ROBOT INTELLIGENCE LAB

Introduction to Reinforcement Learning

Lecture 3. Model-free Methods

Sungjoon Choi, Korea University

kakaobrain

Recap on the Previous Lecture

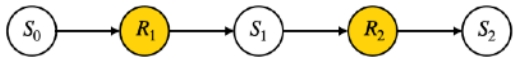
ROBOT INTELLIGENCE LAB

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)

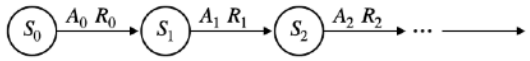
Markov Reward Process

- 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: $r(s) = \mathbb{E}[R_{t+1} \mid S_t = s]$

Markov Decision Process



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

Bellman Optimality Equation

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

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

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

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

Introduction

ROBOT INTELLIGENCE LAB

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

Introduction

ROBOT INTELLIGENCE LAB

Value Iteration

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

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

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

- 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:

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

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

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

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

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

Introduction

ROBOT INTELLIGENCE LAB

Value Iteration

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

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

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

- 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:

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

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

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

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

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

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

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

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

  - 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~~

ROBOT INTELLIGENCE LAB

Monte-Carlo Learning

Monte-Carlo Learning

ROBOT INTELLIGENCE LAB

- **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

ROBOT INTELLIGENCE LAB

- **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

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

  - 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

ROBOT INTELLIGENCE LAB

- 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):

$$\mu_{k+1} = \frac{1}{k+1}\sum_{j=1}^{k+1} x_j = \frac{1}{k+1}\left(x_{k+1} + \sum_{j=1}^{k} x_j\right)$$

$$= \frac{1}{k+1}(x_{k+1} + k\mu_k) = \mu_k + \frac{1}{k+1}(x_{k+1} - \mu_k)$$

Monte-Carlo Learning

ROBOT INTELLIGENCE LAB

- 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

ROBOT INTELLIGENCE LAB

- 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

ROBOT INTELLIGENCE LAB

```python
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 = []
```

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

ROBOT INTELLIGENCE LAB

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

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

- How about using the current value estimate $\widehat{V}(s_t)$?

- In other words, we approximate the value using the current value estimate:

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

Temporal Difference Learning

ROBOT INTELLIGENCE LAB

- 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

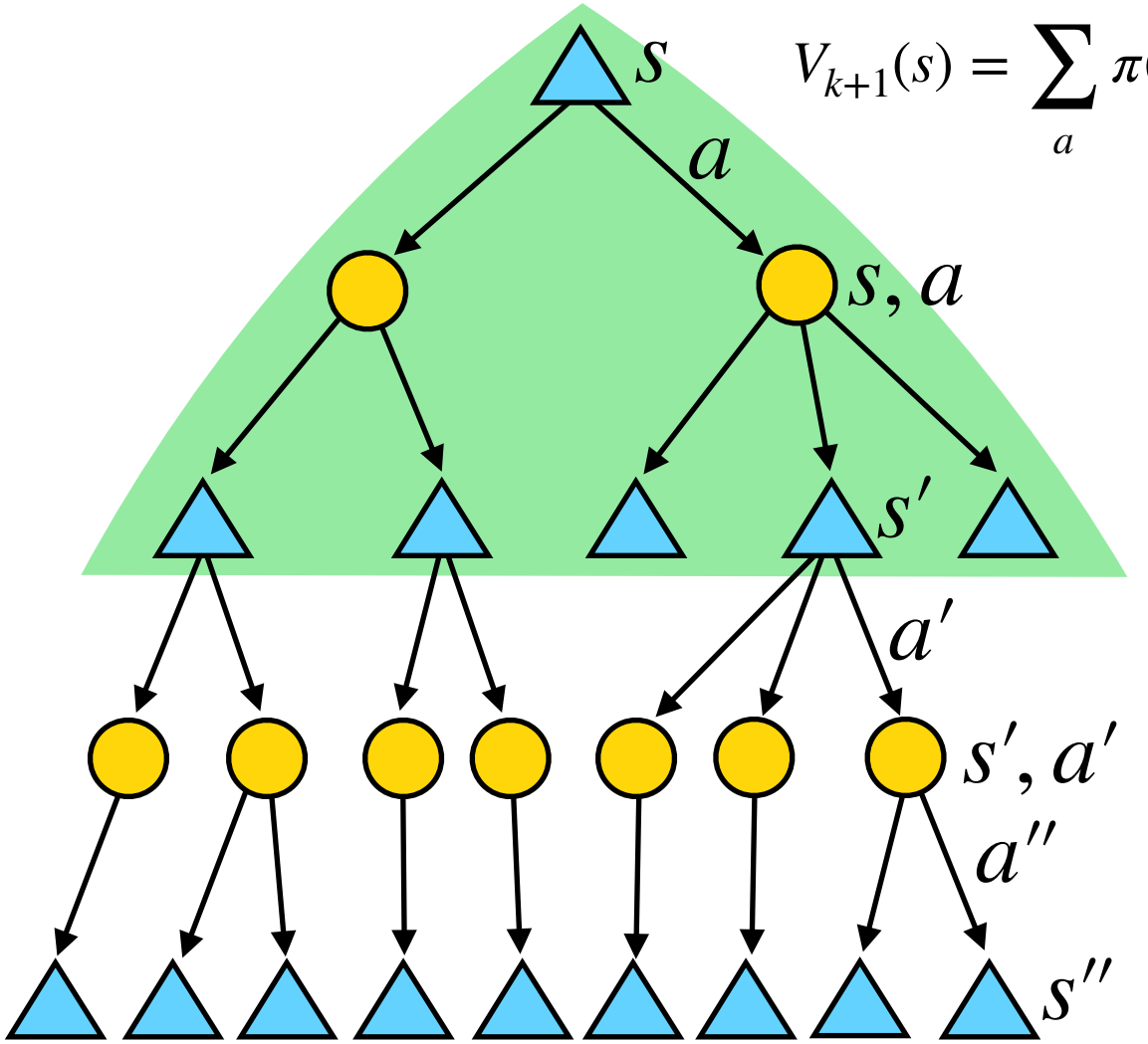
- Advantages of **TD learning**
  - TD learning can learn before knowing the final outcome.
  - TD learning can learn in an **online** manner (after every step (s, a, s')).
  - TD learning works well in infinite-horizon tasks.

