```
from typing import Any, Mapping, Optional, Sequence, Union

import copy
import itertools
import gymnasium
from packaging import version

import torch
import torch.nn as nn
import torch.nn.functional as F

from skrl import config, logger
from skrl.memories.torch import Memory
from skrl.models.torch import Model
from skrl.multi_agents.torch import MultiAgent
from skrl.resources.schedulers.torch import KLAdaptiveLR


# fmt: off
# [start-config-dict-torch]
MAPPO_DEFAULT_CONFIG = {
    "rollouts": 16,                 # number of rollouts before updating
    "learning_epochs": 8,           # number of learning epochs during each
update
    "mini_batches": 2,              # number of mini batches during each
learning epoch

    "discount_factor": 0.99,        # discount factor (gamma)
    "lambda": 0.95,                 # TD(lambda) coefficient (lam) for computing
returns and advantages

    "learning_rate": 1e-3,                  # learning rate
    "learning_rate_scheduler": None,        # learning rate scheduler class (see
torch.optim.lr_scheduler)
    "learning_rate_scheduler_kwargs": {},   # learning rate scheduler's kwargs
(e.g. {"step_size": 1e-3})

    "state_preprocessor": None,             # state preprocessor class (see
skrl.resources.preprocessors)
    "state_preprocessor_kwargs": {},        # state preprocessor's kwargs (e.g.
{"size": env.observation_space})
    "shared_state_preprocessor": None,      # shared state preprocessor class
(see skrl.resources.preprocessors)
    "shared_state_preprocessor_kwargs": {}, # shared state preprocessor's kwargs
(e.g. {"size": env.shared_observation_space})
    "value_preprocessor": None,             # value preprocessor class (see
skrl.resources.preprocessors)
    "value_preprocessor_kwargs": {},        # value preprocessor's kwargs (e.g.
{"size": 1})

    "random_timesteps": 0,          # random exploration steps
    "learning_starts": 0,           # learning starts after this many steps

    "grad_norm_clip": 0.5,              # clipping coefficient for the norm of
the gradients
    "ratio_clip": 0.2,                  # clipping coefficient for computing the
clipped surrogate objective
    "value_clip": 0.2,                  # clipping coefficient for computing the
value loss (if clip_predicted_values is True)
    "clip_predicted_values": False,     # clip predicted values during value
loss computation

    "entropy_loss_scale": 0.0,      # entropy loss scaling factor
    "value_loss_scale": 1.0,        # value loss scaling factor
```

```
    "kl_threshold": 0,              # KL divergence threshold for early stopping

    "rewards_shaper": None,         # rewards shaping function: Callable(reward,
timestep, timesteps) -> reward
    "time_limit_bootstrap": False,  # bootstrap at timeout termination (episode
truncation)

    "mixed_precision": False,       # enable automatic mixed precision for
higher performance

    "experiment": {
        "directory": "",            # experiment's parent directory
        "experiment_name": "",      # experiment name
        "write_interval": "auto",   # TensorBoard writing interval (timesteps)

        "checkpoint_interval": "auto",      # interval for checkpoints
(timesteps)
        "store_separately": False,          # whether to store checkpoints
separately

        "wandb": False,             # whether to use Weights & Biases
        "wandb_kwargs": {}          # wandb kwargs (see
https://docs.wandb.ai/ref/python/init)
    }
}
# [end-config-dict-torch]
# fmt: on


class MAPPO(MultiAgent):
    def __init__(
        self,
        possible_agents: Sequence[str],
        models: Mapping[str, Model],
        memories: Optional[Mapping[str, Memory]] = None,
        observation_spaces: Optional[Union[Mapping[str, int], Mapping[str,
gymnasium.Space]]] = None,
        action_spaces: Optional[Union[Mapping[str, int], Mapping[str,
gymnasium.Space]]] = None,
        device: Optional[Union[str, torch.device]] = None,
        cfg: Optional[dict] = None,
        shared_observation_spaces: Optional[Union[Mapping[str, int],
Mapping[str, gymnasium.Space]]] = None,
    ) -> None:
        """Multi-Agent Proximal Policy Optimization (MAPPO)

        https://arxiv.org/abs/2103.01955

        :param possible_agents: Name of all possible agents the environment
could generate
        :type possible_agents: list of str
        :param models: Models used by the agents.
