

L13: Basic Deep RL Algorithms

Hao Su

Spring, 2021

*Contents are based on **Reinforcement Learning: An Introduction** from Prof. Richard S. Sutton and Prof. Andrew G. Barto, and **COMPM050/COMPGI13** taught at UCL by Prof. David Silver.*

Agenda

- Optimal Policy and Optimal Value Function
- Estimating Value Function for a Given Policy
- Q-Learning for Tabular RL
- Deep Q-Learning
- REINFORCE

click to jump to the section.

Optimal Policy and Optimal Value Function

Optimal Value Function

- Due to the Markovian property, the return starting from a state s is independent of its history. Therefore, we can compare the return of all policies starting from s and find the optimal one.
- The optimal state-value function $V^*(s)$ is the maximum value function over all policies
 - $V^*(s) = \max_{\pi} V_{\pi}(s)$
- The optimal action-value function $Q_*(s, a)$ is the maximum action-value function over all policies
 - $Q^*(s, a) = \max_{\pi} Q_{\pi}(s, a)$
- The optimal value function specifies the best possible performance in the MDP.

Optimal Policy

- Define a partial ordering over policies

$$\pi \geq \pi' \text{ if } V_\pi(s) \geq V_{\pi'}(s), \forall s$$

Theorem: For any Markov Decision Process

- *There exists an optimal policy π_* that is better than, or equal to, all other policies, $\pi_* \geq \pi, \forall \pi$*
- *All optimal policies achieve the optimal value function, $V_{\pi^*}(s) = V^*(s)$*
- *All optimal policies achieve the optimal action-value function, $Q_{\pi^*}(s, a) = Q^*(s, a)$*
- An optimal policy can be found by maximizing over $Q^*(s, a)$,

$$\pi^*(a|s) = \begin{cases} 1, & \text{if } a = \operatorname{argmax}_{a \in \mathcal{A}} Q^*(s, a) \\ 0, & \text{otherwise} \end{cases}$$

Bellman Optimality Equation

- Optimal value functions also satisfy recursive relationships

$$\begin{aligned}V^*(s) &= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\&= \max_a \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a]\end{aligned}$$

- Similarly, for action-value function, we have

$$Q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a]$$

- They are called **Bellman Optimality Equations**

Solving the Bellman Optimality Equation

- Bellman Optimality Equation is non-linear (because there is the max operation).
- No closed form solution (in general)
- Many iterative solution methods:
 - Value Iteration
 - Policy Iteration
 - Q-learning (we will talk about this later)
 - SARSA

Estimating Value Function for a Given Policy

Goal: Given a policy $\pi(a|s)$, estimate the value of the policy.

Monte-Carlo Policy Evaluation

- Basic idea: MC uses the simplest possible idea: **value = mean return**

Monte-Carlo Policy Evaluation

- Basic idea: MC uses the simplest possible idea: **value = mean return**
- Learn V_π from K episodes under policy π
 - $\{S_{k,0}, A_{k,0}, R_{k,1}, \dots, S_{k,T}\}_{k=1}^K \sim \pi$
- Recall that the *return* is the total discounted reward:
 - $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T$
- Recall that the value function is the expected return:
 - $V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$

Monte-Carlo Policy Evaluation

- Suppose that we have collected a number of trajectories. Monte-Carlo policy evaluation uses *empirical mean return* instead of *expected return*
 - For each episode:
 - For each time step t :
 - Compute empirical return G_t from the current state s
 - Increment total return $S(s) \leftarrow S(s) + G_t$
 - Increment state visit counter $N(s) \leftarrow N(s) + 1$
 - Value is estimated by mean return $V(s) = S(s)/N(s)$

Monte-Carlo Methods

- Quick facts:
 - MC is unbiased (average of the empirical return is the true return)
 - MC methods learn directly from episodes of experience
 - MC is model-free: no knowledge of MDP transitions / rewards
- Caveat: can only apply MC to episodic MDPs
 - All episodes must terminate

Temporal-Difference Learning

- Basic idea: TD leverages **Bellman expectation equation** to update the value function.

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s]$$

Temporal-Difference Learning

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s]$$

- Learn V_π from K episodes under policy π
 - $\{S_{k,0}, A_{k,0}, R_{k,1}, \dots, S_{k,T}\}_{k=1}^K \sim \pi$
- Simplest temporal-difference learning algorithm: TD(0)
 - Loop for new iterations:
 - Sample $(S_t, A_t, R_{t+1}, S_{t+1})$ with replacement from the transitions in the episodes
 - Update value $V(S_t)$ toward estimated return $R_{t+1} + \gamma V(S_{t+1})$:
 - $V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
 - $R_{t+1} + \gamma V(S_{t+1})$ is called the *TD target*
 - $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD error*

Temporal-Difference Learning

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s]$$

- Learn V_π from K episodes under policy π
 - $\{S_{k,0}, A_{k,0}, R_{k,1}, \dots, S_{k,T}\}_{k=1}^K \sim \pi$
- Simplest temporal-difference learning algorithm: TD(0)
 - Loop for new iterations:
 - Sample $(S_t, A_t, R_{t+1}, S_{t+1})$ with replacement from the transitions in the episodes
 - Update value $V(S_t)$ toward estimated return $R_{t+1} + \gamma V(S_{t+1})$:
 - $V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
 - $R_{t+1} + \gamma V(S_{t+1})$ is called the *TD target*
 - $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD error*
- If we expand one step further, we get TD(1)
 - $V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma R_{t+1} + \gamma^2 V(S_{t+2}) - V(S_t))$
 - Similarly, we can have TD(2), TD(3), ...

