BetaGo:
Predicting Professional Players' Moves in Go Game

Contents

Part One: Introduction

1.1 Go Game

Go or “Weiqi”, as it is known in China, is an ancient board game thought to have been invented around 2300 B.C. by a Chinese emperor with the intention of teaching his son tactics, strategy, and concentration[1]. Go is played by two players on a 19 by 19 grid of lines where players take turns placing either black or white stone pieces on intersection points. If one player’s stones are surrounded by opponent’s stones, they are removed from the board.

The goal of the game is to capture as much board territory on board as possible. As there are 19 x 19=361 intersection points, an odd number of board positions, the game is designed such that one player will win by a minimum of 1 stone in the closest game. Details of the Go rules can be found in reference [1].

Numerous attempts have been made in artificial intelligence research to program computers to play Go[1, 2, 3]. With the recent exception of Google’s AlphaGo[4], none have been able to play at professional level. This is mainly because the game of Go has a massive search space. The number of possible games is approximately $361! \approx 10^{768}$, much more than the total number of atoms in the universe [5]. Brute force search of all Go Games is theoretically possible but computationally intractable.

1.2 AlphaGo and Deep Neural Network

In March 2016, AlphaGo, a Go playing program developed by Google DeepMind defeated the Korean Go world champion Lee Sedol by 4 to 1 in a five-game match. How does AlphaGo do it?

In broad terms, recent advances in deep neural network in image classification and facial recognition were readily and successfully transferred to pattern recognition in the game of Go by Google’s DeepMind. The board of a Go game can be thought of as an image with 361 pixels where each pixel can be expressed as one of three values (empty, black stone, white stone), shown in Figure 1. This tendency to think of the game as an developing image is reflected in the language used by professional Go players. Often, players evaluate the strength of a move in visual and intuitive vocabulary, by saying “this is a good move because it makes a very beautiful shape for white stone in the bottom right corner.” In retrospect, It’s not surprising that the performance of deep learning in image recognition can be used towards Go game.

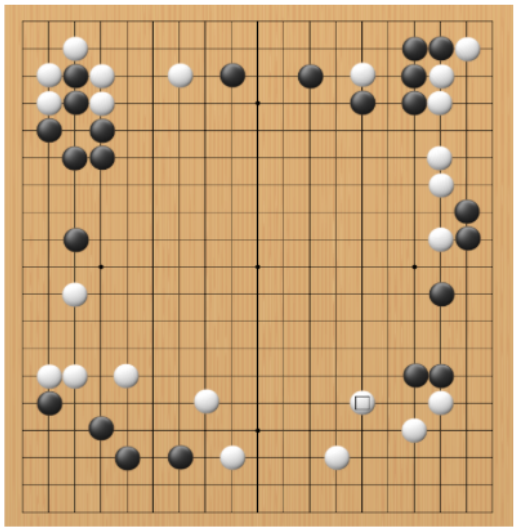

Figure 1: Analogy of a Go board to a picture

More specifically, the major part of solving Go Game is to reduce the size of search trees. Two methods can be used: (1) Reduce the depth of search tree by truncating the search tree and replacing the truncated subtrees with a value function which predicts the outcome. This means that the search doesn't have to look many steps ahead. (2) Reduce the breadth of the search by removing the unlikely subtrees based on a probability distribution, and this means that the search doesn't have to consider very unlikely or obviously bad moves.

Google DeepMind utilized and optimized both of these methods using deep learning: (1) They designed a "value network" to evaluates any board position and predicts the probability of either side winning the game, thus reducing the depth of search tree (2) They designed two policy networks to evaluate any board position and to predict the most likely subsequent moves, thus reducing the width of the search tree. Integrating the value and policy networks with Monte Carlo Tree search, an existing Go strategy which plays at very strong amateur level, and boosted by google's vast hardware infrastructure and resources, AlphaGo was able to push the strength of the AI Go above professional human level.

1.3 Scope of this Project

AlphaGo's policy network has two components: (1) The supervised learning policy network (SL-policy network) takes professionally played Go games from historical records as training examples from which the algorithm tries to maximize the accuracy of predicting the next moves of professional players. (2)The reinforcement learning policy network (RL policy network) allows the program to play against itself, learning from moves that could eventually lead to a victory.

The RL policy network corrects potential errors of the SL policy network as even professional players make bad/suboptimal moves.

In this project, I am implementing a simple version of the SL policy network, i.e, predicting professional player's move based on historical collection of games. The goal is to gain an understanding of deep neural network architecture and practices. I have named it BetaGo because of its suboptimal performance and simplicity. However, it does lend to substantial understanding of the subject of the deep neural network and various challenges faced (and solved) by researchers in this area.

Part Two: Dataset

I downloaded ~86,000 professional games dating as far back as 196 AD from gogodonline.com. Go games files are recorded in "Smart Game Format" (SGF) [6]. Each SGF file has a play by play record of a single Go game while documenting the names of each player, as well as the time and location of the game. The sequential moves are in the format of "W[ed]; B[jp]; W[ef]; B[dd]; W[qf]….", meaning "white stones goes to position ed, black stones goes to position jp, white stone goes to position ef", and so on.

I wrote a script that parses SGF files and and returns a series of matrices for each game. At any point t in a game, the board position is p_t followed by a move M_t, the following steps are performed: (1) Evaluate if the move if legal, for example, if the board already has a stone at the position then it is illegal to put a new stone at that position. If the move is illegal, an error is raised. Since all the games downloaded are professionally played game, no illegal moves were identified. (2) Add either a black or a white stone specified by M_t to the board p_t (3) Evaluate p_t plus the stone added by M_t. If opponent's stones are surrounded after M_t, then the surrounded opponent stones are removed from board. (4) Generate a new board position p_{t+1} as feature of the next training example, completed by the next move M_{t+1} as the label.

If a game has n steps, then n matrices (board positions) $[p_1, p_2, \ldots, p_n]$ and n corresponding nexts moves $[M_1, M_2, \ldots, M_n]$ are generated from that game, in which the board positions are the features, while the next moves are the labels to predict. The conversion from a go game to an SGF file and finally to a matrix is illustrated in **figure 2**.

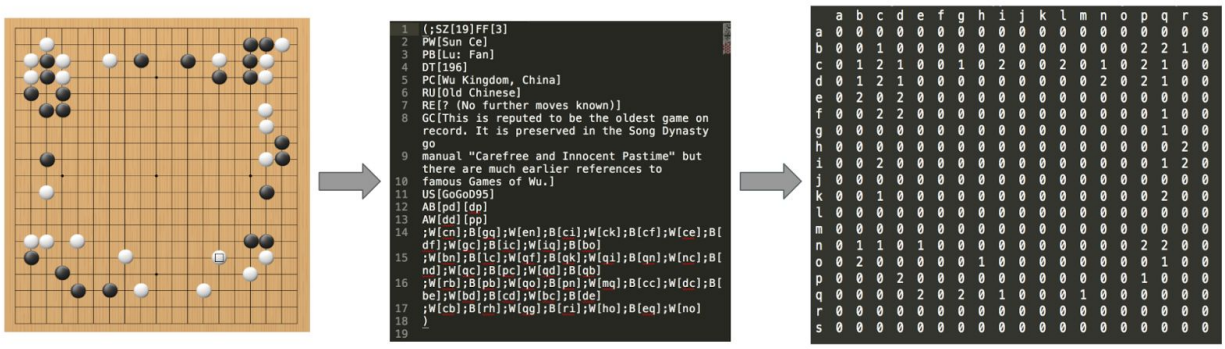

Figure 2: Conversion of a go board to a SGF format to a matrix.

On average there are 207 steps in each of the 85931 games, generating a total of 17,801,121 training examples (comparable to the 30 million training example that AlphaGo is based on). In the models, I separated the training examples to 60% training, 20% validation and 20% testing samples.

Despite having a large quantity of samples, due to the lack of computing resources such as parallelizing infrastructure, CPUs and GPUs, using all 17 million sample is too computationally draining for my laptop or the 8 GPUs in hyades cluster that I have access to. Therefore in the end, only about 1% of the samples are actually used in my project to produce results within a reasonable amount of time.

Part Three: Models and Results

3.1 General Result

The best model from this project reaches a ~4% prediction accuracy. That is a 10 times improvement over 0.42% accuracy of random guess. However, AlphaGo reports a ~50% accuracy in its paper [4]. The reasons for AlphaGo's 10 times better performance are multiple. They include: (1) many more convolutional and fully connected layers in their neural network structure, (2) extensive feature engineering, (3) 100 times more training data, (4) the hardware infrastructure which made the use of the complicated network structure and much more training data possible, (5) team size and the subject expertise that DeepMind has on this project.

It should be noted, my prediction of the first ~10 moves has ~40% accuracy. As the game progresses along, the prediction accuracy drops., The rate at which accuracy declines could be significantly lessened by training with more training data. This is illustrated in **Figure 3.**

Using only the first 10 moves of the games reduces training data size by 20 times and subsequently allows for speedy experiments while also giving very good performance. Therefore, for the rest of the report I have focused my effort on predicting the first 10 moves.

