Reinforcement Learning #4

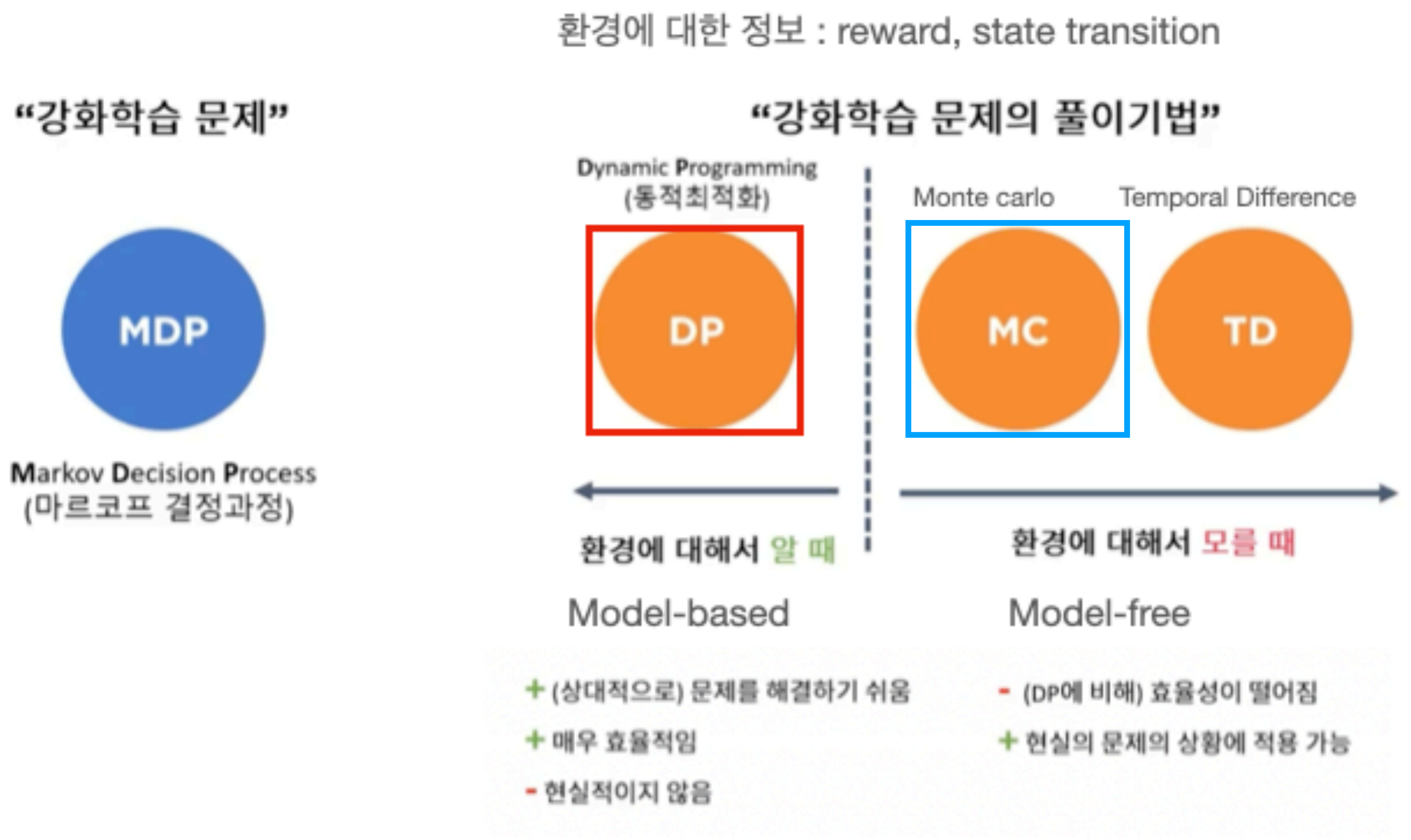
#MonteCarlo

정규헌

Reinforcement Learning

강화학습 문제

- 최적 정책를 찾는 것!
- **Model based : DP, Asynchronous DP**
    - **알고 있는 값**들을 활용해 특정 공식을 수렴할 때까지 반복해 최적해를 찾음
- **Model free : MC, TD**
    - 현실에서는 환경에 대해서 모를때가 많음
    - **MC** : 데이터를 활용해서 최적해를 찾음
    - **TD** : 알고있는 일부 경험과 함께 데이터를 활용



Reinforcement Learning
강화학습 문제

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

알고있는 값들을 활용, 수렴할때 까지 DP를 활용해

일반화된 정책 반복 (Generalized Policy iteration)

입력: 임의의 정책 정책 π

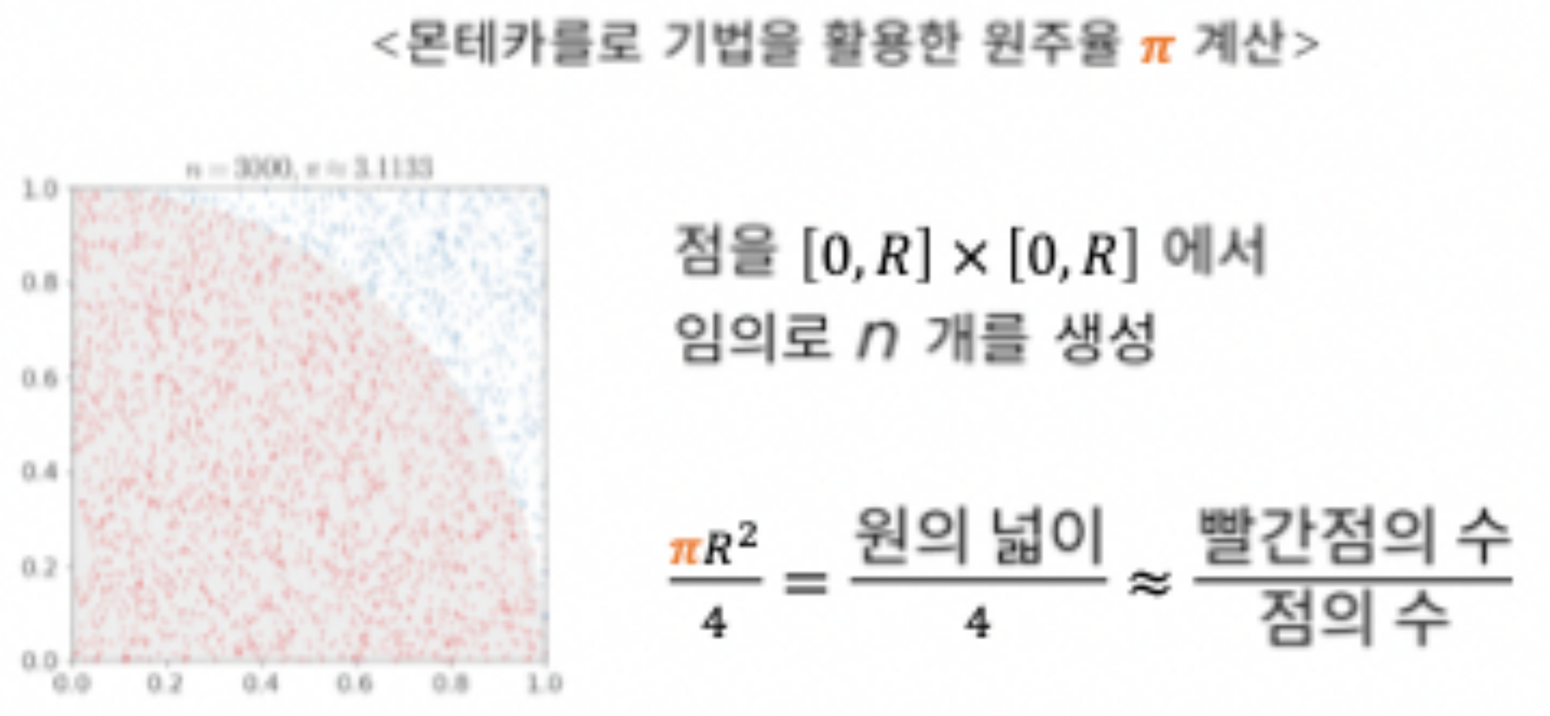
출력: 개선된 정책 π'

1. 임의의 방식을 활용해 적용해 $V^{\pi}(s)$ 계산
2. 임의의 방식을 활용해 적용해 π' 계산 ($\pi' \geq \pi$ 를 만족)

Monte Carlo

Monte Carlo?

