UNIVERSITÄT OSNABRÜCK

Implementing Artificial Neural Networks with TensorFlow
WS 2017/18

A soft implementation of ”DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess”

Renato Garita, Hamza Kebiri and Turan Orujlu.

Abstract
This work encompasses a simplified implementation of DeepChess [1]. It is a deep neural network architecture that compares two chessboard positions and decides which one is more likely to result in a win for the whites.No knowledge of chess is provided to the network during training.The training process combines supervised and unsupervised learning. A Deep Belief Network consisting of Auetoencoders is pre-trained using unsupervised learning.It serves as a high-level feature extractor for the Siamese network that is trained using supervised learning.This final network is what the authors dubbed DeepChess. An alpha-beta pruning algorithm is implemented where DeepChess serves as the heuristic function. Because of computational constraints, we have created a chess playing program that can only compete with novice to amateur chess players.

Institute of Cognitive Science, Osnabrueck University.
Supervisor: Lukas Braun

Contents

1. Introduction

The game of chess is the most widely-studied domain in the history of artificial intelligence. Chess engines have been developed since decades, however incorporating machine learning techniques has not been as successful as other approaches in challenging grandmaster-level players. By the end of the 20^{st} century, *IBM Deep Blue* [2], which used a complex hand written board evaluation function that reflects the wisdom of top human chess players, was the first chess program to beat the world chess champion at that time, Garry Kasparov. Twenty years later, *Google AlphaZero* [3] and a chess engine called *Giraffe* [4] deployed a reinforcement learning approach that used no a priori knowledge about strategies except the basic rules. Both approaches have reached a standard of play comparable to international masters. DeepChess [1] has the novelty of being trained from scratch, learning winning and losing positions from millions of games. Learning chess rules was only a side effect of learning to predict which position is best among two. Due to computational limitations, we have implemented a soft version of DeepChess that achieved decent results while playing against humans.

2. Data

The CCRL dataset [5] was used for training. It contains roughly 800,000 chess games, out of which White won 35% and Black won 25% of games, the remaining games ended in a draw. According to the authors of the DeepChess paper, the inclusion of games that ended in a draw is not beneficial, so only the games which ended in a win were used for training. For computational reasons, we had to train the network on a smaller dataset. We processed each game with a probability of 0.5, thus reducing the dataset in half. From each selected game, we randomly extracted ten positions, with the restriction that the selected position cannot be from one of the first five moves in the game, and that the actual move played in the selected position is not a capture. According to the authors, capture moves are misleading as they mostly result in a transient advantage since the other side is likely to capture back right away. And with big enough depth limit (ply), the alpha-beta pruning algorithm can be successfully implemented without having the data points from the very first moves.

After loading the games using the Python chess library, their board representations were converted to a binary bit-string representation called bitboard. There are two sides (White and Black), 6 piece types (pawn, knight, bishop, rook, queen, king), and 64 squares. Therefore, in order to represent a position as a binary bit-string, 2 x 6 x 64 = 768 bits are required. There are an additional five bits that represent the side to move (1 for White and 0 for Black) and castling rights (White can castle kingside, White can castle queenside, Black can castle kingside, and Black can castle queenside).

3. Network structure

Each position in each state of the chessboard was mapped to a 773-bit representation. A deeb belief network (DBN [6]) which consists of a stack of autoencoders (Section 4.1) is used to learn a compressed representation of 100 bits (Figure 1, Stage-1). This was performed in stages by five fully connected layers $773-600-400-200-100$ with the rectified linear unit (ReLU) as an activation function. First, a three-layer autoencoder $773-300-773$ was trained. The obtained weights are fixed and then the second autoencoder $600-400-600$ was trained, and so on until the final 100 bit is reached. This phase is referred to as ***Pos2Vec***.

This feature learning phase is followed by a supervised phase that learns to predict winning and losing positions. ***DeepChess*** (Figure 1, Stage-2) network consists of two concatenated copies of Pos2Vec (shared weights) that are placed in parallel, and on top of them four fully connected layers 400, 200, 100, 2 are added. ReLU was used as the activation function, except for the last layer, where softmax was used.. The output from the feature learning phase was used for weights initialization. During training, DeepChess is completely modified including the two Pos2Vec components. And since the inference is too computationally expensive, network distillation (Section 4.2) was applied to improve speed. In the feature extraction part of DeepChess, a small four-layer network $773-100-100-100$ was first trained to mimic the original one $773-600-400-200-100$. Afterwards, three-layers $100-100-2$ were added on the top instead of the original $400-200-100-2$. Finally, the entire network was trained to mimic DeepChess.

Due to computational constraints, we have only implemented the distilled network (Figure 2). Its correspondent Tensorflow graph is shown in Figure 3

4. Theoretical basis

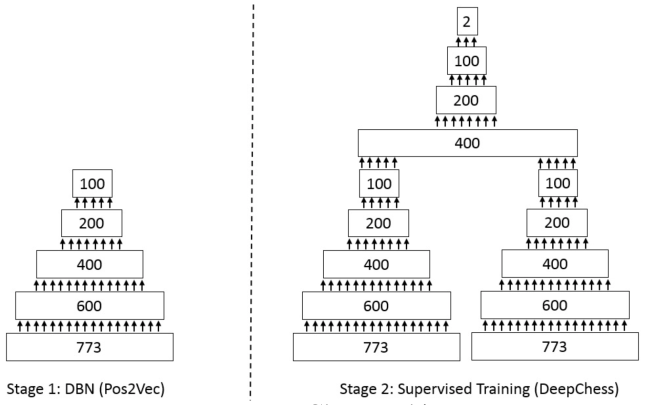

Figure 1. DeepChess architecture

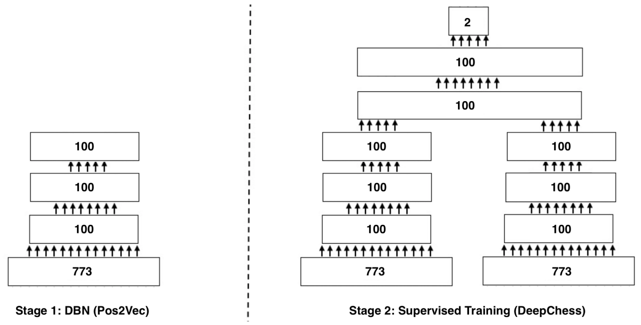

Figure 2. Implemented version of DeepChess

4.1 Autoencoders

The feature learning phase *Pos2Vec* uses a Deep Belief Network which is a multi-stack of sequentially trained autoencoders. An autoencoder is an unsupervised learning model used for feature learning and dimensionality reduction. It consists of a neural network that can be split into two parts: an encoder and a decoder (Figure 4). This neural network learns to approximate the identity function so that $\hat{x}$ is very close to x. In our case the network is forced to learn a compressed representation of the input, i.e. the hidden units. Nevertheless, an autoencoder can also learn a sparse representation of the data if the number of hidden units is higher than the number of input/output units.

4.2 Distilled networks

A smaller network was trained by David O.E. et al. [1] to mimic the original network. This technique is referred to as *network distillation* where a *student network* is trained to produce softened outputs of an ensemble of wider networks, *teacher network* [8]. The student network has to learn additionally to the labels, the information structure learned by the teacher network.

Formally, let T be a teacher network with an output softmax $P_T = softmax(\mathbf{a}_T)$ where $\mathbf{a}_T$ is the vector of teacher pre-softmax activations. Similarly for a student network S, $P_S = softmax(\mathbf{a}_S)$ where $\mathbf{a}_S$ is the vector of student pre-softmax activations. S is trained such that P_S is similar to the true labels $\mathbf{y}_{true}$ and to P_T, resulting to the following loss

