Evaluation & Tuning in Chess Engines

Andrew Grant

Abstract

Ethereal implements a modernized version of Peter Österlund's Texel Tuning using Gradient Decent techniques. In this paper we outline evaluation functions for chess engines, beginning with the simplest form and ending with real outlines from modern alpha-beta engines. We then derive the equations needed to compute the gradient of an engine's evaluation for linear terms as well as groupings of evaluation features which can be described as a linear set of operations later adjusted by a nonlinear function. These nonlinear functions include making use of quadratic terms, $min()$, $max()$, and $sign()$. Finally, we outline a highly-optimized method of implementing this tuning schema by performing updated evaluations without having to run an engine's evaluation function. The result is a tuning method capable of tuning thousands of evaluation weights in tandem with extreme speed and elo-gaining precision.

1 General Evaluation Components

1.1 Evaluation Purpose

In the ideal case, every leaf node of an engine's search tree would be a terminal node, corresponding to a win, loss, or draw. The Shannon number, 10^{120}, is a conservative lower bound on the necessary size of the search tree from the start of the game needed in order to have all leaf nodes be terminal. This number far exceeds the number of atoms in the universe. Thus, the majority of nodes in the search tree are not terminal, and must be handled via a static evaluation function. This function converts the board to an integer value, where a positive value indicates an advantage for white.

Evaluation functions are often extremely naive. Strong evaluation functions will consider threats and minor tactical plays, but by design the evaluation is not a search. An extreme example would be a position where White's Queens is left en prise. The evaluation may note the hanging piece and apply a small malus, but will not understand that the Queen is lost. This shortsightedness of the evaluation is inevitable. However, we still aim to maximize the correlation between the static evaluation and the eventual outcome of the game by identifying features of a position and trying to map those features to the eventual outcome.

1.2 Basic Evaluation

A basic evaluation can be described as a vector of linear weights $\vec{L}$, containing each evaluation feature, and two vectors of coefficients $\vec{C_w}, \vec{C_b}$, for White and Black. The most well known evaluation feature is that which counts the number of each type of piece on the board. Most engines also implement a PSQT (Piece-Square Table), which has a value for placing a particular piece on any given square. The number of potential terms are unlimited. Ethereal has approximately 650 evaluation terms that fall under a basic evaluation. An evaluation of a position, from White's perspective, can be written as follows ...

$$E = \vec{L} \cdot (\vec{C_w} - \vec{C_b})$$

1.3 Phased Evaluation

It is common knowledge that some features of a chess board are more important in one stage of the game than in another. Thus most engines define two values for each term. One for the midgame, and another for the endgame, typically expressed as (mg, eg). A well known method for interpolating the evaluation between stages is to define a phasing schema based on the remaining material on the board.

Ethereal's method to compute the phase, ρ, is derived directly from Fruit ...

$$\lambda = [1, 1, 2, 4] \cdot [N, B, R, Q]$$
$$\rho = \left\lceil \frac{256(24 - \lambda)}{24} \right\rceil$$

Given an evaluation of both phases, E_{mg} and E_{eg}, we compute a phased evaluation. It is common to scale the evaluation of the endgame using a Scale Factor ξ . Typically in an engine, ξ ranges from 0 to 1. ξ is used to handle special cases, like opposite coloured bishop (OCB) endgames, unwinnable material configurations, and more. In most positions, ξ is very close if not exactly 1. Ethereal will use a value of ξ over 1 for some heavily won positions.

$$E = \frac{1}{256}((256 - \rho)E_{mg} + \rho\xi E_{eg})$$

1.4 King Safety Evaluation

Engines often perform non-linear operations when computing King Safety. In some cases the King Safety function can be described as a set of linear operations, with a final adjusting function. Let $S = \vec{L_s} \cdot \vec{C_s}$, which is the pre-finalized value for Safety for a player. $\vec{L_s}$ is the set of linear weights that impact Safety, and $\vec{C_s}$ are the corresponding coefficients for some player. Ethereal's new evaluation is as follows ...

$$f_{mg}(x) = \frac{-x}{720} \cdot max(0, x), f_{eg} = \frac{-1}{20} \cdot max(0, x)$$
$$E_{mg} = \vec{L_{mg}} \cdot (\vec{C_w} - \vec{C_b}) + f_{mg}(S_w) - f_{mg}(S_b)$$
$$E_{eg} = \vec{L_{eg}} \cdot (\vec{C_w} - \vec{C_b}) + f_{eg}(S_w) - f_{eg}(S_b)$$

1.5 Complexity Evaluation

Stockfish may have been the first to use what is called Complexity. Complexity adjusts the evaluation based on how dynamic the structure of the board is. In Ethereal we look at remaining pawns, whether there are pawns on both flanks, and whether we are in a pawn endgame. In a position in which White is deemed winning, complexity may not revise the evaluation to have Black winning. At most, the position may be dampened to a draw. Complexity can be described as a dot product of coefficients and linear weights, adjusted with a final function, just like King Safety.

