Homework 4 [100 pts]: Complex Decisions and Reinforcement Learning

General Information

Deadline

Due: 11:59 PM, Friday, November 18th, 2022
Late homework will be penalized by 10% per day (where each day starts at 12:00 AM on the due day). There are only 3 late days for this homework.

Gradescope

Please familiarize yourself with Gradescope if you have never used it before. In particular, when you submit an assignment, Gradescope asks you to select pages of your solution that correspond to each problem. This is to make it easy for graders to find where in your work they should grade.
For the written questions, put your answers in a PDF and submit to the corresponding assignment on Gradescope. You can submit this PDF as many times as you would like before the deadline. You can use the provided Word template and fill in your answer. You may also submit your written response homework in a separate document.
For the coding questions, submit your code file (MCTS.py) to the Gradescope autograder. A template starter code is provided for you. Do not modify any of the imports in these files. You can submit to the autograder as many times as you would like before the deadline. While you will not be graded on style, we encourage you to uphold best-practice style guidelines.

Setup

You are recommended to use a Python virtual environment for the coding portion. You may follow this tutorial from EECS 485 to create and set up a virtual environment.
We have provided you with a requirements.txt. After creating the environment, as mentioned in the tutorial, activate the environment by executing source env/bin/activate in the terminal. Install the necessary library by using pip install -r requirements.txt.
Feel free to visit us in office hours if you need help on the above or help on setting up tools for debugging.

Written Section [50 pts]

Exercise 1 [10 pts]: Short questions

Provide a brief, clear explanation of the distinction between two concepts.

a. Policy iteration and Value iteration
   Policy iterations are expensive and information flows rapidly between states. Value iterations are cheaper, and information flows slowly. Policy iteration uses a random policy, and value iteration uses an arbitrary value function and iteratively finds the utility of each state

b. Hard SVM and soft SVM
   Hard SVMs require separable sets, while SVMs introduce the slack variable as a penalty term for points that are misclassified or are within the margin boundary

c. MDP and POMDP
   POMDPs and partially observable MDPs. MDPs are fully observable and allow for no uncertainty.

d. Value function and q-function
   Value functions give the expected utility of a policy given a state. Q-functions take into account state and action to calculate the expected utility of the policy.

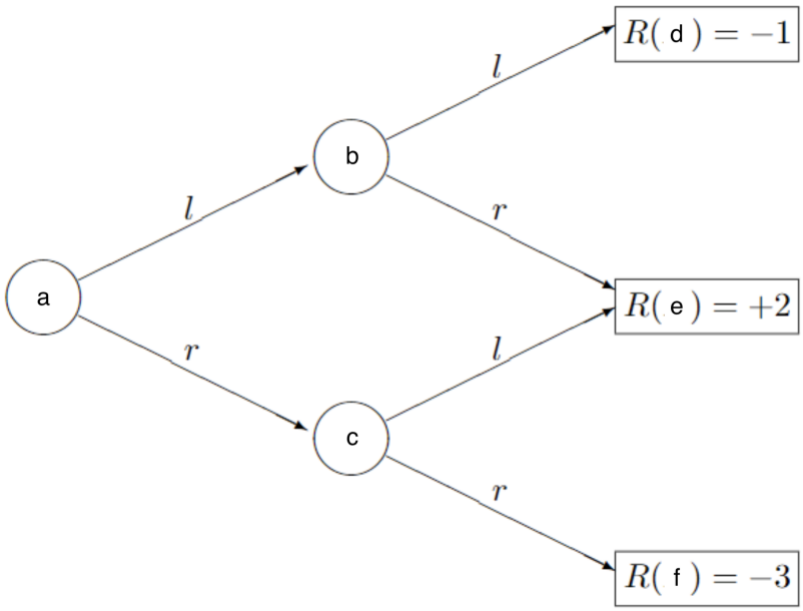
Exercise 2 [10 pts]: Utility and Decision Theory

Suppose you are looking to buy a used car. You can either go to the dealer immediately and make a decision, or you can first pay a website for more information about the quality of each car. If you don't pay the website, you'll buy a good quality used car with probability 0.5 and a poor quality car with probability 0.5. If you pay for information, however, you'll buy a good quality car with probability 0.8 and a poor quality car with probability 0.2. A good quality car will cost $1000 dollars in maintenance, but a poor quality car will cost $5000 in maintenance. How much money is the information worth in expectation? Show your work below for partial credit.

