Московский государственный технический университет им. Н.Э. Баумана

Факультет «Информатика, искусственный интеллект и системы управления»

Кафедра «Системы обработки информации и управления»



**Отчет по лабораторной работе №7**

**по дисциплине «Методы машинного обучения»**

Алгоритмы Actor-Critic .

(тема работы)

ИСПОЛНИТЕЛЬ:

Пасатюк А.Д.

группа ИУ5-23М

ПРЕПОДАВАТЕЛЬ:

Гапанюк Ю.А.

Москва, 2023

**Цель работы**

Ознакомление с базовыми методами обучения с подкреплением на основе алгоритмов Actor-Critic.

**Задание**

Реализовать любой алгоритм семейства Actor-Critic для произвольной среды.

**Выполнение**

Реализуем алгоритм Actor-Critic для среды Toy Text / CliffWalking-v0.

Код программы:

import gym, os

from itertools import count

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

from torch.distributions import Categorical

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

env = gym.make("CartPole-v0").unwrapped

state\_size = env.observation\_space.shape[0]

action\_size = env.action\_space.n

lr = 0.001

class Actor(nn.Module):

    def \_\_init\_\_(self, state\_size, action\_size):

        super(Actor, self).\_\_init\_\_()

        self.state\_size = state\_size

        self.action\_size = action\_size

        self.linear1 = nn.Linear(self.state\_size, 128)

        self.linear2 = nn.Linear(128, 256)

        self.linear3 = nn.Linear(256, self.action\_size)

    def forward(self, state):

        output = F.relu(self.linear1(state))

        output = F.relu(self.linear2(output))

        output = self.linear3(output)

        distribution = Categorical(F.softmax(output, dim=-1))

        return distribution

class Critic(nn.Module):

    def \_\_init\_\_(self, state\_size, action\_size):

        super(Critic, self).\_\_init\_\_()

        self.state\_size = state\_size

        self.action\_size = action\_size

        self.linear1 = nn.Linear(self.state\_size, 128)

        self.linear2 = nn.Linear(128, 256)

        self.linear3 = nn.Linear(256, 1)

    def forward(self, state):

        output = F.relu(self.linear1(state))

        output = F.relu(self.linear2(output))

        value = self.linear3(output)

        return value

import matplotlib.pyplot as plt

episode\_durations = []

def plot\_durations():

    plt.figure(2)

    plt.clf()

    durations\_t = torch.FloatTensor(episode\_durations)

    plt.title('Training...')

    plt.xlabel('Episode')

    plt.ylabel('Duration')

    plt.plot(durations\_t.numpy())

    # Take 100 episode averages and plot them too

    if len(durations\_t) >= 100:

        means = durations\_t.unfold(0, 100, 1).mean(1).view(-1)

        means = torch.cat((torch.zeros(99), means))

        plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated

def compute\_returns(next\_value, rewards, masks, gamma=0.99):

    R = next\_value

    returns = []

    for step in reversed(range(len(rewards))):

        R = rewards[step] + gamma \* R \* masks[step]

        returns.insert(0, R)

    return returns

def trainIters(actor, critic, n\_iters):

    optimizerA = optim.Adam(actor.parameters())

    optimizerC = optim.Adam(critic.parameters())

    for iter in range(n\_iters):

        state = env.reset()

        log\_probs = []

        values = []

        rewards = []

        masks = []

        entropy = 0

        env.reset()

        for i in count():

            #env.render()

            state = torch.FloatTensor(state).to(device)

            dist, value = actor(state), critic(state)

            action = dist.sample()

            next\_state, reward, done, \_, \_ = env.step(action.cpu().numpy())

            log\_prob = dist.log\_prob(action).unsqueeze(0)

            entropy += dist.entropy().mean()

            log\_probs.append(log\_prob)

            values.append(value)

            rewards.append(torch.tensor([reward], dtype=torch.float, device=device))

            masks.append(torch.tensor([1-done], dtype=torch.float, device=device))

            state = next\_state

            if done:

                #print('Iteration: {}, Score: {}'.format(iter, i))

                episode\_durations.append(i + 1)

                plot\_durations()

                break

        next\_state = torch.FloatTensor(next\_state).to(device)

        next\_value = critic(next\_state)

        returns = compute\_returns(next\_value, rewards, masks)

        log\_probs = torch.cat(log\_probs)

        returns = torch.cat(returns).detach()

        values = torch.cat(values)

        advantage = returns - values

        actor\_loss = -(log\_probs \* advantage.detach()).mean()

        critic\_loss = advantage.pow(2).mean()

        optimizerA.zero\_grad()

        optimizerC.zero\_grad()

        actor\_loss.backward()

        critic\_loss.backward()

        optimizerA.step()

        optimizerC.step()

    #env.close()

actor = Actor(state\_size, action\_size).to(device)

critic = Critic(state\_size, action\_size).to(device)

trainIters(actor, critic, 300)

Результат работы программы:

