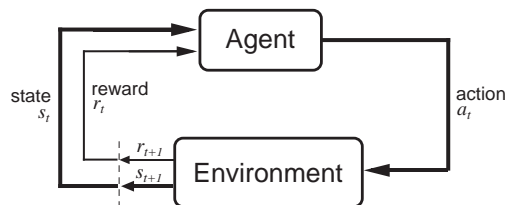

Reinforcement Learning

Slides from Sutton and Barto

The Agent-Environment Interface



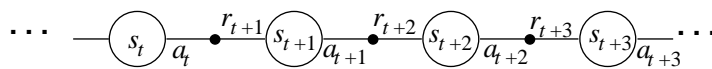
Agent and environment interact at discrete time steps: $t = 0, 1, 2, \dots$

Agent observes state at step t : $s_t \in S$

produces action at step t : $a_t \in A(s_t)$

gets resulting reward: $r_{t+1} \in \mathfrak{R}$

and resulting next state: s_{t+1}



The Agent Learns a Policy

Policy at step t , π_t :

a mapping from states to action probabilities

$\pi_t(s, a)$ = probability that $a_t = a$ when $s_t = s$

- Reinforcement learning methods specify how the agent changes its policy as a result of experience.
- Roughly, the agent's goal is to get as much reward as it can over the long run.

Returns

Suppose the sequence of rewards after step t is :

$$r_{t+1}, r_{t+2}, r_{t+3}, \dots$$

What do we want to maximize?

In general,

we want to maximize the **expected return**, $E\{R_t\}$, for each step t .

Episodic tasks: interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze.

$$R_t = r_{t+1} + r_{t+2} + \dots + r_T,$$

where T is a final time step at which a **terminal state** is reached, ending an episode.

Returns for Continuing Tasks

Continuing tasks: interaction does not have natural episodes.

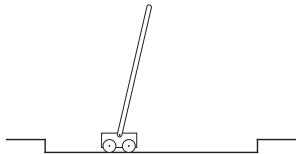
Discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where $\gamma, 0 \leq \gamma \leq 1$, is the **discount rate**.

shortsighted $0 \leftarrow \gamma \rightarrow 1$ farsighted

An Example



Avoid **failure**: the pole falling beyond a critical angle or the cart hitting end of track.

As an **episodic task** where episode ends upon failure:

reward = +1 for each step before failure

\Rightarrow return = number of steps before failure

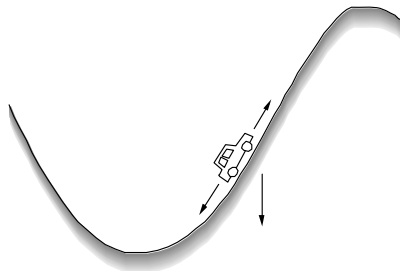
As a **continuing task** with discounted return:

reward = -1 upon failure; 0 otherwise

\Rightarrow return = $-\gamma^k$, for k steps before failure

In either case, return is maximized by avoiding failure for as long as possible.

Another Example



Get to the top of the hill as quickly as possible.

reward = -1 for each step where **not** at top of hill
 \Rightarrow return = - number of steps before reaching top of hill

Return is maximized by minimizing number of steps reach the top of the hill.

A Unified Notation

- In episodic tasks, we number the time steps of each episode starting from zero.
- We usually do not have distinguish between episodes, so we write s_t instead of $s_{t,j}$ for the state at step t of episode j .
- Think of each episode as ending in an absorbing state that always produces reward of zero:



- We can cover all cases by writing $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$,

where γ can be 1 only if a zero reward absorbing state is always reached.

The Markov Property

- By “the state” at step t , the book means whatever information is available to the agent at step t about its environment.
- The state can include immediate “sensations,” highly processed sensations, and structures built up over time from sequences of sensations.
- Ideally, a state should summarize past sensations so as to retain all “essential” information, i.e., it should have the **Markov Property**:

$$\Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \dots, r_1, s_0, a_0\} = \Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}$$

for all s', r , and histories $s_t, a_t, r_t, s_{t-1}, a_{t-1}, \dots, r_1, s_0, a_0$.

