# Homework for Reinforcement Learning

2024 Fall

## Overview

**Due November 30, 23:59, GMT+8**

This homework includes written and programming problems related to Lecture 6-9.

- For **writing component**, please write your answers in a `pdf` file. You can use LaTeX, convert from Word, from notebook apps on tablets, scan your handwritten documents, and etc. Your derivation and proof should be completed in the `pdf` file. Besides, your section titles and subsection titles should be the number of question. i.e., # Q1 ## Q1.1.
- For **programming component**, please submit your modified `reinforcement` project folder. Note that your analysis and some requested results should be included in the report `pdf` file.

Please organize your submitted files as follows:

```
report_${student_id}.pdf
reinforcement
    |--...
```

Finally, please zip the `report_${student_id}.pdf` file and `reinforcement` directory (rather than their parent directory) into `${student_id}.zip`. You should submit your `${student_id}.zip` to Web Learner before **November 30, 23:59, GMT+8**.

# 1 Writing Component (40 pts)

## 1.1 Performance Difference Lemma (15 pts)

Consider infinite horizon MDPs $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \gamma, \mathcal{R}, \mathcal{P}\}$ with inital state $s_0$. Let us denote $V^\pi(s)$ as the expected total reward of policy $\pi$ starting from $s$, i.e.,

$$V^\pi(s) \equiv \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) | \pi, s\right],$$

and denote the performance of policy $\pi$ as $\eta(\pi) = V^\pi(s_0)$.

Similarly, let us denote $Q^\pi(s, a)$ as the expected total reward of policy $\pi$ starting from executing action $a$ on state $s$, and let $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ be the advantage function.

Prove the following equality for two arbitrary policies $\pi$ and $\pi'$:

$$\eta(\pi') - \eta(\pi) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim d^{\pi'}(\cdot), a \sim \pi'(\cdot|s)} \left[A^\pi(s, a)\right],$$

where $d^\pi(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_t = s|\pi, s_0)$ is the discounted visit distribution with starting state $s_0$.

If you have no idea about this problem, you can try one of the following hints. (They may lead to slightly different proofs.)

Hint A: Directly computing the performance difference between $\pi'$ and $\pi$ may be not straightforward. You can imagine that there is an "intermediate policy", which selects action by $\pi'$ at step $t = 0$ and selects action by $\pi$ at step $t \geq 1$. What's the performance difference between $\pi'$ and the "intermediate policy"? What's the performance difference between the "intermediate policy" and $\pi$?

Hint B: Recall that for $V^{\pi'}(s)$, $Q^{\pi'}(s,a)$ and $\mathcal{R}(s,a)$, we have recurrence relations of $V^{\pi'}(s) = \mathbb{E}_{a\sim\pi'(\cdot|s)}[Q^{\pi'}(s,a)]$ and $Q^{\pi'}(s,a) = \mathcal{R}(s,a) + \gamma\mathbb{E}_{s'\sim\mathcal{P}(\cdot|s,a)}[V^{\pi'}(s')]$. Show that we can establish similar recurrence relations among $V^{\pi'}(s) - V^{\pi}(s)$, $Q^{\pi'}(s,a) - Q^{\pi}(s,a) + A^{\pi}(s,a)$ and $A^{\pi}(s,a)$.

## 1.2 Overestimation bias in Q-Learning (10 pts)

In Q-Learning, we encounter the issue of overestimation bias. This issue comes from the fact that to calculate our targets, we take a maximum of $\hat{Q}$ over actions. We use a maximum over estimated values ($\hat{Q}$) as an estimate of the maximum value ($\max_a Q(s,a)$), which can lead to significant positive bias.

