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

ФАКУЛЬТЕТ Информатика, системы управления и искусственный интеллект

КАФЕДРА Системы обработки информации и управления

Лабораторная работа №5

По курсу

«**Методы машинного обучения в АСОИУ**»

«Обучение на основе временных различий»
Cliff Walking

Выполнил:
Студент группы ИУ5-22М
Кириллов Д.С.
06.05.2024

Проверил:
Гапанюк Ю.Е.

Москва 2024 г.

Цель лабораторной работы

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

Задание

На основе рассмотренного на лекции примера реализуйте следующие алгоритмы:

1. SARSA
2. Q-обучение
3. Двойное Q-обучение

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

Ход работы

1. Описание среды Cliff Walking

Как и в предыдущей лабораторной, будем работать со средой Cliff Walking:

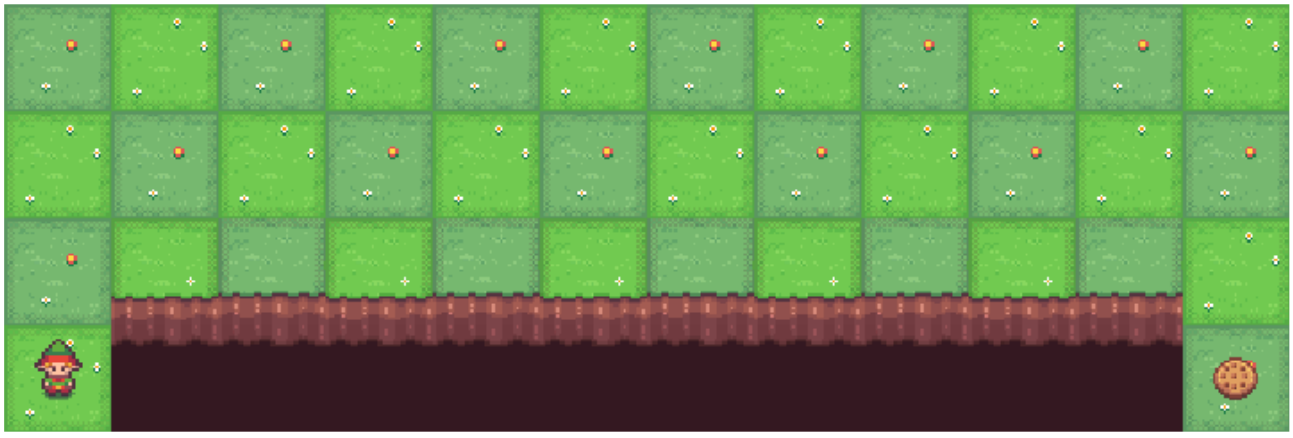

Рис. 1. Окно с демонстраций работы обученного агента в среде Gym[toy_text].

Поле (рис. 1) представляет собой матрицу 4x12. Агент начинает проходить карту с ячейки [3, 0] (левый нижний угол). Ему необходимо достичь ячейки [3,

11], т.е. цель размещена в правом нижнем углу. Также агенту нельзя наступать на обрыв – это ячейки [3, 1...10] (внизу по центру). Если агент наступит на обрыв, он вернется к началу. Эпизод заканчивается, когда агент достигает цели.

Агент может совершить 4 действия:

- 0: переместиться вверх;
- 1: передвинуться вправо;
- 2: передвинуться вниз;
- 3: передвинуться влево.

За каждый шаг полагается -1 награда, а за шаг в обрыв – штраф -100.

2. Программная часть

Произведем обучение с помощью алгоритмов SARSA, Q-обучения и двойного Q-обучения.

При обучении в предыдущей лабораторной (policy iteration) нам был известен граф переходов с вероятностями перехода и вознаграждениями. Однако граф и вероятности могут быть неизвестны. В этом случае можно учиться только на основе проб и ошибок, наблюдая среду, выполняя действия и получая вознаграждения. При этом предполагается, что агент может многократно «тренироваться» выполняя действия в среде в течение нескольких эпизодов, и отдельные эпизоды могут завершаться неудачно.

Основой TD-методов является Q-матрица (количество состояний x количество действий). Q-матрица выполняет роль аналогичную стратегии (политике).

2.1. SARSA

Алгоритм состоит из следующих действий:

1. Инициализация Q-функции произвольными значениями.

2. Выбор действия из состояния с использованием эпсилон-жадной стратегии (ε > 0) и переход в новое состояние.
3. Обновление Q предыдущего состояния по следующему правилу:

$$Q(s, \mathrm{a}) = Q(s, \mathrm{a}) + a(r + yQ(s', \mathrm{a}') — Q(s, \mathrm{a})),$$

где a' — действие, выбранное по эпсилон-жадной стратегии (ε > 0).

Коэффициент альфа часто также обозначают как learning rate.

Эпсилон-жадная стратегия заключается в выборе случайного действия в случае, если случайное число r < ε или в выборе лучшего действия, если r >= ε.

Реализуем алгоритм в коде:

Основу всех экспериментов будет составлять функция play_agent:

```
def play_agent(agent):
    '''
    Проигрывание сессии для обученного агента
    '''
    env2 = gym.make('CliffWalking-v0', 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
```

Для алгоритма SARSA напишем класс SARSA_Agent:

```
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))):
            # 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)
```

Напишем фикцию для инициализации класса SARSE_Agent и выполнения эксперимента:

```
def run_sarsa():
    env = gym.make('CliffWalking-v0')
    agent = SARSA_Agent(env)
    agent.learn()
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)
```

Выполним ее:

```
def main():
    run_sarsa()
    #run_q_learning()
    #run_double_q_learning()


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

На основе алгоритма SARDA произвели обучение для среды Cliff-Walking. Получили следующий график наград:

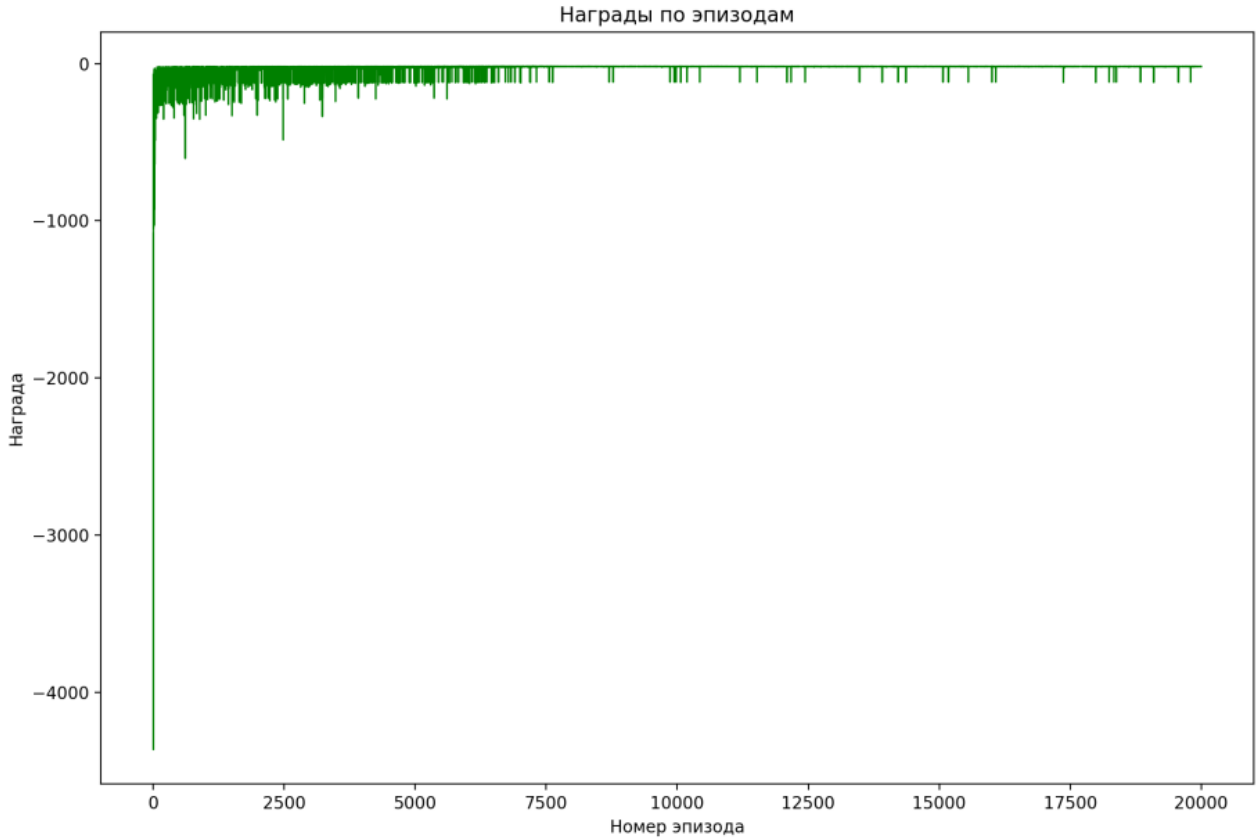

Рис. 2. График наград для алгоритма SARSA.

Как видим, общий штраф для модели постепенно снижается (т.е. награда растет). При попадании в обрыв получаем награду -100, при каждом шаге -1. В начале графика можно видеть, что было много попаданий в обрыв и награда была сильно отрицательной. Постепенно значение награды

приблизилось к нулю, что означает минимальный штраф за шаги до достижения цели.

При запуске обученной модели получили хороший результат, который сильно лучше, чем для policy iteration из предыдущей лабораторной. Агент достиг цели очень быстро и без попадания в обрыв (однако выбрал проход по верхним ячейкам).

2.2. Q-обучение

Алгоритм состоит из следующих действий:

1. Инициализация Q-функции произвольными значениями.
2. Выбор действия из состояния с использованием эпсилон-жадной стратегии ($\varepsilon > 0$) и переход в новое состояние.
3. Обновление Q предыдущего состояния по следующему правилу:

$$Q(s, \mathrm{a}) = Q(s, \mathrm{a}) + a(r + \gamma maxQ(s', \mathrm{a}') — Q(s, \mathrm{a}))$$

4. Повторение шагов 3 и 4 до достижения завершающего состояния.

Отличие от предыдущего алгоритма в том, что берем максимум maxQ(s', a').

Для алгоритма QLearning напишем класс QLearning_Agent:

```
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))):
            # 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)
```

Напишем фикцию для инициализации класса QLearning_Agent и выполнения эксперимента:

```
def run_q_learning():
    env = gym.make('CliffWalking-v0')
    agent = QLearning_Agent(env)
    agent.learn()
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)
```

На основе алгоритма Q-обучение произвели обучение для среды Cliff-Walking. Получили следующий график наград:

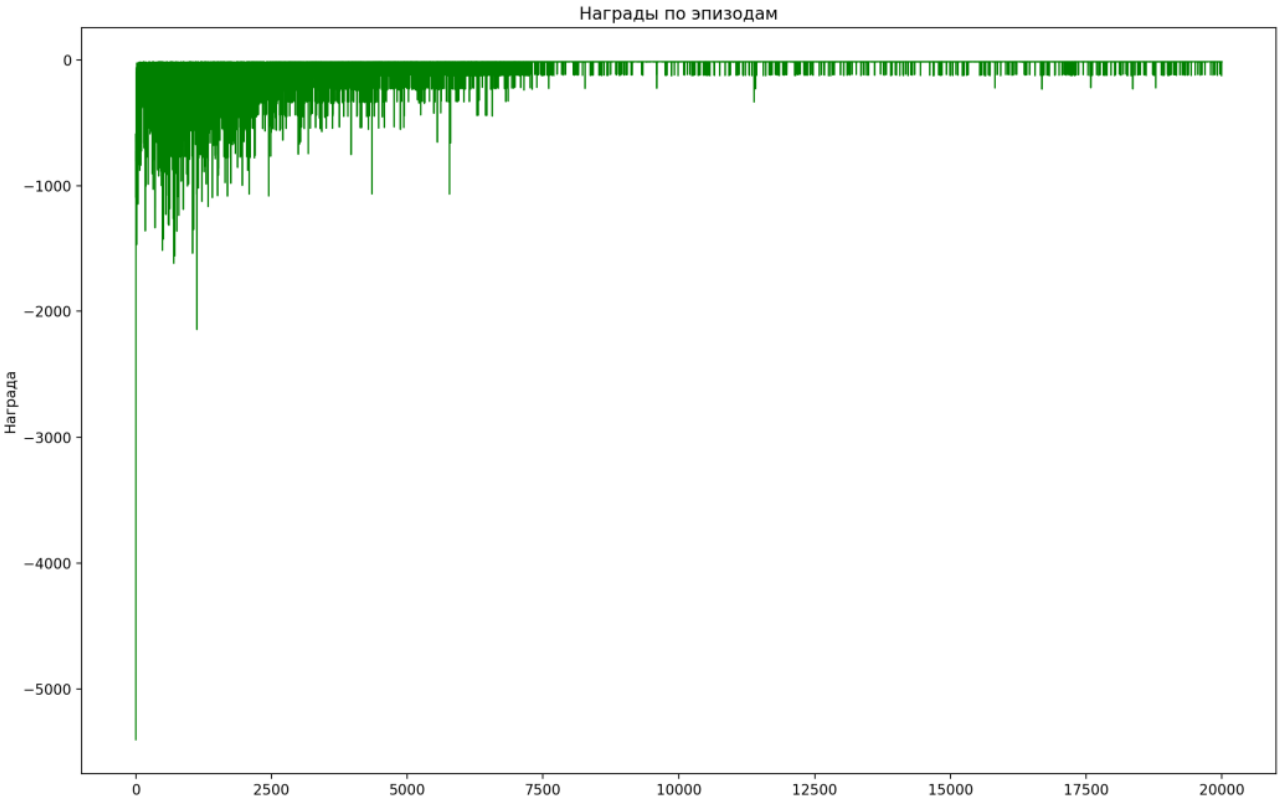

Рис. 3. График наград для алгоритма QLearning.

На этот раз результат оказался еще лучше, т.к. теперь путь агента до награды – самый короткий из возможных. Но при этом в начале агент делал много действий с большим штрафом (т.е. приближении награды к нулю на этот раз было за большее число итераций).

2.3. Двойное Q-обучение

В Q-обучении используется одна и та же выборка как для определения действия, доставляющего максимум, так и для оценки его ценности.

Разобьем все множество игр на два подмножества и будем использовать их для обучения двух независимых оценок, $Q1(a)$ и $Q2(a)$ истинной ценности q(a) для всех действий a. Тогда можно было бы взять одну оценку $Q1$ для определения доставляющего максимум действия.

$A* = argmax_a Q1(a)$, а другую, $Q2$, для оценки ценности этого действия:

$Q2(A*) = Q2(argmax_a Q1(a))$. Тогда эта оценка будет несмещенной в том смысле, что $\mathbb{E}[Q2(A*)] = q(A*)$. Этот процесс можно повторить, поменяв обе оценки ролями, и получить тем самым вторую несмещенную оценку Q1.

Хотя мы обучаем две оценки, при каждой игре обновляется только одна. Двойное обучение требует двойного объема памяти, но не увеличивает объем вычислений на каждом шаге. Обновление производится по правилу:

$$Q_1(S_t, A_t) = Q_1(S_t, A_t) + a[R_{t+1} + \gamma Q_2\big(S_{t+1}, argmax_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)]$$

Для алгоритма DoubleQLearning напишем класс DoubleQLearning_AgentAgent:

```
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('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.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)
```

Напишем фикцию для инициализации класса DoubleQLearning_Agent и выполнения эксперимента:

```
def run_double_q_learning():
    env = gym.make('CliffWalking-v0')
    agent = DoubleQLearning_Agent(env)
    agent.learn()
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)
```

На основе двойного Q-обучения произвели обучение для среды Cliff-Walking. Получили следующий график наград:

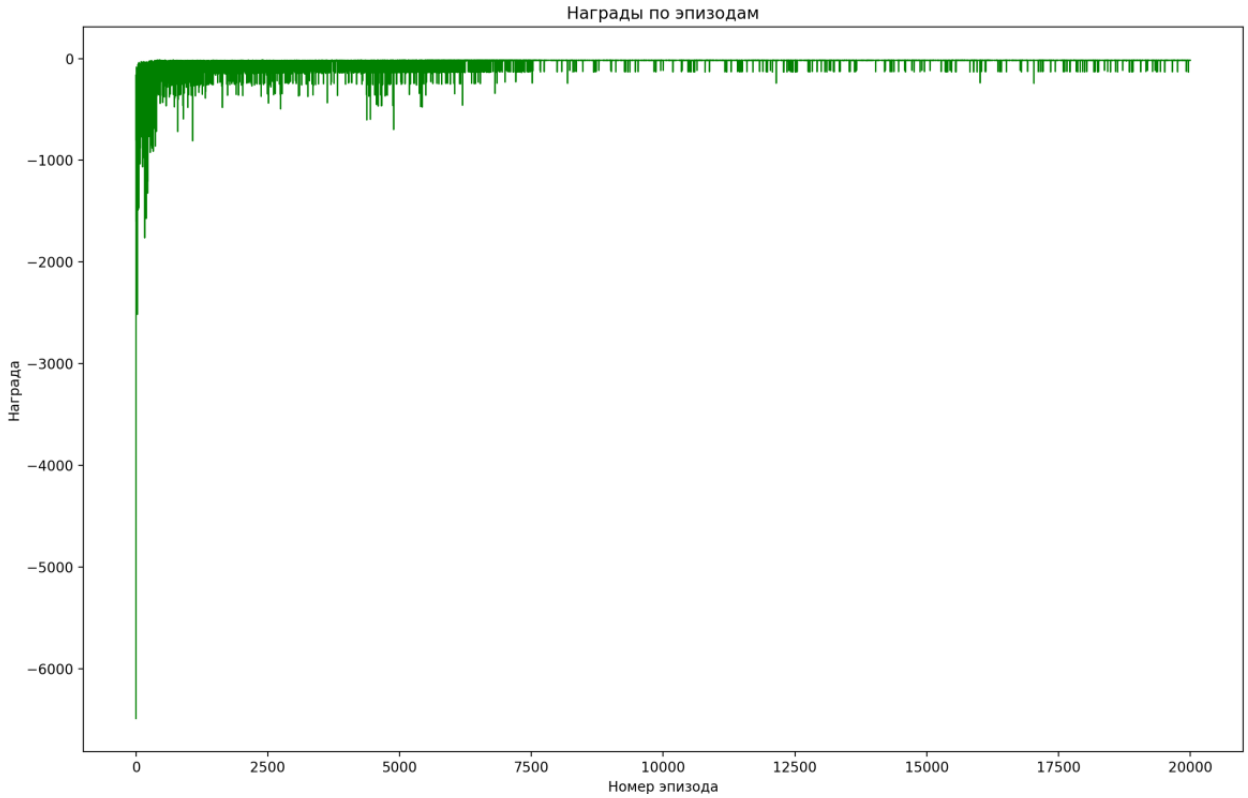

Рис. 3. График наград для алгоритма DoubleQLearning.

Обученный агент дошел до цели также хорошо, как и для Q-обучения, однако штраф уменьшился гораздо быстрее (т.е. быстрее произошло приближении награды к нулю).

Выводы

В ходе выполнения работы ознакомились с базовыми методами обучения с подкреплением на основе временных различий с помощью библиотеки Gym.