Markov Decision Processes

- If a reinforcement learning task has the Markov Property, it is basically a **Markov Decision Process (MDP)**.
- If state and action sets are finite, it is a **finite MDP**.
- To define a finite MDP, you need to give:

- **state and action sets**
- one-step “dynamics” defined by **transition probabilities**:

$$P_{ss'}^a = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \text{ for all } s, s' \in S, a \in A(s).$$

- **reward probabilities**:

$$R_{ss'}^a = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \text{ for all } s, s' \in S, a \in A(s).$$

An Example Finite MDP

Recycling Robot

- ❑ At each step, robot has to decide whether it should (1) actively search for a can, (2) wait for someone to bring it a can, or (3) go to home base and recharge.
- ❑ Searching is better but runs down the battery; if runs out of power while searching, has to be rescued (which is bad).
- ❑ Decisions made on basis of current energy level: high, low.
- ❑ Reward = number of cans collected

Recycling Robot MDP

$S = \{\text{high}, \text{low}\}$

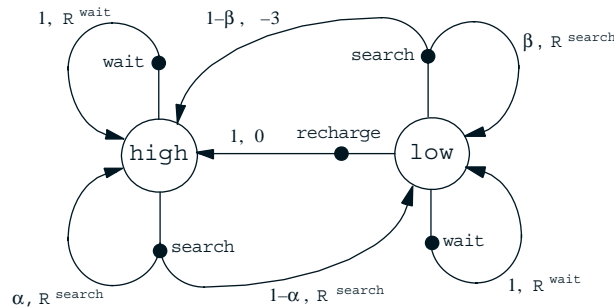
$A(\text{high}) = \{\text{search}, \text{wait}\}$

$A(\text{low}) = \{\text{search}, \text{wait}, \text{recharge}\}$

$R^{\text{search}} =$ expected no. of cans while searching

$R^{\text{wait}} =$ expected no. of cans while waiting

$R^{\text{search}} > R^{\text{wait}}$



Value Functions

- The **value of a state** is the expected return starting from that state; depends on the agent's policy:

State - value function for policy π :

$$V^\pi(s) = E_\pi \{R_t \mid s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\}$$

- The **value of taking an action in a state under policy π** is the expected return starting from that state, taking that action, and thereafter following π :

Action - value function for policy π :

$$Q^\pi(s, a) = E_\pi \{R_t \mid s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\}$$

Bellman Equation for a Policy π

The basic idea:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} \dots \\ &= r_{t+1} + \gamma (r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} \dots) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

So:

$$\begin{aligned} V^\pi(s) &= E_\pi \{R_t \mid s_t = s\} \\ &= E_\pi \{r_{t+1} + \gamma V(s_{t+1}) \mid s_t = s\} \end{aligned}$$

Or, without the expectation operator:

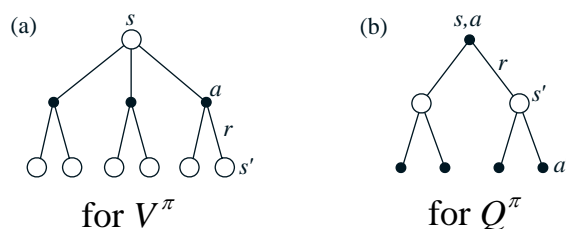
$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')]$$

More on the Bellman Equation

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')]$$

This is a set of equations (in fact, linear), one for each state. The value function for π is its unique solution.

Backup diagrams:



Optimal Value Functions

- ▣ For finite MDPs, policies can be **partially ordered**:
 $\pi \geq \pi'$ if and only if $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in S$
- ▣ There is always at least one (and possibly many) policies that is better than or equal to all the others. This is an **optimal policy**. We denote them all π^* .
- ▣ Optimal policies share the same **optimal state-value function**:

$$V^*(s) = \max_{\pi} V^\pi(s) \quad \text{for all } s \in S$$
- ▣ Optimal policies also share the same **optimal action-value function**:

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a) \quad \text{for all } s \in S \text{ and } a \in A(s)$$

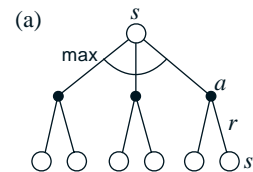
This is the expected return for taking action a in state s and thereafter following an optimal policy.

Bellman Optimality Equation for V^*

The value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned}
 V^*(s) &= \max_{a \in A(s)} Q^{\pi^*}(s, a) \\
 &= \max_{a \in A(s)} E \{ r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a \} \\
 &= \max_{a \in A(s)} \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^*(s')]
 \end{aligned}$$

The relevant backup diagram:

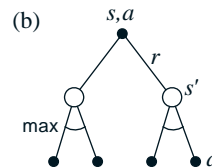


V^* is the unique solution of this system of nonlinear equations.

