IANNwTF - Winter term 2022/23

GPT-2 model learns the rules of chess

Clive Tinashe Marimo (cmarimo@uni-osnabrueck.de)
Moritz Lönker (mloenker@uni-osnabrueck.de)
Silvie Opolka (sopolka@uni-osnabrueck.de)

supervised by Mathis Pink (mpink@uni-osnabrueck.de)

April 1st 2023

Contents

1 Abstract

In this project we investigate the capabilities of a transformer based large language model in the context of learning the rules of chess from recorded games. We show how it is possible to fine-tune a foundation model to learn how to play chess using various techniques. We employ random sampling techniques during model generation and show that performance of the trained model increases and the model performs less legal moves. Since usual language metrics like predictive accuracy and perplexity do not work in this context we introduce new evaluation techniques which are relevant to the context of teaching a transformer model to play chess.

2 Context and motivation

Chess is a complex game that requires a deep understanding of strategy and tactics. According to DeLeo et al. (2022) [1], chess has around 10^{43} unique chess board states, as well as a branching factor of approximately 35, i.e. in most game positions, there are about 35 possible moves that can be made. Additionally, Shannon estimated that there exist 10^{120} unique chess games.[1] This makes chess a computationally challenging problem, even for today's advanced models and capacities.
In the past years, multiple strong chess engines (e.g. AlphaZero and Stockfish) got published that are capable of beating the most advanced human chess players.[2]
However, there is rather little research on using natural language processing (NLP) to train a language model to play chess from chess transcripts exclusively. The objective of this project is to train a Generative Pretrained Transformer (GPT) model to learn how to play chess from chess transcripts (i.e. a series of moves) of several human plays, without having an explicit representation of the game board state and game rules. We want to investigate, if it is possible for such a model to 'learn' and apply the complex rules and theory of chess, much like it is possible for established transformer models to learn and apply complex grammatical rules.
This project has the potential to lead to new insights and understanding of how language models can learn and understand complex concepts. Additionally, it will demonstrate the diverse and powerful nature of transformer models once again.

3 Related literature/research

In 2019, DeepMind introduced the algorithm MuZero, which is able to master games, including chess, without knowing their rules.[3] In the following years, more research projects attempted to train a language model to play chess without encoding the rules of chess explicitly.

DeLeo et al. (2022) [1] applied a BERT model to the game of chess, using a game state encoding in Forsyth-Edwards Notation (FEN), and a move encoding in coordinate algebraic notation. After training the model for three days on a dataset covering only chess game openings (i.e. containing only the first three moves per game), the model was able to generate 75 percent valid moves at 35 moves (per player) into the game. This observation shows that a language model is able to generate chess games, even with very little information, given it is fed with a large enough amount of data.

Stöckl et al. (2021)[2] analyzed the effect of GPT-2 model size (small, medium, large), training time (0 - over 120 train hours), and amount of training data (99604, 577202, 2163417 games) on the model's learning success. For evaluation, the models were saved at multiple training time steps.
In this study, the following important observations were made:
First, the usual evaluation metrics for language models (accuracy, perplexity) seem to give no reliable indication of the model's performance success. Results show that both, accuracy and perplexity measures, do not vary (significantly) between different model sizes, suggesting that smaller models can learn the rules of chess almost similarly well/quickly as larger models. However, chess-specific metrics show a clear difference between the model sizes. Additionally, both, accuracy and perplexity measures, indicate a strongly decreased learning rate when increasing the amount of training data. However, chess-specific metrics show a significant increase in learning progress when using larger amounts of training data. Based on those findings, we will use chess-specific metrics for our model evaluation.

[1] https://github.com/ricsonc/transformers-play-chess
[2] https://www.chess.com/terms/chess-engine
[3] https://www.deepmind.com/blog/muzero-mastering-go-chess-shogi-and-atari-without-rules

Second, the raw pre-trained GPT-2 model was able to generate look-alike chess games. However, those hardly contained legal moves, independent of the model size. This bad performance is explained by the insufficient and inconsistent chess training data: Insufficient because the model was trained more generally, not specifically on chess data. Inconsistent, because the training data might have contained multiple chess notation variants. Those findings highlight the importance of fine-tuning the model on chess data and the need of adjusting its inner architecture where necessary.
Third, they could observe that the performance of the model improves, the more time is invested in training. Their graphical visualizations show a strong but increasingly limiting improvement. The strongest improvement appeared in the first 20 hours of training. This gives us an insight into the training time needed to observe sufficient results.
Fourth, the medium-sized model with medium data volume seems to be the best-performing option, providing a good trade-off between performance capabilities and computational resources. After 20 train hours, this model generates an average sequence of about 28 legal moves (before generating the first illegal move) from a known opening position, an average sequence of 11 legal moves from a game position after ten moves, and an average sequence of 2.8 legal moves from a game position after ten random moves. Confirming the results of Kaplan et al. (2020), their observations suggest that model size and data volume should be increased together for good results.

Noever et al. (2020) [3] fine-tuned a GPT-2 architecture to learn the rules of chess, using a rather large data volume with 2.8 million chess games in Portable Game Notation (PGN). Their evaluation focused on special PGN tokens (castling, pawn promotion, check, checkmate) and common opening moves (like "English Opening" and "Slav Exchange"). After 30,000 training steps, the model was able to generate chess games with 10 percent illegal moves. Common moves were king-side castling (17.74 percent in 1000 generated games) and pawn promotion (13.67 percent in 1000 generated games).

Oshingbesan et al. (2022) [4] conducted research in the field of multi-domain, multi-task learning. Their research goal was to train a small GPT-2 model on code and chess data, such that it performs well in four different tasks (Chess move generation, Chessboard state evaluation, Code generation from an English prompt, and Code summarization using the English language) of the two unrelated domains "Python code" and "chess". When comparing several popular training strategies, the model resulting from GPT-style joint pre-training and joint fine-tuning was able to keep multi-domain knowledge the best, while still performing reasonably well in the individual tasks. Their baseline GPT-2 model is smaller compared to the model used by Noever et al. (2020) [3]. However, the model performance (9.6 percent illegal moves) does not differ significantly, suggesting that even smaller models can perform equally well when using more training steps (50,000 instead of 30,000) and training data (14.3 million games instead of 2.8 million games). Another crucial observation is that all tested models struggled to end the chess game.

