# AI for the Game of Chess: Classical Search and Neural Network Approaches

Your Name
Department of Computer Science
November 2, 2025

## Abstract

This project explores artificial intelligence techniques applied to the game of Chess on an 8x8 board. The goal is to design, train, and evaluate multiple agents including Random, Minimax, Negamax, and Neural Network–assisted search agents. While classical search algorithms rely on deterministic evaluations and handcrafted heuristics, neural network agents attempt to learn positional strength through data-driven methods. We compare their performance, analyze their behaviors, and outline future improvements for deeper search and reinforcement-based self-play.

## 1 Introduction

The objective of this project is to implement AI agents that can play chess intelligently using both classical and machine learning approaches. Chess, with its vast state space and strategic depth, serves as an excellent domain for exploring search algorithms and neural evaluation.

We developed and compared several agents: a Random baseline, a Minimax search agent, a Negamax variant with alpha-beta pruning, and a Neural Network evaluator that replaces handcrafted heuristics with learned board evaluation.

## 2 Project Overview

### 2.1 Game Environment

The environment simulates a standard 8x8 chessboard using the `python-chess` library. All legal rules are implemented, including castling, en passant, and promotion. The system supports human interaction and AI self-play.

### 2.2 Agents Implemented

The following agents were implemented and compared:

- **Random Agent:** Selects valid moves randomly.
- **Minimax Agent:** Uses depth-limited search with heuristic evaluation.
- **Negamax Agent:** Simplified Minimax for zero-sum games, integrated with alpha-beta pruning.
- **Neural Network Evaluator:** CNN-based model estimating positional advantage.

## 3 Training Methodology

### 3.1 Negamax Agent

The Negamax agent searches the game tree recursively and evaluates positions using a scoring function. The recursive update rule is:

$$\text{Negamax}(s, d, \alpha, \beta) = \max_{a \in A(s)} \left[ -\text{Negamax}(s', d-1, -\beta, -\alpha) \right] \tag{1}$$

where:

- $s$ = current state
- $d$ = search depth
- $\alpha, \beta$ = pruning bounds

Alpha-beta pruning eliminates moves that cannot influence the final decision, improving efficiency.

Listing 1: Negamax search implementation snippet

```python
def negamax(board, depth, alpha, beta, color):
    if depth == 0 or board.is_game_over():
        return color * evaluate(board)
    max_eval = -float('inf')
    for move in board.legal_moves:
        board.push(move)
        eval = -negamax(board, depth-1, -beta, -alpha, -color)
        board.pop()
        max_eval = max(max_eval, eval)
        alpha = max(alpha, eval)
        if alpha >= beta:
            break
    return max_eval
```

### 3.2 Neural Network Evaluation

The neural network replaces the handcrafted evaluation function. It maps board states to scalar evaluations indicating the advantage of the current player.

The network is trained using a supervised approach from labeled positions and self-play data. The loss function is the mean squared error between predicted and target evaluations:

$$L = \frac{1}{N} \sum_{i=1}^{N} (V_{\text{target}}^{(i)} - V_{\text{pred}}^{(i)})^2 \tag{2}$$

Listing 2: Neural network evaluation snippet

```python
output = model(board_state)
target = value_target
loss = mse_loss(output, target)
loss.backward()
optimizer.step()
```

## 4 Implementation Details

### 4.1 Environment Setup

Listing 3: Chess environment loop

```python
import chess
from agents import NegamaxAgent, NNAgent

board = chess.Board()
agent = NegamaxAgent(depth=3)
while not board.is_game_over():
    move = agent.select_move(board)
    board.push(move)
print(board.result())
```

### 4.2 Agent Architecture Diagram

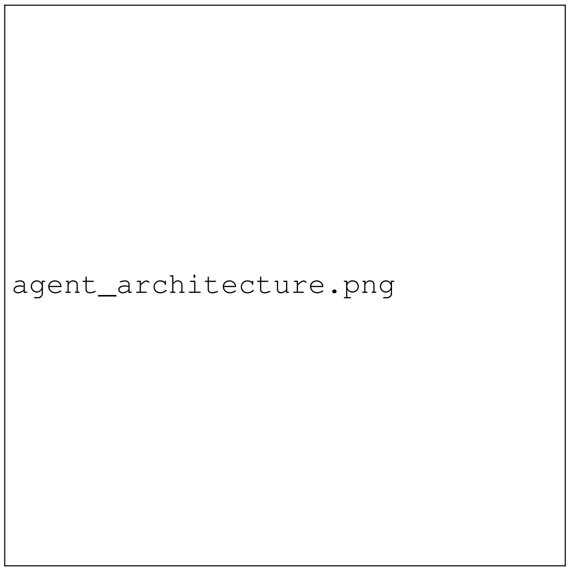

Figure 1: Architecture of Negamax + Neural Network agent.

## 5 Results and Evaluation

### 5.1 Training Performance

Training performance curves were recorded for the neural network's loss and evaluation accuracy.

Figure 2: Neural network training loss over epochs.

### 5.2 Benchmarking Plan

Benchmarking compares each agent on the following criteria:

- Win rate against baseline agents
- Average evaluation accuracy
- Time per move (efficiency)

Table 1: Benchmarking results (to be filled after evaluation).

| **Agent** | **Win vs Random** | **Win vs Minimax** | **Avg Reward** |
|---|---|---|---|
| Random | - | - | - |
| Minimax | - | - | - |
| Negamax | - | - | - |
| NN Agent | - | - | - |

## 6 Discussion

- The Negamax agent performs well at moderate depths but struggles in tactical endgames without deeper search.
- The neural network evaluator captures positional features such as center control and king safety.
- Training quality strongly depends on data diversity and self-play consistency.
- Future work includes reinforcement learning fine-tuning and hybrid MCTS integration.

## 7 Conclusion

This project developed and analyzed multiple AI agents for chess, integrating both classical and learning-based strategies. Initial results demonstrate that hybrid approaches—combining Negamax search with neural evaluation—offer promising improvements over purely classical methods.

## References

- Silver, D. et al. (2017). *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.*
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction.*
- `python-chess` documentation: `https://python-chess.readthedocs.io`
- PyTorch documentation: `https://pytorch.org`