Playing Othello by Deep Learning Neural Network

YIP Tsz Kwan

in collaboration with CHAN Lok Wang

Supervised by

Dr. K.P. Chan

16 April 2017
Abstract

Board game implementation has long been a way of showing the ability of AI. Among different board games, Othello is one of the popular subjects due to its simple rules and well-defined strategic concepts. Today, the majority of the Othello programmes at a master level still require expert knowledge in their development. To choose the best next move from a given board state, the computer Othello searches through the game tree and return move (as a tree node) with the optimal value. Though, the evaluation function used to assign value to the nodes in a game tree relies on the feature studies by human experts.

This project aims to use a machine learning technique, deep learning neural network, to play Othello without any human logic. This implies the computer Othello has to learn the game from scratch through the past gameplays. The targeted winning rate of the computer Othello developed is 50% when playing against a moderate opponent, and 35% when playing against a strong opponent.

With different models being implemented and tested, the model with 3 convolutional layers and 3 pooling layers gave the best performance. It has a fair winning chance when playing against a moderate opponent, though its performance against a strong opponent was not as desirable.
Acknowledgments

I would like to thank my supervisor, Dr. K.P. Chan, for his continuous guidance and support throughout the project, and the second examiner, Dr. Chung Ronald.

I would also like to express my sincere gratitude to my teammate, Chan Lok Wang, Anna, for her effort and significant contribution to this project.

Thank you Ng Argens and Yu Kuai for sharing their data for the neural network model training, and Eric P. Nichols for making the source code of the Othello programme publicly available.

Last but not least, I would like to thank all the staff of the Department of Computer Science for their teaching and support throughout the years.
To my parents and sisters
# Contents

Abstract .............................................................. i
Acknowledgements .................................................. ii
List of Figures ....................................................... vi
List of Tables ......................................................... vii

1 Introduction ......................................................... 1
   1.1 Background ...................................................... 1
   1.2 Project Objectives ........................................... 3
   1.3 Organization of the Report ................................. 3

2 Existing Work ...................................................... 4

3 Methodology ....................................................... 6
   3.1 Training Data Collection and Processing ...................... 6
      3.1.1 Training Data Collection ............................... 6
      3.1.2 Training Data Processing .............................. 7
   3.2 Implementation Language .................................. 8
   3.3 Neural Network Design and Implementation ................. 9
      3.3.1 Overview of Neural Network .......................... 9
      3.3.2 Neural Network Design .............................. 10
   3.4 Decision-making Algorithms ............................... 12
      3.4.1 Minimax Algorithm .................................. 12
      3.4.2 Alpha-beta Pruning ................................ 13
   3.5 Testing ....................................................... 14
4 Division of Work

5 Results and Discussion

5.1 Neural network model 1: 5 Convolutional Layers and 3 Pooling Layers

5.1.1 Parameters specifications

5.1.2 Training Result and Analysis - Data Accuracy

5.1.3 Training Result and Analysis - Testing against a Computer Othello

5.2 Neural network model 2: 3 Convolutional Layers and 1 Dropout Layer

5.2.1 Parameters specifications

5.2.2 Training Result and Analysis - Data Accuracy

5.2.3 Training Result and Analysis - Testing against a Computer Othello

5.3 Neural network model 2: 3 Convolutional Layers and 3 Pooling Layer

5.3.1 Parameters specifications

5.3.2 Training Result and Analysis - Data Accuracy

5.3.3 Training Result and Analysis - Testing against a Computer Othello

6 Conclusion and Future Research

6.1 Conclusions

6.2 Future Research Directions

References

Appendix

A The game of Othello
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The initial board state of Othello</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>A board state and its corresponding representations</td>
<td>7</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of a neural network with three layers</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>A classical CNN architecture</td>
<td>10</td>
</tr>
<tr>
<td>3.4</td>
<td>Demonstration of the minimax algorithm</td>
<td>12</td>
</tr>
<tr>
<td>5.1</td>
<td>Data accuracy of neural network model 1-1</td>
<td>18</td>
</tr>
<tr>
<td>5.2</td>
<td>Data accuracy of neural network model 1-2</td>
<td>19</td>
</tr>
<tr>
<td>5.3</td>
<td>Test Result of neural network model 1-1</td>
<td>20</td>
</tr>
<tr>
<td>5.4</td>
<td>Test result of neural network model 1-2</td>
<td>20</td>
</tr>
<tr>
<td>5.5</td>
<td>Data accuracy of neural network model 2-1</td>
<td>22</td>
</tr>
<tr>
<td>5.6</td>
<td>Data accuracy of the first 30 moves in neural network model 2-2</td>
<td>23</td>
</tr>
<tr>
<td>5.7</td>
<td>Data accuracy of the last 30 moves in neural network model 2-2</td>
<td>23</td>
</tr>
<tr>
<td>5.8</td>
<td>Test Result of neural network model 2-1</td>
<td>24</td>
</tr>
<tr>
<td>5.9</td>
<td>Test result of neural network model 2-2</td>
<td>24</td>
</tr>
<tr>
<td>5.10</td>
<td>Data accuracy of neural network model 3-1</td>
<td>26</td>
</tr>
<tr>
<td>5.11</td>
<td>Data accuracy of neural network model 3-2</td>
<td>27</td>
</tr>
<tr>
<td>5.12</td>
<td>Test Result of neural network model 3-1</td>
<td>28</td>
</tr>
<tr>
<td>5.13</td>
<td>Test result of neural network model 3-2</td>
<td>28</td>
</tr>
<tr>
<td>A.1</td>
<td>Initial board position of Othello</td>
<td>33</td>
</tr>
<tr>
<td>A.2</td>
<td>A legal move in the game Othello</td>
<td>33</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Cell representation scheme .............................................. 7
3.2 Level of difficulties of The Othello ................................. 15
5.1 Parameters of two trained models .................................. 17
5.2 Parameters of two trained models .................................. 18
5.3 Parameters of two trained models .................................. 21
5.4 Parameters of two trained models .................................. 26
Chapter 1

Introduction

1.1 Background

In March 2016, computer programme AlphaGo shocked the world by winning 4 out of 5 games against the professional Go player, Lee Sedol, one of the best players at Go. The rapid improvement in a chess game engine came at a surprise to many experts of artificial intelligence, who once thought computer beating a human champion of Go would not happen in the next decade. [1] As AlphaGo outperformed a human champion of Go, it drove us to investigate the possibility of applying the techniques used by AlphaGo to play another board game - Othello.

