The GFlowNet Tutorial

Yoshua Bengio, Kolya Malkin, Moksh Jain (2022)

(you have to click on the right-pointing triangle of toggle-able sections to see them if they are hidden)

(tinyurl link to this page: https://tinyurl.com/gflownet-tutorial)

(see also Emmanuel Bengio's initial blog post on GFlowNets as well as the GFlowNet as Amortized Samplers and Marginalization Estimators tutorial)

The Gist

A GFlowNet is a trained stochastic policy or generative model, trained such that it samples objects x through a sequence of constructive steps, with probability proportional to a reward function $R(x)$, where R is a non-negative integrable function. This makes a GFlowNet able to sample a diversity of solutions x that have a high value of $R(x)$.

It is a *stochastic* ***policy*** because it makes sense to apply it when constructing x can be done through a sequence of steps, which can be thought of as **internal actions** (they are meant to construct things like thoughts or plans or explanations). That sequential construction is convenient to sample compositional objects, which can often be defined by composing elements in some order, with generally many possible orders leading to the same object.

Figure 1. A constructive trajectory to build a lego object goes through a sequence of steps (each adding a lego block). The same object could be obtained in exponentially many different ways. The **state** of the GFlowNet describes the object (or partially constructed object) and the **trajectories** leading into it specify how it could have been built. The **forward-sampling policy** of the GFlowNet performs one change to the state at a time (e.g. adding one lego block), but we can also define a **backward-sampling policy** that undoes such changes (e.g. removing a lego block), and they can be combined (e.g., to form a Markov chain by which we can fix our current attempt at a lego boat). When we decide to show our construction, we have reached a **terminal state** and we may get a **reward** which, e.g., is higher for boat-looking constructions, and can also be interpreted as exp(-**energy**), i.e., as an unnormalized probability function. The GFlowNet forward-sampling policy tells us how we could generate the potentially exponentially many boat-looking constructions. An intermediate quantity can be estimated, the **flow**, and the flow in the initial state corresponds to a free energy or normalizing constant (summing over all values of the reward of attainable terminal states). This flow can be generalized to estimate marginalized quantities such as marginal and conditional probabilities over subsets of random variables, which require in principle summing over exponentially many paths or terminal states.

A neural net can be used to sample each of these forward-going constructive actions, one at a time. An object x is done being constructed when a special "exit" action or a deterministic criterion of the state (e.g., x has exactly n elements) has been triggered, when we have reached a terminal state x, and after which we can get a reward $R(x)$. Seen as a stochastic policy, the GFlowNet has an action space (for deciding what to do at each step) and a state space (for the partially constructed objects). Each sequence of actions $(a_0, a_1, \ldots)$ forms a trajectory τ that is a sequence of states $(s_0, s_1, s_2, \ldots)$. There may be many trajectories that lead to the same state s_t (think about how one can construct a set by inserting elements in any order, or build a graph by pasting graph pieces again in many possible orders, transforming a vector through a sequence of stochastic steps like in Denoising Diffusion, or like in Figure 1, constructing the same lego boat in many different ways). The last state s_n of a complete trajectory τ is an object $x \in \mathcal{X}$ that the GFlowNet can sample, and the training objective aims at making it sample x with probability proportional to $R(x)$. Note that [2] discusses how to generalize this to having intermediate rewards and not just at the end of the trajectory.

A GFlowNet is a ***generative model*** because after (and during) training we can sample from it, but in its most basic form its training objective is about matching a reward **function** R rather than fitting a finite dataset (the usual objective for generative models). Since R is not an external quantity (like in typical RL) but an internal quantity (e.g. corresponding to an energy function in a world model), it means that the quality of training of a GFlowNet does not depend on the size of a fixed external training dataset but rather on how much compute is available to query R on many trajectories. Training the GFlowNet can be done by repeatedly querying that function, so we can make our GFlowNet as big a neural net as we can afford computationally, without necessarily worrying about overfitting. Underfitting is the real danger, instead. Going back to the lego construction, we are not trying to find the most boat-looking lego structure, but a way to sample from all the boat-looking structures. Because of the sequential construction of objects x (with an arbitrary number of steps), it is very convenient to generate variable-size structured objects, like sets, graphs, lists, programs or other recursively constructed data structures. Note that a GFlowNet can also define a backward-sampling procedure, i.e., given a constructed object, we could sample a plausible trajectory that could have led to constructing it. The forward-sampling policy is called P_F below and the backward-sampling policy is called P_B.

