Convolutional and Recurrent Neural Network for Gomoku

Rongxiao Zhang
Stanford University
Stanford, CA
Rzhang4@stanford.edu

Abstract

This paper proposes to use neural networks to solve a simplified version of Gomoku. More specifically, convolutional neural network and multi-dimensional recurrent network are trained separately using high-level human games. While the convolutional neural network was able to learn specific features for Gomoku without prior knowledge, the human game data set proved to be insufficient for a robust performance. The multi-dimensional recurrent network also suffered from the small database even though it was able to occasionally occasionally expert level moves.

1. Introduction

Games are an interesting domain for artificial intelligence research. They provide a controlled and well-defined environment, generally straightforward rules and clear-cut results. However, game winning formulas are often complicated and nonsingular. These characteristics make games suitable to test out different artificial intelligence approaches.

Deep neural networks involving various specific implementations have made a huge progress in the field of image processing. Convolutional neural networks have proven to be successful in feature recognition while recurrent neural networks have demonstrated their ability in handling sequences of data. Such implementations have been used on game playing. Most notably, Google developed the Go-playing program AlphaGo using deep neural networks[1]. Go has long been regarded as the most complicated chess game with a prohibitively large search space. In fact, many aspects of the game have been proven to be NP-hard and PSPACE-hard. However, AlphaGo has demonstrated its ability by winning 5-0 against the European Go champion as well as leading 3-1 against the second highest ranking professional Go player worldwide.

Although complex in nature, Go is a perfect candidate for evaluating neural network algorithms. The rules are very simple: two players play on a 19x19 board. Each player alternatively places stones onto any of the unoccupied intersections on the board in attempt to conquer the most territory. The player with black stones places first but need to capture a greater area than the player with white stones in order to win. A single stone or a group of connected stones can be captured when it is completely surrounded by its opponents' stones. The only illegal move is when it leads to previously seen position to avoid cycling. The game ends when both players pass. Winning is determined by counting territory captured by each player.

One interesting and important observation is Go's symmetry. It has not only the straightforward axes of symmetry, but also an approximate translational invariance.

Conducting experiments with Go can be very difficult due to the need for distinguishing dead group from alive ones and handling pass moves. Many machine-learning approaches have used simplified versions of Go as a test bed for ideas.

Common simplified Go games include Pente, Renju, Tanbo, Connect, Atari-Go and Gomoku. In this paper we will focus on Gomoku due to its simplicity.

Gomoku, also know as 'five-in-a-row', is played on a 15x15 board, smaller than Go's 19x19 board. The basic rules are the same, two players place stones alternatively on the intersections of the board. However, winning is determined by whoever gets 5 of their stones in a row, vertical, horizontal or diagonal. Human players strategies usually involve blocking opponents' lines, keeping his line unblocked, and perform multiple attacks at the same time. These are all heavily feature based.

These feature-based strategies provide a perfect environment to deploy neural network based techniques. Moreover, human players rely on past game experience to develop these strategies. This also plays in neural networks' advantage.

The fact that AlphaGo did not perform perfectly against top human player leaves room for development. This paper is an attempt at using two different techniques for context free game playing: convolutional neural networks for feature recognition without human input and multi-dimensional recurrent neural networks for scalable performance for different sized boards. The inputs to our algorithm are Gomoku board situations expressed as

15x15x1 matrices and the outputs are the optimal next move for the given player.

2. Related Work

A large number of artificial intelligence approaches have been proposed for board games. The most basic attempt includes game-tree searching which is often coupled with a board evaluation function as well as alpha-beta pruning. However, as William pointed out [2], a complete search to a depth of n moves require evaluations of p!/(p-n)! board situations, where p is the current number of legal moves. Although this approach has been used to develop acceptable Gomoku AIs, the scalability of such an algorithm is poor.

Reinforcement learning together with neural network has been proposed by Freisleben[3]. This network is penalized or rewarded using comparison learning, a special reinforcement learning algorithm. Temporal difference learning as well as adaptive dynamic programming approaches is also two algorithms that achieved some level of success for Gomoku. However, neither of them is suitable for larger scale games such as Go.

Zhao et al. improved the performance of the adaptive dynamic programming by pairing it with a three layer fully-connected neural network to provide adaptive and self-teaching behaviors[3]. However, the inputs to the neural network are consisted of 40 pre-determined patterns. The neural network then uses these layers to evaluate the board situation before deciding the next move. Such a network is only effective for games with available expert knowledge.

