SAC(Soft Actor-Critic)

Paper Link: https://arxiv.org/abs/1801.01290

Key Features

- SAC(Soft Actor-Critic) is a off-policy actor-critic algorithm based on maximum entropy framework
- SAC is a soft policy iteration algorithm based on approximation with respect to policy and value(V, Q)
- The characteristic of SAC is that it secures diversity of samples by constructing policies with high entropy while learning with the same goal of maximizing expected rewards as the existing RL algorithm. So that can achieve SOTA(State-Of-The-Art) performance on task which has continuous state, action domain and high sample complexity

Background

Difficulties in applying Model-free RL algorithms to the real-world

The reason why it is difficult to apply Model-free RL algorithms to the real-world are high sample complexity and high sensitivity with respect to hyperparameters. The problem in real-world, despite being a simple task, millions of data collection are required to solve. On-policy algorithms require samples for each learning step. So, it is not efficient with respect to sample efficiency. Accordingly in this case, off-policy algorithms which reuses the samples is mainly used. However, even in the case of off-policy algorithms, if task has a continuous state and action domain, the applicable algorithm is only DDPG which is policy gradient method combined with Q-function. However, since DDPG has high sensitivity to hyperparameters and weak convergent properties, it is difficult to solve such problems using DDPG.

Therefore, Model-free algorithm that is stable in the continuous state and action space and efficient for samples is needed. Accordingly, SAC was presented.

Maximum Entropy Framework

In SAC, the maximum entropy framework is used. While the standard RL aims to maximize the sum of expected reward, proposed framework considers a more general maximum entropy objective function as follows.

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,\, a_t) \sim \rho_\pi} \left[r(s_t,\ a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right]$$

The term of $\mathcal{H}(\pi(\cdot|s_t))$ is policy entropy and the coefficient a is a proportion whose value exists between 0 to 1. If a is 0, the objective function same with the things of conventional RL. If a increases to 1, the objective function increases the probability of constructing a highly exploratory policy. The parameter a determine importance of entropy term, so that it can be used to control the degree of exploration.

There are two important points in the above equation. First, the policy is incentivized to explore more widely, while giving up on clearly unpromising avenues. Second, the policy can capture multiple modes of near optimal action. In problem settings where multiple actions seem equally attractive, the policy will commit equal probability mass to those actions.

Method

From SPI(Soft Policy Iteration) to SAC

SPI(Soft Policy Iteration) algorithm is a policy iteration algorithm using maximum entropy framework. Soft policy evaluation follows the following equation in calculating policy(1), Q-function(2), and state value function(3) repeatedly applying a modified Bellman backup operator($\mathcal{T}^{\pi}$).

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,\ a_t)\sim\rho_\pi}[r(s_t,\ a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{1}$$

$$\mathcal{T}^{\pi}Q(s_t,\ a_t) \triangleq r(s_t,\ a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}[V(s_{t+1})], \tag{2}$$

$$where \quad V(s_t) = \mathbb{E}_{a_t\sim\pi}[Q(s_t,\ a_t) - log\pi(a_t|s_t)] \tag{3}$$

After that, soft policy improvement update the policy according to equation as follows.

$$\pi_{new} = \underset{\pi'\in\Pi}{argmin} D_{KL}(\pi'(\cdot|s_t) \parallel \frac{exp(Q^{\pi_{old}}(s_t,\cdot))}{Z^{\pi_{old}}(s_t)})$$

The partition function, $Z^{\pi_{old}}(s_t)$ normalizes distribution. while it is intractable in general, it has no effect on calculating gradient, so this can be ignored. As a result, new policy(π_{new}) has higher value with respect to objective than old policy(π_{old}).

Soft policy iteration algorithm alternates between soft policy evaluation and soft policy improvement to find optimal policy for maximum entropy. Although this algorithm will provably find the optimal solution, it has expensive computational cost and can perform in its exact form only in the tabular case.

SAC adopt approximation method to solve this problem. SAC has two kinds of networks: Actor, Critic. Both networks are optimized by SGD(Stochastic Gradient Descent) to approximate policy, Q-function and state value function.

