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

ОТЧЕТ

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

Студент гр. 0306 _____ Гудов Н.Р.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург
2025

Цель работы.

Ознакомиться с DQN и реализовать её с помощью библиотеки pytorch для решения задачи CartPole-v1.

Задание.

1) Реализовать базовую версию DQN для решения задачи CartPole-v1.

2) Проанализировать изменение в скорости обучения при изменении структуры используемой нейронной сети.

3) Проанализировать влияние изменения параметров gamma и epsilon_decay.

4) Провести исследование как изначальное значение epsilon влияет на скорость обучения.

Выполнение работы.

Реализация DQN.

Основная цель агента — максимизировать суммарную награду, удерживая шест в вертикальном положении как можно дольше. В качестве Q-функции применялась полносвязная нейронная сеть с несколькими скрытыми слоями. Базовая архитектура включала входной слой: принимает 4 параметра состояния среды (позиция и скорость тележки, угол и угловая скорость шеста). Скрытые слои: два промежуточных слоя с функцией активации ReLU для нелинейности. Выходной слой: возвращает Q-значения для двух возможных действий (движение влево или вправо). Для экспериментов также были разработаны: Упрощенная версия сети: с меньшим количеством нейронов в скрытых слоях. Усложненная версия сети: с дополнительными слоями и большим числом нейронов.

Experience Replay: Буфер воспроизведения сохранял переходы между состояниями в виде кортежей (состояние, действие, награда, следующее состояние, флаг завершения). Две нейронные сети: Policy Network: использовалась для выбора действий и обновлялась на каждом шаге. Target Network: применялась для стабильного расчета целевых Q-значений и обновлялась периодически с помощью мягкого копирования весов. Агент обучался в течение 600 эпизодов, каждый из которых длился до 500 шагов или до падения шеста.

Изменение архитектуры нейронной сети.

На графике видно, что малая сеть обучается медленнее и не достигает максимальной производительности даже после 600 эпизодов, что связано с недостаточной емкостью модели для сложных данных. Средняя архитектура демонстрирует сбалансированную кривую обучения, постепенно наращивая продолжительность жизни агента и стабилизируясь около 500 шагов к 600-му эпизоду, что подтверждает ее оптимальность для данной задачи. Большая сеть, несмотря на избыточную сложность, показывает наиболее быстрый рост в

начальных эпизодах, но затем ее кривая становится менее стабильной из-за переобучения и высокого времени вычислений, что проявляется в колебаниях на графике после 300 эпизодов.

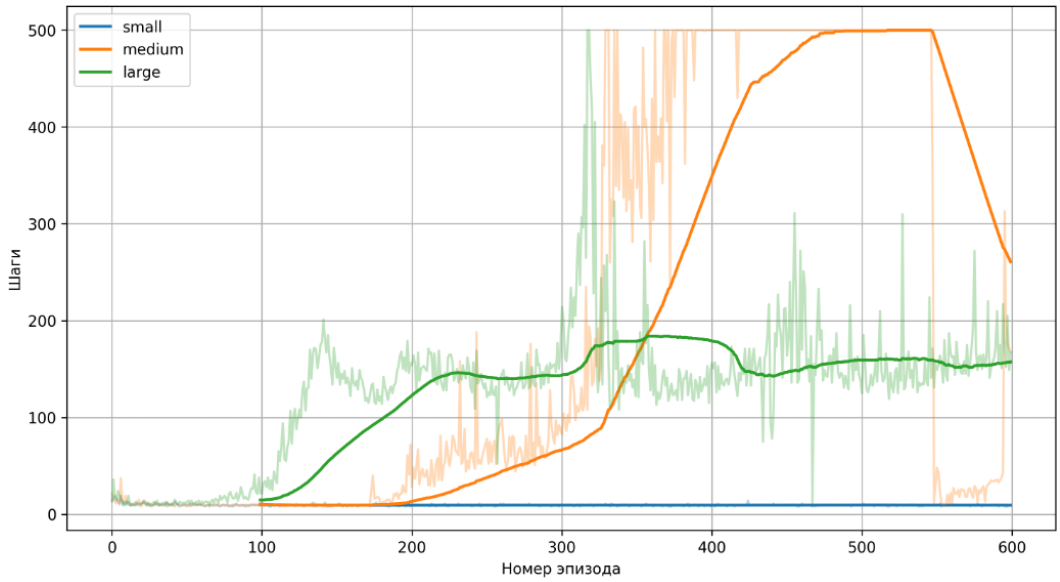

Рисунок 1. Влияние изменения архитектуры нейронной сети.