Another interesting approach from Schaul and Schmidhuber [4] uses multidimensional recurrent neural networks (MDRNNs). In their attempt, four units sweep from four diagonal directions to obtain a heuristic view of the board using recurrent neural networks. The training process for this model involves evolution and doesn't use any expert knowledge.

Our focus is inspired by the last two approaches: the convolutional neural network as well as the recurrent neural network approach. We propose improvements to both structures. For the fully connected neural network approach, instead of using pre-selected features, we add in convolutional layers in front to extract features without expert knowledge. And with the recurrent neural network approach we use professional Gomoku games to train the network instead of using an evolution process.

3. Methods

3.1 Convolutional Neural Network Architecture

In this section we propose a possible improvement over Zhao et. al.'s adaptive dynamic programming with neural network approach[3]. One particular strength of convolutional neural networks is the ability to detect features. They perform well not only in identifying simple features such as edges but also in combining these simple features to understand more complex characteristics of the input image.

Zhao et. al. used 40 features as inputs to their neural network architecture. These features, as identified by experienced players, indicate varying possibilities of victory or defeat. However, such information is not transferrable once the rules are modified even slightly or if the scale of the game is varied. We believe that such feature can be learned through convolutional neural network architecture with a large number of games.

Our architecture is comprised of 3 convolutional layers followed by 3 fully connected layers. The input data consists of recorded games at high-level Gomoku tournaments[6]. Since a winning move in Gomoku forms a pattern of 5 stones in a row, we would like to capture features that are less than 5 consecutive stones. A large number (1024) of 5x5 filters are used for the first convolutional layer. Smaller filters of 3x3 are used for the following 2 convolutional layers in order to provide relationship information for the captured features.

Each specific board incident can be viewed as a 15x15x1 image. Filters for the first convolutional layer have the dimensions of 5x5x1. With zero padding of dimension 2 and a stride of 1, we preserve the input image dimension of 15x15. Since we have 1024 filters, the output of this step is 15x15x1024.

To avoid duplicate or incomplete features, max pooling layers are inserted after the first convolutional network. Due to the size of the convolutional layer output we use a 3x3 max-pooling filter with a stride of 3. This will lead to an output of 5x5x1024. The second and third convolutional layers have 256 and 128 filters respectively and the output from convolutional layers has the dimension 5x5x128.

After the convolutional and pooling layers, the inputs to the fully connected network are the activations for recognized patterns. 3 fully connected nets are then used to evaluate board situations. The output of these fully connected nets are feed into a Softmax classifier in order to select the next move. Similar to image categorization tasks, the Softmax classifier interprets the output from the fully connected layer as the unnormalized log probability for each specific move. The loss function associated with the Softmax function is as follows:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

Illegal moves, such as board positions that are already

occupied, are ignored should it be selected as the next move. Gradient descent using the derivative from the loss function is then back propagated through out the network to update the weights. All weights are initialized to be very small random numbers in order to break symmetry.

3.2 Recurrent Neural Network Architecture

We evaluate the performance of a recurrent neural network using similar implementation from Schaul and Schmidhuber. However, instead of training by evolution, we use the same dataset as we used for the convolutional neural network architecture just like training for an image classification test.

The recurrent neural network is constructed using 4 units that sweep through the board in four diagonal directions. Standard recurrent neural networks are inherently one-dimensional with abilities to handle sequences in the time dimension. In our case, two spatial dimensions replace the single time dimension in order to accommodate the 2D nature of the Gomoku board.

Each unit takes into account 3 inputs: one input comes from the board indicating whether a stone exists at the current location; the second and third input comes from its own output at previous locations. For example, consider a unit u_1 that sweeps from the top left corner to the bottom right corner. At each position (i,j) it receives an input from the board position at (i,j) as well as a position from its own output at $u_{1(i,\,j-1)}$ and at $u_{1(i-1,j)}$. It then process the inputs and produces an output $u_{1(i,j)}$.

Using a recurrent unit this way captures information from the top left corner up to its current location. If we use four units coming in from all four diagonal directions, we can obtain information for the entire board.

Mathematically, we have:

$$u_{\searrow(i,j)} = \tanh[w_i * in_{i,j} + w_h * u_{\searrow(i-1,j)} + w_h * u_{\searrow(i,j-1)}]$$

(and analogous for the other directions)

$$out_{i,j} = \sum_{\diamondsuit \in D} w_o * u_{\diamondsuit(i,j)}$$

where ↘ indicates the unit sweeping from top left corner to bottom right corner and D = {↘↙↖↗}.

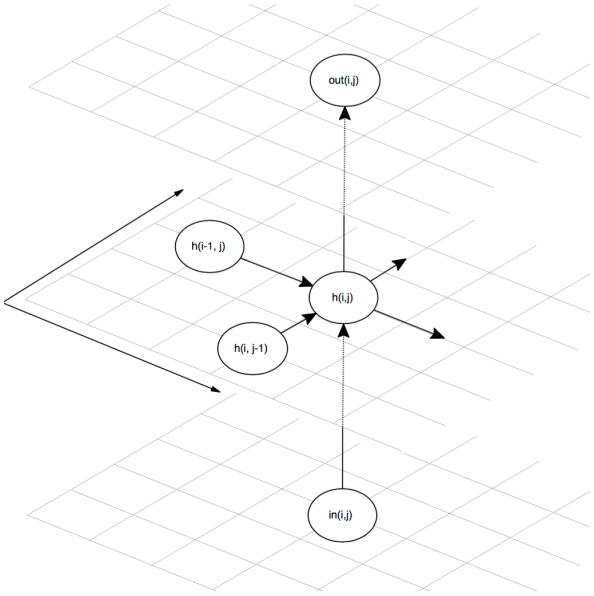

Figure 1. Illustration of the MDRNN Network

The four units will process the board independent of each other. Each will have an output of 15x15x1xk where k is the number of hidden units. The four outputs will then be consolidated using w_o into a 15x15x1 output indicating the probability of the next best move at each board location.

On the boundaries where the unit values are not defined, we initialize them to a fixed value w_b of dimension 1xk. As a result, we have k^2 parameters from w_h, 2k parameters from w_i, k parameters from both w_o ad w_b. This adds up to $(4k+k^2)$ parameters. With a k value of 10 (in accordance with Schaul's paper), we end up with 140 parameters, significantly less than the number of weights for other neural network structures.

The input to the network is of dimension 15x15x2. At each location two inputs are used to specify the presence of a stone. The first one is 1 if a stone of the same color is present and 0 otherwise. The second input is 1 if a stone of the opposite color is present and 0 otherwise.

At each given location, the output expresses the network's preference for playing at this certain location. A move is then chosen using the Softmax classifier as in the convolutional neural network's case with illegal moves ignored. Figure 2 illustrates the network's output given a specific board configuration. Although Gomoku board is 15x15 this network is scalable in a sense that it can be trained on a smaller board and the training results can be generalized to a bigger board.

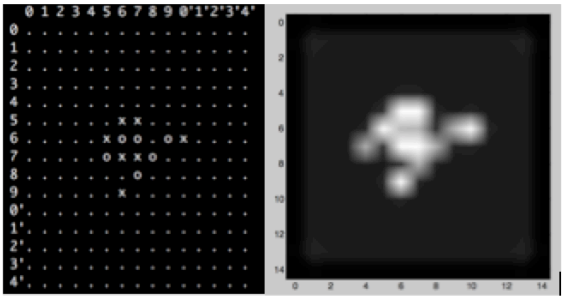

Fig.2 Left: board given as input. Right: output of the network

4. Dataset and Features

The training dataset is obtained from RenjuNet database that is an online database for Gomoku games. The dataset contains 42955 games with more than a million total moves. The data set is split up so that each board configuration is treated as a 15x15 image with the 'ground truth' being the next move in that specific game. This turns the Gomoku game-playing problem into an image classification problem. Given a specific board configuration, the neural network architecture tries to find the most 'logical' next move.

One potential problem is that some moves in the dataset might not be the 'best' move given they are human games and human do make mistakes. But since these games are high level games, we believe even if a move is not the best possible move, it is still a good enough move that a human expert player is likely to play.

Preprocessing involves picking out moves from the game database and mapping them onto 15x15 matrices. Ground truth labels are stored in a separate array matching the indices for its associated board configuration.

5. Experiments

A custom made Gomoku engine was implemented in Python. For evaluation purposes, a basic AI using tree search and alpha-beta pruning was implemented. The level of play for the tree search AI is at approximately human amateur level for a 3-ply search.

The evaluation metrics are different for the two architectures we propose. While both played against the naïve tree-search AI, the convolutional neural network is also evaluated by its ability to capture human recognizable winning patterns while the recurrent neural network is evaluated by its ability to play on different sized boards.

Both networks are custom coded in python without using wrappers for educational purposes. This means the models are neither extremely deep nor complicated. But since Gomoku is a relatively simple game with elementary features, a less complicated network should be robust enough. Moreover, as there are only 140 parameters for the MDRNN, training is not computation intensive.

5.1. Training the convolutional neural network

The dataset is grouped into training set, evaluation set and test set using a 8:1:1 split. Adam update rule is used to help with convergence rate. Recommended values of beta1 = 0.9, beta2 = 0.999 and eps = 1e-8 are used as default. Hyper parameters are selected using smaller datasets of 1000 moves. The hyper parameters are shown in table 1.

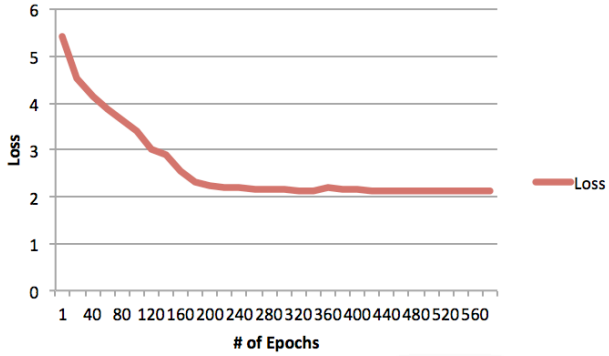

Fig. 3 Loss history of convolutional neural network training

Regularization	0.0001
Dropout	0.8
Learning rate	3.00E-03
Learning rate decay	0.95
Update Rule	Adam
Batch Size	200
Number of Epochs	600

Table 1. Hyper parameters for training the convolutional neural network architecture

5.2. Training the multi-dimensional recurrent network

With similar grouping of the dataset, the recurrent network is configured with 10 hidden units to be in accordance with Schaul's results. However, the initial weights were drawing from a normal distribution of N(0,0.001) instead of N(0,1) as is used in Schaul's paper. This is because our training method is different from Schaul's evolution method and a big initial weight will cause numerical instabilities during the training process.

Training the network on small sets of data determines its hyper parameters (Table 2)

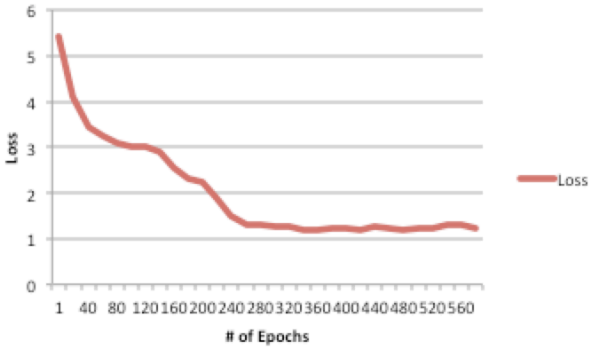

Fig.4 Loss history for training the MDRNN

Regularization	0.001
Hidden Dimension	10
Learning rate	1.00E-04
Learning rate decay	0.99
Update Rule	Adam
Batch Size	200
Number of Epochs	600

Table 2. Hyper parameters for training the convolutional neural network architecture

Both models are then trained on a personal computer with a 2.3GHz Intel i7-3610QM CPU and 16GB of memory. The training time was about 40 hours for the convolutional network and 10 hours for the recurrent network.

6. Results

6.1. Convolutional neural network performance

One problem during training was that we were unable to reduce loss further and the validation accuracy was around 40%. A variety of different hyper parameters were tested out but we were unable to reduce loss further.

The resulting model played the tree search AI and won 0 out of 1000 games. A closer look at the game revealed that although the neural network model played good moves occasionally, it was unable to perform consistently due to the lack of board evaluation functionality and policy network evaluations. We were able to pick out some human recognizable filters from the first convolutional layers but it was inconsistent across the board (Figure 5).

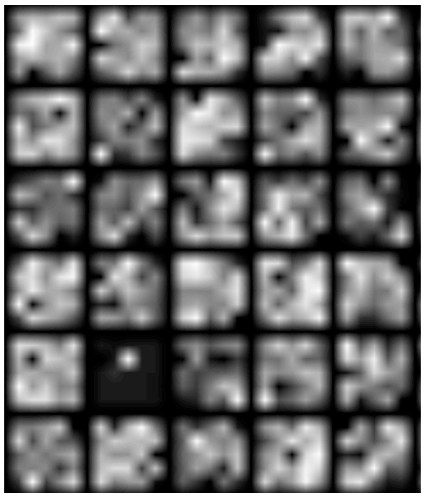

Fig.5 A selection of convolutional filters used in the convolutional neural network model.

6.2. MDRNN performance

The MDRNN performed better compared to the convolutional neural network. The validation accuracy was about 50% but it was also unable to pick out the correct moves consistently. A win rate of 3% against the tree-search AI was unconvincing. An example of the game is shown in figure 6. The MDRNN model (indicated by ‘o’) played a pointless move at (12,10) when the tree search AI had a clear winning move at (7,11).

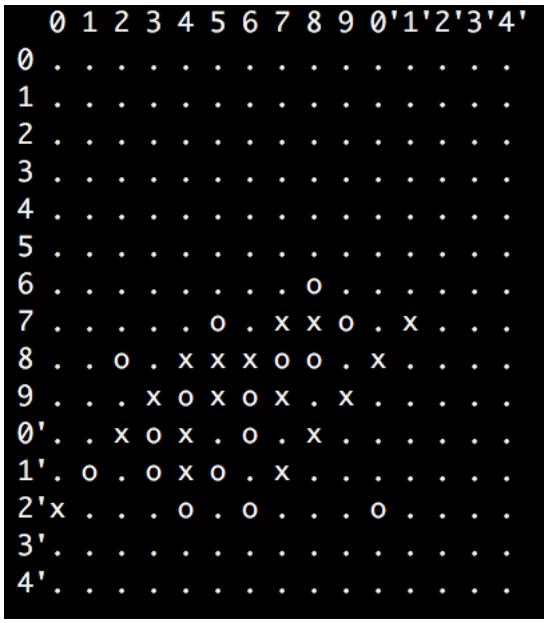

Fig.6 Example game between MDRNN and tree-search AI

The poor performance makes scalability test also unconvincing. We were able to obtain high performance correlations but that was a mere result of consistent poor performances.

7. Discussion

Neither of the two neural networks that we proposed worked as expected. While the convolutional layer captured some features through learning, many other features were merely noises. This makes sense in retrospect: for a simple game such as Gomoku, only a limited number of features are important to the progress of the game. Human players choose to neglect certain patterns when it is clear that no gain can be made from them (for example, a blocked pattern of 3-in-a-row). However, when evaluating using convolutional filters, these 'useless' features can still be recognized, contributing negatively to the board evaluation process.

Another problem lies within the dataset. Although we seemingly have more than a million board configurations, about a third to half of these moves consist of what is called 'starting moves'. These moves are often, if not always, played as a 'fixed' sequence at the start of the game. This significantly decreased the number of valid training data that we have. With a smaller dataset both the convolutional network as well as the MDRNN architecture are affected.

8. Conclusion and future work

The intention for this paper is to use expert human games to train potential human-like networks. However, neither of the proposed neural networks was successful. Some positive performance was seen: convolutional layer was able to pick up some of the expert human knowledge automatically while the MDRNN was very efficient by using a fraction of the parameters in deep neural nets.

Although one of the problems we discovered was with the size of the dataset, applying data augmentation can solve this problem. For example, since convolutional layers try to pick out specific features, translating the board configuration should not change the potential move. Due to board symmetry, we can also flip the board configuration along any of its symmetry axes. This will provide a much larger dataset without accumulating more games.

On the other hand, if developing a stronger AI for a certain board game is the ultimate goal, self-playing is still a powerful technique as proven by AlphaGo. Instead of using coevolution or human game database, the MDRNN model can also be trained to play against itself. The scalability of the model also enables it to play on different sized boards. Together with its efficiency, the MDRNN model might be able to learn faster than existing algorithms.

References

[1] Silver, David, et al. Mastering the game of Go with deep neural networks and tree search, Nature 529.7587: 484-489, 2016

[2] T.K. William, S. Pham, Experience-based learning experiments using Gomoku, in: Proceedings of IEEE International Conference on Systems, Man, and Cybernetics, Charlottesville, Virginia, USA, vol. 2, October 13–16, 1991, pp. 1405–1410.

[3] B. Freisleben, A neural network that learns to play five-in-a-row, in: Second New Zealand International Two-Stream Conference on Artificial Neural Net- works and Expert Systems, 1995, pp. 87–90.

[4] Schaul, Tom, and Jürgen Schmidhuber. A scalable neural network architecture for board games. Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On. IEEE, 2008.

[5] Zhao, D., Zhang, Z., & Dai, Y. Self-teaching adaptive dynamic programming for Gomoku, 2012, Neurocomputing, *78*(1), 23-29.

[6] http://renju.net/downloads/downloads.php, 2015

[7] G. Tesauro, Neurogammon: a neural-network backgammon program, in: Proceedings of International Joint Conference Neural Networks, San Diego, California, USA, June 17–21, 1990, pp. 33–40.

[8] J. Baxter, A. Tridgell, L. Weaver, Learning to play chess using temporal differences, Mach. Learn. 40 (2000) 243–263.