

Neural Bayesian Networks for Modular and Scalable Decision-Making Under Uncertainty

Giovanni Briglia✉

University of Modena and Reggio Emilia
Reggio Emilia, Italy
giovanni.briglia@unimore.it

Stefano Mariani

University of Modena and Reggio Emilia
Reggio Emilia, Italy
stefano.mariani@unimore.it

Marco Lippi

University of Florence
Florence, Italy
marco.lippi@unifi.it

Franco Zambonelli

University of Modena and Reggio Emilia
Reggio Emilia, Italy
franco.zambonelli@unimore.it

Abstract

Bayesian Networks (BNs) are established tools for letting agents reason under uncertainty in sequential decision-making. They split a joint distribution into local conditional mechanisms arranged on a directed acyclic graph, which makes them modular, interpretable, and ready to perform probabilistic inference. Their classical formulation, however, has limitations in both expressiveness and scale: each conditional is usually stored as a table or a linear-Gaussian form, and inference evaluates the graph one node and one query at a time. In this paper we introduce *Neural Bayesian Networks* (NBNs), in which each conditional is a differentiable, tensor-valued mechanism, while the graph and its factorization are unchanged. NBNs retain the modularity and local mechanisms of BNs, can represent any learnable conditional distribution, and run natively on GPUs. We also release NBN, a PyTorch-based library for learning and inference in NBNs, together with a public benchmark. Our evaluation shows gains in both expressiveness and scale: NBNs fit distributions that classical BN libraries struggle to represent, and on large networks and large batches of queries they run two orders of magnitude faster thanks to GPU parallelization. NBN is available at <https://github.com/GiovanniBriglia/NeuralBayesianNetworks>.

Keywords

Bayesian networks; Decision Making; Neural Networks; Model-Based Agents

1 Introduction

An agent that acts under uncertainty would benefit from a model of how its environment responds to its actions to reason about probabilistic outcomes reliably. Model-based reinforcement learning, planning, and diagnosis all can leverage a transition model that can be learned from data and queried under evidence [20, 29, 37]. If the dynamics of the environment are known, at least in the sense of which variables influence which, the decision-making problem can be written as a factored Markov decision process (F-MDP) [10, 19, 30], whose transition model is a Bayesian network (BN) over the state variables. The BN is then the tool that lets the agent reason under uncertainty: it can predict the next state, and it can explain an observation in terms of the state that produced it. This matters most when the state is high-dimensional, because the factored model stays compact where a flat one does not.

Classical BNs, however, were mostly designed for discrete variables and small graphs [33, 44]. A conditional probability table grows combinatorially with the parents and is undefined when a parent is continuous. Additionally, both exact and approximate inference evaluate the conditionals one node and one sample at a time. These limitations stem from representing each conditional as a scalar function and from evaluating the graph sequentially. They are thus inherited by mainstream BN software packages (Table 4).

We propose *Neural Bayesian Network* (NBN) as a new tool for autonomous agents when dealing with decision-making under uncertainty. NBNs keep the structural properties of BNs (i.e., the graph) while changing the conditionals to leverage more expressive and scalable representations based on neural networks. In an NBN each local conditional $p(x_i | x_{\text{Pa}(i)})$ is a differentiable, tensor-valued *mechanism* that takes a batch of parent configurations and returns the full conditional distribution for each row (Figure 1). The graph and the factorization are exactly those of a BN. What changes is their computational representation: the mechanisms can be tabular, parametric, neural or non-parametric; every one of them exposes density, sampling, and gradients through one unified interface; and a whole layer of the graph is evaluated in one tensor operation on a GPU, for thousands of samples and evidence rows at once.

This advancement enables BNs to be used efficiently where they were not practical before: on continuous and non-Gaussian variables without reasonable discretization available, on networks of thousands of nodes, and for thousands of queries answered in one call. As such, NBNs make BNs seamless to integrate with learning-based agent architectures (e.g., reinforcement learning and cognitive architectures) and, more generally, with any machine learning toolchain: the NBN model is a differentiable module that is trained by gradient descent and runs on the same computational accelerators as a neural network (e.g., a GPU), while keeping the explicit graph, modular representation, and local mechanisms factorisation of the BNs’ symbolic model.

Our contributions are:

- (1) **Representation.** We define NBNs (Section 2) and show that they are semantically equivalent to classical BNs, end-to-end differentiable, and whose inference scales with the number of active nodes instead of samples or queries.
- (2) **Library.** We release NBN, a PyTorch implementation with tabular, linear-Gaussian, neural and non-parametric mechanisms

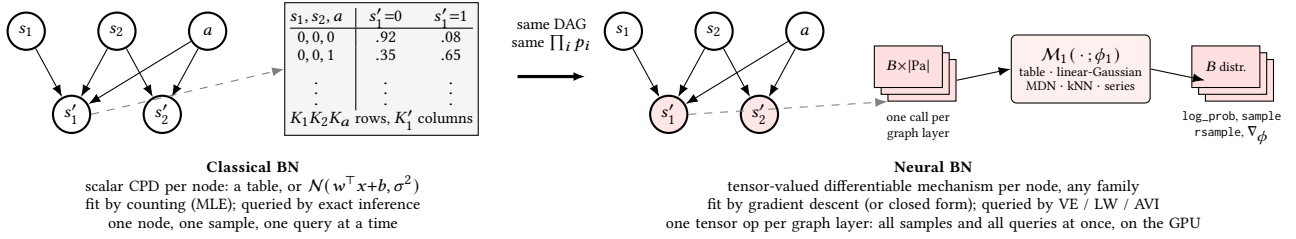


Figure 1: From a classical BN to an NBN. Left: each node stores a scalar conditional, evaluated one node and one configuration at a time. Right: each node stores a tensor-valued differentiable mechanism $\mathcal{M}_i$ that maps a batch of B parent configurations to a batch of conditional distributions. The DAG and the factorization are unchanged.

and with exact, sampling-based and amortized inference engines, all batched on the GPU.

- (3) **Benchmark.** We release a benchmark for BN learning and inference and compare NBN with state-of-the-art BN frameworks on discrete, Gaussian, and non-Gaussian networks of different dimensions (Section 3).

Although NBNs do not yet win all comparisons, the benchmark confirms their potential with regards to scalability and expressiveness, while also pointing to further assessment and development.

The paper is organised as follows. Section 2 introduces NBNs, contrasts them with BNs, and define their properties. Section 3 presents the design of the benchmark and Section 4 its results. Section 5 reviews related works, and Section 6 concludes.

2 Neural Bayesian Networks

A Neural Bayesian Network (NBN) is a BN whose conditional distributions are learnable modules. The graph acts as the architecture of the model: each node hosts one module, its *mechanism*, which receives the values of the node’s parents and returns the distribution of the node (Figure 1). Learning fits the mechanisms to data and inference runs them along the graph, both on batches, as in a neural network.

Similarly to a BN, an NBN over variables $X = (X_1, \dots, X_d)$ is a directed acyclic graph (DAG) $\mathcal{G} = (V, E)$ with $V = \{1, \dots, d\}$ and one conditional distribution $p_i(x_i | x_{\text{Pa}(i)})$ per node, where $\text{Pa}(i)$ are the parents of i in $\mathcal{G}$ [33, 44]. The joint distribution is the product of the conditionals,

$$p(x_1, \dots, x_d) = \prod_{i=1}^d p_i(x_i | x_{\text{Pa}(i)}), \quad (1)$$

so each variable is independent of its non-descendants given its parents. Differently from a BN, where a conditional is a conditional probability table (CPT) or a linear-Gaussian form, an NBN admits arbitrary representations of a conditional distribution: tabular entries, parametric and non-parametric expressions, and neural network based approximations, possibly mixed in the same graph.

Additionally, in a BN, *parameter learning* estimates the conditionals one at a time using the log-likelihood $\sum_n \sum_i \log p_i(x_i^{(n)} | x_{\text{Pa}(i)}^{(n)})$ based on counting for a CPT and on the least squares method for a linear-Gaussian form [22]. In an NBN, instead, every conditional is differentiable, so the same log-likelihood can also be maximized by gradient descent on batches of data—for all the conditionals at

once, and for those that admit no closed-form estimate, too. Also *inference* proceeds one variable at a time in a BN, by computing the posterior $p(x_Q | x_E)$ of query variables Q given observed evidence x_E using variable elimination at a cost exponential in the treewidth of the graph [13]. In an NBN, instead, each conditional is evaluated once for a whole batch of samples and evidence assignments, so that many queries are answered in a single pass over the graph, by exact elimination where the treewidth allows it and by sampling elsewhere. As we will show in the following, BNs are special case of NBNs when the closed-form methods are feasible.

NBN is written in PyTorch [42]. The network is an `nn.Module`. A user builds it from a DAG and a per-node mechanism assignment, calls `fit` once, and then `query`. The benchmark, its configuration, and the complete results are released with the library.¹

2.1 Motivation

BNs are a common “world model” for agents that decide under uncertainty: they are the transition models of factored Markov decision processes in planning and reinforcement learning [10, 19, 30], and the diagnostic models used in troubleshooting, risk analysis, and healthcare [11, 34, 55].

Consider a decision-making agent that has to deal with a Markov decision process (MDP) formalised as $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ with state $s \in \mathcal{S}$, action $a \in \mathcal{A}$ and transition kernel $P(s' | s, a)$. The MDP treats the state as a single object, so P is one distribution over the joint next state for every pair (s, a) , and its size grows exponentially with the number of state variables. If we know which current variables each next-state variable depends on, we can move to a factored MDP [10, 30], which is much lighter (Figure 2):

$$P(s' | s, a) = \prod_{j=1}^n p(s'_j | \text{Pa}(s'_j)), \quad (2)$$

where the parents $\text{Pa}(s'_j)$ of each next-state variable are a few of the current state and action variables. This is a two-slice BN over $X = (s, a, s')$. The same formulation applies to agents that face non-sequential scenarios, such as while doing diagnosis, where the BN has a single slice.

The agent has to deal with two problems:

- P1 Learning.** Given $\mathcal{G}$ and a dataset $\mathcal{D}$ of N samples, estimate every conditional of Eq. 1, for discrete and for continuous variables, *without fixing the shape of the conditional in advance*.

¹<https://github.com/Giovannibriglia/NeuralBayesianNetworks>

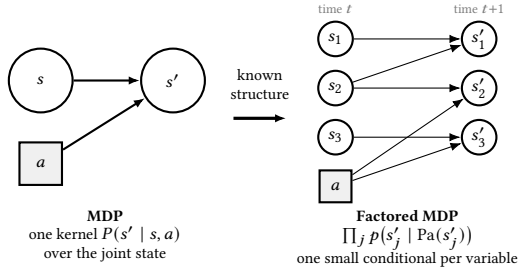


Figure 2: From an MDP to a factored MDP. When it is known which variables influence which, the transition kernel over the joint state becomes a two-slice BN with one conditional per next-state variable.

P2 Inference. Given the learned model, compute $p(x_Q | x_E)$ for query variables Q and evidence x_E , for many evidence assignments at once.

In this work we assume that the structure is given and focus on parameter learning and inference, while leaving structure learning and the sampling of trajectories for planning to future work. More precisely, we make the standard assumptions of parameter learning in BNs. (A1) The graph $\mathcal{G}$ is given and the data distribution factorizes according to it. (A2) The data are complete, so every variable is observed in every sample, and the samples are independent and identically distributed. (A3) Each variable is a discrete variable with finitely many values or a real-valued scalar. All statements below are for a BN over d variables and cover both cases.

Classical BNs solve P1 and P2 only in part, because of three limitations of scalar conditionals evaluated one at a time. (L1) *Representation, which limits P1.* A CPT for X_i has $K_i \prod_{j \in \text{Pa}(i)} K_j$ entries, with K_j the number of values of X_j , and is undefined for a continuous parent. Discretizing a parent introduces a binning parameter and loses information, and the linear-Gaussian alternative cannot represent multi-modal or heteroscedastic conditionals. Both fix the shape of the outcome uncertainty before seeing the data: a table needs the continuous outcome discretized, a Gaussian has one mode and a spread that does not depend on the parents. (L2) *Sequential evaluation, which limits P1 and P2.* A likelihood evaluation on one sample visits the d nodes in order, and each visit is a scalar lookup or a scalar density. Nothing is shared across samples or across queries, so S samples for B evidence assignments cost $\Theta(d \cdot S \cdot B)$ scalar operations. (L3) *Exact inference, which limits P2.* Variable elimination on a network of treewidth w costs $O(d c^w)$ with c being the maximum cardinality [13], and it exists only for discrete and linear-Gaussian networks. Sampling avoids the exponent but inherits (L2).

A solution to P1 therefore needs conditionals that are not restricted to tables and linear-Gaussian forms (L1) and that are fit on batches of data (L2). A solution to P2 needs an evaluation whose cost does not grow with $S \cdot B$ (L2), with exact inference where the treewidth allows it and batched sampling elsewhere (L3). As described in Section 2.2, NBNs provide all of these properties.

2.2 Formal definition and Properties

Let $\mathcal{G}$ be a DAG over $X_1, \dots, X_d$ with $X_i \in \mathcal{X}_i$. An NBN replaces each conditional by a parametrised mapping

$$p_{\phi_i}(x_i | x_{\text{Pa}(i)}) = \mathcal{M}_i(x_{\text{Pa}(i)}; \phi_i), \quad (3)$$

where $\mathcal{M}_i$ takes a batch $X_{\text{Pa}(i)} \in \mathbb{R}^{B \times |\text{Pa}(i)|}$ of parent configurations and returns, for every row, a distribution over $\mathcal{X}_i$ with a density $\log p_{\phi_i}$, a sampler, and where it exists a reparametrised sampler. The joint is still Eq. 1:

$$p_{\phi}(X) = \prod_{i=1}^d p_{\phi_i}(X_i | X_{\text{Pa}(i)}). \quad (4)$$

Two design choices are worth emphasising here. First, $\mathcal{M}_i$ returns the full conditional, not a sample or a point estimate, so learning and inference reach every mechanism through one interface. Second, ϕ_i are tensors on which autograd works, so a table, a linear-Gaussian form and a neural network are instances of the same class and can be mixed in one graph. All nodes at the same depth of $\mathcal{G}$ are evaluated in one batched call.

With this representation, NBNs support the following properties.

PROPOSITION 2.1 (SEMANTIC EQUIVALENCE). *Let p be a BN on $\mathcal{G}$ and p_{ϕ} an NBN on $\mathcal{G}$ with $p_{\phi_i}(x_i | x_{\text{Pa}(i)}) = p_i(x_i | x_{\text{Pa}(i)})$ for all i and all arguments. Then $p_{\phi} = p$, and p_{ϕ} satisfies the global Markov property of $\mathcal{G}$.*

PROOF. Both joints are the product in Eq. 1 of the same factors. The Markov property depends only on the factorization [33]. $\square$

Every classical BN is an NBN with tabular or linear-Gaussian mechanisms, and everything that follows from the factorization (e.g., conditional independences) holds for NBNs.

PROPOSITION 2.2 (DIFFERENTIABILITY). *If every $\log p_{\phi_i}$ is differentiable in ϕ_i , the log-likelihood $\ell(\phi; x) = \sum_i \log p_{\phi_i}(x_i | x_{\text{Pa}(i)})$ is differentiable in ϕ , $\nabla_{\phi_i} \ell$ depends only on the term of node i , and the whole gradient is one backward pass.*

PROOF. ℓ is a sum of terms depending on one ϕ_i ; reverse-mode differentiation of the sum evaluates all terms in a single pass. $\square$

Parameter learning (P1) is therefore gradient descent on $-\ell$, per node since the terms are independent (closed form where the family admits it), or jointly when a downstream differentiable objective couples the nodes.

PROPOSITION 2.3 (COST OF INFERENCE). *Consider a query $p_{\phi}(x_Q | x_E)$ answered by ancestral simulation with S samples and B evidence assignments, and let $A = \text{Anc}(Q \cup E)$ be the active nodes. An NBN issues $O(\text{depth}(\mathcal{G}[A]))$ sequential tensor operations, each of cost $c(\mathcal{M}, B \cdot S)$, so the wall-clock time is $O(\text{depth}(\mathcal{G}[A]) \cdot c(\mathcal{M}, BS))$; a classical implementation issues $\Theta(|A| \cdot B \cdot S)$ scalar operations. For a fixed family, $c(\mathcal{M}, BS)$ is constant in BS until the device saturates.*

PROOF. Nodes outside A are barren and pruned. Within A , all nodes at the same depth have their parents available and are evaluated in one call over the $B \cdot S$ rows; there are $\text{depth}(\mathcal{G}[A])$ such levels. The scalar count is the number of (node, sample, row) triplets. $\square$

The claim is not that NBNs change the dependence on the graph, but that for inference (P2) they remove dependence on S and B up to device saturation, so one or a thousand queries cost the same—on a GPU the saturation point is far, on a CPU it is near.

These gains do not come at the cost of sacrificing the usual BN properties that are behind their adoption: *modularity* is preserved since each mechanism has its own parameters and can be inspected, refit, or replaced without touching the others; the *explicit structure* stands still (in the DAG), which a designer can read and edit; and the “one model for every query” workflow is also there, since a fitted NBN answers $p(x_Q | x_E)$ for any Q and E just like BNs.

Other routes to expressive and scalable probabilistic models give up at least one of them: see Section 5.

