

# Playing Board Games with a Deep Convolutional Neural Network on the Motorola 6809 8-Bit Microprocessor

RÉMI COULOM<sup>1,a)</sup>

**Abstract:** While training deep-learning neural networks often requires considerable amounts of computing power, inference is efficient, and can be run on small devices. Cell phones are a typical example, but they are still rather powerful. The research presented in this paper takes the challenge to the extreme by running a Go-playing convolutional neural network on the 6809 CPU, an 8-bit microprocessor launched by Motorola in 1978. The software was implemented on a Thomson MO5 microcomputer, and reached a playing strength on par with GNU Go.

**Keywords:** deep learning, quantization, neural networks

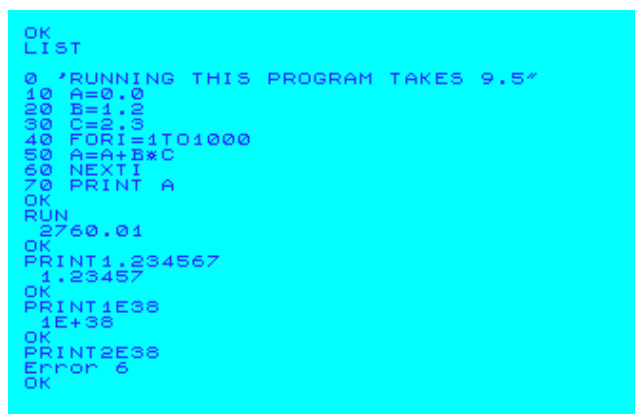
## 1. Introduction

The AlphaZero method [9], [10], [11] has become the most successful approach for the development of board-game AI. It is a reinforcement learning algorithm that can train a convolutional neural network to reach an extremely high level from self-play, without human knowledge. The first ever victory of a computer against a human Go master used this approach, and it is the algorithm used by all current top Go programs.

Training a neural network with this method is rather expensive, but using the resulting neural network to play games is efficient. During last year's GPW, an implementation of a gomoku neural network in Factorio was presented [3].

The research presented in this paper aims to run such a neural network with an even less powerful machine, the Thomson MO5. The Thomson MO5 is a 8-bit microcomputer that was popular in France in the eighties. It was used as an educational tool in French schools. Many French kids had their first experience of a computer with that machine. It is equipped with a 1MHz Motorola 6809 CPU, a  $320 \times 200$  display, 32 kb of RAM, and 16 kb of ROM that includes the Microsoft BASIC interpreter.

Microprocessors of that time had no hardware support for floating-point operations. The BASIC interpreter of the MO5 provides floating-point capabilities, but calculations are done in software with integer operations, and are very slow. Figure 1 shows the output of a simple benchmark experiment: 9.5 seconds for 1,000 multiplications. Such a performance is not enough to run a Go-playing neural network at a reasonable speed.



```

OK
LIST
0 *RUNNING THIS PROGRAM TAKES 9.5*
10 A=0.0
20 B=1.2
30 C=2.3
40 FOR I=1 TO 1000
50 A=A+B*C
60 NEXT I
70 PRINT A
OK
RUN
2760.01
OK
PRINT 1.234567
1.23457
OK
PRINT 1E38
1E+38
OK
PRINT 2E38
ERROR 6
OK

```

Fig. 1 32-bit floating-point capabilities of the MO5

Although floating point operations are too slow, it is still possible to run the neural network fast enough with quantization [5]. The high accuracy of 32-bit floating-point numbers is not at all necessary for correct calculations in a neural network, and faster low-resolution integer operations can be used instead. Unlike the 6502, its competitor that equipped the Apple II, and the Commodore 64, the 6809 provides an 8-bit multiplication operation that runs in 11 clock cycles only. Although it is slower than modern GPUs (see Table 1), its MUL operation makes the 6809 a great processor for deep learning.

The rest of this paper describes the approach for developing this MO5 software, and detailed experiment results. These experiments lead to the production of an artificial Go player with a strength similar to the strength of GNU Go on the  $9 \times 9$  board. But it is likely possible to make it stronger, and potential for further significant improvements will be discussed as well.

