Using Convolutional Neural Networks as Chess Engine

Maximilian Reihn
Dep. Information Science - Matriculated in Mathematical Finance
University of Konstanz
Konstanz, Germany
maximilian-martin.reihn@uni-konstanz.de

***Abstract*—This document will give detailed insight in how Convolutional Neural Networks (referred to as CNNs) are used in order to build a fully working chess engine based on the graphic interface of open source chess engine pythonchess [1].**

I. Motivation

Around 2010 CNNs rose to fame by showing breakthrough results in picture and pattern recognition. Even though the idea on matrix in and vector output networks using convolutions was established in the 1980s already, an efficient implementation and enough computational power was needed in order to make use of this theory. This report will try and use implied pattern recognition ability to exploit CNNs for predicting chess moves.

II. Introduction

A. Game

Chess is thought to have its first adherents in the 6th century A.D. located in Persia or China. It is played by two opposing players making moves alternately, belonging to the class of games with perfect information. The rules are simple yet it is highly complex strategically due to the sheer amount of possibilities.
There are six types of pieces on for each player, one player will inherit the white and his opponent all the black pieces. The game will be played on an 8 by 8 field, where every player owns eight pawns, two rooks, two bishops, two knights, one queen and one king. The aim of the game is to capture the opponents king by forcing it into check mate. White will start the game in the board layout seen in Fig. 1. For further information on chess rules: $wikipedia.org/wiki/Rules\_of\_chess$.
Mathematically chess is a zero-sum game meaning, every gain of any of the players adds up to zero with the losses of the opponent. It can be described as a Markov chain over a certain finite Riemannian manifold usually described as Ω being the set of distinct board positions, $|\Omega| \approx 10^{50}$.

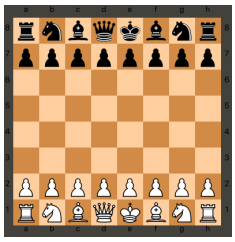

Fig. 1. Starting position of Standard chess game

B. Convolutional Neural Networks

The Theory of Neural Networks using neurons having vectorial input was first introduced 1949 by Donald O. Hebb [2]. With limited computational power and no backpropagation yet invented it had a hard time establishing. Therefore it took until after 1974 in which Paul Werbos invented the backpropagation algorithm [3]. In 1975 it was first used to recognise handwritten letters.
Its successor using matrices as input, are the Convolutional Neural Networks, which are most effective since after 2010, where they outperformed conventional Networks with impressive margins in picture recognition competitions.
Those neural networks will be used to try and build a chess engine which will move, when the opponent (human) moves.

C. Aim

The aim of this project is to establish an architecture of CNN(s) which is able to predict the most favourable move given the current board layout. This will lead to a move calculated, based on which field is the best to move to and which piece is the best to use making the move.
We will have some probabilistic distribution over the 64 tiles of the board and will move based on the field which is expected to yield the highest chance of winning the game.
We will ask ourselves the question if this move is unique or if there is more to take into consideration, one might think about some certain play style or some strategy which the engine wants to consequently follow.

III. Data gathering

A. Assumptions

Before starting to gather data we will have to make some assumptions. Our first and foremost question is how to evaluate a move. The approach in this paper will be to assume that the move of a chess player with an ELO score [1] of above 2600 is the correct move to be played given the current board layout. The moves which will train the CNNs are to be extracted out of ≈10 000 chess games in which only black won by check mate and the black players ELO score was 2729 on average. The CNNs will be trained to play the black pieces.
In a first approach we will also assume that the current board layout is the perfect information needed to make a decision and will not take into consideration the previous moves of the opponent player or his strategy.
Assume q_i to be the move which is most favourable given board layout x_i and $q(x_i, \omega)$ the move the CNN will predict using the current parameters ω. Then the aim is to minimise

$$H(X, \omega) = \frac{1}{n} \sum_{i=1}^{n} f(q_i, q(x_i, \omega)), \tag{1}$$

X being a batch of size n randomly taken from the training data set with elements x_i and their corresponding counterpart q_i which is the desired solution for given x_i.
ω being the parameters describing our CNN, $f(\cdot, \cdot)$ being a convex differentiable function with the characteristic of $x = y \Leftrightarrow f(x, y) = 0$. Finally $q(x_i, \omega)$ is the prediction the CNN takes.

B. Data base

There is a lot of data bases on the internet which provide nearly all recorded chess games on the planet for the last 20 years. We found $ficsgames.org$ to be the one with best the download format.
One will need to pass his filters for the game search, which for us were games after 2010, in which black won, ELO score greater than 2600 and Standard chess game. That resulted in around 10 000 games which on average had 50 moves. Therefore we had 500 000 moves to train from.

C. Data extraction

The file downloaded was a .txt file in which the games were denoted in PGN notation[2]. Using Python3.7 the .txt file was modified in a way such that pythonchess was able to read all moves together with the corresponding board layout. The information was put into the training set consisting of a npy.array of size $(500000, 13, 8, 8)$ representing the board and a npy.array of size $(500000, 64)$ representing the field moved to. Furher information on this representation will be explained in IV. The board layout was extracted as a 'string' in FEN-Notation[3]. Furthermore again using Python we eliminated all double examples, that means every board constellation is unique throughout the training data.

IV. Information encoding

A. Input encoding

The most naive way to encode a chess board would be to number the own pieces with positive numbers $\{1, .., 6\}$, the opponent pieces with the negative correspondents and then put them onto a 8×8 matrix. Even though this seems efficient it is proven not to be a good method, since this method compares different pieces in a quantitive way, but we cannot say that e.g. a king is worth six times more than a pawn.
There is a lot of different suggestions on how to encode information of a chess board, but the most common one is a bitmap solution, which is what was used in this project. Similar to pictures which have three channels for the RGB-colours, we create 12 channels every one corresponding to a certain type of piece[4].

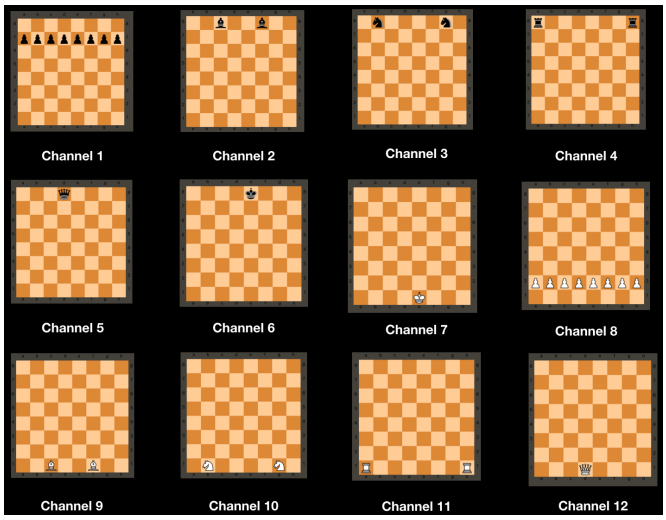

Fig. 2. Visualized bitmap representation

Each piece in each layer is simply represented by a 1 on the corresponding index of the matrix.

B. The 13th layer

As you may have noticed the 12 layers implied in IV-A are not consistent with the 13 layers mentioned in III-C.
After some testing on the training accuracy, having the 12 layer bitmap encoding, the highest accuracy achieved was ≈ 60%, which is consistent with other publications [4].
In this project there is another approach taken into consideration, while a CNN can detect highly complex patterns it is not

[1] ELO is a scoring system to evaluate how good of a chess player someone is. An ELO score of above 2600 is considered World Class.

[2] PGN (Portable Game Notation) is a certain form of notation most commonly used.

[3] Forsyth-Edawards-Notation is used to encode the board layout in a compact and unique way.

[4] Six types of pieces for the two players will have 12 distinguishable pieces.

able to look into future moves like a MINI-MAX algorithm [7]. This will also make it hard for a CNN to consequently follow a certain strategy, therefore we used the 13th layer representing the past three moves with 1 indicating the most recent field moved to, 0.5 the field moved to in the second to last move and 0.25 the field moved to in the third to last move. This may seem random but it boosted accuracy across all sets (training, testing, validation) around 30%.

C. Output encoding

First we have to consider, that for each move to be predicted we need two outputs/information: The field to move to which is represented by a $fields = 64 \times 1$ vector, each element of the vector indicating the probability that a pro chess player (at least the ones considered in the training set) would move to that field. Second we have a $pieces = 6 \times 1$ vector indicating which type of piece to use.
Combining the information held in those vectors we usually have a uniquely calculated move which will be executed by the chess engine, by choosing $(argmax(fields), argmax(pieces))$.

V. Architecture and Training

A. Layers

Both CNNs share one input and have respective outputs of $fields = 64 \times 1$ and $pieces = 6 \times 1$. Their architecture was more or less the same, disregarding the output dimension, we had

- Input shape $(n, 13, 8, 8)$ (n being the number of training examples)
- Convolutional layer with kernel size $(2, 2)$ and stride 1 as well as a L^2 regularizer with weight decay $5 * 10^{-6}$ and elu activation and 32 filters
- Normalization layer
- Dropout layer with $d = 0.15$
- Convolutional layer with kernel $(3, 3)$, same weight decay and activation and 64 filters
- Normalization layer
- Dropout layer with $d = 0.15$
- Convolutional layer with kernel $(3, 3)$, same weight decay and activation and 128 filters
- Normalization layer
- Dropout layer with $d = 0.2$
- Reshaping the matrix to vector (Flatten())
- Dense layer with 32 neurons and relu activation
- Normalization layer
- Dense layer with 64/6 softmax output neurons

That resulted in a total of 359 000 (357 000 for the piece predictor) parameters to be adjusted. Implied layers where realised by the Keras library.

B. Training parameters

The CNNs were trained on a e-GPU using plaidml[6] as Keras backend in order to be able to GPU train on a macOS device. The optimisation algorithm used is Adam [5] because it proved to deliver the best results.
We chose a batch size of 32 and the cross-entropy cost function because of its theoretical properties forcing it to 'learn' reasonably fast. Using Keras callback implementation a learning rate scheduler was used implementing the following properties.
Let α_k be the step-size for the k-th batch. Then assuming an infinite number of training steps the sequence $(\alpha_k)_{k \in \mathbb{N}}$ has the following properties obtained from "Optimization Methods for Large-Scale Machine Learning" [6].

$$\sum_{i=1}^{\infty} \alpha_i = \infty \text{ and } \sum_{i=1}^{\infty} \alpha_i^2 < \infty \tag{2}$$

We also use halving of the learning rate which is a logical implication by the above equation. Training was done over 15 epochs for each network taking around 60 minutes having an average of 4 T/flops and full accuracy.

C. Training results

The results of the above training is summed up in the following table:

TABLE I

Predictor	Parameters			
	Time trained	***Training acc.***	***Validation acc.***	***Test acc.***
Field	64 min	95.4%	94.9%	95.0 %
Piece	45 min	97.2%	97.1 %	96.3%

D. Testing results

To test the performance of the CNNs as chess engine we set it up against stockfish, which is a very sophisticated chess engine being continuously improved over the past decade. Stockfish uses a state-of-the-art MINI-MAX tree search algorithm which is able to search up to 70(!) million possible move combinations every second. We only let them play for 30 moves each time, due to the problems mentioned in VI-B3 regarding endgame.
In sets of 50 games played the board was in favour of stockfish 35 times, in favour of the CNNs eight times and tied seven on average. This result may seem rather bad, but considering hundreds of contributors have worked on stockfish and it has a massive architecture the results are relatively good in our measure. We will discuss if it would be possible to use this approach to get better results in VII.

[6]plaidml is an open source project to be used as Keras backend. For further information visit https://github.com/plaidml/plaidml.

VI. Analysis

In the following we will discuss the results obtained in Table I trying to draw conclusions.

A. Accuracy

Given the results from Table I we want to ask ourselves how the difference in accuracy compared to the papers mentioned in IV-B is composed. One might assume that this is due to the fact it overfits perfectly using the 7th layer as a good indicator doing so. But this is disproved by the fact, the accuracy in validation and test set is similar to the training accuracy even though as mentioned in III-C, every board layout (input) is unique throughout these three sets.
There is no obvious explanation for this phenomenon. Therefore we have to assume there is a certain degree of correlation between the 13th layer and the 'correct' move to take. This makes sense because one can assume that pro chess players make moves not only based on the current board layout, rather than also taking into account what happened the last few moves and what kind of strategy is working against a certain opponent.

B. Problems

Actually implementing those CNNs as a predictor in a way that one can play against it and it responds to ones moves came with a certain number of problems.

1) Legality of moves: Since we used more of a brute force strategy training the network, there is no architecture which prevents the CNNs from predicting a move which is not complying to the rules of chess. This problem was rather easy to solve with a backup MINI-MAX algorithm, which was implemented if the moves predicted were not legal. This leads to $\approx 90\%$ of the predicted moves being legal and the rest being calculated by MINI-MAX.

Algorithm 1 Move legality
Input: $Board\_layout$
Output: $move$
$CNN\_f = CNN1.predict(Board\_layout)$
$CNN\_p = CNN2.predict(Board\_layout)$
$move = (Argmax(CNN\_f), Argmax(CNN\_p))$
CNN_field.remove(move[0])
CNN_piece.remove(move[1])
$move2 = (Argmax(CNN\_f), Argmax(CNN\_p)$
if $||move - move2|| > 5 * 10^{-3}$ **then**
  $move2 = False$
end if
if $not(is\_legal(move))$ **then**
  move = move2
  if $not(is\_legal(move))$ **then**
    $move = MINI\_MAX(Board\_layout)$
  end if
end if
return $move$

2) Distinguishing between alike layouts: We can interpret the space in which the bitmap representation is defined as a Riemannian manifold, in which the distance function is defined as

$$d(x, y) := inf\{L(\gamma)\} \tag{3}$$

fullfilling

$$\gamma : \gamma \in C^1, \gamma(0) = x, \gamma(\gamma^{-1}(y)) = y \tag{4}$$

$$L(\gamma) = \int_x^y ||\gamma'(t)||dt \tag{5}$$

Observing this closely we have the problem arise in which the CNN cannot differ between positions, close in the manifold but which are very different in terms of chess strategy.
Let us demonstrate this by the following example:

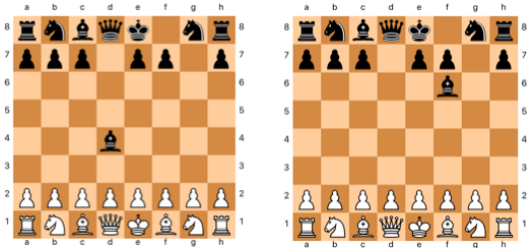

Fig. 3. Left: Example 1; Right: Original position

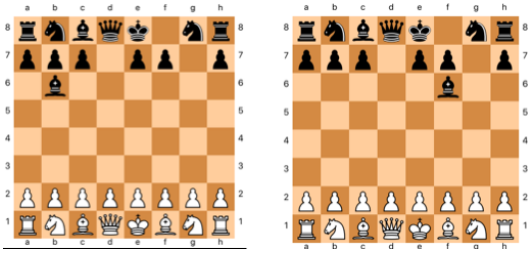

Fig. 4. Example 2; Right: Original position

On the right hand side we can see the same original position. Now if we compare the left board layout of Fig. 3 and Fig. 4 we can observe they have the same distance as per Equation 3.
But if we now compare those in terms of strategy, the two layouts shown in Fig. 3 are essentially the same and yield the same opportunities for upcoming moves because the bishop is in the same diagonal. Whereas if we compare the layouts seen in Fig. 4 we can see two completely different positions strategy wise because the fields which can be attacked by the bishop are different and the possible moves to be taken are different too.
This means that the CNNs have a hard time to choose the best move because it is hard for it to understand just small differences in the bitmap manifold which are huge strategic differences.

3) Strategy and Pressuring: Since the CNNs are only evaluating by the bitmap representation without any other features taken into consideration, it is generally hard for the CNNs to consequently follow a certain strategy. This is most noticeable in end game, where it is essential to pressure opponents such that they are forced into check mate. This usually results in the CNNs predicting unnecessary and repeating moves.

4) Beginner moves: Another problem occuring was it being confused by rather bad moves from beginners. Since it only learned from pro games, it was hard for the CNNs to adapt to beginner-level players since the CNNs have only seen high-level games in training. Playing according to other high-level games the CNNs predicted rather good moves and had solid games. This problem could be solved by training it with some games of worse players in order for the CNN to able to adapt to beginner-level players as well.

VII. Conclusion

As we have discussed earlier, this project is still far from being a sophisticated chess computer. There is a lot of problems occurring which cannot easily be solved without having greater resources and $>$100 hours of additional time to be spent on this project. A good suggestion for a project like this is to train the CNNs with a reinforcement learning strategy in order to train its strategic capabilities better.
Since there is nearly unlimited data of chess games it is a project for which it is easy to gather good data, even though the sheer size of the data can be challenging (The training np.arrays where $>$4GB in size). This is a very good prerequisite for projects like this, therefore with far more time and resources combined with reinforcement learning it should be possible to build a chess engine with a very high level of capabilities.

VIII. GitHub

Your are welcome to checkout, contribute or clone the project on GitHub

github.com/Cluckenstein/chess-engine-CNN.

References

[1] Open source project available on https://github.com/niklasf/python-chess
[2] Donald O. Hebb: ”Organization of Behavior”, 1949
[3] Paul Werbos, Dissertation: ”Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”, 1974
[4] Barak Osri, Nishith Khandwala: ”Predicting Moves in Chess using Convolutional Neural Networks” -Stanford University, 2014
[5] Diederik P. Kingma, Jimmy Ba: ”Adam: A Method for Stochastic Optimization”, 2014
[6] L’eon Bottou, Frank E., Curtis Jorge Nocedal: P.27 ”Optimization Methods for Large-Scale Machine Learning”, 2018
[7] Jerzy Cislo: ”An Analysis of the Min-max Algorithm”, 2011