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 _____ Панкина В. К.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург
2025

Цель работы.

Исследование влияния различных гиперпараметров алгоритма PPO на производительность агента в среде MountainCarContinuous-v0.

Постановка задачи.

1. Реализовать и исследовать алгоритм Proximal Policy Optimization (PPO) для решения задачи управления в среде MountainCarContinuous-v0.
2. Изменение длины траектории и оценка его влияния на процесс обучения.
3. Подбор оптимального коэффициента отсечения для алгоритма PPO в данной среде.
4. Анализ влияния нормализации преимуществ на стабильность и скорость обучения.
5. Исследование влияния количества эпох обучения на сходимость алгоритма.

Выполнение задач.

1. Реализация PRO.

Для этого были разработаны ключевые компоненты: Actor, Critic, функция сбора траекторий и функция обучения. Actor и Critic представляют собой нейронные сети, построенные с использованием PyTorch, с полносвязными слоями и функцией активации Tanh.

Функция collect_trajectories собирала опыт взаимодействия агента с окружением, сохраняя состояния, действия, награды и т.д. Функция compute_returns_advantages вычисляла returns и advantages для каждой точки траектории. Ключевая функция train обновляла параметры Actor и Critic на основе собранных данных, используя clipping для стабильности и энтропийный

штраф для исследования. В итоге получился работающий PPO, готовый к экспериментам.

Дефолтные данные (гиперпараметры):

- env_name = "MountainCarContinuous-v0" (Используемая среда)
- num_iterations = 999 (Максимальное количество итераций обучения)
- num_steps = 2048 (Длина траектории, собираемая за итерацию)
- ppo_epochs = 10 (Количество эпох обучения на одной траектории)
- mini_batch_size = 64 (Размер мини-батча при обучении)
- gamma = 0.99 (Коэффициент дисконтирования)
- clip_ratio = 0.2 (Коэффициент отсечения для PPO)
- value_coef = 0.5 (Коэффициент для потерь Critic)
- entropy_coef = 0.01 (Коэффициент для энтропийного бонуса)
- lr = 3e-4 (Скорость обучения)
- normalize_advantages = False (Отключена нормализация преимуществ в дефолтном запуске)

2. Изменение длины траектории (*num_steps*).

В рамках данной задачи было исследовано влияние длины траектории на процесс обучения алгоритма PPO в среде MountainCarContinuous-v0. Длина траектории определяет, сколько шагов взаимодействия агента со средой собирается в каждом цикле обучения. Эксперименты проводились с тремя различными значениями *num_steps*: 1024, 2048 и 4096. Все остальные гиперпараметры были зафиксированы на дефолтных значениях. Для каждого значения *num_steps* проводилось обучение PPO до достижения максимального количества итераций (*num_iterations* = 999) или достижения целевой награды (90). После завершения обучения для каждого значения *num_steps* строился график зависимости средней награды за эпизод от номера итерации. На рисунке

1 представлен график зависимости средней награды от итерации обучения для различных значений параметра *num_steps*

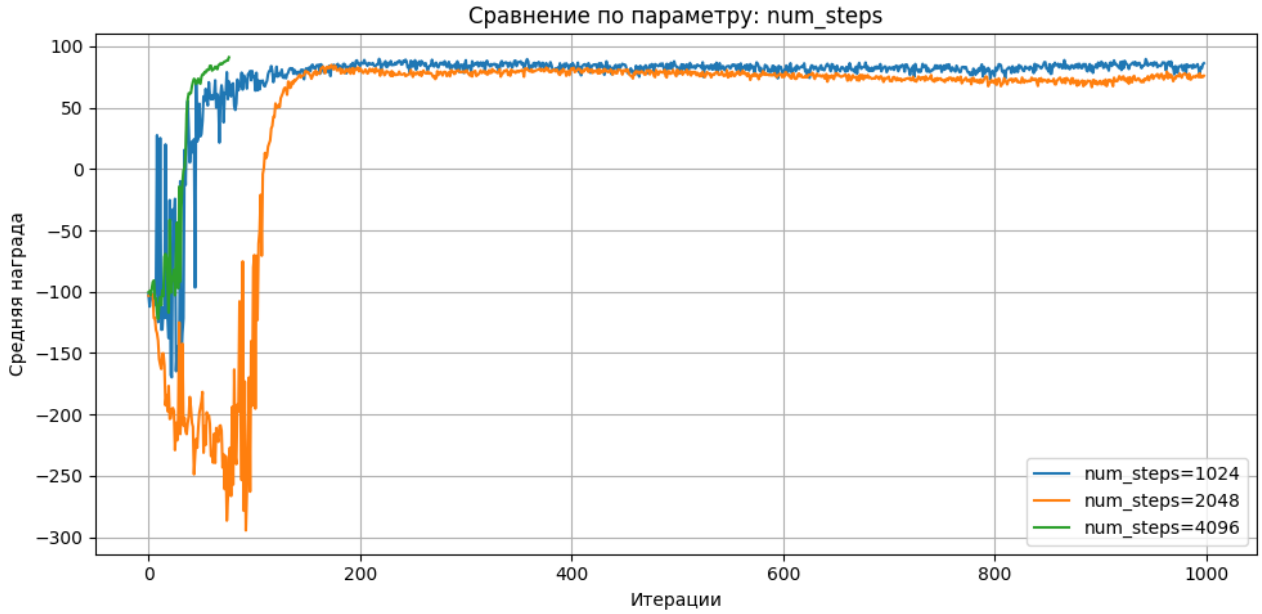

Рисунок 1. Зависимость средней награды от итерации обучения при различных значениях длины траектории

На рис. 1 видно, что *num_steps*=4096 (зеленая линия) обеспечивает самую быструю сходимость, что говорит о более эффективном обучении. *num_steps*=1024 (синяя линия) учится медленнее и менее стабильно в начале. *num_steps*=2048 (оранжевая линия) показывает плохие результаты на старте, а затем выходит на уровень *num_steps*=1024, но всё равно хуже *num_steps*=4096.

Таким образом увеличение *num_steps* положительно влияет на обучение PPO в MountainCarContinuous-v0. *num_steps*=4096 показал наилучший результат.

3. Подбор оптимального коэффициента отсечения (*clip_ratio*)

Коэффициент отсечения является ключевым гиперпараметром PPO, который ограничивает изменение политики на каждом шаге обновления, обеспечивая стабильность обучения. Эксперименты проводились с тремя значениями *clip_ratio*: 0.1, 0.2 и 0.3. Все остальные гиперпараметры были зафиксированы на дефолтных значениях. На рисунке 2 представлен график зависимости средней награды от итерации обучения для различных значений коэффициента отсечения.

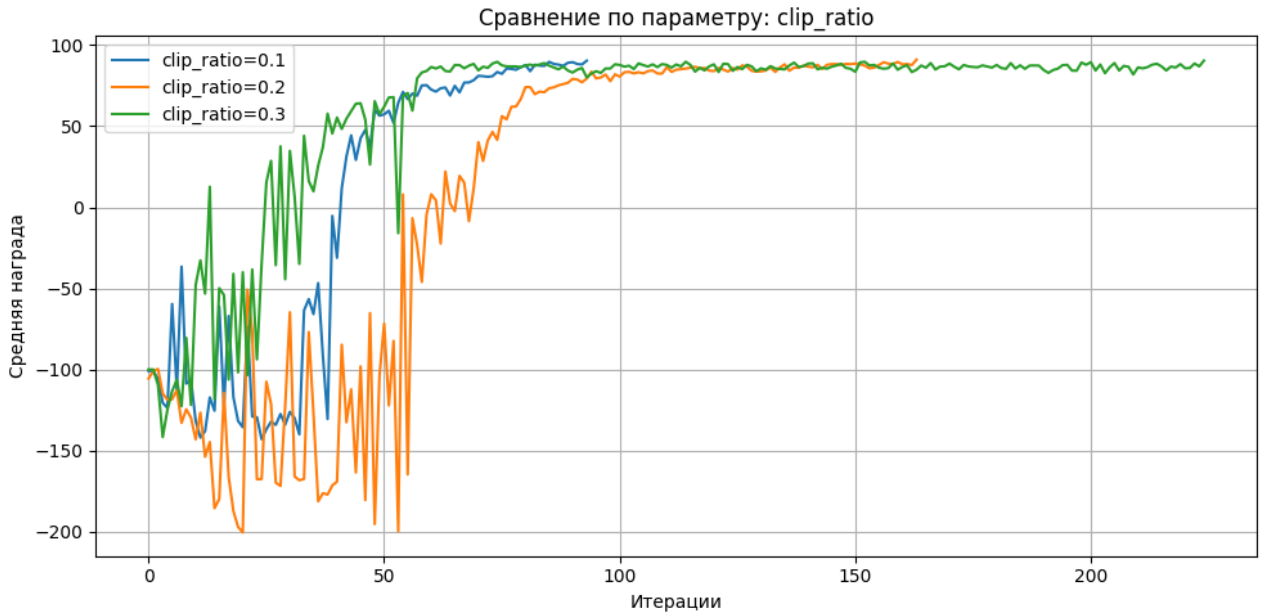

Рисунок 2. Зависимость средней награды от итерации обучения при различных значениях коэффициента отсечения (*clip_ratio*).

При анализе графика видно, что *clip_ratio*=0.3 имеет быстрый старт, но требуется больше итераций, чтобы достичь стабильного высокого результата. *clip_ratio* = 0.1 демонстрирует самую быструю общую сходимость к стабильному, высокому уровню награды. Хотя начальный прогресс может быть медленнее, чем у *clip_ratio* = 0.3, он достигает конечной цели быстрее. *clip_ratio* = 0.2 приводит к наибольшим колебаниям средней награды на ранних этапах обучения, достигает наивысшей награды раньше, чем *clip_ratio* = 0.3, но позже чем *clip_ratio* = 0.1. Таким образом, увеличение значения *clip_ratio* приводит к ухудшению стабильности обучения, а меньшее значение, хоть и медленнее на старте, обеспечивает более быструю и уверенную сходимость.

4. Добавление нормализации преимуществ (*normalize_advantages*).

Нормализация преимуществ заключается в вычитании среднего значения и делении на стандартное отклонение, что приводит преимущества к диапазону со средним 0 и стандартным отклонением 1. Это может улучшить стабильность обучения, уменьшив влияние различных масштабов преимуществ.

Эксперименты проводились с двумя значениями *normalize_advantages*: True (нормализация включена) и False (нормализация выключена). Все

остальные гиперпараметры были зафиксированы на дефолтных значениях. Для каждого значения проводилось обучение PPO до достижения максимального количества итераций или достижения целевой награды. После завершения обучения для каждого значения строился график зависимости средней награды за эпизод от номера итерации. На рисунке 3 представлен график зависимости средней награды от итерации обучения для двух случаев: с включенной нормализацией преимуществ (normalize_advantages = True) и с выключенной нормализацией преимуществ (normalize_advantages = False).

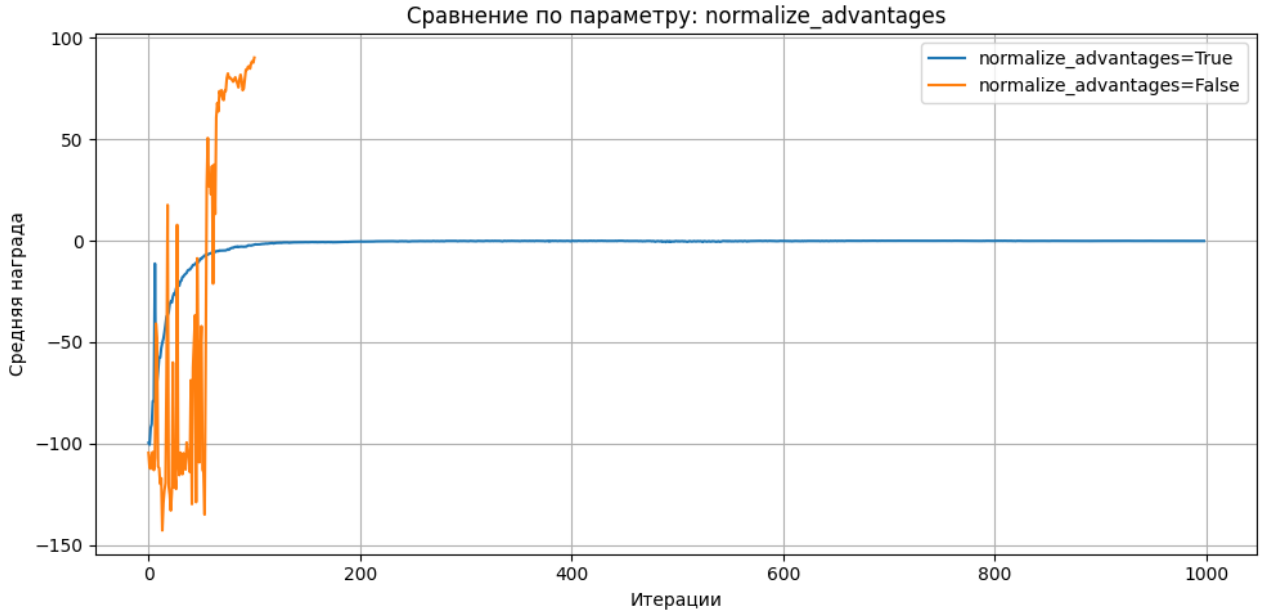

Рисунок 3. Зависимость средней награды от итерации обучения при включенной и выключенной нормализации преимуществ

Включение нормализации преимуществ приводит к более стабильному обучению и плавному росту средней награды, избегая резких падений в отрицательную область. Отключение нормализации приводит к нестабильному поведению на старте.

