DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2019

Training a Convolutional Neural Network to Evaluate Chess Positions

JOEL VIKSTRÖM

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Användning av ett konvolutionellt neuronnät för att evaluera schackpositioner

JOEL VIKSTRÖM

Bacholor of Computer Science
Date: June 7, 2019
Supervisor: Stefano Markidis
Examiner: Örjan Ekeberg
School of Electrical Engineering and Computer Science

Abstract

Convolutional neural networks are typically applied to image analysis problems. We investigate whether a simple convolutional neural network can be trained to evaluate chess positions by means of predicting Stockfish (an existing chess engine) evaluations. Publicly available data from lichess.org was used, and we obtained a final MSE of 863.48 and MAE of 12.18 on our test dataset (with labels ranging from -255 to +255). To accomplish better results, we conclude that a more capable model architecture must be used.

Sammanfattning

Konvolutionella neuronnät används ofta för bildanalys. Vi undersöker om ett enkelt sådant nätverk kan tränas att evaluera schackpositioner genom att förutspå värderingar från Stockfish (en existerande schackdator). Vi använde offentligt tillgänglig data från lichess.org, och erhöll en slutgiltig MSE 863.48 och MAE 12.18 på vår testdata (med data i intervallet -255 till +255). För att uppnå bättre resultat drar vi slutsatsen att en mer kapabel modellarkitektur måste användas.

Contents

Chapter 1

Introduction

Convolutional neural networks are a type of neural network typically used in machine learning problems relating to images [1]. They are capable of learning low-level features and combining several of those into higher-level features, eventually reaching an object-level understand of the image.

Chess is a game played by two players on a chessboard with different types of pieces. Which side is better in a chess position, and to what degree, is a function of the current board state. Many chess engines exist which evaluates board states, but only recently has convolutional neural networks been used for this purpose [2].

We hypothesise that a convolutional network, which is essentially made to detect local, low-level features and combine several of these into higher-level features, might be able to learn to understand chess positions, which are also combinations of many low-level features. This would have implications for the application of convolutional neural networks to other types of problems beyond just images.

Our research question is then: to what degree can a convolutional neural network learn to evaluate chess positions?

Chapter 2

Background

2.1 Artficial Neural Networks

Programatically processing and analysing images has been a long-standing problem within computer science, and continues to be until this day. Classifying an image as belonging to or containing one of a set of classes (classification), or estimating the value of some quantity using the image (known as regression) are common topics in machine learning. An examples of classification is for instance converting images of handwritten text and digits to actual text process-able by the computer, whereas regression can be used for instance to estimate the price of a house based on how it looks.

Recently, so called artificial neural networks (ANN:s) have been successfully applied to especially image classification [3]. ANN:s take inspiration from how the animal brain is structured and consists of several connected neurons which take input from and give output to each other. In this view, the ANN can be thought of as a directed graph, and standard graph terminology applies (neurons are nodes, and the output of a neuron is an edge). Each neuron has a set of weights associated with all of its inputs - i.e. the edges are weighted - which are updated during training.

In a feed-forward ANN, the neurons are organised into layers, where each layer only takes input from the previous one. The neurons in the first layer receives data as input, and the neurons in the last layer produces the output of the program. The layers in between are called hidden and are often densely connected, i.e. they receive weighted input from all nodes in the previous layer. As described, since the output of each neuron is simple a weighted sum of the previous layer, the whole network could be replaced with a single layer. For this reason, the output of a neuron is passed to an activation function,

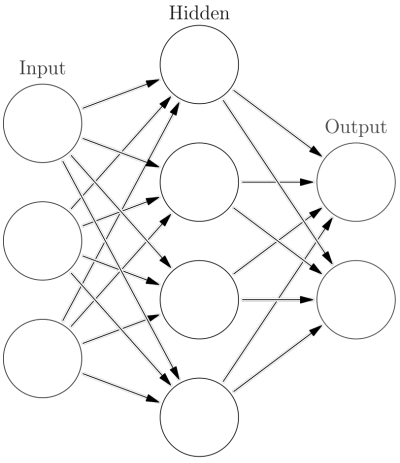

Figure 2.1: An artificial neural network.

