Convolutional Neural Networks with Specific Kernels for Computer Chess

Olivier Goudet[1], Bhaskar Joshi[2], and Tristan Cazenave[2]

[1] LERIA, Université d'Angers, 2 Boulevard Lavoisier, Angers 49045, France
[2] LAMSADE, Université Paris Dauphine - PSL, CNRS, Paris, France

Abstract. We present the use of chess filters for the convolutional layers used in computer chess. We compare different types of blocks with and without chess filters. Our comparison uses the Leela Chess Zero (Lc0) T60 dataset to train the networks with supervised learning.

1 Introduction

The game of chess has long been a benchmark for artificial intelligence (AI), offering a well-defined yet highly complex environment where strategic decision-making is paramount. The advent of computer chess programs marked significant milestones in AI, with early achievements driven by brute-force search algorithms and hand-crafted evaluation functions [8]. However, the limitations of these traditional approaches became evident as the depth of required computations grew exponentially, prompting the exploration of more sophisticated techniques [1].

In recent years, the integration of neural networks into chess engines has revolutionized the field. Notable among these advancements is the development of deep reinforcement learning frameworks, such as AlphaZero, which combine neural networks with Monte Carlo Tree Search (MCTS) to achieve superhuman performance in chess [10, 11]. These approaches have demonstrated that neural networks can learn intricate strategies and generalize across a vast array of positions without relying on domain-specific knowledge [7]. The current state-of-the-art in computer chess is to use Residual Networks (ResNets) [2], enabling deep architectures, which have shown remarkable success in improving the accuracy of move predictions and value estimations.

Despite these advances, there remains a need for a comprehensive analysis that compares the performance of different neural network architectures within the domain of computer chess. This study aims to fill this gap by systematically evaluating the effectiveness of various convolutional neural network models, such as ResNets, but also more recent architectures such as MobileNet [6] and ConvNeXt [5] inspired by recent breakthroughs in other AI domains. In this paper, we utilize the T60 dataset, a robust collection of self-play games generated by the Leela Chess Zero (Lc0) chess engine [12], to train and evaluate multiple neural network architectures. We focus on key performance metrics such as latency of network, memory, accuracy and MSE loss to determine which architectures offer the most promise for future developments in computer chess.

The remainder of this paper is structured as follows: Section 2 describes the different types of blocks for the trunk of the neural network that we compare, including the specific chess filters that we use in the convolutional layers. Section 3 presents our experimental results, and Section 4 concludes the paper with a summary of our contributions and suggestions for future research.

2 Neural network architectures

During training and evaluation, we use the *classic encoding* of Lc0. Each chess position is converted into a tensor input for the neural network which consists of 112 planes of size 8×8. As explain more in detail in [4], the first 6 planes encode the position of the pieces of the player whose turn it is (one plane for each type of piece). The next 6 planes encodes the positions of the pieces for the opponent. Plane 12 is set to all 1 if one or more repetitions have taken place. These 13 planes are repeated to encode not only the current position, but also the seven previous chess positions of the game. The last 8 planes encode further information, such as whether each color has the right to castle on queen's or king's side.

After the encoding step, this tensor input for the neural network of size $(112, 8, 8)$ is then processed like an image with 112 color channels by a chain of blocks using convolutional layers in the trunk of the neural network. After these blocks, the output is fed into two heads, called the policy head and the value head (see Section 3.2 below).

2.1 Different type of blocks for the trunk

In this subsection, we describe the three types of blocks for the neural network trunk that we have compared in this work: residual block, MobileNet block and ConvNeXt block.

Residual block Residual blocks [2] are a widely used architecture in computer chess, due to their ability to train deep networks without suffering from the vanishing gradient problem. Each residual block incorporates two standard convolutional layers for feature extraction, followed by a *squeeze and excitation* (SE) layer [3]. The input is then added back to the output of this SE layer, forming the residual connection.

MobileNet block MobileNet neural networks [6] were designed for efficient deep learning models, particularly in resource-constrained environments. This block begins with a pointwise convolutional layer, which increases the size of the number of channels by a factor called *depthwise multiplier*. The next stage is a depthwise convolutional layer, which processes each input channel independently. Each of these two first steps are followed by a batch normalization (BN) layer and the application of a ReLU activation function. Finally, the last layer applies a second pointwise convolution operation that restores the original dimensionality of the output channels, followed by BN and SE layers. Lastly the input is added to the output like in residual blocks.

