Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Факультет ИУ «Информатика и системы управления»
Кафедра ИУ3 «Информационные системы и телекоммуникации»

Лабораторная работа №1
по курсу «Мультиагентные интеллектуальные системы»
«Табулярное Q-обучение игрового агента Toytext»

Составил: В.Э. Большаков,
ассистент кафедры ИУ3

Москва, 2020

Содержание

FrozenLake-v0

https://gym.openai.com/envs/FrozenLake-v0/

Агент управляет движением персонажа в мире, состоящем из сетки. Некоторые плитки сетки являются проходимыми, а другие приводят к тому, что агент падает в ледяную воду и проигрывает. Кроме того, направление движения агента является неопределенным и только частично зависит от выбранного направления. Агент вознаграждается за то, что он нашел путь к цели.

Легенда

Наступила зима. Вы и ваши друзья бросали фрисби в парке, и вдруг вы сделали сумасшедший бросок, который отправил фрисби на середину озера. Вода в основном замерзла, но есть несколько мест, где лед растаял. Если вы наступите на одну из этих дыр, вы попадете в замерзшую воду. В настоящее время ощущается нехватка фрисби, поэтому абсолютно необходимо, чтобы вы прошли по льду озера и забрали диск. Однако лед скользкий, поэтому вы не всегда будете двигаться в том направлении, в каком собираетесь.

Поверхность описывается с использованием сетки, как показано ниже:

S : начальное положение, безопасная клетка

F : замерзшее озеро, безопасная клетка

H : прорубь, проигрыш при попадании сюда

G : цель, где находится фрисби

Эпизод игры завершается, когда вы попадаете в прорубь (проигрыш, награда = 0), либо когда вы добираетесь до фрисби (выигрыш, награда = 1).

Понятие среды (Environment)

Чтобы понять основы импорта пакетов Gym, загрузки среды и других важных функций, связанных с OpenAI Gym, вот пример среды Frozen Lake.

Загрузка среды Frozen Lake:

```
import gym
env = gym.make('FrozenLake-v0')
```

Далее разберемся, как сбрасывать состояние среды. Пока агент в обучении с подкреплением учится, он многократно повторяет действия и эпизоды обучения. Из-за этого необходимо, чтобы в начале каждого эпизода среда была в начальном состоянии. Сбросить состояние среды можно следующим образом:

```
import gym
env = gym.make('FrozenLake-v0')
s = env.reset()
print(s)

-----------
0 #initial state is 0
```

После совершения какого-либо действия может возникнуть необходимость показать статус агента в среде. Визуализация среды:

```
env.render()

------------
SFFF
FHFH
FFFH
HFFG
```

Описание среды

Так как мир игры представляет собой сетку 4x4, существует 16 возможных состояний.

Узнать количество состояний можно так:

```
print(env.observation_space)

------------
Discrete(16)
```

Для клетки с дырой во льду предусмотрен штраф – 10 очков. Для клетки с целью – награда в 100 очков. Для того, чтобы маршруты не были излишне длинными, за каждый переход на соседнюю ледяную клетку существует штраф 1 очко.

Описание агента

У агента есть 4 действия: пойти влево, вниз, направо, вверх. Задача агента дойти до цели, не упав в ледяную воду. Для агента будет лучше, если он сможет дойти до цели наиболее коротким маршрутом.

Посмотреть действия агента:

```
print(env.action_space)

------------
Discrete(4)
```

Q-обучение игрового агента для FrozenLake-v0

Q-таблица для агента будет иметь 16 строк (по числу возможных состояний в игре) и 4 столбца (по числу доступных агенту действий):

```
print("Number of actions : ", env.action_space.n)
print("Number of states : ", env.observation_space.n)

----------------------
Number of actions : 4
Number of states : 16
```

Алгоритм Q-обучения

The steps involved in Q-learning are as follows:

1. Initialize the Q-table with zeros (eventually, updating will happen with a reward received for each action taken during learning).
2. Updating of a Q value for a state-action pair, that is, Q(s, a) is given by:

$$Q(s,a) <= Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

In this formula:

- s = current state
- a = action taken (choosing new action through epsilon-greedy approach)
- s' = resulted new state
- a' = action for the new state
- r = reward received for the action a
- α = learning rate, that is, the rate at which the learning of the agent converges towards minimized error
- γ= discount factor, that is, discounts the future reward to get an idea of how important that future reward is with regards to the current reward

3. By updating the Q-values as per the formula mentioned in step 2, the table converges to obtain accurate values for an action in a given state.

Формула Q-обучения

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Для обучения будет использована ε-жадная стратегия. Это значит, что агент выбирает свои действия таким образом, что с определенной вероятностью он выберет лучшее действие в данном состоянии среды, в другом же случае ходит случайным образом. Это позволяет найти компромисс между стратегиями исследования среды и использования накопленных знаний.

Например, эпсилон-жадная стратегия может быть реализована так:

```
def epsilon_greedy(state, epsilon):
    if np.random.uniform(0, 1) < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(Q[state, :])
    return action
```

Параметр *epsilon* будет уменьшаться для того, чтобы постепенно агент начал делать более обоснованный выбор, вместо случайных действий. Со временем агент достаточно исследует среду, после чего лучше делать самые выгодные действия всё чаще, а случайные всё реже. Пора начать использовать полученную ранее информацию о среде.

Реализация базового алгоритма Q-обучения

```
import gym
import numpy as np
import pickle
import os
from tqdm import tqdm
from time import sleep

# loading Frozen Lake environment from gym
env = gym.make('FrozenLake-v0')

os.system('cls')
print(f"\nLearning in {str(env.spec)[8:-1]}...\n")

# constant parameters
TOTAL_EPISODES = 20_000
MAX_STEPS = 100
EPSILON_DECAY_RATE = 1 / TOTAL_EPISODES
MIN_EPSILON = 0.001

# RL parameters
gamma = 0.9
lr_rate = 0.5
epsilon = 1

# Q-table initialization
Q = np.zeros((env.observation_space.n, env.action_space.n))


def epsilon_greedy(state):
    if np.random.uniform(0, 1) < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(Q[state, :])
    return action


def learn(state, state2, reward, action):
    # same as formula for Q-learning
    # Q_new[s, a] <- Q_old[s, a] + alpha * (r + gamma * max_a(Q_old[s_new, a]) - Q_old[s,
a])

    Q[state, action] = Q[state, action] + lr_rate * \
                       (reward + gamma * np.max(Q[state2, :]) - Q[state, action])


# Start
for episode in tqdm(range(TOTAL_EPISODES), ascii=True, unit="episode"):
    state = env.reset()
    step = 0

    # decreasing epsilon
    if epsilon > MIN_EPSILON:
        epsilon -= EPSILON_DECAY_RATE
    else:
        epsilon = MIN_EPSILON

    # loop within episode
    while step < MAX_STEPS:
        action = epsilon_greedy(state)
```

```
        new_state, reward, done, info = env.step(action)

        # debug to see what's happening ("Ctrl + /" - uncomment highlighted code)

        # print("state --action--> new_state")
        # print("  {}       {}         {}".format(state, action, new_state))
        # env.render()

        if done and (reward == 0):
            reward = -10  # fell into the hole
        elif done and (reward == 1):
            reward = 100  # goal achieved
        else:
            reward = -1  # step on ice

        # doing the learning
        learn(state, new_state, reward, action)
        state = new_state
        step += 1

        if done:
            break

# print("\nQ-table:\n", Q)

# save Q-table in file on drive (same directory)
with open("frozenLake_qTable.pkl", 'wb') as f:
    pickle.dump(Q, f)


##################
# PLAYING STAGE #
##################

# comment all below, if you only need to train agent

print(f"\nPlaying {str(env.spec)[8:-1]}...\n")

# load q-table from file
with open("frozenLake_qTable.pkl", 'rb') as f:
    Q = pickle.load(f)

win = 0
defeat = 0

# for visualization on win cmd
show_play = True

# Start
for episode in range(1000):
    state = env.reset()
    step = 0

    while step < MAX_STEPS:
        action = np.argmax(Q[state, :])
        new_state, reward, done, info = env.step(action)

        # Windows CLI visualization
        if show_play:
            os.system('cls')

            try:
```

```
                win_rate = win / (win + defeat) * 100
                print("Win rate: {}%".format(win_rate))
                print(f"Wins: {win}\n"
                      f"Defeats: {defeat}\n")
            except ZeroDivisionError:
                print("Win rate: 0.0%")
                print(f"Wins: {win}\n"
                      f"Defeats: {defeat}\n")

            env.render()  # show env

            if done:
                print("\n\tWIN" if reward == 1 else "\n\tDEFEAT")
                sleep(0.6)

        state = new_state

        if done:
            if reward == 1:
                win += 1
            else:
                defeat += 1

            break

        step += 1

    if show_play and (step >= MAX_STEPS):
        print("\nTIME IS OVER (steps > 100)")
        sleep(1)

win_rate = win / (win + defeat) * 100
print("Win rate: {}%".format(win_rate))
```