МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Обучение с подкреплением»

Тема: Реализация DQN для среды CartPole-v1

Студент гр. 0306 _____ Сологуб Н.А.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург

2025

Цель работы.

Изучить и реализовать алгоритм DQN для среды CartPole-v1. Исследовать скорость обучения в зависимости от различных параметров: архитектура сети, значения gamma и epsilon_decay.

Постановка задачи.

1) Реализовать DQN
2) Изучить зависимость скорости обучения от архитектуры используемой нейронной сети
3) Изучить зависимость скорости обучения от параметров gamma и epsilon_decay
4) Изучить зависимость скорости обучения от изначального значения epsilon

Выполнение задач.

1. **Реализация DQN**

DQN был реализован под среду CartPole. Среда CartPole представляет из себя тележку с прикреплённой к ней шестом. Задача состоит в том, чтобы как можно дольше удерживать равновесие шеста путём перемещения тележки. Состояние среды описывается 4 параметрами: позиция тележки, скорость тележки, угол шеста, угловая скорость шеста. Тележку можно перемещать влево или вправо (доступные действия). В данном случае рассматривается среда CartPole-v1, ограниченная 500 шагами (то есть среда завершит выполнение при максимальном прохождении 500 шагов или если шест упадёт). Чем дольше живёт агент (удерживает шест в равновесии), тем больше награды он получает. Реализация DQN задействует оптимизации для более высокой обучаемости, позаимствованные из документации pytorch: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Q-функцией, которая позволяет оценить и выбрать для агента наилучшее действие выступает нейронная сеть, представленная на рис. 1.

```python
class QNetwork(nn.Module):
    def __init__(self, obs_size=4, n_actions=2, hidden_layers_sizes=[128, 128]):
        super(QNetwork, self).__init__()
        layers = []
        prev_size = obs_size
        for hidden_size in hidden_layers_sizes:
            layers.extend([nn.Linear(prev_size, hidden_size), nn.ReLU()])
            prev_size = hidden_size
        out_layer = nn.Linear(prev_size, n_actions)
        layers.append(out_layer)
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

Рисунок 1 — архитектура нейронной сети

Архитектура представляет из себя 3 линейных слоя: один входной с параметрами среды, промежуточный линейный слой и выходной слой с действиями. Также сеть содержит функции активации в виде ReLU.

Нейронная сеть используется для изначальной policy (обучается быстрее) и целевой target_policy (обучается медленнее, но обеспечивает обучение модели и обновление весов) сети.

Также DQN задействует ReplayBuffer – буфер попыток с данными (состояние, действие, награда, следующее состояние, завершён ли эпизод). Данные случайным образом берутся из буфера на размер батча и используются в последующих попытках при обучении, если в буфере лежит достаточно данных.

Алгоритм обучения в рамках одного шага строится следующим образом:

- если в буфере достаточно данных о прошлом опыте агента, то данные извлекаются из буфера
- данные передаются в изначальную сеть policy для оценки награды
- с помощью целевой сети target_policy идёт предсказание возможной будущей награды. Выбирается максимально оценённая награда

- Вычисляются целевые Q-значения, на основе которых будет приниматься решение о выполнении того или иного действиями
- Вычисление функции потерь
- Обратное распространения ошибки и обновление весов для уменьшения функции потерь

Действия выполняются в рамках одного шага. Для обучения было выбрано 700 эпизодов.

Основные параметры DQN и среды представлены на рис. 2