E[u(pay don't for website)] = 0.5(1000) + 0.5(5000) = $3,000
E[u(pay for website)] = 0.8(1000) + 0.2(5000) = $1,800
E[u(pay don't for website)] - E[u(pay for website)] = 3000 – 1800 = $1200
It is worth up to $1,199 for the information in expectation.

Exercise 3 [10 pts]: MDPs

Consider a simple Markov Decision Process with six states. States a, b, and c are non-terminal states, with rewards R(a) = R(b) = R(c) = 0. States d, e, and f are terminal states, with R(d) = −1, R(e) = +2, and R(f) = −3. States a, b, and c each have two defined actions, r (right) and l (left). Performing an action results in the intended state with probability 0.6, and the other state with probability 0.4. For example, from state a, the action l has probability 0.6 of resulting in b, and probability 0.4 of resulting in c. That is, P(b|a, l) = 0.6 and P(c|a, l) = 0.4.



Let q*(S, r) be the expected utility of performing the action "right" in state S. When calculating the q-function, assume γ = 1. Calculate and enter into the cells of the following array, the values of each of the terms along the left column, for the non-terminal state specified at the top of each column. You need to show all of your work to earn the full credits. Make sure to reference the notation from the lecture slides if the notation looks unfamiliar. The notation below follows the lecture rather than the textbook. Show your work below for partial credit.

	S = b	S = c	S = a
q*(S, l)	0.2	0	0.48
q*(S, r)	0.8	-1	0.32
v*(S)	0.8	0	0.48
π*(S)	R	L	L

$$q^*(b,L) = p(b'|b,L)\,(R(b') + \gamma \max_{L} q^*(b',L))$$

$$= p(d|b,L)(R(d) + q^*(d,e)) + p(e|b,L)(R(e) + q^*(e,e))$$

$$= 0.6(-1+0) + (0.4)(2+0)$$

$$= 0.6(-1) + 0.4(2)$$

$$= -0.6 + 0.8 = 0.2$$

$$q^*(c,L) = p(e|c,e)(R(e) + q^*(e,e)) + p(f|c,e)(R(f) + q^*(f,e))$$

$$= 0.6(2) + 0.4(-3)$$

$$= 1.2 - 1.2 = 0$$

$$q^*(a,L) = p(b|a,L)(R(b) + q^*(b,A)) + p(c|a,L)(R(c) + q^*(c,A))$$

$$= 0.6(0+0.8) + 0.4(0+0)$$

$$= 0.48$$

$$q^*(b,r) = p(d|b,r)(R(d) + q^*(d,r)) + p(e|b,r)(R(e) + q^*(e,r))$$

$$= 0.4(-1+0) + 0.6(2+0)$$

$$= -0.4 + 1.2$$

$$= 0.8$$

$$q^*(c,r) = p(e|c,r)(R(c) + q^*(e,r)) + p(f|c,r)(R(f) + q^*(f,r))$$

$$= 0.4(2+0) + 0.6(-3+0)$$

$$= 0.8 - 1.8$$

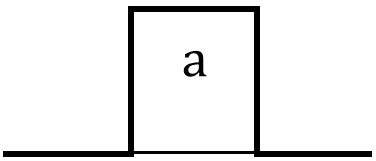
$$= -1$$

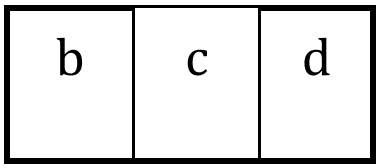
$$q^*(a,r) = p(b|a,r)(R(b) + q^*(b,A)) + p(c|a,r)(R(c) + q^*(c,A))$$

$$= 0.4(0.8) + 0.6(0)$$

$$= 0.32$$

Exercise 4 [20 pts]: POMDP





Consider the simple 4-state environment above. You are not sure exactly where you are, but your belief state is p(a) = 0.5, p(b) = 0.125, p(c)= 0.25, p(d)= 0.125. In any state you can attempt to go north, south, east, or west. Also, in any state, you can perceive if there are 1 or 3 walls around you (the thick lines in the environment represent walls ("a" has 3 walls. "c" has 1 wall)). You will correctly perceive the number of walls around you with probability 0.8 (otherwise you'll perceive incorrectly with probability 0.2). For any action, there is a

- 0.7 chance that you move in the given direction.
- 0.1 chance that you move in a 90 degrees clockwise direction instead
- 0.1 chance that you move in a 90 degrees counterclockwise direction instead
- 0.05 chance that you move in the opposite direction
- 0.05 chance that you don't move at all

1. (4 pts) For each state a, b, c, d, list the possible resulting states of going north starting in that state.

