DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Feature set analysis for chess NNUE networks

September 20, 2024

Martín Emiliano Lombardo
mlombardo9@gmail.com

Directores

Agustín Sansone   Diego Fernández Slezak
agustinsansone7@gmail.com   dfslezak@dc.uba.ar

Abstract

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

Agradecimientos

:)

Contents

1 Introduction

[traducir y agregar mas, es lo de la proposal]

El desarrollo de engines de ajedrez es y ha sido un tema de interés en la comunidad de ajedrez y computación desde hace décadas. IBM DeepBlue [3] fue la primera máquina de ajedrez en alcanzar nivel sobre-humano ganándole a un campeón mundial —Garry Kaspárov— de manera consistente en 1997. A partir de entonces, los engines han evolucionado en fuerza y complejidad.

Los engines tienen dos componentes principales: la búsqueda y la evaluación. La búsqueda es el proceso de explorar el árbol de posibles jugadas. La evaluación determina qué tan buena son esas posiciones para el que juega. Desde el origen de ajedrez por computadora en los años 50 hasta hace unos años, todos los engines han utilizado los algoritmos de búsqueda en árboles Minimax [5], Monte Carlo Tree Search [2] (MCTS) o alguna de sus variantes [6, 15], con funciones de evaluación muy complejas y artesanales que se basan en conocimiento humano sobre el juego.

Hasta los 2010, el desarrollo de engines avanzaba a un paso lento pero consistente. Hasta que en 2017, Google DeepMind publicó AlphaGo Zero [11] y su sucesor AlphaZero [10, 9] (2018), que mostró ser contundentemente superior (28 victorias y 73 empates contra el mejor engine del momento). Introdujeron un nuevo enfoque para el desarrollo de engines de juegos de tablero: entrenar una red neuronal convolucional con un algoritmo de aprendizaje por refuerzo para que aprenda a jugar por sí misma.

Este cambio de paradigma, en donde la evaluación de las posiciones se realiza mediante redes neuronales en vez de funciones construidas con conocimiento humano, alteró el rumbo del desarrollo de todos los engines modernos (no sólo de Go y ajedrez). En 2018, Yu Nasu introdujo las redes neuronales "Efficiently Updatable Neural-Networks" [8] (ИИUӘ) para el juego Shogi. Las redes NNUE permiten evaluar posiciones similares con menos cómputo que si se lo hiciera de forma separada, lo que las hace ideales para ser utilizadas en engines con búsqueda de árbol. A partir de entonces, todos los engines modernos han incorporado redes NNUE o alguna especie de red neuronal a su evaluación.

El motor de ajedrez Stockfish, uno de los más fuertes del mundo, ha incorporado redes NNUE mezclado con evaluación clásica en la versión 12[1] (2020). A partir de Stockfish 16.1[2] (2024) la evaluación se realiza exclusivamente mediante redes NNUE, eliminando todo el aspecto humano.

1.1 Chess Engines

assad

---

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

1.2 Thesis plan

The aim of this thesis is to explore different kinds of board encodings (feature sets) in chess engines. 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 [13]. 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 want 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) [4] 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 the 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 with any other relevant information that may affect the outcome, 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. The value is defined from the perspective of the player to move.

Minimax trees [5] ...

The engine actually implements the negamax algorithm, which...

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).

fixed depth in iterative deepening

explicar que es importante esto porque la red aprende sobre posiciones "quietas"

2.3 Optimizations

A few more minor optimizations were made...

The whole search algorithm is embedded in a loop (it deep)

2.4 Implementation details

[reword esto]

The bot is implemented in the Rust programming language.

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 UCI protocol was also added using a library built on top of it.

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.

3 Feature set (board encoding)

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 built by a cartesian product of smaller sets of features (often called feature blocks), where each set extracts a different aspect of a position. Each tuple in the feature set corresponds to an element in the input vector, which will be set to 1 if the aspects captured by the tuple is present in the position, and 0 otherwise. If a tuple is present in a position, we say that the tuple is *active*.