Presser et al. (2020) [5] made a similar observation, where the model started to blunder around move 13. Like Oshingbesan et al. (2022) [4], they suspect this is due to losing track of the board state. We will check if this problem appears in our model as well.

4 Common chess notations

During our literature review, three different notations (PGN with SAN, PGN with LAN, and FEN) appeared frequently. **Portable Game Notation (PGN)**[4] [5] is a file format for chess games that contains metadata (e.g. names of players and their ELO ratings, date and time of the game, result) and the game transcription (i.e. the sequence of moves). Typically the transcript is given in **Standard Algebraic Notation (SAN)**. In professional games all players must create such a SAN transcript during the game and proficient chess players are able to read SAN almost like natural language. **Long Algebraic Notation (LAN)** can also be used for game transcription, though it is less natural because it doesn't explicitly denote the piece being moved or whether a piece is captured. LAN represents moves using coordinates of the start and end position of the piece being moved (e.g. c2c4). **Forsyth-Edwards Notation (FEN)** describes a particular board state by representing the position of each piece on the board, including empty fields. FEN representations can be generated from SAN/LAN, but not the other way around. A FEN of a specific board position is sometimes given in PGN files.

[4] https://chesspedia.fandom.com/wiki/Chess_Notation
[5] https://support.chess.com/article/658-what-are-pgn-fen

5 The data

For training and evaluation, we use data coming from chess transcripts. The chess transcripts are downloaded from the internet chess server Lichess, which provides an open-source database containing all games played on the server since 2013.[6] Currently (March 10, 2023), there are 1.3 TB available with information on over 4 billion (standard) chess games. The chess games played on Lichess might not be representative of all types of games and players, but they provide enough information about the rules of chess. Therefore, collecting data from multiple sources, to cover as much diversity as possible, is not necessary.
The transcripts come in .pgn.zst format. PeaZip[7] was used to decompress the data to get the PGN files.
With more data available than we could ever use, there was no reason to be conservative in choosing games. We only used games that were ended by checkmate and that were done by move 100. We made no selection according to player skill, because we also wanted to explore suboptimal opening strategies that might not be used by higher ELO players. We used pgn-extract[8] to create a second dataset in LAN format to train the LAN model.

6 The model

Huggingface provides many pre-trained GPT-2 models.[9] We chose to fine-tune the GPT2LMHeadModel, which is a GPT-2 model with a language modeling head. GPT-2 is a type of transformer model first introduced by Radford et al. (2019)[6] that has been pre-trained on a large corpus of English data through a self-supervised learning process. Huggingface provides multiple GPT-2 model sizes, for easier training and because of our constrained available memory we chose the smallest one with 124M Parameters. The Transformer architecture introduced in the paper by Vaswani et al. (2017) [7], is the first sequence transduction model based entirely on attention, replacing the recurrent layers most commonly used in encoder-decoder architectures with multi-headed self-attention. The Transformer follows an auto-regressive path consuming the previously generated symbols as additional input when generating the next architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1.2, respectively.

According to the Vaswani et al. the attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key [7]. Scaled Dot Product Attention is a computation of dot products of the query with all keys and then applying a softmax function to obtain the weights of the values. In Multi Head Attention the queries, keys and values are linearly projected h times with different learned linear projections resulting in several attention layers running in parallel as seen in Figure 1.3.

GPT-2 is trained with a simple objective: predict the next word, given all of the previous words within some text. The diversity of the dataset causes this simple goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data[6]. This means that it has been trained on raw text without any human labeling, allowing it to use publicly available data. The training process involved predicting the next word in a sentence by using an automatic process to generate inputs and labels from the text.

The inputs for the model are sequences of text with a specific length, and the targets are the same sequence, but shifted one token to the right. The model uses a mask-mechanism to ensure that it only uses input from 1 to i, excluding future tokens when predicting token i.

By doing this, the model learns an inner representation of language, which can then be used to extract useful features for downstream tasks. However, the model's primary strength lies in generating texts from a given prompt, which is what it was pre-trained for.

7 Model types

We originally planned to only train a SAN model, but quickly encountered the problem of illegal moves. The training also proved very inefficient as every move was encoded by the GPT-2 byte-pair tokenizer, and thus a move like "Bxd4" is made up of four different tokens. We wanted to directly tokenize the moves, as this would

[6] https://database.lichess.org/
[7] https://peazip.github.io/
[8] https://www.cs.kent.ac.uk/people/staff/djb/pgn-extract/
[9] https://huggingface.co/docs/transformers/model_doc/gpt2/transformers.GPT2LMHeadModel

significantly speed up training time and would allow us to eliminate illegal moves. We therefore analyzed the number of possible moves in SAN notation and observed a logarithmic increase with the number of games, which made direct tokenization impossible. Figure 1.1 shows the results of the analysis. We instead chose to create a second model, trained on games in LAN notation, which only has 1968 possible move notations. The model vocabulary was extended and the added weights were randomly initialized. The direct tokenization of move notations allowed us to force the model to generate only legal moves. Each move the model receives a list of all possible legal moves and then applies constrained beam search/ grid beam search[8] to choose a move. The LAN model is able to play a game against the player from start to finish.

8 The methods

8.1 Byte-Pair Tokenization

Tokenization is the process of representing raw text in smaller units called tokens. These tokens can then be mapped with numbers to further feed to a language model. In this case chess sequences are mapped to a numerical representation which can be used to fine tune the GPT-2 model. Byte pair encoding is used to encode the chess sequences during training as inspired by [2]. It is used to find the best representation of text using the smallest number of tokens. Byte-Pair Encoding (BPE) was initially developed as an algorithm to compress texts, and then used by OpenAI for tokenization when pretraining the GPT model. A typical chess game of 50 moves from both black and white requires about 200 tokens for encoding the whole game. However, a game can be much longer. We use a maximum sequence length of 256 and cut of the rest of the moves. The algorithm can be summarised as follows:

Step 1 Add text identifiers and calculate chess move frequency.

Step 2 Split the chess moves into characters and then calculate the character frequency.

Step 3 Merge the most frequently occurring consecutive byte pairings.