                       External keys are environment agents' names. Internal
keys are the models required by the algorithm
        :type models: nested dictionary of skrl.models.torch.Model
        :param memories: Memories to storage the transitions.
        :type memories: dictionary of skrl.memory.torch.Memory, optional
        :param observation_spaces: Observation/state spaces or shapes (default:
``None``)
        :type observation_spaces: dictionary of int, sequence of int or
gymnasium.Space, optional
        :param action_spaces: Action spaces or shapes (default: ``None``)
        :type action_spaces: dictionary of int, sequence of int or
```

```
gymnasium.Space, optional
        :param device: Device on which a tensor/array is or will be allocated
(default: ``None``).
                       If None, the device will be either ``"cuda"`` if
available or ``"cpu"``
        :type device: str or torch.device, optional
        :param cfg: Configuration dictionary
        :type cfg: dict
        :param shared_observation_spaces: Shared observation/state space or
shape (default: ``None``)
        :type shared_observation_spaces: dictionary of int, sequence of int or
gymnasium.Space, optional
        """
        _cfg = copy.deepcopy(MAPPO_DEFAULT_CONFIG)
        _cfg.update(cfg if cfg is not None else {})
        super().__init__(
            possible_agents=possible_agents,
            models=models,
            memories=memories,
            observation_spaces=observation_spaces,
            action_spaces=action_spaces,
            device=device,
            cfg=_cfg,
        )

        self.shared_observation_spaces = shared_observation_spaces

        # models
        self.policies = {uid: self.models[uid].get("policy", None) for uid in
self.possible_agents}
        self.values = {uid: self.models[uid].get("value", None) for uid in
self.possible_agents}

        for uid in self.possible_agents:
            # checkpoint models
            self.checkpoint_modules[uid]["policy"] = self.policies[uid]
            self.checkpoint_modules[uid]["value"] = self.values[uid]

            # broadcast models' parameters in distributed runs
            if config.torch.is_distributed:
                logger.info(f"Broadcasting models' parameters")
                if self.policies[uid] is not None:
                    self.policies[uid].broadcast_parameters()
                    if self.values[uid] is not None and self.policies[uid] is
not self.values[uid]:
                        self.values[uid].broadcast_parameters()

        # configuration
        self._learning_epochs = self._as_dict(self.cfg["learning_epochs"])
        self._mini_batches = self._as_dict(self.cfg["mini_batches"])
        self._rollouts = self.cfg["rollouts"]
        self._rollout = 0

        self._grad_norm_clip = self._as_dict(self.cfg["grad_norm_clip"])
        self._ratio_clip = self._as_dict(self.cfg["ratio_clip"])
        self._value_clip = self._as_dict(self.cfg["value_clip"])
        self._clip_predicted_values =
self._as_dict(self.cfg["clip_predicted_values"])

        self._value_loss_scale = self._as_dict(self.cfg["value_loss_scale"])
        self._entropy_loss_scale = self._as_dict(self.cfg["entropy_loss_scale"])

        self._kl_threshold = self._as_dict(self.cfg["kl_threshold"])
```

```
        self._learning_rate = self._as_dict(self.cfg["learning_rate"])
        self._learning_rate_scheduler =
self._as_dict(self.cfg["learning_rate_scheduler"])
        self._learning_rate_scheduler_kwargs =
self._as_dict(self.cfg["learning_rate_scheduler_kwargs"])

        self._state_preprocessor = self._as_dict(self.cfg["state_preprocessor"])
        self._state_preprocessor_kwargs =
self._as_dict(self.cfg["state_preprocessor_kwargs"])
        self._shared_state_preprocessor =
self._as_dict(self.cfg["shared_state_preprocessor"])
        self._shared_state_preprocessor_kwargs =
self._as_dict(self.cfg["shared_state_preprocessor_kwargs"])
        self._value_preprocessor = self._as_dict(self.cfg["value_preprocessor"])
        self._value_preprocessor_kwargs =
self._as_dict(self.cfg["value_preprocessor_kwargs"])

        self._discount_factor = self._as_dict(self.cfg["discount_factor"])
        self._lambda = self._as_dict(self.cfg["lambda"])

        self._random_timesteps = self.cfg["random_timesteps"]
        self._learning_starts = self.cfg["learning_starts"]

        self._rewards_shaper = self.cfg["rewards_shaper"]
        self._time_limit_bootstrap =
self._as_dict(self.cfg["time_limit_bootstrap"])

        self._mixed_precision = self.cfg["mixed_precision"]

        # set up automatic mixed precision
        self._device_type = torch.device(device).type
        if version.parse(torch.__version__) >= version.parse("2.4"):
            self.scaler = torch.amp.GradScaler(device=self._device_type,
enabled=self._mixed_precision)
        else:
            self.scaler =
torch.cuda.amp.GradScaler(enabled=self._mixed_precision)

        # set up optimizer and learning rate scheduler
        self.optimizers = {}
        self.schedulers = {}

        for uid in self.possible_agents:
            policy = self.policies[uid]
            value = self.values[uid]
            if policy is not None and value is not None:
                if policy is value:
                    optimizer = torch.optim.Adam(policy.parameters(),
lr=self._learning_rate[uid])
                else:
                    optimizer = torch.optim.Adam(
                        itertools.chain(policy.parameters(),
value.parameters()), lr=self._learning_rate[uid]
                    )
                self.optimizers[uid] = optimizer
                if self._learning_rate_scheduler[uid] is not None:
                    self.schedulers[uid] = self._learning_rate_scheduler[uid](
                        optimizer, **self._learning_rate_scheduler_kwargs[uid]
                    )

            self.checkpoint_modules[uid]["optimizer"] = self.optimizers[uid]

            # set up preprocessors
            if self._state_preprocessor[uid] is not None:
```

```
                self._state_preprocessor[uid] = self._state_preprocessor[uid]
(**self._state_preprocessor_kwargs[uid])
                self.checkpoint_modules[uid]["state_preprocessor"] =
self._state_preprocessor[uid]
            else:
                self._state_preprocessor[uid] = self._empty_preprocessor

            if self._shared_state_preprocessor[uid] is not None:
                self._shared_state_preprocessor[uid] =
self._shared_state_preprocessor[uid](
                    **self._shared_state_preprocessor_kwargs[uid]
                )
                self.checkpoint_modules[uid]["shared_state_preprocessor"] =
self._shared_state_preprocessor[uid]
            else:
                self._shared_state_preprocessor[uid] = self._empty_preprocessor

            if self._value_preprocessor[uid] is not None:
                self._value_preprocessor[uid] = self._value_preprocessor[uid]
(**self._value_preprocessor_kwargs[uid])
                self.checkpoint_modules[uid]["value_preprocessor"] =
self._value_preprocessor[uid]
            else:
                self._value_preprocessor[uid] = self._empty_preprocessor

    def init(self, trainer_cfg: Optional[Mapping[str, Any]] = None) -> None:
        """Initialize the agent"""
        super().init(trainer_cfg=trainer_cfg)
        self.set_mode("eval")

        # create tensors in memories
        if self.memories:
            for uid in self.possible_agents:
                self.memories[uid].create_tensor(name="states",
size=self.observation_spaces[uid], dtype=torch.float32)
                self.memories[uid].create_tensor(
                    name="shared_states",
size=self.shared_observation_spaces[uid], dtype=torch.float32
                )
                self.memories[uid].create_tensor(name="actions",
size=self.action_spaces[uid], dtype=torch.float32)
                self.memories[uid].create_tensor(name="rewards", size=1,
dtype=torch.float32)
                self.memories[uid].create_tensor(name="terminated", size=1,
dtype=torch.bool)
                self.memories[uid].create_tensor(name="truncated", size=1,
dtype=torch.bool)
                self.memories[uid].create_tensor(name="log_prob", size=1,
dtype=torch.float32)
                self.memories[uid].create_tensor(name="values", size=1,
dtype=torch.float32)
                self.memories[uid].create_tensor(name="returns", size=1,
dtype=torch.float32)
                self.memories[uid].create_tensor(name="advantages", size=1,
dtype=torch.float32)

                # tensors sampled during training
                self._tensors_names = [
                    "states",
                    "shared_states",
                    "actions",
                    "log_prob",
                    "values",
                    "returns",
```

```
                    "advantages",
                ]

        # create temporary variables needed for storage and computation
        self._current_log_prob = []
        self._current_shared_next_states = []

    def act(self, states: Mapping[str, torch.Tensor], timestep: int, timesteps:
int) -> torch.Tensor:
        """Process the environment's states to make a decision (actions) using
the main policies

        :param states: Environment's states
        :type states: dictionary of torch.Tensor
        :param timestep: Current timestep
        :type timestep: int
        :param timesteps: Number of timesteps
        :type timesteps: int

        :return: Actions
        :rtype: torch.Tensor
        """
        # # sample random actions
        # # TODO: fix for stochasticity, rnn and log_prob
        # if timestep < self._random_timesteps:
        #     return self.policy.random_act({"states": states}, role="policy")

        # sample stochastic actions
        with torch.autocast(device_type=self._device_type,
enabled=self._mixed_precision):
            data = [
                self.policies[uid].act({"states": self._state_preprocessor[uid]
(states[uid])}, role="policy")
                for uid in self.possible_agents
            ]

            actions = {uid: d[0] for uid, d in zip(self.possible_agents, data)}
            log_prob = {uid: d[1] for uid, d in zip(self.possible_agents, data)}
            outputs = {uid: d[2] for uid, d in zip(self.possible_agents, data)}

            self._current_log_prob = log_prob

        return actions, log_prob, outputs

    def record_transition(
        self,
        states: Mapping[str, torch.Tensor],
        actions: Mapping[str, torch.Tensor],
        rewards: Mapping[str, torch.Tensor],
        next_states: Mapping[str, torch.Tensor],
        terminated: Mapping[str, torch.Tensor],
        truncated: Mapping[str, torch.Tensor],
        infos: Mapping[str, Any],
        timestep: int,
        timesteps: int,
    ) -> None:
        """Record an environment transition in memory

        :param states: Observations/states of the environment used to make the
decision
        :type states: dictionary of torch.Tensor
        :param actions: Actions taken by the agent
        :type actions: dictionary of torch.Tensor
        :param rewards: Instant rewards achieved by the current actions
```

```
        :type rewards: dictionary of torch.Tensor
        :param next_states: Next observations/states of the environment
        :type next_states: dictionary of torch.Tensor
        :param terminated: Signals to indicate that episodes have terminated
        :type terminated: dictionary of torch.Tensor
        :param truncated: Signals to indicate that episodes have been truncated
        :type truncated: dictionary of torch.Tensor
        :param infos: Additional information about the environment
        :type infos: dictionary of any supported type
        :param timestep: Current timestep
        :type timestep: int
        :param timesteps: Number of timesteps
        :type timesteps: int
        """
        super().record_transition(
            states, actions, rewards, next_states, terminated, truncated, infos,
timestep, timesteps
        )

        if self.memories:
            shared_states = infos["shared_states"]
            self._current_shared_next_states = infos["shared_next_states"]

            for uid in self.possible_agents:
                # reward shaping
                if self._rewards_shaper is not None:
                    rewards[uid] = self._rewards_shaper(rewards[uid], timestep,
timesteps)

                # compute values
                with torch.autocast(device_type=self._device_type,
enabled=self._mixed_precision):
                    values, _, _ = self.values[uid].act(
                        {"states": self._shared_state_preprocessor[uid]
(shared_states)}, role="value"
                    )
                    values = self._value_preprocessor[uid](values, inverse=True)

                # time-limit (truncation) bootstrapping
                if self._time_limit_bootstrap[uid]:
                    rewards[uid] += self._discount_factor[uid] * values *
truncated[uid]

                # storage transition in memory
                self.memories[uid].add_samples(
                    states=states[uid],
                    actions=actions[uid],
                    rewards=rewards[uid],
                    next_states=next_states[uid],
                    terminated=terminated[uid],
                    truncated=truncated[uid],
                    log_prob=self._current_log_prob[uid],
                    values=values,
                    shared_states=shared_states,
                )

    def pre_interaction(self, timestep: int, timesteps: int) -> None:
        """Callback called before the interaction with the environment

        :param timestep: Current timestep
        :type timestep: int
        :param timesteps: Number of timesteps
        :type timesteps: int
        """
```

```
        pass

    def post_interaction(self, timestep: int, timesteps: int) -> None:
        """Callback called after the interaction with the environment

        :param timestep: Current timestep
        :type timestep: int
        :param timesteps: Number of timesteps
        :type timesteps: int
        """
        self._rollout += 1
        if not self._rollout % self._rollouts and timestep >=
self._learning_starts:
            self.set_mode("train")
            self._update(timestep, timesteps)
            self.set_mode("eval")

        # write tracking data and checkpoints
        super().post_interaction(timestep, timesteps)

