TOWARDS RETRIEVAL-BASED CHESS ENGINES

Anirudh Ajith, Gudni Nathan Gunnarsson, David Xu

***Index Terms*—** Chess, Retrieval, Vector Search

1. INTRODUCTION

The game of chess has historically been a driver of progress in artificial intelligence due to its high branching factor and the infeasibility of a complete computational solution. Deep Blue's victory over Kasparov in 1996 was a landmark achievement and represented a phase change in the history of both the game, and of AI. However, most modern chess-engines are sophisticated algorithms that utilize complex value estimations, heuristics, tree-search algorithms and table-bases to find good moves.

The recent online chess boom has led to the availability of transcripts of billions of public games played between strong players. The moves played in these games are typically near-optimal as measured by the strongest modern chess engines. We propose a simple retrieval-based chess-engine that utilizes these transcripts to make strong expert-level moves in any given position.

We use a pretrained embedder [1] from prior work to embed chess positions into 64-dimensional vectors, and create a large vector database of 100M chess positions from the lichess dataset. We then train ChessBERT[1], a BERT-like model which learns a mapping from chess position to optimal move when given k retrieved similar chess positions in context. Each context example consists of a board position and its corresponding optimal move. We hope to show that providing the context positions significantly improves the rate of training as well as overall performance.

2. RELATED WORK

2.1. Chess Engines

In the early days of artificial intelligence, solving chess was considered by many as a significant milestone, if not the ultimate test, for achieving true artificial intelligence. Alan Turing was among the first to propose this idea [2]. Development of chess engines was emphasized by world governments and had lots of funding. But when Deep Blue bested Garry Kasparov in 1996 we learned that a good chess player is not necessarily intelligent in a general sense. Chess is considered hard because of its enormous branching factor - at any turn you can usually make about 30 different moves. So, it's absolutely impossible to search the entire game tree in any reasonable amount of time. However, most modern chess-engines are sophisticated algorithms that utilize complex value estimations, heuristics, tree-search algorithms and table-bases to find good moves.

2.2. Chess Embeddings

Machine learning for chess has generally focused around learning strong chess policies using reinforcement learning or evaluation functions for tree-based search. Thus, vector encodings for chess positions such as HalfKP [3] simply focus on providing an accurate representation of piece positions and additional information such as castling rights and current turn.

Chess2Vec [4] is one of the earliest works on generating latent representations for chess positions, and primarily focuses on methods involving matrix factorization. The authors demonstrate that non-negative matrix factorization (NNMF) of a count matrix can function as a weak chess engine. However, this representation is not suited for vector database lookup as NNMF does not impose any explicit structure on the latent space.

ChessPos [1] is a chess position embedding project designed to retrieve similar chess positions given an input position. ChessPos uses a triplet network [5] which takes inputs x, x^+, and x^- and minimizes the L2 distance between x and x^+ while maximizing distance between x and x^-. Given a position x, x^+ is defined as a position that occurs immediately after x in a real chess game between humans, while x^- is simply drawn randomly from training data. ChessPos also optimizes a reconstruction loss to ensure that the learned embeddings still contain information about the original position. Because ChessPos imposes a structure on the embedding space, we select the ChessPos encoder to create our vector database.

3. METHODOLOGY

3.1. Task and Dataset

Given a query chess position, we wish to train a model that returns a reasonable chess move. We additionally want to use retrieval to aide in selecting a good chess move. In order to do this we need a sufficiently large database of chess positions

[1]https://github.com/anirudhajith/ChessBERT

and moves. We used games from the lichess open database of games [6]. They catalog over 5 billion games, which is about 400 billion positions (of which about 320B are unique).

We filtered the dataset so that only games with players above 2000 ELO were included. Additionally we filtered out all bullet chess games. This was so we had a better record of ”good” moves made at each position. Filtering over such a massive dataset and then transforming the full games to individual boards was a challenge in its own right.

We set up a Pinecone index (8 pods of type s1.x2) with the following schema:

```
{
    'id': <FEN>,
    'vector': <ChessPOS embedding>,
    'metadata': {'move': <move>}
}
```

where the Forsyth–Edwards Notation (FEN) of the board position, is the ID. This is so we can ensure unique boards on the index. The vector is the ChessPos embedding of that board, and move is the move in UCI format made by a human player in that position from the dataset.

Our index was only able to hold about 120M vectors with this configuration. A larger database would have been desired but this was already stretching our $250 Pinecone budget.

Now we need a way to use these embeddings to find good moves for our chess engine. A naive assumption might be:

Assumption 1 *If position X is similar to Y, then a good move played at X will likely be a good move for Y.*

3.2. Chesscone

Based on this assumption we made a simple chess engine: *Chesscone*. Chesscone works by simply querying the current board embedding to the database for K nearest neighbors. It will then try to play each move made at those positions until it finds one that is legal. If it finds no legal positions, then it just plays a random move. Chesscone is really just an extension of an opening book and endgame tablebase, trying to extrapolate to ”closest move” if no perfect match is found.

3.3. ChessBERT

However there are several problems with our first assumption. It depends heavily on the metric used for similarity. If the two similar positions have some key differences good moves for each may be wildly different, and the move made at X may not even be a valid move at Y.

This leads us to a more refined assumption:

Assumption 2 *If positions $X_1, X_2, \ldots, X_k$ are similar to Y, then some shallow function of $(X_1, \ldots, X_k)$ can be used to find a good move for Y.*

We train an architecture similar to the BERT [7] model known for its success in masked-language modelling with key architectural differences which we use to engineer inductive priors for modelling chess positions.

Specifically, we represent each in-context chess position as a bag of chess pieces. The entire input to the ChessBERT model takes the form

$$\texttt{Bag}_1 \cdot \texttt{OptMove}_1 \cdot \ldots \cdot \texttt{Bag}_k \cdot \texttt{OptMove}_k \cdot \texttt{QueryBag} \cdot \texttt{[MASK]}$$

where the input embedding for each chess piece is obtained by summing trainable vectors corresponding to its

1. row: We maintain 8 trainable vectors corresponding to each row in the chess board.
2. file: We maintain 8 trainable vectors corresponding to each file in the chess board.
3. piece: We maintain 32 trainable vectors for each distinct piece including 8 distinct pawns and 8 major chess pieces for each color.
4. segment: We use segment embeddings to demarcate different parts of the input to the ChessBERT model.

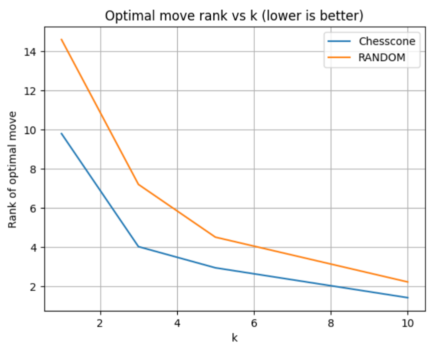
Formally, we obtain the input embedding of piece P placed at row R and column C in segment S as

$$\begin{aligned}\texttt{FullEmbedding}(P, R, C, S) = \\ &\texttt{PieceEmbeddings}[P] + \\ &\texttt{RowEmbeddings}[R] + \\ &\texttt{FileEmbeddings}[C] + \\ &\texttt{SegmentEmbeddings}[S].\end{aligned}$$

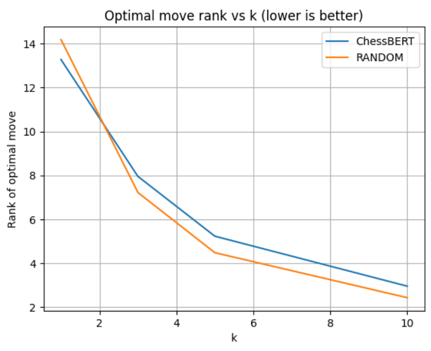
We use distinct segment embeddings for each in-context demonstration, each in-context optimal move, the query board and the mask token whose ChessBERT encoding we consider the model's prediction.

We use the same scheme to encode optimal moves as well to capture information about which piece to move and its destination square. This encoding is consistent with how chess moves are represented in standard PGN algebraic notation for chess boards. For instance, the notation `Ke2` denotes that the king is moved to the square the intersection of the e file and the 2nd row.

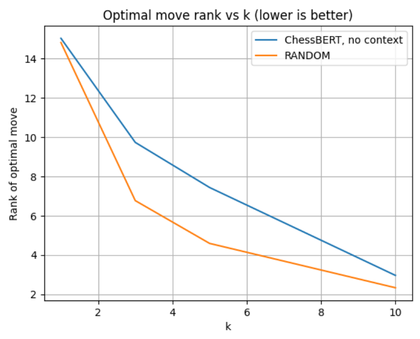
We train a randomly initialized ChessBERT until convergence using a contrastive CLIP [8] loss, trying to ensure that its final encoding of the `[MASK]` token matches the `FullEmbedding` corresponding to the ground truth optimal move for the position represented by `QueryBag`.



(a) Chesscone

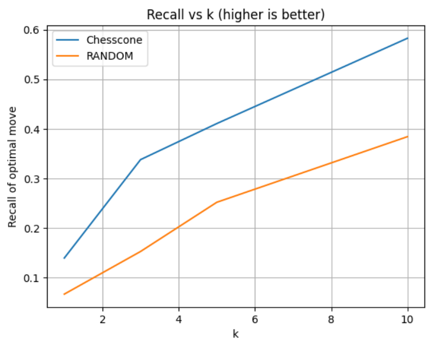


(b) ChessBERT (with context)

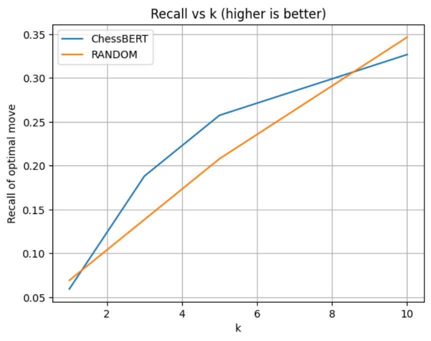


(c) ChessBERT (no context)

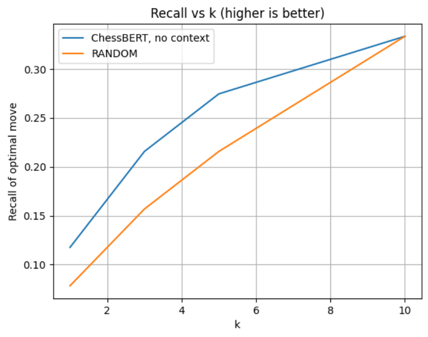
Fig. 1: Average rank of top move in games vs RANDOM



(a) Chesscone



(b) ChessBERT (with context)



(c) ChessBERT (no context)

Fig. 2: Average recall@K of top move in games vs RANDOM

4. EXPERIMENTS

4.1. Metrics

We found that evaluating our chess engines against human players was: 1) too time-consuming to be feasible on a large scale; 2) meaningless as our engines were too weak to properly evaluate using ELO rating. Thus, we introduce 3 metrics to automatically evaluate engine strength.

Average Rank of Top Move We wish to measure the strength of the top engine move since engine play exclusively uses the top move. For a given position, we order all possible legal moves by centipawn value using Stockfish engine and compute the rank of our engine's top move. We report the average top move rank for all positions in the test set.

Recall at K Recall at K is a measurement of the probability that the top K recommendations of the engine contains the optimal move. We define the optimal move in this setting as the top recommendation by the Stockfish engine.

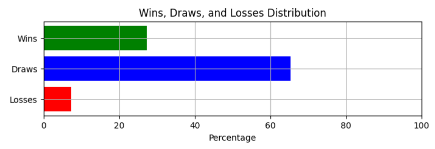
Winrate against RANDOM policy Finally we simulate several games against a random policy and report the winrate of our chess engines. Because a random policy often plays illogically, we expect these games to encounter positions not seen during training.

4.2. Results

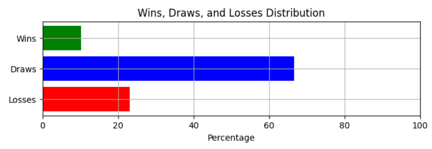
We evaluate using a test set of randomly drawn positions from the lichess dataset. These positions are not included in both the training data and the vector database. Figures 1-6 show the performance of our engines.

As shown in figures 1-3, Chesscone is significantly better than the random policy in every metric. We empirically observe that Chesscone is quite reasonable during the opening of the game, but is far weaker during the middlegame when the query position is unlikely to be seen in the retrieval results. In the endgame it is more likely to have seen the exact boards again and so it can actually mate its opponent. This shows how Chesscone is like an extension to and opening book or endgame tablebase.

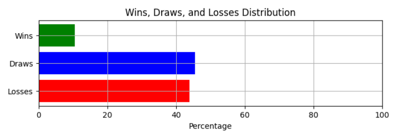
On the other hand, ChessBERT is only on-par or even slightly worse than random policy, even with the added help of the context boards, as shown in figures 4-6. We hypothesize that a lack of training time and data severely impacted performance. Furthermore, we believe that the InfoNCE loss function does not precisely fit our task as two distinct boards can both correspond to the same optimal move. Thus, a loss function such as masked margin softmax [9] may be more



(a) Chesscone



(b) ChessBERT (with context)



(c) ChessBERT (no context)

Fig. 3: Win-rate in games vs RANDOM

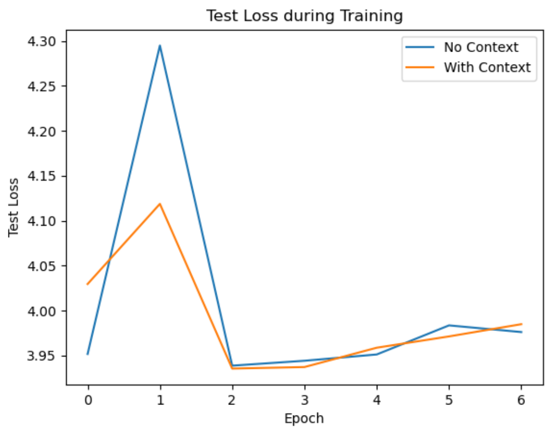


Fig. 4: ChessBERT test loss during training

suitable.

4.3. ChessBERT with no Context

We now demonstrate that training without context boards further degrade ChessBERT performance. We train ChessBERT for the same number of epochs without providing context boards. We summarize the performance of ChessBERT with no context in figures 7-9. We also plot the test loss of with-context training and no-context training in figure 10.

We note that ChessBERT with no context is significantly weaker against RANDOM policy and has worse average rank. However, both recall@K and test loss are quite similar. We hypothesize that ChessBERT with context essentially performs a very weak form of learning from expert advice. Since the context boards are rarely direct matches with the query, ChessBERT's top recommendation rarely matches Stockfish optimal move and so both recall@K and test loss see little benefit. On the other hand, the context boards provide reasonable alternatives to the best move and so average rank and winrate improve.

5. CONCLUSION

We implement 2 retrieval-based chess engines and demonstrate that a simple KNN retrieval can outperform random policy. Additionally, we show that retrieval-augmented training significantly improves results when compared to baseline training. Although our engines are not significantly stronger than the RANDOM policy, we believe that there is room for exciting future work towards creating retrieval-based chess engines.

6. REFERENCES

[1] Patrick Frank, "chesspos: embedding learning for chess positions," .

[2] Alan Turing, "Intelligent machinery (1948)," *B. Jack Copeland*, p. 395, 2004.

[3] Yu Nasu, "Efficiently updatable neural-network-based evaluation functions for computer shogi," *The 28th World Computer Shogi Championship Appeal Document*, 2018.

[4] Berk Kapicioglu, Ramiz Iqbal, Tarik Koc, Louis Nicolas Andre, and Katharina Sophia Volz, "Chess2vec: Learning vector representations for chess," *CoRR*, vol. abs/2011.01014, 2020.

[5] Elad Hoffer and Nir Ailon, "Deep metric learning using triplet network," 2018.

[6] lichess, "Lichess open database," *https://database.lichess.org/*.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[8] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever, "Learning transferable visual models from natural language supervision," 2021.

[9] Gabriel Ilharco, Yuan Zhang, and Jason Baldridge, "Large-scale representation learning from visually grounded untranscribed speech," 2019.