



دانشگاه تهران  
پردیس دانشکده‌های فنی  
دانشکده برق و کامپیوتر

گزارش تمرین شماره ۴  
درس یادگیری تعاملی  
پاییز ۱۴۰۰

نام و نام خانوادگی	سحر رجبی
شماره دانشجویی	۸۱۰۱۹۹۱۶۵

## فهرست

۴	چکیده
۵	سوال ۱ - بررسی روش‌های model-based
۵	هدف سوال
۵	توضیح پیاده‌سازی
۷	نتایج
۸	روند اجرای کد پیاده‌سازی
۹	سوال ۲ - بررسی روش‌های model-free
۹	هدف سوال
۹	توضیح پیاده‌سازی
۱۲	نتایج
۱۲	الگوریتم : off-policy MC
۱۲	توضیح پیاده‌سازی:
۱۳	نتایج:
۱۳	ابتدا سیاست یافته شده توسط هر یک از دو اجرای با اپسیلون ثابت و اپسیلون کاہشی را بررسی می‌کنیم.
۱۴	همانطور که در دو شکل بالا مشاهده می‌شود ایجنت در هیچ‌کدام به بهترین سیاست دست نیافته است و هنوز اپیزودهای بیشتری برای انجام این کار نیاز دارد. اما برای مقایسه عمل کرد این دو، ابتدا نمودار مجموع ریوارد دریافتی در هر اپیزود را مقایسه می‌کنیم.
۱۵	الگوریتم : q-leaning
۱۵	توضیح پیاده‌سازی:
۱۶	نتایج:
۱۸	الگوریتم : Sarsa
۱۸	توضیح پیاده‌سازی:
۱۹	نتایج:
۱۹	الگوریتم : n-step tree backup
۱۹	توضیح پیاده‌سازی:
۲۰	نتایج:
۲۲	روند اجرای کد پیاده‌سازی
۲۳	سوال ۳ - ترکیب روش‌های model-free و model-based

۲۳	هدف سوال
۲۳	توضیح پیاده سازی
۲۳	نتایج
۲۴	روند اجرای کد پیاده سازی
۲۴	برای اجرای کد پیاده سازی کافی است فایل MBPlusMF.ipynb را اجرا کنید
۲۵	سوال ۴ - ترکیب اطلاعات ناقص محیط با Model-free
۲۵	هدف سوال
۲۵	توضیح پیاده سازی
۲۶	نتایج
۲۶	روند اجرای کد پیاده سازی
۲۶	برای اجرای این بخش، فایل MLearning.ipynb را اجرا کنید

## چکیده

---

در این تمرین مباحث زیادی پوشش داده شده‌اند که به صورت کلی شامل مقایسه‌ی روش‌های model-free و همچنین روش‌های on-policy و off-policy بوده‌اند. به طور کلی روند پیشرفت تمرین به این صورت است که ابتدا با فرض شناخت کامل محیط سعی در یافتن بهترین راه با استفاده از این مساله دارد. سپس فرض می‌کنیم که شناختی از محیط نداریم و باید با استفاده از روش‌های model-free به حل مساله پردازیم که در این بین حجم زیادی از مسائل مربوط به مقایسه‌ی کارآمدی این روش‌ها و رفع نقص هر کدام به وسیله‌ی روش‌های دیگر است. در نهایت اما با نگاهی واقع‌بینانه‌تر، با توجه به اینکه ما ممکن است اطلاعات اولیه‌ای از محیط داشته‌باشیم و می‌توانیم از آن‌ها برای حل مساله استفاده کنیم، به ترکیب این روش‌ها با یکدیگر می‌پردازیم.

## سوال ۱ - بررسی روش‌های model-based

### هدف سوال

در این بخش با استفاده از الگوریتم value-iteration که یک الگوریتم model-based است سعی می‌کنیم که سیاست بهینه را با توجه به دانش‌های مساله، که در این بخش ایجنت هم از آن‌ها آگاه است، حل کنیم. این بخش درواقع نقطه‌ی شروع برای مقایسه و درک بهتر روش‌های model-free و بررسی تأثیر نداشتن اطلاعات کامل است.

### توضیح پیاده‌سازی

برای پیاده‌سازی این بخش، یک کلاس Agent که از کلاس AgentBase در پکیج AMALearn ارث بری کرده است پیاده‌سازی شده. در این کلاس متدهای مختلفی وجود دارد که به ترتیب توضیح خواهیم داد. اما از مهم‌ترین فیلدهای این کلاس، می‌توان به value استیت‌ها، مقدار q-value استیت و اکشن‌ها اشاره کرد.

این الگوریتم دقیقاً مشابه سودوکدی که در کتاب ساتن آورده شده است پیاده‌سازی شده و این سودوکد به صورت زیر است.

```
Value Iteration, for estimating  $\pi \approx \pi_*$ 

Algorithm parameter: a small threshold  $\theta > 0$  determining accuracy of estimation
Initialize  $V(s)$ , for all  $s \in \mathcal{S}^+$ , arbitrarily except that  $V(\text{terminal}) = 0$ 

Loop:
|    $\Delta \leftarrow 0$ 
|   Loop for each  $s \in \mathcal{S}$ :
|      $v \leftarrow V(s)$ 
|      $V(s) \leftarrow \max_a \sum_{s', r} p(s', r | s, a)[r + \gamma V(s')]$ 
|      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
until  $\Delta < \theta$ 

Output a deterministic policy,  $\pi \approx \pi_*$ , such that

$$\pi(s) = \arg \max_a \sum_{s', r} p(s', r | s, a)[r + \gamma V(s')]$$

```

شکل ۱- سودوکد پیاده‌سازی شده‌ی الگوریتم value-iteration

در ابتدا در این الگوریتم باید مقادیر  $V$  را مقداردهی اولیه کنیم که به صورت رندم انتخاب خواهند شد و استیت نهایی، که برای ما همان استیت هدف است، مقدار اولیه‌ی صفر خواهد داشت. هرچند ما در این سؤال به دنبال بدست‌آوردن مقادیر q-value هم هستیم پس در تابع زیر هر دوی  $V$  و  $Q$  را مقداردهی می‌کنیم.

```
def init_V_Q(self):
    for i in range(self.i_limit):
        for j in range(self.j_limit):
            if (i, j) == (3, 3):
                self.V[(i, j)] = 0
            else:
                self.V[(i, j)] = np.random.rand()
            for a in self.actions:
                self.Q[((i, j), a)] = 0
```

شکل ۲- کد مقداردهی اولیه‌ی متغیرهای الگوریتم value-iteration

مطابق سودوکد آورده شده ما باید تا زمانی که دلتا، که در هر مرحله برابر با ماسکسیم اختلاف بین ارزش جدید یک استیت و ارزش آن در مرحله قبلی است (به ازای استیت‌های مختلف)، کمتر از یک مقدار تنا شود الگوریتم را ادامه دهیم و در هر مرحله، ارزش استیت را برابر با ماسکسیم مقداری قرار بدھیم که با انتخاب اکشن‌های مختلفاً احتمالاً بدست خواهیم آورد. کد این بخش به صورت زیر است:

```
def value_iteration(self):
    epoch = 0
    while True:
        delta = 0
        epoch += 1
        for state, v in self.V.items():

            temp_v = v
            max_v, max_act = 0, -1
            for act in ACTIONS:
                new_v = self.calculate_v(state, act)
                self.Q[(state, act)] = new_v
                if new_v > max_v:
                    max_v = new_v
                    max_act = act

            self.V[state] = max_v

            old_act = np.argmax(self.policy[state])
            self.policy[state][old_act] = 0
            self.policy[state][max_act] = 1

            delta = max(delta, abs(temp_v - max_v))

        if delta <= self.theta:
            print(epoch)
            break
```

شکل ۳- کد پیاده‌سازی الگوریتم value-iteration با سودوکد شکل ۱

قسمت اصلی الگوریتم در حلقه‌ای که بر روی اعمال مختلف برای هر استیت انجام می‌شود قرار دارد. در این بخش کد ما به ازای هر استیت و اکشن، مقدار  $v$  که همان برگشت احتمالی ماست را محاسبه می‌کنیم. تابع  $\text{calculate}_v$  که کد این بخش را در بر دارد در ادامه آورده شده است.

```
def calculate_v(self, state, action):
    states, probs, fail_probs, dones = self.environment.possible_sequences(action, state)
    new_v = 0
    for next_state, prob, fail_prob, done in zip(states, probs, fail_probs, dones):
        new_v += prob*fail_prob*(FAIL_REWARD+MOVE_REWARD)
        new_v += prob*(1-fail_prob)*(MOVE_REWARD + self.discount*self.V[next_state])
        if done:
            new_v += prob*(1-fail_prob)*(GOAL_REWARD)

    return new_v
```

شکل ۴- کد محاسبه‌ی ارزش انتخاب یک عمل در یک استیت

در این تابع، ما با استفاده از تابع  $\text{possible\_sequences}$  که در کلاس  $\text{environment}$  پیاده‌سازی شده است، تمام استیت‌هایی که ممکن است با اعمال عمل انتخابی، از استیت فعلی به آن‌ها منتقل شویم را به همراه احتمال انتقال به آن‌ها، احتمال سقوط در آن خانه‌ها و اینکه آیا خانه‌ی نهایی هستند یا خیر دریافت می‌کنیم. سپس باید مقدار  $\text{return}$  را در صورتی که این عمل را انجام دهیم با توجه به این اطلاعات به‌دست بیاوریم.

برای هر استیت، ما یک احتمال انتقال به آن استیت را داریم ( $\text{prob}$ ) و یک احتمال سقوط ( $\text{fail\_prob}$ ) با رفتن به آن خانه. پس احتمال رفتن به استیت مدنظر و سقوط در آن، برابر با  $\text{prob} * \text{fail\_prob}$  خواهد بود. در این صورت ما به اندازه ۱۰٪ برای حرکت و ۹۰٪ برای سقوط هزینه

خواهیم داد و بعد از آن اپیزود به پایان می‌رسد و دیگر نیازی به در نظر گرفتن بخش دوم فرمول یعنی  $\text{discount}^*v(s)$  نخواهد بود.

اما با احتمال یک منهای  $\text{fail\_prob}$  در آن خانه سقوط نخواهیم کرد. در نتیجه خط دوم این حلقه، ما احتمال این اتفاق را برابر با  $(1-\text{fail\_prob})^*$  در نظر گرفتیم و پاداشی برابر منفی یک دریافت خواهیم کرد. سپس آن را با مقدار  $v(s)$  جمع می‌کنیم تا تخمینی از  $\text{return}$  این رفتار را داشته باشیم. این بخش حتی در صورتی که ما به خانه‌ی هدف رسیده باشیم هم لازم است چرا که  $v(\text{terminal})$  برابر با صفر است و حتی برای رسیدن به خانه‌ی هدف هم باید هزینه‌ی حرکت یعنی  $-1$  را بپردازیم. اما اگر به خانه‌ی نهایی رسیده باشیم، به این معناست که باید  $50$  امتیاز رسیدن به این خانه را هم در نظر بگیریم که در if ما با همان احتمال  $(\text{prob}^*(1-\text{fail\_prob})$  به این خانه می‌رسیم و در آن سقوط نمی‌کنیم و پاداش نهایی را به دست می‌آوریم.

همچنین در این کد، مقدار policy را هم به روزسانی می‌کنیم و آن را به یک greedy policy که در آن احتمال انتخاب عمل با بیشترین ولیو  $1$  و بقیه  $0$  هستند.

همچنین باید اشاره کرد که ما در این الگوریتم، بزرگترین مقدار را که به ازای یک عمل خاص به دست خواهد آمد به عنوان ارزش یک استیت در نظر می‌گیریم، این مقادیر که به ازای هر عمل محاسبه می‌کنیم (خروجی تابع calculate\_v که در قبل توضیح داده شد) همان مقادیر  $q$  هستند، درواقع یکی از  $q$ ها که مقدار ماکسیمم در بین آنهاست به عنوان ارزش استیت انتخاب می‌شود. به این منظور، ما بعد از محاسبه‌ی این مقدار برای هر اکشن، آن را به عنوان  $Q(s, a)$  ذخیره می‌کنیم تا به این منظور بتوانیم  $q$ -value‌ها را هم در انتهای داشته باشیم.

## نتایج

نقشه و سیاست بهینه هر دو در شکل زیر قابل مشاهده هستند (به جای نمایش عدد هر عمل، جهت بهینه‌ی حرکت را برای ایجنت نمایش دادیم) و همانطور که مشخص است سیاست نهایی سیاستی است که ایجنت را در کم‌خطرتین مسیر هدایت کرده است و هزینه حرکت را هم حداقل نگهداشته است.

<b>0.000</b>	0.001	0.332	0.746
0.696	0.001	0.143	0.998
0.703	0.001	0.001	0.001
0.861	0.401	0.128	0.000

→	↓	↓	↑
→	↓	↓	↓
→	↑	↑	↓
→	↑	↑	↑

شکل ۵- سیاست بهینه برای محیط بالا، پیدا شده توسط الگوریتم value-iteration

همچنین مقادیر Q-value در اجرای فایل ValueIteration.ipynb قابل مشاهده است. بخشی از این مقادیر را در تصویر زیر هم آورده‌ایم و همانطور که مشخص است،  $q$ -value مربوط به عمل بهینه در نهایت از تمامی اعمال دیگر بزرگ‌تر است و محاسبات به درستی انجام شده است.

```

===== state (0, 0) =====
state: (0, 0), action: 0, Q-value: 194.4794221221824
state: (0, 0), action: 1, Q-value: 70.4352357425524
state: (0, 0), action: 2, Q-value: 220.23301811078974
state: (0, 0), action: 3, Q-value: 194.4794221221824
===== state (0, 1) =====
state: (0, 1), action: 0, Q-value: 198.02645484626086
state: (0, 1), action: 1, Q-value: 252.29098315395277
state: (0, 1), action: 2, Q-value: 147.41091822576334
state: (0, 1), action: 3, Q-value: 223.7800508348682
===== state (0, 2) =====
state: (0, 2), action: 0, Q-value: 219.216594896086
state: (0, 2), action: 1, Q-value: 241.58331939898954
state: (0, 2), action: 2, Q-value: 35.32279292750111
state: (0, 2), action: 3, Q-value: 142.84746228698117
===== state (0, 3) =====
state: (0, 3), action: 0, Q-value: 131.21772998055948
state: (0, 3), action: 1, Q-value: -4.822973970027828
state: (0, 3), action: 2, Q-value: 23.693060621079425
state: (0, 3), action: 3, Q-value: 23.693060621079425
===== state (1, 0) =====
state: (1, 0), action: 0, Q-value: 67.94898508737158
state: (1, 0), action: 1, Q-value: 73.97721556140814
state: (1, 0), action: 2, Q-value: 246.2576997746935
state: (1, 0), action: 3, Q-value: 191.9931714670016
===== state (1, 1) =====
state: (1, 1), action: 0, Q-value: 74.25052756186281
state: (1, 1), action: 1, Q-value: 287.524898945157
state: (1, 1), action: 2, Q-value: 246.4150344330037
state: (1, 1), action: 3, Q-value: 224.04830993010017
===== state (1, 2) =====
state: (1, 2), action: 0, Q-value: 250.06424881052598
state: (1, 2), action: 1, Q-value: 328.04712069916053
state: (1, 2), action: 2, Q-value: 9.143479931749214
state: (1, 2), action: 3, Q-value: 145.18418388233655

```

شکل ۶- مقادیر به دست آمده برای Q-value های چند استیت-اکشن توسط الگوریتم value-iteration

## روند اجرای کد پیاده‌سازی

برای اجرای کد پیاده‌سازی هم کافی است فایل ValueIteration.ipynb را اجرا کنید و روند ویژه‌ای برای این بخش مورد نیاز نیست.

## سوال ۲ - بررسی روش‌های model-free

### هدف سوال

در قسمت قبل، با استفاده از روش‌های model-based سعی بر حل مساله داشتیم. اما در صورتی که ایجنت هیچ اطلاعی از محیط نداشته باشد هم باید بتواند مساله را حل کند. در این بخش روش‌های مختلفی که به صورت model-free سعی در حل مساله دارند پیاده‌سازی می‌شود تا با یکدیگر مقایسه شده و اثر تغییرات پارامترهای و یا روش‌هایی که برای تخمین استفاده می‌شوند بررسی شود.

### توضیح پیاده‌سازی

در این بخش به پیاده‌سازی کلی‌ای که در قسمت‌های مختلف این سؤال مشترک هستند می‌پردازیم و در ادامه در هر زیر بخش، پیاده‌سازی الگوریتم مربوطه را هم توضیح خواهیم داد.

هر کدام از الگوریتم‌ها همانطور که در صورت سؤال خواسته شده است باید برای ۲۰ بار اجرا به تعداد اپیزود خواسته شده، میانگین گرفته شوند. و هر بار باید دانش ایجنت ما از محیط به حالت اولیه برگرد و در صورتی که نرخ یادگیری و یا اپسیلون در طول یادگیری کاهش یافته‌اند به مقدار اولیه خود برگردند. به این منظور در هر کلاس agent ما یک متده است reset که تمامی policy‌ها، q-value‌ها و دانش‌های مربوط به هر agent را به حالت اولیه خود بر می‌گرداند و در اول هر تکرار صدای خواهد شد.

برای تعاریف پارامترهای مربوط به هر مساله مثل مقدار دهی اولیه epsilon یا learning rate و غیره، یک فایل CONFIG.py داریم که تمامی مقادیر در آن قرار گرفته‌اند. از جمله سیاست بهینه! که با توجه به سیاست بهینه در سؤال اول در این فایل قرار گرفته است و عمل بهینه در هر استیت را مشخص کرده است.

در تمامی بخش‌ها ما باید نمودار رشد regret را رسم کنیم که برای این کار نیاز بود تا پاداش مسیر بهینه را بدانیم. از آنجایی که مسیر ما دارای احتمال سقوط و یا لیز خوردن است، ما با میانگین گرفتن بر روی ۱۰۰۰ تکرار اجرای پالیسی بهینه (که در پاراگراف قبل اشاره شد) میانگین پاداش را بدست آورده‌یم. پیاده‌سازی این بخش در تابع find\_max\_reward در فایل find\_max\_reward.py است که برای ۱۰۰۰ تکرار، هر بار از خانه اولیه شروع می‌کند و اعمال سیاست بهینه را در هر استیت انتخاب می‌کنه و پاداش و استیت بعدی را از تابع step در environment دریافت می‌کنه تا زمانی که به پایان برسد (در هر تکرار این پایان می‌تواند سقوط در یک خانه و یا رسیدن به خانه هدف باشد) سپس بین ریوارد دریافتی در این ۱۰۰۰ تکرار میانگین گرفته می‌شود. کد این بخش در تصویر زیر قابل مشاهده است.

```
def find_max_reward(env):
    episode_rewards = []
    for e in range(1000):
        rewards = []
        state = env.reset()
        while True:
            next_state, reward, done, _ = env.step(BEST_POLICY[state])
            rewards.append(reward)
            if done:
                break
            state = next_state
        episode_rewards.append(sum(rewards))
    return np.mean(episode_rewards)
```

شکل ۷- کد پیداکردن امیدریاضی ماقسیم ریوارد با دنبال کردن سیاست بهینه

برای رسم نمودار `regret` هم ما مجموع پاداش‌های دریافتی در هر یک از اپیزودها را دریافت می‌کنیم و سپس بین تمامی تکرارها برای این اپیزودها میانگین می‌گیریم. در نهایت اختلاف ماکسیمم ریواردی که ایجنت می‌توانست در هر اپیزود به دست بیاورد (که در تابع قبلی محاسبه شد) را با مقدار میانگین ریوارد دریافتی حساب می‌کنیم و بعد مجموع cumulative `regret` را حساب کرده و در آخر آن را رسم می‌کنیم. تابع رسم این نمودار در `plots.py` قرار دارد و در زیر آورده شده است.

```
def plot_regret(repeats_rewards1, repeats_rewards2, max_expected_reward, label1, label2, window_size=1):
    averaged_rewards1 = get_average(repeats_rewards1, 1)
    averaged_rewards2 = get_average(repeats_rewards2, 1)

    cumulative_regret1 = get_cumulative_regret(averaged_rewards1, max_expected_reward)
    cumulative_regret2 = get_cumulative_regret(averaged_rewards2, max_expected_reward)

    plt.plot(cumulative_regret1, label=label1)
    plt.plot(cumulative_regret2, label=label2)
    plt.xlabel('episodes')
    plt.ylabel('regret')
    plt.legend()
    plt.show()
```

شکل ۸- کد رسم پشیمانی با ورودی گرفتن پادash‌های ایجنت و امیدریاضی ماکسیمم ریوارد

پیاده‌سازی توابع `get_cumulative_reward` و `get_average` در زیر آورده شده.

```
def get_average(repeats_rewards, window_size):
    repeats_rewards = np.array(repeats_rewards)
    episodes = repeats_rewards.shape[1]
    averaged_rewards = [np.mean(repeats_rewards[:, episode]) for episode in range(episodes)]
    averaged_rewards = [np.mean(averaged_rewards[start:start+window_size]) for start in range(episodes>window_size+1)]

    return averaged_rewards

def get_cumulative_regret(averaged_rewards, max_expected_reward):
    regret = [max_expected_reward - reward for reward in averaged_rewards]
    cumulative_regret = [regret[0]]
    for i in range(1, len(regret)):
        cumulative_regret.append(cumulative_regret[i-1]+regret[i])

    return cumulative_regret
```

شکل ۹- کد تابع `get_cumulative_reward` و `get_average` به ترتیب برای گرفتن میانگین بر روی اجراهای مختلف و محاسبه‌ی پشیمانی تجمعی باگذشت زمان و اپیزودها

تابع `get_average` ابتدا مجموع ریوارد دریافتی در هر اپیزود را برای اجراهای مختلف میانگین می‌گیرد و سپس میانگین را در پنجره‌ی متحرک داده شده حساب می‌کند و آن را در خروجی بر می‌گرداند. تابع `get_cumulative_regret` هم میانگین مجموع پادash دریافتی در هر اپیزود را دریافت می‌کند و اختلاف آن با ریوارد حالت بهینه را حساب کرده و مجموع این `regret`‌ها را به صورت cumulative بر می‌گرداند.

همچنین در هر سوال، باید ریوارد دریافتی در هر اپیزود را بین ۲۰ تکرار میانگین بگیریم و نمودار تعییرات این میانگین را رسم کنیم. به این منظور ما باز هم مجموع پادash‌های دریافتی در هر اپیزود را دریافت می‌کنیم، سپس در ۲۰ بار اجرا میانگین می‌گیریم و نمودار آن را رسم می‌کنیم. این پیاده‌سازی در تابع `plot_total_episode_rewards` انجام شده است و ما به عنوان حالت ایده‌آل، مقدار ماکسیمم ریواردی که می‌توانست در حالت سیاست بهینه بدست بیاورد را هم رسم کرده‌ایم و در نمودارهایی که در بخش نتایج می‌بینیم قابل مقایسه هستند.

```

def plot_total_episode_rewards(repeats_rewards1, repeat_rewards2, max_expected_reward, label1, label2, window_size=1):
    averaged_rewards1 = get_average(repeats_rewards1, window_size)
    averaged_rewards2 = get_average(repeat_rewards2, window_size)

    plt.plot(averaged_rewards1, label=label1)
    plt.plot(averaged_rewards2, label=label2)
    plt.plot([max_expected_reward for _ in averaged_rewards], label='expected maximum reward')
    plt.xlabel('episodes')
    plt.ylabel('rewards')
    plt.legend()
    plt.show()

```

شکل ۱۰- کد رسم نمودار میانگین مجموع پاداش در هر اپیزود

در رابطه با بهروزرسانی سیاست‌ها، هر agent یک تابع update\_policies دارد که بسته به اینکه off-policy و یا باشد دو مقدار را بهروزرسانی می‌کند بهر حال سیاستی که agent در حال بهینه کردن آن است با اسم target\_policy در agent است و سیاستی که طبق آن رفتار می‌کند نام b-policy دارد. سیاست رفتاری ایجنت در این سوال‌های epsilon-greedy است و برای آپدیت سیاست رفتاری‌ای که update\_state\_b\_policy است از تابع epsilon-greedy استفاده می‌شود که سیاست را برای یک state خاص بهروزرسانی می‌کند. در این تابع ابتدا بهترین عمل برای هر استیت با توجه به  $q$ -value‌ها بدست می‌آید و سپس تمامی اعمال احتمال انتخاب epsilon/n را دریافت می‌کند با این تفاوت که عمل بهینه به اندازه‌ی epsilon-1 احتمال بیشتری برای انتخاب خواهد داشت.

تصویر بعدی پیاده‌سازی این تابع در پایتون را نمایش می‌دهد.

```

def update_state_b_policy(self, state):
    state_qs = [self.Q[(state, a)] for a in self.actions]
    max_q_idx = np.argmax(state_qs)

    self.b_policy[state] = np.zeros(len(self.actions)) + (self.epsilon / len(self.actions))
    self.b_policy[state][max_q_idx] += (1 - self.epsilon)

```

شکل ۱۱- کد بهروزرسانی سیاست رفتاری ایجنت برای یک استیت ورودی با سیاست اپسیلون گریدی

همچنین در صورتی که سیاست هدف ما یک سیاست گریدی باشد، تابع update\_state\_t\_policy آن را برای یک استیت خاص آپدیت خواهد کرد. این تابع هم در ابتدا بهترین عمل را با مکسیمم گرفتن بر روی  $q$ -value‌ها بهدست می‌آورد و سپس با احتمال همه‌ی اعمال را به جز این عمل که احتمال ۱ دارد، برابر با صفر قرار می‌دهد. پیاده‌سازی آن را در ادامه می‌بینید.

```

def update_state_t_policy(self, state):
    state_qs = [self.Q[(state, a)] for a in self.actions]
    max_q_idx = np.argmax(state_qs)

    self.target_policy[state] = np.zeros(len(self.actions))
    self.target_policy[state][max_q_idx] = 1

```

شکل ۱۲- کد بهروزرسانی سیاست هدف برای یک استیت ورودی به صورت گریدی

یکی دیگر از قسمت‌های مشترک در بین تمامی ایجنت‌ها، تابع take\_action است که در آن تابع کلاس environment صدا زده خواهد شد. این تابع بر اساس سیاست رفتاری (که اپسیلون گریدی است) یک عمل را در استیتی که در آن قرار دارد انتخاب می‌کند و آن را در محیط اعمال می‌کند سپس نتایج را که شامل استیت بعدی، ریوارد، پایان یافتن و یا نیافتن اپیزود را از محیط دریافت می‌کند و آن‌ها را به همراه اکشن انتخاب شده در خروجی قرار می‌دهد.

```

def take_action(self, state):
    action = np.random.choice(self.actions, p=self.b_policy[state])
    next_state, reward, done, _ = self.environment.step(action)

    return action, next_state, reward, done

```

شکل ۱۳- تابع انتخاب یک اکشن و اعمال آن در محیط با توجه به سیاست رفتاری ایجنت

## نتایج

در ابتدا یک مقایسه کلی بین روش‌ها انجام می‌دهیم. با توجه به بخش‌های مختلف این سوال، بهترین نتیجه مربوط به الگوریتم Sarsa است. البته عمل کرد هم ۳ الگوریتم N-step و q-learning backup قابل قبول است اما بر حسب اعداد Sarsa در این مساله بهتر عمل کرده است. همچنین عمل کرد الگوریتم MC اصلاً قابل قبول نبود و این الگوریتم کارایی لازم برای این مساله را نداشت.

نتایج مربوط به هر زیربخش این سوال، در قسمت مربوطه گزارش شده است.

## الگوریتم MC : off-policy

### توضیح پیاده‌سازی:

برای توضیح پیاده‌سازی این الگوریتم، ابتدا سودوکد آن را که در کتاب ساتن آورده شده است بررسی می‌کنیم و با توجه به اینکه پیاده‌سازی ما هم دقیقاً با توجه به همین سودوکد است بعد از این بررسی بخش‌های مختلف کد را توضیح خواهیم داد.

این سودوکد در تصویر زیر آورده شده است.

```

Off-policy MC control, for estimating  $\pi \approx \pi_*$ 

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :
 $Q(s, a) \in \mathbb{R}$  (arbitrarily)
 $C(s, a) \leftarrow 0$ 
 $\pi(s) \leftarrow \arg \max_a Q(s, a)$  (with ties broken consistently)

Loop forever (for each episode):
 $b \leftarrow$  any soft policy
Generate an episode using  $b$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ 
 $G \leftarrow 0$ 
 $W \leftarrow 1$ 
Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :
 $G \leftarrow \gamma G + R_{t+1}$ 
 $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$ 
 $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$ 
 $\pi(S_t) \leftarrow \arg \max_a Q(S_t, a)$  (with ties broken consistently)
If  $A_t \neq \pi(S_t)$  then exit inner Loop (proceed to next episode)
 $W \leftarrow W \frac{1}{b(A_t | S_t)}$ 

```

شکل ۱۴- سودوکد الگوریتم پیاده‌سازی شده‌ی Off-policy MC

در ابتدا مقادیر q-value به صورت رندم مقدار دهی می‌شوند و C برای هر استیت و اکشن برابر با صفر خواهد بود. همچنین سیاستی که می‌خواهیم بهینه کنیم یک سیاست گریدی با توجه به مقادیر q-value است.

سپس هر بار یک episode را با توجه به یک سیاست epsilon-soft-policy که در حالت ما یک سیاست greedy با توجه به q-value را ایجاد می‌کند و سعی می‌کند که با توجه به پاداش‌های دریافتی، مقدار return هر استیت را تخمین بزند و این را از انتهای اپیزود به سمت ابتدای آن انجام می‌دهد برای این کار با استفاده از W و C درواقع learning-rate در مرحله را تعیین می‌کنیم و همچنین سیاست

را بعد از بهروزرسانی تخمین‌ها، آپدیت می‌کنیم در صورتی که در یک مرحله، اکشن انتخاب شده با اکشنی که سیاست هدف انتخاب می‌کرد برابر نباشد، مراحل قبلی آن دیگر به کار نخواهد آمد.  
برای توضیح پیاده‌سازی ما در ابتدا کد این بخش را می‌آوریم.

```
def off_policy_mc(self):
    total_rewards = []
    for epoch in range(REPS):
        self.reset()
        episode_rewards = []
        for e in range(self.episodes):
            self.update_policies()
            states, actions, rewards = self.generate_episode()
            g, w = 0, 1
            for s, a, r in zip(states, actions, rewards):
                g += self.discount*g + r
                self.C[(s, a)] += w
                self.Q[(s, a)] += (w / self.C[(s, a)]) * (g - self.Q[(s, a)])
                self.update_state_t_policy(s)
                if a != np.argmax(self.target_policy[s]):
                    break
                w *= 1 / self.b_policy[s][a]
            episode_rewards.append(sum(rewards))
        if self.decade_epsilon and (e+1)%40 == 0:
            self.epsilon *= 0.99
        self.environment.reset()
        total_rewards.append(episode_rewards)
    return total_rewards
```

شکل ۱۵- پیاده‌سازی الگوریتم Off-policy MC با سودوکد شکل ۱۴

این الگوریتم را تعداد ۲۰ بار انجام می‌دهیم. در ابتدا همهٔ اطلاعات را reset می‌کنیم. سپس برای تعداد ۴۰۰۰ اپیزود، هر بار با استفاده ازتابع generate\_episode یک اپیزود را ایجاد می‌کنیم. نحوهٔ کار این تابع را در ادامه توضیح خواهیم داد.

سپس درست مطابق سودوکد بالا، از انتهای اپیزود شروع به بهروزرسانی مقادیر می‌کنیم هر مرحله

$$G = \text{discount}^*G + R$$

خواهد بود که یعنی مطابق رابطهٔ return ارزش استیت‌های پیش رو در یک ضریب discount ضرب می‌شوند. همچنین مقدار learning-rate را هم مطابق سودوکد محاسبه می‌کنیم و مقدار q-value را به اندازهٔ learning\_rate\*(G - q-value) تغییر می‌دهیم. بعد از این بهروزرسانی دو سیاست خود را آپدیت می‌کنیم. همچنین مجموع ریوارد دریافتی در هر اپیزود را هم ذخیره می‌کنیم تا در نهایت بتوانیم با توجه به این مقادیر نمودارهای خواسته شده را رسم کنیم.

در صورتی که قرار بر کاهش epsilon در اجرای این الگوریتم باشد هم، هر بار بعد از گذشت ۲۰ اپیزود، مقدار آن را در  $0.99^{0.99}$  ضرب می‌کنیم. (البته در هر بار تکرار اجرای این الگوریتم، دوباره مقدار اوایه‌ی ۰.۹۹ را در آن قرار خواهیم داد).

## نتایج:

ابتدا سیاست یافته‌شده توسط هر یک از دو اجرای با اپسیلون ثابت و اپسیلون کاهشی را بررسی می‌کنیم.

0.000	0.001	0.332	0.746
0.696	0.001	0.143	0.998
0.703	0.001	0.001	0.001
0.861	0.401	0.128	0.000

↓|→|→|↑  
→|↓|↑|↑  
↓|→|→|↑  
←|→|→|↑

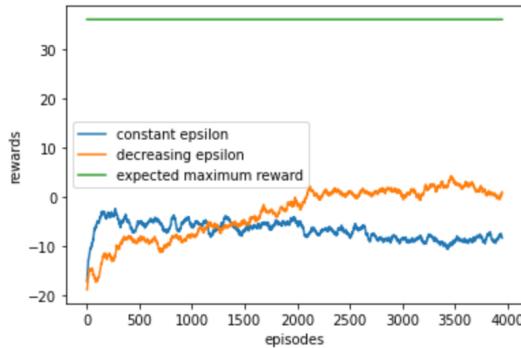
شکل ۱۶- سیاست پیدا شده توسط الگوریتم MC off-policy با اپسیلون ثابت بعد از ۴۰۰۰ اپیزود برای محیط بالای شکل

0.000	0.001	0.332	0.746
0.696	0.001	0.143	0.998
0.703	0.001	0.001	0.001
0.861	0.401	0.128	0.000

→|↓|↑|→  
↓|↓|↑|↑  
↓|→|↓|↑  
↓|→|→|↑

شکل ۱۷- سیاست پیدا شده توسط الگوریتم MC off-policy با اپسیلون کاهشی بعد از ۴۰۰۰ اپیزود برای محیط بالای شکل

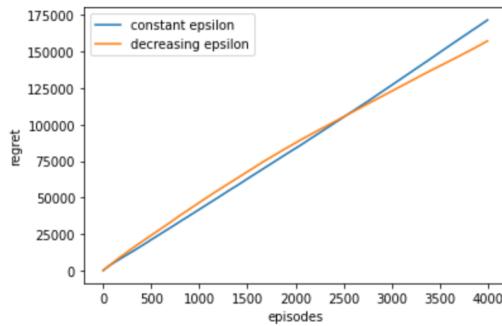
همانطور که در دو شکل بالا مشاهده می‌شود ایجنت در هیچکدام به بهترین سیاست دست نیافته است و هنوز اپیزودهای بیشتری برای انجام این کار نیاز دارد. اما برای مقایسه‌ی عمل کرد این دو، ابتدا نمودار مجموع ریوارد دریافتی در هر اپیزود را مقایسه می‌کنیم.



شکل ۱۸- نمودار میانگین مجموع پاداش هر اپیزود برای ۲۰ بار اجرای الگوریتم MC off-policy یکبار با اپسیلون ثابت و یکبار با اپسیلون کاهشی

نمودار بالا که مقایسه‌ی میانگین پاداش دریافتی در هر اپیزود در ۲۰ بار اجرای الگوریتم به ازای اپسیلون ثابت و کاهشی است، به خوبی نشان می‌دهد که در صورت کاهش مناسب epsilon الگوریتم می‌تواند با کاهش مناسب نرخ exploration پاداش دریافتی خود را افزایش بدهد و با نوسانات کلی کمتری رشد خود را طی کند (برای رسم این نمودار از پنجره‌ی متحرک با اندازه‌ی ۵۰ استفاده شده است تا نوسانات بسیار ریز در آن‌ها گرفته شود). البته همانطور که در تصویر مشخص است، پاداش دریافتی حتی در حالت اپسیلون کاهشی همچنان فاصله‌ی بسیار زیادی تا رسیدن به پاداش سیاست بهینه دارد. علت این غیربهینه بودن این الگوریتم است که با وجود تعداد اپیزودهای نسبتاً بالا، بخش اعظمی از این اپیزودها قابل استفاده برای بهروزرسانی نیستند در نتیجه سرعت پیشرفت الگوریتم بسیار کم است.

همچنین در ادامه برای بررسی regret حاصل از هر یک از این دو اجرای الگوریتم، شکل بعدی را خواهیم دید.



شکل ۱۹ - نمودار پشمیانی تجمعی در هر اپیزود برای ۲۰ بار اجرای الگوریتم off-policy MC یکبار با اپسیلون ثابت و یکبار با اپسیلون کاهشی

همانطور که مشخص است از  $\frac{1}{3}$  آخر یادگیری، نرخ رشد پشمیانی در الگوریتم با اپسیلون کاهشی کمتر از اپسیلون ثابت است و این فاصله در حال افزایش است. یعنی از نظر نرخ رشد پشمیانی، این الگوریتم در بینهایت بسیار بهتر از حالت اپسیلون ثابت عمل می‌کند. علت این امر این است که ما با یادگیری تدریجی محیط، نرخ exploration را کاهش می‌دهیم و در این مساله که محیط ما ثابت است این عمل به نفع ماست و نرخ انتخاب عمل بهینه را در طول زمان افزایش خواهیم داد.

به طور کلی regret در اپسیلون کاهشی نرخ رشد کمتری داشته و سریع‌تر همگرا خواهد شد. همچنین عدد همگرایی آن مقدار کمتری خواهد بود.

### الگوریتم : q-leaning

#### توضیح پیاده‌سازی:

برای توضیح پیاده‌سازی این الگوریتم، ابتدا سودوکد آن را که در کتاب ساتن آورده شده است بررسی می‌کنیم.

```

Q-learning (off-policy TD control) for estimating  $\pi \approx \pi_*$ 

Algorithm parameters: step size  $\alpha \in (0, 1]$ , small  $\varepsilon > 0$ 
Initialize  $Q(s, a)$ , for all  $s \in S^+, a \in \mathcal{A}(s)$ , arbitrarily except that  $Q(\text{terminal}, \cdot) = 0$ 
Loop for each episode:
  Initialize  $S$ 
  Loop for each step of episode:
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
    Take action  $A$ , observe  $R, S'$ 
     $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
     $S \leftarrow S'$ 
    until  $S$  is terminal

```

شکل ۲۰- سودوکد پیاده‌سازی شده‌ی الگوریتم Q-learning

در ابتداء مقادیر  $q$ -value برای تمامی استیت-اکشن‌ها به صورت رندم مقداردهی می‌شود با این تفاوت که برای استیت-اکشن‌هایی که استیت آن‌ها نهایی است این مقدار صفر است. سپس برای هر اپیزود ابتدا در استیت اولیه قرار می‌گیریم، سپس با توجه به سیاست رفتاری خود که در اینجا سیاست-epsilon است یک اکشن را در آن استیت انتخاب می‌کنیم و آن را اجرا می‌کنیم و استیت بعدی و پاداش

این عمل را به دست می‌آوریم. سپس  $q$  را با توجه به فرمولی که در سودوکد آورده شده به روزرسانی می‌کند. در این فرمول ما  $G$  را برابر با پاداش دریافتی به اضافه‌ی  $discount$  ضربدر ماقسیم مقداری که با توجه به  $q$ -value استیت بعدی می‌توانیم به دست بیاوریم تعریف می‌کنیم. سپس این عمل را برای استیت بعدی هم اجرا می‌کنیم تا به انتهای اپیزود برسیم.

پیاده‌سازی ما از این الگوریتم دقیقاً طبق همان سودوکد و به صورت زیر است.

```
def q_learning(self):
    total_rewards = []
    for epoch in range(REPS):
        self.reset()
        episode_rewards = []
        for e in range(self.episodes):
            rewards = []
            state = self.environment.reset()
            while True:
                action, next_state, reward, done = self.take_action(state)
                rewards.append(reward)

                max_next_q = max([self.Q[(next_state, a)] for a in self.actions])
                self.Q[(state, action)] += self.alpha * (reward + self.discount * max_next_q - self.Q[(state, action)])
                self.update_state_b_policy(state)
                state = next_state

                if done:
                    break
            episode_rewards.append(sum(rewards))

            if (e+1)%10 == 0:
                self.epsilon *= 0.99
                if self.decade_lr:
                    self.alpha *= 0.99

            self.environment.reset()
            total_rewards.append(episode_rewards)

        self.update_policies()

    return total_rewards
```

شکل ۲۱- کد پیاده‌سازی الگوریتم Q-learning با سودوکد شکل ۲۰

در این کد، ما الگوریتم را برای ۲۰ بار اجرا می‌کنیم و هر بار داش *agent* را به حالت اولیه برمی‌گردانیم. سپس برای ۲۰۰۰ اپیزود، هر بار عمل بعدی استیتی که در آن قرار داریم را با توجه به سیاست رفتاری خود انتخاب می‌کنیم و استیت بعدی و ریوارد این مرحله را به دست می‌آوریم. سپس ماقسیم  $q$ -value استیت بعدی موجود است را حساب می‌کنیم و با استفاده از آن مطابق رابطه‌ی گفته شده مقدار  $q$ -value استیت فعلی را به روز رسانی می‌کنیم. در صورتی که استیت بعدی ما استیت نهایی اپیزود نباشد، بعد از به روزرسانی سیاست‌ها همین کار را برای استیت بعدی هم تکرار می‌کنیم. در نهایت هم بعد از مجموع ریواردها در هر اپیزود را خروجی می‌دهیم تا برای رسم توابع از آن‌ها استفاده کنیم.

همچنین طبق صورت سؤال اپسیلوون ما باید کاهشی باشد و ما بعد از هر ۲۰ اپیزود مقدار آن را در ۰.۹۹ ضرب می‌کنیم و اگر در حالت learning rate کاهشی باشیم، مقدار آن را هم بعد از هر ۲۰ اپیزود در ۰.۹۹ ضرب می‌کنیم.

### نتایج:

سیاست یافته‌شده توسط الگوریتم q-learning برای نرخ یادگیری ثابت و کاهشی در شکل‌های زیر قابل مشاهده است.

