МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Обучение с подкреплением»
Тема: Реализация DQN для среды CartPole-v1

Студент гр. 0310 _____ Панкина В. К.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург
2025

Цель работы.

Изучить основы работы алгоритма Deep Q-Learning (DQN) и применить его для решения задачи управления маятником в классическом окружении CartPole-v1 из библиотеки OpenAI Gym. Ознакомиться с влиянием архитектуры нейросети и гиперпараметров на процесс обучения агента с подкреплением.

Постановка задачи.

1. Реализовать DQN-агента для среды CartPole-v1, используя библиотеку PyTorch.
2. Провести модификацию базовой архитектуры Q-сети. Проанализировать, как это влияет на стабильность и скорость обучения.
3. Исследовать влияние изменения параметров *gamma* и *epsilon_decay* на поведение агента.
4. Оценить как изначальное значение *epsilon* влияет на скорость обучения.

Выполнение задач.

1. Реализация DQN.

Был реализован агент DQN, основанный на библиотеке PyTorch, для решения задачи управления маятником в окружении CartPole-v1 из OpenAI Gym.

Агент обучается на основе алгоритма Deep Q-Learning с использованием буфера воспроизведения, в котором сохраняются опыты взаимодействия со средой. На каждом шаге агент выбирает действия с использованием ε-жадной стратегии, что позволяет сбалансировать исследование и использование накопленного опыта.

Процесс обучения продолжается в течение заданного количества эпизодов. По результатам каждого эпизода фиксируются значения суммарного вознаграждения и функции потерь, что позволяет визуализировать динамику обучения агента и проводить сравнение различных конфигураций гиперпараметров и архитектур нейросети.

Ниже представлены стандартные параметры, использовавшиеся при обучении агентов:

- Количество эпизодов: 300
- Размер батча: 64
- Коэффициент дисконтирования (γ): 0.99
- Начальное значение ε: 1.0
- Скорость уменьшения ε: 0.955
- Минимальное значение ε: 0.01
- Объём буфера воспроизведения: 1000 переходов
- Оптимизатор: Adam, скорость обучения 1e-3

На таких исходных данных были получены результаты, представленные на рисунке 1 и рисунке 2.

Рисунок 1 демонстрирует изменение суммарного вознаграждения по эпизодам. Этот график позволяет оценить скорость обучения агента и степень стабилизации поведения с течением времени.

На начальных этапах (эпизоды 0–50) агент демонстрирует низкое вознаграждение, постепенно обучаясь взаимодействию со средой и вырабатывая базовые стратегии. Примерно с 50-го эпизода наблюдается резкий скачок в уровне награды до максимального значения (~200), что указывает на достижение устойчивой успешной стратегии балансировки. После этого поведение агента становится стабильным, награда сохраняется на высоком уровне, за исключением редких спадов (в районе 160 и 200 эпизодов), вероятно вызванных эпизодическим отклонением от оптимальной политики. Таким образом, можно заключить, что агент успешно обучился и способен надёжно решать задачу в большинстве эпизодов.

Рисунок 2 иллюстрирует изменение функции потерь (Loss) во время обучения. По нему можно судить о сходимости нейросети и корректности настройки гиперпараметров обучения.

В первые 50 эпизодов функция потерь демонстрирует значительные колебания и высокие значения, что объясняется высокой неопределённостью в

действиях агента и неточными предсказаниями Q-функции. Начиная с 50-го эпизода, потери начинают резко снижаться, что свидетельствует о начале сходимости модели. После 100 эпизода Loss стабилизируется на низком уровне, с минимальными колебаниями, что подтверждает успешную настройку нейросети и аппроксимацию функции полезности. Также имеются выбросы примерно в тех же эпизодах. Это может быть связано с тем, что ε ещё не равен 0 — то есть агент всё ещё иногда действует случайно.