2.3 Learning and Inference Methods

Learning mechanisms differ in how much of the conditional is fixed in advance. *Parametric* mechanisms fix its form and have a closed-form fit: a probability table for a discrete child, by counting under a Dirichlet prior [22], and the linear-Gaussian form $\mathcal{N}(Wx+b, \sigma^2)$ for a continuous one [35], by ridge regression [23]. The *neural* mechanism fixes the output form but not its dependence on the parents: a Mixture Density Network (MDN) [8] maps the parents with a multilayer perceptron to the parameters of a Gaussian mixture, fit by gradient descent on $-\ell$. It is still parametric, but linear in the number of parents and multi-modal. *Non-parametric* mechanisms fix no form and grow with the data: KNN is a local mixture over the $k=\lceil\sqrt{N}\rceil$ training rows with the closest parents [53], and Flex-Code [25] expands the conditional density in a cosine basis, one regression per coefficient.

Three inference engines answer $p_\phi(x_Q | x_E)$. *Variable Elimination* (VE) [40, 57] is exact and discrete-only: it drops the nodes that cannot affect the query and sums out the others by multiplying their tables in one batched tensor contraction, so B evidence rows cost one pass. Being exponential in the induced width, it is our exact reference, not a scalable method. *Likelihood Weighting* (LW) [15, 51] simulates S samples with the evidence clamped and averages the query node, each sample weighted by the probability of the evidence; it needs only a sampler and a density per mechanism, so it works with every family. *Amortized Variational Inference* (AVI) [18, 36] avoids the samples LW wastes when the evidence is unlikely: a recognition network $q_\psi(x_{-E} | x_E)$, trained once per model on the evidence lower bound [32], proposes samples close to the posterior for every query at constant cost; it falls back to LW when the proposal is worse than LW’s own Monte-Carlo noise.

3 Benchmark Design

To the best of our knowledge, a fully reproducible benchmark for BNs has not been provided so far. The bnlearn repository [50] offers domains and their structures, and it is the usual source of test networks, but the benchmark configuration is unspecified: which queries to ask, how to obtain the ground truth, which metrics to report, or how to count a method that fails or does not finish is not established. In this way, results are hardly comparable thus advancements difficult to identify. Hence, we built one and release it with NBN. A benchmark is a configuration file that fixes the problems, the queries, the methods and the budgets, and the harness writes

one row per problem, seed, method, query and metric, failures included. Its problems come from two sources: (i) the real domains of bnlearn, and (ii) synthetic domains that we generate. Synthetic networks are needed to define *whatever* network structure (varying nodes and edges configuration) with *whatever* underlying conditional probabilities (discrete, continuous gaussian and continuous non-gaussian), and evaluate sample efficiency, scalability and accuracy across *whatever* BN.

We use the benchmark on NBN to answer three research questions:

- RQ1.** Where the classical BN assumptions hold, does NBN match the classical BN libraries?
- RQ2.** Where they do not hold, how much accuracy is gained?
- RQ3.** Does the cost model of Proposition 2.3 show up as wall-clock time when many queries are answered at once?

The numerical settings that follow are fields of the configuration file. We report here those needed to read the results, and all the others in Appendix A.

Problems. A problem is a triplet of family, number of nodes n , and seed. The *family* is the type of the conditionals of the true model: *discrete*, where every conditional is a CPT; *linear-Gaussian*; or *non-Gaussian*, where every conditional is a set of components forming a Gaussian mixture (that a linear-Gaussian model cannot represent correctly). The seed draws a sparse random DAG over the n nodes and then the parameters of every conditional. From the true model we draw a training set (20 480 rows), a test set that supplies the evidence values of the queries, and a reference set that supplies the ground truth. Every problem is repeated over 5 seeds. The real networks are the 13 networks of Section 4.4. We define a *cell* as the combination of one method on one problem.

Queries. A query has two components: a target variable, and an evidence set, which pairs each observed variable with its value. The evidence values are always copied from a row of the test set, so the evidence is a configuration that the true model produces. Three selectors choose targets and evidence variables, one per purpose:

- (1) *Random* (accuracy): the target and the evidence variables are drawn uniformly for the problem, and the query is asked for the evidence values of 128 test rows; it measures the typical error over evidence values.
- (2) *Topological* (real networks): targets are chosen by their role in the graph, among the most connected nodes (hubs), the nodes whose removal disconnects the graph (cut nodes), and the deep nodes with few descendants (terminals). Each target is asked in the *prediction* direction, with evidence among its ancestors, or in the *diagnosis* direction, with evidence among its descendants.
- (3) *Heaviest* (scalability and speed): only the top-ranked hubs, cut nodes, and terminals are queried, each in both directions, once with the evidence observed (*full*) and once with it marginalized (*empty*).

Reference and metrics. For a discrete target the reference posterior $\hat{p}$ is its empirical distribution among the reference samples that agree with the evidence, and the metric is the total variation (TV) between $\hat{p}$ and the answer q , $\text{TV}(\hat{p}, q) = \frac{1}{2} \sum_{x_Q} |\hat{p}(x_Q) - q(x_Q)|$ [54]. For a continuous target the reference is a sample from $p(x_Q | x_E)$ obtained by likelihood weighting on the true model, and the metric

is the 1-Wasserstein distance (W1) between the empirical distribution $\hat{\mu}$ of this sample and the empirical distribution ν of the samples returned by the method, $W_1(\hat{\mu}, \nu) = \inf_{\gamma \in \Pi(\hat{\mu}, \nu)} \mathbb{E}_{(x, x') \sim \gamma} |x - x'|$, where $\Pi(\hat{\mu}, \nu)$ is the set of couplings of $\hat{\mu}$ and ν [16]. In both cases the reference is a finite sample and not the exact posterior, so the measured accuracy has a floor that all methods share. On the discrete networks of the accuracy scenario it is a TV of about 0.06 at 10 nodes and 0.08 from 50 nodes on, which are the values exact engines reach in Figure 3. We use a sampled reference because an exact one is not available for every family and size: variable elimination on the true model is exponential in the treewidth, and non-Gaussian conditionals have no closed-form posterior. Computing exact references when tractable is left for future work. The main text reports this accuracy metric (TV or W1) for the accuracy, scalability, and real-network scenarios, and the time per query for the speed batching scenario, measured as the wall-clock time of one call divided by the number of answered queries. Complete fit and query times are in Appendix B.

Budgets and failures. Every cell has a budget of 1200 s for the fit, a separate budget of 1200 s for its queries, and a memory cap: 80% of RAM (~ 100 GB on our server, that is a more favourable scenarios for cpu-based libraries e.g. pgmpy.), and the 12 GB of the GPU. It ends with a status: ok, timeout, oom or not_supported. In the figures a bar is the interquartile mean (IQM) over the seeds a method solved, as [2] suggests; f/n is printed when f of n seeds failed, and a missing bar indicates the failure status.

Scenarios. Each scenario is one configuration. *Accuracy* fixes the query load (128 random queries) and varies the family and the size, $n \in \{10, 50, 100, 500, 1000\}$: on the discrete and the linear-Gaussian family the classical model is correct, which isolates RQ1, and on the non-Gaussian family it is misspecified, which isolates RQ2. *Scalability* pushes the network size to $n \in \{10, 10^2, 10^3, 10^4\}$ on the heaviest queries and records where each method stops. *Batching* fixes the network size ($n=50$) and varies only the number $B \in \{1, 128, 512, 1024\}$ of queries answered in one call, which is the variable of Proposition 2.3 (RQ3). *Real networks* repeat the accuracy scenario on published structures with topological queries. *Learning* scores the fitted conditionals without any query and addresses P1.

Methods and hardware. The comparison baselines are pgmpy with maximum-likelihood and with Bayesian CPTs, both queried by its variable elimination; pgmpy linear-Gaussian, which conditions the joint Gaussian in closed form; pomegranate, which is discrete only and uses its built-in loopy belief propagation; and Pyro, where the BN is written as a program with smoothed tables or linear-Gaussian conditionals and queried by importance sampling from the prior with 50 particles (the largest number its sequential sampler affords within the budgets). On continuous networks pgmpy and Pyro always fit linear-Gaussian conditionals, so on the non-Gaussian family they are the misspecified classical model.

NBN runs every applicable pair of a mechanism and an engine, with 2048 particles for the sampling engines (16 in the batching scenario, which measures time only). To keep the figures readable, each panel shows all applicable external baselines plus the best NBN methods of the family, selected by accuracy. Appendix B reports every method on every scenario. All runs use one server with a

20-core CPU, 126 GB of RAM and one GPU with 12 GB of memory; NBN and pomegranate run on the GPU, pgmpy on the CPU, and Pyro on the CPU except in the batching scenario.

