End-to-End Deep Neural Network for Automatic Learning in Chess

Konstantin Herud and Carsten Mueller

***Abstract*—This research paper presents an End-to-End Software Architecture based on Deep Neural Networks for Automatic Learning in Chess. Initially classifying and regressive approaches are explored to evaluating game configurations by employing deep belief networks. A third research approach which combines these, is then developed by quantizing the value range of the evaluation function. The neural network learns to assess game positions accurately during an unsupervised pre-training and supervised fine-tuning phase, using a dataset solely consisting of binary vector representations of the board and corresponding evaluations. An alpha beta tree search is used to complement the chess engine for finding optimal moves. The experiments show how artificial neural networks can develop a deep understanding of the application domain, despite having no prior knowledge of the game rules or strategies.**

***Index Terms*—Chess, chess engine, artificial intelligence, deep learning, artificial neural 5networks, deep belief network, alpha beta pruning, alpha beta tree search.**

I. Introduction

Artificial Intelligence (AI) is this generation's mantra. In particular, Deep Learning as part of Machine Learning, which is one of the subdomains of AI, has produced phenomenal results. In 2015, DeepMind's "AlphaGo" defeated one of the world's leading Go players for the first time; according to the estimates, this was not expected to happen until 2030 [1], but was particularly enabled by the development of Artificial Neural Networks [2], [3].

Chess is one of the most popular strategic board games in the world. Its complexity makes it a popular target for the development of Artificial Intelligence. In 1996, IBM was the first to succeed in defeating the then reigning world chess champion, Garry Kasparov, with the chess computer "Deep Blue" [4]. This success was the result of more than ten years of research and development by project leader Feng-hsiung Hsu. Even today, chess computers are often optimized by precise, year-long fine-tuning with chess experts, to provide unsurpassed excellence. A great deal of specialist knowledge is needed to master the many aspects and strategies of the game.

Manuscript received September 5, 2018; revised October 8, 2018. This work was supported in part by the Baden-Wuerttemberg Cooperative State University.

K. Herud is with the Department of Applied Informatics, Baden-Wuerttemberg Cooperative State University, Lohrtalweg 10, 74821 Mosbach, Germany (e-mail: kon.herud.15@lehre.mosbach.dhbw.de).

C. Mueller is with the Department of Applied Informatics, Faculty of Informatics and Statistics of the University of Economics, W. Churchill Sq. 4, 130 67 Prague 3, Czech Republic (e-mail: research@ieoca.org).

This research work demonstrates how self-learning 'deep learning' algorithms are used to develop a chess computer which independently learns strategic intelligent moves, without any prior knowledge of the board game and its rules.

II. Software Architecture

To play chess successfully, one needs to gain the deepest possible understanding of the effect of different moves [5].

Conventional chess engines use linear evaluation functions to combine various features of the game board to calculate a number which represents the quality of a position.

For example, the multitude and position of the pieces, safety of the king, central position occupancy, etc. are taken into account in order to arrive at a value by using a linear combination of these properties. The human understanding of any situation, however, goes far beyond that; each situation in the game requires an individual estimation of the configuration of the pieces and possible benefits. To mimic this abstract process through machine learning, we use artificial neural networks trained for the nonlinear evaluation of game positions.

A consecutive tree search then allows all possible consequences to be evaluated to determine the optimal move.

The basis of our software architecture is the development of a Deep Belief Network (DBN). To ensure a flawless design, it follows the SOLID-principles. Appendix A shows the UML modeling of our finished software.

A. Application Layer

The application is divided into two processes: training and prediction. The training phase is used for the highly accurate adaptation of the weights and biases of the artificial neurons and consists of two consecutive sub-processes. These are subdivided into pre-training and fine-tuning. Pre-training is used to optimize the training behavior and to avoid the vanishing gradient problem. "[...] particular to deeper nets [...] the gradients will either shrink towards zero or blow up as they are back-propagated, making learning of the weights before the last few layers nearly impossible" [6]. For this reason, we perform an unsupervised pre-training.