- Monte Carlo : 프랑스의 도시
    - 카지노,도박의 도시로 유명
- **Monte Carlo** 기법
    - **계산하기 어려운 값**을 수 많은 확률 시행을 거쳐 추산하는 기법
    - 1930. 중성자의 특성 연구에 처음으로 사용 (엔리코 페르미)
    - 많은 공학분야에서 해당 기법을 활용



Monte Carlo

State Value function V by Monte Carlo method

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

$$G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-1} R_T$$

- Value function : 현재 상태에서 얻을수 있는 Gt(Return)의 기댓값
  - Gt를 여러번 시뮬레이션을 해서 sample들의 평균을 계산해서 Value function을 추산
- **Unbiased Estimate(불편추정량)**
  - sample을 뽑아 통계량을 계산해서 모집단의 모수를 추정함
  - sample을 평균을 내서 추정량(Estimate)로 사용하기 위해서는 sample들의 편향이 없어야함.
  - MC를 불편추정기법이라고도 함
  - 시뮬레이션의 횟수가 늘어나면 추정치의 분산이 줄어듬.
  - 횟수가 적을수록 추정치의 불확실성이 증가

Monte Carlo

First-visit(최초 방문) Monte Carlo policy evaluation

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

$$G_t \overset{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T$$

데이터 (정책 π 을 따라서 생성): $s_t \in \mathcal{S} = \{s^1, s^2, s^3\}, a_t \in \mathcal{A} = \{a^1, a^2, a^3\}$

Episode 1: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \dots; s_T, a_T, r_T$

Episode 2: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \dots; s_T, a_T, r_T$

Episode 3: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \dots; s_T, a_T, r_T$

state1, action1,reward1, state2, action2,reward2 ......

(상태, 행동, 보상) 정책 π 을 따라서 생성

$\gamma = 1$

Episode 1 : $(\boxed{s^1}, a^2, 1); (s^3, a^1, 5); (s^2, a^3, 3), (s^1, a^3, 10), (s^2, a^2, 2)$ $G_t(s^1)$ =1+5+3+10+2 = **21**

Episode 2 : $(s^3, a^1, 5); (s^2, a^2, 2); (\boxed{s^1}, a^2, 1); (s^2, a^3, 3), (s^1, a^3, 10)$ $G_t(s^1)$ = 1+3+10 = **14**

Episode 3 : $(s^2, a^3, 3); (\boxed{s^1}, a^2, 1); (s^3, a^1, 5);(s^1, a^3, 10), (s^2, a^2, 2)$ $G_t(s^1)$ =1+5+10+2 = **18**

원하는 state에 처음 방문한 시점부터 생성된 모든 return을 고려

$$V_\pi(s^1) = \mathbb{E}_\pi[G_t | s_t = s^1] \approx \frac{1}{N}\sum_{n=1}^{N} G_n^\pi(s^1)$$

$V_\pi(s^1)$ **= 모든 Episode에 대한 리턴의 산술평균 = (21+14+18) / 3 = 17.66**

Monte Carlo

Every-visit(모든 방문) Monte Carlo policy evaluation

$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$

$G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-1} R_T$

데이터 (정책 π 을 따라서 생성): $s_t \in \mathcal{S} = \{s^1, s^2, s^3\}, a_t \in \mathcal{A} = \{a^1, a^2, a^3\}$

Episode 1: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \ldots ; s_T, a_T, r_T$
Episode 2: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \ldots ; s_T, a_T, r_T$
Episode 3: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \ldots ; s_T, a_T, r_T$

state1, action1,reward1, state2, action2,reward2 ......

(상태, 행동, 보상) 정책 π 을 따라서 생성

$\gamma = 1$

Episode 1 : $(\boxed{s^1}, a^2, 1); (s^3, a^1, 5); (s^2, a^3, 3), (\boxed{s^1}, a^3, 10), (s^2, a^2, 2)$ $G_t(s^1)$ =1+5+3+10+2 = **21** $G_t(s^1)$ =10+2 = **12**
Episode 2 : $(s^3, a^1, 5); (s^2, a^2, 2); (\boxed{s^1}, a^2, 1); (s^2, a^3, 3), (\boxed{s^1}, a^3, 10)$ $G_t(s^1)$ = 1+3+10 = **14** $G_t(s^1)$ = 10 = **10**
Episode 3 : $(s^2, a^3, 3); (\boxed{s^1}, a^2, 1); (s^3, a^1, 5); (\boxed{s^1}, a^3, 10), (s^2, a^2, 2)$ $G_t(s^1)$ =1+5+10+2 = **18** $G_t(s^1)$ =10+2 = **12**

원하는 state에 처음 방문한 시점부터 생성된 모든 return을 고려

$$V_\pi(s^1) = \mathbb{E}_\pi[G_t | s_t = s^1] \approx \frac{1}{N}\sum_{n=1}^{N} G_n^\pi(s^1)$$

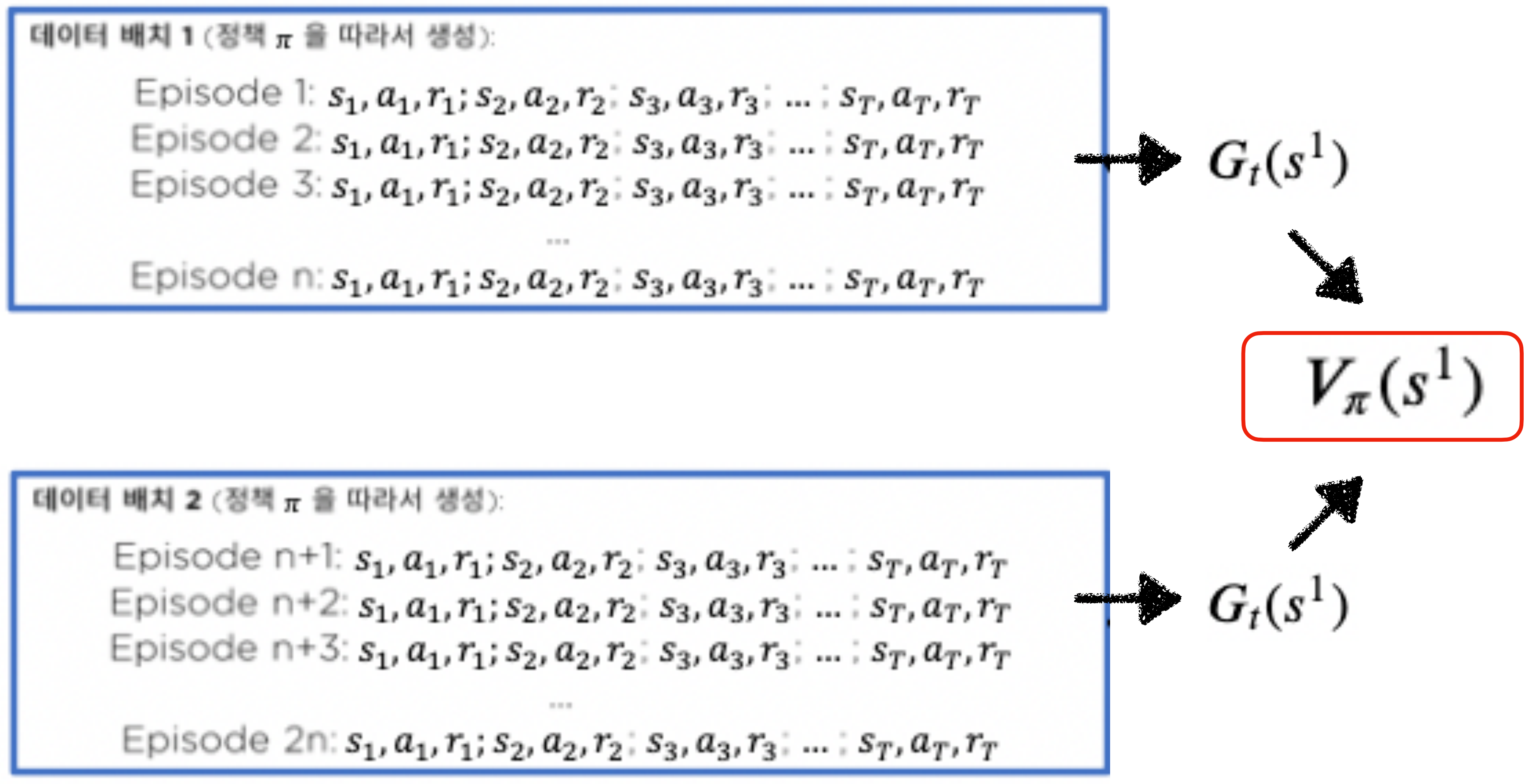
$V_\pi(s^1)$ **= 모든 Episode에 대한 리턴의 산술평균**
= (21+ 14 +18 +12 + 10 + 12) / 6 = 14.5

