A CNN Reflex Agent for Chess

Vasco Portilheiro
CS 221

December 17, 2016

1 Introduction

1.1 Motivation

The main goal of this project was to write a chess engine. This, of course, has been done before; the motivating factor was how this engine would make decisions differently from more traditional approaches. As opposed to defining many hand-crafted features and designing an optimized state-search algorithm around them, the purpose of this project is to see how well a computer can learn to play chess from so called "first principles." In other words, the goal is to have an engine which, given only the knowledge of the basic rules of play, can be trained to recognize the good or "correct" moves to make. Viewed as such, the input-output behavior can simply be viewed as this: given a position and player whose turn it is, pick the best move for the player to make. (We note that, of course, "best" is not necessarily objective, and the evaluation will be discussed further later.)

Such an approach if possible, would have a twofold benefit. Firstly is the advantage common to most if not all applications machine learning, which is reducing engineering effort required for big advances in quality of results. Second is the fact that such an agent would not use lookahead (at least during actual gameplay), thus saving on computation time.

1.2 Inspiration: AlphaGo

Convolutional neural networks have recently been successful in developing agents that learn play the game of Go, without any preconceived notions or hand-coded heuristics — the success story coming from Google Deep Mind's AlphaGo program. This mindset is in fact the same one with which this project was approached. Therefore, the technical approach used for AlphaGo was a starting point for the this project.

To understand the context for this project's approach, we provide a very brief overview of AlphaGo, for those unfamiliar with it. The training occurs, broadly, in two steps. First, there is a supervised learning step, in which a policy network is trained to predict the moves made in top-level professional games, as a probability distribution over possible moves. The second step involves initializing a second network — the reinforcement learned policy —- and having it play itself, performing deep Q-learning with Monte-Carlo tree rollouts. The combination of these steps lead to superhuman level play previously thought nearly impossible to achieve by computers.

1.3 Task Definition for Chess

In order to make this project manageable in the scope of the class, it was chosen to focus on the first step of creating an AlphaGo-like chess agent, which is creating a policy network which is trained to predict human ex-

pert moves. In other words, task is as follows: given a board state from an existing expert game, have the program predict probability distribution for the move that was then made.

Note that this is different from having the program attempt to play optimally. If we make this argument, the assumption being made is that on average, grandmasters play close to optimally (we address this assumption later).

Any due diligence in research shows that this approach has indeed been applied to chess by Oshri and Khandwala, and they provided a key insight into how to overcome a modeling challenge of CNNs for chess. Therefore, this project serves both as the first step to an "AlphaChess," and as an opportunity to reproduce, verify, and possibly expand on the findings of Oshri and Khanwala.

2 Approach

2.1 Baseline and Oracle

As a baseline and oracle for the task, we use two algorithms to represent the most basic and most advanced traditional, search-based approaches. This will allow us to comparatively place the simple policy network, which uses no lookahead, on a spectrum within existing and more conventional methods.

The baseline was implemented as a simple depth-limited minimax search. The evaluation function was used at depth limit two, and was one based of piece values and mobility, as first described by Claude Shannon in 1949. The baseline had the disadvantage of being slow in calculation, and was generally weak, winning zero out of ten games played against a proficient but amateur level human opponent.

The oracle used was the open-source chess engine Stockfish. At a reasonable, decently high setting, Stockfish would always beat an amateur player. This gives it a high value as an oracle for the highest level of the project's goal, which for a program to play well, but does not necessarily translate directly to the task, which is predicting moves from given games. Therefore, part of the experimentation includes a test of Stockfish as a predictor.

2.2 Advanced Method

2.2.1 Challenges of applying CNNs to Chess

While conventionally speaking, Go is a "harder" game for computes to play than chess, this wisdom comes from trying run search algorithms through the game's state-space; in fact, chess provides its own set of challenges which in a sense makes it a harder game to which to apply CNNs than Go.

The first such challenge which we will describe is the complexity of the board. The dimension in terms of which this holds is having many different types of pieces, which Go lacks. Because of this, applying CNNs visually, to the image of a board, is impractical, and better representation of the game state must be found.

The second challenge is that addressed by Oshri and Khandwala, which is the move-space. While Go does have a larger board than chess, the moves only have two dimensions, since they consist of placing a stone at some coordinates; the move-space is therefore 19^2 In chess, however, moves are four dimensional one must first choose a starting square (which we will call the source) from which to pick up a piece, and then an ending square in which to place it (which we will call the target). This makes the move-space 8^4, which is an order of magnitude larger than 19^2.

