****Московский государственный технический университет им. Н.Э. Баумана

Кафедра «Системы обработки информации и управления»

Лабораторная работа №5

по дисциплине

«Методы машинного обучения»

на тему

# **«Обучение на основе временны’х различий»**

Выполнил:

студент группы ИУ5-22М

Джин Шуо

Москва — 2024 г.

**1. Цель лабораторной работы**

Oзнакомление с базовыми методами обучения с подкреплением на основе временных различий.

**2. Задание**

На основе рассмотренного на лекции примера реализуйте следующие алгоритмы:

SARSA

Q-обучение

Двойное Q-обучение

для любой среды обучения с подкреплением (кроме рассмотренной на лекции среды Toy Text / Frozen Lake) из библиотеки Gym (или аналогичной библиотеки).

**3. текст программы**

import numpy as np

import matplotlib.pyplot as plt

import gym

from tqdm import tqdm

class BasicAgent:

''' Базовый агент, от которого наследуются стратегии обучения '''

# Наименование алгоритма

ALГО\_NAME = '---'

def \_\_init\_\_(self, env, eps=0.1, num\_bins=(6, 6, 6, 6)):

# Среда

self.env = env

# Размерности Q-матрицы

self.nA = env.action\_space.n

self.num\_bins = num\_bins

self.state\_bins = [

np.linspace(-4.8, 4.8, num\_bins[0] - 1),

np.linspace(-4, 4, num\_bins[1] - 1),

np.linspace(-0.418, 0.418, num\_bins[2] - 1),

np.linspace(-4, 4, num\_bins[3] - 1)

]

# и сама матрица

self.Q = np.zeros(num\_bins + (self.nA,))

# Значения коэффициентов

# Порог выбора случайного действия

self.eps = eps

# Награды по эпизодам

self.episodes\_reward = []

def print\_q(self):

print('Вывод Q-матрицы для алгоритма ', self.ALGO\_NAME)

print(self.Q)

def discretize\_state(self, state):

''' Возвращает дискретное состояние для текущего состояния'''

state\_adj = [np.digitize(state[i], self.state\_bins[i]) for i in range(len(state))]

return tuple(state\_adj)

def greedy(self, state):

'''<<Жадное>> текущее действие

Возвращает действие, соответствующее максимальному Q-значению

для состояния state'''

return np.argmax(self.Q[state])

def make\_action(self, state):

'''Выбор действия агентом'''

if np.random.uniform(0, 1) < self.eps:

# Если вероятность меньше eps

# то выбирается случайное действие

return self.env.action\_space.sample()

else:

# иначе действие, соответствующее максимальному Q-значению

return self.greedy(state)

def draw\_episodes\_reward(self):

# Построение графика наград по эпизодам

fig, ax = plt.subplots(figsize=(15, 10))

y = self.episodes\_reward

x = list(range(1, len(y) + 1))

plt.plot(x, y, '-', linewidth=1, color='green')

plt.title('Награды по эпизодам')

plt.xlabel('Номер эпизода')

plt.ylabel('Награда')

plt.show()

def learn(self):

'''Реализация алгоритма обучения'''

pass

class SARSA\_Agent(BasicAgent):

'''Реализация алгоритма SARSA'''

ALGO\_NAME = 'SARSA'

def \_\_init\_\_(self, env, eps=0.4, lr=0.1, gamma=0.98, num\_episodes=20000):

# Вызов конструктора верхнего уровня

super().\_\_init\_\_(env, eps)

# Learning rate

self.lr = lr

# Коэффициент дисконтирования

self.gamma = gamma

# Количество эпизодов

self.num\_episodes = num\_episodes

# Постепенное уменьшение eps

self.eps\_decay = 0.00005

self.eps\_threshold = 0.01

def learn(self):

'''Обучение на основе алгоритма SARSA'''

self.episodes\_reward = []

# Цикл по эпизодам

for ep in tqdm(list(range(self.num\_episodes))):

# Начальное состояние среды

state = self.discretize\_state(self.env.reset()[0])

# Флаг штатного завершения эпизода

done = False

truncated = False

# Суммарная награда по эпизоду

tot\_rew = 0

# По мере заполнения Q-матрицы уменьшаем вероятность случайного выбора действия

if self.eps > self.eps\_threshold:

self.eps -= self.eps\_decay

# Выбор действия

action = self.make\_action(state)

# Проигрывание одного эпизода до финального состояния

while not done and not truncated:

# Выполняем шаг в среде

next\_state, rew, done, truncated, \_ = self.env.step(action)

next\_state = self.discretize\_state(next\_state)

# Выполняем следующее действие

next\_action = self.make\_action(next\_state)

# Правило обновления Q для SARSA

self.Q[state][action] = self.Q[state][action] + self.lr \* \

(rew + self.gamma \* self.Q[next\_state][next\_action] - self.Q[state][action])

# Следующее состояние считаем текущим

state = next\_state

action = next\_action

# Суммарная награда за эпизод

tot\_rew += rew

if done or truncated:

self.episodes\_reward.append(tot\_rew)

class QLearning\_Agent(BasicAgent):

'''Реализация алгоритма Q-Learning'''

ALГО\_NAME = 'Q-обучение'

def \_\_init\_\_(self, env, eps=0.4, lr=0.1, gamma=0.98, num\_episodes=20000):

# Вызов конструктора верхнего уровня

super().\_\_init\_\_(env, eps)

# Learning rate

self.lr = lr

# Коэффициент дисконтирования

self.gamma = gamma

# Количество эпизодов

self.num\_episodes = num\_episodes

# Постепенное уменьшение eps

self.eps\_decay = 0.00005

self.eps\_threshold = 0.01

def learn(self):

'''Обучение на основе алгоритма Q-Learning'''

self.episodes\_reward = []

# Цикл по эпизодам

for ep in tqdm(list(range(self.num\_episodes))):

# Начальное состояние среды

state = self.discretize\_state(self.env.reset()[0])

# Флаг штатного завершения эпизода

done = False

truncated = False

# Суммарная награда по эпизоду

tot\_rew = 0

# По мере заполнения Q-матрицы уменьшаем вероятность случайного выбора действия

if self.eps > self.eps\_threshold:

self.eps -= self.eps\_decay

# Проигрывание одного эпизода до финального состояния

while not done and not truncated:

# Выбор действия

action = self.make\_action(state)

# Выполняем шаг в среде

next\_state, rew, done, truncated, \_ = self.env.step(action)

next\_state = self.discretize\_state(next\_state)

# Правило обновления для Q-обучения

self.Q[state][action] = self.Q[state][action] + self.lr \* \

(rew + self.gamma \* np.max(self.Q[next\_state]) - self.Q[state][action])

# Следующее состояние считаем текущим

state = next\_state

# Суммарная награда за эпизод

tot\_rew += rew

if done or truncated:

self.episodes\_reward.append(tot\_rew)

class DoubleQLearning\_Agent(BasicAgent):

'''Реализация алгоритма Double Q-Learning'''

ALGO\_NAME = 'Двойное Q-обучение'

def \_\_init\_\_(self, env, eps=0.4, lr=0.1, gamma=0.98, num\_episodes=20000):

# Вызов конструктора верхнего уровня

super().\_\_init\_\_(env, eps)

# Вторая матрица

self.Q2 = np.zeros(self.Q.shape)

# Learning rate

self.lr = lr

# Коэффициент дисконтирования

self.gamma = gamma

# Количество эпизодов

self.num\_episodes = num\_episodes

# Постепенное уменьшение eps

self.eps\_decay = 0.00005

self.eps\_threshold = 0.01

def greedy(self, state):

''' <<Жадное>> текущее действие

Возвращает действие, соответствующее максимальному Q-значению

для состояния state'''

temp\_q = self.Q[state] + self.Q2[state]

return np.argmax(temp\_q)

def print\_q(self):

print('Вывод Q-матриц для алгоритма ', self.ALGO\_NAME)

print('Q1')

print(self.Q)

print('Q2')

print(self.Q2)

def learn(self):

'''Обучение на основе алгоритма Double Q-Learning'''

self.episodes\_reward = []

# Цикл по эпизодам

for ep in tqdm(list(range(self.num\_episodes))):

# Начальное состояние среды

state = self.discretize\_state(self.env.reset()[0])

# Флаг штатного завершения эпизода

done = False

truncated = False

# Суммарная награда по эпизоду

tot\_rew = 0

# По мере заполнения Q-матрицы уменьшаем вероятность случайного выбора действия

if self.eps > self.eps\_threshold:

self.eps -= self.eps\_decay

# Проигрывание одного эпизода до финального состояния

while not done and not truncated:

# Выбор действия

action = self.make\_action(state)

# Выполняем шаг в среде

next\_state, rew, done, truncated, \_ = self.env.step(action)

next\_state = self.discretize\_state(next\_state)

if np.random.rand() < 0.5:

# Обновление первой таблицы

self.Q[state][action] = self.Q[state][action] + self.lr \* \

(rew + self.gamma \* self.Q2[next\_state][np.argmax(self.Q[next\_state])] - self.Q[state][action])

else:

# Обновление второй таблицы

self.Q2[state][action] = self.Q2[state][action] + self.lr \* \

(rew + self.gamma \* self.Q[next\_state][np.argmax(self.Q2[next\_state])] - self.Q2[state][action])

# Следующее состояние считаем текущим

state = next\_state

# Суммарная награда за эпизод

tot\_rew += rew

if done or truncated:

self.episodes\_reward.append(tot\_rew)

def play\_agent(agent):

'''Проигрывание сессии для обученного агента'''

env2 = gym.make('CartPole-v1', render\_mode='human')

state, \_ = env2.reset()

done = False

while not done:

state = agent.discretize\_state(state)

action = agent.greedy(state)

next\_state, reward, done, truncated, info = env2.step(action)

env2.render()

state = next\_state

if done or truncated:

break

env2.close()

def run\_sarsa():

env = gym.make('CartPole-v1')

agent = SARSA\_Agent(env)

agent.learn()

agent.print\_q()

agent.draw\_episodes\_reward()

play\_agent(agent)

def run\_q\_learning():

env = gym.make('CartPole-v1')

agent = QLearning\_Agent(env)

agent.learn()

agent.print\_q()

agent.draw\_episodes\_reward()

play\_agent(agent)

def run\_double\_q\_learning():

env = gym.make('CartPole-v1')

agent = DoubleQLearning\_Agent(env)

agent.learn()

agent.print\_q()

agent.draw\_episodes\_reward()

play\_agent(agent)

def main():

#run\_sarsa()

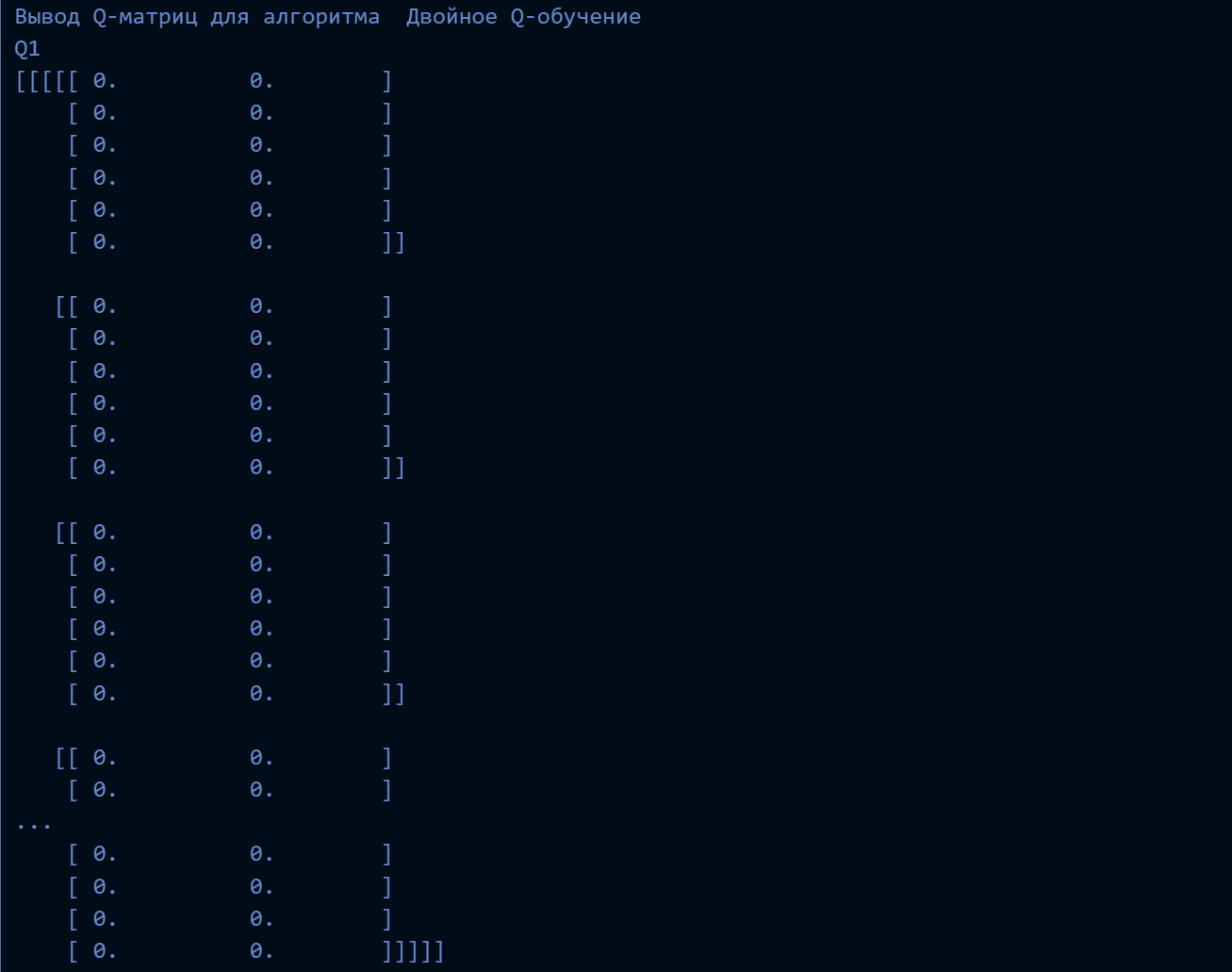
#run\_q\_learning()

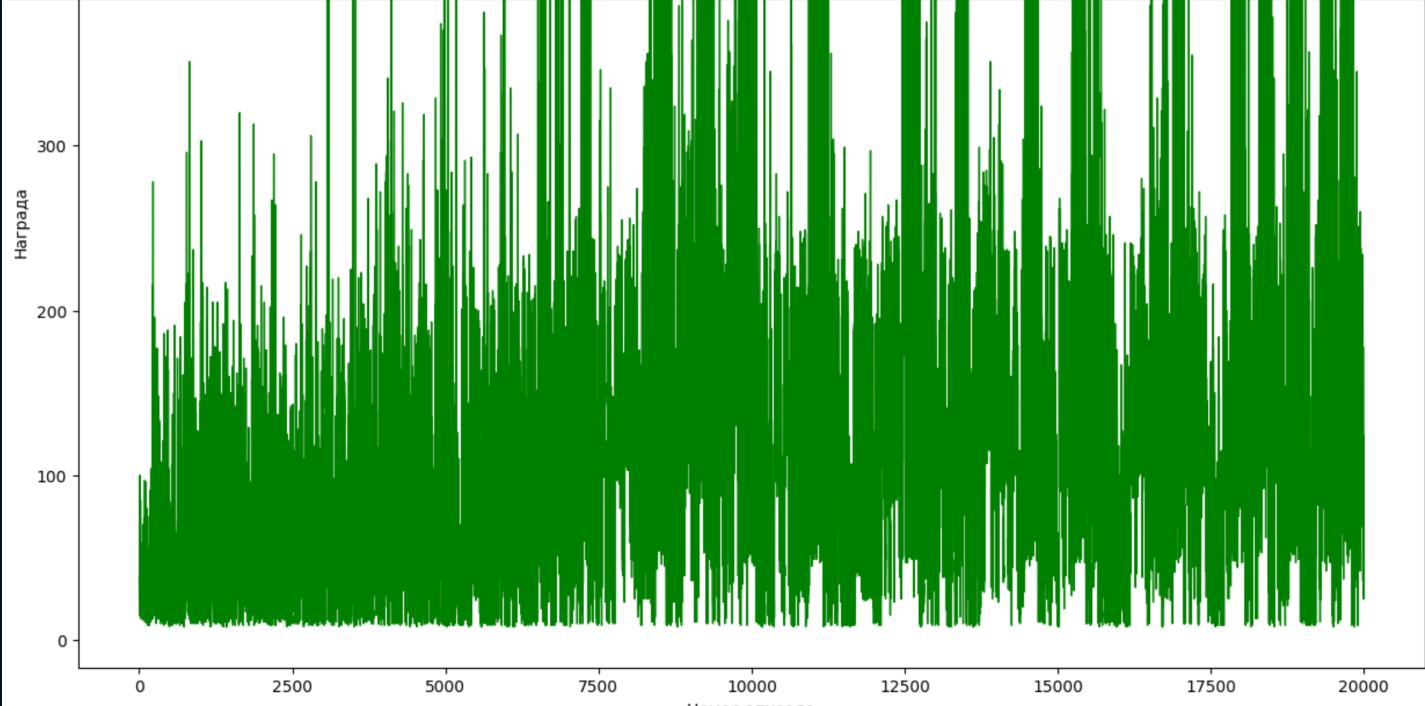
run\_double\_q\_learning()

if \_\_name\_\_ == '\_\_main\_\_':

main()

1. **экранные формы с примерами выполнения программы.**





**Список литературы**

[1] https://github.com/ugapanyuk/courses\_current/wiki/LAB\_MMO\_RL\_TD