Artificial Intelligence Lab

Project Submission

Submitted to: **Navpreet Kaur Ma'am**

Submitted by:	Name	Roll No.
	Sanyam Gupta	102103390
	Jaisika Bhatia	102103378
	Harshal Mehra	102103813
	Sunali	102103375
	Anshika Anand	102103380

Link to the project: https://github.com/SANYAM1612/ConnectFour

---

Problem statement:

Design and implement an AI-based program in Python to play the game of Connect Four against a human player.

Description of problem:

Connect Four is a two-player game played on a vertical board with 7 columns and 6 rows. Each player takes turns dropping a colored disc into a column. The disc falls to the lowest empty cell in the column. The objective of the game is to connect four discs of the same color vertically, horizontally, or diagonally.

The goal of this problem is to create an AI program that can play Connect Four against a human player. The program should be able to evaluate the current state of the game and determine the best move to make based on that evaluation. The AI should be able to make strategic moves that prevent the opponent from winning while also trying to win the game

itself.

Code and Explanation

We use pip install pygame because in google colab doesn't have pygame installed in it plus we also change aur runtime to GPU to support the library.

```
pip install pygame
```

```
Collecting pygame
  Downloading pygame-2.1.1-cp39-cp39-win_amd64.whl (14.0 MB)
     |████████████████████████████████| 14.0 MB 2.2 MB/s
Installing collected packages: pygame
Successfully installed pygame-2.1.1
```

IMPORTING USED MODULES

Brief description of the imported modules:

1. **pygame:** Pygame is a cross-platform set of Python modules designed for creating video games. It provides functionality for handling graphics, sound, input, and event handling.
2. **numpy:** NumPy contains a multi-dimensional array and matrix data structures. It can beutilized to perform a number of mathematical operations on arrays
3. **random:** The random module in Python is a built-in module that provides functions for generating random numbers, sequences, and selections. The random module is used in a wide variety of applications, such as simulations, games, cryptography, and more.
4. **sys:** The sys module provides access to some variables and functions related to

the Python interpreter and its environment. For example, the sys.argv variable provides a list of command-line arguments passed to the Python script, and sys.path provides a list of directories where Python looks for modules. The sys.exit() function can be used to exit the Python interpreter with a specified exit code.

5. **math:** The math module provides access to a range of mathematical functions, constants, and operations. Some of the commonly used functions include sqrt() for square root, exp() for exponential, log() for logarithm, sin() and cos() for sine and cosine, pi for the value of pi, and e for the value of Euler's number. The math module also provides constants such as inf for infinity, nan for not-a-number, and tau for the value of 2*pi.

```
import numpy as np
import random
import pygame
import sys
import math
```

```
def create_board():
    board = np.zeros((ROW_COUNT,COLUMN_COUNT))
    return board

def drop_piece(board, row, col, piece):
    board[row][col] = piece

def is_valid_location(board, col):
    return board[ROW_COUNT-1][col] == 0

def get_next_open_row(board, col):
    for r in range(ROW_COUNT):
        if board[r][col] == 0:
            return r

def print_board(board):
    print(np.flip(board, 0))
```

create_board(): This function creates a 6x7 two-dimensional NumPy array filled with zeros to represent the game board.

drop_piece(board, row, col, piece): This function takes the game board, the row and

column where a piece should be dropped, and the player's piece (either 1 or 2) as input. It updates the corresponding location on the board with the player's piece.

is_valid_location(board, col): This function takes the game board and the column where the player wants to drop a piece as input. It checks if the top row of that column is empty (i.e., contains a zero) and returns True if the location is valid (i.e., the column is not full) and False otherwise.

get_next_open_row(board, col): This function takes the game board and the column where the player wants to drop a piece as input. It returns the next available row in that column where a piece can be dropped (i.e., the first empty cell from the bottom).

print_board(board): This function takes the game board as input and prints it to the console in a visually appealing format, with the bottom row of the board at the top of the output and the top row at the bottom (hence the np.flip(board, 0) command).

```
def winning_move(board, piece):
    # Check horizontal locations for win
    for c in range(COLUMN_COUNT-3):
        for r in range(ROW_COUNT):
            if board[r][c] == piece and board[r][c+1] == piece and board[r][c+2] == piece and board[r][c+3] == pi
                return True

    # Check vertical locations for win
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT-3):
            if board[r][c] == piece and board[r+1][c] == piece and board[r+2][c] == piece and board[r+3][c] == pi
                return True

    # Check positively sloped diaganols
    for c in range(COLUMN_COUNT-3):
        for r in range(ROW_COUNT-3):
            if board[r][c] == piece and board[r+1][c+1] == piece and board[r+2][c+2] == piece and board[r+3][c+3]
                return True

    # Check negatively sloped diaganols
    for c in range(COLUMN_COUNT-3):
        for r in range(3, ROW_COUNT):
            if board[r][c] == piece and board[r-1][c+1] == piece and board[r-2][c+2] == piece and board[r-3][c+3]
```

This function **winning_move(board, piece)** checks if a player has won the game by getting four pieces in a row, either horizontally, vertically, or diagonally.

It takes the current game board and the player's piece as input.

It uses four nested loops to iterate through all possible four-piece combinations in the horizontal, vertical, and diagonal directions.

If it finds a sequence of four pieces of the same player's piece in any direction, it returns True, indicating that the game has been won.

Otherwise, it returns False, indicating that the game is still ongoing.

```
def evaluate_window(window, piece):
    score = 0
    opp_piece = PLAYER_PIECE
    if piece == PLAYER_PIECE:
        opp_piece = AI_PIECE

    if window.count(piece) == 4:
        score += 10000000
    elif window.count(piece) == 3 and window.count(EMPTY) == 1:
        score += 5
    elif window.count(piece) == 2 and window.count(EMPTY) == 2:
        score += 2

    if window.count(opp_piece) == 3 and window.count(EMPTY) == 1:
        score -= 4

    return score
```

The function called **evaluate_window** takes in two parameters, "window" and "piece". The "window" parameter is a list of four consecutive positions on the game board in the game of Connect Four. The "piece" parameter is the piece that the function is evaluating for (either the player's or the AI's).

The function first initializes a variable called "score" to 0. It then checks if the "piece" appears four times in the "window". If so, it adds a very high score of 10,000,000 to the "score" variable. If the "piece" appears three times and there is one empty space in the "window", it adds a score of 5 to the "score" variable. If the "piece" appears two times and there are two empty spaces in the "window", it adds a score of 2 to the "score" variable.

If the opponent's piece appears three times and there is one empty space in the "window", the function subtracts a score of 4 from the "score" variable.

Finally, the function returns the calculated "score" variable. This function is likely part of a larger program that uses these scores to determine the best move for the AI to make in the game of Connect Four.

```
def score_position(board, piece):
    score = 0

    ## Score center column
    center_array = [int(i) for i in list(board[:, COLUMN_COUNT//2])]
    center_count = center_array.count(piece)
    score += center_count * 3

    ## Score Horizontal
    for r in range(ROW_COUNT):
        row_array = [int(i) for i in list(board[r,:])]
        for c in range(COLUMN_COUNT-3):
            window = row_array[c:c+WINDOW_LENGTH]
            score += evaluate_window(window, piece)

    ## Score Vertical
    for c in range(COLUMN_COUNT):
        col_array = [int(i) for i in list(board[:,c])]
        for r in range(ROW_COUNT-3):
            window = col_array[r:r+WINDOW_LENGTH]
            score += evaluate_window(window, piece)
```

```
    ## Score posiive sloped diagonal
    for r in range(ROW_COUNT-3):
        for c in range(COLUMN_COUNT-3):
            window = [board[r+i][c+i] for i in range(WINDOW_LENGTH)]
            score += evaluate_window(window, piece)

    for r in range(ROW_COUNT-3):
        for c in range(COLUMN_COUNT-3):
            window = [board[r+3-i][c+i] for i in range(WINDOW_LENGTH)]
            score += evaluate_window(window, piece)

    return score
```

This code defines a function called **score_position** which takes in two arguments - a board representing the current state of the Connect Four board, and a piece representing the piece for which the score is being calculated. The function calculates the score of a given position on the board for the specified piece.

