torchgfn: A PyTorch GFlowNet library

Joseph D. Viviano*[α], Omar G. Younis*[α], Sanghyeok Choi*[α,β], Victor Schmidt[α,γ]
Mila Quebec AI Institute[α], University of Edinburgh[β], Enthalpic[γ]
Montréal, QC, CA[α], Edinburgh, Scotland, UK[β], Paris, France[γ]
joseph@viviano.ca, omar.younis@mila.quebec, sanghyeok.choi@ed.ac.uk
∗ denotes co-first author, ordered by geographic proximity to Saint Joseph's Oratory of Mount Royal.

Yoshua Bengio
Mila Quebec AI Institute, Université de Montréal
CIFAR, IVADO
Montréal, QC, CA

Salem Lahlou
MBZUAI, work started at Mila
Abu Dhabi, UAE
salem.lahlou@mbzuai.ac.ae

Abstract

The growing popularity of generative flow networks (GFlowNets or GFNs) from 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 focused on maintainability and reliability, with static type checking, style enforcement, and tests ensuring library integrity during ongoing development using continuous integration. The library is available on GitHub (https://github.com/GFNOrg/torchgfn) and on pypi (https://pypi.org/project/torchgfn/).

1 Introduction

Generative Flow Networks [GFlowNets, GFNs; Bengio et al., 2021a,b] 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 density function R, referred to as the reward function.

The rapid growth of GFlowNet research has led to several open-source implementations. The majority of the existing code is bespoke for a given project, and existing libraries are often tailored for specific domains (*e.g.*, molecular generation[1]) or designed around complex experimental frameworks[2]. 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 GFlowNet research across a wide spectrum of applications.

`torchgfn` enables the fast prototyping of GFlowNet related algorithms in PyTorch [Paszke et al., 2019]. It decouples the environment definition, the sampling process, and the parametrization used for the GFN loss. The library aims to introduce new users to GFlowNets and their continuous variants [Lahlou et al., 2023], and facilitate the development of new algorithms. Included examples help users learn the theory of GFNs and the practice of training them with robust implementations

[1] see github.com/recursionpharma/gflownet

[2] see github.com/alexhernandezgarcia/gflownet, used for *e.g.*, AI4Science et al. [2023].

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Frontiers in Probabilistic Inference: Sampling Meets Learning.

of published environments. They also illustrate the proper use of the `torchgfn` library and provide examples of how a user may extend base classes for specific use-cases, such as implementing new environments or GFN training methods.

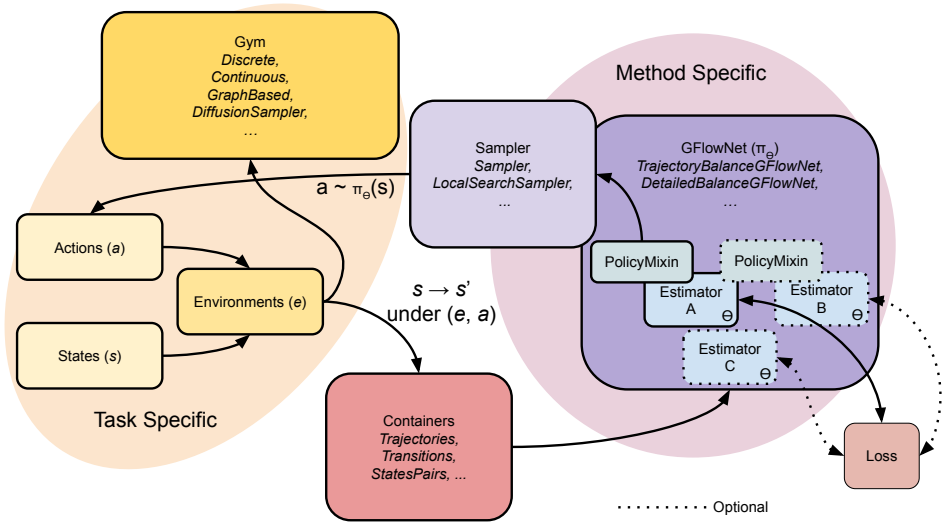

Figure 1: A schematic of the modular `torchgfn` repository structure. `States` and `Actions` are top-level abstractions used to interface between the stateless `Envs`. `Containers` are generic objects used by the remainder of the library and contain `States-Actions` sequences drawn from `Samplers`, which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

2 Core Concepts

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 `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 `Environment` itself. `State-Action Containers` interact with `GFlowNets` parameterized by `Estimators` to sample actions which explore the DAG [3]. A full treatment on each environment element is available in appendix B.

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). Their stateless construction facilitates easy scaling of training in multinode environments where there is a mismatch in the computational load between state sampling and reward calculation. **States** s are the primitive building blocks for GFlowNet training objects, which also contain state-level metadata such as *masks* which facilitate training by disallowing undesirable or impossible state transitions during sampling. 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 which can be sampled from. **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.

[3] See `tutorials/examples` for many working examples.

Aside from `nn.Module`-based support, we also provide support for *sampling graph objects* using `torch_geometric`, see appendix C for more details.

3 Workflow

Here, we show small code snippets to build and train a Trajectory Balanced-based GFN [Malkin et al., 2022] that exemplify the library's modular structure.

```
env = HyperGrid(ndim=2, height=8)
prep = KHotPreprocessor(ndim=env.ndim, height=env.height)

# Define the GFlowNet.
mod_PF = MLP(input_dim=prep.output_dim, output_dim=env.n_actions)
mod_PB = MLP(input_dim=prep.output_dim, output_dim=env.n_actions - 1)
gfn = TBGFlowNet(
    pf=DiscretePolicyEstimator(
        mod_PF, env.n_actions, is_backward=False, preprocessor=prep),
    pb=DiscretePolicyEstimator(
        mod_PB, env.n_actions, is_backward=True, preprocessor=prep),
)
sampler = Sampler(estimator=gfn.pf)
optimizer = torch.optim.Adam(gfn.parameters(), lr=1e-3)

# Training.
for i in (pbar := tqdm(range(1000))):

    trajectories = sampler.sample_trajectories(env=env, n=16)
    optimizer.zero_grad()
    loss = gfn.loss(env, trajectories)
    loss.backward()
```

In order to instead train a SubTBGFlowNet, we simply slightly change the parameterization of the GFlowNet:

```
...
mod_logF = MLP(input_dim=prep.output_dim, output_dim=1)
gfn = SubTBGFlowNet(
    pf=DiscretePolicyEstimator(
        mod_PF, env.n_actions, is_backward=False, preprocessor=prep),
    pb=DiscretePolicyEstimator(
        mod_PB, env.n_actions, is_backward=True, preprocessor=prep),
    logF=ScalarEstimator(module=mod_logF, preprocessor=prep),
)
...
```

4 Positioning Within Existing Ecosystem

4.1 Architectural Philosophy

`torchgfn` is designed as a domain-agnostic, general-purpose library. It provides out-of-the-box support for discrete (e.g., `HyperGrid`, `BitSequence`), continuous (e.g., `Box`), and graph-based (e.g., `GraphBuilding`) environments, all under a unified API. This versatility makes it an ideal platform for foundational research on the GFlowNet framework itself, where innovations can be tested on simple, well-understood environments. In contrast, `recursionpharma/gflownet` is explicitly specialized for graph and molecular generation, leveraging `torch_geometric` and `networkx` as its core data representations. Similarly, `alexhernandezgarcia/gflownet`, while supporting various environments, is most deeply engineered for complex scientific applications like crystal generation. While powerful for their respective domains, their specialization creates a higher barrier for adaptation to novel problem types, and especially so for the development of fundamental methods on GFlowNets themselves.

The most critical differentiator for `torchgfn` is its rigorous adherence to the principle of separation of concerns, resulting in a compositional architecture that fosters modularity. The core abstractions

of the GFlowNet algorithm itself (appendices B.1 to B.3, B.5 and B.7) allow users to mix, match, and replace individual parts without disrupting the entire system. Most notably, the GFlowNet training objective itself is an interchangeable `GFlowNet` object, cleanly separating the algorithm from the neural network models (`Estimators`), and all elements that contribute to the training of a GFlowNet are compositional objects that can be modified by users to implement specific or novel methods. A detailed comparison between this library and other major alternatives can be found in appendix D.

4.2 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 (found in `tutorials/notebooks`), 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., 2023, Deleu et al., 2022, Lahlou et al., 2023, Malkin et al., 2022, Zhang et al., 2023a] (found in `tutorials/examples`). We envision this library supporting a centralized and standardized collection of GFlowNet-related reproductions and code-based learning resources that are broadly useful to the community as the suite of tasks grows to encompass various applications in the literature.

4.3 Limitations

In our quest to make `torchgfn` a modular platform with which to develop *new GFlowNet methods*, we necessarily have had to trade off some increased complexity, particularly in the definition of environments. Improving ease of environment definition is a continued priority of the project, but is more complex in `torchgfn` than packages that make more strict assumptions about the nature of the GFlowNet agent, and/or allow the environments to carry a state, such as `recursionpharma/gflownet` and `alexhernandezgarcia/gflownet`. Other limitations of the library, which we plan to improve in the future, include:

- There is no metadata at the trajectory level which easily permits sampling a trajectory with different GFlowNet policies at different timesteps for some forms of hierarchical sampling. Currently, users need to subclass the `Sampler` and `Trajectories` classes for such use-cases. Next iterations of the library will streamline this process.
- Multi-objective GFNs can be implemented using a *meta-environment* which contains multiple environments, but this not naively supported yet.
- RL baselines are not yet included in the library, which are important for benchmarking.

5 Conclusion and Future Work

`torchgfn` is a general-purpose PyTorch library for GFlowNets that prioritizes modularity, extensibility, and the user experience. Its core contribution is a decoupled architecture that treats environments, function approximators, training objectives, and sampling logic as interchangeable components, supporting discrete, continuous, and graph-based domains through a single, unified API. It represents a significant refinement of GFlowNet concepts into a set of well-documented and tested abstractions. By separating the core concerns of environment dynamics (`Env`), data containers (`States`, `Actions`), function approximators (`Estimator`s, and training algorithms (`GFlowNet`)s, the library represents a flexible, compositional framework for building on existing methods. This design dramatically simplifies experimentation and improves the readability and reproducibility of research built upon it. Our project already has many external contributors and we are in the process of fostering a dynamic developer community for GFlowNets with proper enforcement of standards around contribution[4]. `torchgfn` serves a more fundamental role, intended as the go-to community standard for foundational research, providing the robust and accessible toolkit necessary to accelerate the next wave of GFlowNet innovation across the entire spectrum of the field. We expect the library will continuously be improved, and immediately plan to incorporate more tasks and environments, particularly real-world tasks with much more complex state spaces relevant to the large language model and AI for science communities, to enable benchmarking in domains of broad interest to researchers.

[4] https://github.com/GFNOrg/torchgfn/blob/master/.github/CONTRIBUTING.md

References

Emmanuel Bengio, Moksh Jain, Maksym Korablyov, Doina Precup, and Yoshua Bengio. Flow network based generative models for non-iterative diverse candidate generation. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021a. URL https://openreview.net/forum?id=Arn2E4IRjEB.

Yoshua Bengio, Tristan Deleu, Edward J. Hu, Salem Lahlou, Mo Tiwari, and Emmanuel Bengio. Gflownet foundations, 2021b.

Mila AI4Science, Alex Hernández-García, 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.

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 *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.

Salem Lahlou, T. Deleu, Pablo Lemos, Dinghuai Zhang, Alexandra Volokhova, Alex Hernández-García, L'ena N'ehale Ezzine, Y. Bengio, and Nikolay Malkin. A theory of continuous generative flow networks. *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.

Minsu Kim, Taeyoung Yun, Emmanuel Bengio, Dinghuai Zhang, Yoshua Bengio, Sungsoo Ahn, and Jinkyoo Park. Local search gflownets. *arXiv preprint arXiv:2310.02710*, 2023.

Tristan Deleu, António Góis, Chris Emezue, Mansi Rankawat, Simon Lacoste-Julien, Stefan Bauer, and Yoshua Bengio. Bayesian structure learning with generative flow networks. In *Uncertainty in Artificial Intelligence*, pages 518–528. PMLR, 2022.

David W Zhang, Corrado Rainone, Markus Peschl, and Roberto Bondesan. Robust scheduling with gflownets. *arXiv preprint arXiv:2302.05446*, 2023a.

Ling Pan, Nikolay Malkin, Dinghuai Zhang, and Yoshua Bengio. Better training of gflownets with local credit and incomplete trajectories. In *International Conference on Machine Learning*, pages 26878–26890. PMLR, 2023.

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.

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.

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.

David W. Zhang, Corrado Rainone, M. Peschl, and R. Bondesan. Robust scheduling with gflownets. *International Conference On Learning Representations*, 2023b. doi: 10.48550/arXiv.2302.05446.

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

Alex Hernández-García et al. alexhernandezgarcia/gflownet. `https://github.com/alexhernandezgarcia/gflownet`, 2023.

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

Marcin Sendera, Minsu Kim, Sarthak Mittal, Pablo Lemos, Luca Scimeca, Jarrid Rector-Brooks, Alexandre Adam, Yoshua Bengio, and Nikolay Malkin. Improved off-policy training of diffusion samplers. *Advances in Neural Information Processing Systems*, 37:81016–81045, 2024.

A Example Environments Included

The library is shipped with many example environments in the `Gym` which capture various GFN use-cases. Some examples include 1) a discrete environments where all states are terminating states (*e.g.*, HyperGrid [Bengio et al., 2021a]); 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`), and 5) discrete sampling of graphs (*e.g.*, `Bayesian Structure` [Deleu et al., 2022] and `Ring`). This list cannot be exhaustive as `Gym` will be actively developed by these maintainers and the community as `torchgfn` is adopted into various application domains.

B Core Concepts

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., 2021b], 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 to easily specify 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][5].

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 $-\infty$.

For `DiscreteEnvs`, 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 `Estimators`. Other properties and functions can also be implemented, such as the `log_partition` or the `true_dist_pmf` 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

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

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 to 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.Tensors` 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 $n_{actions} - 1$.

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 $p(s' \mid s)$. Two containers are available:

- Transitions, representing a batch of transitions $s \to s'$.
- Trajectories, representing a batch of complete trajectories $s_0 \to s_1 \to \ldots \to s_n \to s_f$.

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.

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_transitions()` and `Trajectories.to_states()`, 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, `Estimators` 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 `Estimators`, 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` [6]. 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.Distributions`. The distribution required depends on the environment and may also depend on the previous state 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[7]. 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.

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., 2023].

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 `Estimators` 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 `Estimators`, 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., 2021a], the detailed balance loss [Bengio et al., 2021b] 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., 2023b].

[6] There is no exit action for backward policies.

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

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.

D Comparison with Alternatives

To position `torchgfn` within the existing ecosystem, we compare it to two other influential open-source GFlowNet implementations: 1) `recursionpharma/gflownet` [Bengio et al., 2023], a framework specialized for molecular and graph-structured data, and 2) `alexhernandezgarcia/gflownet` [Hernández-García et al., 2023], a powerful framework used for complex, experiment-driven research that implements a large variety of environments.

Many other libraries manage training configurations via a single `GFlowNetAgent` class with a large amount of internal conditional logic which is managed via sophsiticated configurations systems such as `Hydra` files[8]. For example, `alexhernandezgarcia/gflownet` implements the core logic of the GFlowNet algorithm in a `GFlowNetAgent` class which simultaneously implements all supported GFlowNet objectives, as well as a `Policy` module which handles the sampling procedures. For one to innovate or modify the GFlowNet architecture or algorithm itself, one would need to subclass or directly modify this implementation. Furthermore, training is often handled by a dedicated class, with additional configuration available. These design decisions understandably are made to allow the developers to focus their effort on sophisticated environments for various applications. This simplified design tightly couples the training orchestration with the specific GFlowNet algorithmic implementations, making extending the core GFlowNet algorithm difficult without requiring the user to change many elements of the library simultaneously. The architecture of `torchgfn` empowers researchers to develop and test new GFlowNet objectives as self-contained, importable classes, promoting less friction while prototyping new ideas, which we believe addresses an unmet need in the software ecosystem of broad appeal to the research community. A summary of the differences between the libraries is provided in table 1.

Table 1: High-level comparison of GFlowNet libraries.

Aspect	`torchgfn`	`recursionpharma/gflownet`	`alexhernandezgarcia/gflownet`
Philosophy	Modular & Decoupled	Graph Task-Focused	Environment-Centric & Single-Class Algorithm Implementation
Environments	Stateless	Stateless	Stateful
Focus	General-Purpose	Graph & Molecule	Complex Applications
Data Handling	`torch.Tensor` & `torch_geometric.Data`, wrapped in custom classes	`torch_geometric.Batch`	Custom `Batch` Class
Loss Impl.	Interchangeable Classes	Algorithm Classes	Methods in Agent
Extensibility	High (Compositional Architecture)	Moderate (Focus on Graph Sampling)	Moderate (Single-Class GFLowNet Architecture, Hydra-Based Config)

[8] github.com/facebookresearch/hydra

E Library Structure

A full breakdown of the internal library dependency structure can be seen (as of `v2.0.0`) 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.
- `preprocessors.py`: functions for converting `States` into objects that `Estimators` can consume.
- `modules.py`: `nn.Module` wrappers, in the form of `Estimators`, which encompass the learnable parameters of a GFlowNet.
- `samplers.py`: Classes encompassing sampling logic.
- `gflownet/`: Classes that encapsulate the full paramaterization of a trainable GFlowNet, including the loss and sampling procedure.
- `gym/`: A collection of environments.
- `utils`: misc utility functions.

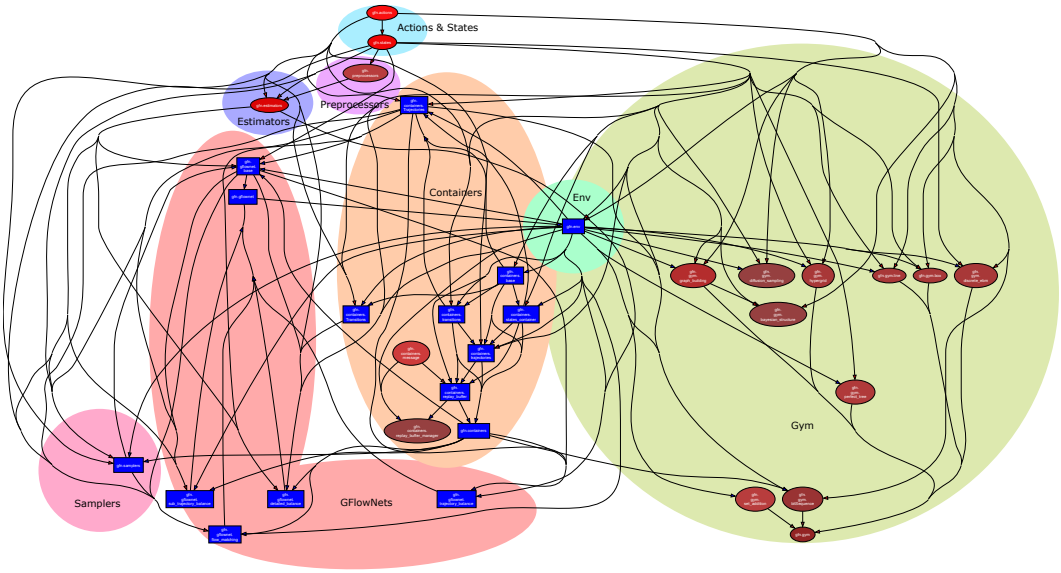

Figure 2: `torchgfn` repository dependency graph as of `v2.3.0`. `States` and `Actions` are top-level abstractions used to interface between the stateless `Env` and a `GFlowNet`. `Containers` are generic objects used by the remainder of the library and contain `States`-`Actions` sequences drawn from `Samplers` via an `Estimator` (wrappers for `pytorch`'s `nn.Module`) which represent the function approximators (*e.g.*, policies, state flows), which are contained with a `GFlowNet` subclass (*e.g.*, `TrajectoryBalanceGFlowNet`) that implements a specific training objective. `Containers` are utilized by both `Samplers` and `GFlowNets` to carry the relevant `Transition` or `Trajectory`-level information. A set of `Environments` is made available through the `Gym` module.

F 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., 2023].
- **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.

F.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. For example, the introduction of conditional `Estimators` and `LocalSearchSampler` [Kim et al., 2023] 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.

G 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.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We present a mature GFlowNet software library with the codebase presented.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We have included both limitations and planned improvements in the limitations section of the paper.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA] .

Justification: Theory is not presented.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [NA]

   Justification: Experiments are not presented.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Justification: The included repo has extensive documentation in the form of README files and docstrings. The non-anonymized version of the code includes a well-formatted readthedocs interactive API documentation hosted online, as well.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so âĂIJNoâĂİ is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [NA]

   Justification: Experiments are not presented.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [NA]

   Justification: Experiments are not presented.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [NA]

   Justification: Experiments are not presented.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This work complies with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: The included examples are all toy and the trained models on these tasks pose no threat or harm to society.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The models trained by `torchgfn` might need to be built with safeguards, but that is beyond the scope of the library developers.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The authors (and their collaborators who are tracked on github) wrote all code presented.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.

- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: A core contribution of this work is the documentation presented online, including the interactive API documentation available on readthedocs as part of the non-anonymized release.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: There were no human experiments.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: There were no human experiments.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.

- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: None of this work relies on LLMs.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.