Optimized Kernels in MobileNet Architectures for Computer Chess

Raphaël Mathiot,
LAMSADE, Université Paris Dauphine - PSL, CNRS, Paris, France
E-mail: raphael.mathiot@dauphine.eu

Olivier Goudet * and
LERIA, Université d'Angers, 2 Boulevard Lavoisier, Angers 49045, France
E-mail: olivier.goudet@univ-angers.fr

Tristan Cazenave
LAMSADE, Université Paris Dauphine - PSL, CNRS, Paris, France
E-mail: tristan.cazenave@dauphine.psl.eu

Abstract. This work investigates the use of various residual and MobileNet architectures within the Leela Chess Zero (Lc0) engine. In particular, it examines convolutional layers using specific kernels inspired by chess piece movement patterns. The most effective configuration, combining knight, rook, and bishop filters, is further optimized at a low implementation level to speed up network inference during move search. All network variants are trained on the recent T80 dataset, consisting of self-played games generated by Lc0. Their relative performance is then evaluated through a series of tournaments conducted under different time controls, including bullet, blitz, and rapid formats. This study extends the work presented in Goudet et al. (2024) at the 12th International Conference on Computers and Games (CG).

Keywords: Deep Learning, Computer Chess, Convolutional Neural Network, CUDA Kernels

1. INTRODUCTION

Chess has long been a challenge for artificial intelligence, offering a deterministic but combinatorially complex environment that requires strategic reasoning. Early AIs for chess were tree search algorithms using manually designed evaluation functions (Shannon, 1950). However, the exponential growth of the search space quickly revealed the intrinsic limitations of these paradigms, motivating the search for more scalable, data-driven methodologies (Campbell et al., 2002). The integration of neural architectures into chess engines has since precipitated a paradigm shift for the game. Deep reinforcement learning frameworks, embodied by AlphaZero, combine policy and value networks with Monte Carlo tree search (MCTS) to achieve superhuman performance in chess and other games (Silver et al., 2017, 2018). These hybrid systems have demonstrated that neural networks can internalize complex positional understanding and strategic planning, exhibiting strong generalization across vast state spaces without relying on explicit domain heuristics or handcrafted features (Schrittwieser et al., 2020).

*Corresponding author. E-mail: olivier.goudet@univ-angers.fr.

In AlphaZero, the game board is viewed as an 8x8 image, with different channels encoding the positions of the pieces on the board. To evaluate a position and predict the best move to play, a convolutional neural network takes this image of the game board as input. These outputs will enable a Monte Carlo Tree Search algorithm (MCTS) to be guided towards the most promising branches of the search tree. An open-source implementation inspired by AlphaZero, known as Leela Chess Zero (Lc0), has subsequently been developed (LCZero, 2018).

More recent developments, including Lc0's adoption of Transformer-based architectures (Monroe and Chalmers, 2024), have yielded substantial improvements over earlier convolutional approaches. Nevertheless, the aim of the present study is of a different nature. Rather than attempting to rival these state-of-the-art architectures in absolute performance, we seek to construct a lightweight experimental framework that makes it possible to isolate and assess the contribution of newly proposed chess-specific filters. Such filters could subsequently be incorporated into more expressive hybrid architectures, such as *EfficientFormer* networks (Li et al., 2022), which combine convolutional operations with attention mechanisms and have already demonstrated promising results in the domain of Go (Sagri et al., 2024).

In Goudet et al. (2024), we showed that MobileNets (Sandler et al., 2018), when augmented with newly designed convolutional kernels, constitute an efficient architecture for chess position evaluation. These kernels were specifically designed to reflect the principal movement patterns of key chess pieces, namely knights, bishops, and rooks. Our results further suggested that combining these filters within convolutional layers may lead to additional performance gains. However, the comparisons reported in Goudet et al. (2024) were based exclusively on predictive accuracy measured on a common validation set and did not account for actual playing strength. Such an evaluation remains incomplete, as practical performance in gameplay may also depend critically on inference latency, which can substantially influence the effectiveness of the architectures under consideration. In fact, the higher the latency associated with an architecture, the fewer nodes the algorithm will be able to evaluate in the allotted time at each turn of the game. Even though our architecture with MobileNets coupled with specific filters for chess had fewer trainable parameters overall than the reference Residual net architecture, its inference time could be longer.

To extend this work, we first evaluate the best architectures introduced in Goudet et al. (2024) in actual gameplay conditions by making them compete in tournaments under different time controls. Second, we propose a low-level CUDA implementation of the calculations performed by the chess filters in the convolution layers in order to reduce the inference time of the best architecture discovered, making it even more competitive.

The remainder of this paper is structured as follows: Section 2 recalls the context with the neural network architectures proposed in Goudet et al. (2024). Section 3 details the low-level CUDA implementation of the chess-specific convolutional kernels, designed to accelerate network inference and thereby improve computational efficiency during gameplay. Section 4 presents our experimental results, and Section 5 concludes the paper with a summary of our contributions and suggestions for future research.

2. NEW NEURAL NETWORKS FOR LEELA CHESS ZERO

Leela Chess Zero is a chess engine that evaluates positions using deep neural networks rather than traditional heuristics, guiding its move selection through Monte Carlo Tree Search. It learns entirely

through self-play, applying reinforcement learning to improve its understanding of chess without relying on human game databases.

In this section, we briefly review how the neural network works in Lc0 and discuss the elements related to the chess filters introduced in Goudet et al. (2024).

2.1. Architecture of neural networks in Lc0

For both training and evaluation, in the classic input encoding of Lc0 each chess position is transformed into a three-dimensional tensor comprising 112 feature planes of size 8x8. As detailed in Klein (2022), the first six planes represent the locations of each piece type for the player to move, while the subsequent six encode the corresponding information for the opponent. An additional plane indicates whether a position repetition has occurred (set to one if at least one repetition is detected). These 13 feature planes are extended by concatenating the encodings of the seven preceding positions, thereby providing the network with a historical context of the game. The remaining eight planes capture auxiliary state information, including castling rights for both sides and other rule-related indicators.

Following this encoding step, the resulting tensor of shape (112,8,8) is processed analogously to an image with 112 channels. It is passed through a sequence of blocks forming the trunk of the neural network. The final shared representation is then forwarded to three distinct output modules: the policy head, responsible for move probability estimation, the value head, which predicts the expected game outcome and the moves-left head predicting the expected number of moves remaining until the game ends. The general architecture of the network is illustrated in Figure 1. In this article, we will focus solely on the neural network trunk (green section) by testing different types of convolutional blocks (see section 2.1.1) and different sizes of neural network trunks parameterized by the number B of convolutional blocks and the number C of channels.

2.1.1. Neural network trunk

In Goudet et al. (2024), three convolutional block architectures were evaluated within the trunk of the neural network: the Residual block (He et al., 2016) (see Figure 2), the MobileNet block (Sandler et al., 2018) (see Figure 3), and the ConvNeXt block Liu et al. (2022). The study reported that the ConvNeXt configuration did not yield competitive performance in the context of the Lc0 framework. Consequently, in the present work, we restrict our analysis to the Residual and MobileNet blocks, which demonstrated superior effectiveness in prior experiments.

2.1.2. Neural network head configurations

On top of the sequence of B blocks, described in previous subsection, we choose a very simple configuration of the three heads used in Lc0, as displayed in Figure 1:[1]

- The *policy head* (orange part in Figure 1) adopts the classical Lc0 architecture. It begins with a pointwise (1×1) convolutional layer comprising 32 output channels, which transforms the feature map produced by the trunk, of dimension $(C, 8, 8)$, into a tensor of shape $(32, 8, 8)$. This representation is subsequently flattened and passed through a fully connected layer, yielding an output vector, of dimension 1858, which corresponds to a preference score for every action available in the current position.

[1] It should be noted that this architecture does not reflect the most recent advances in policy head design within Lc0. The purpose of this study is instead to isolate and assess the contribution of chess-specific filters in the network trunk.