두 방법 모두 수학적으로 유효한 방법이지만 보편적으로 Every-visit 선호!

Monte Carlo

Value function V by Monte Carlo method

- MC 기법을 활용할때 만약 데이터가 많아진다면?
  - 분산이 낮아져서 더 정교한 가치 추산에 유리해짐
  - But, 모든 return들의 합을 계산해 평균을 내서 추산해야 하므로 모든 return들을 메모리에 저장해야함
    - 필요한 메모리가 늘어나 구현상 문제를 유발할 수 있음



Monte Carlo

Incremental Mean (이동 평균법, 단계적 평균법)

- 기존 방법 : sample들의 평균을 계산
    - State Value function V 추산
- 데이터가 추가된다면?
    - sample들이 추가되고 평균계산에 어려움이 있음
    - Incremental Mean으로 좀 더 손쉽게 추가된 sample을 반영

μ_k : k시점까지의 평균 x_k : k시점의 데이터

(데이터 x_k 는 순차적으로 관측된다고 가정)

$$
\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\
&= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \\
&= \frac{1}{k}\left(x_k + (k-1)\mu_{k-1}\right) \\
&= \mu_{k-1} + \frac{1}{k}\left(x_k - \mu_{k-1}\right)
\end{aligned}
$$

"기존의 알고있던 지식" + "새로운 관측으로 바뀐 지식"

Monte Carlo

Incremental MC policy evalutation

Vanila

MC policy evaluation $V(s) \leftarrow \frac{S(s)}{N(s)}$

$S(s)$: 상태 s에 대한 G_t 들의 합
$N(s)$: 상태 s를 (처음) 방문한 횟수

Incremental
MC policy evaluation

상태 s (처음) 방문때마다,

$$N(s) \leftarrow N(s) + 1$$

$$V(s) \leftarrow V(s) + \boxed{\frac{1}{N(s)}}(G_t - V(s))$$

Gt를 추산

N(s)는 counter로 구해야함

현실에서는,
$N(s)$ 을 세는 것조차 어려움

$$V(s) \leftarrow V(s) + \boxed{\alpha}(G_t - V(s))$$

Learning rate

Gt를 추산

N(s)를 세는것도 어려움 - s가 실수인경우, 종류가 모두 알려지지 않은 경우 등..

적당히 작은 값으로 counter 역할

Monte Carlo

Action Value function by Monte Carlo

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

알고있는 값들을 활용, 수렴할때 까지 DP를 활용해

$$V^{\pi}(s) \xrightarrow{P,R} Q^{\pi}(s,a) \rightarrow \pi'$$

Greedy algorithm

$$\pi'(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}}\, Q^{\pi}(s,a)$$

일반화된 정책 반복 (Generalized Policy iteration)

입력: 임의의 정책 정책 π

출력: 개선된 정책 π'

1. 임의의 방식을 활용해 적용해 $V^{\pi}(s)$ 계산
2. 임의의 방식을 활용해 적용해 π' 계산 ($\pi' \geq \pi$ 를 만족)

Monte Carlo를 활용해 Value function은 추산했고, 정책은?

$$V^{\pi}(s) \xrightarrow{P,R} \boxed{\quad ? \quad}$$

policy improvement를 위해서는 action value function Q를 구해야함

Monte Carlo

Action Value function by Monte Carlo

데이터 (정책 π 을 따라서 생성): $s_t \in \mathcal{S} = \{s^1, s^2, s^3\}, a_t \in \mathcal{A} = \{a^1, a^2, a^3\}$

Episode 1: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \dots ; s_T, a_T, r_T$
Episode 2: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \dots ; s_T, a_T, r_T$
Episode 3: $s_1, a_1, r_1; s_2, a_2, r_2; s_3, a_3, r_3; \dots ; s_T, a_T, r_T$

(상태, 행동, 보상) 정책 π 을 따라서 생성

First-visit method

Episode 1 : $(\boxed{s^1, a^2}, 1); (s^3, a^1, 5); (s^2, a^3, 3), (s^1, a^3, 10), (s^2, a^2, 2)$ $Q_\pi(s^1, a^2)$ = 1+5+3+10+2 = **21**
Episode 2 : $(s^3, a^1, 5); (s^2, a^2, 2); (\boxed{s^1, a^2}, 1); (s^2, a^3, 3), (s^1, a^3, 10)$ $Q_\pi(s^1, a^2)$ = 1+3+10 = **14**
Episode 3 : $(s^2, a^3, 3); (\boxed{s^1, a^2}, 1); (s^3, a^1, 5); (s^1, a^3, 10), (s^2, a^2, 2)$ $Q_\pi(s^1, a^2)$ = 1+5+10+2 = **18**

$Q_\pi(s^1, a^2)$ = 모든 Episode에 대한 리턴의 산술평균 = (21+14+18) / 3 = 17.66

위 방식 외에도 Every-visit, Incremental MC도 가능

Monte Carlo

Monte Carlo Policy Evaluation

- **장점**
  - 환경에 대한 지식이 필요하지 않음
  - 직관적이고 구현이 용이
  - 항상 정확한 가치 함수 값을 계산함 (unbiased estimator)
- **단점**
  - Episode가 끝나야만 적용이 가능
    - 미래의 보상들을 무시하고 사용 가능하지만 불편추정량의 특성을 잃게 됨
  - DP와는 다르게 각 상태와 행동의 관계에 대해 전혀 활용하지 않음
  - 정확한 값을 얻기 위해서는 그만큼 많은 데이터(시뮬레이션)을 필요 - 느림

Exercise

코드는 Open AI gym interface를 따름

Monte Carlo prediction

```
mc_agent = ExactMCAgent(gamma=1.0,
                        num_states=nx * ny,
                        num_actions=4,
                        epsilon=1.0) # epsilon=1.0? -> 모든 경우에 랜덤으로 수행. 각 행동 확률이 0.25
```

```
action_mapper = {
    0: 'UP',
    1: 'RIGHT',
    2: 'DOWN',
    3: 'LEFT'
}
```

행동 정책 함수 state의 q값으로 부터 action을 선택 하거나, random으로 선택하거나

```
def get_action(self, state) :
    prob = np.random.uniform(0.0, 1.0, 1)
    # e-greedy policy over Q
    if prob <= self.epsilon : # random
        action = np.random.choice(range(self.num_action))
    else :
        action = self._policy_q[state, :].argmax()
    return action
```

