Environment code

not a full version, extra features like the pacman discussed are in the works.

In [2]:

```
import numpy as np
import gymnasium as gym
from gymnasium.spaces import Box, Discrete, Tuple
from colorama import Fore
import random

class Connect4Env(gym.Env):

    def __init__(self, width=7, height=6, connect=4):
        self.num_players = 2

        self.width = width
        self.height = height
        self.connect = connect

        player_observation_space = Box(low=-1, high=1,
                                       shape=(self.width, self.height),
                                       dtype=np.int32)
        self.observation_space = player_observation_space
        self.action_space = Tuple([Discrete(self.width) for _ in range(self.num_players)

        self.state_space_size = (self.height * self.width) ** 3

        self.reset()

    def reset(self):
        """
        Initialises the Connect 4 gameboard.
        """
        self.board = np.full((self.width, self.height), -1)

        self.current_player = 0
        self.winner = None
        return self.get_player_observations()

    #  0 - empty
    # -1 - one player
    #  1 - another player
    def get_player_observations(self):
        transformed_array = np.array([list(map(lambda x: 0 if x == -1 else -1 if x == 0

        transposed_array =  np.rot90(transformed_array, k=1)

        return transposed_array

    def step(self, movecol):
        """
        Applies a move by a player to the game board in a format which is suitable for a
        """
        if not(movecol >= 0 and movecol <= self.width and self.board[movecol][self.heigh
            raise IndexError(f'Invalid move. tried to place a chip on column {movecol} w
        row = self.height - 1
        while row >= 0 and self.board[movecol][row] == -1:
            row -= 1

        row += 1
```

```python
        self.board[movecol][row] = self.current_player
        self.current_player = 1 - self.current_player

        self.winner, reward_vector = self.check_for_episode_termination(movecol, row)

        info = {'legal_actions': self.get_moves(),
                'current_player': self.current_player}
        return self.get_player_observations(), reward_vector, \
               self.winner is not None, info

    def check_for_episode_termination(self, movecol, row):
        """
        Check for victories in the current state and generate rewards for the state
        """
        winner, reward_vector = self.winner, [0, 0]
        if self.does_move_win(movecol, row):
            winner = 1 - self.current_player
            if winner == 0: reward_vector = [1, -1]
            elif winner == 1: reward_vector = [-1, 1]
        elif self.get_moves() == []:  # A draw has happened
            winner = -1
        return winner, reward_vector

    def clone(self):
        st = Connect4Env(width=self.width, height=self.height)
        st.current_player = self.current_player
        st.winner = self.winner
        st.board = np.array([self.board[col][:] for col in range(self.width)])
        return st

    def get_moves(self):
        """
        :returns: array with columns where there is a possible move
        """
        if self.winner is not None:
            return []
        return [col for col in range(self.width) if self.board[col][self.height - 1] ==

    def does_move_win(self, x, y, me=None):
        """
        Checks whether a newly dropped chip at position param x, param y
        wins the game.
        """
        if me is None:
            me = self.board[x][y]
        for dx, dy in [(0, +1), (+1, +1), (+1, 0), (+1, -1)]:
            p = 1 # positive direction
            while self.is_on_board(x+p*dx, y+p*dy) and self.board[x+p*dx][y+p*dy] == me:
                p += 1
            n = 1 # negative direction
            while self.is_on_board(x-n*dx, y-n*dy) and self.board[x-n*dx][y-n*dy] == me:
                n += 1

            if p + n >= (self.connect + 1): # want (p-1) + (n-1) + 1 >= 4, or more simpl
                return True

        return False

    # swaps one random token of player A with another random token of player B
    def swap_random_tokens(self):
        while True:
            old_col = random.choice(range(self.height))
            old_row = random.choice(range(self.width))
            old_tile_val = self.board[old_row][old_col]

            # print("old_col: ", old_col)
```

```python
            # print("old_row: ", old_row)
            # print("old_tile_val: ", old_tile_val)

            if old_tile_val == -1:

                # print("old_tile_val is -1")

                continue

            new_col = random.choice(range(self.height))
            new_row = random.choice(range(self.width))
            new_tile_val = self.board[new_row][new_col]

            # print("new_col: ", new_col)
            # print("new_row: ", new_row)
            # print("new_tile_val: ", new_tile_val)


            if new_tile_val == -1 or new_tile_val == old_tile_val:

                # print("new tile val is -1 or same as the other")

                continue

            if self.does_move_win(new_row, new_col, me=old_tile_val):

                # print("move wins")

                continue

            if self.does_move_win(old_row, old_col, me=new_tile_val):

                # print("move wins")

                continue

            # print("yeeeeeee")

            self.board[old_row][old_col], self.board[new_row][new_col] = self.board[new_
            break

    def is_on_board(self, x, y):
        return x >= 0 and x < self.width and y >= 0 and y < self.height

    def get_result(self, player):
        if self.winner == -1: return 0  # A draw occurred
        return +1 if player == self.winner else -1

    def render(self, mode='human'):
        if mode != 'human': raise NotImplementedError('Rendering has not been coded yet'
        s = ""
        for x in range(self.height - 1, -1, -1):
            for y in range(self.width):
                s += {-1: Fore.WHITE + '.', 0: Fore.RED + 'X', 1: Fore.YELLOW + 'O'}[sel
                s += Fore.RESET
            s += "\n"
        print(s)
```

