# ECE433/COS435 Introduction to RL
# Assignment 7: Actor-critic Algorithms for continuous action space: SAC
# Spring 2025

| Fill me in |
| --- |
| Your name here. |

Due April 11, 2025

## Collaborators

| Fill me in |
| --- |
| Please fill in the names and NetIDs of your collaborators in this section. |

## Instructions

Writeups should be typesetted in Latex and submitted as PDFs. You can work with whatever tool you like for the code, but **please submit the asked-for snippet and answer in the solutions box as part of your writeup. We will only be grading your write-up.**

In this assignment, we will introduce two modern RL algorithms that aim to tackle MDPs with continuous action space: Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3).

The workflow for the rest of this assignment is as follows: we will implement the SAC algorithm, and then evaluate their performance on the Gym environment Pendulum-v1. We provide a sample implementation to TD3, which can be helpful as a reference for your implementation of SAC.

## Question 1. Soft Actor-Crtic (SAC)

### Question 1.a

First, you want to build your Actor_SAC and Critic_SAC networks. Paste your class below:

Solution

```python
class Actor_SAC(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor_SAC, self).__init__()
        # [HINT] Construct a neural network as the actor. Return its value using forward You need to write down three linear layers.
        # 1. l1: state_dim -> 256
        # 2. l2: 256 -> 256
        # 3. l3: 256 -> mean and log std of the action
        ############################
        # YOUR IMPLEMENTATION HERE #
        pass
        ############################
        self.max_action = max_action

    def forward(self, state):
        # [HINT] Use the three linear layers to compute the mean and log std of the action
        # Apply ReLU activation after layer l1 and l2
        ############################
        # YOUR IMPLEMENTATION HERE #
        pass
        ############################
        log_std = torch.clamp(log_std, min=LOG_STD_MIN, max=LOG_STD_MAX)
        return mean, log_std

    def sample(self, state):
        # [HINT] Use the forward method to compute the action, its log probability
        # 1. Compute the mean and log std of the action
        # 2. Compute the standard deviation of the action
        # 3. Get the normal distribution of the action
        # 4. Sample the action from the normal distribution
        # 5. Apply tanh to the action and multiply by max_action to ensure the action is in the range of the action space
        # 6. Compute the log probability of the action

        ############################
        # YOUR IMPLEMENTATION HERE #
        pass
        ############################
        return action, log_prob
```

```python
class Critic_SAC(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic_SAC, self).__init__()
        # Q1 architecture
        # [HINT] Construct a neural network as the first critic. Return its value using forward You need to write down three linear layers.
        # 1. l1: state_dim+action_dim -> 256
```

```
        # 2. l2: 256 -> 256
        # 3. l3: 256 -> 1
        ############################
        # YOUR IMPLEMENTATION HERE #
        pass
        ############################

        # Q2 architecture
        # [HINT] Construct a neural network as the second critic. Return its value using forward. You need to write down three linear layers.
        # 1. l4: state_dim+action_dim -> 256
        # 2. l5: 256 -> 256
        # 3. l6: 256 -> 1
        ############################
        # YOUR IMPLEMENTATION HERE #
        pass
        ############################


    def forward(self, state, action):
        sa = torch.cat([state, action], 1)
        # [HINT] We use layers l1, l2, l3 to obtain q1
        # 1. Apply ReLU activation after layer l1
        # 2. Apply ReLU activation after layer l2
        # 3. Return output as q1 from layer l3

        # [HINT] We use layers l4, l5, l6 to obtain q2
        # 1. Apply ReLU activation after layer l4
        # 2. Apply ReLU activation after layer l5
        # 3. Return output as q2 from layer l6

        ############################
        # YOUR IMPLEMENTATION HERE #
        pass
        ############################
        return q1, q2


    def Q1(self, state, action):
        sa = torch.cat([state, action], 1)
        # [HINT] only returns q1 for actor update using layers l1, l2, l3
        # 1. Apply ReLU activation after layer l1
        # 2. Apply ReLU activation after layer l2
        # 3. Return output as q1 from layer l3
        ############################
        # YOUR IMPLEMENTATION HERE #
        pass
        ############################
        return q1
```

## Question 1.b

Now we are ready to construct a SAC trainer! In the following function you will need to: (1) Calculate the TD value using target_Q network and update the critic; (2) Compute the actor loss using the current state and sampled action and update the actor; (3) Update the target networks.

**Solution**

```python
def train(self, replay_buffer, batch_size=256):
    state, action, next_state, reward, not_done = replay_buffer.sample(batch_size)

    # [HINT] compute the target Q value
    # 1. Sample the next action and its log probability from the actor with next_state
    # 2. Compute the next Q values (Q1 and Q2) using the critic_target with next_state and next_action
    # 3. Min over the Q values: target_Q = min(Q1, Q2) - log_prob(a'|s') * alpha
    # 4. Compute the target Q value: target_Q = reward + not_done * discount * target_Q

    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################

    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

    # [HINT] compute the actor loss
    # 1. Sample the action and its log probability from the actor with state
    # 2. Compute the Q values (Q1 and Q2) using the critic with state and action
    # 3. Min over the Q values: Q = min(Q1, Q2)
    # 4. Compute the actor loss: actor_loss = alpha * log_prob(a|s) - Q

    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################

    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()

    for param, target_param in zip(self.critic.parameters(), self.critic_target.parameters()):
```

```
            target_param.data.copy_(self.tau * param.data + (1 - self.tau
    ) * target_param.data)
```

## Question 1.c

Time to see how it works. We would expect a reward that converges to negative values close to 0. The estimated wall time for running the whole process is around $10 - 20$ minutes, and you should be able to see a large positive reward at around 50 episodes.

**Solution**

Training curve: