Implementing Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC) on inverted pendulum

We complete this lab as a group of three. All group members contributed equally.

Willem Meijer (willem.meijer@liu.se)
Yikun Hou (yikun.hou@umu.se)
Yiran Wang (yiran.wang@liu.se)

DDPG

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import torch.optim as optim
import random
from collections import deque, namedtuple
from utilities import OUNoise, OUActionNoise

import gymnasium as gym
import copy
from plotting import plot_losses, plot_score
from torch.distributions import MultivariateNormal, Normal
```

In [2]:
```
class DDPG_actor(nn.Module):
    def __init__(self, state_size, action_size, layer_size):
        super(DDPG_actor, self).__init__()

        self.state_size = state_size
        self.fca1 = nn.Linear(self.state_size, layer_size)
        self.fca2 = nn.Linear(layer_size, layer_size)
        """Start your code here. 1 line of code"""
        self.action_values = nn.Linear(layer_size, action_size)
        """End your code here"""


    def forward(self, state):
        x = torch.relu(self.fca1(state))
        x = torch.relu(self.fca2(x))

        # Policy
        """Start your code here. 1 line of code."""
        action_value = 2 * torch.tanh(self.action_values(x)) # tanh normalize output to [-1, 1] but we need [-2, 2]
        """End your code here."""
        return action_value
```

In [3]:
```
class DDPG_critic(nn.Module):
    def __init__(self, state_size, action_size, layer_size):
        super(DDPG_critic, self).__init__()

        self.state_size = state_size
        self.action_size = action_size

        self.fcs1 = nn.Linear(self.state_size, 16)
        self.fcs2 = nn.Linear(16, 32)

        self.fcu2 = nn.Linear(self.action_size, 32)

        self.fcc3 = nn.Linear(64, layer_size)
        self.fcc4 = nn.Linear(layer_size, layer_size)

        """Start your code here. 1 line of code."""
        self.qvalue = nn.Linear(layer_size, 1) # Q is a scalar
        """End your code here"""

    def forward(self, state, action):
        x = torch.relu(self.fcs1(state))
        x = torch.relu(self.fcs2(x))

        u = torch.relu(self.fcu2(action))

        z = torch.cat([x, u], 1)
```

```
        z = torch.relu(self.fcc3(z))
        z = torch.relu(self.fcc4(z))
        """Start your code here. 1 line of code."""
        Q = self.qvalue(z)
        """End your code here."""
        return Q
```

In [4]:
```
class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, buffer_size, batch_size, device, gamma):
        """Initialize a ReplayBuffer object.
        Params
        ======
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.device = device
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        self.gamma = gamma

        self.n_step_buffer = deque(maxlen=1)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""

        self.n_step_buffer.append((state, action, reward, next_state, done))
        if len(self.n_step_buffer) == 1:
            state, action, reward, next_state, done = self.calc_multistep_return()

            e = self.experience(state, action, reward, next_state, done)
            self.memory.append(e)

    def calc_multistep_return(self):
        Return = 0
        for idx in range(1):
            Return += self.gamma ** idx * self.n_step_buffer[idx][2]

        return self.n_step_buffer[0][0], self.n_step_buffer[0][1], Return, self.n_step_buffer[-1][3], \
        self.n_step_buffer[-1][4]

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.stack([e.state for e in experiences if e is not None])).float().to(self.device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).float().to(self.device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(self.device)
        next_states = torch.from_numpy(np.stack([e.next_state for e in experiences if e is not None])).float().to(
            self.device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(
            self.device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

