



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Análisis de feature sets con redes neuronales NNUE para engines de ajedrez

July 19, 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>

Contents

1	Introduction	3
1.1	Chess Engines	3
2	Feature set (board encoding)	4
2.1	Sum \oplus	5
2.2	Indexing	5
2.3	Dead features	5
2.4	Feature sets	5
2.4.1	PIECE	6
2.4.2	COMPACT	6
2.4.3	KING-PIECE	6
2.4.4	PIECE-MOVE	6
2.4.5	HALF-RELATIVE(H V HV)KING-PIECE	7
2.4.6	HALF-TOP(PP)	7
2.5	Summary	7
3	Efficiently updatable neural networks	8
3.1	Layers	8
3.2	Efficient updates	9
3.3	Network	10
3.4	Quantization	12
3.4.1	Stockfish quantization scheme	12
3.5	Implementation	14
4	Engine implementation	15
5	Training	16
5.1	Source dataset	16
5.2	Method 1: Stockfish evaluations	16
5.2.1	CP-space to WDL-space	16
5.2.2	Loss function	17
5.3	Method 2: PQR triplets	17
5.3.1	Loss function	17
5.4	Setup	17
6	Experiments	19
7	Results	20
7.1	Active neurons	20

8	Final words	21
8.1	Conclusions	21
8.2	Future work	21

1 Introduction

1.1 Chess Engines

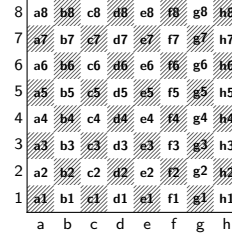
2 Feature set (board encoding)

To evaluate chess positions, we will use a neural network with an architecture explained in detail in the next chapter. In this chapter, we will 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, 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:

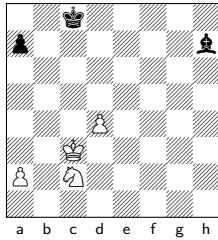
$$\begin{aligned}\text{FILE} &= \{a, b, \dots, h\} \\ \text{RANK} &= \{1, 2, \dots, 8\} \\ \text{SQUARE} &= \{a1, a2, \dots, h8\}\end{aligned}$$



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 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. 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	$(a, \text{○}), (a, \text{●}), (c, \text{●}), (c, \text{○}), (d, \text{○}), (h, \text{●})$	$(a, \text{♙}), (c, \text{♕}), (c, \text{♘}), (d, \text{♙}), (h, \text{♗})$

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.

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

2.1 Sum \oplus

The sum 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. $h, 8, \text{♙}, \bullet$), they **must** have different interpretations. For example, given the feature sets FILE_W where W is any white piece in the board and 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.

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

2.2 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. [example?]

2.3 Dead features

[arreglar, lo movi] 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: $(a8..h8, \text{♙}, \circ)$ and $(a1..h1, \text{♙}, \bullet)$ 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. 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.

2.4 Feature sets

In this section, we will define the feature sets that will be used in the experiments. We will start with some of the most basic yet reasonable feature sets, then move to feature sets that are used by engines or were used in the past, and finally some that have not been tried, to the best of our knowledge.

2.4.1 PIECE

This feature set is the most natural encoding for a chess position. There is a one-to-one mapping between pieces in the board and features:

$$\text{PIECE} = \text{SQUARE}_P \times \text{ROLE}_P \times \text{COLOR}_P$$

for every P piece in the board

$$64 * 6 * 2 = 768 \text{ features}$$

2.4.2 COMPACT

This is a very compact feature set that still retains all the information of the board, meaning everything can be reconstructed by the neural network:

$$\text{COMPACT} = (\text{FILE}_P \times \text{ROLE}_P \times \text{COLOR}_P) \oplus (\text{RANK}_P \times \text{ROLE}_P \times \text{COLOR}_P)$$

for every P piece in the board

$$2 * (8 * 6 * 2) = 192 \text{ features}$$

2.4.3 KING-PIECE

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

where K is the king to move and P is every *non-king* piece in the board

$$64 * (64 * 5 * 2) = 40960 \text{ features}$$

There are variations to this feature set, such as HALFKA2 or notably HALFKA2_HM that is currently the latest feature set used by Stockfish 16.1. I will not consider them in this work.

known as "KP" in the literature

if we skip the king, you may be thinking where does it get the information about the other king's side, blabla arquitectura Half

2.4.4 PIECE-MOVE

This feature set comes up from seeing the patterns recognized by the Piece feature set in section 5.5.5. When we observe... attack patterns...: P..

from y to?