Bellman Optimality Equation for Q^*

$$\begin{aligned}
 Q^*(s, a) &= E \left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right\} \\
 &= \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma \max_{a'} Q^*(s', a') \right]
 \end{aligned}$$

The relevant backup diagram:



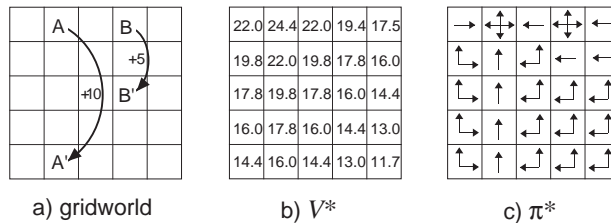
Q^* is the unique solution of this system of nonlinear equations.

Why Optimal State-Value Functions are Useful

Any policy that is greedy with respect to V^* is an optimal policy.

Therefore, given V^* , one-step-ahead search produces the long-term optimal actions.

E.g., back to the gridworld:



What About Optimal Action-Value Functions?

Given Q^* , the agent does not even have to do a one-step-ahead search:

$$\pi^*(s) = \arg \max_{a \in A(s)} Q^*(s, a)$$

Solving the Bellman Optimality Equation

- Finding an optimal policy by solving the Bellman Optimality Equation requires the following:
 - accurate knowledge of environment dynamics;
 - we have enough space and time to do the computation;
 - the Markov Property.
- How much space and time do we need?
 - polynomial in number of states (via dynamic programming methods; Chapter 4),
 - BUT, number of states is often huge (e.g., backgammon has about 10^{20} states).
- We usually have to settle for approximations.
- Many RL methods can be understood as approximately solving the Bellman Optimality Equation.

Policy Evaluation

Policy Evaluation: for a given policy π , compute the state-value function V^π

Recall: **State - value function for policy π :**

$$V^\pi(s) = E_\pi \{R_t \mid s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\}$$


Bellman equation for V^π :

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')]]$$

— a system of $|S|$ simultaneous linear equations

Iterative Methods

$$V_0 \rightarrow V_1 \rightarrow \dots \rightarrow V_k \rightarrow V_{k+1} \rightarrow \dots \rightarrow V^\pi$$

a "sweep" 

A sweep consists of applying a **backup operation** to each state.

A **full policy evaluation backup**:

$$V_{k+1}(s) \leftarrow \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V_k(s')]$$

Iterative Policy Evaluation

```
Input  $\pi$ , the policy to be evaluated
Initialize  $V(s) = 0$ , for all  $s \in \mathcal{S}^+$ 
Repeat
   $\Delta \leftarrow 0$ 
  For each  $s \in \mathcal{S}$ :
     $v \leftarrow V(s)$ 
     $V(s) \leftarrow \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')]$ 
     $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
until  $\Delta < \theta$  (a small positive number)
Output  $V \approx V^\pi$ 
```

Policy Improvement

Suppose we have computed V^π for a deterministic policy π .

For a given state s ,
would it be better to do an action $a \neq \pi(s)$?

The value of doing a in state s is :

$$\begin{aligned} Q^\pi(s, a) &= E_\pi \{ r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a \} \\ &= \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')] \end{aligned}$$

It is better to switch to action a for state s if and only if

$$Q^\pi(s, a) > V^\pi(s)$$

Policy Improvement Cont.

Do this for all states to get a new policy π' that is
greedy with respect to V^π :

$$\begin{aligned} \pi'(s) &= \operatorname{argmax}_a Q^\pi(s, a) \\ &= \operatorname{argmax}_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')] \end{aligned}$$

Then $V^{\pi'} \geq V^\pi$

Policy Improvement Cont.

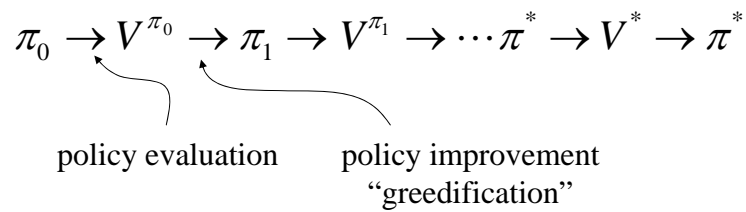
What if $V^{\pi'} = V^{\pi}$?

i.e., for all $s \in S$, $V^{\pi'}(s) = \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^{\pi}(s')] ?$

But this is the Bellman Optimality Equation.