```
# Параметры среды и устройства
parser = argparse.ArgumentParser(description='DQN for CartPole-v1')
parser.add_argument('--gamma', type=float, default=0.99, help='discount factor (default: 0.99)')
parser.add_argument('--epsilon-start', type=float, default=1.0, help='initial epsilon (default: 1.0)')
parser.add_argument('--epsilon-min', type=float, default=0.05, help='final epsilon (default: 0.05)')
parser.add_argument('--epsilon-decay', type=float, default=500, help='epsilon decay rate (default: 500)')
parser.add_argument('--tau', type=float, default=0.005, help='soft update rate (default: 0.005)')
parser.add_argument('--hidden-layers-sizes', type=str, default='128,128',
                    help='comma-separated hidden layer sizes (default: 128,128)')
parser.add_argument('--seed', type=int, default=543, help='random seed (default: 543)')
parser.add_argument('--render', action='store_true', help='render the environment')
parser.add_argument('--log-interval', type=int, default=10, help='interval between training status logs (default: 10)')
BATCH_SIZE = 128  # объём данных, которые будем брать из буфера
lr=1e-4 # скорость обучения
num_episodes = 700  # количество эпизодов для обучения
num_steps = 500  # максимальное количество шагов в рамках одного эпизода
args = parser.parse_args()
```

Рисунок 2 — основные параметры DQN и среды

2. Зависимость скорости обучения от архитектуры используемой нейронной сети

Исследуем зависимость скорости обучения от архитектуры используемой нейронной сети. В рамках эскперимента обучим модель на архитектурах сети при размерах слоёв [4, X] → [X,X] → [X,2], где X = 64,128,256. Был построен график количества шагов от эпизода для каждой архитектуры сети. Результат представлен на рис. 3.

