```python
from numbers import Number

import numpy as np
import tensorflow as tf

from rllab.core.serializable import Serializable
from rllab.misc import logger
from rllab.misc.overrides import overrides

from .base import RLAlgorithm


class SAC(RLAlgorithm, Serializable):
    """Soft Actor-Critic (SAC)

    Example:
    ```python

    env = normalize(SwimmerEnv())

    pool = SimpleReplayPool(env_spec=env.spec, max_pool_size=1E6)

    base_kwargs = dict(
        min_pool_size=1000,
        epoch_length=1000,
        n_epochs=1000,
        batch_size=64,
        scale_reward=1,
        n_train_repeat=1,
        eval_render=False,
        eval_n_episodes=1,
        eval_deterministic=True,
    )

    M = 100
    qf = NNQFunction(env_spec=env.spec, hidden_layer_sizes=(M, M))

    vf = NNVFunction(env_spec=env.spec, hidden_layer_sizes=(M, M))

    policy = GMMPolicy(
        env_spec=env.spec,
        K=2,
        hidden_layers=(M, M),
        qf=qf,
        reg=0.001
    )

    algorithm = SAC(
        base_kwargs=base_kwargs,
        env=env,
        policy=policy,
        pool=pool,
        qf=qf,
        vf=vf,

        lr=3E-4,
        discount=0.99,
        tau=0.01,

        save_full_state=False
    )

    algorithm.train()
    ```

    References
```

```
    ----------
    [1] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine, "Soft
        Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning
        with a Stochastic Actor," Deep Learning Symposium, NIPS 2017.
    """

    def __init__(
            self,
            base_kwargs,

            env,
            policy,
            initial_exploration_policy,
            qf1,
            qf2,
            vf,
            pool,
            plotter=None,

            lr=3e-3,
            scale_reward=1,
            discount=0.99,
            tau=0.01,
            target_update_interval=1,
            action_prior='uniform',
            reparameterize=False,

            save_full_state=False,
    ):
        """
        Args:
            base_kwargs (dict): dictionary of base arguments that are directly
                passed to the base `RLAlgorithm` constructor.

            env (`rllab.Env`): rllab environment object.
            policy: (`rllab.NNPolicy`): A policy function approximator.
            initial_exploration_policy: ('Policy'): A policy that we use
                for initial exploration which is not trained by the algorithm.

            qf1 (`valuefunction`): First Q-function approximator.
            qf2 (`valuefunction`): Second Q-function approximator. Usage of two
                Q-functions improves performance by reducing overestimation
                bias.
            vf (`ValueFunction`): Soft value function approximator.

            pool (`PoolBase`): Replay buffer to add gathered samples to.
            plotter (`QFPolicyPlotter`): Plotter instance to be used for
                visualizing Q-function during training.

            lr (`float`): Learning rate used for the function approximators.
            discount (`float`): Discount factor for Q-function updates.
            tau (`float`): Soft value function target update weight.
            target_update_interval ('int'): Frequency at which target network
                updates occur in iterations.

            reparameterize ('bool'): If True, we use a gradient estimator for
                the policy derived using the reparameterization trick. We use
                a likelihood ratio based estimator otherwise.
            save_full_state (`bool`): If True, save the full class in the
                snapshot. See `self.get_snapshot` for more information.
        """

        Serializable.quick_init(self, locals())
        super(SAC, self).__init__(**base_kwargs)

        self._env = env
```

```
        self._policy = policy
        self._initial_exploration_policy = initial_exploration_policy
        self._qf1 = qf1
        self._qf2 = qf2
        self._vf = vf
        self._pool = pool
        self._plotter = plotter
```

