

Multi-Armed Bandits

David I. Inouye



The multi-armed bandit problem is inspired by a row of slot machines

- A gambler is the agent.
- The row of slot machines is the environment.
- The agent can take an action by pulling a slot machine's "arm".
- The slot machine payout (or lack thereof) is the reward signal.
- **Goal:** Maximize total payout over time.



<https://medium.com/growth-book/guide-to-multi-arm-bandits-what-is-it-and-why-you-probably-shouldnt-use-it-ecc9bb2e5a84>

Our Simulated Environment

We'll simulate a **Bernoulli Bandit** environment:

- Agent can choose from multiple actions
- Each action gives a binary reward (0 or 1)
- Reward probabilities are drawn from a Beta(1,2) distribution
- Probabilities are **unknown** to the agent
- Agent must estimate probabilities while maximizing reward



Environment Implementation

```
1 import torch
2 import matplotlib.pyplot as plt
3
4 class BernoulliBanditEnvironment():
5     def __init__(self, n_actions):
6         probs = torch.distributions.Beta(1,2).sample((n_actions,))
7         self.reward_dists = [
8             torch.distributions.Bernoulli(probs=p)
9             for p in probs
10        ]
11
12    def get_reward(self, action_index):
13        return self.reward_dists[action_index].sample((1,))[0]
14
15    def simulate_step(self, agent):
16        action_index = agent.select_action()
17        reward = self.get_reward(action_index)
18        agent.update(action_index, reward)
19        return action_index, reward
20
21    def simulate(self, agent, n_steps, verbosity=1):
22        cum_reward = 0
23        for i in range(n_steps):
24            action_index, reward = self.simulate_step(agent)
25            cum_reward += reward
26            if verbosity >= 1:
27                print(
```

Testing the Environment

Let's create an environment and see how it works:

```
1 torch.manual_seed(0)
2 env = BernoulliBanditEnvironment(3)
3 print('Reward distributions')
4 print(env.reward_dists)
5 print('\nRandom rewards for first distribution')
6 print([env.get_reward(0) for i in range(20)])
```

Reward distributions

```
[Bernoulli(probs: 0.6865779757499695), Bernoulli(probs: 0.20974847674369812), Bernoulli(probs: 0.16393446922302246)]
```

Random rewards for first distribution

```
[tensor(1.), tensor(1.), tensor(0.), tensor(1.), tensor(1.), tensor(1.), tensor(1.), tensor(0.), tensor(1.), tensor(1.),
tensor(1.), tensor(1.), tensor(1.), tensor(0.), tensor(0.), tensor(0.), tensor(1.), tensor(1.), tensor(1.), tensor(1.)]
```

Notice how the rewards are stochastic (random) but follow the underlying probability.



Agent 1: Interactive Agent (i.e., you!)

```
1 class InteractiveAgent:
2     def __init__(self, n_actions):
3         self.n_actions = n_actions
4
5     def select_action(self):
6         action_index = -1
7         while action_index < 0:
8             print(f'Enter action index from 0-{{self.n_actions-1}}')
9             try:
10                action_index = int(input())
11            except Exception:
12                action_index = -1
13            if action_index >= 0 and action_index < self.n_actions:
14                break
15            else:
16                print('Incorrect input, try again.')
17                action_index = -1
18        return torch.tensor(action_index)
19
20    def update(self, action_index, reward):
21        return self
22
23    torch.manual_seed(2)
24    env = BernoulliBanditEnvironment(2)
25    interactive_agent = InteractiveAgent(len(env.reward_dists))
26    # Uncomment to be interactive
27    #env.simulate(interactive_agent, n_steps=10)
```

Multi-armed bandits are a simplification of RL yet they retain core RL-specific ideas

- The environment only has a **single state**.
 - “Observing” the environment state is not necessary since it’s always the same.
- The environment **does not change** (in the vanilla bandit problem).
 - The **distribution of rewards** does not change over time *or* due to actions.
 - For example, the payout probabilities for each slot machine are fixed.
- At every timestep, the agent can choose **any action**.
- The only problem is **lack of knowledge**.
 - If we knew which machine gave the highest average payout, we would just take that optimal action again and again.
 - If it was supervised learning, only one example of the “correct” action would be enough!
 - The **explore-exploit tradeoff** still exists because of **uncertainty**.



Multi-armed bandits are a simplification of RL yet they retain core RL-specific ideas

- Bandits isolate the unique feature of RL regarding “**feedback**”.
- **Instructive feedback** provides the correct action no matter which action was already taken (e.g., supervised learning).
 - The optimal action a_t^* (equivalently, ground truth label y^*) is the “feedback” given to a supervised learning system **regardless of the actual action** a_t (equivalently, system’s prediction \hat{y}).
- **Evaluative feedback** provides a reward depending on the action actually taken.
 - The reward signal is a function of the action actually taken a_t , i.e., $R(a_t)$.
 - Thus, the environment **evaluates the actual *action/decision*** made.



How do we design a **policy** that maximizes the sum of rewards?

- We could just do a completely random policy that randomly chooses an action at every time step

$$A_t \sim \text{Uniform}(\{1, 2, \dots, K\})$$

- This is good because it is **simple** and achieves an average reward over all choices.
 - It chooses good and bad actions evenly.
 - It completely ignores the past (i.e., ignores its experience).
- However, it will often take an action that gives **suboptimal reward**.



A better approach is to estimate the value of each action to determine optimal actions

- First, we will define the **value** of an action as the expected reward given this action:

$$q_*(a) \triangleq \mathbb{E}[R_t | A_t = a]$$

- R_t represents the reward random variable at time t .
- A_t represents the action random variable at time t .
- a represents a specific action.
- If we knew the q_* , then the problem would be trivial, just repeatedly take $A_t = a^* = \arg \max_a q_*(a)$
- Obviously, we do not know q_* but we can **approximate it** given our previous actions:

$$Q_t(a) \approx q_*(a)$$



A sample average can be used to estimate the expectation

- We can estimate q_* by using a sample average over the past actions and rewards:

$$Q_t(a) \triangleq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{I}(A_i = a)}{\sum_{i=1}^{t-1} \mathbb{I}(A_i = a)}$$

- As an example, suppose the past rewards and actions are:
 - $A = [1, 2, 2, 1, 2, 2, 2]$
 - $R = [0, 1, 1, 1, 0, 1, 1]$
- If $t = 6$, then $Q_6(1) = 1/2$, $Q_6(2) = 2/3$.
- What would it be for $t = 3$?



Given an estimate of the action value $Q_t(a)$, how could we use this information?

- The **greedy action** optimizes the action value approximation $Q_t(a)$

$$A_t = \arg \max_a Q_t(a)$$

- This is good because it approximates the optimal action $a^* = \arg \max_a q_*(a)$
 - Thus, it will tend to have better reward than the random policy.

- Greedy algorithm
 - Initialize $\forall a, Q_1(a) \leftarrow 0, n_a \leftarrow 0$
 - For $t = \{1, 2, \dots, T\}$
 - Choose $A_t \leftarrow \arg \max_a Q_t(a)$
 - Receive reward $R_t \leftarrow \text{Environment}(A_t)$
 - Update $Q_{t+1}(A_t) \leftarrow \frac{Q_t(A_t) \cdot n_{A_t} + R_t}{n_{A_t} + 1}$
 - Update $n_{A_t} \leftarrow n_{A_t} + 1$



Greedy can be suboptimal if Q_t is a bad approximation

- However, greedy can be **bad**, if $Q_t(a)$ is a **bad approximation**

$$\max_a Q_t(a) \neq \max_a q_*(a)$$

- Thus, the core explore-exploit tradeoff remains:
 - **Exploit** – Choose greedy action to maximize rewards.
 - **Explore** – Choose non-greedy action to improve estimate of Q_t .
- Note: The “explore” part is just about improving our understanding about the environment rather than finding new environment states because there is **only one state** in bandits.
- Can we do better than greedy?



