Reinforcement Learning II

Julien Prat (CNRS, IP Paris)

November 2023

Dynamic Programming

- Dynamic Programming (DP) is an optimization method that solves complex problems by breaking them down into simpler subproblems in a recursive manner.

- Dynamic Programming assumes that the agent knows:
  - ❑ The structure of the model
  - ❑ The law of motion of the state variable, including the impact of the control variable

Bayesian Learning

- Combining Bayesian Learning with DP alllows one to relax the assumption that the law of motion of the state variable is known with certainty.

- Bayesian learning still requires that the agent knows:
  - ❑ The structure of the model
  - ❑ The probabilistic structure of the law of motion of the state

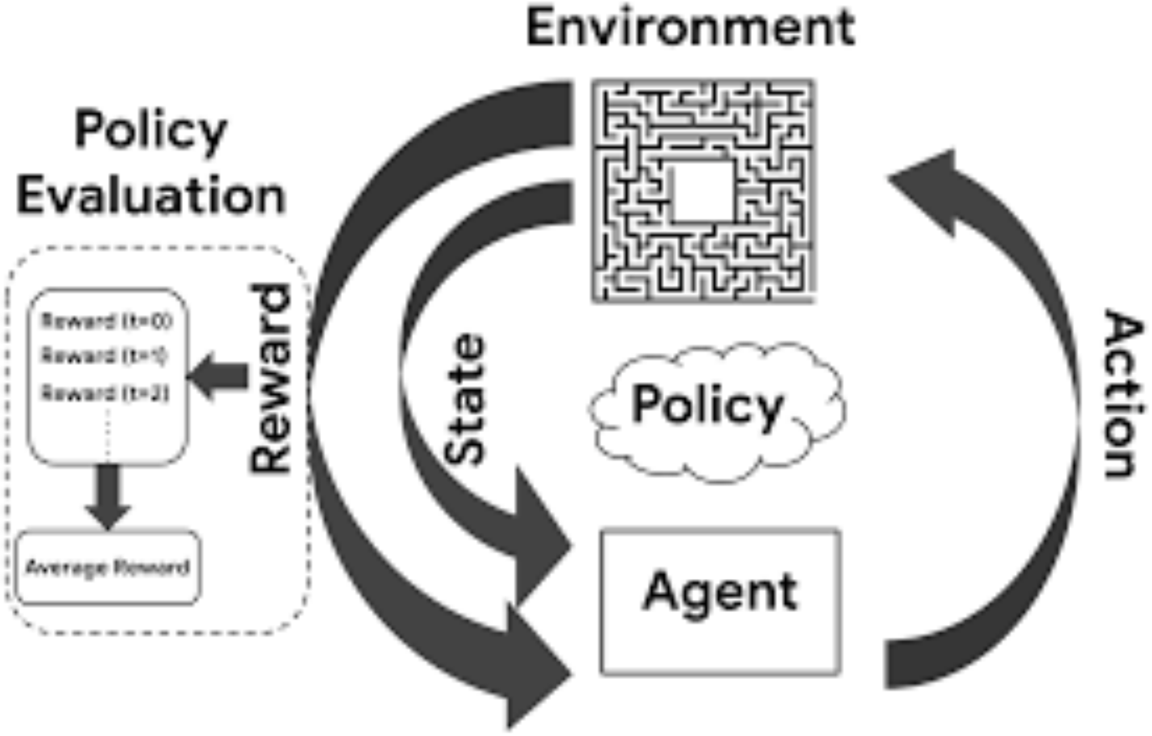
- Furthermore Bayesian learning also requires that we specify the prior of the agent

Reinforcement Learning

- In practice, the environment can be too complex to build an explicit model.

