Gaming AI & Reinforcement Learning: 4th lesson – Deep Reinforcement Learning

So far, our agents have relied on detailed information about how to play the game. The heuristic really provides a lot of guidance about how to select moves!

In this tutorial, you'll learn how to use **reinforcement learning** to build an intelligent agent without the use of a heuristic. Instead, we will gradually refine the agent's strategy over time, simply by playing the game and trying to maximize the winning rate.

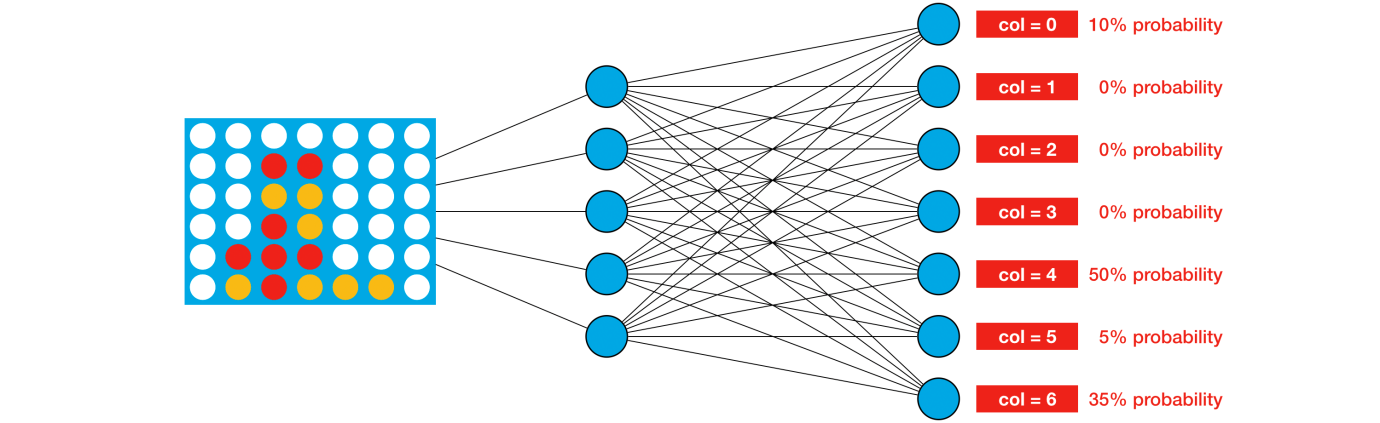
In this notebook, we won't be able to explore this complex field in detail, but you'll learn about the big picture and explore code that you can use to train your own agent.

***Neural networks***

It's difficult to come up with a perfect heuristic. Improving the heuristic generally entails playing the game many times, to determine specific cases where the agent could have made better choices. And, it can prove challenging to interpret what exactly is going wrong, and ultimately to fix old mistakes without accidentally introducing new ones.

Wouldn't it be much easier if we had a more systematic way of improving the agent with gameplay experience?

In this tutorial, towards this goal, we’ll replace the heuristic with a neural network. The network accepts the current board as input. And, it outputs probability for each possible move.



Then, the agent selects a move by sampling from these probabilities. For instance, for the game board in the image above, the agent selects column 4 with 50% probability. This way, to encode a good gameplay strategy, we need only amend the weights of the network so that for every possible game board, it assigns higher probabilities to better moves. At least in theory, that's our goal. In practice, we won't actually check if that's the case; since remember that *Connect Four* has over 4 trillion possible game boards!

***Setup***

How can we approach the task of amending the weights of the network, in practice? Here's the approach we'll take in this lesson:

* After each move, we give the agent a *reward* that tells it how well it did.
* If the agent wins the game in that move, we give it a reward of 1 point.
* Else if the agent plays an invalid move (which ends the game), we give it a reward of -10 points.
* Else if the opponent wins the game in its next move (i.e., the agent failed to prevent its opponent from winning), we give the agent a reward of -1 point.
* Else, the agent gets a reward of 1/42.
* At the end of each game, the agent adds up its reward. We refer to the sum of rewards as the agent's *cumulative reward*.
* For instance, if the game lasted 8 moves (each player played four times), and the agent ultimately won, then its cumulative reward is 3\*(1/42) + 1.
* If the game lasted 11 moves (and the opponent went first, so the agent played five times), and the opponent won in its final move, then the agent's cumulative reward is 4\*(1/42) - 1.
* If the game ends in a draw, then the agent played exactly 21 moves, and it gets a cumulative reward of 21\*(1/42).
* If the game lasted 7 moves and ended with the agent selecting an invalid move, the agent gets a cumulative reward of 3\*(1/42) - 10.

Our goal is to find the weights of the neural network that (on average) maximize the agent's cumulative reward.

This idea of using reward to track the performance of an agent is a core idea in the field of reinforcement learning. Once we define the problem in this way, we can use any of a variety of reinforcement learning algorithms to produce an agent.

***Reinforcement learning***

There are many different reinforcement learning algorithms, such as DQN, A2C, and PPO, among others. All of these algorithms use a similar process to produce an agent:

* Initially, the weights are set to random values.
* As the agent plays the game, the algorithm continually tries out new values for the weights, to see how the cumulative reward is affected, on average. Over time, after playing many games, we get a good idea of how the weights affect cumulative reward, and the algorithm settles towards weights that performed better.
* This way, we'll end up with an agent that tries to win the game (so it gets the final reward of 1 point, and avoids the -1 point and -10 points) and tries to make the game last as long as possible (so that it collects the 1/42 bonus as many times as it can).

\*You might argue that it doesn't really make sense to want the game to last as long as possible; this might result in a very inefficient agent that doesn't play obvious winning moves early in gameplay. And, your intuition would be correct; this will make the agent take longer to play a winning move! The reason we include the 1/42 bonus is to help the algorithms we'll use to converge better. Further discussion is outside of the scope of this course, but you can learn more by reading about the "temporal credit assignment problem" and "reward shaping".

***Code***

There are a lot of great implementations of reinforcement learning algorithms online. In this course, we'll use Stable-Baselines3. There's a bit of extra work that we need to do to make the environment compatible with Stable Baselines. For this, we define the ConnectFourGym class below. This class implements ConnectX as an OpenAI Gym environment and uses several methods:

* reset() will be called at the beginning of every game. It returns the starting game board as a 2D numpy array with 6 rows and 7 columns.
* change\_reward() customizes the rewards that the agent receives (the competition already has its own system for rewards that are used to rank the agents, and this method changes the values to match the rewards system we designed).
* step() is used to play the agent's choice of action (supplied as action), along with the opponent's response. It returns:
* The resulting game board (as a numpy array)
* The agent’s reward (from the most recent move only: one of +1, -10, -1, or 1/42)
* Whether or not the game has ended (if the game has ended, done=True; otherwise done=False)

import random

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

%matplotlib inline

import gym

from kaggle\_environments import make, evaluate

from gym import spaces

class ConnectFourGym(gym.Env):

def \_\_init\_\_(self, agent2="random"):

ks\_env = make("connectx", debug=True)

self.env = ks\_env.train([None, agent2])

self.rows = ks\_env.configuration.rows

self.columns = ks\_env.configuration.columns

*# Learn about spaces here: http://gym.openai.com/docs/#spaces*

self.action\_space = spaces.Discrete(self.columns)

self.observation\_space = spaces.Box(low=0, high=2,

shape=(1,self.rows,self.columns), dtype=int)

*# Tuple corresponding to the min and max possible rewards*

self.reward\_range = (-10, 1)

*# StableBaselines throws error if these are not defined*

self.spec = None

self.metadata = None

def reset(self):

self.obs = self.env.reset()

return np.array(self.obs['board']).reshape(1,self.rows,self.columns)

def change\_reward(self, old\_reward, done):

if old\_reward == 1: *# The agent won the game*

return 1

elif done: *# The opponent won the game*

return -1

else: *# Reward 1/42*

return 1/(self.rows\*self.columns)

def step(self, action):

*# Check if agent's move is valid*

is\_valid = (self.obs['board'][int(action)] == 0)

if is\_valid: *# Play the move*

self.obs, old\_reward, done, \_ = self.env.step(int(action))

reward = self.change\_reward(old\_reward, done)

else: *# End the game and penalize agent*

reward, done, \_ = -10, True, {}

return np.array(self.obs['board']).reshape(1,self.rows,self.columns), reward, done, \_

In this notebook, we'll train an agent to beat the random agent. We specify this opponent in the agent2 argument below.

Loading environment lux\_ai\_s2 failed: No module named 'vec\_noise'

*# Create ConnectFour environment*

env = ConnectFourGym(agent2="random")

The next step is to specify the architecture of the neural network. In this case, we use a convolutional neural network. Note that this is the neural network that outputs the probabilities of selecting each column. Since we use the PPO algorithm (PPO in the code cell below), our network will also output some additional information (called the "value" of the input). This is outside the scope of this course, but you can learn more by reading about "actor-critic networks".

import torch as th

import torch.nn as nn

!pip install "stable-baselines3"

from stable\_baselines3 import PPO

from stable\_baselines3.common.torch\_layers import BaseFeaturesExtractor

*# Neural network for predicting action values*

class CustomCNN(BaseFeaturesExtractor):

def \_\_init\_\_(self, observation\_space: gym.spaces.Box, features\_dim: int=128):

super(CustomCNN, self).\_\_init\_\_(observation\_space, features\_dim)

*# CxHxW images (channels first)*

n\_input\_channels = observation\_space.shape[0]

self.cnn = nn.Sequential(

nn.Conv2d(n\_input\_channels, 32, kernel\_size=3, stride=1, padding=0),

nn.ReLU(),

nn.Conv2d(32, 64, kernel\_size=3, stride=1, padding=0),

nn.ReLU(),

nn.Flatten(),

)

*# Compute shape by doing one forward pass*

with th.no\_grad():

n\_flatten = self.cnn(

th.as\_tensor(observation\_space.sample()[None]).float()

).shape[1]

self.linear = nn.Sequential(nn.Linear(n\_flatten, features\_dim), nn.ReLU())

def forward(self, observations: th.Tensor) -> th.Tensor:

return self.linear(self.cnn(observations))

policy\_kwargs = dict(

features\_extractor\_class=CustomCNN,

)

*# Initialize agent*

model = PPO("CnnPolicy", env, policy\_kwargs=policy\_kwargs, verbose=0)

Collecting stable-baselines3

Downloading stable\_baselines3-1.8.0-py3-none-any.whl (174 kB)

174.5/174.5 kB 4.4 MB/s eta 0:00:00

Requirement already satisfied: pandas in /opt/conda/lib/python3.7/site-packages (from stable-baselines3) (1.3.5)

Requirement already satisfied: torch>=1.11 in /opt/conda/lib/python3.7/site-packages (from stable-baselines3) (1.13.0+cpu)

Requirement already satisfied: cloudpickle in /opt/conda/lib/python3.7/site-packages (from stable-baselines3) (2.2.1)

Collecting importlib-metadata~=4.13

Downloading importlib\_metadata-4.13.0-py3-none-any.whl (23 kB)

Collecting gym==0.21

Downloading gym-0.21.0.tar.gz (1.5 MB)

1.5/1.5 MB 26.9 MB/s eta 0:00:00

Preparing metadata (setup.py) ... - done

Requirement already satisfied: matplotlib in /opt/conda/lib/python3.7/site-packages (from stable-baselines3) (3.5.3)

Requirement already satisfied: numpy in /opt/conda/lib/python3.7/site-packages (from stable-baselines3) (1.21.6)

Requirement already satisfied: typing-extensions<5,>=4.0 in /opt/conda/lib/python3.7/site-packages (from stable-baselines3) (4.4.0)

Requirement already satisfied: zipp>=0.5 in /opt/conda/lib/python3.7/site-packages (from importlib-metadata~=4.13->stable-baselines3) (3.11.0)

Requirement already satisfied: fonttools>=4.22.0 in /opt/conda/lib/python3.7/site-packages (from matplotlib->stable-baselines3) (4.38.0)

Requirement already satisfied: packaging>=20.0 in /opt/conda/lib/python3.7/site-packages (from matplotlib->stable-baselines3) (23.0)

Requirement already satisfied: python-dateutil>=2.7 in /opt/conda/lib/python3.7/site-packages (from matplotlib->stable-baselines3) (2.8.2)

Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.7/site-packages (from matplotlib->stable-baselines3) (0.11.0)

Requirement already satisfied: pillow>=6.2.0 in /opt/conda/lib/python3.7/site-packages (from matplotlib->stable-baselines3) (9.4.0)

Requirement already satisfied: pyparsing>=2.2.1 in /opt/conda/lib/python3.7/site-packages (from matplotlib->stable-baselines3) (3.0.9)

Requirement already satisfied: kiwisolver>=1.0.1 in /opt/conda/lib/python3.7/site-packages (from matplotlib->stable-baselines3) (1.4.4)

Requirement already satisfied: pytz>=2017.3 in /opt/conda/lib/python3.7/site-packages (from pandas->stable-baselines3) (2022.7.1)

Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.7/site-packages (from python-dateutil>=2.7->matplotlib->stable-baselines3) (1.16.0)

Building wheels for collected packages: gym

Building wheel for gym (setup.py) ... - \ | done

Created wheel for gym: filename=gym-0.21.0-py3-none-any.whl size=1616821 sha256=32590118967de8177f605b8b76507663d631efe02c0d04dda57df38ea3801ca1

Stored in directory: /root/.cache/pip/wheels/d3/78/02/af51e23f21c31c0167d288296d764a22abb842ac6e8f52ebfa

Successfully built gym

Installing collected packages: importlib-metadata, gym, stable-baselines3

Attempting uninstall: importlib-metadata

Found existing installation: importlib-metadata 4.11.4

Uninstalling importlib-metadata-4.11.4:

Successfully uninstalled importlib-metadata-4.11.4

Attempting uninstall: gym

Found existing installation: gym 0.23.1

Uninstalling gym-0.23.1:

Successfully uninstalled gym-0.23.1

ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependency conflicts.

librosa 0.10.0.post2 requires soundfile>=0.12.1, but you have soundfile 0.11.0 which is incompatible.

flake8 5.0.4 requires importlib-metadata<4.3,>=1.1.0; python\_version < "3.8", but you have importlib-metadata 4.13.0 which is incompatible.

cmudict 1.0.13 requires importlib-metadata<6.0.0,>=5.1.0, but you have importlib-metadata 4.13.0 which is incompatible.

Successfully installed gym-0.21.0 importlib-metadata-4.13.0 stable-baselines3-1.8.0

WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv

/opt/conda/lib/python3.7/site-packages/botocore/httpsession.py:41: DeprecationWarning: 'urllib3.contrib.pyopenssl' module is deprecated and will be removed in a future release of urllib3 2.x. Read more in this issue: https://github.com/urllib3/urllib3/issues/2680

from urllib3.contrib.pyopenssl import orig\_util\_SSLContext as SSLContext

In the code cell above, the weights of the neural network are initially set to random values.

In the next code cell, we "train the agent", which is just another way of saying that we find weights of the neural network that are likely to result in the agent selecting good moves.

*# Train agent*

model.learn(total\_timesteps=60000)

<stable\_baselines3.ppo.ppo.PPO at 0x7b40a1c1b250>

Finally, we specify the trained agent in the format required for the competition.

def agent1(obs, config):

*# Use the best model to select a column*

col, \_ = model.predict(np.array(obs['board']).reshape(1, 6,7))

*# Check if selected column is valid*

is\_valid = (obs['board'][int(col)] == 0)

*# If not valid, select random move.*

if is\_valid:

return int(col)

else:

return random.choice([col for col **in** range(config.columns) if obs.board[int(col)] == 0])

In the next code cell, we see the outcome of one game round against a random agent.

*# Create the game environment*

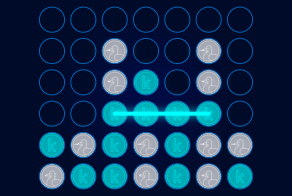
env = make("connectx")

*# Two random agents play one game round*

env.run([agent1, "random"])

*# Show the game*

env.render(mode="ipython")



And, we calculate how it performs on average, against the random agent.

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]))

get\_win\_percentages(agent1=agent1, agent2="random")

Agent 1 Win Percentage: 0.68

Agent 2 Win Percentage: 0.32

Number of Invalid Plays by Agent 1: 0

Number of Invalid Plays by Agent 2: 0

It's important to note that the agent that we've created here was only trained to beat the random agent, because all of its gameplay experience has been with the random agent as opponent.

If we want to produce an agent that reliably performs better than many other agents, we have to expose our agent to these other agents during training. To learn more about how to do this, you can read about self-play.