Modeling LTTT as an imperfect information game, and solving it using two algorithms, namely counterfactual regret-minimization (CFR) and Monte-Carlo CFR with outcome sampling (MCCFR-outcome).

Sree Pooja Akula

Missouri University of Science and Technology

GAME MODEL:

Latent Tic- Tac-Toe (LTTT) defined: Sequential decision-making with multiple agents and imperfect information is commonly modeled as an extensive game. One efficient method for computing Nash equilibria in large, zero-sum, imperfect information games is counterfactual regret minimization (CFR). We try to implement **Latent Tic-Tac-Toe (LTTT)** environment in Python. LTTT is a modified version of the classical Tic-Tac-Toe, where the game's state is not revealed until after the opponent's next move and lost if deemed invalid at the time of revelation. We try to model LTTT as an imperfect information game, and solve it using two algorithms, namely counterfactual regret-minimization (CFR) and Monte-Carlo CFR with outcome sampling (MCCFR-outcome).

Player: A player is denoted by i ∈ N, where N is the set of players. There are 2 players in LTTT.

```
PLAYERS = cast(List[Player], [0,1])
```

Action: Action a, A(h)= a:(h,a)∈H where h ∈ H is a non-terminal history.

```
ACTIONS = cast(List[Action], ['0', '1', '2', '3', '4', '5', '6', '7', '8' ])
```

Actions are indexes of Latent Tic-Tac-Toe board.

History: History h ∈ H is a sequence of actions, and H is the set of all histories. Z⊆H is the set of terminal histories (game over). This defines when a game ends and calculates the utility. The history is stored in a string.

Information set: Information set for player i is similar to a history h ∈ H but only contains the actions visible to player i. That is, the history h will contain actions/events such as moves dealt to the opposing player while Ii will not have them. Ii is known as the **information partition** of player i. h ∈ I is the set of all histories that belong to a given information set, i.e. all those histories look the same in the eye of the player.

Utility: The terminal utility is the utility (or pay off) of a player i for a terminal history h.

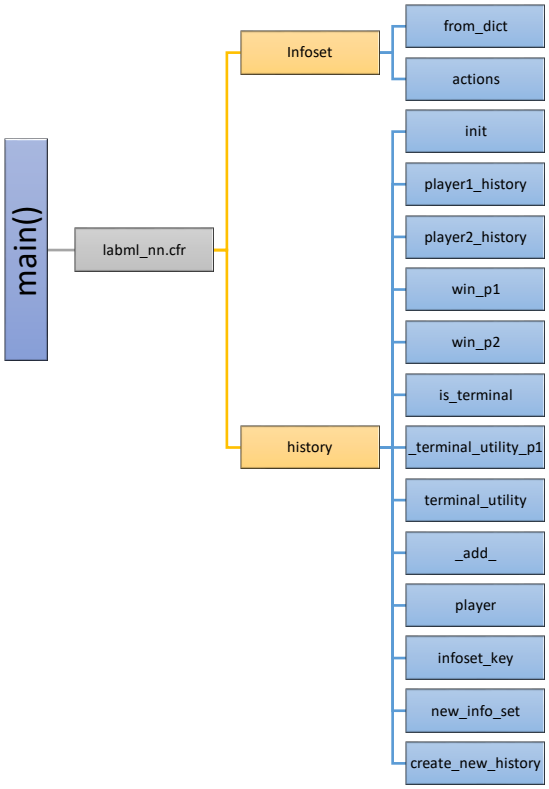
Utility - u_i (h)where h ∈ Z
$u_i(\sigma)$ is the expected utility (payoff) for player i with strategy profile σ.
$u_i(\sigma)=\sum u_i(h)\ \pi^{\sigma}(h)$

SOLUTION CONCEPT:

The fundamental idea of this approach is to model LTTT as an imperfect information game and solve it using two algorithms. Extensive games are a natural model for sequential decision-making in the presence of other decision-makers, particularly in situations of imperfect information, where the decision-makers have differing information about the state of the game.

We implement Counterfactual regret-minimization (CFR) and Monte Carlo Counterfactual Regret Minimization (MCCFR). It iteratively, explores part of the game tree by trying all player actions. Then it calculates, for each action, the regret of following the current strategy instead of taking that action. Then it updates the strategy based on these regrets for the next iteration, using regret matching. Finally, it computes the average of the strategies throughout the iterations, which is very close to the Nash equilibrium if we ran enough iterations.

UML Diagram of LTTT Source Code:



RESULTS:

The unique dependencies for this set of environments are imported from labml package.

Action set at both the players: Each action represents the index of the location on the board.

```
0 | 3 | 6
_________

1 | 4 | 7
_________

2 | 5 | 8
```

- In the history class, initialize the history string to empty.
- Then, player 1's action history is extracted by returning by all the even positioned characters in history.
- And player 2's action history is extracted by returning by all the odd positioned characters in history.
- Then we define winning cases for both the players:

```
def win_p1(self):
    # All the win cases. A player will win if he has any of the following characters in his action string.
    # Ex: If player 0 played '21386', he wins cause he has '1', '3', '6' in his action string.
    win = ['012', '345', '678', '136', '147', '258', '048', '256']
    for i in win:
        player1_win = all(string in self.player1_history for string in i)

        if player1_win == True:
            return True

        else:
            return False
```

- Then, we check for terminal cases, to check whether the game is invalid or if one of the players win the game or if it is a draw match.

```
def is_terminal(self):
    # To check if game is invalid.
    # The game is invalid if both the players choose the same move.
    # We can check this by checking if the latest action entered into the history is already
    # added to the history string
    # Ex: The following code returns true if h = '345' and the last entry in history is '4'
    if len(self.history) > 2:
        if self.history[-1] in self.history[:-1]:
            return True
    # To check if the game resulted in a draw.
```

```
    # If all the locations on the board are filler, i.e., if the len(h) is 9 then its the terminal node.
    elif len(self.history) == 9:
        return True
    # If any of the player wins the game.
    elif self.win_p1 == True:
        return True

    elif self.win_p2 == True:
        return True

    else:
        return False
```

- We calculate terminal utilities just as players of zero-sum games.
- In the information set of each player, the current player cannot observe the previous player's move, all we must do is return history without the last character in the string.
- We then create an experiment and do iterations on it.

In the base configurations of labml_nn.cfr, which was imported, epochs are 1,00,000. My system crashed as it couldn't handle such computational complexity.

```
Prepare cfr...
  Prepare create_new_history...[DONE]    0.04ms
Prepare cfr...[DONE]    0.51ms

latent_tic_tac_toe: 37c66a5a5d1011ec89beacde48001122
        [dirty]: "reptile paper"
Train...
```

REFERENCES:

1. https://nn.labml.ai/cfr/index.html
2. **Counterfactual Regret Minimization:** Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, Advances in Neural Information Processing Systems 20, pp. 1729-1736. MIT Press, Cambridge, MA, 2008.
3. **Counterfactual Regret Minimization:** Todd W. Neller, Marc Lanctot. An Introduction to Counterfactual Regret Minimization. A project description (model AI assignment) in their tutorial on "An Introduction to Counterfactual Regret Minimization" in EAAI-2013.
4. https://www.pettingzoo.ml/classic/tictactoe