MC3-Project-3

From Quantitative Analysis Software Courses

Contents

Updates / FAQs

Overview

In this project you will implement and assess Q-Learning. Because of the limited time available for this project, we're going to have you first test your Q-Learning implementation to solve a navigation problem. Applying Q-Learning to stock trading is offered as an extra credit assignment. Note that your Q-Learning code really shouldn't care which problem it is solving, but in order to apply it to trading, you will have to re-work the testqlearner.py code.

Template and Data

- Download **`mc3_p3.zip`**, unzip inside `ml4t/`
- Implement the `QLearner` class in `mc3_p3/QLearner.py`.
- To test your learner, run **`python testqlearner.py`** from the `mc3_p3/` directory.
- Note that example problems are provided in the `mc3_p3/testworlds` directory

Part 1: Implement QLearner (90%)

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:

- QLearner(...): Constructor, see argument details below.
- query(s_prime, r): Update Q-table with <s, a, s_prime, r> and return new action for state s_prime.
- querysetstate(s): Set state to s, return action for state s, but don't update Q-table.

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)
```

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 uniform random values between -1.0 and 1.0. 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, conduct this 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(), but it does not execute an update to the Q-table. This method is typically only used once, to set the initial state.

The Navigation Problem

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.

An example navigation problem (CSV file) 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.

```
0,0,0,0,3,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,1,1,1,1,1,0,0,0
0,0,1,0,0,0,1,0,0,0
0,0,1,0,0,0,1,0,0,0
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 center. Note that a wall of obstacles blocks its path. 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: south, 3: move west.
- R: The reward is -1.0 unless the action leads to the goal, in which case the reward is +1.0.
- 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 testing code `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
        r = -1
```

A few things to note about this code: The learner always receives a reward of -1.0 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.

Part 2: Implement Dyna (10%)

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().

Contents of Report

There is no report component of this assignment.

Example Solutions

mc3_p3_examples

mc3_p3_dyna_examples

Hints & resources

This paper by Kaelbling, Littman and Moore, is a good resource for RL in general: http://www.jair.org/media/301/live-301-1562-jair.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:

- https://webdocs.cs.ualberta.ca/~sutton/book/ebook/node96.html
- http://www-anw.cs.umass.edu/~barto/courses/cs687/Chapter%209.pdf

What to turn in

Turn your project in via t-square.

- Your code as `QLearner.py`

Extra credit up to 10%

Revise testqlearner.py so that it applies your learner to the task of stock trading. Demonstrate it's efficacy with compelling charts. Summarize your results in a document of no more than 6 pages. Submit your document `report.pdf` to the separate extra credit assignment on t-square. Note that the extra credit component will not be considered unless your regular project completes the test cases perfectly.

Rubric

Only your QLearner class will be tested.

- For basic Q-Learning (dyna=0). We will test your learner against 10 test worlds with 500 iterations. Each test should complete in less than 2 seconds. For the test to be successful, your learner should find a path to the goal <= 1.5 x the number of steps our reference solution finds. We will check this by taking the min of all the 500 runs. Each test case is worth 9 points. We will initialize your learner with the following parameter values:

```
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 = 100. We will test your learner against 2 test worlds with 50 iterations. Each test should complete in less than 10 seconds. For the test to be successful, your learner should find a path to the goal <= 1.5 x the number of steps our reference solution finds. We will check this by taking the min of all 50 runs. Each test case is worth 5 points. We will initialize your learner with the following parameter values:

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

Required, Allowed & Prohibited

Required:

- Your project must be coded in Python 2.7.x.
- Your code must run on one of the university-provided computers (e.g. buffet02.cc.gatech.edu), or on one of the provided virtual images.

Allowed:

- You can develop your code on your personal machine, but it must also run successfully on one of the university provided machines or virtual images.
- 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.
- You may reuse sections of code (up to 5 lines) that you collected from other students or the internet.
- 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 (except for the 5 line rule in the "allowed" section).
- Any Classes (other than Random) that create their own instance variables for later use (e.g., learners like kdtree).
- Print statements (they significantly slow down auto grading).