

Reinforcement Learning

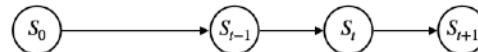
Lecture 2. Model-free Methods

Sungjoon Choi, Korea University

Recap on the Previous Lecture

Markov Process

- A **Markov chain** is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event (aka Markov property).
- Random variable: S_t (state)



Markov Reward Process

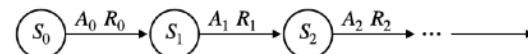
- Now, we obtain rewards as we move to states.



- We have two random variables.
 - State: S_t
 - Reward: R_t
- Since the reward is a random variable, we take expectation to compute the reward function.
 - Reward function: $r(s) = \mathbb{E}[R_{t+1} | S_t = s]$



Markov Decision Process



- Now, we have three random variables:
 - State: S_t
 - Reward: R_t
 - Action: A_t

Bellman Optimality Equation

$$V^*(s) = \max_a Q^*(s, a)$$

$$Q^*(s, a) = \sum_{s'} [r(s, a, s') + \gamma V^*(s')] P(s' | s, a)$$

$$V^*(s) = \max_a \sum_{s'} [r(s, a, s') + \gamma V^*(s')] P(s' | s, a)$$

$$Q^*(s, a) = \sum_{s'} \left[r(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] P(s' | s, a)$$



Introduction

What are **model-free** methods in reinforcement learning?

Introduction

Value Iteration

- Start from the random initial V_0
- For all states $s \in S$:

$$Q_k(s, a) = \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s' | s, a)$$

$$V_{k+1}(s) = \max_{a'} Q_k(s, a')$$

- We now have an explicit form of the policy:

$$\pi_{k+1}(a | s) = 1 \text{ for } a = \arg \max_{a'} Q_k(s, a')$$

Policy Iteration

- Step 1: Policy evaluation:** calculate the value function for a fixed policy until convergence.

- Start from a random initial V_0 .
- Iterate until value converges:

$$V_{k+1}(s) = \sum_a \pi_k(a | s) \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s' | s, a)$$

- Step 2: Policy improvement:** update the policy using one-step lookahead using the converged value function.

- One-step lookahead:

$$Q_{\pi_k}(s, a) = \sum_{s'} [r(s, a, s') + \gamma V_{\pi_k}(s')] P(s' | s, a)$$

$$\pi_{k+1}(a | s) = 1 \text{ for } a = \arg \max_{a'} Q_{\pi_k}(s, a')$$

What is the common **key limitation** of both methods?

Introduction

Value Iteration

- Start from the random initial V_0
- For all states $s \in S$:

$$Q_k(s, a) = \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s' | s, a)$$

$$V_{k+1}(s) = \max_{a'} Q_k(s, a')$$

- We now have an explicit form of the policy:

$$\pi_{k=1}(a | s) = 1 \text{ for } a = \arg \max_{a'} Q_k(s, a')$$

Policy Iteration

- Step 1: Policy evaluation:** calculate the value function for a fixed policy until convergence.

- Start from a random initial V_0 .
- Iterate until value converges:

$$V_{k+1}(s) = \sum_a \pi_t(a | s) \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s' | s, a)$$

- Step 2: Policy improvement:** update the policy using one-step lookahead using the converged value function.

- One-step lookahead:

$$Q_{\pi_k}(s, a) = \sum_{s'} [r(s, a, s') + \gamma V_{\pi_i}(s')] P(s' | s, a)$$

$$\pi_{i+1}(a | s) = 1 \text{ for } a = \arg \max_{a'} Q_{\pi_i}(s, a')$$

What is the common **key limitation** of both methods?

Both methods require state transition probability $P(s' | s, a)$.

Goal

- The goal of a **model-based method** can be written as

Given (S, A, R, P, d) , find π such that $\mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t R_t | \pi \right]$ is maximized.

- The goal of a **model-free method** can be written as

Given (S, A, R) , find π such that $\mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t R_t | \pi \right]$ is maximized.

- Note that the **state transition probability P** and the **initial state distribution d** are omitted in RL.
- Hence, the main challenge is to approximate the **expectation** part!

Model-Free Methods

- In this lecture, we will be learning about model-free prediction methods.
 - Monte-Carlo Learning
 - Temporal-Difference Learning
 - $TD(\lambda)$ and Eligibility Traces
 - SARSA
 - Q-Learning
 - DQN / DDQN / PER

Monte-Carlo Learning

Monte-Carlo Learning

- Monte-Carlo Method
 - A Monte-Carlo method is a broad class of computational algorithms that rely on **repeated random sampling** to obtain numerical results.



Monte Carlo Casino in Monaco

Monte-Carlo Learning

- Monte-Carlo (MC) Method

Expectation of $f(x)$ where $x \sim P(X = x)$: $\mathbb{E}[f(x)] = \int f(x) dP(x)$

MC approximation of $\mathbb{E}[f(x)]$: $\frac{1}{N} \sum_{i=1}^N f(x_i)$ where $x_i \sim P(x)$

- Recall the following definitions:

- The **return** is

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots$$

- The **value function** of π is

$$V_\pi(s) = \mathbb{E}[G_t | S_t = s, \pi]$$

- The **MC policy evaluation** is

$$V_\pi(s) \approx \frac{\sum G_0^k}{K}$$

Monte-Carlo Learning

- Here, an **incremental** Monte-Carlo update rule is used:
- For each state S_t with return G_t

$$N(S_t) \leftarrow N(S_t) + 1$$

$$\widehat{V}(S_t) \leftarrow \widehat{V}(S_t) + \frac{1}{N(S_t)}(G_t - \widehat{V}(S_t))$$

Step size Error correction

- Example (**incremental** averaging):

$$\begin{aligned} \mu_{k+1} &= \frac{1}{k+1} \sum_{j=1}^{k+1} x_j = \frac{1}{k+1} \left(x_{k+1} + \sum_{j=1}^k x_j \right) \\ &= \frac{1}{k+1} (x_{k+1} + k\mu_k) = \mu_k + \frac{1}{k+1} (x_{k+1} - \mu_k) \end{aligned}$$

Step size Error correction

Monte-Carlo Learning

- **Sample** a trajectory with length T (i.e., $\tau = \{s_t, R_t\}_{t=0}^T$)
- Initialize $G_{T+1} = 0$
- From $t = T$ to $t = 0$ (backward)

$$G_t \leftarrow R_{t+1} + \gamma G_{t+1}$$

$$N(s_t) \leftarrow N(s_t) + 1$$

$$\widehat{V}(s_t) \leftarrow \widehat{V}(s_t) + \frac{1}{N(s_t)}(G_t - \widehat{V}(s_t))$$

Monte-Carlo Policy Iteration

- **Sample** a trajectory with length T (i.e., $\tau = \{s_t, a_t, R_t\}_{t=0}^T$)
- Initialize $G_{T+1} = 0$
- From $t = T$ to $t = 0$ (backward)

$$G_t \leftarrow R_{t+1} + \gamma G_{t+1}$$
$$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha(G_t - \widehat{Q}(s_t, a_t))$$

Monte-Carlo Policy Iteration

```

def update_Q(self):
    """
    Update Q
    """

    Q_old = self.Q # [S x A]
    g = 0
    G = Q_old # [S x A]
    for t in reversed(range(len(self.samples))): # for all samples in a reversed way
        state,action,reward,_ = self.samples[t]
        g = reward + self.gamma*g # g = r + gamma * g
        G[state][action] = g # update G

    # Update Q
    self.Q = Q_old + self.alpha*(G - Q_old) # [S x A]
    # Empty memory
    self.samples = []

```

Exponential Moving Average

- Incremental Monte-Carlo update resembles exponential moving average:

$$\widehat{V}(s_t) \leftarrow \widehat{V}(s_t) + \frac{1}{N(s_t)}(G_t - \widehat{V}(s_t)) \text{ (incremental MC update)}$$

$$\widehat{V}(s_t) \leftarrow \left(1 - \frac{1}{N(s_t)}\right) \widehat{V}(s_t) + \frac{1}{N(s_t)} G_t \text{ (rearrange of MC update)}$$

$$\widehat{V}(s_t) \leftarrow (1 - \alpha) \widehat{V}(s_t) + \alpha G_t \text{ (exponential moving average)}$$

- Note that $V(s) = \mathbb{E}[G_t | S_t = s]$.

Monte-Carlo Learning Summary

- MC methods learn directly from episodes of experience.
- MC learning is **model-free**.
 - No knowledge of MDP transitions is required.
- MC learning requires **complete** episodes.
 - It may not be suitable for infinite horizon problems.
- MC learning leverages the simplest possible idea:
 - Value is the expected return.