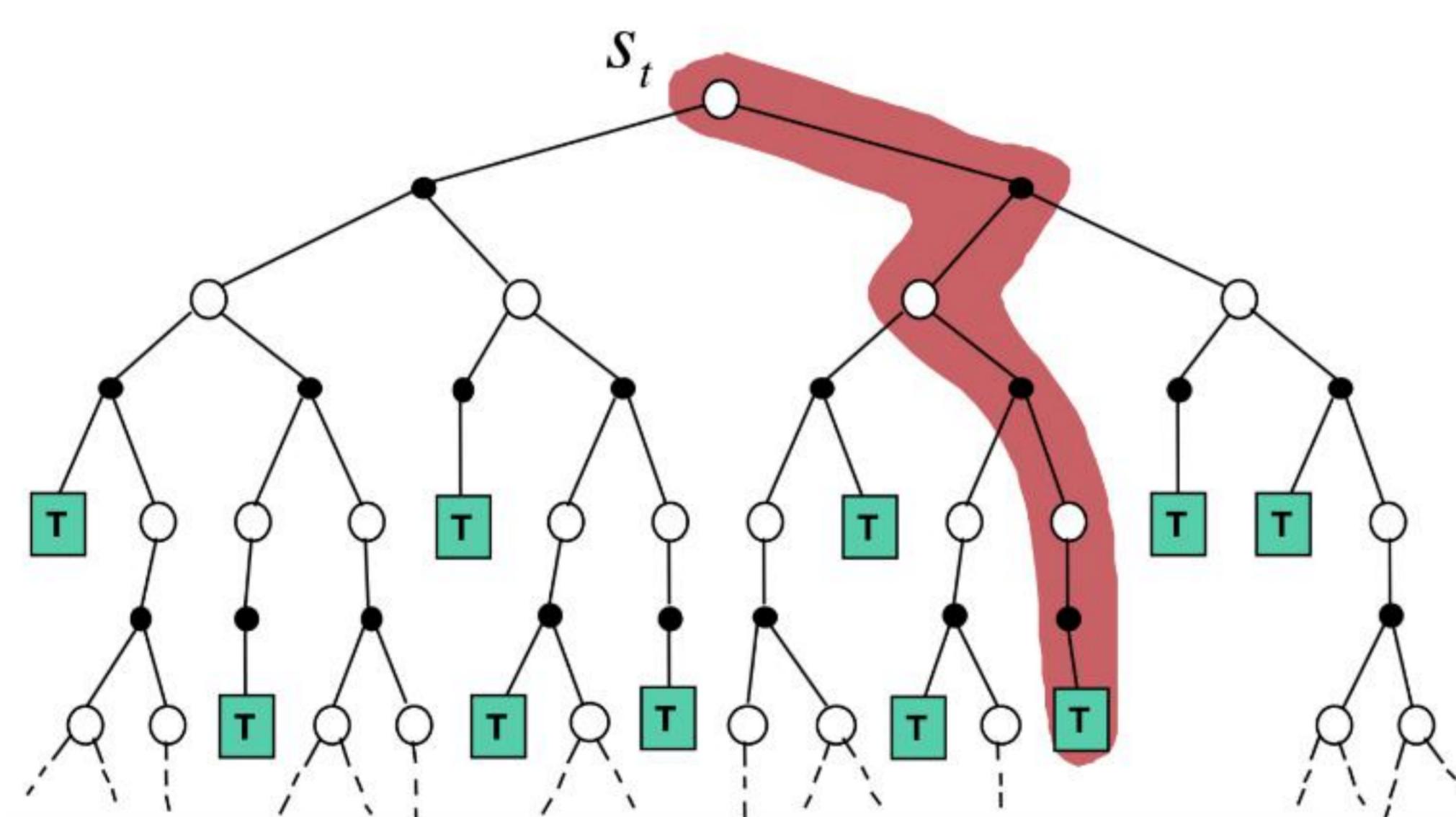
Temporal-Difference Learning

- Quick facts:
 - TD methods learn directly from episodes of experience
 - TD is model-free: no knowledge of MDP transitions / rewards
 - TD learns from incomplete episodes, by **bootstrapping**
 - TD updates a guess towards a guess

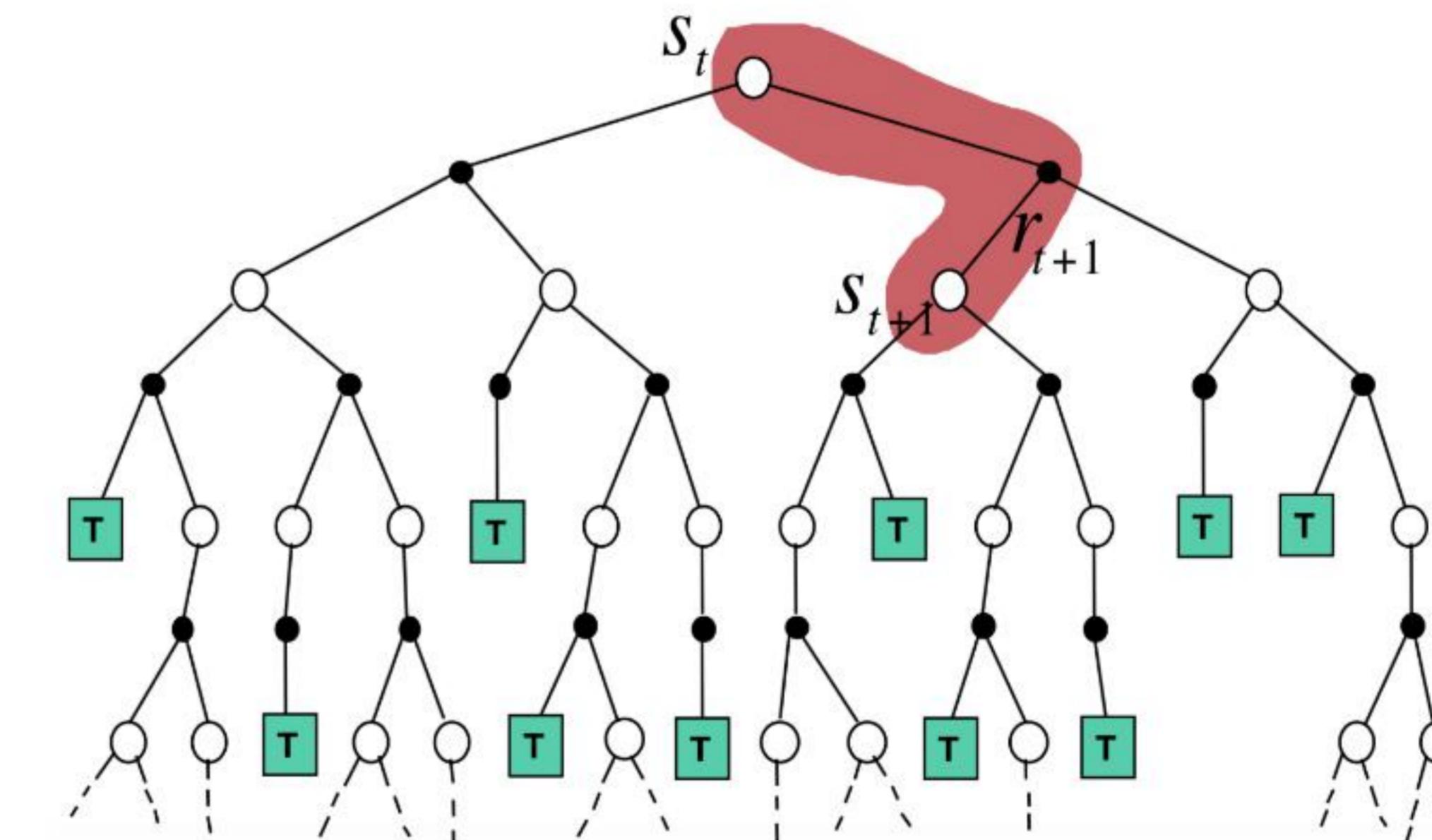
Key Differences between MC and TD

- MC estimates values based on *rollout*
- TD estimates values based on Bellman equation

