MDP & REINFORCMENT LEARNING

Fundamental of artificial intelligence

Rojina kashefi-9831118

۱) تکرار ارزش (value iteration)

میدانیم برای محاسبه ارزش‌ها باید از روابط زیر استفاده کرد که Q(s,a) مقادیر خروجی به ازای یک حالت و action خاص را بسته به ارزش مرحله با عمق کمتر محاسبه میکند و زمانی که ما تمامی Qها را برای یک حالت مشخص کردیم ماکسیمم آنها میشود مقادیر ارزش‌ها.

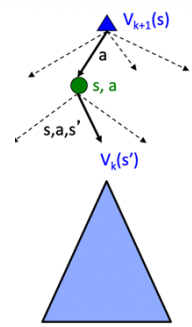
$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

حال رابطه تکرار ارزش فوق تا بی‌نهایت ادامه پیدا میکند و برای جلوگیری از اینکار مشخص میکنیم تا چه عمقی پیش برود و رابطه به رابطه زیر تغییر پیدا میکند.

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



مشاهده میکنیم که مانند درخواست پروژه برای محاسبه ارزش عمق kاز عمق k-1 استفاده میکنیم و از نسخه batch تکرار ارزش‌ها استفاده شده است.

- توضیحات کد:

برای اجرای قسمت ارزش‌ها پس از مقداردهی کردن تخفیف و تعداد پیمایش‌ها که تعداد تکرارهای تکرار ارزش است برنامه تابع زیر را اجرا میکند که با استفاده از تابع getstates کلاس mdp کل حالات که شامل مختصات x و y نقاط مختلف برنامه است را میگیرد .(بدیهی است حالات در زمین‌های مختلف است) و سپس تعداد تکرارها را میگیرد که و به تعداد تکرارهایی که برنامه از ما خواسته ارزش های را حساب میکنیم.در هر تکرار یک values جدید در نظر میگیریم و با مقادیر واقعی کاری نداریم تا در صورت تغییر انها برنامه دچار مشکل نشود. سپس بسته به اینکه هر حالت چه action هایی میتواند انجام دهد به محاسبه Q میپردازیم و برای محاسبه ارزش طبق روابط فوق باید از مقادیر Q ماکسیمم بگیریم و ارزش ها را در کل برنامه اپدیت میکنیم.
دقت شود چون در حالت terminal هیچ action موجود نیست برای بهتر کردن عملکرد برنامه ان را جدا کردیم.

```
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    all_states = self.mdp.getStates()
    num_iterations = self.iterations
    for iteration in range(num_iterations):
        values = util.Counter()  # before each iteration, copy values to not work with real one.
        for state in all_states:
            if self.mdp.isTerminal(state):
                continue
            compute_value = util.Counter()  # make a dictionary for all possible values for each action
            for action in self.mdp.getPossibleActions(state):
                each_action_value = self.computeQValueFromValues(state, action)
                compute_value[action] = each_action_value
            values[state] = max(compute_value.values())
        self.values = values
```

برای محاسبه کردن مقادیر Q طبق روابط فوق از reward و transition و حالت قبلی که محاسبه کردیم و در عمق پایینتر قرار دارد استفاده کردیم (batch). در محاسبه Q برای هر action و حالت چون در محیط عدم قطعیت داریم باید احتمال هر مسیر بعدی را حتی اگر کم محاسبه کنیم.

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

```python
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    q_value = 0
    transitions = self.mdp.getTransitionStatesAndProbs(state, action)
    for next_state, probability in transitions:
        reward = self.mdp.getReward(state, action, next_state)
        q_value += probability * (reward + (self.discount * self.values[next_state]))
    return q_value
```

برای محاسبه action از روی مقادیر از رابطه زیر استفاده میکنیم.

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

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

ابتدا Q ها را حساب میکنیم و با استفاده از تابع argmax سیاست یا action که موجب ماکسیمم شدن Q شده بود را بدست می آوریم.

```python
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    # if self.mdp.isTerminal(state):
    #     return None
    actions = self.mdp.getPossibleActions(state)
    q_values = util.Counter()
    if self.mdp.isTerminal(state):
        return
    for action in actions:
        q_values[action] = self.computeQValueFromValues(state, action)
    return q_values.argMax()
```

- خروجی قسمت یک:

  - خروجی autograder در محیط های مختلف:

```
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 ###


Finished at 11:34:07

Provisional grades
==================
Question q1: 4/4
------------------
Total: 4/4
```

  - خروجی دستور python gridworld.py -a value -i 100 -k 10:

مطابق دستور کار باید میانگین پاداش تجربی حاصل بعد از اتمام ۱۰ دور با مقدار حالت شروع Vstart که در رابط گرافیکی نشان میدهد تقریبا برابر باشد که مشاهده میکنیم.

```
EPISODE 10 COMPLETE: RETURN WAS 0.5904900000000002


AVERAGE RETURNS FROM START STATE: 0.4935233630639611
```

