Monte Carlo Method

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

Value Function

- Value Function

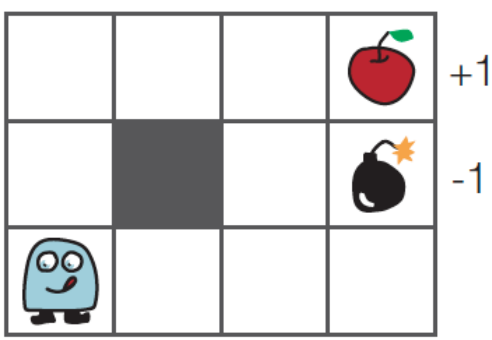
  policy

  action

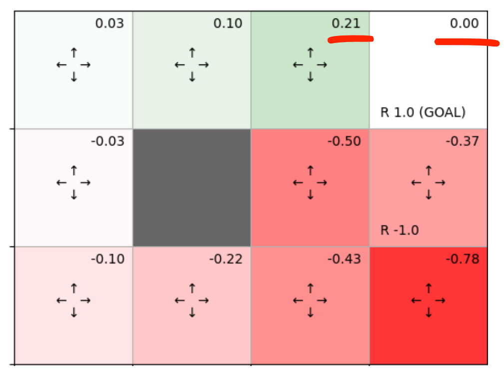
  – State value : $v_\pi(s) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots | S_t = s]$ γ: discount rate 0-1

  – Action value : $q_\pi(s,a) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots | S_t = s, A_t = a]$ Q function

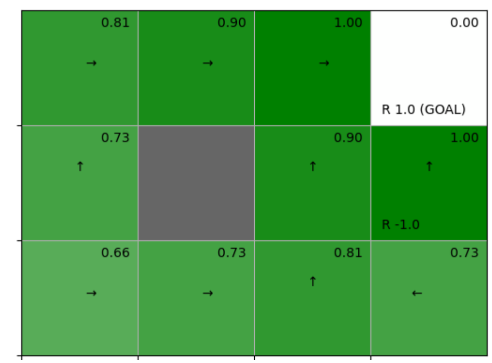
  현재



agent, environment, state, reward, action



(state-value)

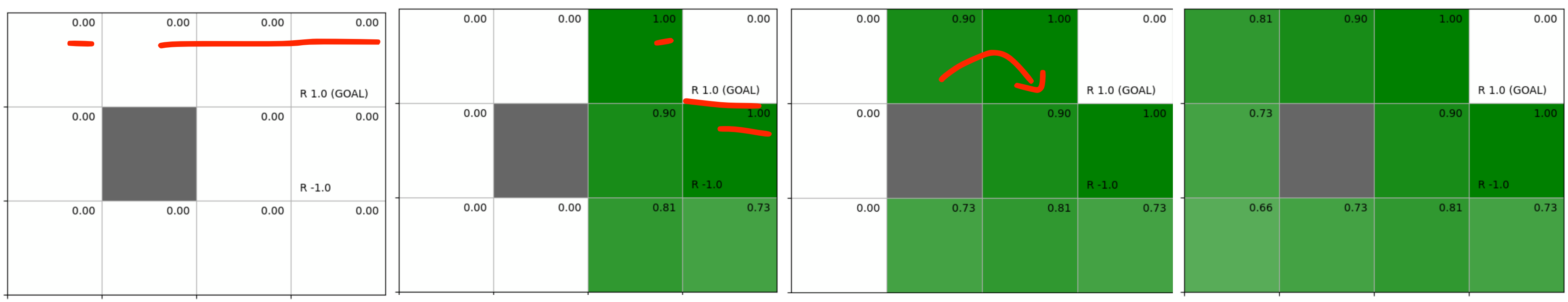


(action-value / Q table)

$v_\pi(s)$ 1 13

Value Function

- Dynamic Programming
  - Calculation of state-value and action value by Bellman equation
  - Iterative evaluation $V_{k+1}(s) = \sum_{a,s'} \pi(a \mid s)\, p(s' \mid s,a) \{ r(s,a,s') + \gamma V_k(s') \}$



