National University of Singapore
School of Computing
CS3243 Introduction to Artificial Intelligence

Project 3: Reinforcement Learning

Issued: 7 October 2020 Due: 13 November 2020, 2359hrs

Objectives

The objective of this project is to:

1. Solve the Pacman Game via Reinforcement Learning
2. Gain experience with reinforcement learning.
3. Become familiar with the implementation of the Q-learning and approximate Q-learning algorithms.
4. Gain experience with feature extraction in a complex environment.

This project is worth 10% of your module grade.

General Project Requirements

The general project requirements are as follows:

- Group size: **2 students**
- Submission Deadline: **13 November 2020** (Friday), **2359 hours** (local time)
- Submission Platform: **LumiNUS** > **CS3243** > **Project 3** > **Project 3 Submission Folder**
- Submission Format: One standard (non-encrypted) **zip file**[1] containing all necessary project files. In particular, it should contain exactly two `.py` files (Pacman). Make only one submission per group.

As to the specific project requirements, you must complete and submit the following:

- You will implement a Q-learning algorithm that will learn how to play Pacman. In particular, you will provide implementations of the:
  1. Q-learning algorithm.
  2. Approximate Q-learning algorithm.
  3. Customised feature extractor.

[1] Note that it is the responsibility of the students in the project group to ensure that this file may be accessed by conventional means.

Academic Integrity and Late Submissions

Do note that any material used does not originate from you (e.g., is taken from another source), should not be used directly. You should do up the solutions on your own. Failure to do so constitutes plagiarism. In any case, you may not share materials between groups.

For projects submitted beyond the submission deadline, there will be a 20% penalty for submitting after the deadline and within 24 hours, 50% penalty for submitting after 24 hours past the deadline but within 48 hours, and 100% penalty for submitting more than 48 hours after the deadline For example, if you submit the project 30 hours after the deadline, and obtain 92%, a 50% penalty applies and you will only be awarded 46%.

The Pacman Game

In this project , you will implement a Q-learning agent as well as an approximate Q-learning agent and train them to play the game of Pacman[1]. This project includes an autograder for the first two subtasks. You can run the following command to grade yourself:

```
python autograder.py
```

It can also be run for one particular task, such as task 2, by:

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

The code for this project contains the following files:

Files you will edit and submit:	
`qlearningAgents.py`	Q-learning agents for Pacman
`featureExtractors.py`	Classes for extracting features on ⟨`state, action`⟩ pairs. Used for the approx. Q-learning agent in `qlearningAgents.py`.
Files you might want to read but not edit:	
`mdp.py`	Defines methods for general MDPs.
`learningAgents.py`	Defines the base class `QLearningAgent`, which you will extend in your implementation.
`util.py`	Utilities, including `util.Counter`, which is particularly useful for designing Q-learning agents.

The remaining files are not as important and need not be reviewed.

[1] Note that this part of the project is based on ***UC Berkeley's CS188 Pac-Man Projects***. We have their permission to use this project. However, it should be noted that your solutions should **NOT** be distributed or published

Submission Specifications

For this project, you will need to submit one folder containing two files: `qlearningAgents.py` and `featureExtractors.py`.

Make only one submission (i.e., one set of two files) per group.

Do not modify the file name.

Place your two files in a folder named `CS3243_P3_XX`, where `XX` is your group number. For example, the folder `CS3243_P3_03/` should contain the `qlearningAgents.py` and `featureExtractors.py` files for Group 3. Then zip this folder up.

Points will be deducted for not following the naming convention, please follow it closely.

Task 1

First, you will implement a Q-learning agent by completing the `QLearningAgent` class in `qlearningAgents.py`. In particular, you need to implement the `update`, `computeValueFromQValues`, `getQValue`, `computeActionFromQValues` and `getAction` methods.

Note: for `computeActionFromQValues`, you might want to break ties randomly for better behavior (though again, this may need to be balanced for exploration). 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.

By calling the `random.choice` function, you may choose an element from a list uniformly at random. 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 $1 - p$.

Important: Ensure that you only access Q values by calling `getQValue`, especially within the `computeValueFromQValues` and `computeActionFromQValues` functions. This abstraction will be useful for Task 2, where you will have to override `getQValue` to use features of state-action pairs rather than the state-action pairs directly.

Testing: Time to play some Pacman! Pacman will play games in two phases. In the first phase, *training*, Pacman learns 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, it 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 its 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 defined based on the `QLearningAgent` you have written. `PacmanQAgent` is only different in that it has default learning parameters that are more effective for the Pacman problem ($\varepsilon = 0.05, \alpha = 0.2, \gamma = 0.8$). 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. To use the autograder to test your answer, run:

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

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 are accessible as `self.epsilon`, `self.gamma` and `self.alpha` within the agent. In the example run above, a total of 2010 games will be played, but 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 games. The number of training games is also passed to your agent as the option `numTraining`. 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 some statistics about Pacman's performance every 100 games. Epsilon is positive during training, so Pacman will play poorly even after having learned a good policy: this is because it occasionally makes a random exploratory move into a ghost. As a benchmark, it should take between 1000 and 1400 games before Pacman's rewards for a 100 episode segment becomes positive, reflecting that it'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, it should win very reliably in test games (at least 90% of the time), since now it is exploiting its learned policy. However, you will find that training the same agent on the seemingly simple `mediumGrid` does not work well.

In this implementation, Pacman's average training rewards remain negative throughout training. At test time, it plays badly, probably losing all of its 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. It has no way of generalizing that running into a ghost is bad for all positions. Obviously, this approach will not scale.

Task 2

In this task, you will implement an approximate Q-learning agent by completing the `ApproximateQAgent` class in `qlearningAgents.py`. It should learn weights for features of states, where many states might share the same features. In particular, you need to implement the `update` and `getQValue` methods.

Approximate Q-learning assumes the existence of a feature function $f(s,a)$ over state and action pairs, which yields a vector $(f_1(s,a), \ldots, f_i(s,a), \ldots, f_n(s,a))$ 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 in a manner that is similar to how you updated the Q-values:

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

Note that the difference term is the same as in normal Q-learning, where $r(s,a)$ is the experienced reward, and where $s' = \delta(s,a)$.

By default, `ApproximateQAgent` uses the `IdentityExtractor`, which assigns a single feature to every $\langle$`state, action`$\rangle$ 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 the provided simple 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.

For grading, your approximate Q-learning agent will be run and checked to determine if it learns the same Q-values and feature weights as our reference implementation (when each is presented with the same set of examples). To test your implementation, run the autograder:

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

Task 3

This final task requires you to implement your own feature extractor for the Pacman game. In particular, you will need to implement the NewExtractor class in featureExtractor.py.

This task will require you to be innovative. While there is no *correct* answer to this task, you should try your best to find the most suitable set of features in the Pacman world. We will grade you based on how well your agent performs.

Note: as we may alter the reward function during grading, please do not attempt to restrict your feature extractor to any specific reward function.

To play around with your implemented feature extractor, use the following command:

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

Please feel free to change the parameters when you are experimenting with your implementation.

Marking Rubrics

- Code for Pacman Solution (10 points)
  - □ Q-learning Implementation on standard Pacman problem - based on expected benchmark scores (3 points)
  - □ Approximate Q-learning Implementation on Standard Pacman problem - based on expected benchmark scores (3 points)
  - □ Solution for Mutated Pacman variant (2 points)
    - ■ Think of this as hidden test cases for task 1 and 2
    - ■ If your agent is correct, you should get these marks for free
    - ■ Requires a universally applicable agent
  - □ Solution for the variant with customised Features (2 point)

If not attempted or not working	0 points
Top 30% of cohort in average performance [2]	1 point
Top 70% of cohort in improvement [3]	2 points

For all tasks, the grades returned by the autograder will be re-scaled to the marks as reflected above in the computation of the scores.

[2] Note that this assumes that your implementation will improve on the features used in Tasks 1 and 2

[3] Note that this assumes that your implementation will improve on the features used in Tasks 1 and 2