Lecture 3 Dynamic Programming

What is Dynamic Programming:

① Divided and Conquer + ② Memory/space swap Time strategy

How to use DP:

Problems that satisfied:

- Has optimal substructure ⇒ ①
- Overlapping sub-problems. ⇒ ②

⇒ MDP satisfies both requirements.

① Bellman equation has recursive structure

② Value function is the expectation of G_t, and G_t will be reused.

Policy Iteration:

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
   ⇒ set to 0 → set to random choose action : $\pi(s,) = \frac{1}{|A|}$
2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))[r + \gamma V(s')]$ ⇒ Bellman Expectation Equation
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ ⇒ end condition: max change < threshold
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
   improvement !
3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$ ⇒ $q(s, \pi(s))$ (greedy strategy)
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

goal of policy iteration is to find the optimal value-table (V^*) and optimal action table / policy (π^*)

why the algorithms above is correct?

① Policy Evaluation:

The algorithm is mean to recursively calculate $V(s)$ using Bellman Expectation Equation (BEE). As the BEE can estimate the value function $V(s)$, thus it's correct.

when it converges, it can estimate the exact value function, but it takes a lot of value iteration. That's why we introduce the θ to determine the accuracy of estimation

② Policy Improvement:

The greedy strategy is mean to perform the Bellman Optimality Equation: $q_\pi^*(s,a) = \max_{a \in A} q_\pi(s,a)$.

For the fact that we have: $V_\pi^*(s) = q_\pi^*(s,a)$, and for $V_\pi^*(s)$, π is the optimal policy. Therefore, this algorithm can improve policy π to optimal policy π^*.

Problem:

Policy iteration updates policy for each Policy Evaluation step. It may take many iterations to break the loop. That makes it relatively slow. ⇒

Improvement 1: introduce θ

Improvement 2: Value Iteration (update policy every iteration)

Value Iteration:

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop: ⇒ Iteration loop
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$ ⇒ update value function for each iteration
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$ ⇒ Get the optimal value function $V^*(s)$, retrieve its policy and it's the optimal policy

why the algorithms above is correct?

The algorithm is mean to calculate the optimal value function $V^*(s)$ through Bellman Optimality Equation.

As the fact that $V^*(s)$'s policy is the optimal policy π^*, thus

$$\pi^* = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma \cdot V^*(s')]$$

Synchronous Dynamic Programming Algorithms

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

- Algorithms are based on state-value function $v_\pi(s)$ or $v_*(s)$
- Complexity $O(mn^2)$ per iteration, for m actions and n states
- Could also apply to action-value function $q_\pi(s,a)$ or $q_*(s,a)$
- Complexity $O(m^2n^2)$ per iteration

Advantages and Drawbacks of DP:

Advantages:

- Fully solves the problem
- Effective for medium-sized (几百万量的states) problems.

Drawbacks

- Need to have advanced knowledge of MDP (R, P)
- Curse of dimensionality (can't have enough memory) in large problems.

⇓

I. Use sample backups method (e.g. MC, TD) to solve ⇒

- Model-free (no knowledge of MDP)
- Constant cost of backup (break the curse)

II. Use approximate DP

Lecture 4 MC, TD for prediction

Monte Carlo

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

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

⇒ from start state s_0 to terminal state S_t

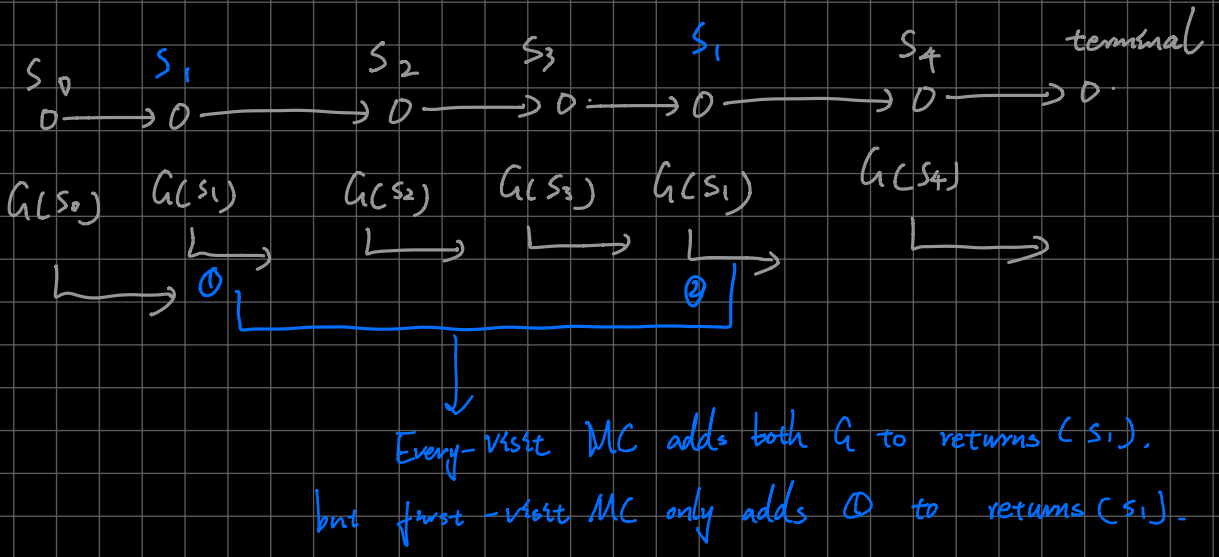
Generate an episode using π

For each state s appearing in the episode:

$G \leftarrow$ return following the first occurrence of s ⇒ every-visit MC append $G_t(s)$ to Returns(s) for each occurrence of s

Append G to $Returns(s)$

$V(s) \leftarrow \text{average}(Returns(s))$ ⇒ Actually this step can be performed outside the loop. 但由于储存 Returns(s) 的空间太大，因此对每个 s 依次计又储存每次优化过的 Returns(s)



Every-visit MC adds both G_t to returns(S_1).

but first-visit MC only adds ① to returns(S_1).

Note: MC use simplest possible idea:

Estimated value = empirical average sample returns

使用经验平均(empirical average)/采样/sampling 估计期望值.

$V(s) = \mathbb{E}(G_t | S_t)$ 为 value function 定义.

Increment MC:

Original: 节省储存空间，提高运算精确度

$$\begin{cases} N(S_t) \leftarrow N(S_t) + 1 \\ V(S_t) \leftarrow \frac{1}{N(S_t)} \cdot \underbrace{[G_t + V(S_t) \cdot N_{old}(S_t)]}_{\text{sum of Returns } (S_t)} = V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \end{cases}$$

Incremental: 这就是增量更新公式的来源.

$$V(S_t) \leftarrow V(S_t) + \underbrace{\alpha}_{\text{constant value}}(G_t - V(S_t))$$

Temporal Difference: (TD(0))

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

Note: TD use the bootstrapping method to learn from incomplete episodes. This makes it more efficient and more general (can deploy in not episodic problems)

Difference between MC and TD:

MC: updates value toward actual return G_t

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

TD: updates value toward estimated return: $R_{t+1} + \gamma \cdot V(S_{t+1})$

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

TD error

= TD target − TD predict

Advantages and Disadvantages:

① TD can learn without the final outcome ⇒

more efficient and more general

② TD has biased, low variance estimate

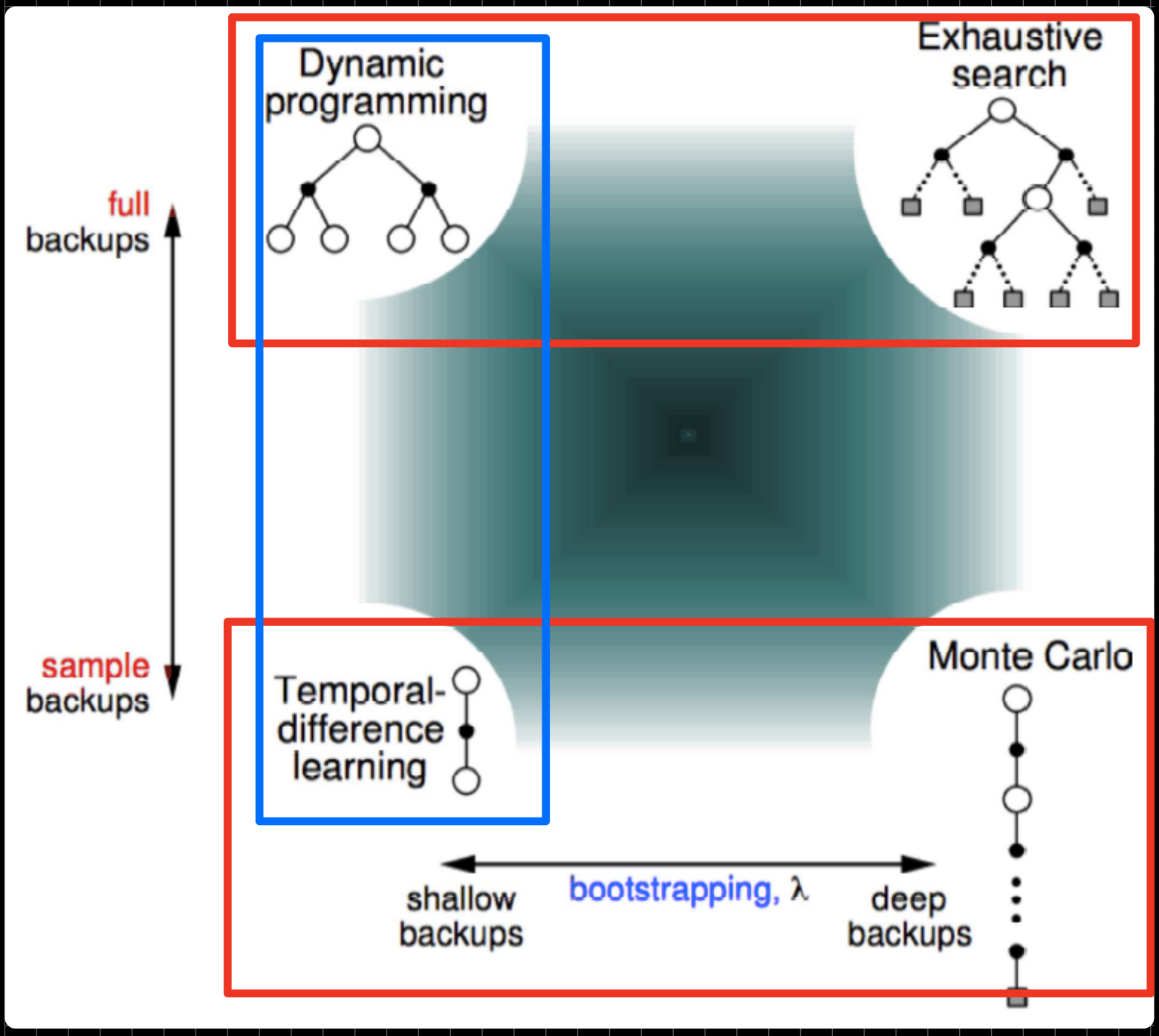
MC has unbiased, high variance estimate

⇒ MC is not very sensitive to initial value,

but TD is sensitive.

③ TD exploits Markov property while MC doesn't. ⇒

- TD is more efficient in MC envs,
- MC is more efficient in non-MC envs,


Dynamic programming
Exhaustive search
full backups
sample backups
Temporal-difference learning
Monte Carlo
shallow backups
bootstrapping, λ
deep backups