دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

گزارش تمرین شماره ۲

درس یادگیری تعاملی

پاییز ۱۴۰۱

نام و نام خانوادگی

مهیار ملکی

شماره دانشجویی

۸۱۰۱۰۰۴۷۶

فهرست

چکیده

هدف این تمرین، بررسی مسائل Multi-Armed Bandit می‌باشد. بدین منظور در بخش اول به سراغ مسائل دنیای واقعی رفته و مدل bandit متناظر با آن‌ها را ارائه می‌کنیم. در این قسمت بازوها، پاداش و نحوه پاسخ‌دهی به مساله را با توجه به حالات مختلف ورودی بررسی خواهیم کرد. همچنین در بخش دوم نیز به پیاده سازی کد یک مسئله Multi-Armed Bandit خواهیم پرداخت.

سوال ۱ – سوال تئوری

هدف سوال

در این بخش با بررسی چند مسئله دنیای واقعی، مدلی مبتنی بر Multi-Armed Bandit برای هر کدام از انها ارائه خواهیم داد. همچنین مجموعه بازوها، پاداش و نحوه پاسخ‌دهی نیز با توجه به حالات مختلف ورودی مشخص خواهندشد.

نتایج

زیر بخش ۱

- بازوها: بازوها در واقع عمل‌هایی هستند که توانایی انجام آنها را داریم. در این سوال در واقع همان ترتیب‌های مختلفی از اجرای حرکات تمرینی می‌باشند که انتخاب آنها به عهده خودمان است. اگر n حرکت متفاوت داشته باشیم، n! ترتیب مختلف را می‌توان متصور شد. به عنوان مثال اگر برنامه‌ای شامل چهار تمرین [A, B, C, D] باشد، بازوهای ما به تعداد جایگشت‌های مختلف این مجموعه (4! = 24) می‌باشد.
- پاداش: در این مسئله، هدف افزایش نرخ سوخت و ساز بدن می‌باشد. میزان انرژی مصرف شده متغیری است که نشان‌دهنده سوخت و ساز بدن بوده و همچنین قابل اندازه‌گیری است، بنابراین میزان انرژی مصرف شده را می‌توان به عنوان پاداش در نظر گرفت.
- حالات مختلف ورودی: در اینجا ما سه برنامه تمرینی مختلف داریم که می‌خواهیم ترتیب بهینه حرکات ورزشی در هر کدام را پیدا کنیم. در نتیجه در این مسئله ما سه حالت متفاوت ورودی داریم که برای هر کدام باید یک عامل یادگیر Multi-Armed Bandit در نظر بگیریم.

نکته: اگر روزهای هفته نیز در نرخ سوخت و ساز تاثیرگزار باشند (یعنی جایگشت y از برنامه x در روزهای مختلف هفته موجب مصرف انرژی متفاوتی شود)، در این صورت تعداد حالات ورودی مسئله افزایش یافته و به عدد ۲۱ (تعداد برنامه‌ها*تعداد روزهای هفته) خواهد رسید.

زیر بخش ۲

- بازوها: در این مسئله، مدت زمان انتظار قبل از تغییر مسیر را می‌توان به عنوان بازو در نظر گرفت. در واقع این که قبل از تغییر مسیر، چقدر پشت چراغ قرمز منتظر بمانیم، عملی می‌باشد که توانایی و اختیار انجام آن را داریم. لازم به ذکر است که بازوهای عمل ما در اینجا، بر خلاف مسئله قبل، متغیری پیوسته می‌باشند و می‌توان آنها را دسته‌بندی کرده و به متغیری گسسته تبدیل کرد. برای مثال می‌توان مدت زمان انتظار را با بازه‌های دو دقیقه‌ای گسسته‌سازی کرد. **نکته:** زمانی که راننده به چراغ می‌رسد باید تصمیم بگیرد که تا چه مدت برای تغییر مسیر منتظر بماند. از آنجایی که در صورت تغییر مسیر ۳۰ دقیقه و در صورت سبز شدن چراغ تنها ۱۰ دقیقه تا مقصد فاصله است، لذا منتظر ماندن بیش از ۲۰ دقیقه منطقی نمی‌باشد چون حتی اگر بعد از بیست دقیقه نیز چراغ سبز شود، باز زمان بیشتری نسبت به اینکه همان ابتدا تغییر مسیر دهد، طول خواهد کشید که به مقصد برسد. لذا بازه زمانی انتظار به صفر تا بیست دقیقه محدود می‌شود.
- پاداش: در اینجا هدف از حل مسئله، رسیدن به دانشگاه در سریع‌ترین زمان ممکن می‌باشد، لذا می‌توان قرینه زمان طی شده برای رسیدن به دانشگاه را به عنوان پاداش در نظر گرفت.
- حالات مختلف ورودی: در این مسئله یک حالت بیشتر نداریم و آن قرمز بودن چراغ راهنمایی می‌باشد (در صورت سبز بودن چراغ، با توجه زمان کمتر مسیر عادی، این حالت را نمی‌توان به عنوان یکی از ورودی‌های مسئله در نظر گرفت زیرا تغییر مسیری صورت نخواهدگرفت) بنابراین برای این مسئله، تنها یک state ورودی داشته و برای آن یک عامل یادگیر Multi-Armed Bandit در نظر می‌گیریم.