With that defined...

$$\text{PIECE-MOVE} = \text{PIECE} \oplus (\text{SQUARE}_P \times \text{SQUARE}_{\text{move}(P)})$$

for every P piece in the board

$$768 + 64 * 64 = 4864 \text{ features}$$

Not friendly to efficiently update the network. It is almost always better to do a full refresh on eval.

2.4.5 HALF-RELATIVE(H|V|HV)KING-PIECE

$\langle side_king_file - piece_file + 7, side_king_rank - piece_rank + 7, piece_type, piece_color \rangle$
excl. king

$15 * 15 * 5 * 2 = 2250$ features (for HV)

only H or only V have $8 * 15 * 5 * 2 = 1200$ features

2.4.6 HALF-TOP(PP)

Statistical feature set, blabla, wasted features blabla

[JUGAR CON DIAGONALES]

2.5 Summary

Feature set	Tuple	# features
PIECE	$SQUARE_P \times ROLE_P \times COLOR_P$	768
COMPACT	$(FILE_P \times ROLE_P \times COLOR_P) \oplus (RANK_P \times ROLE_P \times COLOR_P)$	192
KING-PIECE	$SQUARE_K \times PIECE_P$	40,960
PIECE+MOVES	asd	4864
RELATIVEHV-KING-PIECE	asd	2250
TOPPP	asd	64

Table 1: Comparison of feature sets

3 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 [3], 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 [4].

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 HALFKP 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]

3.1 Layers

For this thesis, I have chosen to use the standard NNUE architecture, which consist of multiple 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 and training methods, I have chosen to stick with the standard architecture.

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.

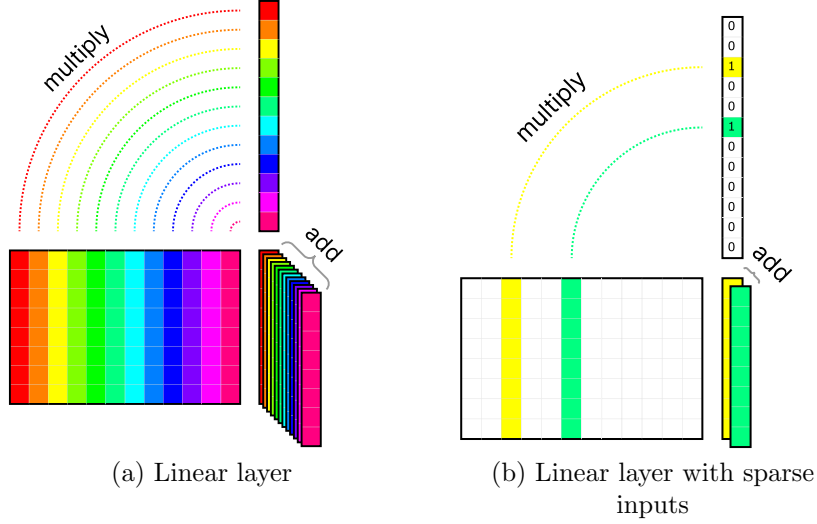


Figure 2: Linear layer operation comparison. Figures from [4].

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.

3.2 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.

