



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Feature set analysis for chess NNUE networks

---

October 8, 2024

Martín Emiliano Lombardo  
mlombardo9@gmail.com

Directores

Agustín Sansone  
agustinsansone7@gmail.com

Diego Fernández Slezak  
dfslezak@dc.uba.ar



**Facultad de Ciencias Exactas y Naturales**

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón Cero + Infinito)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Conmutador: (+54 11) 5285-9721 / 5285-7400

<https://dc.uba.ar>

## **Abstract**

build a Stockfish-like engine  
train networks for different feature sets  
cry

## Agradecimientos

:)

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Thesis plan . . . . .	4
<b>2</b>	<b>Engine implementation</b>	<b>5</b>
2.1	Minimax search . . . . .	5
2.2	Quiescence search . . . . .	6
2.3	Optimizations . . . . .	6
2.4	Implementation details . . . . .	7
<b>3</b>	<b>Feature sets (board encodings)</b>	<b>8</b>
3.1	Sum $\oplus$ . . . . .	9
3.2	Indexing . . . . .	9
3.3	Feature sets . . . . .	9
3.3.1	ALL . . . . .	9
3.3.2	KING-ALL . . . . .	10
3.4	Dead features . . . . .	10
<b>4</b>	<b>Efficiently updatable neural networks</b>	<b>11</b>
4.1	Layers . . . . .	11
4.2	Efficient updates . . . . .	12
4.3	Network . . . . .	14
4.4	Quantization . . . . .	15
4.4.1	Stockfish quantization scheme . . . . .	15
4.5	Implementation . . . . .	17
4.5.1	Quantization error . . . . .	17
<b>5</b>	<b>Training</b>	<b>19</b>
5.1	Source dataset . . . . .	19
5.2	Method 1: Score target . . . . .	20
5.2.1	Score-space to WDL-space . . . . .	20
5.2.2	Loss function . . . . .	20
5.3	Method 2: PQR triplets . . . . .	20
5.3.1	Loss function . . . . .	21
5.4	Setup . . . . .	21
<b>6</b>	<b>Experiments and results</b>	<b>23</b>
6.1	Baseline . . . . .	24
6.2	Axis encodings . . . . .	26
6.3	Pairwise axes . . . . .	29
6.4	Mobility . . . . .	32
6.5	Attacks / Threats . . . . .	34
6.6	mas supongo? . . . . .	34

6.7	Symmetry? / Relativity? . . . . .	34
6.8	Statistical features? . . . . .	34
6.9	PQR . . . . .	35
<b>7</b>	<b>Final words</b>	<b>36</b>
7.1	Conclusions . . . . .	36
7.2	Future work . . . . .	36
<b>A</b>	<b>Appendix</b>	<b>39</b>
A.1	Dataset . . . . .	39
A.2	Baseline runs . . . . .	39
A.3	Axes encoding . . . . .	41
	A.3.1 Examples . . . . .	41
	A.3.2 Runs . . . . .	42
A.4	Pairwise axes runs . . . . .	42
A.5	emitPlainEntry code . . . . .	43

# 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 [3] was the first chess machine to reach superhuman level by consistently beating the world champion, Garry Kasparov, in 1997 [4]. 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 [6] and Monte Carlo Tree Search [2] (MCTS) or some of its variants [7, 18] to accomplish this.

The number of possible positions in chess is vast, estimated by Shannon [10] 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 [13] and its successor AlphaZero [12, 11] 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  $\mathcal{NNUE}$  (or NNUE) “Efficiently Updatable Neural-Networks” [9] 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 [15], 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<sup>1</sup>. Since Stockfish 16.1<sup>2</sup> (2024) the evaluation is done exclusively through NNUE networks, eliminating all human aspect.

---

<sup>1</sup>Introducing NNUE evaluation (Stockfish 12)

<sup>2</sup>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 for NNUE networks, called feature sets. These encodings are the input for a neural network. To do so, I needed a chess engine that supports neural networks with the ability to customize encodings and a way to train them. The initial idea was to use Stockfish with the official Pytorch trainer [16]. However, I quickly realized that implementing some of the features sets I had in mind may be too complicated with Stockfish’s representation and the unconventional training test (PQR) that I wanted to do was impossible. This, and given that Stockfish engine and trainer codebases are huge, I felt there was too much magic involved so I turned away. I could have picked up another less complex engine written in Rust (like Marlin) and modify it, but I choose not to.

So, I decided to implement my own engine and training pipeline from scratch.

## 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) [5] is the best 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.

Note that implementing the engine is not the main focus of this work, so some simplifications have been made to this chapter to keep it brief and simple.

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 with any other relevant information, like castling rights 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 determinate the best move 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 [6].

In a minimax tree search 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.

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. The best move is the one that leads to the highest evaluation at the root node.



Usually we don't 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 iterative deepening is implemented. Iterative deepening runs the search in a loop, starting 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't draw conclusions from any unfinished search, so the best move is the one found at the last iteration. This approach, when combined with a transposition table (a cache for evaluations) is very effective, making next iterations faster.

The actual implementation I did was 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 (prefers branches where the capture has not happened yet).

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 [5].

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 best move found so far. This means that it doesn't affect the result of the search, it only makes it faster.

Each node in the search tree has two values associated with it:  $\alpha$  and  $\beta$ .  $\alpha$  is the best value found so far that the maximizing player can guarantee up to that node.  $\beta$  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”  $\alpha$  up and the minimizing player tries to “pull”  $\beta$  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  $\beta$ , which is worse than the best value found so far.

- **Move ordering:** the order in which the moves are evaluated can have a big impact on the effectiveness of the alpha-beta pruning. 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 and 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 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.

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.

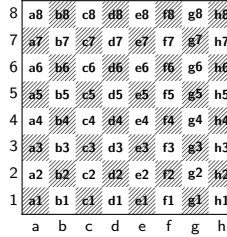
By default the engine uses 128 MB of memory for the transposition table. This value will be used throughout the experiments and in the arena.

### 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 (or feature block) is comprised of elements which represent some information about a chess position. A feature set is usually built by a cartesian product of smaller sets of features, to represent more complex patterns. Each element corresponds to a value in the input vector, which will be set to 1 if the pattern captured by the element is present in the position (we say that the element is *active*), and 0 otherwise.

