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 _____ Кузнецов А.А.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург
2025 г.

Оглавление

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. Изменение длины траектории (steps)

Рассматриваемые длины траекторий: *{1024, 2048, 4096}*.

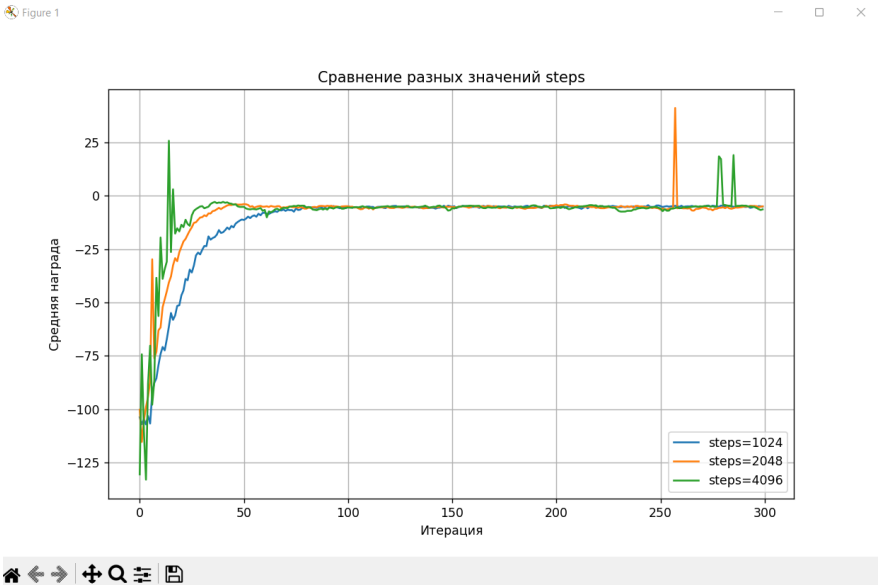


Рисунок 1 – График средних наград при различных значениях steps

Ключевой вывод: судя по рисунку 1, Линия для steps=*4096* достигает высоких значений среднего вознаграждения быстрее, чем для *2048* и *1024*, но также для steps=*4096* наблюдаются резкие скачки. При steps=*1024* агент дольше остается на низких значениях вознаграждения. После *~80* итераций наблюдается улучшение стабильности и ускоряет обучение. После *250* итераций наблюдаются небольшие скачки для steps=*4096* и steps=*2048*.

2. Подбор оптимального коэффициента clip_ratio

Рассматриваемые значения коэффициента clip_ratio: *{0.1, 0.2, 0.3}*.

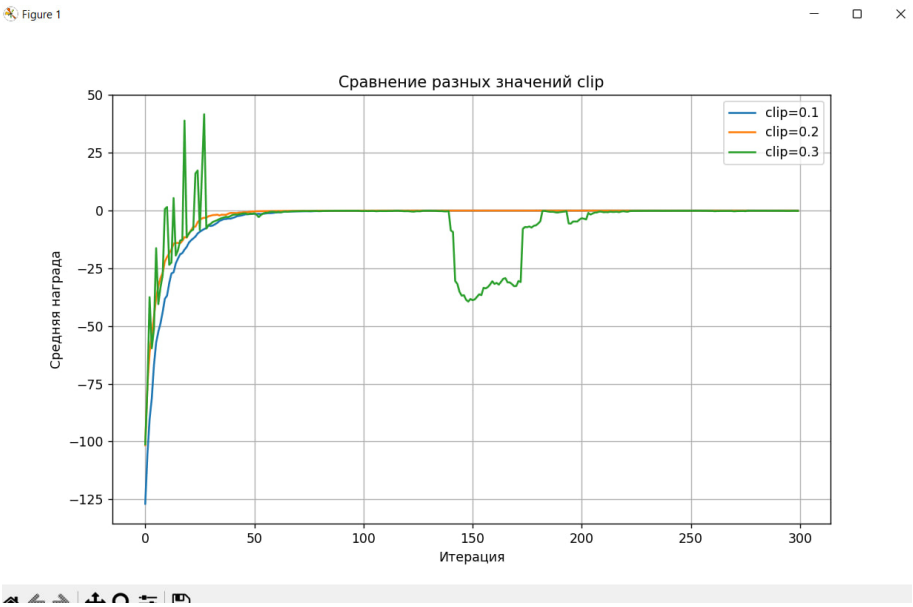


Рисунок 2 – График средних наград при различных значениях коэффициента clip_ratio