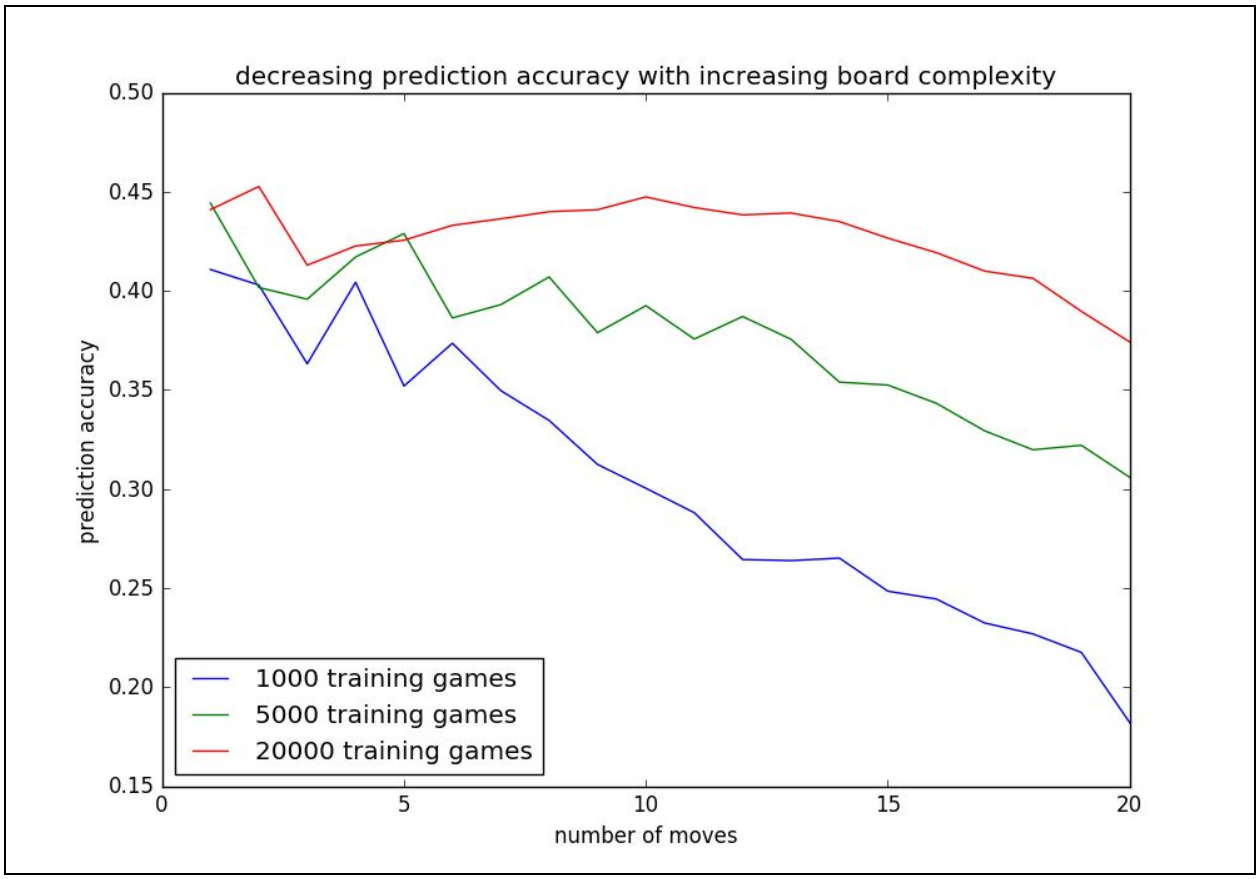

Figure 3: Prediction Accuracy drops with more moves into the game

3.2 Basic neural network model

Figure 4 illustrates the three layer neural network. There are 361 input nodes corresponding to 361 positions on the Go board, and 361 output nodes corresponding to 361 possible moves to make at any given time. As some of the output nodes represent illegal moves, a probability value of 0 should be theoretically assigned to these nodes as output. Work has been done to show that deep neural network is able to learn the game rules to close to 100% accuracy

without the rules being specified [1], therefore I did not hardcode rules for either legal or illegal moves. However, if time allowed, hardcoding the game rulebook would have helped to significantly reduce the complexity of the network. The activation function used is Rectifier Linear Unit, and output is in a softmax layer, which makes the output equivalent to probability distribution of all possible moves over the board.

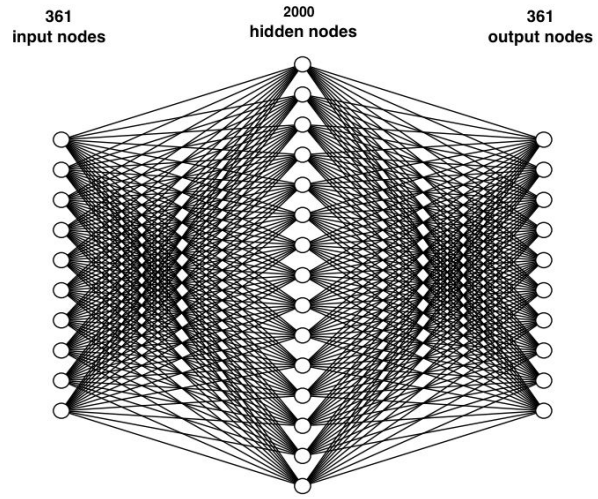

Figure 3 Three Layer neural network without convolution

3.2.1 Optimizing number of hidden units in three-layer neural network

By varying the number of hidden units from 100 to 6000, I trained different neural network models with varying number of hidden units and plotted each model's training and testing accuracy, shown in **Figure 5.** It can be seen that testing accuracy initially increases with more hidden units and plateaus around 2000 hidden units. Meanwhile, training accuracy increases continually with more hidden units. This result confirms the knowledge that more hidden units improves a network's ability to learn while too many hidden units promotes overfitting of the training data without increasing test accuracy. In light of these findings, I used 2000 hidden units for all the subsequent analysis.

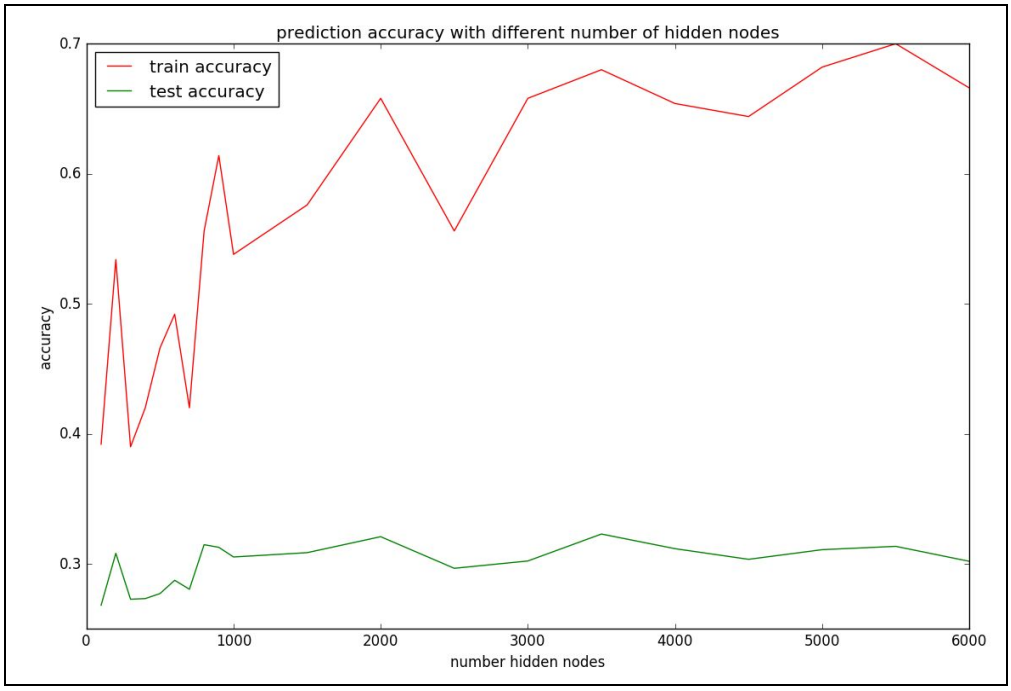

Figure 5. Impact of different number of hidden nodes on training and test accuracy

3.2.2 Trade off between test accuracy and training time

While it would be ideal to use all training data, using 80,000 games turned out to be extremely slow, even though I have access to 8 GPUs (each with ~2000 cores) from another class that I am taking (High throughput computing). In order to find a good trade off between a big training data size and a reasonable training time, I did the following scaling study:
I increased the number of training data from 1,000 games to 50,000 games while monitoring the training accuracy, test accuracy and epoch time. The result is shown in **Figure 6.** The left Y axis shows the training and test accuracy while the right Y axis shows the epoch time in seconds. Epoch time increase linearly with training size. This is expected as each epoch combs through all training data and more data requires more time to process it. Training accuracy drops with more training data because more data reduces overfitting. Testing accuracy increases with more training data because with more data, the model “learns more”. I chose to use 10,000 games for subsequent studies for the relative high test accuracy and relative low epoch time.

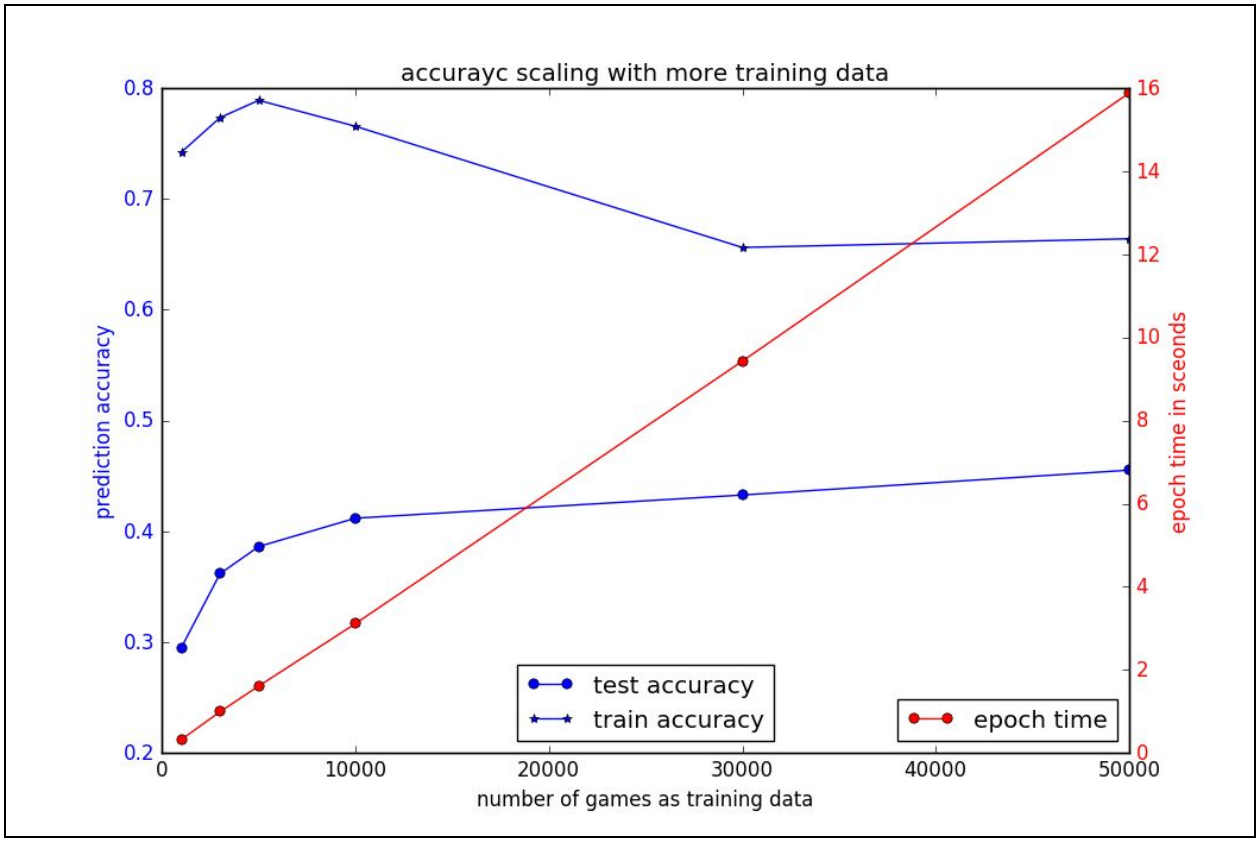

Figure 6: Scaling of training/test accuracy and epoch time with more training data

3.2.3 Regularization by dropout

I varied dropout rate from 0 (no dropout) to 0.8, and trained the model until test accuracy plateaued (defined by when the best test accuracy hasn't been updated for the last 10 epochs). I plotted the training accuracy, test accuracy and total training time in **Figure 7.**

Referencing **Figure 7**, it is apparent that overfitting is greatly reduced by dropout and test accuracy is moderately improved by dropout, while training time increases a lot with higher dropout rate. An optimal dropout rate for higher test accuracy and lower training time is observed around 0.5.

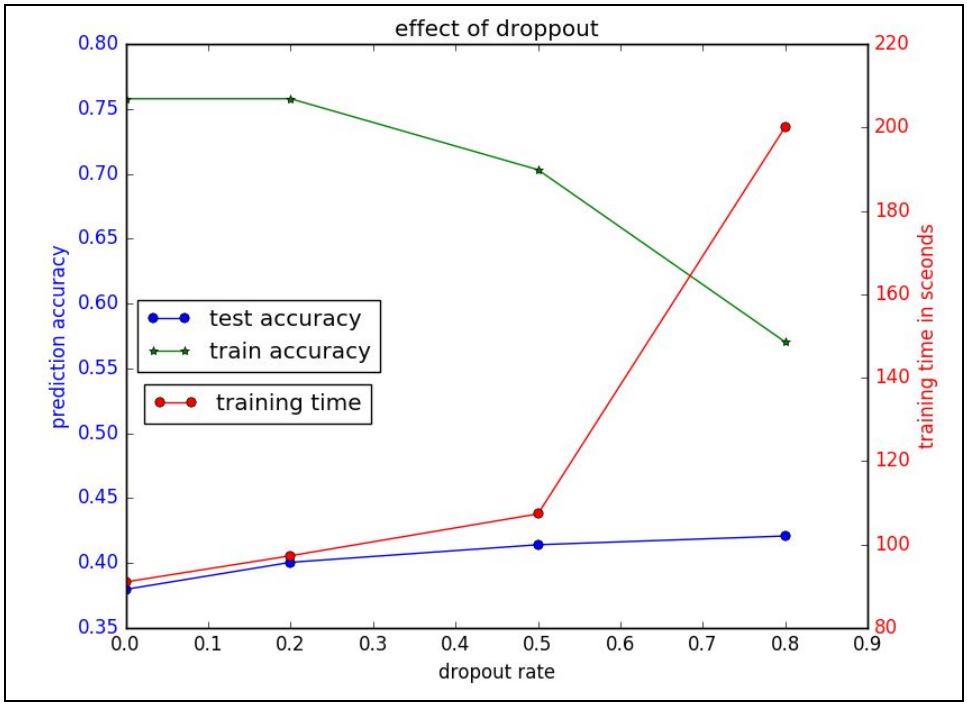

Figure 7: Scaling of training/test accuracy and epoch time with more training data

3.3 Convolutional network model

Figure 8 illustrates the second convolutional net model. Compared to the previous model, there is one convolutional layer after the input followed by a 2x2 max pooling layer. The input layer is soft padded which is why each convolutional layer is the same size as the input layer, and 20 features of 19x19 are generated. The rest of the network is the same with 2000 fully connected layer, ReLu is used as activation function and final output is softmax. Overall result of convolutional network model reaches 43% accuracy, about the same as non-convolutional model, and training time for convolutional model is about twice that of non-convolutional model.

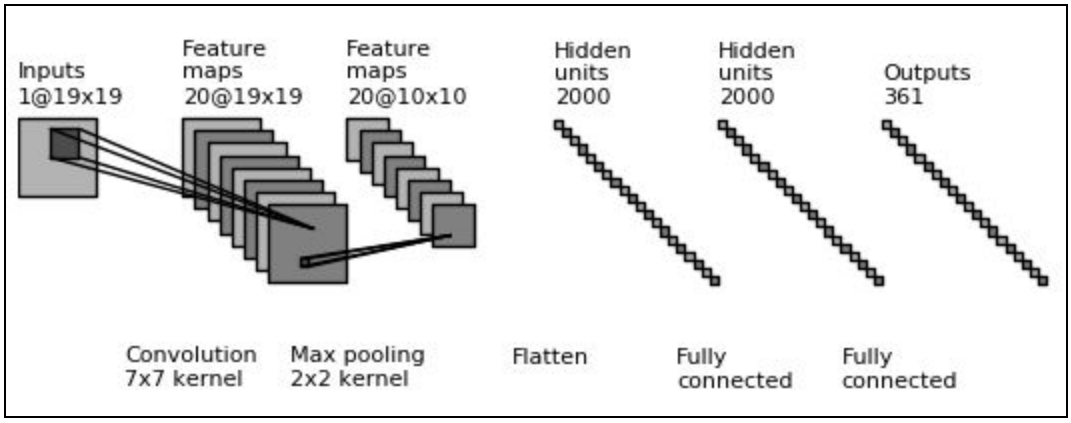

Figure 8: Convolutional Neural Network model

I expected convolution to improve prediction accuracy because: (1) AlphaGo's neural network model has many convolutional layers after input and (2) there are numerous local fights and tricks in the game of Go that involves recognition of local patterns, and regional pattern recognition is what convolution supposed to do. The reason for the lack of improvement with the addition of one convolutional layer could be that my predictions are done for the first 10 moves. The first 10 moves of a Go game are distinct from later moves in that "global" moves are usually played. The amount of available space on the board as well as the distance between stones means that players are not yet engaging in local fights. A better experiment can be performed on prediction of move 100 to move 110, however I have not yet had the chance the work on that before the writing of this report.

Nevertheless, I experimented with having max pooling layer or not, as well as different patch sizes in the convolution, as described in the following two subsections.

3.3.1 Comparing effect of pooling or no pooling

In normal image recognition, Max Pooling layer takes a 2x2 square in the previous layer and only output the largest number in that 2x2 matrix. This hypothetically preserves the predominate feature or pattern in that 2x2 region, reduces the model complexity by a factor of 4, as well as reduces noise and overfitting. However, this is where Go and image recognition differ. In the game of Go, the exact location of every stone matters tremendously, and moving a stone one "pixel" away could change the result of a game dramatically, therefore I suspect that a Pooling layer might not be useful in Go, and it would even do harm to prediction accuracy. I therefore performed experiment to compare the prediction accuracy in two convolutional networks with or without pooling layer, and the result is shown in **Figure 9.** The result shows that pooling or no pooling makes no difference to test accuracy or train accuracy. Again this comparison can be improved if the dataset is chosen to be moves in the mid/late game.

