TorchRL: A data-driven decision-making library for PyTorch

Albert Bou
Universitat Pompeu Fabra
albert.bou@upf.edu

Matteo Bettini
University of Cambridge
mb2389@cl.cam.ac.uk

Sebastian Dittert
Universitat Pompeu Fabra
sebastian.dittert@upf.edu

Vikash Kumar
Meta AI

Shagun Sodhani
Meta AI

Xiaomeng Yang
Meta AI
yangxm@meta.com

Gianni De Fabritiis
ICREA, Universitat Pompeu Fabra, Acellera
g.defabritiis@gmail.com

Vincent Moens
PyTorch Team, Meta
vincentmoens@gmail.com

Abstract

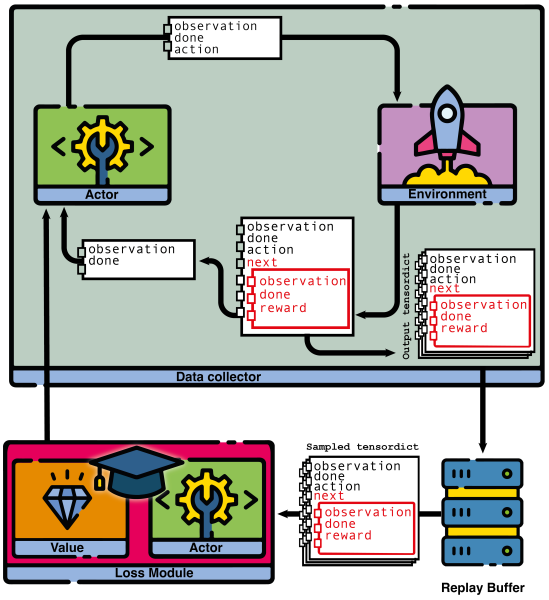
Striking a balance between integration and modularity is crucial for a machine learning library to be versatile and user-friendly, especially in handling decision and control tasks that involve large development teams and complex, real-world data, and environments. To address this issue, we propose TorchRL, a generalistic control library for PyTorch that provides well-integrated, yet standalone components. With a versatile and robust primitive design, TorchRL facilitates streamlined algorithm development across the many branches of Reinforcement Learning (RL) and control. We introduce a new PyTorch primitive, TensorDict, as a flexible data carrier that empowers the integration of the library's components while preserving their modularity. Hence replay buffers, datasets, distributed data collectors, environments, transforms and objectives can be effortlessly used in isolation or combined. We provide a detailed description of the building blocks, supporting code examples and an extensive overview of the library across domains and tasks. Finally, we show comparative benchmarks to demonstrate its computational efficiency. TorchRL fosters long-term support and is publicly available on GitHub for greater reproducibility and collaboration within the research community. The code is opensourced on GitHub.

1 Introduction

The advancements in hardware supporting machine learning (ML) training and deployment pipelines, along with the simultaneous release of robust ML libraries, provide a potent combination that can drive artificial intelligence (AI) forward. Originally, breakthroughs in AI were powered by custom code tailored for specific machines and backends. However, with the widespread adoption of AI accelerators like GPUs and TPUs, as well as user-friendly development frameworks such as PyTorch Paszke et al. [2019], TensorFlow Abadi et al. [2015], and Jax Bradbury et al. [2018], we have moved beyond this point.

Deploying these technologies in the field of decision-making has been more challenging than in other AI domains, such as computer vision or natural language processing, where a few libraries have gained widespread recognition within their respective research communities. We attribute the

Preprint. Under review.




```python
# Environment
env = GymEnv("Pendulum-v1")
# Model: Actor and value
mlp_actor = MLP(num_cells=64, depth=3,
    in_features=3, out_features=1)
actor = TensorDictModule(mlp_actor,
    in_keys=["observation"], out_keys=["action"])
mlp_value = MLP(num_cells=64, depth=2,
    in_features=4, out_features=1)
critic = TensorDictSequential(actor,
    TensorDictModule(mlp_value,
    in_keys=["observation", "action"],
    out_keys=["state_action_value"]))
# Data Collector
collector = SyncDataCollector(
    env, AdditiveGaussianWrapper(actor),
    frames_per_batch=1000, total_frames=1000000)
# Replay Buffer
buffer = TensorDictReplayBuffer(
    storage=LazyTensorStorage(100000))
# Loss Module
loss_fn = DDPGLoss(actor, critic)
optim = torch.optim.Adam(
    loss_fn.parameters(), lr=2e-4)
# Train
for data in collector:  # collect data
    buffer.extend(data)  # populate RB
    sample = buffer.sample(50)  # mini-batch
    loss = loss_fn(sample)  # loss
    loss = loss["loss_actor"] + loss["loss_value"]
    loss.backward()
    optim.step()
    optim.zero_grad()
```

Figure 1: TorchRL overview. The left side showcases the key components of the library, demonstrating the data flow with TensorDict instances passing between modules. On the right side, a code snippet is provided as a toy example, illustrating the training of DDPG. The script provides users with full control over the algorithm's hyperparameters, offering a concise yet comprehensive solution. Still, replacing a minimal number of components in the script enables a seamless transition to another similar algorithm, like SAC or REDQ.

slow progress to the dynamic requirements of decision-making algorithms, which creates a trade-off between modularity and component integration. The current solutions in this field are inadequate for accommodating the variety of applications, ranging from gaming to fine-tuning generative models with human feedback Christiano et al. [2017a], due to being either too high-level and difficult to repurpose across use cases or too low-level for non-experts to use practically. Because of this intricate landscape, a consensus adoption of frameworks within the research community remains limited Brockman et al. [2016]. Nevertheless, we believe that moving forward, the next generation of libraries for decision-making will need to be easy to use and rely on widespread frameworks while still being powerful, efficient, scalable, and broadly applicable, to meet the demands of this rapidly evolving and heterogeneous field. This is the problem that TorchRL addresses.

The ecosystem of decision-making algorithms is complex, starting with RL solutions for games where super-human performance is becoming the rule (Mnih et al. [2013], Silver et al. [2016, 2017, 2018], Vinyals et al. [2019], OpenAI et al. [2019a], Espeholt et al. [2018]). Because it is generalistic and oblivious to the modality, RL has subsequently shown to be effective at solving multiple tasks such as robotic control Tobin et al. [2017], Peng et al. [2018], Rudin et al. [2022], OpenAI et al. [2019b], autonomous driving Aradi [2022], Sallab et al. [2017], finance Ganesh et al. [2019], Hambly et al. [2021], Mosavi et al. [2020], bidding in online advertisement Jin et al. [2018], Cai et al. [2017], Zou et al. [2019], Zhu and Roy [2021], cooling system control Luo et al. [2022], faster matrix multiplication algorithms Fawzi et al. [2022], chip design Mirhoseini et al. [2021] and drug synthesis Popova et al. [2018], among many others. Within the ML community, RL is sometimes used to help train other models through AutoML solutions He et al. [2018], Zoph and Le [2017], Zoph et al. [2018], Baker et al. [2017], Zhong et al. [2018]. More recently, it has emerged as the backbone of large language model fine-tuning for conversational AI Christiano et al. [2017b], von Werra et al. [2023], Nakano et al. [2021] which is having a worldwide and disruptive impact. The available toolbox is extensive, encompassing techniques such as sim-to-real transfer Tobin et al. [2017], Peng et al. [2018], Rudin et al. [2022], OpenAI et al. [2019b], model-based RL Brunnbauer et al. [2022], Wu et al. [2022], Hansen et al. [2022a,b] or offline data collection for batched RL and imitation learning Kalashnikov et al. [2018], Shah et al. [2022]. In addition, algorithms and training pipelines must often meet challenging scaling requirements, such as distributed inference Mnih et al. [2016], Espeholt et al. [2018] or distributed model execution Nakano et al. [2021]. They must also satisfy

imperative deployment demands when used on devices like robots or autonomous cars, which often have limited computational resources and need to operate in real time.

Training decision-making models involves a sequential and algorithmic process, where the cooperation between framework components is vital for adaptability across different applications. Actors, collectors or replay buffers need to be able to handle various modalities. Imposing a fixed input/output format would restrict their reusability, while eliminating all restrictions would create significant difficulties in designing adaptable scripts. Consequently, conventional decision-making libraries must choose between a well-integrated framework with a high-level API or a flexible framework with basic building blocks of limited scope. The former hinders quick coding and testing of new ideas, while the latter leads to complex code and a poor user experience. TorchRL elegantly tackles each of these challenges in a comprehensive and flexible manner.

In this paper, we will explain our approach to overcoming these challenges by focusing primarily on the problem of module portability, communication and code repurposing. We introduce TensorDict, a new PyTorch data carrier, which is central to achieving this goal. By adopting this new coding paradigm, we demonstrate how simple it is to reuse ML scripts and primitives across different tasks or research domains. Building upon this innovative technical foundation, we explore various independent components that rely on this abstraction. These components encompass a range of advanced solutions, including distributed data collectors, cutting-edge models like Decision Transformers, and highly specialized classes such as an environment API compatible with single and multi-agent problems. We demonstrate that the current state of the library encompasses diverse categories of decision-making algorithms, caters to a wide range of applications, performs optimally on both small-scale machines and large clusters, and demands minimal technical knowledge and effort to get started.

Related work. Currently, there is a vast array of libraries available for reinforcement learning. Many of these libraries offer a limited selection of implemented algorithms, such as Dopamine Castro et al. [2018], ChainerRL Fujita et al. [2021], PyTorchRL Bou et al. [2022], TFAgents Guadarrama et al. [2018], RLkit and autonomous-learning-library Nota [2020]. Other libraries are specifically designed for particular niches, such as Pyqlearning, Openspiel Lanctot et al. [2019] and Facebook/ELF Tian et al. [2017]. There are also libraries that only support a small number of environments, like OpenSpiel and reaver Ring [2018], or that lack the capability for parallel environments with distributed learning, such as MushroomRL D'Eramo et al. [2021]. Additionally, some previously popular libraries have fallen out of favor and are no longer actively maintained, including Tensorforce Kuhnle et al. [2017], KerasRL Plappert [2016], rlpyt Stooke and Abbeel [2019], ReAgent Gauci et al. [2018] and IntelLabs/coach Caspi et al. [2017].

