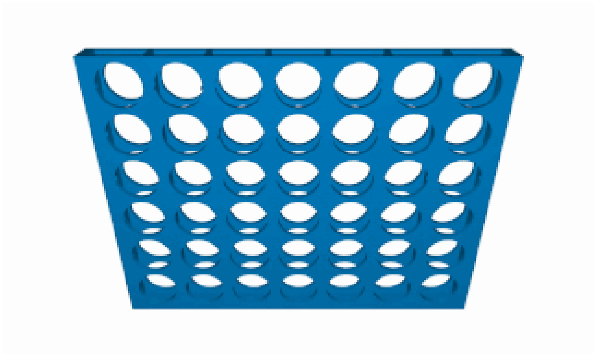
Introduction

Connect Four is a game where two players alternate turns dropping colored discs into a vertical grid. Each player uses a different color (usually red or yellow), and the objective of the game is to be the first player to get four discs in a row.



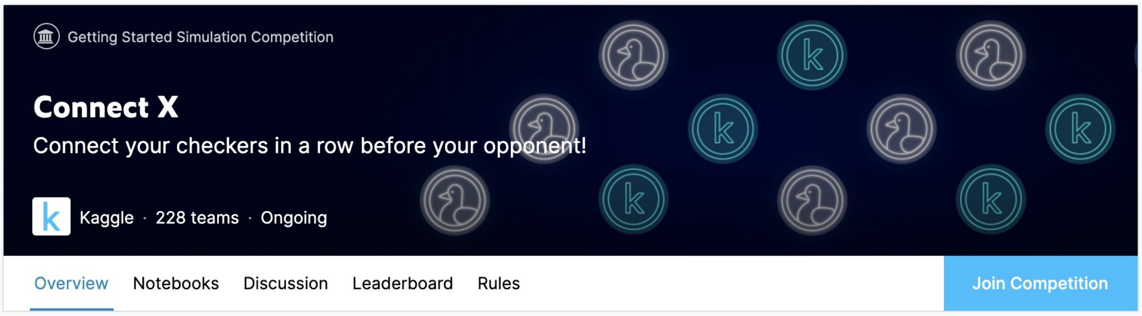
In this course, you will build your own intelligent agents to play the game.

- In the first lesson, you'll learn how to set up the game environment and create your first agent.
- The next two lessons focus on traditional methods for building game AI. These agents will be smart enough to defeat many novice players!
- In the final lesson, you'll experiment with cutting-edge algorithms from the field of reinforcement learning. The agents that you build will come up with gameplay strategies much like humans do: gradually, and with experience.

Join the competition

Throughout the course, you'll test your agents' performance by competing against agents that other users have created.

To join the competition, open a new window with the competition page, and click on the **"Join Competition"** button. (*If you see a "Submit Agent" button instead of a "Join Competition" button, you have already joined the competition, and don't need to do so again.*)



Getting Started Simulation Competition

Connect X

Connect your checkers in a row before your opponent!

Kaggle · 228 teams · Ongoing

This takes you to the rules acceptance page. You must accept the competition rules in order to participate. These rules govern how many submissions you can make per day, the maximum team size, and other competition-specific details. Then, click on **"I Understand and Accept"** to indicate that you will abide by the competition rules.

Getting started

The game environment comes equipped with agents that have already been implemented for you. To see a list of these default agents, run:

In [ ]:

```
from kaggle_environments import make, evaluate
```

```
# Create the game environment
# Set debug=True to see the errors if your agent refuses to run
env = make("connectx", debug=True)

# List of available default agents
print(list(env.agents))
```

The `"random"` agent selects (uniformly) at random from the set of **valid moves**. In Connect Four, a move is considered valid if there's still space in the column to place a disc (i.e., if the board has seven rows, the column has fewer than seven discs).

In the code cell below, this agent plays one game round against a copy of itself.

In [ ]:

```
# Two random agents play one game round
env.run(["random", "random"])

# Show the game
env.render(mode="ipython")
```

You can use the player above to view the game in detail: every move is captured and can be replayed. *Try this now!*

As you'll soon see, this information will prove incredibly useful for brainstorming ways to improve our agents.

Defining agents

To participate in the competition, you'll create your own agents.

Your agent should be implemented as a Python function that accepts two arguments: `obs` and `config`. It returns an integer with the selected column, where indexing starts at zero. So, the returned value is one of 0-6, inclusive.