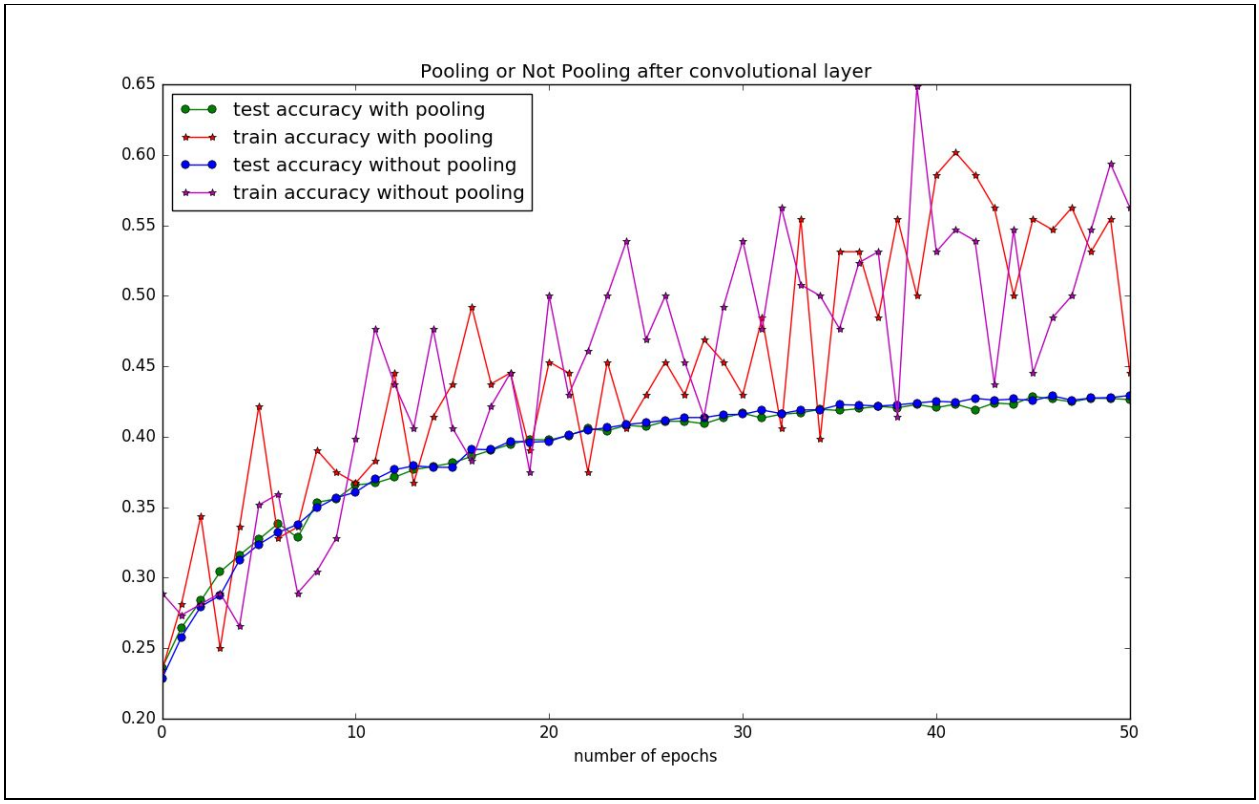

Figure 9. Comparing convolutional model with or without pooling layer

3.3.2 Finding the optimal patch size for convolution

Figure 10 shows the result of varying convolutional patch size and the corresponding training or testing accuracy. Size 7 is the patch size where test accuracy peaks, and it matches the choice of previous papers [1,2,3,4] and Go game playing intuition because when two stones are 3, or 4 stones away from each other then they don't form a real threat, therefore 7 patch size highlights the area of threat to the center stone.

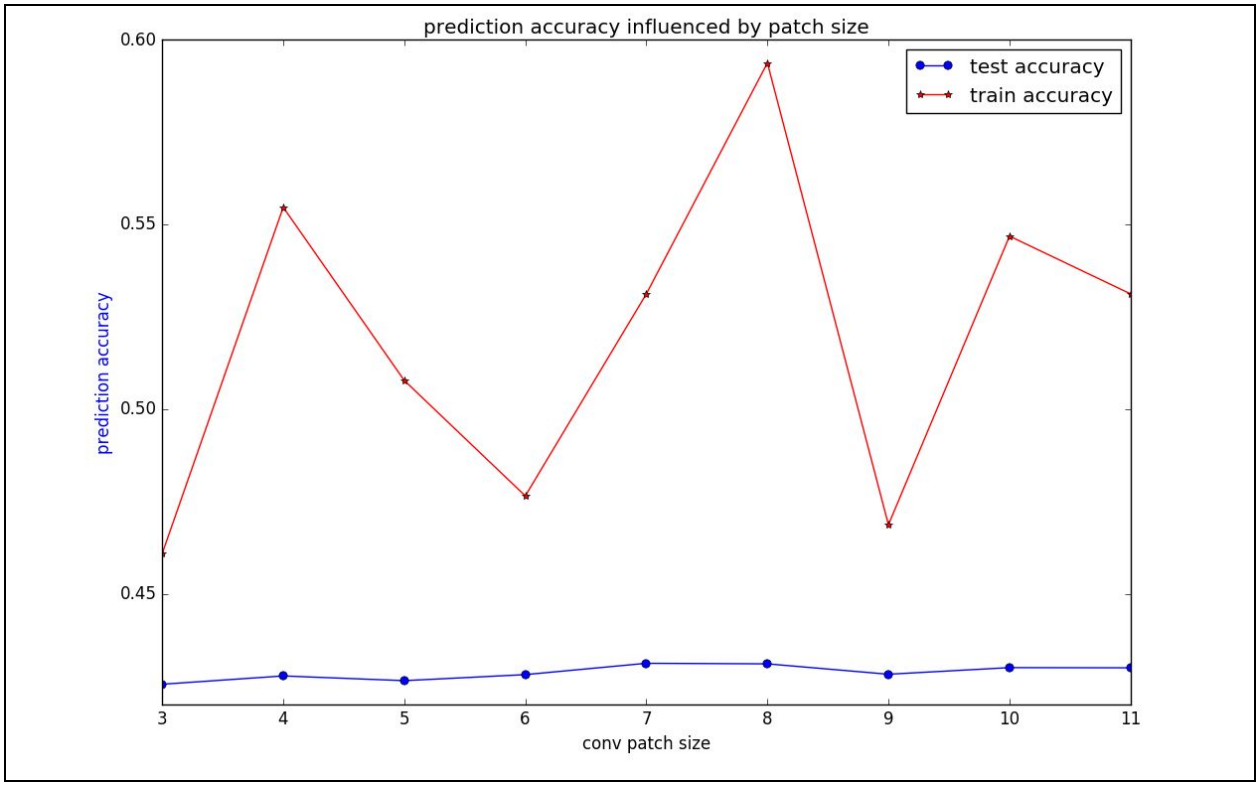

Figure 10. Prediction accuracy with varying convolution patch size

Part Four: Visualization of Prediction in Go board

In this part I am showing two visualization of the probability distribution of the predictions made by the basic neural network models described in section 3.2.

For this visualization, I take the probability distribution for each position on the board. This is visualized as a 19x19 matrix with element values ranging from 0 to 1 and the sum of all element values equal to 1. Because of the highly skewed nature of probabilities, I used this matrix to generate log probability matrix. I plotted the log probability matrix such that the the transparency of a node on a 19x19 grid is inversely proportional to the probability that the next move will be at that position (white: unlikely; dark purple: likely).

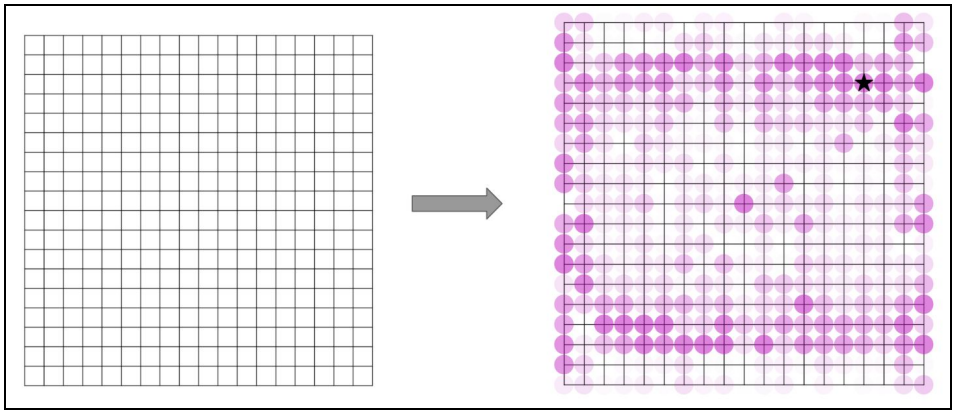

Figure 11. Predicted of moves for empty board, Left: empty board, Right: probability distribution of the next move, the darker the purple, the more likely is the next move to be placed at that position. Star shows the location of the actual move.

Interestingly, the neural network learns the symmetric property of an empty board (**Figure 11**), showing that it is equivalent to place the first stone in any of the four corners. The learning is not perfect, as we can see that the symmetry is not absolute. Also the borders of the board have a relatively darker purple, while moving the first move at the borders are considered very bad moves that nobody ever takes. However, bear in mind that the color is log probability, meaning if a circle is a shade lighter, it has 10 times less probability than a purple which is slightly darker. In fact most of the probability on board is at the order of 10^{-5} with a few being the exception, therefore the coloring on board can not be taken literally as a linearly representation of the probability of the prediction of the next move. Similar things can be said about the move in **Figure 12**.

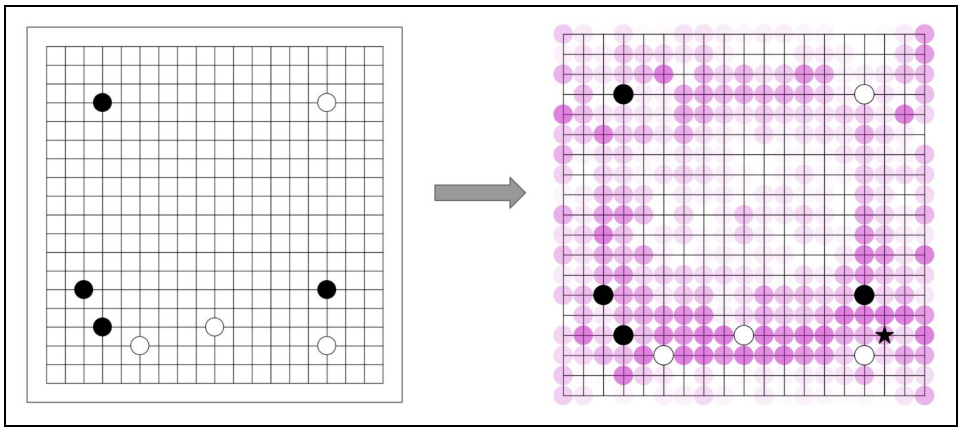

Figure 12. Predicted of moves for a board with 8 stones, Left: board position, note it's black stone's turn to move, Right: probability distribution of the next move, the darker

the purple, the more likely is the next move to be placed at that position. Star shows the location of the actual move.
Visualizations could have potential to be turned into a Go software to help players decide what's the good moves to take at a given game.

Appendix: Links, Tools and Libraries

The code for this project can be found at https://github.com/MollyZhang/AlphaGoPolicyNet

I used the deep learning library TensorFlow and Theano, I have also used Hyades, the UCSC supercomputing cluster from physics department. Various scripts I used are adapted from Tensorflow tutorial and Michael A. Nielsen's online book [7], I also used scripts found from stackoverflow to draw the go board [8], neural network architecture[9], and convolutional neural network architecture [10].

References

[1] van der Werf, Erik Cornelis Diederik. *AI techniques for the game of Go*. UPM, Universitaire Pers Maastricht, 2005.
[2] Sutskever, Ilya, and Vinod Nair. "Mimicking Go experts with convolutional neural networks." *Artificial Neural Networks-ICANN 2008*. Springer Berlin Heidelberg, 2008. 101-110.
[3] Clark, Christopher, and Amos Storkey. "Teaching deep convolutional neural networks to play go." *arXiv preprint arXiv:1412.3409* (2014).
[4] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529.7587 (2016): 484-489.
[5] https://en.wikipedia.org/wiki/Go_(game)
[6] https://en.wikipedia.org/wiki/Smart_Game_Format
[7] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015
[8] http://stackoverflow.com/questions/24563513/drawing-a-go-board-with-matplotlib
[9] https://gist.github.com/craffel/2d727968c3aaebd10359
[10] https://github.com/gwding/draw_convnet