Twin Delayed DDPG (TD3)

Overview

- The TD3 algorithm is an improvement to DDPG.

- Clipped double Q learning
  - Instead of using one critic network, we use two main critic networks to compute the Q value and also use two target critic networks to compute the target value.
- Delayed policy updates
  - While the critic network parameter is updated at every step of the episode, the actor network parameter is delayed and updated only after two steps of the episode.
- Target policy smoothing
  - We add some noise to the target action in order to reduce the variance of the target value.

Clipped Double Q Learning

- The problem of overestimation and its remedy
  - In general, overestimation occurs when the function used to select an action and the function used to estimate its value is the same.

$$y = r + \gamma \max_{a'} Q_{\theta'}(s', a')$$

$$y = r + \gamma Q_{\theta'}(s', \arg\max_{a'} Q_{\theta'}(s', a'))$$

  - If there is an error in the Q value due to parameter θ', the error goes into the target value as well.
  - In double DQN, we use separate neural networks θ and θ' for selecting an action and calculating the target value.

$$y = r + \gamma Q_{\theta'}(s', \arg\max_{a'} Q_{\theta}(s', a'))$$

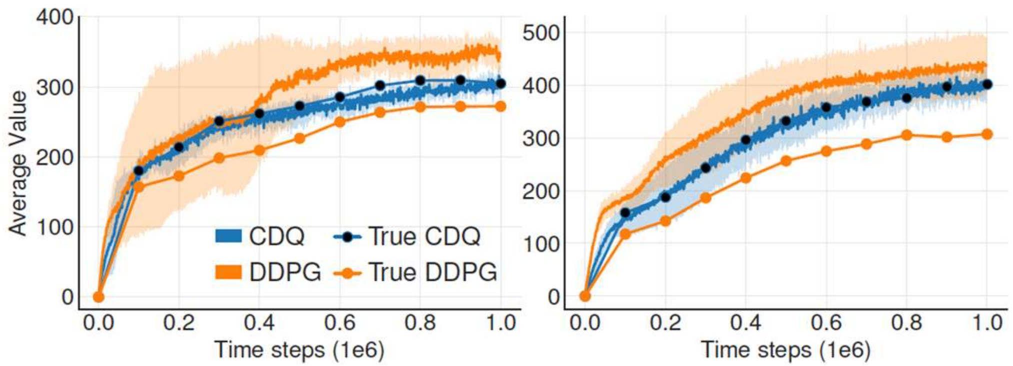
  - Even though θ' is periodically copied from θ and so they are closely related, separating the networks in this way helped reduce the overestimation.

Clipped Double Q Learning

- Overestimation in Actor-Critic networks
  - In the actor-critic method, we choose an action based on the actor network, and compute the target value using the critic network.

$$y = r + \gamma Q_{\theta'}(s', \mu_{\phi'}(s'))$$

  - However, overestimation is observed here as well, primarily due to the fact that the actor network is trained in the direction that will maximize Q value calculated by the critic network.
  - The use of main and target networks do not help much here. The authors claim that it is because in the actor-critic setting, the policies slowly change and thus the difference between main and target networks are small.



(a) Hopper-v1 (b) Walker2d-v1

Clipped Double Q Learning

- Solution: use two critic networks: θ_1 and θ_2
  - We also need two target critic networks: θ_1' and θ_2'

- We compute the Q value of the next state-action pair using two separate target critic networks
  - $Q_{\theta_1'}(s', a')$
  - $Q_{\theta_2'}(s', a')$
  - Here, $a' = \mu_{\phi'}(s')$

- Then, we use the minimum of the two in computing the target value.

$$y = r + \gamma \min_{j=1,2} Q_{\theta_j'}(s', \mu_{\phi'}(s'))$$

- This could result in underestimation, but the authors find that underestimation is better than overestimation.

Clipped Double Q Learning

- Updating the main critic networks
  - To update the main critic network 1, we use the MSE between the target value and the predicted Q value of the main critic network 1.

$$J(\theta_1) = \frac{1}{K}\sum_{i=1}^{K}\left(y_i - Q_{\theta_1}(s_i, a_i)\right)^2$$

  - Similarly, to update the main critic network 2, we use the MSE between the target value and the predicted Q value of the main critic network 2.

$$J(\theta_2) = \frac{1}{K}\sum_{i=1}^{K}\left(y_i - Q_{\theta_2}(s_i, a_i)\right)^2$$

  - We use gradient descent to update the parameters θ_1 and θ_2.

$$\theta_j = \theta_j - \alpha\nabla_{\theta_j}J(\theta_j) \qquad \text{for } j = 1, 2$$

Clipped Double Q Learning

- Updating the target critic networks
  - We use soft replacement to update the target critic networks

$$\theta_j' = \tau\theta_j + (1-\tau)\theta_j' \quad \text{for } j = 1,2$$

Delayed Policy Updates

- In DDPG, actor and critic network parameters are updated at the same time.
  - Possibly in every step of the episode

- In TD3, actor network parameter update is delayed until the critic network parameter is updated several times.
  - The actor network parameter is updated based on feedback given by the critic network: the Q value.
  - When the critic network parameter is not good, it estimates incorrect Q values.
  - If the Q value estimated by the critic network is not correct, the actor network cannot update its parameters correctly.
  - Thus, we delay the update of the actor network so that the critic network can update its parameter several times to do a better estimation of Q values.

- In a typical operation, while the critic network parameters are updated at every step of the episode, the actor network parameters are updated after every 2 steps.

Delayed Policy Updates

- Updating the actor network parameters
  - The objective of the actor network is to maximize the Q value.

$$J(\phi) = \frac{1}{K} \sum_i Q_\theta(s_i, a)$$

  - In TD3, we have two critic networks that compute the Q values.
    - $Q_{\theta_1}(s, a)$
    - $Q_{\theta_2}(s, a)$
  - We can just use any of them, so choose θ_1.

$$J(\phi) = \frac{1}{K} \sum_i Q_{\theta_1}(s_i, a)$$

  - We do gradient ascent to maximize $J(\phi)$.

$$\phi = \phi + \alpha \nabla_\phi J(\phi)$$

Target Policy Smoothing

- In DDPG, we use deterministic policy. because of that, it is possible that the estimated action value may vary greatly for similar actions.
- In order to make "similar actions have similar value", we add a noise to the action when we calculate the target value.
- With clipped double Q learning, we calculate our target value as:

$$y = r + \gamma \min_{j=1,2} Q_{\theta_j'}(s', a')$$

- where $a' = \mu_{\phi'}(s')$
- With target policy smoothing, we modify the action to $\tilde{a}$

$$\tilde{a} = \mu_{\phi'}(s') + \epsilon \; where \; \epsilon \sim (N(0, \sigma), -c, +c)$$

- Here, -c, and +c indicates that the noise is clipped, so that we can keep the target close to the actual action.
- Now we are fitting Q_θ to the "vicinity" of the target action.

