SAC is a off-policy alogrithm. SAC trains the agent by adding entropy of the action into reward. The optimization problem becomes

$$\pi^* = \text{argmax}_\pi \mathbb{E}_{\tau \sim \pi}[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t)))]$$

The value function of the states becomes

$$V^\pi(s) = \mathbb{E}_{\tau \sim \pi}[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(s)))]$$

The state action value is changed to include the entropy at each time step except the first one.

$$Q^\pi(s, a) = \mathbb{E}_{\tau \sim \pi}[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1})) + \sum_{t=1}^{\infty} \alpha \gamma^t H(\pi(s_t)) | s_0 = s, a_0 = a]$$

With these definition of V and Q, we have

$$V^\pi(s) = \mathbb{E}_{s' \in P}[R(s, a, s') + \gamma V^\pi(s')]$$

SAC learns a policy π_θ and two Q-functions Q_{ϕ_1} and Q_{ϕ_2}.
Similarities to DDPG and TD3

- both Q-functions are learned with MSE minimization, by regressing to a single shared target.
- the shared target is computed using target Q-networks and polyak interpolation,
- **clipped double-Q** trick is used to avoid overconfidence

Differences to DDPG and TD3

- target includes term from SAC's entropy regularization
- next state actions used in target come from **current** policy
- no explicit policy smooth (in DDPG and TD3 random noise is added to action for exploration). SAC trains a stochastic policy.

The entropy term and next state action term in Q^π are computed using the **current policy**. The next state s' is sampled from the replay buffer. But this means additional computed is need to sample the next action.
The estimate of Q-value (based one sample from experience replay) is

$$Q^\pi = r + \gamma(Q^\pi(s', \tilde{a'}) - \alpha \log \pi(\tilde{a'}|s')), \tilde{a'} \sim \pi(\cdot|s')$$

We use $\tilde{a'}$ to emphasize that next state action has to come from the current policy.

Update the Q-function
SAC trains 2 Q-function targets, the smaller one is used to compute the sample backup. (like in TD3)
The loss for the Q-networks in SAC are:

$$L(\phi_i, \mathscr{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathscr{D})}[(Q_{\phi_i}(s,a) - y(r,s',d))^2]$$

where

$$y(r,s',d) = r + \gamma(1-d)(\min_{j=1,2} Q_{\phi_{targ,j}}(s', \tilde{a'}) - \alpha \log \pi_\theta(\tilde{a'}|s'))$$

Update the policy
The policy should, in each state, act to maximize the expected future return plus the expected entropy,i.e. it should maximize $V^\pi(s)$

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}[Q^\pi(s,a)] + \alpha H(\pi(\cdot|s))$$
$$= \mathbb{E}_{a\sim\pi}[Q^\pi(s,a) - \alpha \log \pi(a|s)]$$

Use **reparametrization trick** for policy optimization, in which a sample from $\pi(\cdot|s)$ is drawn by computing a deterministic function of state, policy params, and independent noise. We squash the Gaussian policy through tanh fun

$$\tilde{a}_\theta(s,\zeta) = \tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \zeta), \zeta \sim N(0,1)$$

1. tanh changes the distribution of policy (no longer Gaussain), but one can still compute the log probability. If we don't squash the Gaussian dist, then it could have infinite entropy.
2. The standard deviation depends on state. In TRPO, PPO standard deviation can be independent from the state (verity it. In my implementation, std depends on input state). According to openai, SAC does not work if the std is independent from the input state. If entropy is part of learning objective, then naturally std should be a function of input state, because entropy is a function of std.
experiment with constant std
Why reparametrization is useful? It let us write expectation over ζ rather than action, which depends on the policy parameters.

$$\mathbb{E}_{a\sim\pi_\theta}[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)] = \mathbb{E}_{\zeta\sim N}[Q^{\pi_\theta}(s, \tilde{a}(s,\zeta)) - \alpha \log \pi_\theta(\tilde{a}(s,\zeta)|s)]$$

final step to get the policy loss is to replace Q^{π_θ} with one of our function approximators Q_{ϕ_i} (the less confident one). The optimization objective becomes

$$\max_\theta \mathbb{E}_{s\sim\mathscr{D},\zeta\sim N}[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\theta(s,\zeta)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\zeta)|s)]$$

it is almost similiar to DDPG's training objective.
Questions:
Why in target loss, the next state action needs to be sampled from the current policy? The intuition is with better policy, can you make a better estimate of state action value. But it behaves like a on-policy algorithm.

Algorithm 1: Soft Actor-Critic

```
Input: initial policy params $\theta$, $Q$-function params $\phi_1$ and $\phi_2$, empty replay buffer $\mathscr{D}$.
Set target parameters equal to main parameters $\phi_{targ,1} \leftarrow \phi_1, \phi_{targ,2} \leftarrow \phi_2$
repeat
  observe state $s$ and select action $a \sim \pi_\theta(\cdot|s)$
  execute $a$ in the env
  observe next state $s', r, d$ signal from env
  store $(s, a, r, s', d)$ in the buffer
  if $s'$ is terminal, then reset env
  if it's time to update then
    for j in range(num of updates) do
      randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathscr{D}$
      compute target for $Q$ functions

      $$y(r, s', d) = r+\gamma(1-d)(\min_{i=1,2} Q_{\phi_{targ,i}}(s', \tilde{a'})-\alpha \log \pi_\theta(\tilde{a'}|s')), \tilde{a'} \sim \pi_\theta(\cdot|s')$$

      update $Q$-fn by one step of gradient descent

      $$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s, a) - y(r, s', d))^2$$

      Update policy by one step gradien ascent

      $$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} (\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s))$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt $\theta$ via reparam trick.
      update target networks

      $$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1-\rho)\phi_i$$

    end for
  end if
until convergence
```