به نام خدا

دانشکده مهندسی کامپیوتر

مبانی و کاربردهای هوش مصنوعی ترم بهار ۱۴۰۱

پروژه سوم

مهلت تحویل ۶ خرداد ۱۴۰۱

مقدمه

در این پروژه شما **value iteration** و **Q-learning** را پیاده‌سازی خواهید کرد. شما عوامل خود را ابتدا در **Gridworld** آزمایش خواهید کرد، سپس آن‌ها را روی **Crawler** و **Pacman** اعمال خواهید کرد.

نسخه اصلی و به زبان انگلیسی پروژه را می‌توانید در این لینک مشاهده بفرمایید.

مانند پروژه‌های قبلی، برای دیباگ و تست درستی الگوریتم‌های خود می‌توانید دستور زیر را اجرا کنید:

```
python autograder.py
```

برای استفاده از **autograder.py** تنها برای بررسی یک سوال، می‌توانید از دستور زیر استفاده کنید:

```
python autograder.py -q q2
```

همچنین، می‌توان آن را برای یک تست خاص، به فرم زیر نیز اجرا کرد:

```
python autograder.py -t test_cases/q2/1-bridge-grid
```

ساختار پروژه بصورت زیر است و تمام فایل‌های مورد نیاز در یک فایل زیپ، در سامانه کورسز موجود خواهد بود:

فایل‌هایی که باید ویرایش کنید:	
valueIterationAgents.py	یک عامل **value iteration** برای حل **MDP** های شناخته شده.
qlearningAgents.py	عامل‌های **Q-learning** برای **Gridworld**، **Crawler** و **Pacman**.
analysis.py	فایلی برای قرار دادن پاسخ‌های شما به سوالات داده شده در پروژه.
فایل‌هایی که شاید بخواهید آن‌ها را ببینید:	
mdp.py	متدهایی را بر روی **MDP** های متداول تعریف می‌کند..
learrningAgents.py	کلاس‌های پایه **ValueEstimationAgent** و **QLearningAgent** را تعریف می‌کند که عامل‌های شما آن‌ها را گسترش خواهند داد.
util.py	ابزارهای کمکی، از جمله **util.Counter**، که مخصوصاً برای **Q-learner** ها قابل استفاده است.
gridworld.py	پیاده‌سازی **Gridworld**.
featureExtractors.py	کلاس‌هایی برای استخراج ویژگی‌ها در pair های (**state, action**). برای تقریب عامل **Q-learning** استفاده می‌شود (در **qlearningAgents.py**).

فایل‌هایی که می‌توانید آن‌ها را رد کنید:	
enviroment.py	کلاس **abstract** برای محیط‌های یادگیری تقویتی کلی. استفاده شده توسط **gridworld.py**.
graphicsGridworldDisplay.py	صفحه نمایش گرافیکی **Gridworld**
graphicsUtils.py	ابزارهای گرافیکی
textGridworldDisplay.py	پلاگین برای رابط متنی **Gridworld**.
crawler.py	کد **crawler** و تست. شما این را اجرا می‌کنید اما آن را ویرایش نمی‌کنید.
graphicsCrawlerDisplay.py	**GUI** برای ربات **crawler**
autograder.py	تصحیح‌کننده خودکار پروژه
testParser.py	Parse کردن تست‌های مصحح خودکار و فایل‌های راه‌حل
testClasses.py	کلاس‌های کلی تست خودکار
test_cases/	پوشه دربردارنده تست‌های مختلف برای هر سوال
reinforcementTestClasses.py	کلاس‌های تست خودکار پروژه سوم

آنچه باید انجام دهید:

شما باید بخش‌هایی از سه فایل **valueIterationAgents.py** و **qlearningAgents.py** و **analysis.py** را تکمیل کنید. لطفا سایر فایل‌ها را تغییر ندهید.

همچنین گزارشی **کامل** از نحوه پیاده‌سازی‌های انجام شده به همراه **تصاویری** از **کد** خودتان و **خروجی پروژه** تهیه کرده و در نهایت این **دو** فایل را در سامانه بارگذاری کنید.

توجه کنید که گزارش شما بخش بزرگی از نمره را تشکیل می‌دهد.

MDP ها!

پس از بارگیری کد پروژه از سامانه کورسز و خارج کردن آن‌ها از حالت فشرده، می توانید **Gridworld** را در حالت کنترل دستی، که از کلیدهای جهت دار استفاده می‌کند، اجرا کنید:

```
python gridworld.py -m
```

طرح **two-exit** که پیش از این در کلاس دیدید، را خواهید دید. نقطه آبی عامل است.
توجه داشته باشید که وقتی کلید بالا را فشار می‌دهید، عامل فقط در **80** درصد مواقع به سمت شمال حرکت می‌کند. زندگی یک عامل **Gridworld** چنین است!

شما می توانید بسیاری از جنبه‌های شبیه‌سازی را کنترل کنید. یک لیست کامل از گزینه‌های موجود با دستور زیر نمایش داده می‌شود:

```
python gridworld.py -h
```

عامل پیش‌فرض به صورت تصادفی حرکت می‌کند:

```
python gridworld.py -g MazeGrid
```

باید ببینید که عامل تصادفی در اطراف **grid** می‌چرخد تا زمانی که در یک خروجی بیوفتد.

نکته مهم: **Gridworld MDP** به گونه‌ای است که ابتدا باید وارد یک حالت **pre-terminal** شوید (جعبه‌های دوگانه نشان داده شده در رابط کاربری گرافیکی) و سپس **action** ویژه **«exit»** را قبل از پایان اپیزود انجام دهید (در حالت ترمینال واقعی به نام **TERMINAL_STATE**، که در رابط کاربری گرافیکی نشان داده نشده است). اگر یک اپیزود را به صورت دستی[1] اجرا کنید، به دلیل **discount rate**، ممکن است کل مقدار بازگشتی شما کم‌تر از حد انتظار شما باشد (**d-** برای تغییر دادن، **0.9** به طور پیشفرض).

[1] manually

به خروجی کنسول همراه با خروجی گرافیکی نگاه کنید (یا از **t-** برای همه متن استفاده کنید). در مورد هر انتقالی که عامل تجربه می‌کند به شما گفته می‌شود (برای خاموش کردن این، از **q-** استفاده کنید).

مشابه با **Pacman**، موقعیت‌ها با مختصات دکارتی **(x,y)** نشان داده می‌شوند و هر آرایه‌ای با **[x][y]** نمایه‌سازی می‌شود، به طوری که «شمال» جهت افزایش y است. به‌طور پیش‌فرض، اکثر انتقال‌ها پاداش صفر دریافت می‌کنند. هر چند می‌توانید این را با گزینه پاداش زنده **(r–)** تغییر دهید.

۱) تکرار ارزش[2] (۴ امتیاز)

معادله بروزرسانی حالت تکرار ارزش را به یاد بیاورید:

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

یک عامل تکرار ارزش در **ValueIterationAgent** بنویسید که تا حدی برای شما در **valueIterationAgents.py** مشخص شده است. عامل تکرار ارزش شما یک برنامه‌ریز آفلاین است و نه یک عامل یادگیری تقویتی. **training option** مربوط به تعداد تکرارهای تکرار ارزش است که باید در مرحله برنامه‌ریزی اولیه اجرا شود (**option -i**).

ValueIterationAgent یک **MDP** در **constructor** می‌گیرد و تکرار ارزش را برای تعداد مشخصی از تکرارها قبل از اتمام اجرای **constructor**، اجرا می‌کند.

تکرار ارزش، تخمین‌های **k** مرحله‌ای مقادیر بهینه، V_k را محاسبه می‌کند. علاوه بر اجرای تکرار ارزش، متدهای زیر را برای **ValueIterationAgent** با استفاده از V_k پیاده سازی کنید:

- تابع **computeActionFromValues** که بهترین عمل را با توجه به تابع مقدار[3] داده شده توسط **self.values** محاسبه می‌کند.
- تابع **computeQValueFromValues** که ورودی (**state, action**) را می‌گیرد و در ادامه **Q-value** را برای **action** در **state** با توجه به تابع مقدار ذخیره شده در **self.values** محاسبه می‌کند.

این اعداد همگی در رابط کاربری گرافیکی نمایش داده می‌شوند: مقادیر[4]، اعداد داخل مربع هستند. همچنین **Q-value** ها اعداد در ربع مربع و مقررات نیز فلش‌های خارج از هر مربع هستند.

مهم: از نسخه **"batch"** تکرار ارزش استفاده کنید که در آن هر بردار V_k از یک بردار ثابت $V_k - 1$ محاسبه می‌شود (مانند آنچه در اسلایدها و کلاس درس داشتید).

[2] Value Iteration
[3] Value function
[4] Values

راهنمایی: شما می‌توانید به صورت اختیاری از کلاس **util.Counter** در **util.py** استفاده کنید که یک دیکشنری با مقدار پیش فرض صفر است. با این حال، مراقب **argMax** باشید؛ **argmax** واقعی که شما می‌خواهید ممکن است کلیدی باشد که در شمارنده نیست!

توجه: مطمئن شوید حالتی که یک **state** هیچ **action** ممکنی در **MDP** ندارد را در پیاده‌سازی خود در نظر بگیرید. (به معنای این موضوع برای پاداش‌های آینده فکر کنید).

برای آزمایش پیاده‌سازی خود، **autograder** را اجرا کنید:

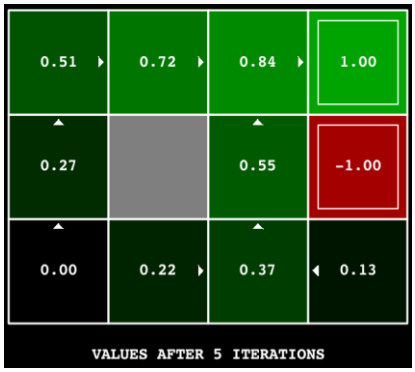
```
python autograder.py -q q1
```

دستور زیر **ValueIterationAgent** شما را لود می‌کند، که یک سیاست[5] را محاسبه و **10** بار اجرا می‌کند. دکمه‌ای را روی صفحه کلید فشار دهید تا مقادیر، **Q-value** ها و شبیه‌سازی را مشاهده کنید. باید متوجه شوید که مقدار حالت شروع (**V(start)**) که می‌توانید آن را از رابط کاربری گرافیکی بخوانید) و میانگین پاداش تجربی حاصل (چاپ شده پس از اتمام **10** دور اجرا) کاملاً نزدیک هستند.

```
python gridworld.py -a value -i 100 -k 10
```

راهنمایی: در **BookGrid** پیش‌فرض، اجرای تکرار مقدار برای **5** تکرار باید این خروجی را به شما بدهد:

```
python gridworld.py -a value -i 5
```



گزارش: توابع پیاده‌سازی شده در این بخش و خروجی تست‌ها را به همراه اسکرین‌شات کامل توضیح دهید.

