Monte Carlo Tree Seach For Go AI

Abstract:

Go:

Go is a strategy boardgame between two players where the aim is to occupy the most area of the board. The players do this by taking turns placing down stones with their respective color.

Go has been a very attractive case study when it comes to game AI. This is because there is an almost infinite amount of board states that are possible. This makes it so that simulating every action, and "Solving" the game would require an absurd/impossibly large amount of processing power.

(For a 19x19 board of go there is 1.2*10170 states. There are 1080 atoms in the observable universe!)

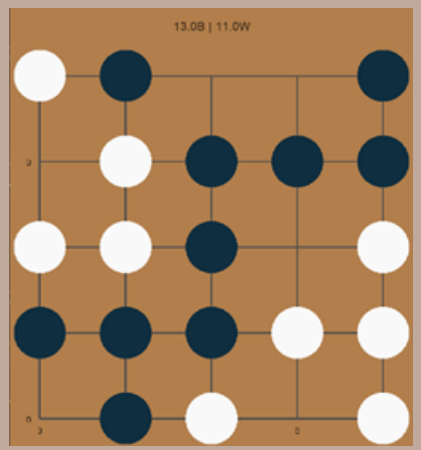
The most popular version of implementation AI in go is AlphaGo, which was developed by Google and uses deep learning to gradually get better and better.

This project was aimed to investigate how the Monte Carlo tree search algorithm, and alterations of rollouts, would work as a go AI. The final result was two variations of the MCTS that played the game at differing levels.

Introduction:

This project investigated the use of the Monte Carlo tree search (MCTS) algorithm in the classical boardgame, Go. The investigation of alterations of MCTS in the form of altering the rollout method used in the algorithm was the focus of the research projects.

The alternate rollout method uses a Convolutional neural network (CNN) to process the board state and make a guess for the best moves in that given scenario. The games were played on a 5x5 board due to time constraints concerning the creation of sufficient training data

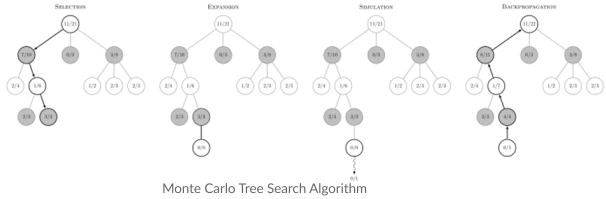


Game from light rollout version versus random moves

Method:

Monte Carlo Tree Search uses the principles of large numbers, randomness, and probability calculations, to limit the number of simulations that are used to assess an action. It does not give an exact solution to what action is best but gives rather a probability of any given action being good.

MCTS is an algorithm that traverses and expands a node tree. It works by following 4 steps



Monte Carlo Tree Search Algorithm

Selection: Selecting which node to traverse. This is done through a formula that considers the amounts of visits a node has recieved, and how rewarding that node was. The node with the highest

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

Formula used for tree traversing, ω: wins, n: visits, c: exploration coefficient, N: number of simulations from start node

Expansion: After finding a leaf node with no other leaf node, we expand the tree. This is done by adding a leaf node to the tree for each action which is possible from this state.

Simulation: Newly created leaf nodes are used as starting points for simulations. The simulation is called a rollout and completes a game from the nodes state. How this game is completed must be decided using a policy. The policy that was used before introducing a CNN was a compilation of random legal moves (Light rollouts). Later, the rollout policy was changed to use a CNN to evaluate the board state and make a prediction of what the best move was (Heavy rollouts). This CNN will be covered more extensively later.

Back Propagation: When the rollout is completed, the state of the game that was played to completion was recorded, and the data was sent throughout the parent nodes and updated the values of all of them. Doing this over a long period of time gave an evaluation of the different nodes' probability of leading to a victory.

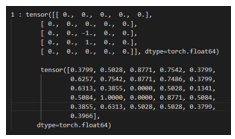
· How it was implemented

The MCTS was implemented from scratch. The GO environment used in the project was given as course material from the professor. The rollouts for the first version of the AI were light rollouts. This was fast and allowed for roughly 2 000 simulations each turn.

· What was the CNN used for

Later on in the project, a supervisor recommended the use of a CNN. The suggested use was to make the rollouts more consistent than just choosing random action until the game was finished. The CNN would take in a board state as a X input and give a probability distribution of actions to take from that position. This probability output would dictate which action was used in the rollout. This would continue until the game was completed.

· How was data fed to CNN



Training data for CNN

The input data was a 5x5 matrix representing the board state. The output was more difficult to determine. The CNN ended up using the total visits of each action from a given board state as the output. Given the principles of MCTS this seemed as a reasonable fit, as a more "correct" action would be visited more often. The dataset was created by running the original version of the MCTS with light rollouts and saving all the game states of a game, and all the visits of each action from that state.

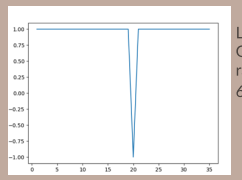
This ended up being about 7 300 000 rollouts, which amounted to roughly 10 000 datapoints. The CNN only had an accuracy of 25-30% for accurately picking the best move after training on the dataset.

Results:

· MCTS light rollout

When then light rollout version was limited to roughly 2 000 simulations each turn, it was able to win 99-100% versus a player who choose random moves. The program was also tried against online computers, but the environment was using other rules than the common online ruleset, so the program, and the online computer would often dispute who won the match.

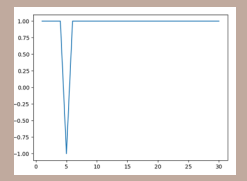
It also lost some matches against the online computer, where the program would make early moves that are considered nonstandard. In go there is a large emphasis on controlling the middle of the board, especially with the small board size used in this project. The program still sometimes suggested moves that where at the edges of the board as starting moves. These were almost always the matches the program lost.



Light rollout 35 Games against random moves 600 simulations

· MCTS heavy rollout

The implementation of the CNN made it so that doing adequate simulations to get decent results was in practice impossible. The rollouts 3 times as long as light rollouts, and produced the same, or worse results when compared.



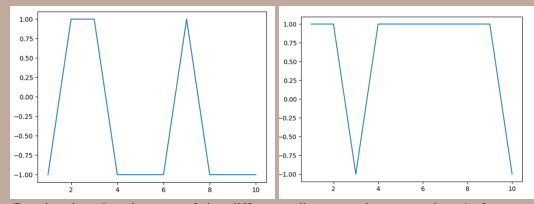
Heavy rollout 30 Games against random moves 600 simulations

Rollouts table:

Simulations	Light (sec)	Heavy (sec)
50	1.32	3.93
100	2.33	6.11
500	13.48	26.53
1000	24.75	54.36
2000	42.25	101.73
4000	85.32	194.12

Table showing how much time the diffrent rollout methods used on simulations

Rollouts Verus:



Graphs showin winrates of the diffrent rollouts against eacother. Left one heavy rollouts starts. (1 = Heavy win, -1 = Light win). Right graph light rollouts start. (1 = Light win, -1 Heavy win). Both were given 20 sec to find the best action

Discussion:

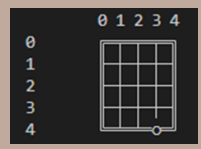
The CNN implementation was too slow and did not give the quality of moves that would make it useable.

This might be because of the implementation not being optimized enough, or just that the dataset was too small for the model to learn anything from it. This could have been improved upon by making more data or renting a supercomputer to run the games.

A common mistake in the CNN predictions was subpar starting moves. Because of the small board size, a bad starting move was a detrimental error. A possible alteration that would help this was adding in another term into the traverse formula that contains some sort of bias to moves that are considered "good" starting moves.

Another possible solution to the problem could be the Y values fed to the CNN. Since the Y values for the training set was the number of times that an action was visited. This is in "theory" a valid assumption as the more successful actions should have been visited more, but because of the small data size, this assumption might not be valid. Also, it might have been smart to lower the exploration coefficient when making the training data, as actions that did not do well, was still visited quite a bit.

If the program will be worked upon more, the creation and processing of training data would be redone. Also, the CNN would have to be more efficient, to make it able to simulate more than just a couple hundred games.



Bad starting move made by heavy rollout (1 000) simulations

Conclusion:

The creation of a GO AI was successful, however the success rate of the AI itself is more debatable. The light rollout version seems more useable as it is more consistent with less time per turn, than the heavy rollout version.

The light rollout version is also scalable to the other board sizes, while if the heavy rollout version was to be used, it could only be used on the 5x5 board size due to its training data.

This was an interesting project where I gained knowledge about machine learning, general planning, and time management.