Feature set analysis for chess NNUE networks

October 25, 2024

Martín Emiliano Lombardo

mlombardo9@gmail.com

Directores

Agustín Sansone   Diego Fernández Slezak

agustinsansone7@gmail.com   dfslezak@dc.uba.ar

Abstract

Historically, chess engines have used highly complex functions to evaluate chess positions. Recently, efficiently updatable neural networks (NNUE) have displaced these functions that do not need human knowledge. The input of these networks are called feature sets and they take advantage of the order in which positions are evaluated in a depth-first search to save computation.

In this thesis, I develop a classical chess engine, where the evaluation function is replaced by a NNUE network trained with a pipeline created from scratch. The main goal of this thesis is to test novel feature sets that can improve performance. Additionally, a way of training the networks is tested using a method proposed years ago but with a higher volume and quality of data available in the post-NNUE era.

Abstract (Spanish)

Históricamente, los motores de ajedrez han utilizado funciones altamente complejas para evaluar posiciones de ajedrez. Recientemente las redes neuronales eficientemente actualizables (NNUE) han desplazado a estas funciones sin necesidad de utilizar conocimiento humano. El input de estas redes se las denomina feature sets y se aprovechan del orden en que se evalúan las posiciones en una búsqueda depth-first para ahorrar cómputo.

En esta tesis realizo un motor de ajedrez clásico, en donde la función de evaluación es reemplazada por una red NNUE entrenada con un pipeline creado de cero. Esta tesis busca probar novedosos feature sets que puedan mejorar el rendimiento. Adicionalmente, se prueba una manera de entrenar las redes utilizando un método propuesto hace años pero con un volumen y calidad de datos superiores disponibles en la era post-NNUE.

Agradecimientos

:)

Contents

1 Introduction

The development of chess engines was and continues to be a topic of study in the chess and computer communities for decades. IBM DeepBlue [4] was the first chess machine to reach superhuman level by consistently beating the world champion, Garry Kasparov, in 1997 [5]. Since then, engines have evolved in strength and complexity.

Chess can be modeled as a tree, where each node is a particular board configuration and the edges are legal moves for that position. With this representation, engines can use tree search algorithms to explore the tree and approximate the best move. Since the 1950s and to this date, engines have used algorithms like Minimax [7] and Monte Carlo Tree Search [3] (MCTS) or some of its variants [8, 21] to accomplish this.

The number of possible positions in chess is vast, estimated by Shannon [12] to be around 10^{43}. This number is based on the average number of legal moves per position and the average game length. This makes it not feasible to explore the entire tree, so every tree search algorithm relies on having an evaluation function: a function that takes the state of the game and returns a single real number. This number is used to encompass information about the whole subtree of that position so it can be propagated up the tree, depending on the algorithm. Until a few years ago, highly complex handcrafted functions were used that were based on human knowledge about the game.

Until the 2010s, the development of engines advanced at a slow but consistent pace. Until 2017 that Google DeepMind published AlphaGo Zero [15] and its successor AlphaZero [14, 13] in 2018, which proved to be overwhelmingly superior (28 wins, 73 draws and 0 losses against the best engine at that time). They introduced a new approach to the development of board game engines, including chess: train a convolutional neural network with a reinforcement learning algorithm to learn to play by itself.

This change of paradigm, where the evaluation of positions is done by neural networks instead of functions built with human knowledge, altered the course of development of all modern engines (not just Go and chess). In 2018, Yu Nasu introduced the networks ƎUͶͶ (or NNUE) "Efficiently Updatable Neural-Networks" [10] for the game Shogi. NNUE networks allow for cheap evaluations when evaluating a sequence of similar positions, making them ideal for use in depth-first search based engines. Since then, all modern engines have incorporated NNUE networks or some kind of neural network in their evaluation.

The chess engine Stockfish [17], modern successor of DeepBlue with improved heuristics and running in commercial hardware is one of the strongest in the world. It incorporated NNUE networks mixed with classical evaluation in version 12[1]. Since Stockfish 16.1[2] (2024) the evaluation is done exclusively through NNUE networks, eliminating all human aspect.

---

[1]Introducing NNUE evaluation (Stockfish 12)
[2]Removal of handcrafted evaluation (Stockfish 16.1)

1.1 Thesis plan

The main goal of this thesis is to explore different kinds of board encodings called feature sets. These encodings are the input for a NNUE network. To do so, I need a chess engine that supports neural networks with the ability to customize encodings and a way to train them.

I decided to implement a simple but capable classic engine based on well-known algorithms and optimizations, and then change the evaluation to use NNUE networks, with a versatile framework to build feature sets. Finally, I implemented a training pipeline to train the networks and measure their performance.

With the setup ready, I run multiple experiments. I propose different feature sets, train networks with them and compare their performance.

2 Engine implementation

Building chess engines is a very discussed topic in the history of chess and thus very well documented. The Chess Programming Wiki (CPW) [6] is a well known source of information to reference, which I will base my engine on. I aim to build a single-threaded classic engine and only make use of the most prominent optimizations to keep it simple. The engine strength is not that relevant, as it is only a tool to measure the relative performance of board encodings. However, a competent one is required.

Classic chess engines are composed of two main components: **the search** and **the evaluation**. The search is the process of exploring the tree of possible moves, which is what this chapter is about. The evaluation determines how good the positions are for who plays. As I mentioned in the introduction, classic engines used to use hand-crafted evaluations based on human knowledge. In my case, I will replace it entirely with a neural network, explained in the following chapters.

2.1 Minimax search

A position p in chess is the state of the board along with any relevant information, like castling rights, en passant and the 50-move clock. Given a position p, we can call $f(p)$ its evaluation, a number that provides an assessment of how good the position is, computed either by a hand-crafted function or a neural network.

One approach to approximate a good move given a reasonable function f could be to evaluate all possible positions that can be reached with a single move and choose the one that leads to the highest evaluation for the player who made the move. This idea can be extended to consider actions taken by the other player, and so on, to a fixed depth. Formally, this is called the minimax search algorithm [7].

In a minimax tree there are two kinds of nodes: maximizing nodes and minimizing nodes.

- △ **Maximizing nodes** are the ones where the player to move is our player. These nodes want to put the player in the best possible position, so they choose the action that maximizes the evaluation. Note that the root node is a maximizing node.

- ▽ **Minimizing nodes** are the ones where the player to move is the opponent. These nodes want to put the player in the worst possible position, so they choose the action that minimizes the evaluation.

Figure 1. A minimax tree of depth 4. The best move for the maximizing player is the one that leads to the highest evaluation, marked in blue.

The algorithm recursively explores the tree to a fixed depth, evaluating the positions at the leaves with f. The evaluation is then propagated up the tree, alternating between maximizing and minimizing nodes, until it reaches the root node. After computing the whole tree to a fixed depth, the "best" move is defined by the move from the root node that maximizes the recursively computed evaluation (maximizing node).

Usually we do not want to run the search to a fixed depth, but rather for a fixed amount of time. The algorithm itself runs to a fixed depth, so what we can do is to run the search in a loop, staring from depth 1 and increasing it by one each iteration until the time runs out. This way the "best" move found so far is always available. Note that we can not draw conclusions from any unfinished search, so the "best" move is the one found at the last iteration. This approach is called iterative deepening, and when combined with a transposition table (a cache for evaluations) is very effective, making following iterations faster.

My implementation uses a variation of the minimax algorithm called *negamax*. Negamax is a simplification of minimax that takes advantage of the zero-sum property of chess, meaning that an evaluation for a player is equivalent to the negation of the evaluation in the opponent's point of view. Instead of having two kinds of nodes, all nodes are maximizing nodes and the evaluation is negated after the recursion. This simplifies the implementation.

2.2 Quiescence search

The search algorithm runs to a fixed depth, which causes a horizon effect. The horizon effect manifests when the search stops at a position where a negative event (such a capture) is inevitable but due the fixed depth, the search results in weaker moves in an effort to avoid the inevitable, prefering branches where the negative event (the capture) has not happened yet.

Figure 2. Demonstration of the horizon effect (not a minimax tree, only showing opponent nodes), when the search stops at depth 2. The capture PxQ (♙ Pawn takes ♛ Queen) is inevitable. In the red branches, the capture has already happened. In the green branches, the capture has not happened yet. Evaluations at the leaves favour the green positions because those have an extra ♛ Queen. Since losing the piece is inevitable, green positions may actually be weaker than red positions, but the search does not know that.

To fix this, instead of returning the evaluation of the position at the leaves, an additional smaller search is done which only considers captures. This way the search can continue until a "quiet" position is reached, where no captures are available.

Since most of the positions the network will be evaluating are quiet due the quiescence search, it is important to make sure that the training set reflect that. Later on, only positions that are quiet will be used to train the network.

2.3 Optimizations

Many optimizations were made to the engine to make it reach a decent depth in a reasonable time, which makes the engine stronger. There are no novel improvements, most are well-known techniques that have been used in engines for decades and can be found in the Chess Programming Wiki [6].

The most prominent optimizations implemented are:

- **Alpha-beta pruning**: a way to eliminate big portions of the search tree by using the branch-and-bound technique. It allows to prune branches that are guaranteed to be worse than the most promising move found so far. This means that it does not affect the result of the search, it only makes it faster.

Each node in the search tree has two values associated with it: α and β. α is the best value found so far that the maximizing player can guarantee up to that node. β is the best value found so far that the minimizing player can guarantee up to that node. Note that $\alpha \leq \beta$ and the maximizing player tries to "push" α up and the minimizing player tries to "pull" β down.

When a node is visited, the algorithm checks if $\alpha \geq \beta$. If this is the case, the branch can be pruned because the minimizing player can guarantee a value of β, which is worse than the best value found so far.

- **Move ordering**: the order in which the moves are visited can have a big impact on the effectiveness of the alpha-beta pruning. If the move ordering is optimal, the effective branching factor is reduced to its square root, which means that the search can go twice as deep for the same amount of computation [11, section 5.3.1]. In the worst case, it is identical to minimax. There are a couple of ways to improve move ordering, the most important being:

  - **MVV/LVA** the most valuable victim, least valuable attacker is a simple heuristic that orders the moves by the value of the captured piece minus the value of the attacking piece. This way the most valuable captures are evaluated first, which are more likely to cause a cutoff.

- **Transposition table**: during search, a position may be visited many times with different sequences of moves. This is called a transposition. The transposition table is a large hash table storing information about positions that have already been visited. This way, if a position is visited again, the engine can use the stored information to avoid re-evaluating it.

  Even if the depth of the stored evaluation is lower than the current depth (insufficient to draw conclusions at the current depth), it can still be used to improve the move ordering.

Many other optimizations were made: history moves, killer moves, late move reductions, null move pruning, and more. They can be found in the documented code.

2.4 Implementation details

The engine is implemented in the Rust programming language. It uses the standard UCI protocol [1] to communicate via standard input/output.

The most performance critical part of the engine aside from the evaluation is move generation, that is, given a position, list all available moves and make them. Fortunately there is a battle-tested library for it called *shakmaty*. The library provides a copy-make interface instead of a make-unmake one, so I have to rely on a stack of positions when doing recursion.

---

[1]Universal Chess Interface specification can be found here.

Time control is hard-coded to use the increment plus 2% of the remaining time per move. Experiments run at a fixed time per move, so this is used in the Lichess arena.

By default the engine uses 128 MB of memory for the transposition table. This value will be used throughout the experiments and in the arena.

3 Feature sets (board encodings)

To evaluate chess positions, the engine will use a neural network with an architecture explained in detail in the next chapter. In this chapter, I will show how to build the one-dimensional input vector for such network, which can be described entirely by a feature set.

A feature set is a set with a predicate attached to it. The elements can be anything, but usually we want to represent chess concepts like piece locations, piece roles, colors, etc. We may want to represent more complex patterns, so we can build feature sets by taking the cartesian product of smaller sets. The predicate $P(e)$ defines if the element or pattern e is present (or *active*) in the (implicit) position. The predicate is generally written using natural language.