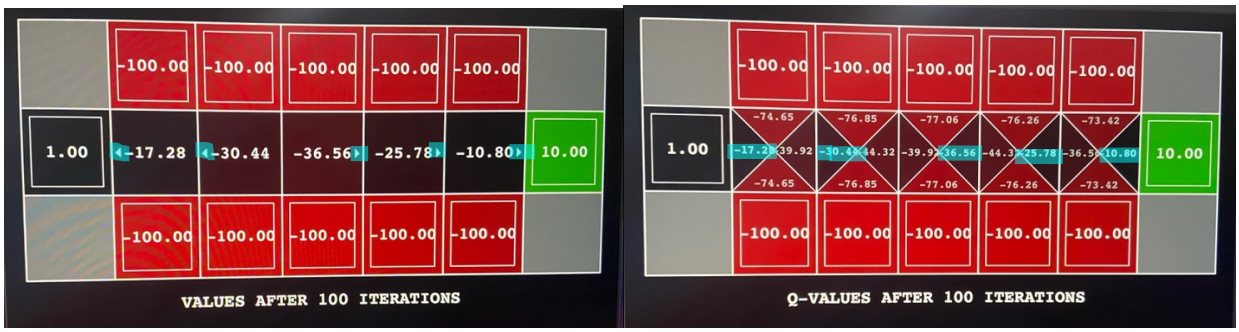
  - خروجی دستور python gridworld.py -a value -i 5:

اجرای تکرار مقدار برای ۵ تکرار خروجی مطابق با دستور کار را میدهد.

۲) تجزیه و تحلیل عبور از پل (bridge crossing analysis)

- با اجرای دستور noise 0.2 -- python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 :

زمانی که نویز بالا باشد در این حالت یا عامل حرکت نمیکند یا به سمت پایانه با پاداش کمتر حرکت میکند .




```
Got reward: 1

EPISODE 1 COMPLETE: RETURN WAS 0.9


AVERAGE RETURNS FROM START STATE: 0.9
```

برای حل این مشکل یا باید تخفیف را نزدیک به ۱ میکردیم که به صورت خودکار ۰.۹ بود و نزدیک بوده یا باید نویز که موجب میشود به حالت ناخواسته ای برسیم را کاهش میدادیم و به یک مقدار بسیار نزدیک به صفر یا خود صفر قرار میدادیم که هیچ نویزی نباشد.

- با اجرای دستور noise 0.0 -- python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9:

مشاهده میکنیم با تغییر دادن نویز به مقدار صفر میانگین پاداش تجربی حاصل بعد از اتمام ۱۰۰ دور با مقدار حالت شروع Vstart که در رابط گرافیکی نشان میدهد تقریبا برابر باشد و عامل به سمت حالت پایانه با پاداش بزرگ حرکت میکند.

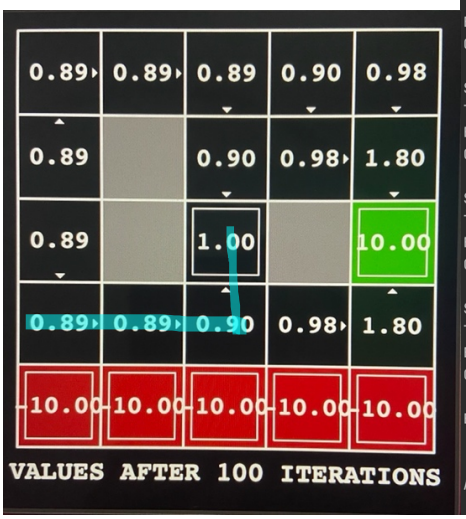
```
Got reward: 10

EPISODE 1 COMPLETE: RETURN WAS 5.904900000000001


AVERAGE RETURNS FROM START STATE: 5.904900000000001
```

۳) سیاست‌ها(policy)

a) خروجی نزدیک را ترجیح دهد(1+) و خطر صخره را ریسک کند (10-) .

```python
def question3a():
    answerDiscount = 0.1
    answerNoise = 0.0
    answerLivingReward = 0.8
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

برای اینکه خروجی نزدیک را ترجیح دهد باید تخفیف(discount) کمی داشته باشد.
زمانی که تخفیف کوچک باشد یعنی با گذر مراحل زیاد احتمال رسیدن به پاداش کم میشود ولی اگر تخفیف زیاد باشد یعنی با گذراندن مراحل زیاد هنوز احتمال رسیدن به پاداش وجود دارد.
از آنجایی که ما میخواهیم پاداش کوچک را به پاداش بزرگ ترجیح دهیم باید احتمال رسیدن به پاداش بزرگ را با کم کردن مقدار تخفیف، کم کنیم.
مقدار نویز را کم میگذاریم که به حالت جانشین ناخواسته نرسیم و مسیر را با ماکسیمم قطعیت پیش ببریم.
و مقدار پاداش زندگی را بیشتر از خطر صخره میگذاریم و نزدیک پاداش خروجی نزدیک که عامل با خود بگوید حتی اگر در صخره بیافتم بازم اگر یک حرکت کنم پاداشی بهتر از صخره و نزدیک خروجی نزدیک میگیرم و موجب حرکت عامل میشود در مسیرخواسته شده و انجام ریسک میشود.




```
RUNNING 1 EPISODES

BEGINNING EPISODE: 1

Started in state: (0, 1)
Took action: east
Ended in state: (1, 1)
Got reward: 0.8

Started in state: (1, 1)
Took action: east
Ended in state: (2, 1)
Got reward: 0.8

Started in state: (2, 1)
Took action: north
Ended in state: (2, 2)
Got reward: 0.8

Started in state: (2, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 1

EPISODE 1 COMPLETE: RETURN WAS 0.8890000000000001


AVERAGE RETURNS FROM START STATE: 0.8890000000000001
```

b) خروجی نزدیک را ترجیح دهد(1+) و خطر صخره را اجتناب کند (10-).

```python
def question3b():
    answerDiscount = 0.3
    answerNoise = 0.2
    answerLivingReward = 0.2
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

مقدار تخفیف نیز باید بیشتر شود تا عامل بداند اگر کمی مسافت و مراحل بیشتر شود تاثیر تخفیف در مقدار کاهش پیدا میکند و با بتواند چند مرحله بیشتر حرکت کند. اما تخفیف نباید خیلی زیاد شود که عامل تا پاداش بزرگ برود.
در این قسمت باید مقدار نویز بیشتر شود که احتمال قرار گرفتن در شرایط ناخواسته بیشتر باشد و عامل بیشتر احتیاط کند و بداند احتمالی برای ان وجود دارد.
در خصوص پاداش زندگی باید کمتر باشد که عامل بداند که عامل با توجه به زیاد شدن نویز و کاهش پاداش زندگی ریسک نکند و از مسیر دیگری به پاداش کوچک برسد.

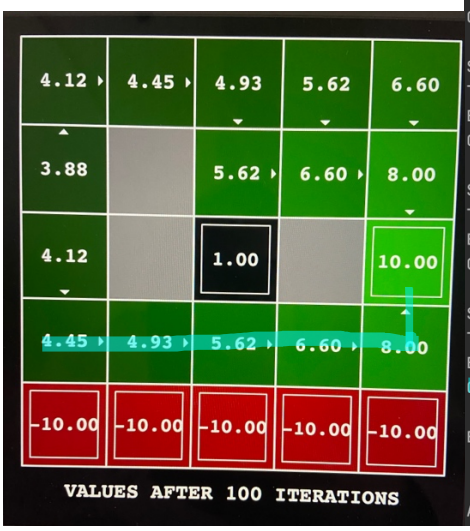
```
RUNNING 1 EPISODES

BEGINNING EPISODE: 1

Started in state: (0, 1)
Took action: north
Ended in state: (0, 2)
Got reward: 0.2

Started in state: (0, 2)
Took action: north
Ended in state: (0, 3)
Got reward: 0.2

Started in state: (0, 3)
Took action: north
Ended in state: (0, 4)
Got reward: 0.2

Started in state: (0, 4)
Took action: east
Ended in state: (1, 4)
Got reward: 0.2

Started in state: (1, 4)
Took action: east
Ended in state: (2, 4)
Got reward: 0.2

Started in state: (2, 4)
Took action: south
Ended in state: (2, 3)
Got reward: 0.2

Started in state: (2, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 1

EPISODE 1 COMPLETE: RETURN WAS 0.2858705


AVERAGE RETURNS FROM START STATE: 0.2858705
```

c) خروجی دور را ترجیح دهد (10+) و خطر صخره را ریسک کند (10-).

```python
def question3c():
    answerDiscount = 0.7
    answerNoise = 0.0
    answerLivingReward = 1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

در این حالت چون مسیر طولانی‌تر شده مقدار تخفیف باید بیشتر شود تا با گذراندن مراحل بیشتر مقدار پاداش نهایی خیلی تغییری نکند و هنوز عامل علاقه‌مند به نزدیک شدن به آنرا داشته باشد.

سپس چون در لبه‌های حساسی هستیم مقدار نویز باید کم باشد و مقدار پاداش زندگی به ازای هر مرحله باید به گونه‌ای باشد که عامل ریسک کند و بگوید حتی اگر در ۱۰- بیافتم در هرمرحله کلی پاداش جذب کردم که حرکت کند.




```
BEGINNING EPISODE: 1

Started in state: (0, 1)
Took action: east
Ended in state: (1, 1)
Got reward: 1.0

Started in state: (1, 1)
Took action: east
Ended in state: (2, 1)
Got reward: 1.0

Started in state: (2, 1)
Took action: east
Ended in state: (3, 1)
Got reward: 1.0

Started in state: (3, 1)
Took action: east
Ended in state: (4, 1)
Got reward: 1.0

Started in state: (4, 1)
Took action: north
Ended in state: (4, 2)
Got reward: 1.0

Started in state: (4, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 10

EPISODE 1 COMPLETE: RETURN WAS 4.453799999999999


AVERAGE RETURNS FROM START STATE: 4.453799999999999
```

d) خروجی دور را ترجیح دهد (+10) و خطر صخره را اجتناب کند (-10).

در این مرحله چون مسیر طولانی تر شده عامل باید تخفیف بیشتری داشته باشد تا در حین گذراندن مراحل در مسیر برای رسیدن به هدف از پاداش هدف خیلی کم نشود.
همچنین به دلیل وجود نویز بالا و پاداش زندگی کمتر عامل از مسیر پرخطر اجتناب میکند و ریسک نمیکند.

```python
def question3d():
    answerDiscount = 0.9
    answerNoise = 0.5
    answerLivingReward = 0.6
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

```
BEGINNING EPISODE: 1

Started in state: (0, 1)
Took action: north
Ended in state: (0, 1)
Got reward: 0.6

Started in state: (0, 1)
Took action: north
Ended in state: (0, 2)
Got reward: 0.6

Started in state: (0, 2)
Took action: north
Ended in state: (0, 2)
Got reward: 0.6

Started in state: (0, 2)
Took action: north
Ended in state: (0, 2)
Got reward: 0.6

Started in state: (0, 2)
Took action: north
Ended in state: (0, 3)
Got reward: 0.6

Started in state: (0, 3)
Took action: north
Ended in state: (0, 4)
Got reward: 0.6

Started in state: (0, 4)
Took action: east
Ended in state: (1, 4)

Started in state: (1, 4)
Took action: east
Ended in state: (2, 4)
Got reward: 0.6

Started in state: (2, 4)
Took action: east
Ended in state: (3, 4)
Got reward: 0.6

Started in state: (3, 4)
Took action: east
Ended in state: (4, 4)
Got reward: 0.6

Started in state: (4, 4)
Took action: south
Ended in state: (4, 3)
Got reward: 0.6

Started in state: (4, 3)
Took action: south
Ended in state: (4, 3)
Got reward: 0.6

Started in state: (4, 3)
Took action: south
Ended in state: (4, 2)
Got reward: 0.6

Started in state: (4, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 10

EPISODE 1 COMPLETE: RETURN WAS 6.823564528378598

AVERAGE RETURNS FROM START STATE: 6.823564528378598
```