Temporal Difference Learning

Temporal Difference Learning

- In **MC learning**, we use G_t as an estimator of $V_\pi(s_t)$.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots$$

- How about using the current value estimate $\widehat{V}(s_t)$?
- In other words, we approximate the value using the current value estimate:

$$V(s_t) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots] \approx R_{t+1} + \gamma \widehat{V}(s_{t+1})$$

Temporal Difference Learning

- The goal of **temporal difference (TD) learning** is to estimate V_π online from the experiences under the policy π .
- $TD(\gamma)$ updates $\widehat{V}(s_t)$ towards the estimated return $R_{t+1} + \gamma \widehat{V}(S_{t+1})$:

$$\widehat{V}(s_t) \leftarrow \widehat{V}(s_t) + \alpha \left(R_{t+1} + \gamma \widehat{V}(s_{t+1}) - \widehat{V}(s_t) \right)$$

- $R_{t+1} + \gamma \widehat{V}(s_{t+1})$ is called the **TD target**.
- $\delta_t = R_{t+1} + \gamma \widehat{V}(s_{t+1}) - \widehat{V}(s_t)$ is called the **TD error**.
- Note that the simplest $TD(0)$ update rule becomes:

$$\widehat{V}(s_t) \leftarrow \widehat{V}(s_t) + \alpha \left(R_{t+1} - \widehat{V}(s_t) \right)$$

Monte-Carlo vs. Temporal-Difference

- Advantages of **TD learning**
 - TD learning can learn before knowing the final outcome.
 - TD learning can learn in an **online** manner (after every step (s, a, s')).
 - TD learning works well in infinite-horizon tasks.
- Disadvantages of **MC learning**
 - MC learning must wait until the end of the episode.
 - MC learning requires the complete sequences.
 - MC learning only works for finite-horizon tasks.

Bias Variance Trade-Off

- The return G_t used in **MC learning** is an **unbiased** estimate of $\widehat{V}(s_t)$:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

- TD target $R_{t+1} + \gamma \widehat{V}(s_{t+1})$ is a **biased** estimate of $\widehat{V}(s_t)$:

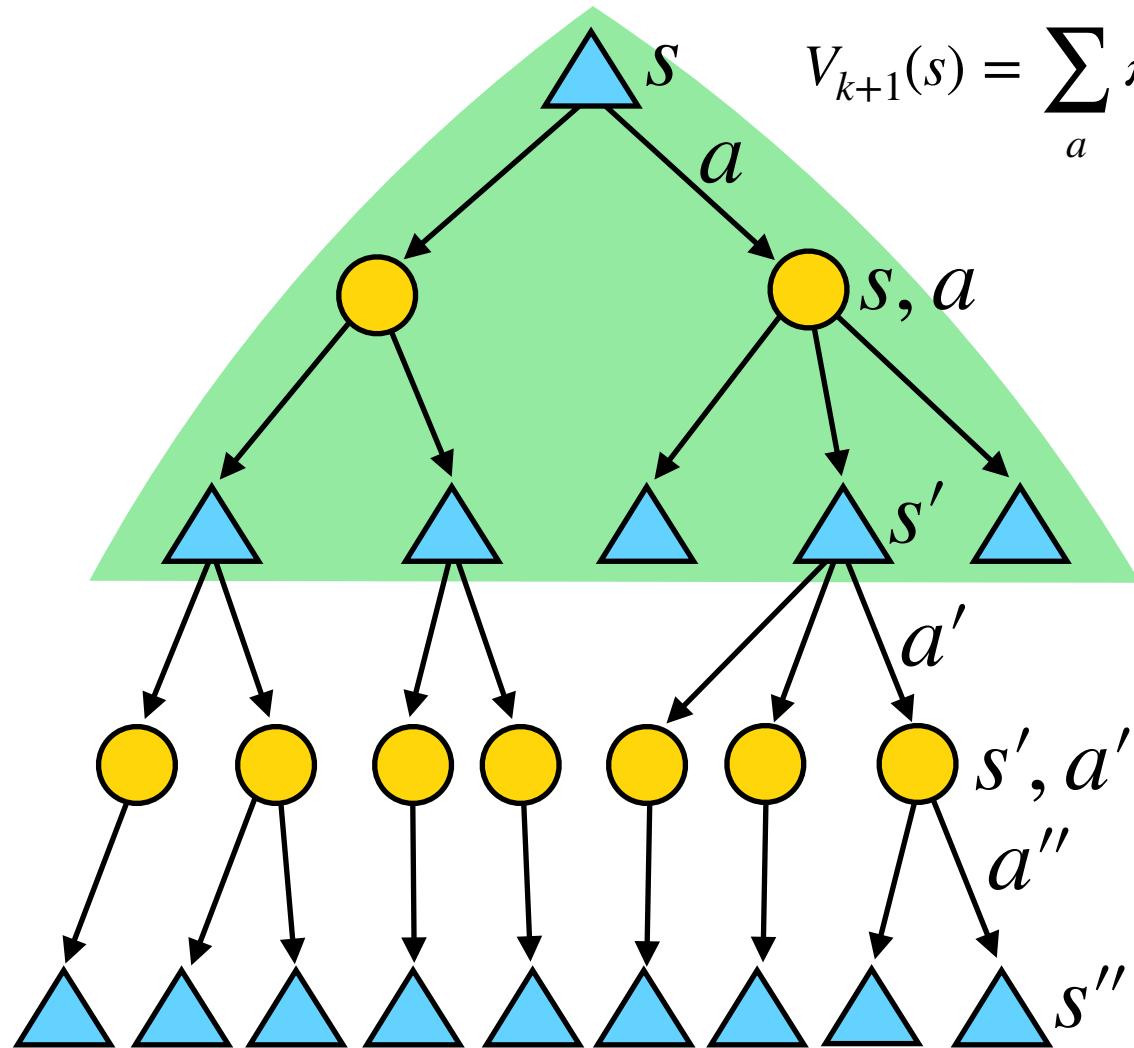
$$\text{TD target} = R_{t+1} + \gamma \widehat{V}(s_{t+1})$$

- However, TD target estimate has much **lower variance** than the return.
 - Return is affected by the **randomness of future** actions, transitions, and rewards.
 - TD target depends on a **single random** action, transition, and reward.

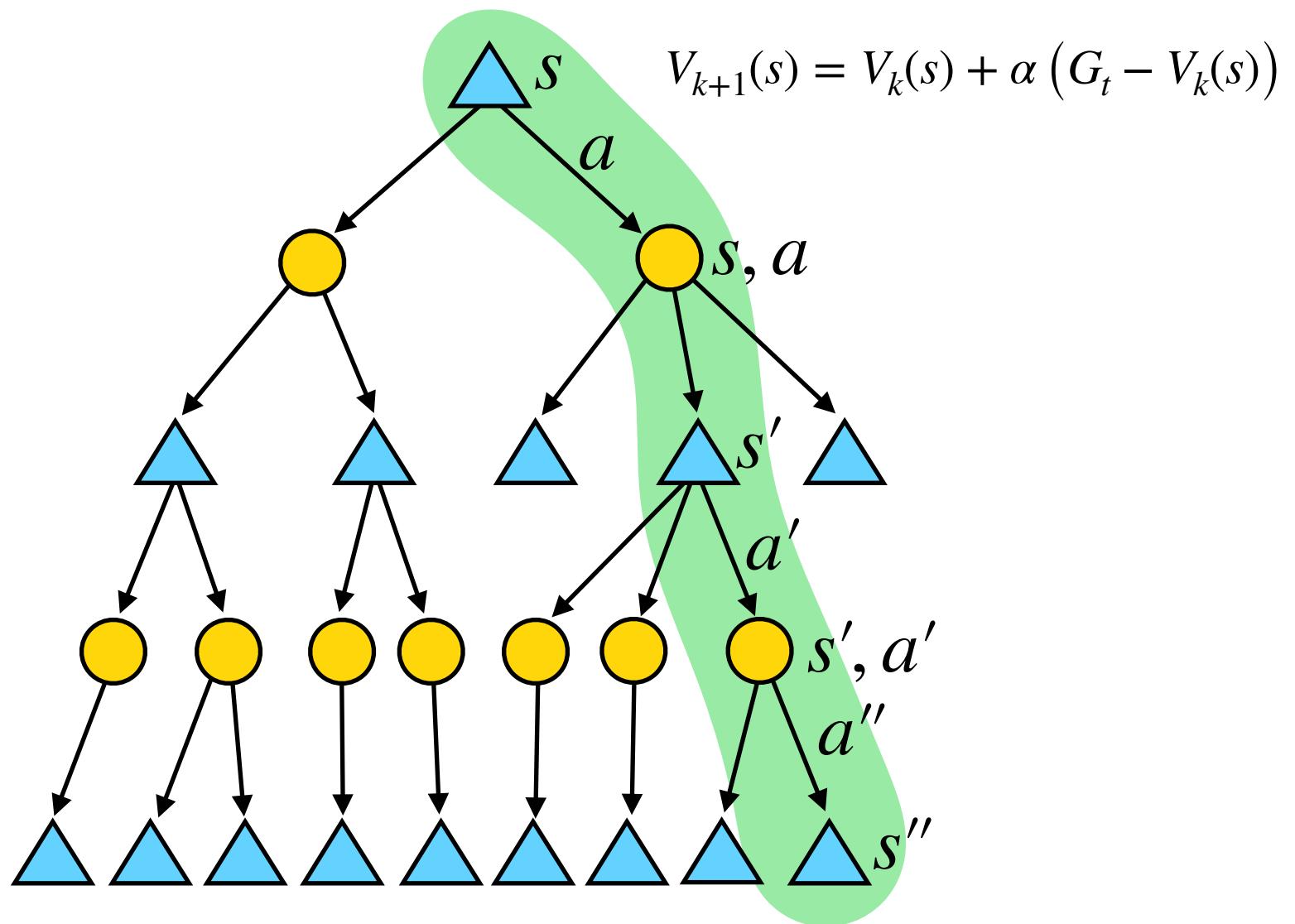
Dynamic Programming

Value Iteration or Policy Iteration

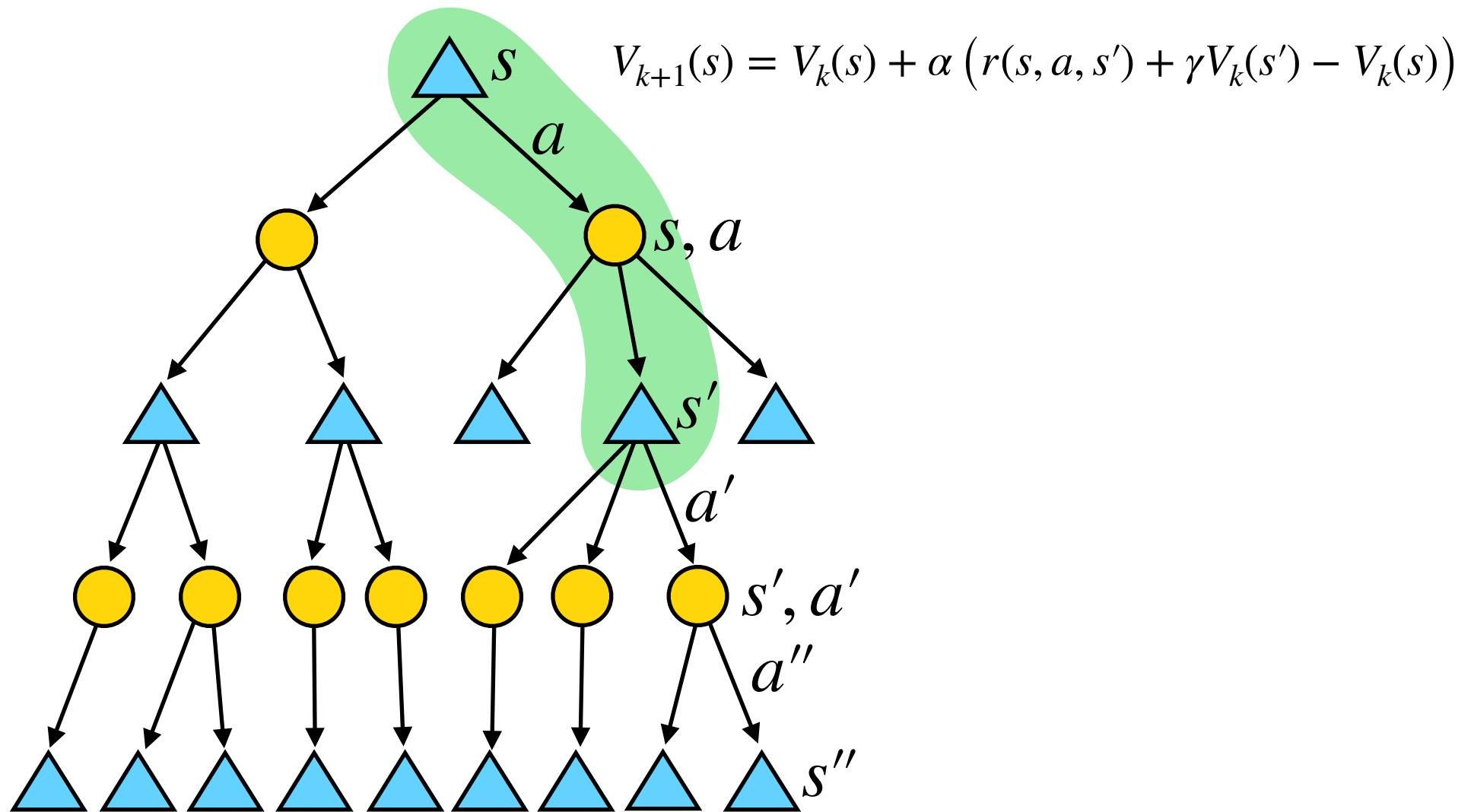
Dynamic Programming



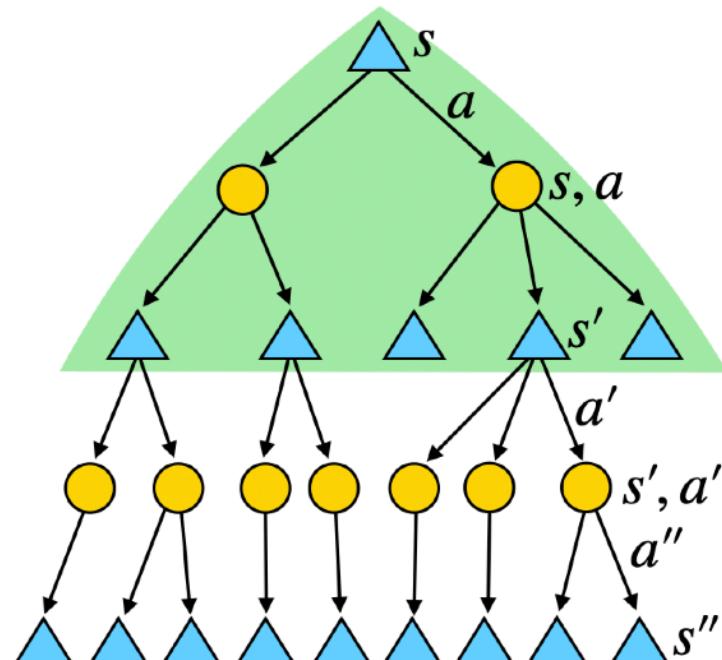
Monte-Carlo Learning



Temporal Difference Learning

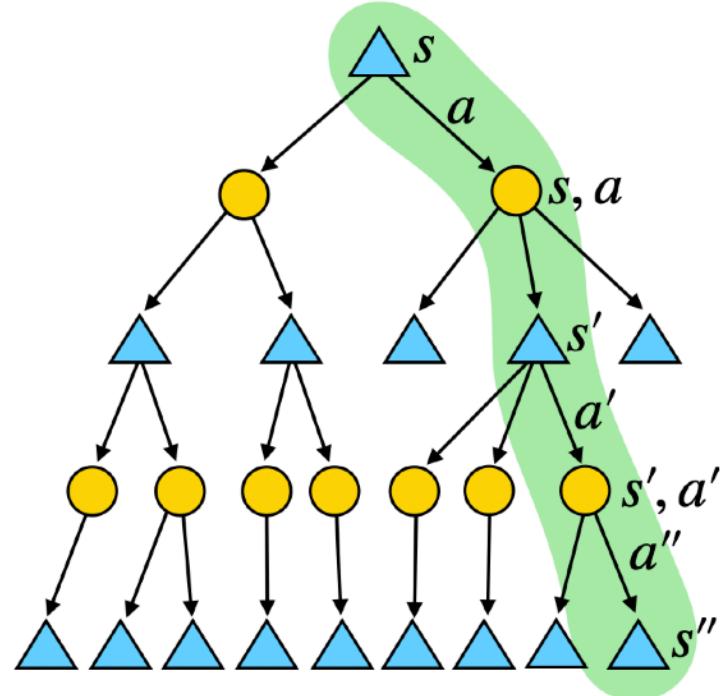


DP vs. MC vs. TD



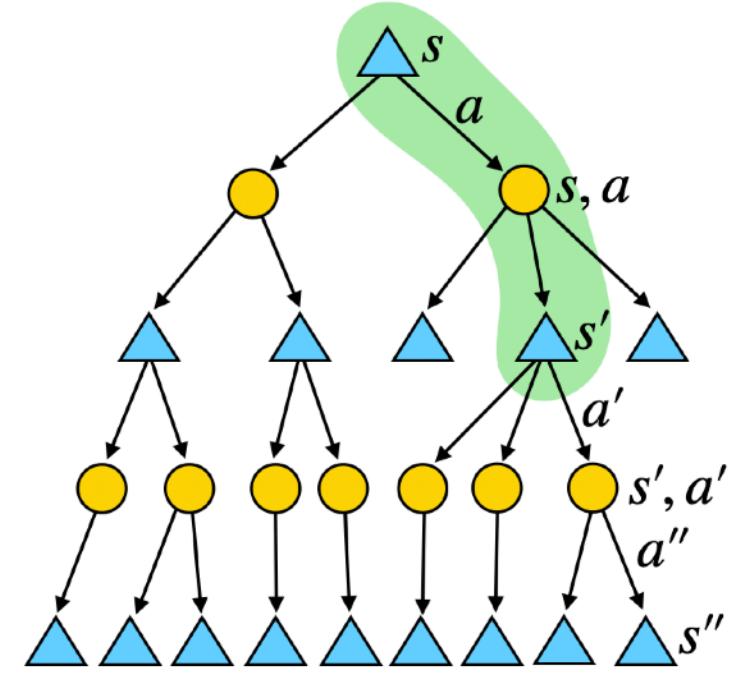
Dynamic Programming

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s'|s, a)$$



Monte-Carlo Learning

$$V_{k+1}(s) = V_k(s) + \alpha (G_t - V_k(s))$$



Temporal Difference Learning

$$V_{k+1}(s) = V_k(s) + \alpha (r(s, a, s') + \gamma V_k(s') - V_k(s))$$

N -step TD Learning

N-step Temporal Difference Learning

- Suppose we have N -step return

$$1\text{-step TD: } G_t^{(1)} = R_{t+1} + \gamma \widehat{V}(s_{t+1})$$

$$2\text{-step TD: } G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 \widehat{V}(s_{t+2})$$

...

$$N\text{-step TD: } G_t^{(N)} = \boxed{R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{N-1} R_{t+N}} + \boxed{\gamma^N \widehat{V}(s_{t+N})}$$

Monte-Carlo Estimate TD Estimate

...

$$\infty\text{-step TD: } G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \gamma^4 R_{t+5} + \cdots \quad (=MC\text{ Learning})$$

- N -step temporal-difference learning

$$V(s) \leftarrow V(s) + \alpha \left(\boxed{G_t^{(N)}} - V(s) \right)$$

N -step TD Target

N-step Temporal Difference Learning



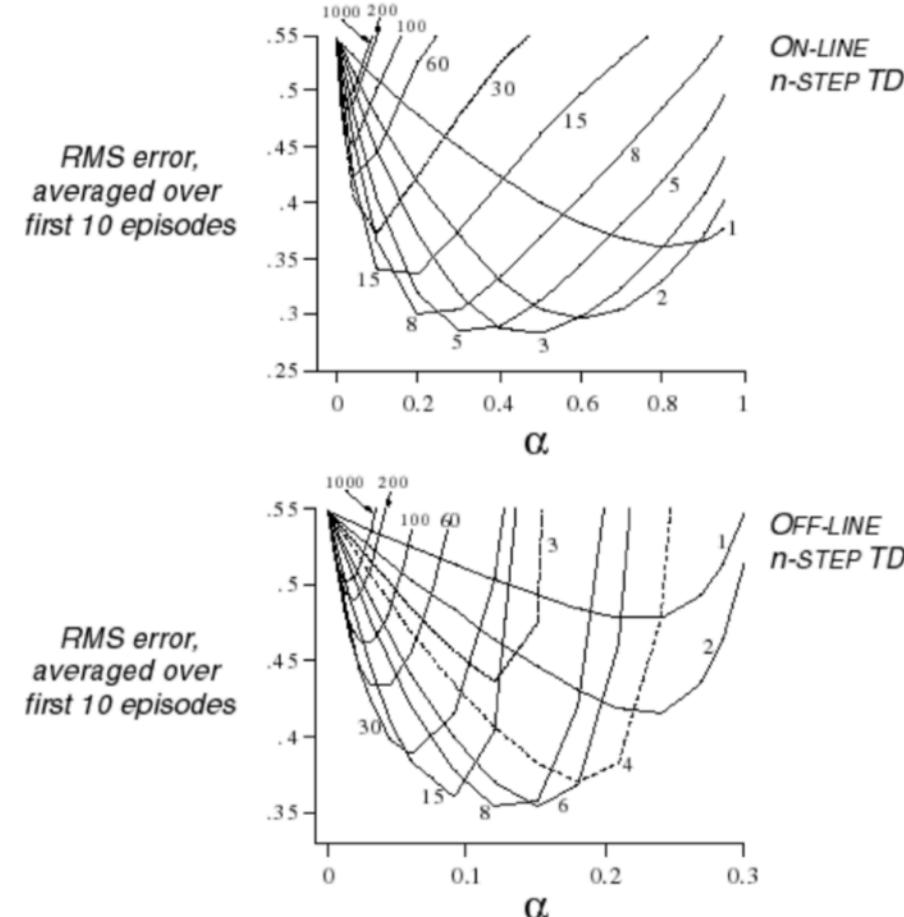
What happens if we increase/decrease N in N -step temporal difference learning?

N-step Temporal Difference Learning



What is the optimal step length N for N -step temporal-difference learning?

N-step Temporal Difference Learning



Recall the **bias-variance trade-off** of Monte-Carlo learning and temporal difference learning.

TD(λ) Learning

TD(λ) Learning

- TD(λ) combines information at every time step using λ -return.
- Like MC learning, it can only be computed from the **complete episodes**.
- λ -Return G_t^λ combines all n -step returns $G_t^{(n)}$.

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{N-1} R_{t+N} + \gamma^N \widehat{V}(s_{t+N})$$

- How?

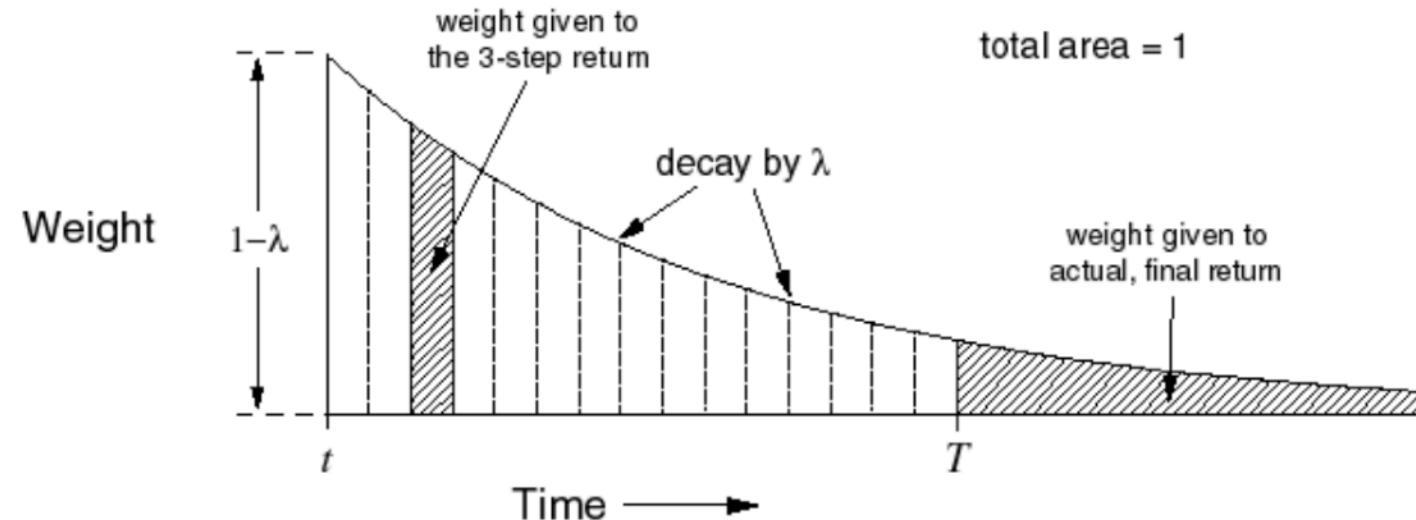
- Using the weight $(1 - \lambda)\lambda^{n-1}$ for each $G_t^{(n)}$

