Proposal: MCTS and self-play for Gomoku

Shreshth Rajan
shreshthrajan@college.harvard.edu

Yiheng Liu
yihengliu@g.harvard.edu

Abstract

AlphaGo is of significant importance for people who enjoy board games. It would be exciting to implement a similar algorithm from scratch. However, due to the overly complex rules of Go and the excessively large game board, the training time would be too long. we decided to switch to implementing *Gomoku* from scratch.

1 Introduction

It is well-know that AI attains sup Gomoku has similar game board to Go. The objective is for either the white or black player to first connect five stones to win. These five stones can be connected horizontally, vertically, or diagonally.

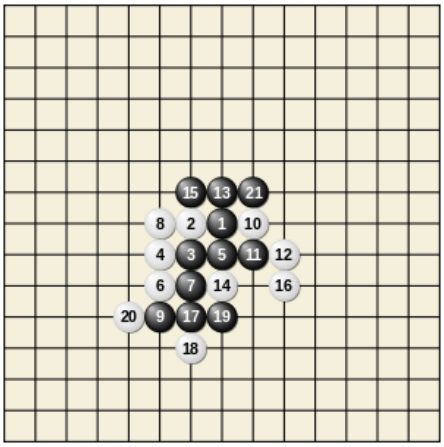

Figure 1: Gomoku

The goal of this project is to implement and train from scratch a model that can play Gomoku well to some extend. The game board is restricted to 7 by 7 in the implementation. For the larger game board, In the field of Go, artificial intelligence has unequivocally outperform human capabilities. The methodology outlined in the paper 'Mastering the Game of Go with Deep Neural Networks and Tree Search' is the following: First it proposed the use of supervised learning to train the policy network with the, followed by a combination of reinforcement learning to further train both the policy and value networks. Once the policy and value networks achieve a certain level of accuracy, Monte Carlo Tree Search (MCTS) is employed for optimal decision-making. Due to the absence of an expert dataset specific to Gomoku, we opted to forgo the supervised learning phase. In the end we will discuss the necessity of the expert move in the model training process.

As this is a technical report about the methodology we adopted, and also these methods are interesting, we will present in detail how we implement the training process. After the technical presentation, we will discuss about challenges and findings.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

2 Methodology

2.1 Policy

Let $p_\theta(a|s)$ denote the policy given the state s. Specifically, $p_\theta(\cdot|s) \in [0,1]^{n^2}$ where n is the board size. Here $p_\theta(a|s)$ is parametrized by some deep convolutional neural network. The reason to choose convolutional neural network is that according to the rule of Gomoku, the winning condition depends locally on the game state. For Go, [1] has also adopted convolutional neural network strategy.

The policy update is from the usual policy gradient method

$$\Delta\theta = \frac{\alpha}{n}\sum_{i=1}^{n}\sum_{t=1}^{T^i}\frac{\partial \log p_\theta(a_t^i|s_t^i)}{\partial\theta}r_i$$

here α is the learning rate, n is batch size for variance reduction. For testing, we select $n = 10$. T^i is the i-th trajectory and r_i is the i-th game reward. To make the game as a zero-sum game, we choose $r = 0$ if draw, $r = 1$ if player A wins and $r = -1$ if player B wins.

2.2 Value

Similar strategy we adopted for value function $v_\theta(s) \in [-1, 1]$. The loss for the value is square error compared to the game final result. i.e. $(v_\theta(s) - r_i)^2$ where r_i is the final result of the i-th game. Follow the strategy in [1], the value gradient method is

$$\Delta\theta = \frac{\alpha}{n}\sum_{i=1}^{n}\sum_{t=1}^{T^i}\frac{\partial v_\theta(s_t)}{\partial\theta}(r_i - v_\theta(s_t))$$

where the notations are the same as policy case. Here v_θ and p_θ are parametrized by independent neural networks, even though in our notation the parameters are denoted as θ.

2.3 Game settings and Neural Network

In the implementation, game status is implemented as a two dimensional array. 0 means the current position is vacant, 1 means is black piece and -1 means it is white piece.

```
Enter your move (row, column): 3,2
0 1 0 0 0 0 0
0 1 0 0 0 0 0
0 0 0 0 -1 0 0
0 0 -1 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

Figure 2: Render

To appriximate the policy and value function, we adopted convolutional neural network with activation function Relu. Relu will introduce non-linearity to the model, for approximating L^p functions. The dense layer we adopted sigmoid as activation function, as Relu in the dense layer tends to make the output policy sparse from our experiment.

However, it is well-known that sigmoid activation has vanishing gradient problem. After this layer, we masks all positions are not vacant, and finally we normalize the output into a probability distribution, which serves as the policy output. In our implementation, we use 4 convolution layers and 2 dense layers for the model.

Board features $\rightarrow$ Convolution $\rightarrow$ Relu $\rightarrow \cdots \rightarrow$ sigmoid $\rightarrow$ Softmax $\rightarrow$ mask $\rightarrow$ scale

2.4 Train the Policy

The training algorithm for the policy is as the following. Initially, we train the 7 by 7 game, and then extend the result to larger game boards. For 7 by 7, as observed the game board is a two dimensional grid, which is kinds of invariant under rotation. The policy will be parameterized under convolution neural networks.

After the 7 by 7 game is trained, the policy can be extended to larger game as the following. First, it is noticed that the wining condition of Gomoku is a local condition. i.e. If black or white pieces satisfy some criterions locally, then the whole game will terminate. Therefore, it is reasonable to apply the previous trained small 6 by 6 policy, as a moving window, to the larger game board. Calculate the policy for each subwindow of a 6 by 6 shape separately , then fill the remaining positions with zeros. Stack all the policy matrices obtained by this method together to form a tensor, which will be used for subsequent training.

To capture the global features, there will be simultaneously a large convolutional neural network trained for the whole large game board, but with fewer parameters. Finally, compress these tensors to a two dimensional matrix with the game board shape, and scale it properly to be probability distribution.

For the training, the opponent's policy is randomly selected from the previous policy (opponentPool), and conduct the self-play. The pesudocode is the following:

Algorithm 1 Training Loop with Gradient Updates

NumEpisodes $\leftarrow 100$
horizon $\leftarrow 30$
$n \leftarrow 10$
for i in NumEpisodes iterations **do**
    if $i \mod 20 == 0$ **then**
        Add opponent to OpponentPool
    end if
    Initialize states, actions, environment
    for j in range(n) **do**
        Initialize values containers
        while $t <$ horizon and Environment.winner $== 0$ **do**
            action $\leftarrow$ PolicyNetwork.sample(state)
            Append state to States
            Append action to Actions
            Environment.step(action)
            if not terminate **then**
                $opponent \leftarrow$ OpponentPool.sample_opponent()
                $opponent\_action \leftarrow$ opponent.sample$(-1 *$ Environment.board$)$
                state, reward, terminate, _ $\leftarrow$ Environment.step(opponent_action)
            end if
            $t \leftarrow t + 1$
        end while
        if reward $== 1$ **then**
            Calculate the i-the game gradient by the policy gradient descent formula
            Calculate the i-the game gradient by the value gradient descent formula
        else if reward $== -1$ **then**
            Calculate the i-the game gradient by the policy gradient descent formula
            Calculate the i-the game gradient by the value gradient descent formula
        end if
        Update PolicyNetwork using Gradient Descent
        Update ValueNetwork using Gradient Descent
    end for
end for

2.5 MCTS

After the policy network is trained by the previous paragrah, then apply MCTS to obtain the optimal move. The method is basically the same as the one covered in the slide. The algorithm: AlphaZero: Action Selection Subroutine. The precise implementation is the following:

Algorithm 2 Monte Carlo Tree Search

$root \leftarrow$ Node
for number of playouts **do**
    $node \leftarrow root$
    $path \leftarrow [node]$
    while $node.children$ **do**
        select best action based on UCB bound.
        $path$.append($node$)
    end while
    $actions \leftarrow [a$ for a in range($node$.board_size $* *2)]$
    for a in $actions$ **do**
        $node$.children$[a] \leftarrow$ next_state$(node, a)$
    end for
    while not meed terminate condition **do**
        randomly select plausible action
        $node \leftarrow nextState$
        $path$.append($node$)
    end while
    Take value network v_θ.
    for n in reversed($path$) **do**
        update number of visit + 1
        if n.current_player $== 1$ **then**
            n.av_value $\leftarrow n$.visits$/(n$.visits $+ 1) \times n$.av_value $+ 1/(n$.visits $+ 1) \times v$
        else
            n.av_value $\leftarrow n$.visits$/(n$.visits $+ 1) \times n$.av_value $- 1/(n$.visits $+ 1) \times v$
        end if
    end for
end for
Select best action based on the UCB bound
return $best\_action$

The above algorithm completely follows the peusodocode presented in class lecture notes. To avoid number of visit equals 0, we add a small number to it.

3 Result and Discussion

The model is trained for 3 hours. However, it is noticed there is no significant improvement of the action strategy.

In principle, the method above should works. However, we notice that there is a vanishing gradient problem during the training process, which we are unable to solve within the time constraint of the project. As a result, the gradient does not update. One reasonable explanation we have so far.

- There is a trad-off between the game length and gradient update rate. The reason is the masked policy. At the beginning, the policy is not masked, which means the action can take values in positions which are not vacant. In this case, the game ends immediately for break the rule. However, due to the input being sparse, at early stage of the game, the action tends to take the position in which there is strongest signal. i.e. it tends to take positions which are not vacant. Since we have not use expert policy to apply supervised training for the policy, this early stage of training is hard to surpass.

References

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

[2] Storkey, Amos. "Training deep convolutional neural networks to play go." Proceedings of the 32 nd International Conference on Machine Learning. 2015.

[3] Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." Advances in neural information processing systems. 2000.

[4] Michael P, Luke J, Mastering the game of Go from scratch, 2018