Importantly, when the reward function R represents the product of a prior (over some random variable) times a likelihood (measuring how well that choice of value of the random variable fits some data), the GFlowNet will learn to sample from the corresponding Bayesian posterior. This makes GFlowNets amortized probabilistic inference machines that can be used both to sample latent variables (as in [14]) or parameters and theories shared across examples (as in [5]). They can also be used to perform approximate and amortized marginalization (without the need for sampling at run-time), as discussed in this other tutorial, and this is useful for training AI systems that can answer probabilistic questions (e.g., about probabilities or expected values of some variables given others, or to estimate information theoretic quantities like conditional entropies or mutual information).

In terms of ***neural net architectures***, the simplest GFlowNet architecture is one where we have a neural net (as large as we can afford) that outputs a stochastic policy $\pi(a_t|s_t)$, where s_t represents a partially constructed object and a_t is one of the possible actions from s_t. In regular GFlowNets, choosing a_t from s_t deterministically yields some s_{t+1}, which means that we can also write $\pi(a_t|s_t) = P_F(s_{t+1}|s_t)$ for that policy. It makes sense to consider deterministic transitions when the policy is an internal policy, not operating in some stochastic external environment, but instead performing computational choices (like what to attend to, what computation to perform, what memory to retrieve, etc). GFlowNets are motivated by the kind of internal policy which could model cognition, i.e., the sequence of micro-choices about internal computation of a thinking agent corresponding to a form of probabilistic inference. P_F stands for the "forward" transition probability along the GFlowNet constructive process to generate these structured internal computational constructions. The same neural net is used at every step, but it produces a stochastic output a_t, from which a next state $s_{t+1} = T(s_t, a_t)$ is obtained (the form of T is application-dependent, e.g., following the rules of chemistry if s_t is a partially constructed molecule represented as a graph and a_t puts an atom as a specific new node connected to an existing node in the graph). Since the state is a variable-size object, the neural net better have an appropriate architecture for taking such objects in input (e.g., an RNN, a graph neural net or a transformer). We can thus more generally see the iterated application of the GFlowNet decisions as a particular form of recurrent stochastic net where the hidden recurrent state (s_t) is stochastic.

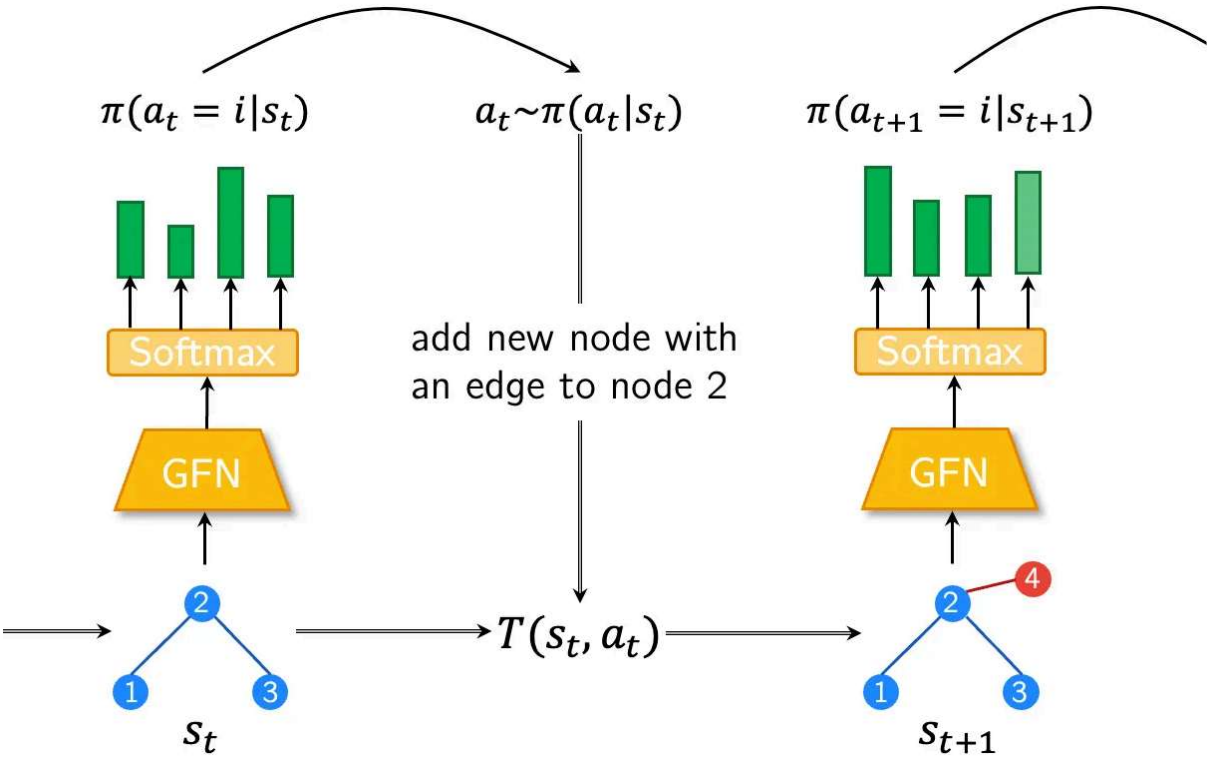

Figure 2: the most basic component of a GFlowNet is a neural network that defines its constructive policy, i.e., how to sample the next state s_{t+1} given the previous state s_t, through the choice of an action a_t. The new state s_{t+1} becomes the input for the next application of the GFlowNet policy denoted P_F or π, until a special "exit" action signals the end of the sequence, that an object $x = s_n$ has been generated, and a reward $R(x)$ is provided to encourage or discourage the policy from generating such a solution. The training objective optimizes the parameters of P_F towards making x be generated with probability proportional to $R(x)$.

GFlowNets for Brain-Inspired Reasoning

Brain sciences show that conscious reasoning involves a sequential process of thought formation, where at each step a competition takes place among possible thought contents (and relevant parts of the brain with expertise on that content), and each thought involves very few symbolic elements (a handful). This is the heart of the Global Workspace Theory (GWT), initiated by Baars (1993,1997) and extended (among others) by Dehaene et al (2011, 2017, 2020) as well as through Graziano's Attention Schema Theory (2011, 2013, 2017). In addition, it is plausible (and supported by works on the link between Bayesian reasoning and neuroscience) that each such step is stochastic. This suggests that something like a GFlowNet could learn the internal policy that selects that sequence of thoughts. In addition, the work on GFlowNet as amortized inference learners [4,5,7,9 below], both from a Bayesian [7] and variational inference [9] perspectives, would make this kind of computation very useful to achieve probabilistic inference in the brain. GFlowNets can learn a type of System 1 inference machine (corresponding to the amortized approximate inference Q in variational inference) that is trained to probabilistically select answers to questions of a particular kind so as to be consistent with System 2 modular knowledge (the "world model" P in variational inference). That System 1 inference machinery Q is also crucial to help train the model P (as shown in several of these papers). Unlike typical neural networks, the inference machinery does not need to be trained only from external (real) data, it can take advantage of internally generated (hallucinated) pseudo-data in order to make Q consistent with P. See also this blog post on model-based machine learning. The real data is of course important to make P (the world-model) compatible with the real world. The kind of sequential probabilistic inference that GFlowNets can perform is a powerful form of learned reasoning machinery, which could be used for interpretation of sensory inputs, interpretation of selected past observations, planning, and counterfactuals (what if an agent had acted differently in the past). The stochastic selection of just a few elements of content (that go into a thought) make GFlowNets a good candidate to implement the "consciousness priors" introduced by Bengio (2017) and elaborated by Goyal and Bengio (2022). In particular, the GWT bottleneck, when applied to such probabilistic inference, would enforce the inductive bias that the graph of dependencies between high-level concepts (that we can manipulate consciously and reason with) is very sparse, in the sense that each factor or energy term only has at most a handful of arguments.

Where would we (or the brain) get the energy or reward function in practice?

A trained GFlowNet is both a sampler (to generate objects x with probability proportional to $R(x)$) and an inference machine (it can be used to answer questions and predict probabilities about some variables in x given other variables, marginalizing over the others). But that is with respect to a given unnormalized probability function $R(x)$ or energy function $\mathcal{E}(x) = -\log R(x)$ for that joint probability model. Where would a learning agent get that energy function from?

We are working on several approaches to do that (more details in this section below):

- The energy function can be parametrized (say with parameters θ) and its parameters learned by maximum likelihood (this is in the realm of energy-based modelling). It turns out that in order to obtain a maximum likelihood gradient on θ, we need to sample from the distribution over objects x captured by the energy function, and GFlowNets can do that for us. If there are latent variables z involved in addition to the observed x (a much more interesting scenario), then GFlowNets can approximately sample both from $p(x, z)$ and $p(z|x)$ which are needed to get this stochastic gradient (and the GFlowNet can also learn to sample from $p(x)$, $p(z)$ or $p(x|z)$ if we train it accordingly).
- We can view parameters θ or the energy function $\mathcal{E}$ itself as a latent variable and be Bayesian about them, i.e., let the GFlowNet sample them too! In this case, given θ (which indexes energy functions) and x (and optionally latent z), the energy is a fixed function (e.g., an MLP with weights θ and inputs (x, z)). See [5] for a first paper on using a GFlowNet to sample from the Bayesian posterior over causal graphs. The GFLowNet objective is naturally suited to learn to sample from Bayesian posteriors because the posterior distribution (which would be estimated by a GFlowNet sampler) must be **proportional** to prior times likelihood. Both log-prior and log-likelihood are easy to compute (or to estimate unbiasedly with a minibatch) and can serve to form the GFlowNet reward.

What Flows?

