Chess in the AI Era: From Classical Engines to Deep Reinforcement Learning

Yinon Kedem

August 20, 2025

Abstract

Chess has long been a benchmark for artificial intelligence research. In this paper, we present the design and implementation of a full-stack chess application that integrates modern web technologies with state-of-the-art chess engines. We analyze two paradigms of chess AI — Stockfish, a classical search-based engine, and AlphaZero, a reinforcement learning–based engine — and compare their performance, strengths, and implications for the future of AI in strategic games.

1 Introduction

Chess has long been a benchmark domain for artificial intelligence research. Its well-defined rules, enormous search space, and strategic depth have made it a fertile ground for testing and advancing computational techniques. Over the decades, chess engines have evolved from brute-force search algorithms to highly sophisticated machine learning models capable of rivaling and surpassing the strongest human grandmasters.

In this project, we developed a full-stack chess application that integrates modern web technologies with state-of-the-art chess engines. The backend of the system is implemented using **FastAPI**, a high-performance Python web framework, while the frontend is built with **React** and **Node.js**, ensuring a responsive and interactive user interface. This design allows players not only to play chess online but also to interact with advanced artificial intelligence agents capable of suggesting optimal moves and providing real-time analysis.

Central to this project is the integration of two different paradigms of chess engines: **Stockfish** and **AlphaZero**. Stockfish, one of the strongest traditional engines, relies on *minimax search with alpha–beta pruning*, enhanced by a finely tuned *handcrafted evaluation function* and efficient use of *endgame tablebases*. In contrast, AlphaZero, developed by DeepMind, represents a paradigm shift: it is a reinforcement learning–based system that learns chess entirely through *self-play* using *deep neural networks* in combination with the *Monte Carlo Tree Search (MCTS)* algorithm.

The purpose of this article is threefold:

1. To describe the design and implementation of the chess application, focusing on its technical stack and system architecture.

2. To provide an in-depth explanation of how both Stockfish and AlphaZero operate, highlighting the differences in their approaches to decision-making.

3. To analyze and compare the performance of these two engines, emphasizing why AlphaZero has surpassed traditional engines such as Stockfish in certain dimensions of play.

By combining practical software development with theoretical insights into artificial intelligence, this project illustrates both the challenges and opportunities of building chess applications that leverage modern AI techniques.

2 The Complexity of Chess

One of the most fundamental challenges in computer chess is the sheer size of the game tree. At any given position, a player may have around 30 to 40 legal moves on average, and the number of possible game states grows exponentially with each move. The total number of unique possible chess games has been estimated to be on the order of 10^{120}, a value known as the *Shannon number*. This figure is astronomically larger than the estimated number of atoms in the observable universe ($\sim 10^{80}$).

Because of this exponential growth, computing the outcome of every possible game is infeasible with current or foreseeable classical computing technology. Even if every atom in the universe were a supercomputer, running from the beginning of time, it would still not be enough to enumerate and evaluate all possible chess positions. While *tablebases* exist for endgames with up to seven pieces, covering the full game of chess remains far beyond reach.

Humanity may only approach perfect chess with radical breakthroughs in computing, such as practical large-scale quantum computers or other yet-unknown paradigms of computation. Until then, brute-force methods cannot solve chess.

Modern artificial intelligence offers a more powerful and feasible alternative. Instead of exhaustively calculating all positions, engines such as Stockfish use deep search combined with heuristic evaluation, while systems like AlphaZero leverage **deep neural networks** and **reinforcement learning**. These methods allow the engine to approximate the value of positions and prioritize promising lines of play, achieving superhuman performance without solving the game completely. This shift illustrates why current AI technologies, particularly neural networks combined with tree search, are so effective at tackling problems with immense combinatorial complexity like chess.

3 Stockfish: A Classical AI Approach to Chess

Artificial Intelligence (AI) broadly refers to the simulation of human intelligence by machines, enabling them to learn, adapt, and make decisions. In the context of games like chess, AI systems aim to **choose optimal moves** through strategic reasoning and calculation. Stockfish, one of the strongest chess engines in the world, exemplifies a *classical* AI approach: instead of learning by example or self-play, it relies on brute-force **search algorithms** combined with human-crafted heuristics to navigate the enormous game tree.

Origins and credits. *Stockfish* began as a 2008 fork of *Glaurung*, an engine by **Tord Romstad**. The project was initiated and led by **Marco Costalba**, with **Joona Kiiski** later joining as a core developer. **Gary Linscott** created and maintains *Fishtest*, the distributed testing framework that powers Stockfish's large-scale, community-driven A/B testing and tuning. Today Stockfish is developed by hundreds of contributors under the GPL, and has won numerous top engine events (e.g., TCEC). The name "Stockfish" nods to the engine's Norwegian–Italian roots (dried cod is a staple in both regions).

This section explains how Stockfish works, covering its core decision-making algorithm (minimax), the crucial optimization of alpha–beta pruning, the design of its evaluation function, data structures that boost efficiency, the use of endgame tablebases, and why these elements make Stockfish exceptionally powerful in practice.

3.1 Minimax Decision-Making Algorithm

At the heart of Stockfish's play is the **minimax algorithm**, a method from game theory for decision-making in adversarial, zero-sum games. In chess, minimax assumes both players play optimally: Stockfish (the maximizing player) tries to maximize the evaluation score, while the opponent (the minimizing player) tries to minimize it. Each position is a node in a game tree, and the values at terminal nodes are given by an *evaluation function.* These values propagate back up the tree: at a Max node, the maximum of the children's values is chosen, and at a Min node, the minimum.

Algorithm 1 Minimax with Alpha-Beta Pruning

```
1:  function AlphaBeta($node$, $depth$, $\alpha$, $\beta$, $maximizing$)
2:      if $depth = 0$ or terminal($node$) then
3:          return evaluate($node$)
4:      if $maximizing$ then                                          ▷ Max player (Stockfish)
5:          $value \leftarrow -\infty$
6:          for all $child$ in children($node$) do
7:              $value \leftarrow \max\{value, \text{AlphaBeta}(child, depth-1, \alpha, \beta, \textbf{false})\}$
8:              $\alpha \leftarrow \max\{\alpha, value\}$
9:              if $\alpha \geq \beta$ then
10:                 break                                                ▷ Beta cutoff
11:         return $value$
12:     else                                                          ▷ Min player (opponent)
13:         $value \leftarrow +\infty$
14:         for all $child$ in children($node$) do
15:             $value \leftarrow \min\{value, \text{AlphaBeta}(child, depth-1, \alpha, \beta, \textbf{true})\}$
16:             $\beta \leftarrow \min\{\beta, value\}$
17:             if $\beta \leq \alpha$ then
18:                 break                                               ▷ Alpha cutoff
19:         return $value$
```

3.2 Alpha–Beta Pruning for Efficient Search

Alpha–beta pruning improves minimax by skipping branches that cannot affect the final decision. It keeps two bounds: α (best score for the maximizer so far) and β (best score for the minimizer so far). If $\alpha \geq \beta$ at any point, the algorithm prunes the rest of that branch. This allows Stockfish to search much deeper than a naive minimax search.

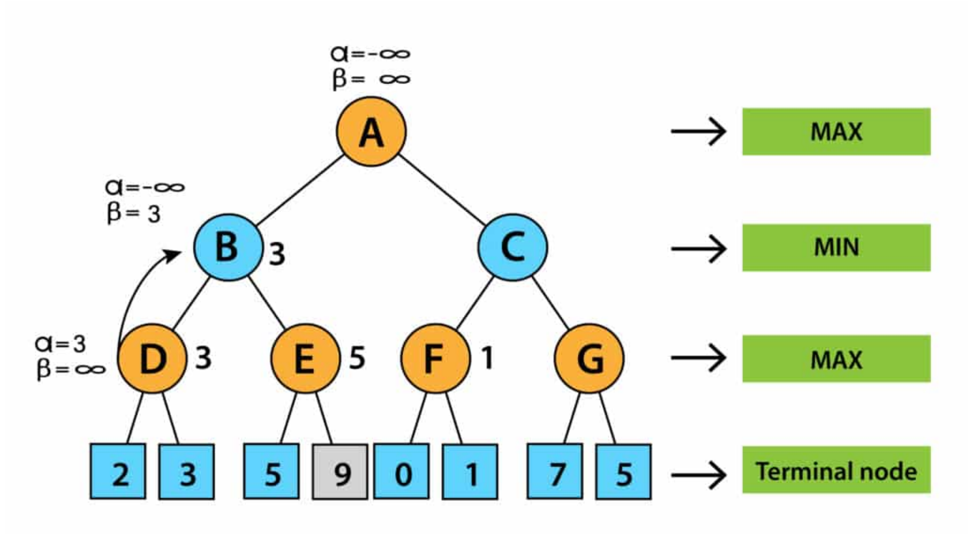

Figure 1: Illustration of alpha–beta pruning in a minimax tree. MAX nodes (orange) try to maximize the score, MIN nodes (blue) try to minimize it, and leaf nodes are evaluated positions. Once node B determines that its value cannot exceed 3, the subtree under node E (value 9) is pruned. This demonstrates how alpha–beta avoids exploring moves that cannot affect the final decision, greatly reducing the number of positions that must be evaluated.

3.3 Stockfish's Evaluation Function

Minimax needs a way to score positions. Stockfish uses a finely tuned **evaluation function** that considers:

- **Material:** The value of pieces (Pawn=1, Knight/Bishop≈3, Rook=5, Queen=9, etc.).
- **Mobility:** How many legal moves pieces have.
- **King Safety:** Pawn shield, exposure to enemy pieces, castling.
- **Pawn Structure:** Isolated, doubled, and passed pawns.
- **Piece Activity:** Piece-square tables and coordination.

Scores are reported in *centipawns*. For example, +100 means White is ahead by the equivalent of one pawn.

3.4 Efficient Data Structures

Stockfish achieves speed by using:

- **Bitboards:** 64-bit integers representing board states, enabling fast bitwise operations for move generation.
- **Magic Bitboards:** Precomputed lookup tables for sliding pieces.
- **Transposition Tables:** Hash tables that store previously evaluated positions to avoid recomputation.

These optimizations allow Stockfish to search millions of positions per second.

3.5 Endgame Tablebases

Stockfish integrates precomputed **endgame tablebases** (e.g., Syzygy) for up to 6–7 pieces. These databases provide perfect information: whether a position is a win, loss, or draw with best play. This ensures flawless play in simplified endgames.

3.6 Why Stockfish is Powerful

Combining:

- Deep search with alpha–beta pruning,
- A highly tuned evaluation function,
- Efficient data structures,
- Endgame tablebases,

makes Stockfish one of the strongest engines ever built. It routinely evaluates billions of positions per game, reaches depths of 30+ plies, and outperforms any human player.

4 AlphaZero: Self-Learning AI Chess Engine

AlphaZero represents a breakthrough in game-playing AI by mastering chess (and, notably, Go and Shogi) *from scratch* using self-play reinforcement learning. Unlike classical engines with human-crafted evaluation rules and opening books, AlphaZero is given only the rules of chess. It then learns strategies and evaluation directly from experience, coupling a deep neural network with Monte Carlo Tree Search (MCTS).

Origins and credits. *AlphaZero* was developed at **DeepMind** (Alphabet) as the chess/shogi/go generalization of the *AlphaGo* and *AlphaGo Zero* line of systems. The first AlphaZero chess/shogi report appeared in late 2017 (*Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*), led by **David Silver** and co-authored by **Thomas Hubert**, **Julian Schrittwieser**, **Ioannis Antonoglou**, **Matthew Lai**, **Arthur Guez**, **Marc Lanctot**, **Laurent Sifre**, **George van den Driessche**, **Thore Graepel**, and **Demis Hassabis**, among others. Training and inference in the original work were accelerated with **TPUs**. While the AlphaZero codebase itself was not open-sourced, the community-created project *Leela Chess Zero (Lc0)* re-implemented the approach from the published papers and released an open engine inspired by AlphaZero's design.

4.1 Self-Play Reinforcement Learning Framework

AlphaZero improves via an iterative self-play loop:

1. **Self-play game generation:** Using the current network, the engine plays many games against itself. At each turn it runs MCTS to pick a strong yet exploratory move.

2. **Data collection:** For every position s_t, it records (s_t, π_t, z_t) where π_t is the MCTS-improved move distribution at that state and $z_t \in \{-1, 0, +1\}$ is the final game result from that position's perspective.

3. **Network update:** The network parameters θ are trained so that the value head $v_\theta(s_t)$ matches z_t and the policy head $p_\theta(s_t)$ matches π_t (typically via a value loss + policy cross-entropy + regularization).

4. **Iteration:** The updated network replaces the old one and the process repeats over millions of self-play positions, steadily improving play.

