# Project 5: Reinforcement Learning

This project is adapted from: <https://inst.eecs.berkeley.edu/~cs188/fa20/projects/>

[Introduction](#_fr5uiyacqiby)

[Question 1: Q-Learning](#_8wf2wjfcb088)

[Question 2: Epsilon Greedy](#_bu4ar76dowbe)

[Question 3: Bridge Crossing Revisited](#_fmb6k0fwks1e)

[Question 4: Q-Learning and Pacman](#_ld4qdvajlusr)

[Question 5: Approximate Q-Learning](#_9l8p7cua7l26)

[Turning in your work](#_x0ianu22gbzo)

## Introduction

This project continues P4 by implementing reinforcement learning. In the end, you’ll have a PacMan agent that learns how to play well!

The code for this project contains the following files, available as a zip archive.

| **Files you'll edit:** | |
| --- | --- |
| 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. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

**Evaluation:** Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder’s judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

**Academic Dishonesty:** We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else’s code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don’t try. We trust you all to submit your own work only; *please* don’t let us down. If you do, we will pursue the strongest consequences available to us.

**Getting Help:** You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours, section, and the discussion forum are there for your support; please use them. If you can’t make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don’t know when or how to help unless you ask.

## Question 1: Q-Learning

Note that your value iteration agent from the previous project 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 Gridword, 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.

*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.

*Important:* Make sure that in your computeValueFromQValues and computeActionFromQValues functions, you only access Q values by calling getQValue . This abstraction will be useful for question 5 when you override getQValue to use features of state-action pairs rather than state-action pairs directly.

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

python 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:



*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 the autograder:

python autograder.py -q q1

## Question 2: 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.

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.

After implementing the getAction method, observe the following behavior of the agent in gridworld (with epsilon = 0.3).

python 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 predicted because of the random actions and the initial learning phase.

You can also observe the following simulations for different epsilon values. Does that behavior of the agent match what you expect?

python gridworld.py -a q -k 100 --noise 0.0 -e 0.1

python gridworld.py -a q -k 100 --noise 0.0 -e 0.9

To test your implementation, run the autograder:

python autograder.py -q q2

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

python 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 the crawling robot from class 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 3: 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.

python 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? **question8**() 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 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:

python autograder.py -q q3

## Question 4: 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:

python 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. Beware of the argmax function from util.Counter!

*Note:* To grade your answer, run:

python autograder.py -q q4

*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:

python 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 1000 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 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.

## Question 5: Approximate Q-Learning

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 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)=fi(s,a)wi

where each weight wiwi is associated with a particular feature fi(s,a)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:

wi←wi+α⋅difference⋅fi(s,a)

difference = (r+γ⋅maxa′ 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.

*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 the autograder:

python autograder.py -q q5

***Congratulations! You have a learning Pacman agent!***

## Turning in your work

1. Run the autograder and save the output in a file called grade.txt.

On a linux/mac you can do this with:

python autograder.py > grade.txt

Otherwise, you can just copy the autograder output to a **plain** text file and save it.

1. Zip the whole p5 directory into a file called p5.zip. On a linux/mac:

zip -r p5.zip p5 (the -r will recursively zip subdirectories).

1. Upload p5.zip to Canvas under P1.