CMSC 352: Final Project Report

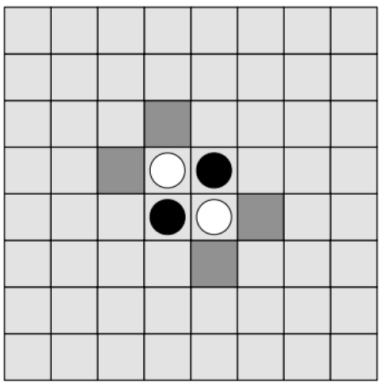
Aleksander Fedchin

December 21^{st}, 2018

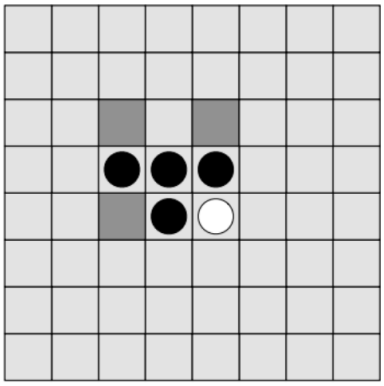
1 Introduction

This project presents a program that can solve (i.e. find the winner in a game where both players play perfectly) a version of Reversi board game for the 6x6 board (the solution can be extended to the standard 8x8 board by changing some constants in the code). This simplified version of Reversi was first solved by J. F. Feinstein in 1993 and he reports that it took his program approximately 1.5 weeks to run.[3]. My program does not improve on these results significantly, given that I have much more computational power, but it does solve the problem in 6 hours (Feinstein also reports the score with which white win, but he does not analyze all the possible openings, i.e. the score he reports is only the lower limit).

If for a certain board state a move that will lead the current player to victory is known, other possible moves don't have to be explored. Hence, one of the ways to reduce the computation time of the analysis is to make the program explore moves that are more likely to lead to victory first. The major underlying problem is that it is difficult to obtain datasets of states for which the winner (score) is known and those datasets that are available are not very reliable, since they are used under an assumption that during each of the games the players played perfectly, which is not the case. Engel[2] proposes a method for constructing alternative training datasets. He uses minimax to built a dataset of end-game states, for which the actual winner can be easily calculated. Afterward, a model (an SVM or Linear/Logistic Regression in Engel's case) is trained on this dataset of end-game states. Then, a dataset of states closer to mid-game is created by minimax algorithm, which employs a model trained before to generate scores for its leaf nodes. The procedure is thus repeated until there is a model which can be used to generate scores for the minimax algorithm launched from the initial board-state. In this project, I use a NN model coupled with the approach that Engel proposes.



(a) Othello initial board state. Black to move.



(b) Board state after black's move. White to move.

Figure 2.1: Examples of Othello board states. The figures are from Liskowski et *al.* [1].

2 Reversi rules - quoted from Liskowski et *al.* [1]

"Othello is a perfect information, zero-sum, two-player strategy game played on an 8x8 board. There are 64 identical pieces which are white on one side and black on the other. The game begins with each player having two pieces placed diagonally in the center of the board (Fig. 2.1(a)). The black player moves first, placing a piece on one of four shaded locations, which may lead to the board state in Fig. 2.1(b). A move is legal if the newly placed piece is adjacent to an opponent's piece and causes one or more of the opponent's pieces to become enclosed from both sides on a horizontal, vertical or diagonal line. The enclosed pieces are then flipped. Players alternate placing pieces on the board. The game ends when neither player has a legal move, and the player with more pieces on the board wins. If both players have the same number of pieces, the game ends in a draw."

3 Methods

3.1 Features

In Reversi, it is possible that some disks become unflippable, i.e. it is known in advance to whose score they will contribute at the end of the game. A disk placed in one of the four corners is unflippable, since there are no two sides from which it can be enclosed. If a disk of the same color is placed

by the corner, it also becomes unflippable and so on. In mid- and end-game, the winner can often be predicted by the number of unflippable disks. The features I use are the configuration of the board (-1, 1, or 0 for each position corresponding to black or white disk or the absence of such), and the configuration of the board with only the unflippable disks considered.

3.2 Network architecture

I use a feed forward neural network with one hidden layer, validation set of 11% of the total dataset, and patience of 2. I initially planned to employ a more sophisticated CNN approach described by Liskowski et *al.* [1], but I chose a wrong ML library (weka), which does not allow any architecture other than the feed-forward one, as I realized post-factum. I should probably have chosen deeplearning4j library instead. I first train a model on a dataset of 25^{th} level board states (where "level" is the number of disks on the board, which goes from 4 to 36) for which the scores were directly assessed using minimax and alpha-beta pruning. I then build two new models on datasets of 18^{th} and 20^{th} level states by using the old model as an evaluation function for leafs in minimax search. The latter two models can then be used by the program to decide which moves to explore first (e.g. if the decision has to be made on level 5, the program uses minimax and the 18^{th} level model)

3.3 Evaluation

I evaluate my classifiers by converting the scores (which go from -36 for a board covered with black disks to 36 for a board covered with white disks) to three classes (black wins, white wins, truce) and calculating the weighted f-score between the actual labels and those predicted by the program. I chose to weight the f-score because the number of board states that lead to truce is very low and the unweighted f-score would suffer unreasonable penalties for misclassifying the truce states.

3.4 Accelerating the algorithm

Because speed is of high importance to this project, I employ different ways to accelerate the program. First of all, once the program finds the winner for a particular board-state, this information is recorder in a hash table. This way, the program does not have to recalculate the same result twice. Secondly, I wrote a method that assigns a code to each state so that no two states have the same code, unless they are rotations or reflections of each other (getCode() function in BoardState class). If a rotation or a reflection

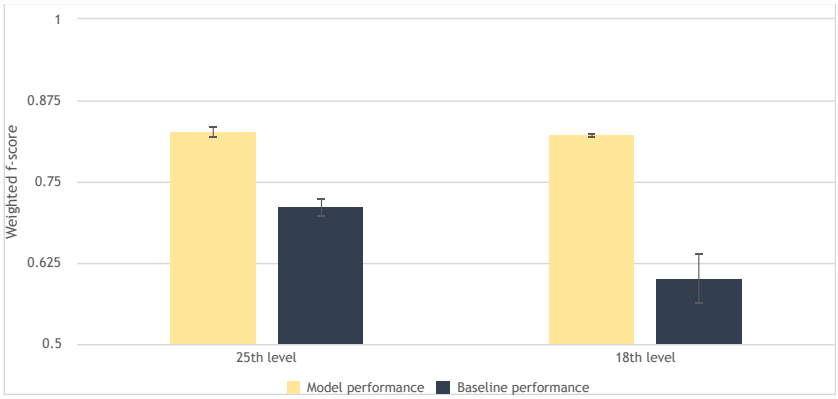


Figure 4.1: Average performance of the baseline system and the model on the datasets of board-states at levels 18 and 25. The averages are over 3 runs on 3 different datasets for each level. The 25th level datasets include 10 000 board states each. The 18th level datasets include 3 000 board states each. The values of the board states on the 18th level were assessed by using one of the 25th level models as an evaluation metric for the minimax. The error bars represent standard deviation. The P-values are (paired T-test): 0.0038 for level 25, and 0.006 for level 18.

can be used to go from one state to the other, the two states will always have the same code. This is useful because states with the same code will always lead to the same player winning. Hence, codes are stored in the hash table instead of the states themselves. I recorded the number of times the hash table was successfully used at each level and my estimate is that the aforementioned approach speeds up the process by a factor of 2. Finally, I employ concurrency to evaluate a number of different board states at the same time.

4 Results

In order to make sure that the models I build achieve useful results I devised a baseline against which they could be compared. I compare the scores achieved by the models to the scores achieved when predicting the winner by looking at the number of unflippable disks only. As expected, the classifiers score much better, especially on the levels closer to beginning of the game, as can be seen from Fig. 4.1.

My program eventually solves the problem and accurately states that white wins.[3]

5 Conclusion

The program can be further developed so that CNNs are used instead of the regular feed-forward networks. While the program could potentially be used with the $8x8$ board, it seems that the current algorithm is not fast enough to solve the $8x8$ board in any reasonable amount of time.

6 Running the program

The program can be compiled and ran from the src folder in the following way (disabling stdout buffering allows to look at the log before the program finishes running):

```
javac reversi/*
stdbuf -oL java reversi.Main > log.txt
```

The dictionaries of evaluated states are saved to hashable[levelName].ser files after completion. The log will be updated with various information such as the effective branching factors (it only takes into account the states analyzed by the program), the minimal level at which a board-state was solved, etc. The log will also record all the states analyzed up to level 13 (this is a constant defined in the BoardState class). The program should finish running in approximately 6 hours. Datasets of evaluated board-states can be created by running the Predictor class:

```
java reversi.Predictor level evaluationLevel datasetSize
```

References

[1] Liskowski et *al*. (2018). "Learning to play Othello with deep neural networks" *IEEE Transactions On Computational Intelligence And AI In Games*.

[2] Engel, K. T. (2015). "Learning a Reversi Board Evaluator with Minimax" URL: http://www.cs.umd.edu/sites/default/files/scholarly_papers/Engel.pdf

[3] Feinstein (1993). "6x6 Othello" Newsgroup: rec.games.abstract URL: https://www.ics.uci.edu/~eppstein/cgt/othello.html