CMPM 118 Final Report – Efficient Chess-AI

Max Ratcliff
Baskin School of Engineering
UC Santa Cruz
mwratcli@ucsc.edu

Avni Gandhi
Baskin School of Engineering
UC Santa Cruz
avgandhi@ucsc.edu

Mann Malviya
Baskin School of Engineering
UC Santa Cruz
mmalviya@ucsc.edu

Abby Kaur
Baskin School of Engineering
UC Santa Cruz
akaur79@ucsc.edu

Abstract

Traditional chess engines use classical evaluation methods which require chess expertise. We built a hybrid method combining minimax search with a learned neural network evaluator and policy generator. Through tournaments against various Stockfish levels, we estimate an ELO of 1647 which outperforms basic minimax(1420 ELO) and random play(400 ELO). This demonstrates how neural networks can replace traditional evaluation heuristics and still maintain computational efficiency.

1 Introduction

Chess engines have steadily evolved from fast, rule-driven searchers to deeply trained neural systems. Traditional engines like Stockfish succeed by exploring vast portions of the game tree and scoring positions with carefully engineered heuristics, while modern neural engines such as AlphaZero learn strategic judgment through massive self-play and guided search. Both approaches are powerful, but they land at opposite ends of the efficiency spectrum: one relies on handcrafted logic and high node throughput, the other on large-scale training and expensive inference. Our work aims to find a practical middle ground. Instead of building a full reinforcement-learning system, we train a lightweight ResNet-based evaluator and policy head on the Lichess Elite database and pair it with an alpha-beta search, allowing the network to provide intuition while the search algorithm handles depth and precision.

In testing, this hybrid design showed that the learned evaluation can meaningfully strengthen the classical search without demanding the heavy computation that defines modern RL engines. Across tournament play, the neural-guided version reached an estimated ELO of 1647, surpassing basic minimax, which is a 1420 ELO and far exceeding random play at 400 ELO. The results suggest that a trained evaluation network can replace manually tuned heuristics while preserving the responsiveness and efficiency expected from traditional engines.

Preprint. Under review.

2 Methodology

2.1 Data and Representation

Our training data was sourced from the Lichess Elite Database, which is a filtered set of games on Lichess of players with 2500+ ELO. These positions provide a reliable distribution of expert decisions, making them well-suited for learning evaluation signals and move tendencies without requiring reinforcement self-play

To represent the state of the chess board understandable by the neural network, we encode each position as a 20 x 8 x 8 tensor. The first twelve channels record the spatial locations of each piece type for both sides, while the remaining channels capture contextual information essential for evaluation: side to move, castling availability, en passant targets, the half-move clock, and a normalized game-phase indicator. When Black is to move, the board is flipped internally so the model always evaluates from the current player's perspective, simplifying the learning problem.

In the next iteration, we extended this representation to incorporate move history. By stacking the previous four board states using a 5 x 20 x 8 x 8 tensor, we provide the model with temporal context reflecting how the position evolved rather than only its current snapshot. This design enables the network to capture momentum, impending threats, and strategic buildup, which are all patterns that can't be inferred from a single board state alone. The history embedding offers a lightweight alternative to full sequence modeling and is motivated by practical constraints: storing a short window of states is far more efficient than full replay buffers, yet still grants the network access to short-term tactical and positional trends.

2.2 Model Architecture

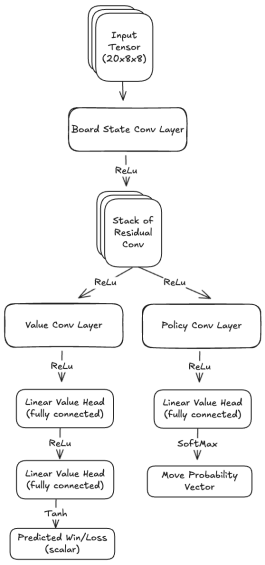

Figure 1: Overview Of the Model Architecture

The core of our evaluation system is a lightweight ResNet-based neural network designed to act as a positional judge rather than a full chess player. This is the key difference from larger RL engines like AlphaZero, which use deep networks both to evaluate and to drive Monte Carlo rollouts. Our design is simpler and more focused: the network produces a single score for a position, and the search algorithm decides what to do with it.

The model receives the chessboard as a 20 x 8 x 8 tensor, which includes piece placement and essential game-state information such as turn, castling rights, and game phase. The first convolutional layer expands this information into a higher-level feature representation while keeping the familiar 8 x 8 layout so the model can still "see" files, ranks, diagonals, and king structure clearly. The Residual

Blocks that follow allow the network to learn more complex patterns like piece coordination, pressure buildup, or stable outposts, without becoming unstable or losing information as depth increases.

Instead of outputting a list of candidate moves, the network reduces everything it has learned about the position into a single value between -1 and 1. This is done through a final 1 x 1 convolution and two fully connected layers, combined with a tanh activation that constrains the output to a stable range. A score close to 1 means the current side is winning, -1 means losing, and 0 means roughly equal. This score becomes the evaluation signal used during search, replacing handcrafted heuristics and material tables with a learned, data-driven assessment.

2.3 Training

After a few unsuccessful training attempts on an m2 mac with mps, we switched to taking advantage of the free colab pro education plan to run our training script on an A100, this massively reduced our training time from over 10 hours per epoch, to just 3 hours per epoch. even then we only had the compute credits to train for 5 epochs, meaning further training could likely improve our model even further. In addition to this we decided to limit our hidden channel size, and number of residual blocks to just 64, and 3 in order to save compute, but our code leaves these as hyper parameters meaning that they can be quickly adjusted before each training session. With more resources it would be interesting to explore how our model size impacts the quality of its play.

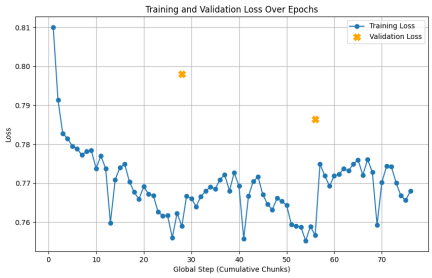

Figure 2: Loss plot for 1st Gen model

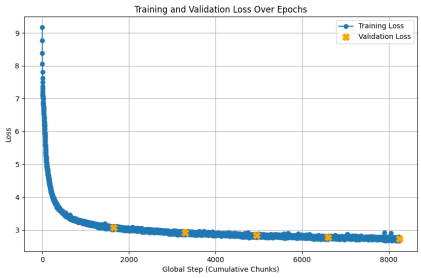

Figure 3: Loss Plot for 2nd Gen model

The first generation model featured only the value head, and a step LR scheduler. This model was very susceptible to saturated gradients, and slightly overfit to the training positions. After running tests and getting feedback on our architecture we implemented cosine LR scheduling, and gradient clipping, and pushed towards the stretch goal of a policy head after realizing that being able to predict the next move is critical to the speed of the minimax engine, not just the quality of the win predictions. After making these small changes, our loss curve instantly looks better, eliminating our overfitting issue, and hugely reducing the gradient saturation issue.

2.4 Search Algorithm

The core of our engine is a Minimax algorithm enhanced with Alpha-Beta pruning. This is the main difference between our architecture and AlphaZero, which uses Monte Carlo Tree Search (MCTS). We chose Minimax primarily because it is significantly easier to implement and debug than MCTS, due to the fact that minimax is deterministic, which allowed us to focus our efforts on the neural network integration.

The algorithm works by generating a list of legal moves and simulating the game forward, assuming that the opponent will always play the best possible response. To make this efficient, we use Alpha-Beta pruning to "prune" (skip) branches of the game tree that are clearly worse than moves we have already found.

The Policy Head of our neural network is essential to this efficiency. In a basic engine, moves might be sorted simply by checking if they are captures. In our hybrid engine, we use the Policy Head to sort the moves based on what the network predicts a strong human would play. By exploring

these high-probability moves first, the algorithm finds the best paths faster and triggers pruning much earlier, avoiding wasting compute time on bad moves.

3 Results and Discussion

3.1 Performance metrics

Search Speed For our search speed, we achieved 22,500 nodes per second for the basic minimax engine and 15,100 nodes per second for the hybrid engine. Although these speeds are substantially slower than production engines(3M+ nodes per second), the performance is adequate for real-time play with our target depth of 6 ply. This depth was chosen based on computational feasibility with Python and real-time response requirements(2-3 seconds per move).

Depth achieved We did 6 plys for testing because both algorithms were fairly efficient due to various techniques we used to speed the time up, such as iterative deepening, alpha-beta pruning which cuts 75% of nodes, and a transposition table to avoid re-searching positions.

Key Findings Our ELO rating estimate was 1647 for the hybrid engine 1420 for the basic minimax engine and 400 for the random engine. These results are for a depth of 6 and with 10 games played against each with color alternation to prevent first-move advantage of the 4 different Stockfish levels(1,2,3, and 4). These estimates are derived from the logistic ELO expectation equation

$$E_A = \frac{1}{1 + 10^{\frac{R_B - R_A}{400}}},$$

which calculates ELO based on tournament results against opponents of known strength. The hybrid engine demonstrates a +227 ELO improvement over basic minimax evaluation which validates our hypothesis that neural networks can effectively replace traditional evaluation algorithms. The 1647 ELO places our engine at an intermediate playing level. The random engine baseline(400 ELO) confirms that both agents have significant knowledge.

Limitations and Context Several factors influence these ratings. Having only 40 games in total for ELO calculation introduces statistical uncertainty. With 40 games and a 45% win rate for the basic minimax engine, the 95% confidence interval is ± 27 ELO, calculated using the binomial standard error formula. As the difference exceeds the uncertainty margin, we can say that the improvement is statistically significant. Another effecting factor is that we are only doing 5-6 plys which is a computational constraint as deeper search would likely improve the ELO rating, but this would require a GPU-accelerated evaluation. Stockfish ELO calibration estimates also introduce measurement uncertainty of $\pm$50-100 ELO depending on the hardware. Despite these limitations, the relative comparison between the engines remains valid and demonstrates the effectiveness of the hybrid chess engine.

3.2 Discussion & Analysis

Comparison to AlphaZero AlphaZero combines reinforcement learning via self-play with Monte Carlo Tree Search(MCTS). The system learns evaluation and move selection policies through millions of self-play games and develops strategies that differ from human intuition. Our approach uses supervised learning on human expert games combined with minimax search with alpha-beta pruning.

Trade-offs:

- *Computational Cost*: AlphaZero requires more compute because it generates its own training data, while our models trains once on the Lichess dataset making it lightweight and more feasible for our team.
- *Evaluation scope*: AlphaZero uses both a policy head and a value head which guides MCTS directly. Our model provides a value estimate and a lightweight policy head used only for move ordering, so the minimax search, instead of the neural network, determines move selection.

- *Performance ceiling*: AlphaZero's self-play is able to discover computer-based strategies, while our supervised approach is bounded by the quality of human training data. However, our alpha-beta search can still find tactical combinations that are not in the training set.

4 Conclusion

In this project, we successfully built an end-to-end machine learning chess engine that combines a neural network evaluator with a classical search framework. Instead of relying solely on hand-crafted heuristics, our model learned how to judge positions directly from expert games, and this gave the engine a noticeable boost in playing strength over basic minimax.

Although most of our final results relied on the value head, our model iterations showed that the policy head offered clear benefits during development, especially in shaping move ordering and improving training stability.

In the future, the most promising next steps to improve our model are to explore MCTS to reduce the horizon effect and experiment with model quantization to speed up inference. Together, these improvements could make the engine both deeper tactically and faster in real-time play, while still keeping the overall design lightweight and understandable.

References

[1] Wikipedia contributors. "Board representation (computer chess)." *Wikipedia, The Free Encyclopedia*, Oct. 12, 2025. [Online]. Available: `https://en.wikipedia.org/wiki/Board_representation_(computer_chess)`

[2] N. Fiekas. "python-chess: a chess library for Python." *python-chess.readthedocs.io*, 2024. [Online]. Available: `https://python-chess.readthedocs.io/en/latest/`

[3] GeeksforGeeks. "Min-Max Algorithm in Artificial Intelligence." *GeeksforGeeks*, Aug. 22, 2024. [Online]. Available: `https://www.geeksforgeeks.org/min-max-algorithm-in-artificial-intelligence/`

[4] Kaggle. "FIDE / Google - Efficient Chess AI Challenge." *Kaggle*, 2025. [Online]. Available: `https://www.kaggle.com/competitions/fide-google-efficiency-chess-ai-challenge/overview`

[5] Lichess. "Lichess Elite Database." *database.lichess.org*, 2025. [Online]. Available: `https://database.lichess.org/`

[6] Chess Programming Wiki. "Bitboards." *The Chess Programming Wiki*. [Online]. Available: `https://www.chessprogramming.org/Bitboards`

[7] Wikipedia contributors. "Bitboard." *Wikipedia, The Free Encyclopedia*, Sep. 28, 2025. [Online]. Available: `https://en.wikipedia.org/wiki/Bitboard`

[8] Wikipedia contributors. "Convolutional neural network." *Wikipedia, The Free Encyclopedia*, Oct. 15, 2025. [Online]. Available: `https://en.wikipedia.org/wiki/Convolutional_neural_network`

[9] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition." *arXiv preprint arXiv:1512.03385*, 2015.

[10] B. Anumukonda. "Building a Chess Engine with Reinforcement Learning." *bhaswanth-a.github.io*, May 28, 2021. [Online]. Available: `https://bhaswanth-a.github.io/posts/chess-engine-rl/`

[11] S. Gunasekaran. "Convolutional Neural Networks." *CMPM118 Course Materials*, 2024. [Online]. Available: `https://github.com/SkyeGunasekaran/CMPM118-NCG/blob/main/edu-material/4_cnn.md`

[12] D. Silver, T. Hubert, J. Schrittwieser, et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play." *Science*, vol. 362, no. 6419, pp. 1140-1144, Dec. 2018.

[13] G. van der Hoorn, A. Pasquale, and the LeelaChessZero team. "Leela Chess Zero." *lczero.org*, 2018. [Online]. Available: `https://lczero.org/`

[14] C. B. Browne, E. Powley, D. Whitehouse, et al. "A survey of monte carlo tree search methods." *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1-43, March 2012.