Reinforcement Learning

Lecture 6. Summary

Sungjoon Choi, Korea University

Markov Process

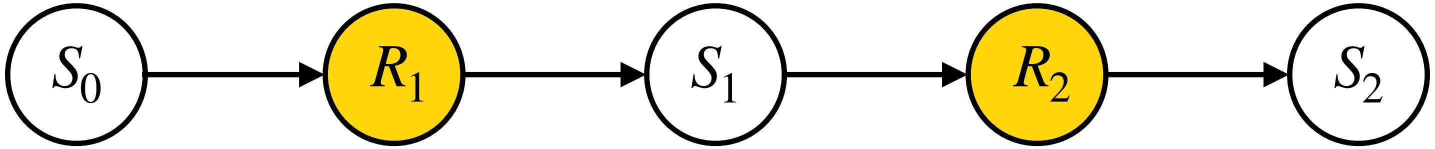
- A **Markov chain** is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event (aka Markov property).

- Random variable: S_t (state)

$$S_0 \longrightarrow S_{t-1} \longrightarrow S_t \longrightarrow S_{t+1}$$

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

$$S_0 \xrightarrow{A_0 \ R_0} S_1 \xrightarrow{A_1 \ R_1} S_2 \xrightarrow{A_2 \ R_2} \cdots \longrightarrow$$

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

Markov Decision Process

- Formally, an MDP is a tuple (S, A, P, R, d):
  - A set of states $s \in S$.
  - A set of actions $a \in A$
  - A state transition function (or matrix)
    - $P(s'|s, a) = P(S_{t=1} = s|S_t = s, A_t = a)$
    - $P_{sas'} = P(s'|s, a)$
  - A reward function
    - $r(s) = \mathbb{E}[R_{t+1}|S_t = s]$
    - It depends on both state and action.
  - An initial state distribution d

Return

- **Return** G_t is the (discounted) sum of future rewards, and hence, also a random variable.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$$

- Discount factor γ:

(State) Value Function

- The **state-value function** $V(s)$ is a function of a state and is the expected return starting from the state s.

$$
\begin{aligned}
V(s) &= \mathbb{E}[G_t \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+1} \cdots \mid S_t = s]
\end{aligned}
$$

(State-Action) Value Function

- The **state-action-value function** $Q(s, a)$ is a function of a state and is the expected return starting from the state s.

$$
\begin{aligned}
Q(s, a) &= \mathbb{E}[G_t \mid S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+1} \cdots \mid S_t = s, A_t = a]
\end{aligned}
$$

Policy

- A **policy** is a **distribution over actions** given state.

$$\pi(a|s) = P(A_t = a | S_t = s)$$

  - In an MDP, a policy is a function of the current state s.
  - A policy is stationary (time-independent).
  - A deterministic policy can also be represented by a distribution.

Optimality

ROBOT INTELLIGENCE LAB

What does it mean by **solving** an MDP?

Optimal Value and Policy

- Optimal state value

$$V^*(s) = \max_{\pi} V_{\pi}(s)$$

- Optimal state-action value

$$Q^*(s, a) = \max_{\pi} Q_{\pi}(s, a)$$

- Optimal policy

$$\pi^*(a \mid s) = 1 \text{ for } a = \arg\max_{a'} Q^*(s, a')$$

Value Iteration

ROBOT INTELLIGENCE LAB

Given an MDP (S, A, P, R, d), how can we **solve** an MDP?

Value Iteration

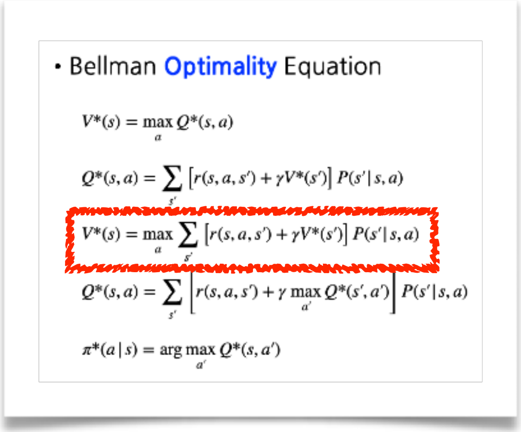
- Value iteration utilizes the **principle of optimality**.

$$V^*(s) = \max_{\pi} V_{\pi}(S)$$

- Then, the solution $V^*(s)$ can be found by one-step lookahead

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

- The main idea is to apply these updates iteratively, repeat until convergence.
  - It is guaranteed to converge to the unique optimal value.
- However, there is no explicit policy.