- 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.

Dynamic Programming

Dynamic Programming

ROBOT INTELLIGENCE LAB

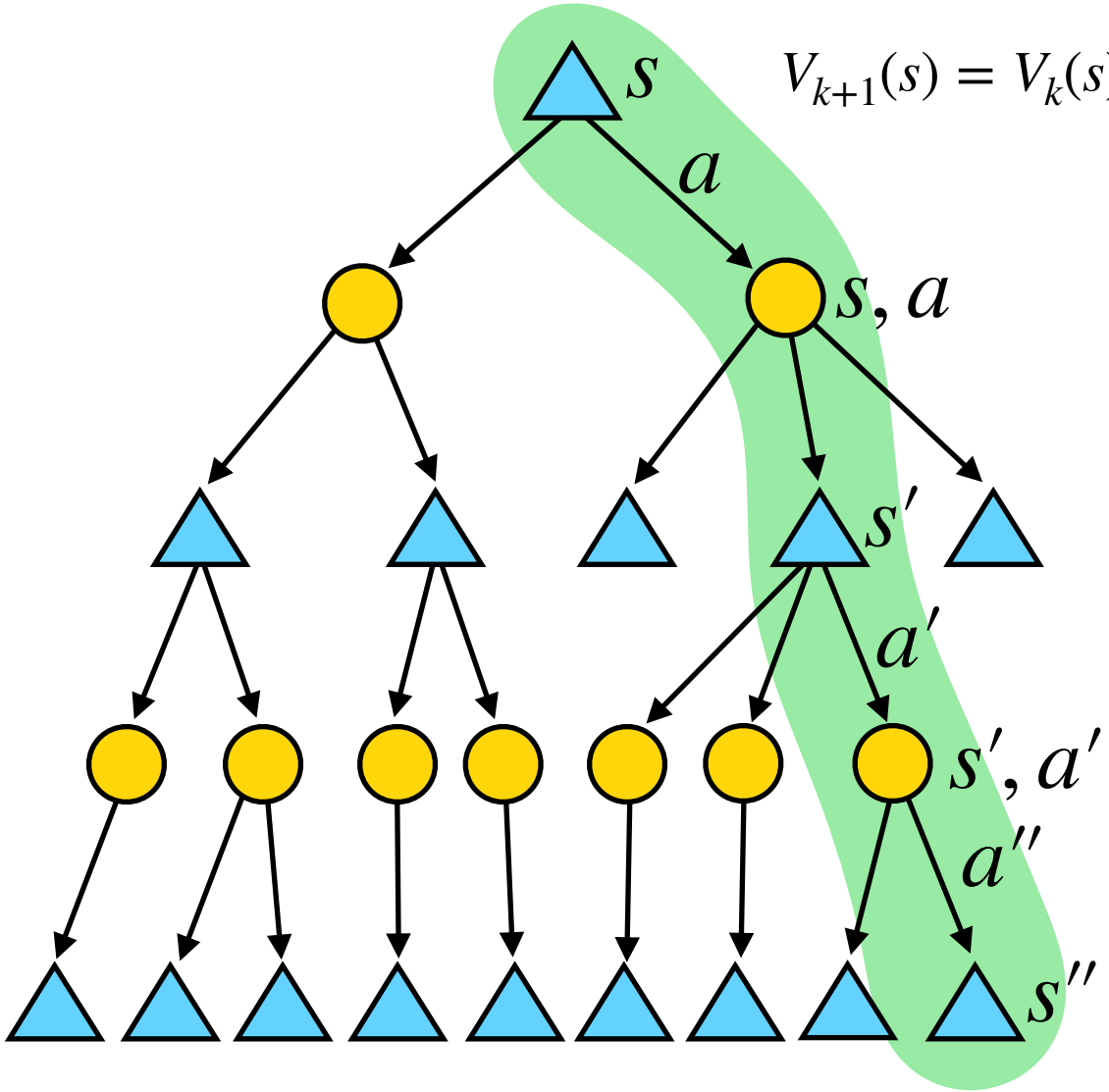


$$V_{k+1}(s) = \sum_{a} \pi(a \mid