Omok AI using RL and NN

Derrick Lor
Dept. of Computer Science
University of Massachusetts Lowell
derrick_lor@student.uml.edu

***Abstract*—Omok (widely known as Gomoku) is a 2D two-player strategy boardgame, where the goal is to get five pieces in a row vertically, horizontally, or diagonally. Widely popular in eastern Asian in countries like China, Japan, and Korea. It goes by many different names, but its many appeals are simplicity, game theory, and almost infinitely possible states. The goal of this paper is to document the process of creating an artificial intelligence to play Omok using a variety of computer search algorithms, reinforcement learning, and machine learning.**

Keywords—omok, gomoku, ai, minimax pruning, alpha-beta pruning, reinforcement learning, neural network.

I. Introduction

Early history of Omok dates back to ancient China almost 4000 years ago. Its considered to be one of the world's greatest strategy games alongside Go and Backgammon [3]. The first English appearance of the game came to Europe in the 1880s. Mathematicians and computer scientists have since studied the game's complexity in terms of game theory and possible board states. Trying to find the most optimal sequences of moves and narrowing down the amount of evaluation needed to compute a (NxN)!, almost infinite amount of positions [2].

With current and modern techniques, I will try to create and examine an AI game bot following classic search algorithms like minimax pruning, alpha-beta pruning, with reinforcement learning techniques to gain insight into its behavior[4]. Then I will try to create my own version of a neural network and train it to play Omok.

II. Ease of Use

A. Environment

The main programming language being used to code the program is python. Along with jupyter notebooks to run the code in more friendly and easily organizable way.

B. Dependencies

The version numbers and name of libraries used are as follows:

- Python version 3.10.6
- Pygame version 2.5.2
- Numpy version 1.23.3
- Matplotlib version 3.7.1
- CSV version 0.13
- Operating system is Windows 11.

Identify applicable funding agency here. If none, delete this text box.

III. Markov Decision process

A. Observation space

The initial starting state of the MDP is an empty board with dimensions 9x9. The first move is decided by the agent using their policy. The state space is 81 possible state space for the first turn. The second turn has 80 state spaces. Continuing until terminal state has been reached. Terminal states result from either player pieces or enemy pieces consuming 5 consecutive squares in the grid, or all possible positions in the grid have been taken equivalently no possible actions are available. Therefore, the theoretical max limit of the state space is 81! = 5.7971e120. An empty space is denoted by integer 0. Player piece is integer 1 and enemy is integer 2.

B. Action space

All possible actions are the row and column positions of the board. The board is ordered row major and column minor. The action at row 0, column 0 is positioned at the top left corner of the board. Action 8, 8 corresponds to the bottom right position. The action space therefore is [(0,0), (0,1), … (8,7), (8,8)] with total 81 actions. If a position in the board is taken, the corresponding action will raise an error.

C. Reward Space

The reward for taking an action that does not lead to a terminal state is -1. The reward for taking an action that leads to a win is 100. The reward for a loss is -100. The reward for a draw is -10.

IV. Design and implementation

The core design of the project must be derived from the ground up.

A. Planning

Board class holds the game state, all available actions, whose current turn, number of pieces and board dimensions. All board variables and functions are public. Functions include placing a piece, switching turns, running the minimax and alpha beta algorithms. As well as checking terminal conditions such as win or draw. There are game functions specifically for rendering player versus player, player versus enemy, and enemy versus enemy games using pygame.

B. Interface

Interface is rendered using pygame library. Custom assets for background board and black and white pieces. The interface is enabled only for player versus player and player versus enemy game modes. The board state is rendered to the screen for player feedback. Place pieces by using mouse button left click on top of available empty squares.

C. Challenges and Solutions

1) Win conditions

Checking the win condition was a matter of brute force and could've been implemented in a smarter and more efficient way. However, checking all possible positions still achieved the same effect.

To start, the win condition can only happen after a player or enemy has placed a piece. The goal is to create a string of 5 of your pieces in a row, then the win condition only needed to check after 11 pieces have been placed on the board.

After that, we need to check 5 squares on the board at a time and count the number of pieces summing to 5. Not only that, but we also need to check if all pieces are all from the player or all from the enemy.

The permutations of a win condition can manifest itself horizontally, vertically, positive slope diagonally, and negative slope diagonally. If found, then we can return true. If not, then the win condition is false, therefore continue playing.

2) Minimax and Alpha Beta Pruning

Implementing minimax algorithm took longer than expected. The pseudocode looks deceivingly easy until we are tasked with checking all conditions and recursively calling ourselves.

The main control loop checks if the exit conditions have been met. The exit conditions include reaching maximum depth, if win condition is met or draw condition is met. If so, we evaluate the board position and return the heuristic score of the board according to whomever called the minimax algorithm.

The tricky part is handling if player called the minimax or enemy called the minimax. Implementing the alpha beta pruning was unexpectedly easy. Simply pass alpha and beta into the minimax and check if alpha is greater or equal to beta, if so then prune the branch.

3) Heuristic Evaluation

Scoring the board position or heuristic evaluation of the board position needed to be implemented. The basic structure of the heuristic evaluation is to score points based on the number of 4 in a row, 3 in a row, and 2 in a row.

Not only do we need to evaluate our position, but also the enemy's position. The sum of these two scores will define the heuristic evaluation of the board.

Point distribution is not equally distributed on both sides. The enemy position score is more heavily weighted against the player position. Depending on whose turn is next and the current position of both sides, the highest priority is to block a move resulting in the enemy winning. Favoring defense more than offense to hopefully sustain the game and prolong the chances of winning.

4) Testing

User testing of score evaluation followed the idea of playing higher defense than offense. In testing, blocking an enemy's pen ultimate move allowed for more favorable chances of an endgame.

A pen ultimate move can be defined as placing a piece that results in 3 in a row with empty spaces on both sides. In this case, no matter what the player does, even if playing optimally or non-optimally, will result in a guaranteed win for the enemy in two moves. Preventing this from happening stops the enemy from winning, and allows the player to better improve the current position.

V. Figures and Tables

TABLE I. WIN RATE AI VS AI

Policy			
	Win	***Lose***	***Draw***
Middle starting move	65.75%	34.25%	0%
Random starting move in middle 3x3	70.25%	23.5%	6.25%
Random starting move	34%	61.75%	4.25%

Played against Enemy AI random starting move in middle 3x3

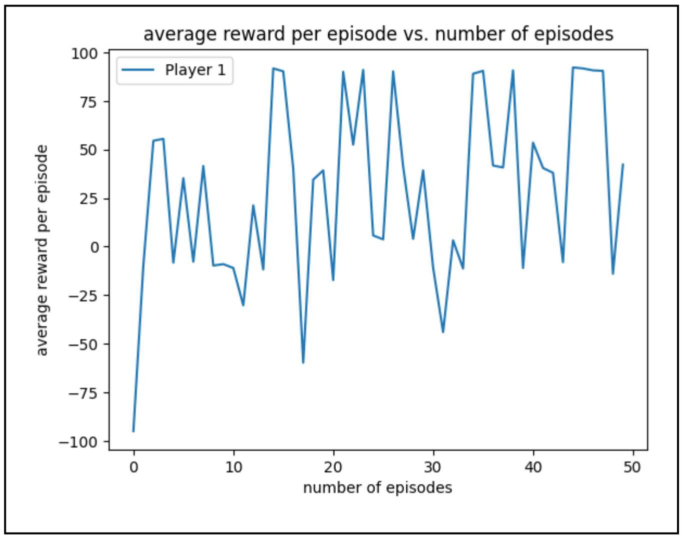

Fig. 1. Average reward per episode plotted against the number of episodes.

