Assignment 1 - Artificial Intelligence

Theodora Gaiceanu

February 9, 2022

1 Description of the solution

- My approach

As Connect Four is a game that involves two players, each of them competing against the other, one could say that the game is a competitive environment with two agents that have conflicting goals [3]. The agents know the configuration of the game at each moment, therefore the environment is fully observable. There can be only one winner in the Connect Four game, which means that a good move of one player is a bad move for the other one. In this situation, the game is a zero-sum one. Given these properties, the Connect Four game is a typical adversarial search problem.

We have two players in the game, MAX and MIN. MAX is the AI agent developed in the assignment and MIN is the server. MAX moves first, MIN moves after. As both players play optimally, a good approach would be to use minimax algorithm to solve the problem. But, as it is stated in [3], the classical minimax search algorithm performs a complete depth-first search exploration of the game tree, resulting in a time complexity of $O(b^m)$ and a space complexity of $O(bm)$, b being the number of possible moves and m being the maximum depth of the tree. Our MAX player needs to be fast to win against the server, so the time complexity makes the standard minimax algorithm a bad choice. But the Alpha-Beta pruning algorithm solves this disadvantage, computing the right minimax decision without evaluating each state by pruning [3]. What is more, in order to be more efficient, the solution is based on Alpha-Beta pruning with cutoff test. This one uses a heuristic evaluation function for states that estimates the utility of one state. This way, the search can be stopped early, treating nonterminal nodes as leafs. The cutoff test cuts off the search if the depth reached is higher than the limit or if the game has reached a terminal state. In this case, a depth limit of 5 is used (the depth must be higher than 3 - the depth limit for the server - but lower enough to enable the MAX player to make a move in less than 5 seconds). The Alpha-Beta pruning algorithm and the heuristic function are described in detail below.

Apart from the heuristic and the Alpha-Beta pruning with cutoff, the solution needs other functions as well. The *skeleton* code and the *gym* environment are used, but the solution reimplements some functions from the environment.

The **make_move** function updates the game after the player makes a move. It has three parameters: state, action and player. Basically, the action is just the column where the agent decides to move and player is the current player. So we start with the last row and go up (because the cells are occupied from the bottom to the top). For each row, we check if the selected column (action) has an empty space. If it has, we assign that cell to the player that decided to make that particular move. The function returns the updated state of the game.

The **is_valid_location** function takes as parameters the state of the game and the selected action (column). This function just checks if the selected cell is free (if it is 0). It returns True if the cell is free and False otherwise. The **get_valid_locations** function goes through all the columns of the game and calls the function **is_valid_location** to see what cells are available in the current state. It returns an array containing the empty locations.

- The Alpha-Beta pruning and the Minimax algorithms

```
"""
Alpha beta pruning algorithm
Returns the selected action and its utility value

References: https://github.com/aimacode/aima-python/blob/
    master/games.py
            https://github.com/KeithGalli/Connect4-Python/blob
    /master/connect4_with_ai.py
"""
def alpha_beta_cutoff(state, d, maxPlayer):

   def max_value(state,alpha,beta,d,maxPlayer):
      #cutoff test
      if d == 0 or winning_move(state, maxPlayer) or
    winning_move(state, not maxPlayer):
         return eval_fn(state)
      #max_value finds the action with the highest utility
    value + the utility value
      #the first element of uv is the action
      #the second element of uv is the utility value
      uv = [-1,-np.inf]
      #get all available actions
      for a in get_valid_locations(state):
         next_state = copy.deepcopy(state)
         #update the state of the game  (do the selected
    action)
         make_move(next_state, a, AI_PIECE)
         #call to min_value function
         uv_new = min_value(next_state, alpha, beta, d-1,
                              False)[1]
         #keep the action only if
         #the found utility is higher then the current utility
         if uv_new>uv[1]:
              uv = [a, uv_new]
         alpha = max(alpha,uv[1])
         if alpha>= beta:
            break
      return uv


   def min_value(state,alpha,beta,d,maxPlayer):
      #cutoff test
      if d == 0 or winning_move(state, maxPlayer) or winning_move(state, not maxPlayer):
         return eval_fn(state)
      #min_value finds the action with the lowest utility value + the utility value
      #the first element of uv is the action
      #the second element of uv is the utility value
      uv = [-1,np.inf]
      #get all available actions
      for a in get_valid_locations(state):
         next_state = copy.deepcopy(state)
         #update the state of the game  (do the selected action)
         make_move(next_state, a, SERVER_PIECE)
         #call to min_value function
         uv_new = max_value(next_state, alpha, beta, d-1,
                             True)[1]
         #keep the action only if
         #the found utility is lower then the current utility
         if uv_new<uv[1]:
              uv = [a, uv_new]
         beta = min(beta,uv[1])
         if alpha>= beta:
            break
      return uv

   #compute the alpha-beta pruning algorithm
   #the first value of best_score is the action (column)
   #the second value is the utility value
   best_score = [-1,-np.inf]
   beta = np.inf
   #get all available actions
   for a in get_valid_locations(state):
      next_state = copy.deepcopy(state)
      #make_move(next_state, a, maxPlayer)
      #uv_new  = min_value(next_state,best_score[1],beta,1, False)[1]
      #update the state of the game  (do the selected action)
      make_move(next_state, a, AI_PIECE)
      #get the lowest utility value of the new state
      uv_new  = min_value(next_state,best_score[1],beta,1, False)[1]
      #keep the action only if
      #the found utility is higher then the current utility
      if uv_new>best_score[1]:
         #update best_score
         best_score = [a, uv_new]
   #return the best action and utility value
   return best_score
```
Listing 1: Alpha-Beta Pruning with cutoff test

The Alpha-Beta pruning follows the pseudocode from [3] and has two functions. One is **max_value** which corresponds to the MAX player (our agent) and the other one is **min_value** which corresponds to the MIN player (the server). Actually, these two functions implement the Minimax algorithm, with just a few changes: they have also the α and β parameters. α is the biggest value choice along the MAX path (α is the lower bound). β is the lowest value choice along the MIN path (β is the upper bound).

Now let's describe the **max_value** function. Basically, the MAX player will choose the state with a maximal value. So first the MAX agent does the cutoff test and returns the evaluation function of the current state in case the search needs to be cut. Otherwise it will iterate through all available actions given the current state. For each action, the MAX agent makes the current action and it updates the new state of the game. Then it calls the MIN function to see what utility value it gets after the MIN player makes its move. After this, the utility value is compared to the previous utility value. If the new one is larger, than the move is good for the MAX player, the utility value is updated and the action is saved. Also, α is updated, as the maximum between the previous α and the utility value. The function returns the best action and the best utility value from the MAX player point of view.

The **min_value** function is very similar, but this one is from the point of view of the MIN player (or the server). The same cutoff test is performed. The MIN player iterates through all possible actions given the current state. For each action, it makes the current action and it updates the state of the game. It calls the MAX function to see what utility value it gets after the MAX player makes its move. Now, the MIN player wants to obtain the minimal utility value for MAX. So if the new utility value is lower than the previous one, the action is saved and the utility value is updated. Also, β is updated, as the minimum between the previous β and the utility value. The function returns the best action and the best utility value from the MIN player point of view (that is the worst case for the MAX player).

The it comes the actual Alpha-Beta search. An iteration through all available actions is done. The player makes the move and the new state of the game is updated. A call to the MIN function is done and the new utility value is saved. If this new utility values is higher than the current utility value, the utility value is updated and the action is saved. The Alpha-Beta pruning function returns the highest utility value and the best action.

- The evaluation function

```python
"""
Evaluation function:
compute the score of the state

References: https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py
"""
def eval_fn(state):
   totalScore = 0 #score for the current combination of connected cells
   state = state

   #check rows (horizontally)
   for row in state:
      for index in range(state.shape[1]-3):
         currentComb = [row[index+i] for i in range(4)]
         #add the current set score to the total score of the game
         totalScore += score_in_a_row(currentComb)
```

```python

    #check columns (vertically)
    for row in np.flip(np.transpose(state)):
        for index in range(state.shape[0]-3):
            currentComb = [row[index+i] for i in range(4)]
            totalScore += score_in_a_row(currentComb)

    #check diagonally
    #positive sloped diagonal
    for row in range(state.shape[0]-3):
        for index in range(state.shape[1]-3):
            currentComb = [state[row+i, index+i] for i in range(4)]
            totalScore += score_in_a_row(currentComb)

    #negative sloped diagonal
    for row in range(state.shape[0]-3):
        for index in range(state.shape[1]-3):
            currentComb = [state[row+3-i, index+i] for i in range(4)]
            totalScore += score_in_a_row(currentComb)

    #place the AI agent in the middle of the game
    for row in range(state.shape[0]):
        if state[row, 3] == AI_PIECE:
            totalScore += 1
    #return -1 if invalid move, totalScore otherwise
    return [-1, totalScore]

"""
Heuristic function
Sets the score
Count the numbers of 4,3,2 in a row
Row is actually a set of 4 cells
Assign a score for each

Reference: https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py
"""
def score_in_a_row(row):
    score = 0
    min_count = 0   #number of cells corresponding to the min player
    max_count = 0   #number of cells corresponding to the max player
    empty_count = 0  #empty cells


    #if it's a 1, then it corresponds to the max player
    max_count = row.count(AI_PIECE)
    #if it's a -1, then it corresponds to the max player
    min_count = row.count(SERVER_PIECE)
    #if it's a 0, then it's an empty location
    empty_count = row.count(0)

    #set a score for each move corresponding to the max player
    if max_count == 4:
        score += 5000001
    elif max_count == 3 and empty_count == 1:
        score += 5000
    elif max_count == 2 and empty_count == 2:
        score += 500

    #set a score for each move corresponding to the min player
    #penalize for good moves of the min player
    elif min_count == 4:
        score -= 5000000
    elif min_count == 3 and empty_count == 1:
        score -= 5001
    elif min_count == 2 and empty_count == 2:
        score -= 501

    return score
```

Listing 2: Evaluation function and heuristic function to set the scores for each combination of 4 cells

According to [3] a heuristic evaluation function should return an estimate of the expected utility of a certain state to a certain player. For terminal states, the evaluation function must be equal to the utility function and for non-terminal states it should be greater than the utility of the loss and less than the utility of the win.

In [2], and [1], a good heuristic function is one that gives scores according to the number of connected cells. So, in function **score_in_a_row**, the parameter is a set of 4 cells (they can be arranged horizontally, vertically or diagonally). For each cell in this set, one counts how many correspond to the MAX player, how many correspond to the server and how many are empty. If there are 4 cells corresponding to the MAX player, it means it's a win, and the score should be the winning one (which is 5000001). This is also a terminal state. If there are three cells corresponding to the MAX player, then the score is 5000 and if there are only 2 cells corresponding to the MAX player, the score is 500. Then if there are 4 cells corresponding to the MIN player, the score gets penalised with -5000000 (this is also a terminal state and it is the loss score). If there are 3 cells corresponding to the server, the score is penalised with 5001,and if there are 2 cells corresponding to the server, the score is penalised with 501. The function returns the score, which is a value.

But, as it was said before, this **score_in_a_row** function takes as input a set of 4 cells. Therefore, a function that splits the state of the game in these groups of 4 cells and computes the score for the state is needed. This is what **eval_fn** does, being actually the EVALUATION FUNCTION of the game. This function has only one parameter, the state. It checks every row in the state and it divides the rows in groups of 4 cells. For each group, it computes the score. Then it does the same thing for the columns and for the possible diagonal combinations. Also, according to [2], a **winning strategy** is to put the MAX player in the middle of the game. Therefore, the MAX player is put in the middle of the game. The function returns the total score of -1, if the move is invalid. It can be seen that for terminal states, this evaluation function returns indeed the utility value (which is 5000001 for the winning situation and 5000000 for the losing situation). And in nonterminal states, the score is somewhere between these values.

The last needed function is **winning_move** that returns True is the state is a winning state for the given player. It has therefore two parameters:

state and player (actually this is the current player). This function checks the rows in the current state and split them in groups of 4 cells. It computes the score for every group using **score_in_a_row** and verifies if it gets a winning score. Then it does the same things for the columns and diagonals.

The time execution needed for the MAX agent to make a move was also measured using the *time* method from the time package (see lines 397 and 427).

2 How to launch and use the solution

The solution has been implemented in Python. It uses the *skeleton* template provided on the Canvas page. Therefore, it needs the zip file with the template and the gym environment.

Unlike the *skeleton* code, it does not need any additional command line arguments to execute. This is because the agent only plays with the server, not with another local agent. In order to run the code, one could simply type *py skeleton_Gaiceanu.py* in a command prompt that is opened in the same folder as the script.

There are some packages that one needs to have in order to be able to execute the code:

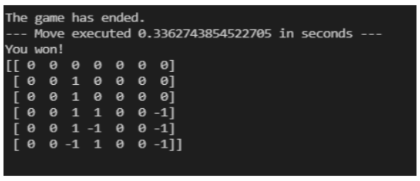
1. gym - for the gym environment
2. random - for generating random choices
3. requests - for communicating with the server
4. numpy - for some numerical values and matrix manipulation
5. copy - for properly copying the state of the game
6. time - for measuring time execution
7. gym_connect_four - for creating the game

Figure 1, 2, 3 illustrate the winning combinations that should be obtained when running the program.

3 Peer-review

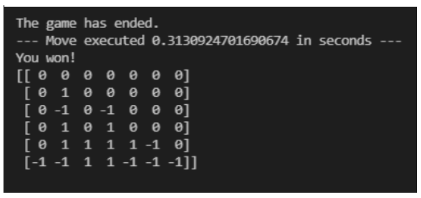
Partner: Azalea Alothmani

Some brief questions and answers:
1. Is the implementation correct? Yes, it is correct.
2. How do you know it? The code runs and the AI agent wins.
3. Is it sufficiently efficient? Yes, it is.
4. How do you measure that? The AI agent is capable to make a move in due time.
5. Does it contain alfa-beta pruning of a minimax tree? Yes, it does (*see* function **alpha_beta_cutoff(state, depth, maximizingPlayer)**).



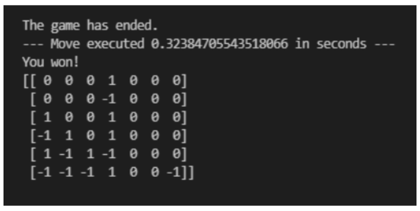
```
The game has ended.
--- Move executed 0.3362743854522705 in seconds ---
You won!
[[ 0  0  0  0  0  0  0]
 [ 0  0  1  0  0  0  0]
 [ 0  0  1  0  0  0  0]
 [ 0  0  1  1  0  0 -1]
 [ 0  0  1 -1  0  0 -1]
 [ 0  0 -1  1  0  0 -1]]
```

Figure 1: A winning combination on the column



```
The game has ended.
--- Move executed 0.3130924701690674 in seconds ---
You won!
[[ 0  0  0  0  0  0  0]
 [ 0  1  0  0  0  0  0]
 [ 0 -1  0 -1  0  0  0]
 [ 0  1  0  1  0  0  0]
 [ 0  1  1  1  1 -1  0]
 [-1 -1  1  1 -1 -1 -1]]
```