Putting It All Together: TD3

- We need six networks
  - Two main critic networks: θ_1 and θ_2
  - Two target critic networks: θ_1' and θ_2'
  - main actor network: ϕ
  - target actor network: ϕ'

- We initialize the two main critic networks θ_1 and θ_2 and the main actor networks ϕ with random values.
- We initialize the two target networks θ_1' and θ_2' and the main actor networks ϕ' by copying parameters from θ_1, θ_2, and ϕ.

- We initialize the replay buffer $\mathcal{D}$.

Putting It All Together: TD3

- Now we start running episodes.
- For each step of the episode, we first select an action a using the actor network.

$$a = \mu_\phi(s)$$

- To ensure exploration, we add some noise ϵ to the action, where $\epsilon \sim \mathcal{N}(0, \delta)$.

$$a = \mu_\phi(s) + \epsilon$$

- Then, we perform the action a, move to the next state s', and get reward r. We store this transition information in a replay buffer $\mathcal{D}$.

Putting It All Together: TD3

- Next, we randomly sample a minibatch of K transitions (s, a, r, s') from the replay buffer. They are used to update the critic and the actor network.

- First, we compute the loss of the critic network.

$$J(\theta_j) = \frac{1}{K}\sum_i \left(y_i - Q_{\theta_j}(s_i, a_i)\right)^2 \quad \text{for } j = 1, 2$$

- In the equation, the target value y_i is:

$$y = r + \gamma \min_{j=1,2} Q_{\theta_j'}(s', a')$$

$$\tilde{a} = \mu_{\phi'}(s') + \epsilon \; where \; \epsilon \sim (N(0, \sigma), -c, +c)$$

- We use gradient descent to update the parameter

$$\theta_j = \theta_j - \alpha \nabla_{\theta_j} J(\theta_j) \quad \text{for } j = 1, 2$$

Putting It All Together: TD3

- Now we update the actor network

$$J(\phi) = \frac{1}{K}\sum_i Q_\theta(s_i, a)$$

- In the equation, a does not come from the mini-batch transition, but is produced by the actor network, $a = \mu_\phi(s_i)$.

- In order to maximize the objective function, we use gradient ascent.

$$\phi = \phi + \alpha \nabla_\phi J(\phi)$$

- Instead of doing parameter update at every time step, we delay the updates so that the parameters are updated after every d steps.

1. If t mod d = 0, then:
    1. Compute the gradient of the objective function $\nabla_\phi J(\phi)$
    2. Update the actor network parameter using gradient ascent $\phi = \phi + \alpha \nabla_\phi J(\phi)$

Putting It All Together: TD3

- Finally, we update the parameters of the target critic networks θ'_1 and θ'_2, as well as the target actor network ϕ' by soft replacement.

$$\theta'_j = \tau\theta_j + (1-\tau)\theta'_j \quad \text{for } j = 1, 2$$

$$\phi' = \tau\phi_j + (1-\tau)\phi'$$

- This soft replacement is also delayed, so that the target networks are updated after every d steps.

1. If t mod d = 0, then:
   1. Compute the gradient of the objective function $\nabla_\phi J(\phi)$ and update the actor network parameter using gradient ascent $\phi = \phi + \alpha\nabla_\phi J(\phi)$
   2. Update the target critic network parameter and target actor network parameter as $\theta'_j = \tau\theta_j + (1-\tau)\theta'_j \quad$ for $j = 1{,}2$, and $\phi' = \tau\phi + (1-\tau)\phi'$, respectively

Solving 'Pendulum-v0' with TD3

- libraries and global variables

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import gym
import numpy as np
import matplotlib.pyplot as plt

#==============================================================================
# global variables
#==============================================================================
device = 'cuda' if torch.cuda.is_available() else 'cpu'
env = gym.make('Pendulum-v0')
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
max_action = float(env.action_space.high[0])

capacity = 50000
batch_size = 128
gamma = 0.99
tau = 0.005
exploration_noise = 0.1
policy_noise = 0.2
num_episodes = 1000
```

Solving 'Pendulum-v0' with TD3

- replay buffer definition

```
class Replay_buffer():

    def __init__(self, max_size=capacity):
        self.storage = []
        self.max_size = max_size
        self.ptr = 0

    def push(self, data):
        if len(self.storage) == self.max_size:
            self.storage[int(self.ptr)] = data
            self.ptr = (self.ptr + 1) % self.max_size
        else:
            self.storage.append(data)

    def sample(self, batch_size):
        ind = np.random.randint(0, len(self.storage), size=batch_size)
        x, y, u, r, d = [], [], [], [], []

        for i in ind:
            X, Y, U, R, D = self.storage[i]
            x.append(np.array(X, copy=False))
            y.append(np.array(Y, copy=False))
            u.append(np.array(U, copy=False))
            r.append(np.array(R, copy=False))
            d.append(np.array(D, copy=False))

        return np.array(x), np.array(y), np.array(u), np.array(r).reshape(-1, 1), np.array(d).reshape(-1, 1)
```

Solving 'Pendulum-v0' with TD3

- actor

```
class Actor(nn.Module):

    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()

        self.fc1 = nn.Linear(state_dim, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, action_dim)

        self.max_action = max_action

    def forward(self, state):
        a = F.relu(self.fc1(state))
        a = F.relu(self.fc2(a))
        a = torch.tanh(self.fc3(a)) * self.max_action
        return a
```

Solving 'Pendulum-v0' with TD3

- critic

```python
class Critic(nn.Module):

    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()

        self.fc1 = nn.Linear(state_dim + action_dim, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, 1)

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

        q = F.relu(self.fc1(state_action))
        q = F.relu(self.fc2(q))
        q = self.fc3(q)
        return q
```

Solving 'Pendulum-v0' with TD3

- TD3 agent definition
  - Create and initialize two main critic networks, two target critic networks, one main actor network and one target actor network.
  - We also define the optimizers and initialize the replay buffer

```
class TD3():
    def __init__(self, state_dim, action_dim, max_action):

        self.actor = Actor(state_dim, action_dim, max_action).to(device)
        self.actor_target = Actor(state_dim, action_dim, max_action).to(device)
        self.critic_1 = Critic(state_dim, action_dim).to(device)
        self.critic_1_target = Critic(state_dim, action_dim).to(device)
        self.critic_2 = Critic(state_dim, action_dim).to(device)
        self.critic_2_target = Critic(state_dim, action_dim).to(device)

        self.actor_optimizer = optim.Adam(self.actor.parameters())
        self.critic_1_optimizer = optim.Adam(self.critic_1.parameters())
        self.critic_2_optimizer = optim.Adam(self.critic_2.parameters())

        self.actor_target.load_state_dict(self.actor.state_dict())
        self.critic_1_target.load_state_dict(self.critic_1.state_dict())
        self.critic_2_target.load_state_dict(self.critic_2.state_dict())

        self.max_action = max_action
        self.memory = Replay_buffer()
```

Solving 'Pendulum-v0' with TD3

- TD3 agent definition
  - The select_action function passes the given state to the main actor network and returns the action.

```
def select_action(self, state):
    state = torch.tensor(state.reshape(1, -1)).float().to(device)
    return self.actor(state).cpu().data.numpy().flatten()
```

Solving 'Pendulum-v0' with TD3

- TD3 agent definition
  - We sample mini-batch from the replay buffer
  - We select the next action using the target actor network.

```
#==========================================================================
# TD3 update rule
#==========================================================================
def update(self, num_iteration):

    for i in range(num_iteration):

        # sample mini-batch from the replay buffer
        x, y, u, r, d = self.memory.sample(batch_size)
        state = torch.FloatTensor(x).to(device)
        action = torch.FloatTensor(u).to(device)
        next_state = torch.FloatTensor(y).to(device)
        done = torch.FloatTensor(d).to(device)
        reward = torch.FloatTensor(r).to(device)

        # select next action according to the target policy
        noise = torch.ones_like(action).data.normal_(0, policy_noise).to(device)
        noise = noise.clamp(-0.5, 0.5)
        next_action = (self.actor_target(next_state) + noise)
        next_action = next_action.clamp(-self.max_action, self.max_action)
```

Solving 'Pendulum-v0' with TD3

- TD3 agent definition
  - We compute the target Q value
  - We get the Q value from two critic networks and use the minimum of the two in calculating the target value.

```
# compute target Q-value
target_Q1 = self.critic_1_target(next_state, next_action)
target_Q2 = self.critic_2_target(next_state, next_action)
target_Q = torch.min(target_Q1, target_Q2)
target_Q = reward + ((1 - done) * gamma * target_Q).detach()
```

Solving 'Pendulum-v0' with TD3

- TD3 agent definition
  - We update critic network 1 and critic network 2

```
# update critic 1
current_Q1 = self.critic_1(state, action)
loss_Q1 = F.mse_loss(current_Q1, target_Q)
self.critic_1_optimizer.zero_grad()
loss_Q1.backward()
self.critic_1_optimizer.step()

# update critic 2
current_Q2 = self.critic_2(state, action)
loss_Q2 = F.mse_loss(current_Q2, target_Q)
self.critic_2_optimizer.zero_grad()
loss_Q2.backward()
self.critic_2_optimizer.step()
```

Solving 'Pendulum-v0' with TD3

- TD3 agent definition
  - Delayed policy updates and soft replacements
    - We update the actor network
    - We update the target critic networks and the target actor network

```
# delayed policy updates
if i % 2 == 0:

    # compute actor loss
    actor_loss = -self.critic_1(state, self.actor(state)).mean()

    # update actor
    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()

    # soft replacement
    for param, target_param in zip(self.critic_1.parameters(), self.critic_1_target.parameters()):
        target_param.data.copy_(((1 - tau) * target_param.data) + tau * param.data)

    for param, target_param in zip(self.critic_2.parameters(), self.critic_2_target.parameters()):
        target_param.data.copy_(((1 - tau) * target_param.data) + tau * param.data)

    for param, target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
        target_param.data.copy_(((1 - tau) * target_param.data) + tau * param.data)
```

Solving 'Pendulum-v0' with TD3

- main function (1/2)
  - Run episodes to collect transitions
  - Once the replay buffer is filled up, we start updating the parameters

```
def main():

    agent = TD3(state_dim, action_dim, max_action)
    ep_r = 0
    ep_r_store = []

    for i in range(num_episodes):
        state = env.reset()

        for t in range(200):
            action = agent.select_action(state)
            action = action + np.random.normal(0, exploration_noise, size=action_dim)
            action = action.clip(env.action_space.low, env.action_space.high)

            # perform action and obtain transition info
            next_state, reward, done, _ = env.step(action)
            ep_r += reward

            # add transition to replay buffer
            agent.memory.push((state, next_state, action, reward, float(done)))
```

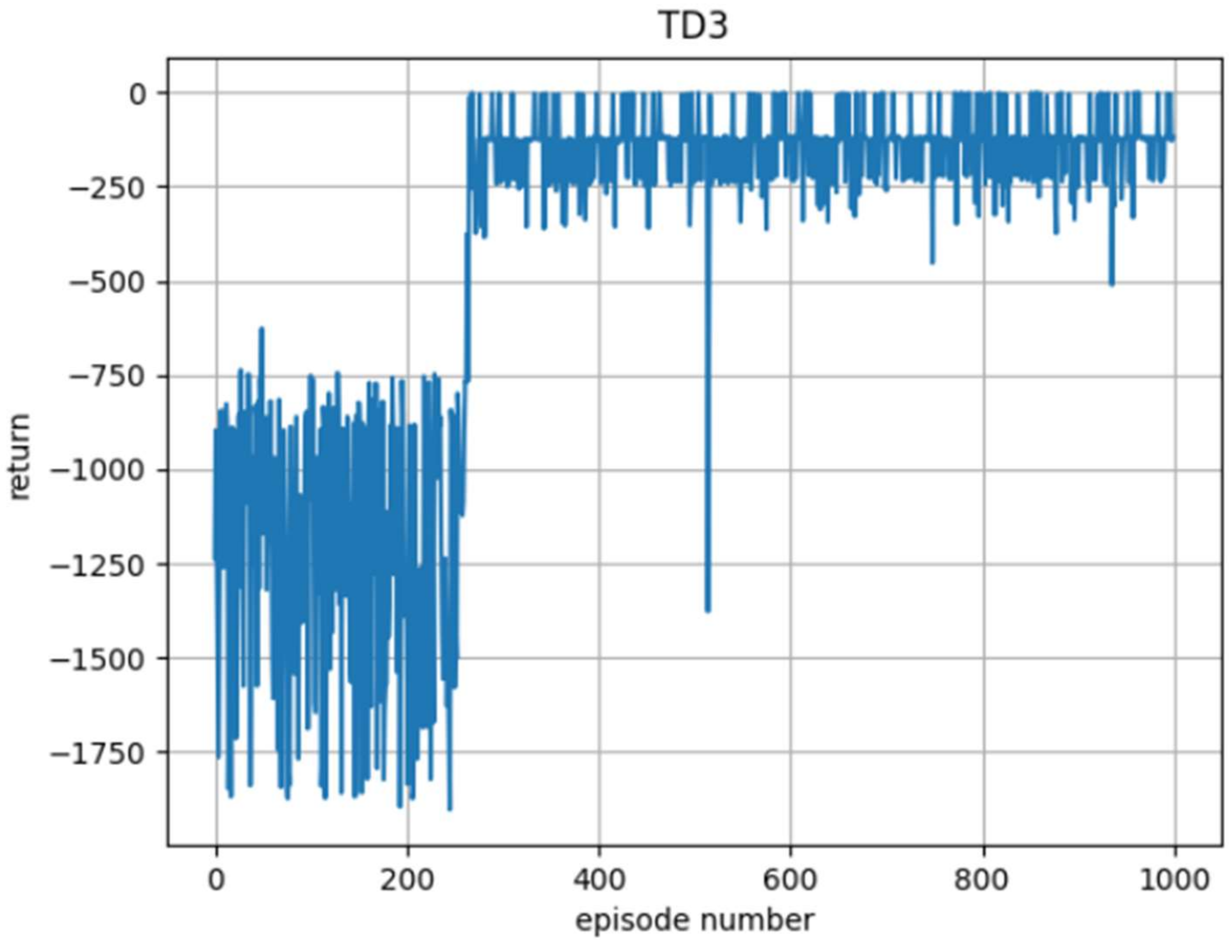
Solving 'Pendulum-v0' with TD3

- main function (2/2)
  - Run episodes to collect transitions
  - Once the replay buffer is filled up, we start updating the parameters

