Reinforcement Learning Assignment 5

Chunlin Chen, 519021910463

May 11, 2023

1 Problem Description

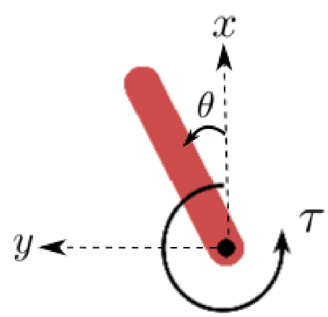

Figure 1: Pendulum

The inverted pendulum swingup problem is based on the classic problem in control theory. The system consists of a pendulum attached at one end to a fixed point, and the other end being free. The pendulum starts in a random position and the goal is to apply torque on the free end to swing it into an upright position, with its center of gravity right above the fixed point.

Figure 1 specifies the coordinate system used for the implementation of the pendulum's dynamic equations. $x - y$ are the Cartesian coordinates of the pendulum's end in meters, θ is the angle in radians, and τ is the torque in $N * m$ and is defined as positive counter-clockwise.

The action is a ndarray with shape (1,) representing the torque τ applied to free end of the pendulum. $\tau \in (-2.0, 2.0)$.

The observation is a ndarray with shape (3,) representing the x-y coordinates of the pendulum's free end and its angular velocity. $x = cos\theta \in [-1.0, 1.0], y = sin\theta \in [-1.0, 1.0], \omega \in [-8.0, 8.0]$.

The reward function is defined as:

$$r = -(\theta^2 + 0.1 * \omega^2 + 0.001 * \tau^2)$$

where θ is the pendulum's angle normalized between [-pi, pi] (with 0 being in the upright position). Based on the above equation, the minimum reward that can be obtained is $-(\pi^2 + 0.1 * 8^2 + 0.001 * 2^2) = -16.2736044$, while the maximum reward is zero (pendulum is upright with zero velocity and no torque applied).

The starting state is a random angle in $[-\pi, \pi]$ and a random angular velocity in [-1,1]. The episode truncates at 200 time steps.

2 Implementation of the A3C Algorithm

The Asynchronous Advantage Actor-Critic (A3C) algorithm is illustrated in Figure 2.

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 2: The A3C Algorithm

2.1 Network

```
class Net(nn.Module):
    def __init__(self, s_dim, a_dim):
        super(Net, self).__init__()
        self.s_dim = s_dim
        self.a_dim = a_dim
        self.a1 = nn.Linear(s_dim, 200)
        self.mu = nn.Linear(200, a_dim)
        self.sigma = nn.Linear(200, a_dim)
        self.c1 = nn.Linear(s_dim, 100)
        self.v = nn.Linear(100, 1)
```

```
        set_init([self.a1, self.mu, self.sigma, self.c1, self.v])
        self.distribution = torch.distributions.Normal

    def forward(self, x):
        a1 = F.relu6(self.a1(x))
        mu = 2 * F.tanh(self.mu(a1))
        sigma = F.softplus(self.sigma(a1)) + 0.001 # avoid 0
        c1 = F.relu6(self.c1(x))
        values = self.v(c1)
        return mu, sigma, values

    def choose_action(self, s):
        self.training = False
        mu, sigma, _ = self.forward(s)
        m = self.distribution(mu.view(1, ).data, sigma.view(1, ).data)
        return m.sample().numpy()

    def loss_func(self, s, a, v_t):
        self.train()
        mu, sigma, values = self.forward(s)
        td = v_t - values
        c_loss = td.pow(2)

        m = self.distribution(mu, sigma)
        log_prob = m.log_prob(a)
        entropy = 0.5 + 0.5 * math.log(2 * math.pi) + torch.log(m.scale)
            # exploration
        exp_v = log_prob * td.detach() + 0.005 * entropy
        a_loss = -exp_v
        total_loss = (a_loss + c_loss).mean()
        return total_loss
```

We first implement the actor-critic network in a Net class. The actor network consists of two 2-layer FCNs, which output μ and σ of a normal distribution of action. The critic network consists of a 2-layer FCN, which outputs the value of a given state.

