

Playing Othello By Deep Learning Neural Network

Ng Argens

The University of Hong Kong

Author Note

Argens Ng, BEng (Comp. Sc.), The University of Hong Kong (UID.: 3035072143)

This paper was prepared in partial fulfillment of the Final Year Project (COMP 4801) required by the curriculum of Bachelor of Engineering (Computer Science)

Table of Contents

Abstract.....	8
Acknowledgement.....	9
Playing Othello By Deep Learning Neural Network.....	10
Previous Work – AlphaGo.....	11
Overall Structure.....	11
Artificial Neural Network.....	12
Value Network.....	12
Policy Network.....	12
Rollout Policy.....	12
Training Algorithm.....	13
Supervised Learning.....	13
Reinforcement Learning.....	13
Tree Search.....	13
Monte Carlo Tree Search.....	13
Literature Review – Strategy Guide for Reversi & Reversed Reversi.....	14
Positional Strategy.....	14
Liberty / Freedom.....	15
Stable Discs.....	16
Frontier.....	16

Literature Review – Writing an Othello Program⁶	17
Othello Specific	17
Disc-Square Tables.	17
Mobility-Based Evaluation.	17
Pattern-Based Evaluation.....	17
General Game Tree	18
Alpha-Beta Pruning.	18
Move Ordering.....	18
Architecture	19
Raw Data Extraction.....	19
decoder.py	19
RawToStates.py.....	20
Neural Network.....	20
Feature Extraction.....	20
Training.....	20
Testing.....	20
Prediction.....	21
Game Rule Enforcement.....	21
othello.py	21
Core Program	22

Neural Network.....	23
Library.....	23
Feature Extraction.....	23
Raw Board Position.....	23
Border.....	24
Rotation.....	24
Corner.....	25
Liberty.....	25
Training and Testing.....	26
Early Stopping.....	26
Hyper-parameter Tuning.....	26
Testing.....	27
Graphical User Interface.....	28
Core Functionality.....	28
Mouse Position.....	28
Disc and Grid Representation.....	28
Immediate Feedback.....	28
Hover Position.....	28
Move Validity.....	29
Available Moves.....	30

Game Tree Expansion.....	31
Core Concept	31
Alpha-Beta Pruning.	31
Move Ordering.....	31
Minimax.....	31
Repeating Player.	31
Implementation	32
Tree.	32
Node.....	32
Multi-threading	33
Bottom-Up Approach.....	33
Leaf.	33
Inner Nodes.....	33
Root.....	33
Operations Beside Expansion	34
Next Node.....	34
Lock.	35
Pruning.....	35
Experiments and Results.....	36
Border Only	36

Border and Corner.....	38
Border, Corner and Freemove.....	39
Justification and Explanation	41
Why is Othello a suitable game?	41
Why is policy network not used?.....	41
Why is Python being used?.....	41
Why is reinforcement learning not being used?	42
Why are more complicated features not being used?	42
Conclusion and Future Work.....	44
Architecture.....	44
Neural Network.....	44
Feature Extraction.....	45
Reinforcement Learning.....	45
Policy Network.....	45
User Experience	45
Tree Traversal	45
References.....	46
Neural Network Structure	47
Table 1	48
Table 2	49

Table 3 **50**

Table 4 **51**

Abstract

Computer Go has always been considered a major hurdle for Artificial Intelligence development¹ due to its enormous number of possible moves and hence large degree of freedom. This hurdle was overcome in October 2015 when AlphaGo became the first Computer Go program to beat a professional human Go player without handicap on a full-sized 19x19 board. Our project aims to replicate the success of AlphaGo in the game of Othello (a.k.a. Reversi). By using similar algorithms and components but with smaller board size and degree of freedom, we hope to investigate the effect of neural network in computer's performance in playing games.

In the coming sections, we will describe the design of AlphaGo and how our program adopted various strategies of it. We will also discuss the adaptations made to fit another game and different challenges faced and overcome throughout the project. Lastly, we will showcase the experiments done and how the data collected can expedite the development in future works.

Keywords: Othello, Computer Othello, Deep Learning, Neural Network

Acknowledgement

I would like to express my greatest gratitude towards Dr. K. P. Chan, my mentor, who has supported and guided me throughout the project. This project would not have such achievement without his support and guidance. I would also like to thank the department of Computer Science for their generous support in equipment and organization, Ray Kurzweil who has inspired me to pursue in the field of Artificial Intelligence, and the selfless contribution of the Google DeepMind team in the development of AlphaGo.

Playing Othello By Deep Learning Neural Network

This final report serves the purpose of concluding my Final Year Project named “Playing Othello By Deep Learning Neural Network”. The project is conducted under the supervision of the Computer Science Department of the University of Hong Kong and in particular, my mentor, Dr. K. P. Chan.

The main goal of my project is to investigate and mimic a highly successful Computer Go program called AlphaGo, which is not only the first computer Go program to beat a professional human Go player without handicaps, but also the number one “player” back on July 18th, 2016, according to Go Ratings, a website which ranks Go players according to their Elo. Its most recent accomplishment was playing on various Go websites against the best players in the world at the end of 2016, resulting in a final score of 60 wins out of 60 games.

For our game of choice, Othello is favored over Go as its simplicity would allow us to create an AlphaGo-like program without having to use a supercomputer. Our game of choice would also be further discussed in greater detail in the later part of *Justification and Explanation*.

In the end, while the mean squared error of our network was rather high, at around 0.71, our game still managed to achieve over 70%-win rate against testers by only searching 4 to 5 moves ahead. This indicates the high effectiveness of neural network, in particular, convolutional neural network, in games like Othello. This will also be discussed in *Justification and Explanation*.

But first, let us study the architecture of AlphaGo.

Previous Work – AlphaGo

Our project is inspired by AlphaGo and hence it will be discussed in details in the coming section.

AlphaGo is the best Computer Go program in the world and the first to defeat a human professional player in a 5-game series. It was ranked number one in July 2016 by Go Ratings, a website dedicated to ranking Go players, humans and computers alike².

Rank	Name	♂ ♀	Flag	Elo
1	Google DeepMind AlphaGo			3612
2	Ke Jie	♂		3608
3	Park Junghwan	♂		3589
4	Lee Sedol	♂		3557
5	Iyama Yuta	♂		3532
6	Mi Yuting	♂		3529
7	Kim Jiseok	♂		3515
8	Lian Xiao	♂		3515
9	Shi Yue	♂		3509
10	Chen Yaoye	♂		3497

Figure 1. AlphaGo in July 2016 reached ranked one in terms of Elo on Go Ratings in July, 2016

Overall Structure

The structure of AlphaGo was understood through the official paper published – Mastering the game of go with deep neural networks and tree search³.

The main structure of AlphaGo is the Monte Carlo Tree Search (MCTS). It is accompanied by two deep learning neural networks, namely the policy network and the value network. The former one is developed for reducing branching factor, while the latter for reducing depth of search when necessary. While the policy network is accurate, a fast rollout policy is developed for side-by-side comparison. To train the two networks, supervised learning and reinforcement learning are used as standard machine learning approaches.

Artificial Neural Network

Artificial Neural Network (or simply, ANN or Neural Network) is believed by many to be the main reason of the success of AlphaGo. Neural Network is a computational model, based on a large collection of Artificial Neurons (or simply, Neurons) to remotely simulate the action in biological brains and neural networks. It is considered superior than traditional artificial intelligence approach for two reasons. Firstly, much less coding is needed when using ANN; And secondly, subtle rules or even rules unknown to human can be captured by ANN. This makes it exceptionally good at image and voice recognition, and as AlphaGo proved to us, in artificial intelligence and gaming as well.

Value Network. value network is analogous to heuristic functions used in traditional game tree search. It provides a value for the current game state if the tree has to be truncated due to time constraint. We can also imagine it as a human professional who can tell whether the dark or the white side is at an advantage at first glance. At first, full game histories were provided to train the value network in a similar manner to policy network. This led to overfitting and the machine actually “memorized” the training set. The network was then trained with distinct states from different games played by itself and earlier versions. The problem was then solved.

Policy Network. policy network was used to regulate the search width in AlphaGo. As Go is a game with such a high degree of freedom, this becomes especially crucial for the program to have a reasonable response time. For AlphaGo, 13-layer policy network was trained from 30 million positions from the KGS Go Server. The network is provided game states and the next move by human expert and would gradually learn to predict this move. The result was an accuracy of up to 57%, compared with a maximum of 44.4% for other research groups at that time. However, it takes 3ms for computation, which hence limits the traversed depth of the tree search.

Rollout Policy. Using a reduced set of feature, rollout policy can rapidly analyze known patterns on the board for the most probable moves. This is also trained with 8 million positions from human games on the Tygem server. The rollout policy achieved a 24.2% accuracy using only 2 μ s (0.067 % of policy

network). In the end, both rollout policy and policy network are used in a parallel manner, and both results are taken into account.

Training Algorithm

Supervised Learning. Supervised learning is used whenever a program is asked to provide answers to questions. It is supervised in the sense that both answers and questions are provided in the training examples. The program can then predict answers from similar questions later on. It is thus commonly used and was applied to all three of the policy network, rollout policy and value network.

Reinforcement Learning. After AlphaGo has achieved a basic benchmark in performance, its policy network underwent reinforcement learning and played games again and again with former versions of itself, which can be the first or, for example, the 524th version. The program can then identify its mistakes and update its network accordingly.

Tree Search

Tree search serves the foundation and backbone of AlphaGo, in a sense that it links every component and actually makes it able to play games.

