Journal of Machine Learning Research 27 (2026) 1-15 Submitted 8/23; Revised 3/26; Published 6/26

torchgfn: A PyTorch GFlowNet library

Joseph D. Viviano*[α], Omar G. Younis*[α], Sanghyeok Choi*[α,η], Victor Schmidt[α,β], Yoshua Bengio[α,γ,δ], and **Salem Lahlou[ζ,ω]**

JOSEPH@VIVIANO.CA, OMAR.YOUNIS@MILA.QUEBEC, SANGHYEOK.CHOI@ED.AC.UK, SALEM.LAHLOU@MBZUAI.AC.AE

Editor: Alexandre Gramfort

[α]Mila, [β]Entalpic, [γ]Université de Montréal, [δ]CIFAR, [ζ]MBZUAI, [η]University of Edinburgh, [ω]Work started at Mila. *denotes co-first authorship [1]

Abstract

The growing popularity of generative flow networks (GFlowNets or GFNs) among a range of researchers with diverse backgrounds and areas of expertise necessitates a library that facilitates the testing of new features (*e.g.*, training losses and training policies) against standard benchmark implementations, or on a set of common environments. We present `torchgfn`, a PyTorch library that aims to address this need. Its core contribution is a modular and decoupled architecture which treats environments, neural network modules, and training objectives as interchangeable components. This provides users with a simple yet powerful API to facilitate rapid prototyping and novel research. Multiple examples are provided, replicating and unifying published results. The library is available on GitHub (https://github.com/GFNOrg/torchgfn) and on PyPI (https://pypi.org/project/torchgfn/).

Keywords: generative flow networks, GFlowNets, probabilistic models, sampling, amortized inference, open-source software

1. Introduction

Generative Flow Networks (GFlowNets, GFNs; Bengio et al., 2021, 2023b) are probabilistic models over discrete or continuous sample spaces with a compositional structure. They are also stochastic sequential samplers that generate objects from a target distribution, which is given by its unnormalized probability mass function R, referred to as the reward function.

The rapid growth of GFN research has led to several open-source implementations. While this signals a healthy ecosystem, the majority of the existing open source code is bespoke for a given project, and existing libraries are often tailored for specific domains (*e.g.*, molecular generation[2]), full applications designed around complex experimental frameworks[3], or are more focused on performance than extensibility or ease of use [4].

We introduce `torchgfn`, the first library to offer a general-purpose, highly modular, and user-centric PyTorch-based (Paszke et al., 2019) toolkit designed from the ground up for extensibility and modification. Its design philosophy, centered on the strict separation of concerns, empowers researchers not only to replicate existing work but, more importantly, to accelerate the next generation of GFN research across a wide spectrum of applications. It decouples the environment definition, the sampling process, and the parametrization used for the GFN loss. The library aims to introduce

1. ordered by geographic proximity to Saint Joseph's Oratory of Mount Royal at the time of publication. Co-first authors agree that they can list their names in any order on their CVs.
2. github.com/recursionpharma/gflownet
3. github.com/alexhernandezgarcia/gflownet, used for *e.g.*, AI4Science et al. (2023).
4. github.com/d-tiapkin/gfnx, a JAX-based library supporting discrete environments.

new users to GFNs and their continuous variants (Lahlou et al., 2023; Sendera et al., 2024; Zhang et al., 2024), and facilitate the development of new algorithms.

The library is shipped with many example environments in the `Gym` which capture various GFN use-cases. Some examples include 1) a discrete environment where all states are terminating states (*e.g.*, `HyperGrid` (Bengio et al., 2021)); 2) discrete environments where all trajectories are of the same length, but only some states are terminating (*e.g.*, `DiscreteEBM`); 3) Simple autoregressive settings (*e.g.*, `BitSequence`); 4) continuous environments with state-dependent action spaces (*e.g.*, `Box`); 5) discrete sampling of graphs (*e.g.*, `BayesianStructure` (Deleu et al., 2022) and `Ring`); and 6) diffusion-based samplers for continuous distributions (*e.g.*, `DiffusionSampling` (Sendera et al., 2024)). This list cannot be exhaustive as `Gym` will be actively developed by maintainers and the community as `torchgfn` is adopted into various application domains. These examples help users learn the theory of GFNs, reproduce published environments, illustrate the proper use of the library, and provide examples of how a user may extend base classes for specific use-cases.

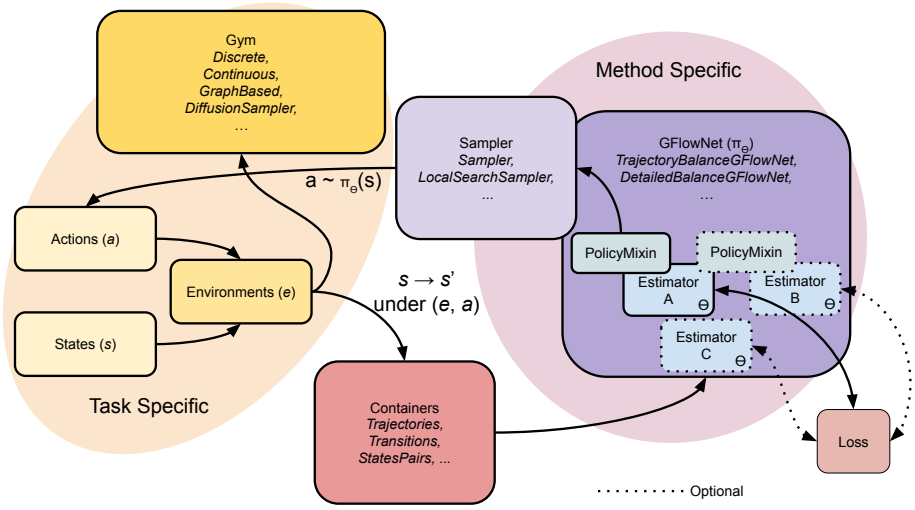

Figure 1: A schematic of the modular `torchgfn` repository structure. `States` and `Actions` are top-level abstractions used to interface between the stateless `Envs`. `Containers` are generic objects which hold `States-Actions` sequences drawn from `Samplers`. These interface (via a `PolicyMixin`) directly with the parametrization of a `GFlowNet` subclass (*e.g.*, `TrajectoryBalanceGFlowNet`) implementing a specific training objective. Parametrization is handled by one or more `Estimators` (`pytorch nn.Module` wrappers) which represent the learned function approximators (*e.g.*, policies π, state flows), trained using the `GFlowNet`'s loss. A set of `Envs` is made available through the `Gym` module.