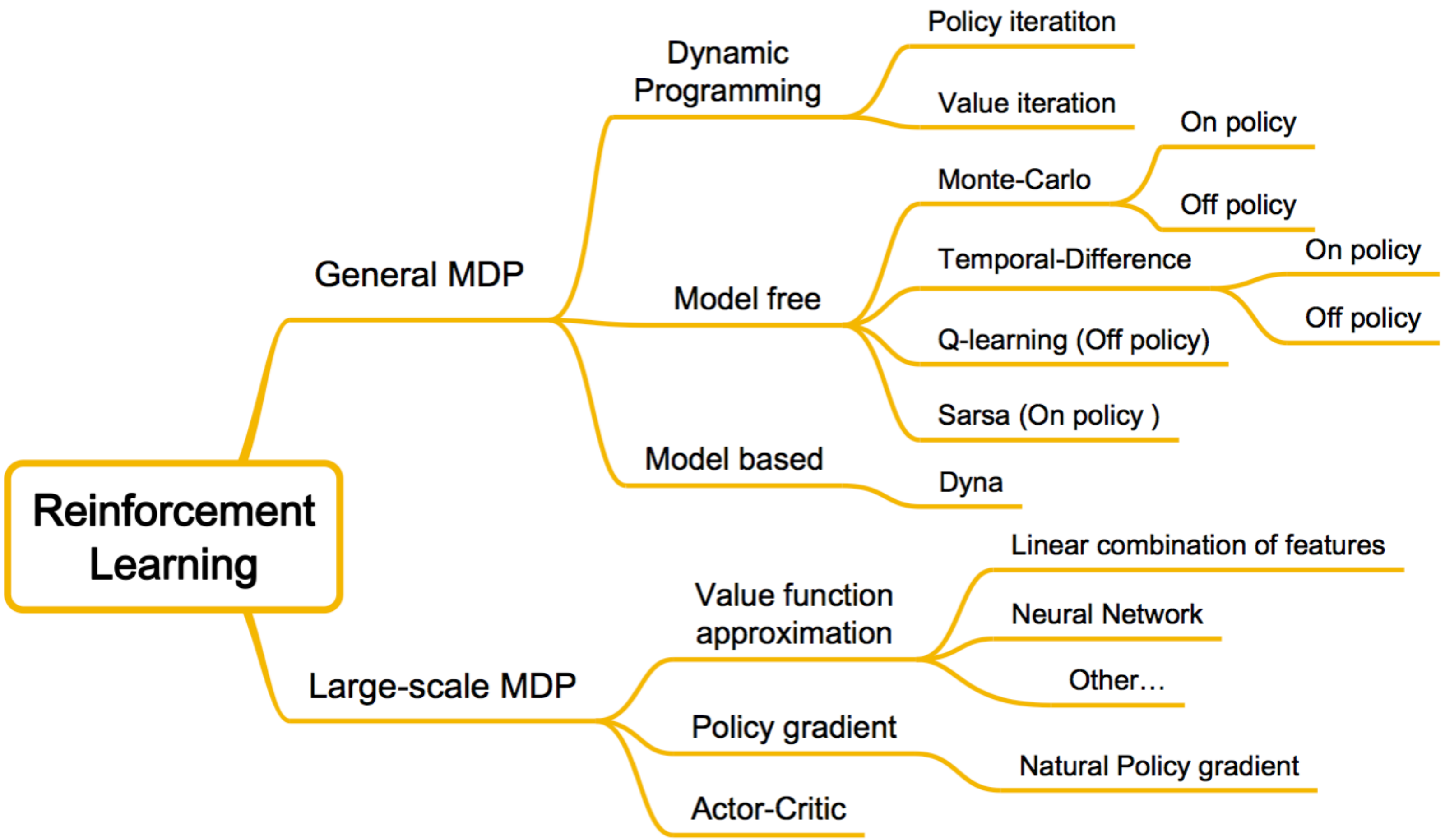
- Reinforcement learning is model free. It does not require knowledge of the payoff function and law of motion of the state.

- The agent learns how to maximize her reward by repeatedly interacting with the environment.



Reinforcement Learning

- DP can be seen as subfield of RL (MDP stands for Markov Decision Process)



Source: Zhao Mingming's class notes

Monte-Carlo

- Fix the policy function π and define the associated value function

$$v^{\pi}(s_0) = E^{\pi}\left[\sum_{t=0}^{T} \gamma^t r(s_{t+1}, s_t \,|\, s_0, a_t^{\pi})\right],$$

where T is the terminal time the episode.

- Run Monte-Carlo simulations to approximate the expected return of the policy function π with the empirical mean return.
- By the law of large numbers, the average value converges to the expected value.
- Note that a simulator for the payoff and transition functions are required.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow \text{average}(Returns(S_t))$

Model Free Control

- Extend insights from prediction to *optimize* the value function of an unknown MDP.

- Model Free control applicable when:
  ✓MDP model is unknown, but experience can be sampled.
  ✓MDP model is known, but is too big to use, except by samples.