0.000	0.001	0.332	0.746
0.696	0.001	0.143	0.998
0.703	0.001	0.001	0.001
0.861	0.401	0.128	0.000

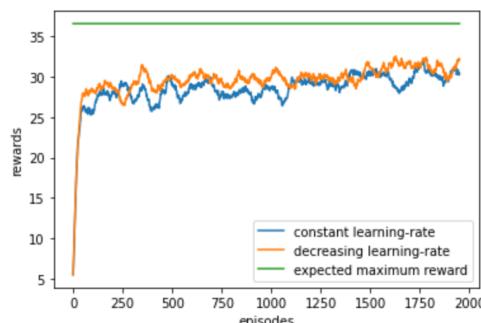
→↓←↑  
↑↓↓↑  
↑→↓↓  
↑→↓↓

شکل ۲۲- سیاست پیدا شده توسط الگوریتم Q-learning با نرخ یادگیری ثابت بعد از ۲۰۰۰ اپیزود برای محیط بالای شکل

0.000	0.001	0.332	0.746
0.696	0.001	0.143	0.998
0.703	0.001	0.001	0.001
0.861	0.401	0.128	0.000

→↓←↑  
↑↓↓↑  
→↓↓  
↑→↓↓

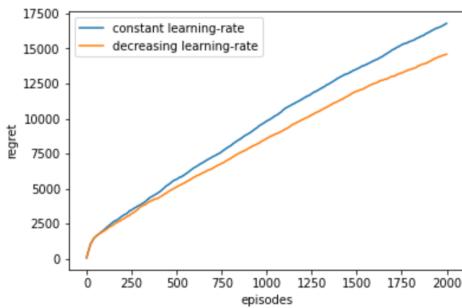
شکل ۲۳- سیاست پیدا شده توسط الگوریتم Q-learning با نرخ یادگیری متغیر بعد از ۲۰۰۰ اپیزود برای محیط بالای شکل در هر دوی این اجراهای، الگوریتم با دقت خوبی (در مسیر بهینه به صورت دقیق) به سیاست بهینه نزدیک شده است. برای مقایسه‌ی بهتر ابتدا نمودار مجموع پاداش دریافتی در اپیزودها با سایز پنجره‌ی متحرک برابر با ۵۰ را بررسی می‌کنیم.



شکل ۲۴- نمودار میانگین مجموع پاداش دریافتی در ۲۰ بار اجرای الگوریتم q-learning یک بار با نرخ یادگیری ثابت و بار دیگر با نرخ یادگیری کاهشی

میانگین پاداش دریافتی در این دو اجرا (یکی با نرخ یادگیری ثابت و دیگری با نرخ یادگیری کاهشی) تفاوت زیادی با یکدیگر ندارند و تقریباً بعد از ۲۰۰۰ اپیزود به مقدار یکسانی رسیده‌اند. چیزی که در این دو حالت متفاوت است، کاهش نوسانات در اواخر یادگیری در حالت نرخ یادگیری کاهشی نسبت به ثابت است. در اواخر یادگیری  $q$ -value‌ها با تقریب خوبی تخمین زده شده‌اند لذا ما باید به دانش قبلی خود متکی‌تر باشیم. اگر نرخ یادگیری را با زمان کاهش ندهیم، به نوسانات ناشی از احتمالات محیط واکنش بیشتری نشان خواهیم داد در صورتی که با مرور زمان باید سعی در نرم کردن این نوسانات داشته‌باشیم.

نمودار regret برای این دو حالت در شکل زیر آورده شده است.



شکل ۲۵-نمودار میانگین پشیمانی تجمعی در ۲۰ بار اجرای الگوریتم q-learning یک بار با نرخ یادگیری ثابت و بار دیگر با نرخ یادگیری کاهشی

به نظر رشد regret هم در حالت decreasing learning-rate کمتر از حالت ثابت است هرچند تفاوت زیادی ندارند. این اتفاق را می‌توان با مثالی توجیه کرد. زمانی که در تخمین ما  $q$ -value را عمل به یکدیگر نزدیک باشند، اگر ما با نرخ یادگیری بالایی، اثر ناشی از یک نوسان در احتمالات را در تخمین خود دخیل کنیم، در سیاست می‌توانیم عمل بهینه رو کاملاً عوض کنیم. این اتفاق ممکن است مدت زمانی طول بکشد تا به حالت بهینه برگردد (مخصوصاً اگر در چند آپدیت بعدی همچنان در قسمت تقریباً یکسانی ازتابع توزیع قرار داشته باشیم) و این می‌تواند باعث افزایش در تابع regret ما بشود.

به صورت کلی نرخ رشد regret به نظر می‌رسد در نرخ یادگیری کاهشی کمتر باشد و نهایتاً به عدد کوچکتر و با سرعت بیشتری همگرا شود.

## الگوریتم : Sarsa

### توضیح پیاده‌سازی:

سودوکد این الگوریتم هم دقیقاً همان سودوکد آورده شده در کتاب ساتن است که به صورت زیر است.

```
Sarsa (on-policy TD control) for estimating  $Q \approx q_*$ 

Algorithm parameters: step size  $\alpha \in (0, 1]$ , small  $\varepsilon > 0$ 
Initialize  $Q(s, a)$ , for all  $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$ , arbitrarily except that  $Q(\text{terminal}, \cdot) = 0$ 
Loop for each episode:
  Initialize  $S$ 
  Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
  Loop for each step of episode:
    Take action  $A$ , observe  $R, S'$ 
    Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
     $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$ 
     $S \leftarrow S'; A \leftarrow A';$ 
  until  $S$  is terminal
```

شکل ۲۶- سودوکد پیاده‌سازی شدهی الگوریتم Sarsa

به عنوان مقداردهی اولیه،  $q$ -value ها برای تمامی استیت اکشن ها به جز آن هایی که استیت آنها terminal است و مقدارشان صفر است به صورت رندم مقداردهی می‌شوند.

سپس برای هر اپیزود یک استیت ابتدایی در نظر می‌گیریم و اکشن آن را با توجه به سیاست اپسیلون-گریدی انتخاب می‌کنیم. سپس با دریافت استیت بعدی و ریواد این اکشن، طبق همان سیاست قبلی تصمیم می‌گیریم که چه اکشنی در استیت بعد انتخاب خواهد شد. سپس  $q$ -value استیت فعلی را

با مقدار  $G$  برابر با ریوارد دریافتی بهاضافه‌ی q-value اکشن بعدی‌ای که انتخاب کردۀ‌ایم در استیت بعدی (که در یک ضرب هم ضرب شده است) بهروزرسانی می‌کنیم و سپس همین کار را برای استیت بعدی هم انجام خواهیم داد.

پیاده‌سازی ما از این الگوریتم به صورت زیر است.

```
def sarsa(self):
    total_rewards = []
    for epoch in range(REPS):
        self.reset()
        episode_rewards = []
        for e in range(self.episodes):
            rewards = []
            state = self.environment.reset()
            action, next_state, reward, done = self.take_action(state)
            while True:
                rewards.append(reward)
                next_action, next_next_state, next_reward, next_done = self.take_action(next_state)
                self.Q[(state, action)] += \
                    self.alpha*(reward+next_reward)*self.discount*self.Q[(next_state, next_action)]-self.Q[(state, action)]
                self.update_state_b_policy(state)
                state = next_state; action = next_action;
            if done:
                break
            next_state = next_next_state; reward = next_reward; done = next_done;
            episode_rewards.append(sum(rewards))
        if (e+1)%10 == 0:
            self.epsilon *= 0.99
        self.environment.reset()
        total_rewards.append(episode_rewards)
    return total_rewards
```

شکل ۲۷- کد پیاده‌سازی الگوریتم Sarsa با سودوکد شکل ۲۶

این الگوریتم را برای ۲۰ بار تکرار می‌کنیم و در هر بار با reset کردن اینجنت شروع خواهیم کرد. سپس با توجه به استیت اولیه، یک اکشن را انتخاب می‌کنیم و ریوارد و استیت بعدی و پایان‌یافتن یا نیافتن اپیزود را هم دریافت خواهیم کرد. سپس انتخاب اکشن را برای استیت بعدی که در قدم قبلی دریافت کردیم انجام خواهیم داد (از آنجایی که بهر حال این اکشن را قسمت‌های بعدی سودوکد اعمال خواهیم کرد، ما در اینجا آن را اعمال کردیم تا ریوارد و استیت بعد از آن را در دور بعدی حلقه استفاده کنیم) سپس با استفاده از q-value عمل انتخاب شده استیت و اکشن مرحله‌ی فعلی الگوریتم را مطابق سودوکد آپدیت می‌کنیم. در نهایت به دور بعدی حلقه می‌رویم تا زمانی که به انتهای اپیزود برسیم. همچنین مجموع پاداش‌ها در هر اپیزود را به عنوان خروجی بر می‌گردانیم تا بتوانیم از آن برای رسم نمودارها استفاده کنیم.

