Connect-4 game implementation in AI

Haodong Yang(hyang85) [1]

Github: https://github.com/a5210467/final_project.git

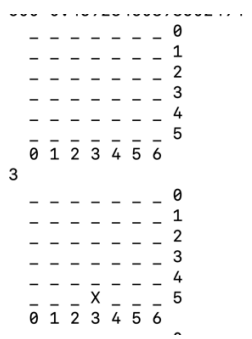
I. Introduction

For this final project I am using the connect 4 gaming. It is a simple game that the first person got 4 connected dice will win the game. More important instead of the normal connect-4 professor asked me to change the rule of the game and that will explain in the later of this report. For the tree search algorithm, I am using the Monte Carlo tree search.[1] Instead of travel through all node in the minimax tree structure in the game, MCTS(Monte Carlo tree search) algorithm only visit some certain pass from root to leaf or from some node to the leaf. The details will also explain in the later of this report. For the Machine learning mode. I am using the NN(Neural network)[2] network with linear function convergence. For the MCTS structure AI even with different size of the board with modified rules for each of the 100 games the non-lose rate for AI is around 90% of the time. Even for the Linear function convergence model for NN the winning rate is around 70%.

II. Domain description

In this section I will describe the modified rules and data structure for the game.

Connect Four is a 2-player game. Player's alternate taking turns. Each side of the Connect Four board has its own color of checkers. There is one color for each player. In my implementation instead of using different color I am using 'X' to present player1 and 'O' present player2 in the terminal. The goal of Connect Four is to get 4 of your color checkers in a row—horizontally, vertically, or diagonally—before your opponent does. When it is your turn, drop 1 of your checkers into the open slot(columns) at the top of the Connect Four board. This allows you to either build your row or stop your opponent from getting 4 in a row. After you drop your checker, it's your opponent's turn. The standard size of the Connect-4 game is 6 rows 7 columns. For each turn one of the players will choose a column that not fully occupied. For example: the image below it. if I enter 3, It will check the 3rd column and going to the lowest row that not being occupied by any player.



Additionally, Professor asked me to modify the rule. For each round each player has 15 percent chance that will fail to drop the dice. More easily to explain there are 15% of the chance will skip your turn and go to the others turn. For example, in the following situation. The terminal will tell which player are skipped. I am using the random [4] library to generate the random variable to decide the skip probability.

```
3
  _ _ _ _ _ _ _ 0
  _ _ _ _ _ _ _ 1
  _ _ _ _ _ _ X 2
  _ _ _ X _ _ O 3
  _ _ _ X _ _ O 4
  _ O X X _ _ O 5
  0 1 2 3 4 5 6
AI drop
```

The last thing is the data struct of the game. I am using NumPy[3] 2d array to generate the board. When the game is over it will tell you who is winning the game. The example will be in the next image.

```
  _ _ _ O _ _ _ 0
  _ _ _ O O _ _ 1
  _ X _ O X _ _ 2
  _ X X O X _ _ 3
  _ X O X X _ _ 4
  _ O O X O _ _ 5
  0 1 2 3 4 5 6
Winner :  AI
```

The rules of this game I am reference this github[5] project. I modified it and use my own version.

III. Tree AI description

For tree AI, MCTS [1] is my choice. Monte Carlo tree search (MCTS) is a heuristic search algorithm for some kinds of decision processes. Instead of using Minimax tree for whole process. For example. Each turn you have 7 columns to choose and 15% change jump to next level of the minimax tree. The total size of the tree has the upper bound O(15^n) which is an exponential grows tree structure. MCTS randomly sampling from the specified to the leaf. Expanding the minimax tree based on the random sampling but it also considers about the weight and visit times to decide next random sample. In tree search, there's always the possibility that the current best action is actually not the most optimal action. In such cases, MCTS algorithm becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the current perceived optimal strategy. This is known as the "exploration-exploitation trade-off". It exploits the actions and strategies that is found to be the best till now but also must continue to explore the local space

of alternative decisions and find out if they could replace the current best. [1] The formular should be

```
Weight = self.score_total/self.visit_count +
np.sqrt(2*np.log(self.parent.visit_count)/self.visit_count)
```

Which is

$$uct = \frac{w_i}{n_i} + c\sqrt{\ln\frac{N_i}{n_i}}$$

w is the number of wins for the node considered after the i-th move, n is the number of total visit times, and the N is the total number of simulations after i-th move. C is a constant that can change by person. However since my rule are changed I need modify this uct rules. The new rule coming:

$$uct = \frac{w_i}{n_i} + 0.85\sqrt{\ln\frac{N_i}{n_i}} + 0.15\sqrt{\ln\frac{N_i}{n_{i+1}}}$$

The reason is the counting for the expected value of skip current turns.

For training the AI. I am using the pseudocode below. I use the reference github code from [6]

Algorithm 1: Training MCT once

Input: Node that you want to train

Output: The same node

If any of the children move can end the game w = 1 return

While children of that node is not None:

If never visit before:

Random choose a valid action

Training the new node

Current Node.visit += 1

Else:

Choose argmax(uct value)

Training the new node

Current Node.visit += 1

Keep doing this algorithm until the leaf node.