    def _update(self, timestep: int, timesteps: int) -> None:
        """Algorithm's main update step

        :param timestep: Current timestep
        :type timestep: int
        :param timesteps: Number of timesteps
        :type timesteps: int
        """

        def compute_gae(
            rewards: torch.Tensor,
            dones: torch.Tensor,
            values: torch.Tensor,
            next_values: torch.Tensor,
            discount_factor: float = 0.99,
            lambda_coefficient: float = 0.95,
        ) -> torch.Tensor:
            """Compute the Generalized Advantage Estimator (GAE)

            :param rewards: Rewards obtained by the agent
            :type rewards: torch.Tensor
            :param dones: Signals to indicate that episodes have ended
            :type dones: torch.Tensor
            :param values: Values obtained by the agent
            :type values: torch.Tensor
            :param next_values: Next values obtained by the agent
            :type next_values: torch.Tensor
            :param discount_factor: Discount factor
            :type discount_factor: float
            :param lambda_coefficient: Lambda coefficient
            :type lambda_coefficient: float

            :return: Generalized Advantage Estimator
            :rtype: torch.Tensor
            """
            advantage = 0
            advantages = torch.zeros_like(rewards)
            not_dones = dones.logical_not()
            memory_size = rewards.shape[0]

            # advantages computation
            for i in reversed(range(memory_size)):
                next_values = values[i + 1] if i < memory_size - 1 else
last_values
```

```
                advantage = (
                    rewards[i]
                    - values[i]
                    + discount_factor * not_dones[i] * (next_values +
lambda_coefficient * advantage)
                )
                advantages[i] = advantage
            # returns computation
            returns = advantages + values
            # normalize advantages
            advantages = (advantages - advantages.mean()) / (advantages.std() +
1e-8)

            return returns, advantages

        for uid in self.possible_agents:
            policy = self.policies[uid]
            value = self.values[uid]
            memory = self.memories[uid]

            # compute returns and advantages
            with torch.no_grad(), torch.autocast(device_type=self._device_type,
enabled=self._mixed_precision):
                value.train(False)
                last_values, _, _ = value.act(
                    {"states": self._shared_state_preprocessor[uid]
(self._current_shared_next_states.float())},
                    role="value",
                )
                value.train(True)
            last_values = self._value_preprocessor[uid](last_values,
inverse=True)

            values = memory.get_tensor_by_name("values")
            returns, advantages = compute_gae(
                rewards=memory.get_tensor_by_name("rewards"),
                dones=memory.get_tensor_by_name("terminated") |
memory.get_tensor_by_name("truncated"),
                values=values,
                next_values=last_values,
                discount_factor=self._discount_factor[uid],
                lambda_coefficient=self._lambda[uid],
            )

            memory.set_tensor_by_name("values", self._value_preprocessor[uid]
(values, train=True))
            memory.set_tensor_by_name("returns", self._value_preprocessor[uid]
(returns, train=True))
            memory.set_tensor_by_name("advantages", advantages)

            # sample mini-batches from memory
            sampled_batches = memory.sample_all(names=self._tensors_names,
mini_batches=self._mini_batches[uid])

            cumulative_policy_loss = 0
            cumulative_entropy_loss = 0
            cumulative_value_loss = 0

            # learning epochs
            for epoch in range(self._learning_epochs[uid]):
                kl_divergences = []

                # mini-batches loop
                for (
```

```
                    sampled_states,
                    sampled_shared_states,
                    sampled_actions,
                    sampled_log_prob,
                    sampled_values,
                    sampled_returns,
                    sampled_advantages,
                ) in sampled_batches:

                    with torch.autocast(device_type=self._device_type,
enabled=self._mixed_precision):

                        sampled_states = self._state_preprocessor[uid]
(sampled_states, train=not epoch)
                        sampled_shared_states =
self._shared_state_preprocessor[uid](
                            sampled_shared_states, train=not epoch
                        )

                        _, next_log_prob, _ = policy.act(
                            {"states": sampled_states, "taken_actions":