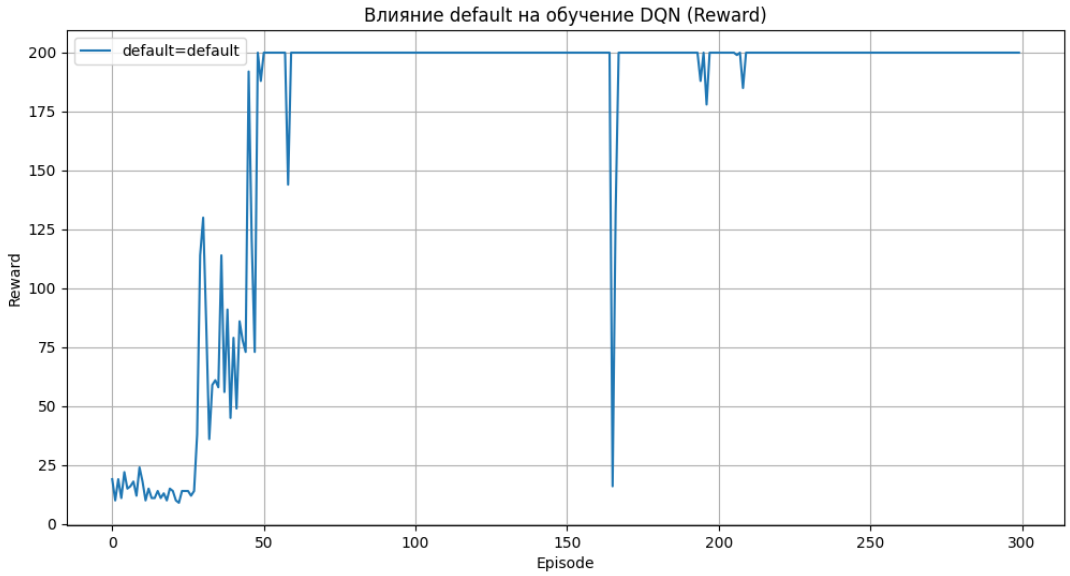

Рисунок 1. Изменение суммарного вознаграждения (Reward) по эпизодам

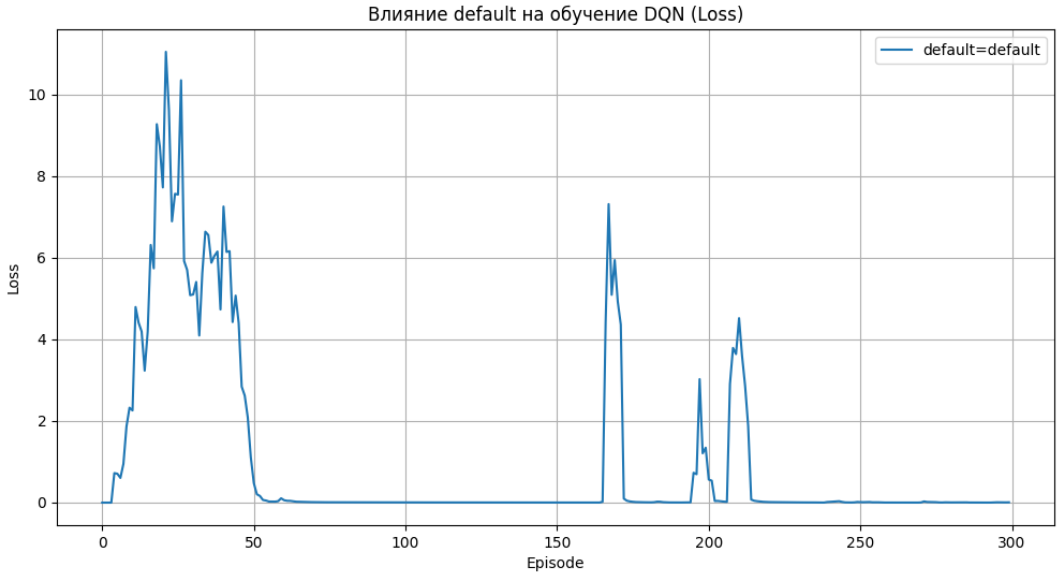

Рисунок 2. Изменение функции потерь (Loss) по эпизодам

2. Влияние изменения архитектуры нейронной сети.

Для оценки влияния архитектуры сети на эффективность и стабильность обучения были протестированы несколько вариантов моделей с различной глубиной и шириной скрытых слоёв. Архитектуры различались количеством скрытых слоёв и числом нейронов в них:

- -Shallow: один скрытый слой из 64 нейронов
- Default (базовая): два скрытых слоя по 64 нейрона
- Deep: три скрытых слоя по 128 нейронов
- Wide: два скрытых слоя по 256 нейронов

Каждая конфигурация использовалась при прочих равных параметрах обучения (см. раздел 1). Для каждой модели был проведён отдельный эксперимент, продолжавшийся 300 эпизодов.

Графики сравнения различных архитектур представлены на рисунках 3 и 4:

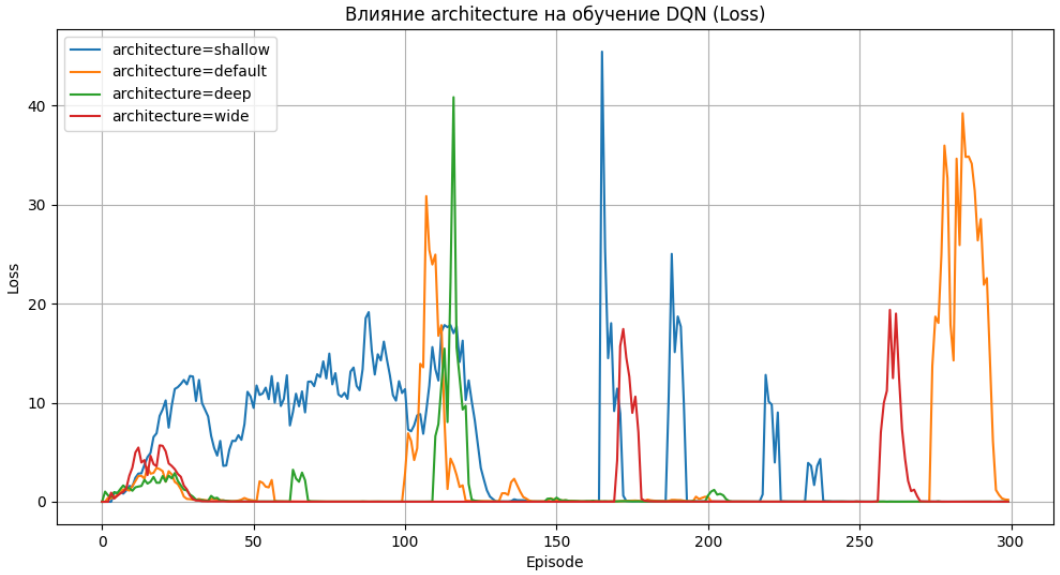

Рисунок 3. Влияние архитектуры на обучение DQN (Loss)

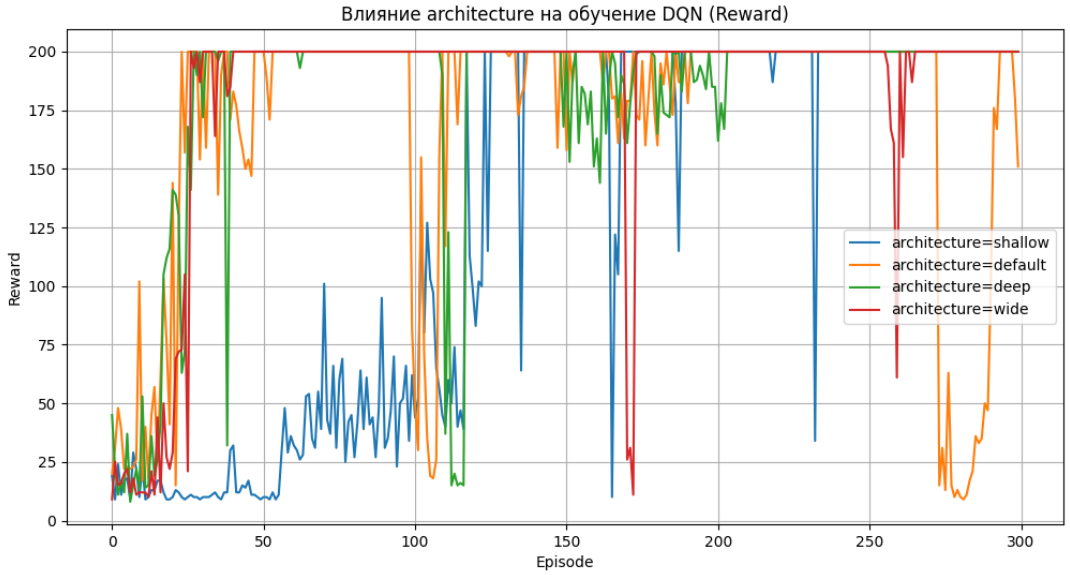

Рисунок 4. Влияние архитектуры на обучение DQN (Reward)

Shallow: Демонстрирует наибольшие колебания loss на протяжении всей тренировки, что говорит о нестабильности и трудности в обучении. Достигает максимального вознаграждения (200), но с большими колебаниями. Это говорит о том, что агент может научиться решать задачу, но его поведение нестабильно. Такая архитектура недостаточно сложна для эффективного обучения, что приводит к неустойчивому поведению.

Default: Также имеет всплески loss, но в целом быстрее стабилизируется, особенно после ~200 эпизода. Достигает максимального вознаграждения (200) быстрее и более стабильно, чем “Shallow”. Базовая архитектура (два слоя по 64 нейрона) показывает хорошую производительность, предлагая баланс между сложностью сети и стабильностью обучения.

Deep: Loss довольно быстро сходится, с небольшими колебаниями. Быстро достигает максимального вознаграждения и демонстрирует более стабильное поведение, чем “Shallow” и “Default” в большей части эпизодов.

Wide: Показывает значительные колебания потерь, похожие на “Default”. Достигает максимального вознаграждения (200), но демонстрирует колебания.

Результаты показывают, что архитектура нейронной сети оказывает значительное влияние на обучение DQN. Необходимо экспериментировать с различными архитектурами для достижения оптимальной производительности и стабильности.

3. Влияние изменения параметра *gamma*.

Коэффициент дисконтирования γ определяет, насколько сильно агент учитывает будущие награды при принятии решений. При значениях γ, близких к 0, агент ориентируется в основном на немедленные вознаграждения, тогда как при γ, приближающемся к 1, он стремится максимизировать сумму долгосрочных наград. Этот параметр критически влияет на стратегию обучения агента и может как ускорить, так и затормозить процесс обучения в зависимости от выбранного значения. Для анализа влияния γ на поведение агента были проведены эксперименты с различными значениями этого параметра. Графики наград и функции потерь для разных γ представлены на рисунках 5 и 6.

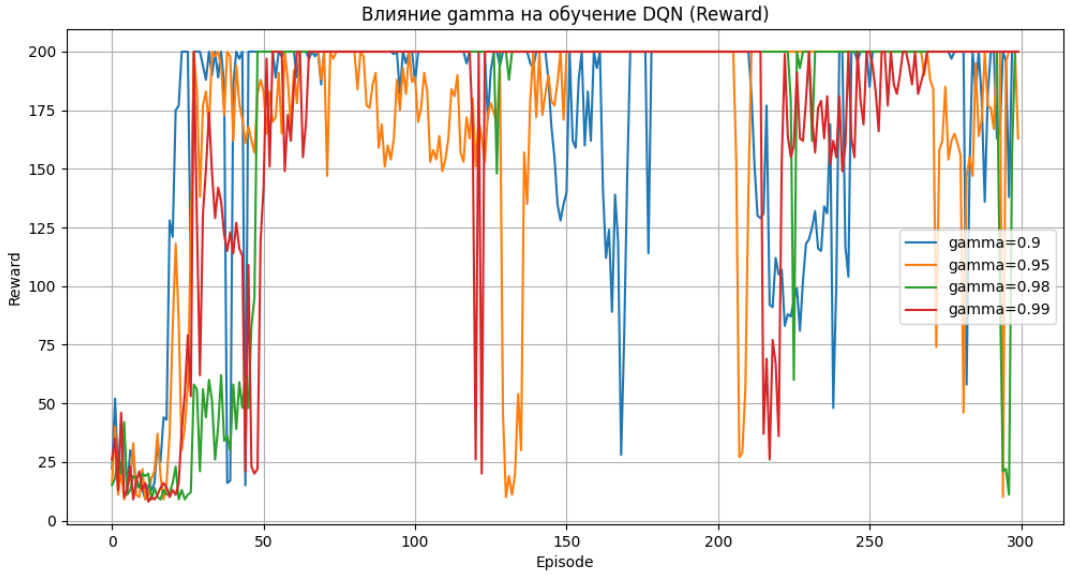

Рисунок 5. Влияние gamma на обучение DQN (Reward)

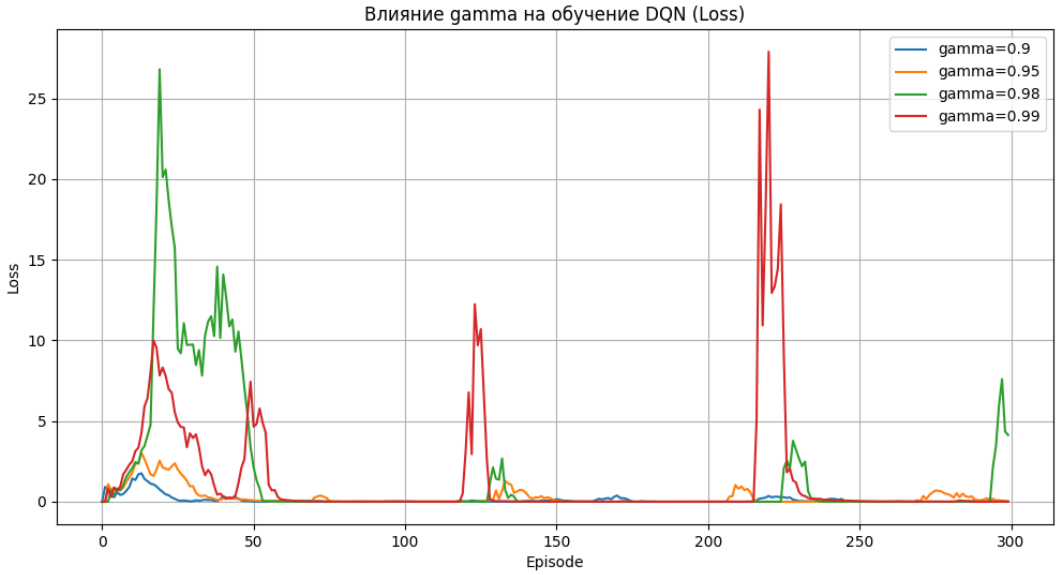

Рисунок 6. Влияние gamma на обучение DQN (Loss)

При анализе функции потерь (Loss) видно, что для γ=0.9 обучение происходит наиболее стабильно, с быстрым снижением потерь до низких значений. Значения γ=0.95 также показывают хорошие результаты, хотя и с несколько большими колебаниями. При γ=0.98 обучение происходит медленнее, а кривая потерь демонстрирует более значительные колебания. Для γ=0.99 наблюдаются значительные колебания потерь на протяжении всего обучения, что

свидетельствует о нестабильности. Рассматривая графики вознаграждения (Reward), видно, что γ=0.9 обеспечивает наиболее быстрое достижение максимального вознаграждения (200) и стабильное поведение. γ=0.95 также позволяет достичь максимального вознаграждения, но с некоторыми колебаниями. Для γ=0.98 достижение максимального вознаграждения занимает больше времени, а поведение менее стабильно. При γ=0.99 поведение агента становится нестабильным, что выражается в значительных колебаниях вознаграждения. Таким образом, можно заключить, что для данной задачи CartPole-v1 оптимальные значения γ находятся в диапазоне 0.9 - 0.95, обеспечивая баланс между учетом будущих наград и стабильностью обучения, в то время как более высокое значение γ может приводить к нестабильности.

4. Влияние изменения параметра *epsilon_decay.*

Данный параметр контролирует скорость уменьшения значения *epsilon*. Слишком быстрое уменьшение *epsilon* (высокий *epsilon_decay*) может привести к преждевременному прекращению исследования и застреванию в неоптимальной стратегии. С другой стороны, слишком медленное уменьшение *epsilon* (низкий *epsilon_decay*) может замедлить обучение, так как агент будет слишком долго выбирать случайные действия.

На рисунке 7 представлена зависимость суммарного вознаграждения от номера эпизода при различных значениях *epsilon_decay.* Рассматривая график вознаграждения, можно увидеть, что *epsilon_decay*=0.9 обеспечивает наиболее быстрое достижение максимального вознаграждения и стабильное поведение. Для *epsilon_decay*=0.95 и 0.97 достижение максимального вознаграждения происходит несколько медленнее, но в конечном итоге тоже достигается. Для *epsilon_decay*=0.99 наблюдаются большие колебания в вознаграждении, что свидетельствует о более нестабильном процессе обучения и, возможно, застревании в субоптимальной стратегии.

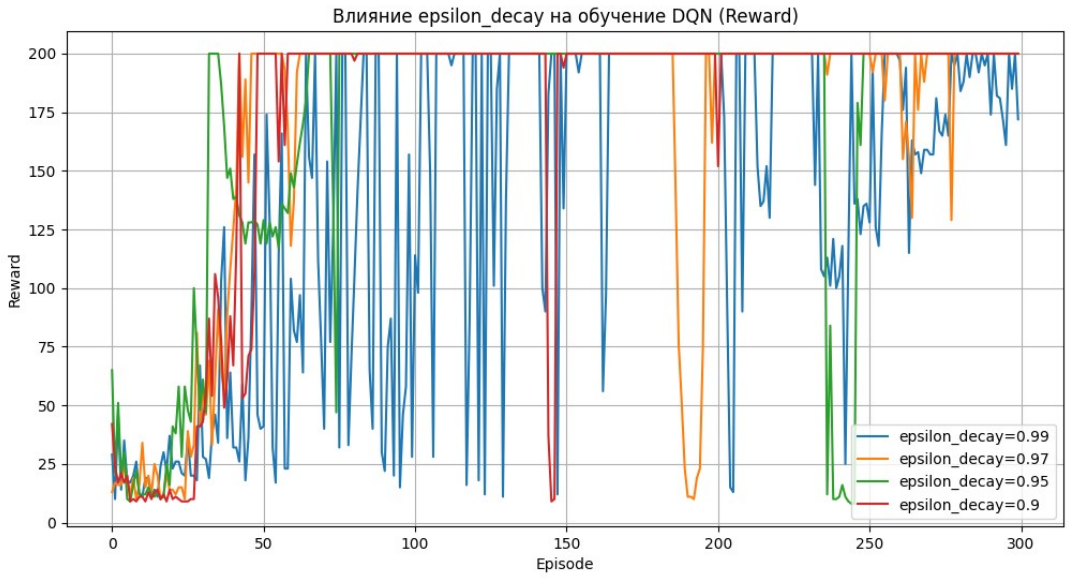

Рисунок 7. Влияние epsilon_decay на обучение DQN (Reward)

На рисунке 8 показано, как меняется функция потерь при различных значениях *epsilon_decay*. Для *epsilon_decay*=0.9 наблюдается наиболее быстрое снижение потерь и их стабильность. Значения потерь для *epsilon_decay*=0.95 и 0.97 также сходятся к нулю, но с некоторыми колебаниями и, возможно, с меньшей скоростью. Для *epsilon_decay*=0.99 наблюдаются более высокие и частые всплески потерь, что свидетельствует о нестабильности процесса обучения, вероятно, из-за слишком быстрого уменьшения уровня исследования.