```
            # start updating networks when the replay buffer is full
            if len(agent.memory.storage) >= capacity-1:
                agent.update(10)

            state = next_state
            if done or t == 199:
                print("Episode %5d: return is %.2f"%(i, ep_r))
                ep_r_store.append(ep_r)
                ep_r = 0
                break

    plt.plot(ep_r_store)
    plt.title('TD3')
    plt.xlabel('episode number')
    plt.ylabel('return')
    plt.grid(True)
    plt.savefig("td3.png")


if __name__ == '__main__':
    main()
```

Solving 'Pendulum-v0' with TD3

- result
  - The agent trained to solve 'Pendulum-v0'.



Soft Actor Critic (SAC)

Overview

- The Soft Actor-Critic (SAC) method is a variation of the actor-critic methods.
- SAC optimizes a stochastic policy in an off-policy way.
  - A bridge between stochastic policy optimization and DDPG-style approaches
- SAC uses the concept of entropy.
  - Entropy is a measure of randomness of a variable

Entropy

- Entropy of a random variable X is the average level of "information", "surprise", or "uncertainty" inherent in the variable's possible outcomes.

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log P(x_i)$$

  – The choice of base for log varies for different applications.

- For example, entropy of a dice is:
  – $H(X) = -6 \times \left(\frac{1}{6} \log \frac{1}{6}\right) = \log 6$

- Suppose we have a dice that always rolls 3. Then, its entropy would be
  – $H(X) = -1 \log 1 = 0$
  – meaning that it is not random at all.

Including Entropy in the Objective Function

- In reinforcement learning, our goal is to maximize the return

$$J(\phi) = E_{\tau \sim \pi_\phi}\left[\sum_{t=0}^{T-1} r_t\right]$$

- In Soft Actor-Critic, we add an entropy term to the objective function.

$$J(\phi) = E_{\tau \sim \pi_\phi}\left[\sum_{t=0}^{T-1} r_t + \alpha \mathcal{H}\big(\pi(\cdot\,|s_t)\big)\right]$$

- By adding the entropy term, our goal becomes maximizing the return and also maximizing the entropy.
- The parameter α is called temperature, which controls the importance of entropy in our objective function.
  - A high α promotes more exploration.
- This objective function is often referred to as maximum entropy reinforcement learning, or entropy regularized reinforcement learning.

Soft Actor-Critic

- Similar to other actor-critic methods, the actor uses the policy gradient to find the optimal policy, and the critic evaluates the policy.

- In SAC, the critic uses both the Q function and the value function to evaluate actor's policy. So the critic consists of two networks: a value network and a Q network.

Value Function with Entropy Term

- The value of a state is the expected return of the trajectory starting from state s following a policy π.

$$V^{\pi}(s) = E_{\tau\sim\pi}\left[\sum_{t=0}^{T-1} r_t \mid s_0 = s\right]$$

- In SAC, we add the entropy term to the value function.
- In every time step, we get an entropy bonus in addition to the reward.

$$V^{\pi}(s) = E_{\tau\sim\pi}\left[\sum_{t=0}^{T-1} \left(r_t + \alpha\mathcal{H}\big(\pi(\cdot\,|s_t)\big)\right) \mid s_0 = s\right]$$

Q Function with Entropy Term

- The Q value of a state-action pair is the expected return of the trajectory starting from state s and action a following a policy π.

$$Q^{\pi}(s) = E_{\tau\sim\pi}\left[\sum_{t=0}^{T-1} r_t \mid s_0 = s, a_0 = a\right]$$

- In SAC, we add the entropy term to the Q function.
  - The entropy bonus is not added to the case where $t = 0$, because the action there is already determined.

$$Q^{\pi}(s,a) = E_{\tau\sim\pi}\left[\sum_{t=0}^{T-1} r_t + \alpha \sum_{t=1}^{T-1} \mathcal{H}\big(\pi(\cdot\,|s_t)\big) \mid s_0 = s, a_0 = a\right]$$

The Soft State-Value Function

- From the equation of V and Q, we can get the relation between the two.

$$V^{\pi}(s) = E_{\tau\sim\pi}[Q^{\pi}(s,a)] + \alpha\mathcal{H}\big(\pi(a|s)\big)$$

- Since the entropy is defined like this,

$$\mathcal{H}(X) = -\sum_{i=1}^{n} P(x_i)\log P(x_i)$$

- We can write the equation as:

$$V^{\pi}(s) = E_{\tau\sim\pi}[Q^{\pi}(s,a) - \alpha\log\pi(a|s)]$$

- This equation is called the soft state-value function.

Components of SAC: Critic - Value Network

- We have a main value network ψ and a target value network ψ'.
- To train the main value network, we will minimize the loss between the target state value and the predicted state value.
- Value of a state is computed as

$$V^{\pi}(s) = E_{\tau\sim\pi}[Q^{\pi}(s,a) - \alpha \log \pi(a|s)]$$

- Since we are going to approximate the expectation by sampling K transitions from the replay buffer, we remove the expectation.

$$y_v = Q(s,a) - \alpha \log \pi(a|s)$$

- We need a Q value to calculate the target value. For that we are going to use the Q network θ. Also the policy π comes from the actor network ϕ.

$$y_v = Q_{\theta}(s,a) - \alpha \log \pi_{\phi}(a|s)$$

Components of SAC: Critic - Value Network