which introduces non-linearity and allows the solving of problems which are not linearly-separable. There are many types of activation functions, as the only thing required of them to introduce non-linearity is that they not be linear. For ANN:s, one of the most common ones is ReLU - short for rectified linear unit- which despite its simplicity is enough break non-linearity.

$$ReLU(x) = max(0, x)$$

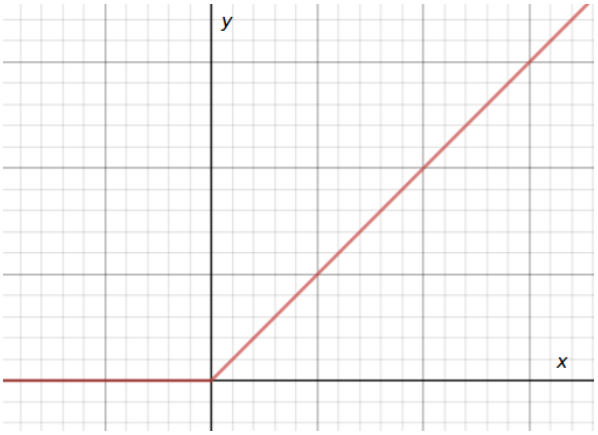

Figure 2.2: The ReLU activation function

The learning itself is done by minimising what is known as a loss function. The loss function can for regression, for instance be the mean squared error (MSE) or the mean absolute error (MAE) of the predictions. To actually minimise this quantity, the error is allowed to propagate backwards, performing a kind of gradient descent on the problem space using each layers' derivative.

This process is known as backpropagation.

One thing to be mindful of when optimising is the amount of training data available. If a capable model is fed too little data, it may overfit - that is, it will learn patterns specific to the training data itself which will not generalise. This can be prevented by either applying methods for simplifying the model, or by acquiring more training data.

2.1.1 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of deep neural network which has during recent years been one of the go-to method for image analysis. They were introduced in 2012 by Krizhevsky, Sutskever, and Hinton [3] when they showed an impressive improvement over existing methods on imagenet dataset. They are inspired by animals' visual systems, where neurons in the visual cortex respond to input only in specific parts of the image.

The basis of a CNN is the convolutional layer. A 2D convolutional layer receives 2D input data (e.g. an image) with one or more channels (e.g. RGB colour channels), and outputs 2D data [1]. The layer has one or more filters of a given size, which are matrices where the elements are weights. These filters "slide" across the input and produce a weighted sum of the sub-regions of the input. Each filters' output is a separate channel of the output of the layer. This sliding operation allows learning to happen with very few parameters - for each filter, the amount of learnable parameters is equal to the size of the filter. For instance a 3x3 filter has 9 parameters per channel regardless of the input size, which allows deeper networks to be used without a large penalty to training time. Another key benefit of sliding is that it allows detecting features wherever they occur once the weights have been learned. This is especially important for image classification, as it often does not matter where in the image the object of interest is.

Convolving an image reduces its size (depending on the size of the filters), whereas applying more filters increase the amount of channels. This process can be repeated using more convolutional layers until the output is just a 1x1 image with a large number of channels, each representing the presence of some feature in the original input. At this point the output can be flattened (reducing the dimensionality from 1x1xN to just N) and then densely connected to an output layer.