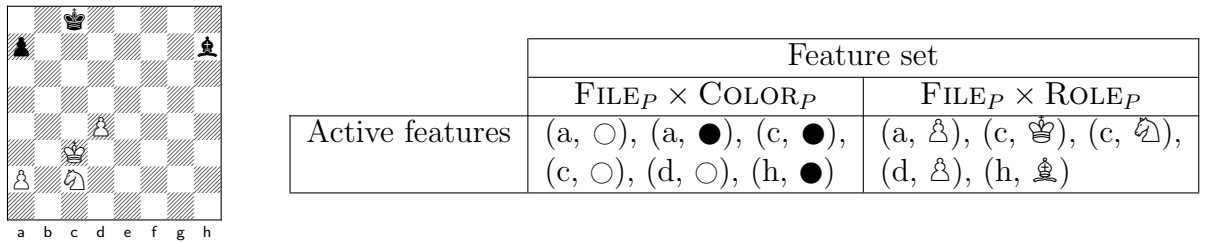
Let's consider some basic sets of features. The following sets encode positional information about the board:

$$\begin{aligned}\text{FILE} &= \{a, b, \dots, h\} \\ \text{RANK} &= \{1, 2, \dots, 8\} \\ \text{SQUARE} &= \{a1, a2, \dots, h8\}\end{aligned}$$


And the following encode information about the pieces:

$$\begin{aligned}\text{ROLE} &= \{ \text{♙ Pawn}, \text{♘ Knight}, \text{♗ Bishop}, \text{♖ Rook}, \text{♕ Queen}, \text{♔ King} \}^1 \\ \text{COLOR} &= \{ \text{○ White}, \text{● Black} \}\end{aligned}$$

Since each set has to capture some pattern from the position, it must be stated explicitly. For example, consider the feature set  $\text{FILE}_P \times \text{COLOR}_P$  where  $P$  is *any* piece in the board, meaning that the tuples  $(file, color)$  that will be active are the ones where there is at least one piece in  $file$  with the color  $color$  (disregarding any other kind of information, like the piece's role). Another possible feature set could be  $\text{FILE}_P \times \text{ROLE}_P$ , with a similar interpretation. Note that  $\text{SQUARE}_Q = \text{FILE}_Q \times \text{RANK}_Q \forall Q$ . An illustration of the active features of these two feature sets for the same board is shown in Figure 1.



**Figure 1.** Active features of the feature sets  $\text{FILE}_P \times \text{COLOR}_P$  and  $\text{FILE}_P \times \text{ROLE}_P$  for the same board.

<sup>1</sup>The color of the pieces have no meaning in the definition. They are present for illustrative purposes.

### 3.1 Sum $\oplus$

The sum (or concatenation) of two feature sets (often called blocks)  $A$  and  $B$ , denoted by  $A \oplus B$ , is a new feature set comprised of the elements of both sets  $A$  and  $B$ . These elements do not interfere with each other, even if they share basic elements (e.g. h, 8, ♖, ●), they **must** have different interpretations. For example, given the feature sets  $\text{FILE}_W$  where  $W$  is any white piece in the board and  $\text{FILE}_B$  where  $B$  is any black piece in the board, the feature set  $\text{FILE}_W \oplus \text{FILE}_B$  will have the basic elements  $\{a, b, \dots, h\}$  for both white and black pieces, but each with a different interpretation. Note that the notation presented in this work is not standard.

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

### 3.2 Indexing

The input to the network is a one-dimensional vector, so we need a way to map the tuples (elements are trivial) 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 in a set  $S$  must correspond to a number between 0 and  $|S| - 1$ . For example, the feature set  $A \times B \times C$  has  $|A| \times |B| \times |C|$  elements, and the tuple  $(a, b, c)$  is mapped to the element indexed at  $a \times |B| \times |C| + b \times |C| + c$ . The same striding logic applies to feature sets built with the sum operator, recursively.

### 3.3 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:

$$\begin{aligned} \text{ALL} &= \text{SQUARE}_P \times \text{ROLE}_P \times \text{COLOR}_P \\ &\text{for every } P \text{ piece in the board} \end{aligned}$$

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  $(e4, \text{♖}, \text{♖})$  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  $64 * 6 * 2 = \mathbf{768 \text{ features}}$ , 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 each possible position of the king of the side to move, it captures all the pieces in the board (ALL):

$$\text{KING-ALL} = \text{SQUARE}_K \times \text{ALL}_P$$

where  $K$  is the king of the side to move and  
 $P$  is every piece in the board

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  $64 \times 768 = \mathbf{49152 \text{ features}}$ . 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 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: (a8..h8, ♖, ○) and (a1..h1, ♜, ●) 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.

## 4 Efficiently updatable neural networks

NNUE (Efficiently updatable neural network) is a neural network architecture that allows for very fast subsequent evaluations for minimal input changes. It was invented for Shogi by Yu Nasu in 2018 [9], later adapted to Chess for use in Stockfish in 2019 and may be used in other board games as well. Most of the information described in this chapter can be found in the excellent Stockfish NNUE documentation [16].

NNUE operates in 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 minimal. 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. [no accelerators, aggressive quantization techniques]

[tradeoff between speed and accuracy]

### 4.1 Layers

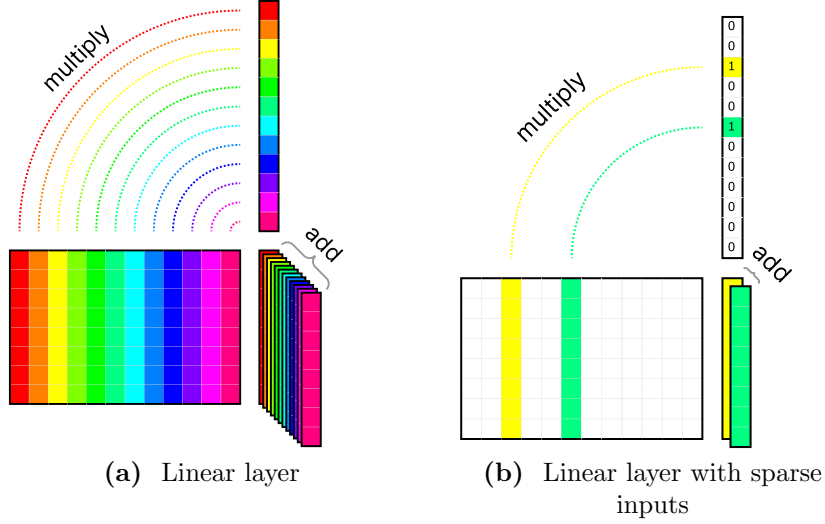
For this thesis, I have chosen to use a very simple NNUE architecture, which consist of two linear (fully connected) layers and clipped ReLU activations. In the literature, there are other architectures that make use of polling layers, sigmoid activations and others, but since this work is about experimenting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

**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  $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ , where:

1.  $\mathbf{x}$  the input column vector of shape **in\_features**.
2.  $\mathbf{W}$  the weight matrix of shape (**out\_features**, **in\_features**).
3.  $\mathbf{b}$  the bias column vector of shape **out\_features**.

4.  $\mathbf{y}$  the output column vector of shape `out_features`.

The operation  $\mathbf{W}\mathbf{x}$  can be simplified to “if  $\mathbf{x}_i$  is not zero, take the column  $\mathbf{A}_i$ , multiply it by  $\mathbf{x}_i$  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 2.



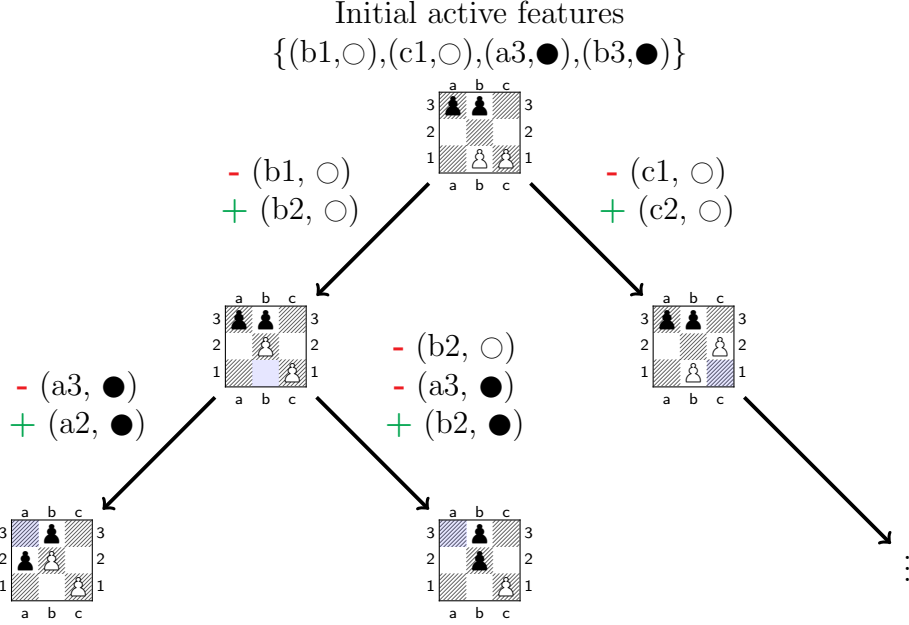
**Figure 2.** Linear layer operation comparison. Figures from [16].

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.

**Clipped ReLU** This is a simple activation that clips the output in the range  $[0, 1]$ . The operation is  $\mathbf{y} = \min(\max(\mathbf{x}, 0), 1)$ . The output of this activation function is the input for the next layer, and because of the aggressive 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), meaning that the amount of features becoming active or inactive is minimal as well. This is depicted in Figure 3.



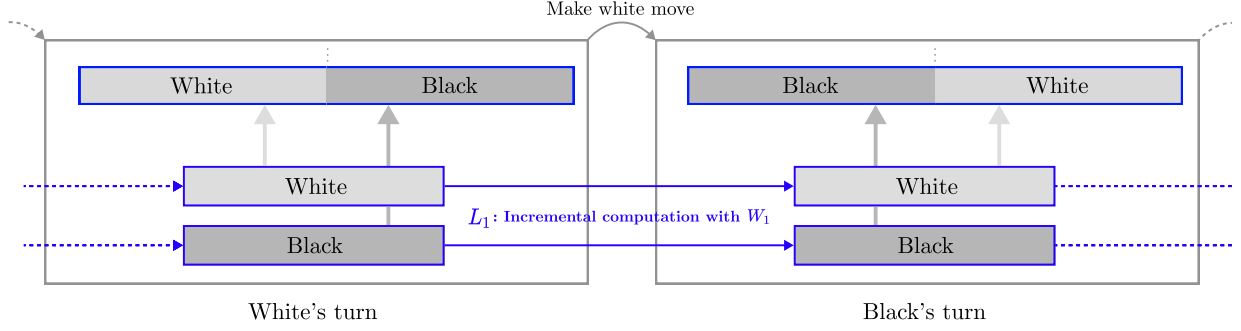
**Figure 3.** Partial tree of feature updates (removals and additions) for  $\text{SQUARE}_P \times \text{COLOR}_P$  (white's point of view) in a simplified  $3 \times 3$  pawn-only board.

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 layer (usually small) has to be computed.

Recall that the way I defined feature sets, they always encode the position from 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 accumulators to take advantage of this. Figure 4 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 4.** Concatenation of the first layer's output after a move is made. Inspired in a CPW figure.

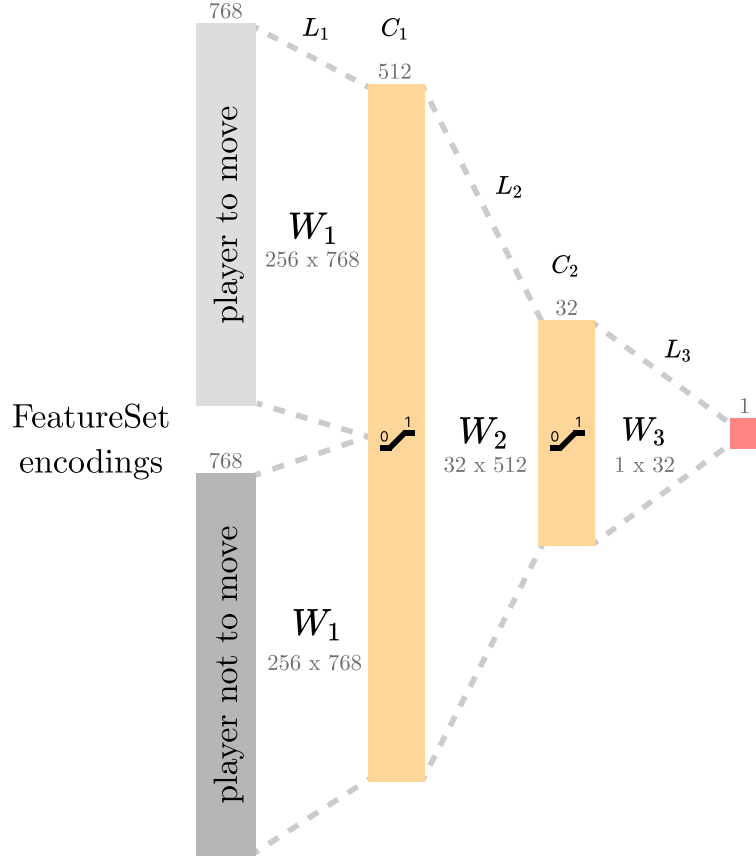
### 4.3 Network

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. [hablar de que no es la unica alternativa?] 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:

$N$ : number of features in the feature set

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



**Figure 5.** 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 outweighs 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

[Quizas hay que explicar mas esta parte, es compleja]

In this thesis, I will use the same quantization scheme used in the engine Stockfish [16], due its simplicity and it has been battle tested. It uses `int8`  $[-128, 127]$  for inputs and weights, and `int16`  $[-32768, 32767]$  where `int8` is not possible. 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:  $s_a \mathbf{x}$ . We want the output to also be scaled to the activation range so it can be passed to the next:  $s_a \mathbf{y}$ .

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):  $s_W \mathbf{W}$ . 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:

$$s_a s_W \mathbf{y} = (s_W \mathbf{W})(s_a \mathbf{x}) + s_a s_W \mathbf{b} \quad (1)$$

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

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 ( $s_a \mathbf{y}$ ), 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:

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

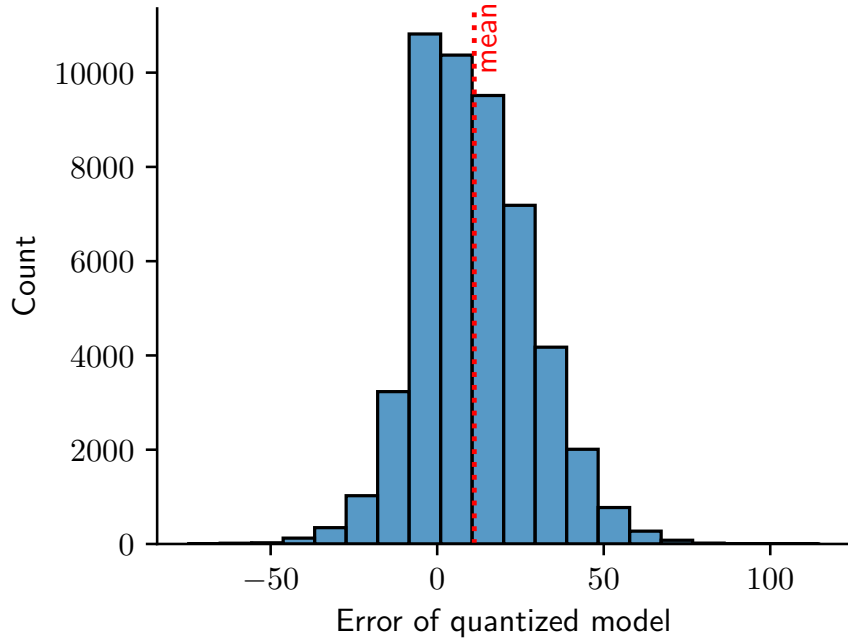
[seguir escribiendo y arreglar cosas arriba]

## 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 measured the error introduced by running a trained model in thousands of positions. The error is computed as the difference between the output of the quantized model (in Rust) and the float model (in Pytorch).