In [5]:
```
class DDPG_Agent():
    """Interacts with and learns from the environment."""

    def __init__(self,
                 state_size,
                 action_size,
                 layer_size,
                 BATCH_SIZE,
                 BUFFER_SIZE,
                 LR_critic,
                 LR_actor,
                 TAU,
                 GAMMA,
                 device):
        self.state_size = state_size
        self.action_size = action_size
        self.device = device
```

```
        self.TAU = TAU
        self.GAMMA = GAMMA
        self.BATCH_SIZE = BATCH_SIZE
        self.noise = OUActionNoise(mean=np.zeros(1), std_deviation=float(0.2) * np.ones(1))

        # Q-Networks
        self.qnetwork_local = DDPG_critic(state_size, action_size, layer_size).to(device)
        self.qnetwork_target = DDPG_critic(state_size, action_size, layer_size).to(device)
        self.optimizer_critic = optim.Adam(self.qnetwork_local.parameters(), lr=LR_critic)

        # Actor-Networks
        self.actor_network_local = DDPG_actor(state_size, action_size, layer_size).to(device)
        self.actor_network_target = DDPG_actor(state_size, action_size, layer_size).to(device)
        self.optimizer_actor = optim.Adam(self.actor_network_local.parameters(), lr=LR_actor)

        # Replay memory
        self.memory = ReplayBuffer(BUFFER_SIZE, BATCH_SIZE, self.device, self.GAMMA)



    def get_action(self, state):
        state = torch.from_numpy(state).float().to(self.device)
        self.actor_network_local.eval()
        with torch.no_grad():
            action = self.actor_network_local(state.unsqueeze(0))
            noise = self.noise()
            action = action.cpu().squeeze().numpy() + noise
            action = np.clip(action, -2, 2)[0]
        self.actor_network_local.train()
        return action

    def learn(self, experiences):
        """Update value parameters using given batch of experience tuples.
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
        """

        # Unpack the sampled data
        states, actions, rewards, next_states, dones = experiences

        ## Critic network
        self.optimizer_critic.zero_grad()

        with torch.no_grad():
            # The Q from the target network
            next_inputs = self.actor_network_target(next_states)
            Q_network_ = self.qnetwork_target(next_states, next_inputs)

        """Start your code here. 3 lines of code."""
        # The target Q for the current state using the target V
        Q_targets = rewards + (self.GAMMA * Q_network_ * (1 - dones))

        # The Q from the local network. We should find the gradient so we do not put it in no_grad
        Q = self.qnetwork_local(states, actions)

        # Compute the critic loss
        loss_critic = F.mse_loss(Q, Q_targets)
        """End your code here."""

        loss_critic.backward()
        # clip_grad_norm_(self.qnetwork_local.parameters(), 1)
        self.optimizer_critic.step()

        ## Actor network
        self.optimizer_actor.zero_grad()

        """Start your code here. 3 lines of code."""
        actions_local = self.actor_network_local(states)
        q_local = self.qnetwork_local(states, actions_local)
        loss_actor = -q_local.mean()
        """End your code"""

        # Minimize the critic and actor losses
        loss_actor.backward()
        self.optimizer_actor.step()


        # ------------------- update target network ------------------- #
        for target_param, local_param in zip(self.qnetwork_target.parameters(), self.qnetwork_local.parameters()):
```

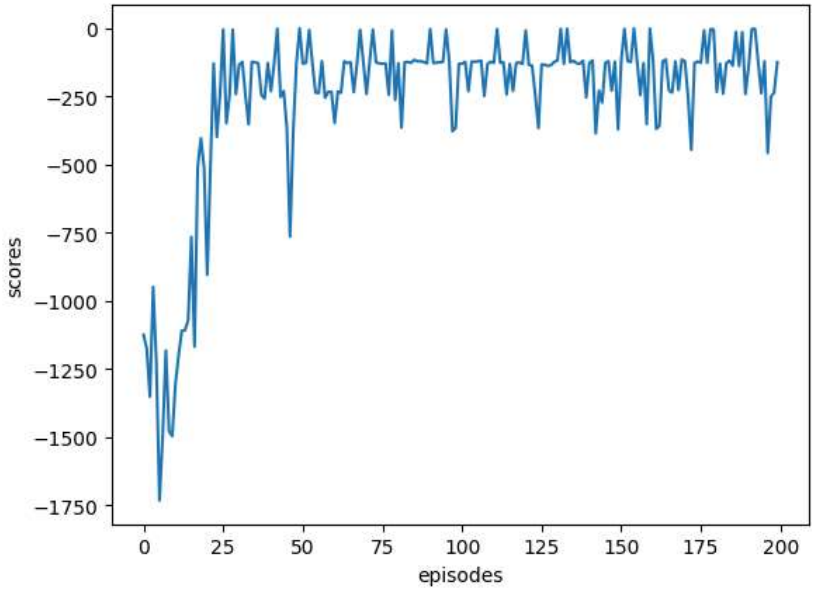
```
            target_param.data.copy_(self.TAU * local_param.data + (1.0 - self.TAU) * target_param.data)

        # ------------------- update target network ------------------- #
        for target_param, local_param in zip(self.actor_network_target.parameters(), self.actor_network_local.parameters()):
            target_param.data.copy_(self.TAU * local_param.data + (1.0 - self.TAU) * target_param.data)

        # self.noise.reset()

        return loss_critic.detach().cpu().numpy()

    def rollout(self, env):
        state, info = env.reset()
        done = False
        truncated = False
        score = 0.0
        loss = 0.0
        Q_losses = []
        while not (done or truncated):

            # Get the action
            action = self.get_action(state)

            # Apply the action
            next_state, reward, done, truncated, _ = env.step(np.array([action]))

            # Add the sample to the memory
            self.memory.add(state, action, reward, next_state, done)

            # Sample from buffer and learn
            if len(self.memory) > self.BATCH_SIZE:
                experiences = self.memory.sample()
                loss = self.learn(experiences)
                Q_losses.append(loss)
                #writer.add_scalar("Q_loss", np.mean(Q_losses))
                loss = np.mean(Q_losses)

            # Update the new state and score
            state = next_state
            score += reward
        return score, loss
```