- Bellman **Optimality** Equation

$$V^*(s) = \max_{a} Q^*(s, a)$$

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

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

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

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

Q-Value Iteration

- However, it is not straightforward to come up with an **optimal policy** solely from $V^*(s)$.
- Hence, we use following relations between $V^*(s)$ and $Q^*(s,a)$ (aka the Bellman optimality equation).

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

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

- Bellman Equation

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a)$$

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

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

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

- Bellman Optimality Equation

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

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

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

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

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

Q-Value Iteration

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

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

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

- We now have an explicit form of the policy:

$$\pi(a \mid s) = 1 \text{ for } a = \arg\max_{a'} Q(s,a')$$

  - Note that this policy is **deterministic**.

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:

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

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

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

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

Limitation of Model-based Methods

Value Iteration

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

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

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

- We now have an explicit form of the policy:

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

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:

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

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

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

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

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

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

Model-free Methods

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

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

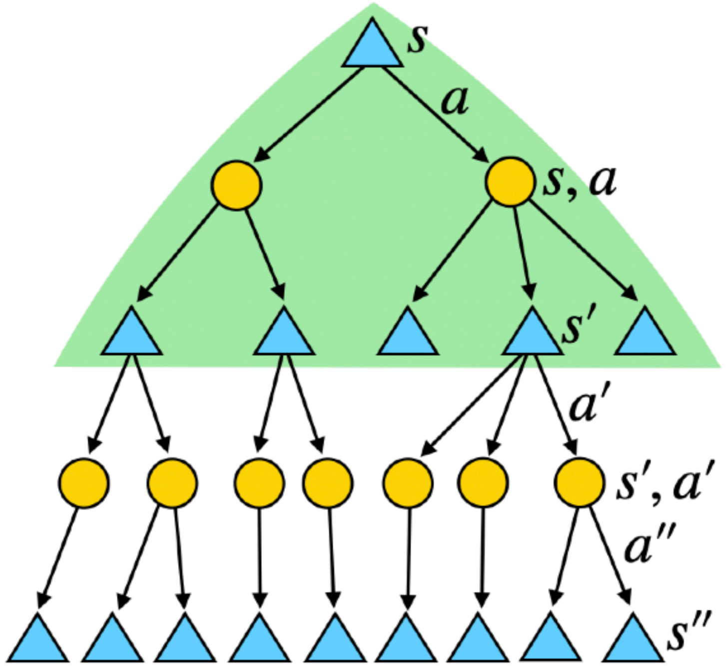
- The goal of a model-free method can be written as

