```python
# Copyright 2022 The Deep RL Zoo Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================
"""Agent57 agent class.

From the paper "Agent57: Outperforming the Atari Human Benchmark"
https://arxiv.org/pdf/2003.13350.
"""

from typing import Iterable, Mapping, Optional, Tuple, NamedTuple, Text
import copy
import multiprocessing
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F

# pylint: disable=import-error
import deep_rl_zoo.replay as replay_lib
import deep_rl_zoo.types as types_lib
from deep_rl_zoo import normalizer
```

```python
from deep_rl_zoo import transforms
from deep_rl_zoo import nonlinear_bellman
from deep_rl_zoo import base
from deep_rl_zoo import distributed
from deep_rl_zoo import bandit
from deep_rl_zoo.curiosity import EpisodicBonusModule, RndLifeLongBonusModule
from deep_rl_zoo.networks.dqn import NguDqnNetworkInputs

# torch.autograd.set_detect_anomaly(True)

HiddenState = Tuple[torch.Tensor, torch.Tensor]


class Agent57Transition(NamedTuple):
    """
    s_t, r_t, done are the tuple from env.step().

    last_action is the last agent the agent took, before in s_t.
    """

    s_t: Optional[np.ndarray]
    a_t: Optional[int]
    q_t: Optional[np.ndarray]  # q values for s_t, computed from both ext_q_network
    prob_a_t: Optional[np.ndarray]  # probability of choose a_t in s_t
    last_action: Optional[int]  # for network input only
    ext_r_t: Optional[float]  # extrinsic reward for (s_tm1, a_tm1)
    int_r_t: Optional[float]  # intrinsic reward for (s_tm1)
    policy_index: Optional[int]  # intrinsic reward scale beta index
    beta: Optional[float]  # intrinsic reward scale beta value
    discount: Optional[float]
    done: Optional[bool]
    ext_init_h: Optional[np.ndarray]  # nn.LSTM initial hidden state, from ext_q_net
    ext_init_c: Optional[np.ndarray]  # nn.LSTM initial cell state, from ext_q_netwo
    int_init_h: Optional[np.ndarray]  # nn.LSTM initial hidden state, from int_q_net
    int_init_c: Optional[np.ndarray]  # nn.LSTM initial cell state, from int_q_netwo


TransitionStructure = Agent57Transition(
    s_t=None,
    a_t=None,
    q_t=None,
    prob_a_t=None,
    last_action=None,
    ext_r_t=None,
    int_r_t=None,
    policy_index=None,
    beta=None,
    discount=None,
    done=None,
    ext_init_h=None,
    ext_init_c=None,
```

```
    int_init_h=None,
    int_init_c=None,
)


def compute_transformed_q(ext_q: torch.Tensor, int_q: torch.Tensor, beta: torch.Tens
    """Returns transformed state-action values from ext_q and int_q."""
    if not isinstance(beta, torch.Tensor):
        beta = torch.tensor(beta).expand_as(int_q).to(device=ext_q.device)

    if len(beta.shape) < len(int_q.shape):
        beta = beta[..., None].expand_as(int_q)

    return transforms.signed_hyperbolic(transforms.signed_parabolic(ext_q) + beta *


def no_autograd(net: torch.nn.Module):
    """Disable autograd for a network."""
    net.eval()
    for p in net.parameters():
        p.requires_grad = False


class Actor(types_lib.Agent):
    """Agent57 actor"""

    def __init__(
        self,
        rank: int,
        data_queue: multiprocessing.Queue,
        ext_q_network: torch.nn.Module,
        int_q_network: torch.nn.Module,
        learner_ext_q_network: torch.nn.Module,
        learner_int_q_network: torch.nn.Module,
        rnd_target_network: torch.nn.Module,
        rnd_predictor_network: torch.nn.Module,
        learner_rnd_predictor_network: torch.nn.Module,
        embedding_network: torch.nn.Module,
        learner_embedding_network: torch.nn.Module,
        random_state: np.random.RandomState,  # pylint: disable=no-member
        ext_discount: float,
        int_discount: float,
        num_actors: int,
        action_dim: int,
        unroll_length: int,
        burn_in: int,
        num_policies: int,
        policy_beta: float,
        ucb_window_size: int,
        ucb_beta: float,
        ucb_epsilon: float,
```

Random distillation network (from NGU)

```
135         episodic_memory_capacity: int,
136         num_neighbors: int,
137         cluster_distance: float,
138         kernel_epsilon: float,
139         max_similarity: float,
140         actor_update_frequency: int,
141         device: torch.device,
142     ) -> None:
143         """
144         Args:
145             rank: the rank number for the actor.
146             data_queue: a multiprocessing.Queue to send collected transitions to lea
147             network: the Q network for actor to make action choice.
148             learner_network: the Q networks with updated weights.
149             rnd_target_network: RND random target network.
150             rnd_predictor_network: RND predictor target network.
151             learner_rnd_predictor_network: RND predictor target network with updated
152             embedding_network: NGU action prediction network.
153             learner_embedding_network: NGU action prediction network with updated we
154             random_state: random state.
155             ext_discount: extrinsic reward discount.
156             int_discount: intrinsic reward discount.
157             num_actors: number of actors.
158             action_dim: number of valid actions in the environment.
159             unroll_length: how many agent time step to unroll transitions before put
160             burn_in: two consecutive unrolls will overlap on burn_in+1 steps.
161             num_policies: number of exploring and exploiting policies.
162             policy_beta: intrinsic reward scale beta.
163             ucb_window_size: window size of the sliding window UCB algorithm.
164             ucb_beta: beta for the sliding window UCB algorithm.
165             ucb_epsilon: exploration epsilon for sliding window UCB algorithm.
166             episodic_memory_capacity: maximum capacity of episodic memory.
167             num_neighbors: number of K-NN neighbors for compute episodic bonus.