**Figure 6.** Error of the quantized model compared to the float model. N=50000

[rever porque esta corrido] In Figure 6 we can see that the error is centered around zero and most of the errors are within 50 units, which is acceptable given that the model output values much higher.

## 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 describe the two methods to train the networks, each with its own loss function and training dataset. [mas?]

### 5.1 Source dataset

Data is needed to train the network. The proposal for the thesis was to use the Lichess database [8], 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 [17], 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 on multiple opening books. Later stages 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 format to store samples yet very complex to decode. Fortunately, Stockfish provides a tool to export this data into a text format. 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.5. The resulting file was 2.59TB in size and contained **48.4 billion samples** with the format:

FEN <sup>1</sup> , Score , 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 exploits 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:

FEN , Score , Best move	( , Actual move , Score , Best move ) * <sup>2</sup>
-------------------------	--

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.

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.

---

<sup>1</sup>Standard notation to describe positions of a chess game. It is a sequence of ASCII characters.

<sup>2</sup>Repeated zero or more times.

## 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

Scores are values ranging from -10000 to 10000. [hablar de que score vs performance ajusta a una sigmoid] [hablar de que se necesita una escala]

### 5.2.2 Loss function

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

$$L(y, f(x, \mathbf{W})) = \left| \sigma \left( \frac{y}{\text{scale}_{\text{in}}} \right) - \sigma \left( \frac{f(x, \mathbf{W})}{\text{scale}_{\text{out}}} \right) \right|^{2.6}$$

where...

1.  $y$  is the target score.
2.  $f$  is the model.
3.  $\mathbf{W}$  are the parameters of the model.
4.  $x$  is the input (encoded feature set).
5.  $\sigma$  is the sigmoid function.
6.  $\text{scale}_{\text{in}}$ ,  $\text{scale}_{\text{out}}$  are the scaling factors of the scores.

## 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 dataset are optimal. In the blog they used human moves from the Lichess database [8], so they rely in that humans make optimal 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 position in succession  $p \rightarrow q$  observed in the game, we will have  $f(p) = -f(q)$ .

2. Going from  $p$ , not to  $q$ , 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 and inequality can be built.

### 5.3.1 Loss function

The loss function is the 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  $f(p) = -f(q)$ . The loss function is given by

$$\begin{aligned} L(x_p, x_q, x_r, \mathbf{W}) = & -\log(\sigma(q - r)) \\ & -\log(\sigma(p + q)) \\ & -\log(\sigma(-(p + q))) \end{aligned}$$

where...

1.  $p = \frac{f(x_p, \mathbf{W})}{\text{scale}_{\text{out}}}$ ,  $q = \frac{f(x_q, \mathbf{W})}{\text{scale}_{\text{out}}}$ ,  $r = \frac{f(x_r, \mathbf{W})}{\text{scale}_{\text{out}}}$
2.  $f$  is the model.
3.  $\mathbf{W}$  are the parameters of the model.
4.  $x_i$  is the input (encoded feature set) for the  $i \in \{p, q, r\}$  position.
5.  $\sigma$  is the sigmoid function.
6.  $\text{scale}_{\text{out}}$  are the scaling factors of the scores.

It is important to note that quantization is happening in this method too, so the output of the model must be scaled appropriately.

[siento que falta explicar algo aca]

## 5.4 Setup

[a esta seccion le falta banda]  
[multithreaded, perf]

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 eval (short for score **evaluations**), it is the size of the feature set times two plus 1 for the target score.

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 **samples-service** that read the training data files and generates training batches for the specified feature set in a shared memory buffer. The Python script copies the data from the buffer at the start of each iteration, allowing Rust to generate the next batch (in the CPU) while Pytorch is training the current one (in the GPU). To coordinate the memory access between the two processes, a single byte is sent using standard I/O.

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.

[wandb? evaluation?]

## 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 computed evaluations or PQR triplets. This determinates the format of the samples as well as the loss function. All experiments will train using computed evaluations, unless specified. Explained in detail in chapter 5.
- **Training hyperparameters:** The usual machine learning hyperparameters for training, such as batch size, learning rate and scheduler. Recall that each epoch is 100 million positions, and the training will last for 256 epochs (25.6 billion positions).

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 outweigh 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 differences in this metric are 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 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 measure an engine’s strength.
- **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.

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.

## 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 the variable, 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**. These values showed acceptable results during early stages of development and will remain fixed for all runs.

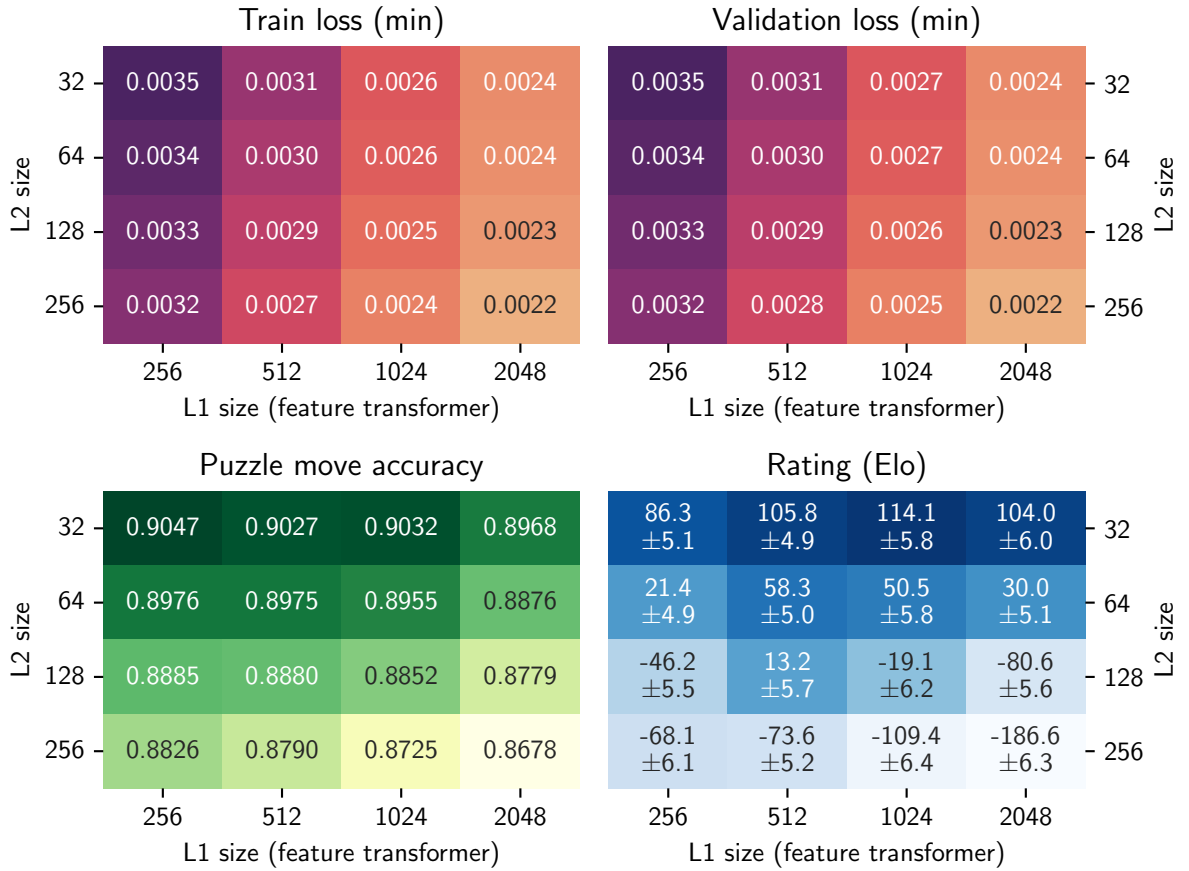
It remains to find a good network architecture. Bigger networks may have lower loss and predict better, but they will also have slower inferences. This is the tradeoff between inference time and node visits (more depth), which are also affected by the quality of the prediction due better pruning. So 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 7, 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 be fast. So, instead of fiddling further with SIMD I decided to **keep L2 at 32**.



**Figure 7.** Network architecture sweep results (L1  $\times$  L2).

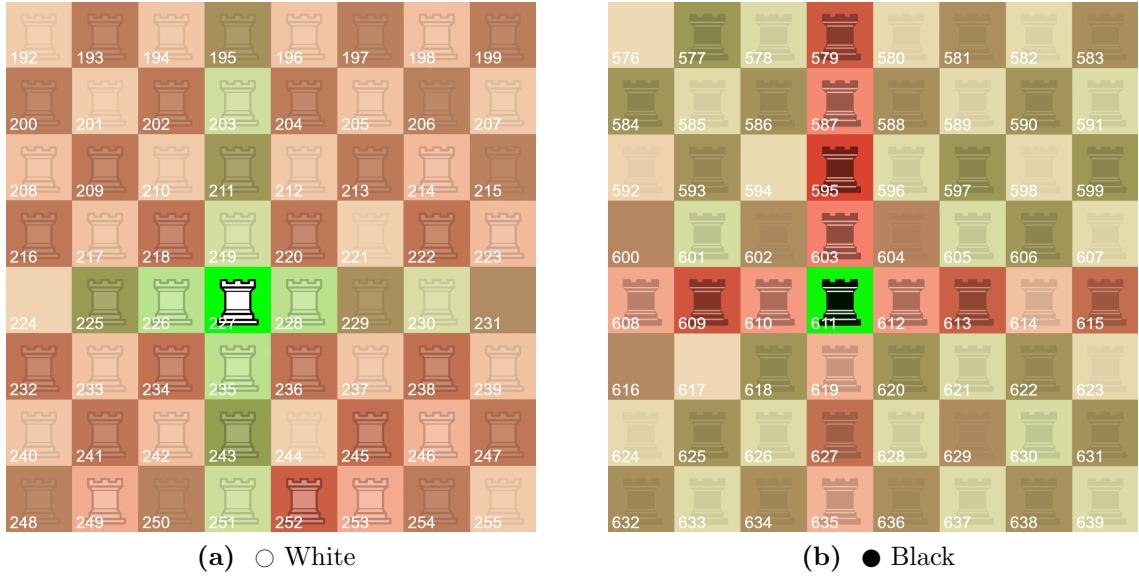
Ratings computed using  $N \approx 11000$  games per model. Table in Appendix A.2.

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 worse. 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 encodings

**Motivation.** Looking back at the networks generated by ALL in baseline runs, the learned weights 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 8, which depicts the SQUARE part of the features where the role is ♖ Rook.



**Figure 8.** 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).

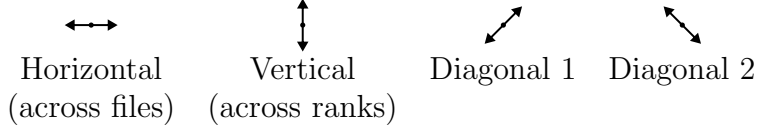
This particular neuron learned to recognize the presence of a ♖ Rook, affected by the pattern of another potential rook in the same file or rank (other pieces may be involved but I am focusing on rooks for the example). Doing so, it had to relate one feature for every potential square where a rook could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

More examples of this behaviour can be found in Appendix A.3.1, 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:



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.** Axes feature blocks

Depiction	Block name	Definition for every piece $P$ in the board		Number of features
	H	$\text{FILE}_P$	$\times \text{ROLE}_P \times \text{COLOR}_P$	96
	V	$\text{RANK}_P$	$\times \text{ROLE}_P \times \text{COLOR}_P$	96
	D1	$\text{DIAG1}_P$	$\times \text{ROLE}_P \times \text{COLOR}_P$	180
	D2	$\text{DIAG2}_P$	$\times \text{ROLE}_P \times \text{COLOR}_P$	180

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 (here treated as a block). 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 encodings feature sets

Depiction	Feature set	Number of features
$\longleftrightarrow \oplus \updownarrow$	$H \oplus V$	192
$\nearrow \oplus \searrow$	$D1 \oplus D2$	360
$\longleftrightarrow \oplus \updownarrow \oplus \nearrow \oplus \searrow$	$H \oplus V \oplus D1 \oplus D2$	552
$ALL \oplus \longleftrightarrow \oplus \updownarrow$	$ALL \oplus H \oplus V$	960
$ALL \oplus \nearrow \oplus \searrow$	$ALL \oplus D1 \oplus D2$	1128
$ALL \oplus \longleftrightarrow \oplus \updownarrow \oplus \nearrow \oplus \searrow$	$ALL \oplus H \oplus V \oplus D1 \oplus D2$	1320

I expect that the feature sets that are sums of single axes ( $\longleftrightarrow \oplus \updownarrow$ ,  $\nearrow \oplus \searrow$  and  $\longleftrightarrow \oplus \updownarrow \oplus \nearrow \oplus \searrow$ ) 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 ( $ALL \oplus \dots$ ) should perform better than without, providing that the idea explained in the motivation holds.



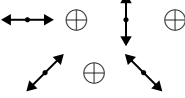








For each of the proposed feature sets, I will train a network and evaluate its performance relative to each other using a tournament. I expect to see them ranked in the reverse order as presented in the table (more extra axes better).

**Results.** The results in table 3 show that indeed, adding the axes blocks make the network validation loss slightly lower, from 0.00316 in ALL to 0.00307 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  $172.5 \pm 4.8$  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 474 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 encodings results

Feature set	Number of features	Val. loss <i>min</i>	Rating <i>elo (rel. to ALL)</i>	Puzzles <i>move acc.</i>
	192	0.00581	$-368.1 \pm 5.7$	0.8618
	360	0.00670	$-474.5 \pm 6.4$	0.8517
	552	0.00389	$-172.5 \pm 4.8$	0.8748
ALL (reference)	768	0.00316	<b>0.0</b>	0.8865
ALL $\oplus$  $\oplus$ 	960	0.00308	$-4.6 \pm 5.0$	0.8851
ALL $\oplus$  $\oplus$ 	1128	0.00309	$-33.3 \pm 5.0$	0.8814
ALL $\oplus$  $\oplus$  $\oplus$  $\oplus$ 	1320	<b>0.00307</b>	$-57.7 \pm 4.8$	0.8766

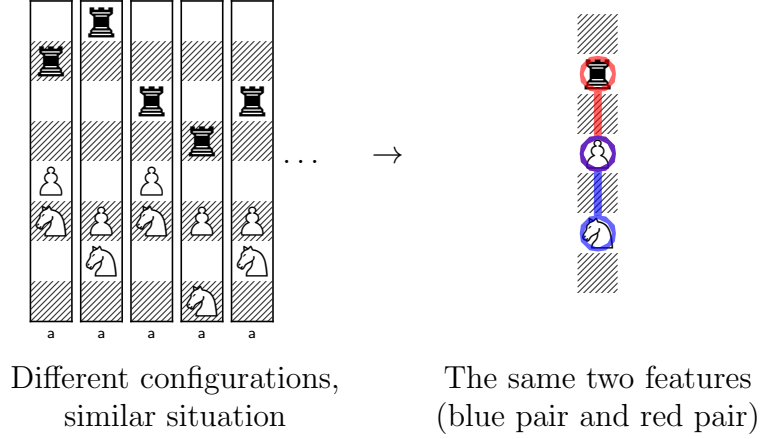
The next experiment will focus on adding more specific features, instead of more broad ones.

### 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:





There are many configurations for the three pieces and the idea is to collapse all of these into two features: the pair of pieces ( $\text{♖}\bullet$ ,  $\text{♙}\circ$ ) and the pair of pieces ( $\text{♙}\circ$ ,  $\text{♘}\circ$ ). This way, the network can learn that the  $\text{♖}$  Rook can capture the  $\text{♙}$  Pawn, and that the  $\text{♘}$  Knight is protected behind the  $\text{♙}$  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  $\circ$  White  $\text{♖}$  Rook in the 4th rank”) the proposed features here are more specific: “there is a  $\bullet$  Black  $\text{♖}$  Rook next to a  $\circ$  White  $\text{♙}$  Pawn in the ‘a’ file”.

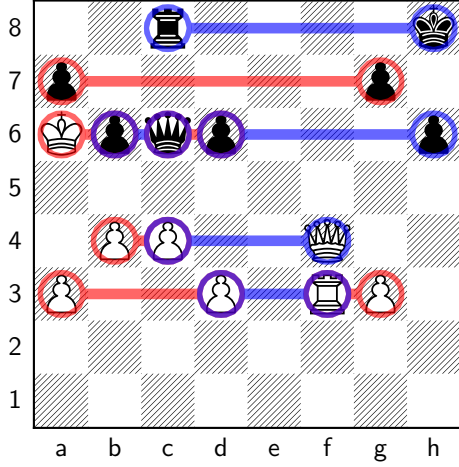
**Experiment.** I developed two feature blocks: for the horizontal and vertical axis. The blocks are defined in table 4:

**Table 4.** Pairwise feature blocks

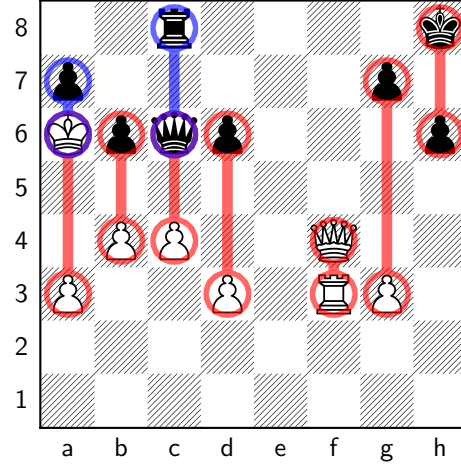
Depiction	Block name	Definition	Number of features
$\circ-\circ$	PH	$\text{RANK}_P \times (\text{ROLE}_P \times \text{COLOR}_P) \times (\text{ROLE}_Q \times \text{COLOR}_Q)$ for every $P, Q$ adjacent pieces in the same rank, such that $\text{file}(P) < \text{file}(Q)$	1152
$\circ$ $\circ$	PV	$\text{FILE}_P \times (\text{ROLE}_P \times \text{COLOR}_P) \times (\text{ROLE}_Q \times \text{COLOR}_Q)$ for every $P, Q$ adjacent pieces in the same file, such that $\text{rank}(P) < \text{rank}(Q)$	1152

Note that it is important to consider the order of the pieces in the pair, so that features are not mirrored (we want to differentiate between both). In code this is taking care of just by iterating over the pieces and building the pair in the same order every time.

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



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  $ALL \oplus PH$  (1920 features),  $ALL \oplus PV$  (1920 features) and  $ALL \oplus PH \oplus PV$  (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 contract the loss in performance due to the big increase in the number of features and more complex updates.

**Results.** The results in table 5 show that...

**Table 5.** Pairwise encodings results

Feature set	Number of features	Val. loss <i>min</i>	Rating <i>elo (rel. to ALL)</i>
ALL (reference)	768	0.00316	<b>0.0</b>
$ALL \oplus \text{PH}$	1920	0.00296	$-0.5 \pm 5.1$
$ALL \oplus \text{PV}$	1920	0.00303	$-14.1 \pm 4.5$
$ALL \oplus \text{PH} \oplus \text{PV}$	3072	<b>0.00287</b>	$-32.8 \pm 4.7$

I did not bother implementing diagonal pairs ( $\text{PH} \oplus \text{PV}$  and  $\text{PV} \oplus \text{PH}$ ) 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.

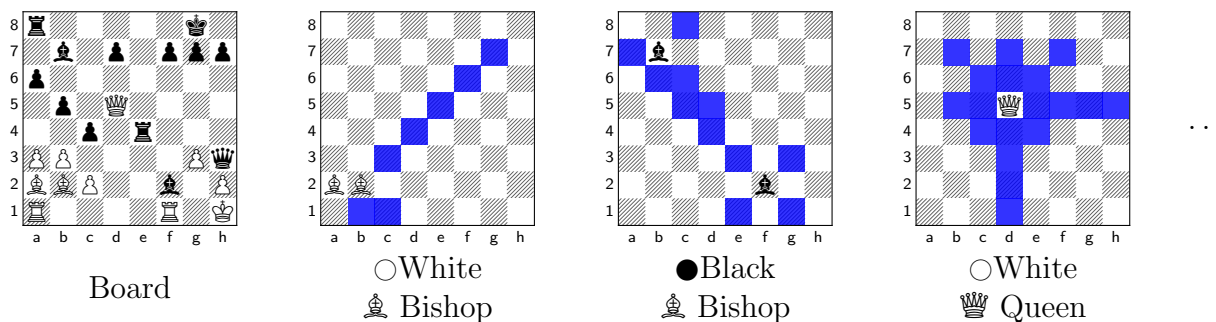
---

## 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 [14] 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.

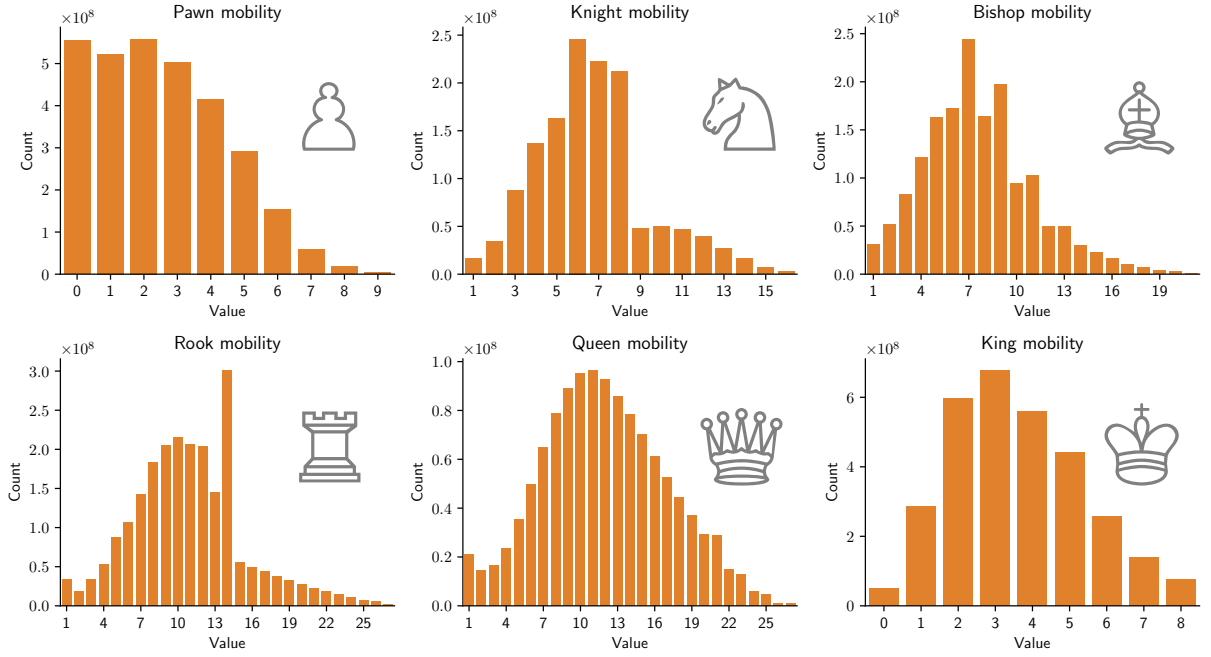
**Experiment.** There are two ways to go about encoding mobility:





- **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.



- **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, with a lot less of row 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 9. 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 9.** 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	$\text{SQUARE}_P \times (\text{ROLE}_Q \times \text{COLOR}_Q)$ for every square $P$ and piece $Q$ such that $Q$ can move to $P$	768
MC	$M(R)_{R,C} \times (\text{ROLE}_R \times \text{COLOR}_C)$ for every role $R$ and color $C$ where $M(R) = \{0 \dots\}$ are the mobility values for the role $R$	206

decir los fs  
asdasdasd

**Results.** asdadasdasd

## 6.5 Attacks / Threats

as bitsets per piece type number of attacks

## 6.6 mas supongo?


## 6.7 Symmetry? / Relativity?

**Motivation.**

BUCKETING

Medir el impacto de agregar simetría al fs. Red mas chica, inf mas rapida, mejor perf?  
probar simetria, eventualmente probar con el mejor feature set de arriba, a ver si mejora  
poniendo a cada bloque individual simetria

HALF-RELATIVE(H—V—HV)KING-PIECE?

inspired by KP, build features relative to the position of the  King

## 6.8 Statistical features?

Define K-ALL-ALL

KING-ALL is a subset of ALL-ALL.

Top P

Hacer un subset de AA (589824).

- Destilar?
- Probar si es lo mismo quedarse con el TOP K de las mas comunes o con las que dice el performance.
- Catboost? PCA?

## 6.9 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 [8] 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.

**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.

asdasdasd

**Results.** aqui pondria los resultados... si los tuviera!

## 7 Final words

### 7.1 Conclusions

maybe implementing a custom engine was not a good idea. bugs and stuff  
feature sets: havent changed much in a long time. thats why it is so hard find anything better

feature engineering slow with this kind of task, iteration times are extremely slow. This is why fishnet exists.

### 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 source, the filtering, .....

**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:** more fs, pruning de features al pepe para evitar updates

## References

- [1] Erik Bernhardsson. *Deep learning for... chess*. 2014. URL: <https://erikbern.com/2014/11/29/deep-learning-for-chess.html>.
- [2] 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.
- [3] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. “Deep Blue”. In: *Artificial Intelligence* (2002). DOI: 10.1016/S0004-3702(01)00129-1.
- [4] 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>.
- [5] *Chess Programming Wiki*. URL: <https://www.chessprogramming.org>.
- [6] 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.
- [7] 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.
- [8] *Lichess Database*. URL: <https://database.lichess.org>.
- [9] 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](https://www.apply.computer-shogi.org/wcsc28/appeal/the_end_of_genesis_T.N.K.evolution_turbo_type_D/nnue.pdf).
- [10] 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>.
- [11] 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.
- [12] 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.
- [13] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* (2017). DOI: 10.1038/nature24270.
- [14] Eliot Slater. “Statistics for the Chess Computer and the Factor of Mobility”. In: *Proceedings of the Symposium on Information Theory, London* (1950).
- [15] *Stockfish*. URL: <https://stockfishchess.org>.



- [16] Official Stockfish. *nnue.md*. URL: <https://github.com/official-stockfish/nnue-pytorch/blob/master/docs/nnue.md>.
- [17] *Stockfish NNUE training data*. URL: <https://robotmoon.com/nnue-training-data>.
- [18] 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