[5] Policy

۲) تجزیه و تحلیل عبور از پل[6] (۱ امتیاز)

BridgeGrid یک نقشه در **grid world** است که یک حالت پایانه با پاداش کم و یک حالت پایانه با پاداش بالا دارد که توسط یک «پل» باریک از هم جدا شده‌اند، که در دو طرف آن حالت‌هایی با پاداش منفی بالا وجود دارد. عامل نزدیک به حالت پاداش کم شروع می‌کند. با تخفیف پیش‌فرض **0.9** و نویز پیش‌فرض **0.2**، سیاست بهینه[7] از پل عبور نمی‌کند. تنها یکی از پارامترهای تخفیف یا نویز را تغییر دهید تا سیاست بهینه باعث شود عامل برای عبور از پل تلاش کند. پاسخ خود را در قسمت **question2()** در فایل **analysis.py** قرار دهید. (نویز به این اشاره دارد که هر چند وقت یک عامل هنگام انجام یک عمل به یک حالت ناخواسته ختم می شود.) خروجی مقدارهای پیش فرض به صورت زیر است:

```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --
noise 0.2
```

بررسی خواهد شد که شما فقط یکی از پارامترهای داده شده را تغییر داده‌اید و با این تغییر، یک عامل تکرار ارزش که به درستی پیاده‌سازی شده است، باید از پل عبور کند. برای بررسی پاسخ خود، **autograder** را با دستور زیر اجرا کنید:

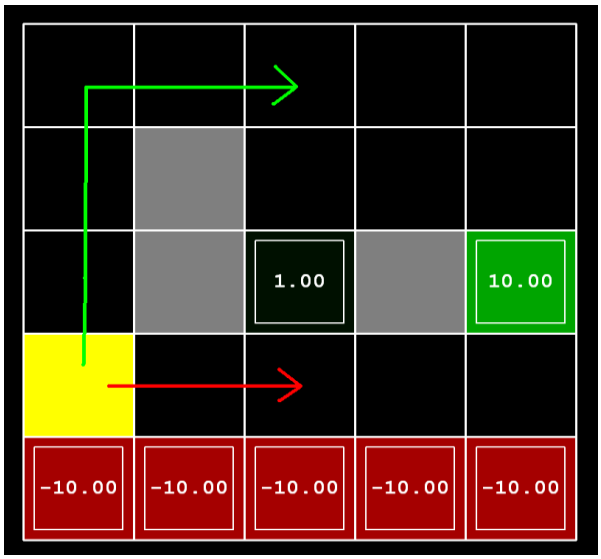
```
python autograder.py -q q2
```

گزارش: دلیل انتخاب مقادیر برای پارامترهای این بخش را توضیح دهید.

[6] Bridge Crossing Analysis
[7] optimal policy

۳) سیاست‌ها (۵ امتیاز)

نقشه ی **DiscountGrid** را که در زیر نشان داده شده است در نظر بگیرید. این **grid** دارای دو حالت پایانی با پاداش مثبت است. یک خروجی نزدیک با پاداش **+1** و یک خروجی دور با پاداش **+10**. ردیف پایین **grid** شامل حالات پایانی با پاداش منفی است (نشان داده شده با رنگ قرمز) و به اصطلاح به این قسمت صخره می‌گوییم. هر حالت در صخره دارای پاداش **-10** است. حالت شروع مربع زرد است. ما بین دو نوع مسیر تمایز قائل می شویم: (1) مسیرهایی که "خطر صخره را به جان می‌خرند" و نزدیک ردیف پایین شبکه حرکت می‌کنند؛ این مسیرها کوتاه‌تر هستند، اما ریسک دریافت پاداش منفی زیادی دارند و با فلش قرمز در شکل زیر نشان داده شده‌اند. (2) مسیرهایی که "از صخره اجتناب می‌کنند" و در امتداد لبه بالایی grid حرکت می‌کنند؛ این مسیرها طولانی‌تر هستند، اما احتمال کم‌تری دارد که بازدهی منفی بزرگی را متحمل شوند. این مسیرها با فلش سبز رنگ در شکل زیر نشان داده شده‌اند.



در این سوال، تنظیمات پارامترهای تخفیف، نویز و پاداش زندگی[8] را برای این **MDP** انتخاب می‌کنید تا چند نوع مختلف از سیاست‌های بهینه را بدست آورید. تنظیم مقدار پارامترها برای هر بخش باید این ویژگی را داشته باشد که اگر عامل شما سیاست بهینه خود را بدون ایجاد نویز دنبال کرد، رفتار داده شده را نشان دهد. اگر رفتار خاصی با هیچ تنظیمی از پارامترها به دست نیامد، با بازگرداندن رشته "NOT POSSIBLE"، نشان دهید که این سیاست غیرممکن است.

[8] living reward

در اینجا انواع سیاست‌های بهینه وجود دارد که باید سعی کنید آن‌ها را تولید کنید:

- خروجی نزدیک را ترجیح دهید (**+1**)، ریسک صخره را بپذیرید (**-10**)
- خروجی نزدیک را ترجیح دهید (**+1**)، اما از صخره اجتناب کنید (**-10**)
- خروجی دور را ترجیح دهید (**+10**)، ریسک صخره را بپذیرید (**-10**)
- خروجی دور را ترجیح دهید (**+10**)، اما از صخره اجتناب کنید (**-10**)
- از هر دو خروجی و صخره اجتناب کنید (بنابراین اجرای آن هرگز نباید پایان یابد)

برای بررسی پاسخ‌های خود، **autograder** را با دستور زیر اجرا کنید:

```
python autograder.py -q q3
```

سوال‌های **question3a()** تا **question3e()** باید هر کدام یک سه تایی از (تخفیف، نویز، پاداش زندگی) را در **analysis.py** برگردانند.

