Implementing a chess AI

by group 40: Felix Japtok 977397 & Jan Kettler 979374

Cognitive Science, Universität Osnabrück

Implementing ANNs with TensorFlow

Prof. Dr. Michael Franke, Leon Schmid, Nion Schürmeyer, Charlotte Lange, Annemarie Witschas

April 4, 2021

Task description

For our project, we implemented a chess AI that trains by playing against itself, similar to what AlphaZero, DeepMind's chess AI, does. We wanted to see how well a similar approach in this regard would work with a very simple network and much fewer resources for training.

To create our chess AI, we split the project into two parts: the chess game itself and a neural network that outputs value estimates for an input board state. In the game, we included all the pieces and their moves, all the rules such as castling and promotion, classes for the board and a board manager that work together to contain all the information we need about the game state. Additionally we implemented an agent for the behavior during the game and a test class to test different parts of the system separately. To decide which move to execute in a given state, the agent performs a Monte Carlo tree search guided by the returned values from the network for all expanded states.

Due to the high complexity of the game chess, with over 100 million possible board states after only a few moves, we also had to incorporate a reasonable limit of moves through the search tree to make the program executable for our technical resources. Runtime is a big concern for us, as sampling a single game with our limited resources takes an average of roughly 3 minutes. Another problem is evaluating our result, because although it is possible to observe our AI's moves and keep track of the loss, it is very difficult to evaluate those moves in a larger perspective. Our initial plan to test our AI by playing against other AIs such as Stockfish was not possible. This is because we have to write each board state into an input array for our network after each move and choose an action based on the network’s output, which takes far too long for a single game.

Progression

In the following we will sum up in short how we approached the project and what problems we encountered while designing our system.

First we started to implement the game of chess by creating the class for the board followed by the classes for the pieces and a more comprehensible way of printing the boardstate. Next we implemented the boardmanager and all the moves for the pieces. Here we encountered the first major issues, especially concerning the validity of moves. These problems occurred because for each move we have to determine if, after performing the move, the own king would be in check, making the move illegal. At this point we realised that the position of both kings is specifically important, for that reason we decided to save it in a variable directly. Eventually we worked around our issues by deep-copying the chessboard, which works, but probably at the expense of efficiency. To finish the implementation of the game we added the termination conditions (checkmate, stale-mate, move-repetition, ...).

Simultaneously we implemented a test class to test the individual parts of our system and our agent to control the behavior during the game. To let our agent pick a move we created an expand method that simulates taking the possible moves. We decided to expand in the form of a tree, to perform Monte Carlo tree search. Our tree class is inspired by resources of the AlphaZero approach we found online.[1] Generally implementing the tree was complicated, especially finding a good exploration-exploitation-tradeoff when expanding. To optimize the network, we needed to create a target value to obtain the loss for the gradients. At first we wanted to use some heuristics, like piece-value, in addition to the game-outcome as the target. But we decided to discard this approach because our results were inconsistent and not promising. Instead we decided to use only the outcome of the game as a target for the network.
We tested three different simple neural network-structures, one with dense-layers, one with convolutional-layers and one with both respectively.

Summary of related approaches

Since there are already different chess AIs we decided to list some of the best chess AIs today, Stockfish and AlphaZero, one old chess computer, deep blue, and compare them to our approach in some general aspects:

[1] https://github.com/AppliedDataSciencePartners/DeepReinforcementLearning/blob/b04e80409a26896ae0e5f1d4cbca603f9ae4eff2/MCTS.py

	Stockfish	AlphaZero	Deepblue	Our approach
Developed by	Tord Romstad, Joona Kiiski and Marco Costalba under the license GNU General Public License	DeepMind	IBM	Our group
Improvement by	Automatic tuning + testing new version vs. old versions	Reinforcement learning	testing new version vs. old versions	Reinforcement learning
Programming language	C++	Multiple languages	C	python
What is evaluated	Board states	Board states	Board states	Board states
Evaluation function	Set of heuristic rules, alpha-beta pruning lookahead algorithm	Deep neural network	fast evaluation function: computes score for board state in single clock style, contains all fast computed features with high values slow evaluation function: scans board state column by column with values for chess concepts (square control, pins, forks...)	Neural network
evaluating states	output of evaluation is centipawn (cp.) value to evaluate board states	predicts the outcome of the match from any given state and a probability -distribution for all possible moves	massively parallel tree search through evaluated positions to choose the best one	predicts outcome of the game from any given state

Fig 1, Sabatelli et al., 2018; Campbell et al., 2002; Grünke, 2020; Silver et al., 2017

Explanation of model and training choices

Our idea of training an agent to play chess is that it learns by playing itself, so we used a model-based approach with reinforcement learning to train our system. The network uses multiple dense and/or convolutional layers. Our goal was to test how well a simple network performs in chess, learning only by self-play. So we kept our network structure basic. Generally it is important not to augment the input to the model in any way, like rotating for example, because the rules of chess are not symmetric and minimal differences in position can be highly influential. Thus the only regularization method we use is batchnormalization.

