بخش1)

در این بخش value iteration را پیاده سازی میکنیم. Value iteration به این شکل بود که در ابتدا همه ی state ها 0 بودند. در هر دور ، توسط معادله به روز رسانی ، value جدیدی وابسته به value قبلی و اکشن ها و احتمال های ممکن در حرکت از آن state به دست می آمد.

```
def runValueIteration(self):
    states = self.mdp.getStates()
    i = 0
    while i < self.iterations:
        new_values = util.Counter()
        for state in states:
            if not self.mdp.isTerminal(state):
                actions = self.mdp.getPossibleActions(state)
                max_val = float("-inf")
                for action in actions:
                    sum = self.computeQValueFromValues(state, action)
                    max_val = max(sum, max_val)
                new_values[state] = max_val
        i += 1
        for s in states:
            self.values[s] = new_values[s]
```

تابع runValueIteration تابع اصلیمان است که در آن value های جدید محاسبه میشود.

تعداد iteration که یک آرگومان ورودی است از طریق self.iterations قابل دسترسی است. به تعداد self.iterartions از معادله به روز رسانی حالت ها استفاده میکنیم و value ها را آپدیت میکنیم (i شمارنده این iterations هست)

برای value های جدید new_value را تعیین میکنیم و از ساختمان داده ی counter استفاده میکنیم که به طور پیشفرض برای همه ی کلید هایش مقدار 0 را قرار میدهد (کلید ها state ها خواهند بود).

حال روی کل state ها پیمایش میکنیم و مقدار جدید هر state را توسط معادله به روزرسانی محاسبه میکنیم. اگر state پایانی بود نیاز به محاسبه نیست و skip میکنیم.

$$V^*(s) = \max_a Q^*(s,a)$$

طبق معادله بالا پیش میرویم. در هر دور، برای هر state همه ی اکشن های ممکن را با self.mdp.getPossibleActions(state) بدست می آوریم و برای هر کدام q-value را محاسبه میکنیم. ماکسیمم q-value ها همان value ی جدید خواهد بود.

بعد از هر iteration باید value ها را با new_values جایگزین کنیم تا در iteration بعدی از value های قبلی استفاده شود و حالت بازگشتی حفظ شود.
در بدست آوردن q-value ها از تابع computeQValueFromValues استفاده شده.

این تابع طبق فرمول محاسبه ی q-value را انجام میدهد. $$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

```
def computeQValueFromValues(self, state, action):
    q_value = 0
    result = self.mdp.getTransitionStatesAndProbs(state, action)
    for res in result:
        reward = self.mdp.getReward(state, action, res[0])
        q_value += res[1] * (reward + self.discount * self.values[res[0]])
    return q_value
```

با توجه به غیر قطعی بودن، هر اکشن لزوما به یک state منتهی نمیشود بلکه با احتمال های مختلف به state های مختلف میتواند برود. این مجموعه ی احتمالات و state های متناظرشان توسط getTransitionStatesAndProbs قابل دسترسی است. به ازای هر state که میتوان به آن منتهی شد،

p * (reward + discount*value(s')) را محاسبه میکنیم و با هم جمع میکنیم و این همان مقدار q-value است و برگردانده میشود.

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

self.discount همان مقدار گاما میباشد. استفاده از دو تابع بالا در واقع ترکیب دو فرمول بالاییست و فرمول بازگشتی رو به رو را نتیجه میدهد.

سومین تابع پیاده سازی شده تابع computeActionFromValues است که در بدست آوردن policy بهینه کاربرد دارد و بهترین action بر اساس value های فعلی state ها را بدست می آورد.

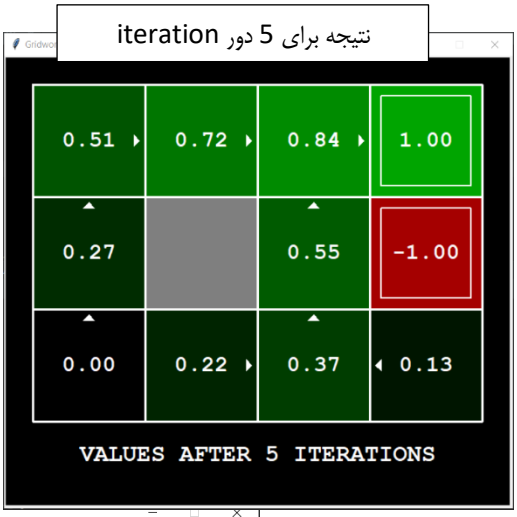
```
def computeActionFromValues(self, state):
    actions = self.mdp.getPossibleActions(state)
    max_value = float('-inf')
    best_action = None
    for action in actions:
        q_value = self.computeQValueFromValues(state, action)
        if q_value > max_value:
            max_value = q_value
            best_action = action
    return best_action
```

در واقع ما در هر state میخواهیم بفهمیم کدام اکشن value ما را بیشتر میکند. برای فهمیدن این موضوع، به ازای همه ی اکشن های ممکن در state ای که قرار داریم ، q-value ها را محاسبه میکنیم. اکشنی که بیشترین q-value را بدهد را انتخاب میکنیم و برمیگردانیم. در صورتی که هی action ای نباشد هم None برمگیردانیم که در جای خود بمانیم.

پیاده سازی توابع بالا به ما در برنامه ریزی اولیه برای بدست آوردن یک policy و value های نزدیک به واقعیت کمک میکند.

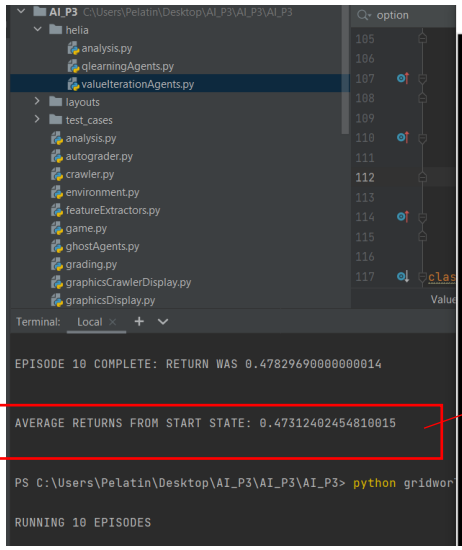
نتایج :

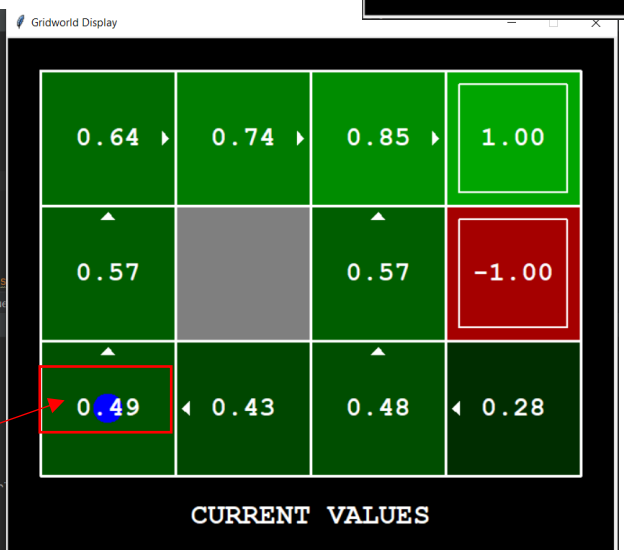
نتیجه برای 5 دور iteration



نتیجه autograder

```
Question q1
===========

*** PASS: test_cases\q1\1-tinygrid.test
*** PASS: test_cases\q1\2-tinygrid-noisy.test
*** PASS: test_cases\q1\3-bridge.test
*** PASS: test_cases\q1\4-discountgrid.test

### Question q1: 4/4 ###
Total: 4/4
```





نتیجه ی grid پس از 100 iteration به شکل زیر است.
Start state در حالت بهینه 0.49 امتیاز به ما میدهد. پس از 10 دور اجرای آزمایش، میانگین 0.473 امتیاز دریافت کرده ایم که بسیار به نتیجه محاسبات policy و value های اولیه نزدیک است.

بخش2)

در حالت پیشفرض noise 0.2 در نظر گرفته شده، یعنی از هر 10 دور انتخاب حرکت ، 2 دورش نتیجه ی دلخواه ما را نمیدهد. با مقادیر پیشفرض به policy و value های زیر میرسیم که به طبع عامل ما از پل عبور نمیکند.

حال ما noise را به شدت کم میکنیم یعنی احتمال افتادن در دره به شدت کاهش میابد پس عامل ریسک عبور از پل را میپذیرد.مقدار noise را انقدر کم میکنیم به نحوی که گویا تقریبا بدون noise کار میکنیم.(دقت شود noise 0 در نظر گرفته نمیشود چون در این صورت عامل ما به صورت قطعی کار میکند و با فرض هایمان در تضاد است.)

حال noise را 0.01 در نظر گرفتیم. همانطور که مشاهده میشود policy به دست آمده نشان میدهد که باید به سمت سوی دیگر پل حرکت کنیم و از پل عبور کنیم. که نتیجه ی روبه رو را داد و عبور ما موفقیت آمیز بود. دقت شود هنوز هم احتمال افتادن در اطراف پل هست اما چون به شدت احتمالش کم هست، عامل ما برای عبور تلاش میکند.


```
Started in state: (6, 1)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 10

EPISODE 1 COMPLETE: RETURN WAS 5.904900000000001


AVERAGE RETURNS FROM START STATE: 5.904900000000001
```

همانطور که در نتیجه میبینیم در یک تلاش ، عامل ما به سمت هدف رفته و 5.9 امتیاز گرفته.

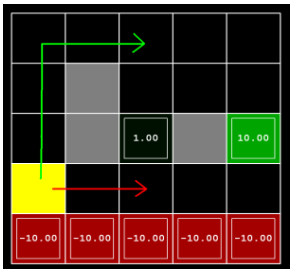
```
*** PASS: test_cases\q2\1-bridge-grid.test

### Question q2: 1/1 ###


------------------
Total: 1/1
```

همانطور که نتایج نشان

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.01
    return answerDiscount, answerNoise
```

میدهد با تغییر answerNoise به 0.01 عامل ما برای بردن تلاش میکند و autograder هم pass شده است.

بخش 3)



- **خروجی نزدیک را ترجیح دهید (+1) و ریسک صخره را بپذیرید(−10) :**

برای ترجیح دادن خروجی نزدیکتر باید تاثیر discount زیاد باشد، طوریکه برای عامل صرفه نداشته باشد حرکات بیشتری برای رسیدن به خروجی 10 را انجام دهد. برای پذیرفتن ریسک صخره باید هم noise را کم بگیریم و هم پاداش زندگی را منفی در نظر بگیریم که عامل بداند باید سریعتر به خروجی برسد چون هر چه بیشتر در grid پیمایش کند امتیاز کمتری میگیرد. ضمنا هزینه زندگی نباید آنقدر زیاد باشد که افتادن در آتش به صرفه تر از طی کردن خانه ها تا رسیدن به خروجی اول باشد.

```
def question3a():
    answerDiscount = 0.31
    answerNoise = 0.01
    answerLivingReward = -0.03
    return answerDiscount, answerNoise, answerLivingReward
```

فرض کنید در گرید بالای صفحه مسیر قرمز را تا رسیدن به 1 بخواهیم پیش برویم.

$3x + y^3 > 5x + 10*y^5$ باید برقرار باشد (x پاداش زندگی و y تخفیف است)

$y^3 > 2x + 10*y^5$ باید برقرار باشد. گفتیم x را منفی در نظر میگیریم یعنی اگر ابتدا مطمین شویم $y^3 > 10*y^5$ برقرار است، کم کردن مقداری منفی از $10y^5$ نامعادله را به هم نمیزند پس $y^3 > 10*y^5$ را حل میکنیم. $y^2 < 0.1$ یعنی $y<0.316$ پس تخفیف را 0.31 در نظر میگیریم و یک مقدار کم و منفی از پاداش برای زندگی نیز قرار میدهیم. برای noise نیز مقدار کمی قرار میدهیم تا عامل حاضر باشد ریسک عبور از کنار صخره را به جان بخرد.

- **خروجی نزدیک را ترجیح دهید (+1) و از صخره اجتناب کنید (−10) :**

برای ترجیح دادن خروجی نزدیکتر باید تاثیر discount زیاد باشد، طوریکه برای عامل صرفه نداشته باشد حرکات بیشتری برای رسیدن به خروجی 10 را انجام دهد. برای اجتناب از ریسک صخره باید noise را زیاد تر در نظر بگیریم. اگر در شکل بالای صفحه بخواهیم تا مسیر سبز برویم و سپس به خروجی نزدیکتر برویم به معادله زیر میتوانیم برسیم.

```
def question3b():
    answerDiscount = 0.31
    answerNoise = 0.3
    answerLivingReward = -0.03
    return answerDiscount, answerNoise, answerLivingReward
```

$7x + y^7 > 9x + 10*y^9$ باید برقرار باشد (x پاداش زندگی و y تخفیف است)

$y^7 > 2x + 10*y^9$ باید برقرار باشد. اگر x را منفی بگیریم کافیست مطمین شویم $y^3 > 10*y^5$ برقرار است، کم کردن مقداری منفی از $10y^9$ نامعادله را به هم نمیزند پس $y^7 > 10*y^9$ را حل میکنیم. $y^2 < 0.1$ یعنی $y<0.316$ پس تخفیف را 0.31 در نظر میگیریم و یک مقدار کم و منفی از پاداش برای زندگی نیز قرار میدهیم. (در واقع همان اعداد بخش قبل هستند و صرفا noise تغییر کرده)

- **خروجی دور را ترجیح دهید (+10) و ریسک صخره را بپذیرید(−10) :**

برای ترجیح دادن خروجی دورتر باید تاثیر discount کم باشد، طوریکه عامل حاضر باشد حرکات بیشتری برای رسیدن به خروجی 10 را انجام دهد. برای پذیرفتن ریسک صخره باید هم noise را کم بگیریم و هم پاداش زندگی را منفی در نظر بگیریم که عامل بداند باید سریعتر به خروجی برسد چون هر چه بیشتر در grid پیمایش کند امتیاز کمتری میگیرد. ضمنا هزینه زندگی نباید آنقدر زیاد باشد که افتادن در آتش به صرفه تر از طی کردن خانه ها تا رسیدن به خروجی دورتر باشد.

```
def question3c():
    answerDiscount = 1
    answerNoise = 0.02
    answerLivingReward = -1
    return answerDiscount, answerNoise, answerLivingReward
```

فرض کنید در گرید بالای صفحه مسیر قرمز را تا رسیدن به 10 بخواهیم پیش برویم.

$5x + 10*y^5 > 3x + y^3$ باید برقرار باشد (x پاداش زندگی و y تخفیف است)

$2x + 10*y^5 > y^3$ باید برقرار باشد. گفتیم discount باید تاثیر کمی داشته باشد پس آنرا 1 میگیریم. در نتیجه باید $2x + 10 > 1$ باشد و x هم منفی است. X>-4.5 پس مثلا x را -1 در نظر میگیریم.

- **خروجی دور را ترجیح دهید (+10) و از صخره اجتناب کنید (-10) :**

برای ترجیح دادن خروجی دورتر باید تاثیر discount کم باشد، طوریکه عامل حاضر باشد حرکات بیشتری برای رسیدن به خروجی 10 را انجام دهد. برای اجتناب از ریسک صخره هم باید noise را زیاد بگیریم که عامل از پایین گرید دور شود.

```
def question3d():
    answerDiscount = 1
    answerNoise = 0.5
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
```

discount را 1 قرار میدهیم و هزینه زندگی را نیز 0 میکنیم. یعنی در هر نقطه از این صفحه برای ما به صرفه است به سمت 10 برویم چون رسیدن به 10 اگر با افتادن در آتش همراه نباشد قطعا سود بیشتر از رسیدن به 1 دارد.

- **از خروجی ها و صخره ها اجتناب کنید:**

```
def question3e():
    answerDiscount = 0.9
    answerNoise = 0.9
    answerLivingReward = 5
    return answerDiscount, answerNoise, answerLivingReward
```

کافیست پاداش زندگی انقدر زیاد باشد که ماندن در گرید بسیار بهتر از رسیدن به اهداف باشد. ضمنا noise را هم زیاد میکنیم که عامل از صخره ها دور شود، هنگامیکه به بالای گرید رسید چون پاداش زندگی زیاد است همان بالا ها میماند و نیازی ندارد به سمت خروجی ها برود (تغییر discounting تاثیرش فقط کمک به کم اهمیت تر شدن خروجی هاست)

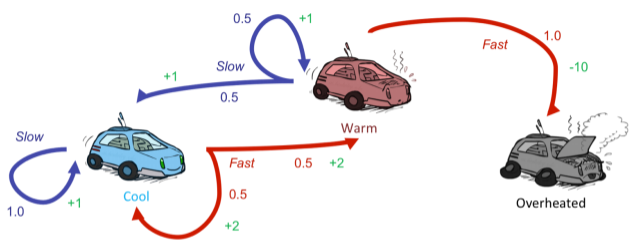
نتیجه autograder:

```
Provisional grades
==================
Question q3: 5/5
------------------
Total: 5/5
```

سوال : آیا لزوما به همگرایی میرسیم در value iteration ؟

در حالت کلی اگر گاما را کمتر از 1 بگیریم به همگرایی خواهیم رسید. فرض کنید تا vk پیش رفته باشیم در محاسبه ی vk+1 تاثیر reward ها در گاما به توان k ضرب میشود که اگر k به اندازه کافی بزرگ باشد گاما به توان k به صفر میل میکند پس reward ها تاثیر تقریبا 0 روی مقدار vk+1 میگذارند و به همگرایی میرسیم.

مثال زیر از اسلاید ها را در نظر بگیرید.



Assume no discount!

ماشین آبی را حالت m و ماشین قرمز را حالت n در نظر بگیرید. در هر دور محاسبه ی value برای حالت m داریم:

$$v_{k+1}(m) = \max(v_k(m) + 1 , 0.5(v_{k(m)} + 2) + 0.5(v_k(n) + 2)$$

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

یعنی به طور قطع، $v_{k+1}(m)$ یا $v_k(m) + 1$ است و یا بیشتر از آن پس یعنی هیچ وقت مقدارش به همگرایی نمیرسد (اگر بازه تغییرات را بیشتر از 1 در نظر نگیریم البته)

بخش4)

در این بخش value iteration را دوباره پیاده سازی میکنیم اما اینبار به جای اینکه در هر دور کل استیت ها را باهم آپدیت کنیم، تک تک value های آنها را بهبود میدهیم.

