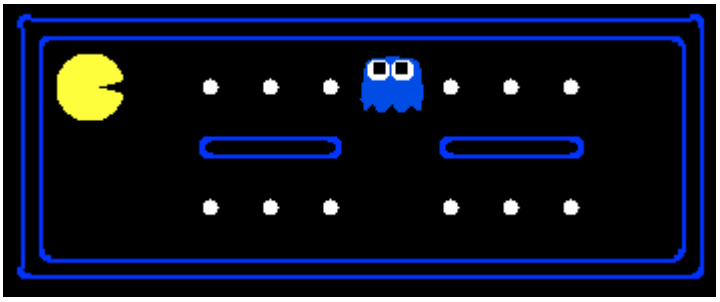
CPSC 470 – Artificial Intelligence
Problem Set #7 – Q-Learning
20 points
Due Friday April 19th, 11:59:59pm

Some reminders:

- **Grading contact:** Emmanuel Adeniran (emmanuel.adeniran@yale.edu) is the point of contact for initial questions about grading for this problem set.
- **Late assignments** are not accepted without a Dean's excuse.
- **Collaboration policy:** You are encouraged to discuss assignments with the course staff and with other students. However, you are required to implement and write any assignment on your own. This includes both pencil-and-paper and coding exercises. You are not permitted to copy, in whole or in part, any written assignment or program as part of this course. You are not to take code from any online repository or web source. You will not allow your own work to be copied. Homework assignments are your individual responsibility, and plagiarism will not be tolerated.
- **Students taking CPSC570:** There IS an extra section for this assignment, and you are looking at the wrong file!

I. Introduction to Q-Learning (Pacman) (20 points)



This problem is adapted from Berkeley's pacman assignment. In problem set 1, you implemented different types searches for pacman. In this assignment, you will revisit pacman and implement Q-learning. You will test your agents first on the Gridworld presented in class (during lectures 20 and 21), then apply them to the Pacman game.

Important Note: Although you are applying Q-learning to Pacman, you should write your agent to be as general as possible so that it can be applied to any game. When writing the agent, think of actions and states as abstract variables that are a part of any general Reinforcement Learning problem rather than the actions and states that are found only in Pacman.

Here are the files you will see in the starter code:

Files you'll edit:

qlearningAgents.py	Q-learning agents for Gridworld and Pacman.

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).

Files you can ignore:

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.
testParser.py	Parses autograder test and solution files
testClasses.py	General autograding test classes
reinforcementTestClasses.py	specific autograding test classes
valueIterationAgents.py	A value iteration agent for solving known MDPs.
analysis.py	A file not relevant to your assignment.

Part 1 Basic Q Learning (8 points)

You will 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(state, action, nextState, reward) method. A stub of a Q-learner is specified in QLearningAgent in *qlearningAgents.py*. For this question, you must implement the following functions:

- update
- computeValueFromQValues
- getQValue
- computeActionFromQValues

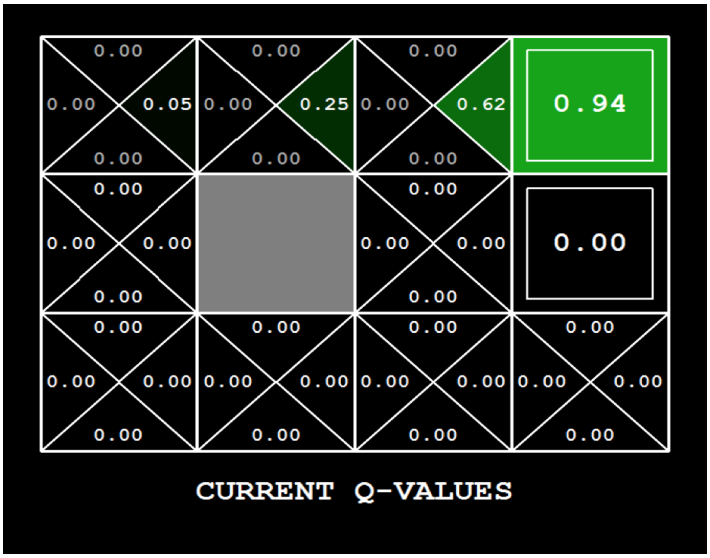
Note: For computeActionFromQValues, you should break ties randomly for better behavior. The random.choice() function will help. In a particular state, actions that your agent hasn't seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before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

Note: Make sure that in your computeValueFromQValues and computeActionFromQValues functions, you only access Q values by calling getQValue . This abstraction will be useful for question 3 when you override getQValue to use features of state-action pairs rather than state-action pairs directly.

After you have the functions implemented, you can watch your Q-learner learn under manual control, using the keyboard:

```
python3 gridworld.py -a q -k 5 -m
```

-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. To grade your implementation, run your code through the autograder on Gradescope.

The action selection used is called epsilon-greedy action selection which has been implemented for you in the function getAction. It chooses random actions an epsilon fraction of the time, and follows its current best Q-values otherwise.

Part 2: Q-learning and Pacman (2 points)

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:

```
python3 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. So there is no implementation needed in this section if you get the previous section right. 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 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 implementation, run your code through the autograder on Gradescope.

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:

```
python3 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 1,400 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 into 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.

Part 3: Approximating Q Learning (10 points)

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 the ApproximateQAgent class in *qlearningAgents.py*, which is a subclass of PacmanQAgent. You will need the complete the following functions:

- update
- getQValue

Note: You don’t need to complete the function final. There is a "*** YOUR CODE HERE *** " there, but the point is to print your weights for debugging purposes.

Note: Approximate Q-learning assumes the existence of a feature function f(s,a) over state and action pairs, which yields a vector f1(s,a) .. fi(s,a) .. fn(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:

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

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

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, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

```
python3 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)

```
python3 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 your code through the autograder on Gradescope.

Congratulations! You have a learning Pacman agent!

III. Submission

Please submit the file **qlearningAgents.py** on Gradescope to Problem Set 7 – Programming.

References/Additional Resources

[1] *Human-level control through deep reinforcement learning*, Nature. Mnih et al. (2015).
[2] Berkeley AI Materials, http://ai.berkeley.edu/project_overview.html.