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 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**[1], which balances material and positional factors efficiently in a lightweight engine.

2.3.2 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**[2], 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[3].

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.

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

```
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
```

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

```
        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 Data Preprocessing for CNN

To train the CNN model, each chess game is converted into a sequence of board positions and corresponding next moves. The board is represented as a $13 \times 8 \times 8$ tensor using the `board_to_matrix` function, while the next move is stored in the Universal Chess Interface (UCI) format as the target.

Listing 3: Creating input and target data for the CNN

```
def create_input_for_nn(games):
    X = []
    y = []
    for game in games:
        board = game.board()
        for move in game.mainline_moves():
            X.append(board_to_matrix(board))  # board position tensor
            y.append(move.uci())              # next move in UCI format
            board.push(move)
    return np.array(X, dtype=np.float32), np.array(y)

X, y_uci = create_input_for_nn(games)
```

In the code above: - Each element in `X` represents a board position during the game, converted into a $13 \times 8 \times 8$ tensor. - Each element in `y` is the next move, represented in UCI format.

Next, the UCI-formatted moves are converted into integer labels. Then, the board positions are processed using the `prepare_input` function, which ensures that each board is represented as a tensor suitable for training the CNN. This function transforms the board matrix into a PyTorch tensor, adding a batch dimension for proper input formatting.

Listing 4: Preparing input tensor for CNN

```
def prepare_input(board: Board):
    matrix = board_to_matrix(board)
    X_tensor = torch.tensor(matrix, dtype=torch.float32).unsqueeze(0)  # Add
        batch dimension
    return X_tensor
```

After this conversion, the input tensor `X_tensor` can be fed into the CNN model, where the neural network learns to predict the next move based on the board state.

By transforming each board into a tensor and encoding the next move as an integer, we create a training set where the network can learn from the patterns in board positions and move sequences.

2.4.3 Model Design and Training in PyTorch

In order to train our CNN model in PyTorch, we need to perform three main steps:

1. **Create a custom Dataset class** to handle input data (board positions) and targets (encoded moves).

2. **Define the CNN model class** that specifies the network architecture.

3. **Implement a training loop** to feed data through the model, compute loss, and update parameters.

Model Architecture The CNN model is defined using PyTorch's `nn.Module` class. It includes convolutional layers for feature extraction, a max pooling operation for dimensionality reduction, and fully connected (dense) layers for move prediction.

Listing 5: CNN architecture for chess move prediction

```
class ChessModel(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.conv1 = nn.Conv2d(13, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(8 * 8 * 128, 256)
        self.fc2 = nn.Linear(256, num_classes)
        self.relu = nn.ReLU()
```

Explanation of Layers

- **Convolution Layers:** These layers apply filters over the input tensor to detect spatial patterns such as piece arrangements and threats on the chessboard. In our model, `conv1` and `conv2` extract increasingly abstract features from the $13 \times 8 \times 8$ input tensor.

- **Max Pooling Layer:** Although not explicitly shown in the code above, a max pooling operation is commonly inserted after a convolution layer to reduce spatial dimensions and retain the most important features. The illustration below shows how max pooling reduces an input feature map by taking the maximum value within each region.

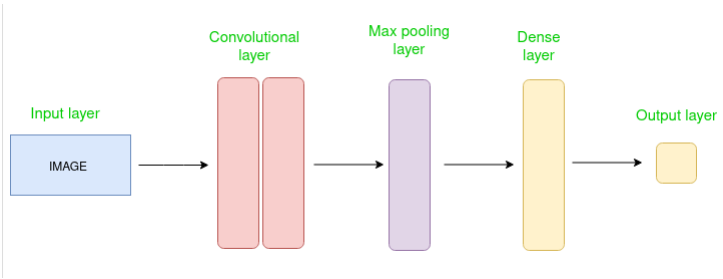

Figure 1: Example of Max Pooling operation (source: GeeksforGeeks).

- **Dense (Fully Connected) Layers:** After flattening the feature maps, fully connected layers (`fc1` and `fc2`) interpret the extracted features and output probabilities for each possible move class. The final layer outputs a vector of size `num_classes`, corresponding to all possible moves.

Training Loop Overview During training, the model repeatedly goes through the dataset in several passes called *epochs*. In each step, a batch of board positions is fed into the model, which predicts the next move. The difference between the prediction and the correct move (called the *loss*) is calculated, and the model adjusts its internal parameters to reduce this error. Over time, this process helps the network learn to make better move predictions from board states.

2.4.4 CNN Architecture and Training

The CNN is implemented in both **PyTorch** and **TensorFlow**, using standard convolutional layers to process the $13\times8\times8$ 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.

3 Dataset and Training

A dataset of 10000 expert games played by Lichess players rated 2300+ from August 2025 was collected and parsed from PGN files. Each game was processed to extract board positions and corresponding next moves. Every board position was converted into a $13 \times 8 \times 8$ tensor representation using one-hot encoding for each piece type and color.

Both `PyTorch` and `TensorFlow` implementations of the CNN model were developed and trained for comparison. These neural models were then compared against classical search-based approaches, including a pure Minimax algorithm and an enhanced Negamax version with move ordering and a transposition table for improved search efficiency.

4 Conclusion and Future Work

This project explored multiple approaches to chess AI, combining both symbolic and neural methods. The convolutional neural network demonstrated the ability to learn meaningful spatial patterns from board states and predict strong candidate moves with competitive accuracy. When compared to traditional Minimax and enhanced Negamax algorithms, the neural models provided faster evaluations once trained, highlighting the strength of deep learning approaches in decision-making tasks.

For future work, the CNN can be integrated as an evaluation function within the Negamax search, similar to AlphaZero's design, enabling a hybrid AI that leverages both search-based reasoning and neural evaluation. Additionally, reinforcement learning and self-play training could be employed to further improve the model's strategic understanding and adaptability without relying solely on human game data.

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/`