|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

**ФАКУЛЬТЕТ** ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

**КАФЕДРА** СИСТЕМЫ ОБРАБОТКИ ИНФОРМАЦИИ И УПРАВЛЕНИЯ

**Отчёт к лабораторным работам по курсу**

**«Методы машинного обучения»**

**Лабораторная работа №7 «Алгоритмы Actor-Critic»**

**Выполнил:**

студент(ка) группы ИУ5И-21М Лю Бэйбэй

подпись, дата

**Проверил:**

к.т.н., доц., Виноградовой М.В.

подпись, дата

Москва, 2022 г.

# 1. описание задания

Реализуйте любой алгоритм семейства Actor-Critic для произвольной среды.

# 2. Текст программы и экранные формы с примерами выполнения программы.

Utils:

class Logger:

    def \_\_init\_\_(self):

        self.writer = SummaryWriter()

    def save\_metadata(self, params):

        with open(f'{self.writer.log\_dir}/params.txt', 'w') as f:

            print(params, file=f)

    def add\_scalar(self, location, val, step):

        self.writer.add\_scalar(location, val, step)

    def write(self, q1\_loss, q2\_loss, policy\_loss, entropy\_loss, total\_steps):

        self.writer.add\_scalar('Loss/q1\_loss', q1\_loss, total\_steps)

        self.writer.add\_scalar('Loss/q2\_loss', q2\_loss, total\_steps)

        self.writer.add\_scalar('Loss/alpha\_loss', entropy\_loss, total\_steps)

        if policy\_loss:

            self.writer.add\_scalar('Loss/policy\_loss', policy\_loss, total\_steps)

class SumTree:

    write = 0

    def \_\_init\_\_(self, capacity):

        self.capacity = capacity

        self.tree = np.zeros(2 \* capacity - 1)

        self.data = np.zeros(capacity, dtype=object)

        self.n\_entries = 0

        self.size = 0

        self.reached\_max\_write = False

    # update to the root node

    def \_propagate(self, idx, change):

        parent = (idx - 1) // 2

        self.tree[parent] += change

        if parent != 0:

            self.\_propagate(parent, change)

    # find sample on leaf node

    def \_retrieve(self, idx, s):

        left = 2 \* idx + 1

        right = left + 1

        if left >= len(self.tree):

            return idx

        if s <= self.tree[left]:

            return self.\_retrieve(left, s)

        else:

            return self.\_retrieve(right, s - self.tree[left])

    def total(self):

        return self.tree[0]

    def get\_max\_idx(self):

      if self.reached\_max\_write:

        return 2 \* self.capacity - 1

      return self.write + self.capacity - 1

    # store priority and sample

    def add(self, p, data):

        idx = self.write + self.capacity - 1

        self.data[self.write] = data

        self.update(idx, p)

        self.write += 1

        if self.write >= self.capacity:

            self.write = 0

            self.reached\_max\_write = True

            print("self.write >= self.capacity")

        if self.n\_entries < self.capacity:

            self.n\_entries += 1

        self.size += 1

    # update priority

    def update(self, idx, p):

        change = p - self.tree[idx]

        self.tree[idx] = p

        self.\_propagate(idx, change)

    # get priority and sample

    def get(self, s):

        idx = self.\_retrieve(0, s)

        dataIdx = idx - self.capacity + 1

        return (idx, self.tree[idx], self.data[dataIdx])

class TimeLimit(gym.Wrapper):

    def \_\_init\_\_(self, env, max\_episode\_steps=None):

        super().\_\_init\_\_(env)

        if max\_episode\_steps is None and self.env.spec is not None:

            max\_episode\_steps = env.spec.max\_episode\_steps

        if self.env.spec is not None:

            self.env.spec.max\_episode\_steps = max\_episode\_steps

        self.\_max\_episode\_steps = max\_episode\_steps

        self.\_elapsed\_steps = None

    # take a step in the envioronment. If reached to the maximal step number, return 'done'.

    def step(self, action):

        observation, reward, done, info = self.env.step(action)