Изменение параметра gamma.

Коэффициент gamma отвечает за интерес к долгосрочной выгоде. При значении 0.9 агент демонстрирует начальный рост, но затем стабилизируется на относительно низком уровне производительности. Значение 0.99 обеспечивает сбалансированный подход. После начального резкого роста около 200 эпизодов кривая плавно выходит на стабильное плато в 500 шагов. Значение 0.999, приводит к замедленному обучению в первых 400 эпизодах, однако в долгосрочной перспективе также достигает максимальной производительности. Все кривые при разных gamma сохраняют общую форму.

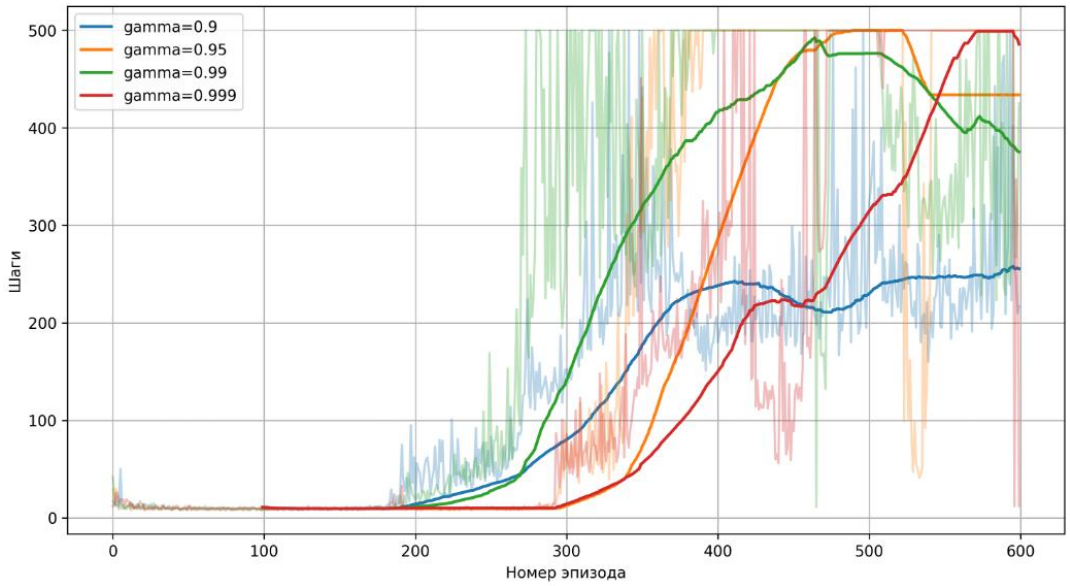

Рисунок 2. Влияние изменения gamma.

Изменение параметров epsilon_decay и epsilon.

Параметры epsilon и epsilon_decay влияют на частоту выбора случайных действий и то как эта частота будет уменьшаться. Лучшие результаты были достигнуты при средних значениях начального эпсилон. При таких условиях модели удается получить достаточно случайного опыта и в то же время эффективно обращаться к нему для выполнения задачи.

