Chessformer: A Chess-Playing Transformer Model

Aron Malmborg and Edwin Östlund

***Abstract*—This project concerns the training of a chess-playing transformer model we call *Chessformer*. The first goal is for the model to learn the rules of the game and play legal moves, as well as to try to win the game. Secondly we train the model on human games only as we want the model to develop a more human style of play than traditional chess bots. Chessformer is trained to predict the next move in a dataset of 16 million human chess games, and through this learns to play both legal and strategic moves. In random positions outside the training data the best version of Chessformer plays a legal move $99.2\% \pm 0.1\%$ of the time. Playing against Stockfish our model achieves an Elo rating of 1198 ± 22. Based on an analysis of its moves Chessformer has a playing style very different from both humans and Stockfish, despite being trained on human chess games.**

***Sammanfattning*—Detta projekt behandlar träningen av en schakspelande transformer modell som vi kallar *Chessformer*. Det första målet är att modellen ska lära sig spelreglerna och spela regelrätta drag, samt att försöka vinna spelet. Det andra är att vi endast tränar modellen på mänskliga spel då vi vill att modellen utvecklar en mer mänsklig spelstil än traditionella schackbotar. Chessformer tränas att förutspå nästa drag i ett dataset av 16 miljoner schackmatcher mellan människor, och lär sig genom detta att spela regelrätta och strategiska drag. I slumpvalda positioner utanför träningsdata spelar den bästa versionen av Chessformer regelrätta drag $99.2\% \pm 0.1\%$ av gångerna. Under spel mot Stockfish uppnår vår modell en Elo-rating på 1198 ± 22. Baserat på en analys av dess drag har Chessformer en spelstil som skiljer sig avsevärt från både människor och Stockfish, trots att den tränas på matcher mellan människor.**

***Index Terms*—Chess, Transformers, Language models, Deep learning**

Supervisor: Amaury Gouverneur

TRITA number:

I. Introduction

Chess-playing programs based on explicit rules or neural networks trained through reinforcement learning can outclass the best human chess players. However, such programs often play in a distinct style different from that of human players. Here we pursue a different approach for chess-playing programs: training a transformer on human chess games. The hope is that this yields a model with a more human-like playing style which can thus be more useful as a coach or learning tool for human players.

We call the resulting model *Chessformer*. Chessformer exists in two versions, Chessformer-small and Chessformer-medium, with the only difference being the number of transformer layers in the models.

We use a GPT-2 style decoder-only transformer, training it on 16 million chess games from the Lichess database to predict the next move given a sequence of earlier moves. Through this, the model learns to play legal and strategic chess moves as an emergent ability. In order to assess the model's chess-playing ability we have it play against Stockfish, a well-known chess engine, and use its average score against Stockfish to calculate an Elo rating for both Chessformer models.

Since we are interested in the playing style of Chessformer and whether it is more human-like than that of Stockfish we also look at the frequencies of moves they make and compare this to humans, as well as the quality of moves and how this is distributed.

Our work relies greatly on the work of Adam Karvonen [1], and is to a large extent an attempted replication of his methods and results. Karvonen also trains transformer models to play chess, and assesses the rate at which they play legal moves and their strategic ability in play against Stockfish. Whereas Karvonen is mainly interested in the internal representations learned by his models, we are more interested in the playing style of ours, and whether it is more human than other chess engines like Stockfish.

The rest of the report is structured as follows. Section II provides an introduction to computer chess, Elo ratings, and reviews the work and results of Karvonen[1]. Section III describes the transformer model architecture we use and all the components in technical detail. Section IV specifies the methodology for training and evaluating our model. In section V the results are presented, and lastly a discussion and commentary on the results follows in section VI. Section VII concludes.

II. Background

A. Computer Chess

Researchers in computer science and AI have long been interested in chess-playing computer programs. Classical chess programs rely on searching the game tree constructed from possible moves and evaluating positions using some *evaluation function* hand-crafted by chess experts. Because the number of possible moves and positions in chess is so large (Claude Shannon estimated the number of possible chess games as 10^{120} [2]) a brute-force approach is implausible and the search process is therefore made more efficient through *alpha-beta pruning* or performed randomly based on statistics using *Monte-Carlo tree search* (MCTS).

The computer chess program Deep Blue, developed by IBM, defeated world chess champion Garry Kasparov in 1997. Deep Blue combined alpha-beta pruning with an evaluation-function fine-tuned by experts on the basis of their experience and analysis of past games. This is an example of an *expert system* AI designed by human experts to accomplish a single task but incapable of transfer.

Since then computer chess programs have come to increas-

ingly rely on machine learning methods. MCTS was famously combined with deep learning to create AlphaGo, the Go-playing system which defeated world champion Lee Sedol in 2016 [3]. Stockfish has for a long time been the premier chess-playing system. It used to be based on a hand-crafted evaluation function, like Deep Blue, but in 2020 it switched to using an efficiently updatable neural network (NNUE) trained through to evaluate positions [4]. This network was originally trained through reinforcement learning in self-play, with later versions trained through supervised learning on games generated by older versions of Stockfish.