Figure 2: A winning combination on the row



```
The game has ended.
--- Move executed 0.32384705543518066 in seconds ---
You won!
[[ 0  0  0  1  0  0  0]
 [ 0  0  0 -1  0  0  0]
 [ 1  0  0  1  0  0  0]
 [-1  1  0  1  0  0  0]
 [ 1 -1  1 -1  0  0  0]
 [-1 -1 -1  1  0  0 -1]]
```

Figure 3: A winning combination on the diagonal

6. Does it contain a heuristic evaluation of nodes in the tree? Yes (*see* functions **eval_function(state)** and **heuristic(row)**).

7. How does the heuristic look like? It returns the score of the state according to the number of connected cells of the AI agent and of the opponent

8. How good is it? It is good, as it is simple and can be computed fast.

9. Is there an improvement potential in this implementation? Maybe to use the AlphaGo approach, but it may be like a waste of resources for such a simple problem. Anyway, other known winning ”tricks” may be hard coded (like trying to make a 7).

3.1 Peer's Solution

She uses the gym environment for creating the game. She uses alpha-beta search with cutoff test to return the chosen actions and the utility value of each action.

The implementation follows the standard procedure, the cutoff test ends the game if the state is terminal or if the depth limit is exceeded. The cutoff test returns the result of the evaluation function, as it is recommended. The heuristic used is the same, respectively the score of the state according to the number of connected cells of the AI agent and of the opponent. Her **heuristic(row)** function returns the score for each combination of connected cells in a row of the game. This function is used in the **eval_function(state)** in order to get the total score for a certain state of the game.

3.2 Technical Differences of the Solutions

Which things were done differently

The implementation details of the functions (**makeAMove**, **eval_function**, **heuristic**, **isAWin**).

3.3 Opinion and Performance

- Which differences are most important?

  Her evaluation function and her function that checks who is the winner may be slightly faster as she does not use a for to form the combination of 4 places. But this is not affecting the performance so much.

- How does one assess the performance?

  One can observe if the AI agent is able to make its moves in due time.

- Which solution would perform better?

  Both solutions perform slightly the same, as they follow the same recommendations.

4 Paper summary AlphaGo

The paper [4] presents the first computer program that has won a game of Go against a human player. The program uses a deep neural network approach. The network is trained by combining supervised learning from human expert games, and reinforcement learning from games of self-play.

The board is represented as a 19x19 image and the representations of the positions are build using convolutional layers. CNNs have also the goal of reducing the depth and the breadth of the Monte Carlo search trees. Monte Carlo search trees (MCTS) use Monte Carlo rollouts with the purpose of estimating the states values of the states in the search tree. Therefore, in this programme, the positions are evaluated using a value network and the actions are sampled using a policy network.

The training of the neural networks makes use of a pipeline which has multiple stages of machine learning. Firstly, a supervised learning policy network p_σ is trained from human expert games. A fast policy p_π is trained having the purpose of sampling actions fast during rollouts. The supervised learning policy network is improved by adding a reinforcement learning policy network p_ρ

that optimizes the final outcome of games of selfplay. Lastly, a value network v_θ is trained with the scope of predicting who wins the games played by the reinforcement learning policy against itself. The policy and value networks are also combined with MCTS.

The supervised learning policy network p_σ that uses professionals moves comprises CNNs with weights σ, ReLU activation functions and a softmax layer in the end. The input of the CNNs is a representation of the game s and the output is the probability distribution over all available moves a. The dataset used for training is represented by state-action pair (s,a). The used optimizer is gradient ascent (it maximizes the likelihood of a human move a in state s). p_π is another policy, that is faster but it has lower accuracy. This one uses a linear softmax of small patter features with weights π.