We use the term flow to talk about unnormalized probabilities that are associated with states s, transitions $s \to s'$ or trajectories τ that the above neural net chooses. The reason we call them flows is because of an analogy with flow networks, with water entering in some places into the network (just one place s_0, in our case) and flowing out of the network through terminal nodes $x \in \mathcal{X}$,as in Figure 3.

We can draw (at least in our head) the directed acyclic graph (DAG) containing all the possible trajectories to construct from scratch all the possible objects $x \in \mathcal{X}$. Each node in that forward-construction DAG corresponds to a state s and each transition corresponds to a constructive action executed from a state s and leading to a next state $s' = T(s, a)$. Imagine water flowing through each state s and into possible next states s' through pipes labeled by one of the possible actions a. The amount of water through the pipe $s \to s'$ (an edge of the graph) relative to the total amount through s can be thought of as the probability that a particle of water going through s will end up in s'. The GFLowNet theory assumes (as a sufficient but maybe not necessary condition) that the graph with these edges is a directed DAG, i.e., there is no way to return to the same state through any action (think about water flowing through pipes and tees, and whether it would make sense to have a cycle, i.e., with water flowing backwards to a place it came from, in a stationary flow of the water, i.e., with the amount of water flowing through each pipe not changing, but different pipes and tees getting different amounts of flow). It does not make much sense for this graph to have cycles (it would not be a flow network). If you want to remove a lego block (undo an action), you can sample the chosen removal action from the backward-sampling policy P_B (which picks a parent of a node of the DAG, while P_F picks a child of a node). Although the graph must be directed and acyclic, it could be an infinite graph (so long as the total amount of water involved is finite). We can define an initial state s_0 from which all trajectories start (and all the water in the DAG must therefore flow through s_0) and a special terminating action $\top$ taken in state s and forcing the transition to a corresponding terminal state x to take place. Since we are not given the actual target distribution $\frac{R(x)}{\sum_x R(x)}$ from which we would like the GFlowNet to sample with its policy but just the unnormalized probabilities $R(x)$, we can just fix the flow into terminal state x to the given value $R(x)$. The total amount of water flowing in the DAG is the flow $F(s_0)$ going through the initial state s_0 and is then distributed through the network-DAG across all the terminal states, through which we we would like to achieve the constraint that the amount of water flow should be $F(x) = R(x)$.

