Reinforcement Learning for Gomoku

Kevin Chen 504863921

Paokuan Chin 504059777

Stathis Megas 104683465

Abstract

We have implemented the *AlphaGo Zero* algorithm [2] to train a program to play the game gomoku on a 9×9 grid. The *AlphaGo Zero* algorithm can be broadly categorized as policy iteration, where the value function is estimated by an artificial neural network and Monte Carlo tree search. The final models that we produced trained over two days on upwards of $7,500$ self-play games, and were able to play at an advanced human-like level.

All code can be found at: github.com/kevinddchen/Gomoku-project/.

1 Introduction

In 2016, Deepmind made a phenomenal breakthrough in the field of artificial intelligence (AI) when its computer program, *AlphaGo*, beat the world champion Lee Sedol at Go, a game which before then was considered too complicated for computers to beat humans. The *AlphaGo* algorithm used a novel approach of supervised reinforcement learning involving an artificial neutral network (NN) trained on the games of professional players and self-play games [1]. In 2017, a newer version, *AlphaGo Zero*, used unsupervised reinforcement learning and trained its NN only on self-play games [2]. This program was stronger than its previous versions and used significantly less computing time to train. The latest version, *AlphaZero*, is generalized to play other games as well, such as chess and shogi [3].

The goal of our project is to reproduce the algorithm of *AlphaGo Zero*. But since it requires 4 TPUs just to play—let alone train—we turn to implementing it for a much simpler game, gomoku. Gomoku is played by two players who take turns to place their stones on a grid. The first player to have a connected line of 5 stones in a vertical, horizontal, or diagonal direction wins the game. The official board size is 15×15, but for simplicity we used 9×9 in our project. Using the *AlphaGo Zero* algorithm, we produced a program that plays gomoku at an advanced human-like level.

The outline of the report is as follows. In section 2, we briefly summarize the *AlphaGo Zero* algorithm adapted for gomoku. In section 3, we describe the specific design choices made in our gomoku implementation and the training procedure. In section 4, we summarize the performance of our programs. In section 5, we conclude with a discussion on possible improvements.

2 *AlphaGo Zero* for gomoku

The *AlphaGo Zero* algorithm, described in [2], can be broadly categorized as policy iteration, where the value function is estimated by a NN and Monte Carlo tree search (MCTS). A state is a configuration of the board, which is a 9×9 matrix, taking on either values 0, 1, or -1, corresponding to the position being empty, occupied by black, or occupied by white, respectively. The NN is a function f_θ with parameters θ that maps a state to (p, v), where p is a 9×9 matrix of probabilities and $v \in [-1, 1]$ is a real number. p represents the prior probability of playing a move at the corresponding position, and v represents the value of the state. *AlphaGo Zero* uses a convolutional neural network for the NN f_θ.

Preprint. Under review.

At each state s_0, the algorithm chooses the next move by first performing MCTS. In the game tree, each node corresponds to a state and each edge corresponds to the move that goes from the previous state to the next state. Associated with each node is a value $v(s)$. Associated with each edge is a prior probability $p(s,a)$, an action value $Q(s,a)$, and a visit count $N(s,a)$. The game tree is initialized with s_0 as the root of the tree. In each simulation, the tree is traversed by choosing moves with maximum upper confidence bound $Q(s,a)+U(s,a)$, where

$$U(s,a) = c \cdot p(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \tag{1}$$

until a leaf node s' is reached. The constant c controls the exploration of the simulation. Next, the leaf node is expanded by evaluating $p(s',a), v(s') = f_\theta(s')$ using the NN on the node. If the leaf node is a terminal state, then $v(s')$ is just ± 1 or 0, depending on which player wins or if it is a tie. Finally, for each edge that was traversed in a simulation, the associated Ns are incremented by 1 and the associated Qs are updated to

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)}(\pm v(s') - Q(s,a)) \tag{2}$$

i.e. $Q(s,a)$ keeps the running mean of the values of all visited descendent nodes. Note that since we are using the same NN for black and for white, we have to flip the sign of $v(s)$ accordingly when updating $Q(s,a)$, depending on which player's turn it is in the game tree.

