Self-Play Reinforcement Learning Chess Engine

Progress Report

Vito Spatafora

CSCE 585: Machine Learning
Learning

Establishing a Baseline: Project Objective

The primary objective of this stage was to establish a foundational control system for future, more complex reinforcement learning experiments, specifically those involving curriculum-based training and energy monitoring protocols.

Control Group Foundation

Create a stable, self-play RL system for comparative analysis.

Search

Utilize a standard alpha-beta search algorithm for move selection.

Value Network Integration

Implement a neural network for efficient position evaluation.

This control system serves as the performance and energy benchmark against which all subsequent algorithmic modifications will be measured.

System Architecture: Modular Design

The system employs a tightly integrated, modular design optimized for training speed and execution efficiency, utilizing a three-component structure.

Python Training Loop

The supervisory control layer built on PyTorch for data aggregation and parameter optimization.

C++ Alpha-Beta Engine

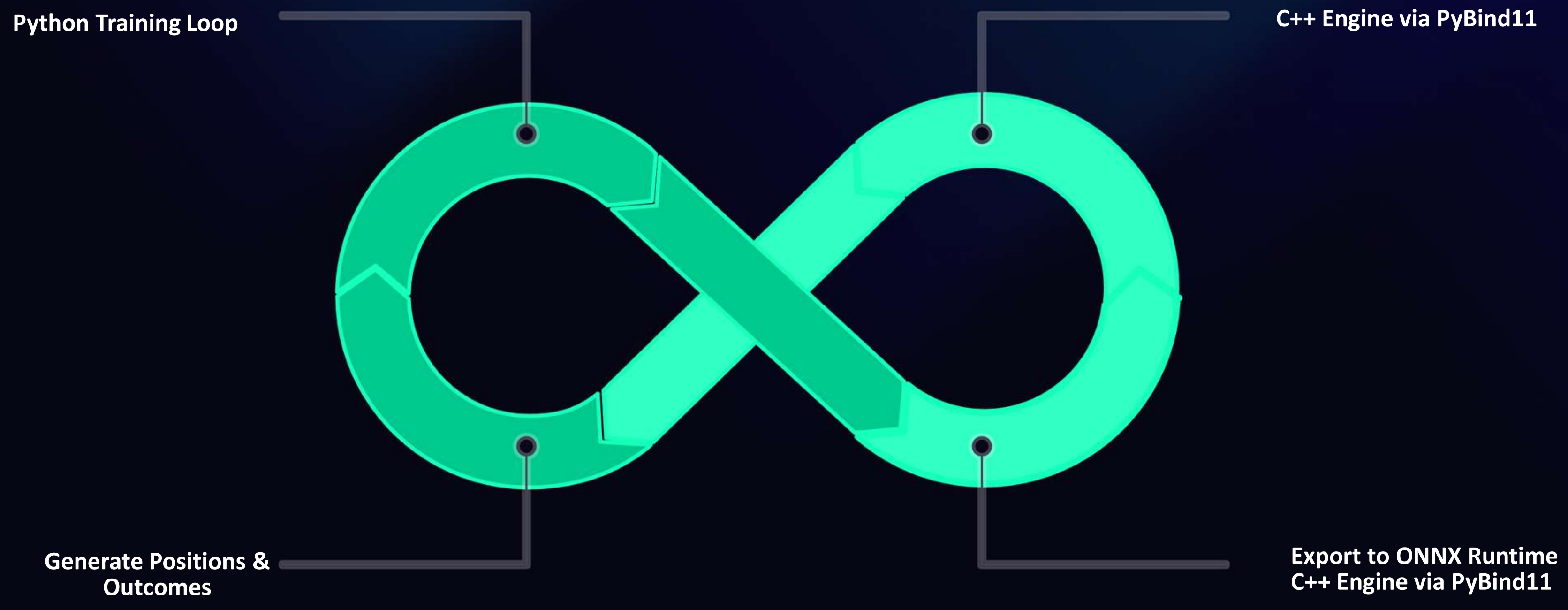
High-performance, low-level engine handling board logic and search operations via the PyBind11 bridge.

ONNX Neural Network

The position evaluation function exported via ONNX runtime for cross-language compatibility and fast inference.

Self-Play Data Flow and Training Cycle

The training mechanism is a closed feedback loop ensuring seamless interaction between the high-level ML framework and the low-level search engine.


Python Training Loop
C++ Engine via PyBind11
Generate Positions & Outcomes
Export to ONNX Runtime C++ Engine via PyBind11

Core Component Details

C++ Alpha-Beta Engine

Primary function for move generation and search. Optimized for speed to minimize self-play game duration.

PyTorch Training Loop

Manages the data pipeline, applying the policy gradient updates and handling discounted reward calculation (*gamma=0.98*).

ONNX Value Network

Standard feed-forward neural network for evaluating board positions. Exported for fast, serialized inference across system components.

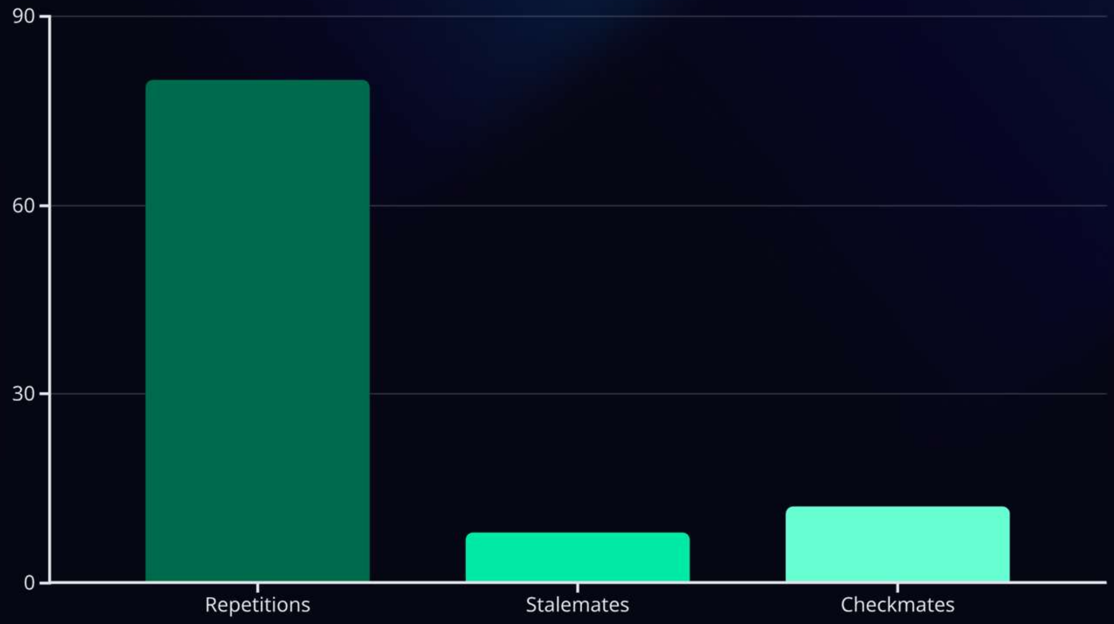
PyBind11 Bridge

Critical interface enabling zero-overhead communication between the Python and C++ environments, crucial for performance.

Control Group Results: Game Outcome Distribution

The initial 1,000 games revealed a significant bias towards repetitive outcomes, reflecting the limitations of the depth-1 deterministic search:

- Draws Via Repetition: 821 or 82.1%
- Draws Via Stalemate: 70 or 7.0%
- Wins Via Checkmate: 109 or 10.9%



Key Finding: 80% draw via repetition

The overwhelming majority of games concluded via draw by repetition. This indicates the engine frequently enters short, deterministic loops when evaluating only one move ahead.

These initial results represent an early proof of concept for the self-play training loop. The current dataset includes 1,000 games, sufficient to validate data flow, reward functions, and convergence tracking. In future iterations, the engine will self-play tens of thousands of games under varied parameters (e.g., learning rate, network depth, curriculum pacing) to achieve statistically robust performance and energy efficiency comparisons.

Challenges and System Limitations

The deterministic self-play setup introduces several risks that must be addressed in subsequent iterations to improve agent quality.

Model Collapse

The deterministic nature of the play promotes early policy collapse, where the agent plays limited, predictable sequences of moves.

Lack of Exploration

Without exploration noise (e.g., *epsilon*-greedy or MCTS), the agent fails to discover superior, non-obvious strategies.

Repetitive Cycles

Search depth 1 heavily favors immediate safety, leading to high-frequency self-loops and subsequent draw by repetition outcomes.

Conclusion & Planned Improvements

The current system successfully establishes a functional baseline (Proof of Concept). Future work will focus on introducing necessary complexity to enhance learning and performance.

1 **Implement Exploration Noise**

Introduce temperature-based exploration to break deterministic cycles and broaden the state-space coverage. This can be done by adding noise to the repetition draws or, more preferably, by adding controlled randomness in the move selection of the engine after evaluation.

2 **Addition of a Replay Buffer**

A replay buffer is a memory module that stores past game states, actions, and outcomes during self-play. Instead of training only on the most recent games, the model samples data from this buffer to learn from a diverse set of past experiences. This increases stability and efficiency

3 **Energy Monitoring Hook**

Integrate system calls for real-time power consumption and thermal tracking during the C++ search phase

4 **Optimizations**

Implementation of Magic Bitboards and other optimization techniques to increase the speed of self-play games and the practicality of higher depth training (Training on depths higher than a ply of one currently takes far too long) will result in in more meaningful data leading to faster and more stable converge