Vanilla MC PE

```
class ExactMCAgent:
    """
    The exact Monte-Carlo agent.
    This agents performs value update as follows:
    V(s) <- s(s) / n(s)
    Q(s,a) <- s(s,a) / n(s,a)
    """

    def __init__(self,
                 gamma: float,
                 num_states: int,
                 num_actions: int,
                 epsilon: float):
        self.gamma = gamma
        self.num_states = num_states # 상태공간의 크기 : 4x4 =16
        self.num_actions = num_actions
        self.epsilon = epsilon

        self._eps = 1e-10  # for stable computation of V and Q. NOT the one for e-greedy !
```

1. `gamma` : 감가율
2. `num_states` : 상태공간의 크기 (서로 다른 상태의 갯수)
3. `num_actions` : 행동공간의 크기 (서로 다른 행동의 갯수)
4. `epsilon` : ϵ-탐욕적 정책의 파라미터
    - 기존 greedy 알고리즘은 제일 높은 가치의 액션을 선택했다면, ϵ값 만큼 랜덤한 액션을 수행해 exploration을 보충함
5. `_eps` : 해당 값으로 나눗셈에 0이 들어갈경우 연산이 불가하므로 매우 작은 값을 넣어줌

Exercise

Monte Carlo prediction

General RL '**Agent-Environment Interaction**' framework

반복 : 주어진 에피소드 수만큼

- **특정 에피소드 시작**
- **반복 : 에피소드 내부**
  - **현재 state** — 환경으로부터 현재 상태 관측
  - **현재 action** — agent의 정책함수에 따른 action 선택
  - **다음 state, reward** — 환경에 현재 action을 가함으로써 얻음
  - **if 다음 상태 == 종결상태 :**
    - **Break**

Policy Evaluation
Policy Improvement
선택적으로 매 episode마다 value function을 평가하거나 개선할 수 있음

```python
env.reset() # 환경 초기화, episode 재시작
step_counter = 0
while True:
    print("At t = {}".format(step_counter))
    env._render() # state를 시각적으로 볼 수 있게 rendering
    |
    cur_state = env.observe() # 현재 상태정보
    action = mc_agent.get_action(cur_state) # greedy 정
    next_state, reward, done, info = env.step(action) #

    print("state : {}".format(cur_state))
    print("aciton : {}".format(action_mapper[action]))
    print("reward : {}".format(reward))
    print("next state : {} \n".format(next_state))
    print("done : {} \n".format(done))
    print("info : {} \n".format(info))
    step_counter += 1
    if done:
        break
```

```
At t = 0
==========
T  o  o  o
o  o  o  o
o  o  o  o
o  o  o  x
==========
state : 15
aciton : DOWN
reward : 0.0
next state : 15

done : True

info : {'prob': 1.0}
```

Exercise

Monte Carlo Policy evaluation

Vanilla MC PE

MC policy evaluation $V(s) \leftarrow \frac{S(s)}{N(s)}$

$S(s)$: 상태 s에 대한 G_t 들의 합
$N(s)$: 상태 s를 (처음) 방문한 횟수

General RL ‘**Agent-Environment Interaction**’ framework

반복 : 주어진 에피소드 수만큼

특정 에피소드 시작

반복 : 에피소드 내부

현재 state — 환경으로부터 현재 상태 관측

현재 action — agent의 정책함수에 따른 action 선택

다음 state, reward — 환경에 현재 action을 가함으로써 얻음

if 다음 상태 == 종결상태 :

Break

Policy Evaluation
Policy Improvement

선택적으로 매 episode마다 value function을 평가하거나 개선할 수 있음

```python
def run_episode(env, agent):
    env.reset()
    states = []
    actions = []
    rewards = []

    while True:
        state = env.observe()
        action = agent.get_action(state)
        next_state, reward, done, info = env.step(action)

        states.append(state)
        actions.append(action)
        rewards.append(reward)

        if done:
            break

    episode = (states, actions, rewards)
    agent.update(episode)
```

Exercise

Monte Carlo Policy evaluation

agent.update(episode) : episode 내에서 Return 값 계산

```
def update(self, episode) :
    states, actions, rewards = episode

    # reversing the inputs!
    # for efficient computation of returns 계산 효율을 위해 reverse시킴
    states = reversed(states)
    actions = reversed(actions)
    rewards = reversed(rewards)

    iter = zip(states, actions, rewards)
    cum_r = 0 # terminal state에서 어떤 action을 하든 terminal로 되돌아옴. 그에 따른 reward는 0을 줌.
    for s, a, r in iter :
        cum_r *= self.gamma # 이전 step에서 계산한 reward이기 때문에 감가.
        cum_r += r          # 현재 reward 추가

        self.n_v[s] += 1
        self.n_q[s, a] += 1

        self.s_v[s] += cum_r
        self.s_q[s, a] += cum_r
```

state counter
(state, action) counter

sum of v
sum of q

```
%%time
mc_agent.reset_statistics() # agent.n_v, agent.n_q, agent.s_v, agent.s_q 을 0으로 초기화 합니다.
for _ in range(10):
    run_episode(env, mc_agent)
```

Wall time: 8.97 ms

Discount factor

Return $G_t = R_t + \boxed{\gamma} R_{t+1} + \boxed{\gamma^2} R_{t+1} + \ldots$

역순으로 계산하는 것이 효율적

$$G_t = \gamma^{t-1} R_t + \gamma^{t-2} R_{t-1} + \ldots + R_1$$

MC policy evaluation $V(s) \leftarrow \frac{S(s)}{N(s)}$

$S(s)$: 상태 s에 대한 G_t 들의 합
$N(s)$: 상태 s를 (처음) 방문한 횟수

Exercise

Monte Carlo Policy evaluation

MC policy evaluation $V(s) \leftarrow \frac{S(s)}{N(s)}$

$S(s)$: 상태 s에 대한 G_t 들의 합
$N(s)$: 상태 s를 (처음) 방문한 횟수

self._eps : 분모가 0이 되는 것을 방지

```python
def compute_values(self):
    self.v = self.s_v / (self.n_v + self._eps)
    self.q = self.s_q / (self.n_q + self._eps)
```

```python
mc_agent.compute_values()
```

```python
mc_agent.v
```

```
array([  0.        ,  -8.53846154, -19.05882353, -15.57692308,
        -8.27272727, -17.17647059, -20.625     , -12.91304348,
       -12.4       , -23.1       , -33.25      , -17.66666667,
       -28.18181818, -32.38461538, -28.        ,   0.        ])
```

```python
mc_agent.q
```

```
array([[  0.        ,   0.        ,   0.        ,   0.        ],
       [-15.48914616, -23.0173913 , -18.70144928,  -1.        ],
       [-20.92094456, -23.12972421, -21.76346357, -16.01351351],
       [-22.37179487, -23.197405  , -21.46428571, -21.46240602],
       [ -1.        , -19.34383954, -20.50297619, -15.49515906],
       [-15.72390572, -20.5430622 , -19.61938534, -15.94159714],
       [-20.89384289, -21.13203463, -20.11885246, -19.02134472],
       [-23.75954592, -21.12553648, -14.29913607, -20.82377477],
       [-14.72048436, -21.3347779 , -23.10115911, -21.76986584],
       [-18.69007804, -18.76239669, -19.68944099, -21.37780149],
       [-22.19651163, -15.63807286, -13.59096176, -21.05502392],
       [-20.15074627, -14.81120944,  -1.        , -18.85381026],
       [-21.05674419, -20.47992352, -22.6268797 , -22.49902724],
       [-20.40585774, -14.77860963, -20.55022321, -22.03560209],
       [-18.34157651,  -1.        , -14.12272727, -20.78742515],
       [  0.        ,   0.        ,   0.        ,   0.        ]])
```

```python
mc_agent.n_v
```

```
array([ 1., 10., 14., 32., 13., 15., 13., 23., 13., 14.,  4.,  9., 17.,
       11.,  4.,  0.])
```

```python
mc_agent.n_q
```

```
array([[ 0.,  0.,  0.,  1.],
       [ 1.,  4.,  1.,  4.],
       [ 3.,  4.,  2.,  5.],
       [15.,  7.,  7.,  3.],
       [ 3.,  0.,  4.,  6.],
       [ 4.,  3.,  3.,  5.],
       [ 2.,  4.,  1.,  6.],
       [ 6.,  6.,  4.,  7.],
       [ 2.,  4.,  4.,  3.],
       [ 7.,  3.,  0.,  4.],
       [ 0.,  1.,  1.,  2.],
       [ 6.,  2.,  1.,  0.],
       [ 0.,  6.,  6.,  5.],
       [ 5.,  2.,  2.,  2.],
       [ 0.,  1.,  1.,  2.],
       [ 0.,  0.,  0.,  0.]])
```

```python
mc_agent.s_v
```

```
array([   0., -108., -260., -725., -213., -227., -181., -310., -317.,
       -322.,  -94., -125., -561., -339.,  -66.,    0.])
```

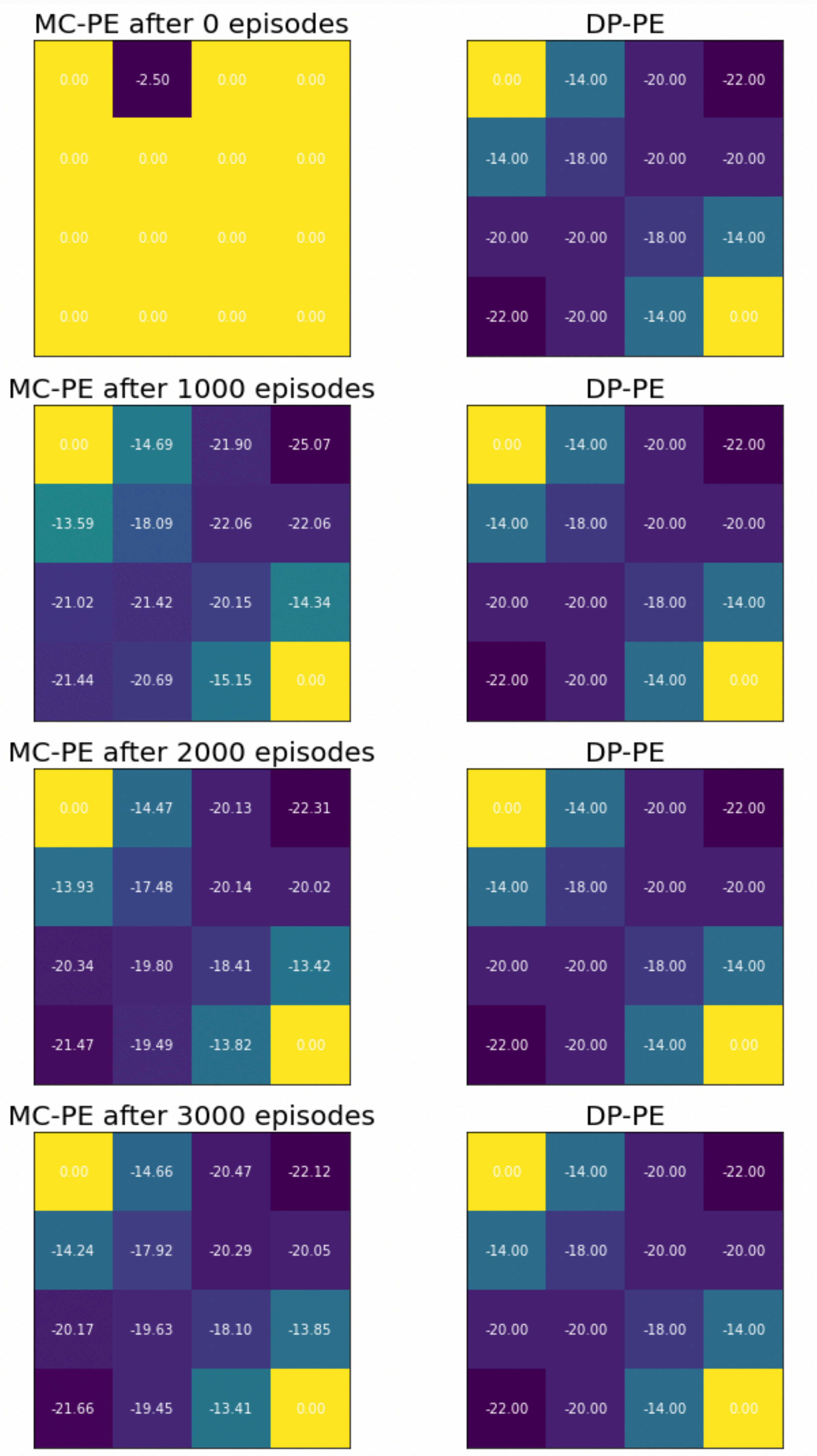
```python
mc_agent.s_q
```

```
array([[   0.,    0.,    0.,    0.],
       [ -25.,  -55.,  -24.,   -4.],
       [ -51., -128.,  -45.,  -36.],
       [-380., -114., -162.,  -69.],
       [  -3.,    0., -102., -108.],
       [ -57.,  -66.,  -37.,  -67.],
       [ -46.,  -36.,  -20.,  -79.],
       [-135.,  -58.,  -40.,  -77.],
       [ -51.,  -72., -142.,  -52.],
       [-129.,  -78.,    0., -115.],
       [   0.,  -29.,   -2.,  -63.],
       [ -77.,  -47.,   -1.,    0.],
       [   0., -198., -170., -193.],
       [-164.,  -40.,  -62.,  -73.],
       [   0.,   -1.,  -28.,  -37.],
       [   0.,    0.,    0.,    0.]])
```

Exercise

Monte Carlo Policy evaluation

- MC vs DP
    - MC는 MDP 환경에 대한 사전정보 없이 수행
        - model-free
        - 수렴이 느림
    - DP는 MDP 환경에 대한 사전정보를 활용해서 수행
        - model-based
        - 수렴이 빠름
- **episode가 많아질수록** MC도 DP와 비슷하게 수렴
- MC는 실행을 할때마다 값이 달라질 수 있음
    - Random 정책을 사용하기 때문



Exercise

Monte Carlo Policy evaluation

General RL '**Agent-Environment Interaction**' framework

반복 : 주어진 에피소드 수만큼

특정 에피소드 시작

반복 : 에피소드 내부

현재 state — 환경으로부터 현재 상태 관측

현재 action — agent의 정책함수에 따른 action 선택

다음 state, reward — 환경에 현재 action을 가함으로써 얻음

if 다음 상태 == 종결상태 :

Break

Policy Evaluation
Policy Improvement
선택적으로 매 episode마다 value function을 평가하거나 개선할 수 있음

Incremental MC PE

$$V(s) \leftarrow V(s) + \boxed{\alpha}(G_t - V(s))$$

Learning rate

```python
def run_episode(env, agent):
    env.reset()
    states = []
    actions = []
    rewards = []

    while True:
        state = env.observe()
        action = agent.get_action(state)
        next_state, reward, done, info = env.step(action)

        states.append(state)
        actions.append(action)
        rewards.append(reward)

        if done:
            break

    episode = (states, actions, rewards)
    agent.update(episode)
```