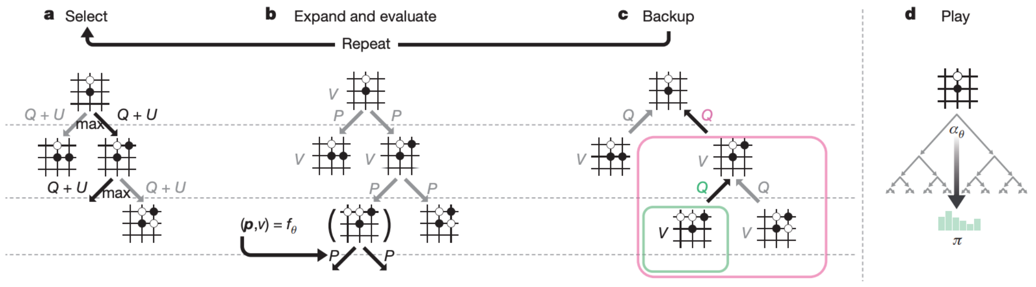


Figure 1: MCTS in *AlphaGo Zero*, taken from [2].

After a fixed number of simulations (500 for our models), the algorithm plays a move according to a policy based on the frequencies of moves made during the MCTS, $\pi(s_0,a) \propto N(s_0,a)^{1/\tau}$. The temperature τ is a constant which controls the exploration of the policy. After playing the move, the game state s_0, policy π, and updated value $z = \sum_a \pi(s_0,a)Q(s_0,a)$ is saved as a tuple (s_0,π,z). After several games, the set of saved $\{(s,\pi,z)\}$ is used to train the NN parameters θ such that the outputs $p, v = f_\theta(s)$ approach z, π. In other words, we minimize the loss function

$$L = (v-z)^2 + \boldsymbol{\pi}^T \log \mathbf{p} + c_2 ||\theta||^2 \tag{3}$$

where c_2 is the L^2 regularization constant. The idea is that the MCTS π and z will always improve upon the prior p and v obtained quickly from the NN. By training the NN to mimic the MCTS output, the next iteration of MCTS using the updated f_θ will further improve the policy. Repeating this process, we obtain a program that can play gomoku very well.

3 Implementation specifics

Although Go and gomoku are played using the same equipment, they are very different games. In gomoku, the first player (black) has a significant advantage over the second player (white). Because of this, many tournaments use rule variations to achieve balance. For our project, we opted not to use these variations and instead kept the simplest ruleset, known as *freestyle gomoku*. In particular, a completed line of 6 or more stones also counts as a win. Moves in gomoku are also more motivated by short-term goals than in Go. For instance, a player will often be forced to play a specific move to avoid outright losing the game. Go is played on a 19×19 grid and gomoku is played on a 15×15 grid, but we decided to use a 9×9 grid to speed up the training.

In *AlphaGo Zero*, the input to the NN was a $19 \times 19 \times 17$ tensor—the 17 channels correspond to features extracted from the game board which track the move history of each player. For our gomoku algorithm, we independently trained two models using different sets of features:

(I) $9 \times 9 \times 1$ tensor containing the raw game board. This is the simplest model.

(II) $9 \times 9 \times 4$ tensor containing four features, described below. These features are designed to address the short-term nature of the game as well as the black-white asymmetry.

- 1 for current player's stones, 0 otherwise.
- 1 for opponent's stones, 0 otherwise.
- 1 for opponent's last move, 0 otherwise.
- all 1 if current player is black, all 0 if current player is white.

The NN architecture is the same for models (I) and (II), and is given below.

input = 9×9 matrix	
Conv, 32 filters, size 3	
Conv, 64 filters, size 3	
Conv, 128 filters, size 3	
Conv, 4 filters, size 1	Conv, 2 filters, size 1
Dense, 9×9 units	Dense, 64 units
	Dense, 1 unit

Figure 2: Neural network architecture.

All the activation functions are ReLU except for the last layer on the left and right heads, which are softmax and tanh, respectively. The left head outputs the probability vector $\mathbf{p}$ for each move, and the right head outputs the value v for the given state. This architecture is from the implementation in [4].

The hyperparameters of the MCTS were chosen to be $c = 4$ and $\tau = 1$ for increased exploration. For each leaf node expansion we also added a small amount of Dirichlet noise to the prior probabilities, $p(s, a) \leftarrow 0.75p(s, a) + 0.25\eta_{s,a}$ where $\eta_{s,a} \sim \text{Dir}(0.3)$. Each move used 500 MCTS simulations and took ~ 1.5 sec on a single CPU. For each (s, π, z) sample saved to the disk, we augmented our dataset eight-fold by also saving all rotations and reflections of the board. This was important as one of the limiting factors for the training speed was the generation of enough self-play games. In training our two models, we used slightly different training regimens:

(I) After 200 self-play games, train for several epochs on the most recent 100,000 samples.

(II) After 750 self-play games, train for several epochs on these games ($\sim$130,000 samples).

Due to technological restrictions, we did not self-play and train in parallel, as done in *AlphaGo Zero*. However, since the largest bottleneck in the algorithm was the generation of self-play games, our approach allowed for self-play games to be generated in parallel on several CPUs. By running several Google Colab sessions in parallel, we were able to generate up to 1,500 self-play games per hour. Each time we collect self-play games (200 or 750 games) and then train, we call that one *iteration*. We perform several iterations to improve the model.

To evaluate the performance of our models, we implemented a simple program that required no training. A feature vector $\mathbf{v}$ of 8 integers is associated with each possible move, given by the number of 2-chains, 3-chains, 4-chains, and 5-chains the move would create for the player, and the number of opponent 2-chains, 3-chains, 4-chains, and 5-chains the move would block. A weight vector $\mathbf{w}$ assigns points for each of these features, and was hard-coded to be $(1, 3, 9, 81, 1, 3, 9, 27)$. The algorithm chose a move with maximal score $\mathbf{v}^T\mathbf{w}$. This simple algorithm does remarkably well, but it can be beaten by an amateur human player. This program served as the benchmark to compare our RL models against. We refer to this model as the *benchmark* model.

4 Results

Model (I) was trained for a total of 26 iterations (5,200 games), and model (II) was trained for a total of 10 iterations (7,500 games). Each model took roughly two days of real time to train. At the end of training, we playtested the final models to gauge their performance in a subjective manner. τ was reduced to 0.1 in order to pick the most optimal moves. We found that both models played exceptionally well and it was difficult for us to beat them consistently, especially when we played as white. There was no discernible difference between the performance of two models.

To obtain more concrete metrics of performance, we playtested our models against each other at $\tau = 0.1$, as well as against the benchmark model. Each pair of models played 20 games as black and 20 games as white, and the win/lose-rates were recorded.

White	*Black*				
	Benchmark	(I) 13 iter.	(I) 26 iter.	(II) 5 iter.	(II) 10 iter.
Benchmark	-	**.8/.1**	**.8/.05**	**.7/0**	**.95/0**
(I) 13 iter.	.35/.55	.95/.05	1/0	1/0	.85/.1
(I) 26 iter.	.1/.65	.25/.7	.25/.3	.25/.6	.45/.1
(II) 5 iter.	.05/.65	.5/.15	.75/.2	.45/.35	.9/.05
(II) 10 iter.	0/.7	.65/.2	.35/.4	.7/.2	.75/0

Figure 3: Win/lose rates between different models. For example, in the first row of the third column, $0.8/0.05$ indicates that model (I) with 26 iterations, when playing as black, has a win rate of 0.8 against the benchmark model, which is playing as white. For the same pairing, the lose rate is 0.05. They don't add up to 1 because there could be ties.

Since black has a significant advantage in gomoku, the performance of a model over another needs to consider both the win-rates as black and white. It was shown by L. Victor Allis that on a 15×15 board, black can always force a win [5]. Therefore, on a 9×9 board, white can at most get a tie if both players play optimally. We indeed see that model (II) with 10 iterations has near-zero lose rate as black, even against itself.

This led us to define the following heuristic for comparing two models: let $B(i, j)$ be the win-rate for model i when it plays as black, and $W(i, j)$ be the win-rate of model i when it plays as white. Then define a score,

$$\text{score}(i, j) = B(i, j) - B(j, i) + W(i, j) - W(j, i) \ . \tag{4}$$

In words, this score is the difference in win-rates as black plus the difference in win-rates as white. A model i is better than a model j when $\text{score}(i, j) > 0$. This happens, for instance, when both the win-rate as black and the win-rate as white are higher for i. Our scoring system has the following desired properties:

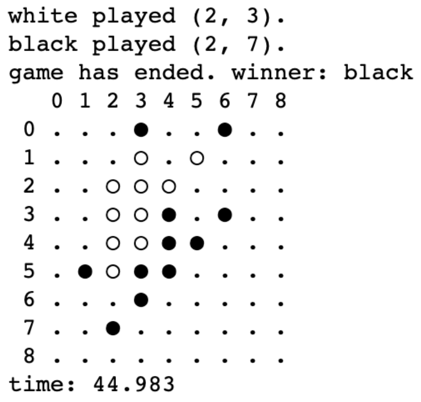
- Antisymmetry; $\text{score}(i, j) = -\text{score}(j, i)$.
- $\text{score}(i, i) = 0$, i.e. a model is equal to itself.
- Win-rates as black are weighed equally to win-rates as white.

B	A				
	Benchmark	(I) 13 iter.	(I) 26 iter.	(II) 5 iter.	(II) 10 iter.
Benchmark	0	0.9	1.3	1.3	1.65
(I) 13 iter.	-0.9	0	1.45	0.65	0.3
(I) 26 iter.	-1.3	-1.45	0	-0.9	0.4
(II) 5 iter.	-1.3	-0.65	0.9	0	0.35
(II) 10 iter.	-1.65	-0.3	-0.4	-0.35	0

Figure 4: $\text{score}(A, B)$ for each pair of models.

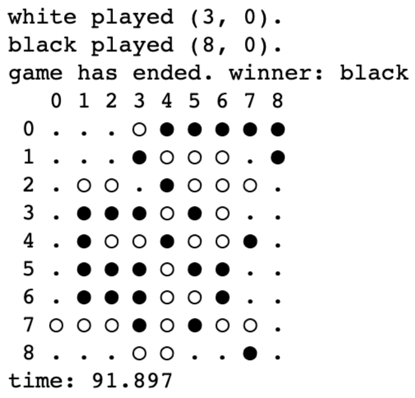
According to our scores, we can see that all our models are able to beat the benchmark model. Both models show improvement with more iterations as the score against the benchmark model increased, as expected. Additionally, we are able to conclude that (II) with 5 iterations is stronger than (I) with 13 iterations, but is weaker than (I) with 26 iterations. (II) with 10 iterations is our strongest model. While the comparison is not exactly on fair footing, since (II) trained on more data than (I), this does seem to suggest that the model is leveraging the additional features available to it.

