The Multi-Armed Bandit Problem

Multi-Armed Bandit (MAB)

- A classic problem in reinforcement learning
- MAB is a slot machine where we pull the arm (lever) and get a payout (reward) based on some probability distribution.
- If we have k slot machines, it is called a k-armed bandit.

- Example: a 3-armed bandit

arm 1 arm 2 arm 3

Multi-Armed Bandit (MAB)

- When the agent selects an "arm" (a slot machine) and pulls its lever, a reward is given based on its probability.
- If the user wins, the reward is +1. If the user loses, the reward is 0.
- The probability of winning is different for each arm.
  - For example, the probability of winning is 70%, 50%, 30% for arm 1, arm 2, and arm 3, respectively.
- The probability of winning for each arm is not known to the agent.

arm 1 arm 2 arm 3

Multi-Armed Bandit (MAB)

- The goal of the agent is to find the best arm that gives us the maximum average reward.
- Average reward of arm a is denoted as $Q(a)$

$$Q(a) = \frac{\text{Sum of rewards obtained from the arm}}{\text{Number of times the arm was pulled}}$$

- The best (optimal) arm a^* is:

$$a^* = \arg\max_a Q(a)$$

- We play the game for multiple rounds.
- In each round, we can pull only one arm.

- How can we find the best arm?

MAB: A naive approach

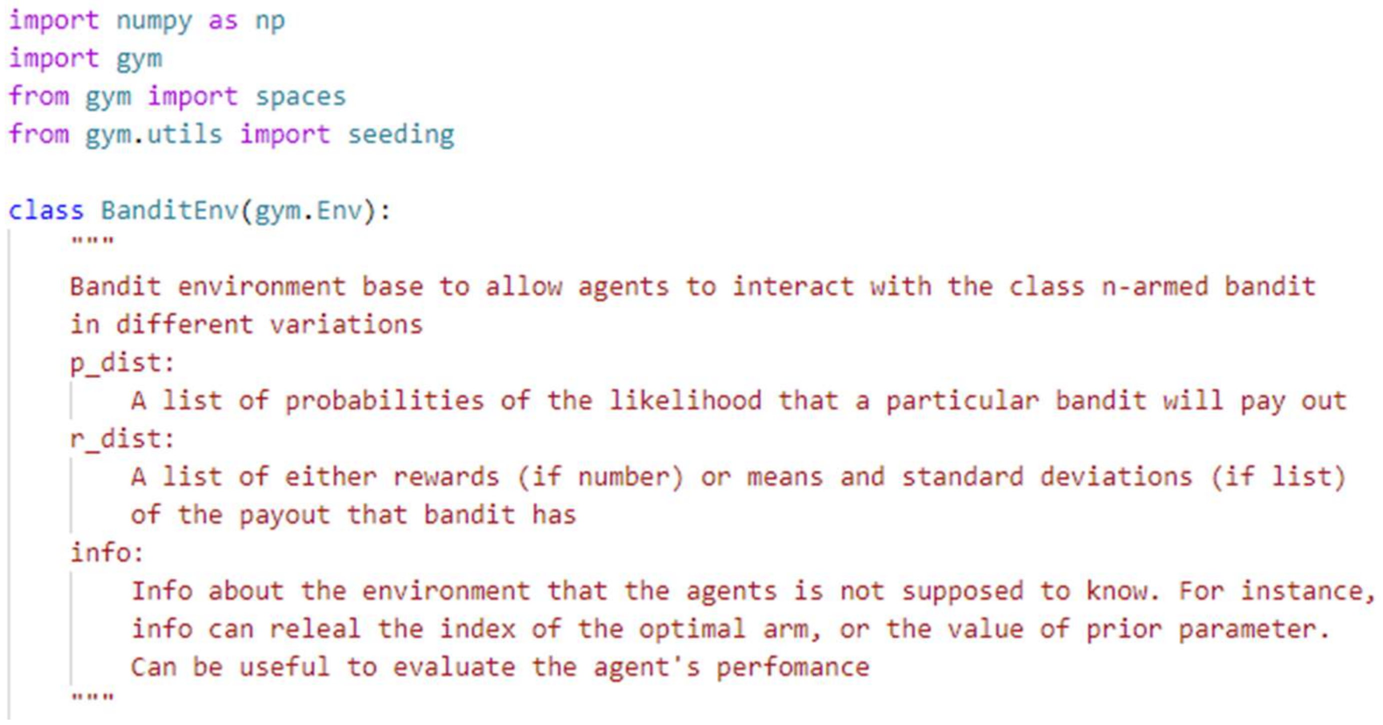
- Approach
  - Suppose we have two arms.
  - In round 1, we choose arm 1. In round 2, we choose arm 2.
  - In odd rounds, we choose arm 1. In even rounds, we choose arm 2.
  - After N rounds, we see which arm gave higher average reward.

- This approach can eventually let us find the best arm.
- However, the cost of identifying the best arm is high.
- The cost of identifying the best arm is called regret.

- We want to find the best arm while minimizing regret.

Defining MAB environment in gym [ex010]

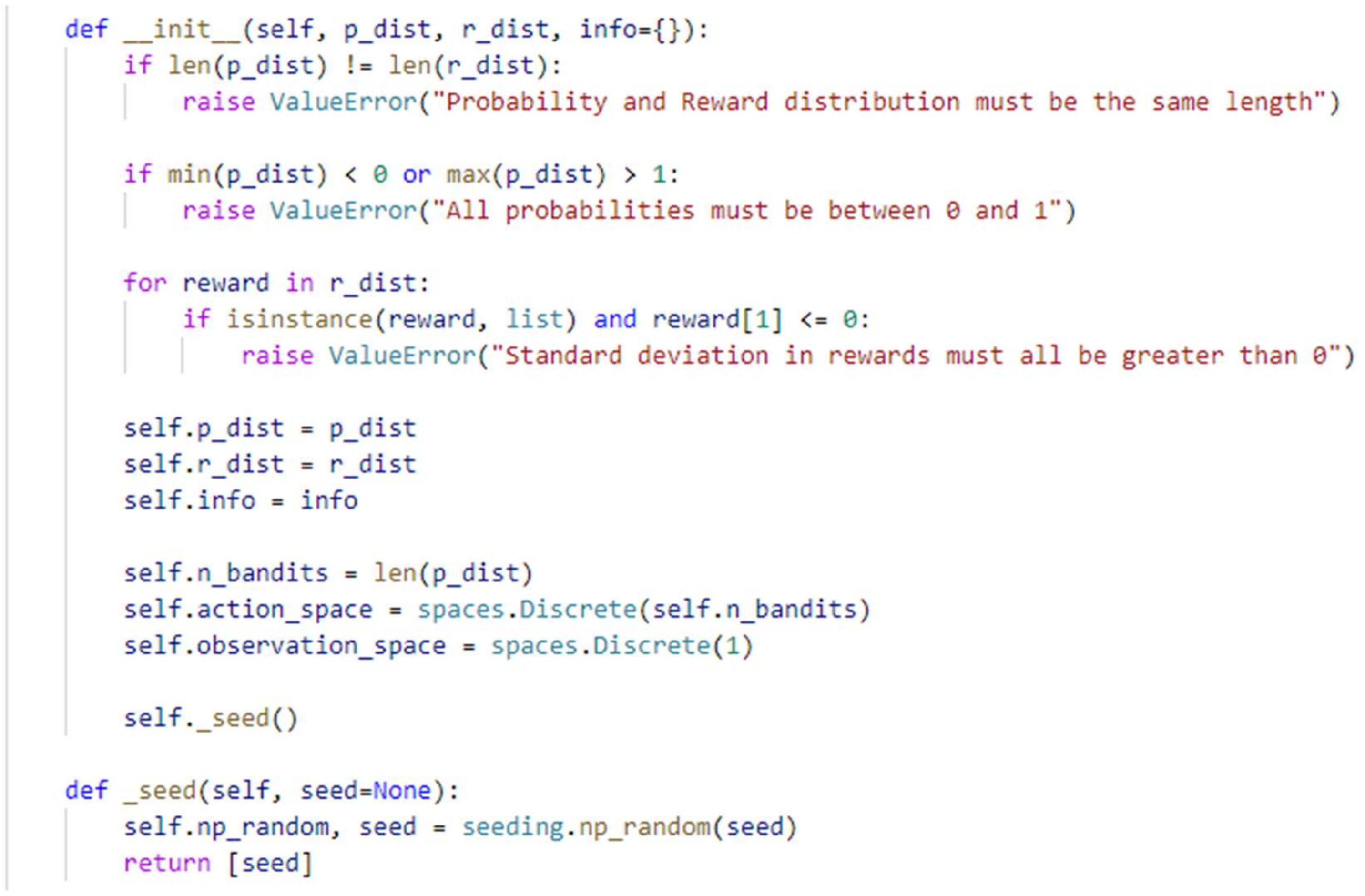
- The OpenAI gym does not have a MAB environment.
- We can define our own environment inheriting gym.Env.
- Defining BanditEnv (1/3)



```python
import numpy as np
import gym
from gym import spaces
from gym.utils import seeding

class BanditEnv(gym.Env):
    """
    Bandit environment base to allow agents to interact with the class n-armed bandit
    in different variations
    p_dist:
        A list of probabilities of the likelihood that a particular bandit will pay out
    r_dist:
        A list of either rewards (if number) or means and standard deviations (if list)
        of the payout that bandit has
    info:
        Info about the environment that the agents is not supposed to know. For instance,
        info can releal the index of the optimal arm, or the value of prior parameter.
        Can be useful to evaluate the agent's perfomance
    """
```

Defining MAB environment in gym [ex010]

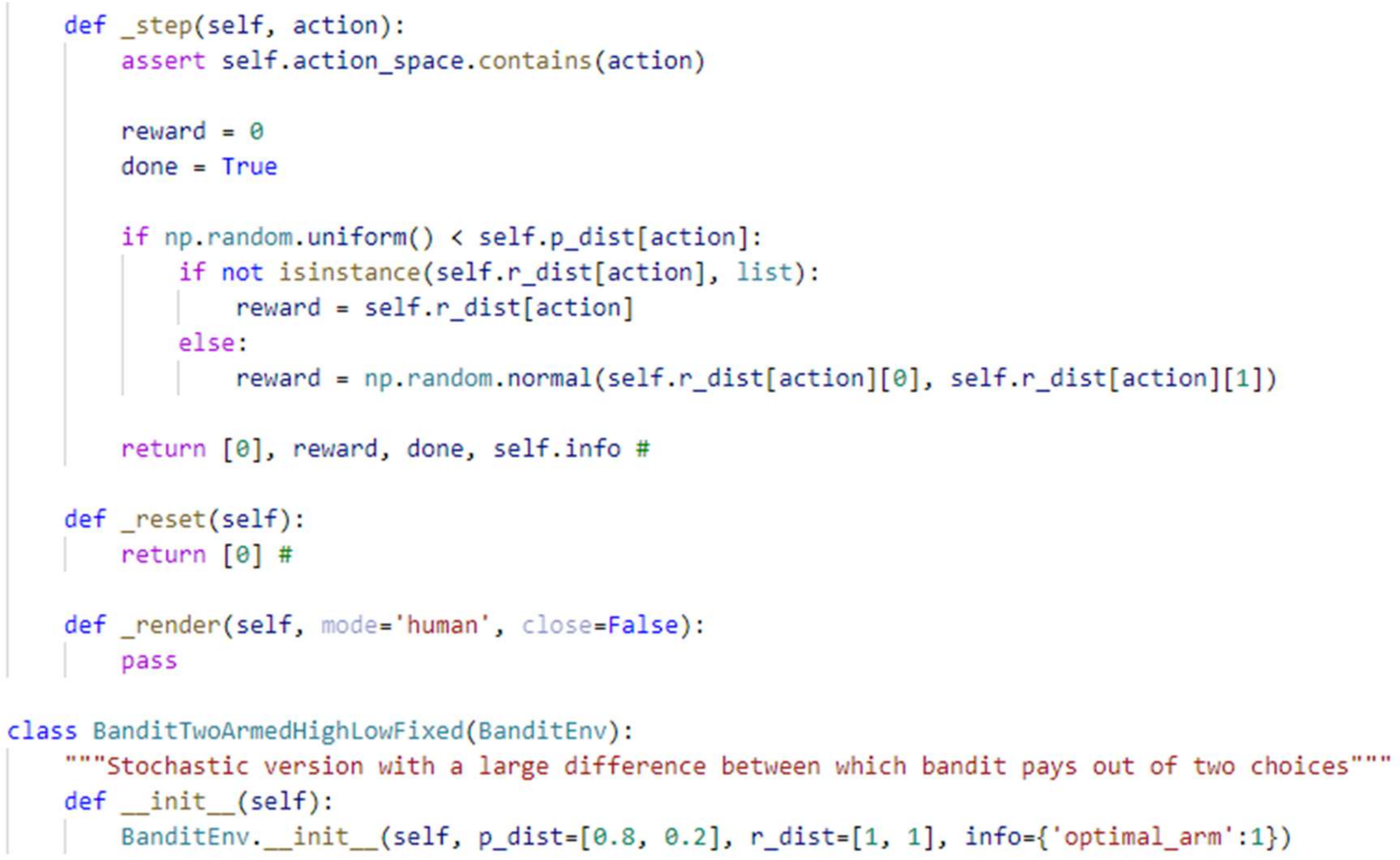
- Defining BanditsEnv (2/3)



```
    def __init__(self, p_dist, r_dist, info={}):
        if len(p_dist) != len(r_dist):
            raise ValueError("Probability and Reward distribution must be the same length")

        if min(p_dist) < 0 or max(p_dist) > 1:
            raise ValueError("All probabilities must be between 0 and 1")

        for reward in r_dist:
            if isinstance(reward, list) and reward[1] <= 0:
                raise ValueError("Standard deviation in rewards must all be greater than 0")

        self.p_dist = p_dist
        self.r_dist = r_dist
        self.info = info

        self.n_bandits = len(p_dist)
        self.action_space = spaces.Discrete(self.n_bandits)
        self.observation_space = spaces.Discrete(1)

        self._seed()

    def _seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]
```

Defining MAB environment in gym [ex010]

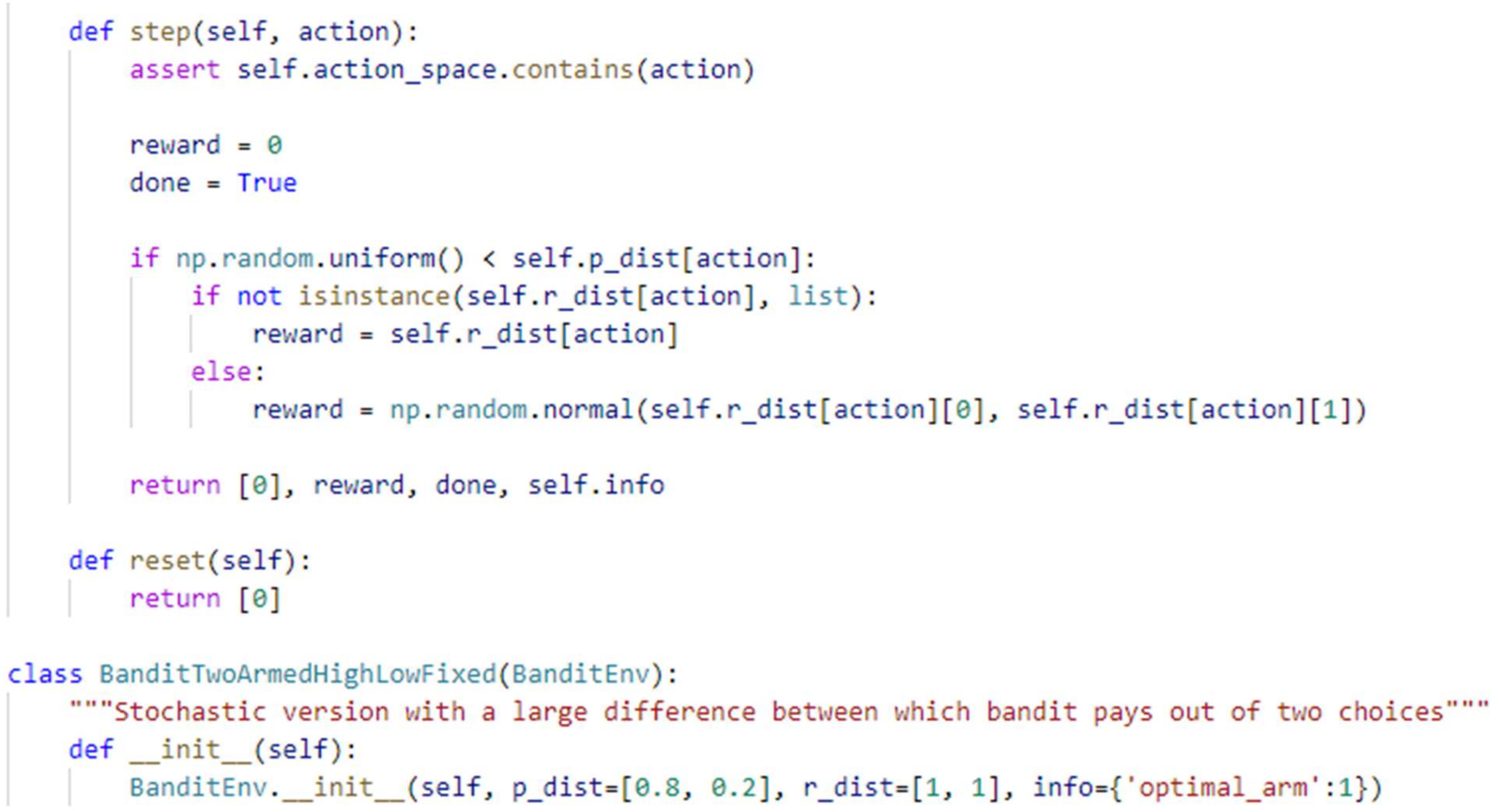
- Defining BanditsEnv (3/3)



```
    def _step(self, action):
        assert self.action_space.contains(action)

        reward = 0
        done = True

        if np.random.uniform() < self.p_dist[action]:
            if not isinstance(self.r_dist[action], list):
                reward = self.r_dist[action]
            else:
                reward = np.random.normal(self.r_dist[action][0], self.r_dist[action][1])

        return [0], reward, done, self.info #

    def _reset(self):
        return [0] #

    def _render(self, mode='human', close=False):
        pass

class BanditTwoArmedHighLowFixed(BanditEnv):
    """Stochastic version with a large difference between which bandit pays out of two choices"""
    def __init__(self):
        BanditEnv.__init__(self, p_dist=[0.8, 0.2], r_dist=[1, 1], info={'optimal_arm':1})
```

Defining MAB environment in gym [ex010]

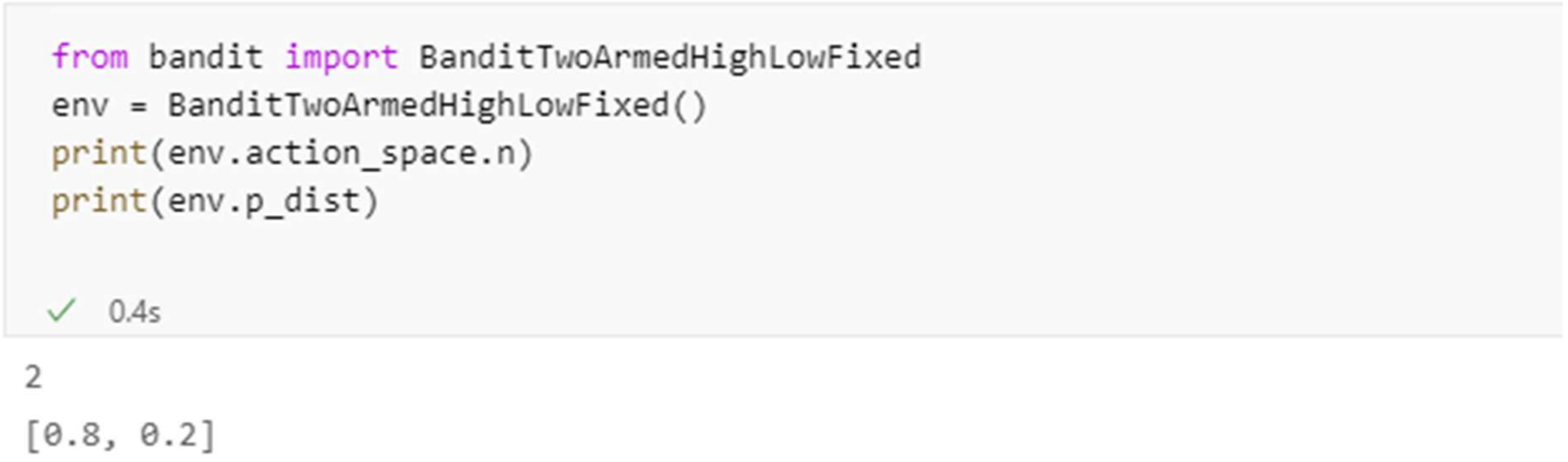
- Defining BanditsEnv (3/3)



```python
    def step(self, action):
        assert self.action_space.contains(action)

        reward = 0
        done = True

        if np.random.uniform() < self.p_dist[action]:
            if not isinstance(self.r_dist[action], list):
                reward = self.r_dist[action]
            else:
                reward = np.random.normal(self.r_dist[action][0], self.r_dist[action][1])

        return [0], reward, done, self.info

    def reset(self):
        return [0]

class BanditTwoArmedHighLowFixed(BanditEnv):
    """Stochastic version with a large difference between which bandit pays out of two choices"""
    def __init__(self):
        BanditEnv.__init__(self, p_dist=[0.8, 0.2], r_dist=[1, 1], info={'optimal_arm':1})
```

Defining MAB environment in gym

- Using the environment



```
from bandit import BanditTwoArmedHighLowFixed
env = BanditTwoArmedHighLowFixed()
print(env.action_space.n)
print(env.p_dist)
```

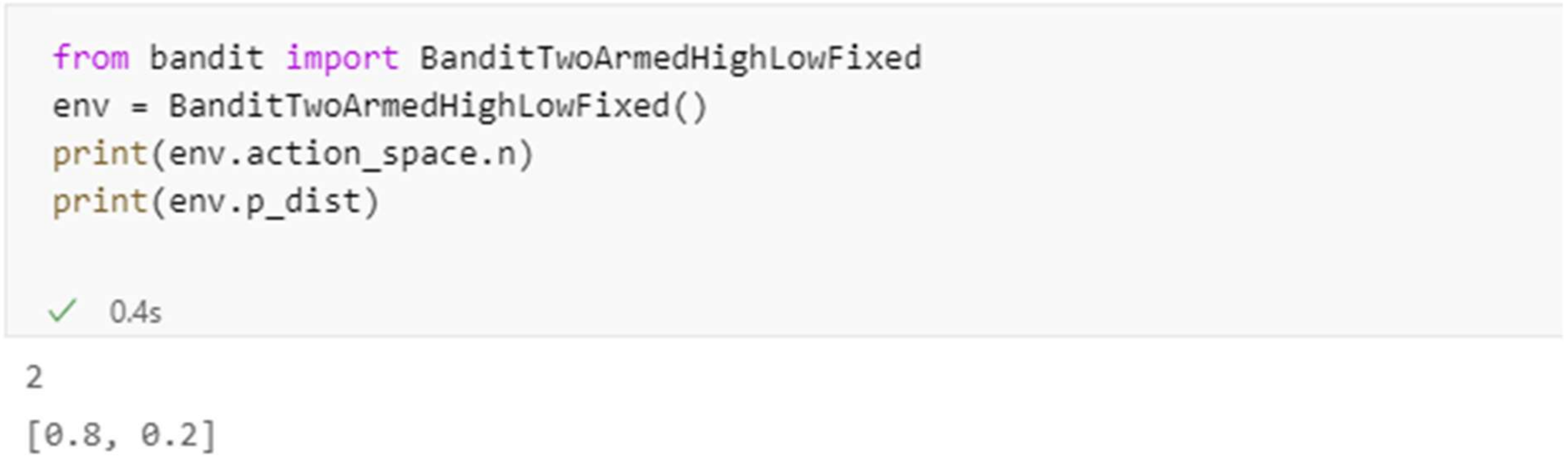
✓ 0.4s

```
2
[0.8, 0.2]
```

- p_dist is the probability of getting the reward for the k arms.
- Here, arm 1 gives reward with probability 80%, and arm 2 gives reward with probability 20%.

Defining MAB environment in gym [ex010]

- Using the environment



```python
from bandit import BanditTwoArmedHighLowFixed
env = BanditTwoArmedHighLowFixed()
print(env.action_space.n)
print(env.p_dist)
```

✓ 0.4s

```
2
[0.8, 0.2]
```

- p_dist is the probability of getting the reward for the k arms.
- Here, arm 1 gives reward with probability 80%, and arm 2 gives reward with probability 20%.

서강대학교 SOGANG UNIVERSITY

The Multi-Armed Bandit Problem

Exploration Strategies

Exploration Strategies

- Many different strategies exist to find the best arm.
- Here we look at some of the representative strategies:

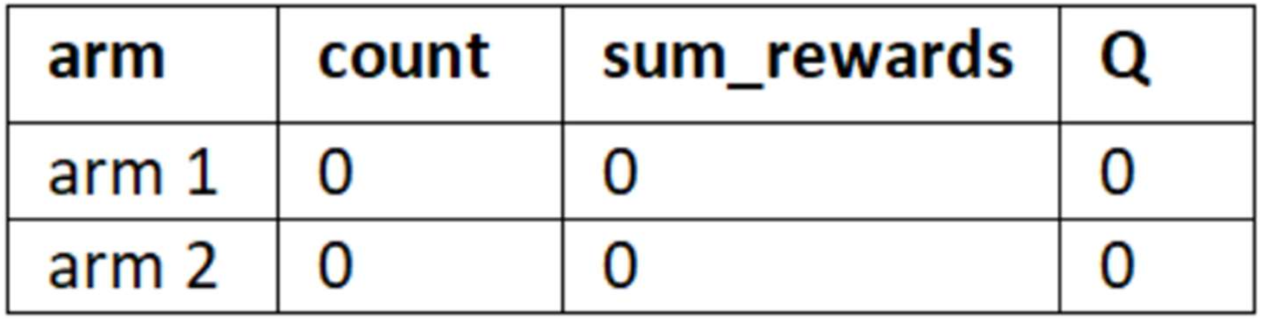
- Epsilon-greedy
- Softmax exploration
- Upper confidence bound (UCB)
- Thompson Sampling

Epsilon-Greedy

- We select the "best" arm with probability $1 - \epsilon$, and a random arm with probability ϵ.

- Suppose we have 2 arms. The winning probability for arm 1 and arm 2 is 80% and 20% respectively.

- We prepare a table with columns 'count', 'sum of rewards', and Q. Everything is initialized to zero.



arm	count	sum_rewards	Q
arm 1	0	0	0
arm 2	0	0	0

Epsilon-Greedy

- In round 1, we use the epsilon-greedy policy to select an arm.
- Suppose we select arm 1 and win. We get a reward of 1.
- We record this in the table.

arm	count	sum_rewards	Q
arm 1	1	1	1
arm 2	0	0	0

- In round 2, we probabilistically select the best arm (the arm with the highest Q value), which is arm 1, and win.

arm	count	sum_rewards	Q
arm 1	2	2	1
arm 2	0	0	0

Epsilon-Greedy

- In round 3, say we select a random arm. We select arm 2 and lose.

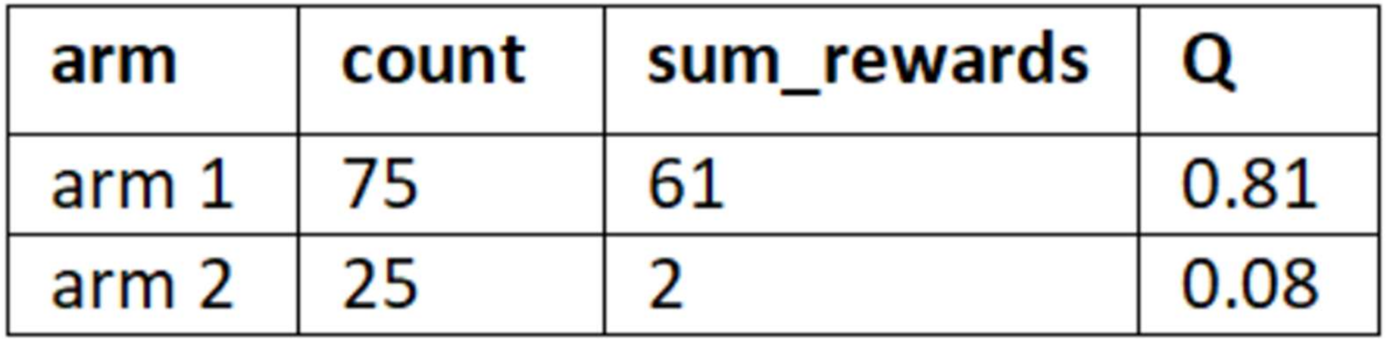
arm	count	sum_rewards	Q
arm 1	2	2	1
arm 2	1	0	0

- In round 4, we select the best arm (arm 1) and lose.

arm	count	sum_rewards	Q
arm 1	3	2	0.66
arm 2	1	0	0

Epsilon-Greedy

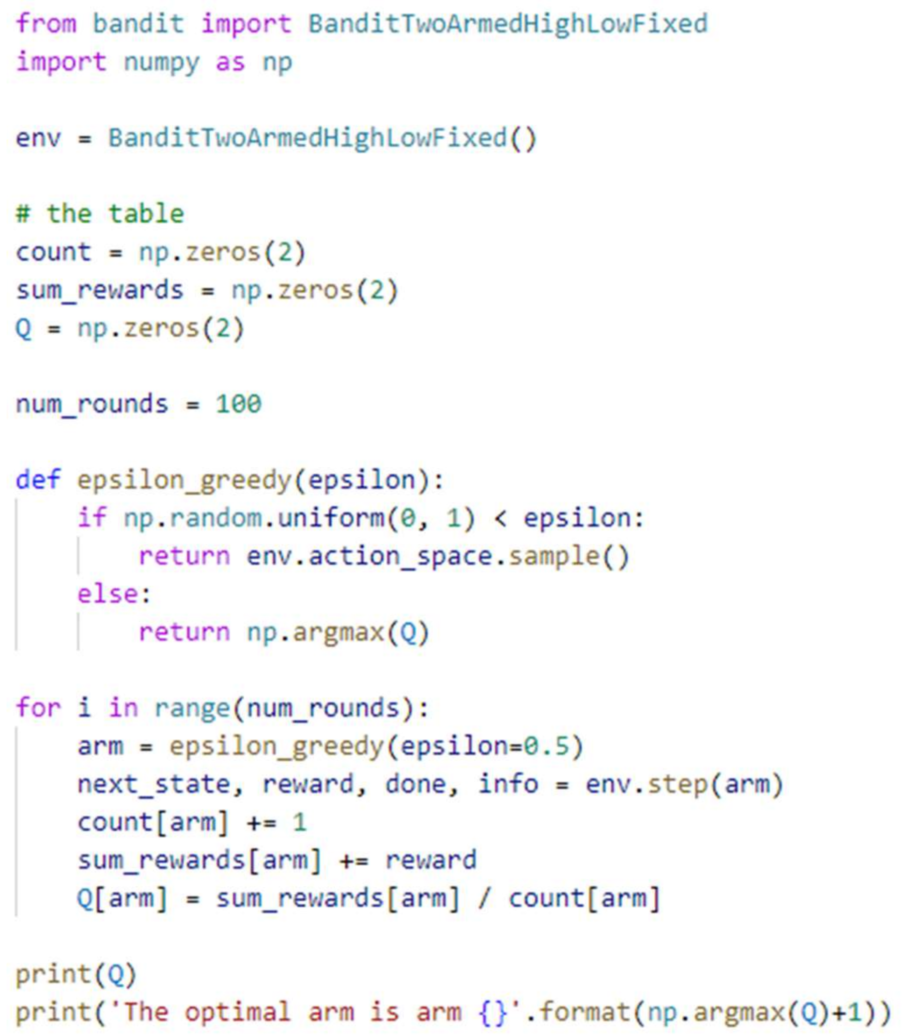
- After 100 rounds:



arm	count	sum_rewards	Q
arm 1	75	61	0.81
arm 2	25	2	0.08

- From this table, we conclude that the best arm is arm 1.

Epsilon-Greedy: Implementation [ex011]



```python
from bandit import BanditTwoArmedHighLowFixed
import numpy as np

env = BanditTwoArmedHighLowFixed()

# the table
count = np.zeros(2)
sum_rewards = np.zeros(2)
Q = np.zeros(2)

num_rounds = 100

def epsilon_greedy(epsilon):
    if np.random.uniform(0, 1) < epsilon:
        return env.action_space.sample()
    else:
        return np.argmax(Q)

for i in range(num_rounds):
    arm = epsilon_greedy(epsilon=0.5)
    next_state, reward, done, info = env.step(arm)
    count[arm] += 1
    sum_rewards[arm] += reward
    Q[arm] = sum_rewards[arm] / count[arm]

print(Q)
print('The optimal arm is arm {}'.format(np.argmax(Q)+1))
```

Softmax Exploration

- The problem of epsilon-greedy policy
  - All non-best arms are explored equally.

- Say we have 4 arms, and currently the best arm is 1.
  - According to epsilon-greedy policy, arms 2, 3, and 4 have the same probability of being explored.
  - However, if arm 3 is found in the early stage to have a very low winning probability, it could be better to give more chances to arms 2 and 4.

- To differentiate probability between non-best arms, we can assign probability based on the Q value of each arm.

Softmax Exploration

- In Softmax Exploration, the probability of selecting an arm is based on its Q value.
- However, since the probabilities must sum to 1, we convert the Q values using the Softmax function to convert to probabilities.

$$P_t(a) \propto \frac{\exp\big(Q_t(a)\big)}{\sum_{i=1}^{n} \exp\big(Q_t(i)\big)}$$

- The problem with softmax function is that is favors action with high Q value too much. To increase exploration, we use a temperature parameter T.

$$P_t(a) \propto \frac{\exp(Q_t(a)/T)}{\sum_{i=1}^{n} \exp(Q_t(i)/T)}$$

- When T is high, all arms have similar probabilities. When T is low, the arm with the highest Q value will have a very high probability.

Softmax Exploration

- Suppose we have the following Q table. Using the softmax function, we can convert the Q table to probabilities.

arm	Q
arm 1	1
arm 2	0
arm 3	0
arm 4	0

$\Rightarrow$

arm	Q	probability
arm 1	1	0.475
arm 2	0	0.174
arm 3	0	0.174
arm 4	0	0.174

- If we set $T = 30$, the probabilities become similar.

arm	Q	probability
arm 1	1	0.253
arm 2	0	0.248
arm 3	0	0.248
arm 4	0	0.248

Softmax Exploration

- In the initial phase, we do not have a clue of which arm is better than others. So we use a high T value to explore more.

- As the learning progresses, we reduce the T value in order to assign high probability to arms with high Q values.