168             cluster_distance: K-NN neighbors cluster distance for compute episodic b
169             kernel_epsilon: K-NN kernel epsilon for compute episodic bonus.
170             max_similarity: maximum similarity for compute episodic bonus.
171             actor_update_frequency: the frequency to update actor's Q network.
172             device: PyTorch runtime device.
173         """
174         if not 0.0 <= ext_discount <= 1.0:
175             raise ValueError(f'Expect ext_discount to be [0.0, 1.0], got {ext_discou
176         if not 0.0 <= int_discount <= 1.0:
177             raise ValueError(f'Expect int_discount to be [0.0, 1.0], got {int_discou
178         if not 0 < num_actors:
179             raise ValueError(f'Expect num_actors to be positive integer, got {num_ac
180         if not 0 < action_dim:
181             raise ValueError(f'Expect action_dim to be positive integer, got {action
182         if not 1 <= unroll_length:
183             raise ValueError(f'Expect unroll_length to be integer greater than or eq
184         if not 0 <= burn_in < unroll_length:
185             raise ValueError(f'Expect burn_in length to be [0, {unroll_length}), got
```

```
186         if not 1 <= num_policies:
187             raise ValueError(f'Expect num_policies to be integer greater than or equ
188         if not 0.0 <= policy_beta <= 1.0:
189             raise ValueError(f'Expect policy_beta to be [0.0, 1.0], got {policy_beta
190         if not 1 <= ucb_window_size:
191             raise ValueError(f'Expect ucb_window_size to be integer greater than or
192         if not 0.0 <= ucb_beta <= 100.0:
193             raise ValueError(f'Expect ucb_beta to be [0.0, 100.0], got {ucb_beta}')
194         if not 0.0 <= ucb_epsilon <= 1.0:
195             raise ValueError(f'Expect ucb_epsilon to be [0.0, 1.0], got {ucb_epsilon
196         if not 1 <= episodic_memory_capacity:
197             raise ValueError(
198                 f'Expect episodic_memory_capacity to be integer greater than or equa
199             )
200         if not 1 <= num_neighbors:
201             raise ValueError(f'Expect num_neighbors to be integer greater than or eq
202         if not 0.0 <= cluster_distance <= 1.0:
203             raise ValueError(f'Expect cluster_distance to be [0.0, 1.0], got {cluste
204         if not 0.0 <= kernel_epsilon <= 1.0:
205             raise ValueError(f'Expect kernel_epsilon to be [0.0, 1.0], got {kernel_e
206         if not 1 <= actor_update_frequency:
207             raise ValueError(
208                 f'Expect actor_update_frequency to be integer greater than or equal
209             )
210 
211         self.rank = rank  # Needs to make sure rank always start from 0
212         self.agent_name = f'Agent57-actor{rank}'
213 
214         self._ext_q_network = ext_q_network.to(device=device)
215         self._int_q_network = int_q_network.to(device=device)
216         self._rnd_target_network = rnd_target_network.to(device=device)
217         self._rnd_predictor_network = rnd_predictor_network.to(device=device)
218         self._embedding_network = embedding_network.to(device=device)
219 
220         self._learner_ext_q_network = learner_ext_q_network.to(device=device)
221         self._learner_int_q_network = learner_int_q_network.to(device=device)
222         self._learner_rnd_predictor_network = learner_rnd_predictor_network.to(devic
223         self._learner_embedding_network = learner_embedding_network.to(device=device
224 
225         # Disable autograd for actor's Q networks, embedding, and RND networks.
226         no_autograd(self._ext_q_network)
227         no_autograd(self._int_q_network)
228         no_autograd(self._rnd_target_network)
229         no_autograd(self._rnd_predictor_network)
230         no_autograd(self._embedding_network)
231 
232         self._update_actor_q_network()
233 
234         self._queue = data_queue
235         self._device = device
236         self._random_state = random_state
```

```
        self._num_actors = num_actors
        self._action_dim = action_dim
        self._actor_update_frequency = actor_update_frequency
        self._num_policies = num_policies

        self._unroll = replay_lib.Unroll(
            unroll_length=unroll_length,
            overlap=burn_in + 1,  # Plus 1 to add room for shift during learning
            structure=TransitionStructure,
            cross_episode=False,
        )

        # Meta-collector
        self._meta_coll = bandit.SimplifiedSlidingWindowUCB(
            self._num_policies, ucb_window_size, self._random_state, ucb_beta, ucb_e
        )

        self._betas, self._gammas = distributed.get_ngu_policy_betas_and_discounts(
            num_policies=num_policies,
            beta=policy_beta,
            gamma_max=ext_discount,
            gamma_min=int_discount,
        )
        self._policy_index = None
        self._policy_beta = None
        self._policy_discount = None
        self._sample_policy()

        # E-greedy policy epsilon, rank 0 has the lowest noise, while rank N-1 has t
        epsilons = distributed.get_actor_exploration_epsilon(num_actors)
        self._exploration_epsilon = epsilons[self.rank]

        # Episodic intrinsic bonus module
        self._episodic_module = EpisodicBonusModule(
            embedding_network=embedding_network,
            device=device,
            capacity=episodic_memory_capacity,
            num_neighbors=num_neighbors,
            kernel_epsilon=kernel_epsilon,
            cluster_distance=cluster_distance,
            max_similarity=max_similarity,
        )

        # Lifelong intrinsic bonus module
        self._lifelong_module = RndLifeLongBonusModule(
            target_network=rnd_target_network,
            predictor_network=rnd_predictor_network,
            device=device,
        )

        self._episode_returns = 0.0
```

```
288         self._last_action = None
289         self._episodic_bonus_t = None
290         self._lifelong_bonus_t = None