Chess engines like Stockfish typically include *skill level* as a parameter. In the case of Stockfish this parameter affects the search depth used for the evaluation function and the rate at which the model selects suboptimal moves per the evaluation function [5]. This logic gives Stockfish a rather distinct style of play at lower skill levels where incredibly good moves might suddenly be followed by blunders that no competent human player would make, Throughout this paper we use the lowest skill level of Stockfish, which we refer to as Stockfish-0, in order to evaluate our own model. To compare the performance of different chess players and chess-playing programs we use the Elo rating system.

B. Elo Rating System

The Elo rating system is used to compare the skill levels of players in any zero-sum game by assigning a numerical score to each player. The system was first devised by Arpad Elo for use in chess [6]. It is used by the International Chess Federation (FIDE) and online chess sites like Chess.com and Lichess to rank and compare chess players. Chess grandmasters are generally rated at 2500 or more while novice players have an Elo of about 1000. Currently the highest rated human chess-player is Magnus Carlsen, with an official Elo of 2837 at the time of writing, while the best version of Stockfish has an Elo of 3641 [7, 8].

Elo's approach is based on treating the performance of a player in any given game as a normally distributed random variable [6]. We let S_A and S_B be random variables representing the score of player A and player B when playing against each other. We have that $S_A = 1$ if A wins, $S_A = 0$ if A loses and $S_A = 0.5$ if there is a draw. The scores of both players must together sum to one: $S_A + S_B = 1$.

The expected score of player A when playing against player B is then determined by their relative Elo ratings. If R_A is the rating of player A and R_B is the rating of player B, then the expected score of A is given by

$$\mathbb{E}[S_A] = \frac{1}{1 + 10^{(R_B - R_A)/400}}. \tag{1}$$

After a game the Elo score of a player is updated based on the result

$$R_A \leftarrow R_A + K(S_A - \mathbb{E}[S_A]) \tag{2}$$

where the K-factor is the maximal adjustment after a game, set to $K = 16$ for masters and $K = 32$ for novices. If the Elo score of another player is known we can use equation (1) to estimate the Elo score of a player using the average score $\hat{\mathbb{E}}[S_A]$ over many games:

$$\hat{R}_A = \hat{R}_B + 400 \cdot \lg\left(\frac{\hat{\mathbb{E}}[S_A]}{1 - \hat{\mathbb{E}}[S_A]}\right). \tag{3}$$

Since $S_A + S_B = 1$ and so $\mathbb{E}[S_B] = 1 - \mathbb{E}[S_A]$ we can rewrite the expression in the logarithm as the quotient of average player scores $\hat{\mathbb{E}}[S_A]/\hat{\mathbb{E}}[S_B]$. We can use equation (3) along with the already estimated Elo rating of a player to estimate the Elo rating of another player.

There are some difficulties associated with comparing the Elo scores of human players and chess-playing programs. The performance of a chess engine like Stockfish may depend not only on what skill level we set but also on the time we give the model per move. Stockfish generates moves by evaluating positions using a search in the game tree, where the extent of this search is limited by the time given to the model. Thus a model given more time per move could generate higher quality moves. Since we want to generate a large number of games we are running Stockfish-0 with a 0.1 s time limit per move, as Karvonen does [1].

C. Previous work

Our work builds on the work of Karvonen [1]. Karvonen trained two transformer models on 16 million chess games fetched from the Lichess games database, resulting in models that can play legal and strategic chess moves, and which has learned an internal representation of the board. Karvonen's best model achieved a 99.8% legal move rate and 64% win rate versus Stockish 0, which he estimates to have en Elo score of $\sim$ 1300, indicating that the model has an Elo score of $\sim$ 1400.

Our work differs from and builds on Karvonen's in that we are more concerned with the playing style of the resulting models, whereas Karvonen was most concerned with the internal representations learned by the models.

III. Transformers

Transformers are a neural network architecture which uses a self-attention mechanism to process sequential data such as natural language text [9]. The sequence is divided into individual *tokens*, typically words or subwords in the natural language context, which are then projected into an embedding space before being processed by several self-attention layers in sequence.

These models have demonstrated remarkable performance on many natural language processing tasks, and form the basis of *large language models* (LLMs) like the GPT and Claude series. Typically a transformer is trained through self-supervised learning on next-token prediction.

In the chess context the transformer is given a string of chess moves in the SAN format, tokenized into individual moves, and tasked with predicting the next move. Thus the model is not actually tasked with winning at chess, but learns to do so as an *emergent ability* [10].

A. Token Embedding

To convert the input tokens $\mathbf{X}$ into a numerical representation representing their semantics, the tokens are first fed into an

embedding layer to produce embeddings $\mathbf{E}$: $\mathbf{E} = \mathbf{W}_E\mathbf{X}$. Here $\mathbf{W}_E \in \mathbb{R}^{V\times d}$ is a learned embedding matrix, where V is the vocabulary size and d is the model dimension. Embedding essentially projects the tokens into a high-dimensional space where their semantics can be encoded per their statistics in the training data. Therefore the embedding is not specified but learned as part of the model.

