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

ОТЧЕТ

по практической работе №2

по дисциплине «Обучение с подкреплением»

Тема: Реализация PPO для среды MountainCarContinuous-v0

Студент гр. 0306 Кумаритов А.О.

Преподаватель Глазунов. С.А.

Санкт-Петербург
2025

Задание:

Реализовать PPO для среды MountainCarContinuous-v0.

Задания для эксперимента:

Измените длину траектории (steps).

Подберите оптимальный коэффициент clip_ratio.

Добавьте нормализацию преимуществ.

Сравните обучение при разных количествах эпох.

Описание среды:

Action space состоит из числа от -1 до 1 умноженного на мощность 0.0015, которое представляет силу, приложенную к машине.

Observation space состоит из 2 чисел:

Car position - позиция машины по оси X, значения от -1.2 до 0.6

Car velocity - скорость машины, значение -0.07 до 0.07

Rewards негативный равен -0.1 * $action^2$, при достижении цели добавляется +100 к негативной награде.

Starting state - position присваивается случайное значение от -0.6 до -0.4, velocity присваивается 0.

Конец эпизода в двух случаях:

Car position больше или равен 0.45

Длительность эпизода равна 999

Описание алгоритма:

Базовое описание алгоритма представлено на рисунке 1.

Алгоритм 21: Proximal Policy Optimization (PPO)

Гиперпараметры: M — количество параллельных сред, N — длина роллаутов, B — размер мини-батчей, $\mathbf{n\_epochs}$ — количество эпох, λ — параметр GAE-оценки, ϵ — параметр обрезки для актёра, $\hat{\epsilon}$ — параметр обрезки для критика, $V_\phi(s)$ — нейросеть с параметрами ϕ, $\pi_\theta(a \mid s)$ — нейросеть для стратегии с параметрами θ, α — коэф. масштабирования лосса критика, SGD оптимизатор.

Инициализировать θ, ϕ
На каждом шаге:

1. в каждой параллельной среде собрать роллаут длины N, используя стратегию π_θ, сохраняя вероятности выбора действий как $\pi^{\text{old}}(a \mid s)$, а выход критика на встреченных состояниях как $V^{\text{old}}(s) \leftarrow V_\phi(s)$
2. для каждой пары s, a из роллаутов посчитать одношаговую оценку Advantage:
$$\Psi_{(1)}(s,a) \coloneqq r + \gamma(1 - \text{done}')V_\phi(s') - V_\phi(s)$$
3. посчитать GAE-оценку:
$$\Psi_{\text{GAE}}(s_{N-1}, a_{N-1}) \coloneqq \Psi_{(1)}(s_{N-1}, a_{N-1})$$
4. для t от $N-2$ до 0:
   - $\Psi_{\text{GAE}}(s_t, a_t) \coloneqq \Psi_{(1)}(s_t, a_t) + \gamma\lambda(1 - \text{done}_t)\Psi_{\text{GAE}}(s_{t+1}, a_{t+1})$
5. посчитать таргет для критика:
$$y(s) \coloneqq \Psi_{\text{GAE}}(s,a) + V_\phi(s)$$
6. составить датасет из шестёрок $(s, a, \Psi_{\text{GAE}}(s,a), y(s), \pi^{\text{old}}(a \mid s), V^{\text{old}}(s))$
7. выполнить $\mathbf{n\_epochs}$ проходов по роллауту, генерируя мини-батчи пятёрок $\mathbb{T} \coloneqq (s, a, \Psi_{\text{GAE}}(s,a), y(s), \pi^{\text{old}}(a \mid s), V^{\text{old}}(s))$ размером B; **для каждого мини-батча:**
   - вычислить лосс критика:
$$\text{Loss}_1(\mathbb{T}, \phi) \coloneqq (y(s) - V_\phi(s))^2$$
$$\text{Loss}_2(\mathbb{T}, \phi) \coloneqq \left(y(s) - V^{\text{old}}(s) - \text{clip}(V_\phi(s) - V^{\text{old}}(s), -\hat{\epsilon}, \hat{\epsilon})\right)^2$$
$$\text{Loss}^{\text{critic}}(\phi) \coloneqq \frac{1}{B}\sum_{\mathbb{T}} \max(\text{Loss}_1(\mathbb{T}, \phi), \text{Loss}_2(\mathbb{T}, \phi))$$
   - сделать шаг градиентного спуска по ϕ, используя $\nabla_\phi \text{Loss}^{\text{critic}}(\phi)$
   - нормализовать $\Psi_{\text{GAE}}(s,a)$ по батчу, чтобы в среднем значения равнялись 0, а дисперсия — 1.
   - посчитать коэффициенты в importance sampling:
$$r_\theta(\mathbb{T}) \coloneqq \frac{\pi_\theta(a \mid s)}{\pi^{\text{old}}(a \mid s)}$$
   - посчитать обрезанную версию градиентов:
$$r_\theta^{\text{clip}}(\mathbb{T}) \coloneqq \text{clip}(r_\theta(\mathbb{T}), 1 - \epsilon, 1 + \epsilon)$$
   - вычислить градиент для актёра:
$$\nabla_\theta^{\text{actor}} \coloneqq \frac{1}{B}\sum_{\mathbb{T}} \nabla_\theta \min\left(r_\theta(\mathbb{T})\Psi_{\text{GAE}}(s,a), r_\theta^{\text{clip}}(\mathbb{T})\Psi_{\text{GAE}}(s,a)\right)$$
   - сделать шаг градиентного подъёма по θ, используя $\nabla_\theta^{\text{actor}}$

