Final Year Project Detailed Final Report

**Project Title:** Playing Othello by Deep Learning Neural Network

**Author:** Kuai Yu (David)

**Supervisor:** Prof. K.P. Chan.

**UID:** 2013507514
Abstract

On March 15th 2016, the Go-playing computer program known as AlphaGo took the world by surprise as it won its fourth and final game against international Go champion Lee Sedol [1]. To see whether AlphaGo’s recent achievement can be recreated for a different board game, this project aimed to create a game-playing artificial intelligence program that uses similar deep learning techniques as AlphaGo but instead of Go, the program plays the much simpler board game Othello using a combination of tree search algorithms and neural networks. The program is implemented in Python and offers a simple graphical user interface to play against human challengers.
Acknowledgement

My whole-hearted gratitude goes to Dr. Kwok-Ping Chan, Associate Professor of the department of Computer Science, University of Hong Kong for advising me on this project. Professor Chan is an expert in the field of machine learning and has been most supportive in providing me with the resources necessary to get my project off the ground as well as helping me with any difficulties I encountered along the way.

Additional Acknowledgement goes to Argens Ng for generously providing me with the Othello World Cup dataset for training.

Abbreviations

AI – Artificial Intelligence
API – Application Programming Interface
CNN – Convolutional Neural Network
CPU – Central Processing Unit
CSV – Comma Separated Values
CUDA – Compute Unified Device Architecture
CV – Cross Validation
GPU – Graphics Processing Unit
MSE – Mean Squared Error
PolicyNet – Policy Neural Network
RAM – Random Access Memory
ReLU – Rectified Linear Unit
ValueNet – Value Neural Network
# Table of Contents

Abstract.................................................................................................................................................. 1
Acknowledgement ................................................................................................................................... 2
Abbreviations.......................................................................................................................................... 2
I. Introduction & Literature Review ........................................................................................................ 8
   Project Background ............................................................................................................................... 8
   Project Objectives & Deliverables ....................................................................................................... 10
   Existing Othello Programs .................................................................................................................. 11
II. Project Methodology ......................................................................................................................... 12
   Hardware & Software Requirements .................................................................................................... 12
   Design & Implementation ..................................................................................................................... 12
      Hardware/Software Preparation Phase ............................................................................................. 13
      Training Data Collection Phase ....................................................................................................... 13
         Recording & Parsing Games – Original Set .................................................................................... 14
         Recording & Parsing Games – World Cup Data Set ..................................................................... 15
         Reconstructing Sample Games ...................................................................................................... 15
      Neural Net Construction Phase ...................................................................................................... 16
         Development Platform .................................................................................................................... 16
         ValueNet Introduction .................................................................................................................... 17
         ValueNet Development Process .................................................................................................... 17
      Training Phase ................................................................................................................................. 18
      Validation Phase ............................................................................................................................... 18
         KFoldCV ....................................................................................................................................... 19
         GridSearchCV ............................................................................................................................... 19
         RandomizedSearchCV .................................................................................................................... 20
      Game Logic Construction Phase ...................................................................................................... 21
         Minimax Algorithm with Alpha-Beta Pruning .................................................................................. 21
         Monte-Carlo Tree Search with Fast Rollout move somewhere else? ........................................... 23
      Graphical User Interface Construction Phase .................................................................................. 23
   Detailed System Architecture ............................................................................................................ 24
   Object Diagram .................................................................................................................................... 24
   Files ..................................................................................................................................................... 24
Risks, Challenges & Mitigation

Risk #1: Overfitting During Regression
Risk #2: Value Network Unable to Detect Board Edges
Risk #3: Insufficient Training Data
Challenge #1: Value Network is Difficult to Debug.
Challenge #2: Inter-Modular Communication
Challenge #3: Transition from Jupyter Notebook to PyCharm
4. Othello Graphical User Interface ................................................................. 40

III. Experiments & Results .................................................................................. 41
Training Results ................................................................................................. 41
Mean Squared Error .......................................................................................... 41
Hyperparameter Tuning Results ......................................................................... 43
Correlation Between Search Depth and Time Taken ......................................... 46
Overall Remarks ............................................................................................... 47

IV. Conclusion & Future Works ........................................................................... 49
Project Overall Review ..................................................................................... 49
Potential for Extension ....................................................................................... 49
Policy Net ........................................................................................................... 49
API ..................................................................................................................... 51
Appendix ............................................................................................................ 52
Works Cited ....................................................................................................... 57
# List of Figures

**Figure 1.** Basic principles of machine learning ................................................................. 8  
**Figure 2.** The importance of corners ...................................................................................... 9  
**Figure 3.** The value network ................................................................................................. 10  
**Figure 4.** The contents of an Othello game file ..................................................................... 14  
**Figure 5.** The contents of WCD.txt ....................................................................................... 15  
**Figure 6.** The number of files in ./Dataset/WCD_DATA ...................................................... 15  
**Figure 7.** Logistic regression model using Python ................................................................. 16  
**Figure 8.** IagoBot_ValueNet.ipynb output ............................................................................. 17  
**Figure 9.** KFoldCV ................................................................................................................ 19  
**Figure 10.** Hyperparameter tuning process #1 ................................................................. 20  
**Figure 11.** Hyperparameter tuning process #2 ................................................................. 21  
**Figure 12.** Minimax with Alpha-Beta Pruning ................................................................. 22  
**Figure 13.** IagoBot GUI ........................................................................................................ 23  
**Figure 14.** The contents of an Othello game file ................................................................. 24  
**Figure 15.** The class definition of BoardState ................................................................... 25  
**Figure 16.** value_net_x_input’s shape .................................................................................... 25  
**Figure 17.** value_net_y_input’s shape .................................................................................... 25  
**Figure 18.** OthelloGameStates class definition ............................................................... 27  
**Figure 19.** IagoBot GUI start-up interface .......................................................................... 29  
**Figure 20.** IagoBot GUI menu bar items ............................................................................. 29  
**Figure 21.** game_states_reader.ipynb output #1 ............................................................. 29  
**Figure 22.** game_states_reader.ipynb output #2 ............................................................. 29  
**Figure 23.** othello_utility.py ............................................................................................... 30  
**Figure 24.** Underfitting & overfitting .................................................................................... 31  
**Figure 25.** Input layer padding ............................................................................................. 32  
**Figure 26.** othello_game_reconstructor_screenshot #1 ....................................................... 36  
**Figure 27.** othello_game_reconstructor_screenshot #2 ....................................................... 36  
**Figure 28.** othello_rank_flipper.ipynb ..................................................................................... 36  
**Figure 29.** WCD_letters_to_num_converter.ipynb ............................................................ 36  
**Figure 30.** WCD_data_to_txt_converter_screenshot #1 ..................................................... 36  
**Figure 31.** WCD_data_to_txt_converter_screenshot #2 ..................................................... 37  
**Figure 32.** othello_game_reconstructor_screenshot #1 ....................................................... 37
Figure 33. othello_game_reconstructor screenshot #2
Figure 34. seed random variable in iagoBot_ValueNet.py
Figure 35. baseline_model in iagoBot_ValueNet.py
Figure 36. baseline_model.fit()
Figure 37. RandomSearchCV tuning + fit
Figure 38. Tuning result summary
Figure 39. ValueNet model save function
Figure 40. baseline_model constructor
Figure 41. neg_mean_squared_error
Figure 42. iagoBot GUI start-up interface
Figure 43. 50-game set’s X inout and y label shapes
Figure 44. Actual vs. predicted labels chart, no training
Figure 45. Actual vs. predicted labels chart, with training
Figure 46. World cup dataset, # of unique states
Figure 47. othello_game_states_reducer output
Figure 48. ValueNet final MSE score
Figure 49. Softplus function
Figure 50. Gameplay screenshot, beginner difficulty
Figure 51. Gameplay screenshot, expert difficulty
Figure 52. ReLU function
Figure A. Othello Cell Labels
Figure B. Visual Input Features of ValueNet and PolicyNet
Figure C. Overall Architecture of ValueNet
Figure D. Overall Architecture of PolicyNet
Figure E. Project Object Diagram
Figure F. 6 endgame states at Beginner difficulty
Figure G. 6 endgame states at Expert difficulty

List of Tables

Table 1. Tournament results between different Go programs
Table 2. Running the program: Starting points
Table 3. Execution times of Minimax
Table 4. PolicyNet input planes
I. Introduction & Literature Review

Project Background

The AlphaGo program combines Monte-Carlo tree search with deep learning neural networks and is trained by datasets of both human-played Go games as well as a series of games in which AlphaGo played against other iterations of itself [2]. This unique architecture allows AlphaGo to effectively “learn” how to play Go over time and improve its odds of winning with every game it analyses. Furthermore, unique properties inherent to any deeply-layered neural network enables the program to “predict” the odds of winning for any particular board layout based off data extrapolated from past games and grants AlphaGo flexibility in playing.

This project was aimed at creating an artificial intelligence program much like AlphaGo in that multiple neural networks will be optimized based on a set of training data and then used to predict the winning moves in a completely independent game. A concise description of this process is shown in figure 1. However, instead of the game of Go, this program (henceforth named “IagoBot”) specializes in the game of Othello.

The layout of Othello is similar in appearance to that of Go. Except instead of a 19x19 board, Othello is played on a much smaller 8x8 board. The rules of Othello are also somewhat different: Although the two players still place alternating pieces of black or white stones on the board, a player may only place a stone if it forms a continuous horizontal, vertical or diagonal line with another stone of his color while sandwiching his opponent’s stones in the process. Once this happens, all sandwiched stones are “captured” and transforms into the player’s color. The game ends whenever the entire board is occupied by stones or neither player can make a legal move. At the end of the game, the player with the greater number of stones on the board wins. Otherwise it is considered a draw. One important rule every Othello beginner should take note of is the fact that the objective of a player at any given point is not to capture as many pieces as possible, but to cut down the amount of possible moves the opponent can make whilst trying to increase (or maintain) the number of possible moves the player can make. The reasoning behind this counter-intuitive strategy is explained in a research paper listed on the works cited page [4].
In terms of specific strategy for winning a game of Othello, one popular approach is corner-capture. To see why corner domination is crucial in a game of Othello, consider figure 2 where the black player has only 1 piece on the board whereas the white player dominates the rest of the board apart from the corner cells. In this situation, white can no longer make a move because any legal move requires that the white player capture one or more black pieces that have not previously been sandwiched between two white pieces. Black on the other hand, can play A1, H1, A8 and H8 in any order. In fact, after the final move, black will win the game 40–24.

