ChessLSTM: LSTM-Based Move Prediction from SAN Sequences Project Report: CS 7643

Pranam Kalla, Calvin Ng
Georgia Institute of Technology
pkalla7@gatech.edu, tng49@gatech.edu

Abstract

Deep learning architectures for chess prediction currently rely on large convolutional or transformer networks applied directly to a board tensor, demanding significant computational resources. In contrast, we present a sequence-based model that predicts human chess moves using the SAN (Standard Algebraic Notation) notation of a game. The model treats the game as a language, and predicts the next move through an LSTM encoder with attention. In addition, we enforce move legality through a custom loss function during training and legal move masking during inference. Trained and evaluated on a large dataset of grandmaster chess games, our model emulates human play while solving the computational resource issue other chess architectures face.

1. Introduction/Background/Motivation

The field of chess engines has been dominated by two different model types: classical search algorithms and deep learning models. Classical search engines such as Stockfish choose moves by exhaustively searching through millions of possible future boards. Meanwhile, deep learning models such as AlphaZero are trained through reinforcement learning on a large amount of self-play. Both of these implementations face the same drawbacks: they require enormous computational resources and produce non-human-like play that does not reflect how professional chess players actually play.

Classical chess solvers like Stockfish achieve superior results by brute-force searching combined with evaluation functions. While extremely powerful (and the hallmark of chess engines), they rely on evaluating millions of future positions. This makes classical chess engines computationally expensive while also being fundamentally different from a human way of reasoning. In the other direction, deep learning models such as AlphaZero [6] and Leela [1] rely on massive neural networks (convolutional and transformers) trained on self-play that is optimized for perfect play rather than human play. Both methods of chess engines are resource-intensive and only evaluate based on the current board state, failing to emulate human ideas that are better encapsulated with a series of moves.

To solve these, we present ChessLSTM, an LSTM-based encoder with attention trained using grandmaster-level chess games. The moves are encoded in SAN, which allows our model to treat moves as a sequence and extract patterns and tendencies that human players tend to make. The use of LSTMs for chess-related tasks is very sparse, but the concept is explored in [2] where sequences of moves are used to predict the outcome of a game (win/draw/loss). However, there are no existing works that apply LSTMs directly to next-move predictions for SAN sequences. To the best of our knowledge, no prior paper has addressed the idea of modeling chess as a purely sequential language, so our work aims to address this gap in knowledge.

Game data for grandmaster player games is obtained at PGN Mentor [4], which provides many high-quality chess games in the PGN format. The PGN files comprise of many games in SAN notation (*e.g.* e4, Nd5, O-O) along with the game result, player names, and player ratings. With this information, we filter the games to only contain high-rated players (2500+ ELO). These games can be divided into input–output pairs for the model, where the input consists of the first 10–50 moves of a game and the output is the subsequent move. To increase the size of our dataset, multiple such pairs can be generated from each game. However, to avoid data leakage, all pairs derived from a single game are kept exclusively in either the training or validation set. The SAN notation for each input/output pair is then tokenized into a vocabulary to be used for a language-based LSTM model. This tokenized language representation preserves the order and structure of human decision making in chess, allowing the model to learn patterns more representative of human chess. The PGNMentor database offers both scale and diversity across players, making it a strong dataset to use for a move-prediction model.

With a lightweight LSTM-based model trained to per-

form like human chess players, our goal shifts from "solving" the game of chess to supporting the learning process.

2. Approach

The core idea of our LSTM approach came from recognizing that chess moves written in SAN form a natural language of sorts. When grandmasters play, they are not just making isolated decisions, but rather following strategic patterns and tactical motifs that unfold over sequences of moves. An LSTM seemed like the right tool because it is designed to capture these kinds of sequential dependencies. We thought it would be successful because LSTMs have proven effective in other sequence prediction tasks such as language modeling and time series forecasting. The novelty in our approach is applying LSTMs directly to SAN sequences for next-move prediction, which has not been explored much in the literature. Most work either uses CNNs on board representations or transformers on massive datasets, but we wanted to see if a simpler recurrent architecture could learn meaningful chess patterns from the move notation itself.

2.1. Model Architecture

We built our model with an embedding layer to convert each chess move into a dense vector representation, followed by a two-layer LSTM with attention mechanism to capture both short-term tactics and long-term strategy. The attention layer was crucial because not all previous moves are equally important for predicting the next one. We added dropout at multiple points (embeddings, LSTM layers, and attention) to prevent the model from just memorizing specific games. The final layer outputs a probability distribution over all possible moves, which we then mask using the python-chess library [3] to ensure only legal moves get nonzero probabilities. This legal move masking was essential because without it, the model would waste capacity learning the rules of chess instead of learning good chess strategy.

