Project 3 : Reinforcement learning

소개:

- P3 프로젝트에서는 **Value iteration**과 **Q-learning**을 구현합니다. (평가 대상은 총 6문제 입니다.)

- 이전 프로젝트에서와 마찬가지로, 자동 채점이 포함되어 있으며, 다음 명령어를 통해 실행시킬 수 있습니다.

  python autograder.py

- 다음과 같이 q2와 같은 특정 질문에 대해서도 채점이 가능합니다.

  python autograder.py -q q2

- 아래 명령어를 통해 특정 테스트를 실행할 수도 있습니다.

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

- 관련한 모든 코드는 zip archive에 포함되어 있습니다.

  https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/reinforcement/v1/002/reinforcement.zip

- P3 프로젝트는 수강생들을 위해 다음 주소의 사이트를 번역 하였습니다. (필히 참고)

  https://edge.edx.org/courses/course-v1:BerkeleyX+CS188+2018_SP/courseware/817468c509ef4b64bcb74cd38766ac44/41c847d3794141a7acb4f6b1e5904215/?activate_block_id=block-v1%3ABerkeleyX%2BCS188%2B2018_SP%2Btype%40sequential%2Bblock%4041c847d3794141a7ac

b4f6b1e5904215

파일 설명

- **수정하거나 실행하는 파일들**

- **valueIterationAgents.py** : A value iteration agent for solving known MDPs.
- **qlearningAgents.py** : Q-learning agents for Gridworld, Crawler and Pacman.
- **analysis.py** : A file to put your answers to questions given in the project.

- **수정하면 안되지만, 꼭 봐야하는 파일들**

- **mdp.py** : Defines methods on general MDPs.
- **learningAgents.py** : Defines the base classes `ValueEstimationAgent` and `QLearningAgent`, which your agents will extend.
- **util.py** : Utilities, including `util.Counter`, which is particularly useful for Q-learners.
- **Gridworld.py** : The Gridworld implementation.
- **featureExtractors.py** : Classes for extracting features on (state,action) pairs. Used for the approximate Q-learning agent (in qlearningAgents.py).

- **무시해도 괜찮은 파일들**

- environment.py : Abstract class for general reinforcement learning environments. Used by gridworld.py.
- graphicsGridworldDisplay.py : Gridworld graphical display.
- graphicsUtils.py : Graphics utilities.
- textGridworldDisplay.py : Plug-in for the Gridworld text interface.
- crawler.py : The crawler code and test harness. You will run this but not edit it.
- graphicsCrawlerDisplay.py : GUI for the crawler robot.
- autograder.py : Project autograder
- testParser.py : Parses autograder test and solution files
- testClasses.py : General autograding test classes
- test_cases/ : Directory containing the test cases for each question.
- reinforcementTestClasses.py : Project 3 specific autograding test classes

P3 welcome

MDPs

시작을 위해 화살표 키를 사용하는 수동 제어 모드에서 Gridworld를 실행하세요.

python gridworld.py -m

두 출구가 있는 것을 볼 수 있습니다. 파란 점이 Agent입니다. Up을 누를 때 Agent는 실제로 북쪽으로 80%만 이동합니다.

시뮬레이션은 여러 부분을 제어할 수 있습니다. 다음을 실행하여 전체 옵션 목록을 볼 수 있습니다.

Python gridworld.py -h

기본 agent가 무작위로 이동하는 명령어 입니다.

Python gridworld.py -g MazeGrid

랜덤 agent가 종료시까지 그리드에서 이동하는 것을 볼 수 있습니다. AI agent에게 가장 좋은 예는 아니라는 것을 알 수 있습니다.

참고 : Gridworld MDP는 먼저 GUI에 표시된 사전 종료 상태(Reward 1, Reward -1)를 통해서 그 상태에 도달하지 않는다면 종료되지 않게끔 설정되어 있습니다. 에피소드를 수동으로 실행하는 경우 the discount rate(변경하려면 -d 를 사용하세요. 기본값은 0.9입니다.)로 인해 리턴 값은 기대했던 값보다 낮을 수 있습니다.

그래픽 출력과 함께 제공되는 콘솔 출력을 확인하세요. Agent가 경험하는 각 transition의 정보에 관해 알려줍니다.(모든 텍스트에 대해 -t, 끄려면 -q)

Pacman에서와 같이 위치는 (x,y) 좌표로 표현되며 배열은 [x][y]로 색인화되며 'north'는 y가 증가하는 방향 등으로 표시됩니다. 기본적으로 대부분의 transitions에는 보상 0을 받지만, 보상 옵션(-r)을 사용하여 이를 변경할 수 있습니다.

Q1 – Value Iteration

ValueIterationAgents.py에서 명시된 ValueIterationAgent에 **value iteration agent**을 작성하세요. Value iteration agent는 오프라인 플래너이기 때문에 관련 학습 옵션은 처음 계획 단계에서 실행되어야 하는 가치 반복 횟수(옵션 -i)입니다. ValueIterationAgent는 생성시 MDP를 사용하고, runValueIteration을 호출합니다. runValueIteration은 생성자가 반환하기 전에 self.iterations 반복에 대한 value iteration을 실행합니다.

Value iteraion은 최적 값 V_k에 대한 k-step estimates를 계산합니다. Value iteration을 실행하는 것 외에도 V_k를 사용하여 ValueIterationAgent에 대해 다음 함수들을 구현하세요.

- **computeActionFromValues(state)**는 self.values에 의해 주어진 value function에 따라 최적 행동을 계산합니다.
- **computerQValueFromValues(state, action)**은 self.values에 의해 주어진 value function에 의해 주어진 (state, action)쌍의 Q 값을 반환합니다.

이 결과는 GUI에 모두 표시됩니다. 값은 정사각형의 숫자, Q값은 정사각형의 숫자, 정책은 각 사각형에서의 화살표로 표시됩니다.

중요 : 각 벡터 V_k는 하나의 단일 가중치 벡터가 제자리에서 업데이트되는 "온라인"버전이 아닌 고정 벡터V_{k-1}(강의에서와 같이)에서 계산되는 value iteration의 "배치"버전을 사용하세요. 이것은 상태 값이 그 후속 상태 값에 따라 반복 k에서 업데이트 될 때, 값 업데이트 계산에 사용된 후속 상태 값은 반복 k-1의 값이어야 한다는 의미입니다. 차이점은 4.1 챕터의 6단락에 있는 Sutton & Barto(http://incompleteideas.net/sutton/book/ebook/node41.html)에서 논의됩니다.

