Chess AI: α–β search with incrementally updatable DNN evaluation

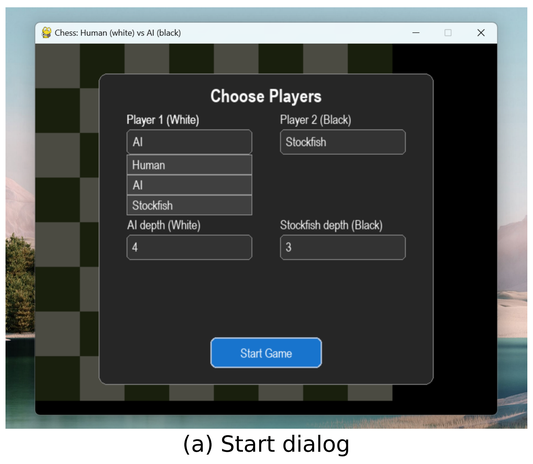
Arjun Vijay Anup & Jainel Liz Boban

ACM40960: **Projects in Maths Modelling, MSc Data & Computational Science**

— University College Dublin (UCD)

Abstract

Our project explores a **hybrid** chess engine that combines classical search techniques with a neural network-based evaluation function. The engine employs **α–β pruning**, a core component of engines like Stockfish, to efficiently search the game tree. Following the "**full mode**" methodology of engines like AlphaZero, our AI uses a **deep neural network (DNN)** to evaluate board positions instead of legacy handcrafted rules. We evaluate the performance of our engine by comparing its **search efficiency** and **strength** against a fixed-depth Stockfish engine.



(a) Start dialog

(b) Checkmate screen

Figure 1. UI/UX: (a) Pygame start popup: choose Human/AI/Stockfish; set search depth for engines. (b) Checkmate: Our engine (White) delivers a checkmate against Stockfish (Black).; Also supports stalemate, threefold repetition, 50-move rule, and insufficient material.

Methodology & Model Training

Positions are encoded as a 782-dimensional, white-perspective **NNUE-style** feature vector and scored by a Deep Neural Network (DNN). During α–β (negamax) search, we use **per-parent mini-batching** to evaluate sibling leaves efficiently while preserving pruning. The engine is implemented in **Python** with **TensorFlow**; inference currently runs in **float32** via a **JIT-compiled** predict graph to cut call overhead.

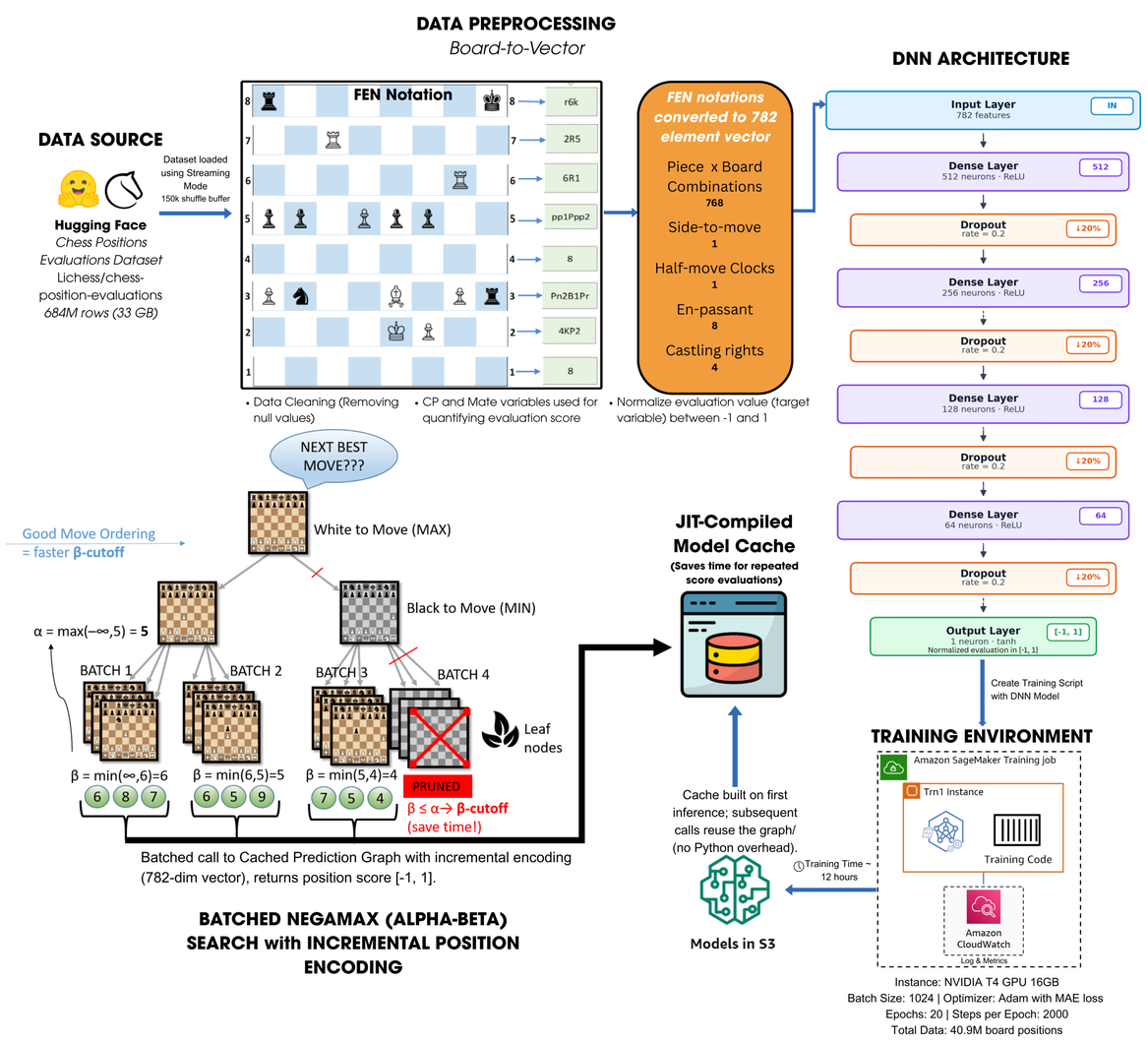
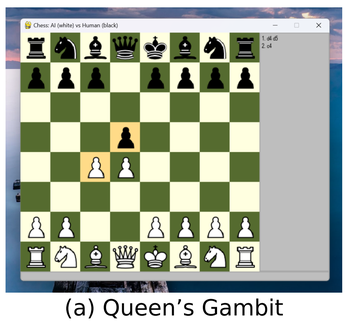