We will address each of these challenges in turn in the following sections.

2.2.2 Bitboards for CNNs

The approach of using bitboards for chess is not novel. The idea of having a collection of

64 bit arrays, where each bit represents some piece of information about the board, has be in use for some time, as it greatly speeds up certain calculations usually done by a chess engine. Such bitboards might include a bitboard for all the white pawns, which in the starting position would look as follows:

$$\begin{bmatrix} 0,0,0,0,0,0,0,0, \\ 0,0,0,0,0,0,0,0, \\ 0,0,0,0,0,0,0,0, \\ 0,0,0,0,0,0,0,0, \\ 0,0,0,0,0,0,0,0, \\ 0,0,0,0,0,0,0,0, \\ 1,1,1,1,1,1,1,1, \\ 0,0,0,0,0,0,0,0 \end{bmatrix}$$

Fig 1. White-Pawn Bitboard

It is not hard to see that such representations could easily provide inputs to neural networks. In order to represent the whole board, these individual piece bitboards must somehow be combined.

An intuitive approach would be to stack all the piece bitboards into a three-dimensional tensor — which notably, has the same dimensionality as an image, the usual input to a CNN. However, using different bitboards for each color would make the data unnecessarily sparse. Instead, bitboards may represent each piece type with both colors present, making the resulting representation a 6x8x8 tensor, or alternatively, an 8x8 6-channel "image." The question remains as to how to distinguish the two players' pieces. At least two basic approaches are possible. One is having one or additional bitboard layers representing whether the piece at any location is friendly or the opponent's. The second, used by Oshri and Khandwala, it to have the opponents pieces be represented by a -1. Due the constraints of this project, the latter was chosen; however all three options should explored in further experimentation, with the intuition that simply having one extra layer for friendly piece, or an absolute difference of 2 in value between friendly (+1) and opponent (-1) pieces, may not be as structurally informative as possible.

2.3 Source and Target Probability Networks

2.3.1 The Idea: Splitting up the Problem

To address the large dimensionality of the move-space, we turn to the solution used by Oshri and Khandwala, which is to create two separate types of networks. One, the source network, predicts the source given a gamestate — that is, given a board, it will pick what square it is good to move a piece from. The second kind of network, the target network for any given kind of piece, gives a value for how good it would be to have that type of piece at some position on the board. For example, the target network for bishops might have a high value (close to 1) on the square c4, indicating that if possible, Bc4 could be a good move prediction. Note that there are six target networks, one for each piece type.

By splitting up the problem of making a move into two components, we now have two problems, each with an output space of only 64, rather than 4096. While the prediction from the two kinds of network can be composed in order to make a move, using the policy:

$$\pi(\text{board}) = \underset{(\text{start},\text{end})\in\text{squares}}{\text{argmax}} \; p_{\text{source}}(\text{start}) p_{\text{pieceAt(start)}}(\text{end})$$

we note also that this is completely unnecessary during training, since each network can be trained independently.

2.3.2 Implementation Details

The networks themselves were implemented to have the same architecture as those used

by Oshri and Khandwala, in order to make comparison more meaningful. The networks had three layers, consisting of one convolution layer with ReLU activation, and two affine layers, with a ReLU activation and softmax output. ReLU was used due to preference both by the AlphaGo and Oshri and Khandwala papers, while softmax simply follows from the idea of outputting a probability distribution.

Optimization was done with the RMSprop algorithm, to increase convergence speed over other methods such as normal SGD. Due to the nature of the task of each, which is similar morphologically to a multiclass classification, categorical cross-entropy was used as the loss function, in order to avoid "stalling out" of the training.

3 Data and Experiments

3.1 Training and Testing Data

The games used in this project are all grandmaster games, pulled from a database of about 0.5 million games available for free from `Chess-DB.com`. Of these, 10,000 were used for hyperparameter tuning, and another 15,000 were used for each experiment.

The games themselves were prepossessed to create input-label pairs, where the inputs are board-tensors representing the game state (as described in 2.2.2), and the outputs are 64 entry one-hot arrays with a 1 in the entry corresponding to the correct square. That is, in the source network labels, the 1 corresponded to the square the move in the game was made from, and for each target network, the 1 corresponded to where the move was made to.