Currently, popular solutions often prioritize high-level approaches that favor component integration. This is particularly necessary for libraries that provide non-standalone components or that rely on limited data carriers, as they require additional code to ensure seamless integration between components. This is the case of Stable-baselines (and Stable-baselines3, SB3) Raffin et al. [2021], ElegantRL Liu et al. [2021], Acme Hoffman et al. [2020], DI-engine engine Contributors [2021], garage garage contributors [2019] or RLlib Liang et al. [2018]. Some of these libraries are compatible with more than one deep learning framework and even allow for distributed training. Others focus on letting researchers compare against pre-existing baselines. However, even the ones that allow for composability, still advocate for high-level implementations to manage the learning processes, creating, training and evaluating agents with single function calls that handle the instantiation and coordination of agent components "under the hood". Another popular resource with some similarities to TorchRL is Tianshou Weng et al. [2022], which offers a large coverage of RL algorithms, high engineering quality standards, a generic data carrier and modular components. However, it does not offer standalone components that can be used independently from the rest of the library. Finally, CleanRLHuang et al. [2022] offers Torch and JAX efficient single-file implementations of RL algorithms. This design choice simplifies code comprehension, enhancing its understandability. However, a notable drawback of this approach is the lack of component reusability, which poses a significant challenge to scaling any project without extensive code duplication.

In Appendix H, we detail how TorchRL design choices contrast with other existing libraries, with a focus on their modularity and communication protocols which are the defining features of our framework.

2 TorchRL's data carrier

TorchRL is built on the fundamental concept of self-contained independent components. The resulting atomicity and modularity facilitate versatile usage of individual components within most machine learning workflows. In Fig. 1 we outline TorchRL's general organization. In this section, we highlight the importance of a data carrier that enables seamless communication between these components. To address this need, we introduce our solution, the *TensorDict* class, which serves as an intuitive and versatile means of data exchange. Additionally, we provide an overview of the key components in the library, which can be used independently or as foundational elements for constructing decision-making algorithms, as depicted in Fig. 1.

Inter-class communication in decision-making workflows.

Seamless class-to-class communication requires a shared communication protocol. Some existing RL libraries trade-off integration over modularity by providing low-level functionals that require an extra amount of boilerplate code to be integrated into a codebase. Others give highly integrated solutions that contain all the necessary elements to train a specific algorithm under a single trainer class. These integrated solutions, often presented as "agents", are prevalent across RL frameworks, making it easy to reproduce results. However, there are several drawbacks to these multi-responsibility classes, including complicated constructors, non-obvious unit-testing workflows, and challenges in modifying specific aspects of the class for a particular use case.

TensorDict. We address this problem by adopting a new class-to-class communication protocol that is both transparent and generic. This common language takes the form of a new data carrier for PyTorch named TensorDict. TensorDict is packaged as an opensource lightweight library that is nearly the only core dependency of TorchRL (together with PyTorch and NumPy). This library enables every component to be developed independently of the requirements of the classes it potentially communicates with. As a result, data collectors can be designed without any knowledge of the policy structure, and replay buffers do not need any information about the data structure that is to be stored within them. Fig. 1 gives an overview of a practical data flow using TensorDict as a communication tool.

```
data = TensorDict({
    "observation": torch.ones(3, 4), # a tensor at the root level
    "next": {"observation": torch.ones(3, 4)}, # a nested TensorDict
}, batch_size=[3])
```

Figure 2: A typical TensorDict instantiation.

A TensorDict is a dictionary-like object that stores tensors(-like) objects. It includes additional features that optimize its use with PyTorch. One of its notable features is the ability to handle batch size, which means that it can be indexed not only by keys but also by shapes. As is the case for a `torch.Tensor`, TensorDict requires a shape to be provided. By design, this shape is supposed to represent the batch dimensions (in contrast with the "feature" dimension). In Fig. 2, we create a TensorDict instance with the shape of `torch.Size([3])`. The data created there can be indexed in two ways: along its shape or along its key dimension:

```
a = data["observation"]
b = data["next"]
```

```
c = data["next", "observation"]
d = data[0]
```

The `a` object retrieves the `"observation"` entry (a tensor), while `b` is the `"next"` nested TensorDict. `c` is the the `"observation"` entry of the `"next"` nested TensorDict (another tensor). Finally, `d` is a new TensorDict that contains the first element of the batch for all entries. Its key structure will match the one of `data` but its shape will be empty.

Features and Functionalities. TensorDict provides a whole stack of extra functionality that naturally follows from these two basic features. Here follows a non-exhaustive list of feature set for a hypothetical `x` tensordict:

```
out = []
obs = env.reset()
hid = None
for _ in range(T):
 act, hid, logp = actor(obs,hidden)
 obs_, rw, done, *_ = env.step(act)
 out += [(obs, act, obs_, rw, done, hid)]
 obs = obs_
```

```
out = TensorDict({},
 batch_size=[T])
data = env.reset()
for i in range(T):
 data = actor(data)
 data = env.step(data)
 out[i] = data
 data = step_mdp(data)
```

Figure 3: Comparison between a rollout implementation without (left) and with (right) TensorDict.

```
module = TensorDictSequential(
 TensorDictModule(lambda x: x + 1, in_keys=["x"], out_keys=["y"]),
 TensorDictModule(nn.Linear(3, 4), in_keys=["y"], out_keys=["z"]),
)
module(data) # updates data in-place
```

Figure 4: Programmable module design with TensorDict.

Shape-based functions:

- `x[idx]` / `x.gather`
- `x.reshape` / `x.view`
- `x.permute`
- `x.(un)squeeze`
- `x.unbind` / `x.split`
- `torch.stack(data_list)`
- `torch.cat(data_list)`

Key-based functions:

- `x.keys` / `x.values`
- `x.items`
- `x.(un)flatten`
- `del x[key]`
- `x.select`
- `x.exclude`
- `x.rename_key_`

Other:

- `x.apply(fn)`
- `x.memmap_` / `x.to_h5`
- `x.share_memory_()`
- `x.to(device)`
- `x.clone()`
- `x.(i)send` / `x.(i)recv`
- `x.lock_`

Additionally, TensorDict provides some functionality for distributed point-to-point communication and an efficient storage interface (through memory-mapped arrays or H5 files).

TensorDict enables the design of functions and classes with a generic signature, where we can safely restrict the input and output to be both TensorDict instances. This allows us to code generic rollout functions that are oblivious to the operations executed by the actor or the environment. In Fig. 3, a recurrent policy is applied in a gym-like environment, demonstrates how TensorDict can simplify a data collection loop.

A code like this is more generic than its more verbose counterpart in that it can be recycled whether or not the actor returns a log-probability and/or a recurrent state. Memory management is also better handled: an empty `TensorDict` container is created beforehand and populated during the rollout. As soon as the first example is assigned to it, the tensors are pre-allocated in memory given the initial batch size provided. This avoids expensive stacking operations or tedious custom tensor preallocations. Notably, the data is returned in a similar format as the one it is presented during the collection, avoiding tedious stacking operations after the collection. Using a similar syntax, TorchRL proposes more readable, shorter and more flexible collection utilities.

The `tensordict` library also comes with a dedicated `tensordict.nn` package that rethinks the way one designs PyTorch models. Various primitives, such as `TensorDictModule` and `TensorDictSequential` allow to design of complex PyTorch operations in an explicit and programmable way. In Fig. 8, we designed a sequential module that wraps a lambda function and an `torch.nn.Module`. These primitives are fully compatible with `torch.compile` through a dedicated symbolic tracer available in the TensorDict package, making TorchRL itself compatible with the latest features of PyTorch 2.0.

3 TorchRL components

Environment API, wrappers, and transforms. OpenAI Gym Brockman et al. [2016] has become the most widely adopted environment interface in RL: its standardization of the environment API was a major leap forward that enabled research reproducibility. Its simplicity (only two functions are exposed during interaction) drove its success, but it presents several drawbacks, such as a fixed tuple signature for its `step` method or its heavy reliance on wrappers (see below).

The TorchRL environment interface aims to maintain the simplicity of Gym while addressing these drawbacks. As it is the case with Gym, only the `reset()` and `step()` methods are required to interact with these objects. However, the usage of TensorDict as a signature for the environment methods greatly facilitates their integration, as discussed in Sec. 2 and presented Fig. 1. TensorDicts enable carrying multidimensional tensor-like data, allowing batched/vectorized simulation Freeman et al. [2021], Bettini et al. [2022] and multidimensional input/output spaces. This makes TorchRL compatible by default with multi-agent and multi-task applications, as the TensorDict dimensionality carries the agent or task identity.

Consequently, TorchRL is by no means an extension of Gym or any other simulation library, unlike other libraries which cover one specific simulator but not others. The generic environment API allows to easily support a multitude of existing simulators: Gym and Gymnasium Brockman et al. [2016] since v0.13, DMControl Tunyasuvunakool et al. [2020], Habitat Szot et al. [2021], RoboHive Rob [2020], OpenML datasets Vanschoren et al. [2013], D4RL datasets Fu et al. [2020], Brax Freeman et al. [2021], Jumanji Bonnet et al., and VMAS Bettini et al. [2022] are some examples, but the list is constantly growing. Notably, the last three of these are vectorized and can pass gradients through the simulations allowing to compute reparameterized trajectories. We provide some more technical information about the environment API and its usage in Appendix B.

Parametric policies often struggle to interpret native environment outputs directly, and data transformations are often required. In the realm of RL, these transforms are traditionally implemented as environment wrappers. However, this approach can be likened to Russian wooden dolls, as accessing the original environment or modifying the transform sequence requires intricate maneuvers within the wrappers. To address this problem, TorchRL draws inspiration from other components of the PyTorch ecosystem maintainers and contributors [2016], Paszke et al. [2019] which rely on the concept of transform sequences to adapt, manipulate and shape module outputs.

Significantly, TorchRL's transforms offer more than just environmental adaptation, as they can be applied wherever data transformation is required. These versatile transforms can be utilized in various components, including replay buffers, collectors, and even ported from environment to models, effectively connecting the training process with real-world applications. Examples include data transformations (e.g. resizing, cropping), target computation (reward-to-go) or even embedding through foundational models Nair et al. [2022], Ma et al. [2022]. As we show in the Appendix, this feature comes at no cost in term of functionality and bring a new, simple way of dynamically building and modifying environment and model transforms.

Data collectors. TorchRL has dedicated classes for data collection that execute a policy in one or multiple parallel environments and return batches of transition data in TensorDict format. Data collectors iteratively compute the actions to be executed, pass those actions to the environments (real or simulated), and can handle resetting when and where required. These collectors are designed for ease of use, requiring only an environment constructor, a policy module, a batch size, and a target number of steps. However, developers can also specify the asynchronous or synchronous nature of data collection, the number of parallel environments, the resource allocation (e.g., GPU, number of workers) and the data postprocessing, giving them fine control and flexibility over the collection process.

To tackle complex RL problems, vast amounts of data are often requiredEspeholt et al. [2018], Wijmans et al. [2019], Horgan et al. [2018], Kapturowski et al. [2018]. TorchRL offers distributed components that enable data collection at scale by coordinating multiple workers in a cluster. These components are compatible with various backends like gloo or NCCL through torch.distributed, and multiple launchers and resource management solutions including submititIncubator [2021] and RayMoritz et al. [2018]. As scaling inevitably adds complexity, the distributed solutions in TorchRL are designed as independent components that provide the same interface and data control as non-distributed components. This enables practitioners to work locally on projects with complete

independence while also providing an easy way to scale up to distributed projects with little extra effort. By replacing non-distributed components with their distributed counterparts, practitioners can increase data collection throughput and easily scale their projects. Crucially, TorchRL distributed dependencies are optional and not required for non-distributed components.

Replay buffers and datasets. Replay buffers (RBs) are crucial elements of RL that enable agents to learn from past experiences by storing, processing, and resampling data. However, creating a flexible RB class that caters to various use cases without duplicating implementations can be challenging. To overcome this, TorchRL provides a single RB implementation that offers complete composability. Users can define distinct components for data storage, batch generation, pre-storage and post-sampling transforms, allowing independent customization of each aspect.

The RB class has a default constructor that creates a buffer with a generic configuration, utilizing a versatile list storage and a sampler that generates batches uniformly. However, users can also specify the storage to be contiguous in either virtual or physical memory. This format provides faster performance and can handle data sizes up to terabytes. These features allow the creation of buffers that can be populated on-the-fly during training or used with static datasets for offline RL. TorchRL offers a range of downloadable datasets for this purpose (D4RL Fu et al. [2020] or OpenML Vanschoren et al. [2013]), with many more to come. Alternative samplers are also available, including a sampler without replacement that ensures all data is presented once before repeating, or a C++ optimized prioritized sampler.

RBs can store any Python object in their native form using the default constructor. Nevertheless, we encourage presenting the data in a TensorDict format, which streamlines the workflow (see Fig. 1) and benefits from TorchRL's optimized storage, sampling and transform techniques.

TorchRL also integrates a remote replay buffer component to gather data from distributed workers. This functionality is detailed in Appendix C.

Modules and models. In RL, the model architectures are distinct not only from those in other machine learning disciplines but also within RL itself. This creates two interconnected issues: Firstly, there is considerable architectural variability to contend with. Secondly, the nature of inputs and outputs for these networks can fluctuate based on the specific environment and algorithm in use. Thus, both challenges necessitate a flexible and adaptable approach.

TorchRL responds to the first problem by providing RL-dedicated neural network architectures, which are organically integrated into PyTorch as native `nn.Modules`. These are available at varying levels of abstraction, ranging from foundational building blocks like Multilayer Perceptron (MLP), Convolutional Neural Networks (CNN), and Transformer, to comprehensive, high-level structures such as `ActorCriticOperator` or `WorldModelWrapper` but also exploration modules like `NoisyLinear` or Planners like `CEMPlanner`.

To tackle the second challenge, TorchRL uses specialized `TensorDictModule` and `TensorDictSequential` primitives subclassed from the TensorDict library (see Fig. 8). `TensorDictModule` wraps PyTorch modules, transforming them into tensordict-compatible objects for effortless integration into the TorchRL framework. Concurrently, `TensorDictSequential` concatenates TensorDictModules, functioning similarly to `nn.Sequential`. Importantly, these TensorDict modules maintain full compatibility with torch 2.0, torch.compile, and functorch. Vectorized maps (vmap) are also well-supported, enabling the execution of multiple value networks in a vectorized manner. All instances of TensorDictModule are simultaneously functional and stateful, permitting parameters to be optionally inputted without additional transformations. This feature notably facilitates the design of meta-RL algorithms within the TorchRL library.

Objectives and value estimators. In TorchRL, objective classes are stateful components that track trainable parameters from `torch.nn` models and handle loss computation, a crucial step for model optimization in any machine learning algorithm.

These components share a common signature defined by two main methods: a constructor method that accepts the models and all relevant hyperparameters, and a forward computation method that takes a TensorDict of collected data samples as input and returns another TensorDict with one or more loss terms. It is worth noting that, in principle, loss modules can accommodate any `nn.Module` and process inputs that are not presented as TensorDict as it is shown in Appendix D.

This design choice makes objective classes very versatile components, abstracting the implementation of a vast array of loss functions derived from various families of RL algorithms using a single template (Fig. 1). These include on-policy algorithms (e.g. Advantage Actor-Critic (A2C) Mnih et al. [2016], Proximal Policy Optimization (PPO) Schulman et al. [2017]), off-policy algorithms (e.g. Deep Q-Network (DQN) Mnih et al. [2013], distributional DQN Bellemare et al. [2017] and subsequent improvements of these Hessel et al. [2017], Deep Deterministic Policy Gradient (DDPG) Lillicrap et al. [2015], Twin Delayed Deep Deterministic Policy Gradient (TD3) Fujimoto et al. [2018a], Soft Actor-Critic (SAC) Haarnoja et al. [2018], Randomized Ensembled Double Q-learning (REDQ) Chen et al. [2021a]), offline algorithms (e.g., Implicit Q-Learning Kostrikov et al. [2021], Decision Transformer Chen et al. [2021b]), and model-based algorithms (e.g., Dreamer Hafner et al. [2019]).

Finally, TorchRL also provides a solution for state value estimations, a crucial step in the loss computation of many RL algorithms that can take various forms. As it is the case for the modules, the value estimators are presented both in a functional, explicit way as well as an encapsulated TensorDictModule class that facilities their integration in TorchRL-based scripts. These replaceable objects, named `ValueEstimators`, can be called outside the objective class or dynamically during the objective forward call, providing a versatile solution to accommodate different forms of loss computation.

Trainers. The modularity of TorchRL's components allows developers to write training scripts with explicit loops for data collection and neural network optimization in the traditional structure adopted by most ML libraries. While this enables practitioners to keep tight control over the training process, it might hinder TorchRL's adoption by first-time users looking for one-fits-all solutions. For this reason, we introduce a `Trainer` classes that abstract this further complexity. By exposing a simple `train()` method, trainers take care of running data collection and optimization while providing several hooks (i.e., callbacks) at different stages of the process. These hooks allow users to customize various aspects of data processing, logging, and other training operations. Trainers also encourage reproducibility by providing a checkpointing interface that allows to abruptly interrupt and restart training at any given time while saving models and RB of any given size. An example of the `Trainer` class usage is provided in Appendix E.

4 Ecosystem

To flourish a thriving community around the library, in addition to open-sourcing the library, we are committing to designing a comprehensive ecosystem to support our users. The ecosystem comprises documentation, datasets, issues-tracking, model library, sample use cases, etc. Below we detail a few of these modules and refer the readers to our webpage for further information.

Table 1: GAE efficiency. The data consisted of a batch of 1000 trajectories of 1000 time steps, with a "done" frequency of 0.001, randomly spread across the data.

Library	Speed (ms)	Standalone	Meta-RL	Adjacent[$]	Backend	Device
Tianshou[†]	5.15	-	-	+	namba	CPU
SB3[†]	14.1	-	-	+	numpy	CPU
RLLib (Ray)[*] [†]	9.38	-	-	-	scipy	CPU
CleanRL	1.43	-	-	+	Jax	GPU
TorchRL-compiled	2.67	+	+	+	PyTorch	GPU
TorchRL-vec	**1.33**	+	+	+	PyTorch	GPU

[$] Using Tianshou's implementation requires transforming tensors to numpy arrays and then transforming the result back to tensors. This overehead is not accounted for here.

[*] A proper usage of Ray's GAE implementation would have needed a split of the adjacent trajectories, which we did not do to focus on the implementation efficiency.

[†] Numpy or similar backends require moving data from and to GPU which can substantially impact performance. Those data moves are unaccounted for in this table but can double the GAE runtime.

Documentation, knowledge-base, engineering. TorchRL has rich documentation, and each public class and function must have a proper set of docstrings before being released. At the time of writing, the coverage is above 90%, and tested across Linux, MacOs and Windows platforms. All the optional dependencies (such as simulation libraries for which environment wrappers are available) are being tested in dedicated workflows. We guarantee compatibility with all versions of the gym starting from v0.13, including the latest transition to Gymnasium. Due to various refactorings and backward-compatibility breaking changes within the Gym API, we have introduced the `implement_for` decorator, which enables writing multiple versions of a same function. This decorator allows TorchRL to dynamically select the appropriate version based on the current state of the virtual environment and the decorator arguments. Regarding the transition from the Gym to the Gymnasium backend, we have introduced a `set_gym_backend` context manager. This context manager allows both libraries to be utilized concurrently within the same environment without the need for specialized wrappers.

Several benchmarks have been put in place to ensure that our modules keep a high-efficiency standard. Most components including TensorDict, the loss modules, the advantage functions and the collectors have been optimized to the bone. Nightly releases of both libraries are available.

Applications. As mentioned earlier, TorchRL covers multiple areas of RL and decision-making including, but not limited to, on-policy, off-policy, offline, model-based and distributional RL. Notably, TorchRL is currently used by some research groups on hardware. An area of interest is *Multi-Agent RL* (MARL), which considers decision-making under uncertainty for multiple agents. Most of TorchRL's components are default-compatible with MARL. TensorDicts allow to carry both per-agent data (e.g., reward in POMGs Littman [1994]) and shared data (e.g., global state, reward in Dec-POMDPs Bernstein et al. [2002]) by storing the tensors at the relevant nesting level, thus highlighting their semantic difference and optimizing storage. In the case of agents with heterogeneous input/output domains, we leverage `LazyStackedTensorDict`, a class that provides a TensorDict-like interface to abstract over lists of tensors, to uniform the data-carrying process. As for single-agent solutions, MARL losses can easily be swapped and are semantically similar to their single-agent counterparts. To bootstrap the adoption of TorchRL for multi-agent use cases, we provide implementations for six state-of-the-art MARL algorithms, and benchmark them on three multi-robot coordination scenarios in the VMAS Bettini et al. [2022] simulator. Results are presented in Appendix G.

5 Experiments

Compute efficiency. To check that our solution does not come at the cost of reduced throughput, we test two selected functionalities (advantage computation and data collection) against other popular libraries. These experiments were run on an AWS cluster with a single node (96 CPU cores and 8 A100 GPUs). These results are reported in Tab. 1 and Tab. 2. Overall, these results show that our

solutions have a comparable if not better throughput than others and in general a greater coverage of functionalities.

Table 2: Data collection speed on the `"PongNoFrameskip-v4"` environment. In each case, a random policy was used.

Library	Speed (fps)	Standalone collector	Environment Compatibility
Tianshou[†]	7272	+	Gym
SB3[‡]	1070	-	Gym
RLLib[†]	1167	-	Any
CleanRL[‡]	3670	+	Gym
TorchRL (sync) [$†]	2991	+	Any
TorchRL (sync-preempted) [$†]	3733	+	Any
TorchRL (async) [$†]	**8845**	+	Any

[$] Async collection means that the collector delivers the data on a first-ready-first-served basis. In the sync scenario, the collector waits for each worker to be ready before delivering the data. Our collectors also have a preemption mechanism to temper the effect of slower workers. In the corresponding example, a preemption threshold of 0.7 was used.