ϵ -Greedy algorithm slightly modifies the greedy algorithm to improve exploration

- One simple idea is to randomly sample arms initially and then do greedy from then on
- A more common approach is to randomly choose between explore (via random algorithm) and exploit (via greedy algorithm).
- **ϵ -Greedy algorithm:**
 - Initialize $\forall a, Q_1(a) \leftarrow 0, n_a \leftarrow 0$
 - For $t = \{1, 2, \dots, T\}$
 - With probability ϵ , choose $A_t \leftarrow \text{RandomAction}()$
 - Otherwise, choose $A_t \leftarrow \arg \max_a Q_t(a)$
 - Receive reward $R_t \leftarrow \text{Environment}(A_t)$
 - Update $Q_{t+1}(A_t) \leftarrow \frac{Q_t(A_t) \cdot n_{A_t} + R_t}{n_{A_t} + 1}$
 - Update $n_{A_t} \leftarrow n_{A_t} + 1$



Demo: Oracle Agent (The Cheater!)

Concept: This agent “cheats” because it knows the true probabilities.

- Always selects the optimal action
- Provides an **upper bound** on performance

```
1 class OracleAgent:
2     def __init__(self, env):
3         self.optimal_action = torch.argmax(torch.tensor([
4             dist.probs for dist in env.reward_dists
5         ]))
6     def select_action(self):
7         return self.optimal_action
8     def update(self, action_index, reward):
9         return self
10    def __str__(self):
11        return 'Oracle'
12
13    torch.manual_seed(2) # 2 and 5
14    env = BernoulliBanditEnvironment(2)
15    oracle = OracleAgent(env)
16    env.simulate(oracle, n_steps=10)
```

```
Step=01, Action=1, Reward=0, Cum reward=0, Avg reward=0.00,
Step=02, Action=1, Reward=1, Cum reward=1, Avg reward=0.50,
Step=03, Action=1, Reward=1, Cum reward=2, Avg reward=0.67,
Step=04, Action=1, Reward=0, Cum reward=2, Avg reward=0.50,
Step=05, Action=1, Reward=1, Cum reward=3, Avg reward=0.60,
Step=06, Action=1, Reward=1, Cum reward=4, Avg reward=0.67,
Step=07, Action=1, Reward=1, Cum reward=5, Avg reward=0.71,
Step=08, Action=1, Reward=0, Cum reward=5, Avg reward=0.62,
Step=09, Action=1, Reward=1, Cum reward=6, Avg reward=0.67,
Step=10, Action=1, Reward=1, Cum reward=7, Avg reward=0.70,
tensor(7.)
```



Challenge: Can You Beat the Oracle?

Try playing interactively and see how close you can get to the oracle's performance!

```
1 env = BernoulliBanditEnvironment(2)
2 n_steps = 10
3
4 interactive_agent = InteractiveAgent(len(env.reward_dists))
5 total_reward = 0
6 # Uncomment to be interactive
7 #total_reward = env.simulate(interactive_agent, n_steps=n_steps)
8 oracle_reward = env.simulate(OracleAgent(env), n_steps=10000, verbosity=0)/10000 * n_steps
9 print(f'\nThe probabilities were {[dist.probs.item() for dist in env.reward_dists]}')
10 print(f'\nYour reward was ${total_reward}, the oracle policy would have received an award close to ${oracle_reward:.2f}')
```

The probabilities were [0.7707958817481995, 0.2072622925043106]

Your reward was \$0, the oracle policy would have received an award close to \$7.76



Demo: Random Agent (The Baseline)

- No learning involved
- Provides a **lower bound** on performance

```
1 class RandomAgent:
2     def __init__(self, n_actions):
3         self.n_actions = n_actions
4
5     def select_action(self):
6         return torch.randint(self.n_actions, (1,))[0]
7
8     def update(self, action_index, reward):
9         return self
10
11     def __str__(self):
12         return f'Random'
13
14 torch.manual_seed(0)
15 env = BernoulliBanditEnvironment(3)
16 random_agent = RandomAgent(len(env.reward_dists))
17 env.simulate(random_agent, n_steps=10)
```

```
Step=01, Action=2, Reward=0, Cum reward=0, Avg reward=0.00,
Step=02, Action=1, Reward=1, Cum reward=1, Avg reward=0.50,
Step=03, Action=0, Reward=1, Cum reward=2, Avg reward=0.67,
Step=04, Action=1, Reward=0, Cum reward=2, Avg reward=0.50,
Step=05, Action=2, Reward=0, Cum reward=2, Avg reward=0.40,
Step=06, Action=1, Reward=1, Cum reward=3, Avg reward=0.50,
Step=07, Action=1, Reward=0, Cum reward=3, Avg reward=0.43,
Step=08, Action=2, Reward=0, Cum reward=3, Avg reward=0.38,
Step=09, Action=2, Reward=0, Cum reward=3, Avg reward=0.33,
Step=10, Action=1, Reward=1, Cum reward=4, Avg reward=0.40,
tensor(4.)
```



Greedy Agent: Implementation Details

Note: Initialization of Q matters. Higher initial values \rightarrow more early exploration.

```
1 class GreedyAgent:
2     def __init__(self, n_actions, init_value=0):
3         self.init_value = init_value
4         self.action_counts = torch.zeros(n_actions) #n_t
5         self.action_value_func = init_value * torch.ones(n_action
6
7     def select_action(self):
8         return torch.argmax(self.action_value_func)
9
10    def update(self, action_index, reward):
11        action_count = self.action_counts[action_index]
12        sum_rewards = action_count * self.action_value_func[actio
13        new_avg_reward = (sum_rewards + reward) / (action_count +
14        self.action_counts[action_index] += 1
15        self.action_value_func[action_index] = new_avg_reward
16        return self
17
18    def __str__(self):
19        return f'Greedy(init={self.init_value})'
20
21 torch.manual_seed(0)
22 env = BernoulliBanditEnvironment(3)
23 # Try init_value = 0, 1 or 100
24 greedy = GreedyAgent(len(env.reward_dists), init_value=1)
25 env.simulate(greedy, n_steps=20)
```

```
Step=01, Action=0, Reward=1, Cum reward=1, Avg reward=1.00,
Step=02, Action=0, Reward=1, Cum reward=2, Avg reward=1.00,
Step=03, Action=0, Reward=0, Cum reward=2, Avg reward=0.67,
Step=04, Action=1, Reward=1, Cum reward=3, Avg reward=0.75,
Step=05, Action=1, Reward=1, Cum reward=4, Avg reward=0.80,
Step=06, Action=1, Reward=0, Cum reward=4, Avg reward=0.67,
Step=07, Action=2, Reward=0, Cum reward=4, Avg reward=0.57,
Step=08, Action=0, Reward=0, Cum reward=4, Avg reward=0.50,
Step=09, Action=1, Reward=1, Cum reward=5, Avg reward=0.56,
Step=10, Action=1, Reward=0, Cum reward=5, Avg reward=0.50,
Step=11, Action=1, Reward=1, Cum reward=6, Avg reward=0.55,
Step=12, Action=1, Reward=1, Cum reward=7, Avg reward=0.58,
Step=13, Action=1, Reward=1, Cum reward=8, Avg reward=0.62,
Step=14, Action=1, Reward=0, Cum reward=8, Avg reward=0.57,
Step=15, Action=1, Reward=0, Cum reward=8, Avg reward=0.53,
Step=16, Action=1, Reward=0, Cum reward=8, Avg reward=0.50,
Step=17, Action=1, Reward=0, Cum reward=8, Avg reward=0.47,
Step=18, Action=0, Reward=1, Cum reward=9, Avg reward=0.50,
Step=19, Action=0, Reward=1, Cum reward=10, Avg reward=0.53,
Step=20, Action=0, Reward=1, Cum reward=11, Avg reward=0.55,
tensor(11.)
```

ϵ -Greedy Implementation

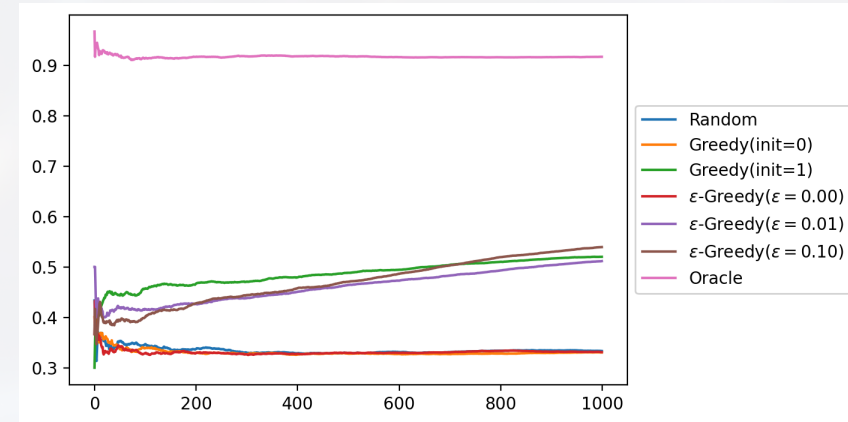
```
1 class EpsilonGreedyAgent(GreedyAgent):
2     def __init__(self, n_actions, epsilon):
3         super().__init__(n_actions, init_value=0)
4         self.epsilon = epsilon
5
6     def select_action(self):
7         if torch.rand((1,))[0] < self.epsilon:
8             return torch.randint(len(self.action_value_func), (1,))
9         else:
10            return torch.argmax(self.action_value_func)
11
12    def __str__(self):
13        return f'$\epsilon$-Greedy($\epsilon$={self.epsilon:.2f})$'
14
15 # Try init_value = 0, 1 or 100
16 torch.manual_seed(0)
17 env = BernoulliBanditEnvironment(3)
18 epsilon_greedy = EpsilonGreedyAgent(len(env.reward_dists), ep
19 env.simulate(epsilon_greedy, n_steps=20)
```

```
Step=01, Action=0, Reward=1, Cum reward=1, Avg reward=1.00,
Step=02, Action=0, Reward=1, Cum reward=2, Avg reward=1.00,
Step=03, Action=0, Reward=1, Cum reward=3, Avg reward=1.00,
Step=04, Action=0, Reward=1, Cum reward=4, Avg reward=1.00,
Step=05, Action=0, Reward=1, Cum reward=5, Avg reward=1.00,
Step=06, Action=1, Reward=0, Cum reward=5, Avg reward=0.83,
Step=07, Action=0, Reward=0, Cum reward=5, Avg reward=0.71,
Step=08, Action=0, Reward=1, Cum reward=6, Avg reward=0.75,
Step=09, Action=0, Reward=1, Cum reward=7, Avg reward=0.78,
Step=10, Action=2, Reward=0, Cum reward=7, Avg reward=0.70,
Step=11, Action=0, Reward=1, Cum reward=8, Avg reward=0.73,
Step=12, Action=0, Reward=0, Cum reward=8, Avg reward=0.67,
Step=13, Action=0, Reward=0, Cum reward=8, Avg reward=0.62,
Step=14, Action=0, Reward=0, Cum reward=8, Avg reward=0.57,
Step=15, Action=0, Reward=1, Cum reward=9, Avg reward=0.60,
Step=16, Action=2, Reward=0, Cum reward=9, Avg reward=0.56,
Step=17, Action=0, Reward=0, Cum reward=9, Avg reward=0.53,
Step=18, Action=0, Reward=1, Cum reward=10, Avg reward=0.56,
Step=19, Action=0, Reward=1, Cum reward=11, Avg reward=0.58,
Step=20, Action=0, Reward=1, Cum reward=12, Avg reward=0.60,
tensor(12.)
```



Comparing All Algorithms

```
1 # Compare algorithms
2 n_actions = 100
3 n_steps = 1000
4 agents = [
5     RandomAgent(n_actions),
6     GreedyAgent(n_actions, init_value=0),
7     GreedyAgent(n_actions, init_value=1),
8     EpsilonGreedyAgent(n_actions, epsilon=0.0),
9     EpsilonGreedyAgent(n_actions, epsilon=0.01),
10    EpsilonGreedyAgent(n_actions, epsilon=0.1),
11 ]
12 label_list = []
13 mean_reward_tensor = []
14 for seed in range(30):
15     mean_reward_list = []
16     torch.manual_seed(seed)
17     env = BernoulliBanditEnvironment(n_actions)
18     #print([dist.probs.item() for dist in env.reward_dists])
19     for agent in agents + [OracleAgent(env)]:
20         torch.manual_seed(seed*1000)
21         mean_reward = []
22         cum_reward = 0
23         for step in range(n_steps):
24             _, reward = env.simulate_step(agent)
25             cum_reward += reward
26             mean_reward.append(cum_reward / (step + 1))
27         mean_reward_list.append(mean_reward)
```



Key Takeaways

What we learned:

- **Random agent:** Worst performance, pure exploration baseline
- **Greedy (init=0):** Can get stuck on suboptimal actions
- **Greedy (init=1):** Optimistic initialization helps early exploration
- **ϵ -Greedy:** Best long-term performance with balanced exploration

The exploration-exploitation dilemma: - Too much exploration \rightarrow waste time on bad actions - Too much exploitation \rightarrow miss better alternatives



Non-stationary / dynamic bandits relax the assumption that the environment is only in one state

- The distribution of rewards changes over time.
 - Though this doesn't necessarily mean that the actions affect the environment.
- The optimal q_* is dependent on time.
- In practice, the estimate Q_t can be updated using a gradient-like rule:

$$Q_{t+1}(A_t) \triangleq Q_t(A_t) + \alpha[R_t - Q_t(A_t)]$$

- This turns out to be a decaying weighted average (i.e., more weight on the most recent rewards):

$$Q_{t+1}(A_t) = (1 - \alpha)^t Q_1(A_t) + \sum_{i=1}^t \alpha(1 - \alpha)^{t-i} R_i$$



Contextual bandits relax the assumption that the agent can observe some clue about the environment state

- Suppose now that the environment changes (nonstationary) **AND** that the agent can observe some clue or **contextual information** about the environment.
- **Examples:**
 - Netflix images – The demographics or previous ratings of the user.
 - Best search result – The search query and user history.
- Now the best action depends on this **context**, or more generally some observation of the **environment state**, denoted S_t .
 - This is the “input” to the action-selection algorithm (like x_i for supervised learning)
- One remaining assumption is that the actions do **NOT** affect the **next state**.
 - Thus, there is still **no notion of planning** in contextual bandits.
 - This is the last remaining assumption to relax to get the full RL problem.



Summary

- Multi-armed bandits are a simplification of RL.
 - Single environment state.
 - Lack of knowledge / uncertainty is the key challenge.
- Bandit problems retain key unique aspects of RL including:
 - **Evaluative feedback** rather than instructive feedback.
 - **Explore-exploit tradeoff**.
- Bandit algorithms
 - Random, Greedy, ϵ -Greedy.
- Variants of bandit problems
 - **Nonstationary bandits** – Environment changes over time.
 - **Contextual bandits** – Agent observes clues/context about the environment state.
 - Both assume that actions do NOT affect future environment states.



Reference

- Based on the excellent RL book by Sutton and Barto:
 - <http://incompleteideas.net/book/the-book-2nd.html>