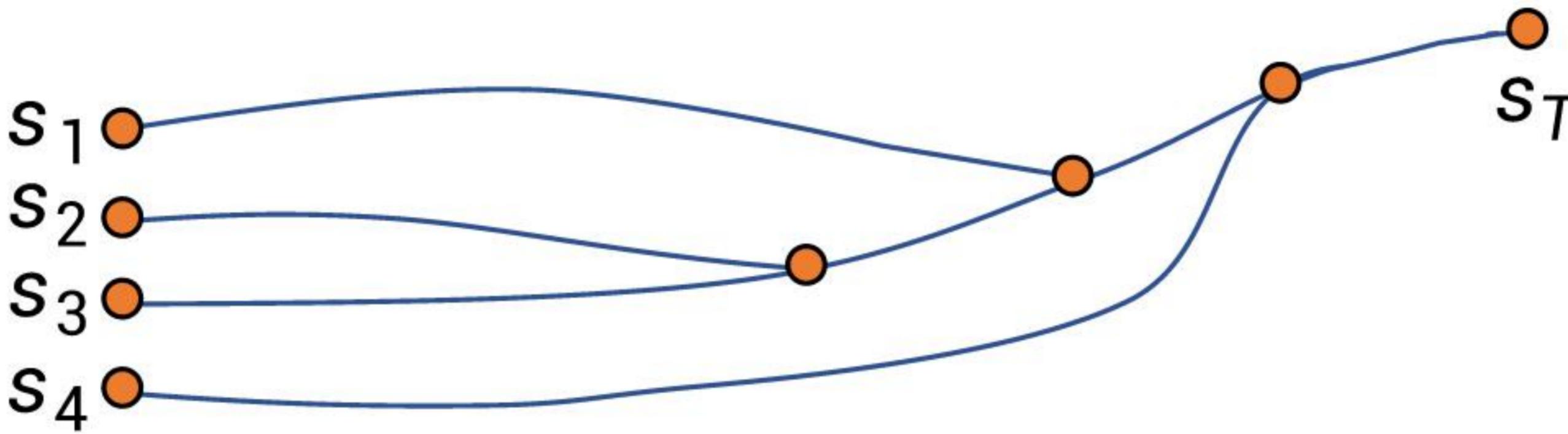
$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t))$$



$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

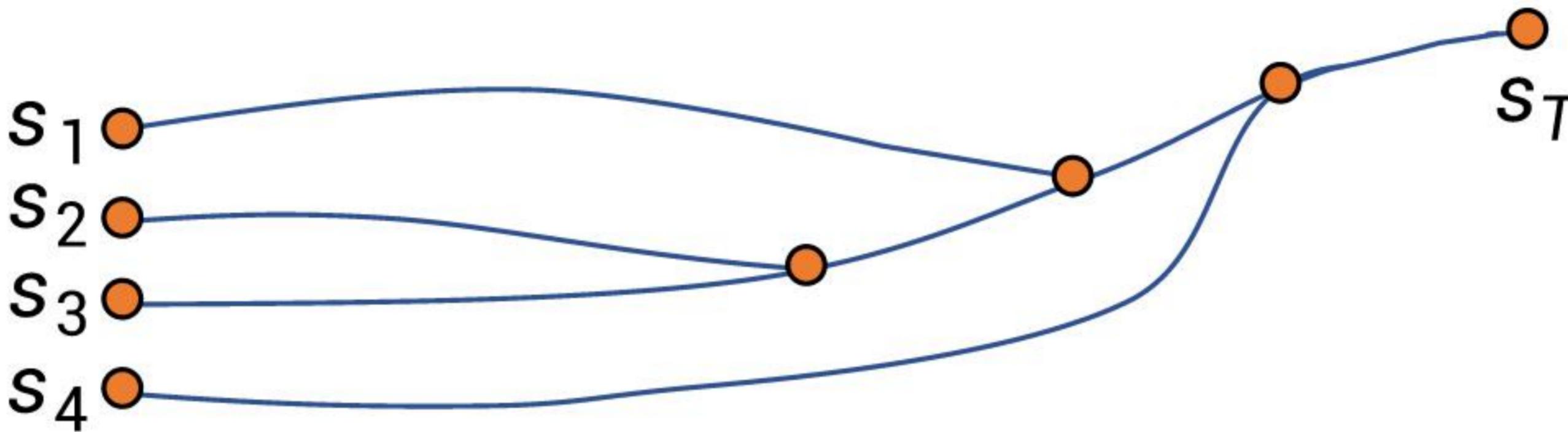


Pros and Cons of MC vs. TD



- TD can learn *before* knowing the final outcome
 - MC must wait until the end of episodes
 - TD can learn online after every step

Pros and Cons of MC vs. TD



- TD can learn *without* the final outcome
 - MC can only learn from complete sequences
 - When some episodes are incomplete, TD can still learn
 - MC only works for episodic (terminating) environments
 - TD works in non-terminating environments

Bias/Variance Trade-Off

- Return $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T$ is always an unbiased estimate of $V_\pi(S_t)$
- The true TD target $R_{t+1} + \gamma V_\pi(S_{t+1})$ is an unbiased estimate of $V_\pi(S_t)$
- If we will **update** π slowly along the learning process,

Bias/Variance Trade-Off

- Return $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T$ is always an unbiased estimate of $V_\pi(S_t)$
- The true TD target $R_{t+1} + \gamma V_\pi(S_{t+1})$ is an unbiased estimate of $V_\pi(S_t)$
- If we will **update** π slowly along the learning process,
 - the TD target $R_{t+1} + \gamma V(S_{t+1})$ becomes a *biased estimate* of $V_\pi(S_t)$, because
 - $V(S_{t+1})$ is a return estimation from the previous π .

Bias/Variance Trade-Off

- Return $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T$ is always an unbiased estimate of $V_\pi(S_t)$
- The true TD target $R_{t+1} + \gamma V_\pi(S_{t+1})$ is an unbiased estimate of $V_\pi(S_t)$
- If we will **update** π slowly along the learning process,
 - the TD target $R_{t+1} + \gamma V(S_{t+1})$ becomes a *biased estimate* of $V_\pi(S_t)$, because
 - $V(S_{t+1})$ is a return estimation from the previous π .
 - However, the TD target also has *much lower variance* than the return G_t , because
 - the return G_t is from a single rollout, heavily affected by the randomness (actions, transitions, and rewards) in all the future steps;
 - whereas the TD target is affected by the randomness in the *next one step*, and the low variance of the $V(S_{t+1})$ estimation from many historical rollouts.

