Soft Actor-Critic Demystified

An intuitive explanation of the theory and a PyTorch implementation guide

Vaishak V.Kumar Follow
Jan 8, 2019 · 11 min read

Soft Actor-Critic, the new Reinforcement Learning Algorithm from the folks at UC Berkley has been making a lot of noise recently. The algorithm not only boasts of being more sample efficient than traditional RL algorithms but also promises to be robust to brittleness in convergence. In this blog post, we'll dive deep into how the algorithm works and also implement it in PyTorch. This tutorial assumes that you are familiar with the problem specification and terminology of Reinforcement Learning. If you are not familiar with this or need a refresher, check out OpenAI's tutorial.

Before we begin, let's just take a quick look at why we care.

Not only does the Minotaur Robot learn in a really short time duration but it also learns to generalize to conditions that it hasn't seen during training! SAC thus brings us ever so close to using Reinforcement Learning in non-simulation environments for applications in robotics and other domains.

Some of the most successful RL algorithms in recent years such as Trust Region Policy Optimization (TRPO), Proximal Policy Optimization (PPO) and Asynchronous Actor-Critic Agents (A3C) suffer from sample inefficiency. This is because they learn in an “on-policy” manner, i.e. they need completely new samples after each policy update. In contrast, Q-learning based “off-policy” methods such as Deep Deterministic Policy Gradient (DDPG) and Twin Delayed Deep Deterministic Policy Gradient (TD3PG) are able to learn efficiently from past samples using experience replay buffers. However, the problem with these methods is that they are very sensitive to hyperparameters and require a lot of tuning to get them to converge. Soft Actor-Critic follows in the tradition of the latter type of algorithms and adds methods to combat the convergence brittleness. Let’s see how.

Theory

SAC is defined for RL tasks involving continuous actions. The biggest feature of SAC is that it uses a modified RL objective function. Instead of only seeking to maximize the lifetime rewards, SAC seeks to also maximize the entropy of the policy. The term ‘entropy’ has a rather esoteric definition and many interpretations depending on the application but I’d like to share an intuitive explanation here. We can think of entropy as how unpredictable a random variable is. If a random variable always takes a single value then it has zero entropy because it’s not unpredictable at all. If a random variable can be any Real Number with equal probability then it has very high entropy as it is very unpredictable. Why do we want our policy to have high entropy? We want a high entropy in our policy to explicitly encourage exploration, to encourage the policy to assign equal probabilities to actions that have same or nearly equal Q-values, and also to ensure that it does not collapse into repeatedly selecting a particular action that could exploit some inconsistency in the approximated Q function. Therefore, SAC overcomes the brittleness problem by encouraging the policy network to explore and not assign a very high probability to any one part of the range of actions.

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

Objective Function consisting of both a reward term and an entropy term H weighted by α

Now that we know what we are optimizing for, let us understand how we go about doing the optimization. SAC makes use of three networks: a state value function V parameterized by ψ, a soft Q-function Q parameterized by θ, and a policy function πparameterized by ϕ. While there is no need in principle to have separate approximators for the V and Q functions which are related through the policy, the authors say that in practice having separate function approximators help in convergence. So we need to train the three function approximators as follows:

1. We train the Value network by minimizing the following error:

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\psi(\mathbf{s}_t) - \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi} \left[Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \right] \right)^2 \right]$$

Don't get scared by this long error formula. All it's saying is that across all the states that we sample from our experience replay buffer, we need to decrease the squared difference between the prediction of our value network and the expected prediction of the Q function plus the entropy of the policy function π (measured here by the negative log of the policy function).

We'll use the below approximation of the derivative of the above objective to update the parameters of the V function:

$$\hat{\nabla}_\psi J_V(\psi) = \nabla_\psi V_\psi(\mathbf{s}_t) \left(V_\psi(\mathbf{s}_t) - Q_\theta(\mathbf{s}_t, \mathbf{a}_t) + \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \right)$$

2. We train the Q network by minimizing the following error:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \right)^2 \right]$$

where

$$\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \, \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right]$$

Minimizing this objective function amounts to the following: For all (state, action) pairs in the experience replay buffer, we want to minimize the squared difference between the prediction of our Q function and the immediate (one time-step) reward plus the discounted expected Value of the next state. Note that the Value comes from a Value function parameterized by ψ with a bar on top of it. This is an additional Value function called the target value function. We'll get into why we need this but for now, don't worry about it and just think of it as a Value function that we're training.