The model architecture uses 256-dimensional embeddings and 512-dimensional LSTM hidden states across 2 layers, giving us approximately 7 million trainable parameters. We implemented separate dropout rates for different parts of the network, and 0.5 for the LSTM layers and fully connected output, but only 0.2 for the embedding layer. This asymmetric dropout strategy came from the observation that embeddings need to be relatively stable (since they're the foundation of all downstream processing), while the LSTM and output layers benefit from more aggressive regularization. One problem we didn't anticipate was the vocabulary size. With 3,951 unique moves in our dataset, the output layer alone has over 2 million parameters (512 hidden units $\times$ 3,951 moves). This made the model larger than we initially planned and contributed to overfitting. We considered reducing vocabulary by grouping similar moves, but decided against it because it would lose important distinctions between moves.

2.2. Dataset and Preprocessing

Initially, our proposal mentioned using Lichess games, but we quickly realized that including lower-rated games was introducing too much noise. Beginner and intermediate players make moves that do not follow sound principles, and we did not want our model learning bad habits. So we switched to PGN Mentor [4], which provides curated collections of games from top grandmasters. We focused on the top 30 players in the world including Magnus Carlsen, Fabiano Caruana, Levon Aronian, Garry Kasparov, and Shakhriyar Mamedyarov. We filtered games to only include those where the opponent had an ELO rating above 2500, ensuring we were training on high-level play throughout.

The preprocessing pipeline converts each PGN file into training examples. For each game, we extract the sequence of moves in SAN notation and create multiple training samples by taking prefixes of different lengths. For instance, a game that goes "e4, e5, Nf3, Nc6, Bb5" generates samples like "e4" $\rightarrow$ "e5", "e4, e5" $\rightarrow$ "Nf3", "e4, e5, Nf3" $\rightarrow$ "Nc6", and so on. This data augmentation strategy significantly increased our training set size, taking it from about 21,000 games to over 167,000 training examples. We split the data 80/20 for training and validation, making sure entire games stayed in one split to avoid leakage.

2.3. Training Setup and Optimization

The very first thing we tried definitely didn't work. Our initial model had no attention, just a basic 2-layer LSTM with 0.4 dropout and no other regularization. It overfit terribly and the training accuracy shot up to 80% while validation stayed around 50%. We also initially tried to precompute all legal move masks and store them, but this used over 10GB of memory and took 30 minutes just to initialize the dataset.

We used the AdamW optimizer rather than standard Adam because it implements weight decay more correctly, which turned out to be crucial for controlling overfitting. The learning rate was set to 1.5e-3, which is slightly higher than the typical 1e-3 default. We deliberately chose this elevated learning rate to help the model escape local minima during training and chess has a complex loss landscape with many suboptimal solutions, and we found that a bit more aggressive learning rate helped the model find better solutions early on. The weight decay was set to 5e-4, which is fairly strong L2 regularization. This was absolutely necessary given our overfitting problems. Without sufficient weight decay, the model would memorize specific game sequences rather than learning general strategic patterns.

The batch size was set to 512, which is quite large com-

pared to typical LSTM training. We could use such a large batch because we had GPU access and it actually helped with training stability, and larger batches give more reliable gradient estimates, which matters when you're trying to learn complex sequential patterns.

For learning rate scheduling, we used ReduceLROnPlateau monitoring validation accuracy rather than loss. The scheduler reduces the learning rate by half if validation accuracy doesn't improve for 3 consecutive epochs, with a minimum learning rate floor of 1e-7. This is different from the typical approach of monitoring loss and we found that accuracy was a more reliable signal for when the model had plateaued, since loss could keep decreasing due to overfitting while accuracy stagnated. Gradient clipping at norm 1.0 was essential for training stability. LSTMs are notorious for gradient explosion, especially on variable-length sequences, and without clipping we'd occasionally see NaN losses that would ruin entire training runs.

The most important optimization we implemented was Automatic Mixed Precision (AMP) training using PyTorch's GradScaler. This was critical because it allowed us to train with much larger batch sizes and faster iteration times without running out of GPU memory. AMP works by running the forward pass and loss computation in 16-bit floating point (half precision) instead of 32-bit, which cuts memory usage roughly in half and speeds up computation on modern GPUs. The backward pass still uses 32-bit precision where it matters for numerical stability. The GradScaler handles the tricky parts automatically, and it scales the loss up before backpropagation to prevent underflow in the small gradients, then unscales the gradients before clipping and optimizer steps. This is why we unscale gradients explicitly before gradient clipping and we need to clip based on the true gradient magnitudes, not the scaled versions. Without AMP, we could only fit batch sizes of 128-256, but with it we could use 512, which significantly improved training efficiency and final model quality. The speedup was substantial and each epoch went from about 8 minutes down to 3-4 minutes on the GPU.