- In order to mitigate overestimation, we use clipped double Q learning, also used in TD3. For that, we calculate two Q values using two Q networks and take the minimum of the two.

$$y_v = \min_{j=1,2} Q_{\theta_j}(s,a) - \alpha \log \pi_\phi(a|s)$$

  – One difference here is that we use the two main Q networks, not the target Q networks. In TD3, the target Q networks were used in computing the next state-action pair $Q(s', a')$.

- Our loss function for the value network is:

$$J_V(\psi) = \frac{1}{K}\sum_i \left(y_{v_i} - V_\psi(s_i)\right)^2$$

Components of SAC: Critic - Value Network

- We use gradient descent to update the value network parameters.

$$\psi = \psi - \lambda \nabla_{\psi} J(\psi)$$

- For the target value network, we use soft replacement to update the parameters.

$$\psi' = \tau\psi + (1 - \tau)\psi'$$

Components of SAC: Critic - Q Network

- In SAC, we do not need a target Q network. We only need the main Q network, θ.
- To train the Q network, we will minimize the loss between the target Q value and the predicted Q value.
- According to Bellman equation, Q(s,a) can be calculated as:

$$Q(s,a) = E_{s' \sim P}[r + \gamma V(s')]$$

- Since we are going to approximate the expectation by sampling K transitions from the replay buffer, we remove the expectation.

$$y_q = r + \gamma V(s')$$

- To compute the value of the next state $V(s')$, we use the target value network ψ'.

$$y_q = r + \gamma V_{\psi'}(s')$$

Components of SAC: Critic - Q Network

- We calculate the loss function of the Q network.

$$J_Q(\theta) = \frac{1}{K}\sum_i \left(y_{q_i} - Q_\theta(s_i, a_i)\right)^2$$

- We used clipped double Q learning for calculating the target value. So we need to update parameters of θ_1 and θ_2 separately.

$$J_Q(\theta_j) = \frac{1}{K}\sum_i \left(y_{q_i} - Q_{\theta_j}(s_i, a_i)\right)^2 \text{ for } j = 1, 2$$

- We update the parameters of the Q network using gradient descent.

$$\theta_j = \theta_j - \lambda \nabla_{\theta_j} J(\theta_j) \text{ for } j = 1, 2$$

Components of SAC: Actor Network

- In TD3, the objective function of the actor network was the following.

$$J(\phi) = \frac{1}{K}\sum_i Q_\theta(s_i, a)$$

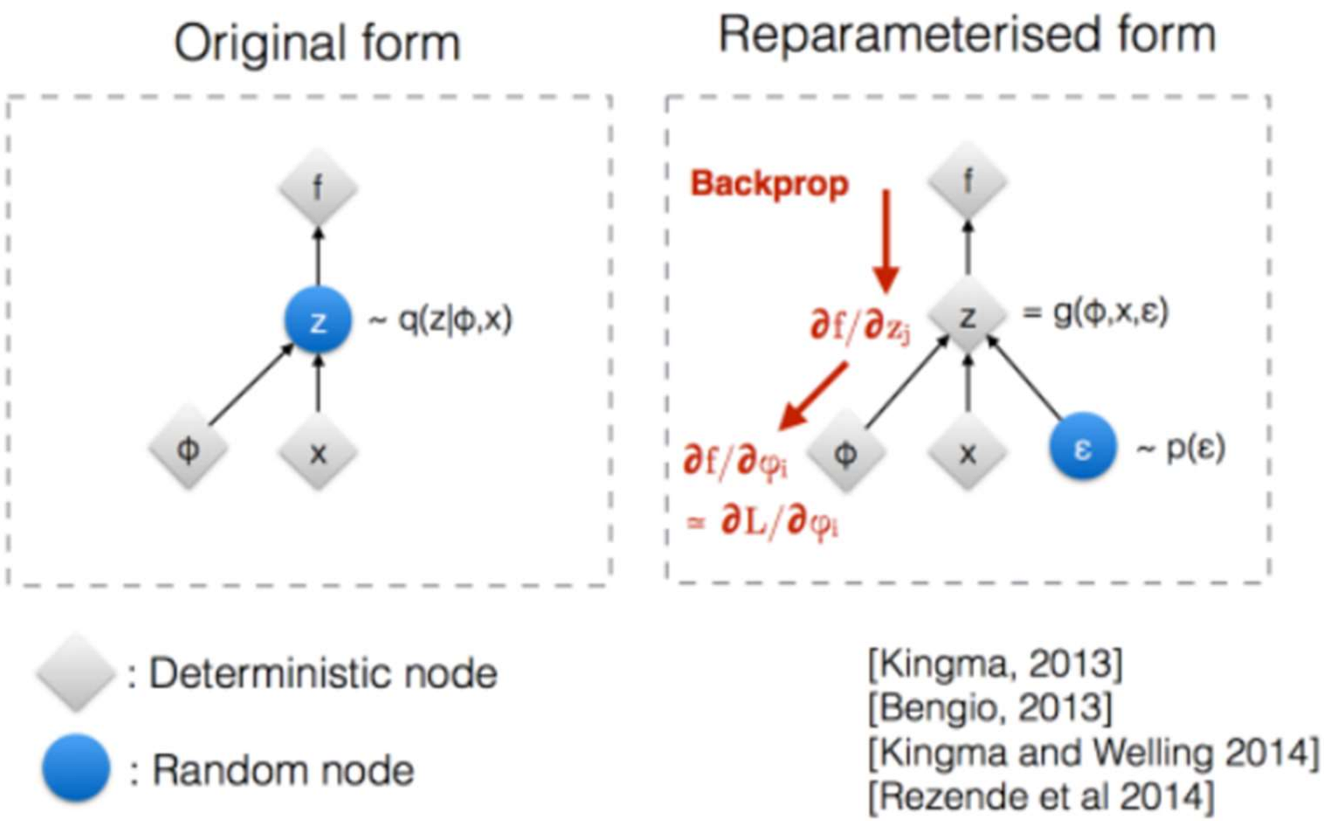
- The goal of the actor is to generate action in such a way that it maximizes the Q value computed by the critic.

- The objective of the actor network in SAC is the same, except that here we use a stochastic policy $\pi_{\phi(a|s)}$, and also, we maximize the entropy.

$$J(\phi) = \frac{1}{K}\sum_i \left[Q_\theta(s_i, a) - \alpha \log \pi_\phi(a|s_i)\right]$$

Components of SAC: Actor Network

- In order to compute the derivative of the objective function, we use what is called the reparameterization trick.
  - Instead of sampling from N(mean, std), we change it to mean + std * N(0, 1) to enable backpropagation.



Putting It All Together: SAC

- In SAC, we use five networks.
  - The main value network ψ
  - The target value network ψ'
  - The two main Q networks θ_1, θ_2
  - The actor network (policy network) ϕ

- First, we initialize the networks
  - the initial parameters of ψ' is copied from ψ

- We also initialize the replay buffer

Putting It All Together: SAC

- Now we run the episodes.
- For each step in the episode, we select an action a using the actor network.

$$a = \pi_{\phi}(s)$$

- We perform the action a, move to the next state s', and get the reward r.
- We store this transition information in the replay buffer $\mathcal{D}$.

- Next, we randomly sample a minibatch of K transitions from the replay buffer. These K transitions (s, a, r, s') are used for updating our value, Q, and the actor network.

Putting It All Together: SAC

- First we compute the loss of the value network.
- The loss function is:

$$J_V(\psi) = \frac{1}{K}\sum_i \left(y_{v_i} - V_\psi(s_i)\right)^2$$

- where the target state value y_{v_i} is:

$$y_{v_i} = \min_{j=1,2} Q_{\theta_j}(s_i, a_i) - \alpha \log \pi_\phi(a_i|s_i)$$

- We calculate the gradients and update the parameter ψ using gradient descent.

$$\psi = \psi - \lambda \nabla_\psi J(\psi)$$

Putting It All Together: SAC

- Next we compute the loss of the Q networks.

$$J_Q(\theta_j) = \frac{1}{K}\sum_i \left(y_{q_i} - Q_{\theta_j}(s_i, a_i)\right)^2 \quad \text{for } j = 1, 2$$

- where the target Q value y_{q_i} is:

$$y_{q_i} = r_i + \gamma V_{\psi'}(s_i')$$

- We calculate the gradients and update the parameters θ_1 and θ_2 using gradient descent.

$$\theta_j = \theta_j - \lambda \nabla_{\theta_j} J(\theta_j) \quad \text{for } j = 1, 2$$

Putting It All Together: SAC

- Next we update the actor network. The objective function of the actor network is:

$$J_\pi(\phi) = \frac{1}{K}\sum_i \left[Q_{\theta_1}(s_i, a) - \alpha \log\pi_\phi(a|s_i)\right]$$

- We calculate the gradients and update parameter ϕ using gradient ascent.

$$\phi = \phi + \lambda \nabla_\phi J(\phi)$$

- Finally, we update the target value network using soft replacement.

$$\psi' = \tau\psi + (1-\tau)\psi'$$

Solving 'Pendulum-v0' with SAC

- Imports and global variables

```
# https://github.com/pranz24/pytorch-soft-actor-critic/tree/SAC_V

import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Adam
from torch.distributions import Normal
import numpy as np
import random

#==============================================================================
# global variables
#==============================================================================
seed = 1
gamma = 0.99
tau = 0.005
alpha = 0.2
lr = 0.0003
hidden_size = 256
epsilon = 1e-6
replay_size = 1000000
start_steps = 10000
updates_per_step = 1
batch_size = 256
num_steps = 1000000
```

Solving 'Pendulum-v0' with SAC

- weights_init_: a function for initializing parameters of a neural network.

```
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)
```

- hard_update: copies parameters from one neural network to another.

```
def hard_update(target, source):
    for target_param, param in zip(target.parameters(), source.parameters()):
        target_param.data.copy_(param.data)
```

- soft_update: copies parameters from one neural network to another using soft replacement.

```
def soft_update(target, source, tau):
    for target_param, param in zip(target.parameters(), source.parameters()):
        target_param.data.copy_(target_param.data * (1.0 - tau) + param.data * tau)
```

Solving 'Pendulum-v0' with SAC

- replay buffer

```
class ReplayMemory:
    def __init__(self, capacity, seed):
        random.seed(seed)
        self.capacity = capacity
        self.buffer = []
        self.position = 0

    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = map(np.stack, zip(*batch))
        return state, action, reward, next_state, done

    def __len__(self):
        return len(self.buffer)
```

Solving 'Pendulum-v0' with SAC

- Q network
  - Two branches of θ_1 and θ_2 are both included in one network.

```
class QNetwork(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_dim):
        super(QNetwork, self).__init__()

        # Q1 architecture
        self.linear1 = nn.Linear(num_inputs + num_actions, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, 1)

        # Q2 architecture
        self.linear4 = nn.Linear(num_inputs + num_actions, hidden_dim)
        self.linear5 = nn.Linear(hidden_dim, hidden_dim)
        self.linear6 = nn.Linear(hidden_dim, 1)

        self.apply(weights_init_)

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

        x1 = F.relu(self.linear1(xu))
        x1 = F.relu(self.linear2(x1))
        x1 = self.linear3(x1)

        x2 = F.relu(self.linear4(xu))
        x2 = F.relu(self.linear5(x2))
        x2 = self.linear6(x2)

        return x1, x2
```

Solving 'Pendulum-v0' with SAC

- Value network

```
class ValueNetwork(nn.Module):
    def __init__(self, num_inputs, hidden_dim):
        super(ValueNetwork, self).__init__()

        self.linear1 = nn.Linear(num_inputs, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, 1)

        self.apply(weights_init_)

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

Solving 'Pendulum-v0' with SAC

- Actor network
  - In SAC, we use a stochastic policy in a continuous action space.
  - The actor network produces a mean and a standard deviation for each action.
  - The action will be sampled from a Gaussian distribution with given mean and std.
  - Instead of computing std, we let the network compute log std and convert it to std afterwards.

```
class GaussianPolicy(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_dim):
        super(GaussianPolicy, self).__init__()

        self.linear1 = nn.Linear(num_inputs, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)

        self.mean_linear = nn.Linear(hidden_dim, num_actions)
        self.log_std_linear = nn.Linear(hidden_dim, num_actions)

        self.apply(weights_init_)
```

Solving 'Pendulum-v0' with SAC

- Actor network
  - Given a state, the actor network produces mean and log_std.
  - log_std is clamped so that it does not produce a very large or a very small std.

```
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, min=-20, max=2)
    return mean, log_std
```

Solving 'Pendulum-v0' with SAC

- Actor network
  - The function sample samples an action using mean and log_std.
  - First, it restores std from log_std.
  - Then, a value (x_t) is sampled from the Gaussian distribution with mean and std.
  - Using normal.rsample() instead of normal.sample() does the reparameterization trick. It calculates mean + std * N(0,1) instead of sampling from the Gaussian distribution.

```python
def sample(self, state):
    mean, log_std = self.forward(state)
    std = log_std.exp()
    normal = Normal(mean, std)
    x_t = normal.rsample()  # for reparameterization trick (mean + std * N(0,1))
    action = torch.tanh(x_t)
    log_prob = normal.log_prob(x_t)
    # Enforcing Action Bound
    log_prob -= torch.log(1 - action.pow(2) + epsilon)
    log_prob = log_prob.sum(1, keepdim=True)
    return action, log_prob, mean, log_std
```

Solving 'Pendulum-v0' with SAC

- Actor network
  - We enforce the action so that it fits in the range of an action.
  - In order to do that we apply tanh to fit the action in the range [-1, 1].
  - Because of that, we change the log probability accordingly.

C. Enforcing Action Bounds

We use an unbounded Gaussian as the action distribution. However, in practice, the actions needs to be bounded to a finite interval. To that end, we apply an invertible squashing function ($\tanh$) to the Gaussian samples, and employ the change of variables formula to compute the likelihoods of the bounded actions. In the other words, let $\mathbf{u} \in \mathbb{R}^D$ be a random variable and $\mu(\mathbf{u}|\mathbf{s})$ the corresponding density with infinite support. Then $\mathbf{a} = \tanh(\mathbf{u})$, where $\tanh$ is applied elementwise, is a random variable with support in $(-1, 1)$ with a density given by

$$\pi(\mathbf{a}|\mathbf{s}) = \mu(\mathbf{u}|\mathbf{s}) \left| \det\left(\frac{\mathrm{d}\mathbf{a}}{\mathrm{d}\mathbf{u}}\right) \right|^{-1}. \tag{20}$$

Since the Jacobian $\mathrm{d}\mathbf{a}/\mathrm{d}\mathbf{u} = \operatorname{diag}(1 - \tanh^2(\mathbf{u}))$ is diagonal, the log-likelihood has a simple form

$$\boxed{\log \pi(\mathbf{a}|\mathbf{s}) = \log \mu(\mathbf{u}|\mathbf{s}) - \sum_{i=1}^{D} \log\left(1 - \tanh^2(u_i)\right),} \tag{21}$$

where u_i is the i^{th} element of $\mathbf{u}$.

Solving 'Pendulum-v0' with SAC

- The SAC agent
  - Initialize parameters of the neural networks

```
class SAC(object):
    def __init__(self, num_inputs, action_space):
        self.gamma = gamma
        self.tau = tau
        self.alpha = alpha
        self.action_range = [action_space.low, action_space.high]
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'

        self.critic = QNetwork(num_inputs, action_space.shape[0], hidden_size).to(device=self.device)
        self.critic_optim = Adam(self.critic.parameters(), lr=lr)

        self.value = ValueNetwork(num_inputs, hidden_size).to(device=self.device)
        self.value_target = ValueNetwork(num_inputs, hidden_size).to(self.device)
        self.value_optim = Adam(self.value.parameters(), lr=lr)
        hard_update(self.value_target, self.value)

        self.policy = GaussianPolicy(num_inputs, action_space.shape[0], hidden_size).to(self.device)
        self.policy_optim = Adam(self.policy.parameters(), lr=lr)
```

Solving 'Pendulum-v0' with SAC

- The SAC agent
  - select_action samples an action from the policy network and rescales the action to fit the given range of actions.

```
def select_action(self, state):
    state = torch.FloatTensor(state).to(self.device).unsqueeze(0)
    action, _, _, _ = self.policy.sample(state)
    action = action.detach().cpu().numpy()[0]
    return self.rescale_action(action)

def rescale_action(self, action):
    return action * (self.action_range[1] - self.action_range[0]) / 2.0 +\
            (self.action_range[1] + self.action_range[0]) / 2.0
```

Solving 'Pendulum-v0' with SAC

- Updating parameters
  - We first get a mini-batch from the replay buffer.

```
def update_parameters(self, memory, batch_size, updates):
    # Sample a batch from memory
    state_batch, action_batch, reward_batch, next_state_batch, mask_batch = memory.sample(batch_size=batch_size)

    state_batch = torch.FloatTensor(state_batch).to(self.device)
    next_state_batch = torch.FloatTensor(next_state_batch).to(self.device)
    action_batch = torch.FloatTensor(action_batch).to(self.device)
    reward_batch = torch.FloatTensor(reward_batch).to(self.device).unsqueeze(1)
    mask_batch = torch.FloatTensor(mask_batch).to(self.device).unsqueeze(1)
```

Solving 'Pendulum-v0' with SAC

- Updating parameters
  - We compute the loss function of the Q network and update its parameters.

```
with torch.no_grad():
    vf_next_target = self.value_target(next_state_batch)
    next_q_value = reward_batch + mask_batch * self.gamma * (vf_next_target)

qf1, qf2 = self.critic(state_batch, action_batch)
qf1_loss = F.mse_loss(qf1, next_q_value)
qf2_loss = F.mse_loss(qf2, next_q_value)
qf_loss = qf1_loss + qf2_loss

self.critic_optim.zero_grad()
qf_loss.backward()
self.critic_optim.step()
```

Solving 'Pendulum-v0' with SAC

- Updating parameters
  - We compute the loss function of the actor network and update its parameters.
  - Optionally, we may use a regularization term in the loss. In that case, we prefer to have small mean and log_std.

```
pi, log_pi, mean, log_std = self.policy.sample(state_batch)

qf1_pi, qf2_pi = self.critic(state_batch, pi)
min_qf_pi = torch.min(qf1_pi, qf2_pi)

policy_loss = ((self.alpha * log_pi) - min_qf_pi).mean()

# Regularization Loss (optional)
reg_loss = 0.001 * (mean.pow(2).mean() + log_std.pow(2).mean())
policy_loss += reg_loss

self.policy_optim.zero_grad()
policy_loss.backward()
self.policy_optim.step()
```

Solving 'Pendulum-v0' with SAC

- Updating parameters
  - We compute the loss function of the value network and update its parameters.
  - Using soft replacement, we update the target value network.

```
vf = self.value(state_batch)

with torch.no_grad():
    vf_target = min_qf_pi - (self.alpha * log_pi)

vf_loss = F.mse_loss(vf, vf_target)

self.value_optim.zero_grad()
vf_loss.backward()
self.value_optim.step()

soft_update(self.value_target, self.value, self.tau)

return vf_loss.item(), qf1_loss.item(), qf2_loss.item(), policy_loss.item()
```

Solving 'Pendulum-v0' with SAC

- The main function
  - Preparation stage
  - We create a SAC agent and prepare a replay buffer.

```python
def main():
    env = gym.make('Pendulum-v0')

    env.seed(seed)
    env.action_space.seed(seed)
    torch.manual_seed(seed)
    np.random.seed(seed)

    agent = SAC(env.observation_space.shape[0], env.action_space)
    memory = ReplayMemory(replay_size, seed)

    # Training Loop
    total_numsteps = 0
    updates = 0
```

Solving 'Pendulum-v0' with SAC

- The main function
  - Run episodes
  - If there are enough transitions to obtain a batch, we start updating network parameters.

```
for i_episode in range(1000):
    episode_reward = 0
    episode_steps = 0
    done = False
    state = env.reset()

    while not done:
        if start_steps > total_numsteps:
            action = env.action_space.sample()
        else:
            action = agent.select_action(state)  # Sample action from policy

        if len(memory) > batch_size:
            for i in range(updates_per_step):  # Number of updates per step in environment
                # Update parameters of all the networks
                value_loss, critic_1_loss, critic_2_loss, policy_loss = agent.update_parameters(memory, batch_size, updates)
                updates += 1

        next_state, reward, done, _ = env.step(action) # Step
        episode_steps += 1
        total_numsteps += 1
        episode_reward += reward
```

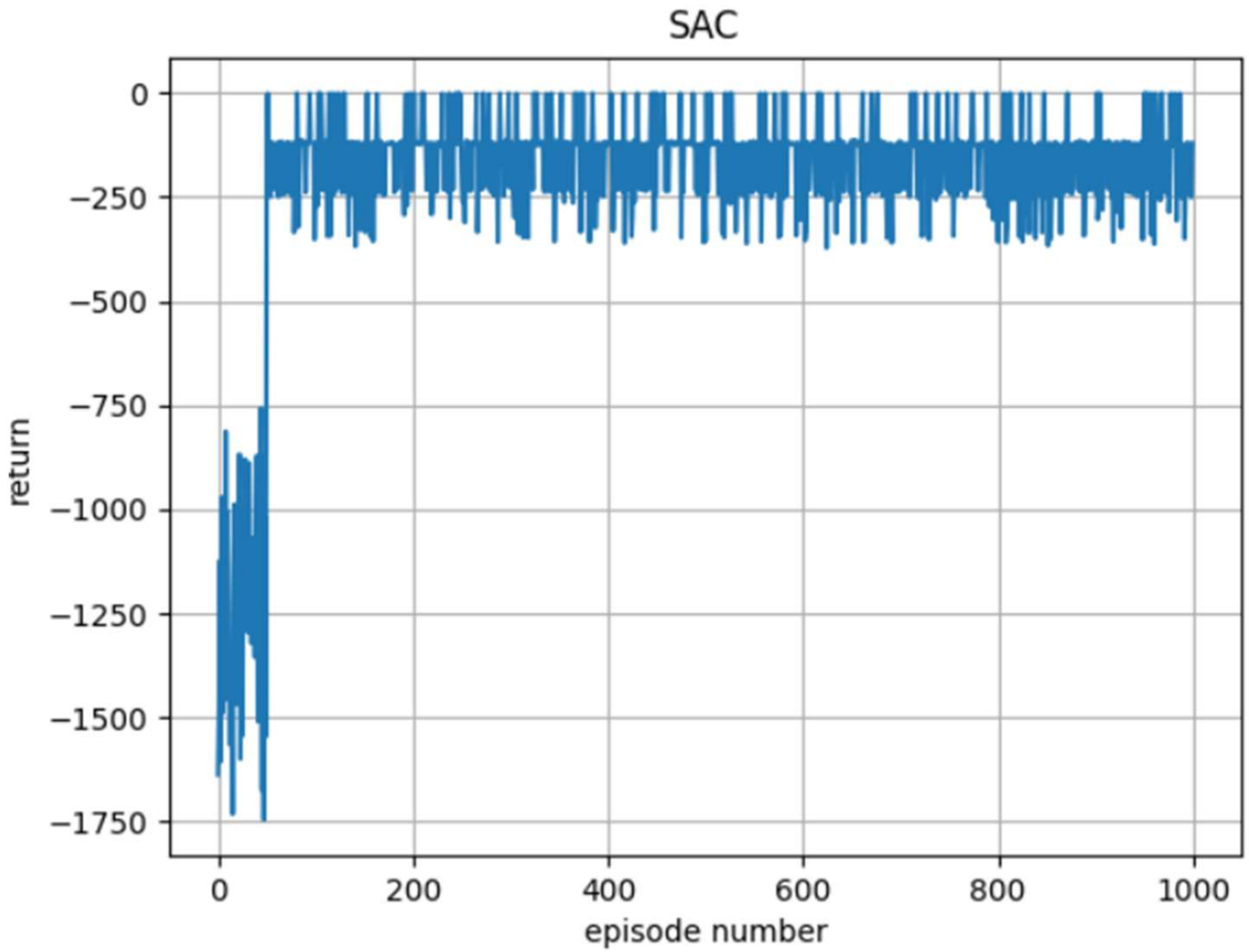
Solving 'Pendulum-v0' with SAC

- The main function
  - continued

```
            # Ignore the "done" signal if it comes from hitting the time horizon.
            # (https://github.com/openai/spinningup/blob/master/spinup/algos/sac/sac.py)
            mask = 1 if episode_steps == env._max_episode_steps else float(not done)

            memory.push(state, action, reward, next_state, mask) # Append transition to memory

            state = next_state

        if total_numsteps > num_steps:
            break

        print("Episode: {}, total numsteps: {}, episode steps: {}, reward: {}".format(
            i_episode, total_numsteps, episode_steps, round(episode_reward, 2)))

    env.close()

if __name__ == '__main__':
    main()
```

Solving 'Pendulum-v0' with SAC

- result
  - The agent trained to solve 'Pendulum-v0' using SAC.



End of Class

Questions?

Email: jso1@sogang.ac.kr