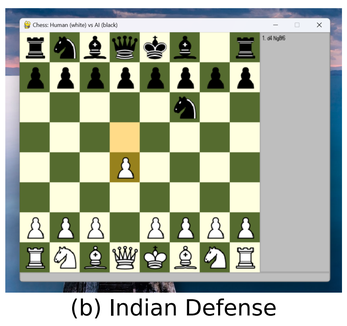
Figure 2. Full system pipeline, including data encoding, the network architecture, batched search, and the training environment.

Other Features

- **Killer moves** (2/ply) and global **history heuristic** to sharpen move ordering and improve pruning efficiency.
- **Transposition table** to prevent redundant evaluations of positions.
- **Anti-ping-pong** bias to avoid draws in favorable positions.
- **Opening book** to reduce AI predictability and early game compute.



(a) Queen's Gambit



(b) Indian Defense

(c) Zukertort Opening

Figure 3. Randomized opening phase: temperature-weighted book moves (≤20 plies) for both colors.

Performance Results

After verifying **move legality** via perft for depths 1-5, we ran 50 fixed-depth engine-vs-engine games against Stockfish. Plots/tables report learning curve (train/val MSE), W–D–L outcomes, and search efficiency.

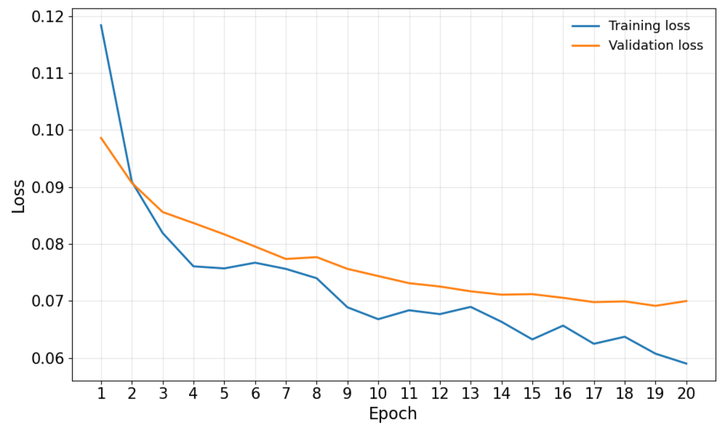


Figure 4. Training vs validation loss (MSE) across 20 epochs; validation converges more slowly than training, as expected for chess positions.

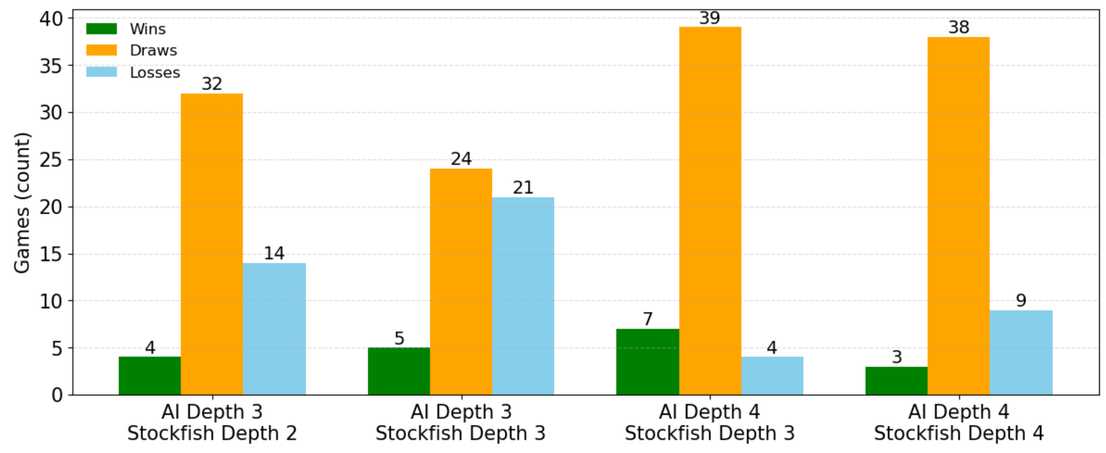


Figure 5. Arena Outcomes vs Stockfish — 50 game simulations per depth combination, W–D–L

Our AI Depth	2	3		4		
Stockfish Depth	2	2	3	2	3	4
Score	0.32	0.40	0.34	0.44	0.53	0.44
ΔElo	-130.944	-70.437	-115.226	-41.894	+20.871	-41.894
SF Elo (est.)	1700	1700	1770	1700	1770	1830
AI Elo (est.)	1569	1630	1655	1658	1791	1788
Median AI time/move (s)	0.225	1.650	1.613	10.916	9.600	10.950
Median SF time/move (s)	0.001	0.001	0.001	0.001	0.001	0.002
Average Nodes/sec (NPS)	127	122	116	144	154	149
Average TT hits	0	5	5	10,184	10,017	9,886
Average Beta-cutoffs	835	5,858	6,736	73,520	69,946	70,285
Average Killer move uses	124	692	726	11,260	11,933	9,659
Average History lookups	125	699	737	11,666	12,417	10,027

Figure 6. Fixed-depth arena matches vs Stockfish (50 games each). Time metrics are medians; NPS/TT/β-cutoffs/killer/history are pooled averages. Positive ΔElo favors our AI. TT = transposition table.

Conclusion

- **Hybrid Approach Success:** Our engine achieved an estimated Elo of ∼1.5–1.8k at depths 2–4. This performance clearly surpassed our classical numeric engine and demonstrated real tactical gains.
- **Effective Heuristics:** With depth, β-cutoffs and TT/killer/history usage increased—evidence of stronger move ordering and more efficient pruning.
- **Optimal Pairing:** The only positive △Elo was observed at a depth pairing of our AI-4 vs. Stockfish-3, providing evidence that a marginal increase in search depth allows our DNN to compete more effectively.
- **Speed:** Much slower than Stockfish at fixed depths—unsurprising given Python + TensorFlow float32 vs Stockfish's C++ NNUE (int8 + SIMD). Median time/move rises with depth, as expected.

Future Scope

- **NNUE Accumulator and INT8 Inference:** Standard in modern engines; big CPU speedups via SIMD, no retrain needed.
- **Iterative Deepening with Time Controls:** Practical play (per-move/total clocks); industry norm.
- **Cython/C++ Hotspots:** Port performance-critical functions; common engineering path for latency/throughput.
- **Reinforcement Learning:** useful but heavier infrastructure.

References

[1] Sharick, E. [Eddie Sharick]. (n.d.). Creating a Chess Engine in Python [Video playlist]. YouTube. https://youtube.com/playlist?list=PLBwF487qi8MGU81nDGaeNE1EnNEPYWKY_

[2] Ferreira, D. R. (2013). The impact of search depth on chess playing strength. https://web.ist.utl.pt/diogo.ferreira/papers/ferreira13impact.pdf

[3] Stockfish Developers. (n.d.). Stockfish [Computer software]. GitHub. https://github.com/official-stockfish/Stockfish

[4] Hugging Face. (n.d.). The AI community building the future. Hugging Face. https://huggingface.co/

[5] Chessprogramming Wiki. (n.d.). https://www.chessprogramming.org/