참고 : 다음 k 보상을 반영하는 깊이 k의 값(다음 k 보상들을 반영하는)에서 나온 정책은 실제로 다음 k+1 보상들을 반영합니다.(즉, π_{k+1}을 반환) 마찬가지로 Q값은 Value 보다 하나 더 진행된 보상을 반영합니다.(즉, , Q_{k+1}을 반환)

합성된 정책 π_{k+1}를 반환해야 합니다.

힌트 : util.py에 정의된 기본값이 0인 dictionary 자료형인 util.Counter 클래스를 사용하세요. totalCount와 같은 함수는 코드를 단순화 시켜줍니다. 하지만, argMax는 주의하세요. 당신이 원하는 key값이 counter안에 없을 수도 있습니다.

참고 : MDP 상태가 없는 경우의 케이스를 처리하세요. (향후 보상을 위해 이것이 무엇을 의미하는지 생각하세요.)

구현을 테스트하려면 autograder를 실행하세요.

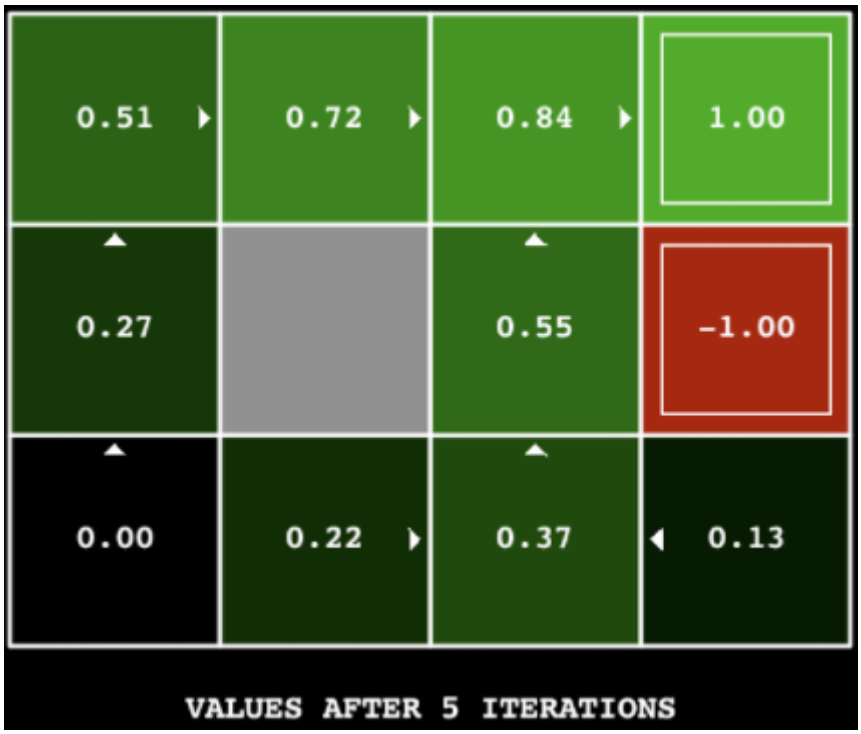
python autograder.py -q q1

다음 명령은 정책을 계산하고 10번 실행하는 valueIterationAgent를 실행합니다. 키를 누르면 값, Q값 및 시뮬레이션을 순환합니다. 시작 상태의 값(GUI에서 읽을 수 있는 V(start))과 경험적 평균 보상(10회 완료 후 출력 됨)이 아주 가까운 것을 발견해야 합니다.

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

힌트 : 기본 BookGrid에서 5번 반복할 경우 실행되는 값이 다음과 같은 결과로 나타납니다.

Python gridwolrd.py -a value -i 5



채점 : value iteration agent는 새로운 그리드에서 채점됩니다. 고정된 반복 횟수 및 수렴 후(예 : 100회 반복 후) 가치, Q-값 및 정책을 확인합니다.

Q2 – Bridge Crossing Analysis

BridgeGrid는 낮은 보상 종료 상태와 높은 보상 종료 상태가 양쪽에 높은 음의 보상의 종료 상태가 있는 좁은 "bridge"로 분리된 그리드 지도입니다

Agent는 보상이 낮은 상태와 가까운 지역에서 시작됩니다. 기본 discount는 0.9, 기본 noise는 0.2인 최적의 정책은 다리를 건너지 않는 것 입니다. 최적 정책으로 agent가 다리를 통과할 수 있도록 discount 및 noise 매개 변수 중 **하나만** 변경하세요. Analysis.py의 question2()에 답을 입력하세요. (noise는 agent가 작업을 수행 할 때 의도하지 않은 후속 상태가 수행되는 빈도를 의미합니다.) 기본값은 다음과 같습니다.

Python gridworld.py -a value -i 100 -g BridgeGrid –discount 0.9 –noise 0.2

채점 : 주어진 매개 변수 중 하나만 변경했는지 확인하고, 이 변경 사항을 적용하면 Value Iteration agent가 다리를 통과해야 합니다. 답변을 확인하려면 autograder를 실행시키세요.