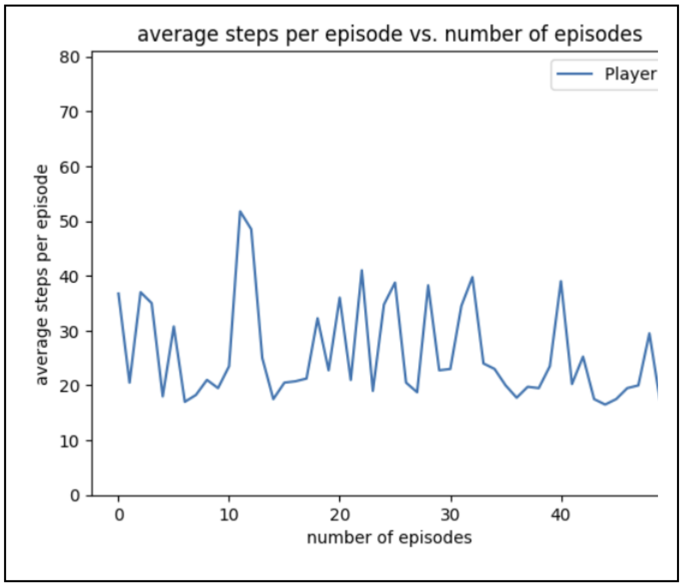

Fig. 2. Average stesps per episode plotted against episodes. The maximum number of steps in an episode is equivalent to N^2, where N is board length. From this plot we can estimate that it takes roughly 20 to 40 moves on average to complete a game, be it win, loss, or draw.

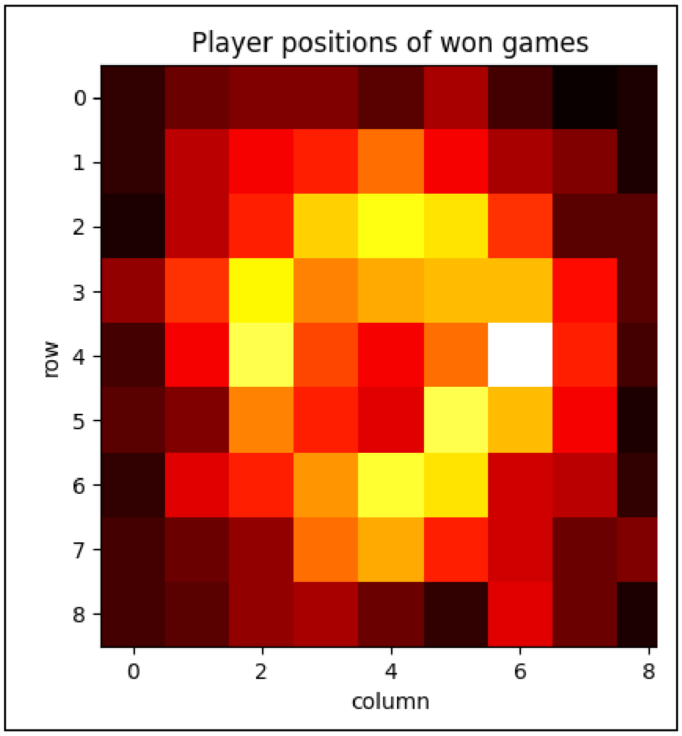

Fig. 3. Heat map of most played positions of random starting move. Lighter colored squares indicate higher amounts of activity or temperature.

TABLE II. NEURAL NET PREDICTED OUTCOME

Evaluation	***Ratio***	***MSE***
Trained Data	0.81	12.918
Testing Data	0.64	15.127

NN being evaluated against the training and testing data. To determine the theoretical win accuracy.
*MSE mean squared error

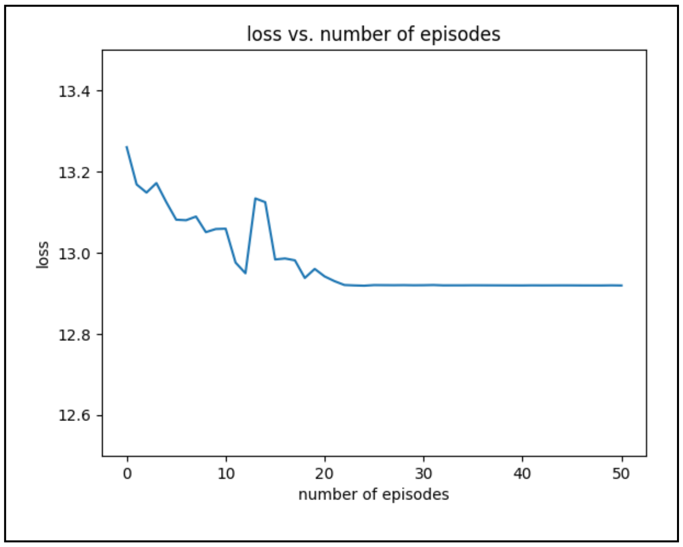

Fig. 4. Plot of the loss over episodes. Lower the loss the better.

VI. Learning Outcomes

A. *Learning from Table 1*

In my testing, I found my AI bot for Omok played with almost 70%, Table 1, winning accuracy, this is comparable to an AI created with deep learning with accuracy of 69% [1]. However, this is heavily under the assumption that my AI or player moves first when playing against enemy AI using the same minimax algorithm. Playing the starting move in any game almost always results in the starting player having an advantage over the opponent. For this reason, the first move is the most crucial part of the entire game. This will dictate the development of enemy actions and can open or close any future trajectories.

1) *Middle starting move*

On the first turn, we always choose the middle square position of the board to be the placement of the player piece. Table 1 results show that on average we have a 65.75%-win percentage against the enemy's 34.25%-win percentage. Also note that there are no chances of drawing. This means that there is always a winner or loser in a game.

2) *Random starting move in middle 3x3*

In this policy, the starting move is chosen at random from the possible 9 positions in the middle of the board. This includes the middle square of the board plus adjacent squares in a 1 square radius, including diagonals. Results show that the winning percentage is up to 70.25% and the losing percentage is down to 23.55. Where this differs from the previous policy is the chance to draw at 6.25%.

Even though the MDP rewards negatively for drawing, we can see that drawing a game is better than losing a game. Losing rewards -100, while drawing rewards -10. From this fact, and the fact that the winning percentage is higher than the previous policy, we can say that this policy is the best policy we have tested.

3) *Random starting move*

As the name suggests, the starting move is completely random. All possible 81 starting moves all have an equally likely chance of being picked. Not surprisingly, our chances of winning significantly plummet down to 34%, chance of losing 61.75%, and drawing 4.25%. By far, this is the worst starting move policy and should not be used if trying to win.

B. *Learning from Fig. 3*

Fig. 3 shows the most played squares of winning games using the starting move policy of random move. The reasoning behind choosing random first move is to uniformly distribute the probability of playing all positions of the board. By doing this, we can truly test all board positions and only highlight squares that lead to the most probable wins.

This form of sampling is rejection sampling. Rejection sampling is running a sample from random starts, then reject samples that do not fit our search criteria. In this case, we reject games that result in loss and draws, only accepting sample games that give us wins.

The edge positions of the board are darker in color and indicate placing pieces in those squares have less likely chances of winning. There is a ring with radius of 2 from the center of the board with high activity. From this, we can assume that playing away from the edges of the board, but near the center of the board gives a higher probability of winning.

C. *The Neural Network.*

Creating the neural network stemmed from [5] as launching off point. Most of the knowledge and code was reused to build the custom NN. The net consists of two hidden layers and two activation functions. Data was first collected from using my RL implementation to generate a long series of winning games and the corresponding states and best possible actions that led to a win. The best action in that current state was generated using my minimax algorithm with alpha beta pruning. This process of generating episodes or trajectories is called Monte Carlo. The idea of generating MC games came from [1].