s) \sum_{s'} \left[r(s, a, s') + \gamma V_k(s') \right] P(s' \mid s, a)$$

Monte-Carlo Learning

ROBOT INTELLIGENCE LAB

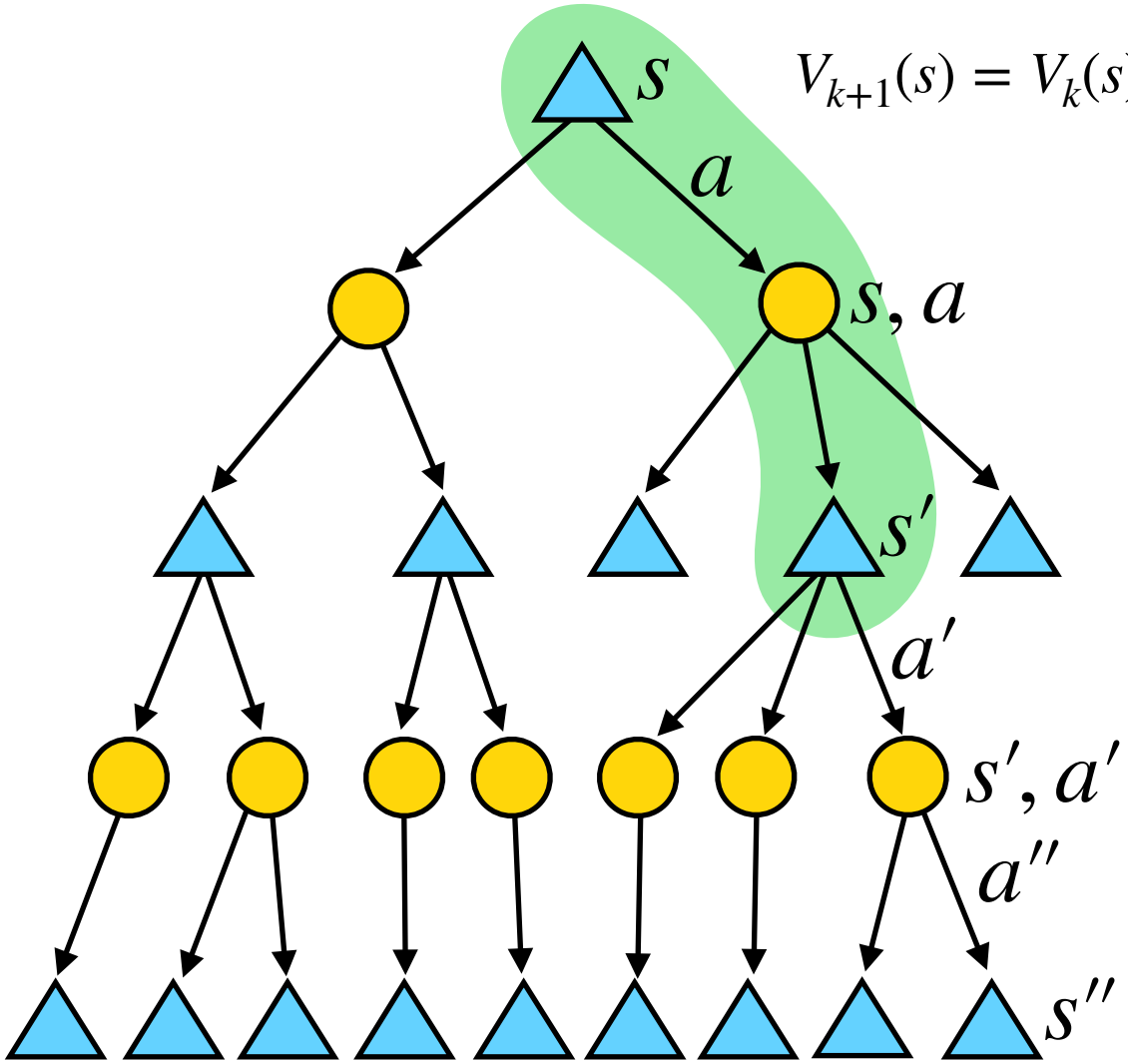
$$V_{k+1}(s) = V_k(s) + \alpha\left(G_t - V_k(s)\right)$$



Temporal Difference Learning

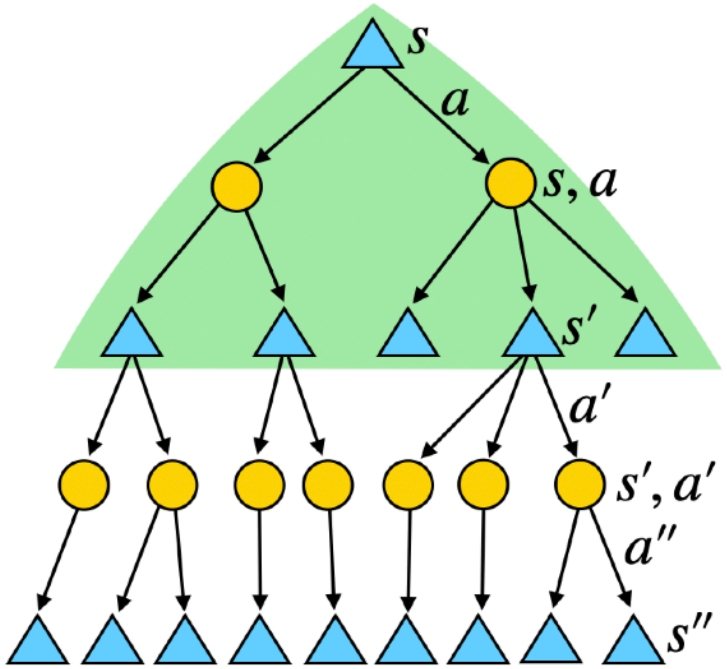
ROBOT INTELLIGENCE LAB

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



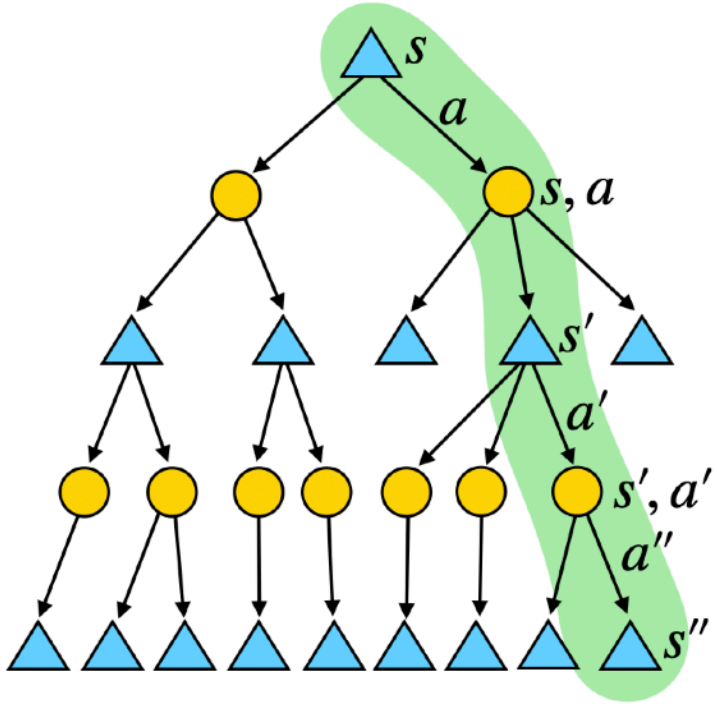
DP vs. MC vs. TD

ROBOT INTELLIGENCE LAB



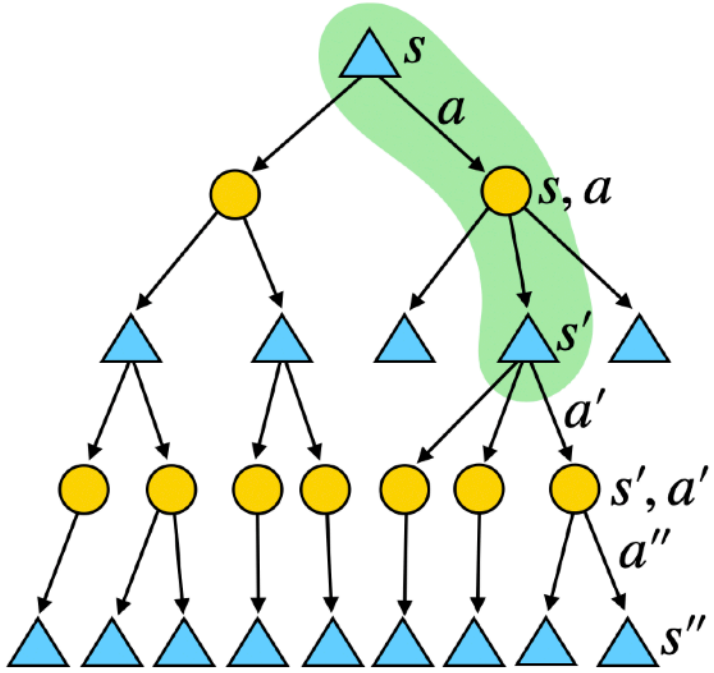
Dynamic Programming

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



Monte-Carlo Learning

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



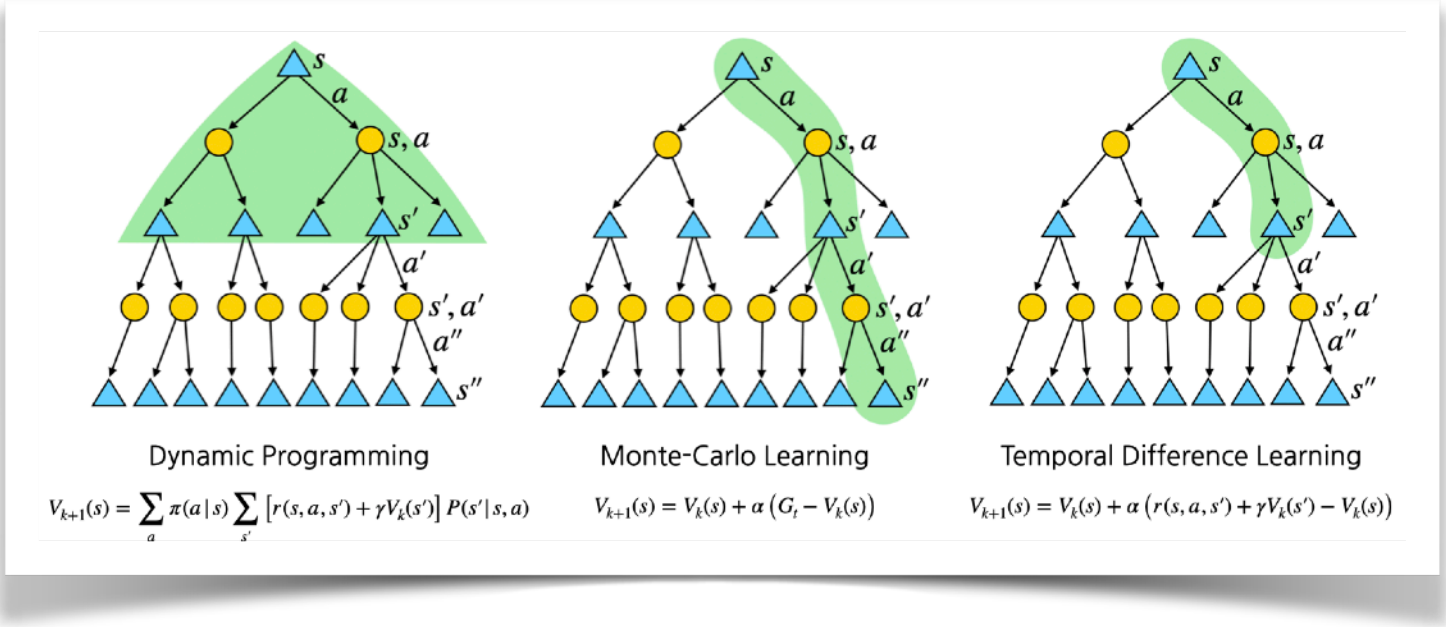
Temporal Difference Learning

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

ROBOT INTELLIGENCE LAB

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:

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

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

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

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

Policy Iteration (revisited)

ROBOT INTELLIGENCE LAB

- **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:

State Transition Probability

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

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

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

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

Policy Iteration (revisited)

ROBOT INTELLIGENCE LAB

- **Step 1: Policy evaluation**
  - Instead of finding $V(s)$, find $Q(s,a)$.
  - Start from a random initial Q_0.
  - Iterate until value converges:

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

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

ROBOT INTELLIGENCE LAB

- **Step 1: Policy evaluation**
  - Estimate $\widehat{Q}(s,a)$ from samples using **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

