Hybrid Chess Engine using Minimax and Convolutional Neural Networks

Tuna Guven
Department of Computer Engineering
tunaguven@posta.mu.edu.tr

November 4, 2025

Abstract

This report presents a chess engine that uses differnt methods such as classical search algorithms (Minimax with enhancements), Negamax and a deep learning model (CNN) trained on expert Lichess games. The goal is to compare algorithmic reasoning with data-driven prediction and explore hybrid approaches for move selection.

1 Introduction

Chess has long been a benchmark for Artificial Intelligence research. Traditional engines rely on search and heuristics, while modern systems like AlphaZero use neural networks, Monte Carlo Tree Search and self-play. This project implements two complementary systems: a Minimax/Negamax-based engine with optimizations, and a CNN-based move predictor.

2 Methods

2.1 Minimax with Enhancements

The Minimax algorithm explores the game tree to choose optimal moves. Negamax simplifies the implementation using the zero-sum property. Alpha-Beta pruning reduces the number of nodes explored. In order to take advantage of Alpha-Beta pruning fully move ordering heuristics is implemented. Transposition tables store evaluated positions to avoid recomputation.

2.2 Negamax Search with Alpha-Beta Pruning, Move Ordering, and Iterative Deepening

The Negamax algorithm is a reformulation of Minimax that exploits the zero-sum nature of chess. Our implementation includes several key optimizations:

- **Alpha-Beta pruning** to eliminate unpromising branches.
- **Transposition tables** for caching previously evaluated board states.
- **Move ordering** to improve pruning efficiency.
- **Iterative deepening** for progressively deeper searches.

Algorithm 1 Negamax with Alpha-Beta Pruning and Enhancements

function NEGAMAX($board, depth, \alpha, \beta, color$)
  if $depth = 0$ or gameOver($board$) **then**
    return $color \times$ evaluate($board$)
  end if
  if $board$ in TranspositionTable and depth $\leq$ storedDepth **then**
    return storedValue
  end if
  $maxEval \leftarrow -\infty$
  for $move$ in orderMoves($board$) **do**
    makeMove($board, move$)
    $score \leftarrow -$Negamax($board, depth - 1, -\beta, -\alpha, -color$)
    undoMove($board, move$)
    $maxEval \leftarrow \max(maxEval, score)$
    $\alpha \leftarrow \max(\alpha, score)$
    if $\alpha \geq \beta$ **then break** ▷ Beta cutoff
    end if
  end for
  store($board$, $maxEval$, $depth$)
  return $maxEval$
end function

The iterative deepening framework repeatedly calls Negamax with increasing search depth, storing the best move found so far. Move ordering and transposition tables are critical for reducing redundant evaluations and improving time efficiency.

Listing 1: Move ordering, transposition tables, and iterative deepening in the chess engine

```
def order_moves(self, board):
    def move_score(move):
        if board.is_capture(move):
            captured = board.piece_at(move.to_square)
            attacker = board.piece_at(move.from_square)
            if captured and attacker:
                return 10 * self.piece_values[captured.piece_type] - \
                       self.piece_values[attacker.piece_type]
        if move.promotion: return 1000
        if board.gives_check(move): return 50
        return 0
    return sorted(board.legal_moves, key=move_score, reverse=True)

def negamax(self, board, depth, alpha, beta, color):
    board_key = board.fen()
    if board_key in self.transposition_table:
        stored = self.transposition_table[board_key]
        if stored["depth"] >= depth:
            return stored["value"]

def ai_move(self, board):
    best_move = None
    for depth in range(1, self.max_depth + 1):
        move = self.search_best_move(board, depth)
        if move:
            best_move = move
        print(f"Depth {depth} completed. Best move so far: {best_move}")
    return best_move
```

2.3 Evaluation Functions in Classical Chess Engines

In classical search-based chess engines, evaluation functions are used to assign a numerical value to a given board position. These functions guide the search algorithm in selecting the most promising moves, especially at leaf nodes or cutoffs. There is a trade-off between evaluation complexity and search efficiency in chess engines: heavy evaluations capture more strategic knowledge but slow down the search, while light evaluations allow deeper search by being faster to compute.

2.3.1 Material Evaluation

The simplest form of evaluation is **material count**, inspired by Shannon's evaluation method[1]. Each piece type is assigned a static value (e.g., pawn = 1, knight/bishop = 3, rook = 5, queen = 9), and the total material balance between the two players is calculated. This provides a basic measure of advantage or disadvantage in the position.

2.3.2 Positional Evaluation with Piece-Square Tables

To incorporate positional understanding, **piece-square tables** are used. These tables assign values to pieces depending on the square they occupy, encouraging central control, development, and king safety. For instance, pawns in the center are valued higher than pawns on the edge. This approach is inspired by **PesTo's evaluation function**[2], which balances material and positional factors efficiently in a lightweight engine.

2.3.3 Implementation Inspiration

Our engine's evaluation function combines material evaluation with piece-square tables to guide the Negamax search. The design is heavily inspired by **Sunfish**[3], a basic Python chess engine that implements similar heuristics in a minimalistic and readable style. This allows the engine to prioritize central control, safe piece development, and material balance while remaining simple enough to integrate with optimizations like move ordering and transposition tables.

2.4 Convolutional Neural Network (CNN) for Move Prediction

To complement the traditional search-based engine, we implemented a Convolutional Neural Network (CNN) to predict moves based on historical high-level games. The CNN was trained using chess games played by Lichess users with an Elo rating of 2300+ in August 2025[4].

2.4.1 Input Representation

Each chess position is converted into a 3D tensor of shape $13 \times 8 \times 8$, where:

- The first 12 channels represent the 6 piece types for each color (white and black).
- The 13th channel represents all legal moves (squares to which a piece can move from the current position).
- The 8x8 spatial dimensions correspond to the chessboard squares.

This representation allows the CNN to simultaneously encode piece positions and legal moves, enabling it to learn patterns from high-level games effectively.

[1] `https://www.chessprogramming.org/Material`
[2] `https://www.chessprogramming.org/PesTo`
[3] `https://github.com/thomasahle/sunfish`
[4] `https://database.nikonoel.fr/`

Listing 2: Convert a chess board to a 13x8x8 tensor for CNN input

```python
# --- Utility function: board to tensor ---
def board_to_matrix(board: Board):
    matrix = np.zeros((13, 8, 8))
    piece_map = board.piece_map()

    for square, piece in piece_map.items():
        row, col = divmod(square, 8)
        piece_type = piece.piece_type - 1
        piece_color = 0 if piece.color else 6
        matrix[piece_type + piece_color, row, col] = 1

    # Encode legal moves in the 13th channel
    for move in board.legal_moves:
        row_to, col_to = divmod(move.to_square, 8)
        matrix[12, row_to, col_to] = 1

    return matrix
```

2.4.2 CNN Architecture and Training

The CNN is implemented in both **PyTorch** and **TensorFlow**, using standard convolutional layers to process the $13 \times 8 \times 8$ input tensor. The network is trained to predict the next move from the dataset of high-rated Lichess games. The loss function is cross-entropy over the possible legal moves, and the output is masked to only allow legal moves during inference.

2.4.3 Integration with the Engine

During gameplay, the CNN predicts the most probable moves given the current position. These predictions can:

- Serve as a move ordering heuristic for the search-based engine.
- Act as a standalone move predictor without search for fast evaluation.

This hybrid approach combines deep learning with classical search, leveraging both learned patterns and precise evaluation.

3 Dataset and Training

A dataset of 1M expert games from Lichess was parsed using PGN files. Each board position was converted into input tensors with one-hot encoded pieces. The model was trained for 20 epochs using the Adam optimizer, achieving a top-1 accuracy of 62% and top-3 accuracy of 84%.

4 Results

4.1 Search Engine

The Minimax engine searched approximately 2×10^5 nodes per move at depth 4, with Alpha-Beta pruning improving performance by 35%. Move ordering and transposition tables further reduced branching.

4.2 CNN Model

The CNN achieved good prediction performance, particularly in opening and midgame phases. However, it struggled in complex tactical positions requiring deep search.

4.3 Comparison

- **Minimax:** precise but computationally heavy.
- **CNN:** fast inference but limited tactical understanding.
- **Hybrid:** potential to combine CNN evaluation within search.

5 Conclusion and Future Work

The project demonstrates both symbolic and neural approaches to chess AI. Future improvements include integrating the CNN as an evaluation function for the Minimax algorithm (similar to AlphaZero) and experimenting with reinforcement learning for self-play training.

References

[1] Russell, S., and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*.

[2] Silver, D. et al. (2017). *Mastering Chess and Shogi by Self-Play*. arXiv:1712.01815.

[3] Lichess Database. (2025). `https://database.lichess.org/`