CIS 522, Spring 2021
Report for Final Project: GomoBot

Varun Lalwani, Chunxi Liu, Jongwon Kim

April 2021

1 Introduction

Gomoku is a strategical board game typically played on a Go board with Go pieces, black and white stones.Its rules are simple: two players take turn to place a stone of their color and the first player to complete an unbroken chain of 5 stones in any direction wins. The AlphaGo and AlphaZero left significant milestones in the field of artificial intelligence and impressed the public audience by showing how powerful an artificial intelligence can be. Since then the reinforcement learning has gained a lot of attention. In this project, we explore reinforcement learning algorithms on the game of Gomoku and compare the agents trained on different algorithms. For simplicity we limit ourselves to a smaller board size of dimension 8 by 8.

For the purpose of this project, we start with the minimax algorithm which is a non-deep learning benchmark tree search algorithm used for various strategy games. Then we move onto the deep Q-learning algorithm which is our base deep model. Finally we consider the AlphaZero based model based on the Monte Carlo tree search discussed during the first and the last week of the course.

To compare how competitive each algorithm is, we let the trained agents play against each other and observe the result distributions. DQN and AlphaZero algorithms share in common that they are both deep learning benchmarks and minimax and AlphaZero are similar in that they are both tree-search based algorithms. One important aspect of Gomoku is that the first player always has an advantage over the second player. Hence we can observe how the agents perform when they play first or second. If an agent has an improved result when played first, then that would be a good sign indicating that the learning captured an important property or strategy of Gomoku.

2 Methods

The basic structure for the game class is slightly different for each algorithm but they all share the same underlying idea. We represent the game as a $n \times n$ tensor or $n \times n \times 2$ tensor where the $n \times n$ dimension accounts for a location on the board and the binary dimension would indicate in one or two dimension as ± 1 or $(1, 0), (0, 1)$. In the following subsections we explain the three different methods in more detail. The goal would be to train an agent to play against itself to lead to a draw game when tested against itself.

2.1 Minimax

MiniMax is a Tree Search Algorithm. It is profoundly used in many Grid board games, where the game is a 2 player turn based game. The algorithm looks at all possible board state after n steps, evaluates all the board states, and select the move associated with the board state with maximum score and minimum risk. The number of steps we want to look into the future is referred as depth.

The algorithm starts with a depth of 0. The algorithm first calls the maximizer. If the depth of n is reached or if the game is over than, the maximizer evaluates the board and sends a score. A high positive score indicates that the board state is favourable to us. A high negative score indicates that the board state is favourable to opponent. If neither the depth of n is reached nor the game is over than, then maximizer calls minimizer for each possible board state after one move. The depth of minimizer is 1 + depth of maximizer.

Minimizer returns a score. Maximizer returns the maximum score out of all the minimizer calls and also returns the move associated with the maximum score.

For minimizer, if the depth of n is reached or if the game is over than, the minimizer evaluates the board and sends a score. A high positive score indicates that the board state is favourable to us. A high negative score indicates that the board state is favourable to opponent. If neither the depth of n is reached nor the game is over than, then minimizer calls maximizer for each possible board state after one move. The depth of maximizer is 1 + depth of minimizer. Maximizer returns a score. Minimizer returns the minimum score out of all the maximizer calls and also returns the move associated with the minimum score.

The evaluator of maximizer and minimizer gives exponentially higher score to longer sequences in a given window. For example, if we want a sequence of 5 to win a game. If my AI model has filled a sequence of 2 positions with rest of the position in the window being empty, then I will reach a score of 100 for that window. If my AI model has filled a sequence of 4 positions, then I will receive a score of 10000 for that window. If the opponent has filled a sequence of 4 positions, with the rest of the positions being empty, then I will receive a score of -10000, for that position. Then I sum my scores for all possible windows, and then return the final score.

Here maximizer emulates, how our AI algorithm would behave and the minimizer emulates how the opponent would behave. The algorithm recursively calls the maximizer and the minimizer. Let the depth of the board be n and the number of possible moves be m, then the runtime complexity of this algorithm is $O(m^n)$.

The higher the depth of Minimax is, the better is the performance of the algorithm, but the inference time also increases exponentially.

2.2 DQN

Before we discuss the method, we'd like to explain that we tried to code everything from the scratch for the DQN but could not train a successful agent at all. Due to the time constraint and technical difficulty, we decided to use a pre-trained model we pulled from github. In this subsection we will discuss what our initial approach to the DQN agent was, what the problem was with the training and how the pre-trained model works.

A DQN is a good algorithm for environments with deterministic optimal policy and discrete spaces. To train agents with no human play data, we thought off policy algorithm DQN would be a good choice for a base deep benchmark. The winning condition for Gomoku is similar to that of tic-tac-toe which is very simple in comparison to other games like chess or Go. However as DQN does not look ahead like other tree-search based algorithms, we expected that the DQN agent to be harder to train to an optimal playing.

A DQN agent consists of a network Q, the policy π, a replay buffer R together with other parameters like the optimizer. Given a board state s, the network Q will compute the Q-values for the actions in the action space:

$$Q(s) = [Q(s, a_1), \ldots, Q(s, a_{n^2})]$$

where n would be the size of the board here. So for a given state s and $Q(s)$, the policy π will epsilon greedily choose an action avoiding the actions that are invalid.

The original design of Gomoku game was that an agent or a player gets a reward of -1 for keep going and 10 for winning the game. We let two agents play each other for a chosen number of times and every time an agent made a move there was a training or update in the Q.

At first, we used the standard Bellman equation recursion. For a given state s and the action a maximizing $Q(s, a)$, let s' be the new state obtained from s by the action a. Then the update function was given as follows

$$Q_{pred} = Q(s, a) = \max_{a'} Q(s, a')$$
$$Q_{target} = \text{reward} + \text{discount} \cdot \max_{a'} Q(s', a')$$

where discount is a chosen parameter and the update happened at the level of a sample batch from the buffer. However as Gomoku is a two player game, it is actually the opponent who plays on the board state s'. So this update equation at least theoretically does not make sense. To solve this problem, we tried to look ahead one more step. Let s' be the same new state and now let a' be the action that would be chosen by the opponent and let s'' be the state after the action a'. (As the training happened concurrently, we had access to both agents at all time in the algorithm). Then we let the agent look ahead to its best move in its next turn with the following update

$$Q_{target} = \text{reward} + \text{discount} \cdot \max_{a''} Q(s'', a'')$$

Now the difficulty we had was with the convergence. Even when we tried the training for the 3×3 size board, the agent did not play optimally. For example, the agent trained to play the first move did not learn to place a stone in the center of the board. To solve the problem, we checked the Q-values as we train the agents and observed that the Q-value estimates kept increasing extremely high. One approach we took was to train one agent at a time. The idea is to have a fixed agent that plays completely randomly and train our agent against it. This way as the random agent does not play optimally, we may not need to look ahead as above. Then we can train our agent against a copy of itself which does not get trained and so on. Unfortunately we were not able to solve the convergence issue with the exploding Q-values.

In fact the pre-trained model we used in experimentation is designed almost exactly the same as our approach. One difference is in the value update. Define s, s', s'' and a as before. But define a' to be the valid action that minimizes $Q^{rival}(s', a')$, that is, a' is the valid action that minimizes the rival's maximal Q-value. Then the update is

$$Q_{target} = \text{reward} - \text{discount}_1 \cdot Q^{rival}(s', a') + \text{discount}_2 \cdot \max_{a''} Q(s'', a'')$$

For both our approach and the pre-trained model, a simple feedforward network with activation layers were used. However as the local configuration is important for the game, one improvement to the DQN can be applying convolution layers of kernel size 5 which is the target number of stones.

2.3 AlphaZero

Alphazero is a deep reinforcement learning algorithm that uses a policy value net to provide heuristics for Monte-Carlo Tree Search (MCTS). The neural net is trained solely on self-play games, which are quite time consuming, and the self-play data is always generated by the latest version of the trained model. To maintain a variety of the self-play data, exploration uses both the visit count (in probability) for each action at each node in the MCTS and a Dirichlet noise:

$$P(s, a) = (1 - \epsilon)P_a + \epsilon(\eta_a)$$

From self-playing, we collect data of the game state(s), the probability (π) computed from the visit counts on the MCTS nodes, and the final outcome of the match(z), where s and z are always from the perspective of the current player.

Because the outcome of a Gomoku match is invariant to rotation and horizontal flip, those transformations are performed to augment the data at the end of each self-play match to help collect data faster and also to balance the data and to increase the variability to some extent.

The self-play data is used to train a policy value network that aims to take in a current board state and output the probability of each possible action under the current state (as the policy) and the value of the current state. The trained policy value network is then used in MCTS for future self-play, which creates a cycle of improvement.

The board state is represented by a 4x8x8 tensor, or 4 8x8 planes, where the first 2 represent the current player positions and the opponent player positions, and the third plane represents the position of the opponent's last move, and the fourth plane represents whether or not the current player starts first (the entire plane is 1 if yes, otherwise all 0's).

The policy value net starts off with 3 Conv2D layers using 32, 64, and 128 3x3 filters. Then it splits into a policy net and a value net for outputting the policy and value, respectively, where both uses a Conv2d layer with 1x1 filter for dimension reduction followed by a dense layer with a non-linear activation.

The training objective is to make the predicted P close to the collected π, and the predicted v close to the collected z. The loss function is thus as follows:

$$l = (z - v)^2 - \pi^T log(P) + c\|\theta\|^2$$

Though trained exclusively on self-play data, the model is evaluated by being played against a pure MCTS algorithm once in a while, and the results show that for the 8x8 board, it takes about 2000 to 3000 self-play matches (2 days of training on GPU) to obtain a good MCTS-AlphaZero model.

Because of the resource constraints, we did not train the 8x8 model ourselves, but rather used a pretrained model for this part.

3 Results

For evaluating the strategy making of our algorithms, we made them play against each other. The player 1 moves first. Each AI played against another AI for 10 rounds. We calculated player 1's win rate and tie rate by taking an average of 10 rounds.

Player 1	Player 2	Player 1 Win Rate	Player 1 Tie Rate
Alpha-Zero	Alpha-Zero	100%	0%
Alpha-Zero	MiniMax	100%	0%
Alpha-Zero	DQN	100%	0%
MiniMax	Alpha-Zero	100%	0%
MiniMax	MiniMax	0%	100%
MiniMax	DQN	100%	0%
DQN	Alpha-Zero	0%	0%
DQN	MiniMax	0%	0%
DQN	DQN	0%	0%

Table 1: AI vs AI Results without randomization

We noticed that all the games between a pair of AIs were played in a similar fashion. This is because these algorithms don't have any randomization is their behavior, which lead to all the games having same outcome and failed to give a generalized evaluation of performance of a pair of AIs in different situations. So, we decided to play the first move from each AIs side randomly. This helped us evaluate the true performance of AI vs AI model.

Player 1	Player 2	Player 1 Win Rate	Player 1 Tie Rate
Alpha-Zero	Alpha-Zero	90%	10%
Alpha-Zero	MiniMax	40%	30%
Alpha-Zero	DQN	100%	0%
MiniMax	Alpha-Zero	90%	10%
MiniMax	MiniMax	10%	80%
MiniMax	DQN	100%	0%
DQN	Alpha-Zero	0%	0%
DQN	MiniMax	0%	0%
DQN	DQN	40%	0%

Table 2: AI vs AI Results with randomization

4 Discussion

4.1 Minimax

With Minimax, We noticed that having a the ability to look ahead all the possible futures within a set of moves provided huge advantage in terms of performance. It had a high win rate even after randomizing the first move, which shows that the algorithm is robust to random play for the first move.

When played 1^{st}, Minimax was able to hold its own against AlphaZero. When played 2^{nd} against AlphaZero, Minimax was able to win 30% of the games and tie in 30% of the games, which is pretty impressive feat against AlphaZero.

When Minimax played against itself, most of the games were tie, 10% of the games were won by Player 1 and 10% of the games were won by Player 2. This shows us that Minimax isn't quite capable of utilizing the advantage of moving first against other Minimax AI models, which is a significant advantage in games like Gomoku. It did utilized this advantage against AlphaZero and DQN.

If we look at the game-play of Minimax, we can notice that it preferred playing initially at the center position and start forming sequence from that point on. It also preferred making diagonal sequences, instead of horizontal and vertical sequences.

