Chess as a Testbed for Language Model State Tracking

Shubham Toshniwal[1], **Sam Wiseman**[2], **Karen Livescu**[1], **Kevin Gimpel**[1]

[1]Toyota Technological Institute at Chicago
[2]Duke University
{shtoshni, klivescu, kgimpel}@ttic.edu, swiseman@cs.duke.edu

Abstract

Transformer language models have made tremendous strides in natural language understanding tasks. However, the complexity of natural language makes it challenging to ascertain how accurately these models are tracking the world state underlying the text. Motivated by this issue, we consider the task of language modeling for the game of chess. Unlike natural language, chess notations describe a simple, constrained, and deterministic domain. Moreover, we observe that the appropriate choice of chess notation allows for directly probing the world state, without requiring any additional probing-related machinery. We find that: (a) With enough training data, transformer language models can learn to track pieces and predict legal moves with high accuracy when trained solely on move sequences. (b) For small training sets providing access to board state information during training can yield significant improvements. (c) The success of transformer language models is dependent on access to the entire game history i.e. "full attention". Approximating this full attention results in a significant performance drop. We propose this testbed as a benchmark for future work on the development and analysis of transformer language models.

1 Introduction

Recently, transformer-based language models have stretched notions of what is possible with the simple self-supervised objective of language modeling, becoming a fixture in state of the art language technologies (**???**). However, the black box nature of these models combined with the complexity of natural language makes it challenging to measure how accurately they represent the world state underlying the text.

In order to better measure the extent to which these models can capture the world state underlying the symbolic data they consume, we propose training and studying transformer language models for the game of chess. Chess provides a simple, constrained, and deterministic domain where the exact world state is known. Chess games can also be transcribed exactly and unambiguously using chess notations (Section 2). Most importantly, the form of chess notations allows us to probe our language models for aspects of the board state using simple prompts (Section 3) and without changing the language modeling objective or introducing any new classifiers.[1]

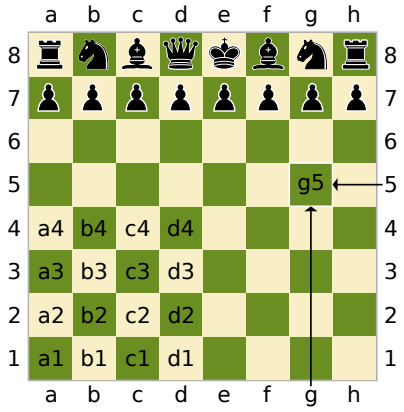
Due to the simplicity and precision of chess, we can evaluate language model predictions at a more fine-grained level than merely comparing them to the ground truth. For example, even if the next move prediction doesn't match the ground truth move, we can still evaluate whether the move is legal given the board state, and if it is illegal, the error can be automatically analyzed (Appendix **??**). Moreover, since world state transitions are deterministic and known, we can evaluate models using counterfactual queries as well. Our proposed evaluation sets and metrics are described in Section 3.2.

While chess represents a controlled domain, it is by no means trivial for a language model. To illustrate the challenges of language modeling for chess, consider the left board shown in Figure 1b, where white is next to move. In order to generate a valid next move, the language model needs to (a) infer that it is white's turn, (b) represent the locations of all pieces, both white and black, (c) select one of the white pieces which can be legally moved, and finally (d) make a legal move with the selected piece. Thus, a language model has to learn to track the board state, learn to generate moves according to the rules of chess, and on top of that learn chess strategies to predict the actual move.

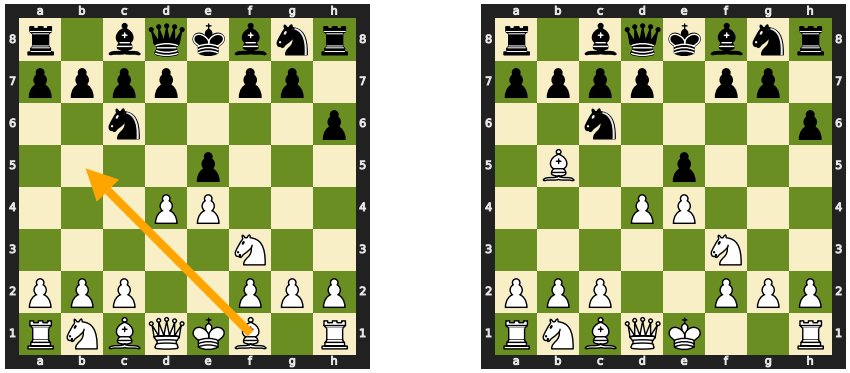
We find that when given enough training data, transformers can learn to both track piece locations and predict legal moves with high accuracy. However, when trained on small training sets, predictive ability suffers. In this more challenging setting, introducing parts of the board state as tokens in the training sequences (Section 3.1) improves piece tracking significantly (Appendix **??**).

Our results also provide some key insights on transformer language models: (i) They are robust to changes in input distribution where additional tokens, related to board state, are added to input sequence *only during training* (Section 3.1). In contrast to LSTMs, transformers achieve this robustness even with smaller training sets (Section 5.3). (ii) Even though chess is Markovian, the model relies on having access to the whole history, and the performance drops when limiting this access (Section 5.3).

[1]Code and data available at - https://github.com/shtoshni/learning-chess-blindfolded



(a) Square naming



(b) Board state before (left) and after (right) the bishop at `f1` is moved to `b5`. UCI notation represents the move as `f1b5`.

Figure 1: Chess Notation

To summarize, our contributions are to:

- Propose chess as a testbed for evaluating world state tracking capabilities of language models which can be used for development and analysis of these models.
- Show that with the appropriate chess notation, we can probe language models for aspects of the world state using simple prompts (Section 3).
- Show that given enough training data, transformer language models can learn to track piece locations and predict legal moves with high accuracy.
- Demonstrate that transformer language models are robust to certain changes in input distribution, and that access to world state during training improves performance with small datasets.

2 Chess Preliminaries

We represent moves using Universal Chess Interface (UCI) notation, which combines the starting square and the destination square to represent a move.[2] The move in Figure 1b is represented as `f1b5` in UCI where `f1` indicates the starting square and `b5` denotes the ending square. While the SAN notation is the standard choice for gameplay, we prefer UCI (see Appendix **??** for why we pick UCI over SAN).

For training language models, we first tokenize games represented in UCI notation using a simple regular expression based tokenizer, which considers a board square symbol such as `b1` as a single token. This gives us a vocabulary of 77 token types, which includes the 64 squares, piece type symbols, and other special symbols (see Table 1).[3] For example, the move sequence "`e2e4 e7e5 g1f3`" is tokenized to "`e2`, `e4`, `e7`, `e5`, `g1`, `f3`". We then train an autoregressive language model on these move sequences, using the standard maximum likelihood objective.

Type	Examples	Count
Square names	`e4`, `d1`	64
Piece type	`P`, `K`, `Q`, `R`, `B`, `N`	6
Promoted Pawn Piece type	q, r, b, n	4
Special symbols	BOS, EOS, PAD	3
Total		77

Table 1: Model Vocabulary

3 Language Model Prompts as Board State Probes

One attractive property of having a language model trained on chess games represented in UCI notation (as described in the previous section) is that the notation *itself* allows us to probe the trained model's state tracking abilities. In particular, by feeding the trained language model a prefix of a game as a prompt, we can determine — using the language model's next-token predictions — what the model understands about the board state implied by this prefix. For example, consider the prompt "`e2e4 e7e5 g1f3 b8c6 d2d4 h7h6 f1`," where the underlined move sequence leads to the left board state in Figure 1b. A language model's next-token prediction (after consuming the prompt) can be interpreted as the ending square predicted for the bishop at `f1`, which can be used to determine the level of board state awareness of the model. If, for instance, the model predicts `g1`, this may indicate that the model does not recognize that the piece type at `f1` is a bishop, as such a move is not possible for a bishop. If, on the other hand, the model predicts `g2`, that may indicate that the model is not aware that another piece is currently at `g2`.

[2] For more details see https://en.wikipedia.org/wiki/Universal_Chess_Interface

[3] In initial experiments we used a delimiter token to indicate move boundary. However, removing it did not degrade performance and made training faster due to reduced sequence length.

