CORE
Control + Optimization Research Lab

Soft Actor-Critic (SAC)

이동민

삼성전자 서울대 공동연구소

Jul 19, 2019

Outline

- Soft Actor-Critic (SAC)
  - Learning process
  - Hyperparameter
  - Main loop
  - Train model
  - Train & TensorboardX
  - Learning curve & Test
- Comparison of algorithms for continuous action

Soft Actor-Critic (SAC)

- SAC Algorithm

Algorithm 1 Soft Actor-Critic

Input: θ_1, θ_2, ϕ ▷ Initial parameters
  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network weights
  $\mathcal{D} \leftarrow \emptyset$ ▷ Initialize an empty replay pool
  for each iteration **do**
    for each environment step **do**
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ ▷ Sample action from the policy
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ ▷ Sample transition from the environment
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$ ▷ Store the transition in the replay pool
    end for
    for each gradient step **do**
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ ▷ Update the Q-function parameters
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
      $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
      $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ ▷ Update target network weights
    end for
  end for
Output: θ_1, θ_2, ϕ ▷ Optimized parameters

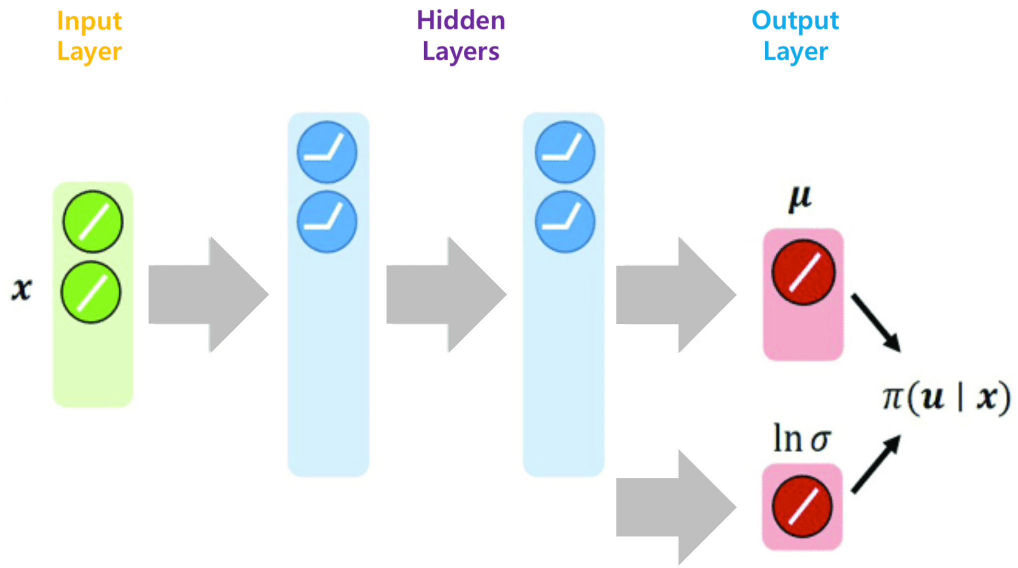
source : https://arxiv.org/pdf/1812.05905.pdf

Soft Actor-Critic (SAC)

- Learning process
  1. 상태에 따른 행동 선택
  2. 환경에서 선택한 행동으로 한 time step을 진행한 후, 다음 상태와 보상을 받음
  3. Sample (s, a, r, s')을 replay buffer에 저장
  4. Replay buffer에서 랜덤으로 sample을 추출
  5. 추출한 sample로 Actor & Critic network 업데이트
  6. Temperature parameter alpha 업데이트
  7. Critic에 대해 soft target 업데이트

Soft Actor-Critic (SAC)