The state value function approximates the soft value. There is no need in principle to include a separate function approximator for the state value, since it is related to the Q-function
and policy(equation (3)). Separate function approximator for the soft value can stabilize training and is convenient to train simultaneously with the other networks. The soft value function is trained to minimize the squared residual error as follows.

$$J_V(\psi) = \mathbb{E}_{s_t \sim \mathcal{D}}[\frac{1}{2}(V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi_\phi}[Q_\theta(s_t, a_t) - log\pi_\phi(a_t|s_t)])^2]$$

where $\mathcal{D}$ is the distribution of previously sampled states and actions, or a replay buffer. The gradient of this equation can be estimated with an unbiased estimator like equation as follows.

$$\hat{\nabla}_\psi J_V(\psi) = \nabla_\psi V_\psi(s_t)(V_\psi(s_t) - Q_\theta(s_t, a_t) + log\pi_\phi(a_t|s_t))$$

where the actions are sampled according to the current policy, instead of the replay buffer. The soft Q-function parameters can be trained to minimize the soft Bellman residual error as follows.

$$J_Q(\theta) = \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}}[\frac{1}{2}(Q_\theta(s_t, a_t) - \hat{Q}(s_t, a_t))^2] \tag{4}$$

$$with \quad \hat{Q}(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}[V_{\bar{\psi}}(s_{t+1})] \tag{5}$$

which again can be optimized with stochastic gradients as follows.

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(a_t, s_t)(Q_\theta(s_t, a_t) - r(s_t, a_t) - \gamma V_{\bar{\psi}}(s_{t+1}))$$

The update makes use of a target value network $V_{\bar{\psi}}$, where $\bar{\psi}$ can be an exponentially moving average of the value network weights, which stabilize training. Alternatively, it can be updated the target weights to match the current value function weights periodically. Finally, the policy parameters can be learned by directly minimizing the expected KL-divergence in equation as follows.

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}}[D_{KL}(\pi_\phi(\cdot|s_t) \parallel \frac{exp(Q_\theta(s_t, \cdot))}{Z_\theta(s_t)})]$$

There are several options for minimizing J_π. In this case, the target density is the Q-function which is represented by a neural network and can be differentiated. So, it is thus convenient
to apply the reparameterization trick, resulting in a lower variance estimator. To that end, it reparameterizes the policy using a neural network transformation $a_t = f_\phi(\epsilon_t; s_t)$ where ϵ_t is an input noise vector, sampled from some fixed distribution, such as a spherical Gaussian. And then, policy can be rewritten as follows.

$$J_\pi(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}}[log\pi_\phi(f_\phi(\epsilon_t; s_t)|s_t) - Q_\theta(s_t, f_\phi(\epsilon_t; s_t))]$$

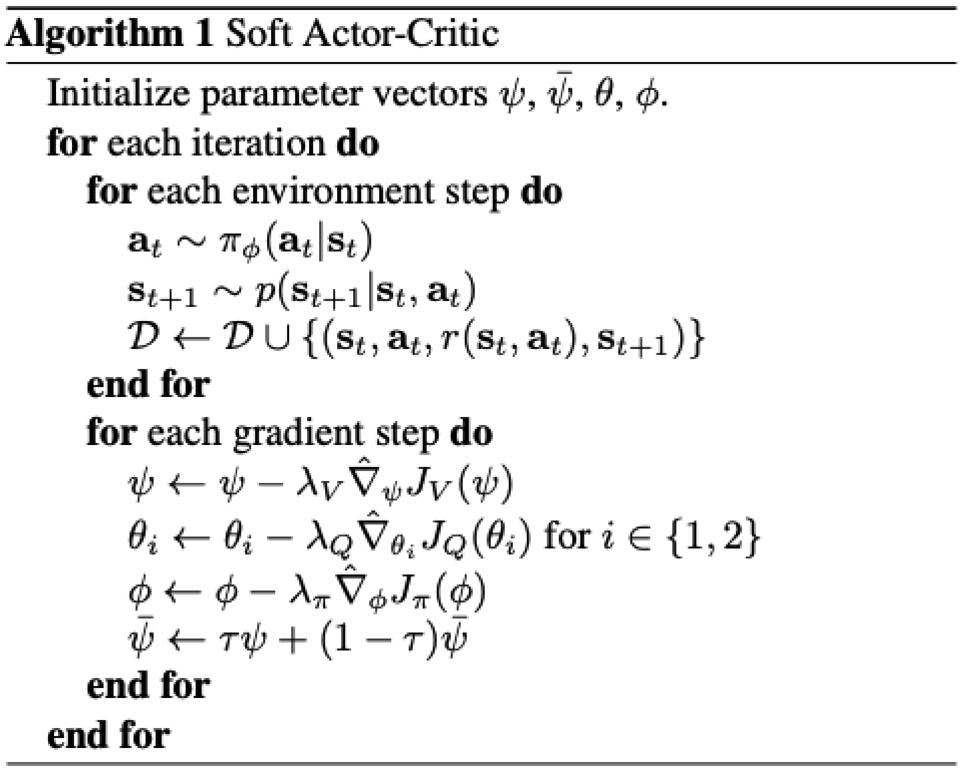
where π_ϕ is defined implicitly in terms of f_ϕ. Partition function is independent of ϕ and can be omitted. So, the equation of policy gradient approximation is as follows.

