Frankfurt University of Applied Sciences
Fachbereich 2: Informatik und Ingenieurwissenschaften
Informatik (B.Sc.)

Generating Chess Commentary using Machine Learning

Abstract

This scientific seminar paper deals with the question how machine learning can be used to generate comments on chess games. On the one hand it is shown how a chess engine can be built to analyze games and on the other hand how the information provided by the engine can be used by a virtual chess commentator to generate comments on them. The engine is built by a neural network and a Monte-Carlo Tree Search algorithm and can be trained by self-play. The virtual chess commentator uses an encoder-decoder model which uses bidirectional LSTMs architecture.

Lecturer: Konstantin Ernst
Course: Künstliche Intelligenz und wissenschaftliches Arbeiten
Winter Semester 22/23

Submitted by:
Max Semdner
Matrikelnr.: 1294899
max.semdner@stud.fra-uas.de

Declaration of authorship

I hereby certify that the following project report was written entirely by me and is based on my work unless otherwise indicated. I am aware of the University's regulations regarding plagiarism, including the following actions in the event of a violation. Any form of use of outside work is identified where appropriate and noted in the sources.

Max Semdner

Approved: _____

Date: _____

Contents

1 Introduction

In the mid-20th century, computer chess experienced its first breakthroughs thanks to the work of scientists like Alan Turing, Claude Shannon and John von Neumann. Alan Turing, the pioneer of artificial intelligence, was convinced that games were an ideal model system for machine learning.[1] This prediction has come true, and machine learning have proven to be an essential part of many chess engines today. In particular, recent projects such as AlphaZero, developed by DeepMind, show how efficient programs which use neural networks are in analyzing board games compared to traditionally used algorithms such as alpha-beta search.[2] Although chess engines have become a powerful tool, they have a lack of transparency regarding the moves they perform. Therefore, professional chess players and commentators are often needed to explain the intention of these moves. This dependence on human chess commentators can be a disadvantage, since moves found by computers can sometimes be misinterpreted or not understood at all. Especially for non-professional chess players, many moves played by humans as well as computers are incomprehensible, since they do not have the appropriate experience. In the following, the construction of a chess engine is discussed, which corresponds in its play and analysis strength to the engines of today. This should serve to analyze moves in their depth. Then the construction of a virtual chess commentator is described, which uses the information gained by the analysis of the chess engine to create detailed comments on a given position and move.

2 Generating Chess Commentary

2.1 Chess Engine

2.1.1 Requirements

Any chess engine must meet a number of requirements in order to function. These requirements include the *representation of the chessboard*, the *search of the possible moves* and the *evaluation of the position*.[3] These requirements can be implemented in a number of ways. Certain implementation options have become widely accepted. In the following, one implementation procedure is presented in detail for each requirement. These are state of the art implementations that have been in use for many years or have achieved great success in the recent past.

2.1.2 Board Representation

Since the computer cannot work with a physical chess board and pieces, these must be converted into a form in which the board, pieces, and position can be interpreted by the computer and later used for the input layer of the neural network. The most common form of representation is the data structures bitboards. A bitboard is implemented as an 8×8 bit array, which is the size of a chessboard. Each array element corresponds to a square on the board. A bitboard is created for each type of piece (pawn, knight, bishop, rook, queen and king) of a given color (black and white). This gives a total number of 12 bitboards. Finally, the squares on which the figures are placed must be marked on the

[1] Cf. Levy and Newborn (1982), pp. 44-45
[2] See Silver et al. (2018), p. 1
[3] Cf. Klein (2022), pp. 75-76

respective boards. This can be done in binary, where 0 means the square is empty and 1 means the square is not empty.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	1	1	0	0	0	0
1	1	0	0	0	1	1	1
0	0	0	0	0	0	0	0

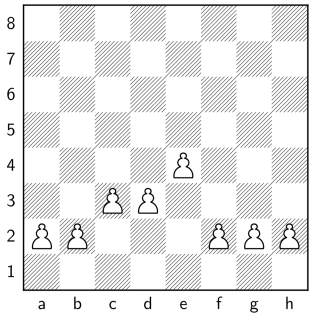

Figure 1: Representation of bitboard with white pawns (left) and the corresponding chessboard (right)

Through the usage of the logical operations, such as AND, OR, NOT the moves can be calculated. An advantage of the logical operations is, that be done quite fast by the processor. Furthermore, with x64 processors the position can be stored in one piece in the memory, as a bit string, since this is exactly 64 bits long due to the number of squares. Another advantage is that due to its simple representation, it can be used as input for the input layer in the neural network without the need to convert it into an understandable format.

2.1.3 Move Search and Position Evaluation

A challenge for both humans and computers is to find the best possible move. In fact, chess is considered unsolved, i.e. it is not known if there is an optimal strategy that always leads to victory, for either sides. The objective of a good chess engine is therefore to find the best move based on its computational capabilities. One factor to consider is the depth of analysis. Thus, each move must be considered not only in terms of the current state of the board, but also what effect it will have on subsequent best moves and positions. In order to find the best move, two tasks must be achieved, one is to find the legal, possible moves in the current and the following positions, and the other is the evaluation of these positions.

To implement these two tasks, many systems such as DeepBlue and earlier versions of Stockfish use human-defined evaluation functions and game tree search algorithms. The evaluation function receives the board as input and evaluates how good the obtained position is. Game tree search algorithms, such as MiniMax, are then used to search for all possible moves (Usually, up to a certain depth, e.g. 15 moves). Each move leads to a new position. Since there is no information about these new positions yet, they need to be evaluated by the evaluation function. In the end, the path that has received the highest value from the evaluation function is chosen. Improvements such as alpha-beta pruning remove paths that are known not to yield high values of the evaluation. The problem with this method is that the evaluation functions have to be written by hand with the help of chess experts and constantly refined to achieve an optimal result.

To overcome this problem, and to get the best results from the engine for the commentator, machine learning is used. Silver et al. (2018) presented an algorithm that uses Monte Carlo tree search (MCTS) and a neural network. The neural network is used to evaluate an input position. It outputs two pieces of information: A vector with the move probabilities of all next legal moves and, a value, which tells what the player's chances of winning are for the given position. MTCS is used to search deeper, i.e., to search many

possible move paths and then select the path with the best evaluation. To select the next move MCTS selects "each valid move, play a number of random games"[4], generates statistics for these randomly selected paths and compares them.[5] The best evaluated move of a path is played. The statistical evaluation for each path can be generated by the evaluations made by the neural network at each node of the search tree.

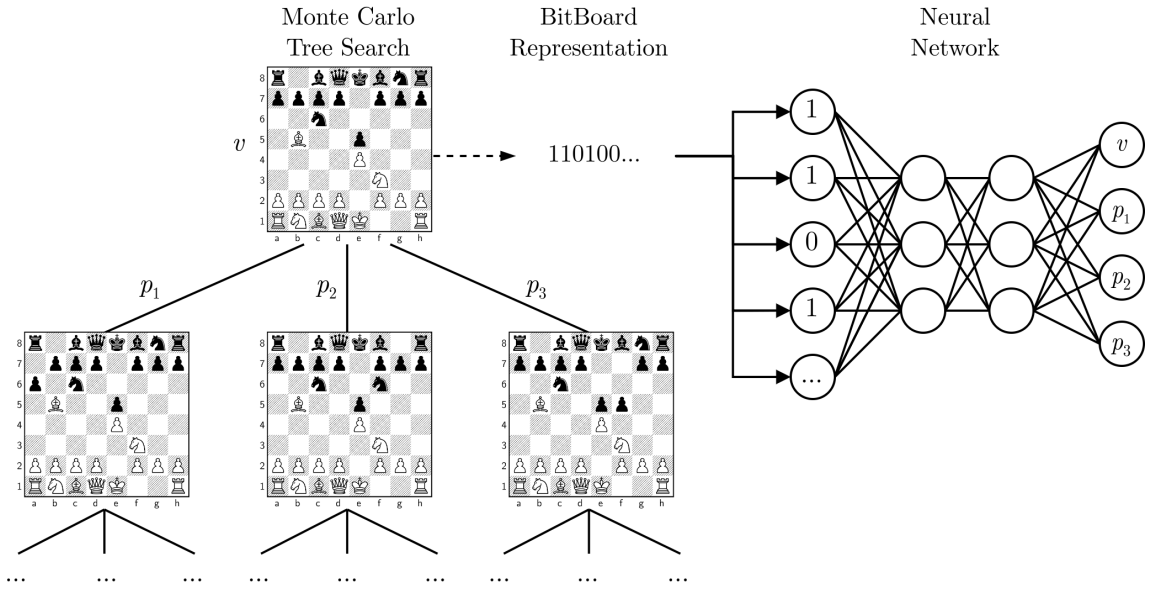

Figure 2: Simplified example of move search and position evaluation using MCTS and a neural network. (p_n move probabilities, v winning chance value)

In order for the neural network to deliver the best possible results, it must be trained with data. Instead of relying on existing data sets, Silver et al. (2018) used the approach of generating their own data sets through self-play. Initially a program exists that only knows the rules of the game, uses an untrained neural network for evaluation and the MCTS algorithm for move selection. The program executes the following four steps to train the neural network: (1) The program plays against itself by searching for several moves in advance through the MTCS, letting the neural network evaluate these moves for move probability and position strength, and finally choosing the path with the best evaluation. Every game is recorded move by move. At the end of each match, the final position of the sides are assigned lost (-1), drawn (0) or won (+1) to create a dataset.[6] (2) The neural network is cloned and the parameters are adjusted "to minimize the error between the predicted result [...] and the game result [...]" and train the network with the generated data set. (3) Let the new program play against the previous one. (4) The winning program is selected and the process is repeated from step 1. As Silver et al. (2018) showed, this method was able to outperform Stockfish, the strongest engine at this time, after only 4 hours.

[4]Klein (2022), pp. 106-107
[5]See Klein (2022), pp. 100
[6]See Silver et. al p. 2

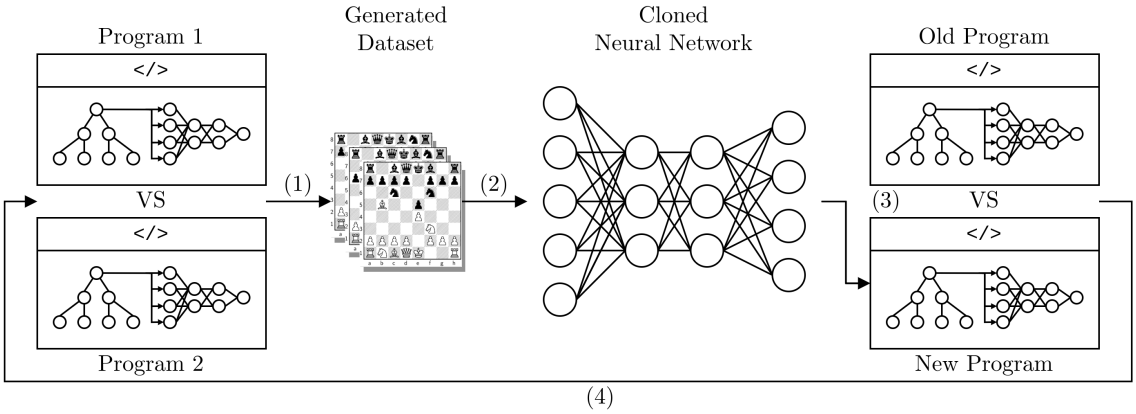

Figure 3: Steps of traning the neural network through self-play

2.2 Chess Commentator

2.2.1 General Approach

The chess commentator has the task to convert certain information, which it receives from the chess engine described before, into human understandable comments. Such a task falls into the domain of *sequence-to-sequence* processing. The idea of sequence-to-sequence models are that an input of a certain length is mapped to an output of a certain length, where input and output length can be different. For such tasks, encoder-decoder architectures with attention has achieved great success in the past. The architecture consists of two parts, an encoder and a decoder, which are two *bidirectional recurrent neural networks (BRNN) using long short-term memory (LSTM) state cells.*

Recurrent neural networks (RNN) are a special type of neural network designed to process sequential inputs and recognize patterns in them. Due to an internal memory, RNNs are able to remember and reuse certain characteristics of the input. In general, an RNN works as follows: (1) recive an input (2) let the neurons generate an output, (3) copy that output (4) take the copied output and the new input and return to (1). Thus, the new RNN input consists of the output generated in the past and the new sequence part added, allowing the network to build a deep understanding of the sequence. An extension of an recurrent neural networks are bidirectional recurrent neural networks. While the neurons of RNNs always pass the generated output to the neuron in the immediate future, in BRNNs there is another level of neurons that can pass the output to past neurons. By including information from both the past and the future, an even deeper understanding of the sequence can be built. When Sutskever et al. (2014) first introducted their sequence to sequence model they used RNN with long short-term memory state cells. Those are a special neuronal architecture of an RNN, which is why it is also simply referred to as LSTM or BLSTM (in case of bidirectional LSTMs). LSTM use a special kind of neurons, called state cells, which solve problems of RNNs and BRNNs that make it difficult to train them.

The encoder-decoder architecture used for generating chess comments makes use of the described BLSTMs. The idea of the architecture is that the encoder receives a sequence of data step by step, processes the data and encodes it into a fixed length vector. This vector are the last states of the encoder, used by the decoder to set its initial states to it to produce the desired translation. One problem that affects the quality of the decoder's translation, is the length of the sequences. The sequence stored in the vector tends to dilute over time, resulting in poor translations. To solve it, an attention mechanisms

can be used. When processing the data, only the most important information received is considered, so the vector, called the context vector, contains only a summary of the sequence. This allows the decoder to focus only on the information relevant for translation. If the decoder now wants to generate comments, it receives a start symbol as input (represented by `<START>` in Figure 4). This start symbol is used to indicate the start of the output sequence. The first output is the first word of the translation. The output is additionally used as new input in the next step, generates an output and which is used as the new input again. This procedure is repeated until an end symbol is reached, which signals the end of the sequence. The generated output sequence is the translation.

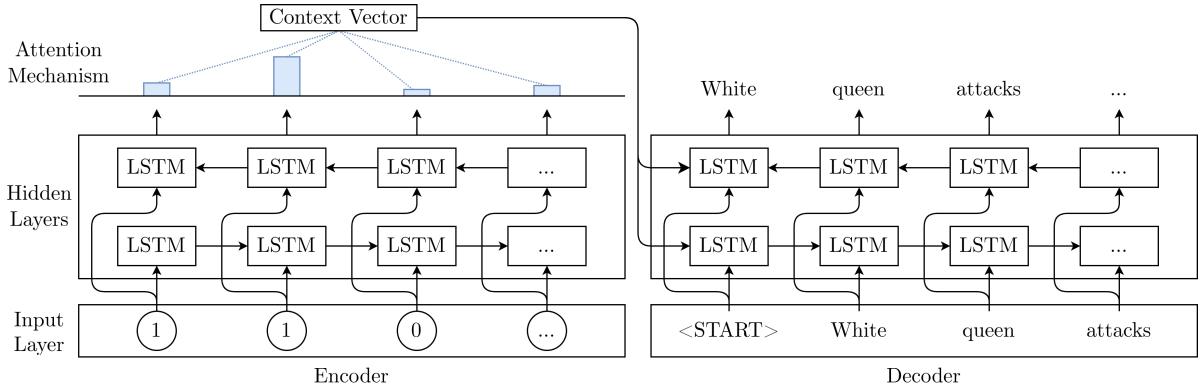

Figure 4: Chess Commentator Encoder-Decoder Architecture with Attention Mechanism

2.2.2 Generation Models

For a given chess position and a played move, the commentator should create comments on different categories. Those categories are the *description of the current move*, the *description of the move quality*, the *comparison of moves*, the *description of the move planning* and *contextual game information*.[7,8]

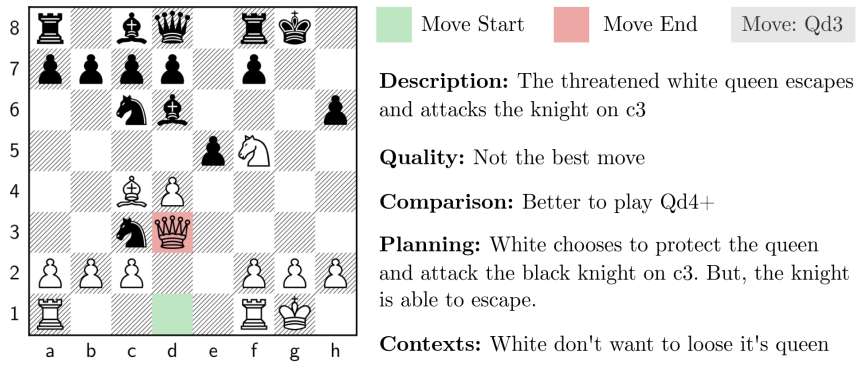

Figure 5: Chess Commentary Example (Note: Figure based on Zang et al. (2019) Figure 1)

Since the procedure for creating comments varies from category to category, it is practically impossible to do this with a single commentator, as simplified described above. Therefore, in the following it will be spoken of five individual ones, each for one of the categories. In addition, instead of categories, it is now spoken of generation models, since these are used to generate comments. These generation models come with two challenges

[7]Cf. Jhamtani et al. (2018), p. 3
[8]Cf. Zang et al. (2019), pp. 3-4

that can have a strong impact on the quality of the generated description: (1) the features on the basis of which the comments for the aspects are generated, (2) the number of moves and the resulting position considered. To overcome the first challenge Jhamtani et al. (2018) used "discrete information (threats, game evaluation scores, etc.)"[9]. How Zang et al. (2019) show, those features information did not provide the best results. Therefore, they have used features provided directly by the internal chess engine. These are the *state of the board before the move*, *start square of the move*, *end square of the move*, *piece on the start square*, *piece on the end square*, *promotion state* and *checking state*. The pieces on the starting square and on the ending square can be different, since the pawns can be replaced by another piece when they reach the opponent's last rank.[10] In this case the promotion state would be set to 1. The advantage of using these features is that they can be easily read from the information provided by the chess engine described in the previous section. The second challenge can be overcome by distinguishing between generation models that need only one move (description and quality) and generation models that need multiple moves (comparision, planning and context) so that the description is accurate. Depending on the model, they implement either a single-move encoder or a multi-move encoder. These two encoders differ in their training so that both can precisely store different amounts of information in the context vector.

[9] Zang et al. (2019), p. 4
[10] See fid (2018) p. 6

Glossary

DeepBlue First chess engine to win against a world chess champion. 4

Stockfish Most use free and open source Sachach engine. 4

References

(2018). *FIDE LAWS of CHESS*. International Chess Federation.

Czech, J., P. Korus, and K. Kersting (2021, May). Improving alphazero using monte-carlo graph search. In *Proceedings of the International Conference on Automated Planning and Scheduling, 31*, pp. 103–111. arXiv.

Hochreiter, S. and J. Schmidhuber (1997, 11). Long Short-Term Memory. *Neural Computation 9*(8), 1735–1780.

Jhamtani, H., V. Gangal, E. Hovy, G. Neubig, and T. Berg-Kirkpatrick (2018, July). Learning to generate move-by-move commentary for chess games from large-scale social forum data. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Melbourne, Australia, pp. 1661–1671. Association for Computational Linguistics.

Klein, D. (2022). Neural networks for chess.

Levy, D. and M. Newborn (1982). *All About Chess and Computers*. Springer Berlin, Heidelberg.

Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Volume 362, pp. 1140–1144.

Sutskever, I., O. Vinyals, and Q. V. Le (2014). Sequence to sequence learning with neural networks. In *Proc. NIPS*, Montreal, CA.

Zang, H., Z. Yu, and X. Wan (2019, July). Automated chess commentator powered by neural chess engine. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy, pp. 5952–5961. Association for Computational Linguistics.