**Game Board**:

For designing the board, we have used numpy and pygame library of Python.

import numpy as np  
import pygame

ROW\_COUNT = 6  
COLUMN\_COUNT = 7

def create\_board():  
 board = np.zeros((ROW\_COUNT, COLUMN\_COUNT))  
 return board

Here, at first we have defined the numbers of rows and columns and in the “create\_board” function we have created the board which is a matrix of size (ROW\_COUNT\*COLUMN\_COUNT)

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

When we place a coin in the board, we need to check if the position is right or wrong. So if the column of the upper most row is zero, then we can place a move in that column. This task is done in “is\_valid\_location” function.

Again, after validating a correct column we need place the coin in lower most rows. This task is done in “get\_next\_open\_row” function.

def drop\_piece(board, row, col, piece):  
 board[row][col] = piece

After validating the rows and column, now we can drop the piece.

def print\_board(board):  
 print(np.flip(board, 0))

while printing the board we need to flip it because numpy starts the (0,0)th location at top-left side.

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] == piece:  
 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] == piece:  
 return True  
  
 # Check positively sloped diagonals  
 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] == piece:  
 return True  
  
 # Check negatively sloped diagonals  
 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] == piece:  
 return True

“winning\_move” function is used here for checking if the player has connected 4 coins vertically, horizontally or diagonally.

For UI design we have used PYGAME library of python.

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 + SQUARESIZE / 2)), RADIUS)  
  
 for c in range(COLUMN\_COUNT):  
 for r in range(ROW\_COUNT):  
 if board[r][c] == 1:  
 pygame.draw.circle(screen, RED, (  
 int(c \* SQUARESIZE + SQUARESIZE / 2), height - int(r \* SQUARESIZE + SQUARESIZE / 2)), RADIUS)  
 elif board[r][c] == 2:  
 pygame.draw.circle(screen, YELLOW, (  
 int(c \* SQUARESIZE + SQUARESIZE / 2), height - int(r \* SQUARESIZE + SQUARESIZE / 2)), RADIUS)  
 pygame.display.update()

**Evaluation function:**

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 += 100  
 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

def score\_position(board, piece):  
 score = 0  
  
  
 center\_array = [int(i) for i in list(board[:, COLUMN\_NUMBERS // 2])]  
 center\_count = center\_array.count(piece)  
 score += center\_count \* 3  
  
  
 for r in range(ROW\_NUMBERS):  
 row\_array = [int(i) for i in list(board[r, :])]//creating a list of a //particular row  
 for c in range(COLUMN\_NUMBERS - 3):  
 window = row\_array[c:c + WINDOW\_SIZE]  
 score += evaluate\_window(window, piece)//to check in a particular row how many pieces are there   
  
  
 for c in range(COLUMN\_NUMBERS):  
 col\_array = [int(i) for i in list(board[:, c])]   
  
 for r in range(ROW\_NUMBERS - 3):  
 window = col\_array[r:r + WINDOW\_SIZE]  
 score += evaluate\_window(window, piece)  
  
  
 for r in range(ROW\_NUMBERS - 3):  
 for c in range(COLUMN\_NUMBERS - 3):  
 window = [board[r + i][c + i] for i in range(WINDOW\_SIZE)] //diagonally  
 score += evaluate\_window(window, piece)  
  
 for r in range(ROW\_NUMBERS - 3):  
 for c in range(COLUMN\_NUMBERS - 3):  
 window = [board[r + 3 - i][c + i] for i in range(WINDOW\_SIZE)]//neg slope  
 score += evaluate\_window(window, piece)  
 return score

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

Here, we are checking all rows, columns horizontally and diagonally (positive slope and negative slope) and counting the number of coins in that particular row and column and based on the counting we evaluate scores for every combinations and in the “Pick\_best\_move” function we get the best column.

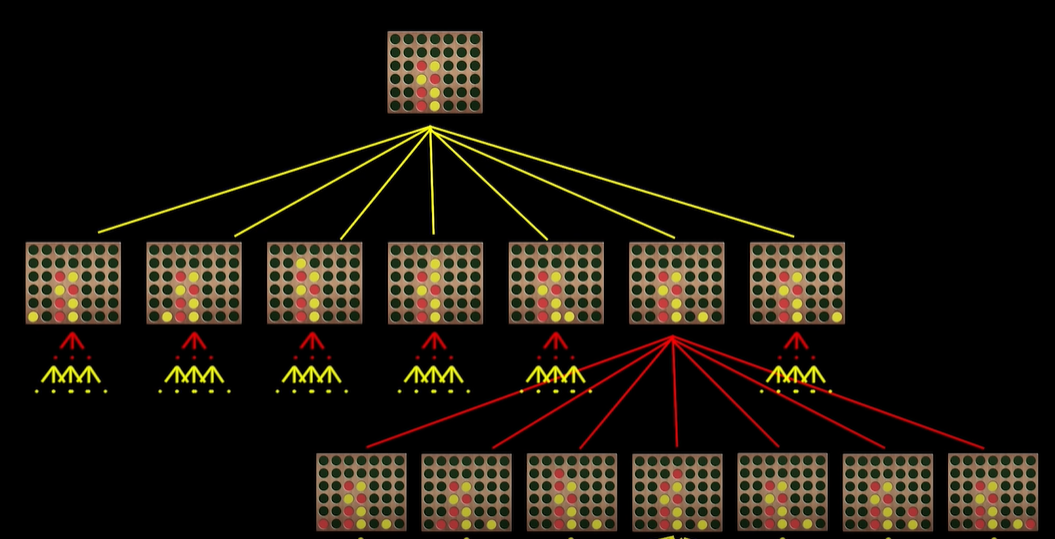
def minimax(board, depth, alpha, beta, maximizingPlayer):  
 valid\_locations = get\_valid\_locations(board)  
 is\_terminal = is\_terminal\_node(board)  
 if depth == 0 or is\_terminal:  
 if is\_terminal:  
 if winning\_move(board, AI\_PIECE):  
 return (None, 100000000000000)  
 elif winning\_move(board, 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)  
 new\_score = minimax(b\_copy, depth - 1, alpha, beta, False)[1]  
 if new\_score > value:  
 value = new\_score  
 column = col  
 alpha = max(alpha, value)  
 if alpha >= beta:  
 break  
 return column, value  
  
 else:  
 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

Here, the Minimax Algorithm has been implemented which is a backtracking algorithm that is used in decision making and game theory to find the optimal move for a player.

In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

To call the MiniMax algorithm for AI:

col, minimax\_score = minimax(board, 5, -math.inf, math.inf, True)



Here in this demonstration the depth is 2.