2. Library Architecture

The conceptual graph of the library hierarchy is shown in Figure 1 (see appendix Figure 2 for a detailed breakdown of the full dependency structure of the library). At a high level, the Markov decision process (MDP) defined over a directed acyclic graph (DAG) is implemented by the interaction of `States` and `Actions` via a stateless `Env` (environment), which produces new `States` instances. As these elements are intimately tied to the training task, they are defined by the user in one location within the `Env` itself. `State-Action Containers` interact with `GFlowNets` parameterized by `Estimators` to sample actions (via a `Sampler`) which explore the DAG. The `Estimators` are trained using the `GFlowNet`'s `.loss()` method, and the interface between these estimators and the sampler is mediated by a `PolicyMixin`, enabling generic samplers to work with any kind of estimator

architecture. A full treatment on each library element is available in appendix B, with up-to-date documentation available in the online documentation[5]. There are also many working examples in the repo tutorials[6].

Environments e are *stateless* objects with a step function which produces a state transition s' from s given an action a, *i.e.*, $s \to s'$ under (e, a). Their stateless construction facilitates easy scaling. **States** s are the primitive building blocks for GFlowNet training objects, which contain state-level metadata such as *masks* for disallowing undesirable or impossible state transitions. They can be of any datatype, but must be castable to a format compatible with `pytorch` and/or `torch_geometric`. **Actions** a represent the internal actions of an agent building a compositional object. **Containers** wrap `States` and `Actions` during the sampling process in the correct format for a given `GFlowNet`. **Preprocessors** are responsible for casting the datatypes contained in `States` into those compatible with your defined `Estimators`. **Estimators** are wrappers for `nn.Modules` or other `pytorch` module-like abstractions which transform preprocessed `States` into distributions of actions. **Samplers** define how actions are sampled at each state, relying on an Estimator to produce a distribution to be sampled from. **GFlowNets** encapsulate the required `Preprocessor`, `Estimator`, `Sampler`, and loss-related logic to produce a trained sampler. Aside from `nn.Module`-based support, we also provide support for *sampling graph objects* using `torch_geometric`, see Appendix C for more details.

This shows how to build and train a Trajectory Balanced-based GFN (Malkin et al., 2022):

```
env = HyperGrid(ndim=4, height=8, reward_fn_kwargs={"R0": 0.01})
prep = KHotPreprocessor(ndim=env.ndim, height=env.height)
module_PF = MLP(input_dim=prep.output_dim, output_dim=env.n_actions)
module_PB = MLP(input_dim=prep.output_dim, output_dim=env.n_actions - 1)
gfn = TBGFlowNet(  # Define the GFlowNet.
    pf=DiscretePolicyEstimator(
        module_PF, env.n_actions, is_backward=False, preprocessor=prep),
    pb=DiscretePolicyEstimator(
        module_PB, env.n_actions, is_backward=True, preprocessor=prep),
)
sampler = Sampler(estimator=gfn.pf)
optimizer = torch.optim.Adam(gfn.parameters(), lr=1e-3)

for i in (pbar := tqdm(range(1000))):  # Training.
    trajectories = sampler.sample_trajectories(env=env, n=16)
    optimizer.zero_grad()
    loss = gfn.loss(env, trajectories)
    loss.backward(); optimizer.step()
```

To instead train a SubTBGFlowNet, simply change the parameterization of the GFlowNet:

```
module_logF = MLP(input_dim=prep.output_dim, output_dim=1)
gfn = SubTBGFlowNet(  # Define the SubTrajectory GFlowNet.
    pf=DiscretePolicyEstimator(
        module_PF, env.n_actions, is_backward=False, preprocessor=prep),
    pb=DiscretePolicyEstimator(
        module_PB, env.n_actions, is_backward=True, preprocessor=prep),
    logF=ScalarEstimator(module=module_logF, preprocessor=prep),
)
```

2.1 Design Philosophy in Contrast with Other Libraries

`torchgfn` is designed as a domain-agnostic, general-purpose library, providing out-of-the-box support for discrete, continuous, and graph-based environments, under a unified API optimized for extensi-

5. https://torchgfn.readthedocs.io/en/latest/
6. https://github.com/GFNOrg/torchgfn/tree/master/tutorials

bility. This versatility makes it an ideal platform for foundational research on the GFlowNet framework itself – innovations can be tested on well-understood environments where execution speed and optimization are not the top priority. `recursionpharma/gflownet` (Bengio et al., 2023a, hereafter `recursion`) is explicitly specialized for graph and molecular generation, leveraging `torch_geometric` and `networkx` as its core data representations, while `alexhernandezgarcia/gflownet` (Hernandez-Garcia et al., 2023, hereafter `gflownet`) is most deeply engineered for complex scientific applications such as crystal generation in addition to standard environments, and the JAX-based `gfnx` (Tiapkin et al., 2025) provides discrete environments and fewer abstractions, in exchange for a JIT-compiled sampling loop with substantial speed advantages. While powerful for their respective domains, each library's specialization creates barriers for adaptation to certain problem types and fundamental GFlowNets methods (see Table 1). The core abstractions of `torchgfn` (Appendices B.1 to B.7) are compositional objects that can be modified by users to cleanly implement specific or novel methods.

Table 1: High-level comparison of GFlowNet libraries.

Library	`torchgfn` PyTorch	`recursion` PyTorch	`gflownet` PyTorch	`gfnx` JAX
Philosophy	Modular & Decoupled	Graph (Molecule)-Focused	Environment-Centric	Low-Level & Fast
Envs	Stateless	Stateless	Stateful	Stateless
Focus	General-Purpose	Graph & Molecule	General-Purpose	Discrete Envs
Batching	`torch.Tensor` & `torch_geometric.Data`	`torch_geometric.Batch`	Custom `Batch` Class	`jax.vmap`
Loss Impl.	Classes	Classes	Methods in Agent	User-Defined

Compared with alternatives, `torchgfn` inhabits the middle ground between performance and usability, aiming to serve the needs of users who want to get ideas off the ground quickly and are less concerned with the fastest training time (Table 2, see Appendix E.3 for full benchmark results).

3. Future Work & Conclusions

Table 2: Per-iteration times averaged across 3 seeds on Hypergrid (batch size 128).