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

- Then, the update of the value function becomes

$$V(S_t) \leftarrow V(S_t) + \alpha (G_t^\lambda - V(S_t))$$

$TD(\lambda)$ Learning

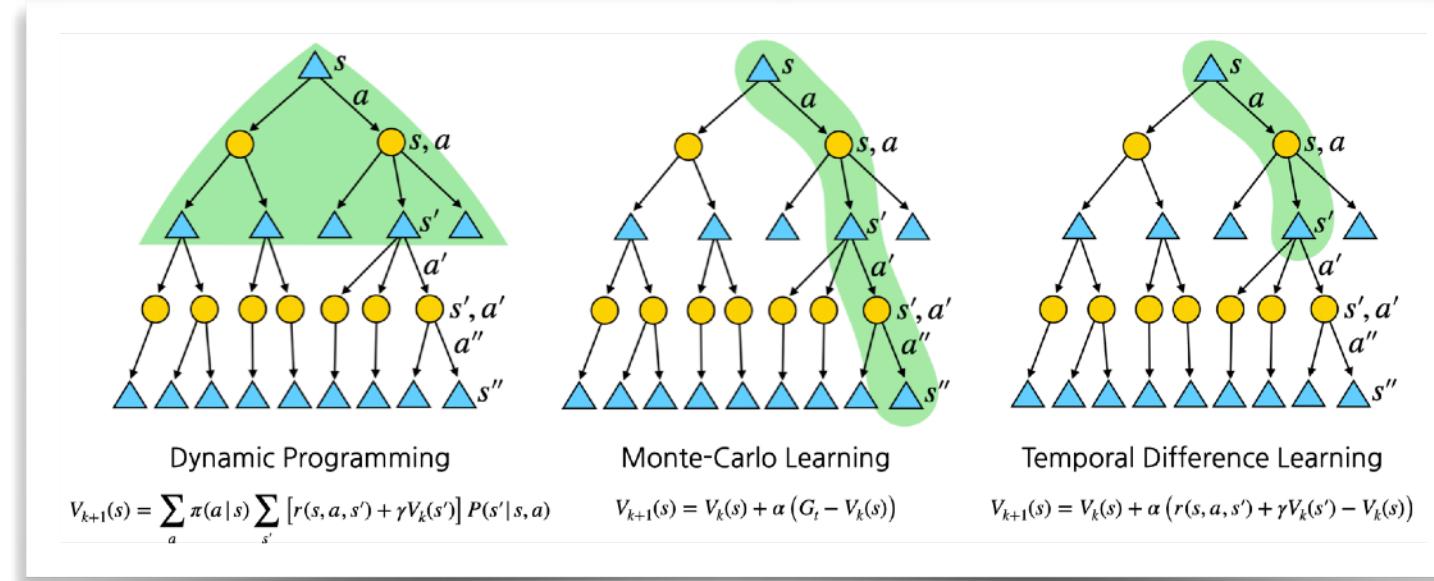


$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{N-1} R_{t+N} + \gamma^N \widehat{V}(s_{t+N})$$

SARSA

SARSA



- So far, we have seen **MC learning**, **TD learning**, and **TD(λ) learning**.
- For such methods, unlike dynamic programming, we need samples (episodes) to estimate the value function.
 - Note that any sequences collected from a random policy will work.
 - However, it may be too **ineffective** to use in practice.
 - How about using an ϵ -greedy policy?

Policy Iteration (revisited)

- **Step 1: Policy evaluation:** calculate the value function for a fixed policy until convergence.

- Start from a random initial V_0 .
- Iterate until value converges:

$$V_{k+1}(s) = \sum_a \pi_i(a | s) \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s' | s, a)$$

- **Step 2: Policy improvement:** update the policy using one-step lookahead using the converged value function.

- One-step lookahead:

$$Q_{\pi_k}(s, a) = \sum_{s'} [r(s, a, s') + \gamma V_{\pi_i}(s')] P(s' | s, a)$$

$$\pi_{i+1}(a | s) = 1 \text{ for } a = \arg \max_{a'} Q_{\pi_i}(s, a')$$

Policy Iteration (revisited)

- **Step 1: Policy evaluation:** calculate the value function for a fixed policy until convergence.

- Start from a random initial V_0 . **State Transition Probability**
- Iterate until value converges:

$$V_{k+1}(s) = \sum_a \pi_i(a | s) \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s' | s, a)$$

- **Step 2: Policy improvement:** update the policy using one-step lookahead using the converged value function.

- One-step lookahead:

$$Q_{\pi_k}(s, a) = \sum_{s'} [r(s, a, s') + \gamma V_{\pi_i}(s')] P(s' | s, a)$$

$$\pi_{i+1}(a | s) = 1 \text{ for } a = \arg \max_{a'} Q_{\pi_i}(s, a')$$

Policy Iteration (revisited)

- **Step 1: Policy evaluation**

- Instead of finding $V(s)$, find $Q(s, a)$.
- Start from a random initial Q_0 .
- Iterate until value converges:

$$Q_{k+1}(s, a) = \sum_{s'} \left[r(s, a, s') + \gamma \sum_{a'} Q_k(s', a') \pi(a' | s') \right]$$

P(s' | s, a)

We still need to handle this.

- **Step 2: Policy improvement**

- Now, we do not need the policy improvement step as we get the policy for free.

$$\pi_{i+1}(a | s) = 1 \text{ for } a = \arg \max_{a'} Q_{\pi_i}(s, a')$$

Model-Free Policy Iteration

- **Step 1: Policy evaluation**

- Estimate $\widehat{Q}(s, a)$ from samples using a model-free method (e.g., MC, TD, or TD(λ)).

$$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha \left(G_t - \widehat{Q}(s_t, a_t) \right) \text{(MC)}$$

$$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha \left(R_{t+1} + \gamma \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t) \right) \text{(TD)}$$

$$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha \left(G_t^\lambda - \widehat{Q}(s_t, a_t) \right) \text{(TD}(\lambda)\text{)}$$

- **Step 2: Policy improvement**

- For the sake of better exploration, we use an ϵ -greedy policy.
 - With probability $1 - \epsilon$, choose the greedy action $a = \arg \max_{a'} Q_{\pi_i}(s, a')$.
 - With probability ϵ , choose a random action.

Model-Free Policy Iteration

- **Step 1: Policy evaluation**

- Estimate $\widehat{Q}(s, a)$ from samples using (S, A, R, S', A')

$$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha \left(R_{t+1} + \gamma \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t) \right) \text{(TD)}$$

- **Step 2: Policy improvement**

- For the sake of better exploration, we use an ϵ -greedy policy.
 - With probability $1 - \epsilon$, choose the greedy action $a = \arg \max_{a'} Q_{\pi_i}(s, a')$.
 - With probability ϵ , choose a random action.

SARSA

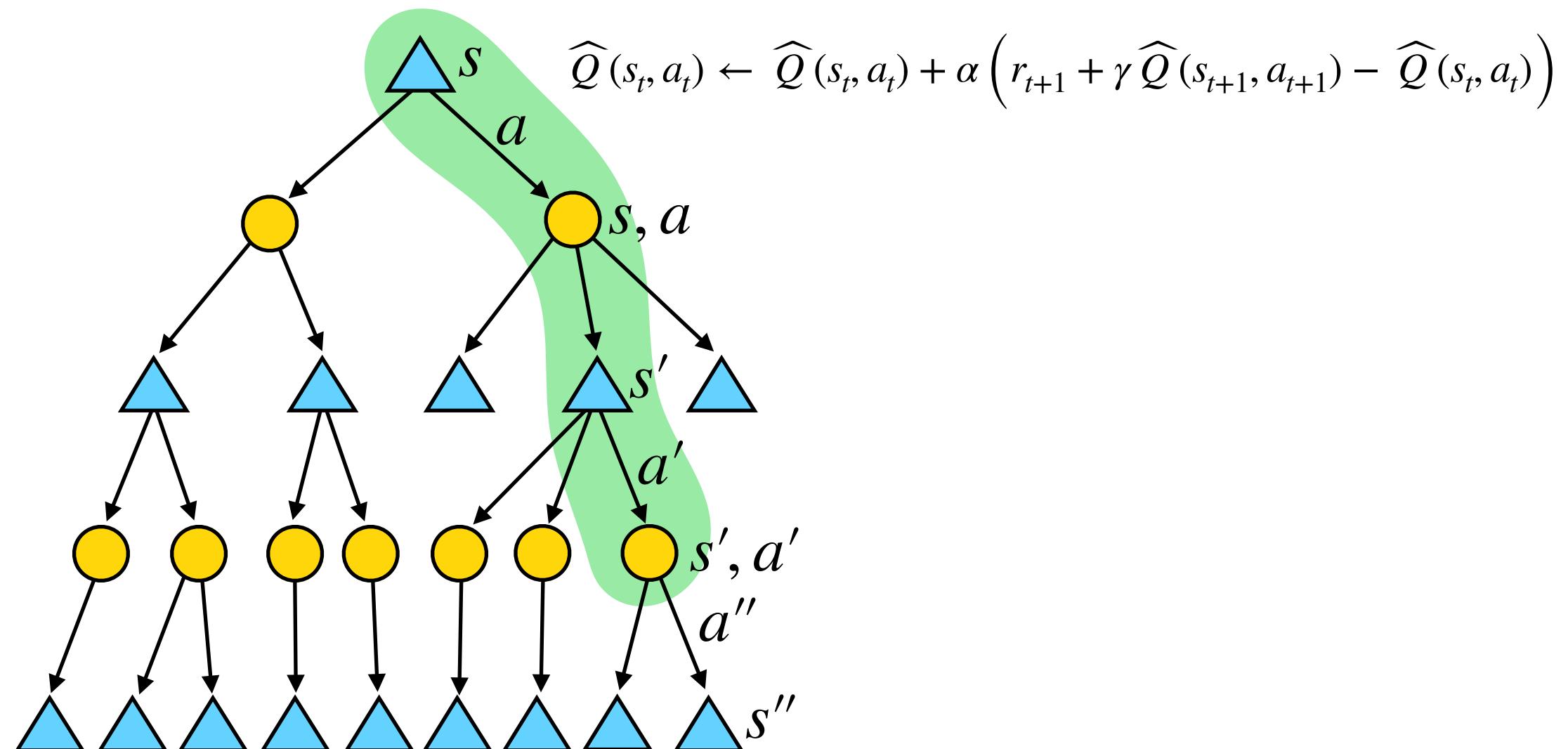
- Initialize $Q(s, a)$
- Repeat (for each episodes)
 - Sample an initial state s_0 .
 - Sample a_0 from an ϵ -greedy policy π .
 - Repeat (for each time step t)
 - Get reward r_{t+1} and next state s_{t+1} .
 - Sample a_{t+1} from the ϵ -greedy policy π .
 - Update $\widehat{Q}(s, a)$ using $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$
 - $\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t) \right)$

SARSA

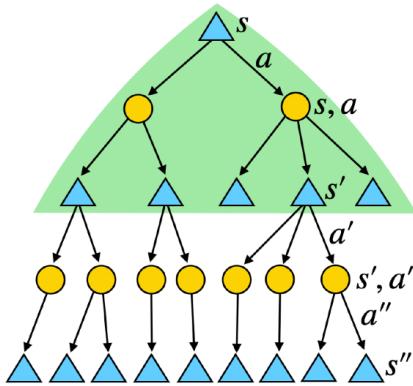


```
def update_Q(self,state,action,reward,state_prime,action_prime,done):
    """
    Update Q value using TD learning
    """
    Q_old = self.Q[state][action]
    if done:
        td_target = reward # for the last step, Q = reward
    else:
        td_target = reward + self.gamma*self.Q[state_prime][action_prime]
    td_error = td_target - Q_old
    # Update Q value
    self.Q[state,action] = Q_old + self.alpha*td_error
```

SARSA

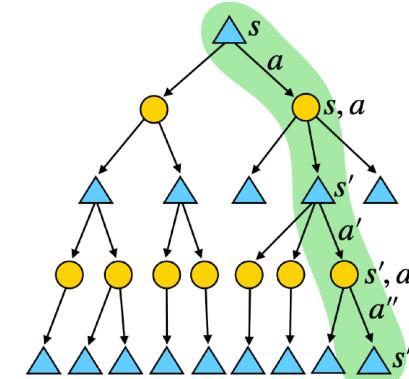


DP vs. MC vs. TD vs. SARSA



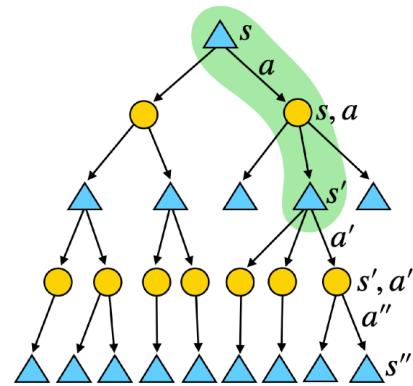
Dynamic Programming

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s'|s, a)$$



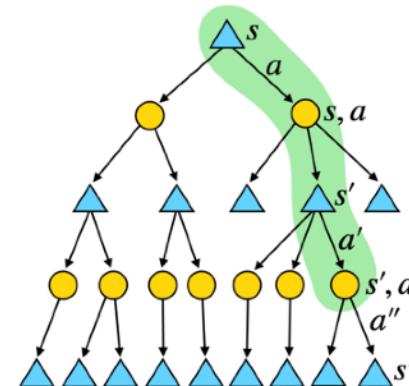
Monte-Carlo Learning

$$V_{k+1}(s) = V_k(s) + \alpha (G_t - V_k(s))$$



Temporal Difference Learning

$$V_{k+1}(s) = V_k(s) + \alpha (r(s, a, s') + \gamma V_k(s') - V_k(s))$$



SARSA

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

On-Policy vs. Off-Policy

On-Policy vs. Off-Policy Learning

- On-Policy Learning (MC, SARSA)
 - Learn the value of the policy π using the episodes sampled from π .
- Off-Policy Learning
 - Learn the value of the policy π using the episodes sampled from **any arbitrary μ** .
- Why is off-policy learning important?
 - Off-policy methods enable learning from observing humans or other agents.
 - Re-use experiences generated from old policies (**experience replay**).
 - Learn the optimal policy while following other exploratory policy.

Importance Sampling for Off-Policy Learning

- Importance Sampling

$$\bullet \mathbb{E}_P[X] \approx \frac{\sum x_i}{N}, x_i \sim P(X)$$

$$\bullet \mathbb{E}_P[X] \approx \frac{\sum \frac{P(x_i)}{Q(x_i)} x_i}{N}, x_i \sim Q(X)$$

- Off-Policy Monte-Carlo

$$\bullet G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots + \gamma^T R_{t+1+T}$$

- When actions are being sampled from μ instead of π :

$$\bullet G_t^{\pi/\mu} = \frac{\pi(a_t | s_t)}{\mu(a_t | s_t)} \frac{\pi(a_{t+1} | s_{t+1})}{\mu(a_{t+1} | s_{t+1})} \dots \frac{\pi(a_{t+1+T} | s_{t+1+T})}{\mu(a_{t+1+T} | s_{t+1+T})} G_t$$

where $\mu()$ is the policy for sampling and $\pi()$ is the true policy to update.

- Importance sample can often dramatically increase the variance.

Importance Sampling for Off-Policy Learning

- Off-Policy Temporal-Difference
 - TD target = $R_{t+1} + \gamma V(S_{t+1})$
 - Weighted TD target: multiply importance sampling corrections along the whole episodes

$$V(s_t) \leftarrow V(s_t) + \alpha \left(\frac{\pi(a_t | s_t)}{\mu(a_t | s_t)} [R_{t+1} + \gamma V(s_{t+1})] - V(s_t) \right)$$

- It has much lower variance than Monte-Carlo importance sampling.

Q-Learning

Q-Learning

- Basic concepts of Q-Learning
 - It is a **model-free value iteration**.

$$V_{k+1}(s) = \max_a \sum_{s'} [r(s, a, s') + \gamma V_k(s')] P(s' | s, a)$$

- Q -values are more useful in terms of getting π .

$$Q_{k+1}(s, a) = \sum_{s'} \left[r(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right] P(s' | s, a)$$

- But we still need the (transition) model $P(s' | s, a)$.
- Suppose that we are using **any behavior policy μ** to get a at any state s , and proceed to the next state s' following $P(s' | s, a)$,

$$Q_{k+1}(s, a) \approx r(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

Q-Learning

- It can be regarded as a **model-free value iteration**.

- **Q-Learning**

- For each time step

$$\widehat{Q}(s, a) \leftarrow \widehat{Q}(s, a) + \alpha \left(r(s, a, s') + \gamma \max_{a'} \widehat{Q}(s', a') - \widehat{Q}(s, a) \right)$$

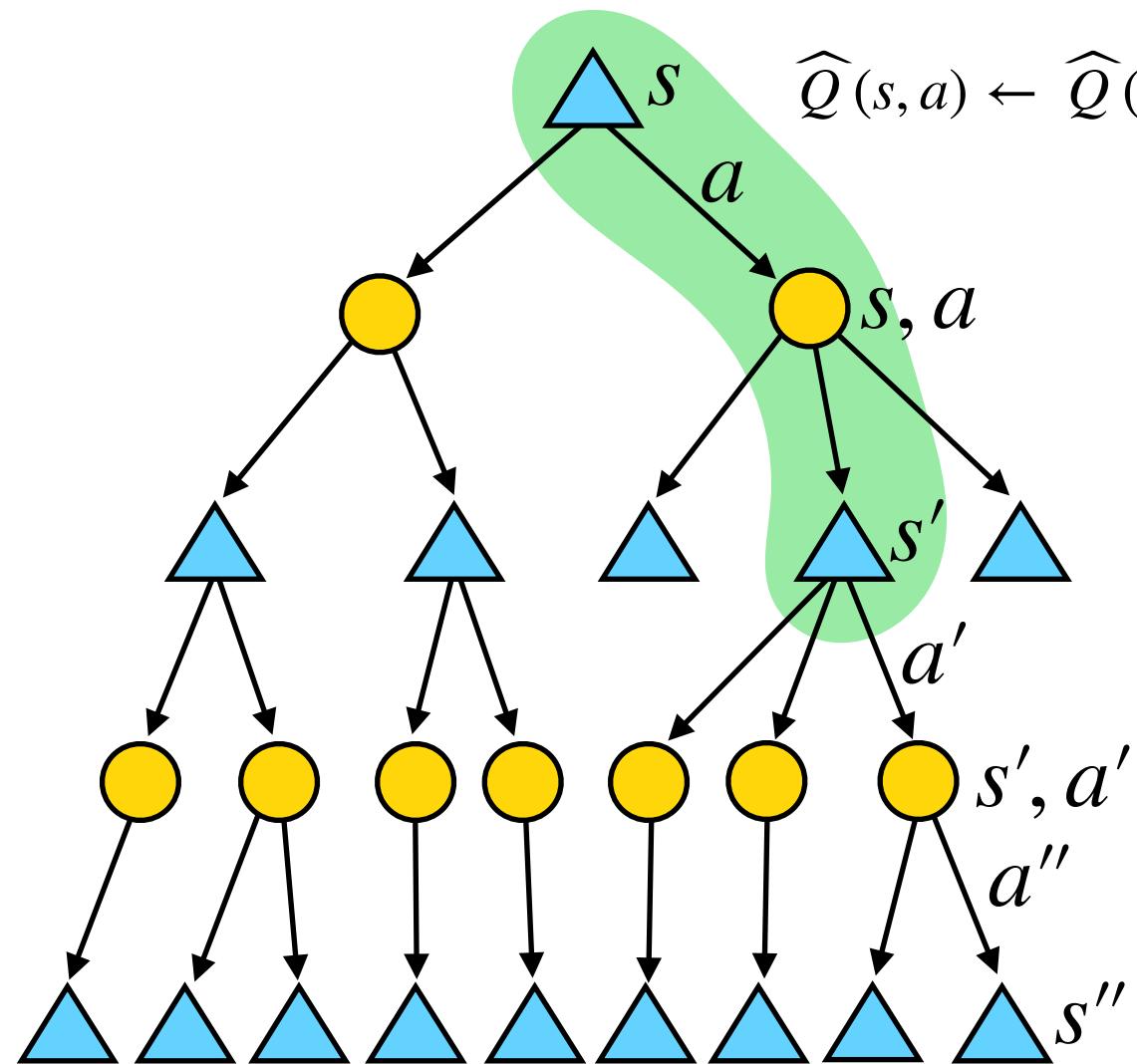
SARSA: $\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha \left(R_{t+1} + \gamma \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t) \right)$ (TD)

- Note that $\max_{a'} \widehat{Q}(s', a')$ does not require the next action (compared to **SARSA**), hence we only need (s, a, s') for **Q-Learning**.
- We can use any **arbitrary behavior policy** to sample episodes as long as $s' \sim P(s'|s, a)$ which enables to utilize experience replays.

Q-Learning

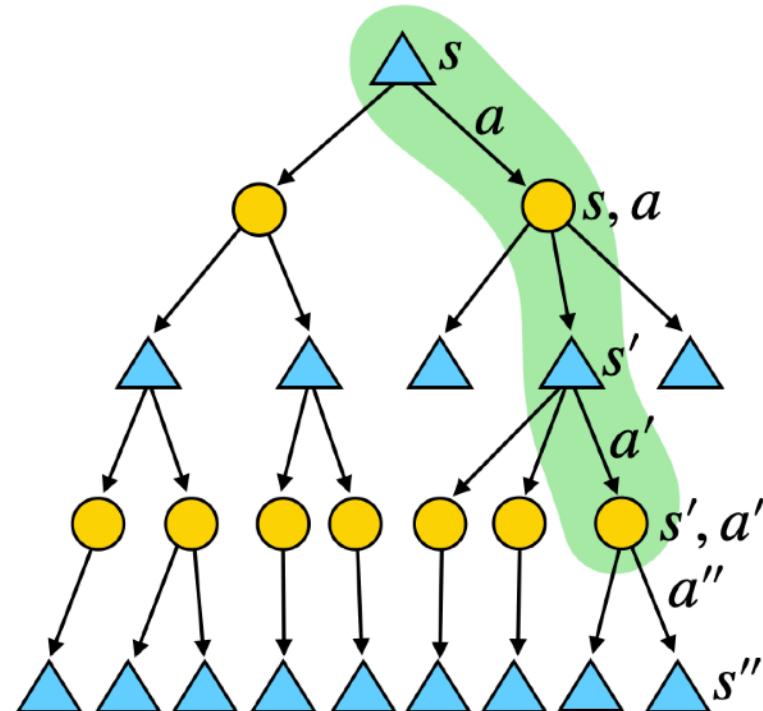
```
def update_value(self,state,action,reward,state_prime,done):
    """
    Update value
    """
    Q_old = self.Q[state][action]
    # TD target
    if done:
        td_target = reward
    else:
        td_target = reward + self.gamma*np.max(self.Q[state_prime])
    td_error = td_target - Q_old # TD error
    self.Q[state,action] = Q_old + self.alpha*td_error # update Q
```

Q-Learning



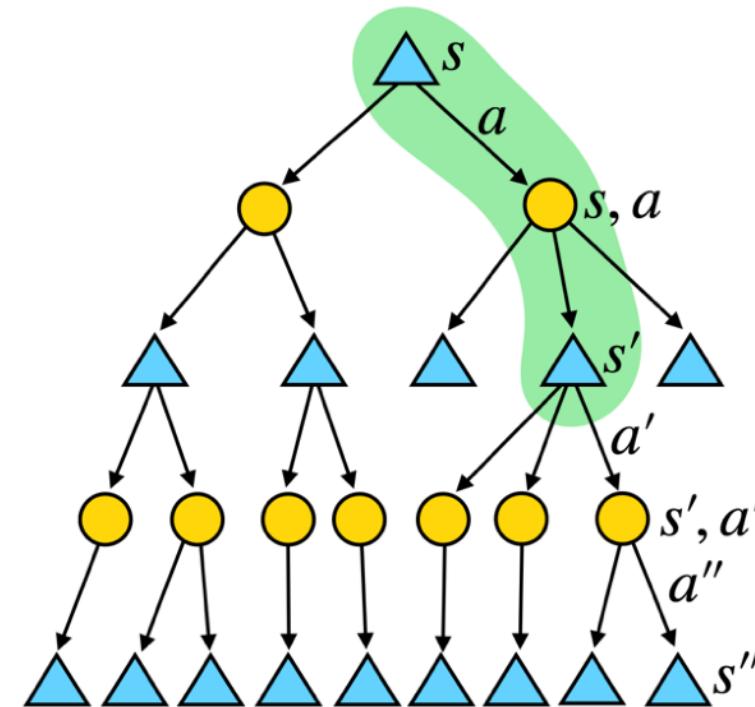
$$\widehat{Q}(s, a) \leftarrow \widehat{Q}(s, a) + \alpha \left(r(s, a, s') + \gamma \max_{a'} \widehat{Q}(s', a') - \widehat{Q}(s, a) \right)$$

SARSA vs. Q-Learning



SARSA

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

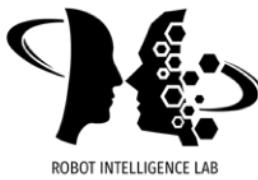


Q-Learning

$$\widehat{Q}(s, a) \leftarrow \widehat{Q}(s, a) + \alpha \left(r(s, a, s') + \gamma \max_{a'} \widehat{Q}(s', a') - \widehat{Q}(s, a) \right)$$

DQN

"Human-level control through deep reinforcement learning," Nature, 2015



Deep Q Network (DQN)

Finally, we have arrived to the **DQN**.