The state of the board is a 9x9 array, consisting of 81 total elements. The first layer of the NN takes those state arrays and flattens them into 81 input neurons. The next step is to feed the first layer into a rectified linear unit (ReLu) activation function. Then into the second hidden layer with 12 neurons. Then finally using a softmax activation function to generate two possible values corresponding to the row and column number of the best possible action.

Data collection used to feed the NN can be found in the training data set states and training data set actions.

Table 2 is the condensed data gathered from predicting the next action, given a state. Passing the training data set into the NN gave a ratio of 0.81 correctly predicted actions. Or 81% accuracy after training with ~5600 states and actions. To truly test the NN, a new batch of generated games representing my AI using minimax with alpha beta pruning was collected and tested against the NN. This time, the ratio was 0.64 or 64% accurate in selecting best action. This is to be expected since the training data only represented winning games, not lost, or drawn games.

Fig 4. Shows the loss function over episodes. The goal of the NN is to minimize the loss function, however my NN can only reach about 12.9 using the mean squared error (MSE).

D. List of completed work

- Setup core game components and object interactions.
- Programmed win requirements and terminal states.
- Created game assets, pieces, board, and rendering using pygame.
- Implemented minimax algorithm to enable enemy bot AI.
- Implemented alpha beta pruning to speed up algorithm.
- Enabled support for two-player games, player vs AI, and AI vs AI game modes.
- Visualize reward distribution and win-rates with graphs using pyplot.
- Generated training and test data set.
- Built a Neural Network and trained it on Omok games using RL MC techniques.

VII. FUTURE IMPROVEMENTS

As for future improvements I can focus either on my AI or NN. My AI with minimax currently uses a depth search of 1. Increasing the search depth to account for more states could improve the winning accuracy. However, the cost of increasing the search depth even by 1 will cause the system efficient to plummet and take exponentially longer to compute.

A. Enemy moves first

In my program, the first move always belongs to the player. I explored some possible first move policies, however, did not test or research if the enemy moved first. Changing the move order is trivial, just change the order of events in the game loop. I've implemented this in function PlayOmokPygame(ENEMY, PLAYER) by simply passing who goes first in the first argument and who goes second in the second argument. But the juypter notebook used for testing and collecting data used a custom game loop where player always goes first.

B. Convolution Neural Network and Deep Learning

References [1] and [4], uses a neural network and supervised reinforcement learning to train a model to play the game of Gomoku. Their accuracy reached upwards of 69% and 42%, respectively. My NN is simple and uses no machine learning nor neural network libraries unlike those in my references. The next step is to implement a convolution based neural network or to add AlphaGo's algorithm in my toolkit.

REFERENCES

[1] Peizhi Yan and Yi Feng. 2018. A Hybrid Gomoku Deep Learning Artificial Intelligence. In Proceedings of the 2018 Artificial Intelligence and Cloud Computing Conference (AICCC '18). Association for Computing Machinery, New York, NY, USA, 48–52. https://doi.org/10.1145/3299819.3299820.

[2] Dongbin Zhao, Zhen Zhang, Yujie Dai, Self-teaching adaptive dynamic programming for Gomoku, Neurocomputing, Volume 78, 2012, https://doi.org/10.1016/j.neucom.2011.05.032.

[3] Demján, Attila. "The History of Gomoku." gomokuworld. Accessed April 1, 2024. http://gomokuworld.com/gomoku/3.

[4] K. Shao, D. Zhao, Z. Tang and Y. Zhu, "Move prediction in Gomoku using deep learning," 2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC), Wuhan, China, 2016, pp. 292-297, doi: 10.1109/YAC.2016.7804906.

[5] Aflak, Omar. "Neural Network from Scratch in Python." Medium, Towards Data Science, 24 May 2021, towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65.