Abstract. In this article, we implement an algorithm to tune the heuristic evaluation function of a chess engine program by self-playing. The evaluation function $H_\theta(s)$ can be described as a linear combination of several features and the corresponding weights. The algorithm can tune the weight of each feature by playing with itself starting from a set of different initial starting chess positions. We run the algorithm with different initial weights, and see that the algorithm works well with random initial weight, but have a bad performance when starting with a strong initial weight.

1 Introduction

It has been almost 15 years after the chess computer IBM Deep Blue II beat the human world champion Kasparov. Nowadays it has not been essential to build such a specific and expensive computer system to play chess with the grandmaster level. The rapid development of hardware and more efficient algorithms make the personal computer be able to do that, too. The world most open source chess program Stockfish has at least a strength of 2900 ELO compared with world champion Kasparov whose ELO score is 2850 until 1997.

Here we should firstly explain what does “chess engine” mean. Chess engine can be regard as a “brain” to think about what the best move to make in the current board state. With a predefined text protocol\(^1\), the chess engine can communicate with the GUI program in order to know what the current board state is, and then think for a best move and tell GUI that best move. Most of the chess engines today use a structure including min-max search with a hand-coded evaluation function. The evaluation function return a specific value to a board state to indicate “how well does the computer think about this board state” in the perspective of one side (white or black). Typically this value should be in a range of $(-INF, INF)$, where $INF$ is a number large enough to represent that max player wins. Each chess board state should always contains a lot of features, such like “material balance”, “king safety”, “pawn structure” and so

---

\(^1\) There are two kind of protocols for computer chess, UCI (universal chess interface) and Xboard (chess board under X window).
The evaluation function can be seen as a linear combination of such features with a corresponding weight for each feature. Our goal in this project is to get such a vector (by training or tuning from an initial value) of weights in order to make the evaluation value more accurate and then get a stronger chess playing strength.

The successful application of reinforcement learning in checker has provided us a basic idea about self-play training. Samuel (1959) first introduced the idea of search bootstrapping in his seminal checkers player. In Samuels work, the heuristic function was updated towards the value of a minimax search in a subsequent position, after black and white had each played one move. In 1999, a chess program Knightcap use an algorithm called TD-Leaf, in which the heuristic function is adjusted so that the leaf node of the principal variation produced by an alpha-beta search is moved towards the value of an alpha-beta search at a subsequent time step. In 2009, Joel Veness published another algorithm called TreeStrap, with which the chess engine can be trained to reach a master level by self-playing.

In this article, we implement the RootStrap and TreeStrap algorithm, and use these algorithms to training the weight of evaluation function. We assigned two different initial weights for each algorithm, one is already pretty good (a master level), and the other is random. Our main task in this article is not only to demonstrate the effectiveness of these algorithms when starting from a weak, random weight, but also the degree of improvement on chess playing strength when applying the algorithm on a master level initial weight.
2 Chess Features

For each chess position\(^2\), the evaluation function should extract several features firstly to prepare for the analysis. It is just the \( \phi(s) \) that mentioned above. People may intuitively thought that more features will lead to a more detailed evaluation. However, this statement is not always true, especially when then number of features exceed 100. The chess program Viper owns a simple evaluation function, in which some most important features have already been extracted. So we can use these features directly. There are 62 features in total. Here we list all of them here to make brief explanation to each to them.

- **Materials**: This the most basic feature of a chess position. Chess is kind of board game that remove pieces from the board, and also the different pieces will have different function and power in the game. So it is important to know what the number of each kind of piece on the current board, and what pieces are the ones that I have but the opponent do not have. We will give each piece a value to indicate how important it is. Typically, we assign 100 to a pawn, 350 to knight, 350 to bishop, 550 to rook and 1100 to queen\(^3\). The king can have a very large number of material value, for example, 10000. Each piece value is multiplied by their corresponding piece number, and then we make a summation for all the products to get the material value of one side player. We give each kind of piece a weight to tune their material value.

- **Piece Square Tables**: The piece square tables is a two dimensional array, say \( \text{PSQ}[13][64] \). For each kind of piece on each square, there is a corresponding value to describe the degree that a piece prefer to be placed to a square. Typically, this numbers are all less than 100. For example, king is encouraged to be stay in middle of the board rather than being at the board edge, because it will reduce the mobility of king, especially in ending games. Also, We give each kind of piece a weight to tune their \( \text{PSQ} \) value.