We can define the Complexity $\Gamma = \vec{L_c} \cdot \vec{C_l}$, the linear weights for Complexity dotted with their coefficients. Usually an engine will only adjust the evaluation for the endgame component via Complexity, not the midgame. The following denotes the adjusted endgame evaluation for a position ...

$$E'_{eg} = E_{eg} + sign(E_{eg}) \cdot max(-|E_{eg}|, \Gamma)$$

2 Peter Österlund's Texel Tuning

2.1 Original Methodology

Peter's original method took a sample of 64,000 games played between various versions of his engine. He extracted each position from each game and filtered out those with a reported mate score. He defined the following sigmoid, aimed at mapping an engine's evaluation to $[0, 1]$, where 0 indicates a win for black, 0.5 indicates a drawn game, and 1.0 indicates a win for white ...

$$\sigma = \frac{1}{1 + 10^{\frac{-KE}{400}}}$$

Here E is an evaluation, and K is a coefficient computed to minimize an error function. A method of computing K will be described later. Peter used a White-relative quiescence search score in his error function, not the actual evaluation, in order to minimize the error of tactical positions. He defined his error function as follows ...

$$\epsilon = \frac{1}{N} \sum_{i=0}^{N} (R_i - \sigma(Q_i))^2$$

R_i refers to the end result of the position and Q_i refers to an evaluation of the position. Peter would loop over the set of evaluation weights and adjust them slightly. He would then recompute an error using the original value of K. If a new set of weights produced a lower error the result was saved. This approach is extremely naive, yet still effective. Peter reported nearly 100 elo over the course of 7 tuning sessions.

2.2 Refined Dataset Generation

For Ethereal, using a large number of positions from the same game has been shown to produce a lower quality dataset. Ethereal's early dataset consisted of the commonly shared Zurichess dataset(≈1.6m), a set of positions found by randomly exploring search trees in Jeffrey and Michael An's Laser (≈1.3m), and positions sampled from Ethereal's self-play games on the OpenBench framework (≈4.9m). This dataset was the basis for dozens of training sessions, both full and partial, which resulted in hundreds of elo.

Currently, Ethereal's dataset is built by sampling positions from a set of one million self-play games at hyper-bullet time controls (1s+.01s) with aggressive adjudication settings. From there, we perform a high depth search on every one of the positions. Those high depth searches produce a principle variation. We take the original position, apply all the moves in the principle variation, and then save that as our final position for the dataset.

This method has many advantages, both from a dataset strength perspective, and from a mathematical perspective in regards to the alternatives. Note before that Peter would use a quiescence search evaluation. If we instead would have used a high depth search evaluation, we would further reduce the error rate of the dataset and improve the quality of training.

However, the ideal conditions found thus far are to still use a static evaluation, without playing out a quiscence search, so long as the positions in the dataset are already the result of following high-depth search principle variations to completion. This gets the many benefits of improving the dataset, like removing positions which appear drawn but are clearly won when searched. This also maintains the mathematical correctness of tuning the true static evaluation of each position in the dataset.

2.3 Sigmoid Modifications

In order to simplify the maths behind computing the gradient when tuning, the Sigmoid was swapped to base e. This removes the need to carry around $ln(10)$ coefficients throughout the derivations. In practice, this adjusts the value of K in expected relation to $\frac{10}{e}$. There is no reason to believe that using base e carries any significance by shaping the Sigmoid. For further simplicity, the $\frac{1}{400}$ that appears in Peter's Sigmoid can be absorbed by the coefficient K. We define a new Sigmoid for Ethereal...

$$\sigma = \frac{1}{1 + e^{-KE}}$$

2.4 Error Function Variants

Peter used a Mean Squared Error (L2 Loss) function in his tuning. Ethereal has tried using a Mean Absolute Error (L1 Loss) function with no success. This is likely due to the inaccuracy of evaluation functions of all chess engines. Even the best engines are limited in their resolution or grain of their evaluations. For Ethereal, the smallest unit of evaluation is approximately one-hundredth of a pawn. Stronger engines like Stockfish have managed to use up to two-hundredths of a pawn. Despite a lack of success, the maths for applying Ethereal's Gradient Descent method follow naturally with an L1 Loss function.