4 Experimental Results

Section 2 makes two claims: an NBN keeps what a BN offers, while enhancing expressiveness of conditionals and scaling up inference performance. The results support both, within limits that the benchmark makes visible. In terms of our research questions:

- RQ1.** *No loss where the classical model is right* (Proposition 2.1). On discrete networks, exact inference in NBN returns the same TV as pgmpy at every size, and on real networks it matches or improves it on seven out of eight; learning is identical as well. The exception is inference on linear-Gaussian networks, where the closed form of pgmpy is more accurate than our sampled posterior (W1 0.039 against 0.074 at 1000 nodes).
- RQ2.** *Expressiveness pays where it is wrong.* On non-Gaussian networks the MDN has between one third and 60% of the W1 of a linear-Gaussian model, and achieves better held-out likelihood: where the outcome has several modes or a state-dependent spread, the linear-Gaussian family misstates the uncertainty while the MDN learns its shape.
- RQ3.** *The cost model holds on the GPU* (Proposition 2.3). With the sampling engines the time per query falls by two orders of magnitude between one query and a batch of 1024, down to tens of microseconds.

While pgmpy achieves top accuracy wherever one of its two families is the right model, it cannot represent non-Gaussian conditionals and answers one query at a time. pomegranate leverages the GPU, but its loopy belief propagation returns undefined posteriors or exceeds budget on all networks but the smallest. Pyro runs a generic sequential importance sampler, which the budget limits to 50 particles: it has the highest error on all synthetic families and times out from 500 nodes onwards. The performance of NBN can still be improved, and the benchmark shows where. The sampling engines pay a Monte-Carlo error where a closed form exists, the MDN exceeds the fit budget between 100 and 500 nodes, and the nearest-neighbour mechanism exhausts the GPU memory at 500.

Below, NBN methods are named by their mechanisms and engines: cat+VE is categorical table queried by variable elimination; likewise for linear-Gaussian (LG) and MDN mechanisms, and for likelihood weighting (LW) and amortized variational inference (AVI).

4.1 Inference accuracy (RQ1, RQ2)

Figure 3 shows the accuracy of the 128 queries for each family.

Discrete. cat+VE and pgmpy return the same TV at every size because they run exact inference on the same maximum-likelihood tables: this is Proposition 2.1, measured. Their common value is the floor of the reference (Section 3). The sampling engines of NBN stay within 0.01 of it, so on discrete networks approximation costs little.

Linear-Gaussian. This is the one family where a baseline is ahead: pgmpy conditions the joint Gaussian in closed form, while NBN samples. The gap grows with the size because, in deeper networks, fewer prior samples agree with the evidence: the median effective sample size falls from all 2048 particles at 10 nodes to 13% of

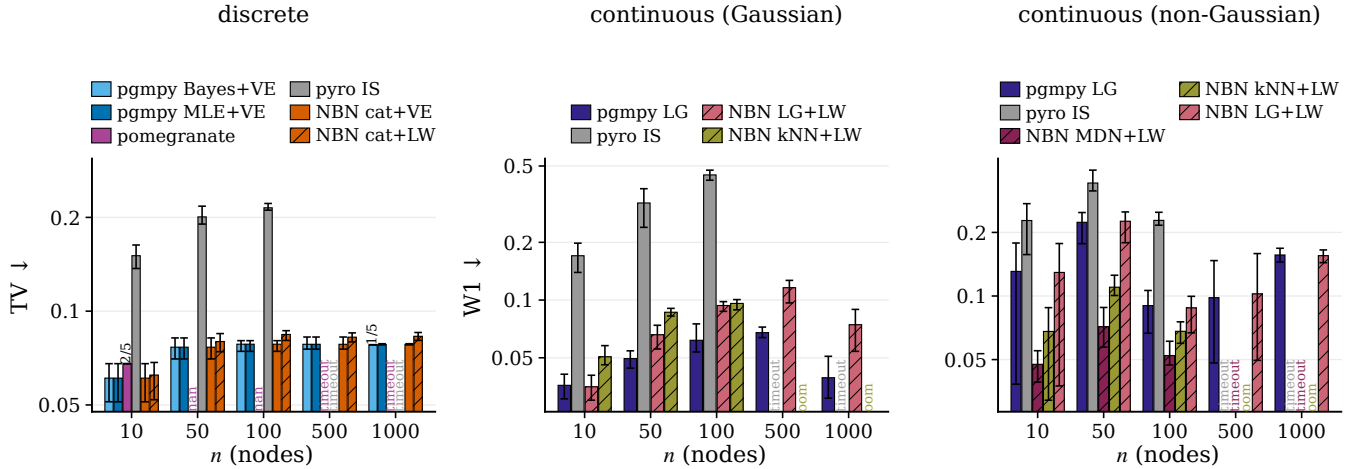


Figure 3: Inference accuracy against network size on synthetic networks (TV for discrete, W1 for continuous, lower is better). All applicable external baselines are shown, plus the selected NBN methods per family. A missing bar is a method that failed on every seed, with its status printed; f/n marks f failed seeds out of n .

them at 1000. The error is due to the engine, not the model, and a closed-form engine for all-Gaussian networks would remove it.

Non-Gaussian. Here the model is decisive: every linear-Gaussian method has the same error, in NBN or in pgmpy and whatever the engine, because the family cannot represent the data. The MDN has one third to 60% of that error and kNN one half to three quarters. The price is paid at fit time (Table 13 in Appendix B.1): the MDN exceeds the fit budget at 500 nodes, where kNN runs out of memory, so beyond 100 nodes only the misspecified models remain.

Table 1: Accuracy on the heaviest queries with full evidence against network size, up to 10 000 nodes (IQM over seeds, lower is better). (f/n) marks f failed seeds out of n . Bold: fewest failed seeds, then best value. Times in Appendix B.2.

Method	$n=10$	$n=100$	$n=1000$	$n=10000$
<i>Discrete, TV</i>				
pgmpy Bayes+VE	0.0491	0.0578	0.0645	0.0323 (1/5)
pgmpy MLE+VE	0.0438	0.0406	0.0582	0.0484
pomegranate	0.0276 (2/5)	nan	nan	timeout
Pyro IS	0.120	0.188	0.151	0.141 (1/5)
NBN cat+VE	0.0435	0.0476	0.0516	0.0376 (2/5)
NBN cat+LW	0.0495	0.0600	0.0619	0.0586
<i>Linear-Gaussian, W1</i>				
pgmpy LG	0.0303	0.0553	0.0378	0.0466
Pyro IS	0.138	0.330	0.338	0.226 (1/5)
NBN LG+LW	0.0291	0.0526	0.0652	0.0560
NBN MDN+LW	0.0326	0.0534	timeout	timeout
kNN+LW	0.0557	0.102	oom	oom
<i>Non-Gaussian, W1</i>				
pgmpy LG	0.115	0.142	0.0830	0.101
Pyro IS	0.232	0.193	0.175	0.209 (1/5)
NBN LG+LW	0.125	0.137	0.0917	0.0992
NBN MDN+LW	0.0504	0.0414 (1/5)	timeout	timeout
NBN kNN+LW	0.0755	0.0779	oom	oom

4.2 Scalability

Table 1 pushes the heaviest queries to 10 000 nodes. Accuracy stays in the same range at every size; what this scenario reveals is why each method stops and how long a query takes. Exact inference is bounded by the elimination itself. pgmpy completes every seed but needs tens of seconds per query from 1000 nodes; cat+VE answers in 0.2 to 0.4 s, two orders of magnitude faster, and exceeds the 12 GB of the GPU on two of five seeds at 10 000 nodes. The sampling engines have no such wall: cat+LW and LG+LW finish every seed at 10 000 nodes, where LG+LW is also faster than the closed form of pgmpy (2.6 s against 17.5 s per query), being the fastest up to 1000 nodes. The flexible mechanisms stop earlier due to the cost of learning: at 1000 nodes the MDN exceeds the fit budget and kNN the memory.

4.3 Batched queries (RQ3)

Proposition 2.3 postulates that the cost of a batch does not grow with its size until the device saturates, so the time per query should fall as $1/B$. We fix 50-node networks and ask each heaviest query for the evidence values of B different rows (Figure 4). The sampling engines follow the prediction closely up to $B=128$, where a query costs about one hundredth of a single one, and then flattens as the GPU saturates: at $B=1024$ the gain is 456× for cat+AVI and 227× for LG+AVI, down to 0.09 and 0.05 ms per query. Exact cat+VE is the fastest method on one query (2.8 ms) but gains only 4×: every factor of the elimination carries the batch axis, so its work grows with B and batching removes only the per-call overhead. Among the baselines only pomegranate batches, and at $B=1024$ it is 60 times slower than cat+AVI; pgmpy answers one query at a time. The gain is in the batch, not in the latency of one query: on a single continuous query the closed form of pgmpy takes 2.6 ms and NBN 12 ms. An agent that asks one question at a time is better served by the classical code; one that scores many candidate actions or particles at once is the case NBNs are built for.