So $V^{\pi'} = V^*$ and both π and π' are optimal policies.

Policy Iteration



Policy Iteration

```
1. Initialization
    $V(s) \in \mathbb{R}$  and  $\pi(s) \in \mathcal{A}(s)$  arbitrarily for all  $s \in \mathcal{S}$ 

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$ 
     For each  $s \in \mathcal{S}$ :
        $v \leftarrow V(s)$ 
        $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} [R_{ss'}^{\pi(s)} + \gamma V(s')]$ 
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
     until  $\Delta < \theta$  (a small positive number)

3. Policy Improvement
   policy-stable  $\leftarrow$  true
   For each  $s \in \mathcal{S}$ :
      $b \leftarrow \pi(s)$ 
      $\pi(s) \leftarrow \arg \max_a \sum_{s'} \mathcal{P}_{ss'}^a [R_{ss'}^a + \gamma V(s')]$ 
     If  $b \neq \pi(s)$ , then policy-stable  $\leftarrow$  false
   If policy-stable, then stop; else go to 2
```

Value Iteration

Recall the **full policy evaluation backup**:

$$V_{k+1}(s) \leftarrow \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V_k(s')]$$

Here is the **full value iteration backup**:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V_k(s')]$$

Value Iteration Cont.

Initialize V arbitrarily, e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat

$\Delta \leftarrow 0$

For each $s \in \mathcal{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π , such that

$\pi(s) = \arg \max_a \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V(s')]$

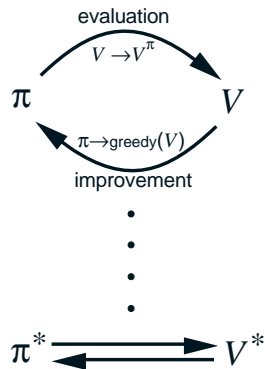
Asynchronous DP

- ❑ All the DP methods described so far require exhaustive sweeps of the entire state set.
- ❑ Asynchronous DP does not use sweeps. Instead it works like this:
 - Repeat until convergence criterion is met:
 - Pick a state at random and apply the appropriate backup
- ❑ Still need lots of computation, but does not get locked into hopelessly long sweeps
- ❑ Can you select states to backup intelligently? YES: an agent's experience can act as a guide.

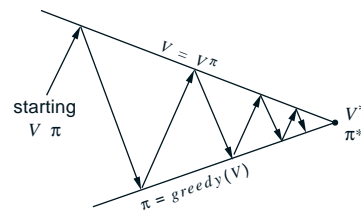
Generalized Policy Iteration

Generalized Policy Iteration (GPI):

any interaction of policy evaluation and policy improvement, independent of their granularity.



A geometric metaphor for convergence of GPI:



Efficiency of DP

- ❑ To find an optimal policy is polynomial in the number of states...
- ❑ BUT, the number of states is often astronomical, e.g., often growing exponentially with the number of state variables (what Bellman called “the curse of dimensionality”).
- ❑ In practice, classical DP can be applied to problems with a few millions of states.
- ❑ Asynchronous DP can be applied to larger problems, and appropriate for parallel computation.
- ❑ It is surprisingly easy to come up with MDPs for which DP methods are not practical.

TD Prediction

Policy Evaluation (the prediction problem):

for a given policy π , compute the state-value function V^π

Recall: Simple every - visit Monte Carlo method :

$$V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)]$$

↑
target: the actual return after time t

The simplest TD method, TD(0) :

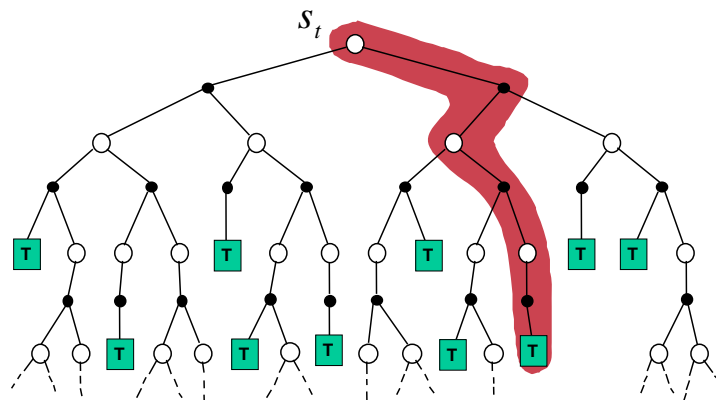
$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

|
target: an estimate of the return

Simple Monte Carlo

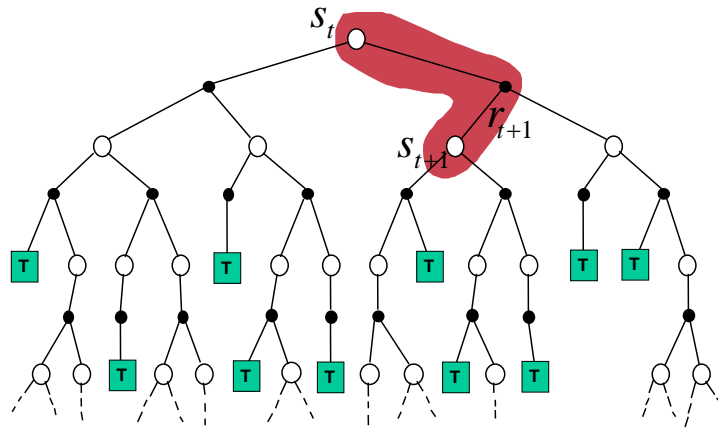
$$V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)]$$

where R_t is the actual return following state s_t .



Simplest TD Method

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

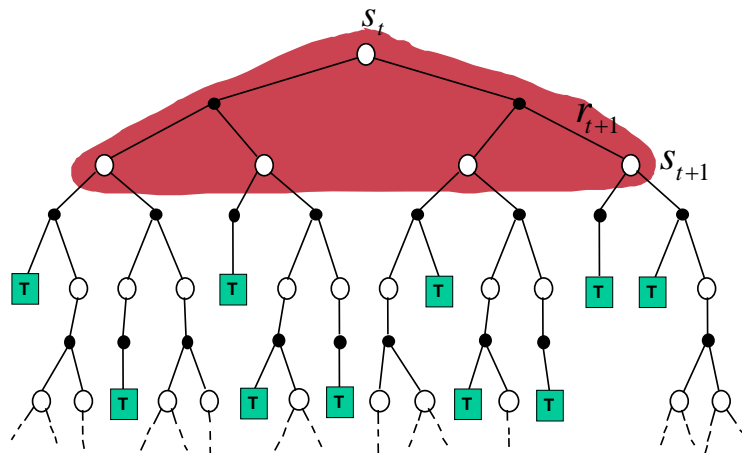


R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction

37

cf. Dynamic Programming

$$V(s_t) \leftarrow E_{\pi} \{r_{t+1} + \gamma V(s_{t+1})\}$$



R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction

38

TD Bootstraps and Samples

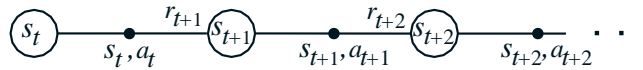
- **Bootstrapping**: update involves an estimate
 - MC does not bootstrap
 - DP bootstraps
 - TD bootstraps
- **Sampling**: update does not involve an expected value
 - MC samples
 - DP does not sample
 - TD samples

Advantages of TD Learning

- TD methods do not require a model of the environment, only experience
- TD, but not MC, methods can be fully incremental
 - You can learn **before** knowing the final outcome
 - Less memory
 - Less peak computation
 - You can learn **without** the final outcome
 - From incomplete sequences
- Both MC and TD converge (under certain assumptions to be detailed later), but which is faster?

Learning An Action-Value Function

Estimate Q^π for the current behavior policy π .



After every transition from a nonterminal state s_t , do this :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1}) = 0$.

Sarsa: On-Policy TD Control

Turn this into a control method by always updating the policy to be greedy with respect to the current estimate:

```
Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
  Repeat (for each step of episode):
    Take action  $a$ , observe  $r, s'$ 
    Choose  $a'$  from  $s'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
     $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'; a \leftarrow a'$ 
  until  $s$  is terminal
```

Q-Learning: Off-Policy TD Control

One - step Q - learning :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Initialize $Q(s, a)$ arbitrarily

Repeat (for each episode):

 Initialize s

 Repeat (for each step of episode):

 Choose a from s using policy derived from Q (e.g., ϵ -greedy)

 Take action a , observe r, s'

$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

$s \leftarrow s'$;

 until s is terminal