r p(s',r|s,a)$, we get:

$$v_\pi(s) = \sum_a \pi(a|s) r(s,a) + \gamma \sum_{s'} \Big[\sum_a \pi(a|s) p(s'|s,a)\Big] v_\pi(s')$$

$$v_\pi(s) = r_\pi(s) + \gamma \sum_{s'} p_\pi(s'|s) v_\pi(s') \tag{8.2}$$

where $r_\pi(s) = \sum_a \pi(a|s) r(s,a)$ and $p_\pi(s'|s) = \sum_a \pi(a|s) p(s'|s,a)$. From here, we can write the Bellman equation in matrix form:

$$\begin{bmatrix} v_\pi(s_0) \\ v_\pi(s_1) \\ \vdots \end{bmatrix} = \begin{bmatrix} r_\pi(s_0) \\ r_\pi(s_1) \\ \vdots \end{bmatrix} + \gamma \begin{bmatrix} p_\pi(s_0|s_0) & p_\pi(s_1|s_0) & \dots \\ p_\pi(s_0|s_1) & p_\pi(s_1|s_1) & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} v_\pi(s_0) \\ v_\pi(s_1) \\ \vdots \end{bmatrix}$$

$$\mathbf{v}_\pi = \mathbf{r}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}_\pi \tag{8.3}$$

8.2 Policy Evaluation

Value functions provide a way of valuating policies such that we can say whether one policy is better than another. Based on this, the process of computing a policy's value function is referred to as *policy evaluation*. Here, we'll detail how we can use the Bellman equation to compute value functions. An observation is that the Bellman equation produces a linear system of equations, which can be written in the form $\mathbf{A}\mathbf{v}_\pi = \mathbf{b}$:

$$\mathbf{v}_\pi = \mathbf{r}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}_\pi$$
$$\mathbf{v}_\pi - \gamma \mathbf{P}_\pi \mathbf{v}_\pi = \mathbf{r}_\pi$$
$$(\mathbf{I} - \gamma \mathbf{P}_\pi)\mathbf{v}_\pi = \mathbf{r}_\pi$$

where $\mathbf{A} = \mathbf{I} - \gamma \mathbf{P}_\pi$ and $\mathbf{b} = \mathbf{r}_\pi$. Based on this, we can solve the linear system with:

$$\begin{aligned}\mathbf{v}_\pi &= \mathbf{A}^{-1}\mathbf{b}\\ \mathbf{v}_\pi &= (\mathbf{I} - \gamma \mathbf{P}_\pi)^{-1}\mathbf{r}_\pi \qquad (8.4)\end{aligned}$$

if $\mathbf{I} - \gamma \mathbf{P}_\pi$ is invertible. Because $\mathbf{P}_\pi$ is a stochastic matrix, the largest absolute value of its eigenvalues is 1. As it is being discounted and subtracted from the identity matrix, the real components of its eigenvalues should be positive, and its inverse exists. We can also solve the system iteratively:

$$\mathbf{v}_{k+1} \leftarrow \mathbf{v}_k - \alpha(\mathbf{A}\mathbf{v}_k - \mathbf{b}) \qquad (8.5)$$

where $\mathbf{v}_k$ converges to $\mathbf{v}_\pi$ as $k \to \infty$ (under some stochastic approximation conditions). Of note, $\alpha = 1$ corresponds to evaluating the Bellman equation for each state simultaneously using the current estimates:

$$\begin{aligned}\mathbf{v}_{k+1} &\leftarrow \mathbf{v}_k - (\mathbf{A}\mathbf{v}_k - \mathbf{b})\\ \mathbf{v}_{k+1} &\leftarrow \mathbf{v}_k - \big((\mathbf{I} - \gamma \mathbf{P}_\pi)\mathbf{v}_k - \mathbf{r}_\pi\big)\\ \mathbf{v}_{k+1} &\leftarrow \mathbf{v}_k - (\mathbf{v}_k - \gamma \mathbf{P}_\pi \mathbf{v}_k - \mathbf{r}_\pi)\\ \mathbf{v}_{k+1} &\leftarrow \mathbf{r}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}_k\end{aligned}$$

A key limitation of these approaches are that they require a model of the world. That is, they need complete specification of the environment's rewards to get $\mathbf{r}_\pi$, as well as the state-transition dynamics for $\mathbf{P}_\pi$.

8.3 Policy Iteration

Policy iteration is a process which allows for identifying the optimal policy. It relies on the policy improvement theorem, where producing a new policy that is greedy with respect to the values of the previous policy will result in a policy that is at least as good as the previous. If the new policy does not change after greedifying with respect to the values of the previous policy, then it is an optimal policy.

We denote a greedy-policy operator g, which produces a policy that is greedy with respect to the values of another policy:

$$gq_{\pi_k} = \pi_{k+1}$$

Starting with some initial policy π_0, policy iteration interleaves policy evaluation and policy improvement until the policy stops changing (and is optimal):

$$\pi_0 \to q_{\pi_0} \to gq_{\pi_0} \to q_{\pi_1} \to gq_{\pi_1} \to q_{\pi_2} \to \cdots \to \pi^*.$$

8.4 Value Iteration

Knowing that an optimal policy is one which is greedy with respect to the optimal value function, another approach is to evaluate the Bellman optimality equation. Considering action values, we have the following Bellman optimality equation:

$$\begin{aligned}q_*(s,a) &= \sum_{s',r} p(s',r|s,a)\big(r + \gamma \max_{a'} q_*(s',a')\big)\\ q_*(s,a) &= r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} q_*(s',a')\end{aligned}$$

Which can similarly be written in matrix form:

$$\mathbf{q}_* = \mathbf{r} + \gamma \mathbf{P}_* \mathbf{q}_* \tag{8.6}$$

Of note, the rows now correspond to every state-action pair of the MDP, and the $\mathbf{P}_*$ matrix maps state-action pairs to state-action pairs. Because the first action is specified, the policy determines the next state-action pair in the transition.

Because the policy now depends on the action-values, we have a non-linear system of equations and can't directly solve it with a matrix inverse. We'll have to resort to an iterative approach where at each iteration, the policy (implicitly represented through $\mathbf{P}$) is greedified with respect to the current values.

$$\begin{aligned} \mathbf{q}_{k+1} &\leftarrow \mathbf{q}_k - \alpha(\mathbf{A}_{g\mathbf{q}_k}\mathbf{q}_k - \mathbf{b}) \\ \mathbf{q}_{k+1} &\leftarrow \mathbf{q}_k - \alpha\big((\mathbf{I} - \gamma \mathbf{P}_{g\mathbf{q}_k})\mathbf{q}_k - \mathbf{r}\big) \end{aligned} \tag{8.7}$$

This can be viewed as interleaving policy evaluation and policy improvement at a finer-grained level, where policy improvement is performed after each step of an iterative approach to policy evaluation. Once the values stop changing, the optimal value function has been found and an optimal policy can be inferred by greedifying with respect to it. This process of iteratively solving the Bellman optimality equation is referred to as *value-iteration.*

8.5 Unifying Continuing and Episodic Tasks

Reinforcement learning tasks are typically categorized as one of two kinds of tasks: *Episodic* tasks, where the underlying MDP has terminal (or absorbing) states, and *continuing* tasks, where it does not. A reason for this distinction lies in that episodic returns are finite and can be undiscounted, while continuing tasks often rely on discounting (or some other reward transform) to ensure the infinite sum of rewards is well defined.

However, episodic tasks can also be framed as a continuing task if we allow the discount rate to vary on a state-dependent basis. Consider the undiscounted MDP specifying an episodic task in Figure 8.1.

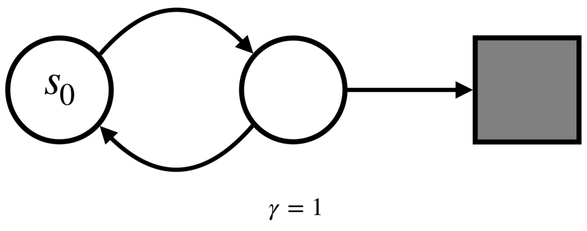


Figure 8.1: An undiscounted episodic task specified using terminal states on a small MDP. Rewards are omitted for clarity.

Here, s_0 denotes a starting state, and a square denotes a terminal state. By allowing state-dependent discounting, a terminal state can equivalently be specified as a state with $\gamma(s) = 0$ and a transition back to a starting state. An MDP with state-dependent discounting which specifies an equivalent undiscounted episodic task can be seen in Figure 8.2.

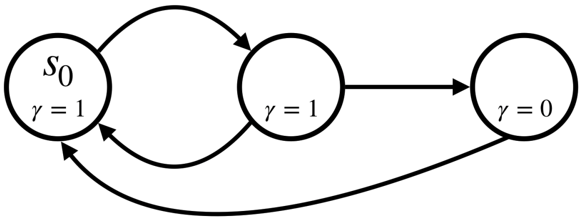


Figure 8.2: An undiscounted episodic task specified as a continuing task with state-dependent discounting. Rewards are omitted for clarity.

Under state-dependent discounting, a future reward is now weighted by the product of discount rates between a time-step and that of when the reward occurs:

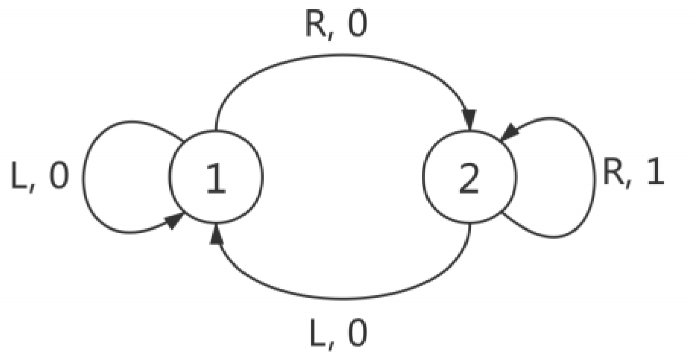
$$G_t = \sum_{k=0}^{\infty} R_{t+k+1} \prod_{i=1}^{k} \gamma(S_{t+i}) \tag{8.8}$$

Based on this, it's evident that the infinite sum is well defined in a continuing setting even with momentary $\gamma(s) \geq 1$, as long as the product of discount rates is 0 in the limit w.p.1.

8.6 Programming Examples

The following subsections contain code for evaluating an equiprobable random policy on the two state MDP detailed in Figure 8.3. The first estimates it by simulating the policy from each state for some number of steps, and averages the sampled returns into each state's value estimate. The second directly solves the Bellman equation as a linear system of equations.

Small Example of MDP



$\gamma = 0.9$

$p(s', r \mid s, a)$	1,0	1,1	2,0	2,1
$p(\cdot, \cdot \mid 1, L)$	1	0	0	0
$p(\cdot, \cdot \mid 1, R)$	0	0	1	0
$p(\cdot, \cdot \mid 2, L)$	1	0	0	0
$p(\cdot, \cdot \mid 2, R)$	0	0	0	1

Figure 8.3: Diagram of the two-state MDP used in the programming examples

8.6.1 Monte Carlo Simulation

```
import numpy as np
import matplotlib.pyplot as plt

# seed for reproducibility
np.random.seed(652)

# exp params
gamma = 0.9
n_runs = 1000
n_steps = 100

# to store stuff
values = np.zeros((2, n_runs + 1))
for s0 in range(2):
  for run in range(n_runs):
    # init state and return
    s = s0
    G = 0.0
    for step in range(n_steps):
      # choose action
      a = np.random.randint(2)
      # env dynamics
      if (s == 0 and a == 0) or (s == 1 and a == 0):
        sp, r = 0, 0
      elif (s == 0 and a == 1):
        sp, r = 1, 0
      else: #(s == 1 and a == 1):
        sp, r = 1, 1
      # accumulate return
      G += r * (gamma ** step)
      # next time step
      s = sp
    # average results
    values[s0, run + 1] = values[s0, run] + (1 / (run + 1)) * (G - values[s0, run])

# print final values for each state
print(values[:, -1].T)

# plot results
for s in range(2):
  plt.plot(np.arange(n_runs + 1), values[s], label='State {}'.format(s))
plt.axis([0 - 0.01 * n_runs, n_runs, 0, 1.01 * np.max(values)])
plt.xlabel('Number of Runs')
plt.ylabel('Estimated Return from Start State')
plt.legend()
plt.show()
```

8.6.2 Solving a Linear System

```
import numpy as np
import matplotlib.pyplot as plt

# discount rate
gamma = 0.9

# expected rewards
r_sa = np.zeros((2, 2))
r_sa[1, 1] = 1.0

# transition dynamics
p_sasp = np.zeros((2, 2, 2))
p_sasp[0, 0, 0] = 1.0
p_sasp[0, 1, 1] = 1.0
p_sasp[1, 0, 0] = 1.0
p_sasp[1, 1, 1] = 1.0

# policy
policy = np.ones((2, 2)) * 0.5

# compute r_pi
r_pi = np.zeros((2, 1))
for s in range(2):
  r_pi[s] = np.dot(policy[s], r_sa[s])
print('r_pi:')
print(r_pi)

# compute P_pi
P_pi = np.zeros((2, 2))
for s in range(2):
  for sp in range(2):
    P_pi[s, sp] = np.dot(policy[s], p_sasp[s, :, sp])
print('P_pi:')
print(P_pi)

# A and b matrices
A = np.eye(2) - gamma * P_pi
b = r_pi

# v = A^(-1)b
v = np.matmul(np.linalg.inv(A), b)
print('v_pi')
print(v)
```