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):
```

```python
        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),
                                                                n
p.array(action_batch),
                                                                n
p.array(reward_batch),
                                                                n
p.array(next_state_batch),
                                                                n
p.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.devi
ce)).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):
```

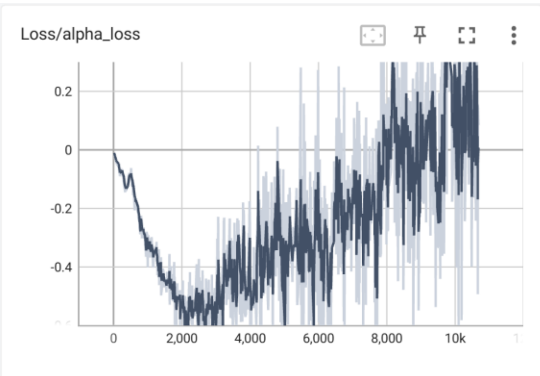
```
        curr_q = q_net.forward(states, actions)
        q_loss = (is_weights * F.mse_loss(curr_q,
expected_q)).mean()
        # update q networks
        optimizer.zero_grad()
        q_loss.backward()
        optimizer.step()
        return q_loss

    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()
```

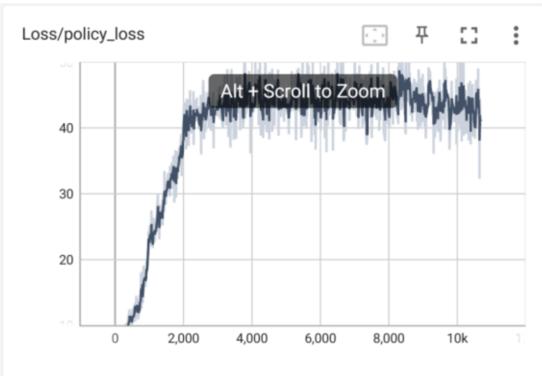
```
        copy_weights(self.target_q_net1, self.q_net1, tau)
        copy_weights(self.target_q_net2, self.q_net2, tau)

        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
```

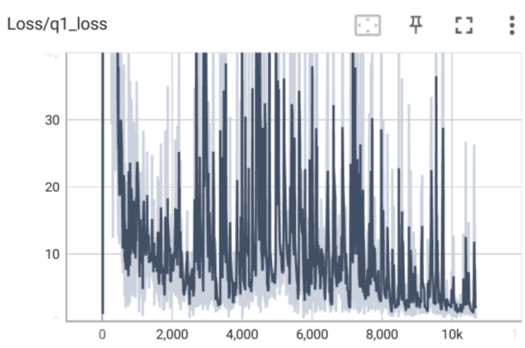
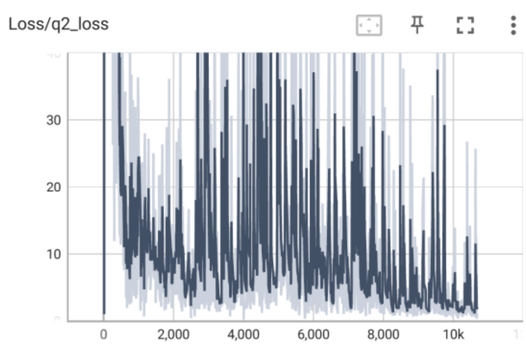
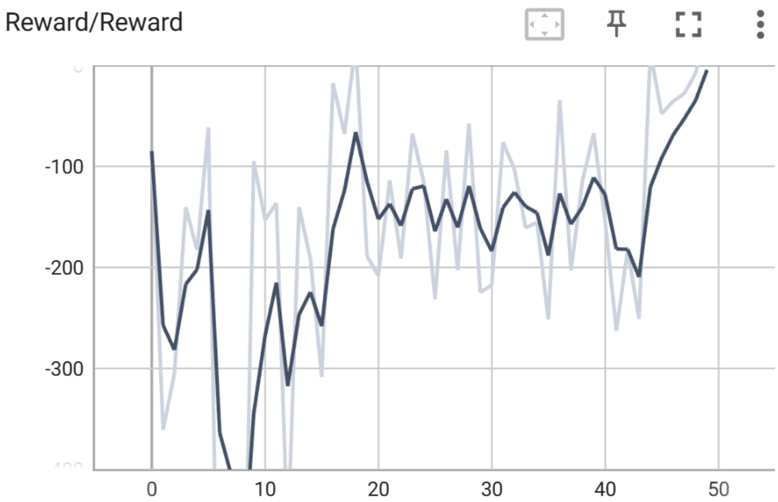
Algorithm

```
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()
```






Loss/q1_loss
30
20
10
0
2,000
4,000
6,000
8,000
10k

Loss/q2_loss
30
20
10
0
2,000
4,000
6,000
8,000
10k

Reward/Reward
-100
-200
-300
0
10
20
30
40
50