ROBOT INTELLIGENCE LAB

- **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

ROBOT INTELLIGENCE LAB

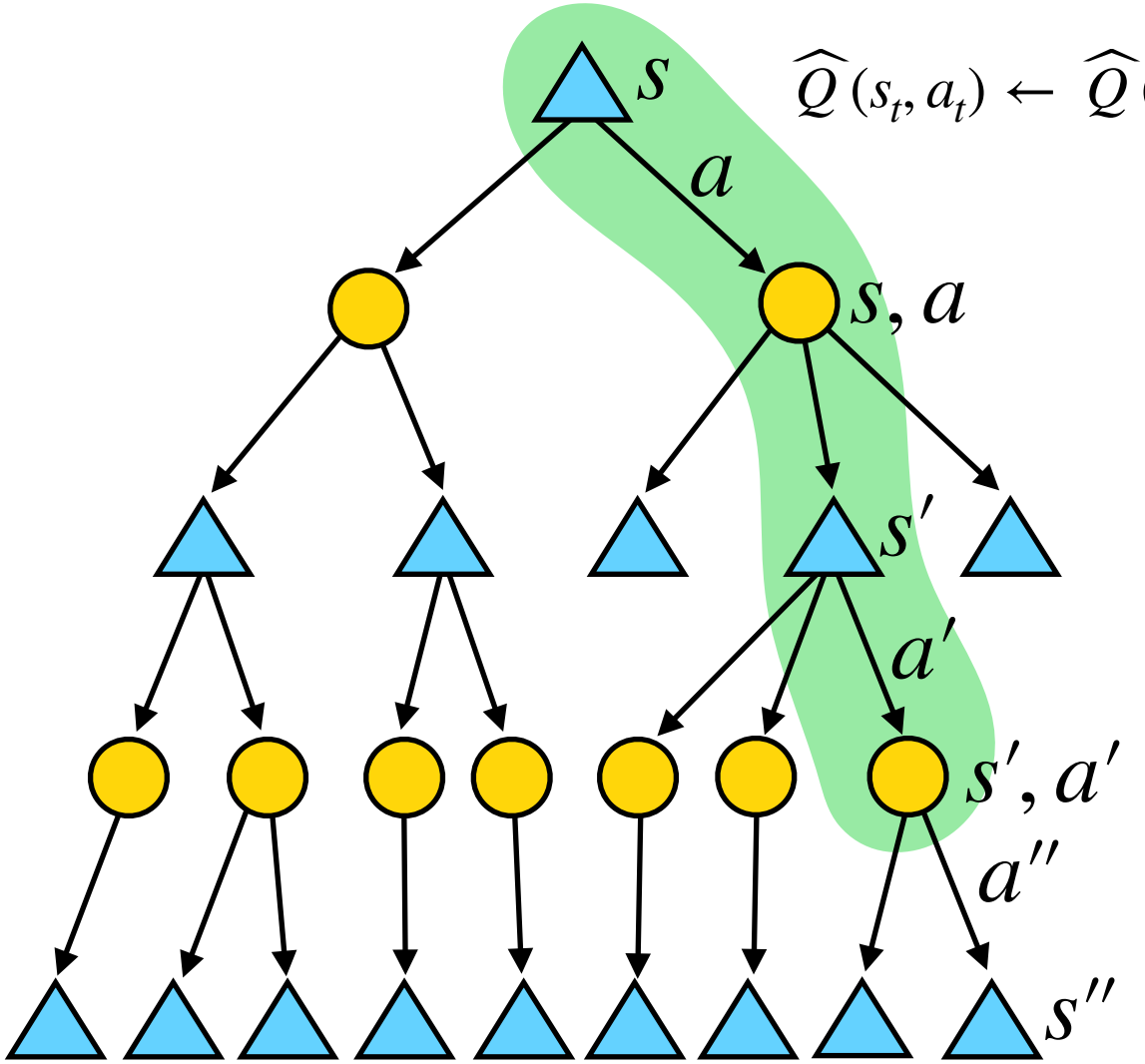
- Initialize $Q(s, a)$
- Repeat (for each episodes)
  - Sample an initial state s_0.
  - Sample a_0 from an ϵ-greedy policy π.
  - Repeat (for each time step t)
    - Get reward r_{t+1} and next state s_{t+1}.
    - Sample a_{t+1} from the ϵ-greedy policy π.
    - Update $\widehat{Q}(s, a)$ using $(s_t,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

