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

July 1, 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

1.1 Chess Engines

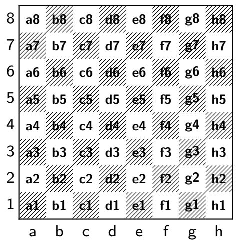
2 Feature set (board encoding)

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:

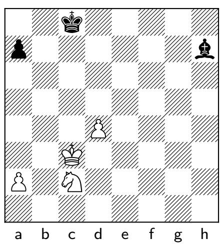
$$\textsc{File} = \{a, b, ..., h\}$$
$$\textsc{Rank} = \{1, 2, ..., 8\}$$
$$\textsc{Square} = \{a1, a2, ..., h8\}$$



And the following encode information about the pieces:

$$\textsc{Role} = \{\text{♙ Pawn, ♘ Knight, ♗ Bishop, ♖ Rook, ♕ Queen, ♔ King}\}^1$$
$$\textsc{Color} = \{\text{○ White, ● Black}\}$$

Since each set has to capture some information from the position, it must be stated explicitly. For example, consider the feature set $\textsc{File}_P \times \textsc{Color}_P$ 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 $\textsc{File}_P \times \textsc{Role}_P$, with a similar interpretation. An illustration of the active features of these two feature sets for the same board is shown in Figure 2.



	Feature set	
	$\textsc{File}_P \times \textsc{Color}_P$	$\textsc{File}_P \times \textsc{Role}_P$
Active features	(a, ○), (a, ●), (c, ●), (c, ○), (d, ○), (h, ●)	(a, ♙), (c, ♔), (c, ♘), (d, ♙), (h, ♗)

Figure 1: Active features of the feature sets $\textsc{File}_P \times \textsc{Color}_P$ and $\textsc{File}_P \times \textsc{Role}_P$ for the same board

[1]The 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, ♜, ●), 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 $\{a, b, ..., h\}$ 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, ♙, ○) 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. 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:

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

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

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:

$$\textsc{Compact} = (\textsc{File}_P \times \textsc{Role}_P \times \textsc{Color}_P) \oplus (\textsc{Rank}_P \times \textsc{Role}_P \times \textsc{Color}_P)$$
$$\text{for every } P \text{ piece in the board}$$

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

2.4.3 King-Piece

$$\textsc{King-Piece} = \textsc{Square}_K \times \textsc{Piece}_P$$
$$\text{where } K \text{ is the king to move and } P \text{ is every } \textit{non-king} \text{ piece in the board}$$

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

There are variations to this feature set, such as HalfKAv2 or notably HalfKAv2_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...

$$\textsc{Piece-Move} = \textsc{Piece} \oplus (\textsc{Square}_P \times \textsc{Square}_{move(P)})$$
$$\text{for every } P \text{ piece in the board}$$

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

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	$\text{Square}_P \times \text{Role}_P \times \text{Color}_P$	768
Compact	$(\text{File}_P \times \text{Role}_P \times \text{Color}_P) \oplus (\text{Rank}_P \times \text{Role}_P \times \text{Color}_P)$	192
King-Piece	$\text{Square}_K \times \text{Piece}_P$	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 (ƎUИИ Efficiently updatable neural network) is a neural network architecture that allows for very fast inferences. It was invented for Shogi by Yu Nasu in 2018.

In essence, NNUEs "Neural Network Update Efficient" are just regular neural networks that allow for really fast inferences.

... Most of the information described here can be found in Stockfish's documentation about NNUEs [1].

It is important to combine this with aggresive quantization techniques.

3.1 Architecture

arquitectura half, dos capas

3.2 Efficient updates

pesada al principio y liviana al final, acumular filas de la primera capa en domove, undomove

3.3 Stockfish quantization scheme

This is text bla bla. [2]

More testing on section **??**.

Rango de activación: en el modelo original usamos ClippedReLU, asi que queremos que el rango vaya de 0..1 a 0..127.

Siendo $\boldsymbol{x}$, $\boldsymbol{w}$ y $\boldsymbol{b}$ los parámetros de una capa lineal sin cuantizar e $\boldsymbol{y}$ la salida de la misma, se tiene que:

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{x}\boldsymbol{w} + \boldsymbol{b} \\ s_a s_w \boldsymbol{y} &= (s_a \boldsymbol{x})(s_w \boldsymbol{w}) + s_a s_w \boldsymbol{b} \end{aligned} \tag{1}$$

$s_o((s_a \boldsymbol{x})(s_w \boldsymbol{w}) + s_a s_w \boldsymbol{b}) = s_a s_w s_o \boldsymbol{y}$

3.4 Network sparsity

o combinar con 3.2? poner graficos con la sparsity de cada feature set, decir que es muy esparso todo y que se podría mejorar aún más

[2] This is a example footnote

4 Engine implementation

:)

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 serveral compressed PGN files[1] 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 [3]. 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

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

[1]Portable 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 $p \to q$ observed in the game, we will have $f(p) \neq f(q)$.

2. Going from p, not to q, but to a *random* position $p \to r$, 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.

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 network input data from the method and feature set, but it is a waste in terms of practicallity and disk space.

As depicted in Figure 2, 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 traning dataset from it. In this case, each sample is a FEN position (in red) and the centipawn evaluation (in blue).

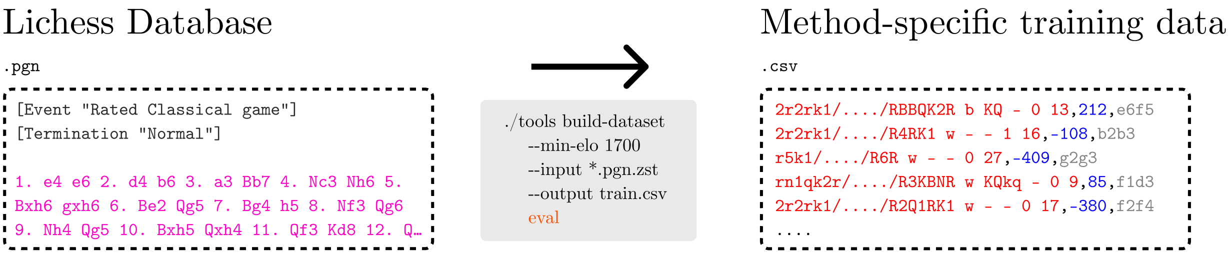


Figure 2: Diagram of the first step of the training process

Once the first step is done, the training can begin. The training process is started by running a Python script (`scrips/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 (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

The memory from the batch is directly copied to the GPU,

6 Results

6.1 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?

7 Final words

7.1 Conclusions

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