![fig. 2: A winning position for black, who has control over all four corners of the board. Since corners are the only cells on the board that, once captured, can no longer be recaptured by the opponent, dominating them generally leads to a win.](image)

As a means of facilitating corner capture, the following cells are named in order of desirability or likelihood to lead to a corner capture: The A cells, B cells, C cells and X cells [5]. (See fig. A in Appendix). To allow an AI to intuitively understand the importance of capturing these corner and side squares, board states corresponding to end game positions are fed into IagoBot’s neural network. Since there is a high positive correlation between a game’s outcome and how many corner squares have been captured by the winning player, this approach is effective in encouraging the AI to seize corners whenever possible.

An interesting characteristic of Othello is that owing to the small size of the 8x8 board, it is entirely possible to construct a strong Othello program by utilizing only tree search algorithms that are capable of analyzing every potential outcome produced by any legal move from a given board position [6]. In fact, most existing Othello programs do not require any neural networks or subsequent “training” to attain a near-perfect level of mastery over the game. Nevertheless, a deep-learning neural network approach to the game warrants further investigation for two reasons: Firstly, it would provide valuable insight into the world of machine learning and what the possibilities are for it to transform existing AI technologies. Secondly, it would be interesting to see whether a deep-learning Othello program holds any significant advantages over traditional tree-search programs, namely whether it is possible to produce a program that is less exacting on physical resources such as CPU / RAM usage, as well as intangible resources such as time.
Project Objectives & Deliverables

The goal of this project was to produce an interactive program that would satisfy the following objectives and criteria toward the end of the project timeline:

Firstly, the key technology behind the program should be a neural network that works to guide the AI in making moves. Although the project specification stipulated the use of deep learning neural networks, it is nonetheless a good idea to analyze the advantages and disadvantages of other potential algorithms for machine learning (e.g. the perceptron model, the support vector machine, the decision tree and linear regression) to see how they compare to the neural network approach. Subsequent findings conclusively point to a neural network being the superior candidate for modeling IagoBot owing to the fact that it allows for a 64 dimensional input space (corresponding to the number of cells on an Othello board), which is where the perceptron and linear regression models fail, as well as its ability to consist of n-numbers of hidden layers in between the input and output layers so the model can be adjusted on the fly as opposed to the support vector machine which is a fixed parametric model.

Within Google’s AlphaGo program, there exists two neural networks: A ValueNet responsible for evaluating the position of a board in any given game for a selected player (in the form of a double) and a PolicyNet that makes decisions on which path to take in the game tree based on the value network. The value and policy networks are then combined in a tree-search algorithm. For this project, only the ValueNet was considered, and an evaluation of this part of the project is detailed in the methodology section.

A further objective which this project aimed to achieve is an effective approach to evaluating a board position at any given time during game play. This evaluation function must be able to take in the current board position, evaluate the positional advantages of both players and output a single double for a given player to be fed into the value network. Furthermore, the evaluation function must be able to project several layers beyond a particular node in the game tree to analyze the value state of future board positions and return those values to the

![fig. 3: The value network. Each node/neuron in this network takes a rudimentary value such as (-1, 0, 1), to indicate the state of that particular cell. This value is combined with an initialized weight value pertaining to that node and then passed into an activation function. Depending on the output of the activation function, nodes in the hidden layer(s) may or may not be activated. The output would be a single float value [7].](image-url)
current layer so as to reflect an accurate prediction of the current state of the board after adjustment.

The final deliverable at the end of the project shall provide a graphical user interface for any human player to play Othello against IagoBot. In terms of target play-strength, IagoBot to be able to beat a relatively good human player at the majority of the games it plays. However, there should also exist an option for a human player to choose the difficulty of IagoBot so both beginners and experts alike may enjoy the game.

Existing Othello Programs

Despite the fact that Othello programs do not need to utilize deep learning neural network principles at all to achieve superhuman levels of play, there still exists a few notable Othello AIs available to the public that are powered by neural networks. For instance, Darwersi [8] is an Othello AI tuned using genetic algorithms and the Evolutionary Neural Network for Othello Game program published in the Procedia journal [9]. However, at this point in time, there does not currently exist any programs that utilizes technologies such as a value network to evaluate the winning likelihood of a board position in a game of Othello. Therefore, there does not exist a comparable program to IagoBot against which any meaningful benchmarks in performance may be established.
II. Project Methodology

Hardware & Software Requirements

Computing power tends to be a major deciding factor in the strength of a board game AI and in the time required for training a neural network. A case in point is the following table of the latest versions of some Go-playing programs along with their performance figures:

<table>
<thead>
<tr>
<th>AI Name</th>
<th>Time restrictions</th>
<th>CPUs</th>
<th>GPUs</th>
<th>Elo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed AlphaGo</td>
<td>5 seconds</td>
<td>1202</td>
<td>176</td>
<td>3140</td>
</tr>
<tr>
<td>AlphaGo</td>
<td>5 seconds</td>
<td>48</td>
<td>8</td>
<td>2890</td>
</tr>
<tr>
<td>CrazyStone</td>
<td>5 seconds</td>
<td>32</td>
<td>-</td>
<td>1929</td>
</tr>
<tr>
<td>Pachi</td>
<td>400,000 sims</td>
<td>16</td>
<td>-</td>
<td>1148</td>
</tr>
<tr>
<td>Fuego</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>431</td>
</tr>
</tbody>
</table>

*Table 1. A table that displays the results of a tournament between various Go programs, where time restrictions is the number of seconds an AI is permitted to think before being forced to make a move. Elo [10] is an arbitrary scale for play-strength that will not be used for this project.*

It is important to note that in table 1, the number of CPUs and GPUs being utilized by each program has a generally positive correlation with play-strength (this is true regardless of whether the game being played is Go or Othello). For the purposes of IagoBot then, the availability of more hardware resources will no doubt also contribute to a better overall performance. For this reason, IagoBot is trained and run on a machine with the following hardware/software specs:

Operating System: Ubuntu 14.04 LTS
Graphics Card: NVIDIA GeForce GTX 960 M
GPU Memory: 8090 MB
GPU Driver: CUDA 5.5
GPU Acceleration Library: cuDNN 5103

For the purposes of development, Jupyter Notebook v. 4.2.1 is used with Python v. 2.7.12.

Libraries:

Scikit-learn: Contains many basic machine-learning tools and models
Keras: A high-level library which offers quick and easy instances of CNN models.
Theano: Used as the backend for Keras and allows for GPU acceleration during training.

Design & Implementation

This project comprises seven distinct phases of construction:
the Hardware/Software Preparation Phase during which the machine for neural network training is configured to run all fitting functions via the GPU,

- the Training Data Collection Phase, during which sample games are created from a mixture of human and AI gameplays and recorded into raw game data files,

- the Neural Net Construction Phase, during which the deep-learning PolicyNet is created,

- the Training Phase, during which the neural network is fed sets of training data to allow for the optimization of their respective weight matrices,

- the Validation Phase, during which the network is tuned via the random selection of the network's various hyperparameters to find the combination of settings which produces the best result for the objective function,

- the Game Logic Construction Phase, during which a tree-search heuristic is built on top of the network to facilitate the assembly of a decision-making algorithm that would be responsible for selecting the best moves to make for the machine, and

- the Graphical User Interface Construction Phase, during which a functional GUI is built on top of the existing game logic that allows players to interact with IagoBot.

**Hardware/Software Preparation Phase**

The problem with only utilizing CPU to run training data sets is that the frequency with which data is required to be copied in and out of the CPU memory usually presents a large computational overhead, and can severely slow down the speed of training. This is where GPU-acceleration provides a huge advantage over traditional CPU-intensive training methods. Most high-end GPUs on the market today have 2 or more gigabytes worth of dedicated memory, which can be used to store values known as "shared variables". These are values that do not incur an input/output overhead because they can be accessed by the GPU natively. For the purpose of this project, a laptop with a Nvidia GeForce GTX 960 M GPU with 8 gigabytes of dedicated memory will be used to train IagoBot. The GPU will utilize the GPU-programming toolchain (CUDA) offered by Nvidia in order to run the many functions offered by Theano [11], a Python library that lets user-defined mathematical expressions and multi-dimensional arrays in particular to be run with GPU acceleration. This allows for a drastically shortened training time as well as evaluation time so IagoBot would be capable of a greater number of calculations within the time limit for each move.

**Training Data Collection Phase**

The collection phase is a joint effort between all 6 students currently working on this project. Each student is responsible for creating 50 sample games. Under the assumption that each game of Othello lasts 50 steps, the total number of game states is 15,000. However, the number of states that can be used as training data (at least in the case of the value network) is much smaller than 15,000 since only those game states that are duplicated between
independent games can have an associated value which corresponds to the likelihood of winning from that state. Currently, all student-created states have been uploaded to the following archive: http://i.cs.hku.hk/~kpchan/Othello/TrainingSet.html. To tackle the problem, my colleague Argens Ng has generously provided me with the Othello World Cup dataset (containing 66,894 number of games) which adds an additional 105,000 independent board states to the existing dataset after initial parsing.

**Recording & Parsing Games – Original Set**

In order to record sample gameplay for the construction of a training set, a third-party application (The Othello [12]) was installed on a mobile device. This application not only provides an existing Othello-playing AI but automatically records player moves that could then be exported. The first 20 sample games for this project were records of games played between a human and the computer on level 15 difficulty. The next 30 samples games were played between two iterations of the application against each other (both at level 30). In order to transform the exported game files into the standardized CSV file, a simple parser [13] was created in Java that helped to automate this process and to ensure no human error was introduced in transcribing the moves to a machine-readable format. The sample game sets can be downloaded from the project website [14].

The reason why human gameplay is mixed with AI gameplay in the sample data is due to the fact that sometimes human players are able to gain an intuitive understanding of where to best place a piece. If the training data consisted of only computer-generated games, then it would be more likely for IagoBot to play nonsensical moves or moves that any human observer would immediately identify as ineffective. However, it also infeasible to construct the entire dataset from human gameplay only, owing to the amount of time such efforts would take as well as the fact that in Othello, human gameplays are seldom as expert as AI gameplays.

The format of each game file is as follows: Each player move takes up one line in the CSV file. The information stored in each line can be represented as x c r, where x is the color of the player (0 representing black and 1 representing white). c and r represent the column number and row number of each move, where both numbers are in the range of [0, 7].

Toward the training stage, these data files will be shared between all 4 groups of students currently taking on the same project. This is so that no group benefits from the advantage of utilizing a better dataset for training. The layout of one such data file is shown below:

---

**fig. 4:** The contents of an Othello game file belonging to the original dataset.
Recording & Parsing Games – World Cup Data Set

One issue with the original game set which was revealed to be problematic only after the entire program has been constructed was that the prediction accuracy of the trained value network was insufficient due to a lack of training data. Additionally, owing to a lack of baseline for comparison from other deep-learning Othello programs, the original set of a maximum of 15,000 states was deemed a sufficient amount. Later in the development process, when it was discovered that the trained model did not produce a strong enough level of game play, the World Cup Data Set was added to the original set. The new dataset is defined in the plaintext document WCD.txt and contains 66,894 independent games. The layout of this file is shown in figure 5.