Notation	Training	Inference
UCI	e2, e4, e7, e5, g1, f3	e2, e4, e7, e5, g1, f3
UCI + RAP 15	e2, e4, P, e7, e5, g1, f3	e2, e4, e7, e5, g1, f3
UCI + RAP 100	P, e2, e4, P, e7, e5, N, g1, f3	e2, e4, e7, e5, g1, f3
UCI + AP	P, e2, e4, P, e7, e5, N, g1, f3	P, e2, e4, P, e7, e5, N, g1, f3

Table 2: Token sequences corresponding to the move sequence e2e4 e7e5 g1f3 for different notations during training and inference. Notice that regardless of the RAP probability used during training, at inference time the token sequences have no piece types.

3.1 Randomly Annotated Piece type (RAP)

While predicting the token representing the ending-square of a move given a prompt allows us to assess the model's state tracking abilities, it also to some extent conflates the model's understanding of the board state with its understanding of chess strategy. If we could easily probe for where the model thinks a piece *currently* is (rather than where it is likely to end up) given a game prefix, this would allow us to more directly probe the model's state tracking abilities. In particular, we would like to give a language model a prompt such as "e2e4 e7e5 g1f3 b8c6 d2d4 h7h6 N", where N represents knight, and expect it to generate a valid starting position for a knight of the correct color. While UCI notation does not ordinarily include these piece type tokens, to allow for testing the model with such prompts, we propose to randomly include these piece types tokens in moves during training with some fixed probability p. We refer to this strategy as "randomly annotated piece type" (RAP) and use the nomenclature "UCI + RAP p" to indicate that with $p\%$ probability, piece type is part of the move notation during training. Note that for $p = 0$, the notation reduces to UCI.

When *testing* with these starting square prediction prompts, we only include piece type for the prompt, not for any moves in the history. Thus, using RAP during training allows us to probe, at test time, where the model thinks each piece is, given any game history's prefix; by simply providing the desired piece type (e.g., N) the model outputs the predicted starting square for a piece of that type. For example, given the prompt "e2e4 e7e5 g1f3 b8c6 d2d4 h7h6 N", a prediction of f3 or b1 shows that the model is aware of where the knights are.

We also experiment with an "oracle" variant of RAP where piece types are added both during training and testing. We refer to this notation as "UCI + AP " where AP stands for "always piece type". For our running example the equivalent prompt in this notation would be "Pe2e4 Pe7e5 Ng1f3 Nb8c6 Pd2d4 Ph7h6 N".

In terms of the language modeling training objective, addition of RAP represents a distribution change between training and inference. Table 2 illustrates how the use of RAP changes the token sequence during training but not during inference. While there's a distribution mismatch, we hypothesize that addition of RAP can aid the model in learning to track the pieces by providing additional supervision which, in turn, can improve language modeling performance as well.

3.2 Board State Probing Tasks

In this subsection we describe the probing tasks introduced above more concretely. In each probing task we feed the model a prefix of a game followed by a single prompt token, and the model is evaluated based on the highest probability next-token under the model given this context. We show an example of each probing task in Table 3 (which we further describe below), assuming the model has been fed the move sequence prefix e2e4 e7e5 g1f3 b8c6 d2d4 h7h6, which is visualized as the left board in Figure 1b. The actual next move played in the game is f1b5, which takes the white bishop at square f1 to square b5, as shown in the right board of Figure 1b.

3.3 Ending Square Tasks

In this set of tasks, the model is given a game prefix and prompted with the starting square of the next move (f1 in the example of Table 3). The model's next-token prediction represents its prediction for the ending square of this move, which tests the model's ability to track the board state and follow the rules of chess, as well as strategic awareness.[4] We consider two task variants:

1. **End-Actual**: Given a move sequence prefix, the model is prompted with the starting square of the actual piece moved next in the game.
2. **End-Other**: Given a move sequence prefix, the model is prompted with the starting square of any piece on the board that can be legally moved according to the rules of chess.