```python
def runValueIteration(self):
    states = self.mdp.getStates()
    i = 0
    states_num = len(states)
    while i < self.iterations:
        state = states[i % states_num]
        if not self.mdp.isTerminal(state):
            actions = self.mdp.getPossibleActions(state)
            max_val = float("-inf")
            for action in actions:
                sum_p = self.computeQValueFromValues(state, action)
                max_val = max(sum_p, max_val)
            self.values[state] = max_val
        i += 1
```

ابتدا با استفاده از تابع get.States مجموعه ی کل حالات در mdp ورودی را میگیریم. متغییر i برای شمردن iteration هایمان است و آنرا با 0 initialize میکنیم. متغییر states_num تعداد حالات را نگه میدارد.

حلقه ای به اندازه iteration داریم.
هر دور استیتی که باید update شود را با باقی مانده ی i در تقسیم بر تعداد حالات به دست می آوریم. چک میکنیم که ترمینال نباشد و اگر ترمینال نبود فرمول

$$V^*(s) = \max_a Q^*(s,a)$$

را برایش محاسبه میکنیم. با توجه به اینکه AsynchronousValueIterationAgent از کلاس ValueIterationAgent ارثبری میکند، متد computeQValueFromValues را نیز دارد و میتوانیم از آن استفاده کنیم. بعد از محاسبه ی value جدید برای state مان، value اش را آپدیت میکنیم تا در iteration های بعدی، بقیه از value اش استفاده کنند.

```
Starting on 1-18 at 19:57:17

Question q4
==========

*** PASS: test_cases\q4\1-tinygrid.test
*** PASS: test_cases\q4\2-tinygrid-noisy.test
*** PASS: test_cases\q4\3-bridge.test
*** PASS: test_cases\q4\4-discountgrid.test

### Question q4: 1/1 ###


Finished at 19:57:17
```

python autograder.py -q q4 را برای تست درستی الگوریتم اجرا میکنیم.
همانطور که در نتایج مشاهده میشود به طور کامل تست ها را با موفقیت رد میکند.
تفاضل زمان شروع و پایان کمتر از یک ثانیه است(در حد میلی ثانیه است و نشان داده نمیشود) پس نیازی که در دستور کار مبنی بر کمتر از 1 ثانیه بودن اجرا گفته شده بود را نیز برآورده میکند.

```
AVERAGE RETURNS FROM START STATE: 0.4861321118100001
```

متوسط امتیاز کسب شده با شروع از (0,0) 0.486 به دست آمده که بسیار نزدیک به value ایست که در گرید توسط Asynchronous Value Iteration Agent به دست آوردیم (0.004 اختلاف دارند.).

سوال)مقایسه روش های بخش 1 و 4

در حالت asynchronous ما نیازی نداریم کل value های state ها را در جایی ذخیره کنیم (در synchronous آرایه ای میگرفتیم پس اشغال فضا از مرتبه o(s) بود)و بعد باهم آپدیتشان کنیم چون در هر مرحله یک state مستقیما آپدیت میشود پس از نظر space complexity فضای کمتری اشغال میکند در حد o(1).

حالت asynchronous سریع تر converge میشود چون value های state ها زودتر آپدیت میشوند. برای مثال اگر در محاسبه v(s2) نیاز به v(s1) میداشتیم، در حالت synchronous از v(s1) آپدیت نشده (برای مرحله k ام ایتریشن) استفاده میشد در حالی که در حالت asynchronous از v(s1) ای استفاده میشود که در مرحله قبل آپدیت شده و بهبود یافته.

اما در asynchronous باید عدد iteration را بزرگتر بدهیم تا همه ی state ها به تعداد بار قابل توجه و خوبی update شوند.

بخش 5)

طبق مراحل دستور کار عمل میکنیم.

از ساختمان داده ی PriorityQueue که در فایل utils تعریف شده استفاده میکنیم تا صف اولویت را بسازیم.

```python
def runValueIteration(self):

    states = self.mdp.getStates()
    predecessors = {}
    priority_queue = util.PriorityQueue()

    for state in states: predecessors[state] = set()
    for state in states:
        if not self.mdp.isTerminal(state):
            actions = self.mdp.getPossibleActions(state)
            for action in actions:
                transitions = self.mdp.getTransitionStatesAndProbs(state, action)
                max_value = float('-inf')
                for nextState, prob in transitions: predecessors[nextState].add(state)
                q_value = self.computeQValueFromValues(state, action)
                max_value = max(q_value, max_value)
            diff = abs(max_value - self.values[state])
            priority_queue.push(state, - diff)
    i = 0
    while i < self.iterations:
        if priority_queue.isEmpty():
            break
        state = priority_queue.pop()
        if not self.mdp.isTerminal(state):
            actions = self.mdp.getPossibleActions(state)
            max_val = float("-inf")
            for action in actions:
                q_value = self.computeQValueFromValues(state, action)
                max_val = max(q_value, max_val)
            self.values[state] = max_val
        for p in predecessors[state]:
            if not self.mdp.isTerminal(p):
                max_value = float('-inf')
                actions = self.mdp.getPossibleActions(p)
                for action in actions:
                    q_value = self.computeQValueFromValues(p, action)
                    max_value = max(q_value, max_value)
                diff = abs(max_value - self.values[p])
                if diff > self.theta:
                    priority_queue.update(p, -diff)
        i += 1

```

گفته شده برای همه ی حالات predecessors ها را نگه داریم. دیکشنریی میسازیم که کلید های آن state ها هستند و value ها مجموعه ی predecessor ها هستند.

در حلقه ای روی state ها، برای هر state ، predecessors ها را به دست میاوریم و در دیکشنری اضافه میکنیم. اگر state نهایی باشد کاری نمیکنیم. اگر نهایی نباشد همه ی action های مجازش را به دست میاوریم. با استفاده از تابع getTransitionStatesAndProbs همه ی نتایج ناشی از یک action را بدست میاوریم و state های حاصل از آن اکشن را به مجموعه یpredecessors های آن state اضافه میکنیم.
ضمنا تفاضل state را با منفی قدرمطلق تفاضل ارزش state و بیشترین مقدار q_value به صف اولویتمان اضافه میکنیم. (در این صف، هرچه اندازه ی اولویت کمتر باشد یعنی باید زودترpop شود.)

حال در حلقه ی اصلی هستیم که به تعدادiteration آپدیت کردن value ها را انجام میدهد. نکته اینجاست اگر صف اولویت خالی شده بود نیازی نیست ادامه دهیم و حلقه را تمام میکنیم.

هر دور، state ای که اولویت بیشتری در صف پیدا کرده(یعنی با آپدیت کردن value اش مقدارش بیشتر عوض میشود (چون منفی تفاضل مقدار فعلی و ماکس q اش اولویت است یعنی هرچه تفاضل بیشتر باشد اولویت بالاتری دارد)) را pop میکنیم.

اگر state ترمینال نبود به آپدیت کردن value آن میپردازیم. اکشن های ممکن آن حالت را به دست میاوریم و به ازای هر اکشن q-value را محاسبه میکنیم. ماکسیمم این q-value ها میشود value ی جدید استیت فعلی.

نکته اینجاست تغییر ارزش این حالت، تفاضل ارزشش با q-value ماکسیمم predecessors هایش را تغییر میدهد یعنی اولویت آنها عوض میشود پس قطعا باید آنها به روز رسانی شوند. برای این بروزرسانی، از دیکشنری predecessors استفاده میکنیم. Predecessors های استیت فعلی را بدست میآوریم. تفاضل مقدار جدید استیت از مقدار ماکسیمم q-value ی هر predecessors را بدست میآوریم و اگر این مقدار از theta کمتر بود در صف اولویت ، آن را با اولویت جدید آپدیت میکنیم.

نکته: کلاس PrioritizedSweepingValueIterationAgent از AsynchronousValueIterationAgent ارثبری میکند و مانند آن state ها را تک تک آپدیت میکند و مثل valueIteraion عادی ، وکتور value ها را یکباره تغییر نمیدهد.

نتیجه ی اجرای تکرار ارزش های اولویت بندی شده با 1000 iteration به شکل بالاست که اگر دقت کنیم مشابه بخش های قبل است.

```
Starting on 1-20 at 2:25:00

Question q5
===========

*** PASS: test_cases\q5\1-tinygrid.test
*** PASS: test_cases\q5\2-tinygrid-noisy.test
*** PASS: test_cases\q5\3-bridge.test
*** PASS: test_cases\q5\4-discountgrid.test

### Question q5: 3/3 ###


Finished at 2:25:00
```

نتیجه ی کد را با autograder تست میکنیم.همانطور که مشخص است تفاضل زمان شروع و پایان در حد میلی ثانیه بوده به همین دلیل زمان شروع و پایان یکی است. در دستور کار خواسته شده بود مدت زمان اجرا کمتر از 1 ثانیه باشد که میبینیم برآورده شده.

بخش 6)

در این بخش میخواهیم q-learning را پیاده سازی کنیم. در این حالت دیگر مانند بخش های قبل، یک policy درست نمیکنیم و بعد به اجرای آن بپردازیم، بلکه با تعدادی آزمایش، محیط را یادمیگیریم و میفهیمم هر action ما چقدر به ما سودمندی میدهد و ... و بر اساس نتایج آزمایشهایمان policy مان را پی در پی بروز رسانی میکنیم تا به حالتی بهینه برسد.

```
def __init__(self, **args):
    ReinforcementAgent.__init__(self, **args)
    self.q_values = {}
```

همانطور که از اسم روش مشخص است ما احتیاج به استفاده از q-value های هر state داریم. در تابع init یک دیکشنری q_values تعریف میکنیم که بعدا در آن همه ی q_value های state ها با کلید هایی به شکل (state,action) نگهداری خواهد شد.

تابع getQValue :

```
def getQValue(self, state, action):
    if (state, action) in self.q_values:
        return self.q_values[(state, action)]
    return 0.0
```

در این تابع مقدار q_value ی state مان به action مورد نظر را برمیگردانیم. برای اینکار چک میکنیم اگر در دیکشنری q_value هایمان این state و action وجود داشت، مقدار q_value متناظر با آنها را برمیگردانیم و درغیر اینصورت 0 برمیگردانیم. در واقع این صفر برگرداندن یعنی در ابتدای کار، همه ی q-value ها را صفر میگیریم.

تابع computeValueFromQValues :

```
def computeValueFromQValues(self, state):
    legalActions = self.getLegalActions(state)
    if len(legalActions) == 0:
        return 0.0
    value = float('-inf')
    for action in legalActions:
        new_value = self.getQValue(state, action)
        value = max(value, new_value)
    return value
```

Value ی هر استیت در واقع همان ماکسیمم q-value هایش است. بنابراین در این تابع ، ماکسیمم q-value های state ورودی را یافته و برمیگردانیم. در صورتی این state ترمینال باشد، هیچ action ای نخواهد داشت پس طول legalActions آن 0 میشود، در این حالت 0.0 را برمیگردانیم.

تابع computeActionFromQValues :

در این تابع میخواهیم action ای که به ما بیشترین value را میدهد برگردانیم. در واقع درq-learning وقتی بنویسیم $\pi(state)$ که همان policy برای state است، داریم این تابع را صدا میزنیم.
مشابه متدcomputeValueFromQValues باید ماکسیمم q-value ها را پیدا کنیم لیستی از اکشن هایی که ما را به این ماکسیمم میرسانند نگه میداریم و در آخر به شکل رندوم از بین آنها یک اکشن را انتخاب میکنیم. ضمنا اگر state ترمینال بود ، هیچ اکشنی ندارد که انجام بدهد پس None برمیگردانیم.

```
def computeActionFromQValues(self, state):
    best_action = []
    legal_actions = self.getLegalActions(state)
    max_value = self.computeValueFromQValues(state)
    best_action = [action for action in legal_actions if self.getQValue(state, action) == max_value]
    return random.choice(best_action)
```

تابع update :

درواقع مهمترین تابع این بخش است و نتایج نمونه ها را در q-value ها تاثیر میدهد.

```
def update(self, state, action, nextState, reward):
    new_q = (1 - self.alpha) * self.getQValue(state, action) + \
            self.alpha * (reward + self.discount * self.computeValueFromQValues(nextState))
    self.q_values[(state, action)] = new_q
```

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[r + \gamma \max_{a'} Q(s',a')\right]$$

طبق فرمول بالا مقدار Q را برای یک sample آپدیت میکنیم. Sample ها آزمایش های ما هستند که در آنها ابتدا در یک state بوده ایم، action ای انجام داده ایم و به حالت بعدی که nextState است رفته ایم و پاداش به اندازه reward را گرفتیم. حال هر Smaple را با 4 property مشخص میکنیم که state,action,nextState,reward هستند و به اختصار مینویسیم s(s,a,s',r). این متد بعد از هر sample ای که اجرا شد صدا زده میشود و طبق s,a,s' و r ، مقادیر Q را تغییر میدهد.

الفا ضریبی است که به اندازه آن به نتیجه ی سمپل جدید اهمیت میدهیم و در مقدار q تاثیرش میدهیم.
(1-alpha) پس میشود میزان تاثیر همان q ی فعلی state.

گاما همان ضریب تخفیف است که با self.discount قابل دسترسی است.

نتیجه ی تست با autograder

```
Question q6
===========

*** PASS: test_cases\q6\1-tinygrid.test
*** PASS: test_cases\q6\2-tinygrid-noisy.test
*** PASS: test_cases\q6\3-bridge.test
*** PASS: test_cases\q6\4-discountgrid.test
Provisional grades
==================
Question q6: 4/4
------------------
Total: 4/4
```

سوال) اگر مقدار اولیه ی Q بسیار کم یا بسیار زیاد باشد چه میشود؟

ما هر بار بین q-value ها ماکسیمم میگیریم و آن اکشنی که q-value اش ماکسیمم شده را به عنوان سیاست انتخاب میکنیم. فرض کنید q اولیه بسیار کم باشد. حالتی را در نظر بگیرید که q-value 3 از یک state مقدار هایشان منفی به دست آمده. منطقا بهتر است در حالات بعدی، action ای که هنوز q-value اش به دست نیامده را امتحان کنیم تا شاید امتیازمان را بیشتر کند اما چون q اولیه خیلی کم است، از q-value 3 ی منفیی که explore شده اند هم کمتر است و وقتی ما ماکسیمم q-value ها را میگیریم هیچ وقت این اکشن را انتخاب نمیکنیم و exploration را کم میکند.

اگر q های اولیه بسیار زیاد باشند، حین ماکسیمم گیری همیشه انتخاب میشوند و یعنی سیاست های ابتدایی ما بیشترین میزان explore را دارند تا حداقل یکبار همه ی state ها را با همه یaction ها بگردند و q-value ها را اپدیت کنند.

بخش 7)

تابع getAction :

```
def getAction(self, state):
    legalActions = self.getLegalActions(state)
    if util.flipCoin(self.epsilon):
        return random.choice(legalActions)
    return self.getPolicy(state)
```

این تابع action بعدیمان را برمیگرداند. مقدار epsilon احتمال اینست که ما به طور تصادفی عمل کنیم و exploration با epsilon تضمین میشود . با استفاده از تابع flipCoin در util که به احتمالی برابر با ورودی اش true برمیگرداند، action ای تصادفی بین action های ممکن حالت فعلی برمیگردانیم (اگر flipCoin(epsilon) به ما true برگرداند، اکشن تصادفی را با تابع random.choice به دست میاوریم).

با احتمال 1-epsilon (همان else برای true بودن flipCoin است)هم از تابع getpolicy استفاده کرده و براساس policy ای که از نتایج نمونه های قبل حاصل شده، بهترین action بعدی را انجام میدهیم.

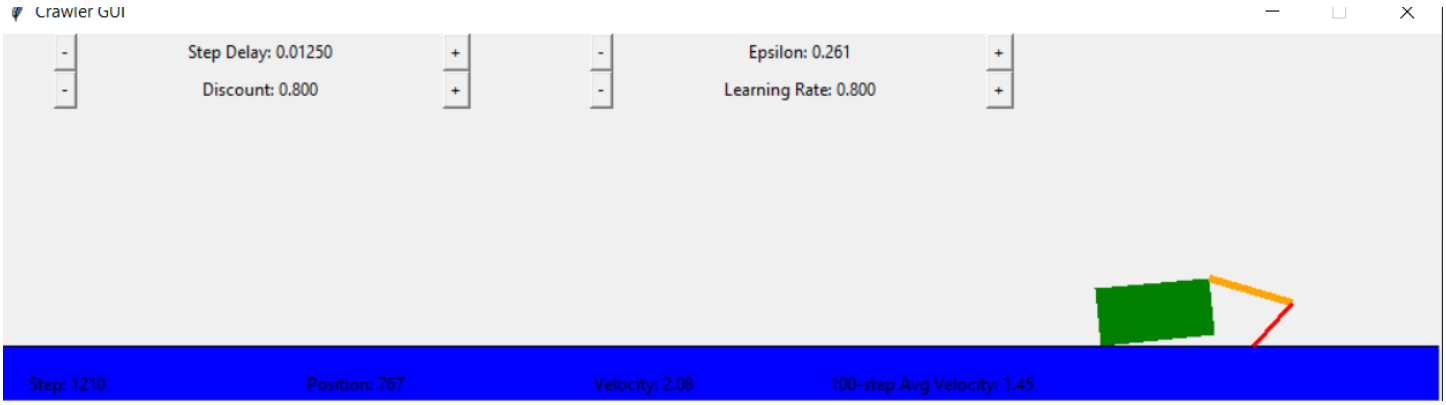
تست کد:

نتیجه ی autograder

```
### Question q7: 2/2 ###


Finished at 14:12:07

Provisional grades
==================
Question q7: 2/2
------------------
Total: 2/2
```

نتیجه ی q-learning برای 100 اپیزود به شکل بالاست. این مقادیر با حالت بهینه و سیاست بهینه متفاوتند و دلیلش تعداد کم اپیزود هاست اگر آزمایش را بیشتر ادامه دهیم به مقادیر بهینه همگرا خواهد شد.

کلاس عامل q-learning ما تکمیل شده و میتواند برای ربات crawler استفاده شود. این ربات قرار است به جلو حرکت کند اما نمیداند چگونه. با همین روش q-learning ابتدا محیط را ازمایش میکند (دو موتور دارد و آنها را مقدار دهی میکند و میسنجد با چه سرعتی و چه قدر جلو رفته که همان reward است.). بعد از تعداد خوبی از نمونه گیری ربات به راحتی جلو میرود.



بخش 8)

در این گرید از خانه ای که خودش ترمینال است داریم شروع به حرکت میکنیم. اگر احتمال مرگ زیاد باشد برای ما بهینه است به خانه شروع برگردیم و سریع تر بازی تمام شود به همین دلیل میبینیم در q-learning با 50 اپیزود، چون کل گرید را نگشته است فکر میکند جلوتر فقط مرگ در انتظارش است پس بهتر است زودتر بمیرد و به خانه ی شروع برمیگردد.

فرض کنید epsilon = 0 باشد ، در این حالت عامل فقط کار های بهینه انجام میدهد. وقتی در نقطه ی شروع است تنها کاری که میتواند بکند حرکت به جلوست پس به جلو حرکت میکند، اما بعد از ورود به خانه ی بعدی، حرکت بهینه همواره برگشتن به خانه ی شروع است چون q بقیه 0 است اما q خانه شروع عددی مثبت است پس هیچگاه پیشروی نخواهیم داشت.

پس باید epsilon زیاد باشد که عامل ترغیب شود روی پل پیش برود اما نکته اینجاست با 50 اپیزود احتمال رسیدن به ته پل بسیار کم است. در یک اپیزود با 5 حرکت، اگر قرار باشد به ته خط برسیم، احتمال آن $\frac{1}{4}^5$ است که یعنی از هر 1024 تا اپیزود، احتمالا یکی اش به ته خط میرسد پس تقریبا اگر میخواهیم بعد از k اپیزود حتما سیاست را پیدا کرده باشیم با k/1024 = 99/100 برقرار باشد یعنی 1013 تا اپیزود حداقل طی شود . احتمال اینکه با 50 اپیزود به ته خط و سیاست بهینه برسیم به شدت کم است و امکان ندارد در 99 درصد مواقع با 50 اپیزود سیاست بهینه را پیدا کنیم.

طبق توضیات بالا برای epsilon و learning_rate نمیتوانیم اعداد مناسب بگذاریم پس NOT POSSIBLE برمیگردانیم.