- Define **Q-function** or state-action value function

$$Q(s,a) = E\left[\sum_{i=0}^{\infty} \gamma^i r(s_{t+i}, a_{t+i}) \,|\, s_t = s, a_t = a\right].$$

Monte-Carlo estimation of action values

- With a model, state values are sufficient to determine a policy. One simply looks one period ahead and choose the action that yields the optimal reward plus next period value.
- Without a model, one must explicitly estimate the value of each action-state pair.
- Monte-Carlo simulations can be used to estimate $Q_\pi(s,a)$, i.e. the value of starting in *(s,a)* and following the policy π .
- Estimation converges under the following hypotheses:
  1. Infinite number of episodes;
  2. Exploring starts: all state-action pairs have a nonzero probability of being selected at the start of an episode.

Monte-Carlo Control

- After each simulated episode, update the guessed policy by making it greedy with respect to the action value function.
- Using the action value function to update the policy ensures that we do not need a model to construct the greedy policy.
- Alternate between evaluation and improvement on an episode-by-episode basis.

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
    $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
    $Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
    Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
    Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append G to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
            $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Temporal-Difference Learning

- Monte-Carlo learns from completed episodes

$$V(s) \leftarrow V(s) + \alpha[G - V(s)],$$

where α is the learning rate parameter and $G \equiv \sum_{t=0}^{T} \gamma^t r_t$ is the simulated return.

TD(0)

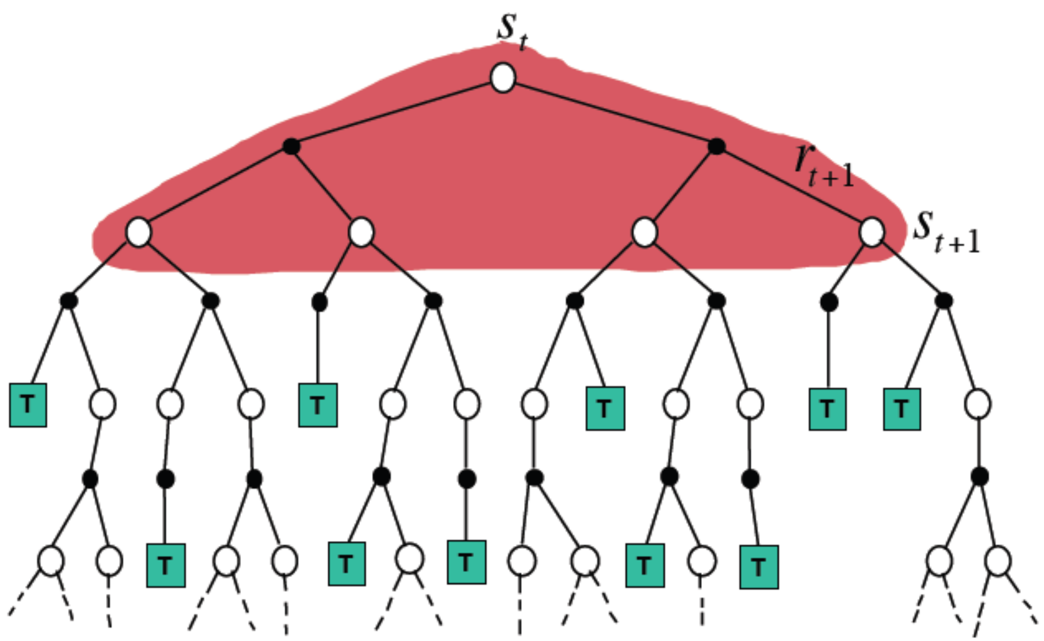
- Temporal-Difference uses bootstrapping to learn from incomplete episodes, i.e. learning occurs at *every step*:

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)] .$$

DP vs. TD

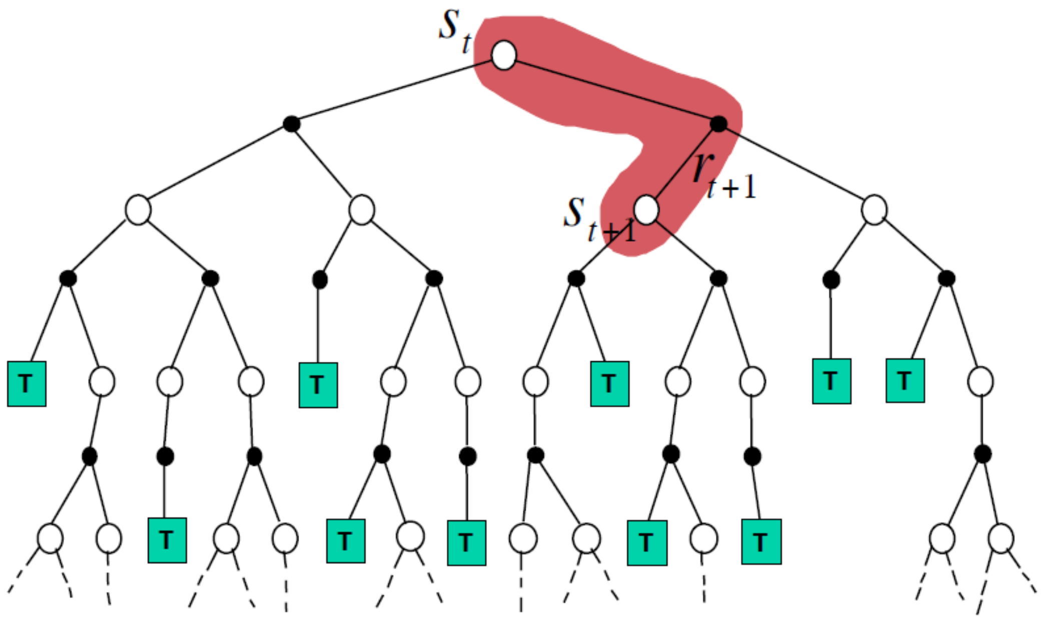
DP Backups

$$v^{\pi}(s) \leftarrow E^{\pi}[r(s', s, a) + \gamma v^{\pi}(s')]$$



TD Backups

$$v^{\pi}(s) \leftarrow v^{\pi}(s) + \alpha[r(s', s, a) + \gamma v^{\pi}(s') - v^{\pi}(s)]$$



Source: David Silver's class notes

Sarsa

- Apply TD to action value function:

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

- Update performed after *every* transition from a nonterminal state.

- Transitions and updates are both determined by an on-policy algorithm.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

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 $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until S is terminal

Q-learning

- Use *off-policy* TD control

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)] .$$

- Update is independent of policy that determines which state-action pairs are visited (and thus updated).

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 $Q(\mathit{terminal}, \cdot) = 0$

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'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

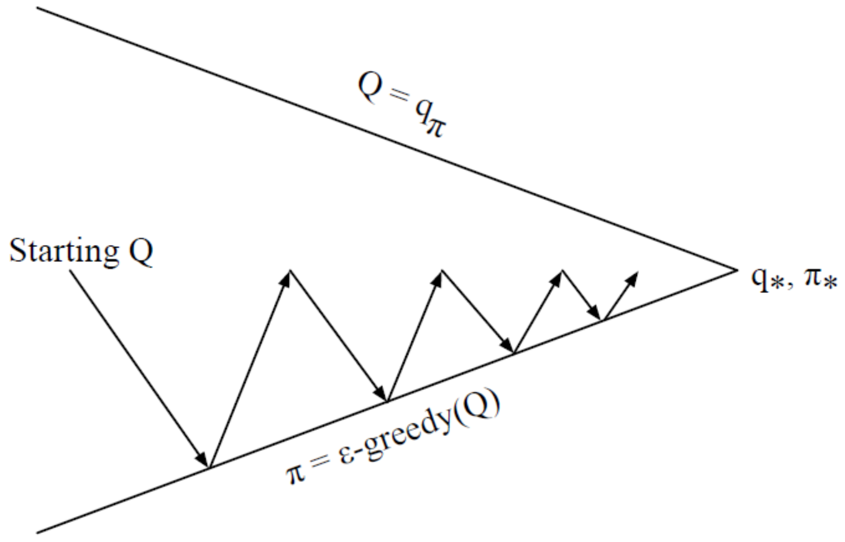
Q-learning

- Since $V(s) = max_a Q(s, a)$, Q-learning algorithm converges to the solution of Value Function iteration.

- However, there is a key difference:
  - Value function iteration requires knowledge of the MDP.
  - Q-learning can be performed without an explicit model of the MDP. Instead, one needs to simulate enough exploration paths.

