My fantastic report for assignment 1

Erik Dahlberg - er8251da-s

February 8, 2023

1 Description of the solution

My AI approach builds upon three actions. First, with alpha-beta pruning, the minimax algorithm is entered, which secondly calls a method for extracting rows, columns and diagonals on the board as arrays and thirdly passes these arrays to an evaluation function. The minimax algorithm is based on a recursive method which passes successive states of the connect4 board in every recursive call. The extraction method transforms rows, columns and diagonals into sets of arrays with length 4 (called cells). These cells are then evaluated by counting the numbers of own pieces, opponent pieces and zeros.

1.1 Minimax function

The mini-max method takes a state of the board, a search depth, and a boolean used for alternating between a maximizing and minimizing player.

```
def alphabeta(Board: ConnectFourEnv, depth, alpha, beta, maximizingPlayer, game_over):
```

Since the provided Connect4 environment gives a variable returning the state of the game with every move, the alpha-beta function also takes the parameter "game-over" as a convenient end requirement. Furthermore, to implement alpha-beta pruning, alphabeta() takes parameters alpha and beta.

Adopting a mini-max approach for a connect4 game requires iterating over and performing available moves, changing between players, and passing on new states of the connect4 environment. The implementation is the same for minimizing and maximizing players except for that the respective minimum or maximum value is saved after evaluation. Below is a code snippet showing the implementation of the maximizing-player call.

```
value = -np.inf
for move in Board.available_moves():
    board = copy.deepcopy(Board)            #Deepcopy of Connect4 Environment
    board.change_player()                   #Change player with each move
    st, res, done, _ = board.step(move)     #Make move
    new_value = alphabeta2(board, depth-1, alpha, beta, False, done)
        if new_value > value:
        value = new_value
```

Notice how the value is set based on a comparison between the recursive call and the value variable. This is what enables the minimax algorithm to return the move with highest value. Also note that the iteration is performed on available moves and not columns of the board, which prevents the algorithm from evaluating impossible game states.

Lastly, the minimax method returns the highest value of available moves. Therefore the call to the alpha-beta method needs to extract the move corresponding to this highest value. This is done by making the first iteration of available moves in the call to the alpha-beta method.

```
for move in environment.available_moves():
        environment.step(move)                #Make move
        values = alphabeta2(environment, 3, -np.inf, np.inf, True, False)
        if value < values:                    #Saves highest value
            value = values
            bestMove = move                   #Saves corresponding move
    return bestMove
```

1.2 Alpha-beta Pruning

Alpha-beta pruning is implemented by introducing alpha and beta values which keep track of the respective lowest and highest value at each depth of a recursive call. Since the minimizing and maximizing player alternates, the alpha-beta method breaks if the maximizing player finds a value larger than or equal to the current beta value or if the minimizing player finds a value smaller than or equal to alpha. Again, taking the example of the maximizing player, this is implemented by simply comparing the beta value to the max value of recursive calls. Also notice how the max alpha value is saved following the if statement.

```
if (value >= beta):
    break
alpha = max(alpha, value)
```

1.3 Evaluation

Evaluation is based on three assumptions. 4 in a row is worth substantially more than 3 in a row. 3 in a row is worth substantially more than 2 in a row. Opponent pieces are worth polar opposites of own pieces. My evaluation method takes arrays of length four and counts opponent pieces, own pieces and zeroes and then returns values based on the result.

```
def evaluateSum(window: list):
    ettor = window.count(1)
    nollor = window.count(0)
    minus = window.count(-1)
    if ettor == 4:                        #Win state
        return 1000
    elif ettor == 3 and nollor == 1:      #Potentially one piece away from win state
        return 100
    elif ettor == 2 and nollor == 2:
        return 5
```

```
elif minus == 4:                    #Game lost
    return -1000
elif minus == 3 and nollor == 1:    #Potentially one piece away from losing
    return -100
elif minus == 2 and nollor == 2:
    return -5
else:
    return 0
```

For clarification, 3 and 2 in a row requires zeroes to fill the remaining spaces. It is important to make values increase exponentially for consecutive pieces so the AI doesn't equate a board with a lot of 3-in-a-rows with one with a 4-in-a-row.

2 How to launch and use the solution

Prerequisites include:

- python3
- gym environment (provided for EDAP01-course at LTH)

The implementation is dependent on packages:

- gym
- random
- requests
- numpy
- argparse
- sys
- Connect4Environment from gym
- copy

To launch the code simply place the skeleton.py file in a workspace with the gym environment and launch with command: "python3 skeleton.py -l/-o/-s". "-l" initiates local play, "-o" plays against server, and "-s" display stats against server.

3 Peer-review

Daniel Björklund (da1452bj-s). We discussed the questions provided by the assignment. These questions were discussed on the 7th of Febuary. At that time Daniel's solution did not work, but he was still working eagerly on it. Therefore, I can not speak for the finished product. Regardless 3.1 is an outline of our discussion at that time with an introductory description of his solution.

3.1 Peer's Solution

Daniel's solution uses a minimax algorithm with alpha-beta pruning. In his call to the minimax algorithm he provides alpha and beta values, an array corresponding to the connect-4 environment state, a depth counter, and a boolean for alternating between maximizing and minimizing player. His evaluation is an iterative loop that returns higher or lower values based on consecutive 1:s and -1:s respectively in rows, columns and diagonals.

- **is the implementation correct?** No the solution is not correct.
- **how do you know it?** Daniel's solution fails in two ways. First of all it doesn't consequently win against the server. Second of all, it takes a long time to execute and, what I think, therefore sometimes produces weird results. To Daniel's defence it seems like the algorithm produces a win, draw or error. It never lost in our tests.
- **is it sufficiently efficient?** No not really. My code takes about 5 seconds to reach an end result. Daniel's takes about 20 if it reaches a result at all.
- **how do you measure that?** We measured that by timing the execution from call "python3 skeleton.py -o" til result.
- **does it contain alfa-beta pruning of a minimax tree?** Yes it does!
- **does it contain a heuristic evaluation of nodes in the tree?** Yes it does!
- **how does the evaluation function look like?** The evaluation function is very different from mine. Instead of saving arrays for evaluation Daniel's solution iterates over the entire board for every evaluation. Furthermore it iterates one time looking for double 1:s, one time looking for triple 1:s, one time looking for quadruple 1:s and then performs the same procedure looking for negative 1:s. Lastly, the evaluation does not take into account if three 1:s is followed by a negative 1 or a zero.
- **how good is it?** I would say the algorithm is about 80% done and thus not especially good (yet). It needs some optimization and should probably include following negative 1:s and zeroes in its evaluation.
- **is there an improvement potential in this implementation?** Yes, there are potential improvements in both the evaluation and minimax function.

3.2 Technical Differences of the Solutions

Daniels approach differs, mainly in its evaluation method, substantially from mine. As mentioned in 3.1 the consequences of Daniels evaluation approach are two-fold. Firstly, efficiency is halted due to the amount of looping required. Secondly, because zeros are ignored in evaluating consecutive 1:s and -1:s, the algorithm is not able to adequately evaluate some scenarios which should yield high values.

In his minimax algorithm Daniel did not use the provided Connect4 environment. Instead he made his own methods for creating boards, finding available

moves, and making moves. Although, this approach can work it has become unnecessarily complicated.

Me and Daniel has obviously tried to solve the assignment with the same methods performing the same task. However, our technical implementation was different. Daniel's solution is complex in comparison to mine and has as a consequence been difficult in finalizing.

3.3 Opinion and Performance

- **Which differences are most important?** The most important difference in our implementations is the evaluation function. With Daniel's solution ignoring zero's a lot of high value moves are dismissed. The second most important is the use and non-use of the provided connect-4 environment. However, I think that choice is more inconvenient for Daniel than destructive for performance.

- **How does one assess the performance?** Performance should be assessed on a performance vs efficiency basis. I think a poor implementation can perform well if it is willing to sacrifice efficiency for depth. A good implementation should be able to evaluate a move without excessive resource consumption.

- **Which solution would perform better?** In most cases, my solution probably performs better from both a performance and efficiency point of view. Based on the stats of our solutions, mine has a max win streak of above 20 while Daniel's is 5. Based on our tests, my solution usually take less than 5 seconds to beat the server, while Daniel's takes about 20 seconds to reach some result.

  On a last note, I don't want to seem too hard on Daniel. It would be interesting to compare our solutions when Daniel has finished since that wasn't possible this time.

4 Paper summary AlphaGo

Historically, the game of go has been difficult for AIs to beat human players in. The paper "Master the game of Go with Deep Neural Networks and Tree search" presents a new approach to the game based on a combination of deep neural networks and Monte Carlo tree search. The development led to the program AlphaGo which was able to defeat both the European and World Game of Go Champions.

Regarding functionality, the team behind AlphaGo combined deep neural networks and Monte Carlo tree search. The deep neural networks were trained on a dataset of human games. This dataset allowed the network to predict the best moves in a given position. Monte Carlo tree search was then used to search the game tree and determine the best move by simulating random plays. AlphaGo's neural network consisted of two layers. One including a convolutional layer and a policy network to predict the next move, and a second value network which predicted the outcome of a game from any given position. The neural networks were trained through supervised and reinforced learning. The algorithm uses the neural network for predicting heuristics in order to make

the tree search more efficient while still ensuring great detail. The authors found that the combination of deep neural networks and Monte Carlo tree search was what enabled the high performance of AlphaGo.

By testing the AlphaGo on human games the algorithm was shown to be able to predict the best moves in a given position. Against other programs AlphaGo had a 99.8% win rate. As previously mentioned, AlphaGo was also able to beat the European Game of Go Champion in a best of five game, where after it played against the world champion and won four games out of five. This made it the first computer program to defeat a human world-champion.

In conclusion, the paper "Mastering the Game of Go with Deep Neural Networks and Tree Search" depicts a fascinating new development in game playing by AI. The combination of the neural network and Monte Carlo tree search enabled the AlphaGo to beat the European and World Champion in a game which was previously very difficult for AIs to compete in. It will be interesting to see what other fields this type of aided Machine-learning tree search can revolutionize.

4.1 Difference to the own solution

The main difference between the above approach and my solution to the Connect4 assignment is using a neural network. My approach uses a pruning to efficient the minimax search. The AlphaGo uses a neural network to predict the best move and then performs searches based on that prediction. This way AlphaGo has a heuristic that guides it through the search tree, and makes it avoid calculation on obvious bad approaches. For instance, in my Connect4 algorithm, the calculation can occasionally be performed on every possible move, even when a 4-in-a-row is possible. This could be avoided with the use of a neural network.

4.2 Performance

I think the AlphaGo approach would result in better performance for my Connect4 AI. However, I am unsure how that performance increase would influence the win rate. On the one hand, with the aid of a neural network to predict the best move, the search could be performed with added depth. This is due to the efficiency increase when reading the board based on a provided dataset. In a sense, this provides a first level of pruning prior to alpha-beta pruning. On the other hand, the Game of Go is a more complex game compared to Connect4. Diminishing returns could be a real factor in adding deeper search, and in that sense the implementation of a neural network might not be worth the effort.

Lastly, I would like to investigate how the uncertainty of the neural network could affect performance. A strength of alpha-beta pruning is that it produces the same results as minimax but with greater efficiency. If a neural network is used to predict moves, what is the possibility of evaluation error? Furthermore, how does that uncertainty compare to the benefit of added depth?