UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Design of AI Systems

Assignment 6: Game playing systems

Group 2:

Tim Boleslawsky, 980524-6478, Applied Data Science Master's, gusbolesti@student.gu.se
Mikołaj Nowacki, 20010918-2014, Applied Data Science Master's, gusnowami@student.gu.se

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.

1. Reading and reflection

The paper starts by discussing the two main problems we face when tackling game playing systems with AI. These problems mirror the problems we have discussed in this weeks lecture and can be summarized like this:

- State Space and Feasibility: The number of possible sequences of moves in a game (e.g., chess or Go) grows exponentially with depth and breadth, making exhaustive search infeasible. Possible solution: Reducing breadth by sampling actions from a probability distribution rather than considering all possible moves. We can for example use sampling techniques like Monte Carlo rollouts or policy-based methods to handle large state spaces.

- Storage Limitations: Computing and storing every possible outcome is infeasible. Possible solution: Truncating search depth with an approximate value function (deep learning), to reduce the number of states considered.

There exist a lot of popular systems based on MCTS. But they have some limitations, especially in large search spaces: Inefficient Exploration – The systems treat all moves equally at first and only refines values after many rollouts. Lack of Generalization – Every game state is evaluated from scratch, meaning no learning happens across different games. Purely Random Rollouts – The game outcome is estimated based on random moves rather than intelligent play.

AlphaGo improves MCTS in two key ways:

- Step 1: Selection (Improved by the Policy Network). In standard MCTS, selection is based on UCT, which balances exploration vs. exploitation. With deep learning, a policy network is used to bias the selection toward promising moves, instead of treating all moves equally at first. This helps focus the search on good moves from the start, rather than needing many rollouts to refine the estimates.

- Step 3: Simulation (Improved by the Value Network). In standard MCTS, simulation (rollouts) plays out a game randomly until the end, which can be noisy and inefficient. In deep learning, the value network directly estimates the probability of winning from a given state, eliminating the need for full random rollouts. This makes the search much faster because instead of playing thousands of games per move, the value network provides a strong estimate instantly.

This means that the main goal of AlphaGo's training pipeline is to develop the policy network and the value network. The policy network determines which moves are likely to be strong and is used by supervised learning (trained on human expert moves) and reinforcement learning (improved through self-play). The value network evaluates board positions to estimate the probability of winning from a given state.

We now discuss the training pipeline in more detail. The first stage of the training pipeline was the supervised learning of the policy network. The goal of this stage was to train a policy network that predicts human expert moves based on historical game data. For this, the input is a board position (s), represented as a 19×19 image with different features such as stone locations, move history, and player turns. The network

structure consists of 13 convolutional layers with rectifier nonlinearities, followed by a softmax layer that outputs a probability distribution over all legal moves. The training method used is supervised learning with stochastic gradient ascent, optimizing the likelihood of choosing the same move as a human expert. Through this, AlphaGo achieved a 57.0% accuracy in predicting expert moves, which is much better than previous models.

The second stage was the reinforcement learning of the policy network. Here the goal was to improve the policy network by training it to maximize its win rate through self-play rather than just copying human moves. For this, the RL policy network starts with the same architecture and weights as the SL policy network. Now the network plays games against itself, selecting moves based on its current policy. A pool of previous policy iterations is maintained, and new versions play against older ones to stabilize training and prevent overfitting. The network is trained using a reward system, where the reward for winning is +1 and the reward for losing is -1. The goal of the model is to maximize this reward. With this strategy, the RL policy network won 80% of games against the SL policy network (showing it had improved).

The third and last stage aims to develop the value network. The goal was to directly predicting the probability of winning from a given position instead of relying on Monte Carlo rollouts (which require playing out a full game). The network architecture was similar to the policy network but outputs a single value instead of a probability distribution. The value network learns an approximate value function: $v_\theta(s) \approx v^(s) \approx v_\rho(s)$, where $v^(s)$ is the ideal value function under perfect play, and $v_\rho(s)$ is the estimated value function under the RL policy. With this strategy the value network approached the accuracy of Monte Carlo rollouts but used 15,000x less computation.

