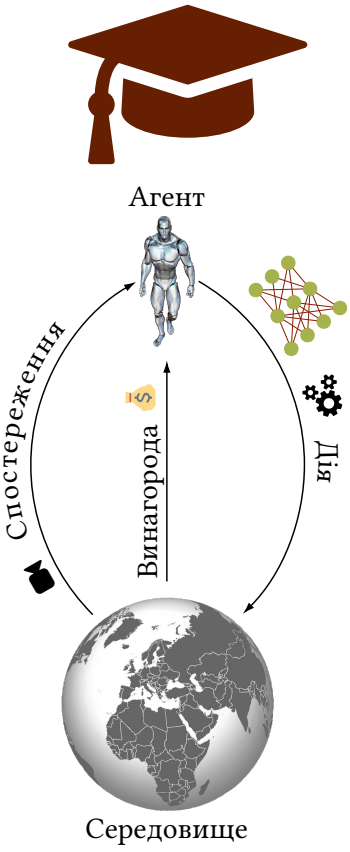
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

Навчання з підкріпленням

Методичні вказівки для виконання практичних робіт | Осінній семестр



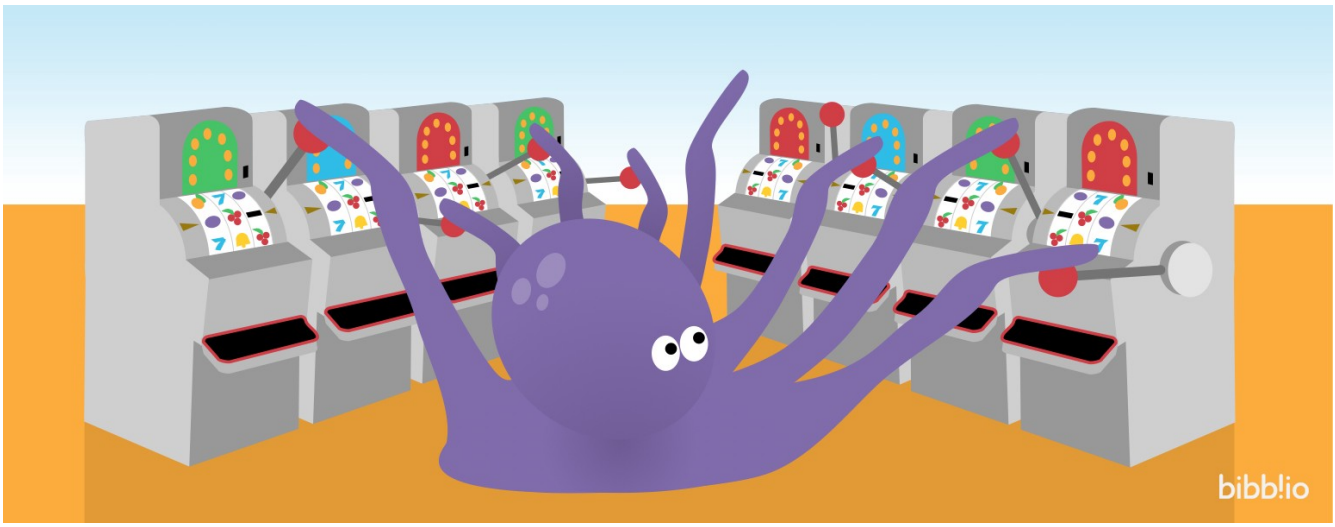
N-рукий бандит

"Efficiency is doing things right; effectiveness is doing the right things."
"Продуктивність – робити речі правильно; ефективність – робити правильні речі."

– Пітер Друкер

Опис завдання

Давайте розглянемо наступну ситуацію. Скажімо, Ви опинились у казино, де перед вами знаходиться 8 ігрових автоматів з кричущим надписом: "Грай безкоштовно! Максимальна виплата становить 10 доларів!" Вау, непогано! Ви заінтриговано запитуєте одну зі співробітниць, що відбувається, тому що це здається занадто дивним, щоб бути правдою, і вона каже: "Це справді правда, грайте безкоштовно скільки завгодно. Кожен ігровий автомат гарантовано дасть Вам винагороду від 0 до 10 доларів. Але слід пам'ятати, що кожен із цих 8 ігрових автоматів має різну середню виплату, тому спробуйте з'ясувати, який із них дає у середньому найбільшу винагороду і ви отримаєте багато грошей!"