B. Position Embedding

To properly capture the position of each token in the sequence we use sinusoidal position embeddings

$$\begin{aligned}\mathbf{P}_{(\mathrm{p},2\mathrm{i})} &= \sin\left(\frac{p}{10000^{\frac{2i}{d}}}\right)\\ \mathbf{P}_{(\mathrm{p},2\mathrm{i}+1)} &= \cos\left(\frac{p}{10000^{\frac{2i}{d}}}\right)\end{aligned} \tag{4}$$

where p is the position in the sequence, i is the hidden dimension index, and d again is the dimension of the model. Thus $\mathbf{P}$ will be a $L\times d$ matrix, where L is the sequence length. The exponent in the denominator allows the positional embedding to capture different frequencies of positional structure. We then add these positional embeddings to get the first hidden state activations $\mathbf{H}_0 = \mathbf{E} + \mathbf{P}$. This particular position embedding is called rotary position embedding [11].

C. Layer Normalization

Before the self-attention layer the activations are normalized:

$$\tilde{\mathbf{H}}_0 = \frac{\mathbf{H}_0 - \hat{\mathbb{E}}[\mathbf{H}_0]}{\sqrt{\mathrm{Var}[\mathbf{H}_0] + \epsilon}} * \boldsymbol{s} + \boldsymbol{t} \tag{5}$$

where $\boldsymbol{s}$ is a learned scaling parameter, $\boldsymbol{t}$ is a learned shift parameter, and the statistics are taken over the batch. This promotes faster training and has a regularizing effect which improves generalization [12].

D. Self attention

Self attention is a mechanism used by transformer models to discover relationships and dependencies between all tokens in the input data. The structure of a transformer model is shown in figure (1).

The input data is segmented and each token assigned a query, key and value vector. The query vector represent what information is desired at that token, and the key vector represents the information it contains. The value vector encodes the actual data content of the token.

The dot product of the key and query vectors represent how closely related the tokens are, and this mechanism allows for calculations of the attention all tokens should pay to all other tokens. A probability distribution is obtained by scaling the dot products and applying the softmax function. The softmax function performs natural exponentiation on each element in the input vector and subsequently normalizes the vector. The weights of the model can then be calculated using

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{6}$$

where Q, K, and V are matrices containing query, key and value vectors respectively and d_k is the dimension of the key/query vectors[9].

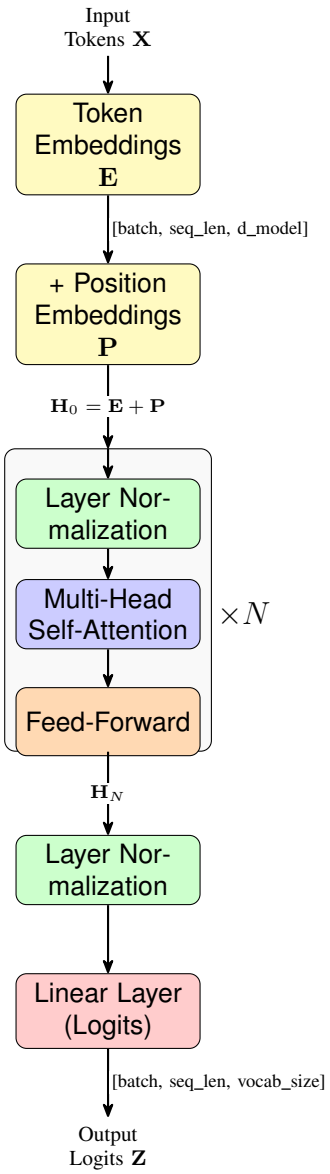

Fig. 1. GPT-2 style encoder-only transformer with N transformer blocks and pre-attention layer norm.

This mechanism is then enhanced further with multi-headed attention in which several key/query pairs are created. The pairs focus on different aspects of the data and each head performs self attention. The results of all heads are then combined and fed forward to the final layers.

E. Output and Inference

After the last transformer block the hidden state $\mathbf{H}_N$ is fed through a final layer normalization step and linear layer to produce the output logits $\mathbf{Z}$. In principle this is the final output of the model, i.e. the basis for computing the loss and updating the model. However to make predictions and perform inference we first convert these logits $\mathbf{Z}$ to a probability distribution over the tokens in the vocabulary using the softmax function where the summation is over the vocabulary.

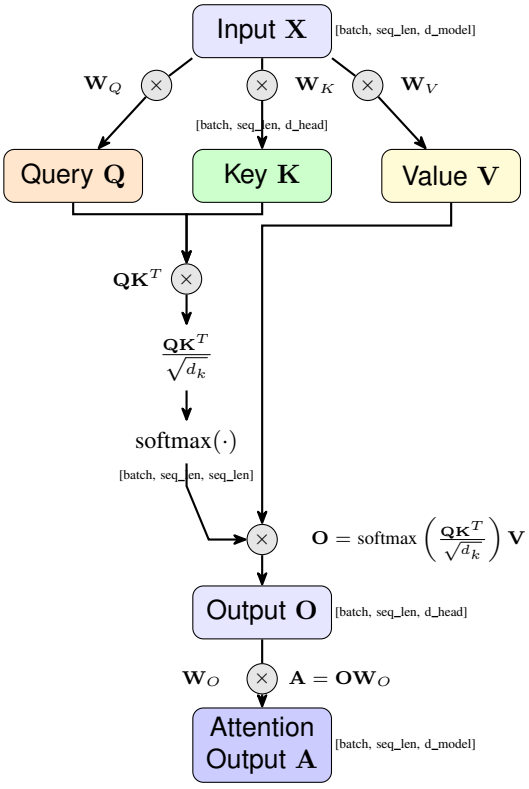

Fig. 2. Single head of self-attention. In multi-head attention we concatenate several attention outputs.

IV. Methodology

A. Data

The dataset we use to train our chess-playing transformer consists of human chess games in the *portable game notation* (PGN) format drawn from the Lichess database [13]. PGN is a text format for recording chess games. Each move is recorded in *simple algebraic notation* (SAN), where a typical move specifies the piece move and the square it moves to. Special characters for captures, checks, checkmates and final scores are also included when applicable. The PGN also includes the Elo of both players and the result of the game.

Table (I) shows some summary statistics for the dataset of chess games used. The average length of games in the dataset is 36.8 full moves or 110.4 tokens per our tokenization scheme. The average Elo of players in the dataset is 1650 ± 289.

TABLE I
Summary statistics for the Lichess dataset

Statistic	Value
Mean Elo	1650 ± 289
Median Elo	1648
Mean length	36.8 ± 14.2
Median length	34
Number of games	$1.6 \cdot 10^6$

Figure (3) shows the distribution of game lengths in our dataset. To save on processing time and memory we have decided to limit the context length of our transformer to 256 tokens, which corresponds to 85 full moves with three tokens per move. As the figure shows this excludes only a small number of games.

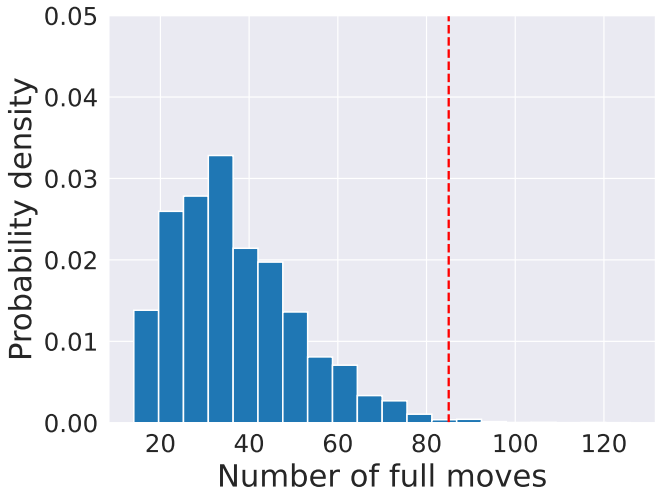

Fig. 3. Distribution of game lengths in our dataset. Chessformer context is marked by a red line.

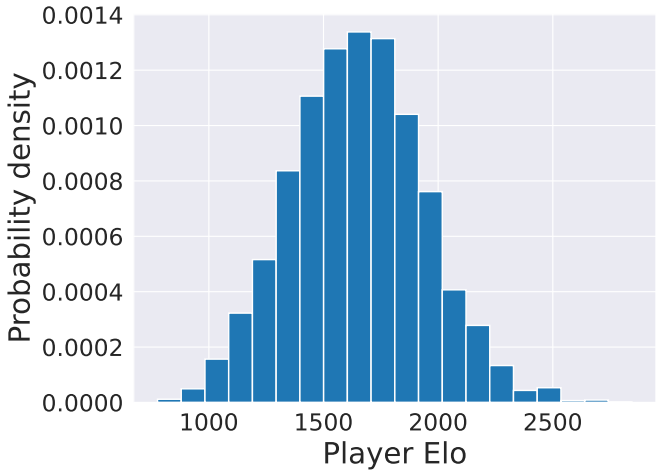

Fig. 4. Elo of players in our dataset

Figure (4) shows the Elo distribution among the white player for games in our dataset. One might suspect that a model trained on games in this distribution would achieve the median or average Elo score.

B. Model architecture

We use the same transformer architectures as Karvonen [1], the details of which are shown in table (II). In particular, we use GPT-2 style decoder-only transformers with $h = 8$ heads, $d = 512$ as the model dimension, and $n = 8$ layers for Chessformer-small and $n = 16$ layers for Chessformer-medium. This gives Chessformer-small $2.63 \cdot 10^7$ parameters and Chessformer-medium $5.15 \cdot 10^7$ parameters in total.