Softmax Exploration: Implementation [ex012]

```
from bandit import BanditTwoArmedHighLowFixed
import numpy as np

env = BanditTwoArmedHighLowFixed()

# the table
count = np.zeros(2)
sum_rewards = np.zeros(2)
Q = np.zeros(2)

def softmax(T):
    denom = sum([np.exp(i/T) for i in Q])
    probs = [np.exp(i/T)/denom for i in Q]
    arm = np.random.choice(env.action_space.n, p=probs)
    return arm

num_rounds = 100
T = 50
for i in range(num_rounds):
    arm = softmax(T)
    next_state, reward, done, info = env.step(arm)
    count[arm] += 1
    sum_rewards[arm] += reward
    Q[arm] = sum_rewards[arm] / count[arm]
    T = T * 0.99

print(Q)
print('The optimal arm is arm {}'.format(np.argmax(Q)+1))
```

Upper Confidence Bound (UCB)

- Based on a principle called "Optimism in the face of uncertainty"

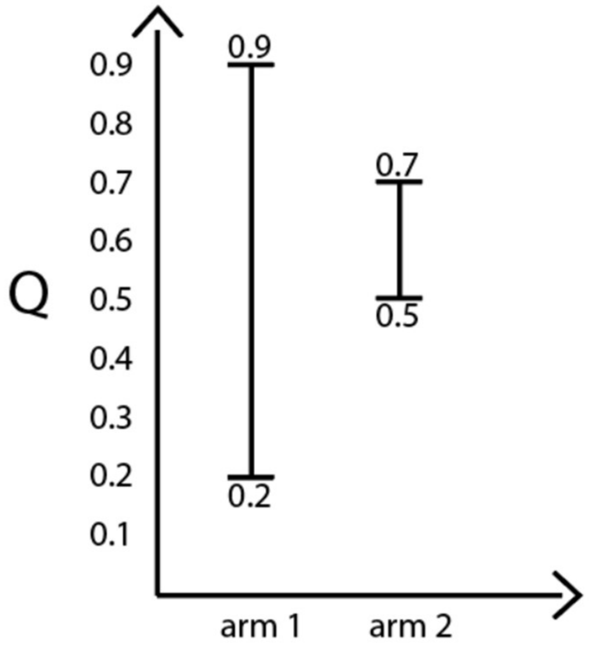
- Suppose we have two arms - arm 1 and arm 2.
- We played the game for 20 rounds by pulling a random arm in each round.
- After 20 rounds, the mean reward of arm 1 is 0.6, and the mean reward of arm 2 is 0.5.

- What is the true mean reward for arm 1 and arm 2?
- We will use the confidence interval to estimate the range of the true mean.

Upper Confidence Bound (UCB)

- Confidence Interval
  - In the graph, the mean of arm 1 is 0.5, and the confidence interval is [0.2, 0.9].
  - The mean of arm 2 is 0.6, and the confidence interval is [0.5, 0.7].
  - Arm 2 has the higher mean, but the maximum of confidence interval is higher for arm 1.
- If the confidence interval is large, it means that we are uncertain about the true mean.
- If the confidence interval is small, it means that we are more confident about the true mean.

- Typically, confidence interval becomes small if the number of samples becomes large.



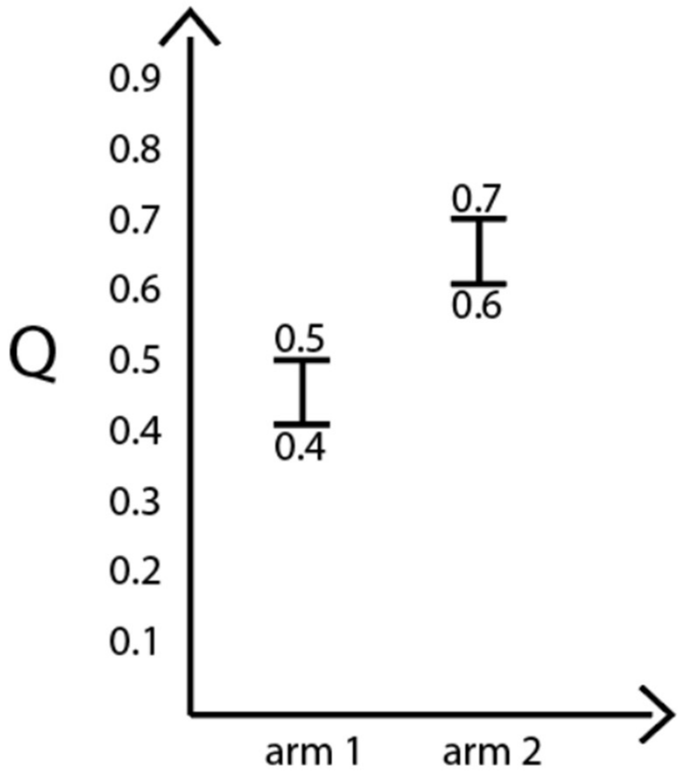
Upper Confidence Bound (UCB)

- In UCB, we always select the arm that has the high upper confidence bound.
- So in the previous example, we select arm 1, because its upper confidence bound is 0.9.

- "Optimism"
  - If arm 1 actually has a high true mean, we are selecting a good arm.
  - If arm 1 does not have a high true mean, it is still good for exploration. When the arm is explored well, its confidence interval becomes smaller.

Upper Confidence Bound (UCB)

- After many rounds, the confidence interval of arm 1 and arm 2 becomes small.
  - We can find out that arm 2 has a better payout.



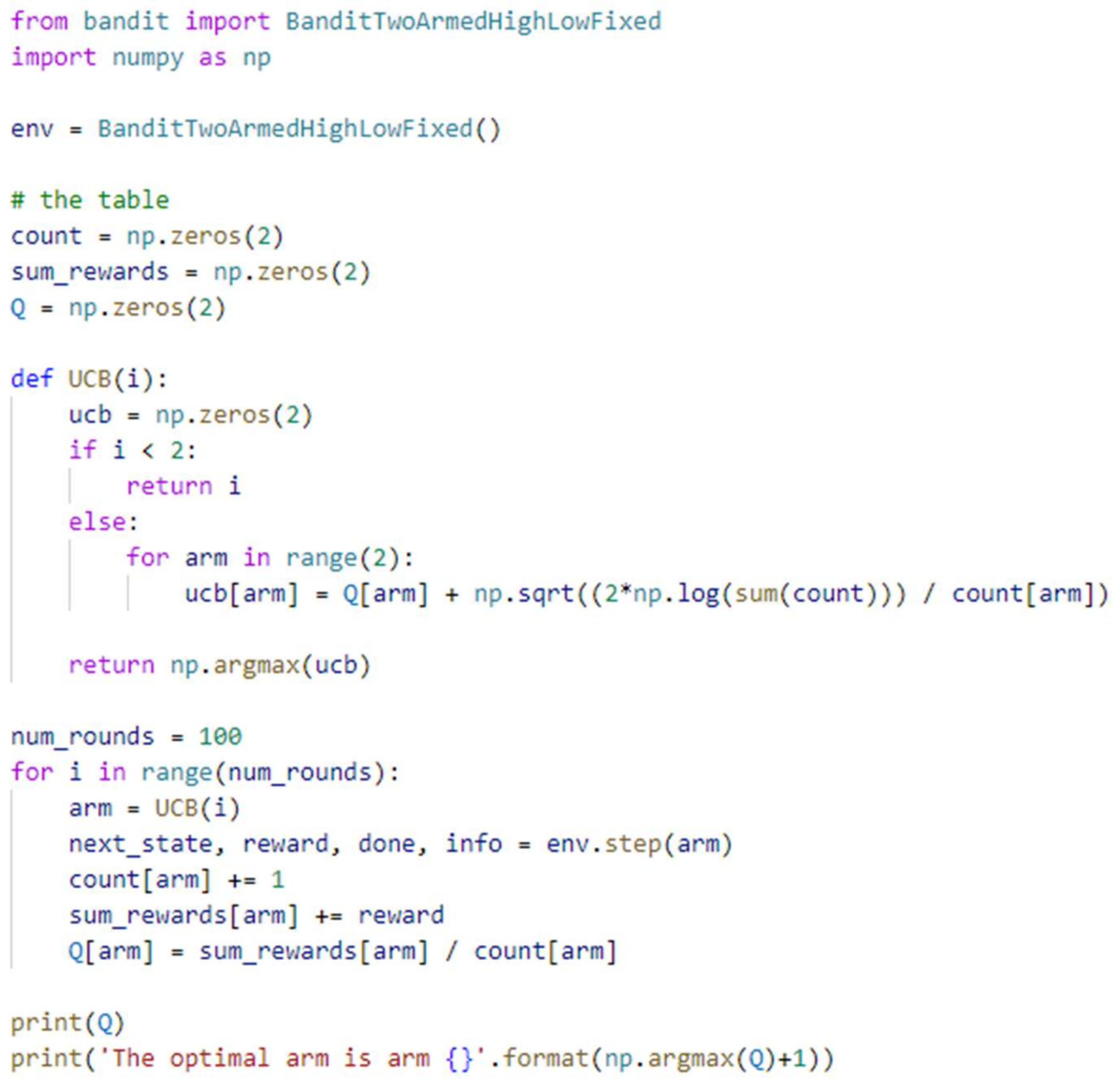
Upper Confidence Bound (UCB)

- To select an arm, we calculate UCB of each arm from the Q table.

- $N(a)$ is the number of times arm a is pulled, and t is the number of total rounds. Then, UCB of arm a is:

$$\mathrm{UCB}(a) = Q(a) + \sqrt{\frac{2\log(t)}{N(a)}}$$

UCB: Implementation [ex013]



```python
from bandit import BanditTwoArmedHighLowFixed
import numpy as np

env = BanditTwoArmedHighLowFixed()

# the table
count = np.zeros(2)
sum_rewards = np.zeros(2)
Q = np.zeros(2)

def UCB(i):
    ucb = np.zeros(2)
    if i < 2:
        return i
    else:
        for arm in range(2):
            ucb[arm] = Q[arm] + np.sqrt((2*np.log(sum(count))) / count[arm])

    return np.argmax(ucb)

num_rounds = 100
for i in range(num_rounds):
    arm = UCB(i)
    next_state, reward, done, info = env.step(arm)
    count[arm] += 1
    sum_rewards[arm] += reward
    Q[arm] = sum_rewards[arm] / count[arm]

print(Q)
print('The optimal arm is arm {}'.format(np.argmax(Q)+1))
```

Thompson Sampling

- Thompson Sampling is based on a beta distribution.