زیر بخش ۳

- بازوها: در این مسئله، چهار درگاهی که امکان دریافت و ارسال بسته‌ها از آنها وجود دارد را می‌توان به عنوان بازوهای عمل در نظر گرفت. به عبارت دیگر هر بسته‌ای که به مسیریاب می‌رسد، چه بسته اصلی باشد چه سیگنال تصدیق، می‌تواند از هر کدام از چهار درگاه مسیریاب عبور کند.
- پاداش: در اینجا هدف از حل مسئله، رسیدن بسته‌ها به مقصد و رسیدن تصدیق بسته‌ها به مبدأ، در سریع‌ترین زمان ممکن می‌باشد. یعنی پس از رسیدن بسته‌ها به مسیریاب، باید با

توجه مقصد آن بسته سریع‌ترین درگاه مسیریاب برای عبور بسته انتخاب شود. لذا می‌توان قرینه مدت زمان ارسال را به عنوان پاداش در نظر گرفت.

- حالات مختلف ورودی: در اینجا با توجه به مقاصد بسته‌ها، حالات متفاوتی خواهیم داشت. به عنوان مثال همان طور که در صورت سوال نیز گفته شده است، بسته‌هایی که به این مسیریاب می‌رسند، عموما به مقاصدی در ترکیه، ایران، چین، روسیه و عربستان ارسال شده‌اند، لذا در این مسئله می‌توانیم پنج حالت متفاوت ورودی داشته‌باشیم که برای هر کدام یک عامل یادگیر Multi-Armed Bandit در نظر می‌گیریم.

سوال ۲ - سوال پیاده‌سازی

هدف سوال

در این بخش برای یک مساله دنیای واقعی یک مدل مبتنی بر مساله Multi-Armed Bandit ارائه می‌کنیم. در بخشهای بعدی نیز به پیاده‌سازی و بررسی الگوریتم‌های مختلف حل مسئله Multi-Armed Bandit در شرایط مختلف پرداخته می‌شود.

زیر بخش ۱

- بازوها: در این مسئله، تسهیلاتی که بانک به مشتریان پیشنهاد می‌دهد را می‌توان به عنوان بازوهای عمل در نظر گرفت. با توجه به این که بانک یکی از سه مقدار ۵، ۲۰ و ۱۰۰ میلیون را به عنوان وام ارائه می‌دهد لذا در مسئله ۳ بازوی عمل خواهیم داشت.
- پاداش: در اینجا هدف از حل مسئله، بیشینه کردن سود بانک است، لذا مقدار سود بانک را به عنوان پاداش در نظر می‌گیریم.
- حالات مختلف ورودی: بانک با توجه به توانایی هر مشتری در بازپرداخت وام، تسهیلاتی متفاوت را پیشنهاد می‌دهد. لذا در اینجا ما سه حالت ورودی متفاوت داریم (دانشجویان، کارمندان دولتی، صاحبان مشاغل آزاد) که برای هر کدام یک مدل Multi-Armed Bandit در نظر می‌گیریم.

زیر بخش ۲

توضیح پیاده سازی

همان طور که در کد ۱ مشخص است، ورودی کلاس Enviroment حالت ورودی مدل را مشخص می‌کند و در تابع calc_reward با توجه ورودی انتخاب شده، میزان پاداش محاسبه می‌شود.

```
class Environment:
    def __init__(self, reward, id=None):
        self.reward = reward

    def calc_reward(self, action):
        return self.reward().get_reward(action)[1]

    def get_available_actions(self):
        return np.arange(3)
```

کد ۱- پیاده سازی محیط

زیر بخش ۳

توضیح پیاده سازی

ابتدا به پیاده‌سازی کد کلاس agent می‌پردازیم. در این کلاس توابعی که استفاده از آنها در بین تمامی الگوریتم‌ها ثابت است، قرار می‌گیرند. در ادامه به شرح توابع مهم‌تر این کلاس می‌پردازیم.

- **take_action**

در این تابع با پاس دادن عمل به دست آمده به تابع calc_reward محیط انتخاب شده به محاسبه پاداش آن عمل می‌پردازیم.

```
def take_action(self, action):
        return self.env.calc_reward(action)
```

کد ۲- پیاده‌سازی انجام عمل

- **calculate_utility**

در این تابع با پیاده‌سازی فرمول ارائه شده در صورت سوال، به محاسبه مطلوبیت عمل انتخاب شده با توجه به پاداشی که نتیجه داده است (ورودی) می‌پردازیم.

$$u = \beta r^{\gamma} + \alpha$$

```
def calculate_utility(self, reward):
        return self.beta * reward ** self.gamma + self.alpha
```

کد ۳- پیاده‌سازی تابع محاسبه مطلوبیت

- **update_value**

با توجه به کد ۴ در این تابع با توجه مطلوبیت محاسبه شده در قسمت قبل، آرایه‌های پاداش (q*) را پس از هر آزمایش به صورت incremental بروز می‌کنیم.

n_a_t یک آرایه صفر و یکی با شکل (تعداد عمل*تعداد آزمایش) است که در واقع مشخص می‌کند در هر آزمایش کدام عمل انتخاب شده‌است. Q_mean_total نشان‌دهنده میانگین پاداش محاسبه شده تا آزمایش فعلی و Q_mean_per_action نشان‌دهنده میانگین پاداش محاسبه شده به ازای هر عمل می‌باشد.

```
def update_value(self, action, util):
        for act in range(3):
            if act == action:
                self.n_a_t[act].append(1)
            else:
                self.n_a_t[act].append(0)
        self.Q_mean_total += ( util - self.Q_mean_total ) / self.n
        self.Q_mean_per_action[action] += ( util - self.Q_mean_per_action[action] ) /
                                          self.n_a[action]
        self.n += 1
        self.n_a[action] += 1
```

کد ۴- پیاده‌سازی تابع بروزرسانی مقادیر آرایه‌ها

- **reset**

در این تابع چنان چه در کد ۵ مشخص است، تمامی مقادیر پاداش‌ها و تعداد عمل‌های انجام شده را به حالت اولیه برمی‌گردانیم.

```
def reset(self):
        self.n = 1                              # Step count
        self.n_a = np.ones(3)                   # Step count for each arm
        self.n_a_t = [[], [], []]               # chosen action en each trial
        self.Q_mean_total = 0                   # Total mean reward
        self.utils = []                         # Total utiltiy
        self.regrets = []                       # Calculated regrets
        self.Q_mean_per_action = np.zeros(3)    # Mean reward per action
        self.utils_per_action = [[], [], []]    # Utiltiy per action
```

کد ۵- پیاده‌سازی تابعی برای بازگردانی به مقادیر اولیه

- **get_rewards**

مطابق کد ۶ این تابع، میانگین تجمعی آرایه مطلوبیت‌های بدست آمده را محاسبه می‌کند.

```
def get_rewards(self):
        return np.cumsum(self.utils) / np.arange(1, len(self.utils)+1)
```

کد ۶- پیاده‌سازی تابع بازگردانی پاداش‌ها

در ادامه هر یک از الگوریتم‌های Epsilon Greedy و Gradient Based و Upper Confidence Bound را پیاده‌سازی می‌کنیم.

- **الگوریتم Epsilon Greedy**

این کلاس از دو تابع choose_action و step تشکیل شده و مقدار اپسیلون را به عنوان ورودی دریافت می‌کند. از تابع choose_action برای انتخاب عمل با توجه سیاست فعلی استفاده می‌شود. الگوریتم به این گونه عمل می‌کند که ابتدا یک مقدار تصادفی بین صفر و یک تولید شده، سپس اگر این مقدار از اپسیلون کمتر باشد، عمل بعدی به صورت تصادفی انتخاب می‌شود(exploration)، در غیر این صورت عملی که بیشترین پاداش را داشته به عنوان عمل بعدی انتخاب می‌شود(exploitation). در تابع step نیز با توجه به تعداد آزمایش‌هایی که قصد انجام آن را داریم، تمامی مراحل اجرای الگوریتم را در یک حلقه فراخوانی می‌کنیم.

```
class Epsilon_Greedy(Agent):
    def __init__(self, eps, env):
        super(Epsilon_Greedy, self).__init__(env)
        self.eps = eps    # epsilon

    def choose_action(self):
        p = np.random.random()
        if p < self.eps:
            self.a = np.random.choice(self.actions)
        else:
            self.a = np.argmax(self.Q_mean_per_action)
        return self.a

    def step(self, best_reward, trial):
        self.reset()
        for step in range(trial):
            act = self.choose_action()
            rwrd = self.take_action(act)
            util = self.calculate_utility(rwrd)
            self.update_value(act, util)
            self.utils.append(util)
            self.utils_per_action[act].append(util)
            self.regrets.append(self.n*best_reward - np.sum(self.utils))
```