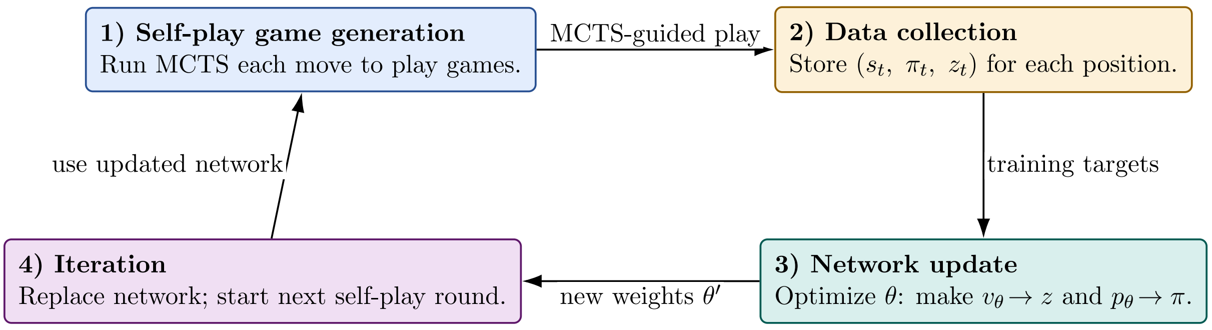
1) Self-play game generation
Run MCTS each move to play games.
MCTS-guided play
2) Data collection
Store $(s_t,\ \pi_t,\ z_t)$ for each position.
training targets
3) Network update
Optimize θ: make $v_\theta \to z$ and $p_\theta \to \pi$.
new weights θ'
4) Iteration
Replace network; start next self-play round.
use updated network

Figure 2: AlphaZero self-play training cycle (four stages). The neural network guides MCTS during self-play; completed games produce (s_t, π_t, z_t) targets; optimization updates θ; the new network is deployed for the next self-play round.

4.2 Deep Neural Networks for Policy and Value

AlphaZero uses a deep neural network $f_\theta(s)$ that outputs:

$$f_\theta(s) \;=\; \big(p_\theta(s),\, v_\theta(s)\big),$$

where $p_\theta(s)$ is a probability distribution over legal moves (the *policy*) and $v_\theta(s) \in [-1, 1]$ estimates the game outcome from the side to move (the *value*). In practice, f_θ is a residual convolutional network that encodes the board into feature maps, then branches into:

- a *policy head* producing move probabilities (illegal moves are masked), and
- a *value head* producing a single scalar.

Rather than encoding human knowledge (e.g., "a queen is worth 9"), the network learns positional judgment directly from the outcomes of self-play.

4.3 Monte Carlo Tree Search (MCTS) in AlphaZero

For each move, AlphaZero runs many MCTS simulations from the current position. Each simulation traverses the tree by selecting moves that balance exploitation (high estimated value) and exploration (high prior probability but low visits). A common selection rule is the PUCT criterion:

$$a^* \;=\; \arg\max_a \left[Q(s,a) \;+\; c_{\text{puct}} \cdot P(s,a)\, \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \right],$$

where $Q(s, a)$ is the running value estimate for move a, $P(s, a)$ is the network prior from p_θ, and $N(s, a)$ is the visit count. At a new leaf, the network provides (p_θ, v_θ); the value v_θ is backpropagated up the path to update Q and N. After many simulations, the root visit counts $\{N(s, a)\}$ define an improved policy π; the move with the highest $N(s, a)$ is played (with temperature/sampling used during training for exploration).

