```
from typing import Any, Dict, Optional, Type, TypeVar, Union

import torch as th
from gym import spaces
from torch.nn import functional as F

from stable_baselines3.common.on_policy_algorithm import OnPolicyAlgorithm
from stable_baselines3.common.policies import ActorCriticCnnPolicy, ActorCriticPolicy,
BasePolicy, MultiInputActorCriticPolicy
from stable_baselines3.common.type_aliases import GymEnv, MaybeCallback, Schedule
from stable_baselines3.common.utils import explained_variance

SelfA2C = TypeVar("SelfA2C", bound="A2C")


class A2C(OnPolicyAlgorithm):
    """
    Advantage Actor Critic (A2C)

    Paper: https://arxiv.org/abs/1602.01783
    Code: This implementation borrows code from https://github.com/ikostrikov/pytorch-a2c-
ppo-acktr-gail and
    and Stable Baselines (https://github.com/hill-a/stable-baselines)

    Introduction to A2C: https://hackernoon.com/intuitive-rl-intro-to-advantage-actor-
critic-a2c-4ff545978752

    :param policy: The policy model to use (MlpPolicy, CnnPolicy, ...)
    :param env: The environment to learn from (if registered in Gym, can be str)
    :param learning_rate: The learning rate, it can be a function
        of the current progress remaining (from 1 to 0)
    :param n_steps: The number of steps to run for each environment per update
        (i.e. batch size is n_steps * n_env where n_env is number of environment copies
running in parallel)
    :param gamma: Discount factor
    :param gae_lambda: Factor for trade-off of bias vs variance for Generalized Advantage
Estimator.
        Equivalent to classic advantage when set to 1.
    :param ent_coef: Entropy coefficient for the loss calculation
    :param vf_coef: Value function coefficient for the loss calculation
    :param max_grad_norm: The maximum value for the gradient clipping
    :param rms_prop_eps: RMSProp epsilon. It stabilizes square root computation in
denominator
        of RMSProp update
    :param use_rms_prop: Whether to use RMSprop (default) or Adam as optimizer
    :param use_sde: Whether to use generalized State Dependent Exploration (gSDE)
        instead of action noise exploration (default: False)
    :param sde_sample_freq: Sample a new noise matrix every n steps when using gSDE
        Default: -1 (only sample at the beginning of the rollout)
    :param normalize_advantage: Whether to normalize or not the advantage
    :param tensorboard_log: the log location for tensorboard (if None, no logging)
    :param policy_kwargs: additional arguments to be passed to the policy on creation
    :param verbose: Verbosity level: 0 for no output, 1 for info messages (such as device
or wrappers used), 2 for
        debug messages
    :param seed: Seed for the pseudo random generators
    :param device: Device (cpu, cuda, ...) on which the code should be run.
        Setting it to auto, the code will be run on the GPU if possible.
    :param _init_setup_model: Whether or not to build the network at the creation of the
instance
    """

    policy_aliases: Dict[str, Type[BasePolicy]] = {
        "MlpPolicy": ActorCriticPolicy,
        "CnnPolicy": ActorCriticCnnPolicy,
        "MultiInputPolicy": MultiInputActorCriticPolicy,
    }

    def __init__(
        self,
        policy: Union[str, Type[ActorCriticPolicy]],
        env: Union[GymEnv, str],
        learning_rate: Union[float, Schedule] = 7e-4,
        n_steps: int = 5,
        gamma: float = 0.99,
        gae_lambda: float = 1.0,
        ent_coef: float = 0.0,
        vf_coef: float = 0.5,
        max_grad_norm: float = 0.5,
        rms_prop_eps: float = 1e-5,
        use_rms_prop: bool = True,
        use_sde: bool = False,
        sde_sample_freq: int = -1,
        normalize_advantage: bool = False,
        tensorboard_log: Optional[str] = None,
        policy_kwargs: Optional[Dict[str, Any]] = None,
        verbose: int = 0,
        seed: Optional[int] = None,
        device: Union[th.device, str] = "auto",
        _init_setup_model: bool = True,
    ):
        super().__init__(
            policy,
            env,
            learning_rate=learning_rate,
            n_steps=n_steps,
            gamma=gamma,
            gae_lambda=gae_lambda,
            ent_coef=ent_coef,
            vf_coef=vf_coef,