Formally, given a set of concepts or tuples S and a predicate P, we can define a feature set as S_P, where each element is called a feature. Each feature corresponds to a value in a vector, which will be set to 1 if the predicate is satisfied for that element in the position, and 0 otherwise. This is the vector that will be used as input to the neural network.

Let's consider some basic sets of concepts. For example, the following sets describe positional information about the board:

$$\textsc{Files} = \{a, b, ..., h\}$$
$$\textsc{Ranks} = \{1, 2, ..., 8\}$$
$$\textsc{Squares} = \{a1, a2, ..., h8\}$$

And the following describe information about the pieces:

$$\textsc{Roles} = \{ \text{♙ Pawn, ♘ Knight, ♗ Bishop, ♖ Rook, ♕ Queen, ♔ King}\}^{[1]}$$
$$\textsc{Colors} = \{\bigcirc \text{ White, } \bullet \text{ Black}\}$$

For example, consider the feature set $(\textsc{Files} \times \textsc{Colors})_P$ where P is defined like $P(\langle f, c \rangle)$: there is a piece in file f with color c. A feature in this set will be active if there is at least one piece in the board that makes the predicate true. In this case, disregarding any other kind of information, like the piece's role. Another possible feature set could be $(\textsc{Files} \times \textsc{Roles})_Q$, with a similar interpretation. An illustration of the active features of these two feature sets is shown in Figure 3.

Note that $\textsc{Squares}_R$ is equivalent to $(\textsc{Files} \times \textsc{Ranks})_R \; \forall R$.

---

[1] The color of the pieces have no meaning in the definition. They are present for illustrative purposes.

	Feature set	
	$(\text{FILES} \times \text{COLORS})_P$	$(\text{FILES} \times \text{ROLES})_Q$
Active features	$\langle a, \bigcirc\rangle$, $\langle a, \bullet\rangle$, $\langle c, \bullet\rangle$, $\langle c, \bigcirc\rangle$, $\langle d, \bigcirc\rangle$, $\langle h, \bullet\rangle$	$\langle a, ♙\rangle$, $\langle c, ♕\rangle$, $\langle c, ♘\rangle$, $\langle d, ♙\rangle$, $\langle h, ♗\rangle$

$P(\langle f, c\rangle)$: there is a piece in file f with color c.
$Q(\langle f, r\rangle)$: there is a piece in file f with role r.

Figure 3. Active features of two feature sets for the same board.

3.1 Sum $\oplus$

The sum (or concatenation) of two feature sets A and B, denoted by $A \oplus B$, is a new feature set comprised of the features of both sets. These features do not interfere with each other at all. Formally, given two feature set S_P and T_Q, we can define the sum operator as

$$S_P \oplus T_Q = (S \cup T)_R$$

$$\text{where } R(e) = \begin{cases} P(e) & \text{if } e \in S \\ Q(e) & \text{if } e \in T \end{cases}$$

The sum operator is useful when we want to let the network find patterns combining information between two sets of features.

Even though the two operands are feature sets, they are usually called "feature blocks" since they are part of a larger feature set. The final feature set that is used for training is a sum of many feature blocks.

3.2 Product $\times$

The product of two feature sets A and B, denoted by $A \times B$, is a new feature set where each new feature is a combination of the features of both sets. One way to interpret this is that each new feature will be active if both features in the original sets are active *at the same time*. Formally, given two feature sets S_P and T_Q, we can define the product operator as

$$S_P \times T_Q = (S \times T)_R$$

$$\text{where } R(\langle e_0, e_1\rangle) = P(e_0) \wedge Q(e_1)$$

This operation is not that useful because it requires both predicates to be independent from each other. This will be used specifically to define the KING-ALL feature set and potentially its variations.

3.3 Known feature sets

In this section, I will define two of the most important feature sets known and used extensively in existing engines.

3.3.1 ALL

This feature set is the most natural encoding for a chess position. It is called "All" because it captures all the pieces. There is a one-to-one mapping between pieces in the board and features:

$$\text{ALL}: (\text{SQUARES} \times \text{ROLES} \times \text{COLORS})_P$$
$$P(\langle s, r, c \rangle): \text{there is a piece in square } s \text{ with role } r \text{ and color } c$$

Tuples in this set are *active* if there is a piece in the board that matches the role, color and position of the tuple. For example, the tuple $\langle e4, \text{♙}, \bigcirc \rangle$ is active if there is a white pawn in the square e4. This way, for every possible piece, in every possible position, there is a feature. The set has $64 * 6 * 2 = \textbf{768 features}$, which makes it very small and it is very easy to compute which features are active.

3.3.2 KING-ALL

Another feature set built on top of ALL is the KING-ALL feature set, or "KA" for short. For every possible position where the king of the side to move can be, there is a complete copy of the ALL set:

$$\text{KING-ALL} = \text{SQUARE}_K \times \text{ALL}$$
$$K(s): s \text{ is the square of the king of the side to move}$$

This encoding allows the network to understand better the position of the pieces in relation to the king, which is very tied to the evaluation of the position.

The number of features is $64 * 768 = \textbf{49152 features}$. There is a variation of this feature set called "KP" which is the same but it does not consider the enemy king, reducing the amount of features to 40960. There are other variations, such as KAv2 or notably KAv2_HM that is currently the latest feature set used by Stockfish 16.1.

The features in this set are easy to compute like in ALL, but since the number of features is much larger, it is a lot harder to train and use in practice. I will restrain this work to smaller feature sets that are easier to manage.

3.4　Indexing

We need a way to map the tuples in a feature set to elements in the input vector. The correct index for a tuple is computed using the order of the sets in the cartesian product and the size of each set, like strides in a multi-dimensional array. For this to work, each element e in a set S must correspond to a number between 0 and $|S| - 1$, we call this bijective mapping $I(e)$.

For example, the feature set $(A \times B \times C)_P$ has $|A| \times |B| \times |C|$ features, and the feature $\langle a, b, c \rangle$ is mapped to the element indexed at $I(a) \times |B| \times |C| + I(b) \times |C| + I(c)$. The same striding logic applies to feature sets built with the sum and product operators, recursively.

3.5　Dead features

Consider the ALL feature set. For every position, role and color each piece could be, there is a feature. There are 16 tuples in the set that will never be active: $\langle$a8..h8, ♙, ○$\rangle$ and $\langle$a1..h1, ♙, ●$\rangle$ that correspond to the white pawns in the last rank and the black pawns in the first rank. This is because pawns promote to another piece when they reach the opponent side of the board, so no pawns will ever be found there. Effectively, these will be dead neurons in the network, but this way we can keep the indexing straightforward. Most feature sets will have dead features, and the same logic applies.

3.6　Summary

1. **S**: set of concepts (roles, colors, squares, files, ranks, etc.).

2. **$P(e)$**: predicate that defines when the feature e is present in the (implicit) position.

3. **S_P**: a feature set. Every element in S_P is a feature. Features that satisfy P are *active*.

4. $S_P \times T_Q = (S \times T)_R$ where $R(\langle e_0, e_1 \rangle) = P(e_0) \ \wedge \ Q(e_1)$

5. $S_P \oplus T_Q = (S \cup T)_R$ where $R(e) = \begin{cases} P(e) & \text{if } e \in S \\ Q(e) & \text{if } e \in T \end{cases}$

4 Efficiently updatable neural networks

NNUE (ƎUИИ Efficiently updatable neural network) is a neural network architecture that allows for very fast subsequent evaluations when changes in the input are minimal. It was invented for Shogi by Yu Nasu in 2018 [10], later adapted to Chess for use in Stockfish in 2019. Most of the information described in this chapter can be found in the excellent Stockfish NNUE documentation [18].

NNUE operates on the following principles:

- **Input sparsity**: The network should have a relatively low amount of non-zero inputs, determined by the chosen feature set. The presented feature sets have between 0.1% and 2% of non-zero inputs for a typical position. Having a low amount of non-zero inputs places a low upper bound on the time required to evaluate the network in its entirety, which can happen using some feature sets like KING-ALL that triggers a complete refresh when the king is moved.

- **Efficient updates**: From one evaluation to the next, the number of inputs changes should be as low as possible. This allows for the most expensive part of the network to be efficiently updated, instead of recomputed from scratch.

- **Simple architecture**: The network should be composed of a few and simple operators, that can be efficiently implemented with low-precision arithmetic in integer domain using CPU hardware (quantization).

There is a tradeoff between inference time and quality of the predictions, which affect the amount of nodes evaluated. If the inference is faster, more nodes can be evaluated thus reaching deeper in the search tree. Having higher quality predictions, which usually comes with a more complex model and/or feature set, can make stronger moves with shallower searches, and may improve prunning.

Models must be so much better to compensate the slowdown in inference when using bigger models or more complex feature sets. This is a tradeoff that will appear further in the experiments.

4.1 Layers

For this thesis, I have chosen to use a very simple NNUE architecture, which consist of two linear (fully connected) layers and clipped ReLU activations. In the literature, there are other architectures that make use of polling layers, sigmoid activations and others. Since this work is about experimenting with feature sets, I have chosen to stick with something simple that has been proven to achieve good results.

Linear layer A linear layer is a matrix multiplication followed by a bias addition. It takes **in_features** input values and produces **out_features** output values. The operation is $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, where:

1. $\boldsymbol{x}$ the input column vector of shape **in_features**.

2. $\boldsymbol{W}$ the weight matrix of shape (**out_features**, **in_features**).

3. $\boldsymbol{b}$ the bias column vector of shape **out_features**.

4. $\boldsymbol{y}$ the output column vector of shape **out_features**.

If we call $\boldsymbol{A_i}$ the i-th column of the weight matrix $\boldsymbol{W}$, the operation $\boldsymbol{W}\boldsymbol{x}$ can be simplified to "if $\boldsymbol{x_i}$ is not zero, take the column $\boldsymbol{A_i}$, multiply it by $\boldsymbol{x_i}$ and add it to the result". This means that we can skip the processing of columns that have a zero input, as depicted in Figure 4.

(a) Linear layer   (b) Linear layer with sparse inputs

Figure 4. Linear layer operation comparison. Figures from [18].

In the case of the first layer, the input is a very sparse one-hot encoded vector. This means that very few columns will have to be processed and the multiplication can be skipped altogether, due all inputs being either 0 or 1. Skipping the multiplication reduces the operations to only additions and subtractions.

Clipped ReLU This is a simple activation that clips the output in the range $[0, 1]$. The operation is $\boldsymbol{y} = \min(\max(\boldsymbol{x}, \boldsymbol{0}), \boldsymbol{1})$. The output of this activation function is the input for the next layer, and because of the aggresive *quantization* that will be described later, it is necessary to restrain the values so it does not overflow.

4.2 Efficient updates

When running a depth-first search algorithm, the state of the position is updated every time the algorithm *makes* and *unmakes* moves, usually before and after the recursion. NNUEs are designed to work with this kind of search, since every time the algorithm *makes* (or *unmakes*) a move, the changes in the position are minimal (at most two pieces are affected in ALL), meaning that the amount of features becoming active or inactive is minimal as well. This is depicted in Figure 5.

Figure 5. Partial tree of feature updates (removals and additions) for (SQUARES × COLORS) (white's point of view) in a simplified 3x3 pawn-only board.

To take advantage of this during search, instead of computing all the features active in a position and then evaluate the network in its entirety, we can **accumulate** the output of the first linear layer and update it with when the position changes. Linear layers can be computed adding the corresponding columns of the weight matrix into the output, so when a feature becomes active or inactive, we can add or subtract the corresponding column to the output. When the evaluation is needed, only the next layers (usually small) have to be computed.

Recall that the way I defined feature sets, they always encode the position from one white's point of view. This means that its not possible to use the same **accumulator** for both players. So when running the search, we have to keep two accumulators, one for white and one for black, where the black board is flipped and has the colors swapped to match the point of view.

During search, the first layer is replaced by two accumualtors to take advantage of this. Figure 6 depicts how the output of both accumulators is concatenated depending on which

player is moving, to later be passed through the rest of the network which is computed as usual.

Figure 6. Concatenation of the first layer's output after a move is made. Inspired in a CPW figure.

4.3 Network

The network will be composed of three linear layers L_1 through L_3, each but the last one followed by a clipped ReLU activation C_1 and C_2. The network has two inputs: it takes the encoding (feature set) of a position from each player's point of view. Each encoding is passed through the same L_1 layer (same weights) and then the output is concatenated before passing it through the rest of the network. The first layer can be seen as a feature transformer, and it must share weights to allow for efficient updates. The network can be described as follows:

$\boldsymbol{N}$: number of features in the feature set

1. $L_1 \times 2$: Linear from $\boldsymbol{N}$ to $\boldsymbol{M}$ ($\boldsymbol{W_1}$ weight, $\boldsymbol{b_1}$ bias)
2. C_1: Clipped ReLU of $\boldsymbol{2 * M}$
3. L_2: Linear from $\boldsymbol{2 * M}$ to $\boldsymbol{O}$ ($\boldsymbol{W_2}$ weight, $\boldsymbol{b_2}$ bias)
4. C_2: Clipped ReLU of $\boldsymbol{O}$
5. L_3: Linear from $\boldsymbol{O}$ to $\boldsymbol{1}$ ($\boldsymbol{W_3}$ weight, $\boldsymbol{b_3}$ bias)

The size of each layer is not fixed since it is a hyperparameter I will experiment with. The network architecture is depicted in Figure 7, with example parameters.

Figure 7. Neural network architecture with $N = 768$, $M = 256$, $O = 32$. Not to scale.

4.4 Quantization

Quantization is the process of converting the operations and parameters of a network to a lower precision. It is a step performed after all training has been done, which do happen in float domain. Floating point operations are too slow to achieve acceptable performance, as it sacrifices too much speed. This was necessary to implement to have a working engine.

Quantizing the network to integer domain will inevitable introduce some error, but it far outweights the performance gain. In general, the deeper the network, the more error is accumulated, but since NNUEs are very shallow by design, the error is negligible. At the end of the chapter I do an analysis of the error introduced by quantization.

Since the objective is to take advantage of modern CPUs that allow doing low-precision integer arithmetic in parallel with 8, 16, 32 or even 64 8-bit integer values at a time, we want to use the smallest integer type possible everywhere, to process more values at once.

4.4.1 Stockfish quantization scheme

In this thesis, I will use the same quantization scheme used in the engine Stockfish [18], due its simplicity and it has been battle tested. It uses `int8` $[-128, 127]$ for inputs and

weights, and `int16` $[-32768, 32767]$ where `int8` does not fit the range of values we need. To convert the float values to integer, we need to multiply the weights and biases by some constant to translate them to a different range of values. Each layer is different, so I'll go through each one.

Input   In float domain inputs are either 0.0 or 1.0, and since they are quantized to `int8` we must scale them by $s_a = 127$ (activation scale), so inputs are either 0 or 127. During inference, the input values are not computed since the first layer is an accumulator. However it is important to note that the rows being accumulated are scaled by $s_a = 127$.

ClippedReLU   The output of the activation in float domain is in the range $[0, 1]$ and we want to use `int8` in the quantized version, so we can multiply by $s_a = 127$ and clamp in the range $[0, 127]$. The input data type may change depending on the previous layer: if it comes from the accumulator, it will be `int32`, and if it comes from a linear layer, it will be `int16`.

Accumulator (L1)   The purpose of this layer is to accumulate rows of the first layer's weight matrix, which is stored in `int16`. The values are stored in column-major order so a single row is contiguous in memory. Since we are accumulating potentially hundreds of values which are stored in `int16` and scaled by $s_a = 127$, we must accumulate in `int32` to avoid overflows. The output of this layer will be the input for the ClippedReLU activation.

Linear layer (L2 and L3)   The input to this layer will be scaled to the activation range because it takes the output of the previous ClippedReLU activation: $s_a \boldsymbol{x}$. We want the output to also be scaled to the activation range so it can be passed to the next: $s_a \boldsymbol{y}$.

To convert the weights to `int8`, we must scale them by some factor $s_W = 64$ (value used in Stockfish, efficient in SIMD because is just a shift): $s_W \boldsymbol{W}$. The value s_W depends on how much precision we want to keep, but if it is too large the weights will be limited in magnitude. The range of the weights in floating point is then determined by $\pm \frac{s_a}{s_W} = \frac{127}{64} = 1.984375$, and to make sure weights don't overflow, it is necessary to clip them to this range during training. The value s_W also determinates the minimum representable weight step, which is $\frac{1}{s_W} = \frac{1}{64} = 0.015625$.

The linear layer operation with the scaling factors applied looks like:

$$s_a s_W \boldsymbol{y} = (s_W \boldsymbol{W})(s_a \boldsymbol{x}) + s_a s_W \boldsymbol{b} \tag{1}$$

$$s_a \boldsymbol{y} = \frac{(s_W \boldsymbol{W})(s_a \boldsymbol{x}) + s_a s_W \boldsymbol{b}}{s_W} \tag{2}$$

From that equation we can extract that, to obtain the result we want, which is the output of the layer scaled to the activation range ($s_a \boldsymbol{y}$), we must divide the result of the operation by s_W (2). Also that the bias must be scaled by ($s_a s_W$).

The last linear layer (L3) is a bit different since there is no activation afterwards, so we don't want any scalings at all:

$$\boldsymbol{y} = \frac{(s_W \boldsymbol{W})(s_a \boldsymbol{x}) + s_a s_W \boldsymbol{b}}{s_a s_W} \tag{3}$$

4.5 Implementation

The Stockfish repository provides an AVX2 implementation of the mathematical operations in C++. They have been carefully ported to Rust for this thesis. The implementation was thoroughly tested using the Pytorch model as reference (output match).

4.5.1 Quantization error

To make sure the quantization is working as expected, I compared the actual output of the quantized model (in Rust) with the float model (in Python) by running them in thousands of positions.

Figure 8. Comparison between the float model and the quantized model. $N = 100000$

In Figure 8 we can see that the distribution of the evaluation of both models is almost identical, indicating that the implementation is correct.

To measure the error introduced it is better to do it in WDL-space, since we can make sense of values in that space. The errors are near zero, and almost all errors are within 0.03 units, which is a 3% difference in winrate. The Stockfish team has reported that errors in quantization up to 5% do not affect the engine's strength.

20

5 Training

Given a feature set, the network architecture is completely defined, along with how to encode a position into its inputs. This section will describes two proposed methods to train the networks, each with its own loss function and training dataset.

5.1 Source dataset

Data is needed to train the network. The proposal for the thesis was to use the Lichess database [9], which provides a CC0 database with all the games ever played on the site, then score the positions using Stockfish. After some initial experiments, the networks were not performing as expected. Upon further reaserch I found out that I was working with datasets too small for this task (order of hundreds of millions). I needed a larger dataset (order of **dozens of billions**), but it was impractical for me to generate it. Fortunately, I can use the same dataset that Stockfish uses to train its networks [20], which should work well. Specifically, I went with the dataset used to train the first stage of the main network for Stockfish 16.1, which is 135GB of compressed `binpack` files. It was built by running Stockfish at 5000 nodes per move on multiple opening books. Later stages use datasets generated by Leela Chess Zero (LC0), which is more expensive to compute but has a higher quality evaluations.

The `binpack` format is a very efficient method of storing samples yet very complex to decode. Fortunately, Stockfish provides a tool to export this data into a text representation. I had to modify it to export it in the format I wanted. I changed the `emitPlainEntry` function in `nnue_data_binpack_format.h` to the code in Appendix A.6. The resulting file was 2.59TB in size and contained **48.4 billion samples**. There is one sample per line with the format:

$$\boxed{\textbf{FEN}^2 \quad , \quad \textbf{Score} \quad , \quad \textbf{Best move}}$$

The file was too big to be practical and it would wear off my SSD, so I made a tool to compact the data into a similar format. The new format expoits the fact that samples in a row belong to the same game. This means that contiguous FENs are a move from a previous one, so it stores the move instead of the FEN:

$$\boxed{\textbf{FEN} \quad , \quad \textbf{Score} \quad , \quad \textbf{Best move}} \left(\boxed{, \quad \textbf{Actual move} \quad , \quad \textbf{Score} \quad , \quad \textbf{Best move}} \right) *^3$$

As you can see, the new format is compatible with the last one, so only one reader was implemented. After compacting the data, the file went down to a manageable 522GB. Also, reading a single FEN and later apply moves to it is much faster than parsing a FEN every time.

---

[2]Standard notation to describe positions of a chess game. It is a sequence of ASCII characters.
[3]Repeated zero or more times.

There are many positions in the dataset that are known to not be good for training. Remember that the engine is doing quiescent search, so it does a smaller search looking for quiet positions to evaluate. This means that positions where the best move is a capture, or there is a check are filtered out when building the training batch.

Each training method will generate a new derived dataset based on this samples.

5.2 Method 1: Score target

The main method to train the network will use the scores provided in the dataset as target. I expect the networks to learn to predict the evaluation of a position as Stockfish would do.

5.2.1 Score-space to WDL-space

Evaluations in the dataset are values ranging from -10000 to 10000. We call this range of values the score-space, also refered to as the *centipawn scale* (or something proportional to it). We want the network to output the same values as the dataset, in *score-space*.

The WDL-space is a different scale in [0, 1] where 0 is a loss, 0.5 is a draw and 1 is a win. The WDL (win-draw-loss) model [19] states that the win rate for a position can be modeled as a function of the evaluation of the position. The data shows that the logistic function (sigmoid) gives a good approximation for the evaluation $f(p)$:

$$\mathcal{W}(f(p)) = \sigma\left(\frac{f(p) - a}{b}\right) = \frac{1}{1 + e^{-\frac{f(p)-a}{b}}}$$

where a and b need to be fitted to the data. The value of a is the evaluation where a 50% winrate is observed and b indicates how fast the winrate changes when the evaluation change. The fitted sigmoid is shown in Figure 9, and the values obtained are $a = 1.28$ and $b = 297.21$.

Figure 9. WDL model function (sigmoid) fitted to 100 million evaluations in the dataset.

During training, it is better to use a loss function with the target and output of the model in WDL-space instead of score-space. WDL-space has some advantages over score-space:

- Large evaluations are "closer" together in WDL-space., since having a score of 7500 or 8000 is not that different in terms of winrate (less than 1%) than between 50 and 550 (more than 30%). This is desirable because the evaluations don't need to be that precise when the outcome of the game is almost decided.

- The result of a game can be interpolated in WDL-space. If we introduce a new parameter λ, we can interpolate the evaluation $f(p)$ and the game result r (in WDL-space) using: $\lambda * \mathcal{W}(f(p)) + (1 - \lambda) * r$. This way, the information about the outcome of the game can be used to steer the network in the right direction. This is not implemented in this work.

- Values in WDL-space are smaller than in score-space, so it avoids large gradients.

5.2.2 Loss function

The loss function chosen is mean squared error (MSE) with a power of 2.6 (the value used by the Stockfish's official trainer) given by

$$\mathcal{L}(y, f(x, \boldsymbol{W})) = \frac{1}{N} \sum_{i}^{N} |\mathcal{W}(y_i) - \mathcal{W}(f(x_i, \boldsymbol{W}))|^{2.6}$$

where. . .

1. N is the number of samples.

2. y are the target scores.

3. f is the model.

4. x are the inputs (encoded feature sets).

