Final Year Project Detailed Intermediate Report

Project Title: Playing Othello by Deep Learning Neural Network

Author: Kuai Yu (David)

Supervisor: Prof. K.P. Chan.

UID: 2013507514

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.

Abbreviations

CNN – Convolutional Neural Network
CPU – Central Processing Unit
CSV – Comma Separated Values
CUDA – Compute Unified Device Architecture
GPU – Graphics Processing Unit
PolicyNet – Policy Neural Network
RAM – Random Access Memory
ReLU – Rectified Linear Unit
ValueNet – Value Neural Network

Table of Contents

List of Figures

List of Tables

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 aims to create a game-playing artificial intelligence program that uses similar deep learning techniques as AlphaGo but instead of Go, the program will play the much simpler board game Othello using a combination of tree search algorithms and neural networks. The program will be implemented in Python and will offer a simple graphical user interface to play against human challengers.

Introduction

Project Background

The AlphaGo program combines Monte-Carlo tree search with deep 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.

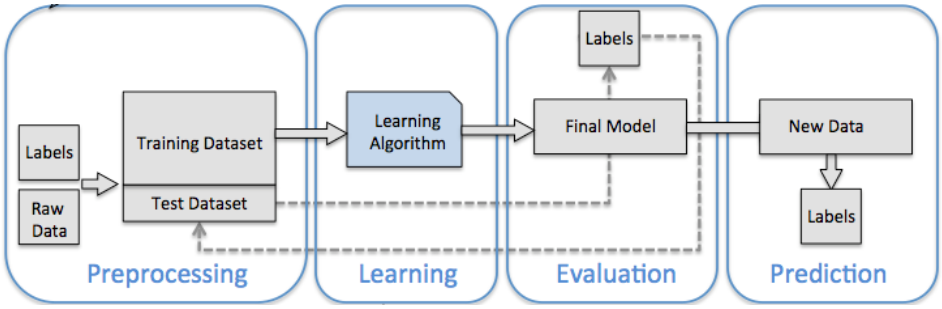

fig.1: The fundamental working principle behind any machine learning algorithm is its ability to adjust a matrix of weight values over time based on an input set of training data and the desired outputs. These weight values are then used in combination with new data to produce a prediction.

This project aims to create 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, my program (henceforth named "IagoBot") will specialize 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 at the very least, minimize) 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 [3].

In terms of specific strategy for winning a game of Othello, this project will focus on the goal of 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.

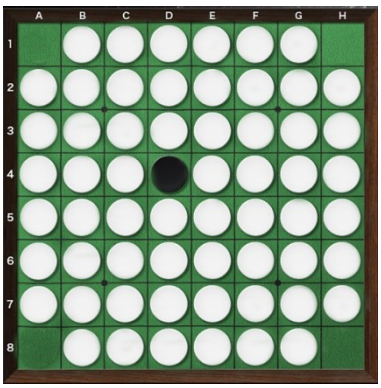

fig.2: A winning position for black, who has control over all four corners of the board. Since corners are the only cells on the board that, once captured, can no longer be recaptured by the opponent, dominating them generally leads to a win.

As a means of facilitating corner capture, the following cells will be named in order of desirability or likelihood to lead to a corner capture: The A cells, B cells, C cells and X cells [4]. (See fig. A in Appendix). These cells will be adjusted for desirability by being assigned different weight values during the training stage of the policy network.

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 [5]. 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 is to produce an interactive program that will satisfy the following objectives and criteria toward the end of the project timeline:

Firstly, the key technology behind the program should be two neural networks that work in tandem to evaluate a game of Othello. 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.

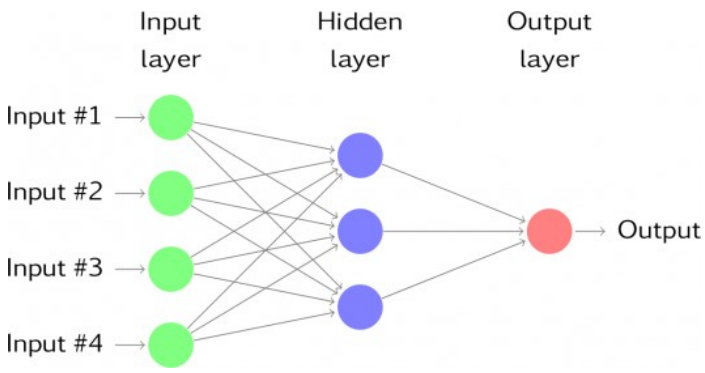

fig.3: The value network. Each node/neuron in this network takes a rudimentary value such as (-1. 0. 1), to indicate the state of that particular cell. This value is combined with an initialized weight value pertaining to that node and then passed into an activation function. Depending on the output of the activation function, nodes in the hidden layer(s) may or may not be activated. The output would be a single double value.

The two neural networks consist of a value network and a policy network. The former is responsible for evaluating the position of a board in any given game for a selected player (in the form of a double) while the latter makes decisions on which path to take in the game tree based on the value network. The value and policy networks must then be combined in a tree-search algorithm with a roll-out policy. An evaluation of this part of the project is detailed in the methodology section.

Another constraint which IagoBot must abide by is that it must be a multi-threaded program. Due to the fact that neural networks are much more costly to run than traditional search heuristics, a multi-threaded approach which takes advantage of both CPU and GPU power is a must. The exact number of threads to use will be determined at a later stage of the project.

A further objective which this project aims 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 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. Depending on how much time is left over at the end, it would also be a good idea to discuss with other groups to develop a unified API by which different versions of Othello programs may play continuously against one another and come up with a ranking list determined by the relative strength of each program. 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.

Project Methodology

Design

This project comprises three phases: the collection phase, during which sample games are created and subsequent training data are recorded, the construction phase, during which two neural networks are created: the policy network and the value network, and the training phase, during which the program is fed sets of training data over a period of several days to allow for the optimization of their respective weight matrices. A tree-search heuristic is then used to bring both networks together in a decision-making algorithm.

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*

Construction Phase

For the construction phase, Python is selected as the language to code IagoBot for several 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 arises.

```
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]')
```

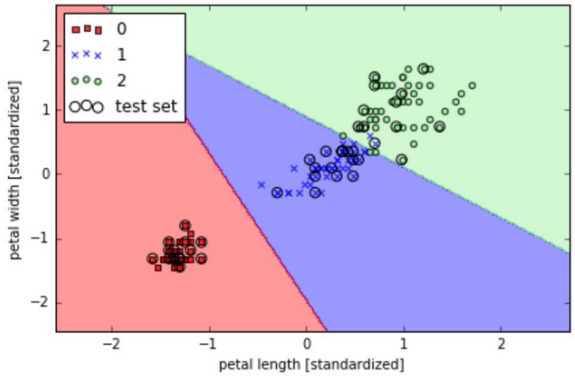

fig.4: An illustration of the various library functions available imported and trained and new test data classified in just a few lines of code using scikit-learn and matplotlib.

PolicyNet

The policy network is the more complex of the two neural networks to be used in IagoBot. It displays 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. They are adept at facial recognition, working with complex imagery and allows for the extrapolation of low level patterns into high level features [6]. For instance, if a strategy in winning Othello is corner control, then it is feasible to use a filter within a CNN implementation which activates whenever corners are captured, thus correlating the number of corners captured to the probability of winning the game. The policy 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 C. 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:

Feature	# of planes	Description
Board layout	3	Visual representation of the board
Player	1	Whose turn to play
Actions	1	Possible moves for the player
Sandwiched	1	Number of opponent's pieces sandwiched
Weighted cells	1	Cell weighting adjusted for A, B, C, X squares

Table 1. The various planes of the input layer of the PolicyNet CNN. Board layout is the visual representation of the board state with 3 planes, each representing the empty cells on the board, the black cells and the white cells respectively. The player feature represents whose turn it is to play. Actions feature represents the location of all legal moves open to a player. Sandwiched feature represents a board state where each cell corresponds to the number of opponent pieces that can be sandwiched if a player chooses to play that particular cell. Weighted cells feature assigns a different value in order of desirability to the A, B, C and X cells on the board. This encourages exploration of A and B cells, and discourages exploration of C and X cells in order to facilitate a corner-capture strategy and to prevent the opponent from capturing the corners instead.

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.

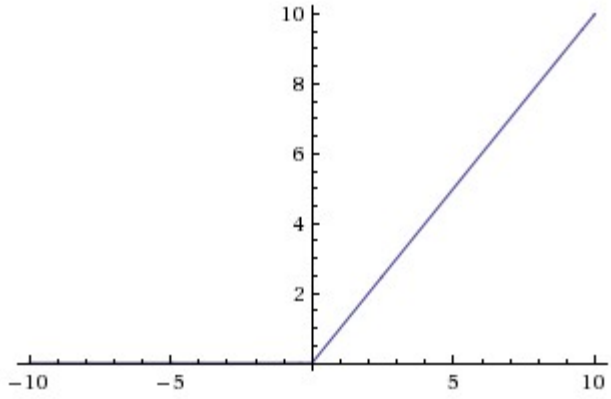

fig.5 ReLU activation function, where the activation output is never below 0 and is not upper-bounded by any value. This allows for the amplification of output after each layer.

As illustrated by Appendix figure C, 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.

ValueNet

The value network is almost identical to the policy network in that it is also a convolutional neural network with 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 (see Appendix figure B) and an output consisting of one scalar value. This value represents the percentage likelihood of winning for the current player given a board layout.

Training Phase

In the third phase of this project, IagoBot will be trained by means of supervised learning. Of course, 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 totally 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 is less than 15,000. 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. IagoBot would take these values as the "golden standard", compare them with its actual output at the end of each iteration, and adjust its weight matrix accordingly. Both networks would be trained in a similar manner.

Finally, IagoBot will utilize 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

and policy networks in a single heuristic that selects what move to make via a lookahead search, essentially acting as the primary decision-making agent within IagoBot. The algorithm would the policy network to work out the probability distribution of the likelihood of making certain moves and the value network to decide the odds of winning from a particular board position. There are two possible algorithms that can be used to achieve this goal: a minimax algorithm with alpha-beta pruning and a Monte-Carlo Tree Search algorithm with a fast rollout policy.

Minimax with Alpha-Beta Pruning

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

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

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 would use the ValueNet to predict the likelihood of winning for every given board position, and then use PolicyNet to decide the optimal move for the player. Since the PolicyNet returns a probability distribution over all possible cells, the minimax algorithm can use these values to label every node in the game tree. However, it would be prudent to reduce the memory load of IagoBot by augmenting minimax with alpha-beta pruning to rule out obvious cases which would lead to a player to lose. Otherwise, the unpruned game tree would still be very computationally expensive to simulate.

Monte-Carlo Tree Search with Fast Rollout

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 gameplays from leaf nodes in each partial game tree. For this project, a MCTS approach is not necessary, and will only be implemented in the case where the minimax method does not provide a satisfactory play-strength.

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:

AI Name	Time restrictions	CPUs	GPUs	Elo
Distributed AlphaGo	5 seconds	1202	176	3140
AlphaGo	5 seconds	48	8	2890
CrazyStone	5 seconds	32	-	1929
Pachi	400,000 sims	16	-	1148
Fuego	-	1	-	431

Table 2. 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 [7] is an arbitrary scale for play-strength that will not be used for this project.

It is important to note that in table 2, 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.

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 6 is a data point from the set of training data in the 64^{th} dimensional feature space, where each feature is a cell on the 8 x 8 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 x_1 and x_2 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 (shown in figure 6). 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.