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

$$\text{FILE} = \{a, b, ..., h\}$$
$$\text{RANK} = \{1, 2, ..., 8\}$$
$$\text{SQUARE} = \{a1, a2, ..., h8\}$$


And the following encode information about the pieces:

$$\text{ROLE} = \{ \text{♙ Pawn}, \text{♘ Knight}, \text{♗ Bishop}, \text{♖ Rook}, \text{♕ Queen}, \text{♔ King}\}^1$$
$$\text{COLOR} = \{\bigcirc \text{ White}, \bullet \text{ Black}\}$$

Since each set has to capture some information 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.
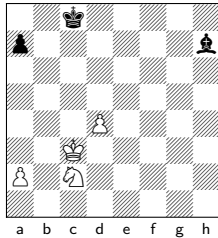
	Feature set	
	$\text{FILE}_P \times \text{COLOR}_P$	$\text{FILE}_P \times \text{ROLE}_P$
Active features	(a, $\bigcirc$), (a, $\bullet$), (c, $\bullet$), (c, $\bigcirc$), (d, $\bigcirc$), (h, $\bullet$)	(a, ♙), (c, ♔), (c, ♘), (d, ♙), (h, ♗)

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.

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 tuples of both sets A and B. These tuples do not interfere with each other, even if they have the same basic elements (e.g. h, 8, ♖, ●), they **must** have different interpretations. For example, given the feature sets $\textsc{File}_W$ where W is any white piece in the board and $\textsc{File}_B$ where B is any black piece in the board, the feature set $\textsc{File}_W \oplus \textsc{File}_B$ will have the basic elements $\{a, b, ..., h\}$ 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 in a feature set to the 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 $\textsc{All}$

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:

$$\textsc{All} = \textsc{Square}_P \times \textsc{Role}_P \times \textsc{Color}_P$$
$$\text{for every } P \text{ piece in the board}$$

The features are very simple to compute, and it has $64 * 6 * 2 = $ **768 features**, which makes it very small and efficient.

---

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

3.3.2 King-All

Another feature set built on top of ALL is the King-All feature set or called "KA" for short. For each possible position of the king of the side to move, it captures all the pieces in the board (ALL):

$$\text{King-All} = \text{Square}_K \times \text{All}_P$$
$$\text{where } K \text{ is the king of the side to move and}$$
$$P \text{ is every piece in the board}$$

This encoding allows the network to better understand 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 $64*768 = \mathbf{49152\ features}$. 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 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 (ƎUИИ 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 [8], 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 [13].

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, aggresive 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 $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, 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**.

The operation $\boldsymbol{Wx}$ 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 2.

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

Figure 2. Linear layer operation comparison. Figures from [13].

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 $\boldsymbol{y} = \mathbf{min}(\mathbf{max}(\boldsymbol{x}, \mathbf{0}), \mathbf{1})$. 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), 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 $SQUARE_P \times COLOR_P$ (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 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 accumualtors 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:

$\boldsymbol{N}$: 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 $\boldsymbol{P}$ 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 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 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

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

In this thesis, I will use the same quantization scheme used in the engine Stockfish [13], 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: $s_a \boldsymbol{x}$ 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:

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

[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).
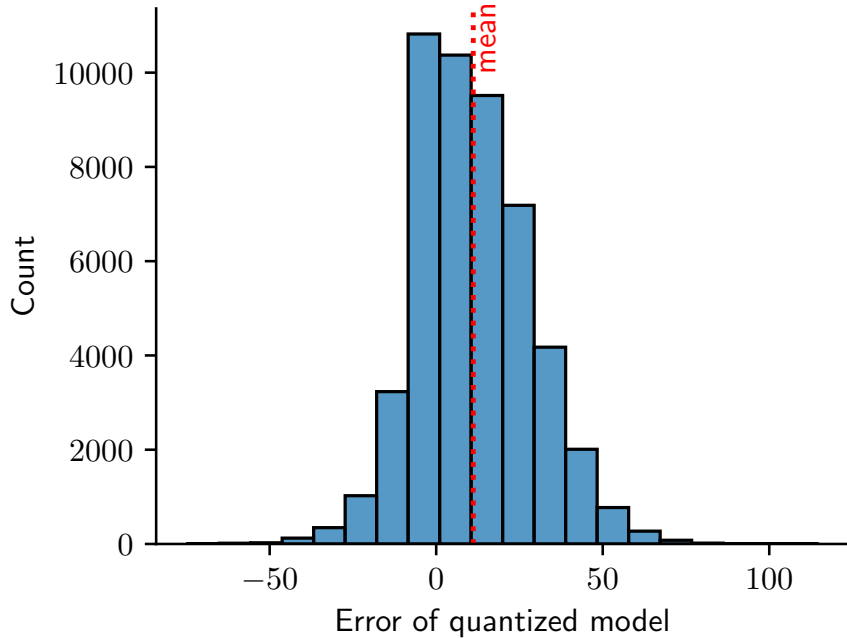

Figure 6.   Error of the quantized model compared to the float model. N=50000

[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 [7], 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 their networks [14], 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.3. The resulting file was 2.59TB in size and contained **48.4 billion samples** with the format:

$$\boxed{\text{FEN}^1 \quad , \quad \text{Score} \quad , \quad \text{Best move}}$$

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:

$$\boxed{\text{FEN} \quad , \quad \text{Score} \quad , \quad \text{Best move}} \left(\boxed{\quad , \quad \text{Actual move} \quad , \quad \text{Score} \quad , \quad \text{Best move}}\right)*^2$$

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.

---

[1] Standard notation to describe positions of a chess game. It is a sequence of ASCII characters.
[2] 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 errror with a power of 2.6 (the value used by the Stockfish's official trainer) given by

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

where. . .

1. y is the target score.
2. f is the model.
3. $\boldsymbol{W}$ are the parameters of the model.
4. x is the input (encoded feature set).
5. σ is the sigmoid function.
6. scale_{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, 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 $p \to r$, 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

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

where. . .

1. $p = \frac{f(x_p, \boldsymbol{W})}{\text{scale}_{\text{out}}}$, $q = \frac{f(x_q, \boldsymbol{W})}{\text{scale}_{\text{out}}}$, $r = \frac{f(x_r, \boldsymbol{W})}{\text{scale}_{\text{out}}}$

2. f is the model.

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

4. x_i is the input (encoded feature set) for the $i \in \{p, q, r\}$ position.

5. σ 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 **eval**uations), 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 usually last for 1024 epochs.

Once training is completed, the models will be evaluated depending on the experiment. To assess the performance 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 lower 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 contant. 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**. 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 better prunning. 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. 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.

Since I will be running a grid search, I will train the models for 256 epochs (25.6 billion positions) instead of my target of 1024 to make this preliminar process faster.

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 with SIMD I decided to **keep L2 at 32**.
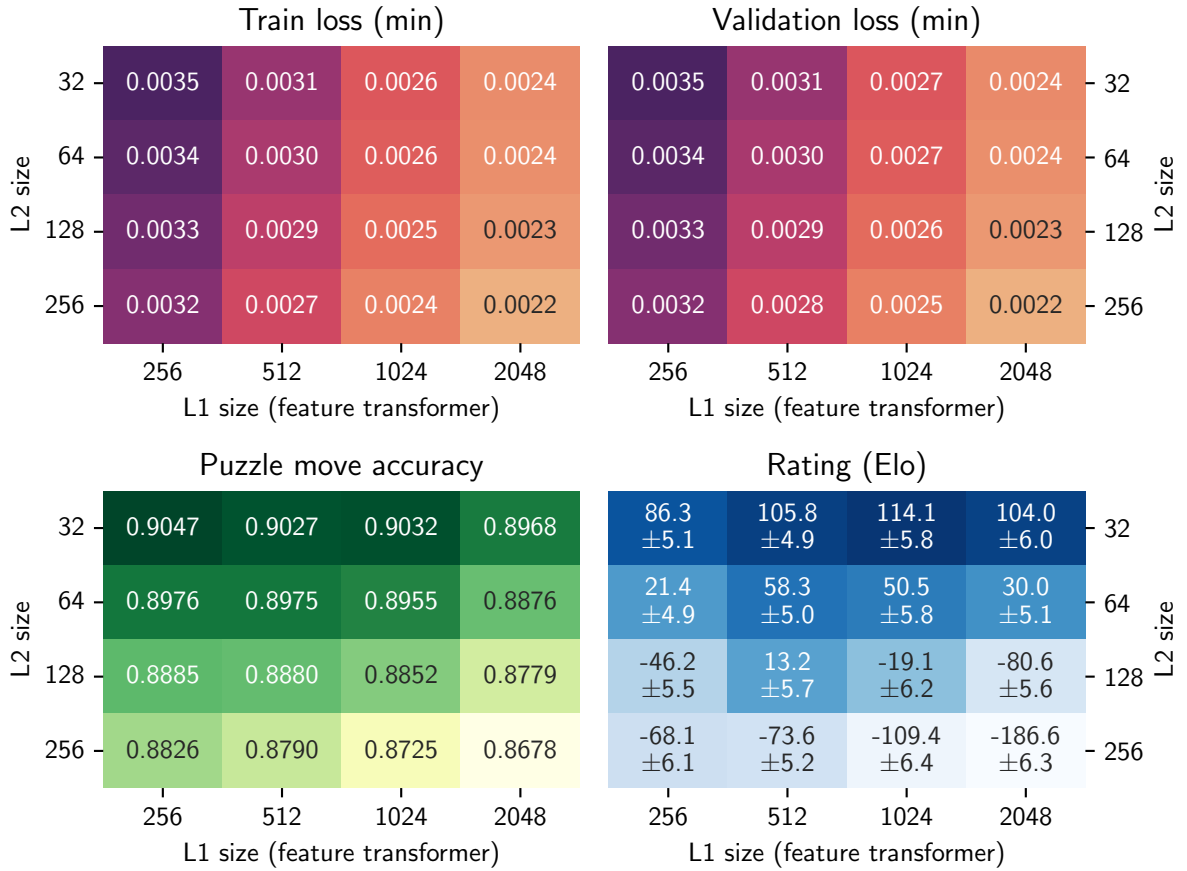
Figure 7. Network architecture sweep results (L1 × L2).
Ratings computed using $N \approx 11000$ games per model. Table in Appendix A.1.

If L2 is kept constant, the best L1 is not the smallest nor biggest.
[escribir elección de L1]
So, further experiments will use L1=512 and L2=32. 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 reference, Stockfish currently uses L1=2560, and employ (lots of) more tricks to make it even faster. 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 8, which depicts the SQUARE part of the features where the role is ♖ Rook.

(a) ○ White **(b)** ● Black
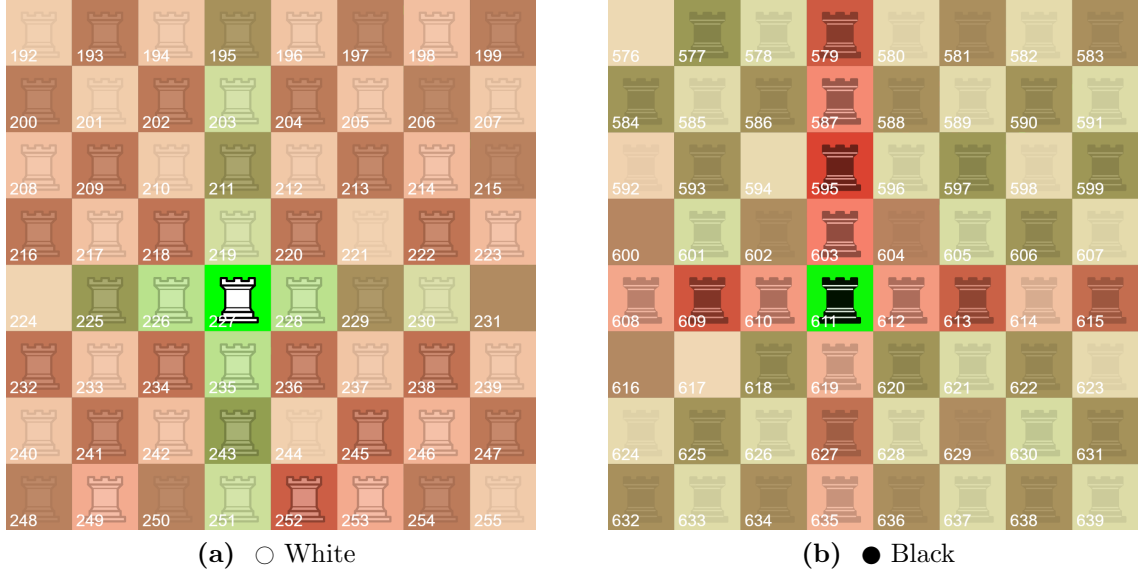
Figure 8. Weights of **a neuron** in the L1 layer, which are connected to features in ALL where the role is ♖ Rook. The intensity represents the weight value, and the color represents the sign (although not relevant).

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. 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, showcasing diagonal patterns and the ♘ Knight movements, although they do not move straight through axes.

Experiment. I will explore combinations of positional encodings for the pieces on the board, using the available axes. The canonical ALL feature set encodes each piece's position using the square it is located. Note that this is the same thing as encoding the position for a piece P as $\text{FILE}_P \times \text{RANK}_P$. So the position of each piece is determined

using the vertical (across ranks) and horizonal (across files) axes.

I will use the natural axes of a chess board:

↔   ↕   ↗   ↘

Horizontal   Vertical   Diagonal 1   Diagonal 2
(across files) (across ranks)
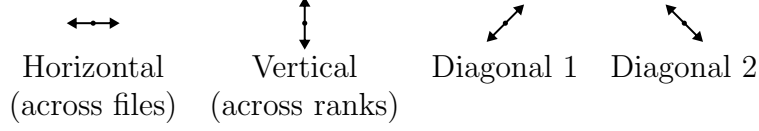
These axes coincide with the movement pattern of the pieces, which make them good candidates to encode the features I proposed. In table 1 I present the feature sets that I decided to try. The feature sets are named according to the axes they combine.

Table 1. Axis encoding feature sets[3]

Depiction	Feature set	Definition for every piece P in the board		# of features
↔ ⊕ ↕	H ⊕ V	$(\text{FILE}_P \oplus \text{RANK}_P)$	$\times\ \text{R}_P \times \text{C}_P$	192
↗ ⊕ ↘	D1 ⊕ D2	$(\text{DIAG1}_P \oplus \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	360
↔ ⊕ ↕ ⊕ ↗ ⊕ ↘	H ⊕ V ⊕ D1 ⊕ D2	$(\text{FILE}_P \oplus \text{RANK}_P \oplus \text{DIAG1}_P \oplus \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	552
✛	HV (PIECE)	$\text{FILE}_P \times \text{RANK}_P$	$\times\ \text{R}_P \times \text{C}_P$	768
✛ ⊕ ↔ ⊕ ↕	HV ⊕ H ⊕ V	$(\text{FILE}_P \times \text{RANK}_P \oplus \text{FILE}_P \oplus \text{RANK}_P)$	$\times\ \text{R}_P \times \text{C}_P$	960
✛ ⊕ ↗ ⊕ ↘	HV ⊕ D1 ⊕ D2	$(\text{FILE}_P \times \text{RANK}_P \oplus \text{DIAG1}_P \oplus \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	1128
✛ ⊕ ↔ ⊕ ↕ ⊕ ↗ ⊕ ↘	HV ⊕ H ⊕ V ⊕ D1 ⊕ D2	$(\text{FILE}_P \times \text{RANK}_P \oplus \text{FILE}_P \oplus \text{RANK}_P \oplus \text{DIAG1}_P \oplus \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	1320

Note: $\text{R}_P \times \text{C}_P$ expands to $\text{ROLE}_P \times \text{COLOR}_P$

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 there is a product of two axes (✛).

The feature sets that in addition to ✛ include lone axes (↔, ↕ , ↗ and ↘) should perform better than without, providing that the idea explained in the motivation holds.

---

[3]Note that one could build ✕, ↗↔, ↘↔, ↕↗ and ↕↘ but they are equivalent to ✛.

Note that even having twice the features, the penalty on inference performance should be minor due L1 being updated efficiently. [saco esto? ya se sabe...]

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. Aca poner los resultados

so........

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

ALL

6.3 Pairwise axes

Pepito o–o asdo ⚬ asd ⚬ asd ⚬ asd
[explicar...]
3 runs HV + PH HV + PV HV + PH + PV
el cosito de los pares

los resultados no son buenos...
Did not bother implementing diagonal pairs.

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 [12] 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.
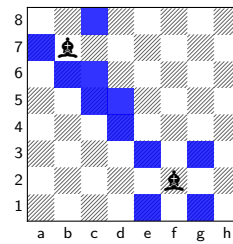
There are a couple of ways to go about encoding mobility:

- **Bitsets (per piece type):** the amount of features changed each turn may negate any gains.
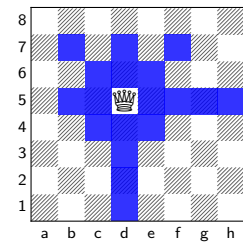
- **Counts (per piece type):**
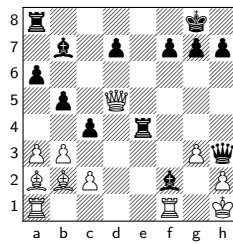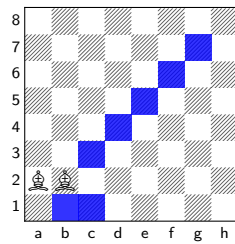
Board   ○White ♝ Bishop   ●Black ♝ Bishop   ○White ♛ Queen   ...

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

PQR. no va a ser tan bueno pero fun shit

6.10 Active neurons

medir si hay feature sets que no usen neuronas, que esto disparo el uso de HalfTopK
average number of features enabled by feature set (cantidad y porcentaje)
measure updates per move average and refreshes average per FS
[ESTO PONERLO EN EL APPENDIX]

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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

[15] 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 personal computer. They are listed here for reference.

A.1 Baseline runs

Table 2. Network architecture sweep results (baseline)