C. Training

1) Objective

Chessformer is trained to predict the next token in a sequence of chess moves. Through this it is supposed to learn to play like the human players in the dataset

The loss function we use to optimize the model is the cross-

TABLE II
MODEL ARCHITECTURES

Parameter	Small	Medium
Model dim. d	512	512
Context length l	256	256
# heads h	8	8
# layers n	8	16
# params	$2.63 \cdot 10^7$	$5.15 \cdot 10^7$

entropy loss

$$\mathcal{L}(\theta) = -\hat{\mathbb{E}}_{\mathcal{D}} \left[p(x) \cdot \ln \hat{p}(x|\theta) \right] \tag{7}$$

where $\hat{\mathbb{E}}_{\mathcal{D}}$ denotes an average over our dataset $\mathcal{D}$, $p(x)$ is the true distribution of tokens and $\hat{p}(x|\theta)$ is the model distribution over moves given the parameters θ. The goal of training is to find parameters θ that minimize the the loss $\mathcal{L}(\theta)$.

2) Tokenization

We use a simple tokenization scheme in which each move is treated as a separate token. This differs from Karvonen's approach, since he uses character-level tokenization, treating each character as a separate token [1]. Move level tokenization allows the model to bypass learning the structure of individual moves in order to focus on the structure of the game itself. This results in a vocabulary of 1855 tokens, including special tokens for padding, unknown tokens and so on.

3) Optimizer

We use the standard AdamW optimizer to train our model [14]. This optimizer combines the ideas of *momentum*, whereby weight updates are made stronger if they tend to align in the same direction, with *root-mean-square propagation* (RMS-prop), where the update is scaled down by a moving-average of squared gradients in order to improve stability.

Additionally, AdamW applies weight decay regularization directly on the weights instead of as part of the gradient update, as the Adam optimization algorithm does, which decouples regularization from the adaptive learning rate.

function *AdamW*(γ(*lr*), β_1, β_2, θ_0(*params*), $\mathcal{L}(\theta)$, ϵ, λ(*weight decay*))
  // Initialize moments
  $m_0 \leftarrow 0$
  $v_0 \leftarrow 0$
  for $t = 1$ **to** ... ***do*** **do**
    // Gradient of loss
    $g_t \leftarrow \nabla_\theta \mathcal{L}(\theta_{t-1})$;
    // Weight decay
    $\theta_t \leftarrow \theta_{t-1} - \gamma \lambda \theta_{t-1}$;
    // Momentum
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$;
    // Moving squared average
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$;
    // Bias correction
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$;
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$;
    // Weight update
    $\theta_t \leftarrow \theta_t - \gamma \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$;
  end
  return θ_t;

Algorithm 1: AdamW Optimization

Here γ is the learning rate, β_1, β_2 control the weight given to the most recent value in computing the moving average, θ_0 are the initial weights and $\mathcal{L}(\theta)$ is the loss as a function of our parameters, computed using a forward pass through the model at each step.

4) Learning Rate Scheduling

To promote training stability and generalization we use a cosine annealing learning rate scheduler [15]. This changes the learning rate as a function of time so that the learning rate decreases for later time-steps. For earlier time-steps we want the optimizer to take fairly large steps in order to learn faster, but for later time-steps we do not want to risk stepping out of local minima in the parameter space.

With cosine annealing the learning rate at step t is set according to

$$\gamma_t = \gamma_{\min} + \frac{1}{2} (\gamma_{\max} - \gamma_{\min}) \left(1 + \cos \left(\frac{t\pi}{\tau} \right) \right) \tag{8}$$

where $\gamma_{\min}, \gamma_{\max}$ are the minimal and maximal learning rates and τ is the desired period of annealing, here the total number of time steps in one epoch.

D. Inference

Once the model is trained we want to use it to generate moves in play vs. itself, other models, or human players. The output of Chessformer is a tensor of logits $\mathbf{Z}$ which can be softmaxed to yield a probability distribution over the next token. One approach is then to always pick the token that maximizes the probability. Another is to sample randomly from this distribution, but where the logits are scaled by a *temperature* parameter T controlling the uniformity of the distribution:

$$\mathbf{x} \sim \text{softmax} \left(\frac{\mathbf{Z}}{T} \right). \tag{9}$$

Here $T \to 0$ corresponds to taxing the argmax, since all components except the largest logit will disappear, while $T \to \infty$ corresponds to a uniform distribution over all tokens.

We utilize this temperature parameter T to generate moves in cases where the model gets stuck in outputting an illegal move, gradually increasing the temperature until the model generates something legal. Thus the model is penalized for predicting illegal moves, since a more random move is likely to be worse. The overall approach for generating moves in games is shown in figure (5).