In [6]:
```
def run_ddpg(env):
    #writer = SummaryWriter("runs/" + "DDPG_test")
    seed = 0
    LAYER_SIZE = 256 # Original: 256
    BUFFER_SIZE = 50000 # Original: 50000
    BATCH_SIZE = 128 # Original: 128
    GAMMA = 0.99 # Original: 0.99
    TAU = 0.005 # Original: 0.005
    LR_critic = 0.002 # Original: 0.002
    LR_actor = 0.001 # Original: 0.001
    NUMBER_OF_EPISODES = 200
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print("Using ", device)
    np.random.seed(seed)
    torch.manual_seed(seed)
    random.seed(seed)

    action_size = env.action_space.shape[0]
    state_size = env.observation_space.shape[0]

    agent = DDPG_Agent(state_size=state_size,
                       action_size=action_size,
                       layer_size=LAYER_SIZE,
                       BATCH_SIZE=BATCH_SIZE,
                       BUFFER_SIZE=BUFFER_SIZE,
                       LR_critic=LR_critic,
                       LR_actor=LR_actor,
                       TAU=TAU,
                       GAMMA=GAMMA,
                       device=device)

    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    for i_episode in range(NUMBER_OF_EPISODES):
        # Run a rollout
        score, _ = agent.rollout(env)
        scores_window.append(score)       # save most recent score
        scores.append(score)              # save most recent score
        if i_episode % 5 == 0:
```

```
            print('\rEpisode {} \tTotal reward: {:.2f} \tAverage Score: {:.2f}'.format(i_episode, score, np.mean(scores_window
    plot_score(scores)
```

In [7]:
```
if __name__ == "__main__":
    env = gym.make('Pendulum-v1')
    run_ddpg(env)
```

```
Using  cpu
Episode 0       Total reward: -1125.11  Average Score: -1125.11
Episode 5       Total reward: -1732.76  Average Score: -1259.65
Episode 10      Total reward: -1305.96  Average Score: -1319.66
Episode 15      Total reward: -766.11   Average Score: -1235.76
Episode 20      Total reward: -904.75   Average Score: -1108.09
Episode 25      Total reward: -4.64     Average Score: -944.36
Episode 30      Total reward: -134.53   Average Score: -823.54
Episode 35      Total reward: -125.49   Average Score: -736.18
Episode 40      Total reward: -230.71   Average Score: -670.56
Episode 45      Total reward: -366.62   Average Score: -618.98
Episode 50      Total reward: -129.78   Average Score: -585.72
Episode 55      Total reward: -237.91   Average Score: -546.54
Episode 60      Total reward: -347.57   Average Score: -521.26
Episode 65      Total reward: -124.93   Average Score: -494.56
Episode 70      Total reward: -240.98   Average Score: -470.09
Episode 75      Total reward: -130.14   Average Score: -445.91
Episode 80      Total reward: -128.80   Average Score: -427.94
Episode 85      Total reward: -116.31   Average Score: -413.03
Episode 90      Total reward: -2.97     Average Score: -395.83
Episode 95      Total reward: -4.42     Average Score: -380.50
Episode 100     Total reward: -130.30   Average Score: -365.31
Episode 105     Total reward: -122.02   Average Score: -308.21
Episode 110     Total reward: -127.55   Average Score: -246.12
Episode 115     Total reward: -130.71   Average Score: -199.85
Episode 120     Total reward: -8.20     Average Score: -171.11
Episode 125     Total reward: -133.38   Average Score: -168.43
Episode 130     Total reward: -118.51   Average Score: -165.17
Episode 135     Total reward: -119.90   Average Score: -159.24
Episode 140     Total reward: -127.20   Average Score: -156.94
Episode 145     Total reward: -128.08   Average Score: -158.49
Episode 150     Total reward: -118.62   Average Score: -154.13
Episode 155     Total reward: -121.28   Average Score: -150.48
Episode 160     Total reward: -119.05   Average Score: -147.03
Episode 165     Total reward: -229.10   Average Score: -150.53
Episode 170     Total reward: -121.71   Average Score: -151.39
Episode 175     Total reward: -126.56   Average Score: -157.04
Episode 180     Total reward: -233.79   Average Score: -153.08
Episode 185     Total reward: -137.06   Average Score: -152.05
Episode 190     Total reward: -139.51   Average Score: -152.53
Episode 195     Total reward: -121.27   Average Score: -152.35
```



Q: Plot the average reward vs number of episode. If you have written your code correctly, you agent should quickly reach an average reward around -250. Analyze your result. Can you get better by changing the hyper parameters? Why?

Due to inherent randomness, results may vary slightly between runs. In the most recent run with the default settings, the agent demonstrated rapid learning during the initial phase, achieving an average reward of approximately −250 after around 25 episodes. However, subsequent performance showed oscillations within the range of 0 to −500, with a concentration between −100 and −300. These results indicate that the agent struggles to consistently converge to an optimal policy.

Optimizing hyperparameters could improve the agent's performance. For example, key parameters like:

- 1, Discount rate (GAMMA): The discount rate balances the emphasis between current and future rewards. A lower discount rate can reduce sensitivity to long-term noise, potentially stabilizing training but sacrificing foresight in policy decisions.
- 2, Batch size (BATCH_SIZE): Larger batch sizes stabilize gradient calculations, mitigating training oscillations. However, excessively large batches may reduce the model's generalization capability.
- 3, Learning rate for the critic/actor (LR_critic/LR_actor): A higher learning rate accelerates training but risks oscillations or even divergence. Conversely, a lower learning rate ensures smoother learning at the cost of slower convergence.
- 4, Network size (LAYER_SIZE): Smaller network layers result in faster computation but may lead to suboptimal performance in complex environments due to their limited capacity. On the other hand, larger networks provide greater modeling power but come with higher computational costs and an increased risk of overfitting.
- 5, Replay (BUFFER_SIZE): A small replay buffer restricts the variety of experiences the agent can sample from, limiting generalization capability. However, it requires less computational cost compared to larger buffers.
- 6, τ is discussed in the next question.

We tested various combinations of these hyperparameters but eventually retained the default parameters for reporting purposes.

Q: Explain the role of τ = 0.005 in this algorithm. Discuss the limiting cases τ = 0 and τ = 1. Can you get a better result by changing it? why?

The parameter τ controls the update of the target network, which is essential for stabilizing learning by providing an appropriate Q-target.

- If $\tau = 0$, the target network is never updated, causing the algorithm to fail to learn, as the Q-target no longer reflects the actual policy.
- On the other hand, if $\tau = 1$, the target network is fully updated to match the local network, resulting in unstable training.

Tuning τ can lead to better results because it balances stability and convergence speed. A smaller τ improves stability but slows convergence, while a larger τ increases responsiveness but risks instability.

We tested various value of τ but eventually retained the default parameters for reporting purposes.

