Assignment 6 - Game Playing Systems

February 27, 2024

1 Group 5 - Module 6: Game Playing Systems

1.0.1 Group Members:

- **Nils Dunlop, 20010127-2359, Applied Data Science, e-mail: gusdunlni@student.gu.se (16 hours)**
- **Francisco Erazo, 19930613-9214, Applied Data Science, e-mail: guserafr@student.gu.se (16 hours)**

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part of other solutions." (This is independent and additional to any declaration that you may encounter in the electronic submission system.)

2 Assignment 6

2.1 Problem 1: Reading and Reflection

AlphaGo is a computer program designed to play the board game Go. It was created by DeepMind Technologies, now a part of Google. Go is known for its complex strategies and the vast number of possible moves, presenting a significant challenge for traditional AI methods. AlphaGo uses advanced deep neural networks and Monte Carlo Tree Search (MCTS), enabling it to learn from a large amount of data, including recorded games between human experts and games it played against itself. Through this learning process, AlphaGo developed judgment and intuition similar to human players, allowing it to accurately predict moves and game outcomes.

The architecture of AlphaGo includes policy networks that suggest probable next moves, and value networks that predict the game's winner from any position, marking a step forward in using AI to tackle complex problems. Its training involved supervised learning from games played by human experts and reinforcement learning from self-play. This approach enabled the system to continuously refine its strategies and adjust to new challenges. The integration of MCTS with neural networks enabled efficient exploration and evaluation of possible moves, balancing between relying on known effective strategies and exploring new ones.

AlphaGo's success was demonstrated by its 99.8% win rate against other Go programs and its historic victory over European Go champion Fan Hui 5-0, marking the first time a computer program defeated a professional human player in Go. Given Go's complexity compared to chess, this achievement was seen as a significant milestone in AI.

The strategies employed by AlphaGo, and its implications, extend beyond just games. They showcase the potential of deep learning and reinforcement learning to address complex issues in various fields.

2.2 Problem 2: Implementation

Tic-Tac-Toe Game Board and Rules

[19]:
```
import math
import random

class TicTacToe:
    def __init__(self):
        # Initialize a 3x3 Tic Tac Toe board
        self.board = [[' ' for _ in range(3)] for _ in range(3)]
        self.current_turn = 'X'  # Start with player 'X'
        self.game_over = False
        self.winner = None

    def move(self, row, col):
        # Place a move on the board if the cell is empty
        if self.board[row][col] == ' ':
            self.board[row][col] = self.current_turn
            self.check_winner()  # Check for a winner after the move
            self.current_turn = 'O' if self.current_turn == 'X' else 'X'  #␣
↪Switch turns
            return True
        return False

    def check_winner(self):
        # Check all win conditions (rows, columns, diagonals)
        for i in range(3):
            # Check rows and columns
            if self.board[i][0] == self.board[i][1] == self.board[i][2] != ' '␣
↪or \
                    self.board[0][i] == self.board[1][i] == self.board[2][i] !=␣
↪' ':
                self.game_over = True
                self.winner = self.board[i][0]
                return
        # Check diagonals
        if self.board[0][0] == self.board[1][1] == self.board[2][2] != ' ' or \
                self.board[0][2] == self.board[1][1] == self.board[2][0] != ' ':
            self.game_over = True
            self.winner = self.board[1][1]
            return
        # Check for a draw (no empty spaces left)
        if all(self.board[row][col] != ' ' for row in range(3) for col in␣
↪range(3)):
            self.game_over = True

    def get_legal_moves(self):
        # Return a list of all empty spaces on the board
        return [(row, col) for row in range(3) for col in range(3) if self.
↪board[row][col] == ' ']

    def clone(self):
        # Create a copy of the game state
        clone = TicTacToe()
        clone.board = [row[:] for row in self.board]
        clone.current_turn = self.current_turn
        clone.game_over = self.game_over
        clone.winner = self.winner
        return clone

    def print_board(self, generation=None):
        # Print the current state of the board
        if generation is not None:
            print(f"Board State at Generation {generation}:")
        for row in self.board:
            print(' ' + ' | '.join(row))
            print('---+---+---')
        print()

    def count_two_in_a_rows(self, player):
        # Count the number of two-in-a-rows for a given player
        opponent = 'O' if player == 'X' else 'X'
        two_in_a_rows = 0
        # Check rows, columns, and diagonals for two-in-a-rows
        for i in range(3):
            if self.board[i].count(player) == 2 and self.board[i].
↪count(opponent) == 0:
                two_in_a_rows += 1
            column = [self.board[j][i] for j in range(3)]
            if column.count(player) == 2 and column.count(opponent) == 0:
                two_in_a_rows += 1
        diagonals = [[self.board[i][i] for i in range(3)], [self.board[i][2-i]␣
↪for i in range(3)]]
        for diag in diagonals:
            if diag.count(player) == 2 and diag.count(opponent) == 0:
                two_in_a_rows += 1
        return two_in_a_rows

    def evaluate_state(self, player):
        # Evaluate the board state for a given player
        if self.winner == player:
            return 100
        elif self.winner is not None:
            return -100
        else:
            score = self.count_two_in_a_rows(player) * 10
            # Adjust score based on opponent's two-in-a-rows
            opponent = 'O' if player == 'X' else 'X'
            opponent_two_in_a_rows = self.count_two_in_a_rows(opponent)
            if opponent_two_in_a_rows > 0:
                score -= opponent_two_in_a_rows * 50
            return score

    def check_immediate_threat(self, player):
        # Check if there's an immediate win available for the opponent
        opponent = 'O' if player == 'X' else 'X'
        for row in range(3):
            for col in range(3):
                if self.board[row][col] == ' ':
                    temp_board = self.clone()
                    temp_board.board[row][col] = opponent
                    temp_board.check_winner()
                    if temp_board.winner == opponent:
                        return True
        return False
```

