Connect4 Sprint 1:

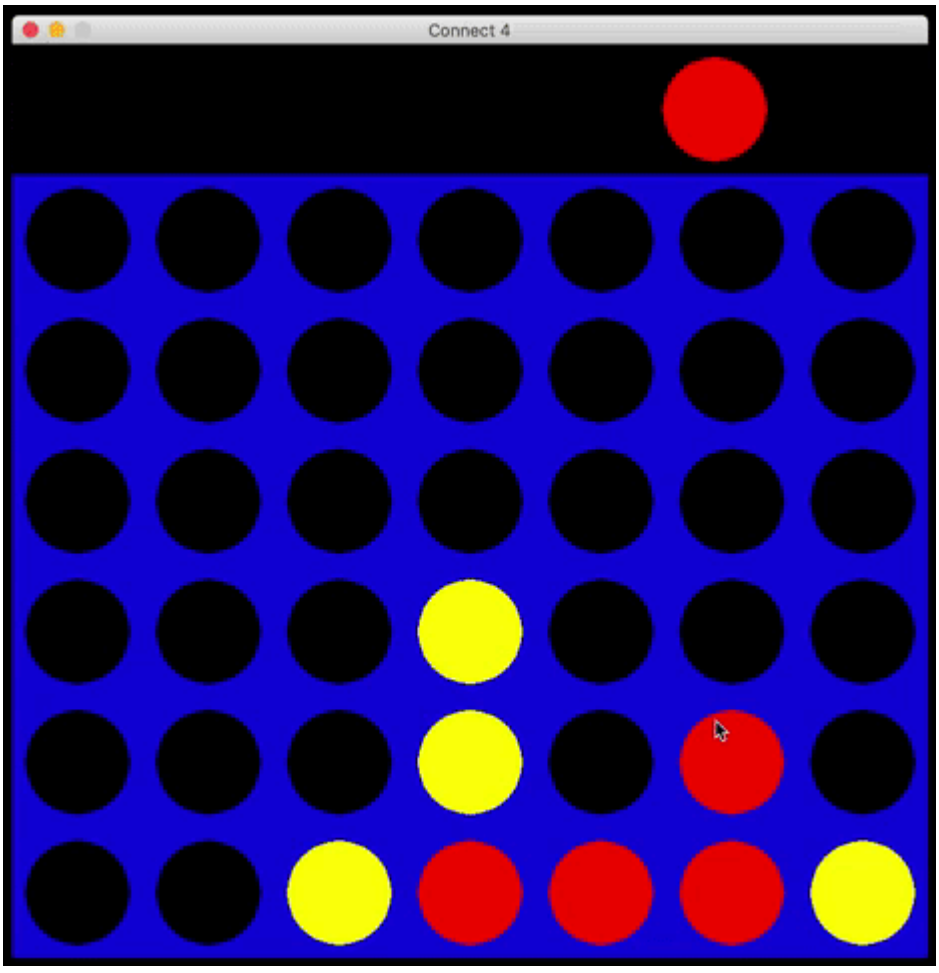
By: Yordan Mitev & Vladislav Stefanov

Date: 2024-05-20

Introduction:

Connect4 is a game in which 2 players choose a color and take turns dropping colored tokens into a grid with raws and columns. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of n (4 by default) of one's own tokens.

Connect Four is a solved game. The first player can always win by playing a certain strategy.



Game Customization:

We take the vanilla game of Connect 4 and we will first try to find the optimal strategy for a vanilla configuration, and then scale it up to a larger grid size, with

larger required chains.

Moreover, we can add stochastic variables like a chance for a token to not go in its right column, or redirect the inputs of certain columns to go to others, but that is to come in the next sprint.

External Contibutions:

In order to understand the problem in details and how to approach it, we utilized an open-source solution by Daniel Hernandez. We have investigated it, researched the software package and collaborated to understand its way of operation.

In the next sprint we will alter, update, improve and extend it. In this sprint our main focus was to get us involved with the gymnasium, advance our skills in building MDPs, and understand better what is happening both conceptually and practically.

Starting Conditions:

Empty playing field where rows and columns counts can be changed by the user. Additionally, the number of tokens to be connected (4 by default) can be given by the user. There is no winner and all positions are empty. One of the players starts first.

Environment:

The environment defines the rules of the game and the playing field:

- the width of the grid
- the height of the grid
- the number of tokens to connect
- the number of players
- the possible positions in the grid

And the environment also holds the current state of the game and actions possible.

States:

It is constructed out of:

- All tokens' positions
- Tokens' positions of Player A
- Tokens' positions of Player B

A naïve calculation of the state space size will be: `(X * Y) ^ 3`, also taking into account invalid states where a token rests above an empty cell.

Actions:

The action space is defined as:

- Releaseing a token in a column which:
  - is in bounds of the playing field.
  - is not full (i.e., there is at least one free spot in the column).

Rewards:

The rewards provided by the environment are:

- Reward for the winner: +1
- Reward for the loser: -1
- Reward when there is a draw: 0

Terminal states:

A terminal state can be reached in the following ways:

- Player A creates a connect4 (i.e., four connected tokens vertically, horizontally, diagonally)
- Player B creates a connect4 (vertically, horizontally, diagonally)
- All fields are filled up and no connect4 exists

Episode Length:

Episode length is not needed to be strictly enforced due to the fact that a victory condition can easily be reached and the board size is not prohibitively large.

Contributions:

- Yordan: I looked at possible ways to implement the MDP and different ways to make it work for our use case. Also, I worked together with Vladislav on understanding and implementing the base MDP.
- Vladislav: I worked with Yordan on the understanding of the code and concepts in RL. Additionally, I was thinking about possible extentions in the next sprint.

Roles:

Both of us are directly involved with the whole process - analysis, design, research, software development, document writing. We discuss and collaborate to ensure that everybody is on the same page all the time and is involved with the project.

Appendices

Appendix A: Code

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


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=0, high=1,
                                       shape=(self.num_players +
1,
                                              self.width,
self.height),
                                       dtype=np.int32)
        self.observation_space = Tuple([player_observation_space
                                        for _ in
range(self.num_players)])
        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()

    def filter_observation_player_perspective(self, player: int):
        opponent = 0 if player == 1 else 1
        empty_positions = np.where(self.board == -1, 1, 0)
        player_chips    = np.where(self.board == player, 1, 0)
        opponent_chips = np.where(self.board == opponent, 1, 0)
        return np.array([empty_positions, player_chips,
opponent_chips])

    def get_player_observations(self) -> list[np.ndarray]:
```

```
        p1_state = self.filter_observation_player_perspective(0)
        p2_state = np.array([np.copy(p1_state[0]),
                             np.copy(p1_state[-1]),
np.copy(p1_state[-2])])
        return [p1_state, p2_state]

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

        row += 1

        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 get_moves(self):
        """
        List 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] == -1]
```

```
    def does_move_win(self, x, y):
        """
        Checks whether a newly dropped chip at position param x,
param y
        wins the game.
        """
        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 simply p + n >- 5
                return True

        return False

    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
```

References:

https://github.com/Danielhp95/gym-connect4

https://en.wikipedia.org/wiki/Connect_Four

https://papergames.io/en/connect4

https://web.mit.edu/sp.268/www/2010/connectFourSlides.pdf