SAC

In [9]:
```
import torch
import torch.nn as nn
from torch.distributions import Normal
import numpy as np
import torch.optim as optim
import torch.nn.functional as F
import random
from collections import deque, namedtuple
from utilities import OUNoise, OUActionNoise

import gymnasium as gym
from plotting import plot_losses, plot_score


def weights_init_(m):
    if isinstance(m, nn.Linear):
        torch.nn.init.xavier_uniform_(m.weight, gain=1)
        torch.nn.init.constant_(m.bias, 0)


class Q_network(nn.Module):
    def __init__(self, state_size, action_size, layer_size):
        super(Q_network, self).__init__()

        self.state_size = state_size
        self.action_size = action_size

        # Q1
        self.fcq1 = nn.Linear(self.state_size + self.action_size, layer_size)
        self.fcq2 = nn.Linear(layer_size, layer_size)
        self.q1_value = nn.Linear(layer_size, 1)

        # Q2
        """Start your code here. 3 lines of code"""
```

```
        self.fcq3 = nn.Linear(self.state_size + self.action_size, layer_size)
        self.fcq4 = nn.Linear(layer_size, layer_size)
        self.q2_value = nn.Linear(layer_size, 1)
        """End your code here."""

        self.apply(weights_init_)

    def forward(self, state, action):
        z = torch.cat([state, action], 1)

        # build Q1
        z1 = torch.relu(self.fcq1(z))
        z1 = torch.relu(self.fcq2(z1))
        Q1 = self.q1_value(z1)

        # build Q2
        z2 = torch.relu(self.fcq3(z))
        z2 = torch.relu(self.fcq4(z2))
        Q2 = self.q2_value(z2)
        return Q1, Q2


class Gaussian_policy_network(nn.Module):
    def __init__(self, state_size, action_size, layer_size, epsilon=1e-6):
        super(Gaussian_policy_network, self).__init__()

        self.log_sig_max = 2.0
        self.log_sig_min = -20.0
        self.max_action = 2.0
        self.min_action = -2.0
        self.epsilon = epsilon

        self.state_size = state_size
        self.action_size = action_size

        self.fcp1 = nn.Linear(self.state_size, layer_size)
        self.fcp2 = nn.Linear(layer_size, layer_size)
        self.policy_mean = nn.Linear(layer_size, action_size)
        self.policy_log_std = nn.Linear(layer_size, action_size)

        self.apply(weights_init_)

        self.action_scale = torch.FloatTensor([(self.max_action - self.min_action) / 2.])
        self.action_bias = torch.FloatTensor([(self.max_action + self.min_action) / 2.])

    def forward(self, state):
        x = torch.relu(self.fcp1(state))
        x = torch.relu(self.fcp2(x))
        """Start your code here. 2-3 lines of code"""
        mean = self.policy_mean(x)
        log_std = self.policy_log_std(x)
        log_std = torch.clamp(log_std, min=self.log_sig_min, max=self.log_sig_max)
        """End your code here."""
        return mean, log_std

    def sample_action(self, state):
        mean, log_std = self.forward(state)
        std = log_std.exp()
        normal = Normal(mean, std)

        """Start your code here"""
        x_t = normal.rsample()
        y_t = torch.tanh(x_t)
        action = y_t * self.action_scale + self.action_bias # Bounded action
        """End your code here"""

        # Enforcing Action Bound
        log_prob = normal.log_prob(x_t)
        log_prob -= torch.log(self.action_scale * (1 - y_t.pow(2)) + self.epsilon)
        log_prob = log_prob.sum(1, keepdim=True)
        # Enforcing Action Bound
        mean = torch.tanh(mean) * self.action_scale + self.action_bias
        return action, log_prob, mean

    def to(self, device):
        self.action_scale = self.action_scale.to(device)
        self.action_bias = self.action_bias.to(device)
        return super(Gaussian_policy_network, self).to(device)
```

```python
class Deterministic_policy_network(nn.Module):
    """
    For evaluaton only.
    """
    def __init__(self, state_size, action_size, layer_size):
        super(Deterministic_policy_network, self).__init__()

        self.log_sig_max = 2.0
        self.log_sig_min = -20.0
        self.max_action = 2.0
        self.min_action = -2.0

        self.state_size = state_size
        self.action_size = action_size

        self.fcp1 = nn.Linear(self.state_size, layer_size)
        self.fcp2 = nn.Linear(layer_size, layer_size)

        self.policy_mean = nn.Linear(layer_size, action_size)
        self.noise = torch.Tensor(action_size)

        self.apply(weights_init_)