Джерело зображення: Multi-Armed Bandits are the New A/B Tests

Що це за казино? Давайте просто з'ясуємо, як можна отримати якнайбільше грошей! До речі, у даному прикладі ігровий автомат називають ще одноруким бандитом! У нього є одна рука (важіль), і він зазвичай краде ваші гроші. Ця ситуація відповідає проблемі 8-рукого бандита, але у загальному випадку можна розглянути задачу N-рукого бандита, де N – кількість ігрових автоматів. Задача багаторукого бандита є прикладом класу задач, які демонструють дилему компромісу між розвідкою[1] (`exploration`) та використанням[2] (`exploitation`) [1].

[1] Вивчення поточної ситуації у середовищі.
[2] Використання раніше набутих знань.

Сформулюємо нашу проблему більш формально. У нас є N можливих дій (для цього випадку $N = 8$), де дія означає: потягнути за ручку (важіль) певного ігрового автомата. Під час кожної гри (k) ми можемо вибрати лише один важіль, щоб це зробити. Після виконання дії a ми отримаємо винагороду R_k – винагорода отримана за гру k. Кожен важіль має унікальний розподіл ймовірностей для виплат грошей (винагород). Наприклад, якщо ми маємо 8 ігрових автоматів і граємо багато разів, то ігровий автомат № 3 може дати середню винагороду скажімо 9 доларів, тоді як ігровий автомат № 1 може дати середню винагороду лише 4 долари. Звичайно, оскільки винагорода за кожну гру є імовірнісною, можливо, що автомат № 1 випадково дасть нам винагороду в розмірі 10 доларів у якійсь грі. Але якщо ми зіграємо багато ігор, ми очікуємо, що у середньому ігровий автомат № 1 буде видавати меншу винагороду, ніж № 3.

Наша стратегія повинна полягати у тому, щоб зіграти кілька разів, вибираючи різні важелі та спостерігати за отриманою винагородою за кожну дію. У наступній грі ми хочемо обрати лише той важіль, який мав найбільшу середню винагороду. Таким чином, нам потрібно визначити середню винагороду за виконання дії a у грі k, беручи до уваги винагороди, які були отримані у попередніх іграх для цієї дії. Позначимо середню винагороду як $Q_k(a)$:

$$Q_k(a) = \frac{R_1 + R_2 + \cdots + R_k}{k_a}$$

Псевдокод:

```
def exp_reward(action, history):
  rewards_for_action = history[action]
  return sum(rewards_for_action) / len(rewards_for_action)
```

Тобто, середня винагорода у грі k за дію a є середнім арифметичним усіх попередніх винагород, які ми отримали, виконуючи дію a. Таким чином, наші попередні дії та спостереження упливають на наші майбутні дії. Функцію $Q_k(a)$ називають функцією цінності (цінність дій), оскільки вона показує нам очікуване значення винагороди для заданої дії. Оскільки ми зазвичай позначаємо цю функцію символом Q, тому її також часто називають Q-функцією.

Розвідка та використання

Коли ми вперше починаємо грати, нам потрібно грати в гру і спостерігати за винагородами, які ми отримуємо для різних автоматів. Ми можемо назвати цю стратегію **розвідкою**, оскільки ми, по суті, випадковим чином вивчаємо результати наших дій. Іншу стратегію, яку ми могли б застосувати – **використання**, яка означає, що ми використовуємо наші поточні знання про те, який автомат, здається, приносить найбільшу винагороду і продовжуємо грати на ньому. Наша загальна стратегія має включати деяку кількість **використань** (вибір найкращого важеля на основі того, що ми знаємо на даний момент) і певну кількість **розвідок** (вибір випадкових важелів, щоб ми могли дізнатися більше). Правильний баланс між використаннями та розвідками буде важливим для максимізації наших винагород.

Як ми можемо розробити алгоритм, який би визначав, який ігровий автомат має найбільшу середню винагороду? Ну, найпростішим алгоритмом було б просто вибрати дію, пов'язану з найвищим значенням Q:

$$a^* = \arg\max_a Q_k(a_i) \qquad \forall a_i \in A_k$$

Псевдокод:

```
def get_best_action(actions, history):
  exp_rewards = [exp_reward(action, history) for action in actions]
  return argmax(exp_rewards)
```

Ми використовуємо нашу вищезгадану функцію $Q_k(a)$ для усіх можливих дій і вибираємо ту дію, яка повертає максимальну середню винагороду. Оскільки $Q_k(a)$ залежить від запису наших попередніх дій та пов'язаних

з ними винагород, цей метод не буде оцінювати ті дії, які ми ще не виконували. Таким чином, якщо ми, скажімо, спочатку спробували важелі № 1 та № 3 і помітили, що важіль № 3 дає нам більшу винагороду, тоді дотримуючись лише цього методу ми ніколи більше не подумаємо спробувати інший важіль, наприклад, № 6, який, насправді міг би давати найвищу середню винагороду. Цей метод, який дозволяє просто вибрати найкращий важіль, називається жадібним (`greedy`) або методом використання.

ε-жадібна стратегія

Відкрити приклад у Colab, який буде розглядатись далі, можна за цим посиланням:

https://github.com/YKochura/rl-kpi/blob/main/homeworks/lab2/N_armed_Bandits.ipynb

Нам потрібно перевірити інші важелі (інші ігрові автомати), щоб знайти дійсно найкращу дію. Для того, щоб це зробити, потрібно просто модифікувати наш попередній алгоритм таким чином, щоб з ймовірністю ε ми обирали дію a випадковим чином, а решту часу з ймовірністю $1-\varepsilon$ ми обирали найкращий важіль на основі попереднього досвіду. Цей метод відомий як ε (епсилон)-жадібна стратегія. У більшості випадків ми будемо грати жадібно ($\varepsilon < 0.5$), але іноді ми ризикуємо і вибираємо випадковий важіль, щоб побачити, що станеться. Результат, звичайно, вплине на наші майбутні жадібні дії.

```
import numpy as np
from scipy import stats
import random
import matplotlib.pyplot as plt

N = 8  # Кількість рук бандита (число ігрових автоматів)
probs = np.random.rand(N) # Приховані ймовірності, пов'язані з кожною рукою
eps = 0.1 # епсилон для епсилон-жадібного вибору дій
```

```
probs
```

```
array([0.84026687, 0.7182308 , 0.52330024, 0.52118617, 0.77030153,
       0.75181956, 0.13122341, 0.18899409])
```

У цьому прикладі ми будемо вирішувати проблему з 8-руким бандитом, тому $N = 8$. Числовий масив `probs` довжиною N, заповнений випадковими числами з плаваючою комою, які можна розглядати як ймовірності. Кожна позиція у масиві `probs` відповідає номеру важеля, тобто є можливою дією. Наприклад, перший елемент у масиві `probs` має позицію індексу 0, тому це дія 0, що відповідає важелю під номером 0. Кожен важіль має свою ймовірність, яка враховує сумарну винагороду, яку можна отримати потягнувши за важіль.

Спосіб, який ми обрали для реалізації розподілу ймовірності винагороди для кожного важелю, такий: кожний важіль матиме ймовірність, наприклад, 0.6, а максимальна винагорода становить 10 доларів. Ми налаштуємо цикл `for`, який буде пробігати до 8, і на кожному кроці він додаватиме 1.25 до винагороди, якщо випадкове число з плаваючою комою буде менше, ніж ймовірність важеля. Таким чином, у першому проході циклом створюється випадковий float (наприклад, 0.4), 0.4 менше 0.6, тому `reward += 1.25`. На наступній ітерації створюється ще один випадковий float (наприклад, 0.55), що також менше за 0.6, тому `reward += 1.25`. Це продовжується до тих пір, поки ми не завершимо 8 ітерацій, а потім повернемо остаточну загальну винагороду, яка може бути у діапазоні від 0 до 10. З ймовірністю важеля 0.6, середня винагорода за виконання цієї дії до нескінченності буде давати 6, але під час будь-якої окремої гри винагорода може бути більшою чи меншою.

```
def get_reward(prob, N=8):
    reward = 0;
    for i in range(N):
        if random.random() < prob:
            reward += 1.25
    return reward
```

```
reward_test = [get_reward(0.6) for _ in range(2000)]
```

```
np.mean(reward_test)
```

```
5.9675
```

Отриманий результат показує, що виконання цього коду 2000 разів з імовірністю 0.6 дійсно дає нам середню винагороду близько 6 (див. гістограму на рисунку 1).

```
plt.figure(figsize=(9,5), dpi=600)
plt.xlabel("Винагорода",fontsize=22)
plt.ylabel("# Спостереження",fontsize=22)
plt.hist(reward_test,bins=7)
```

```
(array([ 13.,  98., 263., 466., 529., 406., 225.]),
 array([ 0.        ,  1.42857143,  2.85714286,  4.28571429,  5.71428571,
         7.14285714,  8.57142857, 10.        ]),
 <BarContainer object of 7 artists>)
<Figure size 5400x3000 with 1 Axes>
```

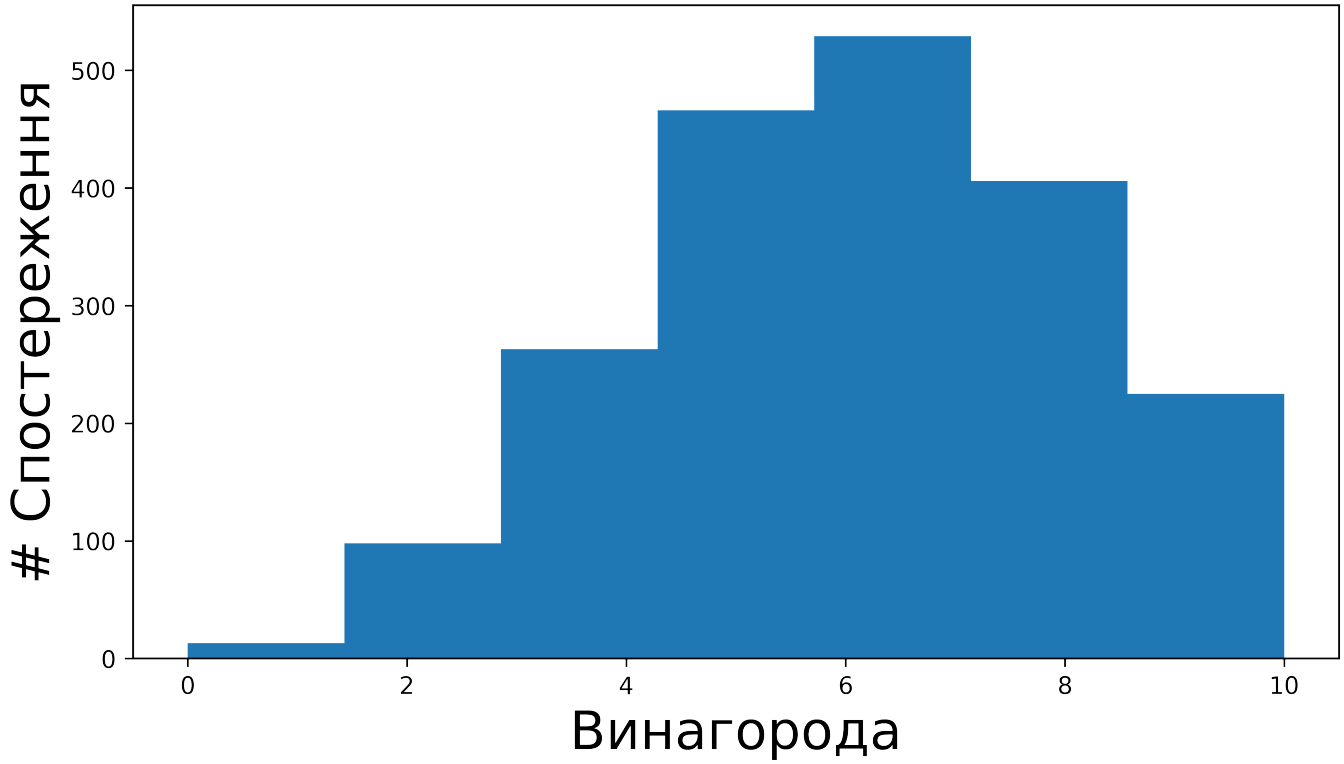


Рис. 1: Розподіл винагород для змодельованого N-рукого бандита з ймовірністю виплати 0.6

