Solution Design in CNN-based Chess Move Prediction

Abdullah Abusamra
AI Dept
UNT
Denton, TX
11165478

Larry Johnson
AI Dept
UNT
Denton, TX
11643477

***Abstract*— In this white paper, we propose a supervised learning approach using Convolutional Neural Networks (CNNs) to predict the next best move in a chess game based on the current board state. The network learns from a dataset of historical games, incorporating player turn information, and outputs the predicted move in PGN (Portable Game Notation). Our method focuses on predicting legal and optimal moves without using reinforcement learning. We describe the mathematical foundations of the model, data encoding, and training process.**

I. INTRODUCTION

Chess is a highly strategic game in which players must evaluate complex board positions and choose optimal moves. Predicting the next move based on the current board state is a critical challenge in chess AI systems. In this paper, we develop a CNN to predict the next move given a board state, using a dataset of games played by various players.

In our approach to training a chess move prediction model, we use a neural network architecture designed to handle the unique dynamics of chess, including the alternating turns between players. The model takes as input a tensor representation of the chessboard, with an additional channel that encodes the player's turn. This step ensures that the model can accurately distinguish between moves appropriate for White and Black, a critical feature often overlooked in traditional chess models.

II. PROBLEM FORMULATION

Let the chessboard be represented as an 8x8 grid. Each square can be occupied by one of 12 different piece types: 6 for White and 6 for Black. We define a position on the board as a matrix $B \in R^{8\times 8}$ where each channel in the third dimension represents the presence or absence of a particular piece type on the board.

The objective of the model is to learn a function $f_\theta(B)$ that maps the current board state B to the next move M. Let M be encoded as an index in a vector of length 4612, representing all possible moves from one square to another:

$$M \in \{0, 1, \dots, 4611\} \qquad (1)$$

Thus, our goal is to minimize the loss function between the predicted move:

$\widehat{M} = f_\theta(B)$ and the actual move M∗ (2)

The project's primary objective is built using a move predictive deep learning CNN. The CNN will consist of the following:

- Six convolutional layers and two fully connected layers, employing ReLU activations and batch normalization.
- Optimization with cross-entropy loss and Adam optimizer.

III. INPUT REPRESENTATION

A. Board Encoding

We represent the current chessboard B as an 8x8x14 tensor, where:

(3)

B(i, j, k) = {1, if square (i, j) contains the kth piece type

0, otherwise

For instance, k=0 might represent a White pawn, k=6 might represent a Black pawn, and so on.

B. Turn Encoding

To distinguish between White and Black's moves, we add a 13th channel to the board tensor:

(4)

T(i, j) = {1, if it is White's turn to move
0, if it is Black's turn to move

C. *Absence or presence of a piece type*

P(i, j) = {1, if the piece type is present
0, if the piece type is not preset

This adds 12 additional attributes to the tensor.

The input tensor now becomes:

$X = [B, T, P] \in R^{8x8x14}$

This ensures that the network knows which player is to move and can adjust its predictions accordingly.

IV. MOVE REPRESENTATION

The output move M is represented as an index in a vector of length 4612, where each index represents a possible move:

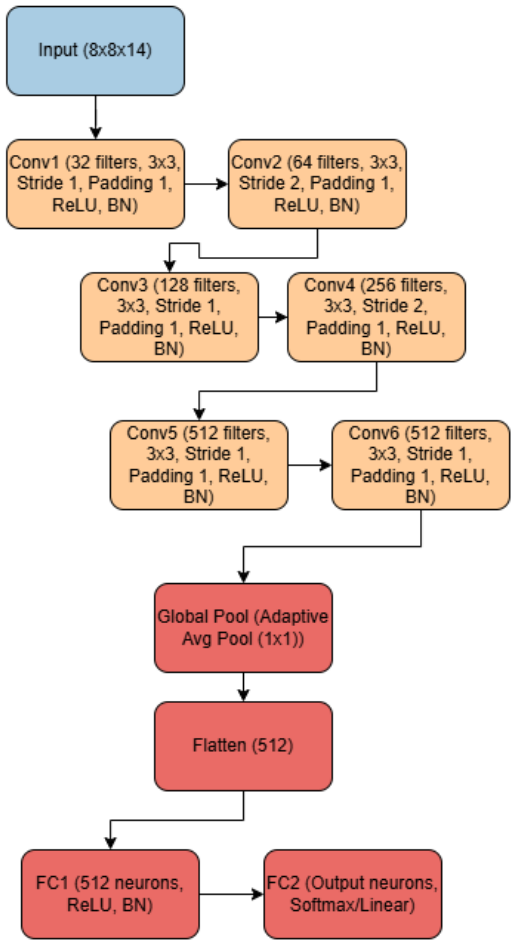
(5)

$M = 64 * F + T$

where F is the index of the "from" square, and T is the index of the "to" square. Thus, the possible moves are all combinations of moving a piece from one square to another.

V. MODEL ARCHITECTURE

Our model is a CNN that processes the input tensor X. The CNN consists of three convolutional layers followed by two fully connected layers. The architecture is mathematically described as follows:



A. *Convolutional Layers*

The first convolutional layer applies 32 filters of size 3×3 with stride 1 and padding 1, as well as batch normalization producing an output tensor:

(6)

$C_1(X) = \text{ReLU}(W_1 * X + b_1)$

Where:

(7)

$W_1 \in R^{32x14x3x3}$

The second convolutional layer applies 64 filters of size:

(8)

$C_2(C_1) = \text{ReLU}(W_2 * C_1 + b_2)$

Where:

(9)

$W_2 \in R^{64x16x3x3}$

The third convolutional layer applies 128 filters of size:

$C_3(C_2) = \text{ReLU}(W_3 * C_2 + b_3)$

Where:

$$W_3 \in R^{128x32x3x3} \tag{10}$$

The fourth convolutional layer applies 256 filters of size:

$$C_4(C_3) = ReLU\,(W_4 * C_3 + b_4)$$

Where:

$$W_4 \in R^{256x32x3x3} \tag{11}$$

The fifth convolutional layer applies 512 filters of size:

$$C_5(C_4) = ReLU\,(W_5 * C_4 + b_5)$$

Where:

$$W_5 \in R^{512x32x3x3}$$

This sixth convolutional layer applies 512 filters of size:

$$C_6(C_5) = ReLU\,(W_6 * C_5 + b_6)$$

Where:

$$W_6 \in R^{512x32x3x3} \tag{12}$$

B. Flattening and Fully Connected Layers

The output from the second convolutional layer is flattened into a vector:

$$F \in R^{512x8x8} \tag{13}$$

and passed through two fully connected layers:

$$F_1 = \text{ReLU}(W_{\{fc1}F + b_{fc1}$$

Where:

$$F_2 = \text{ReLU}\; W_{\{fc2}F_1 + b_{\{fc2}$$

Where:

$$W_7 \in R^{4612x512} \tag{14}$$

C. Output Layer

The output layer produces logits representing the probability of each move:

$$\widehat{M} = Softmax(F_2) \tag{15}$$

VI. Training

A. Loss Function

The loss function is the cross-entropy loss between the predicted move $\widehat{M}$ and the actual move M∗:

$$\frac{\partial L}{\partial \theta} = \text{L}(\theta) = -\sum_{i=1}^{4096} M_i^* log(\widehat{M}_i) \tag{16}$$

where M* is the one-hot encoded actual move and $\widehat{M}$ is the predicted probability distribution over the moves.

B. Optimization

The model is trained using stochastic gradient descent (SGD) with Adam optimizer. The gradient of the loss with respect to the model parameters θ is computed and the weights are updated as follows:

$$\theta \leftarrow \theta - \eta\, \frac{\partial L}{\partial \theta} \tag{17}$$

where η is the learning rate.

VII. Dataset and Training Procedure

We use a dataset of chess games in PGN format from the lichess website. The Lichess Database provides monthly dumps of games in ZIP format. You can download them directly from their database page. Each file is compressed in ZIP format and contains millions of games in PGN format. However, due to limited compute and storage to minimize costs of the research we will use a much smaller subset of the data (ie. thousands of games).

Each board state is paired with the correct move played in that game. The board state is essentially a 8x8 matrix designed by the PyChess. PyChess is a free, open-source chess game application written in Python, designed for users to play chess against computer opponents or other players. It provides a graphical user interface (GUI) for playing chess and includes a built-in chess engine that supports different levels of AI difficulty.

VIII. TRAINING

The training loop processes batches of board states and corresponding move labels. During each epoch, the model performs a forward pass to predict the next move, computes the loss between the predicted move and the actual move, and adjusts the model parameters using backpropagation.

A. Differentiating Player Moves

A key innovation in our method is the explicit encoding of player turns. The model needs to understand whether it is predicting a move for White$W_1 \in R^{64x3x3x14}$ or Black, as each player's valid moves differ. Two primary methods are employed for this:

1- Turn Indicator Channel: An additional channel is added to the input tensor that is filled with 1s if it is White's turn and 0s for Black. This ensures the model has direct access to the player information.
2- Piece Encoding Based on Turn: Alternatively, pieces can be encoded differently based on whose turn it is, allowing the model to infer the current player from the board configuration itself.

IX. RELATED WORK AND EVALUATION METRICS

A. Preprocessing Data & Model Setup

The model, called ChessMoveNet, uses two convolutional layers to process the 13-channel chessboard input (representing pieces and turn information), followed by two fully connected layers. The final output layer produces logits for 4612 possible moves on the chessboard. The activation function used between layers is ReLU, and the convolutional layers capture spatial relationships on the board before flattening the data for the fully connected layers.

B. Loss Function and Optimizer

In the process of training a neural network model for chess move prediction. It uses Cross-Entropy Loss, which is ideal for multi-class classification problems, and the Adam optimizer with a learning rate of 0.001. The training loop iterates through batches of chessboard tensors and their corresponding move labels. During each epoch, it computes the model's predicted moves, calculates the loss compared to the actual moves, and updates the model's parameters through backpropagation. The model's performance is tracked by printing the loss after each epoch, providing a clear metric of progress. Keep going—you're doing great with this setup!

C. Predicting Moves for Both Players

In chess, players alternate turns—White moves first, then Black, and so on. To accurately predict the next move, your model must know which player's turn it is. Without this information, the model wouldn't be able to distinguish between moves appropriate for White versus Black.

Including Player Turn Information in the Model
To ensure the model knows whose turn it is, you can incorporate this information into the input features. Here are ways to achieve this:

Adding a Turn Indicator Channel
Modify the Input Tensor: Expand your input tensor to include an additional channel that represents whose turn it is. If it's White's turn, the entire channel is filled with 1s. If it's Black's turn, the channel is filled with 0s.

D. Evaluation Metrics

For evaluating chess move prediction models, a combination of metrics can provide a comprehensive assessment. Top-k Accuracy is a valuable metric, measuring whether the correct move is within the top k predictions, which is crucial given the vast number of possible moves in chess. Precision and Recall can be used to evaluate the model's performance in predicting specific move types, such as captures or checks, with precision focusing on the accuracy of positive predictions and recall on finding all relevant moves. Additionally, Move Prediction Accuracy measures the overall percentage of correctly predicted moves. Another important metric is Move Matching with Expert Games, which compares the model's moves with those played by grandmasters or in expert games, assessing how well the model aligns with high-level human strategies. These metrics together provide a robust evaluation of both the accuracy and strategic quality of the model's predictions.

X. EXPERIMENTAL RESULTS

As the project is nearly two-thirds complete the initial milestones of the project were reached. In particular, the extract, transform, load (ETL) and machine learning (ML) initial iterations have built a solid foundation for the rest of the project. As the building blocks of the project are in place the Deep Learning (DL) ML algorithm can train on random samples of any existing chess games in PGN format, perform training of the chess data, and lastly actually play games of chess against other computers and humans. This milestone can now enable the team iterate faster on future architectural software changes, configurations, hyper-parameters, and even potential new data transformations without significant software changes.

Given that the initial priority was software architecture training and accuracy were secondary goals. As a result, the training and subsequently accuracy were anticipated to be low. However, we were hoping to achieve much higher accuracy than the current model performs at in the latest rounds of testing and validation (at or near 0%) with losses at around 3%. As a result, we need to analyze the problem from a domain perspective and perhaps do some more research for projects that have a similar set up to achieve some level of higher accuracy. However, even the most sophisticated chess algorithms, training on millions of games, do not achieve accuracy in the 90% range consistently without significant financial investment. A more realistic goal would be trying to achieve something similar to a team from Stanford University[1] given our limited budget of computing and storage resources. Some of these more realistic studies with similar constraints have achieved a training accuracy in the range of 20-30% on about 20,000 games with about 250,000 moves. The random sample dataset for our initial research has approximately 25,000 games with about 200,000 moves.

Towards the end of the project, we made some additional adjustments to improve upon model performance. For example, improving the feature representation of the chessboard can significantly enhance the neural network's ability to predict moves. One way we achieved this was by modifying the values assigned to player pieces in the board tensor. Instead of static values based solely on piece type and color, these values could be adjusted to reflect strategic importance, such as positional influence or mobility. For instance, pawns in central squares or during the endgame could be assigned higher importance. By encoding these values dynamically, the network can better capture the strategic nuances of the board, allowing it to make more informed predictions.

