Reinforcement Learning

Julien Prat (CNRS, IP Paris)

November 2022

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 asumes 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

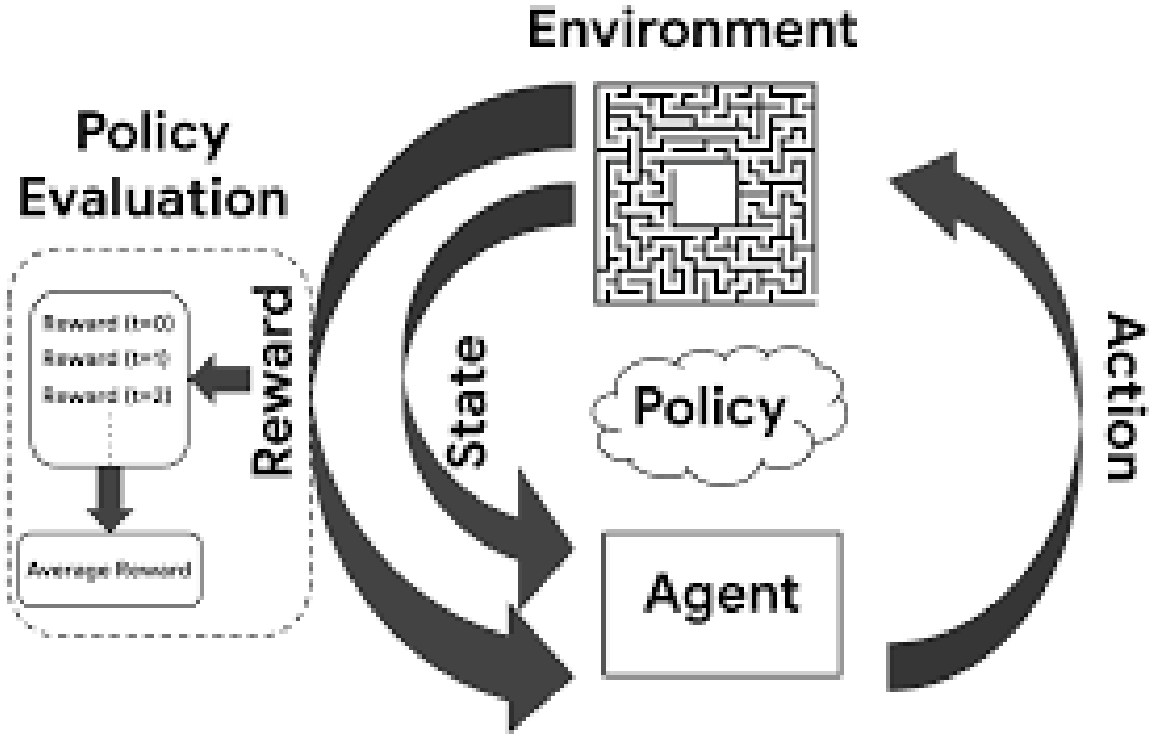
- Combining Bayesian Learning with DP alllows one to relax the assumption that the law of motion of the state variable is known with certainty.

- 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

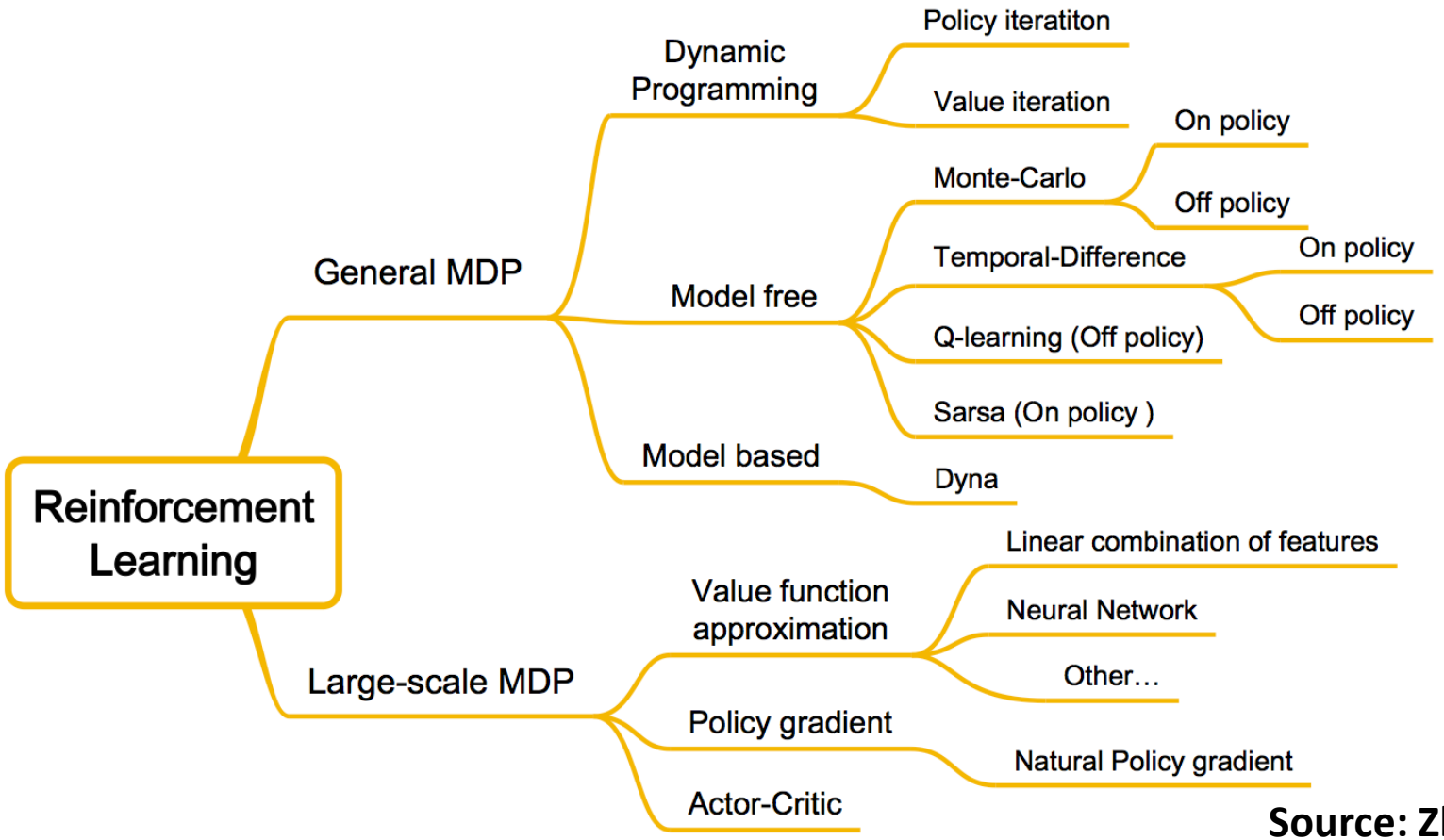
- In practice, the environment can be too complex to build an explicit model.
- Reinforcement learning is model free. It does not require knowledge of the payoff function and law of motion of the state.
- Agent learns how to maximize her reward by repeateadly interacting with the environment.



Reinforcement Learning

- DP can be seen as subfield of RL.

(Note: MDP stands for Markov Decision Process)



Source: Zhao Mingming's class notes

Monte-Carlo

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

$$V^g(x_0) = E^g\left[\sum_{t=0}^{T} \beta^t U(x_{t+1}, x_t | x_0, g)\right]$$

- Run Monte-Carlo experiments to approximate the expected return of the policy function g with the empirical mean return.
- By the law of large number, the average value should converge to the expected value.
- Note that a simulator for the payoff and transition functions are required.

Temporal-Difference Learning

- Monte-Carlo learns from completed episodes

$$V^g(x) \longleftarrow V^g(x) + \alpha\,[G - V^g(x)]$$

where α is the learning rate parameter and $G \equiv \sum_{t=0}^{T} \beta^t U(x_{t+1}, x_t | x_0 = x, g)$ is the simulated return.

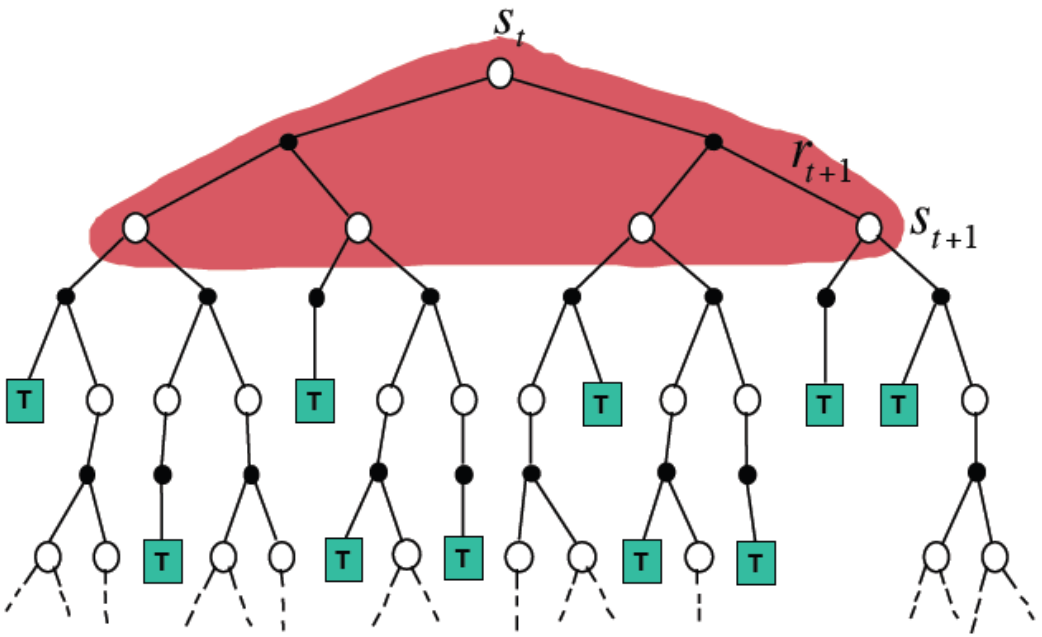
- By contrast Temporal-Difference uses bootstrapping to learn from incompleted episodes. Learning occurs at *every step*:

$$V^g(x) \longleftarrow V^g(x) + \alpha \underbrace{[U(x, x', a) + \beta V^g(x') - V^g(x)]}_{\text{TD Error}}$$

DP vs. TD

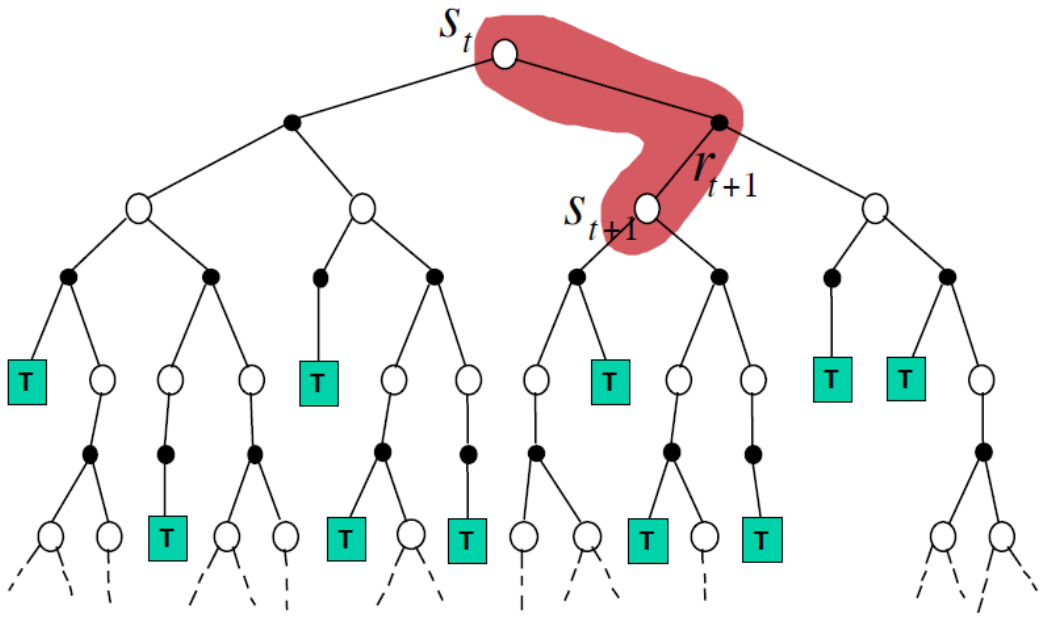
DP Backups

$$V^g(x) \longleftarrow E^g\left[U(x, x', a) + \beta V^g(x')\right]$$



TD Backups

$$V^g(x) \longleftarrow V^g(x) + \alpha\left[U(x, x', a) + \beta V^g(x') - V^g(x)\right]$$

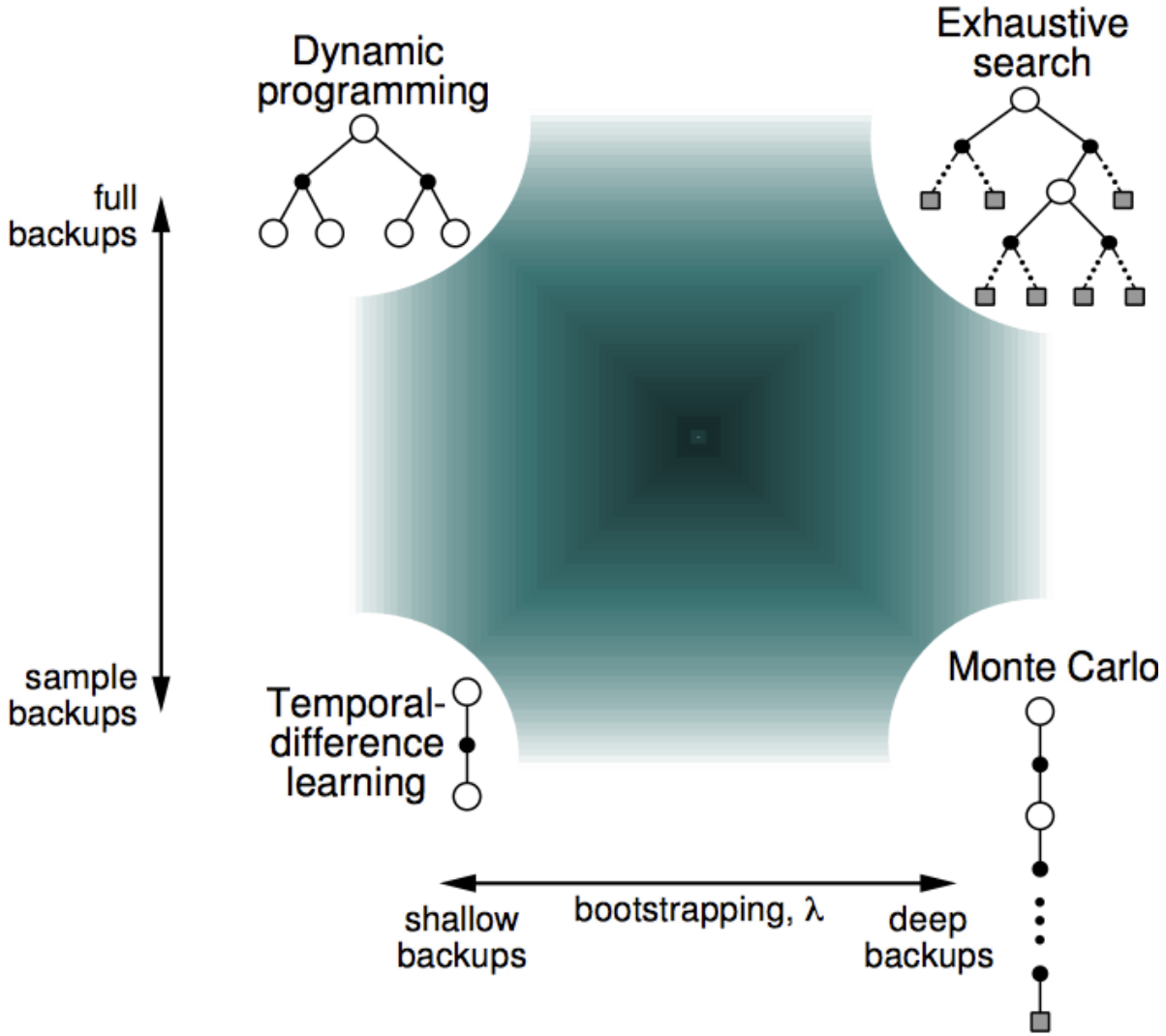


Source: David Silver's class notes

Unifying Matrix

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

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



Source: David Silver's class notes

Model Free Control

- Extend insights from prediction to *optimise* 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

$$
\begin{aligned}
Q(x,a) &\equiv U(x,a) + \beta E\left[V^*(x')\right] \\
&= U(x,a) + \beta E\left[\max_{a'} Q(x',a') | x,a\right]
\end{aligned}
$$

Q-learning

- In a deterministic context, the algorithm below approximates the Q-function

Deterministic Q-learning Algorithm

Initialize: $\hat{Q}_0(x,a) = Q_0(x,a), for\ all\ (x,u)$

for *r=1,2,…* do

Update: $\hat{Q}_{r+1}(x_k,a_k) = U(x_k,a_k) + \beta max_a \hat{Q}_r(x_{k+1},a)$

end

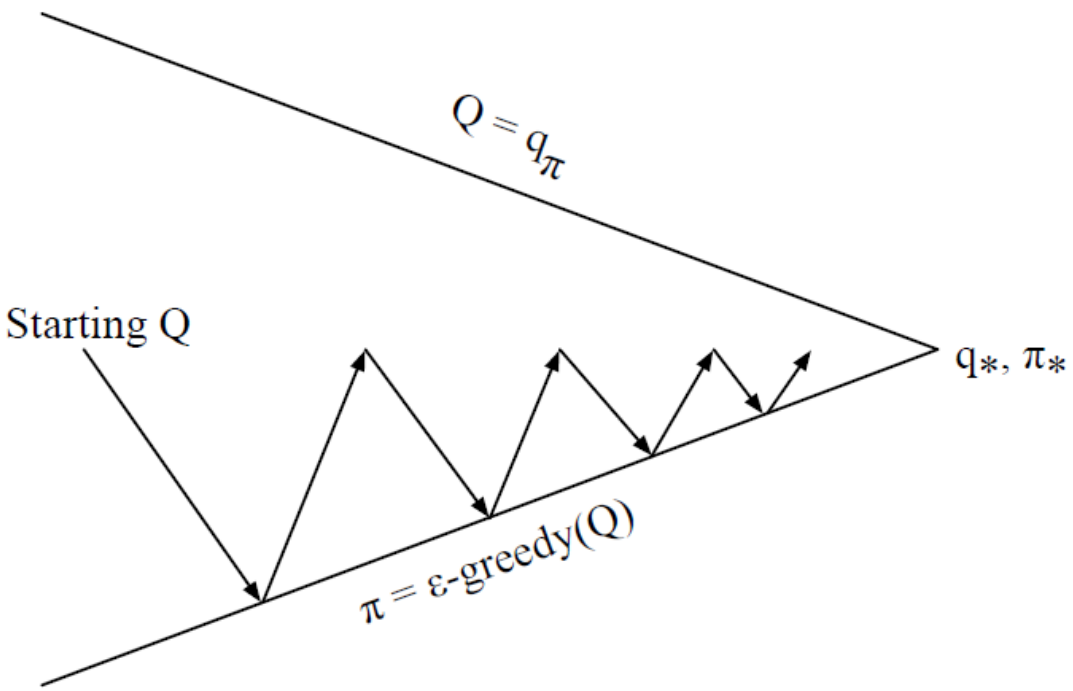
Theorem: If the MDP is deterministic, the approximation $\hat{Q}_r(x,a)$ resulting from the algorithm above converges to the true Q-function $Q(x,a)$ provided that each state-action pair is visited infinitely often as the number of iterations round *r* diverges to infinity.

Q-learning

- Since $V(x) = max_a Q(x, a)$, it should not be surprising that the Q-learning algorithm is similar to Value Function iteration.

- However, there is one 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 to scan the state-action space.

Greedy policy improvements



Note: π denotes the policy function

Q-learning

- Implementation issues:

  - ❑ **Exploration**: Greedy algorithm might prevent exploration (remember the state-action space must be scanned). Simplest remedy is to implement ε-greedy exploration:
    - ✓With probability 1-ε select greedy action
    - ✓With probability ε select action at random

  - ❑ **Learning rate**: Borrow from TD to fine tune learning rate

$$Q(x,a) \longleftarrow Q(x,a) + \alpha \left[U(x,a) + \beta \max_{a'} Q(x',a') - Q(x,a) \right]$$

  - ❑ **Off-policy vs On-policy**: Sarsa use behaviour policy to update the Q-function

$$Q(x,a) \longleftarrow Q(x,a) + \alpha \left[U(x,a) + \beta Q(x',a') - Q(x,a) \right]$$

Large Scale MDPs

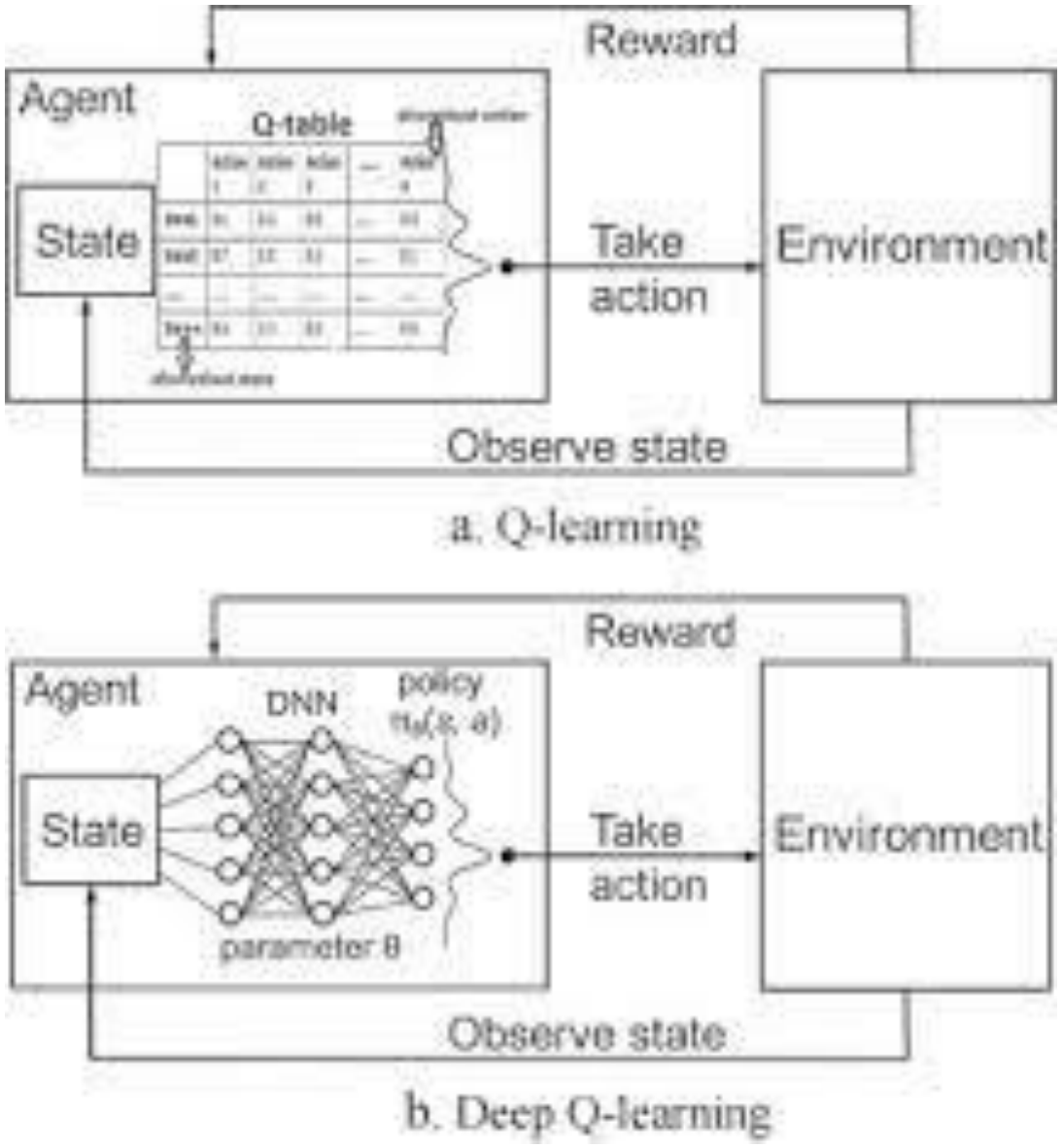
- In large scale problem, the table representation of the Q-function is too big and has to be approximated:

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.



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