E. Playing style

One of the aims of this project is to train a transformer model for human-like chess play. We are therefore interested in comparing the playing style of our model to that of the human players in our dataset and chess engines like Stockfish. Fully characterizing the playing style of a chess player would involve in-depth analysis of games. Here we instead rely on simple statistical methods.

To compare playing styles we generate games of self-play with Stockfish-0 and our model and extract the distributions of move frequencies from these. We compare these to the move frequencies from a selection of games in our dataset. We then compute the pairwise *Jensen-Shannon divergence* for all pairs of distributions. The Jensen-Shannon divergence

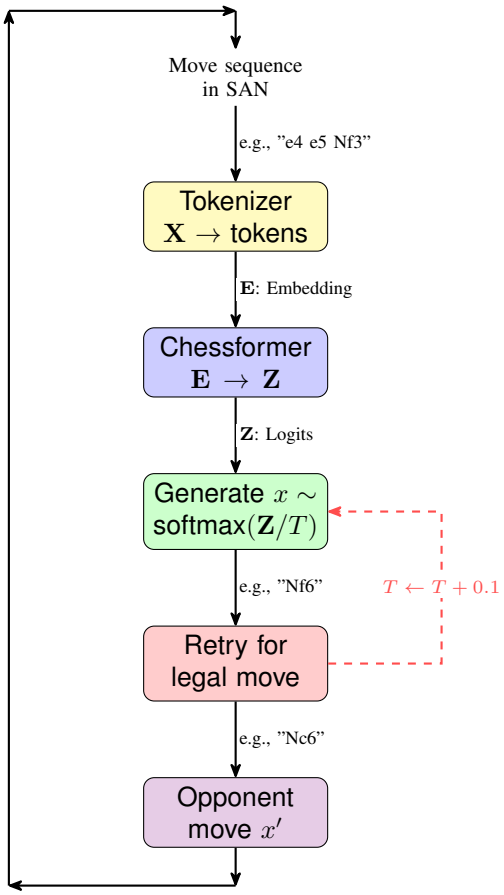

Fig. 5. Generation of moves by a trained Chessformer model

$D_{\mathrm{JS}}(p, q)$ provides a symmetric distance metric for probability distributions p and q, normalized to 1 so that exactly equal distributions have $D_{\mathrm{JS}}(p, q) = 0$ and exactly disjoint distributions have $D_{\mathrm{JS}}(p, q) = 1$. In terms of the Shannon entropy $H(\cdot)$ the Jensen-Shannon divergence is defined

$$D_{\mathrm{JS}}(p, q) = H\left(\frac{1}{2}(p + q)\right) - \frac{1}{2}(H(p) + H(q)). \qquad (10)$$

V. Results

A. Legality of Moves

Table (III) shows the rate of legal moves in random positions achieved by our models. For both models we sample 10000 random positions not present in the training data and check whether the model's first choice of move is legal. Both models achieve a very high rate of legal play, though Chessformer-medium a somewhat greater one.

TABLE III
Rate of Legal Moves

Model	Legality rate
Chessformer-small	$98.7\% \pm 0.1\%$
Chessformer-medium	$99.2\% \pm 0.1\%$

The rate of legal play throughout training is also of interest. Figure (6) shows the moving average of the legality rate of Chessformer-medium throughout training. The model achieves an average legality rate of over 90% after 2000 iterations, and over 99% after 572000 iterations.

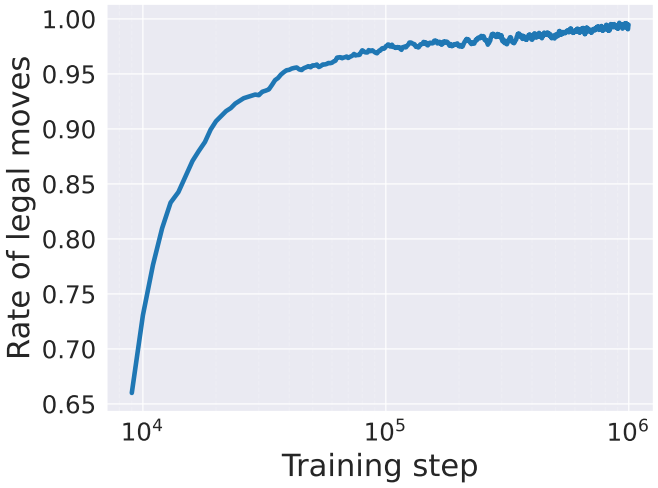

Fig. 6. Legal move rate of Chessformer-medium during training

B. Performance

Table (IV) shows the Elo of each model as measured through playing against Stockfish-0. For each model we play 1000 games vs. Stockfish and then calculate the Elo corresponding to the average score per equation (3), on the basis of an Elo score of 1300 for Stockfish-0. The standard error is then calculated on the basis of bootstrap resampling.

TABLE IV
Elo vs. Stockfish-0

Model	Elo
Chessformer-small	1121 ± 25
Chessformer-medium	1198 ± 22

As seen in the table both models underperform relative to Stockfish-0. The Elo of Chessformer-medium corresponds to a win rate of 36%, while that of Chessformer-small corresponds to 25%.

Figure (7) shows the Elo of Chessformer-medium during training. We measure the Elo every 50000 iterations by playing 100 games against Stockfish-0 and then computing the Elo from the average score using equation (3). There is an upward trend in performance, but with quite large deviations from this trend.

While the Elo rating inferred from play against Stockfish is useful as a summary of the model's capabilities we are also interested in the quality of moves played by our model more generally. To compare this to the quality of moves in human games in our dataset and the quality of moves by Stockfish-0 we take 100 games from each and use the Stockfish evaluation function to measure the change in evaluation of the position from each move. The evaluation is measured in *centipawns*, so that the loss of a pawn corresponds to a -100 change in the evaluation. For the sake of this analysis we define a *blunder* as a move resulting in the loss of 150 centipawns or more. We can then compute the blunder rate for each of our distributions.

Table (V) shows the mean and median change as well as the blunder rate for each distribution. The typical move is slightly negative, but there is a tail of more negative moves which reduces the mean. Stockfish-0 and Chessformer both

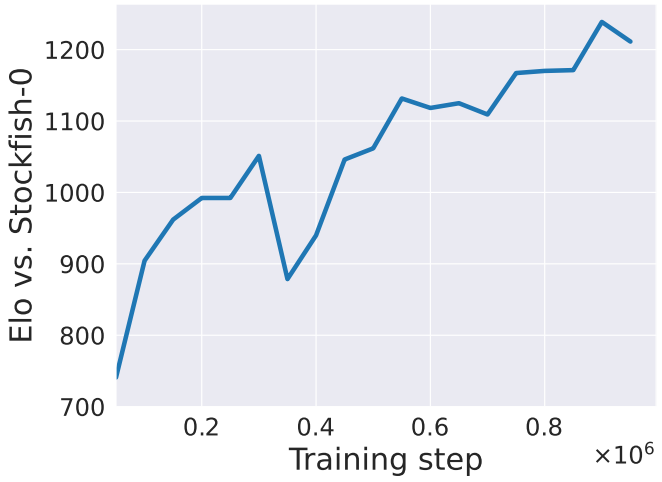

Fig. 7. Elo rating of Chessformer-medium vs. Stockfish-0 during training

have blunder rates greater than the human players, though similar to each other.

TABLE V
MOVE QUALITY STATISTICS

Distribution	Mean	Median	Blunder rate
Human	−40.7 cp	−16.0 cp	0.11
Stockfish	−49.3 cp	−24.0 cp	0.14
Chessformer	−42.6 cp	−11.0 cp	0.14

C. Playing Style

Table (VI) shows the Jensen-Shannon divergence of the distributions generated by our model Chessformer-medium, Stockfish-0, and those found in the human dataset. On the token level our model has a JS-divergence of 0.26 vs. the human distribution and 0.28 vs. the Stockfish distribution, while the divergence between Stockfish and humans is 0.14. The results for bigram and trigram are similar in terms of the relative divergences. Thus our model diverges significantly from both human-like and Stockfish-like play, with the latter two being relatively close to each other.

TABLE VI
JENSEN-SHANNON DIVERGENCES

Model	Token	Bigram	Trigram
Humans vs. Stockfish	0.14	0.42	0.74
Humans vs. Chessformer	0.26	0.62	0.78
Stockfish vs. Chessformer	0.28	0.66	0.81

VI. DISCUSSION

A. Interpretation of Results

Our results show that small transformer models can be trained to play chess legally and strategically, but that their playing style differs from that of human players, despite being trained on human games.

The fact that the model achieves a high rate of legal play so early in training suggests that learning the rules of chess to an acceptable level is quite easy, and may not require anything like a sophisticated representation of the board state in most cases. Despite this the model still generates illegal moves in some situations even after training. Even grandmasters are known to make such mistakes occasionally, but not at a rate close to that of our model. Thus the model can be said to have failed to generalize the rules of chess to the level that a human player would.

Similarly, while the model can play competent chess and can defeat Stockfish-0 in some games, it does not do so a majority of the time and so consequently gets a lower Elo rating. The distribution of move quality suggests that the poor performance of the model is due to sometimes making disastrous blunders, rather than the quality of the typical move. This also suggests a failure of generalization.

Remarkably, the model diverges more from human play than Stockfish, which will make non-intuitive and suboptimal moves to regulate its skill level. As seen in the move quality distributions the blunder rates of Stockfish-0 and Chessformer are very similar.

The particular playing style of Chessformer may be a result of some powerful inductive bias in the model's playing, but no particular mechanism or source of the behavior has been identified. It is possible that more insight into the model's playing style and how it differs from both human and Stockfish could be gained from further probing or human observation by playing against it.