3 Gradient Descent Tuning Derivations

3.1 Basic Overview and End Goals

We are looking to compute partials with respect to the error ϵ for all of our linear weights. We can break this into multiple steps to further isolate the complications of differentiating non-linear evaluation terms. Recall that all weights $\vec{L}$ have both an (mg, eg) component. Some of these derivations are already well known, but for completeness we will recompute all of them here. The final partial derivative below is the most complex and dynamic. We will compute partials for both game phases, for normal, safety, and complexity terms.

$$\frac{\partial \epsilon}{\partial \vec{L}_i} = \frac{1}{N} \sum_{j=0}^{N} \frac{d}{d\sigma}(R_j - \sigma(E_j))^2 \frac{d\sigma}{dE_j} \frac{\partial E_j}{\partial \vec{L}_i} = \frac{-2}{N} \sum_{j=0}^{N} (R_j - \sigma(E_j)) \frac{d\sigma}{dE_j} \frac{\partial E_j}{\partial \vec{L}_i}$$

Additionally, we note the coefficients that appear before E_{mg} and E_{eg} in the interpolated evaluation are as follows. We will reuse these constant definitions throughout the derivations for each evaluation type.

$$\rho_{mg} = \frac{256 - \rho}{256}, \rho_{eg} = \frac{\xi \rho}{256}$$

3.2 Derivative of the Sigmoid

$$\begin{aligned}
\frac{d\sigma}{dE} &= \frac{d}{dE}\left(\frac{1}{1 + e^{-KE}}\right) = \frac{Ke^{-KE}}{(1 + e^{-KE})^2} = \frac{(e^{KE})^2}{(e^{KE})^2} \frac{Ke^{-KE}}{(1 + e^{-KE})^2} \\
&= \frac{Ke^{KE}}{(1 + e^{KE})^2} = \frac{K}{(1 + e^{KE})} - \frac{K}{(1 + e^{KE})^2} = K\sigma(E)(1 - \sigma(E))
\end{aligned}$$

3.3 Derivative of Linear Weights without Complexity

Gradient's for trivial evaluation functions of linear weights follow easily. Special case is taken to note the difference between phases. Even though a pair of weights (mg, eg) appear to have relation to each other, they must be treated separately due to ρ and ξ

$$\frac{\partial E}{\partial \vec{L}_{i,mg}} = \rho_{mg}(\vec{C}_{i_w} - \vec{C}_{i_b}), \frac{\partial E}{\partial \vec{L}_{i,eg}} = \rho_{eg}(\vec{C}_{i_w} - \vec{C}_{i_b})$$

3.4 Derivative of Linear Weights with Complexity

The addition of complexity, which only impacts the endgame, adds an additional dependence on the overall evaluation. The complications of Complexity are needed in order to avoid situations where we would increase an eval term without noting that Complexity would have negated the increase. When $E_{eg} = 0$ the derivative is undefined, but also Complexity is not impacted in this case.

$$\frac{\partial E}{\partial \vec{L}_{i,mg}} = \rho_{mg}(\vec{C}_{i_w} - \vec{C}_{i_b})$$

$$\frac{\partial E}{\partial \vec{L}_{i,eg}} = \begin{cases} \rho_{eg}(\vec{C}_{i_w} - \vec{C}_{i_b}), & E_{eg} = 0 \\ \rho_{eg}(\vec{C}_{i_w} - \vec{C}_{i_b}), & |E_{eg}| + \Gamma \geq 0 \\ 0, & otherwise \end{cases}$$

3.5 Derivative of King Safety Weights without Complexity

For a given player, if $S = 0$, the derivative of the King Safety in the endgame is undefined. However, in this case King Safety would not be applied. We hand-wave this inconsistency, just like with linear weights and complexity. Alternatively, $f_{eg}(S)$ could easily be modified to remove the $min()$ with minimal elo change.

