Optimizing Chess Engine Training Efficiency via Guided Curriculum Self-Play

CSCE 585: Machine Learning Systems

Vito Spatafora

Milestone P1 — Initial Experiment and Evaluation Setup

Overview

This project implements a hybrid chess engine and reinforcement learning framework designed to explore training efficiency through guided curriculum self-play (Bengio et al., 2009). The system combines a C++ engine, responsible for move generation, rule enforcement, and alpha-beta search, with a Python training loop that handles self-play orchestration, neural network evaluation, and data logging. Communication between the two components is achieved via PyBind11, allowing Python's RL logic to leverage C++ performance.

A baseline neural evaluation model is trained in PyTorch and exported to ONNX for fast inference within the C++ engine. Together, these components form an environment for testing curriculum-based reinforcement learning.

To demonstrate functionality, the engine completed 1,000 self-play games as proof of concept. This run confirmed system stability and revealed many early limitations.

System Setup & Design

The system is composed of two tightly integrated components: a C++ chess engine optimized for efficient self-play generation and a Python-based reinforcement learning module that handles training and evaluation. Together, these elements form a feedback loop where gameplay data informs model updates, and the updated model guides subsequent search behavior.

C++ Engine Architecture:

The chess engine is built around a high-performance bitboard state representation, enabling compact storage of board configurations and efficient bitwise operations for move generation. The decision-making process is governed by a minimax search algorithm enhanced through several classical optimizations:

- Alpha-Beta Pruning: Reduces the number of nodes evaluated in the search tree by discarding branches that cannot influence the final decision, substantially lowering computation time per move.
- Iterative Deepening: Repeatedly deepens the search to progressively refine evaluations while maintaining responsiveness, allowing the engine to return the best move found within time constraints.
- Move Ordering: Prioritizes likely strong moves (such as captures or checks) early in the search, improving the effectiveness of alpha-beta pruning.

- Transposition Tables: Cache previously evaluated positions to avoid redundant calculations when the same position appears via different move sequences.

These optimizations collectively decrease the average time required to complete a full self-play game, making large-scale training feasible by minimizing redundant computations and improving node efficiency.

Python Reinforcement Learning Integration:

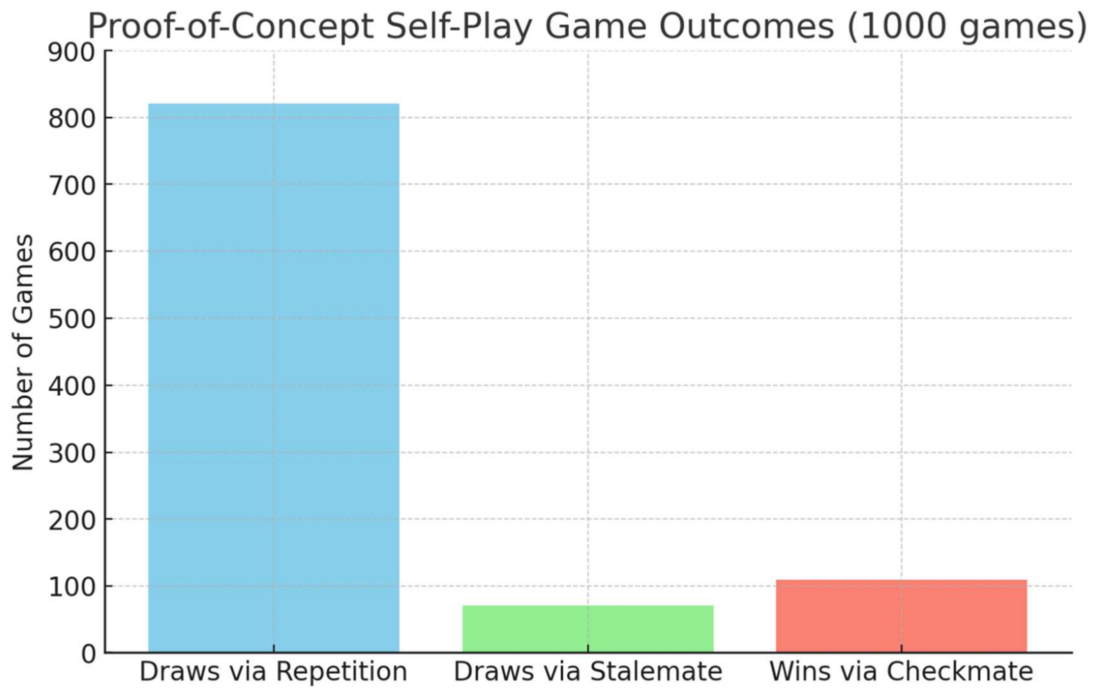
The Python module serves as the learning and orchestration layer. It receives game data—state representations, actions, and outcomes—directly from the C++ engine through PyBind11 bindings, allowing for seamless communication. Using this data, the Python script trains a neural network evaluation function with PyTorch, learning to estimate board position strength from prior experience. Once trained, the model is exported in ONNX format, enabling efficient inference directly within the C++ engine during future self-play sessions. This cyclical design ensures continuous improvement: the engine generates experience through self-play, the Python model learns from it, and the updated network enhances the engine's evaluation accuracy—creating a scalable framework for guided curriculum reinforcement learning in chess.

Neural Network Architecture:

The system employs a Value Network to estimate the expected outcome of a chess position. It accepts a 12-channel 8×8 tensor representing the board, with each channel corresponding to a specific piece type, and processes this input through three sequential 2D convolutional layers (12→64→128→256 channels, 3×3 kernels, ReLU activations) to extract spatial patterns relevant to positional evaluation, such as control, threats, and formations. The output of the convolutional layers is flattened and passed through three fully connected layers (8×8×256→256→64→1) with ReLU activations and a final Tanh, producing a scalar in [-1, 1] representing the position's estimated value for the current player from the perspective of white. Training uses mean squared error loss with the Adam optimizer and supports batch training, including optional weighting for repeated positions. Target values are generated from self-play games with discounted rewards based on the outcome (+1 for win, 0 for draw, -1 for loss) using a discount factor $\gamma = 0.98$. During search, the network evaluates positions to guide move selection.

Preliminary Experiment

As an initial proof-of-concept, I conducted 1,000 self-play games at a search depth of 1 ply to evaluate the stability, efficiency, and general performance of our neural network-based chess evaluation training system:



The outcomes indicated that the majority of games concluded as draws via repetition, with 821 games (82.1%) ending in this manner, while draws via stalemate occurred in 70 games (7.0%), and wins by checkmate were observed in 109 games (10.9%). While this dataset is clearly insufficient to rigorously test our overarching hypothesis, given that effective network training is expected to require tens of thousands of games, these results provide valuable evidence that our system's architecture is functioning as intended. Beyond demonstrating operational stability, the experiment allows for the identification of practical limitations in the current setup, such as tendencies toward repeated draws. It also offers an opportunity to refine both the training pipeline and network structure before committing to more computationally intensive training runs. Overall, this proof-of-concept serves as a foundational validation step, confirming that our design can support large-scale self-play and that the system can generate meaningful training data for subsequent training runs.

Research Trajectory & Current Progress

The trajectory of this research has been guided by a progressive approach to designing and validating a machine learning framework for self-play chess training. At the current stage, I have successfully mapped out a proof-of-concept design for the training loop. This system enables both the training and evaluation of self-play learning within a controlled environment, providing an initial benchmark for understanding the performance and limitations of the framework.

Moving forward, the focus will be on iteratively refining this system. Enhancements will target specific aspects of the training loop to improve efficiency and consistency for the control group, ensuring that the underlying framework is robust and capable of producing meaningful models. The intention is to achieve a reliable baseline in which models can demonstrate competent chess play after a reasonable amount of training.

Once the control system is fully optimized and validated, the research will progress toward the primary objective: investigating the potential benefits of self-play curriculum-based training. This approach involves limiting self-play games to a predefined number of moves, with the goal of accelerating learning while maintaining or improving overall performance. The refined control setup will serve as a benchmark against which the success of the curriculum-based approach can be evaluated, providing a clear point of comparison to assess improvements in efficiency and model performance. By experimenting with various configurations, the aim is to establish a reproducible structure that maximizes both efficiency and effectiveness, ultimately providing insight into more powerful strategies for machine learning-driven chess training.

System Improvements

While the current system provides a functioning framework for self-play training, several obstacles limit its performance and scalability. The primary challenges include model collapse and insufficient training performance. Specifically, the network cannot currently train at sufficiently high search depths due to computational inefficiencies, resulting in poor-quality training data. The combination of limited data quality and the deterministic behavior of the model frequently leads to repeated game sequences, producing a high number of repetitions and ultimately contributing to model collapse.

To address these challenges, several system improvements are proposed:

- Model Architecture Optimization: Redesigning the value network to be more lightweight, akin to architectures used in Stockfish (Nasu, 2018), will reduce computational overhead, allowing for deeper search evaluations and faster training cycles.
- Engine-Level Optimization: Implementing enhancements in the underlying chess engine, such as magic bitboards for more efficient move generation, will further increase search performance and enable the collection of higher-quality game data.

- Replay Buffer Integration: Introducing a prioritized replay buffer (Schaul et al., 2015) stabilizes training by storing and reusing positions across multiple training iterations, reducing the negative impact of repeated sequences and providing more diverse samples to the network.
- Search Depth Adjustment: Dynamically adjusting search depth during self-play will help balance training efficiency with the generation of meaningful positions, ensuring the network receives sufficiently informative and varied data.
- Controlled Randomness in Alpha-Beta Search: Incorporating slight stochasticity at the root move, choosing a move that deviates slightly from the calculated best, can increase game diversity, mitigating repeated game sequences and reducing the likelihood of model collapse.

Collectively, these improvements aim to enhance both the stability and efficiency of the system, facilitating deeper, higher-quality self-play training while mitigating the risks of repetition and collapse inherent in the current deterministic framework.

Reproducibility

At present, the system lacks a mechanism to ensure reproducible results. Randomness in model initialization, combined with stochastic elements in training, prevents the exact replication of experimental outcomes. While the current experiments served primarily as a proof of concept to demonstrate the functionality of the training pipeline, achieving reproducibility will be critical for rigorous evaluation and future comparisons.

To address this, a controlled random seed will need to be implemented across all components of the system. This includes the initialization of network weights and the stochastic elements in training. This reproducibility mechanism will also need to be incorporated into the proposed system improvements, such as replay buffer sampling and randomized root move selection, ensuring that these enhancements produce consistent and replicable results.

References

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. In Proceedings of the 26th annual international conference on machine learning (pp. 41-48).

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv preprint arXiv:1511.05952.

Nasu, Y. (2018). Efficiently Updatable Neural-Network-based Evaluation Function for computer Shogi. Ziosoft Computer Shogi Club.