```
def question8():
    return 'NOT POSSIBLE'
```

نتیجه auto grader :

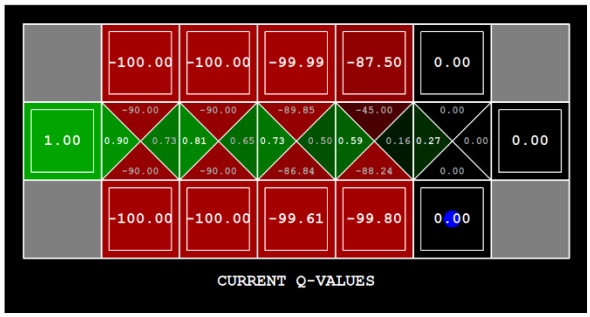
```
Question q8: 1/1
----------------
Total: 1/1
```

البته در اعداد بالا فرض کردیم همه ی اپیزود ها 6 حرکتی اند و در واقعیت این شکلی نیست و هرجا به ترمینال برسیم کار تمام است یعنی ایزود ها از 2 تا هرعددی میتوانند باشند بنابراین عدد 1013 صرفا برای اینست بگیم با 50 تا نمیتوان به سیاست بهینه رسید.

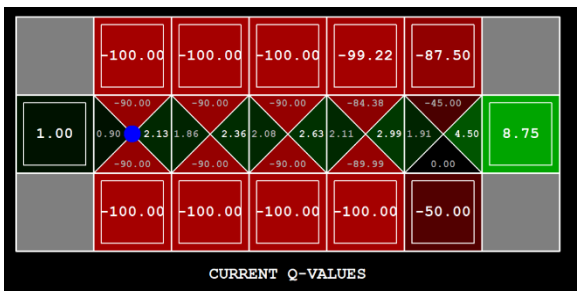
البته بهترین روش اینست ابتدا epsilon بسیار زیاد باشد و بعد کم کم مقدارش را کاهش دهیم.

- فقط برای امتحان 4060 اپیزود را ازمایش میکنیم.

تا 500 اپیزود:



تا 1500 اپیزود:



بخش 9)

همانطور که در دستور کار گفته شده پکمن q-learning را در 2 بخش انجام میدهد، بخش اول یادگیری است و بخش دوم epsilon 0 میشود تا دیگر به شکل رندوم کاری نکند و صرفا از سیاستی که در بخش 1 به دست اورده برای برنده شدن استفاده کند.

نتیجه ی python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid

دستور بالا میگوید 2010 بار در گرید بازی کنیم. از این 2010 بار 2000 بارش نشان داده نمیشوند و همان بازی های آزمایشی برای یادگیری است و بعد از آن 10 بازی اخر به عنوان نتیجه ی استفاده از سیاست به دست آمده از 2000 بازی قبل نشان داده میشود.

```
Reinforcement Learning Status:
        Completed 2000 out of 2000 training episodes
        Average Rewards over all training: -40.38
        Average Rewards for last 100 episodes: 308.46
        Episode took 0.51 seconds
Training Done (turning off epsilon and alpha)
```

نتیجه ی autograder :

```
###

Finished at 16:09:51

Provisional grades
==================
Question q9: 1/1
------------------
Total: 1/1
```

حال وارد فاز دوم میشود و صرفا از سیاستی که بدست اورده استفاده میکند .

```
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 495
Average Score: 499.8
Scores:        503.0, 503.0, 499.0, 502.0, 502.0, 502.0, 495.0, 502.0, 495.0, 495.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

پکمن 10 دور بازی میکند و هر ده دور برنده میشود.

برای امتحان ، با 4000 دور بازی آزمایشی برای mediumGrid ، q-learning را اجرا میکنیم اما نتایج خوب نیست و در هر ده بازی اخر میبازیم

```
Scores:        -493.0, -491.0, -497.0, -517.0, -550.0, -490.0, -501.0, -502.0, -489.0, -492.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

دوباره امتحان میکنیم و 10000 بار برای یادگیری قرار میدهیم و مشاهده میشود سیاست نهایی پکمن بهبود پیدا کرده و 40 درصد بازی ها را میتواند ببرد.

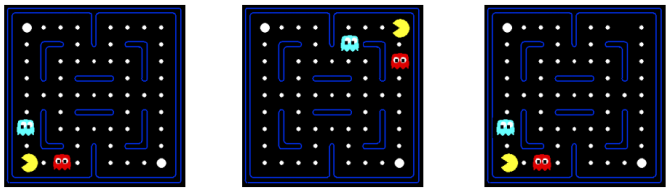
```
Scores:        -522.0, 529.0, 529.0, -537.0, -490.0, -510.0, 522.0, -536.0, -482.0, 529.0
Win Rate:      4/10 (0.40)
Record:        Loss, Win, Win, Loss, Loss, Loss, Win, Loss, Loss, Win
```

حال تعداد یادگیری را دوبرابر میکنیم و با 20000 یادگیری نتایج بسیار بهتر است و در 90 درصد مواقع میبرد. این یادگیری بسیار زمان بر است و هر100 اپیزود حدود 1 ثانیه به طور میانگین زمان میبرد و تقریبا 4 دقیقه یادگیری طول میکشد.

```
Scores:        527.0, 529.0, 527.0, 529.0, 529.0, 529.0, 529.0, 529.0, 527.0, 529.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

بخش 10)

حال میخواهیم q-learning تقریبی را پیاده سازی کنیم.



در حالت q-learning عادی وقتی مثلا پکمن در شرایط روبه رو قرار بگیرد، همه ی آنها برایش جدیدند و نمیتواند از تجربه ی قرار گرفتن در شرایط مشابه استفاده کند (با اینکه ما از بیرون میبینیم با سیاست های مشابه میتوان با این حالات رو به رو شد).

Approximate q-learning هر state را به جای موقعیت و ... با وکتوری از مقادیر ویژگی ها توصیف میکند (در کل state مان را انگار عوض کرده ایم جوری که state های مشابه زیادی به دست بیاوریم). این ویژگی ها اصولا مقادیری بین 0 و 1 هستند. ضمنا برای هر ویژگی وزن تعریف میکنیم به این معنا که برخی ویژگی ها مهمترند و باید تاثیر بیشتری روی سیاست ما داشته باشند پس در محاسباتمان با w بزرگتری قرار میگیرند.

حال هر Q-value را با فرمول $Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$ میتوانیم محاسبه کنیم.

$$\text{transition} = (s, a, r, s')$$
$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a)$$
$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\text{difference}]$$
$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s,a)$$

برای هر transition از s به s' که در آن پاداش r را گرفته ایم و اکشن a را انجام داده ایم طبق فرمولهای بالا ، q-value ها و w ها را بروزرسانی میکنیم. درواقع بر اساس learning rate ای که داریم، تفاوت نتایج این ترنزیشن جدید با state الانمان را در q و w ها تاثیر میدهیم.

تابع getQValue :

```
def getQValue(self, state, action):
    q_value = 0.0
    f_vector = self.featExtractor.getFeatures(state, action)
    for feature in f_vector:
        q_value += self.weights[feature] * f_vector[feature]
    return q_value
```

این تابع مقدار q-value برای یک state و action را برمیگرداند. گفتیم state ها را با وکتوری از ویژگی ها توصیف میکنیم پس هر q-valueرا طبق فرمول $Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$ محاسبه میکنیم. در تابع ، مقدار وکتور ویژگی ها با تابع getFeatures برای state و action مورد نظر به دست می آید. حال هر ویژگی را در وزنش ضرب میکنیم و این مقادیر را با هم جمع میکنیم تا q-value به دست آید. دقت شود وزن ها در دیکشنری weights نگهداری میشوند که کلید های آن ویژگی ها هستند.

تابع update :

```
def update(self, state, action, nextState, reward):
    difference = (reward + self.discount * self.getValue(nextState)) -self.getQValue(state, action)
    f_vector = self.featExtractor.getFeatures(state, action)
    for feature in f_vector:
        self.weights[feature] += self.alpha * difference * f_vector[feature]
```

این تابع بعد از یک ترنزیشن که به شکل (state,action,nest_state,reward) نشانش میدهیم صدا زده میشود و مقادیر وزن ها را به روز رسانی میکند. ابتدا difference با فرمول $\text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a)$ محاسبه میشود که در واقع تفاوتی است که q ی این state و action بعد از این ترنزیشن میتواند با فعلیش داشته باشد. این مقدار را در بروزرسانی وزن ها استفاده میکنیم و طبق فرمول $w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s,a)$ وزن های جدید را محاسبه میکنیم.

از عامل یادگیری q تقریبیمان در همام smallGrid قبلی استفاده میکنیم و مشاهده میشود بازهم پکمن ما سیاست بهینه را یادمیگیرد و به درستی همه ی بازی ها را میبرد. تفاوت کوچکی ددر میانگین امتیاز ها داریم که با این یادگیری تقریبی 0.4 امتیازمان بیشتر شده است.

python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

```
Average Score: 500.2
Scores:        503.0, 495.0, 503.0, 499.0, 499.0, 503.0, 503.0, 499.0, 495.0, 503.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

حال برای درک بهتر تاثیر مثبت این q-learning و کاهش نیاز به تعداد بالای بازی های آزمایشی، از کلاس SimpleExtractor برای استخراج ویژگی ها استفاده میکنیم. توضیحات این کلاس به شکل زیر است:

```
class SimpleExtractor(FeatureExtractor):
    """
    Returns simple features for a basic reflex Pacman:
    - whether food will be eaten
    - how far away the next food is
    - whether a ghost collision is imminent
    - whether a ghost is one step away
    """
```

برای پکمن 4 ویژگی را توسط توابعی محاسبه میکند و وکتور این مقادیر را برمیگرداند.

1-با انجام action از state آیا غذایی خورده میشود؟

2-با انجام action از state ، نزدیکترین غذای بعدی چقدر دور است؟

3-آیا بین روح ها گیر افتاده ایم؟

4-آیا یک روح در یک قدمیمان قرار دارد؟

جواب سوالات بالا را به شکل یک وکتور برمیگرداند و ما از این وکتور برای محاسبه q-value ها استفاده میکنیم.

با استفاده از ویژگی های بالا، دوباره برای mediumGrid یادگیری را اجرا میکنیم اما اینبار برخلاف بخش قبل ، یادگیری q تقریبی است. میبینیم در تعداد بازی های بسیار کمتر، به سیاست بهینه رسیده و همه ی بازی ها را توانسته برنده شود.

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

```
Average Score: 527.6
Scores:        529.0, 525.0, 529.0, 529.0, 527.0, 527.0, 529.0, 527.0, 527.0, 527.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

استفاده از یادگیری تقریبی فقط حل mediumGrid را برایمان ساده تر نمیکند بلکه گرید های سخخت تر بزرگتر را نیز به سرعت میتواند حل کند و سیاست بهینه را برای برنده شدن در این گرید ها پیدا کند. mediumClassic گرید بزرگی است اما باز هم در 50 ازمایش پکمن به خوبی سیاست را پیدا کرده و میتواند در 90 درصد مواقع برنده شود. با توجه به بزرگتر بودن گرید ازمایش ها بیشتر طول میکشند حال فرض کنید مجبور بودیم جای 50 ازمایش 50000 ازمایش برای یادگیری انجام دهیم (بدونq-learning تقریبی) قطعا وقت بسیار زیادی صرف فقط یادگیری میشد و اینجاست که اهمیت approximate q-learning را درک میکنیم.

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic

```
Average Score: 1187.1
Scores:        1337.0, 1303.0, 1330.0, 1329.0, 1332.0, 1341.0, 1328.0, 1292.0, -42.0, 1321.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win
```