Scenario	Library	Time (ms)	Mem (MB)	Params
Small	gflownet	141.4	24	70,915
Small	**torchgfn**	30.8	24	71,430
Small	gfnx	2.5	182	71,429
Medium	gflownet	223.0	26	83,717
Medium	**torchgfn**	77.8	43	84,746
Medium	gfnx	5.9	220	84,745
Large	gflownet	364.7	37	100,101
Large	**torchgfn**	113.3	55	101,130
Large	gfnx	10.8	361	101,129

Some limitations we intend to address during future development include 1) simplifying the extensible, modular design of `States`, `Actions`, and `Envs` to make environment definition easier and be more compatible with `torch.compile`, 2) extend the library with real-world tasks containing more complex state spaces relevant to the language model and AI for science communities, 3) supporting hierarchical sampling natively by storing sampling metadata in `Trajectories`[7] 4) support for multi-objective *meta-environment*s, and 5) the inclusion of reinforcement learning baselines.

In sum, `torchgfn` employs a modular architecture that empowers researchers to develop and test new GFN algorithms as self-contained, importable classes, which we believe addresses an unmet need in the GFN community. It refines GFN building blocks into well-documented, tested abstractions that simplify experimentation and improve reproducibility. We already have many external contributors, and are fostering a developer community for GFNs with clear contribution standards[8]. We intend for it to become the community standard for foundational research.

7. Currently, users would need to subclass the `PolicyMixin` and `Trajectories` classes for such use-cases.
8. github.com/GFNOrg/torchgfn/blob/master/.github/CONTRIBUTING.md

Acknowledgments

The authors would like to extend a great thanks to the members of the community who contributed valuable pull requests: Alexandre Larouche, Mike Arpaia, Idriss Malek, Abhijith Sharma, Ali Parviz, Etienne Collin, Emir Ceyani, Joseph Bunao, Aya Laajil, Sanchit Misra, Chirayu Haryan, & Inel Djafar. We would like to thank Mo Tiwari, Edward Hu, Tristan Deleu, Daniel Jenson, Eric Elmoznino, Nikolay Malkin, Emmanuel Bengio, Alex Hernández-García & Anja Surina for invaluable discussions and feedback. This work was supported by Intel, Samsung, and the Ministry of Economy and Innovation (MEI) of Québec.

References

Mila AI4Science, Alex Hernandez-Garcia, Alexandre Duval, Alexandra Volokhova, Yoshua Bengio, Divya Sharma, Pierre Luc Carrier, Yasmine Benabed, Michał Koziarski, and Victor Schmidt. Crystal-gfn: sampling crystals with desirable properties and constraints. *arXiv preprint arXiv:2310.04925*, 2023.

Emmanuel Bengio, Moksh Jain, Maksym Korablyov, Doina Precup, and Yoshua Bengio. Flow network based generative models for non-iterative diverse candidate generation. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Emmanuel Bengio et al. recursionpharma/gflownet. `https://github.com/recursionpharma/gflownet`, 2023a.

Yoshua Bengio, Salem Lahlou, Tristan Deleu, Edward J Hu, Mo Tiwari, and Emmanuel Bengio. Gflownet foundations. *Journal of Machine Learning Research (JMLR)*, 24(210):1–55, 2023b.

Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. TorchRL: A data-driven decision-making library for PyTorch, 2023.

Tristan Deleu, António Góis, Chris Emezue, Mansi Rankawat, Simon Lacoste-Julien, Stefan Bauer, and Yoshua Bengio. Bayesian structure learning with generative flow networks. *Uncertainty in Artificial Intelligence (UAI)*, 2022.

Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *nature*, 585(7825):357–362, 2020.

Alex Hernandez-Garcia et al. alexhernandezgarcia/gflownet. `https://github.com/alexhernandezgarcia/gflownet`, 2023.

Minsu Kim, Taeyoung Yun, Emmanuel Bengio, Dinghuai Zhang, Yoshua Bengio, Sungsoo Ahn, and Jinkyoo Park. Local search gflownets. *International Conference on Learning Representations (ICLR)*, 2024.

Aya Laajil, Abduragim Shtanchaev, Sajan Muhammad, Eric Moulines, and Salem Lahlou. Curriculum-augmented GFlowNets for mRNA sequence generation. *arXiv preprint arXiv:2510.03811*, 2025.

Salem Lahlou, Tristan Deleu, Pablo Lemos, Dinghuai Zhang, Alexandra Volokhova, Alex Hernández-García, Léna Néhale Ezzine, Yoshua Bengio, and Nikolay Malkin. A theory of continuous generative flow networks. *International Conference on Machine Learning (ICML)*, 2023.

Kanika Madan, Jarrid Rector-Brooks, Maksym Korablyov, Emmanuel Bengio, Moksh Jain, Andrei Nica, Tom Bosc, Yoshua Bengio, and Nikolay Malkin. Learning GFlowNets from partial episodes for improved convergence and stability. *International Conference on Machine Learning (ICML)*, 2023.

Nikolay Malkin, Moksh Jain, Emmanuel Bengio, Chen Sun, and Yoshua Bengio. Trajectory balance: Improved credit assignment in GFlowNets. *Neural Information Processing Systems (NeurIPS)*, 2022.

Ling Pan, Nikolay Malkin, Dinghuai Zhang, and Yoshua Bengio. Better training of GFlowNets with local credit and incomplete trajectories. *International Conference on Machine Learning (ICML)*, 2023.

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. *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Marcin Sendera, Minsu Kim, Sarthak Mittal, Pablo Lemos, Luca Scimeca, Jarrid Rector-Brooks, Alexandre Adam, Y. Bengio, and Nikolay Malkin. Improved off-policy training of diffusion samplers. *Neural Information Processing Systems (NeurIPS)*, 2024.

Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT Press, 1998.

Daniil Tiapkin, Artem Agarkov, Nikita Morozov, Ian Maksimov, Askar Tsyganov, Timofei Gritsaev, and Sergey Samsonov. gfnx: Fast and scalable library for Generative Flow Networks in JAX. *arXiv preprint arXiv:2511.16592*, 2025.

David W. Zhang, Corrado Rainone, M. Peschl, and R. Bondesan. Robust scheduling with GFlowNets. *International Conference on Learning Representations (ICLR)*, 2023.

Dinghuai Zhang, Ricky TQ Chen, Cheng-Hao Liu, Aaron Courville, and Yoshua Bengio. Diffusion generative flow samplers: Improving learning signals through partial trajectory optimization. *International Conference on Machine Learning (ICML)*, 2024.

Appendix A. Library Structure

A full breakdown of the internal library dependency structure can be seen (as of `v2.3.0`) in Figure 2. The high-level library structure is as follows:

- `actions.py`: Objects representing actions.
- `states.py`: Objects representing states.
- `env.py`: Base classes for all environments.
- `containers/`: Objects that hold collections of `States`, `Actions`, `Estimator` outputs, and other auxiliary metadata for training a GFlowNet. This includes `ReplayBuffer`s.
- `preprocessors.py`: functions for converting `States` into objects that `Estimator`s can consume.
- `estimators.py`: `nn.Module` wrappers, in the form of `Estimators` and `PolicyMixin`s, which encompass the learnable parameters of a GFlowNet and their interface with `Sampler`s and log probability calculations.
- `samplers.py`: Classes encompassing sampling logic.
- `gflownet/`: Classes that encapsulate the full parameterization of a trainable GFlowNet, including the loss and sampling procedure.
- `gym/`: A collection of environments.
- `utils/`: misc utility functions.

Appendix B. Codebase Components

B.1 Environments

In `torchgfn`, environments e are stateless objects with a step function which given an action a and state s produces a transformed state s', *i.e.*, $s \to s'$ under (e, a). The stateless nature of the environment is a deliberate design choice which allows for parallelism during training in large scale, multinode settings where queries to the environment (for next state computation and/or reward computation) can be broadcast among as many dedicated compute nodes as required, decoupling the computation bottlenecks potentially imposed when GFlowNet action sampling is significantly faster than environment queries, as is often the case when the environment involves complex energy functions. Environment definition requires the user to specify key components of the MDP, including a representation of the initial state s_0 and terminal state s_f to ensure that the MDP is structured as a pointed DAG (Bengio et al., 2023b), a `States` class factory that contains the logic governing action masking (*i.e.*, keeping track of and enforcing all allowable actions given the current state), and finally an `Actions` class factory that defines a representation of sampled actions legible to the environment. Specific environment abstractions are provided, for example, `DiscreteEnv` allows easily specifying the action space using the total number of actions as an attribute. The environment must either implement a `log_reward()` or `reward()` method which would be called at every terminating state (*i.e.*, a state with only s_f as a child in the DAG) and potentially at non-terminal states if allowed by the environment and the GFlowNet is to be trained using intermediate rewards (Pan et al., 2023)[9].

When defining an environment, besides `s0`, users can optionally define a tensor representing the sink state s_f, which is only used for padding incomplete trajectories. If not specified, `sf` is set to a tensor of the same shape as `s0` filled with $-\infty$.

9. Environment designers should be mindful of numerical stability issues arising from the scale of reward values.

For `DiscreteEnv`s, the user can define a `get_states_indices()` method that assigns a unique integer number to each state, and a `n_states` property that returns an integer representing the number of states (excluding s_f) in the environment. The function `get_terminating_states_indices()` can also be implemented and serves the purpose of uniquely identifying terminating states of the environment, which is helpful for tabular `Estimator`s. Other properties and functions can also be implemented, such as the `log_partition` or the `true_dist` properties.

B.2 States

States are the primitive building blocks for GFlowNet training objects, such as transitions and trajectories, on which losses operate. The provided abstract `States` class must be subclassed for each environment to define s_0, s_f, and the states shape for a single batch element. This enables the `States` instance to contain `Environment`-specific state information, allowing the environment itself to remain stateless. A `States` object is a collection of states, typically stored as a `torch.tensor`-based batch for efficient processing. In cases where the environment requires heterogeneous data, such as graphs with varying numbers of nodes and edges alongside global properties, `torchgfn` leverages `numpy.Array` (Harris et al., 2020) and `tensordict` (Bou et al., 2023), a library from the PyTorch ecosystem. `Numpy` arrays are used for efficient management of `torch_geometric` objects, and `TensorDict` is a dictionary-like container that holds tensors of different shapes and types while allowing for unified batching and device management operations, leading to cleaner and more efficient code which efficiently manages multiple external pytorch libraries towards the goal of training a GFlowNet.

Masks: Not all actions are always possible at all states – this constraint is handled by the `State`'s mask property. `DiscreteStates` and `GraphStates` objects (`States` specialized for environments where states are represented as discrete-variable tensors and graphs, respectively) have both `forward_masks` and `backward_masks`, specifying the actions allowed in each state and the actions that could have produced that state, respectively. Masks can be implemented either as an attribute of the `States` class, which is kept updated via the `update_masks` method defined in the environment, or as a property, allowing them to be generated on demand at each state. While one can technically train a GFlowNet without masks, the number of invalid trajectories sampled during training may be so great that the loss will not converge in reasonable wall-clock time, so it is almost always a practical necessity and important component of state implementation.

DataTypes: Notably, the data contained within a `States` object can take any form (*e.g.*, strings, `numpy` arrays), but `torchgfn` only provides explicit support for some forms which allow for efficient GFN training leveraging `pytorch`, including `torch.Tensor`s and `torch_geometric`'s graph `Data` objects. To handle alternative representations, the user must provide a custom `Preprocessor` to the `Estimator` when defining your `GFlowNet` which transforms these data structures into batched tensors.

B.3 Actions

Actions represent internal actions of an agent building a compositional object: they correspond to transitions $s \to s'$. An abstract `Actions` class is provided. While it is automatically subclassed for simple discrete environments, it must be manually subclassed in other cases because `Actions` require knowledge of both the state representation, and therefore the environment. In the `DiscreteEnv` case, an action is simply an integer representing an index between 0 and $n_{actions} - 1$. Additionally, environments that allow for early termination of trajectories require a `exit_action` tensor corresponding to the termination action ($[n_{actions} - 1]$ for discrete environments), unless all trajectories have a fixed length. For discrete environments, the action set $\{0, \ldots, n_{actions} - 1\}$ contains also a $(n_{actions})$-th *exit* or *terminate* action (*i.e.*, $s \to s_f$), corresponding to the index $n_{actions} - 1$.

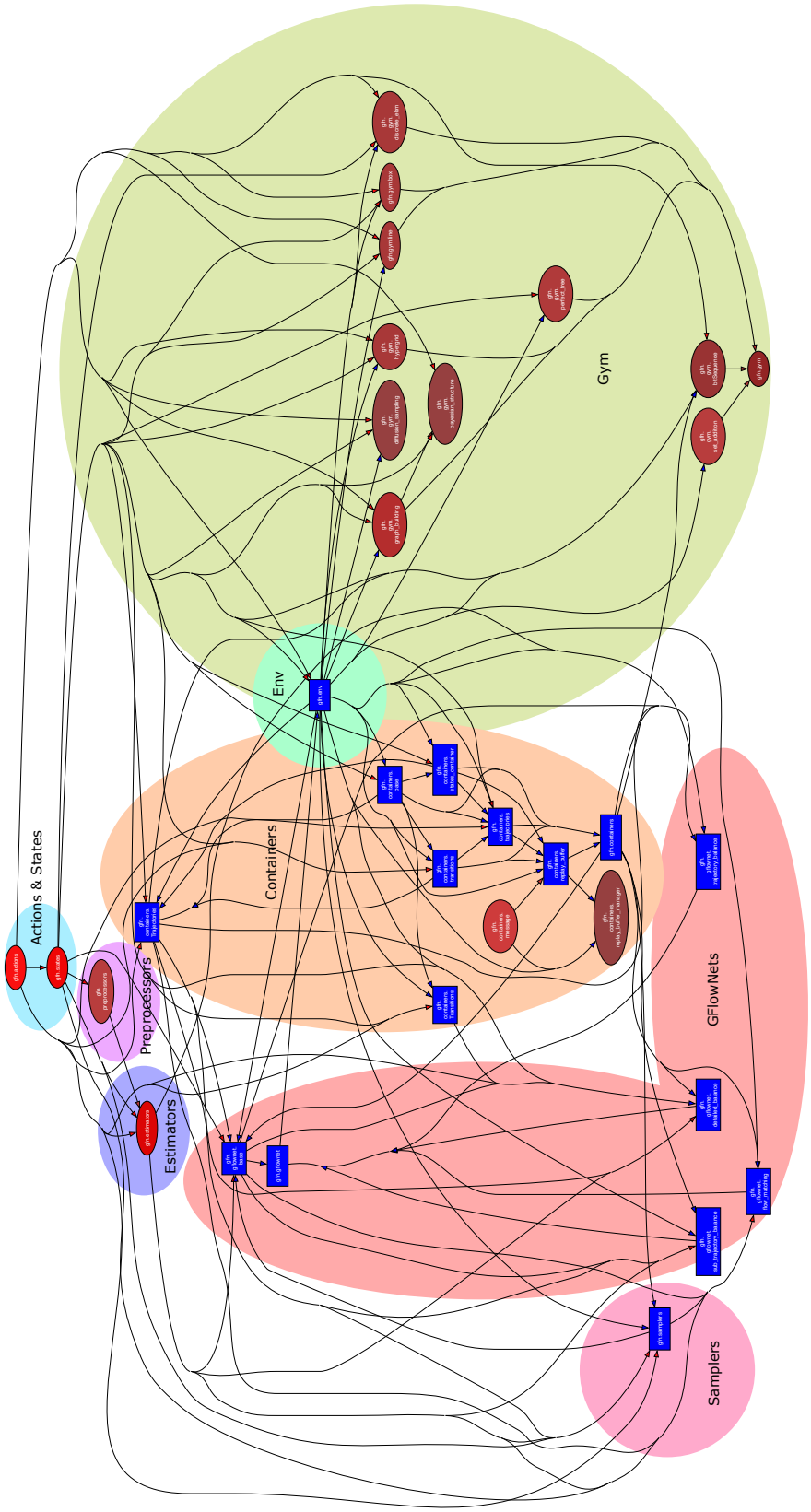

Figure 2: `torchgfn` dependency graph as of `v2.3.0`. `States` and `Actions` are top-level abstractions used to interface between the stateless `Env` and a `GFlowNet`. `Containers` wrap `States`-`Actions` sequences as either `Transition` or `Trajectory`-level information to be used by the remainder of the library. They're drawn from `Samplers` via `Estimators` (wrappers for `pytorch`'s `nn.Module`) representing function approximators (*e.g.*, policies, state flows) housed within the `GFlowNet` subclass (*e.g.*, `TrajectoryBalanceGFlowNet`). The interface between `Samplers` and `Estimators` is mediated by a `PolicyMixin`. The `GFlowNet` subclass contains all loss-relevant logic. `Environments` are made available through the `Gym` module.

B.4 Containers

Containers wrap `States` and `Actions` during the sampling process – they contain all elements required for training a GFlowNet in the form of abstractions such as `Transitions` or `Trajectories`. For example, these can hold the outputs of estimators during the sampling of a trajectory to facilitate the calculation of log probabilities when training a model using off-policy methods, as well as other metadata that might be useful for calculating the loss under a particular parameterization.

Containers are collections of `States`, along with other information, such as reward values or densities $p(s' \mid s)$. Three containers are available:

- Transitions, representing a batch of transitions $s \to s'$.
- Trajectories, representing a batch of complete trajectories $s_0 \to s_1 \to \ldots \to s_n \to s_f$.
- StatesContainer, representing a batch of states (for example, parents and children, *i.e.*, (s_p, s_c), for the flow matching loss (Bengio et al., 2021)).

These containers can either be instantiated using a `States` object or can be initialized as empty containers that can be populated on the fly, allowing the usage of the `ReplayBuffer` class, to enable experience replay while training your GFlowNet off policy to stabilize training with uncorrelated trajectories (Sutton et al., 1998). They inherit from the base `Container` class, indicating some helpful methods. In most cases, one needs to sample complete trajectories. From a batch of trajectories, a batch of states and a batch of transitions can be defined using `Trajectories.to_states()` and `Trajectories.to_transitions()`, in order to train GFlowNets with losses that are edge-decomposable or state-decomposable. These exclude meaningless transitions and dummy states that were added to the batch of trajectories to allow for efficient batching.

B.5 Estimators

Training GFlowNets using a `pytorch` optimizer requires at least one `Estimator` (an abstract subclass of `torch.nn.Module`). In addition to the usual `forward` method, `Estimator`s implement other useful attributes that interface with the rest of the `torchgfn` library, for example, ensuring `torch.nn.Module` outputs have the correct output dimensions for the action space, or the required logic to convert module outputs into probability distributions.

For all `Estimator`s, the `forward` function accepts a `States` object. Neural network estimators require tensors in a particular format, and therefore one may need to define a `Preprocessor` object as part of the environment that transforms the `States` representation into something compatible with the `Estimator` in question (Appendix B.1).

As an example, a `DiscretePolicyEstimator` is an `Estimator` that can be used to define the policies $P_F(. \mid s)$ and $P_B(. \mid s)$ for discrete environments. At initialization, when `is_backward=False`, the required output dimension is `n_actions`, and when `is_backward=True`, it is `n_actions - 1` [10]. These numbers represent the logits of a categorical distribution. The corresponding `to_probability_distribution()` function transforms the logits by masking illegal actions (according to the forward or backward masks contained in the `States` instance), and returns a categorical distribution. Masking is accomplished by setting the corresponding logit to $-\infty$. The function also includes exploration parameters, in order to define a tempered version of P_F, or a mixture of P_F with a uniform distribution. Other simple examples for *discrete environments* include the `Tabular` module, which implements a lookup table that can be used instead of a neural network, and a `UniformPB` module, which implements a uniform backward policy.

For *non-discrete environments*, the user needs to specify their own policies P_F and P_B. Each module should accept a batch of `States` and return batched parameters of `torch.Distribution`s. The distribution required depends on the environment and may also depend on the previous state

10. There is no exit action for backward policies.

itself. Our tutorials and examples contain the various implementation details – for example, in the `Box` environment, the forward policy has support either on a quarter disk or an arc-circle, such that the angle and the radius (for the quarter disk part) are scaled samples from a mixture of Beta distributions[11]. The `to_probability_distribution()` function handles the conversion of the parameter outputs to an actual batched `Distribution` object that implements at least the `sample()` and `log_prob()` functions.

Recurrent Policies: For sequence models that maintain hidden state (*e.g.*, RNN, or autoregressive Transformers), `torchgfn` provides `RecurrentDiscretePolicyEstimator`, which extends the standard policy interface with explicit carry management. The estimator's forward pass accepts both states and a carry dictionary, returning updated logits and the next carry state. A `PolicyMixin` architecture allows these recurrent estimators to integrate seamlessly with the library's sampling and probability calculation utilities through a unified rollout API, handling per-step artifact tracking and context management automatically.

B.6 Samplers

`Sampler` objects define how actions are sampled at each state. They require an `Estimator` that implements the `to_probability_distribution()` method. They also include a method `sample_trajectories()` that samples a batch of trajectories starting from a given set of initial states or s_0. For off-policy sampling, the parameters of `to_probability_distribution()` can be directly passed when initializing the `Sampler`. Samplers can follow any logic, and users can define their own, which can be passed to `GFlowNet` to enable new functionality. For example, see the included `LocalSearchSampler`, which facilitates efficient local exploration by iteratively destroying and re-sampling from high-reward trajectories (Kim et al., 2024).

To accommodate diverse policy architectures, `torchgfn` uses a `PolicyMixin` framework that provides a uniform rollout API for both vectorized (stateless) and non-vectorized (recurrent) estimators. This mixin exposes methods for initializing rollout context, computing distributions, and evaluating log-probabilities, allowing the `Sampler` and probability calculators to remain agnostic to whether an estimator maintains internal state. Per-step artifacts such as log-probabilities and estimator outputs are automatically tracked in a `RolloutContext`, which is managed by the mixin throughout trajectory generation.

B.7 GFlowNets and Losses

GFlowNets can be trained with different losses, each requiring a different parametrization, so these are available as a unified `GFlowNet` object in the library. A key architectural feature of `torchgfn` is the treatment of the GFlowNet loss function as an interchangeable object. A researcher can define a set of `Estimator`s for policies and state flows, and then seamlessly switch the training objective by simply passing these modules to a different `GFlowNet` class instance (*e.g.*, `TrajectoryBalanceGFlowNet`, `DetailedBalanceGFlowNet`, *etc.*). This compositional approach promotes faster and cleaner experimentation. It is a meta-`Estimator` that includes one or multiple `Estimator`s, at least one of which implements a `to_probability_distribution()` function. They must also implement a `loss()` function that takes either states, transitions, or trajectories as input, depending on the loss. The implemented losses are the flow matching loss (Bengio et al., 2021), the detailed balance loss (Bengio et al., 2023b) and its modified variant (Deleu et al., 2022), the trajectory balance loss (Malkin et al., 2022), the sub-trajectory balance loss (Madan et al., 2023), and the log partition variance loss (Zhang et al., 2023).

11. The provided `Box` example shows an intricate scenario, and user-defined environments are not expected to need this much detail in general.

Appendix C. Sampling Graph Objects

Sampling a graph is possible in `torchgfn` using `torch.Tensor` representations (*e.g.*, building adjacency matrices), which can be suboptimal and quite limited in the general case, or by using the included `GraphEnv` and `GraphStates`, which support core operations from `torch_geometric` in the typical `torchgfn` training setup. This allows for `Estimators` to be built using GNN-based `Estimator`, for example, `GraphEdgeActionGNN`, which assumes a fixed number of nodes and samples only edges between them. The action space for graph building can be whichever subset of valid graph actions required for a task, enabling maximum flexibility for environment designers requiring graph-based states. We have multiple examples in the library, *e.g.*, `train_graph_ring.py` and `train_bayesian_structure.py`, exemplifying their use, and will continue to build upon this functionality.

Appendix D. What's New in v2

The v2 release represents a substantial maturation of the library in general, and introduces some major improvements over the initial release from 2022:

- **Graph generation support:** First-class support for graph generation using graph-based states via `torch_geometric` integration into our `GraphStates` class.
- **Conditional GFlowNets:** Support for conditional generation through conditioning tensors passed to estimators, as used in Laajil et al. (2025) for instance.
- **Custom samplers:** Support for user-defined sampling strategies, exemplified by the `LocalSearchSampler` for efficient local exploration (Kim et al., 2024).
- **Recurrent policies:** A `PolicyMixin` architecture that unifies stateless and recurrent estimators through a common rollout API, enabling sequence models (RNN, LSTM, GRU, Transformers) to maintain hidden state during trajectory generation.
- **Diffusion sampling:** Support for diffusion-based samplers through the `DiffusionSampling` environment, enabling GFlowNet training for continuous distributions (Sendera et al., 2024).
- **Rich ecosystem:** Expanded and tested tutorials alongside mature online documentation at https://torchgfn.readthedocs.io.

Appendix E. Developer Experience and Project Maturity

For an open-source library, the quality of the developer experience is critical for adoption and long-term success. `torchgfn` provides a *rich learning ecosystem*, including documentation on ReadTheDocs, a quickstart guide, detailed tutorials with runnable scripts and interactive Jupyter notebooks. This comprehensive approach significantly lowers the barrier to entry for new users. The library employs a *mature testing suite* using `pytest` and *enforces code quality* with `pre-commit` hooks. A continuous integration (CI) pipeline managed by GitHub Actions automatically runs tests, validates tutorials, and handles publishing to PyPI. This level of automation and quality assurance is important for a production-ready library. `torchgfn` utilizes `poetry` for dependency management via a `pyproject.toml` file. This modern approach ensures reproducible environments and simplifies package distribution, aligning with current best practices in the Python ecosystem.

E.1 Greatly Improved Modularity to Support an Expanded Algorithmic Toolkit

Recent development work has focused on greatly improving the modularity of the library to support the introduction of new features. The `PolicyMixin` framework exemplifies this modular design: it

provides a uniform interface for both vectorized (stateless) and non-vectorized (recurrent) policies through a small rollout API that handles distribution computation, log-probability calculation, and per-step artifact tracking. This allows the same `Sampler` and probability utilities to drive different estimator families – discrete, graph-based, conditional, and recurrent – without bespoke integration code. Similarly, the introduction of conditional `Estimators`, `LocalSearchSampler` (Kim et al., 2024), and diffusion sampling support required only local additions or modifications to the library. We believe this modularity will be a major asset going forward as the community develops new methods or applications utilizing GFNs.

E.2 A Rich Hub for Tutorials and Reproducibility

The `torchgfn` library also contains a large volume of interactive tutorials which are tested using continuous integration, ensuring that they continue to function as the library evolves[12], and examples which serve as a rich resource for learning the basics of GFlowNets, as well as reproductions of recently published results, *e.g.*, (Kim et al., 2024; Deleu et al., 2022; Lahlou et al., 2023; Malkin et al., 2022; Zhang et al., 2023)[13]. These examples are also tested under continuous integration. We envision this library supporting a centralized and standardized collection of GFlowNet-related reproductions and code-based learning resources of general use to the community as the suite of tasks grows to encompass various applications in the literature.

E.3 Performance Comparison with Alternative Libraries

We compared `torchgfn`, `gflownet`, and `gfnx` across nine scenario configurations spanning four environment families as of February 10th, 2026. We ran HyperGrid at three scales: small (dimension=2, height=8, state space=$8^2 = 64$), medium (dimensions=4, height=16, state space=$16^4 = 65,536$), and large (dimension=4, height=32, state space= $32^4 = 1,048,576$); Ising at two scales: 6×6 and 10×10; Box2D with either a learned or uniform backward policy; and BitSequence small (sequence size=8, word size=1, modes=2) and medium (sequence size=8, word size=2, modes=4). All scenarios use Trajectory Balance loss. Across all runs, the GFN policy architecture was matched as closely as possible, with some tolerance for library-specific decisions (for example, the handling of exit actions) leading to a standard deviation of parameter count changes $\pm 10.2\%$. Each run begins with 50 warmup iterations that are excluded from timing. This allows PyTorch CUDA kernels to cache and JAX functions to JIT-compile, ensuring subsequent measurements reflect steady-state performance. Each of the 100 timed iterations is measured independently using `time.perf_counter()`, with explicit device synchronization before and after each iteration (`torch.cuda.synchronize()` for PyTorch; `jax.block_until_ready()` for JAX) to eliminate asynchronous execution artifacts. Per-phase timing (sampling, loss computation, backpropagation, optimizer step) is recorded where possible; `gfnx` reports only total step time as its JIT-compiled training step cannot be instrumented without overhead. Each scenario is run at batch sizes 32, 128, and 512 to characterize scaling behavior. `gflownet` is skipped for Ising 10×10 at batch size 512 due to prohibitive $\mathcal{O}$(batch $\times$ action space $\times$ trajectory length) scaling in its action indexing. All experiments are repeated across 3 random seeds (0, 1, 2). We report the mean iteration time across seeds, with standard deviation capturing seed-to-seed variability. For each (scenario, batch_size, library, seed) combination we record: mean and standard deviation of per-iteration wall-clock time, per-phase timing breakdowns, peak device memory allocation, and total trainable parameter count, which are shown in Table 3. Log partition function estimates were checked before and after training as a sanity check that optimization executed correctly.

In Figure 3 a general trend emerges: `torchgfn` occupies a comfortable middle ground between `gflownet` and `gfnx`, with runtimes roughly $2 - 10\times$ faster than the former and $10 - 20\times$ slower

12. github.com/GFNOrg/torchgfn/tree/master/tutorials/notebooks
13. github.com/GFNOrg/torchgfn/tree/master/tutorials/examples

than the latter. With respect to `gflownet`, runtime comparisons were heavily dominated by the batch size, trajectory length, and action space of the environment, which is most noticeable in our experiments in the Ising environments. The `gflownet` library has lower fixed overhead than `torchgfn` leading to superior performance at small batch sizes, but multiple design decisions of the library prevent the user from leveraging tensor vectorization along the batch dimension and cause runtimes to scale with the batch size. There are also some operations in $\mathcal{O}(batch\_size \times action\_space)$ time which can reduce performance with large action spaces. On the other hand, `gfnx` is able to leverage JIT compilation, providing substantial speedups over `torchgfn` at all batch sizes. This process highlighted to us multiple design decisions in `torchgfn` which are incompatible with using `torch.compile`, such as the structure of our `States` and `Actions` classes. We plan to resolve these in a series of improvements on the road map to v3 of the library. We do not have reason to believe that we will be able to fully close the performance gap with JAX, which as a framework is much better suited to fully compiled training loops, but we suspect we will be able to substantially close the gap while retaining the ease of development and extensibility of PyTorch.

These benchmark results can be re-computed on demand (using up-to-date commits from all libraries) using the `benchmark/` folder now included in the `torchgfn` repo. We will use this on an ongoing basis to improve the competitive performance of our library relative to its peers.

Table 3: Per-phase iteration time (ms, batch size 128), peak memory, and parameter count. Phase times averaged over seeds; "—" indicates JIT-compiled step (not decomposable into individual timings).

Scenario	Library	Sample	Loss	Backward	Optimizer	Total	Mem (MB)	Params
bitseq_small	torchgfn	16.1	2.9	2.7	0.4	22.1	25	76,578
bitseq_small	gfnx	—	—	—	—	1.7	519	74,520
bitseq_medium	torchgfn	9.7	2.9	2.7	0.4	15.8	21	75,554
bitseq_medium	gfnx	—	—	—	—	1.4	519	72,468
box_2d	gflownet	22.4	30.4	7.3	0.5	60.7	22	74,784
box_2d	torchgfn	23.2	3.4	14.9	0.8	42.3	23	82,495
box_2d_uniform_pb	gflownet	22.5	30.5	7.3	0.5	60.8	25	74,784
box_2d_uniform_pb	torchgfn	9.6	2.2	6.2	0.4	18.4	23	74,528
ising_6x6	gflownet	2616.3	929.5	88.2	0.9	3635.0	149	131,367
ising_6x6	torchgfn	72.9	2.9	2.0	0.5	78.3	56	112,530
ising_6x6	gfnx	—	—	—	—	3.9	513	103,020
ising_10x10	gflownet	12413.1	3189.6	244.8	1.5	15849.0	703	246,119
ising_10x10	torchgfn	198.3	5.2	4.0	0.6	208.0	166	194,706
ising_10x10	gfnx	—	—	—	—	9.8	513	168,748
hypergrid_small	gflownet	63.4	62.8	14.3	0.8	141.4	24	70,915
hypergrid_small	torchgfn	25.7	3.0	1.6	0.5	30.8	24	71,430
hypergrid_small	gfnx	—	—	—	—	2.5	182	71,429
hypergrid_medium	gflownet	110.1	79.5	32.7	0.8	223.0	26	83,717
hypergrid_medium	torchgfn	71.6	3.0	2.7	0.5	77.8	43	84,746
hypergrid_medium	gfnx	—	—	—	—	5.9	220	84,745
hypergrid_large	gflownet	197.0	122.2	44.8	0.7	364.7	37	100,101
hypergrid_large	torchgfn	105.0	3.0	4.8	0.5	113.3	55	101,130
hypergrid_large	gfnx	—	—	—	—	10.8	361	101,129

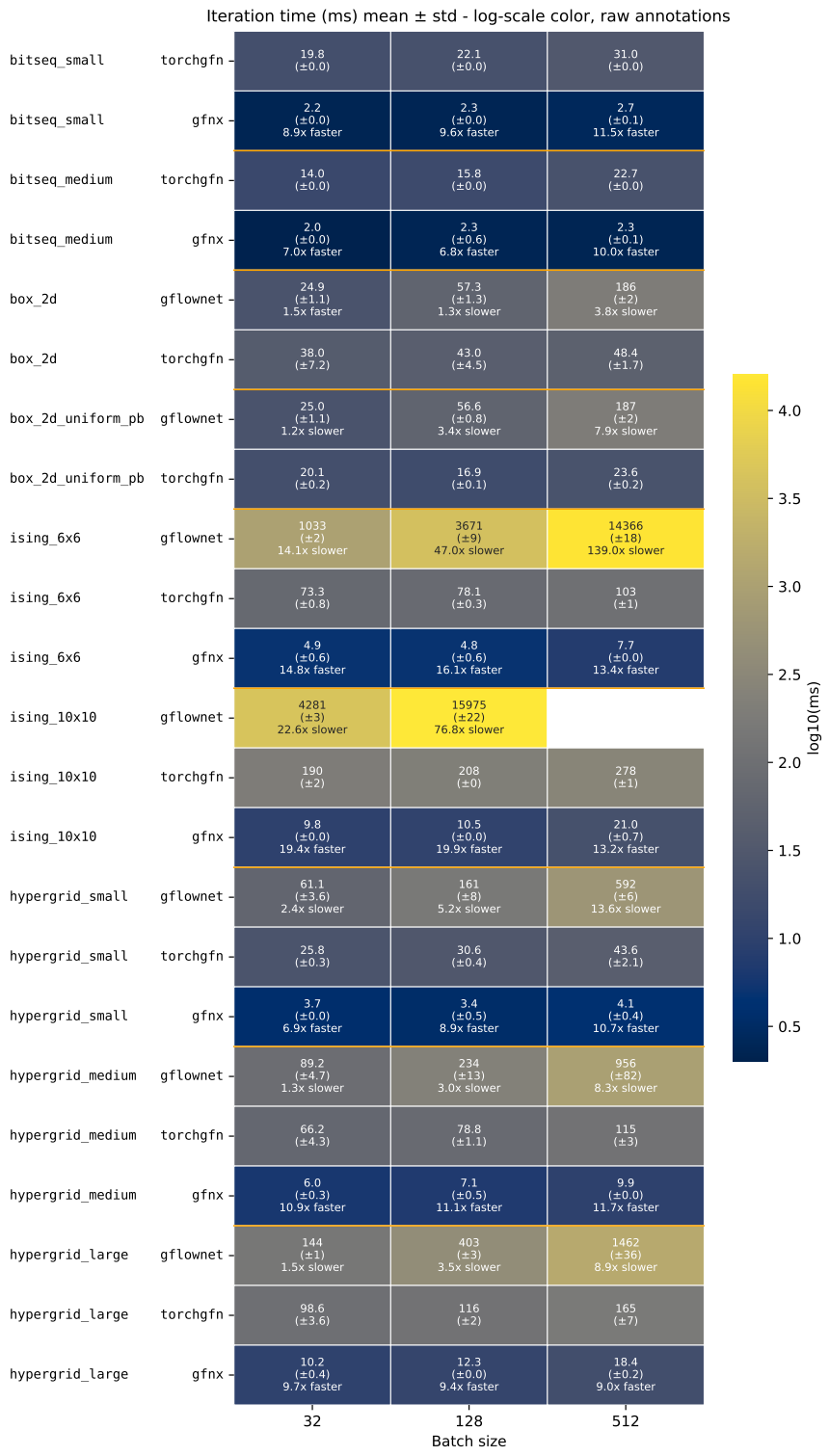

Figure 3: Comparison of the `torchgfn` library per-iteration time (ms) against the `gflownet` and `gfnx` library, across 100 iterations with a 50 iteration burn-in, with variance across 3 seeds. Ising 10×10 is not shown for `gflownet` due to untenable computation time. Colormap is presented in log scale, speedups or slowdowns are computed relative to `torchgfn`.