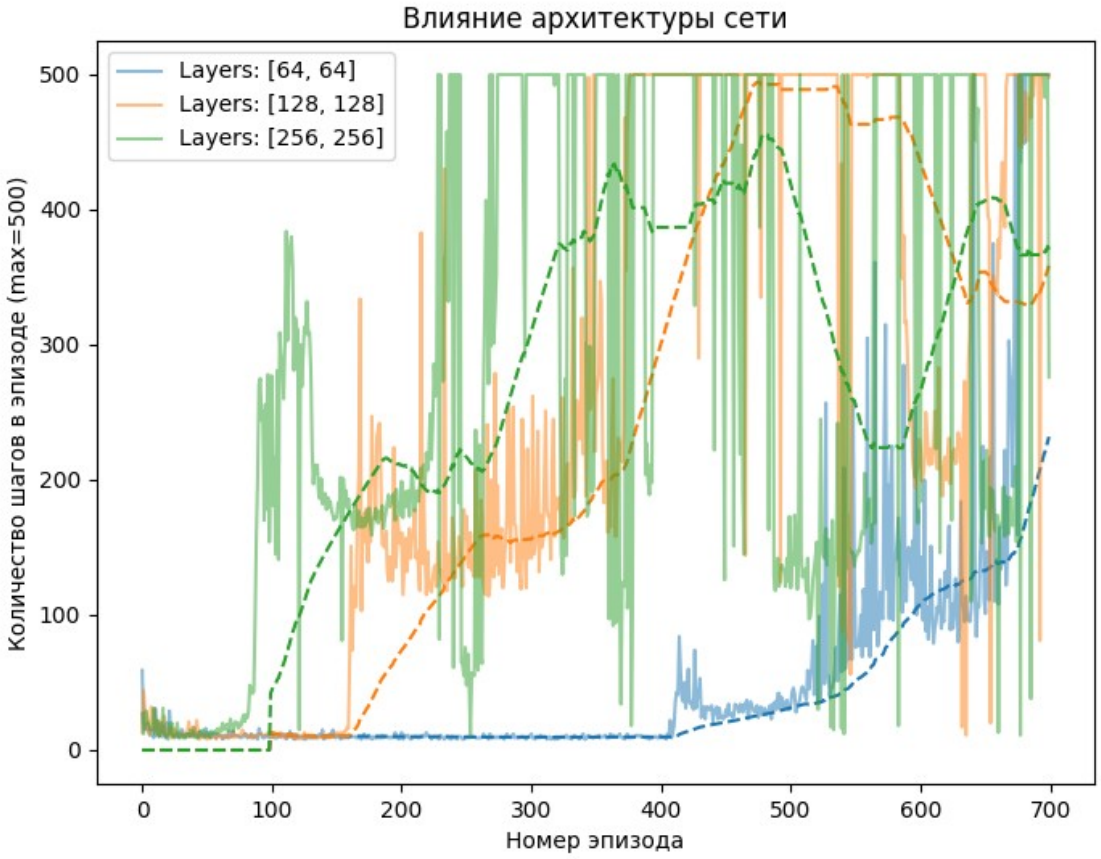

Рисунок 3 — график зависимости от архитектуры нейронной сети

Из рисунка можно сказать, что увеличение архитектуры сети повышает её обучаемость. Наиболее большая архитектура обучается быстрее ближе к 300 эпизодам, но падает в эффективности ближе к 500 эпизодам. Архитектура [128, 128] достигает наивысших результатов.

3. Зависимость скорости обучения от параметров gamma и epsilon_decay

Исследуем зависимость скорости обучения от параметра gamma. В рамках эскперимента обучим модель при параметре gamma = 0.9, 0.95, 0.99, 1.0. Был построен график количества шагов от эпизода для каждого параметра gamma. Результат представлен на рис. 4.

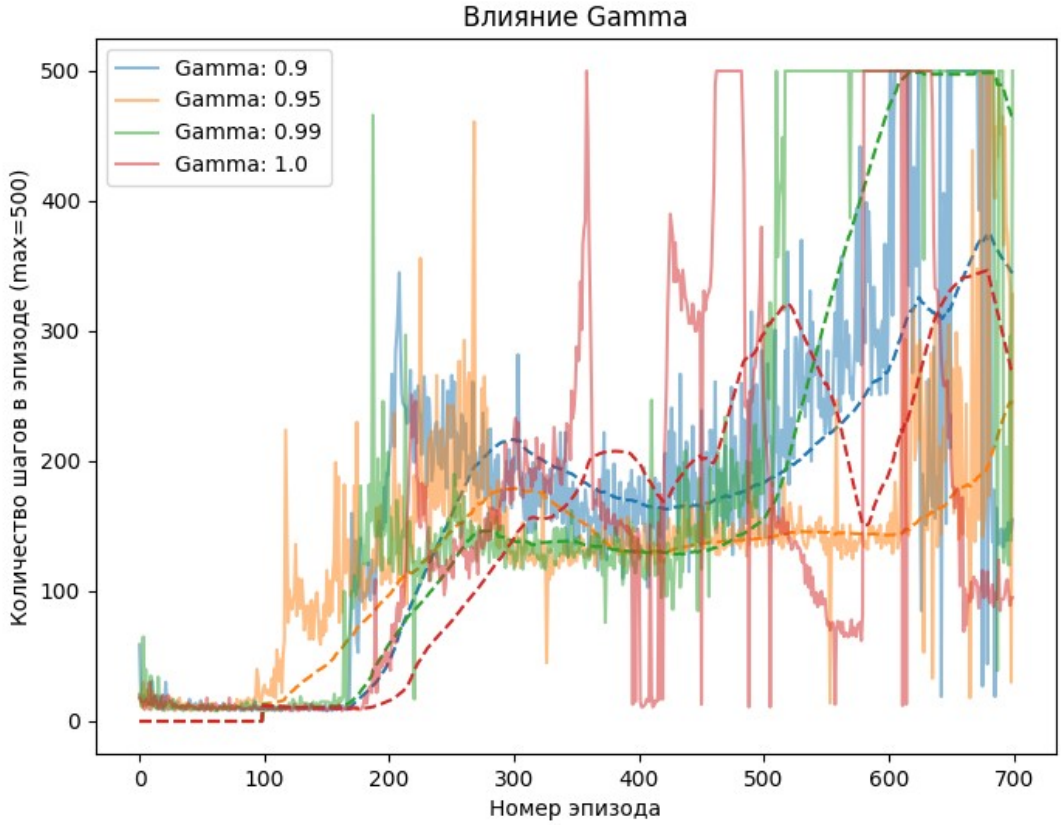

Рисунок 4 — график зависимости от gamma

Из рисунка видно, что наиболее высокий и стабильный результат достигается при gamma = 0.99. Чем ближе gamma к нулю, тем для модели более ценна краткосрочная награда здесь и сейчас. Чем ближе gamma к единице, тем более модель стремиться к долгосрочной выгоде в будущем. Видно, что награда в будущем более положительно сказывается на обучаемости модели.

Исследуем зависимость скорости обучения от параметра epsilon_decay. В рамках эскперимента обучим модель при параметре epsilon_decay = 200, 500, 1000, 2000. Был построен график количества шагов от эпизода для каждого параметра epsilon_decay. Результат представлен на рис. 5.

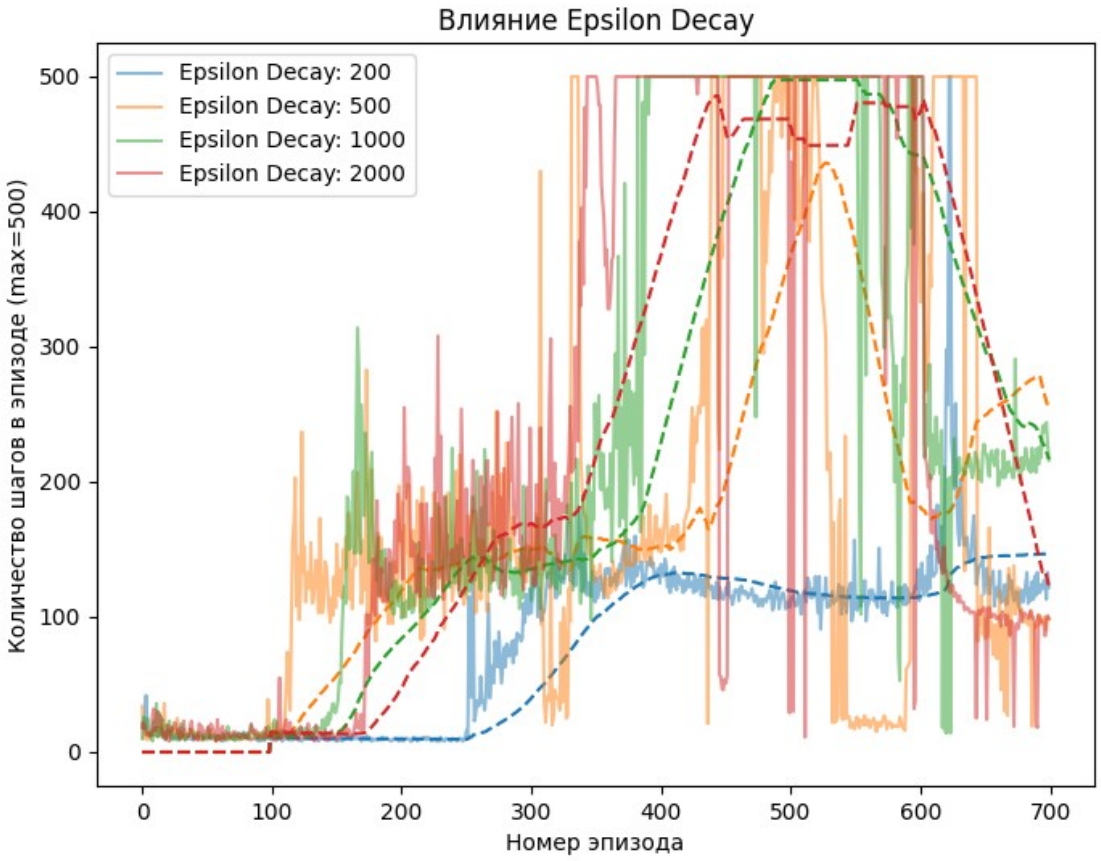

Рисунок 5 — график зависимости от epsilon_decay