sampled_actions}, role="policy"
                        )

                        # compute approximate KL divergence
                        with torch.no_grad():
                            ratio = next_log_prob - sampled_log_prob
                            kl_divergence = ((torch.exp(ratio) - 1) -
ratio).mean()
                            kl_divergences.append(kl_divergence)

                        # early stopping with KL divergence
                        if self._kl_threshold[uid] and kl_divergence >
self._kl_threshold[uid]:
                            break

                        # compute entropy loss
                        if self._entropy_loss_scale[uid]:
                            entropy_loss = -self._entropy_loss_scale[uid] *
policy.get_entropy(role="policy").mean()
                        else:
                            entropy_loss = 0

                        # compute policy loss
                        ratio = torch.exp(next_log_prob - sampled_log_prob)
                        surrogate = sampled_advantages * ratio
                        surrogate_clipped = sampled_advantages * torch.clip(
                            ratio, 1.0 - self._ratio_clip[uid], 1.0 +
self._ratio_clip[uid]
                        )

                        policy_loss = -torch.min(surrogate,
surrogate_clipped).mean()

                        # compute value loss
                        predicted_values, _, _ = value.act({"states":
sampled_shared_states}, role="value")

                        if self._clip_predicted_values:
                            predicted_values = sampled_values + torch.clip(
                                predicted_values - sampled_values, min=-
self._value_clip[uid], max=self._value_clip[uid]
                            )
                        value_loss = self._value_loss_scale[uid] *
```

```
F.mse_loss(sampled_returns, predicted_values)

                    # optimization step
                    self.optimizers[uid].zero_grad()
                    self.scaler.scale(policy_loss + entropy_loss +
value_loss).backward()

                    if config.torch.is_distributed:
                        policy.reduce_parameters()
                        if policy is not value:
                            value.reduce_parameters()

                    if self._grad_norm_clip[uid] > 0:
                        self.scaler.unscale_(self.optimizers[uid])
                        if policy is value:
                            nn.utils.clip_grad_norm_(policy.parameters(),
self._grad_norm_clip[uid])
                        else:
                            nn.utils.clip_grad_norm_(
                                itertools.chain(policy.parameters(),
value.parameters()), self._grad_norm_clip[uid]
                            )

                    self.scaler.step(self.optimizers[uid])
                    self.scaler.update()

                    # update cumulative losses
                    cumulative_policy_loss += policy_loss.item()
                    cumulative_value_loss += value_loss.item()
                    if self._entropy_loss_scale[uid]:
                        cumulative_entropy_loss += entropy_loss.item()

                # update learning rate
                if self._learning_rate_scheduler[uid]:
                    if isinstance(self.schedulers[uid], KLAdaptiveLR):
                        kl = torch.tensor(kl_divergences,
device=self.device).mean()
                        # reduce (collect from all workers/processes) KL in
distributed runs
                        if config.torch.is_distributed:
                            torch.distributed.all_reduce(kl,
op=torch.distributed.ReduceOp.SUM)
                            kl /= config.torch.world_size
                        self.schedulers[uid].step(kl.item())
                    else:
                        self.schedulers[uid].step()

            # record data
            self.track_data(
                f"Loss / Policy loss ({uid})",
                cumulative_policy_loss / (self._learning_epochs[uid] *
self._mini_batches[uid]),
            )
            self.track_data(
                f"Loss / Value loss ({uid})",
                cumulative_value_loss / (self._learning_epochs[uid] *
self._mini_batches[uid]),
            )
            if self._entropy_loss_scale:
                self.track_data(
                    f"Loss / Entropy loss ({uid})",
                    cumulative_entropy_loss / (self._learning_epochs[uid] *
self._mini_batches[uid]),
                )
```

```
            self.track_data(
                f"Policy / Standard deviation ({uid})",
policy.distribution(role="policy").stddev.mean().item()
            )

            if self._learning_rate_scheduler[uid]:
                self.track_data(f"Learning / Learning rate ({uid})",
self.schedulers[uid].get_last_lr()[0])
```