The paper now goes on to describe how the model actually plays. We will express this explanation in the structure we learned in this weeks lecture for better clarity.

- Selection: AlphaGo starts from the root state and descends the search tree by choosing the action that maximizes: $a_t = \arg\max_a(Q(s_t, a)+u(s_t, a))$, where $Q(s, a)$ is the estimated action value from past simulations, $u(s, a)$ is the exploration term that depends on visit count and prior probability and the prior probability $P(s, a)$ is given by the SL policy network.
- Expansion: When the search reaches a state that hasn't been visited before, it is expanded. The SL policy network provides prior probabilities $P(s, a)$ for legal moves.
- Simulation: Each new state is evaluated in two ways. First via the value network which predicts the probability of winning from this position. Second via a rollout policy, which Simulates a game using a fast, less accurate policy until a terminal state is reached, giving an actual game outcome. The two evaluations are then combined.
- Backpropagation: The simulation result is propagated back up the tree, updating visit counts and action values.
- Lastly after running many simulations, the algorithm selects the move that was visited the most times in the root state. This ensures the most reliable move is chosen, rather than just the one with the highest estimated value.

Through the training and playing description the paper explained how the standard MCTS models could be improved by introducing deep learning networks and building upon the principles of MCTS.

2. Implementation

2.1 Description

The overall structure of our script is divided into two classes. The first class (TicTacToe) is the structure of the game and the second class (MCTSNode) represents a node in a MCTS.

2.1.1 Roll-out policy of the player

The roll-out policy for the player is random. When selecting a node of an untried move, we simulate the game from this node by selecting random moves of the available moves until the game is finished. This provides us with a really easy way to check if this node leads to a victory or defeat. But we have to mention that that is obviously not the moste reliable or efficient implementation.

2.1.2 Policy of the opponent

The policy of the opponent is completely random. This means that when it's the opponents turn, we look at the available moves and take one at random. This gives us a good baseline to test our model against.

2.1.3 Selection policy in the search tree

For the selection policy we chose the UCT formula.

$$\frac{w_i}{n_i} + c\sqrt{\frac{lnN_i}{n_i}} \tag{1}$$

w_i - the number of wins for the node considered after the i-th move
n_i - the number of simulations for the node considered after the i-th move
N_i - the total number of simulations after the i-th move run by the parent node of the one considered
c - the exploration parameter—theoretically equal to $\sqrt{2}$

In python we implement this formula like this:

```
max(self.children, key=lambda c: c.value / (c.visits + 1e-6) +
    1.41 * np.sqrt(np.log(self.visits + 1) / (c.visits + 1e-6)))
```

- c.value / (c.visits + 1e-6) = this is the exploitation term. We add a small number to ensure we don't divide by zero.
- 1.41 is the exploration constant.
- np.sqrt(np.log(self.visits + 1) / (c.visits + 1e-6))) = this is the exploration term. We again add a small number to ensure we don't divide by zero and add 1 to each visit to ensure we prevent log(0).

We chose UCT because an easy to implement method that balances exploitation and exploration making it more efficient than greedy methods.

2.1.4 Updates to the search tree ("back-up")

The updates to the search tree in the backpropagation are based on the idea that we "reward" a node with 1 and punish a node with "-1" based on if this node lead to a win or not.

2.2 Evaluation

To get a basic understanding of how good our MCTS performs, we simulate 100 games against the random opponent. We found that the AI wins 79 times, the opponent wins 19 times and we have 2 draws. So our MCTS is by no means unbeatable but it is better than a purely random player.

What we also found surprising is how long simulating these 100 games took. We usually needed about 40 seconds to simulate 100 games of something as easy as tic tac toe. This leads us to believe that this algorithm is not very efficient. We already talked about the random selection process, which leads to inefficiency, but we also need mention, that we don't take any prior domain knowledge into account or use tree pruning to reduce the state space. Our algorithm shines with the combination of simplicity and the combination of exploration and exploitation by UCT, making it a relatively successful but easy implementation. Having said that, this relatively simple and non-heuristic implementation would not scale well to 4x4 or 5x5 grids. The larger the state space the more obvious the poor performance and inefficient/ simple implementation would be.

When trying to play against the algorithm ourselves, we were surprised at how hard it was to actually win. It was not rare that one of us needed two or three tries to win against the algorithm, which is definitely more than we expected.

Code

Code for implementation of classifier:

```
import numpy as np
import random
from collections import defaultdict

class TicTacToe:
    def __init__(self):
        self.board = np.zeros((3, 3), dtype=int)  # 0 = empty, 1 = X, -1 = O
        self.current_player = 1  # X starts

    def get_available_moves(self):
        return [(r, c) for r in range(3) for c in range(3) if self.board[r, c] == 0]

    def make_move(self, move):
        r, c = move
        self.board[r, c] = self.current_player
        self.current_player *= -1  # Switch players

    def undo_move(self, move):
        r, c = move
        self.board[r, c] = 0
        self.current_player *= -1

    def check_winner(self):
        for line in np.vstack([self.board, self.board.T, [self.board.diagonal(), np.fliplr(self.board).diagonal()]]):
            if np.all(line == 1): return 1  # X wins
            if np.all(line == -1): return -1  # O wins
        if not self.get_available_moves(): return 0  # Draw
        return None  # Game not over

    def is_terminal(self):
        return self.check_winner() is not None

    def clone(self):
        new_game = TicTacToe()
        new_game.board = self.board.copy()
        new_game.current_player = self.current_player
        return new_game

class MCTSNode:
    def __init__(self, state, parent=None, move=None):
        self.state = state.clone()
```

```
42          self.parent = parent
43          self.move = move
44          self.children = []
45          self.visits = 0
46          self.value = 0
47          self.untried_moves = state.get_available_moves()
48
49      def select_child(self):
50          return max(self.children, key=lambda c: c.value / (c.
                visits + 1e-6) + 1.41 * np.sqrt(np.log(self.visits +
                1) / (c.visits + 1e-6)))
51
52      def expand(self):
53          move = self.untried_moves.pop()
54          new_state = self.state.clone()
55          new_state.make_move(move)
56          child = MCTSNode(new_state, parent=self, move=move)
57          self.children.append(child)
58          return child
59
60      def backpropagate(self, reward):
61          self.visits += 1
62          self.value += reward
63          if self.parent:
64              self.parent.backpropagate(-reward)
65
66      def best_move(self):
67          return max(self.children, key=lambda c: c.visits).move
                if self.children else random.choice(self.state.
                get_available_moves())
68
69      def simulate(self):
70          temp_state = self.state.clone()
71          while not temp_state.is_terminal():
72              temp_state.make_move(random.choice(temp_state.
                    get_available_moves()))
73          return temp_state.check_winner() or 0
74
75      def run_mcts(self, iterations=1000):
76          for _ in range(iterations):
77              node = self
78              while node.children and not node.untried_moves:
79                  node = node.select_child()
80              if node.untried_moves:
81                  node = node.expand()
82              reward = node.simulate()
83              node.backpropagate(reward)
84          return self.best_move()
```

```
85
86  # Playing the game against the AI
87  game = TicTacToe()
88  while not game.is_terminal():
89      if game.current_player == 1:
90          print("AI (X) is thinking...")
91          root = MCTSNode(game)
92          move = root.run_mcts(1000)
93          print(type(move))
94      else:
95          move = tuple(map(int, input().split()))
96          print(move)
97      game.make_move(move)
98      print(game.board, "\n")
99
100 print(game.check_winner())
101
102 # Simulating against a random opponent
103 def simulateGame():
104     game = TicTacToe()
105     while not game.is_terminal():
106         if game.current_player == 1:
107             root = MCTSNode(game)
108             move = root.run_mcts(1000)
109         else:
110             move = random.choice(game.get_available_moves())  #
                Random opponent
111         game.make_move(move)
112
113     return game.check_winner()
114
115 winners = []
116 for i in range(1,101):
117     print(i)
118     winners.append(simulateGame())
119
120 print("Times AI has won: ", winners.count(1))
121 print("Times opponent has won: ", winners.count(-1))
122 print("Times a draw happened:", winners.count(0))
```