My fantastic report for Assignment 1 - EDAP01

Sophia Kovalenko so4816ko-s

February 7, 2024

1 Description of the solution

In this section, I will describe my solution. Starting with the approach, the implementation of the algorithms and the choice and description of the evaluation function.

1.1 Approach

My approach follows the lecture notes procession. First, I implemented the general programming framework for adversarial play. I then coded the function "score(s)" where I implemented the Minimax algorithm. I was then faced with a big search complexity and an infinite algorithm, I added a depth but having no evaluation function as my goal was first to make it work.
After making the algorithm work, I added $\alpha - \beta$ pruning.
It was finally time to find the evaluation function.

1.2 MiniMax implementation

The Minimax algorithm can be found in the functions "score()", "min_player()" and "max_player()", lines 95-126 in the code.
The principle of the algorithm, is that it will "try" to make a move and then pretend to hand the game over to the other player. To implement this, I am calling the function "env.step(action)" which is used in the *gym* environment. I was rapidly faced with the fact that this function updates the whole environment and its board without giving the possibility to rewind. I decided, instead of creating a function "unstep" which would have taken a lot of code with little advantages, to create copies of the environment at each step. I used the function deepcopy from the package copy.
In my opinion, it is not a very clean way to do it. Moreover, for the game Connect-Four the memory complexity is tolerable, but it would probably be less efficient if used on bigger games like Chess.

Depth I am using a depth variable to limit the depth of my algorithm. It has now a maximal value of 4, this can be changed in the constant MAX_DEPTH line 91. Using a depth of 5 makes it go up to 10 seconds between each move, which is not an acceptable computation time, whereas with 4, it is around 1 second and still performs well against the server.

1.3 Alpha-beta pruning

$\alpha - \beta$ pruning can be found in the functions "min_player()" and "max_player()" lines 98-126. The Minimax algorithm uses it in order to decide for the best move while stopping the search in a branch if the score gets too bad.

1.4 Evaluation function

The evaluation function can be found in the lines between 128-186. It consists of 2 functions: the main "eval(env)" function and the helper "evaluate_score(block)" function.

eval() function This function iterates over the whole board 4 times (rows, columns, diagonals, reversed diagonals), and isolates blocks of 4 cells. It then calls the function "evaluate_score()" on each block of 4 cells. Each time, it updates the *score* variable with the score of each block, this variable is then returned at the end. Apart from the scores assigned by the "evaluate_score()" function, I added a small reward when the agent plays the middle columns for the first row, as it allows more winning combinations.

evaluate_score() function This function assigns a score to each situation possible which the block of cells can be in. As the cells can have values between -1, 0 and 1, the sum represents well the state of the block if there are 3 or 4 discs in a row. There can be several ways of obtaining the value 2 and -2, so one must also count the number of free spots available (see the "np.count_nonzero()" condition).
Moreover, when having 3 discs, it is important to assign a score only if there is a spot available (see the "minimum" or "maximum" condition) to have a possibility of connecting 4. It also gives a small reward when blocking the adversary.
I first assumed that it was better to win if there was the possibility, than to restrict the adversary. The shorter the game is, the less chances there is for the adversary to win. In this motive, the score for placing a winning disc was higher than the (negative) score given for letting the server win. This made the server often win, which was not a satisfactory option. This is the reason why the scores are not symmetric between the loosing and winning states.
At first, the score assigned were $\pm 10, \pm 3, \pm 2, \pm 1$. But the relative distance between the scores was not enough. To work well, the difference must be big and the winning state must have a very high reward to make sure that it will be the branch selected.

2 How to launch and use the solution

To launch the game, you will need python and several packages. Requests, numpy, gym, pygame and argparse that can be installed with the command "pip install".
The code can be found in the file "skeleton.py". To launch the game, the files "connect_four_env.py", "render.py" and "_init_.py" contained in the package envs must be present. Everything can be found in the template given.

To play, one must execute the command "python skeleton.py" with either -o or -l. -o plays against the course server. -l plays locally, against a human (you). The bot has 1 and you have -1. One can select a move by typing a number between 0 and 6 which represents the columns from left to right.

3 Peer-review

In this section, I will be commenting on the code of my fellow peer, Vivian Bai [vi0713ba-s]. I will describe it, compare it to mine and finally discuss the correctness of the implementation, it's efficiency and the potential improvements of her solution.

3.1 Peer's Solution

Vivian's solution is also implementing the Minimax algorithm using $\alpha - \beta$ pruning and an evaluation function.

The evaluation function (score_token_position()) goes over the board 4 times to isolate blocks of 4 cells. For every block, it calls the function "evaluate_block()" to attribute a score to each block.
A score is given if the player has either 4, 3 or 2 in a row (while checking for empty space for the last 2). It only attributes a negative score if the opponent has 3 in a row.
Except for the very high reward (9999) when the player wins, the scoring does not have a big difference between the 2 or 3 in a row.
Moreover, the "score_token_position()" attributes a score if the player puts discs in the middle column.

3.2 Technical Differences of the Solutions

Structure The first difference is the code structure for the Minimax algorithm. Instead of 3 functions, she created only one (min_max()) which has a supplementary boolean parameter which specifies if it is the minimizing or the maximizing player. It also does all the work, as the function "student_move()" only calls it instead of first doing a move.

Depth One other difference is the depth, Vivian's has a depth of 5. This means that it is more efficient than mine as with the same depth, mine takes more than 5 seconds to complete a move.

Evaluation function Her evaluation function (score_token_position()) privileges the middle column more than mine. She gives a higher reward and cares about the whole column, and not only the first row.
The scoring function (evaluate_block()) has less conditions. It does not give any (negative) reward if the opponent has 4 in a row, nor if it has 2 in a row.
Moreover, it does not give a high reward if the player obtains 3 in a row, only 5.

Randomization Finally, she also performs a randomization before iterating over the moves in "min_max()". This maybe helps to reduce the bias, but it is hard to check.

3.3 Opinion and Performance

Which differences are most important? The most striking difference is the structure, but it should not affect performance of the algorithm so it is probably not the most important.
Even though the scoring has less conditions and only penalizes for the opponent's win one time, it seems to be enough.
As said higher, the depth is an important one which highlights the difference in the efficiency between our algorithms.

Is there a potential improvement? The middle column is privileged by the bot because it gets a reward when discs are placed there. This makes the bot put them in that column even when there is no space anymore to Connect4. This could be refined, especially that it gets quite a high score (number of discs in the middle * 6).

How does one assess the performance? I am assessing 2 types of performance: time and victories. I added 2 lines in her code to be able to compare the computation times using the package time.
Victories are harder to assess. But one can estimate it with the "streak_max" value in the server statistics.

Which solution would perform better? Both solutions are great. I do think that Vivian's is better because it is more efficient and can achieve a bigger depth for a acceptable computation time.
Vivian's solution has a higher maximal winning streak (around 50 whereas mine has around 30), which probably means that it has a higher winning chance than mine.

4 Interaction summary with an AI chatbot

In this section, I will discuss the code that ChatGPT, an AI chatbot, produced for this assignment. I will then discuss the differences with my implementation and its performances.

To run the solution, one must have the same packages as explained in section 2. This solution also uses the environment provided. To run it, write "python gpt_skeleton.py" and -o or -l to play online against the server or locally against a random agent.

4.1 Interaction prompts

The prompts that made the chatbot to produce code that could run the 4inarow game against the server.

The prompt I gave it: "I want you to do an assignment. Your task in this assignment consists in writing a program playing and winning a Connect Four game (aka 4-in-a-row). Here is the environment: *I inserted the code for the environment* And here is the skeleton for the player, I want you to fill in the student_move() method and modify the rest accordingly: *I inserted the skeleton's code*"

Its answer: "Certainly! I'll provide you with an implementation of the student_move() method using a basic min-max algorithm with alpha-beta pruning. This algorithm will search for the best move by simulating possible future states up to a certain depth. Here's the modified student_move() method
code
Note: The provided algorithm is a basic implementation, and you might need to fine-tune the evaluation function for better performance. You can experiment with different evaluation functions and adjust the depth of the search based on your system's capabilities."

Changes made to make the code compile: I had to change some lines (they are mentioned in the code), because GPT had some troubles with understanding the env.step function and the tuples it returned. Error messages were "ValueError: too many values to unpack (expected 3)" Also had trouble with the reset function, I added "board =" because of the error message "TypeError: OrderEnforcing.reset() takes 1 positional argument but 2 were given"

Making the game work When compiled, the game did not work, as it finished after the first student's move. The bot's solution did not theoretically make the moves when calculating, and did not change the player when going between the min and max player. So it ended with only 1's on the first line. I had to ask again.

Prompts and discussion to force ChatGPT to get to a working solution
Me: "It does not work, your game does not rewind when calling the step function. Please update your solution."
Bot: "I apologize for the oversight. The issue is related to the fact that the env.step(action) function modifies the internal state of the environment, and the env.reset(state) call is not sufficient to revert the state to the original one. We need to create a copy of the state before performing the step. Here's the corrected implementation:
code
Now, the state_copy is used to keep the original state intact before making a move, and the environment is reset to this original state after evaluating each possible move."

I again had to make the same small corrections as before to make it compile, but it ended up working.
Note: I later modified my previous prompt by omitting "your game does not rewind when calling the step function" to see if the ChatBot would have found the problem by itself. It gave me a solution where the only change was that

instead of the "return value" in the if-statement with alpha and beta, it put a break. So it did not see the problem with the board that was not properly resetting.
At this point, the evaluation function was extremely simple. It was simply summing all the values of the cells and scaling it, which did not evaluate anything useful. I had to ask ChatGPT to improve the evaluation function.

Me: "Your evaluation function is too simple, please improve it to be able to win against another agent."

ChatGPT: "Certainly! Let's improve the evaluation function to make it more effective in guiding the AI to win against another agent. In Connect Four, key factors for evaluation include the number of connected discs, potential threats, and blocking opponent's moves.
Here's an improved version of the evaluation function: *code* "

After this exchange, GPT's version was acceptable. It attributes score depending on the state of each row, column and diagonal in the helper function "evaluate_line()" called in the evaluation function "evaluate_state()". It checks for 4, 3 and 2 (student's and opponent's) discs in a row.

4.2 Difference to the own solution

The overall approach is very similar to my implementation. It also uses a Minimax algorithm with $\alpha - \beta$ pruning and an evaluation function.

Minimax The main difference in the Minimax algorithm is at the stopping state: it always returns the evaluation value, not the "reward" variable, even if it got to reach the end of the game. Which is not a problem with a good evaluation function, but if it is not, it can lead to some problems.

Evaluation function In the evaluation function, the scoring is different. It has more regular intervals (multiples of 10), and there are no different values if either the player or the opponent is winning.
Moreover, in the 2 or 3 in a row condition, it does not test if there are empty spots in the row. Which can lead to poor games as the bot could place a disc where there is no possibility for winning.
Finally, it does not assign any score to place the discs in the middle column, which gives an advantage according to the Connect4 strategy.

Overall, it has small implementation differences, like the way of counting the discs in a row, or the use of internal function instead of having outside ones. But the main structure is the same.

4.3 Performance

It sometimes wins against the server, but it is not a 100% winning implementation. As GPT says "Feel free to experiment with the weights and scoring criteria to find the best configuration for your AI. Adjusting these parameters can significantly impact the performance of your Connect Four agent."

For the time performance, it is faster than my implementation because it is has a smaller depth. But if I increase the depth to 4 or if I decrease my depth to 3, they take the same time.