توجه: می‌توانید سیاست‌های خود را در رابط کاربری گرافیکی بررسی کنید. به عنوان مثال، با استفاده از پاسخ صحیح به **a3**، فلش در **(0, 1)** باید به سمت شرق، فلش در **(1, 1)** نیز باید به سمت شرق، و فلش در **(2, 1)** باید به سمت شمال باشد.

توجه: در برخی از سیستم‌ها ممکن است فلش‌ها برایتان نشان داده نشوند. در این حالت، دکمه‌ای را روی صفحه کلید فشار دهید تا صفحه نمایش **qValue** نمایش داده شود، و به طور ذهنی خط مشی را با گرفتن **arg max**، برای هر حالت **qValue** محاسبه کنید.

در نهایت نیز برای نمره‌دهی، ما بررسی خواهیم کرد که در هر مورد سیاست مورد نظر برگردانده شود.

گزارش: دلیل انتخاب مقادیر برای پارامترهای این بخش را توضیح دهید.

۴) تکرار ارزش ناهمزمان[9] (۱ امتیاز)

یک عامل تکرار ارزش در **AsynchronousValueIterationAgent** بنویسید که تا حدی برای شما در **valueIterationAgents.py** مشخص شده است. عامل تکرار ارزش شما یک برنامه‌ریز آفلاین است و نه یک عامل یادگیری تقویتی. **training option** مربوط به تعداد تکرارهای تکرار ارزش است که باید در مرحله برنامه‌ریزی اولیه اجرا شود **(option -i)**.

AsynchronousValueIterationAgent یک **MDP** را در **constructor** می‌گیرد و تکرار ارزش چرخه‌ای[10] (شرح شده در پاراگراف بعدی) را برای تعداد مشخصی از تکرارها تا قبل از اتمام اجرای **constructor**، اجرا می‌کند. توجه داشته باشید که تمام این کد تکرار ارزش باید در داخل **constructor** (متد **__init__**) قرار گیرد.

دلیل اینکه این کلاس **AsynchronousValueIterationAgent** نامیده می‌شود این است که ما در هر تکرار[11] ، برخلاف انجام یک به روزرسانی به صورت **batch**، فقط یک حالت را بروزرسانی می‌کنیم. در اینجا نحوه عملکرد تکرار ارزش چرخه‌ای آمده است. در اولین تکرار، فقط مقدار حالت اول را در لیست حالت‌ها به‌روز کنید. در تکرار دوم، فقط مقدار دوم را به‌روز کنید. این کار را ادامه دهید تا زمانی که مقدار هر حالت را یک بار به روز کرده باشید، سپس از حالت اول برای تکرار بعدی شروع کنید. اگر حالت انتخاب شده برای بروزرسانی ترمینال باشد، در آن تکرار هیچ اتفاقی نمی‌افتد. شما می‌توانید این را به عنوان **indexing** در متغیر حالت‌های تعریف شده در اسکلت کد پیاده‌سازی کنید.

به عنوان یادآوری، معادله بروزرسانی وضعیت تکرار ارزش در اینجا آمده است:

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

همانطور که در کلاس بحث شد، تکرار ارزش یک معادله **fixed-point** را تکرار می‌کند. همچنین می‌توان مقادیر حالت را به روش‌های مختلف به‌روزرسانی کرد، مثلاً به ترتیب تصادفی (به عنوان مثال، یک حالت را به‌طور تصادفی انتخاب کنید، مقدار آن را بروزرسانی کنید و تکرار کنید) یا به سبک دسته‌ای (مانند سوال اول). در این سوال، تکنیک دیگری را بررسی خواهیم کرد.

---

[9] Asynchronous Value Iteration
[10] cyclic value iteration
[11] iteration

AsynchronousValueIterationAgent از **ValueIterationAgent** در سوال اول ارث‌بری می‌کند، بنابراین تنها متدی که نیاز دارید پیاده‌سازی کنید **runValueIteration** است. از آنجایی که **constructor** سوپرکلاس **runValueIteration** را فراخوانی می‌کند، **override** کردن آن برای تغییر رفتار عامل به نحو خواسته شده، کافی است.

توجه: مطمئن شوید حالتی که یک **state** هیچ **action** ممکنی در **MDP** ندارد را در پیاده‌سازی خود در نظر بگیرید. (به معنای این موضوع برای پاداش‌های آینده فکر کنید).

برای آزمایش پیاده سازی خود، **autograder** را اجرا کنید. اجرای آن باید کمتر از یک ثانیه طول بکشد. اگر خیلی بیشتر طول بکشد، ممکن است بعداً در پروژه با مشکلاتی مواجه شوید، بنابراین همین حالا اجرای خود را کارآمدتر کنید.

```
python autograder.py -q q4
```

دستور زیر **AsynchronousValueIterationAgent** شما را لود می‌کند، که یک سیاست را محاسبه و **10** بار اجرا می‌کند. دکمه‌ای را روی صفحه کلید فشار دهید تا مقادیر، **Q-value** ها و شبیه‌سازی را مشاهده کنید. باید متوجه شوید که مقدار حالت شروع (**V(start)** که می‌توانید آن را از رابط کاربری گرافیکی بخوانید) و میانگین پاداش تجربی حاصل (چاپ شده پس از اتمام **10** دور اجرا) کاملاً نزدیک هستند.

```
python gridworld.py -a asynchvalue -i 1000 -k 10
```

گزارش: توابع پیاده‌سازی شده در این بخش و خروجی تست‌ها را به همراه اسکرین‌شات کامل توضیح دهید.

۵) تکرار ارزش اولویت بندی شده[12] (۳ امتیاز)

این الگوریتم تلاش می‌کند بروزرسانی‌های مقادیر حالت را به سمتی متمرکز کند که احتمالا سیاست‌ها را تغییر دهد. مراحل الگوریتم به صورت زیر است.

- ابتدا باید برای هر حالت، همه پسین‌ها[13] مشخص شود.
- یک صف خالی برای نگه‌داری اولویت‌ها تعریف کنید.
- برای هر حالت غیرپایانی **s**:
  - قدر مطلق تفاضل بین مقدار فعلی حالت **s** (که در **self.values** نگه‌داری می‌شود) و بیشترین مقدار **Q** ممکن از حالت **s** که با استفاده از اقدام‌های ممکن قابل تعریف است را محاسبه کنید. این مقدار را **diff** بنامید.
  - حالت **s** را با اولویت **-diff** به صف اولویت‌ها اضافه کنید. (دلیل اینکه از اولویت منفی استفاده می‌کنیم این است که صف اولویت‌ها به صورت **min heap** است و اولویت با مقدار عددی کمتر به معنی اولویت بالاتر است و ما می‌خواهیم حالتی که بیش‌ترین خطا را دارد اولویت بیشتری داشته باشد و زودتر بروزرسانی شود)
- به ازای تعداد تکرارهای مشخص شده در **self.iterations**:
  - اگر صف اولویت‌ها خالی می‌باشد کار پایان یافته است.
  - در غیر این صورت حالت s را از صف اولویت‌ها بردارید.
  - در صورتی که **s** حالت پایانی نبود، مقدار حالت **s** را (در **self.values**) بروزرسانی کنید.
  - به ازای هر پسین **p** از حالت **s**:
    - قدر مطلق تفاضل بین مقدار فعلی حالت **p** (که در **self.values** نگهداری می‌شود) و بیشترین مقدار **Q** ممکن از حالت **p** که با استفاده از اقدام‌های ممکن قابل تعریف است را محاسبه کنید. این مقدار را **diff** بنامید.

[12] Prioritized Sweeping Value Iteration
[13] Predecessors

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

توجه داشته باشید که هنگام محاسبه پسین‌های یک حالت، برای جلوگیری از به‌وجود آمدن حالت‌های تکراری، آن‌ها را در مجموعه[14] نگهداری کنید.

در این سوال شما باید **PrioritizedSweepingValueIterationAgent** را که تا حدی در **valueIterationAgents.py** تعریف شده است، پیاده‌سازی کنید. توجه داشته باشید که این کلاس از **AsynchronousValueIterationAgent** مشتق شده است، بنابراین تنها قسمتی که باید تغییر کند متد **runValueIteration** است.

برای صحت‌سنجی پیاده‌سازی خود، **autograder** را اجرا کنید. این اجرا باید حدود **1** ثانیه طول بکشد. اگر بیشتر طول بکشد ممکن است در ادامه با مشکلاتی مواجه شوید، بنابراین در همین مرحله پیاده‌سازی خود را بهینه کنید.

برای اینکه **autograder** برای این سوال کار کند، باید تکرار روی حالت‌ها را به ترتیبی که در **self.mdp.getStates()** مشخص شده است انجام دهید.

```
python autograder.py -q q5
```

با استفاده از دستور زیر می‌توانید **PrioritizedSweepingValueIterationAgent** را در **Gridworld** اجرا کنید.

```
python gridworld.py -a priosweepvalue -i 1000
```

گزارش: توابع پیاده‌سازی شده در این بخش و خروجی تست‌ها را به همراه اسکرین‌شات کامل توضیح دهید.

[14] Set

۶) یادگیری **Q** (۴ امتیاز)

عامل تکرار کننده مقادیر، قبل از تعامل با محیط واقعی، مدل **MDP** خود را برای رسیدن به یک مجموعه سیاست کامل در نظر می‌گیرد و در واقع از تجربه کردن یاد نمی‌گیرد. پس از آن، هنگامی که در محیط واقعی قرار بگیرد، به سادگی از سیاست‌هایی که از قبل محاسبه شده پیروی می‌کند. این تمایز ممکن است در یک محیط شبیه‌سازی شده مانند **Gridword** قابل ملاحظه نباشد ولی در دنیای واقعی، جایی که **MDP** واقعی در دسترس نیست، بسیار مهم است. در این سوال شما باید یک عامل یادگیری **Q** بنویسید، که در هنگام ایجاد، تلاش زیادی برای یادگیری نمی‌کند ولی با آزمون و خطا و از تعامل با محیط و از طریق متد **update(state, action, nextState, reward)** یاد می‌گیرد.

قسمتی از یک یادگیرنده **Q** در **QLearningAgent** در **qlearningAgents.py** تعریف شده است.

در این سوال باید متدهای **update** و **computeValueFromQValues** و **getQValue** و **computeActionFromQValues** را پیاده‌سازی کنید.

برای رفتار بهتر در متد **computeActionFromQValues** باید پیوندها را به صورت تصادفی قطع کنید. برای این کار می‌توانید از تابع **random.choice()** استفاده کنید. اقداماتی که عامل قبلاً ندیده است، نیز یک مقدار **Q** دارند (مثلا می‌توان مقدار اولیه صفر به آن‌ها اختصاص داد). در حالتی که همه اقداماتی که عامل قبلاً دیده است مقدار **Q** منفی داشته باشند، یک قبلا مشاهده نشده می‌تواند بهترین اقدام باشد. دسترسی به مقادیر **Q** باید فقط از طریق فراخوانی تابع **getQValue** امکان‌پذیر باشد در غیر این صورت در سوال دهم مشکل خواهید داشت.

با دستور زیر می‌توانید یادگیری یادگیرنده **Q** خود را تحت کنترل دستی مشاهده کنید که **-k** تعداد قسمت‌هایی که عامل برای یادگیری صرف می‌کند را مشخص می‌کند:

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

برای ارزیابی پیاده سازی خود می‌توانید **autograder** را اجرا کنید:

```
python autograder.py -q q6
```

گزارش: توابع پیاده‌سازی شده در این بخش و خروجی تست‌ها را به همراه اسکرین‌شات کامل توضیح دهید.

۷) **epsilon** حریصانه (۲ امتیاز)

عامل یادگیری **Q** خود را با اضافه کردن انتخاب اقدام **epsilon-greedy** در **getAction** تکمیل کنید. این اقدام به این معناست که عامل در کسری از زمان اقدامات تصادفی انتخاب می‌کند و در غیر این صورت از بهترین مقادیر **Q** فعلی خود پیروی می‌کند. انتخاب یک اقدام تصادفی ممکن است منجر به انتخاب بهترین اقدام شود یعنی می‌توانید هر اقدام تصادفی مجازی را انتخاب کنید.

پس از پیاده سازی متد **getAction** با استفاده از دستور زیر رفتار عامل را در **gridworld** مشاهده کنید (از **epsilon** = **0.3** استفاده کنید)

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

برای ارزیابی پیاده سازی خود **autograder** را اجرا کنید:

```
python autograder.py -q q7
```

با پایان این قسمت باید بتوانید یک ربات **crawler** یادگیری **Q** را بدون کد اضافی اجرا کنید. دستور زیر با استفاده از یادگیرنده **Q** طراحی شده شما، ربات خزنده را از کلاس فراخوانی می‌کند.

```
python crawler.py
```

گزارش: توابع پیاده‌سازی شده در این بخش و خروجی تست‌ها را به همراه اسکرین‌شات کامل توضیح دهید.

۸) بررسی دوباره عبور از پل (۱ امتیاز)

ابتدا، یک **Q-learner** کاملا تصادفی را با ضریب یادگیری پیش‌فرض بر روی **BridgeGrid** بدون نویز، با **50** اپیزود آموزش دهید و بررسی کنید که آیا سیاست بهینه در این حالت یافت می‌شود یا خیر.

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

حال، همین کار را با اپسیلون **0** دوباره تکرار کنید. آیا مقدار اپسیلون و ضریب یادگیری‌ای وجود دارد که با استفاده از آن‌ها، سیاست بهینه با احتمال خیلی بالا (بیشتر **99** درصد) بعد از **50** بار تکرار یاد گرفته شود؟

تابع **question8()** در **analysis.py** یا یک تاپل دوتایی **(epsilon, learning rate)** برمی‌گرداند و یا در صورتی که جوابی پیدا نکند، رشته **'NOT POSSIBLE'** برگردانده می‌شود. اپسیلون با **-e** و ضریب یادگیری، با **-l** کنترل می‌شود.

توجه: پاسخ شما نباید به مکانیزم تعیین‌کننده‌ای که برای انتخاب اکشن‌ها استفاده شد، وابسته باشد؛ یعنی پاسخ شما باید حتی در حالتی که مثلا ما جهان **Bridge Grid** را **90** درجه هم چرخانده‌ایم صحیح باشد.

برای ارزیابی پاسخ خود، با استفاده از دستور زیر **autograder** را اجرا کنید:

```
python autograder.py -q q8
```

گزارش: دلیل انتخاب مقادیر برای پارامترهای این بخش را توضیح دهید.

۹) پک من و **Q-Learning** (۱ امتیاز)

حال زمانی بازی پک‌من است! پک‌من بازی‌ها را در دو فاز انجام می‌دهد. در فاز اول، آموزش[15]، پک‌من در مورد امتیاز موقعیت‌ها و اکشن‌ها، آموزش میبیند. از آن جایی که حتی برای **grid** های کوچک هم زمان زیادی طول می‌کشد که **Q-value** های دقیق یاد گرفته شوند، بازی‌های آموزش پک‌من به صورت پیش‌فرض در **quiet mode** اجرا می‌شوند، بدون هیچگونه نمایش گرافیکی یا نماشی در کنسول. زمانی که آموزش پک‌من به انجام رسید، وارد فاز آزمون می‌شود. در زمان آزمون، **self.epsilon** و **self.alpha** در پک‌من به مقدار **0.0**، به جهت توقف **Q-learning** و غیرفعال کردن جستجو، ست خواهند شد. این امر به پک‌من امکان بهره بردن از سیاست‌هایی که یاد گرفته را می‌دهد. بازی‌های آزمون به صورت پیش‌فرض، به صورت **GUI** نمایش داده می‌شوند. بدون هیچ‌گونه تغییری در کد، شما باید بتوانید **Q-**

[15] Training

Learning را برای پک‌من، برای **grid** های بسیار کوچک اجرا کنید. در زیر دستور لازم برای این امر را مشاهده می‌فرمایید:

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

توجه داشته باشید که **PacmanQAgent** از قبل برای شما تعریف شده است. (**QlearningAgent** که از پیش آن را نوشته‌اید). تنها تفاوت این دو در این است که **PacmanQAgent** پارامترهای یادگیری پیش‌فرض را دارد که برای مساله پک‌من، مفیدتر است (**epsilon=0.05, alpha=0.2, gamma=0.8**).

شما نمره کامل این بخش را در صورتی که دستور بالا بدون خطا اجرا شود و عامل شما حداقل در **80** درصد موارد، برنده شود، دریافت می‌کنید. **autograder** بازی‌های آزمون را **100** بار بعد از اجرای **2000** بار بازی‌های آموزش، اجرا خواهد کرد.