Other types of layers typically used for CNN:s are 'pooling layers', where the size of the image is reduced by replacing elements with for instance the average or the maximum of their neighbours. Pooling layers provide invariance

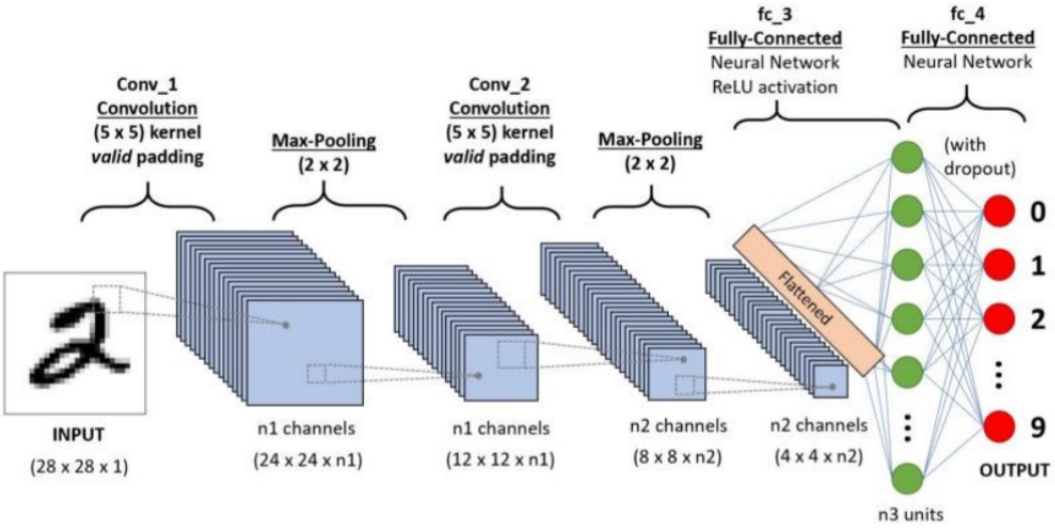

Figure 2.3: An example of a convolutional neural network layout.

to small translations. For chess, we determined this was not appropriate as the value of a piece is hugely dependent on where it is situated.

2.1.2 One-hot Encoding

There is a large variety of different input data used for machine learning problems, but they can be classified into two larger groups: numerical and categorical data. Numerical data is just a number representing some quantity, e.g. the luminosity of a certain pixel in an image. Categorical data represents a category in the input, for instance country of origin for a product. Since neural networks work with numbers, the categorical data has to be converted to numerical data in some fashion. The most intuitive way, assigning each category a separate number (Sweden = 1, Japan = 2, USA = 3, etc.), unfortunately does not work well as it causes higher-valued categories to be considered more important. Also in this representation, the average of Sweden and USA would be Japan - which is nonsensical.

For these reasons, so called one-hot encoding is used. The categorical data is replaced by a binary values, one for each category, where the category of the input is set to one and all others are set to zero. Using the averaging example again, the average of Sweden and USA would now be half-Swedish and half-American, which makes intuitive sense.

2.2 Chess

Chess is a game played by two players, controlling white and black pieces respectively, on an 8x8 board with six different types of pieces. Each piece moves in a specific pattern and can capture other pieces, and the objective of the game is to checkmate the opponent, which is when the opponent has no move which prevents the capture of their king. A player can resign when they consider the game to be hopeless and do not wish to play on. The game can also end in a draw, the most common ways being three-fold repetition where a game can be declared a draw by either player upon the third repetition of a previous board state, and stalemate, which occurs when a player has no legal move but is not checkmated [4].

To facilitate checkmating the opponent, winning material - capturing the opponents pieces without losing your own - is often beneficial, as more pieces allows you more flexibility in your attacks. At higher levels of play, even losing a pawn, the least valuable piece, is often enough to lose the game, and losing any piece more valuable than that often leads to resignation. Each piece is often assigned a numerical value - pawns are worth one point, knights and bishops 3, rooks 5 and the queen 9. This way, the difference in material between the players can be compared when the players don't have the same types of pieces left. This type of measuring forms the basis of many chess engines, which for instance will evaluate a board position to roughly +3 if white has won a knight. Chess engines do look several positions ahead, however, so the evaluation of the position is also affected by whether it thinks a player can win more material in the near future etc. A forcing way to win more material in chess is referred to as a 'tactic', and recognising these is a key element to evaluating the current position.

Chess engines can also forgo numerical evaluation of a position, instead outputting for instance "#3" if white has a checkmate in three moves, and "#-3" if black does.

Stockfish is one such chess engine, and has been ranked the world's strongest chess engine several times. It was defeated by AlphaZero [2] in 2017, a neural-network based chess engine which learned entirely by self-play, and recently by Leela Chess Zero, an open-source adaption of AlphaZero and its predecessor AlphaGo Zero. This has shown that ANN:s are more than capable of learning to play chess. Worth noting is that all state-of-the-art chess engines play at superhuman levels.

2.2.1 Chess as an Image Regression Problem

The idea of the paper is to treat chess evaluation as an image regression problem and use a convolutional neural network to solve it. The board is already two dimensional, and the pieces are analogous to pixel values in an image. Unlike images, however, chess pieces are categorical data, not numerical, which is why a one-hot encoding is suitable.

Our encoding is inspired by Oshri and Khandwala [5]. To do a one-hot encoding of the board state, each of the 6 types of piece is considered separately by generating one chessboard's worth of numerical values per type of piece. A "1" will denote that a white piece occupies the square, a "-1" a black piece, and "0" an empty square. Thus the chessboard becomes 3-dimensional with dimensions (8,8,6). The reason for using the same channel for black and white pieces is to avoid making the data unnecessarily sparse - already less than 1/32nd of the data is non-zero.

Using the image analogy, each of the six positions along the 'piece dimension' can now be considered a separate colour channel and the whole board fed to a 2D convolutional neural network.

2.3 Scope

Since the evaluation of a position is highly dependent on whose turn it is to move, we chose to limit ourselves to training the network from white's perspective (by only training on positions where it was white to move). This should make learning easier since we do not having to consider board rotations. If one needs to evaluate a position from the perspective of black, the colours of the pieces can be swapped and the board mirrored along the Y-axis to frame the position as being white's.

We also chose to ignore other ways of ending the game than checkmate - for instance stalemate, and draws by repetition. Stalemate was avoided since a draw is typically evaluated as zero, which would make it hard to distinguish from positions with evaluation around zero which were not stalemate. In order to account for draws by repetition, the neural network could receive as input the number of times the current position has been played already, but an issue with that is that the game is not drawn automatically at three moves. The ANN would thus need to learn that the position is drawn only if it thinks the position is losing otherwise, which would increase the difficulty of learning.

Chapter 3

Methods

3.1 Training Data

The April 2019 database from https://database.lichess.org/ was used to generate training samples. The original database contained over 33 million games, including several million of which has Stockfish evaluation of the positions. Stockfish version was 10+, evaluated until depth 22. After removing games which did not contain evaluation as well as non-real-time games, roughly 2 million games remained.

Training samples were generated from the first 100000 games at each point when it was white to move. The evaluation of the position as well as a one-hot encoding (see below) of the board position constituted one sample. In total, 310690 samples were generated.

Since the goal was to do regression, non-numerical evaluations (checkmate, checkmate in X moves) where replaced by numerical values. Checkmate was given a value of +/- 255 for checkmates by white respectively black, and forced checkmate in a certain number of moves was set to +/- 127. The evaluation of other positions were capped at +/- 63 with the reasoning this would make it easier to learn to distinguish the classes.

3.2 Model

The neural network was implemented using tf.keras, TensorFlow's implementation of the Keras API specification.

The model had four 2D convolutional layers. The first three had kernel size 3 by 3, and the last one 2 by 2. The number of filters used were, in order, 8, 16, 32 and 64. All these layers used ReLU-activation. The output of the last

convolutional layer was then densely connected to the output node. This type of layout was chosen after doing preliminary training on smaller datasets and finding its performance to be good in comparison to other layouts.

Mean squared error was used as the loss function, and Adam was chosen as optimiser. 20% of the training samples were used as test dataset, and the remaining for training. Among those used for training, 20% was used for validation.

Training took place over 19 epochs, stopping once improvement in MSE had been observed in the last 5 epochs. As no GPU was available, training took place on the CPU which limited the amount of training which could be completed within reasonable time.

Chapter 4

Results

The final results for the model on the training dataset after the 19th epoch can be seen in table 4.1. The results on the test dataset can be seen in table 4.2. Figure 4.1 shows the loss over all epochs, and figure 4.2 shows the MAE. Figure 4.3 shows a plot of predictions vs actual values with a line, fit using least-squares method. Figure 4.4 is a close-up of the central cluster of 4.3.

Lastly, figures 4.5, 4.6, 4.7, and 4.8 are examples of chess positions with their real Stockfish evaluations and their predicted values.

When doing preliminary training on smaller datasets (roughly a tenth of the samples), the model never overfit the data, even when training for extended periods of time.

Loss	MAE	Val Loss	Val MAE
785.65	12.19	878.31	13.32

Table 4.1: Final loss (MSE) and MAE for training and validation datasets.

Loss	MAE
863.48	12.18

Table 4.2: Loss and MAE for the test dataset.

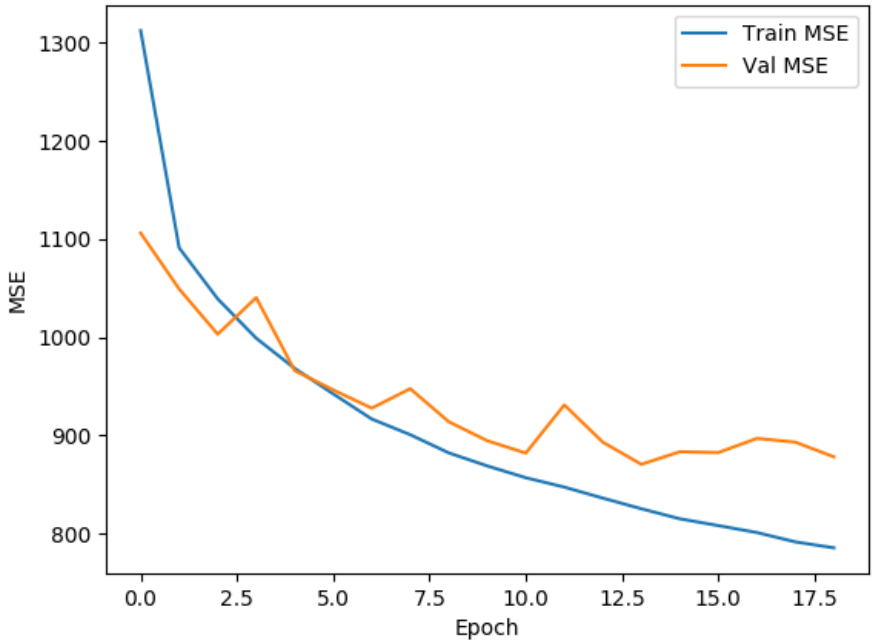

Figure 4.1: Loss of training and validation datasets over the epochs.

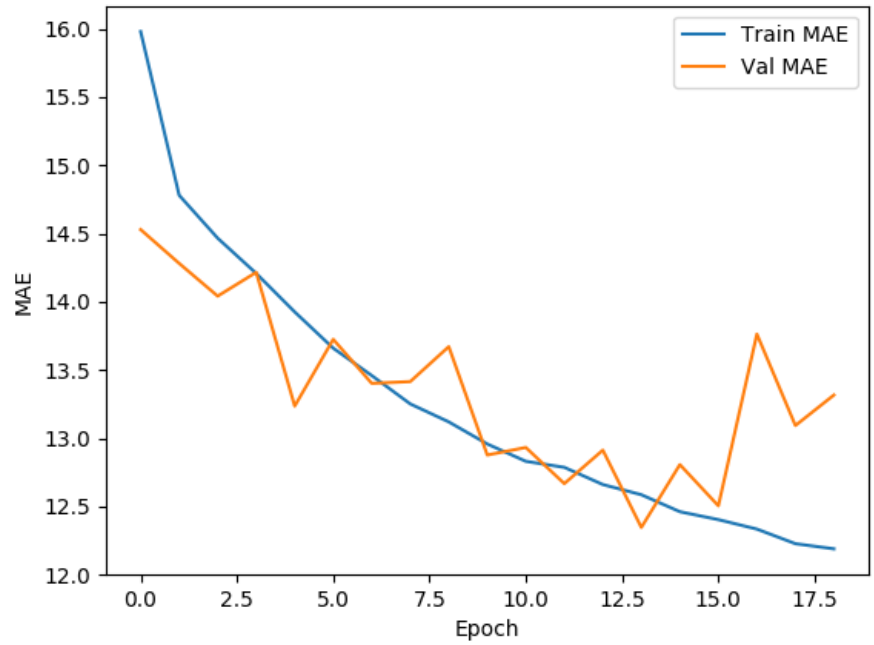

Figure 4.2: MAE of training and validation datasets over the epochs.

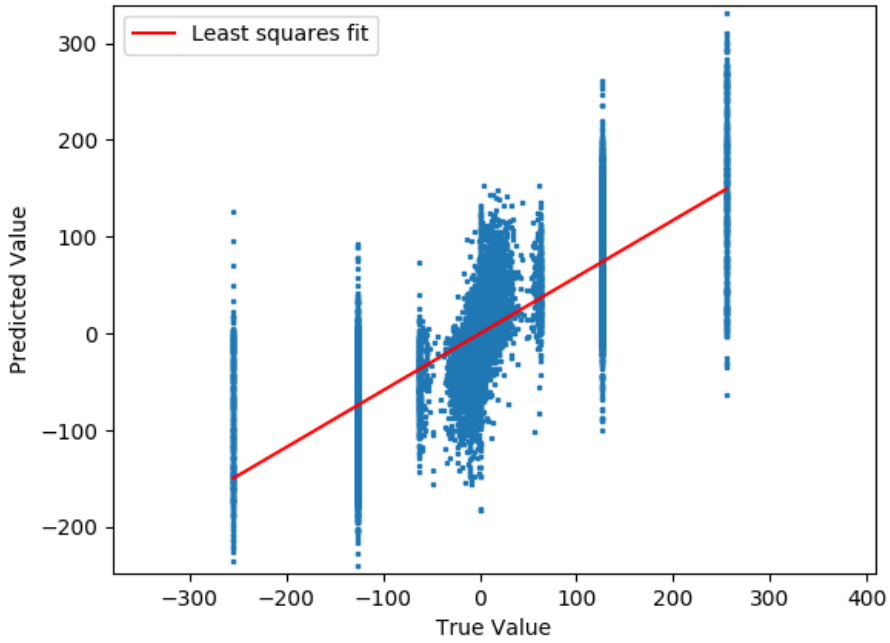

Figure 4.3: Predictions on test set vs actual values, and a least-squares line to show the general trend.

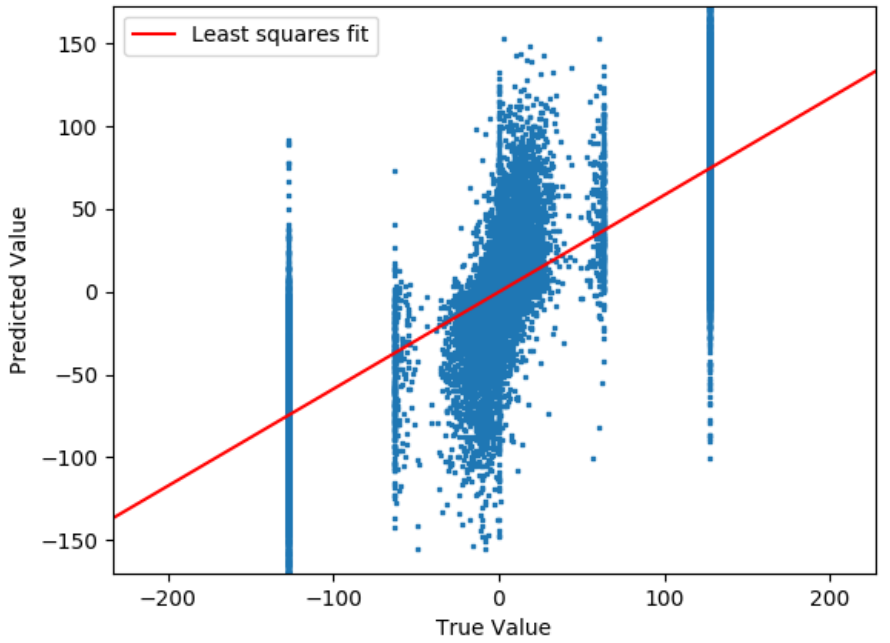

Figure 4.4: Close-up of central region of 4.3.

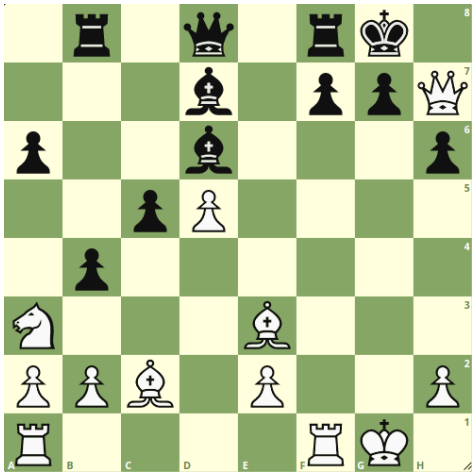

Figure 4.5: Checkmate. Predicted value 271.69.

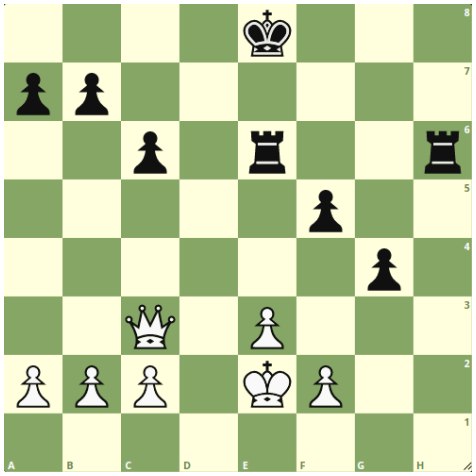

Figure 4.6: Real evaluation +0.5, predicted -5.95.

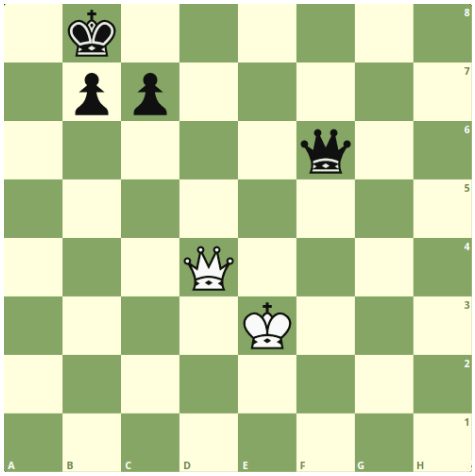

Figure 4.7: Real evaluation is checkmate in 23. The model predicts 1.11.

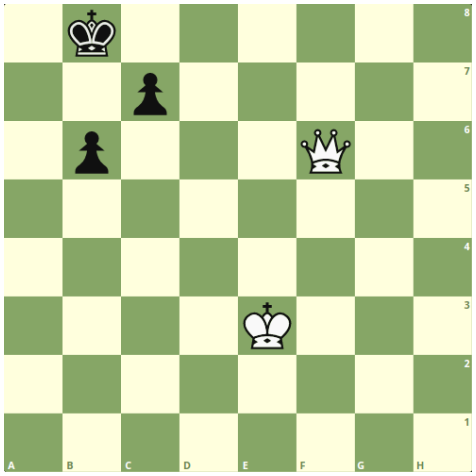

Figure 4.8: One move each after 4.7. Real evaluation is checkmate in 8. The model predicts 271.69.

Chapter 5

Discussion

Overall, we are satisfied with the results of the model. A mean absolute error of 12.18 (table 4.2) on the test set when the data is ranging between negative and positive 255 is quite good.

Looking at figure 4.3, checkmates and checkmates-in-X-moves are visible as vertical bars. The predictions for these classes demonstrate fairly high variance, but the general trend is correct. Notice that a checkmate for white was never interpreted as a checkmate for black and vice versa. Even checkmate-in-X positions had quite good results, especially considering how those often rely on very specific combinations of moves.

Looking at the close-up (figure 4.4), we can see that even the central cluster has a positive trend, though of very high variance. Another thing to note is that these positions were never predicted to be checkmates, with their predictions ranging from roughly -150 to 150.

Looking at specific examples of good vs poor performance, the model seems especially good at counting pieces, which is typically a good indicator of which player is doing better. However, the model fails to detect even rudimentary tactics which is illustrated in figures 4.7 and 4.8. In the first one, the model fails to see that black's queen can simply be captured and thus evaluates the position as almost equal. In the second figure, one move later and now being up a queen, the model agrees that this is indeed very good for white, even predicting a checkmate already.

One thing worth noting is that for usage in a chess engine, the evaluation taking on a specific value for a specific type of position (e.g. 255 for checkmate by white) is less important than that the better positions are more highly rated than the worse ones (e.g. checkmate by white should be evaluated higher than checkmate by black). This is because were one to use it in a chess engine,

positions would only be compared to each other, so only the ordering between them would matter.

The main reason for the model's shortcomings is that it is too weak. Looking at figure 4.1, even at the final epoch the training loss is continuing to decrease, whereas the validation loss has levelled off. This suggests that in order to improve the accuracy, a more complex model is needed. Whether or not a more complex convolutional neural network (more layers using smaller kernels or more filters) could perform significantly better, is unknown, but regardless that would comes at an increase in training time and hardware requirement, which were limiting factors for this report.

The results do imply some general knowledge for the application of convolutional neural networks on categorical data. For instance, it seems that patterns can be learned, as demonstrated by the correctly identified checkmates, and that overall prevalence of different types of categories can also be accounted for. However learning specific low-level patterns in the data seems harder, as demonstrated by the inability of our model to recognise specific tactics and combinations.

5.1 Future Work

An obvious next step would be to acquire better hardware and train for a longer period of time, using the layout presented here but mostly experimenting with more complex and larger models. Examples of such models include:

- Using different sized kernels, which leads to a different number of convolutional layers.
- Using more filters throughout the model.
- Using more densely connected layers.
- Experimenting with pooling.
- Using different stacks of convolutional layers with different sized kernels simultaneously to try targeting different sized features in the original data.

The author does retain the view that learning tactics should a be possible task for a CNN, as tactics are combinations of low-level features of the pieces' placements, and that is something CNN:s are typically good at. Whether or the fault lies in the model layout or the training data is unclear - it is perhaps

possible this layout could learn tactics if it were presented with more carefully chosen board positions.

Learning tactics is also related to handling mate-in-X positions. The only difference between such a position and a regular one, after all, is that in the former there is a forcing sequence which leads to checkmate (a tactic).

Finally, it would be an interesting extension to implement an actual chess engine around our trained model. It's understanding of chess could then be evaluated more easily by simply playing against it.

Chapter 6

Conclusions

We trained a convolutional neural network to evaluate chess positions using publicly available data of chess games analysed by the Stockfish chess engine. The trained model showed some general understanding of what constitutes a good position, but we conclude that a more capable model in combination with more training time is needed to accomplish anything impressive.

Bibliography

[1] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285* (2016).

[2] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[4] *FIDE Laws of Chess*. URL: `https://www.fide.com/fide/handbook.html?id=208&view=article`.

[5] Barak Oshri and Nishith Khandwala. *Predicting moves in chess using convolutional neural networks*. 2016.

TRITA-EECS-EX-2019:377