Из рисунка видно, что при epsilon_decay = 1000 и 2000 достигаются наиболее быстрая скорость обучения ближе к 400-500 эпизодам, что объяснимо тем, что с ростом epsilon_decay быстрее уменьшается шанс того, что модель будет делать действие случайно. То есть наиболее чаще будет делать выбор, основываясь на предсказаниях Q-функции.

4. Зависимость скорости обучения от изначального значения epsilon

Исследуем зависимость скорости обучения от начального параметра epsilon. В рамках эскперимента обучим модель при параметре epsilon = 0.9, 0.95, 0.99, 1.0. Был построен график количества шагов от эпизода для каждого параметра epsilon. Результат представлен на рис. 6.

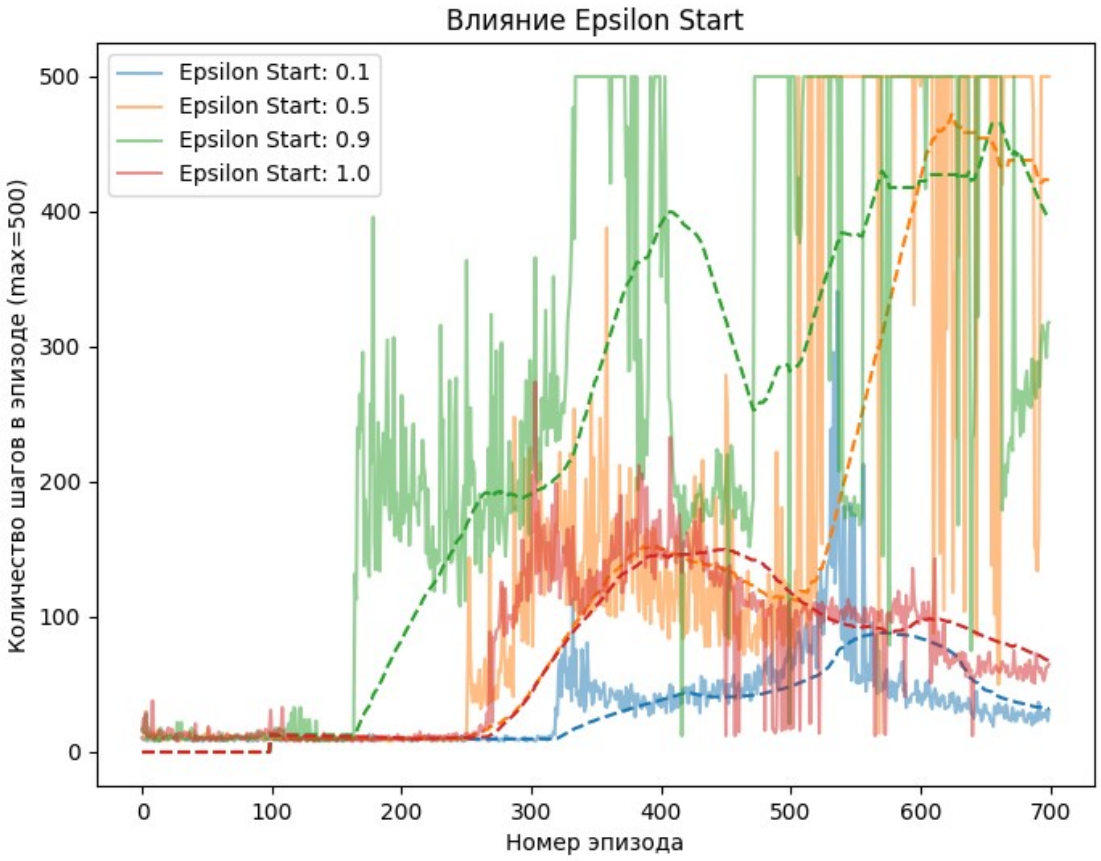

Рисунок 6 — график зависимости от epsilon

Из рисунка видно, что наиболее лучший результат по скорости обучения даёт значение epsilon = 0.5 и 0.9, что позволяет модели вначале делать наиболее чаще случайные выборы.

Разработанный код представлен в приложении А.

Выводы.

В рамках лабораторной работы был изучен алгоритм глубокого обучения DQN на среде CartPole-v1. Была исследована зависимость скорости обучения модели от таких параметров как: архитектура нейронной сети, gamma, epsilon_decay и начальное epsilon. Были построены графики по результатам которых можно сделать вывод о том, что увеличение архитектуры положительно влияет на скорость обучения; gamma близкое к 1 положительно сказывается на обучаемость модели; более высокое epsilon_decay позволяет модели делать чаще не случайный выбор, что повышает её скорость обучения;

среднее значение epsilon позволяет модели в равной степени изучать новые варианты и делать решения на основе предсказаний, что повышает скорость обучения.

ПРИЛОЖЕНИЕ А

Код выполнения лабораторной работы

```
import math
from collections import deque
import gymnasium as gym
import argparse
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
import matplotlib.pyplot as plt
from tqdm import tqdm

# Параметры среды и устройства
parser = argparse.ArgumentParser(description='DQN for CartPole-
v1')
parser.add_argument('--gamma', type=float, default=0.99,
help='discount factor (default: 0.99)')
parser.add_argument('--epsilon-start', type=float, default=1.0,
help='initial epsilon (default: 1.0)')
parser.add_argument('--epsilon-min', type=float, default=0.05,
help='final epsilon (default: 0.05)')
parser.add_argument('--epsilon-decay', type=float, default=500,
help='epsilon decay rate (default: 500)')
parser.add_argument('--tau', type=float, default=0.005, help='soft
update rate (default: 0.005)')
parser.add_argument('--hidden-layers-sizes', type=str,
default='128,128',
                    help='comma-separated hidden layer sizes
(default: 128,128)')
parser.add_argument('--seed', type=int, default=543, help='random
seed (default: 543)')
parser.add_argument('--render', action='store_true', help='render
the environment')
parser.add_argument('--log-interval', type=int, default=10,
help='interval between training status logs (default: 10)')
BATCH_SIZE = 128  # объём данных, которые будем брать из буфера
lr=1e-4 # скорость обучения
num_episodes = 700  # количество эпизодов для обучения
num_steps = 500  # максимальное количество шагов в рамках одного
эпизода
args = parser.parse_args()
```

```
device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
# Инициализация окружения
env = gym.make('CartPole-v1', render_mode='rgb_array')
env.reset(seed=args.seed)
torch.manual_seed(args.seed)
np.random.seed(args.seed)
random.seed(args.seed)


# Класс буфера для хранения сэмплов
class ReplayBuffer:
    def __init__(self, capacity=10000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state,
done))

    # Взять данные для обучения на размер батча
    def sample(self, batch_size=64):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*batch)
        return (
            torch.tensor(np.array(state), dtype=torch.float32),
            torch.tensor(action, dtype=torch.int64),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(np.array(next_state),
dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32),
        )

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


# Нейросеть для DQN
class QNetwork(nn.Module):
    def __init__(self, obs_size=4, n_actions=2,
hidden_layers_sizes=[128, 128]):
        super(QNetwork, self).__init__()
        layers = []
        prev_size = obs_size
        for hidden_size in hidden_layers_sizes:
```

```
            layers.extend([nn.Linear(prev_size, hidden_size),
nn.ReLU()])
            prev_size = hidden_size
        out_layer = nn.Linear(prev_size, n_actions)
        layers.append(out_layer)
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)


# Берём размеры слоёв
hidden_layers_sizes = [int(x) for x in
args.hidden_layers_sizes.split(',')]
loss_history = []  # показатель потерь