The function first initializes a variable score to 0. It then proceeds to calculate the score by evaluating the different ways in which the given piece can be placed on the board.

The function first scores the center column of the board by checking the number of pieces of the given type that are present in the center column, and multiplies it by 3. This is because the center column is generally considered to be a strong position in Connect Four.

Next, the function calculates the score of each row and column on the board by considering each possible window of 4 adjacent slots in that row or column, and passing it to the **evaluate_window** function, which is not defined in the provided code. It is likely that this function will determine the score of a given window based on the number of pieces of the given type present in it.

Finally, the function calculates the score of each possible diagonal on the board using a similar approach, checking both positively sloped and negatively sloped diagonals. The function then returns the total score of the given position for the specified piece.

```
def is_terminal_node(board):
    return winning_move(board, PLAYER_PIECE) or winning_move(board, AI_PIECE) or len(get_valid_locations(board))

def minimax(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = get_valid_locations(board)
    is_terminal = is_terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if winning_move(board, AI_PIECE):
                return (None, 100000000000000)
            elif winning_move(board, PLAYER_PIECE):
                return (None, -10000000000000)
            else: # Game is over, no more valid moves
                return (None, 0)
        else: # Depth is zero
            return (None, score_position(board, AI_PIECE))
    if maximizingPlayer:
        value = -math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, AI_PIECE)
          if new_score > value:
              value = new_score
              column = col
          alpha = max(alpha, value)
          if alpha >= beta:
              break
     return column, value

 else: # Minimizing player
     value = math.inf
     column = random.choice(valid_locations)
     for col in valid_locations:
         row = get_next_open_row(board, col)
         b_copy = board.copy()
         drop_piece(b_copy, row, col, PLAYER_PIECE)
         new_score = minimax(b_copy, depth-1, alpha, beta, True)[1]
         if new_score < value:
             value = new_score
             column = col
         beta = min(beta, value)
         if alpha >= beta:
             break
     return column, value
```

This code defines two functions**, is_terminal_node** and minimax, that are used for implementing the minimax algorithm for the game of Connect Four.

The **is_terminal_node** function checks if the current state of the board is a terminal state, i.e., if either the player or the AI has won, or if there are no more valid moves left. It returns True if any of these conditions are satisfied, and False otherwise.

The minimax function takes in the current state of the board, the current depth of the search, the alpha and beta values for pruning, and a boolean variable maximizingPlayer that determines whether the current player is the AI or the human player. It then recursively evaluates all possible moves from the current state using the minimax algorithm, and returns the column and corresponding score that result in the best possible move for the current player.

The function first checks if the current depth is 0 or if the current state is a terminal state using the **is_terminal_node** function. If either of these conditions are true, it returns the column and score corresponding to the current state.

If the current player is the AI, the function tries to maximize its score by evaluating all possible moves and choosing the move that leads to the highest score. It does this by iterating over all valid locations on the board and recursively calling the minimax function on each resulting state, with the depth reduced by 1 and the maximizingPlayer parameter set to False. It then updates the value of alpha and beta as needed for pruning.

If the current player is the human player, the function tries to minimize the AI's score by choosing the move that leads to the lowest score. It follows the same process as above, but with the maximizingPlayer parameter set to True.
The function then returns the best column and its corresponding score.

```
def get_valid_locations(board):
    valid_locations = []
    for col in range(COLUMN_COUNT):
        if is_valid_location(board, col):
            valid_locations.append(col)
    return valid_locations

def pick_best_move(board, piece):

    valid_locations = get_valid_locations(board)
    best_score = -10000
    best_col = random.choice(valid_locations)
    for col in valid_locations:
        row = get_next_open_row(board, col)
        temp_board = board.copy()
        drop_piece(temp_board, row, col, piece)
        score = score_position(temp_board, piece)
        if score > best_score:
            best_score = score
            best_col = col

    return best_col
```

The function, **get_valid_locations(board)**, takes the current game board as input and returns a list of valid column positions where the player can drop their piece. It does so by iterating over each column of the board and checking if it is a valid move using the **is_valid_location()** function. If a column is valid, its index is appended to the **valid_locations** list.

