EDAP01 - Assignment 1 Report

Filip Kalkan (fi1231ka-s)

February 8, 2022

1 Description of the solution (1-2 pages)

For the solution, I used the provided python skeleton. In the skeleton, the only modification made was that 'student_move(env)' calls an 'alpha_beta(...)' function. The function is an implementation of the minimax algorithm with alpha beta pruning. The function returns a good move along with the expected reward for the move. It finds a good move by representing the game as a tree. The tree consists of the current and possible future game states, represented as nodes, and moves, represented as arcs. The optimal move is such that it maximizes the reward, which is high for winning a game and low for losing a game.

In short, the algorithm runs a simulation of what might happen in the next few moves assuming that the opponent plays optimally. However, since simulating all possibilities is expensive the solution makes use of a maximum depth. The maximum depth tells the algorithm how many subsequent moves should be simulated. In this case, 5 was chosen as the maximum depth to guarantee winning over the opponent using the same algorithm with maximum depth 3.

The code for the algorithm can be found below.

```
1  #Initial call
2  alpha_beta(env, 5, float('-inf'), float('inf'), True, 0)
3
4  #The algorithm
5  def alpha_beta(env: ConnectFourEnv, depth, alpha, beta, maximizing_player, reward):
6      terminal_node = (reward != 0) or (depth == 0)
7      if terminal_node:
8          if maximizing_player:
9              return -reward, 0
10         else:
11             return reward, 0
12
13     if maximizing_player:
14         max_value = float('-inf')
15         best_move = "no_move"
16
17         for move in env.available_moves():
18             env_copy: ConnectFourEnv = gym.make("ConnectFour-v0")
19             env_copy.reset(board = env.board)
20             reward = env_copy.step(move)[1]
21
22             value = alpha_beta(env_copy, depth-1, alpha, beta, False, reward)[0]
23
24             if max_value < value:
25                 max_value = value
26                 best_move = move
27
28             if value >= beta:
29                 break
30
31             alpha = max(alpha, value)
32
33         return max_value, best_move
34
35     else:
36         min_value = float('inf')
37         best_move = "no_move"
38
39         for move in env.available_moves():
40             env_copy: ConnectFourEnv = gym.make("ConnectFour-v0")
41             env_copy.reset(board = env.board)
42             env_copy.change_player()
43             reward = env_copy.step(move)[1]
44
45             value = alpha_beta(env_copy, depth-1, alpha, beta, True, reward)[0]
46
47             if min_value > value:
48                 min_value = value
49                 best_move = move
50
51             if value <= alpha:
52                 break
53
54             beta = min(beta, value)
55
56         return min_value, best_move
```

In the code snippet the alpha-beta pruning is shown on lines 29-32 and 53-56. This pruning makes sure that nodes which have no relevance for the outcome of the exectuion are not explored. The variables alpha and beta, respectively represent the minumum reward that the maximizing player can get and the maximum score that the minimizing player can get. For instance, if a player is being maximized and the evaluation of a move is greater than the maximum score that the minimizing player can get, we know that the opponent cannot outperform this move. This means that no more nodes need to be explored at this level and branch of the game tree.

On lines 25-27 and 49-51 the essense of minimax is shown. For instance, in the first of these occurences it is made sure that the maximum reward along with the move yielding that reward is stored. This means that the reward

maximizing move is eventually found and returned.

2 How to launch and use the solution (as much as needed)

In order to run the solution, a number of of dependencies have to be installed. The dependencies can be installed using pip or conda.

- gym
- numpy
- requests
- Pillow
- Pygame

Furthermore, the python files "connect_four_env.py" and "lab1.py" are needed.

When the dependecies have all been resolved the solution can be run by executing lab1.py with the "-o" flag in order to play against an online server.

3 Peer-review (1-2 pages)

When discussing solutions with my partner Mahir Hambiralovic we touched upon a couple of points:

- Similar solutions
- Shuffling the order of evalutation of moves
- Readability

3.1 Peer's Solution

```
1  MAX_DEPTH = 5
2
3  def alpha_beta(env: ConnectFourEnv, depth, alpha, beta, maximizing_player, reward):
4      if(depth == MAX_DEPTH or reward != 0):
5          if maximizing_player:
6              return -reward, 99
7          else:
8              return reward, 99
9
10     available_moves = list(iter(env.available_moves()))
11     random.shuffle(available_moves)
12     best_move = random.choice(available_moves)
13
14     if maximizing_player:
15         value = float('-inf')
16
17         for move in available_moves:
18             env_copy: ConnectFourEnv = copy_board(env)
19             (_, reward, _, _) = env_copy.step(move)
20
21             res = alpha_beta(env_copy, depth+1, alpha, beta, False, reward)[0]
22
23             if value < res:
```

```
24                value = res
25                best_move = move
26
27            if res >= beta:
28                break
29            alpha = max(alpha, res)
30
31        return value, best_move
32
33    else:
34        value = float('inf')
35
36        for move in available_moves:
37            env_copy: ConnectFourEnv = copy_board(env)
38            env_copy.change_player()
39            (_, reward, _, _) = env_copy.step(move)
40
41            res = alpha_beta(env_copy, depth+1, alpha, beta, True, reward)[0]
42
43            if value > res:
44                value = res
45                best_move = move
46
47            if res <= alpha:
48                break
49            beta = min(beta, res)
50        return value, move
51
52 def copy_board(env):
53     env_copy: ConnectFourEnv = gym.make("ConnectFour-v0")
54     env_copy.reset(board = env.board)
55     return env_copy
```

3.2 Technical Differences of the Solutions

Although the solutions were similar we managed to find some differences in our implementations. However, these differences did not affect performance significanly.

3.3 Opinion and Performance

There were some minor differences between our solutions, however there was only one which could have an impact on performance. This was the fact that the list of available moves was shuffled before being iterated over in Mahir's solution (row 11). In order to do that, his solution first had to create a list from the frozenset of available moves returned from the environment, and then shuffle the list before iterating over it. This means that the list of available moves was iterated over 3 times for each call to the alpha_beta() function. Meanwhile my solution only iterated over the available moves once. Since the list contained at most 7 elements, we deemed the difference in performance with regards to this minimal. However it is the most important difference between our solutions.

Apart from the difference in the ordering of moves, there were some minor differences which didn't impact the performance. For instance, the move we returned from a terminal node differed. In my solution move 0 is returned while, Mahir's solution returns move 99. Since the move represents which column the piece (1 or -1) should be placed in, column 99 doesn't make any sense on a board which only has 7 columns. However, since this value is only retuned from the terminal node in the game tree it is never actually concidered by the algorithm. This is due to the fact that the only move that is actually concidered by the player is the one returned from the initial call of the recursive algorithm. The

initial call of the algorithm never evaluates a terminal node in our solutions as the initial call represents the state that the game is currenly in. If the current state of the game is terminal, alpha_beta() isn't called since the game is already finished.

Another difference that doesn't impact performance significantly is breaking out the copying of the board to a separate function. This was done in Mahir's solution but not in mine. It does however affect the readability of the solution, as one could argue that the code is easier to understand a solution which is written in a way such that contexts aren't mixed in a single function.

4 Paper summary AlphaGo (1-2 pages)

The paper *Mastering the game of Go with deep neural networks and tree search* presents a sophisticated solution to creating a system which plays a game of perfect information at a skill level exceeding that of the world's best human player. The specific game is Go; a board game where a round is about 150 turns long, and each turn has approximately 150 possible moves. This implies that the game has a very large search space. Since exhaustive exploration of the search space is infeasible, a different approach is needed in order to construct play at an expert level.

The authors present a solution which makes use of both deep neural networks and tree search. The presented system makes use of networks with two distinct purposes; estimating the optimal value function and estimating the optimal policy, respectively. The network representing the value function aims to estimate who is going to be the winner of the game given the game state, presuming that both players play optimally. The network representing the policy on the other hand aims to output a distribution over actions with respect to their individual contribution to the goal - winning the game.

The focus of the paper lies on describing the training pipeline of the networks. They are firstly trained using supervised learning. Once this stage is done, the authors employ reinforcement learning through self play. After going through training, the system gets to play against the human world champion of Go and wins.

4.1 Difference to the own solution

Even though the presented solution appears to differ a lot from my solution, the goal of both are the same; navigate through the game tree as efficiently as possible. It is of course a lot less complex to provide an efficient solution for a smaller search space, as is the case in my solution where the number of possible actions is 7 as opposed to 150 in Go. Also, a game in connect four consists of significantly less turns than one in Go.

The main difference of the two solutions is that the solution presented in the paper in a sense actually learns how to play the game. It evaluates the strength of the player's position based on experience from previous games and determines the next move in the same way, that is, it remembers the which child node to go to from a given node as opposed to simulating how the game might develop in the following moves in real time which is what my solution does. This leads

to a significantly faster system as well as a system that is better at playing the game at hand.

4.2 Performance

As mentioned above, the solution presented in the paper traverses the game tree in a near optimal manner, as opposed to evaluating many legal moves at each node and often times not finding the optimal move due to lack of search depth. Given this, it would clearly be more performant to make use of the approach provided in the paper, and since the search space is significantly smaller the stage in the neural network training pipeline where supervised learning is used could be omitted when training a model to play connect four.