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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Обучение с подкреплением»

Тема: Реализация PPO для среды MountainCarContinuous-v0

Студент гр. 0310 Аксенов И.В.

Преподаватель Глазунов С.А.

Санкт-Петербург
2025

СОДЕРЖАНИЕ

ЦЕЛЬ РАБОТЫ

Написать алгоритм PPO для обучения агента в среде MountainCarContinuous-v0.

ЗАДАНИЕ

1. Изменить длину траектории (steps);
2. Подобрать оптимальный коэффициент clip_ratio;
3. Добавить нормализацию преимуществ;
4. Сравните обучение при разных количествах эпох.

ВЫПОЛНЕНИЕ РАБОТЫ

1. Исходные данные

Название среды: Mountain Car Continuous.

Таблица 1 – Пространство наблюдений

Num	Observation	Min	Max	Unit
0	position of the car along the x-axis	−1.2	0.6	position (m)
1	velocity of the car	−0.07	0.07	position (m)

Пространство действий представляет собой одно значение: сила толчка, варьирующаяся от -1 до 1, применяющаяся к машинке.

Награды:

- Равна $-0.1 \times \text{action}^2$ за каждое действие, чтобы машинка не использовала слишком большие толчки;
- Если машинка достигает конца траектории, то к награде добавляется $+100$.

Начальное состояние: положение машинки устанавливается случайным образом в диапазоне от -0.6 до -0.4 на основе равномерного распределения.

Окончание эпизода:

- Если машинка достигает флажка (верхней части горки), то эпизод завершается (если позиция машинки больше или равна 0.45);
- Если количество эпизодов равно 999.

2. Изменение длины траектории

Использовались следующие значения длины траектории:

$$\text{steps} = \{512, 1024, 2048\} \tag{1}$$

Результаты эксперимента представлены на рисунке (Рисунок 1).

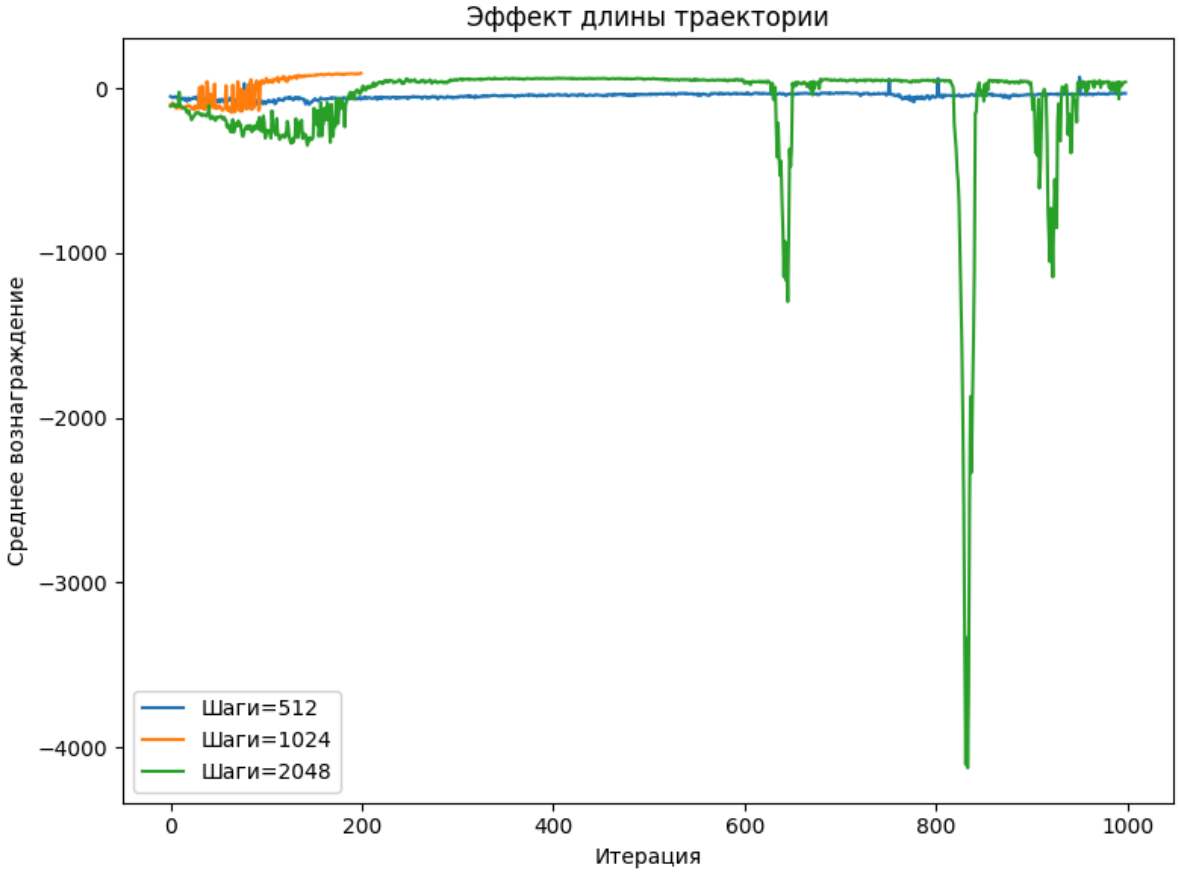

Рисунок 1 – Среднее вознаграждение в зависимости от длины траектории

По полученным результатам можно сделать следующие выводы:

- Значение длины траектории равное 1024 оказалось самым оптимальным ввиду того, что с ним модель достигла финального состояния в районе 200 итераций;
- Малое (512) и большое (2048) значение длины траектории привело к ухудшению качества обучения модели. Ни в одном из двух случаем она не смогла достигнуть терминального состояния, что можно наблюдать на рисунке;
- Худший результат у самого большого значения длины траектории (2048). В некоторых состояниях модель даже достигала награды менее 4000.

3. Подбор оптимального коэффициента clip_ratio

В ходе эксперимента использовались следующие значения:

$$\text{clip\_ratio} : \{0.1, 0.2, 0.3\} \tag{2}$$

Результаты эксперимента представлены на рисунке (Рисунок 2).

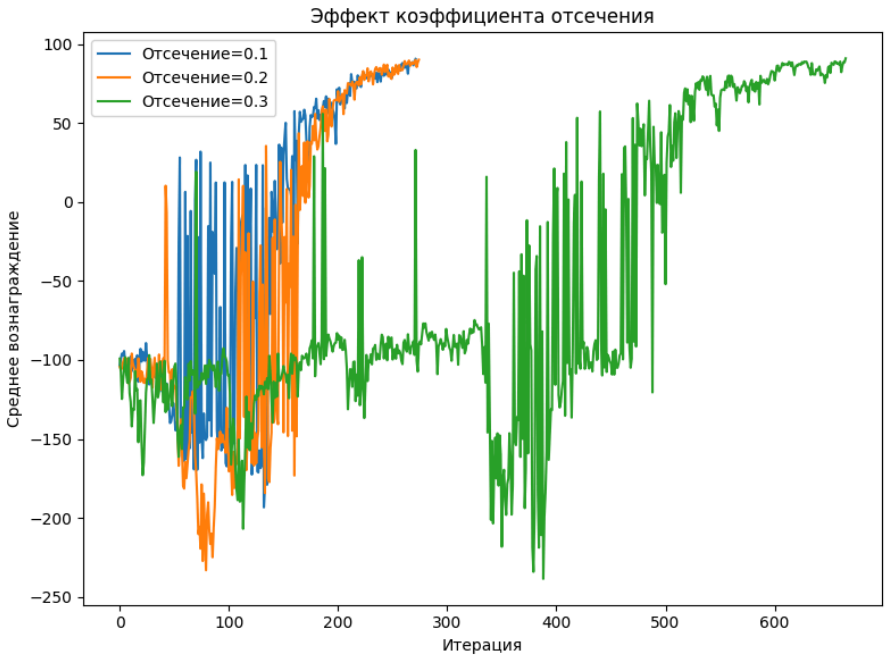

Рисунок 2 – Среднее вознаграждение в зависимости от коэффициента clip_ratio

По полученным результатам можно сделать следующие выводы:

- Значения clip_ratio равное 0.1 и 0.2 оказались наиболее удачными. В данных случаях модель достигла финального состояния в районе 300 итерации.
- Можно предположить, что на меньшем значении (0.1) модель будет работать лучше, ведь она начала выдавать лучшие результаты раньше, хоть и в результате дала приблизительно схожий результат к 300 итерации;
- Увеличение значения clip_ratio привело к ухудшению качества предсказания модели, из-за чего результат оказался наименее стабильным и смог достигнуть терминального состояния только на отметке около 700 итераций.

4. Сравнение обучения при разных количествах эпох

Для проведения эксперимента использовались следующие значения:

$$\text{epochs} : \{5, 10, 20\} \tag{3}$$

Результаты эксперимента представлены на рисунке (Рисунок 3).

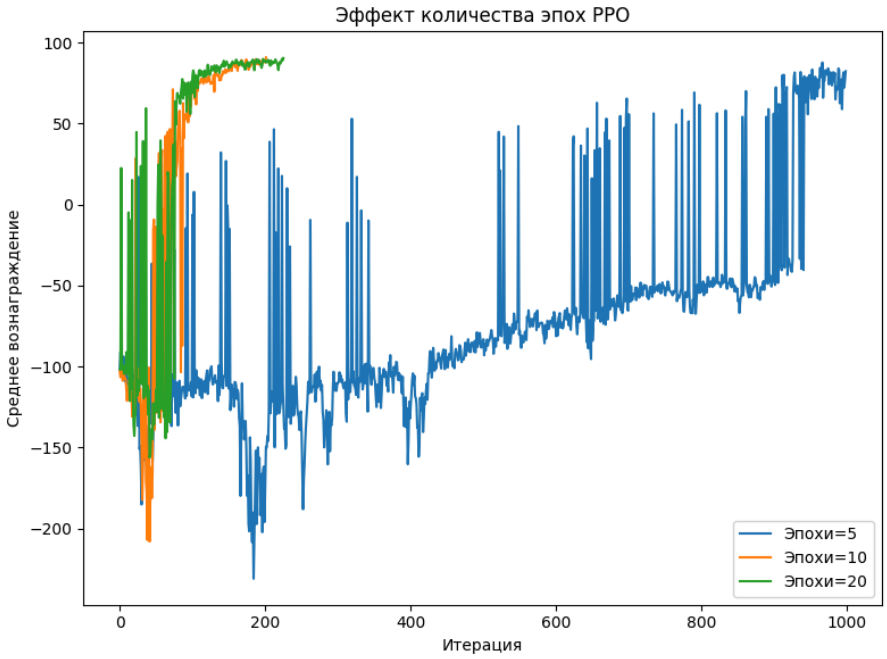

Рисунок 3 – Среднее вознаграждение в зависимости от количества эпох

По полученным результатам можно сделать следующие выводы:

- У значений 10 и 20 получились схожие результаты и модель в данном случае смогла достигнуть терминального состояния уже примерно на 220 итерации;
- Слишком малое значение (5) дало наихудший результат и модель в данном случае не смогла достигнуть терминального состояния по истечению 1000 итераций;
- Можно предположить, что оптимальное значение количества эпох может быть немногим больше значения 20.

5. Добавление нормализации преимуществ

Для нормализации преимуществ используется функция

$$\text{advantages}_{\text{norm}} = \frac{\text{advantages} - \mu_{\text{advantages}}}{\sigma_{\text{advantages}}} + 10^{-8} \tag{4}$$

Где μ и σ - среднее и стандартное отклонение соответственно.

Значение 10^{-8} используется для предотвращения деления на ноль при $\sigma = 0$.

После нормализации получились графики, представленные на рисунке (Рисунок 4).