[†] We used 32 parallel processes, returning a total of 320 steps per batch. Depending on the (a)sync nature of the collection, this quantity was gathered from a single process or combined from all.

[‡] Use vectorized environments from Gym or a version inherited from them.

Algorithm evaluation. TorchRL equips its users with an extensive selection of example scripts that demonstrate cutting-edge algorithms for both online and offline RL. In order to ascertain the completeness and effectiveness of TorchRLs framework components, we provide documented validations of a variety of algorithms in Appendix F and G.

6 Conclusion

We have presented TorchRL, a modular decision-making library with composable, single-responsibility building pieces that rely on TensorDict for an efficient class-to-class communication protocol. TorchRL's design principles are tailored for the dynamic field of decision making, enabling the implementation of a wide range of algorithms. Furthermore, these design principles support the seamless integration of future innovations as new TensorDict compatible objects. We believe TorchRL can be of particular interest to researchers and developers alike. Its key advantage is that researchers can create or extend existing classes on their own and seamlessly integrate them with existing scripts without the need to modify the internal code of the library, allowing for quick prototyping while maintaining a stable and reliable code base. We believe this feature will also be suited for developers looking to apply an AI solution to a specific problem, even if unaccounted for by the library: unlike the Agent classes commonly found in other libraries, TorchRL's design gives users fine-grained control over every aspect of training which we believe to be necessary to push decision-making research and applications to the next frontiers. We demonstrate that TorchRL not only benefits from a wide range of functionalities but also exhibits computational efficiency. The library is currently fully functional, and we are extensively testing it across various scenarios and applications. We look forward to receiving feedback and aim to provide support and further enhance the library based on the input we receive.

References

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/`

9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep Reinforcement Learning from Human Preferences. In I Guyon, U Von Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, and R Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017a. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/d5e2c0adad503c91f91df240d0cd4e49-Paper.pdf.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017. ISSN 1476-4687. doi: 10.1038/nature24270. URL https://doi.org/10.1038/nature24270.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404. URL https://www.science.org/doi/abs/10.1126/science.aar6404.

Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz,

Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning. 2019a. URL https://arxiv.org/abs/1912.06680.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world, 2017.

Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2018. doi: 10.1109/icra.2018.8460528. URL https://doi.org/10.1109%2Ficra.2018.8460528.

Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning, 2022.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand, 2019b.

Szilárd Aradi. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(2):740–759, 2022. doi: 10.1109/TITS.2020.3024655.

Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 29(19):70–76, jan 2017. doi: 10.2352/issn.2470-1173.2017.19.avm-023. URL https://doi.org/10.2352%2Fissn.2470-1173.2017.19.avm-023.

Sumitra Ganesh, Nelson Vadori, Mengda Xu, Hua Zheng, Prashant Reddy, and Manuela Veloso. Reinforcement learning for market making in a multi-agent dealer market, 2019.

Ben M. Hambly, Renyuan Xu, and Huining Yang. Recent advances in reinforcement learning in finance. *SSRN Electronic Journal*, 2021. doi: 10.2139/ssrn.3971071. URL https://doi.org/10.2139/ssrn.3971071.

Amirhosein Mosavi, Yaser Faghan, Pedram Ghamisi, Puhong Duan, Sina Faizollahzadeh Ardabili, Ely Salwana, and Shahab S. Band. Comprehensive review of deep reinforcement learning methods and applications in economics. *Mathematics*, 8(10):1640, September 2020. doi: 10.3390/math8101640. URL https://doi.org/10.3390/math8101640.

Junqi Jin, Chengru Song, Han Li, Kun Gai, Jun Wang, and Weinan Zhang. Real-time bidding with multi-agent reinforcement learning in display advertising. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM, oct 2018. doi: 10.1145/3269206.3272021. URL https://doi.org/10.1145%2F3269206.3272021.

Han Cai, Kan Ren, Weinan Zhang, Kleanthis Malialis, Jun Wang, Yong Yu, and Defeng Guo. Real-time bidding by reinforcement learning in display advertising. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, February 2017. doi: 10.1145/3018661.3018702. URL https://doi.org/10.1145/3018661.3018702.

Lixin Zou, Long Xia, Zhuoye Ding, Jiaxing Song, Weidong Liu, and Dawei Yin. Reinforcement learning to optimize long-term user engagement in recommender systems, 2019.

Zheqing Zhu and Benjamin Van Roy. Deep exploration for recommendation systems, 2021.

Jerry Luo, Cosmin Paduraru, Octavian Voicu, Yuri Chervonyi, Scott Munns, Jerry Li, Crystal Qian, Praneet Dutta, Jared Quincy Davis, Ningjia Wu, Xingwei Yang, Chu-Ming Chang, Ted Li, Rob Rose, Mingyan Fan, Hootan Nakhost, Tingtiansho Liu, Brian Kirkman, Frank Altamura, Lee Ctianshoe, Patrick Tonker, Joel Gouker, Dave Uden, Warren Buddy Bryan, Jason Law, Deeni Fatiha, Neil Satra, Juliet Rothenberg, Mandeep Waraich, Molly Cartiansho, Satish Tallapaka, Sims Witherspoon, David Parish, Peter Dolan, Chenyu Zhao, and Daniel J. Mankowitz. Controlling commercial cooling systems using reinforcement learning, 2022.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022. ISSN 1476-4687. doi: 10.1038/s41586-022-05172-4. URL https://doi.org/10.1038/s41586-022-05172-4.

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. A graph placement methodology for fast chip design. *Nature*, 594 (7862):207–212, 2021. ISSN 1476-4687. doi: 10.1038/s41586-021-03544-w. URL https://doi.org/10.1038/s41586-021-03544-w.

Mariya Popova, Olexandr Isayev, and Alexander Tropsha. Deep reinforcement learning for de novo drug design. *Science Advances*, 4(7), jul 2018. doi: 10.1126/sciadv.aap7885. URL https://doi.org/10.1126%2Fsciadv.aap7885.

Yihui He, Ji tianshou, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition, 2018.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning, 2017.

Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-tianshou Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2423–2432, 2018.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017b.

Leandro von Werra, Jonathan Tow, reciprocated, Shahbuland Matiana, Alex Havrilla, cat state, Louis Castricato, Alan, Duy V. Phung, Ayush Thakur, Alexey Bukhtiyarov, aaronrmm, Fabrizio Milo, Daniel, Daniel King, Dong Shin, Ethan Kim, Justin Wei, Manuel Romero, Nicky Pochinkov, Omar Sanseviero, Reshinth Adithyan, Sherman Siu, Thomas Simonini, Vladimir Blagojevic, Xu Song, Zack Witten, alexandremuzio, and crumb. CarperAI/trlx: v0.6.0: LLaMa (Alpaca), Benchmark Util, T5 ILQL, Tests, March 2023. URL https://doi.org/10.5281/zenodo.7790115.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback. *CoRR*, abs/2112.09332, 2021. URL https://arxiv.org/abs/2112.09332.

Axel Brunnbauer, Luigi Berducci, Andreas Brandstätter, Mathias Lechner, Ramin Hasani, Daniela Rus, and Radu Grosu. Latent imagination facilitates zero-shot transfer in autonomous racing, 2022.

Philipp Wu, Alejandro Escontrela, Danijar Hafner, Ken Goldberg, and Pieter Abbeel. Daydreamer: World models for physical robot learning, 2022.

Nicklas Hansen, Yixin Lin, Hao Su, Xiaolong Wang, Vikash Kumar, and Aravind Rajeswaran. Modem: Accelerating visual model-based reinforcement learning with demonstrations, 2022a.

Nicklas Hansen, Xiaolong Wang, and Hao Su. Temporal difference learning for model predictive control, 2022b.

Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation, 2018.

Dhruv Shah, Arjun Bhorkar, Hrish Leen, Ilya Kostrikov, Nick Rhinehart, and Sergey Levine. Offtiane reinforcement learning for visual navigation, 2022.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A Research Framework for Deep Reinforcement Learning. 2018. URL `http://arxiv.org/abs/1812.06110`.

Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. Chainerrl: A deep reinforcement learning library. *Journal of Machine Learning Research*, 22(77):1–14, 2021. URL `http://jmlr.org/papers/v22/20-376.html`.

Albert Bou, Sebastian Dittert, and Gianni De Fabritiis. Efficient reinforcement learning experimentation in pytorch, 2022. URL `https://openreview.net/forum?id=9WJ-fT_92Hp`.

Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018. URL `https://github.com/tensorflow/agents`. [Online; accessed 25-June-2019].

Chris Nota. The autonomous learning library. `https://github.com/cpnota/autonomous-learning-library`, 2020.

Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019. URL `http://arxiv.org/abs/1908.09453`.

Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

Roman Ring. Reaver: Modular deep reinforcement learning framework. `https://github.com/inoryy/reaver`, 2018.

Carlo D'Eramo, Davide Tateo, Andrea Bonarini, Marcello Restelli, and Jan Peters. Mushroomrl: Simplifying reinforcement learning research. *Journal of Machine Learning Research*, 22(131): 1–5, 2021. URL `http://jmlr.org/papers/v22/18-056.html`.

Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. Web page, 2017. URL `https://github.com/tensorforce/tensorforce`.

Matthias Plappert. keras-rl. `https://github.com/keras-rl/keras-rl`, 2016.

Adam Stooke and Pieter Abbeel. rlpyt: A research code base for deep reinforcement learning in pytorch. *CoRR*, abs/1909.01500, 2019. URL `http://arxiv.org/abs/1909.01500`.

Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Zhengxing Chen, Yuchen He, Zachary Kaden, Vivek Narayanan, and Xiaohui Ye. Horizon: Facebook's open source applied reinforcement learning platform. *arXiv preprint arXiv:1811.00260*, 2018.

Itai Caspi, Gal Leibovich, Gal Novik, and Shadi Endrawis. Reinforcement learning coach, December 2017. URL https://doi.org/10.5281/zenodo.1134899.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

Xiao-Yang Liu, Zechu Li, Ming Zhu, Zhaoran Wang, and Jiahao Zheng. ElegantRL: Massively parallel framework for cloud-native deep reinforcement learning. https://github.com/AI4Finance-Foundation/ElegantRL, 2021.

Matthew W. Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, Léonard Hussenot, Robert Dadashi, Gabriel Dulac-Arnold, Manu Orsini, Alexis Jacq, Johan Ferret, Nino Vieillard, Seyed Kamyar Seyed Ghasemipour, Sertan Girgin, Olivier Pietquin, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Abe Friesen, Ruba Haroun, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020. URL https://arxiv.org/abs/2006.00979.

DI engine Contributors. DI-engine: OpenDILab decision intelligence engine. https://github.com/opendilab/DI-engine, 2021.

The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. https://github.com/rlworkgroup/garage, 2019.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, 23(267):1–6, 2022. URL http://jmlr.org/papers/v23/21-1127.html.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL http://jmlr.org/papers/v23/21-1342.html.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL http://github.com/google/brax.

Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. Vmas: A vectorized multi-agent simulator for collective robot learning. *The 16th International Symposium on Distributed Autonomous Robotic Systems*, 2022.

Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. dm_control: Software and tasks for continuous control. *Software Impacts*, 6:100022, 2020. ISSN 2665-9638. doi: https://doi.org/10.1016/j.simpa.2020.100022. URL https://www.sciencedirect.com/science/article/pii/S2665963820300099.

Andrew Szot, Alex Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah Maestre, Mustafa Mukadam, Devendra Chaplot, Oleksandr Maksymets, Aaron Gokaslan, Vladimir Vondrus, Sameer Dharur, Franziska Meier, Wojciech Galuba, Angel Chang, Zsolt Kira, Vladlen Koltun, Jitendra Malik, Manolis Savva, and Dhruv Batra. Habitat 2.0: Training home assistants to rearrange their habitat. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Robohive – a unified framework for robot learning. https://sites.google.com/view/robohive, 2020. URL https://sites.google.com/view/robohive.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013. doi: 10.1145/2641190.2641198. URL http://doi.acm.org/10.1145/2641190.264119.

Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning, 2020.

Clément Bonnet, Donal Byrne, Victor Le, Laurence Midgley, Daniel Luo, Cemlyn Waters, Sasha Abramowitz, Edan Toledo, Cyprien Courtot, Matthew Morris, et al. Jumanji: Industry-driven hardwareaccelerated rl environments, 2022. *URL https://github. com/instadeepai/jumanji*.

TorchVision maintainers and contributors. Torchvision: Pytorch's computer vision library. https://github.com/pytorch/vision, 2016.

Suraj Nair, Aravind Rajeswaran, Vikash Kumar, Chelsea Finn, and Abhinav Gupta. R3m: A universal visual representation for robot manipulation, 2022.

Yecheng Jason Ma, Shagun Sodhani, Dinesh Jayaraman, Osbert Bastani, Vikash Kumar, and Amy Zhang. Vip: Towards universal visual reward and representation via value-implicit pre-training. *arXiv preprint arXiv:2210.00030*, 2022.

Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. *arXiv*, pages arXiv–1911, 2019.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. 2018.

Facebook Incubator. Submitit. https://github.com/facebookincubator/submitit, 2021. [GitHub repository].

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017. URL http://arxiv.org/abs/1707.06887.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018a.

T. Haarnoja, Aurick Zhou, Kristian Hartikainen, G. Tucker, Sehoon Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications. *ArXiv*, abs/1812.05905, 2018.

Xinyue Chen, Che Wang, Zijian Zhou, and Keith Ross. Randomized ensembled double q-learning: Learning fast without a model. *arXiv preprint arXiv:2101.05982*, 2021a.

Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline reinforcement learning with implicit q-learning. *arXiv preprint arXiv:2110.06169*, 2021.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021b.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4): 819–840, 2002.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018b.

Qinqing Zheng, Amy Zhang, and Aditya Grover. Online decision transformer, 2022.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The starcraft multi-agent challenge. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2186–2188, 2019.

Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zając, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, et al. Google research football: A novel reinforcement learning environment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4501–4510, 2020.

Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

Christian Schroeder de Witt, Tarun Gupta, Denys Makoviichuk, Viktor Makoviychuk, Philip HS Torr, Mingfei Sun, and Shimon Whiteson. Is independent learning all you need in the starcraft multi-agent challenge? *arXiv preprint arXiv:2011.09533*, 2020.

Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35:24611–24624, 2022.

Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2085–2087, 2018.

Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *The Journal of Machine Learning Research*, 21(1):7234–7284, 2020.

A TensorDict functionality

TensorDict's aim is to abstract away functional details of pytorch worflows, enabling scripts to be more easily reused across different tasks, architectures, or implementation details. While primarily a tool for convenience, TensorDict can also provide notable computational benefits for specific operations.

A.1 TensorDictBase and subclasses

The parent class of any TensorDict class is `TensorDictBase`. This abstract class lists the common operations of all its subclasses, and also implements a few of them. Indeed, some generic methods are implemented once for all subclasses, such as `tensordict.apply`, as the output will always be a `TensorDict` instance. Others, such as `__getitem__` have a behaviour that is intrinsically linked to their specifics.

The following subclasses are available:

- `TensorDict`: this is the most common class and usually the only one users will need to interact with.
- `LazyStackedTensorDict`: This class is the result of calling `torch.stack` on a list of `TensorDictBase` subclass instances. The resulting object stores each instance independently and stacks the items only when queried through `__getitem__`, `__setitem__`, `get`, `set` or similar. Heterogeneous TensorDict instances can be stacked together: in that case, `get` may fail to stack the tensors (if one is missing from one tensordict instance or if the shapes do not match). However, this class can still be used to carry data from object to object or worker to worker, even if the data is heterogeneous. The original tensordicts can be recovered simply via indexing of the lazy stack: `torch.stack(list_of_tds, 0)[0]` will return the first element of `list_of_tds`.
- `PersitentTensorDict`: implements a TensorDict class with a persistent storage. At time of writing, only the H5 backend is implemented. This interface is currently being used to integrate massive datasets within TorchRL for offline RL and imitation learning.
- `_CustomOpTensorDict`: This abstract class is the parent of other classes that implement lazy operations. This is used to temporarily interact with a TensorDict on which a zero-copy reshape operation is to be executed. Any in-place operations on these instances will affect the parent tensordict as well. For instance, `tensordict.permute(0, 1).set("a", tensor)` will set a new key in the parent `tensordict` as well.

Performance Through a convenient and intuitive API, tensordict offers some out-of-the-box optimizations that would otherwise be cumbersome to achieve and, we believe, can have a positive impact beyond RL usage. For instance, TensorDict makes it easy to store large datasets in contiguous physical memory through a PyTorch interface with numpy's memory mapped arrays named `MemmapTensor`, available in the tensordict library. Unlike regular PyTorch datasets, TensorDict offers the possibility to index multiple items at a time and make them immediately available in memory in a contiguous manner: by reducing the I/O overhead associated with reading single files (which we preprocess and store in a memory-mapped array on disk), one can read, cast to GPU and preprocess multiple files all-at-once. Not only can we amortize the time of reading independent files, but we can also leverage this to apply transforms on batches of data instead of one item at a time. These batched transforms on device are much faster to execute than their multiprocessed counterparts. Figure 5 shows the collection speed over ImageNet with batches of 128 images and per-image random transformations including cropping and flipping. The workflow used to achieve this speedup is available in the repository documentation.

tensorclass TensorDict also offers a `@tensorclass` decorator aimed at working as the `@dataclass` decorator: it allows for the creation of specialized dataclasses with all the features of Ten-

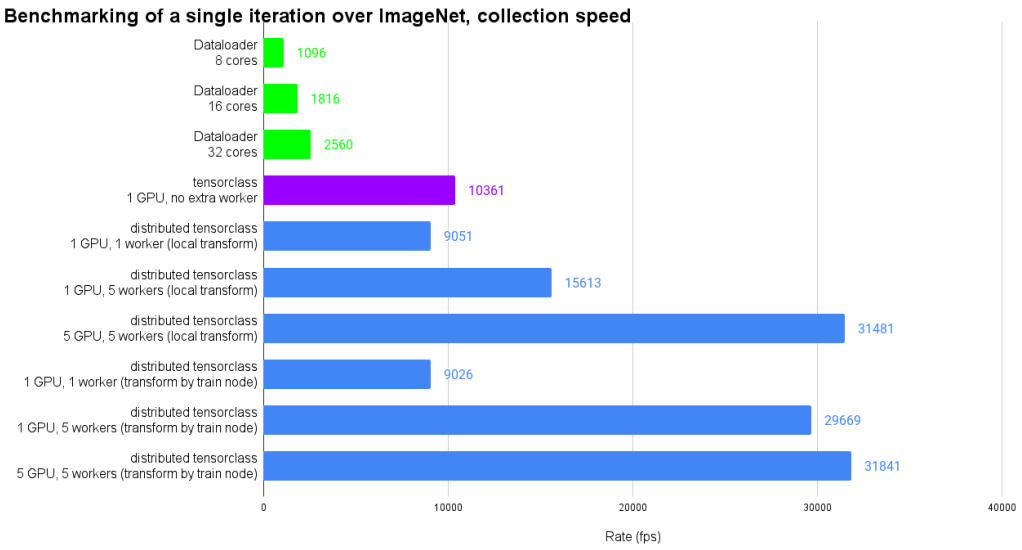


Figure 5: Dataloading speed with TensorDict.

```python
@tensorclass
class MyClass:
    image: torch.Tensor
    label: torch.Tensor

data = MyClass(
   image=torch.zeros(100, 32, 32, 3, dtype=torch.uint8),
   label=torch.randint(1000, (100,)),
   batch_size=[100]
)

# a MyData instance with floating point content
data_float = data.apply(lambda x: x.float())
# a MyData instance on cuda
data_cuda = data.to("cuda")
# the first 10 elements of the original MyData instance
data_idx = data[:10]

```

Figure 6: A `@tensorclass` example.

sorDict: shape vs key (attribute) dimension, device, tensor operations and more. `@tensorclass` instances natively support any TensorDict operation. Figure 6 shows a few examples of `@tensorclass` usage.

A.2 tensordict.nn: Modules and functional API

The second pane of the tensordict library is its module interface, on which TorchRL heavily relies. `tensordict.nn` aims to address two core limitations of `torch.nn`. First, `torch.nn` offers a `Module` class which encourages users to create new components through inheritance. However, in many cases, this level of control in module construction is unnecessary and can even hinder modularity. For example, consider the coding of a simple multi-layered perceptron, as depicted in figure 7, which is commonly encountered.

