Lecture 21–22: Reinforcement Learning

TBA

Contents

1 Overview

In this lecture, we continue our study of Markov reward and decision processes, shifting our emphasis from dynamic programming (which has its foundations in operations research) to reinforcement learning (which is true AI). 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 as well as the reward structure are known. In this lecture, we compute state and action value functions using only agents' trial-and-error "experiences." The algorithms we study, Monte Carlo simulations, TD-learning, Q-learning and SARSA, incrementally estimate state and action values from sample trajectories.

2 Incremental Estimation

One plausible estimate of an unknown quantity is simply the average value, say A_k, of k measurements, say $z_1, \ldots, z_k$. Given A_k and the $k+1$st measurement, rather than recompute the sum of the first k measurements, add the value of the $k+1$st measurement, and divide by $k+1$, we update

A_{k+1} incrementally as follows:

$$\begin{aligned} A_{k+1} &= \frac{1}{k+1}\sum_{t=0}^{k} z_{t+1} \\ &= \frac{1}{k+1}\left[z_{k+1} + \sum_{t=0}^{k-1} z_{t+1}\right] \\ &= \frac{1}{k+1}\left[z_{k+1} + kA_k + A_k - A_k\right] \\ &= \frac{1}{k+1}\left[z_{k+1} + (k+1)A_k - A_k\right] \\ &= A_k + \frac{1}{k+1}\left[z_{k+1} - A_k\right] \quad (1) \\ &= \frac{k}{k+1}A_k + \frac{1}{k+1}z_{k+1} \quad (2) \end{aligned}$$

That is, the new estimate A_{k+1} depends in part on the old estimate A_k and in part on the $k+1$st measurement.

More generally, the value of the $k+1$st measurement 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 measurements, can be generalized by a function $0 < \alpha_k \leq 1$ that decays with time t, in which case $k/(k+1)$ is replaced by $1-\alpha_k$.

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

$$\begin{aligned} A_{k+1} &= (1-\alpha_k)A_k + \alpha_k A \quad (3) \\ &= A_k + \alpha_k\left[A - A_k\right] \quad (4) \end{aligned}$$

Equation 3 generalizes Equation 2; Equation 4 generalizes Equation 1. The reinforcement learning update rules we study are all instances of Equation 4.

3 Learning State Values

Effective techniques for learning state-value functions (*e.g.*, policy evaluation) include Monte Carlo policy evaluation and TD-learning. At a high-level, these methods learn state values in an MDP by repeatedly sampling trajectories, and averaging their rewards.

3.1 Monte Carlo Policy Evaluation

Recall that the value $V(s_t)$ of state s_t is defined as the expected reward that is accrued from time t on; that is, the expected value of ρ_t^τ, where ρ_t^τ is the reward that is accrued along trajectory $\tau = (s_t, s_{t+1}, s_{t+2}, \ldots)$:

$$V(s_t) = \sum_\tau P[\tau \mid s_t]\,\rho_t^\tau \quad (5)$$

Given policy π, Monte Carlo policy evaluation repeatedly generates state trajectories τ according to π and computes $V^\pi(s_t)$ via Equation 4, setting the target value $A = \rho_t^\tau$ whenever trajectory τ

is traversed, as follows:

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha_k[\rho_t^\tau - V^\pi(s_t)] \tag{6}$$

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 $t'' > t'$, $r_{t''} = 0$. Given an MDP, an **absorbing** (or terminal) state, is one at which reward is zero and from which it is impossible to depart. In particular, if an absorbing state is reached at time t', then for all $t'' > t'$, $r_{t''} = 0$. 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 policy evaluation simulates **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.*, $\gamma = 1$).

MC_EVALUATION(MDP, π, γ)

Inputs	policy π
	discount factor γ
Output	value function V^π
Initialize	$V = 0$, α according to schedule

`repeat`

1. initialize s, τ, ρ
2. `while` $s \notin T$ `do`
   (a) let $\tau = \tau \cup \{s\}$
   (b) take action $a = \pi(s)$
   (c) observe reward r and 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 α according to schedule

`forever`

Table 1: Monte Carlo Method for Prediction, assuming $\gamma = 1$.

In the pseudocode given in Table 1, the values of the states that are visited during an episode are updated by letting R_t be the value of the returns following the first visit to state s. A variant of this approach instead lets R_t be the *average* value of the returns following every visit to state s. Both methods converge to $V^\pi(s)$ as the number of visits to state s approaches infinity.

3.2 TD-Learning

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

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

Here the target value $A = r_t + \gamma V^\pi(s_{t+1})$. The difference between A and the current estimate $V^\pi(s_t)$ 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. For this reason, TD-learning does not rely on the assumption that the policy π is proper.

TD_LEARNING(MDP, π, γ)	
Inputs	policy π
	discount factor γ
Output	value function V^π
Initialize	$V = 0$, α according to schedule

`repeat`

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

`forever`

Table 2: TD-Learning.

Given policy π, Monte Carlo simulations and TD-learning are both guaranteed to converge to V^π if the learning rate α_k decreases 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 simulation methods update based on the final return.

