CS7646 ML4T
Machine Learning
for Trading

PROJECT 7: Q-LEARNING ROBOT

DUE DATE

11/01/2020 11:59PM Anywhere on Earth time

REVISIONS

This assignment is subject to change up until 3 weeks prior to the due date. We do not anticipate changes; any changes will be logged in this section.

OVERVIEW

This assigment counts towards 10% of your overall grade.

In this project you will implement the Q-Learning and Dyna-Q solutions to the reinforcement learning problem. You will apply them to a navigation problem in this project. In a later project you will apply them to trading. The reason for working with the navigation problem first is that, as you will see, navigation is an easy problem to work with and understand. Note that your Q-Learning code really shouldn't care which problem it is solving. The difference is that you need to wrap the learner in different code that frames the problem for the learner as necessary.

For this project we have created testqlearner.py that automates testing of your Q-Learner in the navigation problem.

Overall, your tasks for this project include:

- Code a Q-Learner
- Code the Dyna-Q feature of Q-Learning
- Test/debug the Q-Learner in navigation problems

For this assignment we will test only your code (there is no report component).

TEMPLATE

- Download the template code here: File:QLearning_Robot_2020Fall.zip
- Implement the `QLearner` class in `qlearning_robot/QLearner.py`.
- To debug your Q-learner, run **`python testqlearner.py`** from the `qlearning_robot/` directory. The grading script for this project is **`grade_robot_qlearning.py`**.
- Note that example navigation problems are provided in the `qlearning_robot/testworlds` directory.

HINTS & RESOURCES

This paper by Kaelbling, Littman and Moore, is a good resource for RL in general: https://arxiv.org/pdf/cs/9605103.pdf See Section 4.2 for details on Q-Learning.

There is also a chapter in the Mitchell book on Q-Learning.

For implementing Dyna, you may find the following resources useful:

- http://incompleteideas.net/sutton/book/RLbook2018.pdf (Section 8.2)
- http://www-anw.cs.umass.edu/~barto/courses/cs687/Chapter%209.pdf
- https://arxiv.org/pdf/1712.01275.pdf

TASKS

Implement Q-Learner (95 points)

Your QLearner class should be implemented in the file `QLearner.py`. It should implement EXACTLY the API defined below. DO NOT import any modules besides those allowed below. Your class should implement the following methods:

The constructor QLearner() should reserve space for keeping track of Q[s, a] for the number of states and actions. It should initialize Q[] with all zeros. Details on the input arguments to the constructor:

- `num_states` integer, the number of states to consider
- `num_actions` integer, the number of actions available.
- `alpha` float, the learning rate used in the update rule. Should range between 0.0 and 1.0 with 0.2 as a typical value.

- `gamma` float, the discount rate used in the update rule. Should range between 0.0 and 1.0 with 0.9 as a typical value.
- `rar` float, random action rate: the probability of selecting a random action at each step. Should range between 0.0 (no random actions) to 1.0 (always random action) with 0.5 as a typical value.
- `radr` float, random action decay rate, after each update, rar = rar * radr. Ranges between 0.0 (immediate decay to 0) and 1.0 (no decay). Typically 0.99.
- `dyna` integer, number of dyna updates for each regular update. When Dyna is used, 200 is a typical value.
- `verbose` boolean, if True, your class is allowed to print debugging statements, if False, all printing is prohibited.

query(s_prime, r) is the core method of the Q-Learner. It should keep track of the last state s and the last action a, then use the new information s_prime and r to update the Q table. The learning instance, or experience tuple is <s, a, s_prime, r>. query() should return an integer, which is the next action to take. Note that it should choose a random action with probability rar, and that it should update rar according to the decay rate radr at each step. Details on the arguments:

- `s_prime` integer, the the new state.
- `r` float, a real valued immediate reward.

querysetstate(s) A special version of the query method that sets the state to s, and returns an integer action according to the same rules as query() (including choosing a random action sometimes), but it does not execute an update to the Q-table. It also does not update rar. There are two main uses for this method: 1) To set the initial state, and 2) when using a learned policy, but not updating it.

Here's an example of the API in use:

```
import QLearner as ql

learner = ql.QLearner(num_states=100,
                      num_actions=4,
                      alpha=0.2,
                      gamma=0.9,
                      rar=0.98,
                      radr=0.999,
                      dyna=0,
                      verbose=False)
```

```
s = 99 # our initial state

a = learner.querysetstate(s) # action for state s

s_prime = 5 # the new state we end up in after taking action a in state s

r = 0 # reward for taking action a in state s

next_action = learner.query(s_prime, r)
```

Navigation Problem Test Cases

We will test your Q-Learner with a navigation problem as follows. Note that your Q-Learner does not need to be coded specially for this task. In fact, the code doesn't need to know anything about it. The code necessary to test your learner with this navigation task is implemented in testqlearner.py for you. The navigation task takes place in a 10 x 10 grid world. The particular environment is expressed in a CSV file of integers, where the value in each position is interpreted as follows:

- 0: blank space.
- 1: an obstacle.
- 2: the starting location for the robot.
- 3: the goal location.
- 5: quicksand.

An example navigation problem (world01.csv) is shown below. Following python conventions, [0,0] is upper left, or northwest corner, [9,9] lower right or southeast corner. Rows are north/south, columns are east/west.

```
3,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,1,1,1,1,1,0,0,0
0,5,1,0,0,0,1,0,0,0
0,5,1,0,0,0,1,0,0,0
0,0,1,0,0,0,1,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,2,0,0,0,0,0
```

In this example the robot starts at the bottom center, and must navigate to the top left. Note that a wall of obstacles blocks its path, and there is some quicksand along the left side. The objective is for the robot to learn how to navigate from the starting location to the goal with the highest total reward. We define the reward for each step as:

- -1 if the robot moves to an empty or blank space, or attempts to move into a wall
- -100 if the robot moves to a quicksand space
- 1 if the robot moves to the goal space

Overall, we will assess the performance of a policy as the median reward it incurs to travel from the start to the goal (higher reward is better). We assess a learner in terms of the reward it converges to over a given number of training epochs (trips from start to goal). **Important note:** the problem includes random actions. So, for example, if your learner responds with a “move north” action, there is some probability that the robot will actually move in a different direction. For this reason, the “wise” learner develops policies that keep the robot well away from quicksand. We map this problem to a reinforcement learning problem as follows:

- State: The state is the location of the robot, it is computed (discretized) as: column location * 10 + row location.
- Actions: There are 4 possible actions, 0: move north, 1: move east, 2: move south, 3: move west.
- R: The reward is as described above.
- T: The transition matrix can be inferred from the CSV map and the actions.

Note that R and T are not known by or available to the learner. The code in `testqlearner.py` will test your code as follows (pseudo code):

```
Instantiate the learner with the constructor QLearner()
s = initial_location
a = querysetstate(s)
s_prime = new location according to action a
r = -1.0
while not converged:
    a = query(s_prime, r)
    s_prime = new location according to action a
    if s_prime == goal:
        r = +1
        s_prime = start location
    else if s_prime == quicksand:
        r = -100
    else:
        r = -1
```

A few things to note about this code: The learner always receives a reward of -1.0 (or -100.0) until it reaches the goal, when it receives a reward of +1.0. As soon as the robot reaches the goal, it is immediately returned to the starting location.

Here are example solutions. Note that these examples were created before we added "quicksand" to the project. We will be updating the examples to reflect this change. In the meantime, you may find these useful:

mc3_p2_examples

mc3_p2_dyna_examples

Implement Dyna (5 points)

Add additional components to your QLearner class so that multiple "hallucinated" experience tuples are used to update the Q-table for each "real" experience. The addition of this component should speed convergence in terms of the number of calls to query(), **not necessarily running time**.

Note that it is not important that you implement Dyna exactly as described in the lecture. The key requirement is that your code should somehow hallucinate additional experiences. The precise method you use for discovering those experiences is flexible. We will test your code on several test worlds with 50 epochs and with dyna = 200. Our expectation is that with Dyna, the solution should be much better after 50 epochs than without.

Implement author() Method (up to 20 point penalty)

You should implement a method called `author()` that returns your Georgia Tech user ID as a string. This is the ID you use to log into canvas. It is not your 9 digit student number. Here is an example of how you might implement author() within a learner object:

```
class QLearner(object):
    def author(self):
        return 'tb34' # replace tb34 with your Georgia Tech username.
```

And here's an example of how it could be called from a testing program:

```
# create a learner and train it
learner = ql.QLearner() # create a QLearner
print(learner.author())
```

Implementing this method correctly does not provide any points, but there will be a penalty for not implementing it.

WHAT TO TURN IN

Be sure to follow these instructions diligently!

Gradescope:

- (SUBMISSION) Project 7: QLearning Robot
  - Your code as `QLearner.py`
  - Do not submit any other files.

You are only allowed 3 submissions to **(SUBMISSION) Project 7: QLearning Robot** but unlimited resubmissions are allowed on **(TESTING) Project 7: QLearning Robot.**

Note that Gradescope does **not** grade your assignment live; instead, it pre-validates that it will run against our batch autograder that we will run after the deadline. There will be **no** credit given for coding assignments that do not pass this pre-validation.

Refer to the Gradescope Instructions for more information.

RUBRIC

Report

No report

Code

- Is the author() method correctly implemented (-20 if not)

Auto-Grader

- For basic Q-Learning (dyna = 0) we will test your learner against 10 test worlds with 500 epochs in each world. One "epoch" means your robot reaches the goal one time, or after 10000 steps, whichever comes first. Your QLearner retains its state (Q-table), and then we allow it to navigate to the goal again, over and over, 500 times. Each test (500 epochs) should complete in less than 2 seconds. **NOTE**: an epoch where the robot fails to reach the goal will likely take **much** longer (in running time), than one that does reach the goal, and is a common reason for failing to complete test cases within the time limit.
- Benchmark: As a benchmark to compare your solution to, we will run our reference solution in the same world, with 500 epochs. We will take the median reward of our reference across all of those 500 epochs.
- Your score: For each world we will take the median cost your solution finds across all 500 epochs.
- For a test to be successful, your learner should find a total reward >= 1.5 x the benchmark. Note that since reward for a single epoch is negative, your solution can be up to 50% worse than the reference solution and still pass.
- There are 10 test cases, each test case is worth 9.5 points.
- Here is how we will initialize your QLearner for these test cases:

```
learner = ql.QLearner(num_states=100,
                      num_actions=4,
                      alpha=0.2,
                      gamma=0.9,
                      rar=0.98,
                      radr=0.999,
                      dyna=0,
                      verbose=False)  # initialize the learner
```

- For Dyna-Q, we will set dyna = 200. We will test your learner against `world01.csv` and `world02.csv` with 50 epochs. Scoring is similar to the non-dyna case: Each test should complete in less than 10 seconds. For the test to be successful, your learner should find solution with total reward to the goal >= 1.5 x the median reward our reference solution across all 50 epochs. Note that since reward for a single epoch is negative, your solution can be up to 50% worse than the reference solution and still pass. We will check this by taking the median of all 50 runs. Each test case is worth 2.5 points. We will initialize your learner with the following parameter values for these test cases:

```
learner = ql.QLearner(num_states=100,
                      num_actions=4,
                      alpha=0.2,
                      gamma=0.9,
                      rar=0.5,
                      radr=0.99,
                      dyna=200,
                      verbose=False)  # initialize the learner
```

REQUIRED, ALLOWED & PROHIBITED

Required:

- Your project must be coded in Python 3.6.x.
- Your code must be submitted to Gradescope in the appropriate Gradescope assignment.

Allowed:

- You can develop your code on your personal machine, but it must also run successfully on Gradescope.
- Your code may use standard Python libraries.
- You may use the NumPy, SciPy, matplotlib and Pandas libraries. Be sure you are using the correct versions.
- Code provided by the instructor or allowed by the instructor to be shared.

Prohibited:

- Any libraries not listed in the “allowed” section above.
- Any code you did not write yourself.
- Any Classes (other than Random) that create their own instance variables for later use (e.g., learners like kdtree).
- Any use of plot.show()
- Absolute import statements of the **current** project folder such as `from qlearning_robot import XXXX or import qlearning_robot.XXXX`
- Extra directories (manually or code created)
- Extra files not listed in “WHAT TO TURN IN”
- Print statements outside “verbose” checks (they significantly slow down auto grading).
- Any method for reading data besides util.py