



دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر



گزارش تمرین شماره 4  
گروه ...  
درس یادگیری تعاملی  
**پاییز 1400**

نام و نام خانوادگی	مهسا تاجیک
شماره دانشجویی	810198126

## فهرست

3.....	چکیده
4.....	سوال 1 - سوال پیادهسازی
7.....	سوال 2 - سوال پیادهسازی
28.....	سوال 3 - سوال پیادهسازی

## چکیده

---

در این تمرین به پیاده سازی و بررسی الگوریتم های model based و model free و مقایسه ای آن ها پرداخته می شود.

# سوال 1 - سوال پیاده‌سازی

## هدف سوال

در این بخش به پیاده سازی الگوریتم های value iteration که از دسته الگوریتم های model-based می باشد، می پردازیم.

## توضیح پیاده سازی

### 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:

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

until  $\Delta < \theta$

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

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

برای پیاده سازی از روش داینامیک پروگرمینگ استفاده میکنیم و در ابتدا یک تابع با نام possible\_consequences\_val\_iter را به کلاس محیط اضافه میکنیم که مشابه همان تابع است با این تفاوت که مقدار پاداش به ازای یک اکشن را در آن محاسبه میکنیم زیرا برای این الگوریتم نمیتوانیم از تابع step که استیت بعدی را بصورت رندوم انتخاب میکند استفاده کنیم و به پاداش تمام استیت های ممکن بعدی نیاز داریم.

```
def possible_consequences_val_iter(self,action:int,state_now=None):
    if state_now==None:
        state_now = self.state
    state = [state_now[0],state_now[1]]
    states, probs = self.states_transitions(state, action)
    aa = np.array(states)
    fail_probs = self.map[(aa[:,0]),(aa[:,1])]
    dones = np.sum(aa == 3, axis = 1) == 2
    reward=np.zeros(len(probs))
    for p in range(len(probs)):
        if state == states[p]:
            r=0
        else:
            r=-1
        if dones[p]:
            r += 50
        elif np.random.rand()< fail_probs[p]:
            r -= 10
        reward[p] = r
    return states, probs, fail_probs,dones,reward
```

سپس در تابع `value_iteration` به ازای تمام استیت ها مقدار استیت بعدی و سیاستی که پیش میگیریم با استفاده از تابع `bellman_equation` محاسبه شده و تا زمانیکه به تفاوت مقدار قبلی و فعلی استیت ها کمتر از دقت در نظر گرفته شده یا همان تنا نشده اینکار را ادامه میدهیم.

```
def value_iteration(self):
    while True:
        max_value_change = 0
        for i in range(4):
            for j in range(4):
                old_value = self.V.copy()
                new_state_value,p = self.bellman_equation([i,j], self.V[i, j])
                self.V[i, j] = new_state_value
                self.policy[i][j][p] = 1
        max_value_change = abs(old_value - self.V).max()
        if max_value_change < self.theta:
            break
    self.policy = [[[0 for _ in range(4)] for _ in range(4)] for _ in range(4)]
    for i in range(4):
        for j in range(4):
            self.state = [i,j]
            new_state_value,p = self.bellman_equation([i,j], self.V[i, j])
            self.policy[i][j][p] = 1
    print('self_policy',self.policy)
    return self.V[i, j], self.policy
```

در تابع `bellman_equation` روی تمام اکشن ها و همه ای استیت های ممکنی که به ازای انجام هر اکشن میرویم ، `value` را طبق رابطه  $Q(s,a) = \sum_{s'} P(s' | s, a) R(s') + \gamma \sum_{a'} P(a' | s, a) Q(s', a')$  حساب میکنیم و اکشنی که بیشترین مقدار را دارد، بعنوان سیاست بهتر در این استیت انتخاب میشود و مقدارش هم بعنوان `value` جدید اپدیت میشود.

```
def bellman_equation(self, state, action):
    returns = []
    for a in action_name:
        q=0
        (states, probs, fail_probs, dones, reward )= self.environment.possible_consequences_val_iter(a,state)
        for k in range(len(probs)):
            q += (probs[k]*(reward[k]+self.discount*self.V[states[k][0],states[k][1]]))
        returns.append(q)
    returns = np.array(returns)
    max_element = np.amax(returns)
    p = np.argmax(returns)
    return max_element, p
```

## نتایج

نتایج به ازای  $\theta = 0.00001$  و  $\gamma = 0.9$  گزارش شده است.

### Policy:

```
[[[0, 0, 1, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0]],
 [[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 1, 0, 0]],
 [[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 1, 0, 0]],
 [[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 1, 0, 0]]])
```

مسیر بهینه از استیت (0) بصورت راست،پایین،راست،پایین،پایین میباشد.

### Values:

```
[[253.99245014 284.41919708 320.04992223 358.91787519]
 [283.6112898 319.48549833 360.30203918 404.81774731]
 [318.62905194 350.28178611 405.78852192 457.39383085]
 [340.45411406 394.06471234 456.63502645 460.79754024]]
```

### Q-Values:

```
order of actions is:left,down,right,up
Q( 0 0 )= [0, 537.603739943406, 538.4116472271961, 0]
Q( 0 1 )= [538.4116472271961, 603.9046954171945, 604.4691193116741, 0]
Q( 0 2 )= [604.4691193116741, 680.3519614049369, 678.9677974213323, 0]
Q( 0 3 )= [678.9677974213323, 763.735622503974, 0, 0]
Q( 1 0 )= [0, 602.2403417353066, 603.0967881334044, 537.603739943406]
Q( 1 1 )= [603.0967881334044, 669.767284440063, 679.7875375104572, 603.9046954171945]
Q( 1 2 )= [679.7875375104572, 766.0905610925839, 765.1197864875785, 680.3519614049369]
Q( 1 3 )= [765.1197864875785, 862.2115781596592, 0, 763.735622503974]
Q( 2 0 )= [0, 659.0831659965713, 668.9108380419084, 602.2403417353066]
Q( 2 1 )= [668.9108380419084, 744.3464984462004, 756.0703080221328, 669.767284440063]
Q( 2 2 )= [756.0703080221328, 862.4235483653189, 863.1823527646648, 766.0905610925839]
Q( 2 3 )= [863.1823527646648, 918.1913710857376, 0, 862.2115781596592]
Q( 3 0 )= [0, 0, 734.5188264008633, 659.0831659965713]
Q( 3 1 )= [734.5188264008633, 0, 850.6997387893864, 744.3464984462004]
Q( 3 2 )= [850.6997387893864, 0, 917.4325666863917, 862.4235483653189]
Q( 3 3 )= [917.4325666863917, 0, 0, 918.1913710857376]
```

### روند اجرای کد پیاده‌سازی

کد مربوط به این قسمت، در بخش VALUE\_ITERATION ، در نوبوک ارسال شده قابل اجراست.

## سوال 2 - پیاده سازی

### هدف سوال

در این سوال به پیاده سازی الگوریتم های model-free می پردازیم و در هر زیربخش توضیحات مربوط به هر الگوریتم و نتایج آن آمده است.

### زیر بخش 1

On-policy first-visit MC control (for  $\varepsilon$ -soft policies), estimates  $\pi \approx \pi_*$

Algorithm parameter: small  $\varepsilon > 0$

Initialize:

$\pi \leftarrow$  an arbitrary  $\varepsilon$ -soft policy

$Q(s, a) \in \mathbb{R}$  (arbitrarily), for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$  empty list, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following  $\pi$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair  $S_t, A_t$  appears in  $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$ :

Append  $G$  to  $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$  average( $Returns(S_t, A_t)$ )

$A^* \leftarrow \arg \max_a Q(S_t, a)$  (with ties broken arbitrarily)

For all  $a \in \mathcal{A}(S_t)$ :

$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$

### توضیح پیاده سازی

در این قسمت الگوریتم on-policy monte-carlo پیاده سازی شده است. سیاستی که برای عامل درنظر گرفتیم اپسیلون گریدی است که خود یک سیاست نرم است. برای ذخیره  $Q(s,a)$  یک لیست با مقدار اولیه  $4*4*4$  با مقدار اولیه  $0$  صفر تعریف شده و مقادیر return محاسبه شده در یک لیست  $4*4*4$  با مقدار اولیه  $0$  یعنی 4000 اپیزود، اجرا میشوند: ابتدا اپیزود در تابع generate\_episode تولید میشود و یک اپیزود که شامل استیت های اپیزود، اکشنی که در این استیت انتخاب میشود و پاداش دریافتی است، بر میگرداند. سپس از انتهای اپیزود شروع به محاسبه  $return$  به روش first-visit کرده و چک میکنیم که هر استیت و اکشن قبل از طول اپیزود دیده شده یا خیر. اگر دیده نشده بود مقدار  $return$  اضافه میشود و

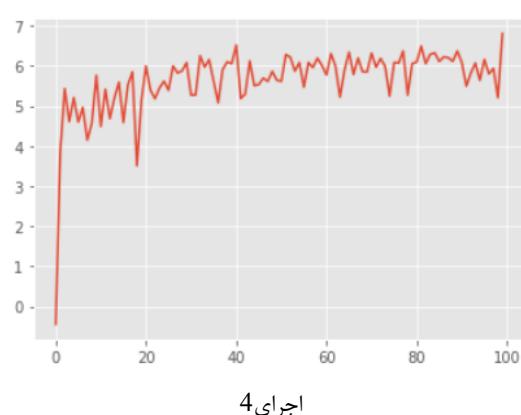
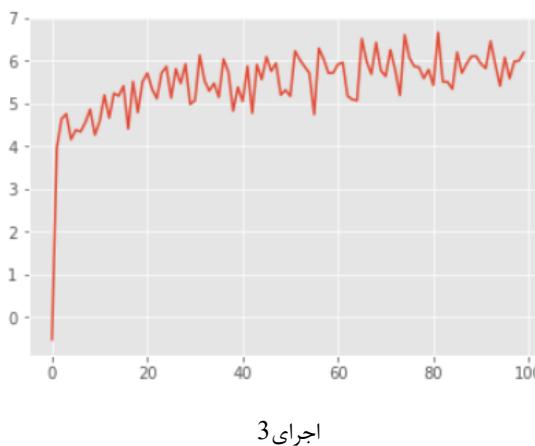
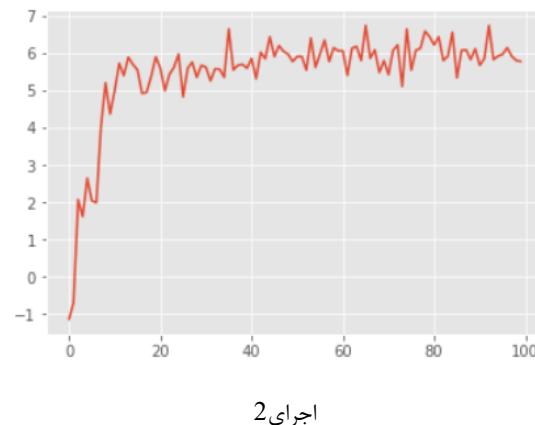
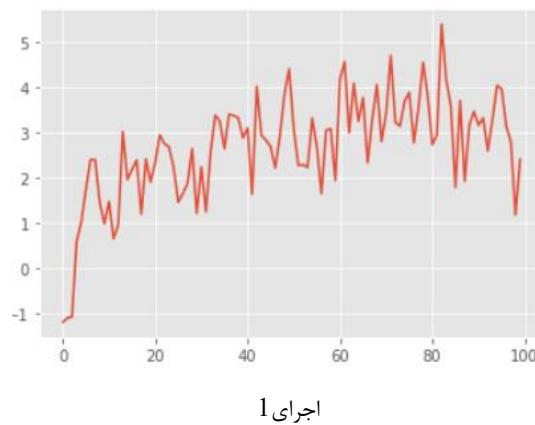
میانگین روی return ها گرفته میشود و اکشنی که بیشترین Q را در آن استیت دارد بعنوان اکشن بهینه انتخاب میکنیم و سیاست طبق سیاست اپسیلون گریدی اپدیت میشود.

