دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

گزارش تمرین شماره دوم بخش اول

گروه 1

درس یادگیری تعاملی

پاییز 1400

نام و نام خانوادگی	ثمین حیدریان
شماره دانشجویی	21365527

فهرست

چکیده

اهداف این تمرین به شرح زیر است:

- پیاده سازی انواع الگوریتم های مسئله n-armed-bandit مانند ε-greedy، gradient based، ucb.
- بررسی تاثیر مقادیر متفاوت پارامتر های آلفا، بتا و لامبدا در utility function و نتیجه آن در الگوریتم های فوق الذکر.
- مسئله social learning. بهره مندی از رفتارها یا اطلاعاتی نظیر واریانس پاداش های دریافتی دیگر عامل ها برای رسیدن به یک سیاست بهتر در رسیدن به اکشن بهینه.
- پیاده سازی الگوریتم reinforcement comparison، رسم نمودار پشیمانی جهت بررسی الگوریتم و بررسی رابطه پارامتر های آلفا و بتا در این الگوریتم.

سوال 1 - سوال پیاده‌سازی

هدف سوال

هدف:

- پیاده سازی انواع الگوریتم های مسئله n-armed-bandit.
- بررسی تاثیر پارامتر های آلفا، بتا و لامبادا در utility function بر سرعت یادگیری و همگرایی.
- بررسی مقادیر مختلف اپسیلون در الگوریتم ϵ-greedy بر سرعت یادگیری و همگرایی.
- مشاهده اکشن های انتخابی دیگر عامل ها و بهره مندی از آن ها برای یک سیاست مناسب جهت رسیدن به اکشن بهینه میباشد.

قسمت اول

الف)

توضیح پیاده سازی

نکته: توضیحات هر قسمت از کد در زیر آن نوشته شده است.

```
import numpy as np


from amalearn.agent import AgentBase
from amalearn.environment import MutliArmedBanditEnvironment
from amalearn.reward import RewardBase
import matplotlib.pyplot as plt
```

در ابتدا کتابخانه های لازم جهت استفاده و ارث بری ایمپورت میشوند.

```
class Reward(RewardBase):
    def __init__(self, firstParameter, secondParameter, p, alpha, beta, lambd
aa):
        super(Reward, self).__init__()
        self.firstParameter = firstParameter
        self.secondParameter = secondParameter
        self.p = p
        self.alpha = alpha
        self.beta = beta
        self.lambdaa = lambdaa
```

```
    def get_reward(self):
        if type(self.firstParameter) == list:
            r = np.random.uniform()
            if r <= self.p:
                mean = self.firstParameter[0]
                std = self.secondParameter[0]
                reward = np.random.normal(loc=mean, scale=std)
                utility = self.utility_function(reward)
                y = [reward, utility]
                return y
            else:
                low = self.firstParameter[1]
                high = self.secondParameter[1]
                reward  = np.random.uniform(low=low, high=high)
                utility = self.utility_function(reward)
                y = [reward, utility]
                return y

        elif type(self.firstParameter) == int:
            mean = self.firstParameter
            std = self.secondParameter
            reward = np.random.normal(loc=mean, scale=std)
            utility = self.utility_function(reward)
            y = [reward, utility]
            return y

    def utility_function(self, reward):
        if reward >= 0:
            u = np.power(reward, self.alpha)
        else:
            u = (-self.lambdaa) * np.power(-reward, self.beta)

        return u
```

با استفاده از تابع `get_reward` از این کلاس با توجه به پارامتر های دریافتی، از توزیع متناظر با اکشن یک پاداش نمونه برداری میشود. خروجی این تابع یک لیست دوتایی از پاداش دریافتی و `utility` است. اگر مقادیر آلفا، بتا و لامبادا مقادیری غیر از یک داده شوند مقدار پاداش و `utility` متفاوت خواهند شد و در الگوریتم های مورد نظر استفاده متفاوت دارند.

```
def agent_run(agent, env, run, trial):
    mean_reward = np.zeros(trial)
    for run in range(1, run+1):
        for step in range(trial):
            obs, r, d, i = agent.take_action()
            mean_reward[step] = ((run - 1) / run) * mean_reward[step] + ( 1 /
 run) * r
```

```
            env.reset()
            agent.reset()

    return mean_reward
```

در این قسمت عامل به تعداد run های مشخص شده اجرا را از سر خواهد گرفت و به تعداد trial های داده شده یک اکشن را انتخاب میکند و پاداش دریافت میکند. در هر استپ یا ترایال تمام پاداش های دریافتی به تعداد run ها میانگین گرفته خواهد شد. ضمنا بعد از هر run تمام اطلاعات مربوط به عامل و محیط ریست خواهند شد.

```
def plot_mean_reward(mean_reward, lower, upper, label):
    step_no = np.arange(len(mean_reward))
    plt.figure(figsize=(16, 10))
    plt.plot(step_no, mean_reward, label=label)
    plt.xlabel("Steps")
    plt.ylabel("Mean Reward")
    plt.ylim([lower, upper])
    new_list = range(int(np.floor(np.min(lower))), int(np.ceil(np.max(upper))
+1))
    plt.yticks(new_list)
    plt.legend(fontsize='large')
    plt.show();
```

تابع بالا نمودار میانگین پاداش های دریافتی را به تعداد استپ هایی که طی کرده است میکشد.

```
# sudent_id = 21365527
a = 7
b = 5
c = 4
d = 2
firstParameters = [b, a, c, [d, -d]]
secondParameters = [2, 1, 1, [2, 1]]

rewards = [Reward(first, second, 0.7, alpha=1, beta=1, lambdaa=1) for first,
second in zip(firstParameters, secondParameters)]
```

در این قسمت پارامتر های توزیع پاداش های متناظر با هر اکشن مقدار دهی شده اند. سپس به تعداد اکشن های موجود، یک نمونه از کلاس `Reward` ساخته ام تا از توزیع متناظر آن پاداش را نمونه گیری کنم. دلیل اینکه پارامتر a را در عنصر دوم آرایه `firstParameters` قرار داده ام این است که این پارامتر بالاترین میانگین پاداش را دارد و برای بررسی عملکرد و توانایی جستجوی الگوریتم بهتر است این پاداش اولین عنصر از آرایه نباشد.

```
class EpsilonGreedyBanditAgent(AgentBase):
    def __init__(self, id, environment, epsilon):
```

```
        super(EpsilonGreedyBanditAgent, self).__init__(id, environment)
        self.epsilon = epsilon
        self.available_actions = self.environment.available_actions()
        self.rewards_history = [[] for i in range(self.available_actions)]
        self.rewards_mean = np.zeros(self.available_actions)


    def take_action(self) -> (object, float, bool, object):
        my_random = np.random.uniform()

        if(my_random < self.epsilon):
            action = np.random.choice(self.available_actions)
        else:
            action = self.take_best_action()

        obs, r, d, i = self.environment.step(action)
        self.update(r[1], action)

        #print(obs, r, d, i)
        #self.environment.render()
        return obs, r[0], d, i

    def take_best_action(self):
        action = np.argmax(self.rewards_mean)
        return action

    def reset(self):
        self.rewards_history = [[] for i in range(self.available_actions)]
        self.rewards_mean = np.zeros(self.available_actions)


    def update(self, reward, action):
        self.rewards_history[action].append(reward)
        self.rewards_mean[action] = np.mean(self.rewards_history[action])
```

این کلاس عاملی را پیاده سازی میکند که از سیاست ϵ-greedy استفاده میکند [1] . این عامل با احتمال ϵ یک اکشن تصادفی را انجام میدهد و با احتمال $1-\epsilon$ اکشنی را انتخاب میکند که میانگین پاداش های آن تا به اینجا ماکسیمم باشد. تابع update بعد از هر اکشنی که عامل انجام میدهد صدا زده میشود تا میانگین پاداش مربوط به آن به روز شود.

```
env = MutliArmedBanditEnvironment(rewards, 1000, '1')
eps_greedy_gent = EpsilonGreedyBanditAgent('1', env, 0.2)
mean_reward = agent_run(eps_greedy_gent, env, 20, 1000)
plot_mean_reward(mean_reward, -2, 9, 'Epsilon-Greedy')
```

در این قسمت یک محیط از کلاس MutliArmedBanditEnvironment ساخته میشود. سپس یه نمونه از کلاس EpsilonGreedyBanditAgent با مقدار اپسیلون برابر 0.2 ساخته میشود تا در محیط شروع به تعامل کند. تعداد باری که محیط از ابتدا شروع به تعامل در محیط میکند 20 تا میباشد و در هر اجرا 1000 استپ برمیدارد.

```
class UCBBanditAgent(AgentBase):
    def __init__(self, id, environment, confidence_level):
        super(UCBBanditAgent, self).__init__(id, environment)
        self.confidence_level = confidence_level
        self.available_actions = self.environment.available_actions()
        self.rewards_history = [[] for i in range(self.available_actions)]
        self.rewards_mean = np.zeros(self.available_actions)
        self.actions_sum = np.zeros(self.available_actions)


    def take_action(self) -> (object, float, bool, object):
        action = self.take_best_action()
        obs, r, d, i = self.environment.step(action)
        self.update(r[1], action)
        #print(obs, r, d, i)
        #self.environment.render()
        return obs, r[0], d, i

    def take_best_action(self):
        t = self.environment.state['length'] + 1
        exploitation = self.rewards_mean
        exploration = self.confidence_level * np.sqrt(np.log(t) / self.action
s_sum_sanitizer())
        ucb_term = exploitation + exploration
        action = np.argmax(ucb_term)
        return action

    def reset(self):
        self.rewards_history = [[] for i in range(self.available_actions)]
        self.rewards_mean = np.zeros(self.available_actions)
        self.actions_sum = np.zeros(self.available_actions)


    def update(self, reward, action):
        self.rewards_history[action].append(reward)
        self.rewards_mean[action] = np.mean(self.rewards_history[action])
        self.actions_sum[action] += 1

    def actions_sum_sanitizer(self):
        eps = np.finfo(np.float32).eps
        return np.where(self.actions_sum == 0, eps, self.actions_sum)
```

این کلاس عاملی را پیاده سازی میکند که از سیاست UCB1 استفاده میکند [1] . این سیاست طبق معادله زیر عمل خواهد کرد:

$$A_t = argmax_a[Q_t(a) + c\sqrt{\frac{lnt}{N_t(a)}}]$$

به عبارتی اکشنی انتخاب میشود که عبارت جلوی argmax را بیشینه کند. اسم عبارت در کد ucb_term میباشد و از دو قسمت exploration ($c\sqrt{\frac{lnt}{N_t(a)}}$) و exploitation ($Q_t(a)$) تشکیل شده است.

```
env = MutliArmedBanditEnvironment(rewards, 1000, '2')
ucb_agent = UCBBanditAgent('2', env, 2.0)
mean_reward = agent_run(ucb_agent, env, 20, 1000)
plot_mean_reward(mean_reward, -2, 9, 'UCB')
```

در این قسمت یک محیط از کلاس MutliArmedBanditEnvironment ساخته میشود. سپس یه نمونه از کلاس UCBBanditAgent با مقدار C برابر 2 ساخته میشود تا در محیط شروع به تعامل کند. تعداد باری که محیط از ابتدا شروع به تعامل در محیط میکند 20 تا میباشد و در هر اجرا 1000 استپ برمیدارد.

```
class GradientBanditAgent(AgentBase):
    def __init__(self, id, environment, alpha):
        super(GradientBanditAgent, self).__init__(id, environment)
        self.available_actions = self.environment.available_actions()
        self.H = np.zeros(self.available_actions)
        self.rewards = []
        self.reward_mean = 0.0
        self.alpha = alpha
        # self.steps = 0


    def take_action(self) -> (object, float, bool, object):
        probabilities = self.softmax_preferences()
        action = np.random.choice(self.available_actions, p=probabilities)
        obs, r, d, i = self.environment.step(action)
        self.update(r[1], action, probabilities)
        #print(obs, r, d, i)
        #self.environment.render()
        return obs, r[0], d, i


    def softmax_preferences(self):
        probabilities = np.exp(self.H) / np.sum(np.exp(self.H))
        return probabilities


    def reset(self):
```

```
        self.H = np.zeros(self.available_actions)
        self.rewards = []
        self.reward_mean = 0.0

    def update(self, current_reward, taken_action, probabilities):

        # update pereference
        self.H[taken_action] = self.H[taken_action] + self.alpha * (current_r
eward - self.reward_mean) * (1 - probabilities[taken_action])
        not_taken_actions = self.available_actions != taken_action
        self.H[not_taken_actions] = self.H[not_taken_actions] - self.alpha *
(current_reward - self.reward_mean) * (probabilities[not_taken_actions])


        # update reward mean
        self.rewards.append(current_reward)
        self.reward_mean = np.mean(self.rewards)
```

این کلاس عاملی را پیاده سازی میکند که از سیاست gradient استفاده میکند [1] . در این سیاست عامل برای انتخاب اکشن از خروجی softmax آرایه از pereference ها استفاده میکند. این pereference ها در کد با عبارات self.H مشخص شده اند و بعد از هر اکشنی که عامل انجام میدهد در تابع update به روز میشوند. این آپدیت با نرخ یادگیری آلفا انجام میشود.

```
env = MutliArmedBanditEnvironment(rewards, 1000, '3')
gradient_agent = GradientBanditAgent('3', env, 0.1)
mean_reward = agent_run(gradient_agent, env, 20, 1000)
plot_mean_reward(mean_reward, -2, 9, 'Gradient')
```

در این قسمت یک محیط از کلاس MutliArmedBanditEnvironment ساخته میشود. سپس یه نمونه از کلاس GradientBanditAgent با مقدار آلفا برابر 0.1 ساخته میشود تا در محیط شروع به تعامل کند. تعداد باری که محیط از ابتدا شروع به تعامل در محیط میکند 20 تا میباشد و در هر اجرا 1000 استپ برمیدارد.

```
def mean_reward_diff_agent(agentTypes):
    mr_list = []
    labels = []
    for agentType in agentTypes:

        env = MutliArmedBanditEnvironment(rewards, 1000, '4')

        if agentType == 'Epsilon-Greedy':
            eps_greedy_agent = EpsilonGreedyBanditAgent('4', env, 0.2)
            mr = agent_run(eps_greedy_agent, env, 20, 1000)

        elif agentType == 'UCB':
```

```
            ucb_agent = UCBBanditAgent('4', env, 2.0)
            mr = agent_run(ucb_agent, env, 20, 1000)


        elif agentType == 'Gradient':
            gradient_agent = GradientBanditAgent('4', env, 0.1)
            mr = agent_run(gradient_agent, env, 20, 1000)


        mr_list.append(mr)
        labels.append(agentType)

    return mr_list, labels
```

این تابع بسته به نوع سیاستی که عامل دارد با محیط تعامل دارد و در نهایت مقادیر میانگین پاداش های عامل ها مختلف در یک لیست به اسم `mr_list` ذخیر میشود.

```
def plot_mean_reward_diff_agent(mr_list, labels, lower, upper):
    step_no = np.arange(len(mr_list[0]))
    plt.figure(figsize=(17, 12))


    for i, mr in enumerate(mr_list):
        plt.plot(step_no, mr, label=labels[i])

    plt.xlabel("Steps")
    plt.ylabel("Mean Reward")
    plt.ylim([lower, upper])
    new_list = range(int(np.floor(np.min(lower))), int(np.ceil(np.max(upper))
+1))
    plt.yticks(new_list)
    plt.legend(fontsize='large')
    plt.show();
```

