Table of Contents

TORCHRL.ENVS PACKAGE

TorchRL offers an API to handle environments of different backends, such as gym, dm-control, dm-lab, model-based environments as well as custom environments. The goal is to be able to swap environments in an experiment with little or no effort, even if these environments are simulated using different libraries. TorchRL offers some out-of-the-box environment wrappers under `torchrl.envs.libs`, which we hope can be easily imitated for other libraries. The parent class `EnvBase` is a `torch.nn.Module` subclass that implements some typical environment methods using `tensordict.TensorDict` as a data organiser. This allows this class to be generic and to handle an arbitrary number of input and outputs, as well as nested or batched data structures.

Each env will have the following attributes:

- `env.batch_size`: a `torch.Size` representing the number of envs batched together.
- `env.device`: the device where the input and output tensordict are expected to live. The environment device does not mean that the actual step operations will be computed on device (this is the responsibility of the backend, with which TorchRL can do little). The device of an environment just represents the device where the data is to be expected when input to the environment or retrieved from it. TorchRL takes care of mapping the data to the desired device. This is especially useful for transforms (see below). For parametric environments (e.g. model-based environments), the device does represent the hardware that will be used to compute the operations.
- `env.observation_spec`: a `Composite` object containing all the observation key-spec pairs.
- `env.state_spec`: a `Composite` object containing all the input key-spec pairs (except action). For most stateful environments, this container will be empty.
- `env.action_spec`: a `TensorSpec` object representing the action spec.
- `env.reward_spec`: a `TensorSpec` object representing the reward spec.
- `env.done_spec`: a `TensorSpec` object representing the done-flag spec. See the section on trajectory termination below.
- `env.input_spec`: a `Composite` object containing all the input keys (`"full_action_spec"` and `"full_state_spec"`).
- `env.output_spec`: a `Composite` object containing all the output keys (`"full_observation_spec"`, `"full_reward_spec"` and `"full_done_spec"`).

If the environment carries non-tensor data, a `NonTensor` instance can be used.

Env specs: locks and batch size

Environment specs are locked by default (through a `spec_locked` arg passed to the env constructor). Locking specs means that any modification of the spec (or its children if it is a `Composite` instance) will require to unlock it. This can be done via the `set_spec_lock_()`. The reason specs are locked by default is that it makes it easy to cache values such as action or reset keys and the likes. Unlocking an env should only be done if it expected that the specs will be modified often (which, in principle, should be avoided). Modifications of the specs such as *env.observation_spec = new_spec* are allowed: under the hood, TorchRL will erase the cache, unlock the specs, make the modification and relock the specs if the env was previously locked.

Importantly, the environment spec shapes should contain the batch size, e.g. an environment with `env.batch_size == torch.Size([4])` should have an `env.action_spec` with shape `torch.Size([4, action_size])`. This is helpful when preallocation tensors, checking shape consistency etc.

Env methods

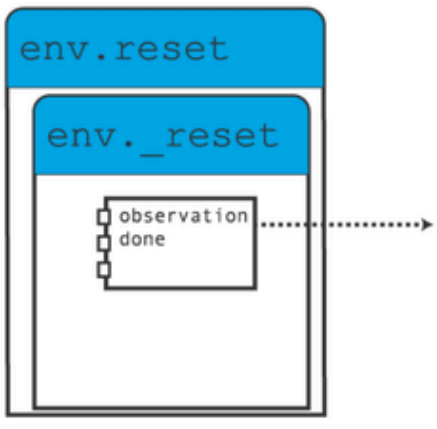
With these, the following methods are implemented:

- `env.reset()`: a reset method that may (but not necessarily requires to) take a `tensordict.TensorDict` input. It return the first tensordict of a rollout, usually containing a `"done"` state and a set of observations. If not present, a *"reward"* key will be instantiated with 0s and the appropriate shape.
- `env.step()`: a step method that takes a `tensordict.TensorDict` input containing an input action as well as other inputs (for model-based or stateless environments, for instance).
- `env.step_and_maybe_reset()`: executes a step, and (partially) resets the environments if it needs to. It returns the updated input with a `"next"` key containing the data of the next step, as well as a tensordict containing the input data for the next step (ie, reset or result or `step_mdp()`) This is done by reading the `done_keys` and assigning a `"_reset"` signal to each done state. This method allows to code non-stopping rollout functions with little effort:

```
>>> data_ = env.reset()
>>> result = []
>>> for i in range(N):
...     data, data_ = env.step_and_maybe_reset(data_)
...     result.append(data)
...
>>> result = torch.stack(result)
```

- `env.set_seed()`: a seeding method that will return the next seed to be used in a multi-env setting. This next seed is deterministically computed from the preceding one, such that one can seed multiple environments with a different seed without risking to overlap seeds in consecutive experiments, while still having reproducible results.
- `env.rollout()`: executes a rollout in the environment for a maximum number of steps (`max_steps=N`) and using a policy (`policy=model`). The policy should be coded using a `tensordict.nn.TensorDictModule` (or any other `tensordict.TensorDict`-compatible module). The resulting `tensordict.TensorDict` instance will be marked with a trailing `"time"` named dimension that can be used by other modules to treat this batched dimension as it should.

The following figure summarizes how a rollout is executed in torchrl.



TorchRL rollouts using TensorDict.

In brief, a TensorDict is created by the `reset()` method, then populated with an action by the policy before being passed to the `step()` method which writes the observations, done flag(s) and reward under the `"next"` entry. The result of this call is stored for delivery and the `"next"` entry is gathered by the `step_mdp()` function.

• NOTE

In general, all TorchRL environment have a `"done"` and `"terminated"` entry in their output tensordict. If they are not present by design, the `EnvBase` metaclass will ensure that every done or terminated is flanked with its dual. In TorchRL, `"done"` strictly refers to the union of all the end-of-trajectory signals and should be interpreted as "the last step of a trajectory" or equivalently "a signal indicating the need to reset". If the environment provides it (eg, Gymnasium), the truncation entry is also written in the `EnvBase.step()` output under a `"truncated"` entry. If the environment carries a single value, it will interpreted as a `"terminated"` signal by default. By default, TorchRL's collectors and rollout methods will be looking for the `"done"` entry to assess if the environment should be reset.

• NOTE

The *torchrl.collectors.utils.split_trajectories* function can be used to slice adjacent trajectories. It relies on a `"traj_ids"` entry in the input tensordict, or to the junction of `"done"` and `"truncated"` key if the `"traj_ids"` is missing.

• NOTE

In some contexts, it can be useful to mark the first step of a trajectory. TorchRL provides such functionality through the `InitTracker` transform.

Our environment tutorial provides more information on how to design a custom environment from scratch.

EnvBase(*args, **kwargs)	Abstract environment parent class.
GymLikeEnv(*args, **kwargs)	A gym-like env is an environment.
EnvMetaData(*, tensordict, specs, ...)	A class for environment meta-data storage and passing in multiprocessed settings.

Partial steps and partial resets

TorchRL allows environments to reset some but not all the environments, or run a step in one but not all environments. If there is only one environment in the batch, then a partial reset / step is also allowed with the behavior detailed below.

Batching environments and locking the batch

Before detailing what partial resets and partial steps do, we must distinguish cases where an environment has a batch size of its own (mostly stateful environments) or when the environment is just a mere module that, given an input of arbitrary size, batches the operations over all elements (mostly stateless environments).

This is controlled via the `batch_locked` attribute: a batch-locked environment requires all input tensordicts to have the same batch-size as the env's. Typical examples of these environments are `GymEnv` and related. Batch-unlocked envs are by contrast allowed to work with any input size. Notable examples are `BraxEnv` or `JumanjiEnv`.

Executing partial steps in a batch-unlocked environment is straightforward: one just needs to mask the part of the tensordict that does not need to be executed, pass the other part to *step* and merge the results with the previous input.

Batched environments (`ParallelEnv` and `SerialEnv`) can also deal with partial steps easily, they just pass the actions to the sub-environments that are required to be executed.

In all other cases, TorchRL assumes that the environment handles the partial steps correctly.

• WARNING

This means that custom environments may silently run the non-required steps as there is no way for torchrl to control what happens within the *_step* method!

Partial Steps

Partial steps are controlled via the temporary key *"_step"* which points to a boolean mask of the size of the tensordict that holds it. The classes armed to deal with this are:

- Batched environments: `ParallelEnv` and `SerialEnv` will dispatch the action to and only to the environments where *"_step"* is *True*;
- Batch-unlocked environments;
- Unbatched environments (i.e., environments without batch size). In these environments, the `step()` method will first look for a *"_step"* entry and, if present, act accordingly. If a `Transform` instance passes a *"_step"* entry to the tensordict, it is also captured by `TransformedEnv`'s own *_step* method which will skip the *base_env.step* as well as any further transformation.

When dealing with partial steps, the strategy is always to use the step output and mask missing values with the previous content of the input tensordict, if present, or a *0*-valued tensor if the tensor cannot be found. This means that if the input tensordict does not contain all the previous observations, then the output tensordict will be 0-valued for all the non-stepped elements. Within batched environments, data collectors and rollouts utils, this is an issue that is not observed because these classes handle the passing of data properly.

Partial steps are an essential feature of `rollout()` when *break_when_all_done* is *True*, as the environments with a *True* done state will need to be skipped during calls to *_step*.

The `ConditionalSkip` transform allows you to programmatically ask for (partial) step skips.

Partial Resets

Partial resets work pretty much like partial steps, but with the *"_reset"* entry.

The same restrictions of partial steps apply to partial resets.

Likewise, partial resets are an essential feature of `rollout()` when *break_when_any_done* is *True*, as the environments with a *True* done state will need to be reset, but not others.

See te following paragraph for a deep dive in partial resets within batched and vectorized environments.

Vectorized envs

Vectorized (or better: parallel) environments is a common feature in Reinforcement Learning where executing the environment step can be cpu-intensive. Some libraries such as gym3 or EnvPool offer interfaces to execute batches of environments simultaneously. While they often offer a very competitive computational advantage, they do not necessarily scale to the wide variety of environment libraries supported by TorchRL. Therefore, TorchRL offers its own, generic `ParallelEnv` class to run multiple environments in parallel. As this class inherits from `SerialEnv`, it enjoys the exact same API as other environment. Of course, a `ParallelEnv` will have a batch size that corresponds to its environment count:

• NOTE

Given the library's many optional dependencies (eg, Gym, Gymnasium, and many others) warnings can quickly become quite annoying in multiprocessed / distributed settings. By default, TorchRL filters out these warnings in sub-processes. If one still wishes to see these warnings, they can be displayed by setting `torchrl.filter_warnings_subprocess=False`.

It is important that your environment specs match the input and output that it sends and receives, as `ParallelEnv` will create buffers from these specs to communicate with the spawn processes. Check the `check_env_specs()` method for a sanity check.

Parallel environment

```
>>> def make_env():
...     return GymEnv("Pendulum-v1", from_pixels=True, g=9.81, device="cuda:0")
>>> check_env_specs(env)  # this must pass for ParallelEnv to work
>>> env = ParallelEnv(4, make_env)
>>> print(env.batch_size)
torch.Size([4])
```

`ParallelEnv` allows to retrieve the attributes from its contained environments: one can simply call:

Parallel environment attributes

```
>>> a, b, c, d = env.g  # gets the g-force of the various envs, which we set to 9.81 before
>>> print(a)
9.81
```

TorchRL uses a private `"_reset"` key to indicate to the environment which component (sub-environments or agents) should be reset. This allows to reset some but not all of the components.

The `"_reset"` key has two distinct functionalities:

1. During a call to `_reset()`, the `"_reset"` key may or may not be present in the input tensordict. TorchRL's convention is that the absence of the `"_reset"` key at a given `"done"` level indicates a total reset of that level (unless a `"_reset"` key was found at a level above, see details below). If it is present, it is expected that those entries and only those components where the `"_reset"` entry is `True` (along key and shape dimension) will be reset.

   The way an environment deals with the `"_reset"` keys in its `_reset()` method is proper to its class. Designing an environment that behaves according to `"_reset"` inputs is the developer's responsibility, as TorchRL has no control over the inner logic of `_reset()`. Nevertheless, the following point should be kept in mind when designing that method.
2. After a call to `_reset()`, the output will be masked with the `"_reset"` entries and the output of the previous `step()` will be written wherever the `"_reset"` was `False`. In practice, this means that if a `"_reset"` modifies data that isn't exposed by it, this modification will be lost. After this masking operation, the `"_reset"` entries will be erased from the `reset()` outputs.

It must be pointed out that `"_reset"` is a private key, and it should only be used when coding specific environment features that are internal facing. In other words, this should NOT be used outside of the library, and developers will keep the right to modify the logic of partial resets through `"_reset"` setting without preliminary warranty, as long as they don't affect TorchRL internal tests.

Finally, the following assumptions are made and should be kept in mind when designing reset functionalities:

- Each `"_reset"` is paired with a `"done"` entry (+ `"terminated"` and, possibly, `"truncated"`). This means that the following structure is not allowed: `TensorDict({"done": done, "nested": {"_reset": reset}}, [])`, as the `"_reset"` lives at a different nesting level than the `"done"`.
- A reset at one level does not preclude the presence of a `"_reset"` at lower levels, but it annihilates its effects. The reason is simply that whether the `"_reset"` at the root level corresponds to an `all()`, `any()` or custom call to the nested `"done"` entries cannot be known in advance, and it is explicitly assumed that the `"_reset"` at the root was placed there to supersede the nested values (for an example, have a look at `PettingZooWrapper` implementation where each group has one or more `"done"` entries associated which is aggregated at the root level with a `any` or `all` logic depending on the task).
- When calling `env.reset(tensordict)()` with a partial `"_reset"` entry that will reset some but not all the done sub-environments, the input data should contain the data of the sub-environments that are __not__ being reset. The reason for this constrain lies in the fact that the output of the `env._reset(data)` can only be predicted for the entries that are reset. For the others, TorchRL cannot know in advance if they will be meaningful or not. For instance, one could perfectly just pad the values of the non-reset components, in which case the non-reset data will be meaningless and should be discarded.

Below, we give some examples of the expected effect that `"_reset"` keys will have on an environment returning zeros after reset:

```
>>> # single reset at the root
>>> data = TensorDict({"val": [1, 1], "_reset": [False, True]}, [])
>>> env.reset(data)
>>> print(data.get("val"))  # only the second value is 0
tensor([1, 0])
>>> # nested resets
>>> data = TensorDict({
...     ("agent0", "val"): [1, 1], ("agent0", "_reset"): [False, True],
...     ("agent1", "val"): [2, 2], ("agent1", "_reset"): [True, False],
... }, [])
>>> env.reset(data)
>>> print(data.get(("agent0", "val")))  # only the second value is 0
tensor([1, 0])
>>> print(data.get(("agent1", "val")))  # only the first value is 0
tensor([0, 2])
>>> # nested resets are overridden by a "_reset" at the root
>>> data = TensorDict({
...     "_reset": [True, True],
...     ("agent0", "val"): [1, 1], ("agent0", "_reset"): [False, True],
...     ("agent1", "val"): [2, 2], ("agent1", "_reset"): [True, False],
... }, [])
>>> env.reset(data)
>>> print(data.get(("agent0", "val")))  # reset at the root overrides nested
tensor([0, 0])
>>> print(data.get(("agent1", "val")))  # reset at the root overrides nested
tensor([0, 0])
```

Parallel environment reset

```
>>> tensordict = TensorDict({"_reset": [[True], [False], [True], [True]]}, [4])
>>> env.reset(tensordict)  # eliminates the "_reset" entry
TensorDict(
    fields={
        terminated: Tensor(torch.Size([4, 1]), dtype=torch.bool),
        done: Tensor(torch.Size([4, 1]), dtype=torch.bool),
        pixels: Tensor(torch.Size([4, 500, 500, 3]), dtype=torch.uint8),
        truncated: Tensor(torch.Size([4, 1]), dtype=torch.bool),
    batch_size=torch.Size([4]),
    device=None,
    is_shared=True)
```

• NOTE

A note on performance: launching a `ParallelEnv` can take quite some time as it requires to launch as many python instances as there are processes. Due to the time that it takes to run `import torch` (and other imports), starting the parallel env can be a bottleneck. This is why, for instance, TorchRL tests are so slow. Once the environment is launched, a great speedup should be observed.

• NOTE

TorchRL requires precise specs: Another thing to take in consideration is that `ParallelEnv` (as well as data collectors) will create data buffers based on the environment specs to pass data from one process to another. This means that a misspecified spec (input, observation or reward) will cause a breakage at runtime as the data can't be written on the preallocated buffer. In general, an environment should be tested using the `check_env_specs()` test function before being used in a `ParallelEnv`. This function will raise an assertion error whenever the preallocated buffer and the collected data mismatch.

We also offer the `SerialEnv` class that enjoys the exact same API but is executed serially. This is mostly useful for testing purposes, when one wants to assess the behavior of a `ParallelEnv` without launching the subprocesses.

In addition to `ParallelEnv`, which offers process-based parallelism, we also provide a way to create multithreaded environments with `MultiThreadedEnv`. This class uses EnvPool library underneath, which allows for higher performance, but at the same time restricts flexibility - one can only create environments implemented in `EnvPool`. This covers many popular RL environments types (Atari, Classic Control, etc.), but one can not use an arbitrary TorchRL environment, as it is possible with `ParallelEnv`. Run *benchmarks/benchmark_batched_envs.py* to compare performance of different ways to parallelize batched environments.

`SerialEnv`(*args, **kwargs)	Creates a series of environments in the same process.Batched environments allow the user to query an arbitrary method / attribute of the environment running remotely.
`ParallelEnv`(*args, **kwargs)	Creates one environment per process.
`EnvCreator`(create_env_fn[, ...])	Environment creator class.

Async environments

Asynchronous environments allow for parallel execution of multiple environments, which can significantly speed up the data collection process in reinforcement learning.

The *AsyncEnvPool* class and its subclasses provide a flexible interface for managing these environments using different backends, such as threading and multiprocessing.

The *AsyncEnvPool* class serves as a base class for asynchronous environment pools, providing a common interface for managing multiple environments concurrently. It supports different backends for parallel execution, such as threading and multiprocessing, and provides methods for asynchronous stepping and resetting of environments.

Contrary to `ParallelEnv`, `AsyncEnvPool` and its subclasses permit the execution of a given set of sub-environments while another task performed, allowing for complex asynchronous jobs to be run at the same time. For instance, it is possible to execute some environments while the policy is running based on the output of others.

This family of classes is particularly interesting when dealing with environments that have a high (and/or variable) latency.

• NOTE

This class and its subclasses should work when nested in with `TransformedEnv` and batched environments, but users won't currently be able to use the async features of the base environment when it's nested in these classes. One should prefer nested transformed envs within an *AsyncEnvPool* instead. If this is not possible, please raise an issue.

Classes

- `AsyncEnvPool`: A base class for asynchronous environment pools. It determines the backend implementation to use based on the provided arguments and manages the lifecycle of the environments.
- `ProcessorAsyncEnvPool`: An implementation of `AsyncEnvPool` using multiprocessing for parallel execution of environments. This class manages a pool of environments, each running in its own process, and provides methods for asynchronous stepping and resetting of environments using inter-process communication. It is automatically instantiated when *"multiprocessing"* is passed as a backend during the `AsyncEnvPool` instantiation.
- `ThreadingAsyncEnvPool`: An implementation of `AsyncEnvPool` using threading for parallel execution of environments. This class manages a pool of environments, each running in its own thread, and provides methods for asynchronous stepping and resetting of environments using a thread pool executor. It is automatically instantiated when *"threading"* is passed as a backend during the `AsyncEnvPool` instantiation.

Example

```
>>> from functools import partial
>>> from torchrl.envs import AsyncEnvPool, GymEnv
>>> import torch
>>> # Choose backend
>>> backend = "threading"
>>> env = AsyncEnvPool(
>>>     [partial(GymEnv, "Pendulum-v1"), partial(GymEnv, "CartPole-v1")],
>>>     stack="lazy",
>>>     backend=backend
>>> )
>>> # Execute a synchronous reset
>>> reset = env.reset()
>>> print(reset)
>>> # Execute a synchronous step
>>> s = env.rand_step(reset)
>>> print(s)
>>> # Execute an asynchronous step in env 0
>>> s0 = s[0]
>>> s0["action"] = torch.randn(1).clamp(-1, 1)
>>> s0["env_index"] = 0
>>> env.async_step_send(s0)
>>> # Receive data
>>> s0_result = env.async_step_recv()
>>> print('result', s0_result)
>>> # Close env
>>> env.close()
```

`AsyncEnvPool`(*args, **kwargs)	A base class for asynchronous environment pools, providing a common interface for managing multiple environments concurrently.
`ProcessorAsyncEnvPool`(*args, **kwargs)	An implementation of *AsyncEnvPool* using multiprocessing for parallel execution of environments.
`ThreadingAsyncEnvPool`(*args, **kwargs)	An implementation of *AsyncEnvPool* using threading for parallel execution of environments.

Custom native TorchRL environments

TorchRL offers a series of custom built-in environments.

`ChessEnv`(*args, **kwargs)	A chess environment that follows the TorchRL API.
`PendulumEnv`(*args, **kwargs)	A stateless Pendulum environment.
`TicTacToeEnv`(*args, **kwargs)	A Tic-Tac-Toe implementation.

`LLMHashingEnv`(*args, **kwargs)	A text generation environment that uses a hashing module to identify unique observations.

Multi-agent environments

TorchRL supports multi-agent learning out-of-the-box. *The same classes used in a single-agent learning pipeline can be seamlessly used in multi-agent contexts, without any modification or dedicated multi-agent infrastructure.*

In this view, environments play a core role for multi-agent. In multi-agent environments, many decision-making agents act in a shared world. Agents can observe different things, act in different ways and also be rewarded differently. Therefore, many paradigms exist to model multi-agent environments (DecPODPs, Markov Games). Some of the main differences between these paradigms include:

- **observation** can be per-agent and also have some shared components
- **reward** can be per-agent or shared
- **done** (and `"truncated"` or `"terminated"`) can be per-agent or shared.

TorchRL accommodates all these possible paradigms thanks to its `tensordict.TensorDict` data carrier. In particular, in multi-agent environments, per-agent keys will be carried in a nested "agents" TensorDict. This TensorDict will have the additional agent dimension and thus group data that is different for each agent. The shared keys, on the other hand, will be kept in the first level, as in single-agent cases.

Let's look at an example to understand this better. For this example we are going to use VMAS, a multi-robot task simulator also based on PyTorch, which runs parallel batched simulation on device.

We can create a VMAS environment and look at what the output from a random step looks like:

Example of multi-agent step tensordict

```
>>> from torchrl.envs.libs.vmas import VmasEnv
>>> env = VmasEnv("balance", num_envs=3, n_agents=5)
>>> td = env.rand_step()
>>> td
TensorDict(
    fields={
        agents: TensorDict(
            fields={
                action: Tensor(shape=torch.Size([3, 5, 2]))},
            batch_size=torch.Size([3, 5])),
        next: TensorDict(
            fields={
                agents: TensorDict(
                    fields={
                        info: TensorDict(
                            fields={
                                ground_rew: Tensor(shape=torch.Size([3, 5, 1])),
                                pos_rew: Tensor(shape=torch.Size([3, 5, 1]))},
                            batch_size=torch.Size([3, 5])),
                        observation: Tensor(shape=torch.Size([3, 5, 16])),
                        reward: Tensor(shape=torch.Size([3, 5, 1]))},
                    batch_size=torch.Size([3, 5])),
                done: Tensor(shape=torch.Size([3, 1]))},
            batch_size=torch.Size([3]))},
    batch_size=torch.Size([3]))
```

We can observe that *keys that are shared by all agents*, such as **done** are present in the root tensordict with batch size (*num_envs,*), which represents the number of environments simulated.

On the other hand, *keys that are different between agents*, such as **action**, **reward**, **observation**, and **info** are present in the nested "agents" tensordict with batch size (*num_envs, n_agents*), which represents the additional agent dimension.

Multi-agent tensor specs will follow the same style as in tensordicts. Specs relating to values that vary between agents will need to be nested in the "agents" entry.

Here is an example of how specs can be created in a multi-agent environment where only the done flag is shared across agents (as in VMAS):

Example of multi-agent spec creation

```
>>> action_specs = []
>>> observation_specs = []
>>> reward_specs = []
>>> info_specs = []
>>> for i in range(env.n_agents):
...     action_specs.append(agent_i_action_spec)
...     reward_specs.append(agent_i_reward_spec)
...     observation_specs.append(agent_i_observation_spec)
>>> env.action_spec = Composite(
...     {
...         "agents": Composite(
...             {"action": torch.stack(action_specs)}, shape=(env.n_agents,)
...         )
...     }
...)
>>> env.reward_spec = Composite(
...     {
...         "agents": Composite(
...             {"reward": torch.stack(reward_specs)}, shape=(env.n_agents,)
...         )
...     }
...)
>>> env.observation_spec = Composite(
...     {
...         "agents": Composite(
...             {"observation": torch.stack(observation_specs)}, shape=(env.n_agents,)
...         )
...     }
...)
>>> env.done_spec = Categorical(
...     n=2,
...     shape=torch.Size((1,)),
...     dtype=torch.bool,
... )
```

As you can see, it is very simple! Per-agent keys will have the nested composite spec and shared keys will follow single agent standards.

• NOTE

Since reward, done and action keys may have the additional "agent" prefix (e.g., (*"agents","action"*)), the default keys used in the arguments of other TorchRL components (e.g. "action") will not match exactly. Therefore, TorchRL provides the *env.action_key*, *env.reward_key*, and *env.done_key* attributes, which will automatically point to the right key to use. Make sure you pass these attributes to the various components in TorchRL to inform them of the right key (e.g., the *loss.set_keys*() function).

• NOTE

TorchRL abstracts these nested specs away for ease of use. This means that accessing *env.reward_spec* will always return the leaf spec if the accessed spec is Composite. Therefore, if in the example above we run *env.reward_spec* after env creation, we would get the same output as *torch.stack*(*reward_specs*)}. To get the full composite spec with the "agents" key, you can run *env.output_spec*[*"full_reward_spec"*]. The same is valid for action and done specs. Note that *env.reward_spec* == *env.output_spec*[*"full_reward_spec"*][*env.reward_key*].

`MarlGroupMapType`(value)	Marl Group Map Type.
`check_marl_grouping`(group_map, agent_names)	Check MARL group map.

Auto-resetting Envs

Auto-resetting environments are environments where calls to `reset()` are not expected when the environment reaches a `"done"` state during a rollout, as the reset happens automatically. Usually, in such cases the observations delivered with the done and reward (which effectively result from performing the action in the environment) are actually the first observations of a new episode, and not the last observations of the current episode.

To handle these cases, torchrl provides a `AutoResetTransform` that will copy the observations that result from the call to *step* to the next *reset* and skip the calls to *reset* during rollouts (in both `rollout()` and `SyncDataCollector` iterations). This transform class also provides a fine-grained control over the behavior to be adopted for the invalid observations, which can be masked with *"nan"* or any other values, or not masked at all.

To tell torchrl that an environment is auto-resetting, it is sufficient to provide an `auto_reset` argument during construction. If provided, an `auto_reset_replace` argument can also control whether the values of the last observation of an episode should be replaced with some placeholder or not.

```
>>> from torchrl.envs import GymEnv
>>> from torchrl.envs import set_gym_backend
>>> import torch
>>> torch.manual_seed(0)
>>>
>>> class AutoResettingGymEnv(GymEnv):
...     def _step(self, tensordict):
...         tensordict = super()._step(tensordict)
...         if tensordict["done"].any():
...             td_reset = super().reset()
...             tensordict.update(td_reset.exclude(*self.done_keys))
...         return tensordict
...
...     def _reset(self, tensordict=None):
...         if tensordict is not None and "_reset" in tensordict:
...             return tensordict.copy()
...         return super()._reset(tensordict)
>>>
>>> with set_gym_backend("gym"):
...     env = AutoResettingGymEnv("CartPole-v1", auto_reset=True, auto_reset_replace=True)
...     env.set_seed(0)
...     r = env.rollout(30, break_when_any_done=False)
>>> print(r["next", "done"].squeeze())
tensor([False, False, False, False, False, False, False, False, False, False,
        False, False, False,  True, False, False, False, False, False, False,
        False, False, False, False, False,  True, False, False, False, False])
>>> print("observation after reset are set as nan", r["next", "observation"])
observation after reset are set as nan tensor([[-4.3633e-02, -1.4877e-01,  1.2849e-02,  2.7584e-01],
        [-4.6609e-02,  4.6166e-02,  1.8366e-02, -1.2761e-02],
        [-4.5685e-02,  2.4102e-01,  1.8111e-02, -2.9959e-01],
        [-4.0865e-02,  4.5644e-02,  1.2119e-02, -1.2542e-03],
        [-3.9952e-02,  2.4059e-01,  1.2094e-02, -2.9009e-01],
        [-3.5140e-02,  4.3554e-01,  6.2920e-03, -5.7893e-01],
        [-2.6429e-02,  6.3057e-01, -5.2867e-03, -8.6963e-01],
        [-1.3818e-02,  8.2576e-01, -2.2679e-02, -1.1640e+00],
        [ 2.6972e-03,  1.0212e+00, -4.5959e-02, -1.4637e+00],
        [ 2.3121e-02,  1.2168e+00, -7.5232e-02, -1.7704e+00],
        [ 4.7457e-02,  1.4127e+00, -1.1064e-01, -2.0854e+00],
        [ 7.5712e-02,  1.2189e+00, -1.5235e-01, -1.8289e+00],
        [ 1.0009e-01,  1.0257e+00, -1.8893e-01, -1.5872e+00],
        [        nan,         nan,         nan,         nan],
        [-3.9405e-02, -1.7766e-01, -1.0403e-02,  3.0626e-01],
        [-4.2959e-02, -3.7263e-01, -4.2775e-03,  5.9564e-01],
        [-5.0411e-02, -5.6769e-01,  7.6354e-03,  8.8698e-01],
        [-6.1765e-02, -7.6292e-01,  2.5375e-02,  1.1820e+00],
        [-7.7023e-02, -9.5836e-01,  4.9016e-02,  1.4826e+00],
        [-9.6191e-02, -7.6387e-01,  7.8667e-02,  1.2056e+00],
        [-1.1147e-01, -9.5991e-01,  1.0278e-01,  1.5219e+00],
        [-1.3067e-01, -7.6617e-01,  1.3322e-01,  1.2629e+00],
        [-1.4599e-01, -5.7298e-01,  1.5848e-01,  1.0148e+00],
        [-1.5745e-01, -7.6982e-01,  1.7877e-01,  1.3527e+00],
        [-1.7285e-01, -9.6668e-01,  2.0583e-01,  1.6956e+00],
        [        nan,         nan,         nan,         nan],
        [-4.3962e-02,  1.9845e-01, -4.5015e-02, -2.5903e-01],
        [-3.9993e-02,  3.9418e-01, -5.0196e-02, -5.6557e-01],
        [-3.2109e-02,  5.8997e-01, -6.1507e-02, -8.7363e-01],
        [-2.0310e-02,  3.9574e-01, -7.8980e-02, -6.0090e-01]])
```

Dynamic Specs

Running environments in parallel is usually done via the creation of memory buffers used to pass information from one process to another. In some cases, it may be impossible to forecast whether an environment will or will not have consistent inputs or outputs during a rollout, as their shape may be variable. We refer to this as dynamic specs.

TorchRL is capable of handling dynamic specs, but the batched environments and collectors will need to be made aware of this feature. Note that, in practice, this is detected automatically.

To indicate that a tensor will have a variable size along a dimension, one can set the size value as `-1` for the desired dimensions. Because the data cannot be stacked contiguously, calls to `env.rollout` need to be made with the `return_contiguous=False` argument. Here is a working example:

```
>>> from torchrl.envs import EnvBase
>>> from torchrl.data import Unbounded, Composite, Bounded, Binary
>>> import torch
>>> from tensordict import TensorDict, TensorDictBase
>>>
>>> class EnvWithDynamicSpec(EnvBase):
...     def __init__(self, max_count=5):
...         super().__init__(batch_size=())
...         self.observation_spec = Composite(
...             observation=Unbounded(shape=(3, -1, 2)),
...         )
...         self.action_spec = Bounded(low=-1, high=1, shape=(2,))
...         self.full_done_spec = Composite(
...             done=Binary(1, shape=(1,), dtype=torch.bool),
...             terminated=Binary(1, shape=(1,), dtype=torch.bool),
...             truncated=Binary(1, shape=(1,), dtype=torch.bool),
...         )
...         self.reward_spec = Unbounded((1,), dtype=torch.float)
...         self.count = 0
...         self.max_count = max_count
...
...     def _reset(self, tensordict=None):
...         self.count = 0
...         data = TensorDict(
...             {
...                 "observation": torch.full(
...                     (3, self.count + 1, 2),
...                     self.count,
...                     dtype=self.observation_spec["observation"].dtype,
...                 )
...             }
...         )
...         data.update(self.done_spec.zero())
...         return data
...
...     def _step(
...         self,
...         tensordict: TensorDictBase,
...     ) -> TensorDictBase:
...         self.count += 1
...         done = self.count >= self.max_count
...         observation = TensorDict(
...             {
...                 "observation": torch.full(
...                     (3, self.count + 1, 2),
...                     self.count,
...                     dtype=self.observation_spec["observation"].dtype,
...                 )
...             }
...         )
...         done = self.full_done_spec.zero() | done
...         reward = self.full_reward_spec.zero()
...         return observation.update(done).update(reward)
...
...     def _set_seed(self, seed: Optional[int]) -> None:
...         self.manual_seed = seed
...         return seed
>>> env = EnvWithDynamicSpec()
>>> print(env.rollout(5, return_contiguous=False))
LazyStackedTensorDict(
    fields={
        action: Tensor(shape=torch.Size([5, 2]), device=cpu, dtype=torch.float32, is_shared=False),
        done: Tensor(shape=torch.Size([5, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        next: LazyStackedTensorDict(
            fields={
                done: Tensor(shape=torch.Size([5, 1]), device=cpu, dtype=torch.bool, is_shared=False),
                observation: Tensor(shape=torch.Size([5, 3, -1, 2]), device=cpu, dtype=torch.float32, is_shared=False),
                reward: Tensor(shape=torch.Size([5, 1]), device=cpu, dtype=torch.float32, is_shared=False),
                terminated: Tensor(shape=torch.Size([5, 1]), device=cpu, dtype=torch.bool, is_shared=False),
                truncated: Tensor(shape=torch.Size([5, 1]), device=cpu, dtype=torch.bool, is_shared=False)},
            exclusive_fields={
            },
            batch_size=torch.Size([5]),
            device=None,
            is_shared=False,
            stack_dim=0),
        observation: Tensor(shape=torch.Size([5, 3, -1, 2]), device=cpu, dtype=torch.float32, is_shared=False),
        terminated: Tensor(shape=torch.Size([5, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        truncated: Tensor(shape=torch.Size([5, 1]), device=cpu, dtype=torch.bool, is_shared=False)},
    exclusive_fields={
    },
    batch_size=torch.Size([5]),
    device=None,
    is_shared=False,
    stack_dim=0)
```

• WARNING

The absence of memory buffers in `ParallelEnv` and in data collectors can impact performance of these classes dramatically. Any such usage should be carefully benchmarked against a plain execution on a single process, as serializing and deserializing large numbers of tensors can be very expensive.

Currently, `check_env_specs()` will pass for dynamic specs where a shape varies along some dimensions, but not when a key is present during a step and absent during others, or when the number of dimensions varies.

Transforms

In most cases, the raw output of an environment must be treated before being passed to another object (such as a policy or a value operator). To do this, TorchRL provides a set of transforms that aim at reproducing the transform logic of *torch.distributions.Transform* and *torchvision.transforms*. Our environment tutorial provides more information on how to design a custom transform.

Transformed environments are build using the `TransformedEnv` primitive. Composed transforms are built using the `Compose` class:

Transformed environment

```
>>> base_env = GymEnv("Pendulum-v1", from_pixels=True, device="cuda:0")
>>> transform = Compose(ToTensorImage(in_keys=["pixels"]), Resize(64, 64, in_keys=["pixels"]))
>>> env = TransformedEnv(base_env, transform)
```

Transforms are usually subclasses of `Transform`, although any `Callable[[TensorDictBase], TensorDictBase]`.

By default, the transformed environment will inherit the device of the `base_env` that is passed to it. The transforms will then be executed on that device. It is now apparent that this can bring a significant speedup depending on the kind of operations that is to be computed.

A great advantage of environment wrappers is that one can consult the environment up to that wrapper. The same can be achieved with TorchRL transformed environments: the `parent` attribute will return a new `TransformedEnv` with all the transforms up to the transform of interest. Re-using the example above:

Transform parent

```
>>> resize_parent = env.transform[-1].parent  # returns the same as TransformedEnv(base_env, transform[:-1])
```

Transformed environment can be used with vectorized environments. Since each transform uses a `"in_keys"`/`"out_keys"` set of keyword argument, it is also easy to root the transform graph to each component of the observation data (e.g. pixels or states etc).

Forward and inverse transforms

Transforms also have an `inv()` method that is called before the action is applied in reverse order over the composed transform chain. This allows applying transforms to data in the environment before the action is taken in the environment. The keys to be included in this inverse transform are passed through the *"in_keys_inv"* keyword argument, and the out-keys default to these values in most cases:

Inverse transform

```
>>> env.append_transform(DoubleToFloat(in_keys_inv=["action"]))  # will map the action from float32 to float64 before calling the base_env.step
```

The following paragraphs detail how one can think about what is to be considered *in_* or *out_* features.

Understanding Transform Keys

In transforms, *in_keys* and *out_keys* define the interaction between the base environment and the outside world (e.g., your policy):

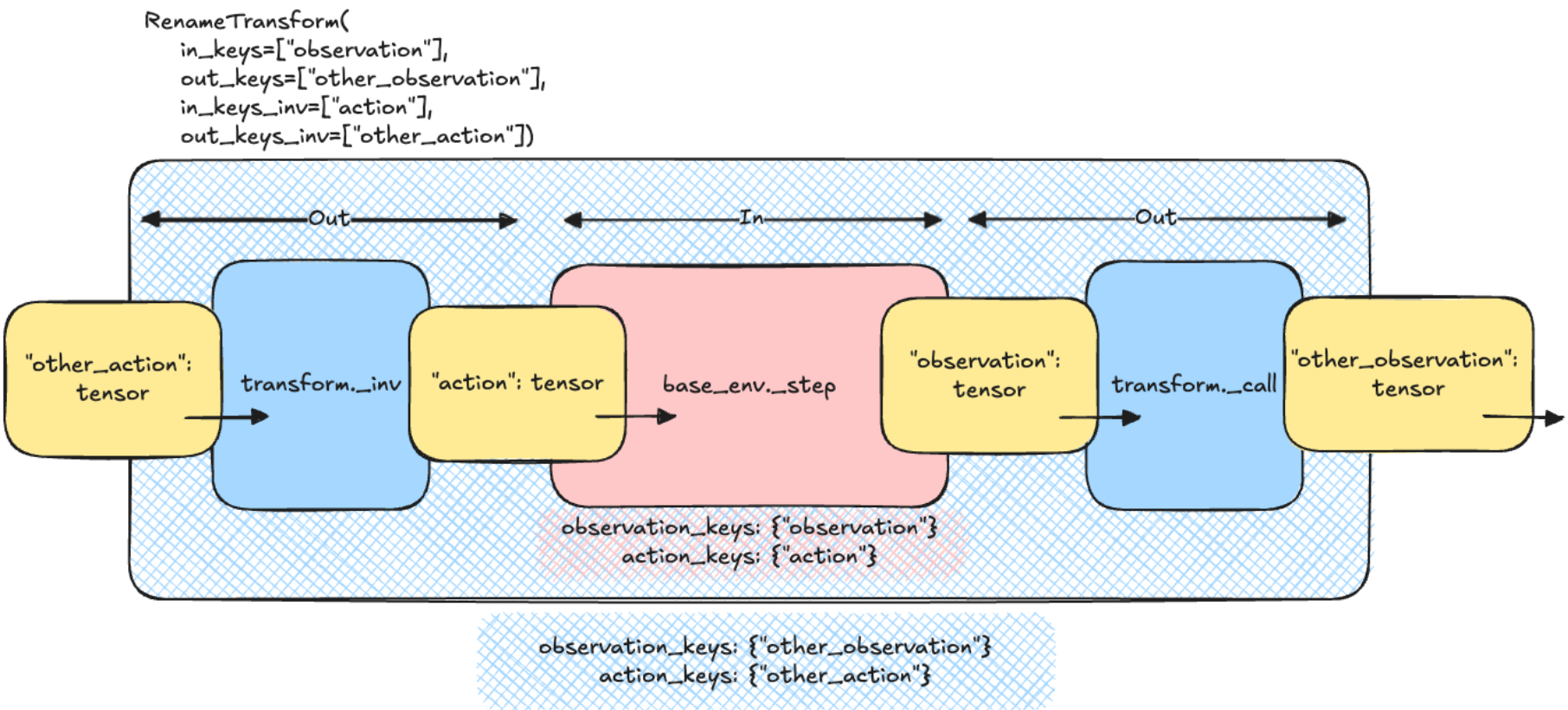
- *in_keys* refers to the base environment's perspective (inner = *base_env* of the `TransformedEnv`).
- *out_keys* refers to the outside world (outer = *policy*, *agent*, etc.).

For example, with *in_keys*=[*"obs"*] and *out_keys*=[*"obs_standardized"*], the policy will "see" a standardized observation, while the base environment outputs a regular observation.

Similarly, for inverse keys:

- *in_keys_inv* refers to entries as seen by the base environment.
- *out_keys_inv* refers to entries as seen or produced by the policy.

The following figure illustrates this concept for the `RenameTransform` class: the input *TensorDict* of the *step* function must include the *out_keys_inv* as they are part of the outside world. The transform changes these names to match the names of the inner, base environment using the *in_keys_inv*. The inverse process is executed with the output tensordict, where the *in_keys* are mapped to the corresponding *out_keys*.



Rename transform logic

• NOTE

During a call to *inv*, the transforms are executed in reversed order (compared to the forward / step mode).

Transforming Tensors and Specs

When transforming actual tensors (coming from the policy), the process is schematically represented as:

```
>>> for t in reversed(self.transform):
...     td = t.inv(td)
```

This starts with the outermost transform to the innermost transform, ensuring the action value exposed to the policy is properly transformed.

For transforming the action spec, the process should go from innermost to outermost (similar to observation specs):

```
>>> def transform_action_spec(self, action_spec):
...     for t in self.transform:
...         action_spec = t.transform_action_spec(action_spec)
...     return action_spec
```

A pseudocode for a single transform_action_spec could be:

```
>>> def transform_action_spec(self, action_spec):
...    return spec_from_random_values(self._apply_transform(action_spec.rand()))
```

This approach ensures that the “outside” spec is inferred from the “inside” spec. Note that we did not call *_inv_apply_transform* but *_apply_transform* on purpose!

Exposing Specs to the Outside World

TransformedEnv will expose the specs corresponding to the *out_keys_inv* for actions and states. For example, with `ActionDiscretizer`, the environment’s action (e.g., *“action”*) is a float-valued tensor that should not be generated when using `rand_action()` with the transformed environment. Instead, *“action_discrete”* should be generated, and its continuous counterpart obtained from the transform. Therefore, the user should see the *“action_discrete”* entry being exposed, but not *“action”*.

Designing your own Transform

To create a basic, custom transform, you need to subclass the *Transform* class and implement the `_apply_transform()` method. Here’s an example of a simple transform that adds 1 to the observation tensor:

```
>>> class AddOneToObs(Transform):
...     """A transform that adds 1 to the observation tensor."""
...
...     def __init__(self):
...         super().__init__(in_keys=["observation"], out_keys=["observation"])
...
...     def _apply_transform(self, obs: torch.Tensor) -> torch.Tensor:
...         return obs + 1
```

Tips for subclassing *Transform*

There are various ways of subclassing a transform. The things to take into considerations are:

- Is the transform identical for each tensor / item being transformed? Use `_apply_transform()` and `_inv_apply_transform()`.
- The transform needs access to the input data to env.step as well as output? Rewrite `_step()`. Otherwise, rewrite `_call()` (or `_inv_call()`).
- Is the transform to be used within a replay buffer? Overwrite `forward()`, `inv()`, `_apply_transform()` or `_inv_apply_transform()`.
- Within a transform, you can access (and make calls to) the parent environment using `parent` (the base env + all transforms till this one) or `container()` (The object that encapsulates the transform).
- Don’t forget to edits the specs if needed: top level: `transform_output_spec()`, `transform_input_spec()`. Leaf level: `transform_observation_spec()`, `transform_action_spec()`, `transform_state_spec()`, `transform_reward_spec()` and `transform_reward_spec()`.

For practical examples, see the methods listed above.

You can use a transform in an environment by passing it to the TransformedEnv constructor:

```
>>> env = TransformedEnv(GymEnv("Pendulum-v1"), AddOneToObs())
```

You can compose multiple transforms together using the Compose class:

```
>>> transform = Compose(AddOneToObs(), RewardSum())
>>> env = TransformedEnv(GymEnv("Pendulum-v1"), transform)
```

Inverse Transforms

Some transforms have an inverse transform that can be used to undo the transformation. For example, the AddOneToAction transform has an inverse transform that subtracts 1 from the action tensor:

```
>>> class AddOneToAction(Transform):
...     """A transform that adds 1 to the action tensor."""
...     def __init__(self):
...         super().__init__(in_keys=[], out_keys=[], in_keys_inv=["action"], out_keys_inv=["action"])
...     def _inv_apply_transform(self, action: torch.Tensor) -> torch.Tensor:
...         return action + 1
```

Using a Transform with a Replay Buffer

You can use a transform with a replay buffer by passing it to the ReplayBuffer constructor:

Cloning transforms

Because transforms appended to an environment are "registered" to this environment through the `transform.parent` property, when manipulating transforms we should keep in mind that the parent may come and go following what is being done with the transform. Here are some examples: if we get a single transform from a `Compose` object, this transform will keep its parent:

```
>>> third_transform = env.transform[2]
>>> assert third_transform.parent is not None
```

This means that using this transform for another environment is prohibited, as the other environment would replace the parent and this may lead to unexpected behviours. Fortunately, the `Transform` class comes with a `clone()` method that will erase the parent while keeping the identity of all the registered buffers:

```
>>> TransformedEnv(base_env, third_transform)  # raises an Exception as third_transform already has a parent
>>> TransformedEnv(base_env, third_transform.clone())  # works
```

On a single process or if the buffers are placed in shared memory, this will result in all the clone transforms to keep the same behavior even if the buffers are changed in place (which is what will happen with the `CatFrames` transform, for instance). In distributed settings, this may not hold and one should be careful about the expected behavior of the cloned transforms in this context. Finally, notice that indexing multiple transforms from a `Compose` transform may also result in loss of parenthood for these transforms: the reason is that indexing a `Compose` transform results in another `Compose` transform that does not have a parent environment. Hence, we have to clone the sub-transforms to be able to create this other composition:

```
>>> env = TransformedEnv(base_env, Compose(transform1, transform2, transform3))
>>> last_two = env.transform[-2:]
>>> assert isinstance(last_two, Compose)
>>> assert last_two.parent is None
>>> assert last_two[0] is not transform2
>>> assert isinstance(last_two[0], type(transform2))  # and the buffers will match
>>> assert last_two[1] is not transform3
>>> assert isinstance(last_two[1], type(transform3))  # and the buffers will match
```

`Transform`([in_keys, out_keys, in_keys_inv, ...])	Base class for environment transforms, which modify or create new data in a tensordict.
`TransformedEnv`(*args, **kwargs)	A transformed_in environment.
`ActionDiscretizer`(num_intervals[, ...])	A transform to discretize a continuous action space.
`ActionMask`([action_key, mask_key])	An adaptive action masker.
`AutoResetEnv`(*args, **kwargs)	A subclass for auto-resetting envs.
`AutoResetTransform`(*[, replace, fill_float, ...])	A transform for auto-resetting environments.
`BatchSizeTransform`(*[, batch_size, ...])	A transform to modify the batch-size of an environment.
`BinarizeReward`([in_keys, out_keys])	Maps the reward to a binary value (0 or 1) if the reward is null or non-null, respectively.
`BurnInTransform`(modules, burn_in[, out_keys])	Transform to partially burn-in data sequences.
`CatFrames`(N, dim[, in_keys, out_keys, ...])	Concatenates successive observation frames into a single tensor.
`CatTensors`([in_keys, out_key, dim, ...])	Concatenates several keys in a single tensor.
`CenterCrop`(w[, h, in_keys, out_keys])	Crops the center of an image.
`ClipTransform`([in_keys, out_keys, ...])	A transform to clip input (state, action) or output (observation, reward) values.

`Compose`(*transforms)	Composes a chain of transforms.
`ConditionalPolicySwitch`(policy, condition)	A transform that conditionally switches between policies based on a specified condition.
`ConditionalSkip`(cond)	A transform that skips steps in the env if certain conditions are met.
`Crop`(w[, h, top, left, in_keys, out_keys])	Crops the input image at the specified location and output size.
`DTypeCastTransform`(dtype_in, dtype_out[, ...])	Casts one dtype to another for selected keys.
`DeviceCastTransform`(device[, orig_device, ...])	Moves data from one device to another.
`DiscreteActionProjection`(...[, action_key, ...])	Projects discrete actions from a high dimensional space to a low dimensional space.
`DoubleToFloat`([in_keys, out_keys, ...])	Casts one dtype to another for selected keys.
`EndOfLifeTransform`([eol_key, lives_key, ...])	Registers the end-of-life signal from a Gym env with a *lives* method.
`ExcludeTransform`(*excluded_keys[, inverse])	Excludes keys from the data.
`FiniteTensorDictCheck`()	This transform will check that all the items of the tensordict are finite, and raise an exception if they are not.
`FlattenObservation`(first_dim, last_dim[, ...])	Flatten adjacent dimensions of a tensor.
`FrameSkipTransform`([frame_skip])	A frame-skip transform.
`GrayScale`([in_keys, out_keys])	Turns a pixel observation to grayscale.
`Hash`(in_keys, out_keys[, in_keys_inv, ...])	Adds a hash value to a tensordict.
`InitTracker`([init_key])	Reset tracker.
`KLRewardTransform`(actor[, coef, in_keys, ...])	A transform to add a KL[pi_current||pi_0] correction term to the reward.
`LineariseRewards`(in_keys[, out_keys, weights])	Transforms a multi-objective reward signal to a single-objective one via a weighted sum.
`MultiAction`(*[, dim, stack_rewards, ...])	A transform to execute multiple actions in the parent environment.
`NoopResetEnv`([noops, random])	Runs a series of random actions when an environment is reset.
`ObservationNorm`([loc, scale, in_keys, ...])	Observation affine transformation layer.
`ObservationTransform`([in_keys, out_keys, ...])	Abstract class for transformations of the observations.
`PermuteTransform`(dims[, in_keys, out_keys, ...])	Permutation transform.
`PinMemoryTransform`()	Calls pin_memory on the tensordict to facilitate writing on CUDA devices.
`R3MTransform`(*args, **kwargs)	R3M Transform class.
`RandomCropTensorDict`(sub_seq_len[, ...])	A trajectory sub-sampler for ReplayBuffer and modules.
`RemoveEmptySpecs`([in_keys, out_keys, ...])	Removes empty specs and content from an environment.

`RenameTransform`(in_keys, out_keys[, ...])	A transform to rename entries in the output tensordict (or input tensordict via the inverse keys).
`Resize`(w[, h, interpolation, in_keys, out_keys])	Resizes a pixel observation.
`Reward2GoTransform`([gamma, in_keys, ...])	Calculates the reward to go based on the episode reward and a discount factor.
`RewardClipping`([clamp_min, clamp_max, ...])	Clips the reward between *clamp_min* and *clamp_max*.
`RewardScaling`(loc, scale[, in_keys, ...])	Affine transform of the reward.
`RewardSum`([in_keys, out_keys, reset_keys, ...])	Tracks episode cumulative rewards.
`SelectTransform`(*selected_keys[, ...])	Select keys from the input tensordict.
`SignTransform`([in_keys, out_keys, ...])	A transform to compute the signs of TensorDict values.
`SqueezeTransform`(*args, **kwargs)	Removes a dimension of size one at the specified position.
`Stack`(in_keys, out_key[, in_key_inv, ...])	Stacks tensors and tensordicts.
`StepCounter`([max_steps, truncated_key, ...])	Counts the steps from a reset and optionally sets the truncated state to `True` after a certain number of steps.
`TargetReturn`(target_return[, mode, in_keys, ...])	Sets a target return for the agent to achieve in the environment.
`TensorDictPrimer`([primers, random, ...])	A primer for TensorDict initialization at reset time.
`TimeMaxPool`([in_keys, out_keys, T, reset_key])	Take the maximum value in each position over the last T observations.
`Timer`([out_keys, time_key])	A transform that measures the time intervals between *inv* and *call* operations in an environment.
`Tokenizer`([in_keys, out_keys, in_keys_inv, ...])	Applies a tokenization operation on the specified inputs.
`ToTensorImage`([from_int, unsqueeze, dtype, ...])	Transforms a numpy-like image (W x H x C) to a pytorch image (C x W x H).
`TrajCounter`([out_key, repeats])	Global trajectory counter transform.
`UnaryTransform`(in_keys, out_keys[, ...])	Applies a unary operation on the specified inputs.
`UnsqueezeTransform`(*args, **kwargs)	Inserts a dimension of size one at the specified position.
`VC1Transform`(in_keys, out_keys, model_name)	VC1 Transform class.
`VIPRewardTransform`(*args, **kwargs)	A VIP transform to compute rewards based on embedded similarity.
`VIPTransform`(*args, **kwargs)	VIP Transform class.
`VecGymEnvTransform`([final_name, ...])	A transform for GymWrapper subclasses that handles the auto-reset in a consistent way.
`VecNorm`(*args, **kwargs)	Moving average normalization layer for torchrl environments.
`VecNormV2`(in_keys[, out_keys, lock, ...])	A class for normalizing vectorized observations and rewards in reinforcement learning environments.

gSDENoise([state_dim, action_dim, shape])	A gSDE noise initializer.

Environments with masked actions

In some environments with discrete actions, the actions available to the agent might change throughout execution. In such cases the environments will output an action mask (under the `"action_mask"` key by default). This mask needs to be used to filter out unavailable actions for that step.

If you are using a custom policy you can pass this mask to your probability distribution like so:

Categorical policy with action mask

```
>>> from tensordict.nn import TensorDictModule, ProbabilisticTensorDictModule, TensorDictSequential
>>> import torch.nn as nn
>>> from torchrl.modules import MaskedCategorical
>>> module = TensorDictModule(
>>>     nn.Linear(in_feats, out_feats),
>>>     in_keys=["observation"],
>>>     out_keys=["logits"],
>>> )
>>> dist = ProbabilisticTensorDictModule(
>>>     in_keys={"logits": "logits", "mask": "action_mask"},
>>>     out_keys=["action"],
>>>     distribution_class=MaskedCategorical,
>>> )
>>> actor = TensorDictSequential(module, dist)
```

If you want to use a default policy, you will need to wrap your environment in the `ActionMask` transform. This transform can take care of updating the action mask in the action spec in order for the default policy to always know what the latest available actions are. You can do this like so:

How to use the action mask transform

```
>>> from tensordict.nn import TensorDictModule, ProbabilisticTensorDictModule, TensorDictSequential
>>> import torch.nn as nn
>>> from torchrl.envs.transforms import TransformedEnv, ActionMask
>>> env = TransformedEnv(
>>>     your_base_env
>>>     ActionMask(action_key="action", mask_key="action_mask"),
>>> )
```

• NOTE

In case you are using a parallel environment it is important to add the transform to the parallel environment itself and not to its sub-environments.

Recorders

Recording data during environment rollout execution is crucial to keep an eye on the algorithm performance as well as reporting results after training.

TorchRL offers several tools to interact with the environment output: first and foremost, a `callback` callable can be passed to the `rollout()` method. This function will be called upon the collected tensordict at each iteration of the rollout (if some iterations have to be skipped, an internal variable should be added to keep track of the call count within `callback`).

To save collected tensordicts on disk, the `TensorDictRecorder` can be used.

Recording videos

Several backends offer the possibility of recording rendered images from the environment. If the pixels are already part of the environment output (e.g. Atari or other game simulators), a `VideoRecorder` can be appended to the environment. This environment transform takes as input a logger capable of recording videos (e.g. `CSVLogger`, `WandbLogger` or `TensorBoardLogger`) as well as a tag indicating where the video should be saved. For instance, to save mp4 videos on disk, one can use `CSVLogger` with a *video_format="mp4"* argument.

The `VideoRecorder` transform can handle batched images and automatically detects numpy or PyTorch formatted images (WHC or CWH).

```
>>> logger = CSVLogger("dummy-exp", video_format="mp4")
>>> env = GymEnv("ALE/Pong-v5")
>>> env = env.append_transform(VideoRecorder(logger, tag="rendered", in_keys=["pixels"]))
>>> env.rollout(10)
>>> env.transform.dump()  # Save the video and clear cache
```

Note that the cache of the transform will keep on growing until dump is called. It is the user responsibility to take care of calling *dump* when needed to avoid OOM issues.

In some cases, creating a testing environment where images can be collected is tedious or expensive, or simply impossible (some libraries only allow one environment instance per workspace). In these cases, assuming that a *render* method is available in the environment, the `PixelRenderTransform` can be used to call *render* on the parent environment and save the images in the rollout data stream. This class works over single and batched environments alike:

```
>>> from torchrl.envs import GymEnv, check_env_specs, ParallelEnv, EnvCreator
>>> from torchrl.record.loggers import CSVLogger
>>> from torchrl.record.recorder import PixelRenderTransform, VideoRecorder
>>>
>>> def make_env():
>>>     env = GymEnv("CartPole-v1", render_mode="rgb_array")
>>>     # Uncomment this line to execute per-env
>>>     # env = env.append_transform(PixelRenderTransform())
>>>     return env
>>>
>>> if __name__ == "__main__":
...     logger = CSVLogger("dummy", video_format="mp4")
...
...     env = ParallelEnv(16, EnvCreator(make_env))
...     env.start()
...     # Comment this line to execute per-env
...     env = env.append_transform(PixelRenderTransform())
...
...     env = env.append_transform(VideoRecorder(logger=logger, tag="pixels_record"))
...     env.rollout(3)
...
...     check_env_specs(env)
...
...     r = env.rollout(30)
...     env.transform.dump()
...     env.close()
```

Recorders are transforms that register data as they come in, for logging purposes.