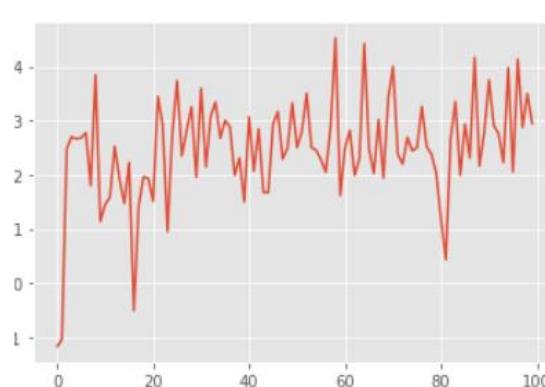
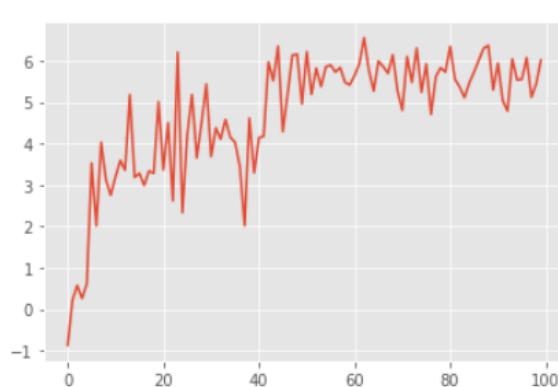
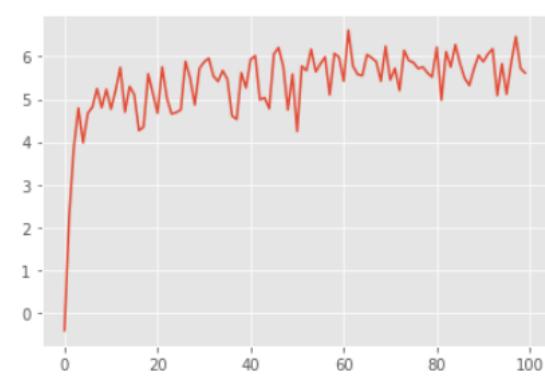
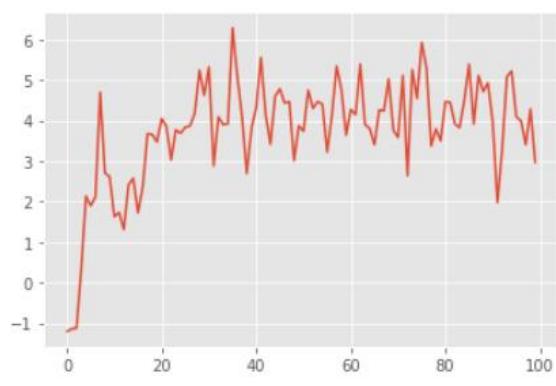
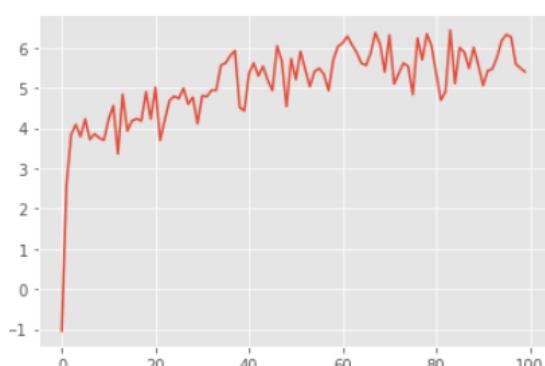
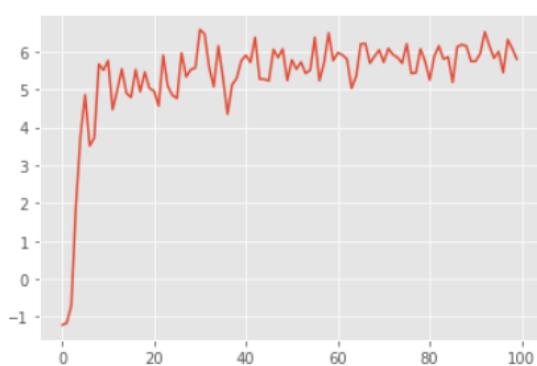
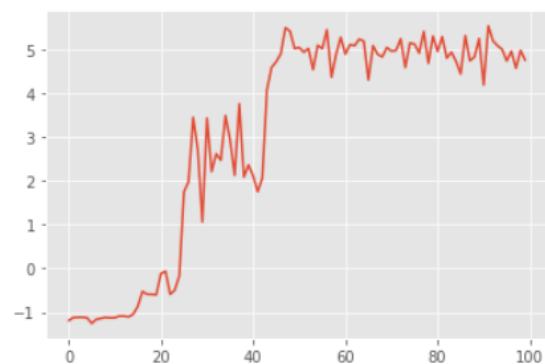
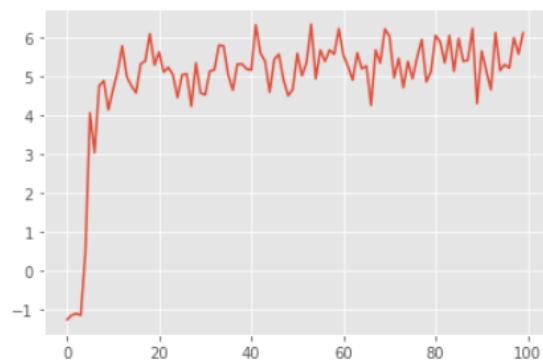
میانگین پاداش ها در هر اپیزود را حساب کرده و با پنجره متحرک به سایز 40 و استراید 40، نمودار میانگین پاداش ها را به ازای هر اپیزود رسم میکنیم.

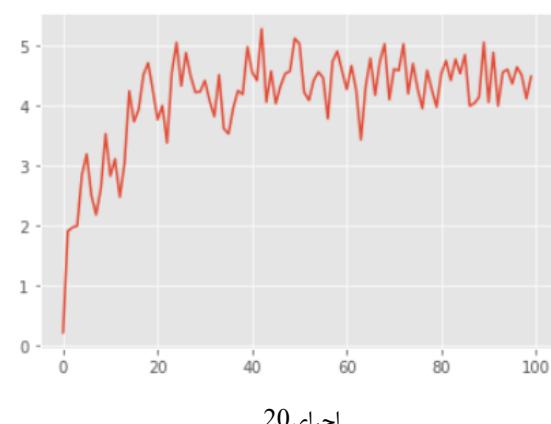
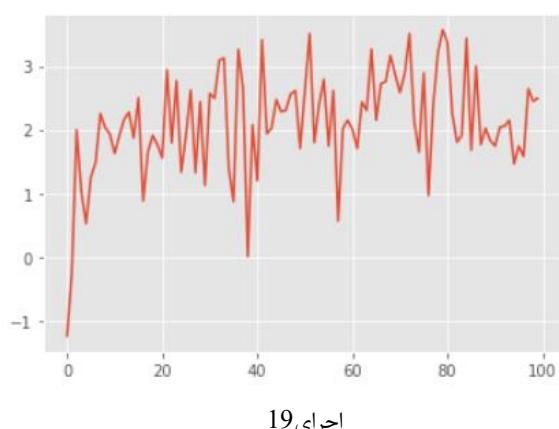
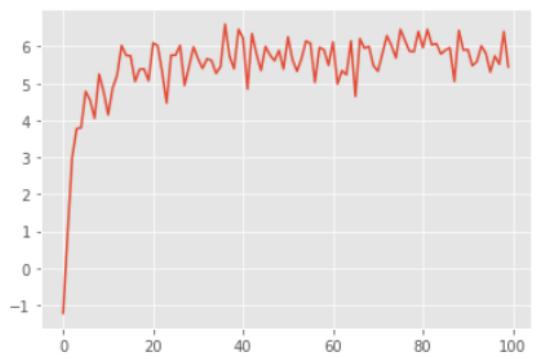
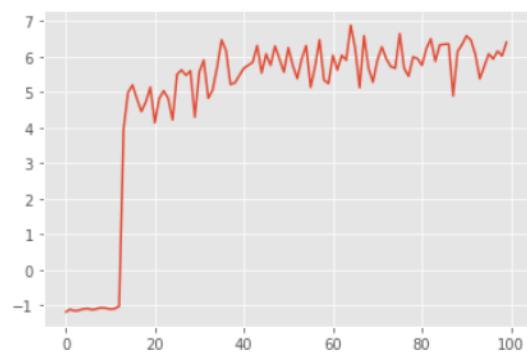
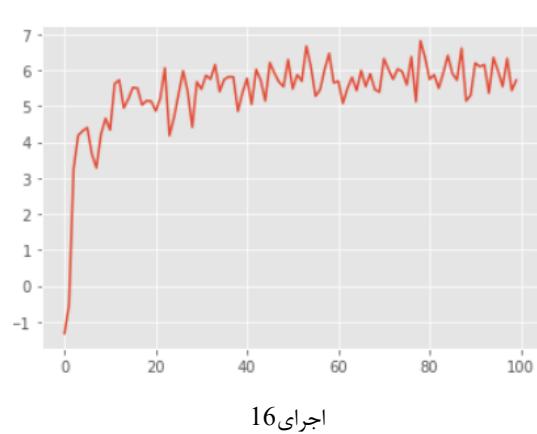
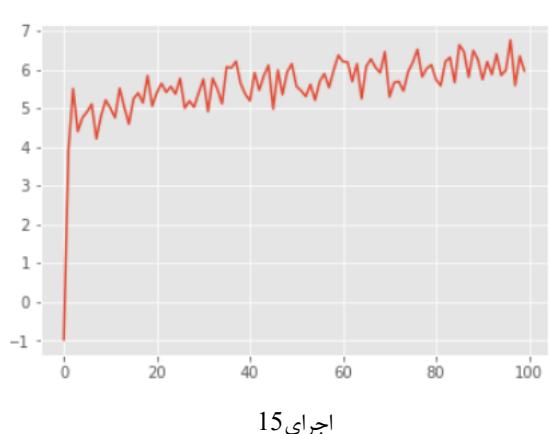
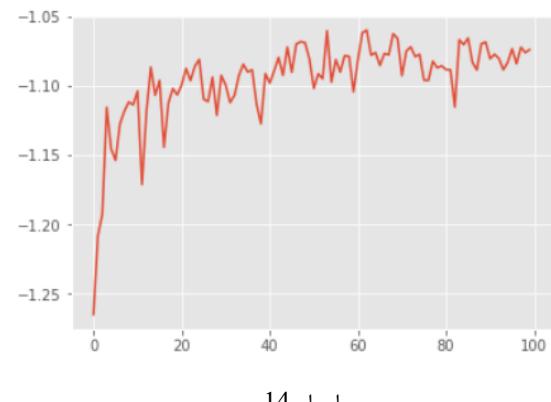
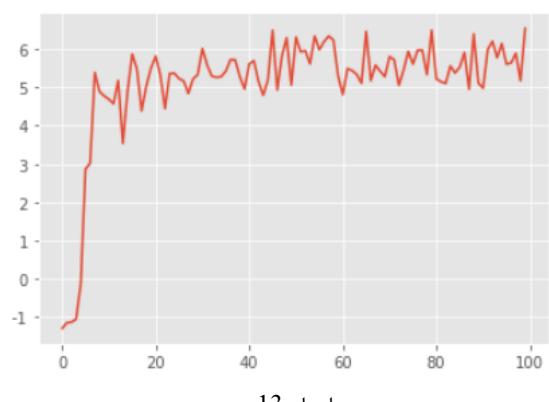
## نتایج

مقدار discount = 0.9 در نظر گرفته شده است. و در ادامه نتایج را به ازای اپسیلون کاهشی و اپسیلون ثابت به ازای 20 بار اجزا میبینیم.

اپسیلون کاهشی با مقدار اپسیلون اولیه 0.1 و decay factor 0.999 برابر با







پالیسی همگراشده:

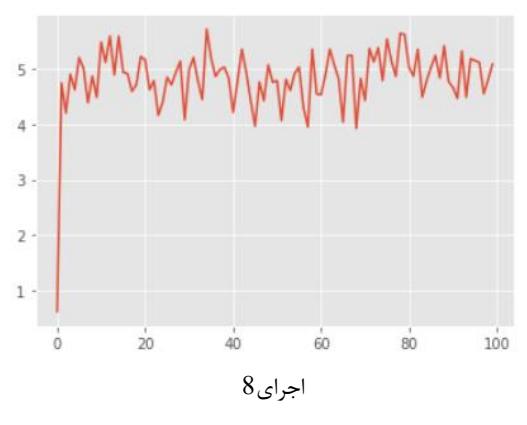
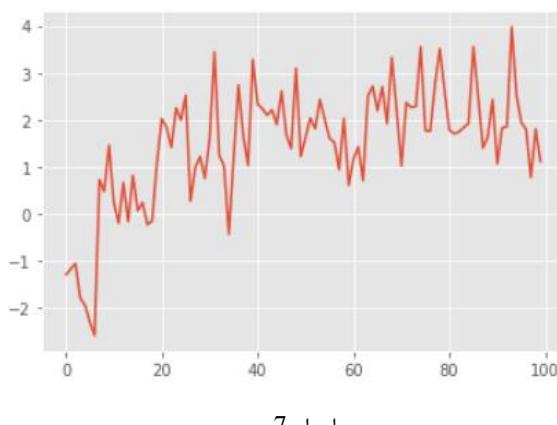
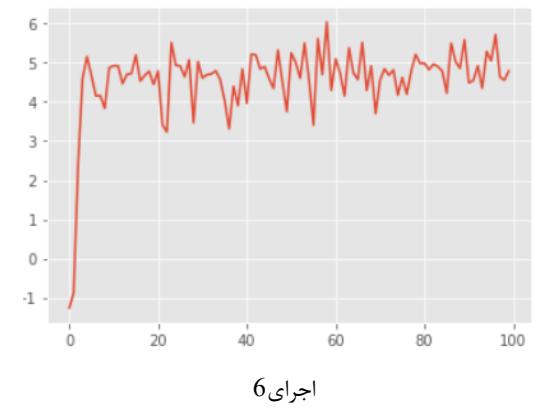
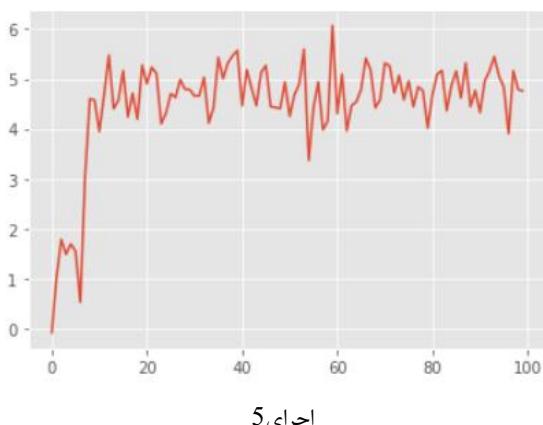
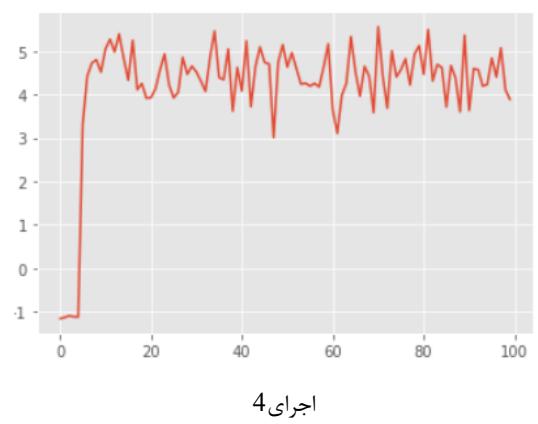
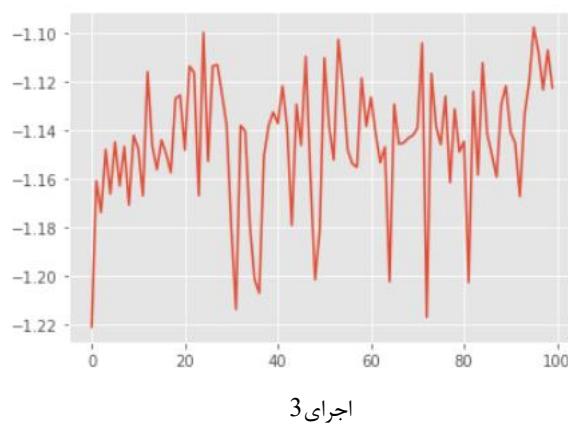
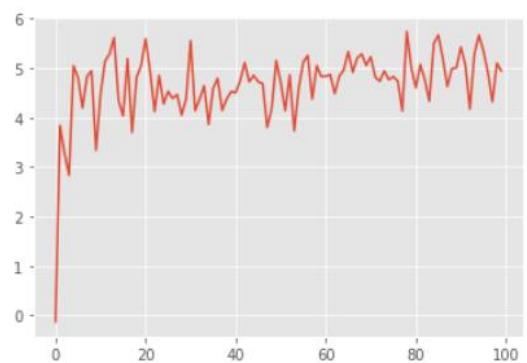
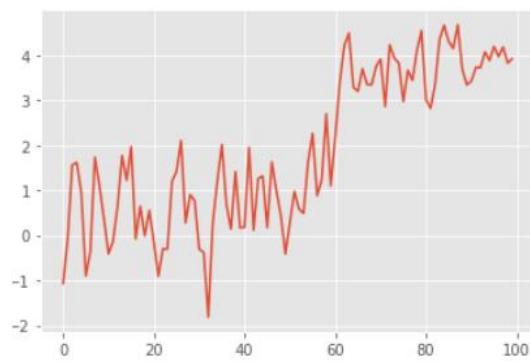
```
[[[0.00045743292861585193, 0.00045743292861585193, 0.9986277012141525,  
0.00045743292861585193],  
[0.0004578908194352872, 0.9986263275416941, 0.0004578908194352872,  
0.0004578908194352872],  
[0.0004643496269559968, 0.998606951119132, 0.0004643496269559968,  
0.0004643496269559968],  
[0.0004804171486594473, 0.9985587485540216, 0.0004804171486594473,  
0.0004804171486594473]],  
  
[[0.00046295797065966, 0.00046295797065966, 0.998611126088021,  
0.00046295797065966],  
[0.0004578908194352872, 0.0004578908194352872, 0.9986263275416941,  
0.0004578908194352872],  
[0.0004578908194352872, 0.0004578908194352872, 0.9986263275416941,  
0.0004578908194352872],  
[0.0004578908194352872, 0.9986263275416941, 0.0004578908194352872,  
0.0004578908194352872]],  
  
[[0.0004643496269559968, 0.0004643496269559968, 0.998606951119132,  
0.0004643496269559968],  
[0.00046203251767631135, 0.00046203251767631135, 0.00046203251767631135,  
0.998613902446971],  
[0.9985645049180605, 0.0004784983606465133, 0.0004784983606465133,  
0.0004784983606465133],  
[0.0004578908194352872, 0.9986263275416941, 0.0004578908194352872,  
0.0004578908194352872]],  
  
[[0.9767128146667374, 0.007762395110875385, 0.007762395110875385,  
0.007762395110875385],  
[0.004124623422417883, 0.9876261297327464, 0.004124623422417883,  
0.004124623422417883],  
[0.9985659404131424, 0.0004780198622858668, 0.0004780198622858668,  
0.0004780198622858668],  
[0.9986277012141525, 0.00045743292861585193, 0.00045743292861585193,  
0.00045743292861585193]]]
```

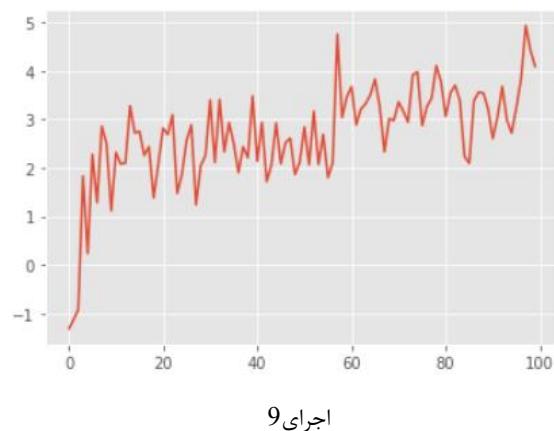
مسیر: راست، پایین، راست، پایین، پایین

زمان اجرا(ثانیه):

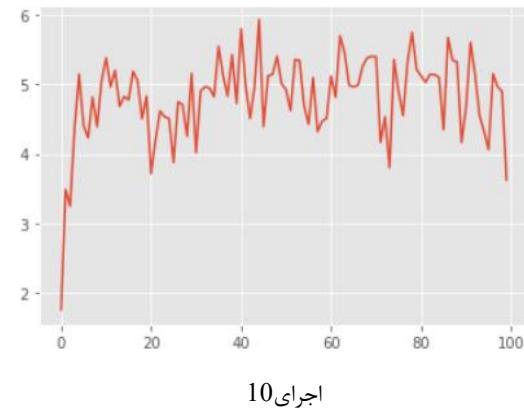
```
run time = 842.6594598293304
```

:0.1 = اپسیلون ثابت

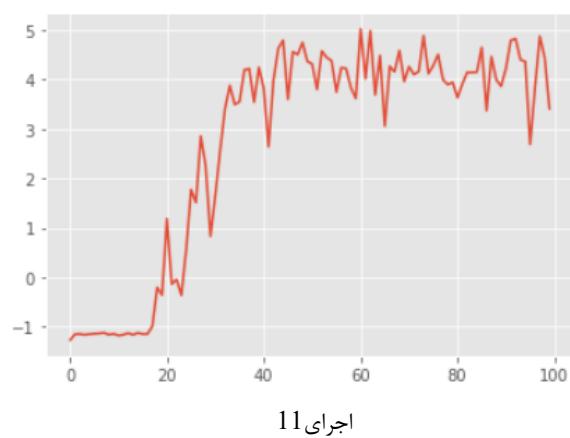




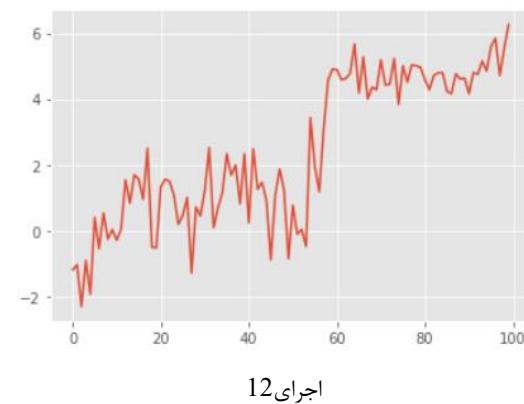
اجرای 9



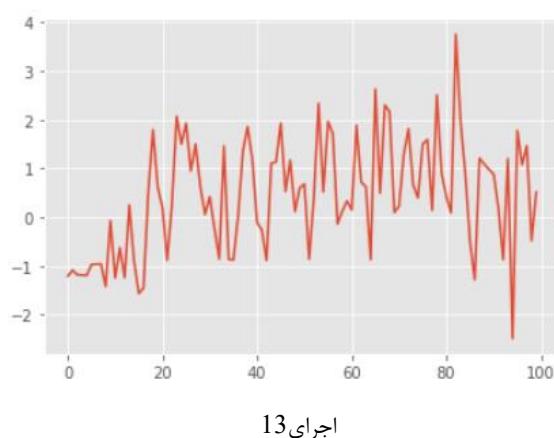
اجرای 10



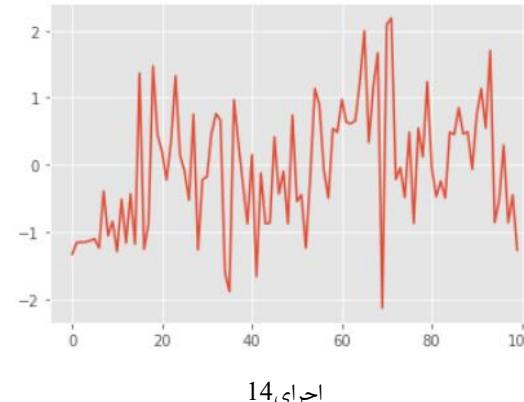
اجرای 11



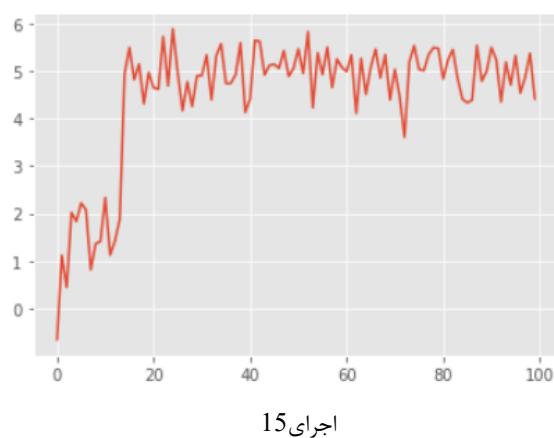
اجرای 12



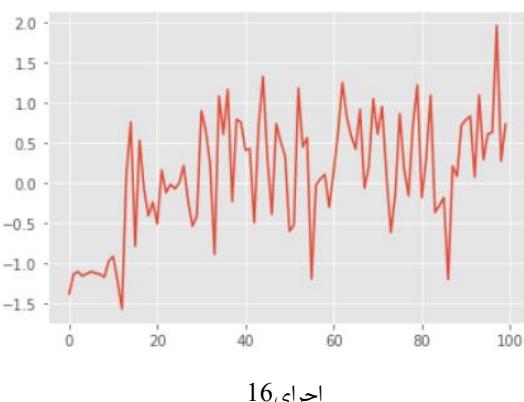
اجرای 13



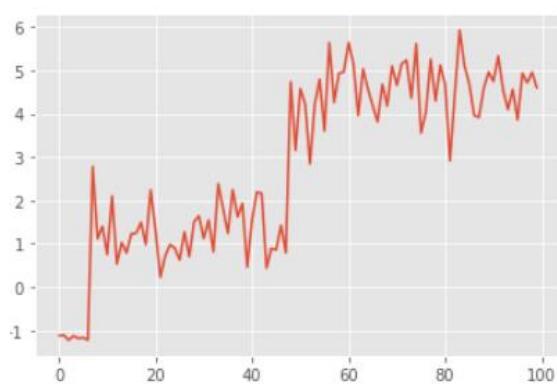
اجرای 14



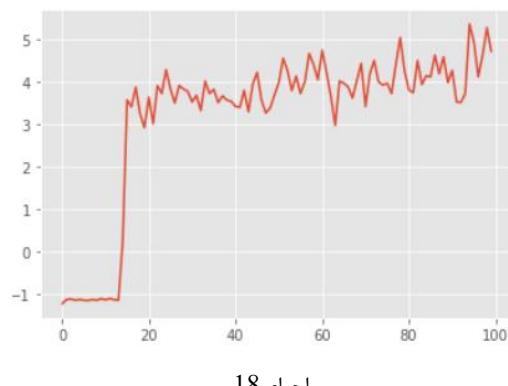
اجرای 15



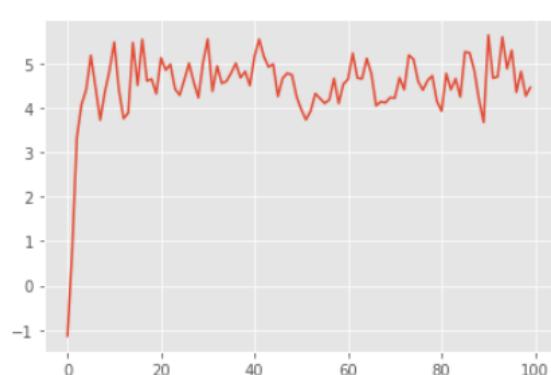
اجرای 16



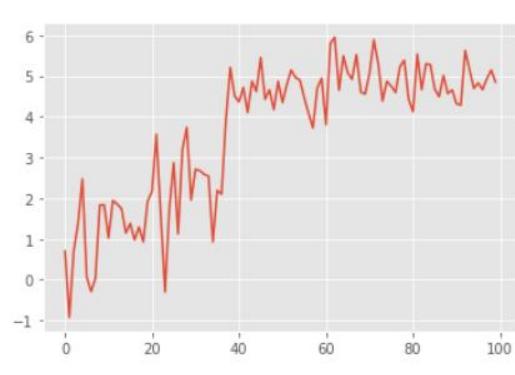
اجرای 17



اجرای 18



اجرای 19



اجرای 20

پالیسی همگرا شده :

```
[[[0.025, 0.925, 0.025, 0.025],
 [0.025, 0.925, 0.025, 0.025],
 [0.025, 0.925, 0.025, 0.025],
 [0.025, 0.925, 0.025, 0.025]],

 [[0.025, 0.025, 0.925, 0.025],
 [0.025, 0.025, 0.925, 0.025],
 [0.025, 0.025, 0.925, 0.025],
 [0.025, 0.925, 0.025, 0.025]],

 [[0.025, 0.025, 0.025, 0.925],
 [0.025, 0.025, 0.925, 0.025],
 [0.025, 0.025, 0.925, 0.025],
 [0.025, 0.925, 0.025, 0.025]],

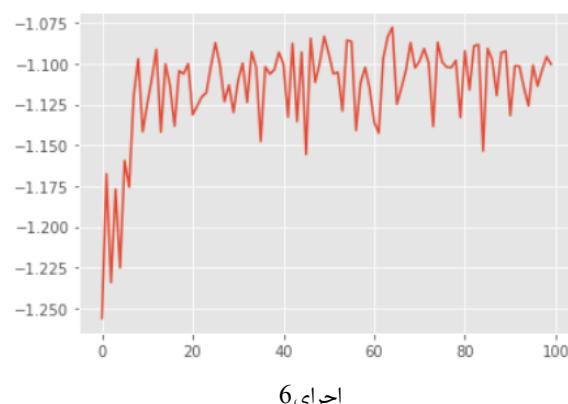
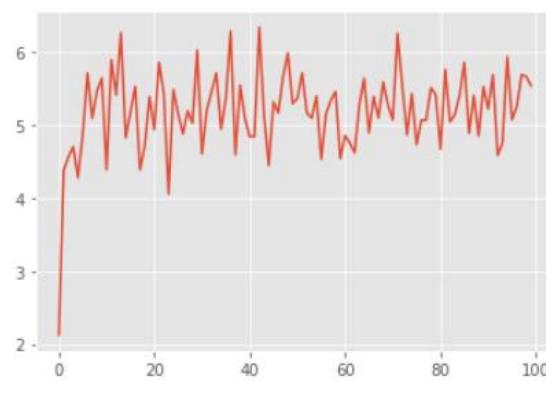
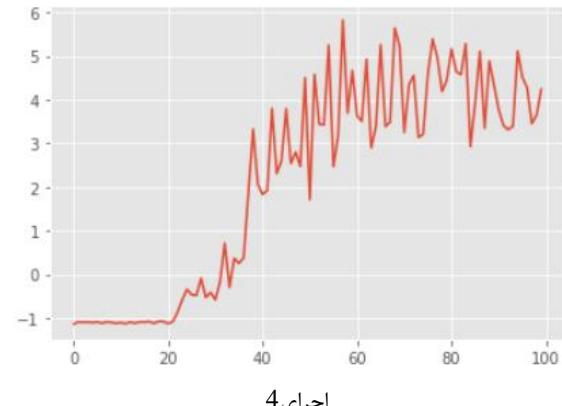
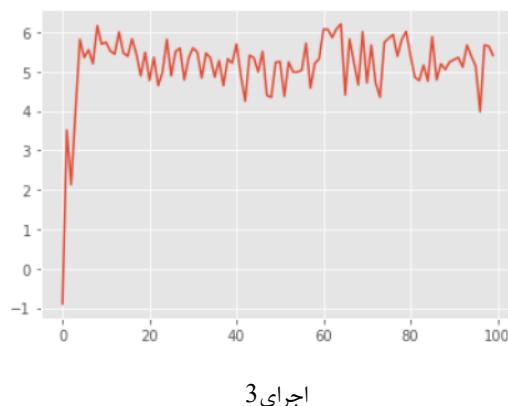
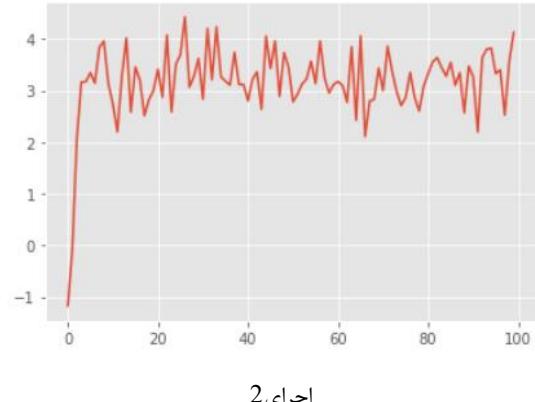
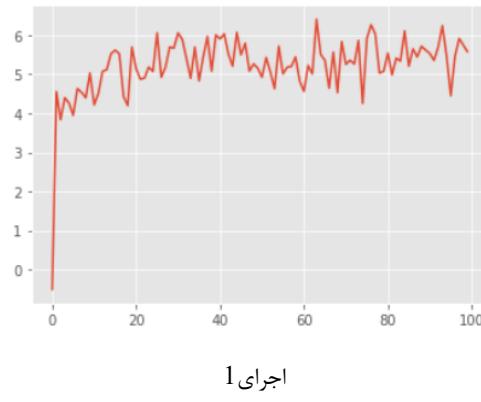
 [[0.025, 0.025, 0.025, 0.925],
 [0.025, 0.925, 0.025, 0.025],
 [0.025, 0.925, 0.025, 0.025],
 [0.025, 0.925, 0.025, 0.025]]]
```

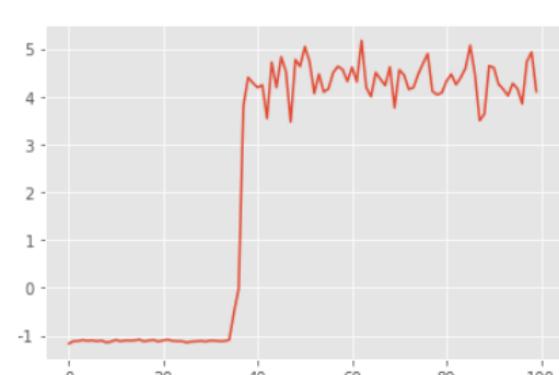
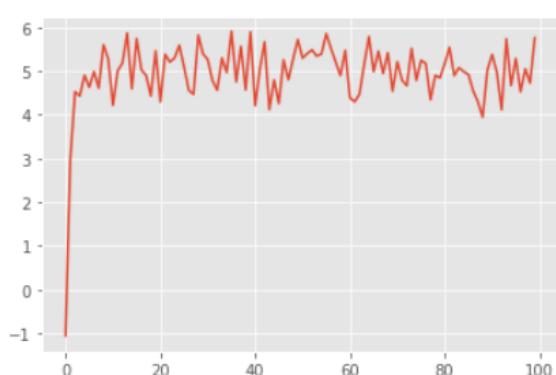
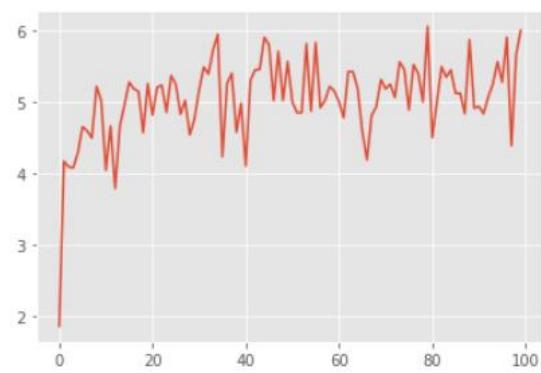
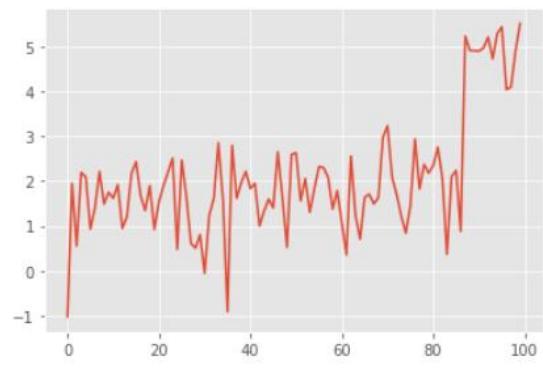
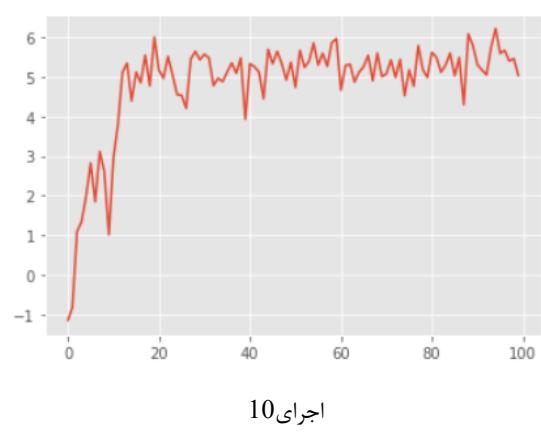
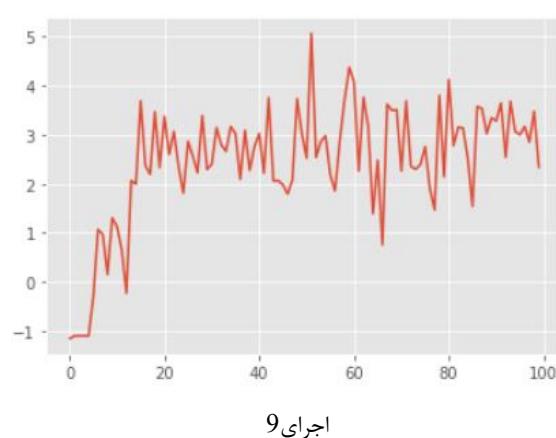
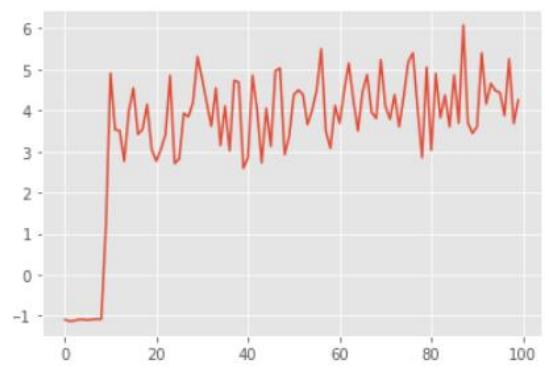
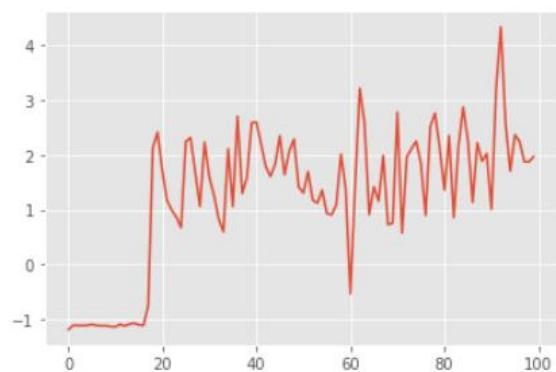
مسیر: پایین، راست، راست، راست، پایین، پایین

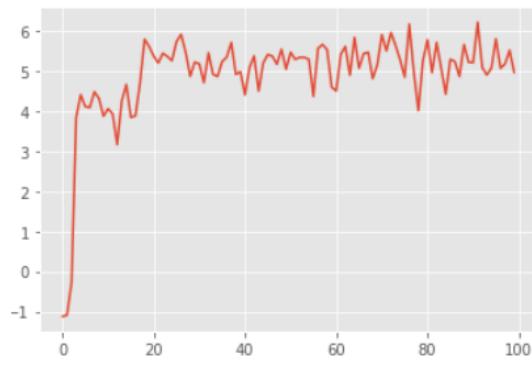
زمان اجرا(ثانیه):

run time = 459.59317994117737

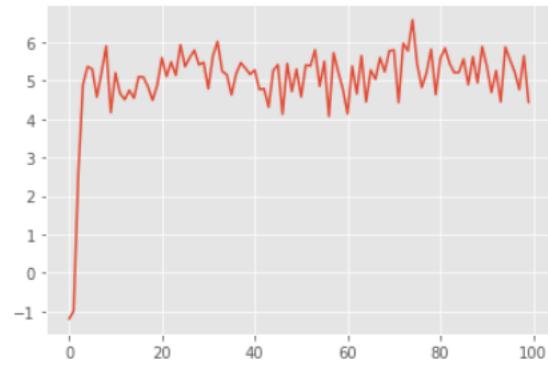
اپسیلوون ثابت = 0.05



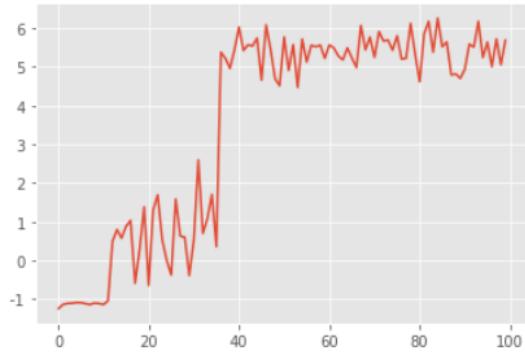




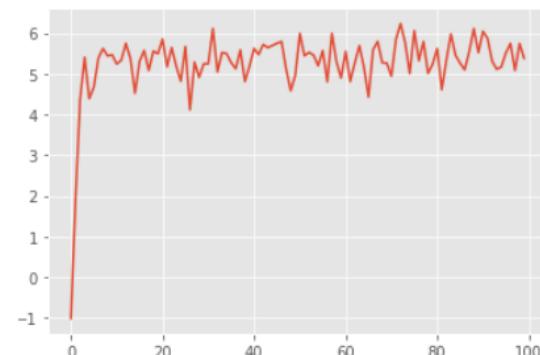
اجرای 15



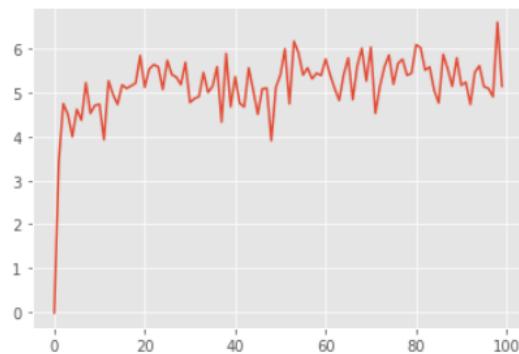
اجرای 16



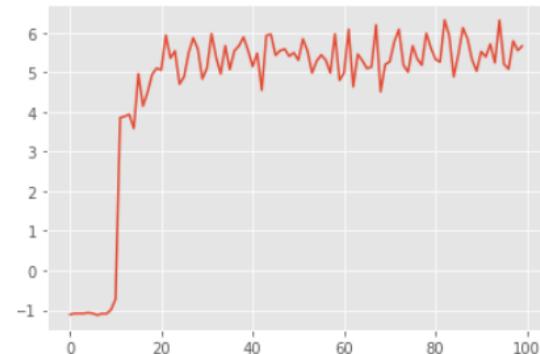
اجرای 17



اجرای 18



اجرای 19



اجرای 20

پالیسی همگرا شده :

```
[[[0.0125, 0.0125, 0.9624999999999999, 0.0125],
 [0.0125, 0.9624999999999999, 0.0125, 0.0125],
 [0.0125, 0.9624999999999999, 0.0125, 0.0125],
 [0.0125, 0.9624999999999999, 0.0125, 0.0125]],

 [[[0.0125, 0.0125, 0.9624999999999999, 0.0125],
 [0.0125, 0.0125, 0.9624999999999999, 0.0125],
 [0.0125, 0.0125, 0.9624999999999999, 0.0125],
 [0.0125, 0.9624999999999999, 0.0125, 0.0125]],

 [[0.0125, 0.0125, 0.9624999999999999, 0.0125],
 [0.0125, 0.0125, 0.9624999999999999, 0.0125],
 [0.0125, 0.0125, 0.9624999999999999, 0.0125],
 [0.0125, 0.9624999999999999, 0.0125, 0.0125]],

 [[0.0125, 0.0125, 0.9624999999999999, 0.0125],
```

```
[0.0125, 0.0125, 0.0125, 0.9624999999999999],
[0.0125, 0.0125, 0.9624999999999999, 0.0125],
[0.0125, 0.9624999999999999, 0.0125, 0.0125]],

[[0.0125, 0.0125, 0.0125, 0.9624999999999999],
[0.0125, 0.9624999999999999, 0.0125, 0.0125],
[0.0125, 0.9624999999999999, 0.0125, 0.0125],
[0.0125, 0.9624999999999999, 0.0125, 0.0125]]]
```

مسیر: راست، پایین، راست، پایین، پایین

زمان اجرا(ثانیه):

run time = 799.6453411579132

	time
Epsilon = 0.1 with decay factor 0.999	842.66s
Epsilon = 0.1	459.6s
Epsilon = 0.05	799.65s

همانطور که قابل مشاهده است زمان اجرا وقتی اپسیلون بزرگتر باشد، کوتاه تر بوده و الگوریتم بیشتر گردیدی عمل میکند و اکسپلوریشن کمتری دارد و میانگین پاداش در حالتی که اپسیلون بزرگتر است و همچنین به نسبت حالتی که اپسیلون کاهشی است، کمتر است.

