Using text information to train a chess-playing agent

(Wykorzystanie informacji tekstowych do wytrenowania agenta grającego w szachy)

Kamil Michalak

Praca magisterska

Promotor: dr Paweł Rychlikowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

7 lipca 2024

Abstract

The goal of this thesis is to train chess-playing agents using text data. In the initial chapters, we analyze existing methods for creating chess AI, utilizing classical algorithms, neural networks, and natural language processing. Next, we present our own research, in which we improved some of the described methods. Our research consists of two parts. In the first part, we implemented an agent that learned to play based on natural language commentary on chess games. In the second part, we trained GPT language models to model chess games and implemented an agent that plays by generating successive tokens representing moves. We achieved satisfactory results in both parts.

Celem niniejszej pracy jest wytrenowanie agentów grających w szachy z wykorzystaniem danych tekstowych. W początkowych rozdziałach analizujemy istniejące metody tworzenia szachowego AI wykorzystujące klasyczne algorytmy, sieci neuronowe i przetwarzanie języka naturalnego. Następnie przedstawiamy własne badania, w których poprawiliśmy niektóre z opisanych metod. Nasze badania składają się z dwóch części. W pierwszej z nich zaimplementowaliśmy agenta, który nauczył się grać na podstawie komentarzy do partii szachowych w języku naturalnym. W drugiej wytrenowaliśmy modele językowe GPT do modelowania gier szachowych i zaimplementowaliśmy agenta grającego poprzez generowanie kolejnych tokenów reprezentujących ruchy. W obu tych częściach uzyskaliśmy satysfakcjonujące wyniki.

Contents

Chapter 1

Introduction

Research on chess-playing artificial intelligence is a very popular branch of AI research with a long history. First chess program that can play the entire game, *Turochamp*, was created in 1948 by Alan Turing and David Champernowne. It was never implemented on any machine because the algorithm was too complex for the early computers.

In 1997, IBM's *Deep Blue* defeated chess world champion Garry Kasparov in a 6-game match $3.5 - 2.5$, becoming the first computer to defeat a grandmaster in a match. Current chess engines are much more powerful than the strongest human player. In 2019, Stockfish, the most popular chess engine, achieved Elo[1] rating of 3438, while the rating of the highest ranked player, Magnus Carlsen, was (and still is) over 2800.

Classic chess engines, like Stockfish before 2020, are based on search algorithms, such as minimax and its variations, and handcrafted functions to evaluate a position. In recent years, a development of deep learning led to the creation of the new type of engines that use neural networks. The most prominent example is IBM's AlphaZero, developed in 2017, that combines neural networks with Monte Carlo Tree Search and reinforcement learning. AlphaZero defeated Stockfish 8, the strongest engine at that time, in a match of 100 games, winning 28 of them and drawing the remaining 72.

1.1 Goal and scope of the thesis

The are two goals of this thesis. The first is to examine the existing methods for creating a chess-playing AI. The second is to create chess agents, using neural networks, natural language processing (NLP) and online text data about chess.

The thesis consists of 5 main chapters. Chapters 2-4 examine the existing chess AI, chapters 5-6 describe our research.

[1] `https://en.wikipedia.org/wiki/Elo_rating_system`

In Chapter 2, we describe classic algorithms for AI in chess and other board games: minimax and Monte Carlo tree search. For minimax, we explain in more detail how it is used in chess engines, in particular Stockfish.

In Chapter 3, we examine research on training neural networks to play chess. These research include AlphaZero (Section 3.1), improving Stockfish with neural network (Section 3.4) and Maia engine, trained to predict moves played by human (Section 3.5).

Chapter 4 describes research on using NLP techniques to train chess agents. First part is the description of SentiMATE [1]: engine that learned from chess comments in natural language (Section 4.1). Second part (Section 4.2) describes research on training language models on text-archived chess games and using them to play by generating subsequent moves.

In Chapter 5 we describe our own, improved version of SentiMATE. We show that changes we made resulted in a stronger agent.

In Chapter 5 we present our research on modeling chess games with GPT language model. We introduce a new tokenization that seems to be more efficient than those presented in previous papers. We show that small GPT models can predict human moves with high accuracy. We also present a new training method to get a stronger agent using the same training data.

Chapter 2

AI in Board Games

This chapter describes search algorithms that form the basis for programs for chess and other board games. Section 2.1 provides a formal definition of a game used in the subsequent sections. Section 2.2 describes minimax – algorithm used in most historical and modern chess engines. Then, Section 2.3 explains the Monte Carlo Tree Search – a different algorithm that achieved very high performance in recent years.

2.1 Zero-sum games

Chess is a two-player, zero-sum game, which means that the loss of one player is the other player's gain. Both players choose their moves with the aim of maximizing their own chances of winning. By doing so they minimize at the same time the winning chances of their opponent. A game can be formally defined with the following elements:

- S_0: The **initial state**.
- To-Move(s): The player whose turn it is to move in state s.
- Actions(s): The set of legal moves in state s.
- Result(s, a): The **transition model**, which defines the state resulting from taking action a in state s.
- Is-Terminal(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- Utility(s, p): A **utility function**, which defines the final numeric value to player p when the game ends in terminal state s. In chess, the outcome is a win, loss, or draw, with tournament values 1, 0, or 1/2. For chess engines, it's more convenient to use values of $+1$, 0, -1 or $+\infty$, 0, $-\infty$

2.2 Minimax Search

Minimax algorithm works as follows. Given the state of the game it considers all possible actions for the current player (we call him MAX, since he wants to maximize game outcome). Then, for each move, it considers all possible replies for second player (MIN). Algorithm recursively search game tree until it reaches all terminal states. Utility function compute value of each terminal state, these results are propagated back to the root node. At each node, the algorithm chooses the minimum or maximum value of child nodes, depending on which player's turn it is. The minimax value can be described with the following formula:

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in \textsc{actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in \textsc{actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

Algorithm 2.1 presents pseudocode for choosing best move with minimax. Figure 2.1 shows an example search tree of the algorithm.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$ The exponential complexity makes MINIMAX impractical for complex games, like chess, where the branching factor is about 35 and the average game has depth around 80 ply (name for one move of one player in game theory), which gives $35^{80} \approx 10^{123}$ nodes in the game tree, impossible to search in a reasonable time. That's why chess engines cutoff search early and apply heuristic **evaluation function** to positions. The formula for this limited-depth minimax's value can look as follows:

$$\text{H-}\textsc{Minimax}(s,d) = \begin{cases} \textsc{Eval}(s, \textsc{max}) & \text{if } \textsc{Is-Cutoff}(s,d) \\ \max_{a \in \textsc{actions}(s)} \text{H-}\textsc{Minimax}(\textsc{Result}(s,a), d+1) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in \textsc{actions}(s)} \text{H-}\textsc{Minimax}(\textsc{Result}(s,a), d+1) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

Terminal test is replaced replaced by **cutoff test**, which returns true for terminal states, and otherwise decides whether to stop searching, based on the search depth and any property of the state that it chooses to consider.

2.2.1 Evaluation function

A heuristic evaluation function $\textsc{Eval}(s,p)$ returns an estimate of the expected utility of state s to player p. For terminal states, it must be that $\textsc{Eval}(s,p) = \textsc{Utility}(s,p)$ and for nonterminal states, the evaluation must be somewhere between a loss and a win: $\textsc{Utility}(loss,p) \leq \textsc{Eval}(s,p) \leq \textsc{Utility}(win,p)$. The EVAL function

Algorithm 2.1 Minimax search

```
function MINIMAX-BEST-MOVE(state)
    player ← TO-MOVE(state)
    value, move ← MAX-VALUE(state)
    return move

function MAX-VALUE(state)
    if ISTERMINAL(state) then
        return UTILITY(state), null
    v_max, move ← −∞, null
    for each a in ACTIONS(state) do
        v, a' ← MIN-VALUE(RESULT(state, a))
        if v > v_max then
            v_max, move ← v, a
    return v_max, move

function MIN-VALUE(state)
    if ISTERMINAL(state) then
        return UTILITY(state, player), null
    v_min, move ← ∞, null
    for each a in ACTIONS(state) do
        v, a' ← MAX-VALUE(RESULT(state, a))
        if v < v_min then
            v_min, move ← v, a
    return v_min, move
```

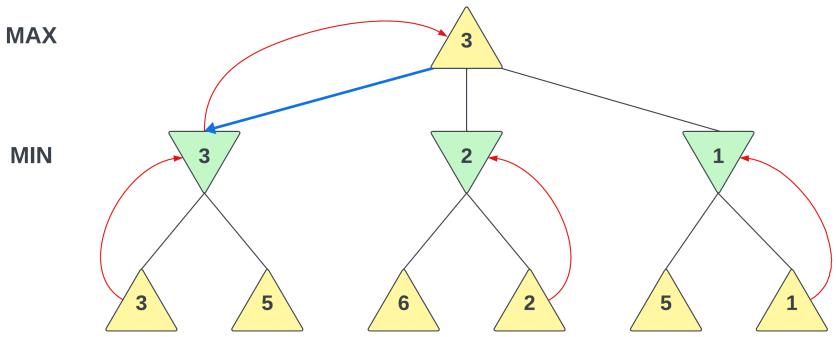


Figure 2.1: Minimax search tree

should compute quickly as it will be called many times during the search process. It should also be strongly correlated with actual chances of winning: if state s_1 is better than s_2 for player p then $\textsc{Eval}(s_1, p) > \textsc{Eval}(s_2, p)$ should hold.

Most chess engines calculate a value of the position by combining the values of various features. Some of the most common features are:

- **Material**: A material advantage gives the player more opportunities to play than his opponent, it is easier for him to attack and defend. Each piece is assigned a numerical value based on its relative strength, the most common assignment is that each pawn is worth 1 point, a knight or bishop: 3, a rook 5, and a queen 9.
- **Mobility**: The mobility of a piece is determined by how many squares it can move to from its current position. The more squares a piece can move to, the more mobile and usually powerful it is.
- **King safety**: Exposed King means it is easier to get checkmate. The engine evaluates the safety of the king by looking at factors such as pawn cover, open files, and the presence of opponent's pieces.
- **Pawn structure**: In general, pawn chains are considered to be good structure because they defend one another and are more difficult to capture. On the other hand, isolated or doubled pawns (two pawns of the same color on the same file) are usually weaknesses.
- **Mobility and control of the center**: The more mobile a piece is, the more useful it might potentially be. The engine assesses how actively each player's pieces can move and influence the board. Pieces with more potential moves are considered stronger. Control of the center is often valued, as pieces in the center can access more squares and exert influence over the board.
- **Development**: An engine prefers positions where the pieces are developed efficiently. Pieces that are still on their starting squares or are not well-connected might be considered less valuable.

Some chess engines also use **endgame tablebases** – databases of precalculated endgame positions. They enable to instatly determine:

- whether the position is winning, losing, or drawn with perfect play from both sides,
- how many moves it will take to checkmate the losing side if the position is not drawn,
- the best move in the position for both sides.

For example, Stockfish uses **7-piece Syzygy tablebase** [2], which contains all possible positions with 7 pieces or less. If that kind of position occurs during searching, it can be immediately evaluate by retrieving value from the table without the need for deeper searching.

Evaluation function calculates features values, which can be added up to obtain evaluation of the position. This can be expressed as weighted linear function:

$$\textsc{Eval}(s) = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n = \sum_{i=1}^{n} w_i f_i$$

The problem with this approach is the assumption that features are independent of each other, which is not the case. For example, during the middle game, when there are still many pieces on board, king should be kept in a safe place, but in the endgame with only kings and pawns, king should be more active. For this reason modern chess engines also use nonlinear combinations of features to capture dependencies between them.

Listing 2.1 shows the main part of the Stockfish's evaluation function. According to the Stockfish Evaluation Guide [3], `main_evaluation` is used in general case when no specialized evaluation or tablebase evaluation is available. Position score is computed by combining middle game and end game evaluation. The proportion between the two evaluation scores, computed by `phase` function, depends on the amount of material left on the board.

The numerical value calculated by the evaluation function indicates which side has an advantage. The score is typically expressed as a positive number for white and a negative number for black. The higher the score, the greater the advantage. Stockfish evaluation value of x means that a player's advantage is equivalent to material advantage of x pawns. If the evaluation is $+1$ in favor of White, it means that White is considered to have an advantage of a full pawn. Conversely, if the evaluation is -0.5, it means that Black has a slight advantage of half a pawn.

The evaluation function should be applied only to positions that are **quiescent** – that is, positions in which there is no pending move that would strongly change the evaluation (such as a capturing the queen). For nonquiescent positions the Is-Cutoff returns false, and the search search continues until quiescent position is reached. This extra quiescence search is usually restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

2.2.2 Alpha-beta pruning

During searching, minimax has to visit a lot of nodes that are irrelevant to the algorithm outcome. These nodes can be skipped using **alpha-beta prunning**. Figure 2.2 shows how the alpha-beta search works.

```
function main_evaluation(pos) {
  var mg = middle_game_evaluation(pos);
  var eg = end_game_evaluation(pos);
  var p = phase(pos), rule50 = rule50(pos);
  eg = eg * scale_factor(pos, eg) / 64;
  var v = (((mg * p + ((eg * (128 - p)) << 0)) / 128) << 0);
  if (arguments.length == 1) v = ((v / 16) << 0) * 16;
  v += tempo(pos);
  v = (v * (100 - rule50) / 100) << 0;
  return v;
}
function middle_game_evaluation(pos, nowinnable) {
  var v = 0;
  v += piece_value_mg(pos) - piece_value_mg(colorflip(pos));
  v += psqt_mg(pos) - psqt_mg(colorflip(pos));
  v += imbalance_total(pos);
  v += pawns_mg(pos) - pawns_mg(colorflip(pos));
  v += pieces_mg(pos) - pieces_mg(colorflip(pos));
  v += mobility_mg(pos) - mobility_mg(colorflip(pos));
  v += threats_mg(pos) - threats_mg(colorflip(pos));
  v += passed_mg(pos) - passed_mg(colorflip(pos));
  v += space(pos) - space(colorflip(pos));
  v += king_mg(pos) - king_mg(colorflip(pos));
  if (!nowinnable) v += winnable_total_mg(pos, v);
  return v;
}
function end_game_evaluation(pos, nowinnable) {
  var v = 0;
  v += piece_value_eg(pos) - piece_value_eg(colorflip(pos));
  v += psqt_eg(pos) - psqt_eg(colorflip(pos));
  v += imbalance_total(pos);
  v += pawns_eg(pos) - pawns_eg(colorflip(pos));
  v += pieces_eg(pos) - pieces_eg(colorflip(pos));
  v += mobility_eg(pos) - mobility_eg(colorflip(pos));
  v += threats_eg(pos) - threats_eg(colorflip(pos));
  v += passed_eg(pos) - passed_eg(colorflip(pos));
  v += king_eg(pos) - king_eg(colorflip(pos));
  if (!nowinnable) v += winnable_total_eg(pos, v);
  return v;
}
function phase(pos) {
  var midgameLimit = 15258, endgameLimit  = 3915;
  var npm = non_pawn_material(pos) + non_pawn_material(colorflip(pos));
  npm = Math.max(endgameLimit, Math.min(npm, midgameLimit));
  return (((npm - endgameLimit) * 128) / (midgameLimit - endgameLimit)) << 0;
}
```

Listing 2.1: Stockfish evaluation

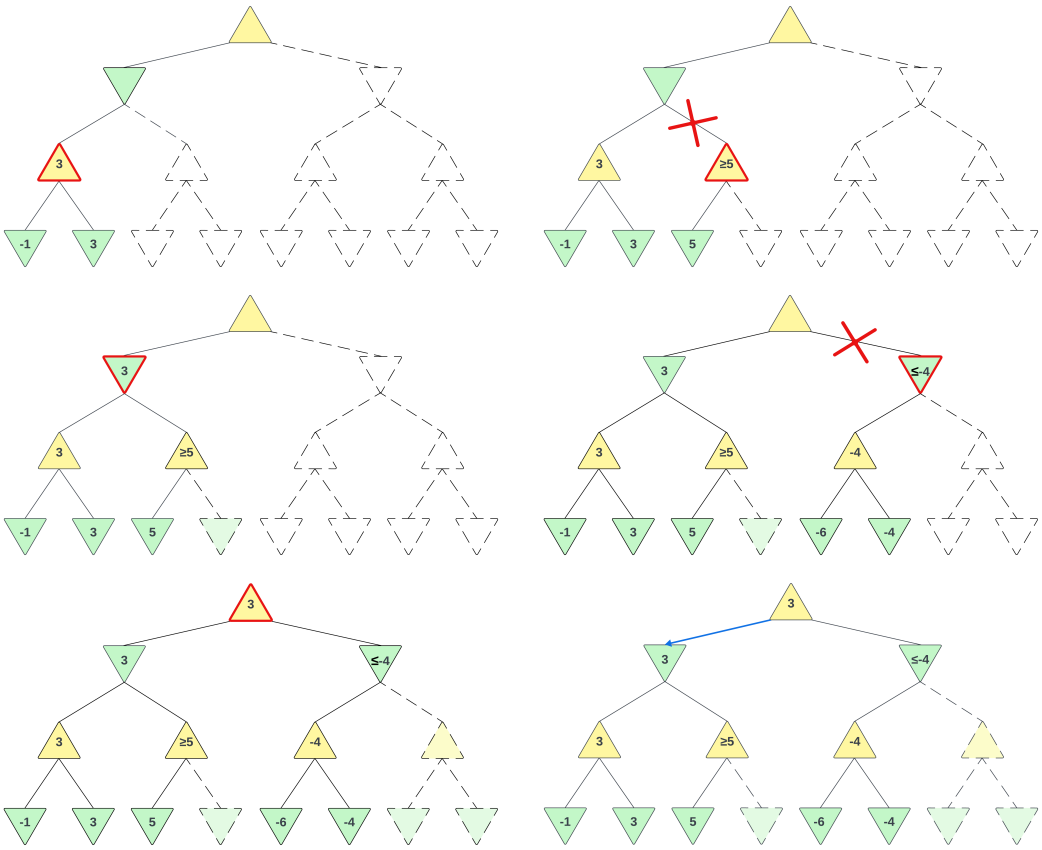


Figure 2.2: Steps of alpha-beta search. Red mark shows currently visited node. Crossed out edge means that the MIN/MAX player won't choose this move, because it is already worse than the alternative, so the remaining part of the subtree can be skipped.

Alpha-beta search is shown in Algorithm 2.2. Compared to the standard minimax, it has 2 extra parameters: α and β that describe bounds on the backed-up values that appear anywhere along the path.

- α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX
- β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. With optimal move ordering (from best to worst move), alpha-beta's complexity would be $O(b^{m/2})$, instead of $O(b^m)$ for standard minimax. This means that the effective branching factor becomes $\sqrt{b}$ instead of b — for chess, around 6 instead of 35. Alpha–beta with perfect move ordering can solve a tree roughly twice as deep as minimax in the same amount of time. It is impossible to achieve – otherwise ordering function could be used to play optimal game – but in chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gives the result within about a factor of 2 of the optimal solution.

Common technique for move ordering is to use **iterative deepening**, which works as follows. First, the algorithm searches one ply deep and record the rating of moves based on their evaluations. Then it searches one ply deeper, using the previous rating to inform move ordering, and so on.

Another common way to speed up search is to use **transposition table**. In chess, during the search, algorithm reaches the same position multiple times because of **transpositions** – different sequences of moves which lead to the same position (e.g. 1. e4 e5 2. Nf3 Nc6 and 1. Nf3 Nc6 2. e4 e5 from initial position). Transposition table caches positions reached during the search with their evaluation scores. If the same position is reached later, its result is read from the table without deeper searching from this node.

2.3 Monte Carlo Tree Search

Alpha-beta search achieved very good results in chess, but it doesn't perform that well in some other games, such as Go. Go has two traits, which show the weaknesses of alpha-beta search. First, it has a high branching factor, around 250 compared

Algorithm 2.2 Alpha-Beta prunning

```
function ALPHA-BETA-BEST-MOVE(state)
    player ← TO-MOVE(state)
    value, move ← MAX-VALUE(state, 0, −∞, ∞)
    return move

function MAX-VALUE(state, depth, α, β)
    if IS-CUTOFF(state, depth) then
        return EVAL(state, player), null
    v_max, move ← −∞, null
    for each a in ACTIONS(state) do
        v, a' ← MIN-VALUE(RESULT(state, a), depth + 1)
        if v > v_max then
            v_max, move ← v, a
            α ← MAX(α, v_max)
        if v_max > β then return v_max, move
    return v_max, move

function MIN-VALUE(state, depth, α, β)
    if IS-CUTOFF(state, depth) then
        return EVAL(state, player), null
    v_min, move ← ∞, null
    for each a in ACTIONS(state) do
        v, a' ← MAX-VALUE(RESULT(state, a), depth + 1)
        if v < v_min then
            v_min, move ← v, a
            β ← MIN(β, v_min)
        if v_min < α then return v_min, move
    return v_min, move
```

to 35 in chess, which significantly reduces possible depth of search. Second, it is difficult to define good evaluation function. **Monte Carlo Tree Search (MCTS)** is a strategy that doesn't need the evaluation function and can handle high branching factor. It was successfully used in modern Go engines. Deepmind's *AlphaGo*, and it's improved version *AlphaGo Zero* combined MCTS with deep reinforcement learning and achieved a level of play beyond grandmaster. MCTS was also successfully used in chess with *AlphaZero* – a version of AlphaGo Zero adapted to chess.

As it was said earlier, MCTS does not use a heuristic evaluation function. Instead, the value of a state is estimated as the average utility over a number of simulations of complete games. Based on these estimated values, the algorithm gradually builds a search tree by focusing on the more promising moves, using the current game state as the root node, as long as there is time left for search. Each round of MCTS consists of the following four steps:

- **Selection**: Starting at the root R of the search tree, algorithm chooses a move, leading to a successor node, and repeats that process, moving down the tree to a leaf L. The root is the current game state and a leaf is any node that has a potential child from which no simulation has yet been initiated. To choose moves, algorithm uses a **selection policy** that selectively focuses the computational resources on the important parts of the game tree. The policy balances between the **exploitation** of deep variants after moves with high average win rate and the **exploration** of moves with few simulations.
- **Expansion**: Algorithm generates new child node C from the selected leaf and marks it with 0/0 (0 games won, 0 games played).
- **Simulation**: Algorthm performs a playout from the newly generated child node, choosing moves for both players according to the **playout policy**. These moves are not recorded in the search tree.
- **Backpropagation**: Algorithm uses the result of the playout to update information in the nodes on the path from C to R.

These four steps are shown in Figure 2.3. They are repeated either for a set number of iterations, or until the allotted time has expired, and then the move with the highest number of playouts is returned. This is better strategy than the intuitive choice of the node with highest average utility, because this node might be underexplored and therefore its score might be uncertain. For example, it is safer to choose the node with 65/100 wins than one with 2/3 wins.

The most common selection policy is called "upper confidence bounds applied to trees" or **UCT**. The policy ranks each possible move based on an upper confidence bound formula called **UCB1**. For a node n, the formula is:

$$\mathrm{UCB1}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\textsc{Parent}(n)}{N(n)}},$$

where

- $U(n)$ is the total utility of all playouts that went through node n,
- $N(n)$ is the number of playouts through node n,
- PARENT(n) is the number of playouts through the parent node of n
- C is a constant that balances between exploitation and exploration.

The first component of the formula above corresponds to exploitation; it is high for moves with high average win ratio. The second component corresponds to exploration; it is high for moves with few simulations.

Algorithm 2.3 Monte Carlo Tree Search

```
function MONTE-CARLO-TREE-SEARCH(state)
    tree ← NODE(state)
    while IS-TIME-REMAINING( ) do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts.
```

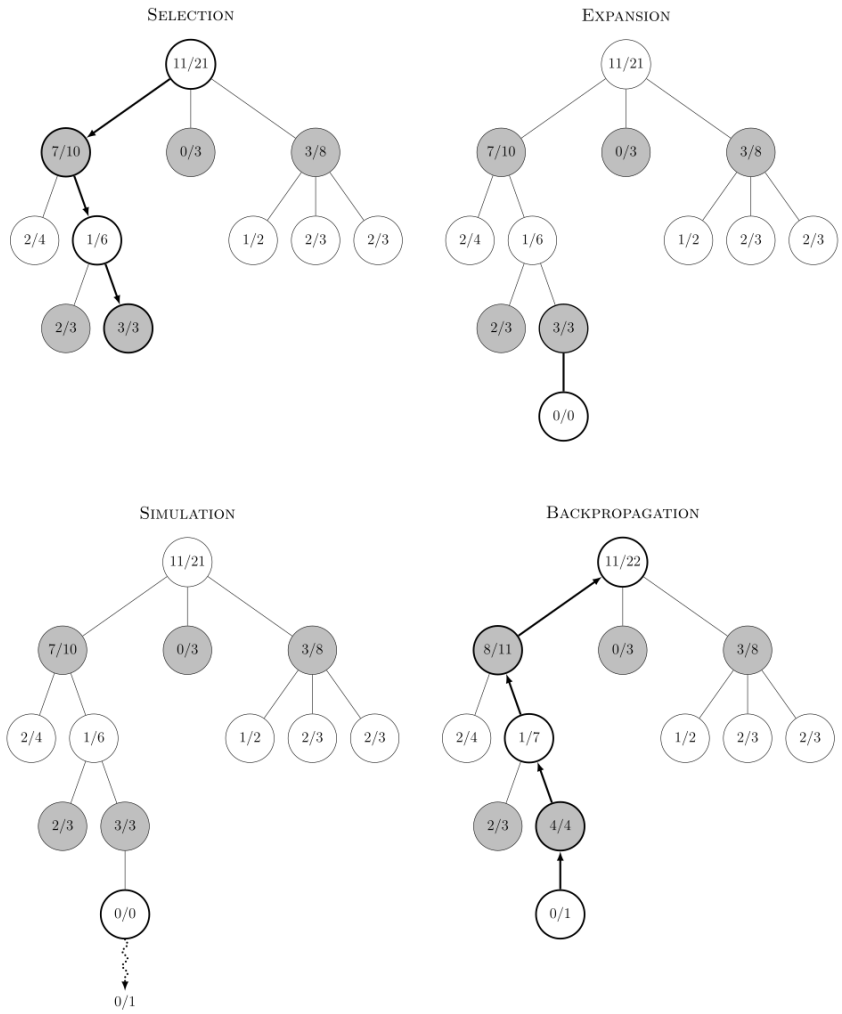


Figure 2.3: Single iteration of the MCTS. Each node shows the ratio of wins to total playouts

Chapter 3

Machine Learning in Chess

Until recent years, all the strongest chess programs were based on methods described in previous chapter. The major change came in 2017 with AlphaZero – a neural network based chess engine. This chapter describes modern approaches to chess AI, that are based on machine learning.

3.1 AlphaZero

AlphaZero was introduced in the paper *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* [4] in 2017. This research was updated and published in *Science* a year later [5]. In contrast to classic chess programs, it does not rely on a handcrafted evaluation function. Instead it uses deep neural network trained via reinforcement learning. The trained network is used to guide MCTS to select the most promising moves in games. AlphaZero is a successor to previous Deepmind's game-playing enginges: AlphaGo [6] and AlphaGo Zero [7], which were designed to play the game of Go. AlphaZero uses more generic approach than its predecessors and can be easily adapted to various board games. The paper's authors applied it to chess, shogi and Go, and outperformed state of the art programs in each of these games. The following sections describe AlphaZero in the context of chess.

3.1.1 Network architecture

AlphaZero's network $(\mathbf{p}, v) = f_\theta(s)$, with parameters θ, takes the board position s as an input and outputs a vector of move probabilities $\mathbf{p}$ with components $p_a = Pr(a|s)$ for each action a, and a scalar value v estimating the expected outcome z of the game from position s, $v \approx E[z|s]$.

Network's input is a position (with a few previous ones) encoded as 3D tensor (a multi-dimensional array with numerical values). The newtork is a **deep con-**

volutional network (CNN) based on *ResNet* [8] architecture. It's main part is a sequence of **residual blocks**. Residual block is a stack of two convolutional layers with a **skip connection**, which means that the The block input is added to the second layer output before applying final nonlinearity. After the residual blocks, the network splits into two heads: policy head and value head, which compute move probabilities and position value. Each of them contains additional convolutional and fully-connected layers. The network scheme is shown in figure 3.1.

Input details

The basic building block of the board encoding is a plane of size 8×8. Encoding of one position requires 14 planes:

- 12 planes are used to encode position of pieces (6 for each color) with one-hot encoding, as shown in figure 3.2.
- Additional 2 planes are used to record the number of repetitions. The first plane is set to all ones if a position has occurred once before and to zeros otherwise, and the second plane serves a similar purpose to encode if a position has occurred twice before.

The input contains current position and 7 previous positions, set to zero in early stages of the game. In addition, there are seven additional planes:

- 1 plane to encode the color of the player whose turn is, filled with ones if it's whites move and with zeros if blacks,
- 4 planes to encode castling rights. Each of these planes is set to ones if corresponding castle right exists,
- 1 plane to encode total move count with a single number,
- 1 plane to count moves without captures and move pawns.

The final input has the shape of $8 \times 8 \times 119$.

Output details

The output of the value head is a single number between -1 and 1. In policy head, probabilities of all possible moves are represented by $8 \times 8 \times 73$ stack of planes, which is flattened in to give the final output. Each of the 8×8 positions identifies the square from which to "pick up" a piece. The first 56 planes encode possible 'queen moves' for any piece: a number of squares [1..7] in which the piece will be moved, along one of eight relative compass directions (N, NE, E, SE, S, SW, W,

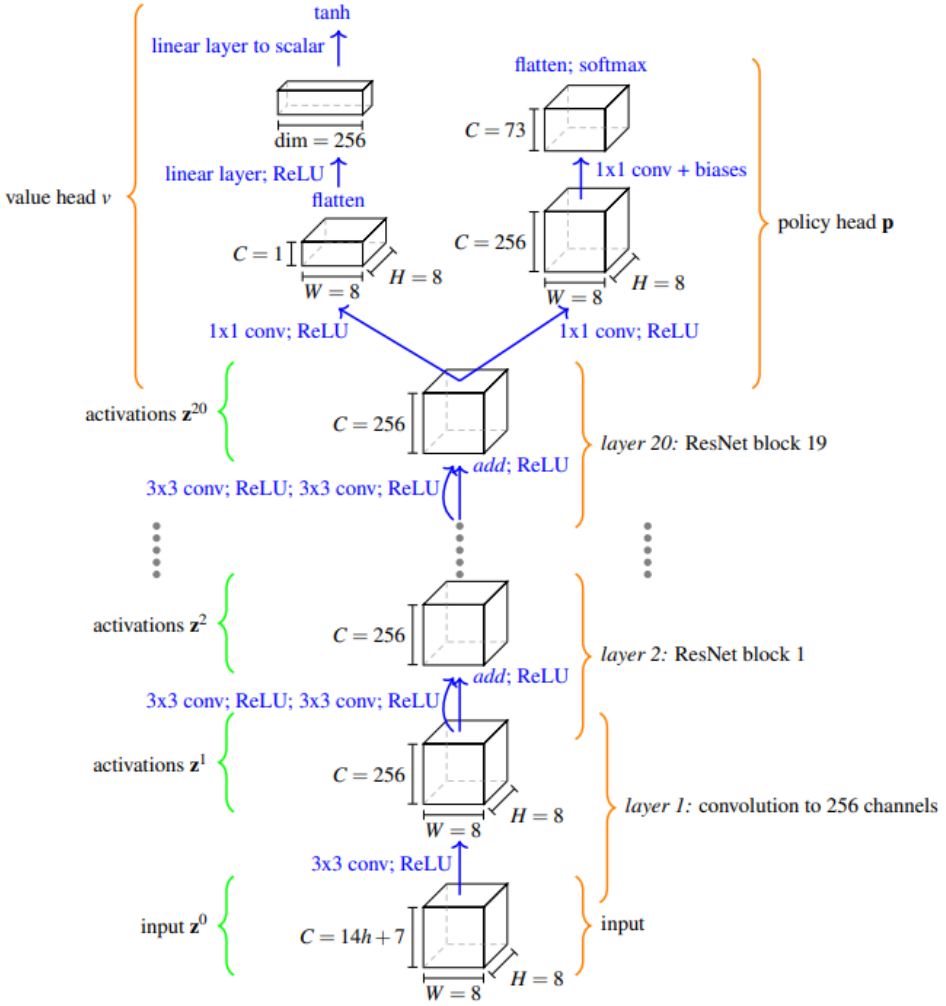


Figure 3.1: The AlphaZero neural network

NW). The next 8 planes encode possible knight moves for that piece. The final 9 planes encode possible underpromotions for pawn moves or captures in two possible diagonals, to knight, bishop or rook respectively, promotions to queen are covered in 'queen moves' part. Illegal moves are masked out by setting their probabilities to zero, and re-normalising the probabilities for the remaining moves.

3.1.2 Search

AlphaZero uses MCTS guided by neural network to search the game tree for the best moves. Each edge of the search tree, which represents a state-action pair (s, a), stores four values: the visit count $N(s, a)$, the total action-value $W(s, a)$, the mean action-value $Q(s, a)$ and the prior probability, of selecting a in s, $P(s, a)$. Steps of MCTS are slightly different than in the standard version described in Section 2.3.

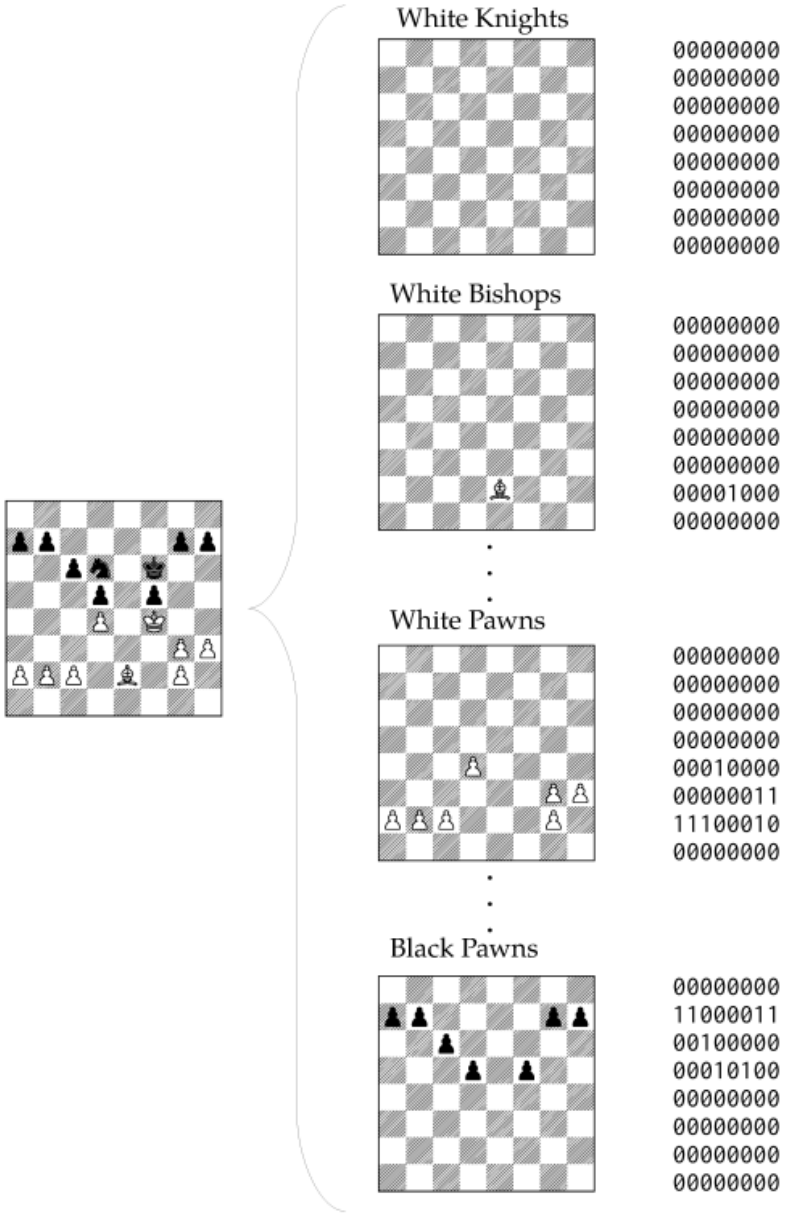


Figure 3.2: Chess board encoding

- **Selection:** Algorithm starts from the root node s_0 and traverses down until it reaches leaf node s_L at time-step L. At each time-step action a_t is selected as

$$a_t = \arg\max_a(Q(s_t, a) + U(s_t, a)),$$

$$U(s, a) = C(s)P(s, a)\frac{\sqrt{N(s)}}{1 + N(s, a)},$$

  where $N(s)$ is the parent visit count and $C(s)$ is the exploration rate, which grows slowly with search time, $C(s) = log((1 + N(s) + c_{base})/c_{base}) + c_{init}$, but is essentially constant during the fast training games.

- **Expansion:** The position of the leaf node s_l is evaluated by the neural network, $(\mathbf{p}, v) = f_\theta(s_L)$. The leaf node is expanded by adding and edge (s_L, a) for each possible action a. Each edge is initialized to $\{N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_a\}$.

- **Simulation:** There are no explicit simulations, they are replaced with evaluation value v, computed by the network.

- **Backpropagation:** Every edge from leaf s_L to root s_0 is updated: $N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$, $W(s_t, a_t) \leftarrow W(s_t, a_t) + v$, $Q(s_t, a) = \frac{W(s_t,a)}{N(s_t,a)}$.

Once the search is complete, the move to play is selected based on the visit count. During evaluation the engine chooses the move with the highest count. During training, the move is chosen either proportionally or greedily.

3.1.3 Training

The AlphaZero's network, with parameters θ, is trained by reinforcement learning from self-play games, starting from randomly initialized parameters. Each game is played according to the procedure described above and gives the following data:

- For each time-step of the game t, the network computes predicted game result v_t (value between -1 and 1) and vector of move probabilities $\mathbf{p}_t$.
- Also for each time-step MCTS computes improved move probabilities $\boldsymbol{\pi}_t$.
- At the end of the game we get the final game outcome z: -1 for loss, 0 for draw and -1 for win.

With this data, the network parameters θ are updated to minimize the error between the predicted outcome v_t and the game outcome z, and to maximize the similarity of the policy vector $\mathbf{p}_t$ to the search probabilities $\boldsymbol{\pi}_t$. Specifically, the parameters θ are adjusted by gradient descent on a loss function l that sums over mean-squared error and cross-entropy losses:

$$l = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c||\theta||^2.$$

where c is a parameter controlling the level of L_2 weight regularization. The updated parameters are used in subsequent games of self-play. During training, AlphaZero played 44 million games which required massive amount of computational power.

3.1.4 Performance

In [4] the researchers trained separate instances of AlphaZero for chess, shogi and Go. Training proceeded for 700,000 steps (mini-batches of size 4,096) starting from randomly initialised parameters. The trained instances were tested against current state of the art programs: Stockfish for chess, *Elmo* for Shogi and AlphaGo Zero for Go, by playing 100 game matches at tournament time controls of one minute per move. AlphaZero defeated all opponents, losing zero games to Stockfish and eight games to Elmo, as well as defeating the previous version of AlphaGo Zero (see Table 3.1). Chess AlphaZero was also tested to play games that started from common human openings. It defeated Stockfish in each opening, which suggests that it has mastered a wide spectrum of chess play.

Game	Opponent	Color	Win	Draw	Loss
Chess	Stockfish	White	25	25	0
		Black	3	47	0
Shogi	Elmo	White	43	2	5
		Black	47	0	3
Go	AlphaGo Zero	White	31	-	19
		Black	29	-	21

Table 3.1: Evaluation of AlphaZero in chess, shogi, and Go.

AlphaZero searches around 60 thousands position per second in chess, compared to 60 million for Stockfish. While Stockfish's search wide range of variants with alpha-beta, AlphaZero's MCTS focus on the most promising moves without losing much time to explore worse variations. This approach is closer to how human-players search for the best moves.

3.2 Leela Chess Zero

Leela Chess Zero (LC0) [9] is an open-source chess engine, based on the AlphaZero. It also uses neural network trained with reinforcement learning, but there are some changes in network architecture and training process.

The network is built from convolutional layers, similar to the AlphaZero. The difference is that the main part is the sequence of Squeeze-and-Excitation [10] blocks, instead of ResNet blocks. After that main part, the network splits into three heads:

- Policy head: outputs probabilities for each possible move.
- Value head: computes probabilities of win draw and loss (3 values instead of 1)
- Moves left head: computes how many moves are left until the end of the game.

The network training is conceptually the same as in AlphaZero, the difference is in the technical details. AlphaZero was trained on supercomputer with computing power unavailable to any individual developer. In order to replicate the results by DeepMind, the authors of Lc0 gave the computer chess community a chance to contribute to the project. There are two parts of the training: feeding the network with data and updating parameters, and generating training games.

Updating the network is the easier part, it can be handled by a reasonable powerful computer system. This part was done on the central server. The more computationally expensive part was generating the training data, this was done by the community. The network parameters were distributed to the volunteers, who generated the training and sent it back to the server. Then the training data was used to update network weights, which were sent to the community, and so on.

3.3 DeepChess

AlphaZero was a completely new approach to chess AI, which not only replaced hand-crafted evaluation function with neural network, but also replaced alpha-beta search algorithm with MCTS. The more conservative idea was to improve or replace the evaluation function with a neural network, while keeping the classic search algorithm. One attempt to do this was *DeepChess* [11], presented in 2016. It was, according to the authors, first end-to-end machine learning-based method that results in a grandmaster-level chess playing performance.

3.3.1 Network and training

Classic evaluation function computes single for a position, and this value is then used to compare position during the search. DeepChess authors took a different approach, they trained neural network to directly compare positions. The network takes two chess position as a input and predicts which one is better from white's perspective.

Each position is represented as binary bit-string of size 773. Pieces placement requires $2 \times 6 \times 64 = 768$ bits (2 colors, 6 piece types, 64 squares). Additionally, 4 bits represent castle rights and the last bit – side to move.

The training consisted of several phases. First, deep autoencoder *Pos2Vec* (see Figure 3.3, Stage 1) was trained to perform nonlinear feature extraction. The net-

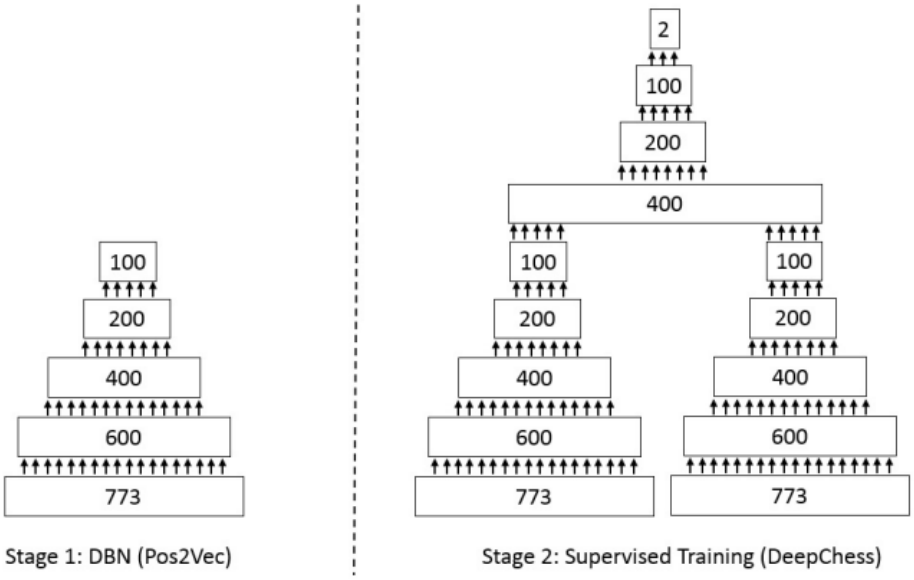


Figure 3.3: DeepChess architecture

work consists of five fully connected layers of sizes: 773–600–400–200–100, with ReLU activations. Each layer was trained separately, first layer was trained as 3-layer 773-600-773 autoencoder, then the weights were fixed and 600-400-600 autoencoder was trained, and so on.

Next phase was to train network *DeepChess* to compare positions (see Figure 3.3, Stage 2). This network consists of two copies of Pos2Vec with shared weights and 4 fully-connected layers conected to both Pos2Vec components. First part of the network extract features from both positions, and the second part compares them. The network ends with 2-value softmax layer, the output is the probabilities of each positions being better than the other one. During the training, the entire network including the Pos2Vec parts was modified. The training data consisted of pairs of positions, in each pair, one position was won by black, and the other one by white.

The last phase was improving the inference speed by shrinking the network. A smaller network was trained to mimic the behaviour of the original network. First, a smaller network of 773–100–100–100 neurons was trained to mimic the feature extraction. Then three layers of 100-100-2 neurons were added, and the entire network was trained to mimic the original DeepChess. This method achieved better accuracy than training the smaller network from scratch in the same way as the bigger model (97.1% vs 95.5%).

3.3.2 Search algorithm

The engine uses version of alpha-beta search algorithm that does not require any position scores for performing the search. Compared to the algorithm description from

chapter 2, each comparison of evaluation scores is replaced with direct comparison of position with the DeepChess network. α and β values are replaced with positions α_{pos} and β_{pos}. For each new position, algorithm compares it with the existing α_{pos} and β_{pos} positions using DeepChess, and if the comparison shows that the new position is better than α_{pos}, it would become the new β_{pos}, and if the new position is better than β_{pos}, the current node is pruned. Since DeepChess always compares the positions from white's perspective, when using it from black's perspective, the predictions should be reversed.

To speed up searching, the engine stores a hash table for positions and their corresponding feature extraction values. For each new position, it first queries the hash table, and if the position has already been processed, it reuses the cached values.

3.3.3 Performance

DeepChess was tested against *Falcon* [12]: a grandmaster-level chess program, which has successfully participated in several World Computer Chess Championships (WCCCs), winning a second place in speed chess in 2008. In a match of 100 games, with time limit of 30 per game for each side, which ended with a result of 51.5 - 48.5 for DeepChess. DeepChess achieved similar strength to Falcon despite being around four times slower. Another match was played where DeepChess was allowed to use 4 times more time than Falcon, this time Falcon was defeated 63.5-36.5. The authors concluded that the inference time of the neural network should be further reduced to achieve better results.

3.4 Efficiently Updatable Neural Network (NNUE)

The major breakthrough in combining alpha-beta search with neural network-base evaluation function came with the *Efficiently Updatable Neural Network* (NNUE). NNUE is a specific kind of network designed to work fast on a CPU. It was first used in Shogi, in 2018 [13], and then ported to chess. In 2020, *Stockfish NNUE* was introduced, with around 80 ELO increase, compared to the version with classic evaluation [14].

3.4.1 Network architecture

The neural network is shown in Figure 3.4, it consists of three hidden layers. The input is the chess position encoded as a binary vector called *HalfKP* (Half-King-Piece). The output is a single value – evaluation of the position in centipawns.

The HalfKP structure consists of two halves. First half encodes the relations

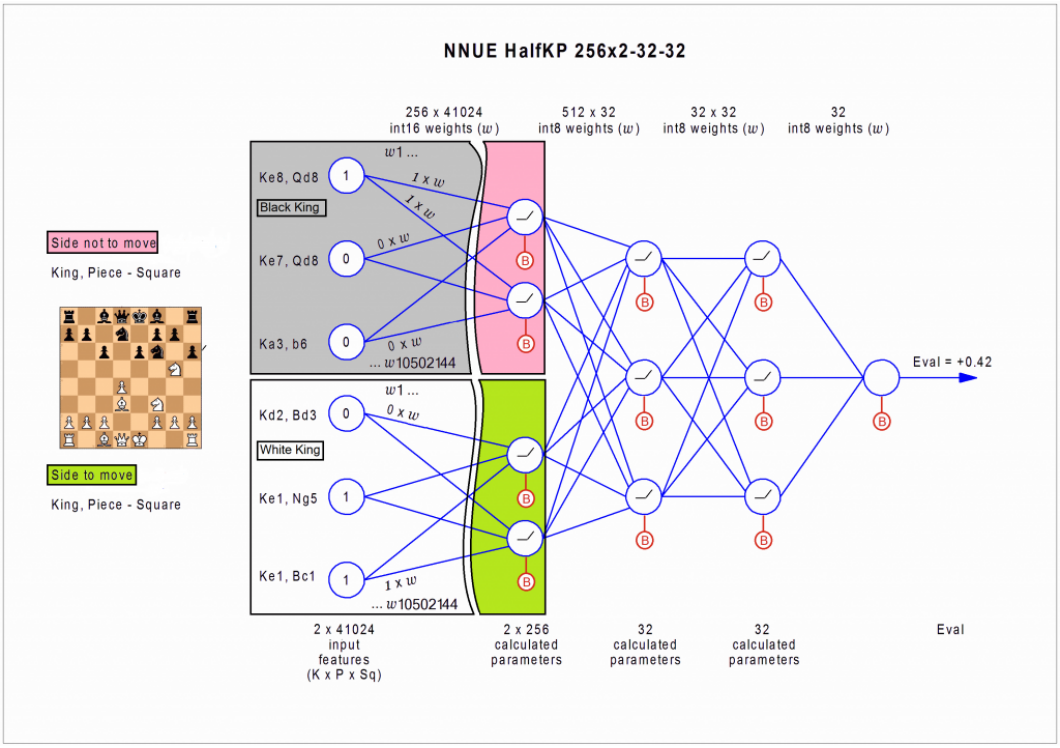


Figure 3.4: NNUE architecture

between current player's king and each non-king piece on the board. Each bit encode the triplet {current player's king position, current players piece type, position of that piece} or {current player's king position, enemy piece type, position of that piece}. For example, in position in Figure 3.4, we have:

- white king on e1, white pawn on a1: 0,
- white king on e1, white knight on f3: 1,
- white king on e1, black rook on a8: 1.

The second half of the input encodes the relations to the other's player king in the same way. The whole input contains, along with a relict from Shogi piece drop, $64 \times (64 \times 10 + 1) = 41,024$ inputs for each half.