We decided to use a Monte Carlo tree search to choose the next move. This algorithm is suitable because our approach is model-based and chess is deterministic. The exploration during Monte Carlo tree search in our approach is driven by a penalty term, where visiting the same node multiple times reduces the probability of visiting that node again. The edges of the tree contain the estimated value of the best reachable state following the path of the edge. When the agent chooses a concrete move it chooses the move that maximizes the estimated value.

We use a replay-buffer[2], containing sampled trajectories, to allow for batch learning and to learn multiple times from the same game. The input board states for the model are one-hot encoded to represent the categorical pieces. The output values of our network should be between -1 for loss, 0 for draw and 1 for a win. For this reason we use leaky ReLu as the activation function between layers and tanh as the activation function of our output layer.

Analysis of results and ablation studies

______As mentioned before, the evaluation of our result is a big problem, since there is no simple evaluation measure for chess, like accuracy for classification tasks. Therefore, our evaluation will be based on runtime, loss, and a subjective evaluation of the behavior during the game.

In terms of time consumption, training the model takes very little time, less than a second per training step, due to its simple structure. The most time-consuming part of the system is playing a game before storing it in our buffer. This is directly related to the parameter "num_steps" of the agent, since "num_steps" limits the times we expand our search tree. The relationship appears to be almost exponential, increasing our required time steps for a single game from ~50s for num_steps = 10 to almost double that for num_steps = 15. This part is also affected by the choice of layers, with more time steps required when convolutional layers are used instead of dense layers. Overall the size of the neural network is not a large contributor to the time consumption. We found that the neural network consisting of both convolutional and dense layers performed best.

The loss is not a great indicator of performance, because the samples are obtained by self play, resulting in high variance. But in general we observed a loss around one for all three networks, which slightly decreased over time.

[2] code from: https://github.com/geronimocharlie/ReAllY/tree/master/really/really

Since we tried two different ideas by first having our network predict the value of a board state, which we changed to predicting the outcome of the game from the given state, we will split the evaluation of the behavior into two parts.

For the first idea, exploration was clearly visible during training, while the model is not yet trained. Even after training, the system tends to move the same figure over and over again. Sometimes both agents simply swap a piece between two positions multiple times, which quickly leads to a draw by repetition. Moves like castling and on-passant were hardly used even after training. Overall, the training success was visible, but inconsistent and very slow.

With the second idea, the behavior improved significantly. While the moves were initially very random, after some training the moves seemed to make more sense from the perspective of a human chess player. For example, if one side made a mistake at the beginning of a game, possibly due to exploration, the other side actually recognized the opportunity and achieved a scholar's mate. We are of the opinion that the agent’s play has continuously improved. Generally we observed an increase in “thinking ahead” of the agent, less often making nonsensical moves like hanging pieces or not taking free opponent pieces. The agent learned to rather take pieces of higher value like Bishops or Rooks instead of Pawns when given the choice. Also promotions to Queens got more likely than to lower value pieces like Knights for example. Nevertheless, castling and on-passant were rarely used.

In conclusion we do not think that our first approach had no potential, but rather other problems in the implementation might have led to unpromising results.
In comparison the second approach definitely showed potential. The agent was able to learn basic concepts and heuristics in chess, although final performance was still not nearly on the level of other well known chess AIs. We are unsure about how well our agent could perform, given more training, because as mentioned earlier training was quite slow for us.
An increase in performance is very likely when the neural network would, in addition to a value estimate of the state, also predict the possible moves just like AlphaZero. This is because the likelihood of moves can be used as an additional heuristic in the Monte Carlo tree search. Furthermore the neural network would be trained to only predict valid moves, which indicates that it learns the rules of chess.

References

Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). Deep blue. *Artificial intelligence*, *134*(1-2), 57-83.

Grünke, P. (2019). Chess, Artificial Intelligence, and Epistemic Opacity. *Információs Társadalom*, *19*, 1-7.

Sabatelli, M., Bidoia, F., Codreanu, V., & Wiering, M. (2018). Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead. In *ICPRAM* (pp. 276-283).

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.