(a) (5 pts) Assume that our estimated $\hat{Q}$ function is an unbiased estimator of $Q$, the true Q function, i.e. $\mathbb{E}\left[\hat{Q}(s,a)\right] = Q(s,a)$ (here the expectation is w.r.t. runs of experiments and data used to estimate $\hat{Q}$). Prove that even in this simplified setting, the overestimation issue occurs, i.e., the following inequality always holds:

$$\forall s, \mathbb{E}\left[\max_a \hat{Q}(s,a)\right] \geq \max_a Q(s,a).$$

(b) (5 pts) Suppose that we use "double Q-Learning". Here, we learn two independent $\hat{Q}$ functions, $\hat{Q}_1$ and $\hat{Q}_2$, and we use one estimate to determine the maximizing action, $\hat{a}^* = \arg\max_a \hat{Q}_1(s,a)$, and the other to estimate the action's value, i.e. $\hat{Q}_{\text{double}} = \hat{Q}_2(s,\hat{a}^*)$ $= \hat{Q}_2(s, \arg\max_a \hat{Q}_1(s,a))$. We assume that both $\hat{Q}_1$ and $\hat{Q}_2$ are unbiased estimates of $Q$. We also assume that $\hat{Q}_1(\cdot,\cdot)$ and $\hat{Q}_2(\cdot,\cdot)$ are independent random variables in the function space. Prove that double Q-Learning can lead to underestimation in the sense that

$$\mathbb{E}\left[\hat{Q}_{\text{double}}\right] \leq \max_a Q(s,a).$$

Double Q-learning and its variants are widely used in modern RL algorithms. You can further read [1] if you are interested.

## 1.3 Baseline function for policy gradients (15 pts)

Assuming that our gradient estimate will be:

$$\widehat{\nabla_\theta J(\theta)} = \mathbb{E}_{s\sim d^{\pi_\theta}(\cdot), a\sim\pi_\theta(\cdot|s)}\left[\left(Q^\pi(s,a) - b(s)\right)\nabla_\theta \log \pi_\theta(a|s)\right].$$

(a) (5 pts) Prove this estimate is unbiased estimate of the true gradient for arbitrary baseline function $b(s)$.

(b) (10 pts) Find $b(s)$ that leads to the minimum variance estimate of the true gradient. Note that the gradient is a vector. For a random vector $\mathbf{x}$, we denote its variance as $\mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])^T(\mathbf{x} - \mathbb{E}[\mathbf{x}])]$. (Alternatively, you can treat $\theta$ as a scalar in this subproblem for simplicity.)

# 2 Programming Component: RL in Pacman (60 pts)

## 2.1 Introduction

In this project, you will implement value iteration and Q-learning. You will test your agents first on Gridworld, then apply them to a simulated robot controller (Crawler) and Pacman.

This project includes an autograder for you to grade your solutions on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It can be run for one particular question, such as q2, by:

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

It can be run for one particular test by commands of the form:

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

You can find all the code and supporting files in `reinforcement` folder. The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

**Files you'll edit**:
`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.

**Files you should read but NOT edit**:
`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`).

You can ignore other files.

**Files to Edit**: You will fill in portions of valueIterationAgents.py, qlearningAgents.py, and analysis.py during the assignment. Please do not change the other files in this distribution.

**Evaluation**: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the

autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

## 2.2 MDP

To get started, run Gridworld in manual control mode, which uses the arrow keys:

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

You will see the two-exit layout from class. The blue dot is the agent. Note that when you press up, the agent only actually moves north 80% of the time. Such is the life of a Gridworld agent!

You can control many aspects of the simulation. A full list of options is available by running:

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

The default agent moves randomly

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

You should see the random agent bounce around the grid until it happens upon an exit. Not the finest hour for an AI agent.

Note: The Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI) and then take the special 'exit' action before the episode actually ends (in the true terminal state called TERMINAL_STATE, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (-d to change; 0.9 by default).

Look at the console output that accompanies the graphical output (or use -t for all text). You will be told about each transition the agent experiences (to turn this off, use -q).

As in Pacman, positions are represented by (x,y) Cartesian coordinates and any arrays are indexed by [x][y], with 'north' being the direction of increasing y, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (-r).

## 2.3 Question 1 (15 pts): Value Iteration

Write a value iteration agent in `ValueIterationAgent`, 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. `ValueIterationAgent` takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computes k-step estimates of the optimal values, $V_k$. In addition to running value iteration, implement the following methods for `ValueIterationAgent` using $V_k$.

