5-2019

Building an Artificial Intelligence to Learn Go

Nathan Skalka
nskalka@unomaha.edu

Building an Artificial Intelligence to Learn Go

Honors Project

Nathan Skalka
Mentor/Supervisor: Dr. Prithviraj (Raj) Dasgupta

Computer Science Department
University of Nebraska at Omaha
29 April 2019

1 Introduction

Go is a 2-player board game played competitively on a 19x19 grid board. Each player takes turns placing a black or white piece, respective to the player, in an empty intersection until the game ends. This goal state can occur if the board is filled or both players agree that no more pieces need to be placed in order to determine the end score of the game. The end score is determined by counting the empty spaces within territories which is defined as the enclosed space of a player's pieces and by subtracting the number of pieces captured by the opposing player surrounding the player's pieces, creating a territory. Capturing occurs when one player's pieces are surrounded on all outside adjacent sides. Figure 1a and 1b describes a few different territories while Figure 1c exemplifies the result of white capturing an opponent's black pieces, creating a territory.

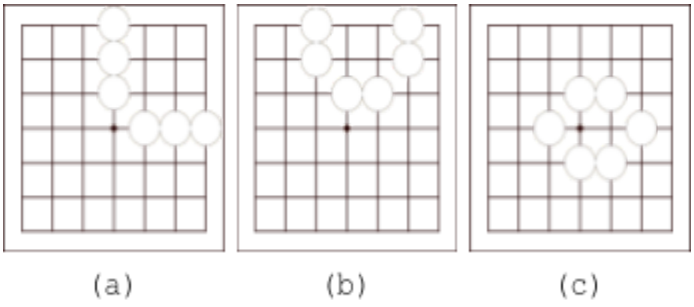


Figure 1. Three examples of territories on a 7x7 grid. 1c is also the end result of capturing exactly two pieces inside of the territory.

The author's interest in this research stems from an interest in artificial intelligence driven game playing. In order to better research the core topic, there were a few in-depth concepts like a game trees, game state space, and reinforcement learning which needed to be covered prior. First, an explanation of the game tree utilized in order to initially solve the challenge of playing Go. Given a specific state of the board, there are a number of possible actions which branch from the current state to a new state as exemplified in transition of Figure

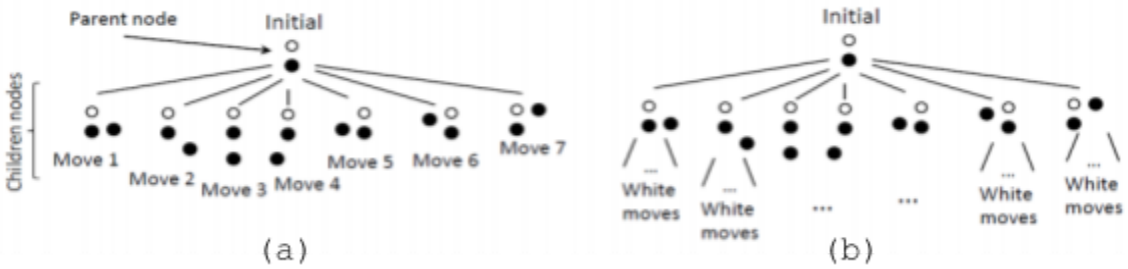


Figure 2. Examples of the branches of a game tree exploring one turn of the game in (a) and further exploration surmised in (b).

2a to 2b. Repeating this branching behavior out to a final game state, this branching can recurse deep into a tree structure given a 19x19 board and approximately 150 to 400 turns in a game. Complexity on the magnitude of 10^{360} due to the high count of nodes in the tree, and too complex

for some of the fastest computers with a large memory space. (Brown et al., 2012) To note, this level of complexity out matches that of chess which is in the magnitude of 10^{123}. Thus, solving the problem of this research required much more recent developments in the artificial intelligence field in order to tackle the higher difficulty of the problem.

As much of artificial intelligence starts out, an algorithm of relative newness to the field called Monte-Carlo Tree Search (MCTS) is used in order to explore the state space of Go and determine the appropriate action. However, due to the extremely large state space introduced by the simple game of Go and the size of a 19 by 19 board, measures were taken in the author's effort to reduce the board size to 9 by 9 in order to observe the results of an MCTS algorithm without requiring the large computation power necessary in order to process the state space. By studying and implementing the process for playing Go with MCTS, the author better understands the benefits of automating the algorithm into a learning process through reinforcement learning. Some steps were skipped in the process due to previous research completed by DeepMind in the field of reinforcement learning (supervised and unsupervised), bridging the gaps between the MCTS process, a supervised learning model, and an unsupervised learning model.

2 Related Work

Past works on the topic of MCTS-oriented solutions include efforts to produce an efficient, time-sensitive program called Pachi. Pachi is an open source project which aims to be a state-of-the-art solution written in C++. "Petr Baudis maintains a Pachi instance running with 8 threads on Intel i7 920 with hyperthreading enabled and 6 GiB of RAM available that plays as users Pachi, PachIV and PachIW. These instances can hold a solid 1-dan rank" (Baudiš & Gailly, 2012). This rank of 1-dan, or 2200 Elo rating, is objectively what will be used in comparison of the various implementations this project is attempting to achieve. The Pachi program utilizes an enhanced MCTS algorithm in order to calculate the appropriate action without simulating the entire game tree. Relevant implementation details of Pachi is the incorporation of each intersecting tile containing data like the intersection color, counts of immediate neighbor colors, group identification based on the group-founding piece's position. (Baudiš & Gailly, 2012) Baudiš acknowledges that a pure MCTS algorithm is not stable enough to play in highly disadvantaged or advantaged situations. Because of this, Pachi further emphasized the ability to optimally compute by incorporating domain specific knowledge pro-players utilize in order to prioritize what areas of the game tree are explored first. (Baudiš & Gailly, 2012) Common 3x3 patterns are used like the one in Figure 3 to describe a locally-scoped scenario and the appropriate play for a given player. In the figure, various forms of the "Hane" pattern are shown with a variety of possibilities for either player to make a move, but the far right option shows the

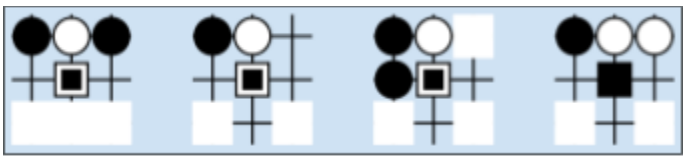


Figure 3. Examples of various Hane patterns.

beneficial option is only for the black-pieced player. These patterns are the driving force behind the solution that is Pachi; however, this requires human observation which lacks the powerful insight of machine learning.

In the past three years, reports published by DeepMind tackled the challenge of utilizing machine learning to improve upon the design of Pachi. It does so by learning the patterns Pachi was hardwired to find from human experience. This leverages the insights a computer can make on a variety of patterns by looking at a board state, and understanding more optimal plays from past experience. At first, DeepMind research focused on the observation and learning from recorded games of professional players, commonly referred to as supervised learning due to the large sample set the program was given to learn. This is referred to as learning through replay where the program learns by only observing past positions. (Silver, Huan, Maddison, Guez, Sifre, Van Den Driessche, & Dieleman, 2016) The result was the program AlphaGo which successfully defeated Lee Sedol, one of the highest ranked 9-dan players at the time. (Silver et al., 2016) After this, further research was put forth in order to build upon the supervised learning of AlphaGo with a less supervised learning approach called reinforcement learning. Precisely, AlphaGo Zero (AGZ) was developed in order to learn from a blank slate, without any knowledge of high-level plays and knowing only the rules of the game. This is referred to in some literature as learning via “tabula rasa”, the latin for “blank slate”. (Silver, Schrittwieser, Simonyan, Antonoglou, Huang, Guez, & Chen, 2017) As an extension of AlphaGo, the transition focuses on learning while playing against itself. This is referred to as self-play, and AGZ starts by only playing random moves. It learns by playing against itself, observing which plays lead to a winning game, and adopting those plays to become better in future iterations. Repeating these steps is the key to building both a consistent as well as more fully “experienced” player.

```
virihaure 462% ./gnugo --mode gtp
1 boardsize 7
=1

2 clear_board
=2

3 play black D5
=3

4 genmove white
=4 C3

5 play black C3
?5 illegal move

6 play black E3
=6

7 showboard
=7
   A B C D E F G
 7 . . . . . . . 7
 6 . . . . . . . 6
 5 . . + X + . . 5
 4 . . . + . . . 4
 3 . . O . X . . 3
 2 . . . . . . . 2     WHITE (O) has captured 0 stones
 1 . . . . . . . 1     BLACK (X) has captured 0 stones
   A B C D E F G

8 quit
=8
```

Figure 4. Examples of commands and responses being sent via gtp. The numbers before commands and after responses are not required. (The Go Text Protocol, 2009)

3 Implementation

3.1 Background Details

Two implementations of a Go playing engine were created for this project, a MCTS-oriented process and a reinforcement learning (unsupervised) process. Both were developed in the Python programming language. A general board with state and

actions related to the game like placing pieces, capturing pieces, calculating liberties, and scoring the board was created for use by both implementations. Finally, an interface for interaction between the various artificial intelligences was chosen to allow the competing AI's to interact in an automated way.

GoGui is a free, packaged suite of programs for playing Go and interacting with Go programs. Within GoGui is a tool called GoGui-TwoGtp which allows two Go Text Protocol (GTP)-enabled processes to receive and respond to Go related actions like determining the next move of a player or playing a move to update the internal model of the GTP-enabled programs. As the name implies, GTP is a text based protocol running on the standard input and output of a process. As exemplified by Figure 4, commands are sent in the common form of command then space separated arguments while a response is given by returning a validation character of "=" for okay or "?" for a failure. Following the response character, an expected value when succeeding or an error message when failing is sent on the same line. In other aspects of computer science, this form of communication is likened to GET or POST web endpoints.

3.2 Monte-Carlo Tree Search and Adaptive Policy Playouts

As MCTS solutions can be viable at a small state-space complexity like a 5x5 board, the initial proposed implementation was to be a 5x5 board. However, this became difficult to test due to Pachi running at a minimum of a 7x7-sized board, leaving an incomplete subset of the board unaccounted for within the game tree. Therefore, the final implementation set the size for the rest of the project to 9x9 which is also the common, smallest board size for teaching moves to beginners. This board size is still computationally possible for the less powerful computers with the use of Adaptive Policy Playouts (APP) to prioritize said computation towards generally well-recognized high-level play patterns. This tactic is used to optimize Pachi's efficiency in determining moves. Similarly, this tactic is being utilized for the lessen computational load just to produce a strong set of moves in a short amount of time, but not potentially the one best move.

The research-driven, python implementation utilizes many of the same features as the Pachi program, namely the focus on prioritizing which branches of the game tree to explore via APP. This utilization of APP mentioned above requires an analysis step of the current board state in 3x3 patterns before exploration begins. While this step does add complexity to the computation of the whole tree were it to be explored, the analysis is giving efficiency in creating a thin tree with few, high potential branches to be searched. There is also a secondary level of limit given in the readout count for exploration so the MCTS did not take up a large portion of time in order to calculate one move, giving a more responsive behavior of at most 10 seconds.

3.3 Unsupervised Learning with Tensorflow

MiniGo is a project focusing on being an open-source implementation based on the AGZ paper. It incorporates many different aspects with the intention of being highly readable code. As mentioned in the core readme, another primary goal of the project is to exemplify the use of the Tensorflow framework, reinforcement learning, and docker as a OS abstraction. While some of these targets align with the objectives of this project, not all parts were translated over for the purposes of this project.

Tensorflow is an open-source machine learning library aiding in the development of reinforcement learning. (Tensorflow, n.d.) More commonly seen applications involving machine learning are image classification, handwriting text recognition, and language translation. Interestingly, a 9x9 grid of patterns falls into much of the same category as the above application of image classification. That is to say, Tensorflow is leveraged to teach an application to recognize patterns from examples and to learn where to use them from the success of initially random decisions. In testing and research, working with the dockerization was difficult to follow running Docker on Windows; however, for the the author's implementation, dockerizing the solution was not within the scope and was not pursued as running in a python virtualenv was the core necessary step in connecting the python code to the Tensorflow-gpu package and GPU drivers.

The utilization of Tensorflow was key in developing the reinforcement learning elements of this project. However, there were some core differences between the the AGZ paper, the minigo research, and the final implementation. First, as per the minigo project itself there is no virtual loss, not enough filters (128 instead of 256), and ambiguity around the MCTS output, pi, the policy network target. The author leveraged Tensorflow's Example class as the initial implementation unlike in the the minigo project. Minigo utilized a Dataset class to handle the training data prior to the evolution of the tfExample class. This allowed for some initial cutaway code to be much more cleanly developed to begin with, and gave the overall Tensorflow learning pipeline a more readable feel.

4 Results and Analysis

4.1 Monte-Carlo Tree Search

For the evaluation of the MCTS implementation with limited readouts, the implementation scored 1 win out of 25 against the Pachi program operating at approximately 1-dan (2200) Elo rating. Given these two values, a calculated score of approximately 1400-1648 Elo was assigned to the MCTS implementation. Relative to reinforcement learning, there is no

learning or improvement from playing more games. Only through further optimization, would there be an improvement in playing. Some optimizations come from the greater library of 3x3 patterns as well as incorporating more than just the 3x3 patterns with certain larger scale patterns like laddering. For reference, laddering occurs when one player attempts to guide the other player's singular line of pieces to an end on the board and captures them. Referring to Appendix I, the gtp output of a losing game between the Pachi and the MCTS implementation shows an ability to play. However, there is also a lack of ability to compete against higher level AI's and players due to static skill of the program.

4.2 Unsupervised Learning

Unlike the MCTS algorithm which lacks a natural ability to learn from what it plays, the unsupervised learning implementation does in fact learn. A version of the Tensorflow model used by the minigo application was used in order to test the running and integration of Tensorflow-gpu and CUDA. The result of testing the given model produced 8 wins out of 25 matches against Pachi, calculating out to an approximate 1750-2035 Elo score. However, there were two issues which appear not to match up with the AGZ paper this project was built on. First, it appears there is some amount of overfitting given a lower prediction accuracy than the paper's 85%. This is thought to be due to the training process not adequately shuffling the dataset in the learning step. Secondly, it appears that certain transformations of the same board produces variable resulting plays due to a variation in the application's value estimations. Lastly, the learning loop portion of the application was tested with the following Figure 5 which represents the learning over generations. However, this learning is at a much slower pace than the AGZ and recorded minigo learning due to both the hardware used and the less efficient learning in fewer samples per generation than both AGZ and minigo (125,000 each). Given Figure 5, there is much more within the learning loop to accomplish without much reduction ability to learn. This exemplifies a good learning process, but with more time needed to fully build the model rendered in either AGZ or minigo.

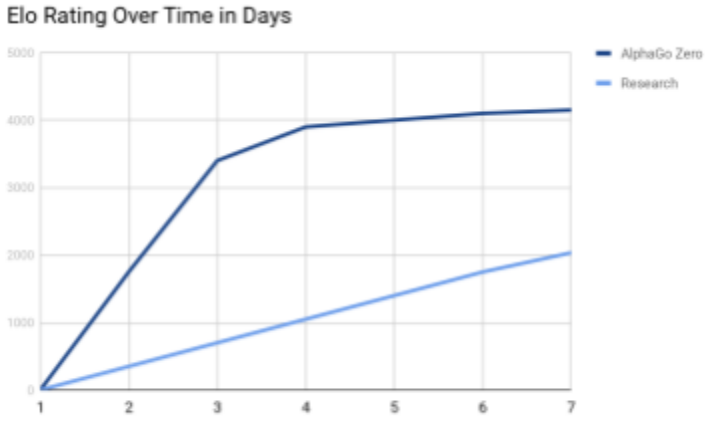


Figure 5. Comparison of the learning between AlphaGo Zero and the research implementation.

5 Future Work

For the potential future work of the MCTS-based algorithm, the author notes a few potential benefits which could increase the strength of play and raise the programs Elo rating. First, diversifying the APP samples the program works by incorporating a file location to load more data from. Given the static structure of APP, an large number of patterns could become known to the Go community. Being able to incorporate these patterns in a singular location as opposed to cramming it into the source code would make for a better designed process. Second, The current program only recognizes 3x3 patterns, so incorporating a large pattern recognition process to the current board analysis would further improve the Elo. It would aid the program by arming it with the knowledge necessary to handle patterns similar to laddering which requires an analysis of a 3x3 space and recursively check in each diagonal direction without larger pattern recognition.

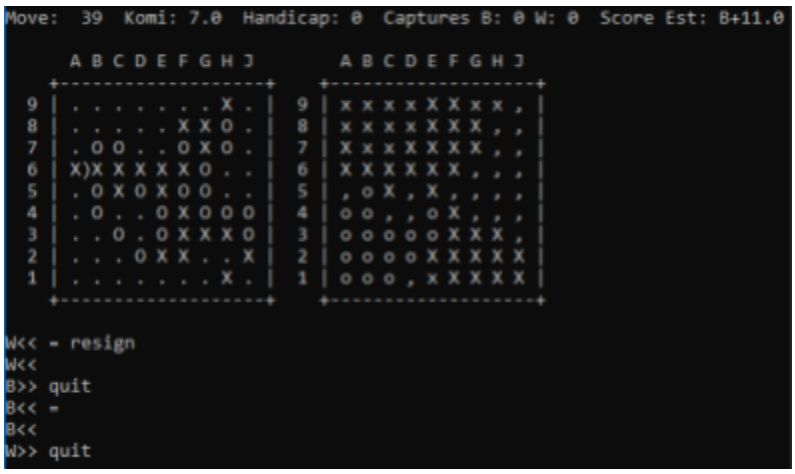
For the future work of the reinforcement learning algorithm, there are a few lacking implementation details which exist in the AGZ paper which inhibit but do not prevent the reinforcement learning process. First, adding a virtual loss to the learning process. Adding such a process would allow for the MCTS to be parallelized without each thread affecting the real loss of a node. This would also point the MCTS to evaluate a single option as opposed to the 8 or 16 proposed due to the thread count. Secondly, increasing the number of filters which was not at the count of the proposed paper could increase the precision of behavior.

6 Conclusion

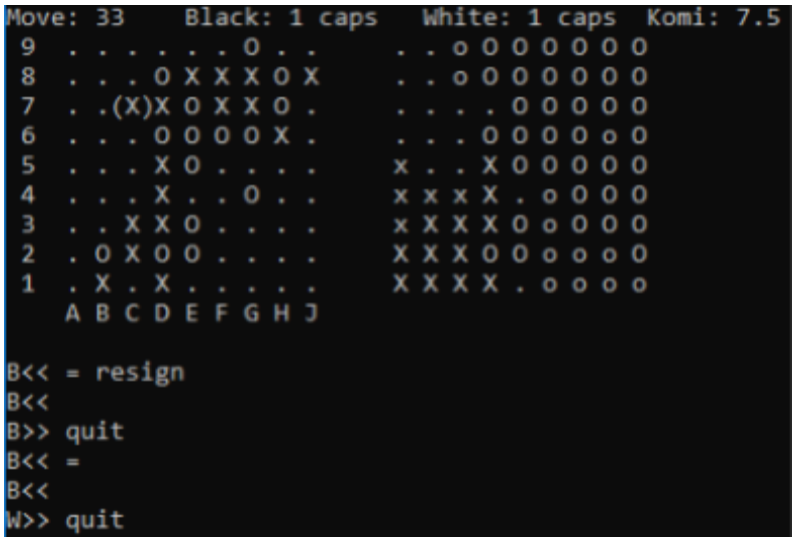
There were a variety of objectives behind this project. These included the following: researching various projects and papers related to the topic, researching specifically the machine learning and reinforcement learning processes of AlphaGo and AlphaGo Zero, implementing similar algorithms utilized in repositories like Pachi and minigo, and systematically analyzing the performance of the written algorithms. Each of these goals was met in one form or another, and the results of each are presented above in this report with similar structure to the execution of the project. This project met my largest of learning-oriented goals I personally set as a takeaway from performing this research. My takeaway that I will use in future research and projects was understanding how an AI interacts with a state space of a game's environment and learning how to model a game state in order to utilize effective processes of machine learning and reinforcement learning.

Appendix I

1. MCTS with APP Game Result (B: Pachi, W: Research Implementation)



2. Reinforcement Learning Game Result (B: Pachi, W: Research Implementation)



Bibliography

Baudiš P., Gailly J. (2012) PACHI: State of the Art Open Source Go Program. In: van den Herik H.J., Plaat A. (eds) Advances in Computer Games. ACG 2011. Lecture Notes in Computer Science, vol 7168. Springer, Berlin, Heidelberg.

Browne, C. et al. A survey of Monte-Carlo tree search methods. IEEE Trans. Comput. Intell. AI in Games 4 (2012).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Chen, Y. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354.

TensorFlow. (n.d.). Retrieved April 27, 2019, from https://www.tensorflow.org/.

The Go Text Protocol. (2009). Retrieved April 27, 2019, from https://www.gnu.org/software/gnugo/gnugo_19.html.