- Actor network



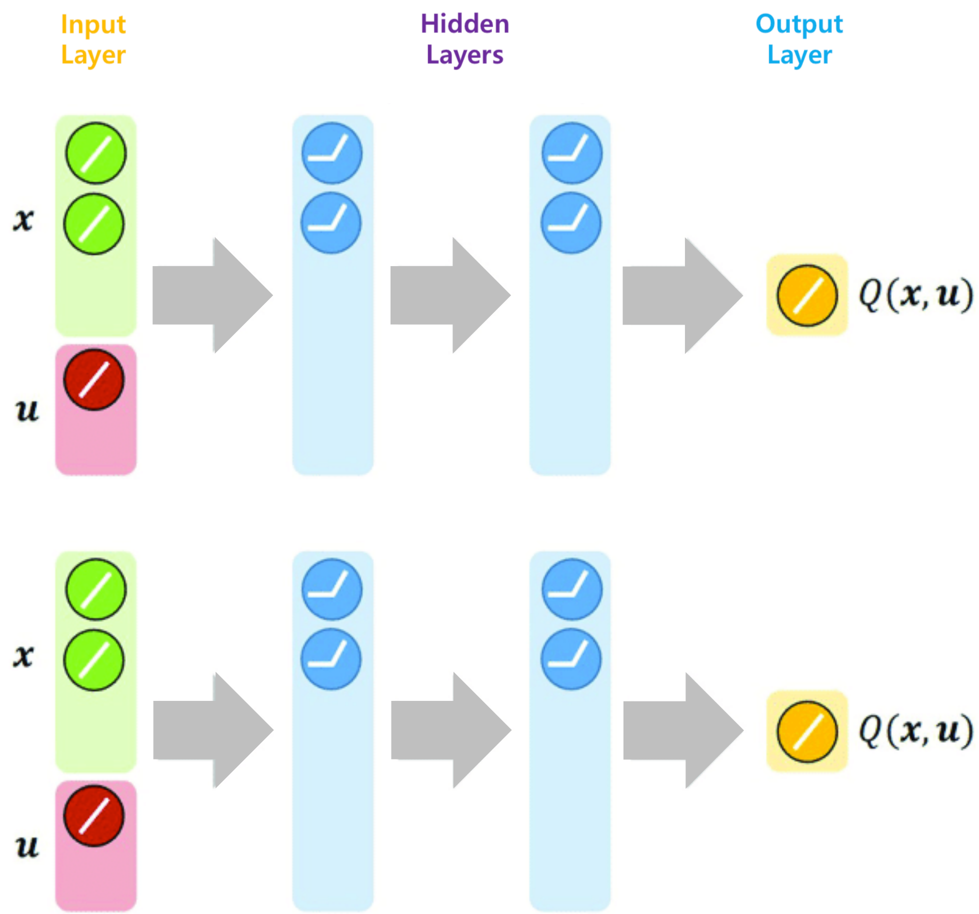
Soft Actor-Critic (SAC)

- Actor network

```python
class Actor(nn.Module):
    def __init__(self, state_size, action_size, args, log_std_min=-20, log_std_max=2):
        super(Actor, self).__init__()
        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.fc1 = nn.Linear(state_size, args.hidden_size)
        self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)

        self.fc3 = nn.Linear(args.hidden_size, action_size)
        self.fc4 = nn.Linear(args.hidden_size, action_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))

        mu = self.fc3(x)
        log_std = self.fc4(x)

        log_std = torch.clamp(log_std, min=self.log_std_min, max=self.log_std_max)
        std = torch.exp(log_std)

        return mu, std
```

Soft Actor-Critic (SAC)

- Critic network



source : [link]

Soft Actor-Critic (SAC)

- Critic network

```python
class Critic(nn.Module):
    def __init__(self, state_size, action_size, args):
        super(Critic, self).__init__()

        # Q1 architecture
        self.fc1 = nn.Linear(state_size + action_size, args.hidden_size)
        self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)
        self.fc3 = nn.Linear(args.hidden_size, 1)

        # Q2 architecture
        self.fc4 = nn.Linear(state_size + action_size, args.hidden_size)
        self.fc5 = nn.Linear(args.hidden_size, args.hidden_size)
        self.fc6 = nn.Linear(args.hidden_size, 1)

    def forward(self, states, actions):
        x = torch.cat([states, actions], dim=1)

        x1 = torch.relu(self.fc1(x))
        x1 = torch.relu(self.fc2(x1))
        q_value1 = self.fc3(x1)

        x2 = torch.relu(self.fc4(x))
        x2 = torch.relu(self.fc5(x2))
        q_value2 = self.fc6(x2)

        return q_value1, q_value2
```

Learning process

1. 상태에 따른 행동 선택

```python
        mu, std = actor(torch.Tensor(state))
        action = get_action(mu, std)
```

train.py

```python
def get_action(mu, std):
    normal = Normal(mu, std)
    z = normal.rsample() # reparameterization trick (mean + std * N(0,1))
    action = torch.tanh(z)

    return action.data.numpy()
```

utils.py

- normal.rsample()

```python
[docs]    def rsample(self, sample_shape=torch.Size()):
        shape = self._extended_shape(sample_shape)
        eps = _standard_normal(shape, dtype=self.loc.dtype, device=self.loc.device)
        return self.loc + eps * self.scale
```

Learning process

2. 환경에서 선택한 행동으로 한 time step을 진행한 후, 다음 상태와 보상을 받음

```
next_state, reward, done, _ = env.step(action)
```

3. Sample (s, a, r, s')을 replay buffer에 저장

```
replay_buffer = deque(maxlen=10000)
```

```
mask = 0 if done else 1

replay_buffer.append((state, action, reward, next_state, mask))
```