	S = a	S = b	S = c	S = d
Resulting State	a, c	b, c	a, b, c, d	c, d

2. (16 pts) Suppose you take a single action, "north." In the state you arrive in, you do receive a <u>3 walls</u> percept. Calculate the resulting belief state. Show your work below for partial credit.

	S = a	S = b	S = c	S = d
Belief States	0.6887	0.1457	0.0199	0.1457

sensor ✓ state a state c

$p'(a) = \alpha(0.8) * (0.95 * 0.5 + 0.7 * 0.25) = \alpha 0.52$

$p'(b) = \alpha(0.8) * (0.9 * 0.125 + 0.1 * 0.25) = \alpha 0.11$

$p'(c) = \alpha(0.2) * ((0.05 * 0.5) + (0.1 * 0.125) + (0.1 * 0.25) + (0.1 + 0.125)) = \alpha 0.015$

$p'(d) = \alpha(0.8) * (0.9(0.125) + (0.1)(0.25)) = \alpha 0.11$

$1 = (0.52 + 0.11 + 0.015 + 0.11)\alpha$

$\alpha = 1.3245$

$p'(a) = 0.6887 \quad p'(b) = 0.1457 \quad p'(c) = 0.0199 \quad p'(d) = 0.1457$

Programming Section [50 pts]

Introduction and Objective

In this assignment, we'll be implementing AlphaZero for othello. There are many parts to AlphaZero but we will only focus on the Monte Carlo Tree Search (MCTS) portion of it. As long as you have a basic knowledge of MCTS and machine learning (which you should at this point of the course!) then you'll be fine with understanding the project. To summarize, you'll be implementing the **search()**, **select()**, **backpropagate()**, **simulate()** functions in **MCTS.py**. Once you have finished your code, you will submit your MCTS.py file to GradeScope. You can find the notebook to start here.

It's important to note that the MCTS you implement in this homework isn't the exact same as the lecture. The main difference is that instead of a rollout for the simulation step, you'll use a neural network to get the value. **Make sure to read the whole spec before coding. It'll save you a huge amount of time.**

Othello

Having an understanding of Othello isn't needed for the implementation. But having knowledge of the game helps with understanding the test cases and the actions you can take. You don't need an in-depth understanding of the game and you can find details here.

AlphaZero

For this project, we will be implementing a different version of the original AlphaZero. You can read the author's implementation of our search() function here. The key idea with AlphaZero is that when you reach the simulation step of MCTS, you'll rely on a neural network to predict the reward instead of doing a rollout. It's important to note that the algorithm doesn't perform very well since we wanted the students to get results quickly (for debugging).

Google Colab Notebook

For this programming section, because we are running neural networks, some computers are unable to run this assignment. To make sure all computers are able to run the code efficiently, we are requiring students to write and run the code through a Google Colab Notebook. To start this programming assignment make a copy of eecs492hw4main.ipynb. To successfully make a copy of the Google Colab notebook, go to “File” >> "Save a Copy in Drive '' on the upper left-hand corner. If your runtime disconnects because of inactivity, you will have to run the necessary code blocks again. Also note that we're not using GPU in our Google Colab, so do not manually enable GPU.

High level view of AlphaZero Overall Algorithm

This section is optional. The variables bolded are hyperparameters discussed in the main.ipynb section. This isn't necessary for the implementation, it's just nice to know how the whole thing works.

