Homework 3

Assigned 2/7/2017 Due 2/21/2017 11:59PM
Extra Credit #1 2/14/2017 Extra Credit #2 2/19/2017

This homework contains both a machine problem and a written component.

Extra Credit #1 (due 2/14/2017)
Code for the MP + First 2 written questions

Extra Credit #2 (due 2/19/2017)
Code for the MP + All written questions (similar to the extra credit of the previous assignments)

In 1972, the game of Pong was released by Atari. Despite its simplicity, it was a ground-breaking game in its day, and it has been credited as having helped launch the video game industry. In this assignment, you will construct and test an ε-greedy Q-learning reinforcement agent on a simple single-player version of Pong by interacting with the Pong environment.

We have provided you with a skeleton Pong environment in Python. You are free to reuse the code, use that logic in some other language, or even start from scratch in a language of your choice. If you choose to use the provided code, based on your implementation you will need to add/remove function parameters, change their values, and add/remove functions. The code provided is a very basic boilerplate of what we think would help you get started.

The objective is to bounce the ball off the paddle as many times as possible. You must investigate its behavior under a selection of different learning parameters: the learning rate α, the discount factor γ, and the exploration parameter ε. Your initial investigation should use α = 0.4, γ = 0.95, ε = 0.04. Your investigation can try any number of alternative parameter values but must try at least three other sets of parameters and each parameter should be varied at least once. For each set of parameters, train as long as you deem necessary. Your best agent should rebound the ball at least nine times before missing it, although your results may be significantly better than this (12-14 or higher).

The Environment for Single Player Pong

The environment will supply your agent with the current state and any reward. Your agent will then select an action to perform.

A state is a 5-tuple (ball_x, ball_y, velocity_x, velocity_y, paddle_y).

- ball_x and ball_y are real numbers on the interval [0,1]. The lines x=0, y=0, and y=1 are walls; the ball bounces off a wall whenever it hits. The line x=1 is defended by your paddle.

- The absolute value of velocity_x is at least 0.03, guaranteeing that the ball is moving either left or right at a reasonable speed.
- paddle_y represents the top of the paddle and is on the interval [0, 1 - paddle_height], where paddle_height = 0.2 (The x-coordinate of the paddle is always paddle_x=1, and so it is not included in the state).

There are three possible actions: do nothing, paddle_y += 0.04, paddle_y -= 0.04. In other words, your agent can either move the paddle up, down, or make it stay in the same place. If the agent tries to move the paddle too high, so that the top goes off the screen, simply assign paddle_y = 0. Likewise, if the agent tries to move any part of the paddle off the bottom of the screen, assign paddle_y = 1 - paddle_height.

The rewards are +1 when your action results in rebounding the ball with your paddle, -1 when the ball has passed your agent's paddle, and 0 otherwise.

Use (0.5, 0.5, 0.03, 0.01, 0.5 - paddle_height / 2) as the initial state (see the state representation above). This represents the ball starting in the center and moving towards your agent in a downward trajectory, where the agent's paddle starts in the middle of the screen.

During training and evaluation, when additional games are required, transition immediately to the initial state after receiving a reward of -1.

The ball is a single point, and your paddle is a line. Therefore, you don't need to worry about the ball bouncing off the ends of the paddle.

To simulate the environment at each time step, you must:

- Increment ball_x by velocity_x and ball_y by velocity_y.
- Bounce:
  - If ball_y < 0 (the ball is off the top of the screen), assign ball_y = -ball_y and velocity_y = -velocity_y.
  - If ball_y > 1 (the ball is off the bottom of the screen), let ball_y = 2 - ball_y and velocity_y = -velocity_y.
  - If ball_x < 0 (the ball is off the left edge of the screen), assign ball_x = -ball_x and velocity_x = -velocity_x.
  - If moving the ball to the new coordinates resulted in the ball bouncing off the paddle, handle the ball's bounce by assigning ball_x = 2 * paddle_x - ball_x. Furthermore, when the ball bounces off a paddle, randomize the velocities slightly by using the equation velocity_x = -velocity_x + U and velocity_y = velocity_y + V, where U is chosen uniformly on [-0.015, 0.015] and V is chosen uniformly on [-0.03, 0.03]. As specified above, make sure that all |velocity_x| > 0.03.
  - **Note:** In rare circumstances, either of the velocities may increase above 1. In these cases, after applying the bounce equations given above, the ball may still be out of bounds. If this poses a problem for your implementation, feel free to impose the

restriction that |velocity_x| < 1 and |velocity_y| < 1.

Because this state space is continuous, to allow for it to be learned with Q-learning, we need to be able to convert the continuous state space into a discrete, finite state space. To do this, please follow the instructions below.

- Treat the entire board as a 12x12 grid, and let two states be considered the same if the ball lies within the same cell in this table. Therefore there are 144 possible ball locations.
- Discretize the X-velocity of the ball to have only two possible values: +1 or -1 (the exact value does not matter, only the sign).
- Discretize the Y-velocity of the ball to have only three possible values: +1, 0, or -1. It should map to Zero if |velocity_y| < 0.015.
- Finally, to convert your paddle's location into a discrete value, use the following equation: discrete_paddle = floor(12 * paddle_y / (1 - paddle_height)). In cases where paddle_y = 1 - paddle_height, set discrete_paddle = 11. As can be seen, this discrete paddle location can take on 12 possible values.
- Add one special state for all cases when the ball has passed your paddle (ball_x > 1). This special state needn't differentiate among any of the other variables listed above, i.e., as long as ball_x > 1, the game will always be in this state, regardless of the ball's velocity or the paddle's location. This is the only state with a reward of -1.
- Therefore, the total size of the state space for this problem is (144)(2)(3)(12)+1 = 10369.

Before each training sequence initialize your Q table to zeros.

To learn a good policy, you may need on the order of 100K training games, which should take just a few minutes given a reasonable implementation.

What to Hand In

Hand in A) a zipped directory of all of the code you wrote B) a pdf that answers the following questions. Note that questions 1-3 pertain directly to Pong while questions 4 and 5 should be answered for reinforcement learning more generally.

1. List the different configurations of α, γ, and ε that you trained. For each briefly say why you chose that α, γ, ε set. And for each report your agent's average score (number of successful paddle bounces) averaged over five consecutive games.

2. Specify the best α, γ, ε set you found and its average score.

3) Suppose the initial state is not fixed at (0.5, 0.5, 0.03, 0.01, 0.5 - paddle_height / 2) but is made up randomly, with the only constraint being that the ball is moving and within the playing field. Briefly say how you would expect the training and final behavior to change?

4) Our Pong environment is technically not a Markov decision process

a) Explain why it is not and outline the major potential problems faced by a Q-learning agent applied to a non-MDP environment.
b) What differences in behavior would you expect if our Pong environment were a true MDP?

5) Suppose we run four different Q-learners Q1, Q2, Q3, Q4 in a world similar to Pong. In each learner, α is 0.1, γ is 0.95, but each has a different value for ε: $\varepsilon 1 = 1.0$, $\varepsilon 2 = 0.5$, $\varepsilon 3 = 0.15$, $\varepsilon 4 = 0.01$. During training we periodically estimate the greedy reward rate of each learner: We freeze the policy, set its exploration parameter to 0.0 (no random moves) and run it for a large number of steps (t=1,2,…N). We estimate greedy reward rate as $\frac{1}{N}\sum_{t=1}^{N} r_t$ where r_t is the reward collected at time step *t*. We then set ε back to its original value, unfreeze the policy and resume training. This procedure continues until the estimated greedy reward rate converges to within 95% of its known optimal value. We find that each Q1, Q2, Q3, Q4 is able to reach the 95% optimal threshold after a somewhat different number of total training time steps: t1, t2, t3, t4. We are interested in the relationship among t1, t2, t3, t4. Rate each statement below on a scale of 0 (certainly or almost certainly false) to 5 (certainly or almost certainly true) with intermediate values in between. For each, briefly explain why you rated it as you did.

a) There is no systematic relationship that holds among t1, t2, t3, t4.

b) The differences are likely to be very small so t1, t2, t3, t4 will have very similar values.

c) It is likely that t1 > t2 > t3 > t4.

d) It is likely that t1 < t2 < t3 < t4.

e) It is likely that another relation holds among t1, t2, t3 & t4. (specify)