Московский государственный технический университет им. Н.Э. Баумана

Факультет «Информатика и системы управления»

Кафедра «Системы обработки информации и управления»



**«Методы машинного обучения в автоматизированных системах обработки информации и управления»**

**Лабораторная работа №7**

**«Алгоритмы Actor-Critic»**

**ИСПОЛНИТЕЛЬ:**

Демирев Н.К.

Группа ИУ5-21М

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

"\_\_"\_\_\_\_\_\_\_\_\_\_\_2023 г.

Москва 2023

## Задание

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

## Листинг

### policy.py

import torch.nn as nn

import torch.nn.functional as F

class Policy(nn.Module):

  def \_\_init\_\_(self):

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

    self.affine1 = nn.Linear(6, 128)

    # actor's layer

    self.action\_head = nn.Linear(128, 3)

    # critic's layer

    self.value\_head = nn.Linear(128, 1)

    # action & reward buffer

    self.saved\_actions = []

    self.rewards = []

  def forward(self, x):

    x = F.relu(self.affine1(x))

    # actor: choses action to take from state s\_t

    # by returning probability of each action

    action\_prob = F.softmax(self.action\_head(x), dim=-1)

    # critic: evaluates being in the state s\_t

    state\_values = self.value\_head(x)

    # return values for both actor and critic as a tuple of 2 values:

    # 1. a list with the probability of each action over the action space

    # 2. the value from state s\_t

    return action\_prob, state\_values

### main.py

import gymnasium as gym

import numpy as np

from itertools import count

from collections import namedtuple

import torch

import torch.nn.functional as F

import torch.optim as optim

from torch.distributions import Categorical

from policy import Policy

import os

import pygame

from tqdm import tqdm

os.environ['SDL\_VIDEODRIVER']='dummy'

pygame.display.set\_mode((640,480))

# Cart Pole

CONST\_ENV\_NAME = 'Acrobot-v1'

env = gym.make(CONST\_ENV\_NAME)

GAMMA = 0.99

SavedAction = namedtuple('SavedAction', ['log\_prob', 'value'])

model = Policy()

optimizer = optim.AdamW(model.parameters(), lr=1e-3)

eps = np.finfo(np.float32).eps.item()

def select\_action(state):

  state = torch.from\_numpy(state).float()

  probs, state\_value = model(state)

  # create a categorical distribution over the list of probabilities of actions

  m = Categorical(probs)

  # and sample an action using the distribution

  action = m.sample()

  # save to action buffer

  model.saved\_actions.append(SavedAction(m.log\_prob(action), state\_value))

  # the action to take (left or right)

  return action.item()

def finish\_episode():

  """

  Training code. Calculates actor and critic loss and performs backprop.

  """

  R = 0

  saved\_actions = model.saved\_actions

  policy\_losses = [] # list to save actor (policy) loss

  value\_losses = [] # list to save critic (value) loss

  returns = [] # list to save the true values

  # calculate the true value using rewards returned from the environment

  for r in model.rewards[::-1]:

    # calculate the discounted value

    R = r + GAMMA \* R

    returns.insert(0, R)

  returns = torch.tensor(returns)

  returns = (returns - returns.mean()) / (returns.std() + eps)

  for (log\_prob, value), R in zip(saved\_actions, returns):

    advantage = R - value.item()

    # calculate actor (policy) loss

    policy\_losses.append(-log\_prob \* advantage)

    # calculate critic (value) loss using L1 smooth loss

    value\_losses.append(F.smooth\_l1\_loss(value, torch.tensor([R])))

  # reset gradients

  optimizer.zero\_grad()

  # sum up all the values of policy\_losses and value\_losses

  loss = torch.stack(policy\_losses).sum() + torch.stack(value\_losses).sum()

  # perform backprop

  loss.backward()

  optimizer.step()

  # reset rewards and action buffer

  del model.rewards[:]

  del model.saved\_actions[:]

def main():

    running\_reward = -500

    # run infinitely many episodes

    for i\_episode in count(1):

        # print(running\_reward)

        # reset environment and episode reward

        state, \_ = env.reset()

        ep\_reward = 0

        # for each episode, only run 9999 steps so that we don't

        # infinite loop while learning

        for t in range(1, 9999):

            # select action from policy

            action = select\_action(state)

            # take the action

            state, reward, done, truncated, \_ = env.step(action)

            model.rewards.append(reward)

            ep\_reward += reward

            if done or truncated:

                break

        # print(ep\_reward)

        # update cumulative reward

        running\_reward = 0.05 \* ep\_reward + (1 - 0.05) \* running\_reward

        # perform backprop

        finish\_episode()

        # log results

        if i\_episode % 10 == 0:

            print(f"Episode {i\_episode}\tLast reward: {ep\_reward:.2f}\tAverage reward: {running\_reward:.2f}")

        # check if we have "solved" the cart pole problem

        if running\_reward > env.spec.reward\_threshold \* 2:

            print(f"Solved! Running reward is now {running\_reward} and the last episode runs to {t} time steps!")

            break

    env2 = gym.make(CONST\_ENV\_NAME, render\_mode='human')

    # reset environment and episode reward

    state, \_ = env2.reset()

    ep\_reward = 0

    # for each episode, only run 9999 steps so that we don't

    # infinite loop while learning

    bar = tqdm(range(1, 10000), bar\_format=' {l\_bar}{bar:20}{r\_bar}{bar:-10b}', colour='CYAN')

    for t in bar:

        # select action from policy

        action = select\_action(state)

        # take the action

        state, reward, done, \_, \_ = env2.step(action)

        model.rewards.append(reward)

        ep\_reward += reward

        if done:

            bar.update(10000-t)

            bar.refresh()

            bar.close()

            break

if \_\_name\_\_ == '\_\_main\_\_':

    main()

## Экранные формы