## زیر بخش 2

### 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  $s \in \mathcal{S}^+$ ,  $a \in \mathcal{A}(s)$ , 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

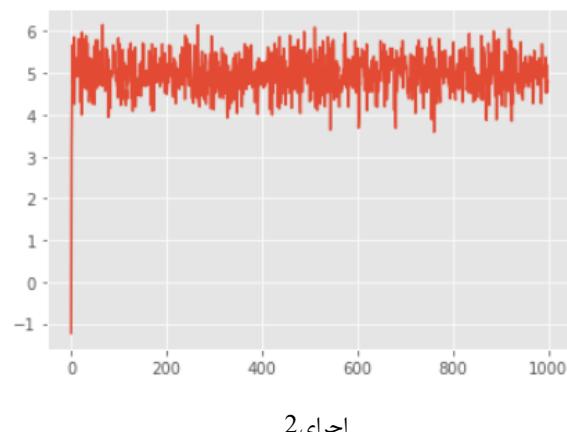
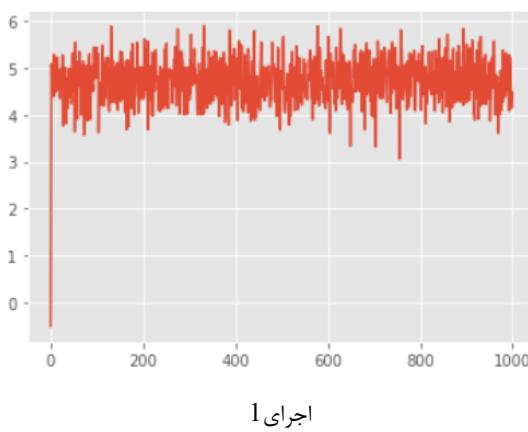
## توضیح پیاده سازی

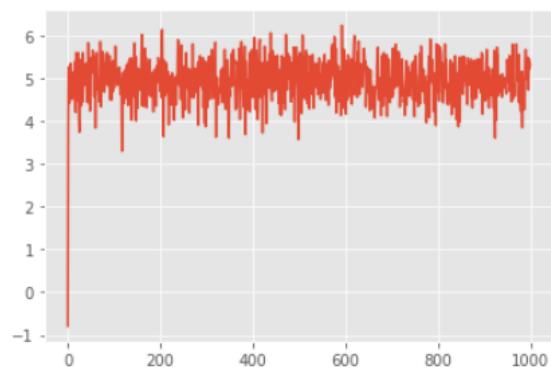
در این قسمت الگوریتم q-learning را پیاده سازی کردیم. در ابتدای هر اپیزود محیط ریست میشود و استیت فعلی به  $(0, 0)$  مقداردهی میشود سپس در هر استپ از اپیزود یک اکشن براساس سیاست اپسیلون گریدی روی  $Q$  ها انتخاب میشود و اکشن با تابع `env.step` به محیط اعمال شده و استیت بعدی و پاداش و فلگ `done` برگردانده میشود. سپس اکشنی که بیشینه  $Q$  را در استیت جدید دارد پیدا کرده و طبق رابطه ای که در الگوریتم بالا دیده میشود،  $Q$  استیت-اکشنی که در آن بودیم، آپدیت میشود و استیت فعلی به استیت جدید تغییر میکند اینکار تا زمانی انجام میشود که به یک ترمینال استیت برسیم.

## نتایج

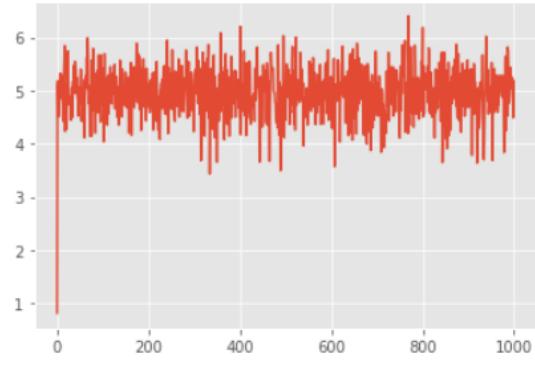
نتایج به ازای اپسیلون 0.1، آلفای کاهشی با  $0.999$  discount factor برابر با 0.9 استیت 40000 اپیزود با سایز پنجره متحرک 40 و استرايد 40 می باشد.

\*\*در ابتدا تعداد اپیزودها مانند حالت قبل(مونت کارلو) روی 4000 اپیزود بود و 704 ثانیه زمان برد که نزدیک به حالت مونت کارلو با اپسیلون ثابت 0.05 بود ولی بدلیل نوسان زیاد نمودار، تعداد اپیزود ها را به 40000 افزایش دادم.

