Reinforcement Learning
Winter 2017
Assignment 3 - Approximate Q-learning

Note: this assignment is a slightly extended version of the assignment proposed here:

http://ai.berkeley.edu/reinforcement.html#Welcome

where we propose additional tasks, but also a significantly larger amount of coding hints.

Please have an environment-friendly policy, avoid printing these sheets !

1. **Q-Learning on the small Pacman game**

   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 a testing (game) 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.

   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 ($\varepsilon = 0.05$, $\alpha = 0.2$, $\gamma = 0.8$). 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: 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 always 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.

   The command above plays a total of 2010 games, 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.

   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).

   If you want to watch your agent training, use the command:

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

   If you want to run "proper statistics" on the performance of your agent, you can run the command (option -q

"mutes" the test games, allowing you to run fast statistics):

python pacman.py -p PacmanQAgent -x 2000 -n 2100 -l smallGrid -q

which will run 100 test games, and report the number of wins. Your agent should win 95%-100% of the time.

You can assess your code via:

python autograder.py -q q7

The autograder will run 100 test games after the 2000 training games.

If you want to further 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.

2. Try now your exact Q-learning agent on the medium grid by e.g. running

   python pacman.py -p PacmanQAgent -x 2000 -n 2100 -l mediumGrid -q

   What happens? Why? Can you fix this? What does it tell you about exact Q-learning?

3. **Approximate Q-Learning** The Q-learning we have developed so far builds a Q-function of the entire state space of the MDP. While doing this is feasible for small MDPs, i.e. small games, for even slightly larger games the complexity of learning a decent Q-function is prohibitive. This problem is often referred to as the "curse of dimensionality" in the context of Dynamic Programming.

   As detailed in the course a technique to circumvent the curse of dimensionality is to use low-dimensional representations of the MDP state, and work on this low-dimensional space instead. This approach is referred to as **function approximation** or **approximate Q-learning**. In this assignment, we will use a simple "feature-based" representation of the Q-value function, i.e. we will use:

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

   where $\phi_{1,\dots,n}(s,a)$ are given feature functions, and $w_{1,\dots,n}$ a set of weights that are to be learned. A set of feature functions are readily built-in for the Pacman environment, in the class **featureExtractors.py**. You can interrogate the class via

   Features = self.featExtractor.getFeatures(state,action)

   which will return a dictionary of features relevant for the current state and action, whose keys are features with associated values.

   You will write your approximate Q-learning agent in the file class **ApproximateQAgent** of the file **qlearningAgents.py**, where the functions you need to edit are similar to what you have done before. Because we are using a linear parametrization of the Q-function, our agent can learn the weights via the simple update rule:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s,a) \tag{2a}$$

$$w_i \leftarrow w_i + \alpha \delta \phi_i(s,a), \quad i = 1,\dots,n \tag{2b}$$

   where r is the reward measured on the transition from s to s'. You will have to work on the following functions:

   1. Method **getQValue** should return the value of the Q-value function for a given state and action

2. Method **update** is where the learning happens. It will tap into (some of) the methods you have developed in the previous assignments (namely computeValueFromQValues). Note that you don't even need to bother about what features are used in the agent, just make sure you implement the weighted sum (1) by sweeping through all the features relevant for your state.
3. Method **final** is useful to display your weights at the end of each game, for debugging purposes, though it is not mandatory to implement it for the code to run.

Important: ApproximateQAgent is a subclass of QLearningAgent, and it therefore shares several methods like getAction with the main class. 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.

You can start with testing your approximate Q-agent by using an Identity extractor, where the feature functions simply match the state-action pairs. In this case, your code should have the same behavior than the exact Q-learning. You can test this using (obs: if your code is **very slow**, then you have not been careful in your implementation!!):

python pacman.py -p ApproximateQAgent -x 2000 -n 2100 -l smallGrid -q (3)

which should give you a 90%-100% win.

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.

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

Your approximate Q-learning agent should win most of the time with these simple features, even with only 50 training games. Run your agent to get proper statistics with

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 150 -l mediumClassic -q

which would typically give you about 90% wins.

To validate your implementation, run the autograder:

python autograder.py -q q8

4. Let us return to the smallGrid. Run

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 150 -l smallGrid -q

What is your winning rate now (you may want to run the test a few times)? How does it compare to the previous case (3) ? Why (note: it can be useful to watch a handful of games and reflect on the Pacman losses to understand)? Can you close this gap? Why? What does it tell you about approximate Q-learning?

Questions

- Explore the feature function self.featExtractor.getFeatures(state,action). What features are proposed in your Pacman game? Is that reasonable? What limitations does that entail?
- Provide an interpretation of the update equation (2)

- Report on question 2
- Report on question 4