Design of AI Systems (DAT410)
Assignment 6: Game-playing systems

Luca Modica
Hugo Manuel Alves Henriques e Silva

February 27, 2024

1 Reading and reflection

The paper "Mastering the game of Go with deep neural networks and tree search", a work by David Silver et al., presents a new innovative approach of training an AI in the field of board games, based on Deep Neural Networks and Tree Search. This work in particular focuses on the game of Go: It represents a well-known game in the field of Artificial Intelligence, mainly due to its profound strategy depth and the large state space with all of the potential moves in the board.

In this specific domain, previous AI heavily relied on look-ahead search strategies such as Monte Carlo Tree Search (MCTS), with the purpose of simulating many possible sequences of moves based on a default policy to make a decision; but this approach usually requires a lot of computational power and is not still effective in a game as complex as Go. This is where the central innovation of "AlphaGo" comes in. By using Deep Neural networks and reinforcement learning, the AI will first use the *"value networks"* to evaluate board positions and the *"policy networks"* to select moves. Then, these insights will be used with the MCTS algorithm to greatly reduce the size of the search tree and focus on the most promising moves.

The development of AlphaGo is divided into the following steps.

1. **Supervised learning from policy networks**: based on a dataset of millions of Go positions (from the *KGS* Go Server), the policy network is trained on sampled state-action pairs (s, a) using gradient ascent. This is done to maximize the likelihood of a human player choosing a from a state s.

2. **Reinforcement learning of policy networks**: the policy network is improved even more with reinforcement learning. The idea is to let play games between a current policy and a randomly selected one from previous

iterations of the policy network; the neural network weight values will be then updated based on the reward function and by using gradient ascent.

3. **Reinforcement learning of value networks**: considered as the final stage of the model training, here the focus is to evaluate the position of the player. This by estimating a value function that predicts the outcome from a specific position, using a policy for both players and a value neural network.

4. **Searching with policy and value networks**: the lookahead search part. Monte Carlo Tree Search algorithm is used to search for optimal action starting from a specific state, by repeated roll-outs and backup strategy. The selection of the leaf node to expand is based on a combined evaluation of the value network and the outcome of a random roll-out.

The performance of AlphaGo, measured with an internal tournament among the strongest Go programs of that period (such as Pachi or Fuego), was impressive enough to mark this work as a monumental step forward in the AI field. The combination of Deep Neural Networks and MCTS allows to reach the level of the strongest human players, as well as showing possible potential advances in domains similar to game-playing systems: classical planning, scheduling, and so on [1].

2 Implementation

In this section, we will describe our approach to implementing an unbeatable AI for the classic Tic-Tac-Toe game, as well as adapted versions for greater scale grids such as 4x4 or 5x5. To let the AI learn how to win, our approach is mainly based on a Monte Carlo Tree Search (MCTS), which will keep track of the number of visits and win of every expanded node.

2.1 TicTacToe Class

First of all, we started by implementing the Tic-Tac-Toe game board which will represent the state of the game throughout the whole process. This class consists of the following simple methods:

- **print_board**: creates a visual representation of the game board to be printed out to the user.
- **make_move**: after the player or the AI choose a move, this method either updates the board or, in case the move chosen was illegal, prints out a warning message.
- **get_winner**: returns the winner of the game, or *None* if a draw occurred.
- **is_game_over**: checks if the current game has ended.
- **clone**: creates and returns a copy of the current game state.
- **get_legal_moves**: returns all the possible moves for a player.
- **get_random_move**: if there is a winning move, returns it, otherwise returns a random move.

2.2 MCTSNode Class

This class represents a Monte Carlo Tree Search Node. Trees will consist of multiple of these objects.

2.2.1 Attributes

The following correspond to a node's attributes:

- **game_state**: represents the state of the game at the current node. It is important to note that each node of a tree represents a different game state. A game's state is only altered when a move is made, but for that, a new node is created.
- **parent**: represents the parent of the current node. If the node corresponds to the root node, then it has no parent.
- **move**: represents the move made which led to the current state of the node.

- **children**: represents the children of the current node. A child corresponds to a new node which results from a move being made on the current state of the game. The children of a node correspond to the nodes formed by all the possible plays out of a state.
- **visits**: represents a counter of how many times the algorithm has gone through this specific node.
- **wins**: represents the number of times that going through this move led to a win. It is relevant to note that if going through this node leads to a loss this attribute's value is decremented, meaning that it can take negative values.

2.2.2 Methods

Now, moving on to the methods of the class:

- **add_child**: adds a child (new MCTS node) to the list of children of the current node.
- **select_child**: among all of the children of a node, selects a child according to the Upper Confidence Tree (UCT) score, which essentially is the node that maximizes the score. The usage of this **selection policy** is a strategy to balance exploration and exploitation when selecting moves to simulate in decision-making processes. The UCT score helps to determine which node (or move) in the game tree to explore next.
- **expand**: adds a child node to the current node for all the possible moves that can be made from the current state.
- **simulate**: simulates the rest of the game from its current state. The roll-out policy used is the one described in the TicTacToe class's *get_random_move* method, which, if there is a winning move, selects it, otherwise selects a random legal move. The game is played until a terminal state is reached.
- **backpropagate**: updates the *wins* and *visits* attributes of a branch of the tree. Recursively calls this same function from leaf until the root node is reached through the *parent* attribute. All the nodes encountered along the way that correspond to the winner's moves have their attribute *wins* incremented, and the nodes that correspond to the loser's moves have the same attribute decremented. All nodes have their attribute *visits* incremented.

2.3 Running the Algorithm

We run the algorithm a chosen number of iterations from the given root node.

Firstly, it chooses a node from the root's children according to our selection policy described above in the MCTSNode's *select_child* method. After that, a check to see if the chosen node corresponds to a terminal state is made. If it does, we do not expand it and save this node as the "*selected_node*". Otherwise, it is expanded and, from the just now expanded node, a new child node is chosen with the same selection policy and saved as the "*selected_node*". From the *selected_node* a game simulation is made. Finally, with this newly obtained result, the back propagation method is called upon this *selected_node* and the branch's node's wins and *visits* are updated according to what was described in the backpropagate method in the MCTSNode class.

This process described above is run for the chosen number of iterations.

2.4 Playing the Game

To put our implementation to the test, we will now describe how a game is played:

To start things out, a game board is created, with the first player being the user playing it. For visual purposes, every time a play is made the board is printed. When it is the user's turn, the process is straightforward, where the user only needs to select a legal move. When it is the AI's turn, a new root node is created with the current game state. From this node, we will run the Monte Carlo Tree Search algorithm described in the previous section a given number of times, which is set to 10000 by default. Until the game is finished, we will alternate between a player's and the AI's turn.

2.5 Results and discussion

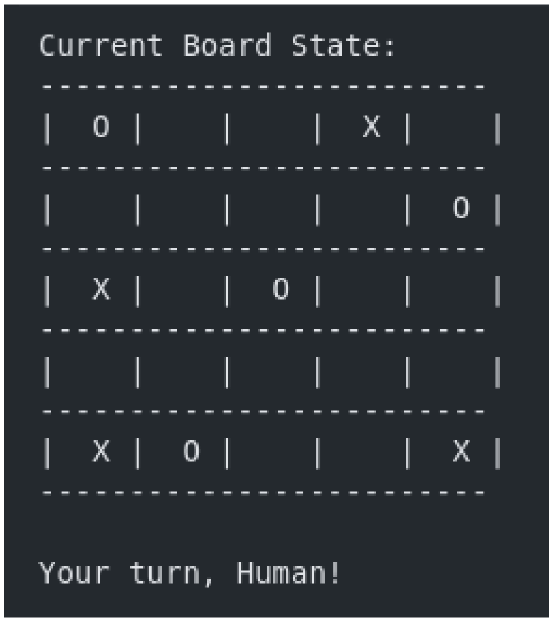
Our implementation of the MCTS algorithm described above was evaluated with a series of matches with human players, in 3 different grid sizes: 3x3, 4x4 and 5x5. For each turn of each match, it will be measured the computational time of the MCTS algorithm to let the AI decide the next move: the goal is to have a measurement of how scalable is our implementation. Other aspects are based on our heuristic observations and domain knowledge expertise.

The following are the results of 3 matches for each grid size taken into consideration. Note that the change in the behavior of the AI will be discussed the more the grid will become larger (that is, the state space increased).

- With the classic size of a game of TicTacToe (3x3), the algorithm seems to be very efficient in terms of computation and sampling: the average measured computational time is 0.25 seconds, which is more than reasonable for a normal game.
- By increasing the size to 4x4, we can start noticing a slightly bigger computational expense and slowness in learning a winning strategy, with 1.8 seconds as the average time to make a move. Despite the mentioned drawbacks, a larger state space also highlights the competitiveness of the policy learned by the algorithm: from our observations, especially in the

first turns, the AI seems to be more focused on blocking the opponent rather than planning how to win the game.

- The last and largest size of the board (5x5) both highlights more the mentioned drawbacks and the competitiveness of the AI. The average computational time rises to 7.6 seconds; even though the time tends to decrease the more we advance into the game due to the more limited search space, it could start to represent a serious issue for the future scalability of the algorithm to more complex games. But despite the cons of the implementation, the AI still keeps its aggressiveness in blocking the player, especially in the first turns (Figure 1).



```
Current Board State:
-------------------------
| O |   |   | X |   |
-------------------------
|   |   |   |   | O |
-------------------------
| X |   | O |   |   |
-------------------------
|   |   |   |   |   |
-------------------------
| X | O |   |   | X |
-------------------------

Your turn, Human!
```

Figure 1: Early state of a game in a 5x5 grid. Despite the AI being slow with a state space of this dimension, its main priority is to immediately block almost any opponent's attempt to win.

All in all, the performance and the results of the algorithm meet a satisfactory result: the AI rarely loses, and overall It seems to learn very fast also thanks to a sampling strategy that prioritizes the winning states. As a future development, we could address the encountered issues with a similar approach to the development of AlphaGo: that is, first learn a prior to be used in a tree search, such as value and policy networks from deep neural networks by using both Supervised and Reinforcement Learning.

References

[1] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.