| Feature set | Train hyperparams | | | Network | | Val loss | Rating | Puzzles | Runtime |
	Batch	LR	Gamma	L1	L2	min	elo (avg=0)	move acc.	hh:mm:ss
HV	16384	5e-04	0.99	256	32	0.00351	86.3 ± 5.1	**0.9047**	1:53:59
HV	16384	5e-04	0.99	256	64	0.00342	21.4 ± 4.9	0.8976	1:54:56
HV	16384	5e-04	0.99	256	128	0.00330	-46.2 ± 5.5	0.8885	1:52:29
HV	16384	5e-04	0.99	256	256	0.00319	-68.1 ± 6.1	0.8826	2:29:26
HV	16384	5e-04	0.99	512	32	0.00309	105.8 ± 4.9	0.9027	1:54:28
HV	16384	5e-04	0.99	512	64	0.00300	58.3 ± 5.0	0.8975	1:53:44
HV	16384	5e-04	0.99	512	128	0.00290	13.2 ± 5.7	0.8880	1:51:06
HV	16384	5e-04	0.99	512	256	0.00279	-73.6 ± 5.2	0.8790	1:51:17
HV	16384	5e-04	0.99	1024	32	0.00268	**114.1 ± 5.8**	0.9032	2:15:18
HV	16384	5e-04	0.99	1024	64	0.00265	50.5 ± 5.8	0.8955	2:03:41
HV	16384	5e-04	0.99	1024	128	0.00257	-19.1 ± 6.2	0.8852	2:06:39
HV	16384	5e-04	0.99	1024	256	0.00246	-109.4 ± 6.4	0.8725	2:32:47
HV	16384	5e-04	0.99	2048	32	0.00241	104.0 ± 6.0	0.8968	3:11:56
HV	16384	5e-04	0.99	2048	64	0.00238	30.0 ± 5.1	0.8876	3:12:46
HV	16384	5e-04	0.99	2048	128	0.00234	-80.6 ± 5.6	0.8779	3:29:07
HV	16384	5e-04	0.99	2048	256	**0.00221**	-186.6 ± 6.3	0.8678	3:27:47
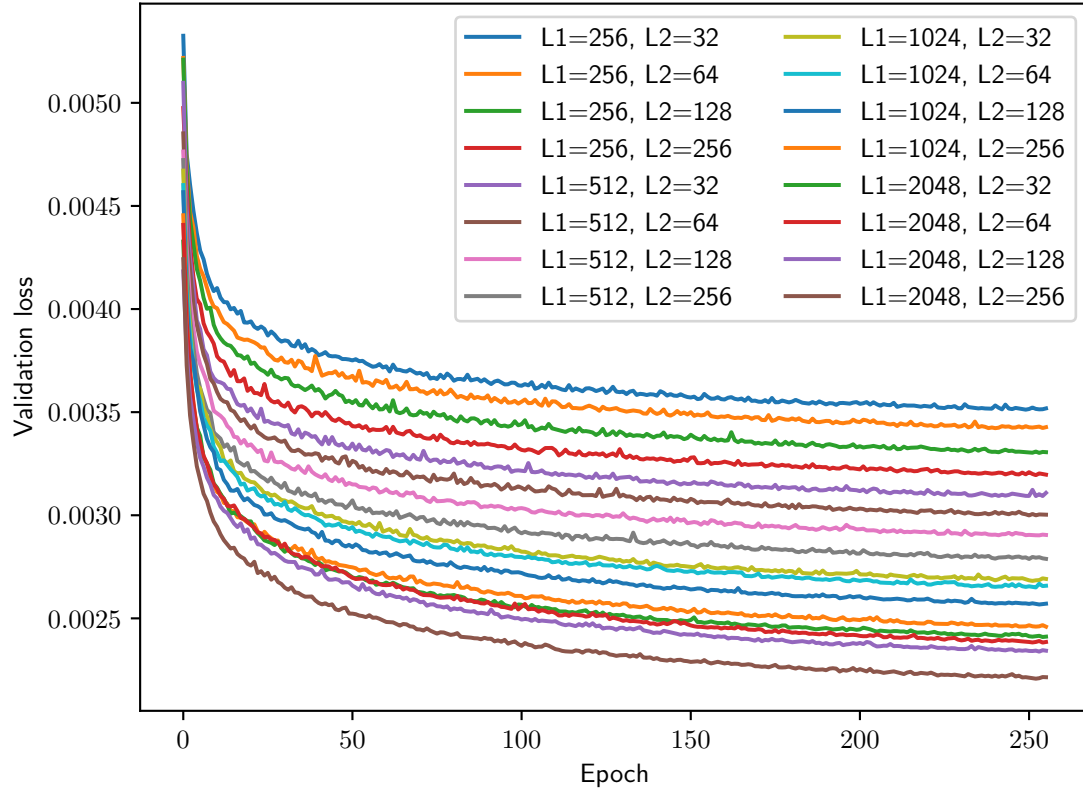
Figure 9. Network architecture sweep validation loss over epochs (baseline)

A.2 Axis encoding examples

(a) ○ White

(b) ○ White

(c) ○ White
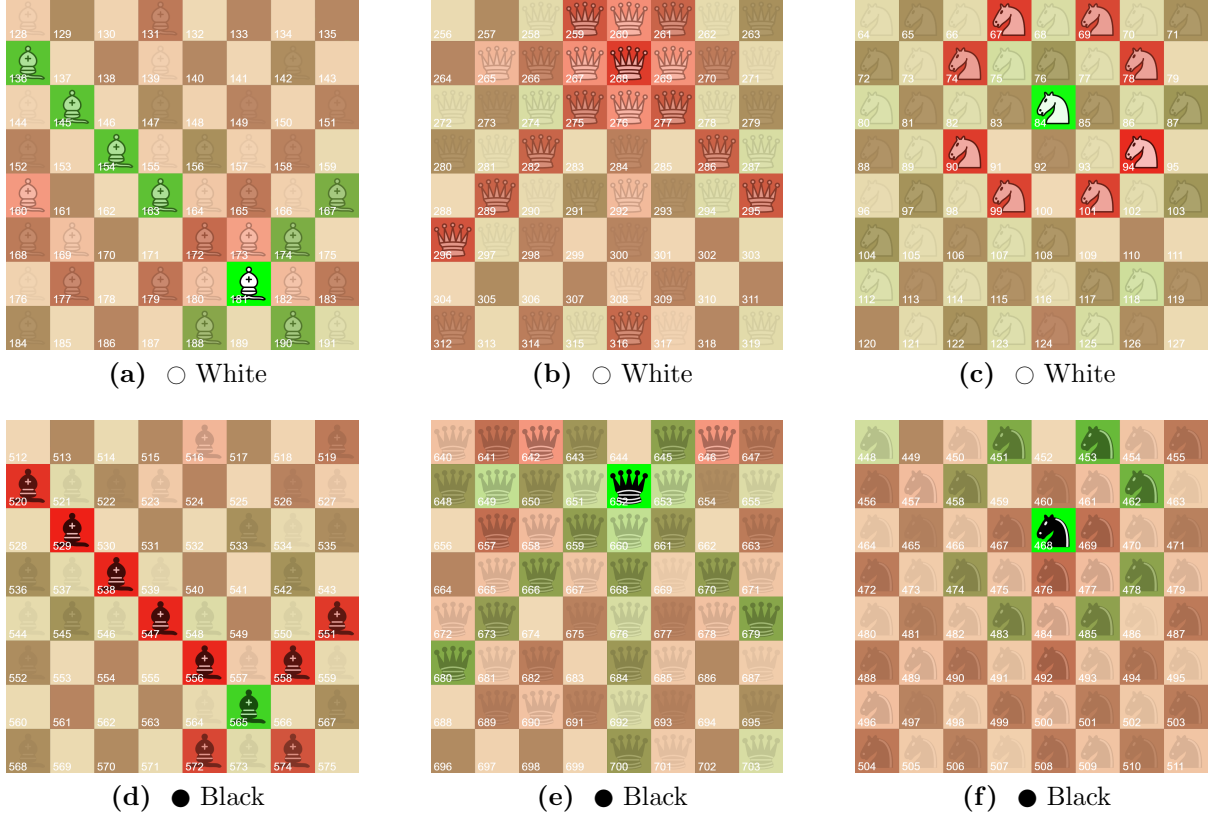
(d) ● Black

(e) ● Black

(f) ● Black

Figure 10. 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 `emitPlainEntry` code

```cpp
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';
}
```