Exercise

Monte Carlo Policy evaluation

Incremental MC PE

$$V(s) \leftarrow V(s) + \boxed{\alpha}(G_t - V(s))$$

Learning rate

```python
def update(self, episode):
    states, actions, rewards = episode

    # reversing the inputs!
    # for efficient computation of returns
    states = reversed(states)
    actions = reversed(actions)
    rewards = reversed(rewards)

    iter = zip(states, actions, rewards)
    cum_r = 0
    for s, a, r in iter:
        cum_r *= self.gamma
        cum_r += r

        self.v[s] += self.lr * (cum_r - self.v[s])
        self.q[s, a] += self.lr * (cum_r - self.q[s, a])
```

$$V(s) \leftarrow V(s) + \boxed{\alpha}(G_t - V(s))$$

Learning rate

vanilla와는 다르게 통으로 내부에서 계산을 끝냄

Discount factor

Return $G_t = R_t + \boxed{\gamma} R_{t+1} + \boxed{\gamma^2} R_{t+1} + ...$

역순으로 계산하는 것이 효율적

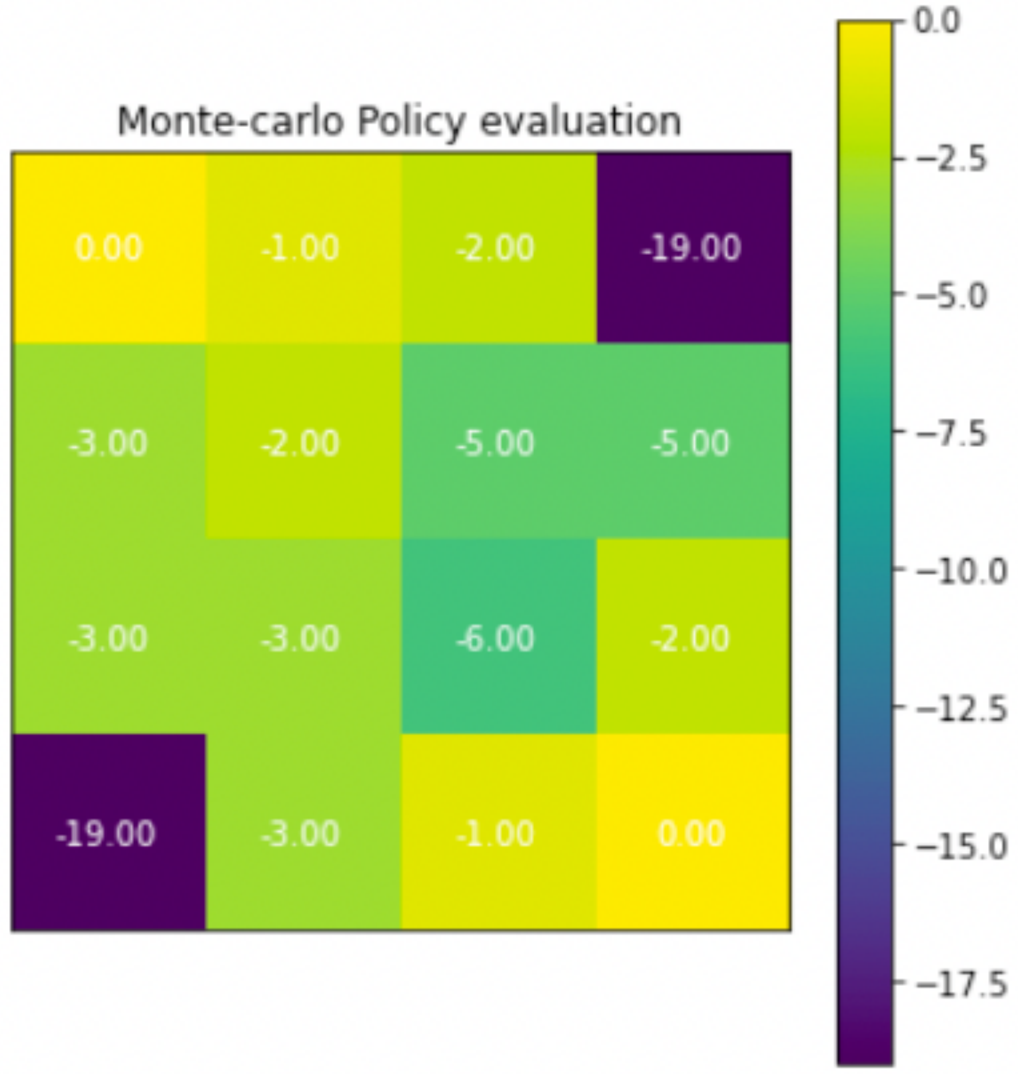
$$G_t = \gamma^{t-1} R_t + \gamma^{t-2} R_{t-1} + ... + R_1$$

Exercise

Monte Carlo Policy evaluation

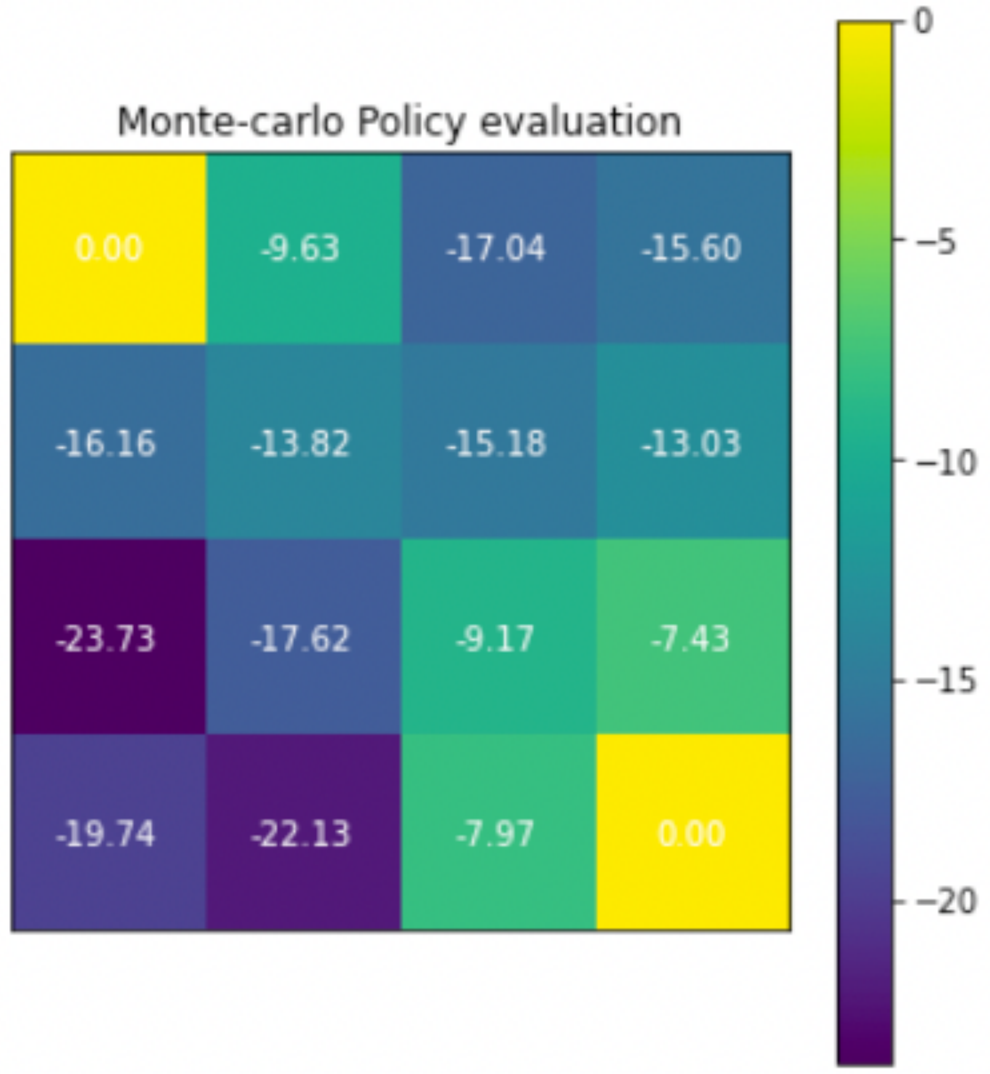
MC

Learning rate = 1



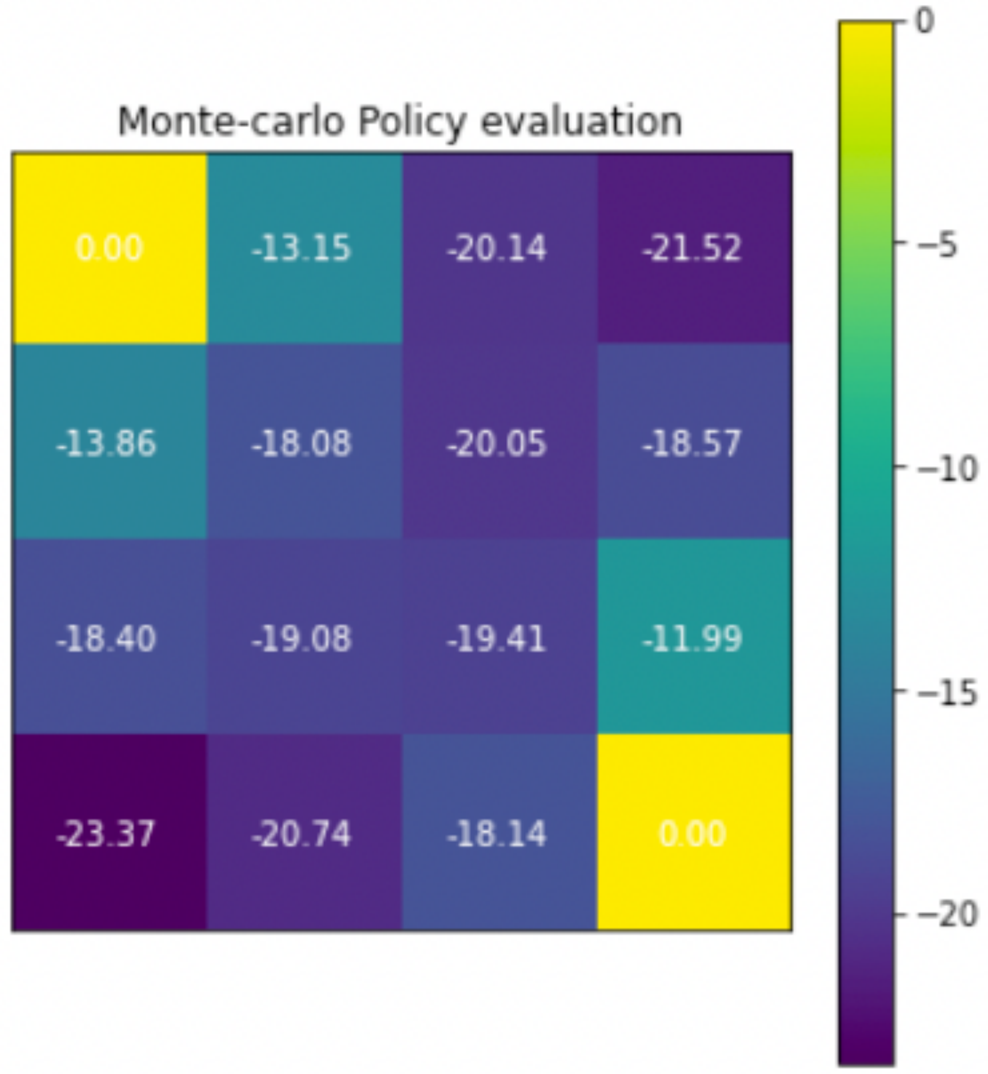
MC

Learning rate = 0.1

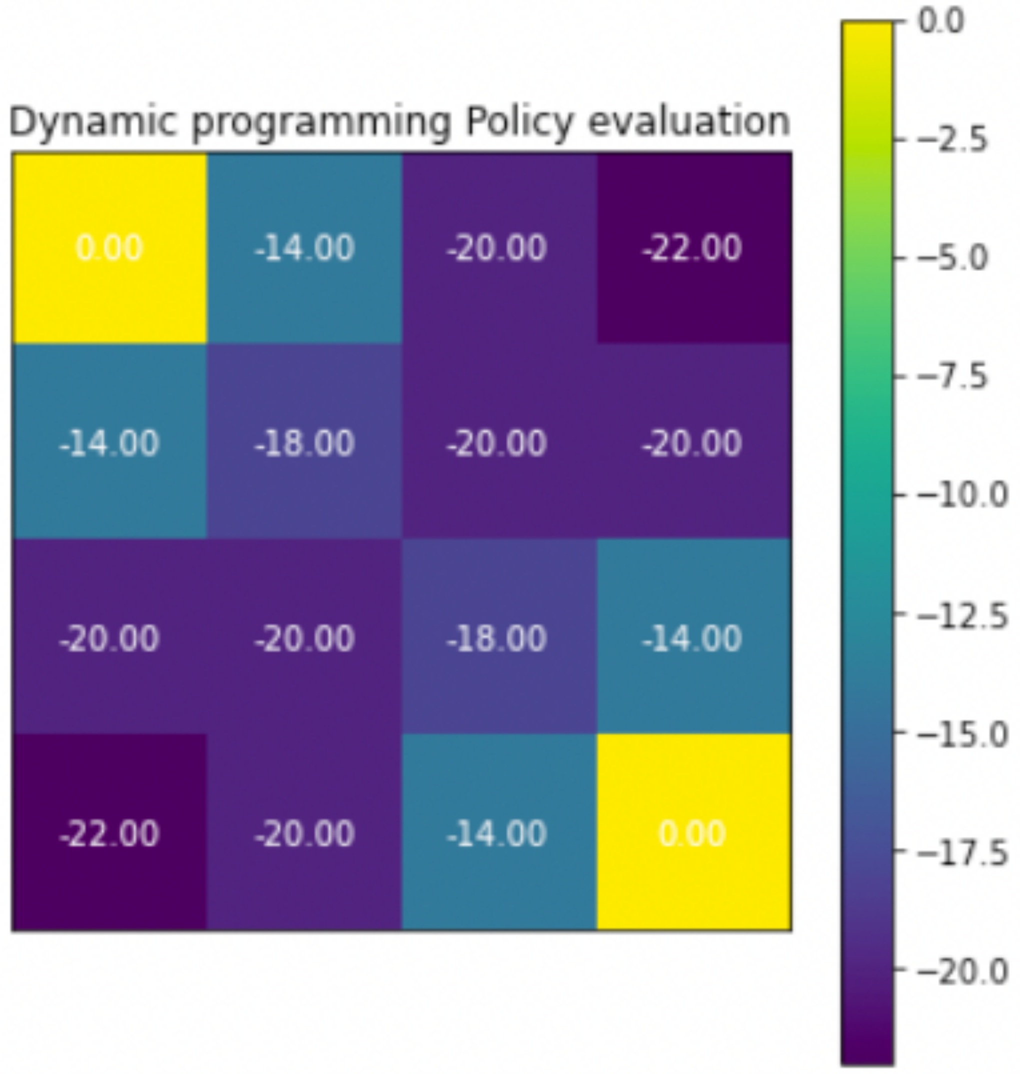


MC

Learning rate = 0.01



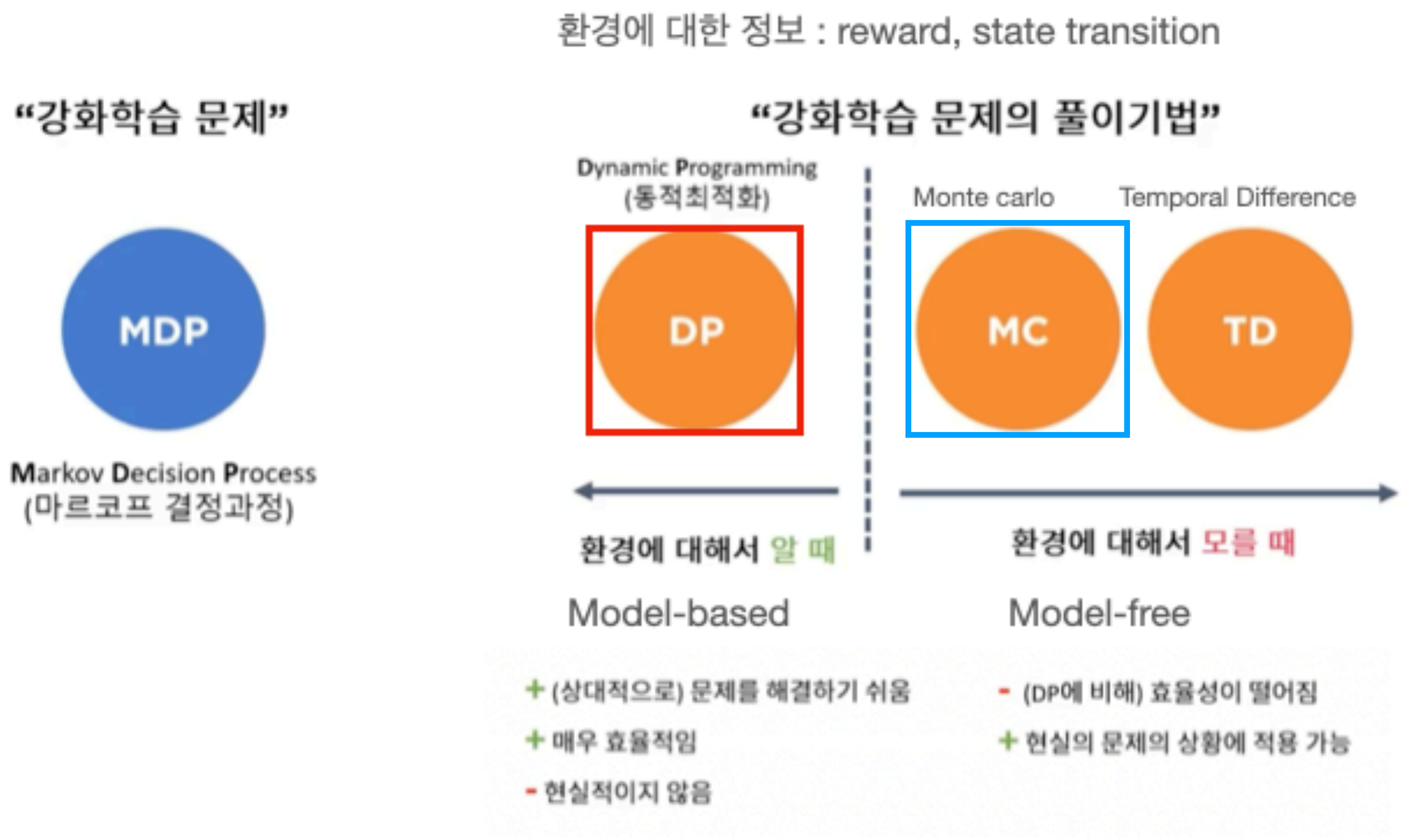
DP



Hyper parameter tuning도 매우 중요!

Reinforcement Learning
강화학습 문제

- 최적 정책를 찾는 것!
- **Model based : DP, Asynchronous DP**
    - **알고 있는 값**들을 활용해 특정 공식을 수렴할 때까지 반복해 최적해를 찾음
- **Model free : MC, TD**
    - 현실에서는 환경에 대해서 모를때가 많음
    - **MC** : 데이터를 활용해서 최적해를 찾음
    - **TD** : 알고있는 일부 경험과 함께 데이터를 활용

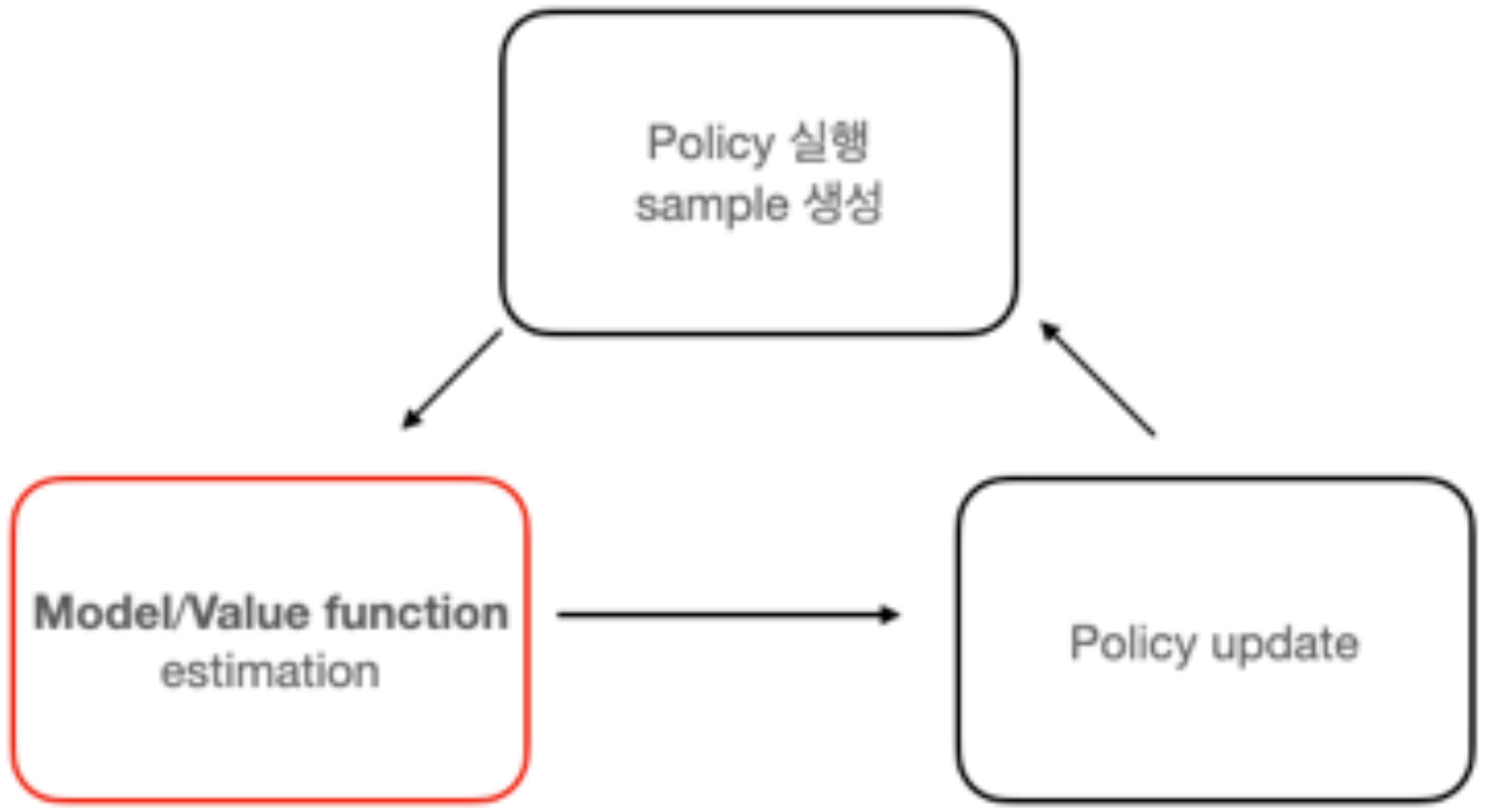


Reinforcement Learning

Model / Value function estimation

- 강화학습 프로세스
  - 최적정책, 상태에 맞는 최적 행동찾기
  - **환경에 대한 정보가 있을 때** (model-based)
    - 주어진 정보를 토대로 DP를 통한 계산, V와 Q를 update 함
  - **환경에 대한 정보가 없을 때** (model-free)
    - 여러 에피소드를 agent가 직접 경험하면서 수집한 데이터 활용
    - MC를 활용해 V, Q를 추산하고 최적 정책을 찾음

강화학습의 공통적인 iteration



DP를 활용해 효율적으로 계산