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

ОТЧЕТ

по исследовательской работе
по дисциплине «Обучение с подкреплением»
Тема: Robust Policy Optimization

Студент гр. 0306 _____ Алексеев Р.В.

Студент гр. 0306 _____ Кирсанов Д.Э.

Студент гр. 0306 _____ Сологуб Н.А.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург
2025

Цель работы.

Изучить улучшение PPO при помощи добавления шума.

Задание.

1. Реализовать алгоритм, описанный в статье «ROBUST POLICY OPTIMIZATION IN DEEP REINFORCEMENT LEARNING».
2. Сравнить стандартный и описанный в статье алгоритмы.

Выполнение работы.

В ходе работ был рассмотрен алгоритм RPO, описанный в статье «ROBUST POLICY OPTIMIZATION IN DEEP REINFORCEMENT LEARNING». Данный алгоритм является модификацией алгоритма PPO, отличие заключается в поддержании уровня энтропии на протяжении обучения при помощи добавления равномерного шума. При этом плотность распределения случайной величины становится менее центрированным вокруг среднего значения, соответствующий график представлен на рис. 1.

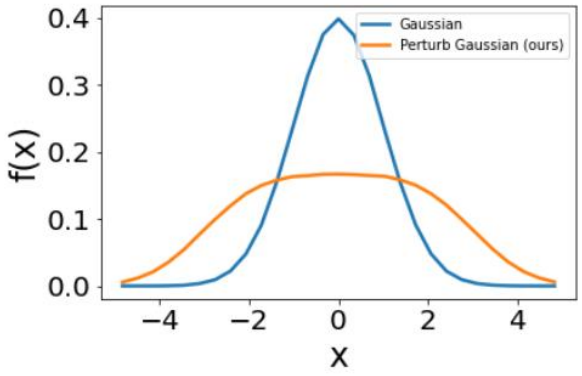

Рисунок 1 – Нормальное распределение Гаусса и соответствующее распределение с добавлением шума.

Для достижения данного результата на каждом шаге к среднему значению μ распределения добавляется случайная величина $z \sim U(-a, a)$, создавая тем самым возмущенное среднее $\mu' = \mu + z$. Этот шаг позволяет поддерживать энтропию по мере обучения, что способствует лучшему исследованию среды.

Помимо этого, преимуществом данного решения по сравнению с другими является его универсальность, так как шум можно добавлять и к другим распределениям.

Авторы статьи протестировали свое решение на 18 различных задачах из четырех платформ. Результаты 9 из них представлены на рис. 2.