2.2 Optimizer

```
class SharedAdam(torch.optim.Adam):
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.99), eps=1e-8,
            weight_decay=0):
        super(SharedAdam, self).__init__(params, lr=lr, betas=betas,
            eps=eps, weight_decay=weight_decay)
        # State initialization
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
```

```
            state['step'] = 0
            state['exp_avg'] = torch.zeros_like(p.data)
            state['exp_avg_sq'] = torch.zeros_like(p.data)

            # share in memory
            state['exp_avg'].share_memory_()
            state['exp_avg_sq'].share_memory_()
```

Here we define a SharedAdam class for the optimization, which can share the parameters in the memory.

2.3 Trainer

```
class Worker(mp.Process):
    def __init__(self, gnet, opt, global_ep, global_ep_r, res_queue,
         name):
        super(Worker, self).__init__()
        self.name = 'w%i' % name
        self.g_ep, self.g_ep_r, self.res_queue = global_ep, global_ep_r,
             res_queue
        self.gnet, self.opt = gnet, opt
        self.lnet = Net(N_S, N_A)      # local network
        self.env = gym.make('Pendulum-v1').unwrapped

    def run(self):
        total_step = 1
        while self.g_ep.value < MAX_EP:
            s = self.env.reset()[0]
            buffer_s, buffer_a, buffer_r = [], [], []
            ep_r = 0.
            for t in range(MAX_EP_STEP):
                if self.name == 'w0':
                    self.env.render()
                a = self.lnet.choose_action(v_wrap(s[None, :]))
                s_, r, done, _, _ = self.env.step(a.clip(-2, 2))
                if t == MAX_EP_STEP - 1:
                    done = True
                ep_r += r
                buffer_a.append(a)
                buffer_s.append(s)
                buffer_r.append((r+8.1)/8.1) # normalize

                if total_step % UPDATE_GLOBAL_ITER == 0 or done: # update
                     global and assign to local net
                    # sync
                    push_and_pull(self.opt, self.lnet, self.gnet, done,
                         s_, buffer_s, buffer_a, buffer_r, GAMMA)
                    buffer_s, buffer_a, buffer_r = [], [], []
```

```
                if done: # done and print information
                    record(self.g_ep, self.g_ep_r, ep_r,
                        self.res_queue, self.name)
                    break
            s = s_
            total_step += 1

        self.res_queue.put(None)
```

The training process is defined in a class Worker.

2.4 Parallel Training

```
gnet = Net(N_S, N_A)     # global network
gnet.share_memory()      # share the global parameters in multiprocessing
opt = SharedAdam(gnet.parameters(), lr=1e-4, betas=(0.95, 0.999)) #
    global optimizer
global_ep, global_ep_r, res_queue = mp.Value('i', 0), mp.Value('d', 0.),
    mp.Queue()

# parallel training
workers = [Worker(gnet, opt, global_ep, global_ep_r, res_queue, i) for i
    in range(mp.cpu_count())]
[w.start() for w in workers]
rewards = []                # record episode reward to plot
while True:
    r = res_queue.get()
    if r is not None:
        rewards.append(r)
    else:
        break
[w.join() for w in workers]
```

The parallel training process is shown above.

3 Implementation of the DDPG Algorithm

The Deep Deterministic Policy Gradient (DDPG) algorithm is illustrated in Figure 3.

3.1 Network

```
class Critic(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size,
        init_w=3e-3):
```

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 3: The DDPG Algorithm

```
        super(Critic, 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)

        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)

    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 Actor(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size,
         init_w=3e-3):
        super(Actor, self).__init__()

        self.linear1 = nn.Linear(num_inputs, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
```

```
        self.linear3 = nn.Linear(hidden_size, num_actions)

        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)

    def forward(self, state):
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))
        x = F.tanh(self.linear3(x))
        return x

    def get_action(self, state):
        state = torch.FloatTensor(state).unsqueeze(0)
        action = self.forward(state)
        return action.detach().cpu().numpy()[0, 0]
```

The actor network and the critic network both consist of 3-layer FCNs.

3.2 Exploration Noise

Here we use an exploration noise called OUNoise, which is implemented as below:

```
class OUNoise(object):
    def __init__(self, action_space, mu=0.0, theta=0.15, max_sigma=0.3,
         min_sigma=0.3, decay_period=100000):
        self.mu           = mu
        self.theta        = theta
        self.sigma        = max_sigma
        self.max_sigma    = max_sigma
        self.min_sigma    = min_sigma
        self.decay_period = decay_period
        self.action_dim   = action_space.shape[0]
        self.low          = action_space.low
        self.high         = action_space.high
        self.reset()

    def reset(self):
        self.state = np.ones(self.action_dim) * self.mu

    def evolve_state(self):
        x  = self.state
        dx = self.theta * (self.mu - x) + self.sigma *
            np.random.randn(self.action_dim)
        self.state = x + dx
        return self.state

    def get_action(self, action, t=0):
        ou_state = self.evolve_state()
```

```
        self.sigma = self.max_sigma - (self.max_sigma - self.min_sigma) *
            min(1.0, t / self.decay_period)
        return np.clip(action + ou_state, self.low, self.high)
```

3.3 DDPG Update

```
def ddpg_update(batch_size,
           gamma = 0.99,
           min_value=-np.inf,
           max_value=np.inf,
           soft_tau=1e-2):

    state, action, reward, next_state, done =
        replay_buffer.sample(batch_size)

    state      = torch.FloatTensor(state)
    next_state = torch.FloatTensor(next_state)
    action     = torch.FloatTensor(action)
    reward     = torch.FloatTensor(reward).unsqueeze(1)
    done       = torch.FloatTensor(np.float32(done)).unsqueeze(1)

    policy_loss = value_net(state, policy_net(state))
    policy_loss = -policy_loss.mean()

    next_action  = target_policy_net(next_state)
    target_value = target_value_net(next_state, next_action.detach())
    expected_value = reward + (1.0 - done) * gamma * target_value
    expected_value = torch.clamp(expected_value, min_value, max_value)

    value = value_net(state, action)
    value_loss = value_criterion(value, expected_value.detach())


    policy_optimizer.zero_grad()
    policy_loss.backward()
    policy_optimizer.step()

    value_optimizer.zero_grad()
    value_loss.backward()
    value_optimizer.step()

    for target_param, param in zip(target_value_net.parameters(),
        value_net.parameters()):
            target_param.data.copy_(
                target_param.data * (1.0 - soft_tau) + param.data *
                    soft_tau
            )
```

```
    for target_param, param in zip(target_policy_net.parameters(),
        policy_net.parameters()):
            target_param.data.copy_(
                target_param.data * (1.0 - soft_tau) + param.data *
                    soft_tau
            )
```

The update process of DDPG is defined in the above funtion.

3.4 Training

```
env = NormalizedActions(gym.make("Pendulum-v1"))
ou_noise = OUNoise(env.action_space)

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
hidden_dim = 256

value_net = Critic(state_dim, action_dim, hidden_dim)
policy_net = Actor(state_dim, action_dim, hidden_dim)

target_value_net = Critic(state_dim, action_dim, hidden_dim)
target_policy_net = Actor(state_dim, action_dim, hidden_dim)

for target_param, param in zip(target_value_net.parameters(),
    value_net.parameters()):
    target_param.data.copy_(param.data)

for target_param, param in zip(target_policy_net.parameters(),
    policy_net.parameters()):
    target_param.data.copy_(param.data)


value_lr = 1e-3
policy_lr = 1e-4

value_optimizer = optim.Adam(value_net.parameters(), lr=value_lr)
policy_optimizer = optim.Adam(policy_net.parameters(), lr=policy_lr)

value_criterion = nn.MSELoss()

replay_buffer_size = 1000000
replay_buffer = ReplayBuffer(replay_buffer_size)

max_steps  = 200
rewards    = []
batch_size = 128
max_episodes = 100
```

```
episode = 0

while episode < max_episodes:
    state = env.reset()[0]
    ou_noise.reset()
    episode_reward = 0

    for step in range(max_steps):
        action = policy_net.get_action(state)
        action = ou_noise.get_action(action, step)
        next_state, reward, done, _, _ = env.step(action)

        replay_buffer.push(state, action, reward, next_state, done)
        if len(replay_buffer) > batch_size:
            ddpg_update(batch_size)

        state = next_state
        episode_reward += reward

        if done:
            break

    episode += 1
    rewards.append(episode_reward)
```

The training process is shown above.

