МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. Н.Э. Баумана

Факультет «Информатика и системы управления»
Кафедра «Систем обработки информации и управления»

ОТЧЕТ

Лабораторная работа №__6__
по дисциплине «Методы машинного обучения»

Тема: «Реализация алгоритма Policy Iteration.»

ИСПОЛНИТЕЛЬ: Коротков Н.К.
ФИО
группа ИУ5-23М
_____
подпись
"__"_____2024 г.

ПРЕПОДАВАТЕЛЬ: Гапанюк Ю.Е.
ФИО
_____
подпись
"__"_____2024 г.

Москва - 2024

Задание

1. На основе рассмотренных на лекции примеров реализуйте алгоритм DQN.
2. В качестве среды можно использовать классические среды (в этом случае используется полносвязная архитектура нейронной сети).
3. В качестве среды можно использовать игры Atari (в этом случае используется сверточная архитектура нейронной сети).
4. В случае реализации среды на основе сверточной архитектуры нейронной сети +1 балл за экзамен.
5. Сформировать отчет и разместить его в своем репозитории на github.

Выполнение

Для реализации была выбрана среда Acrobat-v0 из библиотеки Gym.

По документации: непрерывное пространство из 6 параметров, рассмотренных в таблице 1. Соответственно имеется 3 действия – рассмотрены в таблице 2.

Таблица 1 – параметры налюдения.

№	Параметр	Min	Max
0	Cosine of theta1	-1	1
1	Sine of theta1	-1	1
2	Cosine of theta2	-1	1
3	Sine of theta2	-1	1
4	Angular velocity of theta1	~ -12.567 (-4 * pi)	~ 12.567 (4 * pi)
5	Angular velocity of theta2	~ -28.274 (-9 * pi)	~ 28.274 (9 * pi)

Таблица 1 – пространство действий.

№	Действие	Unit
0	apply -1 torque to the actuated joint	torque (N m)
1	apply 0 torque to the actuated joint	torque (N m)
2	apply 1 torque to the actuated joint	torque (N m)

Действий 3 – при этом они взаимоисключающие. Используем классификационную НС.

Код программы:

```
import gym
import numpy as np
import matplotlib.pyplot as plt
```

```
from pprint import pprint
import pandas as pd
from gym.envs.toy_text.taxi import TaxiEnv


def print_full(x):
    pd.set_option('display.max_rows', len(x))
    print(x)
    pd.reset_option('display.max_rows')

class PolicyIterationAgent:
    '''
    Класс, эмулирующий работу агента
    '''
    def __init__(self, env):
        self.env = env
        # Пространство состояний
        self.observation_dim = 500
        # Массив действий в соответствии с документацией
        self.actions_variants = np.array([0,1,2,3,4,5])
        # Задание стратегии (политики)
        self.policy_probs = np.full((self.observation_dim,
len(self.actions_variants)), 0.16666666)
        # Начальные значения для v(s)
        self.state_values = np.zeros(shape=(self.observation_dim))
        # Начальные значения параметров
        self.maxNumberOfIterations = 1000
        self.theta=1e-6
        self.gamma=0.99


    def print_policy(self):
        '''
        Вывод матриц стратегии
        '''

        if self.policy_probs[0][0] != 0.16666666:
            #np.set_printoptions(threshold=np.inf)
            x = TaxiEnv()
            pos = {0:'R', 1:'G',2:'Y', 3:'B', 4:'T'}
            print('''
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
                  ''')
            print('состояние: x,y,пассажир,назначение')
            print('Стратегия:')
            for i in range(len(self.policy_probs)):
```

```
                t_x,t_y,passeng,dest = x.decode(i)
                print((t_x,t_y,pos[passeng],pos[dest]), self.policy_probs[i])
            #np.set_printoptions(threshold=False)
        else:
            print('Стратегия:')
            pprint(self.policy_probs)


    def policy_evaluation(self):
        '''
        Оценивание стратегии
        '''
        # Предыдущее значение функции ценности
        valueFunctionVector = self.state_values
        for iterations in range(self.maxNumberOfIterations):
            # Новое значение функции ценности
            valueFunctionVectorNextIteration=np.zeros(shape=(self.observation_dim
))
            # Цикл по состояниям
            for state in range(self.observation_dim):
                # Вероятности действий
                action_probabilities = self.policy_probs[state]
                # Цикл по действиям
                outerSum=0
                for action, prob in enumerate(action_probabilities):
                    innerSum=0
                    # Цикл по вероятностям действий
                    for probability, next_state, reward, isTerminalState in
self.env.P[state][action]:
                        innerSum=innerSum+probability*(reward+self.gamma*self.sta
te_values[next_state])
                    outerSum=outerSum+self.policy_probs[state][action]*innerSum
                valueFunctionVectorNextIteration[state]=outerSum
            if(np.max(np.abs(valueFunctionVectorNextIteration-
valueFunctionVector))<self.theta):
                # Проверка сходимости алгоритма
                valueFunctionVector=valueFunctionVectorNextIteration
                break
            valueFunctionVector=valueFunctionVectorNextIteration
        return valueFunctionVector


    def policy_improvement(self):
        '''
        Улучшение стратегии
        '''
        qvaluesMatrix=np.zeros((self.observation_dim,
len(self.actions_variants)))
        improvedPolicy=np.zeros((self.observation_dim,
len(self.actions_variants)))
        # Цикл по состояниям
        for state in range(self.observation_dim):
```

```
            for action in range(len(self.actions_variants)):
                for probability, next_state, reward, isTerminalState in
self.env.P[state][action]:
                    qvaluesMatrix[state,action]=qvaluesMatrix[state,action]+proba
bility*(reward+self.gamma*self.state_values[next_state])

            # Находим лучшие индексы
            bestActionIndex=np.where(qvaluesMatrix[state,:]==np.max(qvaluesMatrix
[state,:]))
            # Обновление стратегии
            improvedPolicy[state,bestActionIndex]=1/np.size(bestActionIndex)
        return improvedPolicy


    def policy_iteration(self, cnt):
        '''
        Основная реализация алгоритма
        '''
        policy_stable = False
        for i in range(1, cnt+1):
            self.state_values = self.policy_evaluation()
            self.policy_probs = self.policy_improvement()
        print(f'Алгоритм выполнился за {i} шагов.')


def play_agent(agent):
    env2 = gym.make('Taxi-v3',render_mode='human')
    state = env2.reset()[0]
    done = False
    while not done:
        p = agent.policy_probs[state]
        if isinstance(p, np.ndarray):
            action = np.random.choice(len(agent.actions_variants), p=p)
        else:
            action = p
        next_state, reward, terminated, truncated, _ = env2.step(action)
        env2.render()
        state = next_state
        if terminated or truncated:
            done = True


def main():
    # Создание среды
    env = gym.make('Taxi-v3')
    env.reset()
    # Обучение агента
    agent = PolicyIterationAgent(env)
    agent.print_policy()
    agent.policy_iteration(1000)
    agent.print_policy()
    # Проигрывание сцены для обученного агента
```

```
    play_agent(agent)


if __name__ == '__main__':
    main()
```

Вывод программы модифицирован для кодировки состояний: см. рис. 2.

Результаты выполнения:

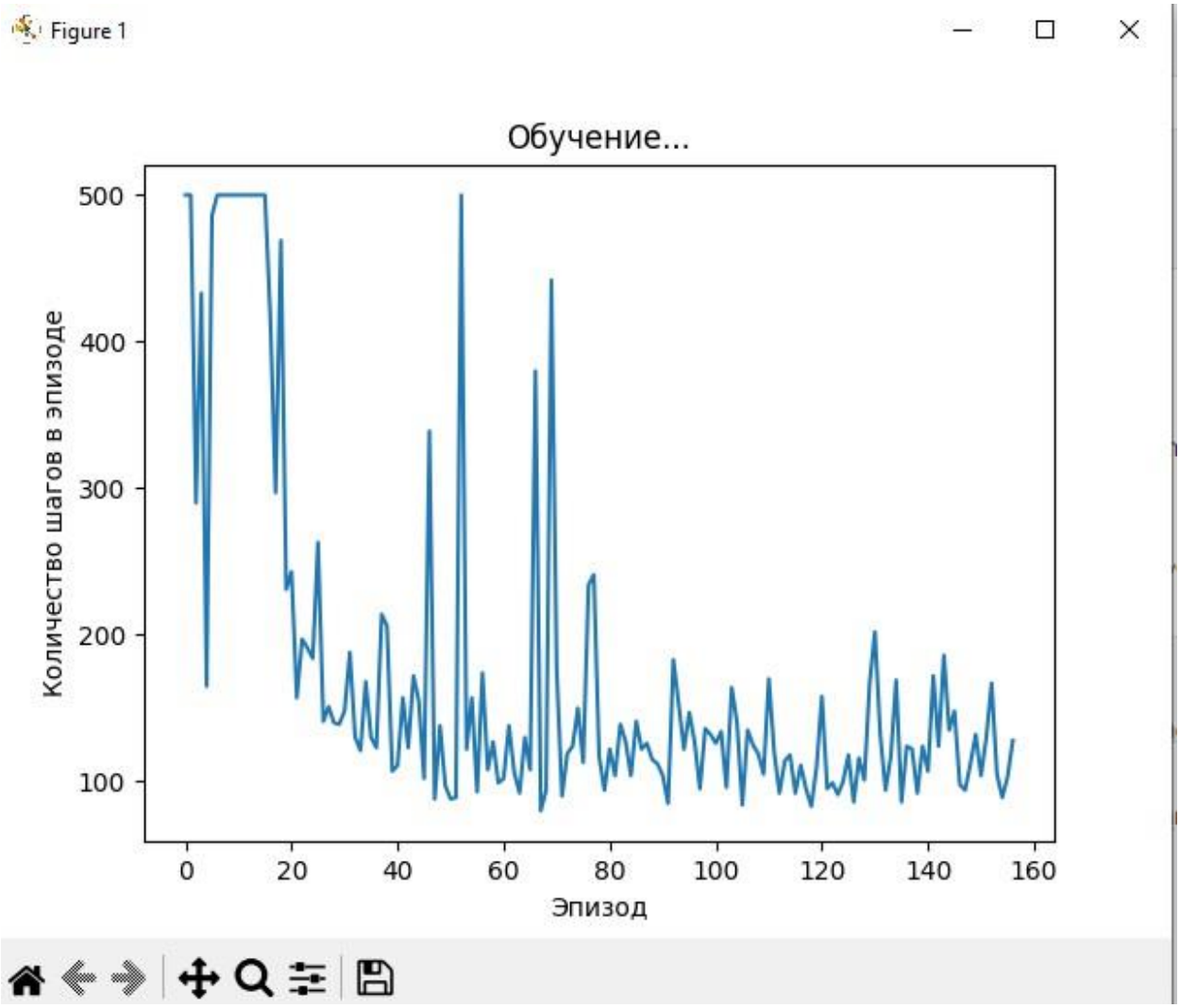

Рис. 1 – Начало обучнения.