Ключевой вывод: судя по рисунку 2, при значении коэффициента *0.2* результаты наиболее лучшие, кривая растёт и достигает стабильного поведения быстрее остальных. При значении коэффициента *0.3* результаты нестабильны, кривая растёт, затем достигает стабильного поведения, но примерно с итерации *140* начинает вести себя нестабильно, что не очень хорошо. При значении коэффициента *0.1* результаты близки к значению коэффициента *0.2*, но всё же стабильного поведения достигает чуть дольше.

3. Добавление нормализации преимуществ

Влияние нормализации преимуществ будет демонстрироваться на сравнении разных значений steps.

Листинг кода (добавление нормализации в код):

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

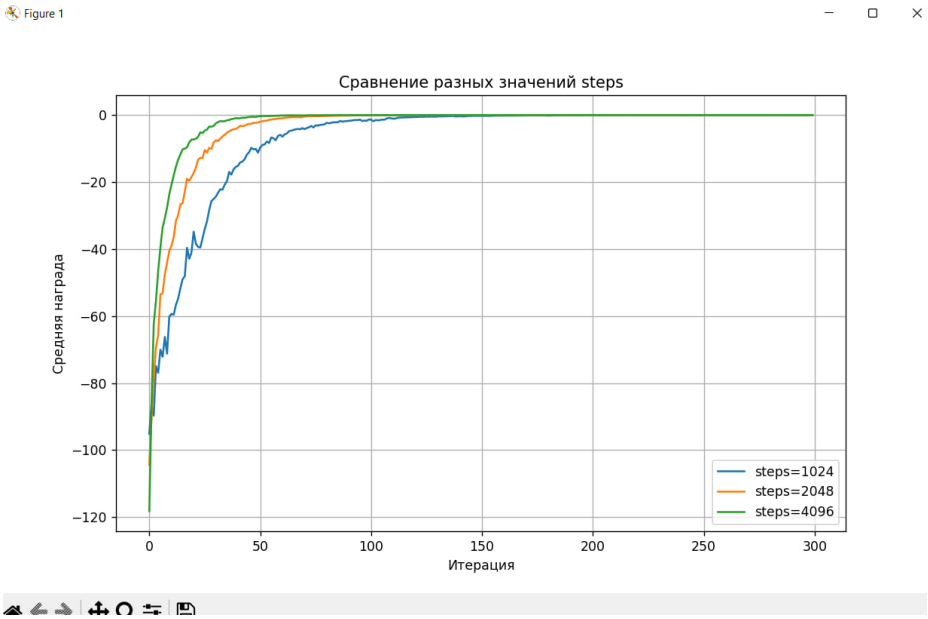
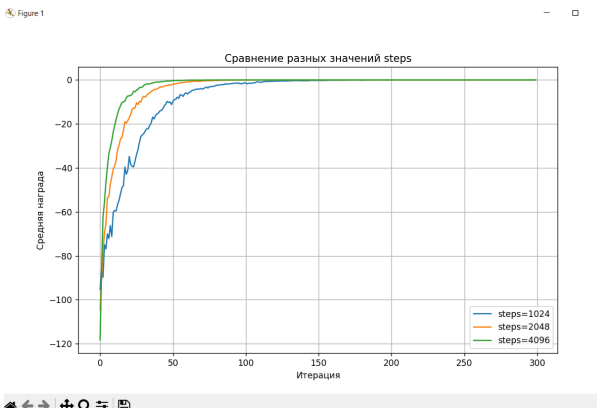
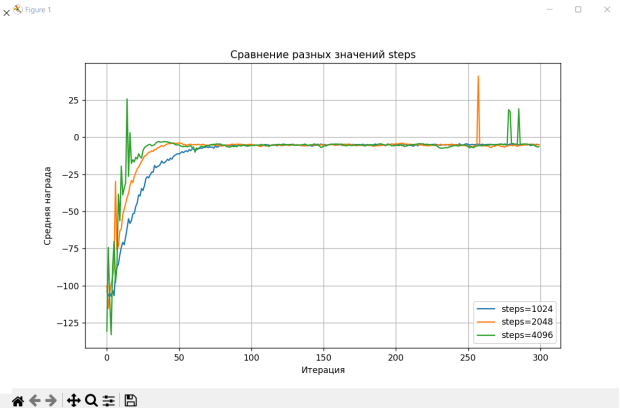