Step 4 Iterate n times to find the best (in terms of frequency) pairs to encode and then concatenate them to find the submoves.

8.2 Top-k Sampling

Top-quality generations from the transformer model rely on randomness in the decoding model and that requires sampling predictions from the most probable choices. One method we use in this study is top k sampling highlighted by Fan et al., 2018 [9] which samples the next chess move from the top k most probable choices instead of aiming to decode text that maximizes the likelihood. Formally, given a probability distribution $P(x|x_{1:i-1})$ we define the top-k vocabulary as $V^{(k)} \subset V$ as the set of size k which maximizes $\sum_{x \in V^{(k)}} P(x|x_{1:i-1})$. If $p^{'} = \sum_{x \in V^{(k)}} P(x|x_{1:i-1})$ then the distribution is re-scaled according to 8.2 and sampling is performed on that distribution. The scaling factor $p^{'}$ can vary dramatically at each step.

$$\mathrm{P}^{'}(x|x_{1:i-1}) = \begin{cases} P(x|x_{1:i-1})/p^{'}, & \text{if } x \in V^{(p)} \\ 0, & \text{otherwise} \end{cases} \quad (1.1)$$

8.3 Top-p Sampling

If k is small, in some contexts there is a risk of generating bland or generic text, while if k is large the top-k vocabulary will include inappropriate candidates which will have their probability of being sampled increased by the renormalization. Hence, we introduce top p sampling. Top-p sampling is used to check the ability of the model to produce not only the most likely move but also other valid moves since more than one possible chess move might be available from the chess position. This random mechanism is used to counter the problem of language models which always generate the same sequence if the word with the highest probability is chosen next.

The sampling allows the model to generate different chess moves sequences and reduce the tendency of repetition. Holtzman et al. (2020) [10] showed how pure sampling or sampling directly from the probabilities predicted by the model results in text that is incoherent and almost unrelated to the context due to the so called unreliable tail. The authors then introduce the concept of nucleus sampling where the intuition lies in the vast majority of probability mass at each time step is concentrated in the nucleus, a small subset of the vocabulary that tends to range between one and a thousand candidates. The authors proposed sampling from the top-p portion of the probability mass, expanding and contracting the candidate pool dynamically as part of the nucleus sampling concept. Given a distribution $P(x|x_{1:i-1})$ the top p vocabulary $V^{(p)} \in V$ as the smallest set such that

$$\sum_{x \in V^{(p)}} P(x|x_{1:i-1}) \geqslant p$$

8.4 Sampling with Temperature

Temperature sampling is used in this study for sample-based generation by shaping the probability distribution through temperature. It can complement the top-k sampling method by shaping the distribution before top-k sampling as shown by the authors in [6] and [9]. With the logits $u_{1:|V|}$ and temperature t, the softmax is reestimated as:

$$p(x = V_{l|x_{i:i-1}}) = \frac{exp(u_l/t}{\sum_{l'} exp(u_l'/t)}$$

9 The implementation

9.1 Data Preprocessing

The selected Lichess data contains millions of games in PGN using SAN. Games in LAN are generated from the same data using *pgn-extract*[10]. During preprocessing, the games are filtered and extracted. The function *read_games()* of the file *read_games.py* gets a specific number of games from the file, which are expressed in either SAN or LAN. The metadata, empty lines, and evaluations (only within the SAN game expression) are removed to obtain a single-line string in SAN or LAN per game. Only games that ended with a checkmate were considered to ensure a definite outcome, and those that finished due to a player's timeout (e.g. in Blitz chess) or resignation were not included. Therefore, the model is expected to generate games ending with checkmate only. Generated games that end under other conditions are considered incorrect. To reduce computational power and avoid memory issues, the game length was limited to a maximum of 99 moves, i.e. games with 100+ moves were cut at move 100 but still integrated in the dataset. The resulting data is expressed as an array (if *'write_to_file = False'*) of all considered games (single-liners containing the moves in SAN or LAN only), each game ending with the result tag (*'1-0\n'* or *'0-1\n'*). This format was used for later evaluation. If *'write_to_file = True'* the extracted games were stored in a file instead. The dataset (used for training the model) was created using the file and *create_dataset* function (of the *create_dataset.py* file), using the *GPT2Tokenizer* of Huggingface.

9.2 Model class

We created a class for our model, which can be found in the *model.py* file. Our model is based on the pre-trained *GPT2LMHeadModel* of Huggingface and uses the *GPT2Tokenizer*. An instance of this class can use the following methods:

- *train()* trains the model using a trainer and training dataset
- *load()* loads a saved model from its directory
- *generate()* generates a predefined number of tokens (i.e. moves or move parts) from a predefined start position in LAN or PGN/SAN (i.e. the prompt); can generate multiple predictions for similar start position
- *generate_legal_move()* is only available for the LAN model and is used as a fallback for the generate() function. The function gets a list of all possible moves in the current board state, the model then performs a constrained beam search to find a suitable move. This means the LAN model can always choose a legal move, but it typically will choose bad moves as soon as it is out of its training distribution.

[10] https://www.cs.kent.ac.uk/people/staff/djb/pgn-extract/

- *add_vocab()* adds missing tokens to the model's vocabulary, which is useful for training the model on specialized domains. This function must always be used for the LAN model together with the *"lan_vocab.txt"* file. It will then add all possible moves in LAN notation, which are around 2000.

9.3 Training

There are two common ways to train such a model that appear frequently in literature:[11]

1. **Sequence modeling approach**: The input is a whole sequence of moves (starting from the initial board state), from which the current game state information can be derived. The model predicts the next move of the sequence, given all previous moves. In summary, the chess move taken is represented explicitly and the game state is represented implicitly.

2. **"Markovian" approach**: The input is a single representation of the current board state (e.g. using FEN) and the model predicts the next board state, based on the current board state (i.e. the last input) solely. In summary, the chess move taken is represented implicitly and the game state is represented explicitly.