4. Replay buffer에서 랜덤으로 sample을 추출 (Batch size : 64)

```
mini_batch = random.sample(replay_buffer, args.batch_size)
```

CORE
Control + Optimization Research Lab

Learning process

5. 추출한 sample로 Actor & Critic network 업데이트

- Critic Loss

$$J_Q(\phi) = \Big(\underbrace{Q_\phi(s,a)}_{\text{Prediction}} - \underbrace{\Big(r + \gamma\big(Q_{\phi^-}(s',a') - \alpha \log \pi_\theta(a'|s') \big)\Big)}_{\text{Target}} \Big)^2$$

```python
# update critic
criterion = torch.nn.MSELoss()

# get Q-values using two Q-functions to mitigate overestimation bias
q_value1, q_value2 = critic(torch.Tensor(states), actions)

# get target
mu, std = actor(torch.Tensor(next_states))
next_policy, next_log_policy = eval_action(mu, std)
target_next_q_value1, target_next_q_value2 = target_critic(torch.Tensor(next_states), next_policy)

min_target_next_q_value = torch.min(target_next_q_value1, target_next_q_value2)
min_target_next_q_value = min_target_next_q_value.squeeze(1) - alpha * next_log_policy.squeeze(1)
target = rewards + masks * args.gamma * min_target_next_q_value

critic_loss1 = criterion(q_value1.squeeze(1), target.detach())
critic_optimizer.zero_grad()
critic_loss1.backward()
critic_optimizer.step()

critic_loss2 = criterion(q_value2.squeeze(1), target.detach())
critic_optimizer.zero_grad()
critic_loss2.backward()
critic_optimizer.step()
```

Learning process

5. 추출한 sample로 Actor & Critic network 업데이트
   - Actor Loss

$$J_\pi(\theta) = \frac{1}{N}\sum \alpha \log \pi_\theta(f_\theta(\epsilon; s)|s) - Q_\phi(s, f_\theta(\epsilon; s))$$

```
    # update actor
    mu, std = actor(torch.Tensor(states))
    policy, log_policy = eval_action(mu, std)

    q_value1, q_value2 = critic(torch.Tensor(states), policy)
    min_q_value = torch.min(q_value1, q_value2)

    actor_loss = ((alpha * log_policy) - min_q_value).mean()
    actor_optimizer.zero_grad()
    actor_loss.backward()
    actor_optimizer.step()
```

Learning process

6. Temperature parameter alpha 업데이트

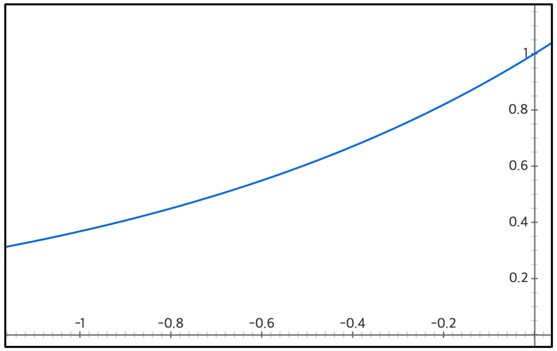
- Alpha Loss

$$J(\alpha) = -\frac{1}{N}\sum \alpha(\log \pi_\theta(f_\theta(\epsilon; s)|s) + \bar{\mathcal{H}})$$

```
    # update alpha
    alpha_loss = -(log_alpha * (log_policy + target_entropy).detach()).mean()
    alpha_optimizer.zero_grad()
    alpha_loss.backward()
    alpha_optimizer.step()

    alpha = torch.exp(log_alpha)

    return alpha
```

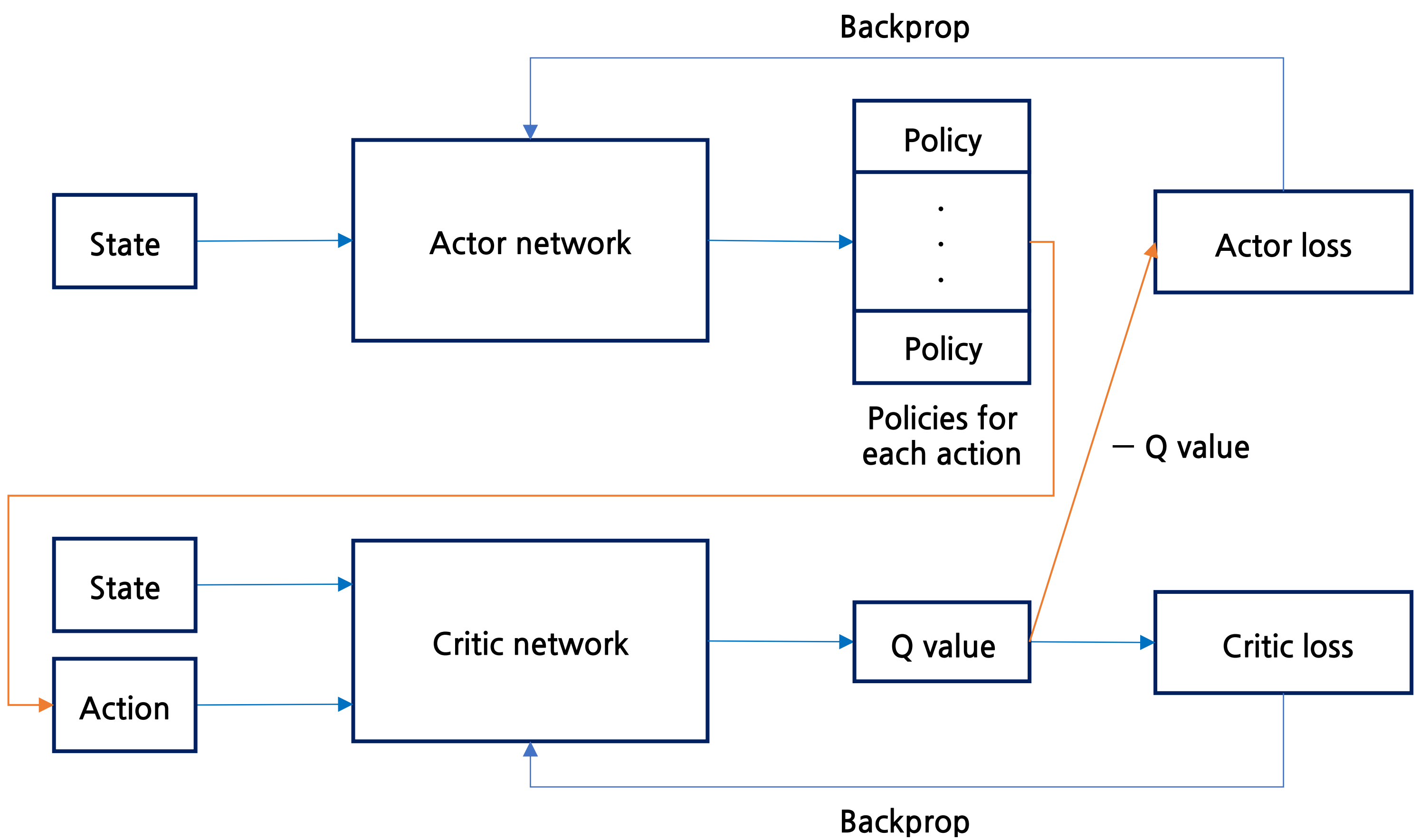
  - log_alpha : alpha를 $(0,1]$ 사이의 값으로 만들어주기 위해 사용

```
log_alpha tensor([0.], requires_grad=True)
alpha tensor([1.], grad_fn=<ExpBackward>)

log_alpha tensor([-0.0001], requires_grad=True)
alpha tensor([0.9999], grad_fn=<ExpBackward>)

log_alpha tensor([-0.0002], requires_grad=True)
alpha tensor([0.9998], grad_fn=<ExpBackward>)

log_alpha tensor([-0.0003], requires_grad=True)
alpha tensor([0.9997], grad_fn=<ExpBackward>)

log_alpha tensor([-0.0004], requires_grad=True)
alpha tensor([0.9996], grad_fn=<ExpBackward>)
```



y = exp(x)

Learning process



Learning process

7. Critic에 대해 soft target 업데이트

- Initialize target model

```
        hard_target_update(critic, target_critic)
```

train.py

```
def hard_target_update(net, target_net):
    target_net.load_state_dict(net.state_dict())
```

utils.py

- Soft target update (τ : 0.005)

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

$$\phi^{\pi'} \leftarrow \tau\phi^{\pi} + (1-\tau)\phi^{\pi'}$$

```
                soft_target_update(critic, target_critic, args.tau)
```

train.py

```
def soft_target_update(net, target_net, tau):
    for param, target_param in zip(net.parameters(), target_net.parameters()):
        target_param.data.copy_(tau * param.data + (1.0 - tau) * target_param.data)
```

utils.py

CORE
Control + Optimization Research Lab

Hyperparameter

```python
    parser = argparse.ArgumentParser()
    parser.add_argument('--env_name', type=str, default="Pendulum-v0")
    parser.add_argument('--load_model', type=str, default=None)
    parser.add_argument('--save_path', default='./save_model/', help='')
    parser.add_argument('--render', action="store_true", default=False)
    parser.add_argument('--gamma', type=float, default=0.99)
    parser.add_argument('--hidden_size', type=int, default=64)
    parser.add_argument('--batch_size', type=int, default=64)
    parser.add_argument('--actor_lr', type=float, default=1e-3)
    parser.add_argument('--critic_lr', type=float, default=1e-3)
    parser.add_argument('--alpha_lr', type=float, default=1e-4)
    parser.add_argument('--tau', type=float, default=0.005)
    parser.add_argument('--max_iter_num', type=int, default=1000)
    parser.add_argument('--log_interval', type=int, default=10)
    parser.add_argument('--goal_score', type=int, default=-300)
    parser.add_argument('--logdir', type=str, default='./logs',
                        help='tensorboardx logs directory')
    args = parser.parse_args()
```

Main loop

- Initialization
  - Seed - random number 고정
  - Actor & Critic network
  - Target critic network
  - Actor & Critic optimizer
  - Hard target update
  - Automatic entropy tuning
    - Target entropy
    - Log alpha, alpha
    - Alpha optimizer
  - TensorboardX
  - Replay buffer
  - Recent rewards

```python
def main():
    env = gym.make(args.env_name)
    env.seed(500)
    torch.manual_seed(500)

    state_size = env.observation_space.shape[0]
    action_size = env.action_space.shape[0]
    print('state size:', state_size)
    print('action size:', action_size)

    actor = Actor(state_size, action_size, args)
    critic = Critic(state_size, action_size, args)
    target_critic = Critic(state_size, action_size, args)

    actor_optimizer = optim.Adam(actor.parameters(), lr=args.actor_lr)
    critic_optimizer = optim.Adam(critic.parameters(), lr=args.critic_lr)

    hard_target_update(critic, target_critic)

    # initialize automatic entropy tuning
    target_entropy = -torch.prod(torch.Tensor(action_size)).item()
    log_alpha = torch.zeros(1, requires_grad=True)
    alpha = torch.exp(log_alpha)
    alpha_optimizer = optim.Adam([log_alpha], lr=args.alpha_lr)

    writer = SummaryWriter(args.logdir)

    replay_buffer = deque(maxlen=10000)
    recent_rewards = deque(maxlen=100)
    steps = 0
```

CORE
Control + Optimization Research Lab

Main loop

- Episode 진행
  - 상태에 따른 행동 선택
  - 다음 상태와 보상을 받음
  - Replay buffer에 저장

```
    for episode in range(args.max_iter_num):
        done = False
        score = 0

        state = env.reset()
        state = np.reshape(state, [1, state_size])

        while not done:
            if args.render:
                env.render()

            steps += 1

            mu, std = actor(torch.Tensor(state))
            action = get_action(mu, std)

            next_state, reward, done, _ = env.step(action)

            next_state = np.reshape(next_state, [1, state_size])
            mask = 0 if done else 1

            replay_buffer.append((state, action, reward, next_state, mask))

            state = next_state
            score += reward
```

Main loop

- Episode 진행
  - Replay buffer에서 랜덤으로 64개의 sample을 추출 → Mini batch
  - Train model
  - Critic에 대해 Soft target update

```python
            if steps > args.batch_size:
                mini_batch = random.sample(replay_buffer, args.batch_size)

                actor.train(), critic.train(), target_critic.train()
                alpha = train_model(actor, critic, target_critic, mini_batch,
                                    actor_optimizer, critic_optimizer, alpha_optimizer,
                                    target_entropy, log_alpha, alpha)

                soft_target_update(critic, target_critic, args.tau)

            if done:
                recent_rewards.append(score)
```

Main loop

- Print & Visualize log
- Termination : 최근 100개의 episode의 평균 score가 -300보다 크다면
    - Save model
    - 학습 종료

```
        if episode % args.log_interval == 0:
            print('{} episode | score_avg: {:.2f}'.format(episode, np.mean(recent_rewards)))
            writer.add_scalar('log/score', float(score), episode)

        if np.mean(recent_rewards) > args.goal_score:
            if not os.path.isdir(args.save_path):
                os.makedirs(args.save_path)

            ckpt_path = args.save_path + 'model.pth'
            torch.save(actor.state_dict(), ckpt_path)
            print('Recent rewards exceed -300. So end')
            break
```

Train model

- Mini batch → Numpy array
- Mini batch에 있는 64개의 sample들을 각각 나눔
  - state - (64, 3)
  - action - (64, 1)
  - reward - (64)
  - next_state - (64, 3)
  - mask - (64)

```python
def train_model(actor, critic, target_critic, mini_batch,
                actor_optimizer, critic_optimizer, alpha_optimizer,
                target_entropy, log_alpha, alpha):
    mini_batch = np.array(mini_batch)
    states = np.vstack(mini_batch[:, 0])
    actions = list(mini_batch[:, 1])
    rewards = list(mini_batch[:, 2])
    next_states = np.vstack(mini_batch[:, 3])
    masks = list(mini_batch[:, 4])

    actions = torch.Tensor(actions).squeeze(1)
    rewards = torch.Tensor(rewards).squeeze(1)
    masks = torch.Tensor(masks)
```

Train model

- Prediction
- Target

```
    # update critic
    criterion = torch.nn.MSELoss()

    # get Q-values using two Q-functions to mitigate overestimation bias
    q_value1, q_value2 = critic(torch.Tensor(states), actions)

    # get target
    mu, std = actor(torch.Tensor(next_states))
    next_policy, next_log_policy = eval_action(mu, std)
    target_next_q_value1, target_next_q_value2 = target_critic(torch.Tensor(next_states), next_policy)

    min_target_next_q_value = torch.min(target_next_q_value1, target_next_q_value2)
    min_target_next_q_value = min_target_next_q_value.squeeze(1) - alpha * next_log_policy.squeeze(1)
    target = rewards + masks * args.gamma * min_target_next_q_value
```

CORE
Control + Optimization Research Lab

Train model

- Prediction
  - q_value1, 2 - (64, 1)
- Target

```python
    # update critic
    criterion = torch.nn.MSELoss()

    # get Q-values using two Q-functions to mitigate overestimation bias
    q_value1, q_value2 = critic(torch.Tensor(states), actions)
```

Train model

- Prediction
  - **q_value1, 2 -** (64, 1)
- Target
  - **mu, std -** (64, 1)

```
    # get target
    mu, std = actor(torch.Tensor(next_states))
    next_policy, next_log_policy = eval_action(mu, std)
    target_next_q_value1, target_next_q_value2 = target_critic(torch.Tensor(next_states), next_policy)

    min_target_next_q_value = torch.min(target_next_q_value1, target_next_q_value2)
    min_target_next_q_value = min_target_next_q_value.squeeze(1) - alpha * next_log_policy.squeeze(1)
    target = rewards + masks * args.gamma * min_target_next_q_value
```

Train model

utils.py

```python
def eval_action(mu, std, epsilon=1e-6):
    normal = Normal(mu, std)
    z = normal.rsample() # reparameterization trick (mean + std * N(0,1))
    action = torch.tanh(z)
    log_prob = normal.log_prob(z)

    # Enforcing Action Bounds
    log_prob -= torch.log(1 - action.pow(2) + epsilon)
    log_policy = log_prob.sum(1, keepdim=True)

    return action, log_policy
```

$$\log \pi_\theta(a|s) = -\frac{\left(a - \mu_\theta(s)\right)^2}{2\sigma^2} - \log \sigma - \log \sqrt{2\pi}$$

Train model

- Prediction
  - **q_value1, 2 -** (64, 1)
- Target
  - **mu, std -** (64, 1)

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

$$\pi(\mathbf{a}|\mathbf{s}) = \mu(\mathbf{u}|\mathbf{s}) \left| \det \left(\frac{d\mathbf{a}}{d\mathbf{u}} \right) \right|^{-1}. \tag{25}$$

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

$$\boxed{\log \pi(\mathbf{a}|\mathbf{s}) = \log \mu(\mathbf{u}|\mathbf{s}) - \sum_{i=1}^{D} \log \left(1 - \tanh^2(u_i)\right)}, \tag{26}$$

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

Train model

- Prediction
  - **q_value1, 2 -** (64, 1)
- Target
  - **mu, std -** (64, 1)

utils.py

```python
def eval_action(mu, std, epsilon=1e-6):
    normal = Normal(mu, std)
    z = normal.rsample() # reparameterization trick (mean + std * N(0,1))
    action = torch.tanh(z)
    log_prob = normal.log_prob(z)

    # Enforcing Action Bounds
    log_prob -= torch.log(1 - action.pow(2) + epsilon)
    log_policy = log_prob.sum(1, keepdim=True)

    return action, log_policy
```

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

Train model

- Prediction
  - **q_value1, 2 -** (64, 1)
- Target
  - **mu, std -** (64, 1)
  - **next_policy, next_log_policy -** (64, 1)
  - **target_next_q_value1, 2 -** (64, 1)
  - **min_target_next_q_value -** (64)
  - **target -** (64)

```
    # get target
    mu, std = actor(torch.Tensor(next_states))
    next_policy, next_log_policy = eval_action(mu, std)
    target_next_q_value1, target_next_q_value2 = target_critic(torch.Tensor(next_states), next_policy)

    min_target_next_q_value = torch.min(target_next_q_value1, target_next_q_value2)
    min_target_next_q_value = min_target_next_q_value.squeeze(1) - alpha * next_log_policy.squeeze(1)
    target = rewards + masks * args.gamma * min_target_next_q_value
```

Train model

- Update critic - MSE Loss
  - $J_Q(\phi) = (\underbrace{Q_\phi(s,a)}_{\text{Prediction}} - \underbrace{\left(r + \gamma\left(Q_{\phi^-}(s',a') - \alpha \log \pi_\theta(a'|s') \right) \right)}_{\text{Target}})^2$

```python
# update critic
criterion = torch.nn.MSELoss()

# get Q-values using two Q-functions to mitigate overestimation bias
q_value1, q_value2 = critic(torch.Tensor(states), actions)

# get target
mu, std = actor(torch.Tensor(next_states))
next_policy, next_log_policy = eval_action(mu, std)
target_next_q_value1, target_next_q_value2 = target_critic(torch.Tensor(next_states), next_policy)

min_target_next_q_value = torch.min(target_next_q_value1, target_next_q_value2)
min_target_next_q_value = min_target_next_q_value.squeeze(1) - alpha * next_log_policy.squeeze(1)
target = rewards + masks * args.gamma * min_target_next_q_value

critic_loss1 = criterion(q_value1.squeeze(1), target.detach())
critic_optimizer.zero_grad()
critic_loss1.backward()
critic_optimizer.step()

critic_loss2 = criterion(q_value2.squeeze(1), target.detach())
critic_optimizer.zero_grad()
critic_loss2.backward()
critic_optimizer.step()
```