- **Pawn Structure**: This feature is used to evaluate the structure of pawns. There are five sub-features describe five aspects: doubled pawn, isolated pawn, backward pawn, pawn chains, candidate passed pawn. Each sub feature also has a corresponding weight to adjust its value.

- **Mobility**: This feature is available for minor pieces and rook. To say simply, mobility is just the number of legal moves in the current position. There are different constant factors to enlarge this bonus according to different kind of pieces. Also, a weight for each kind of piece is given to tune this factor.

- **Development**: This feature tells us the development of minor pieces in the opening games. We know that in the opening, a chess player should try to move the minor pieces and pawns to capture more territory and also prepare for moves of heavy piece in the middle games.

---

\(^2\) In chess domain, the word “position” represents a static board state in any game.

\(^3\) In chess domain, we divide chess pieces into three types: pawns, minor piece, heavy piece. Some times we say “pieces”, the pawns may not be included here because we prefer to describe pawns independently to other pieces. Minor piece are knight and bishop, and heavy piece are rook and queen.
• **Multiple Passive Pieces:** This is a penalty to the pieces that have not enough space to move. Generally this feature will be zero, because we have mobility to get the same effect. However, if there are too many low mobility pieces, this feature can be an additional penalty for the situation of multiple passive pieces.

• **King Activity:** This feature is available in the ending games. It gives an additional bonus to the king with more safe squares to go, because it means that the king will be more hard to be checked by the opponent player.

• **King Shield:** This features describe the “wall” in front of the king so that enemy heavy pieces can not attack our king directly.

![Fig. 2](image)

Fig. 2. The squares (squares with crosses) that king shield pieces are located.

• **Queen Early Activity Penalty:** In the chess theory, people do not recommend to move the queen too early in the opening games. Because there are not enough space for the queen to launch attack, moving the queen too early will waste a lot of moves to find spaces. So we will give a penalty to the early queen move.

• **Block Central Pawn:** The central is very important to capture and board center, give the space to queen moves and protect king in the opening. So the moves of minor pieces in the opening should not block the central pawn.

• **Rook on Open File:** A rook on the open file can easy reach the bottom line of the opponent can launch attack to the king directly. Even the king escaped from the check, to rook can also capture some other piece behind king. So it is strong features to during attacking.

• **Bishop Trapped:** A bishop that surrounded by friendly pieces or enemy pieces will have a much less power and influence. We give penalty to the bishops in such situation.

• **Bishop Pair:** A bonus to the side who has two bishops, so that the moving of these two bishop can cover all white squares can all black squares.

• **Safely Promoted Passed Pawn:** The following several features are about passed pawns. Passed pawn is kind of pawn that there is no friendly pawn and enemy pawns in front of it on its file. The passed pawn is much easier to move to the bottom rank and promote than other pawns. Here we give a bonus to the pawn that is near the bottom rank, can promote safely.

• **Friendly Pawn near the Path:** A friendly pawn near the path of the passed pawn can protect the passed pawn, or recapture the piece who capture the passed pawn to become the next passed pawn on the same file. So we give bonus for pawn a such position.
• **Enemy King on the Path:** Enemy King on the Path is

• **Rook Behind Passed Pawn:** A rook behind a passed pawn will make the passed pawn be much more stronger because of the protection of rook. Sometimes we also call such a cooperation of rook and pawn a “tin opener”, because it can destroy the defense line of opponent with the minimum loss.

![Fig. 3. The structure of pawn-rook tin opener (in the red rectangle).](image)

• **Passed Pawn Against the Knight:** This feature is available when the opponent has only knights. The passed pawn is very strong against the knight because knight can not get enough number of moves to stop the passed pawn before it promotes.

• **Unstoppable Passed Pawn:** This feature is similar to the “Safely Promoted Passed Pawn”, but applied to the pawns that have not been near the bottom ranks.

Another note about the features is that most of the maximum value of the features above is less than 200, except piece materials.

### 3 Bootstrap Learning Algorithm

The idea of search bootstrapping is to adjust the parameters of a heuristic e-
valuation function towards the value of a deep search. The motivation for this approach based on the assumption that: if the heuristic can be adjusted to match the value of a deep search of depth $D$, then a search of depth $k$ with the new heuristic would be equivalent to a search of depth $k + D$ with the old heuristic. Here if we set $k = 0$, then we can see that a heuristic evaluation function is equivalent to the value return by a $D$ depth search with the same evaluation function. So we use the value return by a search of depth $D$ to update the the heuristic evaluation value. Suppose the for chess position $s_t$ at step $t$, $H_\theta(s_t)$ denotes the heuristic evaluation value, and $V_D(s_t)$ denotes the value returned...
by depth $D$ search, the updating is done by the stochastic gradient descent on the square error $\delta_t^2$ between $H_\theta(s_t)$ and $V^D(s_t)$.

$$\delta_t = V^D(s_t) - H_\theta(s_t)$$

$$\Delta \theta = -\eta \nabla_\theta \delta_t^2 = \eta \delta_t \nabla_\theta H_\theta(s_t)$$

Here $\eta$ is the step-size constant. Since we also know that $H_\theta(s) = \theta \phi(s)$, so we can simplify $\Delta \theta$ to the form that $\Delta \theta = \eta \theta \phi(s)$.

The core idea of algorithm below will follow this core idea above. The only differences between different algorithms are what search algorithm do they use and what $s$ are used during updating step.

### 3.1 RootStrap Algorithm

Here we present a algorithm named as RootStrap($\text{minimax}$) algorithm. According the idea above, we give the pseudocode of RootStrap($\text{minimax}$) algorithm.

**Algorithm 1** RootStrap-Minimax($\theta_0$, $S_0$)

- **inputs:** $\theta_0$, the initial weight for each feature in the evaluation function.
- $S_0$, the set of starting positions.

```plaintext
\theta \leftarrow \theta_0

\textbf{for all} s_0 \in S_0 \textbf{ do}
    \textbf{Initialise} t \leftarrow 0
    \textbf{while} s_t \textbf{ is not terminal state} \textbf{ do}
        V \leftarrow \text{minimax}(s_t, H_\theta, D)
        \delta \leftarrow V(s_t) - H_\theta(s_t)
        \theta \leftarrow \theta + \eta \delta \phi(s_t)
        \text{select } a_t = \arg \max_{a \in A} V(s_t \circ a)
        \text{Execute move } a_t, \text{ receive } s_{t+1}
        t \leftarrow t + 1
    \textbf{end while}
\textbf{end for}
```

In this algorithm, we use minimax search as the search algorithm. We use the greedy policy to select the best move $a_t$ at each chess position $s_t$, make the move and get the next state $s_{t+1}$. This process will be repeated until getting a terminal state $s_t$. A game, or we can also say a trajectory, has been generated here. The algorithm will restart with another initial state from $S_0$ and run another game, until using up all the initial states in $S_0$. The number of training games will be equal to the number of initial states.

The advantage of minimax search is that for each time when you call the search, it will return an exact value according to the principal variant, the it
is easy to implement. The disadvantage is obvious: the time complexity grows exponentially with respect to depth, so you can not set the depth $D$ too large or it will take too much time on search.

### 3.2 TreeStrap Algorithm

There are two algorithms: TreeStrap($\text{minmax}$) and TreeStrap($\alpha\beta$). The algorithm of TreeStrap($\text{minmax}$) make some change at the update step.

**Algorithm 2** TreeStrap-Minimax($\theta_0$, $S_0$)

inputs: $\theta_0$, the initial weight for each feature in the evaluation function. 
$S_0$, the set of starting positions.

\[
\theta \leftarrow \theta_0 \\
\text{for all } s_0 \in S_0 \text{ do} \\
\text{Initialise } t \leftarrow 0 \\
\text{while } s_t \text{ is not terminal state do} \\
\quad V \leftarrow \text{minimax}(s_t, H_\theta, D) \\
\quad \text{for } s \in \text{search tree do} \\
\quad \quad \delta \leftarrow V(s) - H_\theta(s) \\
\quad \quad \Delta \theta \leftarrow \Delta \theta + \eta \delta \phi(s) \\
\quad \text{end for} \\
\quad \theta \leftarrow \theta + \Delta \theta \\
\text{select } a_t = \text{arg max}_{a \in A} V(s_t \circ a) \\
\text{Execute move } a_t, \text{ receive } s_{t+1} \\
\quad t \leftarrow t + 1 \\
\text{end while} \\
\text{end for}
\]

Notice that the only difference between RootStrap($\text{minmax}$) and TreeStrap($\text{minmax}$) is the states $s$ used to update $\theta$. In RootStrap($\text{minmax}$), we only use the root node to do the update, while in TreeStrap($\text{minmax}$), all the nodes in the search tree have been used to update $\theta$.

Since TreeStrap($\text{minmax}$) will use all the nodes in the search tree, we need a data structure to store this nodes. We should not only record the value of each state, but also their features, so the conventional transition table is not enough here, because it can not store the full position or the features. To store all the features will cost too much memory, so the best approach is to extend the transition table and store the full board information in each table item. In order to save memory, we use the bit compression to store each position. The table item is define as follow:

```
struct hash_item_t {
    uint64 key;
    uint32 board_rank[8]; // each piece will occupy 4 bits, and there
    // are 8 square in each rank, so 32 bit in total.
```
int16 value, alpha, beta, depth;
};

Another algorithm is called TreeStrap(αβ). Use alpha-beta search to replace the minimax search in TreeStrap(minimax), we will get TreeStrap(αβ). The update step of TreeStrap(αβ) is a little different from TreeStrap(minimax) because alpha-beta search only return a bound for the exact value \( V^D(s) \). We use \( a^D(s) \) and \( b^D(s) \) to denote this two bounds, where \( a^D(s) \leq V^D(s) \leq b^D(s) \). So the updating step will change to \( H_\theta(s) \leftarrow a^D(s) \) if \( H_\theta(s) > a^D(s) \) and \( H_\theta(s) \leftarrow b^D(s) \), if \( H_\theta < b^D(s) \). In addition, the squared errors change to

\[
\delta^a = \begin{cases} 
  a^D(s) - H_\theta(s) & \text{if } H_\theta(s) > a^D(s) \\
  0 & \text{otherwise}
\end{cases}
\]

\[
\delta^b = \begin{cases} 
  b^D(s) - H_\theta(s) & \text{if } H_\theta(s) < b^D(s) \\
  0 & \text{otherwise}
\end{cases}
\]

\[
\Delta \theta = \frac{\eta}{2} \nabla_\theta (\delta^a(s)^2 + \delta^b(s)^2)
\]

\[
= \eta(\delta^a(s) + \delta^b(s))\phi(s).
\]

4 Experiment Setup

4.1 Experiment Environment

- **Hardware.** OSU high performance cluster system.
- **Programming Language.** ANSI C++.
- **Operating System.** Ubuntu Linux 11.04.
- **Other Tools.** GNU Emacs, Arena Chess GUI, cutechess-cli, BayesElo.

4.2 Chess Engine

In the experiment, we used chess engine Viper\(^4\), whose author is the Norwegian chess programmer Tord Romstad, as the training engine. Viper can run with single-thread or multi-thread mode. Here we only run the single-thread Viper. There is no official data to describe strength of the original edition Viper. According to the game results of other engines, for example GNU Chess, the strength of Viper might be at least 2400 ELO score. Viper can connect to a GUI so that human and other chess engines can play with it.

4.3 Opening Positions Set

Before we get the result the of training, the engine will play with itself a large number of games. Although the weight is changing during different games, we

\(^4\) You can find it here: [http://www.glaurungchess.com/viper/](http://www.glaurungchess.com/viper/).
also use a set of starting positions to make sure that each game will not repeat the previous ones. The starting positions set we use is crafty openings suite. It is a set of 4000 positions generated Dr. Robert Hyatt, the author of the famous computer chess engine Crafty. All the positions are in the turn of white move. We can simply ask to engine to play as black side to make the first move, so that we can get two different games from each starting position. As a result, we can run 8000 different games with this starting position set.

**Fig. 4.** The first chess position among 4000 starting positions in the crafty opening suite.

### 4.4 Training Methodology

We extract 62 features from the evaluation function of Viper, that is the \( \phi(s) \) described above. The step-sizes of different algorithms are: \( \eta = 1.0 \times 10^{-5} \) for RootStrap(minimax), \( \eta = 1.0 \times 10^{-6} \) for TreeStrap(minimax), \( \eta = 5.0 \times 10^{-7} \) for TreeStrap(\( \alpha \beta \)). The search depth \( D \) is set to be 4 in RootStrap(minimax) and TreeStrap(minimax). In TreeStrap(\( \alpha \beta \)) depth \( D = 6 \).

