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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Обучение с подкреплением»

Тема: Реализация PPO для среды MountainCarContinuous-v0

Студент гр. 0310 _____ Корсунов А.А.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург

2025

Цель работы.

Реализация алгоритма PPO для среды MountainCarContinuous-v0.

Задание.

1. Реализовать алгоритм PPO для среды MountainCarContinuous-v0;
2. Изменить длину траектории (steps);
3. Подобрать оптимальный коэффициент clip_ratio;
4. Добавить нормализацию преимуществ;
5. Сравнить обучение при разных количествах эпох;

Выполнение работы.

1. Алгоритм PPO реализован на ЯП Python с использованием библиотеки TenserFlow.

Код программы находится в Приложении А.

2. Были рассмотрены три длины траектории: 1024, 2048, 4096 (рис. 1).

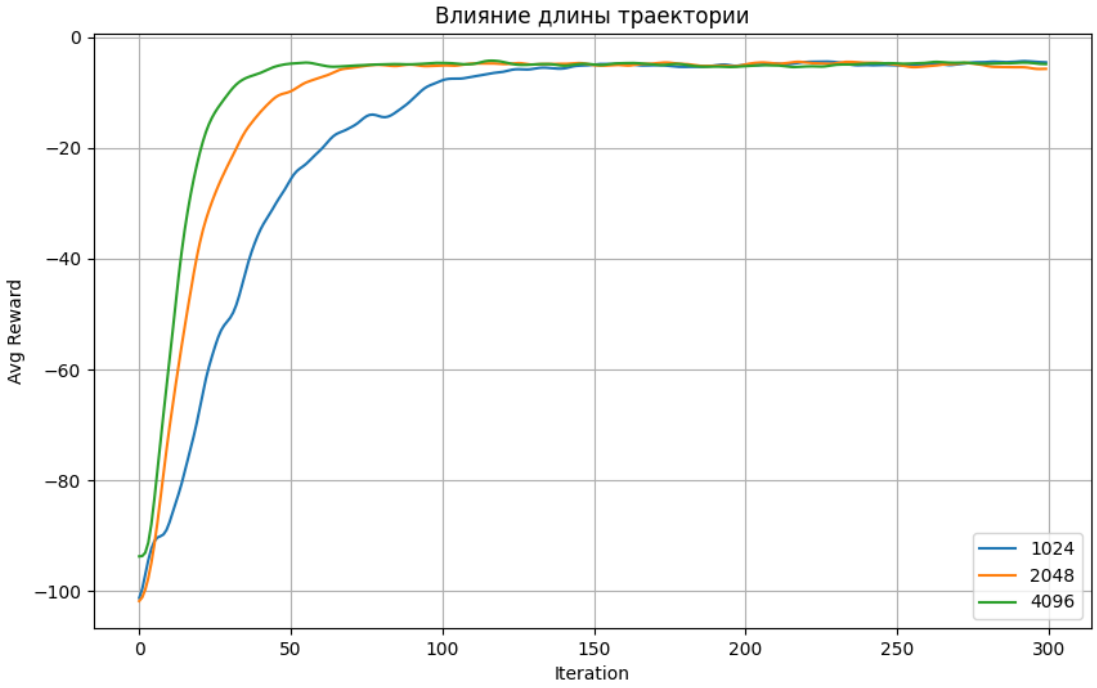

Рисунок 1 – Графики средних наград от итерации при различных длинах траектории

На основе рисунка 1 можно сделать следующие выводы:

Линия для steps=4096 достигает высоких значений среднего вознаграждения быстрее, чем для 2048 и 1024. При steps=1024 агент дольше остается на низких значениях вознаграждения, у него более шумная кривая и медленный рост. После ~150 итераций все подходы достигают примерно одинакового качества политики.

То есть, увеличение длины траектории улучшает стабильность и ускоряет обучение, но после достаточного количества итераций, качество обучения выравнивается независимо от длины траектории. Стоит также отметить, что более длинные траектории требуют более высокую вычислительную стоимость.

3. Были рассмотрены три значения коэффициента clip_ratio: 0.1, 0.2, 0.3 (рис. 2).