2.4. Implementation Details

We built our implementation from scratch using PyTorch, but we relied heavily on the python-chess library [3] for legal move generation and board state management. This library handles all the complex chess rules like castling, and pawn promotion, which would have been difficult to implement ourselves. We used it specifically in our legal move masking function, where we reconstruct the board state from a sequence of moves and query all legal moves at that position.

For data loading, we used PyTorch's Dataset and DataLoader classes with a custom collate function to handle variable-length sequences. The key modification we made was implementing lazy mask computation. Rather than checking if the model's move choice is legal at training time, we pre-generate a mask of legal moves at data creation time. This allowed us to trade way slower ($5\times$ slower) training time for just a little bit of pre-computation time.

The attention mechanism implementation was adapted from standard sequence-to-sequence attention, but simplified since we only need to attend over the input sequence, not do encoder-decoder attention. We compute attention scores by projecting the LSTM outputs through a learned linear layer, masking out padding positions, and taking a weighted sum to get a context vector.

2.5. Challenges and Solutions

We anticipated three major problems from the start: overfitting, computational constraints, and illegal move generation. The overfitting issue turned out to be worse than expected. Our initial model with just basic dropout achieved 70% training accuracy but only 54% validation accuracy, there was a massive 21 percentage point gap. The model was clearly memorizing specific game sequences rather than learning general chess principles. We tried several things to fix this. First, we increased dropout rates across the board. Then we added layer normalization after embeddings and attention, which helped stabilize training. Weight decay (L2 regularization) of 1e-4 on all parameters also helped. But the biggest improvement came from using more training data, we increased our training set from 40,000 to 75,000 examples. This gave the model more diverse positions to learn from, especially middle and endgame positions that were underrepresented when we only used short sequences.

Computational constraints were a constant battle. The legal move masking was particularly expensive and reconstructing the board state and querying legal moves for every training example was slow. Our lazy computation approach helped, but it still meant training took much longer than we'd hoped. We ended up filtering to sequences under 30 moves to keep training time manageable, though ideally we would train on the full dataset given the resources.

The illegal move problem was actually easier to solve than we expected. By masking the output logits before computing loss, we force the model to only consider legal moves during training. At inference time, we do the same masking before sampling from the probability distribution. This means the model literally cannot output an illegal move — the probability is set to zero (technically negative infinity before softmax) for any illegal move. This works much better than trying to train the model to learn legality as part of the prediction task.

Sequence Length	Embed Dim	Hidden Dim	Train Loss	Val Loss	Val Acc
10	128	256	1.0597	1.0805	0.6456
10	128	512	0.9465	1.0592	0.6540
10	256	512	0.9400	1.0519	0.6548
10	**256**	**1024**	**0.9438**	**1.0646**	**0.6566**
20	128	256	1.4552	1.4872	0.5532
20	128	512	1.2773	1.4449	0.5730
20	256	512	1.2564	1.4429	0.5708
20	256	1024	1.0808	1.4218	0.5899
30	128	512	1.6418	1.8119	0.4825

Table 1. ChessLSTM performance across sequence lengths, embedding sizes, and hidden dimensions. Trained for 20 epochs

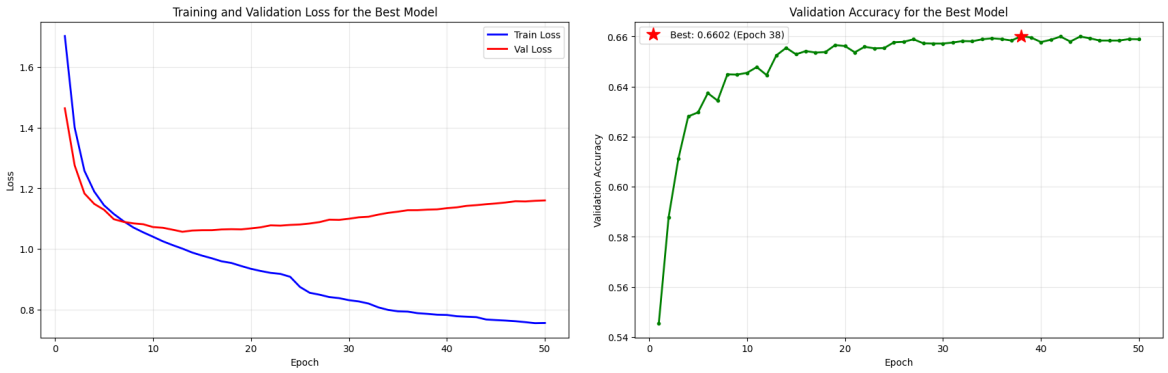

Figure 1. Best model loss and accuracy plots

3. Experiments and Results

Since our model is an LSTM translation model, we can just use binary cross-entropy (BCE) as our metric to evaluate which moves are best, since this method is widely used for classification models. As discussed before, our first naive models would occasionally output illegal moves. While it is probable that the model would learn to not predict illegal moves over time, we found that creating a mask to filter out illegal moves during training expedited our model's learning. This is done by adding a large negative number to the resulting logit for illegal moves, causing only legal moves to appear after softmax (and thus affecting the overall loss function).