Such definition of a dedicated class lacks flexibility. One solution commonly adopted to control the network hyperparameters (number of layers, the activation function or layer width) is to pass them as inputs to the constructor and group the layers together within a `nn.Sequential` instance. This

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(3, 4)
        self.layer2 = nn.Linear(4, 2)
        self.activation = torch.relu

    def forward(self, x):
        y = self.activation(self.linear1(x))
        return self.linear2(y)
```

Figure 7: Programmable module design with TensorDictModule

```
from tensordict.nn import TensorDictSequential as Seq,
                                    TensorDictModule as Mod
module = Seq(
   Mod(nn.Linear(3, 4), in_keys=['input'], out_keys=['hidden0']),
   Mod(torch.relu, in_keys=['hidden0'], out_keys=['hidden1']),
   Mod(nn.Linear(4, 2), in_keys=['hidden1'], out_keys=['output']),
)

```

Figure 8: Programmable module design with TensorDictModule

solution works fine as long as the sequence is a simple chain of operations on a single tensor. As soon as some operations need to be skipped, or when modules output multiple tensors, `nn.Sequential` is not suited anymore and the operations must be explicitly written within the `forward` method, which sets the computational graph once and for all. The `TensorDictModule` class overcomes this limitation and provides a fully programmable module design API, as shown on figure 8.

Formulating a module in this way brings all the flexibility we need to address the shortcomings exposed above: it is easy to programmatically design a new model architecture at runtime for any kind of model. Selecting subgraphs is easy and can be achieved via regular indexing of the module or more advanced sub-graph selection. In the following example, a call to `select_subsequence` will output a new `TensorDictSequential` where the modules are restricted only to those whose operations depend on the `"hidden1"` key:

```
sub_module = module.select_subsequence(in_keys=["hidden1"])
```

To further back our claim that such model construction is beneficial both in terms of clarity and modularity, let us take the example of writing a residual connection with tensordict.nn:

```
block = Seq(
  Mod(nn.Linear(128, 128), in_keys=['x'], out_keys=['intermediate']),
  Mod(torch.relu, in_keys=['intermediate'], out_keys=['intermediate'])
                                    ,
  Mod(nn.Linear(128, 128), in_keys=['intermediate'], out_keys=['
                                    intermediate']),
)
residual = Seq(
  block,
  Mod(lambda x, z: x+z, in_keys=['x', 'intermediate'], out_keys=['x'])
)
```

For clarity, we have separated the the backbone and the residual connection, but these could be grouped under the same `TensorDictSequential` instance. Nevertheless, `select_subsequence` will go through these various levels of nesting if ever called.

Functional usage Functional usage of these modules is a cornerstone in TorchRL as it underpins efficient calls to the same module with different sets of parameters for off-policy algorithms where

multiple critics are executed simultaneously or in situations where target parameters are needed. `tensordict.nn` provides a `make_functional` function that transforms any `torch.nn` module in a functional module that accepts an optional `params` (keyword-)argument. This function will return the parameters and buffers in a `TensorDict` instance whose nested structure mimics the one of the originating module (unlike `Module.state_dict()` which returns them in a flat structure).

Using these functionalized modules is straightforward. For instance, we can zero all the parameters of a module and call it on some data:

```
params = make_functional(module)
params_zero = params.clone().zero_()
module(tensordict, params=params_zero)
```

This allows us to group the calls to multiple critics in a vectorized fashion. In the following example, we stack the parameters of two critic networks and call the module with this stack using `torch.vmap`, the PyTorch vectorized-map function:

```
make_functional(critic)
vmap_critic = vmap(critic, (None, 0))
data = vmap_critic(
  data,
  torch.stack([params_critic0, params_critic1], 0))
```

We refer to the `torch.vmap` documentation for further information on this feature. Similarly, calling the same module with trainable or target parameters can be done via a functional call to it.

Performance The read and write operations executed during the unfolding of a `TensorDictSequential` can potentially impact the execution speed of the underlying model. To address this concern, we offer a dedicated `symbolic_trace` function that simplifies the operation graph to its essential elements. This function generates a module that is fully compatible with `torch.compile`, resulting in faster execution. By combining these solutions, we can achieve module performance that is on par, if not better, than their regular `nn.Module` counterparts.

Using TensorDict modules in tensordict-free code bases Because the library's goal is to be as minimally restrictive as possible, we also provide the possibility of executing TensorDict-based modules without explicitly requiring any interaction with TensorDict: the `tensordict.nn.dispatch` decorator allows to interact with any module from the TensorDict ecosystem with pure tensors:

```
module = TensorDictModule(
  lambda x: x+1, x-2,
  in_keys=["x"], out_keys=["y", "z"])
y, z = module(x = torch.randn(3))
```

In section D, we show how this is used to generalize loss modules beyond TorchRL's ecosystem.

`TensorDictModule` and associated classes also provide a `select_out_keys` method that allows to hide some specific keys of a graph to minimize the output. When use in conjunction with `@dispatch`, this enables a much clearer usage when multiple intermediate keys are present.

B Environment API

To bind input/output data to a specific domain, TorchRL uses `TensorSpec`, a custom version of the feature spaces found in other environment libraries. This class is specially tailored for PyTorch: its instances can be moved from device to device, reshaped or stacked at will, as the tensors would be. TensorDict is naturally blended within `TensorSpec` through a `CompositeSpec` primitive which is a metadata version of TensorDict. The following example shows the equivalence between these two classes:

```
>>> from torchrl.data import UnboundedContinuousTensorSpec, \ ...
    CompositeSpec
>>> spec = CompositeSpec(
```

```
...     obs=UnboundedContinuousTensorSpec(shape=(3, 4)),
...     shape=(3,),
...     device='cpu'
)
>>> print(spec.rand())
TensorDict(
    fields={
        obs: Tensor(shape=torch.Size([3, 4]), device=cpu, dtype=torch.
                                        float32, is_shared=False)},
    batch_size=torch.Size([3]),
    device=cpu,
    is_shared=False)
```

`TensorSpec` comes with multiple dedicated methods to run common checks and operations, such as `spec.is_in` which checks if a tensor belongs to the spec's domain, `spec.project` which maps a tensor onto its L1 closest datapoint within a spec's domain or `spec.encode` which creates a tensor from a numpy array. Other dedicated operations such as conversions from categorical encoding to one-hot are also supported.

Specs also support indexing and stacking, and as for TensorDict, heterogeneous `CompositeSpec` instances can lazyly be stacked together for a better integration in multitask or multiagent frameworks.

Using these specs greatly empowers the library and is a key element of many of the performance optimization in TorchRL. For instance, using the TensorSpecs allows us to preallocate buffers in shared memory or on GPU to maximise the throughput during parallel data collection, without executing the environment operations a single time. For real-world deployment of TorchRL's solutions, these tools are essential as they allow us to run algorithms on fake data defined by the environment specs, checking that there is no device or shape mismatch between model and environment, without requiring a single data collection on hardware.

Designing environments in TorchRL To encode a new environment, developers only need to implement the internal `_reset()` and `_step()` methods, and provide the input and output domains through the specs for grounding the data. TorchRL provides a set of safety checks grouped under the `torchrl.envs.utils.check_env_specs` which should be run before executing and environment for the first time. This optional validation step offloads many checks that would otherwise be executed at runtime to a single initial function call, thereby reducing the footprint of these checks.

C Replay Buffers

TorchRL's replay buffers are fully composable: although they come with "batteries included", requiring a minimal effort to be built, they also support many customizations such as storage type, sampling strategy or data transforms. We provide dedicated tutorials and documentation on their usage in the library. The main features can be listed as follows:

- Various **storage** types are proposed. We provide interfaces with regular lists, or contiguous physical or virtual memory storages through `MemmapTensor` and `torch.Tensor` classes respectively.
- At time of writing, the **samplers** include a generic circular sampler, a sampler without repetition and an efficient, C++ based prioritized sampler.
- One can also pass **transforms** which are notably identical to those used in conjunction with TorchRL's environments. This makes it easy to recycle a data transform pipeline used during collection to one used offline to train from a static dataset made of untransformed data. It also allows us to store data more efficiently: as an example, consider the cost of saving images in `uint8` format against saving transformed images in floating point numbers. Using the former reduces the memory footprint of the buffer, allowing to store more data and access it faster. Because the data is stored contiguously and the transforms are applied on-device, the reduction in memory footprint comes at little extra computational cost. One could also consider the usage of transforms when working with Decision Transformers Chen et al. [2021b], which are typically trained with a different lookback window on the collected data than the one used during inference. Duplicating the same transform in the environment and in the buffer with a different set of parameters facilitates the implementation of these techniques.

```python
from torchrl.objectives import DQNLoss
from torchrl.data import OneHotDiscreteTensorSpec
from torch import nn
import torch

n_obs = 3
n_action = 4

action_spec = OneHotDiscreteTensorSpec(n_action)
value_network = nn.Linear(n_obs, n_action) # a simple value model
dqn_loss = DQNLoss(value_network, action_space=action_spec)

# define data
observation = torch.randn(n_obs)
next_observation = torch.randn(n_obs)
action = action_spec.rand()
next_reward = torch.randn(1)
next_done = torch.zeros(1, dtype=torch.bool)
# get loss value
loss_val = dqn_loss(
  observation=observation,
  next_observation=next_observation,
  next_reward=next_reward,
  next_done=next_done,
  action=action)

```

Figure 9: TensorDict-free DQN Loss usage.

Importantly, this modularity of the buffer class also makes it easy to design new buffer instances, which we expect to have a positive impact for researchers. For example, a team of researchers developing a novel and more efficient sampling strategy can easily build new replay buffer instances using the proposed parent parent class while focusing on the improvement of a single of its pieces.

Distributed replay buffers Our buffer class supports Remote Procedural Control (RPC) calls through `torch.distributed.rpc`. In simple terms, this means that these buffers can be extended or accessed by distant nodes through a remote call to `buffer.extend` or `buffer.sample`. With the utilization of efficient shared memory-mapped storage when possible and multithreaded requests, the distributed replay buffer significantly enhances the transfer speed between workers, increasing it by a factor of three to ten when compared to naive implementations[1].

D Using loss modules without TensorDict

In an effort to free the library from its bounds to specific design choices, we make it possible to create certain loss modules without any interaction with TensorDict or related classes. For example, figure 9 shows how a DQN loss function can be designed without recurring to a `TensorDictModule` instance.

Although the number of loss modules with this functionality is currently limited, we are actively implementing that feature for a larger set of objectives.

E Trainer

TorchRL offers a versatile high-level Trainer class designed to manage the training process effectively with a single function call. The trainer executes a nested loop, consisting of an outer loop responsible for data collection and an inner loop that utilizes this data or retrieves data from the replay buffer to train the model. Throughout this training loop, hooks can be attached and executed at specified

[1]These benchmarks are available in the opensource repository.

Table 3: Training parameters for single-agent on-policy algorithms.

A2C		PPO	
Discount (γ)	0.99	Discount (γ)	0.99
GAE λ	0.95	GAE λ	0.95
num envs	1	num envs	1
Horizon (T)	2048	Horizon (T)	64
Adam lr	$3e^{-4}$	Adam lr	$3e^{-4}$
Minibatch size	64	Minibatch size	64
Policy architecture	MLP	Policy architecture	MLP
Value net architecture	MLP	Value net architecture	MLP
Policy layers	[64, 64]	Policy layers	[64, 64]
Value net layers	[64, 64]	Value net layers	[64, 64]
Policy activation	Tanh	Policy activation	Tanh
Value net activation	Tanh	Value net activation	Tanh
Num. epochs	10		
Clip ϵ	0.2		
Critic coef.	0.5		
Entropy coef.	0.0		

intervals to enhance flexibility and control. The Trainer class does not constitute a fundamental component; rather, it was developed as a simplified entry point to novice practitioners in the field.

In Figure 10 we provide two code listing examples to train a DDPG agent. The left approach utilizes the high-level Trainer class. On the other hand, the right listing explicitly defines the training loop, giving more control to the user over every step of the process.

F Single-agent Reinforcement Learning Experiments

To test the correctness and effectiveness of TorchRL framework components, we provide documented validations of 4 online algorithms and 3 offline algorithms for 2 MuJoCo environments: HalfCheetah-v3 and Hopper-v3.

F.1 Online Reinforcement Learning Experiments

F.1.1 Implementation details

For PPO Schulman et al. [2017], we reproduce the results from the original work on MuJoCo environments, using the same hyperparameters and network architectures. For A2C Mnih et al. [2016], we simply swap the objective component, reduce the collection horizon and compute network gradients a single time with each batch of collected data.

Ensuring a fair comparison among the off-policy algorithms DDPG Lillicrap et al. [2015], TD3Fujimoto et al. [2018b], and SACHaarnoja et al. [2018], we uniformly apply the same architecture, optimizer, learning rate, soft update parameter, and batch size as used in the official TD3 implementation. Exploration-specific parameters for DDPG and TD3 are taken directly from the according paper. To initiate the process, each algorithm begins with 10,000 steps of random actions, designed to adequately populate the replay buffer.

F.1.2 Hyperparameters

Tables 3 and 4 display all hyperparmaters values and network architecture details required to reproduce our online RL results.

