Reinforcement Learning (Draft)

We continue our study of Markov reward processes and decision processes, shifting our emphasis from dynamic programming to reinforcement learning. Reinforcement learning is more generally applicable than dynamic programming, since (i) it does not require sweeps over the entire state space and (ii) it does not depend on the assumption that the probabilistic nature of the environment or the reward structure is known. In this lecture, we compute state and action value functions using only agents' trial-and-error "experiences." The algorithms we study, Monte Carlo estimation, TD-learning, Q-learning and SARSA, incrementally estimate state and action values from sample trajectories.

1 Incremental Estimation

Our present goal is to learn state- and action-value functions from experience.

One plausible estimate of an unknown quantity is simply the average value, say A_k, of k samples, say $z_1, \ldots, z_k$. Given A_k and the $k+1$st sample, rather than recompute the sum of the first k samples, add the value of the $k+1$st sample, and divide by $k+1$, we update A_{k+1} incrementally as follows:

$$\begin{aligned}
A_{k+1} &= \frac{1}{k+1}\sum_{t=1}^{k+1} z_t \\
&= \frac{1}{k+1}\left[z_{k+1} + \sum_{t=1}^{k} z_t\right] \\
&= \frac{1}{k+1}\left[z_{k+1} + kA_k + A_k - A_k\right] \\
&= \frac{1}{k+1}\left[z_{k+1} + (k+1)A_k - A_k\right] \\
&= A_k + \frac{1}{k+1}\left[z_{k+1} - A_k\right] && (1) \\
&= \frac{k}{k+1}A_k + \frac{1}{k+1}z_{k+1} && (2)
\end{aligned}$$

More generally, the value of the $k+1$st sample z_{k+1} in Equation 1 can be replaced by an arbitrary "target" value A. Similarly, the fraction $1/(k+1)$, which decreases with the number of samples, can be generalized by a function $0 < \alpha_t \leq 1$ that decays with time t, in which case $k/(k+1)$ is replaced by $1 - \alpha_t$.

In the following equations, the new estimate A_{t+1} depends in part on the old estimate A_t and in part on the target A, where "in part" is quantified by α_t:

$$\begin{aligned}
A_{t+1} &= (1-\alpha_t)A_t + \alpha_t A && (3) \\
&= A_t + \alpha_t\left[A - A_t\right] && (4)
\end{aligned}$$

Equation 3 generalizes Equation 2; Equation 4 generalizes Equation 1.

2 Learning State Values

Effective techniques for learning state-value functions (i.e., policy evaluation) include Monte Carlo estimation and TD-learning. These algorithms estimate state values incrementally using update rules that specialize Equation 4.

2.1 Monte Carlo Estimation

Given policy π, Monte Carlo methods repeatedly generate state trajectories τ according to π and compute $V^\pi(s_t)$ via Equation 4, setting the target value $A = \rho_t^\tau$ whenever trajectory τ is traversed, as follows:

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha_t[\rho_t^\tau - V^\pi(s_t)] \tag{5}$$

This technique depends on the computation of $\rho_t^\tau = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots$. Thus, it is only applicable if there exists $t' > t$ *s.t.* for all $t'' > t'$, $r_{t''} = 0$: i.e., a zero-reward, absorbing state. (Let $T \subseteq S$ denote the set of absorbing states.)

A policy is called *proper* iff all trajectories it engenders eventually lead to an absorbing state, with probability 1. Assuming the policy π is proper, Monte Carlo methods simulate *episodes*, beginning at a random start state and leading to an absorbing state (with probability 1). Note that for such episodes it is well-defined to simply let ρ_t^τ be the sum of future rewards (i.e., $\gamma = 1$).

```
MONTE_CARLO(MDP, $\pi$, $\gamma$)
  Inputs      policy $\pi$
  Output      value function $V^\pi$
  Initialize  $V = 0$, $\alpha$ according to schedule

repeat
    1. initialize $s, \tau, \rho$
    2. while $s \notin T$ do
        (a) let $\tau = \tau \cup \{s\}$
        (b) take action $a = \pi(s)$
        (c) observe reward $r$, next state $s'$
        (d) for all $s \in \tau$, let $\rho(s) = \rho(s) + r$
        (e) let $s = s'$
    3. for all $s \in \tau$, $V(s) = V(s) + \alpha[\rho(s) - V(s)]$
    4. decay $\alpha$ according to schedule
forever
```

Table 1: Monte Carlo Estimation for $\gamma = 1$. (**Exercise** Rewrite for $0 \leq \gamma \leq 1$.)

2.2 TD-Learning

Recall Bellman's equations for policy evaluation:

$$V^\pi(s_t) = r_t + \gamma \mathbb{E}_{s_{t+1}}[V^\pi(s_{t+1})] \tag{6}$$

TD-learning iteratively computes $V^\pi(s_t)$ via the following instantiation of Eq. 4:

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha_t[r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)] \tag{7}$$

Here $A = r_t + \gamma V^\pi(s_{t+1})$. This error term is called the *temporal difference*. Unlike Monte Carlo methods, which set the target value according to the returns achieved upon termination of a trajectory, TD-learning—inspired by Bellman's theorem—updates based on intermediate rewards. Note that the effectiveness of TD-learning does not depend on the assumption that the policy π is proper.

```
TD_LEARNING(MDP, $\pi$, $\gamma$)
  Inputs      policy $\pi$
              discount factor $\gamma$
  Output      value function $V^\pi$
  Initialize  $V = 0$, $\alpha$ according to schedule

repeat
    1. initialize $s$
    2. while $s \notin T$ do
        (a) take action $a = \pi(s)$
        (b) observe reward $r$, next state $s'$
        (c) $V(s) = V(s) + \alpha[r + \gamma V(s') - V(s)]$
        (d) let $s = s'$
        (e) decay $\alpha$ according to schedule
forever
```

Table 2: TD-Learning.

2.3 Example: Gambler's Ruin

We now compare the behavior of the Monte Carlo method and TD-learning on several sample trajectories in the Gambler's Ruin, for fixed $\alpha = 0.1$ and $\gamma = 1$.

Trajectory	Monte Carlo	TD-learning
4	$V(4) = 0 + .1[1 - 0] = .1$	$V(4) = 0 + .1[1 + 0 - 0] = .1$
$3 \to 4$	$V(3) = 0 + .1[1 - 0] = .1$	$V(3) = 0 + .1[0 + .1 - 0] = .01$
	$V(4) = .1 + .1[1 - .1] = .19$	$V(4) = .1 + .1[1 + 0 - .1] = .19$
$2 \to 3 \to 4$	$V(2) = 0 + .1[1 - 0] = .1$	$V(2) = 0 + .1[0 + .01 - 0] = .001$
	$V(3) = .1 + .1[1 - .1] = .19$	$V(3) = .01 + .1[0 + .19 - .01] = .028$
	$V(4) = .19 + .1[1 - .19] = .271$	$V(4) = .19 + .1[1 + 0 - .19] = .271$
$3 \to 2 \to 1 \to 0$	$V(3) = .19 + .1[0 - .19] = .171$	$V(3) = .028 + .1[0 + .001 - .028] = .0253$
	$V(2) = .1 + .1[0 - .1] = .09$	$V(2) = .001 + .1[0 + 0 - .001] = .0009$
	$V(1) = 0 + .1[0 - 0] = 0$	$V(1) = 0 + .1[0 + 0 - 0] = 0$
	$V(0) = 0 + .1[0 - 0] = 0$	$V(0) = 0 + .1[0 + 0 - 0] = 0$

Given policy π, Monte Carlo estimation and TD-learning are both guaranteed to converge to V^π if the learning rate α_t decreases appropriately over time (fixed values such as 0.1 are often used in practice). TD typically converges faster, because it makes use of intermediate estimates, whereas Monte Carlo estimation methods update based on the final return.

3 Learning Action Values

We now turn our attention to algorithms that learn action-value functions, from which we can derive an optimal policy. We present two learning algorithms: Q-learning and SARSA. Q-learning is an *off-policy* learning algorithm—it learns by exploring the space of states and actions arbitrarily. SARSA is an *on-policy* learning algorithm—it learns by exploring the space of actions prescribed by the current policy. The degree of exploration in Q-learning (or any off-policy algorithm) can greatly exceed that of SARSA (or any on-policy algorithm), leading to faster convergence. In the descriptions that follow, we assume trajectories are given. Later, we describe how to gather data (i.e., trajectories).

Q-Learning Whereas TD-learning is an application of Bellman's theorem for V, Q-learning is based on Bellman's optimality equations for Q:

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma \mathbb{E}[\max_a Q^*(s_{t+1}, a)] \tag{8}$$

The corresponding update rule is the basis for Q-learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{9}$$

In contrast to SARSA, Q-learning is an off-policy learning algorithm, since it need not exploit actions prescribed by the current optimal policy.

SARSA The SARSA learning algorithm is an on-policy algorithm that learns based on $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$. Given state-action pair (s_t, a_t), SARSA simulates the action a_t in state s_t to obtain the reward r_t and transition to state s_{t+1}. The algorithm then uses its current optimal policy—based on the current Q values—to generate its next action a_{t+1} (but with probability ϵ it chooses an action at random). SARSA now updates as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{10}$$

This update is based on the following variant of Bellman's optimality equations:

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma \mathbb{E}[Q^*(s_{t+1}, \pi^*(s_{t+1}))] \tag{11}$$

Exploration vs. Exploitation Recall that in the reinforcement learning framework it is not assumed that the probabilistic nature of the environment is known. Moreover, it is also not assumed that the reward structure is known. Instead, reinforcement learning agents wander through their environments learning about rewards only at the states they visit only for the actions they employ.

Naturally, such agents would aim to reinforce, that is "become more and more likely to employ," those actions that are found to be the most rewarding. With this objective in mind, reinforcement learning agents are susceptible to the trade-offs between exploration and exploitation (as in simulated annealing) while learning action values. By *exploiting* actions that have been proven themselves to be successful in the past, it is possible to perform well; but by *exploring* alternative actions, it is possible to perform even better.

One popular method of exploration is ϵ-greedy: if π is the current optimal policy and s is the current state, with probability $1-\epsilon$, exploit—take action $\pi(s)$—but with probability ϵ, explore—choose an action at random. Typically, ϵ is decayed over time (e.g., $\epsilon \sim 1/t$). This technique, however, explores seemingly optimal and sub-optimal actions with equal probability. An alternative is to use the *softmax* action selection method, which relies on the Boltzmann distribution. Specifically, given state s_t, action a is selected with the following probability:

$$\frac{e^{Q(s_t,a)/T}}{\sum_{a'} e^{Q(s_t,a')/T}}$$

where the temperature parameter T gradually decreases. All actions are nearly equiprobable at initial higher temperatures; in contrast, lower temperatures extol the virtues of some actions but belittle others.

```
Q_LEARNING(MDP, $\gamma$)
  Inputs      discount factor $\gamma$, exploration policy
  Output      action-value function $Q^*$
  Initialize  $Q = 0$, $\alpha$ according to schedule

repeat
    1. initialize $s, a$
    2. while $s \notin T$ do
        (a) take action $a$
        (b) observe reward $r$, next state $s'$
        (c) $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        (d) choose action $a'$ according to the exploration policy
        (e) $s = s'$, $a = a'$
        (f) decay $\alpha$ according to schedule
forever
```

Table 3: Q-Learning: Off-policy reinforcement learning.

3.1 Example: Deterministic Maze

In case of deterministic environments, the update rules for Q-learning and SARSA simplify as follows:

$$Q(s_t, a_t) \leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) \tag{12}$$

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_a Q(s_{t+1}, a) \tag{13}$$

Figure 1 depicts a deterministic maze. Possible moves are indicated by arrows. The final (absorbing) state is **F**; upon transitioning into state **F**, a reward of 100 is obtained. All other rewards are zero.

Let $\gamma = 0.9$. (Since $\gamma < 1$, it functions like a cost in this example.)

```
SARSA(MDP, $\gamma$)
  Inputs      discount factor $\gamma$, exploration rate $\epsilon$
  Output      action-value function $Q^*$
  Initialize  $Q = 0$, $\alpha$ according to schedule

repeat
    1. initialize $s, a, \pi$
    2. while $s \notin T$ do
        (a) take action $a$
        (b) observe reward $r$, next state $s'$
        (c) choose random action $a'$, with probability $\epsilon$
            choose action $a' = \pi(s')$, with probability $1 - \epsilon$
        (d) $Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
        (e) $\pi(s) \in \arg\max_{a''} Q(s, a'')$
        (f) $s = s'$, $a = a'$
        (g) decay $\alpha$ according to schedule
forever
```

Table 4: SARSA: On-policy Reinforcement Learning.

Figure 1: Deterministic Maze.

Value Iteration

$Q(s,a)$	l	r	u	d
A	—	81	81	—
B	0	90	90	—
C	—	90	—	0
D	0	—	100	—
E	0	100	—	0

	$V(s)$
A	81
B	90
C	90
D	100
E	100

$Q(s,a)$	l	r	u	d
A	—	81	81	—
B	73	90	90	—
C	—	90	—	73
D	81	—	100	—
E	81	100	—	81

	$V(s)$
A	81
B	90
C	90
D	100
E	100

Q-Learning

Trajectory	Q-Learning				
D → F	Q(D,u)	=	100 + .9max$_a$ $Q(\mathrm{F},a)$	=	100
E → F	Q(E,r)	=	100 + .9max$_a$ $Q(\mathrm{F},a)$	=	100
C → E → F	Q(C,r)	=	0 + .9max$_a$ $Q(\mathrm{E},a)$	=	90
A → C → E → F	Q(A,u)	=	0 + .9max$_a$ $Q(\mathrm{C},a)$	=	81
B → A → C → E → F	Q(B,l)	=	0 + .9max$_a$ $Q(\mathrm{A},a)$	=	73
D → B → A → C → E → F	Q(D,l)	=	0 + .9max$_a$ $Q(\mathrm{B},a)$	=	66
E → B → D → F	Q(B,r)	=	0 + .9max$_a$ $Q(\mathrm{D},a)$	=	90
	Q(E,d)	=	0 + .9max$_a$ $Q(\mathrm{B},a)$	=	81

SARSA

Trajectory	Q-Learning				
D → F	Q(D,u)	=	100 + .9Q(F,q)	=	100
E → F	Q(E,r)	=	100 + .9Q(F,q)	=	100
C → E → F	Q(C,r)	=	0 + .9Q(E,r)	=	90
A → C → E → F	Q(A,u)	=	0 + .9Q(C,r)	=	81
B → A → C → E → F	Q(B,l)	=	0 + .9Q(A,u)	=	73
D → B → A → C → E → F	Q(D,l)	=	0 + .9Q(B,l)	=	66
E → B → D → F	Q(B,r)	=	0 + .9Q(D,u)	=	90
	Q(E,d)	=	0 + .9Q(B,r)	=	81

4 Eligibility Traces: A Unified View

Recall that TD-learning updates via the target value:

$$R_t^{(1)} = r_t + \gamma V(s_{t+1})$$

In contrast, Monte Carlo methods update via the target value:

$$R_t^{(T)} = r_t + \gamma r_{t+1} + \ldots + \gamma^{T-t} r_T$$

An equally reasonable target value is the n-step return, or lookahead value:

$$R_t^{(n)} = r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n})$$

In fact, any weighted combination of targets of this form is a reasonable target. TD-λ updates as follows: for $0 \leq \lambda < 1$,

$$R_t^{(\lambda)} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

If $\lambda = 0$, TD-λ reduces to simple TD-learning. If $\lambda = 1$, TD-λ reduces to Monte Carlo estimation.