3.1. Parameter Selection

We conducted a series of experiments by varying 3 key hyperparameters: maximum input sequence length, embedding dimension, and hidden dimension. Across all configurations, the trend we saw was that raising the embedding dimension and hidden dimension contributed to better training accuracy/loss and mostly better validation accuracy/loss (although overfitting was observed too). This is expected, as a richer embedding space and hidden space allow the model to capture more patterns in the SAN notation, and model more long-term complexities in the sequences.

However, this came at a significant drawback of training computation time, as raising any of these had a very large effect. Larger dimensionality also caused a wider gap between train loss and validation loss, which was even more exemplified with higher epoch count. This highlights the tradeoff between model capacity and model generalization.

3.2. Sequence Length

As seen in table 1, we evaluated models with maximums sequence length of 10, 2, and 30 moves. One issue with chess games is their arbitrary length, which heavily impact the ability of ChessLSTM to learn as the input sequence grows. The best performing model was only of maximum sequence length 10 compared to models trained on larger sequences.

Longer sequences did not perform well due to the computational difficulty, but we still noticed similar trends with learning across epochs. If more computational resources were available, we believe larger sequence versions of ChessLSTM would also be able to perform quite well.

3.3. Best Model

The strongest performing configuration was:

- **Sequence Length:** 10 SAN moves
- **Embedding Dimension:** 256
- **Hidden Dimension:** 1024

This model achieved a validation accuracy of **0.6566**, the highest among all experiments. The performance of this model shows the possibility for an LSTM-based network to perform move prediction from historic sequences.

4. Conclusions

In this work, we introduced a lightweight, human-style chess move predicted built on an LSTM with attention, trained on SAN-tokenized grandmaster chess games, and augmented with an illegal-move masking mechanism. Our work demonstrates that it is feasible to model chess as a sequential language that reflect human play, all while avoiding the heavy computational cost that current chess engines carry through their use of transformers and CNNs.

Still, our approach has its limitations. The complexity of the model is limited by the simple architecture of LSTMs and the size of the dataset, causing the model to struggle with long-term strategy. Due to the large vocabulary size that is required to encompass the notation of all chess moves, the last linear layer ends up being rather large regardless of any other hyperparameters. With more computation resources to create a more complex architecture (or just increasing the size of embeddings/hidden vectors), this sequence-based model can be pushed further.

Moving forward, there are many processing steps to be made outside of the actual model training to improve the model's ability to learn. Currently, the use of BCE only looks at the predicted probability of the correct move. However, a game like chess has many non-optimal moves that can still result in a favorable position. By implementing a loss function that uses Stockfish evaluation or a similar metric during training, the model will learn how to play optimally without exhaustive search. Even if the limit with simple LSTMs is reached, it would be worth exploring the use of transformers as a sequence prediction tool for chess. For instance, another paper [5] uses GPT-2 trained on a much larger dataset of sequence data to achieve interesting results.

Our LSTM-based chess model represents a first step towards lightweight, sequence-based, human-like chess AI.

References

[1] Leela chess zero. https://github.com/LeelaChessZero/lc0. Open-source deep learning engine. 1

[2] Rafał Drezewski and Grzegorz Wator. Chess as sequential data in a chess match outcome prediction using deep learning with various chessboard representations. https://home.agh.edu.pl/~drezew/papers/drezewski2021chess-as-sequential-data.pdf, 2021. LSTM-based approach. 1

[3] Niklas Fiekas. python-chess: A chess library for python. https://python-chess.readthedocs.io/, 2025. Version X.X.X. 2, 3

[4] PGNMentor. Pgnmentor chess game collection. https://www.pgnmentor.com/files.html. PGN files for Grandmaster games. 1, 2

[5] Karl Johan Schuster. The chess transformer: Mastering play using generative language models. *arXiv preprint arXiv:2008.04057*, 2020. 5

[6] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017. 1

Student Name	Contributed Aspects	Details
Pranam Kalla	Data Creation, Pipeline Framework, Paper	Found and parsed data into training/validation sets, created training pipeline, wrote introduction, experiments/results, conclusion
Calvin Ng	Model Development, Model Tuning, Paper	Researched architecture, implemented and trained model, wrote approach and conclusion

Table 2. Contributions of team members.