e) از هر دو خروجی و صخره اجتناب کنید .(بنابراین اجرای این قسمت هرگز نباید پایان یابد).

```
def question3e():
    answerDiscount = 1
    answerNoise = 0.01
    answerLivingReward = 10
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

مشاهده میکنیم که مقدار تخفیف خیلی بالا در نظر گرفته شده که عامل نگران میزان مراحلی که میگذرد نباشید و بازی پایان نیابد و عمر محدود نداشته باشد. همچنین نویز خیلی پایین گرفته شده که عامل قابلیت ریسک پذیری نیز داشته باشد و پاداش زندگی بالا در نظر گرفته شده که عامل به ازای هر مرحله که حرکت میکند پاداش زیادی دریافت کند به اندازه پاداش بزرگ که خیلی مایل به رسیدن به پاداش بزرگ اصلی نباشد و برای خود بچرخد.

۴)تکرار ارزش ناهمزمان (Asynchronous Value Iteration)

در این قسمت برخلاف قسمت قبل که همه value ها بعد از هر مرحله عوض می‌شد در اینجا در هر مرحله فقط مقادیر یک حالت عوض میشود. که مشاهده میکنیم که اگر تمامی حالات در یک زمین تماما برسی شد و هنوز پیمایش باقی مانده از اول مورد بررسی قرار گیرد و همچنین اگر حالت اتفاق ترمینال باشد در آن تکرار هیچ اتفاق نمی‌افتد و اگر حالت ترمینال نباشد یک دیکشنری برای ان حالت در نظر میگیریم و key ان میشود سیاست و یا عمل هایی که برای آن حالت مجاز است و مقادیر آنها میشود Q هایی که با استفاده از ارثبری از قسمت یک محاسبه میکنیم و سپس مقدار ان حالت خاص را فقط به مقدار ماکسیمم Q ها تغییر میدهیم.

```
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    states = self.mdp.getStates()
    for iteration in range(self.iterations):
        state = states[iteration % len(states)]
        if self.mdp.isTerminal(state):
            continue
        one_state = util.Counter()
        for action in self.mdp.getPossibleActions(state):
            one_state[action] = self.computeQValueFromValues(state, action)
        self.values[state] = max(one_state.values())
```

- خروجی کد:

مطابق دستور کار پس از اجرای دستور python gridworld.py -a asynchvalue -i 1000 -k 10 مشاهده میکنیم مقدار حالت شوع شده vstart که از رابط گرافیکی میخوانیم و میانگین پاداش تجربه حاصل بسیار نزدیک هستند.

```
Got reward: 1

EPISODE 10 COMPLETE: RETURN WAS 0.5904900000000002


AVERAGE RETURNS FROM START STATE: 0.4737017068200001
```

مشاهده میکنیم مطابق به دستور کار autograder کمتر از ۱ ثانیه طول کشیده است.

```
Starting on 1-21 at 18:51:32

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 18:51:32

Provisional grades
==================
Question q4: 1/1
------------------
Total: 1/1
```

۵) تکرار ارزش اولویت‌بندی شده (prioritized sweeping value iteration)

- توضیحات کد

```python
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    states = self.mdp.getStates()
    priority_queue = util.PriorityQueue()
    predecessor = {}
    for state in states:
        if self.mdp.isTerminal(state):
            continue
        q_values = {}
        for action in self.mdp.getPossibleActions(state):
            for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action):
                if next_state not in predecessor:
                    predecessor[next_state] = set()
                predecessor[next_state].add(state)
            q_values[action] = self.getQValue(state, action)
        diff = abs(self.values[state] - max(q_values.values()))
        priority_queue.update(state, -diff)
    for iteration in range(self.iterations):
        if priority_queue.isEmpty():
            break
        state = priority_queue.pop()
        if self.mdp.isTerminal(state):
            continue
        update_values = {}
        for action in self.mdp.getPossibleActions(state):
            update_values[action] = self.getQValue(state, action)
        self.values[state] = max(update_values.values())
        for p in predecessor[state]:
            q_values = {}
            for action in self.mdp.getPossibleActions(p):
                q_values[action] = self.getQValue(p, action)
            diff = abs(self.values[p] - max(q_values.values()))
            if diff > self.theta:
                priority_queue.update(p, -diff)
```

یک صف خالی برای نگه‌داری اولویت‌ها تعریف کنید.

برای هر حالت غیرپایانی s

برای هر حالت،همه پسین‌ها مشخص شود.

هنگام محاسبه پسین‌های یک حالت،برای جلوگیری از به وجودآمدن حالت‌های تکراری،آنها را درمجموعه نگهداری کنید.

مقادریر Q ممکن از حالت s که با استفاده از اقدام‌های ممکن قابل تعریف است.

قدرمطلق تفاضل بین مقدار فعلی s که در self.values نگه‌داری میشود و بیشترین مقدار Q را محاسبه کنید.این مقدار را diff بنامید.

حالت s را با اولویت diff- به صف اولویت اضافه کنید.

به ازای تعداد تکرارهای مشخص شده self.iterations.

اگر صف اولویت خالی می‌باشد کار پایان یافته است.

به ازای هر پسین از حالت s

درغیر این صورت حالت s را از صف اولویت بردارید.

در صورتی که sحالت پایانی نبود، مقدار حالت s را در self.values به روز رسانی کنید.

قدر مطلق تفاضل بین مقدار فعلی حالت p که در self.values نگه داری میشود و بیشترین مقدار Q ممکن از حالت p که با استفاده از اقدام‌های ممکن قابل تعریف است را محاسبه کنید و این مقدار را diff بنامید.

اگر diff>theta بود حالت p را با اولویت diff- به صف اولویت اضافه کنید. در صورتی که حالت p با اولویت مساوی کمتر در صف وجود نداشته باشد.

- خروجی کد

مشاهده میکنیم مطابق به دستور کار autograder کمتر از ۱ ثانیه طول کشیده است.