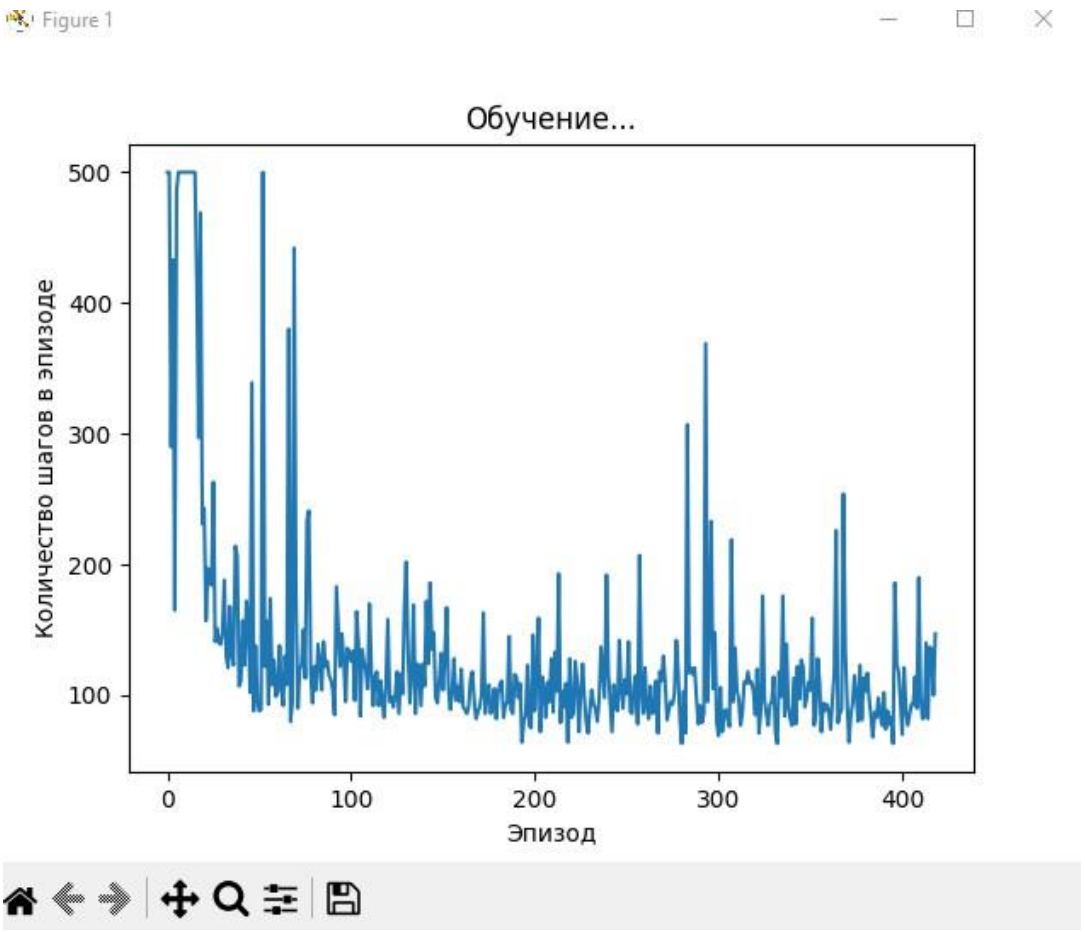

Рис. 2 – прогресс обучения.

