**DEEP LEARNING FOR CHESS AI**

*A.S. Fedorenko*
*Omsk, F.M. Dostoevsky OmSU*

Deep learning - a set of machine learning methods based on feature/representation learning, rather than specialized algorithms for specific tasks. Many deep learning methods were known back in the 1980s, but the results were unimpressive until advances in the theory of artificial neural networks and the computing power of the mid-2000s made it possible to create complex technological architectures of neural networks with sufficient performance and to solve a wide range of problems. who didn't give in to an effective solution earlier, for example, in computer vision, machine translation, speech recognition, and the quality of the solution in many cases is now comparable, but not Otori cases superior to 'live' experts [1].

Chess is a game with a finite number of states. This means that, with endless computing resources, we could find a solution to this game as follows:

1. Assign all final positions values –1, 0, 1;
2. We apply the recursive rule $f(p) = \max_{p \to p'} - f(p')$, where $p \to p'$ denotes all valid moves from position $p$.

As a result, the number of possible positions in chess is 1043. Calculations of this scale cannot be performed. In this paper, we try to resort to the approximation $f(p)$ [2].

**Material and methods.** The game used 400 chess games played from the FICS open database. Learning the function $f(p)$ is based on the following principles:

1. Players choose optimal or near optimal moves. That is, for two consecutive positions $p \to q$ we have $f(p) = -f(q)$.
2. If a player does not go from position p to position q, but to a random position $r(p \to r)$ , then the inequality $f(r) > f(q)$ must hold.

We will write a model - a neural network with a depth of 3 layers, a width of 2048 neurons, with ReLU neurons in each layer. The input is a layer wide, which determines the presence or absence of each figure in each cell. After three matrix multiplications (each with subsequent non-linearity), the final scalar product with a vector of dimension 2048 is calculated to reduce the result to a single value. In total, the network has 10 million unknown parameters.

To train the network, I used triplets $(p,q,r)$. If we denote the sigmoid as $S(x) = 1 / (1 + \exp(-x))$, then the general objective function will have the following form:

$$\sum_{(p, q, r)} \log S(f(q) - f(r)) + k \log(f(p) + f(q)) + k \log(-f(q) - f(p)).$$

These are the log-likelihood [3] of the inequalities $f(r) > f(q)$, $f(p) > -f(q)$, and $f(p) < -f(q)$.

I rented an AWS GPU and trained the model on 200 million lots in seven days using the stochastic gradient descent [4] with Nesterov momentum. I put all the triplets $(p,q,r)$ into an HDF5 file. For some time I experimented with various values of the learning rate, but then I realized that I just want to get a good result in a few days. As a result, I applied a slightly unusual learning speed scheme: $0.03 \cdot exp(-time\ in\ days)$. Since the data is very large, regularization is not required. Therefore, I did not use either dropout or L2 regularization.

I applied a little trick: coding the board in the form of 64 bytes, and then converting to a real vector of dimension 768 on the GPU. This provided a significant performance boost due to a significantly smaller number of I / O operations.

The basis of any chess AI is some function $f(p)$, used to obtain an approximate estimate of the position. This function is called evaluation function.

The evaluation function is used in combination with an algorithm that performs a deep search among millions of positions in the game tree. All chess programs use intelligent search

algorithms, but when moving through the game tree, the number of positions increases exponentially, so it is impossible to search more than 5 to 10 positions ahead. In practice, some approximation is applied to evaluate the leaves, and then some version of the negamax procedure is used to evaluate the possible next moves.

An example of the simplest evaluation function is a function based on the value of the pieces: each pawn – 1 point, each knight – 3 points, etc. We will use the function we have trained to evaluate the leaves of the game tree.

To summarize: to solve the problem, it is necessary to train the function $f(p)$, and then integrate it into the search algorithm.

**Findings and their discussion.** As it turned out, the function that I trained can really play chess. She beat me in all games. I organized a competition for my program and the Sunfish program, sponsored by Thomas Dybdahl Ahle. Did my program win? Sometimes.

I think the written algorithm could play much better with the following optimization:

1. More efficient search algorithm. For example, Sunfish used MTD-f, while I used negamax with alpha-beta pruning. I will not say that MTD-f is better, it is a fundamentally different algorithm and one could test it.
2. More efficient evaluation function. If we use more "complex" examples for training, for example, the results of the game of grandmasters, the result should be a more effective model of the evaluation function.
3. Speed up evaluation function. You could speed up the process if you train a smaller version of the same neural network.
4. Speed up evaluation function. In this work, the GPU for the game was not used - it was used only for training.

**Conclusion.** It is worth remembering that the written algorithm is still far from perfect and did not compete with any advanced chess program. However, it has some positive aspects: There is the opportunity to train the evaluation function directly on the "raw" data without pre-processing; relatively slow evaluation function.


1. Ciresan, D., Meier, U., Schmidhuber, J. Multi-column deep neural networks for image classification – 2012 IEEE Conference on Computer Vision and Pattern Recognition: journal, 2012. – Pp. 3642–3649.
2. Maschler, M., Solan, E., Shmuel, Z. Game Theory – Cambridge University Press, 2013. – Pp. 176–180.
3. Myung, I. J. Tutorial on Maximum Likelihood Estimation – Elsevier Journal of Mathematical Psychology, 2003. – Pp. 90–100.
4. Bottou, L., Bousquet, O. Optimization for Machine Learning – Cambridge MIT Press, 2012. – Pp. 351–368.


## SCANNING CAPACITANCE MICROSCOPY OF TGS–TGS+Cr FERROELECTRIC CRYSTALS


*R.V. Gainutdinov[1], N.V. Belugina[1], A.K. Lashkova[1], V.N. Shut[2], I.F. Kashevich[3], S.E. Mozzharov[2], A.L. Tolstikhina[1]*

*[1]Moscow, Shubnikov Institute of Crystallography of RAS*

*[2]Vitebsk, ITA of NAS of Belarus*

*[3]Vitebsk, VSU named after P.M. Masherov*


The method of Scanning Capacitance Microscopy (SCM) has rarely been applied to ferroelectrics and the observed contrast of capacitive images in the limited number of publications on this issue is interpreted in the literature in different ways. Although the nature of observed contrast was not completely established by the authors, the observed result indicated the possibility of visualization of ferroelectric domain structures by the SCM method [1].

The purpose of this work is to study of the capabilities SCM as a method of local nanodiagnostics of heterogeneous ferroelectric surfaces. The question of the applicability of the SCM method for the composite mapping of ferroelectric crystals with the growth periodic impurity structure is considered.