SARSA

ROBOT INTELLIGENCE LAB

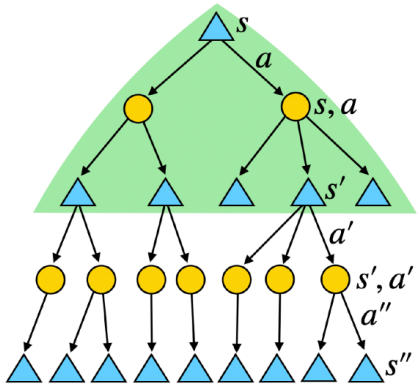
```
def update_Q(self,state,action,reward,state_prime,action_prime,done):
    """
    Update Q value using TD learning
    """
    Q_old = self.Q[state][action]
    if done:
        td_target = reward # for the last step, Q = reward
    else:
        td_target = reward + self.gamma*self.Q[state_prime][action_prime]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

SARSA



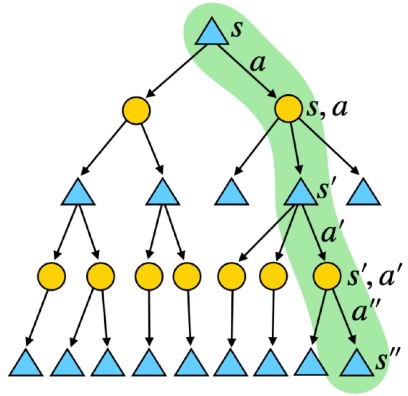
$$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t) \right)$$

DP vs. MC vs. TD vs. SARSA



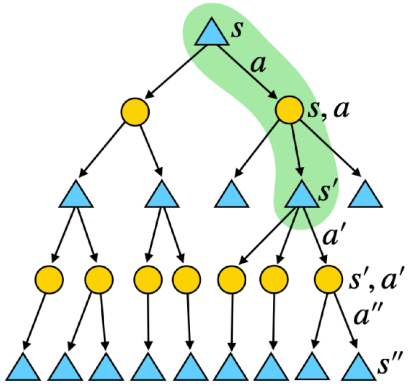
Dynamic Programming

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



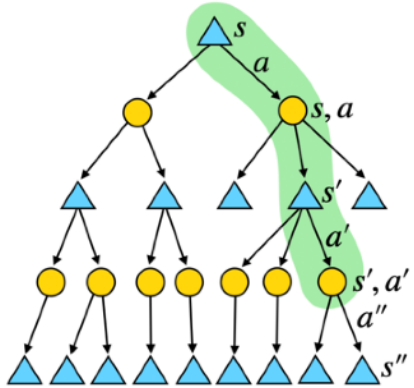
Monte-Carlo Learning

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



Temporal Difference Learning

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



SARSA

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

Q-Learning

Q-Learning

ROBOT INTELLIGENCE LAB

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

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

  - 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

ROBOT INTELLIGENCE LAB

- 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.

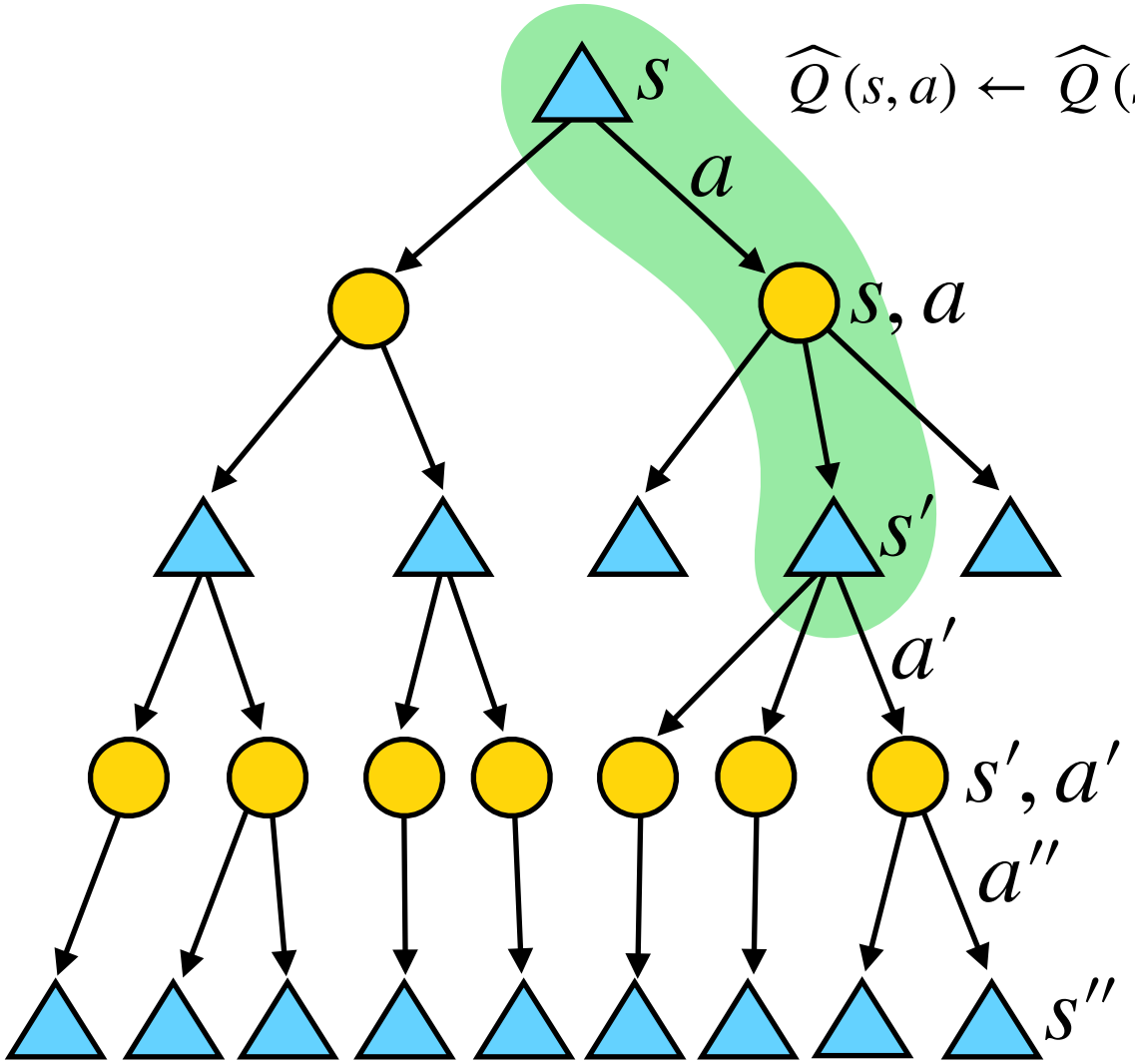
Q-Learning

ROBOT INTELLIGENCE LAB