- Greedy algorithm might prevent exploration (remember the state-action space must be scanned). ε-greedy exploration:
  ✓ With probability 1-ε select greedy action
  ✓ With probability ε select action at random

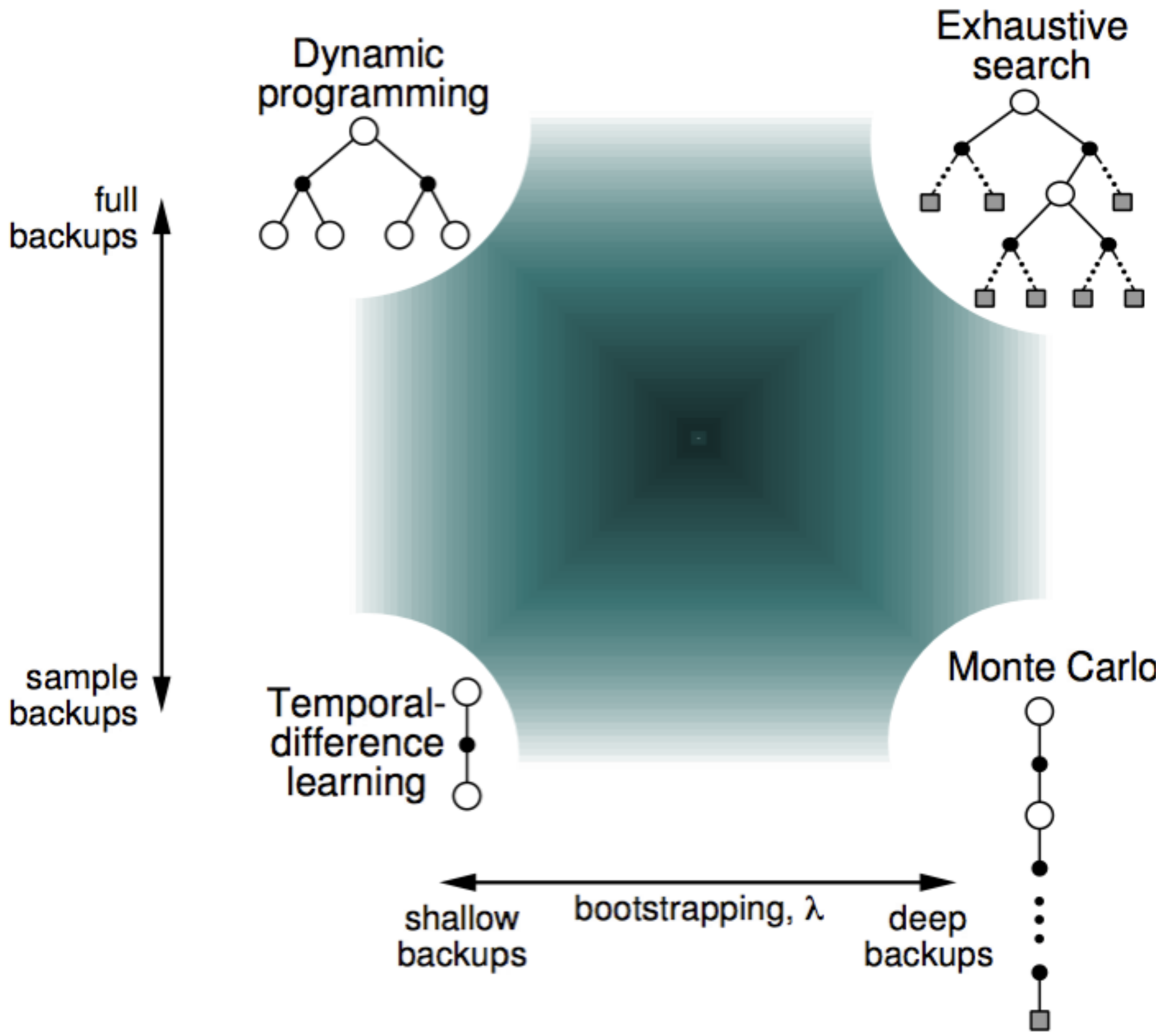
Greedy policy improvements



Unifying Matrix

- **Monte-Carlo:**
  - Pros: No bias, little dependence to initial conditions, works in non-Markovian settings
  - Cons: requires completed episodes, high variance

