Multi-armed Bandits

Facebook: AI powered by Neto-san
YouTube: AibyNeto

ตอนนี้ทุกคนน่าจะเริ่มเข้าใจกันแล้วว่า Reinforcement learning แตกต่างจากการเรียนรู้แบบอื่น ๆ อย่างไร โดย Reinforcement learning ให้ความสำคัญกับการประเมิน ความดี/คุณค่า ของการกระทำ มากกว่าการมาบอกว่า Agent ต้องทำแบบไหนตรง ๆ (อย่างที่ทำใน Supervised learning)

หัวข้อวันนี้จะเป็นพื้นฐานสำคัญสำหรับการสร้าง Agent ที่สามารถเรียนรู้ที่จะตัดสินใน Environment ได้ โดยที่การเรียนรู้จะเกิดขึ้นระหว่างที่ Agent ได้ไปอยู่ใน Environment จริง ๆ เลย แบบนี้เรียกว่า Online learning วันนี้เราขอนำเสนอปัญหา Multi-armed Bandits!!!

Outlines

1. Multi-armed Bandits คืออะไร

Bandit ในที่นี้ให้นึกถึงเครื่อง Slot machine (มีแขนให้เล่นได้หลายอัน) ที่พอเราลงเงินไปปุ๊บ มันก็จะพ่นผลตอบแทน (Reward) ออกมาให้เรา อาจจะมองเป็นชิปซึ่งเอาไปแลกกลับมาเป็นเงินก็ได้ ดังนั้นคำถามที่สำคัญมาก ๆ เลยในที่นี้ก็คือ การหาว่า...แขนไหนของเครื่องที่มันจะให้ผลตอบแทน (โดยเฉลี่ย) ออกมามากที่สุด เรามองว่าการเลือกเล่นแต่ละแขนจะให้รางวัลซึ่งถูกสุ่มจากการแจกแจงความน่าจะเป็นแตกต่างกัน โดยเราจะสนใจที่ค่าเฉลี่ยของรางวัลที่ได้

R_t แทนรางวัล (ไม่แน่นอน, สุ่มจากการแจงแจกความน่าเป็น) ที่ได้รับในเวลา t หลังจากเลือกเล่นแขน a (กระทำ Action a) เราเรียก $q_*(a)$ ว่า Value ของ Action a (สมการที่ (1))

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a] \tag{1}$$

หน้าที่ของเรา คือ พยายามประมาณเทอมนี้ออกมาให้ได้ใกล้เคียงที่สุด เราให้ $Q_t(a)$ แทนค่าประมาณนั้นสำหรับ Action a ค่าประมาณนี้จะอัปเดตไปเรื่อย ๆ ตามเวลา ดังนั้น จึงเป็นฟังก์ชันที่เปลี่ยนแปลงไปตามเวลาด้วย (สมการที่ (2))

$$Q_t(a) = \frac{R_1 + R_2 + \cdots + R_{N_t(a)}}{N_t(a)} \tag{2}$$

พอเรา Track ค่าโดยประมาณของ Value ของ actions ต่าง ๆ ได้แล้ว หลายคนอาจจะเริ่มคิดแล้วว่า ก็ให้ Agent เลือกเล่นแขนที่ให้ $Q_t(a)$ มากที่สุดเลยสิ (สมการที่ (3))

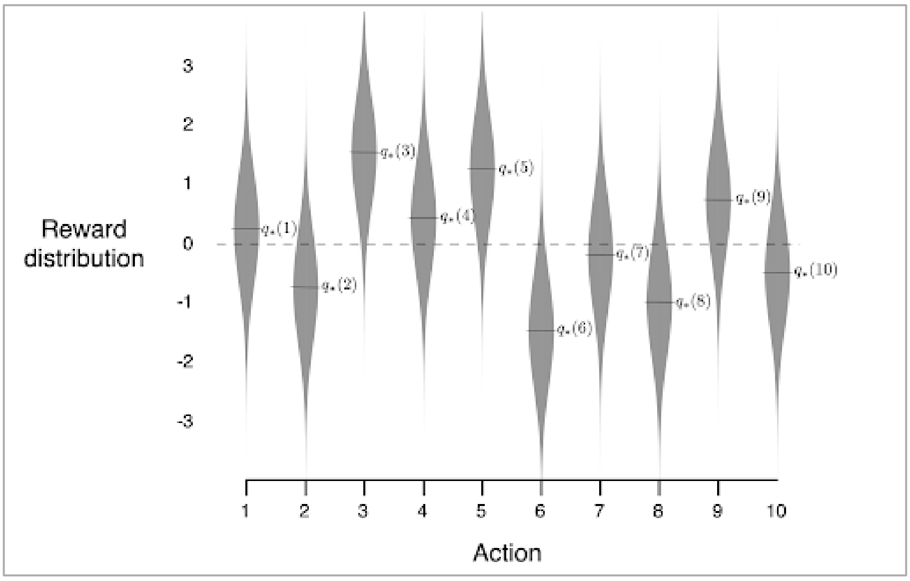
$$A_t = \operatorname*{argmax}_a Q_t(a) \tag{3}$$

แบบนี้เราเรียกว่า **การกอบโกย (Exploitation)** เอาความรู้ที่ Agent มีทำให้เราได้ผลตอบแทนโดยเฉลี่ยสูงที่สุด (Maximize expected reward) แต่ว่าหากเราคิดแบบลึกซึ้งขึ้นนิดหน่อยแล้วเราก็จะพบว่า เราไม่สามารถทำแบบนี้ตลอดเวลาได้ เพราะ เราอาจจะพลาดการเล่นแขนอื่น ๆ หรือขาดค่าประมาณของ Value ของ Action อื่น ๆ ที่หนักแน่นเพียงพอได้

ฉะนั้นแล้ว Agent ไม่ควรที่จะ Exploit ตลอดเวลา แต่ควรจะ Explore ด้วย หรือ ควรจะไปลองเล่นแขนอื่น ๆ ด้วย แม้ว่าแขนที่เพิ่งลองไปล่าสุดโชคดีสุ่มให้ชิป (รางวัล) แก่เรามากก็ตาม **วิธีการแก้ปัญหานี้**ตามธรรมชาติก็คือ การลอง Explore action อื่น ๆ ด้วยสัดส่วนความน่าจะเป็นเล็ก ๆ (ε) ตามแต่ละ time step เราเรียกวิธีแบบนี้ว่า “***ε*-greedy**”

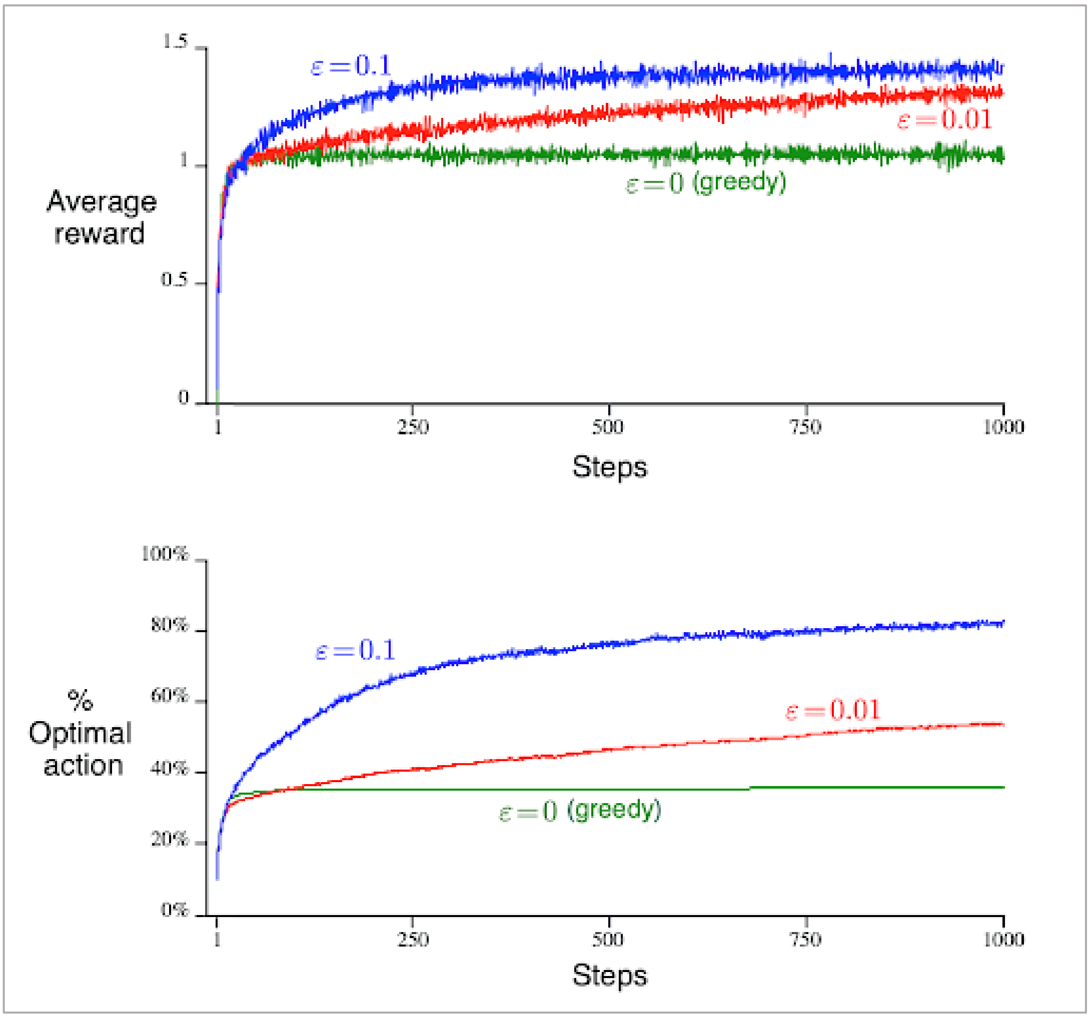
2. ε-greedy + incremental implementation

เพื่อที่จะประเมินความสามารถของ Algorithms เหล่านี้ เราก็จะขอทดสอบมันด้วย 10-armed Testbed หรือกลุ่มของ Simulation ปัญหา Multi-armed Bandits โดยสมมติว่า Reward ที่ได้จากการเล่นแต่ละแขนมีการแจกแจงความน่าจะเป็นแบบ Gaussian ที่ means ของมันเกิดจากการสุ่มจาก Standard Normal N(0, 1) และมี Variance เป็น 1 เราจะสร้าง Testbed ให้มีจำนวนปัญหา Bandits ย่อย ๆ ทั้งหมด 2000 ปัญหา แล้วทำการเฉลี่ยเพื่อวัดผลในตอนหลัง



รูปที่ 1 ตัวอย่างของปัญหาย่อยใน Testbed

หากใครลอง Simulate ตาม Testbed ที่ได้บอกไว้ก็จะพบว่า ***ε*-greedy** ให้ Reward โดยเฉลี่ยสูงกว่าการเลือก Greedy action ในแต่ละ time step อย่างเดียว จริง ๆ แล้วการ Implement สมการที่ (2) ตรง ๆ ลงไปใน Algorithm ของเรานั้น <u>ไม่ใช่วิธีที่ดีนัก</u> เพราะ คอมพิวเตอร์ต้องจำ Rewards ตามเวลาที่ผ่านมาทั้งหมด ทำให้ Memory โตขึ้นแบบเชิงเส้นตามเวลา และต้องเสีย Computational cost ทุกครั้งเพื่อคำนวณตรงตัวเศษของ สมการที่ (2) ใหม่ทุกครั้ง **วิธีที่ดีกว่า** คือ การอัพเดท Q ไปทีละขั้น (Incremental update) โดยจะขึ้นกับ Reward ใหม่ที่เพิ่งเข้ามาเท่านั้น



รูปที่ 2 กราฟเปรียบเทียบ average reward ระหว่าง $\boldsymbol{\varepsilon}$-greedy VS. greedy actions

Incremental Implementation derivation

$$
\begin{aligned}
Q_{k+1} &= \frac{1}{k}\sum_{i=1}^{k} R_i \\
&= \tfrac{1}{k}\left(R_k + \textstyle\sum_{i=1}^{k-1} R_i\right) \\
&= \tfrac{1}{k}\left(R_k + (k-1)Q_k + Q_k - Q_k\right) \\
&= \tfrac{1}{k}\left(R_k + kQ_k - Q_k\right) \\
&= Q_k + \tfrac{1}{k}\left[R_k - Q_k\right]
\end{aligned} \tag{4}
$$

หากเราใช้สมการที่ (4) เพื่อที่จะอัพเดท Q แทนสมการที่ (2) เราก็จะได้ Algorithm อย่างง่ายที่มีประสิทธิภาพมากขึ้นเพื่อแก้ปัญหา Multi-armed Bandit ดังนี้

Initialize, for $a = 1$ to k:

$$Q(a) \leftarrow 0$$

$$N(a) \leftarrow 0$$

Loop forever:

$$A \leftarrow \begin{cases} \text{argmax}_a Q(a) \text{ with probability } 1-\varepsilon \text{ (breaking ties randomly)} \\ \text{a random action with probability } \varepsilon \end{cases}$$

$$R \leftarrow bandit(A)$$

$$N(A) \leftarrow N(A) + 1$$

$$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$$

3. Python code

เพื่อเพิ่มความเข้าใจในเนื้อหา ผมจะลองเอาไป Implement ด้วย Python ดูเหมือนเดิมนะครับ

1. ก่อนอื่นก็ Import libs ต่าง ๆ ที่น่าจะต้องใช้กันก่อน

```
%matplotlib inline
import matplotlib.pyplot as plt

import numpy as np
from tqdm import tqdm
```

2. ในโค้ดเราจะมองปัญหาเป็นเหมือน Instance หนึ่ง ของ Class ก็แล้วกันนะครับผม

```
# Stationary version
N_ACTIONS = 10 # 10-armed
STEPS = 1000
N_EXPS = 2000

class BanditProblem:
    def __init__(self, n_actions):
        self.means = np.random.normal(0, 1, n_actions)
    def get_reward(self, action):
        return np.random.normal(self.means[action], 1, 1)[0]
    def get_best_action(self,):
        return np.argmax(self.means)
```

3. ส่วนนี้จะเป็นส่วนของ Simple bandit algorithm โดยตัวแปร avg_rewards กับตัวแปร percent_optimal_actions ก็จะล้อกับ label แกน y ของกราฟในรูปที่ 2 เลยครับ

```
epsilon = 0.1 # More exploration case
avg_rewards1 = np.zeros(STEPS)
percent_optimal_actions1 = np.zeros(STEPS)

for exp in tqdm(range(N_EXPS)):
    pb = BanditProblem(10)
    optimal_action = pb.get_best_action()

    Q = np.zeros(N_ACTIONS)
    N = np.zeros(N_ACTIONS)
    rewards = []
    count = 0; count_optimal_action = []

    for i in range(STEPS):
        # Calculate A
        if np.random.choice([0, 1], p=[epsilon, 1.0-epsilon]):
            actions = np.argwhere(Q == np.amax(Q))
            if actions.shape[0] > 1:
                A = np.random.choice(np.squeeze(actions))
            else:
                A = actions[0][0]
        else:
            A = np.random.choice(np.arange(0, N_ACTIONS))

        # Calculate R
        R = pb.get_reward(A); rewards.append(R)
        N[A] = N[A] + 1
        Q[A] = Q[A] + (R-Q[A])/N[A]

        if A == optimal_action:
            count += 1
        count_optimal_action.append(100.0*count/(i+1))

    rewards = np.array(rewards)
    avg_rewards1 += rewards

    count_optimal_action = np.array(count_optimal_action)
    percent_optimal_actions1 += count_optimal_action

avg_rewards1 /= N_EXPS
percent_optimal_actions1 /= N_EXPS
```

หากเราลองปรับ parameter epsilon ดูก็จะพบการเปลี่ยนแปลง Average reward โดยเฉลี่ย ไปในทิศทางที่ดีขึ้นตามรูปข้างล่างนี้เลยครับ

สังเกตว่าเราจะได้ผลลัพธ์เหมือนกราฟในรูปที่ 2 ที่ผมยืมมาจากหนังสือ Reinforcement Learning: An Introduction พอดี! จริง ๆ แล้วปัญหา Bandit ที่เรากำลัง Tackle อยู่ตอนนี้อาจจะยังไม่สมจริงมากนัก เพราะมันก็เป็นไปได้ที่ การแจกแจงความน่าจะเป็นของ Reward ที่ได้จากการเล่นแขน a ของ Slot machine อาจเปลี่ยนแปลงไปตามเวลา ทำให้แนวคิดการเฉลี่ย Reward ที่ได้ตามเวลาด้วยน้ำหนักเท่า ๆ กันเป็นเรื่องที่ไม่ควรทำเท่าไหร่นัก

เราเรียกเวอร์ชั่นใหม่ของปัญหาแบบนี้ว่า Nonstationary problem ซึ่งพบได้ทั่วไปใน Reinforcement learning

4. Upper bound confidence 1 (UCB1) theory

- แนวคิด Optimism in the face of uncertainty
    - Construct optimistic guess
- Optimism มาจาก Upper confidence bound
- Upper bound ได้มาจากความพยายามที่จะแก้ปัญหาว่า สำหรับ Action a หนึ่ง ๆ แล้ว Empirical mean $\tilde{Q}(a)$ ที่จะประมาณได้ห่างไกลจาก True mean ของ Action นั้นมากน้อยแค่ไหน
- ถ้าเรารู้ว่า $\left|Q(a) - \tilde{Q}(a)\right| \leq \text{Bound}$

$\rightarrow Q(a) < \tilde{Q}(a) + \text{Bound}$

$\rightarrow$ เลือกแขนที่ทำให้เกิด $\tilde{Q}(a) + \text{ Bound}$ สูงสุดแทน

ควรจะแคบลดลงตามเวลาจนทำให้ $\tilde{Q}(a)$ เข้าใกล้ $Q(a)$ อย่างมีลิมิตในที่สุด

- $\lim_{n\to\infty} UB_n\,(a) = Q(a)$
- $\text{select argmax}_{\,a} UB_n(a)$
- $\lim_{n\to\infty} \text{argmax}_{\,a}\, UB_n\,(a) = a^*$

4.1 Proof by contradiction

- สมมติว่าลู่เข้า Suboptimal arm a แทน หลังจาก infinitely many trials
- ได้ว่า $Q(a) = U\beta_\infty(a) \geq UB_\infty(a') = Q(a')\forall a'$
- แต่ $Q(a) \geq Q(a')\ \forall a'$ ดังนั้น Contradiction!

Goal หา Bound ซึ่ง $P(\left|Q(a) - \tilde{Q}(a)\right| > \text{Bound}) \leq$ ฟังก์ชั่นค่าน้อย ๆ ที่ลู่เข้าสู่ 0 ได้

4.2 Theorem (Hoeffding's inequality)

ให้ $X_1, X_2, X_3, \dots, X_N$ เป็น independent random variables

มีค่าในช่วง $[\mathrm{a}, \mathrm{b}]$ เราเรียก $\mu = \frac{1}{N}\sum_i \mu_i$

$$X = \frac{1}{N}\sum_i X_i$$

เราจะรู้ว่า $\forall t > 0,\ \ P(|X - \mu| > t) \leq 2e^{-2Nt^2/(b-a)^2}$

ลองมอง X_j เป็น payoff ของ Action i จากทุก ๆ Round ที่กระทำ Action i จะได้ว่า

$$P(\left|Q(i) - \tilde{Q}(i)\right| > t) \leq 2e^{-2n_i t^2}$$

ฉลาดเลือกแทน $t = \sqrt{\frac{2logT}{n_i}}$

จะได้ $P(\left|Q(i) - \tilde{Q}(i)\right| > t) \leq 2T^{-4}$

$$P(Q(i) > \tilde{Q}(i) + t) \leq T^{-4}$$

UCB(h)

$V \leftarrow 0,\ n \leftarrow 0,\ n_a \leftarrow 0\ \forall a$

Repeat until $n = h$

$\quad$ Execute $\text{argmax}_a \tilde{R}(a) + \sqrt{\frac{2 \log n}{n_a}}$

$\quad$ Receive r

$\quad$ $V \leftarrow V + r$

$\quad$ $\tilde{R}(a) \leftarrow \frac{n_a \tilde{R}(a) + r}{n_a + 1}$

$\quad$ $n \leftarrow n + 1, n_a \leftarrow n_a + 1$

Return V

References

(1) Ng, A. *Part XIII Reinforcement Learning and Control*. [PDF file]. Retrieved from http://cs229.stanford.edu/notes/cs229-notes12.pdf

(2) Sutton, R. S., & Barto, A. (2018). *Reinforcement learning: An introduction. Cambridge*, MA: The MIT Press.

(3) Russell, Stuart J. (Stuart Jonathan). (2010). *Artificial intelligence: a modern approach. Upper Saddle River*, N.J.: Prentice Hall.

(4) Poupart, P. *CS885 Reinforcement Learning Lecture 8a: May 25, 2018.* [PDF file]. Retrieved from https://cs.uwaterloo.ca/~ppoupart/teaching/cs885-spring18/slides/cs885-lecture8a.pdf