**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.

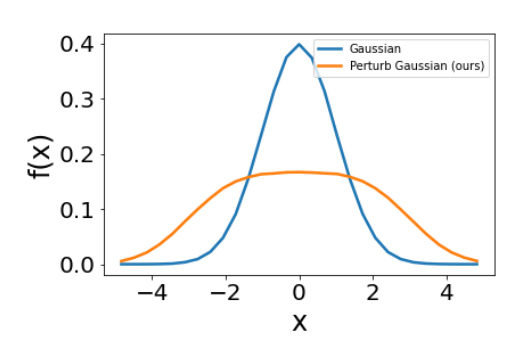
В ходе работ был рассмотрен алгоритм RPO, описанный в статье «ROBUST POLICY OPTIMIZATION IN DEEP REINFORCEMENT LEARNING». Данный алгоритм является модификацией алгоритма PPO, отличие заключается в поддержании уровня энтропии на протяжении обучения при помощи добавления равномерного шума. При этом плотность 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 ~ U(-a, a)*, создавая тем самым возмущенное среднее *μ’= μ+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.

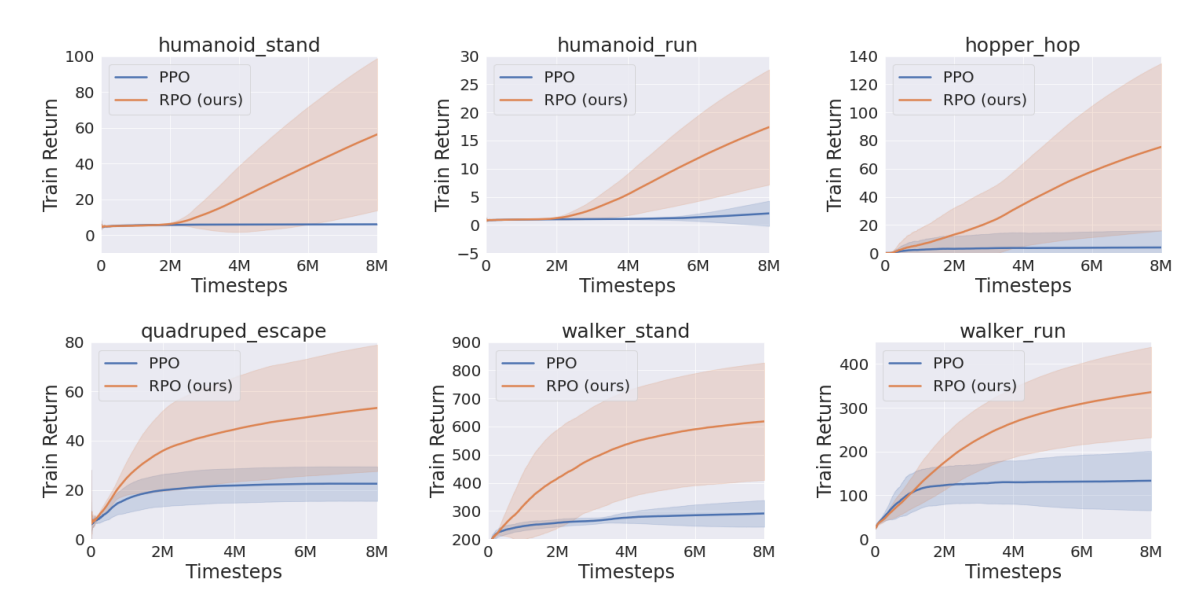
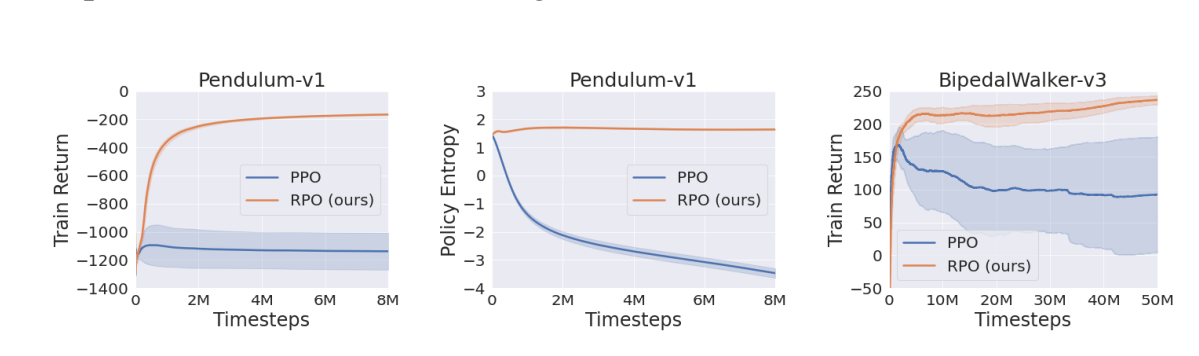
Авторы статьи протестировали свое решение на 18 различных задачах из четырех платформ. Результаты 9 из них представлены на рис. 2.

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 B. Графики представлены на рис. 3 – 5. Сверху PPO, снизу RPO.

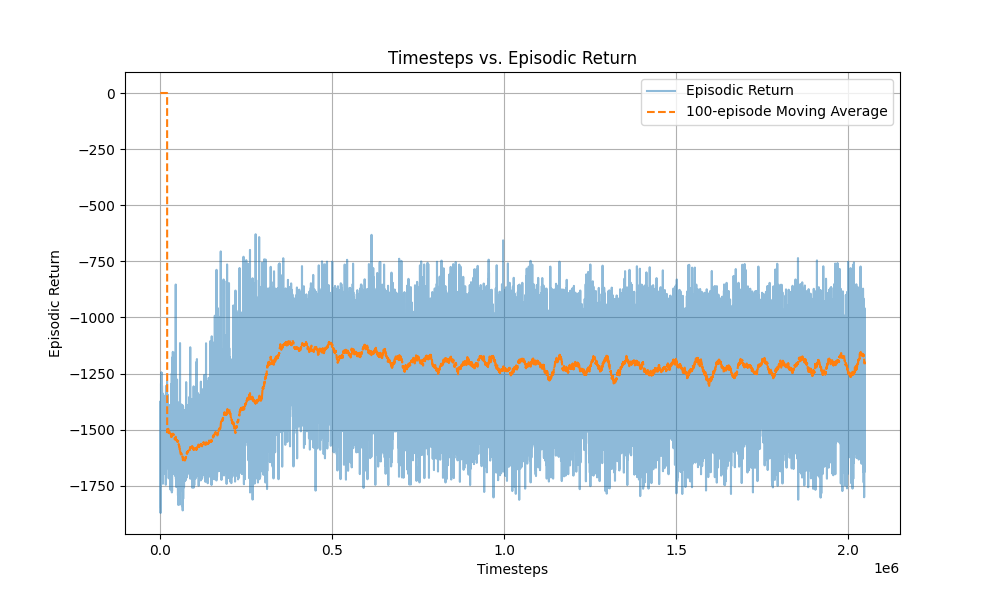
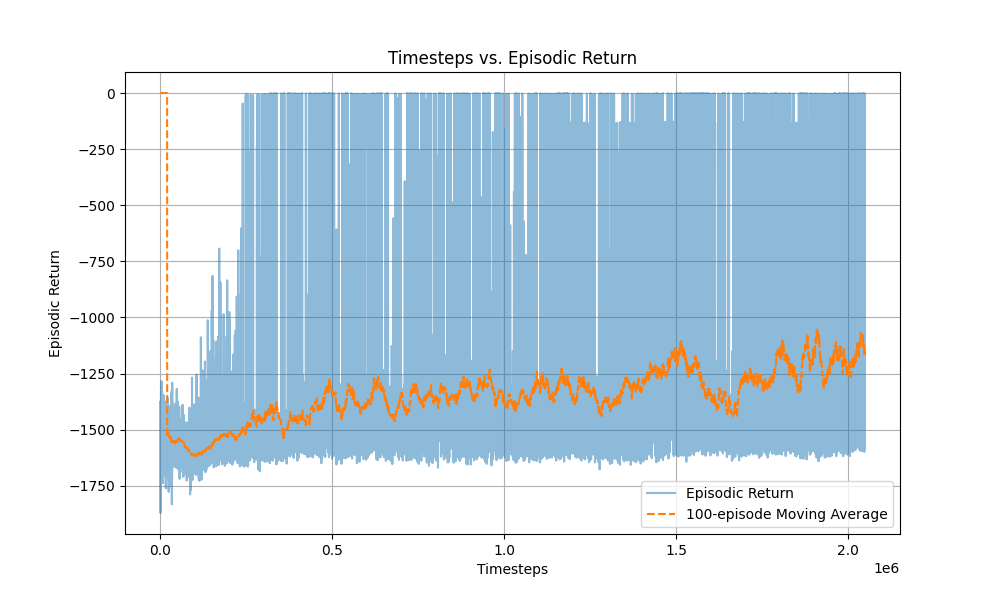


Рисунок 3 – Зависимость награды от количества шагов для сида 69.

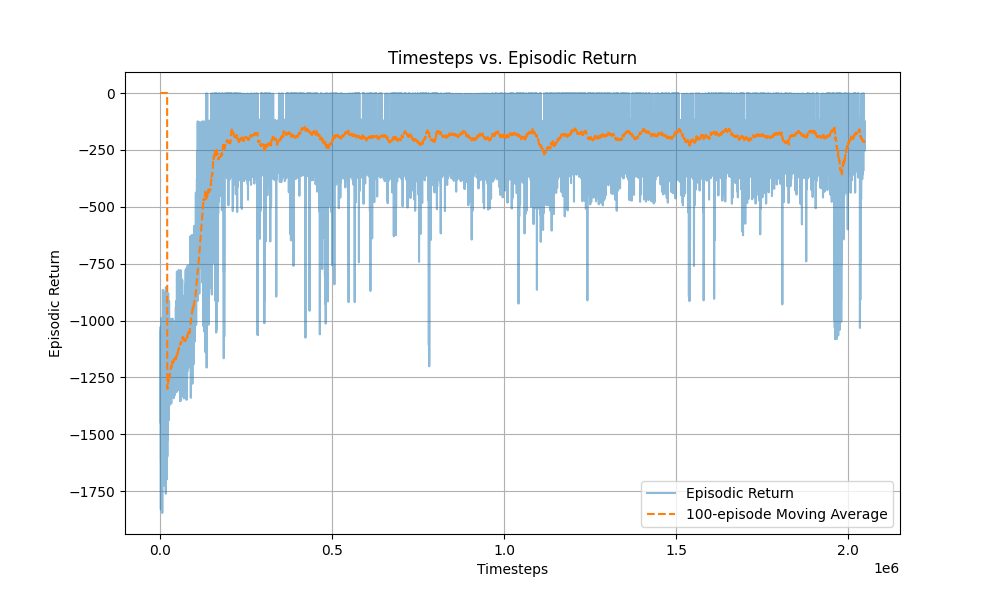
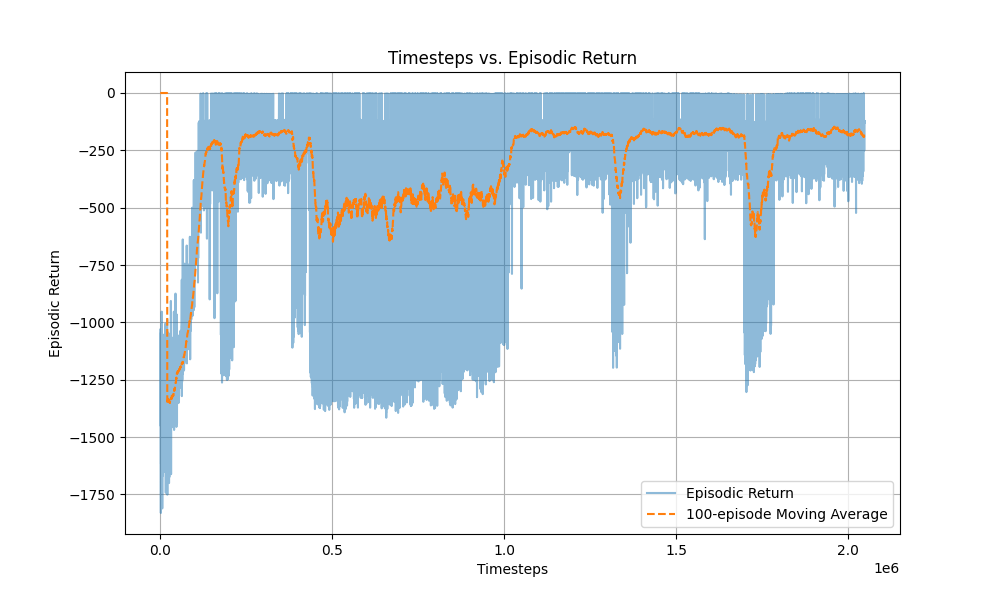


Рисунок 4 – Зависимость награды от количества шагов для сида 666.

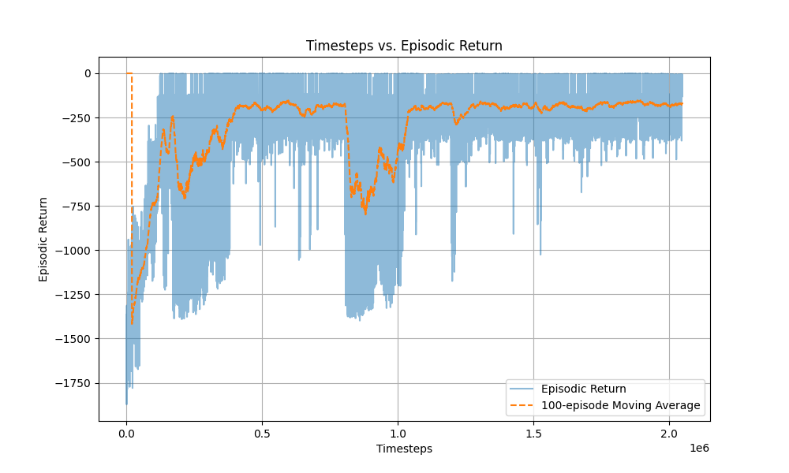
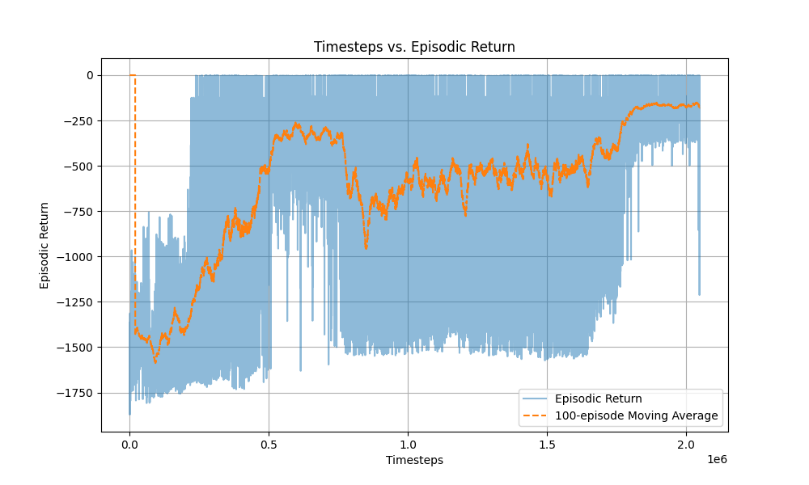


Рисунок 5 – Зависимость награды от количества шагов для сида 1337.

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

# 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=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 значений 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 B ВИДЕОзаписи результатов обучения

Видео доступны в репозитории: https://github.com/WorkNroller/project\_0306\_group\_1