Reinforcement Learning 2020
Assignment 2: Adaptive Sampling

Meng Yao (s2308266), Shutong Zeng (s2384949), Auke Bruinsma (s1594443)

25-03-2020

Abstract

We have written a python implementation of Monte Carlo Tree Search (MCTS), and are able to manually play against it in the game Hex. We let it play against the alphabeta algorithm we wrote in the first assignment as an experiment. To see what the effect was of tuning the number of simuliations N and the UCT exploitation/exploration parameter, we let it play against itself with different hyperparameter settings. We found that increasing the number of simulations increases the performance of MCTS and that C_p has to have a value which makes sure there is a good balance between exploiting good moves and making new moves.

1 Introduction

This is our report on the second assignment (adaptive sampling) of the course Reinforcement Learning. The main objective of this assignment is to implement a Monte Carlo Tree Search algorithm. From now on we will refer to this algorithm as MCTS. Contrary to the first assignment, where we had to implement minimax, alphabeta and iterative deepening & transposition tables (ID&TT), MCTS does not have a heuristic evaluation function. This is really interesting. As is already written in the book, the main paradigm of the reinforcement learning community has for a long time been alphabeta. When it turned out that algorithm such as MCTS are able to create stronger AI's, a new research area was opened. An important consequence of using MCTS instead of alphabeta, is that moves which may give results in for example 20 turns later, will be discovered by MCTS, but will most likely not be discovered by alphabeta. This is why alphabeta is really effective with Chess, but not with Go. Go has a much larger state space then Chess, so alphabeta is extremely likely to miss a lot of moves which might have great potential in the late-game. MCTS, however, is able to find these.

We will first explain the MCTS algorithm and its main constituents. We have divided our code in serveral parts. First there are the two python files called `hex_skeleton.py` and `node.py`. `hex_skeleton.py` is just the file we started with in the first assignment. We added a couple more functions to the class, so it is important to use this file. We create another class called which is contained in `node.py`. We will not explain each function and variable of this class separately, but when we use it from another file, it will be explained then. Besides these two classes there are 3 other files: `mcts_hex.py`, `experiment.py` and `tune.py`. Each file is for one part of the assignment. We have given them the same names as in the assignment instruction `pdf` file. For each file there is a section reserved in this report.

2 Monte Carlo Tree Search

The algorithm consists of 4 parts:

1. Select
2. Expand
3. Play-out
4. Back-propagate

As in the first assignment, we still use the game Hex as our play field. Since we already explained how that game works in the report of the first assignment, we assume we do not have to do that again and everyone is now familiar with the rules. Each board state of this game has to be thought of as a node, which is part of a tree. So the rootnode is a blank field which is the highest node of the tree. When a move is performed in Hex, a children node is added to the tree, and the rootnode and children node are connected. Since the algorithm is already thoroughly explained in the book and on the blog post we received, we will give a very

broad description. In the selection part, MCTS selects a node. It does this by using a selection rule. The most used selection rule is the UCT selection rule:

$$\mathrm{UCT}(j) = \frac{w_j}{v_j} + C_p\sqrt{\frac{\ln v}{v_j}} \tag{1}$$

The node which gives the highest value from this function will be selected. We will explain all the variables used in this function and how you can tune this function in the back-propagate section.

When a node is selected, the node will be expanded and new children are added to the tree. Next a play-out is performed. The play-out we implemented is a random play-out, so the AI and the simulated player both make random moves until someone wins the game (Since the game is Hex a draw is not possible). You might think this randomized playing could be more efficient; this is true. AlphaGo uses neural networks and pattern databases in this part of the algorithm. A win for the AI means $w = 1$. A lose for the AI means $w = -1$. A draw would mean $w = 0$, but that is not possible in our game. Now, in the back-propagate phase. This w value will be back-propagated back all the way to the rootnode, along with the visit rate. Each node has attributes, such as the win rate and visit rate. When a random playout is done, the corresponding node and all its parent nodes until the rootnode will be updated. Their winrate w will increase or decrease depending on the win or loose. Their visit rate v will always increase. When we now look at equation of the UCT selection rule, there are only two variables left to explain, v is the visit rate of the parent, and v_j of the current node. C_p is the tuneable exploration/exploitation parameter. The first term in the equation describes the winrate of the node, so this can be referred to as the exploitation part of the selection rule. When a node has a high win rate it makes sense to exploit this part of the tree. The second term (the part inside the square root) describes the 'ńewness' of a node. This is the exploration part of the formula. Parts of the tree that hasn't been explored a lot make sense to explore more; there might be interesting moves there. So the term C_p determines how much you want to exploit/explore. A high value for C_p means much exploration while a low values means much exploitation. This term needs a lot of experimentation to see what is the best value. This is very globally a description of the MCTS algorithm. In the next part of this report we explain how we implemented this algorithm in python, and where each part of the algorithm is implemented.

3 MCTS_hex

We wrote our python implementation of the MCTS algorithm in `mcts_hex.py`. In the function `MCTS()` we sort of merged the selection and expansion part of the algorithm; in our thinking this made sense at the moment of writing. In this function the following things happen:

- The board is copied, so moves can be played out without really playing them. This way the AI can simulate a game and in the end return the best move it came up with.
- There is the while-loop `while action != True`. This statement can be probably be optimized a bit more, but essentially this happens: New nodes are created when a child does not have any children and it needs to be expanded. When all children nodes of a specific node already have a visit value larger than 1, it selects a node to go to.
- When a node is selected and expanded. The play-out phase begins. This is done using the function `move_check_win` from the node class `node.py`.
- Finally, the back-propagation phase is entered and all win and visit values are updated. This is done using the function `node.update`.

In our code we tried to keep as much as the names and code the same as the pseudocode given to us from the book for the sake of clarity. When running the file, the terminal will allow you to play against the MCTS algorithm as opponent.

4 Experiment

After having written the MCTS algorithm, we were instructed to do some experiments with it. The goal of this part is to let the MCTS play against the ID&TT algorithm from the first assignment and determine the ELO rating. However, in the first assignment we didn't succesfully write the ID&TT part of the assignment. So, we contacted the TA's of the course and were allowed to let the MCTS play against a regular alphabeta algorithm instead. Of course, the alphabeta algorithm without iterative deepening and transposition tables is expected to perform worse, but it is still a good experiment. The code for this part is contained in the file `experiment.py`.

There is no new code in this file; the alphabeta algorithm is taken from the first assignment and the MCTS algorithm has already been explained. We did the experiment on a 4 by 4 grid since alphabeta still performs well on such a field. We kept the C_p equal to 1 and changed the number of simulations for the MCTS algorithm and the search depth for the alphabeta algorithm. The question 'How many games should be played for the rating to stabilize?' was already covered in the first assignment and is equal to 12. Below are the results of the experiments we did.

Round	AB3 vs. MCTS 10	AB3 vs. MCTS 100	AB4 vs. MCTS 10	AB4 vs. MCTS 100
1	MCTS 10	MCTS 100	MCTS 10	MCTS 100
2	MCTS 10	MCTS 100	MCTS 10	MCTS 100
3	MCTS 10	MCTS 100	AB4	MCTS 100
4	AB3	MCTS 100	MCTS 10	MCTS 100
5	MCTS 10	MCTS 100	AB4	MCTS 100
6	AB3	MCTS 100	AB4	MCTS 100
7	AB3	MCTS 100	MCTS 10	MCTS 100
8	MCTS 10	MCTS 100	AB4	MCTS 100
9	MCTS 10	AB4	MCTS 10	MCTS 100
10	AB3	MCTS 100	MCTS 10	MCTS 100
11	AB3	MCTS 100	AB4	MCTS 100
12	MCTS 10	MCTS 100	AB4	MCTS 100
ELO	23.6, 26.3	22.1, 27.9	27.2, 22.8	15.1, 34.9

Table 1: AB3: Alphabeta with search depth 3. AB4: Alphabeta with search depth 4. MCTS 10: Monte Carlo Tree Search with 10 simulations. MCTS 100: Monte Carlo Tree Search with search depth 100. The elo rating is determined in the same way as the first assignment using the libary `trueskill`.
The first rating coresponds the the first player names in the top row of the table.

Discussion: You can see that when the MCTS does enough simulations, alphabeta only has an occasional win. When MCTS does not have enough simulations, it performs worse, as expected.

5 Tune

The last part of this assignment is the tuning part. MCTS has two parameters which can be tuned: the UCT exploration/exploitation parameters C_p and the number of simulations N. The goal is to perform a hyperparameter analysis for these parameters. Since in the assignment it does not specifically state what this hyperparameter analysis is, we came up with our own analysis. We will let two MCTS algorithm play against each other, with different settings for N and C_p. The code for this part of the assignment is contained in the file `tune.py` and also does not really contain any new code; it is just two MCTS algorithms playing against eachother. See below for the results.

Results & analysis

- in the file `tune.py` there are 4 variables which allow you to tune the 2 hyperparameters for the two AI's. We played a lot with these values and in the next points we will tell you our findings.
- When increasing the number of simulations for one of the AI's, it is expected that that one will perform better. This is indeed what we observe.
- When keeping the number of simulations the same but increase C_p we do not really observe any noticeable changes. For absurdly high values of C_p or $C_p = 0$, it performs worse, however.
- Because this algorithm does not search every possible move, it also possible to test on grids larger than 5x5 (contrary to first assignment). It is interesting to observe that, while the algorithm performs random moves in the play-out phase, the end-game looks always a bit similarly. This is because of all the random moves, only the best move is remembered and played. If there are enough simulations done, this best move will have a high probability of being the same one on each different kind of board state.

6 Discussion & Conclusion

We succesfully wrote a python of implementation of Monte Carlo Tree Search. We let it play against itself and against a simple alphabeta algorithm. We found it really interesting that an algorithm such as MCTS, which does not have a heuristic evaluation function, is able to produce such a strong opponent.