$k = 0$ $k = 1$ $k = 2$ $k = 3$

확률 모델

⇒ 많은 계산량, 환경 모델 필요: $p(s' | s, a)$, $r(s, a, s')$

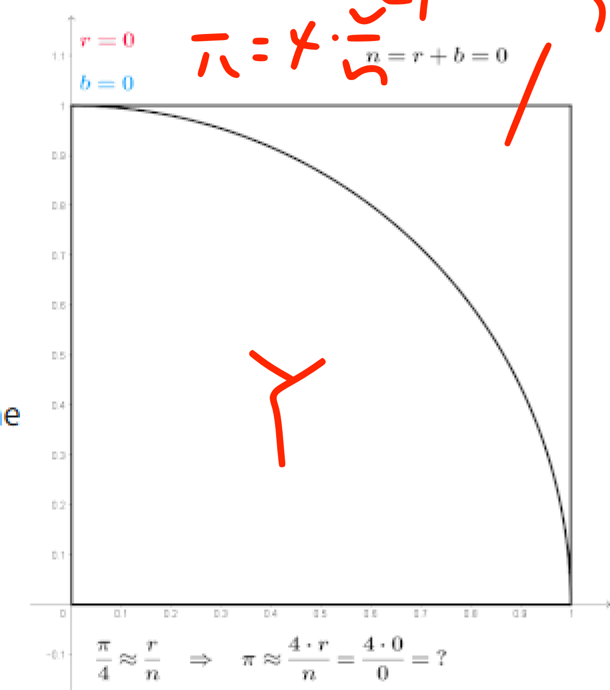
⇒ 복잡한 환경 및 환경 모델을 알 수 없는 문제에 적용 어려움

Basics

- Monte Carlo Method
  - (정의) Computational algorithms that rely on repeated random sampling
  - (어원) Monaco 의 Monte Carlo Casino

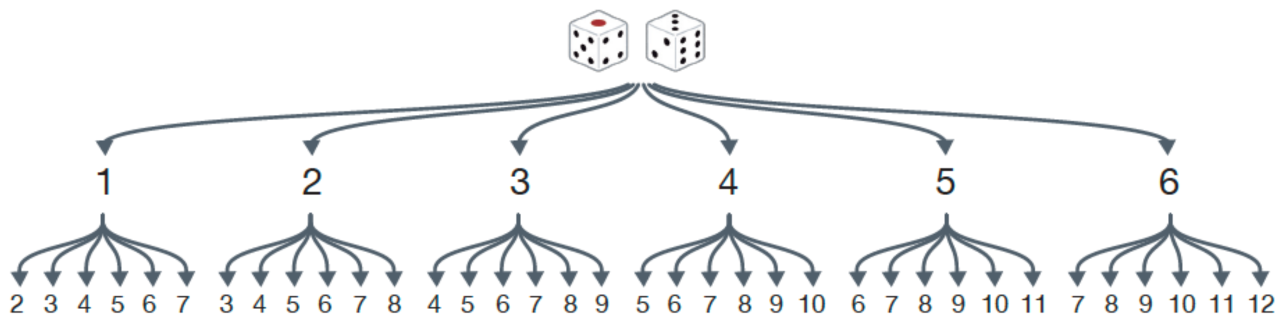
(ex) 원주율 π 값 구하기 (Wikipedia)

1. Draw a square, then inscribe a quadrant within it.
2. Uniformly scatter a given number of points over the square.
3. Count the number of points inside the quadrant, i.e. having a distance from the origin of less than 1.
4. The ratio of the inside-count and the total-sample-count is an estimate of the ratio of the two areas, $\frac{\pi}{4}$. Multiply the result by 4 to estimate π.



