Connect Four

Feng Wang

fwang26

1. Introduction

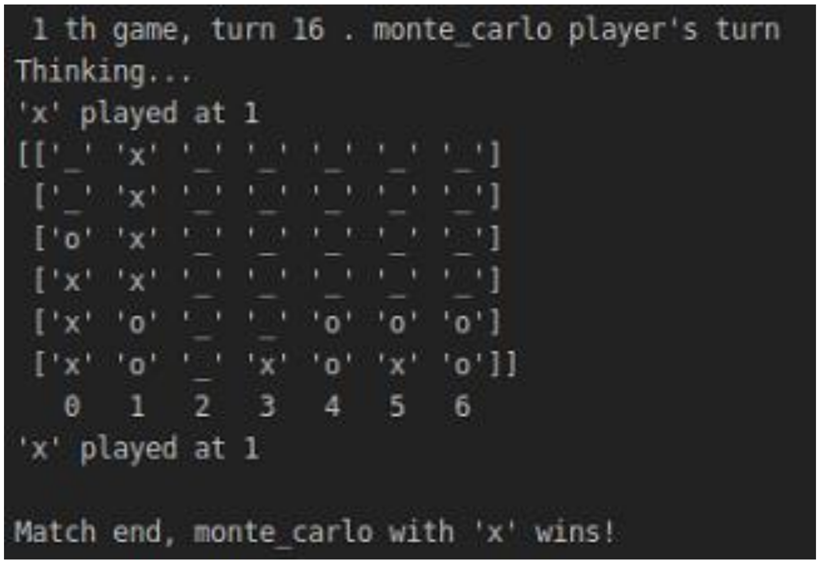
This project implements two artificial intelligence agents on the game connect four (https://en.wikipedia.org/wiki/Connect_Four). This is a game between two players, starts with a 6*7 board, each player at his turn may drop one disc with his color into one of the columns, and the disc will fall to the bottom, beyond previously dropped discs. Any of the players who complete four of his discs in a row, column, or diagonal wins. In my project, after a player make his decision, the disc may fall to other column with 20% probability (equally distributed in each other available columns), and the size of the board and the winning number can vary.

The first agent uses Monte Carlo tree search, and the second agent use convolutional neural network to make decision, and a depth-2 minimax tree to advise (catch immediate win and avoid immediate loose). Both agents have about 99 per cent win rate against random player in all instance sizes, while they lose only when they made one or two random moves while the random player accidentally made several correct decisions.

GitHub link: https://github.com/wfwf10/Connect-four

2. Game

The original setting of the game is an empty board with 6 rows and 7 columns, each hole can hold one disc. Two players with two different color of disc play one of his discs at a time, one by one. He can drop the disc into any of the columns that is not full, and the disc will pile one row beyond the other discs in the column. The first player who has 4 of his discs in a row, column or diagonal wins. I represent this as a 2d NumPy array. Empty slot is '_', the first player is 'x', and the second player is 'o'. Each column whose top row is '_' is one of the valid options. If one player has 4 piece in a line, he wins with 1 point, and if the board is full and no one wins, the game is draw, and each player has 0.5.



```
 1 th game, turn 16 . monte_carlo player's turn
Thinking...
'x' played at 1
[['_' 'x' '_' '_' '_' '_' '_']
 ['_' 'x' '_' '_' '_' '_' '_']
 ['o' 'x' '_' '_' '_' '_' '_']
 ['x' 'x' '_' '_' '_' '_' '_']
 ['x' 'o' '_' '_' 'o' 'o' 'o']
 ['x' 'o' '_' 'x' 'o' 'x' 'o']]
   0   1   2   3   4   5   6
'x' played at 1

Match end, monte_carlo with 'x' wins!
```

The picture beyond is a typical turn of Monte Carlo agent playing against random agent. Monte Carlo player with disc ‘x’ played at the number 1 column, the disc dropped on the top of the column, achieved 4 ‘x’s in a column, and won the match.

In my code, each of the two players can be Monte Carlo agent, CNN agent, random agent (make totally random decision), and human agent (the user watches the board, and type in the column he chooses in every of his turn). The parameters that can vary includes the number of rows R, the number of columns C, and the number in a line to win win_num. All player except CNN agent accepts all combination of parameters so that *min(R, C)>win_num*. I trained CNN with these seven combinations and saved in the project for you to try, and you can also train different models with different parameters (training parameters are automatically set, you only need to decide R, C and win_num). These seven CNN options are:

[R, C, win_num]: [8,10,5], [5,10,4], [6,7,4], [10,5,4], [5,5,4], [6,7,3], [5,5,3]

In the program, I used numpy[1], copy[2], math[3], matplotlib[4], time[5], and keras[6].

3. Tree search

Pseudocode:

```
Check if state is valid
Check valid moves
For each valid move:
    For each iteration:
        Random pick each move until one player win or reach maximum depth
        Record win (or lose or draw) point and depth of this search
Return expected win point, total number of nodes in search, and the move with most win point
```

Branching factor starts with C (for example, 7), and decrease when a column is full. The number of nodes at each depth is close to C^{depth}, while Monte Carlo search only pick one at a time. It randomly sub-sample from possible options, and I limit the maximum of times it searches in each child, and the maximum depth (2 * win_num). I use uniform distribution sub-sampling, and if it reaches limited depth, the leaf will count as draw (0.5 point for each player).

4. Machine learning model

The machine learning model includes an CNN model and a minimax tree. The CNN model includes one convolutional layer, and two dense layers. The number of channels and nodes depends the number of board rows R, the number of board columns C, and the number in a line to win win_num, so the code works for different games with different problem instances. The model is constructed by the codes below (with Keras[6]):

```
model.add(Conv2D(R*C, kernel_size=win_num, activation='relu', input_shape=(R, C, 1)))

model.add(Flatten())

model.add(Dense(R*C, activation='relu'))

model.add(Dense(C, activation='softmax'))
```

When machine learning agent make decision, another minimax tree will check two steps ahead before CNN. The 2-depth minimax tree will search for one move to win and play, otherwise search and exclude moves that opponent player will win within one move. Then, the agent will pick the move available with highest CNN probability.

In this agent, only CNN weights and bias are trainable. I run two Monte Carlo agents with much more sampling and 50% of mismove rate (save the original decision) against each other for 100 games for each problem instances, then train and test CNN with the records of the 100 game repetitions. The loss of updating is the categorical cross entropy against the decision of cleverer Monte Carlo agents. For each move in training and testing data, ‘x’ and ‘o’ are flipped if it’s the ‘o’ player’s turn, and then ‘x’ is 1, ‘o’ is -1, and blank is 0. This 2-d array will reshape to (R, C, 1) to be the input of CNN. I tried different number of layers, channels and nodes, and the one defined in the code above turns out to best fit the 7 different problem instances I tried.

5. Baseline opponent

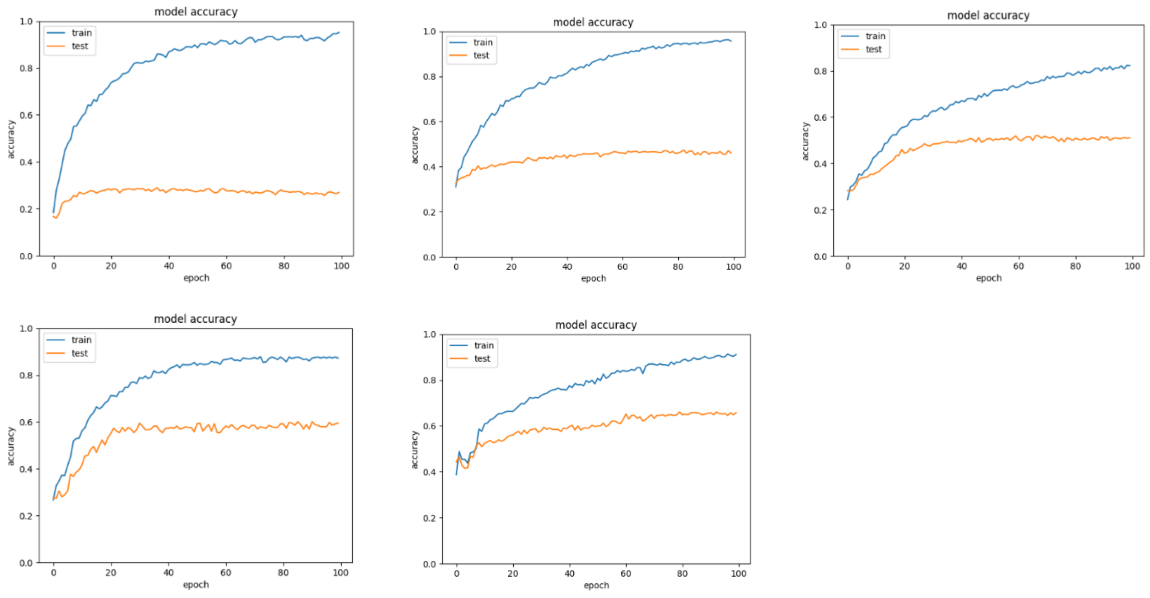
The baseline opponent is uniformly randomly choosing from different available columns. If there’s no 20% random failure (fall into unwished column) when each player makes his decision, the baseline opponent will never win (Actually as a human, I can’t win the two AI agents either). But the baseline opponent has 1% chance to win, when there’s randomness in the game, and AI made 1 or 2 random moves while baseline opponent accidentally makes several correct moves.

6. Experimental results

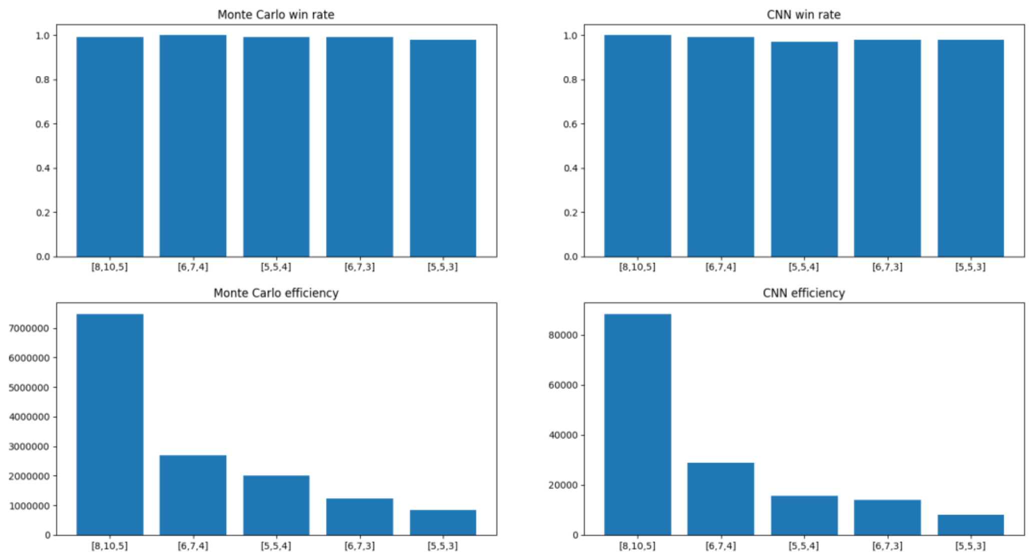
I’m the only member, so there’s one single subsection. The configuration is explicitly explained above. The 5 different problem instance sizes are:

[R, C, win_num]: [8,10,5], [6,7,4], [5,5,4], [6,7,3], [5,5,3]

For each instance size, I ran 100 games, 75 for training and 25 for testing. 100 updates for each is shown in the learning curve below:



In the plot below, Monte Carlo on the left only uses tree search, CNN on the right uses both CNN and minimax tree search. The win rate above is against random baseline, while efficiency below is only measured by the nodes searched in tree search component. Each element in x axis correspond to the 5 problem instance sizes, and each of them run for 100 game repetitions.



7. Conclusion

The most significant result is that the two agents have high wining rate against both random baseline and human. The best configuration is explained above, and Monte Carlo with 1000 search step is the best. The most challenging part is to write everything from nothing. If I was to continue, I will apply reinforcement learning on machine learning part to learn by itself, while CNN is just learning from Monte Carlo tree search result, and advised by another minimax tree.

Bibliography

1. Oliphant, T. E. (2006). A guide to NumPy (Vol. 1). Trelgol Publishing USA.

2. https://docs.python.org/2/library/copy.html

3. https://docs.python.org/3/library/math.html

4. J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

5. https://docs.python.org/3/library/time.html

6. https://keras.io