# Выбор действия в зависимости от нынешнего состояния
"""
    чем меньше ε, тем более чаще будем делать осознанный выбор,
    а не случайный.
    С вероятностью (1 - ε) делаем выбор на основе наибольшего
значения Q.
"""
steps_done = 0


def select_action(state):
    global steps_done
    eps_threshold = args.epsilon_min + (args.epsilon_start -
args.epsilon_min) * math.exp(
        -1.0 * steps_done / args.epsilon_decay
    ) # экспотенциальное уменьшение ε
    steps_done += 1

    if random.random() < eps_threshold:
        return random.randint(0, 1)
    with torch.no_grad():
        state = torch.tensor(state, dtype=torch.float32,
device=device)
        q_values = policy(state)  # получаем Q значения из
политики на основе нашего state
    return torch.argmax(q_values).item()  # возвращаем
максимальное значение Q
```

```
# Обучение на одном батче
def train():
    if len(buffer) < BATCH_SIZE:  # если количество элементов не
хватает в буфере
        return 0
    state, action, reward, next_state, done =
buffer.sample(BATCH_SIZE)  # достаём данные из буфера
    state, action, reward, next_state, done = state.to(device),
action.to(device), reward.to(device), next_state.to(
        device), done.to(device)  # привязываем к устройству

    q_values = policy(state).gather(1,
action.unsqueeze(1)).squeeze(1)  # достаём q значения из
изначальной политики
    next_q_values = target_policy(next_state).max(1)[0]  # достаём
q значения для целевой политики
    targets = reward + args.gamma * next_q_values * (
            1 - done)  # вычисляем target: r + γ * (1 - done) *
max Q(next_state, a)
    loss = nn.SmoothL1Loss()(q_values, targets)  # вычисляем
функцию потерь

    optimizer.zero_grad()  # сбрасываем градиенты
    loss.backward()  # вычисляем градиенты по функции потерь
    torch.nn.utils.clip_grad_value_(policy.parameters(), 100) #
обрезаем градиенты, во избежания взрывов
    optimizer.step()  # обновляем веса сети
    return loss.item()


# Построение графиков
def plot_experiment(episodes_durations, episodes_legends, title,
filename):
    plt.figure(figsize=(8, 6))
    for i, episode_durations in enumerate(episodes_durations):
        durations_t = torch.tensor(episode_durations,
dtype=torch.float)
        plt.plot(
            durations_t.numpy(), alpha=0.5,
label=episodes_legends[i], color=f"C{i}"
        )
        if len(durations_t) >= 100:
            means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
            means = torch.cat((torch.zeros(99), means))
            plt.plot(means.numpy(), color=f"C{i}", linestyle='--')
```

```
    plt.title(title)
    plt.xlabel("Номер эпизода")
    plt.ylabel("Количество шагов в эпизоде (max=500)")
    plt.legend(loc="best")
    plt.savefig(filename)
    plt.close()


# Основной цикл
def train_dqn(hidden_layers_sizes):
    global steps_done, policy, target_policy, optimizer, buffer
    # Инициализация модели
    policy =
QNetwork(hidden_layers_sizes=hidden_layers_sizes).to(device)  #
исходная сеть
    target_policy =
QNetwork(hidden_layers_sizes=hidden_layers_sizes).to(device)  #
целевая сеть
    target_policy.load_state_dict(policy.state_dict())  #
синхронизируем сети для сходимости
    optimizer = optim.Adam(policy.parameters(), lr=lr)  #
инициализируем оптимизатор для обновления весов сети
    buffer = ReplayBuffer()  # инициализируем буфер
    steps_done = 0
    duration_history = [] # время жизни агента в рамках одного
эпизода (количество шагов)

    for episode in tqdm(range(num_episodes)):
        state, _ = env.reset()
        ep_reward = 0  # награда за эпизод
        ep_steps = 0

        for t in range(num_steps):  # проходим максимум 500 шагов
            action = select_action(state)  # выбираем действие
            next_state, reward, done, truncated, _ =
env.step(action)  # передаём действие в среду
            buffer.push(state, action, reward, next_state, done)
# добавляем данные в буфер
            state = next_state  # обновление state
            ep_reward += reward  # считаем награду
            ep_steps = t + 1  # обновляем кол-во шагов

            loss = train()
            loss_history.append(loss)
```

```
            target_state_dict = target_policy.state_dict()
            policy_state_dict = policy.state_dict()
            for key in policy_state_dict:
                target_state_dict[key] = policy_state_dict[key] *
args.tau + target_state_dict[key] * (1 - args.tau)  # Мягкое
обновление целевой сети
            target_policy.load_state_dict(target_state_dict)

            if done or truncated:  # если симуляция завершилась
раньше, то заканчиваем (шест упал)
                break
        duration_history.append(ep_steps)  # Сохраняем количество
шагов в эпизоде

        # Логирование
        if episode % args.log_interval == 0:
            print(
                f"Episode: {episode + 1}: Reward: {ep_reward};")
    return duration_history
def experiment_network_sizes():
    network_configs = [
        [64, 64],
        [128, 128],
        [256, 256]
    ]
    episodes_durations = []
    episodes_legends = [f"Layers: {config}" for config in
network_configs]

    for config in network_configs:
        print(f"\nExperiment with network architecture: {config}")
        durations = train_dqn(config)
        episodes_durations.append(durations)

    plot_experiment(episodes_durations, episodes_legends, "Влияние
архитектуры сети", "network_sizes.png")

def experiment_gamma():
    gamma_values = [0.9, 0.95, 0.99, 1.0]
    episodes_durations = []
    episodes_legends = [f"Gamma: {gamma}" for gamma in
gamma_values]
    fixed_layers = [128, 128]

    for gamma in gamma_values:
```

```
        print(f"\nExperiment with gamma: {gamma}")
        args.gamma = gamma
        durations = train_dqn(fixed_layers)
        episodes_durations.append(durations)

    plot_experiment(episodes_durations, episodes_legends, "Влияние
Gamma", "gamma_values.png")

def experiment_epsilon_decay():
    epsilon_decay_values = [200, 500, 1000, 2000]
    episodes_durations = []
    episodes_legends = [f"Epsilon Decay: {decay}" for decay in
epsilon_decay_values]
    fixed_layers = [128, 128]

    for decay in epsilon_decay_values:
        print(f"\nExperiment with epsilon_decay: {decay}")
        args.epsilon_decay = decay
        durations = train_dqn(fixed_layers)
        episodes_durations.append(durations)

    plot_experiment(episodes_durations, episodes_legends, "Влияние
Epsilon Decay", "epsilon_decay_values.png")

def experiment_epsilon_start():
    epsilon_start_values = [0.1, 0.5, 0.9, 1.0]
    episodes_durations = []
    episodes_legends = [f"Epsilon Start: {start}" for start in
epsilon_start_values]
    fixed_layers = [128, 128]

    for start in epsilon_start_values:
        print(f"\nExperiment with epsilon_start: {start}")
        args.epsilon_start = start
        durations = train_dqn(fixed_layers)
        episodes_durations.append(durations)

    plot_experiment(episodes_durations, episodes_legends, "Влияние
Epsilon Start", "epsilon_start_values.png")

if __name__ == '__main__':
    print("Starting Experiment 1: Different Network
Architectures")
    experiment_network_sizes()
```

```
    print("\nStarting Experiment 2: Different Gamma Values")
    experiment_gamma()

    print("\nStarting Experiment 3: Different Epsilon Decay
Values")
    experiment_epsilon_decay()

    print("\nStarting Experiment 4: Different Epsilon Start
Values")
    experiment_epsilon_start()

    env.close()
```