Winter 2023

CS 441/541

Artificial Intelligence

Programming Assignment #3

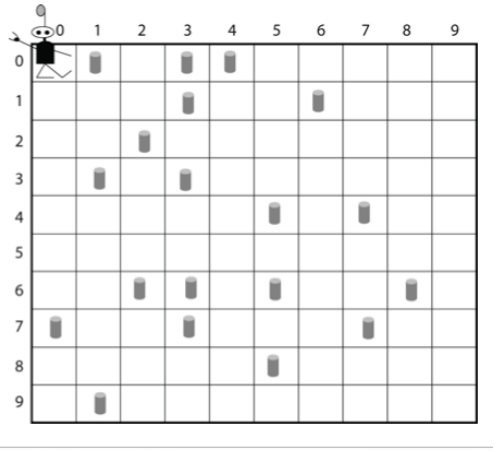
Due Date: Wednesday, 3/15 @ 10pm.

In this assignment you will implement a RL agent to solve the problem given in either Problem #1 OR Problem #2 (the choice is yours – please do not submit solutions to both problems).

Problem #1: Robot Navigation

In this homework you will write code to have Robby the Robot use Q-learning to learn to correctly pick up cans and avoid walls in his grid world.

Robby the Robot:
As was described in class, Robby the Robot lives in a 10 x 10 grid, surrounded by a wall. Some of the grid squares contain soda cans.



Robby has five "sensors": Current, North, South, East, and West. At any time step, these each return the "value" of the respective location, where the possible values are Empty, Can, and Wall.

Robby has five possible actions: Move-North, Move-South, Move-East, Move-West, and Pick-Up-Can. Note: if Robby picks up a can, the can is then gone from the grid.

Robby receives a reward of 10 for each can he picks up; a "reward" of −5 if he crashes into a wall

(after which he immediately bounces back to the square he was in); and a reward of −1 if he tries to pick up a can in an empty square.

Your Assignment:

Part 1: Write a (simple!) simulator for Robby in which he receives sensor input, can perform actions, and receives rewards. Write a Q-learning method for Robby, using a Q-matrix, in which the rows correspond to states and the columns correspond to actions. (We will discuss in class how to easily map sensor input to state-index in the Q-matrix.) The Q-matrix is initialized to all zeros at the beginning of a run.

During a run, Robby will learn over a series of *N* episodes, during each of which he will perform *M* actions. The initial state of the grid in each episode is a random placement of cans, where each grid square has a probability of 0.5 to contain a can (and 0.5 not to contain a can). Robby is initially placed in a random grid square.

At each time step *t* during an episode, your code should do the following:

- Observe Robby's current state s_t
- Choose an action a_t, using ε-greedy action selection
- Perform the action
- Receive reward r_t (which is zero except in the cases specified above)
- Observe Robby's new state s_{t+1}
- Update $Q(s_t, a_t) = Q(s_t, a_t) + \eta(r_t + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$

At the end of each episode, generate a new distribution of cans and place Robby in a random grid square to start the next episode. (Don't reset the Q-matrix — you will keep updating this matrix over the *N* episodes. Keep track of the total reward gained per episode.

To do a run consisting of *N* episodes of *M* steps each, use the following parameter values:

$$N = 5{,}000\ ;\ M = 200\ ;\ \eta = 0.2;\ \gamma = 0.9$$

For choosing actions with ε-greedy action selection, set $\varepsilon = 0.1$ initially, and progressively decrease it every 50 epochs or so until it reaches 0 (recall that ε denotes the probability of performing the non-optimal/non-greedy action from the current state). After that, it stays at 0.

Run the *N* episodes of learning, and plot the total sum of rewards per episode (plotting a point every 100 episodes). This plot—let's call it the *Training Reward* plot—indicates the extent to which Robby is learning to improve his cumulative reward.

After training is completed, run *N test* episodes using your trained Q-matrix, but with $\varepsilon = 0.1$ for all *N* episodes. Again, regenerate a grid of randomly placed cans at the beginning of each episode and also place Robby in a random grid location. Calculate the average over sum-of-rewards-per-episode, and the standard deviation. For simplicity in this writeup, let's call these values *Test-Average* and *Test-Standard-Deviation*. These values indicate how a *trained* agent performs this task in new environments.

In your report, describe the experiment, give the *Training Reward* plot described above (plotted

every 100 episodes), and *Test-Average* and *Test-Standard-Deviation*.

(Optional) Parts 2-5 are optional

Part 2: Experiment with Learning Rate. Choose 4 different values for the learning rate, η, approximately evenly spaced in the range [0,1], keeping the other parameters set as in Part 1. For each value, give the *Training Reward* plot (plotted every 100 episodes), and the *Test-Average* and *Test-Standard-Deviation.* Discuss how changing the learning rate changes these results.

Part 3: Experiment with Epsilon. Try learning with a constant epsilon (choose a value ε in [0,1]). Give the *Training Reward* plot and *Test-Average* and *Test-Standard-Deviation*. How do your results change when using a constant value of epsilon rather than a decreasing value? Speculate on why you get these results.

Part 4: Experiment with negative reward for each action. Modify your code so that a negative reward (an "action tax") of -0.5 is given in addition to the original rewards for each action. Run learning and testing with the parameter values of Part 1, and give the *Training Reward* plot and the *Test-Average* and *Test-Standard-Deviation.* What differences do you see from the results in Part 1?

Part 5: Devise your own experiment, different from those of Parts 1−4 above. This can involve a change to a parameter value, a change in the rewards, a modification of the actions or sensors, etc. Describe your experiment and give plots or values that show the results. Are the results what you expected? Why or why not? One suggestion: use a Neural Network in place of the Q-table.

Here is what you need to turn in:

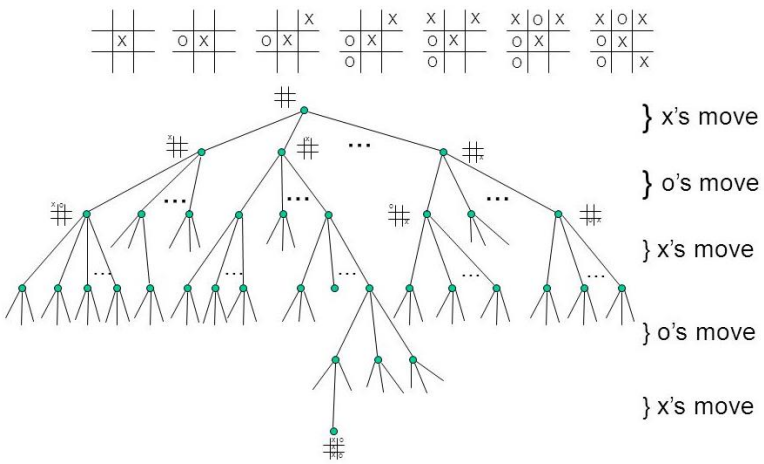
Your spell-checked, double-spaced report with the information requested above. Also, your (briefly) commented code with instructions how to run it.

How to turn it in:

- Send these items in electronic format tour TA on the due date. No hard copy please.
- The report should be in pdf format and the code should be in its original format (e.g., .py, .m, etc.)

Problem #2: Tic-Tac-Toe Agent

In this part you will write code to implement tic-tac-toe using techniques from Reinforcement Learning.



Your Assignment:

Part 1: Write a simple simulator for tic-tac-toe that encodes (and displays) a standard 3x3 grid, with a check for licit moves and "goal state" (i.e. did someone win/draw).

Implement a form of Q-learning to train a tic-tac-toe playing "agent". One straightforward option for Q-learning is to use a Q-matrix, in which the rows correspond to states and the columns correspond to actions (you have the option of using a more sophisticated method than a Q-matrix if you wish, such as a Q-network). The Q-matrix is commonly initialized to all zeros at the beginning of a run (although you may use a different initialization strategy if you prefer – please note this in your write-up).

At each time step t during an episode, your code should do the following:

- Observe the current state s_t
- Choose an action a_t, using ε-greedy action selection (I recommend training *on-policy*)
- Perform the action
- Observe the new state s_{t+1}
- Update $Q(s_t, a_t) = Q(s_t, a_t) + \eta(r_t + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$
- Receive reward r_t (at the conclusion of a game, include "reward" for outcomes, including: win, lose or draw)

For choosing actions with ε-greedy action selection, initialize ε (for example: $\varepsilon = 0.1$), and decrease it by a constant value Δ every m epochs until it reaches 1 (I encourage you to experiment with different choices for Δ and m, as well as the initial value for ε).

After each training epoch, test your agent on 10 games against a baseline random opponent. Record the total score of your agent against the baseline out of these 10 games (+1 for a win, +.5 for draw, 0 for loss). After training is completed, print a plot of the training progress for your agent with the epoch number on the horizontal axis and the (total score) / 10 against the baseline opponent on the vertical axis.

When you have completed training, play 10 games against your agent and report these results.

(Optional) Parts 2-3 are optional

Part 2: Experiment with Learning Rate. Choose 4 different values for the learning rate, η, approximately evenly spaced in the range [0,1], keeping the other parameters set as in Part 1. For each value, give a performance plot as described above. Discuss how changing the learning rate changes these results.

Part 3: Q-network. Use a Q-network in place of a Q-matrix; describe how you train and architected your network. Provide a performance plot.

Here is what you need to turn in:

Your spell-checked, double-spaced report with the information requested above. Also, your (briefly) commented code with instructions how to run it. Your report should include a detailed summary of your approach and results.

How to turn it in:

- Send these items in electronic format to our course TA on the due date. No hard copy please!
- The report should be in pdf format and the code should be in its original format (e.g., .py, .m, etc.)