Additionally, including 12 extra channels—one for each unique piece type and color—introduces a more granular, one-hot encoding representation. This method isolates each piece type into its respective channel, making the board tensor less ambiguous and easier for the network to interpret. For example, white rooks would occupy a unique channel distinct from black rooks or other pieces. This separation decouples overlapping information, enabling the network to learn more discriminative features. Stacking these additional channels with other board features, such as the player's turn, creates a richer multi-dimensional tensor that provides a more comprehensive view of the game state.

Furthermore, replacing the current player turn representation, which uses -1 for black and 1 for white across all squares, with a single one-hot encoded channel can further streamline the input tensor. This representation would mark squares with 1 for the current player's pieces and 0 otherwise, eliminating redundancy and reducing noise. Such a compact and spatially specific encoding aligns with best practices for neural network inputs, improving both the efficiency and effectiveness of feature learning.

Collectively, these refinements—enhancing piece values, adding one-hot encoded channels for piece types, and simplifying player turn encoding—create a richer and more interpretable representation of the chessboard. These changes are expected to improve the model's ability to extract meaningful patterns, resulting in more accurate move predictions. Future experimentation, including ablation studies and performance evaluations, will help determine the impact of each modification and optimize the overall architecture for chess move prediction.

At the end of the project, we were able to significantly increase the accuracy from both the training and validation test sets less than 1% to approximately 30% increase. Furthermore, the dataset sampled totaled approximately 6.2K games and 500K moves.

The purpose of this report is to summarize the performance of a convolutional neural network developed to predict chess moves from board positions. The evaluation metrics include Top-k Accuracy, Precision, Recall, and F1-Score, along with a detailed record of training and validation performance over 10 epochs.

The model achieved a final Top-k Accuracy of 6.8374, indicating that the correct move was ranked among the top predictions in only a small fraction of cases. Precision, Recall, and F1-Score were also relatively low, at 0.3760, 0.3580, and 0.3425, respectively. These results highlight challenges in the model's ability to accurately and consistently identify the correct moves.

Epoch	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
1	0.036	0.0727	0.0322	0.1043
2	0.0291	0.1362	0.0277	0.1496
3	0.0258	0.1694	0.026	0.1703
4	0.0239	0.1913	0.0248	0.1861
5	0.0225	0.2075	0.0242	0.1907
6	0.0213	0.2251	0.0239	0.197
7	0.0203	0.2424	0.0237	0.1993
8	0.0193	0.2607	0.0238	0.2044
9	0.0184	0.2803	0.0239	0.205
10	0.0175	0.3005	0.0241	0.2074

As the Table above demonstrates, during training, the loss steadily decreased across epochs, with the final training loss

[1] https://cs231n.stanford.edu/reports/2015/pdfs/ConvChess.pdf

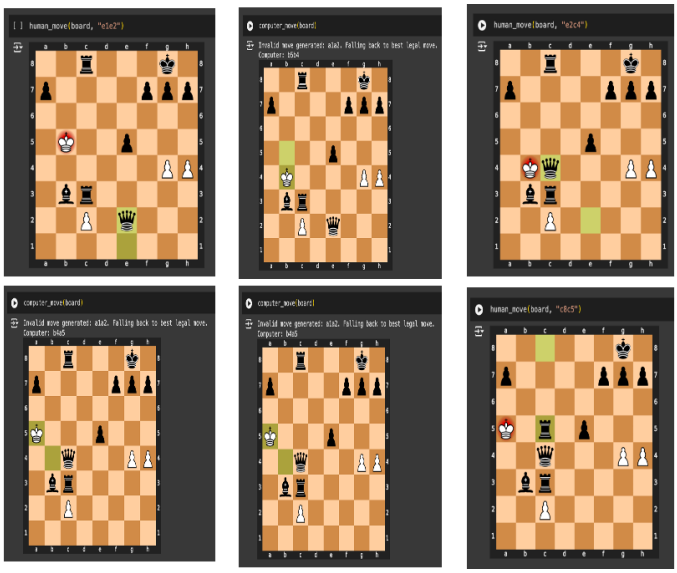
reaching 0.0175. Training accuracy improved incrementally, reaching 30.05% by the 10th epoch. Validation loss remained relatively stable after the 4th epoch, ending at 0.0241, while validation accuracy improved slightly, reaching 20.74% at the final epoch. However, the noticeable gap between training accuracy (30.05%) and validation accuracy (20.74%) suggests some degree of overfitting.

