Московский государственный технический университет им. Н.Э. Баумана
Факультет «Информатика и системы управления»
Кафедра «Системы обработки информации и управления»

«Методы машинного обучения в автоматизированных системах обработки информации и управления»

Лабораторная работа №5

«Обучение на основе временных различий»

ИСПОЛНИТЕЛЬ:

Васильев Д.А.
Группа ИУ5-21М

_____

"__"_____2023 г.

Москва 2023

1. 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).

2. Листинг

2.1. BasicAgent.py

```
import numpy as np
import plotly.express as px
import pandas as pd
import os
import pygame


os.environ['SDL_VIDEODRIVER']='dummy'
pygame.display.set_mode((640,480))


class BasicAgent: #Базовый агент, от которого наследуются стратегии обучения

    # Наименование алгоритма
    ALGO_NAME = 'Base'

    def __init__(self, env, eps=0.1):
        # Среда
        self.env = env
        # Размерности Q-матрицы
        self.nA = env.action_space.n
        self.nS = env.observation_space.n
        #и сама матрица
        self.Q = np.zeros((self.nS, 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 get_state(self, state): #Возвращает правильное начальное состояние

        if type(state) is tuple:
            # Если состояние вернулось с виде кортежа, то вернуть только номер
состояния
            return state[0]
        else:
            return state

    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

        df = pd.DataFrame(data={
            'Номер эпизода': list(range(1, len(y)+1)),
            'Награда': y
        })

        fig = px.line(df, x="Номер эпизода", y="Награда", title='Награды по
эпизодам', height=400, width=600)
        fig.show()
        # plt.plot(x, y, '-', linewidth=1, color='green')
        # plt.title('')
        # plt.xlabel()
        # plt.ylabel('Награда')
        # plt.show()

    def learn(self):
```

```
        '''
        Реализация алгоритма обучения
        '''
        pass
```

2.2. SARSA_Agent.py

```
from tqdm import tqdm
import os
import pygame
from BasicAgent import BasicAgent


os.environ['SDL_VIDEODRIVER']='dummy'
pygame.display.set_mode((640,480))


class SARSA_Agent(BasicAgent):
    '''
    Реализация алгоритма SARSA
    '''
    # Наименование алгоритма
    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)), bar_format='
{l_bar}{bar:20}{r_bar}{bar:-10b}', colour='CYAN'):
            # Начальное состояние среды
            state = self.get_state(self.env.reset())
            # Флаг штатного завершения эпизода
            done = False
            # Флаг нештатного завершения эпизода
```

```
            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 or truncated):
                # Выполняем шаг в среде
                next_state, rew, done, truncated, _ = self.env.step(action)

                # 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)
```

2.3. QLearning_Agent.py

```
import numpy as np
from tqdm import tqdm
import os
import pygame
from BasicAgent import BasicAgent


os.environ['SDL_VIDEODRIVER']='dummy'
pygame.display.set_mode((640,480))


class QLearning_Agent(BasicAgent):
    '''
    Реализация алгоритма Q-Learning
    '''
```

```
    # Наименование алгоритма
    ALGO_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)), bar_format='
{l_bar}{bar:20}{r_bar}{bar:-10b}', colour='CYAN'):
            # Начальное состояние среды
            state = self.get_state(self.env.reset())
            # Флаг штатного завершения эпизода
            done = False
            # Флаг нештатного завершения эпизода
            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 or truncated):
                # Выбор действия
                # В SARSA следующее действие выбиралось после шага в среде
                action = self.make_action(state)

                # Выполняем шаг в среде
                next_state, rew, done, truncated, _ = self.env.step(action)

                # Правило обновления Q для SARSA (для сравнения)
                # self.Q[state][action] = self.Q[state][action] + self.lr * \
                #     (rew + self.gamma * self.Q[next_state][next_action] -
self.Q[state][action])
```

```
                # 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)
```

2.4. DoubleQLearning_Agent.py

```
import numpy as np
from tqdm import tqdm
import os
import pygame
from BasicAgent import BasicAgent


os.environ['SDL_VIDEODRIVER']='dummy'
pygame.display.set_mode((640,480))


class DoubleQLearning_Agent(BasicAgent):
    '''
    Реализация алгоритма Double Q-Learning
    '''
    # Наименование алгоритма
    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.nS, self.nA))
        # 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(f"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)), bar_format='
{l_bar}{bar:20}{r_bar}{bar:-10b}', colour='CYAN'):
            # Начальное состояние среды
            state = self.get_state(self.env.reset())
            # Флаг штатного завершения эпизода
            done = False
            # Флаг нештатного завершения эпизода
            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 or truncated):
                # Выбор действия
                # В SARSA следующее действие выбиралось после шага в среде
                action = self.make_action(state)

                # Выполняем шаг в среде
                next_state, rew, done, truncated, _ = self.env.step(action)

                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])
```

