[geeksforgeeks] Multi-armed Bandit Problem in Reinforcement Learning

August 4, 2025

1 Problém vícerukého bandity

Zdroj: https://www.geeksforgeeks.org/machine-learning/multi-armed-bandit-problem-in-reinforcement-learning/

Problém mnohorukého bandity (Multi-Armed Bandit - MAB) je klasický problém v teorii pravděpodobnosti, který vystihuje podstatu vyvažování mezi průzkumem (**exploration**) a využitím (**exploitation**). Tento problém je pojmenován podle scénáře, kdy hráč čelí více výherním automatům (banditům) a potřebuje určit, na kterém automatu hrát, aby maximalizoval svou odměnu.

1.1 Pochopení problému vícerukého bandity

1.1.1 Definice problému

V problému vícerukého bandity je agentovi nabídnuto několik možností (rukou - ramen), z nichž každá poskytuje odměnu vyvozenou z neznámého rozdělení pravděpodobnosti. Agent se snaží maximalizovat kumulativní odměnu v průběhu série pokusů. Úkol spočívá ve výběru nejlepšího ramene k provedení testu, ve vyvážení potřeby prozkoumat různá ramena a zjistit jejich rozdělení odměn a využít známá ramena, která poskytla vysoké odměny.

1.1.2 Formální zastoupení

Formálně lze problém MAB popsat následovně:

- **Ramena**: K nezávislých ramen, každé s neznámým rozdělením odměn.
- **Odměny**: Každá i-tá větev poskytuje odměnu R_i, která je dána neznámým rozdělením s očekávanou hodnotou μ_i.
- **Cíl**: Maximalizovat kumulativní odměnu v průběhu T pokusů.

Ústředním dilematem problému MAB je kompromis mezi průzkumem (**exploration**) (vyzkoušením různých odvětví za účelem získání informací o jejich odměnách) a využitím (**exploitation**) (výběrem odvětví, které na základě aktuálních informací poskytlo nejvyšší odměny). Vyvážení těchto dvou aspektů je klíčové pro optimalizaci dlouhodobých odměn.

1.2 Strategie pro řešení problému vícerukého bandity

Pro řešení problému MAB bylo vyvinuto několik strategií. Zde je několik z nejvýznamnějších algoritmů:

1.2.1 1. Epsilon - Chamtivý (Epsilon-Greedy)

Epsilonový algoritmus je jednou z nejjednodušších strategií pro řešení problému MAB. Funguje následovně:

- S pravděpodobností ϵ prozkoumejte náhodné rameno.
- S pravděpodobností $1 - \epsilon$ prozkoumejte rameno s nejvyšší odhadovanou odměnou.

Algoritmus Epsilon-Greedy

1. Inicializujte odhadované hodnoty všech ramen na nulu nebo malé kladné číslo.
2. Pro každý pokus:
   - Vygenerujte náhodné číslo mezi 0 a 1.
   - Pokud je číslo menší než ϵ, vyberte náhodnou skupinu (**exploration**).
   - V opačném případě vyberte skupinu s nejvyšší odhadovanou odměnou (**exploitation**).
   - Aktualizujte odhadovanou odměnu vybrané skupiny na základě pozorované odměny.

Implementace v Pythonu

Implementace demonstruje algoritmus Epsilon-Greedy, což je běžná strategie pro řešení problému Multi-Armed Bandit (MAB). Kód si klade za cíl ilustrovat, jak může agent vyvážit průzkum (**exploration**) a VYUŽITÍ (**exploitation**), aby maximalizoval svou kumulativní odměnu.

1. **Simulace problému vícerukého bandity**: Kód simuluje scénář, ve kterém se agent musí rozhodnout, který z několika výherních automatů (ramen) vytáhne, aby maximalizoval celkovou přijatou odměnu.
2. **Implementujte algoritmus Epsilon-Greedy**: Epsilon-Greedy je jednoduchý, ale efektivní algoritmus, který vyvažuje potřebu prozkoumat nové možnosti (ramena) a využít známé a výnosné možnosti.
3. **Vyhodnocení výkonu**: Implementace sleduje celkovou odměnu nashromážděnou během série pokusů, aby vyhodnotila účinnost strategie Epsilon-Greedy.

[10]:
```
import numpy as np

class EpsilonGreedy:
    def __init__(self, n_arms, epsilon):
        self.n_arms = n_arms
        self.epsilon = epsilon
        self.counts = np.zeros(n_arms)  # Number of times each arm is pulled
        self.values = np.zeros(n_arms)  # Estimated values of each arm

    def select_arm(self):
        if np.random.rand() < self.epsilon:
            return np.random.randint(0, self.n_arms)
        else:
            return np.argmax(self.values)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
```

```
        self.values[chosen_arm] = ((n - 1) / n) * value + (1 / n) * reward
```

Importujeme knihovnu `numpy`, která se používá pro práci s vektory, maticemi a náhodnými čísly.

`class EpsilonGreedy:` Definuje agenta, který bude rozhodovat, kterou páku (rameno) tahat.

`__init__` definuje `n_arms` počet pák, `epsilon` pravděpodobnost, že bude agent prozkoumávat, `counts` kolikrát byla každá páka vybrána (pole nul), `values` odhadovaná průměrná odměna každé páky (pole nul).

`select_arm` definuje jak agent vybírá páku. Vygeneruje náhodné číslo mezi 0 a 1. Pokud je menší než *epsilon*, agent exploruje → vybere náhodnou páku. Jinak exploatuje → vybere tu, která má aktuálně nejvyšší odhadovanou hodnotu (argmax). Pokud epsilon = 0.1, pak v 10 % případů náhodně zkouší, jinak využívá nejlepší známou volbu.

`update` definuje aktualizaci hodnot. `counts` zvýšíme počet tahů pro danou páku. `values[chosen_arm]` spočítáme nový odhad průměrné odměny pro tuto páku. Je to způsob, jak postupně zpřesňovat odhad, aniž by bylo potřeba ukládat všechny odměny z minulosti.

- **Příklad použití**

[11]:
```
n_arms = 10
epsilon = 0.1
n_trials = 1000
rewards = np.random.randn(n_arms, n_trials)
agent = EpsilonGreedy(n_arms, epsilon)
total_reward = 0
```

[12]:
```
for t in range(n_trials):
    arm = agent.select_arm()
    reward = rewards[arm, t]
    agent.update(arm, reward)
    total_reward += reward

print("Total Reward:", total_reward)
```

```
Total Reward: 7.768582192994807
```

`for t in range(n_trials):` Opakuje se `n_trials` (1000) pokusů. Každý pokus simuluje jedno rozhodnutí agenta – tedy výběr jedné páky a obdržení odměny. Proměnná `t` označuje aktuální tah, ale v tomto konkrétním kódu se nepoužívá jako index v čase, ale jako sloupec ve `rewards`.

`arm = agent.select_arm()` Agent se rozhodne, kterou páku si zvolí. S pravděpodobností *epsilon* si vybere náhodně (**exploration**). Jinak si zvolí tu, která má zatím nejvyšší očekávanou hodnotu (**exploitation**).

`reward = rewards[arm, t]` Agent získá simulovanou odměnu z matice `rewards` za tahání za páku arm ve `t`-tém pokusu.

`agent.update(arm, reward)` Agent si aktualizuje své znalosti. Zvýší počet tahů pro tuto páku. Přepočítá odhadovanou průměrnou odměnu této páky.

`total_reward += reward` Přičte odměnu z tohoto tahu do celkové odměny. Funguje jako metrika úspěšnosti strategie. Čím víc agent tahá za správné páky, tím vyšší `total_reward`.

1.2.2 2. Horní hranice spolehlivosti (Upper Confidence Bound - UCB)

Algoritmus UCB je založen na principu optimismu tváří v tvář nejistotě. Vybírá skupinu s nejvyšší horní hranicí spolehlivosti a vyvažuje odhadovanou odměnu a nejistotu odhadu.

Algoritmus 1. Inicializujte počty a hodnoty všech ramen. 2. Pro každý pokus: - Vypočítejte horní hranici spolehlivosti pro každé rameno $UCB_i = \hat{\mu}_i + \sqrt{\frac{2 \ln t}{N}}$, kde $\hat{\mu}_i$ je odhadovaná odměna, t je aktuální pokus a N je počet pokusů o provedení i-té větvičky. - Vyberte rameno s nejvyšší hodnotou UCB. - Aktualizujte odhadovanou odměnu vybraného ramena na základě pozorované odměny.

Implementace v Pythonu

Implementace demonstruje algoritmus horní hranice spolehlivosti (UCB). Vysvětlení cílů a kroků této implementace.

1. **Simulace problému vícerukého bandity**: Kód simuluje scénář, kdy agent čelí více výherním automatům (rameným) a musí se rozhodnout, které rameno vybere, aby maximalizoval odměny.
2. **Aplikace algoritmu horní hranice spolehlivosti (UCB)**: Algoritmus UCB vybírá ramena na základě jejich odhadovaných odměn a nejistoty těchto odhadů s cílem efektivně vyvážit průzkum a využití.
3. **Vyhodnocení výkonu**: Implementace sleduje celkovou odměnu nashromážděnou v průběhu série pokusů, aby vyhodnotila, jak dobře algoritmus UCB dosahuje maximalizace odměny.

[13]:
```
import numpy as np

class UCB:
    def __init__(self, n_arms):
        self.n_arms = n_arms
        self.counts = np.zeros(n_arms)
        self.values = np.zeros(n_arms)
        self.total_counts = 0

    def select_arm(self):
        ucb_values = self.values + np.sqrt(2 * np.log(self.total_counts + 1) /␣
 ↪(self.counts + 1e-5))
        return np.argmax(ucb_values)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        self.total_counts += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / n) * value + (1 / n) * reward
```

Importujeme knihovnu `numpy`, která se používá pro práci s vektory, maticemi a náhodnými čísly.

`class UCP`: Definuje agenta, který bude rozhodovat, kterou páku (rameno) tahat.

`__init__` definuje `n_arms` počet pák, `counts` kolikrát byla každá páka vybrána (pole nul), `values` odhadovaná průměrná odměna každé páky (pole nul), `total_counts` celkový počet tahů napříč všemi pákami.

`select_arm` definuje jak agent vybírá páku. **exploitation** = `self.values` (dosavadní průměrné odměny) + **exploration** = `np.sqrt(2 * np.log(self.total_counts + 1) / (self.counts + 1e-5))`, kde `1e-5` je malá konstanta pro zabránění dělení nulou.

$$\sqrt{\frac{2 \cdot \log(\text{celkový počet pokusů} + 1)}{\text{počet pokusů dané páky} + \text{1e-5}}}$$

`np.argmax(ucb_values)` vrací index páky s nejvyšší "upper confidence bound" – tedy páky, která má buď vysokou odměnu, nebo ještě není dostatečně prozkoumaná.

`update` definuje aktualizaci hodnot. `counts` zvýšíme počet tahů pro danou páku. `values[chosen_arm]` spočítáme nový odhad průměrné odměny pro tuto páku. Je to způsob, jak postupně zpřesňovat odhad, aniž by bylo potřeba ukládat všechny odměny z minulosti.

- **Příklad použití**

[14]:
```
n_arms = 10
n_trials = 1000
rewards = np.random.randn(n_arms, n_trials)
agent = UCB(n_arms)
total_reward = 0
```

[15]:
```
for t in range(n_trials):
    arm = agent.select_arm()
    reward = rewards[arm, t]
    agent.update(arm, reward)
    total_reward += reward

print("Total Reward:", total_reward)
```

```
Total Reward: 15.897621183229866
```

`for t in range(n_trials):` Opakuje se `n_trials` (např. 1000) pokusů. Každý pokus simuluje jedno rozhodnutí agenta – tedy výběr jedné páky a obdržení odměny. Proměnná `t` označuje aktuální tah, v tomto kódu se používá jako sloupec v matici `rewards`.

`arm = agent.select_arm()` Agent zvolí páku podle algoritmu UCB.

`reward = rewards[arm, t]` Agent získá simulovanou odměnu z matice rewards za tahání za páku ve `t`-tém pokusu.

`agent.update(arm, reward)` Agent si aktualizuje své znalosti – zvýší počet výběrů této páky a přepočítá její průměrnou odměnu.

`total_reward += reward` Přičte odměnu z tohoto tahu do celkové odměny. Čím častěji agent volí výhodné páky, tím vyšší bude total_reward.

1.2.3 Thompsonovo vzorkování (Thompson Sampling)

Thompsonův přístup je Bayesovský přístup k problému MAB. Zachovává rozdělení pravděpodobnosti pro odměnu každé skupiny a vybírá skupiny na základě výběrů z těchto rozdělení.

Algoritmus

1. Inicializujte parametry rozdělení odměn (např. beta rozdělení) pro každou skupinu.
2. Pro každý pokus:
   - Vyberte odhad odměny z rozdělení každé skupiny.
   - Vyberte skupinu s nejvyšší vzorkovanou odměnou.
   - Aktualizujte parametry rozdělení vybrané skupiny na základě pozorované odměny.

Implementace v Pythonu

Implementace si klade za cíl demonstrovat algoritmus Thompson Sampling, Bayesovský přístup k řešení problému Multi-Armed Bandit (MAB).

- **Simulace problému vícerukého bandity**: Kód simuluje scénář, kdy agent čelí více výherním automatům (rameným) a musí se rozhodnout, které rameno vybere, aby maximalizoval odměny.
- **Aplikace algoritmu Thompson Sampling**: Thompson Sampling je pravděpodobnostní algoritmus, který vyvažuje průzkum a exploataci vzorkováním z posteriorních rozdělení odměny každého ramene.
- **Vyhodnocení výkonu**: Implementace sleduje celkovou odměnu nashromážděnou v průběhu série pokusů, aby vyhodnotila, jak dobře algoritmus Thompson Sampling funguje při maximalizaci odměny.

[16]:
```python
import numpy as np

class ThompsonSampling:
    def __init__(self, n_arms):
        self.n_arms = n_arms
        self.successes = np.zeros(n_arms)
        self.failures = np.zeros(n_arms)

    def select_arm(self):
        sampled_values = np.random.beta(self.successes + 1, self.failures + 1)
        return np.argmax(sampled_values)

    def update(self, chosen_arm, reward):
        if reward > 0:
            self.successes[chosen_arm] += 1
        else:
            self.failures[chosen_arm] += 1
```

Importujeme knihovnu `numpy`, která se používá pro práci s vektory, maticemi a náhodnými čísly.

`class ThompsonSampling`: Definuje agenta, který bude rozhodovat, kterou páku (rameno) tahat.

`__init__` definuje `n_arms` počet pák, `successes` počet úspěšných odměn pro každou páku (pole nul), `failures` počet neúspěchů (pole nul).

`select_arm` definuje jak agent vybírá páku. `np.random.beta(self.successes + 1, self.failures + 1)` Pro každou páku si agent vytáhne vzorek z Beta rozdělení $Beta(spchy + 1, nespchy + 1)$. Tímto způsobem je pravděpodobnější volba pák s vyšším očekáváním, ale občas se zkouší i ty nejisté.

`update` definuje aktualizaci hodnot. Pokud `reward` > 0, považuje se to za úspěch $\rightarrow$ zvýší se počet úspěchů. Jinak jde o neúspěch $\rightarrow$ zvýší se počet neúspěchů.

- **Příklad použití**

[17]:
```
n_arms = 10
n_trials = 1000
rewards = np.random.randn(n_arms, n_trials)
agent = ThompsonSampling(n_arms)
total_reward = 0
```

[18]:
```
for t in range(n_trials):
    arm = agent.select_arm()
    reward = rewards[arm, t]
    agent.update(arm, reward)
    total_reward += reward

print("Total Reward:", total_reward)
```

```
Total Reward: 7.957880789747213
```

`for t in range(n_trials):` Opakuje se `n_trials` pokusů. Každý simuluje rozhodnutí agenta – výběr páky, získání odměny a aktualizaci znalostí.

`arm = agent.select_arm()` Agent zvolí páku podle algoritmu UCB. Agent pro každou páku vygeneruje náhodný vzorek z rozdělení $Beta(spchy + 1, nespchy + 1)$ a vybere tu páku, která má nejvyšší vzorek.

`reward = rewards[arm, t]` Získá simulovanou odměnu za tahání za zvolenou páku v aktuálním kroku `t`.

`agent.update(arm, reward)` Agent aktualizuje počty úspěchů nebo neúspěchů: Pokud `reward` $>$ 0, přičte úspěch. Jinak přičte neúspěch.

`total_reward += reward` Přičte získanou odměnu do celkového skóre.