Рисунок 3 – График сравнения средних наград при разных значениях steps с добавлением нормализации преимуществ

С нормализацией



Без нормализации



Ключевой вывод: Нормализация преимуществ ускоряет обучение, показывает высокую стабильность и при этом средняя награда демонстрирует более плавный рост, без резких скачков в отрицательную область.

4. Сравнение обучения при разных количествах эпох

Рассматриваемые количества эпох *{5, 10, 20}*.

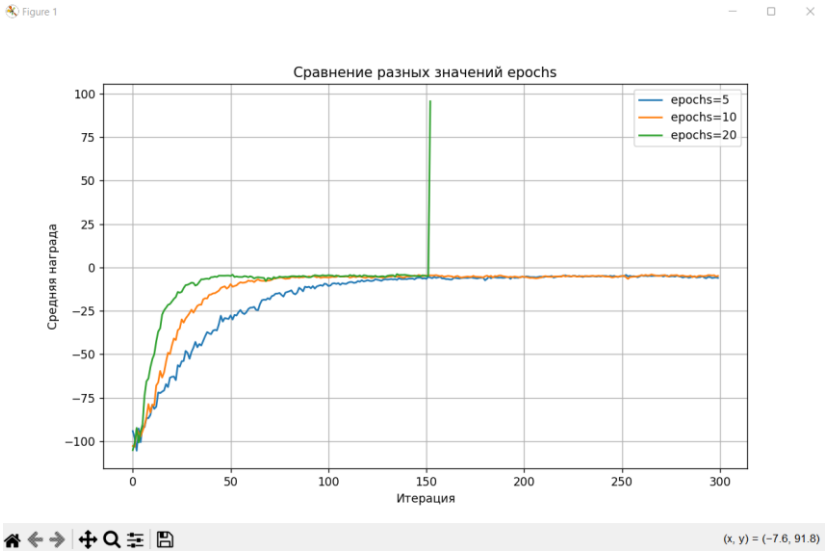


Рисунок 4 – График сравнения средних наград при разных количествах эпох с добавление

Ключевой вывод: судя по рисунку 4, можно сказать, что с повышением количества эпох агент обучается быстрее и стабильнее, причём при *20* эпохах агент успел приблизиться к наивысшей награде за *150* итераций.

ВЫВОДЫ

В процессе выполнения практической работы был реализован алгоритм PPO для среды MountainCarContinuous-v0, были проведены исследования при различных длинах траекторий, а также подобран оптимальных коэффициент clip_ratio, добавлена нормализация преимуществ и проведено сравнение при разных количествах эпох.

ПРИЛОЖЕНИЕ А

Исходный код

```
import os
import gymnasium as gym
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 import SummaryWriter
from tqdm import tqdm
import matplotlib.pyplot as plt
from pathlib import Path

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

class Config:
    def __init__(self):
        self.num_iterations = 300
        self.num_steps = 1024
        self.ppo_epochs = 10
        self.mini_batch_size = 64
        self.gamma = 0.99
        self.lmbda = 0.95
        self.clip_ratio = 0.2
        self.value_coef = 0.5
        self.entropy_coef = 0.01
        self.lr = 3e-4
        self.hidden_size = 64

class ActorCritic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size=64):
        super().__init__()
        self.shared = nn.Sequential(
            nn.Linear(state_dim, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, hidden_size),
            nn.Tanh()
        )
        self.actor_mean = nn.Linear(hidden_size, action_dim)
        self.actor_logstd = nn.Parameter(torch.zeros(1, action_dim))
        self.critic = nn.Linear(hidden_size, 1)

    def forward(self, x):
        shared_out = self.shared(x)
        mean = torch.tanh(self.actor_mean(shared_out)) * 2
        logstd = self.actor_logstd.expand_as(mean)
        value = self.critic(shared_out)
        return mean, logstd.exp(), value

class PPO:
    def __init__(self, env, config, experiment_name=""):
        self.env = env
        self.config = config
        self.state_dim = env.observation_space.shape[0]
        self.action_dim = env.action_space.shape[0]
        self.experiment_name = experiment_name

        self.policy = ActorCritic(self.state_dim, self.action_dim,
config.hidden_size).to(DEVICE)
        self.optimizer = optim.Adam(self.policy.parameters(), lr=config.lr)
        self.writer = SummaryWriter(f"runs/PPO_{ENV_NAME}_{experiment_name}")
        self.rewards_history = []

    def get_action(self, state):
        state = torch.FloatTensor(state).unsqueeze(0).to(DEVICE)
        with torch.no_grad():
```

```
            mean, std, value = self.policy(state)
            dist = Normal(mean, std)
            action = dist.sample()
            log_prob = dist.log_prob(action).sum(-1)
        return action.cpu().numpy()[0], log_prob.item(), value.item()

    def compute_gae(self, rewards, values, dones):
        advantages = np.zeros_like(rewards)
        last_advantage = 0
        last_value = values[-1]

        for t in reversed(range(len(rewards))):
            if dones[t]:
                delta = rewards[t] - values[t]
                last_advantage = 0
                last_value = 0
            else:
                delta = rewards[t] + self.config.gamma * last_value - values[t]
                last_value = values[t]

            advantages[t] = delta + self.config.gamma * self.config.lmbda *
last_advantage
            last_advantage = advantages[t]
        returns = advantages + values
        advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

        return returns, advantages

    def update(self, states, actions, old_log_probs, returns, advantages):
        dataset_size = states.shape[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.mini_batch_size):
                end = start + self.config.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]

                mean, std, values = self.policy(batch_states)
                dist = Normal(mean, std)
                new_log_probs = dist.log_prob(batch_actions).sum(-1)
                entropy = dist.entropy().mean()

                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
                actor_loss = -torch.min(surr1, surr2).mean()

                critic_loss = (batch_returns - values.squeeze()).pow(2).mean()

                loss = actor_loss + self.config.value_coef * critic_loss -
self.config.entropy_coef * entropy

                self.optimizer.zero_grad()
                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.policy.parameters(), 0.5)
                self.optimizer.step()

    def train(self):
        best_reward = -np.inf

        for iteration in tqdm(range(self.config.num_iterations),
desc=self.experiment_name):
            states, actions, log_probs, rewards, dones, values = [], [], [], [], [],
```

```
[]
            episode_rewards = []
            state, _ = self.env.reset()
            episode_reward = 0

            for _ in range(self.config.num_steps):
                action, log_prob, value = self.get_action(state)
                next_state, reward, terminated, truncated, _ = self.env.step(action)
                done = terminated or truncated

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

                state = next_state
                episode_reward += reward

                if done:
                    state, _ = self.env.reset()
                    episode_rewards.append(episode_reward)
                    episode_reward = 0

            if episode_reward > 0:
                episode_rewards.append(episode_reward)

            returns, advantages = self.compute_gae(rewards, values, dones)

            states_t = torch.FloatTensor(np.array(states)).to(DEVICE)
            actions_t = torch.FloatTensor(np.array(actions)).to(DEVICE)
            log_probs_t = torch.FloatTensor(np.array(log_probs)).to(DEVICE)
            returns_t = torch.FloatTensor(returns).to(DEVICE)
            advantages_t = torch.FloatTensor(advantages).to(DEVICE)

            self.update(states_t, actions_t, log_probs_t, returns_t, advantages_t)

            avg_reward = np.mean(episode_rewards) if episode_rewards else 0
            self.rewards_history.append(avg_reward)
            self.writer.add_scalar("Reward/Mean", avg_reward, iteration)

            if avg_reward > best_reward:
                best_reward = avg_reward
                torch.save(self.policy.state_dict(),
f"best_model_{self.experiment_name}.pth")

            if avg_reward >= 90:
                print(f"Решено на итерации {iteration}!")
                break

        return self.rewards_history

def plot_comparison(results, param_name, param_values):
    plt.figure(figsize=(10, 6))
    for value in param_values:
        rewards = results[f"{param_name}_{value}"]
        plt.plot(rewards, label=f"{param_name}={value}")

    plt.title(f"Сравнение разных значений {param_name}")
    plt.xlabel("Итерация")
    plt.ylabel("Средняя награда")
    plt.legend()
    plt.grid(True)
    plt.savefig(f"comparison_{param_name}.png")
    plt.show()

def run_experiments():
    env = gym.make(ENV_NAME)
    config = Config()
```

```
    Path("experiment_results").mkdir(exist_ok=True)

    steps_results = {}
    steps_variants = [1024, 2048, 4096]
    print("\nЭксперимент: длина траектории")
    for steps in steps_variants:
        config.num_steps = steps
        agent = PPO(env, config, experiment_name=f"steps_{steps}")
        rewards = agent.train()
        steps_results[f"steps_{steps}"] = rewards
    plot_comparison(steps_results, "steps", steps_variants)

    clip_results = {}
    clip_variants = [0.1, 0.2, 0.3]
    print("\nЭксперимент: clip ratio")
    for clip in clip_variants:
        config.clip_ratio = clip
        agent = PPO(env, config, experiment_name=f"clip_{clip}")
        rewards = agent.train()
        clip_results[f"clip_{clip}"] = rewards
    plot_comparison(clip_results, "clip", clip_variants)

    epochs_results = {}
    epochs_variants = [5, 10, 20]
    print("\nЭксперимент: количество эпох")
    for epochs in epochs_variants:
        config.ppo_epochs = epochs
        agent = PPO(env, config, experiment_name=f"epochs_{epochs}")
        rewards = agent.train()
        epochs_results[f"epochs_{epochs}"] = rewards
    plot_comparison(epochs_results, "epochs", epochs_variants)

    env.close()

    all_results = {**steps_results, **clip_results, **epochs_results}
    np.savez("experiment_results/all_results.npz", **all_results)

    return all_results

def plot_all_results(results):
    plt.figure(figsize=(15, 10))

    plt.subplot(2, 2, 1)
    for steps in [512, 1024, 2048]:
        plt.plot(results[f"steps_{steps}"], label=f"Steps={steps}")
    plt.title("Сравнение длины траектории")
    plt.xlabel("Итерация")
    plt.ylabel("Награда")
    plt.legend()

    plt.subplot(2, 2, 2)
    for clip in [0.1, 0.2, 0.3]:
        plt.plot(results[f"clip_{clip}"], label=f"Clip={clip}")
    plt.title("Сравнение clip ratio")
    plt.xlabel("Итерация")
    plt.legend()

    plt.subplot(2, 2, 3)
    for epochs in [5, 10, 20]:
        plt.plot(results[f"epochs_{epochs}"], label=f"Epochs={epochs}")
    plt.title("Сравнение количества эпох")
    plt.xlabel("Итерация")
    plt.legend()

    plt.subplot(2, 2, 4)
    for key, rewards in results.items():
        plt.plot(rewards, label=key)
    plt.title("Все эксперименты")
    plt.xlabel("Итерация")
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

```
    plt.tight_layout()
    plt.savefig("experiment_results/all_comparisons.png", bbox_inches='tight')
    plt.show()

if __name__ == "__main__":
    results = run_experiments()
    plot_all_results(results)
```