<sup>1</sup> <https://www.kayufu.com/>

<sup>a)</sup> remi.coulom@gmail.com

Machine	Data Type	Operations per second
MO5	float32	105
	int8	38,461
NVIDIA H100 NVL	sparse tf32	1,979,000,000,000,000
	sparse fp8	7,916,000,000,000,000
	sparse int8	7,916,000,000,000,000

**Table 1** MO5 computing power compared to a modern deep-learning GPU. MO5 numbers are actual practical number of multiplications per second when performing a convolution. NVIDIA numbers [7] are theoretical peak performance. Non-sparse performance is not given on that web site, but should be half of the sparse performance.

## 2. Approach

The most fundamental operation in neural network calculations is the computation of matrix multiplications. Convolutions used for board games are a linear transformation that can be expressed as a multiplication of a matrix of input data by a matrix of neural-network weights. In order to estimate how big a neural network can be run, it is necessary to estimate the amount of time and memory required to compute a matrix multiplication.

Table 2 shows a sketch of a dot product, a fundamental building block of the matrix multiplication. The result of 8-bit multiplications are added into 16-bit accumulators. Because of the risk of overflow when adding up many results of 8-bit multiplications, it will not be possible to use the full 8-bit resolution. Modern 8-bit deep-learning accelerators usually use a 32-bit accumulator to avoid overflow, but that would be very expensive on the 6809.

Code		Cycles	Bytes	Pseudo-C
LDB	-11, U	5	2	B = U[-11]
LDA	#\$07	2	2	A = 7
MUL		11	1	D = A * B
LEAX	D, X	8	2	X = X + D
LDB	-10, U	5	2	
LDA	#\$02	2	2	
MUL		11	1	
LEAY	D, Y	8	2	

**Table 2** Two multiplications in a 8-bit dot product. Weight values are hard-coded as constants into the code. The multiplication is unsigned, so two registers are used: X is used for positive accumulation, and Y is used for negative accumulation. The final result of the dot product is obtained as X - Y

With this approach to the dot product, an optimistic estimate is 26 clock cycles per multiplications. Eight  $3 \times 3$  convolution layers with eight channels on a  $9 \times 9$  board amount to  $8 \times 8 \times 8 \times 3 \times 3 \times 9 \times 9$  multiplications. At 26 clock cycles per multiplication, this is 9,704,448 clock cycles, so about 10 seconds, which is a reasonable pace for a fun game of Go.

The good performance of this dot product routine relies on the hard-coding of weight constants into the code. At 7 bytes per multiplication, the total amount of necessary code would be  $8 \times 8 \times 8 \times 3 \times 3 \times 7 = 32,256$  bytes. This is too much for the MO5. But it is possible to solve this

problem by generating the code on the fly. A routine for computing a convolution for one output and 8 inputs will take  $8 \times 3 \times 3 \times 7 = 504$  bytes, and will be called  $9 \times 9$  times. The cost of writing the weights into the routine code is small compared to the time it will take to run it. So we can solve this code-size problem by just-in-time compiling the dot product before using it 81 times.

## 3. Experiments Details

### 3.1 Training the Neural Network

The neural net uses 8 simple input features (Table 3) that can be computed rapidly on the MO5. Outputs are policy and final point ownership. It is a residual neural network with 4 blocks of 2 8-channel convolutions, with ReLU activation. Batch normalization was used after every convolution.

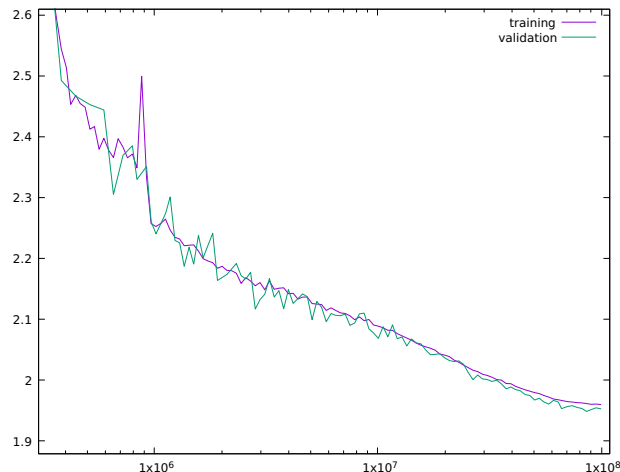
Channel	Description
0	Empty
1	My stone
2	My stone, 1 liberty
3	My stone, 2 liberties
4	Opponent stone
5	Opponent stone, 1 liberty
6	Opponent stone, 2 liberties
7	Opponent stone, 3 liberties

**Table 3** Inputs

Channel	Description
0	Policy
1	Ownership

**Table 4** Outputs

The data used for training is made of 145,408 self-play games of Crazy Stone, a strong AlphaZero-like Go program. The neural network was trained with plain stochastic gradient descent with momentum of 0.9 on a PC with a RTX 2080 Ti GPU. Batch size was 2048. Learning rate was warmed up to 1.0 for 30 seconds, and decayed exponentially to 0.0002 for one hour. Figure 2 shows the plot of policy loss (cross-entropy).



**Fig. 2** policy loss as a function of training samples

Such a small network does not overfit at all with this big a training set.

### 3.2 Estimation of Playing Strength

A 99-game match was played against GNU Go, with Chinese rules, and komi 6.5. GNU Go lost the match with a score of 46-53. This result is not significant of superiority over GNU Go, but indicates that the strength is similar.

### 3.3 Quantization

After training the neural network with floating-point numbers, post-training quantization was applied. It works by measuring the maximum activation value for each input channel over a large amount of test data. There is no value truncation: inputs are scaled so as to not overflow this maximum. Each output channel is also re-scaled so that the maximum absolute value of a weight to this output stays in the range of the weight quantization.

Table 5 show the effect of quantization resolution on policy loss. 8-bit quantization for both activations and weights produces a loss almost identical to floating point. But because using the full 8-bit resolution for both activations and weights will cause 16-bit overflow in the sum accumulator, a lower resolution has to be carefully selected.

activation bits	weight bits	policy loss
float	float	1.946
8	8	*1.951
8	7	1.971
8	6	2.005
8	5	2.212
8	4	3.404
7	8	*1.954
7	7	1.972
7	6	2.008
7	5	2.209
6	8	*1.961
6	7	*1.981
6	6	*2.011
6	5	2.222
5	8	1.992
5	7	2.017
5	6	*2.049
4	8	2.121
4	7	2.146

**Table 5** Policy loss for various post-training quantizations, measured over 10,000 test positions. Activations are unsigned, and weights are signed. A star marks the best choice for a given number of total bits.

The resolution was chosen by testing the range of values taken by the accumulator over a large number of test positions. It turns out that a total resolution of 13 bits for the weight-activation product is enough to prevent 16-bit overflow. According to Table 5, the best choice in terms of optimizing the policy loss is 6 bits for activations, and 7 bits for weights.

### 3.4 Tools

The development and debugging of the Go program was done with DCMOTO [2] (Figure 3). It has a debugger that

makes developing the program much more convenient than the real machine: code can be run step by step, memory and registers can be inspected, *etc.* The assembly code was written on a PC, and assembled with asm6809 [1].

## 4. Conclusion

### 4.1 Summary

Writing a Go-playing program for the Thomson MO5 8-bit microcomputer was a fun challenge. The resulting program is 7kb in size, uses 12kb of RAM when running, and plays a move at GNU Go strength on the 9x9 board in about 12 seconds.

### 4.2 Future Work

The current program could be improved with quantization-aware training [5]. By applying quantization during training, the network can be made more robust to weight rounding. This may allow stronger quantization, which can be an opportunity for significant performance improvements. For instance, by using 4-bit activations and 5-bit weights, the unsigned product would fit in 8 bits, and the very expensive `LEAX D,X` (2 bytes, 8 clock cycles) could be replaced by the much faster `ABX` (1 byte, 3 clock cycles).

Another significant improvement could be made by hard-coding each multiplication into code. Multiplication by zero would take no time and space at all. Multiplication by 1 is a plain addition. Multiplication by powers of 2 can be computed with faster bit-shifting operations. Small weights could use `ABX` instead of `LEAX D,X`. The main problem of this approach is code size. Just-in-time compilation of the convolution kernel would be less profitable, because the compiler would be much more complex. But the gain in code size might be enough to fit the whole network code in the MO5 RAM.

Sparsification [6] is yet another valuable approach to inference optimization. By setting a large number of weights to zero, the multiplication hard-coding approach described previously would be even more profitable.

Although bit-shifting is not very cheap on the 6809, training weights to be powers of 2 might be another interesting idea [4], [12]. Bit shifting also offers the interesting possibility to shift right, that is to say multiply by non-integer numbers. This can be a way to keep a higher resolution in weights, while ensuring the the result of the multiplication fits in 8 bits.