The function, **pick_best_move(board, piece)**, takes the current game board and the player's piece as input and returns the best column position for the player to drop their piece. It does so by first obtaining a list of valid column positions using the **get_valid_locations()** function. It then iterates over each valid column position, simulating the addition of the player's piece to the corresponding row using the **drop_piece()** function, and calculates a score for the resulting board position using the **score_position()** function. The column position with the highest score is returned as the best move for the player. If there are multiple columns with the same highest score, one of them is randomly chosen using the **random.choice()** function.

```
def draw_board(board):
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT):
            pygame.draw.rect(screen, BLUE, (c*SQUARESIZE, r*SQUARESIZE+SQUARESIZE, SQUARESIZE, SQUARESIZE))
            pygame.draw.circle(screen, BLACK, (int(c*SQUARESIZE+SQUARESIZE/2), int(r*SQUARESIZE+SQUARESIZE+SQUARE

    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT):
            if board[r][c] == PLAYER_PIECE:
                pygame.draw.circle(screen, RED, (int(c*SQUARESIZE+SQUARESIZE/2), height-int(r*SQUARESIZE+SQUARESI
            elif board[r][c] == AI_PIECE:
                pygame.draw.circle(screen, YELLOW, (int(c*SQUARESIZE+SQUARESIZE/2), height-int(r*SQUARESIZE+SQUAR
    pygame.display.update()
```

This function **draw_board** draws the Connect Four game board on the screen, as well as any pieces that have been played by either the player or the computer. It does so by iterating over each column and row of the game board and drawing the appropriate rectangle and circle shapes using the Pygame library. Once all the shapes have been drawn, the screen is updated to display the current state of the game board.

```
board = create_board()
print_board(board)
game_over = False

pygame.init()

SQUARESIZE = 100

width = COLUMN_COUNT * SQUARESIZE
height = (ROW_COUNT+1) * SQUARESIZE

size = (width, height)

RADIUS = int(SQUARESIZE/2 - 5)

screen = pygame.display.set_mode(size)
draw_board(board)
```

```
pygame.display.update()

myfont = pygame.font.SysFont("monospace", 75)

turn = random.randint(PLAYER, AI)

while not game_over:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

        if event.type == pygame.MOUSEMOTION:
            pygame.draw.rect(screen, BLACK, (0,0, width, SQUARESIZE))
            posx = event.pos[0]
            if turn == PLAYER:
                pygame.draw.circle(screen, RED, (posx, int(SQUARESIZE/2)), RADIUS)

        pygame.display.update()

        if event.type == pygame.MOUSEBUTTONDOWN:
            pygame.draw.rect(screen, BLACK, (0,0, width, SQUARESIZE))
            #print(event.pos)
            # Ask for Player 1 Input
            if turn == PLAYER:
                posx = event.pos[0]
                col = int(math.floor(posx/SQUARESIZE))

                if is_valid_location(board, col):
                    row = get_next_open_row(board, col)
                    drop_piece(board, row, col, PLAYER_PIECE)

                    if winning_move(board, PLAYER_PIECE):
                        label = myfont.render("Player 1 wins!!", 1, RED)
                        screen.blit(label, (40,10))
                        game_over = True

                    turn += 1
                    turn = turn % 2

                    print_board(board)
                    draw_board(board)


    # # Ask for Player 2 Input
    if turn == AI and not game_over:

        #col = random.randint(0, COLUMN_COUNT-1)
        #col = pick_best_move(board, AI_PIECE)
```

```
        col, minimax_score = minimax(board, 5, -math.inf, math.inf, True)

        if is_valid_location(board, col):
            #pygame.time.wait(500)
            row = get_next_open_row(board, col)
            drop_piece(board, row, col, AI_PIECE)

            if winning_move(board, AI_PIECE):
                label = myfont.render("Player 2 wins!!", 1, YELLOW)
                screen.blit(label, (40,10))
                game_over = True

            print_board(board)
            draw_board(board)

            turn += 1
            turn = turn % 2

    if game_over:
        pygame.time.wait(3000)
```

This code sets up a game of Connect Four using the Pygame library. It creates a game board using the **create_board** function and initializes several variables such as the size of the game window, the radius of the game pieces, and the font used to display messages to the players.