راهنمایی: اگر **QLearningAgent** شما برای **gridworld.py** و **crawler.py** کار می‌کند اما به نظر نمی‌رسد که سیاست خوبی را برای پک‌من در **smallGrid** یاد گرفته است، احتمالا به این دلیل است که توابع **getAction** و یا **computeActionFromQValues** شما در بعضی از موارد، اکشن‌های دیده نشده (**unseen**) را به صورت مناسبی در نظر نگرفته‌اند. در واقع، از آن جایی که اکشن‌های دیده نشده بر اساس تعریف، مقدار **Q-value** برابر با **0** دارند؛ اگر تمام اکشن‌های دیده شده تا به اینجای کار، مقدار **Q-value** منفی داشته باشند، یک اکشن دیده نشده می‌تواند بهینه باشد. حتما هم حواستان به تابع **argmax** از **util.counter** باشد!

نکته: برای ارزیابی پاسخ خود، دستور زیر را اجرا کنید:

```
python autograder.py -q q9
```

نکته: اگر قصد تست کردن پارامترهای یادگیری را دارید، میتوانید از **-a** استفاده کنید.

برای مثال، دستور -a epsilon=0.1,alpha=0.3,gamma=0.7 . این مقادیر در متغیرهای **self.epsilon**، **self.gamma** و **self.alpha** در داخل خود عامل قابل دسترسی خواهند بود.

نکته: هنگامی که **2010** تا بازی انجام میشود، **2000** تا بازی اول به دلیل گزینه **-x 2000** نمایش داده نمی‌شوند، که درواقع نشان می‌دهد که **2000** بازی اول، برای آموزش هستند (خروجی نخواهند داشت). بنابراین، شما فقط بازی پک‌من را در **10** بازی آخر خواهید دید. تعداد بازی‌های آموزشی نیز تحت گزینه **numTraining** پاس داده می‌شود.

نکته: اگر می‌خواهید ببینید که در **10** بازی آموزشی چه اتفاقی می افتد، از دستور زیر استفاده کنید:

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10
```

در طول آموزش، خروجی هر **100** بازی را با آماری در مورد نحوه عملکرد پک‌من مشاهده خواهید کرد. اپسیلون در طول آموزش مثبت است، بنابراین پک‌من حتی پس از آموختن یک سیاست خوب هم ضعیف بازی می‌کند: این به این دلیل است که او گهگاه یک حرکت اکتشافی تصادفی به سمت یک روح انجام می‌دهد. به عنوان یک معیار، بین **1000** تا **1400** بازی طول می‌کشد تا پاداش پک‌من برای یک دسته **100** تایی بازی مثبت شود، که نشان‌دهنده این است که عامل شروع به دریافت تعداد برد های بیشتری از باخت کرده‌است. در پایان آموزش نیز پاداش باید مثبت بماند و نسبتاً بالا باشد (بین **100** تا **350**).

همچنین به این که در اینجا دقیقا چه اتفاقی در حال رخ‌دادن است هم توجه داشته باشید: حالت **MDP** پیکربندی دقیق بردی است که پک‌من با آن روبرو است، با انتقال‌های پیچیده که اکنون کل تغییرات را در آن حالت توصیف می‌کند. پیکربندی‌های میانی بازی که پک‌من در آن‌ها حرکت کرده است، اما ارواح پاسخی نداده‌اند، حالت‌های **MDP** نیستند، بلکه در انتقال‌ها قرار گرفته‌اند.

زمانی که آموزش پک‌من به پایان برسد، باید حداقل در **90** درصد مواقع، بازی‌های تست را ببرد، چرا که در این مرحله، عامل از سیاست آموخته شده استفاده می‌کند. با این حال، متوجه خواهید شد که آموزش همان عامل بر روی **mediumGrid** به ظاهر ساده، به خوبی کار نمی‌کند. در پیاده‌سازی ما، میانگین پاداش‌ها در زمان آموزش پک‌من منفی باقی می‌ماند. در زمان آزمون، پک‌من بد بازی کرده و احتمالا تمام بازی‌های آزمون را خواهد باخت. آموزش نیز با وجود بی‌اثر بودن، زمان زیادی را می‌طلبد. پک‌من در طرح‌بندی‌های بزرگ‌تر نیز برنده نمی‌شود، زیرا هر پیکربندی برد یک حالت جداگانه با مقادیر **Q-value**

جداگانه است. او هیچ راهی برای تعمیم این مساله ندارد که برخورد با یک روح برای همه موقعیت‌ها بد است. بدیهی است که این رویکرد مقیاس‌پذیر نخواهد بود.

گزارش: تغییرات و فعالیت‌هایی که در این بخش انجام داده‌اید را کامل توضیح دهید و برای آنان دلایل کافی ارائه دهید.

قسمت اختیاری و امتیازی: ۱۰) یادگیری تقریبی Q [16]

یک عامل **Q-learning** تقریبی را پیاده‌سازی کنید که وزن ویژگی[17]های حالت‌ها را یاد می‌گیرد (بسیاری از حالت‌ها ممکن است ویژگی‌های مشترکی داشته باشند). پیاده‌سازی خود را در کلاس **ApproximateQAgent** واقع در **qlearningAgents.py** را پیاده‌سازی کنید. این کلاس در واقع زیرکلاسی از **PacmanQAgent** می‌باشد.

نکته: **Q-learning** تقریبی وجود یک تابع ویژگی را فرض می‌کند $f(s,a)$، که ورودی این تابع، حالت و اکشن است. این توابع ویژگی، یک بردار از مقادیر ویژگی‌ها به صورت:

$$[f1(s,a), \dots, fi(s,a), \dots, fn(s,a)]$$

تشکیل می‌دهند. این توابع ویژگی‌ها، در فایل **featureExtractors.py** قرار گرفته‌اند. بردارهای ویژگی‌ها، از جنس آبجکت‌های **util.Counter** هستند (مانند دیکشنری) که دوتایی‌های متشکل از ویژگی‌ها و مقادیر آن‌ها را دارا هستند؛ توجه داشته باشید که تمام ویژگی‌های حذف شده، مقدار **0** را دارا هستند.

Q-function تقریبی، فرم زیر را دارا است:

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

در این رابطه، هر یک از وزن‌های w_i، با یکی از وزن‌ها $f_i(s,a)$ مرتبط است. در کد، شما باید بردار وزن را به عنوان یک دیکشنری پیاده‌سازی کنید که ویژگی‌ها را (که **feature extractor** ها برمی‌گردانند)

[16] Approximate Q-learning
[17] Features

به مقدار وزن‌ها، نگاشت کند. همچنین شما باید بردارهای وزن خود را به همان صورتی که **Q-value** ها را آپدیت کردید، آپدیت نمایید:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$

$$difference = (r + \gamma \max_a Q(s',a')) - Q(s,a)$$

توجه داشته باشید که $difference$ در این جا، در واقع مشابه در **Q-learning** معمول است و **r** هم پاداش آزمایش شده می‌باشد.[18]

به صورت پیش‌فرض، **ApproximateQAgent** از **IdentityExtractor** استفاده می‌کند که هر ویژگی را به هر دوتایی **(state,action)** اختصاص می‌دهد. عامل **Q-learning** تقریبی شما باید مشابه **PacmanQAgent** کار کند. می‌توانید این موضوع را با استفاده از دستور زیر، تست کنید:

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

نکته مهم: **ApproximateQAgent** زیرکلاسی از **QlearningAgent** است؛ بنابراین بسیاری از توابع آن اعم از **getAction** مشترک است. توجه داشته باشید که توابع شما در **QlearningAgent** به جای دسترسی مستقیم به **Q-value** ها، تابع **getQValue** را صدا کنند تا زمانی که شما در کد **approximate agent** خود **getQValue** را **override** می‌کنید، مقادیر جدید **approximate Q-value** ها برای محاسبه اکشن‌ها استفاده شوند.

زمانی که مطمئن شدید که **approximate learner** شما به درستی با **identity feature** کار میکند، عامل **Q-learning** تقریبی خود را با **feature extractor** اختصاصی ما اجرا کنید، که می‌تواند برنده شدن به راحتی را یاد بگیرد:

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

[18] Note that the difference term is the same as in normal Q-learning, and r is the experienced reward.

حتی طرح بندی‌های بزرگ‌تر نیز برای **ApproximateQAgent** شما نیز نباید مشکلی ایجاد کند. (البته دقت داشته باشید که ممکن است که آموزش مدل چند دقیقه‌ای طول بکشد.)

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

اگر به اروری در حین اجرا برخورد نکنید، عامل **approximate Q-learning** شما باید تقریبا در تمامی بازی‌ها با استفاده از همین ویژگی‌های ساده برنده شود، حتی با تنها 50 بازی آموزشی.

ما عامل **approximate Q-learning** شما را اجرا و بررسی خواهیم کرد که وقتی هر کدام با مجموعه‌ای از مثال‌ها ارائه می‌شوند، همان **Q-value** و وزن ویژگی‌های ارائه شده در پیاده‌سازی مرجع ما را یاد می‌گیرد.

برای ارزیابی پیاده سازی خود، **autograder** را اجرا کنید:

```
python autograder.py -q q10
```

گزارش: توابع پیاده‌سازی شده در این بخش و خروجی تست‌ها را به همراه اسکرین‌شات کامل توضیح دهید.

تبریک! شما یک عامل پک من یادگیرنده دارید!

توضیحات تکمیلی

- پاسخ به تمرین ها باید به صورت فردی انجام شود. در صورت استفاده مستقیم از کدهای موجود در اینترنت و مشاهده تقلب، برای همه‌ی افراد نمره صفر لحاظ خواهد شد.
- برای این پروژه به صورت رندوم از تعدادی از دانشجویان تحویل آنلاین گرفته خواهد شد و نمره‌دهی مابقی دانشجویان بر اساس گزارش پروژه، نمره **autograder** و پیاده‌سازی انجام شده است، لذا ضروری است که همه‌ی دانشجویان گزارشی برای پروژه تهیه نموده و در آن به طور کامل نحوه‌ی پیاده‌سازی‌های انجام شده را شرح دهند.
- فایل‌های **valueIterationAgents.py** و **qlearningAgents.py** و **analysis.py** به همراه گزارش را در قالب یک فایل فشرده با فرمت AI_P3_StdNum.zip در سامانه کورسز آپلود کنید.
- در صورت هرگونه سوال یا ابهام از طریق ایمیل ai.aut.spring1401@gmail.com با تدریسیاران در تماس باشید، همچنین خواهشمند است در متن ایمیل به شماره دانشجویی خود اشاره کنید.
- همچنین می‌توانید از طریق تلگرام نیز با آیدی‌های زیر در تماس باشید و سوالاتتان را مطرح کنید:
    - @koroshroohi
    - @elhamrazi
- ددلاین این پروژه ۶ خرداد ۱۴۰۱ است، بنابراین بهتر است انجام پروژه را به روز های پایانی موکول نکنید.