MCTS (Monte Carlo Tree Search)

In [3]:
```python
import numpy as np
import random
import math
```

```python
import copy
import pickle
from multiprocessing import Pool, cpu_count, Manager

class Node:
    def __init__(self, state, parent=None, move=None):
        self.state = state
        self.parent = parent
        self.move = move
        self.children = []
        self.visits = 0
        self.value = 0

    def is_fully_expanded(self):
        return len(self.children) == len(self.state.get_moves())

    def best_child(self, c_param=1.4):
        choices_weights = [
            (child.value / child.visits) + c_param * math.sqrt((2 * math.log(self.visits
            for child in self.children
        ]
        return self.children[np.argmax(choices_weights)]

    def most_visited_child(self):
        visits = [child.visits for child in self.children]
        return self.children[np.argmax(visits)]

    def expand(self):
        moves = self.state.get_moves()
        for move in moves:
            if not any(child.move == move for child in self.children):
                next_state = copy.deepcopy(self.state)
                next_state.step(move)
                child_node = Node(next_state, self, move)
                self.children.append(child_node)
                return child_node

    def rollout(self):
        current_state = copy.deepcopy(self.state)
        while current_state.winner is None:
            possible_moves = current_state.get_moves()
            move = random.choice(possible_moves)
            current_state.step(move)
        return current_state.get_result(1 - current_state.current_player)

    def backpropagate(self, result):
        self.visits += 1
        self.value += result
        if self.parent:
            self.parent.backpropagate(-result)

class MCTS:
    def __init__(self, env, q_table, num_simulations=1000):
        self.env = env
        self.q_table = q_table
        self.num_simulations = num_simulations

    def search(self, state):
        root = Node(state)

        for _ in range(self.num_simulations):
            node = root
            while node.is_fully_expanded() and node.children:
                node = node.best_child()

            if not node.is_fully_expanded():
```

```python
                node = node.expand()

            result = node.rollout()
            node.backpropagate(result)
            self.update_q_table(node)

        return root.most_visited_child().move

    def update_q_table(self, node):
        while node:
            state_str = ''.join(map(str, node.state.board.flatten()))
            key = (state_str, node.move)
            if key not in self.q_table:
                self.q_table[key] = 0
            self.q_table[key] += node.value
            node = node.parent

episode = 0
def get_state_action_key(state, action):
    state_str = ''.join(map(str, state.flatten()))
    return (state_str, action)


def run_episode(args):
    global episode
    episode += 1
    q_table, num_simulations = args
    env = Connect4Env()
    mcts = MCTS(env, q_table, num_simulations=num_simulations)

    state = env.reset()
    done = False
    while not done:
        if env.current_player == 0:
            action = mcts.search(env.clone())
        else:
            possible_moves = env.get_moves()
            action = random.choice(possible_moves)

        next_state, reward, done, info = env.step(action)

        key = get_state_action_key(state, action)
        if key not in q_table:
            q_table[key] = 0

        q_table[key] += reward[env.current_player]
        state = next_state

        if done:
            break
    print(f'episode {episode} done')

    return q_table

def train_mcts(num_episodes=100, num_simulations=1000):
    with Manager() as manager:
        q_table = manager.dict()

        with Pool(processes=cpu_count()) as pool:
            q_tables = pool.map(run_episode, [(q_table, num_simulations) for _ in range(

        q_table = dict(q_table)  # Convert manager.dict() to a regular dictionary

        with open('q_table.pkl', 'wb') as f:
            pickle.dump(q_table, f)
```

```
    print("Q-table generated and saved to 'q_table.pkl'")
```

minimax

In [4]:
```
## minimax
import math
import multiprocessing as mp

def minimax(node, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or node.winner is not None:
        return node.get_result(maximizingPlayer)

    valid_moves = node.get_moves()
    if maximizingPlayer:
        value = -math.inf
        for move in valid_moves:
            child = node.clone()
            child.step(move)
            value = max(value, minimax(child, depth-1, alpha, beta, False))
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return value
    else:
        value = math.inf
        for move in valid_moves:
            child = node.clone()
            child.step(move)
            value = min(value, minimax(child, depth-1, alpha, beta, True))
            beta = min(beta, value)
            if alpha >= beta:
                break
        return value

def minimax_worker(child, depth, alpha, beta, maximizingPlayer, return_dict, idx):
    return_dict[idx] = minimax(child, depth, alpha, beta, maximizingPlayer)

def minimax_decision(env, depth=4):
    manager = mp.Manager()
    return_dict = manager.dict()
    jobs = []
    best_value = -math.inf
    best_move = None

    valid_moves = env.get_moves()
    for idx, move in enumerate(valid_moves):
        child = env.clone()
        child.step(move)
        p = mp.Process(target=minimax_worker, args=(child, depth-1, -math.inf, math.inf,
        jobs.append(p)
        p.start()

    for job in jobs:
        job.join()

    for idx, move in enumerate(valid_moves):
        value = return_dict[idx]
        if value > best_value:
            best_value = value
            best_move = move

    return best_move
```

scenarios

In [5]:

```
def minimax_vs_mcts(q_table, minimax_depth=4):
    env = Connect4Env()
    mcts = MCTS(env, q_table, num_simulations=1000)

    state = env.reset()
    done = False

    print("Starting a game of Connect 4!")
    env.render()

    while not done:
        if env.current_player == 0:
            print("MCTS Agent's turn:")
            action = mcts.search(env.clone())
        else:
            print("Minimax Agent's turn:")
            action = minimax_decision(env.clone(), minimax_depth)

        state, reward, done, info = env.step(action)
        env.render()

        if done:
            if env.winner == -1:
                print("It's a draw!")
            elif env.winner == 0:
                print("MCTS Agent wins!")
            else:
                print("Minimax Agent wins!")
            break
```

In [7]:

```
def mcts_vs_mcts(q_table):
    env = Connect4Env()
    mcts = MCTS(env.clone(), q_table, num_simulations=1000)
    mcts2 = MCTS(env.clone(), q_table, num_simulations=1000)
    state = env.reset()
    done = False

    print("Starting a game of Connect 4!")
    env.render()

    while not done:
        if env.current_player == 0:
            print("MCTS1 Agent's turn:")
            action = mcts.search(env.clone())
        else:
            print("MCTS2 Agent's turn:")
            action = mcts2.search(env.clone())

        state, reward, done, info = env.step(action)
        env.render()

        if done:
            if env.winner == -1:
                print("It's a draw!")
            elif env.winner == 0:
                print("MCTS1 Agent wins!")
            else:
                print("MCTS2 Agent wins!")
            break
```