Рис. 1 - алгоритм Proximal Policy Optimization (PPO)

В реализации следующие параметры:

GAMMA - коэффициент дисконтирования

GAE_LAMBDA - параметр Generalized Advantage Estimation

ENTROPY_COEFFICIENT - коэффициент энтропии

CLIP_RATIO - коэффициент обрезки

STEPS - длина траектории

ITERATIONS - количество итераций обучения

N_EPOCH - количество эпох

BATCH_SIZE - размер батча

LR - скорость обучения оптимизаторов

WITH_NORMALIZATION - флаг нормализации преимуществ

В реализации используется несколько классов:

Agent - класс, реализующий инициализацию агента и все необходимые для обучения методы.

Actor, Critic - две нейросети. Actor рассчитывает действие, а Critic оценивает состояние. Их конфигурация представлена на рисунке 2 и 3 соответственно:

task2_0306_Kumaritov > actor.py > ...

```python
import torch
import torch.nn as nn
from torch.distributions import Normal


class Actor(nn.Module):
    def __init__(self, n_observations, n_actions, hidden_size=256):
        super(Actor, self).__init__()
        self.log_std = nn.Parameter(torch.zeros(n_actions))
        self.model = nn.Sequential(
            nn.Linear(n_observations, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, hidden_size // 2),
            nn.Tanh(),
            nn.Linear(hidden_size // 2, n_actions)
        )

    def forward(self, x):
        return self.model(x)

    def get_dist(self, x):
        mean = self.forward(x)
        std = torch.exp(self.log_std).clamp(1e-6, 1)
        return Normal(mean, std)

    def get_action(self, x):
        x = torch.as_tensor(x, dtype=torch.float32)
        dist_x = self.get_dist(x)
        action = dist_x.rsample()
        log_prob = dist_x.log_prob(action).sum(-1)
        return action, log_prob

```

Рис. 2 - конфигурация нейронной сети Actor

task2_0306_Kumaritov > critic.py > ...

```python
import torch.nn as nn


class Critic(nn.Module):
    def __init__(self, n_observations, hidden_size=128):
        super(Critic, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(n_observations, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, hidden_size // 2),
            nn.Tanh(),
            nn.Linear(hidden_size // 2, 1)
        )

    def forward(self, x):
        return self.model(x)

```

Рис. 3 - конфигурация нейронной сети Critic

Выполнение экспериментов.

Изменение длины траектории (steps):

В рамках эксперимента по сравнению влияния различных значений параметра steps на обучение алгоритма PPO было проведено три запуска с параметром steps равным 1024, 2048, 4096. Этот параметр определяет длину траектории или количество шагов агента до обновления. Результат представлен на рисунке 4:

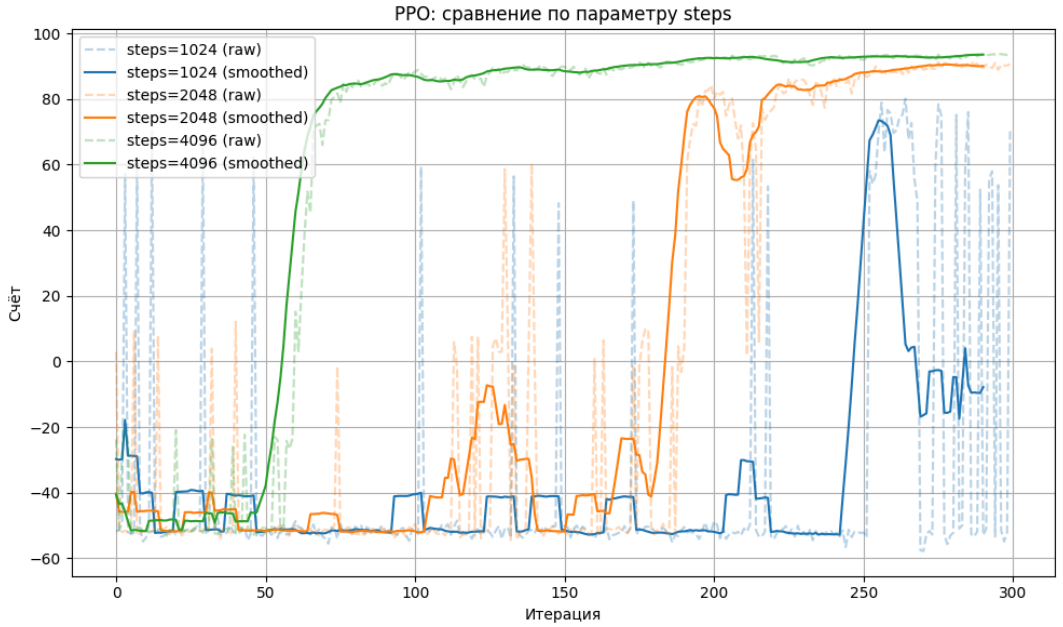

Рис. 4 - влияние различного значения steps

Лучший результат обучения показала длина траектории равная 4096. Стабильность получения награды достигалась к 60-ой итерации. Среднее значение длины траектории в 2048 показало средний результат. Стабильность награды достигалась к 200-ой итерации. Из-за недостатка статистики для обновления политики длина траектории 1024 показала худший результат. Стабильного значения награды достигнуто не было. В данных условиях использование большей длины траектории улучшает процесс обучения, но увеличивает вычислительные затраты.

Подбор оптимального коэффициента clip_ratio:

В рамках эксперимента по сравнению влияния различных значений параметра clip_ratio на обучение алгоритма PPO было проведено три запуска с параметром clip_ratio равным 0.1, 0.2, 0.3. Этот параметр определяет резкость изменения политики. Результат представлен на рисунке 5:

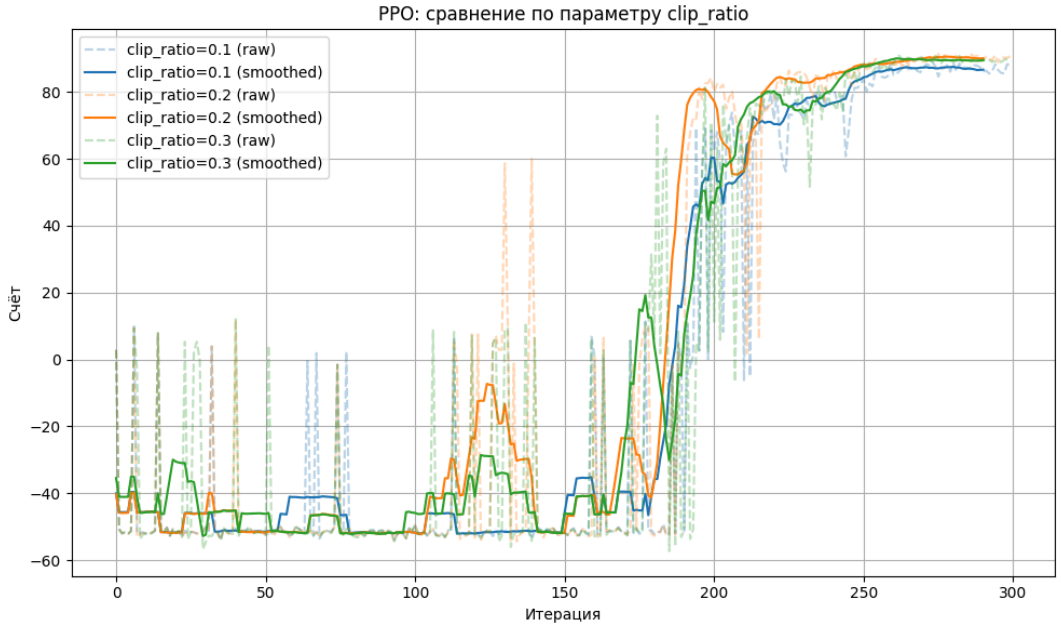

Рис. 5 - влияние различных значений clip_ratio

Лучший результат обучения показало значение clip_ratio равное 0.2. Стабильно высокие награды достигались к 200-ой итерации. Самое большое значение clip_ratio в 0.3 показало сравнимый, но менее стабильный результат. Запуск с clip_ratio равной 0.1 показал худший и нестабильный результат обучения. Это говорит о важности баланса между скоростью обучения и стабильностью.

Добавление нормализации преимуществ:

В рамках эксперимента по сравнению влияния нормализации преимуществ на обучение алгоритма PPO было проведено два запуска с включённой и выключенной нормализацией преимуществ. Результат представлен на рисунке 6:

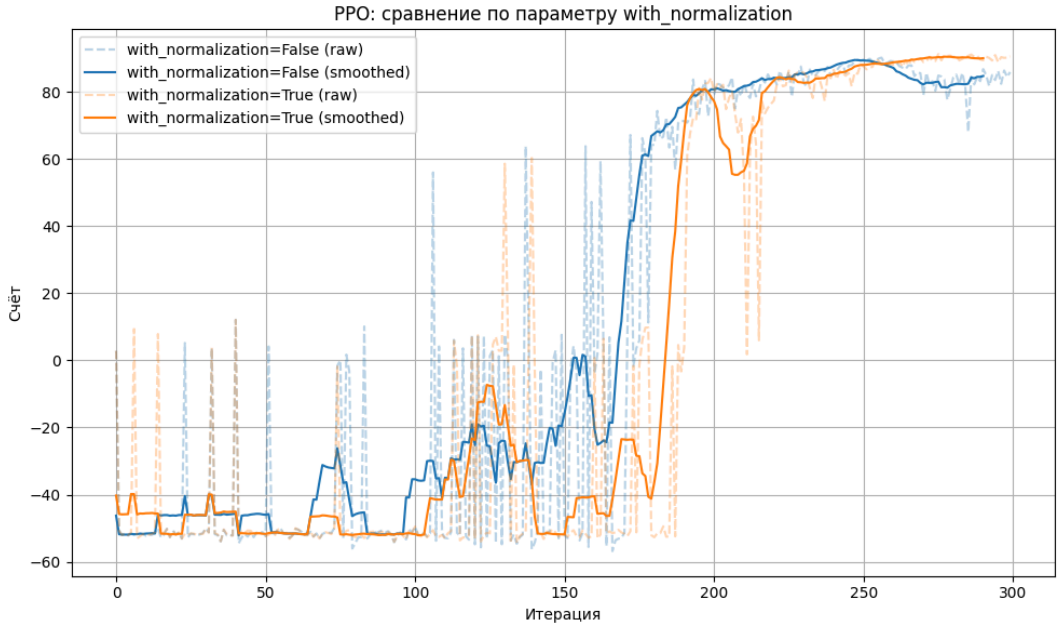

Рис. 6 - влияние нормализации

Запуск с нормализацией не дал ощутимых преимуществ на старте обучения и замедлил процесс. Без нормализации выход к высоким наградам был к 180-ой итерации, в то время как для запуска с нормализацией к 225-ой. После выхода к высоким наградам запуск со нормализацией показывает большую стабильность, чем запуск без нормализации. Таким образом нормализация может замедлить достижение высоких наград, но способствует повышению стабильности.

Сравнение обучения при разных количествах эпох:

В рамках эксперимента по сравнению влияния различных значений параметра n_epoch на обучение алгоритма PPO было проведено три запуска с параметром n_epoch равным 5, 10, 20. Этот параметр определяет количество эпох для итерации. Результат представлен на рисунке 7:

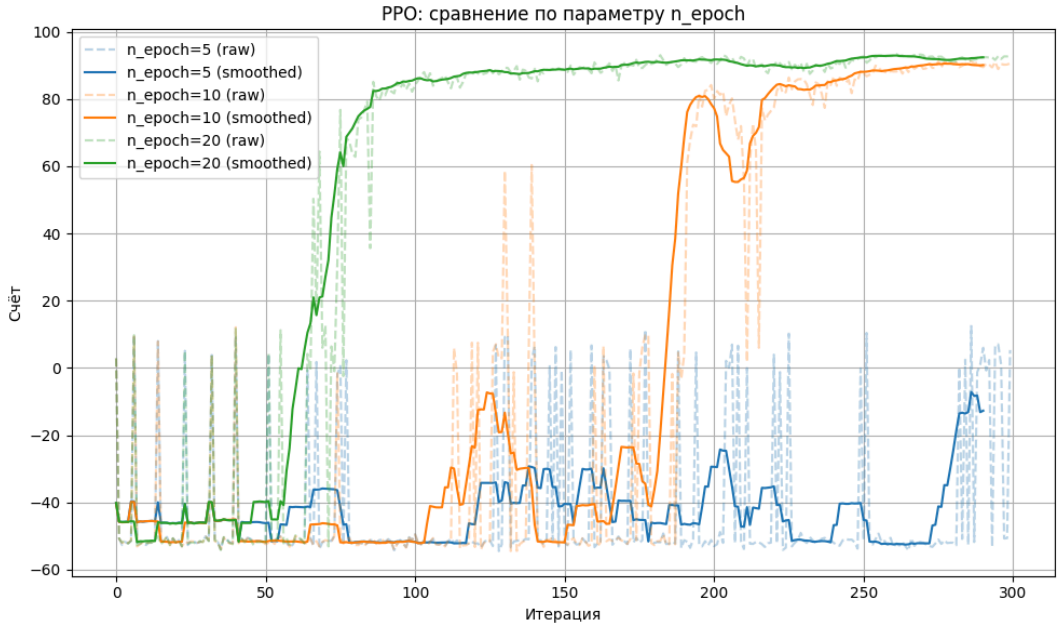

Рис. 7 - влияние различных значений n_epoch

Лучший результат обучения показало значение n_epoch равное 20. Стабильно высокие награды достигались к 75-ой итерации. Значение n_epoch равное 10 достигло стабильно высоких наград к 225-ой итерации. А значение n_epoch равное 5 показало худший результат, не достигнув стабильно высоких наград за 300 итераций. В данных условиях использование большего значения n_epoch улучшает процесс обучения, но увеличивает вычислительные затраты.

Выводы.

Был реализован PPO для среды MountainCarContinuous-v0. Были проведены исследования при различных значениях steps, clip_ratio, нормализации, n_epoch. Оптимальным для высоких и стабильных результатов обучения, исходя из полученных данных, является запуск с steps = 4096, clip_ratio = 0.2, включённой нормализацией, n_epoch = 20.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ.

Исходный код main.py

```
import gymnasium as gym
import matplotlib.pyplot as plt
import numpy as np
import torch

from agent import Agent


def run_experiment(steps, clip_ratio, n_epoch, with_normalization, seed=42):
    env = gym.make("MountainCarContinuous-v0")
    torch.manual_seed(seed)
    env.reset(seed=seed)
    np.random.seed(seed)
    agent = Agent(
        env=env,
        gamma=0.99,
        gae_lambda=0.95,
        entropy_coefficient=0.4,
        clip_ratio=clip_ratio,
        steps=steps,
        iterations=300,
        n_epoch=n_epoch,
        batch_size=32,
        lr=3e-4,
        with_normalization=with_normalization
```

```
    )
    agent.train()
    return agent.scores


def run_and_plot(param_values, param_name, train_kwargs, filename_prefix):
    plt.figure(figsize=(10, 6))

    color_cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']

    for i, val in enumerate(param_values):
        print(f"Обучение при {param_name} = {val}")
        kwargs = train_kwargs(val)
        scores = run_experiment(
            steps=kwargs.get("steps", 2048),
            clip_ratio=kwargs.get("clip_ratio", 0.2),
            n_epoch=kwargs.get("n_epoch", 10),
            with_normalization=kwargs.get("with_normalization", True)
        )

        color = color_cycle[i % len(color_cycle)]

        plt.plot(range(len(scores)), scores, linestyle='--', alpha=0.3,
                 color=color, label=f"{param_name}={val} (raw)")

        smoothed = np.convolve(scores, np.ones(10) / 10, mode='valid')
        plt.plot(range(len(smoothed)), smoothed, linestyle='-', color=color,
                 label=f"{param_name}={val} (smoothed)")
```

```
    plt.title(f"PPO: сравнение по параметру {param_name}")
    plt.xlabel("Итерация")
    plt.ylabel("Счёт")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f"{filename_prefix}_{param_name}.png")


def different_steps():
    steps = [1024, 2048, 4096]
    run_and_plot(steps, "steps", lambda st: {"steps": st}, "results")


def different_clip_ratio():
    clip_ratio = [0.1, 0.2, 0.3]
        run_and_plot(clip_ratio, "clip_ratio", lambda cr: {"clip_ratio": cr}, "results")


def different_normalization():
    with_normalization = [False, True]
      run_and_plot(with_normalization, "with_normalization", lambda wn: {"with_normalization": wn}, "results")


def different_n_epoch():
    n_epoch = [5, 10, 20]
```

```
        run_and_plot(n_epoch, "n_epoch", lambda ne: {"n_epoch": ne},
"results")


def main():
  different_steps()
  different_clip_ratio()
  different_normalization()
  different_n_epoch()


if __name__ == "__main__":
  main()
```

Исходный код agent.py

```
import numpy as np
import torch
import torch.optim as optim

from actor import Actor
from critic import Critic


class Agent:
  def __init__(
      self,
      env,
```

```
        gamma,
        gae_lambda,
        entropy_coefficient,
        clip_ratio,
        steps,
        iterations,
        n_epoch,
        batch_size,
        lr,
        with_normalization
    ):
        self.env = env
        self.n_observations = env.observation_space.shape[0]
        self.n_actions = env.action_space.shape[0]

        self.gamma = gamma
        self.gae_lambda = gae_lambda
        self.entropy_coefficient = entropy_coefficient
        self.clip_ratio = clip_ratio
        self.steps = steps
        self.iterations = iterations
        self.n_epoch = n_epoch
        self.batch_size = batch_size
        self.with_normalization = with_normalization

        self.actor = Actor(self.n_observations, self.n_actions)
        self.critic = Critic(self.n_observations)
        self.actor_optimizer = optim.Adam(self.actor.parameters(), lr=lr)
        self.critic_optimizer = optim.Adam(self.critic.parameters(), lr=lr)
```

```
        self.actor_loss_history = []
        self.critic_loss_history = []
        self.scores = []

    def get_action(self, state):
        state = torch.as_tensor(state, dtype=torch.float32)
        action, _ = self.actor.get_action(state)
        return action.item()

    def make_action(self, action):
        clipped_action = np.clip(action, -1, 1)
                    next_state, reward, terminated, truncated, _ = self.env.step(clipped_action)
        is_terminal = terminated or truncated
        return next_state, reward, is_terminal

    def train(self):
        for _ in range(self.iterations):
                states, actions, rewards, values, is_terminals, log_probs, episode_rewards = self.get_trajectories()
            returns, advantages = self.get_returns_and_advantages(rewards, values, is_terminals)
                self.update_net_weights(states, actions, log_probs, returns, advantages)

            if episode_rewards:
                avg_score = np.mean(episode_rewards)
                self.scores.append(avg_score)
```

```
        self.env.close()

    def update_net_weights(self, states, actions, old_log_probs, returns, advantages):
        self.actor.train()
        self.critic.train()

        actor_losses, critic_losses = [], []

        states = np.array(states)
        actions = np.array(actions)
        old_log_probs = np.array(old_log_probs)

        for _ in range(self.n_epoch):
            indices = np.random.permutation(len(states))
            for start in range(0, len(states), self.batch_size):
                end = start + self.batch_size
                batch_indices = indices[start:end]

                batch_states = torch.as_tensor(states[batch_indices], dtype=torch.float32)
                batch_actions = torch.as_tensor(actions[batch_indices], dtype=torch.float32)
                batch_old_log_probs = torch.as_tensor(old_log_probs[batch_indices], dtype=torch.float32)
                batch_returns = returns[batch_indices]
                batch_advantages = advantages[batch_indices]
```

```
        dist = self.actor.get_dist(batch_states)
        cur_log_probs = dist.log_prob(batch_actions).sum(dim=-1)
        ratio = torch.exp(cur_log_probs - batch_old_log_probs)
        entropy = dist.entropy().mean()
        batch_advantages = batch_advantages.detach()
        loss = batch_advantages * ratio
        clipped_loss = (
            torch.clamp(ratio, 1. - self.clip_ratio, 1. + self.clip_ratio)
            * batch_advantages
        )
        actor_loss = (
            -torch.mean(torch.min(loss, clipped_loss))
            - entropy * self.entropy_coefficient
        )
        cur_value = self.critic(batch_states)
        critic_loss = (batch_returns - cur_value).pow(2).mean()
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()
        actor_losses.append(actor_loss.item())
        critic_losses.append(critic_loss.item())

avg_actor_loss = sum(actor_losses) / len(actor_losses)
avg_critic_loss = sum(critic_losses) / len(critic_losses)
self.actor_loss_history.append(avg_actor_loss)
self.critic_loss_history.append(avg_critic_loss)
```

```
        def get_returns_and_advantages(self, rewards, values, is_terminals):
            gae = 0
            returns, advantages = [], []

            for i in reversed(range(len(rewards))):
                if is_terminals[i]:
                    next_value = 0.0
                delta = rewards[i] + self.gamma * (values[i + 1] if i < len(rewards)
- 1 else 0) - values[i]
                gae = delta + self.gamma * self.gae_lambda * gae
                returns.insert(0, (gae + values[i]).detach().clone().float())
                advantages.insert(0, gae.detach().clone().float())

            returns = torch.tensor(returns)
            advantages = torch.tensor(advantages)

            if self.with_normalization:
                advantages = (advantages - advantages.mean()) / (advantages.std()
+ 1e-8)

            return returns, advantages

        def get_trajectories(self):
                states, actions, rewards, values, is_terminals, episode_rewards,
log_probs = [], [], [], [], [], [], []
            current_reward = 0
            state, _ = self.env.reset()
```

```
        for _ in range(self.steps):
                                state_tensor = torch.as_tensor(np.array(state),
dtype=torch.float32).unsqueeze(0)

            with torch.no_grad():
                action_tensor, log_prob = self.actor.get_action(state_tensor)
                value = self.critic(state_tensor)

            action = np.array([action_tensor.item()])
            next_state, reward, is_terminal = self.make_action(action)

            states.append(state)
            actions.append(action)
            log_probs.append(log_prob.item())
            rewards.append(reward)
            is_terminals.append(is_terminal)
            values.append(value.squeeze())
            current_reward += reward
            state = next_state

            if is_terminal:
                episode_rewards.append(current_reward)
                current_reward = 0
                state, _ = self.env.reset()

              return states, actions, rewards, values, is_terminals, log_probs,
episode_rewards
```

Исходный код actor.py

```
import torch
import torch.nn as nn
from torch.distributions import Normal


class Actor(nn.Module):
  def __init__(self, n_observations, n_actions, hidden_size=256):
    super(Actor, self).__init__()
    self.log_std = nn.Parameter(torch.zeros(n_actions))
    self.model = nn.Sequential(
      nn.Linear(n_observations, hidden_size),
      nn.Tanh(),
      nn.Linear(hidden_size, hidden_size // 2),
      nn.Tanh(),
      nn.Linear(hidden_size // 2, n_actions)
    )

  def forward(self, x):
    return self.model(x)

  def get_dist(self, x):
    mean = self.forward(x)
    std = torch.exp(self.log_std).clamp(1e-6, 1)
    return Normal(mean, std)

  def get_action(self, x):
    x = torch.as_tensor(x, dtype=torch.float32)
```

```
        dist_x = self.get_dist(x)
        action = dist_x.rsample()
        log_prob = dist_x.log_prob(action).sum(-1)
        return action, log_prob
```

Исходный код critic.py

```
import torch.nn as nn


class Critic(nn.Module):
    def __init__(self, n_observations, hidden_size=128):
        super(Critic, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(n_observations, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, hidden_size // 2),
            nn.Tanh(),
            nn.Linear(hidden_size // 2, 1)
        )

    def forward(self, x):
        return self.model(x)
```