5. $\boldsymbol{W}$ are the parameters of the model.

6. $\mathcal{W}$ is the winrate function that maps from score-space to WDL-space.

5.3 Method 2: PQR triplets

This is an additional technique I wanted to try, described in [1]. The method is based in the assumption that moves in the training data are better than random. In the blog they used human moves from the Lichess database [9], so they rely in the fact that humans make good or near-optimal moves most of the time, even if they are amateurs. In my case I will use Stockfish moves, which are extremely good. This method does not use the scores provided, it will have to learn them from scratch. Of course this is way harder to train but I'm curious to see how far the following idea can go.

Remember that we are trying to obtain a function f (the model) to give an evaluation of a position. The idea is based on the following two principles:

1. For two positions in succession $p \rightarrow q$ observed in a game, we will have $f(p) = -f(q)$. This comes from the fact that the game is zero-sum.

2. Going from the position p, not to observed position q, but to a *random* position $p \rightarrow r$, we must have $f(r) > f(q)$ because the random move is better for the next player and worse for the player that made the move.

If this reasonable assumptions hold, a loss function that expresses the equality in (1) and the inequality in (2) can be constructed.

5.3.1 Loss function

The loss function is sum of the negative log-likelihood of the inequalities: $f(r) > f(q)$, $f(p) > -f(q)$ and $f(p) < -f(q)$. The last two are a way to express the equality $f(p) = -f(q)$. Each term is the negative log-likelihood function of the known Bradley-Terry model [2], that models the probability of an item (in our case a position) "beating" another item.

The loss function is given by

$$\mathcal{L}(x^p, x^q, x^r, \boldsymbol{W}) = \frac{1}{N} \sum_i^N - \log\left(\sigma(\hat{r}_i - \hat{q}_i)\right)$$
$$- \log\left(\sigma(\hat{p}_i + \hat{q}_i)\right)$$
$$- \log\left(\sigma(-(\hat{p}_i + \hat{q}_i))\right)$$

where. . .

1. N is the number of samples.

2. x^i are the inputs (encoded feature sets) for the $i \in \{p, q, r\}$ positions.

3. f is the model.

4. $\boldsymbol{W}$ are the parameters of the model.

5. $\hat{p}_i = \frac{f(x_i^p, \boldsymbol{W}) - a}{b}$, $\hat{q}_i = \frac{f(x_i^q, \boldsymbol{W}) - a}{b}$, $\hat{r}_i = \frac{f(x_i^r, \boldsymbol{W}) - a}{b}$, where a and b are the parameters of the WDL model. Note that quantization is happening in this method too, so the output of the model must be scaled appropriately.

Let's break down the loss function in a more intuitive way. We want the loss function to be small when the model is generating the correct evaluations and large when it is not. Let's look at the graph of the function $-\log(\sigma(x))$:
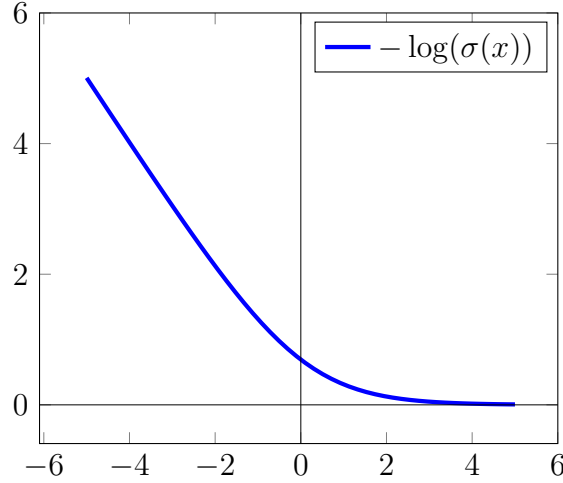
The function approaches 0 when x grows, and approaches ∞ when x goes to $-\infty$. Let's look at each of the terms:

1. $-\log(\sigma(\hat{r} - \hat{q}))$: This term is small when $\hat{r} > \hat{q}$, and large when $\hat{r} < \hat{q}$.

2. $-\log(\sigma(\hat{p} + \hat{q}))$: This term is small when $\hat{p} > -\hat{q}$, and large when $\hat{p} < -\hat{q}$.

3. $-\log(\sigma(-(\hat{p} + \hat{q}))$: This term is small when $\hat{p} < -\hat{q}$, and large when $\hat{p} > -\hat{q}$.

The term (1) holds the inequality $f(r) > f(q)$, and the terms (2) and (3) hold the equality $f(p) = -f(q)$. The loss function is the sum of the three terms, so the model is encouraged when it satisfies the inequalities and penalized when it does not.

5.4 Setup

The project is written in two languages: Rust and Python. The Rust part is used to process dataset files, generate statistics and provide final training batches for Python to

consume. The Python part defines the Pytorch model, runs the training loop, quantizes the model and runs the evaluations.

The training process is started by running a Python script (`scrips/train.py`) and it requires to define the model architecture (number of neurons on each layer), general training parameters (learning rate, batch size, epochs, checkpoints, etc) and the feature set to use, which in turn determines the size of the batches. For example, if PQR is used, the size of a sample is 3 times the size of the feature set times two (because it is siamese), and if it is score target, it is the size of the feature set times two plus 1 for the target score.

To orchestrate training runs, the platform Weights and Biases (WandB) is used. It provides automatic sweeping of hyperparameter, logging of metrics and visualizations. Results are exported from the platform in CSV and then processed by Python scripts.

The training data has to be converted to an actual tensor of floats to be consumed by Pytorch. This is done by a Rust subprocess running the subcommand `batch-loader` that read the training data file and generates training batches for the specified feature set in a shared memory buffer.

The batch generation process is heavily parallelized. Let's call N the number of threads ($N = 8$ was used). When the process starts, it splits the dataset file into N equal parts and assigns each part to a thread. Each thread reads samples sequentially and builds the batch in a buffer. The buffer is then sent to the main thread where it is copied to the shared memory buffer.

The Python script copies the data from the shared buffer at the start of each iteration, allowing Rust to generate the next batch (in the CPU) while Pytorch is training the current batch (in the GPU). To coordinate the memory access between the two processes, a single byte is sent using standard I/O. The sequence of a training loop is shown in Figure 10.

Figure 10. Sequence of steps to send a batch from the `batch-loader` subprocess in Rust to Pytorch.

Given that the input vector is multiple-hot encoded, the data written by the Rust process are not float values. Instead, they are 64-bit integers acting as a bitset. Before passing the vector to the model, it is expanded into floats. This means 64 floats can be packed into a single 64-bit integer, meaning a **96.875%** reduction in memory usage (from 256 to 8 bytes). The speedup obtained by this optimization was substantial. The compression can be further improved using sparse tensors, but it is not implemented in this work.

6   Experiments and results

Now that the engine, the tools and the methodology are defined, we can proceed to the experiments. Experiments will be divided in three sections: motivation, experiment and results. The motivation will explain why I think the experiment is relevant and present possible hypothesis. The experiment will describe configurations to train different models, how they will be evaluated and what are my expectations. The results will present the data, explain whether my hypothesis was correct or not and give a brief conclusion.

Every model's training configuration is defined by the following variables:

- **Feature set**: Determinates the encoding of the position, and thus the number of inputs of the model. It conditions which patterns the network can learn. Experimenting with this is the main focus of this thesis.

- **Network architecture**: The size of each layer in the network. The first layer (L1) is the feature transformer and it is efficiently updated. The following layer (L2) should be tiny due the NNUE architecture. The size of the model (its complexity) roughly determinates how many patterns the network can learn.

- **Dataset**: The positions to train on. The dataset used is explained in detail in chapter 5. In summary, there are 48.5 billion positions to train on and the dataset remains constant across all runs. About 5 million positions are used for validation.

- **Training method**: Can choose to use either score targets or PQR triplets. This determinates the format of the samples as well as the loss function. All experiments will train using score targets, unless specified. Methods were explained in detail in chapter 5.

- **Training hyperparameters**: The usual machine learning hyperparameters for training, such as batch size, learning rate and scheduler. I used the same epoch size used in Stockfish, where each epoch is 100 million positions. Each training run will last for 256 epochs, which means the network is trained in 25.6 billion positions (recall that some of the original 48.5 billion dataset are skipped).

Once training is completed, the models will be evaluated depending on the experiment. To assess the performance of a model or to compare a set of models, the following indicators are used:

- **Loss**: The training and validation loss are used to detect overfitting and other possible problems. It can't be used to measure the performance of a model. Bigger models must have much better predictions to outweight the cost of having slower inferences and thus less node visits. It's a tradeoff.

- **Puzzle accuracy**: The percentage of moves correctly predicted by the engine in Lichess puzzles. Each puzzle may contain multiple moves, and the engine has 100ms per move. Since the engine is not that strong, it does not solve 100% of puzzles like many other engines do, so I expect differences in this metric to be good indicators. A small set of puzzles is used during training as (a very bad) proxy for the engine's strength, to have early insight of the strength and to detect catastrophic failures that did arise. A bigger set of 85000 puzzles is used after training.

- **Relative ELO rating**: A tournament is played between different models to determine their relative strength. Ordo is used to compute the ELO of each model based on the results of the tournament. This is the most important metric, as it is the most reliable way to compare the strength of engines.

- **Training duration**: The amount of time it takes to train a model. This is a one time operation and it does not affect the performance of a model. However, it does condition which and how many experiments I can run.

All networks that are not in the first experiment (the baseline), are trained 4 times and a tournament is played between the epoch 192 and 256 of each network (8 networks in total). I have observed a difference of 30 elo points between runs, so this step is crucial to have sensible results. In the appendix are the results of each run and tournament.

6.1 Baseline

Motivation. Experiments that will follow will focus on trying out different feature sets, so it is natural to keep every other variable constant. Since the dataset is fixed and the feature set is going to be changing, it remains to find acceptable values for the network architecture and the training hyperparameters.

Due time and resources constraints, I decided to set the training hyperparameters to (similar) values which give good results in the official Stockfish trainer: **a batch size of 16384, a learning rate of 0.0005 and a exponential decay factor of 0.99**. This values showed acceptable results during early stages of development and will remain fixed for all runs.

It remains to find a good network architecture. Bigger networks may have lower loss and predict better, but they will also have slower inferences. This is the tradeoff between inference time and node visits (more depth), which are also affected by the quality of the prediction due to better pruning. So the model must be so much better to compensate the slowdown in inference.

Experiment. In this first experiment I will try different sizes of L1 and L2, to find an acceptable tradeoff for future experiments. I will run a grid search with L1 $\in \{256, 512, 1024, 2048\}$ and L2 $\in \{32, 64, 128, 256\}$. The feature set used to train will be ALL, the canonical set with 768 features.

I expect that there will be a model that performs best and other models that are smaller (need stronger predictions) and bigger (need speed to visit more nodes) perform worse.

Results. Looking at the result heatmaps in Figure 11, the first thing to notice is that training and validation losses behave as expected. If the model is more complex, meaning the number of parameters (which is dominated by $768 * L1 + L1 * L2$) is higher, the loss is lower and the model predicts better.

When the models are loaded into the engine and evaluated in a tournament, we can see that when L2 drops, the performance drops dramatically. This is due the fact that the inference time is mostly dominated by L2. This result suggests that it may be a good idea explore even lower values of L2, such as 16 or even 8. However, the SIMD implementation requires L2 to be a multiple of 32 so it needs a refactor to keep being fast. So, instead of fiddling further with SIMD I decided to **keep L2 at 32**.

Figure 11. Network architecture sweep results (L1 × L2).
Table with details in Appendix A.1.

If L2 is kept constant, the best L1 is not the smallest nor biggest. If L2 = 64 or L2

= 128 there is a clear lead of L1 = 512 in both. In the case of L2 = 32, the best L1 is not clear because the differences in rating are small and are within margin of error, excluding L1 = 256 which is definitely wrose. Because training lower values of L1 is faster I opted for **L1 = 512** due the difference being small and being the best in other L2 values.