The first hidden layer consists of two halves of 256 nodes each. Each of the halves of the input layer is fully connected to the corresponding half of 256 nodes (but there is no connection between non-corresponding halves). Then we have two fully connected layers of size 32 and final output node with a single value. Clipped ReLU activation function is used in each hidden layer, values larger than 1 are clipped to 1.

The weights, between input layer and first hidden layer, are arranged in such a way, that color flipped king-piece configurations in both halves share the same weights. For example the weight that corresponds to the arrow that connects the

bit “own king on e1, own pawn on d2” to the first node of the first layer, is the same weight that is used for the bit “enemy king on e8, enemy pawn on d7”.

The efficiency of NNUE is due to incremental update of the input layer outputs in make and unmake move, where only a tiny fraction of its neurons need to be considered in case of none king moves. The remaining three layers are computational less expensive. They can be computed fast with SIMD instructions [15].

3.4.2 Training

Training the network requires pairs of positions and their evaluation scores. For training Stockfish, Lela Chess Zero’s training data was used i.e. positions from self-played games of LC0 and it’s evaluation scores. Since this network is much less complex compared to the other top engines, it requires much less time and computational power to train.

3.5 Maia

Maia [16] is a chess engine which aims to play human-like moves. Current chess programs are far more powerful than the best grandmasters, so it doesn’t make much sense to use them as opponents for human players. They can be weakened by, for example, limiting the depth of the game tree that they are allowed to search. While this method is effective in reducing engine’s performance, it doesn’t necessarily make it play in a more human way.

Maia authors aimed to achieve two different goals. First, to predict move played by humans with higher accuracy than the existing engines. Second, to predict if a human is likely to blunder in a given position.

3.5.1 Predicting moves

To predict moves, Maia uses neural network similar to the AlphaZero. It is built of 6 residual blocks, the policy head and the value head. Unlike AlphaZero, Maia doesn’t use any kind of search to choose which move to play. It simply queries the network and plays the move with the highest probability. The training procedure is also different. Rather than using reinforcement learning, Maia is trained on games played by humans, using supervised learning. For each move in each game, vector of real move probabilities $\boldsymbol{\pi}$ has a value of 1 for the move played and 0 for all other moves.

9 separate versions of Maia were trained and tested. Weakened versions of Stockfish and Leela Chess Zero, in several variants, were also tested on the same task. Evaluation details are described below, figure 3.5 presents comparison of engines

performance.

Evaluation setup

To evaluate the engines, 9 test sets were prepared, one for each rating range from 1100 (1100-1199) to 1900 (1900-1999). Test data was taken from Lichess.org - online chess platform. For each rating bin, 10,000 games, where both players were in the proper rating range, was sampled, ignoring Bullet and HyperBullet games (games with less than 3 minutes per player). For each game, first 10 ply and moves where the player had less than 30 seconds for the rest of the game, were discarded. Eventually, each testing set contained around 500,000 positions.

Stockfish

15 depth-limited versions of Stockfish were tested, one for each depth between $1-15$. Each of them matched human moves between $33-41\%$ of the time. It turned out that the strongest version, of depth-search 15, is the most efficient in matching moves played at all tested ratings. Also, the effectiveness of each version increased as the strength of human players increased. This shows that limiting the search depth of Stockfish doesn't make the engine play more like human.

Leela Chess Zero

As it was explained in section 3.2, Leela is trained through the reinforcement learning, gaining strength over time, so the natural way to get several versions with different levels of play is to take network's weights at various times in the training process. This way, 8 different versions were selected, they are ranked with Leela internal rating system, which is not comparable to Lichess ratings.

Strong versions achieved higher move-matching accuracy than any version of Stockfish, scoring a maximum of 46%. However, all tested versions had prediction curves that are essentially constant, or have a slight positive or slight negative slope. Thus, even Leela does not match moves played by humans at a particular skill level significantly better than it matches moves played by any other skill level.

Maia

9 versions of Maia were trained and tested, one for each rating range between 1100 and 1900. Training tests were constructed in the similar way to test sets. For each rating range, 12 milion games were used for training and 120,000 for validation. Authors noted that 12 milion games is enough for Leela to go from random moving to a rating of 3000 (superhuman performance).

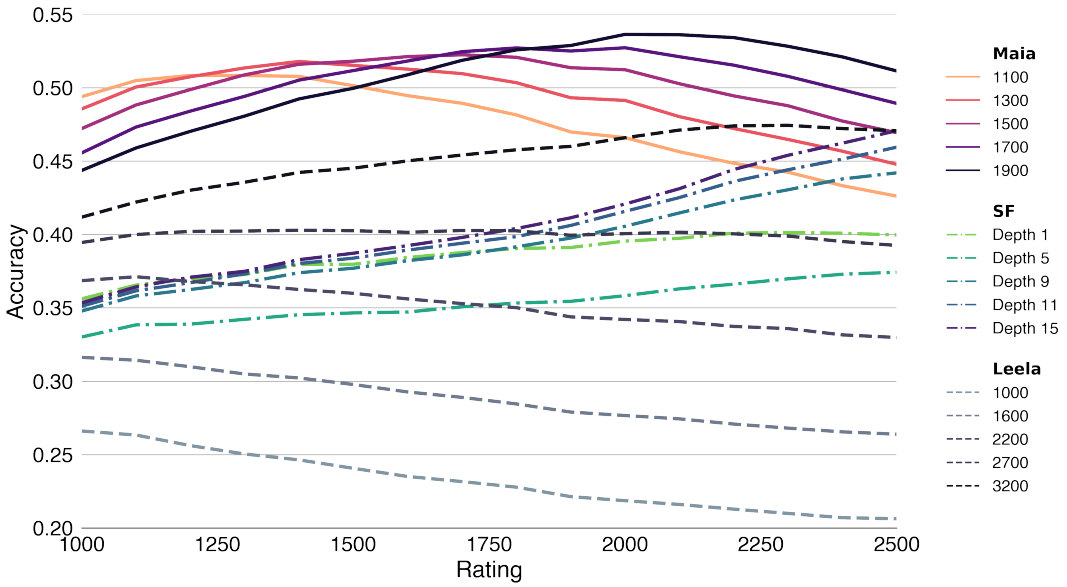


Figure 3.5: Comparison of move-matching performance for Maia, Stockfish, and Leela models.

Maia achieved significantly better than Leela and Stockfish. The lowest accuracy, when Maia trained on 1900-rated players predicts moves made by 1100-rated players, was 46% — as high as the best result of Leela. Highest Maia accuracy was over 52%. Each version of Maia achieved the best accuracy for a rating close to the rating it was trained and the accuracy goes down as the test rating decreases or increases. This shows that Maia has actually learned to predict moves play by human at a particular skill level. Additionaly, two modifications of Maia were tested: using 10 rollouts of MCTS to predict move, and adding history of last 12 ply to the network's input. Adding MCTS decreased performance by 5-10 percentage points, adding history increased it by 3 percentage points in all tests.

3.5.2 Predicting mistakes

Second task was to predict whether human players will make a significant mistake on the next move, often called a blunder. There were 2 versions of this task: in first model takes only position as an input, in second model also takes metadata about the players and the game: ratings, remaining time, material score etc.

Data

Training and test data were created from the same set of games, as for the move prediction task. Some of these games where annotated with Stockfish evaluation scores of given position, which were converted into a probability of winning the game (using the lookup table). Moves, that decrease probability of player's winning

by 10 percentage points or more, were labeled as blunder. The created training set contained 182 million blunders and 272 million non-blunders, the test set: 9 million blunder and 9 million non-blunder.

Results

First, several baseline models were tested: decision tree, logit, linear regression, random forest and naive bayes. After hyperparameters tuning, random forest achieved best accuracy: 56.4% when given only the board, and 63% with metadata.

Then, two neural networks w were trained and evaluated: 3-layer fully conected network and CNN with residual blocks, the same as for move prediction. The fully connected network achieved accuracy of 62.9% (board only) and 66.0% (with metadata). CNN achieved 67.7% and 71.7%.

Neural networks were also tested in predicting whether a significant fraction ($> 10\%$) of players made a mistake in a given position. Fully connected network achieved accuracy of 69.5%, CNN: 75.3% and deeper CNN (with 8 blocks and 256 filters): 76.9%. This task turned out to be easier than individual blunder prediction. Authors suggest that it may be due to the fact that grouping decisions together reduces noise.

Chapter 4

NLP in Chess

With the development of natural language processing techniques, some researchers made attempts to apply them in developing chess AI. This chapter describes two approaches of employing NLP in chess. The first approach is to learn from chess sources in natural language such as books and annotated games. The second is to train language models on text-archived chess games and use them to play by generating subsequent moves.

4.1 SentiMATE - Learning from Natural Language

A lot of natural language sources for learning chess are available online. These sources can potentially be used to train machine learning models. One of the attempts to do it is presented in the *SentiMATE: Learning to play Chess through Natural Language Processing*[1] paper. In this research, the authors used annotated chess games to train the move evaluation function, and then combined it with alpha-beta search to create a chess agent.

4.1.1 Architecture and training

The SentiMATE training pipeline consists of training three different models. The training data, consisting of chess moves and their corresponding comments, was collected from several chess websites. First model was a 'Quality vs Non-quality' classifier trained to predict whether a given comment contains information about quality of the move. It was used to remove the 'non-quality' comments from the dataset. On the remaining data, sentiment analysis was performed by the second classifier, resulting in a dataset of chess moves and their corresponding sentiments. This data was then used to train the third model to evaluate whether a given move is good or bad. The whole training pipeline is presented in figure 4.1. Details about each model and the training are described below.

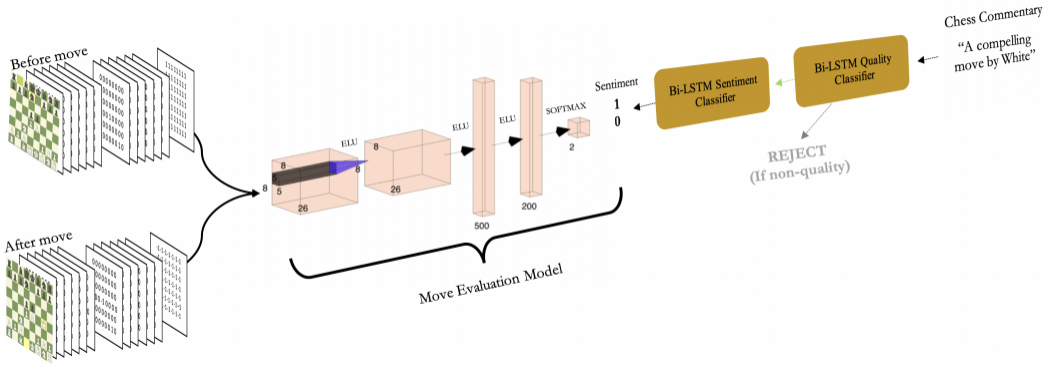


Figure 4.1: SentiMATE training pipeline

Quality move category classification

'Quality' classifier was trained to perform binary classification ('quality' vs 'non-quality') on the individually commented chess moves, scrapped from several chess websites. The authors hand-labeled 2700 commentaries for training and 300 for testing. The data was cleaned to ensure better classification quality: comments represented by single token, comments suffixed with seemingly arbitrary number and non-standard punctuation were removed.

Three different architectures for text classification with pretrained GloVe embeddings [17] were trained: recurrent neural network (RNN), long short-term memory (LSTM) and bi-directional LSTM. The hyper-parameters were finetuned using a hyper-parameter search. Bi-directional LSTM achieved the highest accuracy: 94.2% for 'quality' data and 78.5% for 'non-quality' data. Then different word embeddings were tested with the bi-directional LSTM: BERT [18], Glove, and stacked BERT and GLoVe. BERT embeddings achieved the best result: 96.3% for 'quality' comments and 84% for 'non-quality'.

Sentiment analysis

Another LSTM model was used to predict, wheter a comment describe good or bad move. It was train on 2090 examples and then tested on 233. Training and test sets consisted only of comments from the 'quality' class. The model achieved accuracy of 91.42% and balanced accuracy of 90.83%.

Move evaluation

Move evaluation network, shown in figure 4.1, was composed of convolutional and fully connected layers. The model takes as input a stacked representations of the board before and after a move. Each board is encoded as $8 \times 8 \times 13$ tensor. Each of the first 12 channels encodes positions encodes the position of pieces in one type

and color. +1's are placed at points where white pieces are found, -1's for blacks and 0's at points where no pieces are found. The last channel encodes whether it is black or white to move (all +1's for white pieces, all -1's for black pieces). The network is built of 2 convolutional layers with filters of size 3 and 5, each with 26 filters, 'same' padding, ELU activation function and dropout of 0.25, followed by two fully connected layers of size 500 and 200 with ELU activation, and final layer with 2 outputs and softmax activation to give probabilities of the move being good and bad. This model was trained on 15000 moves with their corresponding sentiments.

4.1.2 Alpha-beta move search

To search for the best move, the evaluation model was used in a modified version of alpha-beta search, in which nodes still represent states S_t however as the algorithm traverses down the tree it also stores a variable with the prior board state S_{t-1}. These two states are then concatenated to give the input $I = [R(S_t), R(S_{t-1})]$ for the model.

4.1.3 Performance

SentiMATE was tested against an agent with random strategy and DeepChess implementation. Because of computational limitations that prevented deep-enough look-ahead, the agent couldn't checkmate its opponent, therefore a game outcome was decided by Stockfish evaluation after 40 moves. In 100 games, SentiMATE won 81% of the time against the random agent and beat DeepChess on search depth 1 as both black and white (there were no randomness in both agents, so only two games were possible from the starting position). Additionally, the material score (using standard value of each piece) was measured for each player at every move, the averaged values over time are presented in figure 4.2. According to the authors, the results suggest, that SentiMATE's strength occurs later in midgame and shows no strategical advantage during opening play. This is a common issue among chess engine, which is why they usually use the opening books to guide them into middle play.

4.2 Modeling Games with Language Models

Transformer-based language models [19] are currently the state of the art technology in the field of natural language processing. They also achieved great performance outside of NLP, for example in image generation [20] and modeling gameplay of popular board games like Go [21] and chess.

This section describes research on modeling chess games with OpenAI's Generative Pre-trained Transformer (GPT-2) [22] – a decoder-only transformer architecture

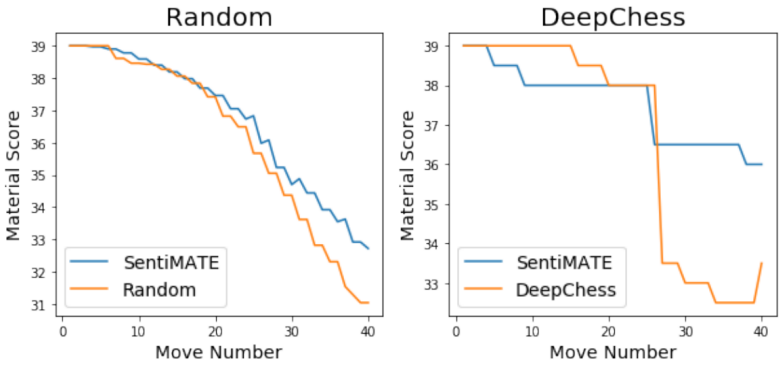


Figure 4.2: Material Score over time averaged over 100 games against Random Player and over 2 games (as Black and White) against DeepChess

for predicting the next word (or token) in a given text. Pretrained models, shared by OpenAI, were trained on 40 GB of text data as a corpus of highly ranked Reddit posts to predict next token in a sequence. They are available in four sizes: small (with 124M parameters), medium (355M), large (774M) and extra-large (1.5B). We describe research on fine-tuning the pretrained model (section 4.2.1) and training the model from scratch (section 4.2.2).

4.2.1 Fine-tuning the transformer

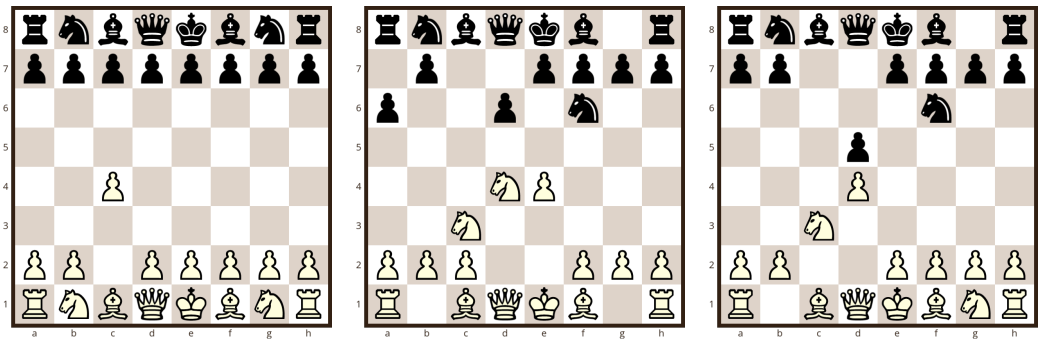
In the research *The Chess Transformer: Mastering Play using Generative Language Models* [23] GPT-2 (small and large version) was fine-tuned on the collection of games in PGN format (see Appendix A for details about chess notation) to generate reasonable gameplay.

Data

Two separate training datasets of PGN games were used: the smaller dataset with 11,291 games and average Elo player rank of 1815, and the larger one with 2.19 milion games and average Elo over 2200. Each game record was transformed to a single line and stripped of all headers except for the result.

Training and testing

For each training run 30,000 training steps were performed, and then games were generated with random sampling without a prefix or trigger prompt. For a generated game to be considered viable, it had to begin with "[Result ...]", with "1-0" if white wins, "0-1" if black or "1/2-1/2" in case of a draw e.g. "[Result 1-0] 1. e4 e5 2.Nf3 Nc6 ...". Different training subsets of the data were tried: games in which only one



(a) English Opening (b) Sicilian Najdorf (c) Slavs Exchange

Figure 4.3: Openings generated by the fine-tuned GPT-2

color wins, only draw and mixed. Each fine-tuned version generated approximately 1000 games, with a 2-8% failure rate for non-viable gameplay or for illegal moves.

Results

Both small and large model, trained whether on small or large dataset, were able to generate plausible gameplay. Large model trained on large dataset generated 971 new valid games imitating 2637 Elo rank. The average length of generated game were 67 moves, similar to the average of smaller dataset - 73. The model was able to generate several classic openings, including English Opening (1. c4), Slav Defense: Exchange Variation (1.d4 d5 2.c4 c6 3.cxd5 4. Nc3 Nf6), Sicilian Defense: Najdorf Variation (1.e4 c5 2.Nf3 d6 3.d4 cxd4 4.Nxd4 Nf6 5.Nc3 a6) and others. Some of them are presented in Figure 4.3.

4.2.2 Training the transformer from scratch

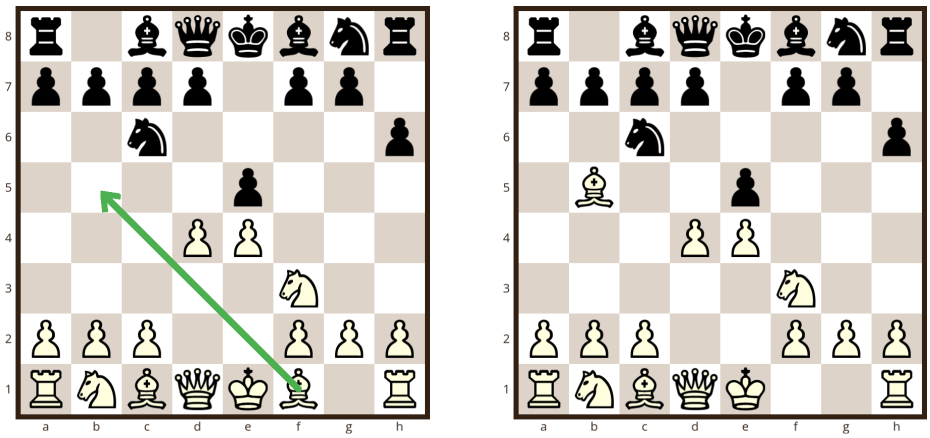
Fine-tuning the transformer to generate chess games gave promising results, but it has some disadvantages. The model's tokenization process and its vocabulary are designed to be effective in processing the natural language, not the chess notation. Therefore, in processing chess games, the tokenization will be ineffective and most of the vocabulary will never be used. For this reason, it may be more effective to design a chess-specific vocabulary and train the model from scratch. This was done with the GPT-2 model in the research *Chess as a Testbed for Language Model State Tracking* [24].

Data format

Custom tokenizer was used to translate games from UCI notation to tokens. In UCI, each move is represented as start and end coordinates of the piece e.g. `e2e4` (see Appendix A). In the model's vocabulary, single token represents board square

Type	Examples	Count
Square names	`e4`, `d1`	64
Piece type	`P`, `K`, `Q`, `R`, `B`, `N`	6
Promoted pawn piece type	`q`, `r`, `b`, `n`	4
Special symbols	BOS, EOS, PAD	3
Total		77

Table 4.1: Model vocabulary



(a) (b)

Figure 4.4: Board state before (a) and after (b) the bishop at `f1` is moved to `b5`. UCI notation represents the move as `f1b5`.

(e.g. `e4`), piece type or pawn promotion. With additional special symbols it gives a vocabulary of 77 token types (see Table 4.1). For example, the move sequence "`e2e4 e7e5 g1f3`" is tokenized to "`e2, e4, e7, e5, g1, f3`".

The UCI notation allows to test the trained model's state tracking abilities. For example, in the prompt "`e2e4 e7e5 g1f3 b8c6 d2d4 h7h6 f1`", the underlined move sequence leads to the board in Figure 4.4a. The next predicted token, that can be interpreted as the ending square predicted for the bishop at `f1`, will provide some information about the board state awareness of the model e.g. prediction `g1` will tell us that the model does not understand that the piece at `f1` is a bishop, `g2` – that it does not know that there is a white pawn on `g2`.

Other variants of notation included tokens with piece type to test the model's ability to track the specific piece. For example, prompt "`e2e4 e7e5 N` tests if a model knows positions of white Knights. In "UCI + RAP p" (RAP – randomly annotated piece) notation the piece type for each move is added with p% probability during training (but not during inference). "UCI + AP" notation includes piece type for each move during training and inference. Examples of each notation are presented in Table 4.2.

Notation	Training	Inference
UCI	e2, e4, e7, e5, g1, f3	e2, e4, e7, e5, g1, f3
UCI + RAP 15	e2, e4, P, e7, e5, g1, f3	e2, e4, e7, e5, g1, f3
UCI + RAP 100	P, e2, e4, P, e7, e5, N, g1, f3	P, e2, e4, P, e7, e5, N, g1, f3
UCI + AP	P, e2, e4, P, e7, e5, N, g1, f3	P, e2, e4, P, e7, e5, N, g1, f3

Table 4.2: Token sequences corresponding to the move sequence e2e4 e7e5 g1f3 for different notations during training and inference.

Board State Probing Tasks

Models' ability to track the game state was tested with a set of tasks. In each tasks the model is fed with the game prefix followed by a single prompt token, and has to predict the starting or the ending square of the next move in the game.

- **Ending Square Tasks** – the model is fed with a game prefix and starting square of the next move and has to predict the ending square.
  - **End-Actual** – the prompt token is a starting square of the actual move from the game.
  - **End-Other** – the prompt token is a starting square of any piece that can be legally moved
- **Starting Square Tasks** – the model is fed with a game prefix and a piece type of the next move and has to predict the starting square of that piece.
  - **Start-Actual** – the prompt token is a piece type of actual pieced that was moved in the game.
  - **Start-Other** – the prompt token is a piece type of any piece that can legally move.

Examples of each task are presented in Table 4.3. Actual prediction were evaluated in terms of both exact move (ExM) accuracy (whether the model predicted the true starting/ending square) and legal move (LgM) accuracy (whether the model predicted a legal square). LgM predictions were also tested with R-Precision – a percentage of legal squares among R squares with the highest probabilities, where R is the total number of legal squares.

Training

The models were trained on games ranging from 10 to 150 moves in length. Three training set were used: "Train-L" with 200K games and its subsets "Train-S" and "Train-M" with 15K and 50K games respectively. Dev and test set with 15K games

Task	Prompt Token	Correct Answers (ExM)	Correct Answers (LgM)
End-Actual	`f1`	`b5`	{`e2`, `d3`, `c4`, `b5`, `a6`}
End-Other	`f3`	N/A	{`d2`, `g1`, `h4`, `g5`, `e5`}
Start-Actual	`B`	`f1`	{`f1`, `c1`}
Start-Other	`N`	N/A	{`f3`, `b1`}

Table 4.3: Examples of each probing task with the corresponding correct answers. All examples assume the language model was fed the prefix `e2e4 e7e5 g1f3 b8c6 d2d4 h7h6` and the actual next move is `f1b5`.

each were used for perplexity evaluation, Dev test perplexity was used for choosing hyperparameters. Another 50K games were used to create 1000 instances of board state probing tasks. Selected game prefixes were not present in the training data, each prefix length was between 51 and 100. Tasks didn't contain prompts for pawns because of their limited move options.

GPT2-small architecture was used as a base model. Other models were also tested, including Reformer [25] and Performer [26] transformers, LSTM and GPT2-small with limited attention window to 50 most recent tokens. For the UCI + RAP p% notation, the models were fine-tuned over p based on their validation perplexity. The best results were achieved with $p = 25$ for Train-S and Train-M, and $p = 15$ for Train-L.

Each model was trained for 10 epochs with a batch size of 60 and early stopping when validation loss, calculated after every epoch, increases. Adam optimizer was used with learning rate of 5×10^{-4} and L2 weight decay of 0.01.

Results

Tables 4.4 and 4.5 show results of predicting starting squares and ending squares with GPT2-small. The authors noted, that the model learned to identify the location of pieces. After training on Train-L, model achieved 100% accuracy and R-precision for actually moved piece with UCI + RAP notation and over 99.5% for other tasks (Table 4.4). The model also learned to predict legal moves with accuracy over 97% (Table 4.5). The ability to predict all legal moves, measured with R-Precision, was significantly lower – about 85%. This was expected, since the model was trained on actual games, so it mostly learned only reasonable moves, not necessarily all legal moves. Most of the illegal moves were classified as *Syntax* (illegal move for the type of the piece, regardless of the board state), *Path Obstruction* (other piece is blocking the path) or *Pseudo Legal* (the moving player's king is in check after the move). Examples of illegal moves are shown in Figure 4.5.

Other tested models performed worse than base GPT2. The closest results were achieved by the Performer and the GPT2 with limited attention window (1-2p.p.

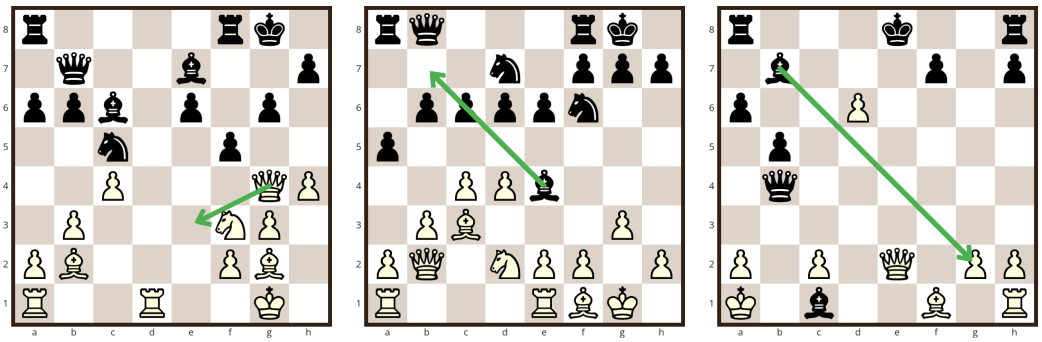
worse in each test).

Training set	Notation	LgM				ExM
		Actual		Other		
		Acc.	R-Prec.	Acc.	R-Prec.	Acc.
Train-S	UCI+RAP	91.3	90.2	89.3	89.2	78.8
	UCI + AP	99.2	99.1	98.8	98.8	86.9
Train-M	UCI + RAP	98.2	98.0	98.6	98.7	88.0
	UCI + AP	99.9	99.8	100.0	100.0	90.2
Train-L	UCI + RAP	100.0	100.0	99.6	99.5	91.8
	UCI + AP	99.9	99.9	99.7	99.7	91.1

Table 4.4: Accuracies and R-Precisions (%) for predicting starting squares

Training set	Notation	Lgm				ExM
		Actual		Other		
		Acc.	R-Prec.	Acc.	R-Prec.	Acc
Train-S	UCI	74.0	61.1	65.5	57.7	26.7
	UCI+RAP	88.4	75.5	80.4	72.1	33.3
	UCI + AP	87.0	77.0	78.8	72.3	36.1
Train-M	UCI	92.9	80.6	85.8	78.5	42.2
	UCI + RAP	94.9	82.2	87.9	78.0	45.9
	UCI + AP	94.7	82.4	88.3	79.1	47.3
Train-L	UCI	97.7	85.6	91.9	83.8	52.0
	UCI + RAP	97.0	86.1	93.1	83.9	54.7
	UCI + AP	98.2	87.3	95.2	86.3	56.7

Table 4.5: Accuracies and R-Precisions (%) for predicting ending squares



(a) *Syntax*: Queen can move like all other piece types except for knight.

(b) *Path Obstruction*: The pawn at `c6` is blocking the bishop.

(c) *Pseudo Legal*: The black king remains in check.

Figure 4.5: Examples of three types of illegal ending squares.

Chapter 5

Learning chess from natural language.

This chapter describes our implementaion of SentiMATE [1] described in section 4.1. Compared to the original paper, we have made some changes in the training pipeline:

1. We omitted the 'Quality vs Non-quality' classification. This would require us to hand-label the data which was not required for sentiment analysis (section 5.1.3 explains why). Also, we think that if the comment is not clearly negative, there is a high probability that the move is fine, so there is no need to discard some part of the data.
2. For sentiment analysis, instead of training the model (bidirectional LSTM in the original paper) from scratch, we fine-tuned the pretrained BERT model. Using the pretrained model should give better results, because it has already learned a lot about relations between words from a large corpus of data, this knowledge should be useful in learning the sentiment of chess comments. According to [18], fine-tuned BERT achieved better results, in multiple tasks, than the state of the art models at that time, including bidirectional LSTM.
3. For move evaluation, we introduced slight changes to the original architecture, they are described in section 5.2.1.

5.1 Sentiment analysis

5.1.1 BERT model

BERT [18] (Bidirectional Encoder Representations from Transformers) is an encoder-only transformer designed to pretrain deep bidirectional representations from unlabeled text. BERT was pretrained using a combination of two tasks:

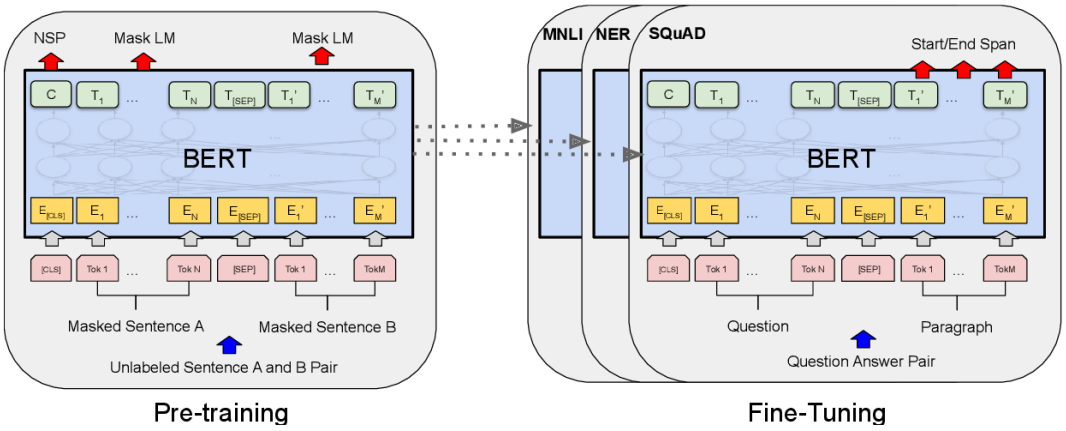


Figure 5.1: Pretraining and fine-tuning of BERT

- Masked language modeling (MLM): the model is given a sentence with 15% of words masked (replaced with `[MASK]` token) and has to predict the masked words.
- Next sentence prediction (NSP): the model is given concatenated sentences `A` and `B`, separated by `[SEP]` token and has to predict whether `B` directly follows `A` in the original text.

The pretrained model can then be fine-tuned for a different task after replacing only its last layer, the whole process is shown in Figure 5.1.

For our task of classifying the sentiment of chess comments, we used DistilBERT [27] – a smaller and faster version of the original BERT. We took the pretrained model `distilbert-base-multilingual-cased` from Hugging Face library. It was pretrained on Wikipedia pages in 104 languages. We used the multilingual version, because our data consisted of comments in multiple languages. The model has 6 layers, 12 heads and hidden size of 768, totalizing 134M parameters. Listing 5.1 shows how to load the model for classification task in Python and the model's tokenizer.

5.1.2 Data

We collected annotated games from multiple sources, including Lichess Studies[1], ChessBase Mega Database 2022, other online databases[2,3,4] and chess books available in PGN format. Then, we extracted annotated moves and their corresponding annotations. We cleaned the comments by removing tags inside square brackets that start with `%` symbol e.g. `[%cal Ge2e4]` (information for a chess GUI to draw

[1] `https://lichess.org/study`
[2] `https://www.chessgames.com/`
[3] `https://gameknot.com/`
[4] `https://www.angelfire.com/games3/smartbridge/`

```
from transformers import AutoModelForSequenceClassification,
                         Autotokenizer

checkpoint = "distilbert-base-multilingual-cased"

# Loading the model
model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)

# Loading the tokenizer
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

# Using the tokenizer
print(tokenizer("white [SEP] Good move for white!"))

# Output:
{
  'input_ids': [101, 15263, 102, 13073, 18577, 10142, 15263, 106, 102],
  'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]
}
```

Listing 5.1: Loading the pretrained model and the tokenizer from Hugging Face. The loaded model has randomly initialized last layer and requires fine-tuning.

green arrow from e2 to e4), `[%clk 0:19:22.4]` (remaining time). Comments, that contained only these tags, were removed. The resulting dataset contained slightly over 5 million of commented moves.

To each comment, we added information whether it describes white or black move, by adding a prefix with color followed by a `[SEP]` token. For example, comment `"Good move!"` describing white's move became `"white [SEP] Good move!"`. This was done, because some comments might be positive or negative, depending on which side's move they describes. For example, `"This move is good for black."` describes good move if it is black's move, but bad move if it is white's move.

5.1.3 Training

To train the sentiment classifier we had to assign labels to certain part of the data. Instead of doing it by hand, as it was done in the original paper, we automatically labeled comments with annotation symbols:

- comments with "!" (good move) and "!!" (briliant move) were labeled as positive,
- comments with "?" (mistake), "??" (blunder) and "?!" (dubious move) were labeled as negative,

- comments annotating white's move with symbols of white's advantage ("+-", "±") and black's move with black advantage ("-+", "∓") were labeled as positive.

2.5 million comments from our dataset were annotated with these symbols. Some of the symbols were located as text in the comments (`"1.e4 {!Good move.}"`), others were in the form of Numeric Annotation Glyphs (NAGs, `"1.e4 $1 {Good move.}"`, see Appendx A for more details). In the first case, the symbol was seen by the model during training, in the second – it was not.

We randomly selected 400K comments for the training set and 10K for the validation set, each set contained equal number of positive and negative comments.

We trained the model using Cross-Entropy Loss. During training, comments of similar length were batched together. Batch size was variable: from 32 for samples shorter than 64 tokens, to 4 for samples longer than 256 tokens. We used AdamW [28] optimizer with learning rate of 2×10^{-5}. We trained the model for 5 epochs, but the highest validation accuracy of 97.8% was achieved after 3 epoch, so this version was later used for classifying the unlabeled comments. Training took approx. 5 hours (40 minutes per epoch). Every training in this thesis was done using the PyTorch Lightning framework. In this chapter each model was trained on a single NVIDIA GeForce GTX 1650 Ti.

5.2 Move evaluation

5.2.1 Models

For move evaluation, we trained and tested the original CNN architecture from 4.1, as well as several variations. We tried the following changes:

- changing the value of a dropout,
- adding a skip connection alongside the convolutional layers,
- changing output size from 2 (probabilities of being good and bad) to 1 (single probability of being good)
- removing input channels that signify whether it is black or white to move, this changes the input size from $8 \times 8 \times 26$ to $8 \times 8 \times 24$.

Listing 5.2 shows our PyTorch module that can handle these changes.

```
class SentimateNet(nn.Module):
    def __init__(self, input_channels = 26, dropout = 0.1,
        output_size = 2, skip_connection = False):
        super().__init__()
        self.skip_connection = skip_connection

        self.conv_layers = nn.Sequential(
            nn.Conv2d(input_channels, input_channels, 5, padding='same'),
            nn.ELU(),
            nn.Conv2d(input_channels, input_channels, 3, padding='same'),
        )

        self.fc_layers = nn.Sequential(
            nn.ELU(),
            nn.Flatten(),
            nn.Dropout(dropout),
            nn.Linear(8*8*input_channels, 500),
            nn.ELU(),
            nn.Linear(500, 200),
            nn.ELU(),
            nn.Linear(200, output_size)
        )

    def forward(self, x):
        out = self.conv_layers(x)
        if self.skip_connection:
            out = (out + x)
        out = self.fc_layers(out)
        return out
```

Listing 5.2: Class of move evaluation models. The network returns logits, to get probabilities, softmax (or sigmoid) has to be applied to the model's output.

5.2.2 Data

To the dataset of 5 million moves with comments, we added 2.3 million moves annotated only with annotation symbols. We also generated additional examples of bad moves, we took 3.7 million positions where the good move was played and labeled one random move in each position as bad. We assumed here that there is a high chance that random move is bad. Our complete dataset contained 11 million moves: 6.5 million bad and 4.5 million good.

5.2.3 Training

We trained our models with cross-entropy loss, AdamW optimizer, learning rate of 1×10^{-3} and batch size of 1024. We used 1% of our dataset for validation and

the rest for training. The models usually achieved prediction accuracy of $68 - 70\%$. Training took approx. 6 minutes per epoch. Later it turned out that the model with lower validation loss and higher accuracy doesn't necessarily give the stronger agent.

5.3 Chess agents

To test the trained models, we implemented three types of chess agents.

- **SentiMATE without search**: a simple agent that takes a position, feeds all legal moves to the model and plays the one with the highest score (score = the predicted probability of move being good)
- **Alpha-beta with SentiMATE evaluation**: agent that uses alpha-beta search with move evaluation model. The algorithm description from the original *SentiMATE* paper is not clear on how the scores of the moves in each variant are combined to give the final score of the variant. In our solution, we simply add the scores of our agent's (maximizing player) moves. Alpha-beta search requires function that evaluates position and our model can only evaluate moves, but we can treat a move score as a gain/loss in the position score.
- **Alpha-beta with material+SentiMATE evaluation**: agent that uses alpha-beta search with evaluation function that only detects checkmate and calculates material difference. In case of multiple moves having the same highest score, the agent feeds them into the model and chooses the one with the highest score.

We tested the agents against the random player and the alpha-beta with material evaluation (the same as in the paragraph above) and random choice among the moves with the highest score. The result of each game was decided by Stockfish after 40 moves, as it was done in the original paper. The game was considered won (or lost) if the engine indicated that one of the players has at least one pawn advantage, otherwise it was considered a draw.

5.3.1 Results

The model with the best results in our tests used the following parameters:

- `output_size`=2, `dropout`=0.1, `skip_connection`=`True`, `input_channels`=24.

It was trained for 3 epochs, longer training resulted in worse results, despite a slight decrease in validation loss. The results of each type of agent using this model are described below.

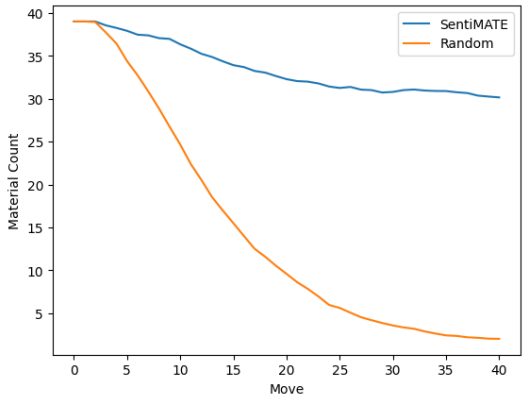


Figure 5.2: Material Score decay over time averaged over 100 games

Agent \ Opponent	random (100 games)	$\alpha - \beta$ search with material evaluation (depth=2, 20 games)
no search	94W/6D/0L	0W/2D/18L
$\alpha - \beta$ move search (depth=3)	97W/3D/0L	0W/1D/19L

Table 5.1: SentiMATE's performance

In a match of 100 games against the random player, the agent witht model and without search won 94% of games. This is a significant improvement over the original SentiMATE, which won 81% of games against the random player. We also measured a material score after each move, which can be seen in Figure 5.2. Compared to the material score of the original sentiMATE (presented in Figure 4.2), our agent starts gaining material advantage much earlier and its advantage grows much greater during the game. The same model with alpha-beta move search on depth 3 (deeper search took too long) won 97% of games. Then we tested these agents against the opponent with alpha-beta search and material evaluation in matches of 20 games, and the results were much worse. Our model couldn't win any games against this opponent with search depth of 2, it lost 18/20 games without search and 19/20 games with alpha-beta move search. The results are shown in Table 5.1.

We also played matches of 100 games between agent with alpha-beta, material evaluation and tie-breaking with the model against the opponent with random tie-breaking on search depth of 2 and 3. Both versions, with and without the model, are strong enough to win 100% games against the random player. The results are shown in Table 5.2. The agent using the model won a match against the opponent without the model with the same and smaller search depth. Adding move evaluation model clearly increased the agent's performance.

Agent \ Opponent	search depth = 2	search depth = 3
search depth = 2	72W/7D/21L	24W/4D/72L
search depth = 3	100W/0D/0L	57W/4D/39L

Table 5.2: Agent with material evaluation and SentiMATE vs opponent with only material evaluation.

5.4 Conclusion

Our research confirmed that it is possible to use a chess data in natural language to develop a chess-playing agent. We improved the training pipeline from the original *SentiMATE* paper [1]. We used better models and gathered more training data, which resulted in a stronger chess agent. We also showed how a move evaluation model can be combined with a classic minimax agent and improve the agent's playing strength.

Chapter 6

Modeling chess games with NanoGPT

In this chapter we present our research on language modeling for the game of chess. We aimed to replicate and potentially improve the methods from [24], described in section 4.2.2. Specifically, we introduced a novel approach to game tokenization that we believe enhances the model's performance. Our objectives were twofold: first, to evaluate GPT's ability to predict human moves in chess, and second, to leverage human game data to train the most strongest chess-playing model possible.

6.1 Model details

6.1.1 NanoGPT

For our experiments we used an open-source GPT implementation called NanoGPT [29]. It is a short and simple implementation of GPT in PyTorch. The model can be trained on a corpus of natural language. Then we can feed the model with a tokenized text and it will return a vector of probabilities for the next token (technically, the model returns logits, which require the application of the softmax function to obtain probabilities). We can generate text by choosing a next token based on the probabilities, appending it to the prefix, feeding the model with longer prefix and so on.

The same code can be used to play chess. We can represent each game as a sequence of moves and train the model with a large corpus of chess games instead of text in natural language. Then we can feed the model and it will return a vector of probabilities for the next move (in our representation each move is represented by a single token, it is explained in details in the next section), and we can play, for example, the most probable legal move.

Listing 6.1 shows how to initialize the model with random weights. Model size

```python
@dataclass
class GPTConfig:
    block_size: int = 1024
    vocab_size: int = 50304
    n_layer: int = 12
    n_head: int = 12
    n_embd: int = 768
    dropout: float = 0.0
    bias: bool = True

    def __str__(self) -> str:
        return str(self.__dict__)
```

```python
model_config = GPTConfig(
    block_size=301,
    vocab_size=tokenizer.vocab_size,
    n_layer=4,
    n_head=4,
    n_embd=256,
    bias=False,
)

model = GPT(model_config)
```

Listing 6.1: `GPTConfig` class and initialization of NanoGPT with random weigths.

can be configured by passing an object of `GPTConfig` class to the model's constructor.

6.1.2 Tokenization

In [24] each game was tokenized in a way that single token represented a single square or a piece type, therefore single move was represented by 2-4 tokens, depending on the notation variant used. We propose a different approach where each move is represented by a single token with information about piece type, starting square, ending square and (for pawn promotions) promoted piece type. Our vocabulary is presented in Table 6.1, it contains only moves that are legal in some positions, there are no tokens that represent e.g. queen moving like a knight. Including special symbols, this gives us 4087 tokens. For the game to be tokenized, it first has to be converted to "UCI + AP" notation, the whole process is presented in Figure 6.1.

We think that our tokenization might be better than the original for several reasons. First, it makes it easier to filter out illegal moves. Our model, after being fed with a game prefix, outputs the vector of logits of each move's probability. To get the most probable legal move, we simply need to find the index with the highest value among indices of legal moves. To get the same information from the original model we need to feed it with multiple prefixes and then combine the output values of separate tokens. This might be the reason why the agent implemented by the authors[1], simply generates 2-3 most probable tokens and plays randomly if the generated move is illegal.

Second, it should make it easier for the model to track the board state. The original model has to analyze several tokens and link them to each other to understand each move. Our model gets all this information in one token.

Third, we think that our tokenization reduces the chance of generating something that is not a valid chess game. In the original tokenization it is possible to

[1] `https://github.com/shtoshni/learning-chess-blindfolded`

Type	Examples	Count
King moves	`Ka1a2`, `Kf8g7`	420
Queen moves	`Qd1h6`, `Qd8e7`	1456
Rook moves	`Ra1b1`, `Re8e1`	896
Bishop moves	`Bf1g5`, `Bg7b2`	560
Knight moves	`Ng1f3`, `Nb7c6`	336
Pawn moves	`Pe2e4`, `Pb4c3`	236
Castles	`Ke1g1`, `Ke8c8`	4
Pawn promotions	`Pe7e8q`, `Pa2b1n`	176
Special symbols	`[EOS]`, `[BOS]`, `[PAD]`	3
Total:		4087

Table 6.1: Model vocabulary

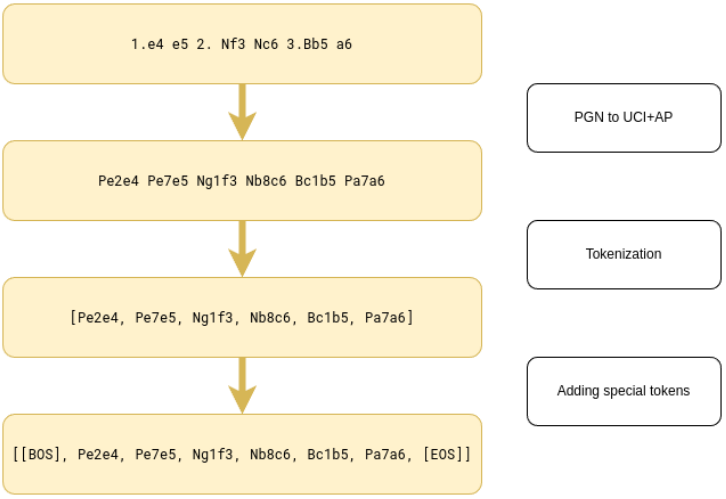


Figure 6.1: Tokenization of the game in PGN. In some experiments `[EOS]` token is not added.

generate moves that can never be legal (e.g. `[N,a1,a2]` – knight from a1 to a2) or something that can't even be considered a move, legal or not (e.g. `[q,K,e4]`). This type of errors are impossible with our tokenization.

6.2 Training details

6.2.1 Data

For training and testing we used games from Lichess Database [2], with a maximum length of 150 moves (300 ply).

[2] `https://database.lichess.org/`

6.2.2 Model configurations

We trained models of two different sizes:

- small: `n_layer=4, n_head=4, n_emb=256`, 4.2 million parameters,
- large: `n_layer=8, n_head=8, n_emb=512`, 27.3 million parameters.

The rest of the config parameters were set to `block_size=301` (maximum game length + `[BOS]` token), `vocab_size=4087`, `dropout=0.0`, `bias=False` (better and faster according to the nanoGPT's author).

In this chapter, we trained the models on a single NVIDIA RTX A4000.

6.3 Predicting human moves

To test the GPT's ability to mimic the human play, we used the idea presented in *Maia* paper [16], described in section 3.5. We created rating bins for each range of 100 rating points from 1100-1199 to 1900-1999, each rating bin contained games where both players are in the same rating range. For each rating bin we created training, validation and test set with 6 million games for training, and 100,000 for both validation and test sets. We discarded all moves done after one player's remaining time dropped below 30 seconds. Then we tested the prediction accuracy of each model on each test set, not counting the first 10 ply in each game, as it was done in the *Maia* paper. We also measured legal move accuracy (whether the model predicted a legal next move) for each model in its own rating range on a sample of 1000 games.

6.3.1 Training

For each rating range we trained a small model. Each model was trained for 10 epochs, using cross-entropy loss, AdamW optimizer, learning rate of 1×10^{-3} and batch size of 64. Batching was done in a way that games of similar length were batched together. Figure 6.2 shows the training and validation loss decay for one of the models. Training each model took approx. 4 hours (25 minutes per epoch).

6.3.2 Results

Figure 6.3 shows the prediction accuracy of each model for each test set. We can see several findings on the plot. First, the highest accuracy, for most of the models, is between 44.5-45%, with the single best value of 44.9%. Compared to the results from the *Maia* paper, it's worse than the highest accuracy of Maia (52%), but better than Stockfish (41%), and only slightly worse than Leela (46%). The lowest accuracy of

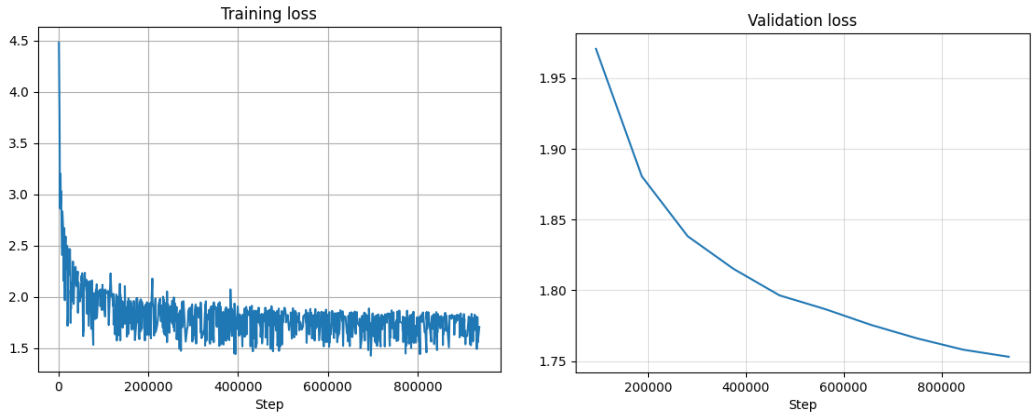


Figure 6.2: Training and validation loss decay during the GPT training.

GPT is 40.9%, which is very close to the highest accuracy of Stockfish. Second, similar to Maia, each model achieved the highest accuracy on a test rating range it was trained on, or the one next to it, and the performance drops for higher and lower ratings. This means that the models were able to learn how human players at a specific skill-level play. In Table 6.3 we present the accuracy of each model for its own training range and also the legal move accuracy. Approx. 99% of the moves predicted by the models were legal.

We also tested how each model's the move-matching accuracy, for the rating range it was trained on, changes over the course of the game. Figure 6.4 shows the mean accuracies for every range of 5 moves (10 ply). The highest accuracy, around 50%, the models achieve for the first 10-15 moves. Then the accuracy drops and stays between 40-45% until 70-95 moves, then it starts to drop quickly and reaches values to the values below 10% at 110-125 moves. We can observe several interesting findings:

- For the first 10 moves, the predictions for the highest rated players are slightly more accurate than the predictions for the lower rated players. The reason might be that better players more likely play common openings and less likely to play something unpopular (and worse) than weaker players. So there is less opening variety among the stronger players.

- Accuracy drops after about 80 moves. This might be due to the fact that there are very few longer games and there is not enough training data. Figure 6.5 shows the histogram of the game lengths in the training dataset for rating range 1500-1599. Games longer than 80 moves constitute less than 0.3% of the dataset.

- The accuracies are higher around 60 moves than for the earlier moves. On Figure 6.5, we can see that most games are much shorter than 60 moves. If the game is longer, it is almost certainly in the endgame around 60 moves.

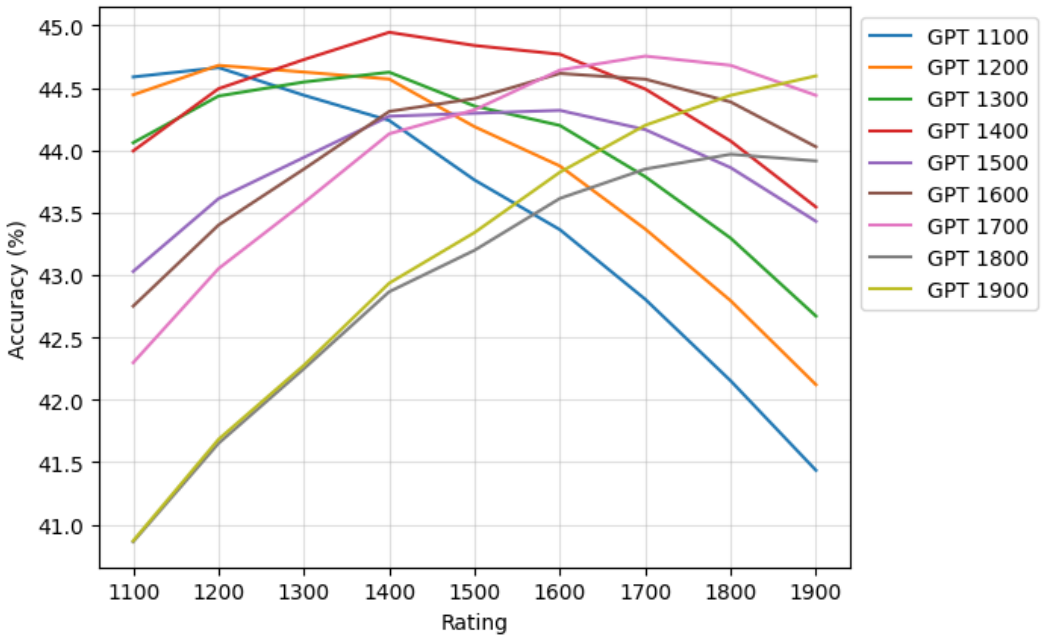


Figure 6.3: Move-matching performance of small GPT models.

In the endgame, there are less pieces on the board and gameplay more often relies on simple schemes like pushing a free pawn. Endgame moves are less diverse and therefore easier to predict.

6.4 Training a strong player

Our second goal was to train the strongest player possible. To achieve it, we modified our training procedure. The problem with standard training for move prediction is that there is no distinction between good and bad moves, each move is equally important. We wanted our model to focus more on better moves during training.

Model	Accuracy (%)	Legal move accuracy (%)
GPT 1100	44.6	98.9
GPT 1200	44.7	98.8
GPT 1300	44.5	98.8
GPT 1400	44.9	99.0
GPT 1500	44.3	98.7
GPT 1600	44.6	98.8
GPT 1700	44.8	99.0
GPT 1800	44.0	98.8
GPT 1900	44.6	99.0

Table 6.2: Models accuracy for their own rating range.

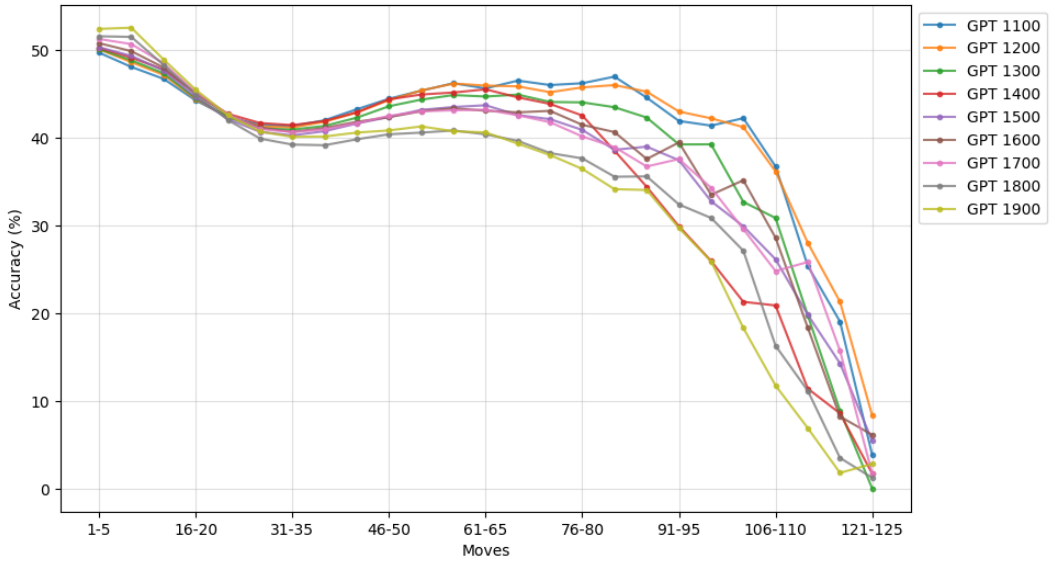


Figure 6.4: Move-matching performance's change over time. Each bullet shows the mean accuracy across the range of 5 moves (10 ply).

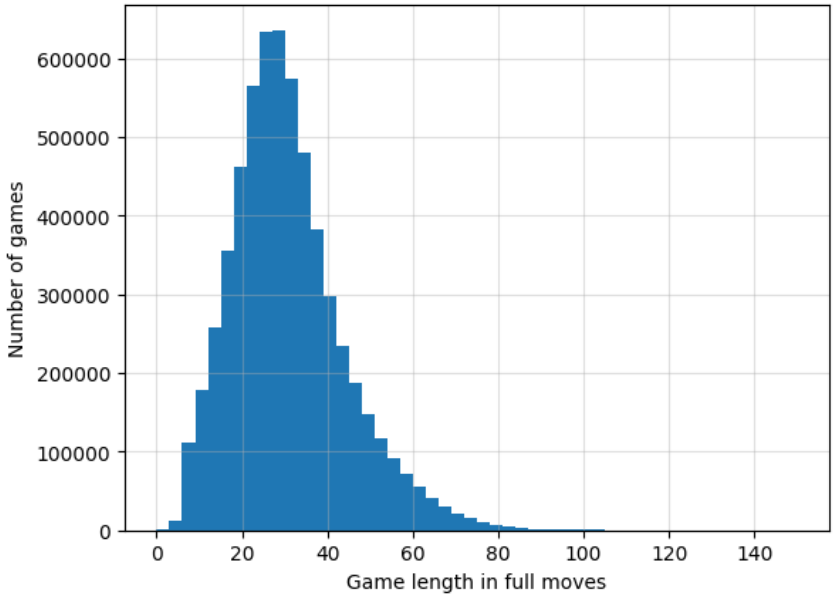


Figure 6.5: Histogram of game lengths for the rating range 1500-1599.

```python
@dataclass
class WeightsConfig:
    loss_weight: float = 0.5
    win_weight: float = 1.0
    draw_weight: float = 0.75

    use_elo: bool = True
    elo_bias: float = -1000
    elo_ratio: float = 1/1000

    def compute_weight(self, elo, result):
        if self.use_elo:
            elo_weight = (elo + self.elo_bias) * self.elo_ratio
            elo_weight = max(0, elo_weight)
        else:
            elo_weight = 1

        if result == 1: #win
            return self.win_weight * elo_weight
        elif result == 0: #loss
            return self.loss_weight * elo_weight
        else: #draw
            return self.draw_weight * elo_weight
```

Listing 6.2: `WeightsConfig class`

Therefore we added move weights to the loss function, so the moves of the winning players and (optionally) moves of the players with higher ratings are more important during training.

6.4.1 Cross-entropy loss with move weights

For each game in the training set, we calculate separate move weight for each player, using the instance of the `WeightsConfig` class, presented in Listing 6.2. During training, instead of calculating the mean of the cross-entropy loss, we calculate the weighted average, as it is shown in Figure 6.6.

6.4.2 Testing the models

For testing our models we implemented the agent that feeds the model previous game moves from the games and plays the legal moves with the highest predicted probability. We performed the following tests:

- Games against 9 Maia models from [16][3].

[3] `https://github.com/CSSLab/maia-chess/tree/master/maia_weights`

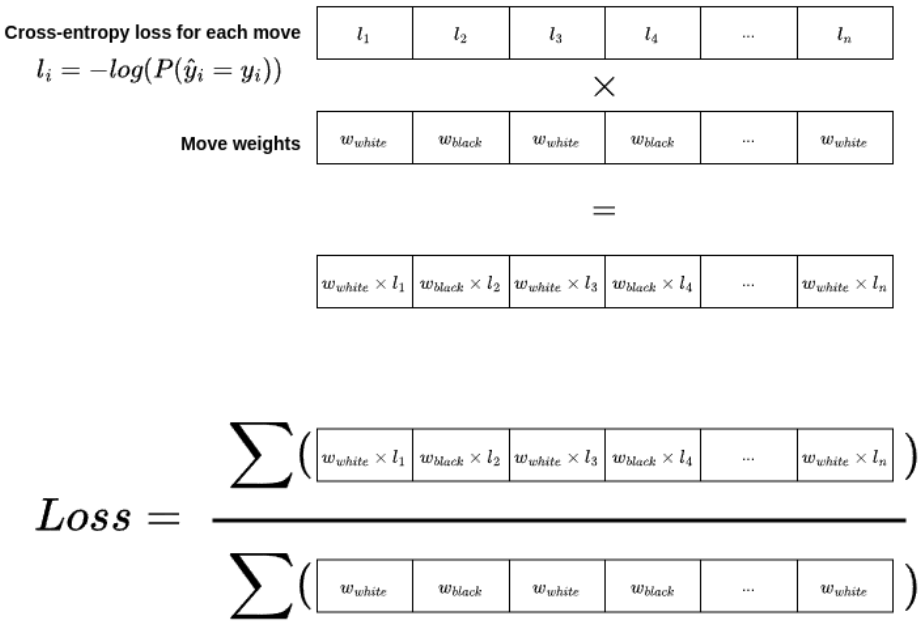


Figure 6.6: Weighted cross entropy calculation

- Games against Maia models starting with different openings. We prepared 100 openings by taking prefixes of the games from the database. Prefixes were 1-10 ply long. We played matches of 200 games, our models played as white and black from each opening.
- Games against agent with alpha-beta search, material evaluation function and random tie-breaking, the same as in Section 5.3.

Unlike the SentiMATE, the GPT models are usually able to checkmate the weaker opponent, so we played the games until the end or until we reached the models maximum game length of 150 moves. If the game was not over after 150 moves, the result was decided by Stockfish, but this occured, less than 5% of the time in tests against alpha-beta agent. More often, games ended in a draw due to the fivefold repetition (the same position occured five times during the course of the game), it happened in 15-40% of the time in tests against Maia agents.

We also implemented an agent with alpha-beta search, material evaluation and tie-breaking with GPT, by taking the most probable move among the best moves according to alpha-beta. We performed some of the tests mentioned above for this agent.

6.4.3 Training

We trained the models on the dataset of randomly sampled Lichess games, the only condition was that the winner (or both players in a case of a draw) had to have

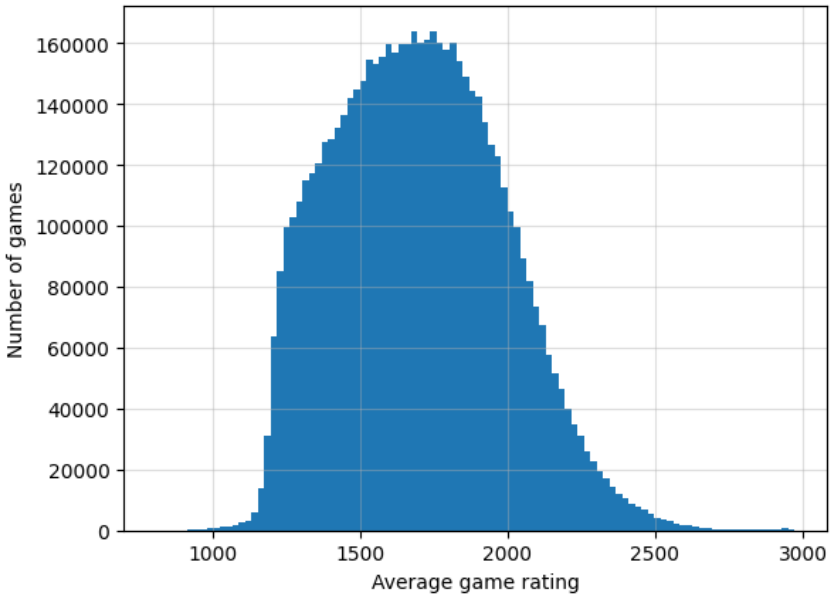


Figure 6.7: Histogram of average game rating in the dataset.

a rating of at least 1200. We used 6 milion games for training and 100,000 for validation. Figure 6.7 shows the histogram of the average game rating, its mean is 1699, median is 1690 and standard deviation is 286.

We wanted to test whether training with move weights gives some advantage over normal training, so we trained our models with three different training configurations:

- **base**: training without move weights.
- **weights 1**:
  - `loss_weight = 0.5, win_weight = 1.0, draw_weight = 0.75`
  - `use_elo = True, elo_bias = -1000, elo_ratio = 1/1000`
- **weights 2**: `use_elo = False`, score weights the same as in weights 1.

We trained small and large models using base training and weights 1, and large model using weights 2.

We trained each model for 5 epochs with AdamW optimizer, learning rate of 1×10^{-4} and batch size of 64. After each epoch of training we saved a model's checkpoint. Training each model took 2 hours for small models and 5 hours for large models.

Model size	Training type	Epochs of training				
		1	2	3	4	5
small	base	4.5	**5.5**	3.0	5.0	5.0
	weights 1	4.0	0.5	3.5	5.0	**6.0**
large	base	3.5	5.5	8.5	**9.0**	8.0
	weights 1	4.0	**11.0**	5.5	10.5	3.5
	weights 2	2.0	4.0	6.0	4.5	**12.5**

Table 6.3: Scores of the GPT models against the 9 Maia models. One match consists of 18 games (as white and black against each Maia)

	Games against Maia								
Training type	1100	1200	1300	1400	1500	1600	1700	1800	1900
base (4 ep.)	W/W	W/W	W/W	D/L	L/L	D/L	L/D	L/D	W/L
weights 1 (2 ep.)	W/W	W/W	W/L	W/W	L/D	D/W	D/D	W/L	L/L
weights 1 (4 ep.)	W/W	W/L	W/W	W/W	L/W	D/W	L/L	W/L	L/L
weights 2 (5 ep.)	W/W	W/L	W/W	W/W	W/W	W/D	W/L	L/L	W/W

Table 6.4: Best large models results against each Maia ("W/L" means that the GPT won as white and lost as black against a particular opponent)

6.4.4 Results

Against Maia

Table 6.3 shows the score each model achieved against 9 Maia models. For both model sizes, the best checkpoint from training with move weights achieved slightly higher score than the best checkpoint from base training. Table 6.4 shows the result of each game of the 3 large models with the highest score. Adding move weights gave slightly worse results against Maia 1100 - 1300 (1p less than base model) and significantly better against Maia 1400-1600 (3-4p more). Against Maia 1700-1900, model trained with game result and rating weights for 2 epochs achieved the same score as base model, but after 4 training epochs it achieved 1 point less. Model trained only with game result weights achieved in 1 point more than the base model in this rating range.

Since Maia models are in fact stronger than their training rating range, for example Maia 1500 has a Lichess rating of over 1650 in rapid chess, it seems that training with move weights might be beneficial against opponents with ratings close to the average rating of the training dataset.

Training type	vs Maia			
	1100	1400	1600	1900
base	96.5 (76/41/83)	83.0 (65/36/99)	72.5 (50/45/105)	60.0 (42/32/126)
weights 1	102.5 (74/57/69)	69.0 (47/44/109)	60.0 (39/42/119)	49.5 (32/35/133)
weights 2	91.5 (70/43/87)	77.0 (59/36/105)	64.5 (40/49/111)*	45.0 (29/32/139)*

Table 6.5: Results of large GPT models in games against Maia models, starting from different openings. Format: score (wins/draws/losses).
*Result after 4 epochs of training (in the table) was better than the result after 5 epochs.

Openings test

Table 6.5 shows how each large model performed against 4 Maia models in games from different openings. In this test, unlike the previous one, longer training almost always gave better results (the only 2 exceptions are marked in the table). Training with move weights didn't improve performance. Both models trained this way achieved worse results than the base model in all cases except one: the model trained with weights of 1 scored more points against Maia 1100 than the base model. This was also the only match won by a GPT. Even if a particular GPT is stronger than a particular Maia when playing from starting position, it is weaker in playing different openings. The reason might be that each Maia was trained on 12 million games, and each GPT only on 6 million, so Maia should have a broader knowledge of different variants.

Against alpha-beta with material evaluation

Table 6.6 presents how each model performed against alpha-beta search with material evaluation function. Here also longer training almost always gave better result. We can see that adding move weights (in version 2) resulted in significant improvement. Large model trained with weights 2 won 21 games more and scored 19.5 points more than the base model against the alpha-beta with a search depth of 3.

Alpha-beta with material evaluation and GPT tie-breaking

We tested alpha-beta agent with search depth of 3 and large GPT trained with weights 2 for 5 epochs. It performed better than other agents in most of the tests. The results are presented in Table 6.7.

Model size	Training type	Opponent search depth	
		2	3
small	base	27.5 (18/19/63)	10.0 (7/6/87)
	weights 1	26.0 (19/14/67)*	10.5 (7/7/86)
large	base	72.5 (67/11/22)	59.0 (53/12/35)
	weights 1	68.0 (61/14/25)	50.0 (46/8/46)*
	weights 2	76.0 (69/14/17)	78.5 (74/9/17)

Table 6.6: Results of the GPT models against the alpha-beta with material evaluation and search depth of 2 and 3 in matches of 100 games.
*Result after 4 epochs of training (in the table) was better than the result after 5 epochs.

Opponent				
Alpha-beta + material (depth=3)	Maia (9 models)	Maia 1400 (100 openings)	Maia 1600	Maia 1900
87.5 (85/5/10)	8.0 (7/2/9)	104.5 (87/35/78)	85.5 (72/27/101)	77.5 (62/31/107)

Table 6.7: Performance of the agent with alpha-beta and tie-breaking with GPT.

6.5 Conclusion

The results of our research shows that the GPT language models can be effective in modeling chess games. Even very small models can be trained to predict the next move played by a human with high accuracy. Larger, but still quite small models can be trained to play on a fairly high level. The new tokenization method, proposed here, seems to be more effective than the methods presented in previous research, and is certainly easier to use in the implementation of chess agents. Our new method of training a model using move weights is promising. It gave better results than the base method in some tests and would require further research.

Chapter 7

Summary

The goal of this thesis was to train chess-playing agents, using text information with neural networks and natural language processing. This goal was achieved.

In Chapters 2-3 we examined the existing methods for creating chess agents: classic AI algorithms (minimax, MCTS), neural networks and NLP techniques.

In Chapter 5 we described our improved version of SentiMATE [1]. We showed that the training pipeline from the original paper can be improved. We used better models and gathered more data, which resulted in higher performance.

In Chapter 6 we presented our research on modeling chess games with the GPT language model. We showed that the GPT architecture can be trained to predict chess moves played by a human with high accuracy. We also presented our novel training method aimed at training the strongest possible player. This method has shown promising results and is worth further research.

7.1 Future work

Research presented in this thesis can be continued in many ways. Some of our ideas are:

- in order to achieve a stronger GPT agent: testing different move weights configurations, training larger models on larger datasets,
- adding value head to NanoGPT and training it with MCTS and reinforcement learning like it was done with AlphaZero,
- combining learning from chess games records and learning from natural language in one chess agent, existing example of this might be ChessGPT [30].

Bibliography

[1] Isaac Kamlish, Isaac Bentata Chocron, and Nicholas McCarthy. Sentimate: Learning to play chess through natural language processing. *CoRR*, abs/1907.08321, 2019.

[2] Syzygy endgame tablebases. `https://syzygy-tables.info`.

[3] Stockfish evaluation guide. `https://hxim.github.io/Stockfish-Evaluation-Guide`.

[4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[6] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[9] Leela chess zero website.
`https://lczero.org`.

[10] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks, 2019.

[11] Eli David, Nathan S. Netanyahu, and Lior Wolf. Deepchess: End-to-end deep neural network for automatic learning in chess. *CoRR*, abs/1711.09667, 2017.

[12] *Falcon* on chess programing wiki.
`https://www.chessprogramming.org/Falcon`.

[13] Yu Nasu. NNUE: Efficiently updatable neural-network-based evaluation functions for computer shogi, 2018.

[14] Stockfish: Introducing NNUE evaluation.
`https://stockfishchess.org/blog/2020/introducing-nnue-evaluation`.

[15] Chess programming wiki: SIMD and SWAR techniques.
`https://www.chessprogramming.org/SIMD_and_SWAR_Techniques`.

[16] Reid McIlroy-Young, Siddhartha Sen, Jon Kleinberg, and Ashton Anderson. Aligning superhuman ai with human behavior: Chess as a model system. In *Proceedings of the 25th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2020.

[17] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[20] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *CoRR*, abs/2102.12092, 2021.

[21] Matthew Ciolino, David Noever, and Josh Kalin. The go transformer: Natural language modeling for game play. *CoRR*, abs/2007.03500, 2020.

[22] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

[23] David Noever, Matthew Ciolino, and Josh Kalin. The chess transformer: Mastering play using generative language models. *CoRR*, abs/2008.04057, 2020.

[24] Shubham Toshniwal, Sam Wiseman, Karen Livescu, and Kevin Gimpel. Chess as a testbed for language model state tracking. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 11385–11393. AAAI Press, 2022.

[25] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *CoRR*, abs/2001.04451, 2020.

[26] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking attention with performers. *CoRR*, abs/2009.14794, 2020.

[27] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019.

[28] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[29] Andrew Karpathy. NanoGPT. `https://github.com/karpathy/nanoGPT`.

[30] Xidong Feng, Yicheng Luo, Ziyan Wang, Hongrui Tang, Mengyue Yang, Kun Shao, David Mguni, Yali Du, and Jun Wang. Chessgpt: Bridging policy learning and language modeling, 2023.

Appendix A

Chess Notation

This appendix describes popular notations for describing chess moves, games and board positions.

A.1 Standard Algebraic Notation (SAN)

The most popular notation for recording the chess moves, used by most books, magazines, and newspapers.

Square and piece names

Square naming Each square of the board is identified by a unique coordinate pair: a letter to denote a file (from `a` to `h`) and a number to denote a rank (from `1` to `8`), see Figure A.1.

Piece naming Each piece type (except for the pawn) is identified by an uppercase letter: `K` for king, `Q` for queen, `R` for rook, `B` for bishop and `N` for knight.

Moves

Each move of a piece is indicated by the piece's letter, plus the coordinates of the destination square e.g `Be5` (bishop moves to `e5`), `Nf3` (knight moves to `f3`). Pawn move is denoted by the destination square alone e.g. `e4`

Captures Captures are denoted by an `x` before the destination square e.g. `Bxe5` (bishop captures a piece on `e5`). For pawn captures, the file of departure is added e.g. `exd5` (pawn from `e4` captures a piece on `d5`). For *En passant* capture the

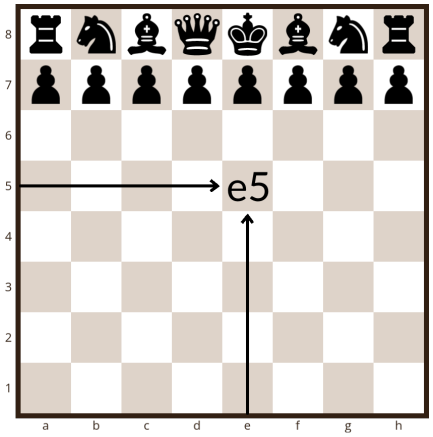
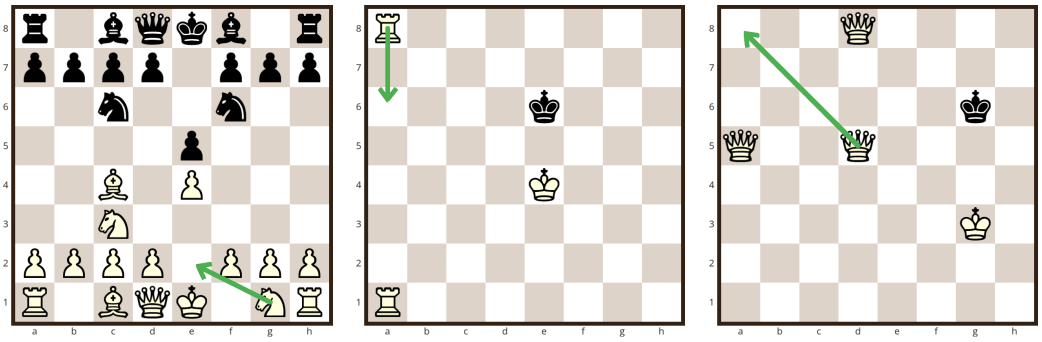


Figure A.1: Square naming



(a) `Nge2` (b) `R8a6` (c) `Qd5a8`

Figure A.2: Ambiguous moves in SAN notation.

destination square (not the square of captured pawn) is followed by `e.p.`, e.g. `exd6 e.p.` (pawn from `e5` moves to `d6` and captures a pawn on `d5`).

Ambiguous moves When two (or more) identical pieces can move to the same square, the moving piece is identified by adding after the piece type (in this order): the file of the departure square (if they differ), the rank of the departure square (if the files are the same but the ranks differ) or both (if neither file nor rank alone is sufficient to identify the piece) e.g. `Nge2`, see Figure A.2.

Promotion When a pawn promotes, the piece promoted to is indicated at the end with `"="` and piece type e.g. `e8=Q`, `exd8=N`.

Castling King-side castle is indicated by `O-O`, queen-side castle – by `O-O-O`.

Check and checkmate Check is represented as `"+"` at the end of the move, checkmate – as `"#"`.

A.2 Universal Chess Interface (UCI)

Notation used in communication between chess engines and the user interfaces, supported by most popular engines.

In UCI, each move is represented by coordinates of the starting square and destination square e.g. `g1f3` (Knight moves from `g1` to `f3`). Pawn promotion is indicated by adding the lowercase symbol of the piece promoted to e.g. `e7e8q`. Castle is denoted as king's move e.g. `e1g1` (white's king-side castle).

A.3 Portable Game Notation (PGN)

Standard format for recording chess games (both the moves and related data), which can be read by humans and is also supported by most chess software. There are two different sections inside a PGN: tag pairs (or headers) and movetext.

Tag pairs

Tag pairs provide metadata about the game, they usually include the following details.

- **Event**: The name of the event or match,
- **Site**: The location of the game,
- **Date**: The date when the game was played;
- **Round**: The specific round in which that game happened;
- **White**: The name of the player who had the white pieces;
- **Black**: The name of the player with the black pieces;
- **Result**: The outcome of the game.

This section can also contain extra fields with more data about the game like the Elo rating of the players, starting position in FEN (this allows to record only part of the game) and others. Each header contains tag name and tag value between square brackets e.g. `[Result "1-0"]`.

Movetext

Movetext contains enumerated moves in SAN and optional comments between curly brackets e.g. `1.e4 {King's pawn opening} 1...e5`.

NAG	Meaning	Symbol
$1	good move	!
$2	poor move or mistake	?
$3	very good or briliant move	!!
$4	very poor move or blunder	??
$16	White has a moderate advantage	±
$18	White has a decisive advantage	+-
$19	Black has a decisive advantage	-+

Table A.1: Examples of popular NAGs

Numeric Annotation Glyphs or **NAG**s are used to annotate chess games when using a computer, typically providing an assessment of a chess move or a chess position. NAGs exist to indicate a simple annotation in a language independent manner. They are composed of a dollar sign and a number e.g. `1.e4 $1`. Some of the popular NAGs are presented in Table A.1.

Full game example

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 {This opening is called the Ruy Lopez.}
3... a6 4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8
10. d4 Nbd7 11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6
16. Bh4 c5 17. dxe5 Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6
21. Nc4 Nxc4 22. Bxc4 Nb6 23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+
26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5 hxg5 29. b3 Ke6 30. a3 Kd6
31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5 35. Ra7 g6 36. Ra6+ Kc5
37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6 Nf2 42. g4 Bd3
43. Re6 1/2-1/2
```

A.4 Forsyth–Edwards Notation (FEN)

A standard notation for describing a particular board position of a chess game. The purpose of FEN is to provide all the necessary information to restart a game from a particular position. FEN record is a one text line with six fields, each separated

by a space. The fields are as follows:

1. **Piece placement**: Each rank is described, starting with rank 8 and ending with rank 1, with a "/" between each one; within each rank, the contents of the squares are described in order from the a-file to the h-file. Each piece is identified by a letter from standard notation, uppercase for White and lowercase for Black. A set of one or more consecutive empty squares within a rank is denoted by a digit from 1 to 8, corresponding to the number of squares.
2. **Side to move**: "w" for White, "b" for Black.
3. **Castling rights**: If neither side has the ability to castle, this field uses the character "-". Otherwise, this field contains one or more letters: "K" if White can castle kingside, "Q" if White can castle queenside, "k" if Black can castle kingside, and "q" if Black can castle queenside. A situation that temporarily prevents castling does not prevent the use of this notation.
4. ***En passant* target square**: This is a square over which a pawn has just passed while moving two squares; it is given in algebraic notation. If there is no en passant target square, this field uses the character "-". This is recorded regardless of whether there is a pawn in position to capture *en passant*.
5. **Halfmove clock**: The number of halfmoves since the last capture or pawn advance, used for the fifty-move rule: the game ends with a draw after 50 moves without a capture or a pawn move.
6. **Fullmove number**: The number of the full moves. It starts at 1 and is incremented after Black's move.

The example of FEN is shown in Figure A.3.

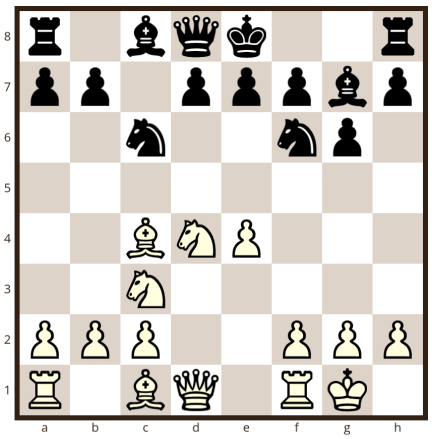


Figure A.3: FEN: `r1bqk2r/pp1pppbp/2n2np1/8/2BNP3/2N5/PPP2PPP/R1BQ1RK1 b kq - 3 7`