Lab Session #4

Building an AI Agent in Python

Minimax algorithm, AI games

Dr Zied M'Nasri

Introduction

The purpose of this lab is to design and implement a simple AI agent capable of playing a game against a human opponent. By the end of the session, you will have created an autonomous game-playing program that can evaluate game states, simulate possible outcomes, and make optimal decisions based on the Minimax algorithm — a fundamental concept in Artificial Intelligence for decision-making under competition.

Learning Objectives

After completing this lab, you should be able to:

- Represent a game environment using suitable data structures (lists and matrices).
- Implement utility functions for evaluating game states and possible actions.
- Apply the Minimax algorithm to simulate optimal decision-making.
- Integrate human–AI interaction through input/output and control flow.
- Understand how recursion enables an AI agent to anticipate future moves.

Main Tasks

To achieve the lab's goal, you will progressively implement and test several components of the AI agent:

1. Display Function (**print_board**): Create a function to visually display the current state of the 3×3 game board.
2. Game Evaluation Function (**check_winner**): Implement logic to check for a winner (X (human) or O (AI agent)) or determine if the game is a draw.
3. Move Generation Function (**available_moves**): Write a function that identifies all valid empty positions on the board.
4. Decision-Making Algorithm (**minimax**): Implement the recursive Minimax algorithm to evaluate all possible game outcomes and compute the optimal move.

5. AI Move Selection (**best_move**): Use Minimax results to select and return the best move for the AI player (O).
6. Game Controller (**play_game**): Build the main loop that alternates turns between the human and AI players, manages input, updates the board, and displays results.
7. Program Entry Point (**if __name__ == "__main__"**): Add a final statement to ensure that the game runs automatically when the script is executed.

Step 1 — Python environment setup

1. Open your Python development environment (Colab/Jupyter) or use any python editor NotePad/VS code/PyCharm.
2. Create a new Python notebook (or file) named AI_game.ipynb (or AI_game.py)
3. At the top of the file, import the following library:

```
import math
```

Step 2 — Displaying the Game Board (print_board)

Implement the function **print_board(board)** to show the current state of the 3×3 grid.

Test: Create a sample board as a list containing "X" (Human) and "O" (Computer) and call the created function

Sample result: The board should print the board's input values with rows and columns separated by lines.

```
board = [["X", "O", "X"], ["O", "X", "O"], ["X", "O", "X"]]
print_board(board)
```

```
X | O | X
---------
O | X | O
---------
X | O | X
```

Step 3 — Checking for a Winner (check_winner)

The function checks all possible winning lines (rows, columns, diagonals).

- If a player has three same marks in a row/column/diagonal, it returns that player's mark ("X" or "O").
- If the board is full with no winner, it returns "Draw".
- If there are still empty spots, it returns None to indicate the game is not finished.

a. Check rows for a winner:

- The algorithm goes through each of the three rows on the board.
- For each row, it checks if all three positions contain the same mark (either "X" or "O") and are not empty (" ").
- If this condition is true, the function returns that mark (the winner).

b. Check columns for a winner:

- Next, it checks each of the three columns.
- For each column, it verifies if all three positions in that column have the same non-empty mark.
- If this is the case, it returns that mark (the winner).

c. Check diagonals for a winner:

- The function then checks the two diagonals of the board.
- If all three cells in a diagonal contain the same non-empty mark, that mark is returned as the winner.

d. Check for a draw:

- If no winner has been found, the algorithm looks through every row to see if there are any empty spaces (" ") left.
- If any empty space exists, it means the game is still ongoing, so it returns None.

e. Declare a draw:

If there are no empty spaces and no winner, the game is a draw, so it returns "Draw".

Test: Use different board configurations to verify that it correctly returns "X", "O", "Draw", or None.

Sample result: Correctly identifies winning rows, columns, diagonals, and draws.

```
board = [["X", "O", "X"], ["O", "X", "O"], ["X", "O", "X"]]
print_board(board)
check_winner(board)
```

```
X | O | X
---------
O | X | O
---------
X | O | X


'X'
```

Step 4 — Generating Available Moves (available_moves)

The function scans the entire 3×3 board and returns a list of all empty spots as (row, column) pairs. If the board is full, it returns an empty list.

a. **Initialize an empty list:**

- It starts by creating an empty list called moves.
- This list will be used to store the positions (as row and column pairs) of all available cells.

b. **Loop through each cell on the board:** The function uses two nested loops:

- The outer loop (i) goes through each of the three rows (from 0 to 2).
- The inner loop (j) goes through each of the three columns (from 0 to 2).

c. **Check if a cell is empty:**

- For each position on the board, it checks if the cell contains a blank space (" ").
- If the cell is empty, that means it's available for a move.

d. **Record the available move:**

- When an empty cell is found, the function adds a tuple (i, j), representing the row and column of that cell, to the moves list.

e. **Return the list of moves:**

- After checking all cells, the function returns the moves list, which contains all the coordinates of available positions.

Test: Call the implemented function on partially filled boards and verify that it returns all valid (row, col) positions.

Sample result: Returns a list of all empty coordinates (e.g., [(0, 1), (1, 2), (2, 0)]).

```
board = [["X", " ", "0"], ["0", "X", " "], [" ", "0", "X"]]
print_board(board)
available_moves(board)
```

```
X |   | 0
---------
0 | X |
---------
  | 0 | X


[(0, 1), (1, 2), (2, 0)]
```

Step 5 — The Minimax Algorithm

Note- *To speed up your work during the lab, you can use the minimax algorithm in the attached python file (minimax.py) (Then implement it later by yourself following the steps below)*

The Minimax algorithm is a recursive strategy used in games to choose the best possible move for a player. It simulates every possible move until the end of the game and assigns scores based on the outcomes to decide the best move.

The function recursively explores every possible move for both players until the game ends.

Each outcome is assigned a numerical value:

- **+1** if the AI agent ("O") wins
- **–1** if the opponent (human user) ("X") wins
- **0** for a draw

The AI agent chooses moves that **maximize** the score, while the opponent chooses moves that **minimize** it. This ensures the AI always plays optimally, assuming both players make the best possible moves.

Steps

a. **Check for a terminal state:**

- The function first checks if the current board already has a winner or if the game is a draw by calling the function **check_winner(board)**.
- Depending on the result:

- If **"O"** wins, return **+1** (good for the AI agent, assuming "O" is the computer).
- If **"X"** wins, return **–1** (bad for the AI agent).
- If it's **a draw**, return **0** (neutral outcome).

- These return values serve as the base cases that stop the recursion.

b. If it is the maximizing player's turn (the AI agent "O"):

- Set `best_score` to **negative infinity** (python value : `-math.inf`) to start.
- Loop through every available move on the board:
  1. Place an "O" in that empty spot to simulate a possible move.
  2. Call minimax recursively for the next turn (`is_maximizing=False`) to see how the opponent would respond.
  3. Undo the move by resetting the cell to " ".
  4. Compare the returned score with the current `best_score`, and keep the **higher** one (since the AI wants to maximize its advantage).
- After checking all moves, return the `best_score`.

c. If it is the minimizing player's turn (the opponent, "X"):

1. Set `best_score` to positive infinity (python value: `math.inf`) to start.
2. Loop through every available move:
3. Place an "X" in that empty spot to simulate the opponent's move.
4. Call minimax recursively for the next turn (`is_maximizing=True`).
5. Undo the move by resetting the cell to " ".
6. Compare the returned score with `best_score`, and keep the lower one (since the opponent tries to minimize the AI agent's score).
7. After all possible moves are checked, return the `best_score`.

Test: Run it on simple game states (e.g., where one move wins or blocks).

Sample result: Returns 1 for AI wins, -1 for human wins, and 0 for draws.

```
board = [["X", "0", " "], ["X", "0", " "], [" ", "O", "X"]]
minimax(board, 1, 0)
```

```
-1
```

Step 6 — Choosing the Optimal Move (best_move)

The **best_move(board)** function systematically tests every available move for the AI player ("O"):

- It uses the Minimax algorithm to predict how good each move is.
- It keeps track of the highest-scoring move.
- Finally, it returns the coordinates of that move so the AI can play it.

a. Initialize starting values:

- It sets `best_score` to negative infinity (use the python value `-math.inf`), which represents the lowest possible score to start with.
- It sets move to None — this will later store the coordinates (row, column) of the best move found.

b. Loop through all available moves:

The function goes through every empty cell on the board using the function **available_moves (board)**.

