Multi Armed Bandit

Shuvrajeet Das

26-06-2020

1 Basics of Reinforcement Learning:

- RL problem mostly comprises of two components:
  - Environment
  - Agent

  Most of the problems we deal with have a single agent. On the other hand, if there is more than one agent ,that problem is called **Multi-Agent RL**, or **MARL** for short. In MARL, the relationship between the agents could be cooperative, competitive or cooperative-competitive.
- The objective of a RL problem is the agent learning what to do in the environment. This defines which **action** to take in the environment.
- The set of all the information that precisely and suffiently describes the situation in the environment is called **State**.
- In some problems, the knowledge of the state is fully known to the agent. In other problems, the knowledge of the state is partially known to the agent. Thus defining the two types of states namely:
  - **Fully Known State**
  - **Partially Known State**

So far, we have not really defined what makes an action good or bad. In RL, every time the agent takes an action, it receives a reward from the environment (albeit it is sometimes zero). Reward could mean many things in general, but in RL terminology, its meaning is very specific: it is a scalar number. The greater the number is, the higher the reward also is. In an iteration of an RL problem, the agent observes the state the environment is in (fully or partially) and takes an action based on its observation.

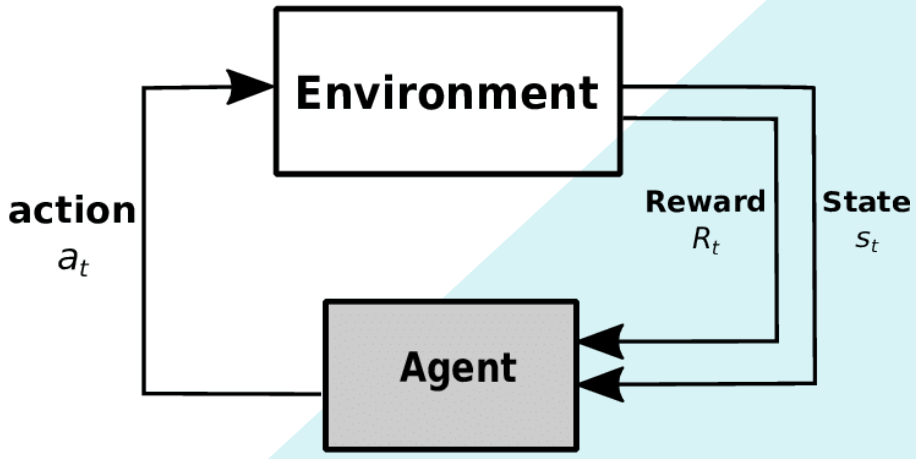

Figure 1: RL process

Remember that in RL, the agent is interested in actions that will be beneficial over the long term.

- If it is the former, the problem is described as an **episodic task**, where an episode is defined as the sequence of interactions from an initial state to a **terminal state**.
- If the problem is defined over an infinite horizon, it is called a **continuing task**.

2 Multi-armed Bandit Problem:

A MAB problem is all about identifying the best action among a set of actions available to an agent through trial and error, such as figuring out the best look for a website among some alternatives, or the best ad banner to run for a product. We will focus on the more common variant of MABs where there are k discrete actions available to the agent, also known as a k-armed bandit problem.

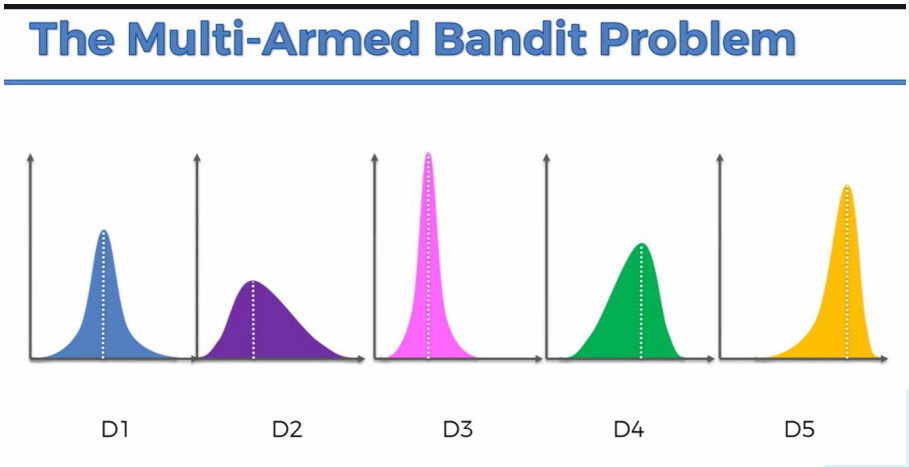

Figure 2: Multi-Armed Bandit Reward Distribution

2.1 Problem Definition:

The MAB problem is defined as follows:

- When the lever gets pulled, the agent receives a reward from the environment under a certain distribution.
- Although the machines looks same, the reward distribution is different.

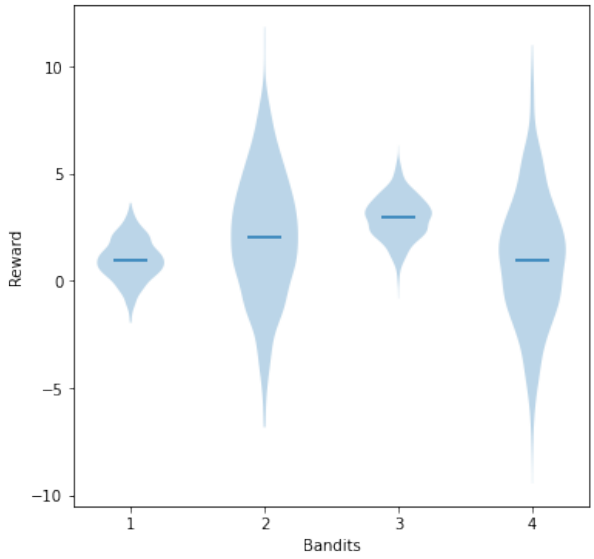

Figure 3: MAB Reward Distribution

The gambler who is interested in the best reward is the one who pulls the lever with the highest reward. So, in each turn, the agent has to choose the lever with the highest reward. Initially, the gambler has no knowledge of the reward distribution.

Clearly, the gambler needs to find a balance between exploiting the one that has been the best so far and exploring the alternatives. Why is that needed? Well, because the rewards are stochastic. A machine that won't give the highest average reward in the long term may have looked like the best just by chance!

2.2 Summary:

So, to summarize what a MAB problem looks like, we can state the following:

- The agent takes sequential actions. After each action, a reward is received.
- An action affects only the immediate reward, not the subsequent rewards.
- There is no "state" in the system that changes with the actions that the agent takes.
- There is no input that the agent uses to base its decisions on. That will come later in the next chapter when we discuss contextual bandits.

3 The Bandit Exercise:

- First, let's create a class for a single slot machine that gives a reward from a normal (Gaussian) distribution with respect to a given mean and standard deviation:

```
class GaussBandit:
    def __init__(self,mean,std):
        self.mean = mean
        self.std = std

    def pull_lever(self):
        return np.random.normal(loc=self.mean,scale=self.std)
```

- Next, we create a class that will simulate the game:

```
class MAB:
    def __init__(self,n_bandits,bandit_l,random):
        self.n_bandits = n_bandits
        self.bandit_l = bandit_l
        # for increasing the randomnes
        if random:
            np.random.shuffle(self.bandit_l)

        self.reset_game()

    def reset_game(self):
        self.avg_reward = 0
        self.rewards = []
        self.total_reward = 0
        self.n_games = 0

    def game_user(self):
        self.reset_game()
        print('Multi arm bandit')
        print('NB : 0 to cancel')
        while True:
            n = int(input('Select between 1 - 4: '))
            if n == 0:
                print('Game Ended')
                break
            elif n >= 1 and n <= 4:
                choice = self.bandit_l[n-1]
                reward = choice.pull_lever()
                self.n_games += 1
                self.rewards.append(reward)
                self.total_reward = sum(self.rewards)
                self.avg_reward = self.total_reward/self.n_games
                print()
                print('Reward on',n,':',reward)
```

```
                print('Avg reward on',self.n_games,':',self.avg_reward)
                print()
```

A game instance receives a list of slot machines as inputs. It then shuffles the order of the slot machines so that you won't recognize which machine gives the highest average reward.

- Then, we create some slot machines and a game instance:

```
    gb1 = GaussBandit(1,1)
    gb2 = GaussBandit(2,3)
    gb3 = GaussBandit(3,1)
    gb4 = GaussBandit(1,3)
    slot_list = [gb1,gb2,gb3,gb4]
```

The output:

```
Multi arm bandit
NB : 0 to cancel
Select between 1 - 4: 1

Reward on 1 : 1.4216402549120943
Avg reward on 1 : 1.4216402549120943

Select between 1 - 4: 2

Reward on 2 : 0.07161096548750612
Avg reward on 2 : 0.7466256101998002

Select between 1 - 4: 3

Reward on 3 : 2.2549546779528424
Avg reward on 3 : 1.2494019661174809

Select between 1 - 4: 4

Reward on 4 : 2.389883798960037
Avg reward on 4 : 1.53452242432812
```

- As we enter our choices we will observe the rewards.

```
Select between 1 - 4: 3

Reward on 3 : 2.2549546779528424
Avg reward on 3 : 1.2494019661174809

Select between 1 - 4: 4

Reward on 4 : 2.389883798960037
Avg reward on 4 : 1.53452242432812
```

4 A/B/n Testing:

One of the most common exploration strategies is A/B testing. A/B testing is a method to determine the best alternative among a set of alternatives.