The model's limited performance highlights areas for improvement. Data augmentation techniques, such as flipping or rotating board positions, could increase the diversity of the training data and improve generalization. Regularization methods, including dropout, weight decay, or early stopping, may help address overfitting. Additionally, experimenting with deeper architectures or integrating attention mechanisms could enhance the model's ability to extract features from the board state. Improving Top-k Accuracy through strategies that better rank predictions and analyzing potential imbalances in the dataset could further enhance performance.

In conclusion, while the model demonstrates steady progress during training, its overall performance on validation data remains limited. Addressing overfitting and improving generalization will be key priorities for future iterations to increase its effectiveness in predicting chess moves accurately.

Software architecture was our initial focus on the project, so that we could ultimately build a chess application. As a result, we met this milestone around the middle of the project, so that we could see the Machine Learning (ML) algorithm actually play other humans and computers in chess to evaluate its real-world applications. Below are some of the early moves made by the ML algorithm, which played white. As you can see it initially made good moves.

However, towards the game we noticed that it did not play as well (ie. figure below), which perhaps negatively contributed to its accuracy. During testing the algorithm by playing chess we found a software bug that had some systematic label inaccuracies that has invalid chess moves in them. However, through additional research the software can continue to be improved by fixing known and future software defects.



XI. NEXT STEPS

A. *Reinforcment Layer*

To enhance the performance of the chess next move prediction model, a reinforcement layer can be incorporated to guide the learning process by leveraging a reward-based feedback mechanism. This layer could be designed to evaluate the predicted move against a game engine's analysis, assigning a reward for moves that align with optimal or near-optimal strategies and penalties for suboptimal choices. By integrating reinforcement learning principles, the model could iteratively refine its predictions to better mimic expert-level decision-making, effectively capturing the dynamic, sequential nature of chess gameplay. This approach aligns with the methodologies discussed in Sutton and Barto's foundational work on reinforcement learning, emphasizing how reward-driven optimization can improve decision-based tasks (Sutton & Barto, 2018).

B. *Gradient-based Visualization Techniques:*

To improve interpretability and visualization of a CNN applied to an 8x8 chessboard, future steps should focus on mapping the impact of all network operations back to the input layer. This involves computing the sensitivity (gradients) of hidden features with respect to each board index, highlighting which portions of the input the network responds to most strongly. This can be achieved using techniques like saliency maps, guided backpropagation, and deconvolutional networks

(deconvnets). For unsupervised learning, an encoder-decoder framework can be implemented, where the encoder and decoder weights are learned jointly to minimize reconstruction error, requiring training from scratch. Additionally, backpropagation-based gradient computations can be extended to determine the sensitivity of features concerning input indices rather than weights, emphasizing transposed convolutional operations to visualize gradients effectively. These improvements would provide deeper insights into feature responses and decision-making processes, enhancing network interpretability.

REFERENCES

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018.

[2] M. Lapan, *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*, 2nd ed. Birmingham, UK: Packt Publishing, 2020, pp. 325–330.

[3] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," 2013, arXiv:1312.6114. [Online]. Available: https://arxiv.org/abs/1312.6114

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," 2013, arXiv:1312.5602. [Online]. Available: https://arxiv.org/abs/1312.5602

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS 2012)*, Lake Tahoe, NV, USA, Dec. 2012, pp. 1097–1105.

[6] H. Sadjadpour, "Predicting chess moves with convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2020, pp. 3205–3212.

[7] F. Tian, and Y. Zhu, "Better computer Go player with neural network and long-term prediction," in *Proceedings of the 19th International Conference on Neural Information Processing (ICONIP)*, Doha, Qatar, Nov. 2012, pp. 1050–1060.

[8] M. Kampmann, P. Schramowski, and K. Kersting, "Playing Atari with deep reinforcement learning," in *Proceedings of the 30th Conference on Neural Information Processing Systems (NeurIPS)*, Barcelona, Spain, 2016, pp. 3860–3870.

[9] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, D. Hassabis, and D. Silver, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, Dec. 2020.

[10] Y. Zhang, R. Sutton, and Z. Zhou, "Reinforcement learning for chess move prediction," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, Honolulu, HI, USA, 2019, pp. 5481–5488.

[11] T. J. Watson, "Improving chess move prediction with a convolutional neural network architecture," M.S. thesis, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2020. [Online].