Pros and Cons of MC vs TD (2)

- MC has high variance, zero bias
 - Good convergence properties (even with function approximation)
 - Independent of the initial value of V
 - Very simple to understand and use
- TD has low variance, some bias
 - Usually more efficient than MC
 - TD(0) converges to $V_\pi(s)$ (but not always with function approximation)
 - Has certain dependency on the initial value of V

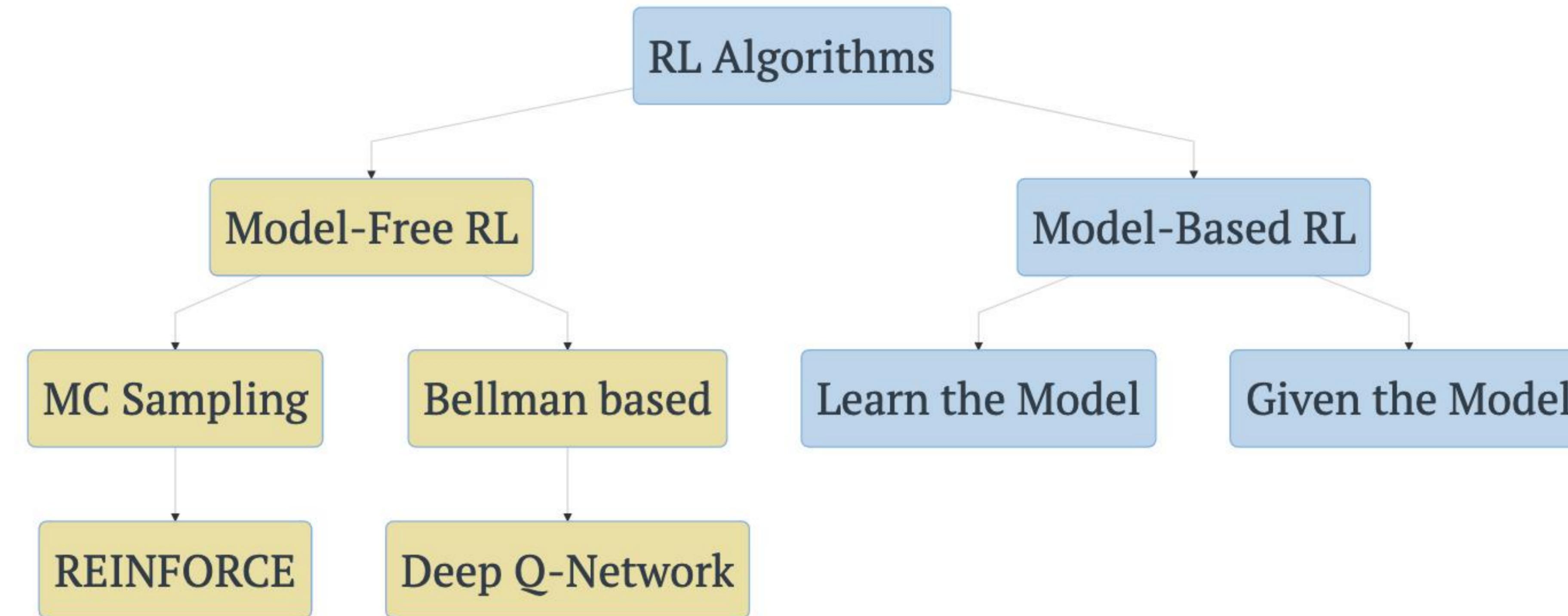
Bellman, or no Bellman, that is the question

- *Monte-Carlo sampling* and *Bellman equation* are two fundamental tools of value estimation for policies.
- Each has its pros and cons.
- Based on them, there are two families of model-free RL algorithms, both well developed. Some algorithms leverage both.
- Fundamentally, it is about the balance between bias and variance (sample complexity).

We start from REINFORCE and Deep Q-Learning

- Reason I:
 - REINFORCE uses Monte-Carlo sampling
 - Deep Q-Learning (DQN) uses Bellman equation
- Reason II:
 - REINFORCE only has a **policy network**
 - DQN only has a **value network**

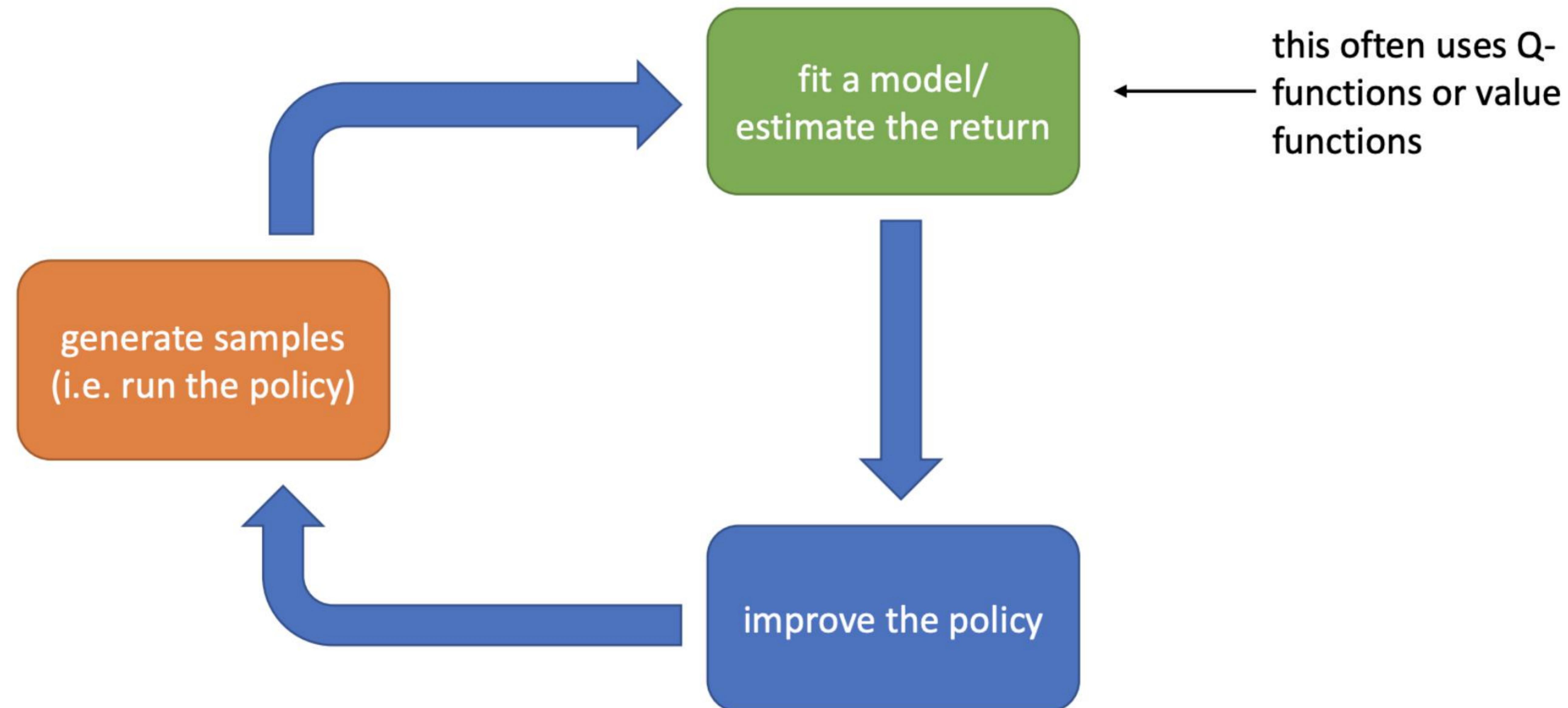
Taxonomy of RL Algorithms and Examples



Q-Learning for Tabular RL

Tabular RL: RL with discrete and finite state space, convenient for algorithm development and convergence analysis

The Anatomy of an RL algorithm



CS285 taught at UC Berkeley by Prof. Sergey Levine.

- Suppose we are going to learn the Q-function. Let us follow the previous flow chart and answer three questions:

1. Given transitions $\{(s, a, s', r)\}$ from some trajectories, how to improve the current Q-function?
2. Given Q , how to improve policy?
3. Given π , how to generate trajectories?

- Suppose we are going to learn the Q-function. Let us follow the previous flow chart and answer three questions:
 1. Given transitions $\{(s, a, s', r)\}$ from some trajectories, how to improve the current Q-function?
 - By Temporal Difference learning, the update target for $Q(S, A)$ is
 - $R + \gamma \max_a Q(S', a)$
 - Take a small step towards the target
 - $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
 2. Given Q , how to improve policy?
 3. Given π , how to generate trajectories?

- Suppose we are going to learn the Q-function. Let us follow the previous flow chart and answer three questions:
 1. Given transitions $\{(s, a, s', r)\}$ from some trajectories, how to improve the current Q-function?
 - By Temporal Difference learning, the update target for $Q(S, A)$ is
 - $R + \gamma \max_a Q(S', a)$
 - Take a small step towards the target
 - $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
- 2. Given Q , how to improve policy?
 - Take the greedy policy based on the current Q
 - $\pi(s) = \operatorname{argmax}_a Q(s, a)$
- 3. Given π , how to generate trajectories?

- Suppose we are going to learn the Q-function. Let us follow the previous flow chart and answer three questions:

1. Given transitions $\{(s, a, s', r)\}$ from some trajectories, how to improve the current Q-function?

- By Temporal Difference learning, the update target for $Q(S, A)$ is
 - $R + \gamma \max_a Q(S', a)$
 - Take a small step towards the target
 - $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

2. Given Q , how to improve policy?

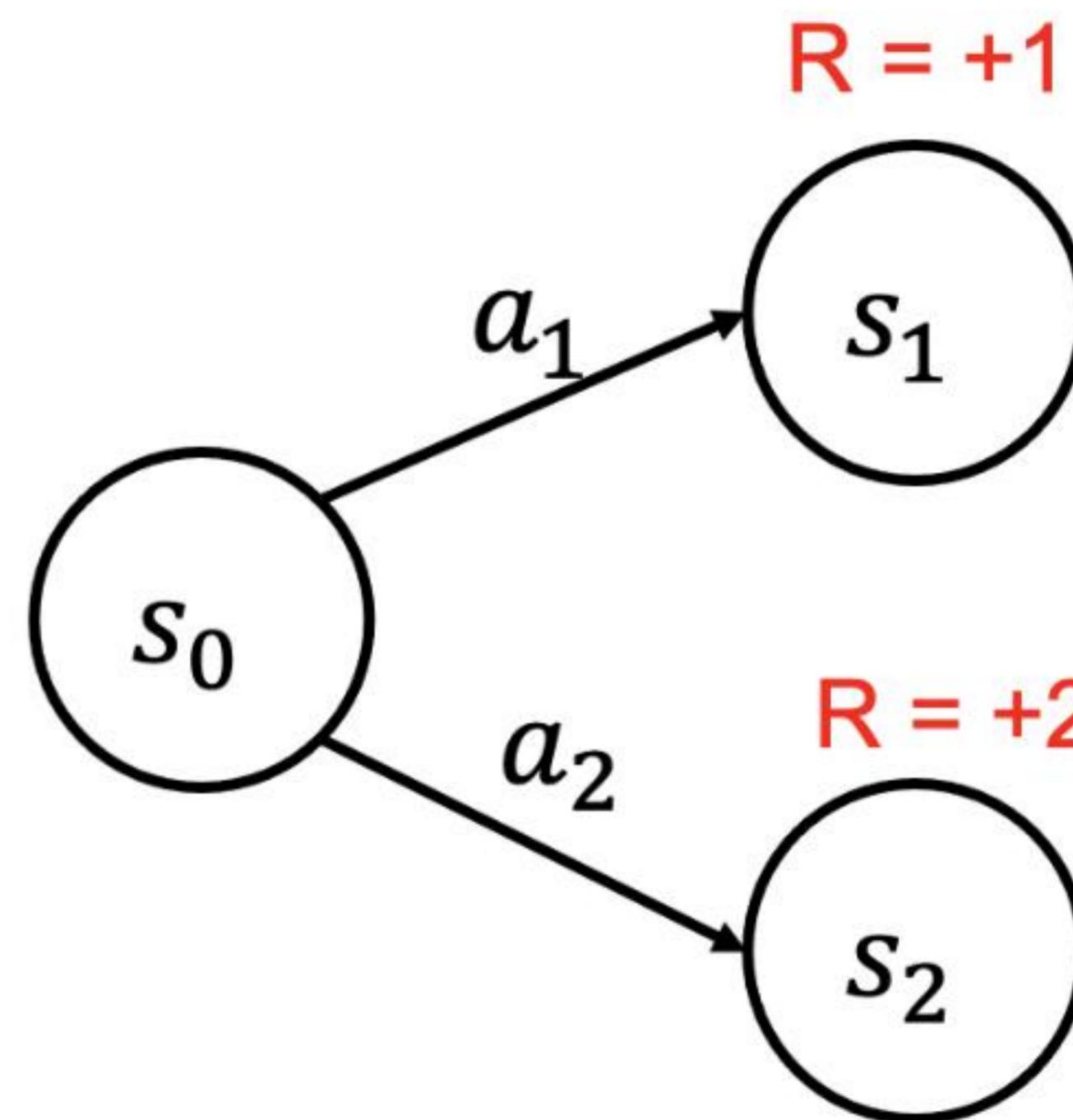
- Take the greedy policy based on the current Q
 - $\pi(s) = \operatorname{argmax}_a Q(s, a)$

3. Given π , how to generate trajectories?

- Simply run the greedy policy in the environment.
- Any issues?

Failure Example

- Initialize Q
 - $Q(s_0, a_1) = 0, Q(s_0, a_2) = 0$
 - $\pi(s_0) = a_1$
- Iteration 1: take a_1 and update Q
 - $Q(s_0, a_1) = 1, Q(s_0, a_2) = 0$
 - $\pi(s_0) = a_1$
- Iteration 2: take a_1 and update Q
 - $Q(s_0, a_1) = 1, Q(s_0, a_2) = 0$
 - $\pi(s_0) = a_1$
- ...
- **Q stops to improve because the agent is too greedy!**



ϵ -Greedy Exploration

- The simplest and most effective idea for ensuring continual exploration
- With probability $1 - \epsilon$ choose the greedy action
- With probability ϵ choose an action at random
- All m actions should be tried with non-zero probability
- Formally,

$$\pi_*(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon, & \text{if } a = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a) \\ \epsilon/m, & \text{otherwise} \end{cases}$$

Exploration vs Exploitation

- Exploration
 - finds more information about the environment
 - may waste some time
- Exploitation
 - exploits known information to maximize reward
 - may miss potential better policy
- Balancing exploration and exploitation is a key problem of RL. We will spend a lecture to discuss advanced exploration strategies.

Q-Learning Algorithm

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

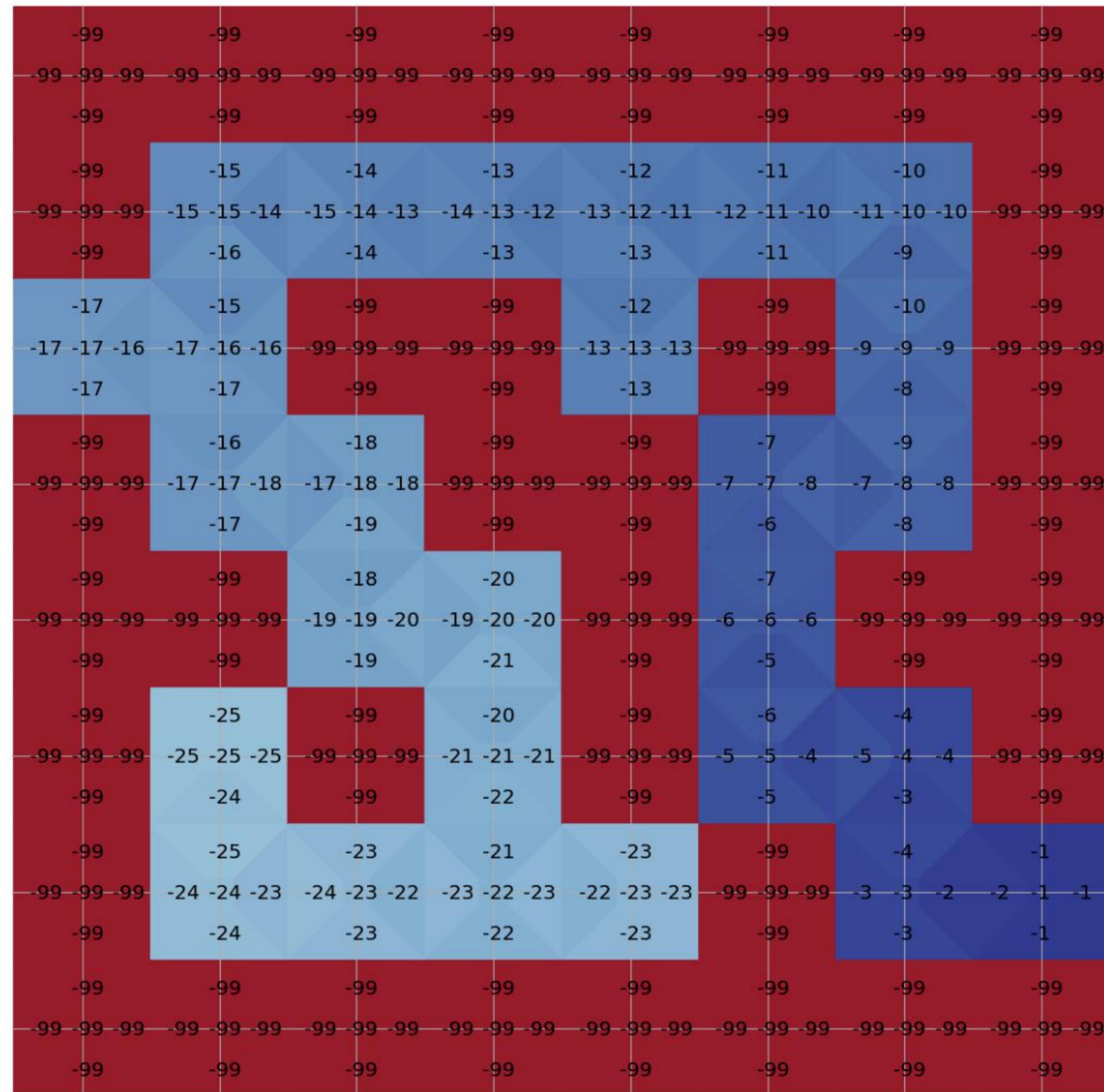
 Choose A from S using policy derived from Q (e.g., ε -greedy)

 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



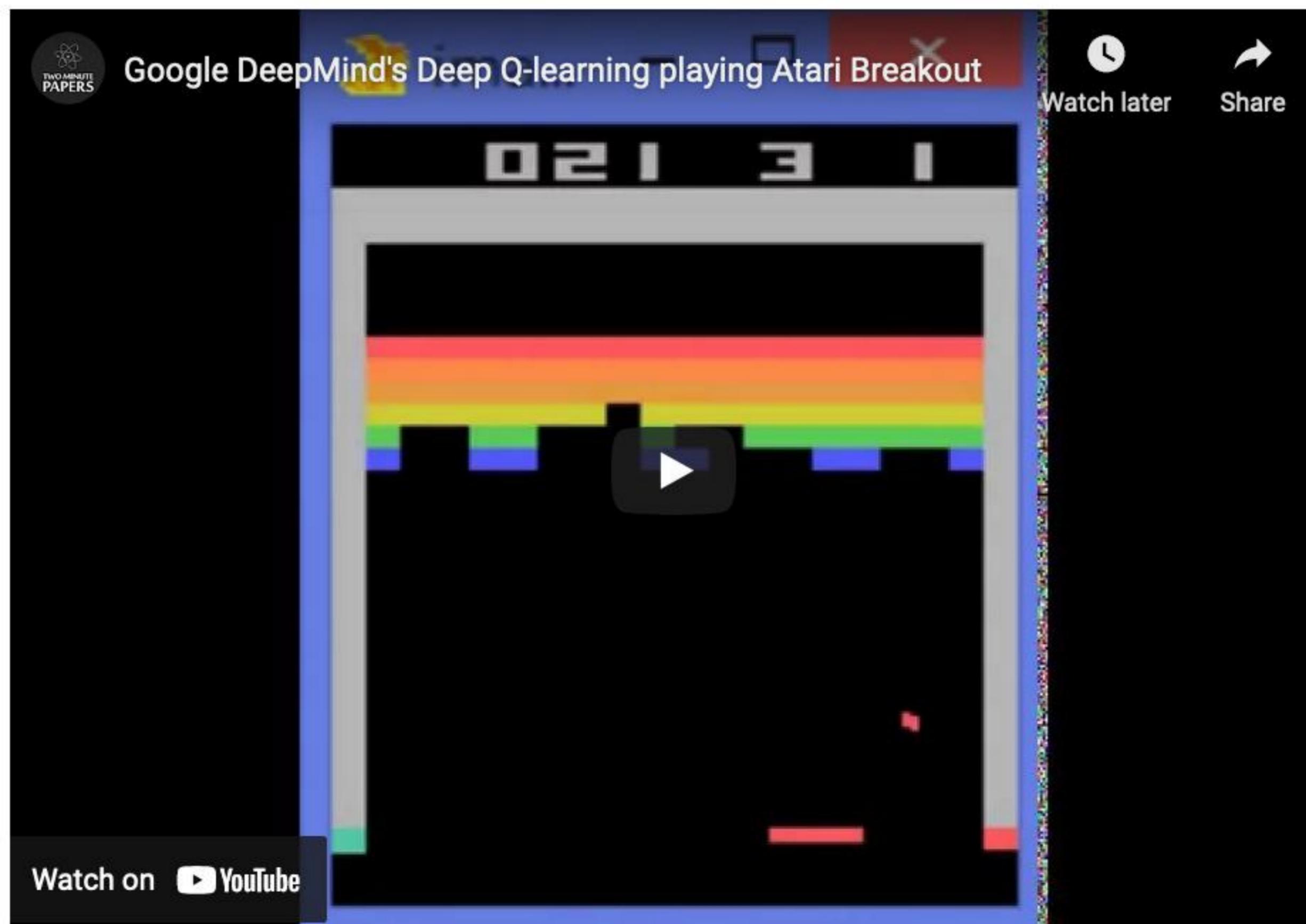
Running Q-learning on Maze

Plot with the tools in [an awesome playground for value-based RL](#) from Justin Fu

Deep Q-Learning

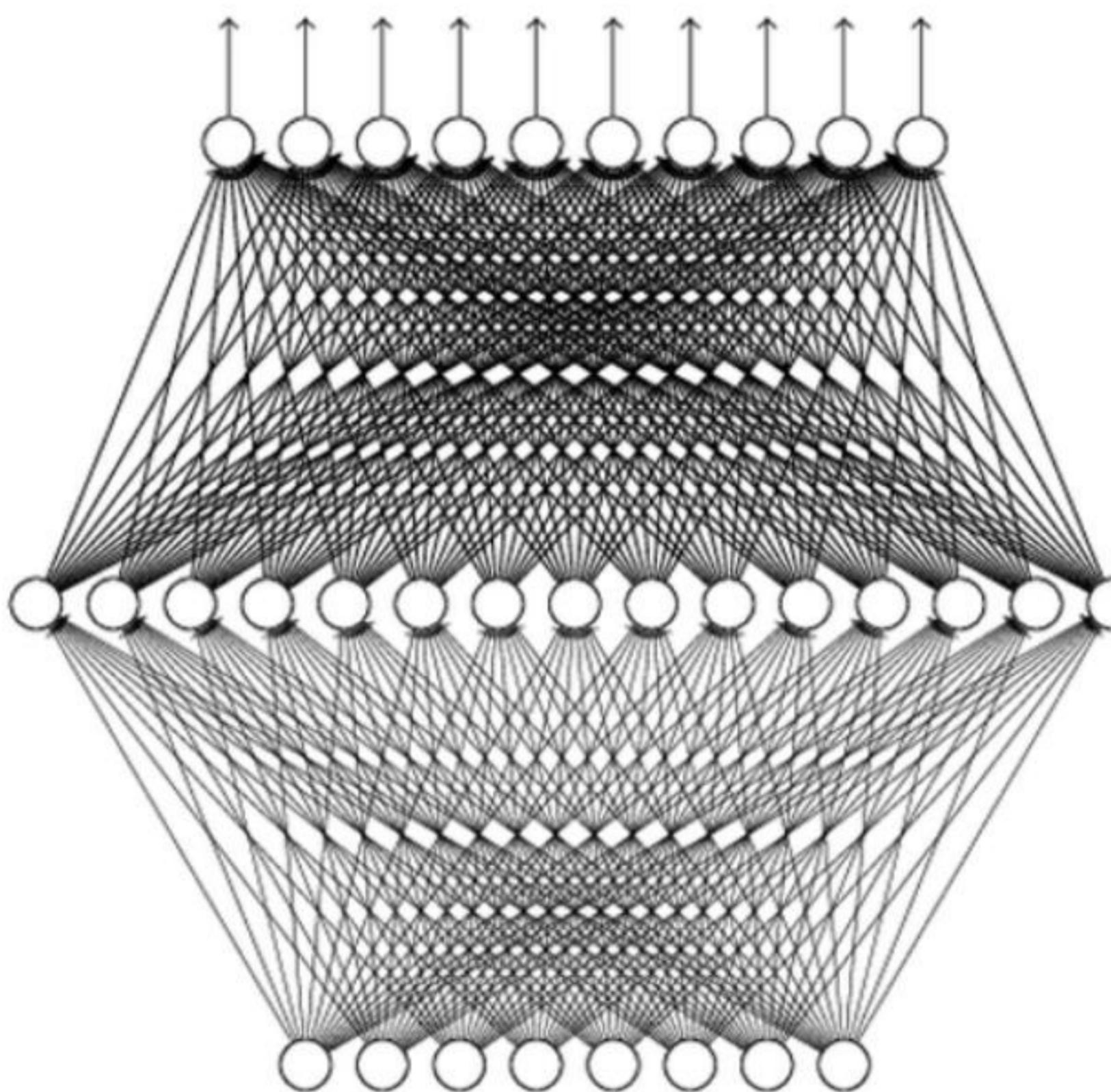
Challenge of Representing Q

- How do we represent $Q(s, a)$?
- Maze has a discrete and small *state space* that we can deal with by an array.
- However, for many cases the state space is continuous, or discrete but huge, array does not work.



Deep Value Network

- Use a neural network to parameterize Q :
 - Input: state $s \in \mathbb{R}^n$
 - Output: each dimension for the value of an action $Q(s, a; \theta)$



Training Deep Q Network

- Recall the Bellman optimality equation for action-value function:

$$Q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a]$$

- It is natural to build an *optimization problem*:

$$L(\theta) = \mathbb{E}_{(s, a, s') \sim Env}[TD_\theta(s, a, s')] \quad (\text{TD loss})$$

where $TD_\theta(s, a, s') = \|Q_\theta(s, a) - [R(s, a, s') + \gamma \max_{a'} Q_\theta(s', a')]\|^2$.

