Piece Square Tables Factorisation

Abstract

This document describes a method of lossy Piece Square Table (PSQT) compression with the goal ofobtaining a solution that uses fewer *tokens* than other existing methods while maintaining engine's ability to play at a competitive level. The compression is based on matrix factorisation with binary components and is obtained using genetic algorithm.

1 Introduction

The main idea behind this method stems from the fact that piece square tables share some similarities that are independent of the piece type and game phase they are tied to. For example pawns are given bonus for reaching 6th and 7th rank regardless of the phase the game is in. During endgame that bonus is greater but we are mainly interested in the fact that it exists and is applied to the same squares. Similarly pieces are often penalised for being placed on the edges of the board.

These similarities can be described using binary *components* which in turn can be represented using bitboards. For example a bitboard with only the bits corresponding to the 6th and 7th rank set, represents a component that can be used to reward pawns for reaching those ranks.

The idea is to perform an element-wise linear combination of these components to obtain an approximation of a given table. While the components are shared among all tables/pieces the weights used in the linear combination are piece-specific.

The challenge is to find the optimal structure of components, values of weights and the number of both so as to minimise the number of literals needed to store the data as well as to minimise the Elo loss of an engine using that approximation.

2 Method

2.1 Formal Description

To formalise the ideas introduced in the previous section I introduce the following definitions

- $\mathcal{P}$ – the set of target piece square tables (matrices of size 8 by 8)
- $P \in \mathbb{Z}^{64\times 12}$ – matrix representing the tables from $\mathcal{P}$ where each column $P_{:,i}$ is a matrix flattened to a column vector representing the $i-th$ table
- $q_i \in \mathbb{Z}$ – piece value of the i-th piece.
- $c_i \in \{0,1\}^{64}$ – the i-th component represented using a binary vector of size 64.
- $C \in \{0,1\}^{64\times n}$ – binary component matrix such that $C_{:,i} = c_i$ where n is the number of components
- $w_i \in \mathbb{Z}^n$ – the i-th vector of weights corresponding to the i-th piece.
- $W \in \mathbb{Z}^{n\times 12}$ – weight matrix such that $W_{:,i} = w_i$.

Using these definitions we can write the approximation of P as matrix multiplication of C and W

$$P \approx \hat{P} = CW \tag{1}$$

Our goal is to solve the following task

$$\arg\min_{C,W} \left\| P - \hat{P} \right\|_F^2 \tag{2}$$

where $\|\cdot\|_F$ is Frobenius norm. I chose this loss metric because it's simple, intuitive and has good mathematical properties.

2.2 Solving

2.2.1 Matrix Factorisation with Binary Components

At first I've tried finding the matrices C and W using the algorithm described in [3]. It seemed perfect for the problem but I found that because of the limitations of the algorithm it could be outperformed by my own solution which is described in the following section.

2.2.2 Genetic Algorithm

To solve the minimisation problem I've used *genetic algorithm* which simulates generations of specimens to imitate evolution using a specified fitness function to select best specimens.

Each specimen s contains *genes* composed of matrices $\tilde{C}$ and $\tilde{W}$ which are contenders to become C and W. During every generation each specimen s_i is evaluated using a fitness function

$$f(s_i) = -\left\|P - \tilde{C}_i\tilde{W}_i\right\|_F^2 \tag{3}$$

The minus sign represents the fact that evolution algorithms in general try to *maximise* the fitness function. The greater the fitness function value the better is the specimen.

A new generation is created by maintaining top k specimens from the previous generation (to keep the current best solution) and creating a set of new specimens. Each new specimen is created by selecting a parent (a specimen from the previous generation) at random (favouring best performing specimens) and slightly mutating it.

Mutation consists of flipping an element inside $\tilde{C}$ from 0 to 1 or vice versa and/or increasing/decreasing the value of a random element inside $\tilde{W}$ by a small amount.

After many generations the specimens eventually converge to a local maximum of the fitness function (minimum of our loss function). At that moment $\tilde{C}$ and $\tilde{W}$ are selected as C and W.