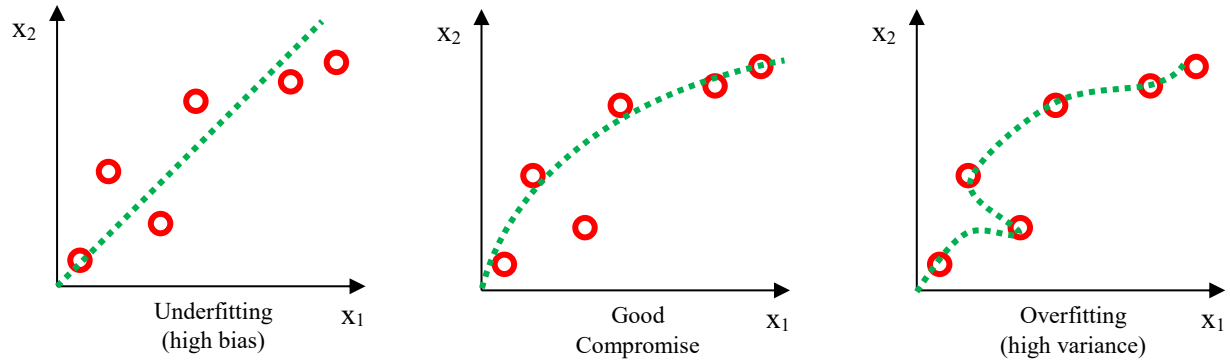

fig.6: An illustration of the curves produced in a regression problem by selecting different regularization values

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:

$$\Phi(X,T,\Theta) = L\big(X,T,Y = f(X,\Theta)\big) + P(\Theta)$$

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.

Risk #2: Unable to Find Suitable Hyperparameters

It is often inevitable during the validation stage for any neural network model that the abundance of hyperparameters would result in a poor prediction accuracy. To see why excessive hyperparameters is a problem, assume that there are 10 hyperparameters, each with 10 distinct settings. In order to find the most suitable setting for every hyperparameter and with a lack of experience or intuition to guide the process, a total of 10^{10} combinations would have to be simulated. This is clearly an infeasible calculation even with today's computing 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.

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

Given a set of parameters $\mathbf{p_i}$ each with $\mathbf{N_i}$ 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. Randomized GridSearchCV is also a library function provided by scikit-learn which makes the validation process easy to implement.

Risk #3: Value Network Unable to Detect Board Edges

Owing to the nature of convolution, the input 8 x 8 grid will be completely garbled by the various filters to pass through the input layer. 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.

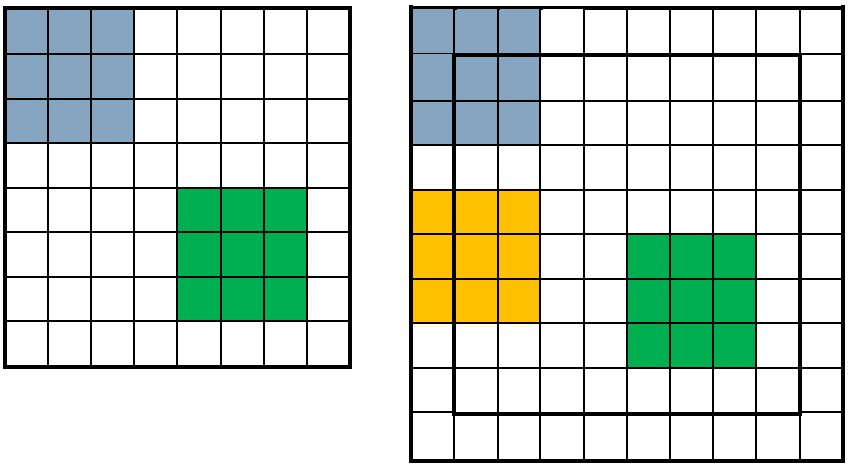

fig.7: Before and after input layer padding.

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

After padding, the CNN will 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. On the other hand if it is necessary for the policy network to be able to know exactly where each filter has traveled during the convolution stage in order to strategize on a corner attack, it may be possible include an additional input feature which describes the physical location of each cell.

Current Progress

Recording Sample Games

In order to record sample gameplay for the construction of a training set, a third-party application (The Othello [8]) 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 [9] 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 [10].

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.

Reconstructing Sample Games from Data

A program called the Dynamic Win-Likelihood Labeler (dywinlilab.ipynb) is built to help reconstruct board layouts from existing game data.

This program reads in each line/move from all the game files and then recreates the layout of the board using Othello rules based on the previous layout. At the same time, the program searches through the database for identical board layouts from other games. If a match is found, then the win likelihood of all these board layouts is updated as follows:

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

Constructing the ValueNet

The value network has been constructed using Jupyter Notebook and is available in the project website under IagoBot_ValueNet.ipynb

Currently, the value network is undergoing the training process and contains two convolutional layers, one max pool layer and two densely connected networks in the end with a softmax activation function. The exact construction is as follows:

```
____________________________________________________________________________________________________ Layer (type)
Output Shape Param # Connected to
==================================================================================================== convolution2d_1
(Convolution2D) (None, 32, 32, 32) 896 convolution2d_input_1[0][0]
____________________________________________________________________________________________________ dropout_1
(Dropout) (None, 32, 32, 32) 0 convolution2d_1[0][0]
____________________________________________________________________________________________________ convolution2d_2
(Convolution2D) (None, 32, 32, 32) 9248 dropout_1[0][0]
____________________________________________________________________________________________________ maxpooling2d_1
(MaxPooling2D) (None, 32, 16, 16) 0 convolution2d_2[0][0]
____________________________________________________________________________________________________ dropout_2
(Dropout) (None, 32, 16, 16) 0 maxpooling2d_1[0][0]
____________________________________________________________________________________________________ flatten_1
(Flatten) (None, 8192) 0 dropout_2[0][0]
____________________________________________________________________________________________________ dense_1 (Dense)
(None, 512) 4194816 flatten_1[0][0]
____________________________________________________________________________________________________ dense_2 (Dense)
(None, 10) 5130 dense_1[0][0]
==================================================================================================== Total params:
4210090
```

fig.8: IagoBot_ValueNet.ipynb output

Note however that this architecture has not been finalized, and is liable to be altered in order to maximize the prediction accuracy of the value network.

Running Memory Intensive Tasks via the GPU

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.

Project Schedule: Tasks & Milestones

Item	Deadline	Deliverables	Completion
1	September 30th, 2016	Familiarize with Monte-Carlo tree search algorithms. Familiarize with linear regression and how it can be used to minimize the loss function for a neural network using supervised learning.	Completed
2	October 8th, 2016	Record 50 sets of sample games to be shared with other groups.	Completed
3	October 31st, 2016	Create a program to parse game data and record every game state within a game. Dynamically update the win-likelihood of each game state that is duplicated in the game set.	Completed
4	November 30th, 2016	Design and implement value network.	Completed
5	December 31st, 2016	Familiarize with minimax algorithms with alpha and beta pruning. Finish Deep Learning Tutorial (by LISA lab, University of Montreal). Continue work on value network.	Completed
6	January 15th, 2017	Train Value Network	In Progress
6	January 22nd, 2017	Interim report Design and Implement policy network.	In Progress
7	February 28th, 2017	Begin training stage of policy network, run continuous training data to optimize neural network loss function. Optimize via hyper-parameter tuning.	Waiting
8	March 10th, 2017	Design and implement Minimax algorithm based on value and policy networks. Implement main game controller.	Waiting
9	March 31st, 2017	Begin final round of human testing, collaborate with other groups to produce standardized API for inter-group tournament if time allows.	Waiting
10	April 21st, 2017	Deliver finalized implementation. Submit final report for review.	Waiting

Appendix

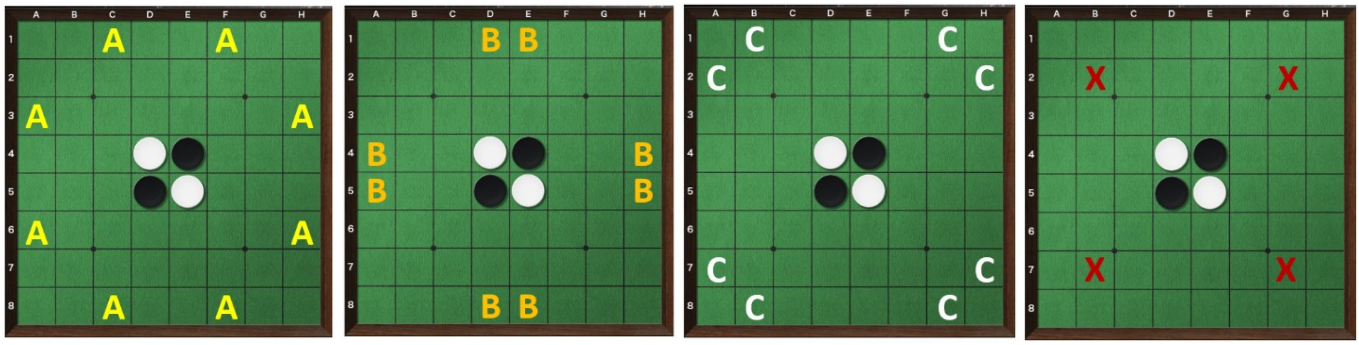

fig. A:

An A cell is advantages 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.

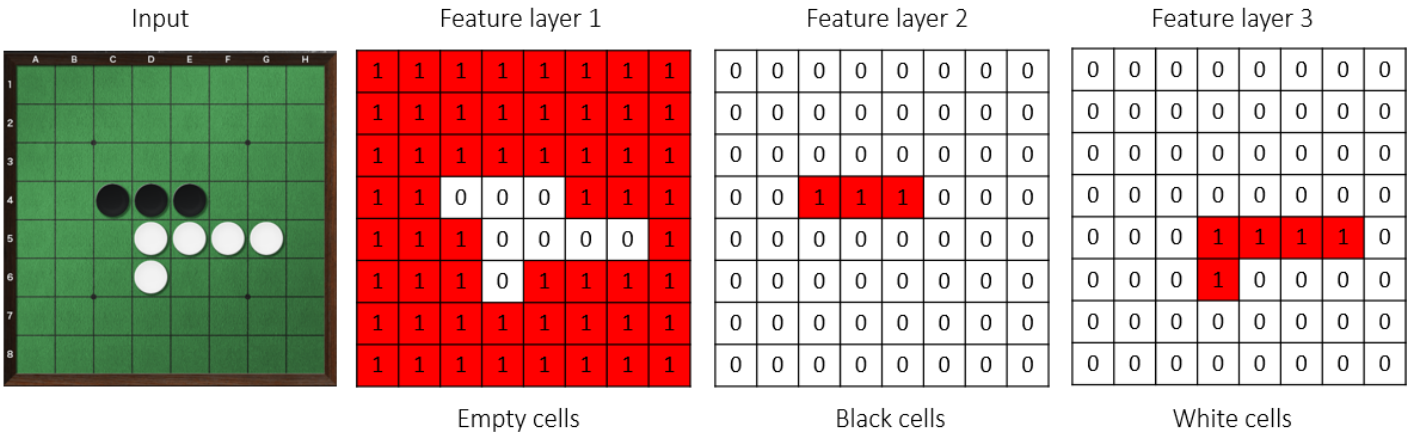

fig. B:

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.

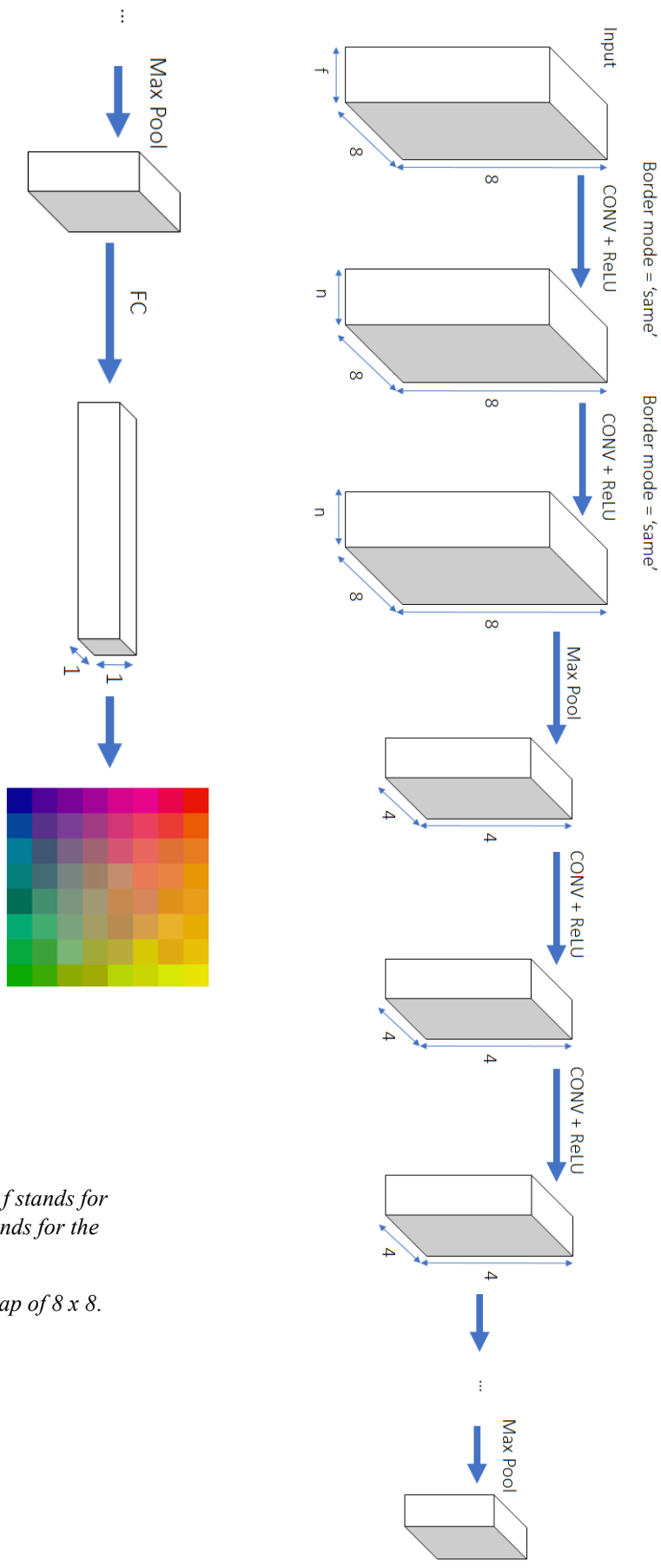

fig. C:

Overall architecture of 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.

Works Cited

[1] Go Game Guru [Internet]. [updated 2016 May 20; cited 2016 Sep 19]. Available from: https://gogameguru.com/tag/deepmind-alphago-lee-sedol/

[2] DeepMind [Internet]. USA: DeepMind. [nd; cited 2016 Sep 19]. Available from: https://deepmind.com/research/alphago/

[3] Rosenbloom P.S. A world-championship-level Othello program. Pittsburgh (U.S.): Carnegie Mellon University; 1981. Available at: http://repository.cmu.edu/cgi/viewcontent.cgi?article=3452&context=compsci

[4] Othello Lesson 2. [Internet]. [updated 2008 Dec 7; cited 2017 Jan 12]. Available from: https://www.youtube.com/watch?v=at0g9zibQZY

[5] Buro M. Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello. New Jersey (USA): NEC Research Institute; nd. Available at: https://skatgame.net/mburo/ps/improve.pdf

[6] CS231n Winter 2016 Lecture 7. [Internet]. [updated 2016 May 14; cited 2017 Jan 12]. Available from: https://www.youtube.com/watch?v=AQirPKrAyDg

[7] Coulom R. Computing Elo ratings of move patterns in the game of Go. ICGA J. 30, 198–208 (2007).

[8] The Othello™ Application: https://play.google.com/store/apps/details?id=jp.co.unbalance.android.othello_free&hl=en

[9] Othello Game Data Parser: http://i.cs.hku.hk/fyp/2016/fyp16038/assets/docs/OthelloGameDataParser.java

[10] Playing Othello By Deep Learning Neural Network [Internet]. HK: HKU. [updated 2016 Oct 20; cited 2016 Oct 20]. Available from: http://i.cs.hku.hk/fyp/2016/fyp16038/

[9] Theano [Internet]. Montreal: LISA Lab. [updated 2016 Nov 10; cited 2016 Oct 20]. Available from: http://deeplearning.net/software/theano/tutorial/using_gpu.html

(fig.1 & fig.5) Raschka S. Python Machine Learning. 1st ed. UK: Packt Publishing; 2015

(fig. 3) Neural Network Models [Internet]. [nd; cited 2016 Sep 19]. Available from: https://www.otexts.org/fpp/9/3

(fig. 5) CS231n Convolutional Neural Networks for Visual Recognition [Internet]. [nd; cited 2017 Jan 12]. Available from: http://cs231n.github.io/neural-networks-1/