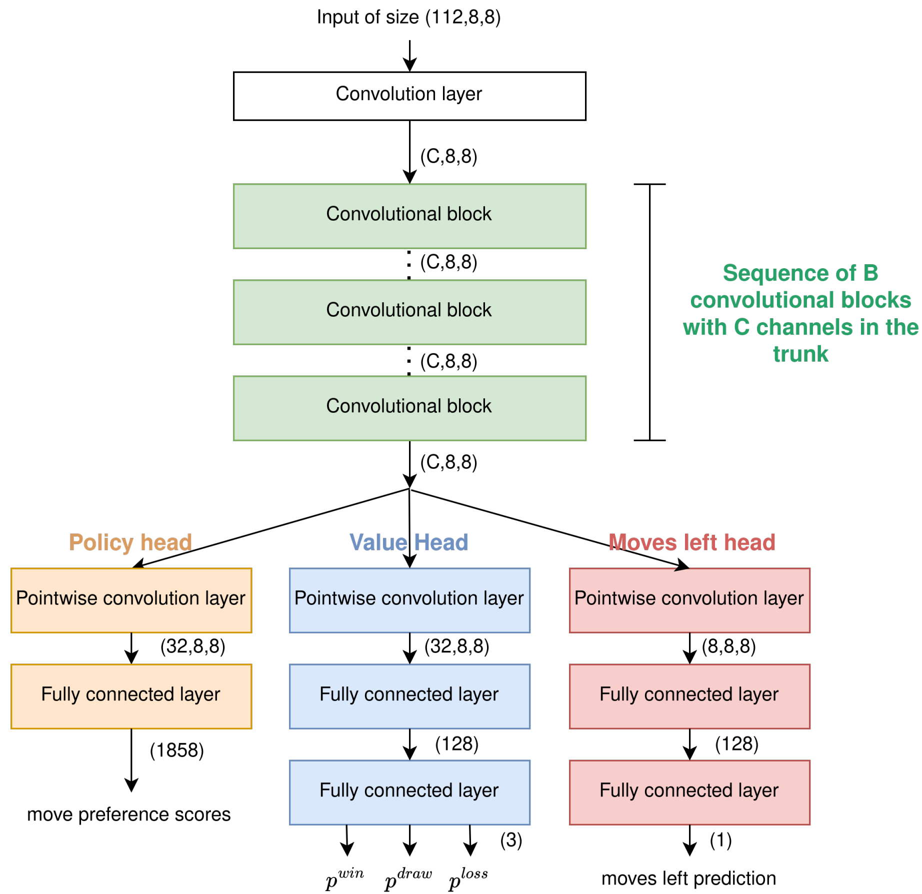

Fig. 1. Lc0 architecture

- The *value head* (blue part) consists in a pointwise (1×1) convolutional layer with 32 output channels, followed by a first dense layer with 128 neurons and a second dense layer with 3 neurons and softmax activation function. The three outputs model the probability of winning, drawing and losing.
- The *moves-left head* in Lc0 (red part) consists of a pointwise convolutional layer followed by two fully connected layers producing a single scalar output. This prediction is normalized and trained jointly with the policy and value heads. It provides the network with temporal context, helping it recognize game phases and impending results. Overall, it enhances training stability and improves the accuracy of value predictions.

2.2. Chess filters in convolutional layers

In Goudet et al. (2024), we introduced several variants of the convolutional layer used in the Residual and MobileNet blocks presented in Section 2.1.1, incorporating chess-specific convolutional kernels within both the standard and depthwise convolutional layers. The central motivation was to enable the network to more effectively capture structural patterns in chess positions that reflect the movement dynamics of different piece types.

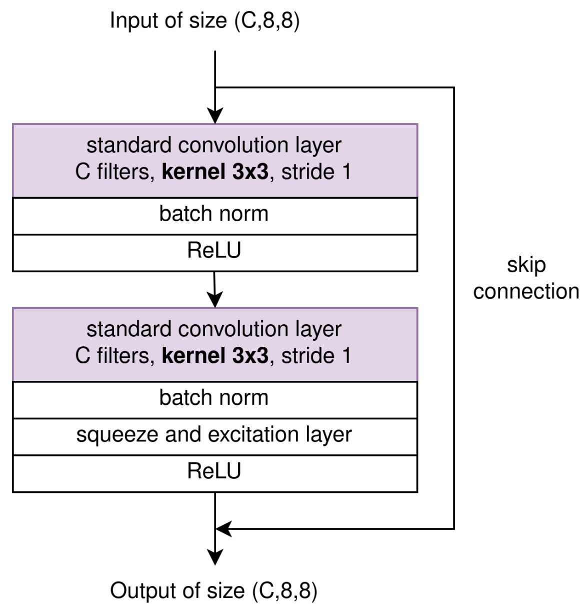

Fig. 2. Residual block with *C* channels in the trunk used in the original version of Lc0

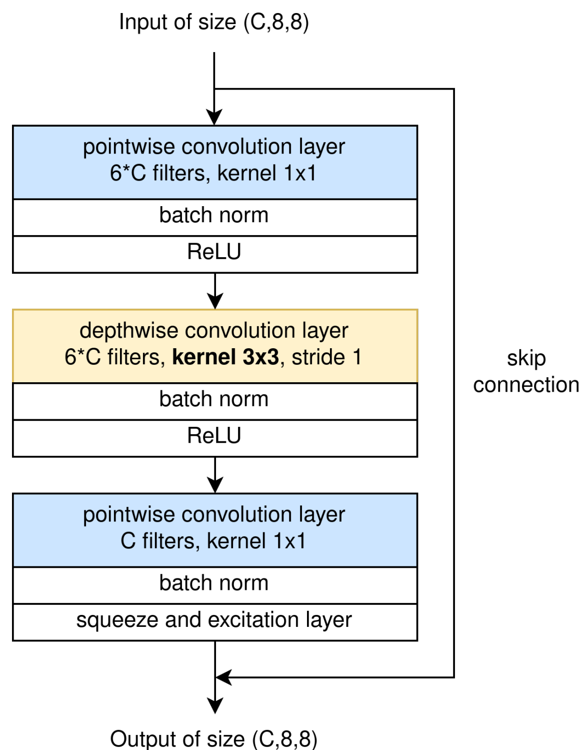

Fig. 3. MobileNet block with *C* channels in the trunk and depthwise multiplier set to 6.

Figure 4 depicts the standard 3x3 kernel used in the Residual network of Lc0. The kernel is computed on a patch of 3 by 3 squares of the board.

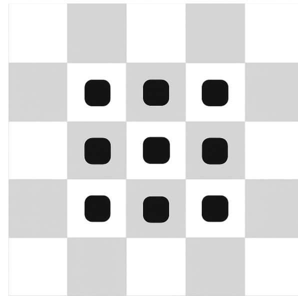

Fig. 4. Standard 3x3 kernel
used in Lc0

Figures 5–7 illustrate the three specific chess kernels introduced in Goudet et al. (2024) to replace the standard 3x3 kernel. These chess-specific kernels are implemented as 5x5 convolutional filters. Each kernel computes features by considering the central element of the patch along with positions within the 5x5 grid that correspond to characteristic movement patterns of different chess pieces: diagonal offsets for the bishop kernel, horizontal and vertical offsets for the rook kernel, and L-shaped offsets for the knight kernel. By encoding these movement priors directly into the convolutional layer, the network is better equipped to capture spatial dependencies and piece-specific interaction patterns, thereby facilitating a more informed positional analysis.

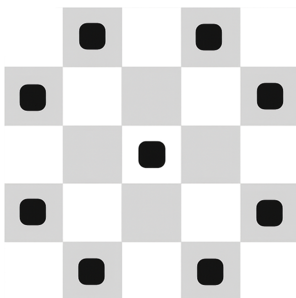

Fig. 5. Knight kernel.

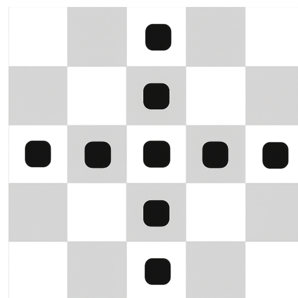

Fig. 6. Rook kernel.

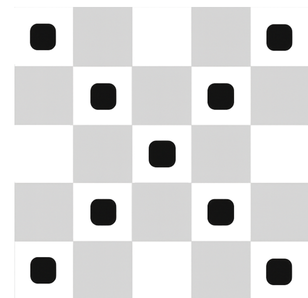

Fig. 7. Bishop kernel.

It is important to note that the computational cost associated with these chess-specific kernels is in principle equivalent to that of a standard 3x3 convolution, as both involve nine active elements in the calculation of each patch. In our previous work (Goudet et al., 2024), convolution operations during training were implemented using full 5x5 kernels combined with binary masks to zero out irrelevant positions. A straightforward approach for integrating these sparse kernels into the inference engine would be to treat them as conventional 5×5 kernels and rely on the standard cuDNN convolution implementation, which processes 25 weights per channel. While this method is functionally correct, we hypothesize that it leaves significant room for optimization. In particular, the increased memory footprint associated with loading and storing 25 weights per channel, rather than only the 9 non-zero coefficients, can substantially degrade performance in a memory-bound operation such as convolution. To address this limitation, Section 3 introduces a low-level CUDA implementation that explicitly stores, loads, and computes only the unmasked elements of the chess kernels. This optimization targets execution times comparable to those of standard 3×3 convolutions, thereby enhancing practical performance during gameplay.

Furthermore, if the central square is excluded from the chess filters illustrated in Figures 5–7, the remaining positions form a partition of the 5x5 grid. This structural complementarity suggests that these filters capture distinct yet synergistic movement patterns, motivating the design of hybrid convolutional layers that integrate multiple chess-specific kernels within a single operation to enhance positional analysis.