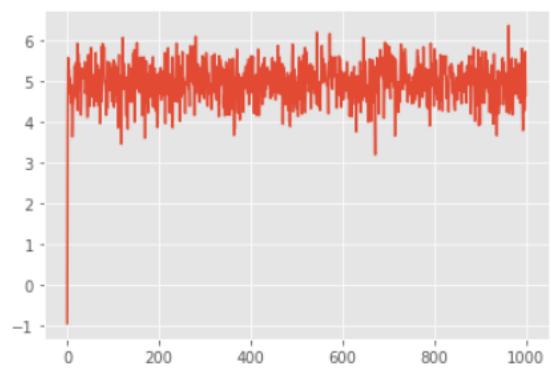




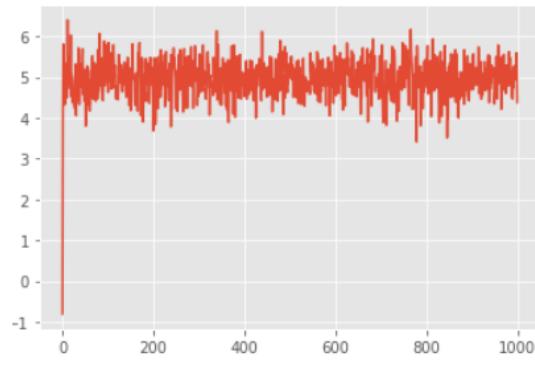
اجرای ۳



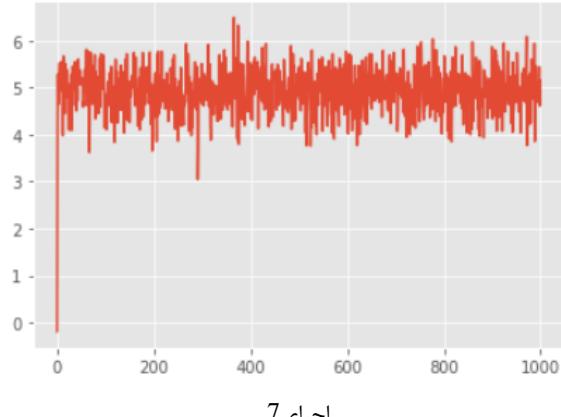
اجرای ۴



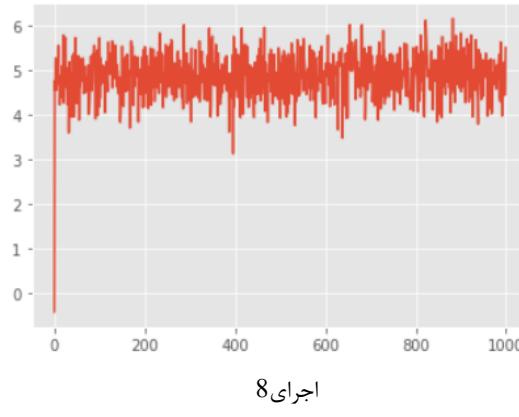
اجرای ۵



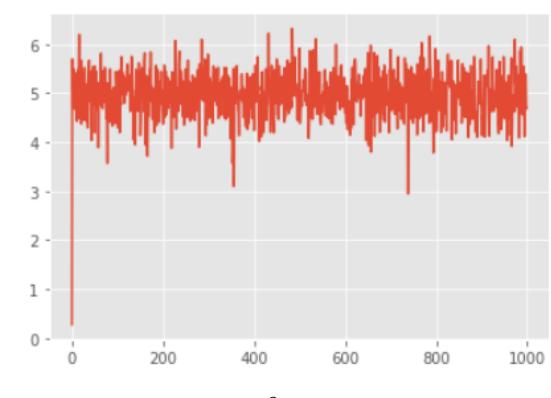
اجرای ۶



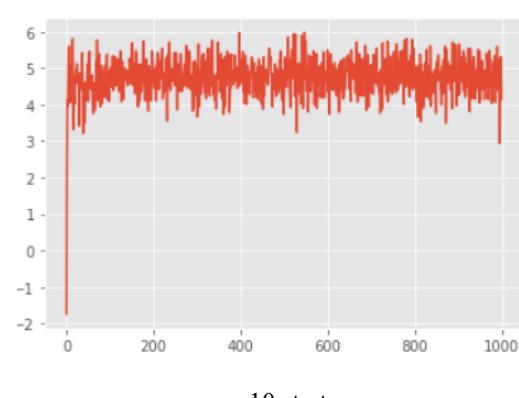
اجرای ۷



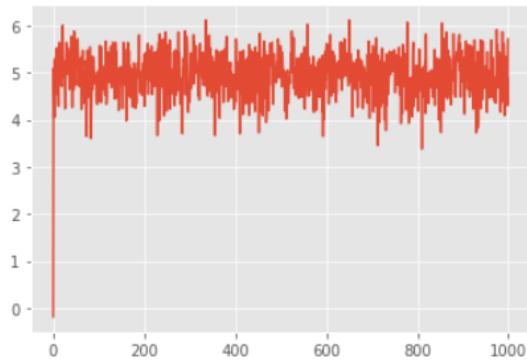
اجرای ۸



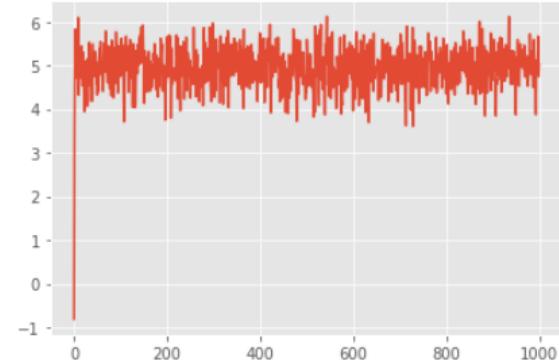
اجرای ۹



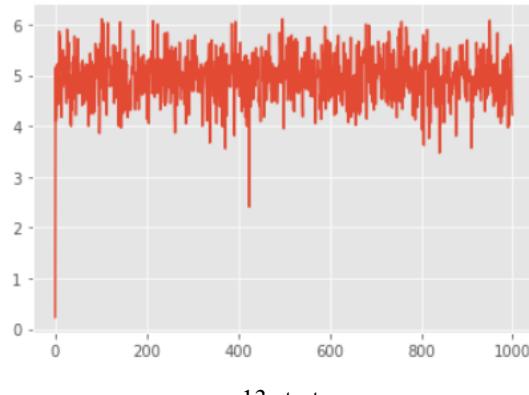
اجرای ۱۰



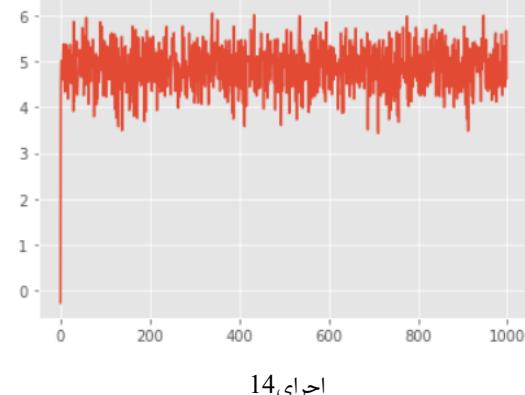
اجرای 11



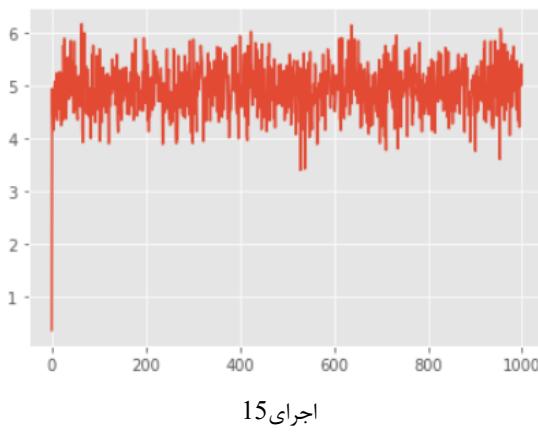
اجرای 12



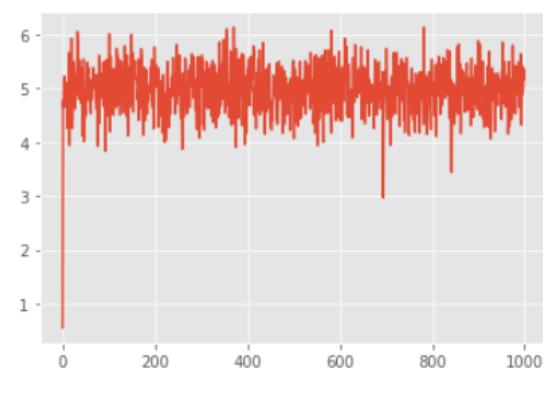
اجرای 13



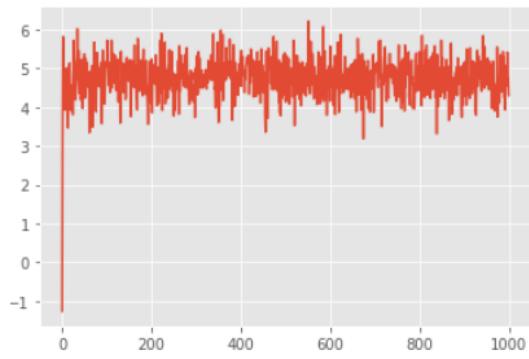
اجرای 14



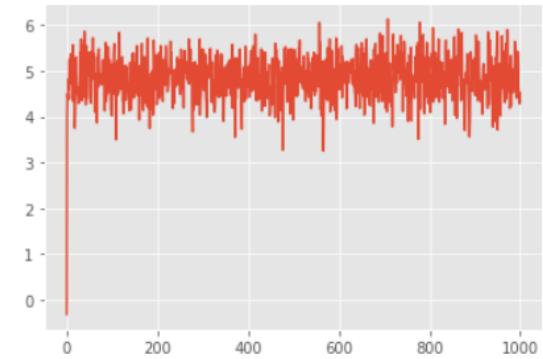
اجرای 15



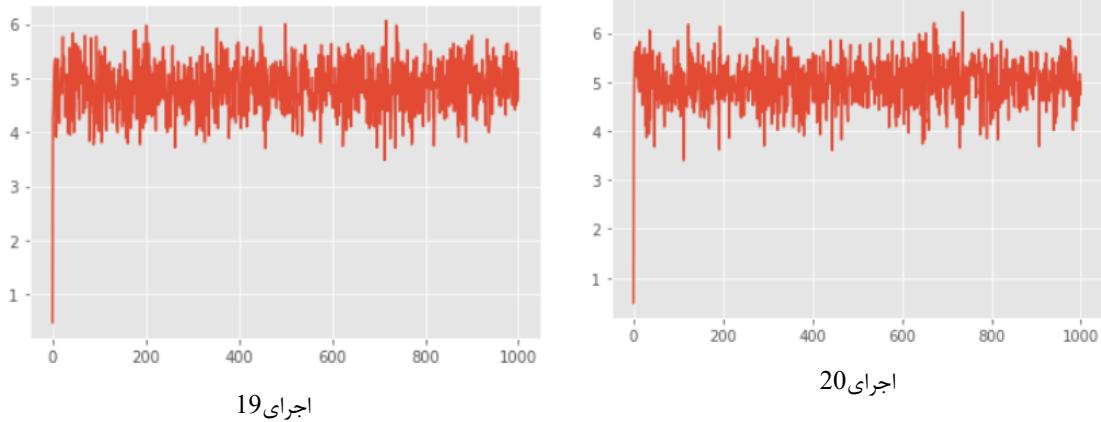
اجرای 16



اجرای 17



اجرای 18



زمان اجرا (ثانیه):

```
run time = 1441.3787972927094
```

در الگوریتم q-learning به نسبت مونت کارلو زمان اجرا کمتر بوده (با در نظر گرفتن این نکته که تعداد اپیزودهایی که در نظر گرفتم در این الگوریتم 10 برابر مونت کارلوست) و میانگین پاداش ها بیشتر است و در نتیجه میزان حسرت کمتر از روش مونت کارلوست.

### زیر بخش 3

#### 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
```

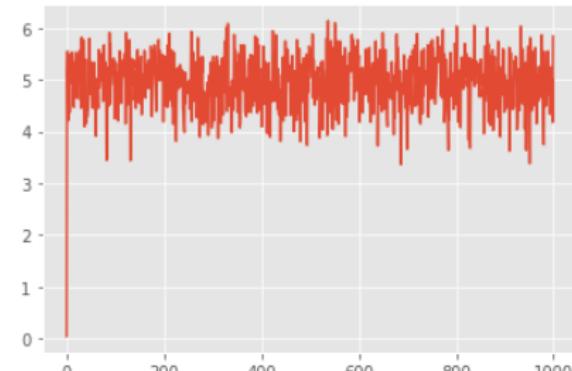
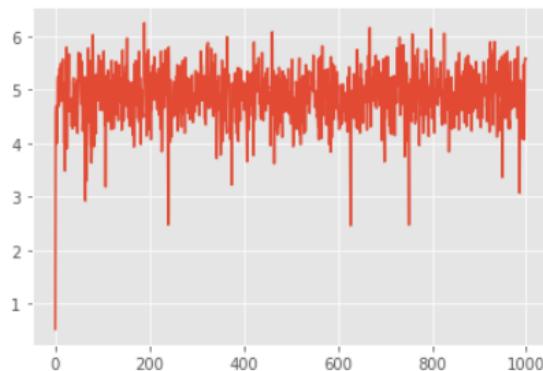
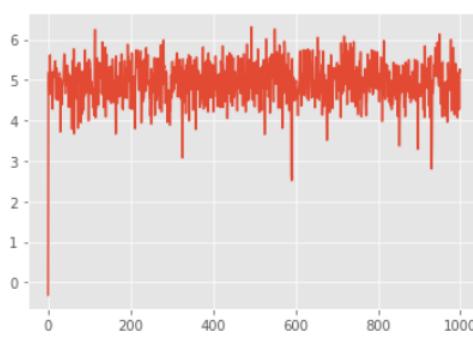
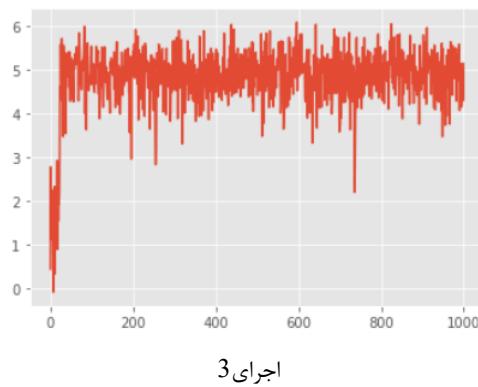
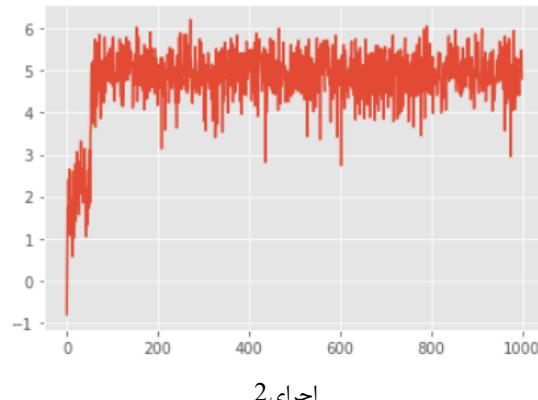
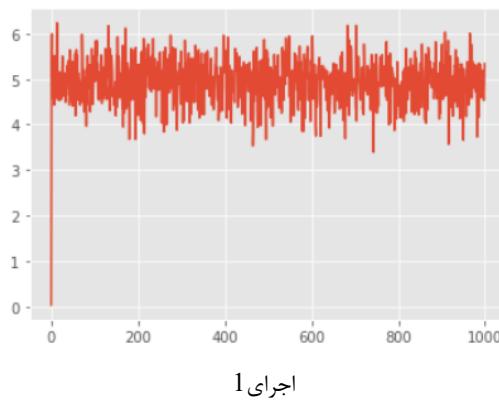
### توضیح پیاده سازی

در این قسمت الگوریتم sarsa را پیاده سازی کردیم. در ابتدای هر اپیزود محیط ریست میشود و استیت فعلی به  $(0, 0)$  مقداردهی میشود و یک اکشن براساس سیاست اپسیلون گریدی نسبت به  $Q(s, a)$  انتخاب میشود. سپس در هر استپ از اپیزود، اکشن با تابع `env.step` به محیط اعمال شده و استیت بعدی و پاداش

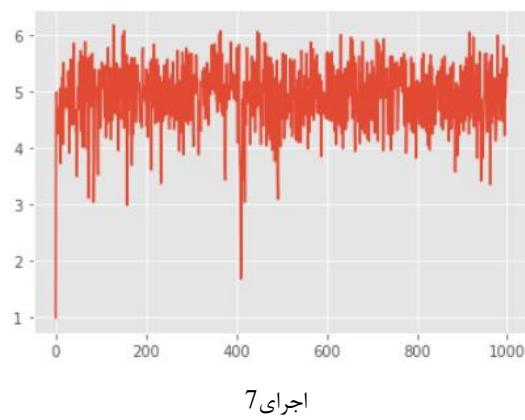
و فلگ done برگردانده میشود. سپس اکشن بعدی براساس سیاست اپسیلون گریدی نسبت به ارزش استیت جدید انتخاب میشود و طبق رابطه ای که در الگوریتم بالا دیده میشود، Q استیت-اکشنی که در آن بودیم، آپدیت میشود و استیت فعلی و اکشنی که در آن انجام دادیم، به استیت جدید و اکشن جدید تغییر میکند. اینکار تا زمانی انجام میشود که به یک ترمینال استیت برسیم.

## نتایج

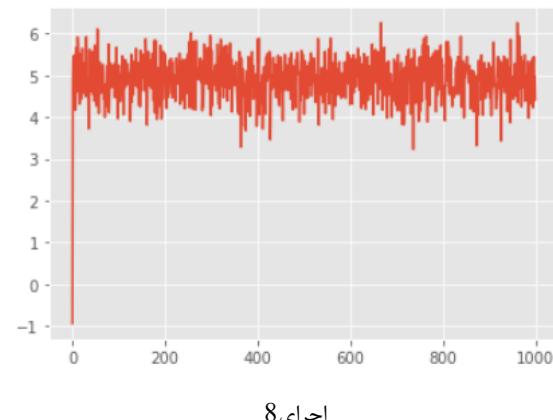
نتایج به ازای آلفا برابر با 0.1 ، اپسیلون 0.1 و 0.9 discount و 40000 اپیزود با سایز پنجره متحرک 40 و استرايد 40 گزارش شده اند.



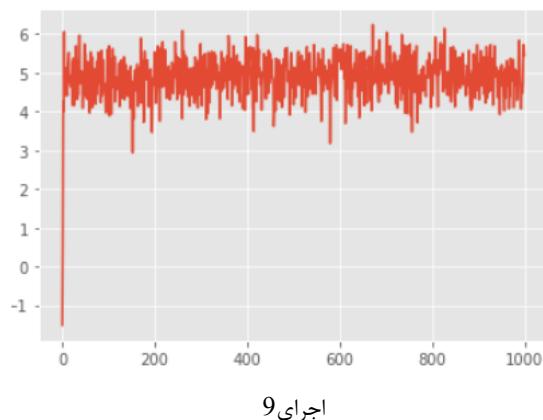
اجرای 5



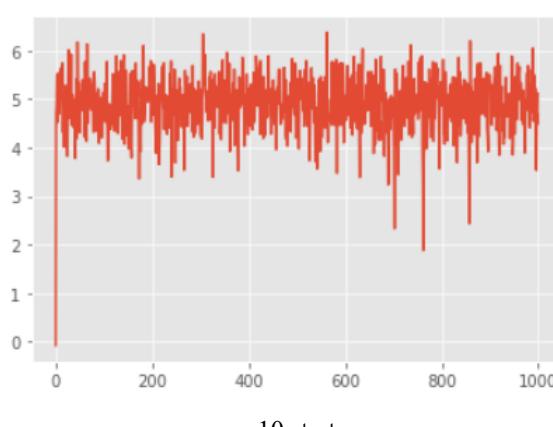
اجرای 6



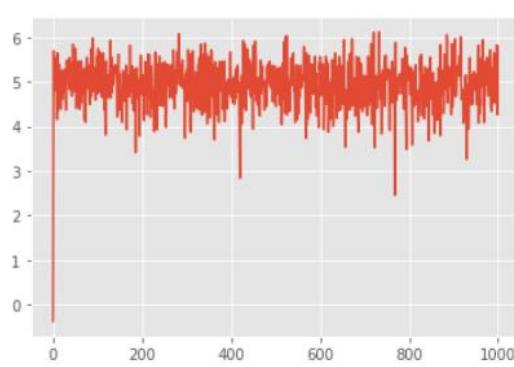
اجرای 7



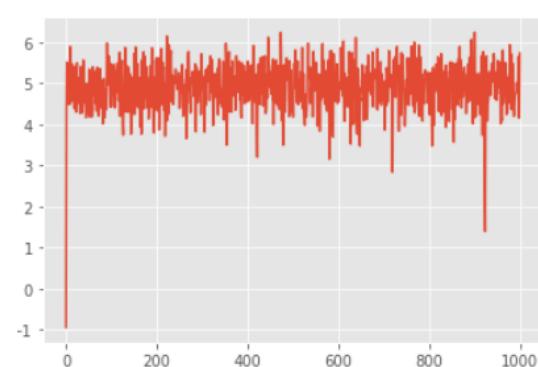
اجرای 8



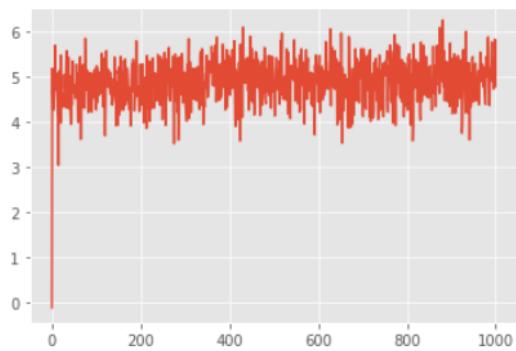
اجرای 9



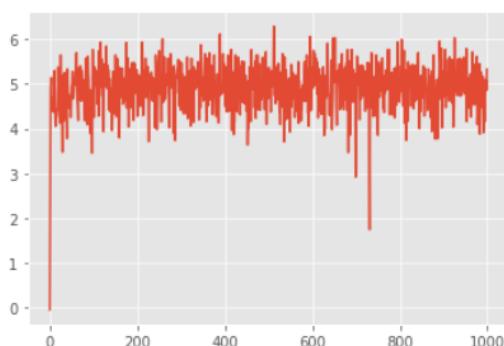
اجرای 10



اجرای 11

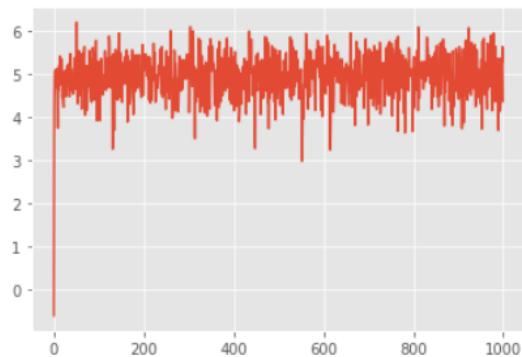


اجرای 12

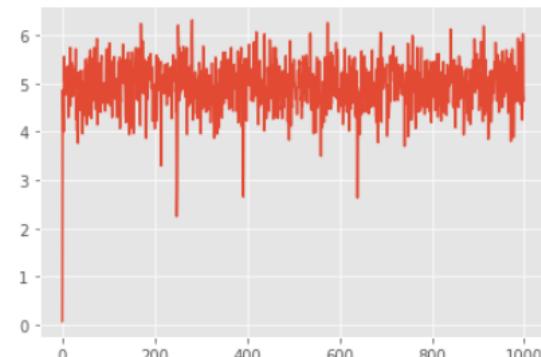


اجرای 13

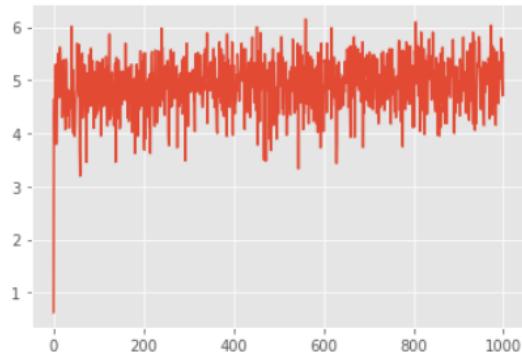
اجرای 14



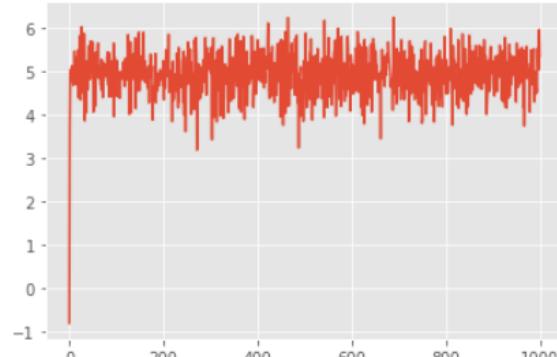
اجرای 15



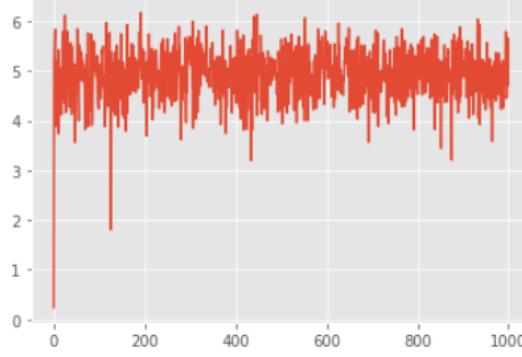
اجرای 16



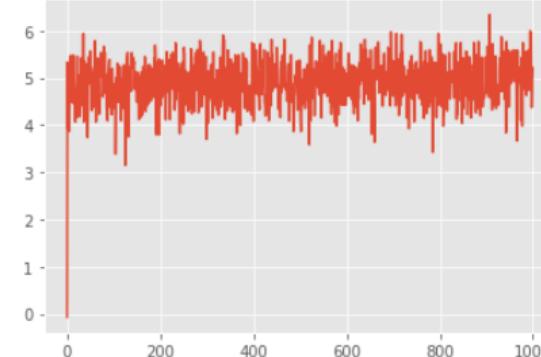
اجرای 17



اجرای 18



اجرای 19



اجرای 20

زمان اجرا(ثانیه) :

```
run time = 1367.797589302063
```

در الگوریتم sarsa، زمان اجرا نسبت به هر دو الگوریتم q-learning و مونت کارلو کمتر بوده و میانگین پاداش ها نزدیک به q-learning و بیشتر از مونت کارلو است بنابراین میزان حسرت نسبت به روش مونت کارلو آن پالیسی، کمتر است.

دلیل اصلی این اتفاق ، آپدیت شدن  $-Q$  value ها پس از هر گام از اپیزود است ، در صورتی که در مونت کارلو ، اپیدا اپیزود را تا آخر میرویم و سپس  $Q$ -value ها را آپدیت میکنیم.

## زیر بخش 4

### $n$ -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  $n + 1$ 

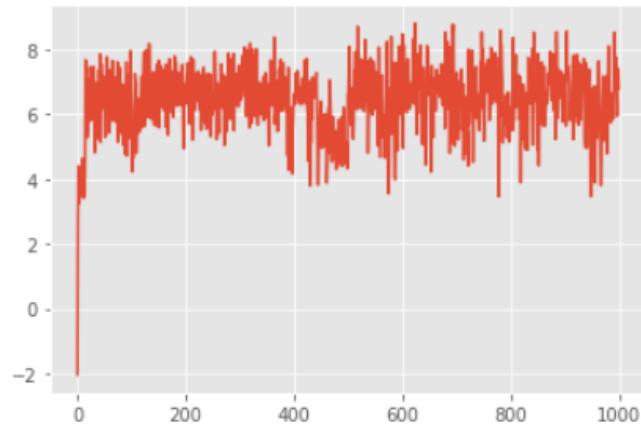
Loop for each episode:
  Initialize and store  $S_0 \neq$  terminal
  Choose an action  $A_0$  arbitrarily as a function of  $S_0$ ; Store  $A_0$ 
   $T \leftarrow \infty$ 
  Loop for  $t = 0, 1, 2, \dots$  :
    | If  $t < T$ :
      |   Take action  $A_t$ ; observe and store the next reward and state 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  $\tau = T - 1$ 

```

## توضیح پیاده سازی

### نتایج

نتایج به ازای آلفا برابر با 0.1 ، اپسیلون 0.1 و 0.9 discount و 40000 اپیزود با سایز پنجره متحرک 40 و استراید 40 گزارش شده اند.



اجرای 20

زمان اجرا(ثانیه) :

```
run time = 1804.0988063812256
```

در الگوریتم 2-step tree backup زمان اجرا از روش مونت کارلو کمتر ولی از دو الگوریتم q-learning و sarsa بیشتر است. میانگین پاداش نسبت به تمام الگوریتم های قبلی بیشتر بوده و در نتیجه حسرت کمتر است.

هر چه مقدار  $n$  در این الگوریتم بیشتر باشد یادگیری با استفاده از اطلاعات دورتر انجام میشود. در نتیجه سرعت یادگیری افزایش می یابد.

### روند اجرای کد پیاده‌سازی

کدهای مربوط به هر زیربخش در نوتبوک ارسالی با نام الگوریتم آن امده است و قابل اجراست.

## سوال 3 – پیاده سازی

### هدف سوال

در این سوال هدف ترکیب روش های مادل بیسد و مادل فری با بکارگیری دو الگوریتم value iteration و q-learning و مشاهده تاثیر آن روی سرعت و مقدار همگرا شده در میانگین مجموع پاداش ها، است.

### توضیح پیاده سازی

برای پیاده سازی این قسمت ابتدا یکبار الگوریتم value\_iteration اجرا میشود و سپس در هر اپیزود الگوریتم q-learning اجرا شده با این تفاوت که در انتخاب اکشن با روش اپسیلون گریدی از  $q$  ترکیبی که رابطه‌ی آن در صورت سوال آمده استفاده میشود و بعد از اپدیت شدن  $q_{mf}(s,a)$  طبق الگوریتم q-learning،  $q$  ترکیبی در انتهای هر استپ آپدیت میشود.

### نتایج

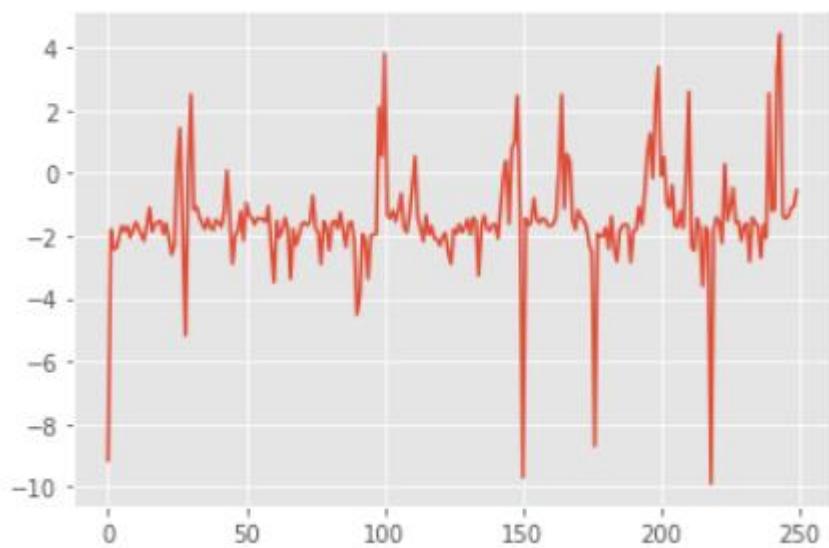
نتایج به ازای آلفا برابر با 0.1 ، اپسیلون 0.1 و 0.9 discount و 10000 اپیزود با سایز پنجره متحرک 40 و استراید 40 گزارش شده اند.

\*به دلیل زمان اجرای طولانی الگوریتم value iteration، تا را به 0.001 دادم و بزرگتر کردم.

$$\mathbf{w} = \mathbf{0}$$

زمان اجرا(ثانیه):

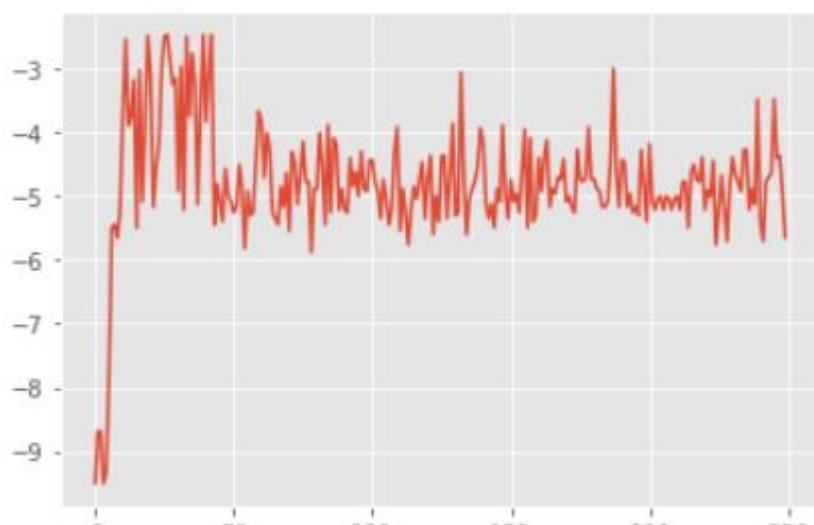
run time = 667.6396405696869



**w = 0.1**

زمان اجرا(ثانیه):

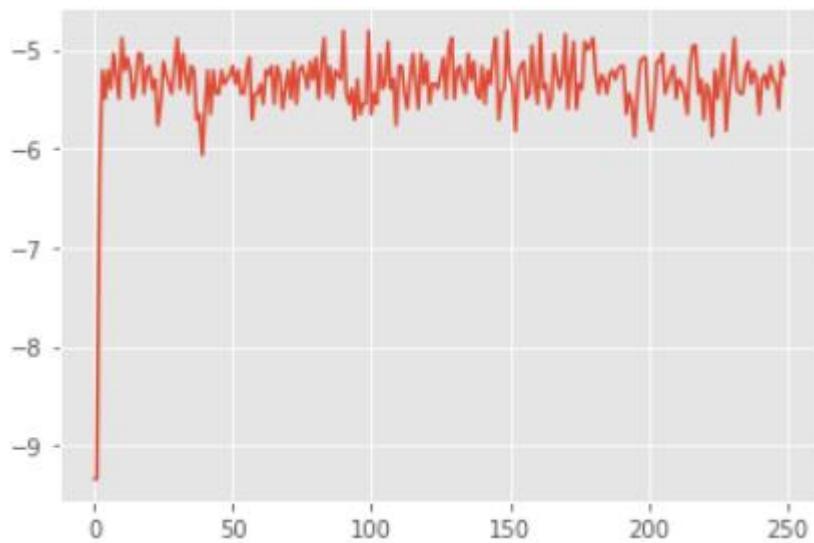
run time = 250.30084347724915



**w = 0.2**

زمان اجرا(ثانیه):

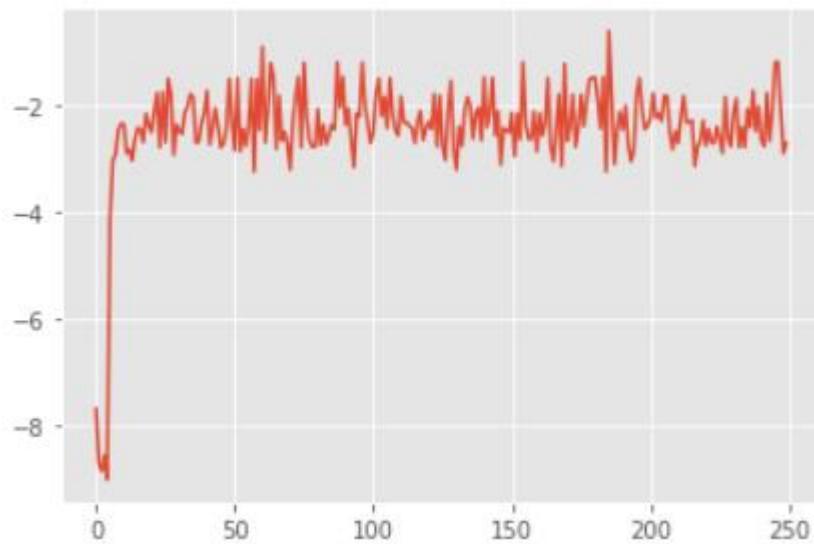
run time = 249.37190127372742



**w = 0.3**

زمان اجرا(ثانیه):

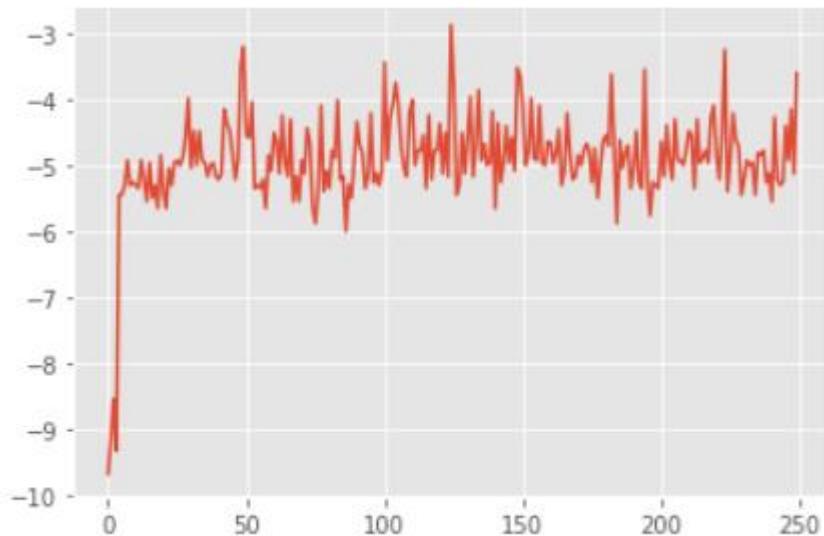
run time = 239.94282484054565



**w = 0.4**

زمان اجرا(ثانیه):

run time = 197.24258375167847

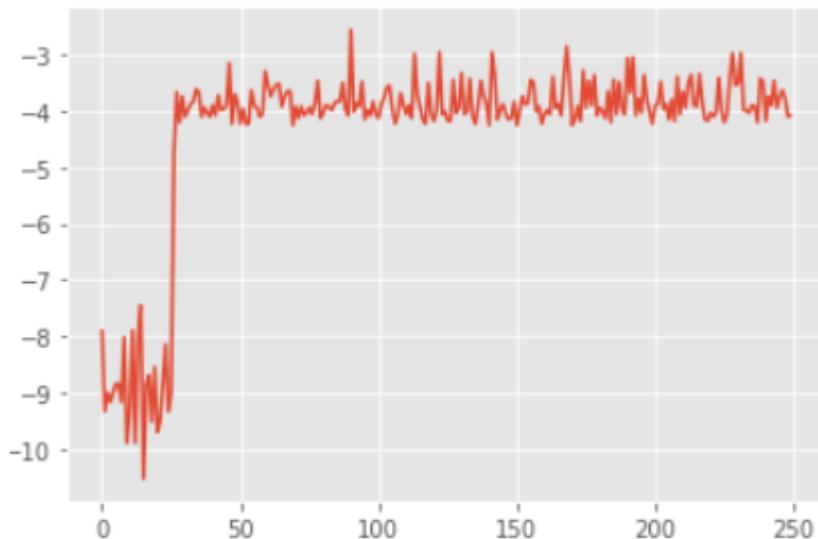


اجرای 20

**w = 0.5**

زمان اجرا(ثانیه):

run time = 185.89833045005798

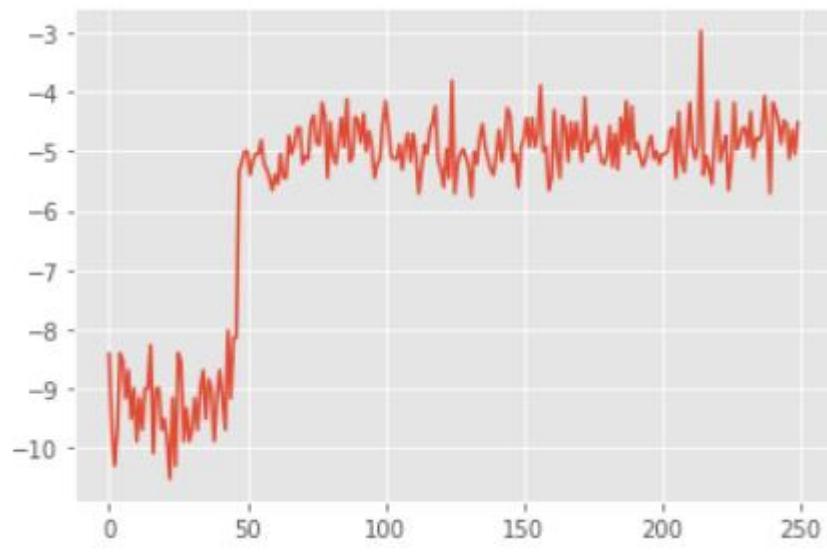


اجرای 20

**w = 0.6**

زمان اجرا(ثانیه):

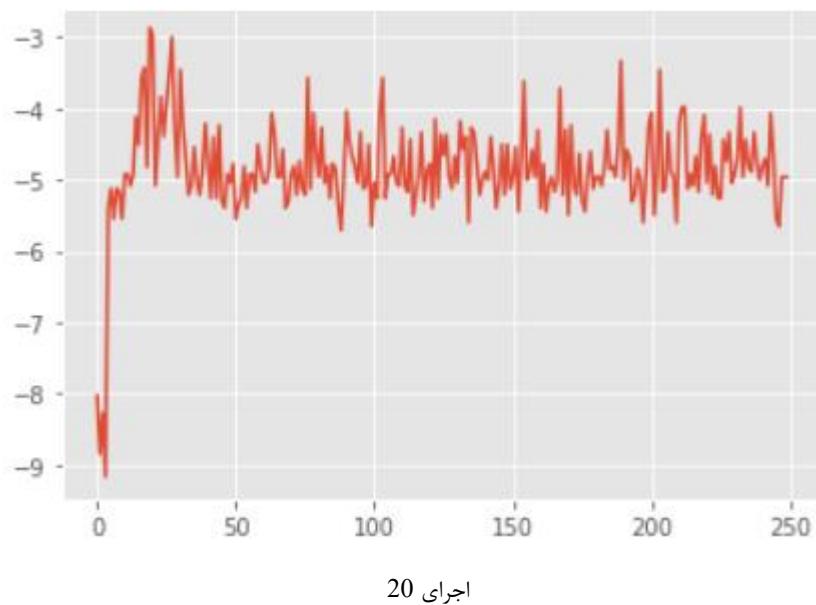
run time = 248.0060338973999



**w = 0.7**

زمان اجرا(ثانیه):

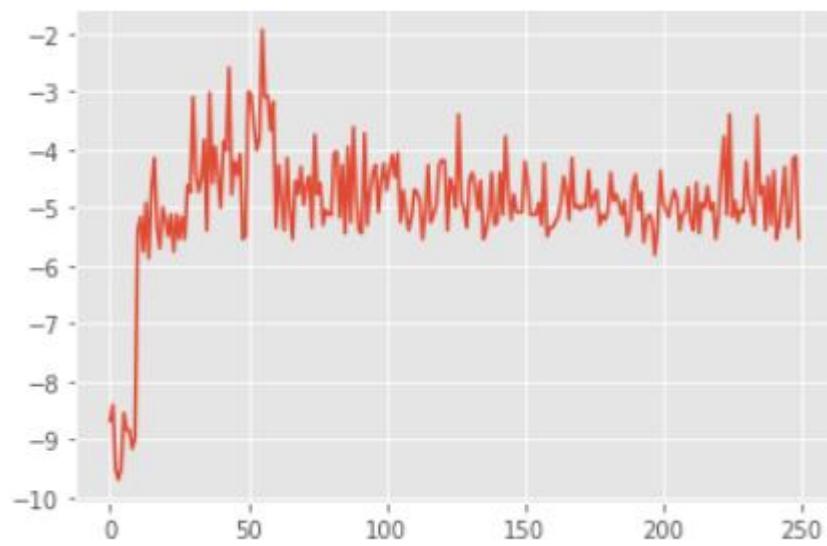
run time = 217.02217960357666



**w = 0.8**

زمان اجرا(ثانیه):

run time = 183.487713098526

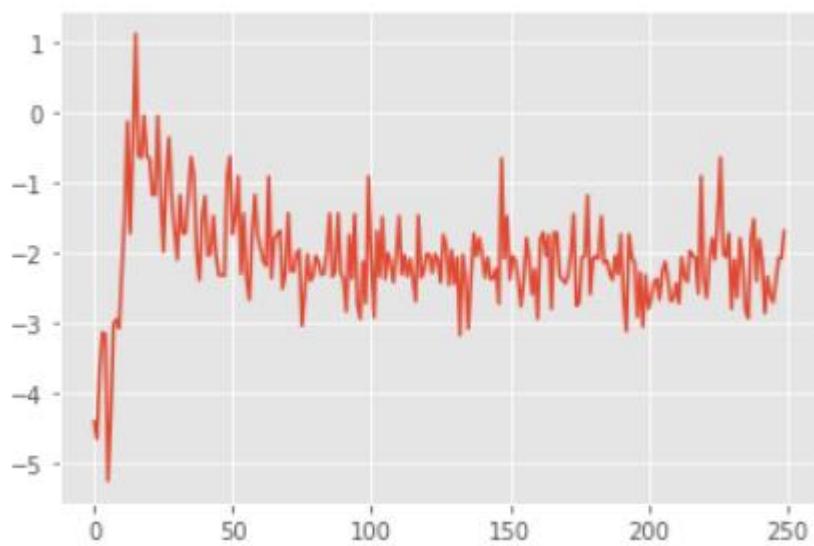


اجرای 20

**w = 0.9**

زمان اجرا(ثانیه):

run time = 189.91094422340393

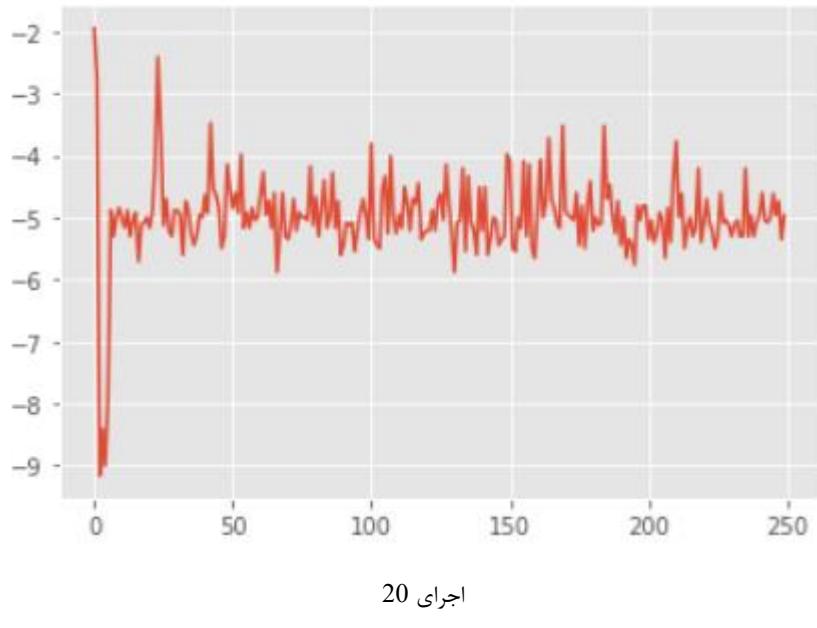


اجرای 20

**w = 1**

زمان اجرا(ثانیه):

run time = 278.632750749588



زمان اجرا به طور کلی نسبت به الگوریتم value iteration به تنها یی بیشتر بوده ولی به ازای  $w$  های بزرگتر، نسبت به الگوریتم q-learning کمتری دارد. زمان اجرا با افزایش  $w$  از صفر تا 0.1 مقدار زیادی کاهش پیدا میکند و سپس تغییر چندانی نمیکند که نشان میدهد ترکیب کردن با روش مادل بیسد زمان را به شدت کاهش داده. مقادیر میانگین پاداش نسبت به روش q-learning کاهش داشته و میزان حسرت بیشتر شده است.

### روند اجرای کد پیاده‌سازی

کد مربوط به این قسمت در بخش MB and MF در نوتبوک ارسال شده، قابل اجراست.