5. Изменение количества эпох обучения (*ppo_epochs*).

Параметр *ppo_epochs* определяет, сколько раз мини-батчи из собранной траектории используются для обновления параметров Actor и Critic. Слишком малое количество эпох может привести к недостаточному обучению на

собранных данных, в то время как слишком большое количество эпох может привести к переобучению и потере стабильности.

Эксперименты проводились с тремя значениями *ppo_epochs*: 5, 10 и 20. Все остальные гиперпараметры были зафиксированы на дефолтных значениях. На рисунке 4 представлен график, показывающий влияние различных значений *ppo_epochs* на обучение.

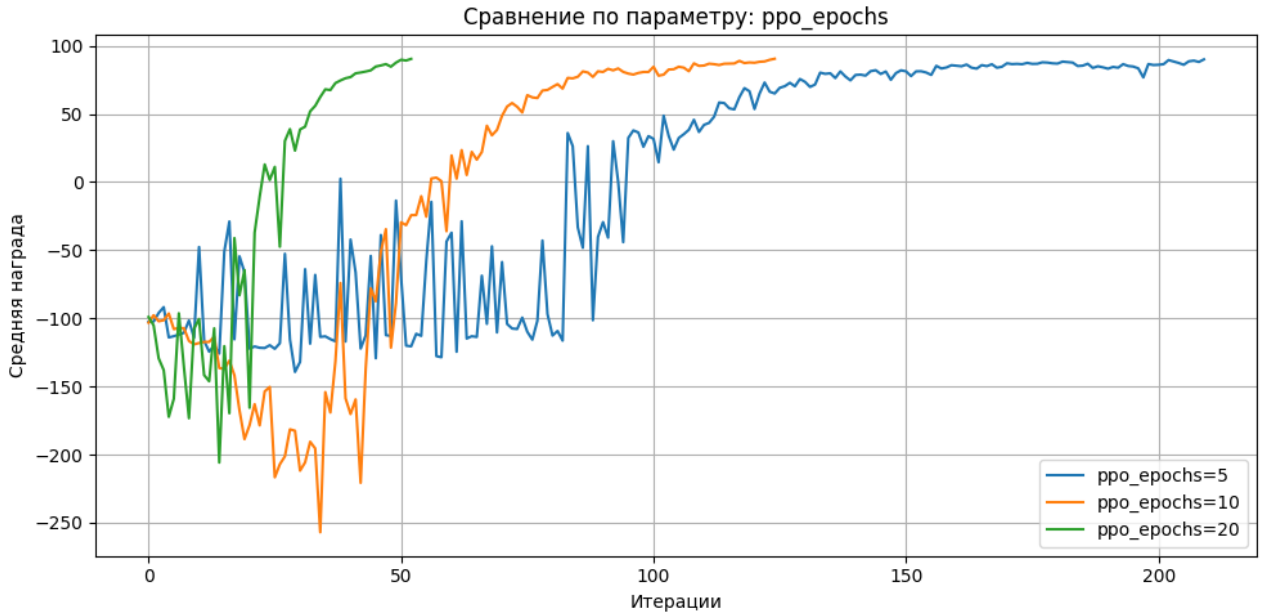

Рисунок 4. Зависимость средней награды от итерации обучения при различных значениях количества эпох обучения

На графике видно, что *ppo_epochs*=20 обеспечивает самую быструю начальную сходимость и высокие награды, *ppo_epochs*=10 показывает неплохие результаты, приближаясь к уровню *ppo_epochs*=20, а *ppo_epochs*=5 приводит к нестабильному и медленному обучению, что указывает на недостаточное использование данных. *ppo_epochs* = 20 - самый эффективный параметр. Можно сделать вывод, что увеличение *ppo_epochs* в данном случае приводит к улучшению результатов обучения. *ppo_epochs*=20 демонстрирует наилучшую производительность, обеспечивая самую быструю сходимость и высокие значения средней награды.

Заключение.

В ходе выполнения лабораторной работы был реализован алгоритм Proximal Policy Optimization (PPO) и исследовано влияние ключевых гиперпараметров на его производительность в среде MountainCarContinuous-v0. величение длины траектории (num_steps) улучшает обучение. Меньший clip_ratio (0.1) обеспечивает лучшую сходимость. Увеличение количества эпох (ppo_epochs) до 20 улучшило результаты. Тщательная настройка гиперпараметров критически важна для эффективности PPO.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
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 tqdm import tqdm

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

plot_dir = "plots_pro"
os.makedirs(plot_dir, exist_ok=True)

env_name = "MountainCarContinuous-v0"
num_iterations = 999
num_steps = 2048
ppo_epochs = 10
mini_batch_size = 64
gamma = 0.99
clip_ratio = 0.2
value_coef = 0.5
entropy_coef = 0.01
lr = 3e-4


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

```
        self.mean = nn.Linear(hidden_size, action_dim)
        self.log_std = nn.Parameter(torch.zeros(action_dim))

    def forward(self, x):
        features = self.net(x)
        mean = self.mean(features)
        return mean, self.log_std.exp()

    def get_dist(self, state):
        mean, std = self.forward(state)
        return Normal(mean, std)

    def act(self, state):
        state = torch.FloatTensor(state).unsqueeze(0).to(device)
        with torch.no_grad():
            dist = self.get_dist(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)
    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

    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_advantages(rewards, dones, values, normalize_ad-
vantages=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)
        advantages.insert(0, R - value)

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

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

    return returns, advantages


def train(env, actor, critic, num_iterations, num_steps, ppo_epochs,
clip_ratio, normalize_advantages):
    actor_optimizer = optim.Adam(actor.parameters(), lr=lr)
    critic_optimizer = optim.Adam(critic.parameters(), lr=lr)
    all_avg_rewards = []

    for i 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_advantages(batch["re-
wards"], 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)

        for epoch in range(ppo_epochs):
            np.random.shuffle(indices)
            for start in range(0, dataset_size, mini_batch_size):
                end = min(start + mini_batch_size, 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_dist(mini_states)
                new_log_probs = dist.log_prob(mini_actions).sum(dim=-1)
                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_esti-
mates).pow(2).mean()

                loss = actor_loss + value_coef * critic_loss - en-
tropy_coef * entropy_loss

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

        avg_reward = np.mean(batch["episode_rewards"])
        # print(f"Iteration {i + 1}: avg_reward = {avg_reward:.2f}")
        all_avg_rewards.append(avg_reward)
```

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

    return all_avg_rewards


def run_experiment(param_name, param_values, param_label,
**train_kwargs):
    all_results = {}

    for value in param_values:
        env = gym.make(env_name)
        state_dim = env.observation_space.shape[0]
        action_dim = env.action_space.shape[0]
        actor = Actor(state_dim, action_dim).to(device)
        critic = Critic(state_dim).to(device)

        kwargs = train_kwargs.copy()
        kwargs[param_name] = value
        rewards = train(env, actor, critic, **kwargs)
        all_results[str(value)] = rewards

    plt.figure(figsize=(10, 5))
    for label, rewards in all_results.items():
        plt.plot(rewards, label=f"{param_label}={label}")
    plt.xlabel("Итерации")
    plt.ylabel("Средняя награда")
    plt.title(f"Сравнение по параметру: {param_label}")
    plt.legend()
    plt.grid()
    plt.tight_layout()

    filename = f"{param_label}.png"
    filepath = os.path.join(plot_dir, filename)
    plt.savefig(filepath)
    plt.show()
```

```
run_experiment("num_steps", [1024, 2048, 4096], "num_steps",
               num_iterations=num_iterations,
               ppo_epochs=ppo_epochs,
               clip_ratio=clip_ratio,
               normalize_advantages=False)

run_experiment("clip_ratio", [0.1, 0.2, 0.3], "clip_ratio",
               num_iterations=num_iterations,
               num_steps=num_steps,
               ppo_epochs=ppo_epochs,
               normalize_advantages=False)

run_experiment("normalize_advantages", [True, False], "normalize_ad-
vantages",
               num_iterations=num_iterations,
               num_steps=num_steps,
               ppo_epochs=ppo_epochs,
               clip_ratio=clip_ratio)

run_experiment("ppo_epochs", [5, 10, 20], "ppo_epochs",
               num_iterations=num_iterations,
               num_steps=num_steps,
               clip_ratio=clip_ratio,
               normalize_advantages=False)
```