An example of such a depthwise convolutional layer that can be used in the MobileNet blocks (see Figure 3) is presented in Figure 8. In this design, the input channels are partitioned into subsets, each processed by a distinct kernel type. Specifically, in the *knight–rook–bishop* (KRB) configuration, the first third of the channels employs kernels encoding knight movements, the second third applies kernels corresponding to rook movements, and the final third utilizes bishop kernels. We also define an alternative configuration, termed *rook–bishop* (RB), in which half of the channels are processed by rook kernels and the remaining half by bishop kernels.[2]

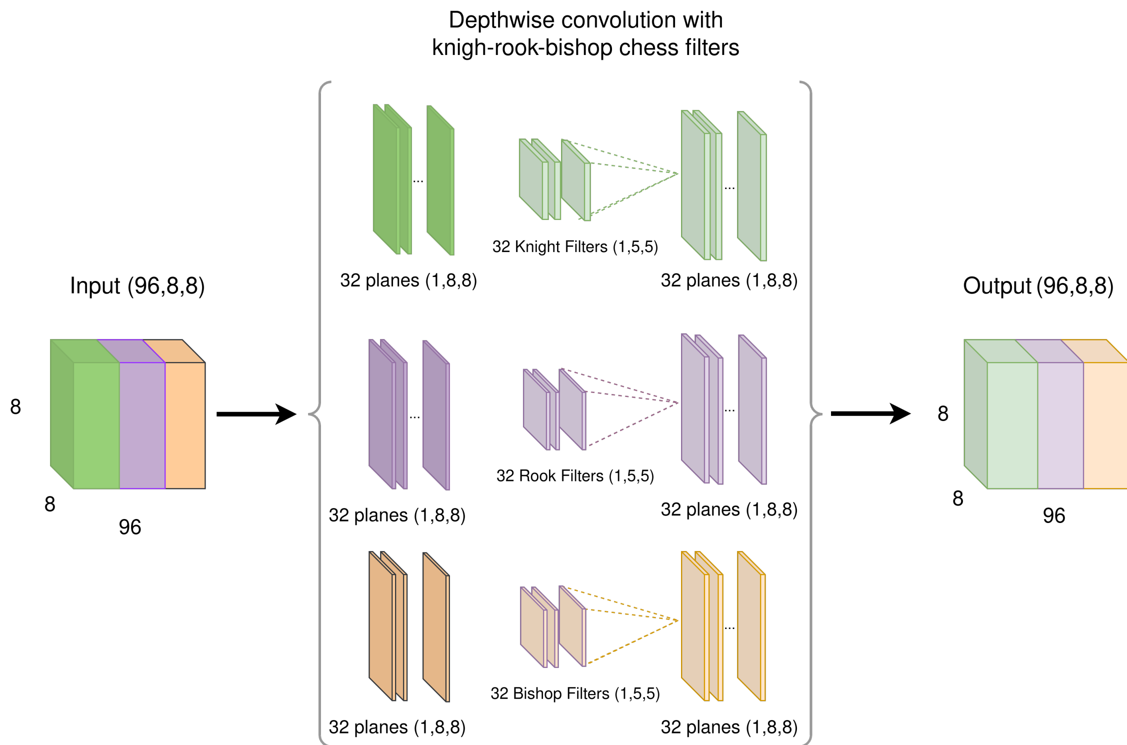

Fig. 8. Depthwise convolution with knight-rook-bishop chess filters applied to input tensor of size $(96, 8, 8)$. Source Goudet et al. (2024).

In Figure 9 is depicted a standard convolutional layer taking as input a tensor of size $(96, 8, 8)$ and using the knight-rook-bishop version of the chess filters. It can be used in Residual blocks (see Figure 2).

Empirical results obtained in Goudet et al. (2024) indicate that combining heterogeneous chess kernels within the same convolutional layer leads to improved performance compared to using a single filter

[2] It has been shown in Goudet et al. (2024) that these two kernels, rook and bishop, are the most effective, probably because they involve the movement of more different chess pieces. Whereas the kernel related to the movement of knights is more specific to a particular piece. In fact, it only makes an additional contribution when the number of channels is very large, as we will illustrate in the experimental section.

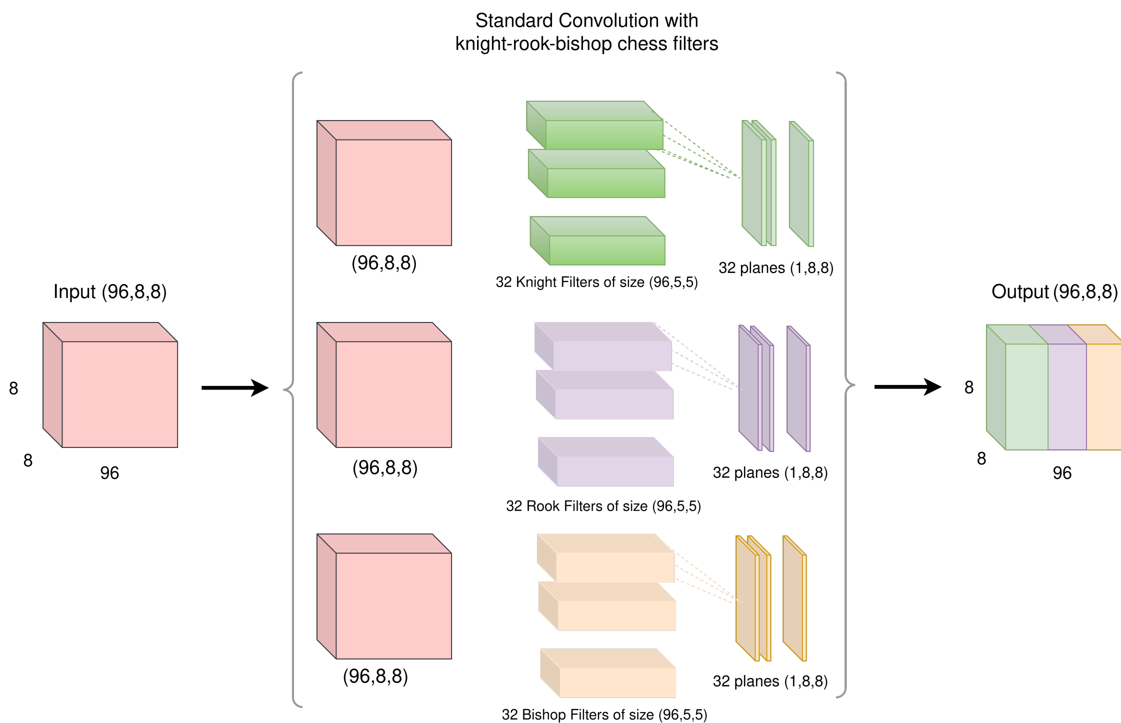

Fig. 9. Standard convolution with knight-rook-bishop chess filters applied to input tensor of size $(96, 8, 8)$.

type. This suggests that integrating multiple movement-based priors enhances the network's capacity to extract complementary spatial and strategic features from chess positions. In the following section, we detail how these convolution filters were implemented in the Lc0 neural network architecture.

3. LOW-LEVEL CUDA IMPLEMENTATION OF THE CHESS-SPECIFIC CONVOLUTIONAL KERNELS FOR THE LC0 CHESS ENGINE

In (Goudet et al., 2024), after training on the same dataset, the neural network architectures introduced in the previous section were evaluated by comparing the accuracy of their policy and value computed on the same validation set. This analysis was conducted using the training framework of the Lc0 project, implemented in Python and based on the TensorFlow library.[3] However, accuracy alone does not always directly translate into a measurable difference in playing strength in this context. In fact, inference speed is just as crucial, because the ability to select strong moves within strict time limits also depends heavily on the number of MCTS nodes opened per second, and therefore on the computational efficiency of the neural network.

Consequently, one of the contributions of this work is the effective integration of the new architectures proposed in the part of the Lc0 source code that enables the chess engine to run. To achieve this,

[3] Our training code for the Lc0 neural network, incorporating the new architectures proposed in (Goudet et al., 2024), is adapted from the official training repository https://github.com/LeelaChessZero/lczero-training and is publicly available at https://github.com/LEAHPARAPHAEL/lczero-training-new.

we adapted the original Lc0 implementation for gameplay[4] and released an updated version of the program, available at https://github.com/LEAHPARAPHAEL/lc0. The Lc0 engine used to play is written in C++ and relies on several GPU-acceleration libraries, most notably cuDNN. Because our architectures introduce a novel convolutional operator based on depthwise separable convolution, it was necessary to implement this new layer type within the C++ inference pipeline.

In this section, we introduce three distinct implementations of custom depthwise convolutional layer used in MobileNet block:

(1) **Baseline implementation:** a straightforward, naive version relying on the standard cuDNN convolution operator.
(2) **First optimized implementation:** an improved variant of the depthwise convolutional layer, tailored to the specific size of the chessboard.
(3) **Second optimized implementation:** a further refined version integrating the chess kernel at a lower level of CUDA implementation to achieve superior performance.

The source code of this last optimized version of the depthwise convolution layer with the combination of the knight-rook-bishop filters is given in Appendix A.

3.1. A first naive implementation using cuDNN

Writing memory-efficient Deep Learning layers in CUDA can be a tricky task, so our first approach was to use a preexisting implementation of a convolution operation, and adapting it to model a depthwise convolution. This can be done very easily using cuDNN, the NVIDIA library offering predefined methods for Deep Learning operations. Starting from a standard convolution, we modify the number of groups along the channels dimension using the `cudnnSetConvolutionGroupCount` method.[5] By setting the group count equal to the number of input channels, the grouped convolution effectively becomes a depthwise convolution, thereby reproducing the desired operation with minimal implementation overhead.

In the remaining of the paper, the version of the neural network using this depthwise convolution operator with the combination of the knight-rook-bishop filters (cf. Figure 8) is named `mob_krb`.

Note that, because we use a generic cuDNN implementation, the grouped convolution computes the full convolution operation by loading the 25 elements of the kernel. This is an unsatisfactory practical solution as it requires loading 16 extra parameters per kernel, which can lead to serious performance bottlenecks for depthwise convolutions, which are known to be memory-bandwidth bound. Thus, we quickly shifted our attention to an implementation more suited to our structure, for which we had to delve into CUDA programming.

3.2. Custom depthwise kernel in CUDA

Now, we describe how we implemented a custom depthwise convolution kernel in CUDA.

[4]https://github.com/LeelaChessZero/lc0

[5]This method, now deprecated, was the standard way to implement a depthwise convolution in cuDNN 7, which corresponds to the version of cuDNN used in the Lc0 repository. Potential faster versions of this method are now available in the cuDNN 9 version.

One of the most challenging aspects of writing a CUDA kernel is that we have to manage the parallelization by hand, by means of threads partition and alignment. However, this also allows us to design a specific parallelization configuration that is optimized for the specific task at hand.

The efficiency of this kernel relies on the fact that the chess board is already well suited for GPU optimization. Indeed, when designing a CUDA kernel, the computations are parallelized across several blocks, which are groups of threads executed on the same streaming multiprocessor of the GPU. Inside these blocks, the threads are organized in groups of 32, called warps. Threads inside of the same warp can exchange information through registers very efficiently.

We can now see the advantage of having a board made of 64 squares. It is easy to partition each input plane, of shape (8,8), into two rectangles of shape (8,4), each containing 32 input elements. Therefore, one input plane can be processed by two warps of 32 threads. This is a nice solution because it ensures that threads from the same warp access contiguous parts of the global memory when loading the weights and inputs : the first warp in charge of the upper half processes the 32 first elements from the input (organized in row-major order), and the second one the last 32.

Using this layout, we address the issue of memory-bandwidth by cutting in half the number of load operations, by packing fp16 weights and inputs from two adjacent channels as a single `half2` number, each half corresponding to one channel.

The result of this optimization is illustrated in Figure 10, where we display the 64 threads from two warps responsible for processing two adjacent input channels using this `half2` datatype.

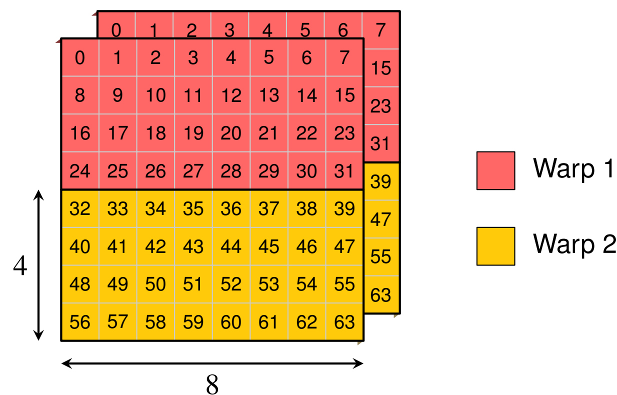

Fig. 10. Warp alignment and `half2` packing of two consecutive input channels in CUDA

The introduction of these two optimizations results in a new version of our previous network `mob_krb`, named `mob_krb_opti1`.

3.3. Low level CUDA implementation of chess kernels

Now that we have implemented an efficient depthwise convolution operator, we can add on top of the previous optimizations switching from a regular 5×5 kernel to a chess kernel with only 9 nonzero weights. Because we have full control over the convolution operation, we load only the 10 necessary weights (9 weights + bias), and spare the overhead of loading the 16 others. Building on the previous work, we also use the fact that threads from the same warp process the same input channel (more precisely, the same combination of two adjacent 16-bit input channels packed into one), to load only once the 9 kernel weights per warp, and share them through registers (see lines 63-74 of the code

given in Appendix A) which is extremely fast. Doing this, we minimize memory access redundancy by only accessing twice the same weight (once per warp) instead of once per thread.

Moreover, because we do not need the 16 zero weights anymore, we can skip saving them after training the model. This means that we save and manipulate only 9 weights per kernel in memory, which has the positive side effect of aligning them for efficient warp loading. For this, we chose to order these weights from top to bottom, then from left to right. This process is illustrated in Figure 11. Then, we multiply these 9 weights with the corresponding squares of the input channel, whose indices depend on the type of filter and thus on the index of the channel (first third of the channels for the knight kernels, second for the rook kernels, and third for the bishop kernels), and add the results using a fused multiply-add operation to obtain a complete convolution. This is implemented three times for the three types of possible chess kernels (see lines 101 to 198 of the code given in Appendix A).

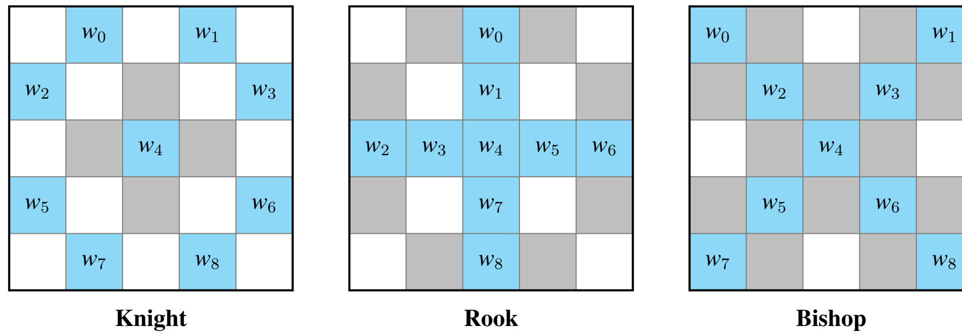

Fig. 11. Keeping only the 9 relevant weights for each chess kernel type

To sum up all of our optimizations, we can take a look at Figure 12, which displays on the left part the way warps are organized by processing half planes, and on the right part a padded convolution operation with a knight filter on one of these input channels. Replacing the standard kernels with chess kernels gives rise to a last fully optimized version of the network, named `mob_krb_opti2`, which we will compare to the two other versions of the same network in the following section.

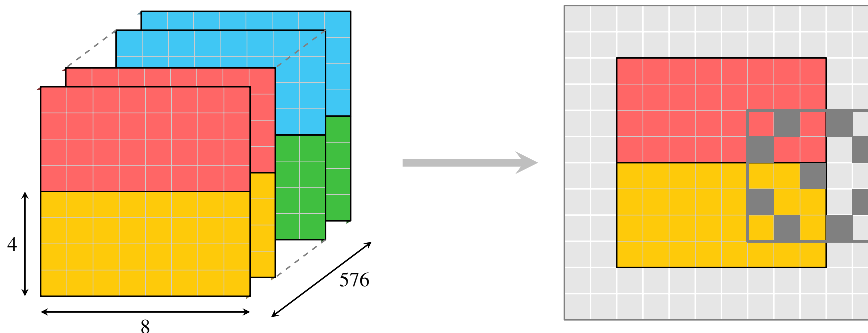

Fig. 12. Efficient implementation of a depthwise convolution kernel in CUDA.

4. EXPERIMENTAL RESULTS

This section aims to address two research questions through empirical evaluation. The first concerns the comparative impact of different convolutional block configurations on neural network performance, assessed in terms of gameplay strength. The second examines the effect of the low-level CUDA implementation of the chess-specific kernels, introduced in Section 3, on the computational efficiency of the Lc0-based network. These questions are critical for practical deployment because gameplay strength depends not only on the accuracy of position evaluation but also on inference latency, which directly influences the number of nodes explored within time constraints, a decisive factor in competitive chess engine performance.

4.1. Training and validation test sets

In this paper, we use one million recent chess games from the T80 dataset[6], collected from June 13, 2025 to June 14, 2025. These games encompass a wide range of board states and movement sequences, providing a varied basis for training and testing various neural network architectures.

From the one-million-game dataset, 80% of the games are allocated to the training set, while the remaining 20% are reserved for validation. The training and validation sets contain entirely distinct games. Each game of this dataset is composed of many chess positions, corresponding to the training inputs, that we encode with the *classic encoding* of Lc0 (see Section 2).

Thus, our prototype networks are trained on a substantially smaller number of games than production networks which are generally trained on hundreds of millions games, which naturally leads to lower absolute playing strength. Our objective in this paper, however, is not to match production performance but to provide a controlled setting for evaluating the impact of the proposed chess-specific filters.

The training target for each position is a vector of preference scores of size 1858 corresponding to all the possible moves (source square plus destination square), a probability vector of size 3, corresponding to the probabilities of losing, winning and drawing the game when in the current position and the number of remaining moves from the current position. These probabilities were evaluated with the MCTS during the self-played games performed by Lc0.

4.2. Neural network configurations

The neural networks compared in this paper use two different types of blocks as described in Section 2.1.1, namely Residual, or MobileNet blocks. In the MobileNet blocks the *depthwise multiplier* parameter is set to 6 as seen in Figure 3.

For the convolutional blocks used in the trunk of the neural network, we test the following configurations:

- `res_3x3` and `mob_3x3`: Residual and MobileNet blocks with the standard kernel of size 3x3.
- `res_5x5` and `mob_5x5`: Residual and MobileNet blocks with standard kernel of size 5x5.

[6]https://storage.lczero.org/files/training_data/test80/

- `res_rng` and `mob_rng`: Residual and MobileNet blocks with kernels randomly constructed for each convolutional layer by randomly selecting 8 squares that are not in the center of the patch and are assigned the value 1, while the other squares remain at value 0. The center of the patch is always set to 1, to be comparable with other filter types. Each random filter has the same number of 1's as the chess filters shown in Figures 5-7.
- `res_rb` and `mob_rb`: Residual or MobileNet blocks with a combination using only rook and bishop kernels in each convolutional layer. For this version we exclude the knight kernel as it was shown in Goudet et al. (2024) that knight kernel does not always have a positive impact on network training for certain block configurations.
- `res_krb` and `mob_krb`: Residual or MobileNet blocks with the *knight-rook-bishop* kernel combination as depicted in Figures 8 and 9.

In addition to these ten block variants all using the baseline implementation in cuDNN of the convolutional operations (see Section 3.1), we also define two optimized versions:

- `mob_krb_opti1`: MobileNet blocks constructed from combinations of the three knight–rook–bishop chess filters, each employing the first optimized depthwise operator described in Subsection 3.2.
- `mob_krb_opti2`: MobileNet blocks constructed from the same combinations of knight–rook–bishop filters, but using the fully optimized depthwise operator that incorporates all low-level CUDA improvements presented in Subsections 3.2 and 3.3.

Note that the `mob_krb`, `mob_krb_opti1`, and `mob_krb_opti2` models use identical trained weights. The sole distinction between these variants concerns inference efficiency due to the low-level implementation described in Section 3.

For each of the twelve versions, we construct three architectural variants differing in network width and depth. Specifically, the number of convolutional blocks (B) and the number of channels in the trunk (C) are set to $(6, 96)$, $(6, 192)$, and $(18, 96)$, respectively.

4.2.1. Training of the networks

For each configuration of the neural network, we launch a training run on an NVIDIA V100 GPU with 32 GiB of memory during 100,000 steps of gradient descent with a default batch size of 1,024. At each training step, a gradient is calculated to minimize the cross-entropy for the policy head, the cross-entropy for the value head (calculated over 3 outputs) and the MSE for the moves-left head. In order to calculate the loss for the policy head, a legal mask is first applied to calculate only the cross-entropy for legal moves, as is usually the case when training Lc0 networks. We use a stepwise decreasing learning rate which is set at the value of 0.02 during the first 30,000 steps, then set at the value of 0.002 until step 60,000, and finally set at the value of 0.0005 for the remaining steps.

To monitor potential overfitting during training, an evaluation on a held-out validation set is performed every 2,000 gradient steps. The validation set consists of 200,000 games that are completely different from the training dataset. Examples of evolutions of the policy head accuracy and the value head mean squared error (MSE), measured on the validation set throughout training, is provided in Appendix B. These curves are representative of the behavior observed across all model variants: both policy accuracy and value head MSE on the validation set improve consistently over the course of training, with no indication of overfitting.

4.2.2. Computational efficiency

Before assessing the gameplay performance of the different architectures, we first examine their computational efficiency when the same NVIDIA V100 GPU is used. Specifically, we measure both the

inference time and the number of Monte Carlo Tree Search (MCTS) nodes opened per second when the neural network is evaluated on batches of 100 chess positions.[7] The results of these measurements are reported in Table 1 for all twelve architectural variants described in Section 4.2, and for the three trunk configurations $(B, C) = (6, 96), (6, 192), (18, 96)$.

To put all of these models on an equal footing, we use half precision to measure all inference times.

Table 1
Inference average time in millisecond required to process a batch of 100 positions and number of MCTS nodes explored per second (nps) reported for the 12 different AIs all of size (B, C) equal to $(6, 96)$,$(6, 192)$ and $(18, 96)$.

size (B, C)	$(6, 96)$		$(6, 192)$		$(18, 96)$	
Block type	time (ms)	nps	time (ms)	nps	time (ms)	nps
res_3x3	1.42	70556	1.68	59172	3.21	31120
res_5x5	1.65	60520	2.89	34553	3.85	25957
res_rng	1.63	61237	2.86	34992	3.84	26017
res_rb	1.64	60977	2.85	35061	3.89	25686
res_krb	1.64	61150	2.86	34971	3.87	25821
mob_3x3	1.70	58968	3.52	28432	4.43	22587
mob_5x5	2.02	49628	4.16	23987	5.43	18407
mob_rng	2.02	49478	4.17	23969	5.44	18374
mob_rb	2.01	49671	4.16	23993	5.43	18427
mob_krb	2.02	49611	4.17	23947	5.41	18483
mob_krb_opti1	1.41	70908	4.11	24275	3.51	28482
mob_krb_opti2	1.36	73350	2.86	34845	3.52	28353

First, as expected, Table 1 shows that, across all network configurations, the inference time (column "time") required to process a batch of 100 positions increases with network size, while the number of nodes processed per second (column "nps") decreases accordingly.

Furthermore, increasing the kernel size from 3×3 to 5×5 consistently leads to higher computation times for both Residual and MobileNet blocks, regardless of network depth or width. It is important to note that the naive chess-filter implementations (see Section 3.1) are internally computed as 5×5 convolutions in which 16 entries are fixed to zero. This explains why the corresponding chess-filter variants, `res_rng`, `res_rb`, and `res_krb`, exhibit identical computational characteristics to the `res_5x5` architecture. The same observation holds for the MobileNet variants `mob_rng`, `mob_rb`, and `mob_krb`, whose inference times match those of `mob_5x5`.

Second, we observe that architectures based on Residual blocks are consistently faster than their MobileNet-based counterparts with naive implementation. However, the `mob_krb_opti1` architecture, which incorporates the first level of optimization of the depthwise convolution operator presented in Section 3.2, achieves inference times comparable to those of Residual blocks, and in certain cases (e.g., architectures of size $(6, 96)$ and $(18, 96)$) even slightly surpasses them.

It is interesting to notice that, between the `mob_krb` and the `mob_krb_opti2` versions, the observed speed up factor is similar across all architecture sizes : 1.49 for (6,96), 1.46 for (6,192) and 1.54 for (18,96). However, this acceleration is not obtained at the same step of the transition `mob_krb` $\rightarrow$ `mob_krb_opti1` $\rightarrow$ `mob_krb_opti2`. Indeed, for (6,96) and (18,96), the gains in inference speed between `mob_krb` and `mob_krb_opti2` are mostly observed between

[7] In Lc0's MCTS implementation, positions requiring evaluation are collected in a queue and processed in batches to accelerate inference and improve GPU utilization.

mob_krb and mob_krb_opti1. However, for (6,192), the most noticeable acceleration is between mob_krb_opti1 and mob_krb_opti2.

The substantial performance improvement observed when moving from mob_krb to mob_krb _opti1 in narrower architectures can be attributed to the transition from half to half2, which reduces the number of load operations by a factor of two. When the number of channels remains sufficiently small, for instance, 576, the full set of weights occupies only $576 \times 25 \times 2 \approx 28.8$ KB, thereby fitting comfortably within the 64 KB shared memory cache available on modern GPUs. In this regime, the reduction in load operations does not introduce any additional bottleneck, resulting in a clear acceleration.

By contrast, in wider architectures such as $(6, 192)$, where the depthwise layer contains 1152 channels, the dominant limitation arises from the memory footprint rather than from the number of load operations. In this case, storing all 25 weights requires approximately 57.6 KB of cache, which approaches the 64 KB capacity limit and frequently leads to spillover into slower levels of the memory hierarchy. Under such conditions, weight storage becomes more critical than weight loading, which explains the limited performance gains observed when switching to half2. However, once the 16 redundant weights are removed, the model once again fits entirely within the highest-level cache. Consequently, memory storage no longer constitutes the primary bottleneck, thereby allowing cache-level load optimizations to translate into measurable performance improvements.

4.2.3. Round-robin tournaments

We now compare the different architectures, trained on the same dataset, in terms of actual gameplay performance. We made round robin tournaments between the different variant of neural networks in order to compare their level of play.

We first conduct tournaments among architectures sharing the same trunk size (B, C). Each tournament includes 11 of the models described in Section 4.2 We exclude the mob_krb_opti1 variant, which was very often outperformed in inference speed by the mob_krb_opti2 version, as seen in the last subsection.

Every AI plays against each of its 10 opponents using 50 predefined opening positions from a standard opening book, alternating colors to ensure symmetry. Consequently, each pairing consists of 100 games, 50 as White and 50 as Black, resulting in a total of 1,000 games per AI in each tournament. Scoring follows the conventional system: one point for a win, 0.5 points for a draw, and zero points for a loss. The maximum attainable score for any AI in a tournament is therefore 1,000.

The chess tournaments were conducted using the *Fastchess* software[8] under three time-control settings: bullet (60 seconds with a 0.6-second increment per move), blitz (3 minutes with a 2-second increment), and rapid (10 minutes without increment). Each tournament is run using an NVIDIA V100 GPU.

4.3. Results of the tournaments between AI with networks of the same size

The scores obtained by the 11 AI systems in tournaments with the different time controls are reported in Tables 2, 3, and 4, for the architectures whose trunk configurations are defined by the number of blocks and channels (B, C) set to $(6, 96)$, $(6, 192)$, and $(18, 96)$ respectively. The maximum attainable score in each tournament is 1000 points, and the highest score in each case is indicated in bold.

[8]https://github.com/Disservin/fastchess

The estimated Elo ratings and the associated 95% confidence intervals for each AI system are reported in the final two columns of each table. To obtain these values, we first estimated the Elo rating of the best configuration mob_krb_opti2 for each network size: (6,96), (6,192), and (18,96). For each network size, 200 games were played against Stockfish at a fixed Elo rating of 3000, using *Fastchess* with a time control of 120s + 1s.[9] The resulting average Elo ratings for mob_krb_opti2 were 3159, 3204, and 3154 for the (6,96), (6,192), and (18,96) architectures, respectively. The Elo ratings of the other variants were subsequently inferred from the win–loss–draw outcomes obtained in their respective matchups against the mob_krb_opti2 reference variant during the round-robin tournaments conducted under the three time controls.[10]

Across all network sizes and time controls, the variant mob_krb_opti2, which employs MobileNet blocks composed of combinations of three chess-specific filters and integrates the optimized low-level implementation described in Section 3, consistently achieves the best performance. It is clearly better in particular than the standard res_3x3 baseline used in Lc0. Furthermore, the superiority over the mob_krb variant demonstrates the impact of the optimized kernel implementation, as the only distinction between these two versions lies in inference speed, the predictions for identical board positions remain unchanged.

When non-optimized versions are considered, architectures incorporating MobileNet blocks with chess-specific filters, namely mob_krb and mob_rb, consistently outperform other MobileNet variants. This observation suggests that the inclusion of chess-specific filters enhances predictive accuracy, particularly when compared to mob_5x5, which has a larger number of parameters but lacks such filters. Furthermore, these chess-filter-based models consistently surpass the mob_rng variant, which uses random filters, indicating that the improvement is attributable to the structured design of chess-adapted filters rather than random regularization.

A comparison between mob_krb and mob_rb reveals an architecture-dependent trend: the mob_krb variant, which includes knight-rook-bishop filters, performs worse than mob_rb when the architecture size is $(6,96)$ (cf. Table 2), but surpasses it when the number of channels increases to 192 (cf. Table 3). This suggests that a larger channel capacity enables effective utilization of a more diverse filter set, including the knight filter. Conversely, with fewer channels, prioritizing rook and bishop filters appears advantageous, as these correspond to the movement patterns of more pieces (king, queen, pawn, rook, and bishop), whereas the L-shaped knight filter is more specialized.

Finally, the results indicate that combining chess-specific filters with Residual networks provides limited benefit. In fact, the standard res_3x3 variant consistently outperforms other Residual network configurations, including those with chess filters.

4.4. Results of the tournaments between neural networks of different sizes

Tournaments were subsequently organized using the mob_krb_opti2 variant, with competing neural networks configured with different sizes (B,C) set to $(6,96)$, $(6,192)$, and $(18,96)$. For each time control, every architecture played 100 games against each of the other two, following the same experimental protocol described in the previous section. Under this setting, the maximum attainable score per time control was 200 points. The results are summarized in Table 5. The architecture with $(6,192)$ channels consistently outperformed the others, suggesting that increasing the number of channels

[9] This time control corresponds to the setting under which Stockfish's Elo was calibrated on the CCRL 40/15 scale.

[10] Since Elo ratings are inferred only from direct outcomes against the mob_krb_opti2 reference and not from the full round-robin score matrix, their ordering may differ from the global tournament ranking.

Table 2
Scores obtained by the 11 different AIs, all of size $(B,C) = (6,96)$, competing in three different tournaments with time controls bullet (60s + 0.6s), blitz (3 min + 2 s) and rapid (10 min). The maximum score that a configuration can obtain in each tournament is 1000 points. Best score is in bold.

Time control	60+0.6		3 min + 2 s		10min		Total		Estimated Elo	
Block type	Score	Rank	Score	Rank	Score	Rank	Total	Global Rank	Avg.	CI 95%
mob_krb_opti2	**724.0**	1	**690.5**	1	**677.5**	1	**2092.0**	1	3159	[3103, 3215]
mob_rb	637.5	2	639.0	2	626.0	2	1902.5	2	3101	[3061, 3140]
mob_krb	631.0	3	628.0	3	625.5	3	1884.5	3	3110	[3070, 3150]
mob_5x5	571.0	4	532.5	5	579.0	4	1682.5	4	3046	[3005, 3088]
mob_rng	528.5	5	535.0	4	535.5	5	1599.0	5	3050	[3009, 3091]
res_rb	469.0	8	494.0	6	477.5	6	1440.5	6	3016	[2973, 3059]
res_3x3	479.0	7	476.5	7	467.0	8	1422.5	7	3020	[2978, 3063]
res_5x5	489.5	6	462.0	8	470.0	7	1421.5	8	2971	[2926, 3016]
mob_3x3	385.5	9	402.5	9	427.5	9	1215.5	9	2973	[2928, 3018]
res_krb	319.0	10	349.0	10	332.0	10	1000.0	10	2916	[2867, 2966]
res_rng	266.0	11	291.0	11	278.5	11	835.5	11	2879	[2827, 2932]

Table 3
Scores obtained by the 11 different AIs, all of size $(B,C) = (6,192)$, competing in three different tournaments with time controls bullet (60s + 0.6s), blitz (3 min + 2 s) and rapid (10 min). The maximum score that a configuration can obtain in each tournament is 1000 points.

	60s + 0.6s		3 min + 2 s		10min		Total		Estimated Elo	
Block type	Score	Rank	Score	Rank	Score	Rank	Total	Global Rank	Avg.	CI 95%
mob_krb_opti2	**708.5**	1	**684.0**	1	**682.0**	1	**2074.5**	1	3204	[3144, 3264]
mob_krb	616.5	2	604.0	2	621.5	2	1842.0	2	3124	[3083, 3164]
mob_rb	587.5	3	581.0	3	586.0	3	1754.5	3	3143	[3103, 3183]
res_3x3	587.5	4	548.5	4	535.5	4	1671.5	4	3116	[3076, 3157]
res_rb	499.5	5	520.0	5	493.5	5	1513.0	5	3094	[3053, 3135]
mob_5x5	462.0	6	476.5	6	480.5	6	1419.0	6	3058	[3015, 3101]
res_rng	442.0	8	460.0	7	457.0	7	1359.0	7	3047	[3004, 3090]
res_5x5	450.0	7	452.5	8	455.5	8	1358.0	8	3013	[2968, 3059]
res_krb	414.5	9	442.5	9	420.5	9	1277.5	9	3016	[2971, 3061]
mob_3x3	386.0	10	382.0	10	401.5	10	1169.5	10	2989	[2942, 3036]
mob_rng	346.0	11	349.0	11	362.5	11	1057.5	11	3007	[2961, 3053]

yields greater benefits than increasing the number of blocks. While a substantial increase in the number of blocks can improve prediction accuracy, it also incurs higher inference time, thereby reducing the number of nodes evaluated during move selection and ultimately limiting overall performance.

5. CONCLUSION

In this paper, we show that a MobileNet-type architecture augmented with chess-specific convolutional filters and optimized through a low-level CUDA implementation achieves superior performance compared to Residual networks trained on an equivalent number of games. These filters are designed to encode the characteristic movement patterns of chess pieces.

Table 4

Scores obtained by the 11 different AIs, all of size $(B,C) = (18,96)$, competing in three different tournaments with time controls bullet (60s+0.6s), blitz (3 min + 2 s) and rapid (10 min). The maximum score that a configuration can obtain in each tournament is 1000 points.

	60s+0.6s		3 min + 2 s		10min		Total		Estimated Elo	
Block type	Score	Rank	Score	Rank	Score	Rank	Total	Global Rank	Avg.	CI 95%
mob_krb_opti2	**764.5**	1	**737.5**	1	**723.5**	1	**2225.5**	1	3154	[3098,3210]
mob_krb	625.5	2	602.5	3	628.5	2	1856.5	2	3071	[3031, 3112]
mob_rb	592.5	3	608.5	2	600.0	3	1801.0	3	3037	[2996, 3079]
mob_3x3	555.5	4	554.0	4	563.5	4	1673.0	4	3003	[2960, 3046]
mob_rng	520.5	5	508.5	5	515.5	5	1544.5	5	2972	[2927, 3017]
res_3x3	509.0	6	501.0	6	497.5	6	1507.5	6	2981	[2937, 3025]
res_rb	462.5	7	482.5	7	485.5	7	1430.5	7	2971	[2926, 3016]
mob_5x5	410.0	8	415.0	8	410.0	8	1235.0	8	2926	[2877, 2974]
res_5x5	379.5	9	383.0	9	371.0	9	1133.5	9	2911	[2862, 2961]
res_rng	349.5	10	359.0	10	350.0	10	1058.5	10	2893	[2842, 2944]
res_krb	331.0	11	348.5	11	355.0	11	1034.5	11	2896	[2846, 2947]

Table 5

Scores obtained by the mob_krb_opti2 version for network sizes for different speeds: bullet (60s + 0.6s), blitz (3min + 2s), and rapid (10 min). The maximum score that a configuration can obtain in each tournament is 200 points.

mob_krb_opti2	60s+0.6s	Rank	3min+2s	Rank	10min	Rank	Total	Global Rank	Estimated Elo
Size (6,192)	**120.5**	1	**124.5**	1	**120.0**	1	**365.0**	1	3204
Size (18,96)	90.0	2	94.5	2	90.5	2	275.0	2	3154
Size (6,96)	89.5	3	81.0	3	89.5	3	260.0	3	3159

These results, however, are established within the small-network regime, where model capacity remains relatively limited. Further empirical studies will be required to determine whether the observed performance gains extend to larger neural architectures.

Despite this small-network regime and limited training data, the best-performing variant of our network achieves an Elo rating of approximately 3200 on the CCRL 40/15 scale. This result suggests that the proposed chess-specific filters could be effectively leveraged within larger architectures trained for longer durations on substantially larger datasets, which constitutes a promising direction for future work.

Finally, the consistent performance advantage observed when comparing architectures equipped with chess filters to those using random filters indicates that the gains cannot be attributed to random regularization effects alone. Instead, they stem from the introduction of an inductive bias that facilitates the extraction of relevant spatial features and supports more effective inference from chessboard positions.