        self.action_scale = torch.FloatTensor([(self.max_action - self.min_action) / 2.])
        self.action_bias = torch.FloatTensor([(self.max_action + self.min_action) / 2.])

    def forward(self, state):
        x = torch.relu(self.fcp1(state))
        x = torch.relu(self.fcp2(x))
        mean = torch.tanh(self.policy_mean(x)) * self.action_scale + self.action_bias
        return mean

    def sample_action(self, state):
        mean = self.forward(state)
        noise = self.noise.normal_(0., std=0.1)
        noise = noise.clamp(-0.25, 0.25)
        action = mean + noise
        return action, torch.tensor(0.), mean

    def to(self, device):
        self.action_scale = self.action_scale.to(device)
        self.action_bias = self.action_bias.to(device)
        self.noise = self.noise.to(device)
        return super(Deterministic_policy_network, self).to(device)


class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, buffer_size, batch_size, device, gamma):
        """Initialize a ReplayBuffer object.
        Params
        ======
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.device = device
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        # self.seed = random.seed(seed)
        self.gamma = gamma

        self.n_step_buffer = deque(maxlen=1)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""

        self.n_step_buffer.append((state, action, reward, next_state, done))
        if len(self.n_step_buffer) == 1:
            state, action, reward, next_state, done = self.calc_multistep_return()

            e = self.experience(state, action, reward, next_state, done)
            self.memory.append(e)

    def calc_multistep_return(self):
        Return = 0
        for idx in range(1):
            Return += self.gamma ** idx * self.n_step_buffer[idx][2]
```

```
        return self.n_step_buffer[0][0], self.n_step_buffer[0][1], Return, self.n_step_buffer[-1][3], \
            self.n_step_buffer[-1][4]

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.stack([e.state for e in experiences if e is not None])).float().to(self.device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).float().to(self.device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(self.device)
        next_states = torch.from_numpy(np.stack([e.next_state for e in experiences if e is not None])).float().to(
            self.device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(
            self.device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)


class SAC_Agent():
    """Interacts with and learns from the environment."""

    def __init__(self,
                 state_size,
                 action_size,
                 layer_size,
                 BATCH_SIZE,
                 BUFFER_SIZE,
                 LR,
                 ALPHA,
                 TAU,
                 GAMMA,
                 policy_type,
                 target_update_interval,
                 auto_tuning,
                 device):
        self.state_size = state_size
        self.action_size = action_size
        self.device = device
        self.alpha = ALPHA
        self.TAU = TAU
        self.GAMMA = GAMMA
        self.BATCH_SIZE = BATCH_SIZE
        self.target_update_interval = target_update_interval
        self.auto_tuning = auto_tuning
        self.n_steps = 0
        self.n_updates = 0
        self.start_steps = 1000

        # Q-Network-Local
        self.qnetwork_local = Q_network(state_size, action_size, layer_size).to(device)
        self.optimizer_critic = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Q-network-target
        self.qnetwork_target = Q_network(state_size, action_size, layer_size).to(device)


        # Actor-Networks
        if policy_type == 'Gaussian':

            if self.auto_tuning is True:
                self.target_entropy = -torch.prod(torch.Tensor(self.action_size).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=LR)

            self.policy = Gaussian_policy_network(self.state_size, self.action_size, layer_size).to(self.device)
            self.policy_optim = optim.Adam(self.policy.parameters(), lr=LR)
            print("Gaussian policy")

        else:
            self.alpha = 0
            self.automatic_entropy_tuning = False
            self.policy = Deterministic_policy_network(self.state_size, self.action_size, layer_size).to(self.device)
            self.policy_optim = optim.Adam(self.policy.parameters(), lr=LR)
            print("Deterministic policy")
```