$$\frac{\partial E}{\partial \vec{L}_{i,mg}} = \frac{\partial E}{\partial \vec{L}_{i,mg_w}} + \frac{\partial E}{\partial \vec{L}_{i,mg_b}} = \frac{\rho_{mg}}{360}\left(max(S_b, 0)\vec{C}_{i_b} - max(S_w, 0)\vec{C}_{i_w}\right)$$

$$\frac{\partial E}{\partial \vec{L}_{i,eg}} = \frac{\partial E}{\partial \vec{L}_{i,eg_w}} + \frac{\partial E}{\partial \vec{L}_{i,eg_b}} = \frac{\rho_{eg}}{20}\left(max(sign(S_b), 0)\vec{C}_{i_b} - max(sign(S_w), 0)\vec{C}_{i_w}\right)$$

3.6 Derivative of King Safety Weights with Complexity

Once again the nuance of $S = 0$ applies. However, this time modifying $f_{eg}(S)$ is not enough to fully fix the maths, as Complexity has an undefined derivative when $E_{eg} = 0$

$$\frac{\partial E}{\partial \vec{L}_{i,mg}} = \frac{\partial E}{\partial \vec{L}_{i,mg_w}} + \frac{\partial E}{\partial \vec{L}_{i,mg_b}} = \frac{\rho_{mg}}{360}\left(max(S_b, 0)\vec{C}_{i_b} - max(S_w, 0)\vec{C}_{i_w}\right)$$

$$\frac{\partial E}{\partial \vec{L}_{i,eg}} = \frac{\partial E}{\partial \vec{L}_{i,eg_w}} + \frac{\partial E}{\partial \vec{L}_{i,eg_b}} = \begin{cases} \frac{\rho_{eg}}{20}\left(max(sign(S_b), 0)\vec{C}_{i_b} - max(sign(S_w), 0)\vec{C}_{i_w}\right) & E_{eg} = 0 \\ \frac{\rho_{eg}}{20}\left(max(sign(S_b), 0)\vec{C}_{i_b} - max(sign(S_w), 0)\vec{C}_{i_w}\right) & |E_{eg}| + \Gamma \geq 0 \\ 0 & otherwise \end{cases}$$

3.7 Derivative of Complexity Weights

Complexity is only applied in the endgame, so no computation is done for the midgame.

$$\frac{\partial E}{\partial \vec{L}_{i,eg}} = \begin{cases} sign(\Gamma)\vec{C}_i\rho_{eg} & |E_{eg}| + \Gamma \geq 0 \\ 0 & otherwise \end{cases}$$

3.8 Sources of Error

1. As noted before, the derivative of both Linear and King Safety weights when $E_{eg} = 0$ is undefined due to the Complexity computation.

2. As noted before, the derivative of King Safety in the endgame when $S = 0$ is undefined. This could be fixed by adjusting our definition of $f_{eg}(S)$. Namely, we could remove the $max()$. This has been shown to be elo neutral in Ethereal. It has the added effect of introducting a bonus to a given side when they are safe.

3. If ξ depends on E_{eg} or E_{mg} in any way, then ξ is subject to change and is no longer considered a constant. In Ethereal this can happen in a number of cases. Those cases are likely negligible. At worst, aside from the case where $\xi = 0$ (which is a case where we might want to remove the position from the dataset), the impact of an incorrect derivation is equal to a slight adjustment in the learning rate.

4 Gradient Descent Tuning Implementation

4.1 Evaluation Wrapping

In order to implement the tuner, every application of an evaluation weight in the evaluation routine is followed by a tracing mechanism which updates a vector of applied coefficients. After running the evaluation, one should be able to recompute the evaluation with only $\vec{L}$, $\vec{C_w}, \vec{C_b}$, ρ, and ξ, which should be readily available without remembering the actual structure of the position.

Ethereal will collect the coefficients for White and Black for all evaluation terms applied. Ethereal saves the S values for both White and Black before applying f_{mg} or f_{eg}. Ethereal saves the scale factor ξ. Ethereal saves the evaluation prior to the application of Complexity. Finally, Ethereal saves the Complexity term which would be applied, if not for the restriction on changing the sign of the evaluation.

As a final note, before we present any code, we observe that in the vast majority of cases $C_w = C_b$. Any linear weights where this is true do not need to be adjusted for a given position. Complexity and Safety are an exception, as we still need the coefficients in all cases. Instead of saving all coefficients, which for Ethereal is over 1300, we only save the non zero ones, the Complexity ones, and the Safety ones. We now do vector operations on Skip Vectors. This saves orders of magnitude in memory consumption, which results in an order of magnitude in speed dude to memory access bottlenecks.