- beta (β) distribution

$$f(x) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

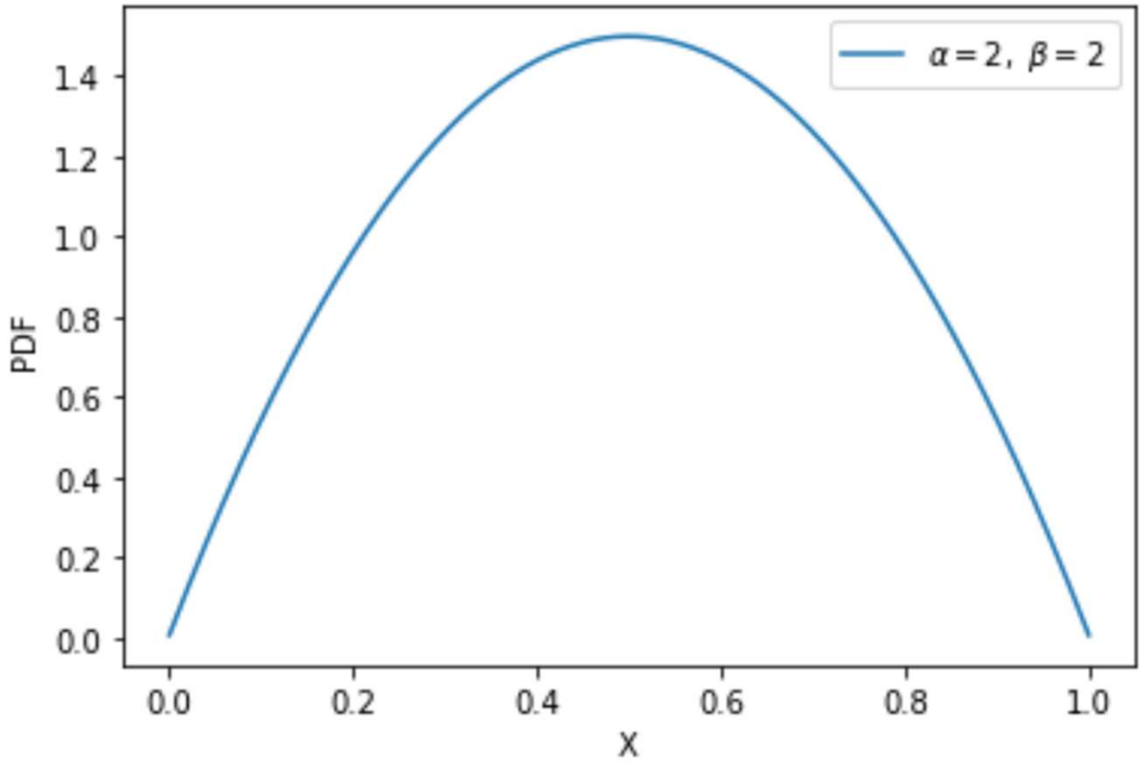
- where $B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$ and $\Gamma(.)$ is the gamma function.

- Gamma (Γ) function: an extension of the factorial function

$$\Gamma(n) = (n-1)! \qquad \Gamma(z) = \int_0^\infty x^{z-1}e^{-x}\,dx$$

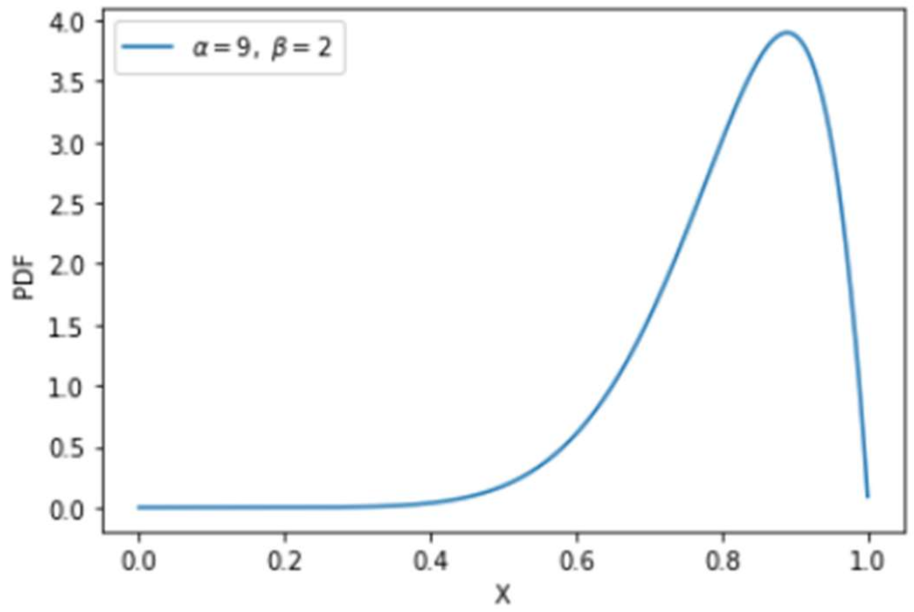
Thompson Sampling

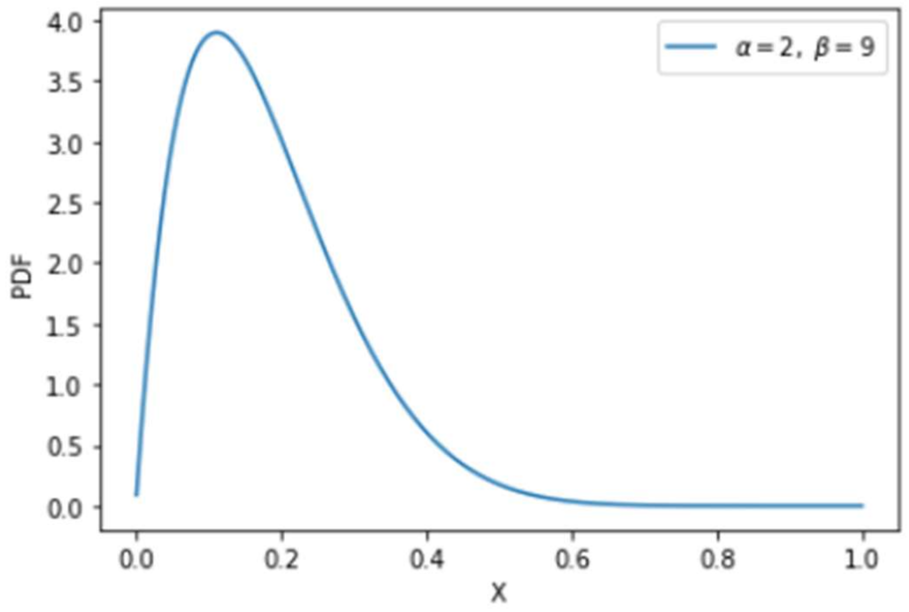
- In a beta distribution, the shape of distribution is controlled by the two parameters α and β.
- If $\alpha = \beta$, then the beta distribution is a symmetric distribution.



Thompson Sampling

- When the value of α is higher than β then we have a probability close to 1 than 0.
- When the value of β is higher than α then we have a probability close to 1 than 0.

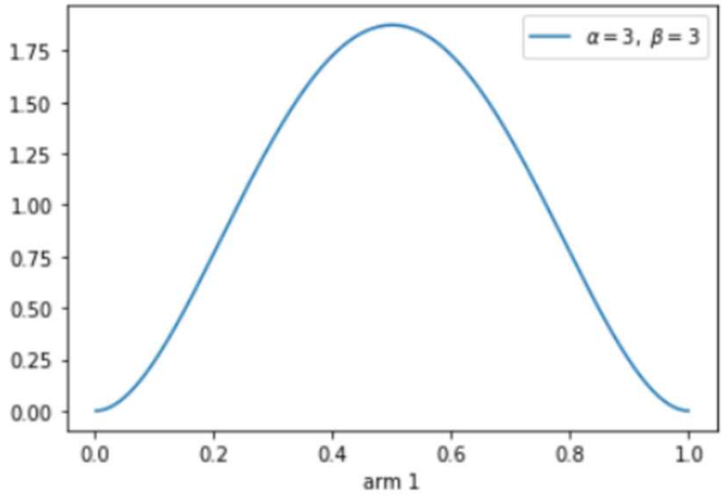


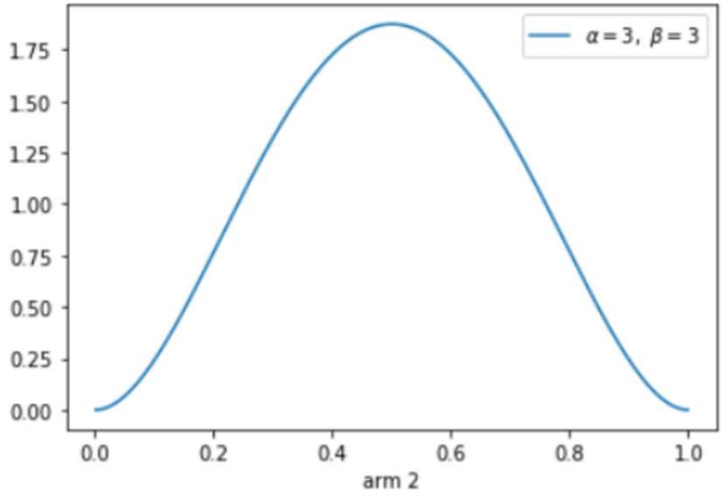


Thompson Sampling

- Thompson Sampling is a probabilistic method based on a prior distribution.
- The prior distribution is the beta distribution.
  - We are assuming that the probability distribution of each arm follows beta distribution.

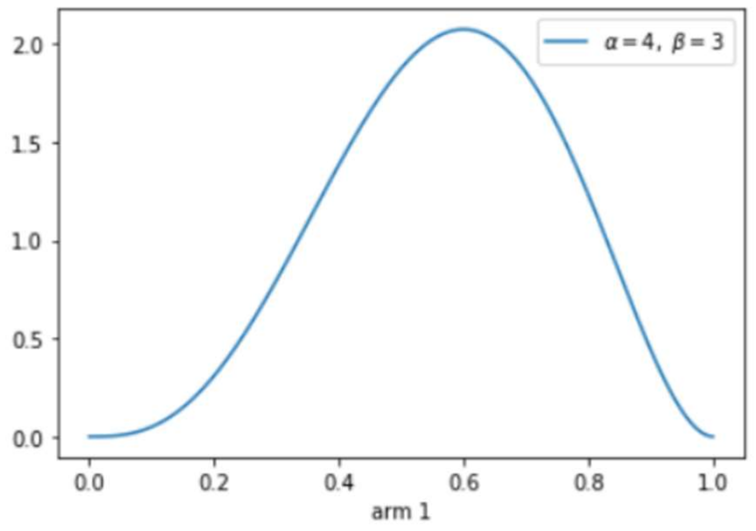
- First, we initialize α and β to the same value for all arms.

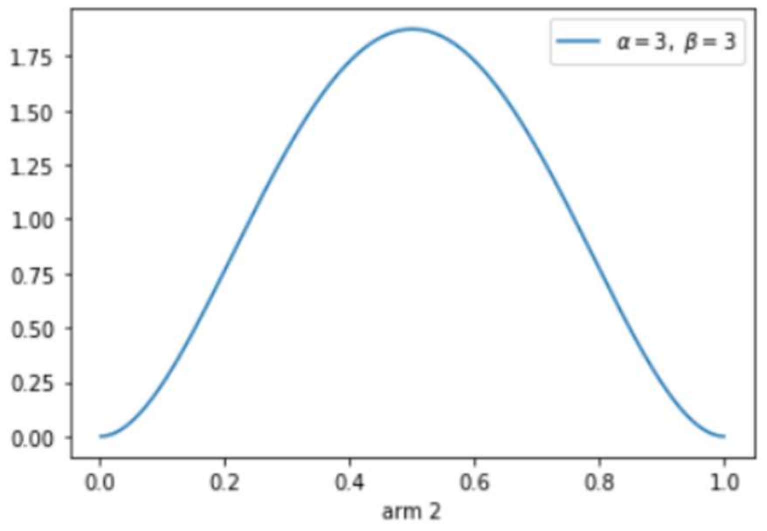




Thompson Sampling

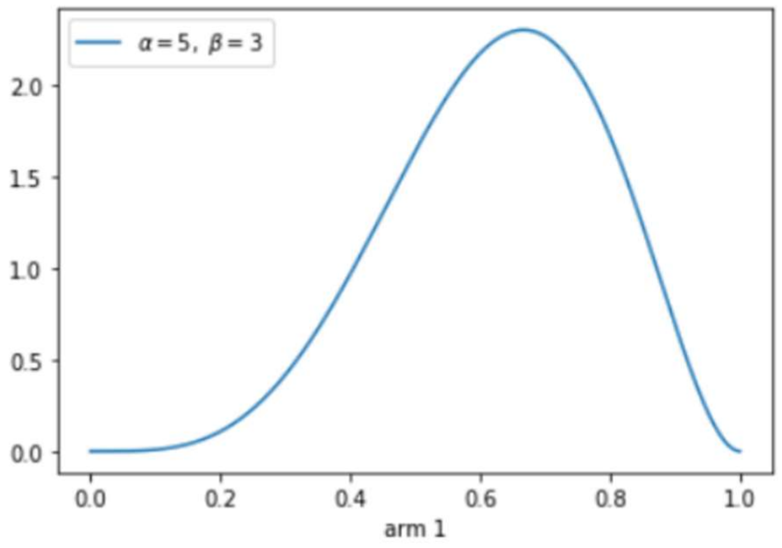
- In the first round, we randomly sample a value from these two distributions and select the arm that has the maximum value.
- Suppose the sampled value of arm 1 is high. We pull arm 1.
- Suppose we win the game. Then, we increment α of arm 1.
- Now, the arm 1's beta distribution has slightly high probability closer to 1 compared to arm 2.

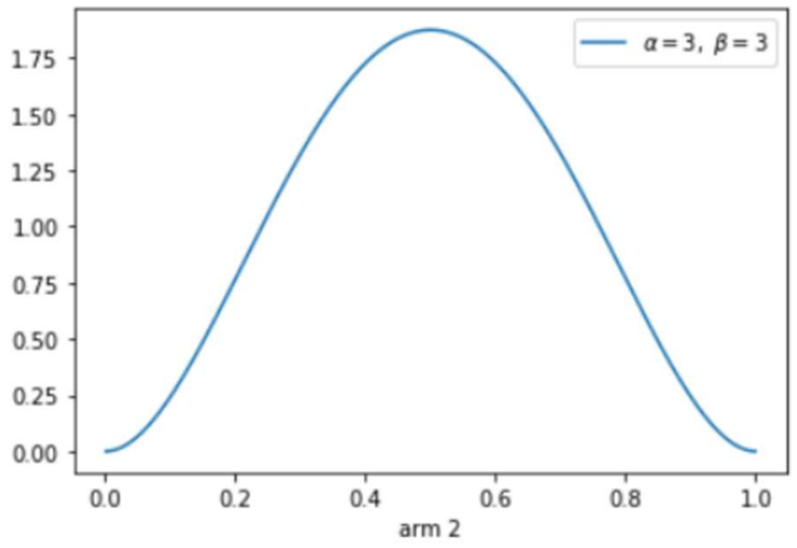




Thompson Sampling

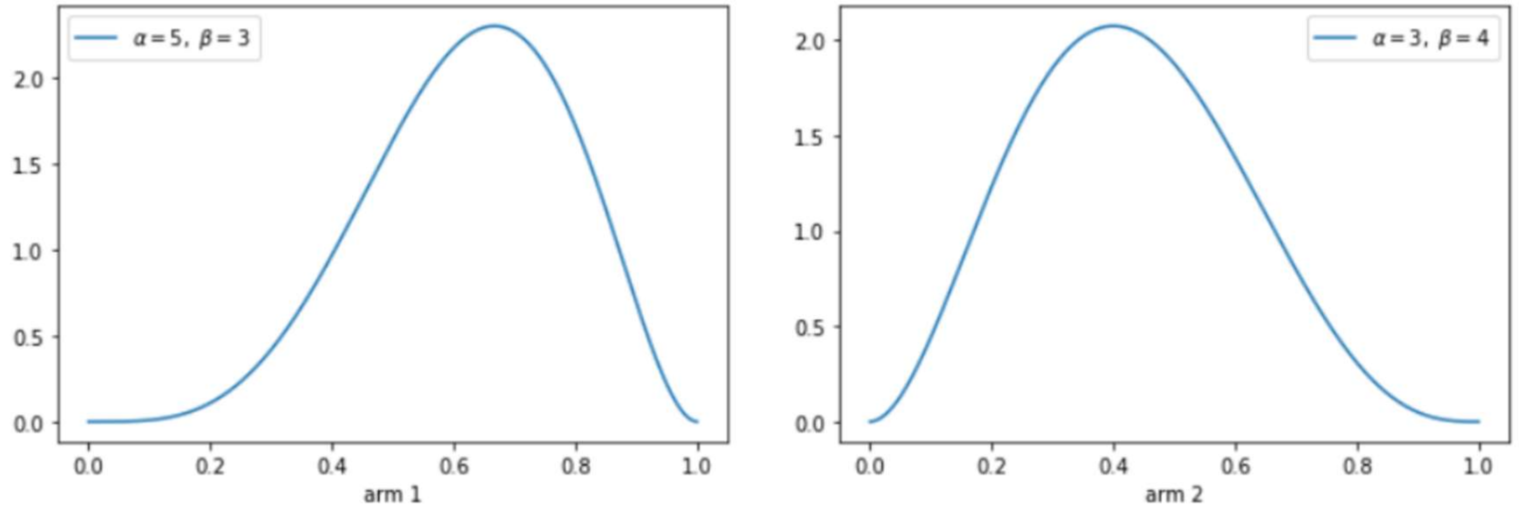
- In the second round, we again sample a value from the two distribution and select the arm with the maximum value.
- Suppose we pull arm 1 and win. Then, α of arm 1 is incremented again.





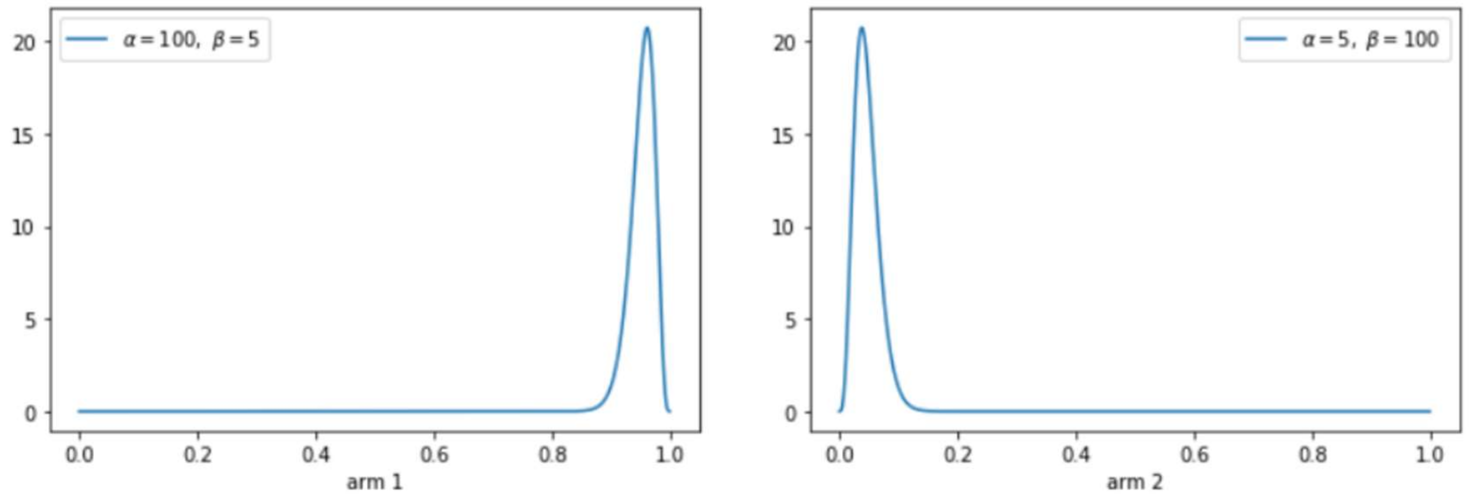
Thompson Sampling

- In the third round, we again sample a value from the two distribution.
- This time, let us suppose we choose arm 2 and lose.
- Then, we increment β of arm 2.



Thompson Sampling

- We continue this for multiple rounds. As we do, the distributions of the arms become closer to their true distributions.

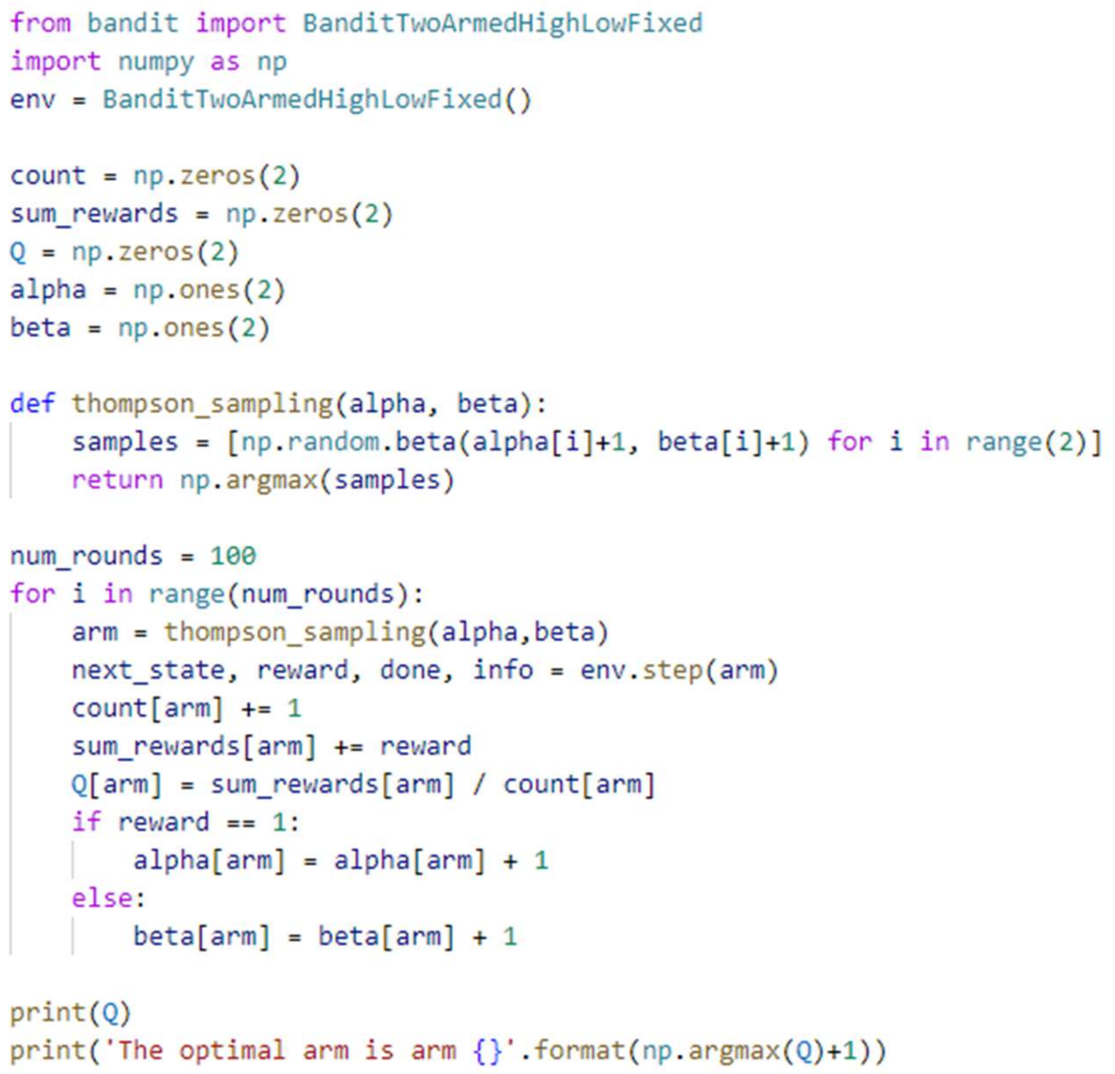


Thompson Sampling

- Algorithm

1. Initialize the beta distribution with alpha and beta set to equal values for all *k* arms
2. Sample a value from the beta distribution of all *k* arms
3. Pull the arm whose sampled value is high
4. If we win the game, then update the alpha value of the distribution to $\alpha = \alpha + 1$
5. If we lose the game, then update the beta value of the distribution to $\beta = \beta + 1$
6. Repeat *steps* 2 to 5 for many rounds

Thompson Sampling: Implementation [ex014]



```python
from bandit import BanditTwoArmedHighLowFixed
import numpy as np
env = BanditTwoArmedHighLowFixed()

count = np.zeros(2)
sum_rewards = np.zeros(2)
Q = np.zeros(2)
alpha = np.ones(2)
beta = np.ones(2)

def thompson_sampling(alpha, beta):
    samples = [np.random.beta(alpha[i]+1, beta[i]+1) for i in range(2)]
    return np.argmax(samples)

num_rounds = 100
for i in range(num_rounds):
    arm = thompson_sampling(alpha,beta)
    next_state, reward, done, info = env.step(arm)
    count[arm] += 1
    sum_rewards[arm] += reward
    Q[arm] = sum_rewards[arm] / count[arm]
    if reward == 1:
        alpha[arm] = alpha[arm] + 1
    else:
        beta[arm] = beta[arm] + 1

print(Q)
print('The optimal arm is arm {}'.format(np.argmax(Q)+1))
```

Variations of MAB

- Contextual bandit
  - In each iteration, an agent has to choose between arms.
  - Before making the choice, the agent sees a d-dimensional feature vector (context vector), associated with the current iteration.
  - The learner uses the context vectors along with the rewards of the arms played in the past to make the choice of the arm to play in the current iteration.
  - Over time, the learner's aim is to collect enough information about how the context vectors and rewards relate to each other.
  - The agent learns to predict the best arm in the context of the feature vector.

- Contextual bandit is widely used for personalizing content according to user's behavior.
  - Recommending movies based on the user's behavior

Variations of MAB

- Adversarial bandit
  - At each iteration, an agent chooses an arm and an adversary simultaneously chooses the payoff structure for each arm.
- Infinite-armed bandit
  - The "arms" are a continuous variable in K dimensions.
- Non-stationary bandit
  - The underlying model can change during play.
- Dueling bandit
  - The gambler is allowed to pull two levers at the same time, but he only gets a binary feedback telling which lever provided the best reward.

End of Chapter

- Do you understand the difference between various exploration strategies?
- Can you write [ex010]-[ex014] yourself?

End of Class

Questions?

Email: jso1@sogang.ac.kr