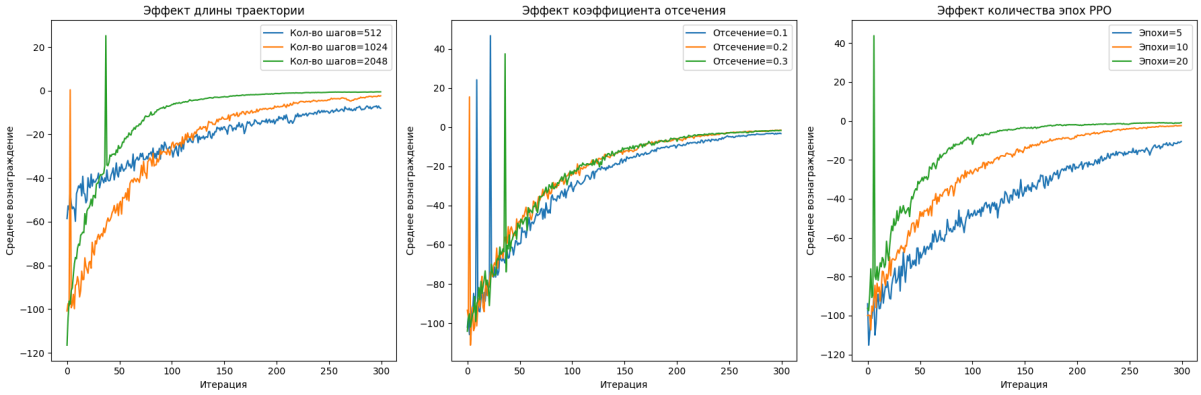

Рисунок 4 – Результаты экспериментов с нормализацией преимуществ

Как можно заметить, среднее значение вознаграждения сместилось ближе к 0.

По полученным графика можно сделать следующие выводы:

- Длина траектории steps
  - Результаты с нормализованными преимуществами получились совершенно иными. Получилось чем больше, тем лучше;
  - steps $= 512$ дало самый худший результат. Тенденция развития в среднем похожа на линейную;
  - steps $= 1024$ результат средний, тенденция развития схожа с логарифмической функцией;
  - steps $= 2048$ наилучший результат, тенденция развития стала значительно похожей на логарифмическую функцию и при этом обладает наибольшей стабильностью.
- Коэффициент clip_ratio
  - Наименьшее значение коэффициента по-прежнему дает худший результат;

    - При этом значения $0.2, 0.3$ дают практически одинаковые результаты. На основе этого можно предположить, что усовершенствование будет происходить до определенного предела при увеличении значения коэффициента. Далее модель либо будет упираться в определенный предел, либо будет ухудшаться.
- Количество эпох epochs
    - Чем больше количество эпох, тем лучше модель;
    - Соответственно, модель со значением эпох равным 5 дает худший результат, а с 20 - лучший.

ВЫВОДЫ

В ходе лабораторной работы был реализован алгоритм PPO для обучения агента в среде MountainCarContinuous-v0. Проведенные эксперименты позволили сделать следующие выводы:

1. Оптимальная длина траектории составляет 1024 шагов, так как она обеспечивает быстрое достижение терминального состояния и стабильное обучение модели.
2. Коэффициент $\text{clip\_ratio} = 0.2$ оказался наиболее подходящим, обеспечивая баланс между стабильностью и скоростью обучения.
3. Увеличение количества эпох положительно влияет на качество обучения, при этом значение 20 эпох показало наилучший результат.
4. Нормализация преимуществ улучшает обучение, смещая среднее вознаграждение ближе к нулю. После нормализации длинные траектории (2048 шагов) стали более предпочтительными, а влияние коэффициента clip_ratio стабилизировалось.

Таким образом, оптимизация параметров и добавление нормализации преимуществ позволили улучшить качество обучения агента.

ПРИЛОЖЕНИЕ А

Исходный код

```
import os

import gymnasium as gym
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Normal
from torch.utils.tensorboard.writer import SummaryWriter
from tqdm import tqdm

env_name = "MountainCarContinuous-v0"

# Настройка TensorBoard для графиков
writer = SummaryWriter(log_dir=f"runs/ppo_{env_name}")

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

num_iterations = 999
num_steps = 1024
ppo_epochs = 10
mini_batch_size = 256
gamma = 0.99
clip_ratio = 0.2
value_coef = 0.5
entropy_coef = 0.01
lr = 3e-4


# Значения для экспериментов
reward_history = []
epoch_variants = [5, 10, 20]
clip_ratio_variants = [0.1, 0.2, 0.3]
steps_variants = [512, 1024, 2048]


class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size=64):
        super(Actor, self).__init__()
        self.shared_net = nn.Sequential(
            nn.Linear(state_dim, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, hidden_size),
            nn.Tanh(),
        )
        self.mean_net = nn.Linear(hidden_size, action_dim)
        self.log_std = nn.Parameter(torch.zeros(action_dim))

    def forward(self, x):
        shared_features = self.shared_net(x)
        mean = self.mean_net(shared_features)
        return mean, self.log_std.exp()
```

```
    def get_distribution(self, state):
        mean, std = self.forward(state)
        dist = Normal(mean, std)
        return dist

    def act(self, state):
        state = torch.FloatTensor(state).unsqueeze(0).to(device)
        with torch.no_grad():
            dist = self.get_distribution(state)
            action = dist.sample()
            log_prob = dist.log_prob(action).sum(dim=-1)
        return action.cpu().numpy().flatten(), log_prob.item()


class Critic(nn.Module):
    def __init__(self, state_dim, hidden_size=64):
        super(Critic, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, 1),
        )

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


def collect_trajectories(policy, num_steps):
    env = gym.make(
        env_name,
        render_mode=None,
    )

    states = []
    actions = []
    log_probs = []
    rewards = []
    dones = []
    episode_rewards = []

    state, _ = env.reset()
    ep_reward = 0.0

    for _ in range(num_steps):
        action, log_prob = policy.act(state)

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

        states.append(state)
        actions.append(action)
        log_probs.append(log_prob)
        rewards.append(reward)
        dones.append(done)

        state = next_state
```

```
            ep_reward += float(reward)

            if done:
                state, _ = env.reset()
                episode_rewards.append(ep_reward)
                ep_reward = 0.0

        # Если эпизод не закончился на последнем шаге, добавляем частичное вознаграждение за эпизод
    if len(episode_rewards) == 0 or ep_reward > 0:
        episode_rewards.append(ep_reward)

    return {
        "states": np.array(states),
        "actions": np.array(actions),
        "log_probs": np.array(log_probs),
        "rewards": np.array(rewards),
        "dones": np.array(dones),
        "episode_rewards": np.array(episode_rewards),
    }


def compute_returns_and_advantages(rewards, dones, values, normalize_advantages=True):
    returns = []
    advantages = []
    R = 0.0

    for reward, done, value in zip(
        reversed(rewards), reversed(dones), reversed(values)
    ):
        if done:
            R = 0.0

        R = reward + gamma * R
        returns.insert(0, R)
        adv = R - value
        advantages.insert(0, adv)

    returns = np.array(returns)
    advantages = np.array(advantages)

    returns = (returns - returns.mean()) / (returns.std() + 1e-8)

    # Нормализация преимуществ (настраиваемая)
    if normalize_advantages:
        advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

    return returns, advantages


def train_ppo(
    actor: Actor,
    critic: Critic,
    num_iterations: int,
    num_steps: int,
    ppo_epochs: int,
    clip_ratio: float,
```

```
    train_name: str,
    normalize_advantages: bool = True,
):
    actor_optimizer = optim.Adam(actor.parameters(), lr=lr)
    critic_optimizer = optim.Adam(critic.parameters(), lr=lr)

    iteration_rewards = []

    for iteration in tqdm(range(num_iterations)):
        batch = collect_trajectories(actor, num_steps)

        states = torch.FloatTensor(batch["states"]).to(device)
        actions = torch.FloatTensor(batch["actions"]).to(device)
        old_log_probs = torch.FloatTensor(batch["log_probs"]).to(device)

        with torch.no_grad():
            values = critic(states).squeeze().cpu().numpy()

        returns, advantages = compute_returns_and_advantages(
            batch["rewards"], batch["dones"], values, normalize_advantages
        )

        returns = torch.FloatTensor(returns).to(device)
        advantages = torch.FloatTensor(advantages).to(device)

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

        loss_value = 0.0

        for epoch in range(ppo_epochs):
            np.random.shuffle(indices)

            for start in range(0, dataset_size, mini_batch_size):
                end = start + mini_batch_size
                if end > dataset_size:
                    end = dataset_size

                mini_indices = indices[start:end]

                mini_states = states[mini_indices]
                mini_actions = actions[mini_indices]
                mini_old_log_probs = old_log_probs[mini_indices]
                mini_returns = returns[mini_indices]
                mini_advantages = advantages[mini_indices]

                dist = actor.get_distribution(mini_states)
                new_log_probs = dist.log_prob(mini_actions).sum(dim=-1)

                # Основной алгоритм PPO
                ratio = torch.exp(new_log_probs - mini_old_log_probs)

                surrogate1 = ratio * mini_advantages
                surrogate2 = (
                            torch.clamp(ratio, 1 - clip_ratio, 1 + clip_ratio)
* mini_advantages
                )
```

```
                actor_loss = -torch.min(surrogate1, surrogate2).mean()

                entropy_loss = dist.entropy().mean()
                value_estimates = critic(mini_states).squeeze()

              critic_loss = (mini_returns - value_estimates).pow(2).mean()  # MSE

                current_loss = (
                          actor_loss + value_coef * critic_loss - entropy_coef
* entropy_loss
                )

                actor_optimizer.zero_grad()
                critic_optimizer.zero_grad()
                current_loss.backward()
                actor_optimizer.step()
                critic_optimizer.step()

                loss_value = current_loss.item()

                writer.add_scalar(
                    f"Loss/Actor_{train_name}", actor_loss.item(), iteration
                )

        avg_reward = np.mean(batch["episode_rewards"])

        writer.add_scalar(f"Reward/{train_name}", avg_reward, iteration)

        iteration_rewards.append(avg_reward)
        # print(
           #       f"Итерация {iteration}: Loss = {loss_value:.4f}, Avg Reward
= {avg_reward:.2f}"
        # )

        if avg_reward >= 90:
            print("Задача выполнена!")
            break

    return iteration_rewards


def run_experiments():
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))

    exp_env = gym.make(env_name)

    state_dim = 2
    action_dim = 1

    # 1. Эксперимент с различными длинами траекторий
    rewards_by_steps = []

    for steps in steps_variants:
        print(f"\nЗапуск эксперимента с {steps} шагов на траектории")
        actor = Actor(2, 1).to(device)
        critic = Critic(state_dim).to(device)

        rewards = train_ppo(
```

```
        actor,
        critic,
        num_iterations=num_iterations,
        num_steps=steps,
        ppo_epochs=10,
        clip_ratio=0.2,
        normalize_advantages=True,
        train_name=f"steps_{steps}",
    )
    rewards_by_steps.append(rewards)

for i, steps in enumerate(steps_variants):
    axs[0].plot(rewards_by_steps[i], label=f"Кол-во шагов={steps}")
axs[0].set_title("Эффект длины траектории")
axs[0].set_xlabel("Итерация")
axs[0].set_ylabel("Среднее вознаграждение")
axs[0].legend()

# 2. Эксперимент с различными коэффициентами отсечения
rewards_by_clip = []
for clip in clip_ratio_variants:
    print(f"\nЗапуск эксперимента с коэффициентом отсечения {clip}")
    actor = Actor(state_dim, action_dim).to(device)
    critic = Critic(state_dim).to(device)

    rewards = train_ppo(
        actor,
        critic,
        num_iterations=num_iterations,
        num_steps=1024,
        ppo_epochs=10,
        clip_ratio=clip,
        normalize_advantages=True,