291         self._ext_lstm_state = None  # Stores nn.LSTM hidden state and cell state. f
292         self._int_lstm_state = None  # Stores nn.LSTM hidden state and cell state. f
293 
294         self._step_t = -1
295 
296     @torch.no_grad()
297     def step(self, timestep: types_lib.TimeStep) -> types_lib.Action:
298         """Given timestep, return action a_t, and push transition into global queue"
299         self._step_t += 1
300         self._episode_returns += timestep.reward
301 
302         if self._step_t % self._actor_update_frequency == 0:
303             self._update_actor_q_network()
304 
305         q_t, a_t, prob_a_t, ext_hidden_s, int_hidden_s = self.act(timestep)
306 
307         transition = Agent57Transition(
308             s_t=timestep.observation,
309             a_t=a_t,
310             q_t=q_t,
311             prob_a_t=prob_a_t,
312             last_action=self._last_action,
313             ext_r_t=timestep.reward,
314             int_r_t=self.intrinsic_reward,
315             policy_index=self._policy_index,
316             beta=self._policy_beta,
317             discount=self._policy_discount,
318             done=timestep.done,
319             ext_init_h=self._ext_lstm_state[0].squeeze(1).cpu().numpy(),  # remove b
320             ext_init_c=self._ext_lstm_state[1].squeeze(1).cpu().numpy(),
321             int_init_h=self._int_lstm_state[0].squeeze(1).cpu().numpy(),  # remove b
322             int_init_c=self._int_lstm_state[1].squeeze(1).cpu().numpy(),
323         )
324 
325         unrolled_transition = self._unroll.add(transition, timestep.done)
326 
327         s_t = torch.from_numpy(timestep.observation[None, ...]).to(device=self._devi
328 
329         # Compute lifelong intrinsic bonus
330         self._lifelong_bonus_t = self._lifelong_module.compute_bonus(s_t)
331 
332         # Compute episodic intrinsic bonus
333         self._episodic_bonus_t = self._episodic_module.compute_bonus(s_t)
334 
335         # Update local state
336         self._last_action, self._ext_lstm_state, self._int_lstm_state = a_t, ext_hid
337 
338         if unrolled_transition is not None:
```

```
339             self._put_unroll_onto_queue(unrolled_transition)
340
341         # Update Sliding Window UCB statistics.
342         if timestep.done:
343             self._meta_coll.update(self._policy_index, self._episode_returns)
344
345         return a_t
346
347     def reset(self) -> None:
348         """This method should be called at the beginning of every episode before tak
349         self._unroll.reset()
350         self._episodic_module.reset()
351         self._episode_returns = 0.0
352
353         # Update embedding and RND predictor network parameters at beginning of ever
354         self._update_embedding_and_rnd_networks()
355                                                                  from bandit problems
356         # Agent57 actor samples a policy using the Sliding Window UCB algorithm, the
357         self._sample_policy()
358
359         # During the first step of a new episode,
360         # use 'fake' previous action and 'intrinsic' reward for network pass
361         self._last_action = self._random_state.randint(0, self._action_dim)  # Initi
362         self._episodic_bonus_t = 0.0
363         self._lifelong_bonus_t = 0.0
364         self._ext_lstm_state = self._ext_q_network.get_initial_hidden_state(batch_si
365         self._int_lstm_state = self._int_q_network.get_initial_hidden_state(batch_si
366
367     def act(self, timestep: types_lib.TimeStep) -> Tuple[np.ndarray, types_lib.Actio
368         'Given state s_t and done marks, return an action.'
369         return self._choose_action(timestep, self._exploration_epsilon)
370
371     @torch.no_grad()
372     def _choose_action(
373         self, timestep: types_lib.TimeStep, epsilon: float
374     ) -> Tuple[np.ndarray, types_lib.Action, float, HiddenState, HiddenState]:
375         """Given state s_t, choose action a_t"""
376         q_ext_input_ = self._prepare_network_input(timestep, self._ext_lstm_state)
377         q_int_input_ = self._prepare_network_input(timestep, self._int_lstm_state)
378
379         pi_ext_output = self._ext_q_network(q_ext_input_)
380         pi_int_output = self._int_q_network(q_int_input_)
381         ext_q_t = pi_ext_output.q_values.squeeze()
382         int_q_t = pi_int_output.q_values.squeeze()
383
384         q_t = compute_transformed_q(ext_q_t, int_q_t, self._policy_beta)
385
386         a_t = torch.argmax(q_t, dim=-1).cpu().item()      greedy action selection
387
388         # Policy probability for a_t, the detailed equation is mentioned in Agent57
389         prob_a_t = 1 - (self._exploration_epsilon * ((self._action_dim - 1) / self._
```

```
390
391         # To make sure every actors generates the same amount of samples, we apply e
392         # otherwise the actor with higher epsilons will generate more samples,
393         # while the actor with lower epsilon will generate less samples.
394         if self._random_state.rand() <= epsilon:        overwrite if exploring
395             # randint() return random integers from low (inclusive) to high (exclusi
396             a_t = self._random_state.randint(0, self._action_dim)
397             prob_a_t = self._exploration_epsilon / self._action_dim
398
399         return (
400             q_t.cpu().numpy(),
401             a_t,
402             prob_a_t,
403             pi_ext_output.hidden_s,
404             pi_int_output.hidden_s,
405         )
406
407     def _prepare_network_input(self, timestep: types_lib.TimeStep, hidden_state: Tup
408         # NGU network expect input shape [T, B, state_shape],
409         # and additionally 'last action', 'extrinsic reward for last action', last i
410         s_t = torch.tensor(timestep.observation[None, ...]).to(device=self._device,
411         last_action = torch.tensor(self._last_action).to(device=self._device, dtype=
412         ext_r_t = torch.tensor(timestep.reward).to(device=self._device, dtype=torch.
413         int_r_t = torch.tensor(self.intrinsic_reward).to(device=self._device, dtype=
414         policy_index = torch.tensor(self._policy_index).to(device=self._device, dtyp
415         hidden_s = tuple(s.to(device=self._device) for s in hidden_state)
416         return NguDqnNetworkInputs(
417             s_t=s_t[None, ...],  # [T, B, state_shape]
418             a_tm1=last_action[None, ...],  # [T, B]
419             ext_r_t=ext_r_t[None, ...],  # [T, B]
420             int_r_t=int_r_t[None, ...],  # [T, B]
421             policy_index=policy_index[None, ...],  # [T, B]
422             hidden_s=hidden_s,
423         )
424
425     def _put_unroll_onto_queue(self, unrolled_transition):
426         # Important note, store hidden states for every step in the unroll will cons
427         self._queue.put(unrolled_transition)
428
429     def _sample_policy(self):
430         """Sample new policy from meta collector."""
431         self._policy_index = self._meta_coll.sample()
432         self._policy_beta = self._betas[self._policy_index]
433         self._policy_discount = self._gammas[self._policy_index]
434
435     def _update_actor_q_network(self):
436         self._ext_q_network.load_state_dict(self._learner_ext_q_network.state_dict()
437         self._int_q_network.load_state_dict(self._learner_int_q_network.state_dict()
438
439     def _update_embedding_and_rnd_networks(self):
440         self._lifelong_module.update_predictor_network(self._learner_rnd_predictor_n
```

```
441         self._episodic_module.update_embedding_network(self._learner_embedding_netwo
442
443     @property
444     def intrinsic_reward(self) -> float:
445         """Returns intrinsic reward for last state s_tm1."""
446         # Equation 1 of the NGU paper.
447         return self._episodic_bonus_t * min(max(self._lifelong_bonus_t, 1.0), 5.0)
448
449     @property
450     def statistics(self) -> Mapping[Text, float]:
451         """Returns current actor's statistics as a dictionary."""
452         return {
453             # 'policy_index': self._policy_index,
454             'policy_discount': self._policy_discount,
455             'policy_beta': self._policy_beta,
456             'exploration_epsilon': self._exploration_epsilon,
457             'intrinsic_reward': self.intrinsic_reward,
458             # 'episodic_bonus': self._episodic_bonus_t,
459             # 'lifelong_bonus': self._lifelong_bonus_t,
460         }
461
462
463 class Learner(types_lib.Learner):
464     """Agent57 learner"""
465
466     def __init__(
467         self,
468         ext_q_network: nn.Module,
469         ext_q_optimizer: torch.optim.Optimizer,
470         int_q_network: nn.Module,
471         int_q_optimizer: torch.optim.Optimizer,
472         embedding_network: nn.Module,
473         rnd_target_network: nn.Module,
474         rnd_predictor_network: nn.Module,
475         intrinsic_optimizer: torch.optim.Optimizer,
476         replay: replay_lib.PrioritizedReplay,
477         target_network_update_frequency: int,
478         min_replay_size: int,
479         batch_size: int,
480         unroll_length: int,
481         burn_in: int,
482         retrace_lambda: float,
483         transformed_retrace: bool,
484         priority_eta: float,
485         clip_grad: bool,
486         max_grad_norm: float,
487         device: torch.device,
488     ) -> None:
489         """
490         Args:
491             network: the Q network we want to train and optimize.
```

```
            optimizer: the optimizer for Q network.
            embedding_network: NGU action prediction network.
            rnd_target_network: RND random network.
            rnd_predictor_network: RND predictor network.
            intrinsic_optimizer: the optimizer for action prediction and RND predict
            replay: prioritized recurrent experience replay.
            target_network_update_frequency: how often to copy online network parame
            min_replay_size: wait till experience replay buffer this number before s
            batch_size: sample batch_size of transitions.
            burn_in: burn n transitions to generate initial hidden state before lear
            unroll_length: transition sequence length.
            retrace_lambda: coefficient of the retrace lambda.
            transformed_retrace: if True, use transformed retrace.
            priority_eta: coefficient to mix the max and mean absolute TD errors.
            clip_grad: if True, clip gradients norm.
            max_grad_norm: the maximum gradient norm for clip grad, only works if cl
            device: PyTorch runtime device.
        """
        if not 1 <= target_network_update_frequency:
            raise ValueError(
                f'Expect target_network_update_frequency to be positive integer, got
            )
        if not 1 <= min_replay_size:
            raise ValueError(f'Expect min_replay_size to be integer greater than or
        if not 1 <= batch_size <= 512:
            raise ValueError(f'Expect batch_size to in the range [1, 512], got {batc
        if not 1 <= unroll_length:
            raise ValueError(f'Expect unroll_length to be greater than or equal to 1
        if not 0 <= burn_in < unroll_length:
            raise ValueError(f'Expect burn_in length to be [0, {unroll_length}), got
        if not 0.0 <= retrace_lambda <= 1.0:
            raise ValueError(f'Expect retrace_lambda to in the range [0.0, 1.0], got
        if not 0.0 <= priority_eta <= 1.0:
            raise ValueError(f'Expect priority_eta to in the range [0.0, 1.0], got {

        self.agent_name = 'Agent57-learner'
        self._device = device
        self._online_ext_q_network = ext_q_network.to(device=device)
        self._online_ext_q_network.train()
        self._ext_q_optimizer = ext_q_optimizer
        self._online_int_q_network = int_q_network.to(device=device)
        self._online_int_q_network.train()
        self._int_q_optimizer = int_q_optimizer
        self._embedding_network = embedding_network.to(device=self._device)
        self._embedding_network.train()
        self._rnd_predictor_network = rnd_predictor_network.to(device=self._device)
        self._rnd_predictor_network.train()
        self._intrinsic_optimizer = intrinsic_optimizer

        self._rnd_target_network = rnd_target_network.to(device=self._device)
        # Lazy way to create target Q networks
```

```
543         self._target_ext_q_network = copy.deepcopy(self._online_ext_q_network).to(de
544         self._target_int_q_network = copy.deepcopy(self._online_int_q_network).to(de
545 
546         # Disable autograd for target Q networks, and RND target networks.
547         no_autograd(self._target_ext_q_network)
548         no_autograd(self._target_int_q_network)
549         no_autograd(self._rnd_target_network)
550 
551         self._batch_size = batch_size
552         self._burn_in = burn_in
553         self._unroll_length = unroll_length
554         self._total_unroll_length = unroll_length + 1
555         self._target_network_update_frequency = target_network_update_frequency
556         self._clip_grad = clip_grad
557         self._max_grad_norm = max_grad_norm
558 
559         self._observation_normalizer = normalizer.Normalizer(eps=0.0001, clip_range=
560 
561         self._replay = replay
562         self._min_replay_size = min_replay_size
563         self._priority_eta = priority_eta
564 
565         self._retrace_lambda = retrace_lambda
566         self._transformed_retrace = transformed_retrace
567 
568         self._step_t = -1
569         self._update_t = 0
570         self._target_update_t = 0
571         self._ext_q_loss_t = np.nan
572         self._int_q_loss_t = np.nan
573         self._embedding_rnd_loss_t = np.nan
574 
575     def step(self) -> Iterable[Mapping[Text, float]]:
576         """Increment learner step, and potentially do a update when called.
577 
578         Yields:
579             learner statistics if network parameters update occurred, otherwise retu
580         """
581         self._step_t += 1
582 
583         if self._replay.size < self._batch_size or self._step_t % self._batch_size !
584             return
585 
586         self._learn()
587         yield self.statistics
588 
589     def reset(self) -> None:
590         """Should be called at the beginning of every iteration."""
591 
592     def received_item_from_queue(self, item) -> None:
593         """Received item send by actors through multiprocessing queue."""
```

```
        # Use the unrolled sequence to calculate priority
        priority = self._compute_priority_for_unroll(item)
        self._replay.add(item, priority)

    def _learn(self) -> None:
        transitions, indices, weights = self._replay.sample(self._batch_size)
        priorities = self._update(transitions, weights)
        self._update_action_prediction_and_rnd_predictor_networks(transitions, weigh

        if priorities.shape != (self._batch_size,):
            raise RuntimeError(f'Expect priorities has shape ({self._batch_size},),
        priorities = np.abs(priorities)
        self._replay.update_priorities(indices, priorities)

        # Copy online Q network parameters to target Q network, every m updates
        if self._update_t > 1 and self._update_t % self._target_network_update_frequ
            self._update_target_network()

    def _update(self, transitions: Agent57Transition, weights: np.ndarray) -> np.nda
        weights = torch.from_numpy(weights).to(device=self._device, dtype=torch.floa
        base.assert_rank_and_dtype(weights, 1, torch.float32)

        # Get initial hidden state for both extrinsic and intrinsic Q networks, hand
        init_ext_q_hidden_state, init_int_q_hidden_state = self._extract_first_step_
        burn_transitions, learn_transitions = replay_lib.split_structure(transitions
        if burn_transitions is not None:
            # Burn in for extrinsic Q networks.
            hidden_ext_online_q, hidden_ext_target_q = self._burn_in_unroll_q_networ
                burn_transitions,
                self._online_ext_q_network,
                self._target_ext_q_network,
                init_ext_q_hidden_state,
            )
            # Burn in for intrinsic Q networks.
            hidden_int_online_q, hidden_int_target_q = self._burn_in_unroll_q_networ
                burn_transitions,
                self._online_int_q_network,
                self._target_int_q_network,
                init_int_q_hidden_state,
            )
        else:
            # Make copy of hidden state for extrinsic Q networks.
            hidden_ext_online_q = tuple(s.clone().to(device=self._device) for s in i
            hidden_ext_target_q = tuple(s.clone().to(device=self._device) for s in i
            # Make copy of hidden state for intrinsic Q networks.
            hidden_int_online_q = tuple(s.clone().to(device=self._device) for s in i
            hidden_int_target_q = tuple(s.clone().to(device=self._device) for s in i

        # Do network pass for all four Q networks to get estimated q values.
        ext_q_t = self._get_predicted_q_values(learn_transitions, self._online_ext_q
        int_q_t = self._get_predicted_q_values(learn_transitions, self._online_int_q
```

```
645         with torch.no_grad():
646             target_ext_q_t = self._get_predicted_q_values(learn_transitions, self._t
647             target_int_q_t = self._get_predicted_q_values(learn_transitions, self._t
648 
649         # Update extrinsic online Q network.
650         self._ext_q_optimizer.zero_grad()
651         ext_q_loss, ext_priorities = self._calc_retrace_loss(learn_transitions, ext_
652         # Multiply loss by sampling weights, averaging over batch dimension
653         ext_q_loss = torch.mean(ext_q_loss * weights.detach())
654         ext_q_loss.backward()
655         if self._clip_grad:
656             torch.nn.utils.clip_grad_norm_(self._online_ext_q_network.parameters(),
657         self._ext_q_optimizer.step()
658 
659         # Update intrinsic online Q network.
660         self._int_q_optimizer.zero_grad()
661         int_q_loss, int_priorities = self._calc_retrace_loss(learn_transitions, int_
662         # Multiply loss by sampling weights, averaging over batch dimension
663         int_q_loss = torch.mean(int_q_loss * weights.detach())
664         int_q_loss.backward()
665 
666         if self._clip_grad:
667             torch.nn.utils.clip_grad_norm_(self._online_int_q_network.parameters(),
668         self._int_q_optimizer.step()
669 
670         priorities = 0.8 * ext_priorities + 0.2 * int_priorities
671         self._update_t += 1
672 
673         # For logging only.
674         self._ext_q_loss_t = ext_q_loss.detach().cpu().item()
675         self._int_q_loss_t = int_q_loss.detach().cpu().item()
676         return priorities
677 
678     def _get_predicted_q_values(
679         self,
680         transitions: Agent57Transition,
681         q_network: torch.nn.Module,
682         hidden_state: HiddenState,
683     ) -> torch.Tensor:
684         """Returns the predicted q values from the 'q_network' for a given batch of
685 
686         Args:
687             transitions: sampled batch of unrolls, this should not include the burn_
688             q_network: this could be any one of the extrinsic and intrinsic (online
689             hidden_state: initial hidden states for the 'q_network'.