4 Results Analysis

4.1 A3C

We set the discount factor $\gamma = 0.9$ and the learning rate to be 0.001. The global network updates every 5 steps. And the result is shown in Figure 4.

We can find that the rewards converge to about -300 after around 3000 episodes.

4.2 DDPG

We set the discount factor $\gamma = 0.99$ and the learning rate for the actor network and the critic network to be 0.0001 and 0.001 respectively. The global network updates every 1 step. The size of the replay buffer is 1000000. And the result is shown in Figure 5.

We can find that the rewards converge to about -400 after around 20 episodes.

4.3 Conclusion

The DDPG can converge faster than the A3C.

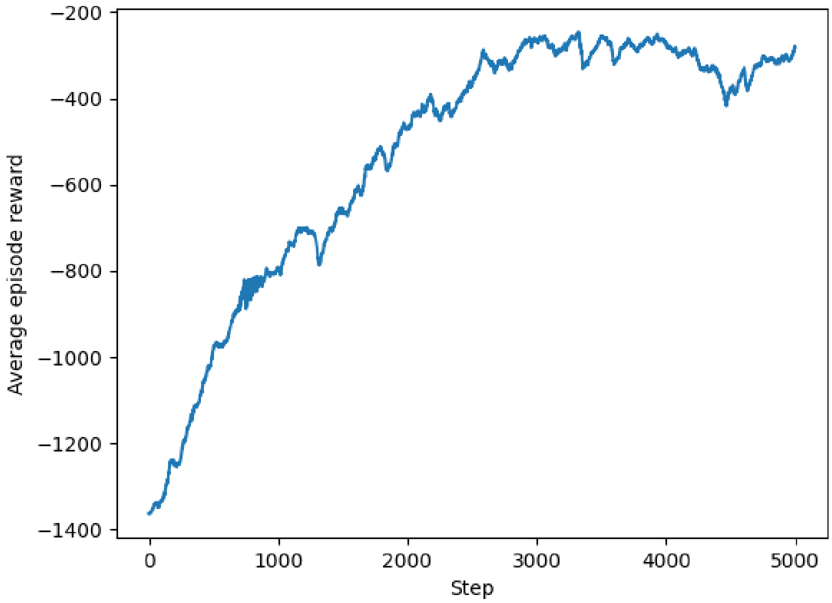

Figure 4: The A3C Result

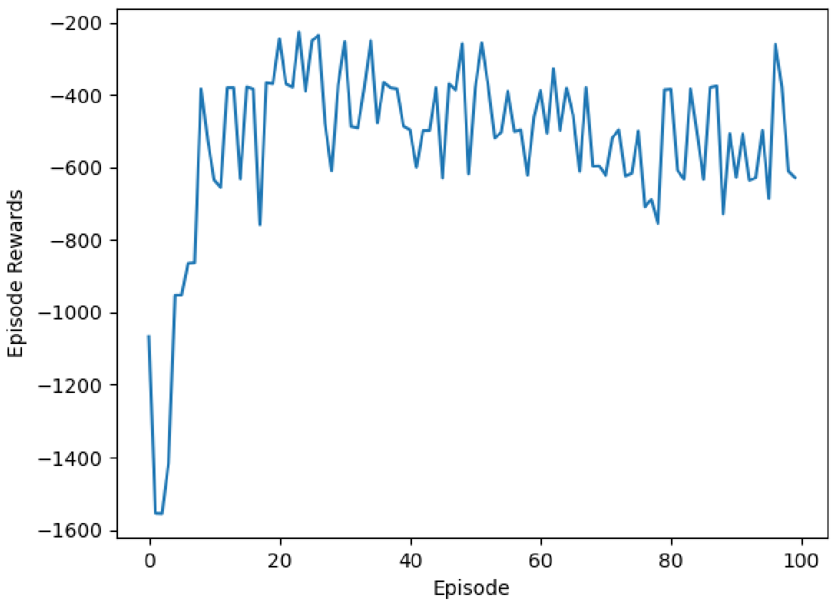

Figure 5: The DDPG Result