

Reinforcement Learning #3

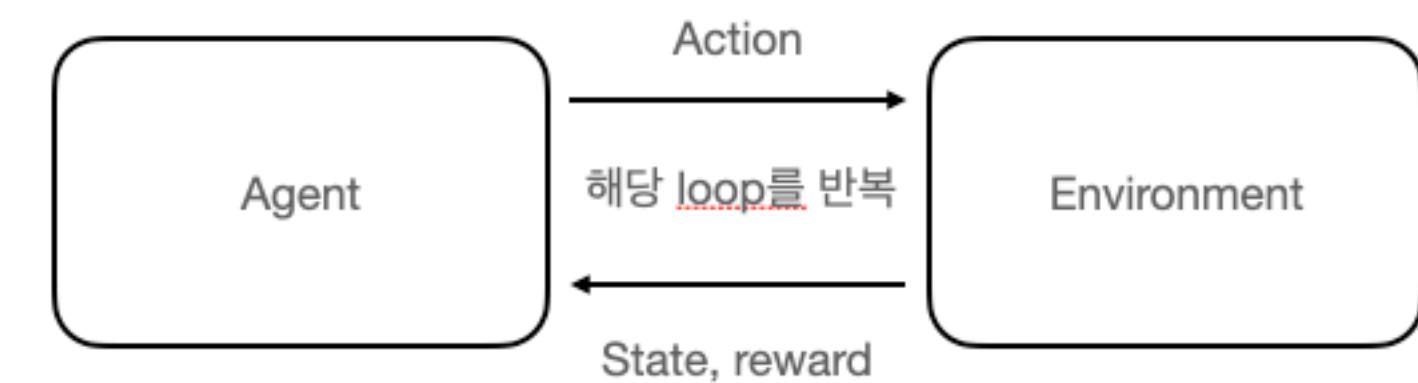
#model-based RL , DP, Policy iteration

정규현

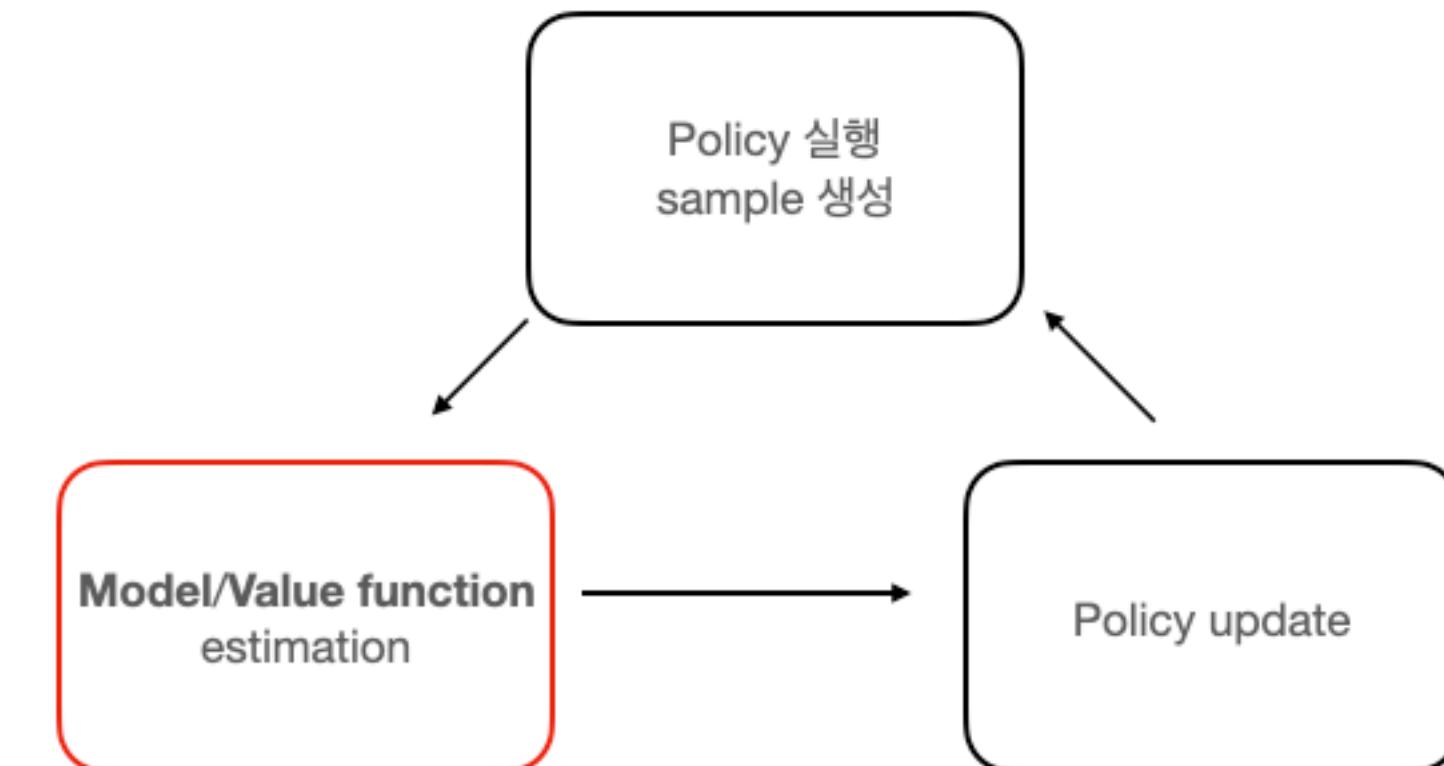
Reinforcement Learning

이전 내용 요약

- 강화학습의 특징
 - 순차적 의사 결정 문제 : 행동을 하고, 환경이 변하고, 또 행동을 하고..
 - Reward hypothesis : reward 누적 합을 최대화 시키는 문제
- Reward
- Environment
- Agent
 - Policy : Agent의 Action을 결정
 - Value function : Agent의 state에 따른 총 reward의 기댓값
 - Model : Agent의 내부에서 환경이 어떻게 변할지 예측



강화학습의 공통적인 iteration

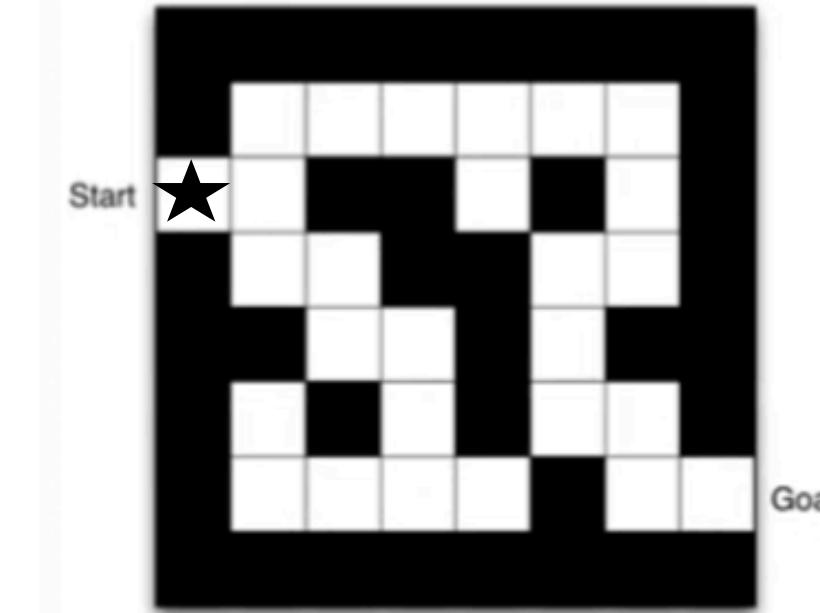


Reinforcement Learning

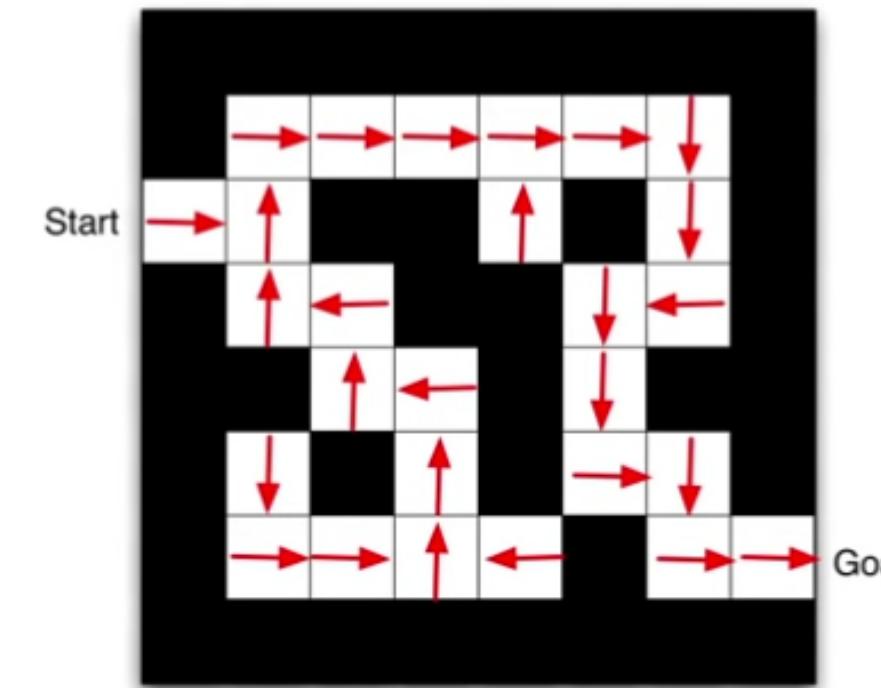
이전 내용 요약

- 강화학습의 특징
 - 순차적 의사 결정 문제 : 행동을 하고, 환경이 변하고, 또 행동을 하고..
 - Reward hypothesis : reward 누적 합을 최대화 시키는 문제
- Reward
- Environment
- Agent
 - Policy : Agent의 Action을 결정
 - Value function : Agent의 state에 따른 총 reward의 기댓값
 - Model : Agent의 내부에서 환경이 어떻게 변할지 예측

목표 : Start로 부터 Goal 지점까지 한 칸씩 이동

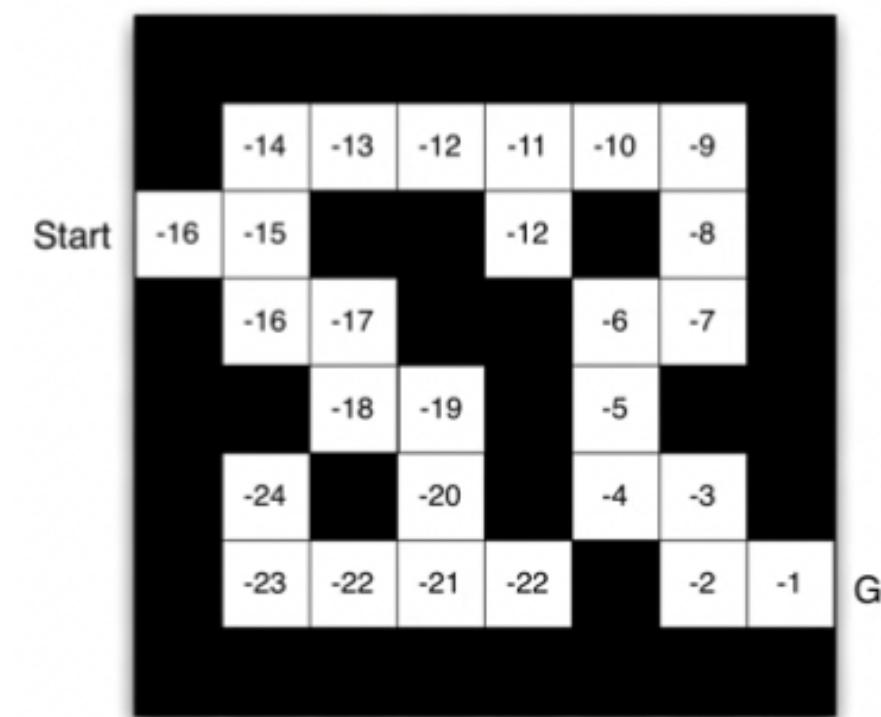


- Rewards: -1 per time-step
- Actions: N, E, S, W
- States: Agent's location



Policy

화살표 : Agent의 최적 Policy



Value function

Value function을 통해 얻은
Agent의 state(위치)에 따른 누적 reward 기대값

Reinforcement Learning

MDP를 풀려면

- 강화학습 문제 = 순차적 의사결정문제!
 - 순차적 의사결정문제 : 행동 -> 변화 -> 행동 -> 변화
- MDP : 순차적 의사결정 문제를 수식화
 - MP -> MRP -> MDP
- 최적의 policy, value function을 찾자
 - 최적의 -> Reward를 최대로 하는
 - 수학적으로 어떻게 최적화?
- Bellman equation을 풀면됨
 - V,Q 사이의 관계
 - V,next V, Q,next Q사이의 관계

- **Value function** in MDP

- **State-Value function** : 현재 state의 가치

- 현재 state부터 기대되는 Return

- **Action-Value function** : 현재 action의 가치

- 현재 Action으로 부터 기대되는 Return

- **Optimal Policy**

- **state-value function**을 최대로 하는 policy

- 현재 state부터 기대되는 Return을 최대로 하는 policy

State-value function : 현재 state S_t 부터 모든 action을 다했을 때 기대되는 Return

$$V(S_t) = \int_{a_t:a_\infty} G_t p(a_t, S_{t+1}, a_{t+1}, \dots | S_t) da_t : a_\infty$$

action-value function : S_t 에서 action a_t 를 했을 때부터 기대되는 Return

$$Q(S_t, a_t) = \int_{S_{t+1}:a_\infty} G_t p(S_{t+1}, a_{t+1}, S_{t+2}, a_{t+2}, \dots | S_t, a_t) ds_{t+1} : a_\infty$$

Optimal Policy

$V(S_t)$ 을 **maximize** 하는 Policy $P(a_t|S_t), P(a_{t+1}|S_{t+1}), P(a_\infty|S_\infty)$

Reinforcement Learning

Bellman 방정식

State-value function : 현재 state S_t 부터 모든 action을 다했을 때 기대되는 Return

$$V(S_t) = \int_{a_t:a_\infty} G_t p(a_t, S_{t+1}, a_{t+1}, \dots | S_t) da_t : a_\infty$$

상태 가치함수: $V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$
(State value function)

현재 t 상태 s 에서 정책 π 를 따른다면 업을 미래의 가치의 감가 총합

action-value function : S_t 에서 action a_t 를 했을 때부터 기대되는 Return

$$Q(S_t, a_t) = \int_{S_{t+1}:a_\infty} G_t p(S_{t+1}, a_{t+1}, S_{t+2}, a_{t+2}, \dots | S_t, a_t) ds_{t+1} : a_\infty$$

행동 가치함수: $Q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$
'상태-행동 가치함수' 라고도 불림
(State-action value function)

Bellman equation V -> next V

$$V(S_t) = \int_{a_t, S_{t+1}} (R_t + \gamma V(S_{t+1})) p(a_t, S_{t+1} | S_t) d_{a_t, S_{t+1}}$$

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s] = R_s^\pi + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^\pi V_\pi(s')$$

Bellman equation Q -> next V

$$Q(S_t, a_t) = \int_{S_{t+1}} (R_t + \gamma V(S_{t+1})) p(S_{t+1} | S_t, a_t) d_{S_{t+1}}$$

$$Q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_\pi(s')$$

Reinforcement Learning

Optimal Policy

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

$V^*(s)$: 존재하는 모든 정책들 중에 모든 상태 s 에서 가장 높은 $V_{\pi}(s)$ 를 만드는 π 를 적용했을 때 얻는 $V_{\pi}(s)$

$Q^*(s, a)$: 존재하는 모든 정책들 중에 모든 상태 / 행동 조합 (s, a) 에서 가장 높은 $Q_{\pi}(s, a)$ 를 만드는 π 를 적용했을 때 얻는 $Q_{\pi}(s, a)$

- 최적 정책 정리(Optimal policy theorem)

- 최적 정책 π^* 가 존재하며, 모든 존재하는 정책 π 에 대하여 $\pi^* \geq \pi$ 를 만족한다. (최적 정책의 존재성)
- 최적 정책들은 π^* 최적 상태 가치함수를 성취한다. 즉, $V_{\pi^*}(s) = V^*(s)$ 이다.
- 최적 정책들은 π^* 최적 행동 가치함수를 성취한다. 즉, $Q_{\pi^*}(s, a) = Q^*(s, a)$ 이다.

$$1. \quad V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a)$$

- 현재의 상태 / 행동 s, a
- 한 스텝 뒤의 상태 / 행동 s', a'

$$2. \quad Q^*(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V^*(s')$$

(1)에 (2)를 대입

$$V_{\pi}(s) = \max_{a \in \mathcal{A}} \left(R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V^*(s') \right)$$

