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 DQN для среды MountainCarContinuous-v0

Студент гр. 0310 _____ Низовцов Р.С.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург

2025

Цель работы

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

Задача

Задания для эксперимента:

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

Выполнение работы

1)Алгоритм PPO реализован с использованием библиотеки PyTorch на языке Python. Код приведен в Приложении А.

За значения по умолчанию были взяты следующие показатели:

- Gamma – 0.99
- Clip_ratio – 0.2
- Value_coef – 0.5
- Entropy_coef – 0.01
- Num_iterations – 300
- Num_steps – 2048

Результат работы приведен на рис. 1:

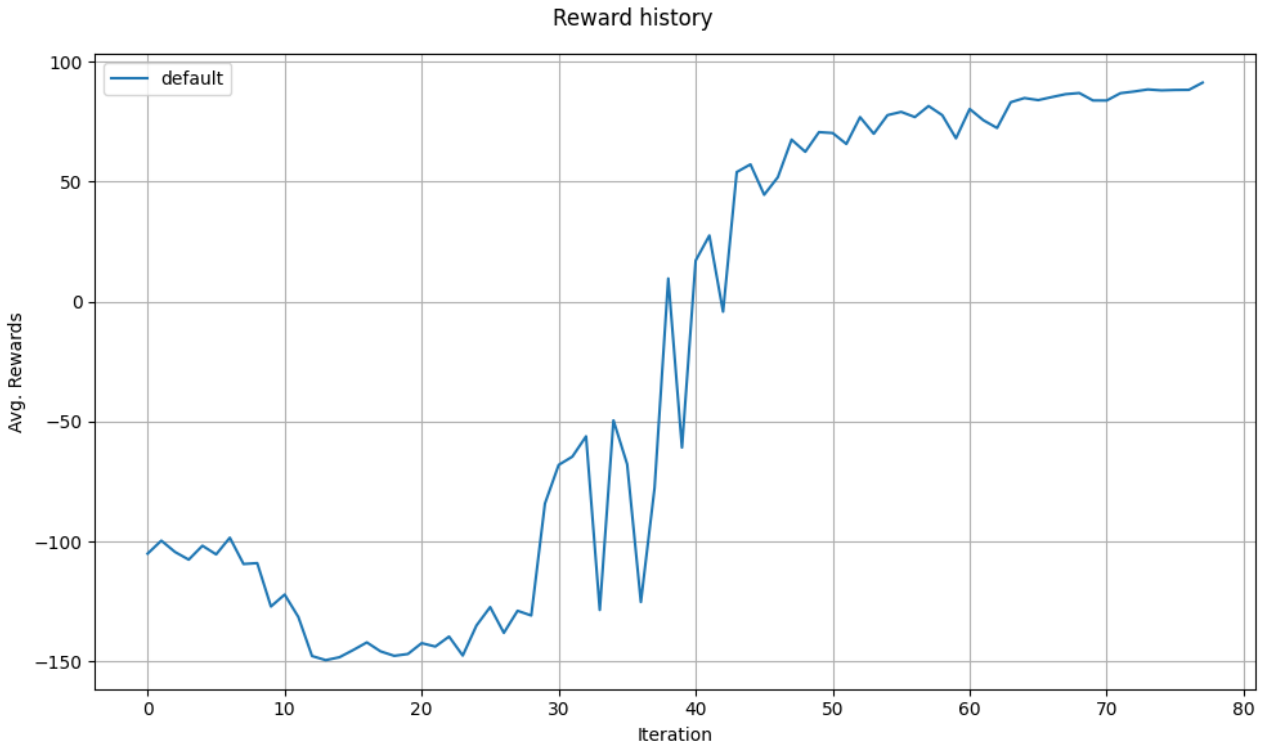

Рисунок 1 – Средняя награда при обучении на значениях по умолчанию

По графику можно увидеть, что алгоритм закончил работу раньше, достигнув более 90 значения награды примерно к 80 итерации.