We chose to train the model using the sequence modeling approach - a similar approach used in language modeling, where the next word (or word sequence) is predicted based on the previous words. Natural language is a sequence of characters and words. The SAN can be seen as a natural language sentence, where the individual moves correspond to words. Making a chess move is equivalent to adding a new word to the string. In Language modeling, the next word (or word sequence) is predicted based on the previous words. First, an instance of the *GPT2LMHeadModel* is created, which uses the *GPT2Tokenizer* of Huggingface's pre-trained GPT-2 foundation model. The *train()* function gets called on this model instance. Both models were trained on 100,000 games with a train/test split ratio of 0.1. The SAN model was trained with a block size of 128, with a batch size of 24 and on 7 epochs. The LAN model was trained with a block size of 96, with batch size of 32 and on 10 epochs. Both models were therefore trained for about 12 hours, we trained the SAN model on a longer batch size because the games in SAN notation consist of more tokens, we successively had to lower the batch size because of memory constraints.

9.4 Evaluation

The evaluation of the model is inspired by the work of Stöckl et al. (2021) [2].
During training, the model was saved every 2000 steps. Six models with different training times (1k - the model after warm-up, 10k, 20k, 30k, 40k, and 54k - the final model) were evaluated on five metrics. Each metric provides a different type of start position from which the model must generate a valid game. The following table summarizes the metrics.

Metric	Start position
eval1	Typical opening position (two moves into a known human game of train dataset)
eval2	Five moves into a human game of test dataset
eval3	Ten moves into a human game of test dataset
eval4	Five moves into an artificial game
eval5	Ten moves into an artificial game

Table 1.1: Overview over metrics used in the evaluation.

The metric *eval1* is used to test the model's ability to remember training data, rather than checking if it "learned" the rules of chess. *Eval2* and *eval3* are used to test if the model has "learned" the rules of human chess. *Eval4* and *eval5* are used to test if the model has "learned" the rules well enough to apply them in random, more abstract, and possibly human-unlike gameplay. This sets the largest challenge for the model and should clarify if the model actually applies rules "learned" during training, or if it relies on pure memorization of the sample games. Even for good human chess players, it seems to be very difficult to handle random positions.[12] We decided to generate games on different input sequence lengths (two moves, five moves, ten moves) to check if the problem of losing

[11] Following project mentions both approaches: https://github.com/ricsonc/transformers-play-chess
[12] http://www.chrest.info/Fribourg_Cours_Expertise/Articles-www/II%20Donnees%20empiriques/Gobet%20&%20Simon-1998-Memory.pdf

track of the board state occurs in our model as well.
For each metric, five start positions were prepared. The functions used can be found in the *create_startpos.py* file.

- *generate_common_openings* finds the five most common openings of the train dataset, which are used as start positions for *eval1*.
- *generate_startpos* extracts the first five games of the test dataset. The first five (or ten) moves of each game are used as start positions for *eval2* and *eval3*.
- *generate_random_startpos* generates five random legal sequences (of five or ten moves) from the initial board state, which are used as start positions for *eval4* and *eval5*. The *python-chess* library is used to get all legal moves of the current board state.

Each model was tested on similar start positions to maximize comparability. The functions used can be found in the *eval_funcs.py*. The model generated a game from every start position of each metric, i.e. a total of five games were generated for each metric. For the game generation, we used top-p sampling (p=0.95) to reduce repetition and increase randomness among the possible legal moves generated. The games were generated bit by bit (30 tokens each time) to save computational power. It was ensured that each move of the generated sequence contained both players' moves and that both moves were captured fully. Each move in the generated sequence was checked - first if it is a result, and then for correctness. A correct result, wrong result, or an incorrect move does end the current test loop. While a correct result signifies that a completely legal game was generated, an erroneous game is denoted by the finding of a wrong result or an incorrect move. In case the move is correct, the next move will be checked. If there is no move left in the generated sequence, then the start position will be updated by the generated sequence. Then, the next sequence part will be generated by the model and checked after. The *python-chess* library was used to check for move correctness and to keep track of the board state.
Each metric test will return the incorrect move in SAN (None type if fully correct game generated), the number of correct moves generated until the first incorrect move, and the type of the incorrect move (None type if fully correct game generated). The following table summarizes the error types.

Error type	Meaning
Wrong result	The model detected game end correctly but generated wrong result.
No game over	The model detected that game is finished but it is not yet.
Invalid move	The model generated a move that is not legal in the current board state (e.g. a pawn mowing forward two squares when it is not on its starting position).
Illegal move	The model generated a move that is not legal according to the rules of chess (e.g. making a move that puts own king in check).
Ambiguous move	The model generated a move that is not clear which piece is intended to move (e.g. target square is reachable by both pawns).
Wrong notation	The model generated a move in wrong SAN (i.e. notation mistake).
Unknown error	An undefined error occurred.

Table 1.2: Overview over error types.

The evaluation results are visualized using the following functions that can be found in the file *create_eval_visual.py*.

- *create_graph* calculates the average amount of generated correct moves until the first incorrect move (of the five games generated) for each evaluation of each model. This mean value is used as a data point for the *matplotlib* graph that visualizes the model's training progress.
- *create_table* uses tabulate to create a table that contains all data obtained during evaluation, i.e. the model, metric, amount of generated correct moves until first incorrect move (all games), incorrect move in SAN (all games), and type of incorrect move (all games).

Figure 1.5 visualizes the evaluation procedure.

9.5 The gameplay interface

We have developed a gameplay interface (*demo.ipynb*), where human players can challenge the trained model in live play. We added the following customization:

- The human player can use either SAN ('san' or 'pgn') or LAN ('lan' or 'uci') but is strongly advised to stick to one throughout the game. The type of chess notation can be defined when initializing the game with the string parameter *model_format*. The default is LAN.
- The human player can either play as White (i.e. starts the game) or Black. This can be defined when initializing the game with the bool parameter *human_start*. The default is set to True, i.e. the human player starts the game.
- The start position of the game can be chosen, which allows resuming old games or trying multiple strategies for certain game scenarios. The start position can be defined when initializing the game with the parameter *start_position*. The default is set to the usual start position.

We made use of the *python-chess* library for displaying the board, as well as checking and conducting the moves. The game class can be found in *game.py*. It contains the following three methods:

1. *play()* is the main loop of gameplay. Until the game ends, it displays the current board state as a chessboard and in the defined notation, requests and receives the move from the player on the turn, checks its correctness, and plays the move if it is valid.
2. *get_move()* retrieves the move from either the human player or the AI model depending on whose turn it is. If it is the human's turn, it prompts the user to enter their move in SAN or LAN notation. In the model's turn, the model generates the top five possible moves. One of the correctly generated moves is chosen as the move the model makes. If all generated moves are incorrect, the model will resign (if SAN/PGN format) or force a legal move using constrained beam search (if LAN/UCI format).
3. *get_board_position()* returns the current board state in either SAN, LAN or FEN.