`computeActionFromValues(state)` computes the best action according to the value function given by self.values.

`computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by self.values.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.

*Important*: Use the "batch" version of value iteration where each vector $V_k$ is computed from a fixed vector $V_{k-1}$, not the "online" version where one single weight vector is updated in place. This means that when a state's value is updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration k-1 (even if some of the successor states had already been updated in iteration k).

*Note*: A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next k+1 rewards (i.e. you return $\pi_{k+1}$. Similarly, the Q-values will also reflect one more reward than the values (i.e. you return $Q_{k+1}$).

You should return the synthesized policy $\pi_{k+1}$.

*Hint*: Use the `util.Counter` class in `util.py`, which is a dictionary with a default value of zero. Methods such as `totalCount` should simplify your code. However, be careful with `argMax\verb`: the actual argmax you want may be a key not in the counter!

*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).

The following command loads your `ValueIterationAgent`, 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 value -i 100 -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 100 iterations).

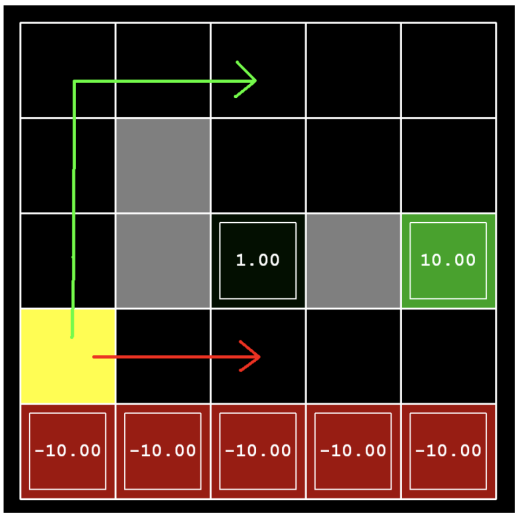
Please run the below command to see if your implementation passes all the autograder test cases.

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

The screenshot of the output should be included in the report `pdf` file.

## 2.4 Question 2 (10 pts): Policies

Consider the `DiscountGrid` layout, shown below. This grid has two terminal states with positive payoff (in the middle row), a close exit with payoff +1 and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this "cliff" region has payoff -10. The starting state is the yellow square. We distinguish between two types of paths: (1) paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in the figure below. (2) paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in the figure below.




In this question, you will choose settings of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types. (You can check `gridworld.py` for the roles of these parameters.) Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior. If a particular behavior is not achieved for any setting of the parameters, assert that the policy is impossible by returning the string `'NOT POSSIBLE'`.

Here are the optimal policy types you should attempt to produce:

- a. Prefer the close exit (+1), risking the cliff (-10)
- b. Prefer the close exit (+1), but avoiding the cliff (-10)
- c. Prefer the distant exit (+10), risking the cliff (-10)
- d. Prefer the distant exit (+10), avoiding the cliff (-10)
- e. Avoid both exits and the cliff (so an episode should never terminate)

`question2a()` through `question2e()` should each return a 3-item tuple of (discount, noise, living reward) in `analysis.py`.

*Note*: You can check your policies in the GUI. For example, using a correct answer to 3(a), the arrow in (0,1) should point east, the arrow in (1,1) should also point east, and the arrow in (2,1) should point north.

*Note*: On some machines you may not see an arrow. In this case, press a button on the keyboard to switch to qValue display, and mentally calculate the policy by taking the arg max of the available qValues for each state.

*Grading*: We will check that the desired policy is returned in each case.

Please run the below command to see if your implementation passes all the autograder test cases.

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

The screenshot of the output should be included in the report `pdf` file.

## 2.5 Question 3 (15 pts): Q-Learning

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridword, but it's very important in the real world, where the real MDP is not available.

You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update` method. A stub of a Q-learner is specified in `QLearningAgent` in `qlearningAgents.py`, and you can select it with the option `'-a q'`. For this question, you must implement the `update`, `computeValueFromQValues`, `getQValue`, and `computeActionFromQValues` methods.