Deep Q Network (DQN)

- Recall the original Q-Learning

$$Q_{k+1}(s, a) \leftarrow Q_k(s, a) + \alpha \left(\boxed{r(s, a, s') + \gamma \max_{a'} Q_k(s', a')} - \boxed{Q_k(s, a)} \right)$$

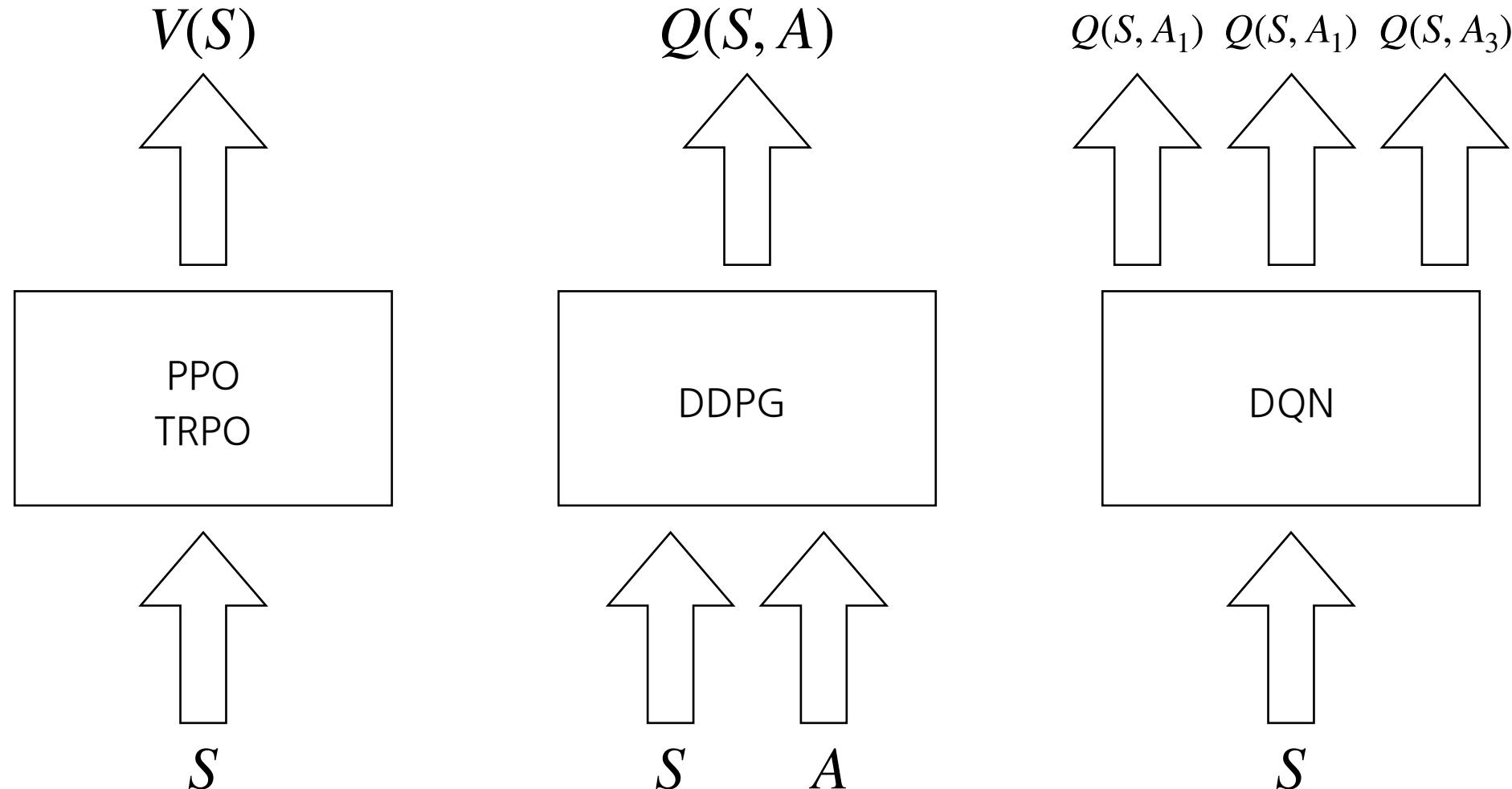
Target Prediction

- From the Q-learning objective, we can derive the following loss function:

$$L(\theta) = \sum_i \left(\boxed{r_i + \gamma \max_{a'} Q(s'_i, a'; \theta)} - \boxed{Q(s_i, a_i; \theta)} \right)^2$$

Target Prediction

Deep Q Network (DQN)



Deep Q Network (DQN)

- (Stable) Update Rule

$$L(\theta) = \sum_i \left(r_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-) - Q(s_i, a_i; \theta) \right)^2$$

- Delayed update
 - For numerical stability, slowly update the target network
 - θ^- : previous parameter (for the Q estimation)
 - θ : current parameter to update
- Other tricks
 - Gradient clipping
 - Input normalization

Deep Q Network (DQN)

- (Stable) Update Rule

$$L(\theta) = \sum_i \left(r_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-) - Q(s_i, a_i; \theta) \right)^2$$

- Delayed update

- For numerical stability,
- θ^- : previous parameter
- θ : current parameter to

- Other tricks

- Gradient clipping
- Input normalization

```

def update_main_network(self, o_batch, a_batch, r_batch, o1_batch, d_batch):
    o1_q = self.target_network(o1_batch)
    max_o1_q = o1_q.max(1)[0].detach().numpy()
    d_batch = d_batch.astype(int)
    expected_q = r_batch + self.gamma*max_o1_q*(1.0-d_batch)
    expected_q = expected_q.astype(np.float64) # R + gamma*max(Q)
    expected_q = torch.from_numpy(expected_q)
    main_q = self.main_network(o_batch).max(1)[0]
    loss = F.smooth_l1_loss(main_q.float(), expected_q.float())

    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    return loss
  
```

Experience Replay

- Experience replay stores transitions (s, a, r, s') to memory.
- To resolve the correlated data problem.
 - Most of machine learning methods assumes that the training data are collected from iid distributions.
- Why is it possible?
 - Q-learning is off-policy learning.
 - Hence, it does not care about the sampling policy.

Double Deep Q Network (DDQN)

- Recall the original DQN loss function

$$L(\theta) = \sum_i \left(r_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-) - Q(s_i, a_i; \theta) \right)^2$$

- Note that the function for selecting the current optimal action and evaluating the optimal action is the same, $Q(s, a; \theta^-)$.
- It often leads to **over-optimism** of the Q function.
- To resolve this issue,

$$L(\theta) = \sum_i \left(\boxed{r_i + \gamma Q(s'_i, \boxed{\arg \max_{a'} Q(s'_i, a'; \theta^-)}; \theta^-)} - \boxed{Q(s_i, a_i; \theta)} \right)^2$$

Target Prediction

Prioritized Experience Replay (PER)

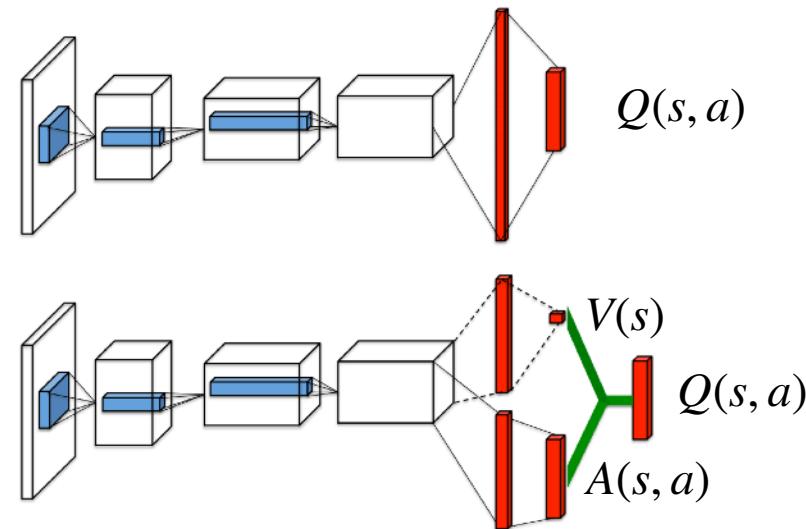
- The intuition is to give **more emphasis** on unfitted data.
- Implementation is simple:
 - Sample k transitions from the experience replay from $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$ where $p_i = |\mathbf{r}_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-) - Q(s_i, a_i; \theta)|$.

- The (weighted) update rule is:

$$L(\theta) = \sum_i w_i \left(\mathbf{r}_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-) - Q(s_i, a_i; \theta) \right)^2$$

where $w_i = \left(\frac{1}{n \cdot p_i} \right)^\beta / \max(w)$.

Dueling Architecture



- The main idea is to separate $Q(s, a)$ into $V(s)$ and $A(s, a)$ which is the advantage.
- Advantage function
 - $Q_\pi(s, a) = V_\pi(s) + A_\pi(s, a)$
 - $\arg \max_{a'} A(s, a') = \arg \max_{a'} Q(s, a')$



ROBOT INTELLIGENCE LAB