        self.\_elapsed\_steps += 1

        if self.\_elapsed\_steps >= self.\_max\_episode\_steps:

            info["TimeLimit.truncated"] = not done

            done = True

        return observation, reward, done, info

    # reset envirinment and nullify number of taken steps.

    def reset(self, \*\*kwargs):

        self.\_elapsed\_steps = 0

        return self.env.reset(\*\*kwargs)

def initialize\_parameters(obs\_dim, action\_dim, q\_lr, policy\_lr):

    q\_net1 = QNet(obs\_dim, action\_dim).to(device)

    q\_net2 = QNet(obs\_dim, action\_dim).to(device)

    target\_q\_net1 = QNet(obs\_dim, action\_dim).to(device)

    target\_q\_net2 = QNet(obs\_dim, action\_dim).to(device)

    policy\_net = PolicyNet(obs\_dim, action\_dim).to(device)

    q1\_optimizer = optim.Adam(q\_net1.parameters(), lr=q\_lr)

    q2\_optimizer = optim.Adam(q\_net2.parameters(), lr=q\_lr)

    policy\_optimizer = optim.Adam(policy\_net.parameters(), lr=policy\_lr)

    models = (target\_q\_net1, target\_q\_net2, q\_net1, q\_net2, policy\_net)

    optimizers = (q1\_optimizer, q2\_optimizer, policy\_optimizer)

    return models, optimizers

def copy\_weights(target\_model, model, tau=1):

    for target\_param, param in zip(target\_model.parameters(), model.parameters()):

        target\_param.data.copy\_(tau \* param + (1 - tau) \* target\_param)

buffer:

import random

import numpy as np

from collections import deque

class BasicBuffer:

    def \_\_init\_\_(self, max\_size):

        self.max\_size = max\_size

        self.buffer = deque(maxlen=max\_size)

    # add sample to buffer

    def push(self, state, action, reward, next\_state, done, agent=None):

        experience = (state, action, np.array([reward]), next\_state, done)

        self.buffer.append(experience)

    def sample\_to\_tensor(self, state, action, reward, next\_state, done):

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

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

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

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

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

        return (state, action, reward, next\_state, done)

    # sample from environment

    def sample(self, batch\_size):

        batch = random.sample(self.buffer, batch\_size)

        state\_batch, action\_batch, reward\_batch, next\_state\_batch, done\_batch = zip(\*batch)

        states, actions, rewards, next\_states, dones = self.sample\_to\_tensor(np.array(state\_batch),

                                                                             np.array(action\_batch),

                                                                             np.array(reward\_batch),

                                                                             np.array(next\_state\_batch),

                                                                             np.array(done\_batch))

        dones = dones.view(dones.size(0), -1)

        return (states, actions, rewards, next\_states, dones), None, 1

    def update(self, samples, idxs, agent):

        pass

    def \_\_len\_\_(self):

        return len(self.buffer)

class PrioritizedBuffer(BasicBuffer):  # stored as ( s, a, r, s\_ ) in SumTree

    def \_\_init\_\_(self, max\_size):

        self.tree = SumTree(max\_size)

        self.max\_size = max\_size

        self.alpha = 0.6

        self.beta = 0.4

        self.beta\_increment\_per\_sampling = 0.001

    # return the priority of a samlpe based on its error

    def \_get\_priority(self, error):

        return (np.abs(error) + EPSILON) \*\* self.alpha

    # add sample to buffer

    def push(self, state, action, reward, next\_state, done, agent):

        sample = (state, action, np.array([reward]), next\_state, done)

        T\_sample = self.sample\_to\_tensor(state,

                                         action,

                                         np.array([reward]),

                                         next\_state,

                                         np.array(done))

        T\_sample = tuple(T.unsqueeze(0) for T in T\_sample[:-1]) + (T\_sample[-1],)

        error = agent.get\_priority\_error(T\_sample)

        p = self.\_get\_priority(error)

        self.tree.add(p, sample)

    # sample from environment

    def sample(self, n):

        batch = []

        idxs = []

        segment = self.tree.total() / n

        priorities = []

        self.beta = np.min([1., self.beta + self.beta\_increment\_per\_sampling])

        for i in range(n):

            a = segment \* i

            b = segment \* (i + 1)

            assert b <= self.tree.total()

            idx = self.tree.get\_max\_idx() + 1

            while idx > self.tree.get\_max\_idx():

              s = np.random.uniform(a, b)

              (idx, p, data) = self.tree.get(s)

            priorities.append(p)

            batch.append(data)

            idxs.append(idx)

        sampling\_probabilities = priorities / self.tree.total()

        is\_weight = np.power(self.tree.n\_entries \* sampling\_probabilities, -self.beta)

        is\_weight /= is\_weight.max()

        try:

            state\_batch, action\_batch, reward\_batch, next\_state\_batch, done\_batch = zip(\*batch)

        except TypeError:

            raise TypeError

        states, actions, rewards, next\_states, dones = self.sample\_to\_tensor(np.array(state\_batch),

                                                                      np.array(action\_batch),

                                                                      np.array(reward\_batch),

                                                                      np.array(next\_state\_batch),

                                                                      np.array(done\_batch))

        dones = dones.view(dones.size(0), -1)

        is\_weight = torch.Tensor(is\_weight).to(device)

        batch = (states, actions, rewards, next\_states, dones)

        return batch, idxs, is\_weight

    # update priority of given samples

    def update(self, samples, idxs, agent):

        errors = agent.get\_priority\_error(samples)

        for idx, error in zip(idxs, errors):

            self.\_update(idx, error)

    # update a priority of a given sample

    def \_update(self, idx, error):

        p = self.\_get\_priority(error)

        self.tree.update(idx, p)

    def \_\_len\_\_(self):

        return self.tree.n\_entries

models:

class QNet(nn.Module):

    def \_\_init\_\_(self, num\_inputs, num\_actions, hidden\_size=256, init\_w=3e-3):

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

        self.linear1 = nn.Linear(num\_inputs + num\_actions, hidden\_size)

        self.linear2 = nn.Linear(hidden\_size, hidden\_size)

        self.linear3 = nn.Linear(hidden\_size, 1)

    def forward(self, state, action):

        x = torch.cat([state, action], 1)

        x = F.relu(self.linear1(x))

        x = F.relu(self.linear2(x))

        x = self.linear3(x)

        return x

class PolicyNet(nn.Module):

    def \_\_init\_\_(self, num\_inputs, num\_actions, hidden\_size=256, init\_w=3e-3, log\_std\_min=-20, log\_std\_max=2):

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

        self.log\_std\_min = log\_std\_min

        self.log\_std\_max = log\_std\_max

        self.linear1 = nn.Linear(num\_inputs, hidden\_size)

        self.linear2 = nn.Linear(hidden\_size, hidden\_size)

        self.mean\_linear = nn.Linear(hidden\_size, num\_actions)

        self.log\_std\_linear = nn.Linear(hidden\_size, num\_actions)

    def forward(self, state):

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

        x = F.relu(self.linear2(x))

        mean    = self.mean\_linear(x)

        log\_std = self.log\_std\_linear(x)

        log\_std = torch.clamp(log\_std, self.log\_std\_min, self.log\_std\_max)

        return mean, log\_std

    def sample(self, state, reparameterize=True, epsilon=1e-6):

        mean, log\_std = self.forward(state)

        std = log\_std.exp()

        normal = Normal(mean, std)

        if reparameterize:

            z = normal.rsample()

        else:

            z = normal.sample()

        action = torch.tanh(z)

        log\_pi = normal.log\_prob(z) - torch.log(1 - action.pow(2) + epsilon)

        log\_pi = log\_pi.sum(1, keepdim=True)

        return action, log\_pi

class BetaPolicyNet(nn.Module):

    def \_\_init\_\_(self, num\_inputs, num\_actions, hidden\_size=256, init\_w=3e-3):

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

        self.linear1 = nn.Linear(num\_inputs, hidden\_size)

        self.linear2 = nn.Linear(hidden\_size, hidden\_size)

        self.alpha = nn.Linear(hidden\_size, num\_actions)

        self.beta = nn.Linear(hidden\_size, num\_actions)

        self.softplus = nn.Softplus()

    def forward(self, state):