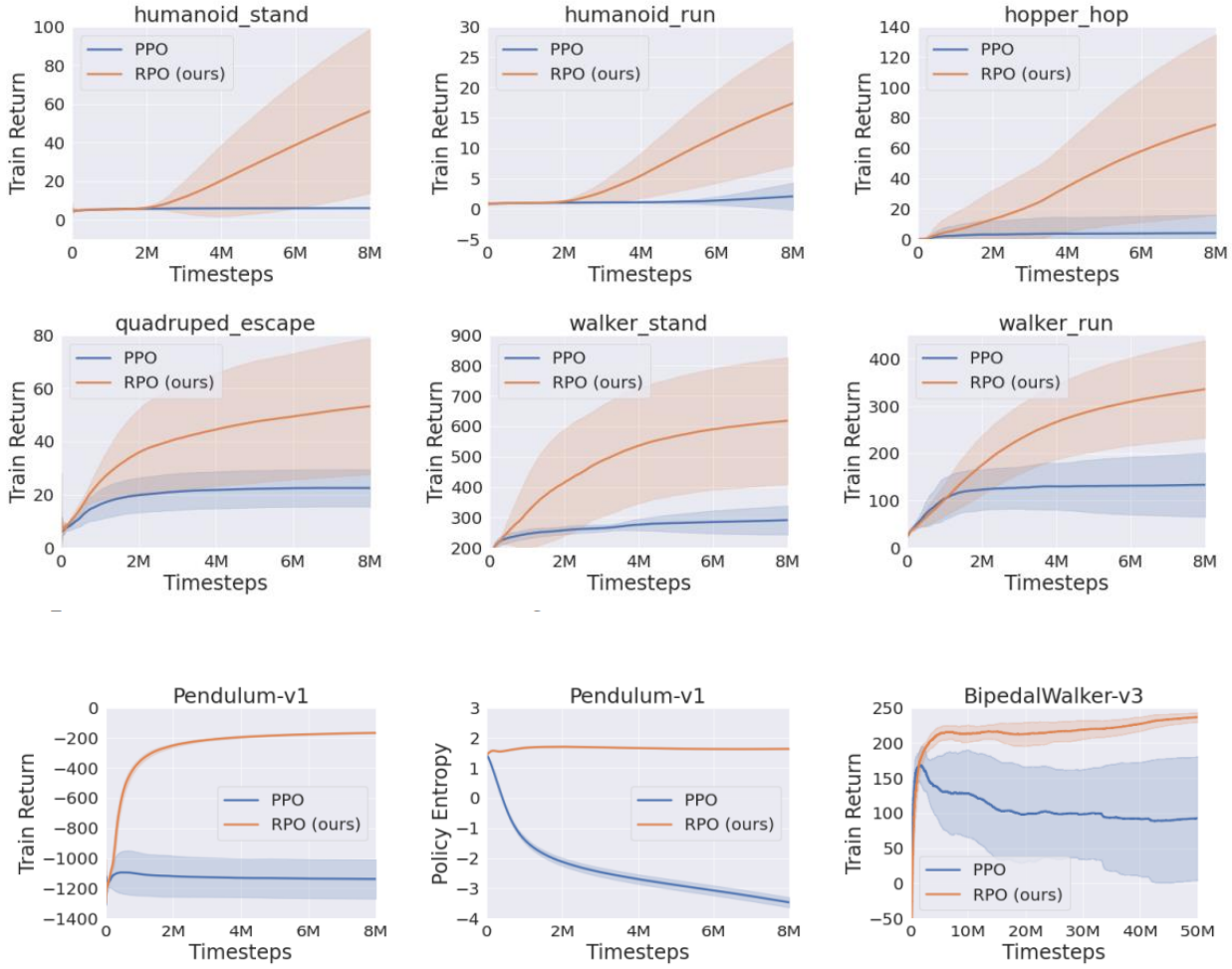

Рисунок 2 – Результаты обучения.

По рис. 2 видно, что при использовании PPO (синие линии) по мере обучения результаты или деградируют, или перестают изменяться, а с RPO (оранжевые линии) улучшаются. Особенно это заметно в задачах pendulum-v1, bipedalWalker-v3, humanoid_run и humanoid_hop. В первых двух случаях PPO деградирует и результаты ухудшаются с увеличением количества шагов, а в двух последних результаты стагнируют и остаются около 0. При этом везде предложенный алгоритм RPO демонстрирует улучшение результатов обучения.

Для изучения представленного в статье алгоритма RPO, он был реализован совместно с PPO, код представлен в приложении А. В данной реализации решается задача обучения для pendulum, то есть, чтобы перевернутый маятник оказался в вертикальном положении с точкой крепления внизу. Решение этой задачи выбрано в связи с тем, что она входит в число использованных в статье.

Были проведены запуски с различными сидами обоих реализованных алгоритмов, результаты обучения смотрите в приложении В. Графики представлены на рис. 3 – 5. Сверху PPO, снизу RPO.

Рисунок 3 – Зависимость награды от количества шагов для сида 69.

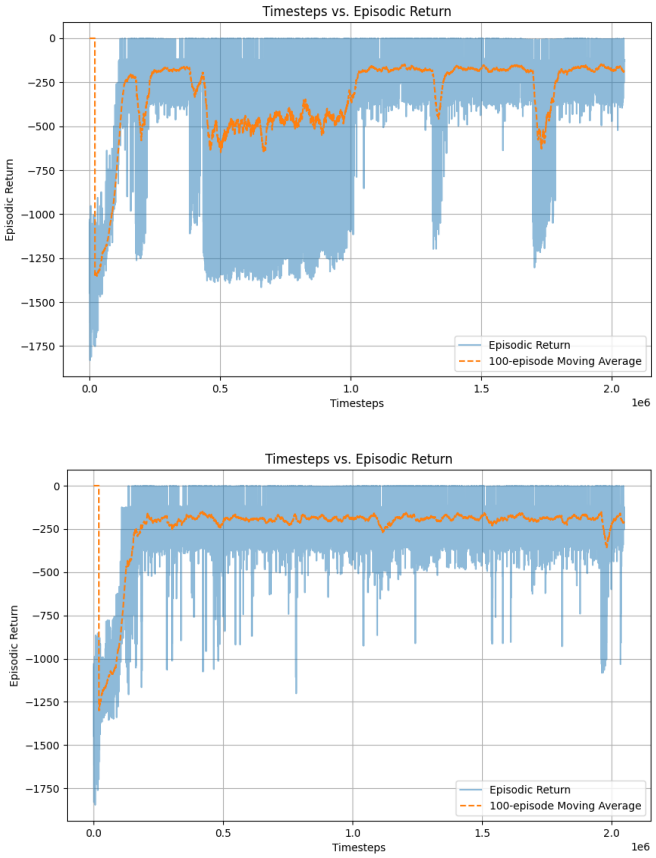

Рисунок 4 – Зависимость награды от количества шагов для сида 666.

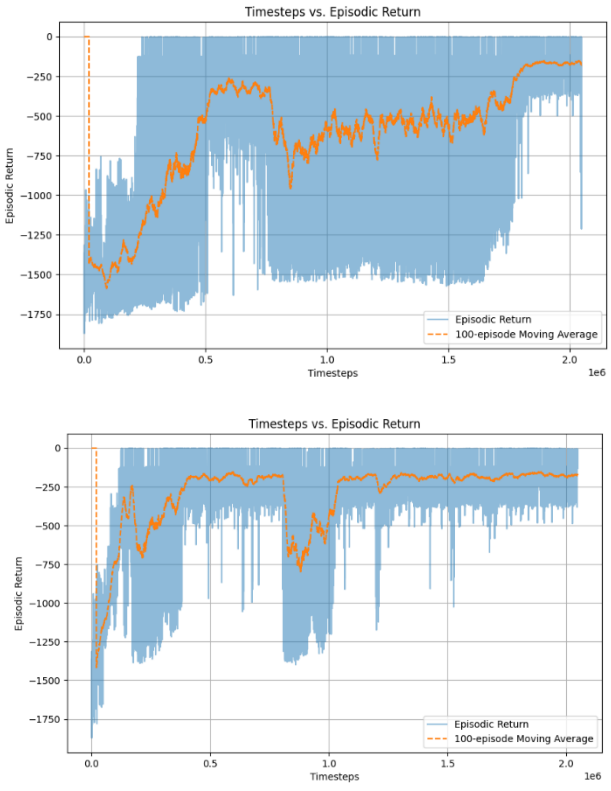

Рисунок 5 – Зависимость награды от количества шагов для сида 1337.

По рисункам 3 – 5 видно, что алгоритм PPO показывает менее стабильные результаты по мере увеличения количества шагов. Например, для сида 666 заметно уменьшение награды на промежутке от 500 000 до 1 000 000. При этом подобного нет на графике для RPO. На рисунке 5 также заметны уменьшения на обоих графиках, но на верхнем графике, который отображает зависимость для PPO, область уменьшения награды примерно от 800 000 до 1 800 000, а для RPO от 800 000 до 1 000 00. А также график RPO показывает, что награда увеличивается гораздо быстрее на начальных этапах.

Приведенные графики показывают, что алгоритм RPO действительно показывает результаты лучше, чем PPO. Тем самым можно сделать следующие выводы. Поддержание уровня энтропии на протяжении всего обучения при помощи наличия шума позволяет улучшить результаты, способствуя лучшему исследованию среды. Таким образом проведенное исследование подтверждает выводы статьи «ROBUST POLICY OPTIMIZATION IN DEEP REINFORCEMENT LEARNING».

Выводы.

В ходе исследовательской работы была рассмотрена статья «ROBUST POLICY OPTIMIZATION IN DEEP REINFORCEMENT LEARNING», в которой предлагается модификация алгоритма PPO путем добавления шума для поддержания уровня энтропии на протяжении всего обучения.

Алгоритмы RPO и PPO были реализованы и протестированы. Результаты подтвердили выводы изучаемой статьи. Поддержание уровня энтропии на протяжении обучения действительно способствует улучшению результатов и лучшему исследованию среды.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: pendulum_ppo.py

```
import gymnasium as gym
import argparse
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
import matplotlib.pyplot as plt
from gymnasium.wrappers import RecordVideo
from torch.distributions import Normal
from tqdm import tqdm
import os

# Парсинг аргументов
parser = argparse.ArgumentParser(description='PPO for Pendulum')
parser.add_argument('--num-iterations', type=int, default=1000,
help='number of iterations for learning')
parser.add_argument('--num-epochs', type=int, default=10,
help='number of epochs for updating policy')
parser.add_argument('--clip-ratio', type=float, default=0.2,
help='clip value for PPO loss')
parser.add_argument('--gamma', type=float, default=0.99,
help='discount factor')
parser.add_argument('--value-coef', type=float, default=0.5,
help='value loss coefficient')
parser.add_argument('--entropy-coef', type=float, default=0.01,
help='entropy loss coefficient')
parser.add_argument('--sub-batch-size', type=int, default=32,
help='size of sub-samples')
parser.add_argument('--steps', type=int, default=2048, help='number
of steps per trajectory')
parser.add_argument('--gae-lambda', type=float, default=0.95,
help='lambda for general advantage estimation')
parser.add_argument('--normalize-advantages', action='store_true',
default=True, help='normalize advantages')
parser.add_argument('--seed', type=int, default=1337, help='random
seed')
parser.add_argument('--render', action='store_true', help='render
the environment')
parser.add_argument('--log-interval', type=int, default=10,
help='interval between training status logs')
args = parser.parse_args()

# Параметры
lr = 3e-4
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Инициализация окружения
env = gym.make('Pendulum-v1', render_mode='rgb_array')

env.reset(seed=args.seed)
```

```
torch.manual_seed(args.seed)
np.random.seed(args.seed)
random.seed(args.seed)

# Размеры пространства состояний и действий
state_dim = env.observation_space.shape[0]  # 3
action_dim = env.action_space.shape[0]  # 1
action_low = float(env.action_space.low[0])  # -2.0
action_high = float(env.action_space.high[0])  # 2.0

# Класс буфера
class RolloutBuffer:
    def __init__(self, max_steps):
        self.max_steps = max_steps
        self.states = []
        self.actions = []
        self.rewards = []
        self.log_probs = []
        self.dones = []
        self.episode_rewards = []
        self.episode_lengths = []
        self.episode_timesteps = []
        self.episode_reward = 0
        self.episode_length = 0
        self.current_steps = 0
        self.global_timesteps = 0

    def reset(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.log_probs = []
        self.dones = []
        self.episode_reward = 0
        self.episode_length = 0
        self.current_steps = 0

    def add(self, state, action, reward, log_prob, done):
        self.states.append(state)
        self.actions.append(action)
        self.rewards.append(reward)
        self.log_probs.append(log_prob)
        self.dones.append(done)
        self.episode_reward += reward
        self.episode_length += 1
        self.current_steps += 1
        self.global_timesteps += 1

    def end_episode(self):
        self.episode_rewards.append(self.episode_reward)
        self.episode_lengths.append(self.episode_length)
        self.episode_timesteps.append(self.global_timesteps)
        self.episode_reward = 0
        self.episode_length = 0

    def is_full(self):
        return self.current_steps >= self.max_steps
```

```
        def get_data(self):
            return {
                "states": torch.FloatTensor(self.states).to(device),
                "actions": torch.FloatTensor(self.actions).to(device),
                "rewards": self.rewards,
                "log_probs":
torch.FloatTensor(self.log_probs).to(device),
                "dones": self.dones,
            }

    # Модель актора
    class Actor(nn.Module):
        def __init__(self, state_dim, action_dim, hidden_dim=64):
            super(Actor, self).__init__()
            self.mean_net = nn.Sequential(
                nn.Linear(state_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, action_dim),
                nn.Tanh()
            )
            self.log_std = nn.Parameter(torch.zeros(action_dim))

        def forward(self, state):
            mean = self.mean_net(state) * action_high
            std = torch.exp(self.log_std).expand_as(mean)
            return mean, std

        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)
            dist = self.get_dist(state)
            action = dist.sample()
            action = torch.clamp(action, min=action_low, max=action_high)
            log_prob = dist.log_prob(action).sum(dim=-1)
            return action.cpu().numpy()[0], log_prob.item()

    # Модель критика
    class Critic(nn.Module):
        def __init__(self, state_dim, hidden_dim=64):
            super(Critic, self).__init__()
            self.network = nn.Sequential(
                nn.Linear(state_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, 1)
            )

        def forward(self, state):
            return self.network(state)
```

```
    # Вычисление возвратов и преимуществ с GAE
    def compute_returns_and_advantages(rewards, dones, values):
        returns = []
        advantages = []
        last_gae_lam = 0
        next_value = 0

        for t in reversed(range(len(rewards))):
            if t == len(rewards) - 1:
                next_nonterminal = 1.0 - dones[t]
                next_value = next_value
            else:
                next_nonterminal = 1.0 - dones[t + 1]
                next_value = values[t + 1]

            delta = rewards[t] + args.gamma * next_value * next_nonterminal - values[t]
            last_gae_lam = delta + args.gamma * args.gae_lambda * next_nonterminal * last_gae_lam
            advantages.insert(0, last_gae_lam)
            returns.insert(0, last_gae_lam + values[t])
            next_value = values[t]

        returns = torch.FloatTensor(returns).to(device)
        advantages = torch.FloatTensor(advantages).to(device)
        if args.normalize_advantages:
            advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)
        return returns, advantages

    # Обновление политики
    def update_ppo(states, actions, log_probs_old, returns, advantages):
        for _ in range(args.num_epochs):
            indices = np.random.permutation(len(states))
            for start in range(0, len(states), args.sub_batch_size):
                batch_indices = indices[start:start + args.sub_batch_size]
                state_batch = states[batch_indices]
                action_batch = actions[batch_indices]
                log_prob_old_batch = log_probs_old[batch_indices]
                return_batch = returns[batch_indices]
                advantage_batch = advantages[batch_indices]

                dist = actor.get_dist(state_batch)
                log_prob = dist.log_prob(action_batch).sum(dim=-1)
                value = critic(state_batch).squeeze()

                ratio = torch.exp(log_prob - log_prob_old_batch)
                surr1 = ratio * advantage_batch
                surr2 = torch.clamp(ratio, 1 - args.clip_ratio, 1 + args.clip_ratio) * advantage_batch
                actor_loss = -torch.min(surr1, surr2).mean()
                entropy_loss = dist.entropy().mean()
                actor_loss = actor_loss - args.entropy_coef * entropy_loss
```

```
                critic_loss = args.value_coef * nn.MSELoss()(value,
return_batch)

                actor_optimizer.zero_grad()
                actor_loss.backward()
                actor_optimizer.step()

                critic_optimizer.zero_grad()
                critic_loss.backward()
                critic_optimizer.step()

    # Сбор траекторий
    def collect_trajectories(actor, buffer):
        state, _ = env.reset()
        buffer.reset()
        while not buffer.is_full():
            action, log_prob = actor.act(state)
            next_state, reward, terminated, truncated, _ =
env.step(action)
            done = terminated or truncated

            buffer.add(state, action, reward, log_prob, done)
            state = next_state
            if done:
                buffer.end_episode()
                state, _ = env.reset()

        return buffer.get_data()

    # Построение графика timesteps vs. episodic return
    def plot_timesteps_vs_rewards(timesteps, rewards, filename):
        plt.figure(figsize=(10, 6))
        plt.plot(timesteps, rewards, label='Episodic Return', alpha=0.5)
        if len(rewards) >= 100:
            rewards_t = torch.tensor(rewards, dtype=torch.float)
            means = rewards_t.unfold(0, 100, 1).mean(1).view(-1)
            means = torch.cat((torch.zeros(99), means))
            plt.plot(timesteps, means.numpy(), label='100-episode Moving
Average', linestyle='--')
        plt.xlabel('Timesteps')
        plt.ylabel('Episodic Return')
        plt.title('Timesteps vs. Episodic Return')
        plt.legend(loc='best')
        plt.grid(True)
        plt.savefig(filename)
        plt.close()

    # Основной цикл обучения
    def train_ppo():
        global actor, critic, actor_optimizer, critic_optimizer
        actor = Actor(state_dim, action_dim).to(device)
        critic = Critic(state_dim).to(device)
        actor_optimizer = optim.Adam(actor.parameters(), lr=lr)
        critic_optimizer = optim.Adam(critic.parameters(), lr=lr)

        buffer = RolloutBuffer(args.steps)
```

```
        for iteration in tqdm(range(args.num_iterations),
desc="Training"):
            batch = collect_trajectories(actor, buffer)
            states = batch["states"]
            actions = batch["actions"]
            log_probs_old = batch["log_probs"]
            with torch.no_grad():
                values = critic(states).squeeze()
            returns, advantages =
compute_returns_and_advantages(batch["rewards"], batch["dones"], values)
            update_ppo(states, actions, log_probs_old, returns,
advantages)

            if (iteration + 1) % args.log_interval == 0:
                avg_reward = np.mean(buffer.episode_rewards[-
args.log_interval:]) if buffer.episode_rewards else 0
                avg_length = np.mean(buffer.episode_lengths[-
args.log_interval:]) if buffer.episode_lengths else 0
                print(f"Iteration {iteration + 1}:")
                print(f"  Average Reward: {avg_reward:.2f}")
                print(f"  Average Episode Length: {avg_length:.2f}")

        return buffer.episode_timesteps, buffer.episode_rewards

    if __name__ == '__main__':
        os.makedirs("results", exist_ok=True)
        timesteps, rewards = train_ppo()
        plot_timesteps_vs_rewards(timesteps, rewards,
"results/timesteps_vs_rewards.png")
        env.close()
```