Finally, pushing quantization to the extreme with 1-bit quantization [8] may be efficient on the 6809.

Most of these techniques can be applied to modern devices as well. Tensor cores of modern NVIDIA GPU support low-resolution and sparse matrix multiplication in hardware. It seems that their usage is not yet very widespread among competitive game AI programmers, but it will certainly become more popular since quantization and sparsification can bring very significant gains.

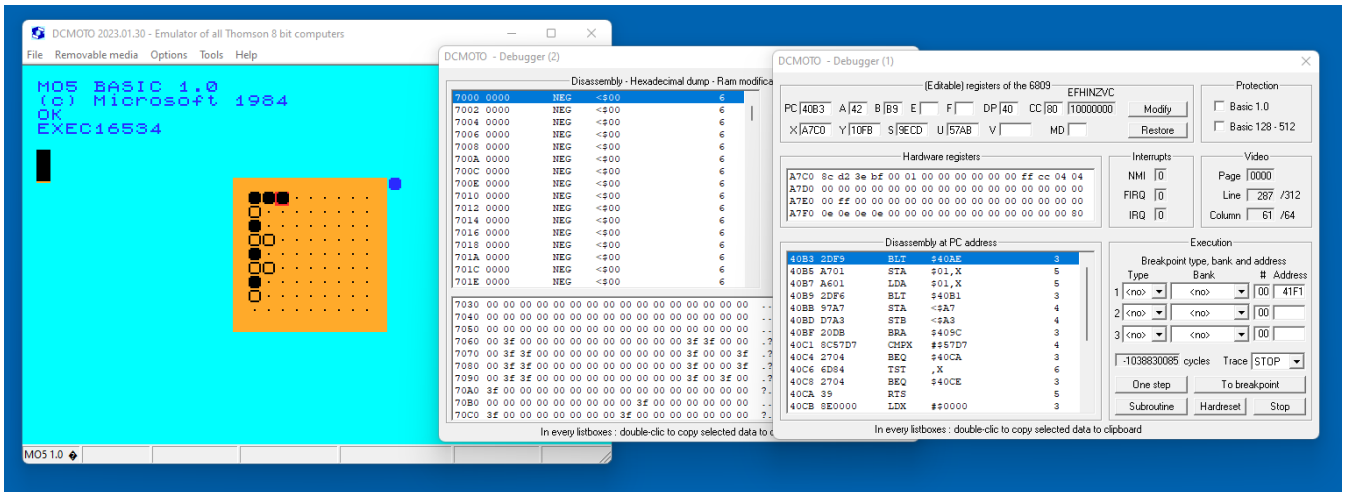


Fig. 3 Debugging the input tensor with DCMOTO

## References

- [1] Ciaran Ancomb. asm6809, 2023. <https://www.6809.org.uk/asm6809/>.
- [2] Daniel Coulom. DCMOTO, Émulateur de tous les ordinateurs 8 bits Thomson, 2023. <http://dcmoto.free.fr/emulateur/index.html>.
- [3] Rémi Coulom. Deep-Learning Gomoku AI in Factorio, 2022. [https://www.youtube.com/watch?v=-Ng8E\\_s6Xgo](https://www.youtube.com/watch?v=-Ng8E_s6Xgo).
- [4] Mostafa Elhoushi, Farhan Shafiq, Ye Henry Tian, Joey Yiwei Li, and Zihao Chen. Deepshift: Towards multiplication-less neural networks. *CoRR*, abs/1905.13298, 2019.
- [5] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021.
- [6] Torsten Hoefer, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *CoRR*, abs/2102.00554, 2021.
- [7] NVIDIA. NVIDIA H100 Tensor Core GPU, 2023. <https://www.nvidia.com/en-us/data-center/h100/>.
- [8] Hadi Pouransari, Zhucheng Tu, and Oncel Tuzel. Least squares binary quantization of neural networks. In *CVPR Workshop*, 2020.
- [9] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, January 2016.
- [10] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140–1144, December 2018.
- [11] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, October 2017.
- [12] Haoran You, Xiaohan Chen, Yongan Zhang, Chaojian Li, Sicheng Li, Zihao Liu, Zhangyang Wang, and Yingyan Lin. Shiftaddnet: A hardware-inspired deep network. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 2771–2783. Curran Associates, Inc., 2020.