Viewing the self-play games and playing against it as human, we see qualitatively how the models develop strategies over iterations. We will describe the progress of model (II) below. With zero iterations, the model mostly makes random moves, but it can make a 5-chain and win if it already has a 4-chain. This is because in the MCTS search, it only needs one move to be rewarded. After the 3rd iteration, the model learns to make adjacent moves to form longer chains. It also knows that it needs to block 3-chains and 4-chains of the opponent. After the 5th iteration, the model always tries to start with playing the center position, if it hasn't already been taken. It can also prioritize correctly whether to block opponents or to form chains. By the 10th iteration, it knows which direction to extend a chain and which direction to block the opponent for better positioning. It has also learned to not blindly create 3-chains or 4-chains whenever it can, but can play other moves that may be more beneficial in the long run. It can also see potential moves that create double chains several steps ahead and play or block accordingly.



```
white played (2, 3).
black played (2, 7).
game has ended. winner: black
   0 1 2 3 4 5 6 7 8
 0 . . . ● . . ● . .
 1 . . . ○ . ○ . . .
 2 . . ○ ○ ○ . . . .
 3 . . ○ ○ ● . ● . .
 4 . . ○ ○ ● ● . . .
 5 . ● ○ ● ● . . . .
 6 . . . ● . . . . .
 7 . . ● . . . . . .
 8 . . . . . . . . .
time: 44.983
```

(a)



```
white played (3, 0).
black played (8, 0).
game has ended. winner: black
   0 1 2 3 4 5 6 7 8
 0 . . . ○ ● ● ● ● ●
 1 . . . ● ○ ○ ○ . ●
 2 . ○ ○ . ● ○ ○ ○ .
 3 . ● ● ● ○ ● ○ . .
 4 . ● ○ ○ ● ○ ○ ● .
 5 . ● ● ● ○ ● ● . .
 6 . ● ● ● ○ ○ ● . .
 7 ○ ○ ○ ● ○ ● ○ ○ .
 8 . . . ○ ○ . . ● .
time: 91.897
```

(b)

Figure 5: Sample games between models. (a) (I) as black, (II) as white. (b) (II) as black, (I) as white. The models tend to give up on blocking once it sees that there is no way to win. Hence we see the free 5-chain on the left.

5 Conclusion and discussion

We have achieved our goal of reproducing the *AlphaGo Zero* algorithm for the game of gomoku and produced programs that played at an advanced human-like level. Our final models were evaluated against a baseline program, and were able to beat it consistently. During playtests, our models felt very strong and it was difficult to consistently beat them, especially when we played as white.

During our testing, we found that model (II), which contained additional features, performed marginally better than model (I), which used the raw game board as its sole feature. We once again emphasize that this comparison is not exactly on fair footing, since model (II) trained on 7,500 games whereas model (I) trained on 5,200 games. Rather, we conclude that the similarity of the final models actually demonstrates the robust nature of the *AlphaGo Zero* algorithm and, in general, the *AlphaZero* algorithm [3].

It is expected that additional training will continue to improve these models. Enlarging the NN architecture may help develop more sophisticated models. Increasing the number of MCTS simulations may also increase performance, but the heavy cost to computing time in obtaining self-play games may make this approach not worth it.

It would be interesting to train the model to handle various rule variations, and to see how they hold against the current state-of-the-art gomoku programs. If training on a larger 15×15 board is too slow, we had an idea to employ transfer learning: by copying the weights of the convolutional layers to the new model, it may be possible to train up to a strong model with fewer games. These possibilities we leave for possible future projects. One potential problem that may arise is that since black can force a win on a 15×15 board, and since the model tends to "give up" once it sees that there's no way of winning, the model may realize that it cannot win as white very early in a game and start playing illogical moves (from the human perspective). A potential solution would be to include a tiny γ factor for white, so that at least white will try to prolong the game as much as possible.

Personally, this project was very rewarding. We had lots of fun playtesting our AI, and we invite you to play as well.

Contributions

Each team member had roughly equal contribution to this project. K. C. implemented and trained model (II). P. C. implemented and trained model (I). S. M. developed the MCTS algorithm.

References

[1] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), pp. 484–. URL: `http://dx.doi.org/10.1038/nature16961`.

[2] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–. URL: `http://dx.doi.org/10.1038/nature24270`.

[3] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: `1712.01815 [cs.AI]`.

[4] junxiaosong. *AlphaZero_Gomoku*. URL: `https://github.com/junxiaosong/AlphaZero_Gomoku`.

[5] Louis Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. 1994.