There are several perspectives for this work. First, the current low-level implementation of the MobileNet block relies on successive convolution operations, each requiring access to global memory. We have identified that these operations are particularly detrimental to the performance of the MobileNet network during inference and therefore during the AI's reflection time to select the move to play. Building on the work of Qararyah et al. (2024), it may be possible to merge the depthwise and pointwise convolution operations into a single operation in order to minimize access to global memory and thus further increase the speed of the architecture we propose.

A second research direction would be the development of an EfficientFormer-type architecture, which has already demonstrated strong performance in the context of Go (Sagri et al., 2024). This approach would involve coupling our newly proposed convolutional blocks equipped with chess-specific filters, used at the early stages of the network trunk to extract low-level features from the board, followed by transformer-based encoder blocks, such as those introduced for computer chess in Monroe and Chalmers (2024). An alternative avenue would be to integrate the chess filters more directly into the attention mechanism itself, for instance by incorporating them into the attention layers in a manner analogous to causal masks. Such a design could further facilitate inference from chess positions by embedding domain-specific structural priors directly within the attention computation.

6. ACKNOWLEDGEMENTS

This work was granted access to the HPC resources of IDRIS under the allocation 2025-AD010611887R3 made by GENCI.

REFERENCES

Campbell, M., Hoane Jr, A.J. & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, *134*(1-2), 57–83.

Goudet, O., Joshi, B. & Cazenave, T. (2024). Convolutional Neural Networks with Specific Kernels for Computer Chess. In *International Conference on Computers and Games* (pp. 14–24). Springer.

He, K., Zhang, X., Ren, S. & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

Klein, D. (2022). Neural networks for chess. *arXiv preprint arXiv:2209.01506*.

LCZero, D.T. (2018). Leela Chess Zero (LCZero). Available at https://lczero.org.

Li, Y., Yuan, G., Wen, Y., Hu, J., Evangelidis, G., Tulyakov, S., Wang, Y. & Ren, J. (2022). Efficientformer: Vision transformers at mobilenet speed. *Advances in neural information processing systems*, *35*, 12934–12949.

Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T. & Xie, S. (2022). A convnet for the 2020s. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 11976–11986).

Monroe, D. & Chalmers, P.A. (2024). Mastering chess with a transformer model. *arXiv preprint arXiv:2409.12272*.

Qararyah, F., Azhar, M.W., Maleki, M.A. & Trancoso, P. (2024). Fusing depthwise and pointwise convolutions for efficient inference on GPUs. In *Workshop Proceedings of the 53rd International Conference on Parallel Processing* (pp. 58–67).

Sagri, A., Arjonilla, J., Saffidine, A. & Cazenave, T. (2024). Vision Transformers for Computer Go. In *EvoApps*. Springer.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. & Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4510–4520).

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609.

Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, *41*, 256–275.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *nature*, *550*(7676), 354–359.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*(6419), 1140–1144.

APPENDIX A. SOURCE CODE OF THE OPTIMIZED DEPTHWISE CONVOLUTION WITH KNIGHT-ROOK-BISHOP FILTERS

In this appendix we give the code of our new optimized low level implementation of the depthwise convolution layer with the combination of the knight-rook-bishop filters, described in Section 3. For more details, this source code is also available at https://github.com/LEAHPARAPHAEL/lc0.

```
1  __global__ void DepthwiseKernel(int C_in,
2                                  half* output,
3                                  const half2* input,
4                                  const half2* weights,
5                                  const half2* biases){
6
7  #if __CUDA_ARCH__ >= 700
8
9      const int block_depth = C_in / (2 * PARALLEL_BLOCKS);
10     const int thread_depth = block_depth / DW_PARALLEL_D;
11
12     const int thread_w = threadIdx.x;
13     const int thread_h = threadIdx.y;
14     const int thread_d = threadIdx.z;
15
16     const int block_n = blockIdx.x;
17     const int block_d = blockIdx.y;
18     const int block_h = blockIdx.z;
19
20     // Absolute horizontal position of the thread
21     const int abs_w = thread_w;
22
23     // Absolute channel number of the beginning of the thread
24     const int abs_d = block_depth * block_d +
25         thread_depth * thread_d;
26
27     // Absolute vertical position of the thread
28     const int abs_h = block_h * DW_BLOCK_H + thread_h;
29
30     // The nine depthwise weights and the bias to be loaded
31     half2 w0, w1, w2, w3, w4, w5, w6, w7, w8, b;
32
33     // Loops over the channels covered by the thread executing the
        ↪  kernel.
34     #pragma unroll
35     for (int c = 0; c < thread_depth ; c+=1){
36
37         // Current channel (beginning of the thread + offset)
38         const int current_d = abs_d + c;
39
```

```
40          // Used to share the 9 weights among the threads of the
             ↪  same warp
41          half2 shared_weight;
42
43          /*
44            Each thread fetches at most one weight and shares it with
               ↪  the warp.
45            For the sake of simplicity, the 9 first threads fetch the
               ↪  9 weights and the 10th gets the bias.
46          */
47          if (thread_h * DW_BLOCK_W + thread_w < 9){
48              shared_weight = weights[
49                  current_d * 9 + thread_h * DW_BLOCK_W + thread_w
50              ];
51          }
52          if (thread_h * DW_BLOCK_W + thread_w == 9){
53              shared_weight = biases[current_d];
54          }
55
56
57          /*
58            The threads of the warp share the 9 weights and the bias
               ↪  using registers.
59            For example, the first weight (w0) was fetched by the
               ↪  first thread of the warp
60            (the thread at index 0), so the last argument of the
               ↪  __shfl_sync function is 0.
61          */
62
63          unsigned active_threads_mask = __activemask();
64
65          w0 = __shfl_sync(active_threads_mask, shared_weight, 0);
66          w1 = __shfl_sync(active_threads_mask, shared_weight, 1);
67          w2 = __shfl_sync(active_threads_mask, shared_weight, 2);
68          w3 = __shfl_sync(active_threads_mask, shared_weight, 3);
69          w4 = __shfl_sync(active_threads_mask, shared_weight, 4);
70          w5 = __shfl_sync(active_threads_mask, shared_weight, 5);
71          w6 = __shfl_sync(active_threads_mask, shared_weight, 6);
72          w7 = __shfl_sync(active_threads_mask, shared_weight, 7);
73          w8 = __shfl_sync(active_threads_mask, shared_weight, 8);
74          b = __shfl_sync(active_threads_mask, shared_weight, 9);
75
76
77          /*
78            Batch + channel index :
79            - misses only the height and width offsets to have
80              the final index.
```

```
81            - counts with packed channels, hence the C_in / 2, so
               ↪  will have to be doubled when
82              going back to half format for the output.
83          */
84
85          const int offset_nc = (block_n * C_in / 2 + current_d) *
86              64;
87
88
89
90          // Row in the 8 x 8 input of the channel : substract 2 for
             ↪  top padding
91          const int abs_h_input = block_h * DW_BLOCK_H + thread_h -
92              2;
93
94          // Column in the 8 x 8 input of the channel : substract 2
             ↪  for left padding
95          const int abs_w_input = thread_w - 2;
96
97          const int index_input = offset_nc + abs_h_input * 8 +
98              abs_w_input;
99
100         // Accumulator
101         half2 sum = make_half2(0.0f, 0.0f);
102
103
104         // Knight filter
105         if (2 * current_d < (C_in / 3)){
106             sum = __hfma2(w0, get_input_half2_at(input,
107                 abs_h_input, abs_w_input + 1, index_input + 1),
108                 sum);
109             sum = __hfma2(w1, get_input_half2_at(input,
110                 abs_h_input, abs_w_input + 3, index_input + 3),
111                 sum);
112             sum = __hfma2(w2, get_input_half2_at(input,
113                 abs_h_input + 1, abs_w_input, index_input + 8),
114                 sum);
115             sum = __hfma2(w3, get_input_half2_at(input,
116                 abs_h_input + 1, abs_w_input + 4, index_input +
117                 12), sum);
118             sum = __hfma2(w4, get_input_half2_at(input,
119                 abs_h_input + 2, abs_w_input + 2, index_input +
120                 18), sum);
121             sum = __hfma2(w5, get_input_half2_at(input,
122                 abs_h_input + 3, abs_w_input, index_input +
123                 24), sum);
124             sum = __hfma2(w6, get_input_half2_at(input,
125                 abs_h_input + 3, abs_w_input + 4, index_input +
```

```
126                 28), sum);
127             sum = __hfma2(w7, get_input_half2_at(input,
128                 abs_h_input + 4, abs_w_input + 1, index_input +
129                 33), sum);
130             sum = __hfma2(w8, get_input_half2_at(input,
131                 abs_h_input + 4, abs_w_input + 3, index_input +
132                 35), sum);
133         }
134 
135         // Rook filter
136         else if (2 * current_d < (2 * C_in / 3)) {
137             sum = __hfma2(w0, get_input_half2_at(input,
138                 abs_h_input, abs_w_input + 2, index_input + 2),
139                 sum);
140             sum = __hfma2(w1, get_input_half2_at(input,
141                 abs_h_input + 1, abs_w_input + 2, index_input +
                  ↪  10),
142                 sum);
143             sum = __hfma2(w2, get_input_half2_at(input,
144                 abs_h_input + 2, abs_w_input, index_input + 16),
145                 sum);
146             sum = __hfma2(w3, get_input_half2_at(input,
147                 abs_h_input + 2, abs_w_input + 1, index_input +
148                 17), sum);
149             sum = __hfma2(w4, get_input_half2_at(input,
150                 abs_h_input + 2, abs_w_input + 2, index_input +
151                 18), sum);
152             sum = __hfma2(w5, get_input_half2_at(input,
153                 abs_h_input + 2, abs_w_input + 3, index_input +
154                 19), sum);
155             sum = __hfma2(w6, get_input_half2_at(input,
156                 abs_h_input + 2, abs_w_input + 4, index_input +
157                 20), sum);
158             sum = __hfma2(w7, get_input_half2_at(input,
159                 abs_h_input + 3, abs_w_input + 2, index_input +
160                 26), sum);
161             sum = __hfma2(w8, get_input_half2_at(input,
162                 abs_h_input + 4, abs_w_input + 2, index_input +
163                 34), sum);
164         }
165 
166         // Bishop filter
167         else {
168             sum = __hfma2(w0, get_input_half2_at(input,
169                 abs_h_input, abs_w_input, index_input),
170                 sum);
171             sum = __hfma2(w1, get_input_half2_at(input,
172                 abs_h_input, abs_w_input + 4, index_input + 4),
```

```
173                     sum);
174                 sum = __hfma2(w2, get_input_half2_at(input,
175                     abs_h_input + 1, abs_w_input + 1, index_input + 9),
176                     sum);
177                 sum = __hfma2(w3, get_input_half2_at(input,
178                     abs_h_input + 1, abs_w_input + 3, index_input +
179                     11), sum);
180                 sum = __hfma2(w4, get_input_half2_at(input,
181                     abs_h_input + 2, abs_w_input + 2, index_input +
182                     18), sum);
183                 sum = __hfma2(w5, get_input_half2_at(input,
184                     abs_h_input + 3, abs_w_input + 1, index_input +
185                     25), sum);
186                 sum = __hfma2(w6, get_input_half2_at(input,
187                     abs_h_input + 3, abs_w_input + 3, index_input +
188                     27), sum);
189                 sum = __hfma2(w7, get_input_half2_at(input,
190                     abs_h_input + 4, abs_w_input, index_input +
191                     32), sum);
192                 sum = __hfma2(w8, get_input_half2_at(input,
193                     abs_h_input + 4, abs_w_input + 4, index_input +
194                     36), sum);
195             }
196 
197             // Adds the bias to the sum
198             sum = __hadd2(sum, b);
199 
200             // ReLU !
201             sum = __hmax2(sum, __float2half2_rn(0.0f));
202 
203             /*
204               Writes the result in the output, splitting the two halves
                   ↪  of the sums, which correspond to the channels
205               at position 2*current_d and 2*current_d + 1, hence the 64
                   ↪  offset.
206             */
207             output[2*offset_nc + abs_h*8 + abs_w] = __low2half(sum);
208             output[2*offset_nc + 64 + abs_h*8 + abs_w] =
209                 __high2half(sum);
210 
211         }
212 
213 #endif
214 }
215 
216 
217 
218 void DepthwiseEval(int N,
```

```
219                         int C_in,
220                         half* output,
221                         const half* input,
222                         void* scratch,
223                         const half2* w1,
224                         const half2* b1,
225                         cudaStream_t stream){
226
227     convert_half_to_half2_nchw(
228         input, (half2*)scratch, N, C_in, 8, 8
229     );
230
231     dim3 threads(DW_BLOCK_W, DW_BLOCK_H, DW_PARALLEL_D);
232
233     dim3 blocks(N, PARALLEL_BLOCKS, 8 / DW_BLOCK_H);
234     DepthwiseKernel<<<blocks, threads>>>(
235         C_in, output, (half2*)scratch, w1, b1
236     );
237 }
238
239
```

APPENDIX B. EXAMPLES OF VALIDATION PERFORMANCE EVOLUTION DURING THE TRAINING PROCESSES.

Figures 13 and 14 respectively display the evolution of the policy accuracy and the value head MSE on the validation set during the training of the MobileNet architecture with chess filter (`mob_krb` variant) with different sizes of network : (6,96), (6,192) and (18,96). This highlights that no overfitting behaviour is observed, and that the larger networks (6,192) and (18,96) achieve better performance than the smaller (6,96) network.

Figures 15 and 16 show these same curves for the `mob_krb` variant with chess filters and `mob_5x5` variant without chess filters. Both networks have size (6,192). It highlights that the variant with chess filters attains better performance on the validation set at the end of training, which we attribute to the inductive bias introduced by these filters.

Note that the curves exhibit a change in behavior before and after 30,000 training steps. Prior to this point, the learning dynamics are faster but more irregular, whereas after 30,000 steps the evolution becomes smoother. This transition corresponds to the learning-rate reduction applied at 30,000 steps.

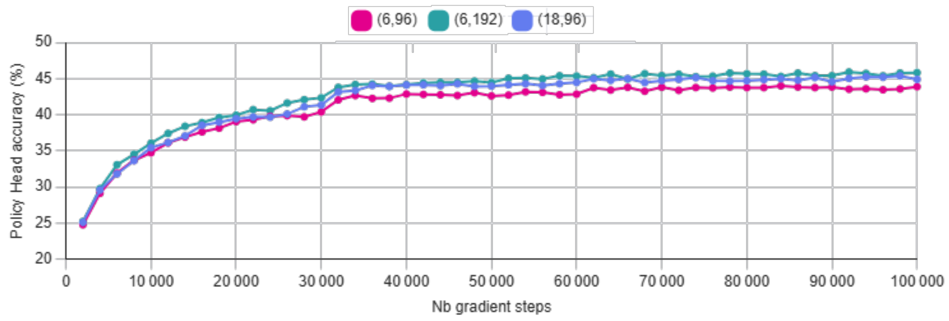

Fig. 13. Evolution of the policy accuracy on the validation set during training of the `mob_krb` variant with sizes (6,96), (6,192) and (18,96).

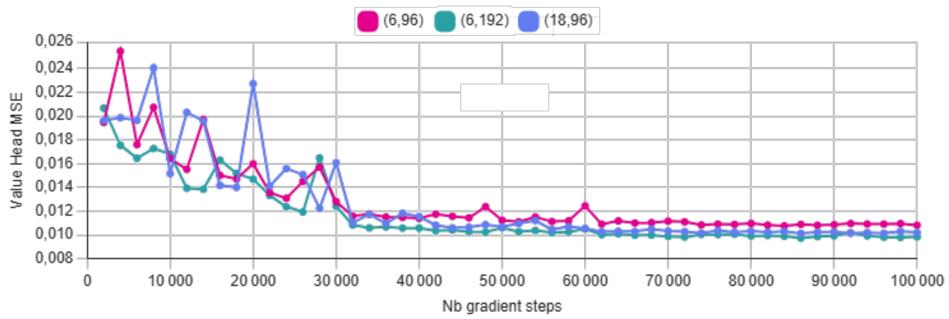

Fig. 14. Evolution of the value head MSE on the validation set during training of the `mob_krb` variant with sizes (6,96), (6,192) and (18,96).

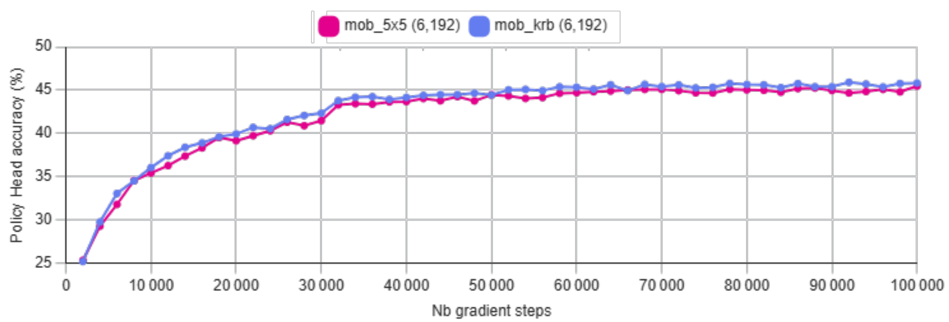

Fig. 15. Evolution of the policy accuracy on the validation set during training of the `mob_krb` variant with chess filters and `mob_5x5` variant without chess filters. Both networks have size (6,192).

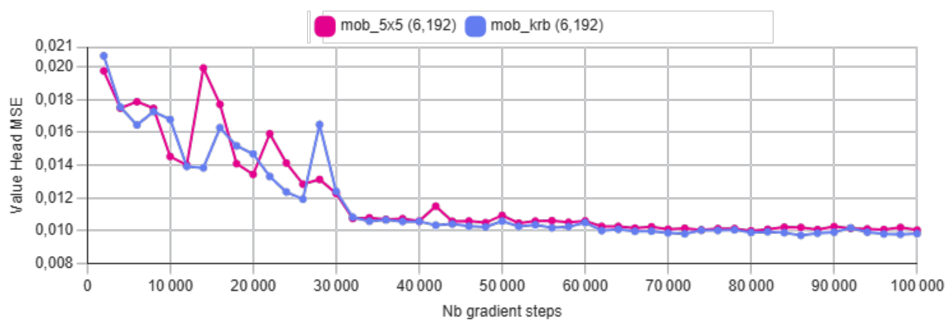

Fig. 16. Evolution of the value head MSE on the validation set during training of the `mob_krb` variant with chess filters and `mob_5x5` variant without chess filters. Both networks have size (6,192).