- **Temporal Difference:**
  - Pros: Usually faster than MC, works in non-terminating environments (infinite horizon), low variance
  - Cons: sensitive to initial conditions, biased



Source: David Silver's class notes

Large Scale MDPs

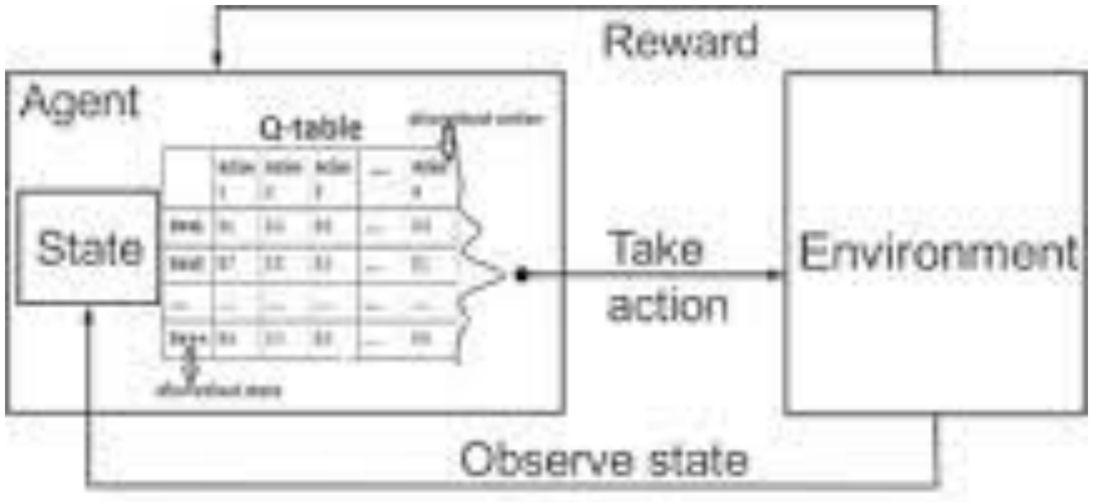
- In large scale problem, the table representation of the Q-function is too big and has to be approximated:

1. Linear approximation of the value function

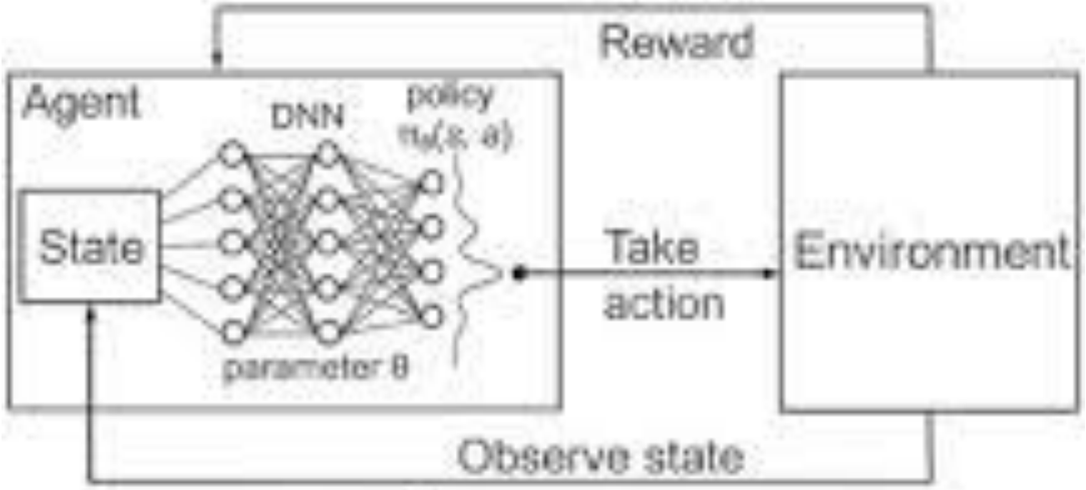
$$Q_\theta(x,a) = \sum_{r=1}^{K} \theta_r \phi_r(x,a) = \phi^T(x,a)\theta$$

where vector θ is identified by gradient descent to minimize error.

2. Neural network as function approximater.



a. Q-learning



b. Deep Q-learning

Source: “Deep reinforcement learning enabled self-learning control for energy efficient driving”, Qi et al., 2019