690         """
691         s_t = torch.from_numpy(transitions.s_t).to(device=self._device, dtype=torch.
692         last_action = torch.from_numpy(transitions.last_action).to(device=self._devi
693         ext_r_t = torch.from_numpy(transitions.ext_r_t).to(device=self._device, dtyp
694         int_r_t = torch.from_numpy(transitions.int_r_t).to(device=self._device, dtyp
695         policy_index = torch.from_numpy(transitions.policy_index).to(device=self._de
```

```
696
697         # Rank and dtype checks, note we have a new unroll time dimension, states ma
698         base.assert_rank_and_dtype(s_t, (3, 5), torch.float32)
699         base.assert_rank_and_dtype(last_action, 2, torch.long)
700         base.assert_rank_and_dtype(ext_r_t, 2, torch.float32)
701         base.assert_rank_and_dtype(int_r_t, 2, torch.float32)
702         base.assert_rank_and_dtype(policy_index, 2, torch.long)
703
704         # Rank and dtype checks for hidden state.
705         base.assert_rank_and_dtype(hidden_state[0], 3, torch.float32)
706         base.assert_rank_and_dtype(hidden_state[1], 3, torch.float32)
707         base.assert_batch_dimension(hidden_state[0], self._batch_size, 1)
708         base.assert_batch_dimension(hidden_state[1], self._batch_size, 1)
709
710         # Get q values from Q network,
711         q_t = q_network(
712             NguDqnNetworkInputs(
713                 s_t=s_t,
714                 a_tm1=last_action,
715                 ext_r_t=ext_r_t,
716                 int_r_t=int_r_t,
717                 policy_index=policy_index,
718                 hidden_s=hidden_state,
719             )
720         ).q_values
721
722         return q_t
723
724     def _calc_retrace_loss(
725         self,
726         transitions: Agent57Transition,
727         q_t: torch.Tensor,
728         target_q_t: torch.Tensor,
729     ) -> Tuple[torch.Tensor, np.ndarray]:
730         a_t = torch.from_numpy(transitions.a_t).to(device=self._device, dtype=torch.
731         behavior_prob_a_t = torch.from_numpy(transitions.prob_a_t).to(device=self._d
732         ext_r_t = torch.from_numpy(transitions.ext_r_t).to(device=self._device, dtyp
733         int_r_t = torch.from_numpy(transitions.int_r_t).to(device=self._device, dtyp
734         beta = torch.from_numpy(transitions.beta).to(device=self._device, dtype=torc
735         discount = torch.from_numpy(transitions.discount).to(device=self._device, dt
736         done = torch.from_numpy(transitions.done).to(device=self._device, dtype=torc
737
738         # Rank and dtype checks, note we have a new unroll time dimension, states ma
739         base.assert_rank_and_dtype(behavior_prob_a_t, 2, torch.float32)
740         base.assert_rank_and_dtype(a_t, 2, torch.long)
741         base.assert_rank_and_dtype(ext_r_t, 2, torch.float32)
742         base.assert_rank_and_dtype(int_r_t, 2, torch.float32)
743         base.assert_rank_and_dtype(beta, 2, torch.float32)
744         base.assert_rank_and_dtype(discount, 2, torch.float32)
745         base.assert_rank_and_dtype(done, 2, torch.bool)
746
```

```
747         r_t = ext_r_t + beta * int_r_t  # Augmented rewards
748         discount_t = (~done).float() * discount  # (T+1, B)
749 
750         # Derive target policy probabilities from q values.
751         target_policy_probs = F.softmax(q_t, dim=-1)  # [T+1, B, action_dim]
752 
753         if self._transformed_retrace:
754             transform_tx_pair = nonlinear_bellman.SIGNED_HYPERBOLIC_PAIR
755         else:
756             transform_tx_pair = nonlinear_bellman.IDENTITY_PAIR  # No transform
757 
758         # Compute retrace loss.
759         retrace_out = nonlinear_bellman.transformed_retrace(
760             q_tm1=q_t[:-1],
761             q_t=target_q_t[1:],
762             a_tm1=a_t[:-1],
763             a_t=a_t[1:],
764             r_t=r_t[:-1],
765             discount_t=discount_t[:-1],
766             pi_t=target_policy_probs[1:],
767             mu_t=behavior_prob_a_t[1:],
768             lambda_=self._retrace_lambda,
769             tx_pair=transform_tx_pair,
770         )
771 
772         # Compute priority.
773         priorities = distributed.calculate_dist_priorities_from_td_error(retrace_out
774 
775         # Sums over time dimension.
776         loss = torch.sum(retrace_out.loss, dim=0)
777 
778         return loss, priorities
779 
780     def _update_action_prediction_and_rnd_predictor_networks(
781         self, transitions: Agent57Transition, weights: np.ndarray
782     ) -> None:
783         """Use last 5 frames to update the embedding and RND predictor networks."""
784         b = self._batch_size
785         weights = torch.from_numpy(weights[-b:]).to(device=self._device, dtype=torch
786         base.assert_rank_and_dtype(weights, 1, torch.float32)
787 
788         self._intrinsic_optimizer.zero_grad()
789         # [batch_size]
790         rnd_pred_loss = self._calc_rnd_predictor_loss(transitions)
791         act_pred_loss = self._calc_action_prediction_loss(transitions)
792         loss = rnd_pred_loss + act_pred_loss
793         # Multiply loss by sampling weights, averaging over batch dimension
794         loss = torch.mean(loss * weights.detach())
795 
796         loss.backward()
797         if self._clip_grad:
```

```
798             torch.nn.utils.clip_grad_norm_(self._rnd_predictor_network.parameters(),
799             torch.nn.utils.clip_grad_norm_(self._embedding_network.parameters(), sel
800
801         self._intrinsic_optimizer.step()
802
803         # For logging only.
804         self._embedding_rnd_loss_t = loss.detach().cpu().item()
805
806     def _calc_rnd_predictor_loss(self, transitions: Agent57Transition) -> torch.Tens
807         s_t = torch.from_numpy(transitions.s_t[-5:]).to(device=self._device, dtype=t
808         # Rank and dtype checks.
809         base.assert_rank_and_dtype(s_t, (3, 5), torch.float32)
810         # Merge batch and time dimension.
811         s_t = torch.flatten(s_t, 0, 1)
812
813         # Compute RND predictor loss.
814         # Update normalize statistics and normalize observations before pass to RND
815         if len(s_t.shape) > 3:
816             # Make channel last, we normalize images by channel.
817             s_t = s_t.swapaxes(1, -1)
818             self._observation_normalizer.update(s_t)
819             s_t = self._observation_normalizer(s_t)
820             # Make channel first so PyTorch Conv2D works.
821             s_t = s_t.swapaxes(1, -1)
822         else:
823             self._observation_normalizer.update(s_t)
824             s_t = self._observation_normalizer(s_t)
825
826         pred_s_t = self._rnd_predictor_network(s_t)
827         with torch.no_grad():
828             target_s_t = self._rnd_target_network(s_t)
829
830         # Compute L2 loss, shape [5*B,]
831         loss = torch.sum(torch.square(pred_s_t - target_s_t), dim=-1)
832         # Reshape loss into [5, B].
833         loss = loss.view(5, -1)
834         # Sums over time dimension. shape [B]
835         loss = torch.sum(loss, dim=0)
836         return loss
837
838     def _calc_action_prediction_loss(self, transitions: Agent57Transition) -> torch.
839         s_t = torch.from_numpy(transitions.s_t[-6:]).to(device=self._device, dtype=t
840         a_t = torch.from_numpy(transitions.a_t[-6:]).to(device=self._device, dtype=t
841
842         # Rank and dtype checks.
843         base.assert_rank_and_dtype(s_t, (3, 5), torch.float32)
844         base.assert_rank_and_dtype(a_t, 2, torch.long)
845
846         s_tm1 = s_t[0:-1, ...]  # [5, B, state_shape]
847         s_t = s_t[1:, ...]  # [5, B, state_shape]
848         a_tm1 = a_t[:-1, ...]  # [5, B]
```

```
849
850         # Merge batch and time dimension.
851         s_tm1 = torch.flatten(s_tm1, 0, 1)
852         s_t = torch.flatten(s_t, 0, 1)
853         a_tm1 = torch.flatten(a_tm1, 0, 1)
854
855         # Compute action prediction loss.
856         embedding_s_tm1 = self._embedding_network(s_tm1)  # [5*B, latent_dim]
857         embedding_s_t = self._embedding_network(s_t)  # [5*B, latent_dim]
858         embeddings = torch.cat([embedding_s_tm1, embedding_s_t], dim=-1)
859         pi_logits = self._embedding_network.inverse_prediction(embeddings)  # [5*B,
860
861         # [5*B,]
862         loss = F.cross_entropy(pi_logits, a_tm1, reduction='none')
863         # Reshape loss into [5, B].
864         loss = loss.view(5, -1)
865         # Sums over time dimension. shape [B]
866         loss = torch.sum(loss, dim=0)
867         return loss
868
869     def _burn_in_unroll_q_networks(
870         self,
871         transitions: Agent57Transition,
872         online_q_network: torch.nn.Module,
873         target_q_network: torch.nn.Module,
874         hidden_state: HiddenState,
875     ) -> Tuple[HiddenState, HiddenState]:
876         """Unroll both online and target q networks to generate hidden states for LS
877         s_t = torch.from_numpy(transitions.s_t).to(device=self._device, dtype=torch.
878         last_action = torch.from_numpy(transitions.last_action).to(device=self._devi
879         ext_r_t = torch.from_numpy(transitions.ext_r_t).to(device=self._device, dtyp
880         int_r_t = torch.from_numpy(transitions.int_r_t).to(device=self._device, dtyp
881         policy_index = torch.from_numpy(transitions.policy_index).to(device=self._de
882
883         # Rank and dtype checks, note we have a new unroll time dimension, states ma
884         base.assert_rank_and_dtype(s_t, (3, 5), torch.float32)
885         base.assert_rank_and_dtype(last_action, 2, torch.long)
886         base.assert_rank_and_dtype(ext_r_t, 2, torch.float32)
887         base.assert_rank_and_dtype(int_r_t, 2, torch.float32)
888         base.assert_rank_and_dtype(policy_index, 2, torch.long)
889
890         hidden_online = tuple(s.clone().to(device=self._device) for s in hidden_stat
891         hidden_target = tuple(s.clone().to(device=self._device) for s in hidden_stat
892
893         # Burn in to generate hidden states for LSTM, we unroll both online and targ
894         with torch.no_grad():
895             hidden_online_q = online_q_network(
896                 NguDqnNetworkInputs(
897                     s_t=s_t,
898                     a_tm1=last_action,
899                     ext_r_t=ext_r_t,
```

```
900                     int_r_t=int_r_t,
901                     policy_index=policy_index,
902                     hidden_s=hidden_online,
903                 )
904             ).hidden_s
905             hidden_target_q = target_q_network(
906                 NguDqnNetworkInputs(
907                     s_t=s_t,
908                     a_tm1=last_action,
909                     ext_r_t=ext_r_t,
910                     int_r_t=int_r_t,
911                     policy_index=policy_index,
912                     hidden_s=hidden_target,
913                 )
914             ).hidden_s
915
916         return (hidden_online_q, hidden_target_q)
917
918     def _extract_first_step_hidden_state(self, transitions: Agent57Transition) -> Tu
919         """Returns ext_hidden_state and int_hidden_state."""
