Chapter 5 몬테카를로법

4장에서는 동적 프로그래밍을 이용하여 최적 가치 함수와 최적 정책을 찾았다. DP를 이용하려면 '환경 모델(상태 전이 확률과 보상 함수')를 알고 있어야 한다. 하지만, 실제 세상에서는 환경 모델을 알 수 없는 문제도 많다. 혹은 알더라도 계산량이 너무 많아 풀 수 없을 때도 많다. 이처럼 환경 모델을 알 수 없는 상황에서 문제를 풀려면 에이전트가 실제로 행동하여 얻은 경험을 토대로 학습해야 한다.

이것이 몬테카를로법이다. 데이터를 반복적으로 샘플링하여 그 결과를 토대로 추정하는 방법을 일컫는다. 강화 학습에서는 몬테카를로법을 통해 경험으로부터 가치 함수를 추정할 수 있다. 이 목표가 달성되면 최적 정책을 찾을 수 있다.

5.1. 몬테카를로법 기초

지금까지는 환경 모델이 알려진 문제를 다뤘다. 그리드 월드 문제에서는 에이전트의 행동에 따라 다음 상태와 보상이 명확했다. 수식으로 표현하면 상태 전이 확률 $p(s'|s,a)$와 보상 함수 $r(s,a,s')$를 이용할 수 있었다. 이처럼 환경 모델이 알려진 문제에서는 에이전트 측에서 '상태, 행동, 보상'의 전이를 시뮬레이션할 수 있다.

하지만, 현실에서는 환경 모델을 알 수 없는 문제가 많다. 예를 들어 '상품 재고 관리' 문제에서 '상품이 얼마나 팔릴 것인가'가 환경 상태 전이 확률에 해당한다. 그런데 상품 판매량은 여러 요인이 복잡하게 얽혀 결정되기 때문에 완벽하게 알아내기가 현실적으로 불가능하다.

또한, 상태 전이 확률을 이론적으로 알 수 있더라도 계산량이 너무 많은 경우도 있다.

5.1.1. 주사위 눈의 합

주사위 두개를 굴리는 문제이다. 각 눈이 나올 확률은 정확하게 1/6이라고 가정한다. 이때 주사위 눈의 합을 확률 분포로 표현할 수 있다.

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

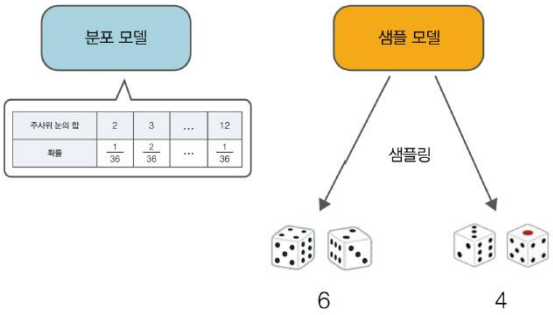
이 확률 분포를 이용하여 기댓값을 계산할 수 있다. 코드로 구현하면 다음과 같다.

```
ps = {2: 1/36, 3: 2/36, 4: 3/36, 5: 4/36, 6: 5/36, 7: 6/36,
      8: 5/36, 9: 4/36, 10: 3/36, 11: 2/36, 12: 1/36}

V = 0
for x, p in ps.items():
    V += x*p
print(V)  # [출력 결과] 6.999999999999999
```

이와 같이 확률 분포를 알면 기댓값을 계산할 수 있다.

5.1.2. 분포 모델과 샘플 모델

주사위 눈의 합처럼 확률 분포로 표현된 모델을 분포 모델이라고 한다. 샘플 모델이란 방법도 있다. 샘플 모델이이란 '표본을 추출(샘플링) 할 수만 있으면 충분하다'라는 모델이다. 주사위를 예로 들면 주사위를 실제로 굴려서 (=샘플링해서) 나온 눈의 합을 관찰하는 방법이다.



그림의 샘플 모델에서는 6이나 4와 같은 구체적인 샘플 데이터를 얻을 수 있다. 샘플 모델에서는 전체의 확률 분포는 필요하지 않으며 단순히 샘플링만 할 수 있으면 충분하다. 다만, 샘플링을 무한히 반복하면 그 분포가 곧 확률 분포와 같아진다.

파이썬으로 구현할 수 있다.

```
# 샘플모델 주사위 던지기
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
```

```
print(sample())
print(sample())
print(sample())
```

```
4
4
2
```

주사위를 랜덤으로 두 번 굴리는 함수를 만들고 실행하면, 실행할때마다 결과가 달라진다. 샘플 모델로 기댓값을 계산하려면 샘플링을 많이 하고 평균을 구하면 된다. 이 방법이 바로 몬테카를로법이다. 숫자를 평균내지만 표본 수를 무한대로 늘리면 그 평균이 참 값으로 수렴한다.

5.1.3. 몬테카를로법 구현

몬테카를로법으로 기댓값을 구할 수 있다.

```
# 몬테카를로법으로 기댓값 구현
trial = 1000 # 샘플링 횟수

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

V = sum(samples) / len(samples) # 평균을 계산하여 가치함수 구하기
print(V)
```

```
6.831
```

샘플링을 1000번 수행하여 평균을 구했다. 실행 결과를 보면 정답인 7과 얼추 맞는 값이라고 할 수 있다. 이어서 샘플 데이터를 얻을 때마다 평균을 구할수도 있다.

```
# 샘플 데이터를 얻을 때마다 평균 구하기
trial = 1000

samples = []
for _ in range(trial):
    s = sample()
    samples.append(s)
    V = sum(samples) / len(samples)
    print(V)
```

```
5.0
4.0
5.333333333333333
5.0
4.6
4.666666666666667
5.285714285714286
5.125
5.222222222222222
```

앞에서와 마찬가지로 샘플 데이터를 추가하고, 그 리스트에서 평균을 구한다. 더 효율적으로 증분 구현 방법을 떠올릴 수 있다.

〈일반적인 방식〉 $$V_n = \frac{s_1 + s_2 + \cdots + s_n}{n}$$

〈증분 방식〉 $$V_n = V_{n-1} + \frac{1}{n}(s_n - V_{n-1})$$

```
# 증분 구현 방식으로 기댓값 구하기
trial = 1000
V, n = 0,0

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

```
7.0
9.5
9.0
8.75
9.0
9.5
9.0
```

5.2. 몬테카를로법으로 정책 평가하기

몬테카를로법은 실제로 샘플링 하고 샘플 데이터로부터 기댓값을 계산한다. 이 방법을 강화 학습에 적용하면 에이전트가 실제로 행동하여 얻은 경험으로 가치 함수를 추정할 수 있다. 이번 절에서는 정책 π가 주어졌을 때, 그 정책의 가치 함수를 몬테카를로법으로 계산한다.

5.2.1. 가치 함수를 몬테카를로법으로 구하기

가치 함수는 다음 식으로 표현된다.

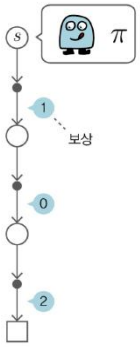
$$v_\pi(s) = \mathbb{E}_\pi[G|s]$$

상태 s에서 출발하여 얻을 수 있는 수익을 G로 나타냈다. 가치 함수 $v_\pi(s)$는 정책 π에 따라 행동했을 때 얻을 수 있는 기대 수익으로 정의된다.

위 식의 가치 함수를 몬테카를로법으로 계산할 수 있다. 이를 위해 에이전트에게 정책 π에 따라 실제로 행동을 취하도록 한다. 이렇게 해서 얻는 실제 수익이 샘플 데이터이고, 이런 샘플 데이터를 많이 모아서 평균을 구하는 것이 몬테카를로법이다. 수식으로는 다음과 같다.

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

몬테카를로법으로 계산하려면 식과 같이 에피소드를 n번 수행하여 얻은 샘플 데이터의 평균을 구하면 된다. (몬테카를로법은 일회성 과제에서만 이용할 수 있다.)

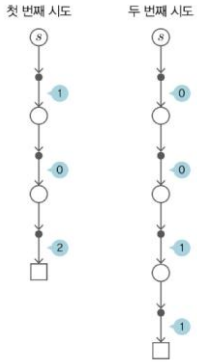


그림을 보면 에이전트가 상태 s에서 출발하여 정책 π에 따라 행동한 결과를 나타낸다. 얻은 보상이 1, 0, 2라고 가정하고, 할인율 γ을 1로 가정하면 상태 s에서의 수익을 다음과 같이 구할 수 있다.

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

이것이 첫 번째 샘플 데이터이다. 이 시점에서의 가치 함수는 3으로 추정할 수 있다.

이어서 두 번째 시도가 그림처럼 진행됐다고 해보자.

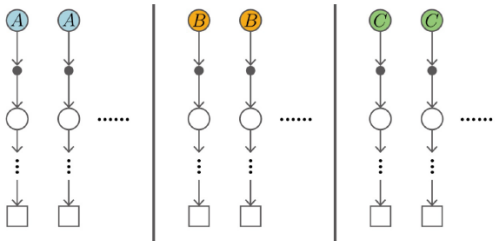


이번에는 상태 s에서 시작하여 0, 0, 1, 1의 보상을 얻었다. 에이전트의 정책이 확률적이거나 환경의 상태 전이가 확률적이기 때문에 보상의 액수가 달라질 수 있다. 두 번째 에피소드에서 수익은 0+0+1+1 = 2와 같다.

1차 수익이 3이고, 2차 수익이 2이므로 평균은 2.5이다. 이 시점의 가치 함수 $V_\pi(s)$는 2.5가 된다. 이처럼 실제로 행동하여 수익의 평균을 구함으로써 가치 함수를 근사할 수 있다. 그리고 시도 횟수를 늘리면 정확도가 높아진다.

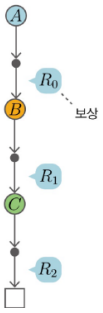
5.2.2. 모든 상태의 가치 함수 구하기

이번에는 모든 상태에서의 가치 함수를 구할 수 있다. 단순하게 생각하면 시작 상태를 바꿔가며 앞의 과정을 반복하면 된다. 예를 들어 상태가 총 세 가지 (A,B,C)라면 각 상태의 가치 함수를 그림처럼 구할 수 있다.



각 상태에서부터 출발하여 실제로 행동을 수행하고 샘플 데이터를 수집한다. 그런 다음 각 상태에서의 수익을 평균하면 가치 함수를 구할 수 있다. 하지만 각 상태의 가치 함수를 독립적으로 구한다는 점에서 계산 효

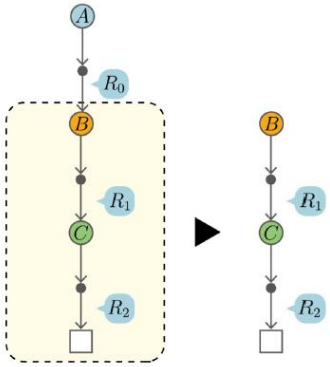
율이 떨어진다. 상태 A에서 시작하여 얻은 수익은 $V_\pi(A)$를 계산하는 데만 사용되며, 다른 가치 함수 계산에는 기여하지 않는다.



그림은 상태 A에서 출발하여 정책 π에 따라 행동한 결과이다. A,B,C 순서로 상태를 거쳐 목표에 도달했다고 가정했다. 이 과제의 할인율을 γ이라고 하면 상태 A에서 출발하여 얻는 수익은 다음과 같다.

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

다음으로 상태 B로부터 아래로의 전이를 볼 수 있다.



상태 B에서 시작했을 때의 수익은 $G_B = R_1 + \gamma R_2$와 같다. 마찬가지로 상태 C에서 시작했을 때의 수익은 $G_C = R_2$와 같다. 이처럼 한번의 시도 만으로 세 가지 상태에 대한 수익을 얻었다.

5.2.3. 몬테카를로법 계산 효율 개선

마지막으로 수익을 효율적으로 계산하는 방법에 대해 더 알아볼 수 있다. 그림의 예에서 우리는 다음 세 가지 수익을 계산해야 한다.

$$\begin{aligned} G_A &= R_0 + \gamma R_1 + \gamma^2 R_2 & G_A &= R_0 + \gamma G_B \\ G_B &= R_1 + \gamma R_2 & G_B &= R_1 + \gamma G_C \\ G_C &= R_2 & G_C &= R_2 \end{aligned}$$

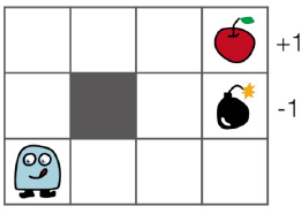
주목할 곳은 G_A 계산에 G_B를 이용한다. 마찬가지로 G_B 계산에는 G_C를 이용했다. 이 패턴을 응용해서 뒤에서부터 계산하면 된다.

$$\begin{aligned} G_C &= R_2 \\ G_B &= R_1 + \gamma G_C \\ G_A &= R_0 + \gamma G_B \end{aligned}$$

이와 같이 G_C를 구한다. 후에 그 다음 G_B, G_A를 구하면 중복 계산이 사라진다.

5.3. 몬테카를로법 구현

'3*4 그리드 월드' 문제를 몬테카를로법으로 풀 수 있다.



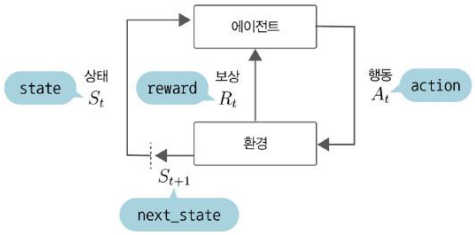
이번에는 환경 모델(상태 전이 확률과 보상 함수)를 이용하지 않고 정책을 평가한다.

5.3.1. step() 메서드

GridWorld 클래스에는 에이전트에게 행동을 시키는 step() 메서드가 있ㄸ싸.

```
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)
```

step() 메서드는 행동을 매개변수로 받는다. 호출하면 현재 환경에서 행동을 수행하고, 그 결과로 next_state, reward, done이라는 세 가지 값을 반환한다.



그림처럼 현재 시간을 t라고 했을 때, 시간 t에 에이전트가 행동을 하면, 보상을 얻고 다음 상태 S_{t+1}로 전이한다. GridWorld 클래스에서는 step() 메서들르 통해 에이전트에게 행동하도록 해 샘플 데이터를 얻는다. 또한 reset() 메서드도 있다. 환경을 초기 상태로 재설정 하는 메서드이다.

5.3.2. 에이전트 클래스 구현

이제 몬테카를로법을 이용하여 정책 평가를 수행하는 에이전트를 구현할 수 있다.

```
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)
```

초기화 메서드인 __inint__()에서 할인율과 행동의 가짓수를 설정한다. 그리고 무작위 행동을 할 확률 분포를 만들어 설정한다. self.memory는 에이전트가 실제로 행동하여 얻은 경험('상태, 행동, 보상')을 담는 역할이다.

get_action() 메서드에서는 state에서 수행할 수 있는 행동을 하나 가져오고 확률 분포에 따라 행동을 한 개씩 샘플링한다. 다음은 RandomAgent 클래스의 나머지 코드 부분이다.