We'll use the below approximation of the derivative of the above objective is to update the parameters of the Q function:

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(\mathbf{a}_t, \mathbf{s}_t) \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - r(\mathbf{s}_t, \mathbf{a}_t) - \gamma V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right)$$

3. We train the Policy network π by minimizing the following error:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\mathrm{D}_{\mathrm{KL}} \left(\pi_\phi(\cdot | \mathbf{s}_t) \,\middle\|\, \frac{\exp\left(Q_\theta(\mathbf{s}_t, \cdot)\right)}{Z_\theta(\mathbf{s}_t)} \right) \right]$$

This objective function looks complex but it's actually saying something very simple. The DKL function that you see inside the expectation is called the Kullback-Leibler Divergence. I highly recommend that you read up on the KL divergence since it shows up a lot in deep learning research and applications these days. For the purposes of this tutorial, you can interpret it as how different the two distributions are. So, this objective function is basically trying to make the distribution of our Policy function look more like the distribution of the exponentiation of our Q Function normalized by another function Z.

In order to minimize this objective, the authors use something called the reparameterization trick. This trick is used to make sure that sampling from the policy is a differentiable process so that there are no problems in backpropagating the errors. The policy is now parameterized as follows:

$$\mathbf{a}_t = f_\phi(\epsilon_t; \mathbf{s}_t)$$

The epsilon term is a noise vector sampled from a Gaussian distribution. We will explain it more in the implementation section.

Now, we can express the objective function as follows:

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

The normalizing function Z is dropped since it does not depend on the parameter ϕ. An unbiased estimator for the gradient of the above objective is given as follows:

$$\begin{aligned} \hat{\nabla}_\phi J_\pi(\phi) = & \nabla_\phi \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \\ & + (\nabla_{\mathbf{a}_t} \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) - \nabla_{\mathbf{a}_t} Q(\mathbf{s}_t, \mathbf{a}_t)) \nabla_\phi f_\phi(\epsilon_t; \mathbf{s}_t) \end{aligned}$$

That's it for the math!

Implementation

Now that we understand the theory behind the algorithm, let's implement a version of it in Pytorch. My implementation is modeled on higgsfield's but with a critical change: I've used the reparameterization trick which makes training converge better due to lower variance. First off, let's look at the main body of the algorithm so that we understand what is happening at a high level so that we can then dive into the details of individual components.

```
env = NormalizedActions(gym.make("Pendulum-v0"))

action_dim = env.action_space.shape[0]
state_dim  = env.observation_space.shape[0]
hidden_dim = 256

value_net        = ValueNetwork(state_dim, hidden_dim).to(device)
target_value_net = ValueNetwork(state_dim, hidden_dim).to(device)

soft_q_net1 = SoftQNetwork(state_dim, action_dim, hidden_dim).to(device)
soft_q_net2 = SoftQNetwork(state_dim, action_dim,
hidden_dim).to(device)policy_net = PolicyNetwork(state_dim,

action_dim, hidden_dim).to(device)

for target_param, param in zip(target_value_net.parameters(),
```

```
value_net.parameters()):
    target_param.data.copy_(param.data)


value_criterion  = nn.MSELoss()
soft_q_criterion1 = nn.MSELoss()
soft_q_criterion2 = nn.MSELoss()

lr  = 3e-4

value_optimizer  = optim.Adam(value_net.parameters(), lr=lr)
soft_q_optimizer = optim.Adam(soft_q_net.parameters(), lr=lr)
policy_optimizer = optim.Adam(policy_net.parameters(), lr=lr)


replay_buffer_size = 1000000
replay_buffer = ReplayBuffer(replay_buffer_size)
```

First off, we initialize an OpenAI Gym environment in which our agent will play the Reinforcement Learning game. We store information about the dimension of the observations of the environment, the dimension of the action space, and then, set the hyperparameter of how many hidden layers we want in our networks. Then we initialize the three networks that we want to train along with a target V network. You will note that we have two Q networks. We maintain two Q networks to solve the problem of overestimation of Q-values. To combat this we maintain two Q networks and use the minimum of the two to do our policy and V function updates.