2.3 Additional Considerations

2.3.1 Importance Mask

One way we can improve the solutions is to realise that some values inside P are not important in any way. Specifically the values of the 1st and the 8th rank for pawns do not influence the evaluation since pawns can never be placed there. We can represent this fact but creating an importance mask $M \in \{0,1\}^{64\times 12}$ which consists entirely of ones except for elements corresponding to the aforementioned squares which are set to zero. We can then rewrite the fitness function as

$$f(s_i) = -\left\|M \odot (P - \tilde{C}_i\tilde{W}_i)\right\|_F^2 \tag{4}$$

where $\odot$ represents element wise multiplication.

We can extend the use of this importance mask by altering its values. If, for example, we want the approximation to be better for rooks we can simply set all the elements corresponding to rook values so something greater than one. I have not tested how this affects the final result Elo-wise however I think it's an interesting thing to consider.

2.3.2 Iterative Component Generation

To ensure good convergence the mutation rate inside a genetic algorithm should be kept relatively low. However in this case it can lead to situation in which several components evolve similarly.

This results in sub-optimal solutions because simply removing on of the identical components and summing the corresponding weight vectors would result in the same approximation with less data.

To address this problem I perform the genetic algorithm in several iterations. At iteration $i \in [n]$ specimens only have i components and $12 \cdot i$ weights to optimise. After the evolution converges we can advance to the next iteration by adding a new component and a new set of weights both consisting of only zeros.

In practice this approach, while slower, leads to much better results.

2.3.3 Mean Adjusted PSQTs

Another thing I've implemented is mean adjusted PSQTs values. For each table $p_i \in \mathcal{P}$ and corresponding piece value q_i I calculate

$$\bar{p_i} = \lfloor \frac{\sum_{j,k=1}^{j,l=8} p_{i_{jk}}}{64} \rceil \tag{5}$$

and update the values

$$p_i \leftarrow p_i - \bar{p_i}$$
$$q_i \leftarrow q_i + \bar{p_i}$$

The logic behind this method is that we do not want factorise the bias of a given table.

I have not tested this approach thoroughly but it seems to lead to slightly better results.

2.4 MSE of Win Probability as Fitness Function

Apart from the previously mentioned fitness function we can define a different one that is more closely related to chess. It's based on calculating the winning probability in many (500 000) positions and comparing the results obtained using the target PSQT and the approximation. Win percentage for a given position (board state) B using PSQT P was calculated using the following formula [1]

$$W_P(B) = \frac{1}{1 + 10^{-A(B)/4}} \tag{6}$$

where $A(B)$ is pawn advantage in that position.

After calculating the win probability of many positions (board states) using the target PSQT we can use the following fitness function

$$f(s_i) = -\frac{1}{N}\sum_{j=1}^{N}(W_P(B_j) - W_{\hat{P}}(B_j))^2 \tag{7}$$

However, in practice, this approach is much slower since we have to evaluate many positions for many specimens across thousands of generations.

One way to still use Win Probability based fitness function but maintain the speed of Frobenius Norm based fitness function is to generate many solutions using the latter one and pick the solution with the maximum value of the former one (bearing in mind that minus sign in front of the function).

I think this approach is useful because from my tests lower MSE does not necessarily translate to lower Elo loss and MSE of win percentage seems to to be a better predictor of that. However I did not have the time to test this enough so I can't say anything for sure.

2.5 Optimal n

The last thing to solve is to find the optimal n i.e. the optimal number of components so that we save tokens by using few literals but get a good approximation of P at the same time.

I have no good theoretical approach to this problem but from many rounds of testing I've found that $12 < n < 16$ is probably the best value. For values less than 12 the Elo loss was too big to be justified and for values greater than 16 I've encountered quickly diminishing returns.

2.6 Packing/Unpacking

Finally there is the issue of packing and unpacking the data. Apart from C and W we also need to include q to fully reconstruct the value of any given piece on any given square.

We can leverage the fact that the chess board has 64 squares so each component can be represented as a bitboard i.e. single `ulong`. Similarly a `decimal` can store 12 useful bytes of data. This means that if we limit our weights to only take up 8 bits of memory we can use one decimal to store all the weights corresponding to a given component. In practice the weights are in range $[-128, 127]$.