Monte Carlo Search Node

[20]:
```
class MCTSNode:
    def __init__(self, game_state, parent=None, move=None):
        self.game_state = game_state
        self.parent = parent
        self.move = move
        self.children = []
        self.wins = 0
        self.visits = 0
        self.untried_moves = game_state.get_legal_moves()

    def UCB1(self, exploration_parameter):
        # Calculate the Upper Confidence Bound for tree node selection
        if self.visits == 0:
            return float('inf')
        # Balances exploration and exploitation.
        return self.wins / self.visits + exploration_parameter * math.sqrt(2 *␣
↪math.log(self.parent.visits) / self.visits)

    def select_child(self, exploration_parameter=2):
        # Select a child node using the UCB1 formula
        return max(self.children, key=lambda child: child.
↪UCB1(exploration_parameter))

    def add_child(self, move, game_state):
        # Create a new child node for a given move and game state.
        child = MCTSNode(game_state=game_state.clone(), parent=self, move=move)
        self.untried_moves.remove(move)
        self.children.append(child)
        return child

    def update(self, result):
        # Update this node's win/visit statistics based on simulation result.
        self.visits += 1
        if result == self.game_state.current_turn:
            self.wins += 1  # Win for the current player.
        elif result == 'Draw':
            self.wins += 0.5  # Half-win for a draw.
```

[21]:
```
def evaluate_board_for_rollout(board, player):
    # Initial variables
    opponent = 'O' if player == 'X' else 'X'
    center = (1, 1)
    corners = [(0, 0), (0, 2), (2, 0), (2, 2)]
    edges = [(0, 1), (1, 0), (1, 2), (2, 1)]

    # Check for immediate wins or blocks
    for move in board.get_legal_moves():
        temp_board = board.clone()
        temp_board.move(*move)
        if temp_board.winner:
            return move, True  # Immediate win or block

    # Take the center if available
    if board.board[center[0]][center[1]] == ' ':
        return center, True

    # Take a corner, prioritizing those adjacent to opponent's marks to disrupt␣
↪their strategy
    available_corners = [corner for corner in corners if board.
↪board[corner[0]][corner[1]] == ' ']
    if available_corners:
        return random.choice(available_corners), True

    # Take an edge as a last resort
    available_edges = [edge for edge in edges if board.board[edge[0]][edge[1]]␣
↪== ' ']
    if available_edges:
        return random.choice(available_edges), True

    return None, False

def rollout_policy(board, player):
    # Use a rollout policy to select moves
    move, found = evaluate_board_for_rollout(board, player)
    if found:
        return move
```

Monte Carlo Tree Search Algorithm

```
[22]: def MCTS(root_state, iterations, exploration_parameter=2):
    root_node = MCTSNode(game_state=root_state)

    for _ in range(iterations):
        node = root_node
        state = root_state.clone()

        # Selection with tweaked exploration parameter
        while node.untried_moves == [] and node.children != []:
            node = node.select_child(exploration_parameter)

        # Expansion
        if node.untried_moves:
            move = random.choice(node.untried_moves)
            state.move(*move)
            node = node.add_child(move, state)

        # Simulation with improved rollout policy
        while state.get_legal_moves():
            move = rollout_policy(state, state.current_turn)
            state.move(*move)

        # Backpropagation with nuanced scoring
        while node is not None:
            node.update(state.evaluate_state(node.game_state.current_turn))
            node = node.parent

    return sorted(root_node.children, key=lambda c: c.visits)[-1].move
```

Playing the Game

[23]:
```
def play_game():
    game = TicTacToe()
    generation = 0

    # Play the game until it's over
    while not game.game_over:
        # AI's turn
        if game.current_turn == 'X':
            move = MCTS(game.clone(), iterations=1000)
            print(f"AI makes move at {move}:")
            game.move(*move)
            generation += 1
            game.print_board(generation)
        # Opponent's turn
        else:
            possible_moves = game.get_legal_moves()
            if possible_moves:
                move = random.choice(possible_moves)
                print(f"Opponent makes move at {move}:")
                game.move(*move)
                generation += 1
                game.print_board(generation)
            else:
                print("No legal moves available. Game over.")
                break

    # Announce the game result.
    if game.winner:
        print(f"Game over. Winner: {'AI' if game.winner == 'X' else␣
↪'Opponent'}")
    else:
        print("Game over. It's a draw.")

play_game()
```

```
AI makes move at (1, 1):
Board State at Generation 1:
   |   |
---+---+---
   | X |
---+---+---
   |   |
---+---+---

Opponent makes move at (0, 1):
Board State at Generation 2:
   | O |
---+---+---
```

```
   | X |
---+---+---
   |   |
---+---+---

AI makes move at (0, 2):
Board State at Generation 3:
   | O | X
---+---+---
   | X |
---+---+---
   |   |
---+---+---

Opponent makes move at (0, 0):
Board State at Generation 4:
 O | O | X
---+---+---
   | X |
---+---+---
   |   |
---+---+---

AI makes move at (2, 0):
Board State at Generation 5:
 O | O | X
---+---+---
   | X |
---+---+---
 X |   |
---+---+---

Game over. Winner: AI
```

- **Roll-Out Policy:**
  For the roll-out policy, which determines the moves during the simulation phase of MCTS, we decided for a simplified heuristic approach. This approach prioritizes moves that can lead to immediate wins or blocks, captures the center if available, select for corners to potentially create multiple winning paths, and finally selects edges. This hierarchy mirrors common Tic-Tac-Toe strategies where control of the center and corners can lead to multiple winning paths and block opponent's opportunities. This approach is fast and ensures that the simulation phase doesn't randomly move through all possible game states, which improves both speed and relevance of the simulations. Additionally, it reduces the variance in the simulation outcomes, making the simulations more meaningful and the AI more competitive, as it simulates somewhat rational play instead of random moves.

- **Opponent Policy:**
  The opponent's strategy in this implementation is deliberately simplified, where the opponent chooses moves randomly. This approach was chosen to keep the game's logic straightforward

and manageable, especially for initial iterations. It allows us to focus on the AI development without the complexities of a sophisticated opponent logic. While this method does not simulate a strategic human player, it serves as a foundation for more complex algorithms in the future. Specifically, in subsequent iterations or given more time, we plan to explore implementing a feature allowing the user to play against the AI and potentially integrating a minmax algorithm. This will enhance the game by preparing the AI for a broader range of opponent behaviors, including both predictable and highly strategic human play.

- **Selection Policy:**
  The selection policy utilizes the Upper Confidence Bound (UCB1) formula, which balances the exploration of less-visited nodes with the exploitation of nodes with a high win ratio. This balance is crucial because it prevents the algorithm from over-focusing on early successful paths (exploitation) and encourages the discovery of potentially better paths that have not been explored thoroughly (exploration). The UCB1 formula's efficiency lies in its mathematical foundation, which statistically ensures that given enough time, the algorithm will explore all parts of the tree adequately. This method is particularly well-suited for the finite and discrete nature of Tic-Tac-Toe's game space.

- **Updates to the Search Tree (Back-up):** The back-up step is where the results of the simulations are propagated back up the tree to update the statistics of the nodes. This process is implemented through the `update method` in the `MCTSNode class`. This process ensures that the nodes reflect the latest information about the likelihood of winning from that position. The update method adjusts the wins and visits based on the outcome (win, draw, or loss), which influences future selections and roll-outs. This feedback loop, facilitated by a scoring mechanism, is what allows the AI to learn and adapt its strategy over time. This approach is particularly effective in games with clear win/loss outcomes at each node. However, the binary nature of the updates (win/loss with a simplistic scoring mechanism) might oversimplify complex strategic considerations in games more complex than Tic-Tac-Toe.

- **Sampling in MC:**

  1. Selection Phase: During the selection phase, the algorithm uses the Upper Confidence Bound 1 (UCB1) formula to select nodes for exploration. This is not random sampling per se but a strategic selection process that balances exploration and exploitation based on the outcomes of previous simulations. The exploration parameter in the UCB1 formula influences the degree of randomness in selection: a higher value increases the emphasis on exploration (thus introducing more variability or "sampling" of less-visited nodes), whereas a lower value emphasizes exploitation of known good paths.

  2. Expansion Phase: When a new move is explored, and a node is added to the tree, this expansion is based on the available legal moves from the current game state. The choice of which move to explore next could be considered a form of sampling, especially when the selection is made randomly from the set of untried moves, as is the case here.

  3. Simulation Phase (Rollout): This is where the core of the sampling occurs. In the simulation phase, the game is played out to the end (a win, loss, or draw) from the current state using a predefined policy. The rollout_policy function dictates this process, but instead of purely random moves, it uses a heuristic-based strategy that mimics rational play: checking for immediate wins or blocks, prioritizing center, corners, and then edges. This method still represents a form of sampling but with a bias towards strategic moves.

It's a deviation from traditional Monte Carlo sampling, which would involve completely random moves. The choice to use a heuristic approach is a design decision to increase the efficiency and relevance of the simulations by reflecting more realistic game scenarios.

4. Backpropagation Phase: After each simulation, the results (win, loss, draw) are propagated back up the tree, updating the statistics (wins, visits) of the nodes along the path taken. This step doesn't involve sampling but affects future sampling decisions by altering the UCB1 scores based on new information, thus guiding where the algorithm samples next.

- **Evaluation of the Algorithm:**
  - **Pros**:
    - Adaptability: MCTS does not require a comprehensive understanding of the game's strategy, making it versatile for different games, including Tic-Tac-Toe.
    - Competitiveness: The algorithm can play at a competitive level, especially with a well-implemented roll-out policy and sufficient iterations for exploration and exploitation. For casual players or those without a clear strategy, the algorithm could prove to be challenging due ot its strategic roll-out policy and ability to learn and adap over the course of the game.
    - No Need for a Predefined Strategy: It discovers optimal moves through exploration and exploitation, reducing the need for hardcoded strategies.
    - Fast and Efficient: For a relatively simple game like Tic-Tac-Toe, this MCTS implementation can be considered fast, especially since the decision space is limited. However, the speed could decrease as the number of iterations increases to ensure more accurate and competitive gameplay. For real-time decision-making in Tic-Tac-Toe, the algorithm is sufficiently fast, but for more complex games or larger grids, performance optimizations might be necessary.
  - **Cons**:
    - Computational Demands: The need for numerous simulations for each move decision can be computationally expensive, especially as the game progresses and possibilities decrease.
    - Sample Efficiency: MCTS may require many iterations to converge on an effective strategy, particularly against diverse or unpredictable opponents, which is not sample efficient.
    - Scalability: Scaling MCTS to larger grids (e.g., 4x4, 5x5) increases the computational complexity exponentially, making it less practical for more extensive games without optimization.
    - Predictability: The algorithm's strategy might become predictable, especially if the roll-out policy is not sufficiently complex or if the exploration parameter is not well-tuned. This means that it can lose. A human player could beat or draw against this algorithm by using a good strategy.

2.3 References

- Choudhary, A. (2018). Reinforcement Learning Guide: Solving the Multi-Armed Bandit Problem from Scratch in Python. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/ [Accessed 23 Feb. 2024].
- Nested Software (2019). Tic-Tac-Toe with MCTS. [online] DEV Community. Available at: https://dev.to/nestedsoftware/tic-tac-toe-with-mcts-2h5k [Accessed 26 Feb. 2024].
- GfG (2019). ML Monte Carlo Tree Search (MCTS). [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/ [Accessed 26 Feb. 2024].

2.4 Self Check

- Have you answered all questions to the best of your ability? Yes, we have.
- Is all the required information on the front page, is the file name correct etc.? Indeed, all the required information on the front page has been included.
- Anything else you can easily check? (details, terminology, arguments, clearly stated answers etc.?) We have checked, and everything looks good.