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

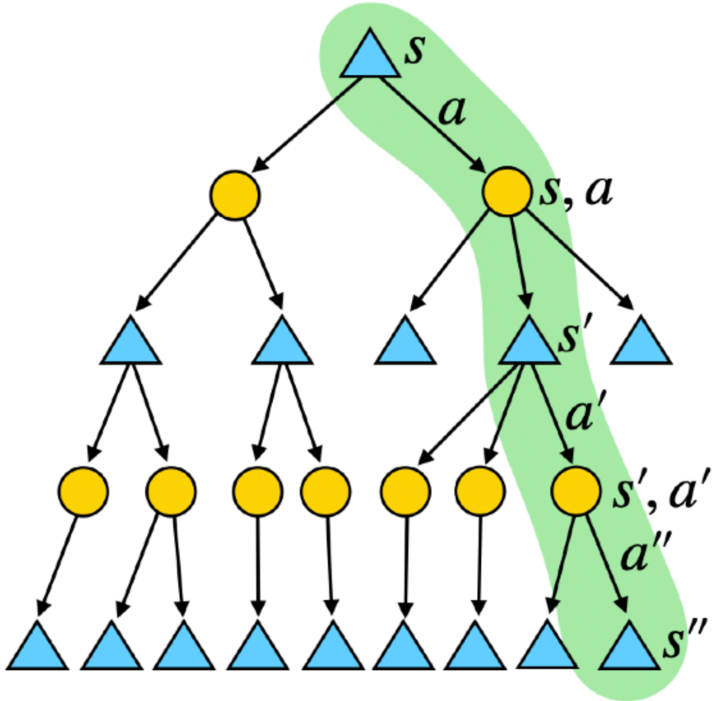
Model-free Methods

ROBOT INTELLIGENCE LAB



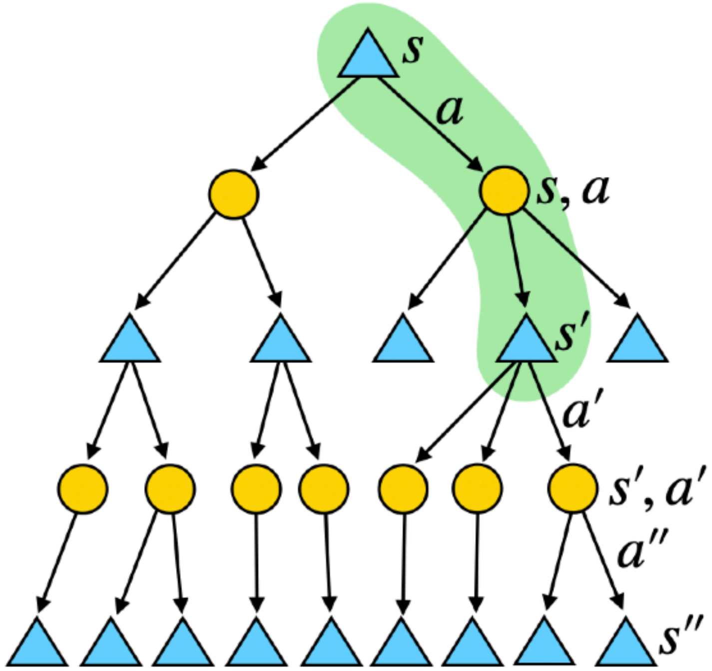
Dynamic Programming

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



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

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

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

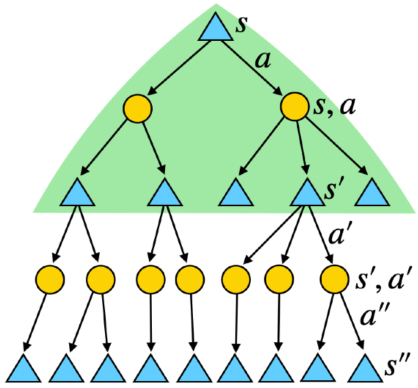
- Step 2: Policy improvement
  - For the sake of better exploration, we use an ϵ-greedy policy.
    - With probability $1 - \epsilon$, choose the greedy action $a = \arg\max_{a'} Q_{\pi_i}(s, a')$.
    - With probability ϵ, choose a random action.

SARSA

ROBOT INTELLIGENCE LAB

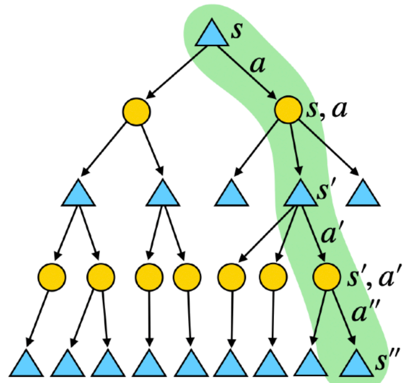
- Initialize $Q(s, a)$
- Repeat (for each episodes)
  - Sample an initial state s_0.
  - Sample a_0 from an ϵ-greedy policy π.
  - Repeat (for each time step t)
    - Get reward r_{t+1} and next state s_{t+1}.
    - Sample a_{t+1} from the ϵ-greedy policy π.
    - Update $\widehat{Q}(s, a)$ using $(s_t,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

DP vs. MC vs. TD vs. SARSA



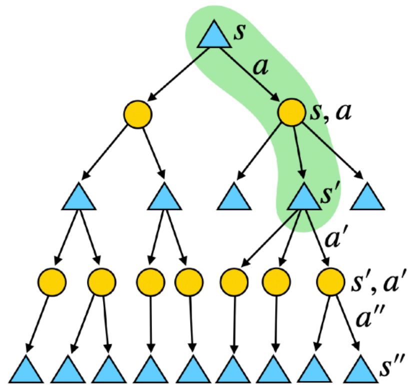
Dynamic Programming

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



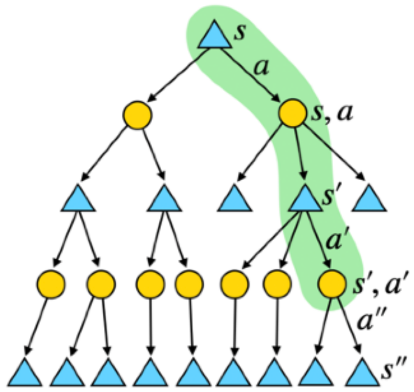
Monte-Carlo Learning

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



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

- 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

- It can be regarded as a model-free value iteration.
- Q-Learning
  - For each time step

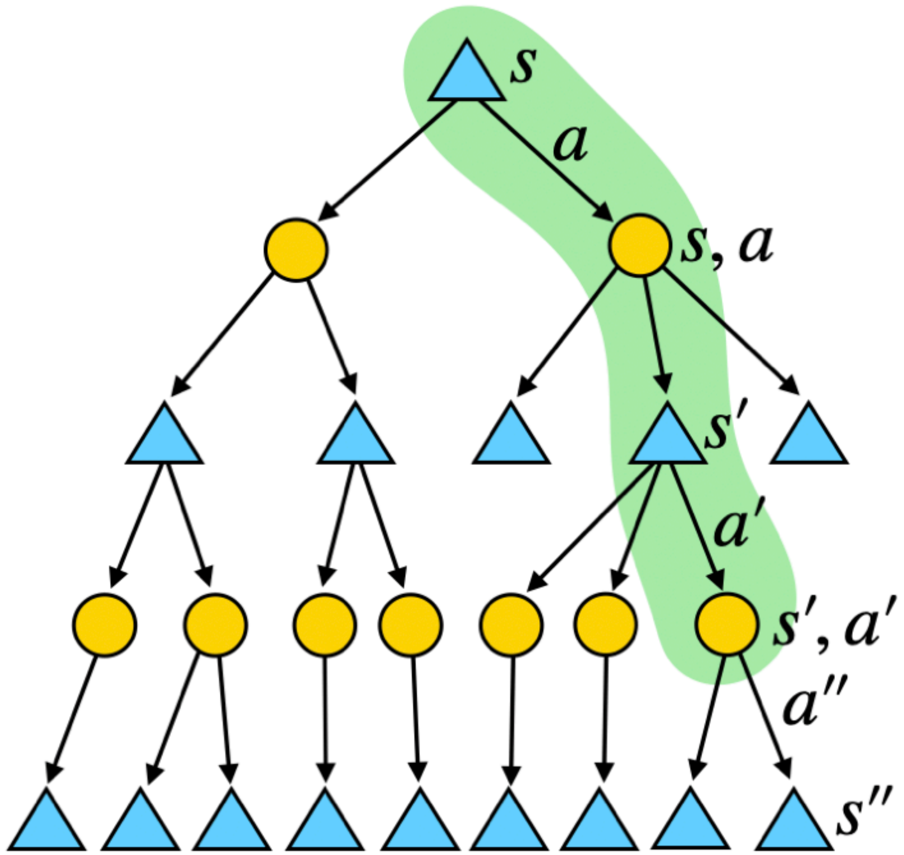
$$\widehat{Q}(s,a) \leftarrow \widehat{Q}(s,a) + \alpha\left(r(s,a,s') + \gamma \max_{a'} \widehat{Q}(s',a') - \widehat{Q}(s,a)\right)$$

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.

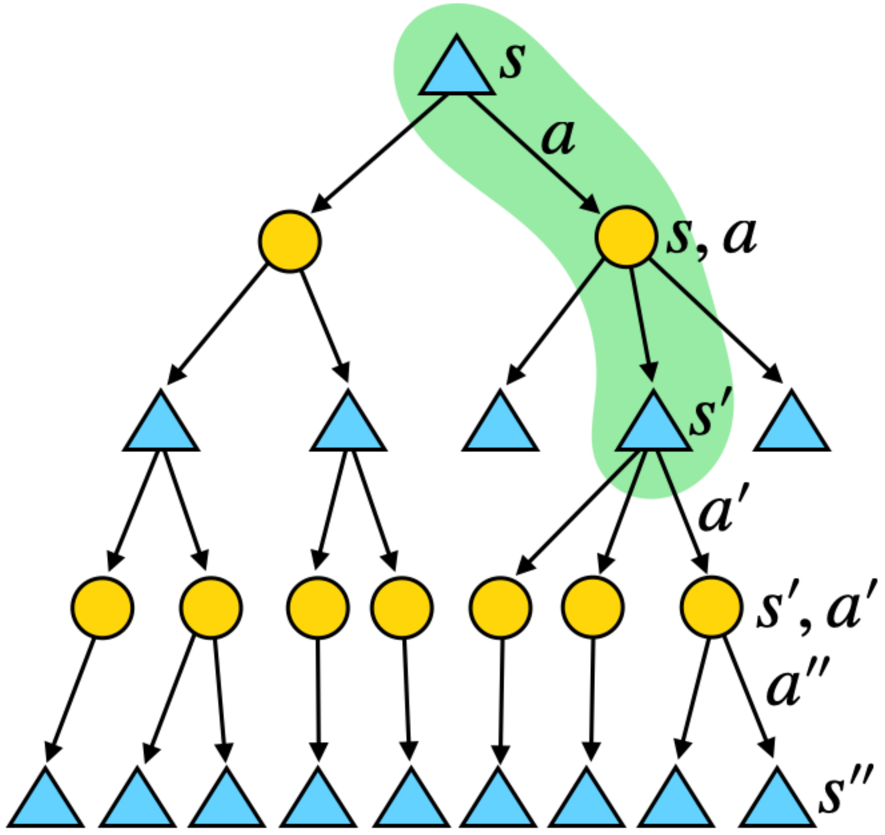
Q-Learning

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



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

Deep Q Network (DQN)

ROBOT INTELLIGENCE LAB

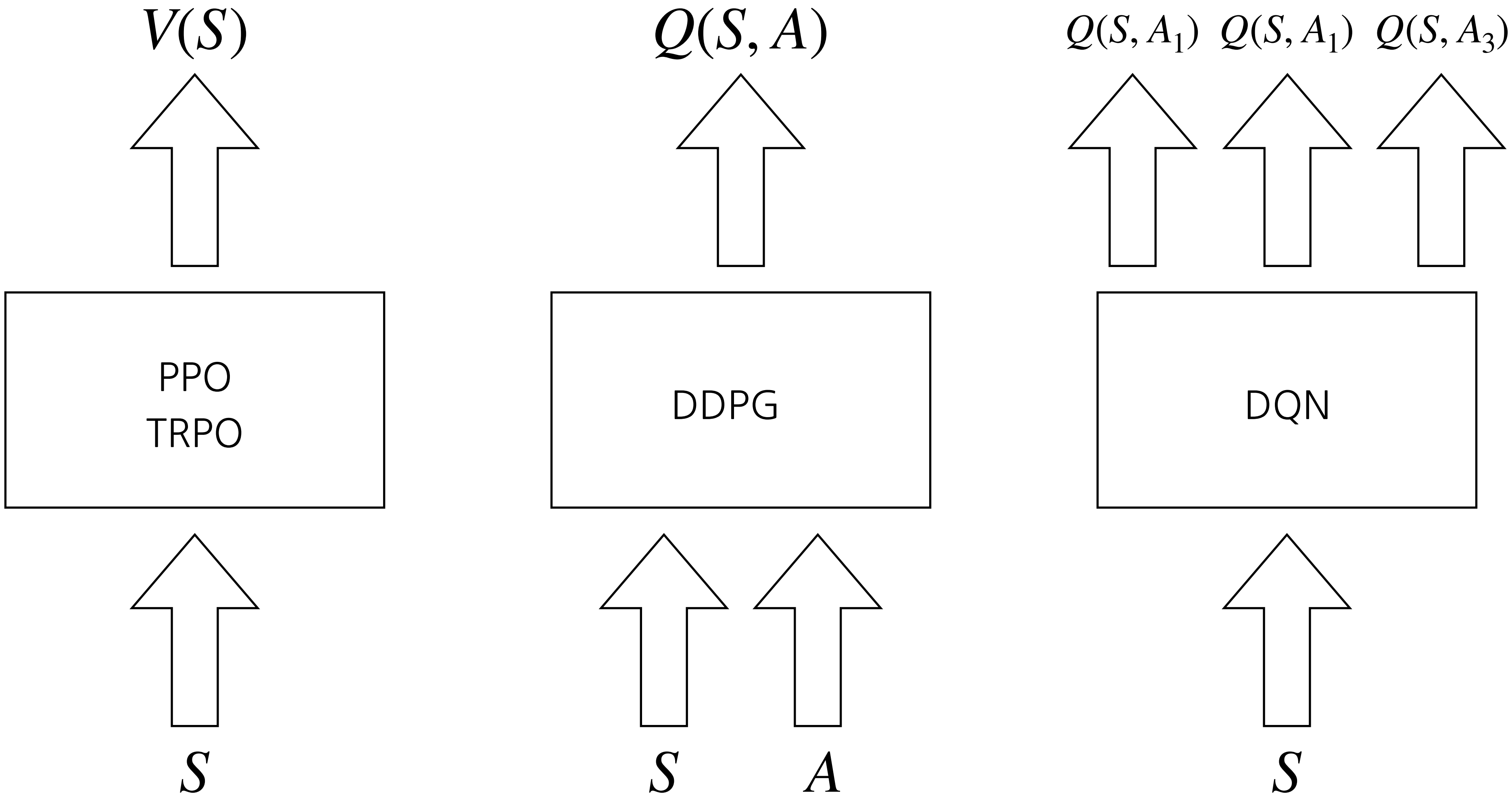
- Recall the original Q-Learning

$$Q_{k+1}(s,a) \leftarrow Q_k(s,a) + \alpha \left(\underbrace{r(s,a,s') + \gamma \max_{a'} Q_k(s',a')}_{\text{Target}} - \underbrace{Q_k(s,a)}_{\text{Prediction}} \right)$$

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

$$L(\theta) = \sum_i \left(\underbrace{r_i + \gamma \max_{a'} Q(s_i', a'; \theta)}_{\text{Target}} - \underbrace{Q(s_i, a_i; \theta)}_{\text{Prediction}} \right)^2$$

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

```python
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
```

Policy Gradients

$$\nabla_\theta \eta(\pi_\theta) = \nabla_\theta \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \,|\, \pi_\theta\right]$$

- Policy Gradient Theorem:

$$\nabla_\theta \eta(\pi_\theta) = \frac{1}{(1-\gamma)} \sum_s \rho_{\pi_\theta} \sum_a \nabla_\theta \pi_\theta(a\,|\,s) Q^{\pi_\theta}(s,a)$$

$$\nabla_\theta \eta(\pi_\theta) \approx \nabla_\theta \log \pi_\theta(a_t\,|\,s_t) Q_{\pi_\theta}(s_t, a_t)$$

- Note that we only require the gradient of $\pi_\theta(\cdot)$ not $Q^{\pi_\theta}(\cdot)$!

R. Sutton et al, "Policy Gradient Methods for Reinforcement Learning with Function Approximation", NeurIPS, 2000

Trust Region Policy Optimization

$$\max_{\theta_{i+1}} L_{\pi_{\theta_i}}(\pi_{\theta_{i+1}}) = \mathbb{E}_{s\sim\rho_{\pi_{\theta_i}}, a\sim\pi_{\theta_i}}\left[\frac{\pi_{\theta_{i+1}}(a\,|\,s)}{\pi_{\theta_i}(a\,|\,s)} A_{\pi_{\theta_i}}(s,a)\right]$$

$$\text{subject to } D_{KL}^{\rho}(\pi_\theta, \pi_{\theta_{i+1}}) \le \delta$$

- In summary,
  - TRPO is a minorization maximization framework for RL.
  - Interpretation of the trust region method:
    1. Update policy distribution slowly
    2. Consider the geometry of the distribution space
  - There are two approximations: 1) $\mathbb{E}_{s\sim\rho_{\pi'}} \Rightarrow \mathbb{E}_{s\sim\rho_\pi}$ and 2) $D_{KL}^{\max} \Rightarrow D_{KL}^{\rho}$

Proximal Policy Optimization (Adaptive KL Penalty)

- The TRPO objective is:

$$\max_{\theta} \mathbb{E}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \text{ s.t. } D_{KL}^{\rho}\left[\pi_{old}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right] \le \delta$$

- The unconstrained objective of TRPO is:

$$L(\theta) = \max_{\theta} \mathbb{E}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta D_{KL}^{\rho}\left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right]\right]$$

- The proposed adaptive KL penalty method is to adaptively change β by checking $d = \mathbb{E}_t\left[D_{KL}[\pi_{\theta_{old}}, \pi_\theta]\right]$:
    - If $d < d_{targ}/1.5$, $\beta \leftarrow \beta/2$
    - If $d > d_{targ} \times 1.5$, $\beta \leftarrow \beta \times 2$

Soft Actor-Critic

ROBOT INTELLIGENCE LAB

- SAC learns three functions: $V_\psi(s)$, $Q_\theta(s,a)$, and $\pi_\phi(a \mid s)$.
- For learning $V_\psi(s)$:

$$J_V(\psi) = \mathbb{E}_{s_t \sim \mathcal{D}}\left[\frac{1}{2}\left(V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi_\phi}\left[Q_\theta(s_t, a_t) - \log \pi_\phi(a_t \mid s_t)\right]\right)^2\right]$$

where actions are being sampled from the current policy $\pi_\phi(a \mid s)$ not from the replay.

- For learning $Q_\theta(s,a)$:

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim \mathcal{D}}\left[\frac{1}{2}\left(Q_\theta(s_t,a_t) - \hat{Q}(s_t,a_t)\right)^2\right] \text{ where } \hat{Q}(s_t,a_t) = r(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1}}\left[V_\psi(s_{t+1})\right]$$

- For learning $\pi_\phi(a \mid s)$:

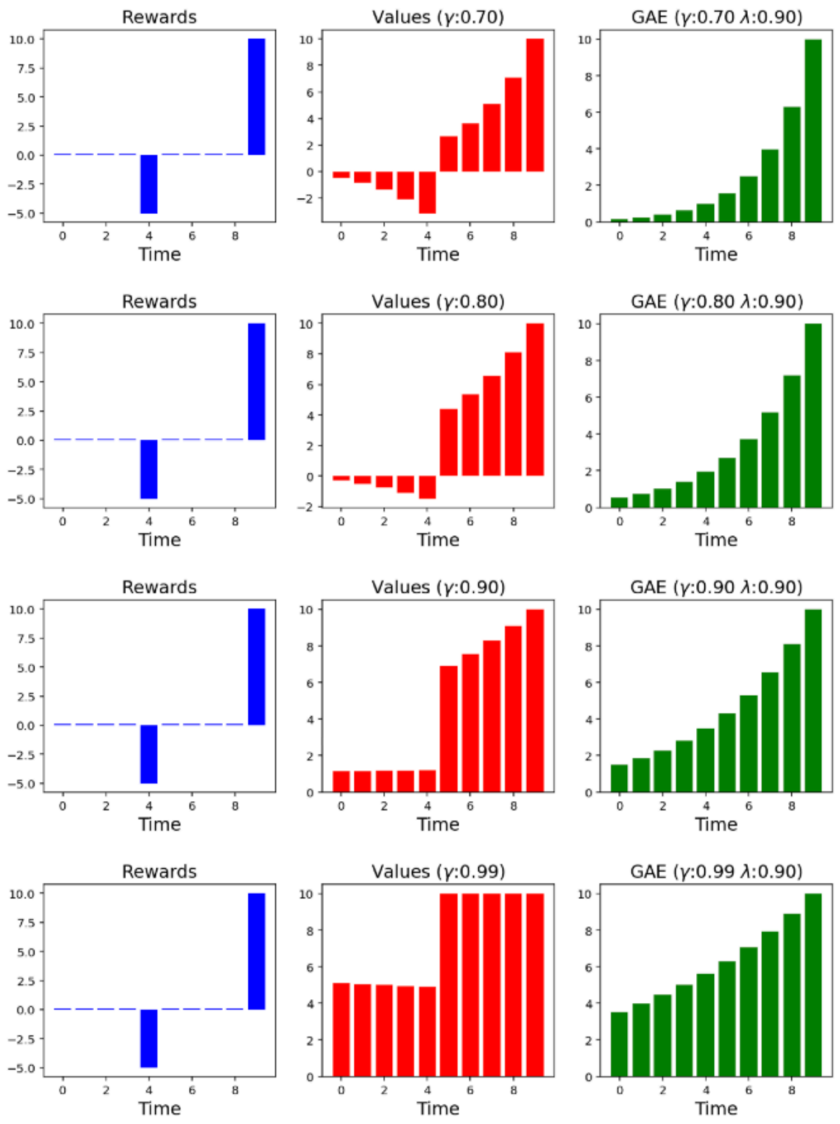
$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}}\left[D_{KL}\left(\pi_\phi(\cdot \mid s_t) \middle\| \frac{\exp\left(Q_\theta(s_t,\cdot)\right)}{Z_\theta(s_t)}\right)\right]$$

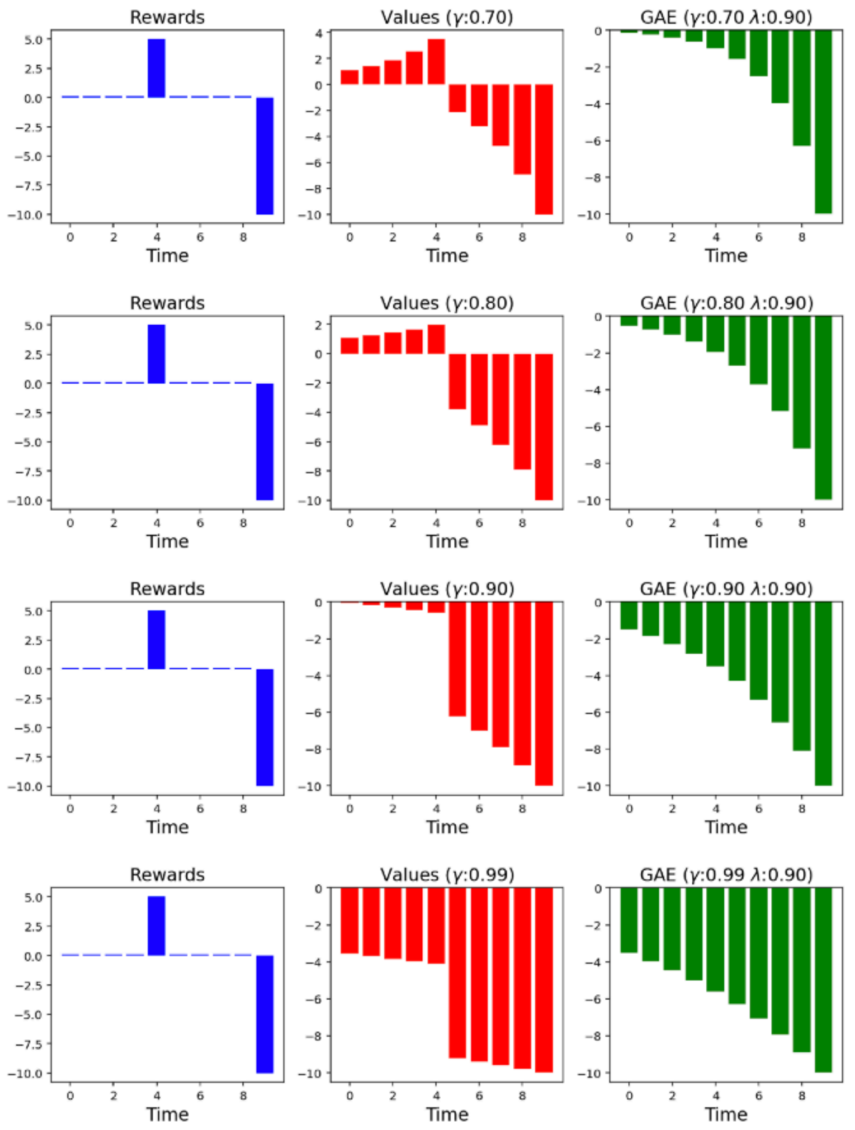
If we reparameterize the stochastic policy $a_t = f_\phi(\epsilon_t; s_t)$ where ϵ_t is sampled from some distribution,

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D, \epsilon_t \sim \mathcal{N}}\left[\log \pi_\phi\left(f_\phi(\epsilon_t; s_t) \mid s_t\right) - Q_\theta\left(s_t, f_\phi(\epsilon_t; s_t)\right)\right]$$

Generalized Advantage Estimation

ROBOT INTELLIGENCE LAB





Augmented Random Search

ROBOT INTELLIGENCE LAB

Algorithm 2 Augmented Random Search (ARS): four versions **V1**, **V1-t**, **V2** and **V2-t**

Hyperparameters: step-size α, number of directions sampled per iteration N, standard deviation of the exploration noise ν, number of top-performing directions to use b ($b < N$ is allowed only for **V1-t** and **V2-t**)

Use top b search directions.

Initialize: $M_0 = \mathbf{0} \in \mathbb{R}^{p \times n}$, $\mu_0 = \mathbf{0} \in \mathbb{R}^n$, and $\Sigma_0 = \mathbf{I}_n \in \mathbb{R}^{n \times n}$, $j = 0$.

while ending condition not satisfied **do**

Sample $\delta_1, \delta_2, \ldots, \delta_N$ in $\mathbb{R}^{p \times n}$ with i.i.d. standard normal entries.

Collect $2N$ rollouts of horizon H and their corresponding rewards using the $2N$ policies

$$\textbf{V1:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu \delta_k) x \\ \pi_{j,k,-}(x) = (M_j - \nu \delta_k) x \end{cases}$$

$$\textbf{V2:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu \delta_k) \boxed{\operatorname{diag}(\Sigma_j)^{-1/2} (x - \mu_j)} \\ \pi_{j,k,-}(x) = (M_j - \nu \delta_k) \boxed{\operatorname{diag}(\Sigma_j)^{-1/2} (x - \mu_j)} \end{cases}$$

Input normalization

for $k \in \{1, 2, \ldots, N\}$.

Sort the directions δ_k by $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$ denote by $\delta_{(k)}$ the k-th largest direction, and by $\pi_{j,(k),+}$ and $\pi_{j,(k),-}$ the corresponding policies.

Make the update step:

$$M_{j+1} = M_j + \frac{\alpha}{b \sigma_R} \sum_{k=1}^{b} \left[r(\pi_{j,(k),+}) - r(\pi_{j,(k),-}) \right] \delta_{(k)},$$

where σ_R is the standard deviation of the $2b$ rewards used in the update step.

V2 : Set μ_{j+1}, Σ_{j+1} to be the mean and covariance of the $2NH(j+1)$ states encountered from the start of training.[2]

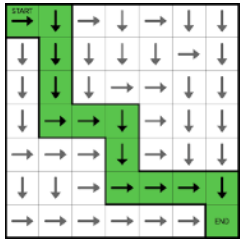
$j \leftarrow j + 1$

end while

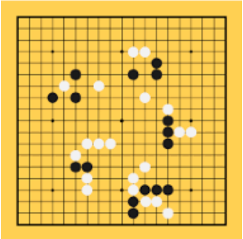
.

Rule of Thumb

- Discrete (and small) state space & discrete action space
  - E.g., grid world
  - In this case, the state-transition model can easily be defined.
  - Use **value iteration** or **policy iteration**.



- Discrete (and large) state space & discrete action space
  - E.g., Go
  - In this case, the state-transition model is cumbersome to be defined.
  - Use **Q-learning**.



- Continuous state space & discrete action space
  - E.g., Atari games
  - In this case, the state-transition model is impossible to be defined.
  - Use **DQN**.

- Continuous state space & continuous action space
  - E.g., Robotics
  - In this case, the state-transition model is impossible to be defined.
  - Use **PPO** or **SAC**.

ROBOT INTELLIGENCE LAB