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi log\pi_\phi(a_t|s_t) + (\nabla_{a_t} log\pi_\phi(a_t|s_t) - \nabla_{a_t} Q(s_t, a_t))\nabla_\phi f_\phi(\epsilon_t; s_t) \quad (6)$$
$$where \quad a_t \; is \; evaluated \; at \; f_\phi(\epsilon_t; s_t) \quad (7)$$

This unbiased gradient estimator extends the DDPG style policy gradients to any tractable stochastic policy.

Additionally, in the policy improvement step, two Q-functions are used to mitigate the positive bias, which is a problem with the value-based algorithm. After going through training to optimize the two Q-functions independently, the minimum of Q-functions is used for calculating state value function gradient. Using two Q-functions like this, training speed can be faster.

SAC updates function estimators to the SGD method using samples extracted from the environment, policy, and replay buffer.



Algorithm 1 Soft Actor-Critic

```
Initialize parameter vectors $\psi$, $\bar{\psi}$, $\theta$, $\phi$.
for each iteration do
    for each environment step do
        $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
        $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
    end for
    for each gradient step do
        $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
        $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
        $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
        $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
    end for
end for
```

Implementation on JORLDY

- SAC JORLDY Implementation

```
### learn function ###
# 1. calculate target Q
# 2. calculate critic_loss and optimize critic_network
# 3. calculate actor_loss and optimize actor_network
# 4. calculate alpha(term of exploration)

def learn(self):
...
    q1 = self.critic1(state, action)
    q2 = self.critic2(state, action)

    with torch.no_grad():
        mu, std = self.actor(next_state)
        next_action, next_log_prob = self.sample_action(mu, std)
        next_q1 = self.target_critic1(next_state, next_action)
        next_q2 = self.target_critic2(next_state, next_action)
        min_next_q = torch.min(next_q1, next_q2)
        target_q = reward + (1 - done) * self.gamma * (
            min_next_q - self.alpha * next_log_prob
        )

    max_Q = torch.max(target_q, axis=0).values.cpu().numpy()[0]

    # Critic
    critic_loss1 = F.mse_loss(q1, target_q)
    critic_loss2 = F.mse_loss(q2, target_q)

    self.critic_optimizer1.zero_grad()
    critic_loss1.backward()
    self.critic_optimizer1.step()

    self.critic_optimizer2.zero_grad()
    critic_loss2.backward()
    self.critic_optimizer2.step()

    # Actor
    mu, std = self.actor(state)
    sample_action, log_prob = self.sample_action(mu, std)

    q1 = self.critic1(state, sample_action)
    q2 = self.critic2(state, sample_action)
    min_q = torch.min(q1, q2)

    actor_loss = ((self.alpha.to(self.device) * log_prob) - min_q).mean()
    self.actor_optimizer.zero_grad(set_to_none=True)
    actor_loss.backward()
    self.actor_optimizer.step()

    # Alpha
    alpha_loss = -(
        self.log_alpha * (log_prob + self.target_entropy).detach()
    ).mean()
    self.alpha = self.log_alpha.exp()

    if self.use_dynamic_alpha:
        self.alpha_optimizer.zero_grad(set_to_none=True)
        alpha_loss.backward()
        self.alpha_optimizer.step()

...
```

References

Relevant papers

- Soft Actor-Critic Algorithms and Applications
  (Haarnoja et al, 2018)
- Learning to Walk via Deep Reinforcement Learning
  (Haarnoja et al, 2018)

Public implementations

- SAC release repo
  (original “official” codebase)
- Softlearning repo
  (current “official” codebase)
- Yarats and Kostrikov repo