Assignment 6
Module 6

Design of AI Systems
Group 15

Game playing Systems

Patricia Reichhelm, Stavros Chatzistavros
patricia.reichhelm@uni-ulm.de, T727-981126, ERASMUS program
gusstavrch@student.gu.se, 950418-0234, Mathematical Sciences N2MAT, GU

February 28, 2023

Abstract

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.
Number of hours spent: 15 for each of us

Part I: Gameplaying Systems

1. Reading and Reflection

by Stavros Chatzistavros

The paper describes a new approach to developing artificial intelligence capable of playing the ancient board game of Go at a world-class level.
The researchers used a combination of deep neural networks and Monte Carlo tree search algorithms to build their system, which they named AlphaGo. AlphaGo was trained on a large dataset of expert-level Go games and was then able to learn to play the game at a superhuman level by playing against itself.
The neural network component of AlphaGo was trained to predict the moves that expert players would make in different game situations, while the tree search component was used to explore different possible moves and outcomes in a given game state.
The system was first tested against other computer Go programs and was found to be significantly stronger than any previous program. It was then tested against human players, including the world champion Lee Sedol, and won four out of five games in a best-of-five match.
The success of AlphaGo demonstrated the potential of deep neural networks and machine learning algorithms to solve complex problems in a wide range of domains beyond games. It also highlighted the importance of combining different techniques, such as deep learning and tree search, to achieve the best results.

by Patricia Reichhelm

Even though it was considered impossible for a long time, researcher were finally able to build an AI model that was able to beat the best human player in the game of *Go*. How they did this is described in the paper. The game has an incredibly big search space of possible moves and states and on top of that, strategies are not straight forward, as a currently bad position can turn out to be beneficial later on in the game.
Inspired by Image Classification tasks, *Convolutional Neural Networks* were successfully applied to "image" inputs of the board and this was combined with Monte Carlo Tree Search in order to train the model on finding the best moves. To be more concrete, a combination of two different kinds of networks were used: *Value networks*, that were implemented to assess the current game state, and *Policy Network* in order to determine the best action to take. The policy nTeworks was trained on a huge Go-database of different boad positions and then Reinforcement Learning was applied by letting the current policy network play the game against older (more outdated) versions of itself, using a reward function and updating the weights of the network respectively.
The Value network uses an evaluation function to determine the value of a given state with a certain policy. Both networks are further combined using MCTS, based on the value, visit count and prior probability of each state.
In the end, a tournament of the AlphaGo program against other Go-playing programs showed that AlphaGo wins about 99% of all games - even when providing advantages to the opposing programs. The most significant proof for AlphaGo's strength is defeating the European Championship in Go in all five games that were played.

2. Implementation

The game is played on a 3x3 board, with 'X' and 'O' being the two players. The user is always 'O', and the computer is 'X'. The board is represented as a list of lists, where each inner list contains the symbols representing the players' moves. Initially, the board is empty and each cell is represented by a period (".").
The code defines several functions that are used to simulate the game and calculate the computer's next move. The key functions are:

- *hasMovesLeft*: checks if there are any empty cells left on the board.
- *hasWon*: checks if the specified player has won the game.
- *getBestNextMove*: uses Monte Carlo Tree Search to determine the best move to do next
- *getNextMoves*: generates a list of all possible next moves for the current player.

The *getBestNextMove* function works as follows: it simulates a number of possible games (controlled by the *numberOfSimulations* variable), starting from the current board state. For each simulation, it randomly chooses moves for both players until the

game ends (either because one player wins or because there are no more moves left). It then evaluates the final state of the game, assigning a score to it based on whether the computer player won or lost, and how quickly the game ended. The function then returns the move that leads to the highest average score across all simulations.
Finally, the code runs a loop that alternates between the user and the computer player. In each iteration, it displays the current state of the board and prompts the user (if it's their turn) to enter a valid move. If it's the computer's turn, it calls the *getBestNextMove* function to determine the next move. The loop continues until either the game ends in a win for one player, or the board is filled and the game ends in a tie.
Monte Carlo Tree Search (MCTS) is a popular algorithm for solving games and decision-making problems, and this implementation of MCTS is relatively simple and accessible for beginners. The algorithm learns and improves over time by iteratively expanding its search tree and updating its evaluations based on the results of simulations. MCTS has the ability to handle large search spaces and complex game scenarios, making it useful for games with many possible moves or complex strategies. Additionally, the simulation-based approach allows for flexibility in modeling game scenarios, allowing the algorithm to handle a wide range of game rules and complexities. Moreover, the specific algorithm operates really fast, probably faster than any human being. It also operates on a 4x4 grid or larger and we both tried to win games over it without success. However, MCTS can be computationally expensive, especially for large or complex games. This can limit its applicability in real-time systems or resource-constrained environments. Additionally, the quality of the algorithm's results can depend heavily on the quality of its evaluations, which in turn depends on the quality of the simulation model used. If the simulation model is inaccurate or incomplete, the algorithm may not perform well.
Finally, MCTS may suffer from issues such as over-exploration or over-exploitation, which can result in suboptimal or inefficient behavior. These limitations should be carefully considered when deciding whether to use MCTS for a particular game or problem.

Part II: Lecture 6 - Game playing Systems, 21.02.2023

Summary by Patricia Reichhelm

First of all, it was made clear that a lot of game implementations don't actually require machine learning algorithms to have computers learn the rules.
Zero-sum games are an example for that. For such games, for one player winning, another loses. *Branching paths* show sequences of all possible actions. In order to win, the best path should be chosen. This choice is not a trivial task, but the implementation of *search trees* can help here. The value of the states refers to the winning possibility after choosing this state. Choosing the path of leaf nodes (end states) that show a *win* and backtracking this to the next actions is thus the goal strategy. The fact that you can only make plans since you can't control your opponent is the challenge in this approach. If the opponent had a fixed policy, one could use the probability of each winning states to determine the best actions that are most likely to succeed. If that policy changes,

the winning probabilities change as well. So in so-called *Minimax Games* the goal is to find a strategy that hast the least probability to lose with an opponent that has the highest probability to make us lose. Limitations here are the enormous search space and that strategies to win quickly might turn out negatively long-term.
When looking at games as *decision processes*, this describes an *agent* taking *actions* with respect to its *policy* that results in *transitions* from the current *state* to another, oftentimes also making use of a *reward function*. Whereas it is possible to enumerate all possible states in gmaes like *Tic-Tac-Toe*, it is not always possible to do that for more complex games like *Chess*. In order to speed things up, one can limit the number of steps ahead that are processed or only try a number of actions instead of all of them. *Monte Carlo tree search* (MCTS) uses a tree policy to select paths, expands them by simulating based on the policy and then uses backpropagation to update the value of paths based on the value of the terminal node that is reached. The collected statistics can also be used to improve the selections and the simulations. *Upper confidence bounds* are an approach that trade off exploration and exploitation and can be applied to search trees. After repeating MCTS several times, actions are chosen based on UCT. Nowadays, functions are used to approximate Q-values of nodes inside of machine learning algrotihms. This yields the possibility of andling larger state spaces.

Summary by Stavros Chatzistavros

At the beginning, it was established that many games do not require machine learning algorithms to teach computers the rules, particularly zero-sum games where one player's win is the other's loss. In these games, the best path is chosen from a sequence of all possible actions represented by branching paths. The implementation of search trees can be helpful in this regard by assigning a value to each state that represents its probability of winning. The goal is to choose the path that leads to a leaf node indicating a win and backtrack to the previous actions. However, the challenge is that since the opponent's actions cannot be controlled, only plans can be made. In games with a fixed opponent policy, one can use the probability of winning states to choose the best actions. However, when the policy changes, so do the winning probabilities, making it challenging to find the best strategy in minimax games.

Games can be viewed as decision processes in which an agent takes actions based on its policy, resulting in transitions from the current state to another, often using a reward function. While it is feasible to enumerate all possible states in games like Tic-Tac-Toe, it is not possible in more complex games like Chess. One can speed up the process by limiting the number of steps or actions considered. Monte Carlo tree search (MCTS) can be used to select paths by using a tree policy, simulating expansions based on the policy, and updating the value of paths using backpropagation. Upper confidence bounds can be applied to search trees to trade off exploration and exploitation. Nowadays, machine learning algorithms use functions to approximate Q-values of nodes, making it possible to handle larger state spaces.

Part III: Reflection on Diagnostic Systems

by Patricia Reichhelm

Regarding the general feedback to the assignments, I was glad to realize that we fulfilled most of the stated bulletpoints. One thing to improve could definitely be the use of visualizations in our texts, it was good that this was once again brought back to our attention.
Discussing whether black-box AI surgeons are good to use was an interesting extension of this module's topic. It is a highly critical and important topic. While it feels weird to use a system that people don't really know how it works and how it will behave in new situation, even though the high cure rate and time-efficiency is a great advantage in such life-critical areas. I personally still feel like, especially in life-critical situations, applied AI's should be interpretable and this module strengthened this opinion. A high cure rate is only useful if you can rely on it. Predicting and testing behavior of new situations is especially hard in such systems. This is even more important looking at it from another point of view: it was made clear again that it is hard determining whether the increased score after using AI instead of humans for surgeries is actually a causality that comes from the replacement (and not by different testing circumstances). These discussion is something I'll take with me long-term wise.

by Stavros Chatzistavros

I'm generally pleased with our performance on the assignments in this module, but I've identified an area where we could improve: using visual aids in my written work. During the module, we discussed the topic of black-box AI surgeons, which I found to be a fascinating addition of the subject. I recognize the significant importance of this topic, and while I understand the benefits of high cure rates and time saving in life or death situations, I'm uneasy about using a system with unpredictable behavior. In my opinion, applied AI should be interpretable, especially in life-critical situations where we need to rely on it.
The module reinforced my belief that a high cure rate is only valuable if it's dependable. It's challenging to predict and test the behavior of AI systems, especially those that are black-box, so it's crucial to have interpretable systems in such cases. Additionally, I found it difficult to determine whether the increased score obtained after using AI instead of humans for surgeries is a causal effect of the replacement or due to different testing circumstances. All in all, it was certainly an interesting module.