4.2 General Structure

The following outlines the general structure, which is a massive array of TEntry structs, which contain enough information to recompute the original evaluation, as well as computed an updated-evaluation based on the adjusments made to the evaluation weights. The eval, safety, and complexity terms in the TEntry struct are the usual encoded form of $S(mg, eg)$. We let $X_{mg} = ScoreMG(X)$ and $X_{eg} = ScoreEG(X)$.

```
// A TTuple is used for all coefficients that are non-zero or specific to the Safety or
// Complexity evaluations. An array of TTuples acts as a Skip-Vector or Skip-Matrix. In
// practice, smaller datatypes are used than integers to save Memory.

typedef struct TTuple { int index, wcoeff, bcoeff; } TTuple;


// seval      > A Static Evaluation, used to compute the optimal K value
// phase      > The phase coefficent rho that is defined in (1.3)
// turn       > Side to move of the position. Not used in this paper.
//
// eval       > A evaluation of the position before applying complexity written as (mg, eg)
// safety     > A Safety score, pre-adjusted, for both players, written as (mg, eg)
// complexity > An unclamped Complexity score, written as (mg, eg)
//
// result     > The result of the game this position came from. [0.0, 0.5, 1.0]
// sfactor    > The scale factor xi used to adjust the Endgame evaluation
// pfactors   > The rho_mg and rho_eg that are defined in (3.1)
//
// tuples     > Array of TTuples for all needed coefficients
// ntuples    > Length of the above tuples Array

typedef struct TEntry {
    int seval, phase, turn;
    int eval, safety[COLOUR_NB], complexity;
    double result, sfactor, pfactors[PHASE_NB];
    TTuple *tuples; int ntuples;
} TEntry;


// Some trivial renamings for simplicty

typedef int TArray[NTERMS];
typedef double TVector[NTERMS][PHASE_NB];
enum { NORMAL, COMPLEXITY, SAFETY, METHOD_NB };
```

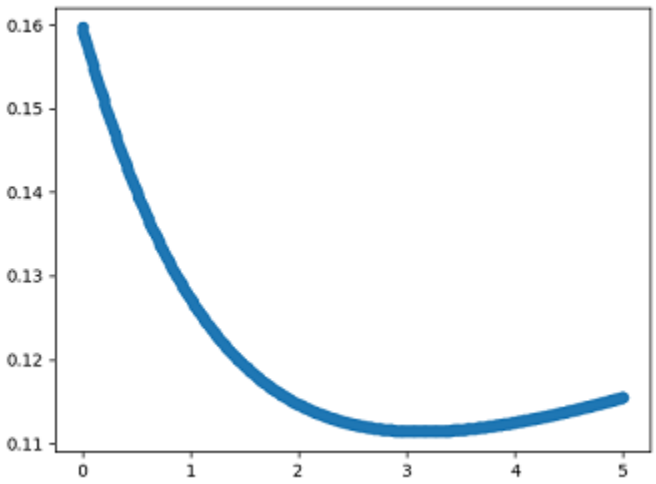
4.3 Initializing the Tuner Entries

Ethereal uses its own internal memory mamagement for allocationg TTuples, in order to ensure better read times by minmizing cache misses. That implemenation can be found in the source files tuner.c and tuner.h. In the following implementation we will remove some Ethereal specific things, in order to provide as close to a minimal implementation as possible. The TArray is used to track the type of each term. The possible term types are defined in an enum above, as being NORMAL, COMPLEXITY, or SAFETY. We do this so that we can lump all the tuned terms in together for simplicity.

```
void initTunerEntries(TEntry *entries, TArray methods) {

    Board board;
    char line[256];
    FILE *fin = fopen(TUNINGFILE, "r");

    for (int i = 0; i < NPOSITIONS; i++) {

        if (fgets(line, 256, fin) == NULL) exit(EXIT_FAILURE);

        // Find the result { W, L, D } => { 1.0, 0.0, 0.5 }
        if      (strstr(line, "[1.0]")) entries[i].result = 1.0;
        else if (strstr(line, "[0.0]")) entries[i].result = 0.0;
        else if (strstr(line, "[0.5]")) entries[i].result = 0.5;

        // Set the board with the current FEN and initialize
        boardFromFEN(board, line);
        initTunerEntry(&entries[i], board, methods);
    }
}

void initTunerEntry(TEntry *entry, Board *board, TArray methods) {

    // Use the same phase calculation as evaluate()
    int phase = 24 - 4 * popcount(board->pieces[QUEEN ])
                   - 2 * popcount(board->pieces[ROOK  ])
                   - 1 * popcount(board->pieces[BISHOP])
                   - 1 * popcount(board->pieces[KNIGHT]);

    // Save time by computing phase scalars now
    entry->pfactors[MG] = 1 - phase / 24.0;
    entry->pfactors[EG] = 0 + phase / 24.0;
    entry->phase = (phase * 256 + 12) / 24;

    // Save a white POV static evaluation
    TVector coeffs; T = EmptyTrace;
    entry->seval = evaluateBoard(board);
    if (board->turn == BLACK) entry->seval = -entry->seval;

    // evaluate() -> [[NTERMS][COLOUR_NB]]
    initCoefficients(coeffs);
    initTunerTuples(entry, coeffs, methods);

    // Save some of the evaluation modifiers
    entry->eval        = T.eval;
    entry->complexity  = T.complexity;
    entry->sfactor     = T.factor;
    entry->turn        = board->turn;

    // Save the Linear version of King Safety
    entry->safety[WHITE] = T.safety[WHITE];
    entry->safety[BLACK] = T.safety[BLACK];
}

void initTunerTuples(TEntry *entry, TVector coeffs, TArray methods) {

    int length = 0, tidx = 0;

    // Count the needed Coefficients
    for (int i = 0; i < NTERMS; i++)
        length += (methods[i] == NORMAL &&  coeffs[i][WHITE] - coeffs[i][BLACK] != 0.0)
               || (methods[i] != NORMAL && (coeffs[i][WHITE] != 0.0 || coeffs[i][BLACK] != 0.0));

    // Allocate space for new Tuples (Normally, we don't malloc())
    entry->tuples   = malloc(sizeof(TTuple) * length;
    entry->ntuples  = length;

    // Finally setup each of our TTuples for this TEntry
    for (int i = 0; i < NTERMS; i++)
        if (   (methods[i] == NORMAL &&  coeffs[i][WHITE] - coeffs[i][BLACK] != 0.0)
            || (methods[i] != NORMAL && (coeffs[i][WHITE] != 0.0 || coeffs[i][BLACK] != 0.0)))
            entry->tuples[tidx++] = (TTuple) { i, coeffs[i][WHITE], coeffs[i][BLACK] };
}
```

4.4 Computing an Optimal K Value

The follow is a graph of the error rate for Ethereal using various K values on the primary dataset. This loosely confirms the correctness of our optimization method, which just scans over the search space looking for new local mimimums in increasingly smaller spaces with increasingly smaller grains. Ethereal uses a value of 10 for KPRECISION.



K vs ϵ for Ethereal

```
double computeOptimalK(TEntry *entries) {

    double start = 0.0, end = 10, step = 1.0;
    double curr = start, error;
    double best = staticEvaluationErrors(entries, start);

    for (int i = 0; i < KPRECISION; i++) {

        // Find the minimum within [start, end] using the current step
        curr = start - step;
        while (curr < end) {
            curr = curr + step;
            error = staticEvaluationErrors(entries, curr);
            if (error <= best)
                best = error, start = curr;
        }

        // Adjust the search space
        end   = start + step;
        start = start - step;
        step  = step  / 10.0;
    }

    return start;
}


double staticEvaluationErrors(TEntry *entries, double K) {

    // Compute the error of the dataset using the Static Evaluation.
    // We provide simple speedups that make use of the OpenMP Library.

    double total = 0.0;

    #pragma omp parallel shared(total)
    {
        #pragma omp for schedule(static, NPOSITIONS / NPARTITIONS) reduction(+:total)
        for (int i = 0; i < NPOSITIONS; i++)
            total += pow(entries[i].result - sigmoid(K, entries[i].seval), 2);
    }

    return total / (double) NPOSITIONS;
}
```

4.5 Gradient Computation per Position

Computing the Gradient for the evaluation is very time consuming, and has a lot of moving parts. The following code is nearly verbatim. It demonstrates the significant increase in difficulty from computing trivial terms like normal weights in the midgame, to computing safety weights in the endgame. Note that in the following example, the params TVector is NOT the new evaluation weights. It is the difference between the updated evaluation weights and the original evaluation weights. This is needed in order to tune only a subset of the engine's evaluation at any given time. A trick which single-handedly allows for this trivial vector recomputation, as opposed to calling the evaluate() routine again.

```
double linearEvaluation(TEntry *entry, TVector params, TArray methods, TGradientData *data) {

    int sign, mixed;
    double midgame, endgame, wsafety[2], bsafety[2];
    double normal[PHASE_NB], safety[PHASE_NB], complexity;
    double mg[METHOD_NB][COLOUR_NB] = {0}, eg[METHOD_NB][COLOUR_NB] = {0};

    // Save any modifications for MG or EG for each evaluation type
    for (int i = 0; i < entry->ntuples; i++) {
        mg[methods[i]][WHITE] += (double) entry->tuples[i].wcoeff * params[entry->tuples[i].index][MG];
        mg[methods[i]][BLACK] += (double) entry->tuples[i].bcoeff * params[entry->tuples[i].index][MG];
        eg[methods[i]][WHITE] += (double) entry->tuples[i].wcoeff * params[entry->tuples[i].index][EG];
        eg[methods[i]][BLACK] += (double) entry->tuples[i].bcoeff * params[entry->tuples[i].index][EG];
    }

    // Grab the original "normal" evaluations and add the modified parameters
    normal[MG] = (double) ScoreMG(entry->eval) + mg[NORMAL][WHITE] - mg[NORMAL][BLACK];
    normal[EG] = (double) ScoreEG(entry->eval) + eg[NORMAL][WHITE] - eg[NORMAL][BLACK];

    // Grab the original "safety" evaluations and add the modified parameters
    wsafety[MG] = (double) ScoreMG(entry->safety[WHITE]) + mg[SAFETY][WHITE];
    wsafety[EG] = (double) ScoreEG(entry->safety[WHITE]) + eg[SAFETY][WHITE];
    bsafety[MG] = (double) ScoreMG(entry->safety[BLACK]) + mg[SAFETY][BLACK];
    bsafety[EG] = (double) ScoreEG(entry->safety[BLACK]) + eg[SAFETY][BLACK];

    // Remove the original "safety" evaluation that was double counted into the "normal" evaluation
    normal[MG] -= MAX(0, ScoreMG(entry->safety[WHITE])) * -ScoreMG(entry->safety[WHITE]) / 720)
                - MAX(0, ScoreMG(entry->safety[BLACK])) * -ScoreMG(entry->safety[BLACK]) / 720);
    normal[EG] -= -MAX(0, ScoreEG(entry->safety[WHITE]) / 20) + MAX(0, ScoreEG(entry->safety[BLACK]) / 20);

    // Compute the new, true "safety" evaluations for each side
    safety[MG] = MAX(0, wsafety[MG]) * -wsafety[MG] / 720)
               - MAX(0, bsafety[MG]) * -bsafety[MG] / 720);
    safety[EG] = -MAX(0, wsafety[EG] / 20) + MAX(0, bsafety[EG] / 20);

    // Grab the original "complexity" evaluation and add the modified parameters
    complexity = (double) ScoreEG(entry->complexity) + eg[COMPLEXITY][WHITE];
    sign       = (normal[EG] + safety[EG] > 0.0) - (normal[EG] + safety[EG] < 0.0);

    // Save this information since we need it to compute the gradients
    *data = (TGradientData) {
        normal[EG] + safety[EG], complexity,
        wsafety[MG], bsafety[MG], wsafety[EG], bsafety[EG]
    };

    midgame = normal[MG] + safety[MG];
    endgame = normal[EG] + safety[EG] + sign * fmax(-fabs(normal[EG] + safety[EG]), complexity);

    mixed = (midgame * (256 - entry->phase)
          +  endgame * entry->phase * entry->sfactor) / 256;

    return mixed + (entry->turn == WHITE ? Tempo : -Tempo);
}
```

```
void computeGradient(TEntry *entries, TVector gradient, TVector params, TArray methods, double K, int batch) {

    #pragma omp parallel shared(gradient)
    {
        TVector local = {0};

        #pragma omp for schedule(static, BATCHSIZE / NPARTITIONS)
        for (int i = batch * BATCHSIZE; i < (batch + 1) * BATCHSIZE; i++)
            updateSingleGradient(&entries[i], local, params, methods, K);

        for (int i = 0; i < NTERMS; i++) {
            gradient[i][MG] += local[i][MG];
            gradient[i][EG] += local[i][EG];
        }
    }
}

void updateSingleGradient(TEntry *entry, TVector gradient, TVector params, TArray methods, double K) {

    TGradientData data;

    double E = linearEvaluation(entry, params, methods, &data);
    double S = sigmoid(K, E);
    double X = (entry->result - S) * S * (1 - S);

    double complexitySign = (data.egeval > 0.0) - (data.egeval < 0.0);
    double mgBase = X * entry->pfactors[MG], egBase = X * entry->pfactors[EG];


    for (int i = 0; i < entry->ntuples; i++) {

        int index  = entry->tuples[i].index;
        int wcoeff = entry->tuples[i].wcoeff;
        int bcoeff = entry->tuples[i].bcoeff;

        if (methods[index] == NORMAL)
            gradient[index][MG] += mgBase * (wcoeff - bcoeff);

        if (methods[index] == NORMAL && (data.egeval == 0.0 || data.complexity >= -fabs(data.egeval)))
            gradient[index][EG] += egBase * (wcoeff - bcoeff) * entry->sfactor;

        if (methods[index] == COMPLEXITY && data.complexity >= -fabs(data.egeval))
            gradient[index][EG] += egBase * wcoeff * complexitySign * entry->sfactor;

        if (methods[index] == SAFETY)
            gradient[index][MG] += (mgBase / 360.0)
                                 * (fmax(data.bsafetymg, 0) * bcoeff - (fmax(data.wsafetymg, 0) * wcoeff));

        if (methods[index] == SAFETY && (data.egeval == 0.0 || data.complexity >= -fabs(data.egeval)))
            gradient[index][EG] += (egBase /  20.0)
                                 * ((data.bsafetyeg > 0.0) * bcoeff - (data.wsafetyeg > 0.0) * wcoeff);
    }
}
```

4.6 Final AdaGrad Implementation

Ethereal's implementation is shown below. Everything here is "standard" for a Gradient Decent or AdaGrad implementation. We track a running sum of all previous squared Gradients. In our Gradient computation we incorrectly neglected to factor in the $\frac{-2K}{N}$. This was done to save time and to avoid precision loss in the Gradient. We also include code for how mini-batches would work in this setup. However, Ethereal currently does full batches, where BATCHSIZE equals NPOSITIONS. Previous datasets have shown promise with mini-batching over batching.

```
void runTuner() {

    TArray methods = {0};
    TVector params = {0}, cparams = {0}, adagrad = {0};
    double K, error, rate = LRRATE;
    TEntry *entries = calloc(NPOSITIONS, sizeof(TEntry));

    initMethodManager(methods);
    initTunerEntries(entries,methods);
    K = computeOptimalK(entries);

    for (int epoch = 0; epoch < MAXEPOCHS; epoch++) {

        // We include code for Mini-Batching, even though Ethereal uses only Batches
        for (int batch = 0; batch < NPOSITIONS / BATCHSIZE; batch++) {

            TVector gradient = {0};
            computeGradient(entries, gradient, params, methods, K, batch);

            for (int i = 0; i < NTERMS; i++) {

                // Sum of all previous squared Gradients
                adagrad[i][MG] += pow(2.0 * gradient[i][MG] / BATCHSIZE, 2.0);
                adagrad[i][EG] += pow(2.0 * gradient[i][EG] / BATCHSIZE, 2.0);

                // Note that we did not have the -2/BATCHSIZE in the actual Gradient. This is done as
                // both a speedup by saving operations, and a resolution increase by limiting divisions.
                params[i][MG] += (K * 2.0 / BATCHSIZE) * gradient[i][MG] * (rate / sqrt(1e-8 + adagrad[i][MG]));
                params[i][EG] += (K * 2.0 / BATCHSIZE) * gradient[i][EG] * (rate / sqrt(1e-8 + adagrad[i][EG]));
            }
        }

        error = tunedEvaluationErrors(entries, params, methods, K);
        printf("Epoch [%d] Error = [%g], Rate = [%g]\n", epoch, error, rate);

        // Pre-scheduled Learning Rate drops
        if (epoch && epoch % LRSTEPRATE == 0) rate = rate / LRDROPRATE;
        if (epoch % REPORTING == 0) printParameters(params, cparams);
    }
}
```