The Disadvantage of using Minimax is that it has high inference time, which only rises with increase in complexity of the game. So, Minimax is a great algorithm for playing games with low to mid complexity.

```
Player 1 with X
Player 2 with O

      0   1   2   3   4   5   6   7
  7   _   _   X   _   O   _   _   _
  6   _   _   O   X   _   _   _   _
  5   _   O   X   X   X   _   O   _
  4   O   X   X   O   O   X   _   _
  3   X   O   X   X   X   O   X   _
  2   O   O   O   O   X   _   _   _
  1   _   _   X   _   _   O   _   _
  0   _   _   _   _   _   _   _   _

Game end. Winner is Minimax 1
```

Figure 1: Minimax game-play result against AlphaZero

4.2 DQN

As we can see from the results, DQN formed worst out of all the 3 algorithms. This is because DQN always aimed to complete its sequences and never tried to block opponents sequences. This behaviour caused their high loss rate and can be noticed in their game-play.

We also noticed that DQN tried to capture Diagonal positions as opposite to horizontal and vertical positions. DQN also tried to capture place its points near the center of the board. DQN always placed similar type of moves at the initial stage, which shows that DQN always follows the same strategy and is not robust to randomization of first move.

```
Player 1 with X
Player 2 with O

      0     1     2     3     4     5     6     7
  7   _     _     X     _     _     _     _     _

  6   _     _     _     _     _     X     _     X

  5   _     _     _     _     _     _     X     _

  4   _     _     _     _     _     X     _     _

  3   _     O     O     O     O     O     _     _

  2   _     _     _     X     _     _     _     _

  1   _     _     _     _     _     _     _     O

  0   _     _     _     _     _     _     _     _

Game end. Winner is Minimax 2
```

Figure 2: DQN game-play result against Minimax

4.3 AlphaZero

With AlphaZero, we noticed that it was really good at taking advantage of first move, unlike Minimax. This can be seen when AlphaZero played against another AlphaZero. Player 1 had a 90% win rate and a 10% tie rate, which is a really good performance. Alpha Zero also preferred forming Horizontal and Vertical Sequences, instead of Diagonal Sequences, which can be noticed in its game-play.

We noticed that AlphaZero had a more aggressive moveset, and wasn't quite focused on defensive plays. This lead to AlphaZero being evenly matched against Minimax, when AlphaZero played first. We can see in Results section, where AlphaZero being Player 1, had a Win rate of 40%, tie rate of 30%, which implies a loss rate of 30%.

In the game-play, we also noticed that AlphaZero, occasionally played moves that didn't contributed to any sequences. This is because AlphaZero outputs a probability distribution over possible actions, and sometimes, the moves can be a bit out of no where.

We noticed that the performance of AlphaZero didn't depreciate much after randomizing the first move.

This shows that AlphaZero is also quite robust to randomization of the first move.

```
Player 1 with X
Player 2 with O

     0   1   2   3   4   5   6   7
  7  _   _   _   _   _   _   X   _
  6  _   _   _   _   X   _   _   _
  5  _   _   _   _   X   _   _   _
  4  _   O   _   _   X   X   _   _
  3  _   _   O   X   X   O   X   _
  2  _   _   O   O   X   O   _   _
  1  _   _   _   _   O   _   _   _
  0  _   _   _   O   _   _   _   _
```

Figure 3: AlphaZero game-play result against AlphaZero

5 Conclusion

In conclusion as we expected during the project proposal, the AlphaZero algorithm performed very well. On the other hand, it is interesting to note that the minimax agent, which is based on the relatively simpler non-deep learning algorithm, performed very well. When minimax agent played first against the AlphaZero agent, it actually had a higher winning rate in both of the randomized and unradomized experiments. Also the DQN agent performed very poorly not learning to defend most of the time. This may be because other algorithms are more suitable for Gomoku as tree-based algorithms are often powerful for games played on grid board.

In addition, due to the limitations we had in training such as the time constraint and limited resources, we limited the board size to 8 which is a lot smaller than the typical board size of 19. Because of this smaller board size, the number of possible game states reduce dramatically, which gives less advantage to deep learning algorithms like AlphaZero over non-DL methods like MiniMax.

Also, we noticed that the training of DQN was quite volatile. After 100 epochs, the performance of DQN got worse. In future, we would like to use develop more stable versions of DQN so that it can perform as well as Minimax, if not better. DQN has lower inference time than the other 2 algorithms, which is why it would be great to train a stable form of DQN.

After looking at these algorithms, we can say that Minimax would work well for low-mid board complexities. For higher complexity, Minimax has quite high inference time. For higher complexity versions of Gomoku, we would have to rely on AlphaZero. For future work, we can also try evaluating the performance of MuZero on this game of Gomoku.