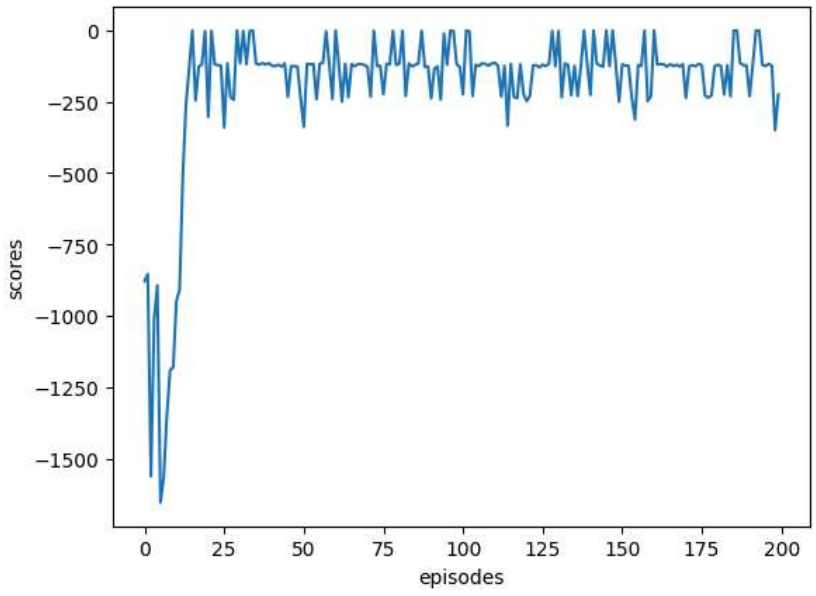
```
        # Replay memory
        self.memory = ReplayBuffer(BUFFER_SIZE, BATCH_SIZE, self.device, self.GAMMA)

    def select_action(self, state, evaluate=False):
        state = torch.FloatTensor(state).to(self.device).unsqueeze(0)
        if evaluate is False:
            action, _, _ = self.policy.sample_action(state)
        else:
            _, _, action = self.policy.sample_action(state)
        return action.detach().cpu().numpy()[0]

    def learn(self, experiences, updates):
        """Update value parameters using given batch of experience tuples.
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
        """

        # Unpack the sampled data
        states, actions, rewards, next_states, dones = experiences

        ## Critic network
        self.optimizer_critic.zero_grad()

        with torch.no_grad():
            """Start your code here. 4 lines of code"""
            next_state_action, next_state_log_pi, _ = self.policy.sample_action(next_states)
            qf1_next_target, qf2_next_target = self.qnetwork_target(next_states, next_state_action)
            v_target = torch.min(qf1_next_target, qf2_next_target) - self.alpha * next_state_log_pi
            q_target = rewards + (1-dones) * self.GAMMA * (v_target)
            """End your code here"""

        qf1, qf2 = self.qnetwork_local(states, actions)  # Two Q-functions to mitigate positive bias in the policy improvement
        qf1_loss = F.mse_loss(qf1, q_target)
        qf2_loss = F.mse_loss(qf2, q_target)
        qf_loss = qf1_loss + qf2_loss

        self.optimizer_critic.zero_grad()
        qf_loss.backward()
        self.optimizer_critic.step()

        pi, log_pi, _ = self.policy.sample_action(states)

        qf1_pi, qf2_pi = self.qnetwork_local(states, pi)
        min_qf_pi = torch.min(qf1_pi, qf2_pi)

        policy_loss = ((self.alpha * log_pi) - min_qf_pi).mean()
        self.policy_optim.zero_grad()
        policy_loss.backward()
        self.policy_optim.step()

        if self.auto_tuning:
            alpha_loss = -(self.log_alpha * (log_pi + self.target_entropy).detach()).mean()

            self.alpha_optim.zero_grad()
            alpha_loss.backward()
            self.alpha_optim.step()

            self.alpha = self.log_alpha.exp()
            alpha_tlogs = self.alpha.clone()  # For TensorboardX logs
        else:
            alpha_loss = torch.tensor(0.).to(self.device)
            alpha_tlogs = torch.tensor(self.alpha)  # For TensorboardX logs

        if updates % self.target_update_interval == 0:
            for target_param, local_param in zip(self.qnetwork_target.parameters(), self.qnetwork_local.parameters()):
                target_param.data.copy_(self.TAU * local_param.data + (1.0 - self.TAU) * target_param.data)

        return qf1_loss.item(), qf2_loss.item(), policy_loss.item(), alpha_loss.item(), alpha_tlogs.item()

    def rollout(self, env):
        state, info = env.reset()
        done = False
        truncated = False
        score = 0.0
        loss = 0.0
        Q1_losses = []
        Q2_losses = []
        Q_losses =[]
        entropy_losses = []
```

```
        policy_losses = []

        while not (done or truncated):

            # Get the action
            if self.start_steps > self.n_steps:
                action = env.action_space.sample()  # Sample random action
            else:
                action = self.select_action(state)  # Sample action from policy

            # Apply the action
            next_state, reward, done, truncated, _ = env.step(action)

            # Add the sample to the memory
            self.memory.add(state, action, reward, next_state, done)

            # Sample from buffer and learn
            if len(self.memory) > self.BATCH_SIZE:
                experiences = self.memory.sample()

                critic_1_loss, critic_2_loss, policy_loss, ent_loss, alpha = self.learn(experiences, self.n_updates)

                Q1_losses.append(critic_1_loss)
                Q2_losses.append(critic_2_loss)
                Q_losses.append((critic_1_loss+critic_2_loss)/2)
                policy_losses.append(policy_loss)
                entropy_losses.append(ent_loss)

                loss = np.mean(Q_losses)
                self.n_updates += 1

            # Update the new state and score
            state = next_state
            score += reward
            self.n_steps += 1
        return score, loss


def run_sac(env):
    seed = 0
    LAYER_SIZE = 256
    BUFFER_SIZE = 50000
    BATCH_SIZE = 128
    GAMMA = 0.99
    TAU = 0.005
    LR = 0.001
    ALPHA = 0.1
    NUMBER_OF_EPISODES = 200
    policy_type = 'Gaussian'
    target_update_interval = 1
    auto_tuning = False
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print("Using ", device)
    np.random.seed(seed)
    torch.manual_seed(seed)
    random.seed(seed)

    action_size = env.action_space.shape[0]
    state_size = env.observation_space.shape[0]

    agent = SAC_Agent(state_size=state_size,
                      action_size=action_size,
                      layer_size=LAYER_SIZE,
                      BATCH_SIZE=BATCH_SIZE,
                      BUFFER_SIZE=BUFFER_SIZE,
                      LR=LR,
                      ALPHA=ALPHA,
                      TAU=TAU,
                      GAMMA=GAMMA,
                      policy_type=policy_type,
                      target_update_interval=target_update_interval,
                      auto_tuning=auto_tuning,
                      device=device)

    scores = []  # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    for i_episode in range(NUMBER_OF_EPISODES):
        # Run a rollout
        score, _ = agent.rollout(env)
        scores_window.append(score)  # save most recent score
```

```
        scores.append(score)  # save most recent score
        if i_episode % 5 == 0:
            print('\rEpisode {} \tTotal reward: {:.2f} \tAverage Score: {:.2f}'.format(i_episode, score,
                                                                                       np.mean(scores_window)))
    plot_score(scores)
```