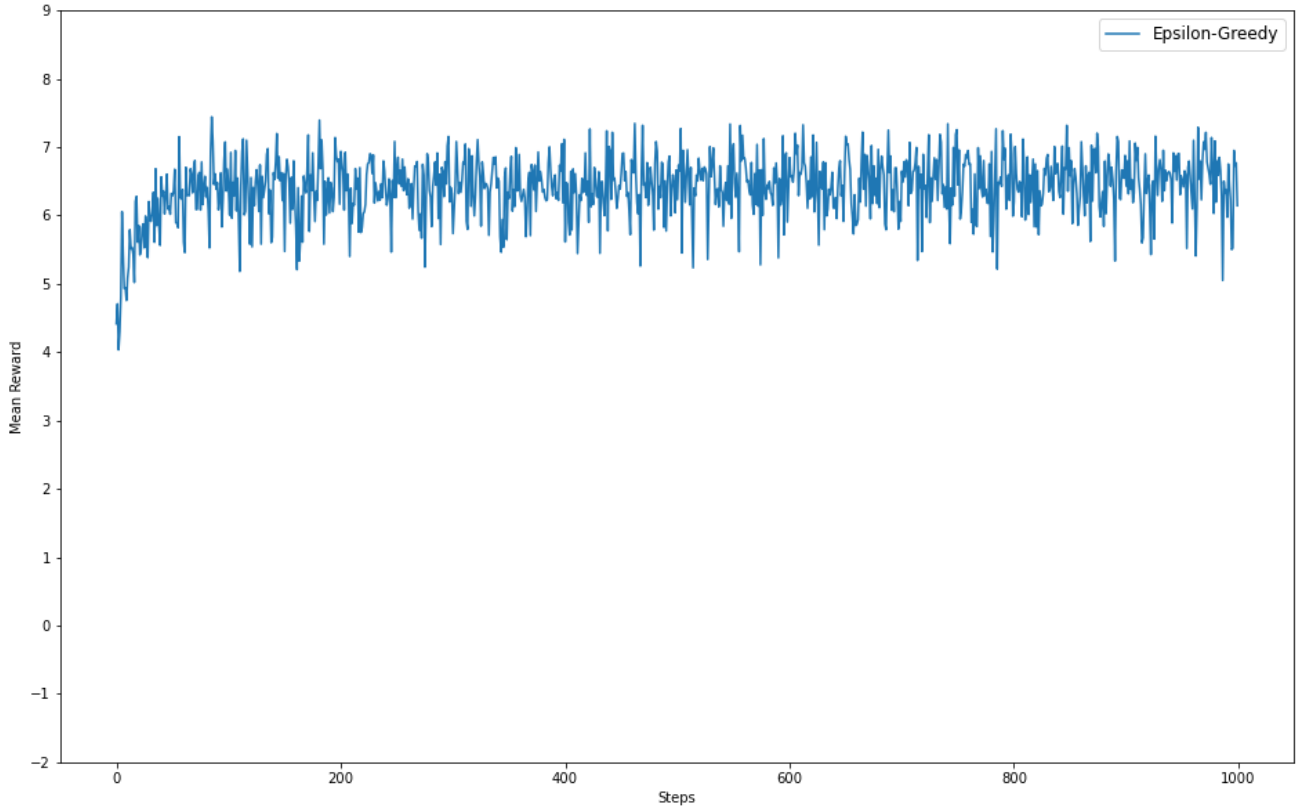
این تابع نمودار میانگین پاداش عامل ها با سیاست مختلف را رسم میکند.

```
agentTypes = ['Epsilon-Greedy', 'UCB', 'Gradient']
mr_list, labels = mean_reward_diff_agent(agentTypes)
plot_mean_reward_diff_agent(mr_list, labels, -3, 11)
```

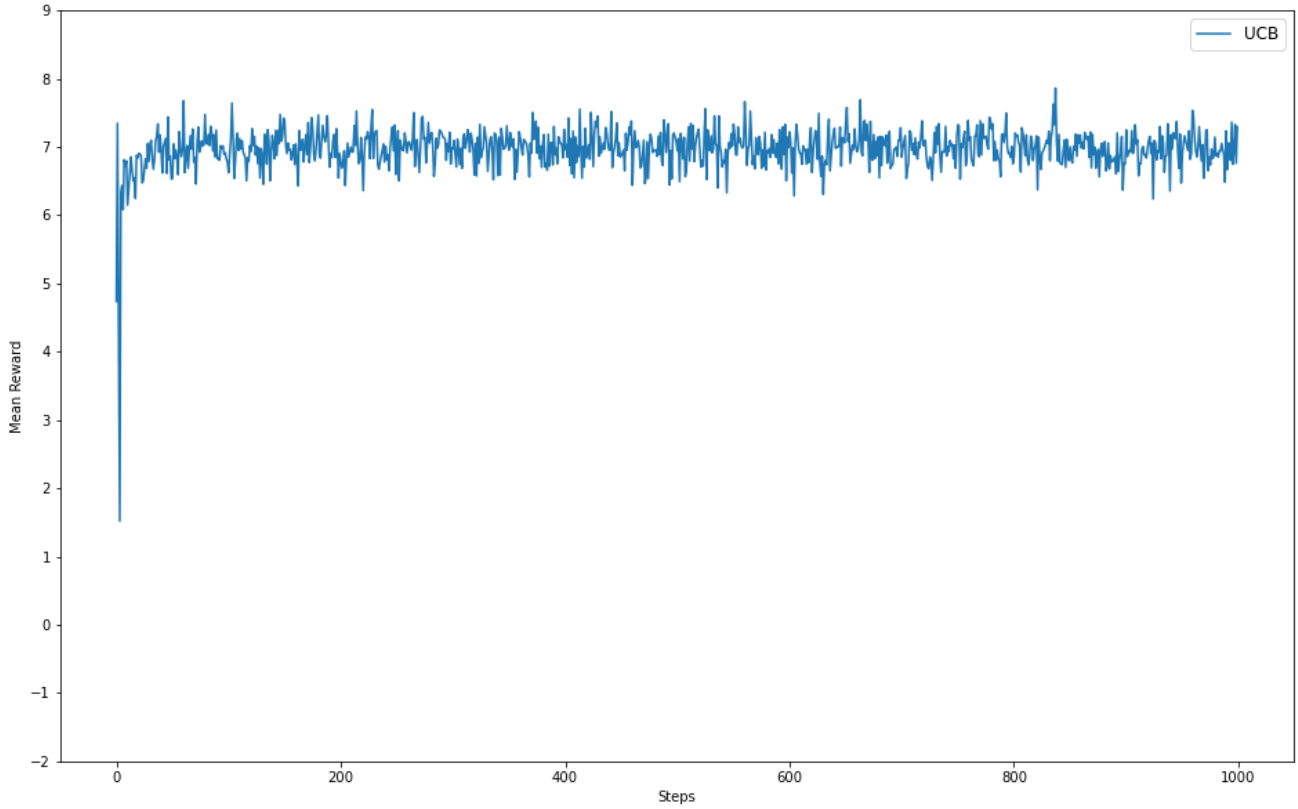
در این قسمت هم تابع هایی که در بالا ذکر کردم صدا زده شده.

نتایج

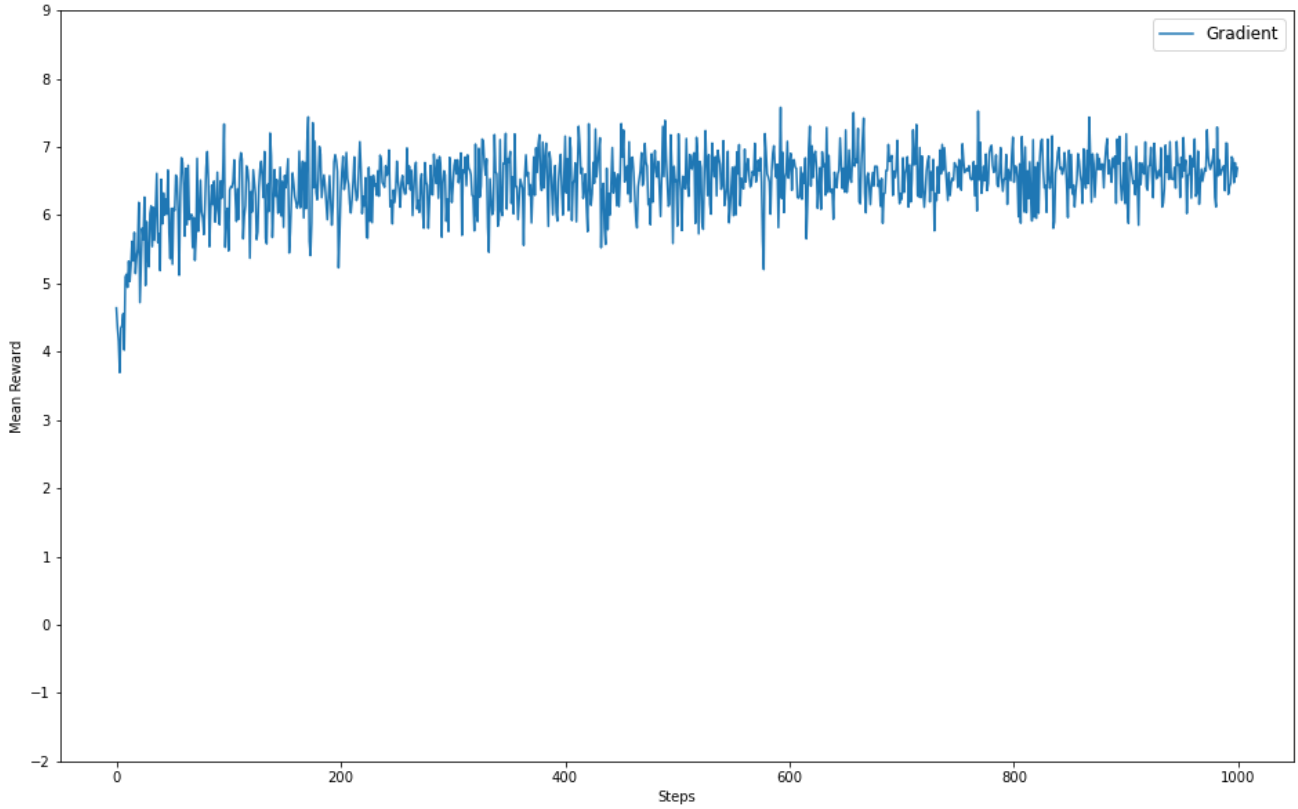
نمودار میانگین پاداش ها برای 20 اجرا که در هر اجرا عامل 1000 بار اکشن انجام میدهد (تعداد استپ ها) برای به صورت زیر است.



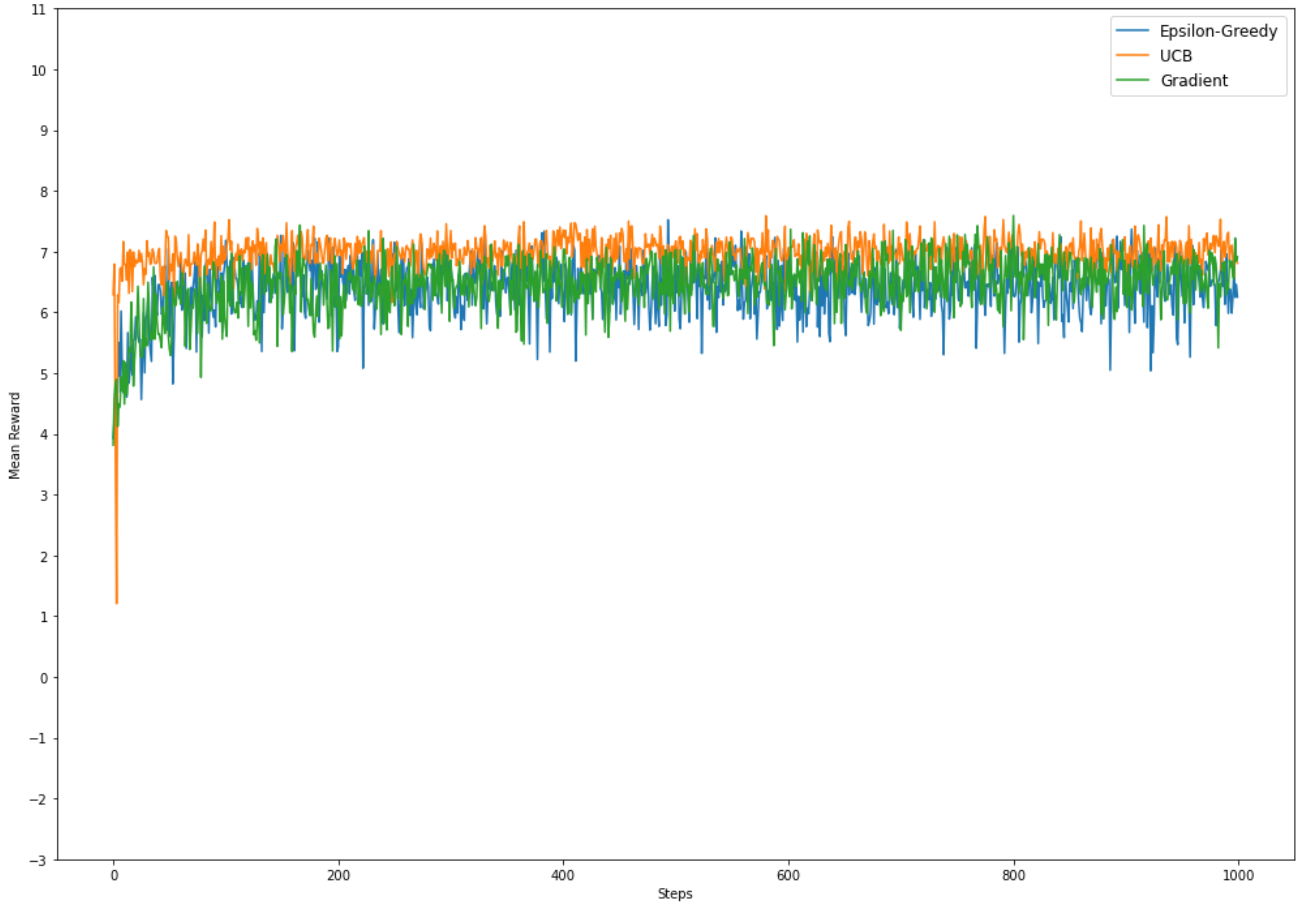
شکل 1- نمودار میانگین پاداش عامل **eps-greedy**



شکل 2 – نمودار میانگین پاداش عامل **ucb**



شکل 3 – نمودار میانگین پاداش عامل **gradient**



شکل 4 – مقایسه سرعت و مقدار همگرایی الگوریتم های مختلف

در شکل 4 الگوریتم های مختلف برای مقایسه رسم شده اند. طبق مشاهدات بنظر میرسد که سرعت یادگیری ucb از دو الگوریتم دیگر بیشتر است. همچنین مقدار همگرا شده در ای الگوریتم بیشتر است و نوسان کمتری در طی اجرا دارد. سرعت همگرایی ϵ-greedy و gradient شبیه یکدیگر میباشد اما مقدار همگرا شده در gradient مقداری بیشتر است.

ب)

توضیح پیاده سازی

```
def cumulative_mean_reward(mean_reward):
    cmr = np.zeros(len(mean_reward))

    for i in range(len(mean_reward)):
        cmr[i] = np.mean(mean_reward[:i+1])

    return cmr
```

این تابع میانگین تجمعی پاداش ها را محاسبه میکند. یعنی مقدار هر عنصر در آرایه میانگین پاداش ها از ابتدای آرایه تا اندیس این عنصر میباشد.

```
def mean_reward_diff_uparam(utility_parameters, agentType):
    mr_list = []
    cmr_list = []
    labels = []
    for parameter in utility_parameters:
        rewards = [Reward(first, second, 0.7, alpha=parameter[0], beta=parame
ter[1], lambdaa=parameter[2]) \
                   for first, second in zip(firstParameters, secondParameters
)]
        env = MutliArmedBanditEnvironment(rewards, 1000, '2')

        if agentType == 'eps-greedy':
            eps_greedy_agent = EpsilonGreedyBanditAgent('2', env, 0.2)
            mr = agent_run(eps_greedy_agent, env, 20, 1000)
            cmr = cumulative_mean_reward(mr)

        elif agentType == 'ucb':
            ucb_agent = UCBBanditAgent('2', env, 2.0)
            mr = agent_run(ucb_agent, env, 20, 1000)
            cmr = cumulative_mean_reward(mr)


        elif agentType == 'gradient':
            gradient_agent = GradientBanditAgent('2', env, 0.1)
            mr = agent_run(gradient_agent, env, 20, 1000)
            cmr = cumulative_mean_reward(mr)


        mr_list.append(mr)
        cmr_list.append(cmr)
        labels.append('alpha: ' + str(parameter[0]) + ', beta: ' + str(parame
ter[1]) + ', lambda: ' + str(parameter[2]))

    return mr_list, cmr_list, labels
```

این تابع یک لیست از میانگین و میانگین تجمعی پاداش ها با پارامتر های مختلف در utitlity function برمیگرداند. همچنین برای اینکه مشخص شود هر نمودار مربوط به کدام یک از پارامتر ها میباشد به آن یک لیبل تخصیص داده شده است.

```
def plot_mean_reward_diff_uparam(mr_list, labels, agentType, lower, upper):
    step_no = np.arange(len(mr_list[0]))
    plt.figure(figsize=(17, 11))
    for i, mr in enumerate(mr_list):
        plt.plot(step_no, mr, label=labels[i])
        plt.title(agentType)
```

```
    plt.xlabel("Steps")
    plt.ylabel("Cumulative Mean Reward")
    plt.ylim([lower, upper])
    plt.legend()
    plt.show();
```

در این تابع نمودار خروجی تابع `mean_reward_diff_uparam` کشیده میشود. ورودی این تابع میتواند میانگین پاداش ها یا میانگین تجمعی پاداش ها باشد.

```
u_param = [[0.2,1,1],[0.8,1,1],[1,1,1]]
mr_list, cmr_list, labels = mean_reward_diff_uparam(u_param, 'eps-greedy')
plot_mean_reward_diff_uparam(cmr_list, labels, 'Epsilon-Greedy', 3, 7.5)
```

از آن جایی که تابع utility function سه پارامتر دارد؛ برای بررسی تاثیر هر پارامتر مقادیر دو پارامتر دیگر با مقدار ثابت نگه داشته شده اند. در کد بالا اولین پارامتر یعنی آلفا بررسی شده است و همچنین الگوریتم آن ϵ-greedy میباشد.

ضمنا به دلیل نوسان نمودار میانگین پاداش ها، برای بررسی پارامتر ها از میانگین تجمعی استفاده شده است. به این ترتیب هم مشاهده نمودار هم تحلیل آن دقیقتر خواهد بود.

تاثیر تمام پارامتر ها با الگوریتم های مختلف پیاده سازی شده اند اما چون کد آن شبیه کد بالا میباشد برای جلوگیری از طولانی شدن بیهوده گزارش از آوردن آن صرف نظر کرده ام.

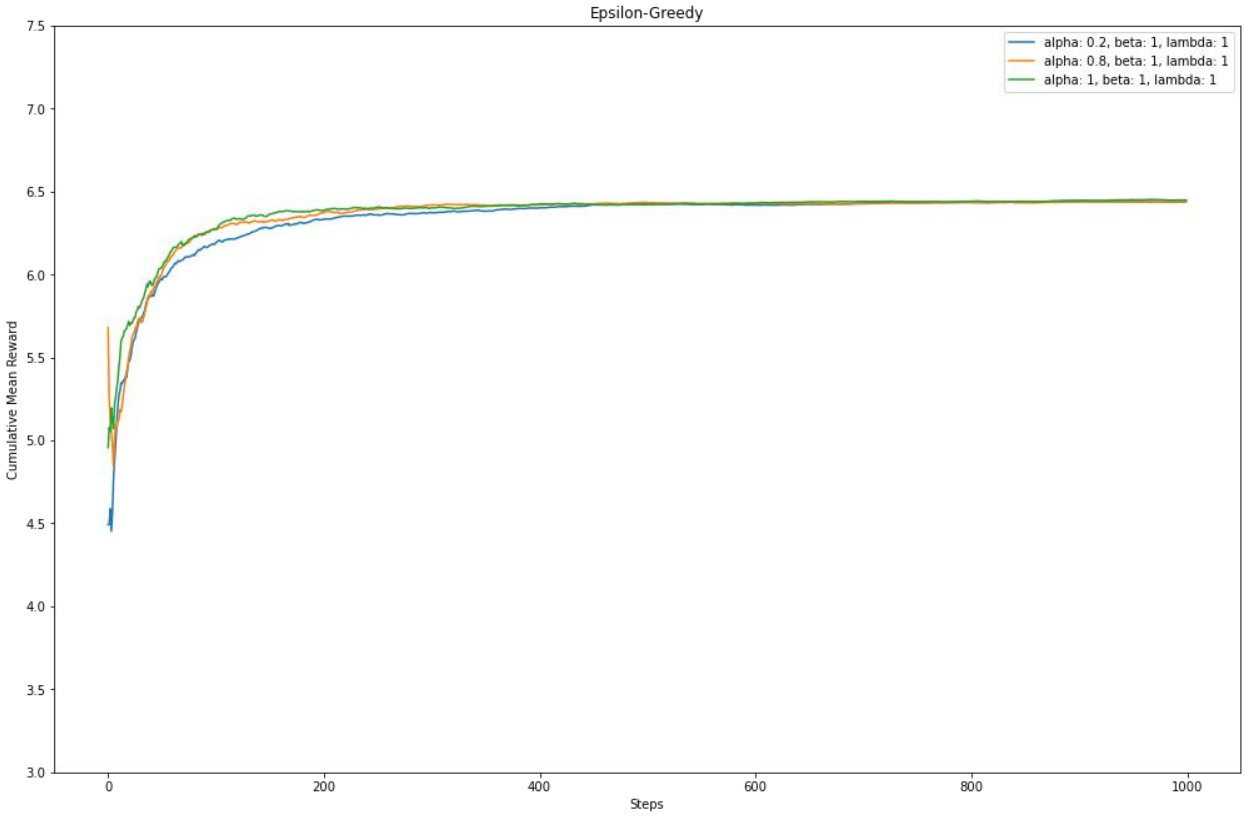
نتایج

برای بررسی تاثیر utitliy مقادیر آلفا و بتا و لامبدا اینگونه تنظیم شده اند.

آلفا: 0.2 و 0.8 و 1

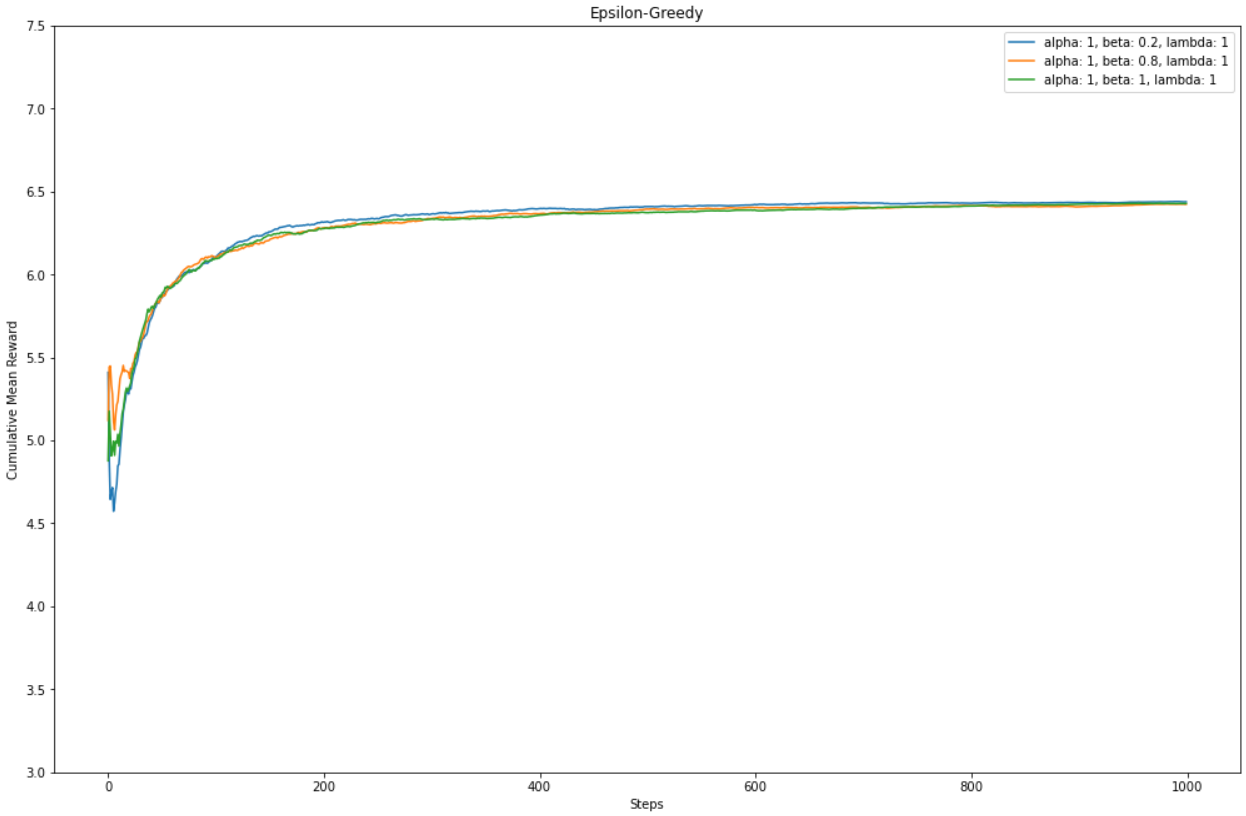
بتا: 0.2 و 0.8 و 1

لامبادا: 1 و 2.5 و 3.5



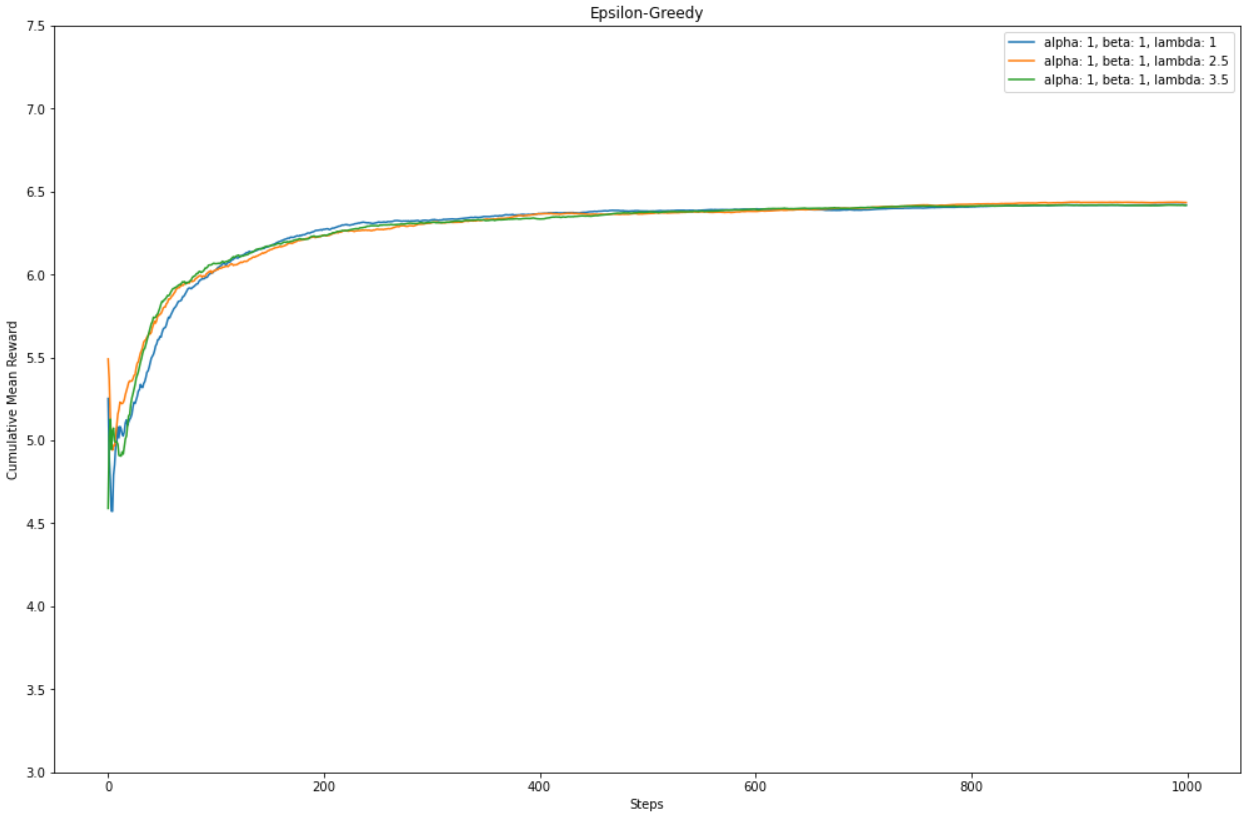
شکل 5 – تاثیر آلفا – الگوریتم **ϵ-greedy**

با توجه به مشاهدات نمودار شکل 5 سرعت یادگیری آلفا برابر با 0.2 در شروع یادگیری کندتر از دو پارامتر دیگر میباشد. اما در نهایت همه آن ها به یک مقدار یکسان همگرا شده اند. دلیل اینکه سرعت یادگیری الگوریتم با آلفا برابر با 0.2 کمی کندتر است این است که فاصله مقدار نگاشت شده ناشی از تابع منفعت (utility function) بین مقادیر پاداش های مثبت کم میشود. برای مثال با اعمال تابع منفعت روی اعداد 7 و 5 (که مربوط به میانگین دو توزیع نرمال اعمال میباشند)، خروجی 1.47 و 1.37 حاصل میشود که باعث میشود الگوریتم در پیدا کردن اکشن مربوط به پاداش بالاتر یعنی عدد 7 دیرتر عمل کند.



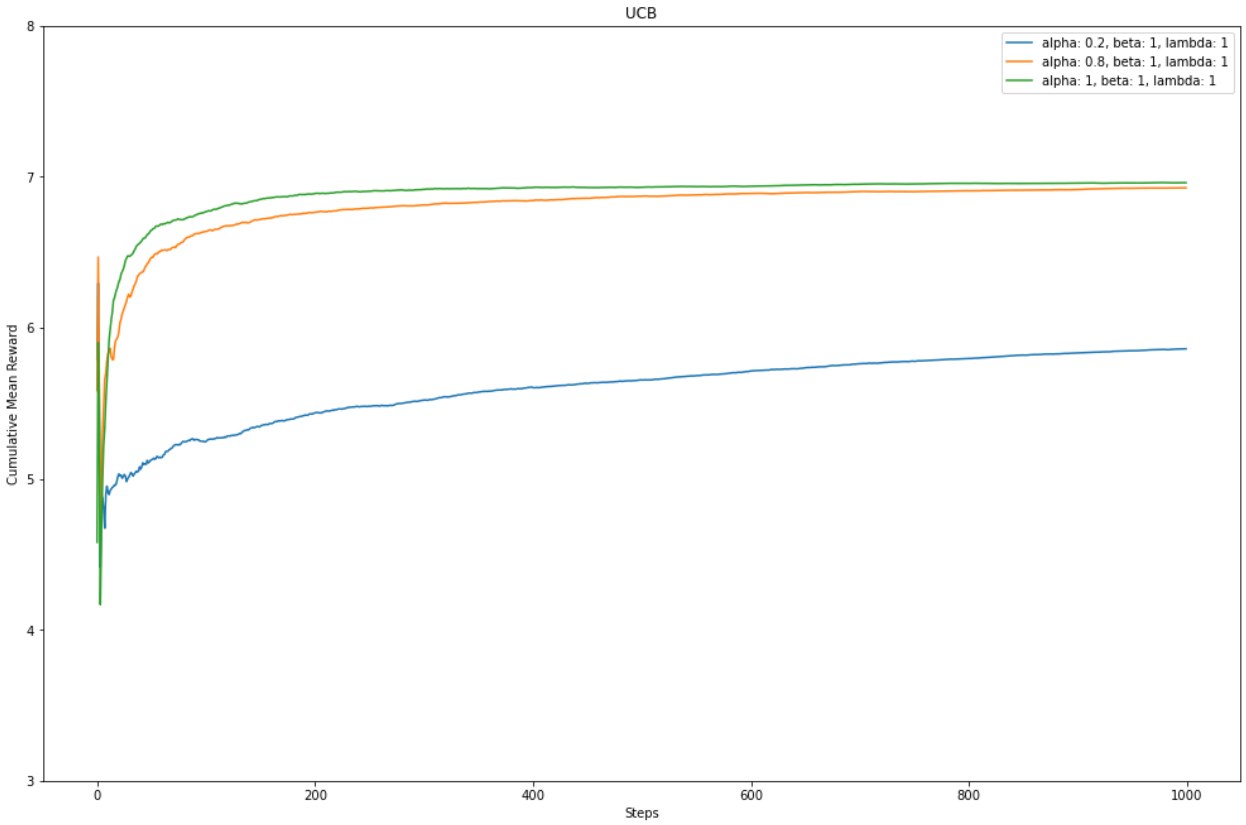
شکل 6 – تاثیر بتا – الگوریتم **ϵ-greedy**

با توجه به مشاهدات شکل 6، بتا روی این الگوریتم تاثیر آشکاری ندارد و دلیل آن احتمال پایین دریافت پاداش های منفی میباشد. اگر به فرض الگوریتم اپسیلون گریدی با احتمال کمتر مساویِ اپسیلون یک اکشن را به صورت تصادفی بردارد پس احتمال انتخاب آن اکشن بین 4 اکشن 0.25 میباشد. همچنین از آنجایی که فقط یک اکشن مقدار پاداش منفی آن هم با احتمال 0.3 را میدهد پس در کل دریافت پاداش منفی کم میباشد و به این ترتیب تاثیر پارامتر بتا ناچیز است.



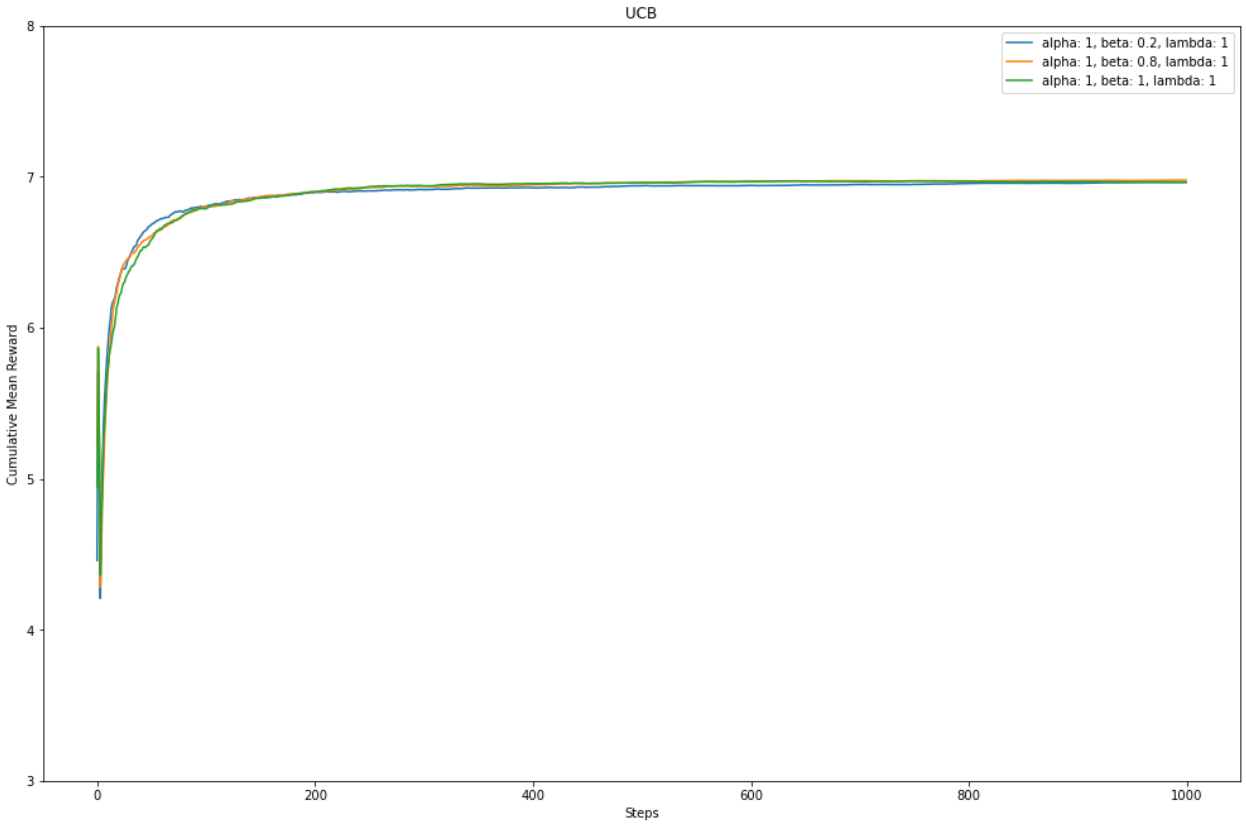
شکل 7 – تاثیر لامبدا – الگوریتم **ϵ-greedy**

با توجه به مشاهدات شکل 7، پایین ترین سرعت یادگیری در ابتدای کار مربوط به لامبدا برابر با 1 میباشد. هر سه الگوریتم در نهایت به یک مقدار همگرا میشوند. دلیل اینکه لامبدا هم تاثیر آشکاری ندارد به همان دلیل ذکر شده برای پارامتر بتا میباشد.



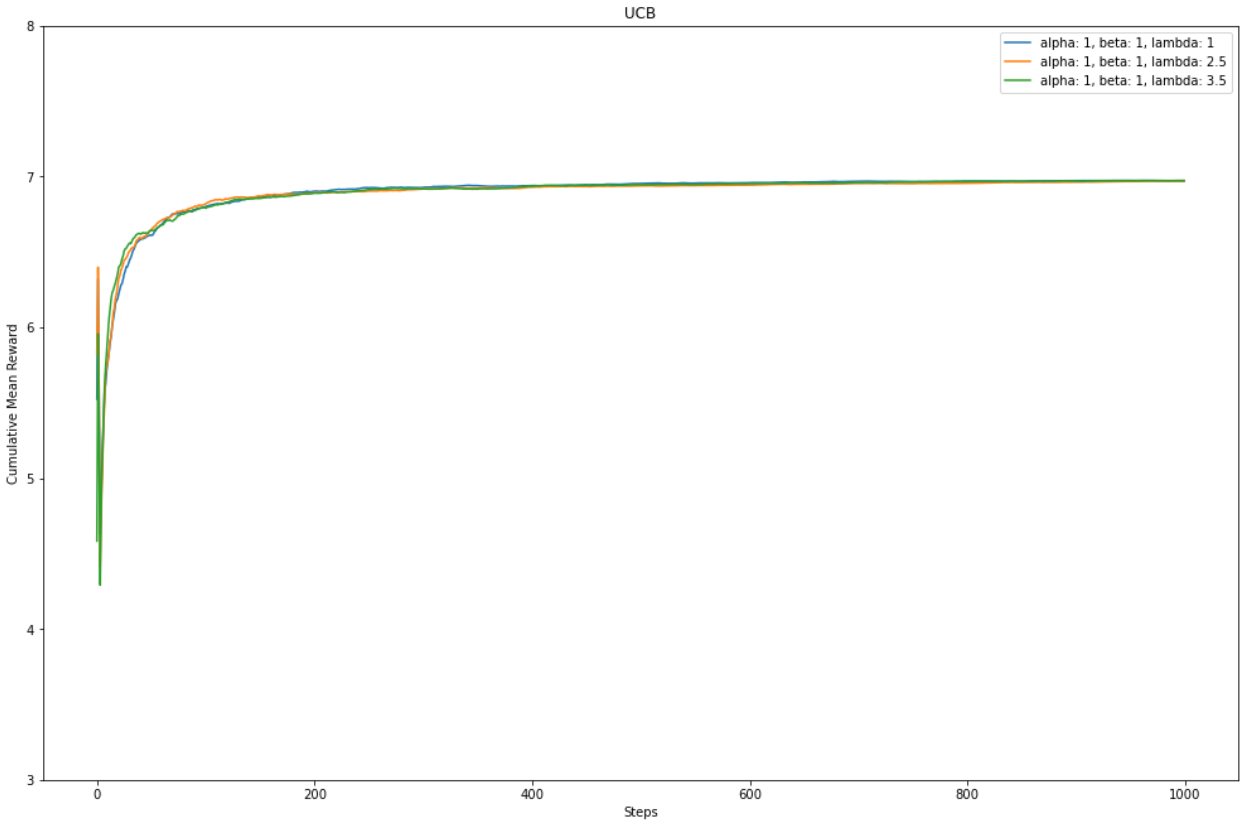
شکل 8 – تاثیر آلفا – الگوریتم **ucb**

با توجه به مشاهدات شکل 8، آلفا برابر با 1 و 0.8 و 0.2 به ترتیب بالاترین سرعت یادگیری را دارند. برای آلفا برابر با 0.2 هم سرعت یادگیری و هم مقدار همگرا شده تفاوت فاحشی دارد. همانطور که قبلا گفتیم با آلفا برابر 0.2 مقادیر خروجی تابع منفعت اختلاف کمتری نسبت به پاداش های اصلی دارند. ضمن اینکه اگر نمودار های پاداش الگوریتم ucb را با دو الگوریتم دیگر یعنی اپسیلون گریدی و گرادیان مقایسه کنیم متوجه میشویم ucb به نسبت قابل توجهی سریعتر اکشن بهینه را پیدا میکند و از آن جایی که این الگوریتم همیشه بهترین اکشن را انتخاب میکند پس ممکن است در یکی از اکشن های بهینه محلی گیر کند.



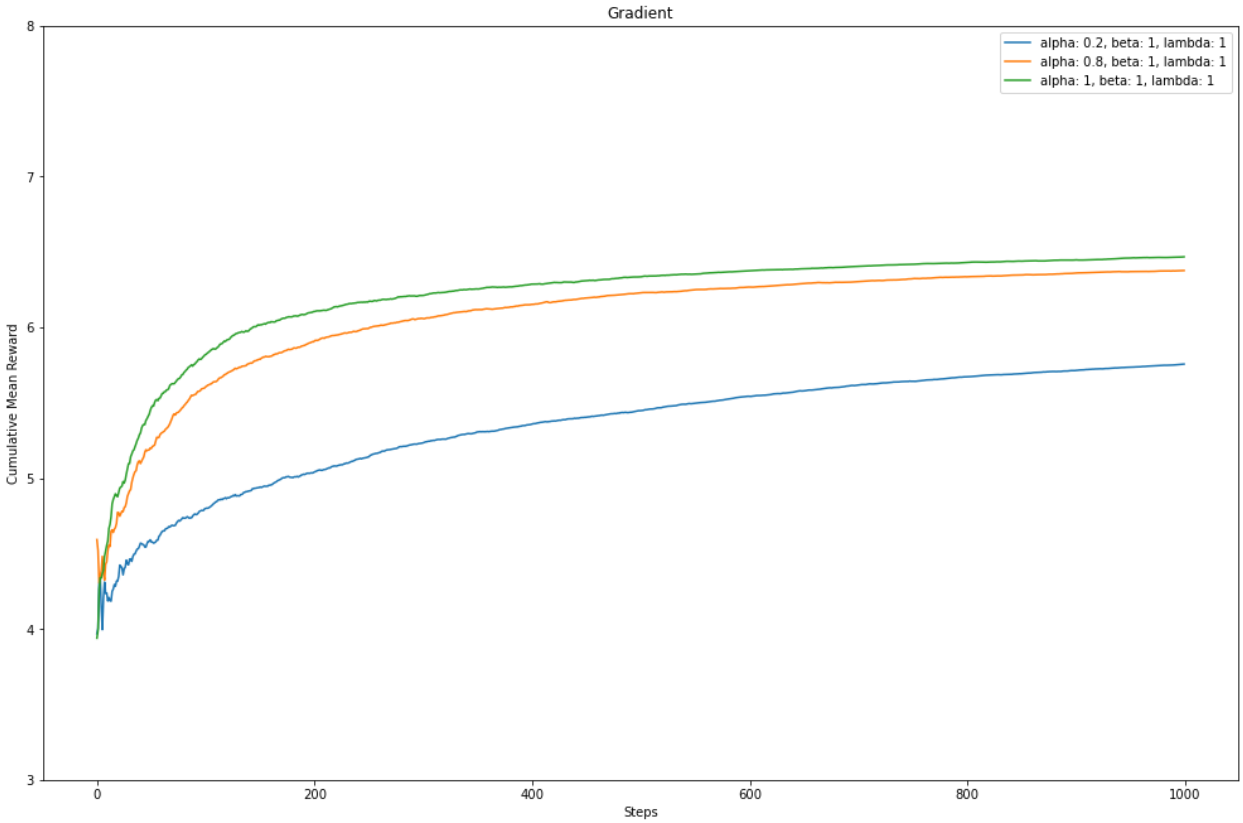
شکل 9 – تاثیر بتا – الگوریتم **ucb**

با توجه به مشاهدات شکل 9 بنظر میرسد مقدار بتا در این الگوریتم هم تاثیر آشکاری در سرعت و مقدار همگرایی ندارد و دلیل آن مانند دلیل ذکر شده برای الگوریتم اپسیلون گریدی میباشد. (البته در الگوریتم ucb سیاست الگوریتم همیشه برمبنای انتخاب اکشن با بالاترین میانگین utility میباشد و مانند الگوریتم اپسیلون گریدی که با احتمال کمی تا پایان اجرا جستجو انجام میدهد نیست)



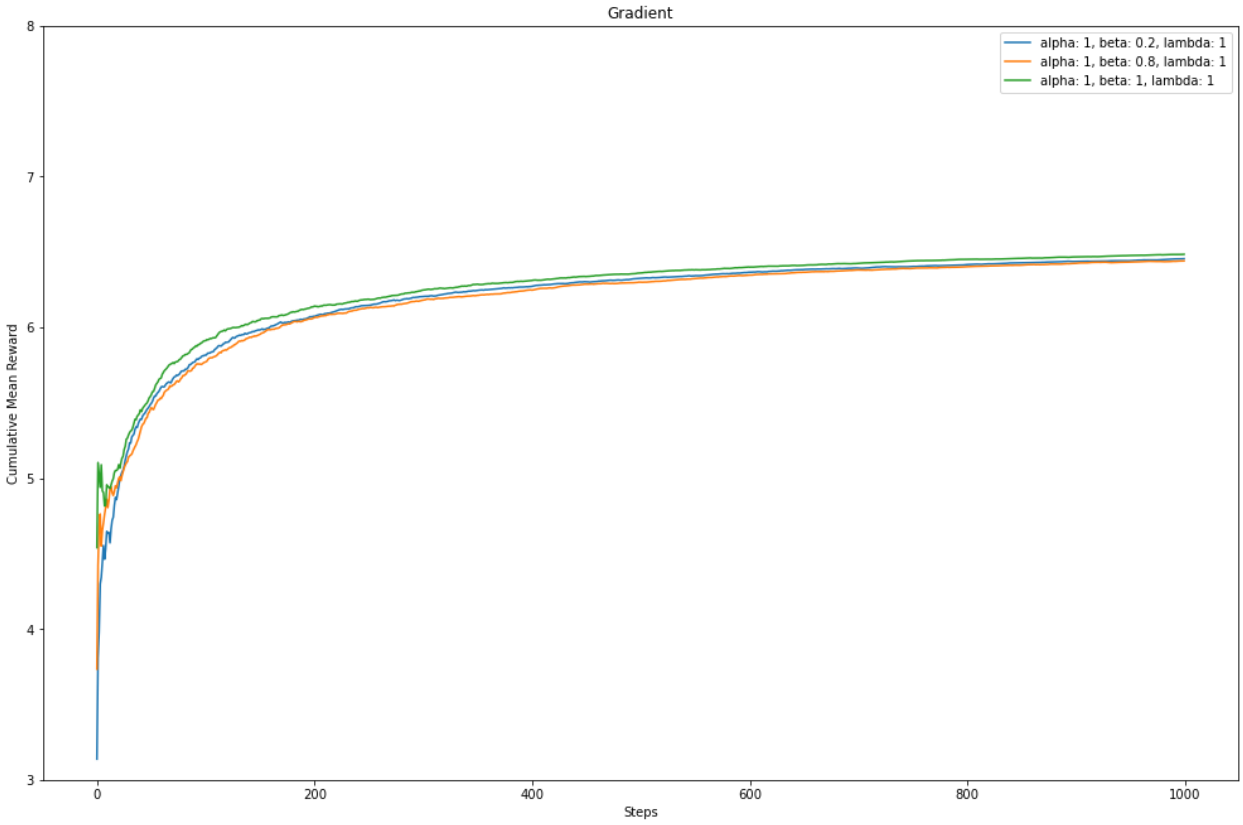
شکل 10 – تاثیر لامبدا – الگوریتم **ucb**

با توجه به مشاهدات شکل 10 بنظر میرسد مقدار لامبدا در این الگوریتم هم تاثیر آشکاری در سرعت و مقدار همگرایی ندارد و دلیل آن مانند دلیل ذکر شده برای الگوریتم اپسیلون گریدی میباشد.



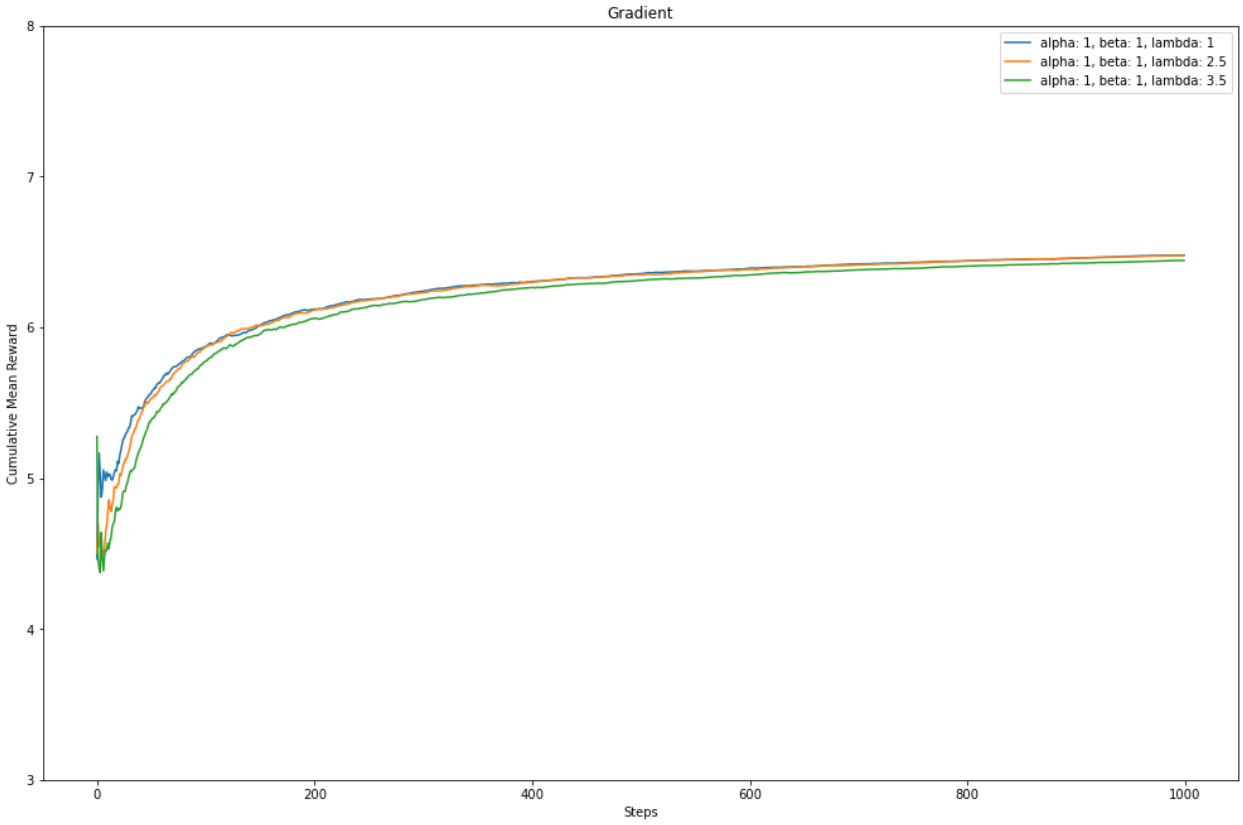
شکل 11 – تاثیر آلفا – الگوریتم **gradient**

با توجه به مشاهدات شکل 11، آلفا برابر با 1 و 0.8 و 0.2 به ترتیب بالاترین سرعت یادگیری را دارند. در اینجا هم مانند الگوریتم ucb آلفا برابر با 0.2 هم در سرعت و هم در مقدار همگرایی اختلاف فاحشی با دو مقدار دیگر دارد و دلیل آن کم بودن مقدار اختلاف خروجی ناشی از تابع منفعت و همچنین گیر کردن در اکشن بهینه محلی میباشد. در الگوریتم گرادیان اکشن های با ترجیح بالاتر، بیشتر انتخاب میشوند. این الگوریتم مانند الگوریتم اپسیلون گریدی نیست که با احتمال کمتر مساوی اپسیلون اکشن های تصادفی را تا پایان اجرا انتخاب کند و با جستجو بتواند از بهینه محلی فرار کند.



شکل 12 – تاثیر بتا – الگوریتم **gradient**

با توجه به مشاهدات شکل 12، بتا برابر با 1 به مقدار کمی سرعت یادگیری بالاتری در ابتدای کار دارد. البته با تقریب نسبتا خوبی میتوان گفت مقدار بتا در این الگوریتم هم تاثیر آشکاری در مقدار همگرایی ندارد و دلیل آن مانند دلیل ذکر شده برای الگوریتم اپسیلون گریدی میباشد. (البته در الگوریتم گرادیان سیاست الگوریتم همیشه برمبنای انتخاب اکشن با ترجیح بالاتر است و مانند الگوریتم اپسیلون گریدی که با احتمال کمی تا پایان اجرا جستجو انجام میدهد نیست)



شکل 13 – تاثیر لامبدا – الگوریتم **gradient**

با توجه به مشاهدات شکل 13، مقدار لامبدا برابر با 3.5 به مقدار کمی سرعت یادگیری پایین تری در ابتدای کار دارد. . البته با تقریب نسبتا خوبی میتوان گفت مقدار لامبدا در این الگوریتم هم تاثیر آشکاری در مقدار همگرایی ندارد و دلیل آن مانند دلیل ذکر شده برای الگوریتم اپسیلون گریدی میباشد.

قسمت دوم

الف)

توضیح پیاده سازی

```
class DecayingEpsilonGreedyBanditAgent(EpsilonGreedyBanditAgent):
    def __init__(self, id, environment, epsilon):
        super(DecayingEpsilonGreedyBanditAgent, self).__init__(id, environmen
t, epsilon)
        self.initial_epsilon = epsilon

    def update(self, reward, action):
        self.rewards_history[action].append(reward)
        self.rewards_mean[action] = np.mean(self.rewards_history[action])
        t = self.environment.state['length']
        self.epsilon = self.initial_epsilon * np.exp(-t/100)
```

```
    def get_epsilon_value(self):
        return self.epsilon
```

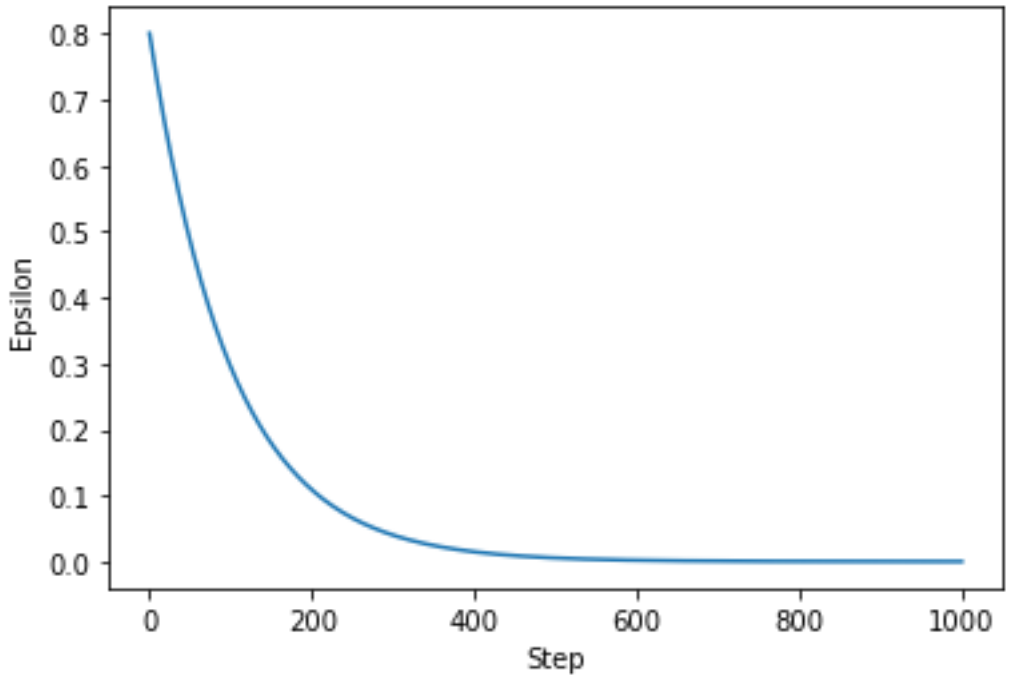
کلاس بالا پیاده سازی الگوریتم DecayingEpsilonGreedyBanditAgent است که از کلاس EpsilonGreedyBanditAgent ارث بری میکند. این الگوریتم مقدار ϵ را طبق فرمول زیر کاهش میدهد[2].

```
self.epsilon = self.initial_epsilon * np.exp(-t/100)
```

با استفاده از این فرمول ϵ در طول زمان به صورت نمایی کاهش میابد. t تعداد استپ هایی است که عامل تا آن لحظه طی کرده است.

```
env = MutliArmedBanditEnvironment(rewards, 1000, '4')
decaying_eps_greedy_agent = DecayingEpsilonGreedyBanditAgent('4', env, 0.8)
trials = 1000
epsilons = np.zeros(trials)

for step in range(trials):
    epsilons[step] = decaying_eps_greedy_agent.get_epsilon_value()
    obs, r, d, i = decaying_eps_greedy_agent.take_action()

step_no = np.arange(trials)
plt.plot(step_no, epsilons)
plt.xlabel("Step")
plt.ylabel("Epsilon")
plt.show();
```

در قطعه کد بالا یک نمونه از عامل با سیاست decaying-ϵ-greedy ساخته شده است. سپس طی 1000 استپ با محیط تعامل داشته و تمام مقادیر ϵ در یک آرایه ذخیره شده است. در نهایت نمودار آن را رسم کرده ام که در شکل 14 قابل مشاهده است.



شکل 14 – نمودار ϵ به زمان (استپ های طی شده)

```
def mean_reward_diff_eps(epsilons):
    mr_list = []
    labels = []

    for key, value in epsilons.items():
        if key == "eps-greedy":
            for epsilon in value:
                env = MutliArmedBanditEnvironment(rewards, 1000, '4')
                eps_greedy_agent = EpsilonGreedyBanditAgent('4', env, epsilon
)
                mr = agent_run(eps_greedy_agent, env, 20, 1000)
                mr_list.append(mr)
                labels.append(key + ':  ' + str(epsilon))

        elif key == "decaying-eps-greedy":
            for epsilon in value:
                env = MutliArmedBanditEnvironment(rewards, 1000, '4')
                decaying_eps_greedy_agent = DecayingEpsilonGreedyBanditAgent(
'4', env, epsilon)
                mr = agent_run(decaying_eps_greedy_agent, env, 20, 1000)
                mr_list.append(mr)
                labels.append(key + ':  ' + str(epsilon))


    return mr_list, labels
```

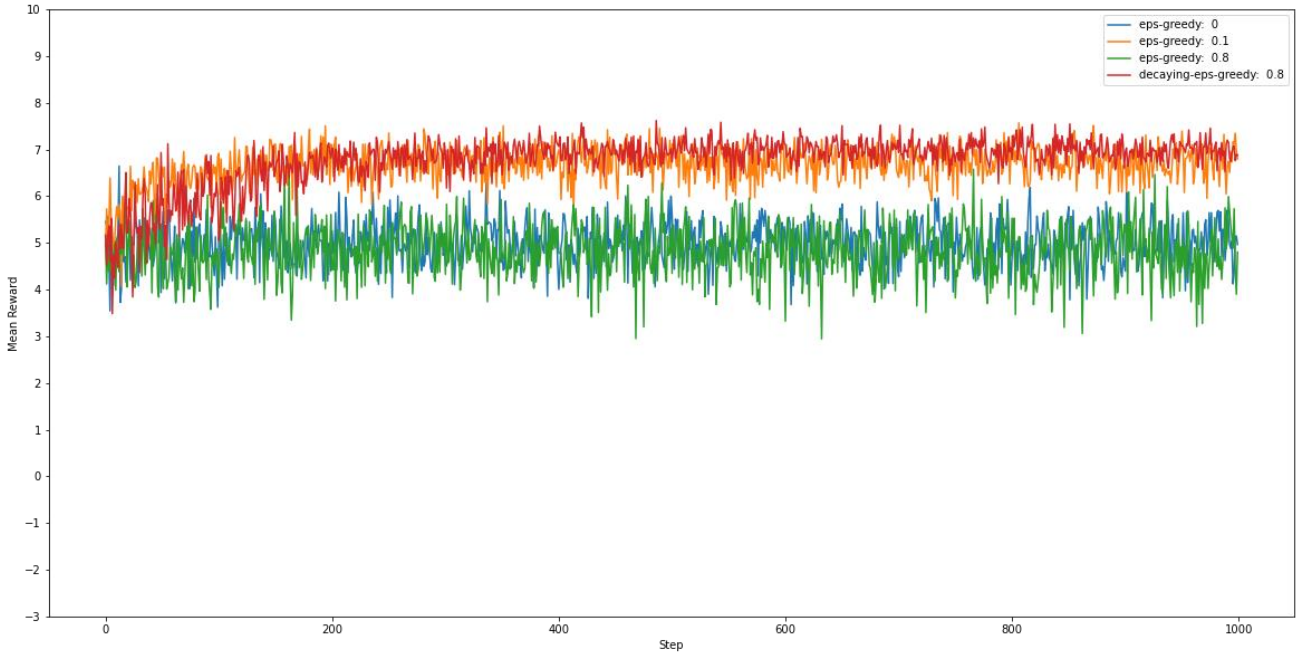
این تابع دیکشنری ϵ های مختلف را گرفته و بسته به اینکه الگوریتم decaying باشد یا نه لیست میانگین پاداش ها را به همراه لیبل مقادیر ϵ برمیگرداند (این لیبل ها برای تشخیص نمودار های متفاوت از هم استفاده میشود).

```
def plot_mean_reward_diff_eps(mr_list, labels, lower, upper):
    step_no = np.arange(len(mr_list[0]))
    plt.figure(figsize=(20, 10))

    for i, mr in enumerate(mr_list):
        plt.plot(step_no, mr, label=labels[i])


    plt.xlabel("Step")
    plt.ylabel("Mean Reward")
    plt.ylim([lower, upper])
    new_list = range(int(np.floor(np.min(lower))), int(np.ceil(np.max(upper))
+1))
    plt.yticks(new_list)
    plt.legend()
    plt.show();
```

در این قطعه کد نمودار میانگین پاداش ها با انواع مختلف ϵ رسم میشود.

```
epsilons = {"eps-greedy":[0, 0.1, 0.8],
            "decaying-eps-greedy":[0.8]}


mr_list, labels = mean_reward_diff_eps(epsilons)
plot_mean_reward_diff_eps(mr_list, labels, -3, 10)
```

لیست ϵ های دلخواه به تابع `mean_reward_diff_eps` پاس داده میشود و تابع `plot_mean_reward_diff_eps` صدا زده میشود. مقدار دهی اولیه ϵ در الگوریتم `decaying-ϵ-greedy` برابر با 0.8 میباشد.



شکل 15 – میانگین پاداش به زمان با ϵ های مختلف

با توجه به مشاهدات شکل 15 ترتیب سرعت یادگیری در ابتدای کار به صورت زیر میباشد:

$$\epsilon\text{-greedy: } 0 = \epsilon\text{-greedy: } 0.8 > \epsilon\text{-greedy: } 0.1 > \text{decaying-}\epsilon\text{-greedy: } 0.8$$

همچنین مقدار همگرا شده در استپ 1000 به صورت زیر میباشد:

$$\text{decaying-}\epsilon\text{-greedy: } 0.8 > \epsilon\text{-greedy: } 0.1 > \epsilon\text{-greedy: } 0 > \epsilon\text{-greedy: } 0.8$$

الگوریتم با اپسیلون ϵ-greedy: 0 به دلیل اینکه در مقدار بهینه محلی گیر می افتد تفاوت آشکاری با decaying-ϵ-greedy: 0.8 وϵ-greedy: 0.1 دارد. همچنین طبق مشاهدات دیده میشود الگوریتم ϵ-greedy: 0.8 هم به مقدار مناسبی همگرا نشده است. دلیل این اتفاق این است که الگوریتم در 80 درصد مواقع سیاست انتخاب اکشن تصادفی میباشد و به مقدار میانگین های به دست آمده برای هر اکشن توجهی ندارد.

بهترین مقدار همگرایی مربوط به الگوریتم decaying-ϵ-greedy: 0.8 میباشد. این الگوریتم در ابتدای کار در 80 درصد مواقع اکشن را به صورت تصادفی انتخاب میکند و به عبارتی به جستجو گسترده و جمع آوری دانش میپردازد. سپس با افزایش زمان مقدار اپسیلون کاهش میابد تا جایی که تقریبا به صفر میرسد و الگوریتم با سیاست انتخاب بهترین اکشن یا اکشن بهینه پیش میرود. به عبارتی از دانشی که در ابتدای کار کسب کرده است بهره میبرد تا انتخاب بهینه را انجام دهد.

ب)

1)

شبه کد پیشنهادی تغییر یافته الگوریتم $UCB1$ میباشد و به صورت زیر است[3] :

1: Initialization:
2: Set appropriate $\beta_1, \beta_2, \beta_3, \beta_4$. Set $t = 1$, $\bar{\mu}t(i) = 0$, $\forall i \in I$.
3: Repeat at the beginning of each time slot
4: Let $I_t = t$, try action I_t, observe action $I_t^{target-a}, I_t^{target-b}, I_t^{target-c}$;
5: Get reward $r_t(I_t)$, $t = t + 1$;
6: Until $t > K$;
7: Repeat at the beginning of each time slot
8: Update $N_t(i)$ and $A_t(i)$, $B_t(i)$, $C_t(i)$ as in (1), and (2) , and (3) , and (4);
9: Update $\bar{\mu}_t(i)$ and $c_t^{OUCB}(i)$ as in (5), and (6);
10: Determine $I_t = argmax_{i \in I} \bar{\mu}_t(i) + c_t^{OUCB}(i)$;
11: Try action I_t, observe action $I_t^{target-a}, I_t^{target-b}, I_t^{target-c}$;
12: Get reward $r_t(I_t)$, $t = t + 1$;
13: Until $t > T$;

مجموعه بازوها یا همان اکشن ها $I := \{1, 2, \cdots, K\}$

معادله های اشاره شده در شبه کد به شرح زیر میباشند:

(1) $N_t(i) := \sum_{s=1}^{t-1} I\{I_s = i\}$,
(2) $A_t(i) := \sum_{s=1}^{t-1} I\{I_s^{target-a} = i\}$
(3) $B_t(i) := \sum_{s=1}^{t-1} I\{I_s^{target-b} = i\}$
(4) $C_t(i) := \sum_{s=1}^{t-1} I\{I_s^{target-c} = i\}$

معادله 1: تعداد دفعاتی که بازو i توسط بازیکن Mr. Nobody تا زمان t انجام داده شده است.

معادله 2: تعداد دفعاتی که بازو i توسط بازیکن a تا زمان t انجام داده شده است.

معادله 3: تعداد دفعاتی که بازو i توسط بازیکن b تا زمان t انجام داده شده است.

معادله 4: تعداد دفعاتی که بازو i توسط بازیکن c تا زمان t انجام داده شده است.

(5) $\bar{\mu}_t(i) := \frac{1}{N_t(i)} \sum_{s=1}^{t-1} r_s(i) I\{I_s = i\}$

معادله 5 همان تخمین پاداش یک بازو تا زمان t میباشد.

$$(6)\ c_t^{OUCB}(i) := \sqrt{\frac{2lnt}{N_t(i)}}\ (\beta_1 + \beta_2\ [\delta_{t,a}(i)]_+ + \beta_3\ [\delta_{t,b}(i)]_+ + \beta_4\ [\delta_{t,c}(i)]_+)$$

معادله 6 از دو قسمت تشکیل شده است. عبارت $\sqrt{\frac{2lnt}{N_t(i)}}$ از الگوریتم $UCB1$ گرفته شده است و یک عبارت جدید در آن ضرب میشود.

$$(7)\ [\delta_{t,a}(i)]_+ := [\frac{A_t(i)-N_t(i)}{t}]$$
$$(8)\ [\delta_{t,b}(i)]_+ := [\frac{B_t(i)-N_t(i)}{t}]$$
$$(9)\ [\delta_{t,c}(i)]_+ := [\frac{C_t(i)-N_t(i)}{t}]$$

معادله 7: مقدار اهمیت بازو هایی را نشان میدهد که بازیکن a آن را بیشتر از بازیکن Mr. Nobody انجام داده است.

معادله 8: مقدار اهمیت بازو هایی را نشان میدهد که بازیکن b آن را بیشتر از بازیکن Mr. Nobody انجام داده است.

معادله 9: مقدار اهمیت بازو هایی را نشان میدهد که بازیکن c آن را بیشتر از بازیکن Mr. Nobody انجام داده است.

4 پارامتر $\beta_1,\ \beta_2,\ \beta_3,\ \beta_4$ باید به صورت مناسب تنظیم شوند و همچنین مقادیر آن ها باید شرط $\beta_1 + \beta_2 + \beta_3 + \beta_4 \leq 1$ را ارضا کند.

به صورت کلی اگر U عامل داشته باشیم که U-1 از آن ها target (یعنی بتوان از رفتار آن ها برای سیاست بهتر تقلب کرد) باشند و یکی از آن ها Mr. Nobody، آنگاه معادله 6 به صورت کلی به شکل زیر در می آید:

$$(10)\qquad c_t^{OUCB}(i) := \sqrt{\frac{2lnt}{N_t(i)}}\ (\beta_1 + \sum_{u=2}^{U} \beta_u\ [\delta_{t,u}(i)]_+)$$

$$\text{و}\quad \sum_{u=1}^{U} \beta_u \leq 1$$

(2

بهتر است اجازه دهد یک زمان معینی بگذرد و سپس تقلب کند. زیرا برای اینکه ترم $[\delta_{t,u}(i)]_+$ بتواند اثر خود را نشان بدهد و اهمیت یک اکشن بهینه را بالا ببرد عامل نیاز دارد در ابتدا رفتار دیگر عامل ها را مشاهده و جمع آوری نماید و بعد از آن ها استفاده کند. ضمنا از آنجایی که Mr. Nobody از سیاست عامل ها خبر ندارد ممکن است با

تقلب در ابتدای کار با رفتار بعضی عامل ها به اشتباه بیفتد. برای مثال اگر در $t = 1$، انتخاب اکشن عامل ها به صورت زیر باشد:

Mr. Mobody: action 1

Target a (ϵ-greedy): action 3

Target b (ucb): action 1

Target c (random): 3

از آنجایی که در قست های قبلی با مشاهده نمودار به این نتیجه رسیدیم ucb سرعت یادگیری و مقدار همگرایی بهتری دارد پس بهتر است Mr. Nobody به رفتار عامل ucb بیشتر توجه کند اما اگر در همین استپ بخواهد تقلب کند مشاهده میکنیم که اهمیت اکشن 3 بالا خواهد رفت.

$[\delta_{t,a}(1)]_+ = -1,\ [\delta_{t,b}(1)]_+ = 0,\ [\delta_{t,a}(1)]_+ = -1$

$[\delta_{t,a}(2)]_+ = 0,\ [\delta_{t,b}(2)]_+ = 0,\ [\delta_{t,a}(2)]_+ = 0$

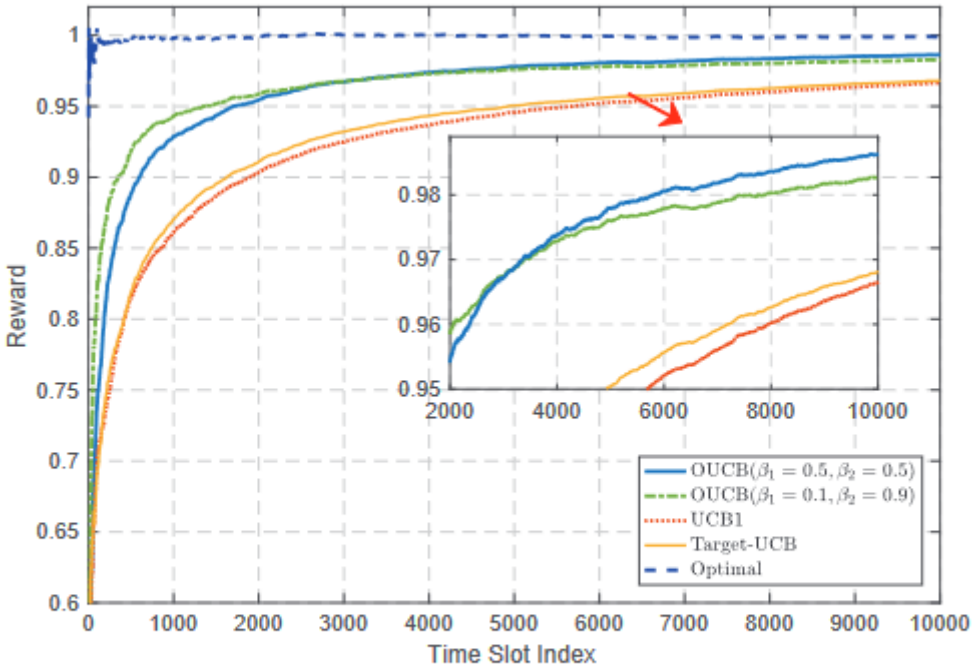
$[\delta_{t,a}(3)]_+ = 1,\ [\delta_{t,b}(3)]_+ = 0,\ [\delta_{t,a}(3)]_+ = 1$

در نتیجه مقدار $c_t^{OUCB}(3)$ بیشتر از بقیه اکشن ها میشود و میتواند به عنوان اکشن بهینه انتخاب شود. این در حالی است که عامل رندوم اکشن 3 را بدون هیچ منطقی انتخاب کرده است. ولی اگر Mr. Nobody به جمع آوری اطلاعات بپردازد بعد از مدتی تعداد تکرار اکشن 1 توسط الگوریتم ucb افزایش میابد و باعث بالا رفتن اهمیت این اکشن خواهد شد.

در الگوریتم پیشنهاد شده سوال قبلی تعداد استپی که عامل Mr. Nobody به مشاهده و شمارش اکشن های عامل های دیگر میپردازد K میباشد. که K تعداد بازو ها یا همان فضای اکشن میباشد. سپس بعد از گذشت K استپ این مشاهدات را با سیاست خود ترکیب و استفاده میکند.

3)

با توجه به مقاله Social Bandit Learning: Strangers Can Help اگر عامل target در 30 درصد مواقع بهینه عمل کند و در 70 درصد رندوم (اسم این عامل در مقاله Hybrid نام گذاری شده است)، باز هم تقلب برای عاملی که از رفتار target را تقلب میکند سودمند خواهد بود. این موضوع در شکل 16 از مقاله مشاهده میشود. همچنین $\beta_1 = 0.5,\ \beta_2 = 0.5$ در این حالت مناسب خواهد بود. حال به مسئله خود برمیگردیم. بنظر میرسد این رفتار Hybrid معادل همان خطای چشمی Mr. Nobody میباشد. پس همچنان تقلب برای او سودمند خواهد بود و میتواند مقدار $\beta_1 = 0.25,\ \beta_2 = 0.25,\ \beta_3 = 0.25,\ \beta_4 = 0.25$ بگذارد.



(c) HYBRID target.

شکل 16

روند اجرای کد پیاده‌سازی

فایل اجرایی این سوال n_armed_bandit.ipynb میباشد که با run all از کل نوت بوک خروجی های مورد نظر در هر سلول چاپ میشود.

سوال 2 - سوال تئوری

هدف سوال

بهره گرفتن از اطلاعاتی نظیر واریانس پاداش تخمین زده شده توسط عامل های دیگری که با ما در یک محیط حضور دارند.

الف)

شبه کد پیشنهادی شبیه سوال قبل میباشد اما در اینجا چون واریانس پاداش ها را داریم از این مقادیر استفاده خواهیم کرد. تعداد عامل ها U میباشد که عامل اول یادگیرنده و بقیه هدف (برای تقلب) هستند.

1: Initialization:
2: Set appropriate $\beta_1, \beta_2, \dots, \beta_U$. Set $t = 1$, $\bar{\mu}t(i) = 0$, $\forall i \in I$.
3: Repeat at the beginning of each time slot
4: Let $I_t = t$, try action I_t, observe action $I_t^{target-2}, I_t^{target-3}, \dots, I_t^{target-U}$;
5: Get reward $r_t(I_t)$, $t = t + 1$;
6: Until $t > K$;
7: Repeat at the beginning of each time slot
8: Update $N_t(i)$ and $M_{t,2}(i), M_{t,3}(i), \dots, M_{t,U}(i)$ as in (1) and (2);
9: Update $\bar{\mu}_t(i)$ as in (3);
10: Update $\bar{6}_{t,1}^2(i)$
11: Get $\bar{6}_{t,2}^2(i)$, $\bar{6}_{t,3}^2(i)$, … , $\bar{6}_{t,U}^2(i)$ from targets agent;
12: Update $c_t^{OUCB}(i)$ as in (4);
13: Determine $I_t = argmax_{i \in I} \bar{\mu}_t(i) + c_t^{OUCB}(i)$;
14: Try action I_t, observe action $I_t^{target-2}, I_t^{target-3}, \dots, I_t^{target-U}$;
15: Get reward $r_t(I_t)$, $t = t + 1$;
16: Until $t > T$;

مجموعه بازوها یا همان اکشن ها $I := \{1, 2, \cdots, K\}$

$\bar{6}_{t,u}^2(i)$ تخمین واریانس پاداش بازوی i است که بازیکن u به ما میدهد.

معادله های اشاره شده در شبه کد به شرح زیر میباشند:

(1) $N_t(i) := \sum_{s=1}^{t-1} I\{I_s = i\}$,
(2) $M_{t,u}(i) := \sum_{s=1}^{t-1} I\{I_s^{target-u} = i\}$

معادله 1: تعداد دفعاتی که بازو i توسط بازیکن 1 یا Mr. Nobody تا زمان t انجام داده شده است (یادگیرنده)

معادله 2: تعداد دفعاتی که بازو i توسط بازیکن u تا زمان t انجام داده شده است (هدف)

(3) $\bar{\mu}_t(i) := \frac{1}{N_t(i)} \sum_{s=1}^{t-1} r_s\,(i)\mathrm{I}\{\mathrm{I}_s = \mathrm{i}\}$

معادله 5 همان تخمین پاداش یک بازو تا زمان t میباشد.

(4) $c_t^{OUCB}(i) := \sqrt{\frac{2lnt}{N_t(i)}}\,(\beta_1 + \sum_{u=2}^{U} \beta_u\,[\delta_{t,u}(i)]_+)$

معادله 4 از دو قسمت تشکیل شده است. عبارت $\sqrt{\frac{2lnt}{N_t(i)}}$ از الگوریتم $UCB1$ گرفته شده است و یک عبارت جدید در آن ضرب میشود.

(5) $$\beta_v = \frac{e^{\lambda_{t,v}(i)}}{\sum_{u=1}^{U} e^{\lambda_{t,v}(i)}}$$

در معادله 5 خروجی، سافت مکس ترم $\lambda_{t,v}(i)$ میباشد. پس جمع تمام بتا ها برابر با یک میباشد. اینکه بتا باید در چنین شرطی صدق کند برای این است که اهمیت رفتار بعضی از عامل ها بیشتر از باقی آن هاست.

(6) $$\lambda_{t,v}(i) = \frac{1}{\bar{6}^2_{t,v}(i)}$$

معادله 6 از واریانس پاداش ها استفاده میکند. میدانیم که هر چی واریانس پاداش یک بازو برای یک عامل بزرگتر باشد یعنی آن عامل این اکشن را به تعداد زیادی انجام نداده است و کم تجربه است. کم بودن تجربه یعنی باید اهمیت تصمیم این عامل کم شود. در معادله 6 هر چه مخرج همان واریانس است بزرگتر باشد λ کمتر میشود و به این صورت تاثیر تصمیم عامل در معادله 4 کمتر میشود.

توضیح بقیه معادلات در این قسمت اورده نشده است چون مانند سوال قبل میباشند.

ب)

اگر از الگوریتم هایی استفاده کنیم که از میانگین پاداش برای انتخاب اکشن استفاده میکنند روند یادگیری کند خواهد شد. برای مثال اگر از فرمول زیر برای به روزرسانی میانگین پاداش بعد از دریافت یک پاداش جدید استفاده کنیم:

$$meanR = meanR + alpha * (R - meanR)$$

آنگاه تغییرات پاداش دریافتی در طول زمان به گونه ای است که باعث تغییر meanR به صورت نوسانی میشود. در نتیجه انتخاب اکشن هم به صورت نوسانی خواهد بود و باعث کندی همگراهی خواهد شد.

سوال 3 - سوال پیاده سازی

هدف سوال

پیاده سازی الگوریتم Reinforcement Comparison، رسم نمودار پشیمانی و بررسی مقدار مناسب آلفا و بتا در فرمول های الگوریتم.

الف)

توضیح پیاده سازی

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from amalearn.agent import AgentBase
from amalearn.environment import MutliArmedBanditEnvironment
from amalearn.reward import RewardBase
```

در ابتدا کتابخانه های مورد نیاز رو ایمپورت میکنیم.

```
df = pd.read_csv('Q3.csv')
```

داده های مسئله را با کتابخانه پانداس میخوانیم و در دیتافریم df ذخیره میکنیم.

```
def get_value(scores, costs):
    values = [((2 * score) - cost) for score, cost in zip(scores, costs)]
    means = [np.mean(value.to_numpy()) for value in values]
    return values, means
```

این تابع طبق امتیاز و هزینه ورودی معیار ارزشی را طبق فرمول صورت سوال محاسبه میکند.

```
scores = [df['adidas 1'],
          df['adidas 2'],
          df['adidas 3'],
          df['Nike 1'],
          df['Nike 2'],
          df['Nike 3']]
```

```
costs = [df['cost of adidas 1'],
         df['cost of adidas 2'],
         df['cost of adidas 3'],
         df['cost of Nike 1'],
         df['cost of Nike 2'],
         df['cost of Nike 3']]

values, means = get_value(scores, costs)
print("Mean of values: ",means)
```

در این قسمت امتیاز ها و هزینه های هر برند و طرح به تابع `get_value` داده میشود تا ارزش هر کدام محاسبه شود. طبق داده های مسئله این برند ها که هر کدام 3 دارند بازو ها یا اکشن های مسئله هستند.

```
class Reward(RewardBase):
    def __init__(self, value):
        super(Reward, self).__init__()
        self.value = value

    def get_reward(self):
        return self.value.sample().to_numpy()
```

در کلاس پاداش ستون `value` از دیتافریم داده میشود و با هر بار صدا زدن تابع `get_reward` یک نمونه `value` از دیتافریم به عنوان پاداش به عامل داده خواهد شد.

```
rewards = [Reward(value) for value in values]
len(rewards)
```

در این قسمت شی کلاس پاداش مربوط به هر اکشن ساخته میشود. تعداد اکشن ها 6 میباشد پس تعداد شی های کلاس پاداش هم 6 است.

```
def agent_run(agent, env, run, trial):
    mean_reward = np.zeros(trial)
    for run in range(1, run+1):
        for step in range(trial):
            obs, r, d, i = agent.take_action()
            mean_reward[step] = ((run - 1) / run) * mean_reward[step] + ( 1 /
 run) * r

        env.reset()
        agent.reset()
```

```
    return mean_reward
```

در این قسمت مانند سوال اول تمرین تعداد اجرا و استپ را طی میکند و خروجی آن میانگین پاداش کل اجرا ها در هر استپ میباشد.

```
optimal_reward = np.max(means)
optimal_action = np.argmax(means)
print('Optimal reward: ', optimal_reward)
print('Optimal action: ', optimal_action)
```

در لیست مربوط به means میانگین value های متناظر با هر اکشن ذخیره شده است. این مقادیر میانگین های واقعی جامعه آماری هستند و برای به دست آوردن بیشترین مقدار پاداش و پیدا کردن اکشن متناظر آن به ترین از np.max و np.argmax استفاده میکنیم.

```
def plot_mean_reward(mean_reward, optimal_reward):
    step_no = np.arange(len(mean_reward))
    plt.figure(figsize=(20, 12))
    plt.plot(step_no, mean_reward, color='blue', label='Agent policy')
    plt.hlines(y = optimal_reward, xmin = 0, xmax = len(mean_reward), color =
 'r', linestyle = '--', label='Optimal policy')
    plt.xlabel("Steps")
    plt.ylabel("Mean Reward")
    #plt.ylim([lower, upper])
    #new_list = range(int(np.floor(np.min(run_no))), int(np.ceil(np.max(run_n
o))+1))
    #plt.xticks(new_list)
    plt.legend(loc='lower right', fontsize='x-large')
    plt.show();
```

این تابع نمودار میانگین پاداش ها را در طی زمان رسم میکند. همچنین کد plt.hlines میانگین پاداش مربوط به اکشن بهینه را رسم میکند تا بتوان عملکرد عامل را از منظر regert بررسی کرد.

```
def plot_regret(regrets):
    step_no = np.arange(len(regrets))
    plt.figure(figsize=(20, 12))
    plt.plot(step_no, regrets, color='blue')
    plt.xlabel("Steps")
    plt.ylabel("Regret")
    #plt.ylim([lower, upper])
    #new_list = range(int(np.floor(np.min(run_no))), int(np.ceil(np.max(run_n
o))+1))
    #plt.xticks(new_list)
```

```
    plt.show();
```

همچنین میتوان مقدار پشیمانی در هر استپ را نیز رسم کرد و روند پشیمانی را بصورت نزولی مشاهده کرد.

```
class ReinforcementComparisonBanditAgent(AgentBase):
    def __init__(self, id, environment, alpha, beta, greedy):
        super(ReinforcementComparisonBanditAgent, self).__init__(id, environm
ent)
        self.alpha = alpha
        self.beta = beta
        self.greedy = greedy
        self.available_actions = self.environment.available_actions()
        self.p = np.zeros(self.available_actions)
        self.mean_reward = 0




    def take_action(self) -> (object, float, bool, object):
        probabilities = self.softmax_preferences()

        if self.greedy == False:
            action = np.random.choice(self.available_actions, p=probabilities
)
        else:
            action = np.argmax(probabilities)

        obs, r, d, i = self.environment.step(action)
        self.update(r, action)
        #print(obs, r, d, i)
        #self.environment.render()
        return obs, r, d, i

    def softmax_preferences(self):
        probabilities = np.exp(self.p) / np.sum(np.exp(self.p))
        probabilities = np.where(np.isnan(probabilities), 1, probabilities)
        return probabilities

    def reset(self):
        self.p = np.zeros(self.available_actions)
        self.mean_reward = 0


    def update(self, reward, action):
```

```
        self.p[action] = self.p[action] + self.beta * (reward - self.mean_rew
ard)
        self.mean_reward = self.mean_reward + self.alpha * (reward - self.mea
n_reward)
```

در این کلاس الگوریتم Reinforcement Comparison پیاده سازی میشود [4] . این الگوریتم برای اینکه تشخیص دهد پاداش دریافتی ناشی از یک اکشن خوب است یا نه از یک مرجع استفاده میکند (referenced reward). در اینجا این مرجع همان میانگین پاداش تا آن زمان است (self.mean_reward). همچنین این الگوریتم از pereference ها در انتخاب اکشن خود استفاده میکند. پارامتر greedy زمانی استفاده میشود که بخواهیم سیاست الگوریتم حریصانه باشد و همیشه عمل با بالاترین مقدار احتمال را انتخاب کند. به روز رسانی pereference و referenced reward بعد از انجام هر اکشن به صورت زیر است:

```
self.p[action] = self.p[action] + self.beta * (reward - self.mean_reward)

self.mean_reward = self.mean_reward + self.alpha * (reward - self.mean_reward)
```

```
env = MutliArmedBanditEnvironment(rewards, 1000, '1')
rc_agent = ReinforcementComparisonBanditAgent('1', env, 0.9, 0.1, greedy=Fals
e)
mean_reward = agent_run(rc_agent, env, 20, 1000)
```

در این قسمت هم اشیا مربوط به کلاس محیط و عامل ساخته میشوند و میانگین پاداش محاسبه میشود.

نتایج

در این قسمت فقط محاسبه میانگین پاداش ها مدنظر بوده است و نمودار خاصی ندارد.

ب)

توضیح پیاده سازی

```
plot_mean_reward(mean_reward, optimal_reward)
regrets = optimal_reward - mean_reward
plot_regret(regrets)
```

در این قسمت توابع مربوط به رسم نمودار پشیمانی فراخوانی میشوند.

```
regret = np.sum(regrets)
regret
```

این مقدار هم سطح زیر منحنی میانگین پاداش دریافتی تا میانگین پاداش بیشینه را حساب میکند. که مقدار آن در خروجی کد برابر مقدار زیر شد:

```
28162.707385568705
```

```
def mean_reward_diff_param(parameters):
    mr_list = []
    labels = []
    env = MutliArmedBanditEnvironment(rewards, 1000, '2')

    for parameter in parameters:
        rc_agent = ReinforcementComparisonBanditAgent('2', env, parameter[0],
 parameter[1], greedy=False)
        mean_reward = agent_run(rc_agent, env, 20, 1000)
        mr_list.append(mean_reward)
        labels.append('alpha: ' + str(parameter[0]) + ', beta: '+ str(paramet
er[1]))

    return mr_list, labels
```

این تابع لیست میانگین پاداش ها را برای دو پارامتر مختلف آلفا و بتا محاسبه و برمیگرداند. همچنین یک لیست لبیل برای نمایش بهتر نمودار مقدار دهی میشود.