B. Pre-training

Before the actual training of the model there will be an unsupervised pre-training of every single hidden layer. "During each phase of the greedy unsupervised training strategy, layers are trained to represent the dominant factors of variation extant in the data. This has the effect of leveraging knowledge of X to form, at each layer, a representation of X consisting of statistically reliable features

of X that can then be used to predict the output [...]" [7]. In each of the hidden layers, patterns are extracted from within the data. At this point, the network is not yet aware of any information about target data, such as class affiliation or rating of the data record. In order to train each layer separately in advance, so-called Restricted Boltzmann Machines (RBMs) are used. An RBM is a non-directional model consisting of a visible and a hidden layer with symmetrically connected units [8]. This model is trained using the Contrastive Divergence algorithm [9], [10].

Thus, the various hidden layers of the DBN are trained in succession, starting with the input layer and the subsequent hidden layer. Once a layer has completed the pre-training, the network is used as a normal Feed-Forward Neural Network, whereupon this layer takes over the task of being the visible layer within the RBM [11].

C. Fine-tuning

The weights of the network have now been adapted to the patterns within the training data. The network does not know any characteristics of the data yet.

The fine-tuning has the following tasks:

- Add an output layer and perform supervised training.
- Final adjustments of the network parameters.

Since the hidden layers have already been trained, it is sufficient in this phase to adapt the parameters of the output layer. All the network parameters are readjusted using the Stochastic Gradient Descent Algorithm [12], [13].

D. Prediction

After the training has been completed, the model can be used to predict the characteristics of new datasets. This process can be represented as a simple matrix multiplication. Thus, on each artificial neuron *j* of a layer *l*, the incoming data *a* is summed, after multiplying it with its weights w_j. The value that is obtained from adding a bias b_j to the neuron and performing an activation function σ over this sum, is then propagated to the next layer, until the output layer eventually provides a result.

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \quad (1)$$

E. Persistence Layer

For the administration of the dataset, we fill an HSQL database table with the dataset created in the form of a CSV file in advance. The connection of this database to the DBN avoids the need to parse the data records during the training process and additional computational effort during runtime.

F. Presentation Layer

During the training phase the output visualizes key figures on the current epoch, the accuracy and the error rate of the network as well as the current training process.

The intuitive Forsyth-Edwards Notation for the chess engine is used to represent positions in the game and the detailed algebraic notation for playing moves.

III. Training of the Models

The goal is to provide the models with as little prior knowledge as possible to examine how well the models were able to learn on their own.

A. Dataset and Model Approaches

For the dataset, one hundred thousand random game positions from high quality games (ELO rating ≥ 2400) were extracted from the "FICS Games Database (www.ficsgames.org)" in PGN file format. To convert these game positions into a DBN-compatible input format, the respective game boards are transformed into binary vectors.

For each playing field, the presence or absence of one of the twelve individual figures is represented by a zero or one, resulting in a vector of length $12 \times 64 = 768$ for each game position. In addition the four castling rights were passed on, as these cannot be derived from the game position. Thus, each data record is represented by a binary vector of length 772, which represents the number of input neurons of the DBN.

In the *first approach*, the model should learn to determine whether each individual position would result in a victory or a defeat. The dependent variables for this approach are therefore three exclusive classes: victory, draw and defeat.

For the *regressive approach*, the dependent variables are determined in the form of a numerical value using the evaluation function of Stockfish, the best open source chess engine to date. These numbers, in the range [-300; 300], were normalized.

In the *third approach* developed, these numerical values are quantized into seven classes in order to again use the superiority of classifying models over regressive models. Equation 2 represents the function by which we subdivided the value range of the dependent variable of the regressive approach into different classes.

$$f(x) = \begin{cases} \text{Too far behind} & \text{for } x < -10 \\ \text{Significantly behind} & \text{for } x < -2 \\ \text{Imperceptibly behind} & \text{for } x < -0.5 \\ \text{Equal opportunities} & \text{for } -0.5 \le x \le 0.5 \\ \text{Imperceptibly ahead} & \text{for } x > 0.5 \\ \text{Significantly ahead} & \text{for } x > 2 \\ \text{Too far ahead} & \text{for } x > 10 \end{cases} \quad (2)$$

B. Parameter Settings

In this research three models for each of the three approaches are examined, each with the following parameter settings. The selection is based on results from research work [14], [15].