For my baseline AI I set the basic rules check next step if current player can win go there if current player will lose go there other just a random choice the valid column to put dice. For limit the size of the tree search. I set up a time limit such that for each step there at most 2 second to training for a given node. So, the respond time will be limited.

IV. Neural network description

The trainable parameters of my model is the one liner layer and one hidden layer. It is simple. Since there is only one hidden layer, the formulas should be:

$$layer_1\ output = Bias_1 + W_1X$$

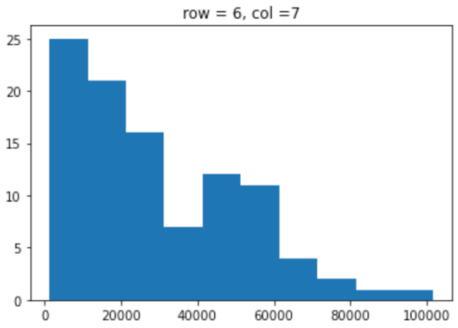
$$laye_2\ output = Bias_2 + W_2 layer_1\ output$$

After I got the training data, I am using two layers take the derivate each step and try to converge it. separate it with training error and testing error. After multiple times of training, I want both of them converge to 0 as possible as they can. For the NN(Neural network) I am using the depth-limit tree search to generate the training example. Then reshape it from a 2d array to a 3d array. Then I am using the pytorch [7] to training the data. The data structure for the board is a 2d tensor array and encode it to a 3d tensor array. One layer for empty spot and one for player1 and the last one is for player 2. I am only use 2 layers for one hidden layer one output layer.

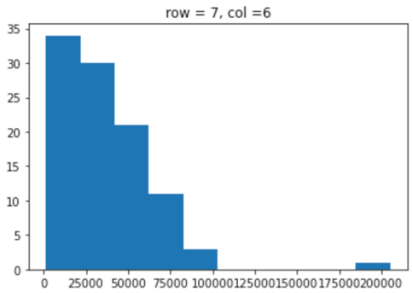
V. Tree based AI experiment

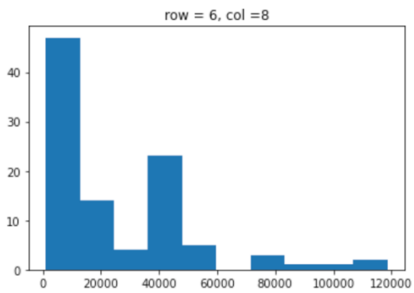
I am using different board size to different experiments. I take 5 different sizes of the board. Each of them run 100 times for simple AI against my tree AI. Histogram is below it.

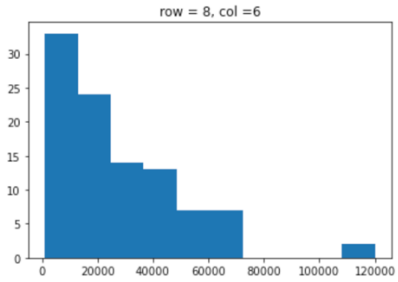
y axis means in 100 games how much games visit node between x axis range. For example in this histogram there are 20 games visit node between 10000 to 20000.

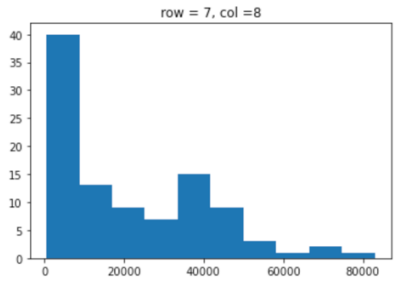


There are 4 more image below.

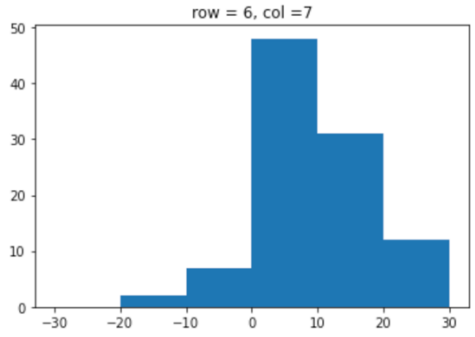


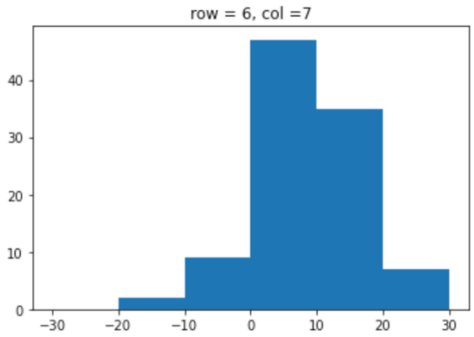


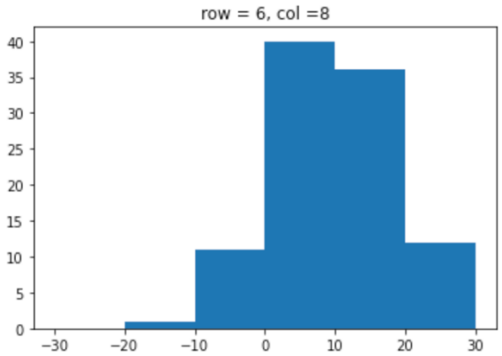


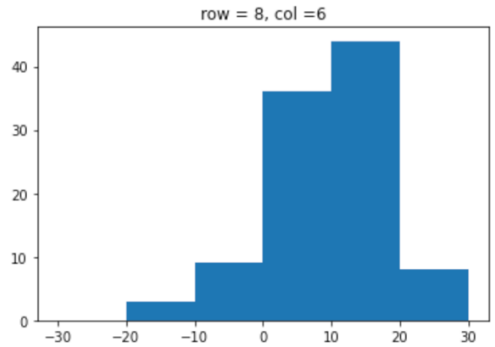


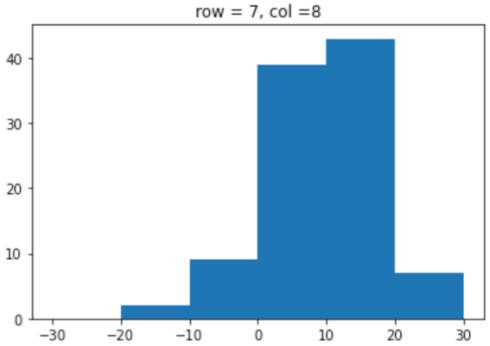
For score count I am not using the simple count win or lose for AI win I check the board if there is a 3 connect for against player I give the AI 5. For 2 connect I give AI 10 and for only 1 I give AI for 20. Otherwise, is a negative point. Here is the image.







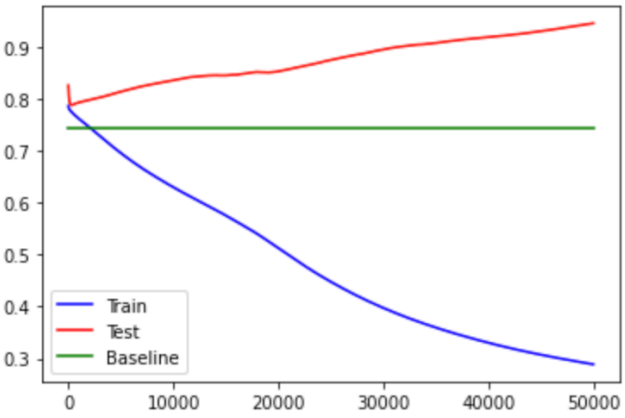




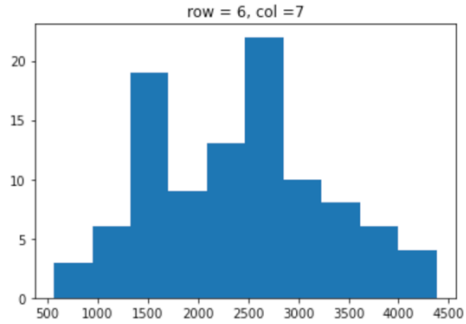
VI. NN Network experience.

Since I work alone everything is done by myself.

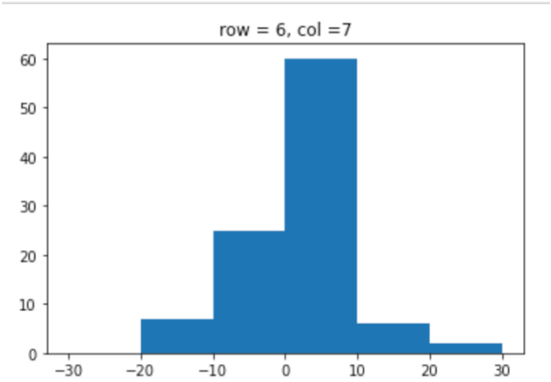
Learning curve is below:



Node that visited:



Score count:



VII. Conclusion

For the whole project. I am implemented different rules connect 4 game and modify the uct rules to fit the new game rules. The most significant result is that for the Tree AI the winning rate is round 90% of

the whole games. It is impressive. The most challenge part is coming from the NN network. I basic using the same for the in-class example to implement my AI. However, it is not well. Since I am working alone and don't have too much time to implement multi-layer NN network. The result is not good enough. If I will continue working on it. the first thing is using more layers to implement the NN network and using non-linear function to converge.

References

[1]. *Rémi Coulom (2007). "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29–31, 2006. Revised Papers. H. Jaap van den Herik, Paolo Ciancarini, H. H. L. M. Donkers (eds.). Springer. pp. 72–83. CiteSeerX 10.1.1.81.6817. ISBN 978-3-540-75537-1.*

[2]. Billings, S. A. (2013). Nonlinear System Identification: NARMAX Methods in the Time, Frequency, and Spatio-Temporal Domains. Wiley. ISBN 978-1-119-94359-4.

[3] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2. (Publisher link).

[4] Van Rossum, G. (2020). The Python Library Reference, release 3.8.2. Python Software Foundation.

[5] https://github.com/KeithGalli/Connect4-Python.git

[6] https://github.com/floriangardin/connect4-mcts.git

[7] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf