Защищено:
Гапанюк Ю.Е.

"__"_____2024 г.

Демонстрация:
Гапанюк Ю.Е.

"__"_____2024 г.

Отчет по лабораторной работе № 6 по курсу Методы машинного обучения

Тема работы: " Реализация алгоритма Policy Iteration "

11
(количество листов)
Вариант № **15**

ИСПОЛНИТЕЛЬ:

студент группы ИУ5-22М _____
(подпись)

Чиварзин А.Е.

"__"_____2024 г.

Москва, МГТУ - 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.

Ход выполнения работы

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 math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import torch
from torch import nn
from torchvision import transforms as T
from PIL import Image
import numpy as np
from pathlib import Path
from collections import deque
import random, datetime, os

# Gym is an OpenAI toolkit for RL
import gym
from gym.spaces import Box
from gym.wrappers import FrameStack

# Название среды
CONST_ENV_NAME = 'CartPole-v1'
# Использование GPU
CONST_DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Элемент ReplayMemory в форме именованного кортежа
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))

# Реализация техники Replay Memory
class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        '''
        Сохранение данных в ReplayMemory
        '''
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        '''
        Выборка случайных элементов размера batch_size
        '''
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)


class DQN_Model(nn.Module):

    def __init__(self, n_observations, n_actions):
        '''
        Инициализация топологии нейронной сети
        '''
        super(DQN_Model, self).__init__()
        c, h, w = n_observations

        if h != 84:
            raise ValueError(f"Expecting input height: 84, got: {h}")
        if w != 84:
            raise ValueError(f"Expecting input width: 84, got: {w}")

        # Основная модель
        self.online = self.__build_cnn(c, n_actions)
        # Вспомогательная модель, используется для стабилизации алгоритма
        # Обновление контролируется гиперпараметром TAU
```

```
        # Используется подход Double DQN
        self.target = self.__build_cnn(c, n_actions)
        self.target.load_state_dict(self.online.state_dict())

        # Q_target parameters are frozen.
        for p in self.target.parameters():
            p.requires_grad = False

    def forward(self, x):
        '''
        Прямой проход
        Вызывается для одного элемента, чтобы определить следующее действие
        Или для batch'a во время процедуры оптимизации
        '''
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)

    def __build_cnn(self, c, output_dim):
        return nn.Sequential(
            nn.Conv2d(in_channels=c, out_channels=32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(3136, 512),
            nn.ReLU(),
            nn.Linear(512, output_dim),
        )

class DQN_Agent:

    def __init__(self, env,
                 BATCH_SIZE = 128,
                 GAMMA = 0.99,
                 EPS_START = 0.9,
                 EPS_END = 0.05,
                 EPS_DECAY = 1000,
                 TAU = 0.005,
                 LR = 1e-4
                 ):
        # Среда
        self.env = env
        # Размерности Q-модели
        self.n_actions = env.action_space.n
        state, _ = self.env.reset()
        self.n_observations = len(state)
        # Коэффициенты
        self.BATCH_SIZE = BATCH_SIZE
        self.GAMMA = GAMMA
        self.EPS_START = EPS_START
        self.EPS_END = EPS_END
        self.EPS_DECAY = EPS_DECAY
        self.TAU = TAU
        self.LR = LR
        # Модели
        # Основная модель
        self.nets = DQN_Model(self.n_observations, self.n_actions).to(CONST_DEVICE)

        # Оптимизатор
        self.optimizer = optim.AdamW(self.policy_net.parameters(), lr=self.LR,
amsgrad=True)
        # Replay Memory
        self.memory = ReplayMemory(10000)
        # Количество шагов
        self.steps_done = 0
        # Длительность эпизодов
```

```
        self.episode_durations = []


    def select_action(self, state):
        '''
        Выбор действия
        '''
        sample = random.random()
        eps = self.EPS_END + (self.EPS_START - self.EPS_END) * \
            math.exp(-1. * self.steps_done / self.EPS_DECAY)
        self.steps_done += 1
        if sample > eps:
            with torch.no_grad():
                # Если вероятность больше eps
                # то выбирается действие, соответствующее максимальному Q-значению
                # t.max(1) возвращает максимальное значение колонки для каждой строки
                # [1] возвращает индекс максимального элемента
                return self.policy_net(state).max(1)[1].view(1, 1)
        else:
            # Если вероятность меньше eps
            # то выбирается случайное действие
            return torch.tensor([[self.env.action_space.sample()]], device=CONST_DEVICE,
dtype=torch.long)


    def plot_durations(self, show_result=False):
        plt.figure(1)
        durations_t = torch.tensor(self.episode_durations, dtype=torch.float)
        if show_result:
            plt.title('Результат')
        else:
            plt.clf()
            plt.title('Обучение...')
        plt.xlabel('Эпизод')
        plt.ylabel('Количество шагов в эпизоде')
        plt.plot(durations_t.numpy())
        plt.pause(0.001)  # пауза


    def optimize_model(self):
        '''
        Оптимизация модели
        '''
        if len(self.memory) < self.BATCH_SIZE:
            return
        transitions = self.memory.sample(self.BATCH_SIZE)
        # Транспонирование batch'a
        # (https://stackoverflow.com/a/19343/3343043)
        # Конвертация batch-массива из Transition
        # в Transition batch-массивов.
        batch = Transition(*zip(*transitions))

        # Вычисление маски нефинальных состояний и конкатенация элементов batch'a
        non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                                batch.next_state)), device=CONST_DEVICE,
dtype=torch.bool)
        non_final_next_states = torch.cat([s for s in batch.next_state
                                                    if s is not None])
        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)

        # Вычисление Q(s_t, a)
        state_action_values = self.policy_net(state_batch).gather(1, action_batch)

        # Вычисление V(s_{t+1}) для всех следующих состояний
        next_state_values = torch.zeros(self.BATCH_SIZE, device=CONST_DEVICE)
        with torch.no_grad():
```

```
            next_state_values[non_final_mask] =
self.target_net(non_final_next_states).max(1)[0]
        # Вычисление ожидаемых значений Q
        expected_state_action_values = (next_state_values * self.GAMMA) + reward_batch

        # Вычисление Huber loss
        criterion = nn.SmoothL1Loss()
        loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

        # Оптимизация модели
        self.optimizer.zero_grad()
        loss.backward()
        # gradient clipping
        torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 100)
        self.optimizer.step()


    def play_agent(self):
        '''
        Проигрывание сессии для обученного агента
        '''
        env2 = gym.make(CONST_ENV_NAME, render_mode='human')
        state = env2.reset()[0]
        state = torch.tensor(state, dtype=torch.float32,
device=CONST_DEVICE).unsqueeze(0)
        done = False
        res = []
        while not done:

            action = self.select_action(state)
            action = action.item()
            observation, reward, terminated, truncated, _ = env2.step(action)
            env2.render()

            res.append((action, reward))

            if terminated:
                next_state = None
            else:
                next_state = torch.tensor(observation, dtype=torch.float32,
device=CONST_DEVICE).unsqueeze(0)

            state = next_state
            if terminated or truncated:
                done = True

        print('Данные об эпизоде: ', res)


    def learn(self):
        '''
        Обучение агента
        '''
        if torch.cuda.is_available():
            num_episodes = 600
        else:
            num_episodes = 50

        for i_episode in range(num_episodes):
            # Инициализация среды
            state, info = self.env.reset()
            state = torch.tensor(state, dtype=torch.float32,
device=CONST_DEVICE).unsqueeze(0)
            for t in count():
                action = self.select_action(state)
                observation, reward, terminated, truncated, _ =
self.env.step(action.item())
                reward = torch.tensor([reward], device=CONST_DEVICE)
```

```
                done = terminated or truncated
                if terminated:
                    next_state = None
                else:
                    next_state = torch.tensor(observation, dtype=torch.float32,
device=CONST_DEVICE).unsqueeze(0)

                # Сохранение данных в Replay Memory
                self.memory.push(state, action, next_state, reward)

                # Переход к следующему состоянию
                state = next_state

                # Выполнение одного шага оптимизации модели
                self.optimize_model()

                # Обновление весов target-сети
                # θ′ ← τ θ + (1 − τ )θ′
                target_net_state_dict = self.target_net.state_dict()
                policy_net_state_dict = self.policy_net.state_dict()
                for key in policy_net_state_dict:
                    target_net_state_dict[key] = policy_net_state_dict[key]*self.TAU +
target_net_state_dict[key]*(1-self.TAU)
                self.target_net.load_state_dict(target_net_state_dict)

                if done:
                    self.episode_durations.append(t + 1)
                    self.plot_durations()
                    break


class SkipFrame(gym.Wrapper):
    def __init__(self, env, skip):
        """Return only every `skip`-th frame"""
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        """Repeat action, and sum reward"""
        total_reward = 0.0
        for i in range(self._skip):
            # Accumulate reward and repeat the same action
            obs, reward, done, trunk, info = self.env.step(action)
            total_reward += reward
            if done:
                break
        return obs, total_reward, done, trunk, info

class GrayScaleObservation(gym.ObservationWrapper):
    def __init__(self, env):
        super().__init__(env)
        obs_shape = self.observation_space.shape[:2]
        self.observation_space = Box(low=0, high=255, shape=obs_shape, dtype=np.uint8)

    def permute_orientation(self, observation):
        # permute [H, W, C] array to [C, H, W] tensor
        observation = np.transpose(observation, (2, 0, 1))
        observation = torch.tensor(observation.copy(), dtype=torch.float)
        return observation

    def observation(self, observation):
        observation = self.permute_orientation(observation)
        transform = T.Grayscale()
        observation = transform(observation)
        return observation

class ResizeObservation(gym.ObservationWrapper):
    def __init__(self, env, shape):
        super().__init__(env)
```

```
        if isinstance(shape, int):
            self.shape = (shape, shape)
        else:
            self.shape = tuple(shape)

        obs_shape = self.shape + self.observation_space.shape[2:]
        self.observation_space = Box(low=0, high=255, shape=obs_shape, dtype=np.uint8)

    def observation(self, observation):
        transforms = T.Compose(
            [T.Resize(self.shape, antialias=True), T.Normalize(0, 255)]
        )
        observation = transforms(observation).squeeze(0)
        return observation

def main():
    env = gym.make(CONST_ENV_NAME)
    agent = DQN_Agent(env)
    agent.learn()
    agent.play_agent()

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

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

Результаты выполнения:

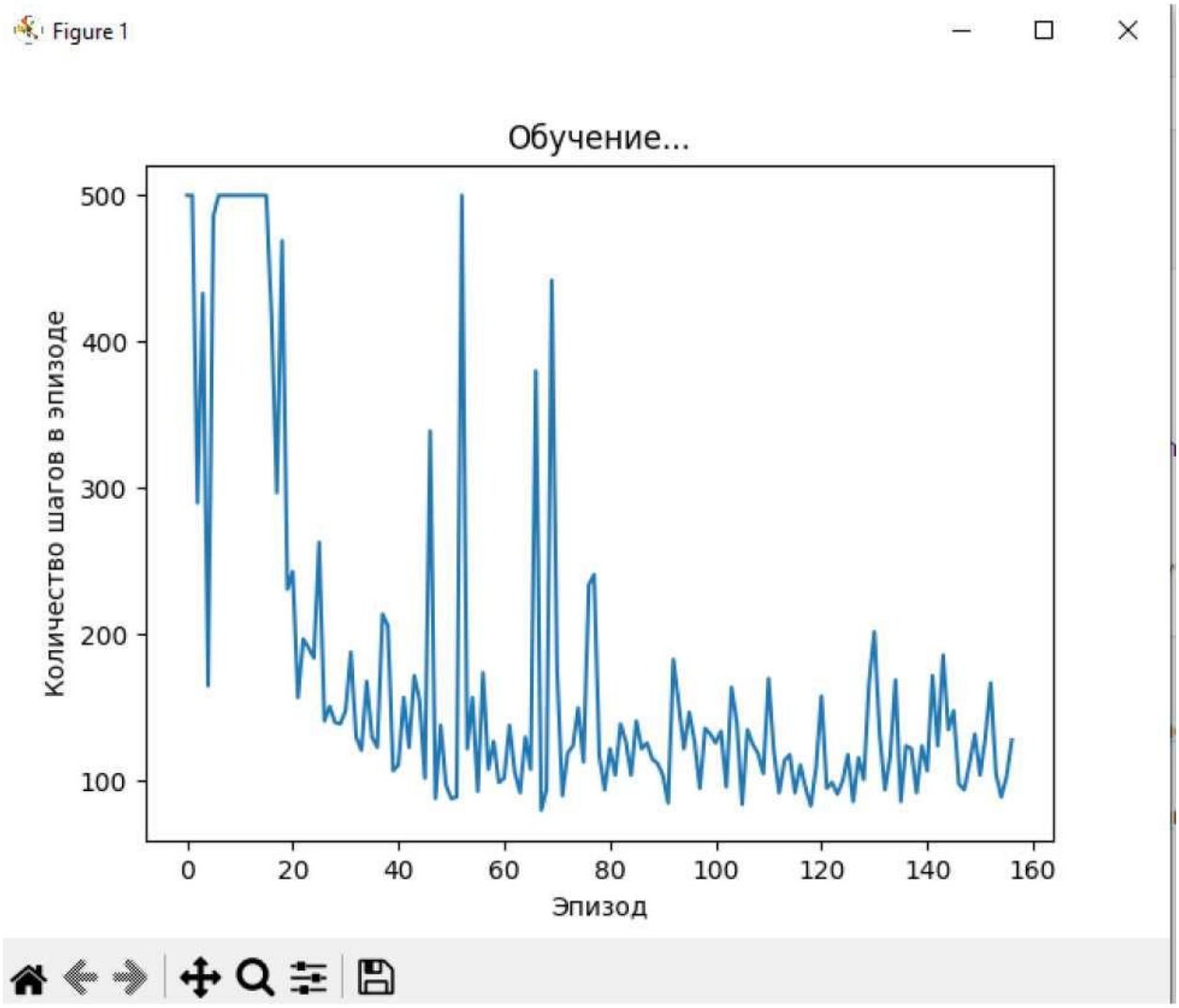

Рис. 1 - Начало обучнения.

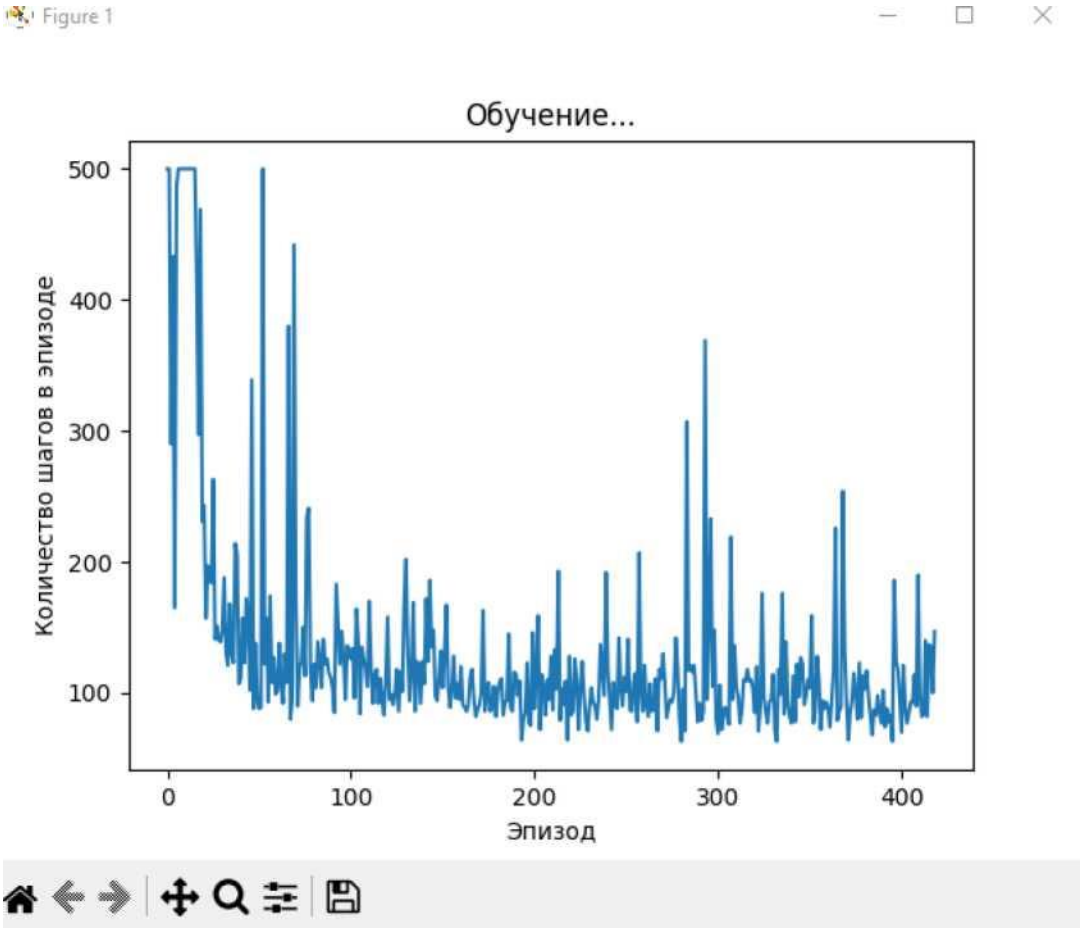

Рис. 2 - прогресс обучения.

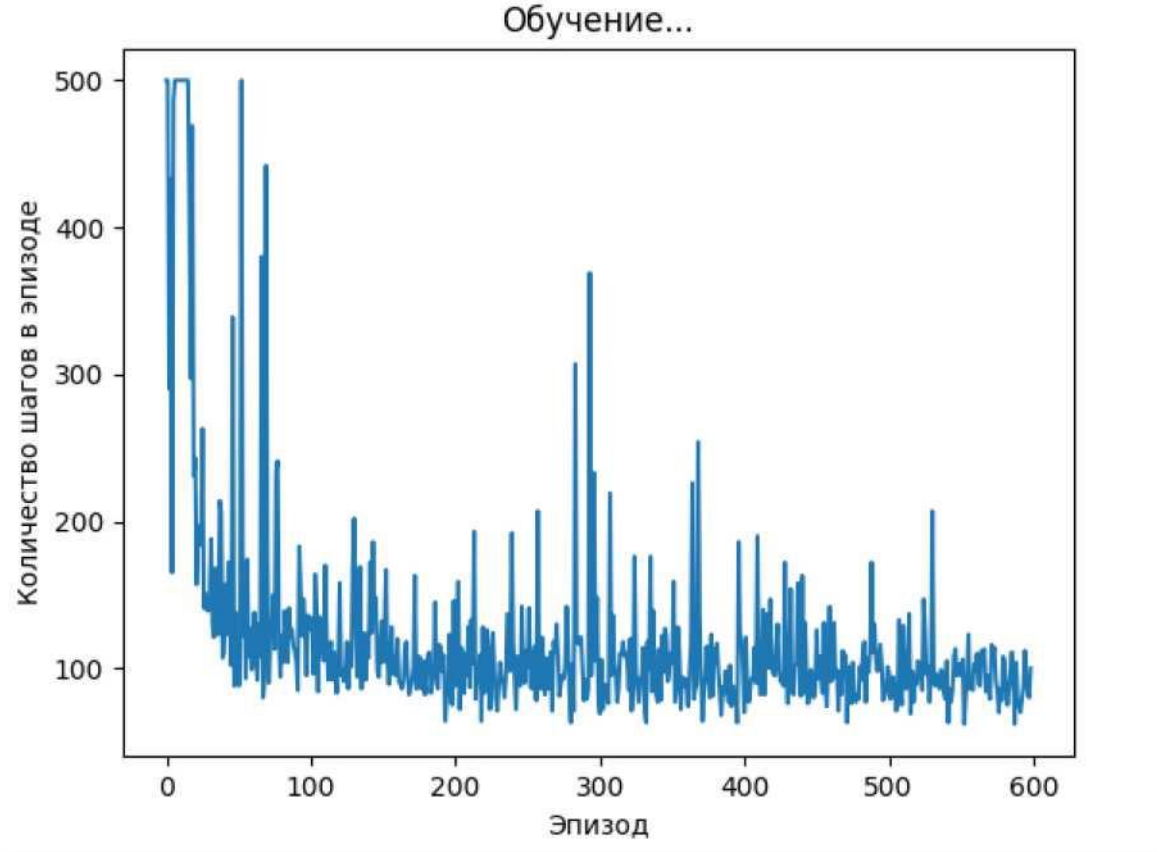

Рис. 3 - Конец обучения.

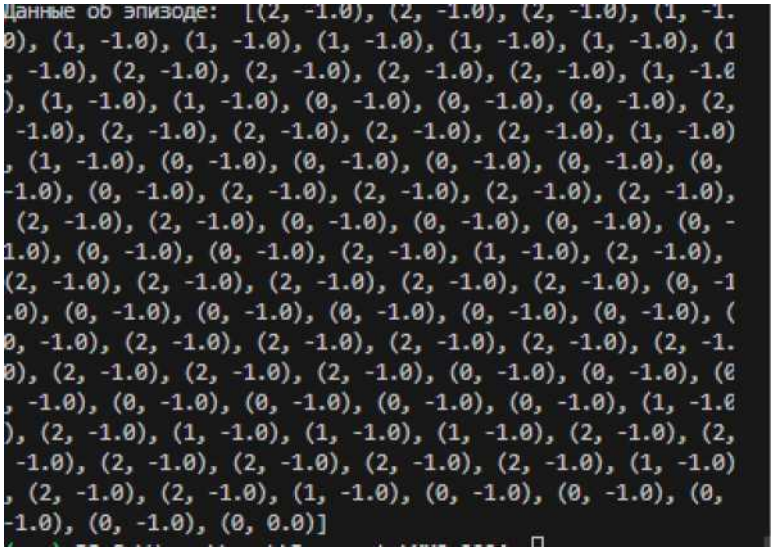

Рис. 4 - Финальный эпизод.

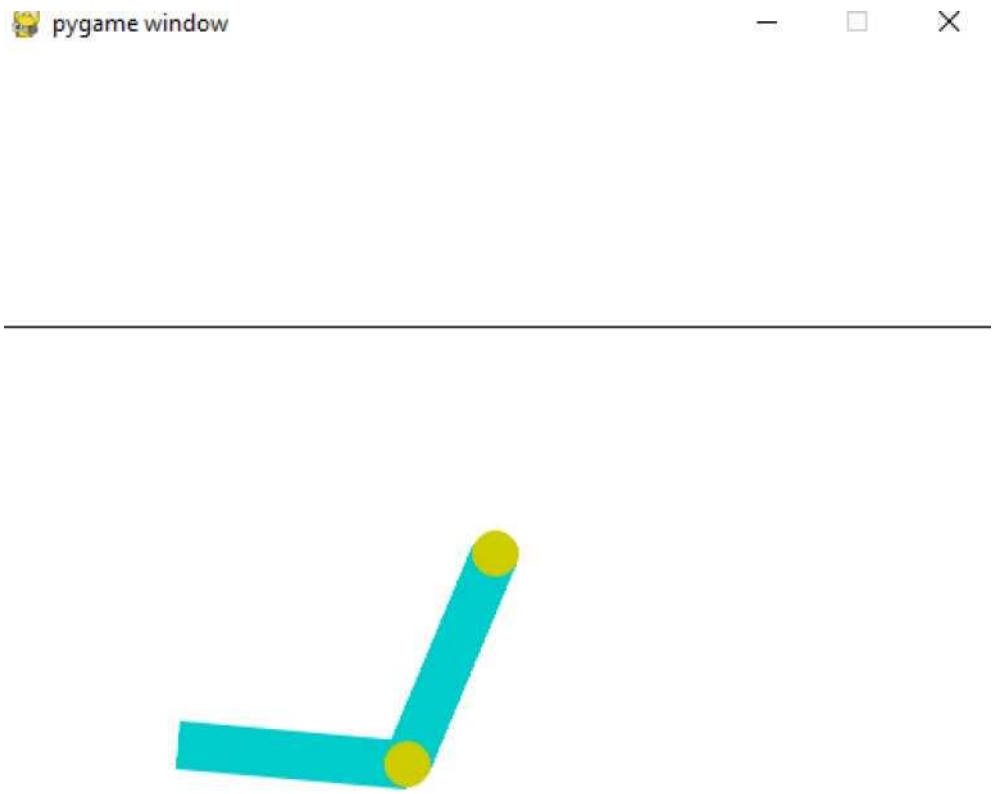

Рис. 5 - Пример движения актёра.

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.