(2)에 (1)를 대입

$$Q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a' \in \mathcal{A}} Q^*(s', a')$$

정책함수 π 간의 대소관계를 다음과 같이 정의 한다.

$$\pi' \geq \pi \leftrightarrow V_{\pi'}(s) \geq V_{\pi}(s), \forall s \in \mathcal{S}$$

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

Reinforcement Learning

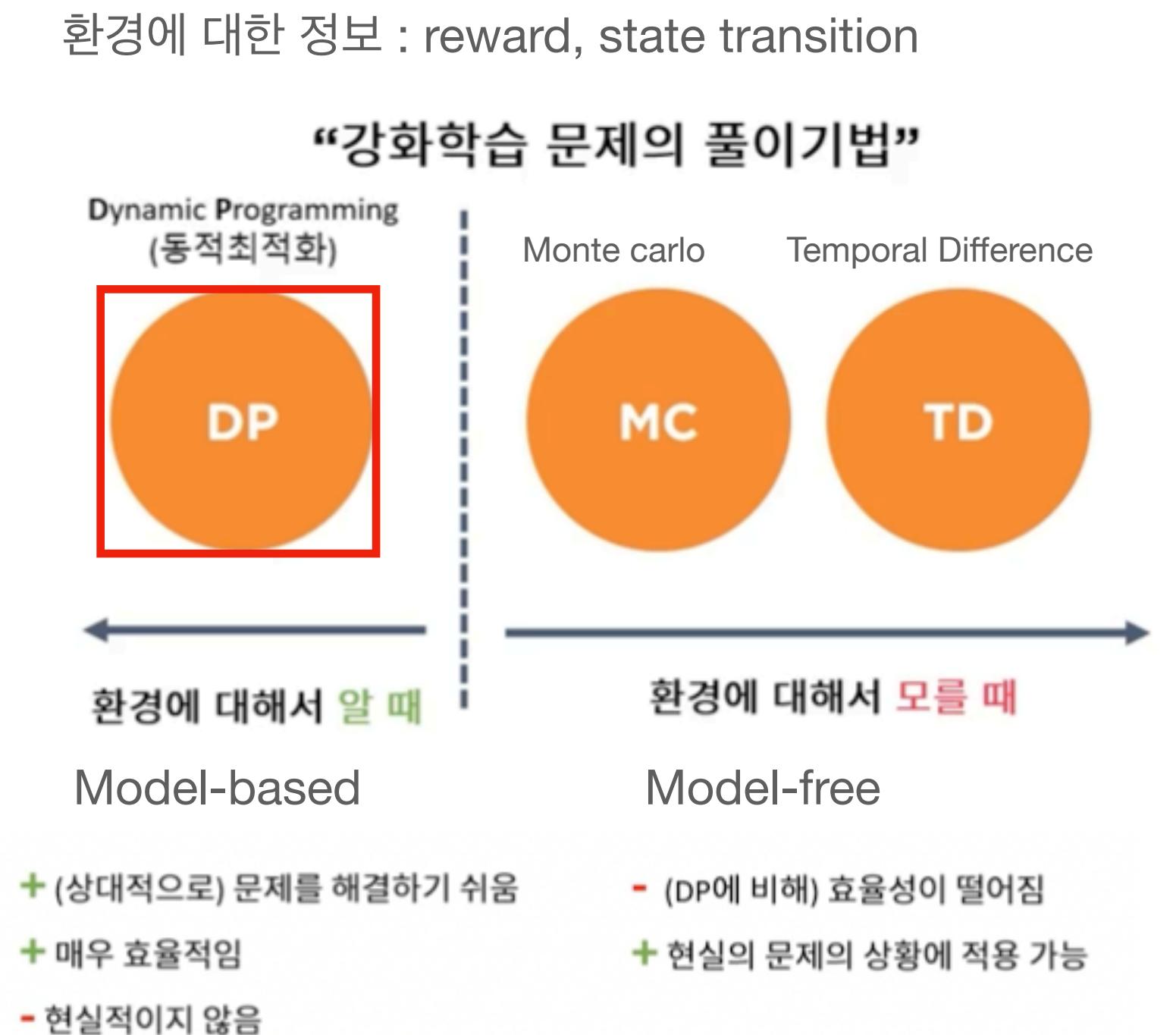
강화학습 문제의 풀이기법

- **Model-based learning** (환경(reward, transition)에 대해 알 때)

- DP (Dynamic Programming)
 - 문제 해결이 쉽고 효율적임 (가능한 모든 다음 state 정보를 이용하여 업데이트)
 - reward, transition을 현실에 적합하게 정의해야 함

- **Model-free learning** (환경에 대해 모를 때)

- MC (Monte-Carlo)
 - Model이 필요없다 -> reward, state 정의가 불필요
 - 현실의 문제의 상황에서 적용이 쉬움
 - 효율성이 떨어짐 (실제로 경험한 state 정보들만으로 업데이트)
 - TD (Temporal Difference)
 - DP의 장점과 MC의 장점을 차용
 - Off-policy : **Q-learning**
 - On-policy : SARSA



Dynamic programming

introduction

- 동적 계획법 (Dynamic programming) - 리처드 E. 벨먼
 - 반복적인 구조를 활용해 문제를 빠르게 해결, 계산량을 효과적으로 줄임

Fibonacci 수열: $F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n$ $F_6 = ??$

해법 1) 그냥 계산한다 (Top-down)

$$F_6 = \textcolor{blue}{F}_5 + \textcolor{brown}{F}_4$$

$$\begin{aligned}\textcolor{blue}{F}_5 &= F_4 + F_3 \\ F_4 &= F_3 + F_2 \\ F_3 &= F_2 + F_1 \\ F_2 &= F_1 + F_0 \\ F_2 &= F_1 + F_0 \\ F_3 &= F_2 + F_1 \\ F_2 &= F_1 + F_0\end{aligned}$$

$$\begin{aligned}\textcolor{brown}{F}_4 &= F_3 + F_2 \\ F_3 &= F_2 + F_1 \\ F_2 &= F_1 + F_0 \\ F_2 &= F_1 + F_0\end{aligned}$$

해법 2) 동적 계획법을 활용
(Bottom-up)

$$F_6 = \textcolor{blue}{F}_5 + \textcolor{brown}{F}_4$$

$$\begin{aligned}F_3 &= F_2 + F_1 \\ \textcolor{brown}{F}_4 &= F_3 + F_2 \\ \textcolor{blue}{F}_5 &= F_4 + F_3\end{aligned}$$

Dynamic programming

Problem

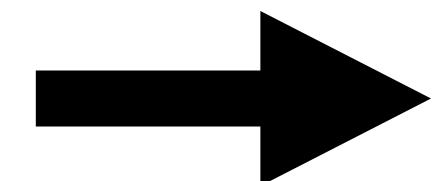
- 동적 계획법으로 해결가능한 문제의 특징

1. 최적 하위구조 (Optimal substructure)

- Principle of optimality
- 큰 문제의 최적값 = 분할한 작은 문제의 최적값

2. 중복 하위문제 (Overlapping problems)

- 큰 문제의 해를 구하기 위해, 작은 문제의 최적 해를 재사용
- 여러 번 재사용하기 때문에 일반적으로 테이블에 저장해둠



해법 2) 동적 계획법을 활용
(Bottom-up)

$$F_6 = F_5 + F_4$$

$$F_3 = F_2 + F_1$$

$$F_4 = F_3 + F_2$$

$$F_5 = F_4 + F_3$$

Bellman equation
Value function

$$\begin{aligned} V &\leftrightarrow Q \\ V_t &\rightarrow V_{t+1} \\ Q_t &\rightarrow Q_{t+1} \end{aligned}$$

Dynamic programming

Policy iteration with DP

벨만 최적방정식

1. $V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a)$
2. $Q^*(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V^*(s')$

(1)에 (2)를 대입

$$V_\pi(s) = \max_{a \in \mathcal{A}} \left(R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V^*(s') \right)$$

- 벨만 최적 방정식을 DP로 해결하자

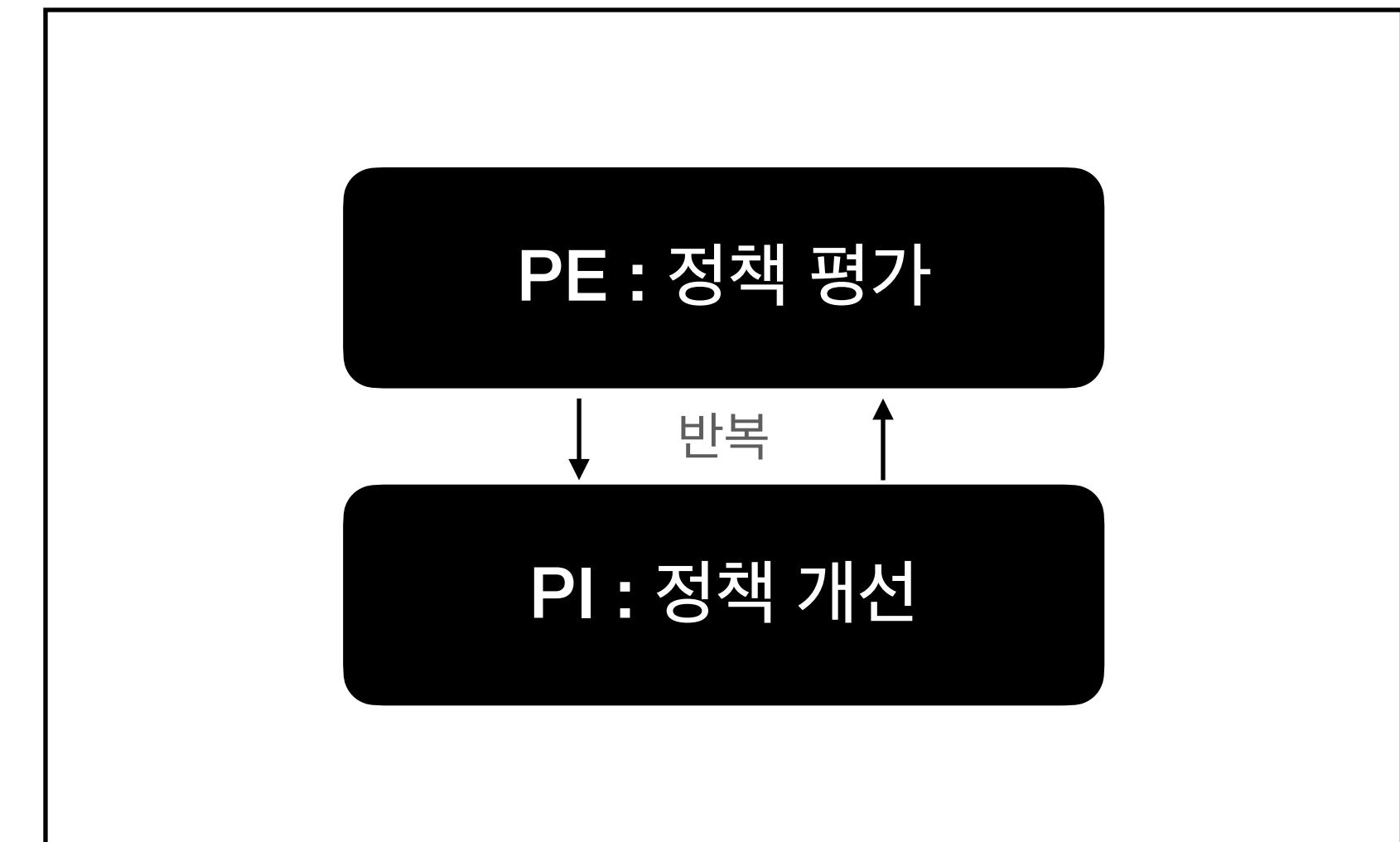
(2)에 (1)를 대입

$$Q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a' \in \mathcal{A}} Q^*(s', a')$$

- 정책 반복 알고리즘(Policy iteration)

- 정책 평가(Policy evaluation : PE)
- 정책 개선(Policy Improvement : PI)

Policy Iteration



Dynamic programming

Policy Evaluation

$$R_s^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) R_s^a$$

- 정책 평가(Policy evaluation : PE)

$$T^\pi(V) \leftarrow R^\pi + \gamma P^\pi V$$

γ 는 감가율, R^π 는 정책 π 에 대한 보상함수, P^π 는 정책 π 에 대한 상태천이 행렬'

Bellman equation $V \rightarrow$ next V

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s] = R_s^\pi + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^\pi V_\pi(s')$$

정책 평가 알고리즘

꼭 0이 아니라 어떤 값으로 시작해도 하나의 값으로 알고리즘의 결과는 수렴함.

초기화 $V_0^\pi(s) \leftarrow 0$ 모든 $s \in \mathcal{S}$

반복 ($t = 0, \dots$):

각 상태에 s 대하여 :

"행렬표현: $V_{t+1}^\pi = R^\pi + \gamma P^\pi V_t^\pi$ "

$$V_{t+1}^\pi(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_t^\pi(s') \right)$$

까지 $\max_{s \in \mathcal{S}} |V_{t+1}^\pi(s) - V_t^\pi(s)| \leq \epsilon$

알고리즘이 유일한 하나의 값으로 수렴하는가? Yes

Dynamic programming

Policy Improvement

- 정책 개선(Policy Improvement : PI)

Bellman equation Q -> next V

$$Q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\pi}(s')$$

정책 개선 알고리즘

입력: 현재 정책 π , 가치함수 $V^\pi(s)$

출력: 개선된 정책 π'

1. $Q^\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^\pi(s')$ 의 관계식을 활용해 $Q^\pi(s, a)$ 계산.

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

Dynamic programming

Policy Iteration

- 정책 반복(Policy Iteration)

정책 반복 (Policy iteration)

입력: 임의의 정책 정책 π

출력: 개선된 정책 π'

1. 정책 평가 (PE) 를 적용해 $V^\pi(s)$ 계산

2. 정책 개선 (PI) 를 적용해 π' 계산

Dynamic programming

Policy Evaluation

```

def get_r_pi(self, policy):
    r_pi = (policy * self.R).sum(axis=-1) # [num. states x 1]
    return r_pi

def get_p_pi(self, policy):
    p_pi = np.einsum("na,anm->nm", policy, self.P) # [num. states x num. states]
    return p_pi

```

초기화 $V_0^\pi(s) \leftarrow 0$ 모든 $s \in \mathcal{S}$

반복 ($t = 0, \dots$):

각 상태에 s 대하여 : “행렬표현: $V_{t+1}^\pi = R^\pi + \gamma P^\pi V_t^\pi$ ”

$$V_{t+1}^\pi(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_t^\pi(s') \right)$$

까지 $\max_{s \in \mathcal{S}} |V_{t+1}^\pi(s) - V_t^\pi(s)| \leq \epsilon$

```

def policy_evaluation(self, policy=None, v_init=None):
    """
    :param policy: policy to evaluate (optional)
    : 정책을 받지 않는다면 기존에 set된 policy를 사용
    :param v_init: initial value 'guesstimation' (optional)
    : 선택적으로 더 좋은 값을 guesstimation으로 넣어줄 수 있지만 0으로 초기화해도됨
    : 더 좋은 값을 안다면 더 빨리 수렴할 수 있음
    :return: v_pi: value function of the input policy
    """

    if policy is None:
        policy = self.policy

    r_pi = self.get_r_pi(policy) # [num. states x 1]
    p_pi = self.get_p_pi(policy) # [num. states x num. states]

    if v_init is None:
        v_old = np.zeros(self.ns)
    else:
        v_old = v_init

    while True:
        # perform Bellman expectation backup : 수식과 동일
        v_new = r_pi + self.gamma * np.matmul(p_pi, v_old)

        # check convergence
        # v_new - v_old의 차이가 충분히 작다면 while문을 break, 그렇지 않다면 반복
        # 차이가 충분히 작을 때 까지 알고리즘을 반복
        bellman_error = np.linalg.norm(v_new - v_old)
        if bellman_error <= self.error_tol:
            break
        else:
            v_old = v_new
    return v_new

```

Dynamic programming

Policy Improvement

정책 개선 알고리즘

입력: 현재 정책 π , 가치함수 $V^\pi(s)$

출력: 개선된 정책 π'

1. $Q^\pi(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V^\pi(s')$ 의 관계식을 활용해 $Q^\pi(s, a)$ 계산.

$$2. \pi'(a|s) = \begin{cases} 1, & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q^\pi(s, a) \\ 0, & \text{otherwise} \end{cases}$$

```
def policy_improvement(self, policy=None, v_pi=None):
    if policy is None:
        policy = self.policy

    if v_pi is None:
        v_pi = self.policy_evaluation(policy)

    # (1) Compute Q_pi(s, a) from V_pi(s)
    r_pi = self.get_r_pi(policy)
    q_pi = r_pi + self.P.dot(v_pi)

    # (2) Greedy improvement
    policy_improved = np.zeros_like(policy) # 개선된 정책의 값이 대부분 0과 특정부분이 1일 것
    policy_improved[np.arange(q_pi.shape[1]), q_pi.argmax(axis=0)] = 1 # q_pi가 가장 큰 값에 대해서만 1설정
    return policy_improved
```

Dynamic programming

Policy Iteration

```
def policy_iteration(self,policy=None) :  
    """  
    임의의 policy를 받아서 사용.  
    info : 학습의 목적으로 과정중에 값들을 확인하기 위해  
    converge : 언제 수렴했는지 확인하기 위해  
    """  
  
    if policy is None :  
        pi_old = self.policy  
    else :  
        pi_old = policy  
  
    info = dict()  
    info['v'] = list()  
    info['pi'] = list()  
    info['converge'] = None # 수렴전 step을 기록  
  
    steps = 0  
    converged = False  
    # 정책 반복 알고리즘  
    while True :  
        v_old = self.policy_evaluation(pi_old) # Policy Evaluation  
        pi_improved = self.policy_improvement(pi_old, v_old) # Policy improvement  
        steps += 1  
  
        info['v'].append(v_old)  
        info['pi'].append(pi_old)  
  
        # check convergence : 수렴확인  
        policy_gap = np.linalg.norm(pi_improved - pi_old)  
  
        if policy_gap <= self.error_tol :  
            if not converged : # 수렴되었으므로 수렴된 step을 기록  
                info['converged'] = steps  
            break  
        else : # 수렴되지 않았으므로 개선된 pi를 pi_old로 대체  
            pi_old = pi_improved  
    return info
```

정책 반복 (Policy iteration)

입력: 임의의 정책 정책 π

출력: 개선된 정책 π'

1. 정책 평가 (PE) 를 적용해 $V^\pi(s)$ 계산
2. 정책 개선 (PI) 를 적용해 π' 계산

Dynamic programming

Policy Iteration

- 정책 개선(Policy Improvement : PI)

정책 반복 (Policy iteration)

입력: 임의의 정책 정책 π

출력: 개선된 정책 π'

1. 정책 평가 (PE) 를 적용해 $V^\pi(s)$ 계산 반증 알고리즘

2. 정책 개선 (PI) 를 적용해 π' 계산 반복 알고리즘

Dynamic programming

Value Iteration

- 가치 반복(Value Iteration : VI)

가치 반복 (Value iteration)

꼭 0 이 아니라 어떤 값으로 시작해도 하나의 값으로 알고리즘의 결과는 수렴함.

입력: 임의의 가치 함수 $V_0(s) \leftarrow 0$ 모든 $s \in \mathcal{S}$

출력: 최적 가치 함수 $V^*(s)$

반복: ($k = 0, \dots, n$):

<Bellman optimal backup>

- 모든 상태 s 에 대해서,
$$V_{k+1}(s) = \max_{a \in \mathcal{A}} (R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_k(s'))$$
 적용

- 모든 상태 s 에 대해서, $V_{k+1}(s) \sim V_k(s)$ 이면, 반복문 탈출

반환: $V^*(s) \leftarrow V_{k+1}$

Dynamic programming

Value Iteration

$$T^*(V) \stackrel{\text{def}}{=} \max_{a \in \mathcal{A}} (R^a + \gamma P^a V)$$

```
# 가치반복 알고리즘은 외적으로 policy를 input으로 받지는 않음  
# 가치 반복 알고리즘은 최적 가치를 계산해주는 알고리즘  
# 모델, 최적가치를 알고 있으면 모델을 통해 q*를 알게 되고  
# q*로 greedy policy improvement를 적용하게 되면 최적 정책함수를 계산할 수 있음
```

```
def value_iteration(self, v_init=None, compute_pi=False):  
    """  
    param v_init :  
    param compute_pi :  
    return :  
  
    """  
    if v_init is not None:  
        v_old = v_init  
    else:  
        v_init = np.zeros(self.ns)  
  
    info = dict()  
    info['v'] = list()  
    info['pi'] = list()  
    info['converge'] = None  
  
    steps = 0  
    converged = False  
  
    while True:  
        # Bellman optimality backup  
        v_improved = (self.R.T + self.P.dot(v_old)).max(axis=0) # [num actions x num states] : action의 dimension이 0이므로 axis=0  
        info['v'].append(v_improved)  
  
        if compute_pi:  
            # compute policy from v  
            # compute v->q  
            q_pi = (self.R.T + self.P.dot(v_improved)) # [num actions x num states]  
  
            # Construct greedy policy : 가장 높은 q값을 가진 action을 1, 나머지 0  
            pi = np.zeros_like(self.policy) # policy안에 모두 0을 넣음 [num states x num actions]?  
            pi[np.arange(q_pi.shape[1]), q_pi.argmax(axis=0)] = 1 # action중 q값이 가장 큰 index에만 1  
            info['pi'].append(pi)  
        steps += 1  
  
        # check convergence  
        value_gap = np.linalg.norm(v_improved - v_old)  
  
        if value_gap <= self.error_tol:  
            if not converged:  
                info['converge'] = steps  
            break  
        else:  
            v_old = v_improved  
  
    return info
```

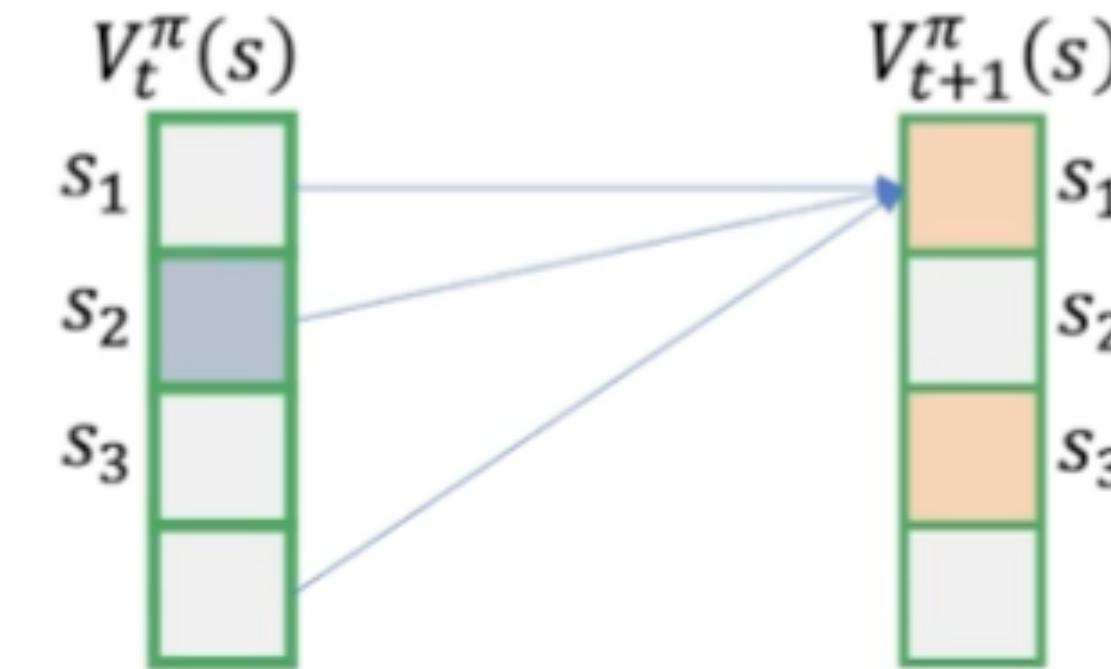
- Bellman optimality backup operator

Dynamic programming

더 나은 DP 활용법?

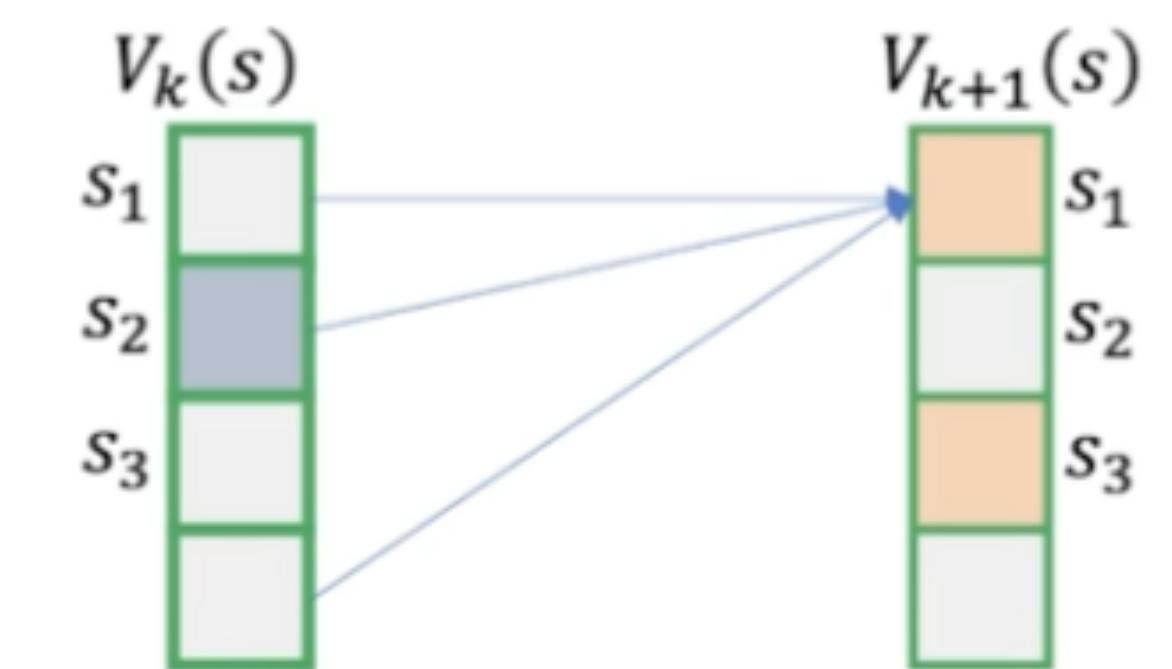
- 앞서 소개한 알고리즘들
 - 전체 상태에 대한 value function을 계속해서 업데이트 함.
 - 모든 상태에 대해 업데이트

<정책 평가: PE>



s_n

<가치 반복: VI>



s_n

s_n

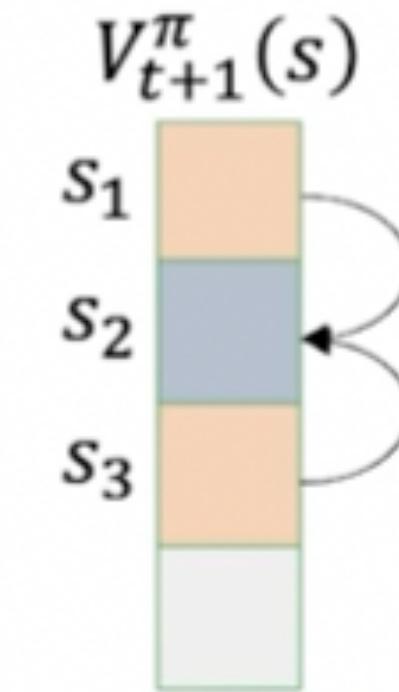
모두 V를 업데이트 하기 위해서는 기존의 V값들을 모두 알고있어야 함
-> 메모리에 전체 상태 개수의 두 배의 테이블이 필요
-> 연산량이 많고, 학습속도가 느려지게 됨

Dynamic programming

Asynchronous DP

- 비동기적 DP알고리즘
 - 모든 상태에 대해서 항상 업데이트를 하지 않음
 - 현재 알고있는 값을 활용해 곧바로 새로운 값을 만들어 냄
- 대표적 Asynchronous DP
 - In-place
 - Prioritized sweeping
 - Parital sweeping

“In place” 연산



```
def compute_q_from_v(self, value):
    return self.R.T + self.gamma * self.P.dot(value) # [num. actions x num. states]
```

Asynchronous DP

In Place VI(Full Sweeping)

하나의 Value function만 가지고 VI를 수행

```
def compute_q_from_v(self, value):
    return self.R.T + self.gamma * self.P.dot(value) # [num. actions x num. states]
```

In-place 가치 반복 (full-sweeping)

입력: 임의의 가치 함수 $V(s) \leftarrow 0$ 모든 $s \in \mathcal{S}$

출력: 최적 가치 함수 $V^*(s)$

반복: ($k = 0, \dots$):

<Bellman optimal backup>

- 모든 상태 s 에 대해서, $V(s) = \max_{a \in \mathcal{A}} (R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V(s'))$ 적용
- 모든 상태 s 에 대해서, $V(s)$ 수렴하면, 반복문 탈출

$V(s)$ 만 필요

반환: $V^*(s) \leftarrow V(s)$

가치 반복 (Value iteration)

입력: 임의의 가치 함수 $V_0(s) \leftarrow 0$ 모든 $s \in \mathcal{S}$

출력: 최적 가치 함수 $V^*(s)$

New $V(s)$, old $V(s)$ 필요

반복: ($k = 0, \dots$):

<Bellman optimal backup>

- 모든 상태 s 에 대해서, $V_{k+1}(s) = \max_{a \in \mathcal{A}} (R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_k(s'))$ 적용
- 모든 상태 s 에 대해서, $V_{k+1}(s) \sim V_k(s)$ 이면, 반복문 탈출

반환: $V^*(s) \leftarrow V_{k+1}$

```
def in_place_vi(self, v_init=None) :
    """
    param v_init : 기존에 가진 v값
    """
    if v_init is not None :
        value = v_init
    else :
        value = np.zeros(self.ns) # 임의값인 0으로 시작
```

tracking을 위한 dict 생성

```
info = dict()
info['v'] = list()
info['pi'] = list()
info['converge'] = None
info['gap'] = list()
```

steps = 0

```
# inplace value iteration
while True :
```

값을 저장하기 위한 코드

```
delta_v = 0
info['v'].append(value)
pi = self.construct_policy_from_v(value)
info['pi'].append(pi)
```

full sweeping을 진행

어느 순서로 진행하느냐에 따라 속도는 차이가 있지만 수렴하는 점은 결국 동일해짐

```
for s in range(self.ns) : # ns : 0~24
    qs = self.compute_q_from_v(value)[:, s]
    v = qs.max(axis = 0)
```

```
delta_v += np.linalg.norm(value[s] - v) # 수렴확인 : 거리를 계산.
value[s] = v # 새로운 v를 할당
```

info['gap'].append(delta_v)

```
if delta_v < self.error_tol : # delta_v가 충분히 줄었다면
    info['converge'] = steps # step 저장
    break
```

```
else :
    steps += 1
```

return info

Asynchronous DP

Prioritized Sweeping VI

우선순위를 통해 value 값을 update

- **Bellman error**

$$e(s) = \left| \max_{a \in \mathcal{A}} (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V(s')) - V(s) \right|$$

- update한 값과 기존 V값의 차이
- PriorityQueue()를 활용
- update할 정도가 큰 순서로 update

```
def prioritized_sweeping_vi(self, v_init=None) :  
    """  
    param v_init : v 초기값 설정  
    """  
    if v_init is not None :  
        value = v_init  
    else :  
        value = np.zeros(self.ns)  
  
    # 기록하기 위한 dict  
    info = dict()  
    info['v'] = list()  
    info['pi'] = list()  
    info['converge'] = None  
    info['gap'] = list()  
  
    step = 0  
    while True :  
        # compute Bellman error : 기존 value - 새롭게 추산한 value  
        bellman_errors = value - (self.R.T + self.P.dot(value)).max(axis=0)  
        # update 순서를 위한 Index  
        state_indices = range(self.ns)  
  
        # Priority Queue() : python에서 우선순위(값이 낮은 것부터) 값을 뺏어주는 Queue  
        # priority queue put (우선순위, 뺏어낼 값) 형태로 입력  
        # priority_queue : (bellman errors , state index)  
  
        priority_queue = PriorityQueue()  
        for bellman_error, s_idx in zip(bellman_errors, state_indices) :  
            priority_queue.put((-bellman_error, s_idx)) # -bellman error가 우선순위 (값이 낮은것부터 이므로 -)  
  
        # value -> policy  
        info['v'].append(value)  
        pi = self.construct_policy_from_v(value)  
        info['pi'].append(pi)  
        delta_v = 0  
  
        while not priority_queue.empty() : # queue가 빌 때까지 반복. full sweeping을 가정했기 때문.  
            be, s = priority_queue.get() # (Bellman error, state index) 우선순위대로 받음  
            qs = self.compute_q_from_v(value)[:,s] # num action x num state  
            v = qs.max(axis=0) # get max value along the action  
  
            delta_v += np.linalg.norm(value[s] - v) # 기존에 있던 v - update된 v, 누적오차를 확인하기 위해 사용  
            value[s] = v # s state의 value값 업데이트  
  
            info['gap'].append(delta_v)  
  
            if delta_v < self.error_tol :  
                info['converge'] = steps  
                break  
  
            else :  
                steps += 1  
  
    return info
```

Asynchronous DP

Partial Sweeping

부분적으로 Sweeping

- **Full Sweeping**

- 매 value iteration마다 모든 state, action에 대해서 업데이트
- value function을 구하기 위해서 모든 가능한 state, action을 알고 있어야 함
- 매번 엄청난 수의 sample이 필요함

- **Partial Sweeping**

- “전체 value iteration 동안 모든 s 가 업데이트” 된다면 매 iteration마다 모든 s 를 업데이트 하지 않아도 수렴한다.
- 대부분의 RL환경에서 사용한다.
- iteration을 늘려 많은 sample을 가지면 결국 최적 value function으로 수렴한다.

```
def in_place_vi_partial_update(self,
                               v_init=None,
                               updated_prob=0.5,
                               vi_iters: int =100) :
    """
    param v_init : v 초기값
    param v_updated_prob : update할 state의 비율
    param vi_iters : 반복 횟수
    """

    if v_init is not None :
        value = v_init
    else :
        value = np.zeros(self.ns)

    # 이번에는 반복횟수를 설정했으므로 converge 필요 x
    info = dict()
    info['v'] = list()
    info['pi'] = list()
    info['gap'] = list()

    # 횟수가 정해져 있으므로 for문 사용
    # partial sweeping
    for steps in range(vi_iters) :
        delta_v = 0

        for s in range(self.ns) :
            # 임의의 확률로 sampling. 1이 나올 확률 : update_prob
            # 1이 나오지 않으면 다음 for loop으로
            perform_update = np.random.binomial(size=1, n=1, p=update_prob)
            if not perform_update :
                continue

            # 1이 나온 경우에 v update
            qs = self.compute_q_from_v(value)[:,s]
            v = qs.max(axis=0)

            delta_v += np.linalg.norm(value[s] - v)
            value[s] = v

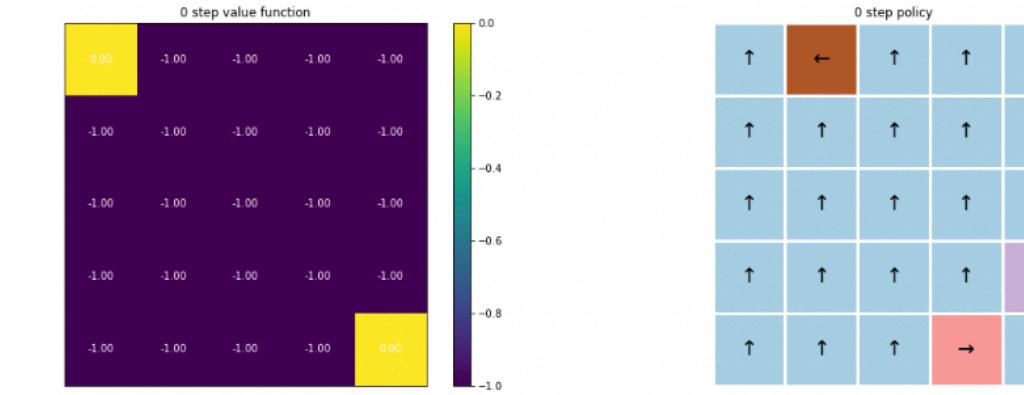
        info['gap'].append(delta_v)
        info['v'].append(value.copy())
        pi = self.construct_policy_from_v(value)
        info['pi'].append(pi)

    return info
```

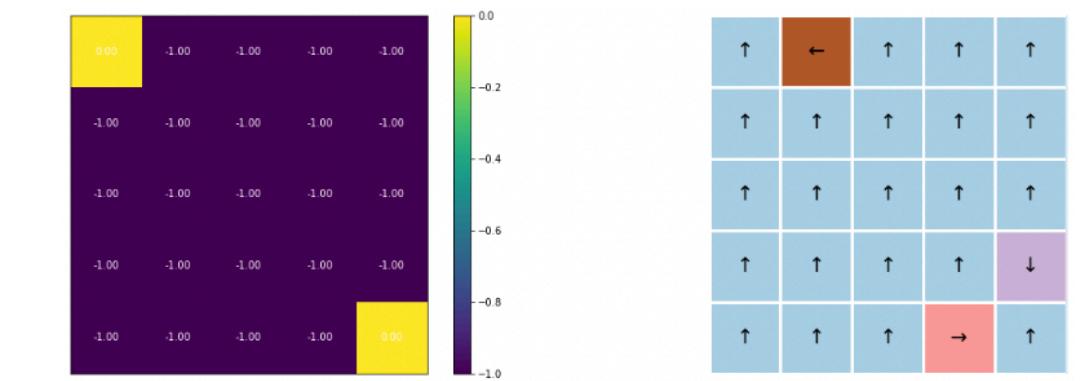
Asynchronous DP Visualization

수렴해 얻은 최적해는 결국 동일하다.

In place vi (full sweeping)



Prioritized sweeping vi



The figure consists of two side-by-side heatmaps. The left heatmap, titled "1 step value function", shows a 5x5 grid where each cell's value ranges from -2.00 (dark purple) to 0.00 (yellow). The right heatmap, titled "1 step policy", shows the same 5x5 grid but with arrows indicating the direction of the policy: up (↑), down (↓), left (←), right (→), or stay (↔).

The figure consists of two heatmaps. The left heatmap shows energy as a function of two parameters, with a color scale ranging from -2.00 (dark purple) to 0.00 (yellow). The right heatmap shows the effect of varying one parameter while keeping others fixed at their minimum values, with a color scale ranging from -2.00 (dark purple) to 0.00 (yellow).

The figure consists of two side-by-side heatmaps. The left heatmap is titled "2 step value function" and has a color scale from -3.0 (dark purple) to 0.0 (yellow). The right heatmap is titled "2 step policy" and shows arrows indicating actions: up (↑), down (↓), left (←), right (→), and stay (↔).

The figure consists of two parts. The left part is a heatmap of a 5x5 matrix. The values are as follows:

	0.00	-1.00	-2.00	-3.00	-3.00
0.00	0.00	-1.00	-2.00	-3.00	-3.00
-1.00	-1.00	-2.00	-3.00	-3.00	-3.00
-2.00	-2.00	-3.00	-3.00	-3.00	-2.00
-3.00	-3.00	-3.00	-3.00	-2.00	-1.00
-3.00	-3.00	-3.00	-2.00	-1.00	0.00

The right part is a 5x5 grid of symbols representing the spin state of each matrix element. The symbols are: ↑, ←, ←, ←, ↑; ↑, ↑, ↑, ↑, ↓; ↑, ↑, ↑, →, ↓; ↑, ↑, →, →, ↓; ↑, →, →, →, ↑.

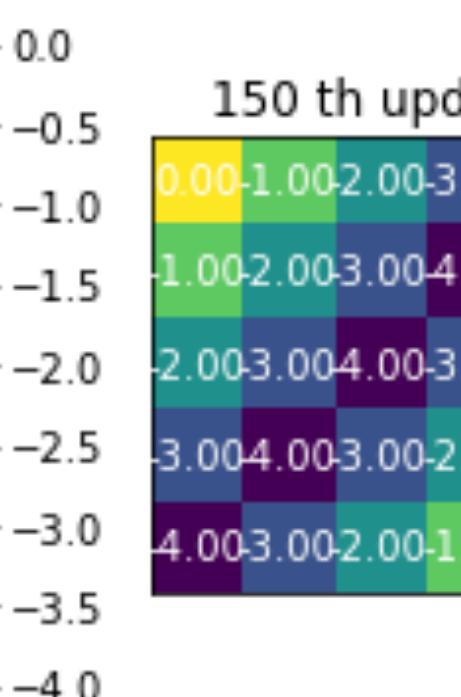
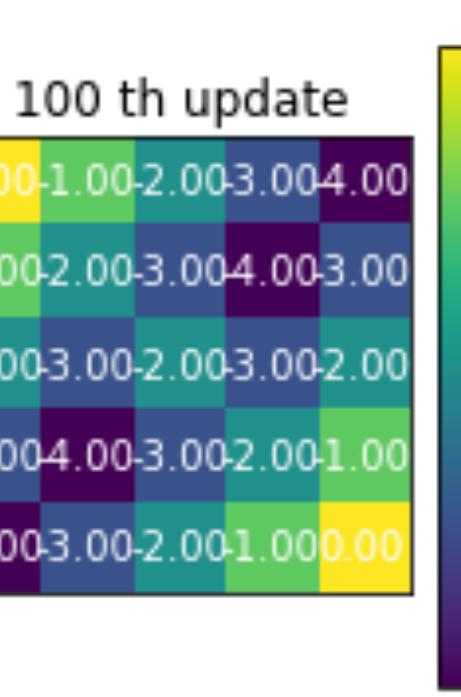
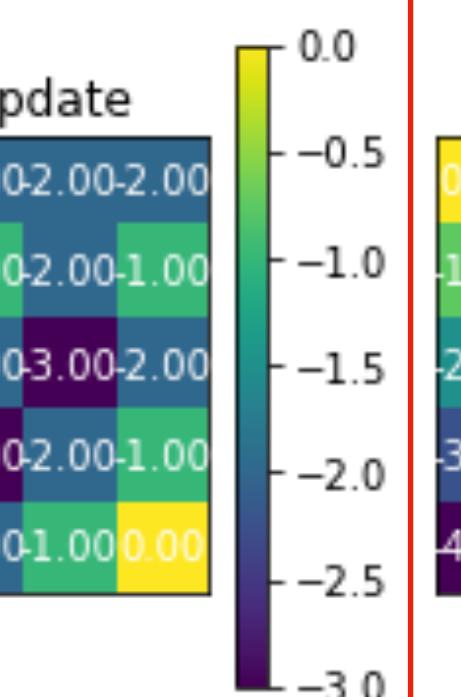
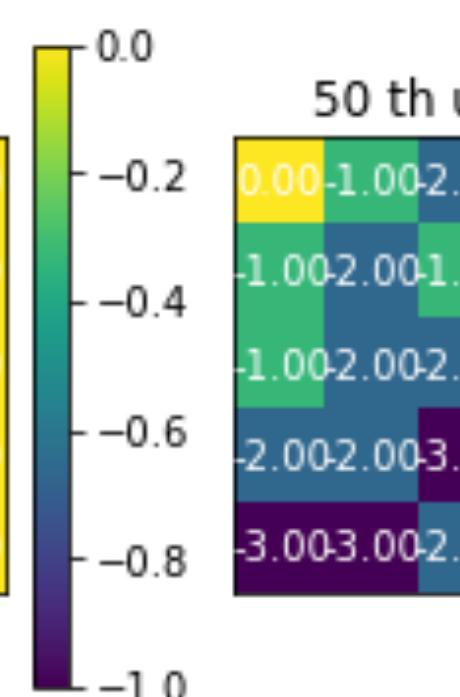
The figure displays two 5x5 grids representing different aspects of a reinforcement learning task over three time steps. The left grid, titled '3 step value function', uses a color scale from -3.5 (dark purple) to 0.0 (yellow) to represent numerical values. The right grid, titled '3 step policy', uses arrows to indicate the direction of action: up (↑), down (↓), left (←), and right (→).

The figure consists of two side-by-side diagrams. On the left is a 5x5 heatmap where each cell contains a numerical value representing energy. The color scale ranges from -4.00 (dark purple) to 0.0 (yellow). The values are as follows:

	1	2	3	4	5
1	0.00	-1.00	-2.00	-3.00	-4.00
2	-1.00	-2.00	-3.00	-4.00	-3.00
3	-2.00	-3.00	-4.00	-3.00	-2.00
4	-3.00	-4.00	-3.00	-2.00	-1.00
5	4.00	-3.00	-2.00	-1.00	0.00

On the right is a 5x5 grid representing a spin configuration. Each cell contains a symbol indicating the spin direction: \uparrow , \downarrow , \leftarrow , or \rightarrow . The pattern shows alternating spins along the rows and columns.

In place vi (Partial sweeping)

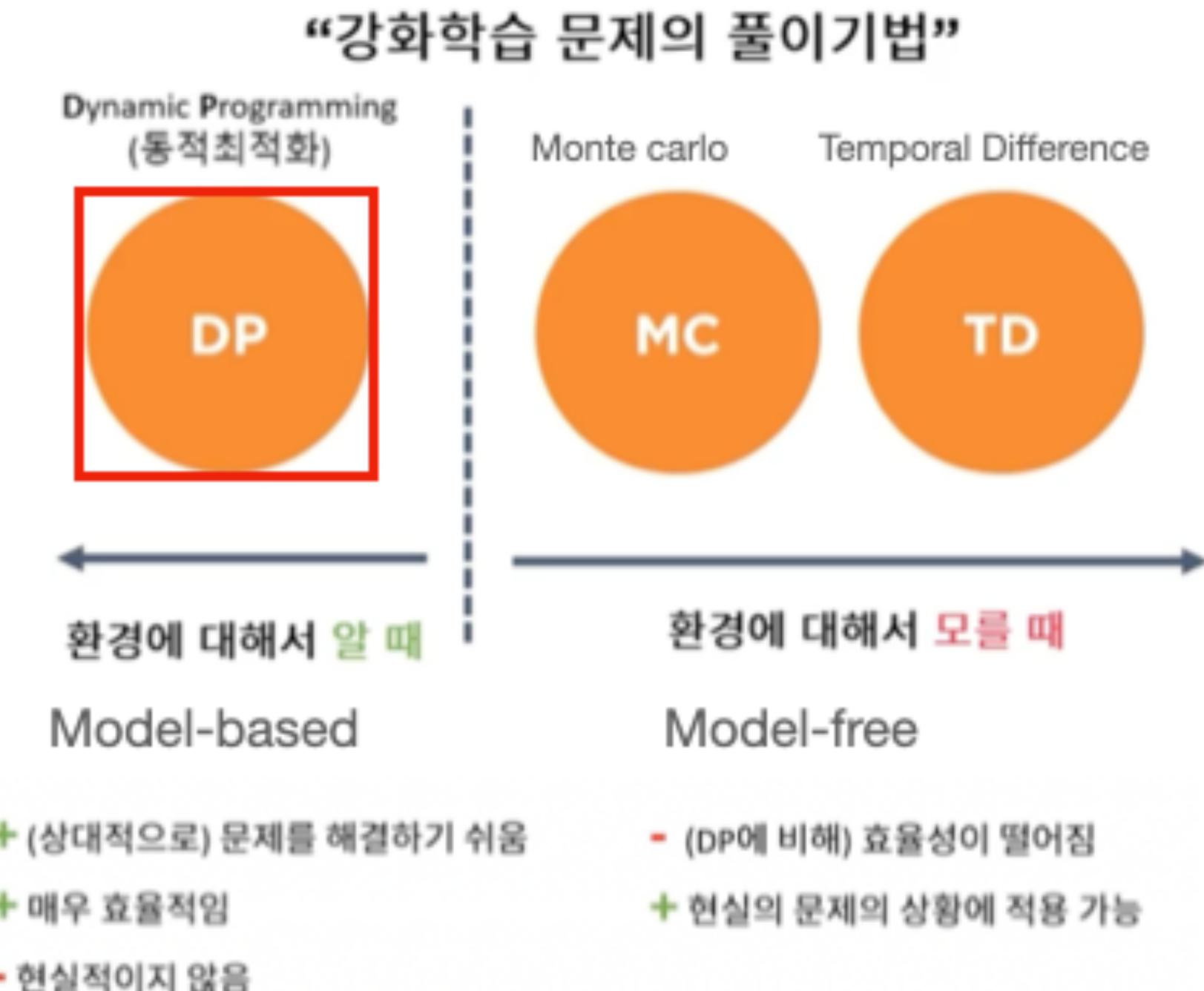


Full sweeping에 비해 state가 부분부분 업데이트!

Reinforcement Learning

Dynamic Programming

- 환경, 상태 천이함수에 대해서 안다는 것
 - action을 통해 Reward, state에 대해 안다는 것
 - 그 값을 통해 value function을 구한다.
 - Value function : 해당 state에서 기대되는 Return
 - 그 값을 통해 optimal policy를 구한다.
 - 해당 state에서 기대되는 Return이 가장 큰 Action



Reinforcement Learning

Dynamic Programming 왜 중요?

- 강화학습 프로세스
 - 최적정책, 상태에 맞는 최적 행동찾기
 - 최적 기준 : value function
 - 계산을 통해 수렴할때 까지 계속해서 업데이트
 - 해당 과정을 DP로 효율적으로 계산

