Temporal Difference Method

Prof. Tae-Hyoung Park
Dept. of Intelligent Systems & Robotics, CBNU

Temporal Difference 방법

- MC 법 vs. TD 법
  - MC (Monte Carlo) 법
    - 에피소드가 끝에 도달한 후 가치함수(Q) 계산 → 정책평가 (policy evaluation) → 정책갱신 (policy update)
    - 일회성 과제 (O), 지속성 과제 (X)
  - TD (Temporal Difference (시간차)) 법
    - 에피소드가 끝날 때 까지 기다리지 않고, 일정 시간 마다 정책평가 및 정책갱신
    - 일회성 과제 (O), 지속성 과제 (O)
    - TD 법 = MC 법 + DP 법

Policy Evaluation

- Return

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots$$
$$= R_t + \gamma G_{t+1}$$

- Value Function

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$
$$= \mathbb{E}_\pi[R_t + \gamma G_{t+1} \mid S_t = s]$$

- DP 법
  - 다음상태의 가치로 부터 현재상태의 가치 추정 (bootstrap)
  - 환경에 대한 모델링을 통하여, 모든 경우의 상태 전이를 고려함

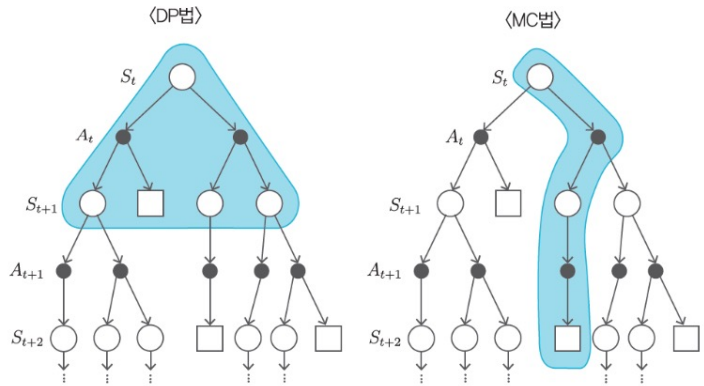
$$V'_\pi(s) = \sum_{a,s'} \pi(a \mid s)\, p(s' \mid s,a)\{r(s,a,s') + \gamma V_\pi(s')\}$$

(Bellman equation)

- MC 법
  - 특정한 sample data (현재상태→최종상태) 의 return 평균

$$V'_\pi(S_t) = V_\pi(S_t) + \alpha\{G_t - V_\pi(S_t)\}$$

(Exponential moving average)



Policy Evaluation

- TD 법
  - DP 법 처럼 bootstrap 을 통해 가치함수를 순차적으로 갱신 (다음상태→현재상태)
  - MC법 처럼 환경에 대한 정보 없이 sampling 된 데이터 만으로 가치함수 갱신

$$V'_\pi(S_t) = V_\pi(S_t) + \alpha\{R_t + \gamma V_\pi(S_{t+1}) - V_\pi(S_t)\}$$

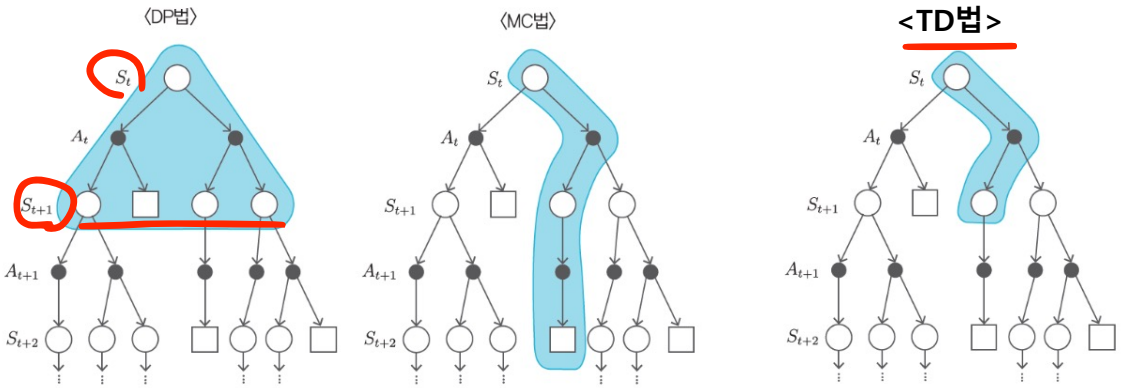
〈MC법〉 $V'_\pi(S_t) = V_\pi(S_t) + \alpha\{G_t - V_\pi(S_t)\}$

〈TD법〉 $V'_\pi(S_t) = V_\pi(S_t) + \alpha\{R_t + \gamma V_\pi(S_{t+1}) - V_\pi(S_t)\}$