Monte Carlo Tree Search. Monte Carlo Tree Search (MCTS) is a general game tree search without branching at all. Instead, each probe is conducted all the way to the end by moving randomly at each state⁴. The rewards and visit count are then back propagated to better enhance the next probe⁴. This continues until time runs out, which makes it ideal for playing under time constraint. However, as we can easily tell, moving randomly does not match a rational player (or opponent) and thus policy network is employed to better reflect reality. In the case AlphaGo, the matrix of play percentage returned by policy network can be directly employed by MCTS to facilitate its searching.

Literature Review – Strategy Guide for Reversi & Reversed

Reversi

Accessed from: <http://www.samssoft.org.uk/reversi/strategy.htm> ^[5]

To better adapt AlphaGo to the game of Othello (a.k.a. Reversi), I have looked into a wide array of literature guide. This strategy guide in particular provided valuable insight as to how the strategies of Othello differ from Go.

Positional Strategy

Similar to Go, corner and border discs have different meanings from those in the center. For one, corner discs can never be flipped while border discs can only be flipped from one direction. This can be observed from values assigned to various squares in the figure below.

	a	b	c	d	e	f	g	h	
1	99	-8	8	6					1
2		-24	-4	-3					2
3			7	4					3
4				0					4
5									5
6									6
7									7
8									8
	a	b	c	d	e	f	g	h	

Figure 2. An example of positional evaluation

Not only are corner squares extremely valuable, this scoring algorithm also reasonably punishes squares which grants the opponent access to the corner square, such as B2 (X-square) or B1 and A2 (the C

squares). It also shows the symmetry of every Othello game, where the first move by black is effectively immaterial to how the game progresses.

Later on, this scoring algorithm also contributed to my first prototype and first performance benchmark. It later evolved to contribute to the feature extraction of my program, discussed in (*Architecture – Neural Network – Feature Extraction*).

Liberty / Freedom

Freedom (or liberty as named in my program) is a concept that applies to Othello but not Go (although the same word is used). This is because in Go, players can virtually put their discs anywhere they want.

Whereas in Othello, moves available to each player depends highly on the opponents' discs and moves.

This makes limiting opponents moves a high priority target in the early to mid-game.

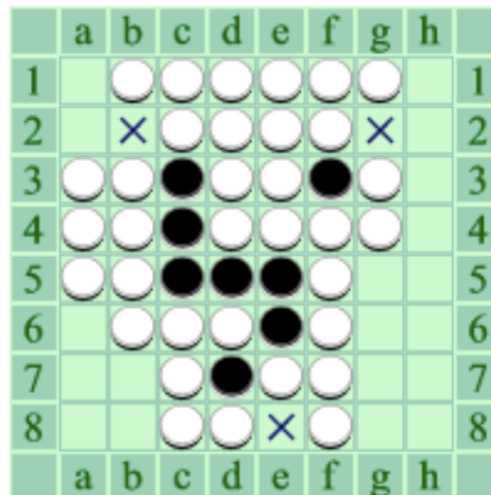


Figure 3. An example of liberty. Black has only 3 moves in this case

For example, in the above figure, white has only 3 moves available (highlighted in X), with 2 of which being highly averted X-squares (squares diagonal to corners). Black has then successfully limited the choices white can make, and if it can continue to convert as little white discs as possible in the following rounds, should be able to force white into making a terrible move.

Stable Discs

Stable discs refer to discs that cannot be converted in subsequent moves. Intuitively, corner discs and discs that are adjacent to them are stable discs. In the following figure (left), black has 23 stable discs while white only has one. Hence black only needs 10 more stable discs to secure a win (33/64 possible discs) and white's apparently higher disc count at this moment would not be a factor of concern.

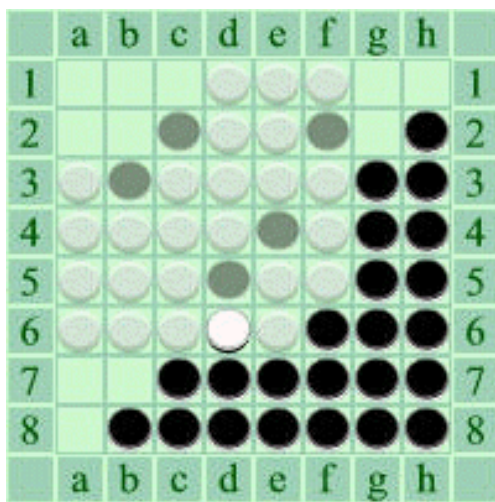


Figure 4. An example of stable discs

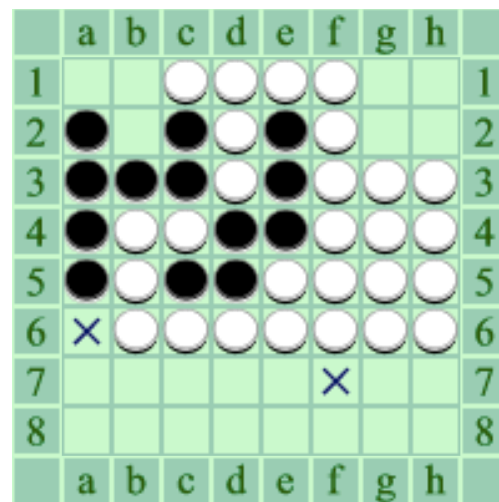


Figure 5. An example of the benefit of having a small number of frontier

Frontier

From stable discs and mobility comes the concept of frontier. As each move must be played adjacent to an opponent's disc into an empty square, the discs which have empty neighboring square forms the frontier while those that don't have such neighboring squares are the interior discs. A simple method of keeping opponent's mobility low is thus to maintain a low number of frontier discs. For example, in the above figure (right), A6 would be a better move in general. While it does flip 3 white discs, none of them is a frontier disc and it is thus a quiet move. Whereas in the case of F7, although only one white disc is converted, 2 frontier discs are created in the progress and thus granted the opponent more liberty in subsequent moves

Literature Review – Writing an Othello Program⁶

Accessed from: <http://www.radagast.se/othello/>

When writing an Othello program, there is no better way to start than learning from someone who already did it. In 1998, Zebra was declared the third best Othello algorithm in the world and has since then continued to develop until 2003. While the approaches adopted by G. Andersson in his Zebra program and me in this project are way different, many principles have proven itself to be universal, some in the world of computer gaming in general.

Othello Specific

Disc-Square Tables. This is in principle the same as the positional strategy covered in the previous literature. Basically, corners are good, squares next to corners are bad and each side has a total score calculated from whether or not one has his/her disc at that particular square. This strategy has been disregarded by Andersson, who commented programs evaluated this way are invariably weak.

Mobility-Based Evaluation. This takes the principle of frontier discs and liberty into consideration and is in generally stronger than using disc-square tables. By levitating into a more global approach, this style of evaluation can be a strong candidate in contesting corners by limiting opponent's move and increasing one's liberty.

Pattern-Based Evaluation. This is only vaguely described by Andersson but in essence it means evaluating row-by-row, column-by-column and diagonal-by-diagonal while taking the global evaluation into account. It also appears to involve breaking down a board into patterns such as crosses and sandwiched straights. However as too little detail was provided, it barely contributed to my program.

General Game Tree

Being a traditional artificial intelligence gaming program, Zebra had a lot of focus in efficient game tree traversal, with its main point highlighted below.

Alpha-Beta Pruning. While traversing a tree, as soon as we know that a node cannot be larger than or smaller than a certain value v , we can use this value to compare with its siblings' value. If the node in question is in a maximizing layer and is upper-bounded by a value smaller than its siblings' value, the node need not be expanded as it should not be chosen by the player in question. This also works other way round and pruning the game tree in this manner saved lots of time in the end.

Move Ordering. To further facilitate alpha-beta pruning, nodes with the best prospect are placed in front of those that seemed to be less favorable. This way, if the latter nodes are upper-bounded by a value smaller than the value of the first node, or the other way round, we do not need to expand the latter nodes at all. This is done by evaluating to a smaller depth first, then evaluate to the full depth according to the order found in the first evaluation.

Architecture

In this section, the architecture of the program is being presented. However detailed justifications and explanations would be left out to be answered in the next few sections.

Raw Data Extraction

To collect training data for our program, we first went to **Othello World Cup 2013** (<http://www.othello.com/live/>) and extracted match record in common human readable format. As the training data size has later proven to be inadequate, the large amount of data in WZebra is extracted instead.

decoder.py To extract data from the database in WZebra, “.wtb” format files have to be decoded. While we failed at finding a ready-made program (besides WZebra) that can understand such format, a website that describes such procedure in detail was found (**Computer Reversi, Part 1.5: The Thor Database:** <http://ledpup.blogspot.hk/2012/03/computer-reversi-part-15-thor-database.html>). We then proceed to write a snippet that converts it into human readable format and append it to the 150 games previously collected.

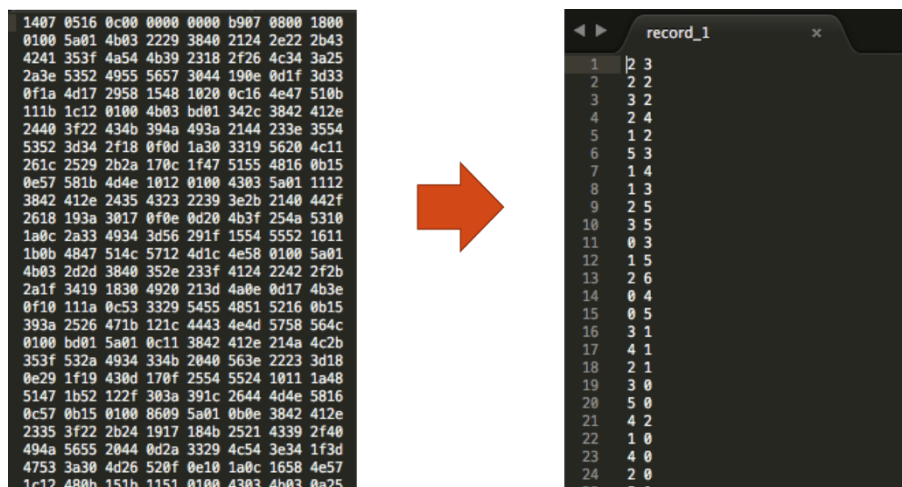


Figure 5. decoder.py in action

RawToStates.py. To train our network, we only need a particular state of a game instead of the match history. This code reads in, understands and produces a state of game randomly chosen for each line of match history. It also appends at the end the move chosen by the professional and the final winner corresponding to the current player. To further facilitate the network training process, the random seed can also be set before the whole process, which later proves to be extremely useful in network validation.

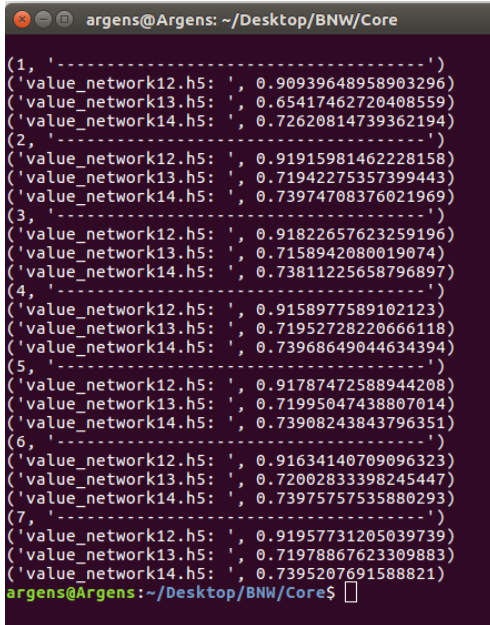
Neural Network

The four major steps in producing neural networks are modularized to facilitate future expansion and version control.

Feature Extraction. For feature extraction, the State class in *othello.py* plays a large part as it rotates the board according to the first move made by black. This effectively quadruples the sample size for training our network. The State class also returns a list of viable moves which helps our program in taking liberty into account. This would be described in further detail in the coming section **Neural Network – Feature Extraction.**

Training. This is solely conducted in *Trainer.py*, where a training set can be read in and cast into suitable formats (eg. Numpy Arrays) after extracting suitable features. We can then set some hyper-parameters in a grouped up manner, set the network architecture accordingly and save the models automatically without fear of overriding existing networks.

Testing. This is solely conducted in *Tester.py*, where any network can be tested on all the 7 (8 in the later stages) available data sets. They can also specify the required feature extraction independently so that all the testing can be conducted in a batch manner. The mean-squared-error (MSE) of each network on each training set would then be printed out in a neat and tidy manner.



```

argens@Argens: ~/Desktop/BNW/Core
(1, '-----')
('value_network12.h5: ', 0.90939648958903296)
('value_network13.h5: ', 0.65417462720408559)
('value_network14.h5: ', 0.72620814739362194)
(2, '-----')
('value_network12.h5: ', 0.91915981462228158)
('value_network13.h5: ', 0.71942275357399443)
('value_network14.h5: ', 0.73974708376021969)
(3, '-----')
('value_network12.h5: ', 0.91822657623259196)
('value_network13.h5: ', 0.7158942080019074)
('value_network14.h5: ', 0.73811225658796897)
(4, '-----')
('value_network12.h5: ', 0.9158977589102123)
('value_network13.h5: ', 0.71952728220666118)
('value_network14.h5: ', 0.73968649044634394)
(5, '-----')
('value_network12.h5: ', 0.91787472588944208)
('value_network13.h5: ', 0.71995047438807014)
('value_network14.h5: ', 0.73908243843796351)
(6, '-----')
('value_network12.h5: ', 0.91634140709096323)
('value_network13.h5: ', 0.72002833398245447)
('value_network14.h5: ', 0.73975757535880293)
(7, '-----')
('value_network12.h5: ', 0.91957731205039739)
('value_network13.h5: ', 0.71978867623309883)
('value_network14.h5: ', 0.7395207691588821)
argens@Argens:~/Desktop/BNW/Core$

```

Figure 6. tester.py in action

Prediction. To conduct the final prediction while traversing game tree, *evaluator.py* provides the class **Evaluator** which can be initialized to different feature extraction required by different network. It would then conduct packaging prior to the native *predict* function, which allows us to predict the value of single state efficiently.

Game Rule Enforcement

Like most board games, Othello has its own unique set of rules. For example, a player's round can be skipped completely when no move is possible. Also, whether or not a player has possible move depends on the opponent's discs as much as it depends on his/hers. This makes game rule enforcement crucial from early stages of the project.

othello.py The State class in *othello.py* automatically attempts to find possible moves after each move and changes the current player if deemed necessary. It also preemptively ends the game when neither player has a possible move and calculates the number of white and black discs at each round. Its

collection of possible moves at each round became extremely valuable as we trained our network to value liberty, while the player changing function proved to be crucial when decoding Othello match history from the web. The last function of calculating black and white discs were particularly useful when playing to a win in the last few moves.

Core Program

The core program can be divided into 3 main parts, namely artificial intelligence component, graphical user interface and flow management. The flow management basically acts as judge or the neutral board in between. Whenever it is time for the computer to make a move, it would urge the AI component to make a move, and update after necessary validation. After that, it would wait indefinitely before the user makes a move, which either prompts a reaction from the computer or leads to another endless wait. This continues until the flow management determines it is time to stop. The other 2 components would be discussed in detail in **Graphical User Interface** and **Game Tree Expansion** respectively.

Neural Network

Neural network, being a core component in this Final Year Project is treated separately since the early stages and was not merged to observe its influence until later on. It is also hence packed with most experiments and observations.

Library

From training to prediction, I used the open source library *Keras* which is both easy to use and powerful. It can utilize Tensorflow to train networks in a parallel manner using graphics card which sped up the training progress exponentially. It also allows the users to configure networks in a stack manner, with minimal input of data at each level and the rest taken care by *Keras* library.

Feature Extraction

Raw Board Position. At the earliest stage, only raw board data was fed to neural networks with 4 to 6 fully connected layers. As black disc, white disc and empty space do not form an ordinal sequence by themselves, we separated the board into 3 layers before passing on to our networks.

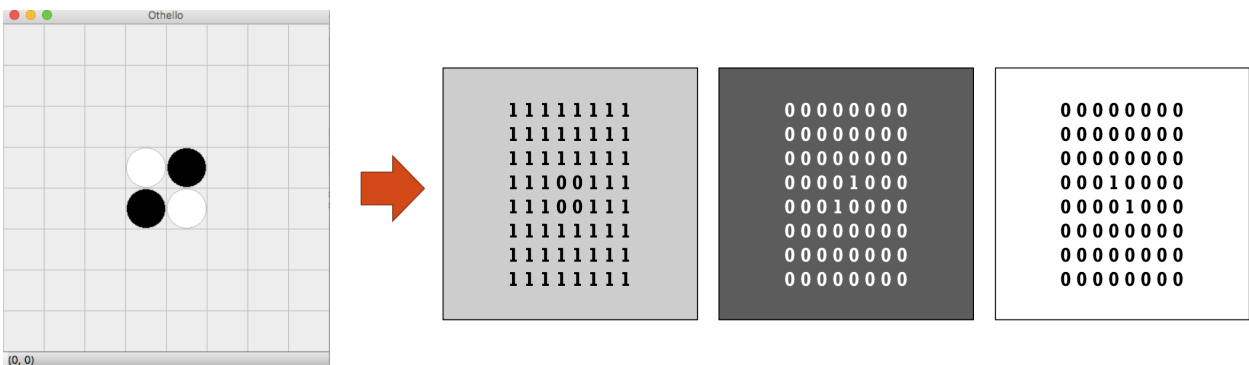


Figure 7. Feature extraction of raw board position

The result was either overfitting or undesirably high mean-squared error (MSE) of ~ 0.82 . Using convolutional neural network barely helped and instead we turned our focus to feature extraction.

Border. The first feature that got our attention is the border. At first, using older version of *Keras*, I was able to enlarge the size of the grid at each level, effectively preserving the uniqueness of border from convolution. As this feature was blocked from me later on, I instead pass on an additional layer (so in total, 4 layers) with 1s at the border and 0s in the center. The resulting MSE was on average around 0.79.

1	1	1	1	1	1	1	1
1							1
1							1
1							1
1							1
1							1
1							1
1	1	1	1	1	1	1	1

Figure 8. Padding layer of border

Rotation. I then utilize the symmetry of Othello and added rotation as a part of feature extraction. As each **state** object is created from match history, it would remember the first move played by dark. Later on, as we pass on the state to train our network, it would always rotate itself before doing so. By applying rotation, the average MSE dropped to 0.75.

Corner. The next feature that got my attention is corner. Intuitively, corner is very much different from border and should not be treated in the same manner. Along the same line, the square diagonal to the corners (X-squares) should not be treated equally as other squares in the center, and the squares touching the corners (C-squares) should not be seen as equal as other squares in the border. As a result, the layer passing along raw board becomes modified as follow. After this modification, the MSE drops to as low as 0.73 without significant signs of overfitting. Some networks still only have a MSE of 0.75 and these became indicators of the suitable number of nodes to be chosen in the dense layers and dropout layers.

	a	b	c	d	e	f	g	h	
1		C	A	B	B	A	C		1
2	C	X					X	C	2
3	A							A	3
4	B			○	●			B	4
5	B			●	○			B	5
6	A							A	6
7	C	X					X	C	7
8		C	A	B	B	A	C		8
	a	b	c	d	e	f	g	h	

Figure 9. Official names for special squares in Othello

4	-1	1	1	1	1	-1	4
-1	-1					-1	-1
1							1
1							1
1							1
1							1
-1	-1					-1	-1
4	-1	1	1	1	1	-1	4

Figure 10. Padding layer of border and corner

Liberty. The final feature that I added is liberty. As discussed in the 2 literature reviews, liberty based evaluation is widely regarded as a superior approach than positional approach alone. As such, this feature are important for our network if we were to take global integrity into account and rate it properly alongside greedy local evaluation. Since we were using convolutional neural network, we pass in a matrix of available moves. The complete input matrix is then being illustrated below.

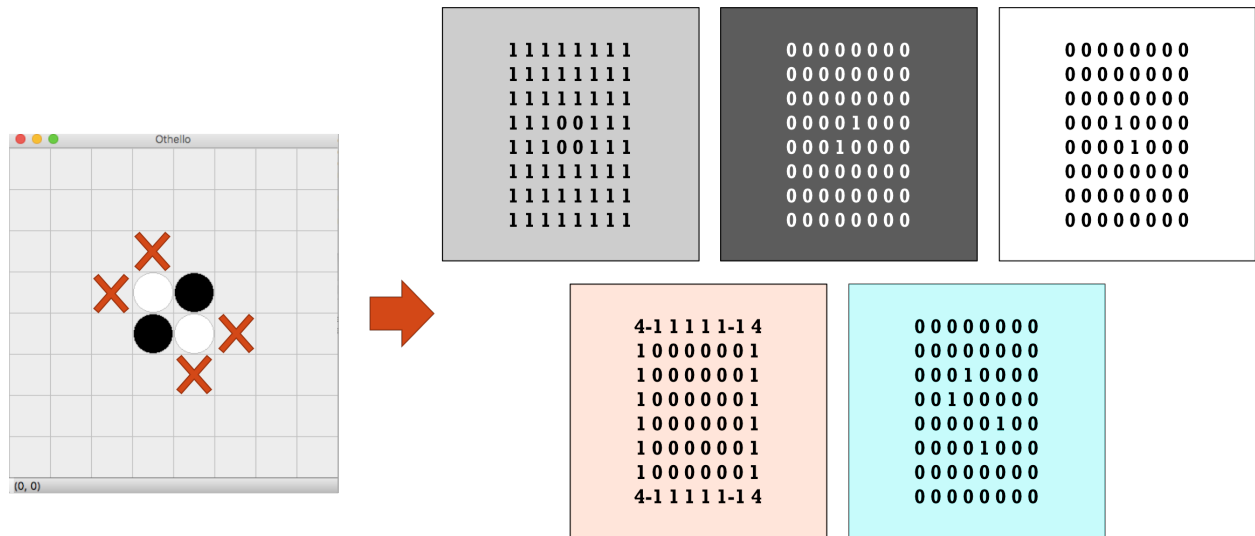


Figure 11. Full input matrix when “Corner, Border and Freemode” is chosen as feature extraction

Training and Testing

By setting random seed into different values, we were able to generate 7 sets of data (with the first set produced by setting seed as 1, second set with its seed as 2, so on). All our 31 neural networks are trained using the first set with 0.2 of its data reserved as testing data.

Early Stopping. It was soon discovered that our network would train for unproductively long period of time. Setting the correct epoch became crucial in order for us to sufficiently train our network while not wasting time that could be used to test more hyper-parameters. Luckily it was soon discovered that early stopping with a patience of 5 was enough to train a network sufficiently well performing, especially with an independent validation set.

Hyper-parameter Tuning. Having dealt with early stopping, the epoch basically became a non-factor and we experimented with different learning rate, momentum, dropout rate and nodes per layer. It was soon discovered that a slight variation of learning rate from 0.001 and momentum from 0.9 was devastating. It was then decided that these two hyper-parameters would not be changed in future experiments.

In comparison, the relationship between dropout rate, nodes per layer and MSE was relatively unclear. While we learnt that in general, higher dropout rate decreases overfitting at the cost of higher MSE, and nodes per layer does the exact opposite, there were still considerable exceptions. As a result, we fall back to testing all combinations.

Testing. Using the data set generated from different seeds, we were able to provide near independent data set for the purpose of testing. The result was clearly observable with the differences between the MSE of the 6 testing sets usually in 10^{-3} , while the MSE of the first set differs from the others in the scale of 10^{-2} .

Set 1	Set 2	Set 3	Set 4	Set 5	Set 6	Set 7	Avg (2-6)	Diff with Set 1
0.6966243	0.7338179	0.7329133	0.7338478	0.7340187	0.7358826	0.7329159	0.7339	-0.037
0.675049	0.7297714	0.7288691	0.7302293	0.7306241	0.7316783	0.727873	0.7298	-0.055
0.7146764	0.7392533	0.7383323	0.7387331	0.7398558	0.7410367	0.7377851	0.7392	-0.024
0.7457019	0.760025	0.7586022	0.7592436	0.7592714	0.7605961	0.7591491	0.7595	-0.014
0.6920794	0.7302734	0.7289112	0.730615	0.7304122	0.7317721	0.7280188	0.7300	-0.038

Figure 12. MSE results of neural networks using “corner and border” as feature.

As we now have a fair judgement for MSE and overfitting, we eventually discovered the best performing hyper-parameter to be the combination of (Nodes per layer = 512, CNN dropout = 0.3, Dense layer dropout = 0.3). We then train the network using the entire Data Set 1 and the combination of 7 Data Sets to observe the difference in performance.

Graphical User Interface

To construct our GUI, PyQt has been chosen for its simplicity and crisp response. Different features are highlighted below and our rationale explained.

Core Functionality

As mentioned in **Architecture – Core Program**, the flow management, or the integrity of the game is handled by another module. Hence core functionality in this case has its scope limited within GUI, which explicitly stands for accepting user input and responding to the degree that permits the progression of a game.

Mouse Position. To obtain the mouse position, it was as simple as unpacking the mouse position from *MouseMoveEvent*, then calculate the relative grid position by dividing it over the width of each square.

Disc and Grid Representation. To present the state of the current board, our program simply draws lines that form the grid of the board, and white and black circles which form the discs on the board. This is refreshed after every move to display the correct state.

Immediate Feedback

In order for our program to have better user experience, it was essential to provide immediate feedback at every movement done by the user.

Hover Position. At every *MouseMoveEvent*, our program sends feedback to the window after calculating mouse position as a string. Hence whenever the user moves its mouse to a new square, he/she would notice immediately and ensure that the program is responsive.

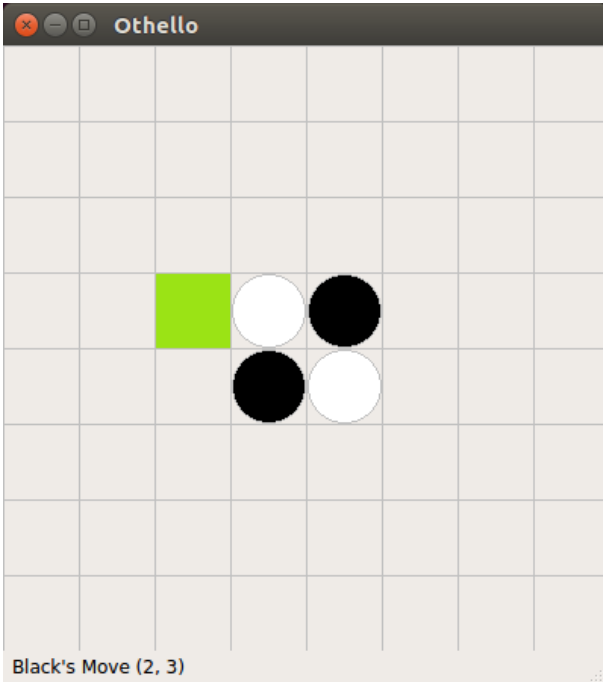


Figure 13. As the mouse hovers above (2, 3). A string would indicate that black is trying to make a move at (2, 3). Also, the square would be highlighted green as it is a valid move.

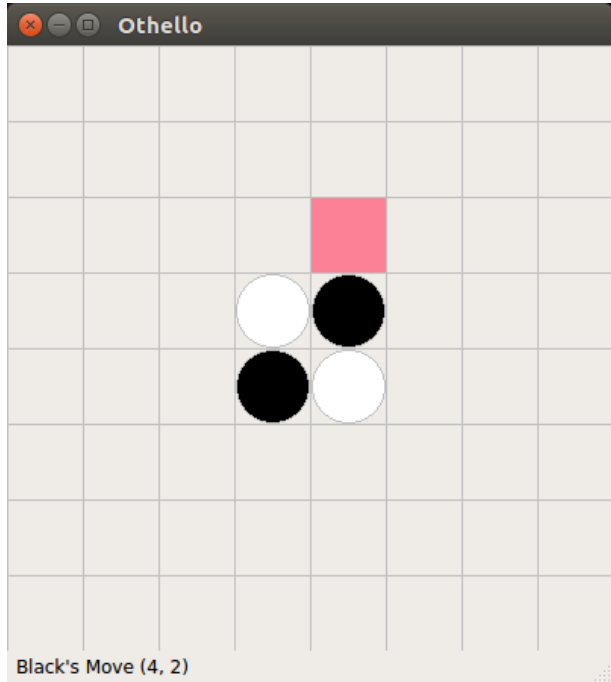


Figure 14. In this case, as the move is invalid, the square is highlighted red.

Move Validity. To further facilitate the user to determine whether a move is valid, our program also highlights the hovering square green or red, depending on the validity of placing a disc there. This aims at helping users make a move faster.

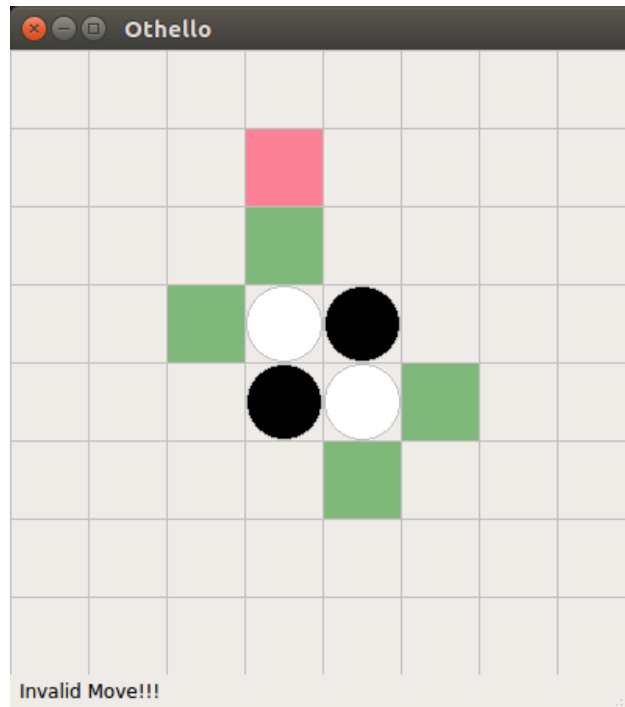


Figure 15. As the player attempts to play a move at (3, 1). A prompt would show saying that it is an invalid move. The available moves by black would also flash as a result.

Available Moves. Whenever an invalid move is played by the user, all the squares that are legal moves would be flashing in green to remind him/her the correct moves. This is another feature devised to facilitate user experience.

Game Tree Expansion

For our program to utilize an accurate neural network, an effective and fast tree search is essential to achieve improvement in performance. As such, two core concepts were inherited from the literature review of Zebra, namely alpha-beta pruning and move ordering.

Core Concept

Alpha-Beta Pruning. The common concept of alpha-beta pruning is still closely adhered to reduce tree traversal time. As soon as the value of a child node is lower or greater than the value of a node's sibling, the expansion would stop at once and continue at its sibling or parent.

Move Ordering. Every time a node is explored, its child node would be ordered according to its value, with a reversed order if the node is in a maximizing layer, or increasing order if it is a minimizing layer.

Minimax. The core concept of minimax still holds, however since the evaluation (ie. value network prediction) can be done in both ways, we have arbitrarily set the evaluation to be conducted in the computer's color at every move. This way, the game tree is more or less sandwiched layers of min and max layers.

Repeating Player. In Othello, it is possible for one player to play multiple moves in a row. Hence at every calculation, instead of a fixating alternation of min and max layer, we have to check whether the current player is the original player to determine if we want to minimize or maximize at that layer.

Implementation

To implement our concept, we added two classes, namely *Tree* and *Node*. *Tree* would be responsible for reporting to the main program and providing an overview of the status of the game tree, whereas *Node* tracks the relationships of nodes with their parents and children.

Tree.

getBestMove. Tree returns the best move when queried by returning the “left-most” children of the current node (root). As the algorithm should keep the children of all nodes in order, the “left-most” children would hence always be the best move. If the depth of the current node happens to be less than 4, our program would then block itself until the tree has been expanded to a depth of 4.

updateMove. Our tree simply overwrites the current node with its appropriate children, effectively reducing the depth of all nodes by one at the same time.

Node. As the design of the *Node* class is largely dependent on the design choice of multithreading, most details would be left for the next section.

Evaluate. All nodes would only access value network evaluation once in its lifetime. Afterwards, the value is stored in each node’s internal variable and returned whenever queried. This is to reduce the number of times needed for our program to access the hefty calculation of neural networks.

Multi-threading

To further reduce response time, a method for background expansion of the game tree was devised. Two requirements were needed for this method to be feasible. Firstly, the process should be done in a multithreading manner, so that the GUI would remain crisp and responsive throughout the duration. Secondly, the expansion should be consisted of small steps, so whenever a move is required from the tree, a response could be given as soon as possible.

Bottom-Up Approach

Looking into minimax with alpha-beta pruning, we see that the usual game tree search can already be seen from a bottom-up approach. While it is most commonly implemented as a top-down recursive function, we can easily see that a bottom-up, divide-and-conquer approach should work just as well, as long as we can locate the correct node to expand afterwards.

Leaf. For a leaf node, it should start by creating all its children nodes. After all its children have been evaluated, it should be able to calculate the minimum (or maximum) across all children and rank them in order. Afterwards, it should return its immediate sibling for expansion. If there is no more sibling to be expanded, it should return its parent which can then calculate the maximum (or minimum) across all siblings and order them accordingly.

Inner Nodes. For inner nodes, the process should be the same besides the fact that all children should have been created and evaluated, since our algorithm goes bottom-up. Hence, the progress becomes simply retrieving the value, finding the minimum (maximum) value and putting the children nodes into order. It should also then attempt to return the next sibling or parent.

Root. For the root of a fully expanded tree, the next move is of course to increment the search depth. To accomplish that in our bottom-up approach, the next node should be the left-most leaf node.

Hence, after doing the necessary calculations, that is finding minimum (or maximum) and ordering the immediate children nodes, our program traverse recursively to reach the left-most leaf node.

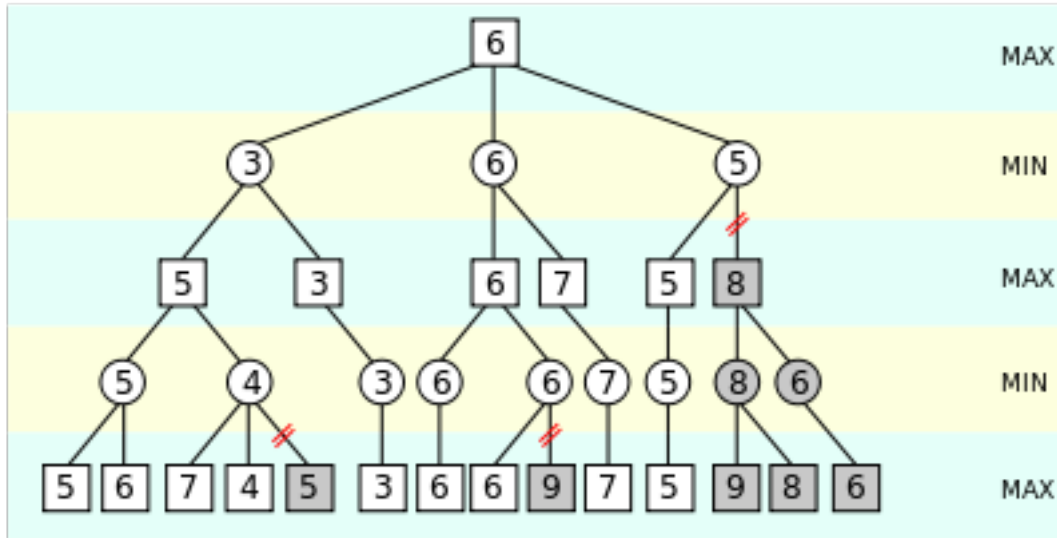


Figure 16. Illustration of Alpha-Beta Pruning

Operations Beside Expansion

Things are a little more complicated when we attempt to change the tree structure while another thread is rapidly expanding it. These scenarios are generally handled in two manners.

Next Node. Finding the next node becomes harder when the tree structure changes from the root level. Luckily all scenarios turn out to be classifiable into two. Either the *nextNode* pointer was travelling in the subtree chosen, or it was travelling elsewhere. If it was travelling elsewhere, then the chosen subtree is a complete one and hence should be expanded from the top (or at the left-most leaf node). If it was travelling within the chosen subtree, then things would simply work if we continue to do so, as long as we remembered to discard the reference to the previous state of game (ie. the new root's parent node).

Lock. To facilitate the access from two different threads – one trying to expand it and the other trying to reduce depth or determine which is the best move, a lock is obviously needed. However, locks in Python are not fair, in a sense that the threads that waited longer do not have higher chances of acquiring the lock than other threads. This makes our program highly unresponsive at times. To combat this problem, we introduced a global Boolean, which when set to true, makes the expansion thread sleep for 0.3 seconds, sufficiently long enough for either *getBestMove* or *updateMove* to acquire the lock and return responsively.

Pruning

To mimic the effect of alpha-beta pruning, after expanding a child node, it should see if this updates the parent's alpha and beta value such that further expansion is unnecessary. If this is the case, in our bottom-up approach, the node should return its parents, instead of sibling as the next node. This effectively mimics the early termination of exploring nodes and by letting the parent take control, should terminate the expansion and pass on the control in a more elegant manner.

Experiments and Results

With the effect of various hyper-parameters on MSE discussed in earlier section **Neural Network – Training and Testing**, we would only focus on how various training method results in different playstyle of our program.

Border Only

While our program was loaded with a network trained only with an extra layer highlighting the border nodes, it has an apparent liking towards long, straight, diagonal lines. This is perplexing at first but the reason becomes clearer as we gain understanding in the game of Othello. In early stages where players try to limit their disc count, a diagonal expansion is usually more favorable as it is along the initial axis, such that opponent would have a harder time trying to eliminate every disc in subsequent move. A diagonal expansion is also usually the move to break through after confining one's disc count, as it opens up the most choices in available move.

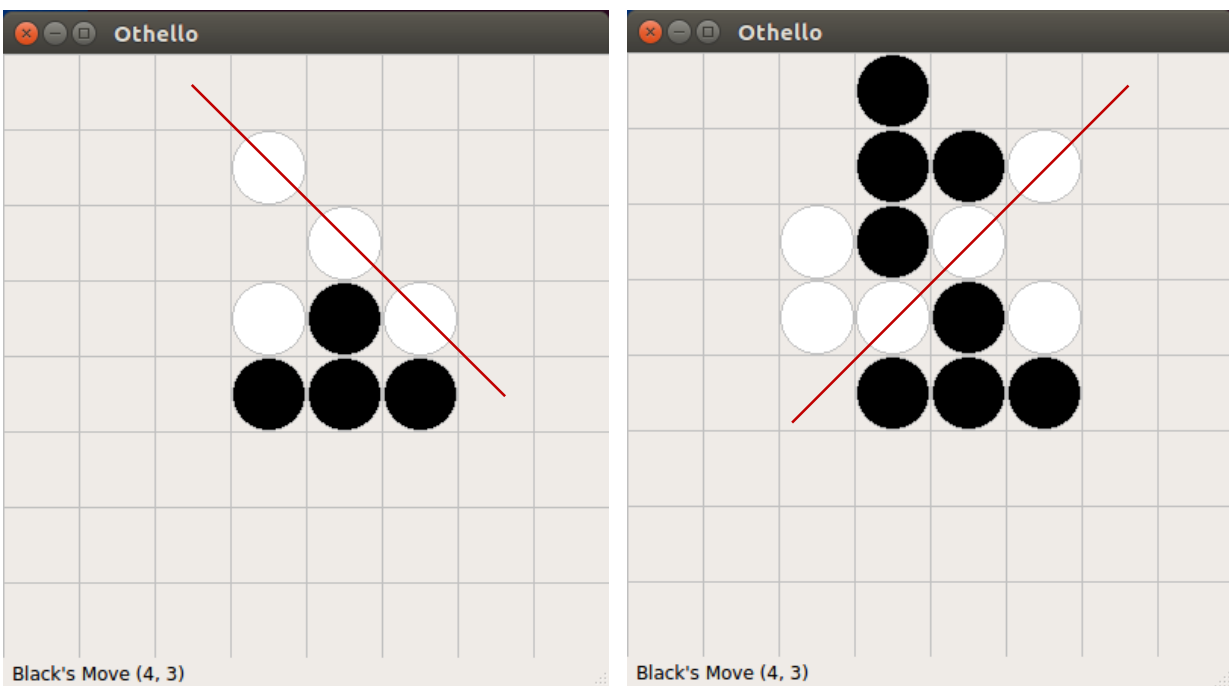


Figure 17, Figure 18. Examples of affection towards diagonal expansion.

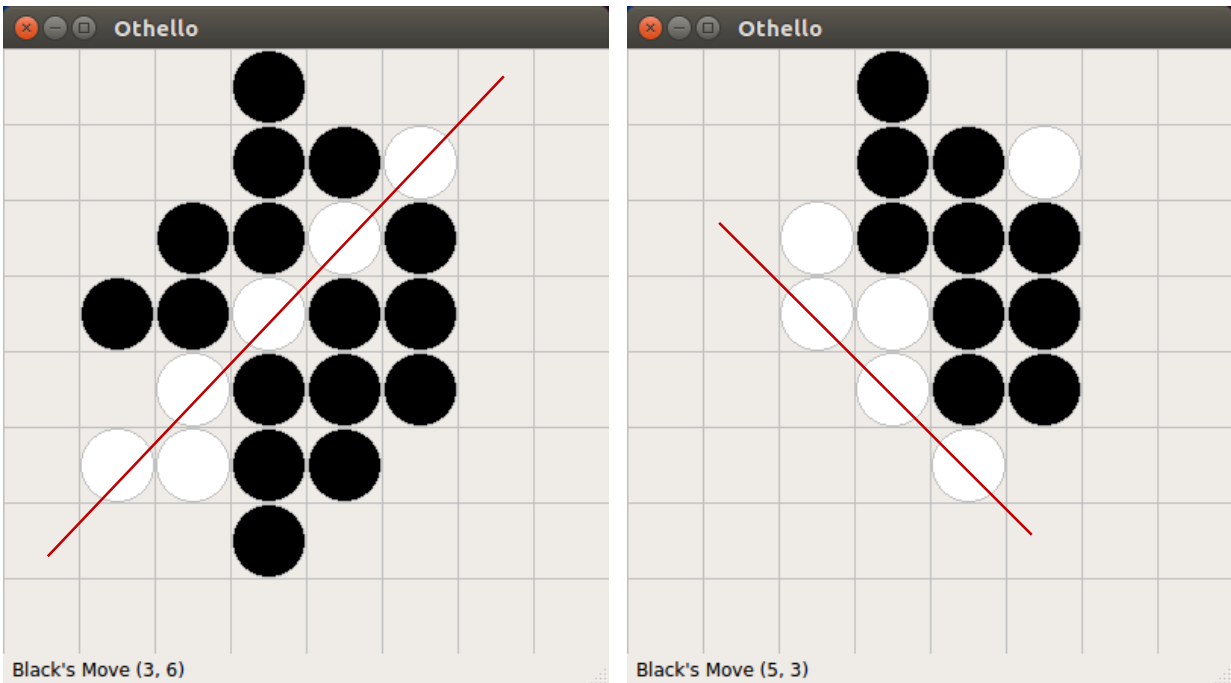


Figure 19, Figure 20. Examples of affection towards diagonal expansion.

However, our network fails to acknowledge the importance of corners and always expand into the X-squares (squares diagonal to corners), resulting in an easy capture of corner by human testers. This can be regarded a result of CNN's tolerance towards transposition and magnification, and hence while professionals keep their diagonal expansion off the main axis and short, our network incorrectly favors every diagonal expansion alike. This makes our program a very weak player overall.

Border and Corner

When a border and corner network is loaded into our program, its patterns become much less predictable with a mild avoidance of squares adjacent to corners. However, it is very aggressive towards getting the early discs on the border and as a result usually opens up a lot of liberty to the opponent.

This play style either rewards our program as it gains a corner and expand its stable discs aggressively, or punishes it as it loses a corner and all the border discs that were gained earlier became opponent's stable disc.

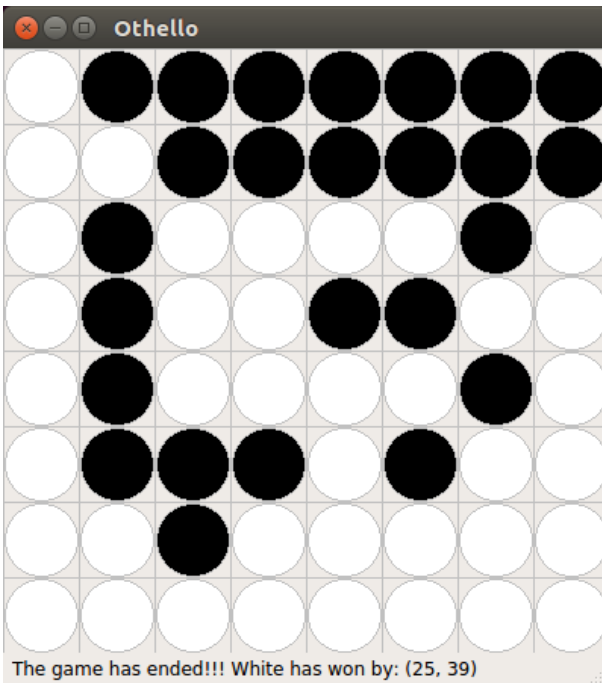


Figure 21. Cases where an aggressive move was rewarded.

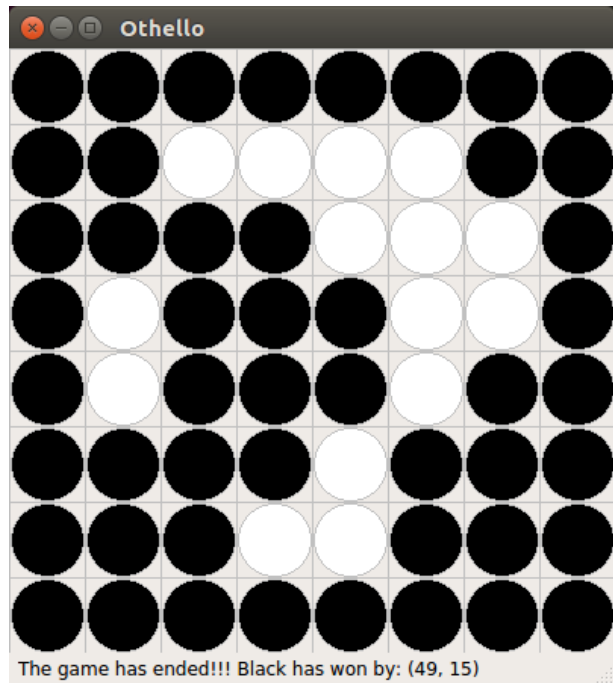


Figure 22. Cases where an aggressive move was punished.

However, there are also many cases where our program manages to hold the ground even as it loses corner, sometimes even winning after losing the first two corners. Overall, neural networks trained with an extra padding layer of border and corners create unpredictable results.

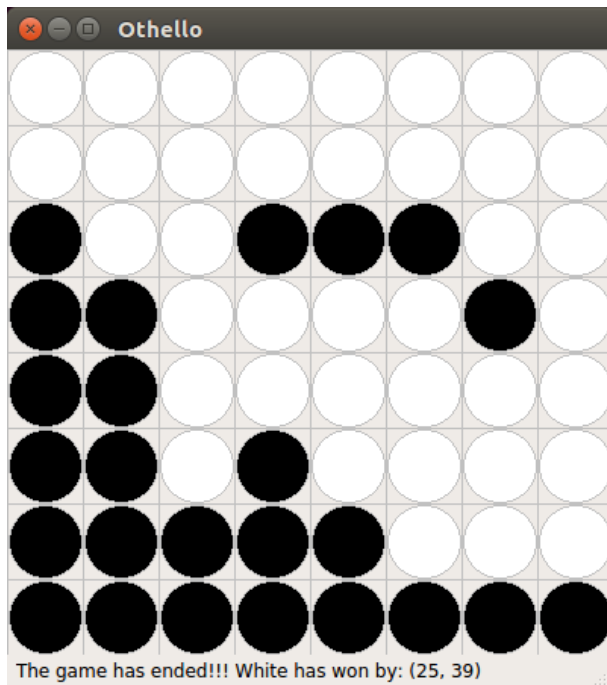


Figure 23. Our program winning after losing the first 2 corners.

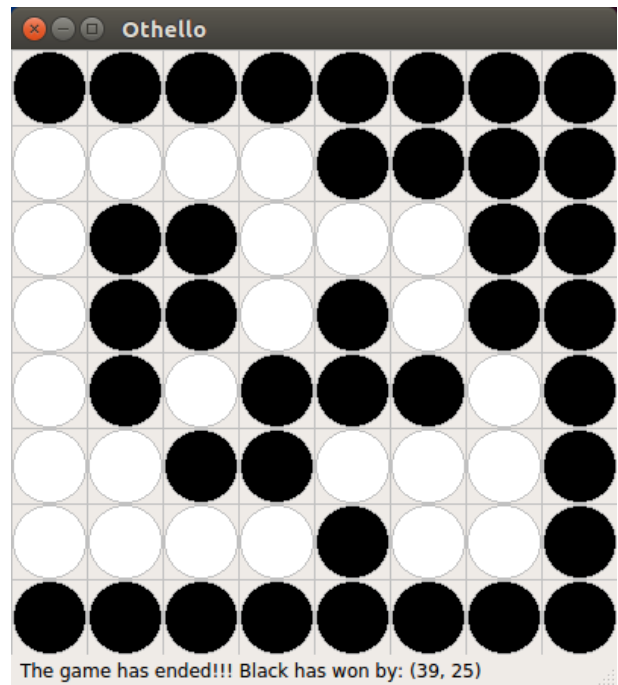


Figure 24. Our program holding massive ground in the middle while all 4 corners are lost.

Border, Corner and Freemove

When freemove is passed in as a feature, the first thing noticed is an immediate boost of playing strength. The program is still being able to value corners and borders from the convolution of the positional layer and opponent's freemove layer, but usually values liberty and freedom of movement over simple positional advantage. Our program has one remarkable feat reported by many testers, which is the ability to give up corners but limits the human players' choice of moves significantly, such that little stable discs was gained from getting that corner and in the end, still manages to snatch a win despite losing first corner.

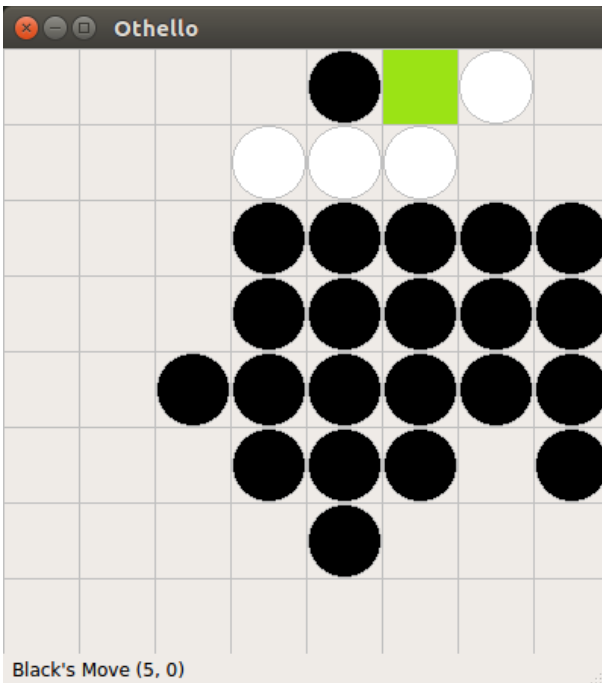


Figure 25. Black has only 3 liberties as a result of white's maneuver.

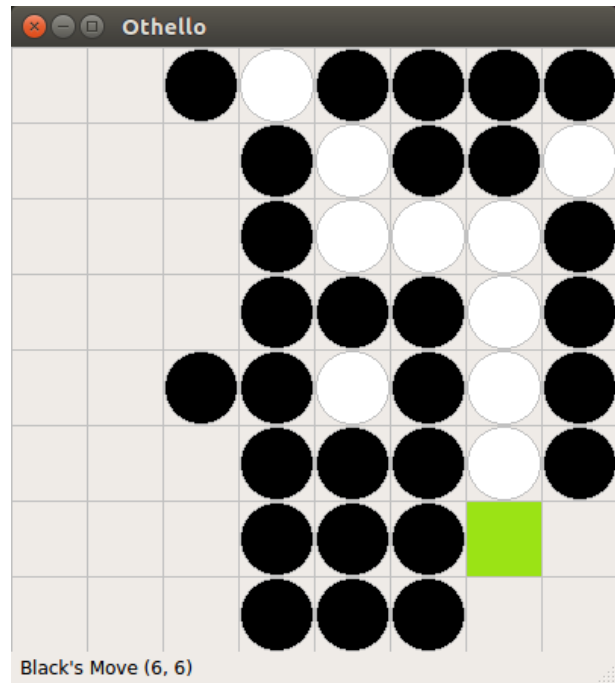


Figure 26. Black is forced to play into C-square and X-square after taking the upper left corner.

For example, in the left figure (fig. 25), our program has successfully limited the number of moves black can make by limiting its own disc count. In this case black only has 3 moves possible, 1 of which would lead to a loss of control of the upper border, and further reducing its liberty to 1.

The other figure (fig. 26), shows a later stage of the game evolved from the left one. Here we can see that white has given up the upper right corner but while doing so, limits black's possible move to 2, both of which are adjacent to the lower right corner square. We can also see that our program has slipped 2 of its discs between black's border, such that it cannot expand its stable disc efficiently. By losing the upper right corner, our program has gained advantage in liberty, which results in the lead in game and one of the other corners while limiting black's gain all at once.

Justification and Explanation

The justification to various choices made for our program and project is provided here.

Why is Othello a suitable game?

Similar to Go and Chess, Othello is a zero-sum, perfect-information, partisan, deterministic strategy game. This means that it is a game with no win-win situation, no concealed information (and thus no bluffing), practically equal position for both players and no randomness involved. This type of game is ideal for computers to play and is thus one of the reason it was chosen to be the topic. Also, unlike Chess but similar to Go, the values of pieces in Othello are determined only by their positions and there are no face values unique to each piece. This makes it easier for us to adapt AlphaGo's algorithm (in particular CNN) to our program, and also to compare them with each other.

Why is policy network not used?

Unlike Go, Othello has a much smaller degree of freedom at each move. As such, policy network is not as necessary and was abandoned due to time constraint. Also, the requirement of accuracy of policy network is much higher when we are not using Monte Carlo Tree Search, as we may accidentally prune off an entire valid branch if we combine traditional tree search with policy network. We also think that since Multi-Prob cut would always remain an option if we have an accurate enough value network. Given the situation, it would hence be better for us to focus on one network only. In the end, the search rate was satisfactory using Alpha-Beta Pruning alone.

Why is Python being used?

The reason is three folded. First of all, Python has always been used for machine learning. As such, it has lots of libraries readily available for us to prepare and fine-tune our neural networks. Also, the large

machine learning community allows us to spend less time troubleshooting as there would always be answers to my queries and fixes to my problems.

Secondly, Python as a language is remarkably simple to code and ideal for prototyping. For example, its list traversal, dictionary and memory management all allows users to achieve what they want with minimal coding. It can also be used via interpreter, which allows users to extract variable values at any time, ideal for doing debugging. These features all make Python the ideal language for a 1-year project.

Lastly, Python has support for both GUI and parallel computing. Thanks to its large number of libraries available, it can interact with users via PyQt, while also accessing the computation power of graphics card with Tensorflow. These 2 features are exactly what an AI program playing against human wants.

Why is reinforcement learning not being used?

The main reason is the limitation of time. There are two approaches using which we could generate data for our networks to learn from. The first one would be playing against humans, but since our network only started to mature late in March, it was unlikely for us to collect significant number of games such that the effect of reinforcement learning would be observable and could be studied.

The second approach would be playing games with earlier iterations of itself. Again, this requires us to reprogram the interface. As we had the more urgent task of fixing the human-computer interface back then, this idea was also postponed and eventually abolished.

Why are more complicated features not being used?

As mentioned in the literature review, some more complicated features such as *pattern* and *stable discs* were used by human players and the Zebra program. AlphaGo also used features such as *Turns since*,

Capture size, Self-atari size, Ladder capture, Ladder escape, Sensibleness, etc. These more complicated features were not used for two reasons. The first and more trivial reason is the limitation of computation and time resource. Accommodating these features would simply be too much for our program and defeats the purpose of using a simpler game. The second reason is that this somewhat defeats the purpose of using neural network. A program targeting a complicated game, such as Go, deserves every help it can get to overcome the everlasting obstacle in Artificial Intelligence, as the game itself is hardly a solved game. However, doing the same for Othello would simply be overdoing the matter, since AI programs such as WZebra has already defeated human professions decades ago.

Conclusion and Future Work

Having created a program that can defeat regular human players, I honestly felt that I have learnt a lot in the course of this project. Following the footsteps of AlphaGo, I have learnt how to implement the popular neural networks in the area of AI gaming. I have also understood the principles to a stage where I could make adjustments to its architecture and adapt it to another game. Venturing off the path, I have learnt how feature extraction can affect performance and behavioral pattern significantly; I have also investigated a parallel approach of deterministic tree search, different from the MCTS suggested by AlphaGo.

Despite all the accomplishments, I experience an even stronger feeling of emptiness for not able to deliver a perfect program that plays Othello. These regrets are summed up as follow, such that future work, conducted by others and myself, could be expedited and the torture and tumble of early stages could be avoided.

Architecture

To better the current architecture, I believe that the game flow control, GUI and AI components should be separated and communicated using signals. They should run on 3 different threads inherently. This way, our program can play with other programs with the help of a wrapper, or it could play with earlier iterations of itself by creating 2 AI components, each loading a different network. This also allows us to easily code the background expansion of game tree.

Neural Network

For our neural network, I believe that the accuracy can be further improved. While the reasons for not using more complicated features were provided above, it is still the most promising approach if the goal for future project is simply to increase winning rate of the program.

Feature Extraction. Stable discs can form another layer that could be fed to our neural network. Also, the pattern where two discs on the border are separated by one square and two squares should also be highlighted as they form a zone of vulnerability.

Reinforcement Learning. After each game, the program could fit that particular game by itself immediately, although this would require us finding the suitable learning rate. It would also be interesting to observe the effect of training a network specifically to predict the opponent's move. In theory, this should allow our program to search to deeper depths and make state predictions more accurate.

Policy Network. Policy network would undoubtedly be a suitable direction of investigation next time the project is continued or restarted. In theory, it should lead to a deeper and more efficient tree search.

User Experience

To deliver a better user experience, match history could be displayed at a side panel and a click at it would bring the user back to that particular state of game. The program should also provide an option for the user to save the match history down. This would also help us find the weaknesses of our program.

Tree Traversal

Various tree traversal techniques such as Monte Carlo Tree Search (MCTS) and Multi-Prob cut can be adopted if this project were to be continued.

References

1. Johnson G. To test a powerful computer, play an ancient game. The New York Times [Internet]. 1997 Jul 29. [cited 2016 Sep 17]; Technology: [about 5 screens]. Available from: http://www.nytimes.com/1997/07/29/science/to-test-a-powerful-computer-play-an-ancient-game.html?_r=0
2. West P. [updated 2016 Sep 07, cited 2016 Oct 26]. Available from: http://www.bnext.com.tw/ext_rss/view/id/1847944
3. Silver D, Huang A, Maddison CJ, Guez A, Sifre L, van den Driessche G, Schrittwieser J, Antonoglou J, Panneershelvam V, Lanctot M, Dieleman S, Grewe D, Nham J, Kalchbrenner N, Sutskever I, Lillibrap T, Leach M, Kavukcuoglu K, Graepel T, Hassabis D. Mastering the game of Go with deep neural networks and tree search. Nature [Internet]. 2016 [cited 2016 Sep 17]; 529. doi:10.1038/nature16961
4. Jeff B. [updated 2015 Sep 07, cited 2016 Sep 18]. Available from: <https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/>
5. Strategy Guide for Reversi & Reversed Reversi [updated 2011 Jan 04, cited 2017 Apr 16]. Available from: <http://www.samssoft.org.uk/reversi/strategy.htm>
6. Andersson G. Writing an Othello program [updated 2007 Apr 02, cited 2017 Apr 16]. Available from: <http://www.radagast.se/othello>

Neural Network Structure

Default Neural Network Structure

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 32, 8, 8)	1472
dropout_13 (Dropout)	(None, 32, 8, 8)	0
conv2d_8 (Conv2D)	(None, 32, 8, 32)	2336
dropout_14 (Dropout)	(None, 32, 8, 32)	0
conv2d_8 (Conv2D)	(None, 32, 8, 32)	9248
flatten_3 (Flatten)	(None, 8192)	0
dense_13 (Dense)	(None, 512)	4194816
dropout_15 (Dropout)	(None, 512)	0
dense_14 (Dense)	(None, 512)	262656
dropout_16 (Dropout)	(None, 512)	0
dense_15 (Dense)	(None, 512)	262656
dropout_17 (Dropout)	(None, 512)	0
dense_16 (Dense)	(None, 512)	262656
dropout_18 (Dropout)	(None, 512)	0
dense_17 (Dense)	(None, 64)	32832
dense_18 (Dense)	(None, 1)	65
<hr/>		
Total params:	5, 028, 727.0	
Trainable params:	5, 028, 727.0	
Non-trainable params:	0.0	

Note1: All layers have the following parameters {W_constraint = maxnorm (2); activation = 'relu'; border_mode = "same"} whenever applicable.

Table 1

Performance of Neural Networks (with identical dense layer and CNN dropout rate)

Network #	Nodes Per Layer	Dropout Rate	Feature	Independent Testing MSE	MSE on Training Set	Diff.
/	512	0.3	Border	0.7427	0.7058	-0.037
1	1024	0.3	Border	0.7312	0.6624	-0.069
2	1024	0.4	Border	0.7331	0.6886	-0.044
3	1024	0.5	Border	1.0817	1.0802	-0.002
4	512	0.4	Border	0.7576	0.7496	-0.008
5	512	0.3	Corner and Border	0.7339	0.6966	-0.037
6	1024	0.3	Corner and Border	0.7298	0.6750	-0.055
7	1024	0.4	Corner and Border	0.7392	0.7147	-0.024
8	1024	0.5	Corner and Border	0.7595	0.7457	-0.014
9	512	0.4	Corner and Border	0.7300	0.6921	-0.038
10	1024	0.4	Freemove, Corner and Border	0.7230	0.6394	-0.084
11	1024	0.3	Freemove, Corner and Border	0.7207	0.6666	-0.054
12	1024	0.5	Freemove, Corner and Border	0.9178	0.9094	-0.008
13	512	0.3	Freemove, Corner and Border	0.7191	0.6542	-0.065
14	512	0.4	Freemove, Corner and Border	0.7393	0.7262	-0.013

Note1: All the above networks are trained with the following values in unspecified hyper-parameters.

{Max_Epoch = 500; Learning Rate = 0.001; Momentum = 0.9; Decay = Learning Rate / Max Epoch; Batch Size = 5}

Note2: **Independent testing MSE** refers to the average MSE from testing on data sets generated using different random seeds (2 – 7). **MSE on training set** refers to the MSE calculated when testing on training set, validation set and testing set as a whole, usually this is Train_1 unless specified otherwise. **Diff.** refers to the difference in the previous two values, usually taken as an indicator of overfitting.

Note3: Best performing (taking average MSE, overfitting index and game performance into consideration) neural networks are bolded.

Table 2

Performance of Neural Networks (with different dense layer and CNN dropout rate, using freemove, corner and border as feature selection)

Network #	Nodes Per Layer	CNN Dropout Rate	Dense Layer Dropout Rate	Independent Testing MSE	MSE on Training Set	Diff.
15	1024	/	0.4	0.7525	0.6874	-0.065
16	1024	0.3	0.4	0.7231	0.6907	-0.032
17	1024	0.3	0.5	0.7383	0.7254	-0.013
18	1024	0.4	0.3	0.7318	0.7149	-0.017
19	1024	0.4	0.5	0.7430	0.7339	-0.009
20	1024	0.5	0.3	0.8176	0.8116	-0.006
21	1024	0.5	0.4	0.7643	0.7567	-0.008
22	512	0.3	0.4	0.7278	0.6723	-0.056
23	512	0.4	0.3	0.7549	0.7425	-0.012
24	512	/	0.3	0.7719	0.6125	-0.159
25	512	/	0.4	0.7624	0.6068	-0.156

Note1: All notes in Table1 apply.

Note2: All variations have lower or similar performance compared with network13 and network12 and hence it was decided that in this particular case, asymmetric dropout rates have no positive effect on decreasing MSE.

Table 3*Performance of variations of network 13 (with different number of CNN and Dense layers)*

Network #	# of CNN layers	# of Dense Layers	Independent Testing MSE	MSE on Training Set	Diff.
26	3	4	0.7202	0.6904	-0.030
27	2	5	0.7237	0.6654	-0.058
28	3	5	0.7238	0.6962	-0.028
29	1	4	0.7356	0.6256	-0.110
30	2	3	0.7309	0.6752	-0.056
31	1	3	0.7379	0.6007	-0.137

Note1: All notes in Table1 apply.

Note2: All variations have lower or similar performance than network13 and hence it was decided that in this particular case, adjusting the number of layers have no positive effect on decreasing MSE.

Table 4

Performance of variations of network 13 (using the entirety of Training Set)

Network Name	Nodes Per Layer	Average MSE From Testing Set	Average MSE From Training Set	Difference
final_network1	Used entire set 1 to train	0.7246	0.6916	-0.033
final_network2	Use training set 1 to 7 to train	0.6608	0.5330	-0.128
final_network3	Set corner value to 10 instead of 4	0.7181	0.6453	-0.073

Note1: All notes in Table1 apply. Also, all networks are trained with “Freemove, Corner and Border” as feature selection.

Note2: All variations either have lower performance in terms of MSE or suffer from serious overfitting. This was also later confirmed when loaded into our program to observe their actual performance in game.