We evaluate End-Actual predictions in terms of both exact move (ExM) accuracy (whether the model predicted the true ending square, b5 in our running example) and legal move (LgM) accuracy (whether the model predicted a legal ending square for the piece starting at the square in the prompt). For LgM evaluation, we also calculate the R-Precision which is the Precision@R where R is the total number of legal ending squares (**?**). In our running example, there are 5 legal ending squares, and R-Precision will be calculated for the model's top-5 predictions. ExM accuracy evaluation is similar to the typical evaluation of language models on natural language data, while LgM is less stringent and focuses on testing just the model's understanding of chess rules and the board state. Note that for End-Other, only LgM evaluation is available. See Table 3 for examples.

[4] Strategic capabilities of a chess language model are strongly tied to the quality of training games.

Task	Prompt Token	Correct Answers (ExM)	Correct Answers (LgM)
End-Actual	f1	{b5}	{e2, d3, c4, b5 ,a6 }
End-Other	f3	N/A	{d2, g1, h4, g5, e5}
Start-Actual	B	{f1}	{f1, c1}
Start-Other	N	N/A	{f3, b1}

Table 3: Examples of each probing task, as well as the corresponding exact move (ExM) and legal move (LgM) correct answers, are shown below. All examples assume the language model was fed the prefix e2e4 e7e5 g1f3 b8c6 d2d4 h7h6 (see Figure 1b), and that the actual next move was f1b5. While there is only one valid prompt token for both End-Actual and Start-Actual tasks, there are many valid prompt tokens for the other tasks, and we show just one possibility for each. Start-tasks (bottom sub-table) assume the model was trained on games described in UCI+RAP notation.

3.4 Starting Square Tasks

In this category of task, the model is again given a game prefix, but prompted with just the piece type of the next move, such as B for bishop in the example in Table 3. The model's next-token prediction thus represents its prediction for where the prompted piece type currently is on the board. This task tests the model's ability to track pieces.[5] Note that only models which have seen piece types during training, i.e. "UCI + RAP" models, can actually be tested on this task. Also, no piece types are used in the game prefix. We again have two variants of this task:

1. **Start-Actual**: Given a move sequence prefix, the model is prompted with the piece type of the actual piece moved next in the game.
2. **Start-Other**: Given a move sequence prefix, the model is prompted with the piece type of any piece on the board that can be legally moved according to the rules of chess.

We again evaluate Start-Actual both in terms of ExM accuracy (whether the model predicts the starting square of the piece actually moved next in the game), as well as in terms of LgM accuracy (whether the model predicts the starting square of a legally movable piece of the given piece type) and LgM R-Precision (precision of the model's top-R predictions with respect to all of the R starting squares of legally movable pieces of the given piece type). For Start-Other, only LgM evaluation is applicable; see Table 3 for examples.

4 Experimental Setup

Data We use the Millionbase dataset which is freely available and has close to 2.9 million quality chess games.[6] After filtering out duplicate games, games with fewer than 10 moves, and games with more than 150 moves (for the complete game to fit into one transformer window), we are left with around 2.5 million games. From this filtered set we randomly select 200K games for training, 15K games each for dev and test, and another 50K games to create board state probing evaluation sets described in Section 3.2. The dev and test sets are used for perplexity evaluations. The dev set perplexity is used for choosing hyperparameters. From the 200K training set, we create subsets of size 15K and 50K which we refer to as "Train-S" and "Train-M", while the full training set is referred to as "Train-L". For detailed statistics, see Table **??** in Appendix. All the data processing steps requiring chess knowledge, including parsing chess databases, are carried out using python-chess (**?**).

To create the board state probing evaluation sets, we use the 50K games reserved for this task. We only consider prompts for non-pawn pieces since the dynamics of pawns are fairly limited. We ensure that the game prefixes selected are never seen in the training data. The final evaluation set consists of 1000 instances with prefix length (in number of moves) in the range $51 \leq l \leq 100$.

Model Details We use the GPT2-small architecture for our base language model (**??**). GPT2-small is a 12-layer transformer model with 12 attention heads and an embedding size of 768 dimensions. The context size of the model is limited to 512, which is sufficient to cover the longest game in our training set. Note that we only borrow the model architecture; the models themselves are *trained from scratch*. [7]

For the UCI + RAP p models, we tune over $p \in \{5, 15, 25, 50, 75, 100\}$ based on perplexity on the validation set. Note that for perplexity evaluation, logits corresponding to piece type tokens are masked out since piece type tokens are only available during training. We find that $p = 25$ performs the best for Train-S and Train-M, while $p = 15$ is best for Train-L (Figure 2). Larger values of p lead to greater mismatch between training and inference, while smaller values likely do not provide enough training signal.

We also experiment with other transformer and non-transformer models in Section 5.3. Among the transformer models, we experiment with two "approximate" attention models (i.e., models which approximate the full attention of vanilla transformer models), namely, Reformer (**?**) and Performer (**?**). We set the number of layers and attention heads to 12 for both architectures, as in GPT2-small. We also train LSTM language models with and without RAP. For details on hyperparameters and tuning, see Appendix **??**.

Training Details Models are trained for 10 epochs with a batch size of 60. Validation is performed at the end of

[5] In certain cases, this task also tests understanding of chess rules. For example, in Figure 1b only the rook at h1 can be moved.

[6] Download link available at https://rebel13.nl/rebel13/rebel\%2013.html

[7] Colab notebook to play chess against the base language model https://github.com/shtoshni/learning-chess-blindfolded/blob/master/GPT2_Chess_Model.ipynb

Training Set	Model	Dev set	Test set
Train-S	UCI	23.6	23.6
	UCI + RAP	15.9	15.9
	UCI + AP	16.1	16.2
Train-M	UCI	11.6	11.6
	UCI + RAP	10.4	10.4
	UCI + AP	10.1	10.0
Train-L	UCI	7.7	7.7
	UCI + RAP	7.4	7.4
	UCI + AP	7.2	7.2

Table 4: Canonical validation and test set perplexity. By canonical we mean that one move, say `f1b5`, counts as one token.

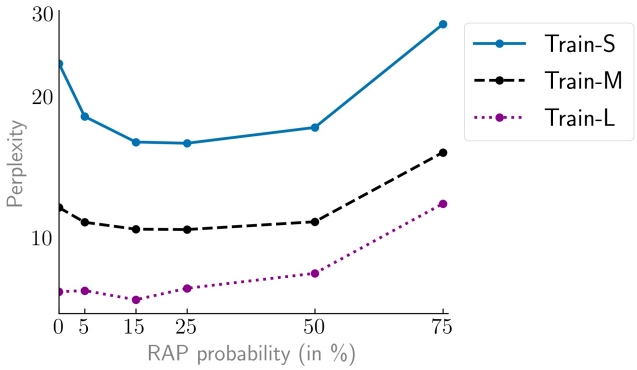


Figure 2: Validation set perplexities as a function of RAP probabilities for the different training set sizes. RAP 0 is the standard UCI notation. RAP 100 is not shown as perplexities are too high.

every epoch and training stops whenever the validation loss starts increasing. For optimization we use Adam (**?**) with learning rate of 5×10^{-4} and L2 weight decay of 0.01. The learning rate is warmed up linearly over the first 10% of training followed by a linear decay. To accelerate training, we use mixed precision training (**?**). All experiments are carried out using the PyTorch Lightning framework built on top of PyTorch (**??**). We use the transformers library (**?**) for all models[8] except for the Performer model for which we use a popular unofficial implementation. [9]

5 Results

We first present language modeling results, where we show significant improvements with the addition of RAP (Section 5.1). Next, we show results on the board state probing tasks for the base language model, where we demonstrate that the model trained on the large training set can learn to track pieces and predict legal moves with high accuracy (Section 5.2). Finally, we present results on the probing task with approximate attention transformer architectures and LSTMs, where we show a performance drop in comparison to the base model with full attention (Section 5.3).

5.1 Language Modeling

Table 4 presents the perplexity results on the validation and test sets. Figure 2 plots the validation set perplexities as a function of RAP probability for different training set sizes. The addition of RAP and AP leads to a decrease in perplexity for all training sizes, particularly for small training sets. For small training sets, RAP probabilities as high as 50% can improve the validation perplexity, but for larger training sets, lower RAP probabilities are preferred. The reductions in perplexity for RAP are surprising given that the extra tokens added via RAP are not present in the validation and test sets, and thus there is a data distribution shift. Models trained with UCI + AP achieve the lowest perplexities on larger training sets. Both RAP and AP aid the model in piece tracking, as we will see in later results, and in the case of chess this can significantly improve the language modeling results as well. Note that for calculating the perplexity of UCI + RAP models, we mask out the logits corresponding to piece type tokens since they are never present during inference.

5.2 Board State Tracking

Tables 5 and 6 show results when predicting starting squares and ending squares, respectively. There are several observations to note. First, **transformers can learn to identify where pieces are located**. This is shown by the LgM accuracies in Table 5. UCI + RAP can predict legal starting positions with perfect accuracy and R-Precision. However, this capability requires Train-L, and the accuracy drops to 91.3% for Train-S. The gap between UCI + RAP and its "oracle" counterpart, UCI + AP, also reduces with an increase in training set size with UCI + RAP achieving parity for Train-L. When asked to identify the location of a piece other than the one selected to be moved next, this accuracy drops only slightly to 99.6%. Typically, the piece location tracking is slightly better for the piece type that is actually moved than for other piece types.

The difference between the location of the piece in the exact move (ExM) and the location of either piece of the given type (LgM) is substantial, at more than 8% absolute. However, this difference relates to chess strategy rather than board state tracking.

Second, **transformers can learn to predict legal moves**. This is shown by the LgM accuracies in Table 6, for which both UCI and UCI + RAP exceed 97% accuracy. However, while the top predictions of the models have high accuracy, their ability to predict all legal moves is significantly lower, with R-precision of about 85%. This is to be expected, since the model is trained on only actual games, where the emphasis is on "meaningful" moves rather than any legal move. Due to similar reasons, there's a significant drop in performance when predicting ending squares for starting squares

[8] Reformer implementation in `transformers` library is still a work in progress. The presented results are with the 4.2.2 version.

[9] https://github.com/lucidrains/performer-pytorch

	Notation	LgM				ExM
		Actual		Other		
		Acc.	R-Prec.	Acc.	R-Prec.	Acc.
S	UCI + RAP	91.3	90.2	89.3	89.2	78.8
	UCI + AP	99.2	99.1	98.8	98.8	86.9
M	UCI + RAP	98.2	98.0	98.6	98.7	88.0
	UCI + AP	99.9	99.8	100.0	100.0	90.2
L	UCI + RAP	100.0	100.0	99.6	99.5	91.8
	UCI + AP	99.9	99.9	99.7	99.7	91.1
Random Legal		-	-	-	-	86.0

Table 5: Accuracies and R-Precisions (%) for predicting starting squares ("Start-Actual" and "Start-Other" tasks). S, M, L in the first column refer to the training set sizes.

	Notation	LgM				ExM
		Actual		Other		
		Acc.	R-Prec.	Acc.	R-Prec.	Acc.
S	UCI	74.0	61.1	65.5	57.7	26.7
	UCI + RAP	88.4	75.5	80.4	72.1	33.3
	UCI + AP	87.0	77.0	78.8	72.3	36.1
M	UCI	92.9	80.6	85.8	78.5	42.2
	UCI + RAP	94.9	82.2	87.9	78.0	45.9
	UCI + AP	94.7	82.4	88.3	79.1	47.3
L	UCI	97.7	85.6	91.9	83.8	52.0
	UCI + RAP	97.0	86.1	93.1	83.9	54.7
	UCI + AP	98.2	87.3	95.2	86.3	56.7
Random Legal		-	-	-	-	19.6

Table 6: Accuracies and R-Precisions (%) for predicting ending squares ("End-Actual" and "End-Other" tasks). S, M, L in the first column refer to the training set sizes.

other than the one in the actual game. The "other" starting square would, by design, have legal continuations, but lack any "meaningful" ones (see examples in Appendix **??**).

We find consistent gains in almost all metrics with the addition of RAP during training, with the gains being particularly impressive for small training sets. Thus, not only are the transformers robust to distribution shift due to RAP (available only during training), they are in fact able to utilize this additional information. Error analysis of illegal predictions shows that the addition of RAP improves piece tracking related errors (Appendix **??**).

The relatively low ExM accuracies of the models can be attributed to the inherent difficulty of the task. Randomly selecting an ending square from all legal ending squares has an accuracy of only around 20%, implying that on average there are roughly 5 legal choices, which might explain the difficulty of the task.

	Model	LgM				ExM
		Actual		Other		
		Acc.	R-Prec.	Acc.	R-Prec.	Acc.
S	GPT2	74.0	61.1	65.5	57.7	26.7
	GPT2 ($w = 50$)	69.5	57.4	60.4	53.2	23.1
	Reformer	71.0	57.2	61.5	53.5	24.8
	Performer	65.4	54.3	57.9	49.5	20.5
	LSTM	60.2	51.0	52.5	46.4	20.9
	LSTM + RAP	59.5	50.5	52.4	46.0	21.9
M	GPT2	92.9	80.6	85.8	78.5	42.2
	GPT2 ($w = 50$)	86.0	74.9	80.9	71.3	35.8
	Reformer	86.4	73.2	76.6	68.6	32.4
	Performer	89.2	76.3	80.5	71.5	36.0
	LSTM	73.8	61.6	67.2	59.8	32.0
	LSTM + RAP	77.5	64.9	69.7	61.7	32.1
L	GPT2	97.7	85.6	91.9	83.8	52.0
	GPT2 ($w = 50$)	95.8	84.5	90.5	82.7	51.6
	Reformer	88.0	74.9	77.0	68.1	33.5
	Performer	95.8	84.5	90.5	82.7	51.6
	LSTM	93.4	79.5	86.1	76.0	45.2
	LSTM + RAP	92.8	80.4	87.3	77.1	46.0

Table 7: Accuracy and R-Precision (%) for predicting ending squares ("End-Actual" and "End-Other" tasks) with varying attention window sizes. LSTM + RAP refers to LSTM trained with UCI + RAP.

5.3 Compressing the Game History

The base transformer language model, based on GPT2, attends to the entire history (i.e., it uses "full attention"), which results in complexity quadratic in the length of the sequence. We might wonder whether attending to this entire history is necessary for the impressive state tracking performance observed in the previous section. We accordingly explore models that do not attend to the entire history in Table 7.

We first experiment with a variant of the GPT2 model that limits its attention to a window of only the 50 most recent tokens ("GPT2 ($w = 50$)"). In Table 7 we see worse performance for this model across data sizes, but especially for small- and medium-sized datasets.

In Table 7 we also consider a language model based on the LSTM (**?**), which considers only its current hidden state and cell state in making its predictions, and does not explicitly attend to the history. Here we find an even more significant drop in performance, in all settings. (Interestingly, we also find that training LSTM language models on sequences with RAP improves performance, but only for larger training sets; transformer language models generally improve when trained with RAP data).

The results of GPT2 ($w = 50$) and of the LSTM language model suggest that attending to the full game history is, unsurprisingly, useful for board state tracking in chess. This finding further suggests that the task of board state tracking in chess can serve as an excellent testbed for recently proposed transformer variants (**???**, *inter alia*) that attempt to make use of long histories or contexts, but *without* incurring

a quadratic runtime.

Approximate Attention Transformers We experiment with the recently proposed Reformer (**?**) and Performer (**?**) architectures. Reformer replaces the "full attention" with attention based on locality-sensitive hashing, while Performer approximates the "full attention" with random features.[10]

The results, in Table 7, suggest that the Performer generally outperforms the Reformer, except in the small dataset-setting. Furthermore, we find that neither of these architectures significantly outperforms the GPT2 ($w = 50$) baseline, except for Performer in the medium-sized data setting. These models do, however, typically outperform the LSTM models. These results demonstrate the challenge of modeling chess with an approximate attention. We hope that future work will use this task as a way of benchmarking more efficient transformer architectures.

6 Related Work

Simulated Worlds. There have been several prior efforts in relating simulated worlds to natural language. The bAbI framework simulates a world modeled via templates to generate question answering tasks (**?**). The recent TextWorld framework facilitates generating, training, and evaluating interactive text-based games (**?**). **?** and **?** develop and use 3D world simulations for learning grounded language. These efforts are similar to our work in the sense that the true world state is, by construction, available, but our setup differs in that it provides a natural way of probing the state tracking of a model trained with an LM objective.

Cloze Tasks for Natural Language Models. There has been a great deal of work on cloze tasks for evaluating natural language models (**??**). These tasks range from testing general text understanding (**?**) to targeting particular aspects of natural language, such as commonsense/pragmatics (**??**), narrative understanding (**?**), and factual knowledge (**?**). Creating these tasks often requires human curation, and the evaluation is typically limited to exact match.[11] Our proposed tasks are a form of cloze tasks, but can be precisely automated so that they require no human curation, and can be evaluated at a fine-grained level.

Probing. One of the goals of this work is to probe the language model's board state tracking capability. A typical solution used by prior work is to train a probing model on top of a pretrained model (**?????**). This setup is time-consuming as it requires training probing models for all tasks. Moreover, the complexity of the probing model can also affect the conclusions (**?**). In our case, by using an appropriate choice of notation, probing for board state can be accomplished via simple prompts (Section 3).

Deep Learning for Chess. Deep networks have been used in prior work to predict the next move given the true game state (**??**). For example, using only self-play and the rules of chess, AlphaZero achieves superhuman performance starting from random play (**?**). The focus of this prior work is the quality of game play given the true board state, while we use chess as a testbed for evaluating a language model's board state tracking capability. Recently there has also been work focusing on transformer language models for chess (**???**). This work is similar to ours in the sense that the input is limited to the move sequence without the true board state, but the focus is again the quality of game play rather than the model's awareness of the underlying state.

7 Conclusion

We propose the game of chess as a testbed for evaluating how well language models capture the underlying world state. We show that with an appropriate choice of chess notation, a language model can be probed for different aspects of the board state via simple prompts. The simple and precise dynamics of chess allow for (a) training models with varying amount of explicit state, and (b) evaluating model predictions at a fine-grained level. Results show that transformer language models are able to track the board state when given enough data, but with limited data, providing access to board state information during training can yield consistent improvement.

Wider Implications for Natural Language Processing. Our results shed light on the following properties of transformers: (a) they are robust to RAP-like changes in input distribution, and (b) for high performance the models require access to the entire context, as well as large training sets (Section 5.3). Future work can use the first finding to introduce the world state, or more specifically the output of linguistic analyzers such as coreference, via RAP-like tokens during pre-training and fine-tuning of transformers. RAP-like tokens can also be used for debugging/diagnosing a model's understanding, similarly to the starting square prediction tasks. The second finding implies that the proposed benchmark can guide the search for new transformer architectures that are adept at understanding long text, and that can learn from small training sets. The proposed framework allows for probing and understanding new architectures that address these challenges.

Acknowledgements

We thank Ed Schröder for permitting us to use the Millionbase database for this project. We thank Allyson Ettinger and colleagues at TTI Chicago for their valuable feedback. This material is based upon work supported by the National Science Foundation under Award No. 1941178.

Magni consequatur debitis architecto natus mollitia molestias rerum vel porro expedita explicabo, ipsum minima nemo reprehenderit tenetur sit exercitationem id?Molestiae voluptatibus veniam asperiores voluptatem amet itaque minima doloremque sint, ipsam similique nemo excepturi pariatur sed, exercitationem pariatur voluptatibus illum quia inventore neque libero.Maiores rem excepturi adipisci assumenda et eos unde facere quis, voluptatem est veniam explicabo asperiores iusto mollitia incidunt, perferendis necessitatibus delectus harum at cum natus laborum modi repre-

[10] In practice, these models often use a combination of the proposed approximate global attention and simple local attention (for details see Appendix **??**).

[11] Automated cloze tasks without human filtering can yield instances which even humans can't answer (**?**).

henderit.Quibusdam quod porro, non quasi excepturi rem placeat commodi perspiciatis unde repellat?Magni accusantium omnis autem iusto animi dolores dolorum eligendi,