- Note: How to obtain the *Env* distribution has many options!
 - It does not necessarily sample from the optimal policy.
 - A suboptimal, or even bad policy (e.g., random policy), may allow us to learn a good Q .
 - It is a cutting-edge research topic of studying how well we can do for non-optimal *Env* distribution.

Replay Buffer

- As in the previous Q-learning, we consider a routine that we take turns to
 - Sample certain transitions using the current Q_θ
 - Update Q_θ by minimizing the TD loss
- **Exploration:**
 - We use ϵ -greedy strategy to sample transitions, and add (s, a, s', r) in a **replay buffer** (e.g., maintained by FIFO).
- **Exploitation:**
 - We sample a batch of transitions and train the network by gradient descent:

$$\nabla_\theta L(\theta) = \mathbb{E}_{(s,a,s') \sim \text{ReplayBuffer}} [\nabla_\theta TD_\theta(s, a, s')]$$

Deep Q-Learning Algorithm

- Initialize the replay buffer D and Q network Q_θ .
- For every episode:
 - Sample the initial state $s_0 \sim P(s_0)$
 - Repeat until the episode is over
 - Let s be the current state
 - With prob. ϵ sample a random action a . Otherwise select $a = \arg \max_a Q_\theta(s, a)$
 - Execute a in the environment, and receive the reward r and the next state s'
 - Add transitions (s, a, s') in D
 - Sample a random batch from D and build the batch TD loss
 - Perform one or a few gradient descent steps on the TD loss

Some Engineering Concerns about Deep *Q*-Learning

- States and value network architecture
 - First of all, a good computer vision problem worth research.
 - Need to ensure that states are sufficient statistics for decision-making
 - Common practice: Stack a fixed number of frames (e.g., 4 frames) and pass through ConvNet
 - If long-term history is important, may use LSTM/GRU/Transformer/... to aggregate past history
 - May add other computing structures that are effective for video analysis, e.g., optical flow map
 - Not all vision layers can be applied without verification (e.g., batch normalization layer may be harmful)
- Replay buffer
 - Replay buffer size matters.
 - When sampling from the replay buffer, relatively large batch size helps stabilizing training.

Some Theoretical Concerns about Q -Learning

- Behavior/Target Network: Recall that $TD_{\theta}(s, a, s') = \|Q_{\theta}(s, a) - [R(s, a, s') + \gamma \max_{a'} Q_{\theta}(s', a')]\|^2$. We keep two Q networks in practice. We only update the blue network by gradient descent and use it to sample new trajectories. Every few episodes we replace the red one by the blue one. The reason is that the blue one changes too fast. The red one is called *target network* (to build target), and the blue one is called *behavior network* (to sample actions).
- Value overestimation: Note that the TD loss takes the maximal a for each $Q(s, \cdot)$. Since TD loss is not unbiased, the max operator will cause the Q -value to be overestimated! There are methods to mitigate (e.g., double Q -learning) or work around (e.g., advantage function) the issue.
- Uncertainty of Q estimation: Obviously, the Q value at some (s, a) are estimated from more samples, and should be more trustable. Those high Q value with low confidence are quite detrimental to performance. Distributional Q -Learning quantifies the confidence of Q and leverages the confidence to recalibrate target values and conduct exploration.
- Theoretically, Q -learning (more precisely, a variation of it) is an **optimal online learning algorithm** for tabular RL.

REINFORCE

Key Idea

- Unlike Q-learning, REINFORCE method does not need to maintain the value function of states or state-action pairs!
- It uses a neural network to parameterize the policy.
- We update the policy network by applying stochastic gradient descent of the return.

Policy Gradient

- The return of a policy π_θ parameterized by a neural network is:

$$J(\theta) = \mathbb{E}_{\tau \sim P(\pi_\theta)}[R(\tau)] = \int_{\tau} P(\tau)R(\tau)d\tau$$

- Its gradient is

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} \int_{\tau} R(\tau)P(\tau|\theta)d\tau = \int_{\tau} R(\tau)\nabla_{\theta} P(\tau|\theta)d\tau$$

Policy Gradient

- Note that

$$\nabla_{\theta} f(\theta) = \frac{\nabla f(\theta)}{f(\theta)} \Rightarrow \nabla f(\theta) = f(\theta) \nabla \log f(\theta)$$

- Therefore,

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \nabla_{\theta} \int_{\tau} R(\tau) P(\tau|\theta) d\tau = \int_{\tau} R(\tau) \nabla_{\theta} P(\tau|\theta) d\tau \\ &= \int_{\tau} \mathbf{P}(\tau|\theta) R(\tau) \nabla_{\theta} \log P(\tau|\theta) d\tau \\ &\approx \frac{1}{n} \sum_k R(\tau_k) \nabla_{\theta} \log P(\tau_k|\theta)\end{aligned}$$

- We can estimate the gradient of $J(\theta)$ by empirical mean!

Policy Gradient

- Note that

$$P(\tau|\theta) = \log P(s_0) + \sum_i \log \pi_\theta(a_i|s_i) + \log P(s_{i+1}|s_i, a_i)$$

- The environment model does not depend on θ . Therefore, the gradient of the return is

$$\nabla_\theta J(\theta) \approx \frac{1}{n} \sum_k R(\tau_k) \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i)$$

Intuitive Explanation

$$\nabla_{\theta} J(\theta) \approx \frac{1}{n} \sum_k R(\tau_k) \sum_i \nabla_{\theta} \log \pi_{\theta}(a_i | s_i)$$

- Weighted sum of (log) policy gradients for all the trajectories.
- Higher weights for trajectories with higher rewards.

Implementing Policy Gradient

- For discrete actions, π_θ is a soft-max function.
- For continuous actions, π_θ is a Gaussian distribution. We use the policy network to predict the mean and variance.

Policy Gradient Algorithm

- Initialize a policy network π_θ .
- Repeat
 - Sample trajectories $\{\tau_{1:T}^k\}_{k \leq n}$
 - Compute the gradient ∇J by policy gradient

$$\nabla_\theta J(\theta) \approx \frac{1}{n} \sum_k R(\tau_k) \sum_i \nabla_\theta \log \pi_\theta(a_i | s_i)$$

- Update

$$\theta \leftarrow \theta + \alpha \nabla J(\theta)$$

Some Comments on Policy Gradient Algorithm

- We will introduce improved version of policy gradient in subsequent lectures.
- As an MC-based method, the gradient estimation is unbiased.
- However, its estimate of gradient has a large variance. Therefore, stabilizing the update of π_θ is the key.
- As a high-variance method, it is not quite efficient (even its improved version, e.g, TRPO/PPO). But since it is unbiased, for some hard tasks, it may outperform seemingly more sample-efficient Q-learning methods.

Convergence of Reinforcement Learning Algorithms

We state the facts without proof:

- Q-Learning:
 - Tabular setup: Guaranteed convergence to the optimal solution. A simple proof (using contraction mapping).
 - Value network setup: No convergence guarantee due to the approximation nature of networks.
- Policy Gradient: Next lecture.