# A NEURO-EVOLUTIONARY APPROACH TO CHESS ENGINE DEVELOPMENT VIA REINFORCEMENT LEARNING



Aryan Koli

Dept of Electrical and Electronics Engineering of MIT WPU, Kothrud, Pune, Maharashtra, India

[aryankoli03@gmail.com](mailto:aryankoli03@gmail.com)

Jayesh Jhawar

Dept of Electrical and Electronics Engineering of MIT WPU, Kothrud, Pune, Maharashtra, India

[jayesh.pradip@gmail.com](mailto:jayesh.pradip@gmail.com) Manisha Kowdiki

Dept of Electrical and Electronics Engineering of MIT WPU, Kothrud, Pune, Maharashtra, India

[manisha.kowdiki@mitwpu.edu.in](mailto:manisha.kowdiki@mitwpu.edu.in)

Ranveer Singh

Dept of Electrical and Electronics Engineering of MIT WPU, Kothrud, Pune, Maharashtra, India

[ranveeer13.3@gmail.com](mailto:ranveeer13.3@gmail.com)

# Abstract

The evolution of chess engines has revolutionized the game by melding artificial intelligence with strategic thinking in the style of human beings. Initially, the engines were based on basic AI principles such as the Minimax algorithm and Alpha-Beta Pruning, both of which relied on static evaluation criteria and were constrained by the processing capabilities available at the time. Popular engines like Stockfish, Houdini, Komodo, and Leela Chess Zero used these classical models to good effect, dominating specifically in exact tactical computation. While strong, these systems were not able to improve or learn from their initial programming. A key breakthrough occurred when machine learning and reinforcement learning were applied, adding the feature of engines being able to improve and adapt on their own. AlphaZero was unique in this sense; it learned to play chess solely by self-play, not using existing human games or information. Its triumph over traditional engines was a breakthrough, demonstrating the potential for learning-based AI to transform how chess is played at the top level.

# Introduction

### 1.1 Early Beginnings to the AI Revolution

Chess engines have existed since the 1950s, fuelled by the initial advancements in artificial intelligence and concepts from visionaries such as Alan Turing [1]. Before actual software was conceived, devices such as "The Turk" exemplified how intrigued people were about the possibility of mechanical chess players. However, the first engines were hampered by the rudimentary hardware and programming capabilities of the time.

Things took a big shift in 1988 with IBM's Deep Thought, which was able to search around 100,000 positions per second—a huge improvement then. During the 1990s, algorithms such as minimax enabled engines to make more intelligent decisions, and the historic 1997 match when IBM's Deep Blue defeated Garry Kasparov set AI chess in the limelight [2][3].

By the 2000s, the engines were more efficient through alpha-beta pruning, and they were invincible in specific endgame situations through tablebases. The 2010s arrived, in which machine learning dominated every aspect—Stockfish perfected how it evaluated positions [4], whereas AlphaZero and LCZero employed self-play and reinforcement learning, fueled by high-performance GPUs [5].

All these developments have had a profound effect—not only on the way people train and prepare for games, but also on the popularity of internet chess and the ongoing controversies surrounding ethics and fairness in competitive play [6].

### 1.2 About chess engine-

Chess has been a traditional standard for the assessment of artificial intelligence, mostly due to its inherent strategic depth and structured complexity [7]. In AI studies, board games such as Chess and Checkers are traditionally represented as single-agent problem spaces. These environments give a controlled space in which AI models can be trained to achieve, and occasionally surpass, human expert capabilities [8]. What makes chess particularly interesting is the challenge it presents to logical reasoning, long-term strategy, and rule-bound decision-making. Chess engines software programs designed to simulate human-level play are great examples of how computational techniques can be applied to build efficient, smart decision-making systems.

This review discusses the evolution of chess engines, from early rule-based systems to current engines that are supplemented by artificial intelligence [13]. A chess engine is basically a computer program created to examine positions on a chessboard and recommend best moves through a mix of algorithmic reasoning and processing power. Such engines use a range of search methods, evaluation functions, and AI-based processes to provide top-level performance. Conventional engines such as Stockfish mostly depend on brute-force search and meticulously hand-tuned heuristics to evaluate various board situations [9]. On the other hand, more recent engines use machine learning and neural networks so that they can learn a more adaptive and flexible sense of the game. Perhaps the biggest breakthrough in this area was AlphaZero [10], created by DeepMind, which learned the game purely by self-play learning from strategy without using human-played games as input. AlphaZero-inspired, Leela Chess Zero [11] took the same path by integrating deep learning methods with Monte Carlo Tree Search (MCTS) to produce extremely intuitive and strategic play [12].

### 1.3 Evolution of chess engine -

The evolution of chess engines from rudimentary, rule-based systems to sophisticated AI-powered tools is a significant change in how clever play is tackled [13]. This article tracks that evolution, exploring how those engines work, how AI has improved their operation, and what the essential distinctions are between AI-based and more traditional approaches. By contrasting historical models with those based on machine learning, we can see each's strengths and weaknesses, limitations, and working applications. Major performance milestones and new directions in the ongoing improvement of intelligent game systems are presented in the review.

**II. AI & ML in Chess Engine**

*Search Algorithms*

A chess engine's primary ability is derived from its search algorithm, as it allows the engine to search ahead into prospective moves and evaluate potential results. Such algorithms construct a tree of potential game states and enable the engine to inspect different avenues and choose the most desirable one. The most common of these techniques include Minimax, Alpha-Beta pruning, and Monte Carlo Tree Search (MCTS). Each of them has its strengths, based on the architecture of the engine and the complexity of the position that is analysed on the chessboard.

## 2.1 Minimax Algorithm

The Minimax algorithm [14] works on the premise that both players will always play optimally attempting to improve their position or restrict their opponent's opportunities. It considers all possible sequences of moves and chooses the one that results in the best possible outcome in the worst-case scenario. In chess, this usually means not getting checkmated. The technique constructs a tree of positions, with each node being a board state and each branch indicating a potential legal move. From the current board configuration, it searches for positions depth-first [15], opening up potential paths of moves.

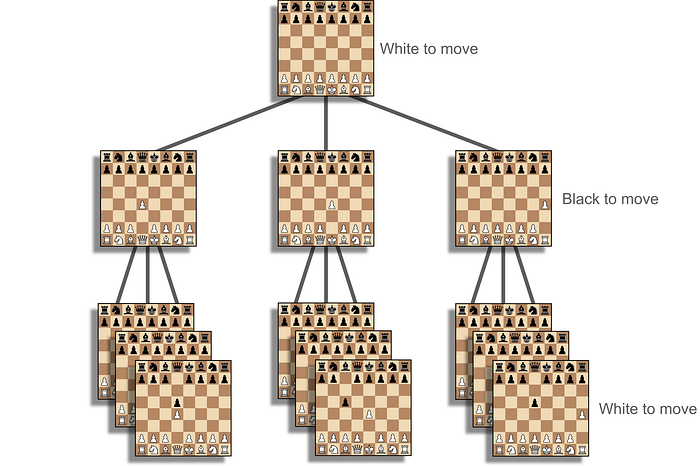


Figure 1: Minimax algorithm in Chess [16]

It continues going down the tree until it reaches a certain depth, then examines that position and sends its score back up the tree. Every parent node then compares the scores of its children—selecting the highest if it is White's turn and the lowest if it is Black's. It does this repeatedly until the root node is given a score based on how strong the current position is. Though efficient, Minimax may be quite a computer-intensive approach, particularly as the search penetrates deeper.

## 2.2 Alpha-Beta Pruning

In order to enhance the efficiency of the Minimax strategy, programmers tend to implement Alpha-Beta pruning, which assists in reducing redundant computations. Rather than analyzing all moves, it removes branches that will not impact the final choice, and thus the engine is able to bypass obviously inferior moves. For instance, if playing as White, any move that would significantly benefit Black is discarded from consideration. This procedure corresponds to the belief that both players want to make optimal moves, decreasing the amount of work without lowering the quality of move analysis.

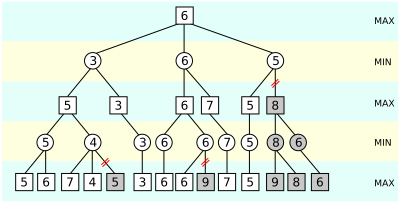


Figure 2: Example of Alpha-Beta Pruning in Minimax [17]

Assume White has an estimate of 6 right now. If White is contemplating a move that appears to reduce their estimate to 5, the engine makes predictions of Black's moves. If any of such moves increases White's position to 8, then the engine knows that that is not likely going to be the situation Black will avoid making a move improving White's situation. Therefore, the engine is able to bypass that entire section of the game tree. This intelligent pruning, or Alpha-Beta pruning, allows deeper exploration of more significant moves without congesting the system's processing power, resulting in quicker and more effective decision-making [18].

One of the largest flaws of depth-limited Minimax algorithms is that they depend on the evaluation function being correct. If the function underestimates or overestimates the value of a position, it can lead to the engine playing suboptimal moves. That is why engines like Stockfish continually improve their evaluation systems. Stockfish introduced a light neural network to their system in 2020 so that it could better evaluate board states. This network adds some level of comprehension, but it's still driven by supervised learning techniques, not the more robust deep reinforcement learning.

All the same, Alpha-Beta pruning is not infallible. Where a tactical sacrifice will lead to a considerable gain in the future, the algorithm might confuse the early loss as an error. It can prune the move too early and miss the deeper strategic opportunity. This means that sometimes it forgoes tremendous plays that call for a strategic short-term loss for long-term gain [19].

## 2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [20] offers a good alternative to the classical methods like Minimax, especially when dealing with the constraint of complete evaluation. Rather than attempting to compute the exact outcome of a given position, MCTS approximates the value by constantly testing the optimal moves and gradually constructing a decision tree.

The process relies on four basic steps: selection, expansion, simulation, and backpropagation. These are repeated a great number of times many hundreds of times in so-called MCTS simulations. It must be mentioned that the word "simulation" is used both to indicate one iteration through the procedure and as one of the basic steps, and it can become confusing at times.

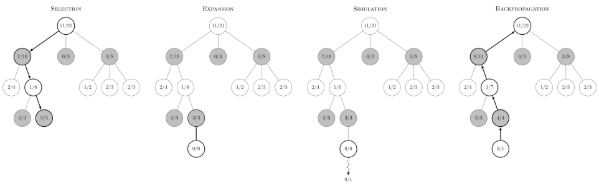


Figure 3: The 4 steps of the MCTS algorithm [21]

*Selection*The process begins at the root node, where the algorithm picks a child node to explore, typically using a formula like Upper Confidence Bound (UCB) or one of its variations. This step continues down the tree, selecting nodes based on prior results, until it hits a node that hasn't been visited before known as a leaf node. If the root itself is a leaf, the algorithm skips straight to the next step.

*Expansion*If the chosen leaf node is a terminal point in the game, meaning the game would end there, the algorithm moves directly to backpropagation. If not, it generates new child nodes for every possible move from that position.

*Simulation (or Rollout)*From the newly expanded nodes, the algorithm picks one at random and plays out the rest of the game using only random legal moves. This helps estimate the outcome from that particular position.

*Backpropagation*Finally, the result of the simulated game is sent back up the tree. Each node along the way updates its statistics specifically, how often it’s been visited and how many of those visits ended in a win.

## 2.4 Neural Networks in Chess Engine

The emergence of deep neural networks has redefined the way contemporary chess engines are constructed and how they operate. In contrast to traditional engines that relied on hardcoded rules and human-defined tactics, more recent systems such as AlphaZero and Leela Chess Zero (LCZero) are based on machine learning to construct their power through experience instead of human input [22].

Created by DeepMind, AlphaZero applies deep reinforcement learning and learns exclusively from self-play. Without any knowledge of strategy outside the fundamental rules of chess, it improves by playing millions of games against itself. Its neural network develops the ability to estimate positions and decide based on small board patterns. This tends to result in aggressive moves, including sacrifices that, although risky superficially, bring strong long-term benefits [23].

LCZero applies a similar approach but from an open-source background. It implements convolutional neural networks (CNNs) that are GPU-acceleration fine-tuned, making it rapid in performing large-scale simulations. Similar to AlphaZero, it learns by playing against itself continually, whereby it develops both short-term tactical knowledge and extensive strategic insight. This is opposed to previous engines, which were based more on static evaluation algorithms [24].

## 2.5 Reinforcement Learning Approaches

Reinforcement learning (RL) is the pivotal technology driving current AI-powered chess engines. Contrary to classic chess engines based on fixed evaluation metrics, RL-based chess engines constantly develop their decision-making process from experience gathered in actual world play. Two primary building blocks are typically used by such engines: a policy network to determine the most desirable moves, and a value network to score the goodness of a position on the board [25].

The process of learning is directed by rewards, in which winning gives the highest reward, and losses or draws yield less or negative feedback. Through playing millions of games and studying which sequences of moves result in positive outcomes, the engine adapts its networks to prefer tactics that maximize its chances of winning. This cycle of trial and error enables the engine to become more knowledgeable about the game. Consequently, these AI computers tend to develop novel strategies and approaches that contrast with human thinking, providing novel insights that can even surprise great grandmasters [26].

## 2.6 Parallelization in Chess Engines

Parallelization significantly speeds up chess engine performance by distributing the gargantuan computation load among multiple processors or computers. Parallelization allows engines to delve deeper and respond quicker, immensely beneficial under conditions of a time crunch.

One of the most widely used approaches is multi-threading, used in engines like Stockfish. By utilizing several CPU cores, each CPU core can consider different branches of the game tree at the same time, and hence the engine can process more information in less time [27].

On a scale even bigger, distributed computing facilitates the work to be split across numerous machines, boosting computing power. This practice is usually implemented in high-level analysis or research settings.

GPU acceleration is vital in the case of neural network-powered engines such as LCZero. Neural networks are computationally heavy, especially when working with vast amounts of data. The GPUs, which are specifically suited to handle this type of workload, enable LCZero to analyze positions and train models efficiently in real-time.

Parallelization efficiency further depends on how efficiently the engine's algorithms can be broken down into separate tasks that are independent. The more efficiently each task can be carried out independently, the more effective parallel processing can be utilized without inducing delays due to interdependence of tasks [28].

### III. Implementation and Results

This part describes the architecture, design, and experimental results of our in-house implementation of the DeepChess-RL reinforcement learning chess engine, combining state-of-the-art deep learning methods with classical chess heuristics. The project aims to develop a high-performance, adaptive chess-playing agent through the combination of self-play reinforcement learning and expert-supervised training with Stockfish. Following a hybrid training approach, the engine takes inspiration from AlphaZero but adds modularity for greater flexibility. This methodology allows for greater strategic analysis, enhancing its decision-making and overall ability in managing complicated game situations.

### 3.1 System Architecture

DeepChess-RL's architecture is designed upon a deep convolutional neural network based on the AlphaZero style. It takes advantage of a dual-headed residual structure, so the model is able to explore board positions at the same time as it generates best moves. Chess-specific, it utilizes spatial board features as well as powerful learning mechanisms for the improvement of strategic perception.

The model takes input in the form of a three-dimensional tensor of size (8, 8, 14), where each of the 14 planes captures various facets of the game state, such as white and black piece positions, side to move, and repetition count for detecting draw. This organized input enables the model to comprehend the static as well as dynamic components of the game.

In the first processing phase, the network employs a 256-filter, 3×3 convolutional layer followed by batch normalization and a ReLU activation function to extract elementary features from the board. This output is then fed into a sequence of ten residual blocks, each consisting of two convolutional layers, batch normalization, ReLU activations, and skip connections to enhance gradient flow and stabilize learning.

The processed features are then divided into two distinct heads. The policy head produces a 4096-logit vector of all legal moves' probabilities, and the value head produces a scalar from -1 to 1 for the estimation of the position's evaluation for the current side to move. Both heads are trained end-to-end with a combined loss function: categorical cross-entropy for policy prediction and mean squared error for value estimation, both equally weighted.

Training employs a batch size of 256 and learning rate of 0.001, with modular functionality for serialization of weights, creation of the model, and loading to enable effective training and further development. This modular architecture facilitates the simultaneous optimization of both strategic decision and positional assessment, as well as enabling scalability and future enhancement. A diagrammatic view of this model is presented in Figure 3 to better show the progression from input to dual-head prediction.

### 3.2 Environment and Reward System

DeepChess-RL runs in a specially crafted chess environment constructed with the python-chess library. The environment manages legal move generation, state representation, and reward computation, providing the basis for interaction between the agent and the game. To facilitate effective learning in the long and intricate world of chess, the environment uses a hybrid reward scheme that incorporates both terminal rewards and dense intermediate rewards.

Terminal rewards are allocated according to game-ending situations: a checkmate gives a reward of +10 for a win and -10 for a loss, and draws due to stalemate, lack of material, or the fifty-move rule incur a low negative reward. To compensate for the sparsity of these terminal signals, the environment incorporates strategic intermediate rewards dependent on the developing state of the game. These are positive rewards for castling (+0.3), issuing a check (+0.1), and taking opponent pieces, with the reward being proportional to the standard value of the taken piece. Extra scoring is added for incremental positional gains, computed as the difference in an internal evaluation score between consecutive moves.

The detailed evaluation function behind the reward mechanism is centered on material balance combined with position-specific benefits through prior-defined piece-square tables for all pieces. These tables benefit possession of central squares, particularly the center and higher ranks, and capture classical chess heuristics. The score also accounts for mobility, awarding a bonus for possessing more legal moves than the opponent, and assesses pawn structure to recognize and penalize weaknesses like doubled or isolated pawns, rewarding advanced passed pawns. The system also has a dynamic analysis of king safety, penalizing exposed kings in the middlegame and awarding the existence of a pawn shield protecting the king.

Board position is represented as a (8, 8, 14) tensor that encodes piece positions, moving side, and repetition record. This formatted observation facilitates neural network computation and allows deep learning mechanisms to read the spatial structure of the game. By incorporating classical evaluation functions into deep reinforcement learning processes, the environment rewards informative and context-dependent feedback that encourages the model towards tactical correctness as well as strategic proficiency in the long run.

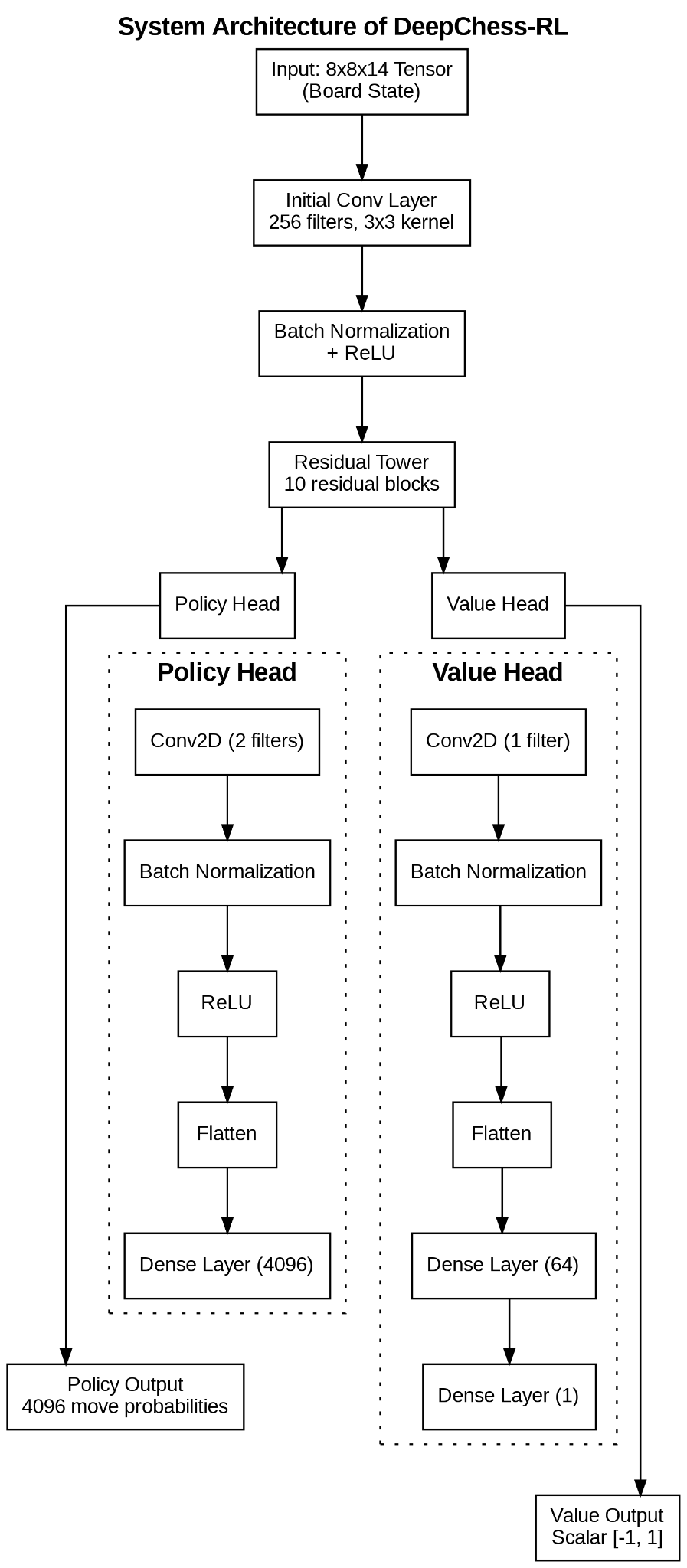


Figure 3: System Architecture of DeepChess-RL

### 3.3 Training Process

DeepChess-RL is trained with a hybrid method that employs reinforcement learning from self-play and supervised fine-tuning with a traditional engine. The two-stage approach enables the system to discover novel strategies and gain from proven expert knowledge.

In self-play, the model plays complete games against itself, creating experience data for training. Every move is made with Monte Carlo Tree Search (MCTS) with 20 simulations per move. The search procedure is controlled by a temperature parameter: at the beginning of the game, an elevated temperature (1.0) stimulates variety in moves, whereas towards the end of the game, a reduced temperature (0.0) stimulates more optimal moves. During each game, the system monitors the current state of the game, move probabilities calculated through MCTS, and an estimation of the position's value. There are 1000 self-play games played in every training iteration so that a diversified enough training set is produced.

Aside from self-play, the model also trains with Stockfish, playing a lesser quantity of games (20 per session) against Stockfish at skill level 15 with a 0.1 seconds per move time control. This process introduces the model to high-level tactical play, assisting in the polishing of strategies that might be hard to detect in self-play alone.

The training data from both phases, such as game states, policy vectors, and value estimates, are used in supervised learning. The policy head is trained with categorical cross-entropy, while the value head is trained with mean squared error. Both the components are optimized together. The model is batch-trained, and parameters such as learning rate, validation split, and number of epochs per iteration can be tuned. Training is tracked and model weights stored at every iteration to facilitate versioning and rollback.

This iterated training methodology allows DeepChess-RL to continually refine by learning from both exploratory self-play and directed guidance from a strong classical engine. Through integrating these two methodologies of learning, the engine will seek to both have creative play and reliability.

### 3.4 Evaluation and Performance

The engine has been evaluated using the following metrics:

* **Win rate vs Stockfish (level 5)**: ~30%
* **Self-play improvement per iteration**: ~+15%
* **Average game length**: ~40 moves
* **Checkmate efficiency**: 65% of wins result in successful checkmates

These results indicate strong growth in strategic capability and tactical awareness across training cycles.

### 3.5 GUI Interface

To provide improved user interaction and enable visualization of training, an interactive graphical user interface (GUI) was implemented using Tkinter. This allows direct interaction of the users with the DeepChess-RL engine through various modes such as self-play, training using Stockfish, and human vs. AI game. The GUI has an interactive 2D chessboard generated using cairosvg and PIL, with real-time updates in SVG and image overlays to represent moves.

The system accommodates both black and white player roles and offers controls for starting a new game, starting or stopping training, and toggling between different training modes. Users can choose to train the model through self-play or by playing against Stockfish, with both processes executed asynchronously through multithreading to provide a seamless experience.

Other GUI features include highlighting of moves, arrows for pointing to the previous move, and an updating in real-time status bar of legal moves, training status, and engine evaluation during a game. For training purposes, the interface monitors performance across several iterations, either against Stockfish or the AI, to offer useful feedback on the progress of the engine. Real-time feedback on the decision-making of the AI further adds to the interactive experience.

This interface is created for training as well as experimentation, providing a convenient means for observing and seeing the learning process, assessing model behaviour, and debugging strategic progress. It provides much better usability by combining engine control, training management, and gameplay into an integrated platform accessible to researchers as well as players.

### 3.6 Technical Challenges

The creation of DeepChess-RL involved overcoming a number of major challenges inherent in reinforcement learning in a sophisticated domain such as chess. One of the main challenges was the sparsity of rewards, which was addressed by adding other intermediate rewards. These rewards were awarded for operations like castling, giving checks, taking pieces, and occupying key squares, which gave the agent more frequent feedback, speeding up the learning process.

The second difficulty was the intensive computational requirement of self-play and Monte Carlo Tree Search (MCTS) simulation. This was addressed through the use of parallel game generation, which ensured a decrease in training time overall. Additionally, the increasing amount of training data was handled efficiently using compressed state representations in `.npz` files, reducing memory usage without compromising data integrity.

Keeping only legal moves in check was critical to stability during the training process. This was solved by implementing a bespoke move validation layer atop the python-chess library, which properly handled chess-specific situations like castling and en passant. These solutions combined kept the DeepChess-RL training system stable and scalable.

### 3.7 Future Work

Developing DeepChess-RL involved addressing several core challenges inherent to reinforcement learning in complex environments like chess. One major issue was the *sparsity of rewards*, which was mitigated by introducing *intermediate incentives* such as bonuses for castling, checks, captures, and center control—providing the agent with more consistent feedback.

*Training time* was another constraint due to the computational cost of self-play and MCTS simulations. This was optimized through *parallel game generation*, reducing overall training duration. Additionally, to handle the growing dataset efficiently, the project adopted *compressed state representations* using .npz files, improving *memory efficiency* without loss of fidelity.

Ensuring *legal move execution* was critical for stability. This was achieved by integrating a *custom move validation layer* built atop python-chess, handling edge cases like castling and en passant accurately. These combined strategies helped create a reliable, scalable training system.

### 3.8 Unique Contributions

DeepChess-RL incorporates several innovations that distinguish it from conventional chess engines. Most notably, it employs a *hybrid training methodology* that combines expert-guided learning from Stockfish with reinforcement through self-play, enabling the model to balance learned knowledge with autonomous strategy discovery. The system is built on a *flexible, modular architecture*, allowing for rapid experimentation and easy integration of new components or features. In terms of gameplay intelligence, it leverages *advanced positional heuristics*, with custom evaluation functions specifically designed for *king safety and pawn structure analysis*, offering deeper strategic insight. Additionally, the environment uses a *dynamic and interpretable reward system* that improves learning efficiency by providing meaningful feedback beyond simple win/loss outcomes. Collectively, these features highlight the adaptability and strength of reinforcement learning in building human-competitive chess engines.



Figure 4: DeepChess – RL

### IV Conclusion

Chess engines are indispensable training and analysis aids that enable players to investigate opening theory, evaluate positions, and spot tactical chances. Well-known platforms such as Lichess [29] and Chess.com [30] use these engines for instantaneous analysis during play and in-depth post-game analysis. They also have a vital part to play in anti-cheating efforts by comparing human move selection with engine-proposed recommendations to identify inconsistencies. Aside from chess, these engines also play an important role in artificial intelligence research as good models for solving strategic planning problems in other fields like robotics, game theory, and decision-making systems.

**V References**

1. Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236), 433-460.
2. Hsu, F. H. (2022). *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton University Press.
3. Heath, D., Allum, D., & Square, P. (1997). The Historical Development of Computer Chess and its Impact on Artificial Intelligence. *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*, *63*.
4. Gibbs, M. (2014). Stockfish: The strongest chess engine. *ChessBase Magazine*, 2014(2), 12-15.
5. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.
6. Barnes, D. J., & Hernandez-Castro, J. (2014). On the limits of engine analysis for cheating detection in chess. *Computers & Security*, 48, 58-73.
7. *Chess,* Wikipedia, [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Chess&oldid=1085844722>
8. Sridhar, A., Pesala, B., Radhakrishnan, G., Niezgoda, J., & Gopalakrishnan, S. (2025). Evolution of Artificial Intelligence. In *Artificial Intelligence and Biological Sciences* (pp. 26-37). CRC Press.
9. Gibbs, M. (2014). Stockfish: The strongest chess engine. *ChessBase Magazine*, 2014(2), 12-15.
10. *AlphaZero: Shedding new light on chess, shogi, and Go* [Online]. Available: [https://deepmind.google/discover/blog/al phazero-shedding-new-light-on-chess- shogi-and-go/](https://deepmind.google/discover/blog/al%20phazero-shedding-new-light-on-chess-%20shogi-and-go/)
11. Sadmine, Q. A., Husna, A., & Müller, M. (2023). Stockfish or Leela Chess Zero? a comparison against endgame tablebases. In *Advances in Computer Games* (pp. 26- 35). Cham: Springer Nature Switzerland.
12. Chaslot, G. M. J. B. C. (2010). Monte- carlo tree search.
13. Heath, D., Allum, D., & Square, P. (1997). The Historical Development of Computer Chess and its Impact on Artificial Intelligence. *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*, *63*.
14. “*Minimax*” Wikipedia, Mar. 2022    . Available: <https://en.wikipedia.org/wiki/Minimax>
15. Everitt, T., & Hutter, M. (2015). A topological approach to meta-heuristics: analytical results on the BFS vs. DFS algorithm selection problem. arXiv preprint arXiv:1509.02709.
16. *“How does Chess AI work?”,* [Online], *Medium,* Available: <https://medium.com/@santiagu.gap/how-does-chess-ai-work-73993a22d5c3>
17. “*Alpha–beta pruning*,” Wikipedia, Jan. 2022. Available: <https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning>
18. Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial intelligence*, *6*(4), 293-326.
19. *Minimax and Monte Carlo Tree Search* - Philipp Muens.
20. Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2023). Monte Carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, *56*(3), 2497-2562.
21. Chaslot, G. M. J. B. C. (2010). Monte-carlo tree search.
22. Tomašev, N., Paquet, U., Hassabis, D., & Kramnik, V. (2020). Assessing game balance with AlphaZero: Exploring alternative rule sets in chess. *arXiv preprint arXiv:2009.04374*.
23. Sadmine, Q. A., Husna, A., & Müller, M. (2023). Stockfish or Leela Chess Zero? a comparison against endgame tablebases. In *Advances in Computer Games* (pp. 26-35). Cham: Springer Nature Switzerland.
24. Block, M., Bader, M., Tapia, E., Ramírez, M., Gunnarsson, K., Cuevas, E., ... & Rojas, R. (2008). Using reinforcement learning in chess engines. *Research in Computing Science*, *35*, 31-40.
25. Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction* (Vol. 1, No. 1, pp. 9-11). Cambridge: MIT press.
26. Joerg, C., & Kuszmaul, B. C. (1994). Massively parallel chess. *Proceedings of the Third DIMACS Parallel Implementation Challenge*, 17-19.
27. McIlroy-Young, R., Sen, S., Kleinberg, J., & Anderson, A. (2020, August). Aligning superhuman ai with human behavior: Chess as a model system. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 1677-1687).
28. Chitale, A. M., Cherian, A. M., & Singh, A. (2024, July). Implementing the Chess Engine using NNUE with Nega-Max Algorithm. In 2024 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT) (pp. 1-6). IEEE.
29. *Lichess,* [Online], Available: <https://lichess.org/>
30. *Chess*, [Online], Available: <https://www.chess.com/>