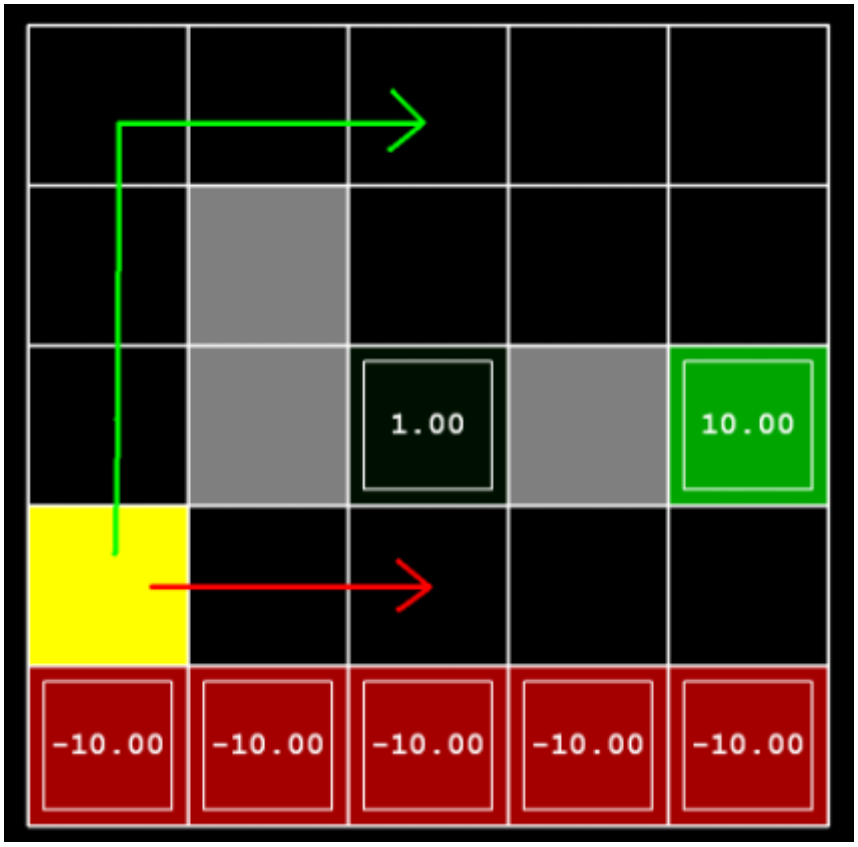
python autograder.py -q q2

Q3 – Policies

아래 표시된 DiscountGrid 레이아웃을 고려하세요. 이 그리드에는 두개의 종료 상태가 있습니다. 양수 보상(가운데 열에서)이 가까운 곳은 +1 먼곳은 +10을 줍니다. 그리드에서의 하단 행은 음수 보상을(적색으로 표시) 주는 상태로 구성되어 있습니다. 이 "cliff"영역의 각 상태는 보상 -10을 가지고 있습니다. 시작 상태는 노란색 정사각형입니다. 우리는 두가지 종류의 경로를 구별합니다.

(1) "risk the cliff"경로와 그리드의 하단 행 근처를 이동하는 것. 이 경로는 짧지만 큰 부정적 보상을 얻을 위험이 있고, 그림의 빨간 화살표로 표시되어 있습니다.

(2) "avoid the cliff"그리드의 상단 가장자리를 따라 이동하는 것. 이 경로는 길지만 큰 부정적인 비용을 발생시킬 가능성이 더 낮습니다. 그림의 녹색 화살표로 표시되어 있습니다.



이 질문에서 몇가지 다른 유형의 최적 정책을 생성하기 위해 discount, noise, living reward 매개변수의 설정을 선택하여야 할 것입니다. 각 부분의 매개변수 값 설정은 agent가 noise가 없는 최적의 정책을 따랐을 때 주어진 동작을 나타낼 수 있는 속성을 가져야 합니다. 매개 변수의 설정에 대해 특정 동작이 수행되지 않으면 문자열 "NOT POSSIBLE"을 반환하여 정책이 불가능 하다고 하세요.

다음은 **생성해야 할 최적의 정책 유형**입니다.

1. **절벽(-10)을 무릅쓰고 가까운 출구(+1)를 선호하는 것**
2. **절벽(-10)을 피하고 가까운 출구(+1)을 선호하는 것**
3. **절벽(-10)을 무릅쓰고 먼 출구(+10)을 선호하는 것**
4. **절벽(-10)을 피하고 먼 출구(+10)을 선호하는 것**
5. **모든 보상(출구 및 절벽)을 피하는 것. 따라서 에피소드는 종료되지 않아야 합니다.**

답변을 확인하려면 aotugrader를 실행하세요

python autograder.py -q q3

question3a() ~ question3e()는 각각 analyze.py에서 (discount, noise, living reward)의 3개를 반환해야 합니다.

참고 : GUI에서 정책을 확인할 수 있습니다. 예를 들어 1번의 정답을 사용하면 (0,1)의 화살표는 동쪽을 가리키고, (1,1)의 화살표는 동쪽을, (2,1)의 화살표는 북쪽을 가리켜야 합니다.

참고 : 일부 컴퓨터에서는 화살표가 표시되지 않을 수 있습니다. 이 경우 키보드의 버튼을 눌러 qValue 디스플레이로 전환하고 각 상태에 대해 사용 가능한 q 값의 arg max를 취하여 직접 계산하세요.

채점 : 각각의 경우에 원하는 정책이 반환되는지 확인합니다.

Q4 – Q-Learning

Value iteration agent는 실제 경험에서 배우지 않습니다. 오히려 실제 환경과 상호 작용하기 전에 완전한 정책에 도달하기 위해 MDP 모델에 의존합니다. 환경과의 상호작용을 할 때, 사전 계산된 정책(예 : reflex agent)을 따릅니다. 이러한 구분은 그리드 월드와 같은 시뮬레이션 환경에선 미세할 수 있지만, 실제 MDP를 사용할 수 없는 현실 세계에서는 매우 중요합니다.

이제 여러분은 구조에 거의 영향을 주지 않는 **Q-learning agent**를 작성할 것입니다. 대신 업데이트(state, action, nextState, reward)를 통해 환경과의 상호 작용에 대한 시행 착오를 통해 배우게 될 것입니다. Q-learner의 stub이 qlearningAgents.py의 QLearningAgent에 정의되어 있고, '-a q' 옵션을 사용하여 선택할 수 있습니다. 이 문제는 **update, computeValueFromQValues, getQValue, 그리고 computeActionFromQValues 함수를 구현하는 것**입니다.

참고 : computeActionFromQValues의 경우 더 나은 동작을 위해 임의로 묶음을 끊어야 합니다. Random.choice()함수가 도움이 될 것입니다. 특정 상태에서 이전에 보지 못한 agent의 Q 값, 특히 Q 값이 0이고 이전에 보았던 모든 작업에 음의 Q값이 있는 경우 보이지 않는 행동이 최적일 수 있습니다.

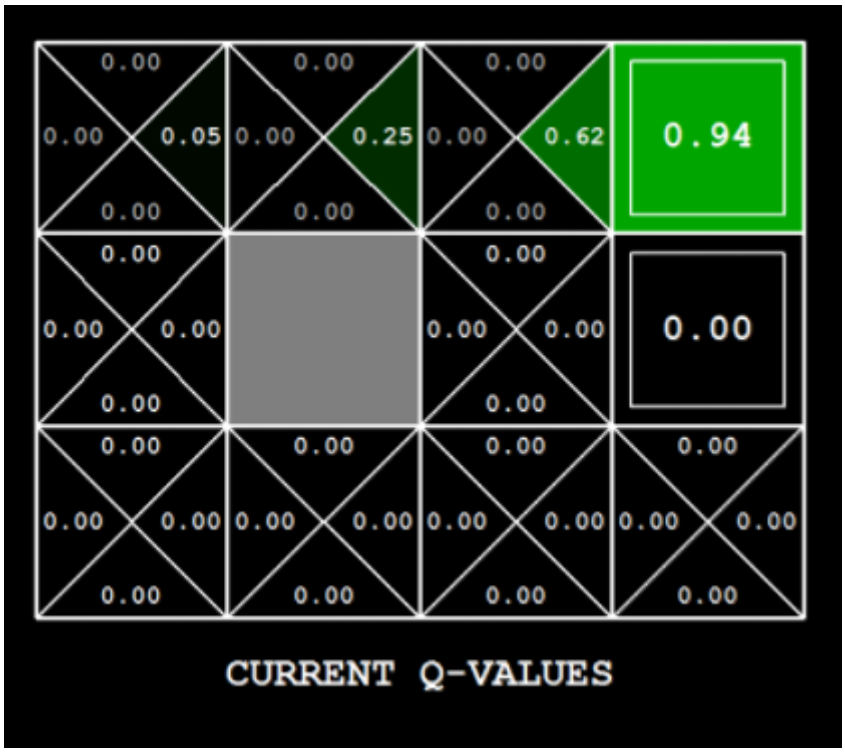
중요 : computeValueFromQValues 및 computeActionFromQValues 함수에서 getQValue를 호출하여 Q 값에만 액세스 해야 합니다. 이 abstraction은 state-action 쌍 대신에 state-action 쌍의 기

능을 직접 사용하기 위해 getQValue를 재정의 할 때 question 8에 유용합니다.

Q-learning 업데이트가 준비된 상태에서 Q-learner가 키보드를 사용하여 수동 제어를 통해 학습하는 것을 볼 수 있습니다.

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

-k는 agent가 배울 수 있는 에피소드의 수를 제어한다는 것을 기억하세요. Agent가 이동한 상태가 아닌 현재의 상태에 대해 어떻게 학습하는지를 확인하고 "leaves learning in its wake"(깨우기 학습)을 봅니다. 힌트 : 디버깅을 돕기 위해 –noise 0.0 매개변수를 사용하여 노이즈를 끌 수 있습니다.(물론 Q-learning을 덜 흥미롭게 만듭니다.) 수동으로 4개의 에피소드에 대한 최적 경로를 따라 수동으로 Pacman을 북쪽과 동쪽으로 조종하면 다음과 같은 Q 값이 표시됩니다.



채점 : Q-learning agent를 실행하고 각 사례에 동일한 예제 세트가 제공 될 때 참조 구현과 동일한 Q 값 및 정책을 학습하는지 확인합니다. 구현을 채점하려면 autograder를 실행하십시오.

python autograder.py -q q6

Q5 – Epsilon Greedy

getAction에서 엡실론-탐욕 행동 선택 구현하여 Q-learning agent를 완성하세요. 즉, 엡실론 값에서는 무작위로 선택하고, 그렇지 않은 경우 현재 최상의 Q-value를 따르세요. 임의의 액션 선택이

최상의 선택일 수 있습니다. 즉, 임의의 차선의 액션을 선택해서는 안되며, 행동가능 한 액션 중에서 선택을 하여야 합니다.

Random.choice 함수를 호출하여 리스트에서 무작위로 균일하게 요소를 선택할 수 있습니다. 확률 p로 True를 리턴하고 확률 1-p로 False를 리턴하는 util.flipCoin(p)를 사용하여 성공 확률 p의 2진수로 시뮬레이션 할 수 있습니다.

getAction 함수를 구현 한 후, gridworld에 있는 agent의 다음 동작을 관찰하십시오. (엡실론 = 0.3)

python gridworld.py -a q -k 100

최종 Q 값은 특히 가치 있는 경로(잘 학습되어 잘 가는 경로)인 경우, value iteration agent와 유사해야 합니다. 그러나 무작위 작업 및 초기 학습 단계로 인해 평균 수익률이 Q 값 예측보다 낮습니다.

다른 엡실론 값에 대해 다음과 같은 시뮬레이션을 관찰 할 수도 있습니다. Agent의 행동이 예상한 것과 일치합니까?

python gridworld.py -a q -k 100 --noise 0.0 -e 0.1

python gridworld.py -a q -k 100 --noise 0.0 -e 0.9

구현을 테스트 하려면 autograder를 실행하세요.

Python autograder.py – q q7

이제 추가 코드가 없어도 Q-learning crawler robot을 실행 할 수 있습니다.

python crawler.py

이 방법이 효과가 없다면, 아마도 GridWorld 문제에 너무 특화된 코드를 작성했을 겁니다. 모든 MDP에 일반화 하도록 만들어야 합니다.

그러면 Q-learner를 사용하는 수업에서 크롤링 로봇이 호출됩니다. 다양한 학습 매개변수를 사용하여 에이전트의 정책 및 동작에 미치는 영향을 확인하십시오. 단계 지연(the step delay)은 시뮬레이션의 매개 변수이지만 학습 속도와 엡실론은 학습 알고리즘의 매개 변수이며 discount factor는 환경의 속성이라는 점에 유의하십시오.

Q6 – Bridge Crossing Revisited

먼저 50개의 에피소드에 대한 무의미한 BridgeGrid의 기본 학습 속도로 완전 무작위 Q-learner를 학습시키고 최적의 정책을 찾는지 관찰합니다.

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

이제 엡실론(epsilon) 0을 사용하여 같은 실험을 시도해보세요.

50회 반복 후에 최적의 정책을 배울 가능성이 높은 (99% 이상) 엡실론과 학습률(learning rate)이 있습니까? Analysis.py의 Question8()에 그 **값이 있는 경우 (엡실론, 학습률)를 반환**하고 **없을 경우 "NOT POSSIBLE"을 반환**해야 합니다. 엡실론은 -e에 의해 학습률은 -l에 의해 제어됩니다.

참고 : 답은 행동들을 선택하는 데 사용된 정확한 측정 메커니즘에 의존해서는 안됩니다. 예를 들어 전체 BridgeGrid를 90도 회전했더라도 정답이 되어야 한다는 것을 의미합니다.

채점을 위해서는 autograder를 실행시키세요.

Python autograder.py – q q8

아래 4 문제는 세종대 인공지능 강의 평가 대상 문제들이 아닙니다. 그렇지만 버클리 코스 상에서 나오는 문제이므로 참고하시고자 하시는 분은 아래의 문제들을 참고하시면 되겠습니다.

Q – Asynchronous Value Iteration

- 세종대 강의의 채점에 안 들어갑니다.

비동기 값 반복

valueIterationAgents.py에서 부분적으로 지정한 AsynchronousValueIterationAgent에 값 반복 에이전트를 작성합니다. 값 반복 에이전트는 강화 학습 에이전트가 아닌 오프라인 계획자이므로 관련 교육 옵션은 초기 계획 단계에서 실행해야하는 값 반복의 반복 횟수 (옵션 -i)입니다. AsynchronousValueIterationAgent는 생성시 MDP를 사용하고 생성자가 반환하기 전에 지정된 반복 횟수 동안 순환 값 반복 (다음 단락에서 설명 함)을 실행합니다. 이 값 반복 코드는 모두 생성자 (__init__ 메소드) 안에 있어야합니다.

이 클래스를 AsynchronousValueIterationAgent라고 부르는 이유는 배치 스타일 업데이트를 수행하는 것과 달리 각 반복에서 하나의 상태 만 업데이트하기 때문입니다. 순환 값 반복이 작동하는 방식은 다음과 같습니다. 첫 번째 반복에서는 상태 목록의 첫 번째 상태 값만 업데이트하십시오. 두 번째 반복에서 두 번째 값만 업데이트하십시오. 각 상태의 값을 한 번 업데이트 할 때까지 계속 진행 한 다음 후속 반복의 첫 번째 상태에서 다시 시작하십시오. 업데이트를 위해 선택된 상태가 터미널 인 경우 해당 반복에서 아무 것도 발생하지 않습니다. 코드 스켈레톤에 정의 된 states 변수를 인덱싱해야합니다.

다시 말해, 값 반복 상태 업데이트 방정식은 다음과 같습니다.

$$V_{i+1}^{*}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_{i}^{*}(s') \right]$$

값 반복은 클래스에서 논의 된 것처럼 고정 소수점 방정식을 반복합니다. 임의의 순서 (즉, 상태를 임의로 선택, 값을 업데이트 및 반복) 또는 일괄 처리 스타일 (예 : Q1)과 같이 여러 가지 방법으로 상태 값을 업데이트 할 수도 있습니다. Q4에서는 또 다른 기술을 연구 할 것입니다.

AsynchronousValueIterationAgent는 Q1의 ValueIterationAgent에서 상속되므로 구현해야하는 유일한 방법은 runValueIteration뿐입니다. 슈퍼 클래스 생성자가 runValueIteration을 호출하기 때문에 이를 재정의하면 원하는대로 에이전트의 비헤이비어를 변경하기에 충분합니다.

참고 : MDP에 상태가없는 경우 케이스를 처리해야합니다 (향후 보상에 대해 이것이 무엇을 의미하는지 생각하십시오).

구현을 테스트하려면 자동 기록기를 실행하십시오. 실행할 시간은 1 초 미만입니다. 시간이 오래 걸리면 나중에 프로젝트에서 문제가 발생할 수 있으므로 구현을보다 효율적으로 수행하십시오.

python autograder.py -q q4

다음 명령은 정책을 계산하고 10 번 실행하는 Gridworld에 AsynchronousValueIterationAgent를 로드합니다. 키를 누르면 값, Q 값 및 시뮬레이션을 순환합니다. 시작 상태의 값 (GUI에서 읽을 수있는 V (시작))과 경험적으로 얻은 평균 보상 (10 회 실행 완료 후 인쇄 됨)은 아주 가깝습니다.

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

채점 : 가치 반복 에이전트는 새로운 그리드에서 채점됩니다. 고정 된 반복 횟수 및 수렴 후 (예 : 1000 회 반복 후) 가치, Q- 값 및 정책을 확인합니다.

Question 4 (1 point): Asynchronous Value Iteration

Write a value iteration agent in `AsynchronousValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option `-i`) in its initial planning phase. `AsynchronousValueIterationAgent` takes an MDP on construction and runs *cyclic* value iteration (described in the next paragraph) for the specified number of iterations before the constructor returns. Note that all this value iteration code should be placed inside the constructor (`__init__` method).

The reason this class is called `AsynchronousValueIterationAgent` is because we will update only **one** state in each iteration, as opposed to doing a batch-style update. Here is how cyclic value iteration works. In the first iteration, only update the value of the first state in the states list. In the second iteration, only update the value of the second. Keep going until you have updated the value of each state once, then start back at the first state for the subsequent iteration. **If the state picked for updating is terminal, nothing happens in that iteration.** You should be indexing into the `states` variable defined in the code skeleton.

As a reminder, here's the value iteration state update equation:

$$V_{i+1}^{*}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_{i}^{*}(s') \right]$$

Value iteration iterates a fixed-point equation, as discussed in class. It is also possible to update the state values in different ways, such as in a random order (i.e., select a state randomly, update its value, and repeat) or in a batch style (as in Q1). In Q4, we will explore another technique.

`AsynchronousValueIterationAgent` inherits from `ValueIterationAgent` from Q1, so the only method you need to implement is `runValueIteration`. Since the superclass constructor calls `runValueIteration`, overriding it is sufficient to change the agent's behavior as desired.

Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder. It should take less than a second to run. **If it takes much longer, you may run into issues later in the project, so make your implementation more efficient now.**

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

The following command loads your `AsynchronousValueIterationAgent` in the Gridworld, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

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

Grading: Your value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g., after 1000 iterations).

Q – Prioritized Sweeping Value Iteration

우선 순위가 적용된 반복 값 반복

이제 valueIterationAgents.py에서 부분적으로 지정된 PrioritizedSweepingValueIterationAgent를 구현합니다. 이 클래스는 AsynchronousValueIterationAgent에서 파생되므로 변경해야하는 유일한 메서드는 실제로 값 반복을 실행하는 runValueIteration뿐입니다.

우선 순위 스위핑은 정책을 변경할 가능성이있는 방식으로 상태 값의 업데이트를 집중시킵니다.

이 프로젝트에서는 표준 우선 순위 스위핑 알고리즘의 단순화 된 버전을 구현합니다.이 알고리즘

은이 백서에서 설명합니다. 이 알고리즘을 우리의 설정에 적용했습니다. 먼저, 상태 s의 전임자를 몇 가지 조치를 취하여 0에 도달 할 확률이 0 인 모든 상태로 정의합니다. 매개 변수로 전달 된 theta는 상태 값을 업데이트할지 여부를 결정할 때 오류에 대한 우리의 허용 오차를 나타냅니다. 구현시 따라야 할 알고리즘은 다음과 같습니다.

• 모든 주에서 전임자를 계산합니다.

• 비어있는 우선 순위 대기열을 초기화합니다.

• 각 비 종단 상태 s에 대해 다음을 수행합니다.

(참고 :이 질문에 대한 자동 검색 기능을 사용하려면 self.mdp.getStates ()가 반환 한 순서대로 상태를 반복해야합니다.

• self.values에있는 s의 현재 값과 s에서 가능한 모든 동작에 대한 가장 높은 Q 값의 차이의 절대 값을 찾습니다 (이 최대 Q 값은 s 값이 무엇인지 나타냄). 이 번호를 전화하세요. 이 단계에서 self.values [s]를 업데이트하지 마십시오.

우선 순위 -diff (우선 순위가 음수 임)로 우선 순위 대기열에 s를 넣습니다. 우선 순위 큐가 최소 힙이므로 음수를 사용하지만 더 높은 오류가있는 상태 업데이트의 우선 순위를 지정하려고합니다.

• 0, 1, 2, ...의 반복에서 self.iterations - 1은 다음을 수행합니다.

• 우선 순위 대기열이 비어 있으면 종료합니다.

• 우선 순위 큐에서 상태를 팝합니다.

• s 값을 self.values에 업데이트합니다 (터미널 상태가 아닌 경우).

• s의 각 선행 p에 대해 다음을 수행하십시오.

• self.values에서 p의 현재 값과 p에서 가능한 모든 동작에 대해 가장 높은 Q 값 (이 최대 Q 값은 p의 값을 나타냄)의 차이의 절대 값을 찾습니다. 이 번호를 전화하세요. 이 단계에서 self.values [p]를 업데이트하지 마십시오.

• diff> theta 인 경우 p가 우선 순위 큐와 같지 않거나 낮은 우선 순위 인 경우 우선 순위가 -diff 인 우선 순위 대기열로 p를 푸시 (음수)합니다. 이전과 마찬가지로 우선 순위 큐가 최소 힙이므로 음수를 사용하지만 더 높은 오류가있는 상태 업데이트의 우선 순위를 지정하려고합니다.

구현에 관한 몇 가지 중요한 참고 사항 :

• 상태의 선행 작업을 계산할 때는 중복을 피하기 위해 목록이 아닌 집합에 해당 상태를 저장해야합니다.

• 구현시 util.PriorityQueue를 사용하십시오. 이 클래스의 업데이트 메서드는 유용 할 수 있습니다.

문서를보십시오.

구현을 테스트하려면 자동 기록기를 실행하십시오. 실행하려면 약 1 초가 걸립니다. 시간이 오래 걸리면 나중에 프로젝트에서 문제가 발생할 수 있으므로 구현을보다 효율적으로 수행하십시오.

python autograder.py -q q5

다음 명령을 사용하여 Gridworld에서 PrioritizedSweepingValueIterationAgent를 실행할 수 있습니다.

python gridworld.py -a priosweepvalue -i 1000

채점 : 우선 순위가 매겨진 값 반복 에이전트가 새 그리드에서 채점됩니다. 고정 된 반복 횟수 및 수렴 후 (예 : 1000 회 반복 후) 가치, Q- 값 및 정책을 확인합니다.

Question 5 (3 points): Prioritized Sweeping Value Iteration

You will now implement `PrioritizedSweepingValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Note that this class derives from `AsynchronousValueIterationAgent`, so the only method that needs to change is `runValueIteration`, which actually runs the value iteration.

Prioritized sweeping attempts to focus updates of state values in ways that are likely to change the policy.

For this project, you will implement a simplified version of the standard prioritized sweeping algorithm, which is described in this paper. We've adapted this algorithm for our setting. First, we define the **predecessors** of a state `s` as all states that have a **nonzero** probability of reaching `s` by taking some action. `theta`, which is passed in as a parameter, will represent our tolerance for error when deciding whether to update the value of a state. Here's the algorithm you should follow in your implementation.

- Compute predecessors of all states.
- Initialize an empty priority queue.
- For each non-terminal state `s`, do:
  (note: to make the autograder work for this question, you must iterate over states in the order returned by `self.mdp.getStates()`)

- Find the absolute value of the difference between the current value of `s` in `self.values` and the highest Q-value across all possible actions from `s` (this max Q-value represents what the value of `s` should be); call this number `diff`. Do NOT update `self.values[s]` in this step.
- Push `s` into the priority queue with priority `-diff` (note that this is **negative**). We use a negative because the priority queue is a min heap, but we want to prioritize updating states that have a **higher** error.
- For `iteration` in `0, 1, 2, ..., self.iterations - 1`, do:
- If the priority queue is empty, then terminate.
- Pop a state `s` off the priority queue.
- Update `s`'s value (if it is not a terminal state) in `self.values`.
- For each predecessor `p` of `s`, do:
- Find the absolute value of the difference between the current value of `p` in `self.values` and the highest Q-value across all possible actions from `p` (this max Q-value represents what the value of `p` should be); call this number `diff`. Do NOT update `self.values[p]` in this step.
- If `diff > theta`, push `p` into the priority queue with priority `-diff` (note that this is **negative**), as long as it does not already exist in the priority queue with equal or lower priority. As before, we use a negative because the priority queue is a min heap, but we want to prioritize updating states that have a **higher** error.

A couple of important notes on implementation:

- When you compute predecessors of a state, make sure to store them in a **set**, not a list, to avoid duplicates.
- Please use `util.PriorityQueue` in your implementation. The `update` method in this class will likely be useful; look at its documentation.

To test your implementation, run the autograder. It should take about 1 second to run. **If it takes much longer, you may run into issues later in the project, so make your implementation more efficient now.**

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

You can run the `PrioritizedSweepingValueIterationAgent` in the Gridworld using the following command.

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

Grading: Your prioritized sweeping value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g., after 1000 iterations).