```
# Environment
env = GymEnv('Pendulum-v1')
# Model: Actor and value
mlp_actor = MLP(
    num_cells=64,
    depth=3,
    in_features=3,
    out_features=1
)
actor = TensorDictModule(
    mlp_actor,
    in_keys=['observation'],
    out_keys=['action']
)
mlp_value = MLP(
    num_cells=64, depth=2,
    in_features=4,
    out_features=1
)
critic = TensorDictSequential(
    actor,
    TensorDictModule(
        mlp_actor,
        in_keys = [
            'observation',
            'action',
        ],
        out_keys =
        ['state_action_value']
    )
)
# Data Collector
collector = SyncDataCollector(
    env,
    AdditiveGaussianWrapper(
        actor
    ),
    frames_per_batch=1000,
    total_frames=1000000,
)
# Replay Buffer
buffer = TensorDictReplayBuffer(
    storage=LazyTensorStorage(
        max_size=100000,
    ),
)
# Loss Module
loss_fn = DDPGLoss(
    actor, crititc,
)
optim=torch.optim.Adam(
    loss_fn.parameters(),
    lr=2e-4,
)
# Trainer
Trainer(
    collector=collector
    total_frames=1000000,
    frame_skip=1,
    optim_steps_per_batch=1,
    loss_module=loss_fn,
    optimizer=optim,
)
trainer.train()
```

```
 # Environment
env = GymEnv('Pendulum-v1')
# Model: Actor and value
mlp_actor = MLP(
    num_cells=64,  depth=3,
    in_features=3,
    out_features=1
)
actor = TensorDictModule(
    mlp_actor,
    in_keys=['observation'],
    out_keys=['action']
)
mlp_value = MLP(
    num_cells=64,
    depth=2,
    in_features=4,
    out_features=1
)
critic = TensorDictSequential(
    actor,
    TensorDictModule(
        mlp_actor,
        in_keys = [
            'observation',
            'action',
        ],
        out_keys =
        ['state_action_value']
    )
)
# Data Collector
collector = SyncDataCollector(
    env,
    AdditiveGaussianWrapper(
        actor
    ),
    frames_per_batch=1000,
    total_frames=1000000,
)
# Replay Buffer
buffer = TensorDictReplayBuffer(
    storage=LazyTensorStorage(
        max_size=100000,
    ),
)
# Loss Module
loss_fn = DDPGLoss(
    actor, crititc,
)
optim=torch.optim.Adam(
    loss_fn.parameters(),
    lr=2e-4,
)
# Training loop
for data in collector:
    buffer.extend(data)
    sample = buffer.sample(50)
    loss = loss_fn(sample)
    loss = loss['loss_actor'] + \
           loss['loss_value']
    loss.backward()
    optim.step()
    optim.zero_grad()
```

Figure 10: Trainer class example usage (left). Fully defined training loop (right).

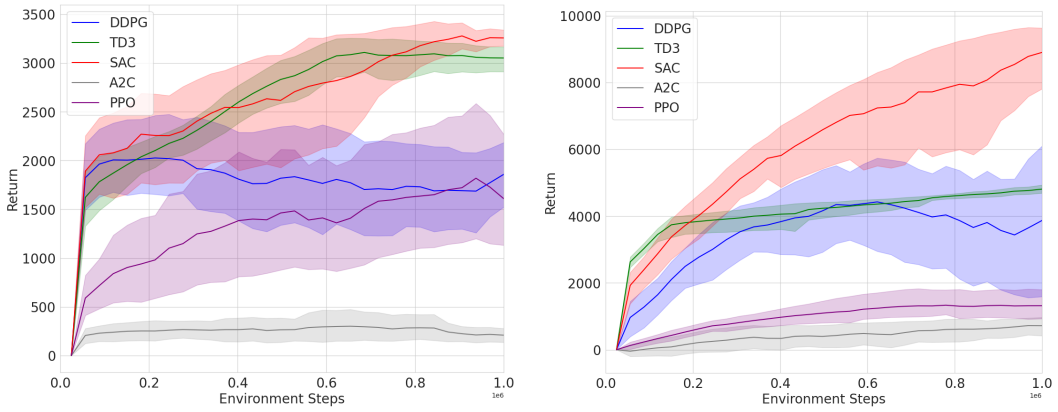


Figure 11: Online RL algorithms trained on Hopper-v3 (left) and HalfCheetah-v3 (right) MuJoCo environments. We report the mean and standard deviation reward over 3 random seeds. Each was trained for 1M frames.

Table 4: Training parameters for single-agent off-policy algorithms.

SAC		TD3		DDPG	
Discount (γ)	0.99	Discount (γ)	0.99	Discount (γ)	0.99
Adam lr (all nets)	$3e^{-4}$	Adam lr (all nets)	$3e^{-4}$	Adam lr (all nets)	$3e^{-4}$
Batch size	256	Batch size	256	Batch size	256
Policy net	MLP	Policy net	MLP	Policy net	MLP
Q net	MLP	Q net	MLP	Q net	MLP
Policy layers	[256, 256]	Policy layers	[256, 256]	Policy layers	[256, 256]
Q net layers	[256, 256]	Q net layers	[256, 256]	Q net layers	[256, 256]
Policy activation	ReLU	Policy activation	ReLU	Policy activation	ReLU
Q net activation	ReLU	Q net activation	ReLU	Q net activation	ReLU
Target polyak	0.995	Target polyak	0.995	Target polyak	0.995
Buffer size	1000000	Buffer size	1000000	Buffer size	1000000
Init rand. frames	10000	Init rand. frames	10000	Init rand. frames	10000
		Exploration noise	$N(0.0, 0.1)$	Exploration noise	OU(0.0, 0.2)
		Target noise	$N(0.0, 0.2)$	ϵ init	1.0
		Noise clip	0.5	ϵ end	0.1
		Policy delay	2	ϵ annealing steps	1000
				θ	0.15

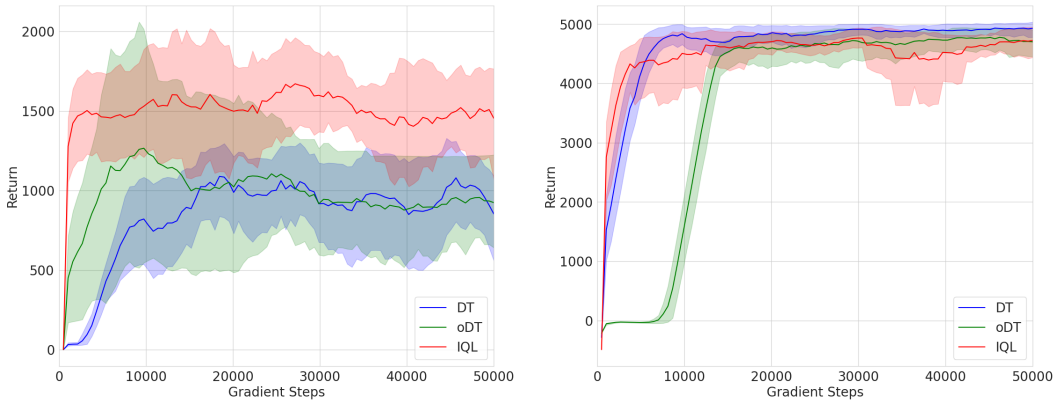


Figure 12: Offline RL algorithms pre-trained on hopper-medium-v2 (left) and halfcheetah-medium-v2 (right) D4RL datasets. We report the mean and standard deviation reward over 3 random seeds. Each run consists of 50,000 network updates.

Table 5: Training parameters for offline algorithms Decision Transformer (DT), Online Decision Transformer (oDT) and Implicit Q-Learning (IQL) for the hopper-medium-v2 (Ho) and halfcheetah-medium-v2 (HC) D4RL datasets.

DT		oDT		IQL	
Lamb lr	$1e^{-4}$	Lamb lr	$1e^{-4}$	Adam lr	$3e^{-4}$
Batch size	64	Batch size	256	Batch size	256
Weight decay	$5e^{-4}$	Weight decay	$5e^{-4}$	Policy net	MLP
Scheduler	LamdaLR	Scheduler	LamdaLR	Q net	MLP
Warmup steps	10000	Warmup steps	10000	Value net	MLP
Train context	20	Train context	20	Policy layers	[256, 256]
Eval context Ho	20	Eval context Ho	20	Q net layers	[256, 256]
Eval context HC	5	Eval context HC	5	Value net layers	[256, 256]
Policy net	GPT2	Policy net	GPT2	Policy activation	ReLU
Embd dim	128	Embd dim	512	Q net activation	ReLU
Hidden layer	3	Hidden layer	4	Value net activation	ReLU
Attn heads	1	Attn heads	4	Target polyak	0.995
Inner layer dim	512	Inner layer dim	2048	temperature (β)	3.0
Activation	ReLU	Activation	ReLU	expectile (τ)	0.7
Resid. pdrop	0.1	Resid. pdrop	0.1	Discount (γ)	0.99
Attn pdrop	0.1	Attn pdrop	0.1		
Action head	[128]	Action head	[512]		
Reward scaling	0.001	Reward scaling	0.001		
Target return Ho	3600	Target return Ho	3600		
Target return HC	6000	Target return HC	6000		
		init alpha	0.1		

F.2 Offline Reinforcement Learning Experiments

F.2.1 Implementation details

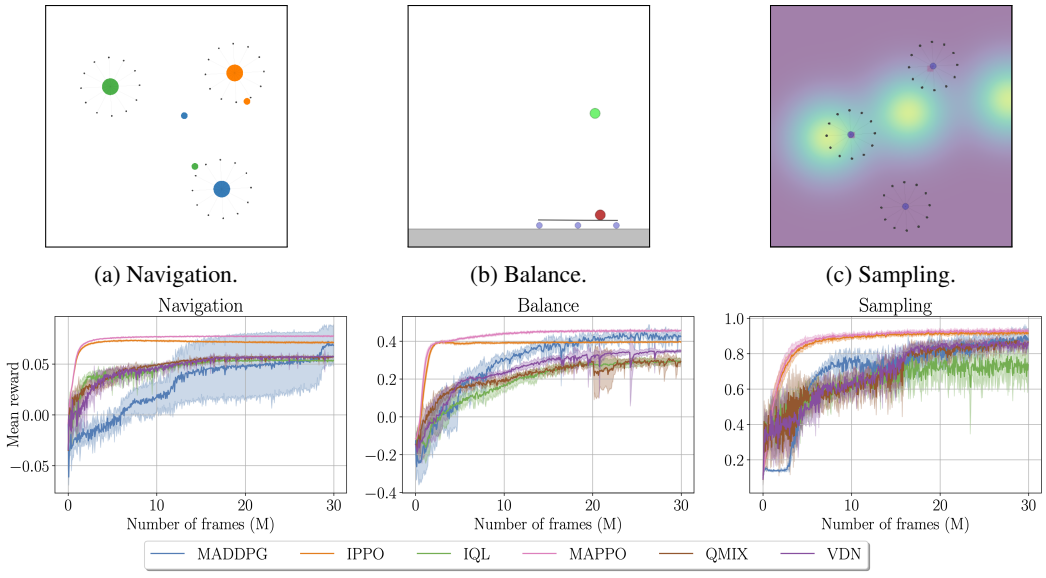
To replicate the pre-training results of both the Decision Transformer Chen et al. [2021b] and the Online Decision Transformer Zheng et al. [2022], we utilize the base GPT-2 transformer from Hugging Face Wolf et al. [2020], as presented in the official implementations. All parameters concerning architecture and training procedures are adopted directly from the specifications provided in the publications. Correspondingly, for IQL Kostrikov et al. [2021], we have selected the architecture and parameters mentioned in the paper to enable performance reproduction.

F.2.2 Hyperparameters

Table 5 displays all hyperparameters values and network architecture details required to reproduce our offline RL results.

G Multi-Agent Reinforcement Learning Experiments

To showcase TorchRL's MARL capability, we implement six state-of-the-art algorithms and benchmark them on three tasks in the VMAS simulator Bettini et al. [2022]. Unlike existing MARL benchmarks (e.g., StarCraft Samvelyan et al. [2019], Google Research Football Kurach et al. [2020]), VMAS provides vectorized on-device simulation and a set of scenarios focused on continuous and partially observable multi-robot cooperation tasks. TorchRL and VMAS both use a PyTorch backend, enabling performance gains when both sampling and training are run on-device. We implement MADDPG Lowe et al. [2017], IPPO de Witt et al. [2020], MAPPO Yu et al. [2022], IQL Tan [1993], VDN Sunehag et al. [2018], and QMIX Rashid et al. [2020]. This selection of algorithms presents many of the different flavours discussed in this section. To uniform evaluation, we perform all training on-policy with the same hyperparameters and networks. For algorithms that require discrete actions, we use the default VMAS action discretization (which transforms 2D continuous actions into 5 discrete directions). We evaluate all algorithms in three environments: (i) *Navigation* (Fig. 13a), where agents need to reach their target while avoiding collisions using a LIDAR sensor, (ii)



(a) Navigation. (b) Balance. (c) Sampling.

(d) Training curve.

Figure 13: MARL algorithm evaluation in three multi-robot control tasks. We report the mean and standard deviation reward over 6 random seeds. Each run consists of 500 iterations of $60,000$ steps each, with an episode length of 100 steps.

Balance (Fig. 13b), where agents affected by gravity have to transport a spherical package, positioned randomly on top of a line, to a given goal at the top, and (iii) *Sampling* (Fig. 13c), where agents need to cooperatively sample a probability density field. More details on the tasks, hyperparameters, and training scripts are available in the additional material. Fig. 13d shows the results. In all tasks, we can observe the effectiveness of PPO-based methods with respect to Q-learning ones, this might be due to the suboptimality of discrete actions in control tasks but also aligns with recent findings in the literature Yu et al. [2022], de Witt et al. [2020]. In the *sampling* scenario, which requires more cooperation, we see how IQL fails due to the credit assignment problem, while QMIX and VDN are able to achieve better cooperation.

Overall, our experiments highlight how different MARL solutions can be compared on diverse tasks with minimal user effort. In fact, the scripts used for this comparison present minimal differences (mainly only in the loss and policy class installations), proving TorchRL to be a flexible solution also in MARL settings.

G.1 Implementation details

We now describe the implementation details for our experiments.

G.1.1 Hyperparameters

To uniform the training process across all algorithm, we train all algorithms on-policy. The hyperparemeters used are the same for all experiments and are shown in Tab. 6. In general, the training scripts have the following structure; There is an outer loop performing sampling. At each iteration, *Batch size* frames are collected using *# VMAS vectorized envs* with *Max episode steps*. $\frac{\textit{Batch size}}{\textit{Minibatch size}}$ optimization steps are then preformed for *SDG Iterations* using an Adam optimizer. The training ends after *# training iterations*. Critics and actors (when used) are two layer MLPs with 256 cells and tanh activation. Parameters are shared in all algorithms apart from MADDPG to follow the original paper implementation.

Table 6: MARL training parameters.

Training		General		PPO	
Batch size	60000	Discount γ	0.9	Clip ϵ	0.2
Minibatch size	4096	Max episode steps	100	GAE λ	0.9
SDG Iterations	45	NN Type	MLP	Entropy coeff	0
# VMAS vectorized envs	600	# of layers	2	KL coeff	0
Learning rate	5e-5	Layer size	256		
Max grad norm	40	Activation	Tanh		
# training iterations	500	# agents	3		

G.1.2 Environments

The tasks considered are scenarios taken from the VMAS Bettini et al. [2022] simulator. They all consider agents in a 2D continuous workspace. To move, agents take continuous 2D actions which represent control forces. Discrete action can be set and will map to the five options: up, down, left, right, and staying still. In the following, we describe in detail the three environments used in the experiment:

- *Navigation* (Fig. 13a): Randomly spawned agents (circles with surrounding dots) need to navigate to randomly spawned goals (smaller circles). Agents need to use LIDARs (dots around them) to avoid running into each other. For each agent, we compute the difference in the relative distance to its goal over two consecutive timesteps. The mean of these values over all agents composes the shared reward, incentivizing agents to move towards their goals. Each agent observes its position, velocity, lidar readings, and relative position to its goal.
- *Balance* (Fig. 13b) Agents (blue circles) are spawned uniformly spaced out under a line upon which lies a spherical package (red circle). The team and the line are spawned at a random x position at the bottom of the environment. The environment has vertical gravity. The relative x position of the package on the line is random. In the top half of the environment a goal (green circle) is spawned. The agents have to carry the package to the goal. Each agent receives the same reward which is proportional to the distance variation between the package and the goal over two consecutive timesteps. The team receives a negative reward of -10 for making the package or the line fall to the floor. The observations for each agent are: its position, velocity, relative position to the package, relative position to the line, relative position between package and goal, package velocity, line velocity, line angular velocity, and line rotation $\mathrm{mod}\pi$. The environment is done either when the package or the line fall or when the package touches the goal.
- *Sampling* (Fig. 13c) Agents are spawned randomly in a workspace with an underlying gaussian density function composed of three gaussian modes. Agents need to collect samples by moving in this field. The field is discretized to a grid (with agent-sized cells) and once an agent visits a cell its sample is collected without replacement and given as reward to the whole team. Agents can use a lidar to sense each other in order to coordinate exploration. Apart from lidar, position and velocity observations, each agent observes the values of samples in the 3x3 grid around it.

H Comparison of design decisions

As a conclusion note, we provide some more insight on the differences between TorchRL and other libraries UX and design choices.

In contrast with Stable-baselinesRaffin et al. [2021] or EfficientRL Liu et al. [2021], TorchRL aims to provide researchers with the necessary tools to build the next generation of control algorithms, rather than providing precisely benchmarked algorithms. For this reason, TorchRL's code will usually be more verbose than SB3's because it gives users full control over the implementation details of their algorithm. For example, TorchRL does not make opinionated choices regarding architecture details or data collection setups. Nevertheless, the example repertory and tutorials are available to assist those who wish to get a sense of what configurations are typically deemed more suitable. The trainer class and multiple examples, tutorials and a rich documentation are available to help users get started with their specific problems.

On a similar note, we note that Tianshou Weng et al. [2022] entry point for most algorithms is usually a `Policy` class that contains an actor, possibly a value network, an optimizer and other components. In TorchRL, these items are kept separate to allow users to orchestrate these components at will.

Another difference between TorchRL and other frameworks lies in the data carriers they use: relying on a common class to facilitate inter-object communication is not a new idea in the RL and control ecosystem. Nevertheless, we believe that TensorDict elegantly brings features that will drive its adoption by the ML community. In contrast, Tianshou's `Batch` has a narrower scope than `TensorDict`. Whereas the former is tailored for RL, the latter can be used across ML domains. Additionally, `TensorDict` has a larger set of functionalities and a dedicated `tensordict.nn` package that blends it within the PyTorch ecosystem as shown above.

Next, TorchRL is less an extension of a simulators than other libraries. For instance, Tianshou mainly supports Gym/Gymnasium environments Brockman et al. [2016] while TorchRL is oblivious to the simulation backend and work indifferently with DeepMind control, OpenAI Gym or any other simulator.

Finally, TorchRL opts for a minimal set of core dependencies (PyTorch, NumPy and tensordict) but a maximal coverage of optional external backends whether it is in terms of environments and simulators, distributed tools (Ray or submitit) or loggers. Restricting the core dependencies comes with multiple benefit, both in terms of usability and efficiency: we believe that RLlib's choice of supporting multiple ML frameworks severely constraints code flexibility and increments the amount of code duplication. This has a significant impact on its `SampleBatch` data carrier, which is forced to be a dictionary of NumPy arrays, thus leading to multiple inefficient conversions if both sampling and training are performed on GPU.