Basics

- 주사위 눈의 합
  - 주사위 두 개를 던져서 나오는 눈의 합



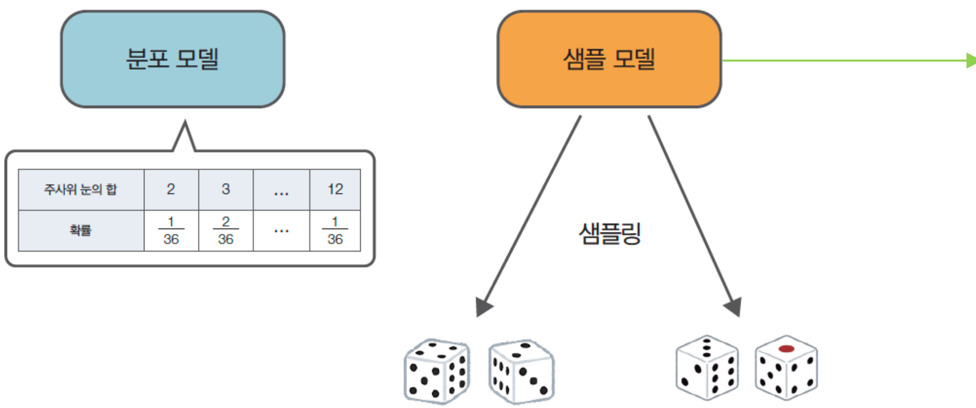
  - Probability distribution

주사위 눈의 합	2	3	4	5	6	7	8	9	10	11	12
확률	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

  - Expectation value = 7.0

Basics

- 주사위 눈의 합 모델
  - Distribution model (분포모델)
  - Sample model (샘플모델)



주사위 눈의 합	2	3	…	12
확률	$\frac{1}{36}$	$\frac{2}{36}$	…	$\frac{1}{36}$

```python
import numpy as np

def sample(dices=2):
    x = 0
    for _ in range(dices):
        x += np.random.choice([1, 2, 3, 4, 5, 6])
    return x
```

```python
print(sample())  # [출력 결과] 10
print(sample())  # [출력 결과] 4
print(sample())  # [출력 결과] 8
```

Basics

- 주사위 눈의 합 – Monte Carlo Method
  - $V_n = \frac{s_1+s_2+\cdots+s_n}{n}$

```
trial = 1000  # 샘플링 횟수

samples = []
for _ in range(trial):  # 샘플링
    s = sample()
    samples.append(s)

V = sum(samples) / len(samples)  # 평균 계산
print(V)
```

```
6.98
```

  - $V_n = V_{n-1} + \frac{1}{n}(s_n - V_{n-1})$

```
trial = 1000
V, n = 0, 0

for _ in range(trial):
    s = sample()
    n += 1
    V += (s - V) / n  # 또는 V = V + (s - V) / n
    print(V)
```

```
4.0
6.0
5.333333333333333
...
6.959959959959965
6.960000000000005
```

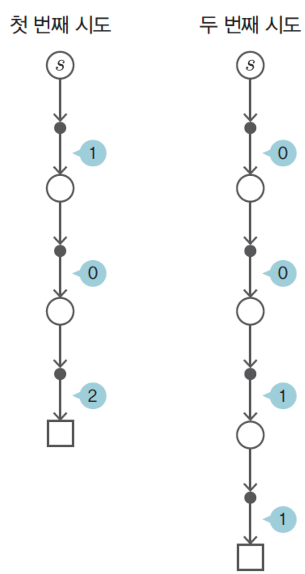
State-Value Function

- State-value function
  - (Definition) $v_\pi(s) = \mathbb{E}[G \mid s]$

    정책 (policy) π 에 따라 행동 (action) 하는 경우, 상태 (state) s 에서 출발하여 얻을 수 있는 기대 수익 (return)