        train_name=f"clip_{clip}",
    )
    rewards_by_clip.append(rewards)

for i, clip in enumerate(clip_ratio_variants):
    axs[1].plot(rewards_by_clip[i], label=f"Отсечение={clip}")
axs[1].set_title("Эффект коэффициента отсечения")
axs[1].set_xlabel("Итерация")
axs[1].set_ylabel("Среднее вознаграждение")
axs[1].legend()

# 3. Эксперимент с различным количеством эпох
rewards_by_epochs = []
for epochs in epoch_variants:
    print(f"\nЗапуск эксперимента с {epochs} эпохами PPO")
    actor = Actor(state_dim, action_dim).to(device)
    critic = Critic(state_dim).to(device)

    rewards = train_ppo(
        actor,
        critic,
        num_iterations=num_iterations,
        num_steps=1024,
        ppo_epochs=epochs,
        clip_ratio=0.2,
```

```
            normalize_advantages=True,
            train_name=f"epochs_{epochs}",
        )
        rewards_by_epochs.append(rewards)

    for i, epochs in enumerate(epoch_variants):
        axs[2].plot(rewards_by_epochs[i], label=f"Эпохи={epochs}")
    axs[2].set_title("Эффект количества эпох PPO")
    axs[2].set_xlabel("Итерация")
    axs[2].set_ylabel("Среднее вознаграждение")
    axs[2].legend()

    plt.tight_layout()

    # Создаем директорию для сохранения графиков, если она не существует

    os.makedirs("./fig", exist_ok=True)

    # Сохраняем скомбинированный график
    plt.savefig("./fig/ppo_experiments.png")
    print(
                "Сохранены комбинированные результаты экспериментов в fig/
ppo_experiments.png"
    )

    # Сохраняем отдельные графики
    for i, title in enumerate(["trajectory_length", "clip_ratio", "ppo_epochs"]):
        plt.figure(figsize=(8, 6))

        if i == 0:  # График длины траектории
            for j, steps in enumerate(steps_variants):
                plt.plot(rewards_by_steps[j], label=f"Шаги={steps}")
            plt.title("Эффект длины траектории")
        elif i == 1:  # График коэффициента отсечения
            for j, clip in enumerate(clip_ratio_variants):
                plt.plot(rewards_by_clip[j], label=f"Отсечение={clip}")
            plt.title("Эффект коэффициента отсечения")
        else:  # График эпох PPO
            for j, epochs in enumerate(epoch_variants):
                plt.plot(rewards_by_epochs[j], label=f"Эпохи={epochs}")
            plt.title("Эффект количества эпох PPO")

        plt.xlabel("Итерация")
        plt.ylabel("Среднее вознаграждение")
        plt.legend()
        plt.tight_layout()

        # Сохраняем отдельный график
        plt.savefig(f"./fig/ppo_{title}.png")
        print(f"Сохранены результаты эксперимента {title} в fig/ppo_{title}.png")

    plt.show()


def main():
    # Инициализируем случайные семена для воспроизводимости
    torch.manual_seed(42)
    np.random.seed(42)
```

```
    # Создаем новую среду для тестирования
    test_env = gym.make(env_name)

    print(f"Окружение: {env_name}")
    print(f"Пространство наблюдений: {test_env.observation_space}")
    print(f"Пространство действий: {test_env.action_space}")

    if (
        test_env.observation_space is not None
        and hasattr(test_env.observation_space, "shape")
        and test_env.observation_space.shape is not None
    ):
        state_dim = test_env.observation_space.shape[0]
    else:
        state_dim = 2  # Дефолтное значение для MountainCarContinuous-v0

    if (
        test_env.action_space is not None
        and hasattr(test_env.action_space, "shape")
        and test_env.action_space.shape is not None
    ):
        action_dim = test_env.action_space.shape[0]
    else:
        action_dim = 1  # Дефолтное значение для MountainCarContinuous-v0

    actor = Actor(state_dim, action_dim).to(device)
    critic = Critic(state_dim).to(device)

    # Тренировка модели с параметрами по умолчанию
    print("\nТренировка с параметрами по умолчанию:")
    train_ppo(
        actor,
        critic,
        num_iterations,
        num_steps,
        ppo_epochs,
        clip_ratio,
        normalize_advantages=True,
        train_name="default",
    )

    # Сохранение обученной модели
    torch.save(
        {
            "actor_state_dict": actor.state_dict(),
            "critic_state_dict": critic.state_dict(),
        },
        "./ppo_mountain_car_model.pth",
    )

    # Запуск экспериментов для сравнения различных параметров
    run_experiments()


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