SAC Code Practice

Dongmin Lee

RL Korea Bootcamp

October 27, 2019

Outline

- Soft Actor-Critic (SAC)
- Model
- Learning Process
- Train & Test

Soft Actor-Critic (SAC)

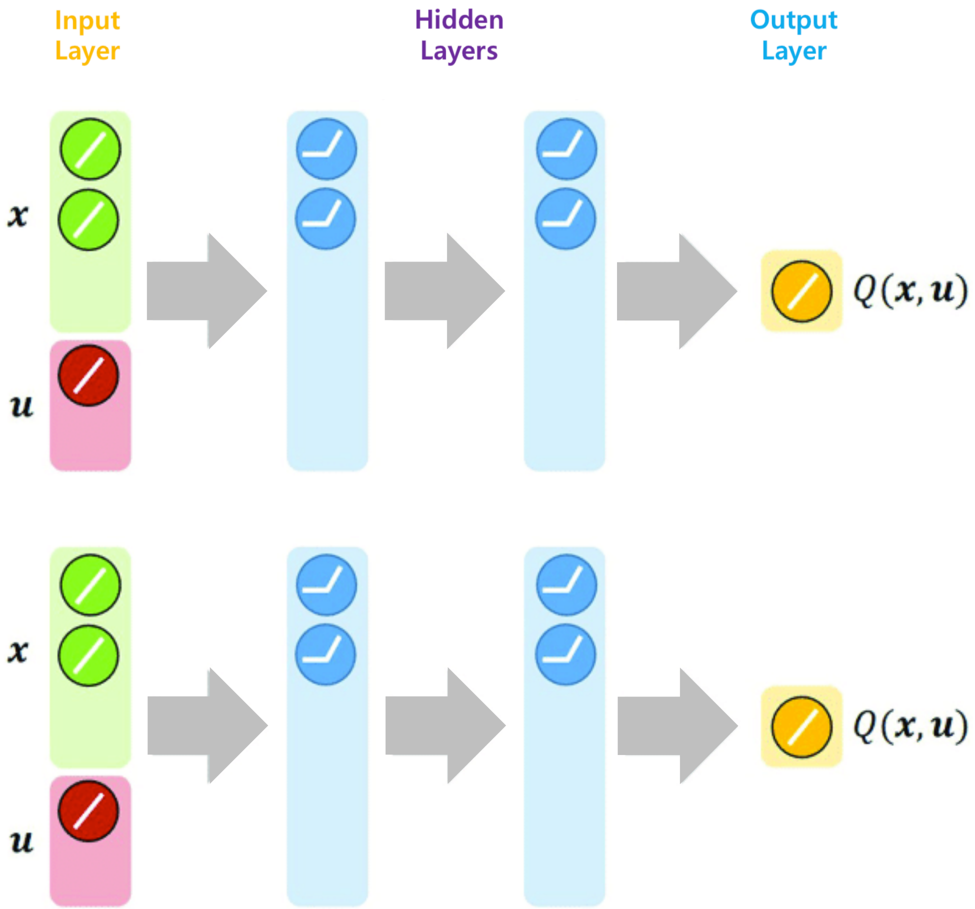
Automating entropy adjustment on SAC (ASAC)

