DRL_HW3

107081028 常安彥

1. Parking
   a. Implementation (DDPG + HER)

Actor

```
class Actor(nn.Module):
    def __init__(self, input_dims, n_actions, learning_rate):
        super(Actor, self).__init__()
        self.input = input_dims
        self.fc1 = nn.Linear(2 * input_dims, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, n_actions)
        self.optimizer = optim.RMSprop(self.parameters(), lr=learning_rate)
        self.loss = nn.MSELoss()

        self.device_type = 'cuda:0' if torch.cuda.is_available() else 'cpu'
        self.device = torch.device(self.device_type)
        self.to(self.device)

    def forward(self, data):
        fc_layer1 = F.relu(self.fc1(data))
        fc_layer2 = F.relu(self.fc2(fc_layer1))
        actions = F.tanh(self.fc3(fc_layer2))
        return actions
```

Two fully connected layer with ReLU function and one fully connected layer with Tanh function in the end.

First, the **input dimension** is (input_dim * 2) where input_dim = 6, because the input is a concatenation of observation state and the goal, which are both with length equal to 6.

Next, the **Tanh function** is set for the (-1, 1) interval of action space, shown below. Finally, return the output of Tanh function as the action with n_actions = 2.

```
print(env.action_space.high)
print(env.action_space.low)
```

```
[1. 1.]
[-1. -1.]
```

Optimizer of actor network is set as RMSprop, and loss is MSE loss. The mentioned settings are followed by teacher's slides about DDPG:

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:

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

Critic

The difference between actor and critic network is the input dimension and the activation of neural network. Input of critic network is not only the concatenation of state and goal but also the action.

```
class Critic(nn.Module):
    def __init__(self, input_dims, n_actions, learning_rate):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(2 * input_dims + n_actions, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 1)

        self.optimizer = optim.RMSprop(self.parameters(), lr=learning_rate)
        self.loss = nn.MSELoss()
        self.device_type = 'cuda:0' if torch.cuda.is_available() else 'cpu'
        self.device = torch.device(self.device_type)
        self.to(self.device)

    def forward(self, data1, data2):
        fc_layer1 = F.relu(self.fc1(torch.cat((data1, data2), 1)))
        fc_layer2 = F.relu(self.fc2(fc_layer1))
        value = self.fc3(fc_layer2)
        return value
```

I didn't implement any activation function after the final fully connected layer, because the output is the value of the state and action.

HER

```
class HindsightExperienceReplayMemory(object):
    def __init__(self, memory_size, input_dims, n_actions):
        super(HindsightExperienceReplayMemory, self).__init__()
        self.max_mem_size = memory_size
        self.counter = 0
        self.state_memory = np.zeros((memory_size, input_dims), dtype=np.float32)
        self.next_state_memory = np.zeros((memory_size, input_dims), dtype=np.float32)
        self.reward_memory = np.zeros(memory_size, dtype=np.float32)
        self.action_memory = np.zeros((memory_size, n_actions), dtype=np.float32)
        self.terminal_memory = np.zeros(memory_size, dtype=bool)
        self.goal_memory = np.zeros((memory_size, input_dims), dtype=np.float32)

    def add_experience(self, state, action, reward, next_state, done, goal):
        curr_index = self.counter % self.max_mem_size
        self.state_memory[curr_index] = state
        self.action_memory[curr_index] = action
        self.reward_memory[curr_index] = reward
        self.next_state_memory[curr_index] = next_state
        self.terminal_memory[curr_index] = done
        self.goal_memory[curr_index] = goal
        self.counter += 1

    def get_random_experience(self, batch_size):
        rand_index = np.random.choice(min(self.counter, self.max_mem_size), batch_size, replace=False)
        rand_state = self.state_memory[rand_index]
        rand_action = self.action_memory[rand_index]
        rand_reward = self.reward_memory[rand_index]
        rand_next_state = self.next_state_memory[rand_index]
        rand_done = self.terminal_memory[rand_index]
        rand_goal = self.goal_memory[rand_index]

        return rand_state, rand_action, rand_reward, rand_next_state, rand_done, rand_goal
```

Above is the class definition of the replay buffer, which is simply a buffer with random sampling function and stores experiences of states, actions, rewards, next states, termination flags and goals.

In training process, I use the current policy to generate whole trajectories:

```
    for p in range(50):
        if not done:
            action = agent.choose_action(state, goal, True)
            obs, reward, done, info = env.step(action)
            next_state = obs['observation']
            agent.store_experience(state, action, reward, next_state, done, goal)
            transitions.append((state, action, reward, next_state, info))
            state = next_state
            if done:
                success += 1
```

for episode = 1, M **do**
  Sample a goal g and an initial state s_0.
  for $t = 0, T-1$ **do**
    Sample an action a_t using the behavioral policy from $\mathbb{A}$:
      $a_t \leftarrow \pi_b(s_t||g)$
    Execute the action a_t and observe a new state s_{t+1}
  end for

As the architecture mentioned in the teacher's slides about HER (above), I set T as 50 and sample actions using the behavioral policy from the agent. Also, compute the rewards and store the transition.

for $t = 0, T-1$ **do**
  $r_t := r(s_t, a_t, g)$
  Store the transition $(s_t||g,\ a_t,\ r_t,\ s_{t+1}||g)$ in R ▷ standa
  Sample a set of additional goals for replay $G := \mathbb{S}$(**current episode**)
  for $g' \in G$ **do**
    $r' := r(s_t, a_t, g')$
    Store the transition $(s_t||g',\ a_t,\ r',\ s_{t+1}||g')$ in R
  end for
end for
for $t = 1, N$ **do**
  Sample a minibatch B from the replay buffer R
  Perform one step of optimization using $\mathbb{A}$ and minibatch B
end for

```
if not done:
    rdm = random.sample(range(0, 50), 20)
    rdm.sort()
    # print(rdm)
    for q in range(20):
        if q != 19:
            p = rdm[random.randint(q, 19)]
            new_goal = transitions[p][0]
        else:
            new_goal = np.copy(state)
        transition = transitions[rdm[q]]
        new_reward = env.compute_reward(transition[3], new_goal, transition[4])
        agent.store_experience(transition[0], transition[1], new_reward,
                               transition[3], np.array_equal(transition[3], new_goal), new_goal)
for i in range(50):
    agent.learn()
```

Later on (above), I sample a set of transitions from the previous step and set additional goals for those states. The additional goals are picked from the preceding sampled states, which means that the later on pass-by states are set as temporary additional goals. After computing the rewards of the states and new rewards, store the new transition into the buffer.

Finally, make the agent learn from the experience.

DDPG

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

```
class DDPGAgent:
    def __init__(self, actor_learning_rate, critic_learning_rate, n_actions,
                 input_dims, gamma, memory_size, batch_size, tau=0.001):
        self.actor_learning_rate = actor_learning_rate
        self.critic_learning_rate = critic_learning_rate
        self.n_actions = n_actions
        self.input_dims = input_dims
        self.gamma = gamma
        self.memory_size = memory_size
        self.batch_size = batch_size
        self.tau = tau

        self.actor = Actor(input_dims=input_dims, n_actions=n_actions,
                           learning_rate=actor_learning_rate)

        self.critic = Critic(input_dims=input_dims, n_actions=n_actions,
                             learning_rate=critic_learning_rate)

        self.target_actor = Actor(input_dims=input_dims, n_actions=n_actions,
                                  learning_rate=actor_learning_rate)

        self.target_critic = Critic(input_dims=input_dims, n_actions=n_actions,
                                    learning_rate=critic_learning_rate)

        self.memory = HindsightExperienceReplayMemory(memory_size=memory_size,
                                                      input_dims=input_dims, n_actions=n_actions)

        self.ou_noise = OrnsteinUhlenbeckActionNoise(mu=np.zeros(n_actions))
```

First, initialize the networks, buffer and action noise.

Ornstein-Uhlenbeck Noise:

```
def __call__(self):
    x = self.x_prev + self.theta * (self.mu - self.x_prev) * self.dt + \
        self.sigma * np.sqrt(self.dt) * np.random.normal(size=self.mu.shape)
    self.x_prev = x
    return x
```

```python
def store_experience(self, state, action, reward, next_state, done, goal):
    self.memory.add_experience(state=state, action=action,
                               reward=reward, next_state=next_state,
                               done=done, goal=goal)

def get_sample_experience(self):
    state, action, reward, next_state, done, goal = self.memory.get_random_experience(
        self.batch_size)

    t_state = torch.tensor(state).to(self.actor.device)
    t_action = torch.tensor(action).to(self.actor.device)
    t_reward = torch.tensor(reward).to(self.actor.device)
    t_next_state = torch.tensor(next_state).to(self.actor.device)
    t_done = torch.tensor(done).to(self.actor.device)
    t_goal = torch.tensor(goal).to(self.actor.device)

    return t_state, t_action, t_reward, t_next_state, t_done, t_goal
```

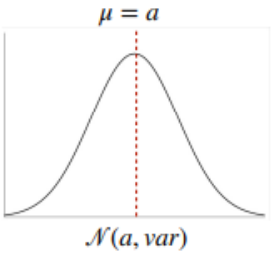
Next, the store experience function calls the add experience function of the HER buffer, and the get sample experience function calls the get random experience function of the HER buffer.

```python
def choose_action(self, observation, goal, train):
    if train:
        if np.random.random() > 0.1:
            state = torch.tensor([np.concatenate([observation, goal])], dtype=torch.float).to(self.actor.device)
            mu = self.actor.forward(state).to(self.actor.device)
            action = mu + torch.tensor(self.ou_noise(), dtype=torch.float).to(self.actor.device)

            self.actor.train()
            selected_action = action.cpu().detach().numpy()[0]
        else:
            selected_action = np.random.uniform(-1, 1, 2)
    else:
        state = torch.tensor([np.concatenate([observation, goal])], dtype=torch.float).to(self.actor.device)
        mu = self.actor.forward(state).to(self.actor.device)
        action = mu
        selected_action = action.cpu().detach().numpy()[0]

    return selected_action
```

Furthermore, the choose action function adds noise terms on the selection of the actor during the training phase, adjusting the variance of noise to control the extent of exploration behavior. On the other hand, I implemented epsilon greedy with epsilon equals to 0.1.

- Solution: adding noise terms during the training phase

$$a = \mu(s, \theta_\mu) + \mathcal{N}_{noise}$$



  - $\mathcal{N}_{noise}$: Ornstein-Uhlenbeck process (Link)
  - $\mu(s, \theta_\mu)$: The policy function parameterized by θ_μ
  - Adjust the variance of $\mathcal{N}_{noise}$ to control the extent of exploration behavior

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:

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

```
def learn(self):
    if self.memory.counter < self.batch_size:
        return

    self.actor.optimizer.zero_grad()
    self.critic.optimizer.zero_grad()

    state, action, reward, next_state, done, goal = self.get_sample_experience()
    concat_state_goal = torch.cat((state, goal), 1)
    concat_next_state_goal = torch.cat((next_state, goal), 1)

    target_actions = self.target_actor.forward(concat_state_goal)
    critic_next_value = self.target_critic.forward(concat_next_state_goal, target_actions).view(-1)

    actor_value = self.actor.forward(concat_state_goal)
    critic_value = self.critic.forward(concat_state_goal, action)

    critic_value[done] = 0.0
```

Finally, the learn function. First of all, sample a batch of transitions with get sample experience function, concatenating states & goals as required of HER.

```
    target = (reward + self.gamma * critic_next_value).view(self.batch_size, -1)
```

Next, set the target to reward + gamma * next critic value.

```
    loss_critic = self.critic.loss(target, critic_value)
    loss_critic.backward()
    self.critic.optimizer.step()

    loss_actor = -torch.mean(self.critic.forward(concat_state_goal, actor_value))
    loss_actor.backward()
    self.actor.optimizer.step()
```

Then, update the critic network with its loss function, MSE, and update the actor policy using the sampled policy gradient.

```
actor_parameters = dict(self.actor.named_parameters())
critic_parameters = dict(self.critic.named_parameters())
target_actor_parameters = dict(self.target_actor.named_parameters())
target_critic_parameters = dict(self.target_critic.named_parameters())

for i in actor_parameters:
    actor_parameters[i] = self.tau * actor_parameters[i].clone() + (1 - self.tau) * target_actor_parameters[i].clone()

for i in critic_parameters:
    critic_parameters[i] = self.tau * critic_parameters[i].clone() + (1 - self.tau) * target_critic_parameters[i].clone()

self.target_actor.load_state_dict(actor_parameters)
self.target_critic.load_state_dict(critic_parameters)
```

Finally, update the target networks with parameter tau.

b. Experiment and Discussion

In the beginning, my agent keeps circling until it accidentally reaches the parking spot. The network output actions with activation layer ReLU at that time due to my mistake on not checking the action space. Later on, I changed it in to tanh, and the agent finally works.

2. Racetrack
    a. Implementation (PPO clipping)

Actor & Critic

```
# actor
self.actor = nn.Sequential(
            nn.Conv2d(state_dim, out_channels=8, kernel_size=3, padding=1),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(),
            nn.Conv2d(8, out_channels=16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(),
            nn.Flatten(),
            nn.Linear(2304, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, action_dim),
            nn.Tanh()
            )

# critic
self.critic = nn.Sequential(
            nn.Conv2d(state_dim, out_channels=8, kernel_size=3, padding=1),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(),
            nn.Conv2d(8, out_channels=16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(),
            nn.Flatten(),
            nn.Linear(2304, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, action_dim),
            )
```

Two convolutional layers with batch normalization and leaky ReLU activation, connected to a flatten layer and 3 fully connected layers with tanh activation. Input state is (2, 12, 12) representing occupancy map with 2 channels, and output action dimension is 1.

```
def act(self, state):
    action_mean = self.actor(state)
    cov_mat = torch.diag(self.action_var).unsqueeze(dim=0)
    dist = MultivariateNormal(action_mean, cov_mat)

    action = dist.sample()
    action_logprob = dist.log_prob(action)
    return action.detach(), action_logprob.detach()
```

In the act function, model outputs the mean of the distribution of action selection. Using the covariation matrix constructed with the current action standard deviation, which decays during the training process due to reducing proportion of exploration, the model builds a multivariate normal distribution of the action selection. Finally, sample the action from the distribution and also return the corresponding logarithm probability. Below is the setup of action variance using the current action standard deviation.

```
def set_action_std(self, new_action_std):
    self.action_var = torch.full((self.action_dim,), new_action_std * new_action_std).to(device)
```

```
def evaluate(self, state, action):
    action_mean = self.actor(state)
    action_var = self.action_var.expand_as(action_mean)
    cov_mat = torch.diag_embed(action_var).to(device)
    dist = MultivariateNormal(action_mean, cov_mat)
    if self.action_dim == 1:
        action = action.reshape(-1, self.action_dim)

    action_logprobs = dist.log_prob(action)
    dist_entropy = dist.entropy()
    state_values = self.critic(state)

    return action_logprobs, state_values, dist_entropy
```

Above is the evaluation function which will be called when updating the PPO agent. Similarly, it constructs a multivariate normal distribution with the action mean received from actor model and the covariance matrix built with the current variance. It returns the log probability of the sampled action from the distribution, the state value provided by the critic model and the entropy of the multivariate Gaussian distribution.

