Chapter 1 밴디트 문제

1.1 머신러닝 분류와 강화 학습

머신러닝(Machine Learning): 기계, 즉 컴퓨터에 데이터를 주고 컴퓨터가 스스로 어떤 규칙이나 패턴을 찾아내도록 하는 것. (규칙을 사람이 알려주는 것이 아니라 컴퓨터가 스스로 찾아 학습)

대표적으로는 지도학습, 비지도학습, 강화 학습이 있다.

1.1.1. 지도 학습

지도 학습은 입력(문제)과 출력(정답)을 쌍으로 묶은 데이터가 주어진다. '정답 레이블'이 존재한다는 점이 지도 학습의 가장 큰 특징.
정답 레이블을 준비해야 하기 때문에 시간이 많이 소요될 수 있다. ('레이블링', '애너테이션'이라고 말함)

1.1.2. 비지도 학습

비지도 학습은 '정답 레이블' 없이 오직 데이터만 존재한다. 데이터에 숨어 있는 구조나 패턴을 찾는 용도로 주로 쓰이며 군집화(클러스터링), 특성 추출, 차원 축소 등이 있다.
정답 레이블이 필요하지 않아 준비하기 비교적 쉽다.

1.1.3. 강화 학습

강화 학습은 에이전트와 환경이 서로 상호작용한다. 에이전트는 어떤 환경에 놓여져, 환경의 상태를 관찰하고, 상태에 적합한 행동을 취한다. 행동을 취한 결과로 환경의 상태가 변화하고, 환경으로부터 보상을 받음과 동시에 변화된 '새로운 상태'를 관찰한다.

강화 학습의 목표는 에이전트가 얻는 보상의 총합을 극대화하는 행동 패턴을 익히는 것. 이 과정에서 스스로 시행착오를 겪으면서 데이터를 수집하고 수집된 데이터를 바탕으로 더 나은 행동의 패턴을 익혀 나간다.

cf) 강화 학습에서 보상은 지금까지 취한 행동의 최적인지 여부를 판단할 수 없다.

강화 학습에서의 보상은 지도 학습에서의 레이블과 다르다. 보상은 행동에 대한 피드백일 뿐, 그 보상으로부터 지금까지 취한 행동이 최적인지 여부를 판단할 수 없다. 반면, 지도 학습에서는 '정답'이 존재해, 어떤 행동을 취했을 때 최적의 행동이 무엇인지 알려줄 수 있다.

강화학습에서의 보상은 특정 상태에서 취한 행동에 대한 피드백으로, 그 행동이 좋은 결과를 낳았다는 신호일 뿐이다. 최적의 행동을 찾기 위해서는 여러 번의 시도와 학습을 통해 다양한 상황에서 행동을 평가하고 비교해야 한다. 보상을 받았다고 해도, 그것이 좋은 행동일 수 있지만 다른 상황에서는 그 행동이 적합하지 않을 수 있기 때문이다. (미로의 출구를 찾는 상황에서, 미로를 탈출만 하면 보상을 얻어 무조건 좋은 것 만이 아니라 최적의 탈출 경로를 찾는 것이 강화 학습의 목표)

1.2. 밴디트 문제

1.2.1. 밴디트 문제란?

밴디트(bandit)는 '슬롯머신'의 또 다른 이름이다. 슬롯머신에는 손잡이가 있고, 손잡이를 당기면 그림들이 한번에 바뀐다. 그리고 바뀐 그림들의 조합이 무엇이냐에 따라 얻는 코인의 액수가 정해진다.

밴디트 문제를 정확하게 멀티-암드 밴디트 문제(multi-armed bandit problem)이라고 한다.
슬롯 머신이 여러 대인 상황을 상상하며 문제를 다뤄 보자.

밴디트 문제에서는 각각의 슬롯머신에서 좋은 그림이 나올 확률이 제 각각이다. 이런 상황에서 플레이어는 정해진 횟수(1000회)를 플레이한다. 처음에는 어떤 슬롯 머신이 좋은 지 등의 정보를 알지 못하고, 실제로 플레이하면서 좋은 머신을 찾아내야 한다. 이 문제에서 목표는 정해진 횟수 안에 코인을 최대한 많이 얻는 것이다.

이 상황에서 슬롯머신은 '환경', 플레이어는 '에이전트'에 해당한다. 이렇게 환경과 에이전트가 주어진 사이에 플레이어가 여러 슬롯머신 중 한대를 선택해 플레이 하는 것을 '행동'이라고 한다. 그 행동의 결과로 플레이어는 슬롯에서 코인을 얻는다. 이 코인이 '보상'이다.

cf) 일반적인 강화 학습 문제에서는 환경에 상태 정보가 있다. 에이전트가 어떤 행동을 하면 환경의 상태가 바뀌고, 에이전트는 새로운 상태를 관찰하여 적절한 행동을 취하게 된다. 밴디트 문제에서는 플레이어가 이용하는 슬롯머신들의 확률 설정에 변화가 없으므로, 환경의 상태가 변하지 않는다.

이 문제에서 목표는 코인을 최대한 많이 얻는 것이다. 이를 위해서는 '좋은 슬롯머신'을 골라야 한다.

1.2.2. 좋은 슬롯머신이란?

슬롯머신의 가장 중요한 특성은 무작위성이다. 얻을 수 있는 코인 개수(보상)가 플레이할 때마다 달라진다. 이 무작위성을 파악하기 위해 '확률'을 활용한다.

예를 들어 슬롯머신이 두 대가 잇고, 보상과 확률이 다음과 같이 설정되었다고 가정한다.

슬롯머신 a

얻을 수 있는 동전 개수	0	1	5	10
확률	0.70	0.15	0.12	0.03

슬롯머신 b

얻을 수 있는 동전 개수	0	1	5	10
확률	0.50	0.40	0.09	0.01

좋은 슬롯머신을 판단하기 위해서는 평균적으로 얻게 되는 코인의 개수, 즉 기댓값을 기준으로 삼아야 한다.

기댓값이 더 큰 쪽이 더 좋은 슬롯머신이라고 할 수 있다.

각 머신들의 기댓값은
- 슬롯머신 a: (0*0.70) + (1*0.15) + (5*0.12) + (10*0.03) = 1.05
- 슬롯머신 b: (0*0.50) + (1*0.40) + (5*0.09) + (10*0.01) = 0.95

이 기댓값은 두 슬롯머신을 플레이 했을 때 얻을 수 있는 코인의 평균이고, 총 1000번의 플레이 기회가 주어진다면, 매번 a를 선택하는 것이 좋은 전략일 것이다.

여기서 중요한 것은, 슬롯머신 플레이 같은 확률적 사건은 '기댓값으로 평가할 수 있어, 무작위성에 현혹되지 않기 위해 '기댓값'을 기준으로 삼아야 한다.

1.2.3. 수식으로 표현하기

보상의 'Reward'의 머리글자를 따 R로 표기한다. 밴디트 문제에서는 슬롯머신이 돌려주는 코인의 개수가 해당된다. R은 {0,1,5,10} 중 하나가 되며 각 값을 '얻을 가능성'이 확률로 정해져 있다.

에이전트가 수행하는 행동은 'Action'의 머리글자를 따 A로 표기한다. 슬롯머신 a와 b를 선택하는 행동을 각각 a와 b라고 한다면, 변수 A는 {a,b} 중 하나의 값을 취하게 된다.

다음은 확률 변수로 정의되는 '기댓값'이다. 기댓값은 'Expectation'의 머리글자르 따 $\mathbb{E}$라고 쓴다. 예를 들어 보상 R의 기댓값은 $\mathbb{E}[R]$로, 행동 A를 선택했을 때 보상 기댓값은 $\mathbb{E}[R|A]$로 표기한다. 예를 들어 a 행동을 선택했을 때 보상 기댓값은 $\mathbb{E}[R|a]$라고 적는다.

보상에 대한 기댓값을 행동 가치라고 한다. 'Qyality'의 머리글자를 따 Q(q)라고 표기하며, 행동 A의 행동가치는 q(A) = $\mathbb{E}[R|A]$처럼 표기할 수 있다.

1.3. 밴디트 알고리즘

밴디트 문제에서 플레이어는 슬롯머신의 '가치(보상 기댓값)'을 알 수 없다. 따라서 플레이어는 실제로 슬롯머신을 플레이하고 그 결과를 토대로 각 슬롯머신의 가치를 추정해야 한다.

1.3.1. 가치 추정 방법

슬롯머신 a와 b를 각각 세 번씩 플레이해서 다음과 같은 결과가 나왔다.

슬롯머신	결과		
	첫 번째	두 번째	세 번째
a	0	1	5
b	1	0	0

각 슬롯머신의 가치 추정치는 회당 코인을 평균내서 구할 수 있다.

$$Q(a) = \frac{0+1+5}{3} = 2 \qquad Q(b) = \frac{1+0+0}{3} = 0.33\cdots$$

위 결과를 통해 슬롯머신 a가 한 회당 2개의 코인을 얻을 수 있어 더 좋을 것이라고 추정할 수 있다.

1.3.2. 평균을 구하는 코드

슬롯머신 한 대에만 집중하여 총 n번 플레이하는 경우, 실제로 얻은 보상은 $R_1, R_2, \cdots, R_n$이라 하면 행동 가치 추정치는 다음과 같이 수식으로 표현할 수 있다.

$$Q_n = \frac{R_1 + R_2 + \cdots + R_n}{n}$$

n번째 행동 가치 추정치는 n개의 보상에 대한 표본 평균으로 구할 수 있다. n개가 늘어날수록 매번 표본평균을 구하는 것은, 시간과 메모리를 많이 사용해 비효율적이므로 효율적으로 구현할 방법이 필요하다.

n-1번째 시점의 행동 가치 추정치인 Q_{n-1}의 수식을 표현할 수 있다.

$$Q_{n-1} = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}$$

이 식의 양번에 n-1를 곱하고 좌우변을 바꾸면

$$R_1 + R_2 + \cdots + R_{n-1} = (n-1)Q_{n-1}$$

이를 활용해 Q_n을 표현하면

$$\begin{aligned} Q_n &= \frac{R_1 + R_2 + \cdots + R_n}{n} \\ &= \frac{1}{n}(\underbrace{R_1 + \cdots + R_{n-1}}_{(n-1)Q_{n-1}} + R_n) \\ &= \frac{1}{n}\{(n-1)Q_{n-1} + R_n\} \\ &= \left(1 - \frac{1}{n}\right)Q_{n-1} + \frac{1}{n}R_n \end{aligned}$$

이를 활용해서, 매번 각 시점의 보상을 구하여 계산하지 않아도, Q_{n-1}, R_n, n의 값을 안다면 Q_n을 구할 수 있다. 이 식을 좀 더 변형시킬 수 있다.

$$\begin{aligned} Q_n &= \left(1 - \frac{1}{n}\right)Q_{n-1} + \frac{1}{n}R_n \\ &= Q_{n-1} + \frac{1}{n}(R_n - Q_{n-1}) \end{aligned}$$

이때, 식을 보면 Q_{n-1}에서 Q_n으로 갱신될 때, R_n 방향으로 얼마나 진행되느냐는 $\frac{1}{n}$ 이 결정한다. $\frac{1}{n}$은 갱신되는 양을 조정하기 때문에 학습률의 역할을 한다.

cf)이해하기

식에서 ($R_n - Q_{n-1}$) 현재 보상과 이전 Q값의 차이를 의미한다. 이 차이가 얼마나 큰지가 중요하고, 이 차이를 얼마나 반영할지는 학습률이 결정하게 된다. 이때 $\frac{1}{n}$은 처음에는 시도가 적기 때문에 크고, 나중에는 시도가 많아져 작아진다. 초기에는 학습률이 커 Q값은 크게 변하게 된다. 하지만 시도가 반복될수록 학습률이 작아져 갱신되는 양이 적어진다. 따라서 Q값의 변화폭도 줄어들고, 학습이 안정화된다는 것을 의미한다. 즉, 시간이 지날수록 이전에 얻은 경험을 더 많이 반영

하고, 새로운 정보에 대한 영향을 적게 주게 된다.

이 방식이 의미있는 이유는 강화 학습에서 안정적으로 학습하기 위해 필요하기 때문이다. 단순히 보상을 받았다고 해서 그 순간의 경험만으로 결정을 내리는 것이 아니라, 여러 번의 경험을 종합하여 최적의 행동을 찾도록 유도하는 과정이다. 학습 초반에는 경험이 부족하기 때문에 새로운 정보를 적극적으로 반영하고, 나중에는 경험이 쌓이기 때문에 안정적인 학습을 위해 조금만 반영해야 한다. 또한, 학습률이 계속 크다면 한 두번의 보상에 의해 Q값이 크게 흔들릴 수 있어, 노이즈(불확실한 정보)를 제거하는 효과도 볼 수 있다.
-〉 탐색 초반에는 빠르게 배우고, 학습 후반에는 점진적으로 조정하는 기법

1.3.3. 플레이어의 정책

이제 플레이어가 가치가 가장 큰 슬롯머신을 찾는 방법을 고민해봐야 한다. 가장 직관적인 정책은 실제로 플레이어가 각각의 슬롯머신을 플레이해보고 가치 추정치가 가장 큰 슬롯머신을 선택하는 정책이다. 이러한 정책을 탐욕정책이라 한다. 지금까지 플레이한 경험(가치 추정치)만으로 최선의 머신을 선택하는 것이다.

하지만, 슬롯머신의 가치 추정치에는 '불확실성'이 스며 있기 때문에 잘못해서 확실하지 않은 추정치를 전적으로 신뢰하면 최선의 행동을 놓칠 수 있다는 문제가 발생한다. 그래서 플레이어는 탐색을 통해 불확실성을 줄여 추정치의 신뢰도를 높여야 한다.

- 활용: 지금까지 실제로 플레이한 결과를 바탕으로 가장 좋다고 생각되는 슬롯머신을 플레이 (탐욕정책)
- 탐색: 슬롯머신의 가치를 정확하게 추정하기 위해 다양한 슬롯머신을 시도

강화 학습 알고리즘은 '활용과 탐색의 균형'이 중요하다. 이 균형을 맞추는 방법 중 가장 기본적인 알고리즘은 ε-탐욕 정책이다.

ε-탐욕 정책은 ε의 확률로 '탐색'을 하고, 나머지는 '활용'을 하는 방식이다. 탐색할 차례에서는 다음 행동을 무작위로 선택하여 다양한 경험을 쌓는다. 이렇게 쌓여진 경험을 바탕으로 신뢰도를 높이고, 나머지 1 - ε의 확률로 활용을 수행한다.

1.4. 밴디트 알고리즘 구현

밴디트 알고리즘을 코드로 구현해보자. 여기서는 조건을 단순화하여 슬롯 머신이 반환하는 코인을 최대 1개로 제한한다. 즉, 슬롯머신을 플레이하면 승리(1)와 패배(0) 중 하나를 보상으로 얻는다. 그리고 슬롯머신에는 승리 확률이 무작위로 설정되 있구, 총 10개의 슬롯머신이 있다.

1.4.1. 슬롯머신 구현

[1]:
```
import numpy as np
```

[13]:
```
# np.random.rand()는 0.0 이상 1.0 미만의 무작위 수 생성, 그 안에 인자를 넣으면 해당 숫자만큼의 무작위 수 생성

class Bandit:
    def __init__(self, arms = 10): # arms = 슬롯머신 대수
        self.rates = np.random.rand(arms) # 슬롯머신 각각의 승률 설정(무작위)
    def play(self, arm): # arm 은 몇 번째 슬롯머신을 플레이 할지 지정
        rate = self.rates[arm]
        #arm번째 머신의 승률을 가져옴
        if rate > np.random.rand(): # 승률과 무작위 수 하나를 비교하여 승률이 더 크면 승리
            return 1
        else: # 승률이 무작위 수보다 작으면 패배
            return 0
```

[15]:
```
bandit = Bandit()

for i in range(3):
    print(bandit.play(0)) #0번째 슬롯머신을 3회 연속으로 플레이
```

```
1
1
1
```

1.4.2. 에이전트 구현

```
# 0번째 슬롯머신에만 집중하여 해당 슬롯머신의 가치 추정치 구하기
bandit = Bandit()
Q = 0

for n in range(1,11): # 10번 반복
    reward = bandit.play(0) # 0번째 슬롯머신 플레이
    Q += (reward - Q) / n # 가치 추정치 갱신
    print(Q)
```

```
0.0
0.5
0.6666666666666666
0.5
0.6
0.5
0.5714285714285714
0.5
0.4444444444444444
0.5
```

```
# 10대의 슬롯머신 각각의 가치 추정치 구하기
bandit = Bandit()
Qs = np.zeros(10) # 각 슬롯머신의 가치 추정치
ns = np.zeros(10) # 각 슬롯머신의 플레이 횟수

for n in range(10):
    action = np.random.randint(0,10) # 무작위 행동 (임의의 슬롯머신 선택)
    reward = bandit.play(action)

    ns[action] += 1 # action번째 슬롯머신을 플레이한 횟수 추가
    Qs[action] += (reward - Qs[action]) / ns[action]
    print(Qs)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0.  0.  0.  0.  0.5 0.  0.  0.  0.  0. ]
[0.  0.  0.  0.  0.5 0.  0.  0.  0.  0. ]
[0.  0.  0.  0.  0.5 0.  0.  0.  0.  0. ]
[0.  0.  0.  0.  0.5 0.  0.  0.  0.  0. ]
[0.         0.         0.         0.         0.66666667 0.
 0.         0.         0.         0.        ]
[0.         0.         0.         0.         0.66666667 0.
 0.         0.         0.         0.        ]
[0.  0.  0.  0.  0.5 0.  0.  0.  0.  0. ]
[0.  0.  0.  0.  0.5 0.  0.5 0.  0.  0. ]
[0.  0.  0.  0.  0.4 0.  0.5 0.  0.  0. ]
```

이렇게 해서 각 슬롯머신의 가치를 추정할 수 있다.

이렇게 익힌 가치 추정치를 바탕으로 Agent 클래스를 구현할 수 있다. Agent 클래스는 ε-탐욕정책을 따라 행동을 선택하도록 한다.

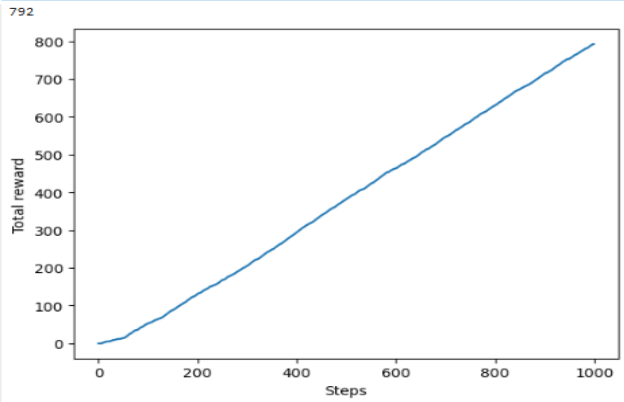
```
# 각 슬롯머신의 가치 추정치를 활용하여 ε-탐욕정책을 따라 행동하는 Agent 생성
class Agent:
    def __init__(self, epsilon, action_size = 10): # action_size는 에이전트가 행동할 수 있는 선택의 가지 수 (슬롯머신 개수)
        self.epsilon = epsilon # 무작위로 행동할 확률 (탐색 확률)
        self.Qs = np.zeros(action_size)
        self.ns = np.zeros(action_size)

    def update(self, action, reward): # 슬롯머신의 가치 추정
        self.ns[action] += 1
        self.Qs[action] += (reward - self.Qs[action]) / self.ns[action]

    def get_action(self): # 행동 선택 (ε-탐욕정책)
        if np.random.rand() < self.epsilon:
            return np.random.randint(0, len(self.Qs)) # self.epsioln의 확률로 무작위 행동 선택
        return np.argmax(seld.Qs) # 그 외는 가치 추정치가 가장 큰 탐욕 행동 선택
```

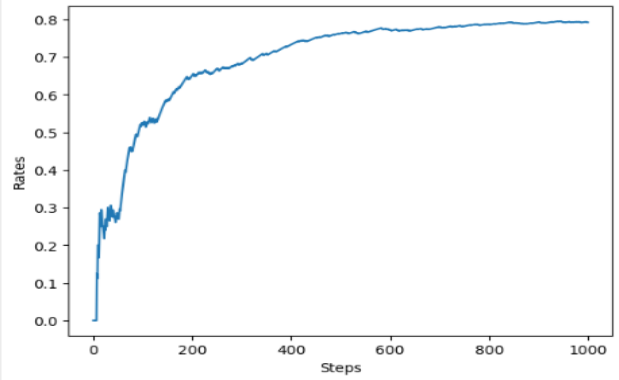
1.4.3. 실행해보기

이제 Bandit 클래스와 Agent 클래스를 활용하여 행동을 1000번 수행하여 보상을 얼마나 받는지 보자.

```python
import matplotlib.pyplot as plt

steps = 1000
epsilon = 0.1

bandit = Bandit()
agent = Agent(epsilon)
total_reward = 0
total_rewards = [] # 보상 합
rates = [] # 승률

for step in range(steps):
    action = agent.get_action() # 행동 선택
    reward = bandit.play(action) # 실제로 플레이하고 보상을 받음
    agent.update(action, reward) # 행동과 보상을 통해 학습
    total_reward += reward

    total_rewards.append(total_reward) # 현재까지의 보상 합 저장
    rates.append(total_reward / (step +1)) # 현재까지의 승률 저장
print(total_reward)

# 그래프 그리기: 단계별 보상 총합
plt.ylabel('Total reward')
plt.xlabel('Steps')
plt.plot(total_rewards)
plt.show()

# 그래프 그리기: 단계별 승률
plt.ylabel('Rates')
plt.xlabel('Steps')
plt.plot(rates)
plt.show()
```

```
792
```





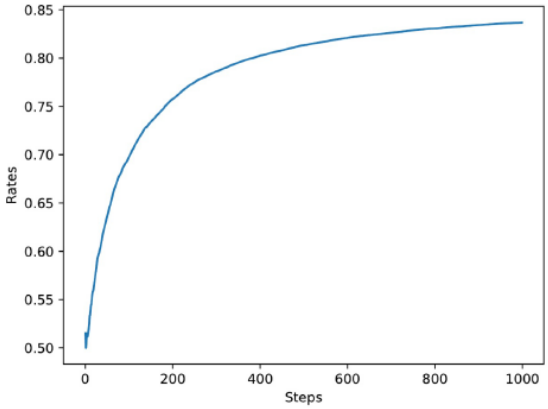
1.4.4. 알고리즘의 평균적인 특성

위의 코드는 실행할 때마다 결과가 달라진다. 각 슬롯머신 10대의 승률을 무작위로 설정했고, 에이전트가 ε-탐욕정책에서도 행동을 무작위로 선택했기 때문이다.

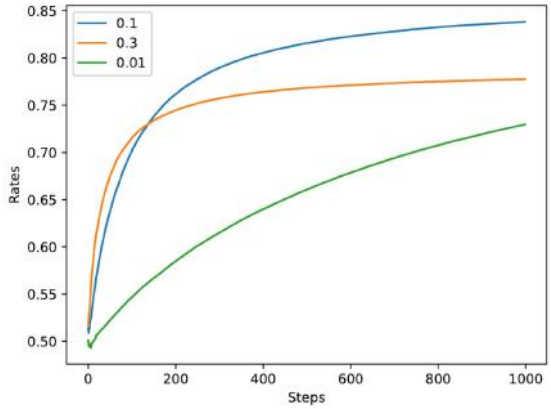
강화 학습 알고리즘을 비교할 때 무작위성으로 인해 한 번의 실험만으로 판단하기 보다 알고리즘의 '평균적 우수성'을 평가해야 한다.

그래서 슬롯머신을 1000번 플레이하는 실험을 총 200번 반복하여 평균을 내보겠다.

```
# 무작위성 때문에 한 번의 실험만으로 평가하긴 어려움 (매 시험을 반복할때마다 결과값이 달라짐) -> 1000번을 플레이하는 실험을 200번 반복하여 평균 내기
runs = 200
steps = 1000
epsilon = 0.1
all_rates = np.zeros((runs, steps)) # (200, 1000) 형상 배열

for run in range(runs): # 200번 실험
    bandit = Bandit()
    agent = Agent(epsilon)
    total_reward = 0
    rates = []

    for step in range(steps):
        anction = agent.get_action()
        reward = bandit.play(action)
        agent.update(action, reward)
        total_reward += reward
        rates.append(total_reward / (step + 1))

    all_rates[run] = rates # 보상 결과 기록

avg_rates = np.average(all_rates, axis = 0) # 각 단계의 평균 저장

# 그래프 그리기: 단계별 승률(200번 실험 후 평균)
plt.ylabel('Rates')
plt.xlabel('Steps')
plt.plot(avg_rates)
plt.show()
```



그래프를 보면 처음 승률은 0.5정도 시작하지만 단계를 거듭할수록 빠르게 높아지고 600단계에서 거의 최대를 찍는다. ε을 다르게 설정하면 이 값도 바뀌게 된다.



ε=0.3이면 승률은 빠르게 상승하지만 400단계를 넘어가면서 상승세가 급격히 꺾인다. 30%라는 높은 확률로 탐색을 시도해 최적의 머신을 선택하는 '활용의 비율이 너무 낮기 때문'이라고 짐작된다. 즉 탐색을 너무 많이 한 것이다.

ε=0.01인 경우, 보상은 꾸준히 상승하지만 상승 속도는 가장 느리다. 탐색 비율이 너무 작아서 최적의 머신을 찾을 확률이 낮아기 때문이다.

이처럼 ε의 값으로 '활용과 탐색의 균형'을 조절할 수 있다. 최적의 ε값은 문제에 따라 달라진다.

1.5. 비정상 문제

지금까지 문제들은 분류상 정상 문제이다. 정상 문제란 보상의 확률 분포가 변하지 않는 문제이다. 슬롯머신에서 설정된 승률은 계속 고정되었기 때문이다. 즉, 슬롯머신의 속성은 한 번 설정되면 에이전트가 플레이하는 동안 절대 변하지 않는다. (구현 코드에서 self.rates는 초기화 후 변경되지 않는다.)

self.rates를 플레이할 때마다 달라지게 설정할 수 있다.

```
# 플레이할 때마다 self.rates에 작은 노이즈를 추가
# np.random.rand()는 평균0, 표준편차 1의 정규분포에서 무작위 수 생성
class NonStatBandit:
    def __init__(self, arms = 10):
        self.arms = arms
        self.rates = np.random.rand(arms)

    def play(self, arm):
        rate = self.rates[arm]
        self.rates += 0.1 * np.random.rand(self.arms) # 노이즈 추가
        if rate > np.random.rand():
            return 1
        else:
            return 0
```

이 효과로 플레이할 때마다 슬롯머신의 가치(승률)가 달라진다. 이처럼 보상의 확률 분포가 변하도록 설정된 문제를 비정상 문제라고 한다.

1.5.1. 비정상 문제를 풀기 위해서

앞서 살펴본 표본 평균 식을 보면 다음 식으로 표현이 된다.

$$
\begin{aligned}
Q_n &= \frac{R_1 + R_2 + \cdots + R_n}{n} \\
&= \frac{1}{n}R_1 + \frac{1}{n}R_2 + \cdots + \frac{1}{n}R_n
\end{aligned}
$$

모든 보상 앞에 $\frac{1}{n}$이 붙어 있어, 각 보상에 대해 똑 같은 가중치를 보유하게 된다. 즉, 새로 얻은 보상이든 오래전에 얻은 보상이든 모두 동등하게 취급한다는 것이다.

비정상 문제에서는 시간이 흐르면 환경(슬롯머신)이 변하기 때문에 과거 데이터(보상)의 중요도는 점점 낮아져야 한다. 반대로 새로 얻은 보상의 가중치는 점점 커져야 한다.

이를 위해 $\frac{1}{n}$의 값을 α의 고정값으로 바꾼다. ($0\langle\alpha\langle 1$)

$$
Q_n = Q_{n-1} + \alpha(R_n - Q_{n-1})
$$

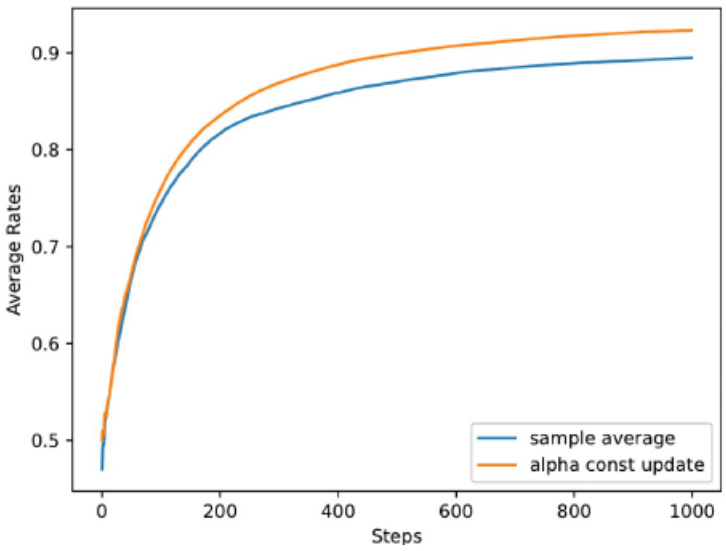
고정값 α로 갱신하면 오래전에 받은 보상일수록 가중치가 작아지게 된다. 이러한 가중치는 기하급수적으로

감소하기 때문에 지수 이동 평균, 또는 지수 가중 이동 평균이라고 한다.

1.5.2. 비정상 문제 풀기

```python
# alpha의 고정값으로 갱신
class AlphaAgent:
    def __init__(self, epsilon, alpha, actions = 10):
        self.epsilon = epsilon
        self.Qs = np.zeros(action)
        self.alpha = alpha # 고정값 a

    def update(self, action, reward):
        # a로 갱신
        self.Qs[action] += (reward - self.Qs[action]) * self.alpha

    def get_action(self):
        if np.random.rand() < self.epsilon:
            return np.random.randint(0, len(self.Qs))
        return np.argmax(self.Qs)
```

그림 1-20 표본 평균과 고정값 α에 의한 갱신 비교



Alpha = 0.8로 설정하여 진행했을 때, 시간이 지날수록 고정값 α로 갱신할 때의 결과가 더 좋아진다. 한편, 표본 평균 방식은 처음에는 잘 작동하지만 시간이 지날수록 격차가 벌어지게 된다. 시간의 변화에 잘 대응하지 못하는 모습이다.