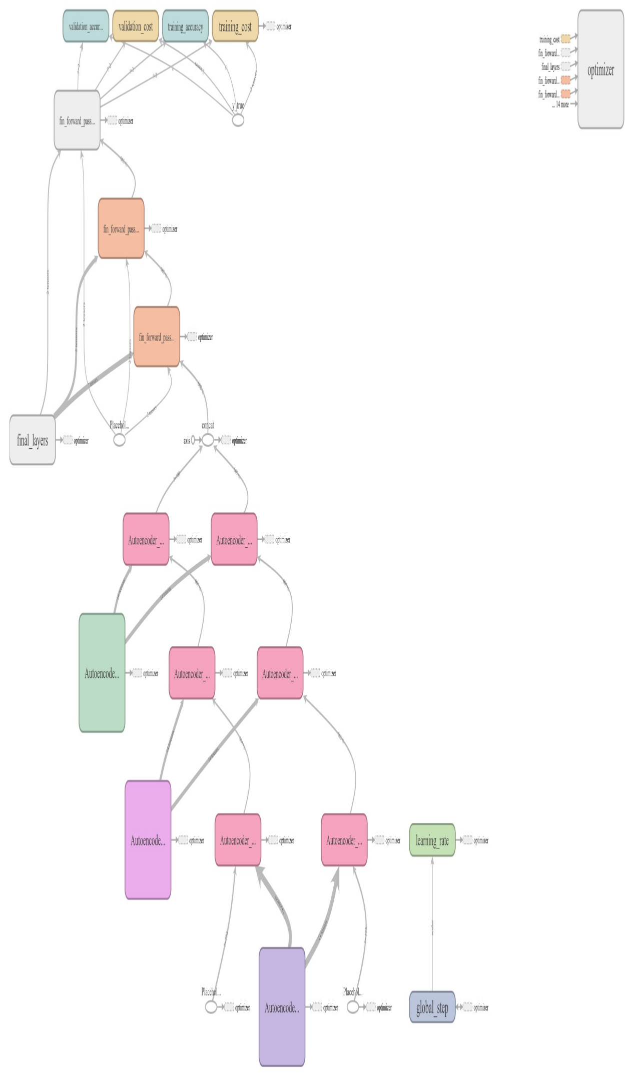

Figure 3. Tensorflow Graph of implemented network

function:

$$L(\mathbf{Ws}) = \mathscr{H}(\mathbf{y}_{true}, P_S) + \lambda \mathscr{H}(P_T^\tau, P_S^\tau)$$

Ws are the parameters of the student network S to be optimized, $\mathscr{H}$ is the cross entropy and λ is a tunable parameter to balance both cross entropies. A relaxation term $\tau > 1$ to soften the output from the teacher network was introduced since P_T can be very close to the true label. $P_T^\tau = softmax(\frac{\mathbf{a}_T}{\tau})$ and $P_S^\tau = softmax(\frac{\mathbf{a}_S}{\tau})$.

4.3 Minimax and alpha-beta pruning

An adapted version of Minimax with alpha-beta pruning was used to simulate chess games. Minimax is a widely used algorithm in 2-player games, where the two players, *max* and *min*, are competing against each other and both have complete information about the game. The whole game can be represented by a tree where each node represents a state of

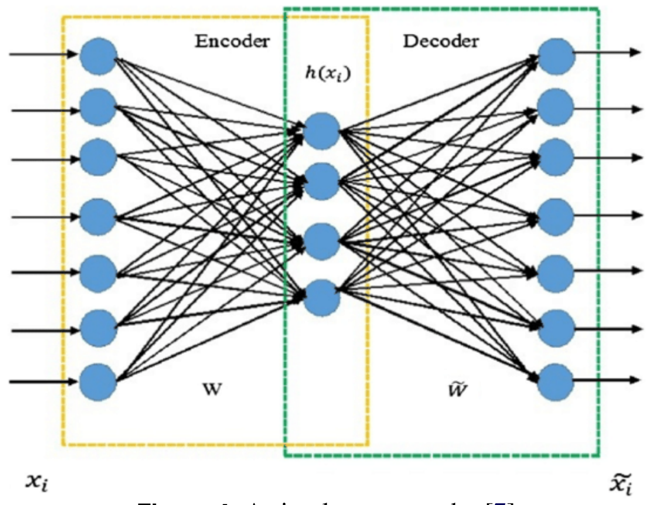

Figure 4. A simple autoencoder [7]