            max_grad_norm=max_grad_norm,
            use_sde=use_sde,
            sde_sample_freq=sde_sample_freq,
            tensorboard_log=tensorboard_log,
            policy_kwargs=policy_kwargs,
            verbose=verbose,
            device=device,
            seed=seed,
            _init_setup_model=False,
            supported_action_spaces=(
                spaces.Box,
                spaces.Discrete,
                spaces.MultiDiscrete,
                spaces.MultiBinary,
            ),
        )

        self.normalize_advantage = normalize_advantage

        # Update optimizer inside the policy if we want to use RMSProp
        # (original implementation) rather than Adam
        if use_rms_prop and "optimizer_class" not in self.policy_kwargs:
            self.policy_kwargs["optimizer_class"] = th.optim.RMSprop
            self.policy_kwargs["optimizer_kwargs"] = dict(alpha=0.99, eps=rms_prop_eps,
weight_decay=0)

        if _init_setup_model:
            self._setup_model()

    def train(self) -> None:
        """
        Update policy using the currently gathered
        rollout buffer (one gradient step over whole data).
        """
        # Switch to train mode (this affects batch norm / dropout)
        self.policy.set_training_mode(True)

        # Update optimizer learning rate
        self._update_learning_rate(self.policy.optimizer)

        # This will only loop once (get all data in one go)
        for rollout_data in self.rollout_buffer.get(batch_size=None):
            actions = rollout_data.actions
            if isinstance(self.action_space, spaces.Discrete):
                # Convert discrete action from float to long
                actions = actions.long().flatten()

            values, log_prob, entropy =
self.policy.evaluate_actions(rollout_data.observations, actions)
            values = values.flatten()

            # Normalize advantage (not present in the original implementation)
            advantages = rollout_data.advantages
            if self.normalize_advantage:
                advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

            # Policy gradient loss
            policy_loss = -(advantages * log_prob).mean()

            # Value loss using the TD(gae_lambda) target
            value_loss = F.mse_loss(rollout_data.returns, values)

            # Entropy loss favor exploration
            if entropy is None:
                # Approximate entropy when no analytical form
                entropy_loss = -th.mean(-log_prob)
            else:
                entropy_loss = -th.mean(entropy)

            loss = policy_loss + self.ent_coef * entropy_loss + self.vf_coef * value_loss

            # Optimization step
            self.policy.optimizer.zero_grad()
            loss.backward()

            # Clip grad norm
            th.nn.utils.clip_grad_norm_(self.policy.parameters(), self.max_grad_norm)
            self.policy.optimizer.step()

        explained_var = explained_variance(self.rollout_buffer.values.flatten(),
self.rollout_buffer.returns.flatten())

        self._n_updates += 1
        self.logger.record("train/n_updates", self._n_updates, exclude="tensorboard")
        self.logger.record("train/explained_variance", explained_var)
        self.logger.record("train/entropy_loss", entropy_loss.item())
        self.logger.record("train/policy_loss", policy_loss.item())
        self.logger.record("train/value_loss", value_loss.item())
        if hasattr(self.policy, "log_std"):
            self.logger.record("train/std", th.exp(self.policy.log_std).mean().item())

    def learn(
        self: SelfA2C,
        total_timesteps: int,
        callback: MaybeCallback = None,
        log_interval: int = 100,
        tb_log_name: str = "A2C",
        reset_num_timesteps: bool = True,
```

Given that there is a single call, the critic + actor are in a single architecture

optional to include entropy

```
        progress_bar: bool = False,
    ) -> SelfA2C:
        return super().learn(
            total_timesteps=total_timesteps,
            callback=callback,
            log_interval=log_interval,
            tb_log_name=tb_log_name,
            reset_num_timesteps=reset_num_timesteps,
            progress_bar=progress_bar,
        )
```