Reconstructing Sample Games

The game data files could not be passed to the neural network in their raw form as the network would not be able to make sense of each line which corresponds to a players' move. To allow the model to form an intuitive understanding of each move, it must be passed an entire frame of gameplay representing the current state of the board. To this end, a program called the othello_game_reconstructor.ipynb is built to help reconstruct board layouts from existing game data. This program reads in each line/move from all the game files, then reconstructs the layout of each board state using Othello rules and outputs a list of unique board states across all sets of game data analyzed.

An additional program called the othello_game_states_reducer.ipynb reads in the output from the previous module and eliminates any board state that is either not shared between
at least two separate games or not an endgame state. For any shared/duplicated board state, its associated win likelihood value may be calculated as follows:

\[
\text{win likelihood} = \frac{\text{number of wins from current layout}}{\text{number of distinct games containing this layout}}
\]

This is the evaluation function as defined in the Project Objectives & Deliverables section.

For example, if one particular board state is shared between four distinct games where three of these games have been won, then the likelihood of winning from said board state would be \( \frac{3}{4} \). Note for this project, all win/lose qualifiers are calculated with respect to the black player. Hence \( \frac{3}{4} \) refers to the likelihood for the black player to win.

**Neural Net Construction Phase**

**Development Platform**

Thus far into the development process, both Java and Python have been used to implement various modules and programs. However, for the rest of IagoBot and the neural network in particular, Python is selected as the sole development language for the following reasons: Firstly, it is a dynamically typed language which allows for rapid-prototyping development techniques. Secondly, it provides rich and diverse sets of library functions and APIs developed by the open-source community, making it possible to replace the need for rudimentary or generic functions with library imports, which in turn cuts down the overall development time. Such libraries include but are not limited to: numpy for easy matrix manipulation, matplotlib for quick visualization of the output of neural networks, pandas for data analysis and scikit-learn, which is the most widely-used machine learning library. Moreover, Python is extremely popular as a machine learning language and there exists an active online community specializing in Python machine learning which should prove to be a valuable source for help and support should the need ever arise.

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(C=1000.0, random_state=0)
lr.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std, y_combined,
classifier=lr, test_idx=test_idx,
xlabel='petal length [standardized]',
ylabel='petal width [standardized]')
```

**fig. 7:** An illustration of the various library functions available to Python. Here, a logistic regression model is imported and trained and new test data classified in just a few lines of code using scikit-learn and matplotlib.
**ValueNet Introduction**

The Value Network is the weight matrix model upon which the game logic is built. The network itself is based on a convolutional neural network because CNNs generally perform very accurate image classifications, and are thus suitable for analyzing board states which are simply static images. CNNs are also adept at facial recognition, working with complex imagery and allows for the extrapolation of low level patterns into high level features [15]. This lets the ValueNet develop an intuitive understanding of the game. Eventually, given enough training data (and under the assumption that all training data are drawn from expert gameplay), it would be capable of accurately predicting the likelihood of winning from the current board position.

The architecture of ValueNet is shown in Appendix figure C. As clearly illustrated by the figure, the network consists of an input dimension of 1 x 3 x 10 x 10, and an initial convolutional layer of shape 32 x 10 x 10, where 32 represents the batch size and the 10 x 10 plane represents the 8 x 8 Othello board padded with an extra layer of empty cells with input features. These input features are illustrated in Appendix figure B and represent the placement of empty cells, black cells and white cells on the board.

The value network has a {CONV, CONV, Max-Pool} architecture. It is also a regression network like the PolicyNet. However, the ValueNet differs in having only 3 input features and an output consisting of one scalar value. This value represents the percentage likelihood of winning for the current player given a board layout.

**ValueNet Development Process**

The value network has been constructed in parallel along with the other parts of the program since the exact architecture of the network is difficult to finalize and takes much trial and error as well as analysis to get right.

Again as depicted by figure C in the appendix and the following ValueNet output summary (figure 8), the network consists of one max pool layer and two densely connected networks in the end with a softplus activation function. The exact construction is as follows:

![ValueNet Output](image)

**fig. 8: IagoBot_ValueNet.ipynb output**
This architecture is standard for a convolutional neural network, since a series of convolutional layers and max pool layers serve to extrapolate the input data into higher level features with unique activation outputs. The max pool stage also serves to reduce the size of the board from 10 x 10 to 5 x 5 as eventually the deep learning model would be rescaled to only produce a single value as output. Furthermore, adding fully-connected layers at the end of a CNN serves as an efficient way of learning non-linear combinations of these high-level features. Finally, the output from the output node is a float value in the range of [0, 1], where 0 stands for the case where white would certainly win and 1 stands for the case where black would certainly win.

**Training Phase**

In the third phase of this project, IagoBot was trained by means of supervised learning. To be sure, there are other methods of training, for instance reinforcement learning which is popularly used to train AI programs to play classic Atari games such as Mario or Breakout. However, it should be noted that a reinforcement learning approach is not necessary suitable for IagoBot because it is overly complex and unnecessary for a relatively simple game (simple meaning computationally inexpensive) like Othello. Furthermore, AlphaGo was trained using reinforcement learning only after exhausting 30 million sets of training data, whereas the amount of Othello training data available for this project started out at 555 and later increased to 13,966. Hence, instead of resorting to reinforcement learning to increase prediction accuracy, a better method would be to simply create more training data.

Supervised learning is a data-driven learning method. In the case of ValueNet, a supervised learning approach would stipulate feeding the program sets of training data which consist of sample games of Othello as well as an analysis (target outcomes) of every player’s strength/likelihood to win within each of those games. Following the construction of othello_game_states_reducer.ipyn in Training Data Collection Phase, the sample games and their respective winning-likelihood are the two output files of this module, namely: value_net_x_input.pickle and value_net_y_input.pickle. IagoBot then takes the y input values as the target labels, compare them with its actual output at the end of each iteration after reading in the x input values, and adjust its weight matrix accordingly.

**Validation Phase**

Training of a neural network is almost always carried out in tandem with a cross validation of the network’s hyperparameters. This is a necessary and crucial step to developing an accurate prediction model owing to the number of parameters available for adjustment in any given model (see figure 8’s param # for each layer). This process of modification is known as hyperparameter tuning, and serves to optimize the meta-properties of a neural
network (characteristics such as optimization functions, number of epochs, activation functions etc.), whereas the training stage only optimizes the internal weight matrix of the network. The decision to choose certain hyperparameters over others is often not an exact science and is influenced by experience and circumstance when performed manually. Therefore, it is necessary for this project that hyperparameter optimization be carried out in a systematic manner. At the same time, it is important to note that validation data must be separate from the testing data to prevent the network from recycling used datasets to produce a biased performance result. The following algorithms were considered as candidates for performing validation checking:

**KFoldCV**

One of the most straightforward cross validation strategies which involves manually picking hyperparameters. All automated cross validation techniques introduced later are based on the ideas from KFoldCV. Note cross validation in this context refers to the statistical sampling of training data to obtain a random distribution of data points to be the validation data, since insufficient shuffling of the training set may result in inconsistent validation outputs [16].

The KFoldCV method partitions the training set into K mutually exclusive sets or folds. Then over K iterations, each fold within the partition data assumes the role of validation data. A weighted average of the performance measure across all the iterations are then found and returned as the validation result. To prevent the model from being biased on all the training folds (known as cumulative learning), K separate instances of the model should be used.

**GridSearchCV**

This automated tuning process performs an exhaustive search over all the possible combinations of hyperparameters available to a particular model to see which combination of settings would produce the optimal performance measure. Like the KFoldCV technique described above, GridSearchCV allows the user to specify the number of folds with which to divide the data into training and validation sets [18].
However, GridSearchCV is flawed in that in a practical situation, it would suffer from the curse of dimensionality, which states that any increase in dimensions within a Euclidean space would cause an exponential increase in the complexity of a model [19]. For instance, if a model has 10 parameters which require tuning (a conservative estimate), each of which has 10 different settings (again a conservative estimate), then GridSearchCV would already need to perform $10^{10}$ (or 10 billion) fits on the training data, which is infeasible with current levels of computational power. This is especially true for a convolutional neural network which can take many hyperparameters such as the number of filters, number of layers, performance metrics, drop-out rate, momentum, learning-rate, decay, activation function etc.

**RandomizedSearchCV**

An easy mitigation strategy to this problem is to use a randomized grid search cross validation algorithm, which is described as follows:

Given a set of parameters $p_i$ each with $N_i$ different values, search through all combinations $\prod_i N_i$ with random sampling.

This approach generally works well for machine learning models with a large number of hyperparameters, and is typically able to find the optimal (or near-optimal) settings for the model in no more than 100 iterations. RandomizedSearchCV is also a library function provided by scikit-learn which makes the automated validation process easy to implement [20].

For the purposes of IagoBot, the RandomizedSearchCV approach was used. During optimization, a total of 11,172 unique boards states were passed into this function as training/validation data. 10 cross-validations were performed for each iteration, meaning that around 1,117 states (10% of the original X_train set) were used to validate the performance of the tuned model.

*fig. 10: Hyperparameter tuning process #1.*

Since the number of cross-validations are set to be 10 and the number of iterations to be performed set to be 200, the total number of fits was $10 \times 200 = 2000$. 

```python
Using gpu device 0: GeForce GTX 980M (CUDA is enabled with initial
X.shape (13966, 1, 3, 10, 10)
X.shape (13966,
X_train.shape (11172, 1, 3, 10, 10)
y_train.shape (11172,)
X_test.shape (2794, 1, 3, 10, 10)
y_test.shape (2794),
Grid fit in process..
Fitting 10 folds for each of 280 candidates, totalling 2800 fits.
[CIF] kernel_initializer=random_uniform, init-mode=he_normal, output
```
Game Logic Construction Phase

In this the sixth phase of the development process, a program was required to make game decisions (i.e. whether to capture a particular square) based on the likelihoods of winning received from the ValueNet. To this end, IagoBot utilized a form of tree-search algorithm to help it decide which move to make at each turn. Ideally, the tree-search algorithm will be able to combine both the value network and an additional policy network (responsible for determining the probability of a move occurring under expert play) in a single heuristic that selects which move to make via a lookahead search, essentially acting as the primary decision-making agent within IagoBot. There are two possible algorithms that could be used to achieve this goal: a minimax algorithm with alpha-beta pruning (currently used in IagoBot) and a Monte-Carlo Tree Search algorithm with a fast rollout policy.

**Minimax Algorithm with Alpha-Beta Pruning**

The minimax algorithm has the following depth-first recursive structure:

\[
\text{minimax}(s) = \begin{cases} 
\text{utility}(s) & \text{if terminal}(s) \\
\max_{a \in \text{action}(s)} \text{minimax}(\text{result}(s, a)) & \text{if player}(s) = \text{MAX} \\
\min_{a \in \text{action}(s)} \text{minimax}(\text{result}(s, a)) & \text{if player}(s) = \text{MIN}
\end{cases}
\]

Minimax is widely used as the decision-making logic for simple games such as tic-tac-toe or connect-four. However, it is also possible to apply minimax to games like Othello. With IagoBot, the minimax algorithm used the ValueNet to predict the likelihood of winning for every given board position, which is the utility function called at the leaf nodes of the game tree. The algorithm used these values to label every node in the game tree. When a board
state is passed in, all possible combinations of play leading from that state are found and the utility value calculated for each of these combinations if it is a leaf node. However, it would be prudent to reduce the memory load of IagoBot by setting a maximum search depth to the minimax algorithm to prevent the entire game tree from being searched for every play. This was done via a global DEPTH variable defined in the othello_main.py file and passed into the minimax algorithm as an argument (See table 3 for details).

The load on the program’s memory was further reduced by augmenting minimax with alpha-beta pruning to rule out obvious cases which would lead to a player to lose. This is done by assigning each node in the game tree with an \( \alpha \) and a \( \beta \) value, where \( \alpha \) represents the best already-explored value along the path to the root for the MAX player and \( \beta \) represents the best already-explored value along the path to the root for the MIN player. In this project, black is arbitrarily defined as the MAX player and white the MIN player. \( \alpha \) is assigned a value of \(-\infty\) and \( \beta \) a value of \(+\infty\) during the initialization step. Whenever the recursive minimax() function returns a value to the current node, this value is compared with the \( \alpha \) and a \( \beta \) member variables of that node. \( \alpha \) is updated to the value if the value is greater than the current value of \( \alpha \) but smaller than \( \beta \), and \( \beta \) is updated to the value if the value is greater than the current value of \( \beta \) but smaller than \( \alpha \). The memory-saving property of alpha-beta pruning states that the subtree under consideration can be pruned whenever the value returned from minimax() is higher than \( \beta \) if the current node is a MAX node or if the value is lower than \( \alpha \) if the current node is a MIN node. The following figure illustrates this property, where the white nodes are nodes that have been pruned from the game tree.

![fig. 12: Example of a minimax algorithm with alpha-beta pruning. The alpha and beta values for every node are shown, as well as the conditions for which pruning occurs (highlighted in yellow). [21]](image)
Monte-Carlo Tree Search with Fast Rollout \(\Leftarrow\) move somewhere else?

An alternative algorithm based on the policy and value networks is the Monte-Carlo Tree Search algorithm. This is the method utilized by AlphaGo to estimate the value of each state in the game tree. A fast rollout policy is also used in AlphaGo in order to execute simulations of gameplay from leaf nodes in each partial game tree. For this project, a Monte-Carlo Tree Search approach is not necessary, but could be implemented as an extension should the minimax method not provide a satisfactory play-strength.

Graphical User Interface Construction Phase

This is the final phase of the development process and the final layer which is built on top of the game logic layer. Since the ultimate goal of this project is for end users to be able to play real-time Othello against the computer, a GUI client program was deemed necessary. The client program lets the users pick either white or black as their color (which determines whether they have the first go) and users make a move by selecting a square on the board. This will initiate a set of minimax algorithms to be called where the computer simulates the odds of winning from positions that immediately follow the user's move, and selects a placement that maximizes the value of minimax() if the computer is playing as white, or minimizes the value if the computer is playing as black. Points are also kept track in a message box and the player with the most points at the end of the game wins. A detailed overview of all the features of the GUI are given in the next section.

As illustrated by figure 13, whenever othello_GUI.py is executed, a window representing the game board would appear. Players have the option to choose their game piece color as well as the difficulty of the AI opponent via the menu bar at the top. A game would begin when a selects Game \(\rightarrow\) New Game, after which the board would be automatically set up as shown. In the instance where the player is black, the player would have the right of play. A player adds a piece by selecting any square on the board. As the cursor hovers over a square, that square would light up as shown. If the player decides to move to an illegal square, no action

\begin{center}
\textbf{fig. 13: The client-end graphical user interface for IagoBot.}
\end{center}
would take place; instead, a warning message would display in the status bar at the bottom of the window.

To allow for the adjustment of difficulty, two possible approaches were taken into consideration: 1. Allow for minimax_ab.py to load different pre-trained ValueNet models via different .hdf5 files. 2. Let different DEPTH global variables as defined in othello_main.py determine the strength of the AI. Upon careful evaluation, option 2 was deemed easier and faster to implement as option 1 meant training multiple instances of ValueNet with varying numbers of input data which would take longer than was permitted by the timeframe of this project.

**Detailed System Architecture**

**Object Diagram**

The object diagram in Appendix figure E shows all the files and modules created for this project and their association with each other as well as the flow of execution from one module to another. Please reference this object diagram for a comprehensive overview of all the components that are part of this project.

The following list summarizes the inputs/outputs as well as functionalities and flow of all the files/modules shown in the object diagram (organized according to category).

**Files**

**raw.txt**
The raw.txt file is the raw game state file which represents all the moves played by either player in an entire game of Othello. Each line in the file represents a move and the three space-separated numbers in each line represents the player color, x-coordinate (column number) and y-coordinate (row number) respectively. Following is a snippet of the contents from one such file:

In this file, black player (the first to play) is represented by 0 and white player is represented by 1.

The first move by black places a piece at x=5, y=4.

The file format is proposed by Prof. K.P. Chan and followed by all groups who are working with the Othello project.

**othello_game_states.pickle**
This is the output file from othello_game_reconstructor.ipynb, and contains a dictionary of key value pairs where the key is a string representation of a board layout that is unique to that board layout, and the value is a BoardState object that is defined as follows:

![fig. 14: The contents of an Othello game file belonging to the original dataset.](image)
The BoardState class stores a board state as an 8x8 array as well as the meta attribute win_outcome_list which is a list of win/lose outcomes derived from the current state. For instance, if a board layout produced two wins for black and one win for white, then the win_outcome_list would contain [0,0,1]. The win_outcome_list is updated by othello_game_reconstructor.ipynb and is useful in determining whether a state is duplicated across more than one game.

**value_net_x_input.pickle**
This is the X training data outputted from the othello_games_reducer.ipynb module and has the shape of (N, 1, 3, 10, 10), where N is the number of training data, 1 is an empty dimension, 3 is the number of feature planes (illustrated in Appendix figure b.) and the 10 + 10 left over dimensions describe the 10x10 shape of the padded input grid. More information regarding padding is explained in Risk #2 of the Risks, Challenges and Mitigation section.

```
In [15]: np.array(value_net_x_input_list).shape
Out[15]: (13966, 1, 3, 10, 10)
```

**value_net_y_input.pickle**
This is the Y target labels outputted from the othello_games_reducer.ipynb module and has the shape of (N,), where N is the number of training data/target labels.

```
In [16]: np.array(value_net_y_input_list).shape
Out[16]: (13966,)
```

**value_net_model.hdf5**
This file stores the trained ValueNet model and associated weight matrices from IagoBot_ValueNet.py. This file is read in by minimax_ab.py so the model’s predict() function can be called on an ad hoc basis whenever a leaf node value (win-likelihood value) is needed. The alternative to writing the model to disk would be to train IagoBot_ValueNet.py from scratch and keep the trained network in main memory when the game program starts, which would be inefficient and inconvenient. On the other hand time and resources can be saved by training the network only once and saving the trained model so all parameters and weight values are loaded to a fresh instance of the ValueNet very quickly. In fact, this file is read and
loaded whenever the user opens the IagoBot GUI window to hide the network’s start-up time, which takes around 5 seconds even without the training step.

**Debug Files**

*othello_game_states_reduced.pickle*

Whereas othello_game_states.pickle contains a dictionary of unique board layouts, othello_game_states_reduced.pickle contains a dictionary of board layouts that are also either duplicated across more than one game or is an end game state. This means that othello_game_states_reduced.pickle contains only the states for which a definite win-likelihood is found. This file is outputted by othello_game_states_reducer.ipynb and can be read by othello_game_states_reader.ipynb. The purpose of this file is to ensure that othello_game_states_reducer.ipynb operates as intended and to visualize the reduced game set which is intended to become the training input for IagoBot_ValueNet.py.

**Modules**

*othello_rank_flipper.ipynb*

This module was constructed to reformat the game files created by Anna & Josephine’s group which reversed the Othello row indices. The module reads in a batch of files in a given directory and overwrites them so that all rows begin at #0 at the top rank instead of #7 at the top rank.

*WCD_letters_to_num_converter.ipynb*

This module takes in the original WCD.txt file and outputs another .txt file after having mapped all coordinate letters to numbers. For example, the coordinate A4 become 03 and the coordinate H1 become 70 after mapping.

*WCD_data_to_txt_converter.ipynb*

The output file of WCD_letters_to_num_converter.ipynb is the input to this module, which uses the successors() and place_piece() modules from othello_utility.py, allowing it to reconstruct the game while reading through the input file according to Othello rules. This is necessary in determining the color of each play (which does not simply alternate as users are allowed to pass if no legal moves can be made), because this detail is missing from the raw world cup dataset but is a necessary component of the input files to the game reconstructor module. WCD_data_to_txt_converter.ipynb outputs each individual game as a separate .txt file in a directory of the user’s choosing.

*othello_game_reconstructor.ipynb*

As shown by the object diagram above, this module reads in raw.txt and outputs othello_game_states.pickle. For every line of every raw file read in, the module reconstructs an entire game of Othello, frame by frame. Each individual frame is stored as a BoardState object (as in figure 15), and every game is stored as an OthelloGameStates object thus defined:
One instance of OthelloGameStates called curr_game_states is maintained throughout an entire iteration (defined as a pass over a raw file) of the module and contains a list of BoardState objects that comprise a game of Othello. At the end of the iteration, this list is transferred to othello_game_states.pickle as a dictionary and curr_game_states is set to an empty list and the process continues.

Owing to the vast numbers of raw files that require processing, this module analyzes all the raw files in batch. The raw files are pasted into the ./Datasets/Data directory and a for loop ensures all files are passed into the module for processing.

**othello_game_states_reducer.ipynb**

This program reads in othello_game_states.pickle and outputs three files: othello_game_states_reduced.pickle, value_net_x_input.pickle and value_net_y_input.pickle. In order to determine whether a BoardState object in the input dictionary should be kept, the module first checks for the number of times the state in question has occurred across different games. This is done by verifying that the length of the state's win_outcome_list is greater than 1. If not, the reducer checks whether the state is an endgame state by checking for the list of successor states as a black player and as a white player. The list of successor states is returned from the function successors() in the othello_utility.py module. If either the state has duplicates across different games or if it has no successors, then it is marked as a valid state by being copied to an output dictionary called new_game_states_dict. The value_net_x_input.pickle output file is produced by taking every 8x8 board layout from new_game_states_dict and passing it as an argument to othello_utility.py's divide_state() function, which slices the 8x8 array into a shape of (1, 3, 10, 10), where 3 refers to the 3 feature planes of the X input and (10, 10) refers to the 10x10 padded board grid.

**IagoBot_ValueNet.py**

This is the main Python module of this project and contains a CNN baseline model trained via RandomSearchCV. The module takes in value_net_x_input.pickle as the input array and value_net_y_input.pickle as the target labels (or target values since this project deals with a regression problem). IagoBot_ValueNet.py performs two mutually exclusive tasks: training/fitting + hyperparameter tuning and value prediction. These tasks are performed in
order with training and hyperparameter optimization being bundled together as an integrated process. These two tasks must also be performed before prediction can take place. As an output from training, an instance of the model along with its trained weight values are saved within the file value_net_model.hdf5. This way, when an instance of the baseline model is invoked by minimax_ab.py, the trained parameters and weights can be instantly loaded, allowing the predict() function to be called. Owing to these reasons, the user must decide when to comment out certain parts of the IagoBot_ValueNet.py module to ensure fit() and predict() do not coexist at the same time.

**minimax_ab.py**

minimax_ab is the module containing game logic and the minimax with alpha-beta pruning algorithm. This program reads in value_net_model.hdf5 as an instance of the ValueNet and passes it in as an argument to the predict() method defined in othello_utility.py. This method then returns the likelihood of winning (for black) of a given board state. The predict() method is called as the base case for the recursive minimax algorithm. Note that prior to passing in the 3x10x10 argument to predict(), this module calls divide_state() as defined in othello_utility.py to reshape the 8x8 board layout into the (1, 3, 10, 10) shape accepted by the value network. minimax_ab.py also takes in the following arguments: color, state, depth, alpha and beta. Where color is the color of the current player (black is MAX, white is MIN), state is the current board state, depth is the maximum depth of search before the algorithm reaches the base case (currently set to 3 for Normal difficulty) and alpha and beta are the initialized to $-\infty$ and $+\infty$ respectively for a node within the game tree. A float value variable is returned from minimax_ab.py which corresponds to the likelihood of winning for black under the current state. If color = black, then this value would be maximized for the calling function, and the value minimized if color = white.

**othello_main.py**

This module is built on top of minimax_ab.py and under othello_GUI.py. It contains the global variables DEPTH, ALPHA and BETA used in minimax_ab.py. This module contains the function recommend_move() which is invoked by othello_GUI.py whenever it is the computer’s turn to make a move. recommend_move() takes in two arguments: the color assigned to the computer and the current 8x8 board layout. Within the function, a for loop iterates over all the possible successor states to the current state (as well as the likelihood of winning from each state) and returns a ((y, x) tuple, resultant state) tuple, where (y, x) is the (column #, row #) tuple that leads to the resultant state (an 8x8 grid).

**othello_GUI.py**

This is the graphical user interface presented to the end user whenever they play the game. It is also the only program which needs to be executed for IagoBot to work as long as the ValueNet has already been trained. This program utilizes the tkinter library to output an interactive Othello board to the player. The program interface is shown below:
Debug Modules

**othello_game_states_reader.ipynb**

This module is capable of reading either the othello_game_states.pickle output file or the othello_game_states_reduced.pickle output file (i.e. the outputs of the game state reconstructor and the game states reducer modules). This debug module is capable of printing out the dictionary values from either input files, calculating the ratio of duplicate states to total states, as well as recreating visual representations of board states for every state analyzed using the pandas library. Both examples are illustrated below:

![Fig. 19: IagoBot’s graphical user interface start-up interface.](image)

![Fig. 20: IagoBot’s Color, Difficulty and Game submenus](image)

![Fig. 21: game_states_reader.ipynb output #1](image)

![Fig. 22: game_states_reader.ipynb output #2](image)
**othello_value_net_input_reader.ipynb**

This module is used to read either value_net_x_input.pickle or value_net_y_input.pickle (outputs of othello_game_states_reducer.ipynb) to ensure that the reduced x and y input lists are of the correct format. Since many types of errors in the ValueNet’s input may be difficult to debug from the ValueNet module itself, it is important to place a quality check before training data are passed in so that potential defects (such as incorrect board rules, duplicate layouts and out of bounds target labels) are easily identified.

**debug_timer.py**

This is a simple module designed to measure the time taken by function calls. Within this project, the debug_timer.py was used to time the minimax() function in order to determine the highest DEPTH value so that minimax() returned the most accurate win/lose prediction within a reasonable time limit.

**Utility Modules**

**othello_utility.py**

This is arguably the most integrated Python module within the project, providing the following helper functions so as to reduce code reuse. Detailed descriptions of each function are included as comments shown below:

```python
# Takes a single othello state (8x8 grid) and returns 3 feature planes with padding:
# A 8x8 empty feature plane
# A 8x8 black feature plane
# A 8x8 white feature plane
# Return grid is of depth 3 and padded by 1 in the +X and +y directions
def divide_state(instate):

# Game logic for placing a piece on a NON-Occupied cell
# Arguments: board state, row index, column index
# Returns board state after piece placed according to Othello rules
# Returns None if placement is not possible
def place_piece(board_state, color, x, y):

# Returns a list of possible states and actions
# which result in these states.
# @param state: 8x8 grid, color: curr player's color.
# @param ret_dict AFTER piece is placed.
# @param ret_dict --> (xy, resultant_board_state)
def successors(state, color):

# Passes state through ValueNet and returns win/lose likelihood.
def predict(model, state):
```

**fig. 23**: othello_utility.py’s public functions

Please reference Appendix figure E for a list of modules dependent on othello_utility.py functions.
Risks, Challenges & Mitigation

Risk #1: Overfitting During Regression

Regression is a category of machine learning that is used to model the two neural networks used in this project. It simplifies down to a problem of curve-fitting. We can assume that each point in figure 24 is a data point from the set of training data in the 64th dimensional feature space, where each feature is a cell on the 8 x 8 Othello board represented by a neuron in the fully connected layer of the neural network. Here, only 2 dimensions are shown for ease of illustration where x₁ and x₂ correspond to two cells on the Othello board. The aim of IagoBot is to adjust its weight matrix so that a curve is produced that models the trend of the data as accurately as possible without overfitting (third image in figure 24). The problem with overfitting is that sample data often contains noise, and an overfitted curve rarely reflects what the actual model would look like. Furthermore, an overfitted curve would introduce unnecessary complexity and slow down the program. On the other hand, we would also like to avoid underfitting by producing an overly simplistic curve.

The solution to this problem is to attach a penalty function or regularization function P(θ) to the loss function which we are trying to minimize. Consider the following objective function Φ of the neural network:

\[
Φ(X, T, θ) = L(X, T, Y = f(X, θ)) + P(θ)
\]

P(θ) is an independent variable which can be adjusted to favor a simple curve over a complex one by evaluating to a higher value for higher order curves, where θ represents the weight value parameters.

In regression, many various regularization methods can be used, for instance ridge regression, which penalizes the excess of parameters based on the Euclidean distance measure. Alternatively, the Least Absolute Shrinkage & Selection Operator (LASSO)
regularization method can be used which is based on the Manhattan distance measure. The Elastic Net is a third possible approach which combines the methods above.

In terms of practical applications, this penalty variable is included in the Keras library in the form of regularizer functions.

**Risk #2: Value Network Unable to Detect Board Edges**

Owing to the nature of convolution, the input 8 x 8 grid would be completely garbled by the various filters to pass through the input layer without any modifications to the inputs. This creates a problem illustrated by the following figure where the ValueNet is unable to determine the difference between the blue and green regions of the board.

![Fig. 25: Before and after input layer padding.](image)

Prior to padding, the CNN would see no distinction between the blue and green regions.

After padding, the CNN would treat the blue region as a corner region, the green region as an interior region and the yellow region as an edge region.

The solution to this problem is simple: Pad every input feature by 1 cell on each side. This means that visually, the CNN can distinguish corner or edge regions from the interior regions. This alone should be sufficient for the value network to have an intuitive understanding of the importance of corner-capture.

**Risk #3: Insufficient Training Data**

There are 6 students who are currently working on this project. During the planning stage, it was agreed that each student produce 50 independent games of Othello, resulting in an estimated maximum total of 15,000 board states. However, upon running these states through the game reducer module, the number of usable game states have been shown to be only 555. Assuming 20% of this data is reserved for testing, then only 444 instances of game states may be used for training. This is clearly not enough training data, even for a simple game like Othello for which there are $10^{28}$ possible board states [22].

This was a risk since the beginning of the project, but early in the development process, it was still too soon to decide whether the lack of training data was going to have a considerable impact on the performance measure of the neural network, and at that time, the development schedule (see Project Schedule under section IV. Conclusion & Future Works)
called for the construction of the ValueNet first. It was only later into the development process that it turned out that the mean squared error measurement produced as well as real-time testing against the Othello app proved that the trained network did not produce an accurate enough result.

As a result, additional training data had to be procured, which was supplied by Argens Ng. The new data (which was drawn from the Othello World Cup dataset) contains 105,000 independent states.

**Challenge #1: Value Network is Difficult to Debug**

Unlike other Python programs, the Value Network is slow to train and outputs only a single prediction value. This makes the network difficult to debug, especially seeing how it is based on other modules, including othello_game_reconstructor.ipynb and othello_game_states_reducer.ipynb which create its input data. It is easy for defects to escape the Training Data Collection Phase and into the Neural Net Construction Phase, particularly if the defects are not obvious (such as incorrect win-likelihood estimates or incorrect board layouts that do not raise any formatting errors). Therefore, during development, phase gates were implemented for each stage of development to ensure that quality checks passed before the next stage of construction could commence. These phase gates are in the form of debug files and modules and have proven to be very useful at not only defect detection but also visualizing the inputs and outputs of various modules which aid in the development process.

**Challenge #2: Inter-Modular Communication**

One of the central problems that needed to be solved before development can begin is the question of how modules can communicate with one another. The following approaches have been considered:

**Pass data as arguments and return values:** This is the simplest method to implement, involving the caller module invoking a function within the callee module which takes in a single value argument and returns a single value result. However, there are two issues with this implementation: First, any transactions between two modules would be stored within their respective namespaces. This means that all data (including items such as game states, reduced game states and x/y value net inputs) would be stored in main memory. While this approach is faster than writing to the disk, it prevents the modularization of the process pipeline and is subject to a higher risk of failure. Consider a situation where many hundreds of thousands of game data need to be processed. To successfully interpret and reconstruct the data and then parse it into a ValueNet-readable form, the execution platform has to stay active for the duration of the entire procedure, which may take anywhere from hours to days or even fail completely. Not only would any potential cause for failure be difficult to detect owing to a lack of physical data, but the entire pipeline would have to be executed again from the beginning. Secondly, the execution platform may not have enough RAM to process the function call if an entire set of game data were stored in a member variable (stored in the stack), and since it is difficult to estimate how much memory would be used for such a pipeline, data should not be passed between modules in this manner.
Use RPyC calls: RPyC is a remote procedure call library that allows inter-process communication across different execution platforms [23]. This is a powerful tool as it uses object-proxying to create objects that can be treated as local instance by each process. This inter-module communication method was considered during the planning stage so a distributed computing system would be able to handle multiple data-parsing tasks to reduce the workload on each machine. Ultimately however, this idea was discarded as it amount of effort required to construct and maintain a distrusted system was deemed too high for the amount of data that was available at the time (300 independent games), and only one extra workstations was available as a resource for this project.

Read/Write to .pickle files: This method solves the memory problem as well as lack of modularization issue inherent to the first solution, and is also very easy to implement [24]. The pickle module is a Python serialization and deserialization package that comes with most versions of Python. Pickling allows the marshalling of Python objects into a bytestream and then saved to a file. The advantages of this approach is obvious: It allows intermediate data to be saved to disk (thus allowing the pipeline to be broken down into stages for better management and defect detection), and it prevents large member variables to be created, potentially causing a stack overflow. Furthermore, pickling is a fast process and if so desired, can be speeded up even further by a factor of 1000 if cPickling is used where the serialization/deserialization process is performed in C. For these reasons, both the Othello game reconstructor and the Othello game states reducer modules read/write in .pickle files.

Challenge #3: Transition from Jupyter Notebook to PyCharm
During the initial development of this project, all code had been written in .ipynb files. The decision to base development activities on top of Jupyter Notebook was partly due to the widely available machine-learning and support libraries such as scikit-learn, matplotlib and pandas which outputs graphics and illustrations straight to the notebook. Jupyter Notebook additionally allows sections of code to be run independently so it is easy to visualize outputs from different parts of a module for the purpose of debugging. Furthermore, Jupyter Notebook is easily installed using Anaconda, which also includes the scipy and scikit-learn packages, making it ideal for quick prototyping work in machine learning.

The problem with Jupyter however, as is revealed during the Neural Network Construction Phase of the project, is that it does not support read/write operations of trained NeuralNet models into .hdf5 file formats. According to the official Keras documentation, .hdf5 is preferred to .pickle when it comes to saving neural networks instances and their associated weight values [25]. By simply invoking model.save(filepath), the entire Keras model will be saved to a .hdf5 file containing data such as the model's architecture, weights, training configuration data (i.e. loss and optimization functions) as well as the current state of the optimizer, allowing for the breakdown of training into sessions. Loading the model is equally straightforward with the load_model(filepath) function offered by the keras.model library.
Running the Program

Following is a list of detailed instructions regarding IagoBot’s initialization/training steps to get it up and running as well as how to operate the Othello Graphical User Interface. Note: please check beforehand whether the following files exist in the current directory (main directory where all .py and .ipynb files are stored), and if so, skip to the appropriate step as instructed by the following table:

<table>
<thead>
<tr>
<th>Files in current directory:</th>
<th>Proceed from step:</th>
<th>1.</th>
<th>2.</th>
<th>3.</th>
<th>4.</th>
</tr>
</thead>
<tbody>
<tr>
<td>othello_game_states.pickle</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>othello_game_states_reduced.pickle</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>value_net_x_input.pickle</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>value_net_y_input.pickle</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>value_net_model.hdf5</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

*Table 2. Starting point in running IagoBot depending on the availability of certain files in the current directory.*

0. Pre-requisites

As a pre-requisite, please ensure that the following packages/libraries are installed on the execution platform:

- Jupyter Notebook v. 4.2.1 or above (needed for steps 1 – 2)
- Python 3.4 or above (needed for all steps)
- Scikit-learn latest version (needed for step 3)
- Keras latest version (needed for step 3)
- Theano latest version (needed for step 3)
- SciPy latest version (needed for steps 1-3)
- CUDA 5.5 (Optional for step 3)
- cuDNN 5103 (Optional for step 3)

1. Training Data Parsing

Firstly, make sure all original datasets are in the form of .txt files and according to the format described under the heading raw.txt of the subsection Files in the Detailed System Architecture section. Ensure that each file represents only one game and are stored in the directory ./Datasets/Data. Then, open othello_game_reconstructor.ipynb with Jupyter Notebook and execute. Make sure that the following lines in the file are the same as in figures 26 and 27. Once this check is made, execute the program.

```
In [6]:
path = "Datasets/Data/"
dirs = os.listdir(path)

  game_file_count = 0;
  for file in dirs: # For all raw data files in the specified directory
```

*fig. 26: othello_game_reconstructor screenshot #1*
Note that some files (such as the ones provided by Anna & Josephine’s group) list row coordinates in the opposite direction: Instead of starting with row #0 at the top and row #7 at the bottom, their convention is to start with row #7 at the top and row #0 at the bottom. To adjust for this difference in format, run othello_rank_flipper.ipynb and make sure the location of the directory containing the differently formatted files are specified in the following location:

```python
In [8]: import os
path_a = "Datasets/Anna-Flipped/
path_j = "Datasets/Josephine-Flipped/"
path = path_j  # Set path here.
dirs = os.listdir(path)
```

Once this is confirmed, run othello_rank_flipper.ipynb, and the original files would be overwritten with their correctly-formatted versions. Move these files to ./Datasets/Data and execute othello_game_reconstructor.ipynb in the same way as described before.

To parse the Othello World Cup dataset in WCD.txt, first make sure that the WCD.txt is correctly converted to the right format by running WCD_letters_to_num_converter.ipynb and WCD_data_to_txt_converter.ipynb respectively. Make sure that the input and output files are specified correctly in the following locations:

```python
In [21]: import os
import sys
with open("Datasets/World_Cup/WCD.txt", 'r') as raw_input:
    with open("Datasets/World_Cup/WCD2.txt", 'w') as raw_output:
        count = 0  # Counter for number of txt files outputted
        for line in raw_input:  # For every game/file
            out_line = 
            curr_color = 1 - curr_color
            file_name = 'Datasets/WCD_Data/wcd' + str(num) + '.txt'
            with open(file_name, 'w') as raw_output:
                num += 1
                out_str = line.join(out_line)
                raw_output.write(out_str)
```

For WCD_data_to_txt_converter.ipynb, make sure that the output directory is ./Datasets/WCD_Data as shown above. This creates a set of .txt files with automatically
generated file names in the WCD_Data folder. After this is done, run othello_game_reconstructor.ipynb again but modify the path as Datasets/WCD_Data/so that this time, the world cup data is also added to the othello_game_states.pickle output.

```
In [6]:
path = "Datasets/WCD_Data/"
dirs = os.listdir(path)

game_file_count = 0;
for file in dirs:
    # For all raw data files in the specified directory

    count = curr_game_states.get_count()

    with open(os.path.join("Datasets/WCD_Data/", file), "r") as raw_input
        print("OPENING ",file)
        first_run_flag = True
        black_player = 0  # Init black_player var, which indicates wheth
```

**fig. 32: othello_game_reconstructor screenshot #1**

``` python
count = curr_game_states.get_count()

# with open(os.path.join("Datasets/WCD_Data/", file), 'r') as raw_input
    print("OPENING ",file)
    first_run_flag = True
    black_player = 0  # Init black_player var, which indicates wheth
```

**fig. 33: othello_game_reconstructor screenshot #2**

2. Training Data Reduction

With the othello_game_states.pickle file in the current directory, simply run othello_game_states_reducer.ipynb with Jupyter Notebook to produce the output files othello_game_states_reduced.pickle, value_net_x_input.pickle and value_net_y_input.pickle.

3. Neural Network Training

The ValueNet training step is an involved process. First, the model would have to be trained/cross validated using RandomSearchCV. During this process, make sure the following two lines are uncommented to ensure that the model does not initialize in a random state so that hyperparameter tuning may be conducted under reproducible states.

```
# fix random seed for reproducibility. Ensures that models are consistently compared during tuning.
seed = 7	np.random.seed(seed)
```

**fig. 34: seed random variable in IagoBot_ValueNet.py**

Secondly, make sure the following two lines are commented as shown in figure 35. For the purpose of hyperparameter optimization, the KerasRegressor wrapper instance estimator is used. Hence any sequential model instance such as baseline_model should not be created.

```
# >>>>>>> build the model
estimator = KerasRegressor(build_fn=baseline_model, verbose=2)
# baseline_model = baseline_model()
# print(baseline_model.summary())
```

**fig. 35: baseline_model in IagoBot_ValueNet.py**
Furthermore, comment out the baseline_model.fit() function.

```python
# >>>>> Fit the model without RandomizedSearchCV
# baseline_model.fit(X_train, y_train, shuffle=True, epochs=24, batch_size=11, verbose=1)
```

**fig. 36:** `baseline_model.fit()` function in `IagoBot_ValueNet.py`

Uncomment the hyperparameter tuning code:

```python
# >>>>> RandomizedSearchCV Hyper-parameter Tuning + Fit
init_modes = ['uniform', 'lecun uniform', 'normal', 'zero', 'glorot normal', 'glorot uniform',
             'kernel_initializers = ['random normal', 'random uniform']
batch_sizes = sp_randint(10, 100)
epochs = sp_randint(10, 100)
optimizers = ['SGD', 'RMSprop', 'Adagrad', 'Adam', 'Adamax', 'Nadam']
output_functions = ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'linear']
activation_functions = ['softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'linear']
n_iter_search = 200
param_distributions = dict(kernel_initializer=kernel_initializers, init_mode=init_modes, batch
nb_epoch=epochs, output_function=output_functions, optimizer=optimizers,
activation_function=activation_functions)

N.B. MSE is negative because RandomizedGridSearchCV is minimizing the result when performing
The actual MSE is simply the positive version of the result. Hence use 'neg_mean_squared_error'
to replace deprecated 'mean_squared_error'. Although output will still be negative.

```RSCV = RandomizedSearchCV(estimator=estimator, scoring='neg_mean_squared_error', param_distr:
n_iter=n_iter_search, verbose=2)
print("Grid fit in progress...")
rscv_result = RSCV.fit(X_train, y_train)
```

**fig. 37:** `RandomizedSearchCV tuning + fit in IagoBot_ValueNet.py`

Uncomment the code responsible for evaluating the tuned ValueNet model using
`neg_mean_squared_error` after tuning:

```python
# >>>>> summarize results for RandomSearchCV
print("Best: %f using %s" % (rscv_result.best_score_, rscv_result.best_params_))
means = rscv_result.cv_results_['mean_test_score']
stds = rscv_result.cv_results_['std_test_score']
params = rscv_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

**fig. 38:** Result summary in `IagoBot_ValueNet.py`
And finally, comment out the following evaluation code for the sequential baseline_model and the save function for the baseline_model. (The Keras Regressor wrapper model does not support saving to .hdf5 via a save() function).

```python
# >>>>>> evaluate the model after hyperparam tuning
# score = mean_squared_error(y_test, baseline_model.predict(X_test))
# print("\nFinal MSE score", score)
#
# >>>>>> save the model
# Model can only be saved once hyperparameter tuning is concluded, and hyperparameters plug
# the baseline_model
# fname = "value_net_model.hdf5"
# baseline_model.save(fname, overwrite=True)
```

**fig. 39: Model save function in IagoBot_ValueNet.py**

Afterward, train the model by running the IagoBot_ValueNet.py file. And specify the number of iterations in the `n_iter_search = N` field. Additionally, the number of cross validations can also be supplied as `cv = M` to the RandomizedSearchCV() constructor. The total number of fits to be conducted would be `N * M`. Depending on the number of fits, the hyperparameter tuning process may take hours to days. Once hyperparameter tuning is concluded, an output would be produced to the console to indicate the best negative mean squared error as well as the list of parameters for the sequential model which resulted in this performance measure.

To fit the model, comment out all the uncommented code in figures 34, 35, 37 and 38 and uncomment the commented code in figures 35, 36 and 39. Supply the list of best parameters returned from the tuning step as the default variables for the baseline_model’s constructor as shown in figure 40.

```python
# This function that we must define is responsible for creating the neural network model to be evaluated.
def baseline_model(kernel_initializer='random_normal', init_mode='glorot_uniform', output_function='softplus', optimizer='Nadam', activation_function='Linear'):
    # create model
```

**fig. 40: baseline_model constructor in IagoBot_ValueNet.py**

Then, run IagoBot_ValueNet.py. This would call the fit() function on a baseline_model with no parameters given, resulting in an instance of the baseline_model built with the default parameters. The fit() function should only take a fraction of the time it took the hyperparameter stage to run. Afterward, mean squared error between X_test and y_test would be outputted to the console. The trained model would also be saved to `value_net_model.hdf5` in the same directory. To expand the available amount of training data, it is encouraged that the fit() function be supplied with the original X and y inputs rather than X_train and y_train. However in this situation, the evaluation code in figure 39 must be commented out.
Note that according to the scikit-learn documentation, the performance measure supplied to the KerasRegressor model is `neg_mean_squared_error` rather than the now deprecated `mean_squared_error` for the following reason:

![neg_mean_squared_error in scikit-learn documentation](image)

Hence MSE results from hyperparameter tuning are all negative while the compilation of the baseline model using the attribute `mean_squared_error` as well as the evaluation of the model using the `mean_squared_error()` function are supported by the Keras Library so they return a positive value during the fitting stage [27].

### 4. Othello Graphical User Interface

Finally, assuming the value_net_model.hdf5 file containing the fully trained model is present in the current directory, the client-end program othello_GUI.py may be run using Python 3. The module will take between 5 – 10 seconds to start up as the ValueNet model is loaded, and present the following user interface:

![IagoBot's graphical user interface start-up interface](image)

To play a game, the user has the option of choosing the color and difficulty of their game using the menu bar. The default setting is that the user plays as black on normal difficulty. To begin the game, select Game → New Game. At any point during the game, the user may clear the board by selecting Game → Clear Board. Alternatively, selecting Game → New Game would initialize the board to the starting state. During gameplay, the user may run out of
legal moves, at which point he can select Game → Pass to yield the turn to the computer. On the other hand, if the computer passes, the status bar will reflect this decision.

III. Experiments & Results

In terms to evaluate this project, two factors must be taken into consideration: The improvement in the performance measure of ValueNet before and after training, as well as the strength of gameplay of IagoBot when playing against a benchmark once the network is fully trained. These evaluation criteria are analyzed in detail in the following sections:

Training Results

Mean Squared Error

As the most popular performance measure for a regression model [28], it was decided during early planning that MSE would be used as the loss function for the CNN. MSE measure the simple squared difference between the predicted label and the actual label. The smaller the difference, the more accurate the model is.

To determine whether the training process of IagoBot has been successful, a baseline must first be established using the MSE of an untrained model. In order to do so, a dataset of 50 games (created by the author) was parsed and then fed into IagoBot_ValueNet.py with the following X and y dimensions.

![In [7]: np.array(value_net_x_input_list).shape](image)

Out[7]: (191, 1, 3, 10, 10)

![In [8]: np.array(value_net_y_input_list).shape](image)

Out[8]: (191,)

Afterward, these input data are split into training and testing data with a ratio of 8:2. The testing data is then used to evaluate the model performance without fitting. The result is illustrated using matplotlib in figure 44 as follows:

![fig. 43: 50-game set’s X input and y label shapes.](image)

![fig. 44: Actual labels vs Predicted labels for 38 independent y_test outcomes. No prior training.](image)
Since the values of the labels range between 0 and 1 (that is 100% certainty for white to win and 100% certainty for black to win), a score of 0.248 is clearly undesirable. One interesting feature of the predicted labels curve when no fits have occurred is that it initializes at the 0.0 mark, whereas an ideal MSE initialization mark should be 0.5. However, as figure 45 shows, this problem is really only an issue for models with little to no training, and that once training has been conducted using an input data size of as small as 191, the predicted labels curve already form a decent approximation of the actual labels curve.

![Graph showing actual vs predicted labels](image)

**fig. 45:** Actual labels vs Predicted labels for 38 independent y_test outcomes. After training.

To reduce the MSE even further, a total of 235,145 unique states was drawn from the Othello World Cup dataset. However, owing to the length of time it took to parse all these states for the ValueNet, 13,966 unique states were constructed instead. These included 9002 duplicated states, with the rest being end-game states.

![Table showing unique states](image)

**fig. 46:** Number of unique states from the world cup dataset.

![Output from othello_game_states_reducer](image)

**fig. 47:** Output from othello_game_states_reducer

After hyperparameter tuning using RandomSearchCV using these data, ValueNet produced a mean squared error of 0.071. On the surface this seems to be only marginally more accurate than the network trained with 50 sets of sample games. However, one must keep
in mind that Othello is still a very complex game with a very high order regression curve. Hence, the marginal increase in mean squared error may be caused the variedness of the new dataset that was passed in, and that the mean squared error is by no means the only criteria for judging actual play strength.

However, in order to determine the actual play strength of the neural net, a total of 12 sample games were played between IagoBot and the Othello application on Android. The final states of these games are shown in Appendix fig. F/G. Out of these 12 games, 6 games were played using Beginner difficulty (minimax DEPTH = 2) and the other 6 were played using Expert difficulty (minimax DEPTH = 4). IagoBot on Beginner difficulty managed to obtain the result of 2Wins/4Losses while on Expert difficulty managed to obtain 3Wins/2Losses/1Draw. While this amount of sample size is in no way representative of the precise play strength of IagoBot, this experiment does prove two things. First, IagoBot Expert mode is at least able to play competitively against the Othello app on level 10, and secondly, the strength of gameplay can be markedly improved by increasing the search depth within minimax_ab.py at the cost of longer computation times.

Hyperparameter Tuning Results
During the hyperparameter tuning stage, various tests were conducted to evaluate the performance results of different combinations of CNN settings. The following is a list of accuracy measures outputted during optimization under a fixed random seed (i.e. deterministic model initialization) but with different numbers of iterations and cross validations for each test case. The tests were also conducted on both the original dataset as well as the world cup dataset.
### Original Dataset

Worst case (3 fits): Best MSE = -0.100521
RandomizedSearchCV params: {n_iter_search = 2, cv = 3}
Best params: {'activation_function': 'linear', 'init_mode': 'he_normal', 'output_function': 'relu', 'kernel_initializer': 'random_uniform', 'batch_size': 23, 'nb_epoch': 40, 'optimizer': 'Adagrad'}

Avg case 1 (50 fits): Best MSE = -0.068161
RandomizedSearchCV params: {n_iter_search = 5, cv = 10}
Best params: {'init_mode': 'glorot_uniform', 'kernel_initializer': 'random_uniform', 'output_function': 'tanh', 'activation_function': 'tanh', 'optimizer': 'Nadam', 'nb_epoch': 24, 'batch_size': 11}

Avg case 2 (50 fits): Best MSE = -0.112121
RandomizedSearchCV params: {n_iter_search = 10, cv = 5}
Best params: {'optimizer': 'Nadam', 'kernel_initializer': 'random_uniform', 'output_function': 'linear', 'activation_function': 'relu', 'init_mode': 'lecun_uniform', 'batch_size': 53, 'nb_epoch': 83}

Avg case 3 (100 fits): Best MSE = -0.106094
RandomizedSearchCV params: {n_iter_search = 5, cv = 20}
Best params: {'init_mode': 'glorot_uniform', 'activation_function': 'tanh', 'optimizer': 'Adamax', 'nb_epoch': 74, 'batch_size': 90, 'kernel_initializer': 'random_uniform', 'output_function': 'softplus'}

Avg case 4 (100 fits): Best MSE = -0.097367
RandomizedSearchCV params: {n_iter_search = 20, cv = 5}
Best params: {'nb_epoch': 17, 'batch_size': 28, 'kernel_initializer': 'random_normal', 'optimizer': 'RMSprop', 'activation_function': 'linear', 'init_mode': 'he_normal', 'output_function': 'relu'}

Good case (5000 fits): Best MSE = -0.67662
RandomizedSearchCV params: {n_iter_search = 500, cv = 10}
Best params: {'optimizer': 'Nadam', 'activation_function': 'tanh', 'nb_epoch': 24, 'output_function': 'tanh',}
Seems to be the most efficient optimizer. This is no surprise because the ADAM optimizer works well with regression models [29], and NADAM is simply the ADAM algorithm compounded with a Nesterov momentum component which makes the learning rate adaptive and can result in a quicker rate of convergence [30].

2. random_normal is always preferred when trained with the large world cup dataset, while random_uniform is sometimes preferred when training with the original dataset.

3. glorot_uniform is the preferred initializer for the weight values. This is when initialization values are selected from the range $[-\text{limit}, \text{limit}]$, where

$$\text{limit} = \frac{6}{\sqrt{\# \text{ of weight matrix inputs} + \# \text{ of weight matrix outputs}}}$$

4. softplus is the preferred output function for the final layer under the world cup dataset, while relu, tanh and softplus all produce good results under the original dataset. In actuality, softplus is simply a smoother version of ReLU as shown below, and is defined as: $f(x) = \ln(1 + e^x)$
5. It is not certain whether a linear activation function does indeed work better for a large dataset because it is known that functions such as ReLU allow for a faster convergence because they are not bounded by an upper limit. Hence this result remains to be verified.

6. There does not seem to be a correlation between batch size / number of epochs and the performance measure, however this may be caused by the large range values for these variables to choose from.

Correlation Between Search Depth and Time Taken

The DEPTH parameter in minimax_ab.py is altered whenever the user chooses a difficulty. By default, Beginner difficulty corresponds to a DEPTH of 2, Intermediate corresponds to a DEPTH of 3 and Expert corresponds to a depth of 4. In order to determine whether these settings produce any variation in the time taken for the minimax algorithm to run, the debug_timer.py module was used to run minimax_ab.py under different DEPTH values. The following table is the result of that experiment:

<table>
<thead>
<tr>
<th>DEPTH</th>
<th>Minimax</th>
<th>Minimax with α β Pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.95</td>
<td>0.60</td>
</tr>
<tr>
<td>4</td>
<td>8.00</td>
<td>4.00</td>
</tr>
<tr>
<td>5</td>
<td>113.00</td>
<td>25.00</td>
</tr>
<tr>
<td>6</td>
<td>Did not terminate</td>
<td>Did not terminate</td>
</tr>
</tbody>
</table>

*Table 3. Execution times of Minimax and α β Minimax algorithms with respect to different depths.*

To ensure that the experiment is fair, all minimax algorithm instances were tested on the starting grid of an Othello game with the same board layout as shown in figure 13. Two conclusions can be drawn from this experiment: First, the minimax algorithm does indeed perform much better when augmented with alpha-beta pruning, although it is difficult to conclude from these results that the decrease in time taken is sufficient to warrant the upscaling of the DEPTH variable during gameplay. Second, the Normal difficulty should indeed be set with a DEPTH of 3 as it strikes the perfect balance between search time and performance.

![fig. 49: Softplus function vs. ReLU](image)
Overall Remarks

The following remarks address both points of interest with respect to elements in this project as well as deficiencies within the current implementation of the project:

1. Owing to the large amount of endgame data available relative to the duplicated midgame board states, the neural network tends to overfit on endgame data. This is problematic because end states are usually very different from middle states in that before the endgame, the player with the least number of pieces on the board would generally have an advantage because he would tend to have the greatest number of choices in placing the next piece, and therefore dictate the outcome of the game. However, when the neural network fits on the endgame data, intuitively the more piece a player has on a board, the more likely he is going to win. For this reason, IagoBot tends to maximize the capture of the opponent’s pieces early on in the game, resulting in a bad strategy. Figure 50 illustrates this problem:

![Figure 50](beginner-setting-game-3-on-level-10-difficulty-with-the-othello-app-iagoBot-loses)

2. Because of the way win likelihood is calculated (form endgames and duplicate states), any training labels passed in tends to be either 0, 1, or some simple fractions such as $\frac{1}{2}$, $\frac{1}{4}$ etc. This does not form a smooth regression curve, which is best illustrated by figure 45, which is bad for a regression problem. A possible solution to this problem is to modify the way win likelihoods are calculated by also evaluating the number of potential moves a player is able to make for any given state. A large number of potential moves would increase the win likelihood and vice versa.

3. An interesting observation about the way IagoBot plays is that it seems to always capture edges whenever possible (figure 51). However, because of the limited search depth on the game tree, often times it doesn't know how to get there. Again, this seems to be a symptom of overfitting on endgame states and not having enough midgame states that capture edges/corners. One way of fixing this issue would be to simply save a limited number of endgame states during the game reduction stage.
4. Finally, a characteristic of IagoBot is that it does not perform well when facilitating corner captures. Again depicted in figure 51, IagoBot does not take steps to avoid the capture of the undesirable X-cell at (1, 6), or the C-cell at (0, 6), nor does it see the value in capturing A/B cells (See Appendix X for detailed analysis of labeled cells). Because corner-capture is an important strategy for Othello, if this rule is incorporated as a PolicyNet property, then the AI would likely perform much better than it does currently.

**fig. 51:** Expert setting, game 2 on level 10 difficulty with The Othello App.
IV. Conclusion & Future Works

Project Overall Review

Overall, the IagoBot project achieved the desired goal of achieving a good quality play strength via deep learning neural networks. Additionally, it was demonstrated the deep learning neural network was indeed able to achieve a high level of play provided a sufficient amount of mid-game training data was provided to prevent the model from overfitting on the endgame states. The use of convolutional neural networks as the regression model in ValueNet was also shown to be effective, as it was able to achieve a low mean squared error even with a low number of training data. However, one of the original goals of this project was to examine the time and memory consumption levels of a neural network approach to an Othello AI and whether this approach would prove more efficient than traditional tree search approaches. This hypothesis could not be confirmed at the end of the project, since IagoBot was not able to produce faster gameplay than The Othello application unless it is in Easy mode. However, this again may be attributed to the fact that the ValueNet was provided with insufficient midgame data compared with endgame data. Therefore, in order to increase the performance accuracy of IagoBot, the most importance change to make would be to supply the network with high quality training data.

Potential for Extension

The following extensions can be taken into consideration in order to bolster the performance levels of IagoBot in upcoming iterations:

Policy Net

The policy network can be regarded as a helper neural network IagoBot that enhances the predictive accuracy of the existing ValueNet. It would be responsible for displaying a probability distribution (probability map) of the likelihood that an expert player would move to a particular cell over all legal moves on the board at a particular state within the game. The PolicyNet is based on a convolutional neural network because CNNs generally perform very accurate image classifications. The policies can be expressed as $p(a | s)$, where $p$ is the probability distribution for all action $a$ undertaken from the previous state to the current state $s$.

The architecture of PolicyNet is shown in Appendix figure D. As clearly illustrated by the figure, the network consists of an input layer of dimension 8 x 8 x f, representing the 8 x 8 Othello board with f number of input features. These input features are as follows:

<table>
<thead>
<tr>
<th>Feature</th>
<th># of planes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board layout</td>
<td>3</td>
<td>Visual representation of the board</td>
</tr>
<tr>
<td>Player</td>
<td>1</td>
<td>Whose turn to play</td>
</tr>
<tr>
<td>Actions</td>
<td>1</td>
<td>Possible moves for the player</td>
</tr>
<tr>
<td>Sandwiched</td>
<td>1</td>
<td>Number of opponent’s pieces sandwiched</td>
</tr>
<tr>
<td>Weighted cells</td>
<td>1</td>
<td>Cell weighting adjusted for A, B, C, X squares</td>
</tr>
</tbody>
</table>
Once all the input features are put together, one or more number of filters of will convolve through the input layer, outputting the dot product of all the input values at every step of the process. The depth of the output plane is determined by the number of filters used. This is a hyperparameter that will be tuned by the validation dataset. Owing to the nature of convolution, the output layer will naturally have both reduced height and width compared with the input layer. Therefore, in order to prevent successive layers from shrinking in size, the ‘border_mode’ hyperparameter will be set to the value ‘same’. This is a hyperparameter provided by the Keras CNN library, and it will zero-pad each layer to ensure that dimensionality is preserved. The output layer from the convolution process will in turn act as the input layer for the third layer until the final layer is reached.

At the end of each layer, an activation function is required. A rectified linear unit activation function is best suited for this project. ReLU is popular for deep learning since a deeply layered neural network tends to have a diminishingly smaller gradient toward the output layer. Hence it may be desirable to amplify the magnitude of the output after each activation. The following figure is a visual representation of the ReLU activation function.

![ReLU activation function](fig. 52)

As illustrated by Appendix figure D, IagoBot’s PolicyNet will consist of two convolutional layers before every max-pool layer. The max-pool layer serves to reduce the dimensionality of its input layer so as to progressively cut down the spatial size of the board representation. This decreases the amount of parameters and computational load on the network, and hence also controls overfitting.
The pattern of \{CONV, CONV, Max-Pool\} will continue until the end of the PolicyNet, where the board has been reduced to a 1x1 fully connected layer. This is a dense 1-layer perceptron vector with a softmax output function. The final output of this fully connected layer would be an 8x8 probability map of the likelihood of the current player to place a piece on various board locations under perfect play.

API

Currently, in order to evaluate the play strength of IagoBot, games would have to be played manually against an opponent. This can be very time consuming since on Expert difficulty, each move performed by IagoBot takes anywhere between 1 second to 15 seconds. The lack of quantity in the amount of testing games played means that any measure of play strength is subject to bias and not statistically representative of the actual play strength. For these reasons, it may be pertinent to construct an API for IagoBot that would allow the program to play repeated games automatically with other Othello programs. This API would be implemented as an extension to othello_main.py, and would be responsible for alternating the execution between IagoBot and an opponent AI. The results generated from this API would then be conclusive in judging the play strength of IagoBot relative to another program.
Appendix

fig. A:

An A cell is advantageous to obtain, as it can lead to the opponent capturing a C cell, which in turn allows the player to capture a corner.

The B cells are neutral cells and can lead to the capture of a corner depending on the situation.

C cells should be avoided as they do not facilitate corner captures and instead allows the opponent to capture a corner.

X cells should almost never be played as they allow for the immediate capture of a corner by the opponent.

fig. B:

The three visual input features of the ValueNet and PolicyNet which represent the visual layout of the Othello board.

Feature layer 1: Empty cells on the board are activated.

Feature layer 2: Cells containing black pieces are activated.

Feature layer 3: Cells containing white pieces are activated.
**fig. C:**

Overall architecture of ValueNet. The initial input layer is of dimension $32 \times 10 \times 10$ where 32 is an arbitrary batch size and 10 is the length/width of the augmented Othello board. The red Dropout layers are not physical layers, but the result of performing the dropout function on the previous convolutional layers. FC stands for fully connected layer, which are outputted from CONV2D #2 after being flattened into a 1-dimensional dense layer.

The final output after the Softplus activation function is a single value in the range of $[0, 1]$, where 0 stands for 100% likelihood that the white player will win the game and 1 stands for 100% likelihood that the black player will win the game for any given board position.
fig. D:

Overall architecture of PolicyNet. \( f \) stands for the number of input features. \( n \) stands for the number of filters.

The final output is a probability map of 8 x 8.
Annotation: An arrow from A to B indicates a dependency relation from A to B. The orange dashed region shows files and programs that are created for the purpose of debugging. The orange arrow indicates a dependency that can be refactored into the code as a future extension. The OthelloGameDataParser.java file is considered a separate entity and is not included in this diagram.

Works Cited


[27] Keras [Internet]. [nd; cited 2017 Apr 10]. Available from: https://keras.io/losses/#usage-of-loss-functions


[31] Keras [Internet]. [nd; cited 2017 Apr 10]. Available from: https://keras.io/initializers/#glorot_uniform