  - Monte Carlo method

$$\boxed{V_\pi(s) = \frac{G^{(1)} + G^{(2)} + \cdots + G^{(n)}}{n}}$$



$$G^{(1)} = 1 + 0 + 2 = 3$$

$$G^{(2)} = 0 + 0 + 1 + 1 = 2$$

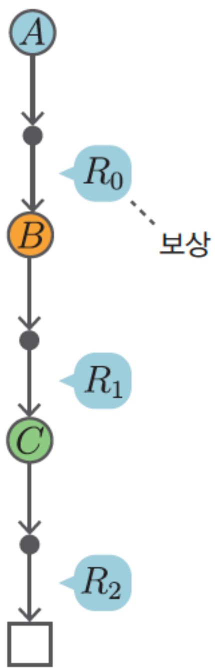
$$\frac{G^{(1)} + G^{(2)}}{2} = \frac{3+2}{2} = 2.5$$

(cf) dynamic programming

$$V_{k+1}(s) = \sum_{a,s'} \pi(a \mid s)\, p(s' \mid s,a) \{ r(s,a,s') + \gamma V_k(s') \}$$

State-Value Function

- State-value function for all states
  - Considering the computational efficiency



1) Inefficient case

$$G_A = R_0 + \gamma R_1 + \gamma^2 R_2$$
$$G_B = R_1 + \gamma R_2$$
$$G_C = R_2$$

2) Efficient case

$$G_A = R_0 + \gamma G_B$$
$$G_B = R_1 + \gamma G_C$$
$$G_C = R_2$$

$\Rightarrow$

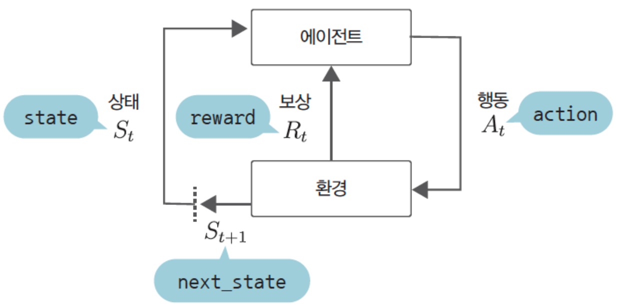
$$G_C = R_2$$
$$G_B = R_1 + \gamma G_C$$
$$G_A = R_0 + \gamma G_B$$

역방향 (reversed)계산

Policy Evaluation

- GridWorld 클래스
  - step()

```python
def step(self, action):
    state = self.agent_state
    next_state = self.next_state(state, action)
    reward = self.reward(state, action, next_state)
    done = (next_state == self.goal_state)

    self.agent_state = next_state
    return next_state, reward, done
```



  - reset()

```python
def reset(self):
    self.agent_state = self.start_state
    return self.agent_state
```

(ex)

```python
from common.gridworld import GridWorld

env = GridWorld()
action = 0  # 더미 행동
next_state, reward, done = env.step(action)  # 행동 수행

print('next_state:', next_state)
print('reward:', reward)
print('done:', done)
```

```
next_state: (1, 0)
reward: 0.0
done: False
```

Policy Evaluation

- RandomAgent 클래스

```python
class RandomAgent:
    def __init__(self):
        self.gamma = 0.9
        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)
        self.cnts = defaultdict(lambda: 0)   # count (n)
        self.memory = []                     # (state, action, reward)

    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 add(self, state, action, reward):
        data = (state, action, reward)
        self.memory.append(data)

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

    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]
```

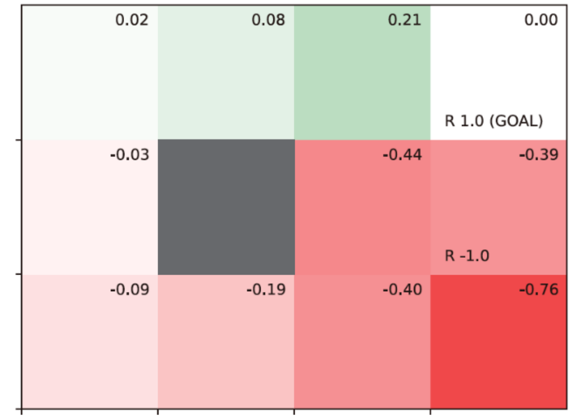
```python
# agent.memory
[(S0, A0, R0), (S1, A1, R1), ..., (S8, A8, R8)]
```

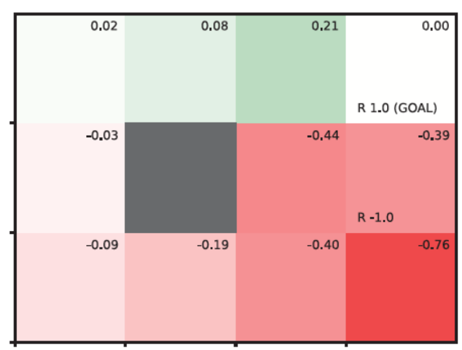
$$V_n = V_{n-1} + \frac{1}{n}(s_n - V_{n-1})$$

Policy Evaluation

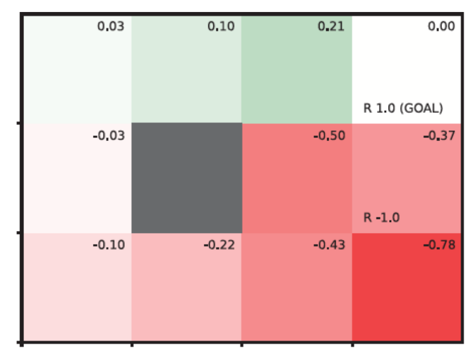
- Monte Carlo Method

```python
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)
```





Monte Carlo



Dynamic Programming

Policy Evaluation

실습 #1 mc_eval.py

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


class RandomAgent:
    def __init__(self):
        self.gamma = 0.9
        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)
        self.cnts = defaultdict(lambda: 0)
        self.memory = []

    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 add(self, state, action, reward):
        data = (state, action, reward)
        self.memory.append(data)

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

    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]
```

몬테카를로

```python
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

# 몬테카를로법으로 얻은 가치 함수
env.render_v(agent.V)
```

Policy Control

- Optimal policy

state s

에서의 action

$$\mu(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$$

$$= \underset{a}{\operatorname{argmax}} \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma V(s')\}$$

  - Monte Carlo method
    - state-value function evaluation
      - 일반적인 방식: $V_n(s) = \frac{G^{(1)} + G^{(2)} + \cdots + G^{(n)}}{n}$
      - 증분 방식: $V_n(s) = V_{n-1}(s) + \frac{1}{n}\{G^{(n)} - V_{n-1}(s)\}$
    - Q function evaluation
      - 일반적인 방식: $Q_n(s,a) = \frac{G^{(1)} + G^{(2)} + \cdots + G^{(n)}}{n}$
      - 증분 방식: $Q_n(s,a) = Q_{n-1}(s,a) + \frac{1}{n}\{G^{(n)} - Q_{n-1}(s,a)\}$

$G^{(n)}$: n 번째 에피소드에서 얻을 수 있는 수익

Policy Control

- McAgent 클래스

```python
class McAgent:
    def __init__(self):
        self.gamma = 0.9
        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)  # V가 아닌 Q를 사용
        self.cnts = defaultdict(lambda: 0)
        self.memory = []

    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 add(self, state, action, reward):
        data = (state, action, reward)
        self.memory.append(data)

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

Policy Control

- update()
  - 가치함수 Q 를 갱신하고, 행동 μ 를 구하여 정책 π 를 찾는다

```python
def greedy_probs(Q, state, action_size=4):
    qs = [Q[(state, action)] for action in range(action_size)]
    max_action = np.argmax(qs)

    action_probs = {action: 0.0 for action in range(action_size)}
    # 이 시점에서 action_probs는 {0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0}이 됨
    action_probs[max_action] = 1  # ❶
    return action_probs  # 탐욕 행동을 취하는 확률 분포 반환

class McAgent:
    ...
    def update(self):
        G = 0
        for data in reversed(self.memory):
            state, action, reward = data
            G = self.gamma * G + reward
            key = (state, action)
            self.cnts[key] += 1
            # [식 5.5]에 따라 self.Q 갱신
            self.Q[key] += (G - self.Q[key]) / self.cnts[key]  # ❷

            # state의 정책 탐욕화
            self.pi[state] = greedy_probs(self.Q, state)
```

