The 3D Tic-Tac-Toe model designed based on reinforcement learning

Introduction

The current board state is stored using a 3-layer list: **board**, and the **states_value** dictionary is used to store the mapping from board states to their corresponding values.

Three objects were created: two computer players and one human player. One computer player takes the first move, while the other takes the second. The computer players can automatically choose the optimal solution under the current model based on the values in the states_value dictionary.

The training process is as follows: The two computer players, P1 and P2, created during initialization are used, where P1 plays first and P2 plays second. During the training, process rewards are given to the current player based on the game progress, and result rewards are provided at the end of the game based on the outcome. This process is repeated 100,000 times.

This model was designed with reference to the framework of a 2D open-source model, but the following sections will focus only on the innovations and optimizations made beyond the original foundation.

Reward mechanism

The model is designed with two reward mechanisms: one based on the outcome and the other based on the process.

While rewarding victories, the model also penalizes moves that fail to block the opponent's winning opportunities, which leads to the opponent's victory.

The details of the outcome-based reward are as follows:

- The winner receives a reward of 1.
- The loser receives a reward of -0.1.

In the case of a draw, since the first player has an advantage, the rewards are different:

- The first player receives a reward of 0.5.
- The second player receives a reward of 1.

The details of the process-based reward are as follows:

- A special reward of 0.8 is given for the first move made by the first player if the move is placed in the center of the board.
- A reward of 0.8 is also given for blocking the opponent from forming a line.

The model also includes a human-computer battle evolution feature. The

computer provides rewards and penalties when playing against human players, which helps in assisting the model training.

State update

The state update follows the formula:

$$S_{(t+1)} \leftarrow S_{(t)} + lr\left(gamma \times reward - S_{(t)}\right)$$

lr is the learning rate, set to 0.3. Tune the learning rate to balance between fast convergence and stable learning. A learning rate that is too high may cause unstable training, while one that is too low may result in slow learning.

gamma is the discount factor, set to 0.9. A lower discount rate focuses more on immediate rewards, while a higher one focuses on long-term planning.

Board state mapping

The 3D board considers that some game states have the same topological structure. For example, a piece located in the top right corner can be moved to the bottom left corner through operations like rotational and reflective symmetry. Unlike traditional hashing, this model uses a more complex state-mapping algorithm:

1. First, it processes rotations and reflections based on the arrays in the board, storing the resulting array list in **symmetric_states**.
2. Then, the arrays in symmetric_states are flattened and lexicographically sorted.
3. The lexicographically smallest array is converted into a string to be used as the key in states_value.

By applying this algorithm, any game states with the same topological structure will generate the same key, greatly improving the model's training efficiency.

```
def rotate_board(self, board, axis):
    return np.rot90(board, k=1, axes=axis)

# Define a function to reflect the board across an axis
def reflect_board(self, board, axis):
    return np.flip(board, axis=axis)

# Generate all symmetric states (rotations and reflections)
def generate_symmetric_states(self, board):
    symmetric_states = []
    axes = [(1, 2), (0, 2), (0, 1)]  # Rotate around different pairs of axes

    for axis in axes:
        for _ in range(4):  # Four 90-degree rotations
            board = self.rotate_board(board, axis)
            symmetric_states.append(board)
            # Reflect along each axis after rotating
            symmetric_states.append(self.reflect_board(board, axis=0))
            symmetric_states.append(self.reflect_board(board, axis=1))
            symmetric_states.append(self.reflect_board(board, axis=2))

    return symmetric_states

# Choose the canonical state
def get_canonical_state(self, board):
    symmetric_states = self.generate_symmetric_states(board)
    # Convert states to lists and find the lexicographically smallest one
    canonical_state = min(symmetric_states, key=lambda x: x.flatten().tolist())
    return canonical_state;

# get unique hash of current board state
def getHash(self):
    self.boardHash = str(self.get_canonical_state(self.board));
```

Exploration mechanism

Fine-tune the exploration strategy. Use epsilon-greedy exploration, where the model explores more at the beginning and gradually reduces exploration (epsilon decay) over time to focus more on exploitation. This step is inspired by the ant colony optimization (ACO) method previously encountered. In ACO, when the pheromone concentration is low in the early stages, ants engage in more random exploration rather than selecting the path with the highest pheromone concentration.

```
if isRandom and np.random.uniform(0, 1) <= (1 - (1 - self.exp_rate) * trainingProgress):
    # take random action
    idx = np.random.choice(len(positions))
    action = positions[idx]
```

In the code shown in the figure, a base random threshold of **exp_rate** = 0.3 is set. **trainingProgress** is a coefficient that represents the training progress, increasing from 0 to 1. Based on the current formula, there is a 100% chance of random selection in the first round. The probability of random exploration decreases with the training progress, eventually reducing to the base random threshold of 30% in the final round.

Analysis of match results

```
Human: O    |      Computer: X
-----------------------
It's p1's trun to play
- - -    - - -    - - -
- - -    - X -    - - -
- - -    - - -    - - -
```

We can see that in this game, the computer played first and effectively executed the strategy of placing the first move in the center.

```
-----------------------
It's human's trun to play
- - -    - - -    - - -
- O -    - x -    - - -
- - -    - - -    - - -
-----------------------
It's p1's trun to play
- - -    - - -    - - -
- o -    - x -    - - -
- - -    - - -    - - X
```

In this round, while the human player blocked the computer player's chance of winning, they also created a potential winning opportunity for themselves, but this was immediately blocked by the computer.

```
-----------------------
It's human's trun to play
O - -    - - -    - - -
- o -    - x -    - - -
- - -    - - -    - - x
-----------------------
It's p1's trun to play
o - -    - - -    - - -
- o -    - x -    - - -
- - X    - - -    - - x
```

In the final round, the human player was unable to block both of the

computer player’s winning opportunities at the same time. The computer player seized the chance and ended the game, achieving victory.

```
----------------------
It's human's trun to play
o - -    - - -    - - -
- o -    - x -    - - -
- - x    - - O    - - x
----------------------
It's p1's trun to play
o - -    - - -    X - -
- o -    - x -    - - -
- - x    - - o    - - x
p1 wins!
```