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

        x = F.relu(self.linear2(x))

        alpha = 1 + self.softplus(self.alpha(x))

        beta = 1 + self.softplus(self.beta(x))

        return alpha, beta

    def sample(self, state, reparameterize=True, epsilon=1e-6):

        alpha, beta = self.forward(state)

        beta\_dist = Beta(alpha, beta)

        if reparameterize:

            z = beta\_dist.rsample()

        else:

            z = beta\_dist.sample()

        action = z \* 2 - 1

        log\_pi = beta\_dist.log\_prob(z)

        log\_pi = log\_pi.sum(1, keepdim=True)

        return action, log\_pi

actor critic:

class SACAgent:

    def \_\_init\_\_(self, env, gamma, tau, alpha, q\_lr, policy\_lr, a\_lr, buffer\_maxlen):

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

        self.env = env

        self.obs\_dim = env.observation\_space.shape[0]  # 3

        self.action\_dim = env.action\_space.shape[0]  # 1

        self.gamma = gamma

        self.tau = tau

        self.update\_step = 0

        self.delay\_step = 2

        # initialize networks

        self.q\_net1 = QNet(self.obs\_dim, self.action\_dim).to(self.device)

        self.q\_net2 = QNet(self.obs\_dim, self.action\_dim).to(self.device)

        self.target\_q\_net1 = QNet(self.obs\_dim, self.action\_dim).to(self.device)

        self.target\_q\_net2 = QNet(self.obs\_dim, self.action\_dim).to(self.device)

        # self.policy\_net = PolicyNet(self.obs\_dim, self.action\_dim).to(self.device)

        self.policy\_net = BetaPolicyNet(self.obs\_dim, self.action\_dim).to(self.device)

        # copy params to target param

        self.target\_q\_net1.load\_state\_dict(self.q\_net1.state\_dict())

        self.target\_q\_net2.load\_state\_dict(self.q\_net2.state\_dict())

        # initialize optimizers

        self.q1\_optimizer = optim.Adam(self.q\_net1.parameters(), lr=q\_lr)

        self.q2\_optimizer = optim.Adam(self.q\_net2.parameters(), lr=q\_lr)

        self.policy\_optimizer = optim.Adam(self.policy\_net.parameters(), lr=policy\_lr)

        # entropy temperature

        self.alpha = alpha

        self.target\_entropy = -torch.prod(torch.Tensor(self.env.action\_space.shape).to(self.device)).item()

        self.log\_alpha = torch.zeros(1, requires\_grad=True, device=self.device)

        self.alpha\_optim = optim.Adam([self.log\_alpha], lr=a\_lr)

    def sample\_action(self, state):

        state = torch.FloatTensor(state).unsqueeze(0).to(device)

        with torch.no\_grad():

            action, \_ = self.policy\_net.sample(state, reparameterize=False)

        action = action.cpu().detach().squeeze(0).numpy()

        # for pendulum

        # action \*= 2

        return action

    # calculate the priority of a given sample

    def get\_priority\_error(self, sarsd):

        state, action, reward, next\_state, done = sarsd

        next\_action, \_ = self.policy\_net.sample(next\_state)

        with torch.no\_grad():

            q1 = self.q\_net1(state, action)

            q2 = self.q\_net2(state, action)

            next\_q1 = self.target\_q\_net1(next\_state, next\_action)

            next\_q2 = self.target\_q\_net2(next\_state, next\_action)

        next\_q\_target = torch.min(next\_q1, next\_q2)

        q\_target = abs(q1 + q2)/2.0 + EPSILON

        error = reward + (1 - done) \* self.gamma \* next\_q\_target - q\_target

        return error.cpu().detach().squeeze(0).numpy()

    def gradient\_step(self, q\_net, optimizer, states, actions, expected\_q, is\_weights):

        curr\_q = q\_net.forward(states, actions)

        q\_loss = (is\_weights \* F.mse\_loss(curr\_q, expected\_q)).mean()

        # update q networks

        optimizer.zero\_grad()

        q\_loss.backward()

        optimizer.step()

        return q\_loss

    def update\_q\_parametes(self, sarsd, is\_weights):

        states, actions, rewards, next\_states, dones = sarsd

        next\_actions, next\_log\_pi = self.policy\_net.sample(next\_states)

        with torch.no\_grad():

            next\_q1 = self.target\_q\_net1(next\_states, next\_actions)

            next\_q2 = self.target\_q\_net2(next\_states, next\_actions)

        next\_q\_target = torch.min(next\_q1, next\_q2) - self.alpha \* next\_log\_pi

        expected\_q = rewards + (1 - dones) \* self.gamma \* next\_q\_target

        expected\_q = expected\_q.detach()

        q1\_loss = self.gradient\_step(self.q\_net1, self.q1\_optimizer, states, actions, expected\_q, is\_weights)

        q2\_loss = self.gradient\_step(self.q\_net2, self.q2\_optimizer, states, actions, expected\_q, is\_weights)

        return q1\_loss, q2\_loss

    def update\_policy\_weights(self, sarsd):

        states, actions, rewards, next\_states, dones = sarsd

        new\_actions, self.log\_pi = self.policy\_net.sample(states)

        policy\_loss = None

        min\_q = torch.min(

            self.q\_net1.forward(states, new\_actions),

            self.q\_net2.forward(states, new\_actions)

        )

        policy\_loss = (self.alpha \* self.log\_pi - min\_q).mean()

        self.policy\_optimizer.zero\_grad()

        policy\_loss.backward()

        self.policy\_optimizer.step()

        copy\_weights(self.target\_q\_net1, self.q\_net1, tau)

        copy\_weights(self.target\_q\_net2, self.q\_net2, tau)

        return policy\_loss

    def adjust\_temperature(self):

        entropy\_loss = (self.log\_alpha \* (-self.log\_pi - self.target\_entropy).detach()).mean()

        self.alpha\_optim.zero\_grad()

        entropy\_loss.backward()

        self.alpha\_optim.step()

        self.alpha = self.log\_alpha.exp()

        return entropy\_loss

Algorithm

gamma = 0.99

tau = 0.01

alpha = 0.2

a\_lr = 3e-4

q\_lr = 3e-4

# p\_lr = 3e-4 # gauss policy

p\_lr = 1e-3 # beta policy

buffer\_maxlen = 1000000

max\_steps = 500

max\_episodes = 50

batch\_size = 64

# env = TimeLimit(gym.make("Pendulum-v0"))

env = TimeLimit(gym.make("LunarLanderContinuous-v2"))

max\_ep\_len = env.\_max\_episode\_steps

logger = Logger()

state = env.reset()

def Algorithm():

    agent = SACAgent(env, gamma, tau, alpha, q\_lr, p\_lr, a\_lr, buffer\_maxlen)

    buffer = PrioritizedBuffer(buffer\_maxlen)

    total\_steps = 0

    for episode in range(max\_episodes):

        state = env.reset()

        episode\_reward = 0

        for step in range(max\_steps):

            action = agent.sample\_action(state)

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

            buffer.push(state, action, reward, next\_state, done, agent)

            episode\_reward += reward

            if len(buffer) > batch\_size:

                sarsd, idxs, is\_weights = buffer.sample(batch\_size)

                buffer.update(sarsd, idxs, agent)

                q1\_loss, q2\_loss = agent.update\_q\_parametes(sarsd, is\_weights)

                if step % 2 == 0:

                    policy\_loss = agent.update\_policy\_weights(sarsd)

                entropy\_loss = agent.adjust\_temperature()

                logger.write(q1\_loss, q2\_loss, policy\_loss, entropy\_loss, total\_steps)

                total\_steps += 1

            if done or step == max\_steps-1:

                print("Episode " + str(episode) + ": " + str(episode\_reward))

                logger.add\_scalar('Reward/Reward', episode\_reward, episode)

                break

            state = next\_state

    return agent

Agent = Algorithm()