- Neurons per hidden layer:
  - 1st model: [772, 100, 100, 100]
  - 2nd model: [772, 400, 200, 100]
  - 3rd model: [772, 600, 400, 200, 100]
- Learning rate: $0.005 \times 0.98^{\text{Epoch}}$
- Batchsize: 32
- Pre-training epochs per layer: 50
- Fine-tuning epochs: 200
- Hidden layer activation: rectified linear unit
- Output layer activation and loss function:

- Classification: Normalized exponential function and categorical crossentropy

- Regression: Tangens hyperbolicus and huber loss

In order to ensure optimal generalization through the models, a Dropout Regularization is used for each of the hidden layers at a rate of 0.5 during fine-tuning. "By doing this scaling, 2n networks with shared weights can be combined into a single neural network to be used at test time. Training a network with dropout and using this approximate averaging method at test time leads to significantly lower generalization error on a wide variety of classification problems compared to training with other regularization methods" [16], where n is the number of artificial neurons in the network.

C. Model Evaluation

In order to objectively assess our models, a 10-fold cross validation for each of them is performed. The classifying approach achieves a very high maximum accuracy of *99.0*%. In contrast, the regressive approach, shows no improvement in accuracy (<*15*%), with the attribute regarding the difficulty of accurately predicting floating-point numbers. Since the loss of the corresponding models nevertheless decreases, it is assumed that the training process is successful. Both approaches show a generalized learning process without overfitting based on the Dropout Regularization. The quantized classification approach achieves a maximum accuracy of *85.24%*.

In addition, a significant over-adaptation after a small number of epochs is observed. Fig. 1 shows the monitored supervised fine-tuning error rates for the three model approaches. Since no significant improvement was apparent between the different configurations and the number of artificial neurons per hidden layer, the models were selected with *772-100-100*-100 neurons per hidden layer as favorites, because of the lower computational effort needed.

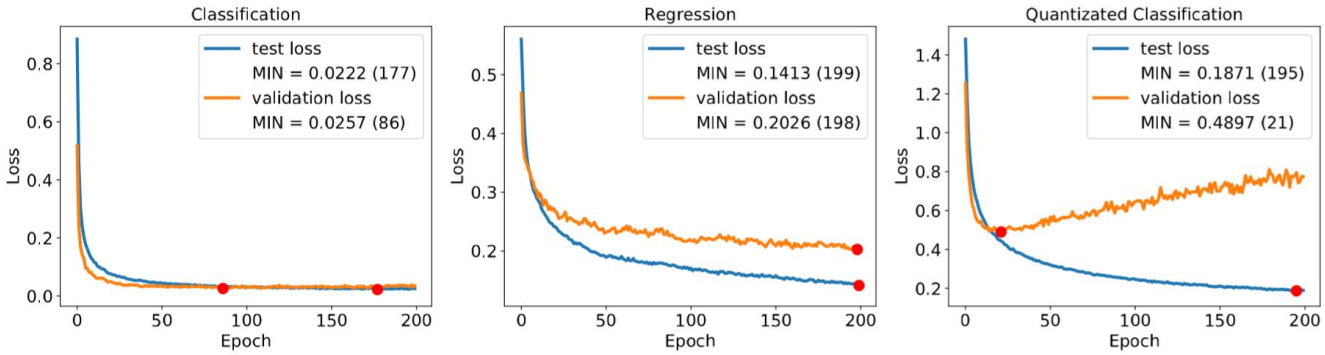


Fig. 1. Plotted loss of our models.

IV. Proof of Concept

To complement the capabilities of the DBN to evaluate chess positions and implementing a powerful software architecture for an end-to-end chess computer, and therefore to examine its playing strength, an Alpha-Beta Tree Search is adapted.

The aim of this search is to iteratively examine possible follow-up positions of a game position to the highest possible game depth, in order to be able to make the best choice in advance.

A. Alpha-Beta-Search

The principle of the algorithm is to update the α and β values for each node in the course of a depth-first search, in order to remove those subtrees with root nodes above or below the respective maximum or minimum for α or β [17].

α is the minimum result that player A will reach, and β, the maximum value that player B will achieve. The effectiveness of the algorithm depends heavily on the order in which the best moves are investigated. In the optimal case, the algorithm can reduce the exponential search complexity to $O(b^{d/2})$, where b is the number of average successor nodes of each node (about 35 in chess) and d is the search depth. The worst case complexity would be $O(b^d)$ [17].

By using a transposition table in the form of a hashmap we achieve two additional advantages: On the one side, the search depth are increased iteratively by keeping the results of the search tree within the hashmap, on the other side, multiple evaluation of nodes with the same game position is not needed, which considerably reduces the computational effort. "Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess" [17].

B. Skill Level

In order to compare the strength of the three approaches in the first step, the three models were compared in 100 games. They all used the same search algorithm and had to demonstrate how they use their learned evaluation function to select the optimal sub-tree leading to the strongest move.

The quantized classifying model proved to be most successful, with a slight lead over the regressive approach while the simple classifying model was clearly inferior. Out of a total of 300 games, the best scored 121 wins, the second 117 wins and the last, 62 wins. Hence, the quantized classifying model is used for further research experiments.

In the next step, the learned skill level was qualitatively examined. Thus various problems were identified for both sides of the game, black and white, based on which the model had to prove its degree of understanding of the game. Fig. 2 illustrates three of these problems. In this case, the moves of the chess computer are marked orange, whereas predetermined logical reactions of the other side are marked in blue.

The research experiments showed that the Deep Belief Network is not only able to learn simple concepts, such as the

roles and rules of the characters and checkmate independently, but also to develop advanced techniques, such as sacrificing pieces and foresighted positioning. Because of the high configuration count of all possible chess games (after 40 moves between 10^{115} and 10^{120} [18]), it is assumed that the DBN has not yet seen every play position. Hence it is evident that a learning process has successfully taken place.

In the final step, two instances of the same chess computer are placed side-by-side in a game against each other at a search depth of 8, to examine its playing behavior. The excerpt in Appendix B demonstrates how the DBN has learned to tactically plan ahead and think positionally.

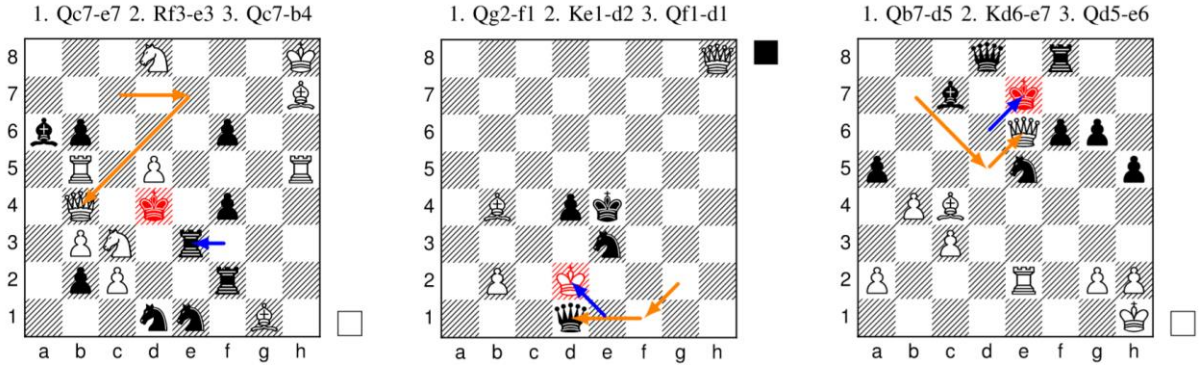


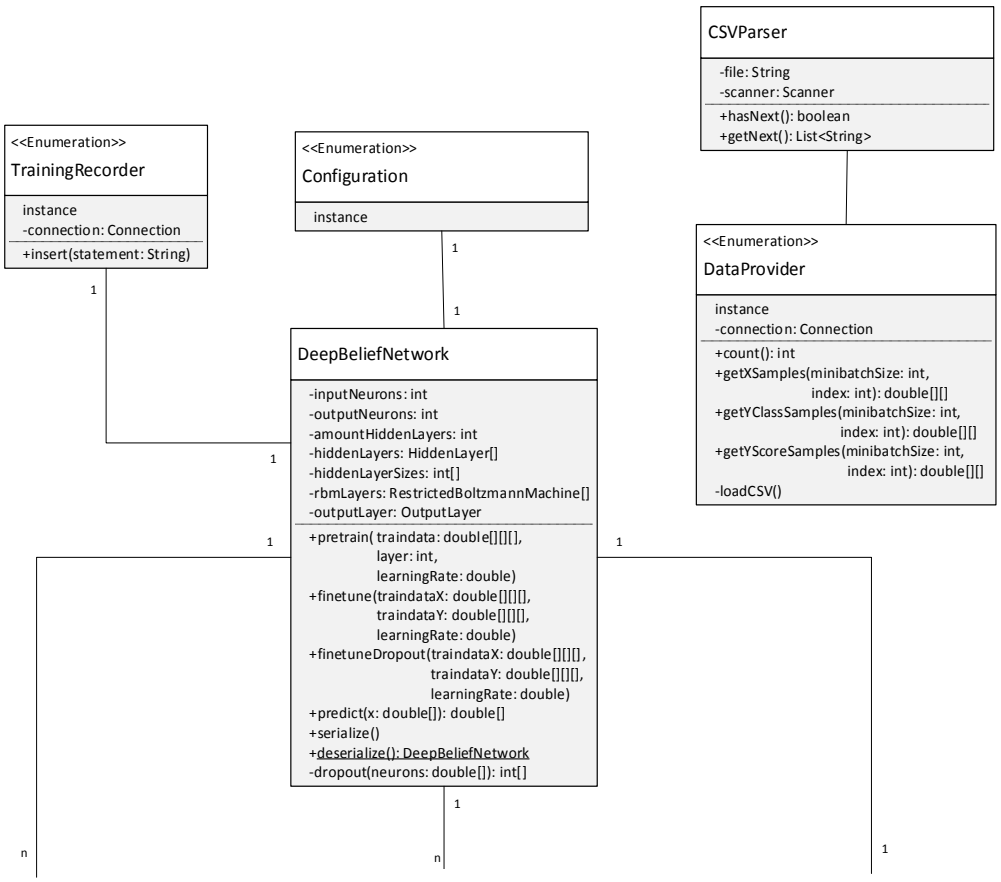
Fig. 2. Three different Test Challenges solved by our Application.

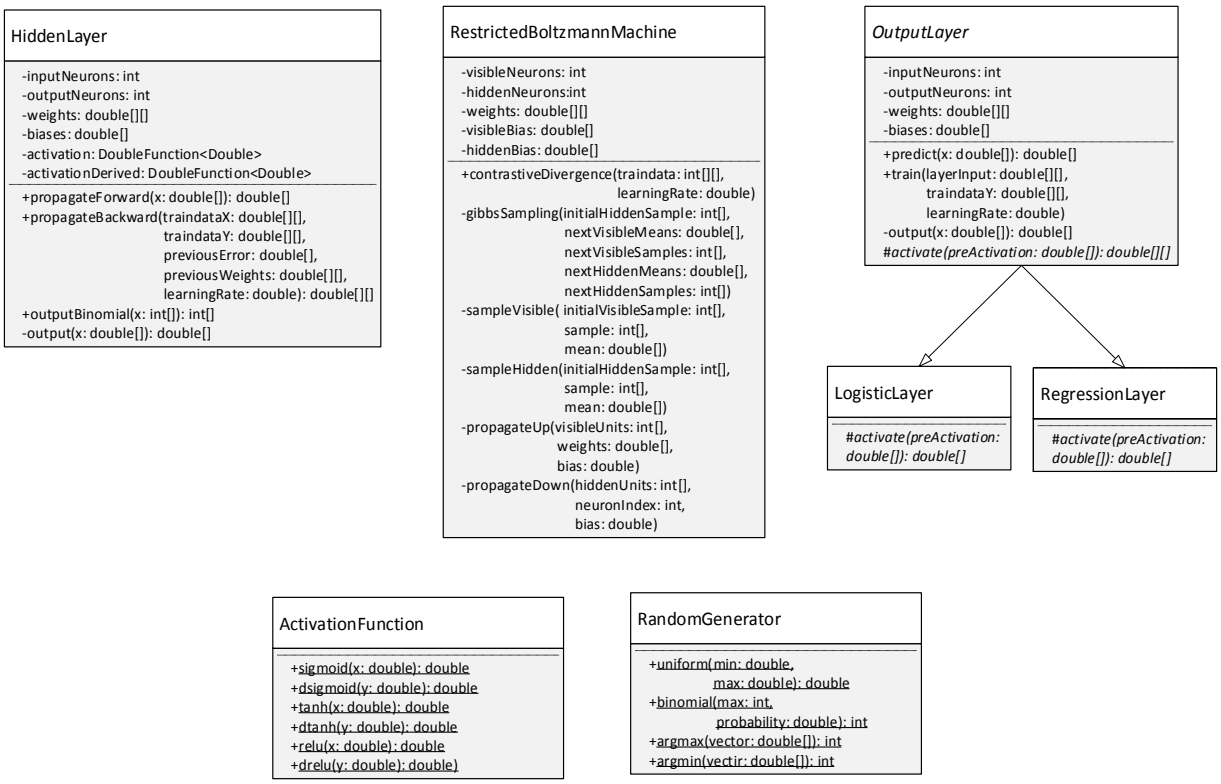
V. Conclusion

This research work demonstrates that the developed software architecture based on DBN is able to successfully learn paradigms and rules of chess independently. In contrast to the linear combination of various game board features of the evaluation function of conventional chess computers, these functions are expressed as a powerful purely mathematical matrix operation, thus laying the foundation for a highly parallelizable chess engine.

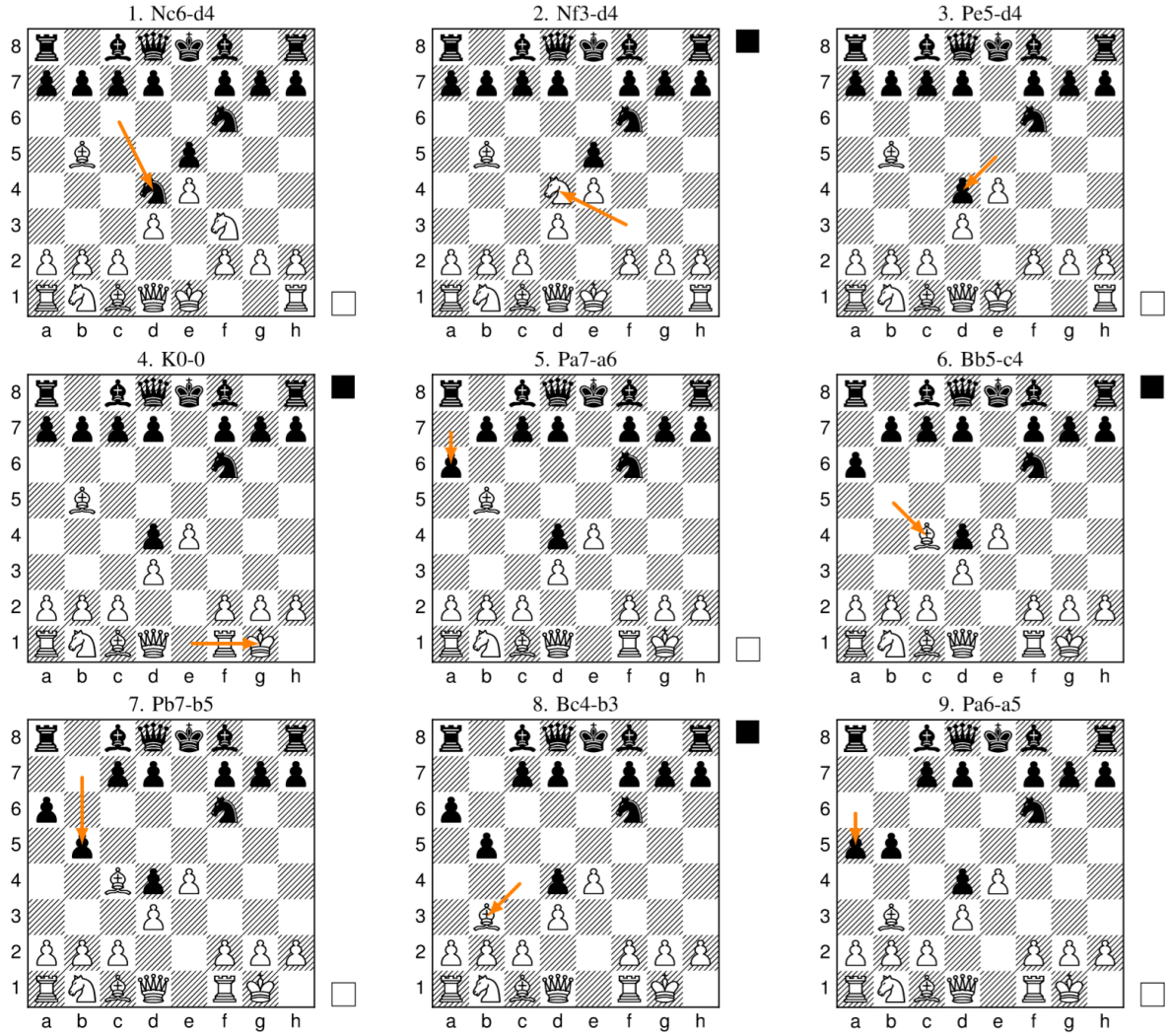
The experiments demonstrate only a fraction of the potential of the software by using only a single processor with about 50 GFLOPs and a very limited number of data sets, for example compared to Google's Alpha Zero chess computer with 5000 TPUs (180 TFLOPS / TPU) used for training [19]. It was presented how machine-based learning develops a human-like understanding of the field of application without a priori knowledge. This research work is a fundamental basis for the further development of chess computers in the future.

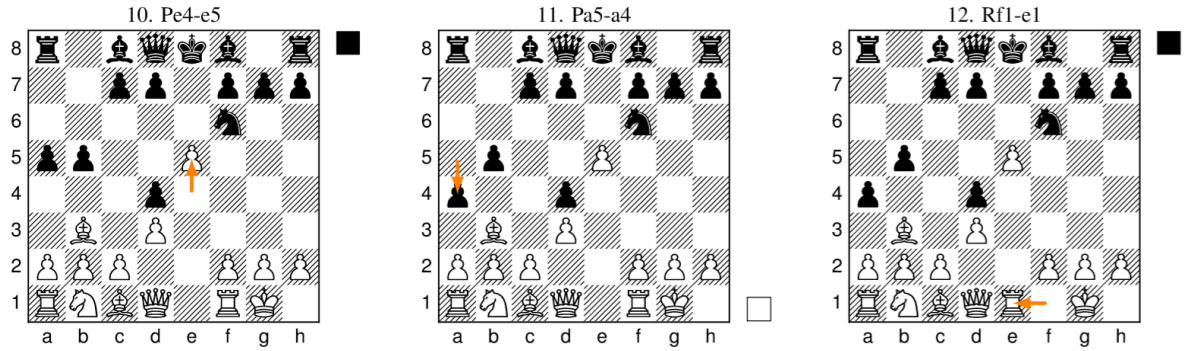
Appendix A
UML-Model of Our Software Architecture





