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 _____ Бодунов П.А.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург

2025

Цель работы.

Реализовать алгоритм PPO с помощью библиотеки pytorch для решения задачи MountainCarContinuous-v0.

Постановка задачи.

1) Реализовать базовую версию PPO для решения задачи MountainCarContinuous-v0.
2) Проанализировать изменение в скорости обучения при изменении длины траектории (steps).
3) Подобрать оптимальный коэффициент clip_ratio.
4) Добавить нормализацию преимуществ и исследовать результаты.
5) Сравнить обучение при разных количествах эпох.

Выполнение задач.

1) **Реализация PPO.**

Среда MountainCarContinuous-v0 представляет собой задачу, где агент управляет машинкой, которая находится в долине между двумя холмами. Цель – разогнаться так, чтобы преодолеть вершину правого холма. Однако двигатель машинки недостаточно мощный, чтобы подняться напрямую, поэтому агенту нужно научиться раскачиваться, набирая инерцию. Состояние среды описывается 2 вещественными числами: положение машинки и её скорость. Пространство действий представляет собой одно значение: сила толчка, варьирующаяся от −1 до 1, применяющаяся к машинке. Награда равна $-0.1 \times action^2$ за каждое действие, чтобы машинка не использовала слишком большие толчки, и если машинка достигает конца траектории, то к награде добавляется +100. Агент заканчивает взаимодействие со средой, если машинка достигает флажка или если количество эпизодов равно 999.

Реализовано 2 нейронных сети: Actor и Critic.

Actor – отвечает за выбор действий (политику, policy). Оптимизируется, чтобы максимизировать ожидаемую награду.

Critic – ценивает "полезность" состояний, предсказывая value-функцию.

Таким образом Actor фокусируется на улучшении политики, Critic — на точной оценке качества состояний.

У Actor также есть методы:

- get_dist(self, state) — получение нормального распределения, т.к. агент соврешает действие из непрерывного диапазона [-1, 1].
- act(self, state) - выбор действия.

Также был реализован агент PPOAgent, содержащий следующие методы:

- compute_gae(self, rewards, values, dones) – вычисляет GAE.
- collect_trajectories(self, env) – собирает данные в среде: состояния, действия, логарифмы вероятностей, дисконтированные суммарные награды и флаги завершения.
- update(self) – обновляет Actor и Critic на основе собранных данных.

Обучение агента происходит в функции train(agent, env), где agent – агент, env – среда в которой находится агент.

2) **Влияние изменения длины траектории.**

Длина траектории определяет, сколько шагов агент совершает в среде перед каждым обновлением. Эксперимент проводился на значениях 512, 1024, 2048 (см. рисунок 1):

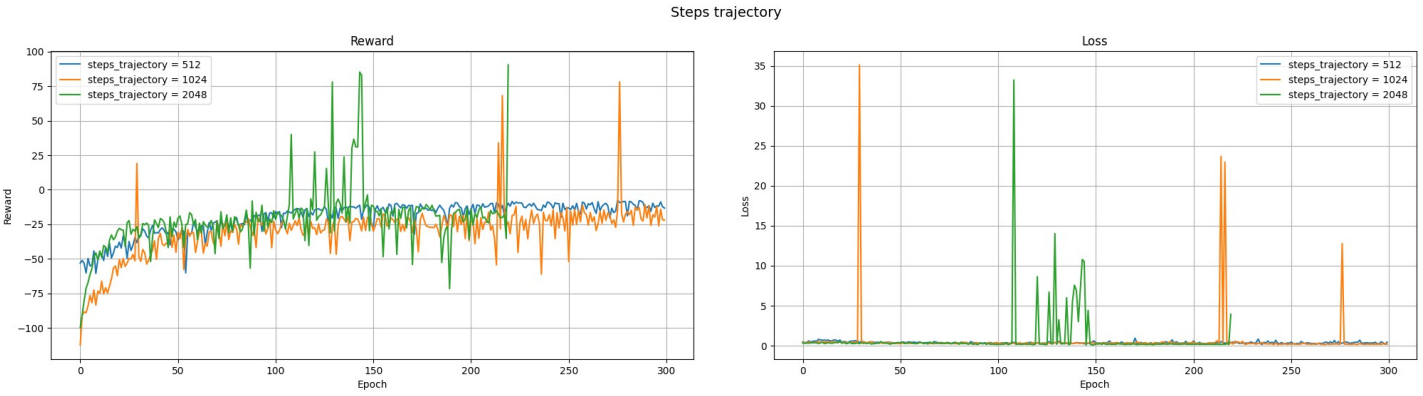

Рисунок 1 – Обучение с разной длиной трактории

Исходя из построенных графиков, можно сделать вывод, что при длине траектории 2048 модель обучилась быстрее всего, при длине траектории 1024 видно, что модель несколько раз была близка к завершению, а при длине

траектории 512 модель плохо обучается, из-за недостатка информации в окружении.

3) **Влияние изменения параметра *clip_ratio*.**

Параметр *clip_ratio* Ограничивает изменение политики, обрезая отношение вероятностей (ratio = new_prob / old_prob) в диапазоне [1 - clip_ratio, 1 + clip_ratio]. Эксперимент проводился на значениях *clip_ratio*: 0.1, 0.2, 0.3 (см. рисунок 2):

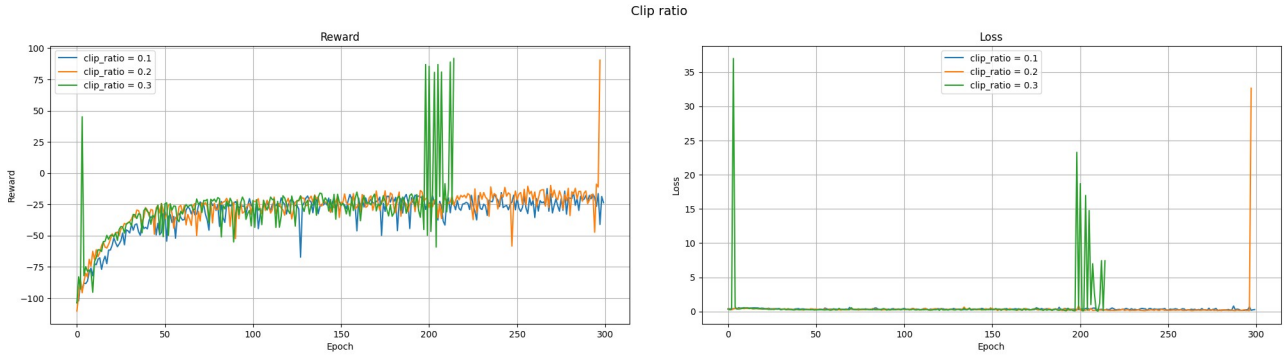

Рисунок 2 – Обучение с разными значениями *clip_ratio*

Исходя из полученных результатов, можно сделать вывод, что оптимальным является параметр 0.3, т.к. при нем модель обучилась быстрее всего.

4) **Влияние изменения параметра normalize_advantages.**

Параметр *normalize_advantages* отвечает за нормализацию advantages.

Графики награды от количества эпох и функции потери от количества эпох результатов обучения моделей представлены на рисунке 3:

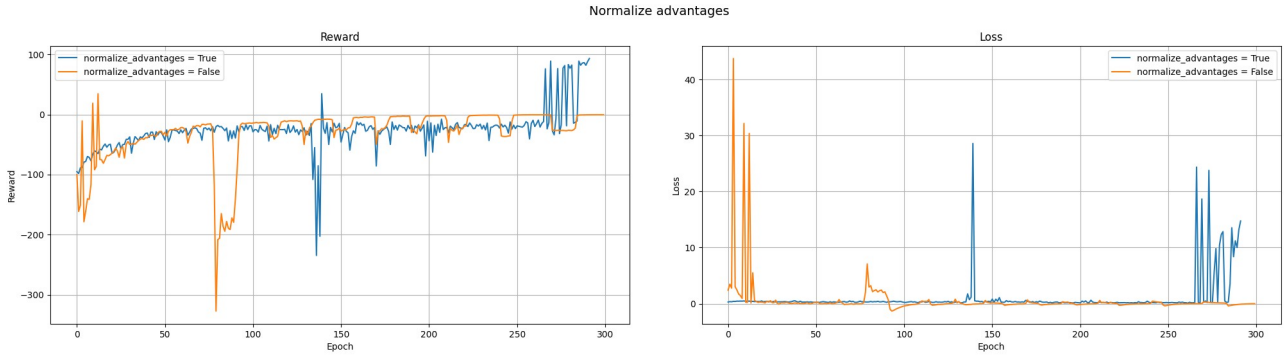

Рисунок 3 - Обучение с разными значениями *normalize_advantages*

Исходя из полученных графиков, можно сделать вывод, что нормализовав данные модель лучше обучается, что связано с уменьшением дисперсии градиентов.

5) **Влияние изменения *количества эпох*.**

Количество эпох определяет, сколько раз агент проходит по одним и тем же данным траектории перед следующим сбором новых данных. Эксперимент проводился на значениях *epoch*: 10, 20, 30 (см. рисунок 4):

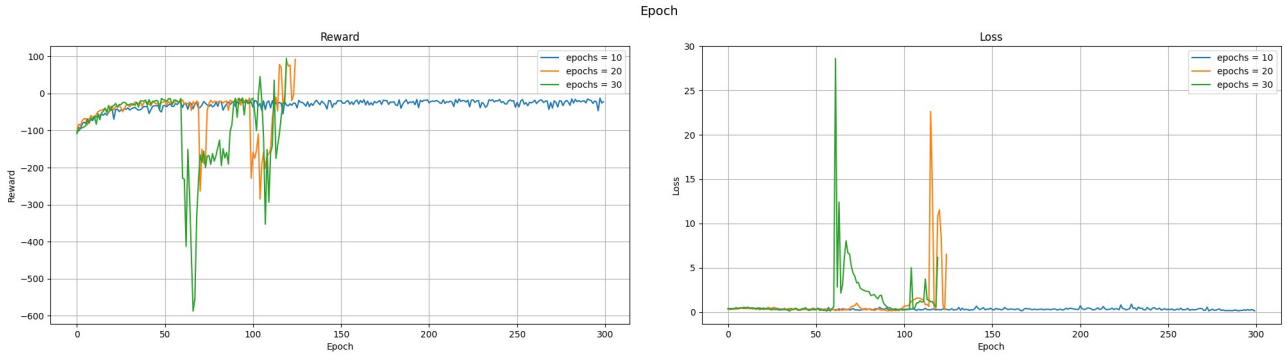

Рисунок 4 - Обучение с разными значениями *epoch*

Исходя из полученных графиков, можно сделать вывод, что модель с большим количеством эпох быстрее обучается.

Выводы.

В ходе работы был изучен и реализован алгоритм обучения с подкреплением PPO. Было исследовано влияние различных параметров на обучение PPO в среде MountainCarContinuous-v0. При большей длине траектории модель обучалась быстрее, но стоит помнить, что система может сильно улечься изучением окружения, что не будет обучаться. Параметр clip_ratio = 0.3 оказался оптимальным. Необходимо нормализовывать преимущества для уменьшения дисперсии градиентов. С увеличением количества эпох ускоряется обучение.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import math
import random
from collections import namedtuple, deque
from itertools import count
from typing import Type
from gymnasium.wrappers import RecordVideo
import gymnasium as gym
import torch
from tqdm import tqdm
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from dataclasses import dataclass
from torch.distributions import Categorical, Normal
import numpy as np

device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

torch.manual_seed(42)
np.random.seed(42)

@dataclass
class Params:
    num_episodes: int = 300
    epochs: int = 10
    batch_size: int = 64
    gamma: float = 0.99
    steps_trajectory: int = 1024
    lamb: float = 0.95
    clip_ratio: float = 0.2
    value_coef: float = 0.5
    entropy_coef: float = 0.01
    lr_actor: float = 3e-4
    lr_critic: float = 1e-3
    normalize_advantages: bool = True
```

```
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size=64):
        super(Actor, self).__init__()
        self.shared_net = nn.Sequential(
            nn.Linear(state_dim, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, hidden_size),
            nn.Tanh(),
        )
        self.mean_net = nn.Linear(hidden_size, action_dim)
        self.log_std = nn.Parameter(torch.zeros(action_dim))

    def forward(self, x):
        shared_features = self.shared_net(x)
        mean = self.mean_net(shared_features)
        return mean, self.log_std.exp()

    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)
        with torch.no_grad():
            dist = self.get_dist(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).flatten()

class PPOAgent:
    def __init__(self, env: gym.Env, params: Params = None):
        self.env = env
        self.params = params if params else Params()

        state_dim = env.observation_space.shape[0]
        self.is_continuous = isinstance(env.action_space,
gym.spaces.Box)
        action_dim = env.action_space.shape[0] if
isinstance(env.action_space, gym.spaces.Box) else
env.action_space.n

        self.actor = Actor(state_dim, action_dim).to(device)
        self.critic = Critic(state_dim).to(device)

        self.actor_optimizer = optim.Adam(self.actor.parameters(),
lr=self.params.lr_actor)
        self.critic_optimizer =
optim.Adam(self.critic.parameters(), lr=self.params.lr_critic)

    def compute_gae(self, rewards, values, dones):
        advantages = np.zeros_like(rewards)
        last_gae = 0

        for t in reversed(range(len(rewards))):
            if t == len(rewards) - 1:
                next_value = 0
            else:
                next_value = values[t + 1]

            delta = rewards[t] + self.params.gamma * next_value *
(1 - dones[t]) - values[t]
            advantages[t] = delta + self.params.gamma *
self.params.lamb * last_gae * (1 - dones[t])
            last_gae = advantages[t]
```

```
        returns = advantages + values
        if self.params.normalize_advantages:
            advantages = (advantages - advantages.mean()) /
(advantages.std() + 1e-8)

        return returns, advantages

    def collect_trajectories(self, env):
        states, actions, log_probs, rewards, dones = [], [], [],
[], []
        episode_rewards = []

        state, _ = env.reset()
        ep_reward = 0.0

        for _ in range(self.params.steps_trajectory):
            action, log_prob = self.actor.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)

            ep_reward += reward
            state = next_state

            if done:
                episode_rewards.append(ep_reward)
                state, _ = env.reset()
                ep_reward = 0.0

        if len(episode_rewards) == 0 or ep_reward > 0:
           episode_rewards.append(ep_reward)

        with torch.no_grad():
```

```
            values =
self.critic(torch.FloatTensor(states).to(device)).cpu().numpy()

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

        return {
            "states": np.array(states),
            "actions": np.array(actions),
            "log_probs": np.array(log_probs),
            "returns": np.array(returns),
            "advantages": np.array(advantages),
            "episode_rewards": np.array(episode_rewards)
        }

    def update(self, batch):
        states = torch.FloatTensor(batch["states"]).to(device)
        actions = torch.LongTensor(batch["actions"]).to(device)
        old_log_probs =
torch.FloatTensor(batch["log_probs"]).to(device)
        returns = torch.FloatTensor(batch["returns"]).to(device)
        advantages =
torch.FloatTensor(batch["advantages"]).to(device)

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

        iteration_loss = []

        for _ in range(self.params.epochs):
            np.random.shuffle(indices)

            for start in range(0, dataset_size,
self.params.batch_size):
                end = start + self.params.batch_size
                if end > dataset_size:
                    end = dataset_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 = self.actor.get_dist(batch_states)
                new_log_probs =
dist.log_prob(batch_actions).sum(dim=-1)

                ratio = torch.exp(new_log_probs -
batch_old_log_probs)

                entropy_loss = dist.entropy().mean()

                surr1 = ratio * batch_advantages
                surr2 = (torch.clamp(ratio, 1.0 -
self.params.clip_ratio, 1.0 + self.params.clip_ratio) *
batch_advantages)

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

                values = self.critic(batch_states)
                critic_loss = nn.functional.mse_loss(values,
batch_returns)

                loss = actor_loss + self.params.value_coef *
critic_loss - self.params.entropy_coef * entropy_loss

                self.actor_optimizer.zero_grad()
                self.critic_optimizer.zero_grad()
                loss.backward()
                self.actor_optimizer.step()
                self.critic_optimizer.step()

                iteration_loss.append(loss.item())

        return iteration_loss

def train(agent, env):
    reward_history = []
```

```
    loss_history = []
    episode_tqdm = tqdm(range(agent.params.num_episodes))
    for episode in episode_tqdm:
        batch = agent.collect_trajectories(env)

        iteration_loss = agent.update(batch)

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

        reward_history.append(avg_reward)
        loss_history.append(avg_loss)

        episode_tqdm.desc = f"Avg Reward: {avg_reward}, Loss:
{avg_loss}"

        if avg_reward >= env.spec.reward_threshold:
            print(f"\nTask is completed for {episode} episodes")
            break

    return reward_history, loss_history

def plot_graphics(results, labels, main_title):
    fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(21,
6))
    fig.tight_layout(pad=5.0)
    fig.suptitle(main_title, fontsize=14)

    ax1.set_title("Reward")
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Reward')
    ax1.grid(True)

    ax2.set_title("Loss")
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Loss')
    ax2.grid(True)

    for (rewards, losses), label in zip(results, labels):
        ax1.plot(rewards, label=label)
```

```
        ax2.plot(losses, label=label)

    ax1.legend()
    ax2.legend()

    plt.show()

def experiment_steps_trajectory(env):
    steps_trajectories = [512, 1024, 2048]
    labels = []
    results = []
    for steps_trajectory in steps_trajectories:
        label = f"steps_trajectory = {steps_trajectory}"
        labels.append(label)
        agent = PPOAgent(env,
Params(steps_trajectory=steps_trajectory))

        print(f"{label} model train")
        results.append(train(agent, env))

    plot_graphics(results, labels, "Steps trajectory")

def experiment_clip_ratio(env):
    clip_ratios = [0.1, 0.2, 0.3]
    labels = []
    results = []
    for clip_ratio in clip_ratios:
        label = f"clip_ratio = {clip_ratio}"
        labels.append(label)
        agent = PPOAgent(env, Params(clip_ratio=clip_ratio))

        print(f"{label} model train")
        results.append(train(agent, env))

    plot_graphics(results, labels, "Clip ratio")

def experiment_normalize_advantages(env):
    normalizations = [True, False]
    labels = []
    results = []
```

```
    for normalize_advantages in normalizations:
        label = f"normalize_advantages = {normalize_advantages}"
        labels.append(label)
        agent = PPOAgent(env,
Params(normalize_advantages=normalize_advantages))

        print(f"{label} model train")
        results.append(train(agent, env))

    plot_graphics(results, labels, "Normalize advantages")

def experiment_epoch(env):
    epochs = [10, 20, 30]
    labels = []
    results = []
    for epoch in epochs:
        label = f"epochs = {epoch}"
        labels.append(label)
        agent = PPOAgent(env, Params(epochs=epoch))

        print(f"{label} model train")
        results.append(train(agent, env))

    plot_graphics(results, labels, "Epoch")

env = gym.make("MountainCarContinuous-v0")
experiment_steps_trajectory(env)
experiment_clip_ratio(env)
experiment_normalize_advantages(env)
experiment_epoch(env)



def visualize_agent_performance(agent):
    env = gym.make("MountainCarContinuous-v0",
render_mode="rgb_array")
    env = RecordVideo(env,
                      video_folder="./videos",
                      name_prefix="mountaincar_ppo",
                      episode_trigger=lambda x: True)
```

```
    state, _ = env.reset()
    done = False
    total_reward = 0

    while not done:
        action, _ = agent.actor.act(state)
        state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        total_reward += reward



    env.close()
    print("Видео сохранены в папку ./videos")

env = gym.make("MountainCarContinuous-v0")
agent = PPOAgent(env, Params(num_episodes=500, epochs=20,
batch_size=128, clip_ratio=0.3, entropy_coef=0.05))
results = train(agent, env)
plot_graphics([results], ['1'], "Graphics")
visualize_agent_performance(agent)
```