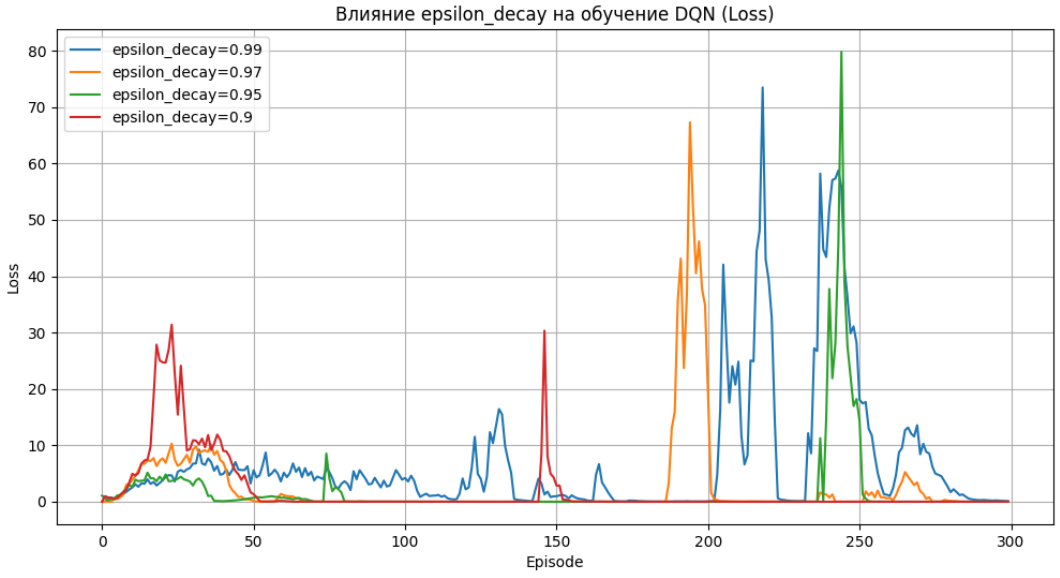

Рисунок 8. Влияние epsilon_decay на обучение DQN (Loss)

В целом, результаты указывают на то, что для задачи CartPole-v1 *epsilon_decay*=0.9 является наиболее подходящим значением, обеспечивающим оптимальный баланс между исследованием и использованием накопленного опыта. Более высокие значения могут привести к нестабильности, а более низкие — замедлить обучение.

5. Влияние изменения параметра *epsilon*.

Параметр *epsilon* определяет начальную вероятность выбора случайного действия агентом. Для оценки влияния параметра *epsilon* были проведены эксперименты с различными начальными значениями этого параметра.

На рисунке 9 представлена зависимость суммарного вознаграждения от номера эпизода при различных значениях начального *epsilon*. Значение *epsilon*=1.0 демонстрирует медленный старт, но в конечном итоге достигает максимального вознаграждения. *Epsilon*=0.5 показывает более быстрый рост вознаграждения, но и большую нестабильность поведения. Для *epsilon*=0.2 наблюдается быстрый начальный рост, который сменяется периодом низкого вознаграждения, указывающим на застревание в неоптимальном состоянии.

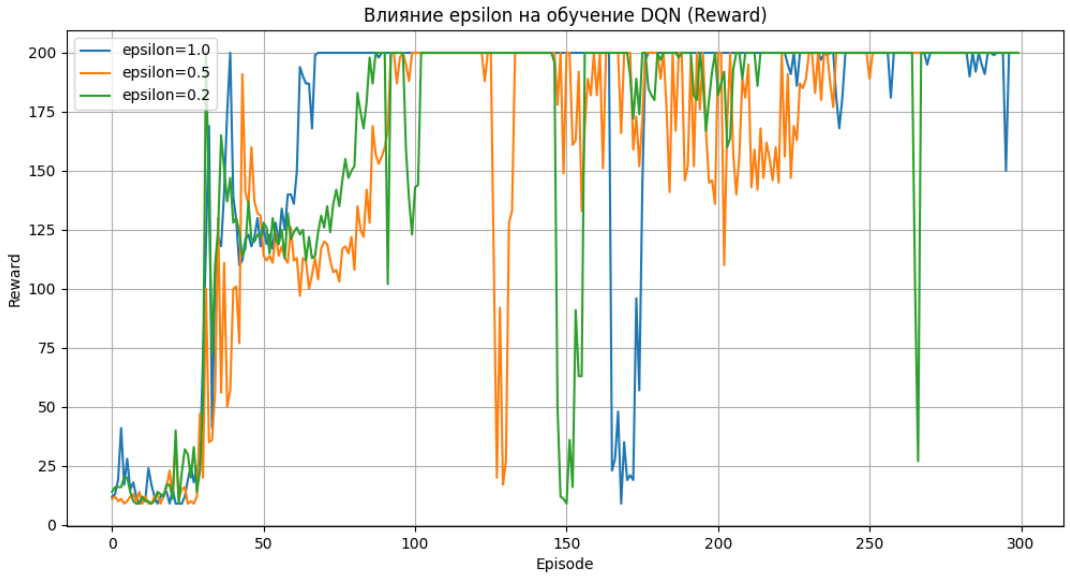

Рисунок 9. Влияние epsilon на обучение DQN (Reward)

Рисунок 10 демонстрирует изменение функции потерь при разных начальных значениях *epsilon*. Для *epsilon*=1.0 наблюдаются высокие значения

потерь в начале обучения, которые быстро снижаются, но с продолжением обучения наблюдаются скачки. Для *epsilon*=0.5 снижение потерь также происходит быстро, но видны значительные всплески. При *epsilon*=0.2 начальный этап характеризуется низкими значениями потерь, но затем возникают длительные скачки, говорящие о проблемах в обучении.

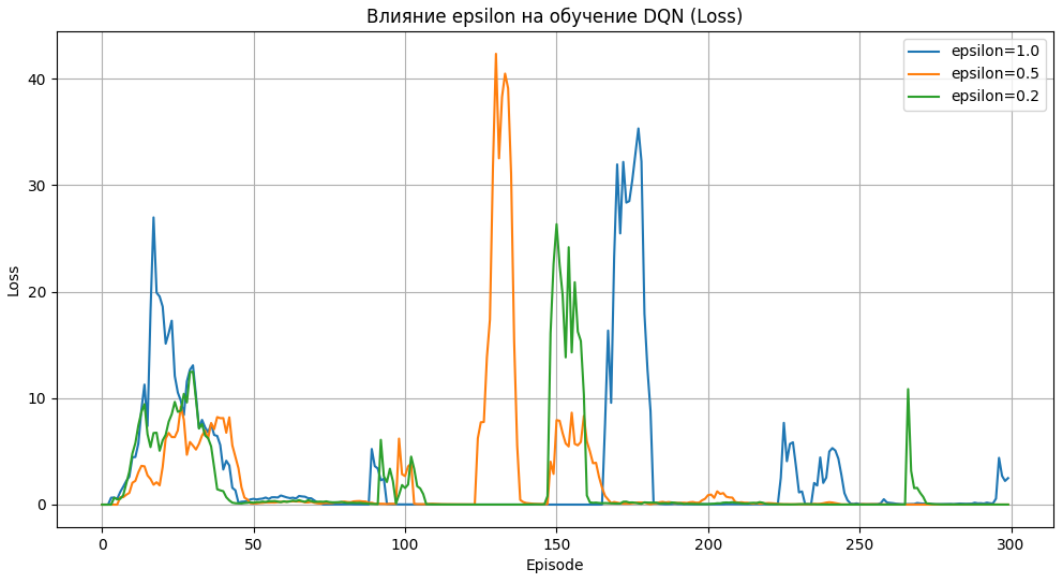

Рисунок 10. Влияние epsilon на обучение DQN (Loss)

В целом, результаты показывают, что для задачи CartPole-v1, epsilon=1.0 обеспечивает надежное обучение за счет тщательного исследования, epsilon=0.5 может ускорить обучение, но повышает нестабильность, а epsilon=0.2 может привести к застреванию в неоптимальном состоянии.

Заключение.

В ходе выполнения лабораторной работы была успешно реализована модель DQN для решения задачи управления маятником в среде CartPole-v1 с использованием библиотеки PyTorch. Проведен анализ влияния различных гиперпараметров (архитектура нейронной сети, коэффициент дисконтирования γ, скорость уменьшения *epsilon_decay*, начальное значение *epsilon*) на процесс обучения агента. Экспериментально определены оптимальные значения этих параметров для достижения стабильного и эффективного обучения.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

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

class ReplayBuffer:
    def __init__(self, capacity=1000):
        self.buffer = deque(maxlen=capacity)

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

    def sample(self, batch_size):
        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)

class QNetwork(nn.Module):
    def __init__(self, obs_size, n_actions, arch_type="default"):
        super(QNetwork, self).__init__()
        if arch_type == "shallow":
            self.net = nn.Sequential(
```

```
                nn.Linear(obs_size, 64),
                nn.ReLU(),
                nn.Linear(64, n_actions)
            )
        elif arch_type == "default":
            self.net = nn.Sequential(
                nn.Linear(obs_size, 64),
                nn.ReLU(),
                nn.Linear(64, 64),
                nn.ReLU(),
                nn.Linear(64, n_actions)
            )
        elif arch_type == "wide":
            self.net = nn.Sequential(
                nn.Linear(obs_size, 256),
                nn.ReLU(),
                nn.Linear(256, 256),
                nn.ReLU(),
                nn.Linear(256, n_actions)
            )
        elif arch_type == "deep":
            self.net = nn.Sequential(
                nn.Linear(obs_size, 128),
                nn.ReLU(),
                nn.Linear(128, 128),
                nn.ReLU(),
                nn.Linear(128, 128),
                nn.ReLU(),
                nn.Linear(128, n_actions)
            )

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

class DQNAgent:
    def __init__(self, obs_size, n_actions, arch_type="default",
gamma=0.99, epsilon=1.0, epsilon_decay=0.955):
        self.device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")
```

```
        self.q_net = QNetwork(obs_size, n_actions, arch_type).to(self.de-
vice)
        self.target_net = QNetwork(obs_size, n_actions,
arch_type).to(self.device)
        self.target_net.load_state_dict(self.q_net.state_dict())

        self.optimizer = optim.Adam(self.q_net.parameters(), lr=1e-3)

        self.gamma = gamma
        self.batch_size = 64
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = 0.01

        self.replay_buffer = ReplayBuffer(1000)

        self.loss_fn = nn.MSELoss()

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, 1)
        else:
            with torch.no_grad():
                state_tensor = torch.tensor(state,
dtype=torch.float32).to(self.device)
                q_values = self.q_net(state_tensor)
                return torch.argmax(q_values).item()

    def train(self):
        if len(self.replay_buffer) < self.batch_size:
            return None

        state, action, reward, next_state, done = self.replay_buffer.sam-
ple(self.batch_size)
        state = state.to(self.device)
        action = action.to(self.device)
        reward = reward.to(self.device)
        next_state = next_state.to(self.device)
        done = done.to(self.device)
```

```
        q_values = self.q_net(state).gather(1, ac-
tion.unsqueeze(1)).squeeze(1)
        target_q_values = reward + self.gamma * self.tar-
get_net(next_state).max(1)[0] * (1 - done)

        loss = self.loss_fn(q_values, target_q_values)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        return loss.item()

    def update_target(self):
        self.target_net.load_state_dict(self.q_net.state_dict())

def run_experiment(arch_type="default", gamma=0.99, epsilon=1.0, epsi-
lon_decay=0.955, episodes=300, label=None):
    env = gym.make("CartPole-v1", render_mode=None)
    agent = DQNAgent(
        obs_size=4,
        n_actions=2,
        arch_type=arch_type,
        gamma=gamma,
        epsilon=epsilon,
        epsilon_decay=epsilon_decay
    )

    rewards = []
    losses = []

    for episode in tqdm(range(episodes), desc=label):
        state, _ = env.reset()
        total_reward = 0
        episode_losses = []

        for _ in range(200):
            action = agent.select_action(state)
            next_state, reward, done, _, _ = env.step(action)
```

```
            agent.replay_buffer.push(state, action, reward, next_state,
done)
            loss = agent.train()
            if loss is not None:
                episode_losses.append(loss)
            state = next_state
            total_reward += reward
            if done:
                break

        agent.update_target()
        agent.epsilon = max(agent.epsilon * agent.epsilon_decay,
agent.epsilon_min)
        rewards.append(total_reward)
        losses.append(np.mean(episode_losses) if episode_losses else 0)

    env.close()
    return rewards, losses


def plot_experiment_results(param_name, values, param_key):
    results = {}

    for val in values:
        kwargs = {param_key: val}
        rewards, losses = run_experiment(**kwargs, label=f"{param_name}:
{val}")
        results[val] = {"rewards": rewards, "losses": losses}

    os.makedirs("plots", exist_ok=True)

    plt.figure(figsize=(12, 6))
    for val in values:
        plt.plot(results[val]["rewards"], label=f"{param_name}={val}")
    plt.title(f"Влияние {param_name} на обучение DQN (Reward)")
    plt.xlabel("Episode")
    plt.ylabel("Reward")
    plt.legend()
    plt.grid()
    filename_reward = f"plots/reward_vs_{param_name}.png"
```

```
    plt.savefig(filename_reward)
    # plt.show()

    plt.figure(figsize=(12, 6))
    for val in values:
        plt.plot(results[val]["losses"], label=f"{param_name}={val}")
    plt.title(f"Влияние {param_name} на обучение DQN (Loss)")
    plt.xlabel("Episode")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid()
    filename_loss = f"plots/loss_vs_{param_name}.png"
    plt.savefig(filename_loss)
    # plt.show()

plot_experiment_results("default", ["default"], "arch_type")

plot_experiment_results("architecture", ["shallow", "default", "deep",
"wide"], "arch_type")

plot_experiment_results("gamma", [0.90, 0.95, 0.98, 0.99], "gamma")

plot_experiment_results("epsilon", [1.0, 0.5, 0.2], "epsilon")

plot_experiment_results("epsilon_decay", [0.99, 0.97, 0.95, 0.90], "epsi-
lon_decay")
```