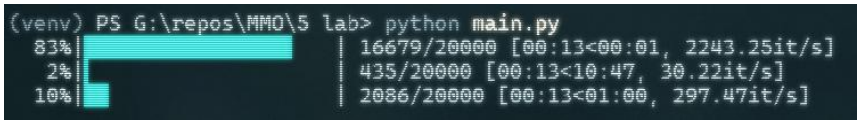
```
                else:
                    # Обновление второй таблицы
                    self.Q2[state][action] = self.Q2[state][action] + self.lr * \
                        (rew + self.gamma *
self.Q[next_state][np.argmax(self.Q2[next_state])] - self.Q2[state][action])

                # Следующее состояние считаем текущим
                state = next_state
                # Суммарная награда за эпизод
                tot_rew += rew
                if (done or truncated):
                    self.episodes_reward.append(tot_rew)
```

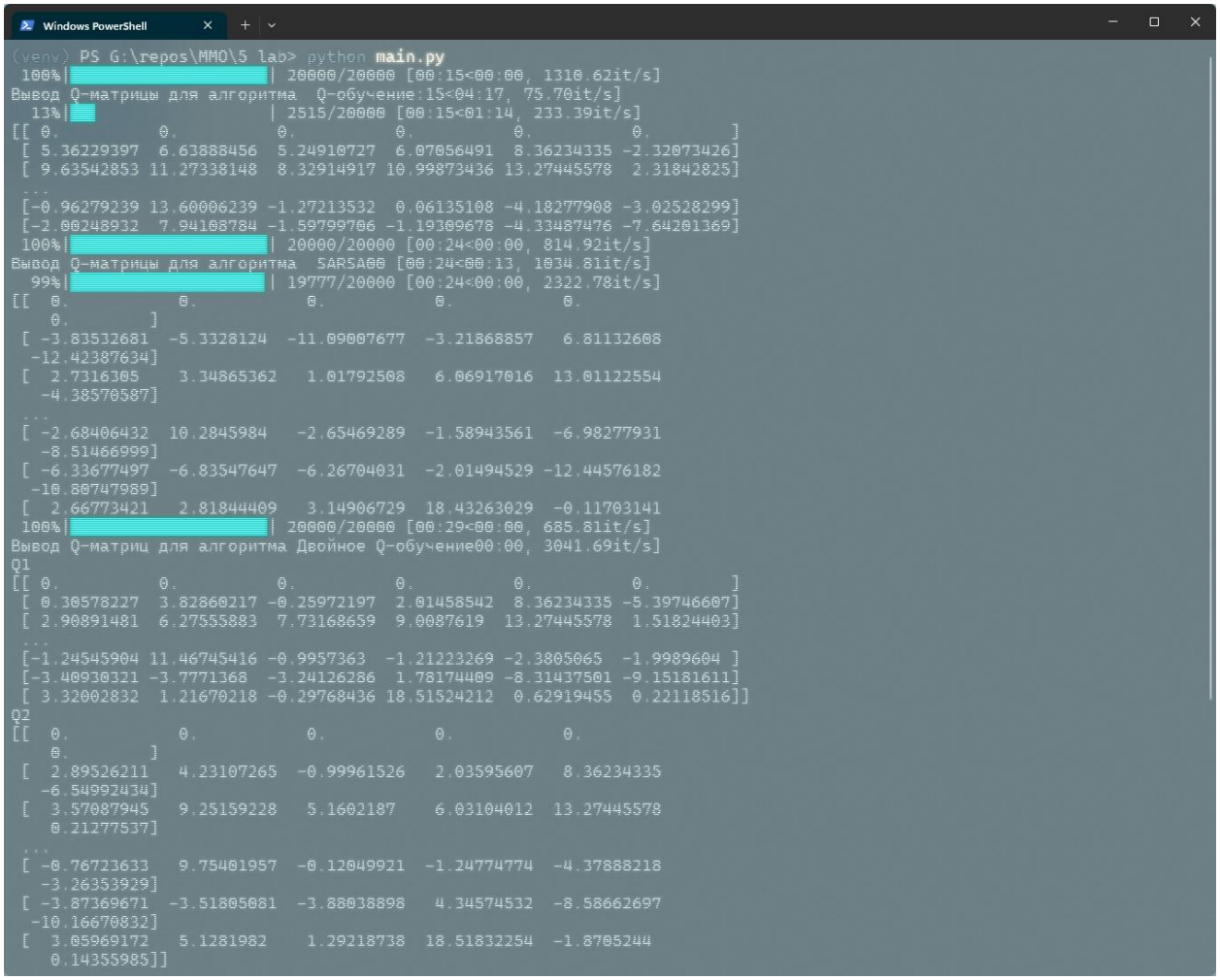
2.5. main.py

```
import gymnasium as gym
import os
import pygame
import asyncio


from DoubleQLearning_Agent import DoubleQLearning_Agent
from QLearning_Agent import  QLearning_Agent
from SARSA_Agent import SARSA_Agent


os.environ['SDL_VIDEODRIVER']='dummy'
pygame.display.set_mode((640,480))


def play_agent(agent):
    '''
    Проигрывание сессии для обученного агента
    '''
    env2 = gym.make('Taxi-v3', render_mode='human')
    state = env2.reset()[0]
    done = False
    while not done:
        action = agent.greedy(state)
        next_state, reward, terminated, truncated, _ = env2.step(action)
        env2.render()
        state = next_state
        if terminated or truncated:
            done = True


def run_sarsa():
    env = gym.make('Taxi-v3')
    agent = SARSA_Agent(env)
    agent.learn()
```