4.1 Notation:

- First, we denote the reward of the best alternative as R after selecting action a for the i^{th} time by R_i.
- The average reward obeserved after n times is Q_n

$$Q_n = \frac{1}{n-1}\sum_{i=1}^{n-1} R_i$$

- This is also called action value of α. Here, this is Q_n estimates of the action value after selecting this action $n-1$ times.
- Now, we need a bit of simple algebra and we will have a very convenient formula to update the action values:

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i$$

$$\frac{R_1 + R_2 + ... + R_{n-1}}{n-1} + \frac{R_n}{n}$$

$$\frac{n-1}{n-1}.\frac{R_1 + R_2 + ... + R_{n-1}}{n} + \frac{R_n}{n}$$

$$\frac{n-1}{n}.\frac{R_1 + R_2 + ... + R_{n-1}}{n-1} + \frac{R_n}{n}$$

$$\frac{n-1}{n}.Q_n + \frac{R_n}{n}$$

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

4.2 A/B Testing Approach:

- We start with creating the variables to keep track of the rewards in the experiment:

```
train_size = 10000
test_size = 1000
```

```
3
4  Q = np.zeros(self.n_bandits)
5  N = np.zeros(self.n_bandits)
6  total_reward = 0
7  avg_reward = []
```

- Run the experiment for train_size times:

```
1  for i in range(train_size):
2      choice = np.random.randint(self.n_bandits)
3      R = self.bandit_l[choice].pull_lever()
4      N[choice] += 1
5      Q[choice] += (1/N[choice])*(R-Q[choice])
6      total_reward += R
7      avg_reward.append(total_reward/(i+1))
```

Remember that we are using the average reward to estimate the action value. We update the counter value after each time we pull the lever.

- At the end of the test period, we choose the winner as the one that has achieved the highest action value:

```
1      best_action = np.argmax(Q)
```

- Display the results:

```
1  Best one is : 3
2  Reward approx train: 1.7285412210846047
3  Reward approx test: 3.0086961900869422
```

4.3 The A/B Testing Results:

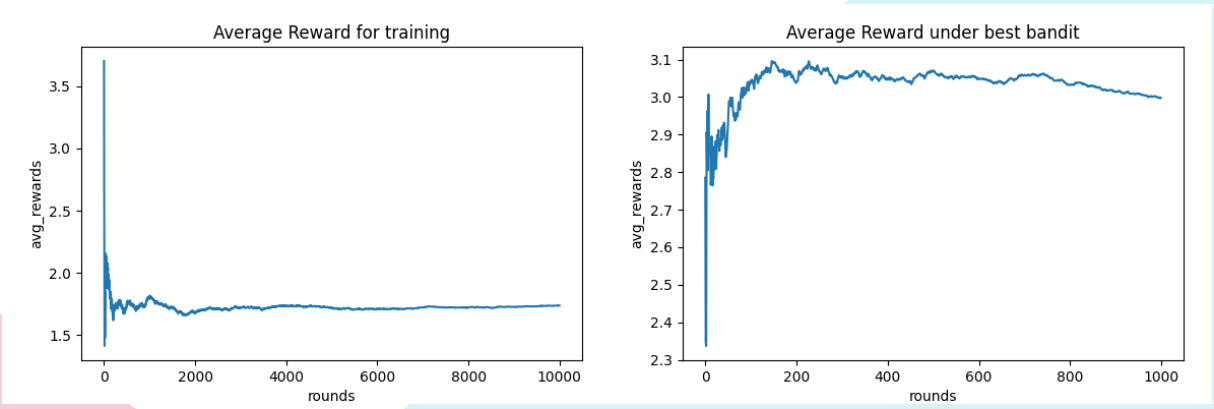

Figure 4: The results

4.4 A/B Testing pros and cons:

- A/B/n testing is inefficient as it does not modify the experiment dynamically by learning from the observations.
- It is unable to correct a decision once it's made.
- It is unable to adapt to changes in a dynamic environment.
- The length of the test period is a hyperparameter to tune, affecting the efficiency of the test.

5 Epsilon-Greedy Strategy:

Another exploration strategy is epsilon-greedy. Epsilon-greedy is a method to explore the best alternative among a set of alternatives. This approach suggests, most of the time, greedily taking action that is best according to the rewards observed by that point in time.Here, ϵ is a number between 0 and 1, usually closer to zero (for example, 0.1) to "exploit" in most decisions.