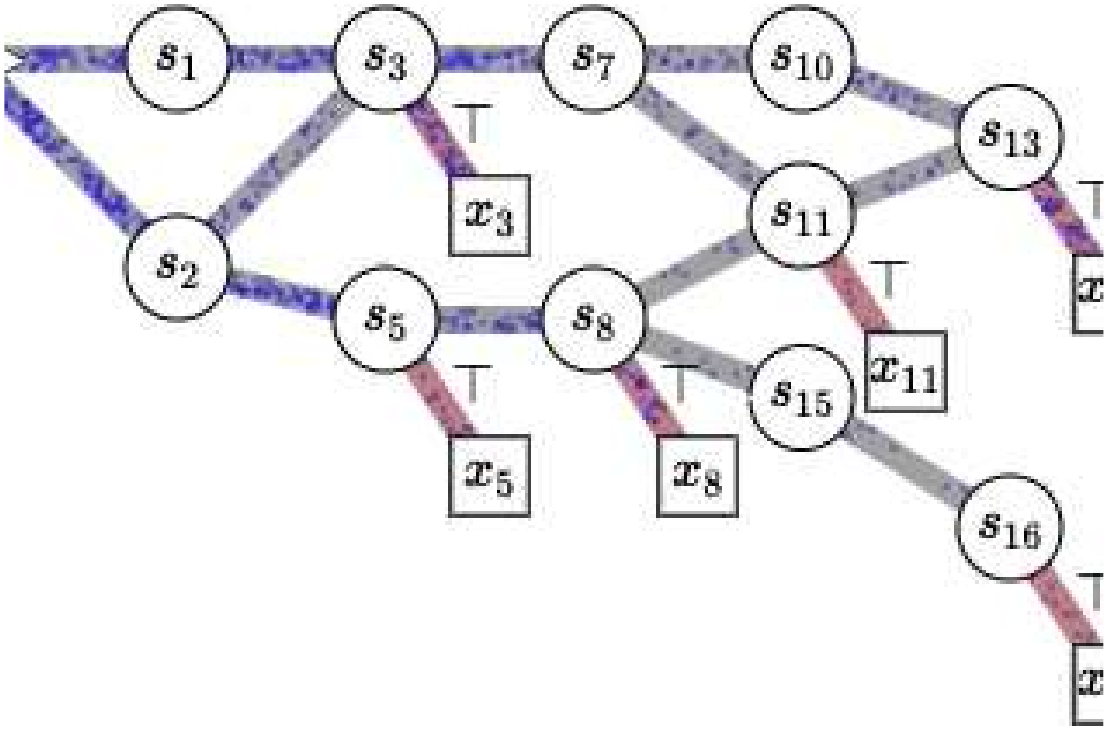

Figure 3: illustration of the flow network (the blue dots can be thought of as water particles flowing through the network) seen as a DAG with all the possible trajectories through states (as circles, partially constructed objects, or squares for terminal states with a fully constructed sample) and transitions between states (where water is flowing). The initial state s_0 is illustrated with a triangle, and the flow through it must equal the sum of the flows through all the terminal states. Transitions between states correspond to picking a particular action in that state. In general, there can be multiple possible actions leading from one state to another (e.g., $s_3 \to s_7$ and $s_3 \to s_8$). The terminal states are reached via a special terminating action $\top$ (red transitions) available in some states (and possibly the only option in some states like s_{13} or s_{16}). Note also that the number of water particles entering an internal state must equal the number of particles coming out: this is the flow-matching constraint. We also want to constrain the flow through each terminal state x to be $R(x)$.

It should be obvious but can be formally proven that if the flows are "matched" (like they would if it was really water flowing through pipes and we had reached a stationary regime of water flowing), we should have $F(s_0) = \sum_x R(x)$, which is the partition function of the unnormalized density R, i.e., the total amount of water into the terminal nodes should match the water entering by s_0. Training the GFlowNet then could be done by estimating the flow functions $F(s)$ and $F(s \to s')$, specifying the amount of water flowing through states and transitions, given that we need to achieve some desired flow in the terminal states $F(x) = R(x)$. The stochastic policy associated with the GFlowNet corresponds to the flows on the outgoing transitions, normalized:

$$\pi(a|s) = \pi(s \to s' = T(s,a)|s) = P_F(s'|s) = \frac{F(s \to s')}{\sum_{s''} F(s \to s'')} \quad (1)$$

The forward policy can thus be used to sample trajectories, starting at the initial state s_0. Similarly, the backward policy is

$$P_B(s|s') = \frac{F(s \to s')}{\sum_{s''} F(s'' \to s')}. \tag{2}$$

The above two equations are forced to be consistent (i.e. there is an F that gives rise to both P_B and P_F) when F satisfies the flow-matching constraint (the amount of entering flow equals the amount of outgoing flow), which is not necessarily true when F is estimated by a neural network that is being trained (and we have not fully completed training and brought the training loss to 0 everywhere). We can parametrize the GFlowNet in various ways, involving estimators for P_F, P_B and F which in practice may not be perfectly consistent with each other and the flow matching constraint.

▼ Markovian Flow

For those who want, we can dig a bit deeper into the semantics of flows, but much more can be found in [2]. We can define the flow function not just on states and transitions but on whole trajectories τ, and in fact define everything from the trajectory flow function $F(\tau)$:

$$F(s) = \sum_{\tau : s \in \tau} F(\tau) \tag{3}$$

$$F(s \to s') = \sum_{\tau : (s \to s') \in \tau} F(\tau) \tag{4}$$

The above should make a lot of intuitive sense if you think of F as counting the number of water particles together sharing some property, e.g. going through a node s, and edge $s \to s'$, or a complete path $\tau = (s_0, s_1, \ldots, s_n)$.

A flow function is called **Markovian** if the future path of particles of water flowing through s do not depend on their previous history to each s. Equivalently,

$$F(\tau = (s_0, s_1, \ldots, s_n)) = Z \prod_{t=1}^{n} P_F(s_t | s_{t-1}). \tag{5}$$

We can thus see an equivalence between two ways of specifying a GFlowNet:

1. we specify its edge flow, from which we can get the node flow by summing incoming or outgoing edges, and we can get the transition probabilities by normalizing as in Eq. (1), or
2. we specify just its total flow $Z = F(s_0)$ and its transition probabilities P_F, from which we can get the trajectory flow by Eq. (5) and the state and edge flows (in principle) by the above two sums (Eqs 3-4). For example, using a recursive definition, we could get

$$F(s) = \sum_{s'} P_F(s | s') F(s'). \tag{6}$$

Note that if we divide both of the F's in the above equation by $F(s_0) = Z$, we get the usual Markov transition, with $P(s) = F(s)/Z$ the fraction of the total flow going through s, and also the probability of passing through s when sampling complete trajectories from the GFlowNet. Note that $\sum_s P(s)$ is not 1 in general, because this is the probability that a particle of water goes through s, but that is not mutually exclusive with the event where the particle goes though some s' that is reachable from s in the DAG.

Note that if we had a non-Markovian flow (i.e., particles choose their next transition based not just on the current state but on earlier events in the trajectory), we could just make it Markovian by incorporating the information encoded in the earlier time points of the trajectory into a richer state.