Now, it's time to explain the whole target V network business. The use of target networks is motivated by a problem in training V network. If you go back to the objective functions in the Theory section, you will find that the target for the Q network training depends on the V Network and the target for the V Network depends on the Q network (this makes sense because we are trying to enforce Bellman Consistency between the two functions). Because of this, the V network has a target that's indirectly dependent on itself which means that the V network's target depends on the same parameters we are trying to train. This makes training very unstable.

The solution is to use a set of parameters which comes close to the parameters of the main V network, but with a time delay. Thus we create a second network which lags the main network called the target network. There are two ways to go about this. The first way is to have the target network copied over from the main network regularly after a set number of

steps. The other way is to update the target network by Polyak averaging (a kind of moving averaging) itself and the main network. In this implementation, we use Polyak averaging. We initialize the main and target V networks to have the same parameters.

```
while frame_idx < max_frames:
    state = env.reset()
    episode_reward = 0

    for step in range(max_steps):
        if frame_idx >1000:
            action = policy_net.get_action(state).detach()
            next_state, reward, done, _ = env.step(action.numpy())
        else:
            action = env.action_space.sample()
            next_state, reward, done, _ = env.step(action)


        replay_buffer.push(state, action, reward, next_state, done)

        state = next_state
        episode_reward += reward
        frame_idx += 1

        if len(replay_buffer) > batch_size:
            update(batch_size)

        if frame_idx % 1000 == 0:
            plot(frame_idx, rewards)

        if done:
            break

    rewards.append(episode_reward)
```

We have nested loops here. The outer loop initializes the environment for the beginning of the episode. The inner loop is for the individual steps within an episode. In the inner loop, we sample an action from the Policy network — or randomly from the action space for the first few time steps— and record the state, action, reward, next state, and done — a variable indicating if we entered the terminal state of the episode — to the replay buffer. We do this till we have a minimum number of observations in the buffer. Then, we do network updates in each run of the inner loop after recording to the buffer.

The following is the code for the network update:

```
def update(batch_size,gamma=0.99,soft_tau=1e-2,):

    state, action, reward, next_state, done =
replay_buffer.sample(batch_size)

    state      = torch.FloatTensor(state).to(device)
    next_state = torch.FloatTensor(next_state).to(device)
    action     = torch.FloatTensor(action).to(device)
    reward     = torch.FloatTensor(reward).unsqueeze(1).to(device)
    done       =
torch.FloatTensor(np.float32(done)).unsqueeze(1).to(device)

    predicted_q_value1 = soft_q_net1(state, action)
    predicted_q_value2 = soft_q_net2(state, action)
    predicted_value    = value_net(state)
    new_action, log_prob, epsilon, mean, log_std =
policy_net.evaluate(state)

# Training Q Function
    target_value = target_value_net(next_state)
    target_q_value = reward + (1 - done) * gamma * target_value
    q_value_loss1 = soft_q_criterion1(predicted_q_value1,
target_q_value.detach())
    q_value_loss2 = soft_q_criterion2(predicted_q_value2,
target_q_value.detach())
    print("Q Loss")
    print(q_value_loss1)
    soft_q_optimizer1.zero_grad()
    q_value_loss1.backward()
    soft_q_optimizer1.step()
    soft_q_optimizer2.zero_grad()
    q_value_loss2.backward()
    soft_q_optimizer2.step()

# Training Value Function
    predicted_new_q_value = torch.min(soft_q_net1(state,
new_action),soft_q_net2(state, new_action))
    target_value_func = predicted_new_q_value - log_prob
    value_loss = value_criterion(predicted_value,
target_value_func.detach())
    print("V Loss")
    print(value_loss)
    value_optimizer.zero_grad()
    value_loss.backward()
    value_optimizer.step()

# Training Policy Function
    policy_loss = (log_prob - predicted_new_q_value).mean()

    policy_optimizer.zero_grad()
    policy_loss.backward()
    policy_optimizer.step()
```

```
    for target_param, param in zip(target_value_net.parameters(),
value_net.parameters()):
        target_param.data.copy_(
            target_param.data * (1.0 - soft_tau) + param.data * soft_tau
        )
```

First, we update the two Q function parameters by reducing the MSE between the predicted Q value for a state-action pair and its corresponding (reward + (1 — done) * gamma * target_value).