Othello is another board game played by two players, who take turns to place their discs, in black and white, on the game board. Due to the smaller board size of and limited moves in each round of the game, Othello is considered as an easier game for AI to play. The development of Computer Othello is quite mature. In 1997, Logistello won against a human champion by 6-0 games.

Game-playing has long been an interest in artificial intelligence due to its complexity. The computer player is required to evaluate the current board state and select the best next move constantly throughout the game, as if a human player does. Hence, it becomes a way to test the ability of the computer and to evaluate how well a computer can perform against human.

Traditionally, a computer plays a board game by the searching through the game tree, a tree that contains all the possible moves with the corresponding weights. Due to the limited storage capacity of a computer and the huge computational complexity of the game tree search, for most of the games, it is impossible get the weights of each next state by propagating the final result of the leaf nodes back to the upper layers. As a result, an evaluation function is used to assign the weight to the possible moves in a tree. [2] The problem of how accurate an evaluation function can assign the weight to the given board configuration arises and has become the study focus for developing a game engine. The major technique used to generate an evaluation function in AlphaGo is by deep learning neural network, a machine learning algorithm.

Inspired by the human brain, neural network is a way of information processing. Several layers of nodes in the neural network are connected together. As the system is trained, the connections between the nodes will change accordingly. A large amount of training data is used to fine-tune the connection during the training process so that the network can
produce a specific output corresponding to the given input. A deep learning neural network is a network with many hidden layers. More features can be captured and studied in each hidden layer to increase the accuracy of the final result. A combination of the deep learning neural network and convolutional neural network (CNN), which explicitly assumes the training data as images, is used in this project.

This project aims to demonstrate how powerful deep learning neural network can be for gameplay by developing a computer Othello with board size being 8x8 using deep learning neural network. Othello is chosen as the size and the complexity of this game is suitable for a one-year long project with limited resources. The technique of deep learning neural network can also be applied to this game. More information on the rules of Othello can be found in Appendix I. When both players play the game perfectly, it appears that the game will very likely end with a draw. Therefore, the targeted winning rate when playing against a moderate opponent is 50% and that when playing against a strong opponent is 35%.

In general, there are two main types of neural networks for the game Othello - the policy network and the value network. The policy network outputs a probability distribution over the possible moves, while the value network evaluate the winning probability of the given board state. As the possible moves at each round of the game is limited, the implementation of the policy network is of a lower priority. Due to the time limitation, only the value network was implemented and tested for the computer Othello.

The main contribution of this project is to provide an alternative to build an evaluation function for a computer Othello without any human logic. While different algorithms of machine learning, such as reinforcement learning, have been used to train a computer Othello, at the time when this project was proposed, no research on training a computer Othello by deep learning neural network has been made available. Hence, the result and analysis of this project can serve as a reference for future development of computer Othello using deep learning neural network.
1.2 Project Objectives

The objective of this project is to develop a desktop computer Othello with the technique of deep learning neural network and the following attributes:

- The game board configuration size being 8 x 8;
- A winning rate of 50% when playing against a moderate (computer) opponent;
- A winning rate of 35% when playing against a strong (computer) opponent; and
- A user-friendly user interface (UI) for easy playing

While the ultimate outcome is to deliver the programme with the specification above, the key of this project is to construct the evaluation function of the programme by the programme itself via the value network without any human logic.

1.3 Organization of the Report

The organization of this report is as follows:

**Chapter 2** This Chapter describes the detailed background of the project, including the existing solution and the technical gap.

**Chapter 3** This Chapter explains the detailed design and implementation of the project, including the training data collection, the algorithm used, neural network design and game engine implementation.

**Chapter 4** This Chapter presents the work division among the team members.

**Chapter 5** This Chapter presents, analyzes and discusses the results of different neural networks trained.

**Chapter 6** This Chapter concludes the project, and suggests potential areas of improvement based on the current work, as well as future research directions.
Chapter 2

Existing Work

Othello is a strategy game played by two players in black and white discs. While many board sizes are available for this game, the standard board size is 8-squares-by-8-squares. With the initial board state shown in Figure 2.1, the black and white players take turn to make a move with the black player starting playing first. The black player makes the first move by playing a black disc on an empty square adjacent to his opponent's disc. As shown in Figure 2.1, the valid move of the black player at this state are C4, D3, E6, and F5. Opponents' discs that are sandwiched between the newly placed disc and another disc of the same color along a line in any direction (horizontal, vertical, or diagonal) are flipped to match with the color of the newly made move. If a player has no valid move, the player passes. The game ends when no players can make a valid move. The player with the most discs of his colour wins the game, while his opponent loses the game. When the number of black and white discs are equal, a draw is declared.

![Figure 2.1: The initial board state of Othello](image)

Note that as there are finitely many agents, actions and states in Othello, Othello is a finite game. Given a particular instance of an Othello game, since every agent is clear about the current state of the game, all the possible action and what they should do, this game is of perfect information. Finally, since the gain of a player in each flip equals to the loss of his opponent, the total payoff throughout the game is constant. Hence, Othello is a zero-sum game. These properties are critical for implementing the game tree and the search algorithms.