PPO

```
def set_action_std(self, new_action_std):
    self.action_std = new_action_std
    self.policy.set_action_std(new_action_std)
    self.policy_old.set_action_std(new_action_std)

def decay_action_std(self, action_std_decay_rate, min_action_std):
    self.action_std = self.action_std - action_std_decay_rate
    self.action_std = round(self.action_std, 4)
    if (self.action_std <= min_action_std):
        self.action_std = min_action_std
    self.set_action_std(self.action_std)
```

The PPO agent decays the action standard deviation every decided timesteps, and affects the construction of normal distribution.

```
def select_action(self, state):
    with torch.no_grad():
        state = torch.FloatTensor(state).to(device)
        action, action_logprob = self.policy_old.act(state)
    self.buffer.states.append(state)
    self.buffer.actions.append(action)
    self.buffer.logprobs.append(action_logprob)

    return action.detach().cpu().numpy().flatten()
```

The select action function calls the actor of the old policy to act to the input state, simultaneously stores the state, action and action log probability into the buffer.

```
def update(self):
    rewards = []
    discounted_reward = 0
    for reward, is_terminal in zip(reversed(self.buffer.rewards), reversed(self.buffer.is_terminals)):
        if is_terminal:
            discounted_reward = 0
        discounted_reward = reward + (self.gamma * discounted_reward)
        rewards.insert(0, discounted_reward)

    rewards = torch.tensor(rewards, dtype=torch.float32).to(device)
    rewards = (rewards - rewards.mean()) / (rewards.std() + 1e-7)

    old_states = torch.squeeze(torch.stack(self.buffer.states, dim=0)).detach().to(device)
    old_actions = torch.squeeze(torch.stack(self.buffer.actions, dim=0)).detach().to(device)
    old_logprobs = torch.squeeze(torch.stack(self.buffer.logprobs, dim=0)).detach().to(device)
```

In the update function, first of all, calculate the rewards with respect to the discount factor. Here, the calculation is implemented reversely of the storing process, and the reward should be updated to a new one whenever it is a termination. Later on, normalize the rewards.

```
for _ in range(self.K_epochs):
    logprobs, state_values, dist_entropy = self.policy.evaluate(old_states, old_actions)

    state_values = torch.squeeze(state_values)

    ratios = torch.exp(logprobs - old_logprobs.detach())
```

Next, in a loop of updating epochs, call the evaluation function of the policy with input states and actions picked from the buffer. Calculate the ratio of log probability of the policy and the log probability of the old policy, just like the following picture captured from the teacher's slides.

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}$$

```
advantages = rewards - state_values.detach()
surr1 = ratios * advantages
surr2 = torch.clamp(ratios, 1-self.eps_clip, 1+self.eps_clip) * advantages
```

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)],$$

Then, calculate the Surrogate loss, and later on find the clipped surrogate objective. First object of the surrogate is the product of the previous ratio and advantages, and the second object is the product of advantages and the clamp of ratio, 1 – epsilon and 1 + epsilon to allow limitation of L^{CLIP}.

```
loss = -torch.min(surr1, surr2) + 0.5*self.MseLoss(state_values, rewards) - 0.01*dist_entropy

self.optimizer.zero_grad()
loss.mean().backward()
self.optimizer.step()
```

$$L(\theta) = \mathbb{E}[L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta](s_t)],$$

Finally, calculate the overall objective function. I set c_1 to 0.5 and c_2 to 0.01. The L^{VF} is the squared-error loss of the value function of the critic model, and the S is the entropy bonus of the action distribution provided by the evaluation function.

```
self.policy_old.load_state_dict(self.policy.state_dict())
self.buffer.clear()
```

After updating through the update epochs, update the old policy to the policy, and clear the buffer.

Parameters

Maximum episode length: 300 (environment maximum length is 300)
Initial action std: 0.7
Action std decay rate: 0.02
Action std decay frequency: 25000
K_epochs: 40
Clipped epsilon: 0.2
Gamma: 0.9
Actor learning rate: 0.0003
Critic learning rate: 0.0005

b. Experiments and Discussion

In the beginning, I trained twice and received two different models: one is familiar at driving in the inside lane and capable to overtake another car on the inside lane, but unable to overtake the car on the outside lane when initially driving in the outside lane. Another one is the opposite.

Therefore, I continued training the specific agent with a smaller episode length in order to emphasize the importance of avoiding crashing in a short

lifetime. Finally, I received a better model which is very good at driving on one lane and usually capable to drive on another lane.