Recitation 13 April 17

1 MCTS Primer: `FlatMCSearch`

Here, we will be implementing (together) a class called `FlatMCSearch`, which is a simplified version of 2-player Monte Carlo Tree Search. You should fill in the blank lines, denoted by blanks that look like this:

.....

Make sure to read the comments which describe what the lines are doing. These questions will help you understand the interface of our AlphaZero assignment (HW12).

1.a `rollout_uniform`

Before we can implement the full `FlatMCSearch` class we must first implement the `rollout_uniform` helper function which performs a rollout from a given game state. Here are some useful Game API methods/attributes that might be useful:

- `game.get_valid_moves()`
- `game.make_move(action_index)`
- `game.done`
- `game.get_reward(player_index)`

For now don't worry about the API implementation, you can assume the functions do what you think they do.

Hint: You might want to make use of `np.random.choice`

```
1  def rollout_uniform(game, rollout_player=None):
2      if rollout_player is None:
3          rollout_player = game.current_player
4
5      while ............... :
6          actions = ............. # How can we get actions?
7          a = ................ # How can we pick an action?
8
9          ................... # What else is there left to do?
10
11     return .............  # What should we return?
```

Based on the implementation above answer these questions:

1. If players alternate turns, whose perspective is the returned reward from?

   It's from the starting player's perspective (`rollout_player`)

2. Why can two rollouts from the same state produce different values?

   Random action sampling makes different rollouts end differently.

3. In AlphaZero, what component replaces rollouts?

   AlphaZero replaces rollouts with a neural net value estimate (with policy priors).

1.b `FlatMCSearch`

Now, let's implement `__init__` and `search` of `FlatMCSearch`.

Hint: You might want to make use of `np.random.choice` and other `np` functions. Also, as a reminder, the softmax function is as follows:

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

```
class FlatMCSearch:
    def __init__(self, num_simulations=100):

        ..................... # What should we initialize?

    def search(self, game):
        # Don't mutate this game! Make a copy with game.clone() instead
        actions = game.get_valid_moves()
        root_player = game.current_player

        N_a = defaultdict(int) # What does this store?
        sum_g_a = defaultdict(float) # What does this store?

        for _ in range(..............): # What do we want to loop over?
            game_copy = game.clone()

            a = ........................ # Pick a random action
            N_a[a] += 1

            game_copy.make_move(a)

            # Since we already did game_copy.make_move(a), the current player is
                now root player's opponent, so we must pass in root_player

            g = rollout_uniform(game_copy, root_player) # What does g stand for?
            sum_g_a[a] += g # What does sum_g_a store?

        # Why do we initialize policy to a vector of -infinity?
        policy = float('-inf')*np.ones(game.get_action_size(), dtype=np.float32)

        for a in N_a:

            policy[a] = ..................... # Average score of each action

        # Softmax
        numer = .....................

        denom = .....................

        policy = ......... / ..........

        return policy
```

(Select all that apply) Which MCTS phases are implemented in the code above?

- ☐ Selection
- ☐ Expansion
- ☐ Rollout/Simulation

☐ Backup

Only Rollout and Backup, Selection and Expansion are done in MCTS but not in the code above, hence `FlatMCSearch`.

2 Approximate Q-Learning

You decide to go to Kennywood this weekend. Our RL problem is based on choosing a ride from a set of many rides.

You start off feeling well, getting positive rewards from rides, some larger than others. However, there is some chance of each ride making you sick. If you continue going on rides while sick there is some chance of becoming well again, but you don't enjoy the rides as much, receiving lower rewards (possibly negative).

You have never been to an amusement park before, so you don't know how much reward you will get from each ride (while well or sick). You also don't know how likely you are to get sick on each ride, or how likely you are to become well again. What you do know about the rides is:

Actions / Rides	**Type**	**Wait**	**Speed**
Steel Curtain	Rollercoaster	Long	Fast
Lil' Phantom	Rollercoaster	Short	Slow
Cranky's Tower	Drop Tower	Short	Fast
Pirate	Pendulum	Short	Slow
Leave the Park	Leave	Short	Slow

We will formulate this as an RL problem with two states, **well** and **sick**. Each ride corresponds to an action. The 'Leave the Park' action ends the current run through the problem. Taking a ride will lead back to the same state with some probability or take you to the other state. We will use a feature-based approximation to the Q-values, defined by the following four features and associated weights:

Features	**Initial Weights**
$f_0 = f_{\text{sick}}(s,a) = \begin{cases} 0, s = \text{Well} \\ -5, s = \text{Sick} \end{cases}$	$w_0 = 1$
$f_1 = f_{\text{type}}(s,a) = \begin{cases} 1, \text{ if } \textit{action} \text{ type is Rollercoaster} \\ 0, \text{otherwise} \end{cases}$	$w_1 = 2$
$f_2 = f_{\text{wait}}(s,a) = \begin{cases} 1, \text{ if } \textit{action} \text{ wait is Short} \\ 0, \text{otherwise} \end{cases}$	$w_2 = 1$
$f_3 = f_{\text{speed}}(s,a) = \begin{cases} 1, \text{ if } \textit{action} \text{ speed is Fast} \\ 0, \text{otherwise} \end{cases}$	$w_3 = 0.5$

1. Calculate Q-values for each action, given the state is $'Sick'$.

Recall we use the following function, where a denotes the corresponding action:

$$Q(s,a) = \sum_a w_a \cdot f_a(s,a)$$

(a) Q('Sick', 'Steel Curtain') = 1(-5) + 2(1) + 1(0) + 0.5(1) = -2.5

(b) Q('Sick', 'Lil' Phantom') = 1(-5) + 2(1) + 1(1) + 0.5(0) = -2

(c) Q('Sick', 'Cranky's Tower') = 1(-5) + 2(0) + 1(1) + 0.5(1) = -3.5

(d) Q('Sick', 'Pirate') = 1(-5) + 2(0) + 1(1) + 0.5(0) = -4

(e) Q('Sick', 'Leave the Park') = 1(-5) + 2(0) + 1(1) + 0.5(0) = -4

2. Apply a Q-learning update based on the sample ('Well', 'Steel Curtain', 'Sick', -10.5), using a learning rate of $\alpha = 0.5$ and discount of $\gamma = 0.5$. What are the new weights?

Recall we use the following function:

$$w_i \leftarrow w_i + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \cdot f_i$$

We know that $Q(s, a) = Q(\text{Well}, \text{Steel Curtain}) = 1(0) + 2(1) + 1(0) + 0.5(1) = 2.5$.
We have that $r = -10.5$, $\alpha = 0.5$, and $\gamma = 0.5$.
Our next state is "Sick", so we can use the information from part 1 of this question.
Difference $= [r + \gamma \max_{a'} Q(s', a') - Q(s, a)] = -10.5 + 0.5 * max(-2.5, -2, -3.5, -4, -4) - 2.5 = -14$

$w_0 = 1 + (0.5 * -14) * 0 = 1$
$w_1 = 2 + (0.5 * -14) * 1 = -5$
$w_2 = 1 + (0.5 * -14) * 0 = 1$
$w_3 = 0.5 + (0.5 * -14) * 1 = -6.5$

3. Now we will consider the exploration exploitation tradeoff in this amusement park. Assume we have the initial weights from the table above. What action will an ϵ-greedy approach choose from the "Well" state? If multiple actions could be chosen, give each action and its probability.

We can calculate the Q-values for "Well" state:

(a) Q('Well', 'Steel Curtain') = 1(0) + 2(1) + 1(0) + 0.5(1) = 2.5

(b) Q('Well', 'Lil' Phantom') = 1(0) + 2(1) + 1(1) + 0.5(0) = 3

(c) Q('Well', 'Cranky's Tower') = 1(0) + 2(0) + 1(1) + 0.5(1) = 1.5

(d) Q('Well', 'Pirate') = 1(0) + 2(0) + 1(1) + 0.5(0) = 1

(e) Q('Well', 'Leave the Park') = 1(0) + 2(0) + 1(1) + 0.5(0) = 1

We can see the that the action that gives us the largest Q-value is 'Lil' Phantom'.

Recall our ϵ-greedy formulation:

$$\text{action at time } t = \begin{cases} \underset{Q(s,a)}{\arg\max} & \text{with probability } 1-\epsilon \\ \text{any action } a & \text{with probability } \epsilon \end{cases}$$

With probability $(1 - \frac{4\epsilon}{5})$ will choose Lil' Phantom. Each other action will be chosen with probability $\frac{\epsilon}{5}$.

Since we have an ϵ probability of randomly selecting a random action, each action (including Lil' Phantom) has a $\frac{\epsilon}{5}$ probability of getting chosen. So, the probability of selecting Lil' Phantom is $(1-\epsilon) + \frac{\epsilon}{5} = (1 - \frac{4\epsilon}{5})$. There are 4 non-maximal actions, so there is a $\frac{\epsilon}{5}$ probability of selecting each of those.

4. When running Q-learning another approach to dealing with this tradeoff is using an exploration function:

$$f(u, n) = u + \frac{k}{n}$$

(a) How is this function used in the Q-learning equations?

The update replaces the max over Q values with a max over the exploration function (with Q and N passed in as arguments).

(b) What does u represent in the exploration function?

The utility, given by Q.

(c) What does n represent in the exploration function?

The number of times this state has been visited.

(d) What does k represent in the exploration function?

A constant. Adjusting this constant allows control over how "optimistic" we are about states we haven't visited much.