Machine Learning
for Trading

PROJECT 7: Q-LEARNING ROBOT

Table of Contents

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.

1 OVERVIEW

In this assignment, you implement a Reinforcement Learning algorithm called Q-learning, which is a model-free RL algorithm. You will also extend your Q-learner implementation by adding a Dyna, model-based, component. You will submit the code for the project in Gradescope SUBMISSION. There is no report associated with this assignment.

1.1 Learning Objectives

The specific learning objectives for this assignment are focused on the following areas:

- **Q-Learning (Model-free RL algorithm)**: Develop a learner to “reinforce” an understanding of the Q-learning model-free algorithm.
- **Dyna-Q (Model-based addition to Q-learner)**: Develop a Dyna-Q implementation to “reinforce” an understanding of the Dyan-Q model-based addition to the Q-learner implementation.

2 ABOUT THE PROJECT

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 a different code that frames the problem for the learner as necessary.

3 YOUR IMPLEMENTATION

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

Overall, your tasks for this project include:

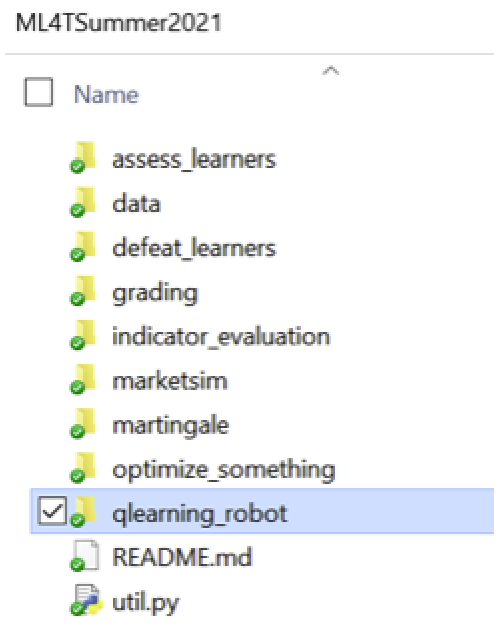
- Code a Q-Learner
- Code the Dyna-Q feature of Q-Learning
- Test/debug the Q-Learner in navigation problems

You must write your own code for this project. You are NOT allowed to use other people’s code or packages to implement the Q-learner. For this assignment, we will test only your code (there is no report component).

Before the deadline, make sure to pre-validate your submission using Gradescope TESTING. Once you are satisfied with the results in testing, submit the code to Gradescope SUBMISSION. **Only code submitted to Gradescope SUBMISSION will be graded. If you submit your code to Gradescope TESTING and have not also submitted your code to Gradescope SUBMISSION, you will receive a zero (0).**

3.1 Getting Started

To make it easier to get started on the project and focus on the concepts involved, you will be given a starter framework. This framework assumes you have already set up the local environment and ML4T Software. The framework for Project 7 can be obtained from: QLearning_Robot_2022Summer.zip.


ML4TSummer2021

Name

assess_learners
data
defeat_learners
grading
indicator_evaluation
marketsim
martingale
optimize_something
qlearning_robot
README.md
util.py

Extract its contents into the base directory (e.g., ML4T_2022Summer). This will add a new folder called “qlearning_robot” to the course directory structure:

The framework for Project 7 can be obtained in the qlearning_robot folder alone. Within the qlearning_robot folder are several files:

- QLearner.py
- testqlearner.py
- grade_robot_qlearning.py

Note: Example navigation problems are provided in the qlearning_robot/testworlds directory.

Your q-learning class will be implemented in the QLearner.py file. The testqlearner.py file contains a simple testing scaffold that you can use to test your learners, which is useful for debugging. It must also be modified to run the experiments. The grade_robot_qlearners.py file is a local pre-validation script that mirrors the script used in the Gradescope TESTING environment.

3.2 Task & Requirements

You will implement the following files:

- QLearner.py – Contains the code for the Q-Learner (and Dyna-Q) implementation

All your learner code must be placed into the file above. No other code files will be accepted. The testqlearner.py file that is used to conduct your experiments must also reside in the qlearning_robot and is run using the following command:

```
PYTHONPATH=../:. python testqlearner.py
```

test_qlearner hosted with ❤ by **GitHub** **view raw**

3.3 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. This is useful when debugging in a local environment.

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 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.

- **s** integer, a state

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:

```python
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)
```

qlearner_api.py hosted with ❤ by GitHub view raw

3.4 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.

3.5 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:

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

qlearner_author.py hosted with ❤ by **GitHub** **view raw**

3.6 Navigation Problem Test Cases

Note*: Understanding how testing will be performed may facilitate coding. So, we are placing this section here rather than in the “testing” section below. The standard testing text will be retained in the testing section below.*

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
```

navigation_example hosted with ❤ by **GitHub** **view raw**

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: row location * 10 + column 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
```

QLearner_pseudocode hosted with ❤ by GitHub view raw

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.

3.6.1 Additional Example Solutions

Here are example solutions. Note that these examples were created before we added “quicksand” to the project. In the future, 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

3.7 Technical Requirements

The following technical requirements apply to this assignment:

1. The value of Dyna cannot be "creatively" decreased (i.e., one cannot ignore the first few iterations (e.g.., ignore the first 5 Dyna steps) or take only the nth iteration (e.g., every 10 Dyna cycles) to perform the Dyna steps).

1. Each test (500 epochs, see rubric below) should complete in less than 2 seconds.

3.8 Hints and 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

If after submitting the project for grading you are not entirely satisfied with the implementation, you are encouraged to continue to improve the Q-learner as it can play a role in a future project (project 8).

4 CONTENTS OF REPORT

There is no report associated with this assignment.

5 TESTING RECOMMENDATIONS

To test your code, we will invoke each of the functions. You are encouraged to perform any tests necessary to instill confidence that the code will run properly when submitted for grading and will produce the required results. You should confirm that testqlearner.py runs as expected from the qlearning_robot.

Additionally, we provide the grade_robot_qlearning.py file that can be used for lightweight testing. This local grading/pre-validation script is the same script that will be run when the code is submitted to GradeScope TESTING. To run and test that the file will run from within the qlearning_robot directory, use the command:

```
PYTHONPATH=../:. python grade_robot_qlearning.py
```

grade_robot_qlearner hosted with ❤ by GitHub view raw

In addition to testing on your local machine, you are encouraged to submit your file to Gradescope TESTING, where some basic pre-validation tests will be performed against the code. No credit will be given for coding assignments that do not pass this pre-validation. **Gradescope TESTING does not grade your assignment.** The Gradescope TESTING script is not a complete test suite and does not match the more stringent private grader that is used in Gradescope SUBMISSION. Thus, the maximum Gradescope TESTING score of 81, while instructional, does not represent the minimum score one can expect when the assignment is graded using the private grading script. You are encouraged to develop additional tests to ensure that all project requirements are met.

You are allowed **unlimited** resubmissions to Gradescope **TESTING**. Please refer to the Gradescope Instructions for more information.

6 SUBMISSION REQUIREMENTS

This is an individual assignment. All work you submit should be your own. Make sure to cite any sources you reference and use quotes and in-line citations to mark any direct quotes.

Assignment due dates in your time zone can be found by looking at the Project in the Assignment menu item in Canvas (ensure your Canvas time zone settings are set up properly). This date is 23:59 AOE converted to your time zone. Late submissions are allowed for a penalty. The times and penalties are as follows:

- -10% Late Penalty: +1 Hour late: submitted by 00:59 AOE (next day)
- -25% Late Penalty: +12 Hours Late: submitted by 11:59 AOE (next day)
- -50% Late Penalty: +24 Hours Late: submitted by 23:59 AOE (next day)
- -100% Late Penalty: > 24+ Late: submitted after 23:59 AOE (next day)

Assignments received after Monday at 23:59 AOE (even if only by a few seconds) are not accepted without advanced agreement except in cases of medical or family emergencies. In the case of such an emergency, please contact the Dean of Students.

6.1 Report Submission

There is no report associated with this assignment.

6.2 Code Submission

This class uses Gradescope, a server-side auto-grader, to evaluate your code submission. No credit will be given for code that does not run in this environment and students are encouraged to leverage Gradescope TESTING prior to submitting an assignment for grading. **Only code submitted to Gradescope SUBMISSION will be graded. If you submit your code to Gradescope TESTING and have not also submitted your code to Gradescope SUBMISSION, you will receive a zero (0).**

Please submit the following file to Gradescope **SUBMISSION**:

QLearner.py

Do not submit any other files.

Important: You are allowed a MAXIMUM of three (3) code submissions to Gradescope SUBMISSION.

7 GRADING INFORMATION

The submitted code (which is worth 100% of your grade) is run as a batch job after the project deadline. The code will be graded using a 100-point scale coinciding with a rubric design to mirror the implementation details above. Deductions will be applied for unmet implementation requirements or code that fails to run.

We do not provide an explicit set timeline for returning grades, except that all assignments and exams will be graded before the institute deadline (end of the term). As will be the case throughout the term, the grading team will work as quickly as possible to provide project feedback and grades.

Once grades are released, any grade-related matters must follow the Assignment Follow-Up guidelines and process alone. Regrading will only be undertaken in cases where there has been a genuine error or misunderstanding. Please note that requests will be denied if they are not submitted using the Summer 2022 form or do not fall within the timeframes specified on the Assignment Follow-Up page.

7.1 Grading Rubric

7.1.1 Report [0 points]

There is no report associated with this assignment.

7.1.2 Code

Code deductions will be applied if any of the following occur:

- If the author() method is not correctly implemented in the QLearner file: (-20 points)

7.1.3 Auto-Grader (Private Grading Script) [100 points]

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 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 the 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
```

QLearner_initialization hosted with ❤ by **GitHub** **view raw**

- 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 a solution with total reward to the goal >= 1.5 x the median reward our reference solution across all 50 epochs. Note that since the 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
```

dyna_initialization hosted with ❤ by **GitHub** **view raw**

8 DEVELOPMENT GUIDELINES (ALLOWED & PROHIBITED)

See the Course Development Recommendations, Guidelines, and Rules for the complete list of requirements applicable to all course assignments. **The Project Technical Requirements are grouped into three sections: Always Allowed, Prohibited with Some Exceptions, and Always Prohibited**.

The following exemptions to the Course Development Recommendations, Guidelines, and Rules apply to this project:

- N/A

9 OPTIONAL RESOURCES

Although the use of these or other resources is not required; some may find them useful in completing the project or in providing an in-depth discussion of the material.

Videos:

- Reinforcement Learning (David Silver Video Lectures) – Videos and PowerPoint presentations.
- CS7642 Reinforcement Learning (Georgia Tech OMSCS Course Videos)
- CS234 Reinforcement Learning (Stanford University Course Videos)