2)Были протестированы несколько вариантов длин траектории:

- Steps – 512, 1024, 2048

Результат работы представлен на рис. 2:

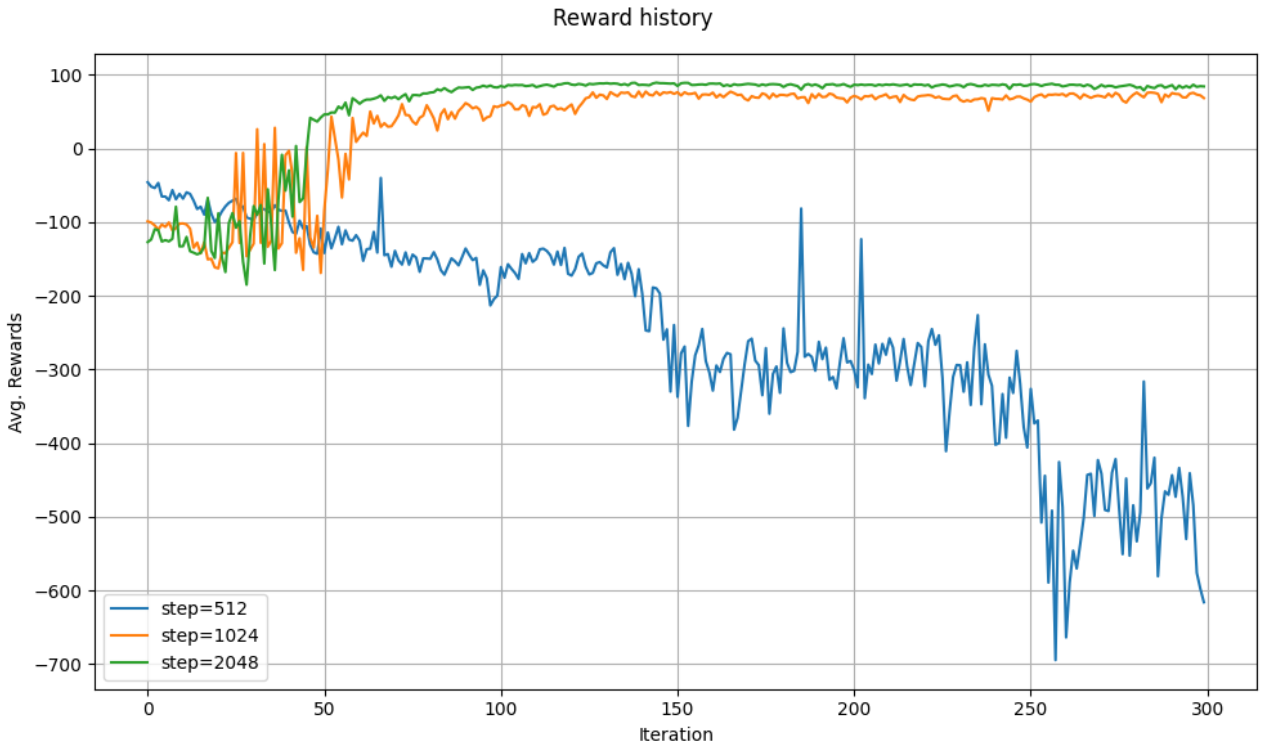

Рисунок 2 – Средняя награда при различных вариантах steps

Из графика можно отметить, что со временем при малом количестве шагов алгоритм начинает выдавать хуже результат с каждой итерацией. При больших же значениях steps алгоритм обучается сравнительно лучше.

3)Далее были протестированы различные значения clip_ratio для определения оптимального варианта:

- Clip_ratio - 0.1, 0.2, 0.3

На рис. 3 представлены результаты анализа:

Рисунок 3 – Средняя награда при различных вариантах clip_ratio

Из графика можно сделать вывод, что меньшее значение параметра приводит к лучшим результатам. Это видно из досрочного обучения алгоритма при значении равном 0,1.

4)Далее были добавлена нормализация преимуществ. Все параметры были взяты как значения по умолчанию. Результат представлен на рис. 4:

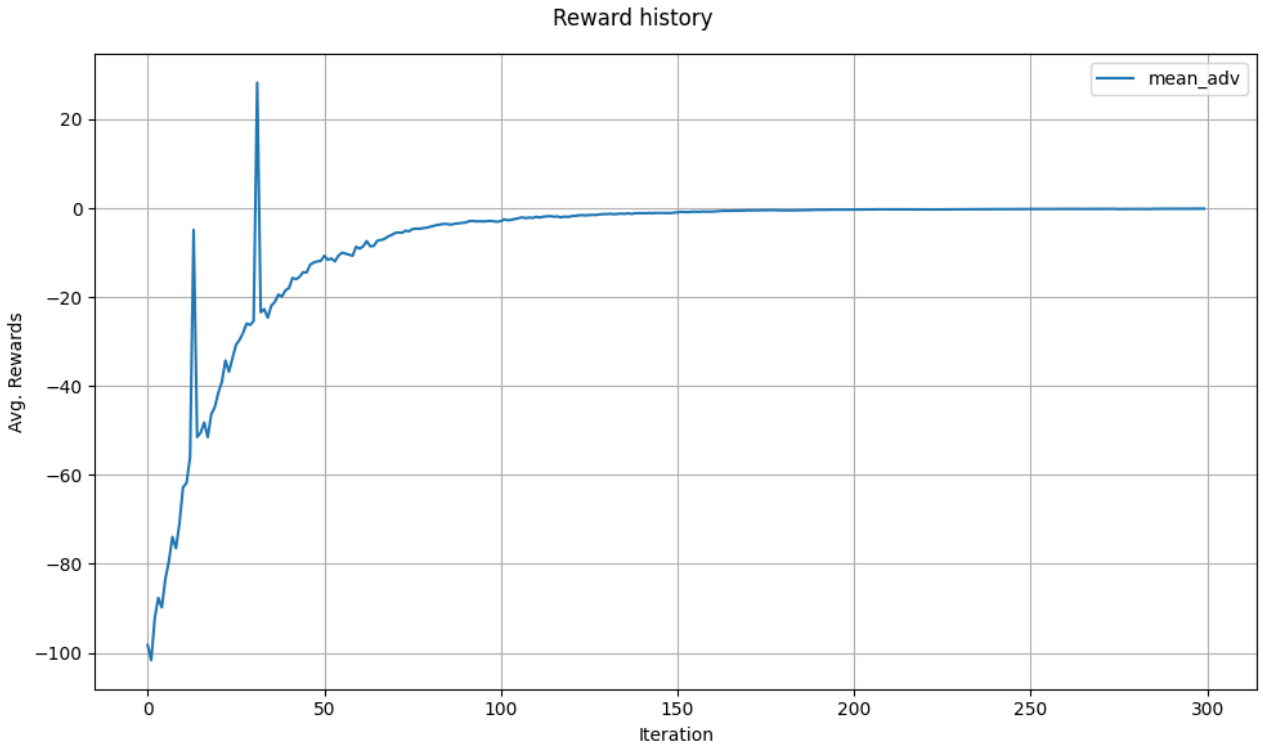

Рисунок 4 – Средняя награда при нормализации преимуществ

Как можно заметить, среднее значение вознаграждения сместилось ближе к 0, а сам график стал похож на логарифмическую функцию и при этом обладает большей стабильностью.

5)Далее были протестированы различные значения epoch:

- Epoch – 5, 10, 20

На рис. 5 представлены результаты анализа:

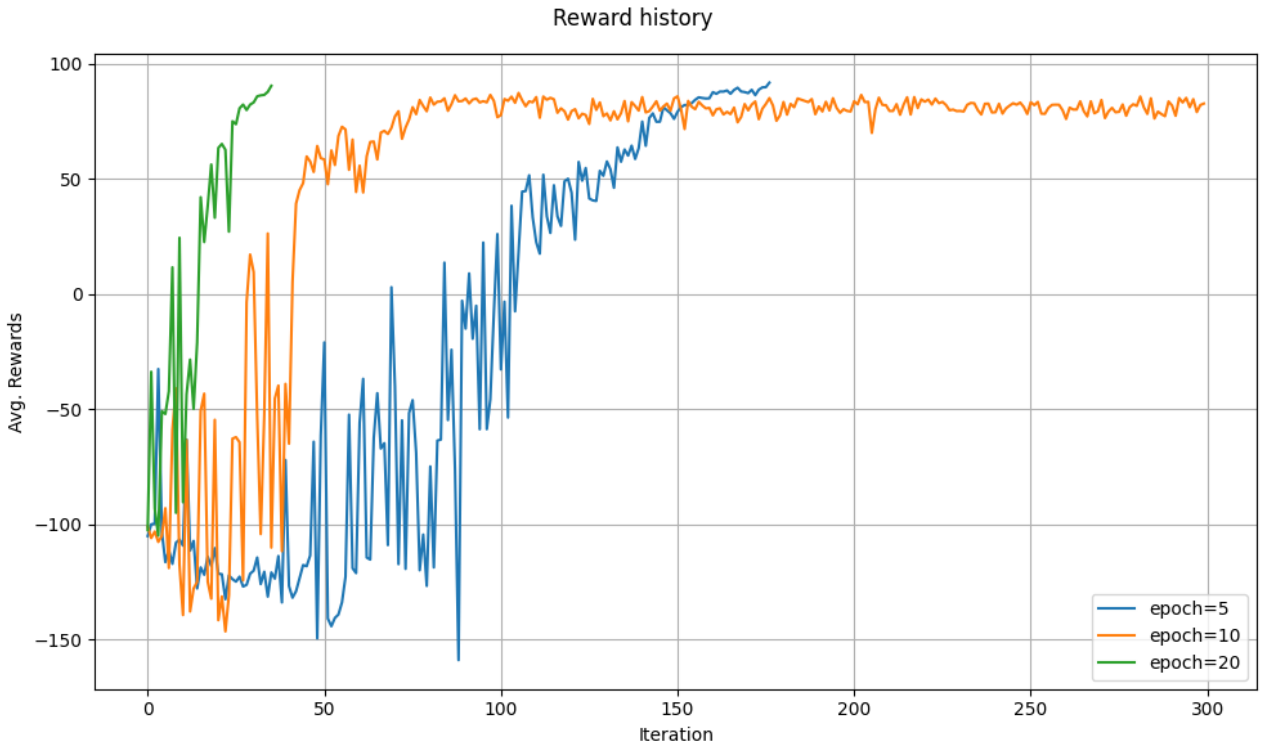

Рисунок 5 – Средняя награда при различных вариантах epoch

Из графика можно сделать вывод, что большее количество эпох позволяет обучать модель за меньшее количество итераций.

Выводы.

В ходе выполнения лабораторной работы был реализован алгоритм PPO и исследовано влияние различных параметров и архитектур нейросети на процесс обучения агента в среде MountainCarContinuous.

В результате было установлено, что для параметров steps и epoch следует выбирать большие значения, а для параметра clip_ratio для получения наилучших результатов нужно брать значения около 0,1. Нормализация преимуществ улучшает обучение, смещая среднее вознаграждение ближе к нулю.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРИЛОЖЕНИЯ

```
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")

default_params = {
    "env_name": "MountainCarContinuous-v0",
    "num_iterations": 300,
    "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_distribution(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_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(default_params["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_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 + default_params["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)

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

    return returns, advantages


class Experiment:
    def __init__(
        self,
        env: gym.Env,
        actor: Actor,
```

```
    critic: Critic,
    num_iterations: int = 1000,
    num_steps: int = 2048,
    ppo_epochs: int = 10,
    clip_ratio: float = 0.2,
    normalize_advantages: bool = True,
):
    self.env = env
    self.actor = actor
    self.critic = critic

    self.num_iterations = num_iterations
    self.num_steps = num_steps
    self.ppo_epochs = ppo_epochs
    self.clip_ratio = clip_ratio
    self.normalize_advantages = normalize_advantages

    self.rewards_history = []

def train_ppo(self):
    actor_optimizer = optim.Adam(self.actor.parameters(), lr=default_params["lr"])
    critic_optimizer = optim.Adam(self.critic.parameters(), lr=default_params["lr"])

    for _ in tqdm(range(self.num_iterations)):
        batch = collect_trajectories(self.actor, self.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 = self.critic(states).squeeze().cpu().numpy()

        returns, advantages = compute_returns_and_advantages(
            batch["rewards"], batch["dones"], values, self.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(self.ppo_epochs):
   np.random.shuffle(indices)

   for start in range(0, dataset_size, default_params["mini_batch_size"]):
      end = start + default_params["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 = self.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 - self.clip_ratio, 1 + self.clip_ratio) * mini_advantages
      )

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

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

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

```
                current_loss = (
                    actor_loss + default_params["value_coef"] * critic_loss - default_params[
                    "entropy_coef"] * entropy_loss
                )

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

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

            self.rewards_history.append(avg_reward)

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

        return

    def run(self):
        self.train_ppo()

def plot(results, file_name):
    fig, ax = plt.subplots(figsize=(10, 6))
    fig.suptitle("Reward history")

    for name, result in results.items():
        ax.plot(result["reward"], label=f"{name}")
        ax.set_xlabel("Iteration")
        ax.set_ylabel("Avg. Rewards")

    ax.legend()
    ax.grid()
    fig.tight_layout()
    fig.savefig(f"{file_name}_reward_history.png")
```

```
def run_default_experiment():
    env = gym.make(default_params["env_name"])
    results = {}

    print("Default experiment")

    state_dim = 2
    action_dim = 1

    actor = Actor(state_dim, action_dim).to(device)
    critic = Critic(state_dim).to(device)
    experiment = Experiment(
        env,
        actor,
        critic,
        default_params["num_iterations"],
        default_params["num_steps"],
        default_params["ppo_epochs"],
        default_params["clip_ratio"],
        normalize_advantages=False,
    )

    experiment.run()

    results["default"] = {
        "reward": experiment.rewards_history,
    }

    plot(results, file_name="default_experiment")


def run_steps_experiment():
    env = gym.make(default_params["env_name"])
    steps = [512, 1024, 2048]

    results = {}
```

```
    for idx in range(0, len(steps)):
        print(f"Step experiment: step={steps[idx]} [{idx + 1}/{len(steps)}]")

        state_dim = 2
        action_dim = 1

        actor = Actor(state_dim, action_dim).to(device)
        critic = Critic(state_dim).to(device)
        experiment = Experiment(
            env,
            actor,
            critic,
            default_params["num_iterations"],
            steps[idx],
            default_params["ppo_epochs"],
            default_params["clip_ratio"],
            normalize_advantages=False,
        )

        experiment.run()

        results[f"step={steps[idx]}"] = {
            "reward": experiment.rewards_history,
        }

    plot(results, file_name="step_experiment")


def run_clip_ratio_experiment():
    env = gym.make(default_params["env_name"])
    clip_ratios = [0.1, 0.2, 0.3]

    results = {}

    for idx in range(0, len(clip_ratios)):
        print(f"Clip ratio experiment: ratio={clip_ratios[idx]} [{idx + 1}/{len(clip_ratios)}]")
```

```
        state_dim = 2
        action_dim = 1

        actor = Actor(state_dim, action_dim).to(device)
        critic = Critic(state_dim).to(device)
        experiment = Experiment(
            env,
            actor,
            critic,
            default_params["num_iterations"],
            default_params["num_steps"],
            default_params["ppo_epochs"],
            clip_ratios[idx],
            normalize_advantages=False,
        )

        experiment.run()

        results[f"clip_ratio={clip_ratios[idx]}"] = {
            "reward": experiment.rewards_history,
        }

    plot(results, file_name="clip_ratio_experiment")


def run_mean_advantages_experiment():
    env = gym.make(default_params["env_name"])
    results = {}

    print("Mean advantages experiment")

    state_dim = 2
    action_dim = 1

    actor = Actor(state_dim, action_dim).to(device)
    critic = Critic(state_dim).to(device)
    experiment = Experiment(
        env,
```

```
            actor,
            critic,
            default_params["num_iterations"],
            default_params["num_steps"],
            default_params["ppo_epochs"],
            default_params["clip_ratio"],
            normalize_advantages=True,
        )

        experiment.run()

        results["mean_adv"] = {
            "reward": experiment.rewards_history,
        }

        plot(results, file_name="mean_adv_experiment")

    def run_epochs_experiment():
        env = gym.make(default_params["env_name"])
        epochs = [5, 10, 20]

        results = {}

        for idx in range(0, len(epochs)):
            print(f"Epoch experiment: epoch={epochs[idx]} [{idx + 1}/{len(epochs)}]")

            state_dim = 2
            action_dim = 1

            actor = Actor(state_dim, action_dim).to(device)
            critic = Critic(state_dim).to(device)
            experiment = Experiment(
                env,
                actor,
                critic,
                default_params["num_iterations"],
                default_params["num_steps"],
                epochs[idx],
```

```
            default_params["clip_ratio"],
            normalize_advantages=False,
        )

        experiment.run()

        results[f"epoch={epochs[idx]}"] = {
            "reward": experiment.rewards_history,
        }

    plot(results, file_name="epoch_experiment")


if __name__ == "__main__":
    run_default_experiment()
    run_steps_experiment()
    run_clip_ratio_experiment()
    run_mean_advantages_experiment()
    run_epochs_experiment()
```