CS 214: Artificial Intelligence Lab
Spring 2022-23, IIT Dharwad
Assignment-8
Artificial Intelligence Lab

Instructions

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

As in previous projects, 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

See the autograder tutorial in Project 0 for more information about using the autograder.

The code for this project contains the following files, which are available in a zip archive:

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).

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.
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

Files to Edit and Submit: You will fill in portions of valueIterationAgents.py, qlearningAgents.py, and analysis.py during the assignment. You should submit these files with your code and comments. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

MDPs

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).

Question 1: Approximate Q-Learning (5 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.

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 wi is associated with a particular feature fi(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 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.

To grade your implementation, run the autograder:

python autograder.py -q q8

NOTE:

- Download *reinforcement.zip*, unzip and rename it with your *group number*.
- Write your code in respective files.
- Test your code and submit it on moodle.
- Due date for the Assignment is *7th April 2023 (11:59PM)*
- Penalty for late submission is *10%* of secured marks.
- We will run a plagiarism check for all the submissions, if found copied *100%* penalty will be applied.