دانشکده مهندسی برق

یادگیری تقویتی در کنترل

تمرین اول: شبیه‌سازی مسئله MAB

استاد: دکتر سعید شمقدری

دانشجو: سیده ستاره خسروی

پائیز ۱۴۰۳

چکیده

در این تمرین به پاسخ سوال آخر تمرین سری دوم و بخش کدنویسی تمرین پرداخته می‌شود.

واژه‌های کلیدی: یادگیری تقویتی، راهزن چند دست

فهرست مطالب

فهرست تصاویر و نمودارها

عنوان صفحه

فصل ۱: شبیه‌سازی مسئله راهزن چنددست

۱.۱ مقدمه

سوال هفتم تمرین مرتبط با شبیه‌سازی مسئله راهزن چند دست برای ۱۰ بازو است. در ادامه به حل آن می‌پردازیم.

۱.۲ پیاده سازی GreedyAgent

ابتدا مطابق کد زیر، کتابخانه‌های مورد نیاز را فراخوانی می‌کنیم. کلاس‌هایی نیز لازم است که فراخوانی شوند، کد این کلاس‌ها را از گیتهاب پیدا کرده و در همان مسیری که می‌خواهیم کدنویسی کنیم، قرار می‌دهیم تا فراخوانی آن۲ها با مشکل مواجه نشود.

Part A

```
# Import necessary libraries
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import time

from rlglue.rl_glue import RLGlue
import main_agent
import ten_arm_env
import test_env
```

شکل ۱-۱: فراخوانی کلاس‌ها و کتابخانه‌ها

سپس کد argmax مطابق با صورت تمرین نوشته شده و آن را تست می‌کنیم که پاسخ صحیح برگرداند.

```python
# -----------
# Graded Cell
# -----------
def argmax(q_values):
    """
    Takes in a list of q_values and returns the index of the item
    with the highest value. Breaks ties randomly.
    returns: int - the index of the highest value in q_values

    Because the argmax fuunction returns the first instace of the highest value,
    this is why we need to record all top value index and randomly choice one
    """
    top_value = float("-inf")
    ties = []

    for i in range(len(q_values)):
        # if a value in q_values is greater than the highest value update top and reset ties to zero
        # if a value is equal to top value add the index to ties
        # return a random selection from ties.
        if q_values[i] > top_value:
            top_value = q_values[i]
            ties = []
        if q_values[i] == top_value:
            ties.append(i)

    return np.random.choice(ties)
```

شکل ۲-۱: کد argmax

در ادامه با استفاده از کدهای درون صورت تمرین و اضافه کردن دو print صحت عملکرد argmax را می‌سنجیم، همانطور که در تصاویر زیر قابل مشاهده است، تابع مذکور به درستی کار می‌کند.

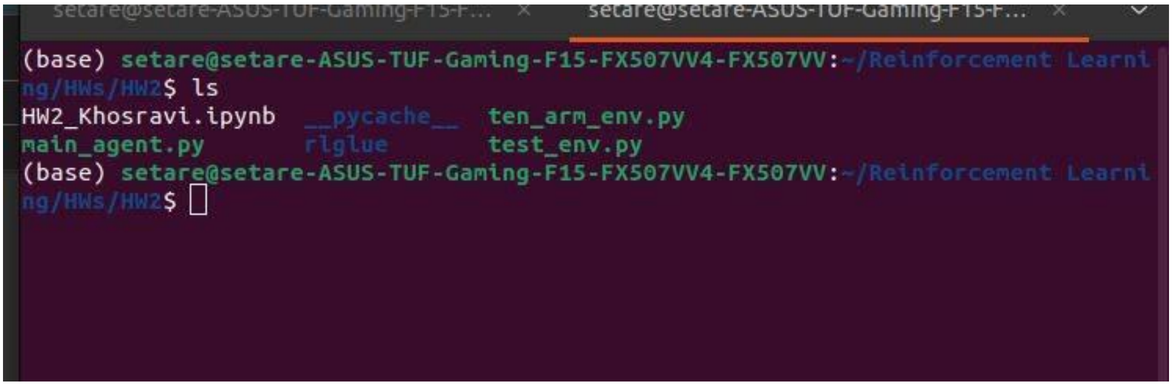
```python
# -------------
# Debugging Cell
# -------------
# Feel free to make any changes to this cell to debug your code

test_array = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
assert argmax(test_array) == 8, "Check your argmax implementation returns the index of the largest value"
print(argmax(test_array))
```
[18]
```
8
```

```python
# make sure np.random.choice is called correctly
np.random.seed(0)
test_array = [1, 0, 0, 1]

assert argmax(test_array) == 0
print(argmax(test_array))
```
[19]
```
3
```