```
class RandomAgent:
    ...
    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]
```

add() 메서드는 실제로 수행한 행동과 보상을 기록해준다. (상태, 행동, 보상)을 튜플 단위로 저장해준다. 여기서 마지막 상태는 저장되지 않는다. 왜냐하면 마지막 상태의 가치 함수는 항상 0이기 때문에 추가하지 않는다.

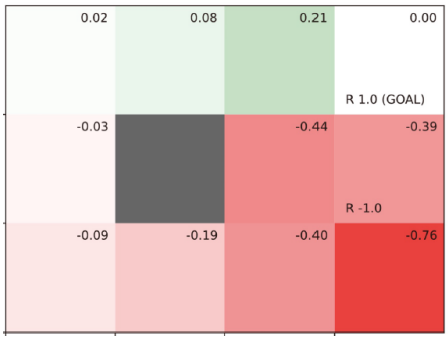
eval() 메서드는 실제로 얻은 self.memory를 역방향으로 따라 가면서 각 상태에서 얻은 수익을 계산한다. 그리고 각 상태에서의 가치 함수를 그때까지 얻은 수익의 평균으로 구한다.

5.3.3. 몬테카를로법 실행

```
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

# 모든 에피소드 종료
```

에피소드를 총 1000번 실행했다. 먼저 에이전트에게 행동하게 하고 그 결과로 얻은 '상태, 행동, 보상'의 샘플 데이터를 기록한다. 목표에 도달하면 그동안 얻은 샘플 데이터를 이용하여 몬테카를로법으로 가치 함수를 갱신한다. 에피소드가 끝나면 다음 에피소드를 시작한다.

다음은 이 코드를 실행하여 얻은 가치 함수를 시각화한 모습이다.



이번에는 무작위 정책의 가치 함수를 평가했다. 에이전트의 시작 위치는 왼쪽 맨 아래의 한 곳으로 고정되어 있지만 무작위 정책이기 때문에 어떠한 위치든 경유할 수 있다. 그래서 모든 위치(상태)에서의 가치 함수를 평가할 수 있었다. (에이전트의 시작 위치가 정해져 있고 정책이 결정적이라면, 에이전트는 정해진 상태만을 경유한다. 그러면 일부 상태에서는 수익 데이터를 수집하지 못할 수 있다.)

5.4. 몬테카를로법으로 정책 제어하기

정책 평가를 수행했으면, 최적 정책을 찾는 '정책 제어'가 필요하다. 핵심은 평가와 개선을 번갈아 반복하는 것이다.

5.4.1. 평가와 개선

최적 정책은 평가와 개선을 번갈아 반복하며 얻는다. 평가 단계에서는 정책을 평가하여 가치 함수를 얻는다. 그리고 개선 단계에서는 가치 함수를 탐욕화하여 정책을 개선한다.

앞에서는 몬테카를로법으로 정책을 평가했다. 예를 들어 π라는 정책이 있다면, 몬테카를로법을 이용해 $V_\pi(s)$를 얻을 수 있었다. 그 다음은 개선 단계이다. 개선 단계에서는 탐욕화를 수행한다.

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

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

개선 단계에서는 가치 함수의 값을 최대로 만드는 행동을 선택한다. Q 함수의 경우 식과 같이 Q 함수가 최대값을 반환하는 행동을 선택한다. 이때 행동이 단 하나로 결정되므로 함수 μ(s)로 나타낼 수 있다.

만약 가치 함수 V를 사용하여 정책을 개선한다면 위의 식을 계산하면 된다. 하지만 일반적인 강화 학습 문제에서는 환경 모들 상태 전이 함수와 보상을 알 수 없다. 따라서, Q 함수를 이용하는 식을 이용해야 한다. 식에서는 단순히 Q(s,a)가 최대가 되는 행동 a를 찾아내기만 하면 되므로 환경 모델이 필요 없다.

Q 함수를 대상으로 개선할 경우 Q 함수를 평가해야 한다. 몬테카를로법의 갱신식에서 V(s)를 Q(s,a)로 전환하면 된다.

[상태 가치 함수 평가]

- 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 함수 평가]

- 일반적인 방식: $Q_n(s,a) = \frac{G^{(1)} + G^{(2)} + \cdots + G^{(n)}}{n}$
- 증분 방식: $Q_n(s,a) = Q_{n-1}(s,a) + \frac{1}{n}\{G^{(n)} - Q_{n-1}(s,a)\}$

이 식에서 알 수 있듯이 상태 가치 함수든, Q 함수든 대상이 바뀌었을 뿐, 몬테카를로법으로 하는 계산 자체는 변하지 않는다.

cf) 이해하기: 왜 식이 다르지 않을까?

(1) 상태 가치 함수: 특정한 상태 s에 있을 때, 거기서부터 얻을 수 있는 기대 보상을 평가하는 함수 = "어떤 상태 s가 얼마나 좋은 상태인가?"를 평가, 모든 행동을 고려한 평균적인 보상을 구하는 방식
(2) 행동 가치 함수: 특정한 상태 s에서 행동 a를 했을 때, 거기서부터 얻을 수 있는 기대 보상을 평가하는 함수

= “어떤 상태에서 특정 행동을 했을 때 얼마나 좋은가?”를 평가

➔ 식이 같은 이유? 어떤 상태 s 또는 상태-행동 쌍(s,a)에 대한 기댓값을 구하려면 여러 번 방문한 결과를 평균을 내야 한다. 즉 단순히 여러 번 샘플링한 결과를 평균 내는 방식이라서 두 식이 동일한 형태가 된다. (두 개 다 과거 경험의 평균을 계산하는 문제이기 때문이다.) 두 개의 차이는 “행동을 고려하느냐”이고, 기대값을 근사하는 과정 자체는 동일하다.

5.4.2. 몬테카를로법으로 정책 제어 구현

몬테카를로법으로 정책을 제어하는 에이전트를 구현할 수 있다.

```
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()
```

앞서 구현한 RandomAgent 클래스와 거의 같다. 이어서 핵심인 정책 제어이다.

```
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)
```

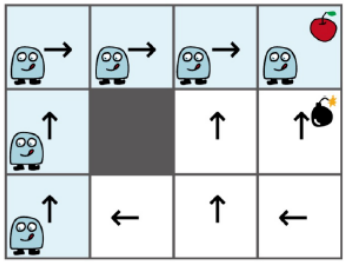
먼저 greedy_probs() 함수를 준비한다. 이 함수는 탐욕 행동을 취하도록 하는 확률 분포를 반환한다. 즉, 매개변수로 받은 state 상태에서 Q 함수의 값이 가장 큰 행동만을 취하게끔 확률 분포를 만들어준다. 예를 들어 주어진 상태에서 0번째 행동의 Q 함수 값이 가장 크다면 {0:1.0, 1: 0.0, 2:0.0, 3:0.0}을 반환한다.

update()는 self.Q를 갱신한다. 증분 방식으로 갱신하고, 갱신이 끝나면 state의 정책을 탐욕화한다.

하지만, 이 코드에서는 다음 두 가지를 개선해야 한다. 1. 완전한 탐욕이 아닌 ε-탐욕 정책으로 변경해야 한다. 2. Q 갱신을 ‘고정값 a 방식’으로 수행해야 한다.

5.4.3. ε-탐욕 정책으로 변경 (첫 번째 개선 사항)

에이전트는 개선 단계에서 정책을 탐욕화한다. 탐욕화의 결과로 해당 상태에서 취할 수 있는 행동이 단 하나로 고정된다. 예를 들어 정책을 탐욕화하여 그림처럼 행동하게 되었다고 가정해보자.



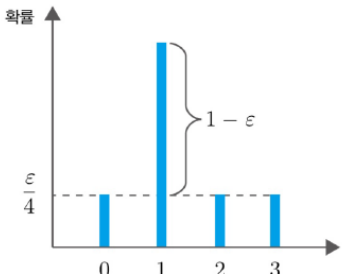
그림과 같이 탐욕 행동만을 수행하면 에이전트의 경로가 한 가지 경로로 고정된다. 그러면 모든 상태와 행동 조합에 대한 수익 샘플 데이터를 수집할 수 없다. 이 문제를 해결하려면 에이전트가 '탐색'도 시도하도록 해야 한다.

cf) 이해하기: ε-탐욕 정책이 필요한 이유
완전한 탐욕 정책을 하면 현재 상태에서 가장 높은 가치를 가진 행동만 수행한다. 그렇다면 새로운 행동을 탐색할 기회가 없다. 우리는 모든 상태-행동 쌍의 Q 값을 정확하게 알고 있는 것이 아니다. 그래서 탐욕적으로 행동하면 초기 선택한 행동만 계속하게 되고, 다른 좋은 행동을 시도하지 않게 된다. 이렇게 되면 지역 최적해에 갇힐 수 있다. 그래서 ε-탐욕 정책을 통해 가끔씩 랜덤한 행동을 시도하면서 더 나은 행동을 찾을 기회를 만들게 된다.
몬테라를로법에서는 완전한 에피소드를 실행한 후, 그 데이터를 바탕으로 가치 함수를 업데이트 한다. 그래서 한 번의 에피소드에서 특정 행동을 전혀 선택하지 않으면 그 행동에 대한 정보가 없어서 학습이 불가능하다. 그래서 ε-탐욕 정책을 통해 모든 행동에 대한 데이터를 수집하고 최적 정책을 선택할 확률을 높일 수 있다.

에이전트에게 탐색을 시키는 대표적인 방법이 ε-탐욕 정책이다. 기본적으로 Q 함수의 값이 가장 큰 행동을 선택하되, 무작위성을 첨가하여 낮은 확률로 아무 행동이나 선택하도록 하는 정책이다. 이렇게 하면 각 상태에서 정해진 행동만 선택되는 문제를 방지할 수 있다. 그러면서도 대다수 경우에 탐욕 행동을 취하기 때문에 최적 정책에 가까운 결과를 얻을 수 있다.

```
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
```

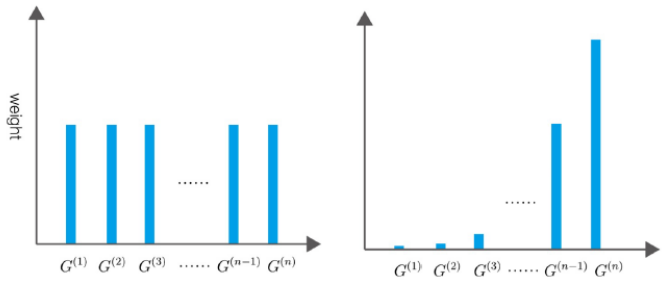
확률 분포를 ε-탐욕 형태로 만들기 위해 우선 모든 행동의 확률을 $\varepsilon/4$로 설정하고, Q 함수의 가장 큰 행동에 따로 1 - ε 확률을 더했다.



5.4.4. 고정값 a 방식으로 수행(두 번째 개선 사항)

```
# 수정 전
# self.Q[key] += (g - self.Q[key]) / self.cnts[state]  # ❷

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

2 부분을 고정값 alpha로 바꿔준다. 수정 전과 후는 다음과 같은 차이가 있다.



수정 전 방식은 모든 샘플 데이터에 균일하게 가중치를 주고 평균을 낸다. (표본 평균) 표본 평균에서는 각 데이터에 대한 가중치가 모두 1/n이다.

반면 고정값 a로 갱신하는 방식은 각 데이터에 대한 가중치가 기하급수적으로 커진다. 이를 지수 이동 평균이라고 한다. 지수 이동 평균은 최신 데이터일수록 가중치를 훨씬 크게 준다.

몬테카를로법을 이용한 정책 제어에는 지수 이동 평균이 적합하다. '수익'이라는 샘플 데이터가 생성되는 확률 분포가 시간에 따라 달라지기 때문이다. 에피소드가 진행될수록 정책이 갱신되기 때문에 수익이 생성되는 확률 분포도 달라진다. 밴디트 문제의 용어를 빌려 '비정상 문제'라고 할 수 있다.

cf) 이해하기: 몬테카를로법에서 지수 이동 평균이 적합한 이유
몬테카를로법에서 정책 제어는 정책을 계속 갱신하면서 최적 정책을 찾아가는 과정이다. 이 과정에서 수익의 확률 분포가 계속 변할 수 있다. 정책이 계속 업데이트 되면 행동도 달라지고 따라서 보상도 달라지기 때문이다. 이 문제를 비정상 문제라고 한다. 즉, 과거의 보상 분포가 현재와 다를 수 있다는 것이다.
모든 보상에 대해 동일한 가중치를 갖게 되면, 환경이 변할 때 빠르게 적응하지 못한다는 단점이 있다. 예전의 수익이 현재의 정책과 관련이 없을 수도 있는데 평균값에 동일하게 포함돼 문제가 된다. 하지만, 지수 이동 평균을 이용하여 최근 데이터에 더 높은 가중치를 주면 빠르게 변화하는 환경에 적응할 수 있다.

5.4.5. 몬테카를로법으로 정책 반복법 구현(개선 버전)

앞의 두 개선사항을 반영한 코드이다.

```
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()
```

```
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  # ❶
        self.pi[state] = greedy_probs(self.Q, state, self.epsilon)  # ❷
```

self.epsilon은 무작위로 행동할 확률이다. 지금처럼 0.1로 설정하면 10%의 확률로 무작위 행동을 선택하고, 90%의 확률로 탐욕 행동을 선택한다. self.alpha는 Q 함수를 갱신할 때 사용하는 고정값이다.

5.5. 오프-정책과 중요도 샘플링

앞에서 몬테카를로법에 ε-탐욕 정책을 결합하여 최적에 가까운 정책을 얻었다. 하지만 그 결과는 완벽한 최적 정책이 아니다. 왜냐하면 탐색을 강제로 포함시키기 때문에 최적 행동만 하는 것이 아니라 일부 확률로 무작위 행동을 수행하기 때문에 무조건 최적 행동만 선택하는 정책이 아니게 된다.

5.5.1. 온-정책과 오프-정책

사람은 다른 사람의 행동을 관찰하여 자신의 능력을 개선할 힌트를 얻기도 한다. 강화학습에서 자신과 다른 환경에서 얻은 경험을 토대로 자신의 정책을 개선한다는 것은 오프-정책이라고 한다. 반면 스스로 쌓은 경험을 토대로 자신의 정책을 개선하는 방식은 온-정책이라고 한다.

역할 측면에서 보면 에이전트의 정책은 두 가지이다. 하나는 정책에 대해 평가한 다음 개선한다. 이러한 정책을 대상 정책이라고 한다. 다른 하나는 에이전트가 실제로 행동을 취할 때 활용하는 정책이다. 이러한 정책을 행동 정책이라고 한다.

지금까지는 '대상 정책'과 '행동 정책'을 구분하지 않았다. 즉 평가와 개선의 대상인 대상 정책과 실제 행동을 선택하는 행동 정책을 동일시 한 것 이다. 이처럼 두 정책을 같이 생각하는 경우가 온-정책, 따로 생각하는 경우르 오프-정책이라고 한다.

오프 정책을 통해 다른 정책(행동 정책)에서 얻은 경험을 토대로 자신의 정책(대상 정책)을 평가하고 개선할 수 있다. 오프-정책이라면 행동 정책에서는 탐색을, 대상 정책에서는 활용만을 할 수 있다. 행동 정책에서 얻은 데이터로부터 대상 정책과 관련된 기댓값을 계산하는 방법에서 등장한 것이 중요도 샘플링 기법이다.

5.5.2. 중요도 샘플링

중요도 샘플링은 어떤 확률 분포의 기댓값을 다른 확률 분포에서 샘플링한 데이터를 사용하여 계산하는 기법이다. $\mathbb{E}_{\pi}[x]$라는 기댓값 계산을 예로 중요도 샘플링을 해볼 수 있다. 여기서 x는 확률 변수이며 x의 확률은 $\pi(x)$로 표기한다. 그래서 확률 분포의 기댓값은 다음과 같다.

$$\mathbb{E}_{\pi}[x] = \sum x\pi(x)$$

이 기댓값을 몬테카를로법으로 근사하려면 x의 확률 분포 π에서 샘플링하여 평균을 내면 된다.

$$\text{샘플링}: x^{(i)} \sim \pi \quad (i = 1, 2, \cdots, n)$$
$$\mathbb{E}_{\pi}[x] \simeq \frac{x^{(1)} + x^{(2)} + \cdots + x^{(n)}}{n}$$

지금 문제어서는 x가 다른 확률 분포에서 샘플링된 경우의 문제를 풀고자 한다. 예를 들어 x가 π가 아닌 b라는 확률 분포에서 샘플링되었다고 가정해보자. 이 경우 기댓값은 다음과 같은 식 변형에 있다.

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

두번째 식에서 $\frac{b(x)}{b(x)}$는 항상 1이므로 등식이 성립한다. 마지막 식으로 형식을 바꾸면 확률 분포 b(x)애서의 기댓값으로 간주할 수 있다.

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

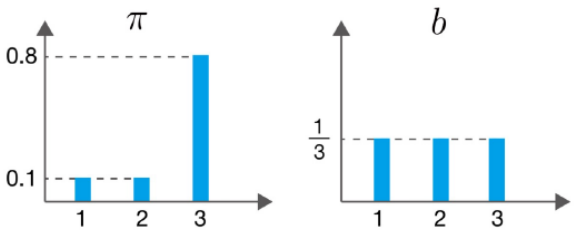
여기서는 확률 분포 π에 대한 기댓값을 확률 분포 b에 대한 기댓값으로 표현한 것이다. 또한 각각의 x에 $\frac{\pi(x)}{b(x)}$가 곱해진다. 여기서 $p(x) = \frac{\pi(x)}{b(x)}$라고 하면, 각각의 x에는 가중치로서 p(x)가 곱해진다고 볼 수 있다.

위의 식을 근거로 몬테카를로법을 수식으로 표현하면 다음과 같다.

$$\text{샘플링: } x^{(i)} \sim b \quad (i = 1, 2, \cdots, n)$$
$$\mathbb{E}_{\pi}[x] \simeq \frac{\rho(x^{(1)})x^{(1)} + \rho(x^{(2)})x^{(2)} + \cdots + \rho(x^{(n)})x^{(n)}}{n}$$

이제 다른 확률 분포 b에서 샘플링한 데이터를 이용하여 $\mathbb{E}_{\pi}[x]$를 계산할 수 있다.

아래 그림의 확률 분포를 대상으로 중요도 샘플링을 수행해보자.



먼저 확률 분포 π의 기댓값을 일반적인 몬테카를로법으로 구할 수 있다.

```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))
```

출력 결과

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

여기서는 확률 분포 pi에서 데이털르 100개만 샘플링하여 평균을 구했다. 그 결과는 2.78로 참값 2.7과 가깝게 나왔다. 분산은 0.27로 나왔다.

분산은 데이터가 얼마나 흩어져 있는가를 나타내고, 기대값과 분산의 관계를 수식으로 표현할 수 있다.

$$\mathrm{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2]$$

이어서 중요도 샘플링을 이용하여 기댓값을 구할 수 있다.

ch05/importance

```
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))
```

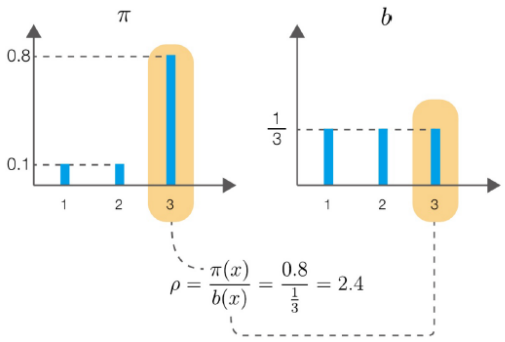
출력 결과

```
중요도 샘플링: 2.95 (분산: 10.63)
```

확률 분포 b를 사용하여 샘플링 했다. 출력 결과는 2.95로 참값인 2.7과는 거리가 있고, 분산은 10.63으로 몬테카를로법 때의 0.27보다 훨씬 큰 것을 알 수 있다.