```
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

ROBOT INTELLIGENCE LAB



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

ROBOT INTELLIGENCE LAB

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)

ROBOT INTELLIGENCE LAB

- Recall the original Q-Learning

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

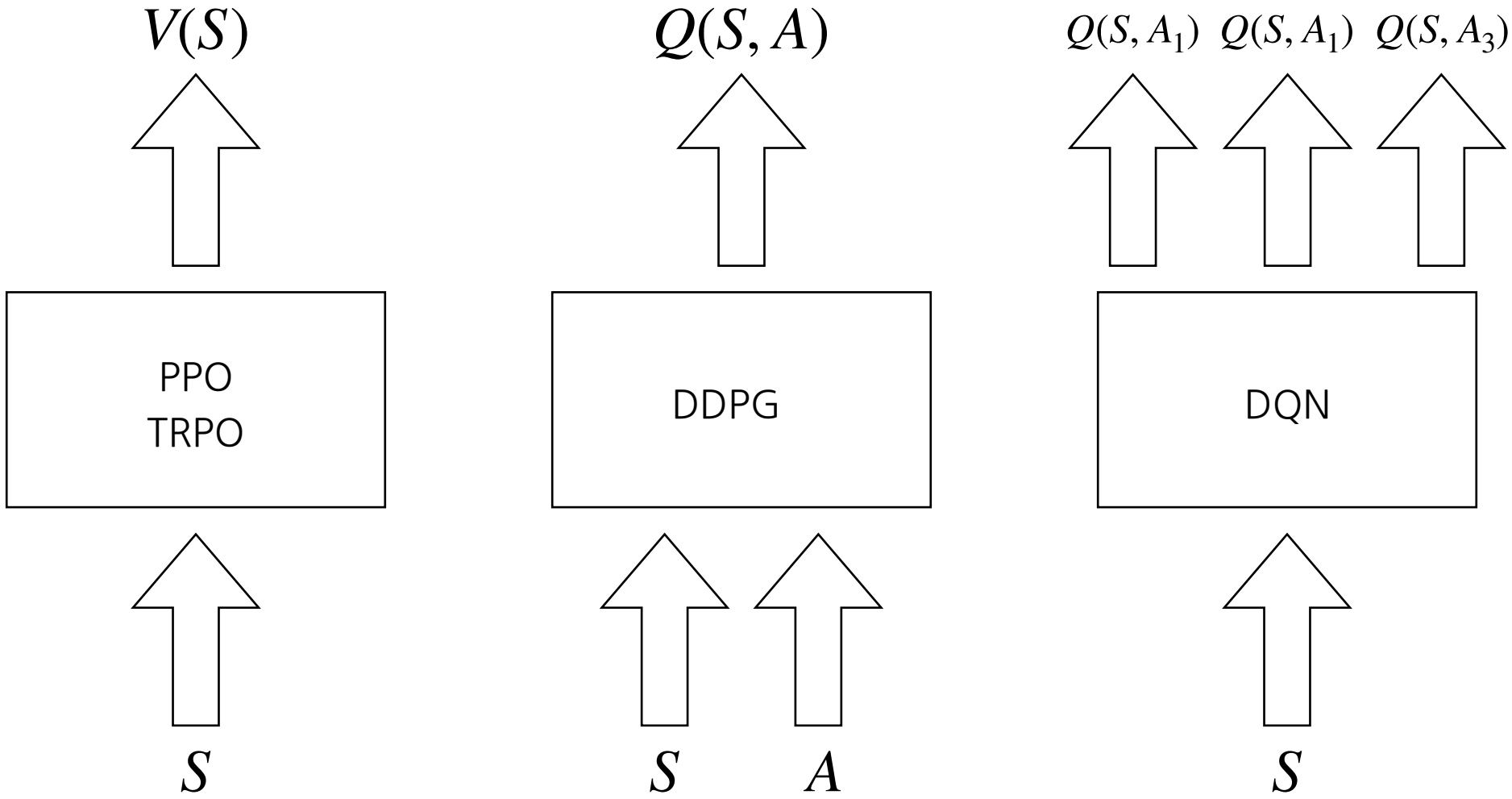
Target Prediction

- 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)

ROBOT INTELLIGENCE LAB

- (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)

ROBOT INTELLIGENCE LAB

- (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

ROBOT INTELLIGENCE LAB

- 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.

Thank You