For the V network update, we take the minimum of the two Q values for a state-action pair and subtract from it the Policy's log probability of selecting that action in that state. Then we decrease the MSE between the above quantity and the predicted V value of that state.

Then, we update the Policy parameters by reducing the Policy's log probability of choosing an action in a state **log(π(S))** minus the predicted Q-value of that state-action pair. Note here that in this loss, the predicted q value is composed of the policy : **Q(S, π(S))**. This is important because it makes the term dependent on the Policy parameters ϕ.

Lastly, we update the Target Value Network by Polyak averaging it with the Main Value Network.

Next, let's take a quick look at the network structures:

```
class ValueNetwork(nn.Module):
    def __init__(self, state_dim, hidden_dim, init_w=3e-3):
        super(ValueNetwork, self).__init__()

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

        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)

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

```
class SoftQNetwork(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size, init_w=3e-
3):
        super(SoftQNetwork, self).__init__()

        self.linear1 = nn.Linear(num_inputs + num_actions, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, 1)

        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)

    def forward(self, state, action):
        x = torch.cat([state, action], 1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)
        return x


class PolicyNetwork(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size, init_w=3e-3,
log_std_min=-20, log_std_max=2):
        super(PolicyNetwork, self).__init__()

        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

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

        self.mean_linear = nn.Linear(hidden_size, num_actions)
        self.mean_linear.weight.data.uniform_(-init_w, init_w)
        self.mean_linear.bias.data.uniform_(-init_w, init_w)

        self.log_std_linear = nn.Linear(hidden_size, num_actions)
        self.log_std_linear.weight.data.uniform_(-init_w, init_w)
        self.log_std_linear.bias.data.uniform_(-init_w, init_w)

    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, self.log_std_min,
self.log_std_max)

        return mean, log_std

    def evaluate(self, state, epsilon=1e-6):
        mean, log_std = self.forward(state)
        std = log_std.exp()
```

```
        normal = Normal(0, 1)
        z      = normal.sample()
        action = torch.tanh(mean+ std*z.to(device))
        log_prob = Normal(mean, std).log_prob(mean+ std*z.to(device)) -
torch.log(1 - action.pow(2) + epsilon)
        return action, log_prob, z, mean, log_std


    def get_action(self, state):
        state = torch.FloatTensor(state).unsqueeze(0).to(device)
        mean, log_std = self.forward(state)
        std = log_std.exp()

        normal = Normal(0, 1)
        z      = normal.sample().to(device)
        action = torch.tanh(mean + std*z)

        action  = action.cpu()
        return action[0]
```

The Q and V networks are pretty standard so let's take a closer look at the Policy network. The policy has two outputs : the mean and the log standard deviation — we use log standard deviations since their exponential always gives a positive number. The log standard deviation is clamped to be in a sane region. Then to get the action, we use the reparameterization trick.

$$\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \qquad \xi \sim \mathcal{N}(0, I)$$

from Open AI Spinning Up

For this, we sample some noise from a Standard Normal distribution and multiply it with our standard deviation, and then add the result to the mean. Then this number is activated with a tanh function to give us our action. Finally, the log probability is calculated using an approximator of the log likelihood of tanh(mean + std* z).

That's all for the important implementation details! The full code can be found here. Make sure you run it and play around with the different hyperparameters to understand how they affect the training. I hope this has been helpful. Please send me any comments, corrections or links to any cool projects that you make using SAC!

UPDATE: Tuomas Haarnoja let me know over email that there is a new version of the algorithm that uses only a Q function and disposes of the V function. It also adds automatic

discovery of the weight of the entropy term called the ‘temperature’. You can check out the new paper in [2].

References

[1] T. Haarnoja et al. 2018. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*

[2] T. Haarnoja et al. 2018. *Soft Actor-Critic Algorithms and Applications*.