Название файла: pendulum_rpo.py

```
    import gymnasium as gym
    import argparse
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import numpy as np
    import random
    import matplotlib.pyplot as plt
    from gymnasium.wrappers import RecordVideo
    from torch.distributions import Normal
    from tqdm import tqdm
    import os

    # П а р с и н г  а р г у м е н т о в
    parser = argparse.ArgumentParser(description='PPO for Pendulum')
    parser.add_argument('--num-iterations', type=int, default=1000,
help='number of iterations for learning')
    parser.add_argument('--num-epochs', type=int, default=10,
help='number of epochs for updating policy')
    parser.add_argument('--clip-ratio', type=float, default=0.2,
help='clip value for PPO loss')
    parser.add_argument('--gamma', type=float, default=0.99,
help='discount factor')
```

```
    parser.add_argument('--value-coef', type=float, default=0.5,
help='value loss coefficient')
    parser.add_argument('--entropy-coef', type=float, default=0.01,
help='entropy loss coefficient')
    parser.add_argument('--sub-batch-size', type=int, default=32,
help='size of sub-samples')
    parser.add_argument('--steps', type=int, default=2048, help='number
of steps per trajectory')
    parser.add_argument('--gae-lambda', type=float, default=0.95,
help='lambda for general advantage estimation')
    parser.add_argument('--normalize-advantages', action='store_true',
default=True, help='normalize advantages')
    parser.add_argument('--seed', type=int, default=1, help='random
seed')
    parser.add_argument('--render', action='store_true', help='render
the environment')
    parser.add_argument('--log-interval', type=int, default=10,
help='interval between training status logs')
    parser.add_argument('--rpo-alpha', type=float, default=0.5,
help='RPO noise scale for policy update')  # Добавлено
    args = parser.parse_args()

    # Параметры
    lr = 3e-4
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Инициализация окружения
    env = gym.make('Pendulum-v1', render_mode='rgb_array')
    env = RecordVideo(
        env,
        video_folder="./videos",
        episode_trigger=lambda t: t % 24 == 0,  # Record every episode
        video_length=200,
        name_prefix="pendulum-rpo"
    )
    env.reset(seed=args.seed)
    torch.manual_seed(args.seed)
    np.random.seed(args.seed)
    random.seed(args.seed)

    # Размеры пространства состояний и действий
    state_dim = env.observation_space.shape[0]  # 3
    action_dim = env.action_space.shape[0]  # 1
    action_low = float(env.action_space.low[0])  # -2.0
    action_high = float(env.action_space.high[0])  # 2.0

    # Класс буфера
    class RolloutBuffer:
        def __init__(self, max_steps):
            self.max_steps = max_steps
            self.states = []
            self.actions = []
            self.rewards = []
            self.log_probs = []
            self.dones = []
            self.values = []  # Добавлено для хранения зн
ачений критика
```

```
            self.episode_rewards = []
            self.episode_lengths = []
            self.episode_timesteps = []
            self.episode_reward = 0
            self.episode_length = 0
            self.current_steps = 0
            self.global_timesteps = 0

        def reset(self):
            self.states = []
            self.actions = []
            self.rewards = []
            self.log_probs = []
            self.dones = []
            self.values = []
            self.episode_reward = 0
            self.episode_length = 0
            self.current_steps = 0

        def add(self, state, action, reward, log_prob, done, value):
            self.states.append(state)
            self.actions.append(action)
            self.rewards.append(reward)
            self.log_probs.append(log_prob)
            self.dones.append(done)
            self.values.append(value)
            self.episode_reward += reward
            self.episode_length += 1
            self.current_steps += 1
            self.global_timesteps += 1

        def end_episode(self):
            self.episode_rewards.append(self.episode_reward)
            self.episode_lengths.append(self.episode_length)
            self.episode_timesteps.append(self.global_timesteps)
            self.episode_reward = 0
            self.episode_length = 0

        def is_full(self):
            return self.current_steps >= self.max_steps

        def get_data(self):
            return {
                "states": torch.FloatTensor(self.states).to(device),
                "actions": torch.FloatTensor(self.actions).to(device),
                "rewards": self.rewards,
                "log_probs":
torch.FloatTensor(self.log_probs).to(device),
                "dones": self.dones,
                "values": torch.FloatTensor(self.values).to(device),   #
Добавлено
            }

    # Модель актора
    class Actor(nn.Module):
        def __init__(self, state_dim, action_dim, hidden_dim=64):
            super(Actor, self).__init__()
```

```
            self.rpo_alpha = args.rpo_alpha  # Добавлено для RPO
            self.mean_net = nn.Sequential(
                nn.Linear(state_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, action_dim),
                nn.Tanh()
            )
            self.log_std = nn.Parameter(torch.zeros(action_dim))

        def forward(self, state):
            mean = self.mean_net(state) * action_high
            std = torch.exp(self.log_std).expand_as(mean)
            return mean, std

        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)
            dist = self.get_dist(state)
            action = dist.sample()
            action = torch.clamp(action, min=action_low, max=action_high)
            log_prob = dist.log_prob(action).sum(dim=-1)
            return action.cpu().numpy()[0], log_prob.item()

        def get_action_and_value(self, state, action=None):
            action_mean, action_std = self.forward(state)
            probs = Normal(action_mean, action_std)
            if action is None:
                action = probs.sample()
            else:  # RPO: добавление стохастичности
                z = torch.FloatTensor(action_mean.shape).uniform_(-
self.rpo_alpha, self.rpo_alpha).to(device)
                action_mean = action_mean + z
                probs = Normal(action_mean, action_std)
            action = torch.clamp(action, min=action_low, max=action_high)
            return action, probs.log_prob(action).sum(1),
probs.entropy().sum(1)

    # Модель критика
    class Critic(nn.Module):
        def __init__(self, state_dim, hidden_dim=64):
            super(Critic, self).__init__()
            self.network = nn.Sequential(
                nn.Linear(state_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, 1)
            )

        def forward(self, state):
            return self.network(state)
```

```
    # Вычисление возвратов и преимуществ с GAE
    def compute_returns_and_advantages(rewards, dones, values):
        returns = []
        advantages = []
        last_gae_lam = 0
        next_value = 0

        for t in reversed(range(len(rewards))):
            if t == len(rewards) - 1:
                next_nonterminal = 1.0 - dones[t]
                next_value = next_value
            else:
                next_nonterminal = 1.0 - dones[t + 1]
                next_value = values[t + 1]
            delta = rewards[t] + args.gamma * next_value *
next_nonterminal - values[t]
            last_gae_lam = delta + args.gamma * args.gae_lambda *
next_nonterminal * last_gae_lam
            advantages.insert(0, last_gae_lam)
            returns.insert(0, last_gae_lam + values[t])
            next_value = values[t]

        returns = torch.FloatTensor(returns).to(device)
        advantages = torch.FloatTensor(advantages).to(device)
        if args.normalize_advantages:
            advantages = (advantages - advantages.mean()) /
(advantages.std() + 1e-8)
        return returns, advantages

    # Обновление политики
    def update_ppo(states, actions, log_probs_old, returns, advantages):
        for _ in range(args.num_epochs):
            indices = np.random.permutation(len(states))
            for start in range(0, len(states), args.sub_batch_size):
                batch_indices = indices[start:start +
args.sub_batch_size]
                state_batch = states[batch_indices]
                action_batch = actions[batch_indices]
                log_prob_old_batch = log_probs_old[batch_indices]
