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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Обучение с подкреплением»

Тема: Реализация PPO для среды MountainCarContinuous-v0

Студент гр. 0310 _____ Афанасьев Н. С.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург

2025

Цель работы.

Реализация PPO для среды MountainCarContinuous-v0

Задание.

Окружение: mountain_car_continuous

Задания для эксперимента:

- Измените длину траектории (steps).
- Подберите оптимальный коэффициент clip_ratio.
- Добавьте нормализацию преимуществ.
- Сравните обучение при разных количествах эпох.

Теоретические положения.

Алгоритм Proximal Policy Optimization (PPO) – это популярный метод обучения с подкреплением, который сочетает эффективность и стабильность. Он относится к семейству policy gradient методов и использует специальный механизм clipped surrogate objective, чтобы избежать слишком больших обновлений политики.

Основные параметры:

- Длина траектории – определяет, сколько шагов агент делает в среде перед обновлением политики.
- Коэффициент отсечения – контролирует, насколько сильно политика может отклоняться от старой политики.
- Нормализация преимуществ – преимущества вычисляются через GAE (Generalized Advantage Estimation) и нормализуются для уменьшения дисперсии.
- Количество эпох – сколько раз пройти по одним и тем же данным для обновления политики.

Алгоритм PPO собирает траектории фиксированной длины, вычисляет преимущества через GAE (Generalized Advantage Estimation) и нормализует их, затем в течение нескольких эпох обновляет политику, минимизируя clipped

surrogate objective – ограничивая изменения вероятностей действий с помощью clip_ratio, чтобы избежать резких обновлений, а также оптимизирует функцию ценности (value function) через MSE. Весь процесс повторяется, пока политика не сойдётся.

Выполнение работы.

Реализация алгоритма

Нейронная сеть актора (Actor) состоит из последовательности линейных слоев с активацией Tanh, за которыми следуют отдельные слои для предсказания среднего значения и логарифма стандартного отклонения нормального распределения действий. Метод act позволяет получить действие и его логарифмическую вероятность для заданного состояния. Критик (Critic) реализован как нейронная сеть, предсказывающая значение состояния. Функция compute_advantages вычисляет возвраты и преимущества для обновления политики, с возможностью нормализации преимуществ.

Основная функция run_experiment выполняет обучение агента в среде. На каждой итерации собираются траектории, вычисляются преимущества и обновляются параметры актора и критика с использованием мини-пакетов. Функция plot_results сохраняет графики с результатами экспериментов. Проводятся эксперименты с различными значениями гиперпараметров: размером буфера (experiment_num_steps), коэффициентом отсечения (experiment_clip_ratio), нормализацией преимуществ (experiment_advantage_norm) и количеством эпох PPO (experiment_epochs). Результаты каждого эксперимента сохраняются в виде графиков в папке plots.

Изменение длины траектории

Алгоритм был запущен для разных длин траекторий (рис. 1)

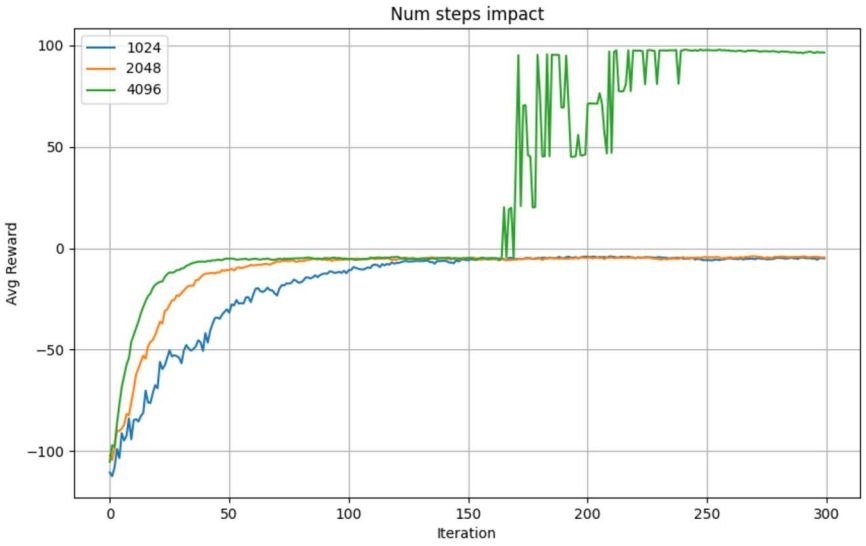

Рисунок 1 – Средняя награда для разных длин траекторий

По результатам можно видеть, что чем больше длина траекторий тем быстрее поднимается вознаграждение, и тем меньше происходят скачки. При достижении 150 итераций значения награждений становятся примерно одинаковыми, но потом награда для длины траектории 4096 скачками стремится к 100, то есть это единственный случай достижения цели.

Изменение коэффициента clip_ratio

Алгоритм был запущен для разных коэффициентов clip_ratio (рис. 2)

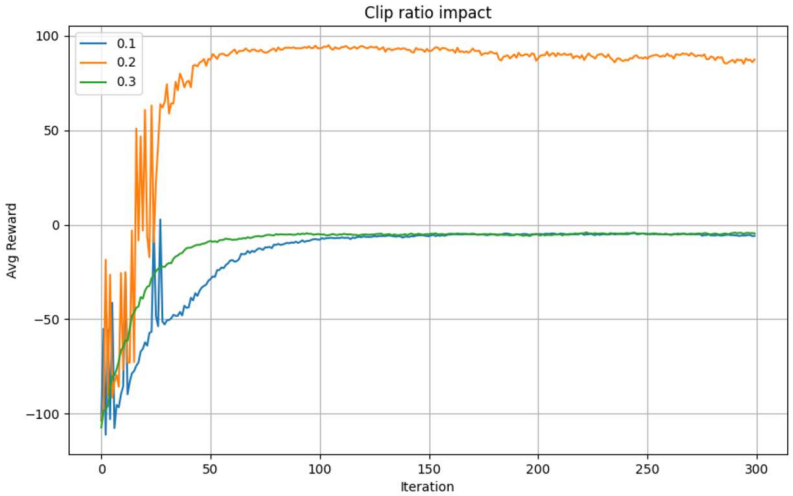

Рисунок 2 – Средняя награда для разных clip_ratio

По результатам можно видеть, что при малом значении коэффициента отсечения (clip_ratio=0.1) обучение медленное и нестабильное. При высоком значении (clip_ratio=0.3) более быстрое и гладкое. При оптимальном значении (clip_ratio=0.2) агент достигает цели, при этом само обучение более резкое и скачкообразное.

Добавление нормализации преимуществ

Алгоритм был запущен с добавлением нормализации преимуществ (рис. 3)

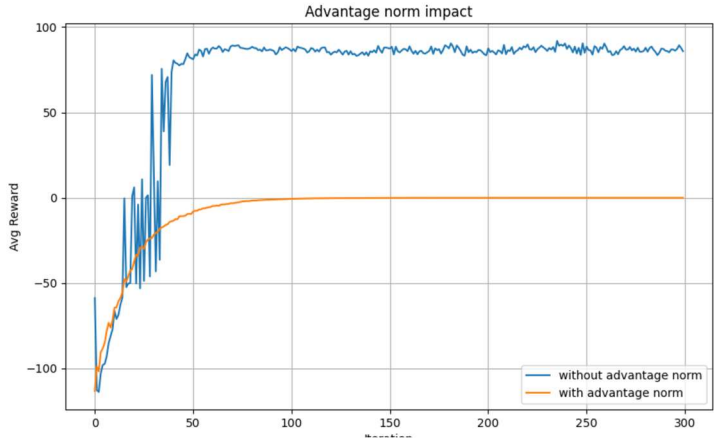

Рисунок 3 – Средняя награда при добавлении нормализации преимуществ

По результатам можно видеть, что с добавлением нормализации преимуществ обучение становится быстрее и стабильнее, но в данном случае результат стал хуже, так как агент перестал достигать цели.

Изменение количества эпох

Алгоритм был запущен для разного количества эпох (рис. 4)

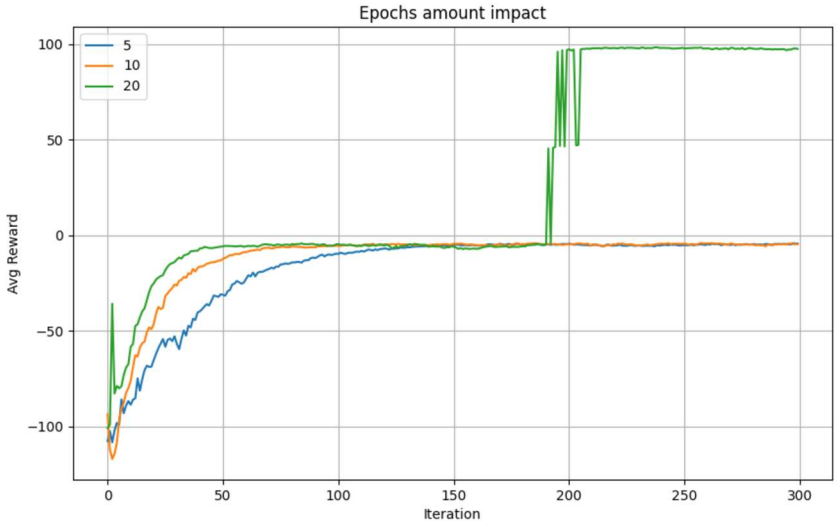

Рисунок 4 – Средняя награда для разного количества эпох

По результатам можно видеть, что при увеличении числа эпох процесс обучения увеличивается, скачков становится меньше. При значении количества эпох=20 агент достигает цели.

Разработанный программный код см. в приложении А.

Выводы.

Был изучен на практике алгоритм PPO. Алгоритм был реализован для среды MountainCarContinuous-v0, также был проведён анализ работы алгоритма в зависимости от различных входных параметров.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.py:

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

os.makedirs('plots', exist_ok=True)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

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(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()

params = {
    "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
}

def run_experiment(num_steps=params["default_num_steps"],
                   clip_ratio=params["default_clip_ratio"],
                   use_advantage_norm=False,
                   ppo_epochs=params["default_ppo_epochs"]):

    env = gym.make("MountainCarContinuous-v0")
    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=params["lr"])
    critic_optim = optim.Adam(critic.parameters(), lr=params["lr"])

    rewards_history = []

    for iteration in range(params["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, params["mini_batch_size"]):
                end = start + params["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 + params["value_coef"] * critic_loss -
params["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 + params["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

def experiment_num_steps():
    steps_params = [1024, 2048, 4096]
    results_steps = {}
    for steps in steps_params:
        print(f"num_steps={steps}")
        results_steps[steps]         =         run_experiment(num_steps=steps,
use_advantage_norm=False)
    return results_steps

def experiment_clip_ratio():
    clip_params = [0.1, 0.2, 0.3]
    results_clip = {}
    for clip in clip_params:
        print(f"clip_ratio={clip}")
        results_clip[clip]          =          run_experiment(clip_ratio=clip,
use_advantage_norm=False)
    return results_clip

def experiment_advantage_norm():
    results_norm = {
        "without advantage norm": run_experiment(use_advantage_norm=False),
        "with advantage norm": run_experiment(use_advantage_norm=True)
    }
    return results_norm

def experiment_epochs():
    epochs_params = [5, 10, 20]
    results_epochs = {}
    for epochs in epochs_params:
        print(f"ppo_epochs={epochs}")
        results_epochs[epochs] = run_experiment(ppo_epochs=epochs)
    return results_epochs

if __name__ == "__main__":
    print("Running num steps experiment...")
    r_steps = experiment_num_steps()
    plot_results(r_steps, "Num steps impact", "experiment_steps.png")

    print("Running clip ratio experiment...")
    r_clip = experiment_clip_ratio()
    plot_results(r_clip, "Clip ratio impact", "experiment_clip.png")

    print("Running advantage norm experiment...")
    r_norm = experiment_advantage_norm()
    plot_results(r_norm, "Advantage norm impact", "experiment_norm.png")

    print("Running epochs amount experiment...")
    r_epochs = experiment_epochs()
    plot_results(r_epochs, "Epochs amount impact", "experiment_epochs.png")
```