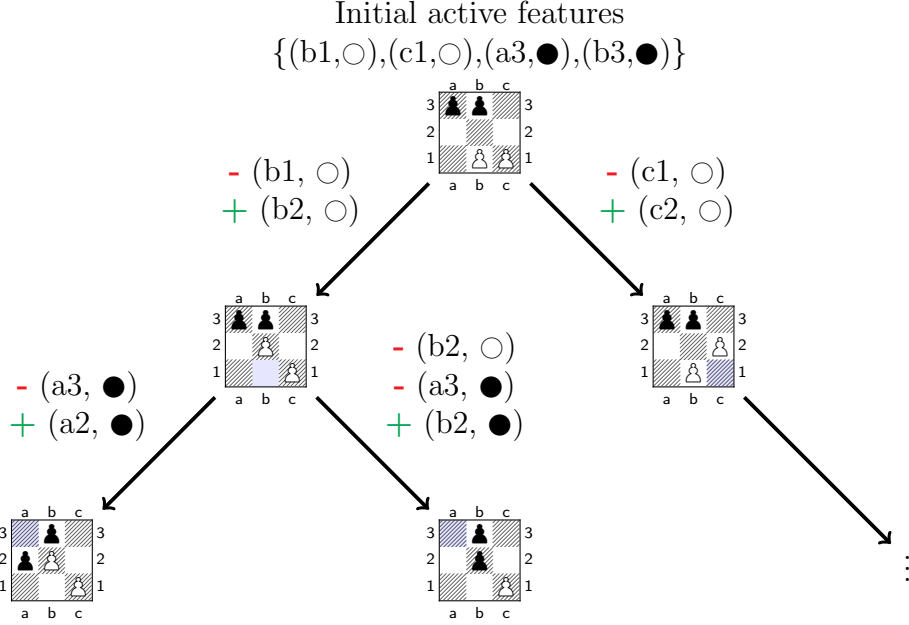


Figure 3: Partial tree of feature updates (removals and additions) for $\text{SQUARE}_P \times \text{COLOR}_P$ (white's point of view) in a simplified 3x3 pawn-only board.

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 following layers (usually small) have 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. [mencionar que tambien realmente es porque queremos codificar el que mueve y se va swapeando]

[agregar grafico de black \rightarrow white board \rightarrow encode, para mostrar como se flipea / swapea. arriba el white \rightarrow encode; poner los features activos quizas?]

3.3 Network

The network will be composed of four linear layers L_1 through L_4 , each but the last one followed by a clipped ReLU activation C_1 through C_3 . 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 O to P (W_3 weight, b_3 bias)
6. C_3 : Clipped ReLU of P
7. L_4 : Linear from P to 1 (W_4 weight, b_4 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 4, with example parameters.

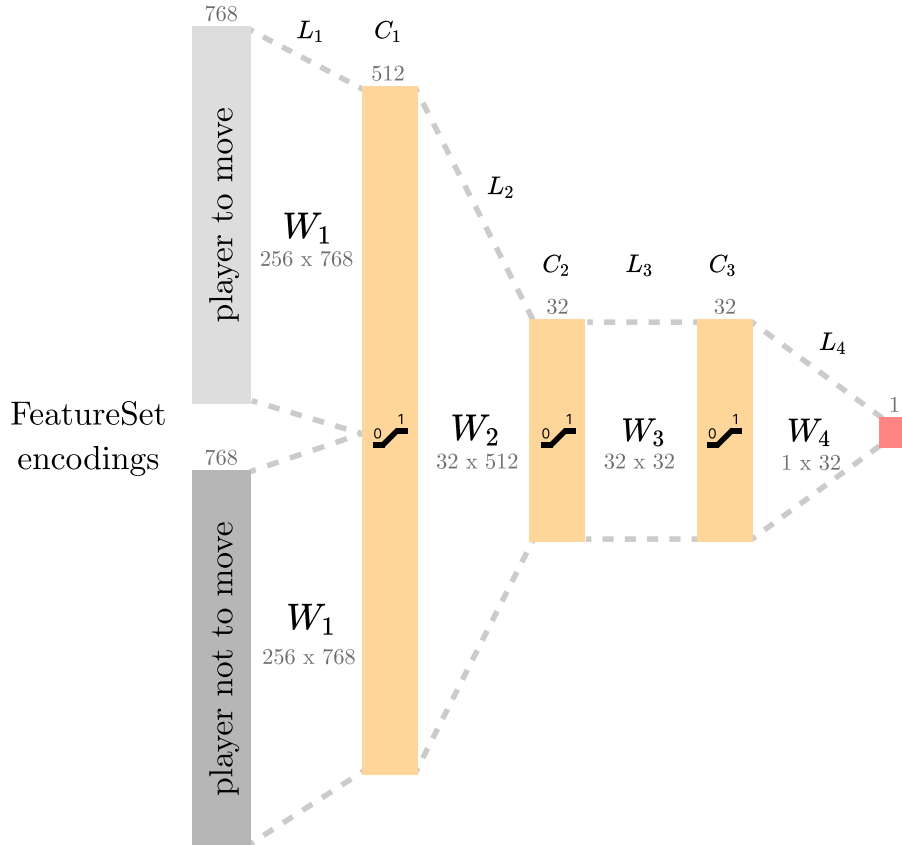


Figure 4: Neural network architecture with $N = 768$, $M = 256$, $O = P = 32$. Not to scale.

During search, the first layer L_1 is replaced by two accumulators to take advantage of efficient updates, as explained in the previous section. Figure 5 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 are computed as usual.

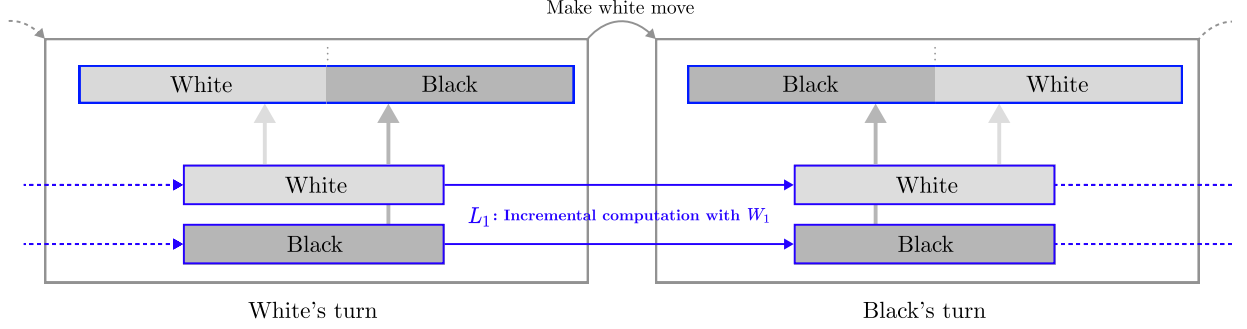


Figure 5: Concatenation of the first layer's output after a move is made.

3.4 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 maximum performance, as it sacrifices too much speed. 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.

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. To achieve this, the best is to use the smallest integer type possible everywhere, to process more values at once.

3.4.1 Stockfish quantization scheme

In this thesis, I will use the same quantization scheme used in the engine Stockfish [4]. 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.

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 clamp in the range $[0, 127]$ instead. 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`.

ACTIVATION RANGE SCALING = 127

Accumulator The purpose of this layer is to accumulate rows of the first layer's weight matrix. Later linear layers expect the input in `int8`,

. Since the output of this layer will be the input for the next linear layer and it has the ClippedReLU activation, the output will also be in 8 bits. But since we are accumulating 8 bits values and

and to output a clipped value of 8 bits for the next layer.

we can't accumulate using 8 bits since it would overflow.

COLUMN MAJOR

Linear layer The input to this layer will be scaled to the activation range because it takes the output of the previous ClippedReLU activation. We want the output to also be scaled to the activation range so it can be passed to the next. The activation range scaling is $s_a = 127$, as explained before.

To convert the weights to `int8`, we must scale them by some factor $s_W = 64$ (value used in Stockfish). 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 is a bit different since there is no activation afterwards, so we don't want the output to be scaled to the activation range (s_a). To be consistent with the Stockfish engine, the output values should be in the range $[-10000, 10000]$.

$$s_o = 9600$$

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

no se tiene el mismo problema que en el accumulator layer porque la multiplicacion en SIMD se hace en 32 bits (osea sin hacer overflow), para despues aplicar clippedrelu a eso.

3.5 Implementation

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

4 Engine implementation

:)

stockfish too complicated :)

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

Lichess is a free online site to play chess, and thankfully it provides a CC0 database [2] with all the games ever played on the site. It consists of several compressed PGN files¹ splitted by month since 2013, that add up to 1.71TB compressed. The whole database contains over 5.5 billion games, that equates to around 200 billion positions. In practice, that many positions are too much to handle so I'll use only a fraction of them and take only one sample per game to increase the diversity of positions.

A single game can have lots of positions, most of which are shared with millions of other games, mostly during the early game. This is a problem of its own: trying to sample positions from a game with a suitable distribution. In this work, I have chosen to only consider positions 20 half-moves into the game.

Each training method will generate a new derived dataset based on the positions described above, to later train the network.

5.2 Method 1: Stockfish evaluations

The main method to train the network will use the latest Stockfish evaluations as target. The objective is to train the network to predict the evaluation of a position as Stockfish would do.

First, we need to generate the training data. It is not known what makes a dataset good, but usually you can use the previous version of an engine to evaluate positions to train the next version. Stockfish uses a combination of datasets generated this way and evaluations from Lc0 that are more expensive to compute but have a higher quality, given the type of engine (it uses MCTS with a deep neural network).

I have chosen to generate the training set using evaluations from Stockfish version 16.1 at depth 9, as recommended by the authors of nnue-pytorch [4]. For each game, I uniformly sample a position (after 20 half-moves), run Stockfish and store the centipawn evaluation.

5.2.1 CP-space to WDL-space

The evaluations from Stockfish are in centipawns, which is not the exact number the network has to use as target.

decir que no usamos el outcome de la partida para el score

$$L_{\varepsilon}(y, f(x, w)) = \max\{0, |y - f(x, w)| - \varepsilon\}$$

¹Portable Game Notation: a textual format to store chess games (moves and metadata)

5.2.2 Loss function

$$L_\varepsilon(y, f(x, w)) = \max\{0, |y - f(x, w)| - \varepsilon\}$$

5.3 Method 2: PQR triplets

This is an additional technique I wanted to try, described in [1]. Remember that we are trying to obtain a function f (the model) to give an evaluation of a position. The method is based in the assumption that players make optimal or near-optimal moves most of the time, even if they are amateurs.

1. For two position in succession $p \rightarrow q$ observed in the game, we will have $f(p) \neq f(q)$.
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.

With infinite compute, f would be the result of running minimax to the end of the game, since minimax always finds optimal moves.

5.3.1 Loss function

$$L_\varepsilon(y, f(x, w)) = \max\{0, |y - f(x, w)| - \varepsilon\}$$

5.4 Setup

The project is written in two languages: Rust and Python. The Rust part is used to process PGN files, generate training data 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 separated in two steps:

1. Generate the training data from the Lichess database (the source dataset), for **a specific method**.
2. Train the network using the generated training data and **a specific feature set**.

Doing it this way allows to generate the training data once per method and train the network with different feature sets. Since generating the training data is the most time-consuming part of the process and I was iterating lots of different feature sets, it is ideal to have it separated. I could have an intermediate step, to generate the raw batch data from the method and feature set, but it is a waste in terms of practicality and disk space.

As depicted in Figure 6, the first step takes PGN files from the Lichess database and a training method (in this case *eval*, which stands for Stockfish evaluations) and builds

a training dataset from it. In this case, each sample is a FEN position (in red) and the centipawn evaluation (in blue).

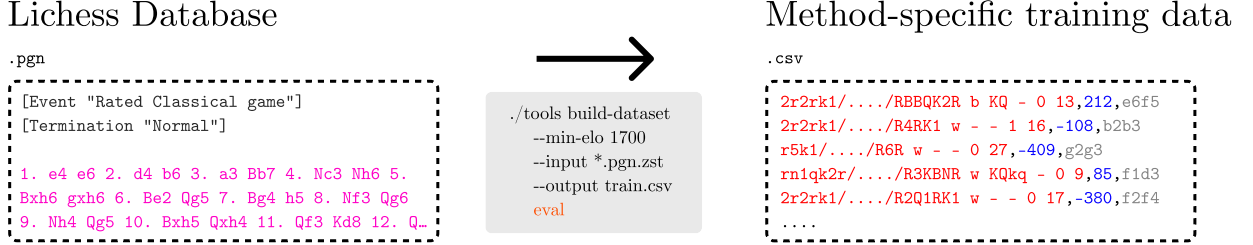


Figure 6: Diagram of the first step of the training process

Once the first step is done, the training can begin. The training process is started by running a Python script (`scripts/train.py`) and it requires to define the model architecture (number of neurons and hidden layers), 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, and if it is eval, it is the size of the feature set plus 1 for the centipawn evaluation (the target).

The training data obtained in the previous step 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

1. Diagonales?
2. Hacer un subset de PP (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?
- 3.

7 Results

hablar del tradeoff de los feature sets, la primera capa, y demás

vertical and horizontal data, probar dataset sin info vertical u horizontal / ambos y ver que pasa ver si agregar capas posteriores ayuda o no "layer layers small increase in perf"
measure updates per move average and refreshes average per FS

7.1 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)
future work: triplet loss?

8 Final words

8.1 Conclusions

8.2 Future work

future work: hacer que no sea uniforme el sampling de las posiciones para armar los datasets
deduplication de posiciones (al computar el score de Stockfish)

References

- [1] Erik Bernhardsson. *Deep learning for... chess*. 2014. URL: <https://erikbern.com/2014/11/29/deep-learning-for-chess.html>.
- [2] *Lichess Database*. URL: <https://database.lichess.org>.
- [3] 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.
- [4] Official Stockfish. *nnue.md*. URL: <https://github.com/official-stockfish/nnue-pytorch/blob/master/docs/nnue.md>.