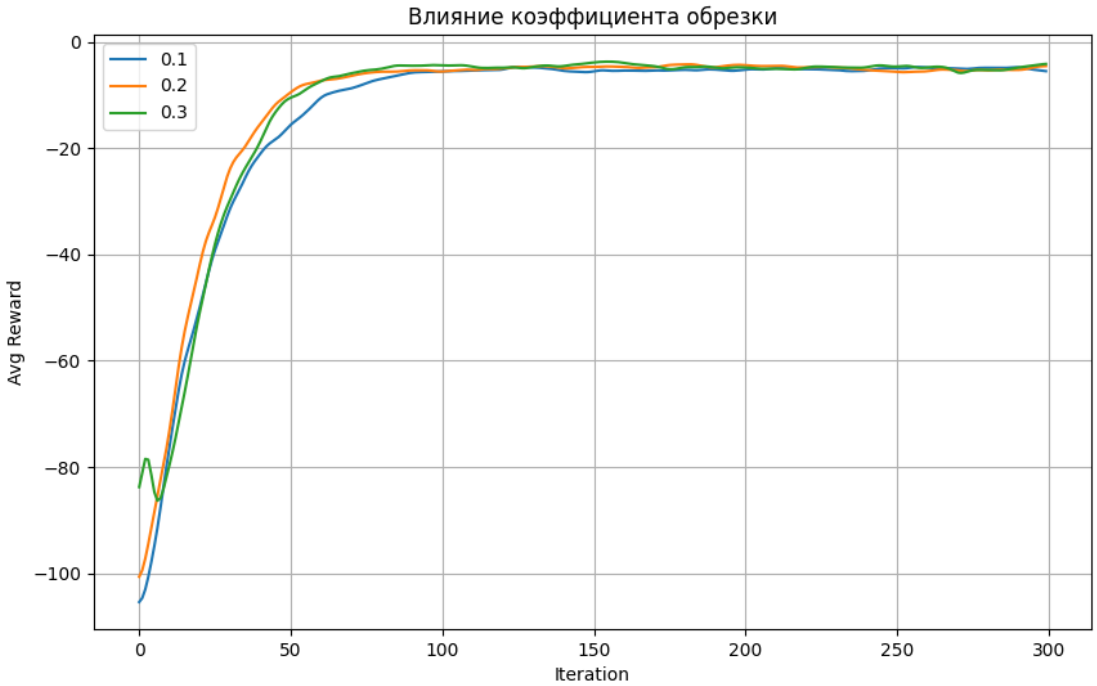

Рисунок 2 – Графики средних наград от итерации при различных значениях коэффициента clip_ratio

На основе рисунка 2 можно сделать следующие выводы:

Различия между кривыми небольшое. При коэффициенте 0.2 результаты наиболее лучшие: кривая растет и достигает стабильного поведения немного быстрее остальных.

4. Была добавлена нормализация преимуществ (рис. 3, 4).

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

Рисунок 3 – Нормализация преимуществ в коде

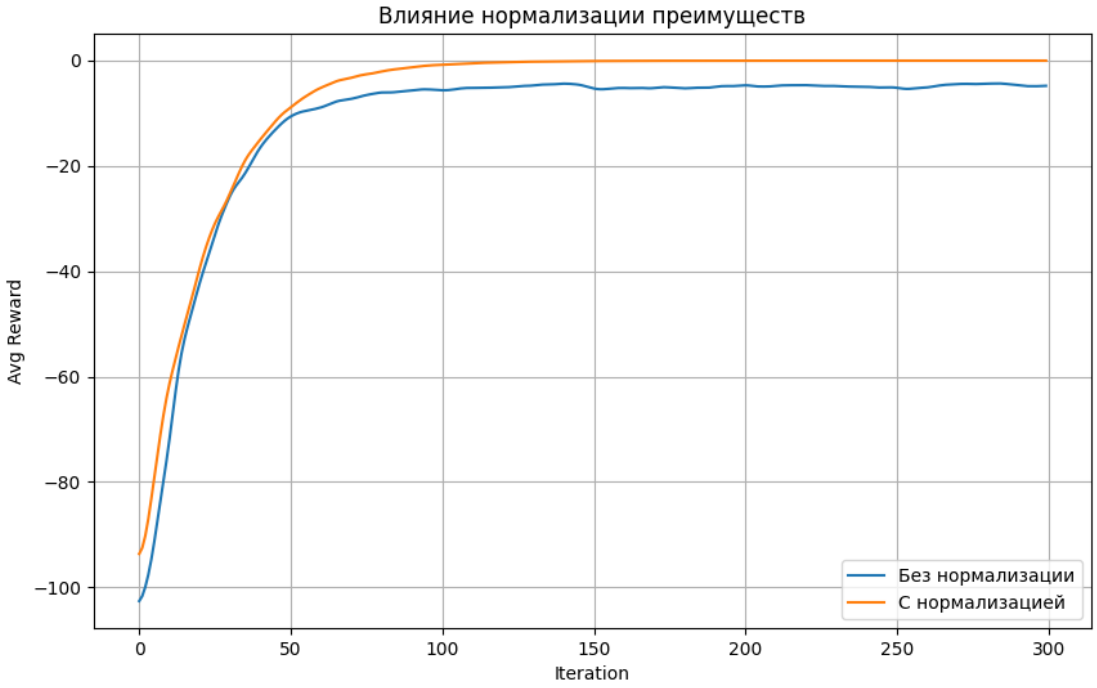

Рисунок 4 – Графики средних наград от итерации без нормализации и с нормализацией преимуществ

На основе рисунка 4 можно сделать следующие выводы:

Нормализация преимуществ ускоряет обучение: оранжевая кривая достигает высоких значений быстрее, чем синяя. Причем, с нормализацией агент достигает более высокий финальных наград.

5. Было проведено сравнение обучения при следующих количествах эпох: 5, 10, 20 (рис. 5).

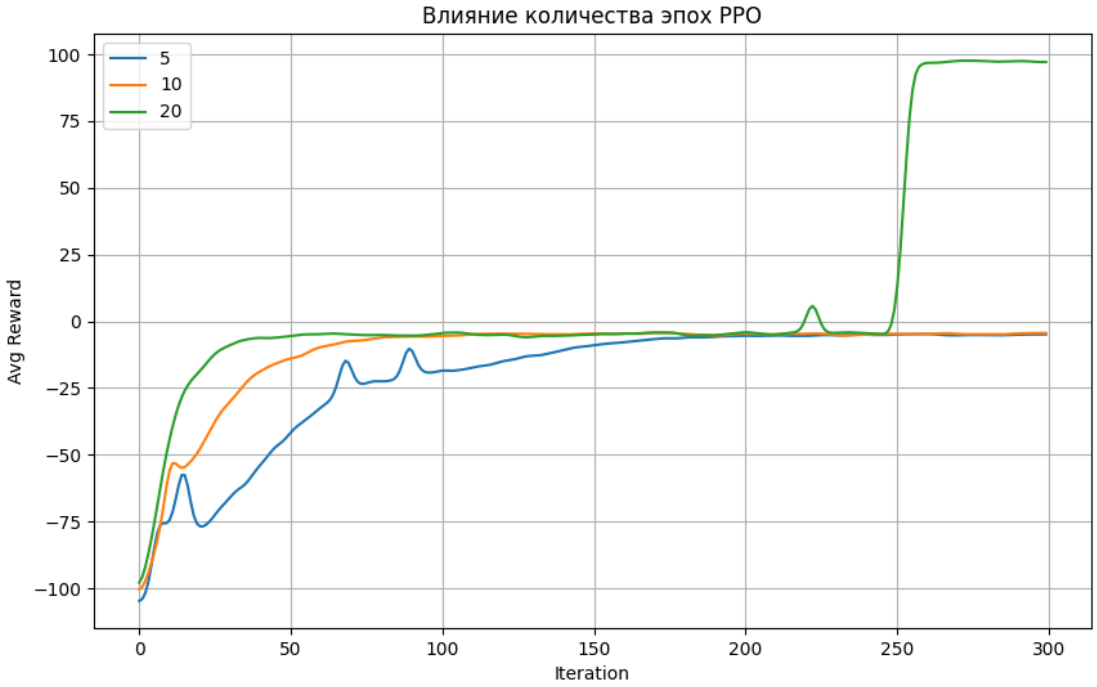

Рисунок 5 – Графики средних наград от итерации при различных количествах эпох

На основе рисунка 5 можно сделать следующие выводы:

С повышением количества эпох агент обучается быстрее и стабильнее. Причем, только при 20 эпохах агент успел приблизится к наивысшей награде за 300 итераций.

Выводы.

Реализован алгоритм PPO для среды MountainCarContinuous-v0. Были проведены исследования при различных длинах траекторий. Был подобран оптимальных коэффициент clip_ratio. Была добавлена нормализация преимуществ. Было проведено сравнение при разных количествах эпох.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ
MAIN.PY

```
import os
import gymnasium as gym
import torch
import numpy as np
import matplotlib.pyplot as plt
from torch import nn
from torch import optim
from torch.distributions import Normal
from scipy.ndimage import gaussian_filter1d

os.makedirs('plots', exist_ok=True)

env_name = "MountainCarContinuous-v0"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

num_iterations = 300
default_num_steps = 2048
default_clip_ratio = 0.2
default_ppo_epochs = 10
mini_batch_size = 64
gamma = 0.99
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.mu = nn.Linear(hidden_size, action_dim)
```

```
            self.log_std = nn.Parameter(torch.zeros(action_dim))

        def forward(self, x):
            x = self.net(x)
            return self.mu(x)

        def get_dist(self, state):
            mu = self.forward(state)
            std = torch.exp(self.log_std)
            return Normal(mu, std)

        def act(self, state):
            state = torch.FloatTensor(state).to(device)
            dist = self.get_dist(state)
            action = dist.sample()
            log_prob = dist.log_prob(action).sum(-1)
            return                          action.detach().cpu().numpy(),
log_prob.detach().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  plot_results(results,  title,  filename,  xlabel='Iteration',
ylabel='Avg Reward'):
        plt.figure(figsize=(10, 6))
        for label, data in results.items():
            plt.plot(smooth(data), label=str(label))
```

```
        plt.title(title)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.legend()
        plt.grid()
        plt.savefig(os.path.join('plots', filename))
        plt.close()

    def smooth(y, sigma=2):
        return gaussian_filter1d(y, sigma=sigma)

    def run_experiment(num_steps=default_num_steps,
                       clip_ratio=default_clip_ratio,
                       use_advantage_norm=False,
                       ppo_epochs=default_ppo_epochs):

        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)
        actor_optim = optim.Adam(actor.parameters(), lr=lr)
        critic_optim = optim.Adam(critic.parameters(), lr=lr)

        rewards_history = []

        for iteration in range(num_iterations):
            states, actions, rewards, dones, log_probs, ep_rewards = [],
[], [], [], [], []
            state, _ = env.reset()
            ep_reward = 0

            for _ in range(num_steps):
                action, log_prob = actor.act(state)
                next_state,  reward,  terminated,  truncated,  _  =
env.step(action)
                done = terminated or truncated
```

```
                states.append(state)
                actions.append(action)
                rewards.append(reward)
                dones.append(done)
                log_probs.append(log_prob)
                ep_reward += reward
                state = next_state

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

            states = torch.FloatTensor(np.array(states)).to(device)
            actions = torch.FloatTensor(np.array(actions)).to(device)
            old_log_probs                                                =
torch.FloatTensor(np.array(log_probs)).to(device)

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

            returns, advantages = compute_advantages(rewards, dones,
values, use_advantage_norm)
            returns = torch.FloatTensor(returns).to(device)
            advantages = torch.FloatTensor(advantages).to(device)

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

            for _ in range(ppo_epochs):
                np.random.shuffle(indices)
                for start in range(0, dataset_size, mini_batch_size):
                    end = start + mini_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 = actor.get_dist(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 - clip_ratio, 1 + clip_ratio) * batch_advantages
                    actor_loss = -torch.min(surr1, surr2).mean()

                    entropy = dist.entropy().mean()
                    critic_loss = (critic(batch_states).squeeze() - batch_returns).pow(2).mean()

                    loss = actor_loss + value_coef * critic_loss - entropy_coef * entropy

                    actor_optim.zero_grad()
                    critic_optim.zero_grad()
                    loss.backward()
                    actor_optim.step()
                    critic_optim.step()

            avg_reward = np.mean(ep_rewards) if ep_rewards else 0
            rewards_history.append(avg_reward)
            if iteration % 20 == 0:
                print(f"Iter {iteration}: Avg Reward {avg_reward:.2f}")

        env.close()
        return rewards_history

    def compute_advantages(rewards, dones, values, use_norm=False):
        returns = []
        advantages = []
        R = 0
```

```
        for r, d, v in zip(reversed(rewards), reversed(dones), reversed(values)):
            R = r + gamma * R * (1 - d)
            returns.insert(0, R)
            advantages.insert(0, R - v)

        advantages = np.array(advantages)
        if use_norm:
            advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

        return returns, advantages

    # Эксперимент 1: Различные длины траектории
    print("Эксперимент 1: Различные длины траектории")
    steps_params = [1024, 2048, 4096]
    results_steps = {}
    for steps in steps_params:
        print(f"\nЗапуск с num_steps={steps}")
        results_steps[steps] = run_experiment(num_steps=steps, use_advantage_norm=False)
    plot_results(results_steps, "Влияние длины траектории", "experiment1_steps.png")

    # Эксперимент 2: Различные clip_ratio
    print("\nЭксперимент 2: Различные clip_ratio")
    clip_params = [0.1, 0.2, 0.3]
    results_clip = {}
    for clip in clip_params:
        print(f"\nЗапуск с clip_ratio={clip}")
        results_clip[clip] = run_experiment(clip_ratio=clip, use_advantage_norm=False)
    plot_results(results_clip, "Влияние коэффициента обрезки", "experiment2_clip.png")

    # Эксперимент 3: С нормализацией преимуществ и без
    print("\nЭксперимент 3: Нормализация преимуществ")
```

```
    results_norm = {
        "Без нормализации": run_experiment(use_advantage_norm=False),
        "С нормализацией": run_experiment(use_advantage_norm=True)
    }
    plot_results(results_norm, "Влияние нормализации преимуществ",
"experiment3_norm.png")

    # Эксперимент 4: Различное количество эпох PPO
    print("\nЭксперимент 4: Количество эпох PPO")
    epochs_params = [5, 10, 20]
    results_epochs = {}
    for epochs in epochs_params:
        print(f"\nЗапуск с ppo_epochs={epochs}")
        results_epochs[epochs] = run_experiment(ppo_epochs=epochs)
    plot_results(results_epochs, "Влияние количества эпох PPO",
"experiment4_epochs.png")
```