Yale University
CPSC 474/574 - Computational Intelligence for Games
Fall 2020

Assignment 5 - Q-Learning for NFL Strategy

Objectives

- to implement a model-free, bootstrapping learning algorithm

Introduction

NFL Strategy is a football (North American gridiron football) simulation game from the 1970s. The player on offense chooses one of 34 plays; the player on defense simultaneously chooses one of 12 defensive plays. The outcome of the play is determined by the combination of plays selected and an element of chance.

We seek strategies for a simulation of a situation in which the team currently on offense (the Patriots) must score a touchdown to win the game against the team currently on defense (the Rams): there are 2 minutes (120 seconds) remaining in the game and the Patriots must gain 80 yards in order to score a touchdown and win the game. We make the simplifying assumptions that neither teams has any time-outs left, that any turnover, safety, or field goal leads to a win for the Rams (we assume the Patriots will not get the ball back after such situations), that a touchdown wins the game for the Patriots no matter how much time is remaining (the Rams will not be able to regain the lead against the Patriots defense), and that there will be no penalties. An offense in such a situation is said to be executing a "two-minute drill". To simulate the quick decision making that happens during such a hurry-up situation, we limit the offense and defense to a few plays each.

Assignment

Write a function called `q_learn` in a module called `qfl` that takes an object that models the Two-Minute Drill version of NFL strategy and a time limit in seconds and returns a function that takes a non-terminal position in the game and returns the index of the selected offensive play. The function that it returns should execute in less than 10 microseconds for any position. Because there are too many positions to compute a Q-value for each individually within the time bound, you should use a function approximator such as a linear approximator.

Your `q_learn` function can observe outcomes from a particular position given a choice of action (offensive play) by calling the `result` method on the model object passed to `q_learn` module. The `result` method takes a position and action and returns a pair whose first component is the resulting position. Positions are given as 4-tuples with components for the remaining yards needed to score, the downs left, the yards needed to reset the number of downs left, and the time remaining in 5-second ticks. The second component of the pair returned from `result` is a triple specifying the number of yards gained by that action, the time elapsed after that action (in 5-second ticks), and a Boolean flag indicating whether the action resulted in a turnover (in which case the game is over and the offense loses). That second component can be useful if you want to shape rewards.

The model object will also have some additional methods to allow you to simulate games. You may only access methods whose names do not begin with underscores. These methods are:

- the `result` method;
- the `initial_position` method, which returns the initial position of the game;
- the `offensive_playbook_size` method, which returns the number of offensive plays to choose from and the plays are then numbered 0, 1, ..., `model.offensive_playbook_size() - 1` (there is also a `defensive_playbook_size` method, but you should not need to use that method);

- the `game_over` method, which takes a position and determines if it is a terminal position;
- the `win` method, which takes a terminal position and determines whether the offense won; and
- the `simulate` method, which takes two arguments: 1) an offensive policy function – -- a function that takes a position and returns the index of the offensive play to choose in that position; and 2) a positive integer for the number of games to simulate using that policy. The method returns the winning percentage achieved over those games by the policy.

So for the standard Q-learning algorithm

- for as many games as you can simulate in the allotted time
  - $s \leftarrow s_0$
  - while s is not terminal
    - choose an action a
    - observe the transition (s, a, s', r)
    - update $Q(s, a)$

the `initial_position` method returns s_0, the `offensive_playbook_size` methods determines what actions are available to choose as a, the `result` method returns s' given s and a, and the `game_over` and `win` methods allow you to determine r.

Grading

The test driver `test_qfl` in `/c/cs474/hw5/Required` defines two models that choose defensive plays uniformly randomly. Your `q_learn` function will be tested on other models as well.

You will be graded on how well the policy returned by `q_learn` performs. Better policies will receive better grades, and the more consistently your Q-learning process can return good policies, the better your grade will be. You should be able to achieve a better winning percentage better than the best policy that always chooses the same play (about 0.38 for model 0

[always choosing play 2] and 0.285 for model 1 [always choosing play 2]) or always chooses randomly (0.275 and 0.105 for models 0 and 1 respectively). For comparison, an offensive player optimized against a random defensive player scores about 0.53 for model 0 and 0.48 for model 1.

Additional Requirements

- Tests will be executed with `pypy3`, so write your code for Python 3.5 and only use modules are available for `pypy3` on the Zoo (this excludes `numpy`).

Examples

```
[jrg94@cricket code]$ pypy3 test_qfl.py 0 9 250000
0.509548
[jrg94@cricket code]$ pypy3 test_qfl.py 1 9 250000
0.467852
```

Submissions

Submit just your `qfl.py` module along with any other supporting modules and your log. There is no need to submit a makefile or executable; the test scripts create an executable called `QFL` that is used to run the test drivers.