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

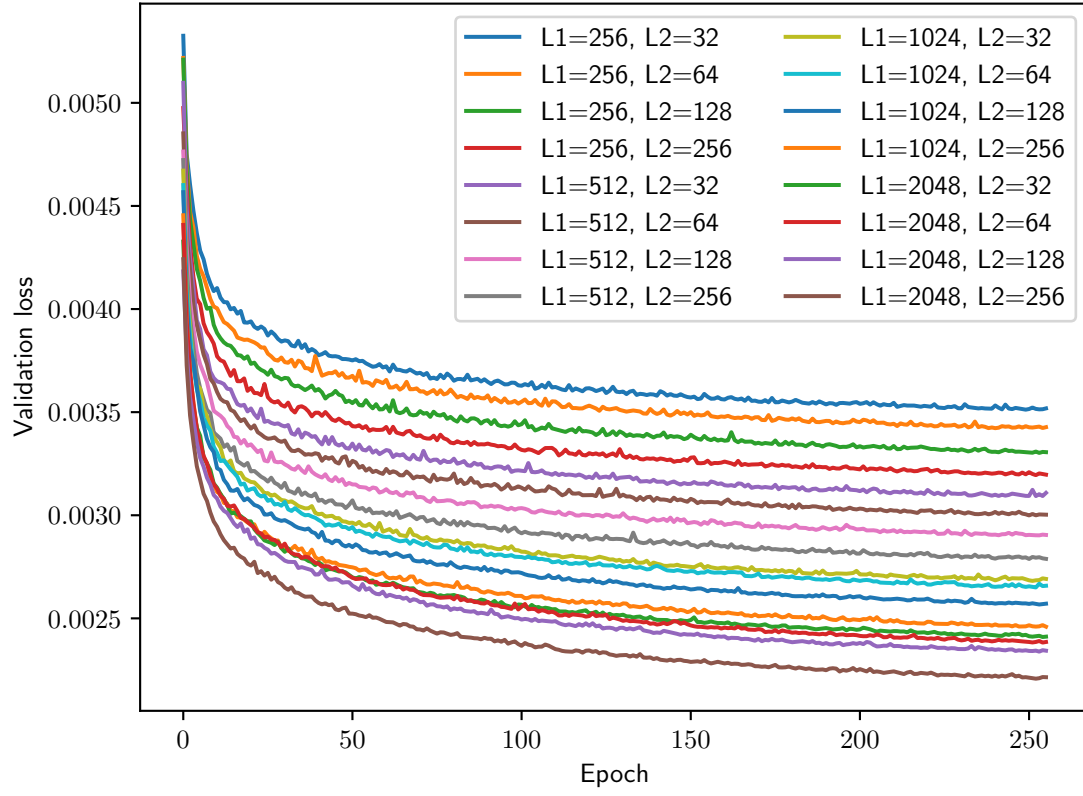
### A.1 Dataset

asdasd

### A.2 Baseline runs

**Table 7.** Network architecture sweep results (baseline)  
(Games in tournament  $\approx 40000$  each)

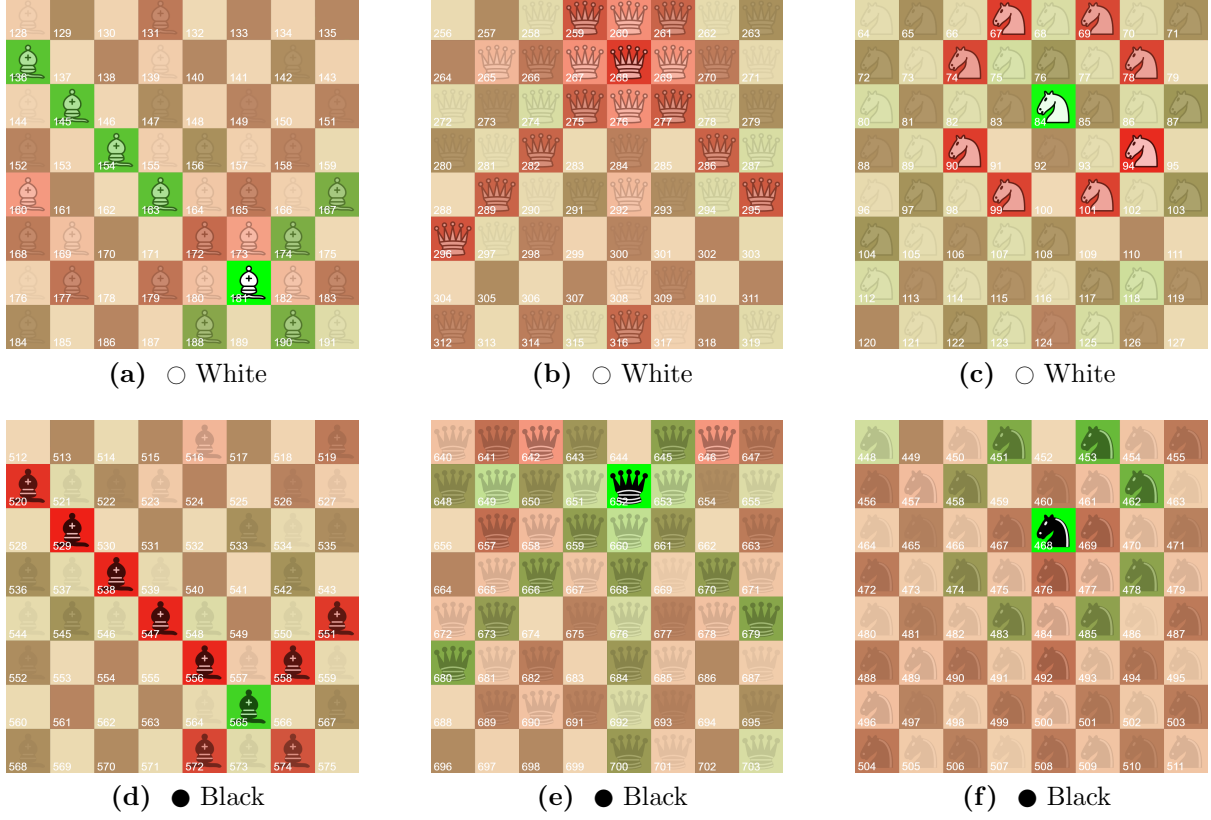
Feature set	Train hyperparams			Network		Val. loss <i>min</i>	Rating <i>elo (avg=0)</i>	Puzzles <i>move acc.</i>	Runtime <i>hh:mm:ss</i>
	Batch	LR	Gamma	L1	L2				
ALL	16384	5e-04	0.99	256	32	0.00351	$86.3 \pm 5.1$	<b>0.9047</b>	1:53:59
ALL	16384	5e-04	0.99	256	64	0.00342	$21.4 \pm 4.9$	0.8976	1:54:56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



**Figure 10.** Network architecture sweep validation loss over epochs (baseline)

## A.3 Axes encoding

### A.3.1 Examples



**Figure 11.** 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.3.2 Runs

**Table 8.** Axes feature sets sweep results  
(Games in tournament  $\approx 33000$  each)

Feature set	Train hyperparams			Network		Val. loss <i>min</i>	Rating <i>elo (rel. to ALL)</i>	Runtime <i>hh:mm:ss</i>
	Batch	LR	Gamma	L1	L2			
ALL	16384	5e-04	0.99	512	32	0.00316	<b>0.0 <math>\pm</math> 0.0</b>	1:50:55
ALL + H + V	16384	5e-04	0.99	512	32	0.00308	-4.6 $\pm$ 5.0	2:13:23
ALL + D1 + D2	16384	5e-04	0.99	512	32	0.00309	-33.3 $\pm$ 5.0	2:26:23
ALL + H + V + D1 + D2	16384	5e-04	0.99	512	32	<b>0.00307</b>	-57.7 $\pm$ 4.8	2:50:04
H + V + D1 + D2	16384	5e-04	0.99	512	32	0.00389	-172.5 $\pm$ 4.8	2:27:34
H + V	16384	5e-04	0.99	512	32	0.00581	-368.1 $\pm$ 5.7	1:51:13
D1 + D2	16384	5e-04	0.99	512	32	0.00670	-474.5 $\pm$ 6.4	1:58:50

### A.4 Pairwise axes runs

**Table 9.** Pairwise feature sets sweep results  
(Games in tournament  $\approx 24000$  each)

Feature set	Train hyperparams			Network		Val. loss <i>min</i>	Rating <i>elo (rel. to ALL)</i>	Runtime <i>hh:mm:ss</i>
	Batch	LR	Gamma	L1	L2			
ALL + PH	16384	5e-04	0.99	512	32	0.00296	<b>-0.5 <math>\pm</math> 5.1</b>	2:26:00
ALL + PV	16384	5e-04	0.99	512	32	0.00303	-14.1 $\pm$ 4.5	2:20:57
ALL + PH + PV	16384	5e-04	0.99	512	32	<b>0.00287</b>	-32.8 $\pm$ 4.7	3:32:35

## A.5 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';
}
```