3.2 Experimental Results and Analysis

3.2.1 Oracle Accuracy

In order to test how well move prediction corresponds to good play, Stockfish was used tested in predicting the moves from the grandmaster game dataset. Running Stockfish on 10 of the held out 10,000 games gave a move predicting accuracy of 51.2%. This is rather surprising, suggesting that perfect play is fairly defined for any given state, given the schools of thought used in chess today, since people play like the engine (or conversely, the engine plays like people). It also suggests that the metrics found in the final experiments below suggest that a fully trained "AlphaChess" could indeed achieve expert level play.

3.2.2 Weights and Hyperparameters

Informal experimentation showed that initializing the weights to 0 would lead to a convergence in all network to an accuracy measure of around 4%. Changing the initialization to a uniform distribution improve results drastically, leading to accuracies such as those described below in final results, of over 30% for all networks.

Tuning of hyperparameters was performed on the learning rate and decay parameters of RMSprop, with values of [0.001, 0.0015, 0.005] and [0, 0.2, 0.4, 0.8], respectively. Each combination with run for 10 epochs over the reserved data set of 10,000 games. The best performance was achieved with a learning rate of 0.001 and no decay, giving an accuracy of 36.13%.

This is interesting to contrast with the findings of Oshri and Khandwala, whose best results were at 0.0015 learning rate and 0.999 decay (the decay metric is especially surprising), trained in 15 epochs rather than 10. These metrics gave only around 30% accuracy in the hyperparameter training. Number of epochs does not seem to be the issue, since the learning rate used was lower than that of Oshri and Khandwala. Nevertheless, the findings of both this project and the former suggest high sensitivity to these hyperparameters.

3.2.3 Final Experiment

In the experiment, all 7 networks were trained on 15,000 games for 10 epochs. At the end of the training, the following validation categorical-accuracies were obtained:

Source 0.40
Pawn 0.53
Knight 0.58
Bishop 0.49
Rook 0.32
Queen 0.30
King 0.54

Reported in terms of accuracy and validation accuracy (non-categorial, and thus less optimistic), we have the following training histories:

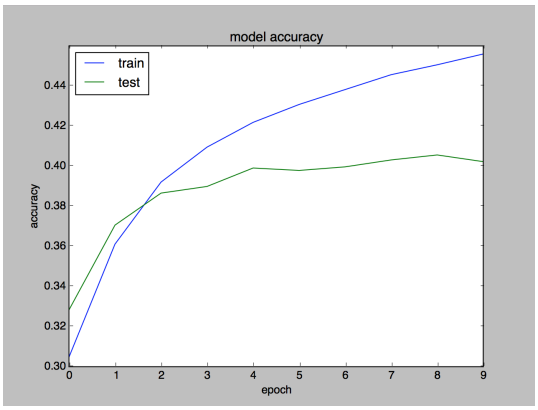

Fig. 2 Source network

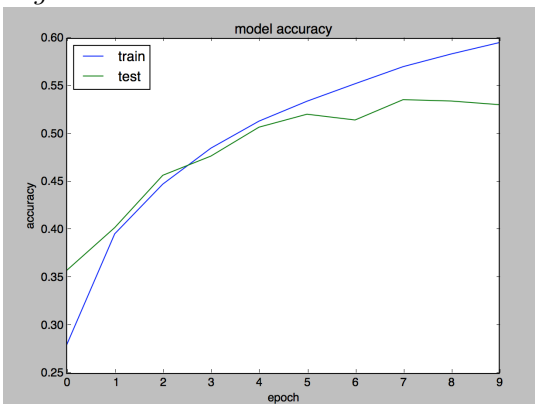

Fig. 3 Pawn target network

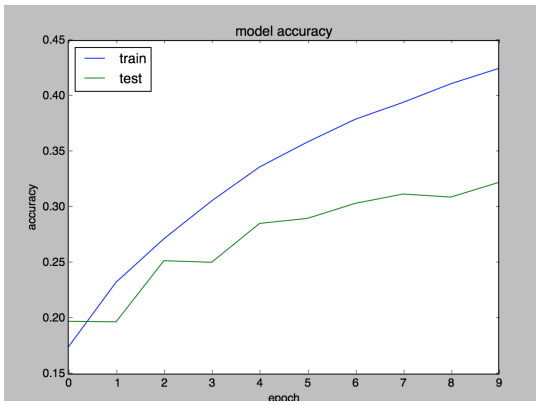

Fig. 4 Knight target network

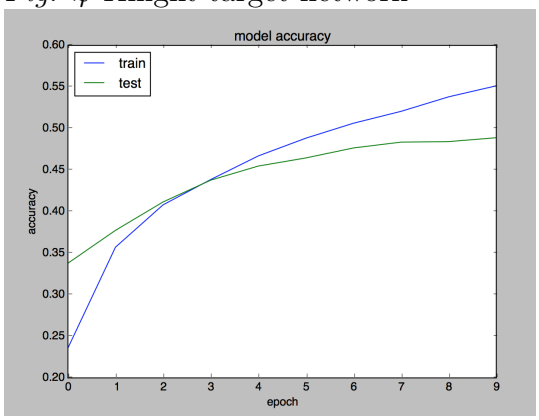

Fig. Bishop target network

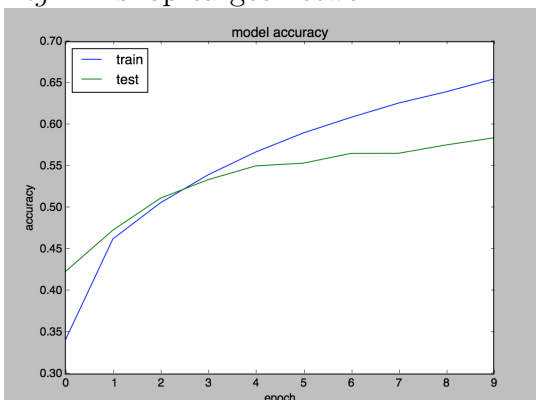

Fig. 6 Rook target network

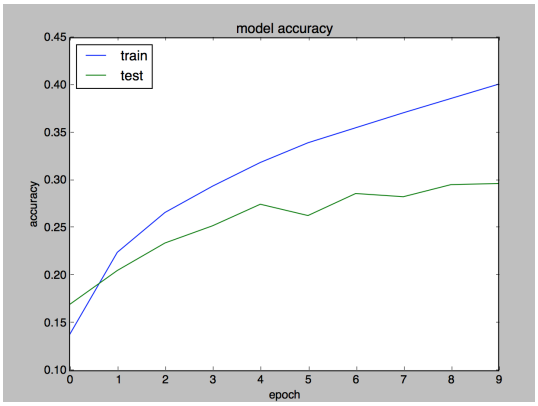

Fig. 7 Queen target network

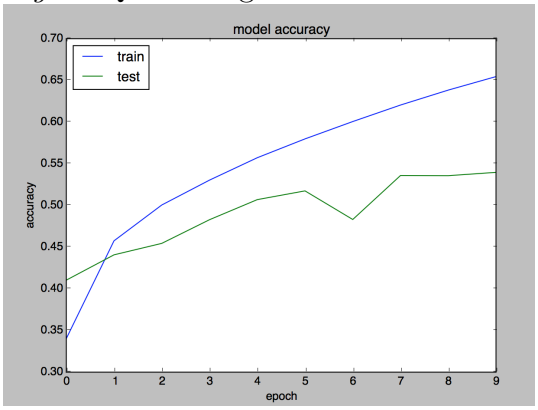

Fig. 8 King target network

These findings confirm the findings of Oshri and Khandwala that pieces that can move across the board are harder to predict. This is most likely due to the local nature of CNNs. Thus, we may conclude that CNNs are best suited for local "pattern-recognition" type move search.

Another point to note is that there is a consistently large discrepancy between accuracy and validation accuracy, suggesting some amount of overfitting. This could potentially be solved in future experiments with some form of regularization.

4 Conclusion

This project's results are ambiguous in terms of the actual task defined in the introduction. The patterns in the piece accuracies suggest that predictions are better for "locally moving" pieces, which confirms the local nature of CNNs. However, the fact that the total accuracies resemble those made by a high-level chess engine suggests that training the networks using reinforcement learning could prove to be a good way of training an effective chess engine.

5 References

Barak Oshri, Nishith Khandwala. Predicting Moves in Chess using Convolutional Neural Networks

Erik Bernhardsson. Deep learning for... chess. 2014.

Matthew Lai. Giraffe: Using Deep Reinforcement Learning to Play Chess. 2015.

Silver, et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016.

James D. McCaffrey. Training a Deep Neural Network using Back-Propagation. 2016.