

Project 5: Reinforcement Learning

Due: Thursday, 5/24/2018 (end of the day)

Introduction

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:

```
python3 autograder.py
```

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

```
python3 autograder.py -q q2
```

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

```
python3 autograder.py -t test_cases/q2/1-bridge-grid
```

You can download the starter code for this project from Nexus.

Files you'll edit (**and please do not edit any other files**):

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 might 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.

Files you can ignore: featureExtractors.py, environment.py, graphicsGridworldDisplay.py, graphicsUtils.py, textGridworldDisplay.py, crawler.py, graphicsCrawlerDisplay.py, autograder.py, testParser.py, testClasses.py, reinforcementTestClasses.py, and the directory test_cases/.

Acknowledgments This project was developed by John DeNero and Dan Klein at UC Berkeley.

Getting Started

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

```
python3 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:

```
python3 gridworld.py -h
```

The default agent moves randomly

```
python3 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: 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 . (Note: In class we have been using the term *utility* and the letter U for the same thing that here is called *value* and are referred to using the letter V .) In addition to running value iteration (in the `__init__` method), implement the following methods:

`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} (like we did in class), 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: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder:

```
python3 autograder.py -q q1
```

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.

```
python3 gridworld.py -a value -i 100 -k 10
```

Hint: On the default `BookGrid`, running value iteration for 5 iterations should give you this output:

```
python3 gridworld.py -a value -i 5
```

0.51 ▶	0.72 ▶	0.84 ▶	1.00
▲ 0.27		▲ 0.55	-1.00
▲ 0.00	0.22 ▶	▲ 0.37	◀ 0.13
VALUES AFTER 5 ITERATIONS			

Question 2: Bridge Crossing Analysis

BridgeGrid is a grid world map with the a low-reward terminal state and a high-reward terminal state separated by a narrow “bridge”, on either side of which is a chasm of high negative reward. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in `question2()` of `analysis.py`. (Noise refers to how often an agent ends up in an unintended successor state when they perform an action.) The default corresponds to:

```
python3 gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

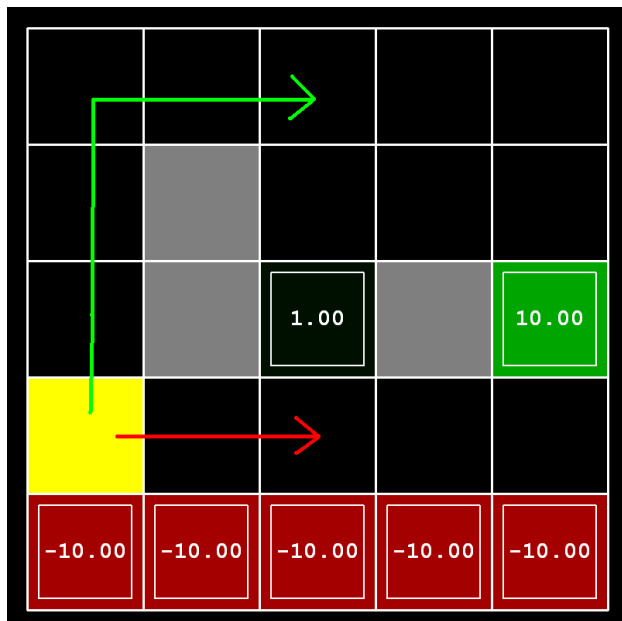


To check your answer, run the autograder:

```
python3 autograder.py -q q2
```

Question 3: 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. 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)

To check your answers, run the autograder:

```
python3 autograder.py -q q3
```

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.

Question 4: 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 Gridworld, but it's very important in the real world, where the real MDP is not available.

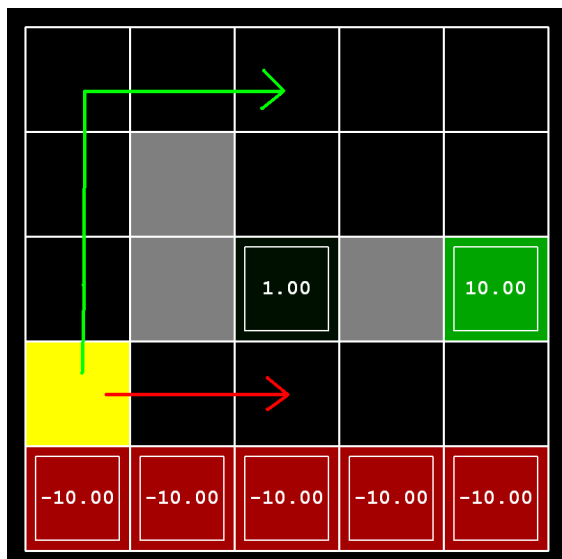
You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update(state, action, nextState, reward)` method. A stub of a Q-learner is specified in `QLearningAgent` in `qlearningAgents.py`, and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

Notes: 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.

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

```
python3 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:



To grade your implementation, run the autograder:

```
python3 autograder.py -q q4
```


Question 5: 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.

```
python3 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 $1 - p$.

To test your implementation, run the autograder:

```
python3 autograder.py -q q5
```

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

```
python3 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.

This will invoke a crawling robot using your Q-learner. 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.

Question 6: Bridge Crossing Revisited

First, train a completely random Q-learner with the default learning rate on the noiseless **BridgeGrid** for 50 episodes and observe whether it finds the optimal policy.

```
python3 gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```

Now try the same experiment with an epsilon of 0. Is there an epsilon and a learning rate for which it is highly likely (greater than 99%) that the optimal policy will be learned after 50 iterations? `question6()` in `analysis.py` should return EITHER a 2-item tuple of (epsilon, learning rate) OR the string 'NOT POSSIBLE' if there is none. Epsilon is controlled by `-e`, learning rate by `-l`.

Note: Your response should be not depend on the exact tie-breaking mechanism used to choose actions. This means your answer should be correct even if for instance we rotated the entire bridge grid world 90 degrees.

To grade your answer, run the autograder:

```
python3 autograder.py -q q6
```

Question 7: Q-Learning and Pacman

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. `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.

Note: To grade your answer, run:

```
python3 autograder.py -q q7
```

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:

```
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 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. To scale Q-learning you would need to represent states as feature vectors which can generalize the important aspects of a state.

How to Submit

Submit `valueIterationAgents.py`, `qlearningAgents.py`, and `analysis.py` on Nexus

Make sure all your code is commented.