کد ۷- پیاده‌سازی الگوریتم **epsilon greedy**

- **الگوریتم Gradient Based**

این الگوریتم نیز از دو تابع choose_action و step و همچنین از تابع دیگری به نام update_preferences تشکیل شده است. در تابع choose_action احتمال انتخاب عمل‌ها با

استفاده از تابع سافتمکس و مقدار preference محاسبه می‌شود. سپس با استفاده از مقادیر به دست آمده، یک عمل انتخاب می‌شود. لازم به ذکر است که مقدار اولیه preference را صفر در نظر می‌گیریم تا در ابتدا تمام عمل‌ها شانس انتخاب یکسانی داشته‌باشند. در ادامه در تابع update_preferences با استفاده از پاداش‌های بدست آمده و سرعت آموزش انتخاب شده، مقدار preference را مطابق فرمول ارائه شده در درس بروزرسانی می‌کنیم. در تابع step همانند قبل، تمام مراحل مورد نیاز برای اجرای الگوریتم را به ترتیب در یک حلقه فراخوانی می‌کنیم. در اینجا عملیات بروزرسانی مقادیر preference نسبت به الگوریتم قبل اضافه شده است. لازم به ذکر است که در تابع سافتمکس برای جلوگیری از سرریز عبارات نمایی، بیشینه مقادیر preference را از مقادیر توان‌ها کم می‌کنیم.

```
class Gradient_Based(Agent):
    def __init__(self, lr, env):
        super(Gradient_Based, self).__init__(env)
        self.lr = lr                # learning rate
        self.H = np.zeros(3)        # Initialize preferences

    def choose_action(self):
        # Update probabilities with softmax
        exp = np.exp(self.H - np.max(self.H))
        self.p_per_action = exp / np.sum(exp, axis=0)
        # choose highest preference action
        self.a = np.random.choice(self.actions, p=self.p_per_action)
        return self.a

    def update_preferences(self, action, util):
        for a in range(3):
            if a == action:
                self.H[a] += self.lr * (util - self.Q_mean_total) *
                             (1 - self.p_per_action[a])
            else:
                self.H[a] -= self.lr * (util - self.Q_mean_total) *
                               self.p_per_action[a]

    def step(self, best_reward, trial):
        self.reset()
        self.H = np.zeros(3)
        for step in range(trial):
            act = self.choose_action()
            rwrd = self.take_action(act)
            util = self.calculate_utility(rwrd)
            self.update_value(act, util)
            self.utils.append(util)
            self.utils_per_action[act].append(util)
            self.regrets.append(self.n*best_reward - np.sum(self.utils))
            self.update_preferences(act, util)
```

کد ۸- پیاده‌سازی الگوریتم **gradient based**

- **الگوریتم Upper Confidence Bound**

در این الگوریتم به دلیل عدم قطعیتی که وجود دارد، بازه‌ای را که تخمین پاداش در آن قرار می‌گیرد را محاسبه می‌کنیم. با کاهش یافتن این بازه، عدم قطعیت نسبت به پاداش محاسبه شده کاهش یافته و نشان‌دهنده این است که تخمین ما به مقدار واقعی نزدیکتر شده است.

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

این کلاس نیز مانند دو الگوریتم قبلی از دو تابع choose_action و step تشکیل شده‌است. در تابع choose_action با پیاده‌سازی فرمول ذکر شده، به ازای هر عمل بازه تخمین را محاسبه می‌کنیم، سپس عملی که مقدار بیشتری داشته‌باشد را انتخاب می‌کنیم. در تابع step نیز همانند قبل، تمام مراحل مورد نیاز برای اجرای الگوریتم را به ترتیب در یک حلقه فراخوانی می‌کنیم.

```
class UCB(Agent):
    def __init__(self, c, env):
        super(UCB, self).__init__(env)
        self.c = c   # exploration factor

    def choose_action(self):
        self.a = np.argmax(self.Q_mean_per_action +
                 self.c * np.sqrt(np.log(self.n) / self.n_a)
        return self.a

    def step(self, best_reward, trial):
        self.reset()
        for step in range(trial):
            act = self.choose_action()
            rwrd = self.take_action(act)
            util = self.calculate_utility(rwrd)
            self.update_value(act, util)
            self.utils.append(util)
            self.utils_per_action[act].append(util)
            self.regrets.append(self.n*best_reward - np.sum(self.utils))
```

کد ۹- پیاده‌سازی الگوریتم **UCB**

همچنین تابعی را نیز برای اجرای چند باره الگوریتم‌ها و همچنین برگردانی مقادیر regret و reward پیاده‌سازی می‌کنیم. در این تابع به دو روش به محاسبه مقدار regret می‌پردازیم.

- در روش اول با استفاده از فرمول زیر، در هر بار اجرا مقدار regret را محاسبه می‌کنیم (این فرمول در تابع step هر الگوریتم پیاده‌سازی شده‌است). سپس بین مقادیر به دست آمده در اجراهای مختلف میانگین می‌گیریم.

$$Regret_{run\,j,\ h} = hq^*(a^*) - \sum_{i=1}^{h} r_{i,j}$$

- در روش دوم که در کد ۱۰ قابل مشاهده است، با استفاده از فرمول زیر به محاسبه میانگین regret تمام اجراها می‌پردازیم. در این روش چون از مقدار expected پاداش هر عمل استفاده کرده و آن را از مقدار پاداش اکشن بهینه کم می‌کنیم، لذا تمام مقادیر مثبت خواهندشد و در نتیجه نمودار حاصل اکیدا صعودی می‌شود.

$$Regret_h = \sum_{i=1}^{h} \Delta_i \mathbb{E}[T_i]$$

```
def Run(run, trial, enviroment, eps, lr, c):
    rewards = np.zeros([3, run, trial])
    regrets1 = np.zeros([3, run, trial])
    regrets2 = np.zeros([3, trial])
    Ti = np.zeros([run, 3, trial])
    best_reward = max(enviroment().expected_rewards())
    delta = np.expand_dims(best_reward -
                           np.array(enviroment().expected_rewards()), 1)

    for p, policy in enumerate([Epsilon_Greedy(eps ,enviroment),
                                Gradient_Based(lr ,enviroment),
                                UCB(c ,enviroment)]):
        for r in range(run):
            Pi = policy
            Pi.step(best_reward, trial)
            rewards[p,r,:] = Pi.get_rewards()
            regrets1[p,r,:] = Pi.get_regrets()
            Ti[r,:,:] = np.cumsum(np.array(Pi.n_a_t), axis=1)
        regrets2[p,:] = np.sum(np.mean(Ti, axis=0) * delta, 0)

    rewards = np.mean(rewards, 1)
    regrets1 = np.mean(regrets1, 1)
    return rewards, regrets2, regrets1
```

کد ۱۰- پیاده‌سازی تابع اجرای چند باره الگوریتم‌ها و برگردانی مقادیر **regret** و **reward**

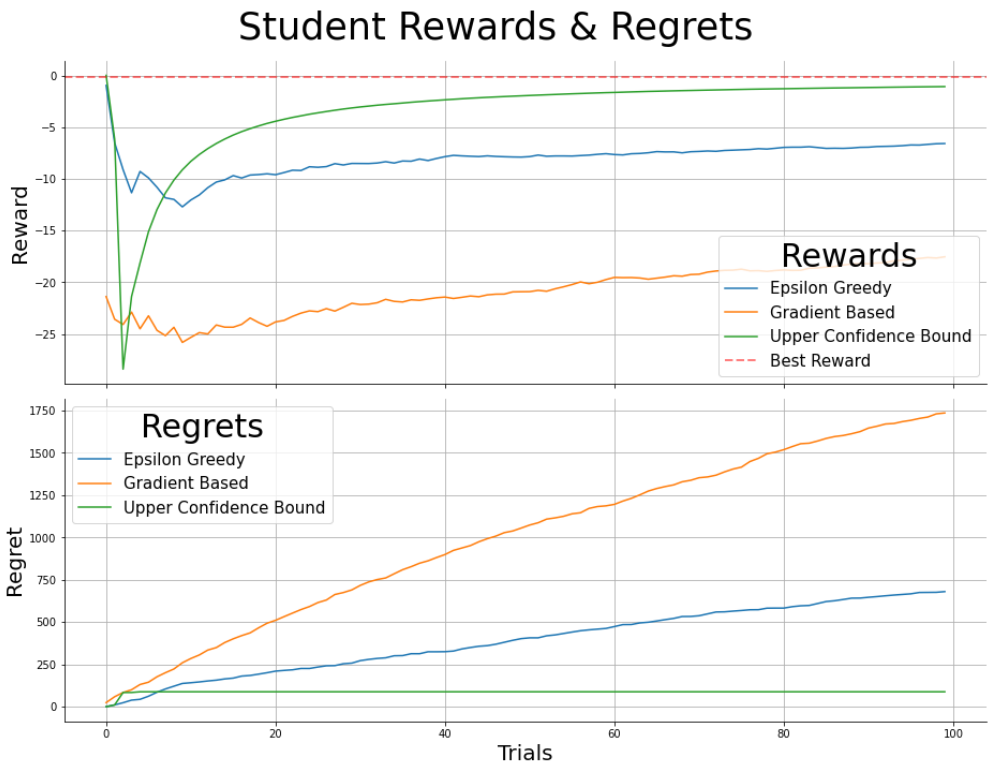
زیر بخش ۴

نتایج

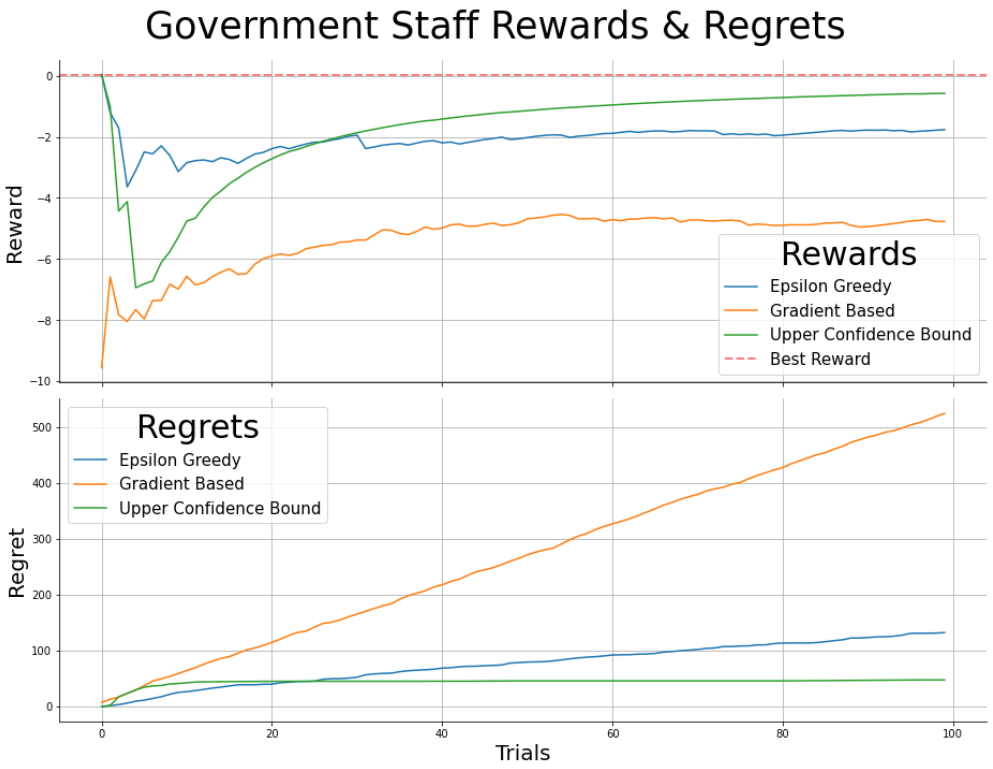
همانطور که در نمودار پاداش شکل ۱ قابل مشاهده است، در صورتی که حالت ورودی مسئله، پرداخت تسهیلات به دانشجویان باشد، با پارامترهایی که صورت سوال پیشنهاد داده‌است، الگوریتم upper confidence bound عملکرد بهتری داشته و با سرعت بیشتری به پاداشی که در صورت انتخاب عمل بهینه به آن خواهیم رسید، همگرا می‌شود. این نتایج از نمودار regret نیز قابل برداشت است. همچنین بدترین عملکرد نیز به الگوریتم gradient based تعلق دارد که این مشاهده می‌تواند ناشی از مقدار کوچک پارامتر سرعت آموزش (۰.۰۰۱) باشد. این نتایج را در نمودار حالات ورودی دیگر نیز می‌توان مشاهده کرد.

همچنین چنان چه در شکل ۳ قابل مشاهده است، در نمودار حالت ورودی مشاغل آزاد نوسان زیادی مشاهده می‌شود. این می‌تواند ناشی از نزدیک بودن میانگین و توزیع پاداش عمل‌ها به هم باشد که باعث می‌شود عمل بهینه به خوبی انتخاب نشده و مدام بین عمل‌ها تعویض انجام شود.

نکته دیگری که می‌توان به آن اشاره کرد، این است که مقدار regret الگوریتم ucb به دلیل وجود ترم $\ln(t)$ به صورت لگاریتمی افزایش می‌یابد.



شکل ۱- نمودار پاداش و پشیمانی سه الگوریتم برای حالت ورودی دانشجویان



شکل ۲- نمودار پاداش و پشیمانی سه الگوریتم برای حالت ورودی کارمندان دولت

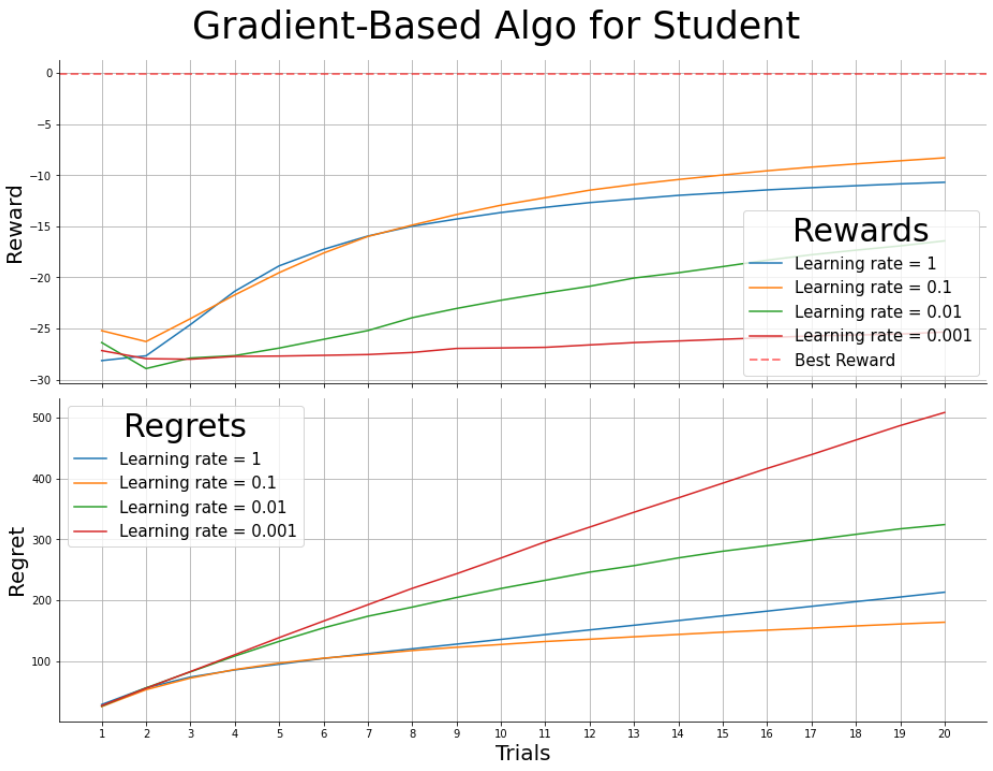
شکل ۳- نمودار پاداش و پشیمانی سه الگوریتم برای حالت ورودی مشاغل آزاد