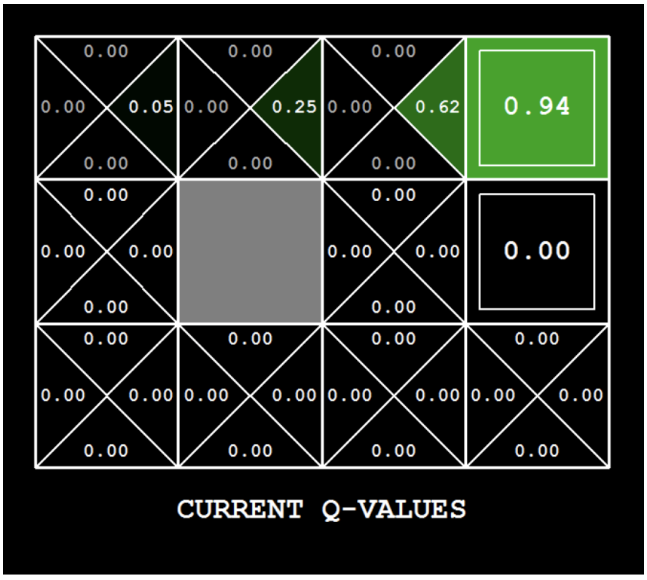
*Note*: For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function will help. In a particular state, actions that your agent hasn't seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.

Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q values by calling `getQValue`. This abstraction will be useful for question 8 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.

With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:

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

Recall that `-k` will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to, and "leaves learning in its wake." Hint: to help with debugging, you can turn off noise by using the `--noise 0.0` parameter (though this obviously makes Q-learning less interesting). If you manually steer Pacman north and then east along the optimal path for four episodes, you should see the following Q-values:




*Grading*: We will run your Q-learning agent and check that it learns the same Q-values and policy as our reference implementation when each is presented with the same set of examples.

Please run the below command to see if your implementation passes all the autograder test cases.

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

The screenshot of the output should be included in the report `pdf` file.

## 2.6 Question 4 (5 pts): Epsilon Greedy

Complete your Q-learning agent by implementing epsilon-greedy action selection in `getAction`, meaning it chooses random actions an epsilon fraction of the time, and follows its current best Q-values otherwise. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather any random legal action.

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

Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predict because of the random actions and the initial learning phase.

You can choose an element from a list uniformly at random by calling the `random.choice` function. You can simulate a binary variable with probability p of success by using `util.flipCoin(p)`, which returns True with probability p and False with probability 1-p.

With no additional code, you should now be able to run a Q-learning crawler robot:

```
python crawler.py
```

If this doesn't work, you've probably written some code too specific to the `GridWorld` problem and you should make it more general to all MDPs.

Play around with the various learning parameters to see how they affect the agent's policies and actions. Note that the step delay is a parameter of the simulation, whereas the learning rate and epsilon are parameters of your learning algorithm, and the discount factor is a property of the environment.

Please run the below command to see if your implementation passes all the autograder test cases.

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

The screenshot of the output should be included in the report `pdf` file.

## 2.7 Question 5 (10 pts): Q-Learning and 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: 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.

Please run the below command to see if your implementation passes all the autograder test cases.

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

The screenshot of the output should be included in the report `pdf` file.

## 2.8 Question 6 (5 pts): 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 $f_1(s, a) \dots f_i(s, a) \dots f_n(s, a)$ 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:

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

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:

$$w_i \leftarrow w_i + \alpha \cdot \delta \cdot f_i(s, a) \tag{2}$$

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a) \tag{3}$$

Note that the $\delta$ 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.

Please run the below command to see if your implementation passes all the autograder test cases.

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

The screenshot of the output should be included in the report `pdf` file.

# Grading

This assignment counts for a final grade of 10 points. Following is the conversion of your homework score (implied by points after each exercise) and final grade:

$$\text{Final Grade} = \left\lceil \frac{\text{This homework score}}{100} \times 10 \right\rceil$$

**We will actively be checking for plagiarism.** If plagiarism is found, your final grade will be **F**.

# Acknowledgment

This homework references UC Berkeley and CMU courses.

# References

[1] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.