ConvNeXt block ConvNeXt neural networks [5] represent a modern evolution of traditional convolutional neural networks, offering superior performance in certain scenarios. The ConvNeXt block begins with a depthwise convolution layer, which is followed by a normalization layer to ensure stability during training. Next, a pointwise convolutional layer with GELU activation function is applied. It expands the number of channels by a factor of four. A second pointwise convolutional layer (with linear activation) is used to retrieve the number of channels of the input. Finally, the block includes a residual connection as in the other two block types.

2.2 Chess filters for convolutional layers

In this work, we propose some variants of the three types of blocks described in the previous section, using specific chess kernels in the standard and depthwise convolutional layers. The underlying idea is to help the neural network extract relevant patterns in chess positions related to the movement of different types of pieces in the chess game. This idea has certain similarities with the concept of local shape features used for the game of Go in [9], with the difference that in our case, different predefined masks are used to guide the gradient descent towards a better parameterization of the neural network (like a regularization tool), rather than focusing on extracting features that can be directly used at inference time.

Figures 1-3 represent three types of masks with 0 and 1 values that can be applied to a convolutional filter of size $(5, 5)$ to retain only those filter parameters corresponding to the most important types of movement in chess: vertical, horizontal, diagonal and knight moves. Any $(5, 5)$ filter parameters that do not correspond to a square with a piece symbol in these masks are set to 0.

Fig. 1: Mask for knight kernel.

Fig. 2: Mask for rook kernel.

Fig. 3: Mask for bishop kernel.

These three types of *chess filters* can be combined together to build new convolutional layers. An example of depthwise convolutional layer with chess filters is depicted in Figure 4. We see on this figure that the first third of input channels are processed using knight filters, the second third using rook filters and the final third using bishop filters (version called *knight-rook-bishop*). We

also introduce an other version with one half of the channels processed by rook filters and the other half by bishop filters (version called *rook-bishop*). We have experimentally observed that using different types of chess kernels combined together instead of a single type of filter in the same convolutional layer yields better results.

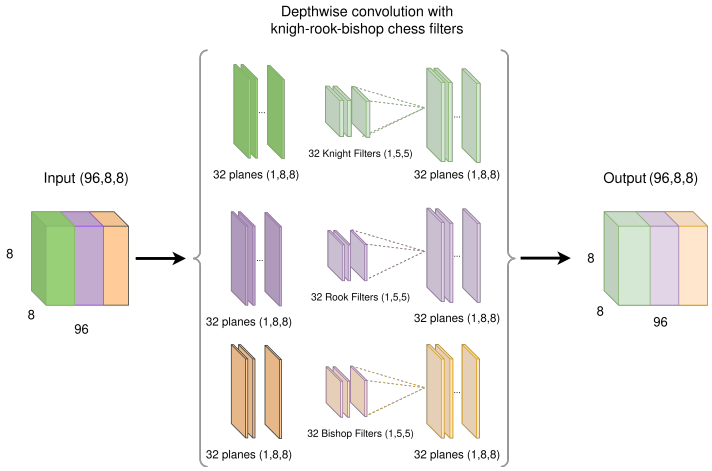

Fig. 4: Depthwise convolution with knight-rook-bishop chess filters applied to input tensor of size $(96, 8, 8)$.

3 Experimental Results

The aim of this section is to answer two questions experimentally. The first concerns the comparison of different block types on neural network performance. The second question concerns the impact of chess filters in combination with each type of block used in the neural network.

3.1 Training and validation test sets

In this paper, we use one million chess games from the T60 dataset[3] which is a classical dataset used to train Lc0 neural networks. Collected over a period from July 26, 2019, to January 8, 2022, the T60 dataset comprises millions of self-play games generated by Lc0. 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 this data set of one million games, we retain 80% for a training dataset and the remaining 20% for the validation dataset. Each game of this dataset is

[3] `https://storage.lczero.org/files/training_data/test60/`

composed of many chess positions, corresponding to the training inputs, that we encode with the *classic encoding* of Lc0 (see Section 2).

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

3.2 Experimental settings

The neural networks that we compare in this paper use different type of blocks as described in Section 2: residual, MobileNet or ConvNeXt blocks. With each type of block we build architectures with 6, 12 and 18 blocks, using Lc0's open-source training code[4] as a starting point. For each number of blocks, denoted nb_{blocks}, we use $nb_{filters} \in \{96, 192\}$ filters in the trunk. In the MobileNet blocks the *dephtwise multiplier* parameter is set to 6. For all the standard and depthwise convolutional layers used in this work we use kernels of size $(5, 5)$ (with the exception of the pointwise convolutional layers used in the mobile net and ConvNeXt architectures) .

On the top of each network trunk, the same two heads are used for each neural network configuration:

- the policy head is the *classical policy head* of Lc0. It consists in a pointwise convolutional layer with 32 output channels, used to convert the output of the trunk of size $(nb_{filters}, 8, 8)$ into a tensor of size $(32, 8, 8)$. This tensor is then flattened and processed with a dense layer with a softmax activation function to obtain a vector of size 1858 corresponding to the probabilities for all the possible moves.
- the value head also consists in a pointwise convolutional layer with 32 output channels, but 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.

Training of the networks For each configuration of the neural network, we launch 5 independent training runs on a V100 Nvidia graphic card with 32 GiB of memory during 100,000 steps of gradient descent with a default batch size of 1,024. We use a reduced batch size of size 512 for the biggest architectures (when $nb_{filters} = 192$, with $nb_{blocks} = 12$ or $nb_{blocks} = 18$), in order to reduce the memory required on the GPU card during training.

At each training step, a gradient is calculated to minimize the cross-entropy for the policy head in addition to the cross-entropy for the value head (calculated over 3 outputs). In order to calculate the loss for the policy head, a legal mask

[4] `https://github.com/LeelaChessZero/lczero-training`

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.

Every 2,000 training steps, the neural network is evaluated on the validation set. On all the position extracts from the 200,000 games in the validation set, we calculate two metrics:

- the policy's average precision, which consists in evaluating the percentage of times when the movement associated with the highest probability calculated with the policy head corresponds exactly to the movement with the highest probability in the target (this is the move which was selected during the Lc0 games after applying MCTS).
- the average MSE loss corresponding to the average mean square error loss between the converted z_i output of the value head and the scalar v_i target value for each position of the validation set. For a position i, from the vector of probability $(p_i^{win}, p_i^{draw}, p_i^{loss})$ given by the value head, the scalar value z_i is computed as $z_i = p_i^{win} - p_i^{loss}$.

3.3 Network features

Table 1 displays different characteristics of the networks we compare in this paper: the number of trainable parameters in millions, the memory required to process a batch of 1024 chess positions (each position is a tensor of size $(112, 8, 8)$ as seen in Section 2) and the latency, or time in seconds required to process this batch of size 1024.

We can see from this table that, with the same number of blocks and the same number of filters, the architectures with residual blocks have the highest number of parameters. This high number of parameters comes mainly from the number of weights in the convolutional kernels of size 5 by 5 used in the residual blocks. We see that when applying the chess masks displayed in Figures 1-3, which are broadcast according to the depth of each kernel, that the number of trainable parameters is drastically reduced. Indeed, the application of chess filters reduces the number of parameters in each convolutional kernel with a ratio of $9/25$.

Secondly, we observe on this table, that the architectures using the MobileNet blocks with the depthwise convolution operations have less parameters than the architectures with the residual blocks, even when using a depthwise multiplier of 6. This is due to the reduced size of the convolution kernels used in depthwise convolution layers, which have a depth of just one.

Thirdly, we find that ConvNeXt architectures have the smallest number of parameters. Its latency is high due to the rather slow normalization layer and quadruple expansion of the number of channels with the pointwise convolution used in each block.

Table 1: Number of trainable parameters (in millions), memory (in GiB) and inference time (in seconds) required to process a batch of 1024 chess positions with different neural network architectures.

Residual net					
Nb blocks	Nb filters	Nb params (M.)		Memory (GiB)	Latency (s)
		Standard	Chess filters		
6	96	7.195	5.426	2.357	0.0940
12	96	10.046	6.507	2.357	0.108
18	96	12.897	7.589	2.357	0.126
6	192	16.017	8.939	4.405	0.121
12	192	27.414	13.258	4.405	0.165
18	192	38.811	17.578	4.405	0.211
Mobile net					
Nb blocks	Nb filters	Nb params (M.)		Memory (GiB)	Latency (s)
		Standard	Chess filters		
6	96	4.921	4.865	3.637	0.103
12	96	5.755	5.645	3.637	0.133
18	96	6.590	6.424	3.637	0.172
6	192	7.265	7.154	6.965	0.144
12	192	10.427	10.206	6.965	0.218
18	192	13.589	13.257	6.965	0.294
ConvNeXt					
Nb blocks	Nb filters	Nb params (M.)		Memory (GiB)	Latency (s)
		Standard	Chess filters		
6	96	4.546	4.537	2.639	0.137
12	96	5.006	4.987	2.639	0.203
18	96	5.466	5.438	2.639	0.268
6	192	5.907	5.889	8.007	0.210
12	192	7.712	7.675	8.007	0.344
18	192	9.516	9.460	8.007	0.479

3.4 Results on the validation set

Figure 5 displays the average evolution (over 5 runs) of the policy accuracy and the MSE loss computed on the validation set every 2,000 steps of training for all the different architectures with 18 blocks and 96 filters in the trunk, with chess filters and without chess filters (versions called "standard"). For residual network architectures, we use the *rook-bishop* version of the chess filters, while for MobileNet and ConvNeXt architectures, we use the *knight-rook-bishop* version. These are the versions that work best for each of these block types, as we will see in more detail in the next subsection. The range of colors around each average curve corresponds to a spread of plus one standard deviation and minus one standard deviation from the average score.

We first see in these figures that for all the architectures there is a huge gap for both metrics when we reach the step 30,000. It corresponds to the first change

in learning rate. Next, we observe that the use of chess filters always improves the results for each type of block. This is interesting, as it shows that better results can be obtained with fewer trainable parameters in the various neural networks. (cf. Table 1). It seems that these filters act as a kind of regularizer well suited to chess positions, helping the neural network to extract relevant features related to the movement of the pieces.

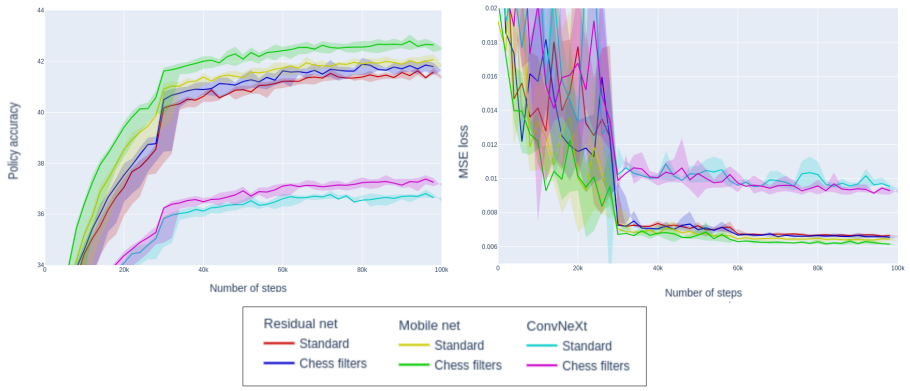

Fig. 5: Policy accuracy and MSE loss on the validation set for architectures with 18 blocks and 96 filters in the trunk.

Table 2 displays a comparison of the average results obtain by the different architectures on the validation set at the end of the training process (after 100,000 steps). From this table we draw the same conclusion regarding the impact of the chess filters. We see that using chess filters almost always improve the results in term of accuracy and MSE loss for each network configuration, but the impact is actually really significant for the network using depthwise convolution operations (MobileNet and ConvNext), or residual nets but with a high number of blocks and high number of filters in the trunk.

3.5 Impact of different chess filters

In this section, we propose a more in-depth analysis of the impact of different versions of chess filters, particularly in comparison with random filters.

Random filters correspond to masks of size $(5, 5)$ 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 display in Figures 1-3.

Figure 6 on the left displays the average evolution of the policy accuracy on the validation set for the residual networks with always 18 blocks and 96 filters

Table 2: Average accuracy scores and MSE loss on the validation set obtain after 100,000 training steps for different neural network architectures. The best results are in bold. Significantly better results for a version with chess filters in comparison with the corresponding standard version are indicated with stars. The stars indicate the results of t-tests with p-value 0.05 (*), 0.01 (**) and 0.001 (***).

Accuracy

Nb blocks	Nb filters	Residual net		Mobile net		ConvNeXt	
		Standard	Chess filters	Standard	Chess filters	Standard	Chess filters
6	96	40.98	41.01	40.90	**41.53****	36.28	37.48***
12	96	41.32	41.60*	41.73	**42.16***	36.54	37.42***
18	96	41.58	41.79	42.07	**42.65*****	36.64	37.25***
6	192	42.55	42.74	42.29	**42.87****	37.86	39.38***
12	192	**42.75**	42.42	42.26	42.72**	37.36	38.19***
18	192	41.97	42.54***	42.39	**43.11*****	37.54	38.26***

MSE loss

Nb blocks	Nb filters	Residual net		Mobile net		ConvNeXt	
		Standard	Chess filters	Standard	Chess filters	Standard	Chess filters
6	96	0.00702	0.00705	0.00682	**0.00673**	0.00963	0.00912**
12	96	0.00670	0.00668	0.00657	**0.00631**	0.00952	0.00917***
18	96	0.00665	0.00654	0.00642	**0.00613*****	0.00953	0.00927**
6	192	0.00616	0.00607	0.00609	**0.00596***	0.00869	0.00812***
12	192	**0.00591**	0.00620	0.00616	0.00608	0.00898	0.00865**
18	192	0.00625	0.00611*	0.00607	**0.00593***	0.00906	0.00857***

in the trunk, but using different types of filters in the convolutional layers used in the blocks. We compare four different versions with kernels of size $(5, 5)$: the red line corresponds to the standard residual net without filters, the blue line corresponds to the application of random filters, the yellow line to *rook-bishop* chess filters and the green line to *knight-rook-bishop* chess filters.

The graph on the left of Figure 6 shows that using a combination of rook and bishop filters that take into account only vertical, horizontal and diagonal moves gives better results than other filter types for residual networks. Using the knight filter as a complement to the rook and bishop filters seems to be more useful with the depthwise convolution layer used in MobileNets, as shown in Figure 6 on the right. As the number of planes in the MobileNet blocks with depthwise multiplier of 6 is really much higher than for residual blocks, and the different planes are processed independently by the different chess filters, this seems to allow a wider variety of filters to be used in combination to improve results.

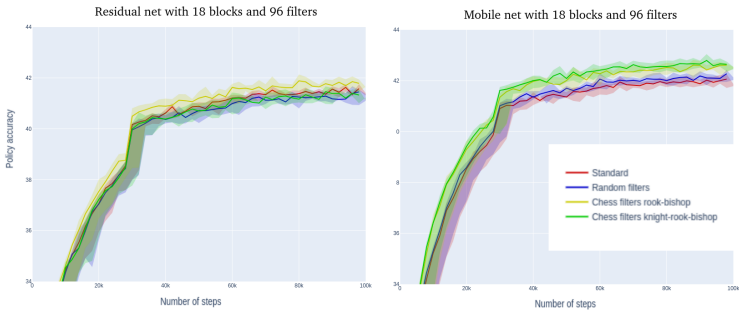

Fig. 6: Evolution of the policy accuracy on the validation set for the residual net (left) and MobileNet (right) with 18 blocks and 96 filters in the trunk and different types of filters.

4 Conclusion

Our study first explores the impact of using different types of blocks for the neural networks used in computer chess engines. We notice that using MobileNet blocks instead of the classical residual blocks can help to improve the results in term of policy accuracy and MSE loss but at the cost of more latency and more memory required for the training and inference steps.

We also find that the use of specific chess filters reduces the number of trainable parameters while improving results for almost all neural network configurations. These chess filters work better than random filters. This means that imposing some kind of structure related the movement of pieces in convolutional layers can be useful for computer chess.

Future work could involve a low-level CUDA or TensorRT implementation of the depthwise convolution layer with chess kernels (without using a mask) that could be optimized for processing 8 by 8 images to reduce the latency. Future work will also involve assessing the impact of these new architectures in terms of ELO, in particular compared to conventional architectures used in Lc0.

Acknowledgment

This work was granted access to the HPC resources of IDRIS (Grant No. AD010611887R1) from GENCI. We are grateful to the reviewers for their comments.

References

1. Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

2. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
3. Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.
4. Dominik Klein. Neural networks for chess. *arXiv preprint arXiv:2209.01506*, 2022.
5. Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11976–11986, 2022.
6. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.
7. Julian Schrittwieser et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.
8. Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.
9. David Silver. Reinforcement learning and simulation-based search in computer go. 2009.
10. David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
11. David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.
12. LCZero Development Team. Leela chess zero (lczero), 2018. Available at `https://lczero.org`.