                return_batch = returns[batch_indices]
                advantage_batch = advantages[batch_indices]

                action, log_prob, entropy =
actor.get_action_and_value(state_batch, action_batch)
                value = critic(state_batch).squeeze()

                ratio = torch.exp(log_prob - log_prob_old_batch)
                surr1 = ratio * advantage_batch
                surr2 = torch.clamp(ratio, 1 - args.clip_ratio, 1 +
args.clip_ratio) * advantage_batch
                actor_loss = -torch.min(surr1, surr2).mean()
                entropy_loss = entropy.mean()
                actor_loss = actor_loss - args.entropy_coef *
entropy_loss

                critic_loss = args.value_coef * nn.MSELoss()(value,
return_batch)
```

```
                actor_optimizer.zero_grad()
                actor_loss.backward()
                actor_optimizer.step()

                critic_optimizer.zero_grad()
                critic_loss.backward()
                critic_optimizer.step()

    # Сбор траекторий
    def collect_trajectories(actor, critic, buffer):
        state, _ = env.reset()
        buffer.reset()
        while not buffer.is_full():
            action, log_prob = actor.act(state)
            value                                                    =
critic(torch.FloatTensor(state).unsqueeze(0).to(device)).item()
            next_state,   reward,   terminated,   truncated,   _   =
env.step(action)
            done = terminated or truncated

            buffer.add(state, action, reward, log_prob, done, value)
            state = next_state
            if done:
                buffer.end_episode()
                state, _ = env.reset()

        return buffer.get_data()

    # Построение графика timesteps vs. episodic return
    def plot_timesteps_vs_rewards(timesteps, rewards, filename):
        plt.figure(figsize=(10, 6))
        plt.plot(timesteps, rewards, label='Episodic Return', alpha=0.5)
        if len(rewards) >= 100:
            rewards_t = torch.tensor(rewards, dtype=torch.float)
            means = rewards_t.unfold(0, 100, 1).mean(1).view(-1)
            means = torch.cat((torch.zeros(99), means))
            plt.plot(timesteps, means.numpy(), label='100-episode Moving
Average', linestyle='--')
        plt.xlabel('Timesteps')
        plt.ylabel('Episodic Return')
        plt.title('Timesteps vs. Episodic Return')
        plt.legend(loc='best')
        plt.grid(True)
        plt.savefig(filename)
        plt.close()

    # Основной цикл обучения
    def train_ppo():
        global actor, critic, actor_optimizer, critic_optimizer
        actor = Actor(state_dim, action_dim).to(device)
        critic = Critic(state_dim).to(device)
        actor_optimizer = optim.Adam(actor.parameters(), lr=lr)
        critic_optimizer = optim.Adam(critic.parameters(), lr=lr)

        buffer = RolloutBuffer(args.steps)
```

```
        for iteration in tqdm(range(args.num_iterations),
desc="Training"):
            batch = collect_trajectories(actor, critic, buffer)
            states = batch["states"]
            actions = batch["actions"]
            log_probs_old = batch["log_probs"]
            values = batch["values"]
            returns, advantages =
compute_returns_and_advantages(batch["rewards"], batch["dones"], values)
            update_ppo(states, actions, log_probs_old, returns,
advantages)

            if (iteration + 1) % args.log_interval == 0:
                avg_reward = np.mean(buffer.episode_rewards[-
args.log_interval:]) if buffer.episode_rewards else 0
                avg_length = np.mean(buffer.episode_lengths[-
args.log_interval:]) if buffer.episode_lengths else 0
                print(f"Iteration {iteration + 1}:")
                print(f"  Average Reward: {avg_reward:.2f}")
                print(f"  Average Episode Length: {avg_length:.2f}")

        return buffer.episode_timesteps, buffer.episode_rewards

    if __name__ == '__main__':
        os.makedirs("results", exist_ok=True)
        timesteps, rewards = train_ppo()
        plot_timesteps_vs_rewards(timesteps, rewards,
"results/timesteps_vs_rewards_rpo_seed_666.png")
        env.close()
```

ПРИЛОЖЕНИЕ В
ВИДЕОЗАПИСИ РЕЗУЛЬТАТОВ ОБУЧЕНИЯ

Видео доступны в репозитории: https://github.com/WorkNroller/project_0306_group_1