Main Theorem

A core theorem for GFlowNet theory, already shown in the NeurIPS paper [1] says that if we manage to train the flow function F such that the amount of water entering any state equals the amount of water coming out of it (the flow-matching constraint) and the flow through terminal states matches or is defined by $R(x)$, then sampling from the GFlowNet policy (Eq. 1) will generate terminal states x with probability $R(x)/\sum_{x'} R(x')$.

See a more elaborate mathematical construction of flows and their properties in the GFlowNet Foundations paper [2]. See the theory for the extension to continuous action and state spaces (or hybrid one) [12].

Training Objectives

This section can be skipped if you do not care about how GFlowNets can be trained. However, you may enjoy another tutorial which focuses on the amortization principle exploited by GFlowNets and giving rise to their training objectives.

Several training objectives have already been proposed for GFlowNets, all aiming at constructing a neural net that outputs the flows and/or the transition probabilities of the forward policy such that they define a proper flow, that matches the desired flow through the terminal states x, i.e., $F(x) = R(x)$. A fundamental difference between training of GFlowNets and training of typical functions or probability distributions in ML is that the training of the GFlowNet itself (when the reward function is given) is NOT with respect to a finite dataset, but with respect to an infinite object, i.e., the given reward function, which specifies a reward value for each of the exponentially large (also possibly infinite in case we have continuous actions) set of all possible trajectories. Hence we have to rethink the notions of overfitting and how to evaluate training progress, with GFlowNets (a better analogy would be what you get in RL or online learning, where you can call the environment's reward function as much as your compute resources allow, and there is generally no point in iterating multiple times on the same set of examples). It means that if we are willing to spend a lot of compute cycles, then we can afford a larger GFlowNet: it is not the dataset size that determines the optimal GFlowNet size but rather the amount of compute you're willing to put in the GFlowNet neural net for both its use (one pass through the network for each action) and its training (how many training trajectories are you willing to go through).

▼ Flow-matching objective

With the convention that $F(s \to s') = 0$ if there are no actions to turn s into s', we can turn the flow-matching constraint

$$\sum_{s'} F(s' \to s) = \sum_{s''} F(s \to s'') \tag{7}$$

into a loss measuring the mismatch between the sum of entering flows into a state and the sum of outgoing flows from a state. For example, the NeurIPS 2021 paper [1] used

$$(\log(\sum_{s'} \hat{F}(s' \to s)) - \log(\sum_{s''} \hat{F}(s \to s')))^2 \tag{8}$$

where $\hat{F}$ indicates that this is an estimator for a correct flow (e.g. the output of the GFlowNet neural net) and the role of the $\log$ is to emphasize the matching of relative probabilities rather than absolute ones (but the latter would also be possible). The paper also inserted a small additive δ inside these logs in order to emphasize the absolute errors on large flows, thus obtaining a compromise between relative and absolute error. A similar loss is defined to make $\log \hat{F}(x) \approx \log R(x)$ for the visited terminal states x.

▼ Detailed balance objective and backward policy

The GFlowNet Foundations paper [2] introduced another objective which also indirectly yields flow-matching, but using a different parametrization: instead of the flow (from which one can derive the forward policy), the neural net with s as input can directly output a softmax over the possible actions in each state, i.e., the forward policy $\pi(a|s) = P_F(s'|s)$ where $s' = T(s,a)$ or if there is no such action, as well as a *backward policy* $P_B(s|s')$ which specifies a distribution over parents s of each state s'. That backward policy can be used to sample a trajectory backwards, starting at a given object x (e.g., an example from a dataset, as in [4]), and tells us a possible way to construct x. The constraint that the detailed balance objective tries to satisfy during training is

$$F(s)P_F(s'|s) = P_B(s|s')F(s'). \tag{9}$$

Because the detailed balance objective does not require computing a sum over all parents of a state, it is useful when the number of parents is large. A modified variant of the DB loss was introduced in [5] (called "SubTB" in that paper) and leads to better training and less mode collapse in settings where a termination action is possible from every intermediate state.

Uniform backward policy. It can be shown that for any chose of the backward policy P_B, there exists a unique forward policy P_F and flow function F that, together with the backward policy, satisfy (5). This means that we can fix P_B to a simple distribution of our choice, such as uniform over the parents of every state. This is a common choice for initial development of GFlowNets in new environments, as having one less neural net to learn makes coding and debugging easier. However, it removes extra degrees of freedom from the optimization and usually leads to slower learning and a poorer learned policy.