the game, i.e. the chessboard configuration in our case. A simplified example is shown in Figure 5. Alpha-beta pruning is an algorithm that is built on top of minimax, aiming to optimize the search efficiency by pruning unpromising branches of the tree. Given a search depth D and a branching factor B, alpha-beta pruning optimizes the depth-search algorithm from B^D to $B^{\frac{B}{2}}$. In an alpha beta search, two values α and β are updated for each node. α is the best already explored option along the path to the root for *max* and β is the best already explored option along the path to the root for *min*. Algorithm 1) updates α, β and the current node value by comparing the latter to alpha and beta values. However in the novel version used in our context (Algorithm 2), values are replaced by positions, and the comparison between positions is decided by the output of *DeepChess*. For instance, the current position is compared to an α_{pos} variable and to a β_{pos} variable using *DeepChess*.

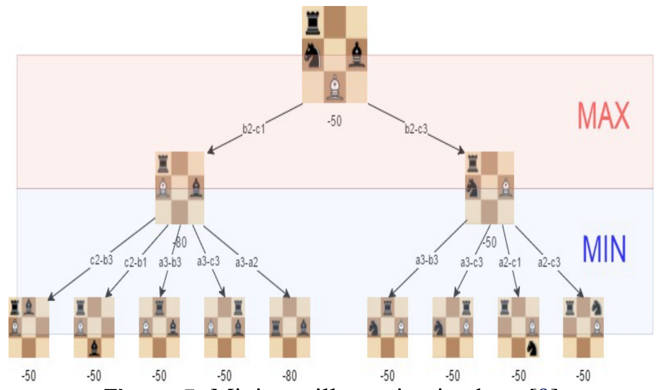

Figure 5. Minimax illustration in chess [9]

Algorithm 1. Original minimax with alpha-beta pruning [10]

```
AlphaBetaMax($\alpha$, $\beta$, $depthLeft$):
    if $depthLeft == 0$ return $-evaluate()$
    for ($allmoves$):
        score=AlphaBetaMin($\alpha$, $\beta$, $depthLeft - 1$)
        if $score \geq \beta$, return $\beta$
        if $score > \alpha$, $\alpha = score$
    return $\alpha$

AlphaBetaMin($\alpha$, $\beta$, $depthLeft$):
    if $depthLeft == 0$ return $-evaluate()$
    for ($allmoves$):
        score=AlphaBetaMax($\alpha$, $\beta$, $depthLeft - 1$)
        if $score \leq \alpha$, return $\alpha$
        if $score < \beta$, $\beta = score$
    return $\beta$

Initialization: $score = AlphaBetaMax(-\infty, +\infty, depth)$;
```

Algorithm 2. DeepChess minimax with alpha-beta pruning

```
AlphaBetaMax($\alpha$, $\beta$, $depthLeft$):
    if $depthLeft == 0$ return $evaluate()$
    for ($allmoves$):
        $pos$=AlphaBetaMin($\alpha$, $\beta$, $depthLeft - 1$)
        if DeepChess($pos$, $\beta_{pos}$)=$pos$, return $\beta_{pos}$
        if DeepChess($pos$, $\alpha_{pos}$)=$pos$, $\alpha_{pos} = pos$
    return $\alpha_{pos}$

AlphaBetaMin($\alpha$, $\beta$, $depthLeft$):
    if $depthLeft == 0$ return $evaluate()$
    for ($allmoves$):
        $pos$=AlphaBetaMax($\alpha$, $\beta$, $depthLeft - 1$)
        if DeepChess($pos$, $\alpha_{pos}$)=$\alpha_{pos}$, return $\alpha_{pos}$
        if DeepChess($pos$, $\beta_{pos}$)=$\beta_{pos}$, $\beta_{pos} = pos$
    return $\alpha_{pos}$

Initialization: $score = AlphaBetaMax(\alpha_{pos}^{worst}, \beta_{pos}^{worst}, depth)$;
```

It is worth mentioning that *min* is only used for simulation and not for actual moves. Typically, DeepChess uses maxmin algorithm with alpha-beta pruning to reduce the number of position comparisons and hence save computational time.

5. Performance evaluation

For the evaluation of DeepChess, we have implemented the file entitled *Chess_Playing_Environment.ipynb*. This IPython notebook file can be used to play matches against DeepChess. To run it, all cells have to be run. One of our group members, Turan Orujlu, who is a mediocre amateur chess player, played several matches against DeepChess. Unfortunately, in all cases, he defeated DeepChess quite easily. The program is susceptible to mistakes characteristic of novice players. According to the authors of the paper, DeepChess can achieve grandmaster-level proficiency. We followed all the guidelines provided in the paper with the exception of the data size, which we halved, and the network size which we downgraded significantly by using the parameters corresponding to the distilled network from the paper for computational reasons. Hence, the disparity in performance can only be attributed to these differences.

One other cause for the performance deficiency might be the alpha-beta algorithm. The exact implementation of the algorithm was not provided in the paper, so we had to improvise. The depth limit of the alpha-beta pruning algorithm could only be increased up to 4 due to computational limitations we had, which is not ideal as our data set lacked the opening 5 moves. One alternative to comparison-base alpha-beta algorithm is the probabilistic search algorithm of the Giraffe engine [4] that has trained parameters. The authors claim that the probabilistic search algorithm is superior to the classical minimax alpha-beta pruning algorithm. One idea for future might be to combine the probabilistic search of Giraffe and the evaluation network of DeepChess.

Figure 6 shows cross entropy and accuracy graphs. We did not achieve the training and validation accuracies of 98.2% and 98.0% that the DeepChess authors had achieved. Our training and validation accuracies were around 90.0-91.0% and 86.0-87.0%

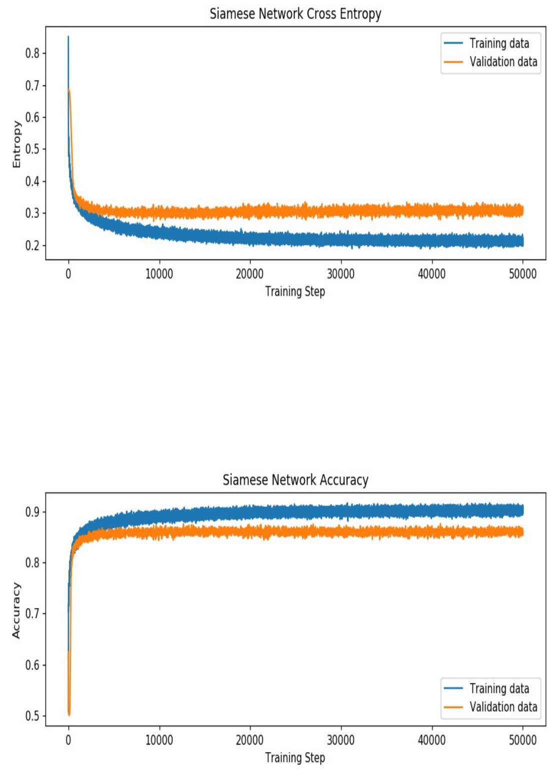

Figure 6. Network performance

References

[1] David O.E., Netanyahu N.S., Wolf L. (2016) DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess. In: Villa A., Masulli P., Pons Rivero A. (eds) Artificial Neural Networks and Machine Learning - ICANN 2016. ICANN 2016. Lecture Notes in Computer Science, vol 9887. Springer, Cham.

[2] Feng-hsiung Hsu IBM's Deep Blue chess grandmaster chips IEEE Micro (March-April 1999), pp. 70-81

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. CoRR, abs/1712.01815, 2017a.

[4] Lai, M. Giraffe: Using Deep Reinforcement Learning to Play Chess. MSc thesis, Imperial College London (2015)

[5] `www.computerchess.org.uk/ccrl/4040/`

[6] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. NIPS, 2007.

[7] `https://www.researchgate.net/figure/Autoencoder-architecture_fig1_318204554`

[8] Hinton, G. Vinyals, O. and Dean, J. Distilling knowledge in a neural network. In Deep Learning and Representation Learning Workshop, NIPS, 2014.

[9] `https://medium.freecodecamp.org/simple-chess-ai-step-by-step-1d55a9266977`

[10] `https://chessprogramming.wikispaces.com/Alpha-Beta#Implementation-Max`