An example output of the interface can be seen in Figure 1.6.

10 The results

10.1 Loss development of SAN model and LAN model during training

Figure 1.4 shows the loss development for the SAN and LAN model during training. The SAN model started with a loss of about 1.35 and reached a loss of about 0.9 after training for seven epochs. The LAN model started with a loss of about 2.9 and reached a loss of about 1.6 after training for ten epochs.

10.2 Evaluation of the SAN model

An exhaustive evaluation was only conducted for the SAN model because the LAN model was programmed to "force" legal moves to add flexibility in the type of notation being processed, as well as to enable complete games in the human-ai gameplay interface.

Figure 1.7 shows the training progress of the SAN model based on the average generated amount of correct moves until the first illegal move for all five evaluations. All metrics, except *eval5*, show clear variation in performance across the training. Stronger variation is found in *eval1* and *eval4*. Overall, it is clearly observable that the performance decreased with the increasing size of the input sequence (i.e. start position), confirming the problem of loosing track of the board state that has been found by several other papers as well.[4] [5]
The games generated from the opening position (*eval1*) contain by far the highest amount of correct moves, ranging between 19.6 and 25.2 correct moves on average. The model after 1,000 train steps appears to be the best performing, and the model after 54,000 train steps is the worst performing. One can observe alternating patterns of increase and decrease with a downward trend. This good performance was expected since first, the model knew all start positions (i.e. they were taken from the training data), and second, the start position was comparably short with only two moves into the game. Based on the good performance measured, we can surely say that the model has a very good ability to recall the training data.
The games generated from a position that is five moves into the game (*eval2*) range between 5.0 and 10.8 correct moves on average, where the model after 1,000 train steps appears to be the best performing and the models after 30,000 and 40,000 train steps appear to be the worst. One can observe a continuous downward trend in the graph.
The games generated from a position that is ten moves into the game (*eval3*) range between 3.0 and 5.6 correct

moves on average, where the model after 40,000 train steps appears to be the best performing and the model after 54,000 steps the worst performing. One can observe a trend upwards with a stronger downfall in the performance of the last model evaluated.
The games generated from a position that is five random moves into the game (*eval4*) range between 4.2 and 9.0 correct moves on average, where the models after 1,000 and 40,000 train steps appear to be the best performing, and the model after 30,000 steps the worst. One can observe very strong fluctuations.
The games generated from a position that is ten random moves into the game (*eval5*) show a very stable amount of about 1.8 correct moves on average. Overall, the games generated from random start positions show a slightly lower correctness, compared to games generated from start positions generated by humans.
The comparable weak performance of the model in *eval2*, *eval3*, *eval4*, and *eval5* might suggest that the model only grasped the rules of chess shallowly, and rather relied on its memorization and replication abilities.
Table 1.3 compares our results to the results of Stöckl et al. (2021)[2].

Start position	Our model (average over all tested models)	Stöckl et al.
Common opening (train data)	22 correct moves	28 correct moves
After 10 moves (test data)	4.1 correct moves	11 correct moves
After 10 random moves	1.8 correct moves	2.8 correct moves

Table 1.3: Comparing our evaluation results with Stöckl et al.

Figure 1.8 gives an overview of all games generated by the SAN model during evaluation. The overall decreasing performance visualized by the graph might not represent the real model's progress during training. When looking at the non-averaged data, one can observe a strong variation among the five games generated for an evaluation (except *eval5*). For example, all models in *eval2* generated a game with zero valid moves but also a game with 11/15/23/24 valid moves. Therefore, five games might not be enough for correct representation.
The table also shows that there are multiple examples in *eval2*, *eval4*, and *eval5*, where the model directly generated an incorrect move. This might suggest that there wasn't enough training data for the model to make sense of the rules of chess.
With regard to the incorrect moves generated, one can observe that capture moves have the highest error quote, being mostly illegal (except two times ambiguous for *'Nxf6'*). This suggests that there wasn't enough training data on capture moves specifically. In fact, capture and check are the only special moves appearing in the list of incorrect moves. Castling, pawn promotion, and checkmate do not appear. The lack of pawn promotion and checkmate might not mean that those moves are understood properly by the model. Rather, it can be explained by the fact that the training data focused on the chess game beginnings, such that those rather at the end appearing moves were not represented enough to be considered as possible moves.
Besides normal moves, which are all illegal moves, there are multiple result tags (marked as invalid moves) that have the correct notation but a comma wrongly added to it. This can be explained by the fact that the training data contained an additional comma to separate the games from each other. In future work, we might want to make use of a <endoftext> token to avoid such errors.
The error types 'Wrong result', 'No game over', 'Wrong notation', and 'Unknown error' do not appear in the table. The lack of result-related errors can be explained by the fact that most games weren't far enough to be ended (except the games in *eval1*). Also, the result tags generated were in the wrong notation, such that we could not check if those results generated were used correctly by the model. The lack of the 'Wrong notation' error could be the result of the Byte Pair Encoding used because it allowed the model to handle out-of-vocabulary words and reduce the vocabulary size efficiently.
Finally, there is a lot of repetition. In *eval2* the incorrect moves *'Qxd8+'*, *'cxd5'*, *'Nxd5'*, and *'Bxc4'* are generated in almost every model for the same start board state. A similar observation can be made in the other evaluations. This suggests that either the model memorized specific sequences intensely such that it performs similar tactics, or the model had no other, as 'correct' identified move to consider. The last point could be checked by reducing the p-value used in the top-p sampling to allow more randomness.