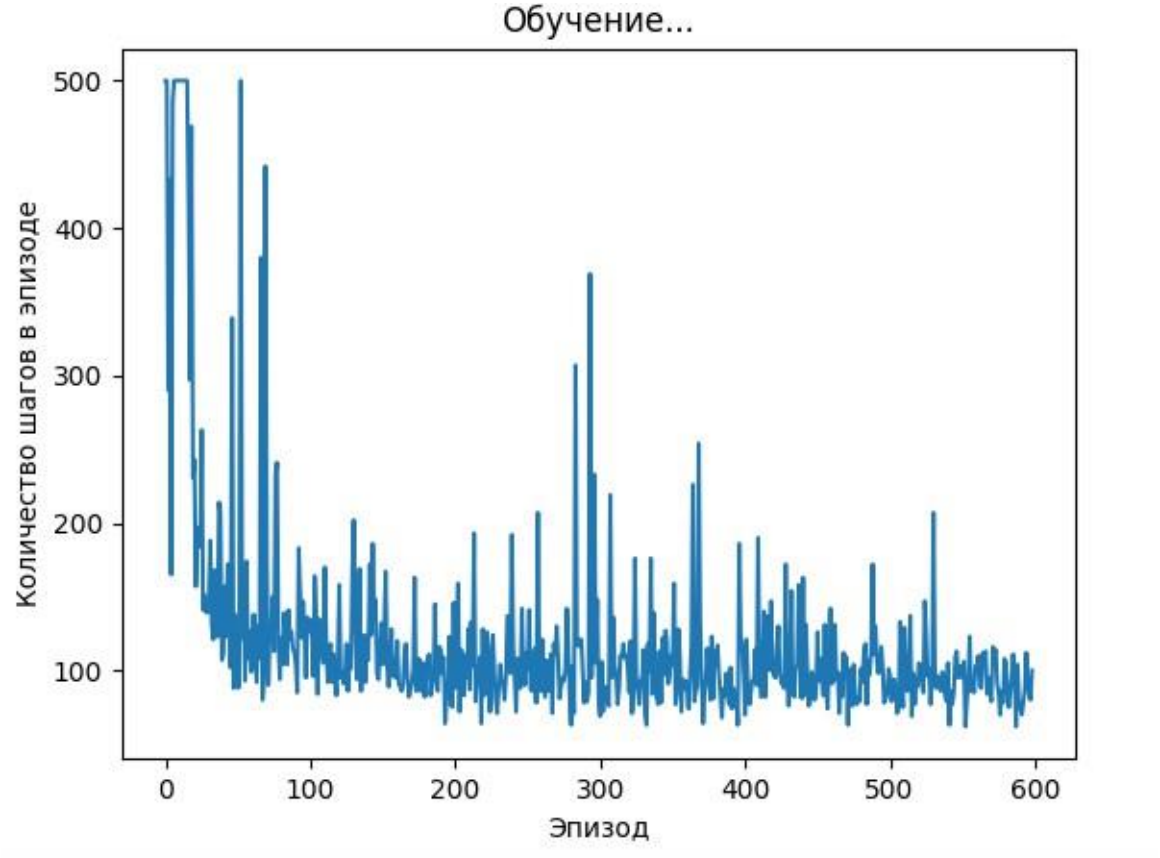

Рис. 3 – Конец обучения.

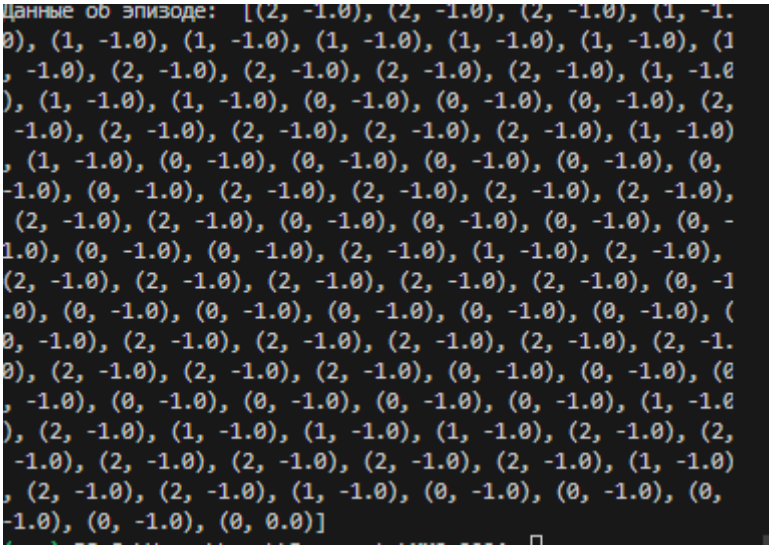

Рис. 4 – Финальный эпизод.

Рис. 5 – Пример движения актёра.

Вывод:

В ходе выполнения лабораторной работы мы ознакомились с базовыми методами обучения с подкреплением на основе глубоких Q-сетей.

Фактически, теперь вместо применения Q-матрицы используется обучающаяся нейронная сеть. Это позволяет совместить double-Q подход с инструментарием Н.С.

Пики на графе обучения – и выбросы НС, встретившейся с новым поведением среды, и выбор случайного действия жадным алгоритмом. Таким образом, исследуется сразу 2 направления оптимизации – градиентный спуск для известного НС пространства, и получение данных о неизвестном пространстве.