1. For ***numIters*** number of iterations, execute episodes ***numEps*** number of times
2. For each episode, perform MCTS (search()) on the tree until a terminal node is reached
    a. Your implementation will be used at this search step
3. Each search will generate test samples
4. Train the neural network on the generated test samples
5. Have the neural network play against the old version of the neural network ***arenaCompare*** amount of times
6. If it did better, make the neural network the new best one
7. Repeat

Overview

These are the relevant python files for the assignment files:

- main.ipynb
- Coach.py
- MCTS.py

Your implementation of **search()** will update policy values for a given state and action. So when the algorithm in **executeEpisode()** tries to take an action, it'll take one according to the policy values you have updated. The actions the algorithm takes in **executeEpisode()** is what we will use to compare your results against the instructor's.
The MCTS.py file will contain additional comments to help you understand what functions are relevant for your code. Below is high level information that may be helpful for your understanding of the algorithm. Lower level details will be found within the files under the function headers in the python files.

main.ipynb

This is the main file you will use to run and test your solution. This file sets up the program so you can start to train with MCTS. You do not need to worry about setting up the board or running the code, that will be done for you.
There is an argument variable (args) for you to tune hyperparameters and toy around if you like. For the actual project, do not modify these values. The existing hyperparameters are needed for grading. Only modify the hyperparameters if you're curious about how the model will perform when optimized. The most important hyperparameters are given below.

- numIters
    - The number of iterations the algorithm will take when learning
- numEps
    - The number of episodes each iteration takes. This will determine how many training samples the model will have each iteration
- arenaCompare

    - The number of times the new neural network plays against the old one. See the bullet point below for additional details

Coach.py

This class is the helper function to train your algorithm. Most of the learning will be done here. These are the most relevant methods

- learn(self)
    - This performs the training of the model
- executeEpisode(self)
    - This is where search() will be indirectly called through getActionProb(). Your implementation will update the policy values. When executeEpisode() tries to take an action, it'll take one according to the policy values you have updated. **The action this function takes will be compared against the instructor's solution set.**

MCTS.py

This class will contain your implementation. These are the relevant methods for the homework. You **shouldn't** call any methods outside of these. The bolded methods are the ones you have the implement for the homework

- def gameEnded(self, canonicalBoard)
    - Determines if the current board position is the end of the game. Will return ***r***, a value that returns 0 if the game hasn't ended, 1 if the player won, -1 if the player lost
- def predict(self, state, canonicalBoard)
    - Computes, **r**, the reward from the neural network. Use this for the simulation step instead of a rollout. Returns the reward of the simulated rollout
- def getValidActions(self, state)
    - Returns **all valid actions** you can take in terms of a list of integers
- def updateValues(self, r, state, action)
    - Updates the necessary values for backpropagation for a given state and action
- def getConfidenceVal(self, state, action)
    - Returns **u**, an upper confidence bound value for a given state and action
- def expand(self, state)
    - Expands the state in MCTS
- **def search(self, initial_board)**
    - This is the main function for your Monte Carlo Tree Search
- **def select(self, state, board)**
    - This is the selection phase of MCTS
- **def backpropagate(self, seq)**
    - This is the backpropagation phase of MCTS
- **def simulate(self, state, board)**
    - This is the simulate phase of MCTS

Below is the pseudocode for what you have to implement in the homework. Implement the homework according to it.

```
function SEARCH(b_0)
    inputs: b_0, a board object representation of the initial othello board

    s_0 ← stringRepresentation(b_0) // a string representation of the board
    repeat
        s ← s_0
        b ← b_0
        initialize seq
        r ← none
        repeat
            s', b', action, r' ← SELECT(s, b)
            add (s, action, r') to seq // (s, action, r') is a tuple
            s ← s'
            b ← b'
            r ← r'
        until r is not none
        BACKPROPAGATE(seq)
    until time limit T
    return b_0
```

```
function SELECT(s, b)
    inputs: s, a string representation of an othello board, b, a board
object representation of an othello board

    r ← gameEnded(b)
    if game has ended
        return none, none, none, -r
    else if s has not been visited
        EXPAND(s) // EXPAND(s) is already implemented for you
        r ← SIMULATE(s, b)
        return none, none, none, -r
    u ← -inf
    action ← none
    for action' ∈ valid actions
        u' ← the confidence value given s, action'
        if u' > u
            u ← u'
            action ← action'
    b ← nextState(b, action)
    s ← stringRepresentation(b)
    return s, b, action, none
```

```
function BACKPROPAGATE(seq)
    r ← 0
    for s, action, r' in reversed(seq)
        if r' is not none
            r ← r'
        else
            updateValues(r, s, action)
            r ← -r
```

```
function SIMULATE(s, b)
    r ← predict(s, b)
    return r
```