Q – Q-learning and pacman

Q-Learning 및 Pacman

일부 Pacman을 플레이 할 시간! Pacman은 2 단계로 게임을합니다. 첫 번째 단계 인 교육에서 Pacman은 직책과 행동의 가치에 대해 배우기 시작합니다. 작은 그리드에서도 정확한 Q 값을 배우는 데는 매우 오랜 시간이 걸리기 때문에 Pacman의 교육 게임은 GUI (또는 콘솔) 디스플레이없이 기본적으로 자동 모드로 실행됩니다. Pacman의 교육이 완료되면 테스트 모드로 들어갑니다. 테스트 할 때 Pacman의 self.epsilonand self.alpha는 0.0으로 설정되어 Pacman이 학습 한 정책을 악용 할 수 있도록 효과적으로 Q-Learning을 중지하고 탐색을 중지합니다. 테스트 게임은 기본적으로 GUI에 표시됩니다. 코드를 변경하지 않으면 다음과 같이 매우 작은 그리드 용 Q-Learning Pacman을 실행할 수 있습니다.

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

PacmanQAgent는 이미 작성한 QLearningAgent의 측면에서 이미 정의되어 있습니다. PacmanQAgent는 팩맨 문제 (엡실론 = 0.05, 알파 = 0.2, 감마 = 0.8)에보다 효과적인 기본 학습 매개 변수를 가지고 있다는 점이 다릅니다. 위의 명령이 예외없이 작동하고 상담원이 최소 80 %의 시간에 이기면이 질문에 대한 전체 크레딧을 받게됩니다. Autograder는 2000 년 훈련 경기 이후 100 번의 테스트 게임을 실행합니다.

힌트 : QLearningAgent가 gridworld.py 및 crawler.py에서 작동하지만 smallGrid에서 Pacman에 대한 좋은 정책을 배우지 못하는 경우 getAction 및 / 또는 computeActionFromQValues 메소드가 보이지 않는 작업을 적절하게 고려하지 않을 수 있기 때문일 수 있습니다. 특히 보이지 않는 동작은 정의에 따라 0의 Q 값을 가지므로 표시된 모든 동작에 음의 Q 값이있는 경우 보이지 않는 동작이 최적 일 수 있습니다. util.Counter에서 argmax 함수를 조심하십시오!

참고 : 답변을 채점하려면 다음을 실행하십시오.

python autograder.py -q q9

참고 : 학습 매개 변수를 실험하려는 경우 -a 옵션을 사용할 수 있습니다 (예 : -a 엡실론 = 0.1, 알파 = 0.3, 감마 = 0.7). 그러면이 값은 에이전트 내부에서 self.epsilon, self.gamma 및 self.alpha로

액세스 할 수 있습니다.

참고 : 총 2010 개의 게임이 재생되지만 2000 년 첫 번째 게임은 훈련을위한 최초의 2000 게임 (출력 없음)을 지정하는 -x 2000 옵션 때문에 표시되지 않습니다. 따라서 Pacman이이 게임의 마지막 10 개만 플레이하는 것을 볼 수 있습니다. 훈련 게임의 수는 옵션 numTraining으로 상담원에게 전달됩니다.

참고 : 10 가지 훈련 게임을보고 무슨 일이 일어나고 있는지 보려면 다음 명령을 사용하십시오.

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

교육을받는 동안 Pacman이 어떻게 수행되는지에 대한 통계로 100 개의 게임마다 출력물을 볼 수 있습니다. 엡실론은 훈련 도중 긍정적 인 반응을 보였으므로 Pacman은 좋은 정책을 배운 후에도 제대로 플레이하지 못합니다. 이는 가끔 무작위 탐색을 유령으로 만들기 때문입니다. 벤치마킹으로 Pacman의 100 회 에피소드에 대한 보상이 포지티브가되기까지 1,000 회에서 1400 회 정도의 게임이 필요합니다. 훈련이 끝날 때까지, 그것은 긍정적으로 유지되어야하며 상당히 높습니다 (100에서 350 사이).

여기서 일어나는 일을 이해했는지 확인하십시오. MDP 상태는 Pacman이 직면 한 정확한 보드 구성이며, 복잡한 전환이 해당 상태로 변경 사항 전체를 설명합니다. Pacman이 이동했지만 유령이 응답하지 않은 중간 게임 구성은 MDP 상태가 아니지만 전환으로 묶입니다.

팩맨이 훈련을 마친 후에는 시험 게임에서 매우 안정적으로 승리해야합니다 (적어도 90 %의 시간). 이제 그는 배운 정책을 악용하고 있습니다.

그러나 겉으로는 단순한 mediumGrid에서 동일한 상담원을 교육하면 잘 작동하지 않습니다. 우리의 구현에서 Pacman의 평균 교육 보상은 교육을 통해 부정적입니다. 테스트 시간에, 그는 그의 테스트 게임을 모두 잃어 버릴 정도로 나쁘게 플레이합니다. 교육은 비효율에도 불구하고 오랜 시간이 걸릴 것입니다.

Pacman은 각 보드 구성이 별도의 Q 값을 가진 별도의 상태이므로 더 큰 레이아웃에서 승리하지 못합니다. 그는 유령으로 뛰는 것이 모든 위치에 나쁜 것이라는 것을 일반화 할 길이 없습니다. 분명히이 접근법은 확장되지 않습니다.

Question 9 (1 point): Q-Learning and Pacman

Time to play some Pacman! Pacman will play games in two phases. In the first phase, *training*, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacman's training games run in quiet mode by default, with no GUI (or console) display. Once Pacman's training is complete, he will enter *testing* mode. When testing, Pacman's `self.epsilon`and `self.alpha` will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his

learned policy. Test games are shown in the GUI by default. Without any code changes you should be able to run Q-learning Pacman for very tiny grids as follows:

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

Note that `PacmanQAgent` is already defined for you in terms of the `QLearningAgent` you've already written. `PacmanQAgent` is only different in that it has default learning parameters that are more effective for the Pacman problem (`epsilon=0.05, alpha=0.2, gamma=0.8`). You will receive full credit for this question if the command above works without exceptions and your agent wins at least 80% of the time. The autograder will run 100 test games after the 2000 training games.

Hint: If your `QLearningAgent` works for `gridworld.py` and `crawler.py` but does not seem to be learning a good policy for Pacman on `smallGrid`, it may be because your `getAction` and/or `computeActionFromQValues` methods do not in some cases properly consider unseen actions. In particular, because unseen actions have by definition a Q-value of zero, if all of the actions that *have* been seen have negative Q-values, an unseen action may be optimal. Beware of the argmax function from util.Counter!

Note: To grade your answer, run:

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

Note: If you want to experiment with learning parameters, you can use the option `-a`, for example `-a epsilon=0.1,alpha=0.3,gamma=0.7`. These values will then be accessible as `self.epsilon, self.gamma` and `self.alpha` inside the agent.

Note: While a total of 2010 games will be played, the first 2000 games will not be displayed because of the option `-x 2000`, which designates the first 2000 games for training (no output). Thus, you will only see Pacman play the last 10 of these games. The number of training games is also passed to your agent as the option `numTraining`.

Note: If you want to watch 10 training games to see what's going on, use the command:

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

During training, you will see output every 100 games with statistics about how Pacman is faring. Epsilon is positive during training, so Pacman will play poorly even after having learned a good policy: this is because he occasionally makes a random exploratory move into a ghost. As a benchmark, it should take between 1,000 and

1400 games before Pacman's rewards for a 100 episode segment becomes positive, reflecting that he's started winning more than losing. By the end of training, it should remain positive and be fairly high (between 100 and 350).

Make sure you understand what is happening here: the MDP state is the *exact* board configuration facing Pacman, with the now complex transitions describing an entire ply of change to that state. The intermediate game configurations in which Pacman has moved but the ghosts have not replied are *not* MDP states, but are bundled in to the transitions.

Once Pacman is done training, he should win very reliably in test games (at least 90% of the time), since now he is exploiting his learned policy.

However, you will find that training the same agent on the seemingly simple `mediumGrid` does not work well. In our implementation, Pacman's average training rewards remain negative throughout training. At test time, he plays badly, probably losing all of his test games. Training will also take a long time, despite its ineffectiveness.

Pacman fails to win on larger layouts because each board configuration is a separate state with separate Q-values. He has no way to generalize that running into a ghost is bad for all positions. Obviously, this approach will not scale.

Q – Approximate Q-learning

- 세종대 강의의 채점에 안 들어갑니다.

대략적인 Q-Learning

많은 상태에서 동일한 기능을 공유 할 수 있는 상태 기능에 대한 가중치를 학습하는 대략적인 Q-learning 에이전트를 구현합니다. PacmanQAgent의 하위 클래스 인 qlearningAgents.py의 ApproximateQAgent 클래스에 구현을 작성합니다.

참고 : 근사 Q- 학습은 상태 및 동작 쌍에 대한 특징 함수 f (s, a)의 존재를 가정하며, 이는 벡터 f1 (s, a) .. fi (s, a) .. fn (s, a)를 나타냅니다. featureExtractors.py에서 기능을 제공합니다. 기능 벡터는 util.Counter (사전과 유사) 기능과 값의 0이 아닌 쌍을 포함하는 개체입니다. 생략 된 모든 피쳐는 값이 0입니다.

근사 Q 함수는 다음 형식을 취합니다.

각각의 가중치 wi는 특정 특징 fi (s, a)와 관련된다. 코드에서, 체이 벡터를 체력 값 (체력 추출기가 반환하는)을 매핑하는 사전으로 구현해야합니다. Q 값을 업데이트 한 것과 마찬가지로 체중 벡터

를 업데이트합니다.

차이 용어는 일반적인 Q-learning에서와 동일하고, r은 경험상의 보상이다.

기본적으로 ApproximateQAgent는 모든 (상태, 동작) 쌍에 단일 기능을 할당하는 IdentityExtractor를 사용합니다. 이 기능 추출기를 사용하면 대략적인 Q-learning 에이전트가 PacmanQAgent와 동일하게 작동해야합니다. 다음 명령을 사용하여이를 테스트 할 수 있습니다.

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

중요 : ApproximateQAgent는 QLearningAgent의 서브 클래스이므로 getAction과 같은 여러 메소드를 공유합니다. QLearningAgent의 메소드가 Q 값에 직접 액세스하는 대신 getQValue를 호출하는지 확인하십시오. 그러면 대략적인 에이전트에서 getQValue를 대체 할 때 새로운 대략적인 q 값이 조치를 계산하는 데 사용됩니다.

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

근사자가 신원 기능으로 올바르게 작동한다고 확신하고 나면 쉽게이기는 법을 배울 수있는 맞춤형 기능 추출기로 대략적인 Q-learning 에이전트를 실행하십시오.

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

ApproximateQAgent에 대한 더 큰 레이아웃 일지라도 아무런 문제가 없어야합니다. (경고 : 이것은 훈련하는 데 몇 분이 걸릴 수 있습니다)

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

오류가 없다면 약 50 개의 교육용 게임만으로도이 간단한 기능으로 대략적인 Q-Learning 에이전트가 거의 언제나 승리해야합니다.

채점 : 귀하의 대략적인 Q-learning 에이전트를 실행하고 동일한 Q-value 및 기능 가중치를 참조 구현과 동일한 배움을 얻을 수 있는지 확인합니다. 구현을 채점하려면 자동 검색기를 실행하십시오.

python autograder.py -q q10

Question 10 (3 points): Approximate Q-Learning

Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation

in `ApproximateQAgent` class in `qlearningAgents.py`, which is a subclass of `PacmanQAgent`.

Note: Approximate Q-learning assumes the existence of a feature function f(s,a) over state and action pairs, which yields a vector $f_1(s,a)$ .. $f_i(s,a)$ .. $f_n(s,a)$ of feature values. We provide feature functions for you in `featureExtractors.py`. Feature vectors are `util.Counter` (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have value zero.

The approximate Q-function takes the following form

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

where each weight w_i is associated with a particular feature $f_i(s,a)$. In your code, you should implement the weight vector as a dictionary mapping features (which the feature extractors will return) to weight values. You will update your weight vectors similarly to how you updated Q-values:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$
$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

Note that the $\mathrm{difference}$ term is the same as in normal Q-learning, and r is the experienced reward.

By default, `ApproximateQAgent` uses the `IdentityExtractor`, which assigns a single feature to every `(state,action)` pair. With this feature extractor, your approximate Q-learning agent should work identically to `PacmanQAgent`. You can test this with the following command:

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

Important: `ApproximateQAgent` is a subclass of `QLearningAgent`, and it therefore shares several methods like `getAction`. Make sure that your methods in `QLearningAgent` call `getQValue` instead of accessing Q-values directly, so that when you override `getQValue` in your approximate agent, the new approximate q-values are used to compute actions.

Once you're confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

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

Even much larger layouts should be no problem for your `ApproximateQAgent`. (*warning*: this may take a few minutes to train)

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

If you have no errors, your approximate Q-learning agent should win almost every time with these simple features, even with only 50 training games.

Grading: We will run your approximate Q-learning agent and check that it learns the same Q-values and feature weights as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

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

Congratulations! You have a learning Pacman agent!