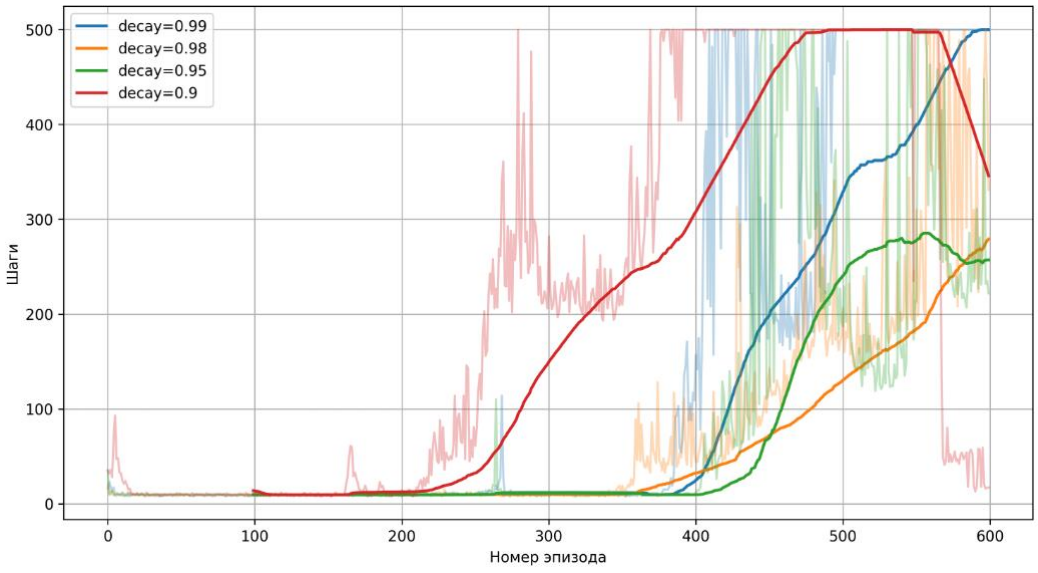

Рисунок 3. Влияние изменения decay.

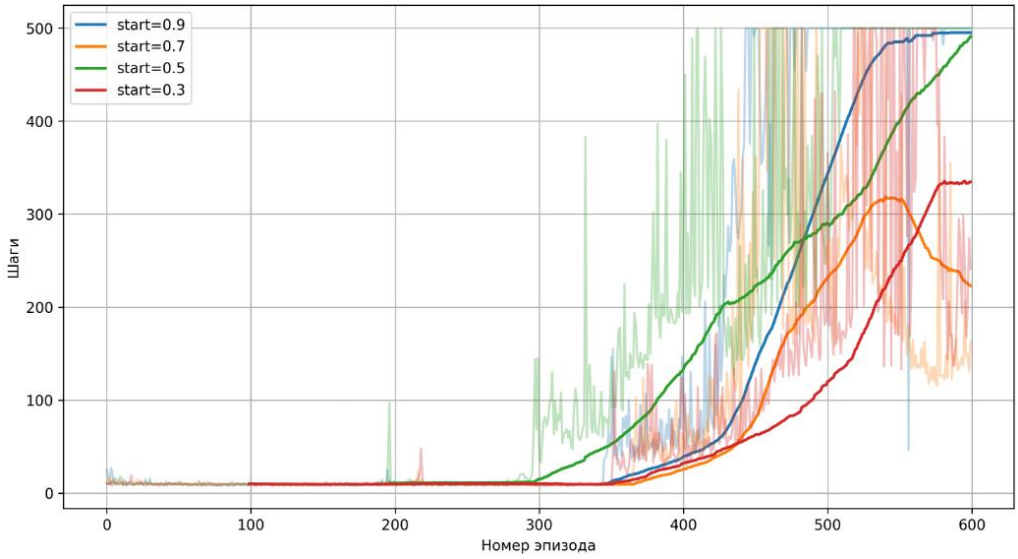

Рисунок 4. Влияние изменения epsilon.

Вывод.

В ходе лабораторной работы рассмотрен алгоритм глубокого обучения с подкреплением DQN и осуществлена его практическая реализация с использованием фреймворка PyTorch. Проведен эксперимент по оценке влияния ключевых гиперпараметров на эффективность обучения модели в среде CartPole v1.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

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

DEVICE = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

class NeuralArchitectures:
    def small(obs_size, n_actions):
        return nn.Sequential(
            nn.Linear(obs_size, 32),
            nn.ReLU(),
            nn.Linear(32, n_actions))

    def medium(obs_size, n_actions):
        return nn.Sequential(
            nn.Linear(obs_size, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, n_actions))

    def large(obs_size, n_actions):
        return nn.Sequential(
            nn.Linear(obs_size, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, n_actions))


class ExperienceReplay:
    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))
```

```
    def sample(self, batch_size):
        states, actions, rewards, next_states, dones =
zip(*random.sample(self.buffer, batch_size))
        return (
            torch.FloatTensor(np.array(states)).to(DEVICE),
            torch.LongTensor(np.array(actions)).to(DEVICE),
            torch.FloatTensor(np.array(rewards)).to(DEVICE),
            torch.FloatTensor(np.array(next_states)).to(DEVICE),
            torch.FloatTensor(np.array(dones)).to(DEVICE))

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

class DQNAgent:
    def __init__(self, obs_size, n_actions,
                 network_arch='medium',
                 gamma=0.99,
                 epsilon_start=1.0,
                 epsilon_min=0.01,
                 epsilon_decay=0.995,
                 lr=1e-4,
                 tau=0.005):

        self.policy_net = getattr(NeuralArchitectures,
network_arch)(obs_size, n_actions).to(DEVICE)
        self.target_net = getattr(NeuralArchitectures,
network_arch)(obs_size, n_actions).to(DEVICE)

self.target_net.load_state_dict(self.policy_net.state_dict())

        self.optimizer = optim.Adam(self.policy_net.parameters(),
lr=lr)
        self.memory = ExperienceReplay()

        self.gamma = gamma
        self.epsilon = epsilon_start
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.tau = tau
        self.n_actions = n_actions

        self.steps_history = []
        self.steps_done = 0

    def select_action(self, state):
        self.steps_done += 1
        self.epsilon = max(self.epsilon_min, self.epsilon *
self.epsilon_decay)

        if random.random() < self.epsilon:
            return random.randint(0, self.n_actions - 1)

        with torch.no_grad():
            state_tensor =
torch.FloatTensor(state).unsqueeze(0).to(DEVICE)
            return self.policy_net(state_tensor).argmax().item()

    def update_model(self):
```

```
        if len(self.memory) < BATCH_SIZE:
            return

        states, actions, rewards, next_states, dones =
self.memory.sample(BATCH_SIZE)

        current_q = self.policy_net(states).gather(1,
actions.unsqueeze(1))
        next_q = self.target_net(next_states).max(1)[0].detach()
        target_q = rewards + (1 - dones) * self.gamma * next_q

        loss = nn.MSELoss()(current_q.squeeze(), target_q)
        self.optimizer.zero_grad()
        loss.backward()

torch.nn.utils.clip_grad_norm_(self.policy_net.parameters(), 100)
        self.optimizer.step()

        for target_param, policy_param in
zip(self.target_net.parameters(), self.policy_net.parameters()):
            target_param.data.copy_(self.tau * policy_param.data +
(1 - self.tau) * target_param.data)

    def train(self, env, num_episodes=800, max_steps=500,
log_interval=50):
        progress = tqdm(range(num_episodes), desc="Обучение")

        for episode in progress:
            state, _ = env.reset()
            steps = 0

            for step in range(max_steps):
                action = self.select_action(state)
                next_state, reward, done, truncated, _ =
env.step(action)

                self.memory.push(state, action, reward, next_state,
done)
                self.update_model()

                state = next_state
                steps += 1

                if done or truncated:
                    break

            self.steps_history.append(steps)

            if episode % log_interval == 0:
                progress.set_description(
                    f"Episode {episode}: Steps {steps}, "
                    f"Epsilon {self.epsilon:.2f}"
                )

        return self.steps_history

class ExperimentRunner:
    def run_architecture_experiment(env, num_episodes=600):
```

```
        """Эксперимент с различными архитектурами нейросетей"""
        architectures = ['small', 'medium', 'large']
        results = {}

        for arch in architectures:
            print(f"\nЗапуск эксперимента с архитектурой: {arch}")
            agent = DQNAgent(env.observation_space.shape[0],
env.action_space.n, network_arch=arch)
            steps = agent.train(env, num_episodes=num_episodes)
            results[arch] = steps

        ExperimentRunner._plot_results(
            results,
            title="Сравнение различных архитектур нейросетей",
            filename="architecture_comparison.png")


    def run_gamma_experiment(env, num_episodes=600):
        """Эксперимент с различными значениями gamma"""
        gammas = [0.9, 0.95, 0.99, 0.999]
        results = {}

        for gamma in gammas:
            print(f"\nЗапуск эксперимента с gamma={gamma}")
            agent = DQNAgent(env.observation_space.shape[0],
env.action_space.n, gamma=gamma)
            steps = agent.train(env, num_episodes=num_episodes)
            results[f"gamma={gamma}"] = steps

        ExperimentRunner._plot_results(
            results,
            title="Сравнение различных значений gamma",
            filename="gamma_comparison.png")

    def run_epsilon_decay_experiment(env, num_episodes=600):
        """Эксперимент с различными значениями epsilon_decay"""
        decays = [0.99, 0.98, 0.95, 0.9]
        results = {}

        for decay in decays:
            print(f"\nЗапуск эксперимента с epsilon_decay={decay}")
            agent = DQNAgent(
                env.observation_space.shape[0],
                env.action_space.n,
                epsilon_decay=decay)
            steps = agent.train(env, num_episodes=num_episodes)
            results[f"decay={decay}"] = steps

        ExperimentRunner._plot_results(
            results,
            title="Сравнение различных значений epsilon_decay",
            filename="epsilon_decay_comparison.png")


    def run_epsilon_start_experiment(env, num_episodes=600):
        """Эксперимент с различными начальными значениями
epsilon"""
        starts = [0.9, 0.7, 0.5, 0.3]
```

```
        results = {}

        for start in starts:
            print(f"\nЗапуск эксперимента с epsilon_start={start}")
            agent = DQNAgent(
                env.observation_space.shape[0],
                env.action_space.n,
                epsilon_start=start)
            steps = agent.train(env, num_episodes=num_episodes)
            results[f"start={start}"] = steps

        ExperimentRunner._plot_results(
            results,
            title="Сравнение различных начальных значений epsilon",
            filename="epsilon_start_comparison.png")


    def _plot_results(results, title, filename):
        """Визуализация количества шагов в эпизоде с одинаковыми
цветами для среднего"""
        plt.figure(figsize=(10, 6))

        for label, steps in results.items():
            color = f"C{list(results.keys()).index(label)}"
            plt.plot(steps, label=label, alpha=0.3, color=color)

            if len(steps) >= 100:
                window_size = 100
                cumsum = np.cumsum(np.insert(steps, 0, 0))
                moving_avg = (cumsum[window_size:] - cumsum[:-
window_size]) / window_size
                plt.plot(np.arange(window_size-1, len(steps)),
                        moving_avg,
                        color=color,
                        linestyle='-',
                        linewidth=2,
                        label=f'{label} (среднее)')

        plt.title(f"Количество шагов в эпизоде\n{title}")
        plt.xlabel("Номер эпизода")
        plt.ylabel("Шаги")

        handles, labels = plt.gca().get_legend_handles_labels()
        unique_labels = {}
        for h, l in zip(handles, labels):
            if '(' not in l:
                unique_labels[l] = h

        legend_elements = [lines.Line2D([0], [0],
color=h.get_color(),
                        lw=2, label=k) for k, h in
unique_labels.items()]

        plt.legend(handles=legend_elements, loc='best')
        plt.grid(True)
        plt.tight_layout()

        os.makedirs("results", exist_ok=True)
```

```
        plt.savefig(f"results/{filename}", dpi=300,
bbox_inches='tight')
        plt.close()

BATCH_SIZE = 128

if __name__ == "__main__":
    env = gym.make('CartPole-v1')

    print("\n=== Эксперимент с архитектурами нейросетей ===")
    ExperimentRunner.run_architecture_experiment(env)

    print("\n=== Эксперимент с различными gamma ===")
    ExperimentRunner.run_gamma_experiment(env)

    print("\n=== Эксперимент с различными epsilon_decay ===")
    ExperimentRunner.run_epsilon_decay_experiment(env)

    print("\n=== Эксперимент с различными epsilon_start ===")
    ExperimentRunner.run_epsilon_start_experiment(env)

    env.close()
```