The game alternates between the player and the AI, and each player takes a turn by clicking on the desired column on the game board. The game checks if the move is valid, drops a piece into the appropriate row, and checks if the move results in a win. If a win is detected, the game ends and a message is displayed indicating the winner.

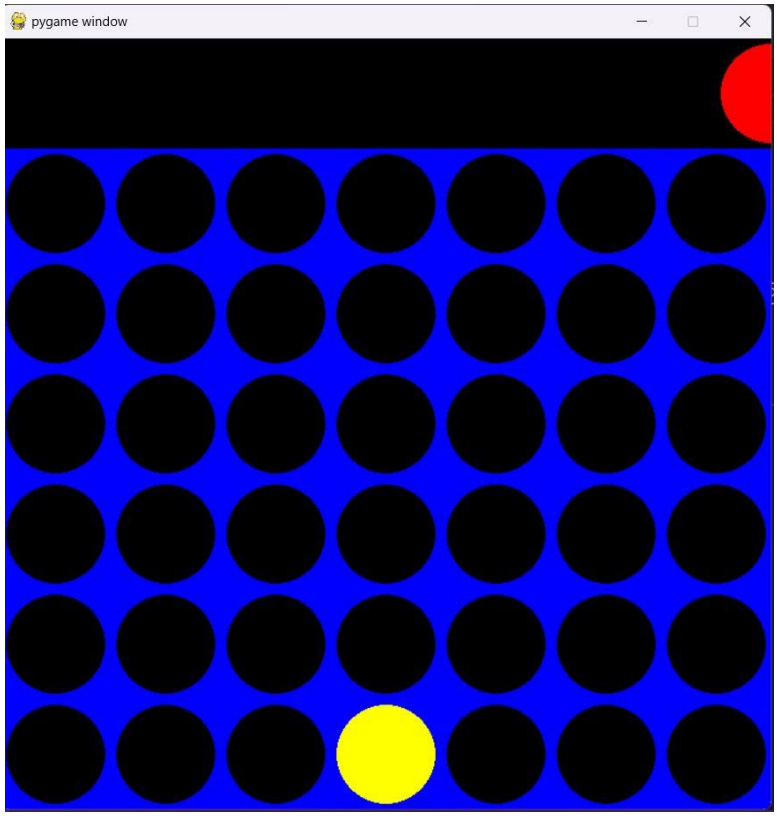
The AI's moves are determined by the "minimax" function, which uses a recursive algorithm to search for the best possible move based on a given depth level. The **pick_best_move** function is also provided as an alternative method for the AI to select a move.

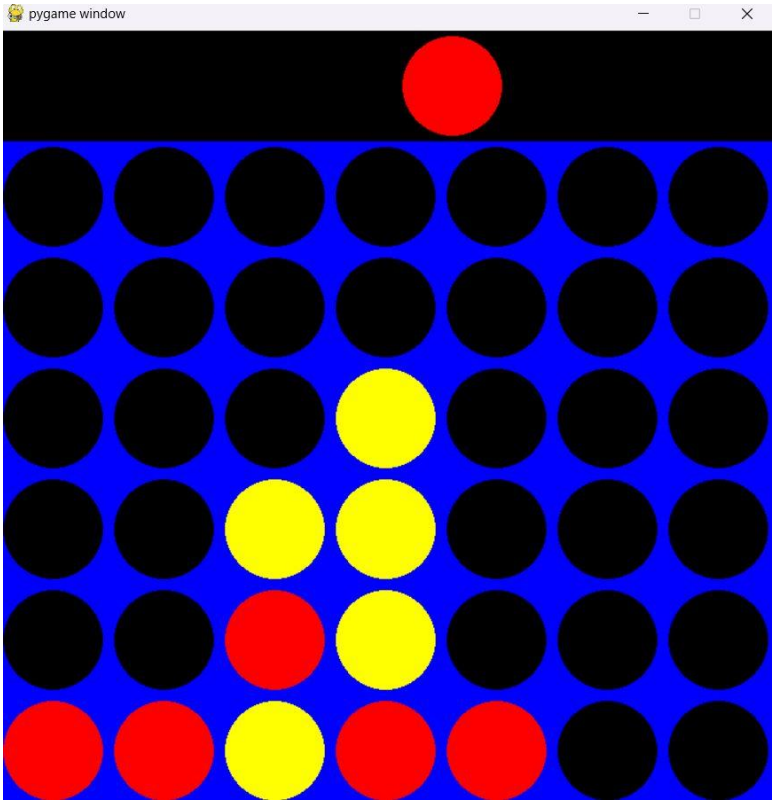
The game window is updated after each move and displays the current state of the game board. When the game ends, the program waits for three seconds before exiting.

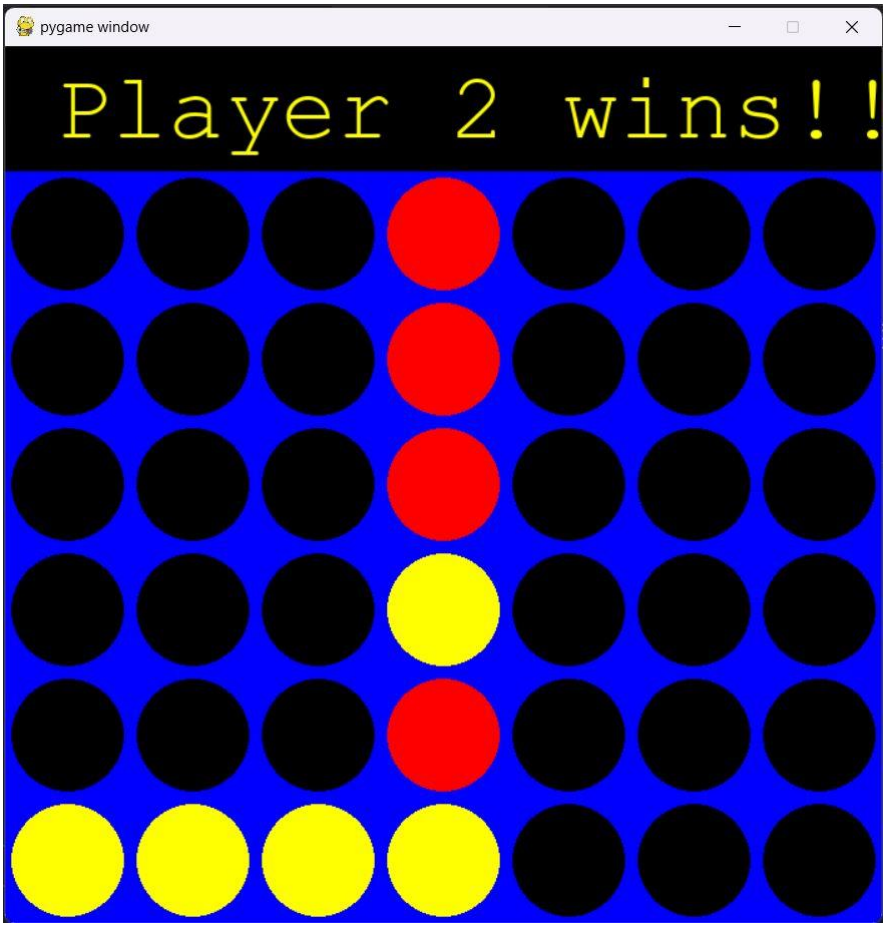
Link to the project:

https://github.com/SANYAM1612/ConnectFour

The above code helps to play the game of minmax algorithm in a user-friendly manner as demonstrated below with the help of pictures:




pygame window


pygame window
Player 2 wins!!

#