We'll start with a few examples, to provide some context. In the code cell below:

- The first agent behaves identically to the `"random"` agent above.
- The second agent always selects the middle column, whether it's valid or not! Note that if any agent selects an invalid move, it loses the game.
- The third agent selects the leftmost valid column.

In [ ]:

```
import random
import numpy as np
```

In [ ]:

```
# Selects random valid column
def agent_random(obs, config):
    valid_moves = [col for col in range(config.columns) if obs.board[col] == 0]
    # égal à 0 signifie que la case est vide, est disponible
    # si la case est occupé celle-ci contient 1 ou 2
    return random.choice(valid_moves)

# Selects middle column
def agent_middle(obs, config):
    return config.columns//2

# Selects leftmost valid column
def agent_leftmost(obs, config):
    # config.columns is the number of columns
    valid_moves = [col for col in range(config.columns) if obs.board[col] == 0]
    return valid_moves[0]
```

So, what are `obs` and `config`, exactly?

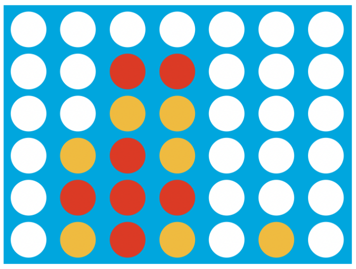
So, what are `obs` and `config`, exactly?

`obs`

`obs` contains two pieces of information:

- `obs.board` - the game board (a Python list with one item for each grid location)
- `obs.mark` - the piece assigned to the agent (either `1` or `2`)

`obs.board` is a Python list that shows the locations of the discs, where the first row appears first, followed by the second row, and so on. We use `1` to track player 1's discs, and `2` to track player 2's discs. For instance, for this game board:



`obs.board` would be `[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 1, 2, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 2, 1, 2, 0, 2, 0]`.

`config`

`config` contains three pieces of information:

- `config.columns` - number of columns in the game board (`7` for Connect Four)
- `config.rows` - number of rows in the game board (`6` for Connect Four)
- `config.inarow` - number of pieces a player needs to get in a row in order to win (`4` for Connect Four)

Take the time now to investigate the three agents we've defined above. Make sure that the code makes sense to you!

Evaluating agents

To have the custom agents play one game round, we use the same `env.run()` method as before.

In [ ]:

```
# Agents play one game round
env.run([agent_leftmost, agent_random])

# Show the game
env.render(mode="ipython")
```

The outcome of a single game is usually not enough information to figure out how well our agents are likely to perform. To get a better idea, we'll calculate the win percentages for each agent, averaged over multiple games. For fairness, each agent goes first half of the time.

To do this, we'll use the `get_win_percentages()` function (defined in a hidden code cell). *To view the details of this function, click on the "Code" button below.*

In [ ]:

```
def get_win_percentages(agent1, agent2, n_rounds=100):
    # Use default Connect Four setup
    config = {'rows': 6, 'columns': 7, 'inarow': 4}
    # Agent 1 goes first (roughly) half the time
    outcomes = evaluate("connectx", [agent1, agent2], config, [], n_rounds//2)
    # Agent 2 goes first (roughly) half the time
    outcomes += [[b,a] for [a,b] in evaluate("connectx", [agent2, agent1], config, [], n
_rounds-n_rounds//2)]
    print("Agent 1 Win Percentage:", np.round(outcomes.count([1,-1])/len(outcomes), 2))
    print("Agent 2 Win Percentage:", np.round(outcomes.count([-1,1])/len(outcomes), 2))
    print("Number of Invalid Plays by Agent 1:", outcomes.count([None, 0]))
    print("Number of Invalid Plays by Agent 2:", outcomes.count([0, None]))
```

In [ ]:

```
evaluate("connectx", [agent_middle, agent_random], {'rows': 6, 'columns': 7, 'inarow': 4
}, [], 100//2)
```

Which agent do you think performs better against the random agent: the agent that always plays in the middle (`agent_middle`), or the agent that chooses the leftmost valid column (`agent_leftmost`)? Let's find out!

In [ ]:

```
get_win_percentages(agent1=agent_middle, agent2=agent_random)
```

In [ ]:

```
get_win_percentages(agent1=agent_leftmost, agent2=agent_random)
```

It looks like the agent that chooses the leftmost valid column performs best!

Your turn

These agents are quite simple. As the course progresses, you'll create increasingly complex agents! Continue to make your first competition submission.

Have questions or comments? Visit the Learn Discussion forum to chat with other Learners.