4.1 TD-λ

$$e_{t+1}(s) = \begin{cases} \gamma\lambda e_t(s) + 1 & \text{if } s = s_t \\ \gamma\lambda e_t(s) & \text{otherwise} \end{cases} \tag{14}$$

```
TD(λ)_LEARNING(MDP, $\pi$, $\gamma$, $\lambda$)
 Inputs      policy $\pi$
             discount factor $\gamma$
             exponential weight $\lambda$
 Output      value function $V^\pi$
 Initialize  $V = 0$, $\alpha$ according to schedule

repeat
   1. reset $e$, initialize $s$
   2. while $s \notin T$ do
      (a) $e(s) = e(s) + 1$
      (b) take action $a = \pi(s)$
      (c) observe reward $r$, next state $s'$
      (d) for all $s \in S$
           i. $V(s) = V(s) + \alpha[r + \gamma V(s') - V(s)]e(s)$
          ii. $e(s) = \gamma\lambda e(s)$
      (e) let $s = s'$
      (f) decay $\alpha$ according to schedule
forever
```

Table 5: TD(λ)-Learning.

4.2 SARSA-λ

$$e_{t+1}(s,a) = \begin{cases} \gamma\lambda e_t(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_t(s,a) & \text{otherwise} \end{cases} \tag{15}$$

```
SARSA($\lambda$)(MDP, $\gamma$, $\lambda$)
  Inputs       discount factor $\gamma$
               exponential weight $\lambda$
  Output       action-value function $Q^*$
  Initialize   $Q = 0$, $\alpha$ according to schedule

repeat
    1. reset $e$, initialize $s, a, \pi$
    2. while $s \notin T$ do
        (a) take action $a$
        (b) observe reward $r$, next state $s'$
        (c) choose random action $a'$, with probability $\epsilon$
            choose action $a' = \pi(s')$, with probability $1 - \epsilon$
        (d) $e(s,a) = e(s,a) + 1$
        (e) for all $s \in S$
            i. for all $a \in A$
                A. $Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]e(s,a)$
                B. $e(s,a) = \gamma\lambda e(s,a)$
        (f) $\pi(s) \in \arg\max_{a''} Q(s,a'')$
        (g) $s = s'$, $a = a'$
        (h) decay $\alpha$ according to schedule
forever
```

Table 6: SARSA(λ).

Problems

#1 The figure below depicts a Markov reward process with seven states. Every state except the terminal state has two possible successors, each of which occurs with probability 0.5. Rewards are associated with transitions as shown.

Suppose the following sample trajectories are observed:

1. $6 \xrightarrow{-3} 5 \xrightarrow{-3} 4 \xrightarrow{-3} 3 \xrightarrow{-3} 2 \xrightarrow{-3} 1 \xrightarrow{-1} 0$
2. $6 \xrightarrow{-3} 4 \xrightarrow{-3} 2 \xrightarrow{-3} 0$

Assume the values $V(s)$ are initialized to 0, the learning rate $\alpha = 0.5$, and the discount factor $\gamma = 1$.

(a) After learning from the first of these trajectories, what value does Monte-Carlo learning assign to each state?

(b) After learning from the second of these trajectories, what value does TD learning assign to each state?

(c) *Model-based learning.* An alternative to TD or Monte-Carlo learning is to use the observed sample trajectories to construct a "best guess" at the model that generated those trajectories, and then to solve that model using value iteration. The transition probabilities of the "best guess" model are defined as follows:

$$P(s'|s) = \frac{\text{number of transitions } s \to s' \text{ in all observed trajectories}}{\text{total number of visits to state } s \text{ in all observed trajectories}}$$

The rewards model $R(s, s')$ of state transitions out of s into s' is given by:

$$R(s, s') = \text{the mean of all rewards observed on transitions } s \to s'$$

Draw the "best-guess" model corresponding to the two observed trajectories, and solve it using value iteration.