B. Limitations

A problem with the method used for estimating the Elo score of the model is that one may expect the model to generally match the skill level of its opponent, since it is trained not to win but to predict the next move in the sequence. When matched with a lower skill player the model will therefore tend to make lower quality moves. It may therefore be misleading to attribute a single Elo level to the model, since the estimated score would depend on the level of the player it is matched against. Nevertheless, a model capable of playing at a particular Elo level should get an average score of roughly 0.5 when playing against a player at this level. Thus we can still say that our model cannot play at a 1300 Elo level when faced with an opponent at this skill level.

In studying the playing style of Chessformer we have been limited to a statistical analysis of the frequency and quality of moves, rather than a study of specific games or move combinations. Distinguishing between playing styles in this way would be very time-consuming and may require more extensive knowledge of chess than we possess.

Limitations of compute meant that we could not scale the models beyond the size of Chessformer-medium. Based on our results and scaling laws in machine learning we would expect a larger model to perform better, though perhaps with diminishing returns with regards to the number of parameters.

C. Future Work

This project lends itself well to further research and exploration. An obvious extension would be to train larger models with different architectures and to see how much this affects the performance.

Another approach would be to use reinforcement learning to improve the performance of the model. After being trained

on next-token prediction over large datasets, large language models are trained through reinforcement learning from human feedback (RLHF) to exhibit such traits as helpfulness, honesty, and harmlessness. Here we could mimic this approach by adding an RL step with rewards for winning the game. This would plausibly be easier than training an RL-playing chess model from scratch since the model has already internalized the rules of chess and some notion of good play.

One can also imagine exploration of the internal state of the model. Karvonen trained linear probes to classify the state of each of the squares of the board. The probes were remarkably accurate, the best reaching 99.6%, indicating that there had emerged an internal representation of the board within the model. This was then verified by examining the effects of casual intervention of the board state. Karvonen then performed an intervention by sampling a piece the model intends to move and then erasing the said piece from the residual stream by subtracting the relevant encoding vector. New moves generated by the modified model using the same SAN input were found to be 92% legal [1]. This confirmed that the probes had identified an accurate internal representation of the board state. It would be interesting to see if such probes could reproduce Karvonen's results with our model.

Karvonen focused on using linear probes to predict the board state, but the approach generalizes to other chess concepts downstream of this representation like an evaluation of the board state, whether one is in check, control of squares, and so on. To what extent do the concepts learned by a model correspond to the concepts of human chess-players? Our results show that the playing style of the model diverges greatly from that of human players, but what causes this discrepancy?

Chess provides a good testing ground for general interpretability methods, being a more constrained and easily studied domain than natural language. Further studies could reveal much about the inner workings of chess-playing transformers that can be generalized to other domains.

VII. Conclusion

We have trained Chessformer, a chess-playing transformer model. The model is trained to predict the next move but through this manages to make both legal and strategic moves, achieving a legal move rate of $99.2\% \pm 0.1\%$ and an Elo of about 1198 ± 22 as measured through play against Stockfish-0. The emergent playing style is more similar to human than to Stockfish, but differs more from both humans and Stockfish than they do from each other, despite being trained on human games.

Acknowledgment

We would like to thank our supervisor Amaury Gouverneur for his advice and support throughout the project, and for giving us such an exciting idea to work on in the first place.

References

[1] Adam Karvonen. "Emergent World Models and Latent Variable Estimation in Chess-Playing Language Models". In: *First Conference on Language Modeling*. 2024,

[2] Claude E. Shannon and. "XXII. Programming a computer for playing chess". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275.

[3] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489.

[4] Yu Nasu. *Efficiently Updatable Neural-Network-based Evaluation Functions for Computer Shogi*. (2018, Apr). URL: https://oscarbalcells.com/assets/nnue_paper_english.pdf.

[5] Stockfish Team. *Stockfish*. (2025, Apr). URL: https://github.com/official-stockfish/Stockfish.

[6] Arpad E. Elo. *The Rating of Chessplayers, Past and Present*. New York: Arco Pub., 1978.

[7] International Chess Federation. *World Top Players - Top 100 Players April 2025*. FIDE. (2025, Apr). URL: https://ratings.fide.com/top_lists.phtml?list=open.

[8] Computer Chess Rating List. *CCRL 40/15 Rating List*. (2025, Apr). URL: https://computerchess.org.uk/ccrl/4040/index.html.

[9] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017.

[10] Jason Wei et al. "Emergent Abilities of Large Language Models". In: *Transactions on Machine Learning Research* (2022). Survey Certification.

[11] Jianlin Su et al. "RoFormer: Enhanced transformer with Rotary Position Embedding". In: *Neurocomputing* 568 (2024), p. 127063.

[12] Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *ArXiv* abs/1607.06450 (2016).

[13] Lichess. *Lichess Standard Chess Games Dataset*. (2025, Feb).

[14] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization". In: *International Conference on Learning Representations*. 2019.

[15] Ilya Loshchilov and Frank Hutter. "SGDR: Stochastic Gradient Descent with Warm Restarts". In: *International Conference on Learning Representations*. 2017.