The policy network is improved by policy gradient reinforcement learning. This RL policy network p_ρ has the same structure as the supervised learning policy and its weights are initialized in an identical way. In order to prevent oferfitting, the authors let the current policy network p_ρ play against a randomly selected previous iteration of the policy network. The output is represented by the terminal reward at the end of the game from the point of view of the current player (1 for winning, -1 for losing). The used optimmizer is again stochastic gradient ascent, maximizing the expected outcome. This reinforcement learning policy network is the most accurate.

The last part of the training pipeline is represented by the value networks, which has the purpose of estimating a value function $v^p(s)$ predicting the outcome from position s of games played by using policy p for the players. This value function is approximated using the reinforcement learning policy network p_ρ. The value function is approximated using a value network v_θ. The network structure is almost the same as the structure of the policy network, but the output is only one prediction. This time the used optimzer is stochastic gradient descent and it minimizes the mean squared error between the predicted value $v_\theta(s)$ and the related outcome z.

The policy and the value networks are combined in an MCTS algorithm, selecting actions by lookahead search. In order to make an efficeint combination between MCTS and deep neural newtorks, an asynchronums mult-threated search is used. This way, simulations are computed on CPUs, while the policy and value network are computed on GPUs. In order to evaluate the performance, AlphaGo played games against other Go programs, AlphaGo being the winner the most of times. AlphaGo is also the first computer Go program that won against a human profesional player.

4.1 Difference to the own solution

There are many differences between the AlphaGo approach and my solution.

First of all, instead of using a classical heuristic evaluation function (like I use the score of each state according to the number of connected cells), the AlphaGo approach uses deep neural networks. More precise, it uses the reinforcement learning value network instead of a classical evaluation function.

The supervised learning policies are used to in order to make use of the known human expert good moves. So instead of searching through a list of possible moves and selecting the move that maximizes the utility function, AlphaGo uses deep learning to select the best moves (using in this way a bigger amount of human knowledge - I only coded in the **eval_fn** that the player should be placed in the middle of the board.

Last but not least, AlphaGo uses MCTS, not Alpha-Beta Tree Search. Actually, MCTS in general do not use a heuristic evaluation function. They estimate the value of a state as the average utility over multiple rollouts of complete games starting from the state of the game [3]. Also, MCTS use a playout policy that selects the good moves using the game of the rules. This way, MCTS search is suitable for games with high branching factor or for it is not easy to implement an evaluation function.

4.2 Performance

Would you play better using the AlphaGo approach instead? Why or why not?

It is true that using AlphaGo one could use more knowledge from human experts, thanks to the supervised learning policy networks. And this could make the AI agent more invincible. Moreover, more available actions could be explored when using MCTS. But, on the other hand, MCTS have one known disadvantage: when a single move can change the course of the game, the stochastic property of the Monte Carlo search might make the algorithm not even consider that move. This disadvantage might be overcome though by the Reinforcement Learning value network or by the Reinforcement Learning policy.

Anyway, in order to use the AlphaGo approach a multithreaded search and GPUs would be needed. Because otherwise the AI agent might fail to give the response in less than 5 seconds, as Alpha Go uses deep neural networks.

All things considered, I would say that maybe AlphaGo would make that AI agent more accurate, but the Connect Four game is a very simple game and, as long as a simpler (and less expensive from the computational effort and resources involved point of view) reasonable solution is available, the simpler solution should be used.

References

[1] Galli K. *Connect 4 - Python.* URL: `https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py`. (accessed: 04.02.2022).

[2] Jonathan C.T. Kuo. *Artificial Intelligence at Play — Connect Four (Minimax algorithm explained).* URL: `https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f`. (accessed: 04.02.2022).

[3] Norvig P. Russel S. *Artificial Intelligence - A Modern Approach.* Pearson Education, 2021. Chap. 6. Adversarial Search and Games. ISBN: 9780134610993.

[4] Maddison C. et al. Silver D. Huang A. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.10 (2016), pp. 484–489. DOI: `https://doi.org/10.1038/nature16961`.