5.1 Epsilon-Greedy Approach:

- Similar to A/B testing, we start with creating the variables to keep track of the rewards in the experiment:

```
train_size = 10000
test_size = 1000

eps = 0.1
Q = np.zeros(self.n_bandits)
N = np.zeros(self.n_bandits)
total_reward = 0
avg_reward = []
```

- Run the experiment for train_size times:

```
for i in range(train_size):
    if np.random.uniform() <= eps:
        choice = np.random.randint(self.n_bandits)
    else:
        choice = np.argmax(Q)
    R = self.bandit_l[choice].pull_lever()
    N[choice] += 1
    Q[choice] += (1/N[choice])*(R-Q[choice])
    total_reward += R
    avg_reward.append(total_reward/(i+1))
```

Remember that we are using the average reward and the best reward after a getting a certain ϵ value passing the threshold to estimate the action value. We update the counter value after each time we pull the lever.

- At the end of the test period, we choose the winner as the one that has achieved the highest action value:

```
best_action = np.argmax(Q)
```

- Display the results:

```
Best one is : 3
Reward approx train: 2.876162198776939
Reward approx test: 3.001766713295964
```

5.2 The Epsilon-Greedy Results:

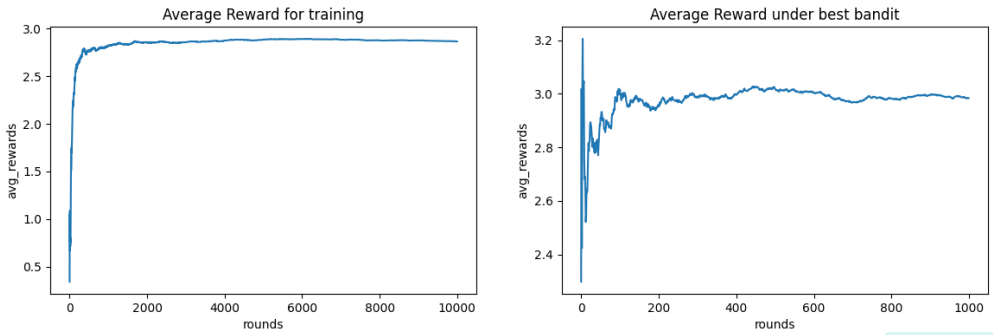

Figure 5: The results

Rewards under multiple epsilon values:

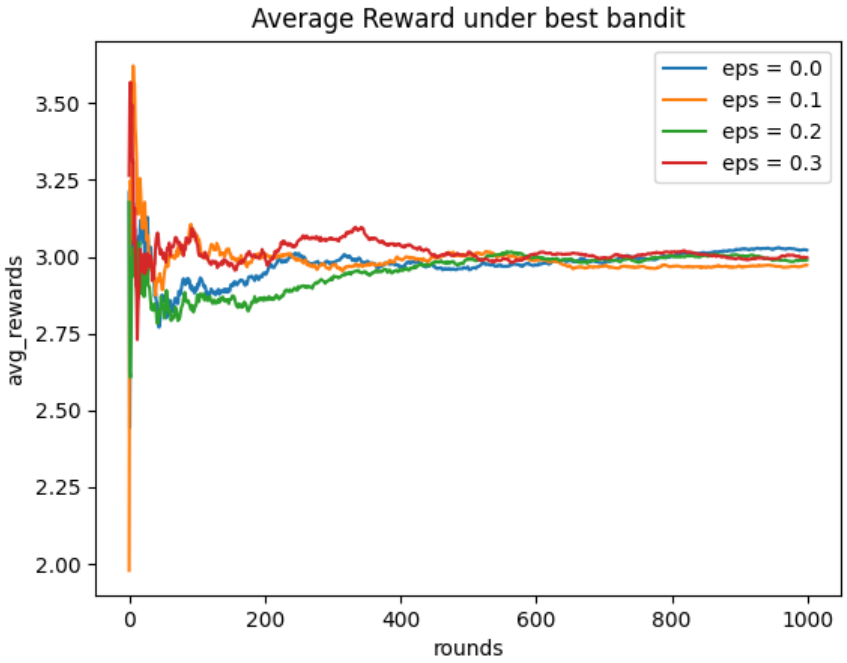

Figure 6: The results

5.3 Epsilon-Greedy pros and cons:

- ϵ-greedy actions and A/B/n tests are similarly inefficient and static in allocating the exploration budget.
- With ϵ-greedy actions, exploration is continuous, unlike in A/B/n testing.
- The ϵ-greedy actions approach could be made more efficient by dynamically changing the ϵ value.
- The ϵ-greedy actions approach could be made more dynamic by increasing the importance of more recent observations.
- Modifying the ϵ-greedy actions approach introduces new hyperparameters, which need to be tuned.