APPENDIX B
EXCERPT OF A GAME BETWEEN TWO INSTANCES OF OUR CHESS ENGINE





ACKNOWLEDGMENT

I respect and thank Dr. Carsten Mueller, for providing me an opportunity to do this research project and giving me all support, feedback and guidance.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and working on this project. Thank you.

REFERENCES

[1] W. Duch and J. Mandziuk, *Challenges for Computational Intelligence*, Springer Berlin Heidelberg, 2007.
[2] D. Silver *et al*., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484-489, Jan. 2016.
[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, and A. Huang, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354-359, Oct. 2017.
[4] F. H. Hsu, "Ibm's deep blue chess grandmaster chips," *IEEE Micro*, vol. 19, no. 2, pp. 70-81, Mar. 1999.
[5] I. Bratko, D. Hristova, and M. Guid, "Search versus knowledge in human problem solving: A case study in chess," in *Model-Based Reasoning in Science and Technology*, L. Magnani and C. Casadio, Eds. Cham: Springer International Publishing, 2016, pp. 569-583.
[6] J. Martens, "Deep learning via hessian-free optimization," in *Proc. the International Conference on Machine Learning*, 2010, pp. 735-742.
[7] D. Erhan, Y. Bengio, A. Courville, P. A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *J. Mach. Learn. Res.*, vol. 11, pp. 625-660, Mar. 2010.
[8] G. E. Hinton, S. Osindero, and Y. W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527-1554, Jul. 2006.
[9] B. Marlin, K. Swersky, B. Chen, and N. Freitas, "Inductive principles for restricted boltzmann machine learning," in *Proc. the Thirteenth International Conference on Artificial Intelligence and Statistics*, May 2010, pp. 509-516.
[10] G. E. Hinton, "A practical guide to training restricted Boltzmann machines," in *Neural Networks: Tricks of the Trade (2nd ed.)*, Lecture Notes in Computer Science, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Springer, 2012, vol. 7700, pp. 599-619.
[11] Y. Sugomori, *Java Deep Learning Essentials*, Packt Publishing, 2016.
[12] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT'2010*, 2010, pp. 177-186.
[13] M. Hardt, B. Recht, and Y. Singer. (2015). Train faster, generalize better: Stability of stochastic gradient descent. *CoRR*. [Online]. Available: https://arxiv.org/abs/1509.01240
[14] O. E. David, N. S. Netanyahu, and L. Wolf, *Deep Chess: End-to-End Deep Neural Network for Automatic Learning in Chess*. Springer International Publishing, 2016, pp. 88-96.
[15] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807-814.
[16] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929-1958, Jan. 2014.
[17] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.
[18] E. Bonsdorff, K. Fabel, and O. Riihimaa, *Schach und Zahl: Unterhaltsame Schachmathematik*, Rau, 1966.
[19] D. Silver *et al*. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*. [Online]. Available: https://arxiv.org/abs/1712.01815

Konstantin Herud was born in Germany on the August 25, 1996. Herud graduated with his German Abitur in 2015 at the Wirsberg-Gymnasium in Würzburg. Currently, Herud is studying at the DHBW Mosbach, Germany with his major field in applied computer science, and is expected to graduate with his bachelor of science in September, 2018. Since 2015 he simultaneously works for the Flyeralarm Dienstleistungs GmbH in Würzburg as software developer. In 2012, he did an internship in the Faculty of Experimental Physics IV at the University of Würzburg, Germany. His current research interests lie in the field of artificial intelligence and in particular in machine learning.