$$\mu(s) = \underset{a}{\operatorname{argmax}}\, \boxed{Q(s,a)}$$

$$\pi(s|a)$$

$$Q_n(s,a) = Q_{n-1}(s,a) + \frac{1}{n}\{G^{(n)} - Q_{n-1}(s,a)\}$$

Policy Control

- ε-Greedy policy
  - Exploitation (활용) & exploration (탐색)

```python
def greedy_probs(Q, state, action_size=4):
    qs = [Q[(state, action)] for action in range(action_size)]
    max_action = np.argmax(qs)

    action_probs = {action: 0.0 for action in range(action_size)}
    # 이 시점에서 action_probs는 {0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0}이 됨
    action_probs[max_action] = 1  # ❶
    return action_probs  # 탐욕 행동을 취하는 확률 분포 반환
```

if max_action=1

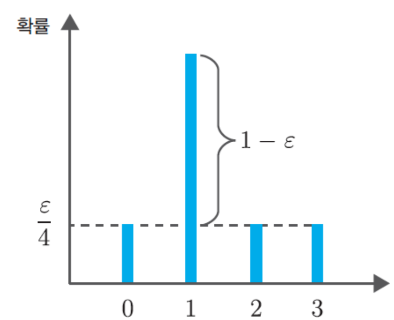
action_probs ={0: 0.0, 1: 1.0, 2: 0.0, 3: 0.0}

```python
def greedy_probs(Q, state, epsilon=0, action_size=4):
    qs = [Q[(state, action)] for action in range(action_size)]
    max_action = np.argmax(qs)

    base_prob = epsilon / action_size
    action_probs = {action: base_prob for action in range(action_size)}
    # 이 시점에서 action_probs = {0: ε/4, 1: ε/4, 2: ε/4, 3: ε/4}
    action_probs[max_action] += (1 - epsilon)
    return action_probs
```

- 무작위성을 추가하여 다양한 방향의 탐색을 가능하게 함

if max_action=1, and ε=0.4

action_probs ={0: 0.1, 1: 0.7, 2: 0.1, 3: 0.1}



Policy Control

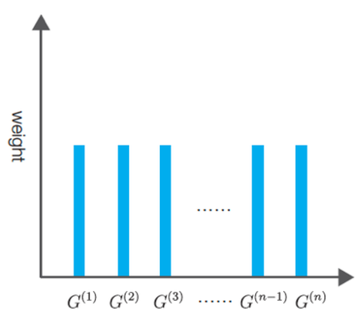
- Exponential moving average

```
def update(self):
    G = 0
    for data in reversed(self.memory):
        state, action, reward = data
        G = self.gamma * G + reward
        key = (state, action)
        self.cnts[key] += 1
        # [식 5.5]에 따라 self.Q 갱신
        self.Q[key] += (G - self.Q[key]) / self.cnts[key]  # ❷

        # state의 정책 탐욕화
        self.pi[state] = greedy_probs(self.Q, state)
```

```
# 수정 후
alpha = 0.1
self.Q[key] += (g - self.Q[key]) * alpha  # ❷
```

$$Q_n(s,a) = Q_{n-1}(s,a) + \frac{1}{n}\{G^{(n)} - Q_{n-1}(s,a)\}$$

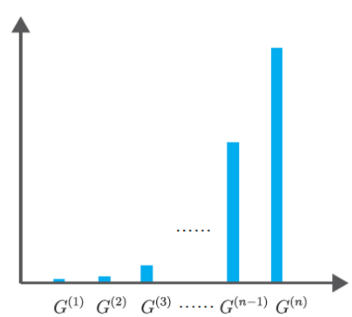
⇨ $G^{(1)}, G^{(2)}, \cdots, G^{(n)}$ 의 가중치 (weight) 가 $\frac{1}{n}$ 로 같음



$$Q_n(s,a) = Q_{n-1}(s,a) + \alpha\{G^{(n)} - Q_{n-1}(s,a)\}$$

⇨ $G^{(1)}, G^{(2)}, \cdots, G^{(n)}$ 의 가중치가
$\alpha^{n-1}, \alpha^{n-2}, \cdots, \alpha$ 로 exponential 하게 증가

⇨ 최신 데이터의 가중치를 크게 함



Policy Control

- Monte Carlo Method

(policy evaluation)

```python
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 control)

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

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

        state = next_state

env.render_q(agent.Q)
```

Policy Control

실습 #2 mc_control.py

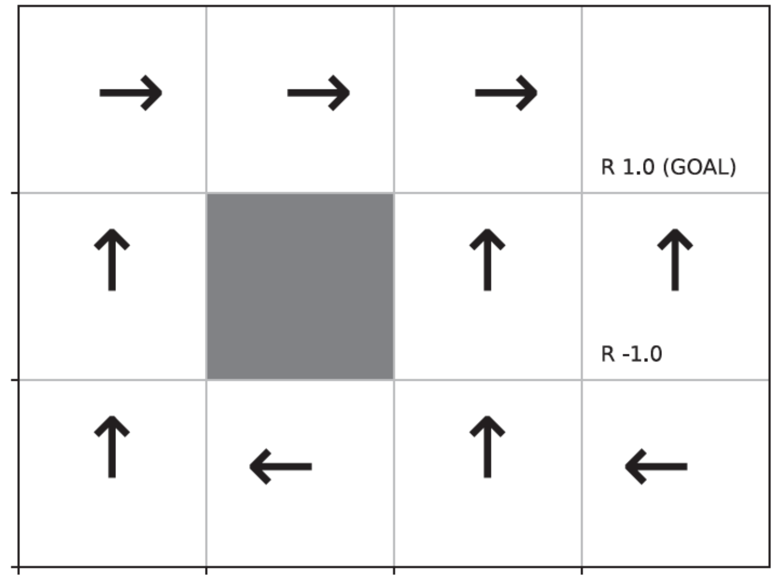
```python
import numpy as np
from collections import defaultdict
from common.gridworld import GridWorld

def greedy_probs(Q, state, epsilon=0, action_size=4):
    qs = [Q[(state, action)] for action in range(action_size)]
    max_action = np.argmax(qs)

    base_prob = epsilon / action_size
    action_probs = {action: base_prob for action in range(action_size)}
    action_probs[max_action] += (1 - epsilon)
    return action_probs

class McAgent:
    def __init__(self):
        self.gamma = 0.9
        self.epsilon = 0.1  # (첫 번째 개선) ε-탐욕 정책의 ε
        self.alpha = 0.1    # (두 번째 개선) Q 함수 갱신 시의 고정값 α
        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.cnts = defaultdict(lambda: 0)
        self.memory = []

    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 add(self, state, action, reward):
        data = (state, action, reward)
        self.memory.append(data)

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

```python
    def update(self):
        G = 0
        for data in reversed(self.memory):
            state, action, reward = data
            G = self.gamma * G + reward
            key = (state, action)
            # self.cnts[key] += 1
            # self.Q[key] += (G - self.Q[key]) / self.cnts[key]
            self.Q[key] += (G - self.Q[key]) * self.alpha
            self.pi[state] = greedy_probs(self.Q, state, self.epsilon)

env = GridWorld()
agent = McAgent()

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

        state = next_state
#
env.render_q(agent.Q)
```

Policy Control

실습 #2

Q 함수 시각화



Q 함수에 대한 greedy policy

요약

- Monte Carlo Method 80 20%
  - 환경 모델 없이 정책을 평가하고 제어할 수 있음
    - Q 함수 평가 Q V π
    - ε-greedy
  - 에피소드가 끝나야 정책을 평가할 수 있음
    - 일회성 과제에만 적용 가능
    - 지속성 과제에는 적용 어려움

Quiz

(Q) Monte Carlo Method 를 적용하여 5x5 Grid World 에 대한 value function 및 policy 를 구하라.