920         # We only need the first step hidden states in replay, shape [batch_size, nu
921         ext_init_h = torch.from_numpy(transitions.ext_init_h[0:1]).squeeze(0).to(dev
922         ext_init_c = torch.from_numpy(transitions.ext_init_c[0:1]).squeeze(0).to(dev
923         int_init_h = torch.from_numpy(transitions.int_init_h[0:1]).squeeze(0).to(dev
924         int_init_c = torch.from_numpy(transitions.int_init_c[0:1]).squeeze(0).to(dev
925
926         # Rank and dtype checks.
927         base.assert_rank_and_dtype(ext_init_h, 3, torch.float32)
928         base.assert_rank_and_dtype(ext_init_c, 3, torch.float32)
929         base.assert_rank_and_dtype(int_init_h, 3, torch.float32)
930         base.assert_rank_and_dtype(int_init_c, 3, torch.float32)
931
932         # Swap batch and num_lstm_layers axis.
933         ext_init_h = ext_init_h.swapaxes(0, 1)
934         ext_init_c = ext_init_c.swapaxes(0, 1)
935         int_init_h = int_init_h.swapaxes(0, 1)
936         int_init_c = int_init_c.swapaxes(0, 1)
937
938         # Batch dimension checks.
939         base.assert_batch_dimension(ext_init_h, self._batch_size, 1)
940         base.assert_batch_dimension(ext_init_c, self._batch_size, 1)
941         base.assert_batch_dimension(int_init_h, self._batch_size, 1)
942         base.assert_batch_dimension(int_init_c, self._batch_size, 1)
943
944         return ((ext_init_h, ext_init_c), (int_init_h, int_init_c))
945
946     @torch.no_grad()
947     def _compute_priority_for_unroll(self, transitions: Agent57Transition) -> float:
948         """Returns priority for a single unroll, no network pass and gradients are r
949         # Note we skip the burn in part, and use the same q values for target.
950         _, learn_transitions = replay_lib.split_structure(transitions, TransitionStr
```

```
951
952         q_t = torch.from_numpy(learn_transitions.q_t).to(device=self._device, dtype=
953         a_t = torch.from_numpy(learn_transitions.a_t).to(device=self._device, dtype=
954         ext_r_t = torch.from_numpy(learn_transitions.ext_r_t).to(device=self._device
955         int_r_t = torch.from_numpy(learn_transitions.int_r_t).to(device=self._device
956         beta = torch.from_numpy(learn_transitions.beta).to(device=self._device, dtyp
957         discount = torch.from_numpy(learn_transitions.discount).to(device=self._devi
958         done = torch.from_numpy(learn_transitions.done).to(device=self._device, dtyp
959         behavior_prob_a_t = torch.from_numpy(learn_transitions.prob_a_t).to(
960             device=self._device, dtype=torch.float32
961         )  # [T+1, ]
962
963         # Rank and dtype checks, single unroll should not have batch dimension.
964         base.assert_rank_and_dtype(q_t, 2, torch.float32)
965         base.assert_rank_and_dtype(a_t, 1, torch.long)
966         base.assert_rank_and_dtype(ext_r_t, 1, torch.float32)
967         base.assert_rank_and_dtype(int_r_t, 1, torch.float32)
968         base.assert_rank_and_dtype(beta, 1, torch.float32)
969         base.assert_rank_and_dtype(discount, 1, torch.float32)
970         base.assert_rank_and_dtype(done, 1, torch.bool)
971         base.assert_rank_and_dtype(behavior_prob_a_t, 1, torch.float32)
972
973         r_t = ext_r_t + beta * int_r_t  # Augmented rewards
974         discount_t = (~done).float() * discount
975
976         # Derive policy probabilities from q values
977         target_policy_probs = F.softmax(q_t, dim=-1)  # [T+1, action_dim]
978
979         # Compute retrace loss, add a batch dimension before retrace ops.
980         if self._transformed_retrace:
981             transform_tx_pair = nonlinear_bellman.SIGNED_HYPERBOLIC_PAIR
982         else:
983             transform_tx_pair = nonlinear_bellman.IDENTITY_PAIR  # No transform
984         retrace_out = nonlinear_bellman.transformed_retrace(
985             q_tm1=q_t[:-1].unsqueeze(1),
986             q_t=q_t[1:].unsqueeze(1),
987             a_tm1=a_t[:-1].unsqueeze(1),
988             a_t=a_t[1:].unsqueeze(1),
989             r_t=r_t[:-1].unsqueeze(1),
990             discount_t=discount_t[:-1].unsqueeze(1),
991             pi_t=target_policy_probs[1:].unsqueeze(1),
992             mu_t=behavior_prob_a_t[1:].unsqueeze(1),
993             lambda_=self._retrace_lambda,
994             tx_pair=transform_tx_pair,
995         )
996
997         priority = distributed.calculate_dist_priorities_from_td_error(retrace_out.e
998         return priority.item()
999
1000     def _update_target_network(self):
1001         self._target_ext_q_network.load_state_dict(self._online_ext_q_network.state_
```

```
1002            self._target_int_q_network.load_state_dict(self._online_int_q_network.state_
1003            self._target_update_t += 1
1004
1005        @property
1006        def statistics(self) -> Mapping[Text, float]:
1007            """Returns current agent statistics as a dictionary."""
1008            return {
1009                # 'ext_q_learning_rate': self._ext_q_optimizer.param_groups[0]['lr'],
1010                # 'int_q_learning_rate': self._int_q_optimizer.param_groups[0]['lr'],
1011                # 'embedding_rnd_lr': self._intrinsic_optimizer.param_groups[0]['lr'],
1012                'ext_q_retrace_loss': self._ext_q_loss_t,
1013                'int_q_retrace_loss': self._int_q_loss_t,
1014                'embedding_rnd_loss': self._embedding_rnd_loss_t,
1015                'updates': self._update_t,
1016                'target_updates': self._target_update_t,
1017            }
```