### نتایج:

نتایج باید با الگوریتم ۲ step tree backup مقایسه شود. لذا بعد از بررسی این الگوریتم این دو را مقایسه خواهیم کرد.

## الگوریتم : n-step tree backup

### توضیح پیاده‌سازی:

تصویر زیر سودوکد ارائه شده برای این الگوریتم را در کتاب ساتن نشان می‌دهد.

```

n-step Tree Backup for estimating  $Q \approx q_*$  or  $q_\pi$ 

Initialize  $Q(s, a)$  arbitrarily, for all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
Initialize  $\pi$  to be greedy with respect to  $Q$ , or as a fixed given policy
Algorithm parameters: step size  $\alpha \in (0, 1]$ , a positive integer  $n$ 
All store and access operations can take their index mod  $n + 1$ 

Loop for each episode:
  Initialize and store  $S_0 \neq$  terminal
  Choose an action  $A_0$  arbitrarily as a function of  $S_0$ ; Store  $A_0$ 
   $T \leftarrow \infty$ 
  Loop for  $t = 0, 1, 2, \dots$ :
    | If  $t < T$ :
      |   Take action  $A_t$ ; observe and store the next reward and state as  $R_{t+1}, S_{t+1}$ 
      |   If  $S_{t+1}$  is terminal:
          |      $T \leftarrow t + 1$ 
      |   else:
          |     Choose an action  $A_{t+1}$  arbitrarily as a function of  $S_{t+1}$ ; Store  $A_{t+1}$ 
          |      $\tau \leftarrow t + 1 - n$  ( $\tau$  is the time whose estimate is being updated)
      |   If  $\tau \geq 0$ :
          |     If  $t + 1 \geq T$ :
              |        $G \leftarrow R_T$ 
          |     else
              |        $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a)$ 
          |     Loop for  $k = \min(t, T - 1)$  down through  $\tau + 1$ :
              |        $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k)Q(S_k, a) + \gamma \pi(A_k|S_k)G$ 
              |        $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$ 
          |     If  $\pi$  is being learned, then ensure that  $\pi(\cdot|S_\tau)$  is greedy wrt  $Q$ 
    Until  $\tau = T - 1$ 

```

شکل ۲۸- سودوکد پیاده‌سازی شده‌ی الگوریتم N-step tree backup

در ابتدا مقادیر اولیه  $Q$  به صورت رندم مقداردهی خواهد شد و سیاست یک سیاست اپسیلون-گریدی بر حسب  $Q$  خواهد بود.

برای هر اپیزود، استیت اولیه انتخاب می‌شود. سپس عمل مورد نظر در آن استیت هم انتخاب خواهد شد. در یک حلقه، تا زمانی که به یک استیت ترمینال برسیم، هر بار یک اکشن انتخاب شده و استیت بعدی و ریوارد این انتخاب را دریافت می‌کنیم. همچنین زمانی که در حلقه در استپ  $t$  قرار داریم، برای آپدیت  $q\text{-value}$  استپ  $t+1-n$  حلقه می‌زنیم و طبق فرمول‌های سودوکد، هم از ریوارد دریافتی در این  $n$  استپ استفاده می‌کنیم و هم از  $q\text{-value}$  اکشن‌های انتخاب نشده تا آپدیت را برای این استپ انجام دهیم.

### نتایج:

پالیسی‌های پیدا شده توسط این دو الگوریتم را در زیر مشاهده می‌کنید.

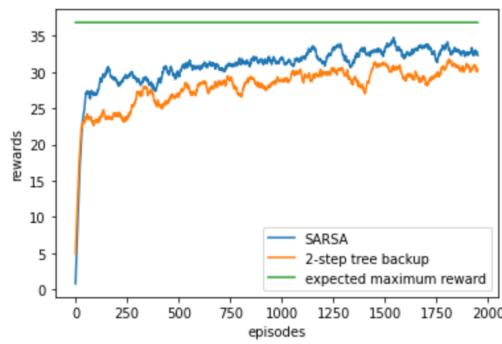
<b>0.000</b>	0.001	0.332	0.746
0.696	0.001	0.143	0.998
0.703	0.001	0.001	0.001
0.861	0.401	0.128	0.000
→ ↓ ← ↑	→ ↓ ← ↓	→ ↓ ↓ ↓	→ → ↓ ↑

شکل ۲۹- سیاست پیدا شده توسط الگوریتم Sarsa بعد از ۲۰۰۰ اپیزود برای محیط بالای شکل

0.000	0.001	0.332	0.746
0.696	0.001	0.143	0.998
0.703	0.001	0.001	0.001
0.861	0.401	0.128	0.000
→↓↓←	→↓↓↑	→→↓↓	↑↑↑↑

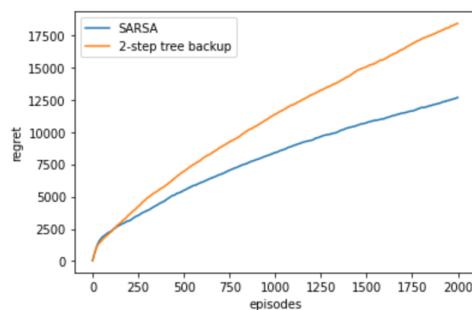
شکل ۳۰- سیاست پیدا شده توسط الگوریتم ۲ step tree backup بعد از ۲۰۰۰ اپیزود برای محیط بالای شکل

هر دو الگوریتم تا حدی به سیاست بهینه همگرا شده‌اند و در ادامه با مقایسه‌ی دو نمودار خواسته شده عملکرد آن‌ها را مقایسه خواهیم کرد.



شکل ۳۱- نمودار میانگین مجموع پاداش دریافتی در هر اپیزود برای ۲۰ برا اجرا بر روی دو الگوریتم Sarsa و ۲ step tree backup

در نمودار مجموع پاداش در اپیزودها که در ۲۰ بار اجرا میانگین گرفته شده است، الگوریتم Sarsa عمل کرد بهتری داشته و در نهایت به نظر می‌رسد که به تدریج فاصله‌ی n-step tree backup با آن کمتر می‌شود. در این مساله با توجه به ثابت بودن محیط بودن، به روزرسانی آنی‌تر سیاست در الگوریتم Sarsa ظاهراً مؤثر بوده و به نتیجه‌ی بهتری منجر شده است. این نتیجه‌گیری با بررسی نمودار regret این دو الگوریتم هم قابل تأیید دوباره است.



شکل ۳۲- نمودار میانگین پشیمانی تجمعی در هر اپیزود برای ۲۰ برا اجرا بر روی دو الگوریتم Sarsa و ۲ step tree backup کمتر از ۲۰۰۰ اپیزود برای اجرا این مساله عمل کرد بهتری داشته است.

در مجموع به نظر می‌رسد نرخ رشد regret در الگوریتم Sarsa به مراتب کمتر است و در بینهایت به عدد کوچکتر و با سرعت بیشتری همگرا خواهد شد.

### روند اجرای کد پیاده‌سازی

برای پیاده‌سازی کلیه‌ی بخش‌ها باید فایل جوپیتر آن را ران کنیم. برای قسمت اول سؤال Qlearning.ipynb برای قسمت دوم OffPolicyMC.ipynb، برای قسمت سوم ValueIteration.ipynb و برای قسمت چهارم هم Sarsa\_2StepBackup.ipynb باید اجرا شود. از آنجایی که توابعی که در فایل‌ها مجزا قرار دارند، مشکلی برای اجرای این بخش‌ها به وجود نخواهد آمد.

## سوال ۳ - ترکیب روش‌های model-free و model-based

### هدف سوال

در مسائل دنیای واقعی، ممکن است که ما تسلک جدیدی را در محیطی قدیمی انجام دهیم. در این موقع به طور پیش‌فرض اطلاعاتی داریم که برخی از آن‌ها تغییر کرده‌اند و برخی خیر. برای مثال احتمالاً پاداش دریافتی در ازای هر ترنزیشن در تسلک جدید متفاوت است اما احتمال انتقال‌ها یکسان. انسان در این موقع به نظر می‌رسد که از دانش پیشین خود استفاده می‌کند. به همین منظور در این سؤال ما سعی می‌کنیم دانش به دست آمده از الگوریتم value-iteration را که در قسمت اول سؤال در اختیار داشتیم، در کنار الگوریتم Q-learning که یک روش یادگیری model-free است استفاده کنیم و با استفاده از ترکیب این دو در یک محیط داینامیک به یادگیری و تصمیم‌گیری بپردازیم.

### توضیح پیاده‌سازی

برای پیاده‌سازی این قسمت از قسمت‌های قبل استفاده شده‌است. ما یک Agent داریم که سه q-value متفاوت نگهداری می‌کند. اولین آن‌ها  $Q_{vi}$  است که های q-value حاصل از اجرای الگوریتم value-iteration است. دومین آن‌ها  $Q_{ql}$  است که مقادیر یادگرفته شده با الگوریتم Q-learning است و سومین  $Q_{dm}$  که نام دارد و برای تصمیم‌گیری استفاده می‌شود، میانگین وزن دار دو  $Q$  قبلی است.

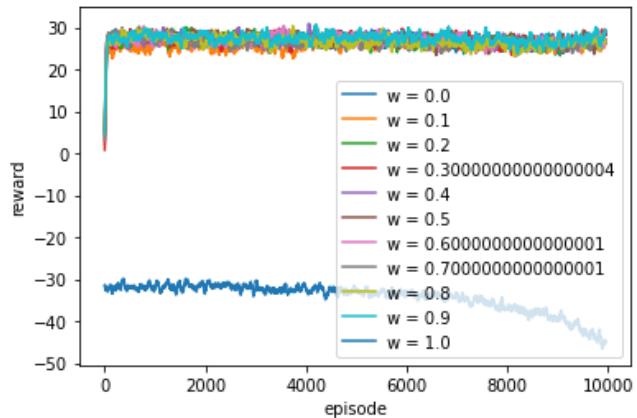
در این بخش، در ابتدا محیطی مشابه با محيط ۲ سؤال قبل خواهیم داشت. در همین مرحله و قبل از تغییر محیط، الگوریتم value-iteration را درست مشابه سؤال یک اجرا می‌کنیم و های q-value ذخیره می‌کنیم. سپس الگوریتم Q-learning را برای ۱۰۰۰۰ اپیزود درست مشابه قسمت ۲ سؤال دوم اجرا می‌کنیم و در هر مرحله سیاست را با توجه به یک میانگین وزن دار از این دو مقدار به روزرسانی می‌کنیم (که یک سیاست اپسیلون-گریدی است) و این کار را برای وزن‌های متفاوت تکرار می‌کنیم.

برای بررسی جزئیات پیاده‌سازی الگوریتم value-iteration به اینجا: ۵ مراجعه کنید. همچنین برای الگوریتم Q-learning هم به اینجا: ۱۵ مراجعه کنید.

همانطور که اشاره شد الگوریتم value-iteration در لحظه‌ی اول اجرا می‌شود و بعد دیگر محیط شروع به تغییر می‌کند و الگوریتم q-learning با توجه به این تغییرات خود را به روز می‌کند. همچنین در هر تکرار ما محیط را از ابتدا مقدار دهی می‌کنیم و دانش agent را هم مجدداً ریست می‌کنیم.

### نتایج

نمودار مجموع پاداش دریافتی در هر اپیزود برای  $w$ ‌های متفاوت به صورت زیر است:



شکل ۳۳- نمودار مجموع پاداش‌های دریافتی در هر اپیزود برای ۲۰ بار تکرار الگوریتم

تقریباً تمامی آن‌هایی که هم از MF و هم از MB استفاده کرده‌اند در طول ۱۰۰۰۰ اپیزود نتیجه‌ی یکسانی گرفته‌اند. از آنجایی که تغییرات محیط هم ناگهانی نیست این اختلاف تا حدی توجیه می‌شود اما اگر تنها متکی به اطلاعات model-based باشیم، همان‌طور که در تصویر مشاهده می‌شود به تدریج در انتهای پاداش دریافتی ما کاهش پیدا می‌کند و هیچ پیشرفتی در جهت یادگیری بهتر محیط اتفاق نمی‌افتد. (متأسفانه به علت زمان اجرای بسیار بالای الگوریتم موفق به اجرای مجدد برای رسم بهتر و واضح‌تر نمودار نشدم)

### روند اجرای کد پیاده‌سازی

برای اجرای کد پیاده‌سازی کافی است فایل MBPlusMF.ipynb را اجرا کنید.

## سوال ۴ - ترکیب اطلاعات ناقص محیط با Model-free

### هدف سوال

در این حالت، اطلاعات ما نسبت به سؤال ۳ هم ناقص‌تر است. یعنی ما نمی‌دانیم که خطر سقوط در هر خانه چقدر است و تنها کفش خود را می‌شناسیم! تنها دانستن احتمال ترنزیشن راه مناسبی برای بهدست آوردن اولیه‌ی ارزش هر استیت ندارد. اما در این مساله قرار است که تلاش کنیم تا این داش را در کد خود استفاده کنیم.

### توضیح پیاده‌سازی

برای پیاده‌سازی این قسمت، من الگوریتم Q-learning را به عنوان الگوریتم پایه در نظر گرفتم. با این تفاوت که مقدار دقیق احتمال ترنزیشن با انجام یک عمل را می‌دانم (که با  $P(s'|s, a)$  نمایش داده شده است و در هر مرحله، به جای اینکه  $R$  را قرار بدهیم، سعی می‌کنیم امیدریاضی  $R$  با انجام عمل  $A$  را بسازیم. به این منظور، از آنجایی که احتمالات انتقال را می‌دانیم، ابتدا احتمال انتقال به تمام استیت‌های دیگر به جز  $S'$  و ارزش حاصله از آن‌ها را حساب می‌کنیم (برای جمع  $Q$ ‌ها هم از سیاست استفاده می‌کنیم تا بتوانیم امیدریاضی  $Q$ -value‌ها در استیت‌ها را جایگزین کنیم) سپس از ریوارد بهدست آمده استفاده می‌کنیم و با در نظر گرفتن احتمال این ترنزیشن،  $R$  را به جای تخمین قبلی قرار می‌دهیم (تمام این عبارت به جای  $R$  در الگوریتم Q-learning قرار می‌گیرید و در شکل هم مشخص شده است) سپس به جای اینکه ماقسیم  $Q$ -value در استیتی که در آن قرار داریم را انتخاب کنیم، باز هم امید ریاضی آن‌ها را استفاده می‌کنیم و اختلاف آن با تخمین قبلی را در نرخ یادگیری ضرب می‌کنیم.

MB-learning

- algorithm parameters: step size  $\alpha \in (0,1]$ , small  $\epsilon > 0$
- initializing  $Q(s,a)$  for  $s,a$  except terminal states which are 0
- $Q(\text{terminal}, \cdot) = 0$
- $P(s'|s,a)$  is given.
- loop for each episode:
  - initialize  $S$
  - loop for each step of episode:
    - choose  $A$  from  $S$  using policy derived from  $Q$
    - take action  $A$ , observe  $R, S'$
    - $Q(s,A) \leftarrow Q(s,A) + \alpha \left[ R + \gamma \max_a Q(s',a) - Q(s,A) \right]$

$$S' \leftarrow S$$

شکل ۳۴- سودوکد الگوریتم ارائه شده برای MB-learning

برای پیاده‌سازی این الگوریتم از همان پیاده‌سازی الگوریتم Q-learning استفاده کردیم که جزئیات آن را می‌توانید در اینجا: ۱۵ مشاهده کنید. اما به جای قانون بروزرسانی  $Q$ -value قبلی، این‌بارتابع

را قرار دادیم که درست مشابه توضیحات قبلی کار می‌کند و پیاده‌سازی آن به صورت زیر است.

```
def update_q(self, state, next_state, action, reward):
    states, probs, _ = self.environment.possible_consequences(action, state)
    other_states = sum([p * sum([self.b_policy[s][a] * self.Q[(s, a)] for a in self.actions]) \
                        for s, p in zip(states, probs) if s != next_state])
    current_state = [p for s, p in zip(states, probs) if s == next_state][0] * reward
    next_state_return = sum([self.b_policy[next_state][a] * self.Q[(next_state, a)] for a in self.actions])
    self.Q[(state, action)] += self.alpha * (other_states + self.discount * next_state_return - self.Q[state, action])
```

شکل ۳۵- کد به روز سانی  $q$ -value ها مطابق سودوکد شکل ۳۴

## نتایج

در طی ۲۰۰۰ اپیزود، سیاست یافته شده توسط این الگوریتم به صورت زیر است:

0.000	0.001	0.332	0.746
0.696	0.001	0.143	0.998
0.703	0.001	0.001	0.001
0.861	0.401	0.128	0.000

→|↓|←|→  
→|↑|↓|↓  
→|→|↓|↓  
→|→|→|↑

شکل ۳۶- سیاست پیدا شده توسط الگوریتم ارائه شده

## روند اجرای کد پیاده‌سازی

برای اجرای این بخش، فایل `MLearning.ipynb` را اجرا کنید.