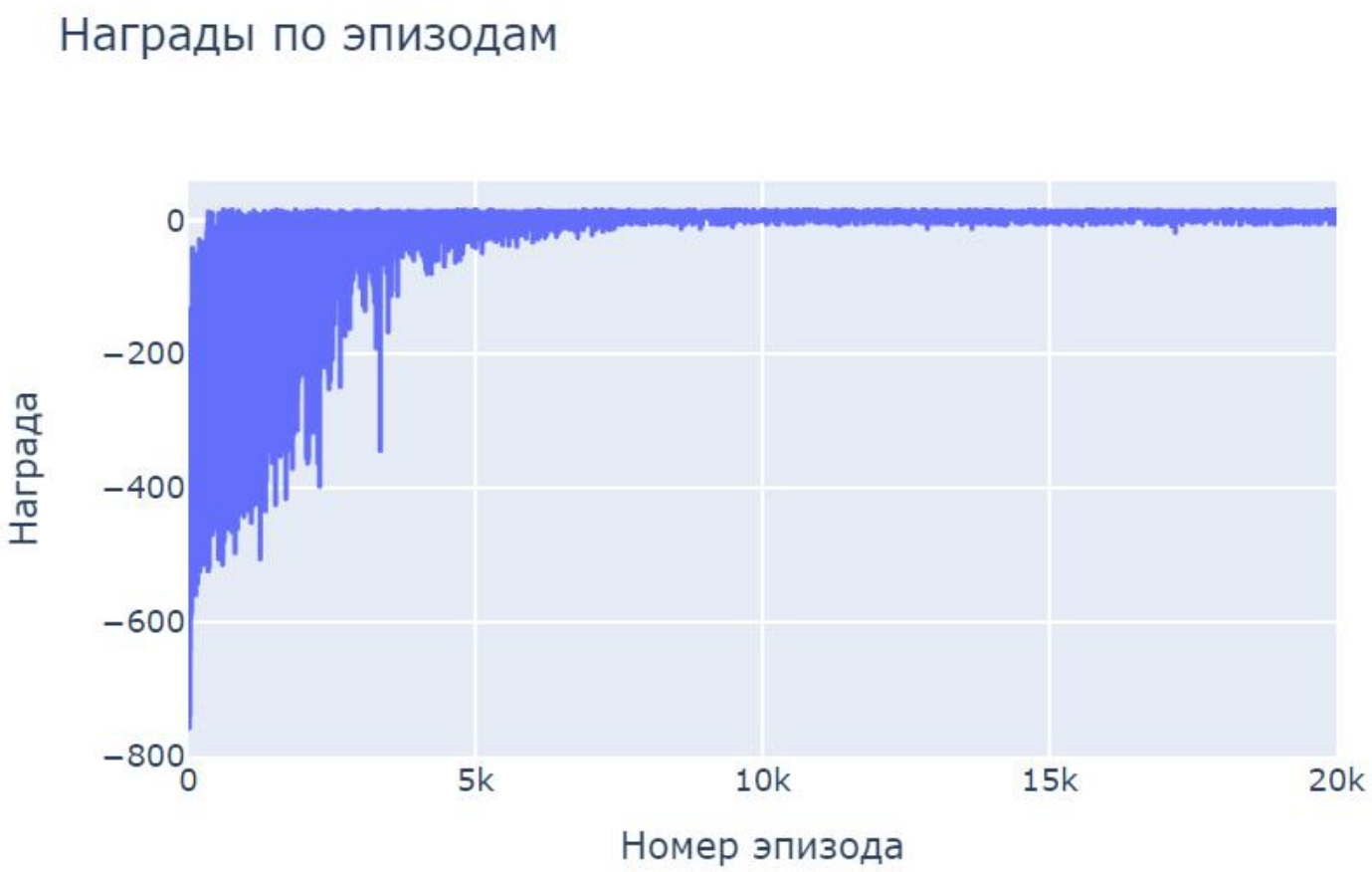
```
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)


def run_q_learning():
    env = gym.make('Taxi-v3')
    agent = QLearning_Agent(env)
    agent.learn()
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)


def run_double_q_learning():
    env = gym.make('Taxi-v3')
    agent = DoubleQLearning_Agent(env)
    agent.learn()
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)


async def main():
    treads = [asyncio.to_thread(run_q_learning), asyncio.to_thread(run_sarsa),
asyncio.to_thread(run_double_q_learning)]

    tasks = [asyncio.create_task(tread) for tread in treads]

    await asyncio.gather(*tasks)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    loop.close()
```

3. Экранные формы

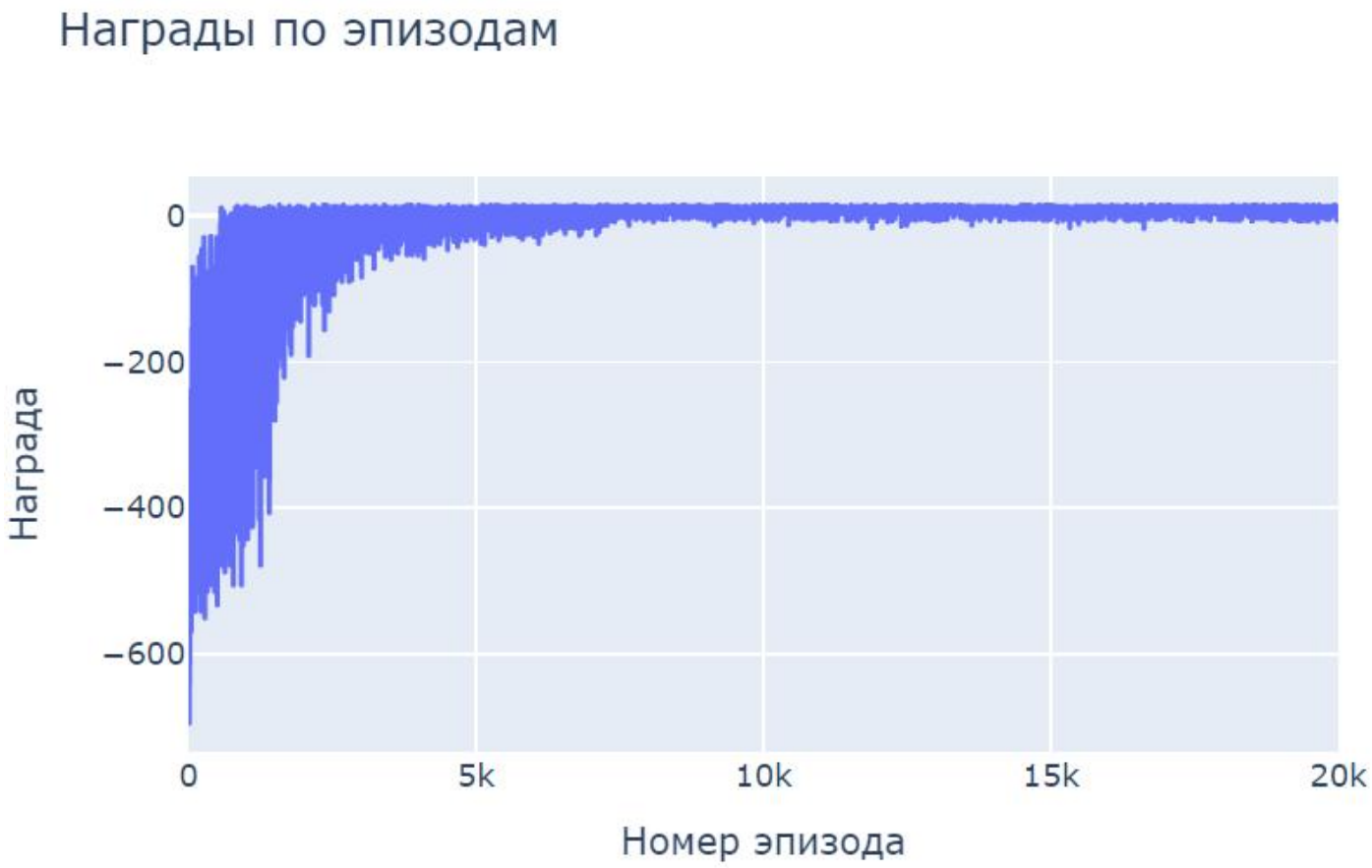




3.1. Q-обучение



3.2. SARSA


Награды по эпизодам
Награда
0
−200
−400
−600
0
5k
10k
15k
20k
Номер эпизода

3.3. Двойное Q-обучение