For each possible move (a pair of coordinates (i, j)):

1. It temporarily places an "O" (the AI agent's mark) in that spot to simulate making that move.
2. It calls the **minimax** function to calculate the outcome score of that move, assuming both players play optimally.
3. The parameters `depth=0` and `is_maximizing=False` mean it is now the opponent's turn ("X").
4. After evaluating, it resets the cell back to " " (an empty space) to restore the board for the next test.

c. Compare and update the best move:

After getting the score for a move:

- If this score is greater than the current best_score, that means this move is better than all previously tested ones.
- The algorithm updates:
  - `best_score` to the new score.
  - move to the coordinates (i, j) of this move.

d. Return the best move:

After all possible moves are tested, the function returns the (row, column) position of the move that gives the highest score — i.e., the optimal move for the AI.

Test: Call **best_move(board)** on different boards and check if it selects the winning or blocking move.

Sample result: Returns a tuple (row, col) corresponding to the best possible move for "O".

```
board = [["X", "0", " "], ["X", "0", " "], [" ", "O", "X"]]
best_move(board)
```

```
(2, 0)
```

Step 7 — Playing the full game (play_game)

The function runs the full game loop between a human player ("X") and an AI player ("O"), handling moves, checking for a winner, and printing the board.

The function **play_game()** runs the complete game:

1. Initializes and prints the board.
2. Alternates human and AI moves.
3. Validates moves and handles user input errors.
4. Uses **best_move** and **minimax** functions to let the AI play optimally.
5. Checks for a winner or draw after each move and prints the result.

Essentially, it's the main game loop that ties together all the smaller functions (check_winner, available_moves, minimax, best_move, print_board) into a playable game.

a. Initialize the board:

- Creates a 3×3 game board filled with empty spaces (" ").
- Prints a welcome message and instructions for the human player.
- Displays the initial empty board using **print_board(board)**.

b. Main game loop:

The game continues in a loop until there is a winner or a draw.

c. Human player's turn:

1. Prompt the user to enter a move as two numbers (row and column).
    - If the input is invalid (non-numbers or wrong format), show an error and ask again.

2. Check if the move is **within bounds** (0–2 for row and column) and the chosen cell is empty.
    - If not, display "**Invalid move**" and ask again.
3. Place the human's mark "X" on the chosen cell.
4. Display the updated board.
5. Check if the game has ended (win or draw) using **check_winner(board)**:
    - If someone won, print the winner's mark.
    - If it's a draw, print "It's a draw!".
    - Break the loop to end the game if the game is over.

d. AI player's turn:

1. Print a message "AI is thinking...".
2. Call **best_move(board)** to calculate the optimal move for the AI.
3. If a valid move is returned, place the AI's mark "O" in that cell.
4. Display the updated board.
5. Check again if the game has ended (win or draw) using **check_winner(board)**:
    - If someone won, print the winner's mark.
    - If it's a draw, print "It's a draw!".
    - Break the loop if the game is over.

e. Repeat turns:

- The loop alternates between the human and AI moves until the board reaches a **terminal state** (someone wins or the board is full).

Test: Run the full program, play several rounds, and verify that the AI never loses.

Sample result: An interactive game on the terminal.

```
play_game()

Welcome to Tic-Tac-Toe!
You are X, AI is O


  |   |
---------
  |   |
---------
  |   |


Enter your move (row and col: 0-2 0-2): 1 1
```

Final note — Adding the Entry Point

To execute the program directly using the command prompt, e.g. :

```
C:\Users> python3   AI_game.py
```

Add the following code

```
if __name__ == "__main__":
    play_game()
```

Sample result: An interactive game!

```
if __name__ == "__main__":
    play_game()
```

```
Welcome to Tic-Tac-Toe!
You are X, AI is O


  |   |
---------
  |   |
---------
  |   |


Enter your move (row and col: 0-2 0-2): 0 0


X |   |
---------
  |   |
---------
  |   |


AI is thinking...


X |   |
---------
  | O |
---------
  |   |


Enter your move (row and col: 0-2 0-2): 11
Invalid input. Use two numbers between 0 and 2 separated by a space.
Enter your move (row and col: 0-2 0-2): 1 2
```

```
X |   |
---------
  | O | X
---------
  |   |


AI is thinking...


X | O |
---------
  | O | X
---------
  |   |


Enter your move (row and col: 0-2 0-2): 2 0


X | O |
---------
  | O | X
---------
X |   |


AI is thinking...


X | O |
---------
  | O | X
---------
X | O |


O wins!
```