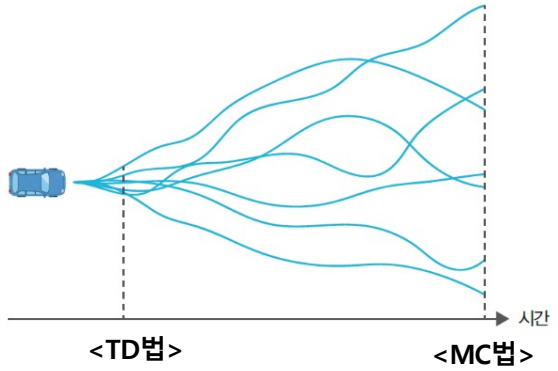
TD Target

G_t: 목표에 도달 시 얻을 수 있는 수익,
현재→목표까지의 샘플필요

$R_t + \gamma V_\pi(S_{t+1})$: 목표에 도달 시 얻을 수 있는 수익 (추정치)
현재→다음까지의 샘플 필요



✓ TD법이 MC법 대비 데이터의 변동성 감소



Policy Evaluation

- TD 법 구현
  - TdAgent Class

```python
class TdAgent:
    def __init__(self):
        self.gamma = 0.9
        self.alpha = 0.01
        self.action_size = 4

        random_actions = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
        self.pi = defaultdict(lambda: random_actions)
        self.V = defaultdict(lambda: 0)

    def get_action(self, state):
        action_probs = self.pi[state]
        actions = list(action_probs.keys())
        probs = list(action_probs.values())
        return np.random.choice(actions, p=probs)

    def eval(self, state, reward, next_state, done):
        next_V = 0 if done else self.V[next_state]  # 목표 지점의 가치 함수는 0
        target = reward + self.gamma * next_V

        self.V[state] += (target - self.V[state]) * self.alpha
```

매 step 마다 계산

$$V'_{\pi}(S_t) = V_{\pi}(S_t) + \alpha\{R_t + \gamma V_{\pi}(S_{t+1}) - V_{\pi}(S_t)\}$$

<MC 법>

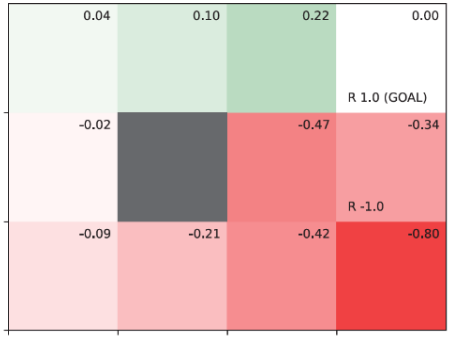
목표 도달 시 계산

```python
def eval(self):
    G = 0
    for data in reversed(self.memory):  # 역방향으로(reserved) 따라가기
        state, action, reward = data
        G = self.gamma * G + reward
        self.cnts[state] += 1
        self.V[state] += (G - self.V[state]) / self.cnts[state]
```

Policy Evaluation

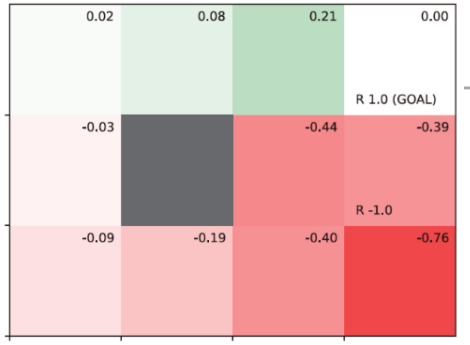
- TD 법 구현

```
env = GridWorld()
agent = TdAgent()

episodes = 1000
for episode in range(episodes):
    state = env.reset()

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

        agent.eval(state, reward, next_state, done) # 매번 호출
        if done:
            break
        state = next_state

env.render_v(agent.V)
```



<MC 법>

```
env = GridWorld()
agent = RandomAgent()

episodes = 1000
for episode in range(episodes):  # 에피소드 1000번 수행
    state = env.reset()
    agent.reset()

    while True:
        action = agent.get_action(state)                # 행동 선택
        next_state, reward, done = env.step(action)  # 행동 수행

        agent.add(state, action, reward)  # (상태, 행동, 보상) 저장
        if done:  # 목표에 도달 시
            agent.eval()  # 몬테카를로법으로 가치 함수 갱신
            break         # 다음 에피소드 시작

        state = next_state

# 가치 함수 시각화
env.render_v(agent.V)
```



Policy Evaluation

실습 #1 td_eval.py

```python
from collections import defaultdict
import numpy as np
from common.gridworld import GridWorld

class TdAgent:
    def __init__(self):
        self.gamma = 0.9
        self.alpha = 0.01
        self.action_size = 4

        random_actions = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
        self.pi = defaultdict(lambda: random_actions)
        self.V = defaultdict(lambda: 0)

    def get_action(self, state):
        action_probs = self.pi[state]
        actions = list(action_probs.keys())
        probs = list(action_probs.values())
        return np.random.choice(actions, p=probs)

    def eval(self, state, reward, next_state, done):
        next_V = 0 if done else self.V[next_state]  # 목표 지점의 가치 함수는 0
        target = reward + self.gamma * next_V
        self.V[state] += (target - self.V[state]) * self.alpha
```

```python
env = GridWorld()
agent = TdAgent()

episodes = 1000
for episode in range(episodes):
    state = env.reset()

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

        agent.eval(state, reward, next_state, done)  # 매번 호출
        if done:
            break
        state = next_state

# 가치 함수 시각화
env.render_v(agent.V)
```

SARSA

- Value Function Update (TD 법)
  - State-value function

$$V'_{\pi}(S_t) = V_{\pi}(S_t) + \alpha\{R_t + \gamma V_{\pi}(S_{t+1}) - V_{\pi}(S_t)\}$$

  - Action-value function

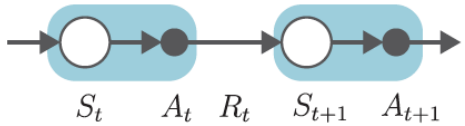
$$Q'_{\pi}(S_t, A_t) = Q_{\pi}(S_t, A_t) + \alpha\{R_t + \gamma Q_{\pi}(S_{t+1}, A_{t+1}) - Q_{\pi}(S_t, A_t)\}$$

- Policy Update

$$\pi'(a \mid S_t) = \begin{cases} \underset{a}{\text{argmax}}\, Q_{\pi}(S_t, a) & (1-\varepsilon\text{의 확률}) \\ \text{무작위 행동} & (\varepsilon\text{의 확률}) \end{cases}$$

SARSA

- SARSA
  - TD 법으로 Q 함수를 구하고, ε-greedy 로 정책을 개선하는 방법



$(S_t, A_t, R_t, S_{t+1}, A_{t+1})$

S. A. R. S. A

SARSA

- Implementation

```
class SarsaAgent:
    ......
def __init__(self):
    self.gamma = 0.9
    self.alpha = 0.8
    self.epsilon = 0.1
    self.action_size = 4

    random_actions = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
    self.pi = defaultdict(lambda: random_actions)
    self.Q = defaultdict(lambda: 0)
    self.memory = deque(maxlen=2)  # ❶ deque 사용
def update(self, state, action, reward, done):
    self.memory.append((state, action, reward, done))
    if len(self.memory) < 2:
        return

    state, action, reward, done = self.memory[0]
    next_state, next_action, _, _ = self.memory[1]
    # ❸ 다음 Q 함수
    next_q = 0 if done else self.Q[next_state, next_action]

    # ❹ TD법으로 self.Q 갱신
    target = reward + self.gamma * next_q
    self.Q[state, action] += (target - self.Q[state, action]) * self.alpha

    # ❺ 정책 개선
    self.pi[state] = greedy_probs(self.Q, state, self.epsilon)
```

FIFO, 최근 2개 데이터 보관

$$Q'_{\pi}(S_t, A_t) = Q_{\pi}(S_t, A_t) + \alpha\{R_t + \gamma Q_{\pi}(S_{t+1}, A_{t+1}) - Q_{\pi}(S_t, A_t)\}$$

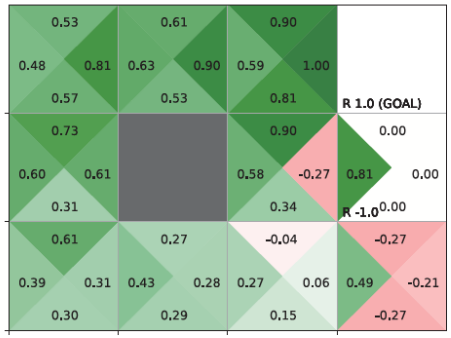
$$\pi'(a \mid S_t) = \begin{cases} \underset{a}{\text{argmax}}\, Q_{\pi}(S_t, a) & (1-\varepsilon\text{의 확률}) \\ \text{무작위 행동} & (\varepsilon\text{의 확률}) \end{cases}$$

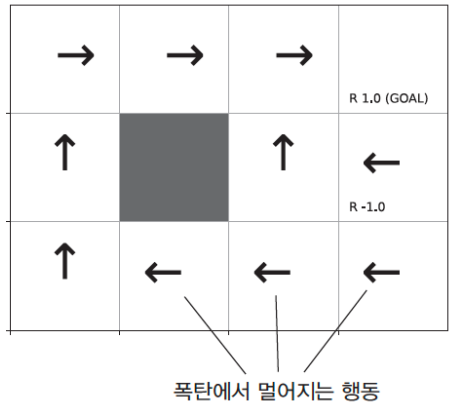
SARSA

- Implementation

```
env = GridWorld()
agent = SarsaAgent()

episodes = 10000
for episode in range(episodes):
    state = env.reset()
    agent.reset()

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

        agent.update(state, action, reward, done)  # ❶ 매번 호출

        if done:
            # ❷ 목표에 도달했을 때도 호출
            agent.update(next_state, None, None, None)
            break
        state = next_state

env.render_q(agent.Q)  # Q 함수 시각화
```

agent.update() 는 2번의 호출을 1세트로 정책을 갱신함
→ 목표 도달 시 호출





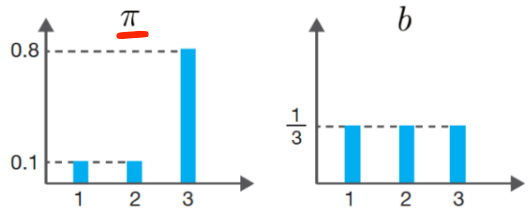
SARSA

실습 #2 sarsa.py

```python
from collections import defaultdict, deque
import numpy as np
from common.gridworld import GridWorld
from common.utils import greedy_probs

class SarsaAgent:
    def __init__(self):
        self.gamma = 0.9
        self.alpha = 0.8
        self.epsilon = 0.1
        self.action_size = 4

        random_actions = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
        self.pi = defaultdict(lambda: random_actions)
        self.Q = defaultdict(lambda: 0)
        self.memory = deque(maxlen=2)  # deque 사용

    def get_action(self, state):
        action_probs = self.pi[state]  # pi에서 선택
        actions = list(action_probs.keys())
        probs = list(action_probs.values())
        return np.random.choice(actions, p=probs)

    def reset(self):
        self.memory.clear()

    def update(self, state, action, reward, done):
        self.memory.append((state, action, reward, done))
        if len(self.memory) < 2:
            return
        state, action, reward, done = self.memory[0]
        next_state, next_action, _, _ = self.memory[1]
        next_q = 0 if done else self.Q[next_state, next_action]  # 다음 Q 함수

        # TD법으로 self.Q 갱신
        target = reward + self.gamma * next_q
        self.Q[state, action] += (target - self.Q[state, action]) * self.alpha

        # 정책 개선
        self.pi[state] = greedy_probs(self.Q, state, self.epsilon)

env = GridWorld()
agent = SarsaAgent()

episodes = 10000
for episode in range(episodes):
    state = env.reset()
    agent.reset()

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

        agent.update(state, action, reward, done)  # 매번 호출

        if done:
            # 목표에 도달했을 때도 호출
            agent.update(next_state, None, None, None)
            break
        state = next_state

# 시각화
env.render_q(agent.Q)
```

Importance Sampling

- Definition
  - 확률분포 π 의 기대값을 다른 확률분포 (b) 에서 샘플링한 데이터를 사용하여 계산하는 기법
- 원리



$$\begin{aligned}\mathbb{E}_{\pi}[x] &= \sum x\pi(x) \\ &= \sum x\frac{b(x)}{b(x)}\pi(x) \\ &= \sum x\frac{\pi(x)}{b(x)}b(x) \\ &= \mathbb{E}_{b}\left[x\frac{\pi(x)}{b(x)}\right]\end{aligned}$$

- 방법

확률분포 b 에서의 샘플링: Sampling: $x^{(i)} \sim b \quad (i = 1, \cdots, n)$

$$\rho(x) = \frac{\pi(x)}{b(x)}$$

$$\mathbb{E}_{\pi}[x] \approx \frac{\rho(x^{(1)})x^{(1)} + \cdots + \rho(x^{(n)})x^{(n)}}{n}$$

Importance Sampling

```python
import numpy as np

x = np.array([1, 2, 3])         # 확률 변수
pi = np.array([0.1, 0.1, 0.8])  # 확률 분포

# 기댓값의 참값 계산
e = np.sum(x * pi)
print('참값(E_pi[x]):', e)

# 몬테카를로법으로 계산
n = 100  # 샘플 개수
samples = []
for _ in range(n):
    s = np.random.choice(x, p=pi)  # pi를 이용한 샘플링
    samples.append(s)
    mean = np.mean(samples)  # 샘플들의 평균
    var = np.var(samples)    # 샘플들의 분산
    print('몬테카를로법: {:.2f} (분산: {:.2f})'.format(mean, var))

#중요도 샘플링으로 계산
b = np.array([1/3, 1/3, 1/3])  # 확률 분포
n = 100  # 샘플 개수
samples = []

for _ in range(n):
    idx = np.arange(len(b))          # b의 인덱스([0, 1, 2])
    i = np.random.choice(idx, p=b)  # b를 사용하여 샘플링
    s = x[i]
    rho = pi[i] / b[i]               # 가중치
    samples.append(rho * s)          # 샘플 데이터에 가중치를 곱해 저장

mean = np.mean(samples)
var = np.var(samples)
print('중요도 샘플링: {:.2f} (분산: {:.2f})'.format(mean, var))
```

$\pi(x)$

$E_\pi(x)$

$\pi \sim x^{(1)} + \cdots + x^{(n)}$

$$E_\pi(x) = \frac{x^{(1)} + \cdots + x^{(n)}}{n}$$

$b \sim x^{(1)} + \cdots + x^{(n)}$

$$E_\pi(x) = \frac{\rho(x^{(1)})x^{(1)} + \cdots + \rho(x^{(n)})x^{(n)}}{n}$$

두 확률 분포가 유사하면 샘플링의 분산이 작아짐

(ex) b = np.array(0.2,0.2,0.6)
=> mean = 2.72, var = 2.48)

```
참값(E_pi[x]): 2.7
몬테카를로법: 2.78 (분산: 0.27)
```

```
중요도 샘플링: 2.95 (분산: 10.63)
```

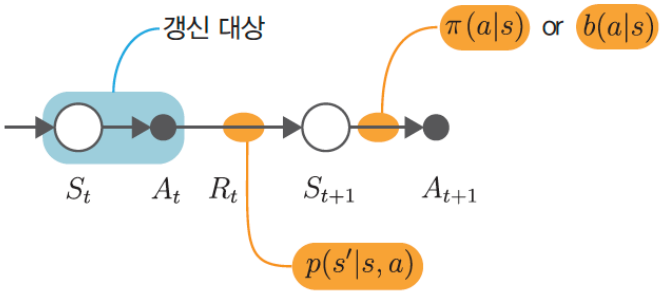
On-Policy vs. Off-Policy

- On-Policy
  - 스스로 쌓은 경험을 토대로 자신의 정책을 개선
  - '대상 정책 (target policy)' 와 '행동 정책 (behavior policy)' 가 동일함
- Off-Policy
  - 자신과 다른 환경에서 쌓은 경험을 토대로 자신의 정책을 개선

    (ex) 다른 테니스 선수가 스윙하는 모습을 보고 자신의 스윙 자세를 고침
  - '대상 정책' 과 '행동 정책' 이 구분됨
    - '행동 정책'(b) 에서 얻은 샘플 데이터로부터 '대상 정책'(π) 의 기댓값을 계산하는 방법
    - 중요도 샘플링 (importance sampling) 필요
  - 정책의 최적성이 향상됨

Off-Policy SARSA

- 행동 정책과 대상 정책
  - 대상 정책 π
    - Policy upgrade: greedy (exploitation)
  - 행동 정책 b
    - Policy upgrade: e-greedy (exploration)



$$\rho = \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$$

TD target

샘플링: $A_{t+1} \sim \pi$

$$Q'_\pi(S_t, A_t) = Q_\pi(S_t, A_t) + \alpha\{R_t + \gamma Q_\pi(S_{t+1}, A_{t+1}) - Q_\pi(S_t, A_t)\}$$

샘플링: $A_{t+1} \sim b$

$$Q'_\pi(S_t, A_t) = Q_\pi(S_t, A_t) + \alpha\{\rho(R_t + \gamma Q_\pi(S_{t+1}, A_{t+1})) - Q_\pi(S_t, A_t)\}$$

TD target 보정

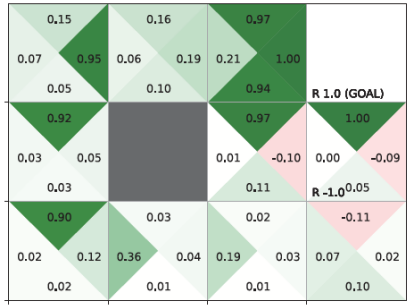
Off-Policy SARSA

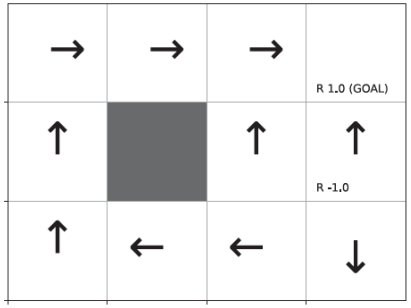
- Implementation

```python
class SarsaOffPolicyAgent:
    def __init__(self):
        self.gamma = 0.9
        self.alpha = 0.8
        self.epsilon = 0.1
        self.action_size = 4

        random_actions = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
        self.pi = defaultdict(lambda: random_actions)
        self.b = defaultdict(lambda: random_actions)
        self.Q = defaultdict(lambda: 0)
        self.memory = deque(maxlen=2)

    def get_action(self, state):
        action_probs = self.b[state]  # ❶ 행동 정책에서 가져옴
        actions = list(action_probs.keys())
        probs = list(action_probs.values())
        return np.random.choice(actions, p=probs)

    def reset(self):
        self.memory.clear()
```

행동 정책

```python
def update(self, state, action, reward, done):
    self.memory.append((state, action, reward, done))
    if len(self.memory) < 2:
        return

    state, action, reward, done = self.memory[0]
    next_state, next_action, _, _ = self.memory[1]

    if done:
        next_q = 0
        rho = 1
    else:
        next_q = self.Q[next_state, next_action]
        # ❷ 가중치 rho 계산
        rho = self.pi[next_state][next_action] / self.b[next_state][next_action]

    # ❸ rho로 TD 목표 보정
    target = rho * (reward + self.gamma * next_q)
    self.Q[state, action] += (target - self.Q[state, action]) * self.alpha

    # ❹ 각각의 정책 개선
    self.pi[state] = greedy_probs(self.Q, state, 0)
    self.b[state] = greedy_probs(self.Q, state, self.epsilon)
```





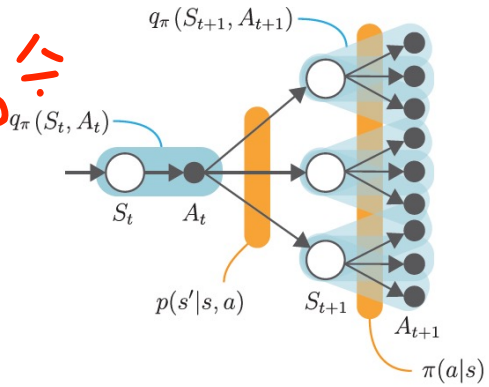
Q Learning

- SARSA vs Q Learning

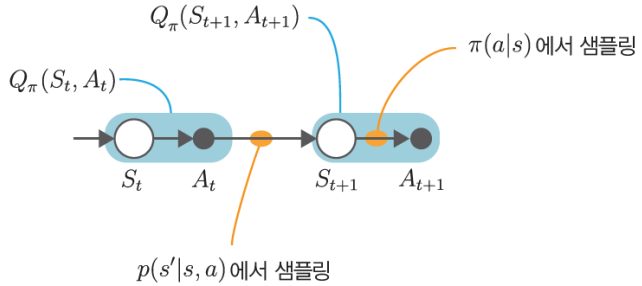
SARSA

(Bellman equation)

$$q_\pi(s,a) = \sum_{s'} p(s' \mid s,a)\left\{r(s,a,s') + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s',a')\right\}$$



(Sampling version)

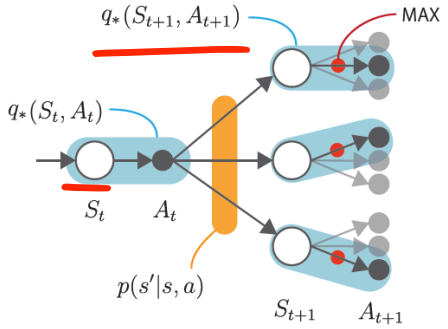


$$Q'_\pi(S_t,A_t) = Q_\pi(S_t,A_t) + \alpha\{R_t + \gamma Q_\pi(S_{t+1},A_{t+1}) - Q_\pi(S_t,A_t)\}$$

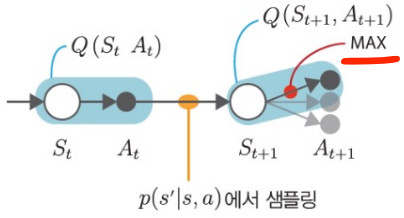
Q Learning

(Bellman optimality equation)

$$q_*(s,a) = \sum_{s'} p(s' \mid s,a)\left\{r(s,a,s') + \gamma \max_{a'} q_*(s',a')\right\}$$



(Sampling version)

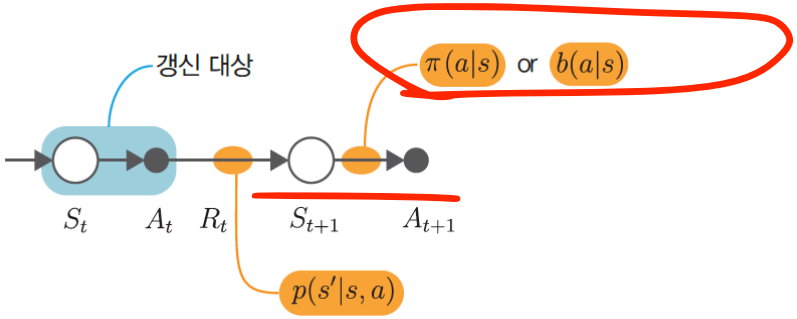


$$Q'(S_t,A_t) = Q(S_t,A_t) + \alpha\left\{R_t + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\right\}$$

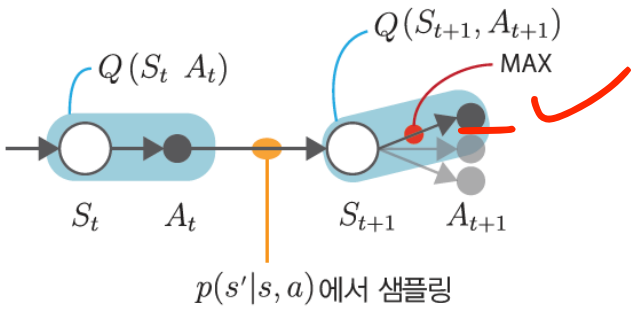
Q Learning

- Off-Policy
  - SARSA
    - A_{t+1} 을 확률분포 π 또는 b 에서 샘플링
    - 중요도 샘플링 필요, Q 갱신 불안정
  - Q Learning
    - A_{t+1} 을 Q 함수가 가장 큰 (MAX) 행동으로 선택
    - 중요도 샘플링 불필요, Q 갱신 안정
    - 최적화 성능 향상

(SARSA)



(Q Learning)



Q Learning

- Implementation

```python
class QLearningAgent:
    def __init__(self):
        self.gamma = 0.9
        self.alpha = 0.8
        self.epsilon = 0.1
        self.action_size = 4

        random_actions = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}

        self.b = defaultdict(lambda: random_actions)  # 행동 정책
        self.Q = defaultdict(lambda: 0)

    def get_action(self, state):
        action_probs = self.b[state]  # 행동 정책에서 가져옴
        actions = list(action_probs.keys())
        probs = list(action_probs.values())
        return np.random.choice(actions, p=probs)
```

```python
def update(self, state, action, reward, next_state, done):
    if done:  # 목표에 도달
        next_q_max = 0
    else:     # 그 외에는 다음 상태에서 Q 함수의 최댓값 계산
        next_qs = [self.Q[next_state, a] for a in range(self.action_size)]
        next_q_max = max(next_qs)

    # Q 함수 갱신
    target = reward + self.gamma * next_q_max
    self.Q[state, action] += (target - self.Q[state, action]) * self.alpha

    # 행동 정책            갱신
    self.b[state] = greedy_probs(self.Q, state, self.epsilon)
```

$$Q'(S_t,A_t) = Q(S_t,A_t) + \alpha\{R_t + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\}$$

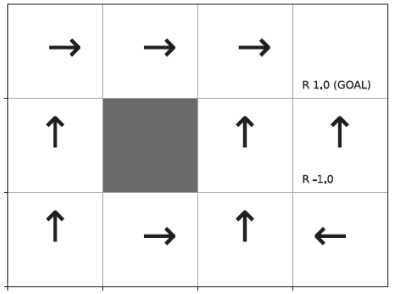
(off-policy SARSA)

```python
def update(self, state, action, reward, done):
    self.memory.append((state, action, reward, done))
    if len(self.memory) < 2:
        return

    state, action, reward, done = self.memory[0]
    next_state, next_action, _, _ = self.memory[1]

    if done:
        next_q = 0
        rho = 1
    else:
        next_q = self.Q[next_state, next_action]
        # ❷ 가중치 rho 계산
        rho = self.pi[next_state][next_action] / self.b[next_state][next_action]

    # ❸ rho로 TD 목표 보정
    target = rho * (reward + self.gamma * next_q)
    self.Q[state, action] += (target - self.Q[state, action]) * self.alpha

    # ❹ 각각의 정책 개선
    self.pi[state] = greedy_probs(self.Q, state, 0)
    self.b[state] = greedy_probs(self.Q, state, self.epsilon)
```

Q Learning

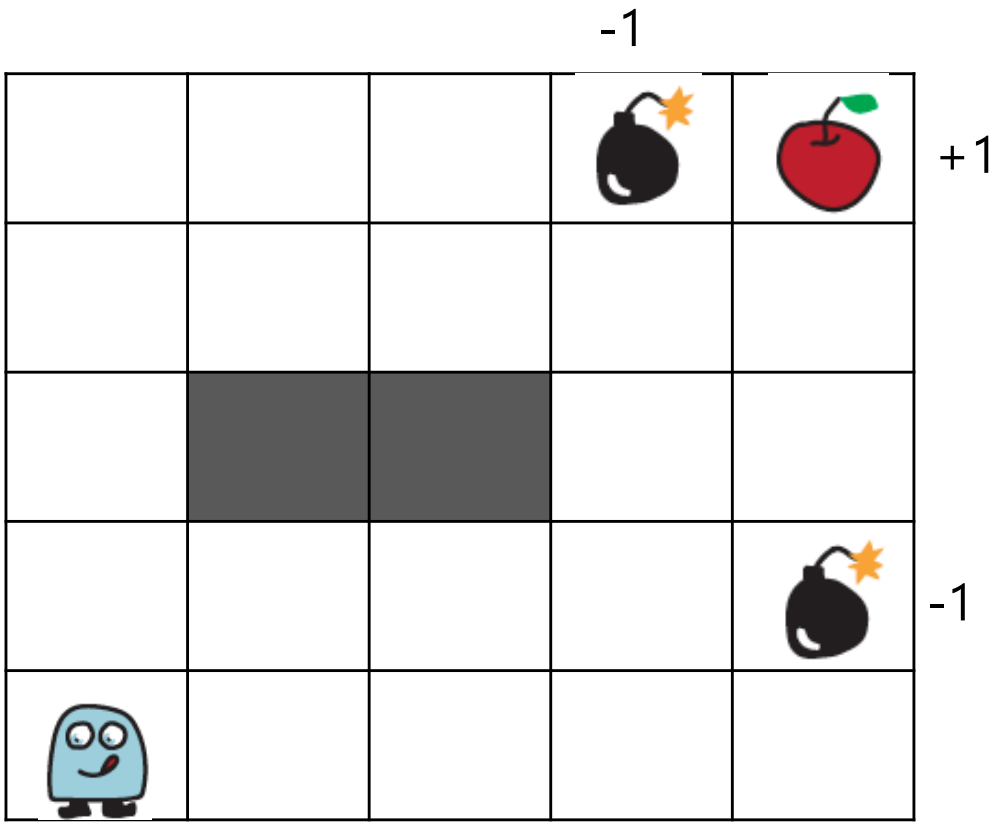
실습 #3 q_learning.py

```python
from collections import defaultdict
import numpy as np
from common.gridworld import GridWorld
from common.utils import greedy_probs

class QLearningAgent:
    def __init__(self):
        self.gamma = 0.9
        self.alpha = 0.8
        self.epsilon = 0.1
        self.action_size = 4

        random_actions = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}

        self.b = defaultdict(lambda: random_actions)  # 행동 정책
        self.Q = defaultdict(lambda: 0)

    def get_action(self, state):
        action_probs = self.b[state]  # 행동 정책에서 가져옴
        actions = list(action_probs.keys())
        probs = list(action_probs.values())
        return np.random.choice(actions, p=probs)

    def update(self, state, action, reward, next_state, done):
        if done:  # 목표에 도달
            next_q_max = 0
        else:     # 그 외에는 다음 상태에서 Q 함수의 최댓값 계산
            next_qs = [self.Q[next_state, a] for a in range(self.action_size)]
            next_q_max = max(next_qs)

        # Q 함수 갱신
        target = reward + self.gamma * next_q_max
        self.Q[state, action] += (target - self.Q[state, action]) * self.alpha

        # 행동 정책:              갱신

        self.b[state] = greedy_probs(self.Q, state, self.epsilon)

env = GridWorld()
agent = QLearningAgent()

episodes = 10000
for episode in range(episodes):
    state = env.reset()

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

        agent.update(state, action, reward, next_state, done)
        if done:
            break
        state = next_state

# 시각화
env.render_q(agent.Q)
```



Quiz

(Q) Q Learning 을 적용하여 5x5 Grid World 에 대한 value function 및 policy 를 구하라.



Q Learning

- 요약
  - 상태 (state) 와 행동 (action) 에 대해 Q-value 를 업데이트 하면서, 보상을 최대화 하는 정책(policy) 을 학습
    - TD 법으로 가치함수 평가
      - MC 와 같이 샘플링 데이터 기반의 가치함수 평가
      - '지금' 과 '다음' 정보만 사용
    - Bellman optimality equation

$$Q'(S_t, A_t) = Q(S_t, A_t) + \alpha\{R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\}$$

    - Off-Policy, but 중요도 샘플링 사용하지 않음

⇒ Q 함수를 효율적이고 안정적으로 갱신

⇒ 정책 결정의 최적화 성능 향상

⇒ 강화학습의 핵심기법으로 많은 연구의 기반 알고리즘으로 사용

Q Learning

- History
  - 1989, Q-Learning 등장
    - Watkins & Dayan, "Q-Learning", Machine Learning
    - Q-table 학습
  - 1990~2000년대 초반: 알고리즘 확장
    - SARSA (On-policy), Dyna-Q 등
    - 게임, 로보틱스 등에서 실험적으로 사용
  - 2013, Deep Q-Network (DQN) 등장
    - Deep Mind (Mnih et al.), "Playing Atari with Deep Reinforcement Learning", 2013 (arXiv)
    - Q-table 대신 신경망으로 Q-value 를 근사
    - 인간 수준의 Atari 게임 플레이
  - 최근, 다양한 DQN 변형 등장
    - Double DQN, Dueling DQN, Rainbow DQN .....