So, further experiments will use L1 = 512 and L2 = 32. For reference, Stockfish currently uses L1=2560, and employ (lots of) more tricks to make it even faster. The values selected here are specific to the current implementation of the engine, since it may change if more optimizations are made (tradeoff is altered). For this reason, no further modifications to the engine were made after starting with the experiments. We can now proceed with more interesting experiments.

6.2 Axis encoding

Motivation. Looking back at the networks generated by ALL in baseline runs, the learned weigths of most neurons in the feature transformer layer (L1) are related with the movement pattern of the pieces. Let's take the example in Figure 12, which depicts the SQUARE part of the features where the role is ♖ Rook.

(a) ○ White   **(b)** ● Black

Figure 12. Weights of **a neuron** in the L1 layer, which are connected to features in ALL where the role is ♖ Rook. The intensity represents the weight value, and the color represents the sign (although not relevant).

This particular neuron learned to recognize the presence of a ♖ Rook, affected by the pattern of another potential rook in the same file or rank (other pieces may be involved but I am focusing on rooks for the example). Doing so, it had to relate one feature for every potential square where a rook could be for that specific center location, which restrains the

network from learning more complex patterns and it is harder to train, because you need more samples to account for all possible combinations.

What if we add a feature which describes *"there is a ○ White ♖ Rook in the 4th rank"*? Certainly, this would make the network's job easier, as it would only need to learn the presence of rooks in the corresponding file or rank, instead of every square. This idea can be extrapolated to diagonals, to ease patterns with ♗ Bishops and the ♕ Queen.

More examples of this behaviour can be found in Appendix A.2.1, showcasing diagonal patterns and the ♘ Knight movements, although they do not move straight through axes.

Experiment. I built blocks of features for each natural axis of a chess board, which coincide with the movement pattern of the pieces:

↔	↕	↗	↘
Horizontal	Vertical	Diagonal 1	Diagonal 2
(across files)	(across ranks)		
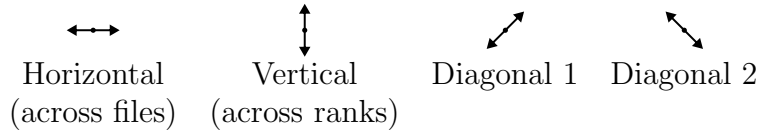
In table 1 I present the feature blocks. Each block will encode whether there is a piece with the role and color in a specific location along that axis, as explained in the example.

Table 1. Axis feature blocks

Depiction	Block name	Definition			Number of features
↔	H	(FILES	× ROLES × COLORS)$_P$		96
↕	V	(RANKS	× ROLES × COLORS)$_P$		96
↗	D1	(DIAGS1	× ROLES × COLORS)$_P$		180
↘	D2	(DIAGS2	× ROLES × COLORS)$_P$		180

$P(\langle x, r, c \rangle)$: there is a piece in x with role r and color c

With this blocks, I built different feature sets (listed in table 2): one group of feature sets is just combinations of all the blocks, and another group which is the same as the first but alongside the ALL feature set. The second group is the aim of the experiment, it has the classic ALL feature set but includes the axis blocks to see if the network can benefit from them. The first group, which does not include ALL is to know how far the network can go only with the blocks presented.

Table 2. Axis feature sets

Depiction	Feature set	Number of features
↔ ⊕ ↕	H ⊕ V	192
↗ ⊕ ↘	D1 ⊕ D2	360
↔ ⊕ ↕ ⊕ ↗ ⊕ ↘	H ⊕ V ⊕ D1 ⊕ D2	552
ALL ⊕ ↔ ⊕ ↕	ALL ⊕ H ⊕ V	960
ALL ⊕ ↗ ⊕ ↘	ALL ⊕ D1 ⊕ D2	1128
ALL ⊕ ↔ ⊕ ↕ ⊕ ↗ ⊕ ↘	ALL ⊕ H ⊕ V ⊕ D1 ⊕ D2	1320

I expect that the feature sets that are sums of single axes (↔ ⊕ ↕ , ↗ ⊕ ↘ and ↔ ⊕ ↕ ⊕ ↗ ⊕ ↘) will perform worse overall, since to capture the exact position of pieces in the board, the network will have to learn to relate at least two features for every location. This information is already available when ALL is present.

The feature sets that include ALL (ALL ⊕ . . .) should perform better than without, providing that the idea explained in the motivation holds.

For each of the proposed feature sets, I will train a network and evaluate its performance relative to each other using a tournament. I expect to see them ranked in the reverse order as presented in the table (more extra axes better).

Results. The results in table 3 show that indeed, adding the axis blocks makes the network validation loss slightly lower, from 0.00313 in ALL to 0.00306 including all four blocks. However, this improvement in loss is not significant enough to make the engine stronger to compensate the (small) performance hit of having more features. As you can see in the table, including more axes makes the loss decrease slightly yet the rating decreases from almost no difference to a huge factor.

All three feature sets that do not include ALL unsurprisingly perform much, much worse even having less features (thus being faster). The feature set H+V+D1+D2 has a 25% higher loss than ALL and 183.5 ± 4.1 less rating than ALL. The other feature sets in this group perform even worse, as it was expected.

I discovered that the accuracy of puzzles is not a good a proxy of an engine's strength, given that there is a 444 rating difference yet 3% a difference in move accuracy. I believe that the reason lies on the fact that puzzles may be more strategic than positional. I will drop the puzzle accuracy metric in future experiments.

Table 3. Axis feature sets results

Feature set	Number of features	Val. loss min	Rating elo $(rel.\ to\ \text{ALL})$	Puzzles $move\ acc.$
↔ ⊕ ↕	192	0.005810	-384.3 ± 5.1	0.8618
⤢ ⊕ ⤡	360	0.006707	-444.1 ± 5.1	0.8517
↔ ⊕ ↕ ⊕ ⤢ ⊕ ⤡	552	0.003907	-183.5 ± 4.1	0.8748
ALL (reference)	768	0.003134	**0.0**	0.8865
ALL ⊕ ↔ ⊕ ↕	960	0.003082	-27.1 ± 4.1	0.8851
ALL ⊕ ⤢ ⊕ ⤡	1128	0.003087	-26.1 ± 3.8	0.8814
ALL ⊕ ↔ ⊕ ↕ ⊕ ⤢ ⊕ ⤡	1320	**0.003067**	-58.7 ± 3.7	0.8766

The next experiment will focus on adding more specific features, instead of more broad ones.

6.3   Pairwise axes

Motivation. Imagine that in a file there are three pieces: an enemy ♖ Rook, a ♙ Pawn and a ♘ Knight. There are many possible configurations for these pieces on the file. The influence in the evaluation by those pieces is very related with the position of pieces everywhere else, however I want to see if to understand a single file, the actual position of the pieces is less important than the relative order between them: ♙♘♖, ♙♖♘, ♘♙♖, ♘♖♙, ♖♙♘, ♖♘♙. In other words, provide the network features based on the order of the pieces instead of the actual position. This way, I believe that the network can pick up whether pieces are pinned, protected by other pieces or can attack other pieces.

I propose to make a feature for each possible pair of adjacent role and color over an axis. Lets consider the a file (vertical axis), following the example before:

Different configurations,
similar situation

The same two features
(blue pair and red pair)
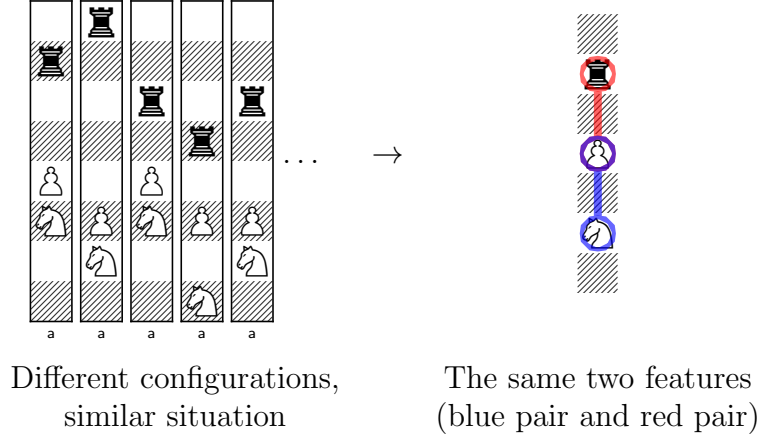
There are many configurations for the three pieces and the idea is to collapse all of these into two features: the pair of pieces (♜●, ♙○) and the pair of pieces (♙○, ♞○). This way, the network can learn that the ♜ Rook can capture the ♙ Pawn, and that the ♞ Knight is protected behind the ♙ Pawn. The network can learn this situation using two features instead of learning it for every possible configuration.

In contrast to the previous experiment where the features were more general (*"there is a ○ White ♜ Rook in the 4th rank"*) the proposed features here are more specific: *"there is a ● Black ♜ Rook next to a ○ White ♙ Pawn in the 'a' file"*.

Experiment. I developed two feature blocks: for the horizonal and vertical axis. The blocks are defined in table 4:

Table 4. Pairwise feature blocks

Depiction	Block name	Definition	Num. of features
○–○	PH	(RANKS × (ROLES × COLORS) × (ROLES × COLORS))$_P$ P($\langle r, r_1, c_1, r_2, c_2 \rangle$): there is a piece in rank r with role r_1 and color c_1 to the left of a piece with role r_2 and color c_2	1152
○ over ○	PV	(FILES × (ROLES × COLORS) × (ROLES × COLORS))$_Q$ Q($\langle f, r_1, c_1, r_2, c_2 \rangle$): there is a piece in file f with role r_1 and color c_1 below a piece with role r_2 and color c_2	1152

Note that it is important to consider the order of the pieces in the pair, as expressed in the direction of the definition (left and below). This makes sure features are not mirrored, since we want to differentiate between both. In code this is handled by iterating over the pieces and building the pair in the same order every time.

35

The following figure shows what pairs of pieces (features) are considered for the horizonal and vertical axes in a complete board:

O—O Pairwise horizontal (PH)   Pairwise vertical (PV)

Since the blocks need at least two pieces to generate a feature, if there is only one piece over an axis, there are no active features. So, this blocks can't be used alone, they need to be combined with other features that provide that information. The most obvious choice is to combine them with the ALL block.

The feature sets to be evaluated are ALL ⊕ PH (1920 features), ALL ⊕ PV (1920 features) and ALL ⊕ PH ⊕ PV (3072 features). Like before, a network will be trained for each feature set and a tournament will be played to determinate the relative elo to the ALL baseline.

I expect that the networks are able to take advantage of the specific features, enough to conteract the loss in performance due to the big increase in the number of features and slower updates.

Results. The results in table 5 show that there is a clear difference in performance between O—O and . The feature set ALL ⊕ has a lower loss and rating than its counterpart ALL ⊕ O—O. It is not clear why the vertical pairs achieve a better rating than the horizontal pairs, since they have a similar amount of feature updates (Appendix A.5).

Both ALL ⊕ and ALL ⊕ O—O perform worse than ALL. It seems that the networks were able to take advantage of the pairs, since the loss is lower than the reference. However, it is not enough to conteract the increase in feature updates.

Suprisingly the feature set with both axes (ALL ⊕ O—O ⊕) has a similar rating to ALL ⊕ , probably counteracted by having an even lower loss.

Table 5. Pairwise encodings results

Feature set	Number of features	Val. loss _min_	Rating _elo (rel. to_ ALL_)_
ALL (reference)	768	0.003134	**0.0**
ALL ⊕ O—O	1920	0.003033	-38.2 ± 4.8
ALL ⊕ ⑧	1920	0.002946	-8.4 ± 5.0
ALL ⊕ O—O ⊕ ⑧	3072	**0.002868**	-37.6 ± 4.9

Future work could gather some statistics about the pairs and determinate if skipping some pairs is worth it. For example, pairs related to pawns cause many updates since it is the most common piece and may not be that useful. Reducing the amount of pairs would lower the amount of updates and may overtake ALL.

I did not bother implementing diagonal pairs (⟋ and ⟍) due the adverse result of the other axes.

Up to this point, I have been trying to encode the position of the pieces in different or smarter ways, with no avail. It may seem that the network is able to extract all the information it needs from the most basic ALL feature set. Making the information available in another form makes no difference, as opposed to what I originally thought.

Further experiments will focus on features not related to the position of the pieces, but to other aspects of the game, inspired by hand crafted evaluations.

6.4 Mobility

Motivation. Mobility in chess is a measure of the available moves a player can make in a given position. The idea is that if a player has more available moves, the position is stronger. In [16] it was shown that there is a strong correlation between a player's mobility and the number of games won. This metric has been used extensively in hand crafted evaluations, and I propose to include this information as features for the neural network.

Experiment. There are two ways to go about encoding mobility:

- **Bitsets (per piece type):** Provide the exact squares each piece type can move to. The number of features would be $64 * 6 * 2 = 768$. The problem with this approach is not the amount of features, but the number of updates to the accumulator per move is very high, which slows down the search.

Board   ○White   Bishop   ●Black   Bishop   ○White   Queen   ...

- **Counts (per piece type):** Use the number of available moves per piece type as features. This means having a feature for each possible count value, which are a lot less feature updates. To find which values to include as features I computed the total mobility for each piece role in 2 billion boards, shown in figure 13. From the data we can extract the range of values to use as features:

Piece role	Min	Max
♙ Pawn	0	8+
♘ Knight	0	15+
♗ Bishop	0	16+
♖ Rook	0	25+
♕ Queen	0	25+
♔ King	0	8

Figure 13. Total mobility values for each piece on the board. Computed using 2 billion boards. The value 0 for the ♘ Knight, ♗ Bishop, ♖ Rook and ♕ Queen has been excluded from the plot, as it is very common.

Each approach was implemented as a block:

Table 6. Mobility feature blocks

Block name	Definition	Number of features
MB	(SQUARES × ROLES × COLORS)$_P$ P($\langle s, r, c \rangle$): there is a piece of role r and color c that **can move to** square s	768
MC	($\{0, 1, \ldots\}$ × ROLES × COLORS)$_P$ P($\langle m, r, c \rangle$): the value of mobility for a piece of role r and color c is m	206

The blocks will be combined with the ALL feature set. Neither of the blocks can be used alone, since they do not carry the information to deduce every piece on the board (trivially).

The feature sets to be trained and evaluated are ALL ⊕ MB (1536 features) and ALL ⊕ MC (974 features). Like prior experiments, a network will be trained for each feature set and evaluated in a tournament.

Results. The results in table 7 show that the blocks did not provide features of enough quality to improve the network predictions enough to overcome the cost of making more feature updates. I have underestimated the cost of a feature update, that was not a problem in previous experiments. The block MB has, in average, 10.03 feature updates per move and the block MC has 3.82 (see Appendix A.5). In comparison, the ALL feature set has 1.58 feature updates per move. Even though the block MB has almost 3 times the number of feature updates, it has a better performance than MC. This is attributed to having a 7% lower loss, which compensates the cost of the updates.

Table 7. Mobility encodings results

Feature set	Number of features	Val. loss *min*	Rating *elo (rel. to ALL)*
ALL (reference)	768	0.003134	**0.0**
ALL ⊕ MB	1536	0.002824	-260.9 ± 5.4
ALL ⊕ MC	974	0.003032	-280.9 ± 5.6

6.5 PQR

Motivation. During the initial research for a thesis subject, I came across [1] which seemed an interesting approach to train a neural network to evaluate positions. Since it was released in 2014, it predates the NNUE era and the training data was suboptimal (Lichess database [9] with human moves). So I decided to try to replicate the idea using modern datasets, better moves and a proper engine. The "PQR" method itself was explained in detail in the previous chapter. Remember that p is a position in the dataset, q is the position obtained by making the best move according to the dataset and r is a random position obtained making a random move from p such that $r \neq q$.

Before starting the experiment, I checked if existing networks trained with the conventional method behave under the principles of the PQR method: $f(p) = -f(q)$ and $f(r) > f(q)$. In the left plot of figure 14, we can see that values of $f(p)$ and $f(q)$ are negatively correlated, which supports the principle that $f(p) = -f(q)$. In the right plot, we can see that the distribution of the difference between $f(r)$ and $f(q)$ is mostly positive, which supports the principle that $f(r) > f(q)$. This shows that the principles that the PQR method relies on are properties that manifest in existing models.
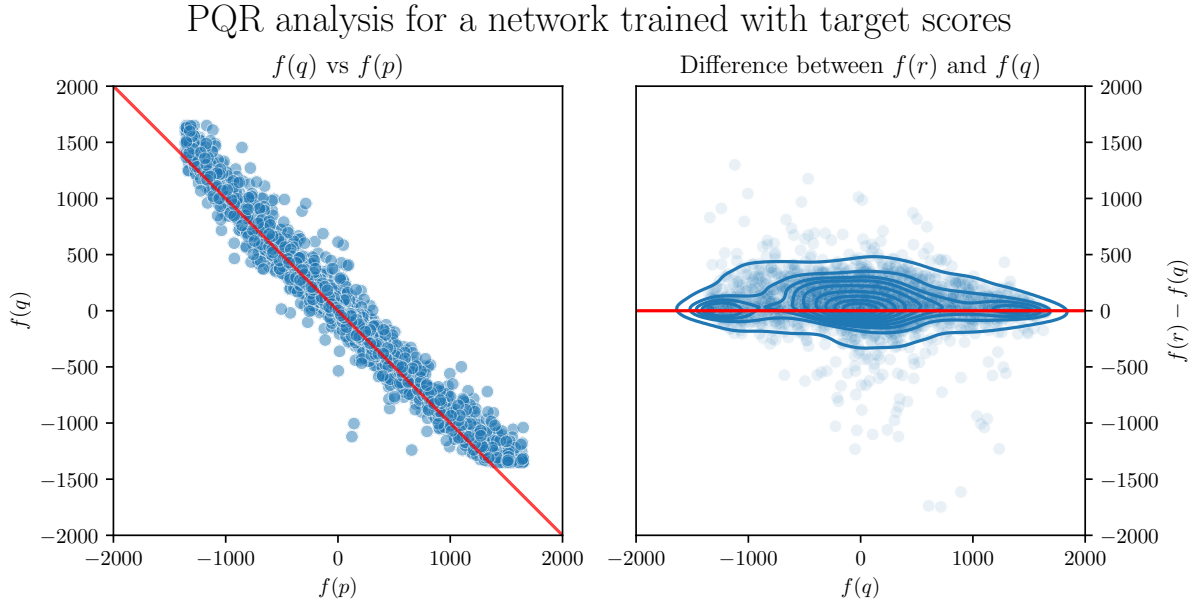

PQR analysis for a network trained with target scores

Figure 14. Analysis of $N = 4000$ PQR positions using a model trained with target scores and the feature set ALL.

Experiment. I will train the canonical ALL feature set with this method in two ways:

- **Train from scratch.** The network is initialized with random weights and trained with the PQR method. This is what the original authors did, and I do not expect to reach the performance of models trained with the evaluations method. Using

precomputed evaluations as a target is a lot simpler for the model, since it only has to learn to mimic the scores.

- **Train from a checkpoint.** A strong checkpoint trained with the other method is used to initialize the network. This way, the network does not have to learn too much at once and may enable it to improve the existing parameters. I believe that two scenarios are likely to happen: the model improves very slowly, or it completely forgets what it have learned before and ends up like a model trained from scratch. The best scenario is that the model improves slowly, proving that it can be used to further optimize existing models.

Results. After training a single network from scratch, the results do not look promising. Looking at figure 15, we can see that the network improves steadily up to the epoch 32 (trained on 3.2 billion samples), but then it starts to get consistently worse. There are no indications of overfitting, as the validation loss is still decreasing.
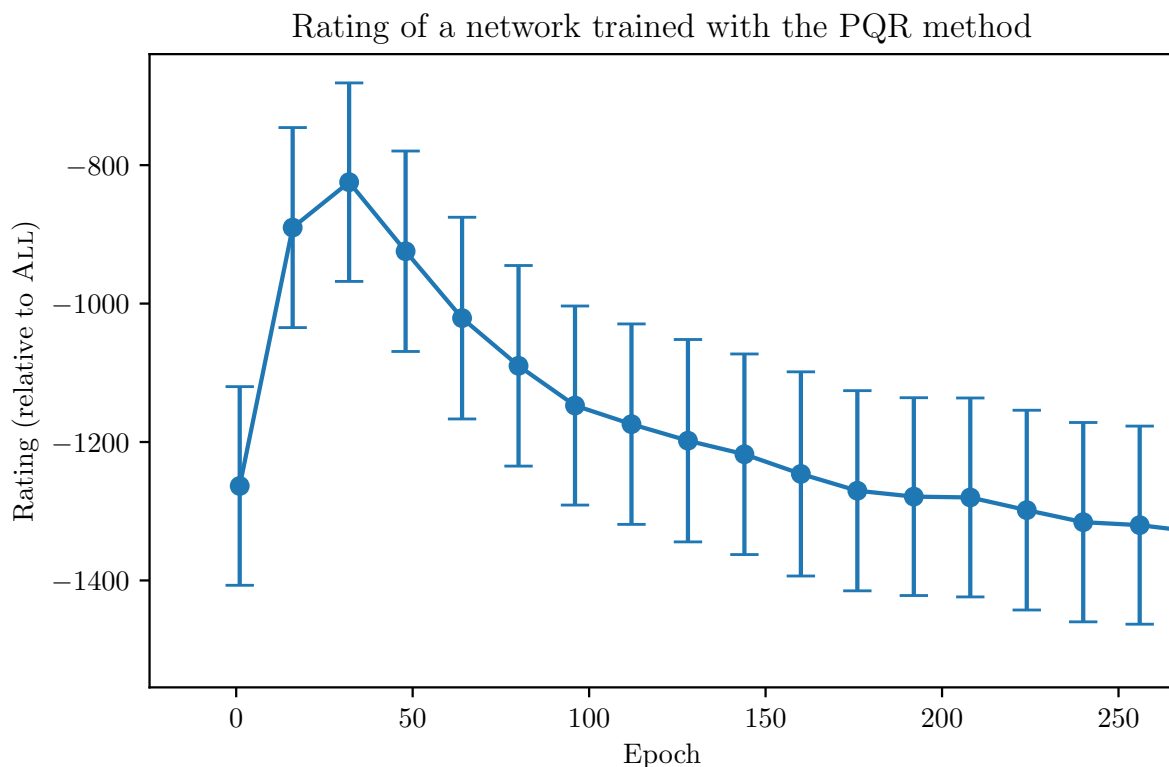

Figure 15. Rating evolution of a network trained with the PQR method from scratch.

To investigate, I tried comparing the loss of
.

seguir...

Positions that are very similar in score f(q) f(r) are being penalized the same way that more extreme ones. A way to mitigate this is generate a dataset which takes into account multiple lines of play, and only include samples where there is a significant difference between the moves.

—-

Further work:

* change the filter on the minimum number of moves to draw during training * lines of play * data with more depth

7 Final words

7.1 Conclusions

The main takeaway from this work is that the ALL feature set is at a sweetspot that is hard to beat. Bigger feature sets are known to be more effective, like KING-ALL, but this work was restricted to feature sets with a few thousands features to be practical to train.

The feature set ALL has all the information of the board, it is easy to implement, cheap to compute which features are active, and it has a low number of feature updates per move (1.58). This makes it very effective, and thus very fast when used with a NNUE network.

When building other feature sets, I have underestimated the cost of extra feature updates. The performance hit of making more updates is, in the feature sets I tested, greater than the performance gain of having more information available. In the end, NNUE networks are trying to reduce the number of dot products, which is what each feature update does.

It has been a few years since the introduction of NNUE networks and feature sets, in particular variations of the KING-ALL feature set, which uses at its core the ALL feature set. The community has not found a better feature set or additional blocks of features to add alongside it. Not only new features need to improve the network prediction, but also have few updates per move and be fast to compute, which is a hard balance to achieve. To make it worse, feature engineering in this kind of taks is very slow. Each iteration takes hours (or days) of training and evaluations to see if there is a (usually small) improvement.

[...]
[hablar en la conclusion de PQR]

7.2 Future work

Training NNUEs is a daunting task, and there are lots of variables that affect dramatically the performance of the networks. Many decisions were made in this work to reduce the scope of the project, so naturally many variables were left unexplored.

The following are some key points that could be explored in a future work:

Dataset: A great deal of effort is put into good training data. The training data I used was generated using very specific parameters: depth 9, 5000 nodes and selected opening books. It is known that higher depth data results in worse networks. It is believed that the reason is that data becomes too hard for the network to learn. Also datasets generated with different books also affect the performance of the network. Generating new data is a very slow process so it is harder to experiment with, which means that not that much reasearch has been done in this area.

Filtering of the data (skipping checks, captures, etc.) also affects the performance dra-

matically, but it is a lot easier to work with since it can be done after the data has been generated. New filtering conditions can be tried.

Alternative to PQR: Instead of the loss function used to train PQR, the triplet loss function could be tried, where the anchor is the P position, the positive is the observed position (Q) and the negative is the random position (R). I don't expect this to improve the that much, but it is worth trying.

Network architecture: The architecture of NNUE-like networks has gone through multiple iterations since its inception. This work focused on the first and most basic iteration of it. Maybe it is worth exploring more complex architectures with a fixed feature set rather than a fixed architecture with a variable feature set. Almost certainly try lower values of L2, which may bring better results.

Feature sets: There are many aspects of the game that could be tried as features. A good place to start looking for new features are existing handcrafted evaluations. I had many ideas for new feature sets but I had to discard them because the thesis was already too long.

References

[1] Erik Bernhardsson. *Deep learning for... chess*. 2014. URL: https://erikbern.com/2014/11/29/deep-learning-for-chess.html.

[2] R. A. Bradley and M. E. Terry. "A rank-ordering method for comparing the individual performances of competitors". In: *Biometrika* (1952). DOI: 10.2307/2334029.

[3] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* (2012). DOI: 10.1109/TCIAIG.2012.2186810.

[4] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* (2002). DOI: 10.1016/S0004-3702(01)00129-1.

[5] Rajiv Chandrasekaran. "Kasparov Proves No Match for Computer". In: *Washington Post* (May 12, 1997). URL: https://www.washingtonpost.com/wp-srv/tech/analysis/kasparov/kasparov.htm.

[6] *Chess Programming Wiki*. URL: https://www.chessprogramming.org.

[7] Claude G. Diderich and Marc Gengler. "A Survey on Minimax Trees And Associated Algorithms". In: *Minimax and Applications*. Springer US, 1995. DOI: 10.1007/978-1-4613-3557-3_2.

[8] Ahmed Elnaggar et al. "A Comparative Study of Game Tree Searching Methods". In: *International Journal of Advanced Computer Science and Applications* (2014). DOI: 10.14569/IJACSA.2014.050510.

[9] *Lichess Database*. URL: https://database.lichess.org.

[10] Yu Nasu. "NNUE: Efficiently Updatable Neural-Network-based Evaluation Functions for Computer Shogi". In: *Ziosoft Computer Shogi Club* (2018). URL: https://www.apply.computer-shogi.org/wcsc28/appeal/the_end_of_genesis_T.N.K.evolution_turbo_type_D/nnue.pdf.

[11] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2021. ISBN: 9780134610993.

[12] Claude Shannon. "Programming a Computer for Playing Chess". In: *Philosophical Magazine* (1950). URL: https://d1yx3ys82bpsa0.cloudfront.net/chess/programming-a-computer-for-playing-chess.shannon.062303002.pdf.

[13] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* (2018). DOI: 10.1126/science.aar6404.

[14] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: (2017). DOI: 10.48550/arXiv.1712.01815.

[15] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* (2017). DOI: 10.1038/nature24270.

[16] Eliot Slater. "Statistics for the Chess Computer and the Factor of Mobility". In: *Proceedings of the Symposium on Information Theory, London* (1950).

[17] *Stockfish*. URL: https://stockfishchess.org.

[18] Official Stockfish. *nnue.md*. URL: https://github.com/official-stockfish/nnue-pytorch/blob/master/docs/nnue.md.

[19] Official Stockfish. *WDL model*. URL: https://github.com/official-stockfish/WDL_model.

[20] *Stockfish NNUE training data*. URL: https://robotmoon.com/nnue-training-data.

[21] Maciej Świechowski et al. "Monte Carlo Tree Search: a review of recent modifications and applications". In: *Artificial Intelligence Review* (2022). DOI: 10.1007/s10462-022-10228-y.

A   Appendix

The experiments are all run in the same hardware: Intel 14900K CPU (24 cores, 32 threads) for dataset generation, batching and evaluation, and a single NVIDIA RTX 4090 24GB GPU for training.

Runtime may be affected by other processes running on the machine, since it was my everyday computer. They are listed here for reference.

Tournaments are held with 100ms per move, and the opening book used is `UHO_Lichess_4852_v1.epd`. Each network plays **at least** 10000 games. Ratings are computed using Ordo, relative to the average (rating=0 is the average) or to the best network (rating=0 is the best network), depending on the experiment.

A.1   Baseline

Table 8. Network architecture sweep results (L1 × L2)

Feature set	Train hyperparams			Network		Val. loss	Rating	Puzzles	Runtime
	Batch	**LR**	**Gamma**	**L1**	**L2**	*min*	*elo (avg=0)*	*move acc.*	*hh:mm:ss*
ALL	16384	5e-04	0.99	256	32	0.00351	86.3 ± 5.1	**0.9047**	1:53:59
ALL	16384	5e-04	0.99	256	64	0.00342	21.4 ± 4.9	0.8976	1:54:56
ALL	16384	5e-04	0.99	256	128	0.00330	-46.2 ± 5.5	0.8885	1:52:29
ALL	16384	5e-04	0.99	256	256	0.00319	-68.1 ± 6.1	0.8826	2:29:26
ALL	16384	5e-04	0.99	512	32	0.00309	105.8 ± 4.9	0.9027	1:54:28
ALL	16384	5e-04	0.99	512	64	0.00300	58.3 ± 5.0	0.8975	1:53:44
ALL	16384	5e-04	0.99	512	128	0.00290	13.2 ± 5.7	0.8880	1:51:06
ALL	16384	5e-04	0.99	512	256	0.00279	-73.6 ± 5.2	0.8790	1:51:17
ALL	16384	5e-04	0.99	1024	32	0.00268	**114.1 ± 5.8**	0.9032	2:15:18
ALL	16384	5e-04	0.99	1024	64	0.00265	50.5 ± 5.8	0.8955	2:03:41
ALL	16384	5e-04	0.99	1024	128	0.00257	-19.1 ± 6.2	0.8852	2:06:39
ALL	16384	5e-04	0.99	1024	256	0.00246	-109.4 ± 6.4	0.8725	2:32:47
ALL	16384	5e-04	0.99	2048	32	0.00241	104.0 ± 6.0	0.8968	3:11:56
ALL	16384	5e-04	0.99	2048	64	0.00238	30.0 ± 5.1	0.8876	3:12:46
ALL	16384	5e-04	0.99	2048	128	0.00234	-80.6 ± 5.6	0.8779	3:29:07
ALL	16384	5e-04	0.99	2048	256	**0.00221**	-186.6 ± 6.3	0.8678	3:27:47

Ratings are relative to the average (rating=0)
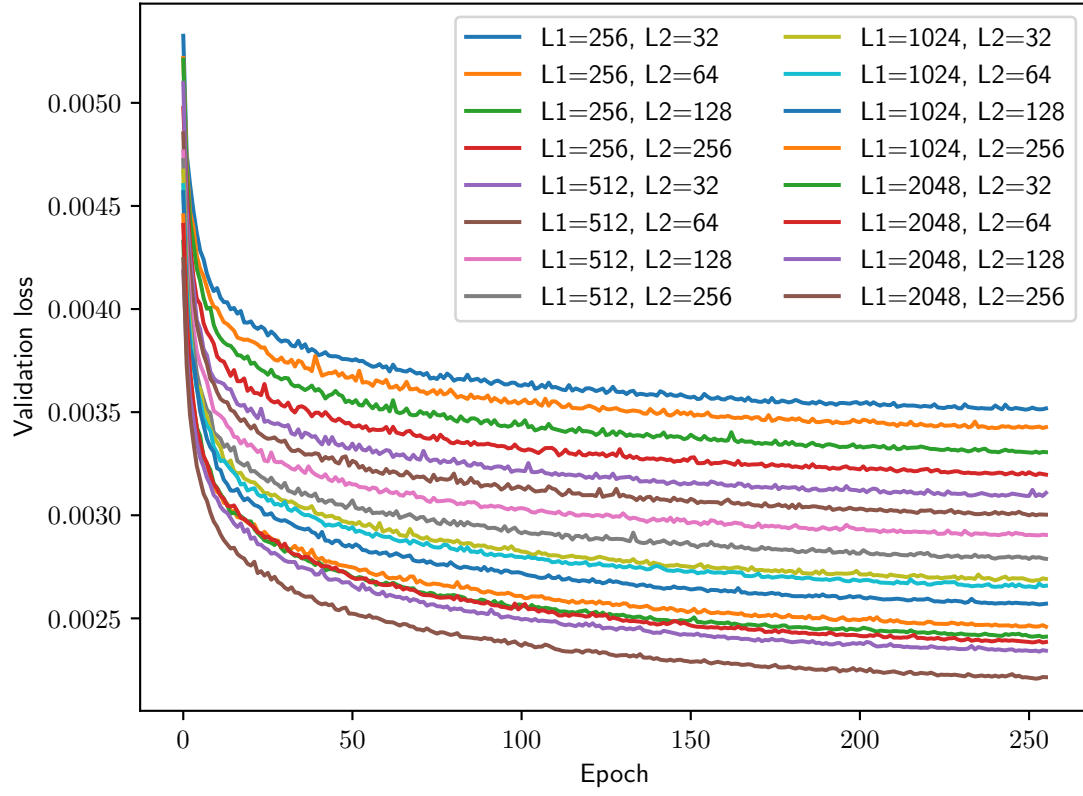
Figure 16. Network architecture sweep validation loss over epochs (baseline)

A.2 Axis encoding

A.2.1 Examples

(a) ○ White   **(b)** ○ White   **(c)** ○ White

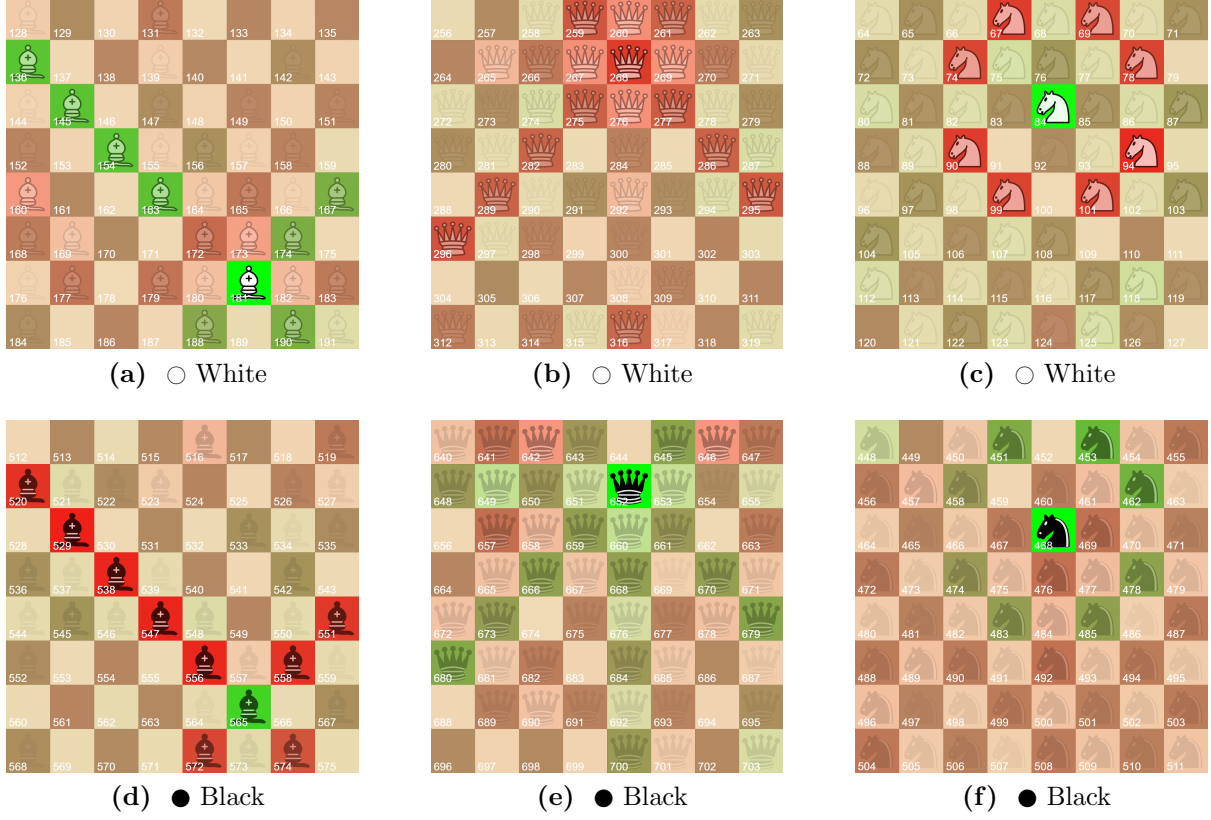**(d)** ● Black   **(e)** ● Black   **(f)** ● Black

Figure 17. Weights of different neurons in the L1 layer, which are connected to features in ALL with different roles. The intensity represents the weight value, and the color represents the sign. The number is the feature index, specifically VH instead of HV (both are ALL), because it was prior to the first experiment. Refer to section 6.2.

A.2.2 Preliminar runs

Table 9. Axis feature sets preliminar runs

Feature set	Run	Val. loss _min_	Runtime _hh:mm:ss_	Rating @ 192 _TC=100ms/m_	Rating @ 256 _TC=100ms/m_
D1 + D2	1	**0.006707**	1:44:25	2.1 ± 4.3	**13.5 ± 4.6**
	2	0.006716	1:45:46	-3.9 ± 5.1	-0.5 ± 5.0
	3	0.006729	1:47:58	-4.7 ± 4.8	-1.6 ± 5.3
	4	0.006721	1:51:24	-0.9 ± 5.3	-4.0 ± 5.0
H + V	1	**0.005810**	1:42:35	-8.6 ± 5.2	**9.5 ± 5.5**
	2	0.005827	1:42:29	-2.6 ± 5.4	-6.5 ± 5.1
	3	0.005816	1:42:59	4.8 ± 4.8	2.4 ± 5.4
	4	0.005825	1:43:13	-6.3 ± 4.9	7.4 ± 5.2
H + V + D1 + D2	1	**0.003885**	2:26:05	-14.3 ± 4.9	-18.1 ± 4.3
	2	0.003907	2:27:30	7.2 ± 5.0	**15.4 ± 4.7**
	3	0.003905	2:27:35	0.1 ± 5.3	5.2 ± 4.4
	4	0.003906	2:45:19	5.7 ± 5.0	-1.2 ± 4.5
ALL	1	**0.003121**	1:30:34	-2.9 ± 4.7	4.6 ± 4.4
	2	0.003129	1:30:13	-4.2 ± 5.0	10.1 ± 5.4
	3	0.003134	1:30:14	-10.0 ± 5.2	**10.4 ± 5.1**
	4	0.003147	1:30:18	-9.6 ± 5.0	1.6 ± 4.8
ALL + D1 + D2	1	0.003093	2:06:54	-5.0 ± 4.4	1.7 ± 4.5
	2	**0.003087**	2:12:30	8.6 ± 4.3	**12.0 ± 4.7**
	3	**0.003087**	2:26:29	-3.1 ± 4.9	7.9 ± 3.9
	4	0.003095	2:38:25	-6.1 ± 4.5	-16.0 ± 4.4
ALL + H + V	1	0.003086	2:05:02	1.0 ± 4.8	9.0 ± 6.0
	2	0.003082	2:06:16	**12.9 ± 4.8**	7.1 ± 5.5
	3	**0.003079**	2:04:53	-14.6 ± 5.1	2.3 ± 5.5
	4	0.003085	2:07:18	-10.1 ± 4.9	-7.6 ± 4.4
ALL + H + V + D1 + D2	1	0.003071	2:49:23	-18.7 ± 4.9	4.3 ± 4.6
	2	0.003052	2:42:18	-6.6 ± 4.6	-0.6 ± 4.8
	3	0.003067	2:44:26	6.5 ± 4.8	**9.5 ± 4.6**
	4	**0.003050**	2:44:34	-2.9 ± 5.4	8.5 ± 4.6

Batch size: 16384, **LR**: 5e-04, **Gamma**: 0.99, **L1**: 512, **L2**: 32
Each tournament rating is relative to its average (rating=0)

A.2.3 Final results

Table 10. Axis feature sets final results

Feature set	Run	Epoch	Val. loss _min_	Rating _TC=100ms/m_
ALL	3	256	0.003134	**0.0 ± 0.0**
ALL + D1 + D2	2	256	0.003087	-26.1 ± 3.8
ALL + H + V	2	192	0.003082	-27.1 ± 4.1
ALL + H + V + D1 + D2	3	256	**0.003067**	-58.7 ± 3.7
H + V + D1 + D2	2	256	0.003907	-183.5 ± 4.1
H + V	1	256	0.005810	-384.3 ± 5.1
D1 + D2	1	256	0.006707	-444.1 ± 5.1

Batch size: 16384, **LR**: 5e-04, **Gamma**: 0.99, **L1**: 512, **L2**: 32
Tournament rating is relative to ALL (rating=0)

A.3 Pairwise runs

A.3.1 Preliminar runs

Table 11. Pairwise feature sets preliminar runs

Feature set	Run	Val. loss _min_	Runtime _hh:mm:ss_	Rating @ 192 _TC=100ms/m_	Rating @ 256 _TC=100ms/m_
ALL + PV	1	0.002954	1:56:52	-2.9 ± 4.5	3.8 ± 4.0
	2	0.002969	1:56:24	1.7 ± 4.7	3.3 ± 4.7
	3	0.002953	1:56:12	-8.7 ± 4.9	-1.4 ± 4.4
	4	**0.002946**	1:56:33	-8.8 ± 4.7	**13.0 ± 4.7**
ALL + PV + PH	1	**0.002860**	3:08:28	-17.9 ± 5.4	-5.4 ± 4.9
	2	0.002865	4:10:56	1.7 ± 5.6	7.8 ± 5.5
	3	0.002873	3:41:16	-5.3 ± 5.0	6.2 ± 5.1
	4	0.002868	3:41:38	0.8 ± 5.3	**12.0 ± 4.9**
ALL + PH	1	**0.003022**	1:57:13	2.4 ± 4.4	6.7 ± 4.6
	2	0.003023	2:31:41	-5.5 ± 4.6	1.8 ± 4.4
	3	0.003053	2:42:28	-21.7 ± 4.7	0.8 ± 4.2
	4	0.003033	2:44:46	7.5 ± 4.5	**8.0 ± 4.5**

Batch size: 16384, **LR**: 5e-04, **Gamma**: 0.99, **L1**: 512, **L2**: 32
Each tournament rating is relative to its average (rating=0)

A.3.2 Final results

Table 12. Pairwise feature sets final results

Feature set	Run	Epoch	Val. loss _min_	Rating _TC=100ms/m_
ALL + PV	4	256	0.002946	**-8.4 ± 5.0**
ALL + PV + PH	4	256	**0.002868**	-37.6 ± 4.9
ALL + PH	4	256	0.003033	-38.2 ± 4.8

Batch size: 16384, **LR**: 5e-04, **Gamma**: 0.99, **L1**: 512, **L2**: 32
Tournament rating is relative to ALL (rating=0)

A.4 Mobility runs

A.4.1 Preliminar runs

Table 13. Mobility feature sets preliminar runs

Feature set	Run	Val. loss _min_	Runtime _hh:mm:ss_	Rating @ 192 _TC=100ms/m_	Rating @ 256 _TC=100ms/m_
ALL + MB	1	**0.002809**	2:46:15	0.1 ± 5.7	-7.5 ± 5.9
	2	0.002824	2:51:49	-6.1 ± 6.1	**12.1 ± 6.5**
	3	0.002812	2:49:58	-2.4 ± 5.2	11.4 ± 5.2
	4	0.002828	2:53:24	-7.5 ± 6.0	0.0 ± 5.7
ALL + MC	1	0.003045	2:23:00	15.5 ± 5.2	-30.5 ± 5.3
	2	0.003060	2:22:55	-41.5 ± 5.5	2.6 ± 5.3
	3	0.003040	2:38:55	0.7 ± 5.4	12.6 ± 4.3
	4	**0.003032**	2:21:59	19.0 ± 5.6	**21.5 ± 4.9**

Batch size: 16384, **LR**: 5e-04, **Gamma**: 0.99, **L1**: 512, **L2**: 32
Each tournament rating is relative to its average (rating=0)

A.4.2 Final results

**Table 14.** Mobility feature sets final results

Feature set	Run	Epoch	Val. loss _min_	Rating _TC=100ms/m_
ALL + MB	2	256	**0.002824**	**-260.9 ± 5.4**
ALL + MC	4	256	0.003032	-280.9 ± 5.6

Batch size: 16384, **LR**: 5e-04, **Gamma**: 0.99, **L1**: 512, **L2**: 32
Tournament rating is relative to ALL (rating=0)

A.5 Feature set statistics

Table 15. Feature set statistics

Depiction	Feature block	Number of features	Average features...		
			active per position	added per move	removed per move
↔↕	ALL	768	14.68	0.98	0.60
↔	H	96	14.68	0.60	0.43
↕	V	96	14.68	0.61	0.43
↗	D1	180	14.68	0.77	0.52
↘	D2	180	14.68	0.77	0.52
o–o	PH	1152	8.23	0.92	0.57
⫯	PV	1152	8.30	0.83	0.53
MB	MB	768	48.93	5.68	4.35
MC	MC	206	12.00	2.34	1.48

To obtain the previous data, 100 million positions were visited. The average number of added and removed features per move is calculated iterating over every legal move of each position and counting the features that change after making the move.

A.6 `emitPlainEntry` code

```
void emitPlainEntry(std::string& buffer, const TrainingDataEntry& plain)
{
    buffer += plain.pos.fen();
    buffer += ',';
    buffer += std::to_string(plain.score);
    buffer += ',';
    buffer += chess::uci::moveToUci(plain.pos, plain.move);
    buffer += '\n';
}
```