شکل ۳-۱: صحت عملکرد argmax

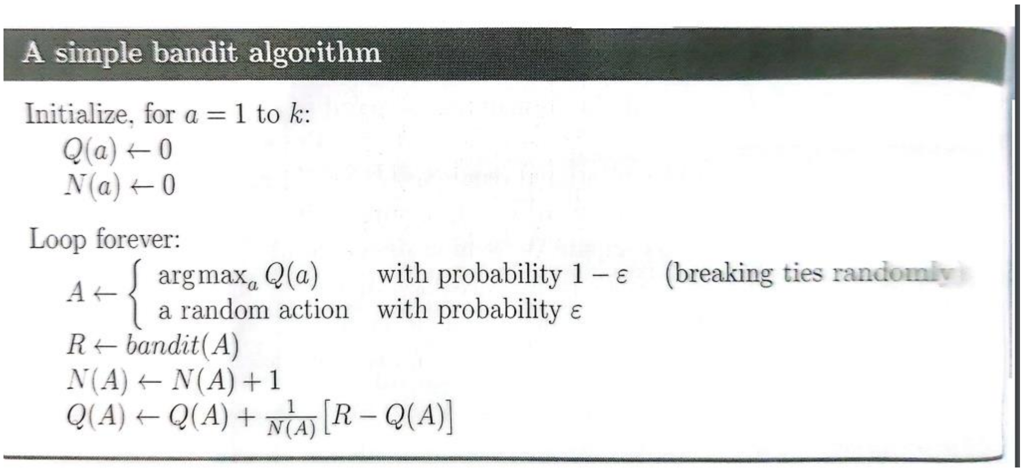
سپس لازم است کلاسی بنویسیم به نام GreedyAgent، بدین منظور به الگوریتم Greedy که در کتاب به آن اشاره شده مراجعه می‌کنیم، ضمن اینکه در نوشتن کلاس باید حواسمان باشد از توابعی استفاده کنیم یا درواقع نام توابع به گونه‌ای باشد که با کلاس‌های فراخوانی شده، مطابقت داشته باشد، در صورت تفاوت ارور پیش می‌آید، که یا باید کلاس‌های از پیش نوشته را تغییر دهیم یا نام توابع را اصلاح کنیم، که ما مورد دوم را انجام دادیم و از همان action_step استفاده کردیم.

برای اینکه مشکلی پیش نیاید، دایرکتوری کد باید به صورت زیر باشد:



شکل ۴-۱: دایرکتوری کد نویسی

الگوریتم Greedy مطابق توضیحات کتاب به صورت زیر است:



A simple bandit algorithm

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

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

$R \leftarrow bandit(A)$

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

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

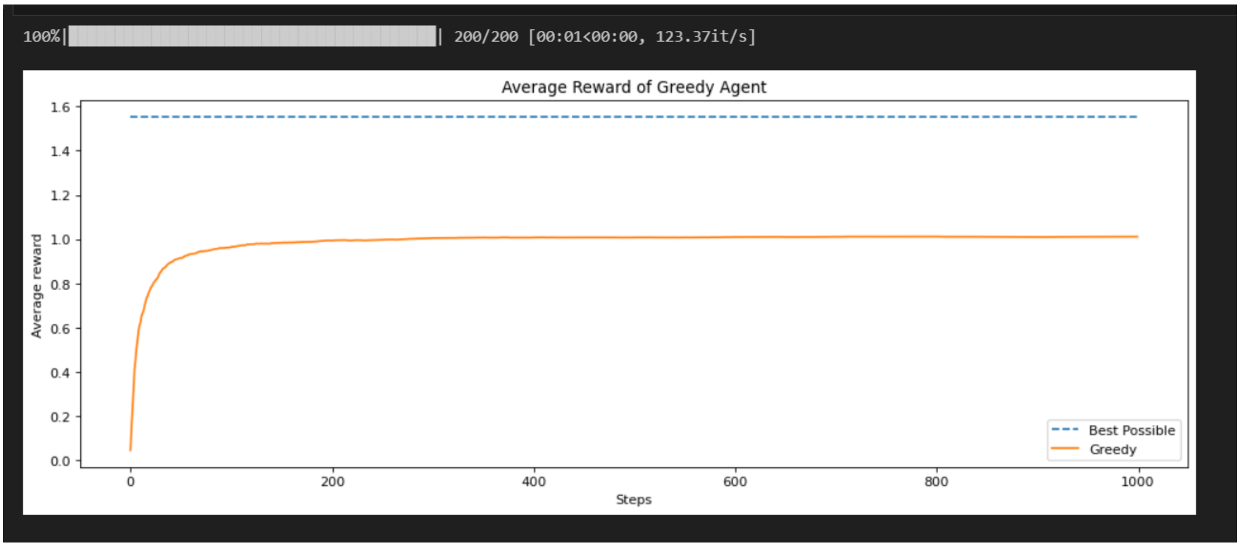
شکل ۵-۱: الگوریتم Greedy

بنابراین ابتدا در کد، مقدار N را بروز می‌کنیم، سپس با توجه به رابطه‌ی موجود در شکل ۵-۱ برای بروز کردن Q، درون کد Q را به روز می‌کنیم، در نهایت براساس خروجی تابع argmax اکشن جدید را انتخاب می‌کنیم، و کلاس ما این اکشن را برمی‌گرداند.

```python
class GreedyAgent(main_agent.Agent):
    def agent_step(self, reward, observation=None):
        # for previous action
        # update N(A)
        self.arm_count[self.last_action] += 1

        # Update Q(A)
        self.q_values[self.last_action] = self.q_values[self.last_action] + (1/self.arm_count[self.last_action])*(reward - self.q_values[self.last_action])

        # select new action
        self.last_action = argmax(self.q_values)

        return self.last_action
[40]
```

شکل ۶-۱: کد کلاس GreedyAgent

با استفاده از کد موجود در صورت تمرین، نمودار را رسم می‌کنیم. توجه: مقدار بهینه با توجه به توضیحات کتاب ۱.۵۵ است.



شکل ۷-۱: خروجی GreedyAgent

همانطور که مشاهده می‌گردد، مطابق انتظار، در حالتی که Agent صرفا حریصانه عمل کند، در کمتر از ۴۰ درصد مواقع عمل بهینه را انتخاب خواهد کرد و به مقدار ارزش بهینه که برابر ۱.۵۵ است نخواهد رسید و در حالت زیر بهینه رفتار خواهد کرد.

۱.۳ پیاده سازی EpsilonGreedyAgent

در این بخش، روش EpsilonGreedy را پیاده کردیم. کد کلاس آن مشابه قبل است، با این تفاوت که، عددی رندوم تولید می‌کنیم، اگر این عدد کمتر از مقدار epsilon باشد، Agent به جست و جو می‌پردازد و اگر بیشتر از epsilon شود، greedy عمل می‌کند. کد نوشته شده، مطابق زیر است:

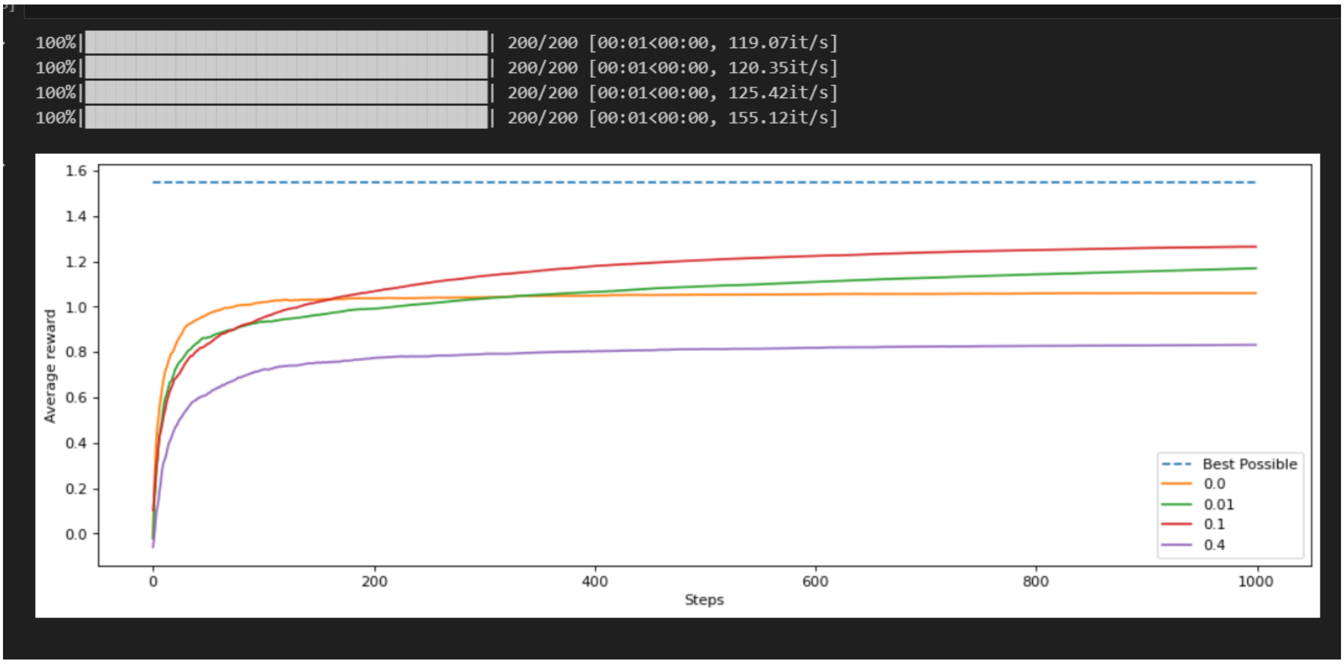
Part B

```
class EpsilonGreedyAgent(main_agent.Agent):
    def agent_step(self, reward, observation=None):
        # for previous action
        # update N(A)
        self.arm_count[self.last_action] += 1

        # Update Q(A)
        self.q_values[self.last_action] = self.q_values[self.last_action] + (1/self.arm_count[self.last_action])*(reward - self.q_values[self.last_action])

        # choose new action
        if np.random.random() < self.epsilon: # np.random.random() is default [0, 1]
            self.last_action = np.random.randint(0, len(self.q_values))
        else:
            self.last_action = argmax(self.q_values)

        return self.last_action
```

[52]

شکل ۸-۱: کد بخش EpsilonGreedyAgent

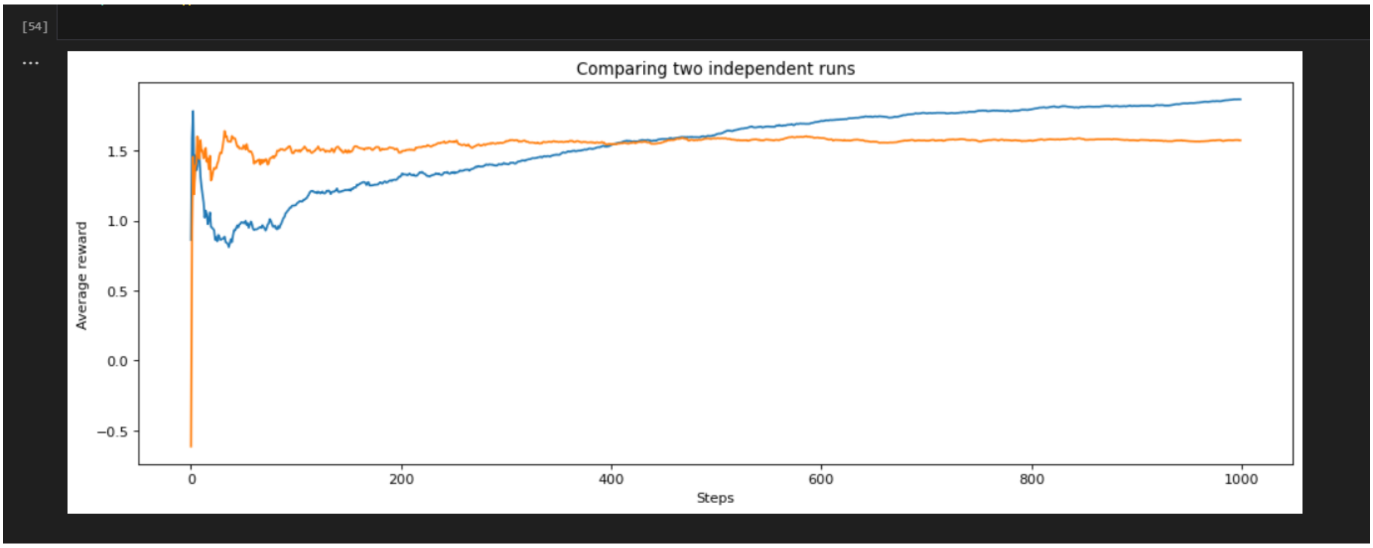
با استفاده از کد موجود در صورت تمرین، نمودار زیر رسم گردید.



شکل ۹-۱: مقایسه‌ی epsilon های مختلف

در این میزان اجرا، همانطور که مشـــاهده می‌گردد، مقدار epsilon برابر با ۰.۱ نســـبت به ســـایرین وضـعیت بهتری دارد، epsilon برابر با صـفر همان عمل صـرفا greedy اسـت که گفتیم حالت زیر بهینه خواهد داشــت. انتظار می‌رود، در طولانی مدت حالت epsilon برابر با ۰.۰۱ از حالت epsilon برابر با ۰.۱ جلو بزند و به مقدار ۱.۵۵ بهینه برسد، چون احتمال انتخاب اکشن بهینه در آن به ۹۹ درصد می‌رسد. از آنجایی که این احتمال برای epsilon برابر با ۰.۱ نیز ۹۰ درصـد اسـت و انتظار می‌رود این نمودار نیز به ۱.۵۵ برسـد، هرچه epsilon افزایش بیابد، نیز ممکن اسـت به حالت زیر بهینه برویم، این موضـوع برای epsilon برابر ۰.۴ نیز مشهود است، در این حالت احتمال جست و جو افزایش یافته است که این موضوع باعث می‌شود، احتمال انتخاب عمل بهینه کمتر از حالات دیگر بشود.

نمودار بعدی نیز در ادامه در شکل ۱۰-۱ موجود است.



شکل ۱۰-۱: مقایسه‌ی دو اجرای متفاوت

تفاوت دو اجرا می‌تواند به دلیل تصادفی بودن محیط باشد، همه چیز به این بستگی دارد که عامل، به صورت تصادفی کدام عمل را برای آغاز انتخاب کند، و چه زمانی به صورت تصادفی شروع به جست و جو بکند، این موارد می‌تواند باعث ایجاد تفاوت بشود. زمانی که محیط نیز پاداش با توزیع احتمال خاص مثلا گاوسی می‌دهد، انتخاب یک عمل مشابه نیز می‌تواند منجر به نتایج متفاوت شود، ممکن است یکبار به ازای یک عمل مشابه پاداش بسیاری دریافت کنیم، یا اینکه پاداش کمتری دریافت شود و همین موضوع فرایند یادگیری را کند کند.

۱.۴ پیاده سازی EpsilonGreedyAgentConstantStepsize

در این قسمت صرفا بجای استفاده از step size متغیر، از خود پارامتر step_size که در کلاس والد وجود داشت استفاده کردیم (با استفاده از ارث بری این ویژگی به کلاسی که نوشتیم منتقل شده است)

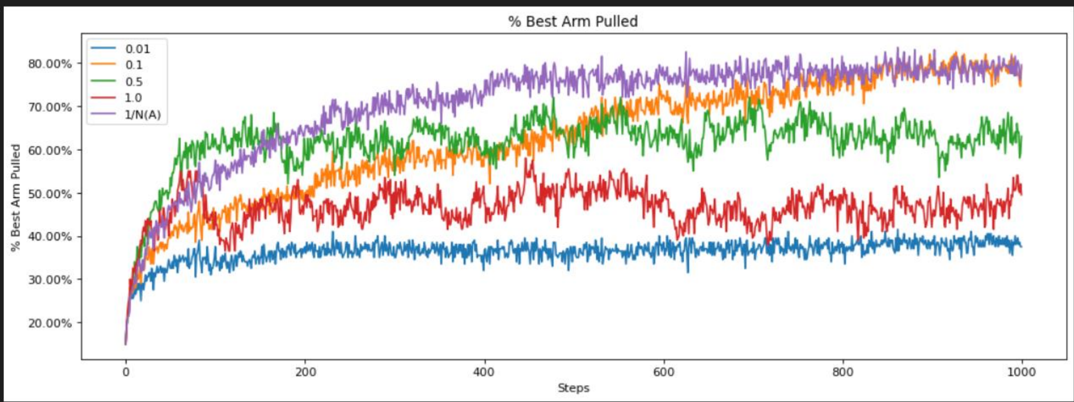
کدی که نوشته شد در شکل ۱۱-۱ موجود است.

```python
class EpsilonGreedyAgentConstantStepsize(main_agent.Agent):
    def agent_step(self, reward, observation=None):
        # for previous action
        # update N(A)
        self.arm_count[self.last_action] += 1

        # Update Q(A)
        self.q_values[self.last_action] = self.q_values[self.last_action] + (self.step_size)*(reward - self.q_values[self.last_action])

        # choose new action
        if np.random.random() < self.epsilon: # np.random.random() is default [0, 1]
            self.last_action = np.random.randint(0, len(self.q_values))
        else:
            self.last_action = argmax(self.q_values)

        return self.last_action
```

شکل ۱۱-۱: کد FixedStepSize

خروجی کد رسم نمودار که در صورت تمرین آمده است نیز به صورت زیر است:

```
100%|██████████████████████████████| 200/200 [00:01<00:00, 125.05it/s]
100%|██████████████████████████████| 200/200 [00:01<00:00, 126.34it/s]
100%|██████████████████████████████| 200/200 [00:01<00:00, 123.62it/s]
100%|██████████████████████████████| 200/200 [00:01<00:00, 116.15it/s]
100%|██████████████████████████████| 200/200 [00:01<00:00, 122.65it/s]
/tmp/ipykernel_9060/2012287133.py:52: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:,.2%}'.format(x) for x in vals])
```



شکل ۱۲-۱: نمودار FixedStepSize

همانطور که مشــاهده می‌گردد، بهترین عملکرد برای حالتی اســت که مشــابه EpsilonGreedy عمل می‌کنیم و اندازه گام ثابت نیســت. اندازه گام متغیر باعث می‌شــود که در ابتدا جســت و جوی بیشــتری داشــته باشـیم، فرصـت برای میانگین‌گیری باشـد و همگرایی نیز براسـاس این گام تضمین می‌شـود، بعلاوه اینکه تاثیر بایاس و شرایط اولیه نیز به زودی از بین می‌رود.

اندازه گام ۰.۱ نیز می‌تواند عملکرد خوبی را از خود نشان بدهد، اما مشکلی که دارد این است که در این حالت یادگیری کندتر می‌شود، درواقع به نوعی این گام نرخ یادگیری است و هرچقدر کمتر باشد، می‌تواند ما را به سمت همگرایی سوق دهد اما روند یادگیری را کندتر می‌کند، با افزایش مقدار گام به ۰.۵ و ۱، ممکن است سیستم در شرایط زیر بهینه گیر بیفتد، در حالتی که گام برابر با ۱ است، مطابق رابطه‌ی موجود برای بروز رسانی ارزش، به طور کلی تخمین با پاداش جدید جایگزین می‌شود که این موضوع مطلوب نیست و باعث نوسان می‌گردد.

در حالتی که اندازه گام را خیلی کوچک انتخاب کنیم، بروز شدن ارزش بسیار کند می‌شود، و زمان زیادی لازم است تا ارزش به صورت قابل توجهی بروز شود و همگرایی بشدت کند می‌گردد.

۱.۵ پیاده سازی UCBAgent

در این قسمت با استفاده از کلاسی که برای GreedyAgent نوشتیم، فقط با اضافه کردن ترم عدم قطعیت، آن را به UCB تبدیل کردیم، با توجه به اینکه رسم نمودار آن با مشکل مواجه گردید، از پیاده سازی دیگری استفاده شد، در این پیاده سازی منطق مطابق کلاس قبلی است که برای UCBAgent و EpsilonGreedyAgent نوشتیم و تفاوتی در کد موجود نیست فقط رسم نمودار این دو در کنار هم به درستی انجام می‌گردد.

کد اولیه به صورت زیر است:

```
Part D

class UCBAgent(main_agent.Agent):
    def __init__(self):
        super().__init__()
        self.total_count = 0  # Total number of steps/actions taken
        self.c = 2  # coefficient of UCB

    def agent_step(self, reward, observation=None):
        # for previous action
        # update N(A)
        self.arm_count[self.last_action] += 1
        self.total_count += 1

        # Update Q(A)
        self.q_values[self.last_action] = self.q_values[self.last_action] + (1/self.arm_count[self.last_action])*(reward - self.q_values[self.last_action])

        # select new action
        self.last_action = argmax(self.q_values + self.c* np.sqrt(np.log(self.total_count) / (self.arm_count + 1e-5)) )

        return self.last_action
[73]
```

شکل ۱۳-۱: کد UCB

پیاده سازی صحیح آن نیز به صورت زیر است:

```python
class Bandit:
    # @k_arm: # of arms
    # @epsilon: probability for exploration in epsilon-greedy algorithm
    # @initial: initial estimation for each action
    # @step_size: constant step size for updating estimations
    # @sample_averages: if True, use sample averages to update estimations instead of constant step size
    # @UCB_param: if not None, use UCB algorithm to select action
    def __init__(self, k_arm=10, epsilon=0., initial=0., step_size=0.1, sample_averages=False, UCB_param=None, true_reward=0.):
        self.k = k_arm
        self.step_size = step_size
        self.sample_averages = sample_averages
        self.indices = np.arange(self.k)
        self.time = 0
        self.UCB_param = UCB_param
        self.average_reward = 0
        self.true_reward = true_reward
        self.epsilon = epsilon
        self.initial = initial

    def reset(self):
        # real reward for each action
        self.q_true = np.random.randn(self.k) + self.true_reward

        # estimation for each action
        self.q_estimation = np.zeros(self.k) + self.initial

        # # of chosen times for each action
        self.action_count = np.zeros(self.k)

        self.best_action = np.argmax(self.q_true)

        self.time = 0
```

شکل ۱۴-۱: بخش اول کد

```python
    # get an action for this bandit
    def act(self):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.indices)

        if self.UCB_param is not None:
            UCB_estimation = self.q_estimation + \
                self.UCB_param * np.sqrt(np.log(self.time + 1) / (self.action_count + 1e-5))
            q_best = np.max(UCB_estimation)
            return np.random.choice(np.where(UCB_estimation == q_best)[0])

        q_best = np.max(self.q_estimation)
        return np.random.choice(np.where(self.q_estimation == q_best)[0])

    # take an action, update estimation for this action
    def step(self, action):
        # generate the reward under N(real reward, 1)
        reward = np.random.randn() + self.q_true[action]
        self.time += 1
        self.action_count[action] += 1
        self.average_reward += (reward - self.average_reward) / self.time

        if self.sample_averages:
            # update estimation using sample averages
            self.q_estimation[action] += (reward - self.q_estimation[action]) / self.action_count[action]
        else:
            # update estimation with constant step size
            self.q_estimation[action] += self.step_size * (reward - self.q_estimation[action])
        return reward
```

شکل ۱۶-۱: بخش دوم کد UCB

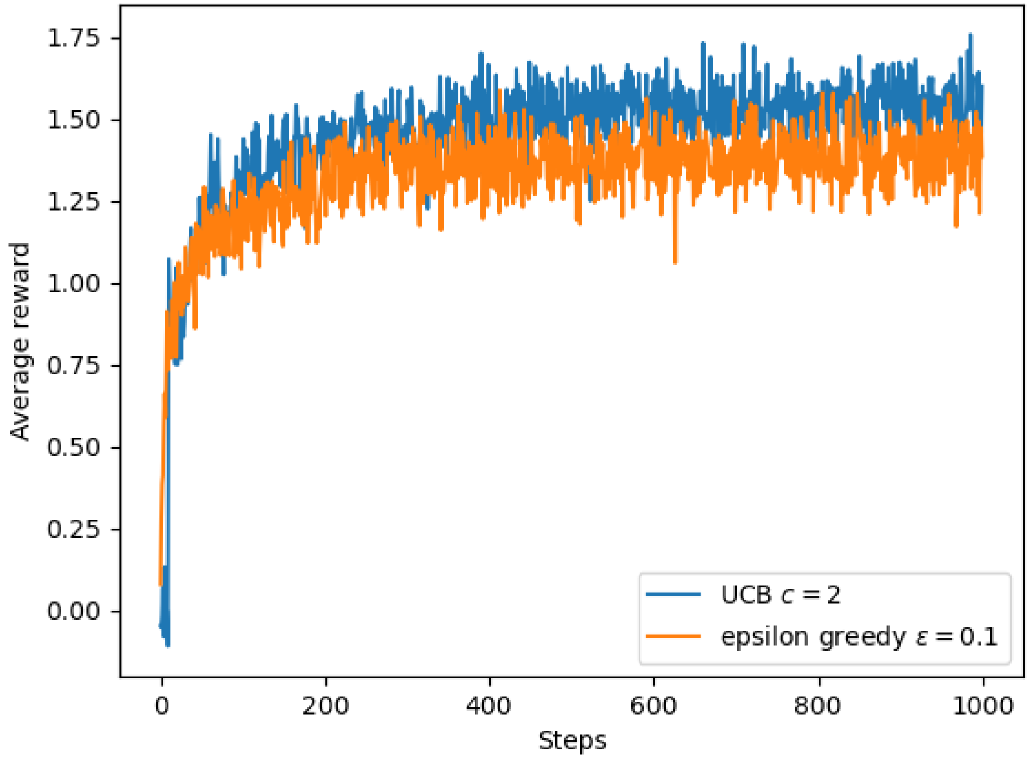
در بخش دوم کد همانطور که می‌بینید، هم بروز رســـانی مطابق الگوریتم موجود است هم به صورت sample average که ما مطابق الگوریتمی که اشـــاره کردیم، رفتار می‌کنیم و sample average را در ابتدا false کردیم. سایر بخش‌های کد نیز مشابه کلاس‌هایی است که نوشتیم.

در نهایت نیز اجرا و رسم نمودار است.

```python
def simulate(runs, time, bandits):
    rewards = np.zeros((len(bandits), runs, time))
    best_action_counts = np.zeros(rewards.shape)
    for i, bandit in enumerate(bandits):
        for r in range(runs):
            bandit.reset()
            for t in range(time):
                action = bandit.act()
                reward = bandit.step(action)
                rewards[i, r, t] = reward
                if action == bandit.best_action:
                    best_action_counts[i, r, t] = 1
    mean_best_action_counts = best_action_counts.mean(axis=1)
    mean_rewards = rewards.mean(axis=1)
    return mean_best_action_counts, mean_rewards


def figure(runs=200, time=1000):
    bandits = []
    bandits.append(Bandit(epsilon=0, UCB_param=2, sample_averages=True))
    bandits.append(Bandit(epsilon=0.1, sample_averages=True))
    _, average_rewards = simulate(runs, time, bandits)

    plt.plot(average_rewards[0], label='UCB $c = 2$')
    plt.plot(average_rewards[1], label='epsilon greedy $\epsilon = 0.1$')
    plt.xlabel('Steps')
    plt.ylabel('Average reward')
    plt.legend()

    plt.savefig('figure.png')
    plt.close()


if __name__ == '__main__':
    figure()
```

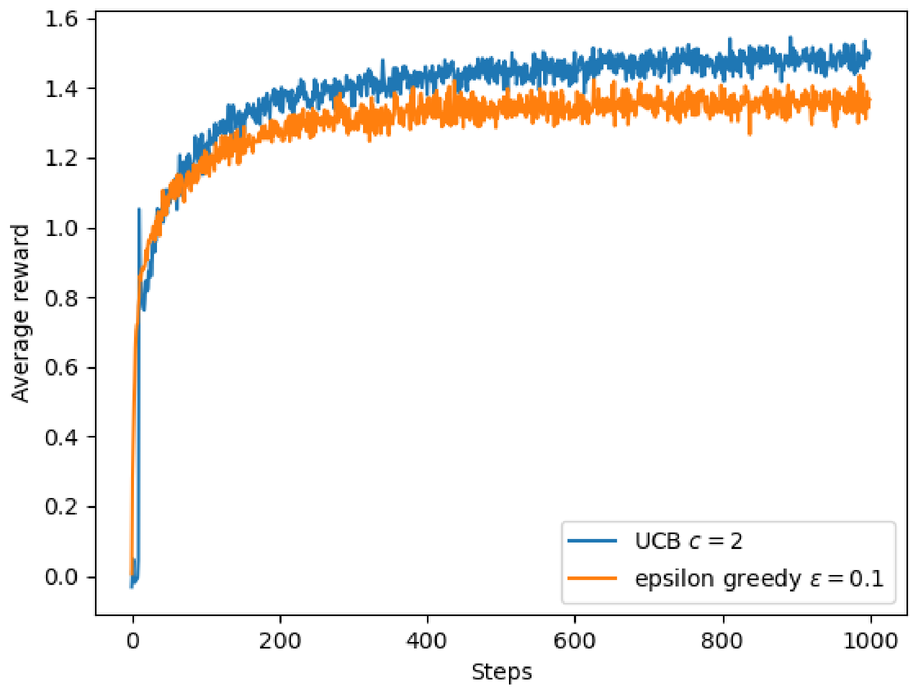
شکل ۱۵-۱: بخش آخر کد UCB

خروجی نیز در تعداد اجرای ۲۰۰ به صورت زیر است:



شکل ۱۷-۱: خروجی UCB و EpsilonGreedy

و پس از ۲۰۰۰ اجرا نیز خروجی در شکل ۱۸-۱ قابل مشاهده است.



شکل ۱۸-۱: خروجی UCB و EpsilonGreedy پس از ۲۰۰۰ اجرا

همانطور که مشـــاهده می‌گردد، عملکرد UCB از EpsilonGreedy بهتر اســـت، خصـــوصـــا زمانی که دفعات اجرا بالاتر می‌رود، UCB به بهترین شکل، مسبت به روش‌های دیگری میان بهره برداری و جست و جو تعادل را برقرار می‌کند، که این موضـــوع به علت افزودن ترم عدم قطعیت می‌باشـــد. در UCB ابتدا، تمامی عمل‌ها انتخاب می‌شوند، در ادامه نیز اعمال با توجه به تعداد دفعاتی که انتخاب شده‌اند و توجه به گذر زمان، شـــانس انتخاب مجدد دارند حتی اگر بهینه نباشـــند، این روش تضـــمین می‌دهد که همه‌ی عمل‌ها حتی اعمال غیر بهینه با گذر زمان انتخاب خواهند شـــد، و در نهایت نیز به حلت بهینه همگرا می‌شـویم، این در حالی اسـت که در EpsilonGreedy بحث عدم قطعیت مطرح نیسـت و جسـت و جو با احتمال یکسـان و کاملا تصـادفی میان اعمال انجام می‌پذیرد. UCB به دلیل انجام اکتشـاف حتی پس از رسـیدن به حالت بهینه، باعث ایجاد عملکرد بهتری می‌شـود، زیرا تضـمین می‌دهد که اطلاعات کافی از تمام اعمال را جمع‌آوری می‌کند. اگر پاداش‌ها نیز تصـــادفی باشـــند، به دلیل اینکه UCB براســـاس عدم قطعیت نیز عمل می‌کند و میان جسـت و جو و انتخاب حریصـانه تعادل خوبی برقرار می‌کند، و جسـت و جو را همچنان ادامه می‌دهد، می‌تواند زمانی که محیط تصـــادفی اســـت نیز بهتر از روش‌های دیگر عمل کند. همانطور که مشـــاهده می‌گردد با افزایش دفعات اجرا نیز تفاوت میان UCB و Epsilon Greedy

مشـــهود تر اســـت، به این علت که UCB با توجه به عملکرد خاص خود و برقراری تعادل میان explore و exploit به نحوی بهتر از ســـایر روش‌ها، در دراز مدت اطلاعات مفیدی از محیط کســـب می‌کند، و عملکردی بسیار نزدیک به حالت بهینه دارد و عامل با این روش می‌تواند بهینه‌تر عمل کند.

رفرنس‌ها:

کتاب Sutton and Barto

ریپازیتوری‌های:

https://github.com/imimali/reinforcement-learning-specialization

https://github.com/setarekhosravi/reinforcement-learning-an-introduction