In [10]:
```
if __name__ == "__main__":
    env = gym.make('Pendulum-v1')
    run_sac(env)
```

```
Using  cpu
Gaussian policy
Episode 0       Total reward: -878.30   Average Score: -878.30
Episode 5       Total reward: -1653.94  Average Score: -1142.26
Episode 10      Total reward: -947.57   Average Score: -1189.62
Episode 15      Total reward: -1.17     Average Score: -931.21
Episode 20      Total reward: -303.70   Average Score: -747.61
Episode 25      Total reward: -340.18   Average Score: -631.10
Episode 30      Total reward: -116.17   Average Score: -552.27
Episode 35      Total reward: -116.89   Average Score: -482.29
Episode 40      Total reward: -124.36   Average Score: -438.01
Episode 45      Total reward: -232.98   Average Score: -406.13
Episode 50      Total reward: -337.98   Average Score: -385.02
Episode 55      Total reward: -117.59   Average Score: -363.42
Episode 60      Total reward: -1.07     Average Score: -341.50
Episode 65      Total reward: -118.94   Average Score: -328.37
Episode 70      Total reward: -128.52   Average Score: -313.90
Episode 75      Total reward: -224.54   Average Score: -302.58
Episode 80      Total reward: -117.78   Average Score: -289.81
Episode 85      Total reward: -121.67   Average Score: -279.91
Episode 90      Total reward: -238.73   Average Score: -271.26
Episode 95      Total reward: -121.14   Average Score: -263.78
Episode 100     Total reward: -224.33   Average Score: -249.21
Episode 105     Total reward: -125.56   Average Score: -194.27
Episode 110     Total reward: -114.07   Average Score: -137.84
Episode 115     Total reward: -118.25   Average Score: -129.04
Episode 120     Total reward: -248.33   Average Score: -131.68
Episode 125     Total reward: -120.53   Average Score: -131.84
Episode 130     Total reward: -2.86     Average Score: -128.47
Episode 135     Total reward: -124.99   Average Score: -134.31
Episode 140     Total reward: -226.59   Average Score: -135.38
Episode 145     Total reward: -1.20     Average Score: -131.86
Episode 150     Total reward: -119.65   Average Score: -128.52
Episode 155     Total reward: -122.29   Average Score: -130.57
Episode 160     Total reward: -0.92     Average Score: -131.84
Episode 165     Total reward: -118.88   Average Score: -129.48
Episode 170     Total reward: -236.65   Average Score: -130.63
Episode 175     Total reward: -123.42   Average Score: -129.71
Episode 180     Total reward: -121.19   Average Score: -134.33
Episode 185     Total reward: -1.42     Average Score: -135.42
Episode 190     Total reward: -230.49   Average Score: -135.21
Episode 195     Total reward: -125.06   Average Score: -132.52
```



Q: Why do we approximate log of standard deviation and not the standard deviation here?

This is to ensure numerical stability and prevent invalid (negative) standard deviation values. It also improves optimization, as the log-space representation helps constrain the network outputs and prevents instability caused by excessively large or small standard deviation values.

Q: When enforcing the action bounds, we use a transformation to bound the selected action. This changes the pdf for the action distribution. Read Appendix C of [2] and verify that the block in Gaussian_policy_network.sample_action denoted by # Enforcing Action Bound actually implements (20-21) in [2]. Why do we have self.epsilon there?

They include noise to improve both the robustness of the solution and to stimulate exploration.

Q: Analyze your result.

The plot shows that the SAC agent improves quickly in the first 25 episodes, with scores rising from around -1500 to more than -250. Then the performance does not get much better. We could probably improve the current performance by adjusting hyperparameters such as learning rate, layer size, batch size, exploration rate and so on.