****Московский государственный технический университет им. Н.Э. Баумана  
Кафедра «Системы обработки информации и управления»

Лабораторная работа №7  
по дисциплине  
«Методы машинного обучения»  
на тему

## «Алгоритмы Actor-Critic.»

Выполнил:  
студент группы ИУ5И-23М  
ЧжанЦзюй

Москва — 2023 г.

**1. Цель лабораторной работы**

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

**2. Задание**

* Реализуйте любой алгоритм семейства Actor-Critic для произвольной среды.

1. **Текст программы**

Среда: CartPole-v0

from tqdm import tqdm  
import numpy as np  
import torch  
import collections  
import random  
  
class ReplayBuffer:  
 def \_\_init\_\_(self, capacity):  
 self.buffer = collections.deque(maxlen=capacity)   
  
 def add(self, state, action, reward, next\_state, done):   
 self.buffer.append((state, action, reward, next\_state, done))   
  
 def sample(self, batch\_size):   
 transitions = random.sample(self.buffer, batch\_size)  
 state, action, reward, next\_state, done = zip(\*transitions)  
 return np.array(state), action, reward, np.array(next\_state), done   
  
 def size(self):   
 return len(self.buffer)  
  
def moving\_average(a, window\_size):  
 cumulative\_sum = np.cumsum(np.insert(a, 0, 0))   
 middle = (cumulative\_sum[window\_size:] - cumulative\_sum[:-window\_size]) / window\_size  
 r = np.arange(1, window\_size-1, 2)  
 begin = np.cumsum(a[:window\_size-1])[::2] / r  
 end = (np.cumsum(a[:-window\_size:-1])[::2] / r)[::-1]  
 return np.concatenate((begin, middle, end))  
  
def train\_on\_policy\_agent(env, agent, num\_episodes):  
 return\_list = []  
 for i in range(10):  
 with tqdm(total=int(num\_episodes/10), desc='Iteration %d' % i) as pbar:  
 for i\_episode in range(int(num\_episodes/10)):  
 episode\_return = 0  
 transition\_dict = {'states': [], 'actions': [], 'next\_states': [], 'rewards': [], 'dones': []}  
 state = env.reset()  
 done = False  
 while not done:  
 action = agent.take\_action(state)  
 next\_state, reward, done, \_ = env.step(action)  
 transition\_dict['states'].append(state)  
 transition\_dict['actions'].append(action)  
 transition\_dict['next\_states'].append(next\_state)  
 transition\_dict['rewards'].append(reward)  
 transition\_dict['dones'].append(done)  
 state = next\_state  
 episode\_return += reward  
 return\_list.append(episode\_return)  
 agent.update(transition\_dict)  
 if (i\_episode+1) % 10 == 0:  
 pbar.set\_postfix({'episode': '%d' % (num\_episodes/10 \* i + i\_episode+1), 'return': '%.3f' % np.mean(return\_list[-10:])})  
 pbar.update(1)  
 return return\_list  
  
def train\_off\_policy\_agent(env, agent, num\_episodes, replay\_buffer, minimal\_size, batch\_size):  
 return\_list = []  
 for i in range(10):  
 with tqdm(total=int(num\_episodes/10), desc='Iteration %d' % i) as pbar:  
 for i\_episode in range(int(num\_episodes/10)):  
 episode\_return = 0  
 state = env.reset()  
 done = False  
 while not done:  
 action = agent.take\_action(state)  
 next\_state, reward, done, \_ = env.step(action)  
 replay\_buffer.add(state, action, reward, next\_state, done)  
 state = next\_state  
 episode\_return += reward  
 if replay\_buffer.size() > minimal\_size:  
 b\_s, b\_a, b\_r, b\_ns, b\_d = replay\_buffer.sample(batch\_size)  
 transition\_dict = {'states': b\_s, 'actions': b\_a, 'next\_states': b\_ns, 'rewards': b\_r, 'dones': b\_d}  
 agent.update(transition\_dict)  
 return\_list.append(episode\_return)  
 if (i\_episode+1) % 10 == 0:  
 pbar.set\_postfix({'episode': '%d' % (num\_episodes/10 \* i + i\_episode+1), 'return': '%.3f' % np.mean(return\_list[-10:])})  
 pbar.update(1)  
 return return\_list  
  
  
def compute\_advantage(gamma, lmbda, td\_delta):  
 td\_delta = td\_delta.detach().numpy()  
 advantage\_list = []  
 advantage = 0.0  
 for delta in td\_delta[::-1]:  
 advantage = gamma \* lmbda \* advantage + delta  
 advantage\_list.append(advantage)  
 advantage\_list.reverse()  
 return torch.tensor(advantage\_list, dtype=torch.float)

import gym  
import torch  
import torch.nn.functional as F  
import numpy as np  
import matplotlib.pyplot as plt  
import rl\_utils  
  
  
class PolicyNet(torch.nn.Module):  
 def \_\_init\_\_(self, state\_dim, hidden\_dim, action\_dim):  
 super(PolicyNet, self).\_\_init\_\_()  
 self.fc1 = torch.nn.Linear(state\_dim, hidden\_dim)  
 self.fc2 = torch.nn.Linear(hidden\_dim, action\_dim)  
  
 def forward(self, x):  
 x = F.relu(self.fc1(x))  
 return F.softmax(self.fc2(x),dim=1)  
class ValueNet(torch.nn.Module):  
 def \_\_init\_\_(self, state\_dim, hidden\_dim):  
 super(ValueNet, self).\_\_init\_\_()  
 self.fc1 = torch.nn.Linear(state\_dim, hidden\_dim)  
 self.fc2 = torch.nn.Linear(hidden\_dim, 1)  
  
 def forward(self, x):  
 x = F.relu(self.fc1(x))  
 return self.fc2(x)  
class ActorCritic:  
 def \_\_init\_\_(self, state\_dim, hidden\_dim, action\_dim, actor\_lr, critic\_lr, gamma, device):  
 self.actor = PolicyNet(state\_dim, hidden\_dim, action\_dim).to(device)  
 self.critic = ValueNet(state\_dim, hidden\_dim).to(device)   
 self.actor\_optimizer = torch.optim.Adam(self.actor.parameters(), lr=actor\_lr)  
 self.critic\_optimizer = torch.optim.Adam(self.critic.parameters(), lr=critic\_lr)   
 self.gamma = gamma  
  
 def take\_action(self, state):  
 state = torch.tensor([state], dtype=torch.float)  
 probs = self.actor(state)  
 action\_dist = torch.distributions.Categorical(probs)  
 action = action\_dist.sample()  
 return action.item()  
  
 def update(self, transition\_dict):  
 states = torch.tensor(transition\_dict['states'], dtype=torch.float)  
 actions = torch.tensor(transition\_dict['actions']).view(-1, 1)  
 rewards = torch.tensor(transition\_dict['rewards'], dtype=torch.float).view(-1, 1)  
 next\_states = torch.tensor(transition\_dict['next\_states'], dtype=torch.float)  
 dones = torch.tensor(transition\_dict['dones'], dtype=torch.float).view(-1, 1)  
  
 td\_target = rewards + self.gamma \* self.critic(next\_states) \* (1 - dones)   
 td\_delta = td\_target - self.critic(states)   
 log\_probs = torch.log(self.actor(states).gather(1, actions))  
 actor\_loss = torch.mean(-log\_probs \* td\_delta.detach())  
 critic\_loss = torch.mean(F.mse\_loss(self.critic(states), td\_target.detach()))   
 self.actor\_optimizer.zero\_grad()  
 self.critic\_optimizer.zero\_grad()  
 actor\_loss.backward()   
 critic\_loss.backward()   
 self.actor\_optimizer.step()   
 self.critic\_optimizer.step()   
actor\_lr = 1e-3  
critic\_lr = 1e-2  
num\_episodes = 1000  
hidden\_dim = 128  
gamma = 0.98  
device = torch.device("cuda") if torch.cuda.is\_available() else torch.device("cpu")  
  
env\_name = 'CartPole-v1'  
env = gym.make(env\_name)  
env.seed(0)  
torch.manual\_seed(0)  
state\_dim = env.observation\_space.shape[0]  
action\_dim = env.action\_space.n  
agent = ActorCritic(state\_dim, hidden\_dim, action\_dim, actor\_lr, critic\_lr, gamma, device)  
  
return\_list = rl\_utils.train\_on\_policy\_agent(env, agent, num\_episodes)  
  
episodes\_list = list(range(len(return\_list)))  
plt.plot(episodes\_list,return\_list)  
plt.xlabel('Episodes')  
plt.ylabel('Returns')  
plt.title('Actor-Critic on {}'.format(env\_name))  
plt.show()  
  
mv\_return = rl\_utils.moving\_average(return\_list, 9)  
plt.plot(episodes\_list, mv\_return)  
plt.xlabel('Episodes')  
plt.ylabel('Returns')  
plt.title('Actor-Critic on {}'.format(env\_name))  
plt.show()

1. **Результат**