5.5.3. 분산을 작게 하기

분산이 작을수록 적은 샘플로도 정확하게 근사할 수 있다. 반대로 분산이 크면 샘플을 더 많이 사용해야 한다. 중요도 샘플링을 쓰면 분산이 커지는 이유가 있다.



그림은 샘플 데이터로 3을 선택한 예이다. 이때 가중치 p는 2.4이다. 즉 3을 뽑았음에도 3 * 2.4 = 7.2를 얻는다는 뜻이다. 이유는 다음과 같다.

- 확률 분포 π를 기준으로 했을 때 3은 대표적인 값이기 때문에 3이 많이 샘플링 돼야 한다.
- 하지만 확률 분포 b에서는 3이 특별히 많이 선택되지 않는다.
- 이 간극을 메우기 위해 3이 샘플링 된 경우, 그 값이 커지도록 '가중치'를 곱하여 조정한다.

이처럼 확률 분포 π와 b의 차이를 고려하여 샘플링된 값에 가중치를 곱하여 조정하는 일은 의미가 있다. 하지만 3의 추정값이 7.2로 추정된다는 문제가 생긴다. 이처럼 실제 얻은 값에 부여하는 가중치 p의 보정 효과가 클수록 분산이 커진다.

분산을 줄이기 위해서 두 확률 분포를 가깝게 만들면 된다. 이렇게 하면 가중치 p의 값을 1에 가깝게 만들 수 있다. 직전 코드에서 확률 분포 b의 값만을 바꿔볼 수 있다.

```python
b = np.array([0.2, 0.2, 0.6])  # 확률 분포 변경 (앞에서는 [1/3, 1/3, 1/3]로 설정)
n = 100
samples = []

for _ in range(n):
    idx = np.arange(len(b))  # [0, 1, 2]
    i = np.random.choice(idx, p=b)
    s = x[i]
    rho = pi[i] / b[i]
    samples.append(rho * s)

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

출력 결과

```
중요도 샘플링: 2.72 (분산: 2.48)
```

이와 같이 확률 분포 b를 [0.2, 0.2, 0.6]으로 변경하여 확률 분포 pi에 가깝에 만들 수 있다. 결과는 평균이 2.72로 참값에 가까워졌고, 분산은 2.48로 더 작아졌다.

이처럼 중요도 샘플링 시 두 확률 분포를 비슷하게 설정하면 분산을 줄일 수 있다. 단, 강화 학습에서 핵심은 한쪽 정책은 탐색에 다른 쪽 정책은 활용에 이용하는 것이다. 이 조건을 염두에 둔 확률 분포를 최대한 가깝게 조정하면 분산을 줄일 수 있다.

이것이 바로 중요도 샘플링이다. 중요도 샘플링을 이용하면 오프-정책을 구현할 수 있다. 즉, 행동 정책이라는 확률 분포에서 샘플링된 데이터를 토대로 대상 정채겡 대한 기댓값을 계산할 수 있다.

cf) 이해하기: 중요도 샘플링
강화학습에서는 최적 정책을 찾고 싶어한다. 그래서 탐색을 하면서 최적 정책을 찾는다. 하지만 나중에는 탐색 없이 최적 행동만 선택하는 정책을 학습해야 한다. 문제는, 탐색 정책에서 모은 데이터를 최적 정책 학습에 그대로 쓰면 안된다는 것이다. 탐색 정책은 랜덤 행동을 포함하기 때문에 실제로 최적의 행동이 아닐 수 있다. 그래서 이 데이터를 그냥 쓰는 것이 아니라 탐색 정책에서 얻은 데이터가 최적 정책에서 얼마나 중요한지 보정을 해야 한다. 이를 해결하는 방법으로 중요도 샘플링을 사용한다.

중요도 샘플링은 탐색 정책에서 수집한 데이터를 최적 정책에서 얻은 것처럼 보정하는 방법이다. 탐색 정채과 최적 정책이 다르기 때문에, 데이터를 보정하려면 가중치를 곱해줘야 한다.

$$w = \frac{\pi(a|s)}{b(a|s)}$$

- w = 중요도 샘플링 비율 (데이터를 얼마나 보정할지 결정하는 값)
- $\pi(a|s)$ = 최적 정책(목표 정책)에서 행동 a를 선택할 확률
- $b(a|s)$ = 탐색 정책(탐색을 포함한 행동 정책)에서 행동 a를 선택할 확률

탐색 정책이 랜덤하게 행동했을 경우 b(a|s)의 값이 커져 보정을 작게 한다. 반대로 최적 정책에서도 이 행동을 많이 했을 경우 π(a|s)의 값이 커져 보정을 크게 한다. 즉, 탐색 정책에서 선택한 행동이 최적 정책에서도 자주 나오는 행동이라면 더 중요하게 반영하고, 그렇지 않다면 덜 반영하는 것이다.