11 Discussion

Overall the project was a success, when playing against the base GPT2 model and our finetuned models, a clear difference can be seen. The models are able to play correct opening moves and play reactively to the player. The

models also often use suboptimal moves, but this was to be expected, as we trained mostly on amateur games. Interestingly the SAN model seems to have a passive bias and avoids capturing pieces, unless absolutely necessary, at least after the first few opening moves. This is probably due to an under-representation of capture moves (e.g. Nxe5) in the data. At around move 10-12 the SAN models starts playing illegal moves, which might be due to a limited blocksize, limited training data and the overall complexity of chess positions and the SAN notation. The LAN model is typically able to play good reactive moves until move 20, when it starts playing moves without a clear objective. This is probably also due to training limitations like a limited blocksize. With enough memory and computational resources our methods could probably be used to train a model which performs on the level of an average player, over the duration of a whole game.

Follow-up questions

1. Beginners and advanced players differ in the moves that they use. Do the average "goodness" and range of "goodness" of training games have an effect on the model's success in generating legal moves? The "goodness" of a chess game can be inferred from the move evaluations (e.g. by Stockfish[13]) and ELO ratings of the players[14] that are calculated based on their past performance. Future models could be trained with a focus on high ELO games, which might lead to a better performance.

2. It is still questionable whether the transformer approach could perform well in the mid-end game when completely novel board positions are reached. As our models were only trained on the first few moves of the game, this was not the focus of our project.

3. Due to the high time consumption and computational cost that training and evaluation took, we weren't able to conduct a repetition study involving a greater number of generated games used for evaluation. Such a repetition study could be done in the future to check if the more realistic performance tracking by more extensive game generation differs from our results.

4. While a clear disparity in difficulty for input length was observed, we could only find a small difference in performance between the moves of the human game and the random moves generated due to the fluctuations in the data. Therefore, we suggest a follow-up study to look deeper into the performance difference between human-like input and random, unnatural input.

[13] https://stockfishchess.org/
[14] https://www.chess.com/terms/elo-rating-chess

Bibliography

[1] M. DeLeo and E. Guven, "Learning chess and nim with transformers," 2022.

[2] A. Stöckl, "Watching a language model learning chess," in *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2021)*, 2021, pp. 1369–1379.

[3] D. Noever, M. Ciolino, and J. Kalin, "The chess transformer: Mastering play using generative language models," *arXiv preprint arXiv:2008.04057*, 2020.

[4] A. Oshingbesan, C. Ekoh, G. Atakpa, and Y. Byaruagaba, "Extreme multi-domain, multi-task learning with unified text-to-text transfer transformers," *arXiv preprint arXiv:2209.10106*, 2022.

[5] S. Presser and G. Branwen, "A very unlikely chess game," Jul 2020. [Online]. Available: https://slatestarcodex.com/2020/01/06/a-very-unlikely-chess-game/

[6] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] C. Hokamp and Q. Liu, "Lexically constrained decoding for sequence generation using grid beam search," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 1535–1546. [Online]. Available: https://aclanthology.org/P17-1141

[9] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical neural story generation," *arXiv preprint arXiv:1805.04833*, 2018.

[10] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.

Figures

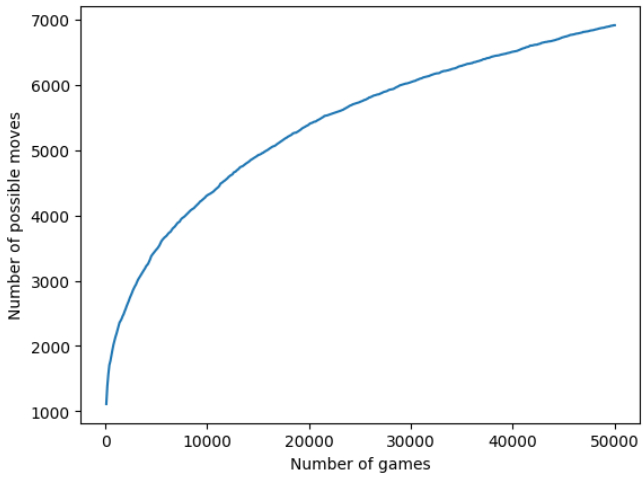

Figure 1.1: The number of possible moves

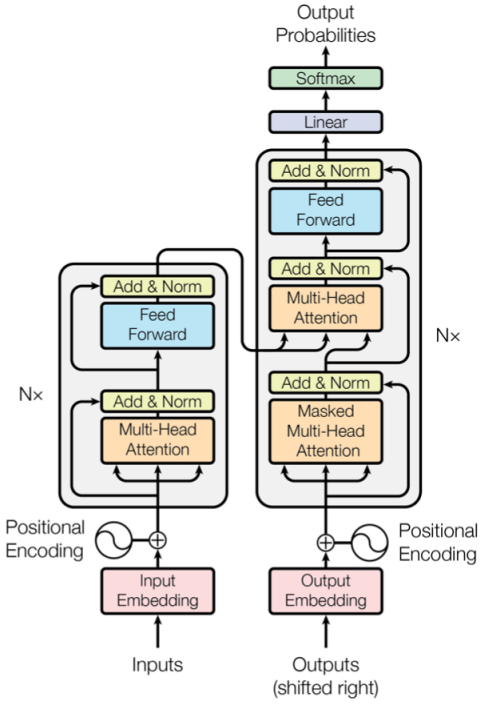

Figure 1.2: Transformer model architecture by Vaswani at al. (2017)[7]

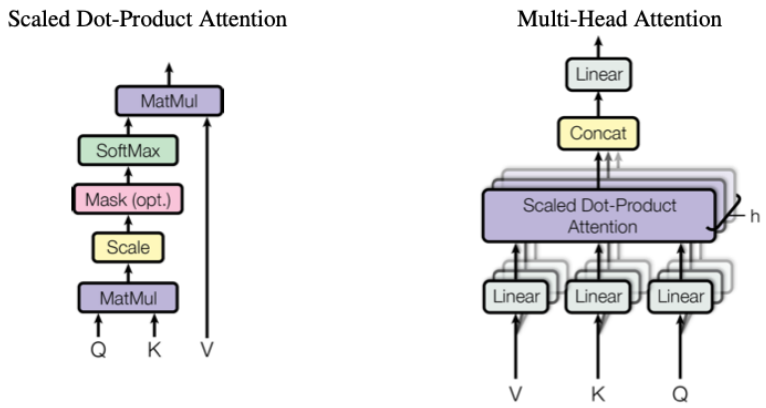

Figure 1.3: Scaled Dot Product Attention on the left and Multi Head Attention on the right [7]