```
Starting on 1-21 at 20:26:48

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 20:26:48

Provisional grades
==================
Question q5: 3/3
------------------
Total: 3/3
```

و پس از اجرای دستور python gridworld.py -a priosweepvalue -i 1000 مشاهده میکنیم:

۶) یادگیری Q

در واقعیت اگر بخواهند مقادیر را اندازه گیری کنند نیاز به transition function که همان احتمالات است دارند چون Q به ان وابسته است پس به جای محاسبه value به یادگیری Q میپردازند که مراحل ان به صورت زیر است:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

Learn Q(s,a) values as you go

- Receive a sample (s,a,s',r)
- Consider your old estimate: $Q(s,a)$
- Consider your new sample estimate:

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

- Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

- توضیحات کد

ابتدا مطابق مستندات درون کد در تابع init یک q_value تعریف میکنیم و سپس در تابع getQvalue مقدار Q را با توجه به حالتی که هستیم و action که داریم از دیکشنری q_value برمیگردانیم.دقت شود چون q_values از نوع util.counter است زمانی که یک حالت یا ترکیب حالت و سیاست را ندیده بودیم به طور خودکار صفر برمیگرداند.

```
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)
    self.q_values = util.Counter()
    "*** YOUR CODE HERE ***"

def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we have never seen a state
      or the Q node value otherwise
    """
    "*** YOUR CODE HERE ***"
    return self.q_values[(state, action)]
```

سپس برای محاسبه مقادیر از روی Q همان روند گذشته را میرویم با این تفاوت که مطابق خواسته این قسمت از تابع getlegalactions استفاده میکنیم و با استفاده از تابع getQvalues به محاسبه مقادیر Q برای action هایی که داریم میپردازیم و برای value بین مقادیر Q های موجود ماکسیمم میگیریم و اگر عملی مجاز نبود صفر برمیگردانیم.

```
def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
    "*** YOUR CODE HERE ***"
    if self.getLegalActions(state):
        q_values = util.Counter()
        for action in self.getLegalActions(state):
            q_values[action] = self.getQValue(state, action)
        return max(q_values.values())
    return 0.0
```

برای محاسبه action ها از مقادیر Q ابتدا value ها را بدست می‌اوریم و سپس به ازای action هایی که مجاز هستیم انهایی که که اگر Q هایشان را حساب کنیم برابر value شود. بین آنها که اکشن‌هایی هستند که موجب ماکسیمم شدن value شده‌اند با استفاده از تابع random.choice انتخاب میکنیم.

```
def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    "*** YOUR CODE HERE ***"
    if self.getLegalActions(state):
        best_q_values = self.computeValueFromQValues(state)
        actions_max_qValues = []
        for action in self.getLegalActions(state):
            if best_q_values == self.getQValue(state, action):
                actions_max_qValues.append(action)
        return random.choice(actions_max_qValues)
    return None
```

و در تابع update نیز مطابق فرمولی که در بالا اورده شده مقادیر Q ها با استفاده از الفا، تخفیف و پاداش محاسبه میشود.

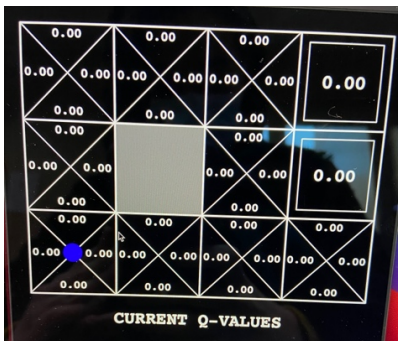
```
def update(self, state, action, nextState, reward):
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here

      NOTE: You should never call this function,
      it will be called on your behalf
    """
    "*** YOUR CODE HERE ***"

    self.q_values[(state, action)] = (1 - self.alpha) * self.getQValue(state, action) + \
                                     self.alpha * (reward + self.discount * self.getValue(nextState))
```

- ## خروجی قسمت ششم

با اجرای دستور زیر مشاهده میکنیم:

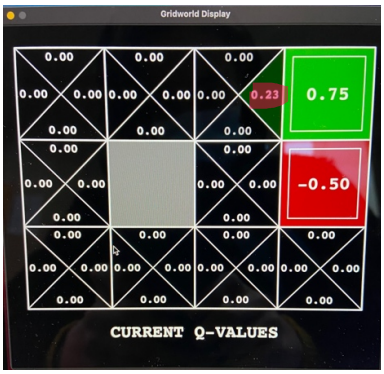
```
python gridworld.py -a q -k 5 -m
```

مشاهده میکنیم ابتدا چون اقداماتی را داریم که عامل قبلا ندیده است تمام مقادیر اولیه Q مطابق دستور کار برابر صفر قرار داده شده است.



سپس در مرحله بعدی با حرکت به صورت دستی مشاهده میکنیم مقادیر پاداش های نهایی بزرگ را مرحله مرحله بدست می آورد.

و پس از هر مرحله مقادیر Q ها را اپدیت میکند و یاد میگیرد.مشاهده میکنیم ابتدا در خانه بالا چون پاداش برابر ۱ است و الفا برابر ۰.۵ است و با محاسبه .q مطابق قسمت بالا ۰.۵ در میاید و سپس اگر بین ۰.۵ و ۱ با الفای ۰.۵ معادله را حل کنیم برابر ۰.۷۵ میشود.



۷) epsilon حریصانه

یکی از راهترین راه ها برای اکتشاف کردن استفاده از الگوریتم epsilon greedy است.
در هر مرحله یک سکه پرتاب میکنیم.با احتمال های کوچک (epsilon) رندوم رفتار میکینم و با احتمال‌های بزرگ بر اساس سیاست فعلی رفتار میکنیم.
انتخاب یک اقدام تصادفی ممکن است منجر به انتخاب بهترین اقدام شود و یعنی میتوانید از هر اقدام تصادفی مجازی انتخاب کنید.
مشکلی که این الگوریتم دارد در گذر زمان درست است اکتشاف صورت میگیرد اما زمانی که یادگیری تمام شود عامل هنوز به صورت رندوم عمل میکند و ممکن است از بین میرود و راهکاری که میتوانیم برای این مشکل ارائه دهیم که epsilon را در گذر زمان کاهش دهیم.

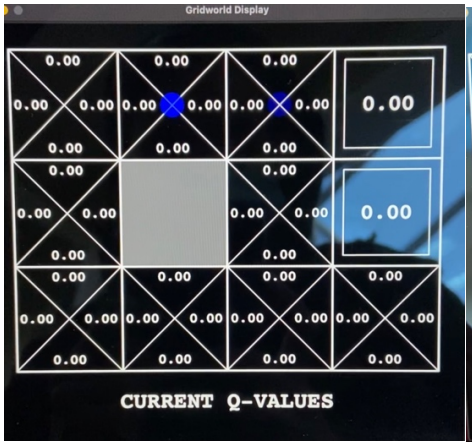
- توضیحات کد :

اگر action مجاز وجود داشته باشد بسته به مقدارepsilon یا از اکشن‌های مجاز به صورت رندوم انتخاب میکنیم یا بر اساس بهترین مقادیر Q فعلی انتخاب میکنیم.در صورتی که هیچ action مجازی نداشته باشیم NONE بر میگردانیم.

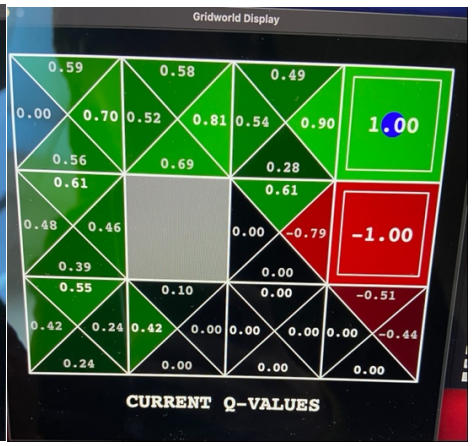
```python
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.

      HINT: You might want to use util.flipCoin(prob)
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    if legalActions:
        if util.flipCoin(self.epsilon):
            return random.choice(legalActions)
        return self.computeActionFromQValues(state)
    return action
```

با اجرای دستور زیر مشاهده میکنیم که عامل در طول زمان در حال یادگیری محیط است.

```
python gridworld.py -a q -k 100 -e 0.3
```



1

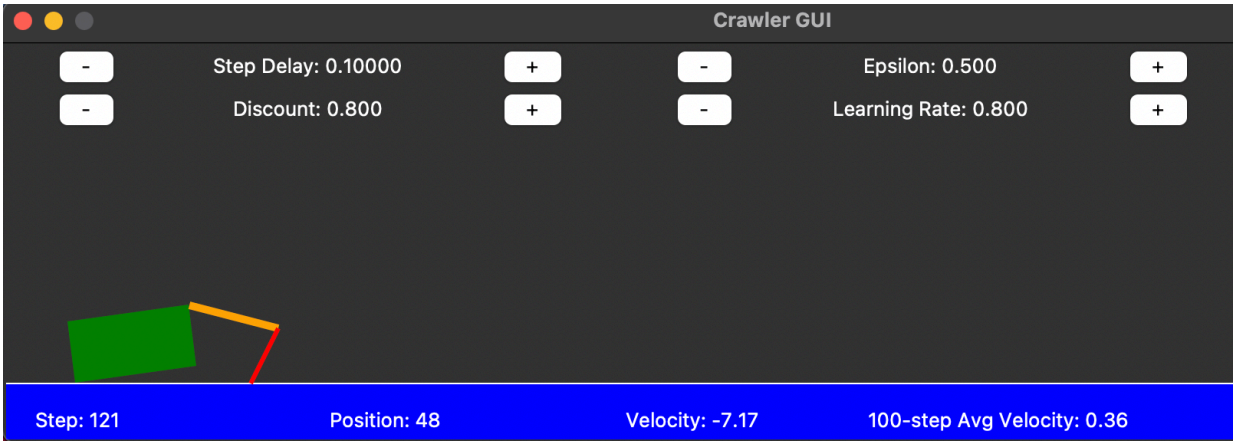


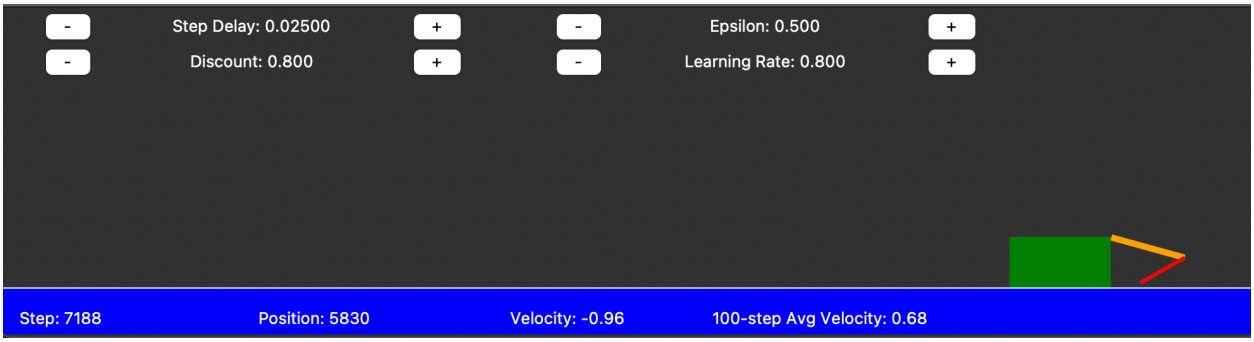
2

3 4

و با دستور زیر مشاهده میکنیم عامل در حال یادگیری حرکت بازوانش برای حرکت است و سپس اینکار را ادامه میدهد و تا انتها حرکن میکند و چون اپسیلون حریصانه است درحال یادگیری است.اگر اپسیلون زیاد کنیم موجب میشود کندتر شود و رندومانه حرکت نکند.

```
python crawler.py
```





۸) بررسی دوباره عبور از پل

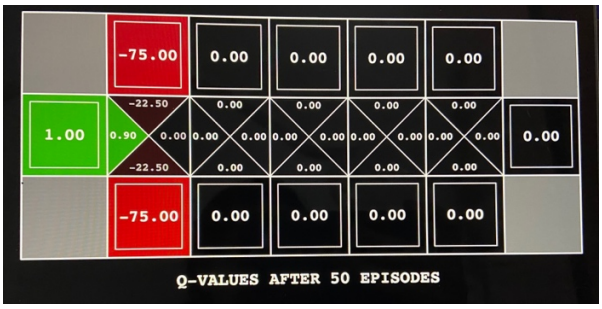
با اجرای دستور

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```

مشاهده میکنیم که عامل در ۵۰ مرحله در حال یادگیری است که در این حالت سیاست بهینه یافت نشد و به شکل زیر رسیدیم.
در شکل زیر بسیاری از حالت بررسی نشده اند و در حا ل حاضر به سمت پایانه با پاداش کمتر حرکت میکند چون پایانه با پاداش بزرگ هنوز کشف نشده و عامل کاملا رندومانه رفتار میکند.

حال اگر همان دستور بالا با epsilon صفر انجام دهیم مشاهده میکنیم که عامل تنها بین دو خانه جابه جا میشود و از حالت قبل بدتر است زیرا موجب میشود ۱-اپسیلون ماکسیمم شود که یعنی بر اساس سیاست رفتار کن و نه رندومانه.

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 0
```



به ازای هیچ مقدار اپسیلون و نرخ یادگیری در ۵۰ مرحله یادگیری حالت سیاست بهینه ممکن نبود.

```
def question8():
    answerEpsilon = None
    answerLearningRate = None
    # return answerEpsilon, answerLearningRate
    return 'NOT POSSIBLE'
    # If not possible, return 'NOT POSSIBLE'
```

۹) پک‌من و Q-learning

پس از اجرای دستور فوق مشاهده میکنیم که عامل حدود ۲۰۰۰ بار مورد اموزش قرار میگیرد و سپس در ۱۰ مرحله وارد فاز ازمون میشود که در ان الفا و اپسیلون برای متوقف کردن فرایند یادگیری و محاسبه و تغییر Q-learning ها صفر میشوند و مشاهده میکنیم در همه بازی‌های ازمون پیروز شد.

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

```
Beginning 2000 episodes of Training
Reinforcement Learning Status:
        Completed 100 out of 2000 training episodes
        Average Rewards over all training: -510.15
        Average Rewards for last 100 episodes: -510.15
        Episode took 0.30 seconds
Reinforcement Learning Status:
        Completed 200 out of 2000 training episodes
        Average Rewards over all training: -506.77
        Average Rewards for last 100 episodes: -503.39
        Episode took 0.45 seconds
Reinforcement Learning Status:
        Completed 300 out of 2000 training episodes
        Average Rewards over all training: -505.55
        Average Rewards for last 100 episodes: -503.10
        Episode took 0.47 seconds
Reinforcement Learning Status:
        Completed 400 out of 2000 training episodes
        Average Rewards over all training: -487.21
        Average Rewards for last 100 episodes: -432.22
        Episode took 0.52 seconds
Reinforcement Learning Status:
        Completed 500 out of 2000 training episodes
        Average Rewards over all training: -455.95
        Average Rewards for last 100 episodes: -330.91
        Episode took 0.53 seconds
Reinforcement Learning Status:
        Completed 600 out of 2000 training episodes
        Average Rewards over all training: -426.50
        Average Rewards for last 100 episodes: -279.24
        Episode took 0.55 seconds
Reinforcement Learning Status:
        Completed 700 out of 2000 training episodes
        Average Rewards over all training: -409.95
        Episode took 0.54 seconds
Reinforcement Learning Status:
        Completed 1500 out of 2000 training episodes
        Average Rewards over all training: -165.19
        Average Rewards for last 100 episodes: 247.08
        Episode took 0.56 seconds
Reinforcement Learning Status:
        Completed 1600 out of 2000 training episodes
        Average Rewards over all training: -146.39
        Average Rewards for last 100 episodes: 135.62
        Episode took 0.56 seconds
Reinforcement Learning Status:
        Completed 1700 out of 2000 training episodes
        Average Rewards over all training: -128.08
        Average Rewards for last 100 episodes: 164.86
        Episode took 0.59 seconds
Reinforcement Learning Status:
        Completed 1800 out of 2000 training episodes
        Average Rewards over all training: -109.53
        Average Rewards for last 100 episodes: 205.84
        Episode took 0.57 seconds
Reinforcement Learning Status:
        Completed 1900 out of 2000 training episodes
        Average Rewards over all training: -92.89
        Average Rewards for last 100 episodes: 206.56
        Episode took 0.56 seconds
Reinforcement Learning Status:
        Completed 2000 out of 2000 training episodes
        Average Rewards over all training: -75.90
        Average Rewards for last 100 episodes: 246.93
        Episode took 0.56 seconds
Training Done (turning off epsilon and alpha)
---------------------------------------------
```

```
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Average Score: 501.0
Scores:        503.0, 499.0, 495.0, 503.0, 503.0, 503.0, 503.0, 499.0, 503.0, 499.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

۱۰) Q-learning تقریبی (Approximate Q-learning)

در واقعیت به جای اینکه خود مقادیر Q را محاسبه و در یکجا در یک جدول ذخیره کنیم، آنها را تخمین میزنیم. چون محاسبه Q ها چه از نظر حافظه و زمان محاسبه انها و تعداد زیاد حالت ها در واقعیت غیر ممکن است.
در این حالت ما یک سری حالت را امتحان میکنیم و سپس تلاش میکنیم که در شرایط مشابه تخمین بزنیم که این شبیه حالات امتحان شده است.
در این حالت میدانیم مقادیر Q تابعی از ویژگی ها با ضرایبشان است.و برای محاسبه ضرایب با یک سری ضرایب اولیه شروع میکنیم و آنها را با استفاده از اکشن‌هایی که از قبل انجام دادیم با توجه به reward های انها محاسبه میکنیم.

Q-learning with linear Q-functions:

transition $= (s, a, r, s')$

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

$Q(s, a) \leftarrow Q(s, a) + \alpha$ [difference] Exact Q's

$w_i \leftarrow w_i + \alpha$ [difference] $f_i(s, a)$ Approximate Q's

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a)$$

همانطور که در بالا توضیح داده شده بود مقدار Q برابر است با جمع وزنی ویژگی ها .

$$Q(s, a) = \sum_{i=1}^{n} f_i(s, a) w_i$$

```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    features = self.featExtractor.getFeatures(state, action)
    return sum([features[feature] * self.weights[feature] for feature in features])
```

و برای اپدیت کردن وزن ها ابتدا به محاسبه difference مطابق فرمول فوق حساب کردیم و سپس مطابق رابطه فوق به اپدیت کردن وزن‌ها پرداختیم.

```
def update(self, state, action, nextState, reward):
    """
       Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    difference = (reward + (self.discount * self.getValue(nextState))) - self.getQValue(state, action)
    features = self.featExtractor.getFeatures(state, action)
    for feature in features:
        self.weights[feature] = self.weights[feature] + (self.alpha * difference * features[feature])
```

- مشاهده خروجی ها:

با استفاده از اجرای دستور زیر مشاهده میکنیم.

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

در قسمت قبل نیز پس از ۲۰۰۰ مرحله اموزش و ۱۰ مرحله ازمون همه مراحل را برد و در این مرحله نیز همان اتفاق با میانگین ۱ امتیاز بیشتر اتفاق افتاد.

```
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Average Score: 502.2
Scores:        503.0, 495.0, 503.0, 503.0, 503.0, 503.0, 503.0, 503.0, 503.0, 503.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

با اجرا کردن برنامه با استفاده از simple extractor مشاهده میکنیم:

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

در تمامی مراحل در mediumGrid نیز برنده شد. که این خود به برتری approximate Q-learning اشاره میکند که در محیط های مختلف در حال پاسخگویی است.

```
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 523
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 525
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Average Score: 527.6
Scores:        529.0, 529.0, 527.0, 523.0, 529.0, 527.0, 529.0, 525.0, 529.0, 529.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

با اجرای برنامه در mediumclassic نیز با مشکلی مواجه نشدیم تقریبا در تمام بازی‌ها میبرد و تنها زمانبر بود.

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

```
Beginning 50 episodes of Training
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman emerges victorious! Score: 1340
Pacman emerges victorious! Score: 1333
Pacman emerges victorious! Score: 1328
Pacman died! Score: -324
Pacman emerges victorious! Score: 1336
Pacman emerges victorious! Score: 1503
Pacman emerges victorious! Score: 1337
Pacman emerges victorious! Score: 1316
Pacman emerges victorious! Score: 1318
Pacman emerges victorious! Score: 1327
Average Score: 1181.4
Scores:        1340.0, 1333.0, 1328.0, -324.0, 1336.0, 1503.0, 1337.0, 1316.0, 1318.0, 1327.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win
```