زیر بخش ۵

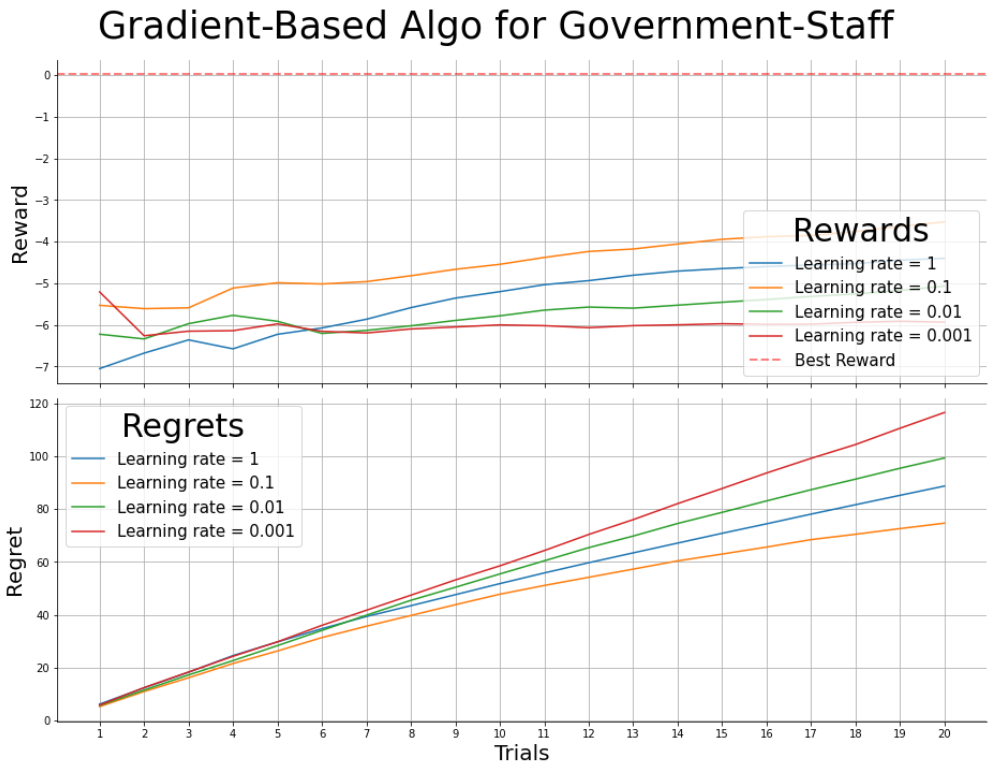
نتایج

الگوریتم gradient based را با سرعت آموزش‌های ۱ و ۰.۱ و ۰.۰۱ و ۰.۰۰۱ اجرا می‌کنیم. همان طور که در شکل‌های ۴ تا ۶ قابل مشاهده است، در بیست آزمایش اول سرعت آموزش ۰.۱ بهترین عملکرد را داشته، یعنی پاداش آن به مقدار پاداش بهینه نزدیکتر شده و regret کمتری نیز دارد. البته مقایسه الگوریتم‌ها در تعداد آزمایش محدود خیلی منطقی نیست، زیرا ممکن است الگوریتمی در این تعداد آزمایش پاداش بهتری را نتیجه دهد ولی در ادامه وارد بهینه محلی شده و به پاداش عمل بهینه همگرا نشود.

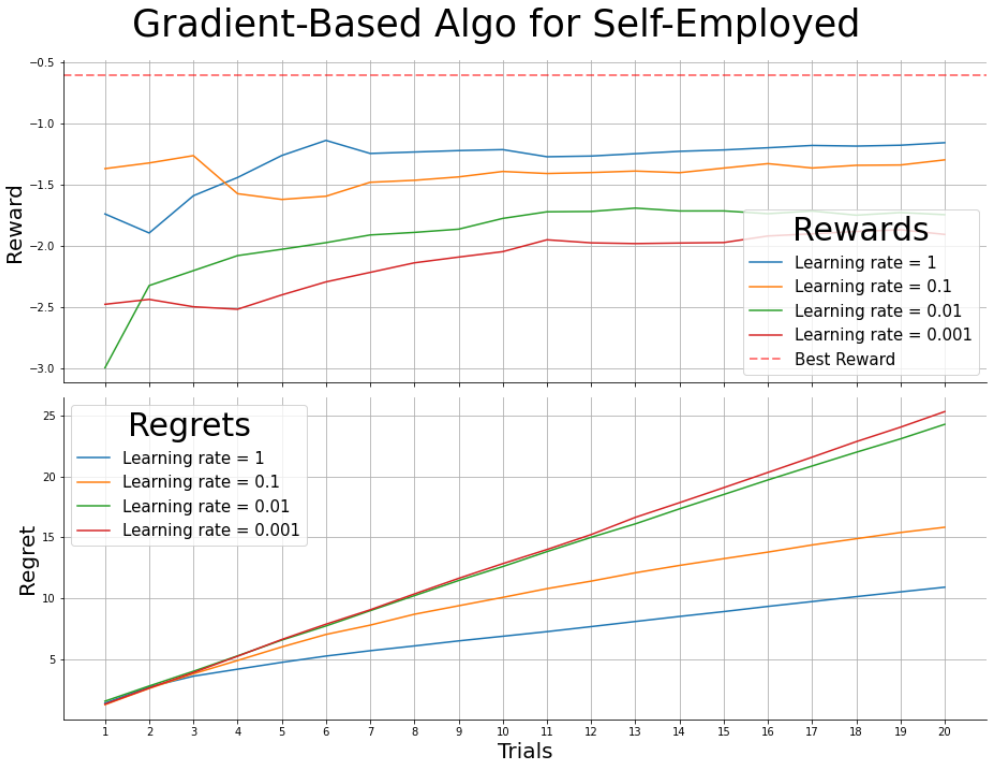
لازم به ذکر است که به دلیل عدم قطعیت پاداش عمل‌ها، هر بار اجرای کدها نتایج متفاوتی را حاصل می‌شد، لذا تعداد اجراها را از ۲۰ به ۵۰۰ افزایش داده تا حتی‌الامکان این عدم قطعیت را برطرف کرده و به نتایج پایدارتری برسیم.



شکل ۴- نمودار پاداش و پشیمانی الگوریتم **gradient** با سرعت آموزش‌های مختلف برای حالت ورودی دانشجویان



شکل ۵- نمودار پاداش و پشیمانی الگوریتم **gradient** با سرعت آموزش‌های مختلف برای حالت ورودی کارمندان دولت



شکل ۶- نمودار پاداش و پشیمانی الگوریتم **gradient** با سرعت آموزش‌های مختلف برای حالت ورودی مشاغل آزاد

روند اجرای کد پیاده‌سازی

تمام کدها و پیاده سازی‌ها در فایل < HW2_Maleki_810100476.ipynb > قرار دارد. تنها لازم است که تمام سلول‌ها به ترتیب از ابتدا اجرا شوند تا نتایج و نمودارها به دست آیند.

منابع

[1] https://www.datahubbs.com/multi-armed-bandits-reinforcement-learning-2/

[2] https://medium.com/@isurualagiyawanna/step-up-into-artificial-intelligence-and-reinforcement-learning-solving-the-multi-armed-bandit-d82b8b28544a

[3] https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/