Different computer Othello’s have been developed to demonstrate the strength of AI throughout the years. In 1980, the Othello program Moor, written by Mike Reeve and David Levyl, won one game in a six-game match against world champion Hiroshi Inoue. It was the first time a computer Othello could win a game of Othello against a human world champion. 17 years later, in 1997, Logistello won all the games in a six-game match against world cham-
pion Takeshi Murakami. It proves that a computer Othello can play Othello better than any human players.\[6\]

To train the programme Logistello, features of discs at the end of the game were studied and mapped by logistic regression. Millions of training positions were thoroughly studied and labeled with a true minimax value or an approximation. A large sparse linear regression was used to assign values to pattern configurations at 13 game stages, with each being defined by the number of discs on the board.\[7\] It takes feature correlations into accounts when doing the value assignment, which was not achieved by previous computer Othello’s and was considered as the reason of its success.

Despite the success of the current computer Othello’s, one may still think if it is possible for a computer to learn the game all by itself, i.e. the ability of the computer Othello to determine the value of a board state from scratch by itself without any human knowledge of the game being involved. This is where machine learning can be applied to the computer Othello training.

There were attempts to train the computer Othello by neural network over the years, which can be divided into two categories in terms of data treatment. The first type is learn the game in the process of game play. In the study done by M. van der Ree in 2013\[8\], different strategies in reinforcement learning were implemented, including TD-learning and Q-learning. The computer Othello was trained to learn from three types of opponents, including from self-play, from a fixed opponent, and the fixed opponent’s move. A good performance in the test was achieved.

The second type of computer Othello training is by learning from the previous game. This type of training often involves Othello heuristics. Favourable moves, for example, move at the corner, were often given a higher score during data processing for later training. K. A. Cherry trained a computer Othello using neural network with genetic algorithm. During the data processing, he assigned higher score to the advantageous moves. \[9\] While this approach gave a better performance for training, human knowledge was involved in the training process.

The problem of how well a computer Othello can learn the game without any human logic arises. With the success of AlphaGo, it is worth investigating how well deep learning neural network may be a good way to train a computer Othello without any human knowledge of the game.
Chapter 3

Methodology

3.1 Training Data Collection and Processing

To facilitate the training process, 885 sets of game move records were collected throughout the project. 172812 different board configurations were obtained from the game move records and used in the neural network training. The value of each board configuration were assigned in two ways total sum and averaging.

3.1.1 Training Data Collection

100 sets of game move records were collected at the initial stage of the project. 50 sets of those records were obtained from the games played by human against the Android and iOS application The Othello with different levels of difficulty. Another 50 sets of the records were obtained from the games played by the application against itself with different levels of difficulty. Together with another 100 sets of game move records provided by Ng Argens and Yu Kuai, 200 sets of game move records were first being processed and used in the initial stage of the neural network training.

During the training process, another 685 sets of game move records were collected from the games played in the Othello WorldCup 2013 and World Othello Championship 2016 in hope to resolve the overfitting problem. These game move records, with relatively higher reliability, were processed and used in the later neural network trainings.

For each Othello game, the moves are recorded sequentially in the same file with the format of each move being specified as

\[ c_i x_i y_i \]

where \( c_i \), an integer with two possible values 0 and 1, represents the color of the disc in the \( i^{th} \) move, with 0 being white and 1 being black. \( x_i \) and \( y_i \), where \( 0 \leq x_i, y_i \leq 7 \) are two integers representing the x-coordinate and the y-coordinate of the disc position in the \( i^{th} \) move respectively.
3.1.2 Training Data Processing

A programme was written to translate each move in a game move record to the corresponding board state. Each board state is represented by a string of 64 characters (without border) or a string of 100 characters (with border). The representation schemes are shown as the table below:

<table>
<thead>
<tr>
<th>Value</th>
<th>Scheme 1</th>
<th>Scheme 2</th>
<th>Scheme 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>An empty cell</td>
<td>Empty cell</td>
<td>Border</td>
</tr>
<tr>
<td>1</td>
<td>Cell occupied by a white disc</td>
<td>Cell occupied by a white disc</td>
<td>Cell occupied by a white disc</td>
</tr>
<tr>
<td>2</td>
<td>Cell occupied by a black disc</td>
<td>Cell occupied by a black disc</td>
<td>Cell occupied by a black disc</td>
</tr>
<tr>
<td>3</td>
<td>N/A</td>
<td>Border</td>
<td>Empty cell</td>
</tr>
</tbody>
</table>

Table 3.1: Cell representation scheme

Scheme 1 is used to represent the board state without the any border consideration, while Scheme 2 and Scheme 3 are used to represent the board state with the consideration of border. Figure 3.1 shows a board state of one game. Using Scheme 1, its corresponding board state representation is “0000000000000000000000000022200000000000000000000000000000000”, while using Scheme 2 and Scheme 3, its corresponding board state presentations are

"3333333333300000000330000000033000000003300222000033000000000000000000000033333333333" and

"0000000000033333300333333300333333300332223300333213330033333330033333330033333330000000000000000000" respectively.

Figure 3.1: A board state and its corresponding representations.

After converting all the moves in the game move records to their corresponding board states, each board state will be assigned a value. If the white player wins the game, +1 score will be added to all the board states in that game record. If the white player loses the game, -1 score will be added to all the board states in that game record. If it is a draw, 0 will be added to all the board states in that game record. Due to the symmetric property of the board, while assigning a value to each board state in the game, a value is also assigned to all its corresponding rotation board state.

A programme is written to do the value assignment. It opens each file containing all the board states of the game and read the final result of the game from the last line. Then, for each board state in the file, the programme looks up the board state and all its rotation from the vector which storing all the board states. If the board state read from the file is found in the vector, the value of the board state will be updated according to the final result of that game, i.e. either +1, -1, or 0 will be added to the existing board value. If the board state is...
not found in the vector, it will be pushed back to the vector with the initial value being the result of the game. Below is the pseudo-code of the board value assignments:

```
Data: number of games
Result: a vector containing the board states in all the game move records and their corresponding value
vector(string, int) boards;
i = 0;
while i < number of games do
    read result of the game;
    if result = white wins then
        value = 1;
    else if result = white loses then
        value = -1;
    else
        value = 0;
end
while not the last line of the file do
    for each rotation of the board state do
        look up the board state from boards;
        if found then
            boards.value += value;
        else
            boards.push_back(state, value);
        end
    end
    ++i;
end
```

Algorithm 1: Board state value assignment

Two sets of output files are generated from the above algorithm for neural network training - one with all the board states and their corresponding total score, and another with all the board states and their corresponding weighted score, where the weighted score of each board state is calculated by weighted score = \( \frac{\text{total score}}{\text{number of games}} \).

### 3.2 Implementation Language

The main implementation language for this project should support both functional design and UI design. For functional design, it must support neural networking training. Python is chosen as the programming language for this reason.

Python, first released in 1991, is a general-purposed, interpreted, dynamic programming language. Python is among the programming languages which are used for data processing. More importantly, there are many libraries in Python that support deep learning, for instance, Caffe, TensorFlow, and Theano. In particular, Keras was chosen for implementation.

Keras, written in Python, is a high-level neural networks API. It can run on top of either TensorFlow or Theano, which are both popular deep learning libraries. In addition, the user-
friendliness, modularity and extensibility of Keras allow easy and fast prototyping [10]. The simple APIs enable us to observe the neural network design directly from the source code, which helps the implementation and modification of the neural network. In this project, Theano was chosen as the backend library. Theano is among the oldest and the most stable deep learning library. [11] It is also widely used in academic research. With Keras as wrapper, Theano performs better in terms of speed, classification accuracy and source lines of code for deep learning. [12]

3.3 Neural Network Design and Implementation

3.3.1 Overview of Neural Network

Neural network is a way of information processing. The basic structure of a neural network can be described by the input layer, hidden layer(s), and the output layer. The neural network accepts a number of input nodes with their corresponding labels in the input layer and gives out the training result from the output node(s) in the output layer. The nodes in the input layer and the output layer are connected to the nodes in one or more hidden layers, where the training takes place (see Figure 3.2).

![Figure 3.2: An example of a neural network with three layers.](image)

There are two input nodes in the input layer. The input nodes are connected to three nodes in the hidden layer with assigned weight. The hidden layers nodes are connected to the only output layer node. The activation function of this example is $y = \frac{1}{1+e^{-x}}$.

A bias, $b_i$, is assigned to every node $i$ in the hidden layer and the output layer as the threshold. A weight, $w_{ij}$, is also assigned to each edge connecting the two nodes $i$ and $j$ different layers. The value of each node, $v$, is calculated by the formula

$$v = f \times \sum b_i w_{ij}$$

where $f$ is an activation function. This value assignment step repeats until the output layers nodes are reached. This process is known as forward propagation. If the final output is not desirable, the set of weights and bias will be adjusted by the process back-propagation. In back-propagation, the calculation for each bias is similar to that in forward propagation, except that the calculation starts from the output layer instead of the input layer. The adjustment of the weights and biases are determined by the error function and the learning
rate of the neural network.

A deep learning neural network is neural network with more than one hidden layer. In particular, a convolutional neural network was implemented in this project.

![Input -> Conv -> ReLU -> Conv -> ReLU -> Pool -> ReLU -> Pool -> Fully Connected](image)

**Figure 3.3: A classical CNN architecture**

The input layer takes the input training data and its corresponding labels. The first convolutional layer captures the lower-level features of the image and generates an activation map according to the filters defined. Then an elementwise activation function is applied on the activation map, and passes the result to the next convolutional layer. Pooling layers are added in between to reduce the network complexity, including the number of parameters and computation time. Finally, a fully-connected layer is added at the end to combine the previous training result and give an output of the training.

A CNN is a neural network that consists of some convolutional layers, which helps reduce the number of parameters, and hence reduce the overfitting problem. [12] It also explicitly assumes taking 2D structure of an input image in the input layer, which is favourable for processing the input training data, i.e. board states, in this project. Figure 3.3 shows a classical CNN architecture. Similar to all the neural network, a CNN has the input layer as its first layer. Right after the input layer is a convolutional layer, which is essential to capture lower features of the image. Then the activation function and pooling layer(s) will be applied together with the convolutional layer has the hidden layers of the deep learning neural network. The CNN ends with a fully connected layer, which gives the final output of the training. Below is a detailed description of the layers in a convolutional neural network:

- **Input layer** It takes the input data and its corresponding labels for the neural network training.
- **Convolutional layer** It is the first layer after the input layer. A number of n-by-n filters are applied and adjusted to extract the most useful information for the training in this layer. After filtering on the input, a activation map (also known as a feature map) is obtained. On the next convolutional layer, another set of filters are applied on the activation map obtained from the previous convolutional layer. This results in an activation map that represents higher level features.
- **Pooling Layer** It is commonly added between successive convolutional layer to reduce the spatial size of the representation. This can reduce the complexity of the neural network.
- **Fully-connected layer** It is fully-connected with the previous layer. Weighted sum of the node(s) are computed by matrix multiplication in this layer.
- **Dropout layer** This layer drops some units and their connections from the neural network during the training to prevent overfitting.

### 3.3.2 Neural Network Design

Different neural network models were implemented and tested throughout the project. These models can be classified into 3 designs according to their network architecture. The following subsections describes their neural network architecture.
3.3.2.1 Neural Network Design 1: 5 Convolutional Layers and 3 Pooling Layers

This model design is the most complex among all the three designs. A larger model with more convolutional layers and pooling layers is first designed and trained in the hope of capturing more features during the training. 5 convolutional layers in total are added to the neural network, with a pooling layer added after every two successive convolutional layers and after the last convolutional layer. The neural network is flattened at the end and 2 fully-connected layers are added to get the ultimate result of the training. The source code of this model implementation can be found as follows:

```python
def baseline_model():
    model = Sequential()
    model.add(Convolution2D(32, 5, 5, border_mode='same', input_shape=(1, 10, 10), activation='relu'))
    model.add(Convolution2D(16, 5, 5, border_mode='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Convolution2D(16, 3, 3, border_mode='same', activation='relu'))
    model.add(Convolution2D(16, 3, 3, border_mode='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Convolution2D(16, 3, 3, border_mode='same', activation='relu'))
    model.add(Flatten())
    model.add(Dense(16, activation='relu'))
    model.add(Dense(output_dim=1, init='normal'))
    model.compile(loss='mean_squared_error', optimizer=adam, metrics=['accuracy'])
    return model
```

Listing 3.1: Neural network model 1, with 5 convolutional layers, 3 pooling layers and a fully-connected layer

3.3.2.2 Neural Network Design 2: 3 Convolutional Layers and a Dropout Layer

After training and testing the first model, it is discovered that overfitting problem arises due to insufficient training data. The complexity of the neural network is hence reduced in hope to solve the overfitting problem. Besides, a dropout layer was added to replace the pooling layers in the previous model. The dropout layer drops the units and their connection in the neural network according to the parameter specified. This helps prevent the units from co-adapting too much, which leads to overfitting. In this model, 3 convolutional layers added to the neural network, followed by a dropout layer with a dropout rate of 0.25. The neural network is flattened at the end and a fully-connected layer is added to get the ultimate result of the training. The source code of this model implementation can be found as follows:

```python
def baseline_model():
    model = Sequential()
    model.add(Convolution2D(32, 5, 5, border_mode='same', input_shape=(1, 10, 10), activation='relu'))
    model.add(Convolution2D(16, 3, 3, border_mode='same', activation='relu'))
    model.add(Convolution2D(16, 3, 3, border_mode='same', activation='relu'))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(16, activation='relu'))
    model.add(Dense(output_dim=1, init='normal'))
    model.compile(loss='mean_squared_error', optimizer=sgd, metrics=['accuracy'])
    return model
```

Listing 3.2: Neural network model 1, with 3 convolutional layers and a dropout layer
3.3.2.3 Design 1 - 3 Convolutional Layers and 3 Pooling Layers

As the training result given by Model 2 was unsatisfactory, 3 pooling layers are used to replace the dropout layer. In this model, 3 convolutional layers and 3 pooling layers added to the neural network alternately. The neural network is flattened at the end and a fully-connected layer is added to get the ultimate result of the training. The source code of this model implementation can be found as follows:

```python
def baseline_model():
    # create model
    model = Sequential()
    # It takes an array of input shape 10x10, the board size with border. 32 filters of size 5 x 5 are applied.
    model.add(Convolution2D(32, 5, 5, border_mode='same', input_shape=(1, 10, 10), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Convolution2D(16, 3, 3, border_mode='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Convolution2D(16, 3, 3, border_mode='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(16, activation='relu'))
    model.add(Dense(output_dim=1, init='normal'))

    # Compile model
    model.compile(loss='mean_squared_error', optimizer=sgd, metrics=['accuracy'])
    return model
```

Listing 3.3: Neural network model 1, with 3 convolutional layers and 3 pooling layers

3.4 Decision-making Algorithms

The search algorithms, minimax algorithm and alpha-beta pruning, are implemented as the game engines for the computer Othello.

3.4.1 Minimax Algorithm

The minimax algorithm is a popular algorithm in decision making for a game. For a game involving two players, consider the player making the next move as MAX and his opponent in the game as MIN. The MAX player is responsible for making the best move out of all the available next moves. To maximize his payoff, the MAX player has to explore and find the least favourable move of MIN. Similarly, for MIN player, he has to choose the move with the minimum score out of all MAX’s possible next moves given his state. This process recurs until it reaches the terminating level, where the leaf nodes of the game tree will be evaluated by a utility function. Figure 3.4 demonstrates how the minimax algorithm works.

![Figure 3.4: Demonstration of the minimax algorithm](image-url)

Figure 3.4: Demonstration of the minimax algorithm
The concrete minimax algorithm is given as follows:

```python
function minimax (node, level, isMax);
Input : Current node node, the search depth, depth and a boolean value indicating
        if the player is MAX, isMax
Output: Value of the best next move
if level == 0 then
    evaluate the leaf node;
else if isMax then
    v = -infty;
    for each child of node do
        v = max(v, minimax (child, level-1, False));
    end
    return v;
else
    v = infty;
    for each child of node do
        v = min(v, minimax (child, level-1, False));
    end
    return v;
end
```

**Algorithm 2: Minimax Algorithm**

A game engine (minimaxE.py) was written using the minimax algorithm for selecting the next best move in each round. Note that for MIN player selecting the move with the minimum score, it is equivalent to multiplying -1 to all the MAX’s moves and select the move with the maximum score. This variation of the minimax algorithm was implemented in minimaxE.

The depth of search was set to be 4. Since the minimax algorithm performs a complete depth-first exploration of the game tree, the time complexity of the minimax algorithm with b legal moves at each state is $O(b^d)$. To improve the efficiency of the minimax algorithm, alpha-beta pruning is also implemented.

### 3.4.2 Alpha-beta Pruning

Another widely-adopted search algorithm in decision-making for a game is minimax with alpha-beta pruning. The full minimax search explores some part of the tree that is not necessary to explore. Alpha-beta pruning is an algorithm to avoid searching the parts of a tree with nodes value not lying in the range $[\text{min}, \text{max}]$. This algorithm can increase the efficiency of searching the game tree and is usually used together with the minimax algorithm.
for game tree search. The algorithm is outlined as follows:

```python
function alphaBeta (node, alpha, beta, level, isMax);
Input : Current node node, alpha value alpha, beta value beta, the search depth
depth and a boolean value indicating if the player is MAX isMax
Output: The value of the best next move
if level == 0 then
  evaluate the leaf node;
else if isMax then
  for each child of node do
    alpha = max(alpha, alphaBeta (child, alpha, beta, level-1, False));
  end
  beta := alpha
  return alpha;
else
  v = infty;
  for each child of node do
    beta = min(beta, AlphaBeta (child, alpha, beta, level-1, False));
  end
  if beta != alpha then
    return beta;
  return beta;
end
```

Algorithm 3: The algorithm of minimax with alpha-beta pruning

A game engine (alphaBetaE.py) was written using the minimax algorithm with alpha-beta pruning for selecting the next best move in each round. The depth of search was set to be 4. Depending on the arrangement of the nodes in the tree, the worst case time complexity of this algorithm can be as bad as that of the minimax algorithm, i.e. $O(b^d)$, where $b$ is the number of legal moves at each state. In general, the time complexity of the this algorithm with $b$ legal moves at each state is roughly $O(b^3)$. The time complexity can reach to $O(b^2)$ if the tree is well-ordered. [15]

### 3.5 Testing

The computer Othello developed will be tested against two types of opponents, moderate computer opponents and strong computer opponents. The application The Othello was chosen for testing. This is an application available on both Google’s Play Store and Apple’s App Store. Monte Carlo’s tree search algorithm is implemented in the application. 30 levels of difficulty were defined by the number of playouts in this application, with Level 1 being the easiest and Level 30 being the most difficult. These levels were grouped to six levels of difficulties as shown in Table 3.2.

Due to the unexpected long testing time, the two types of opponents are only tested with 5 games each. When playing against a moderate computer opponent, the targeted winning rate is 50%, while the targeted winning rate for playing against a strong computer opponent is 35%.
There are two reasons for the decision on the winning rate.

Firstly, it is believed that when both sides play the game perfectly, the game will very likely end with a draw. Hence, when the winning rate of 50% is achieved, the computer Othello we developed can be said to be comparable with the existing computer Othello developed with the traditional method. Therefore, the winning rate of 50% when playing against a moderate opponent is chosen.

Secondly, due to the limited time for training and hardware availability, the evaluation function obtained from the neural network is not optimal. To obtain an optimal evaluation function, the neural network needs to be well-trained, which takes a lot of time. Without the availability of graphical processing unit for training, the neural networking training slow down by 8.5 times.

<table>
<thead>
<tr>
<th>Level of difficulties</th>
<th>Level 1 - Level 4</th>
<th>Level 5 - Level 9</th>
<th>Level 10 - Level 14</th>
<th>Level 15 - Level 22</th>
<th>Level 23 - Level 30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Very easy</td>
<td>Easy</td>
<td>Moderate</td>
<td>Strong</td>
<td>Extremely strong</td>
</tr>
</tbody>
</table>

Table 3.2: Level of difficulties of The Othello

Level 1 to Level 4 was considered as very easy. Level 5 to Level 9 was considered as easy. Level 10 to Level 14 was considered as moderate. Level 15 to Level 22 was considered as strong. Level 23 to Level 30 was considered as extremely strong.
Chapter 4

Division of Work

The work is divided among the team members as follows:

<table>
<thead>
<tr>
<th>Member</th>
<th>Responsibility</th>
<th>Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAN Lok Wang</td>
<td>Training data collection and processing</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Design and implementation of the model in building the evaluation function of the game tree</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>Design and implementation of the decision making algorithm</td>
<td>70%</td>
</tr>
<tr>
<td></td>
<td>Testing</td>
<td>30%</td>
</tr>
<tr>
<td>YIP Tsz Kwan</td>
<td>Training data collection and processing</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>Design and implementation of the model in building the evaluation function of the game tree</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Design and implementation of the decision making algorithm</td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td>Testing</td>
<td>70%</td>
</tr>
</tbody>
</table>
Chapter 5

Results and Discussion

The training results of the three models described in section 3.3.2 are presented in this chapter.

For the neural network training, the running environment is specified as below:

<table>
<thead>
<tr>
<th>Running Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine</td>
</tr>
<tr>
<td>MacBook Pro (Retina, 13-inch, Early 2015)</td>
</tr>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>2.7GHz Dual Core Intel i5</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>8GB 1867MHz DDR3</td>
</tr>
</tbody>
</table>

Table 5.1: Parameters of two trained models

5.1 Neural network model 1: 5 Convolutional Layers and 3 Pooling Layers

As described in section 3.3.2.1, this neural network has 5 convolutional layers, 3 pooling layers and 2 fully-connected layers. The layer structure in sequential order and the details are described as follows:

- Convolutional layer with 32 filters of size 5x5. Input data taken was of size 10x10 and the activation function rectified linear unit was applied.
- Convolutional layer with 16 filters of size 5x5. The activation function rectified linear unit was applied.
- Pooling layer with the pooling size of 2x2
- Convolutional layer with 16 filters of size 3x3. The activation function rectified linear unit was applied.
- Convolutional layer with 16 filters of size 3x3. The activation function rectified linear unit was applied.
- Pooling layer with the pooling size of 2x2
- Convolutional layer with 16 filters of size 3x3. The activation function rectified linear unit was applied.
• Pooling layer with the pooling size of 2x2
• Fully connected layer with output size 16. The activation function rectified linear unit was applied.
• Fully connected layer which gave an output of dimension one.

5.1.1 Parameters specifications

With the same network design, different parameters have been used for training. Two models selected for the discussion. The table below summarize the parameters used in the two training models:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Model 1-1</th>
<th>Model 1-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training data score</td>
<td>Weighted</td>
<td>Normalized</td>
</tr>
<tr>
<td>Epoch</td>
<td>2000</td>
<td>1000</td>
</tr>
<tr>
<td>Validation dataset</td>
<td>50%</td>
<td>50%</td>
</tr>
</tbody>
</table>

Table 5.2: Parameters of two trained models

5.1.2 Training Result and Analysis - Data Accuracy

The training data accuracy and the validation data accuracy were analyzed in the graphs below:

Figure 5.1: Data accuracy of neural network model 1-1
As shown in Figure 5.1, while the data loss showed an exponential decrease, the validation data loss dropped for the first 250 epoch, and showed a slight increase afterwards. Moreover, the difference between the validation data accuracy and the data accuracy increased as the training went on. This shows that overfitting occurred in this model training.

![Figure 5.1: Data accuracy of neural network model 1-2](image)

Similar to Figure 5.1, Figure 5.2 also shows that overfitting problem occurred in the neural network training. After the first 200 epochs, the validation data loss started to increase slightly with some occasional spikes. The training data loss kept decreasing throughout the training. As the validation data loss remained higher than the training data loss for the training, it indicated the overfitting problem.

Consider the first 1000 epochs between the two models, Model 1-2 showed a trend with less spikes, which may indicate the stability in training, and a better performance.

The causes of overfitting problem may be the insufficient training data and the complexity of the neural network. With 9 hidden layers, it require more training data to capture the features and give a more accurate training result. To resolve this problem, a simpler network model was designed.

5.1.3 Training Result and Analysis - Testing against a Computer Othello

The trained networks were used as the evaluation function of computer Othello we developed and tested against the application The Othello. The results were shown as below:
Figure 5.3 shows that the computer Othello with model 1-1 as evaluation function loss all games when playing against a moderate and strong opponent. On the other hand, Figure 5.4 shows that the computer Othello with model 1-2 as evaluation function could occasionally win against a moderate opponent and a strong opponent. This matches with the graphs shown in Figure 5.1 and Figure 5.2.

It is worth noting that during the game play, the computer Othello we developed tended to make moves around the corner (the dangerous zones) but not at the corner, which shows that it could not identify and capture the favourable moves during the training.

### 5.2 Neural network model 2: 3 Convolutional Layers and 1 Dropout Layer

As described in section 3.3.2.2, this neural network has 3 convolutional layers, a dropout layer and 2 fully-connected layers. The layer structure and the details are described sequentially as follows:
• Convolutional layer with 32 filters of size 5x5. Input data taken was of size 10x10 and the activation function rectified linear unit was applied.

• Convolutional layer with 16 filters of size 3x3. The activation function rectified linear unit was applied.

• Convolutional layer with 16 filters of size 3x3. The activation function rectified linear unit was applied.

• Dropout layer with a dropout rate of 0.25

• Fully connected layer with output size 16. The activation function rectified linear unit was applied.

• Fully connected layer which gave an output of dimension one.

5.2.1 Parameters specifications

Similar to Model 1, with the same network design, different parameters have been chosen for training. Two models were selected for the discussion. The table below summarizes the parameters used in the two training models:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Model 2-1</th>
<th>Model 2-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training data score</td>
<td>Normalized</td>
<td>Normalized</td>
</tr>
<tr>
<td>Epoch</td>
<td>1200</td>
<td>1200</td>
</tr>
<tr>
<td>Validation dataset</td>
<td>50%</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 5.3: Parameters of two trained models

While only one neural network was trained for Model 2-1, two neural networks were trained for Model 2-2. As it was believed that a single evaluation function could not be used to evaluate all the board state accurately at any instance, it is suggested that two neural networks could be trained for two stages of the game respectively. A more accurate evaluation function might be generated in this way. Hence, the training data were split into two half - the first 30 moves of the game and the last 30 moves of the game. Two neural networks were trained using the two sets of data, and a variant of the minimaxE, minimax2E was developed to incorporate the changes.

5.2.2 Training Result and Analysis - Data Accuracy

The training data accuracy and the validation data accuracy were analyzed in the graphs below:
As shown in Figure 5.5, both the data loss and the validation data loss dropped throughout the training. The data loss decreased more rapidly compared with the validation data loss. Less spikes were observed in the graph and the fluctuation range of the validation data loss is smaller compared with Model 1. Even though there was a notable difference between the validation data loss and the data loss, such difference reduced compared with Model 1. This shows that while the overfitting problem still existed in this model, this model was better than previously trained model.
Figure 5.6: Data accuracy of the first 30 moves in neural network model 2-2

Figure 5.7: Data accuracy of the last 30 moves in neural network model 2-2

Figure 5.6 shows the training result of the neural network trained for the first 30 moves, while Figure 5.7 shows the result of the neural network for the last 30 moves. Even though the results show a similar trend as that Model 2-1, the result of the last-30-moves model appeared to be less desirable. Its data loss remained at less than 0.25, which was satisfactory, but its validation data loss was above 1.7, which was more than double of that in Model 2-1. One
of the explanation to this could be less moves were made in the later part of the game as the game may terminated earlier when no available could be made. This might reduce the size of dataset for the training.

As observed from the graphs, while the overfitting problem still existed, this problem was alleviated with a smaller network model. One way to further improve the performance is by having more training data.

### 5.2.3 Training Result and Analysis - Testing against a Computer Othello

The trained networks were used as the evaluation function of computer Othello we developed and tested against the application The Othello. The results were shown as below:

![Figure 5.8: Test Result of neural network model 2-1](image)

![Figure 5.9: Test result of neural network model 2-2](image)

Figure 5.9 shows that the computer Othello with model 2-1 as evaluation function had a winning rate of 20% when playing against a moderate and strong opponent respectively. On the other hand, Figure 5.9 shows that the computer Othello with model 2-2 as evaluation
function had a winning rate of 40% when playing against a moderate opponent and a winning rate of 17% when playing against a strong opponent. This showed the it performed better than the previous computer Othello.

During the game play, the computer Othello we developed with Model 2 could occupy the corner once in a while, which shows that it became more sensitive to the preferable moves. In addition, it showed an effort to avoid the dangerous zones, i.e. the cells around the 4 corners. This all shows that there is an improvement of game play from Model 1.

5.3 Neural network model 2: 3 Convolutional Layers and 3 Pooling Layer

As described in ??, this neural network has 3 convolutional layers, 3 pooling layers and 2 fully-connected layers. The layer structure and the details are described as follows:

- Convolutional layer with 32 filters of size 5x5. Input data taken was of size 10x10 and the activation function rectified linear unit was applied .
- Pooling layer with the pooling size of 2x2
- Convolutional layer with 16 filters of size 3x3. The activation function rectified linear unit was applied.
- Pooling layer with the pooling size of 2x2
- Convolutional layer with 16 filters of size 3x3. The activation function rectified linear unit was applied.
- Pooling layer with the pooling size of 2x2
- Fully connected layer with output size 16. The activation function rectified linear unit was applied.
- Fully connected layer which gave an output of dimension one.

5.3.1 Parameters specifications

With the same network design, different parameters have been used for training. Two models selected for the discussion. The table below summarize the parameters used in the two training models:

5.3.2 Training Result and Analysis - Data Accuracy

The training data accuracy and the validation data accuracy were analyzed in the graphs below:
As shown in Figure 5.10, the data loss showed a sharp decrease at the beginning and remained relatively stable afterward. The key of this graph is that the validation loss was lower than the data loss throughout the training, which shows a drastic improvement from the previously models. This may be due to the right choice of border value and more dataset being used for training.
Similar to Figure 5.10, Figure 5.11 also shows that the validation data loss and the data loss being quite close throughout the neural network training. However, a lot of spikes were found and overall, both the validation data loss and the data loss fluctuated a lot, which indicated that the network model was not good as the previous Model 3.1.

While the overfitting problem seems to be solved, to improve the performance of the model, larger model can be built and trained with more dataset.

### 5.3.3 Training Result and Analysis - Testing against a Computer Othello

The trained networks were used as the evaluation function of computer Othello we developed and tested against the application The Othello. The results were shown as below:
Figure 5.12: Test Result of neural network model 3-1

Figure 5.13: Test result of neural network model 3-2

Figure 5.12 shows that the computer Othello with model 3-1 as evaluation function had a winning rate of 60% when playing against a moderate opponent, which indicates the achievement of one of the project objectives. It had a winning rate of 20% when playing against a strong opponent. On the other hand, Figure 5.13 shows that the computer Othello with model 3-2 as evaluation function had a winning rate of 40% when playing against a moderate opponent and no winning chance playing against a strong opponent. This matches with the data accuracy results as analyzed above.

During the game play, the computer Othello we developed with Model 3 could occupy the corner once in a while, which shows that it became more sensitive to the preferable moves. While it still made moves at the dangerous zone, it could capture the opponent’s mistake and turned into his advantage, hence winning the game eventually. This shows the improvement of the computer Othello in the game play.
Chapter 6

Conclusion and Future Research

6.1 Conclusions

This project aims to develop a computer Othello of size 8x8, where its evaluation function being constructed without any human logic involved. The machine learning technique to be used to achieve this purpose is deep learning neural network. The targeted winning rate is 50% when playing against a moderate opponent, and 35% when playing against a strong opponent. This objective was partially achieved by Model 3-1, where 3 convolutional layers, 3 pooling layers and a fully-connected layer were being implemented as the hidden layers. While the overall result was not as satisfactory as expected, the experience and results obtained from this project might be useful for future computer Othello development using deep learning neural network.

6.2 Future Research Directions

For future development, more training data should be collected for the training purpose. It is estimated that to develop a strong computer Othello, a million sets data are required. Moreover, it is important to capture the border during the training, hence in the cell value assignment, it is advised to use a larger value to indicate the border region.

To allow more feature to be captured, a larger neural network should be designed and implemented for training. Though, developer must ensure the data sufficiency.

Different optimizers and loss function can also be used for the training in the hope to achieve better result.
List of References


Appendix A

The game of Othello

Othello is a strategy game played by two players in black and white discs. The standard board size is 8x8. The game starts from the board configuration with the four centered squares occupied as follows:
The black player makes a move first, and the white player follows. A legal move requires players to place a disc on an empty square adjacent to his opponents disc. Opponents discs sandwiched between the newly placed disc and another disc of the same color along a line in any direction (horizontal, vertical, or diagonal) are flipped to match with the color of the newly made move. An example of a legal move is shown below:

Figure A.1: Initial board position of Othello

Figure A.2: A legal move in the game Othello.
In Figure A.2 Black player makes a move at the empty space C6, flipping the disc at positions B5 across the horizontal line, B6 across the diagonal line, and C4 and C5 across the vertical line.

The game continues until no legal moves are available. When the game terminates, the player who has more discs on the board is the winner. When both players have the same amount of discs on the board, they tie the game.