To store q we can divide the values by $d \in \mathbb{N}$ so that they fit in the specified weight range. Then we can encode them as another set of weights with a component that is a matrix of size $64 \times n$ and contains only d. Since for $d > 1$ we cannot represent this component using a bitboard we need to declare it implicitly. I achieve this by using a ternary operator that checks if the index of a given weight set is $n + 1$ in which case I multiply the weight by d. Otherwise I multiply the weight by the corresponding bit inside the component.

The code snippet for unpacking looks as follows

```
UnpackedPestoTables = Enumerable.Range(0, 768).Select(i =>
    weights.Select((w, j) =>
        (sbyte)Buffer.GetByte(decimal.GetBits(w), i / 64) * (j == n ? d :
        (int)((components[j] >> i % 64) % 2))
    ).Sum()).ToArray();
```

where `weights` and `components` are arrays of respectively decimals and ulongs of size n.

3 Results

Using $n = 14$ (29 literals) I was able to find matrices C and W such that the engine using approximated PeSTO PSQTs [2] was comparable to the engine using lossless compression.

Running an SPRT between the two engines resulted in about -4 Elo difference. Overall the method is 27 tokens less expensive than other compression methods.

For the chosen 14 component solution the mathematical properties of the approximation are present in Table 1 A visualisation of the target PSQT (PeSTO) and the

Property	Value
Max Absolute Error	30
Mean Absolute Error	5.79
Mean Squared Error	59.35
Mean Relative Error	0.021
MSE of Win Probability	0.0013
Accuracy of WDL Prediction	0.9335

Table 1: Properties of the PeSTO Approximation

obtained approximation can be found in, respectively, Figure 1 and 2. Additionally,

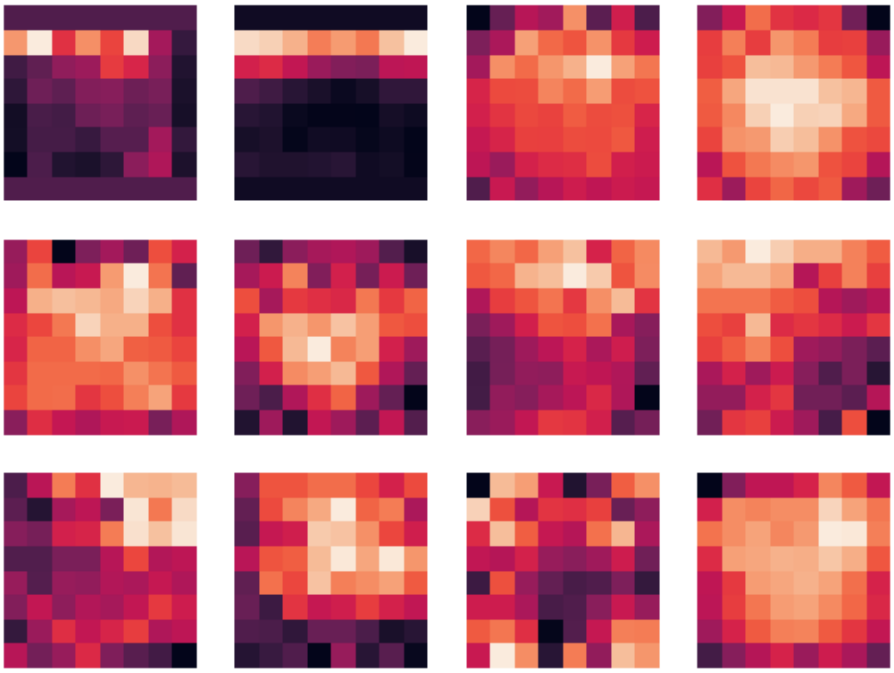

Figure 1: PeSTO Tables

a visualisation of the matrix multiplication $\hat{P} = CW$ can be seen in Figure 3

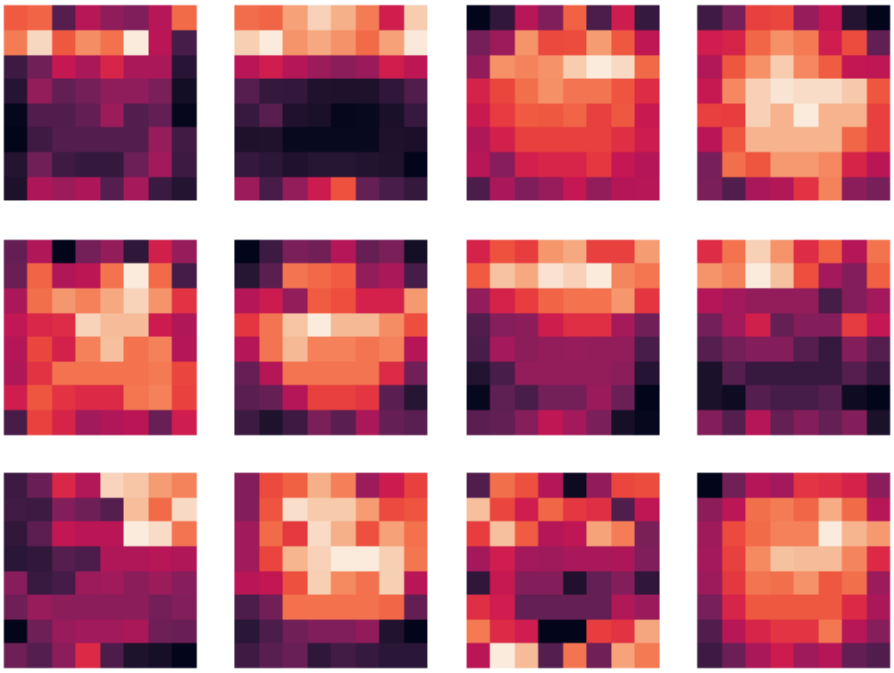

Figure 2: Approximated PeSTO Tables

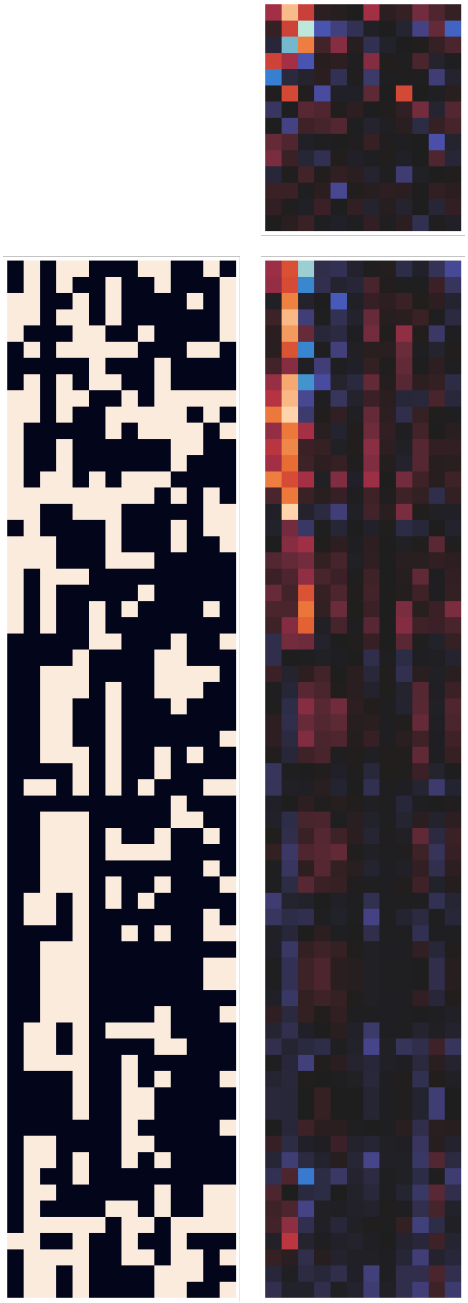

Figure 3: Visualisation of matrix multiplication $\hat{P} = CW$. $\hat{P}$ is the central matrix. C and W are respectively the matrix on the left and the matrix on top of $\hat{P}$.

References

[1] Pawn advantage, win percentage and elo. https://www.chessprogramming.org/Pawn_Advantage,_Win_Percentage,_and_Elo.

[2] Pesto's evaluation function. https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function.

[3] Martin Slawski, Matthias Hein, and Pavlo Lutsik. Matrix factorization with binary components, 2014.