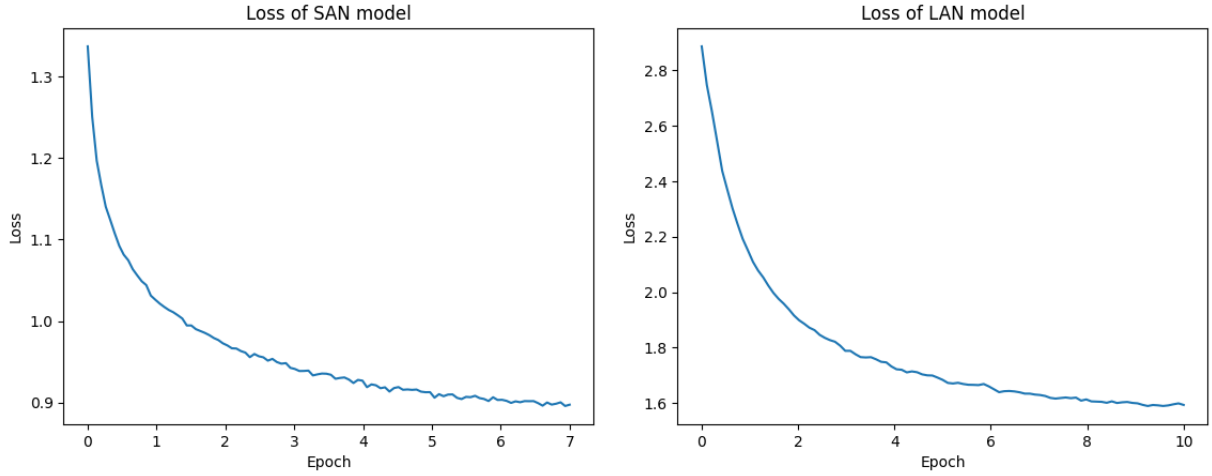

Figure 1.4: Comparison of loss development for the SAN and LAN model

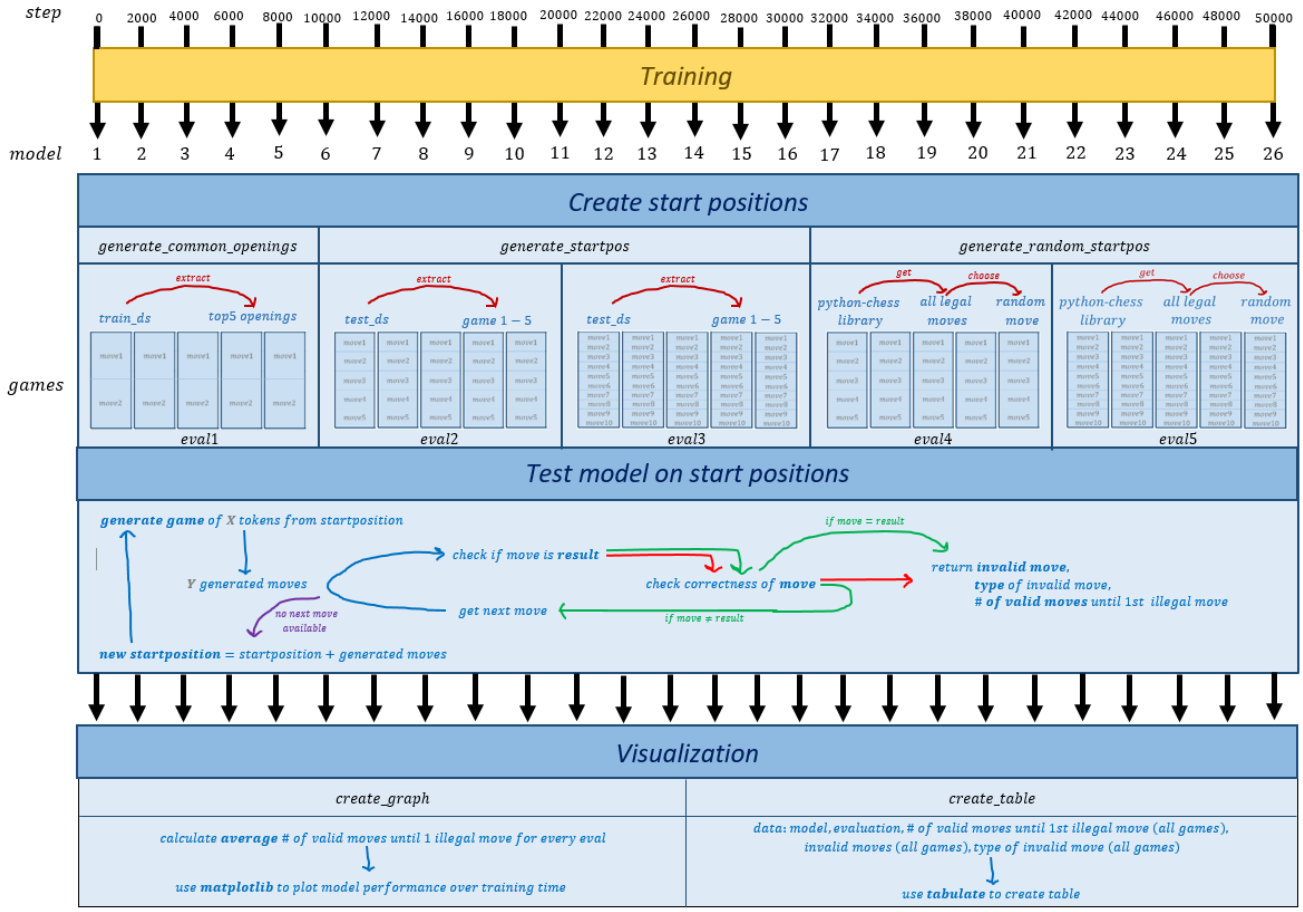

Figure 1.5: Evaluation procedure

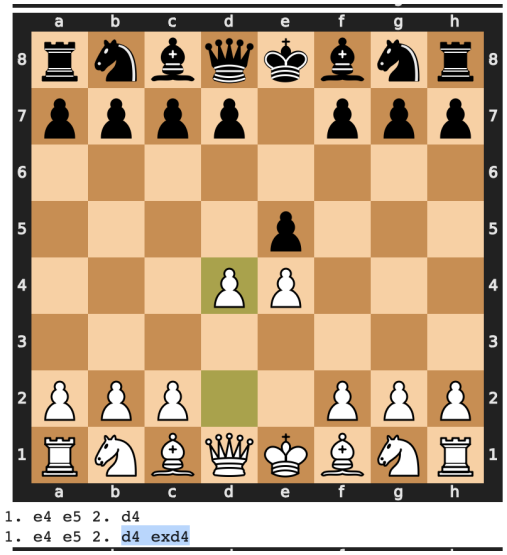

Figure 1.6: Example move (in SAN) generated - black takes white's pawn

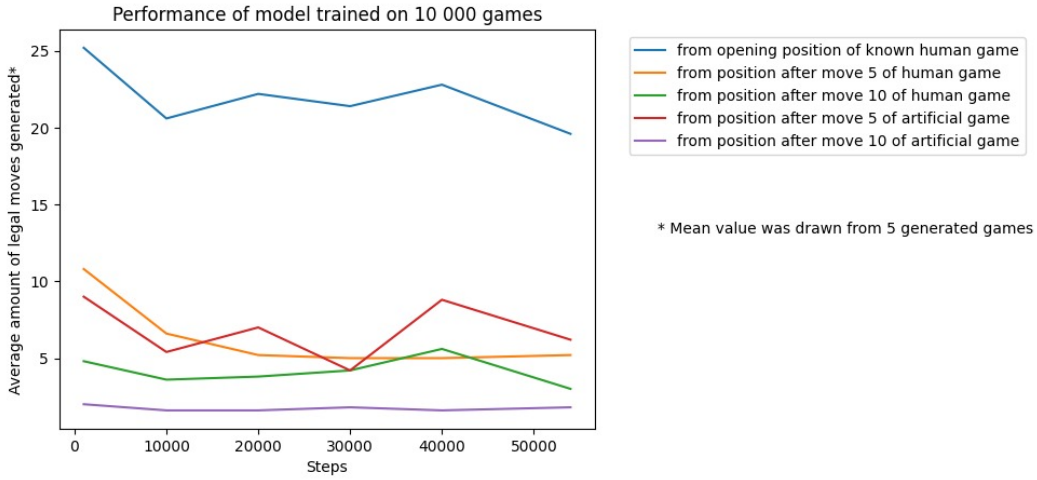

Figure 1.7: Training progress of SAN model

Evaluation	Model	Amount valid moves	Illegal move	Type of Illegal move
1	1	[13, 21, 29, 27, 36]	['Nxd5', 'Bg4', 'bxc3', 'Bxg6', 'Qe4']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
1	2	[11, 24, 24, 28, 16]	["1-0\\n",", 'Bxg6', 'Bxg6', 'g6', 'Bxg6']	['Invalid move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
1	3	[11, 28, 24, 32, 16]	["1-0\\n",", 'Bxd5', 'Bh6', 'gxf6', 'dxe4']	['Invalid move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
1	4	[11, 25, 29, 28, 14]	["1-0\\n",", 'Bxg5', 'bxc3', 'g6', 'Bg2']	['Invalid move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
1	5	[11, 26, 29, 31, 17]	["1-0\\n",", 'hxg6', 'bxc3', 'Bg7', 'Bg4']	['Invalid move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
1	6	[11, 25, 18, 28, 16]	["1-0\\n",", 'Bxg5', 'b4', 'gxh6', 'dxe4']	['Invalid move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
2	1	[4, 0, 22, 24, 4]	['Qxd8+', 'cxd5', 'Qxe5', 'Rae1', 'Bxc4']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
2	2	[4, 0, 2, 23, 4]	['Qxd8+', 'cxd5', 'Nxd5', 'Rxc2+', 'Bxc4']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
2	3	[4, 0, 2, 11, 9]	['Qxd8+', 'cxd5', 'Nxd5', 'Qh4', 'Bg4']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
2	4	[4, 0, 2, 15, 4]	['Qxd8+', 'cxd5', 'Nxd5', 'Be6', 'Bxc4']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
2	5	[4, 0, 2, 15, 4]	['Qxd8+', 'cxd5', 'Nxd5', 'Nxg3', 'Bxc4']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
2	6	[4, 0, 2, 15, 5]	['Qxd8+', 'cxd5', 'Nxd5', 'Nxg3', 'Bd6']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
3	1	[3, 2, 2, 6, 11]	['Bxf6', 'Qxa1', 'cxd5', 'Bxd5+', 'Bxe5']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
3	2	[3, 2, 2, 3, 8]	['Bxf6', 'Qxa1', 'cxd5', 'Qxd4', 'Bxe5']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
3	3	[3, 3, 2, 3, 8]	['Bxf6', 'Bxb3', 'cxd5', 'exd4', 'Bxe5']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
3	4	[3, 2, 2, 3, 11]	['Bxf6', 'Bxa1', 'cxd5', 'exd4', 'Nxf6']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Ambiguous move'
3	5	[3, 2, 8, 4, 11]	['Bxf6', 'Qxa1', 'axb5', 'Bxf5', 'Nxf6']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Ambiguous move'
3	6	[3, 1, 2, 1, 8]	['Bxf6', 'Nxb4', 'cxd5', 'Bg4', 'Bxc6']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
4	1	[11, 20, 0, 13, 1]	['c6', 'Nh4', 'Bb5+', 'Nxd4', 'Bxe7']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
4	2	[7, 15, 0, 4, 1]	['c6', 'Bxc5', 'Bb5+', 'Bxh6', 'Bxe7']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
...				
5	4	[0, 0, 4, 1, 4]	['c3', 'Qxc2', 'Nxe4', 'bxc6', 'Bb5+']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']
5	5	[0, 0, 3, 1, 4]	['c3', 'Qxc2', "1-0\\n",", 'bxc6', 'exd5']	['Illegal move', 'Illegal move', 'Invalid move', 'Illegal move', 'Illegal move']
5	6	[0, 0, 4, 1, 4]	['c3', 'Qxc2', 'cxd6', 'bxc6', 'Bb3']	['Illegal move', 'Illegal move', 'Illegal move', 'Illegal move', 'Illegal move']

Figure 1.8: Overview over all games generated by SAN model