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

Model

Critic network



Model

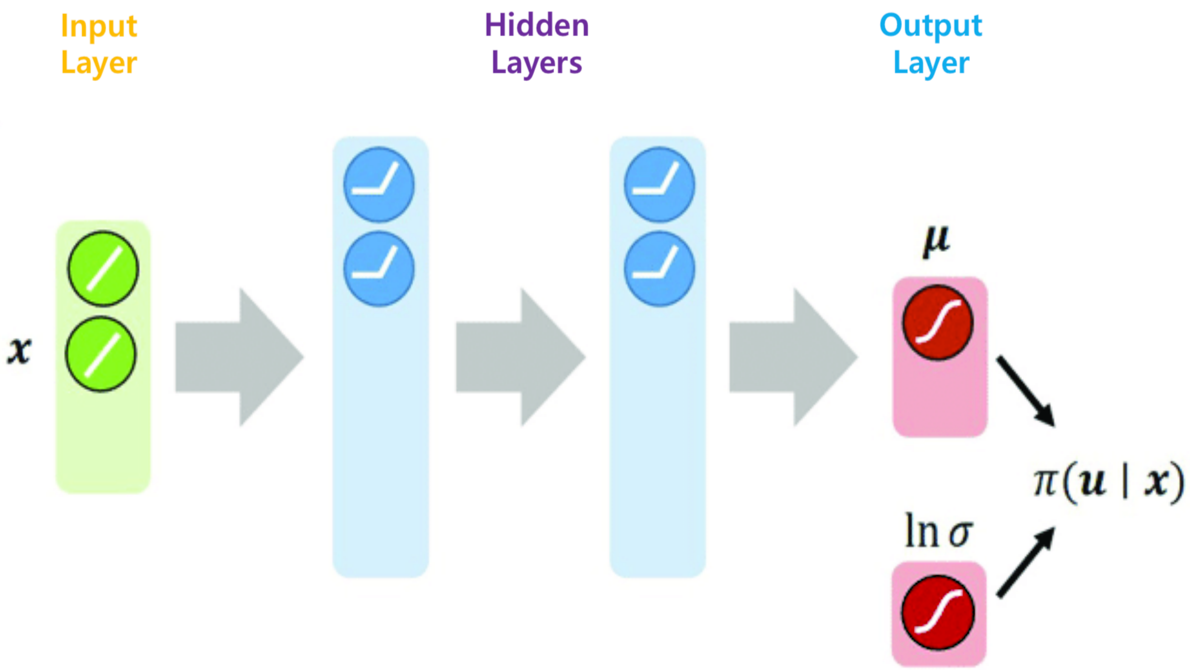
Critic network (model.py)

```
class MLP(nn.Module):
    def __init__(self,
                 input_size,
                 output_size,
                 hidden_sizes=(128,128),
                 activation=F.relu,
                 output_activation=identity,
                 use_output_layer=True,
    ):
        super(MLP, self).__init__()

        self.input_size = input_size
        self.output_size = output_size
        self.hidden_sizes = hidden_sizes
        self.activation = activation
        self.output_activation = output_activation
        self.use_output_layer = use_output_layer

        # Set hidden layers
        self.hidden_layers = nn.ModuleList()
        in_size = self.input_size
        for next_size in self.hidden_sizes:
            fc = nn.Linear(in_size, next_size)
            in_size = next_size
            self.hidden_layers.append(fc)

        # Set output layers
        if self.use_output_layer:
            self.output_layer = nn.Linear(in_size, self.output_size)
        else:
            self.output_layer = identity

    def forward(self, x):
        for hidden_layer in self.hidden_layers:
            x = self.activation(hidden_layer(x))
        x = self.output_activation(self.output_layer(x))
        return x
```

```
class FlattenMLP(MLP):
    def forward(self, x, a):
        q = torch.cat([x,a], dim=-1)
        return super(FlattenMLP, self).forward(q)
```

Model

Actor network



Model

Actor network (model.py)

```python
LOG_STD_MAX = 2
LOG_STD_MIN = -20

class GaussianPolicy(MLP):
    def __init__(self,
                 input_size,
                 output_size,
                 hidden_sizes=(128,128),
                 activation=F.relu,
    ):
        super(GaussianPolicy, self).__init__(
            input_size=input_size,
            output_size=output_size,
            hidden_sizes=hidden_sizes,
            activation=activation,
            use_output_layer=False,
        )

        in_size = hidden_sizes[-1]

        # Set output layers
        self.mu_layer = nn.Linear(in_size, output_size)
        self.log_std_layer = nn.Linear(in_size, output_size)

    def clip_but_pass_gradient(self, x, l=-1., u=1.):
        clip_up = (x > u).float()
        clip_low = (x < l).float()
        clip_value = (u - x)*clip_up + (l - x)*clip_low
        return x + clip_value.detach()

    def apply_squashing_func(self, mu, pi, log_pi):
        mu = torch.tanh(mu)
        pi = torch.tanh(pi)
        # To avoid evil machine precision error, strictly clip 1-pi**2 to [0,1] range.
        log_pi -= torch.sum(torch.log(self.clip_but_pass_gradient(1 - pi.pow(2), l=0., u=1.) + 1e-6), dim=-1)
        return mu, pi, log_pi
```

```python
    def forward(self, x):
        x = super(GaussianPolicy, self).forward(x)

        mu = self.mu_layer(x)
        log_std = torch.tanh(self.log_std_layer(x))
        log_std = LOG_STD_MIN + 0.5 * (LOG_STD_MAX - LOG_STD_MIN) * (log_std + 1)
        std = torch.exp(log_std)

        dist = Normal(mu, std)
        pi = dist.rsample() # reparameterization trick (mean + std * N(0,1))
        log_pi = dist.log_prob(pi).sum(dim=-1)
        mu, pi, log_pi = self.apply_squashing_func(mu, pi, log_pi)

        return mu, pi, log_pi
```

Learning Process

1. Collect experience (s_t, a_t, r_t, s_{t+1}) using some policy, add them to **Buffer**;
2. Sample a mini-batch $\{(s_i, a_i, r_i, s_{i+1})\}$ from **Buffer**;
3. Set target $y_i^- \triangleq r_i + \gamma(Q_{\theta^-}\left(s_{i+1}, \pi_\phi(s_{i+1})\right) - \alpha \log \pi_\phi(a_{i+1}|s_{i+1}))$;
4. Update the critic network by minimizing
   $J_Q(\theta) \triangleq \mathbb{E}\left[\frac{1}{2}(Q_\theta(s_i, a_i) - y_i^-)^2\right]$;
5. Update the actor network by minimizing
   $J_\pi(\phi) \triangleq \mathbb{E}\left[\alpha \log \pi_\phi(a_i|s_i) - Q_\theta(s_i, a_i)\right]$;
6. Update the alpha by minimizing $\mathbb{E}\left[-\alpha \log \pi_\phi(a_i|s_i) - \alpha H_0\right]$
7. Update the target networks: $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$ with small τ;

Learning Process

1. Collect experience (s_t, a_t, r_t, s_{t+1}) using some policy, add them to **Buffer**;
2. Sample a mini-batch $\{(s_i, a_i, r_i, s_{i+1})\}$ from **Buffer**;

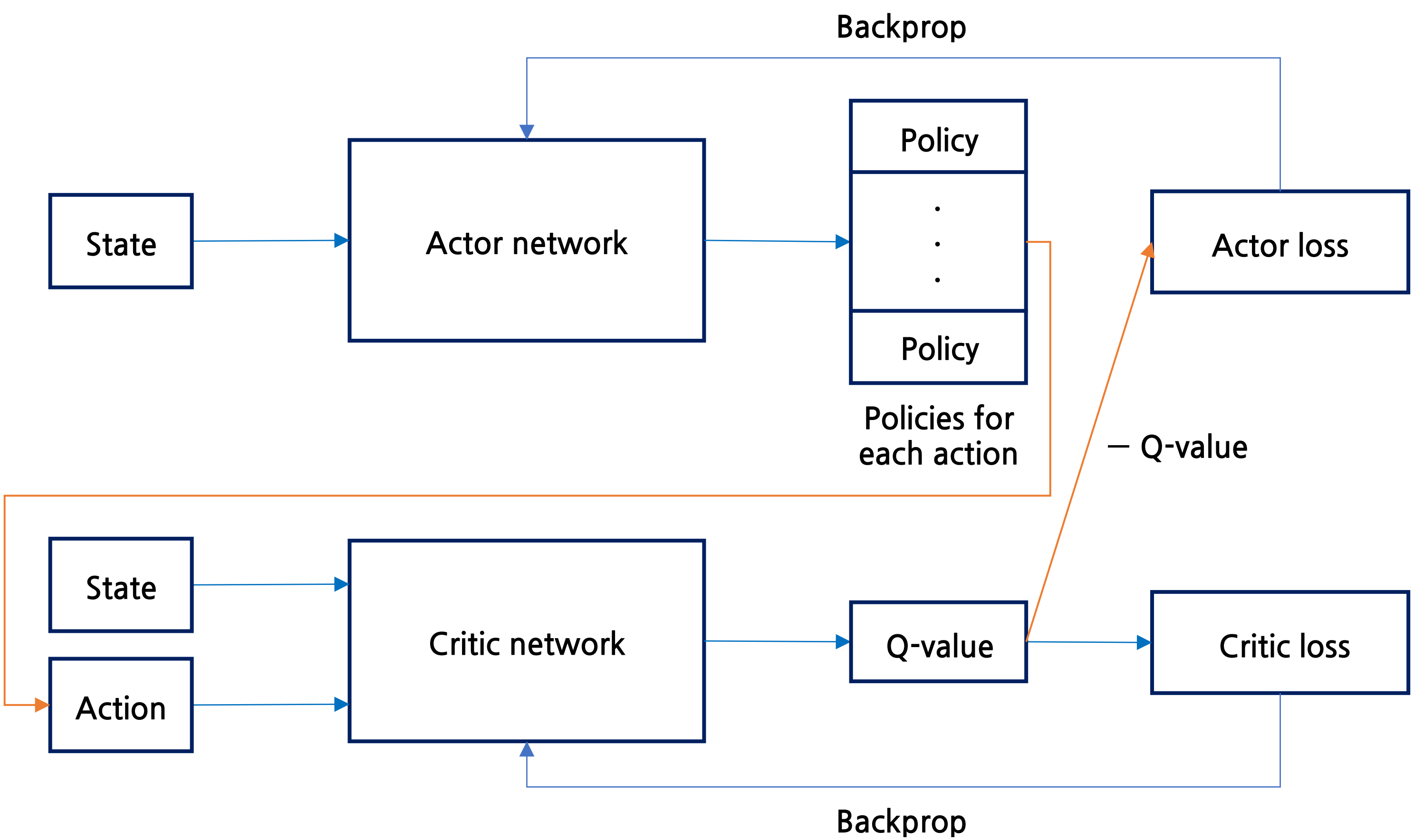
```
# Collect experience (s, a, r, s') using some policy
_, action, _ = actor(torch.Tensor(obs).to(device))
action = action.detach().cpu().numpy()
next_obs, reward, done, _ = env.step(action)

# Add experience to replay buffer
replay_buffer.add(obs, action, reward, next_obs, done)

# Start training when the number of experience is greater than batch size
if steps > args.batch_size:
    batch = replay_buffer.sample(args.batch_size)
    train_model(actor, qf1, qf2, qf1_target, qf2_target,
                actor_optimizer, qf1_optimizer, qf2_optimizer,
                batch, target_entropy, log_alpha, alpha_optimizer)
```

Learning Process

3. Set target $y_i^- \triangleq r_i + \gamma\left(Q_{\theta^-}\left(s_{i+1}, \pi_\phi(s_{i+1})\right) - \alpha \log \pi_\phi(a_{i+1}|s_{i+1})\right)$;
4. Update the critic network by minimizing
   $J_Q(\theta) \triangleq \mathbb{E}\left[\frac{1}{2}\left(Q_\theta(s_i, a_i) - y_i^-\right)^2\right]$;
5. Update the actor network by minimizing
   $J_\pi(\phi) \triangleq \mathbb{E}\left[\alpha \log \pi_\phi(a_i|s_i) - Q_\theta(s_i, a_i)\right]$;

```
    # Prediction π(s), logπ(s), π(s'), logπ(s'), Q1(s,a), Q2(s,a)
    _, pi, log_pi = actor(obs1)
    _, next_pi, next_log_pi = actor(obs2)
    q1 = qf1(obs1, acts).squeeze(1)
    q2 = qf2(obs1, acts).squeeze(1)

    # Min Double-Q: min(Q1(s,π(s)), Q2(s,π(s))), min(Q1‾(s',π(s')), Q2‾(s',π(s')))
    min_q_pi = torch.min(qf1(obs1, pi), qf2(obs1, pi)).squeeze(1).to(device)
    min_q_next_pi = torch.min(qf1_target(obs2, next_pi), qf2_target(obs2, next_pi)).squeeze(1).to(device)

    # Targets for Q and V regression
    v_backup = min_q_next_pi - args.alpha*next_log_pi
    q_backup = rews + args.gamma*(1-done)*v_backup
    q_backup.to(device)

    if 0: # Check shape of prediction and target…

    # Soft actor-critic losses
    actor_loss = (args.alpha*log_pi - min_q_pi).mean()
    qf1_loss = F.mse_loss(q1, q_backup.detach())
    qf2_loss = F.mse_loss(q2, q_backup.detach())
```

Learning Process



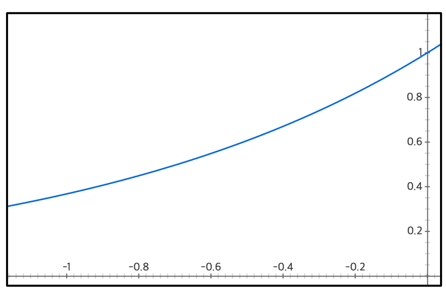
Learning Process

6. Update the alpha by minimizing $\mathbb{E}\left[-\alpha \log \pi_\phi(a_i|s_i) - \alpha H_0\right]$

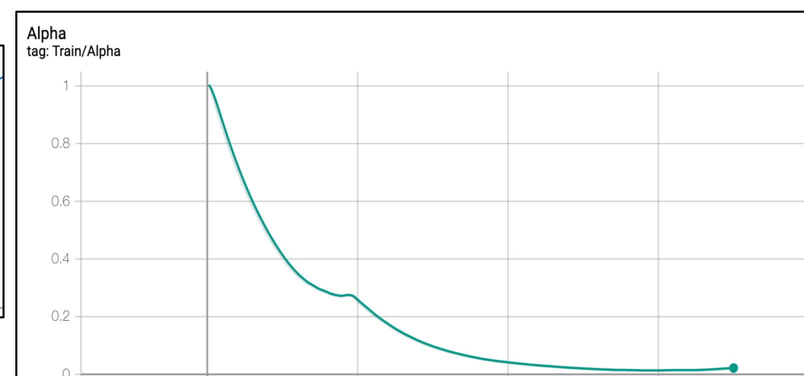
```python
    # If automatic entropy tuning is True, initialize a target entropy, a log alpha and an alpha optimizer
    if args.automatic_entropy_tuning:
        target_entropy = -np.prod((act_dim,)).item()
        log_alpha = torch.zeros(1, requires_grad=True, device=device)
        alpha_optimizer = optim.Adam([log_alpha], lr=args.alpha_lr)
    else:
        target_entropy = None
        log_alpha = None
        alpha_optimizer = None
```

```python
    # If automatic entropy tuning is True, update alpha
    if args.automatic_entropy_tuning:
        alpha_loss = -(log_alpha * (log_pi + target_entropy).detach()).mean()
        alpha_optimizer.zero_grad()
        alpha_loss.backward()
        alpha_optimizer.step()

        args.alpha = log_alpha.exp()
```

```
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

7. Update the target networks: $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$ with small τ;

```
    # Polyak averaging for target parameter
    soft_target_update(qf1, qf1_target)
    soft_target_update(qf2, qf2_target)
```

```
def soft_target_update(main, target, tau=0.005):
    for main_param, target_param in zip(main.parameters(), target.parameters()):
        target_param.data.copy_(tau*main_param.data + (1.0-tau)*target_param.data)
```

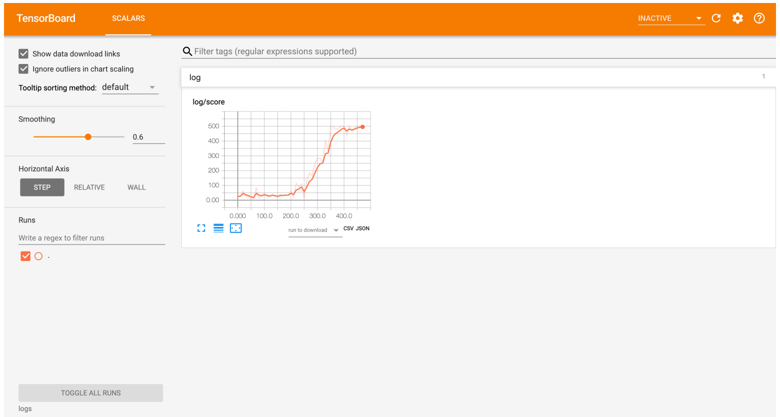
Train & Test

Terminal A - train

- conda activate env_name
- python train.py
- python test.py (after training)

Terminal B - Tensorboard

- conda activate env_name
- tensorboard --logdir=runs
- localhost:6006 (in a web browser)



Thank You!