Trajectory balance objective

A bit of algebra can be used to turn the above per-state constraints into a trajectory-level constraint (satisfied by any complete trajectory starting in s_0 and ending in a terminal state $s_n = x$:

$$F(s_0) \prod_{t=1}^{n} P_F(s_t|s_{t-1}) = R(s_n) \prod_{t=1}^{n} P_B(s_{t-1}|s_t). \qquad (10)$$

The motivation for the corresponding loss (again trying to match logarithms of these products within a squared error form) is that it provides an immediate gradient to *all* the states visited in a complete trajectory (whether it was sampled forward from s_0 or backward from an x). As discussed in [1], the flow-matching objective is similar to temporal difference objectives in RL and the flow-matching constraint to the Bellman equation. This may yield slow credit assignment in long sequences because the mismatch of flows must be propagated from the terminal states to earlier states and earlier states, etc. As shown in [3], using the trajectory balance objective yields more efficient credit assignment and faster convergence of the GFlowNet.

Subtrajectory balance objective

A variant of trajectory balance, introduced in [3] and elaborated in [8], seems to yield better training than the original trajectory balance objective, which was the best-performing training objective until [8]. This is thus the recommended default objective as of this writing. It interpolates between the per-transition objective of the detailed-balance objective and the per-complete-trajectory objective of trajectory balance, and can be considered analogous to TD(λ) in RL, which interpolates between the unbiased high-variance TD(1) — analogous to trajectory balance — and the biased but low-variance TD(0). Instead of distributing a gradient signal equally over a whole trajectory, it relies on the flows as intermediate targets for shorter subsequences.

▼ Forward-Looking GFlowNet for very Long Trajectories

What can we do if the trajectories are very long or terminal states can only occur after extremely long sequences of actions? This paper shows that if we can extend the energy function so that it can be computed over all states and not just terminal states, then it is possible to train the GFlowNet using incomplete trajectories. This occurs naturally if the energy function is additive in terms that can be computed along the trajectory, or if the intermediate states and the terminal states live in the same space over which the energy function can be applied. This is achieved by reparametrizing the flow function in terms of the already accrued energy and a forward-looking flow that has factored it out. This trick allows to considerably accelerate training of the GFlowNet, even when we can sample complete trajectories (with a terminal state), because credit information is available locally. It cannot be combined with the Trajectory Balance objective (which requires complete trajectories) but it can be applied with the Detailed Balance and SubTrajectory Balance objectives.

This trick is also closely related to the Modified DB training objective introduced in [5] (called "SubTB" in that paper), which is useful when it is possible to terminate from **all** intermediate states.

▼ Training policy

One of the important choices in training a GFlowNet is also that of a training (or behavior) policy, i.e., on which trajectories should we measure one of these objectives and compute gradients wrt the GFlowNet parameters. The training policy is generally distinct from the sampling policy P_F learned by the GFlowNet. The training policy can put a different weight (i.e. sampling probability) on different trajectories and thus is part of defining the actual objective function and the gradient for training the GFlowNet. However, any training policy that gives a non-zero probability to all trajectories would yield the same solution if the GFLowNet has enough capacity and is trained long enough, i.e., making the trainign objective 0 for all possible trajectories (which we call "training to completion" and is not realistic in practice but useful to understand the theoretical and asymptotic properties of GFlowNets).

In RL parlance, it means that **GFlowNet training can be off-policy** (not necessarily on the trajectories sampled from the policy itself). However, we suspect that some choices of training policy will lead to more efficient training than others. Similarly to corresponding RL ideas, current GFlowNet implementations insert some amount of exploration during training, either by mixing the online policy P_F with a uniform distribution over actions or temper it (by increasing the temperature in the softmax for P_F, i.e., scaling down its pre-softmax logits).

Definition of a GFlowNet

There are many ways these ideas can be specialized, for example how to parametrize the GFlowNet policy and what the outputs of the GFlowNet should be, what the training objective should be, as well as the training policy, etc. To clarify what we mean by a GFlowNet, [3] proposes this definition:

A GFlowNet is any learning algorithm consisting of:

- *a model capable of providing the forward action distributions $P_F(-|s)$ for any nonterminal state s (which with $F(s_0)$ uniquely determines a Markovian flow F up to a multiplicative constant), although typically the model also provides other quantities, like the other flows $F(s)$, $F(s \to s')$ and/or estimated backward transition probabilities P_B.*
- *an objective function, such that if the model is capable of expressing any action distribution and the objective function is globally minimized, then the flow-matching constraint is satisfied for the corresponding Markovian flow F.*

Specific GFlowNet algorithms typically provide other facilities, such as

- an estimator of *the initial state flow $Z = F(s_0) = \sum_x R(x)$ or partition function*
- a backward policy $P_B(-|s)$ from which one can sample backwards from any given state
- an edge flow function $F(s \to s')$ which estimates the flow through any particular transition
- a state flow function $F(s)$ which estimates the flow through any particular state
- a self-conditional flow function $F(s|s')$ which estimates the flow through s if we only consider the trajectories going through $s' < s$ (occurring on a trajectory from s_0 to s).

As outlined above, with a subset of these one can possibly recover the others (e.g., from the edge flows F, both P_F and P_B can be recovered by normalization). $F(s|s')$ needs its own objective function and can be used to generalize the type of marginalization that can be done, as explained in [2], and recover free energies. **Conditional GFlowNets** (modeling a conditional distribution) are defined in [2] and applied in [11] to obtain Pareto front sampling for multi-objective problems where we are not sure ahead of time how to properly combine multiple objectives.

Marginalization Magic & GFlowNet as Inference Machine

Remember that if we train the GFlowNet perfectly (reaching the flow-matching or trajectory balance constraints, for example), the initial state flow $F(s_0)$ must equal the generally intractable partition function Z, which is the sum over rewards $R(x)$ over all the possible terminal states x. How is that even feasible when the number of training trajectories (seen realistically during training) will be a tiny fraction of the total number of possible trajectories and terminal states? **Training won't generally be perfect but it will work well to the extent that there is generalizable structure in the underlying reward function**, i.e., the learner, by estimating the flow function, is guessing how much probability mass there should be downstream of any kind any state (due to the flows through that state). This can be generalized, as shown in the Foundations paper [2], using what the paper calls conditional flows and self-flows (where the flow function takes an extra state specification as argument), with $F(s|s) = \sum_{x>s} R(x)$, where $x > s$ indicates the terminal states reachable from s. Hence if s represents a graph, then $F(s|s)$ represents the total measure of all the supergraphs (or extensions) of s (i.e., the sum of the rewards $R(x)$ of terminal states x reachable from s).

Exploiting this, the paper then shows how this could be used to approximately answer any question of the form $P(\text{some variables} \mid \text{other variables})$ where the reward function encodes the unnormalized joint over all the variables. Hence, in addition to being **a trainable sampler for any given energy function, a GFlowNet is also a general-purpose trainable inference machine!**

Since intractable sums and intractable sampling are the bread-and-butter of probabilistic machine learning, GFlowNets seem very appealing as neural nets that can represent and perform inference (and possibly Bayesian inference) on complex probabilistic models, especially over compositional data structures like sets or graphs. Note that unlike the use of MCMC for non-parametrically representing very complex posterior distributions, GFlowNets do not represent the distribution via samples but implicitly via their flow or policy neural net.

GFlowNets and MCMC

In [1], we basically replace a classical MCMC sampler by a trained GFlowNet. The advantage of the GFlowNet is that it can exploit the $(x, R(x))$ pairs it has seen while it is being trained to generalize elsewhere, i.e., guess where probability mass might be that it has not visited yet. This makes it possible to generalize from seen modes of the negative energy function to new modes, and directly jump to these modes by sampling from the GFlowNet. Instead, an MCMC has to stochastically search for new modes by making lots of random-walk-like small perturbations. In high dimensions, when the modes are highly separated, it becomes very exponentially unlikely to discover new isolated modes, because most random paths do not lead near that mode (the mode also has to be thin to be difficult to find, and this happens quite a lot with high-dimensional data). The regions of flat low probability between modes are what we call deserts of probability and the challenge of finding these paths, i.e., of mixing between modes, is the main drawback of MCMC methods in high-dimensional distributions. Another difference between standard MCMC and GFlowNets is that the cost of sampling is **amortized** (and this makes GFlowNets close cousins of amortized variational inference methods): we can train the GFlowNet once and then independently sample in one constructive sequence each new x. There is an upfront cost for this, of course, which is to train the GFlowNet, but sampling is then very fast when compared with running an MCMC. There is also no guarantee that the sampler is correct if training the GFlowNet did not work for optimization reasons (say a local minimum in the loss) or because it does not have enough capacity or was not trained long enough. In the case of MCMC, the correctness of the sampler is also only asymptotic (as the number of samples in the chain goes to infinity), but the approximations are different. What we found experimentally is that GFlowNets tend to be better at **finding a diversity modes** that the MCMC may have missed, given some bounded compute budget. This makes GFlowNets interesting to obtain a high diversity of samples, especially if the setting is one where we will want to draw many samples (so the amortization is worth it).

In [4], we also show how a trained GFlowNet (with a forward policy P_F and a backward policy P_B) can be used to form a very efficient large-step Metropolis-Hastings (MH) MCMC: each step of the chain involves resampling only a subset of the variables (a subset of the actions in the GFlowNet trajectory). If the GFlowNet is completely trained (which means long enough, with enough capacity, minimizing the training objective), then the MH reject condition is always satisfied and this is equivalent to an *efficient* large-block Gibbs sampler (i.e. where many of the variables are resampled at each step).