Train model

- Update actor
  - **mu, std -** (64, 1)
  - **policy, log_policy -** (64, 1)
  - **q_value1, 2 -** (64, 1)
  - **min_q_value -** (64, 1)
  - **alpha -** (1)
  - $J_\pi(\theta) = \frac{1}{N}\sum \alpha \log \pi_\theta(f_\theta(\epsilon; s)|s) - Q_\phi(s, f_\theta(\epsilon; s))$

```python
    # update actor
    mu, std = actor(torch.Tensor(states))
    policy, log_policy = eval_action(mu, std)

    q_value1, q_value2 = critic(torch.Tensor(states), policy)
    min_q_value = torch.min(q_value1, q_value2)

    actor_loss = ((alpha * log_policy) - min_q_value).mean()
    actor_optimizer.zero_grad()
    actor_loss.backward()
    actor_optimizer.step()
```

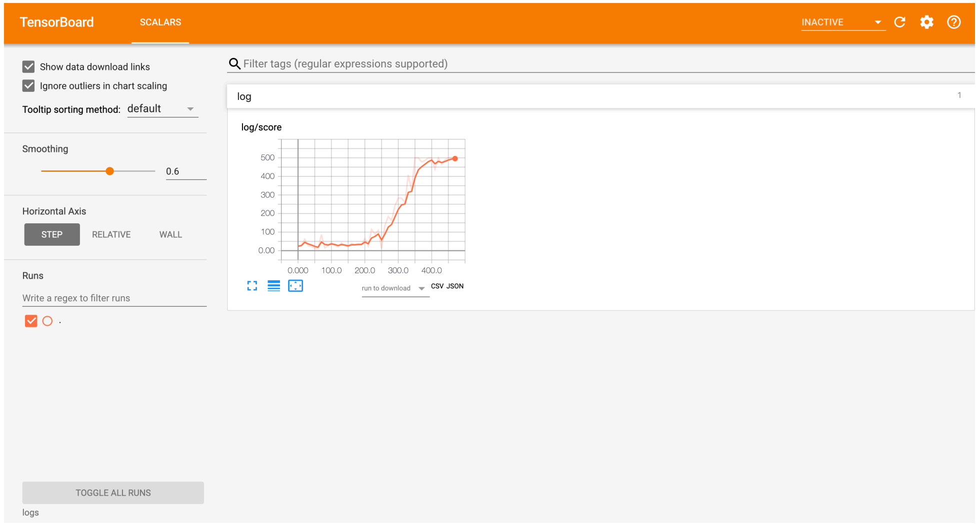
Train model

- Update alpha
  - log_alpha - (1)
  - log_policy - (64, 1)
  - target_entropy → -0.0
  - $J(\alpha) = -\frac{1}{N}\sum \alpha(\log \pi_\theta(f_\theta(\epsilon; s)|s) + \bar{\mathcal{H}})$
  - alpha - (1)

```
    # update alpha
    alpha_loss = -(log_alpha * (log_policy + target_entropy).detach()).mean()
    alpha_optimizer.zero_grad()
    alpha_loss.backward()
    alpha_optimizer.step()

    alpha = torch.exp(log_alpha)

    return alpha
```

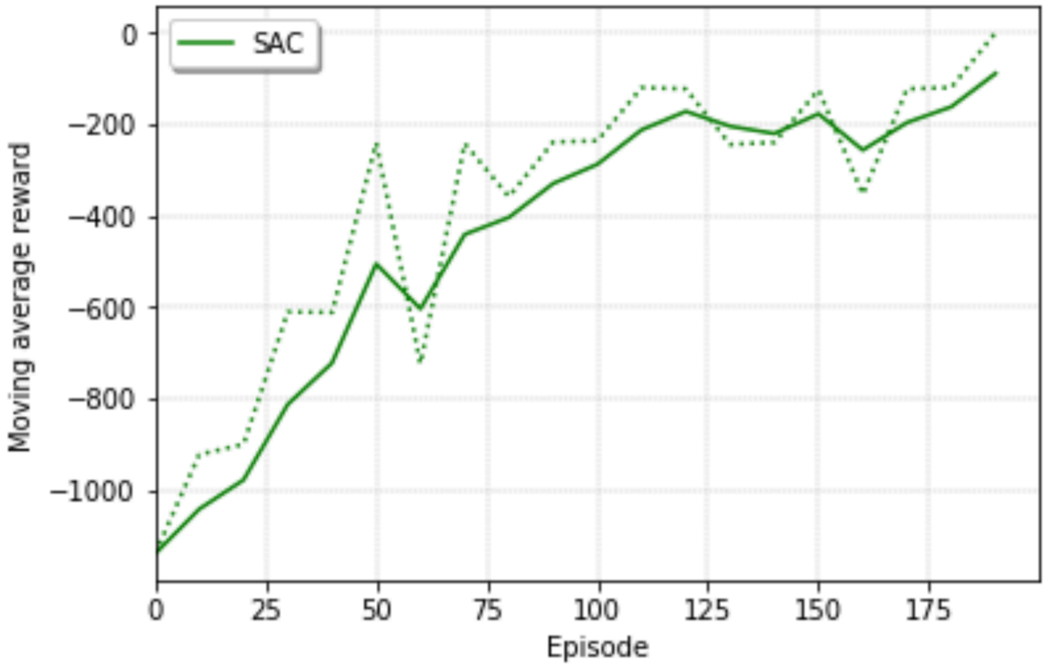
Train & TensorboardX

- Terminal A - train 실행
  - conda activate env_name
  - python train.py
- Terminal B - tensorboardX 실행
  - conda activate env_name
  - tensorboard --logdir logs
  - (웹에서) localhost:6006



Learning curve & Test

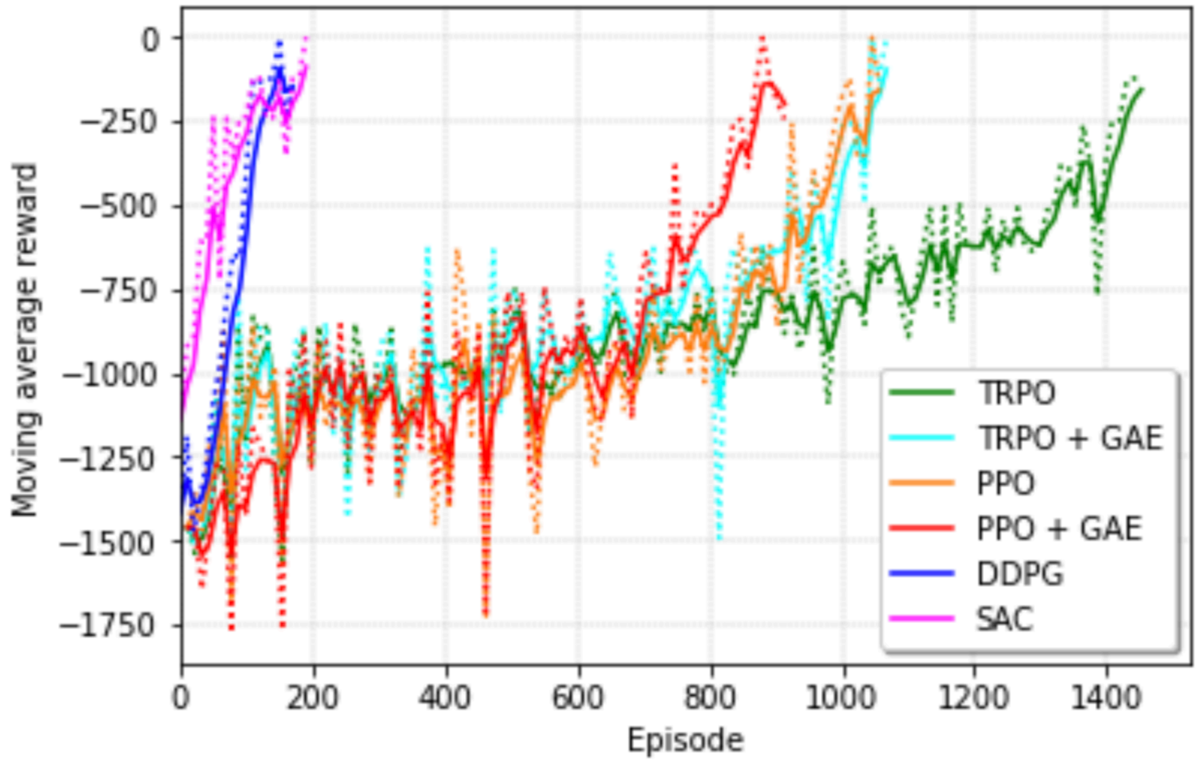
- Learning curve



- Test
  - python test.py

Comparison of algorithms for continuous action

- Learning curve



Comparison of algorithms for continuous action

Continuous action일 때는 어떤 알고리즘이 제일 좋을까요?
그리고 그 알고리즘이 왜 좋을까요?

- TRPO, PPO (On-policy algorithms)
  - **Poor sample efficiency** → 업데이트를 하기 위해서는 새로운 sample들을 수집해야 함
  - The number of gradient step and samples per step that are **extravagantly expensive**
- DDPG (Off-policy algorithm)
  - The interplay between the **deterministic actor** and the Q-function typically makes DDPG extreme **difficult to stabilize and brittle to hyperparameter settings**
  - Exploration noise can cause **sudden failures in unstable environments**
- SAC (Off-policy algorithm)
  - The **stochastic actor** aims to maximize expected reward while also **maximizing entropy**
  - **Automatic** gradient-based **temperature hyperparameter tuning** method

Thank you