Наступна функція, яку ми визначимо – це наша жадібна стратегія вибору найкращого важелю. Нам потрібен спосіб для відстежування важелів, які були потягнуті і яка в результаті була винагорода. Ми могли б просто створити список і додати у цей список кортеж у вигляді спостережень (важіль, винагорода), наприклад, (2, 9), вказуючи, що ми вибрали важіль 2 і отримали винагороду 9. Цей список зростав би у продовж усієї гри.

Однак існує набагато простіший підхід, оскільки нам насправді потрібно відстежувати лише середню винагороду для кожного важелю (руки) – нам не потрібно зберігати кожне спостереження. Нагадаємо, що для обчислення середнього значення для списку чисел x потрібно просто підсумувати усі значення цього списку x_i, а потім розділити отриману суму на кількість елементів у цьому списку:

$$\mu = \frac{1}{k}\sum_i x_i$$

В основному це математичний еквівалент циклу `for`, наприклад:

```
sum = 0
x = [4, 5, 6, 7]
for i in range(len(x)):
    sum = sum + x[i]

mu = sum / len(x)
mu
```

```
5.5
```

Якщо ми вже маємо середню винагороду μ для певного важеля, тоді ми можемо оновити це значення, коли отримаємо нову винагороду в іншій грі, перерахувавши середнє значення. Нам спочатку потрібно скасувати попереднє середнє значення, а потім перерахувати його. Щоб скасувати попереднє середнє значення, ми помножимо μ на загальну кількість до цього зіграних ігор k. Звичайно, це дасть нам лише суму, а не вихідний набір значень – ми не можете скасувати суму. Загальне число зіграних ігор – це те, що нам потрібно знати, щоб перерахувати середнє значення з урахуванням отриманого нового значення винагороди. Ми просто додаємо цю суму до нового значення винагороди x_{k+1} і ділимо на $k+1$ – нова загальна кількість зіграних ігор:

$$\mu_{new} = \frac{k \cdot \mu_{old} + x_{k+1}}{k+1} \tag{1}$$

Ми можемо використовувати це рівняння, щоб постійно оновлювати середню винагороду для кожного важеля, коли ми збираємо нові дані. Таким чином нам потрібно відстежувати лише два числа для кожного важеля: k – кількість отриманих винагород (зіграних ігор) та μ – поточне середнє значення винагороди.

Ми можемо легко зберегти це у масиві `numpy` 8×2 (припускаючи, що у бандита 8 рук):

```
# 8 дій x 2 стовпці
# Стовпці: Кількість зіграних ігор, Середня винагорода
record = np.zeros((N, 2))
record
```

```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])
```

У першому стовпці цього масиву буде зберігатися кількість перетягувань кожної руки автомата, а в другому стовпці зберігатиметься поточна середня винагорода. Давайте напишемо функцію для оновлення цього масиву, коли було виконано нову дію та отримано нову винагороду:

```
def update_record(record, action, r):
    new_r = (record[action, 0] * record[action, 1] + r) / (record[action, 0] + 1)
    record[action, 0] += 1
    record[action, 1] = new_r
    return record
```

Ця функція приймає масив `record`, дію (`action`), яка є просто значенням індексу важеля і нове значення винагороди `r`. Щоб оновити середню винагороду ця функція просто реалізує математичну функцію (1), яку ми описали раніше, а потім збільшує лічильник на одиницю, який зберігає скільки разів цей важіль було використано.

Далі нам потрібно реалізувати функцію, яка вибере, який важіль слід потягнути. Ми хочемо, щоб ця функція обирала важіль, який асоціюється з найвищою середньою винагородою, тому все, що нам потрібно зробити – це знайти рядок у масиві `record` з найбільшим значенням у стовпці 1. Ми можемо це легко зробити, використавши вбудовану функцію `numpy argmax`, яка приймає масив, знаходить найбільше значення в цьому масиві та повертає його позицію у масиві (індекс):

```
def get_best_arm(record):
    arm_index = np.argmax(record[:,1], axis=0)
    return arm_index
```

Тепер ми можемо перейти до основного циклу для гри з N-руким бандитом. Якщо випадкове число буде більше за параметр епсилон (`eps`), ми просто знаходимо найкращу дію за допомогою функції `get_best_arm` і виконуємо її. В іншому випадку ми обираємо випадкову дію, щоб забезпечити певний обсяг розвідки цього середовища. Після вибору важеля ми викликаємо функцію `get_reward`, яка поверне значенням винагороди для цього важеля у поточній грі (нове спостереження). Потім ми оновлюємо масив `record` цим новим спостереження. Ми повторюємо цей процес певну кількість разів, тим самим постійно оновлюючи масив `record`. Важіль з найбільшою ймовірністю винагороди в кінцевому підсумку буде обиратись найчастіше, оскільки цей автомат дасть найвищу середню винагороду (див. нижче масив `record` після 500 зіграних ігор).

Давайте зіграємо 500 разів. Сподіваємось, ми побачимо, що середня винагорода збільшується зі збільшенням кількості зіграних ігор.

```
fig, ax = plt.subplots(figsize=(1,1), dpi=600)
ax.set_xlabel("Кількість зіграних ігор")
ax.set_ylabel("Середня винагорода")
fig.set_size_inches(9, 5)
rewards = [0]
for i in range(500):
    if random.random() > eps:
        choice = get_best_arm(record)
    else:
        choice = np.random.randint(N)
    r = get_reward(probs[choice])
    record = update_record(record, choice, r)
    mean_reward = ((i + 1) * rewards[-1] + r) / (i + 2)
    rewards.append(mean_reward)
ax.scatter(np.arange(len(rewards)), rewards, s=10)
```

```
record
```

```
array([[445.        ,   8.33426966],
       [  4.        ,   6.25      ],
       [  6.        ,   5.41666667],
       [  9.        ,   5.69444444],
       [ 22.        ,   8.23863636],
       [  9.        ,   7.77777778],
       [  1.        ,   1.25      ],
       [  4.        ,   1.25      ]])
```

Як ви можете бачити з рисунку 2, середня винагорода дійсно зростає зі збільшенням кількості зіграних ігор, потім виходить на насичення (знаходження автомату з найбільшою середньою винагородою). Наш алгоритм

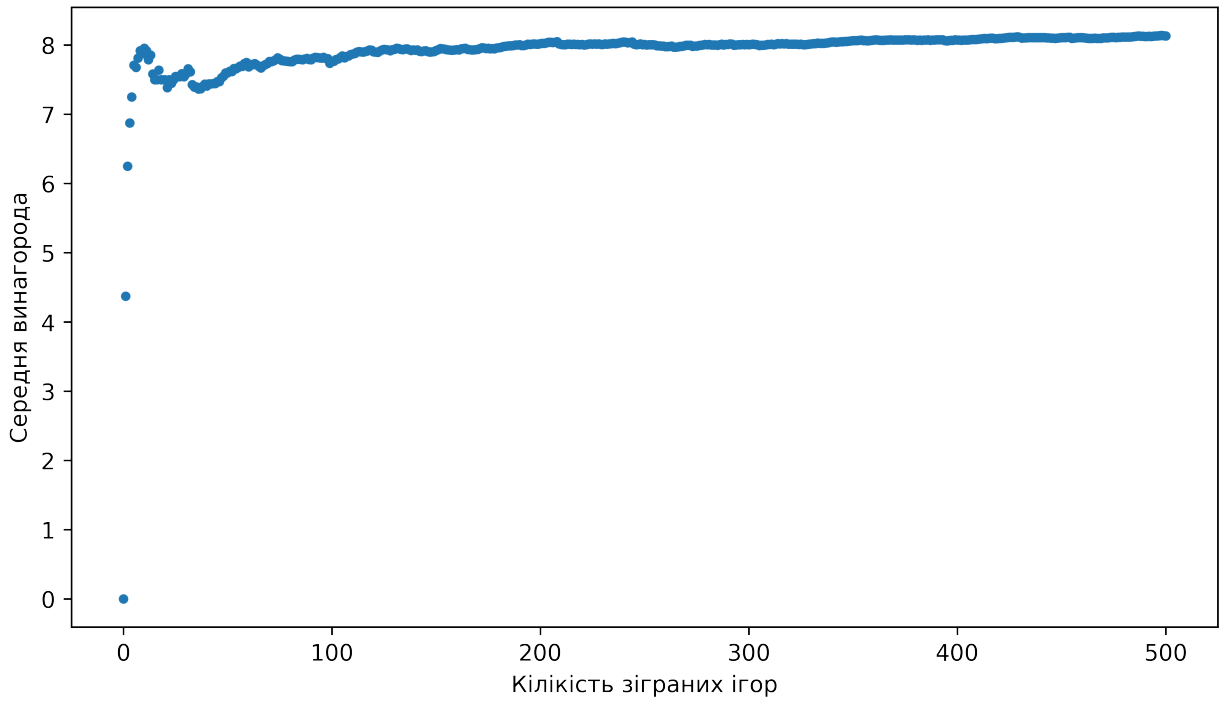


Рис. 2: Цей графік показує, що середня винагорода за кожну гру з часом збільшується. Це вказує на те, що ми успішно вчимося вирішувати проблему N-рукого бандита

навчається, УРА! Процес навчання підкріплюється попередніми добре зіграними іграми! Цей алгоритм, як Ви могли помітити, є досить простим.

Завдання для виконання

Давайте трохи ускладнимо попереднє завдання. Задача, яку ми розглянули вище, є стаціонарною, оскільки основні розподіли ймовірностей винагород для кожного автомату не змінювались з часом. Розгляньте випадок, коли розподіли ймовірностей винагород змінюються – нестаціонарна задача. У цьому випадку Вам потрібно модифікувати `update_record` для знаходження середньозваженого значення винагороди. Крім того, дослідіть для обох випадків: стаціонарної та нестаціонарної задачі про N-рукого бандита, вплив значення ε на швидкість знаходження найкращого автомата, тобто автомата з найбільшою середньою або середньозваженою винагородою. Результати оформити та представити у вигляді звіту.

Оцінювання

Ваша оцінка за виконання завдання буде залежати від:

- 20% – досліджено вплив параметра ε на швидкість знаходження найкращого автомата для стаціонарної та нестаціонарної задачі про N-рукого бандита.
- 40% – програмно реалізовано нестаціонарну задачу про N-рукого бандита.
- 40% – підготовлено звіт у якому подана програмна реалізація нестаціонарної задачі про про N-рукого бандита, а також представлено дослідження впливу параметра ε на швидкість знаходження найкращого автомата для стаціонарної та нестаціонарної задачі. Очікується формальний звіт, написаний в LaTeX. Якщо не бажаєте установлювати додаткове програмне забезпечення, можна скористатися для підготовки звіту www.overleaf.com. Шаблон за яким потрібно підготувати звіт можна звантажити ТУТ.

Структура цього шаблону:

```
...
├── code     # Директорія для файлів з кодом (лістинг коду у звіті)
│   ├── test.py
│   └── hello.c
├── images    # Основна директорія для зображень (.pdf, .jpg, .png)
│   └── NN_.pdf
├── logo      # Директорія для логотипу
│   └── LogoKPI.jpg
├── main.pdf       # Файл отриманий внаслідок компіляції main.tex
├── main.tex       # TeX  файл звіту
├── ref.bib        # Бібліографія (BibTeX)
└── report.cls    # Файл налаштування стилів звіту
```

Здача завдання

Архів `Прізвище Ім'я_Група.zip` відправляйте на перевірку СЮДИ. У архів повинні бути включені:

- блокнот з програмною реалізацією нестаціонарної задачі про N-рукого бандита:

  `Прізвище Ім'я_група_N-armed Bandits.ipynb`

- звіт (.pdf файл) разом з рештою файлів LaTeX, які використовувались для підготовки звіту

Дедлайн: 10 грудня 2023 року о 23:59

Примітка! Виконане завдання після дедлайну оцінюватиметься **не більше** ніж 50% від максимального балу.

Література

[1] L. Weng. (2018) The multi-armed bandit problem and its solutions. [Online]. Available: https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html