In [10]:
```
def human_vs_mcts(q_table):
    env = Connect4Env()
    mcts = MCTS(env, q_table, num_simulations=1000)

    state = env.reset()
    done = False

    print("Starting a game of Connect 4!")
    env.render()

    while not done:
        if env.current_player == 0:
            print("MCTS Agent's turn:")
            action = mcts.search(env.clone())
        else:
            valid_move = False
            while not valid_move:
                try:
                    action = int(input("Your turn! Enter the column number (0-6): "))
                    if action in env.get_moves():
                        valid_move = True
                    else:
                        print("Invalid move. Column is full or out of range. Try again."
                except ValueError:
                    print("Invalid input. Please enter a number between 0 and 6.")

        state, reward, done, info = env.step(action)
        env.render()

        if done:
            if env.winner == -1:
                print("It's a draw!")
            elif env.winner == 0:
                print("MCTS Agent wins!")
            else:
                print("You win!")
            break
```

Agent vs Agent play

Both agents utilize the same Q-table. Agents are able to draw, agents are able to win, which means this is not a fully optimal solution since we know the game is solved, so we are not fully ready with that.

Below we are able to run the code for 2 agents to play against each other with a pre-trained Q-table. Settings: 100 episodes, 1000 simulation depth.

In [12]:
```
# train_mcts(num_episodes=100, num_simulations=1000)

# Load the trained Q-table
with open('q_table.pkl', 'rb') as f:
    q_table = pickle.load(f)

mcts_vs_mcts(q_table)
```

```
Starting a game of Connect 4!
.......
.......
.......
.......
.......
.......
```

```
MCTS1 Agent's turn:
.......
.......
.......
.......
.......
X......

MCTS2 Agent's turn:
.......
.......
.......
.......
O......
X......

MCTS1 Agent's turn:
.......
.......
.......
.......
O......
X..X...

MCTS2 Agent's turn:
.......
.......
.......
.......
O......
XO.X...

MCTS1 Agent's turn:
.......
.......
.......
X......
O......
XO.X...

MCTS2 Agent's turn:
.......
.......
.......
X......
OO.....
XO.X...

MCTS1 Agent's turn:
.......
.......
.......
XX.....
OO.....
XO.X...

MCTS2 Agent's turn:
.......
.......
.......
XX.....
OO.....
XOOX...

MCTS1 Agent's turn:
.......
```

```
.......
.X.....
XX.....
OO.....
XOOX...

MCTS2 Agent's turn:
.......
.......
.X.....
XX.....
OO.....
XOOXO..

MCTS1 Agent's turn:
.......
.......
.X.....
XX.....
OOX....
XOOXO..

MCTS2 Agent's turn:
.......
.......
OX.....
XX.....
OOX....
XOOXO..

MCTS1 Agent's turn:
.......
.X.....
OX.....
XX.....
OOX....
XOOXO..

MCTS2 Agent's turn:
.O.....
.X.....
OX.....
XX.....
OOX....
XOOXO..

MCTS1 Agent's turn:
.O.....
XX.....
OX.....
XX.....
OOX....
XOOXO..

MCTS2 Agent's turn:
.O.....
XX.....
OX.....
XX.....
OOX....
XOOXOO.

MCTS1 Agent's turn:
.O.....
XX.....
OX.....
```

```
XXX....
OOX....
XOOXOO.

MCTS2 Agent's turn:
.O.....
XX.....
OXO....
XXX....
OOX....
XOOXOO.

MCTS1 Agent's turn:
.O.....
XX.....
OXO....
XXX....
OOXX...
XOOXOO.

MCTS1 Agent wins!
```

In [ ]: