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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Обучение с подкреплением»

Тема: Реализация PPO для среды MountainCarContinuous-v0

Студент гр. 0306 _____ Гудов Н.Р.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург

2025

Цель работы.

Ознакомиться с PPO и реализовать её с помощью библиотеки pytorch для решения задачи MountainCarContinuous-v0.

Задание.

1) Измените длину траектории (steps).
2) Подберите оптимальный коэффициент clip_ratio.
3) Добавьте нормализацию преимуществ.
4) Сравните обучение при разных количествах эпох.

Выполнение работы.

Реализация PPO.

Реализация алгоритма состоит из двух нейросетей - Policy Network с выходным нормальным распределением для генерации действий и Value Network для оценки состояний. Процесс обучения инициируется сбором траекторий фиксированной длины, в течение которых фиксируются состояния, действия, награды и терминальные флаги. Для вычисления advantages применяется метод GAE с возможностью нормализации, что стабилизирует процесс обучения. Оптимизация осуществляется через минимизацию составной функции потерь, включающей три компонента: clipped surrogate-потерю для ограничения изменения стратегии, MSE-потерю для функции ценности и регуляризацию энтропией. В экспериментах исследуются влияние длины траектории, коэффициента обрезки, нормализации advantages и количества эпох PPO.

Изменение длины траектории.

На графике видно, что при малой длине траектории кривая демонстрирует резкие скачки, что свидетельствует о недостаточной статистике для качественного обновления политики. Средняя длина траектории обеспечивает более плавный рост награды, при этом сохраняя приемлемую скорость обучения. При большой длине алгоритму требуется больше итераций для достижения сопоставимых результатов. Все три варианта в конечном итоге достигают схожего уровня производительности, но разными путями: короткие траектории быстрее реагируют на изменения, но менее стабильны, тогда как длинные траектории обеспечивают плавное, но более надёжное обучение.

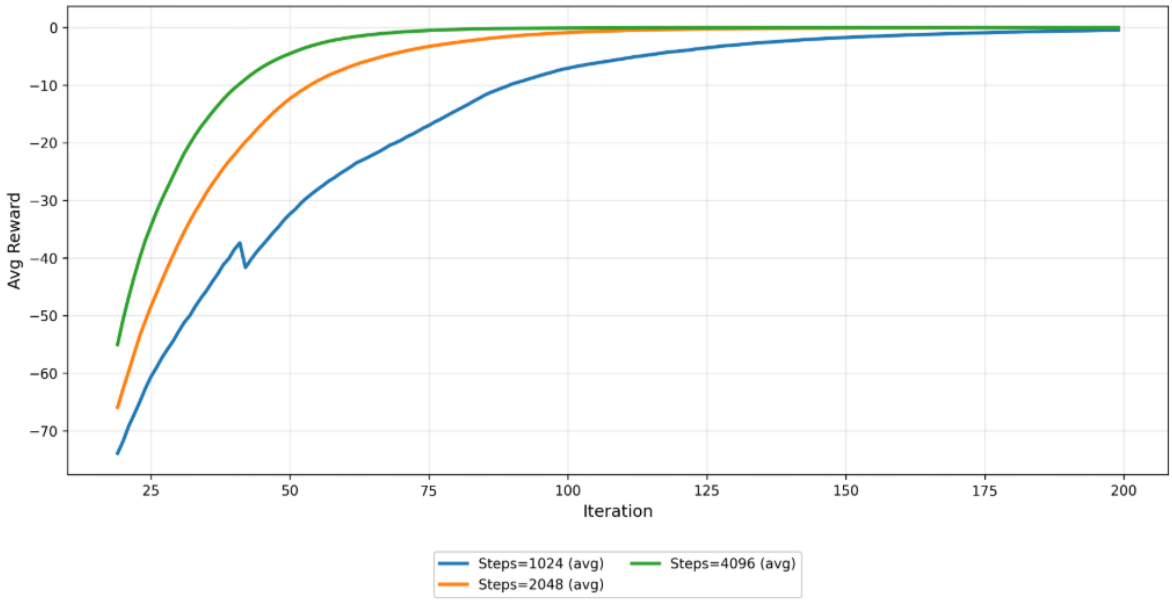

Рисунок 1. Влияние длины траектории.

Подбор оптимального коэффициент clip_ratio.

При минимальном значении наблюдается осторожное обновление политики, что выражается в стабильном, но крайне медленном прогрессе - алгоритм требует значительного числа итераций для достижения приемлемых результатов. Оптимальное значение 0.2 обеспечивает сбалансированную динамику. Быстрое начальное обучение с последующей стабилизацией, умеренные колебания наград между эпизодами и достижение наивысшей итоговой производительности. Увеличение коэффициента до 0.3 приводит к характерным артефактам обучения агрессивным, но нестабильным обновлениям политики, проявляющимся в резких скачках награды и периодических деградациях стратегии.

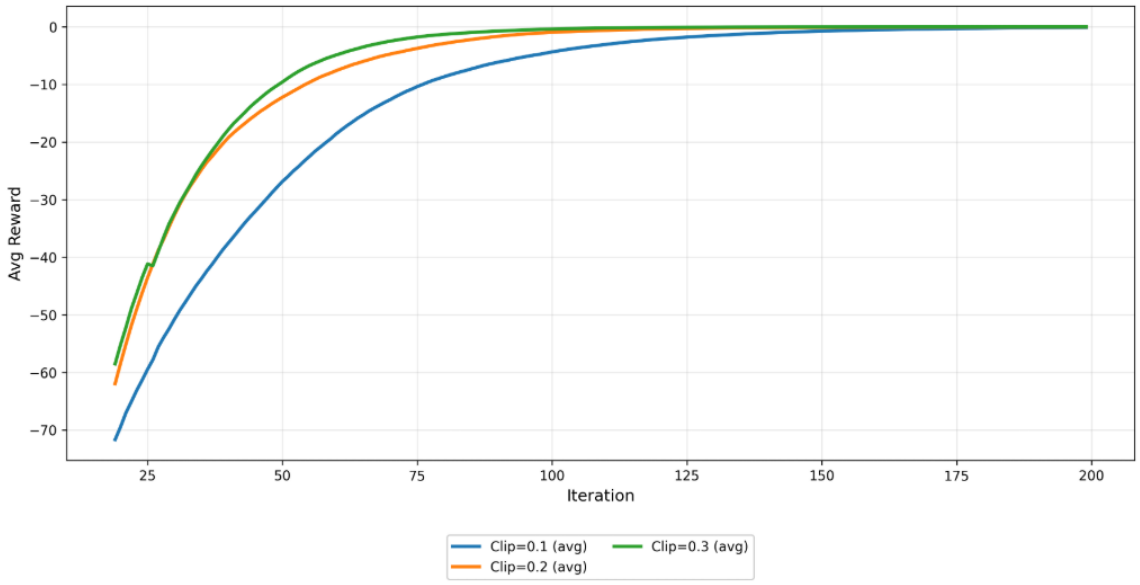

Рисунок 2. Влияние коэффициента clip.

Добавление нормализации.

По графику видно, что нормализация преимуществ стабилизирует обучение, ускоряет сходимость и повышает итоговую награду.

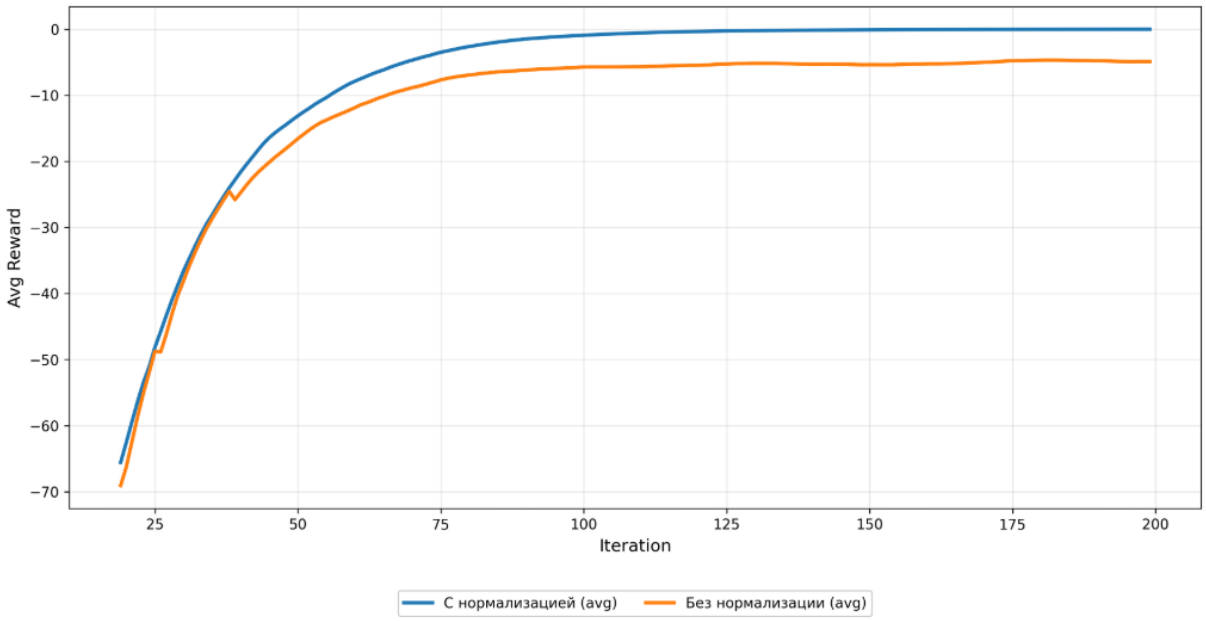

Рисунок 3. Сравнение нормализации.

Сравнение при разном количестве эпох.

Чем больше эпох PPO, тем стабильнее, но медленнее обучение. 5 эпох: учится быстро, но нестабильно. 10 эпох лучший баланс - хорошая скорость и стабильность. 20 эпох: учится очень плавно.

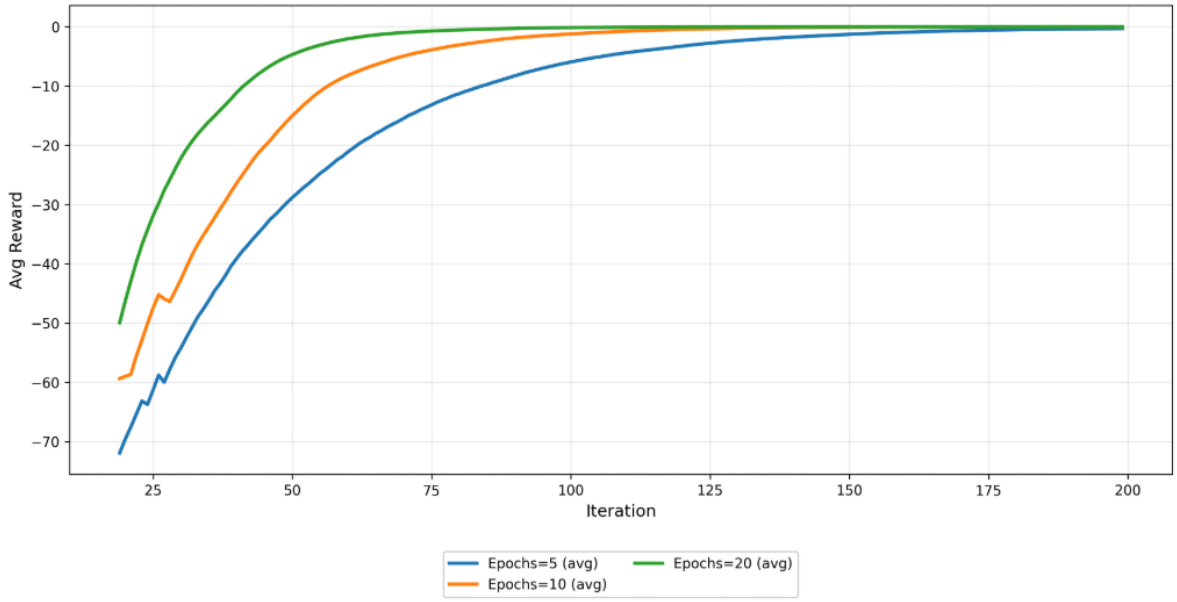

Рисунок 4. Сравнение разного количества эпох.

Вывод.

В ходе лабораторной работы рассмотрен алгоритм глубокого обучения с подкреплением PPO и осуществлена его практическая реализация с использованием фреймворка PyTorch. Проведен эксперимент по оценке влияния ключевых гиперпараметров на эффективность обучения модели в среде MountainCarContinuous-v0.

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 torch.distributions import Normal
from tqdm import tqdm

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

class PolicyNetwork(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_size=128):
        super(PolicyNetwork, self).__init__()
        self.shared_layers = nn.Sequential(
            nn.Linear(input_dim, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU()
        )
        self.mu_layer = nn.Linear(hidden_size, output_dim)
        self.log_std = nn.Parameter(torch.zeros(output_dim))

    def forward(self, state):
        features = self.shared_layers(state)
        return torch.tanh(self.mu_layer(features))

    def get_distribution(self, state):
        mu = self.forward(state)
        std = torch.exp(self.log_std.clamp(-20, 2))
        return Normal(mu, std)

class ValueNetwork(nn.Module):
    def __init__(self, input_dim, hidden_size=128):
        super(ValueNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, state):
```

```
        return self.net(state).squeeze()

class PPOTrainer:
    def __init__(self, env_name, config):
        self.env = gym.make(env_name)
        self.state_dim = self.env.observation_space.shape[0]
        self.action_dim = self.env.action_space.shape[0]
        self.config = config

        self.policy = PolicyNetwork(self.state_dim,
self.action_dim).to(DEVICE)
        self.value_net = ValueNetwork(self.state_dim).to(DEVICE)
        self.policy_optim = optim.Adam(self.policy.parameters(),
lr=config['lr'])
        self.value_optim = optim.Adam(self.value_net.parameters(),
lr=config['lr'])

        self.rewards_history = []

    def collect_trajectories(self):
        states, actions, rewards, dones, log_probs = [], [], [],
[], []
        state, _ = self.env.reset()
        ep_rewards = []
        current_ep_reward = 0

        for _ in range(self.config['num_steps']):
            state_tensor = torch.FloatTensor(state).to(DEVICE)
            with torch.no_grad():
                dist = self.policy.get_distribution(state_tensor)
                action = dist.sample()
                log_prob = dist.log_prob(action).sum()

            next_state, reward, terminated, truncated, _ =
self.env.step(action.cpu().numpy())
            done = terminated or truncated

            states.append(state)
            actions.append(action.cpu().numpy())
            rewards.append(reward)
            dones.append(done)
            log_probs.append(log_prob.item())
            current_ep_reward += reward
            state = next_state

            if done:
                ep_rewards.append(current_ep_reward)
                state, _ = self.env.reset()
                current_ep_reward = 0

        return states, actions, rewards, dones, log_probs,
ep_rewards

    def compute_advantages(self, rewards, dones, values,
normalize=True):
        returns = np.zeros_like(rewards)
        advantages = np.zeros_like(rewards)
        R = 0
```

```
        for t in reversed(range(len(rewards))):
            R = rewards[t] + self.config['gamma'] * R * (1 -
dones[t])
            returns[t] = R
            advantages[t] = R - values[t]

        if normalize:
            advantages = (advantages - advantages.mean()) /
(advantages.std() + 1e-8)

        return returns, advantages

    def update_networks(self, states, actions, old_log_probs,
returns, advantages):
        states = torch.FloatTensor(np.array(states)).to(DEVICE)
        actions = torch.FloatTensor(np.array(actions)).to(DEVICE)
        old_log_probs =
torch.FloatTensor(np.array(old_log_probs)).to(DEVICE)
        returns = torch.FloatTensor(np.array(returns)).to(DEVICE)
        advantages =
torch.FloatTensor(np.array(advantages)).to(DEVICE)

        dataset_size = states.size(0)
        indices = np.arange(dataset_size)

        for _ in range(self.config['ppo_epochs']):
            np.random.shuffle(indices)

            for start in range(0, dataset_size,
self.config['batch_size']):
                end = start + self.config['batch_size']
                idx = indices[start:end]

                batch_states = states[idx]
                batch_actions = actions[idx]
                batch_old_log_probs = old_log_probs[idx]
                batch_returns = returns[idx]
                batch_advantages = advantages[idx]

                dist = self.policy.get_distribution(batch_states)
                new_log_probs = dist.log_prob(batch_actions).sum(-
1)
                ratio = (new_log_probs - batch_old_log_probs).exp()

                surr1 = ratio * batch_advantages
                surr2 = torch.clamp(ratio, 1 -
self.config['clip_ratio'],
                                    1 + self.config['clip_ratio']) *
batch_advantages
                policy_loss = -torch.min(surr1, surr2).mean()

                value_pred = self.value_net(batch_states)
                value_loss = (value_pred -
batch_returns).pow(2).mean()

                entropy = dist.entropy().mean()
```

```
                total_loss = (policy_loss +
                              self.config['value_coef'] * value_loss
-
                              self.config['entropy_coef'] * entropy)

                self.policy_optim.zero_grad()
                self.value_optim.zero_grad()
                total_loss.backward()

torch.nn.utils.clip_grad_norm_(self.policy.parameters(), 0.5)

torch.nn.utils.clip_grad_norm_(self.value_net.parameters(), 0.5)
                self.policy_optim.step()
                self.value_optim.step()

    def train(self):
        for iteration in
tqdm(range(self.config['num_iterations'])):
            states, actions, rewards, dones, log_probs, ep_rewards
= self.collect_trajectories()

            states_tensor =
torch.FloatTensor(np.array(states)).to(DEVICE)
            with torch.no_grad():
                values =
self.value_net(states_tensor).cpu().numpy()

            returns, advantages = self.compute_advantages(rewards,
dones, values,

self.config['normalize_advantages'])

            self.update_networks(states, actions, log_probs,
returns, advantages)

            if ep_rewards:
                avg_reward = np.mean(ep_rewards)
                self.rewards_history.append(avg_reward)

            if iteration % 20 == 0 and ep_rewards:
                print(f"Iteration {iteration}: Avg Reward
{avg_reward:.2f}")

        self.env.close()
        return self.rewards_history

class ExperimentRunner:
    @staticmethod

    def plot_results(results, title, filename, xlabel='Iteration',
ylabel='Avg Reward'):
        plt.figure(figsize=(12, 7))

        for label, rewards in results.items():
            color = f"C{list(results.keys()).index(label)}"

            window_size = max(20, len(rewards) // 10)  # Адаптивный
размер окна
```

```
            cumsum = np.cumsum(np.insert(rewards, 0, 0))
            moving_avg = (cumsum[window_size:] - cumsum[:-
window_size]) / window_size
            plt.plot(np.arange(window_size-1, len(rewards)),
                    moving_avg,
                    color=color,
                    linestyle='-',
                    linewidth=2.5,
                    label=f'{label} (avg)')

        plt.title(f"Результаты обучения\n{title}", fontsize=14,
pad=20)
        plt.xlabel(xlabel, fontsize=12)
        plt.ylabel(ylabel, fontsize=12)
        plt.grid(True, alpha=0.3)

        handles, labels = plt.gca().get_legend_handles_labels()
        if handles:  # Только если есть что отображать
            plt.legend(handles, labels, loc='upper center',
                    bbox_to_anchor=(0.5, -0.15),
                    ncol=2, framealpha=1.0)

        plt.tight_layout()

        os.makedirs("results", exist_ok=True)
        save_path = os.path.join("results", filename)
        plt.savefig(save_path, dpi=300, bbox_inches='tight',
facecolor='white')
        plt.close()

    @staticmethod
    def run_steps_experiment():
        config = {
            'num_iterations': 200,
            'num_steps': 2048,
            'batch_size': 64,
            'gamma': 0.99,
            'clip_ratio': 0.2,
            'ppo_epochs': 10,
            'lr': 3e-4,
            'value_coef': 0.5,
            'entropy_coef': 0.01,
            'normalize_advantages': True
        }

        steps_options = [1024, 2048, 4096]
        results = {}

        for steps in steps_options:
            config['num_steps'] = steps
            trainer = PPOTrainer("MountainCarContinuous-v0",
config)
            results[f"Steps={steps}"] = trainer.train()

        ExperimentRunner.plot_results(results, "Влияние длины
траектории", "steps_experiment.png")

    @staticmethod
```

```
    def run_clip_experiment():
        config = {
            'num_iterations': 200,
            'num_steps': 2048,
            'batch_size': 64,
            'gamma': 0.99,
            'clip_ratio': 0.2,
            'ppo_epochs': 10,
            'lr': 3e-4,
            'value_coef': 0.5,
            'entropy_coef': 0.01,
            'normalize_advantages': True
        }

        clip_options = [0.1, 0.2, 0.3]
        results = {}

        for clip in clip_options:
            config['clip_ratio'] = clip
            trainer = PPOTrainer("MountainCarContinuous-v0",
config)
            results[f"Clip={clip}"] = trainer.train()

        ExperimentRunner.plot_results(results, "Влияние
коэффициента обрезки", "clip_experiment.png")

    @staticmethod
    def run_epochs_experiment():
        config = {
            'num_iterations': 200,
            'num_steps': 2048,
            'batch_size': 64,
            'gamma': 0.99,
            'clip_ratio': 0.2,
            'ppo_epochs': 10,
            'lr': 3e-4,
            'value_coef': 0.5,
            'entropy_coef': 0.01,
            'normalize_advantages': True
        }

        epochs_options = [5, 10, 20]
        results = {}

        for epochs in epochs_options:
            config['ppo_epochs'] = epochs
            trainer = PPOTrainer("MountainCarContinuous-v0",
config)
            results[f"Epochs={epochs}"] = trainer.train()

        ExperimentRunner.plot_results(results, "Влияние количества
эпох PPO", "epochs_experiment.png")

    @staticmethod
    def run_norm_experiment():
        config = {
            'num_iterations': 200,
            'num_steps': 2048,
```

```
            'batch_size': 64,
            'gamma': 0.99,
            'clip_ratio': 0.2,
            'ppo_epochs': 10,
            'lr': 3e-4,
            'value_coef': 0.5,
            'entropy_coef': 0.01
        }

        results = {}

        print("\nЗапуск с нормализацией преимуществ...")
        config['normalize_advantages'] = True
        trainer = PPOTrainer("MountainCarContinuous-v0", config)
        results["С нормализацией"] = trainer.train()

        print("\nЗапуск без нормализации преимуществ...")
        config['normalize_advantages'] = False
        trainer = PPOTrainer("MountainCarContinuous-v0", config)
        results["Без нормализации"] = trainer.train()

        ExperimentRunner.plot_results(results,
                                      "Сравнение нормализации
преимуществ",
                                      "norm_comparison.png")

if __name__ == "__main__":
    os.makedirs("results", exist_ok=True)

    print("=== Эксперимент 1: Длина траектории ===")
    ExperimentRunner.run_steps_experiment()

    print("\n=== Эксперимент 2: Коэффициент обрезки ===")
    ExperimentRunner.run_clip_experiment()

    print("\n=== Эксперимент 3: Количество эпох ===")
    ExperimentRunner.run_epochs_experiment()

    print("\n=== Эксперимент 4: Нормализация преимуществ ===")
    ExperimentRunner.run_norm_experiment()
            if episode % log_interval == 0:
                progress.set_description(
                    f"Episode {episode}: Steps {steps}, "
                    f"Epsilon {self.epsilon:.2f}"
                )

        return self.steps_history

class ExperimentRunner:
    @staticmethod
    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")

    @staticmethod
    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")

    @staticmethod
    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")

    @staticmethod
    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")

    @staticmethod
    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():
            # Основной график (прозрачный)
            color = f"C{list(results.keys()).index(label)}"
            plt.plot(steps, label=label, alpha=0.3, color=color)

            # Скользящее среднее (сплошная линия)
            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")

        # Улучшенная легенда (группировка по цветам)
        handles, labels = plt.gca().get_legend_handles_labels()
        unique_labels = {}
        for h, l in zip(handles, labels):
            if '(' not in l:  # Базовые записи
                unique_labels[l] = h

        # Создаем компактную легенду
        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()

# Параметры обучения
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()
```