We use two different initial weights. One is to assign all the weights as 1.0, which means we use the original weight of engine Viper. We use \( \theta_{1.0} \) to denote this initial weight. Another weight is just a vector of small random weight in the range \((-5.0, 5.0)\), denoted by notation \( \theta_{\text{rand}} \).

The training will apply each algorithm above to run a training. Each training plays 8000 games, and we record the weight in every 2000 games, so that we can draw the curve of learning effects with respect to the number of training games.

The weight after training will be store in a configuration file. We revise the Viper so that it would load the configuration file firstly when start running. As a result, different weight will correspond to different engines at all. That is how Viper use these weights.

---

5 Results

5.1 Testing Methodology

It is a little complex to test the strength of a chess engine. We run two tournaments to test the strength of the engines, one tournament for each initial weight. Since there are will be 12 engines in each tournament with different weights, because we have three training algorithms, and for each algorithms, we also have four results corresponding to the training after 2000, 4000, 6000 and finally 8000 games. That’s why we have a number of 12 weights in the tournament.

We use the software cutechess-cli to run these two tournaments on the Linux cluster system, so that they can be done in parallel. For each tournament, we run more than 500 games, and store all the games in a PGN file. We use a free available software BayesElo to calculate the ELO score of each engine with different weight. The initial score for the tournament $\theta^{1.0}$ is set to be 2500, and the initial score of tournament $\theta^{rand}$ is set to be 2000. Notice that ELO score can only be used to compare the strength of different chess players in the same tournament. It is meaningless to compared the score in different tournaments.

5.2 Data and Forms

The ELO of engines that applying the weights after 8000 training games

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>ELO in Tournament $\theta^{1.0}$</th>
<th>ELO in Tournament $\theta^{rand}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RootStrap($\text{minimax}$)</td>
<td>2661</td>
<td>1592</td>
</tr>
<tr>
<td>TreeStrap($\text{minimax}$)</td>
<td>2579</td>
<td>1710</td>
</tr>
<tr>
<td>TreeStrap($\alpha\beta$)</td>
<td>2240</td>
<td>1792</td>
</tr>
<tr>
<td>Untrained</td>
<td>2500</td>
<td>1092</td>
</tr>
</tbody>
</table>

The $x$ coordinate denotes the number of training games, while $y$ coordinate denotes the corresponding ELO score. Notice that it is not the number of games in the tournament. These algorithm is different from the the previous ones, because its training games and testing game are total separated. The only task for testing to check the exact strength when engine apply corresponding weight.

6 Conclusions and Future Works

From the results we have seen that, although RootStrap and TreeStrap algorithm can improve the strength of the program reasonably from a random initial weight, it does not perform so well when starting from a strong initial weight. Here we have notice some disadvantage or unclear points of RootStrap and TreeStrap algorithms. All the points below are my own points.

- **Bad performance on strong initial weight.** The performance of the algorithm is not so good as we expected when given a strong initial weight. A possible explanation is that the algorithm may suffer from the problem of local optimal, but we still not sure about that.
Fig. 5. The ELO scores of different algorithms in the tournament $\theta_{1.0}$.

Fig. 6. The ELO scores of different algorithms in the tournament $\theta_{\text{rand}}$. 
• **Did not exploit the game results of each trajectory.** The notice that we run for 8000 games during the training, however the algorithms did not exploit these game results. It leads to another question about training: is this algorithm really care about whether the training states is belong to a same game or not? Could we randomly pick a same number of states to run algorithms and get the same result?

• **Unclear condition of convergency.** The algorithms do not have a clear condition of convergency. Currently we only know that the performance of these algorithms can be improved with increasing number of training games. But, what is the upper bound of number of training games? The author of paper did not describe this clearly.

• **Misleading to the update due to the different search depth.** In TreeStrap algorithm, the nodes in search tree may get their $V^D(s)$ with different depth search. For example, the value of root node is returned by a $D$ search, while a node near the leaf may get their value with only 1 or 2 depth of search, which is much less inaccurate compared to the root node. In addition the number of these inaccurate nodes is much more their parent, because the number of nodes increases exponentially with respect to the depth. Dose this causes a misleading to the updating that $H_\theta(s_t) \leftarrow V^D(s_t)$?

The future works may concentrate on solving some of these potential problems in these two algorithms.

**References**