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

4 Learning Action Values

We now turn our attention to algorithms that learn action-value functions, from which we can derive optimal policies. Following the structure of the previous section, we present one Monte-Carlo based learning algorithm for control, and another, called SARSA, which is based on TD-learning. We also present a third algorithm, Q-learning, that uses an update equation inspired by Bellman's optimality equations. But before presenting any reinforcement learning algorithms for control, we revisit the issue of exploration vs. exploitation, which arises again in this application domain.

4.1 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 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 (as in simulated annealing). All actions are nearly equiprobable at initial higher temperatures; in contrast, lower temperatures extol the virtues of some actions but belittle others.

4.2 Monte Carlo Control

Recall that policy iteration alternates between improving the current policy to arrive at a new policy, and then evaluating that new policy. To extend Monte Carlo evaluation to control, it suffices to insert improvement steps between the repeated evaluation steps (see Table 3).

Note that no Monte Carlo control algorithm can converge to a suboptimal policy. If it were to do so, then the value function corresponding to that policy would eventually be learned (via Monte Carlo evaluation), at which point it would be determined that alternative actions are preferable. Convergence requires both the policy and the value function to be optimal.

MC_CONTROL(MDP, π, γ, ϵ)	
Inputs	policy π discount factor γ rate of exploration ϵ
Output	value function V^π
Initialize	$V = 0$, α according to schedule

`repeat`

1. initialize s, a, τ, ρ
2. `while` $s \notin T$ `do`
   (a) let $\tau = \tau \cup \{(s, a)\}$
   (b) take action $a = \pi(s)$ with probability $1 - \epsilon$
   take random action a with probability ϵ
   (c) observe reward r and next state s'
   (d) for all $(s, a) \in \tau$, let $\rho(s, a) = \rho(s, a) + r$
   (e) let $s = s'$, $a = a'$
3. for all $(s, a) \in \tau$, $Q(s, a) = Q(s, a) + \alpha[\rho(s, a) - Q(s, a)]$
4. for all $s \in S$, $\pi(s) \in \arg\max_a Q(s, a)$
5. decay α according to schedule

`forever`

Table 3: Monte Carlo Method for Control, assuming $\gamma = 1$.

4.3 SARSA

Just as Monte Carlo control is a control algorithm that generalizes Monte Carlo evaluation, SARSA (see Table 4 is a control algorithm that generalizes TD-learning. SARSA updates not just on the trajectory (s_t, r_t, s_{t+1}), but rather on the trajectory $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$. More specifically, 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). At this point, SARSA updates $Q(s_t, a_t)$ as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_k[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{8}$$

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

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma \mathbb{E}[Q^*(s_{t+1}, \pi^*(s_{t+1}))] \tag{9}$$

where

$$\pi^*(s_t) \in \arg\max_a Q^*(s_{t+1}, a) \tag{10}$$

SARSA(MDP, γ, ϵ)	
Inputs	discount factor γ rate of exploration ϵ
Output	action-value function Q^*
Initialize	$Q = 0$, random π, α according to schedule

`repeat`

1. initialize s, a
2. `while` $s \notin T$ `do`
   (a) take action a
   (b) observe reward r and next state s'
   (c) choose random action a', with probability ϵ
   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) $s = s'$, $a = a'$
3. decay α according to schedule

`forever`

Table 4: SARSA: On-policy Reinforcement Learning.

4.4 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:

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

The corresponding update rule is the basis for Q-learning (see Table 5):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_k[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{12}$$

SARSA is an **on-policy** reinforcement learning algorithm, which means that the algorithm learns a policy while simultaneously following that policy (or a close approximation thereof). In contrast, Q-learning is an **off-policy** reinforcement learning algorithm. The policy Q-learning follows while learning need not bear any resemblance to the policy the algorithm is following. Because it learns off-policy, the rate of exploration input to Q-learning (or any off-policy algorithm) can greatly exceed that which is input to SARSA (or any on-policy algorithm) leading to faster convergence. But Q-learning is not prevented from taking actions that are on-policy; doing so leads to behavior that is closely related to that of SARSA.

Q_LEARNING(MDP, γ, ϵ)	
Inputs	discount factor γ rate of exploration ϵ
Output	action-value function Q^*
Initialize	$Q = 0$, α according to schedule

`repeat`

1. initialize s, a
2. `while` $s \notin T$ `do`
   (a) take action a
   (b) observe reward r and 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'
   (e) $s = s'$, $a = a'$
3. decay α according to schedule

`forever`

Table 5: Q-Learning: Off-policy reinforcement learning.

4.5 Example: Deterministic Maze

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

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

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_a Q(s_{t+1}, a) \tag{14}$$

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 $\gamma = 0.9$.

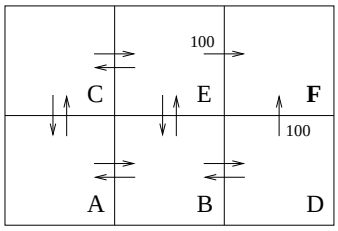

Figure 1: Deterministic Maze.

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