DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Análisis de feature sets en redes NNUE para motores de ajedrez

September 5, 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

Agradecimientos

:)

Contents

1 Introduction

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, 13], 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] (ИИUЭ) 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

1.2 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

[1] Introducing NNUE evaluation (Stockfish 12)

[2] Removal of handcrafted evaluation (Stockfish 16.1)

ability to customize encodings and a way to train them. The initial idea was to use Stockfish with the official Pytorch trainer [12]. 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 implemented in Rust, only making use of the most important 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 like material and pawn structure. In my case, I will replace it entirely with a neural network, explained in the following chapters.

2.1 Alpha-beta

A few more minor optimizations were made...

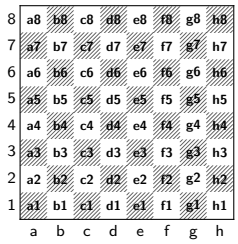
3 Feature set (board encoding)

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:

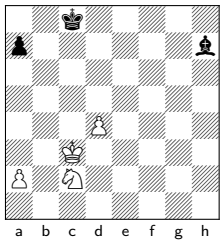
$$\begin{aligned} \textsc{File} &= \{a, b, ..., h\} \\ \textsc{Rank} &= \{1, 2, ..., 8\} \\ \textsc{Square} &= \{a1, a2, ..., h8\} \end{aligned}$$



And the following encode information about the pieces:

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

Since each set has to capture some information from the position, it must be stated explicitly. For example, consider the feature set $\textsc{File}_P \times \textsc{Color}_P$ 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 1.



	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.

3.1 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.

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. [example?]

3.3 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.

3.4 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.

3.4.1 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:

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

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

3.4.2 King-Piece

$$\textsc{King-Piece} = \textsc{Square}_K \times \textsc{Piece}_P$$
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 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

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 [12].

NNUE operates in the following priciples:

- **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, aggresive quantization techniques]

[tradeoff between speed and accuracy]

4.1 Layers

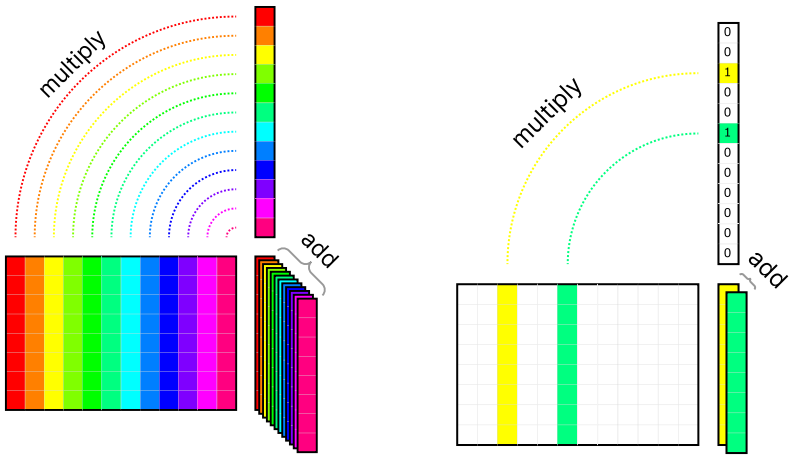
For this thesis, I have chosen to use the standard NNUE architecture, which consist of multiple linear (fully connected) layers and clipped ReLU activations. In the literature, there are other architectures that make use of polling layers, sigmoid activations and others, but since this work is about 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 $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, 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 [12].

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 $\boldsymbol{y = \min(\max(x, 0), 1)}$. 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.