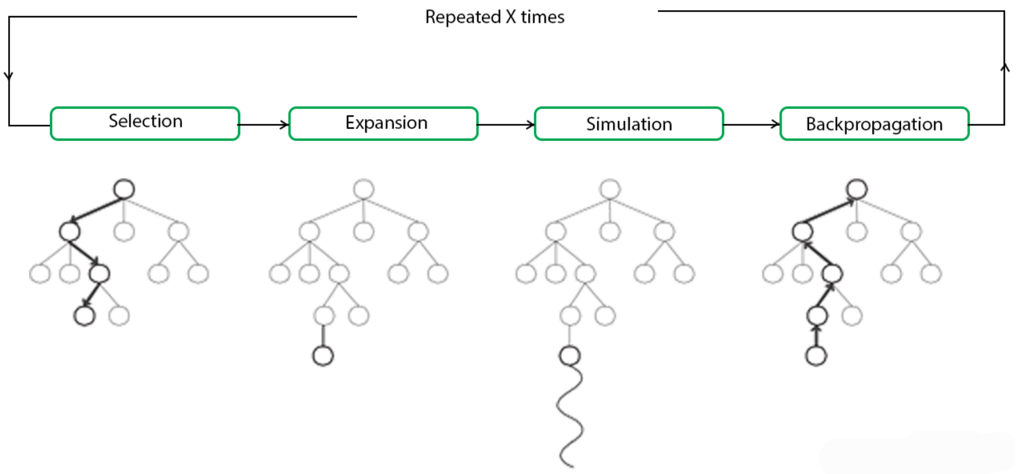

Figure 3: Neural-network–guided MCTS. The policy head provides priors that steer exploration toward promising moves; the value head evaluates leaf nodes. Visit counts at the root form the improved policy used to select the move.

4.4 No Handcrafted Rules or Databases

AlphaZero does not use opening books, endgame tablebases, or handcrafted evaluation terms during training or play. Its knowledge emerges from self-play: the network internalizes strategic patterns (e.g., pawn structures, king safety, initiative) because such patterns lead to better outcomes in practice.

4.5 Move Selection at Play Time

During play, after running a fixed budget of MCTS simulations, AlphaZero selects:

$$a^* = \arg\max_a N(s_{\text{root}}, a),$$

i.e., the move most visited by MCTS. In training games a temperature parameter may be applied to encourage exploration; in evaluation/competition it typically selects the top move deterministically.

In summary, AlphaZero couples a learned position evaluator and move prior with a principled planning algorithm (MCTS). By learning solely from self-play and using the network to focus search on the most relevant continuations, it achieves superhuman strength in chess while remaining free of human-crafted heuristics.

5 AlphaZero vs. Stockfish: Comparative Analysis

This section contrasts AlphaZero and Stockfish along their goals, algorithms, throughput, and empirical performance. We emphasize *how* they obtain strength (learning vs. search), not just final Elo or match results.

5.1 Methodological Differences

- **Knowledge source.** *Stockfish* encodes chess knowledge via a hand-crafted (now hybrid/NNUE-based) evaluation and deep alpha–beta search with sophisticated pruning and heuristics. *AlphaZero* learns *both* policy and value end-to-end from self-play, with no opening books or endgame tablebases during play.

- **Search style.** *Stockfish* expands a very large portion of the game tree using alpha–beta, relying on heavy pruning and precise move ordering. *AlphaZero* performs selective *MCTS* guided by the network's policy priors $p(s)$ and backed by its value estimates $v(s)$.

- **Throughput.** In DeepMind's 2017 evaluation, AlphaZero examined on the order of 10^4–10^5 positions/s while Stockfish evaluated on the order of 10^7–10^8 positions/s; AlphaZero's selectivity (learned guidance) compensates for the far lower node rate.[1]

5.2 Empirical Results (2017 AlphaZero paper context)

AlphaZero trained from scratch and surpassed Stockfish within hours. In a 100-game match at *one minute per move*, AlphaZero scored $+28 = 72 - 0$ vs. Stockfish 8 (evaluation conditions from the original paper).[2] Training curves show rapid Elo growth during early thousands of training steps across chess, shogi, and Go.

[1]Reported figures in contemporary discussions: $\sim$ 80,000 positions/s for AlphaZero vs. $\sim$ 70 million for Stockfish in the match setup.

[2]See Silver et al., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* (2017), arXiv:1712.01815. The supplement shows games from the 1-minute/move match.

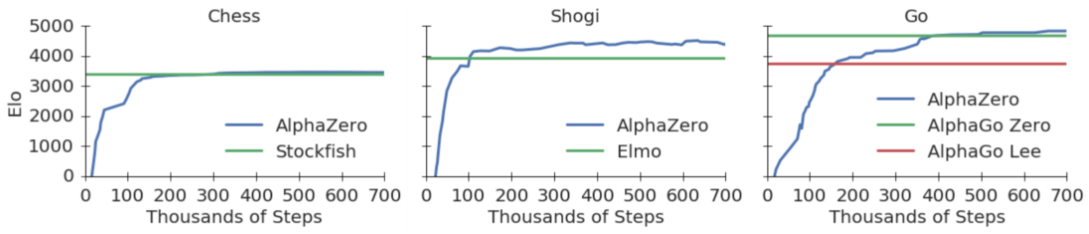

Figure 1: Training *AlphaZero* for 700,000 steps. Elo ratings were computed from evaluation games between different players when given one second per move. **a** Performance of *AlphaZero* in chess, compared to 2016 TCEC world-champion program *Stockfish*. **b** Performance of *AlphaZero* in shogi, compared to 2017 CSA world-champion program *Elmo*. **c** Performance of *AlphaZero* in Go, compared to *AlphaGo Lee* and *AlphaGo Zero* (20 block / 3 day) (*29*).

AZ

vs SF.png

Figure 4: Training curves reproduced from a public summary of the 2017 AlphaZero results: rapid self-play learning in chess (compared to Stockfish 8), shogi (Elmo), and Go (AlphaGo variants).

5.3 Side-by-Side Summary

	Stockfish (classical engine)	**AlphaZero (learning + MCTS)**
Core algorithm	Alpha–beta minimax with heavy pruning, iterative deepening, killer/history heuristics; transposition tables; bitboards/"magic" move gen.	Neural network $f_\theta(s) \to (p, v)$ guides *PUCT* MCTS; value head replaces rollouts; policy head biases exploration.
Knowledge source	Hand-crafted/engineered evaluation (now NNUE neural eval in modern SF); endgame tablebases for perfect small endgames.	No handcrafted chess features; learns purely from self-play; no opening books or tablebases in play.
Nodes/second (match era)	$\sim 10^7$–10^8 positions/s (order of magnitude).	$\sim 10^4 - 10^5$ positions/s (order of magnitude), but highly selective via learned priors/values.
Hardware (2017 paper)	CPU (many threads); no TPUs/GPUs required to play.	Trained and run with TPUs; inference guides MCTS.
2017 head-to-head (paper)	Lost 100-game match to AZ at 1 min/move (0 losses for AZ).	Won 28-0 with 72 draws vs. Stockfish 8 after hours of training.

Table 1: High-level comparison in the 2017 evaluation context. Modern Stockfish later integrated NNUE (neural evaluation), which substantially increased strength; head-to-head outcomes today depend on versions, time controls, and hardware.

5.4 Takeaways

- **Different routes to strength.** Stockfish achieves power through *breadth* (fast, deep alpha–beta) plus refined heuristics; AlphaZero achieves power through *learning* (a strong evaluator and move prior) coupled with targeted MCTS planning.
- **Why AlphaZero could win while searching less.** A learned policy/value steers search into promising lines and away from noise, providing "quality over quantity" expansions.
- **Evolving landscape.** Since 2020, Stockfish's *NNUE* neural evaluation closed much of the gap and often leads modern rating lists; absolute superiority depends on settings (version, time control, hardware).

6 Conclusion and Outlook

This project was as much a software build as it was a learning journey. Designing a full-stack chess application and studying two very different AI paradigms—the classical, search-centric Stockfish and the self-learning AlphaZero—sharpened my understanding of how modern AI systems reason about strategic decision-making. Implementing the backend with **FastAPI** and the frontend with **React/Node.js**, integrating Stockfish for analysis, and experimenting with a basic AlphaZero pipeline gave me a hands-on perspective on the trade-offs between exhaustive search and learned evaluation.

I especially enjoyed exploring how *policy/value* networks coupled with *Monte Carlo Tree Search* can achieve superhuman play while expanding far fewer nodes than classical engines. At the same time, working closely with Stockfish reinforced just how powerful well-engineered alpha–beta search, rich heuristics, and tablebases can be.

Project repository. All code, experiments, and the web application are available in my GitHub repository:

`github.com/yinonkedem/ChessAI`

The repo includes the chess web app (FastAPI + React) and my *basic* AlphaZero implementation used for self-play experiments.

Live website. Play with the app here:

`yinon-chess-ai.vercel.app`

I hope this article helps other builders and learners bridge theory and practice in chess AI. Contributions and feedback are welcome via the repository.

Acknowledgments

I am deeply grateful to **Prof. Jeffrey Rosenschein** of *The Hebrew University of Jerusalem* for his generous guidance and steady encouragement throughout this project.