`TensorDictRecorder`(out_file_base[, ...])	TensorDict recorder.
`VideoRecorder`(logger, tag[, in_keys, skip, ...])	Video Recorder transform.
`PixelRenderTransform`([out_keys, preproc, ...])	A transform to call render on the parent environment and register the pixel observation in the tensordict.

Helpers

`RandomPolicy`(action_spec[, action_key])	A random policy for data collectors.
`check_env_specs`(env[, return_contiguous, ...])	Tests an environment specs against the results of short rollout.
`exploration_type`()	Returns the current sampling type.
`get_available_libraries`()	Returns all the supported libraries.
`make_composite_from_td`(data, *[, ...])	Creates a Composite instance from a tensordict, assuming all values are unbounded.
`set_exploration_type`	alias of `set_interaction_type`
`step_mdp`(tensordict[, next_tensordict, ...])	Creates a new tensordict that reflects a step in time of the input tensordict.
`terminated_or_truncated`(data[, ...])	Reads the done / terminated / truncated keys within a tensordict, and writes a new tensor where the values of both signals are aggregated.

Domain-specific

`ModelBasedEnvBase`(*args, **kwargs)	Basic environment for Model Based RL sota-implementations.
`model_based.dreamer.DreamerEnv`(*args, **kwargs)	Dreamer simulation environment.
`model_based.dreamer.DreamerDecoder`([...])	A transform to record the decoded observations in Dreamer.

Libraries

TorchRL's mission is to make the training of control and decision algorithm as easy as it gets, irrespective of the simulator being used (if any). Multiple wrappers are available for DMControl, Habitat, Jumanji and, naturally, for Gym.

This last library has a special status in the RL community as being the mostly used framework for coding simulators. Its successful API has been foundational and inspired many other frameworks, among which TorchRL. However, Gym has gone through multiple design changes and it is sometimes hard to accommodate these as an external adoption library: users usually have their "preferred" version of the library. Moreover, gym is now being maintained by another group under the "gymnasium" name, which does not facilitate code compatibility. In practice, we must consider that users may have a version of gym *and* gymnasium installed in the same virtual environment, and we must allow both to work concomittantly. Fortunately, TorchRL provides a solution for this problem: a special decorator `set_gym_backend` allows to control which library will be used in the relevant functions:

```
>>> from torchrl.envs.libs.gym import GymEnv, set_gym_backend, gym_backend
>>> import gymnasium, gym
>>> with set_gym_backend(gymnasium):
...     print(gym_backend())
...     env1 = GymEnv("Pendulum-v1")
<module 'gymnasium' from '/path/to/venv/python3.9/site-packages/gymnasium/__init__.py'>
>>> with set_gym_backend(gym):
...     print(gym_backend())
...     env2 = GymEnv("Pendulum-v1")
<module 'gym' from '/path/to/venv/python3.9/site-packages/gym/__init__.py'>
>>> print(env1._env.env.env)
<gymnasium.envs.classic_control.pendulum.PendulumEnv at 0x15147e190>
>>> print(env2._env.env.env)
<gym.envs.classic_control.pendulum.PendulumEnv at 0x1629916a0>
```

We can see that the two libraries modify the value returned by `gym_backend()` which can be further used to indicate which library needs to be used for the current computation. `set_gym_backend` is also a decorator: we can use it to tell to a specific function what gym backend needs to be used during its execution. The `torchrl.envs.libs.gym.gym_backend()` function allows you to gather the current gym backend or any of its modules:

```
>>> import mo_gymnasium
>>> with set_gym_backend("gym"):
...     wrappers = gym_backend('wrappers')
...     print(wrappers)
<module 'gym.wrappers' from '/path/to/venv/python3.9/site-packages/gym/wrappers/__init__.py'>
>>> with set_gym_backend("gymnasium"):
...     wrappers = gym_backend('wrappers')
...     print(wrappers)
<module 'gymnasium.wrappers' from '/path/to/venv/python3.9/site-packages/gymnasium/wrappers/__init__.py'>
```

Another tool that comes in handy with gym and other external dependencies is the `torchrl._utils.implement_for` class. Decorating a function with `@implement_for` will tell torchrl that, depending on the version indicated, a specific behavior is to be expected. This allows us to easily support multiple versions of gym without requiring any effort from the user side. For example, considering that our virtual environment has the v0.26.2 installed, the following function will return `1` when queried:

```
>>> from torchrl._utils import implement_for
>>> @implement_for("gym", None, "0.26.0")
... def fun():
...     return 0
>>> @implement_for("gym", "0.26.0", None)
... def fun():
...     return 1
>>> fun()
1
```

`BraxEnv`(*args, **kwargs)	Google Brax environment wrapper built with the environment name.
`BraxWrapper`(*args, **kwargs)	Google Brax environment wrapper.
`DMControlEnv`(*args, **kwargs)	DeepMind Control lab environment wrapper.
`DMControlWrapper`(*args, **kwargs)	DeepMind Control lab environment wrapper.
`GymEnv`(*args, **kwargs)	OpenAI Gym environment wrapper constructed by environment ID directly.
`GymWrapper`(*args, **kwargs)	OpenAI Gym environment wrapper.
`HabitatEnv`(*args, **kwargs)	A wrapper for habitat envs.
`IsaacGymEnv`(*args, **kwargs)	A TorchRL Env interface for IsaacGym environments.
`IsaacGymWrapper`(*args, **kwargs)	Wrapper for IsaacGymEnvs environments.
`IsaacLabWrapper`(*args, **kwargs)	A wrapper for IsaacLab environments.
`JumanjiEnv`(*args, **kwargs)	Jumanji environment wrapper built with the environment name.
`JumanjiWrapper`(*args, **kwargs)	Jumanji's environment wrapper.

`MeltingpotEnv`(*args, **kwargs)	Meltingpot environment wrapper.
`MeltingpotWrapper`(*args, **kwargs)	Meltingpot environment wrapper.
`MOGymEnv`(*args, **kwargs)	FARAMA MO-Gymnasium environment wrapper.
`MOGymWrapper`(*args, **kwargs)	FARAMA MO-Gymnasium environment wrapper.
`MultiThreadedEnv`(*args, **kwargs)	Multithreaded execution of environments based on EnvPool.
`MultiThreadedEnvWrapper`(*args, **kwargs)	Wrapper for envpool-based multithreaded environments.
`OpenMLEnv`(*args, **kwargs)	An environment interface to OpenML data to be used in bandits contexts.
`OpenSpielWrapper`(*args, **kwargs)	Google DeepMind OpenSpiel environment wrapper.
`OpenSpielEnv`(*args, **kwargs)	Google DeepMind OpenSpiel environment wrapper built with the game string.
`PettingZooEnv`(*args, **kwargs)	PettingZoo Environment.
`PettingZooWrapper`(*args, **kwargs)	PettingZoo environment wrapper.
`RoboHiveEnv`(*args, **kwargs)	A wrapper for RoboHive gym environments.
`SMACv2Env`(*args, **kwargs)	SMACv2 (StarCraft Multi-Agent Challenge v2) environment wrapper.
`SMACv2Wrapper`(*args, **kwargs)	SMACv2 (StarCraft Multi-Agent Challenge v2) environment wrapper.
`UnityMLAgentsEnv`(*args, **kwargs)	Unity ML-Agents environment wrapper.
`UnityMLAgentsWrapper`(*args, **kwargs)	Unity ML-Agents environment wrapper.
`VmasEnv`(*args, **kwargs)	Vmas environment wrapper.
`VmasWrapper`(*args, **kwargs)	Vmas environment wrapper.
`gym_backend`([submodule])	Returns the gym backend, or a sumbodule of it.
`set_gym_backend`(backend)	Sets the gym-backend to a certain value.
`register_gym_spec_conversion`(spec_type)	Decorator to register a conversion function for a specific spec type.

Previous Next