Algorithm 1 in the paper: ϕ θ ψ

```
        self._policy_lr = lr
        self._qf_lr = lr
        self._vf_lr = lr
        self._scale_reward = scale_reward
        self._discount = discount
        self._tau = tau
        self._target_update_interval = target_update_interval
        self._action_prior = action_prior

        # Reparameterize parameter must match between the algorithm and the
        # policy actions are sampled from.
        assert reparameterize == self._policy._reparameterize
        self._reparameterize = reparameterize

        self._save_full_state = save_full_state

        self._Da = self._env.action_space.flat_dim
        self._Do = self._env.observation_space.flat_dim

        self._training_ops = list()

        self._init_placeholders()
        self._init_actor_update()
        self._init_critic_update()
        self._init_target_ops()

        # Initialize all uninitialized variables. This prevents initializing
        # pre-trained policy and qf and vf variables.
        uninit_vars = []
        for var in tf.global_variables():
            try:
                self._sess.run(var)
            except tf.errors.FailedPreconditionError:
                uninit_vars.append(var)
        self._sess.run(tf.variables_initializer(uninit_vars))


    @overrides
    def train(self):
        """Initiate training of the SAC instance."""

        self._train(self._env, self._policy, self._initial_exploration_policy, self._pool)

    def _init_placeholders(self):
        """Create input placeholders for the SAC algorithm.

        Creates `tf.placeholder`s for:
            - observation
            - next observation
            - action
            - reward
            - terminals
        """
        self._iteration_pl = tf.placeholder(
            tf.int64, shape=None, name='iteration')

        self._observations_ph = tf.placeholder(
            tf.float32,
```

```
            shape=(None, self._Do),
            name='observation',
        )

        self._next_observations_ph = tf.placeholder(
            tf.float32,
            shape=(None, self._Do),
            name='next_observation',
        )
        self._actions_ph = tf.placeholder(
            tf.float32,
            shape=(None, self._Da),
            name='actions',
        )

        self._rewards_ph = tf.placeholder(
            tf.float32,
            shape=(None, ),
            name='rewards',
        )

        self._terminals_ph = tf.placeholder(
            tf.float32,
            shape=(None, ),
            name='terminals',
        )

    @property
    def scale_reward(self):
        if callable(self._scale_reward):
            return self._scale_reward(self._iteration_pl)
        elif isinstance(self._scale_reward, Number):
            return self._scale_reward

        raise ValueError(
            'scale_reward must be either callable or scalar')

    def _init_critic_update(self):
        """Create minimization operation for critic Q-function.

        Creates a `tf.optimizer.minimize` operation for updating
        critic Q-function with gradient descent, and appends it to
        `self._training_ops` attribute.

        See Equation (10) in [1], for further information of the
        Q-function update rule.
        """

        self._qf1_t = self._qf1.get_output_for(
            self._observations_ph, self._actions_ph, reuse=True)  # N
        self._qf2_t = self._qf2.get_output_for(
            self._observations_ph, self._actions_ph, reuse=True)  # N

        with tf.variable_scope('target'):
            vf_next_target_t = self._vf.get_output_for(self._next_observations_ph)  # N
            self._vf_target_params = self._vf.get_params_internal()

        ys = tf.stop_gradient(
            self.scale_reward * self._rewards_ph +
            (1 - self._terminals_ph) * self._discount * vf_next_target_t
        )  # N

        self._td_loss1_t = 0.5 * tf.reduce_mean((ys - self._qf1_t)**2)
        self._td_loss2_t = 0.5 * tf.reduce_mean((ys - self._qf2_t)**2)

        qf1_train_op = tf.train.AdamOptimizer(self._qf_lr).minimize(
```

Equation (7)

→ Remember the use of two networks, and we use the min of the two estimates (below eqn 13 in paper)

```python
            loss=self._td_loss1_t,
            var_list=self._qf1.get_params_internal()
        )
        qf2_train_op = tf.train.AdamOptimizer(self._qf_lr).minimize(
            loss=self._td_loss2_t,
            var_list=self._qf2.get_params_internal()
        )

        self._training_ops.append(qf1_train_op)
        self._training_ops.append(qf2_train_op)

    def _init_actor_update(self):
        """Create minimization operations for policy and state value functions.

        Creates a `tf.optimizer.minimize` operations for updating
        policy and value functions with gradient descent, and appends them to
        `self._training_ops` attribute.

        In principle, there is no need for a separate state value function
        approximator, since it could be evaluated using the Q-function and
        policy. However, in practice, the separate function approximator
        stabilizes training.

        See Equations (8, 13) in [1], for further information
        of the value function and policy function update rules.
        """

        actions, log_pi = self._policy.actions_for(observations=self._observations_ph,
                                                   with_log_pis=True)

        self._vf_t = self._vf.get_output_for(self._observations_ph, reuse=True)  # N
        self._vf_params = self._vf.get_params_internal()

        if self._action_prior == 'normal':
            D_s = actions.shape.as_list()[-1]
            policy_prior = tf.contrib.distributions.MultivariateNormalDiag(
                loc=tf.zeros(D_s), scale_diag=tf.ones(D_s))
            policy_prior_log_probs = policy_prior.log_prob(actions)
        elif self._action_prior == 'uniform':
            policy_prior_log_probs = 0.0

        log_target1 = self._qf1.get_output_for(
            self._observations_ph, actions, reuse=True)  # N
        log_target2 = self._qf2.get_output_for(
            self._observations_ph, actions, reuse=True)  # N
        min_log_target = tf.minimum(log_target1, log_target2)

        if self._reparameterize:
            policy_kl_loss = tf.reduce_mean(log_pi - log_target1)
        else:
            policy_kl_loss = tf.reduce_mean(log_pi * tf.stop_gradient(
                log_pi - log_target1 + self._vf_t - policy_prior_log_probs))

        policy_regularization_losses = tf.get_collection(
            tf.GraphKeys.REGULARIZATION_LOSSES,
            scope=self._policy.name)
        policy_regularization_loss = tf.reduce_sum(
            policy_regularization_losses)

        policy_loss = (policy_kl_loss
                       + policy_regularization_loss)

        # We update the vf towards the min of two Q-functions in order to
        # reduce overestimation bias from function approximation error.
        self._vf_loss_t = 0.5 * tf.reduce_mean((
          self._vf_t
```

Equation (12), $\log\pi - Q_\theta$

```
            - tf.stop_gradient(min_log_target - log_pi + policy_prior_log_probs)
        )**2)
```

Equation (5)

```
        policy_train_op = tf.train.AdamOptimizer(self._policy_lr).minimize(
            loss=policy_loss,
            var_list=self._policy.get_params_internal()
        )

        vf_train_op = tf.train.AdamOptimizer(self._vf_lr).minimize(
            loss=self._vf_loss_t,
            var_list=self._vf_params
        )

        self._training_ops.append(policy_train_op)
        self._training_ops.append(vf_train_op)

    def _init_target_ops(self):
        """Create tensorflow operations for updating target value function."""

        source_params = self._vf_params
        target_params = self._vf_target_params
```

soft updates of target networks

```
        self._target_ops = [
            tf.assign(target, (1 - self._tau) * target + self._tau * source)
            for target, source in zip(target_params, source_params)
        ]

    @overrides
    def _init_training(self, env, policy, pool):
        super(SAC, self)._init_training(env, policy, pool)
        self._sess.run(self._target_ops)

    @overrides
    def _do_training(self, iteration, batch):
        """Runs the operations for updating training and target ops."""

        feed_dict = self._get_feed_dict(iteration, batch)
        self._sess.run(self._training_ops, feed_dict)

        if iteration % self._target_update_interval == 0:
            # Run target ops here.
            self._sess.run(self._target_ops)

    def _get_feed_dict(self, iteration, batch):
        """Construct TensorFlow feed_dict from sample batch."""

        feed_dict = {
            self._observations_ph: batch['observations'],
            self._actions_ph: batch['actions'],
            self._next_observations_ph: batch['next_observations'],
            self._rewards_ph: batch['rewards'],
            self._terminals_ph: batch['terminals'],
        }

        if iteration is not None:
            feed_dict[self._iteration_pl] = iteration

        return feed_dict

    @overrides
    def log_diagnostics(self, iteration, batch):
        """Record diagnostic information to the logger.

        Records mean and standard deviation of Q-function and state
        value function, and TD-loss (mean squared Bellman error)
        for the sample batch.
```

```
        Also calls the `draw` method of the plotter, if plotter defined.
        """

        feed_dict = self._get_feed_dict(iteration, batch)
        qf1, qf2, vf, td_loss1, td_loss2 = self._sess.run(
            (self._qf1_t, self._qf2_t, self._vf_t, self._td_loss1_t, self._td_loss2_t),
feed_dict)

        logger.record_tabular('qf1-avg', np.mean(qf1))
        logger.record_tabular('qf1-std', np.std(qf1))
        logger.record_tabular('qf2-avg', np.mean(qf1))
        logger.record_tabular('qf2-std', np.std(qf1))
        logger.record_tabular('mean-qf-diff', np.mean(np.abs(qf1-qf2)))
        logger.record_tabular('vf-avg', np.mean(vf))
        logger.record_tabular('vf-std', np.std(vf))
        logger.record_tabular('mean-sq-bellman-error1', td_loss1)
        logger.record_tabular('mean-sq-bellman-error2', td_loss2)

        self._policy.log_diagnostics(iteration, batch)
        if self._plotter:
            self._plotter.draw()

    @overrides
    def get_snapshot(self, epoch):
        """Return loggable snapshot of the SAC algorithm.

        If `self._save_full_state == True`, returns snapshot of the complete
        SAC instance. If `self._save_full_state == False`, returns snapshot
        of policy, Q-function, state value function, and environment instances.
        """

        if self._save_full_state:
            snapshot = {
                'epoch': epoch,
                'algo': self
            }
        else:
            snapshot = {
                'epoch': epoch,
                'policy': self._policy,
                'qf1': self._qf1,
                'qf2': self._qf2,
                'vf': self._vf,
                'env': self._env,
            }

        return snapshot

    def __getstate__(self):
        """Get Serializable state of the RLALgorithm instance."""

        d = Serializable.__getstate__(self)
        d.update({
            'qf1-params': self._qf1.get_param_values(),
            'qf2-params': self._qf2.get_param_values(),
            'vf-params': self._vf.get_param_values(),
            'policy-params': self._policy.get_param_values(),
            'pool': self._pool.__getstate__(),
            'env': self._env.__getstate__(),
        })
        return d

    def __setstate__(self, d):
        """Set Serializable state fo the RLAlgorithm instance."""
```

```
        Serializable.__setstate__(self, d)
        self._qf1.set_param_values(d['qf1-params'])
        self._qf2.set_param_values(d['qf2-params'])
        self._vf.set_param_values(d['vf-params'])
        self._policy.set_param_values(d['policy-params'])
        self._pool.__setstate__(d['pool'])
        self._env.__setstate__(d['env'])
```