Chapter 11

Simulation and optimization competitions

# Challenges of Reinforcement Learning

Reinforcement Learning is an interesting case among the different branches of machine learning. On the one hand, it is quite demanding from a technical standpoint: various intuition from supervised learning do not hold, and the associated mathematical apparatus is quite a bit more advanced; on the other hand, it is the easiest one to explain to an outsider / layperson. A simple analogy is teaching your pet (I am very intentionally trying to steer clear of the dogs vs cats debate) to perform tricks: you provide a treat for a trick well done, and refuse it otherwise.

Reinforcement learning has been a late comer to the competition party on Kaggle, but the situation has changed in the last few years with the introduction of simulation competitions. In this chapter, we will describe this new and exciting part of the Kaggle universe. So far – as of the time of this writing – there have been four “Featured” competitions and two “playground” ones; this admittedly short list allows us to give a broad overview. We begin with a convenient starting point.

# Start with the basics

If reinforcement learning is a completely new concept for you, it is probably a good idea to get some basic understanding. A very good way to start on the RL adventure is the Kaggle Learning course dedicated to this very topic in the context of Game AI:

<https://www.kaggle.com/learn/intro-to-game-ai-and-reinforcement-learning>

The course introduces basic concepts like agents and policies, providing also a (crash) introduction to deep reinforcement learning. All the examples use the data from a Playground competition ConnectX – the objective is to train an agent capable of playing a game of connecting checkers in a line.

<https://www.kaggle.com/c/connectx/overview>

An important aspect is the environment: due to the very nature of the problem, your solution needs to exhibit more dynamic characteristics than just submitting a set of numbers (as would be the case for ‘regular’ supervised learning contests). A very informative and detailed description of the environment used in the simulation competitions can be found at:

<https://github.com/Kaggle/kaggle-environments/blob/master/README.md>

Below we demonstrate how this simple problem can be approached using deep reinforcement learning. The code is adapted from the fantastic solution introduced by Tom van de Wiele <https://www.kaggle.com/tvdwiele> – if you are new to the concept of using AI for board games, Tom’s presentation is a resource worth exploring <https://tinyurl.com/36rdv5sa> .

In Connect 4, your objective is to get 4 of your checkers in a row – horizontally, vertically, or diagonally – on the game board before your opponent. Players take turns dropping their checkers into one of the columns at the top of the board. This means each move may be trying to either win for you, or trying to stop your opponent from winning.

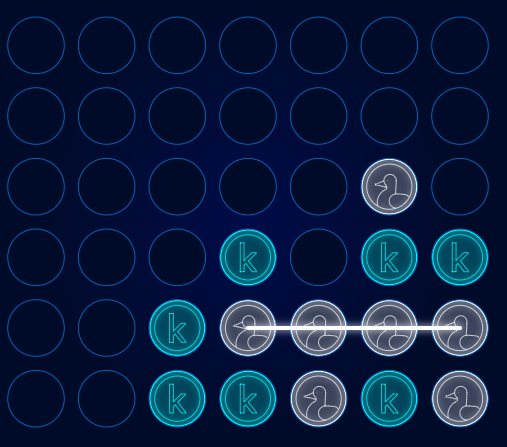


Figure 1: ConnectX board

We begin by installing the necessary pre-requisites:

!pip install 'kaggle-environments>=0.1.6'

!pip install 'recordtype'

And importing the required libraries:

#@title Imports

import numpy as np

import pandas as pd

%tensorflow\_version 1.x # See https://colab.research.google.com/notebooks/tensorflow\_version.ipynb

import tensorflow as tf

import time

from IPython.display import clear\_output

from IPython.display import display

from IPython.display import Image

from kaggle\_environments import evaluate as evaluate\_game

from kaggle\_environments import make as make\_game

from kaggle\_environments import utils as utils\_game

from random import choice

from recordtype import recordtype

from tensorflow.keras import backend as K

from tensorflow.keras.layers import Activation

from tensorflow.keras.layers import Conv2D

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import Flatten

from tensorflow.keras.layers import Input

from tensorflow.keras.models import Model

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.utils import plot\_model

We define some utilities to be used later in the model development

ExperienceStep = recordtype('ExperienceStep', [

'game\_id',

'current\_network\_input',

'action',

'next\_network\_input',

'last\_episode\_action',

'episode\_reward',

])

# Collect user input - Modified from https://www.kaggle.com/marcovasquez/how-to-play-with-computer-and-check-winner

def get\_input(user, observation, configuration):

ncol = configuration.columns

time.sleep(0.1)

input1 = 'Input from player {}: '.format(your\_name)

while True:

try:

print('Enter Value from 1 to 7')

raw\_input = input(input1)

user\_input = int(raw\_input)

except ValueError:

try:

print('Invalid input:', user\_input)

continue

except UnboundLocalError:

user\_input = -1

if raw\_input == 'q':

break

continue

np\_board = obs\_to\_board(observation, configuration)

valid\_actions = np.where(np\_board[0] == 0)[0]

if user\_input <= 0 or user\_input > ncol or (

user\_input-1) not in valid\_actions:

print('invalid input:', user\_input)

print('Valid actions: {}'.format(valid\_actions+1))

else:

return user\_input-1

# Convert the 1D observation list to a 2D numpy array

def obs\_to\_board(observation, configuration):

return np.array(observation.board).reshape(

configuration.rows, configuration.columns)

def check\_winner(observation):

'''

Source: https://www.kaggle.com/marcovasquez/how-to-play-with-computer-and-check-winner

This function returns the value of the winner.

INPUT: observation

OUTPUT: 1 for user Winner or 2 for Computer Winner

'''

line1 = observation.board[0:7] # bottom row

line2 = observation.board[7:14]

line3 = observation.board[14:21]

line4 = observation.board[21:28]

line5 = observation.board[28:35]

line6 = observation.board[35:42]

board = [line1, line2, line3, line4, line5, line6]

# Check rows for winner

for row in range(6):

for col in range(4):

if (board[row][col] == board[row][col + 1] == board[row][col + 2] == (

board[row][col + 3])) and (board[row][col] != 0):

return board[row][col] #Return Number that match row

# Check columns for winner

for col in range(7):

for row in range(3):

if (board[row][col] == board[row + 1][col] == board[row + 2][col] == (

board[row + 3][col])) and (board[row][col] != 0):

return board[row][col] #Return Number that match column

# Check diagonal (top-left to bottom-right) for winner

for row in range(3):

for col in range(4):

if (board[row][col] == board[row + 1][col + 1] == board[

row + 2][col + 2] ==\

board[row + 3][col + 3]) and (board[row][col] != 0):

return board[row][col] #Return Number that match diagonal

# Check diagonal (bottom-left to top-right) for winner

for row in range(5, 2, -1):

for col in range(4):

if (board[row][col] == board[row - 1][col + 1] == (

board[row - 2][col + 2]) == board[row - 3][col + 3]) and (

board[row][col] != 0):

return board[row][col] #Return Number that match diagonal

# No winner: return None

return None

# Custom class to reuse data of subsequent interations with the environment

# FIFO buffer. Experience buffer (also referred to as the replay buffer).

class ExperienceBuffer:

def \_\_init\_\_(self, buffer\_size):

self.buffer\_size = buffer\_size

self.episode\_offset = 0

self.data = []

self.episode\_ids = np.array([])

def add(self, data):

episode\_ids = np.array([d.game\_id for d in data])

num\_episodes = episode\_ids[-1] + 1

if num\_episodes > self.buffer\_size:

# Keep most recent experience of the experience batch

data = data[

np.where(episode\_ids == (num\_episodes-self.buffer\_size))[0][0]:]

self.data = data

self.episode\_ids = episode\_ids

self.episode\_offset = 0

return

episode\_ids = episode\_ids + self.episode\_offset

self.data = data + self.data

self.episode\_ids = np.concatenate([episode\_ids, self.episode\_ids])

unique\_episode\_ids = pd.unique(self.episode\_ids)

if unique\_episode\_ids.size > self.buffer\_size:

cutoff\_index = np.where(self.episode\_ids == unique\_episode\_ids[

self.buffer\_size])[0][0]

self.data = self.data[:cutoff\_index]

self.episode\_ids = self.episode\_ids[:cutoff\_index]

self.episode\_offset += num\_episodes

def get\_all\_data(self):

return self.data

def size(self):

return len(self.data)

def num\_episodes(self):

return np.unique(self.episode\_ids).size

# Masked mse loss - values equal to mask\_val are ignored in the loss

def masked\_mse(y, p, mask\_val):

mask = K.cast(K.not\_equal(y, mask\_val), K.floatx())

if tf.\_\_version\_\_[0] == '2':

masked\_loss = tf.losses.mse(y\*mask, p\*mask)

else:

mask = K.cast(mask, 'float32')

masked\_loss = K.mean(tf.math.square(p\*mask - y\*mask), axis=-1)

# masked\_loss = tf.compat.v1.losses.mean\_squared\_error(y\*mask, p\*mask)

return masked\_loss

# Make the masked mse loss

def make\_masked\_mse(nan\_coding\_value):

def loss(y, p):

return masked\_mse(y, p, mask\_val=nan\_coding\_value)

return loss

Our approach to the problem uses Q-learning to train an agent: it is a model-free (as in: we don’t require a model for interaction with the environment) algorithm well suited to handling stochastic rewards and transitions between states. For a crash introduction to the topic, Wikipedia entry is an excellent starting point: <https://en.wikipedia.org/wiki/Q-learning>

In our context, the Q values represent the expected terminal reward resulting from being in board state *s* and taking action *a*. The core of our training approach is the following:

1. Play a fixed number of games using self-play (<https://www.arxiv-vanity.com/papers/2002.04017/>) and store the results in the experience buffer
2. Update the Q network
3. Evaluate against (random) agent to check progress

We can define possible choices for the Q network architecture

#@title Define possible network architectures of the Q-network

# Basic MLP Connect 4 network

def mlp\_connect4(config):

mlp\_layers = config['mlp\_layers']

inputs = Input((6, 7, 3), name='encoded\_board')

x = inputs

# Flatten the input

x = Flatten()(x)

# MLP layers - sigmoid activation on the final layer

for i, layer\_size in enumerate(mlp\_layers):

x = Dense(layer\_size, activation='linear')(x)

if i < (len(mlp\_layers)-1):

x = Activation('relu')(x)

else:

x = Activation('sigmoid', name='Q-values')(x)

outputs = x

return (inputs, outputs)

# Basic convolutional Connect 4 network

def convnet\_connect4(config):

inputs = Input((6, 7, 3), name='encoded\_board')

x = inputs

# Convolutional layers

conv\_outputs = []

for i, (filters, kernel, strides) in enumerate(

config['filters\_kernels\_strides']):

x = Conv2D(filters=filters, kernel\_size=kernel, strides=strides,

padding='same', activation='linear')(x)

x = Activation('relu')(x)

# Flatten the activations

x = Flatten()(x)

# MLP layers - sigmoid activation on the final layer

for i, layer\_size in enumerate(config['mlp\_layers']):

if i < (len(config['mlp\_layers'])-1):

# Non final fully connected layers

x = Dense(layer\_size, activation='linear')(x)

x = Activation('relu')(x)

else:

# Head of the network

x = Dense(layer\_size, activation='linear')(x)

outputs = Activation('sigmoid', name='Q-values')(x)

return (inputs, outputs)

Next we define the mapping between raw observations (state of the board) and the network (agent) input, as well as the ones in the opposite direction.

# Convert the raw observation to the network input

def obs\_to\_network\_input(observation, configuration, player\_id):

board = np.array(observation.board).reshape(

configuration.rows, configuration.columns)

# One hot encoding of the inputs (empty, player 1, player 2)

obs\_input = (np.arange(3) == board[..., None]).astype(float)

# Swap player 1 and player 2 positions? The network always assumes that 'my'

# stones (player\_id) come first

if player\_id == 1:

tmp = obs\_input[:, :, 1].copy()

obs\_input[:, :, 1] = obs\_input[:, :, 2]

obs\_input[:, :, 2] = tmp

return obs\_input

# Predict network outputs where the number of inputs can be large. Use batching

# when there are more inputs than the max\_batch\_size

def my\_keras\_predict(model, inputs, max\_batch\_size=10000):

num\_inputs = inputs.shape[0]

num\_batches = int(np.ceil(num\_inputs/max\_batch\_size))

outputs = []

for i in range(num\_batches):

end\_id = num\_inputs if i == (num\_batches-1) else (i+1)\*max\_batch\_size

batch\_inputs = inputs[i\*max\_batch\_size:end\_id]

if tf.\_\_version\_\_[0] == '2':

batch\_outputs = model(batch\_inputs)

batch\_outputs = batch\_outputs.numpy()

else:

batch\_outputs = model.predict(batch\_inputs)

outputs.append(batch\_outputs)

return np.concatenate([o for o in outputs])

def select\_action\_from\_q(q\_values, valid\_actions, epsilon\_greedy\_parameter):

# Select the best valid or a valid exploratory action using epsilon-greedy

best\_q = q\_values[valid\_actions].max()

best\_a\_ids = np.where(q\_values[valid\_actions] == best\_q)[0]

best\_a = valid\_actions[np.random.choice(best\_a\_ids)]

exploratory\_a = np.random.choice(valid\_actions)

explore = np.random.uniform() < epsilon\_greedy\_parameter

action = exploratory\_a if explore else best\_a

return action

def get\_agent\_q\_and\_a(agent, board, epsilon\_greedy\_parameter):

# Obtain the Q-values

q\_values = my\_keras\_predict(agent, np.expand\_dims(board, 0))[0]

# Select an action from the Q-values

valid\_actions = np.where(board[0, :, 0] == 1)[0]

action = select\_action\_from\_q(q\_values, valid\_actions,

epsilon\_greedy\_parameter)

return q\_values, action

Next, we specify our self-play mechanics. The concept existed since the 1960s, with the 1992 experiment of Tesauro (a cluster of computers was used to train a neural network to play backgammon that way; the “agent” trained in this manner defeated then world champion and exhibited strategies recognized as superior by human experts). Another important benchmark was the AlphaGo Zero result, which defeated all humans.

An interesting aspect of the approach is that it can also work in very simple environments – which ConnectX undoubtedly is. The most important aspect is that self-play effective allows you to convert compute into data, which is extremely useful given how much it costs to obtain either.

def self\_play(agent, num\_games, verbose, epsilon\_greedy\_parameter):

experience = []

for game\_id in range(num\_games):

this\_game\_data = []

env = make\_game('connectx')

env.reset()

episode\_step = 0

# Take actions until the game is terminated

while not env.done:

if env.state[0].status == 'ACTIVE':

player\_id = 0

elif env.state[1].status == 'ACTIVE':

player\_id = 1

# Obtain the Q-values and selected action for the current state

current\_network\_input = obs\_to\_network\_input(

env.state[player\_id].observation, env.configuration, player\_id)

q\_values, action = get\_agent\_q\_and\_a(

agent, current\_network\_input, epsilon\_greedy\_parameter)

if episode\_step == 0 and game\_id == 0:

print("Start move Q-values: {}".format(np.around(q\_values, 3)))

env.step([int(action) if i == player\_id else None for i in [0, 1]])

# Store the state transition data - swap the player id!

next\_network\_input = obs\_to\_network\_input(

env.state[player\_id].observation, env.configuration, 1-player\_id)

this\_game\_data.append(ExperienceStep(

game\_id,

current\_network\_input,

action,

next\_network\_input,

False, # Last episode action, overwritten at the end of the episode

np.nan, # Terminal reward, overwritten at the end of the episode

))

episode\_step += 1

# Overwrite the terminal reward for all actions

first\_terminal\_reward = env.state[0].reward

for i in range(len(this\_game\_data)):

if i % 2 == 0:

this\_game\_data[i].episode\_reward = first\_terminal\_reward

else:

this\_game\_data[i].episode\_reward = 1-first\_terminal\_reward

# Update statistics which can not be computed before the episode is over.

this\_game\_data[-1].last\_episode\_action = True # Last episode action

experience.extend(this\_game\_data)

if verbose and game\_id % 10 == 9:

print('Completed playing game {} of {}'.format(game\_id+1, num\_games))

return experience

We can also define a function for one step computation of the target – for the theoretical analysis of Q-learning (and to get more familiarity with the concepts discussed here), the reader is referred to the article by Fan et al. “A theoretical analysis of deep Q-learning” <https://arxiv.org/pdf/1901.00137.pdf> .

#@title One step minimax Q-learning target computation - Source of name: https://arxiv.org/pdf/1901.00137.pdf. To be more precise, we actually use negamax Q-learning since we rely on the property that the game is a two-player zero sum game.

def one\_step\_minimax\_q\_targets(next\_q\_vals, experience):

next\_q\_minimax\_star = (1-next\_q\_vals.max(1)).tolist() # Negamax - 2 player 0-sum

terminal\_rewards = [e.episode\_reward for e in experience]

last\_episode\_actions = [e.last\_episode\_action for e in experience]

target\_qs = np.array([t if l else n for(t, l, n) in zip(

terminal\_rewards, last\_episode\_actions, next\_q\_minimax\_star)])

return target\_qs

# N-step minimax Q-learning target computation

def minimax\_q\_n\_step\_targets(this\_q\_vals, next\_q\_vals, experience,

return\_steps\_trace):

# Collect generic experience values of interest

return\_steps, lambda\_ = return\_steps\_trace

actions = [e.action for e in experience]

terminal\_actions = np.concatenate(

[np.array([e.last\_episode\_action for e in experience]),

np.zeros((return\_steps), dtype=np.bool)])

terminal\_rewards = 1-np.array([e.episode\_reward for e in experience])

num\_experience\_steps = len(experience)

# Determine if the actions are exploratory or greedy

greedy\_actions = np.zeros((num\_experience\_steps+return\_steps),

dtype=np.bool)

for i in range(num\_experience\_steps):

valid\_actions = np.where(experience[i].current\_network\_input[:, :, 0].sum(

0) > 0)[0]

best\_valid\_q = this\_q\_vals[i][valid\_actions].max()

greedy\_actions[i] = best\_valid\_q == this\_q\_vals[i, actions[i]]

# Extend and overwrite next q vals to include the true rewards

next\_q\_vals = np.concatenate([next\_q\_vals, -999\*np.ones((

return\_steps, next\_q\_vals.shape[1]))])

next\_q\_vals[terminal\_actions] = np.tile(

np.expand\_dims(terminal\_rewards[terminal\_actions[

:num\_experience\_steps]], -1), [1, next\_q\_vals.shape[1]])

# Consider returns, up to 'return\_steps' into the future. The trace is cut

# at episode boundaries and before considering a non-exploratory action

consider\_targets = np.ones((num\_experience\_steps), dtype=np.bool)

target\_lambda\_sums = np.zeros((num\_experience\_steps))

target\_weighted\_sums = np.zeros((num\_experience\_steps))

trace\_multiplier = 1

for i in range(return\_steps):

best\_qs = next\_q\_vals[i:(i+num\_experience\_steps)].max(-1)

if i % 2 == 0:

best\_qs = 1-best\_qs

target\_weighted\_sums[consider\_targets] += trace\_multiplier\*best\_qs[

consider\_targets]

target\_lambda\_sums[consider\_targets] += trace\_multiplier

# Don't consider any further targets if this was the episode terminal

# action

consider\_targets = np.logical\_and(

consider\_targets, np.logical\_not(

terminal\_actions[i:(i+num\_experience\_steps)]))

# Don't consider the target if the next action is exploratory

# Extending greedy actions with return\_steps False values makes sure

# we don't consider N step returns where there is no data

consider\_targets = np.logical\_and(

consider\_targets, greedy\_actions[(i+1):(i+1+num\_experience\_steps)])

trace\_multiplier \*= lambda\_

targets = target\_weighted\_sums/target\_lambda\_sums

return targets

# Get the q-learning observations and targets

def minimax\_q\_learning(model, experience, nan\_coding\_value, return\_steps\_trace):

# Evaluate the Q-values of the current and next state for all observations

num\_actions = experience[0].current\_network\_input.shape[1]

num\_steps = len(experience)

this\_states = np.stack([e.current\_network\_input for e in experience])

next\_states = np.stack([e.next\_network\_input for e in experience])

this\_q\_vals = my\_keras\_predict(model, this\_states)

next\_q\_vals = my\_keras\_predict(model, next\_states)

# Filter out next Q-values where the next action is not valid. Filtered out

# since every target computation performs a max operation.

# This was an unintended bug in the original version of the notebook!

next\_q\_vals[next\_states[:, 0, :, 0] == 0] = -1

# Compute the target Q-values

num\_return\_steps = return\_steps\_trace[0]

if num\_return\_steps == 1:

target\_qs = one\_step\_minimax\_q\_targets(next\_q\_vals, experience)

else:

target\_qs = minimax\_q\_n\_step\_targets(this\_q\_vals, next\_q\_vals, experience,

return\_steps\_trace)

# Don't learn about non acted Q-values

all\_target\_qs = nan\_coding\_value\*np.ones([num\_steps, num\_actions])

# Set the targets for the actions that were selected

actions = [e.action for e in experience]

all\_target\_qs[np.arange(num\_steps), actions] = target\_qs

return this\_states, all\_target\_qs

# Update the Q-network by minimizing the difference with the target q-values

def update\_agent(experience, agent, config):

nan\_coding\_value = config['nan\_coding\_value']

return\_steps\_trace = config['return\_steps\_trace']

x\_train, y\_train = minimax\_q\_learning(

agent, experience, nan\_coding\_value, return\_steps\_trace)

adam = Adam(lr=config['learning\_rate'])

agent.compile(optimizer=adam, loss=make\_masked\_mse(nan\_coding\_value))

agent.fit(

x\_train,

y\_train,

batch\_size=config['batch\_size'],

epochs=config['num\_epochs'],

verbose=config['verbose\_fit']

)

As mentioned at the start of this section, we will evaluate the performance of our learned agent against a greedy one to monitor performance improvements as a result of learning

#@title Evaluate a greedy agent against the random agent

def eval\_greedy\_versus\_random(agent, num\_eval\_games):

rewards = []

env = make\_game('connectx', debug=False)

my\_agent\_starts = False

for i in range(num\_eval\_games):

my\_agent\_starts = not my\_agent\_starts # Alternate start turns

player\_id = int(not my\_agent\_starts)

episode\_step = 0

while not env.done:

if env.state[0].status == 'ACTIVE':

active = 0

elif env.state[1].status == 'ACTIVE':

active = 1

current\_network\_input = obs\_to\_network\_input(

env.state[active].observation, env.configuration, player\_id)

if active == player\_id:

# Take the greedy action of the agent

\_, action = get\_agent\_q\_and\_a(

agent, current\_network\_input, epsilon\_greedy\_parameter=0)

else:

# Take a random valid action

valid\_actions = np.where(current\_network\_input[0, :, 0] == 1)[0]

action = np.random.choice(valid\_actions)

env.step([int(action) if i == active else None for i in [0, 1]])

episode\_step += 1

rewards.append(env.state[player\_id].reward)

env.reset()

return np.array(rewards).mean()

Finally, we can execute the main logic

# The network weights are saved in this session which is bound to be brittle - store to and load from file when expanding on this logic!

reset\_agent\_weights = False #@param ["False", "True"] {type:"raw"}

reset\_experience\_buffer = False #@param ["False", "True"] {type:"raw"}

plot\_agent\_architecture = True #@param ["False", "True"] {type:"raw"}

config = {

'model': [mlp\_connect4, convnet\_connect4][1],

'model\_config': {

'filters\_kernels\_strides': [

(32, 3, 1), (32, 3, 2), (32, 3, 2)],

'mlp\_layers': [64, 7],

},

'num\_self\_play\_games\_per\_iteration': 200,

'verbose\_self\_play': False, # Show self-play game id progress?

'epsilon\_greedy\_parameter': 0.2,

'max\_experience\_buffer\_games': 2000,

'num\_learning\_updates\_per\_iteration': 2,

'learning\_config': {

# Return-steps: N in N-step returns and lambda (only for N > 1)

'return\_steps\_trace': (20, 0.9),

'nan\_coding\_value': -999,

'learning\_rate': 5e-4,

'batch\_size': 32,

'num\_epochs': 2,

'verbose\_fit': False, # Show learning progress?

},

'num\_evaluation\_games': 50,

}

# Reset the agent weights when the option is selected or when the agent is not

# defined yet - This initializes the network.

if 'my\_trained\_agent' not in locals() or reset\_agent\_weights:

inputs, outputs = config['model'](config['model\_config'])

my\_trained\_agent = Model(inputs=inputs, outputs=outputs)

if plot\_agent\_architecture:

plot\_model(my\_trained\_agent, show\_shapes=True, show\_layer\_names=True,

to\_file='model.png')

display(Image(retina=True, filename='model.png'))

# Clear the experience buffer when the option is selected or when the buffer is

# not defined yet.

if 'experience\_buffer' not in locals() or reset\_experience\_buffer:

experience\_buffer = ExperienceBuffer(config['max\_experience\_buffer\_games'])

while True:

# 1) Add self-play experience to the experience replay buffer.

experience = self\_play(my\_trained\_agent,

config['num\_self\_play\_games\_per\_iteration'],

config['verbose\_self\_play'],

config['epsilon\_greedy\_parameter'],

)

experience\_buffer.add(experience)

# 2) Update the Q network using the experience from the experience buffer

print("Experience buffer size: {} transitions ({} episodes)".format(

experience\_buffer.size(), experience\_buffer.num\_episodes()))

for \_ in range(config['num\_learning\_updates\_per\_iteration']):

update\_agent(experience\_buffer.get\_all\_data(), my\_trained\_agent,

config['learning\_config'])

# 3) Evaluate the greedy trained agent against a random agent to see if we are

# making progress

mean\_eval\_reward = eval\_greedy\_versus\_random(

my\_trained\_agent, config['num\_evaluation\_games'])

print("Mean reward against random agent in {} games: {}".format(

config['num\_evaluation\_games'], mean\_eval\_reward))

print("")

Executing the main logic should yield output similar to this:

Start move Q-values: [0.503 0.536 0.502 0.518 0.47 0.52 0.52 ]

Experience buffer size: 4870 transitions (200 episodes)

Mean reward against random agent in 50 games: 0.64

Start move Q-values: [0.526 0.492 0.502 0.524 0.487 0.501 0.52 ]

Experience buffer size: 8159 transitions (400 episodes)

Mean reward against random agent in 50 games: 0.76

Start move Q-values: [0.48 0.487 0.491 0.495 0.478 0.479 0.486]

Experience buffer size: 11186 transitions (600 episodes)

Mean reward against random agent in 50 games: 0.82

Start move Q-values: [0.498 0.518 0.512 0.505 0.486 0.511 0.49 ]

Experience buffer size: 14574 transitions (800 episodes)

Mean reward against random agent in 50 games: 0.84

Start move Q-values: [0.495 0.504 0.491 0.516 0.502 0.478 0.481]

Experience buffer size: 18693 transitions (1000 episodes)

Mean reward against random agent in 50 games: 0.96

Start move Q-values: [0.479 0.462 0.483 0.524 0.494 0.454 0.459]

Experience buffer size: 22901 transitions (1200 episodes)

Mean reward against random agent in 50 games: 0.92

Start move Q-values: [0.471 0.452 0.469 0.532 0.468 0.425 0.427]

Experience buffer size: 27021 transitions (1400 episodes)

Mean reward against random agent in 50 games: 1.0

Start move Q-values: [0.456 0.445 0.485 0.539 0.482 0.399 0.424]

Experience buffer size: 29927 transitions (1600 episodes)

Self-play is obviously the more interesting option

play\_against\_trained = True #@param ["False", "True"] {type:"raw"}

plot\_resolution = 400 #@param {type:"slider", min:200, max:500, step:1}

print(my\_trained\_agent.predict(np.ones((1, 6, 7, 3))))

# Here we define an agent that picks the best action according to the trained

# Q-value network. There is a lot of code duplication with play\_against\_agent

# becaus I ran into issues making the trained agent accessible to the Graph.

# Always play as first position against the opposing agent.

# Adapted from https://www.kaggle.com/marcovasquez/how-to-play-with-computer-and-check-winner

def play\_against\_trained\_agent(agent):

env = make\_game("connectx", debug=False, configuration={"timeout": 10})

human\_starts = np.random.uniform() > 0.5

human\_id = int(not human\_starts)

human\_move = human\_starts

while not env.done:

if human\_move:

clear\_output(wait=True) # Comment if you want to keep track of every action

print("{}'s color: {}".format(

your\_name, "Blue" if human\_starts else "Grey"))

env.render(mode="ipython", width=plot\_resolution, height=plot\_resolution,

header=False, controls=False)

observation = env.state[human\_id].observation

# Plot the opponent q-values when they are available

if (np.array(observation.board) == 1).sum() > 0:

print("Previous step agent Q-values: {}".format(np.round(q\_values, 2)))

action = get\_input(your\_name, observation, env.configuration)

if action is None:

print("Exiting game after pressing q")

return

env.step([int(action), None] if human\_starts else [None, int(action)])

else:

current\_network\_input = obs\_to\_network\_input(

env.state[1-human\_id].observation, env.configuration,

player\_id=1-human\_id)

# Take the greedy action of the agent

q\_values, action = get\_agent\_q\_and\_a(

agent, current\_network\_input, epsilon\_greedy\_parameter=0)

env.step([None, int(action)] if human\_starts else [int(action), None])

human\_move = not human\_move

observation = env.state[human\_id].observation

if (check\_winner(observation) == (human\_id+1)):

print("You Won, Amazing! \nGAME OVER")

elif (check\_winner(observation) == (2-human\_id)):

print("The agent Won! \nGAME OVER")

if (check\_winner(observation) is None):

print("That is a draw between you and the agent")

env.render(mode="ipython", width=plot\_resolution, height=plot\_resolution,

header=False, controls=False)

if play\_against\_trained:

play\_against\_trained\_agent(my\_trained\_agent)

The code above shows how to setup a solution from scratch for a relatively simple problem (there is a reason why ConnectX is a “playground” and not a “featured” competition). Interestingly, the simple problem can be handled with (almost) SOTA methods like AlphaZero <https://www.kaggle.com/connect4alphazero/alphazero-baseline-connectx> .

With the introduction behind us, you should be ready to dive into the more realistic (or in any case, not toy example based) contests.

# Rock-paper-scissors

It is no coincidence that several problems in simulation competitions refer to playing games: at varying levels of complexity, games offer an environment with clearly defined rules, naturally lending itself to the agent – action – reward framework. With the possible exception of Tic-Tac-Toe, connecting checkers is one of the simplest examples of a competitive game. Moving up the difficulty ladder (of games), let’s have a look at rock-paper-scissors and how a Kaggle contest centered around this game could be approached:

<https://www.kaggle.com/c/rock-paper-scissors/code>

The idea of the competition was the extension from the basic rock-paper-scissors (knows as roshambo in some parts of the world): instead of the usual “best of 3” score, we use “best of 1000”. We describe two possible approaches to the problem: one rooted in the game theoretic approach, and the other more focused on the algorithmic side.

We begin with the Nash equilibrium: Wikipedia gives the definition as the solution to a non-cooperative game involving two or more players; each player is assumed to know the equilibrium strategies of the others, and no player can obtain an advantage by changing only their own strategy. An excellent introduction to RPS in game theoretic framework can be found at: <https://www.youtube.com/watch?v=-1GDMXoMdaY>

The matrix of outcomes of the RPS game can be summarized as follows:

Obraz zawierający tekst, tablica wyników

Opis wygenerowany automatycznie

If we played each action with equal probability 1/3 then the opponent must do the same; otherwise playing Rock all the time, they will: ties against rock, lose against Paper and win against Scissors – each with probability 1/3 (or one third of the time). Expected reward in this case is 0, in which case we can change strategy to Paper and win all the time. The graphs below demonstrate that the same logic applies to opponent strategy being Paper vs Scissors and Scissors vs Rock.

Obraz zawierający tekst, tablica wyników

Opis wygenerowany automatycznie

Obraz zawierający tekst, tablica wyników

Opis wygenerowany automatycznie

Obraz zawierający tekst, tablica wyników

Opis wygenerowany automatycznie

The remaining option in order to be in equilibrium is that both players need to play a random strategy, then there is no point in changing their strategy - which is the Nash equilibrium. Our agent implementing this simple strategy is

%%writefile submission.py

import random

def nash\_equilibrium\_agent(observation, configuration):

return random.randint(0, 2)

The magic at the start (writing from a notebook directly to a file) was necessary to satisfy the submission constraints of this particular competition.

How does our Nash agent perform against others? We can find out by evaluating the performance

!pip install -q -U kaggle\_environments

from kaggle\_environments import make

As of the time of this writing, there is an error popping up after this import (failure to load a module named ‘gfootball’) – an official advice from Kaggle is to ignore it. In practice, it does not seem to have any impact on executing the code. We start by creating RPS environment and setting the limit to 1000 episodes per simulation:

env = make(

"rps",

configuration={"episodeSteps": 1000}

)

We will make use of a kernel created in this competition that implemented numerous agents based on deterministic heuristics

<https://www.kaggle.com/ilialar/multi-armed-bandit-vs-deterministic-agents>

and import the code for the agents we compete against from there.

agent\_copy\_opponent\_path = "../input/rock-paper-scissors-agents-comparison/copy\_opponent.py"

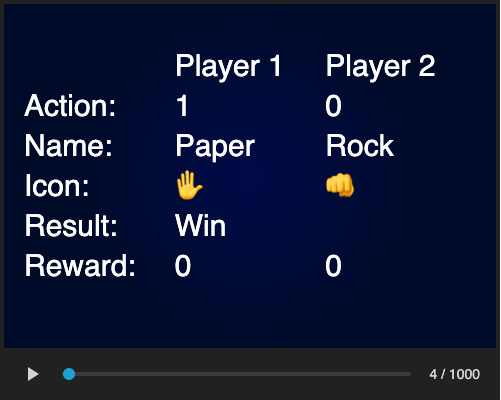
*# nash\_equilibrium\_agent vs copy\_opponent\_agent*

env.run(

["submission.py", agent\_copy\_opponent\_path]

)

env.render(mode="ipython", width=500, height=400)



In supervised learning – both classification and regression – it is frequently useful to start approaching any problem with a simple benchmark, usually a linear model. Even though not SOTA, it can provide useful expectation and a measure of performance. In Reinforcement Learning, an approach worth trying in such capacity is the Multi Armed Bandit (MAB). Below, we demonstrate briefly how it can be used to approach the Rock-Paper-Scissors competition – the approach is based on the code from Ilia Larchenko <https://www.kaggle.com/ilialar> . For a good explanation of methods for approaching the general MAB problem, the reader is referred to <https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html>

import pandas as pd

import numpy as np

import json

*# base class for all agents, random agent*

class **agent**():

def initial\_step(self):

return np.random.randint(3)

def history\_step(self, history):

return np.random.randint(3)

def step(self, history):

if len(history) == 0:

return int(self.initial\_step())

else:

return int(self.history\_step(history))

A MAB-based agent is defined as

def multi\_armed\_bandit\_agent (observation, configuration):

*# bandits' params*

step\_size = 3 *# how much we increase a and b*

decay\_rate = 1.05 *# how much do we decay old historical data*

global history, bandit\_state

def log\_step(step = None, history = None, agent = None, competitorStep = None, file = 'history.csv'):

if step **is** None:

step = np.random.randint(3)

if history **is** None:

history = []

history.append({'step': step, 'competitorStep': competitorStep, 'agent': agent})

if file **is** **not** None:

pd.DataFrame(history).to\_csv(file, index = False)

return step

def update\_competitor\_step(history, competitorStep):

history[-1]['competitorStep'] = int(competitorStep)

return history

*# load history*

if observation.step == 0:

pass

else:

history = update\_competitor\_step(history, observation.lastOpponentAction)

*# updating bandit\_state using the result of the previous step*

*# we can update all states even those that were not used*

for name, agent **in** agents.items():

agent\_step = agent.step(history[:-1])

bandit\_state[name][1] = (bandit\_state[name][1] - 1) / decay\_rate + 1

bandit\_state[name][0] = (bandit\_state[name][0] - 1) / decay\_rate + 1

if (history[-1]['competitorStep'] - agent\_step) % 3 == 1:

bandit\_state[name][1] += step\_size

elif (history[-1]['competitorStep'] - agent\_step) % 3 == 2:

bandit\_state[name][0] += step\_size

else:

bandit\_state[name][0] += step\_size/2

bandit\_state[name][1] += step\_size/2

*# we can use it for analysis later*

with open('bandit.json', 'w') as outfile:

json.dump(bandit\_state, outfile)

*# generate random number from Beta distribution for each agent and select the most lucky one*

best\_proba = -1

best\_agent = None

for k **in** bandit\_state.keys():

proba = np.random.beta(bandit\_state[k][0],bandit\_state[k][1])

if proba > best\_proba:

best\_proba = proba

best\_agent = k

step = agents[best\_agent].step(history)

return log\_step(step, history, best\_agent)

We can evaluate its performance against a simple agent that copies opponent move:

def copy\_opponent\_agent(observation, configuration):

if observation.step > 0:

return observation.lastOpponentAction

else:

return 0

env = make("rps", debug=True)

env.reset()

env.run(["submission.py", "copy\_opponent.py"])

env.render(mode="ipython", width=800, height=700)

Obraz zawierający tekst, ekran, zrzut ekranu

Opis wygenerowany automatycznie

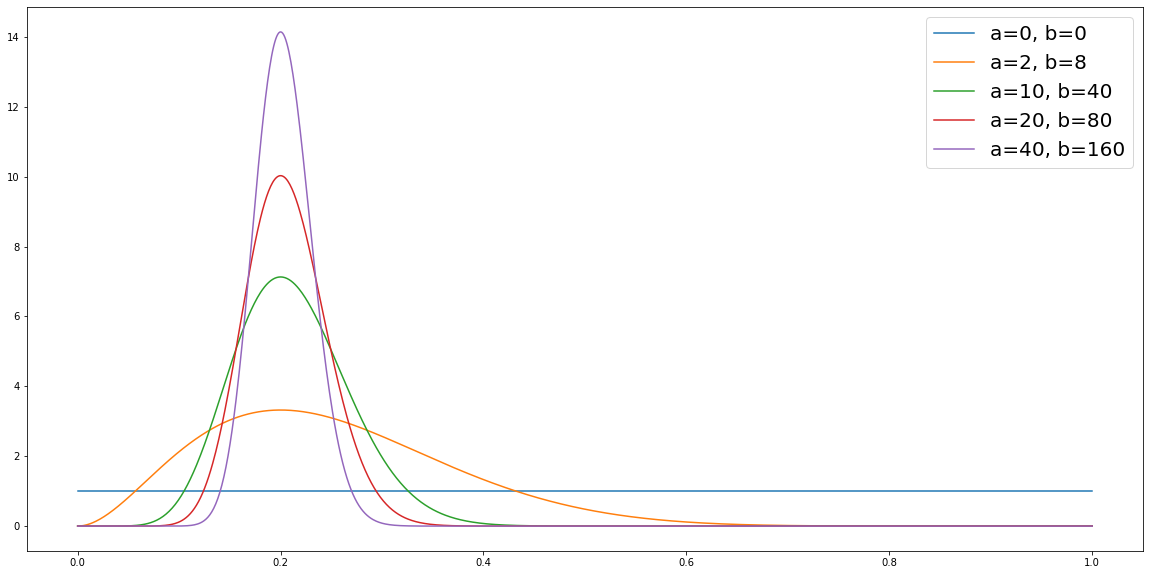
# Santa competition 2020

Over the last few years, a sort of tradition has emerged on Kaggle: in early December, there is a Santa-themed competition. The actual algorithmic side varies from year, but for our purposes the 2020 is an interesting case:

<https://www.kaggle.com/c/santa-2020>

The setup was a classical multi-armed bandit (trying to maximize reward by repeated action on a vending machine), but with two extras:   
1. Reward decay: in each step, probability of obtaining a reward from a machine decreases by 3pct   
2. Competition: we are constrained not only by time (limited number of attempts) but also by another player attempting to achieve the same objective.

The solution we demonstrate is adapted from <https://www.kaggle.com/ilialar/simple-multi-armed-bandit> . Our approach is based on successive updates to the distribution of reward: at each step, we generate a random number from Beta distribution with parameters (a+1, b+1) where a = total reward from this arm (number of wins) and b is the number of historical losses (prior distribution). When we need to make a decision, we select the arm with the highest generated number and use it to generate the next step – our posterior distribution becomes a prior for the next step.



As you can see, initially the distribution is flat (Beta(0,0) is uniform), but as we gather more information it concentrates the probability mass around the mode – which means there is less uncertainty and we are more confident about our judgement.

We can incorporate the competition specific reward decay by decreasing the a parameter every time the arm is used.

bandit\_state = None

total\_reward = 0

last\_step = None

def multi\_armed\_bandit\_agent (observation, configuration):

global history, history\_bandit

step = 1.0 *#you can regulate exploration / exploitation balacne using this param*

decay\_rate = 0.97 *# how much do we decay the win count after each call*

global bandit\_state,total\_reward,last\_step

if observation.step == 0:

*# initial bandit state*

bandit\_state = [[1,1] for i **in** range(configuration["banditCount"])]

else:

*# updating bandit\_state using the result of the previous step*

last\_reward = observation["reward"] - total\_reward

total\_reward = observation["reward"]

*# we need to understand who we are Player 1 or 2*

player = int(last\_step == observation.lastActions[1])

if last\_reward > 0:

bandit\_state[observation.lastActions[player]][0] += last\_reward \* step

else:

bandit\_state[observation.lastActions[player]][1] += step

bandit\_state[observation.lastActions[0]][0] = (bandit\_state[observation.lastActions[0]][0] - 1) \* decay\_rate + 1

bandit\_state[observation.lastActions[1]][0] = (bandit\_state[observation.lastActions[1]][0] - 1) \* decay\_rate + 1

*# generate random number from Beta distribution for each agent and select the most lucky one*

best\_proba = -1

best\_agent = None

for k **in** range(configuration["banditCount"]):

proba = np.random.beta(bandit\_state[k][0],bandit\_state[k][1])

if proba > best\_proba:

best\_proba = proba

best\_agent = k

last\_step = best\_agent

return best\_agent

Similar to the previous case, we are now ready to evaluate the performance of our agent

from kaggle\_environments import make

env = make("mab", debug=True)

env.reset()

env.run(["random\_agent.py", "submission.py"])

env.render(mode="ipython", width=800, height=700)

Obraz zawierający tekst, sprzęt elektroniczny, zrzut ekranu, wyświetlanie

Opis wygenerowany automatycznie

Alternatively, we can also introduce self-play – thus generating more data for improved training of our MAB-based agent:

Obraz zawierający tekst, zrzut ekranu, sprzęt elektroniczny

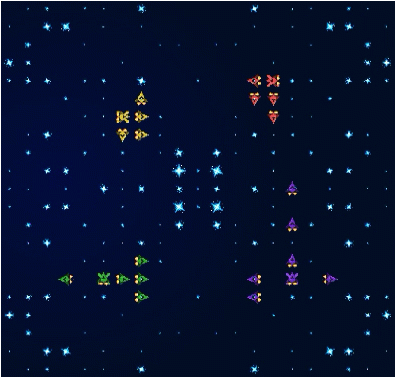
Opis wygenerowany automatycznie

In this section we demonstrate how a vintage Multi-Armed Bandit algorithm can be utilized in a simulation competition on Kaggle. We follow-up with a discussion of approaches based on other methods, in a diverse range of competitions.

# The name of the game

Beyond the relatively elementary games discussed above, simulation competitions involve more elaborate setups. In this section we will briefly discuss those.

“Halite” is a resource management game where you build and control a small armada of ships. Your algorithms determine their movements to collect halite, a luminous energy source. The most halite at the end of the match wins, but it's up to you to figure out how to make effective and efficient moves. You control your fleet, build new ships, create shipyards, and mine the regenerating halite on the game board.



Kaggle organized two competitions around the game: a playground <https://www.kaggle.com/c/halite-iv-playground-edition>

as well as a regular “featured” one: <https://www.kaggle.com/c/halite>. The classic reinforcement learning approach was less useful in this instance, since with an arbitrary number of units (ships / bases) and a dynamic opponent pool the problem of credit assignment was becoming intractable for people with access to “normal” level of computing resources. A description of the winning solution by Tom van de Wiele <https://www.kaggle.com/c/halite/discussion/183543> provides an excellent overview of the hybrid approach that proved successful in this instance.

Another competition organized a relatively sophisticated game was Lux AI: <https://www.kaggle.com/c/lux-ai-2021> . In this competition, the participants were tasked with designing agents to tackle a multi-variable optimization problem combining resource gathering and allocation – competing against other players. In addition, successful agents had to analyze the moves of their opponents and react accordingly. An interesting feature of this contest was the popularity of a ‘meta’ approach: imitation learning <https://paperswithcode.com/task/imitation-learning> A competitive implementation of this idea (as of the time of this writing) is given by user Ironbar <https://www.kaggle.com/c/lux-ai-2021/discussion/293911>

Finally, no discussion of simulation competitions in Kaggle would be complete without the Google Research Football with Manchester City competition <https://www.kaggle.com/c/google-football/overview>

The motivation behind this contest was for researchers to explore AI agents' ability to play in complex settings like football. The sport requires a balance of short-term control, learned concepts such as passing, and high-level strategy, which can be difficult to teach agents. A current environment exists to train and test agents, but other solutions may offer better results. Unlike some examples given above, this competition was dominated by Reinforcement Learning approaches:

* Team Raw Beast (3rd) followed a methodology inspired by AlphaStar <https://www.kaggle.com/c/google-football/discussion/200709>
* Salty Fish (2nd) utilized a form of self-play <https://www.kaggle.com/c/google-football/discussion/202977>
* The winners WeKick was deep learning based with creative feature engineering and reward structure adjustment <https://www.kaggle.com/c/google-football/discussion/202232>

In this chapter we discuss simulation competitions – a new type of contest, which is increasing in popularity. Compared to vision or NLP-centered ones, simulation contests involve a much broader range of methods (with somewhat higher mathematical content), which reflects the difference between supervised learning and reinforcement learning.