```
def plot_mean_reward_diff_param(mr_list, labels):
    step_no = np.arange(len(mr_list[0]))
    plt.figure(figsize=(20, 12))


    for i, mr in enumerate(mr_list):
        plt.plot(step_no, mr, label=labels[i])

    plt.hlines(y = optimal_reward, xmin = 0, xmax = len(mean_reward), color =
 'c', linestyle = '--', label='Optimal policy')


    plt.xlabel("Steps")
    plt.ylabel("Mean Reward")
    #plt.ylim([lower, upper])
    plt.legend(fontsize='x-large')
    plt.show();
```

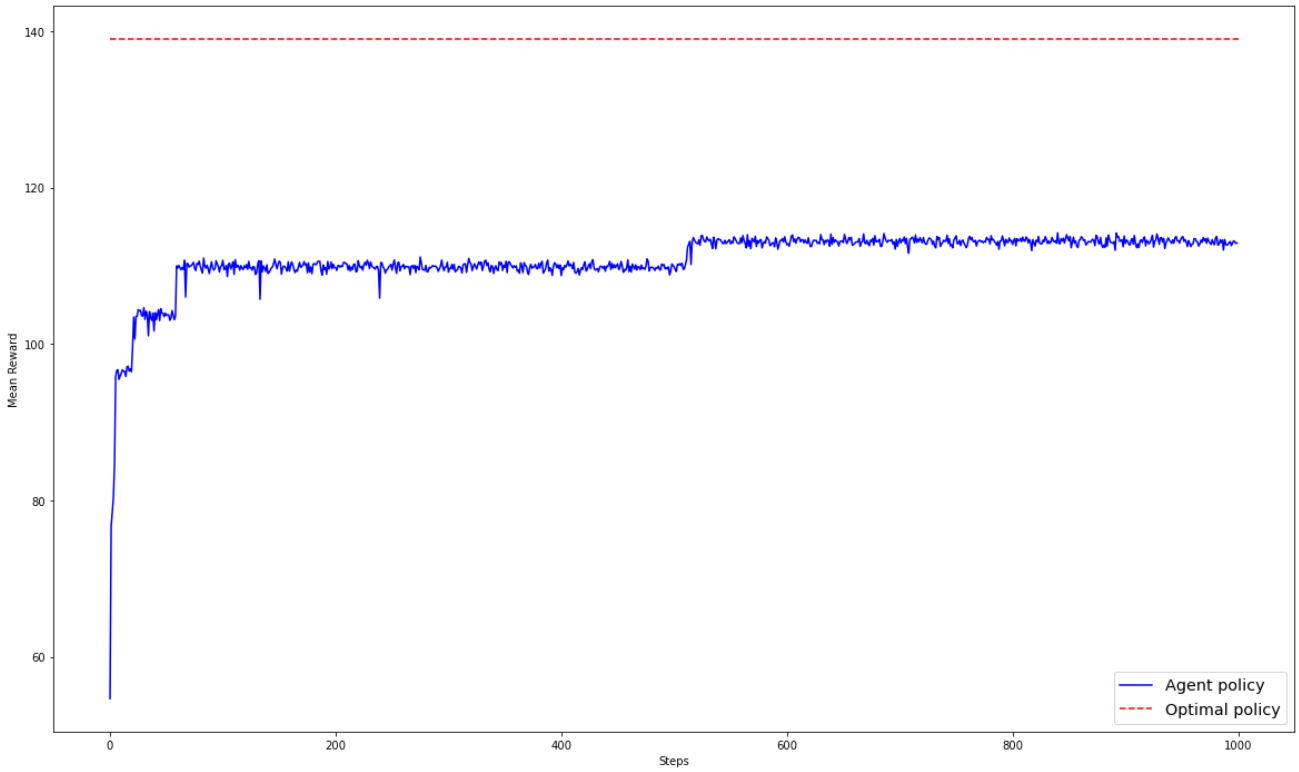
این تابع لیست میانگین پاداش ها با پارامترهای مختلف را گرفته و نمودار مربوطه را رسم میکند. همچنین برای مقایسه خط میانگین پاداش بیشینه رسم میشود.

```
parameters = [[0.1, 0.1],
```

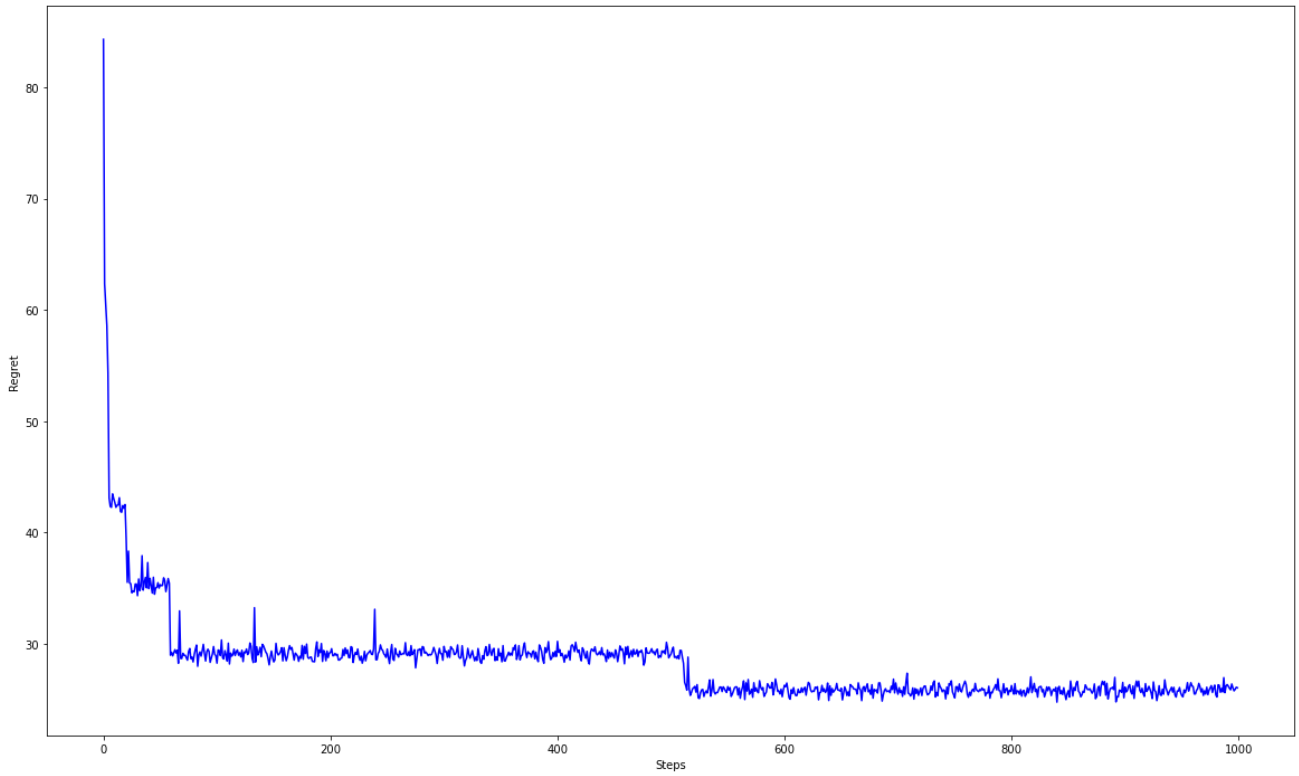
```
                [0.5, 0.5],
                [0.9, 0.9],
                [0.1, 0.9],
                [0.9, 0.1]]

mr_list, labels = mean_reward_diff_param(parameters)
plot_mean_reward_diff_param(mr_list, labels)
```

در متغیر `parameters` انواع مختلف آلفا و بتا مقداردهی شده اند و به تابع های مربوطه پاس داده میشوند.

نتایج

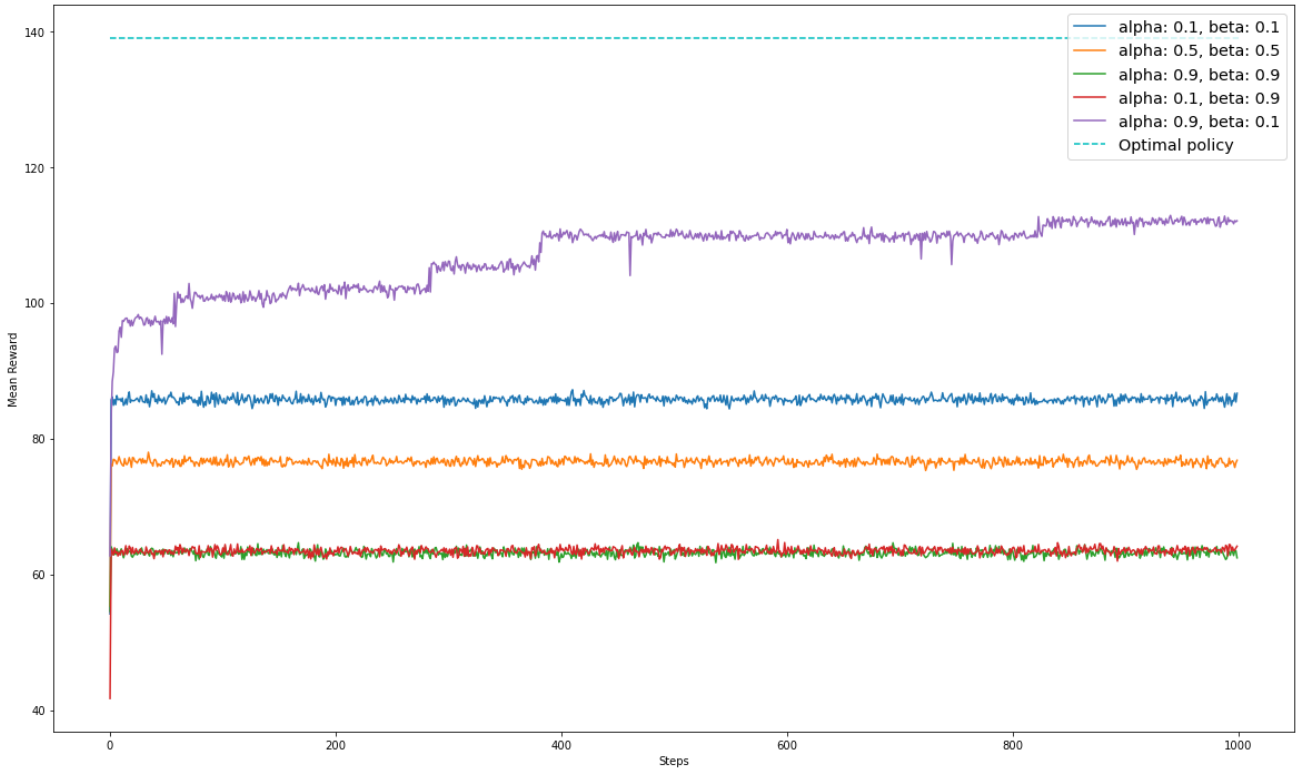


شکل 17 – میانگین پاداش دریافتی و میانگین پاداش بشینه



شکل 18 – نمودار پشیمانی از منظری دیگر

جمع سطح زیر منحنی شکل 18 مقدار پشیمانی را نشان میدهد.



شکل 19 – میانگین پاداش ها با مقادیر متخلف آلفا و بتا

مقایسه مقدار همگرایی با توجه به شکل 19:

alpha: 0.9, beta: 0.1 > alpha: 0.1, beta: 0.1 > alpha: 0.5, beta: 0.5 > alpha: 0.1, beta: 0.9 = alpha: 0.9, beta: 0.9

نکات مشاهده شده از نمودار:

1- به نظر میرسد بدترین نتایج مربوط به زمانی است که بتا مقدار بالایی دارد. (بالا یا پایین بودن مقدار آلفا تاثیری نداشته است)
2- بهترین نتایج مربوط به زمانی است که بتا مقدار مقدار پایینی دارد. (بالا یا پایین بودن آلفا تاثیری نداشته است)
3- در دو حالت alpha: 0.9, beta: 0.1 و alpha: 0.1, beta: 0.1 بتا مقدار پایینی دارد اما زمانی که جمع آن ها برابر 1 شده است بهترین نتیجه حاصل شده است. پس بهتر است آلفا و بتا بهم وابسته باشند.

اگر به لحاظ تئوری هم بررسی کنیم هم این موضوع منطقی میباشد. اگر بتا بزرگ باشد چه اتفاقی می افتد؟ اگر پاداش دریافتی از پاداش مرجع بزرگتر باشد عبارت `self.beta * (reward - self.mean_reward)` با مقدار بتا بزرگ باعث افزایش زیاد `pereference` خواهد شد و بعد از عملیات `softmax` احتمال اکشن متناظر با این پاداش طوری افزایش پیدا میکند که احتمال انتخاب باقی اکشن ها ناچیز خواهد شد. در نتیجه ممکنه است الگوریتم در بهینه محلی گیر بیفتد.

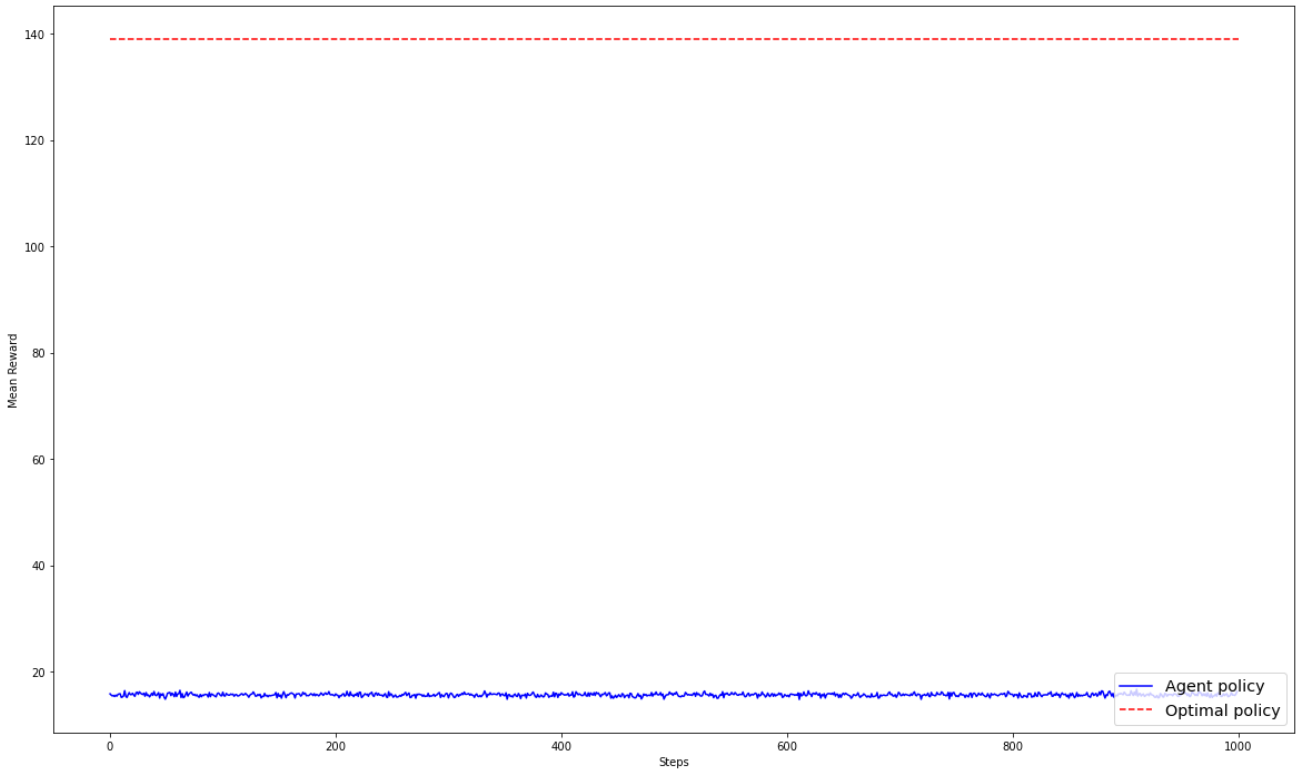
ج)

توضیح پیاده سازی

```
env = MutliArmedBanditEnvironment(rewards, 1000, '3')
rc_agent = ReinforcementComparisonBanditAgent('3', env, 0.9, 0.1, greedy=True
)
mean_reward = agent_run(rc_agent, env, 20, 1000)
plot_mean_reward(mean_reward, optimal_reward)
```

در این قسمت پارامتر greedy را true کرده و یک شی از کلاس ساخته ام تا سیاست حریصانه بررسی شود.

نتایج



شکل 20 – میانگین پاداش با سیاست حریصانه

طبق شکل 20 الگوریتم در بهینه محلی گیر کرده است. اگر اولین اکشن، اکشن بهینه باشد در این صورت الگوریتم به جواب همگرا میشود. در غیر این صورت به جواب همگرا نمیشود.

روند اجرای کد پیاده‌سازی

فایل اجرایی این سوالreinforcement_comparison.ipynb میباشد که با run all از کل نوت بوک خروجی های مورد نظر در هر سلول چاپ میشود.

منابع

[1] A. G. B. Richard S. Sutton, Reinforcement Learning.

[2] "Exponential decay - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Exponential_decay.

[3] "Reinforcement Comparison," [Online]. Available: http://incompleteideas.net/book/ebook/node22.html.

[4] T. L. Z. Z. X. L. H. Q. Jun Zong, "Social Bandit Learning: Strangers Can Help".