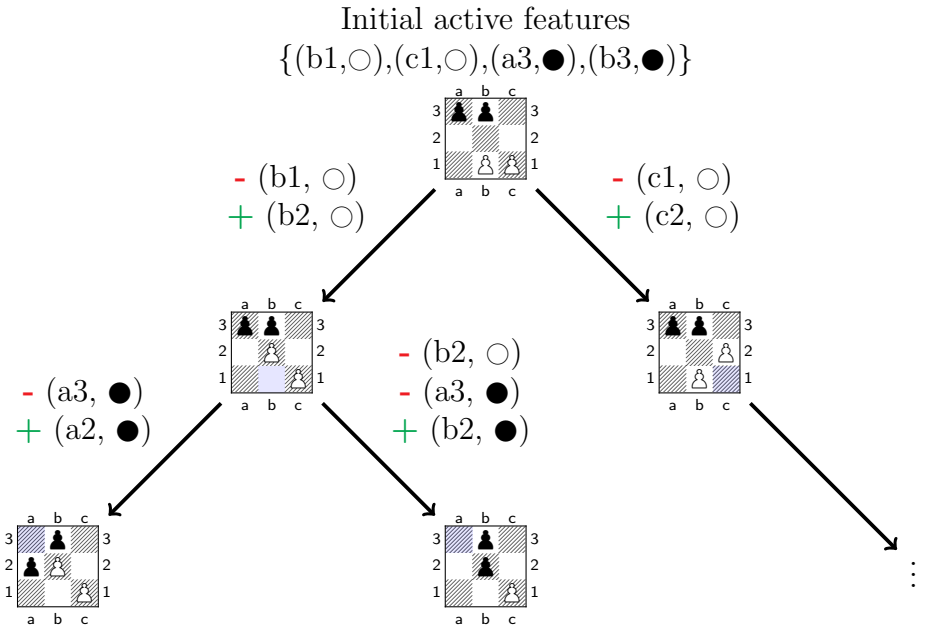


Figure 3. Partial tree of feature updates (removals and additions) for $\text{SQUARE}_P \times \text{COLOR}_P$ (white's point of view) in a simplified 3x3 pawn-only board.

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 → white board → encode, para mostrar como se flipea / swapea. arriba el white → encode; poner los features activos quizas?]

4.3 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 con-

catenated 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 $\boldsymbol{O}$ to $\boldsymbol{P}$ ($\boldsymbol{W_3}$ weight, $\boldsymbol{b_3}$ bias)
6. C_3: Clipped ReLU of $\boldsymbol{P}$
7. L_4: Linear from $\boldsymbol{P}$ to $\boldsymbol{1}$ ($\boldsymbol{W_4}$ weight, $\boldsymbol{b_4}$ 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.

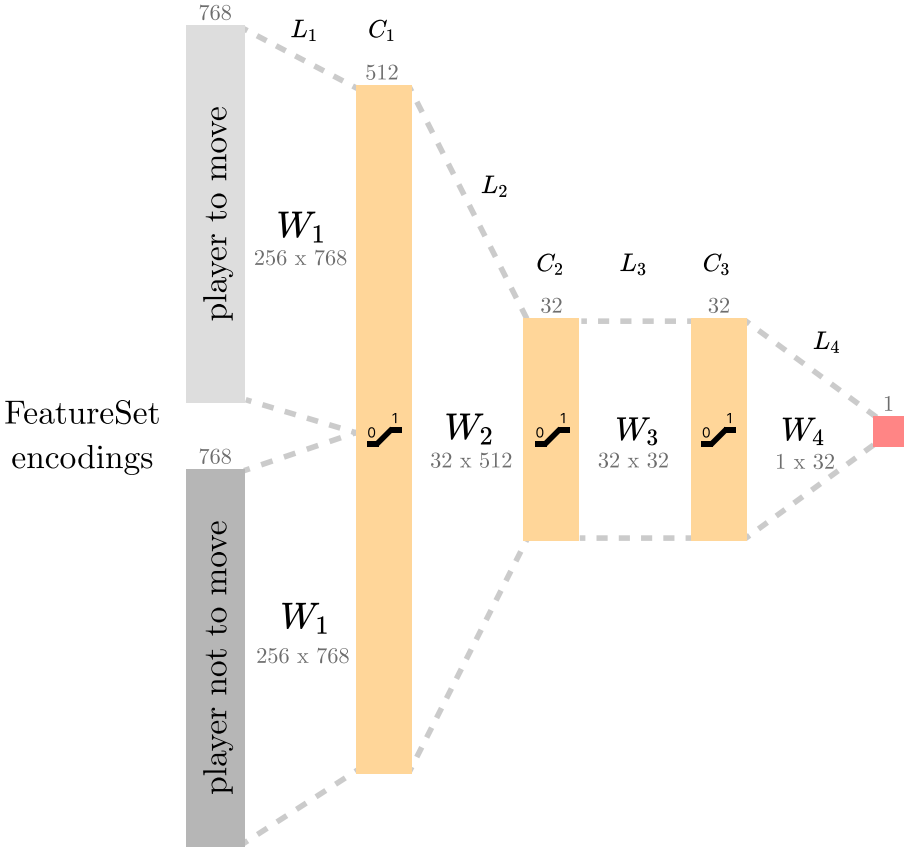


Figure 4. Neural network architecture with $\boldsymbol{N} = 768$, $\boldsymbol{M} = 256$, $\boldsymbol{O} = \boldsymbol{P} = 32$. Not to scale.

During search, the first layer L_1 is replaced by two accumualtors 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.

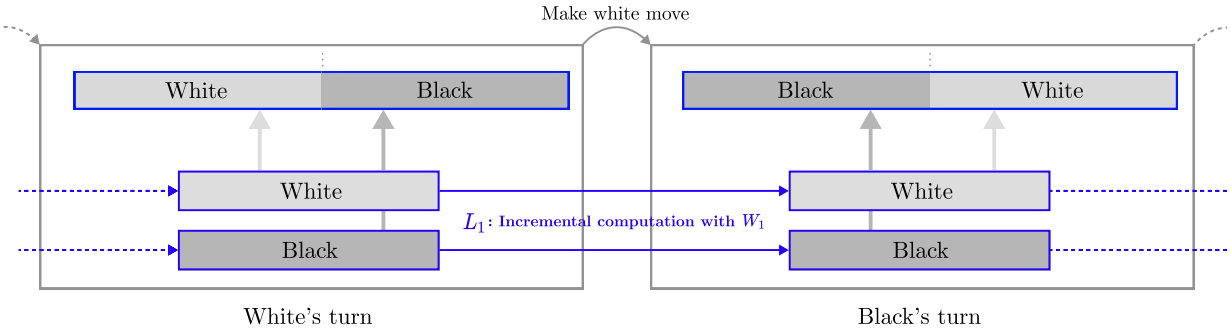


Figure 5. Concatenation of the first layer's output after a move is made.

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 maximum performance, as it sacrifices too much speed. 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.

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.

4.4.1 Stockfish quantization scheme

In this thesis, I will use the same quantization scheme used in the engine Stockfish [12]. 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 rango 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:

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

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

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 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 $[-10000, 10000]$.

$s_o = 9600$

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

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.

.

4.5 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).

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 [7] 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.

135 GB compressed binpacks 2.59 TB

48405107582 samples

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 10, as recommended by the authors of nnue-pytorch [12]. 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.

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

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

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

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 batch data from the method and feature set, but it is a waste in terms of practicallity 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 traning dataset from it. In this case, each sample is a FEN position (in red) and the centipawn evaluation (in blue).

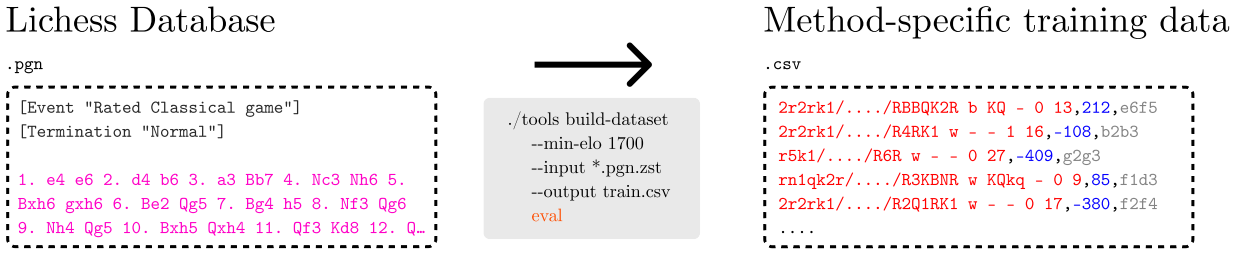


Figure 6. Diagram of the first step of the training process

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

With the tools and methodology defined in the previous sections, we can now proceed to the experiments. Each run is determined by its configuration, that can be divided into the following categories:

- **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.
- **Network architecture**: The size of each layer in the network. The first layer is the feature transformer, and its size roughly determinates how many patterns the network can learn. The following two layers should be tiny due the NNUE architecture.
- **Dataset**: blabla
- **Training method**: Can choose to use either Stockfish evaluations or PQR triplets. This determinates the format of the samples as well as the loss function. Most experiments will train using Stockfish evaluations.
- **Training hyperparameters**: The usual machine learning hyperparameters for training, such as batch size, learning rate and scheduler.

To assess the performance of a run or to compare a set of runs, the following indicators can be used:

[hablar de que hay runs particulares que miden otras cosas pero estas son generales]

- **Puzzle accuracy**:
- **Relative ELO performance**:
- **Inference performance (infs/s)**:
- **Training time (not important, one time)**:

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

6.1 Baselines

The objective is to find good baselines to compare the experiments that will follow.

P and KP Network size? Inference time?

a partir de cierto punto al loss le cuesta bajar pero las otras metricas siguen mejorando, asi que sigo entrenando

hacer side by side de Piece y KingPiece:

matriz con dos ejes: batch size y L1 size. con colores?

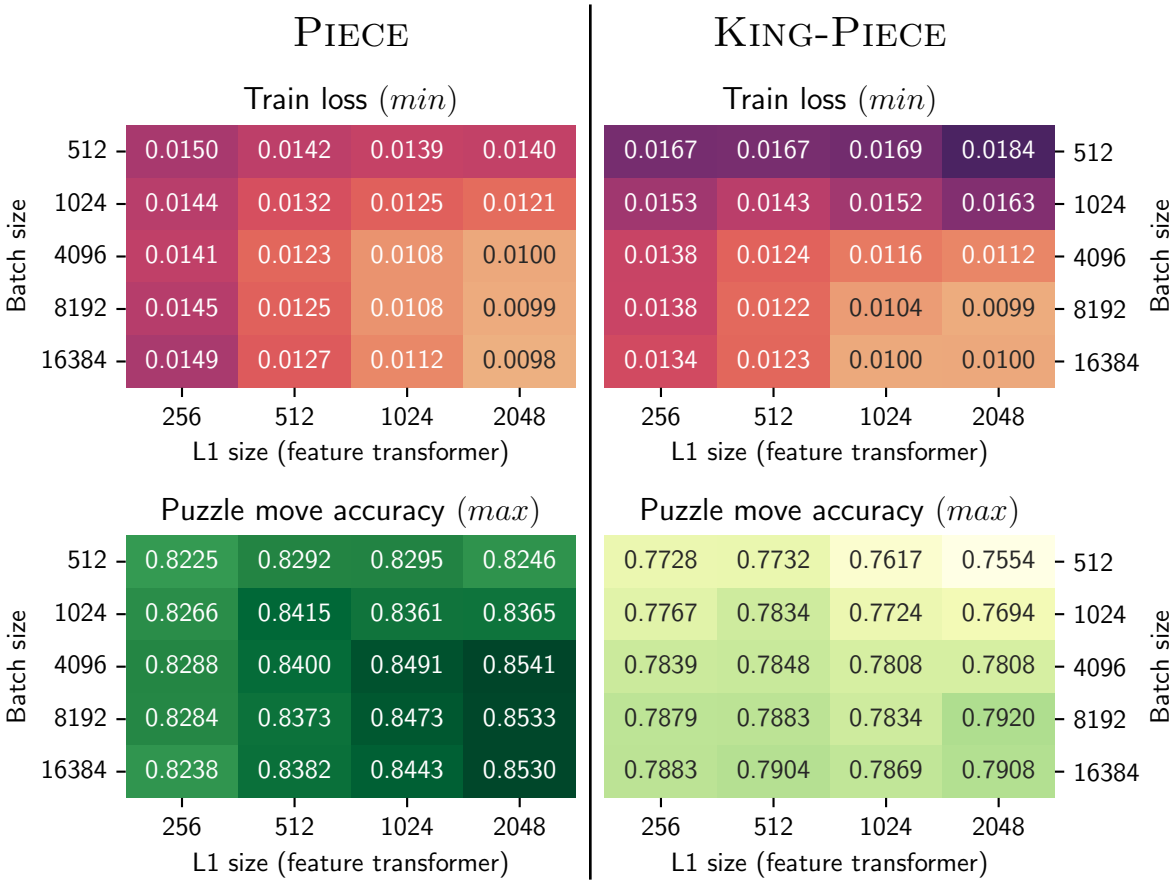
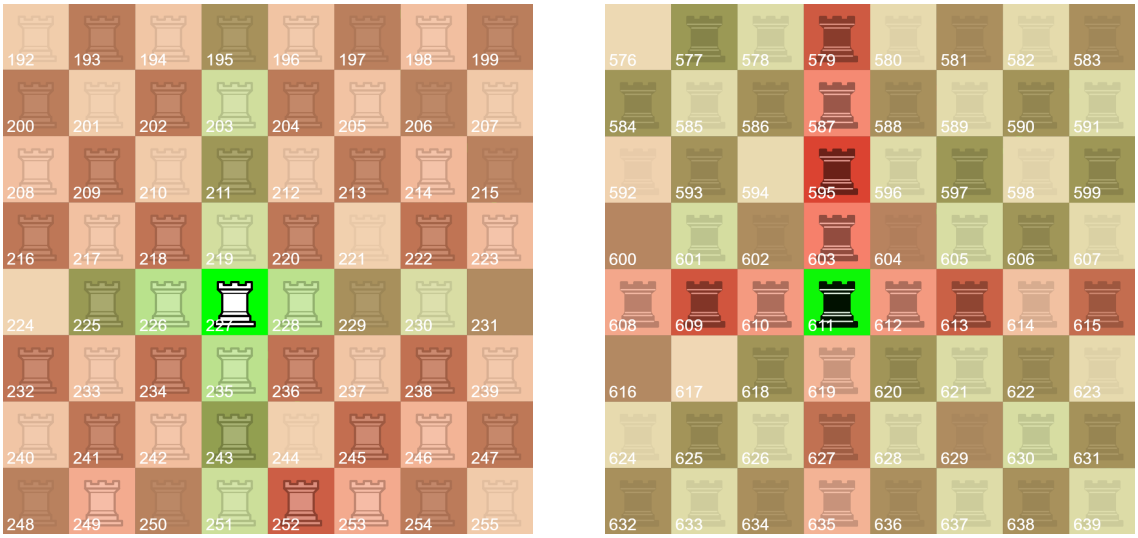


Figure 7. ASDASDASD

6.2 Axis encoding

Motivation. Looking back at the networks generated by PIECE in previous 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 PIECE 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.

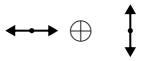
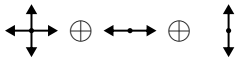
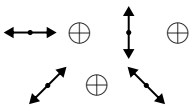
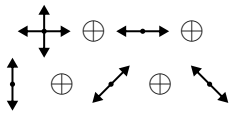
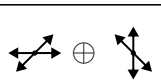
Experiment. In this first experiment, I will explore different positional encodings for the pieces on the board, combining the available axes. The canonical PIECE feature set encodes each piece's position using the square it is located. Note that this is the same thing as encoding the position for a 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. There are los of ways to index the board, but I will stick to the most common axes:

Horizontal (across files) | Vertical (across ranks) | Diagonal 1 | Diagonal 2

These axes coincide with the movement pattern of the pieces, which make them a good candidate to use as indexing dimensions. In table 1 I present the feature sets that I decided to explore. The feature sets are named according to the axes they combine.

Table 1. Axis encoding feature sets

Depiction	**Feature set**	**Definition** for every piece P in the board		**# of features**
	HV (PIECE)	$\text{FILE}_P \times \text{RANK}_P$	$\times\ \text{R}_P \times \text{C}_P$	768
	$\text{H} \oplus \text{V}$ (COMPACT)	$(\text{FILE}_P \oplus \text{RANK}_P)$	$\times\ \text{R}_P \times \text{C}_P$	192
	$\text{HV} \oplus \text{H} \oplus \text{V}$	$(\text{FILE}_P \times \text{RANK}_P \oplus \text{FILE}_P \oplus \text{RANK}_P)$	$\times\ \text{R}_P \times \text{C}_P$	960
	D1D2	$\text{DIAG1}_P \times \text{DIAG2}_P$	$\times\ \text{R}_P \times \text{C}_P$	2700
	$\text{D1} \oplus \text{D2}$	$(\text{DIAG1}_P \oplus \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	360
	$\text{HV} \oplus \text{D1D2}$	$(\text{FILE}_P \times \text{RANK}_P \oplus \text{DIAG1}_P \times \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	3468
	$\text{H} \oplus \text{V} \oplus \text{D1} \oplus \text{D2}$	$(\text{FILE}_P \oplus \text{RANK}_P \oplus \text{DIAG1}_P \oplus \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	552
	$\text{HV} \oplus \text{H} \oplus \text{V} \oplus \text{D1} \oplus \text{D2}$	$(\text{FILE}_P \times \text{RANK}_P \oplus \text{FILE}_P \oplus \text{RANK}_P \oplus \text{DIAG1}_P \oplus \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	1320
	$\text{HD1} \oplus \text{VD2}$	$(\text{FILE}_P \times \text{DIAG1}_P \oplus \text{RANK}_P \times \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	2880
	$\text{VD1} \oplus \text{HD2}$	$(\text{RANK}_P \times \text{DIAG1}_P \oplus \text{FILE}_P \times \text{DIAG2}_P)$	$\times\ \text{R}_P \times \text{C}_P$	2880

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 ($\text{H} \oplus \text{V}, \text{D1} \oplus \text{D2}, \text{H} \oplus \text{V} \oplus \text{D1} \oplus \text{D2}$) will perform worse overall, as to capture the exact position of pieces in the board,

the network will have to learn to relate two features for every location. This information is already available when there is a product of two dimensions.

Results.
RESULTADOS LOCO, RESULTADOS

6.3 Symmetry

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

6.4 Piece movement

Intentar capturar los patrones que se ven en P, asi se pueden reconocer patrones mas complejos.
Piece-Move
Bad perf.

6.5 Statistical features

Define k-Piece-Piece
King-Piece is a subset of Piece-Piece.
Top P
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?

6.6 Human behavior

PQR human behaviour. Medir estilo Maia. comparar? no va a ser tan bueno.
hablar del tradeoff de los feature sets, la primera capa, y demás
vertical and horizontal data, probar dataset sin info vertical u horizonal / 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

6.7 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)

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

future work: triplet loss?

deduplication de posiciones (al computar el score de Stockfish)

maybe implementing a custom engine was not a good idea. bugs and stuff

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

[13] 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.

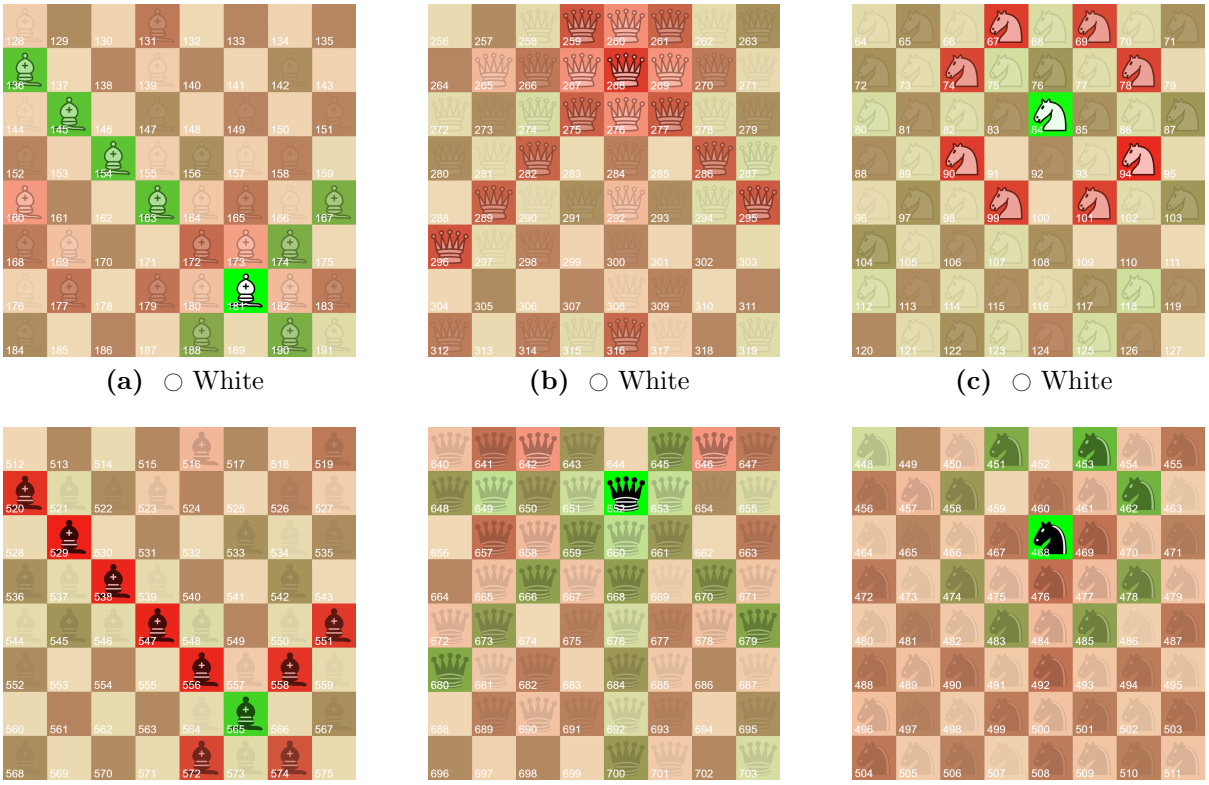
A Appendix

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

A.1 Baseline runs

Feature set	Train hyperparams		Network arch			Train loss	Runtime
	Batch size	Learning rate	L1 (FT)	L2	L3	*min*	*hh:mm:ss*
Piece	512	0.0015	256	32	32	0.01497	1:08:33
Piece	512	0.0015	512	32	32	0.01423	1:10:46
Piece	512	0.0015	1024	32	32	0.01393	1:15:32
Piece	512	0.0015	2048	32	32	0.01404	1:23:58
Piece	1024	0.0015	256	32	32	0.01439	0:36:14
Piece	1024	0.0015	512	32	32	0.01321	0:39:29
Piece	1024	0.0015	1024	32	32	0.01248	0:51:00
Piece	1024	0.0015	2048	32	32	0.01210	1:06:04
Piece	4096	0.0015	256	32	32	0.01412	0:13:52
Piece	4096	0.0015	512	32	32	0.01225	0:17:05
Piece	4096	0.0015	1024	32	32	0.01079	0:24:16
Piece	4096	0.0015	2048	32	32	0.01004	0:43:03
Piece	8192	0.0015	256	32	32	0.01447	0:12:31
Piece	8192	0.0015	512	32	32	0.01245	0:16:18
Piece	8192	0.0015	1024	32	32	0.01079	0:25:26
Piece	8192	0.0015	2048	32	32	0.00994	0:42:37
Piece	16384	0.0015	256	32	32	0.01493	0:12:23
Piece	16384	0.0015	512	32	32	0.01272	0:16:03
Piece	16384	0.0015	1024	32	32	0.01116	0:24:05
Piece	16384	0.0015	2048	32	32	**0.00975**	0:42:51
King-Piece	512	0.0015	256	32	32	0.01674	2:30:34
King-Piece	512	0.0015	512	32	32	0.01669	3:28:30
King-Piece	512	0.0015	1024	32	32	0.01694	4:18:07
King-Piece	512	0.0015	2048	32	32	0.01835	7:23:26
King-Piece	1024	0.0015	256	32	32	0.01528	1:54:48
King-Piece	1024	0.0015	512	32	32	0.01432	2:34:10
King-Piece	1024	0.0015	1024	32	32	0.01520	3:58:19
King-Piece	1024	0.0015	2048	32	32	0.01634	6:47:12
King-Piece	4096	0.0015	256	32	32	0.01379	1:38:58
King-Piece	4096	0.0015	512	32	32	0.01242	2:12:56
King-Piece	4096	0.0015	1024	32	32	0.01161	3:25:08
King-Piece	4096	0.0015	2048	32	32	0.01123	6:36:10
King-Piece	8192	0.0015	256	32	32	0.01375	1:32:59
King-Piece	8192	0.0015	512	32	32	0.01217	2:05:31
King-Piece	8192	0.0015	1024	32	32	0.01043	3:15:06
King-Piece	8192	0.0015	2048	32	32	**0.00991**	5:48:55
King-Piece	16384	0.0015	256	32	32	0.01344	1:35:51
King-Piece	16384	0.0015	512	32	32	0.01227	2:07:04
King-Piece	16384	0.0015	1024	32	32	0.01002	4:00:55
King-Piece	16384	0.0015	2048	32	32	0.01005	6:22:24

A.2 Axis encoding examples



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

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

Figure 9. Weights of different neurons in the L1 layer, which are connected to features in PIECE 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 PIECE), because it was prior to the first experiment. Refer to section 6.2.