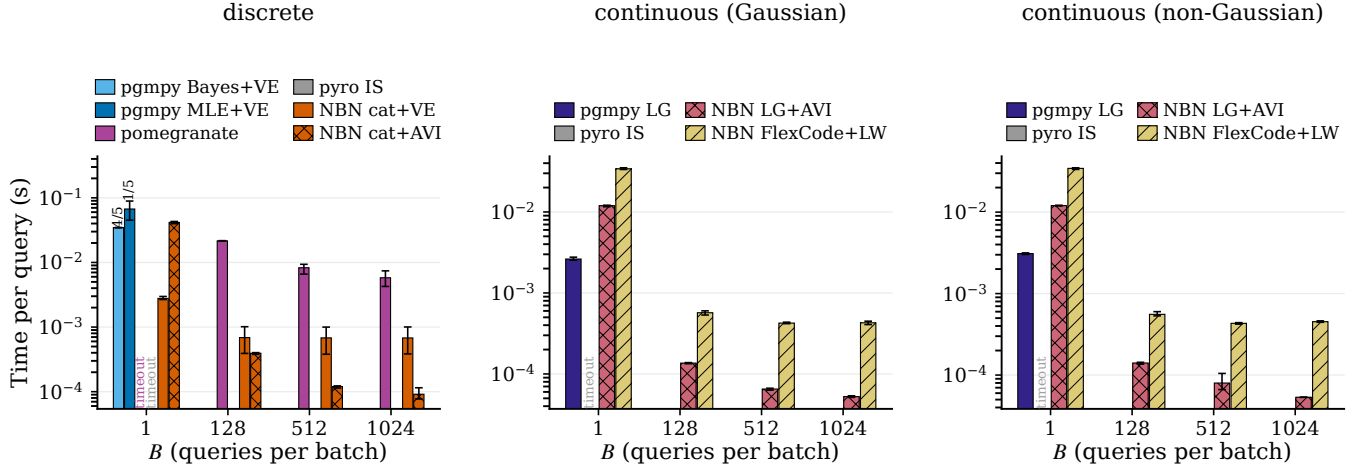


Figure 4: Time per query vs. batch size B (log scale) on 50-node networks, full evidence. Non-batching methods are at $B=1$ only.

4.4 Real networks

Table 2 repeats the accuracy comparison on thirteen bnlearn networks: eight discrete ones, from 8 to 1041 nodes, two Gaussian ones, and three conditional linear-Gaussian (CLG) ones, which mix discrete and continuous variables.

The conclusions of the synthetic scenarios carry over. On the discrete networks cat+VE has the lowest TV on five, including the two largest, and is within 0.001 of the best on two more; pgmpy is better on mildew only. Amortized inference suffers more here than on synthetic networks (up to four times the TV of exact inference), so the exact engine is the one to use whenever the treewidth allows it. On the Gaussian networks NBN is again close to the closed form of pgmpy. The CLG networks are where coverage matters: pgmpy and

pomegranate have no applicable engine, and the best NBN mechanism is below Pyro on all three. Which mechanism is best, however, depends on the network: the MDN wins on sangiovese and mehra and has six times the error of Pyro on healthcare, where kNN wins. The choice of the mechanism is a modelling decision, and the benchmark is the tool to make it.

4.5 Parameter learning and sample efficiency

The learning scenarios score the fitted conditionals with no query: by the parameter-recovery TV on discrete networks, the distance between learned and true rows of the CPTs, and by held-out negative log-likelihood (NLL) per node on continuous ones (Appendix A).

Table 2: Accuracy on bnlearn networks, grouped by kind and ordered by node count n (IQM over seeds; TV for discrete, W1 for Gaussian and CLG, lower is better). Each block names the method of each library; – denotes no applicable engine. pgmpy with Bayesian estimates and the other NBN engines are in Appendix B.4.

Network (n)	pgmpy	pomegr.	Pyro	NBN	NBN
<i>Discrete, TV</i>	<i>MLE+VE</i>	<i>BP</i>	<i>IS</i>	<i>cat+VE</i>	<i>cat+AVI</i>
asia (8)	0.00343	0.00380	0.0336	0.00354	0.00478
sachs (11)	0.0128	0.0122	0.0818	0.0120	0.0162
mildew (35)	0.0325	nan	0.287	0.0465	0.0703
barley (48)	0.0275	nan	0.215	0.0240	0.0401
hailfinder (56)	0.0203	0.0208	0.136	0.0206	0.0266
diabetes (413)	0.0393 (1/5)	nan	timeout	0.0343	0.128
link (724)	0.00197	timeout	timeout	0.00174	0.00612
munin (1041)	0.00464	timeout	timeout	0.00401	0.0105
<i>Gaussian, W1</i>	<i>LG</i>	–	<i>IS</i>	<i>LG+AVI</i>	<i>kNN+LW</i>
ecoli70 (46)	0.0418	–	0.245	0.0541	0.111
arth150 (107)	0.0210	–	0.107	0.0212	0.0418
<i>CLG, W1</i>	–	–	<i>IS</i>	<i>MDN+LW</i>	<i>kNN+LW</i>
healthcare (7)	–	–	106	625	23.4
sangiovese (15)	–	–	0.0583	0.0242	0.0280
mehra (24)	–	–	$3.74 \cdot 10^7$	$1.89 \cdot 10^6$	$2.41 \cdot 10^6$

Table 3: Sample efficiency on 50-node networks against the number of training rows n_{tr} (IQM over seeds, lower is better): parameter-recovery TV on discrete networks, held-out NLL per node on continuous ones. *timeout*: the 1200 s fit budget was exceeded on every seed. Fit times and the network-size sweep are in Appendices B.5 and B.6.

Method	$n_{tr}=4096$	$n_{tr}=16384$	$n_{tr}=32768$	$n_{tr}=65536$
<i>Discrete, parameter-recovery TV</i>				
pgmpy Bayes	0.0977	0.0474	0.0335	0.0236
pgmpy MLE	0.0979	0.0474	0.0336	0.0236
pomegranate	0.0979	0.0474	0.0336	0.0236
Pyro	0.0847	0.0453	0.0328	0.0233
NBN cat	0.0979	0.0474	0.0336	0.0236
NBN cat (Bayes)	0.0847	0.0453	0.0328	0.0233
<i>Gaussian, NLL per node</i>				
pgmpy LG	0.943	0.942	0.944	0.945
NBN LG	0.935 (1/5)	0.942	0.944	0.945
NBN MDN	0.953	0.946	0.958 (1/5)	timeout
NBN kNN	1.01	0.988	0.981	0.966 (1/5)
<i>Non-Gaussian, NLL per node</i>				
pgmpy LG	1.26	1.27	1.27	1.27
NBN LG	1.26	1.27	1.27	1.27
NBN MDN	1.20	1.19	1.19	timeout
NBN kNN	1.25	1.23	1.22	1.21

When the model is correct, NBN learns what competitors do, as the estimator is the same: the tabular mechanism equals the max-likelihood tables of pgmpy and pomegranate at all sizes, its smoothed variant equals the tables of Pyro, and the linear-Gaussian mechanism equals the least squares of pgmpy (Table 3; Appendix B.5). It is also faster: a 1000-node network is fitted in 2.1 s (tables) and 2.6 s (linear-Gaussian), against 9.8 s and 4.0 s for pgmpy. The accuracy drop when data is Gaussian is negligible: the NLL of the MDN is within 0.004 of the exact estimator.

When the model is wrong, more data does not repair it (Table 3). On 50-node non-Gaussian networks the NLL of the linear-Gaussian models stays at 1.26–1.27 from 4096 to 65 536 rows, while the MDN is at 1.20 with 4096 rows already and kNN improves steadily to 1.21. Flexibility is thus sample-efficient but not compute-efficient: the fit time of the MDN grows linearly with the rows and nodes, exceeding the budget at 65 536 rows and from 100 nodes (Appendix B.6).

5 Related Works

NBNs extend classical BNs in two directions, the *expressiveness* of the conditionals and the efficiency and *scale* of learning and inference; we review the works that pursue each of them.

Expressiveness. Kernel density estimation was used for continuous conditionals in Bayesian classifiers [28], and non-parametric BNs based on kernels or copulas followed [21, 24, 38]. Semiparametric BNs [4, 5] mix linear-Gaussian and kernel conditionals in one network; surveys of inference with continuous variables are [39, 48]. Neural structural causal models replace each structural equation with a neural density, typically a normalizing flow [17, 26, 31, 43, 56]. Closest to ours in name, the Neural Bayesian Network Understudy [46] trains a single Neural network to approximate the conditional probabilities of a given BN, with its independences encoded through training. These works fix one family of conditionals and one inference procedure. At the other extreme, deep generative models such as normalizing flows [41, 47] and neural world models [20] are expressive but hide the dependencies in the weights of one network: there is no local conditional to inspect or replace, and the posterior given arbitrary evidence is not directly available. An NBN changes the representation rather than adding a family. With tensor-valued mechanisms the BN is kept exactly (Proposition 2.1); tabular, linear-Gaussian, neural and non-parametric [25] mechanisms can be mixed in one graph and trained by gradient, also jointly with a downstream objective (Proposition 2.2); and any mechanism that can sample and evaluate a density is queried by the same batched engine, with no family-specific algorithm (Proposition 2.3).

Efficiency and scalability. Exact inference has been parallelized on multi-core CPUs [27] and on GPUs [58], and written as tensor contractions [40], but only for discrete networks and at a cost that stays exponential in the treewidth. Probabilistic circuits [12, 45] keep inference tractable and batched, but trade the per-variable conditionals for latent sum and product units. When exact inference is too expensive, a BN is queried by sampling. Likelihood weighting (Section 2.3) is simple and general, but its samples are drawn without looking at the evidence, so most of them are wasted when the evidence is unlikely. Amortized inference [18, 36] trains a neural network to propose samples that already agree with the

Table 4: Bayesian-network and probabilistic-modeling libraries. Disc.: discrete; LG: linear-Gaussian; NG: continuous non-Gaussian. Batch: the API accepts many queries in one call. GPU: BN queries run in parallel on the accelerator. Diff.: differentiable conditionals. do: native interventions. ✓ supported, • partial, ✗ absent.

Library	Families			Compute		Capabilities	
	Disc.	LG	NG	Batch	GPU	Diff.	do
pgmpy [3]	✓	✓	✗	✓	✗	✗	•
pyAgrum [14]	✓	✗	✗	✓	✗	✗	✗
bnlearn [50]	✓	✓	✗	✓	✗	✗	✗
CausalNex [6]	✓	✗	✗	✓	✗	✗	✓
pomegranate [49]	✓	✓	•	✓	•	✗	✗
PyMC [1]	✓	✓	✓	•	✓	•	✗
Pyro [7]	✓	✓	✓	✓	✓	✓	•
DoWhy [52]	✓	✓	•	✗	✗	✗	✓
NBN (ours)	✓	✓	✓	✓	✓	✓	✓

evidence and reuses it for every later query. We do not propose a new estimator: NBNs change how these estimators are executed. All the nodes at the same depth of the graph are evaluated in one tensor operation over every particle and every evidence assignment, so the cost of a query stops growing with the number of samples and of queries (Proposition 2.3).

Libraries. Table 4 summarizes NBNs competitors. Classical BN libraries (pgmpy [3], pyAgrum [14], bnlearn [50], CausalNex [6]) implement tabular or linear-Gaussian conditionals with exact inference; they run on the CPU and accept several queries in one call but execute them one by one, which are the limitations (L1) and (L2) of Section 2.1. pomegranate [49] is PyTorch-based and runs on the GPU, but its BN inference is limited to discrete networks and to loopy belief propagation, which is not guaranteed to converge on graphs with loops. Probabilistic programming systems (Pyro [7], PyMC [1]) support any distribution and GPU execution, but they do not expose the BN as an object: the graph, the per-node conditionals, and the inference procedure over them have to be written by hand for every model, and a generic sampler cannot eliminate variables exactly, prune the nodes that a query does not need, or evaluate a level of the graph at once. DoWhy [9, 52] estimates the causal effect of a treatment on an outcome and does not answer posterior queries over arbitrary variables.

To the best of our knowledge, no existing library combines an explicit DAG with differentiable conditionals of arbitrary family and GPU-batched learning and inference, which is the gap NBN fills. Our benchmark compares against one representative per class: pgmpy for the classical libraries, pomegranate as the PyTorch-based one, and Pyro as the probabilistic programming system.

6 Conclusions and Future Works

Neural Bayesian Networks (NBNs) are an extension to Bayesian networks that combine explicit graphical structure with learnable, tensor-valued conditional mechanisms. NBNs preserve the probabilistic factorization and modularity of BNs while bringing heterogeneous conditional models into a common framework for learning and GPU-batched inference. The accompanying PyTorch

library and reproducible benchmark provide a basis for evaluating the trade-offs between expressiveness, accuracy, and computational cost. NBNs provide to autonomous agents a tool for decision-making under uncertainty that combines structured world models with the flexibility and computational tools of modern machine learning. Explicit dependencies and separately replaceable mechanisms support adaptation of individual model components, while batched inference serve agents that evaluate many candidate actions or hypotheses. Evaluation of NBNs within agent decision making tasks (e.g. causal reinforcement learning), removing the assumption of having the graph available, evaluating sampling, and considering CPU execution along GPU are amongst the future works we are considering.

References

- [1] Oriol Abril-Pla, Virgile Andreani, Colin Carroll, Larry Dong, Christopher J Fonesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C Luhmann, Osvaldo A Martin, et al. 2023. PyMC: a modern, and comprehensive probabilistic programming framework in Python. *PeerJ Computer Science* 9 (2023), e1516.
- [2] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. 2021. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems* 34 (2021), 29304–29320.
- [3] Ankur Ankan and Johannes Textor. 2024. pgmpy: A python toolkit for bayesian networks. *Journal of Machine Learning Research* 25, 265 (2024), 1–8.
- [4] David Atienza, Concha Bielza, and Pedro Larrañaga. 2022. Semiparametric Bayesian Networks. *Information Sciences* 584 (2022), 564–582. doi:10.1016/j.ins.2021.10.074
- [5] David Atienza, Pedro Larrañaga, and Concha Bielza. 2022. Hybrid Semiparametric Bayesian Networks. In *TEST*, Vol. 31. 299–327.
- [6] Paul Beaumont, Ben Horsburgh, Philip Pilgerstorfer, Angel Droth, Richard Oentaryo, Steven Ler, Hiep Nguyen, Gabriel Azevedo Ferreira, Zain Patel, and Wesley Leong. 2021. CausalNex. URL <https://github.com/quantumblacklab/causalnex> (2021).
- [7] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *Journal of machine learning research* 20, 28 (2019), 1–6.
- [8] Christopher M. Bishop. 1994. *Mixture Density Networks*. Technical Report NCRG/94/004. Aston University.
- [9] Patrick Blöbaum, Peter Götz, Kailash Budhathoki, Atalanti A Mastakouri, and Dominik Janzing. 2024. DoWhy-GCM: An extension of DoWhy for causal inference in graphical causal models. *Journal of Machine Learning Research* 25, 147 (2024), 1–7.
- [10] Craig Boutilier, Richard Dearden, Moisés Goldszmidt, et al. 1995. Exploiting structure in policy construction. In *IJCAI*, Vol. 14. 1104–1113.
- [11] John S. Breese and David Heckerman. 1996. Decision-Theoretic Troubleshooting: A Framework for Repair and Experiment. In *UAI '96: Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence, Reed College, Portland, Oregon, USA, August 1-4, 1996*, Eric Horvitz and Finn Verner Jensen (Eds.). Morgan Kaufmann, 124–132. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=360&proceeding_id=12
- [12] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. 2020. *Probabilistic Circuits: A Unifying Framework for Tractable Probabilistic Models*. Technical Report. University of California, Los Angeles. <http://starai.cs.ucla.edu/papers/ProbCirc20.pdf>
- [13] Gregory F. Cooper. 1990. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artif. Intell.* 42, 2-3 (1990), 393–405. doi:10.1016/0004-3702(90)90060-D
- [14] Gaspard Ducamp, Christophe Gonzales, and Pierre-Henri Wuillemin. 2020. aGrUM/pyAgum: a toolbox to build models and algorithms for Probabilistic Graphical Models in Python. In *International Conference on Probabilistic Graphical Models*. PMLR.
- [15] Robert Fung and Kuo-Chu Chang. 1989. Weighing and Integrating Evidence for Stochastic Simulation in Bayesian Networks. In *Proceedings of the 5th Conference on Uncertainty in Artificial Intelligence (UAI)*. 209–219.
- [16] Peyré Gabriel and Cuturi Marco. 2019. Computational optimal transport with applications to data sciences. *Foundations and trends® in machine learning* 11, 5-6 (2019), 355–607.
- [17] Tomas Geffner, Javier Antorán, Adam Foster, Wenbo Gong, Chao Ma, Emre Kiciman, Amit Sharma, Angus Lamb, Martin Kukla, Nick Pawlowski, Miltiadis Allamanis, and Cheng Zhang. 2022. Deep End-to-end Causal Inference. *arXiv preprint arXiv:2202.02195* (2022).
- [18] Samuel J. Gershman and Noah D. Goodman. 2014. Amortized Inference in Probabilistic Reasoning. *Proceedings of the Annual Meeting of the Cognitive Science Society* 36 (2014).
- [19] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. 2003. Efficient Solution Algorithms for Factored MDPs. *Journal of Artificial Intelligence Research* 19 (2003), 399–468.
- [20] David Ha and Jürgen Schmidhuber. 2018. World models. *arXiv preprint arXiv:1803.10122* 2, 3 (2018).
- [21] Anca Hanea, Oswaldo Morales Napoles, and Daniel Ababei. 2015. Non-Parametric Bayesian Networks: Improving Theory and Reviewing Applications. *Reliability Engineering & System Safety* 144 (2015), 265–284.
- [22] David Heckerman. 1998. A tutorial on learning with Bayesian networks. *Learning in graphical models* (1998), 301–354.
- [23] Arthur E. Hoerl and Robert W. Kennard. 1970. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics* 12, 1 (1970), 55–67.
- [24] Katja Ickstadt, Björn Bornkamp, Marco Grzegorzczak, Jakob Wiecek, M. Rashid Sheriff, Hernán E. Grecco, and Eli Zamir. 2010. Nonparametric Bayesian Networks. *Bayesian Statistics* 9 (2010).
- [25] Rafael Izbicki and Ann B. Lee. 2017. Converting High-Dimensional Regression to High-Dimensional Conditional Density Estimation. *Electronic Journal of Statistics* 11, 2 (2017), 2800–2831.
- [26] Adrián Javaloy, Pablo Sánchez-Martin, and Isabel Valera. 2023. Causal Normalizing Flows: From Theory to Practice. In *Advances in Neural Information Processing Systems*, Vol. 36.
- [27] Jiantong Jiang, Zeyi Wen, Atif Mansoor, and Ajmal Mian. 2023. POSTER: Fast Parallel Exact Inference on Bayesian Networks. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. doi:10.1145/3572848.3577476
- [28] George H. John and Pat Langley. 1995. Estimating Continuous Distributions in Bayesian Classifiers. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*. 338–345.
- [29] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. 1998. Planning and acting in partially observable stochastic domains. *Artificial intelligence* 101, 1-2 (1998), 99–134.
- [30] Michael Kearns and Daphne Koller. 1999. Efficient Reinforcement Learning in Factored MDPs. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*. 740–747.
- [31] Ilyes Khemakhem, Ricardo Monti, Robert Leech, and Aapo Hyvärinen. 2021. Causal Autoregressive Flows. In *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS)*. 3520–3528.
- [32] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations (ICLR)*.
- [33] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
- [34] Evangelia Kyrimi, Scott McLachlan, Kudakwashe Dube, Mariana R. Neves, Ali Fahmi, and Norman Fenton. 2021. A Comprehensive Scoping Review of Bayesian Networks in Healthcare: Past, Present and Future. *Artificial Intelligence in Medicine* 117 (2021), 102108.
- [35] Steffen L. Lauritzen and Nanny Wermuth. 1989. Graphical Models for Associations between Variables, some of which are Qualitative and some Quantitative. *The Annals of Statistics* 17, 1 (1989), 31–57.
- [36] Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. 2017. Inference Compilation and Universal Probabilistic Programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*. 1338–1348.
- [37] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. 2020. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *arXiv preprint arXiv:2005.01643* (2020).
- [38] Dimitris Margaritis. 2005. Distribution-Free Learning of Bayesian Network Structure in Continuous Domains. *Proceedings of the AAAI Conference on Artificial Intelligence* (2005).
- [39] Joaquín Mori, Josep Maria Mateo-Sanz, and Antonio Salmerón. 2016. Inference in Hybrid Bayesian Networks: A Survey. *International Journal of Approximate Reasoning* (2016).
- [40] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. 2019. Tensor Variable Elimination for Plated Factor Graphs. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*.
- [41] George Papamakarios, Theo Pavlakou, and Iain Murray. 2017. Masked Autoregressive Flow for Density Estimation. In *Advances in Neural Information Processing Systems*, Vol. 30.
- [42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (2019).
- [43] Nick Pawlowski, Daniel C. Castro, and Ben Glocker. 2020. Deep Structural Causal Models for Tractable Counterfactual Inference. In *Advances in Neural Information Processing Systems*, Vol. 33.

- [44] Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.
- [45] Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. 2020. Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits. In *Proceedings of the 37th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 119)*. 7563–7574.
- [46] Paloma Rabacay, Cedric De Boom, and Thomas Demeester. 2022. Neural Bayesian Network Understudy. *arXiv preprint arXiv:2211.08243* (2022).
- [47] Danilo Jimenez Rezende and Shakir Mohamed. 2015. Variational Inference with Normalizing Flows. In *Proceedings of the 32nd International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 37)*. 1530–1538.
- [48] Antonio Salmerón, Rafael Rumi, Helge Langseth, Thomas D. Nielsen, and Anders L. Madsen. 2018. A Review of Inference Algorithms for Hybrid Bayesian Networks. *Journal of Artificial Intelligence Research* 62 (2018), 799–828.
- [49] Jacob Schreiber. 2018. Pomegranate: fast and flexible probabilistic modeling in python. *Journal of Machine Learning Research* 18, 164 (2018), 1–6.
- [50] Marco Scutari. 2010. Learning Bayesian Networks with the bnlearn R Package. *Journal of Statistical Software* 35, 3 (2010), 1–22.
- [51] Ross D. Shachter and Mark A. Peot. 1989. Simulation Approaches to General Probabilistic Inference on Belief Networks. In *Proceedings of the 5th Conference on Uncertainty in Artificial Intelligence (UAI)*. 221–231.
- [52] Amit Sharma and Emre Kiciman. 2020. DoWhy: An end-to-end library for causal inference. *arXiv preprint arXiv:2011.04216* (2020).
- [53] Charles J. Stone. 1977. Consistent Nonparametric Regression. *The Annals of Statistics* 5, 4 (1977), 595–620.
- [54] Alexandre B. Tsybakov. 2009. *Introduction to Nonparametric Estimation*. Springer. doi:10.1007/B13794
- [55] Philippe Weber, Gabriela Medina-Oliva, Christophe Simon, and Benoît Iung. 2012. Overview on Bayesian Networks Applications for Dependability, Risk Analysis and Maintenance Areas. *Engineering Applications of Artificial Intelligence* 25, 4 (2012), 671–682.
- [56] Kevin Xia, Kai-Zhan Lee, Yoshua Bengio, and Elias Bareinboim. 2021. The Causal-Neural Connection: Expressiveness, Learnability, and Inference. In *Advances in Neural Information Processing Systems*, Vol. 34.
- [57] Nevin Lianwen Zhang and David Poole. 1994. A Simple Approach to Bayesian Network Computations. In *Proceedings of the 10th Canadian Conference on Artificial Intelligence*. 171–178.
- [58] Lu Zheng, Ole Mengshoel, and Jike Chong. 2011. Belief Propagation by Message Passing in Junction Trees: Computing Each Message Faster Using GPU Parallelization. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI)*. 822–830.

A Benchmark Design – Extended Version

This appendix gives the settings of the benchmark that Section 3 leaves out. All of them are fields of the released configuration files or defaults of the released code, and can be changed.

Synthetic generator. The seed draws a DAG over n ordered nodes, where each earlier node becomes a parent of a later one with probability 0.2 and a node keeps at most four parents. The cap bounds the size of a conditional as n grows: with four values per variable, a CPT has at most $4^5 = 1024$ entries. In the *discrete* family every variable has four values and every row of a CPT is the softmax of standard-normal logits. In the *linear-Gaussian* family every node has weights of variance $1/|Pa|$, a standard-normal bias and a noise scale drawn uniformly in $[0.3, 1]$. In the *non-Gaussian* family every node is a three-component Gaussian mixture: the component means are linear in the parents, with weights of standard deviation $0.3/\sqrt{|Pa|}$ and offsets $0.3(z_1 + 0.7|z_2|)$ for standard-normal z_1, z_2 , which are skewed; the component scales are uniform in $[0.3, 0.7]$; and the mixing weights are drawn from a flat Dirichlet distribution and do not depend on the parents.

Data. The training set has 20 480 rows in the accuracy, scalability and learning scenarios and 4096 rows in the batching scenario; the sample-efficiency study uses 4096, 16 384, 32 768 and 65 536 rows. On real networks it has 10 240, 20 480, 40 960 or 81 920 rows, for networks with up to 10^3 , 10^4 , 10^5 or more parameters. The test set has 4096 rows (1024 in the batching scenario and on real networks). The five seeds are 0 to 4.

Query selectors. *Random:* one target and its evidence variables are drawn uniformly for the problem, and the query is asked for the evidence values of 128 test rows. *Topological* (128 queries per cell): targets are allocated by structural role, 30% to hubs, 30% to cut nodes, 30% to terminals and 10% at random. Hubs are ranked by the size of their Markov blanket, that is, the set of their parents, their children and the other parents of their children. Cut nodes are the articulation points of the moral graph: the undirected graph obtained by linking the parents of every node and dropping the direction of the edges, which a cut node disconnects when it is removed. Terminals are ranked by their depth minus their number of descendants. Hubs, cut nodes and random targets are asked in the prediction or in the diagnosis direction with equal probability, and terminals in the prediction direction only. The evidence is taken along the longest path (40% of the queries), inside the Markov blanket (40%) or at random (20%). *Heaviest:* the top-ranked hub, cut node and terminal, each in the prediction and in the diagnosis direction and with full and with empty evidence, for at most twelve queries per cell (eight when the graph has no cut node); the evidence is taken along the longest path for hubs and terminals and inside the Markov blanket for cut nodes. In the batching scenario each of these queries is asked for the evidence values of 1024 rows.

Reference. For a discrete target the reference set has 5000 samples on synthetic networks and as many samples as the training set on real ones; a query whose evidence agrees with fewer than ten reference samples is not scored. For a continuous target the reference is a sample of 2000 draws from $p(x_Q | x_E)$, obtained by likelihood weighting on the true model. Both references are finite samples, which gives the floor of Section 3. For a discrete target it comes from the few reference samples that agree with the evidence, so it varies with the network and with the query. The floor is the average error of an exact answer against the sampled reference and not a bound: an approximate method can fall slightly below the exact engines.

Learning metrics. The learning scenarios score the fitted conditionals directly, without queries. On discrete networks the metric is the parameter-recovery TV: the total variation between a learned and a true row of a CPT, averaged over the parent configurations with their true probabilities and then over the nodes. Continuous families have no common parameters to compare, since a linear-Gaussian model, an MDN and a kNN estimator do not share a parameter space, so we score the fitted density itself with the held-out negative log-likelihood (NLL) per node: minus the log-density of a test row under the fitted network, averaged over the test rows and divided by n . The NLL is a strictly proper scoring rule. Its expectation is the entropy of the true conditionals plus the Kullback–Leibler divergence from the true to the fitted model, so on the same problem the difference between two methods is a difference in divergence from the truth, and it applies to every mechanism that exposes a density. We prefer it to two alternatives. W1, the metric of the inference scenarios, compares samples of a posterior and therefore needs a query and an engine, which would mix the error of learning with the error of inference. The calibration error PIT-KS (the Kolmogorov–Smirnov distance between the probability integral transform of held-out values and the uniform distribution) needs no query, but calibration is necessary and not sufficient: a model that ignores the parents and returns the marginal of each node is perfectly calibrated. Our runs show the difference: on 50-node non-Gaussian networks with 4096 rows FlexCode has less than half the PIT-KS of the linear-Gaussian model (0.028 against 0.065, close to the 0.023 of the MDN) and the worst NLL of the three (1.42 against 1.26 and 1.20). PIT-KS is reported for every method in Appendix B.5.

Batches with empty evidence. With empty evidence (Appendix B.3) the B queries of a batch are the same marginal query, so NBN answers it once and the time per query is the time of one query divided by B ; we do not count this as a batching gain.

Particles. The sampling engines of NBN use 2048 particles in the accuracy, scalability and real-network scenarios and 16 in the batching scenario, which measures time only. Pyro uses 50 particles in every scenario.

Budgets and execution. The fit budget and the query budget are 1200 s each; the query budget is 600 s in the batching scenario. The memory cap of a cell is 80% of the memory that is free on the machine when the cell starts. Each cell runs in its own process, so that a timeout or an out-of-memory failure ends that cell only.

Reporting. In the figures the error bar of a method spans the smallest and the largest value that enters its interquartile mean, and the value axes are logarithmic. Each panel has its own legend; a method has the same colour in every figure, and for NBN the colour is the mechanism and the hatch is the engine. Each panel of the main text shows all applicable external baselines plus two NBN methods, the best parametric or neural one and the best non-parametric one (the two best on discrete networks), selected once per family on the accuracy metric of the scenario. On the continuous families of the accuracy and scalability scenarios the linear-Gaussian mechanism is shown as well, so that the NBN method that reaches the largest networks is always visible. The tables of Appendix B report the IQM $\pm \frac{1}{2}$ of the interquartile range.

Hardware. All runs use one server with an Intel Xeon E5-2640 v4 CPU at 2.40 GHz (20 cores), 126 GB of RAM and an NVIDIA Titan X (Pascal) GPU with 12 GB of memory.

B Complete results

This appendix reports every method on every scenario of Section 3, including the mechanisms and engines that the main text does not discuss (normalizing-flow, kernel-density and neural categorical conditionals, and the amortized importance-sampling engine AIS), and the fit and query times of the scenarios where the main text reports accuracy only. Methods carry the names of the configuration files, `library-model-engine`; for NBN the model is the mechanism of the conditionals. In the figures a bar is the interquartile mean (IQM) over the seeds a method solved and the error bar spans the smallest and the largest value that enters that mean; f/n above a bar means that f of n seeds failed, and a missing bar carries the failure status. The value axes are logarithmic. Each panel has its own legend; a method has the same colour in every figure, and for NBN the colour is the mechanism and the hatch is the engine. In the tables a cell is $\text{IQM} \pm \text{half the interquartile range}$, followed by (f/n) when f of the n seeds failed for any reason (time budget, memory, NaN output, error); a status alone (timeout, oom, nan) means that no seed was solved, and – that the method was not run in that configuration. In each column the best method is in bold: the one with the fewest failed seeds and, among those, the best IQM. In the learning scenarios the negative log-likelihood is the mean over the held-out rows, divided by the number of nodes.

B.1 Inference accuracy

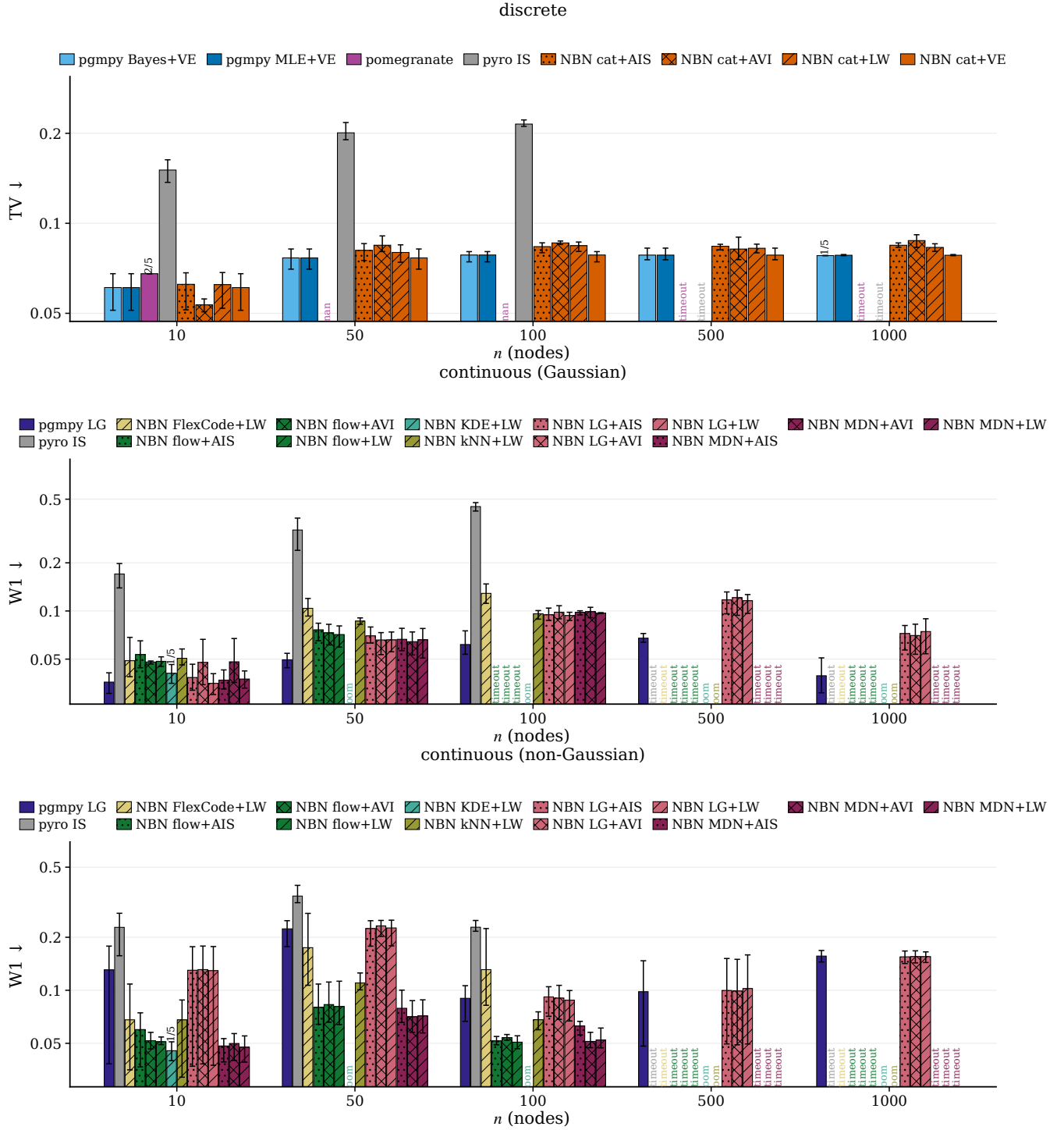


Figure 5: Inference accuracy, all methods: accuracy (TV on discrete targets, W1 on continuous ones) against network size.

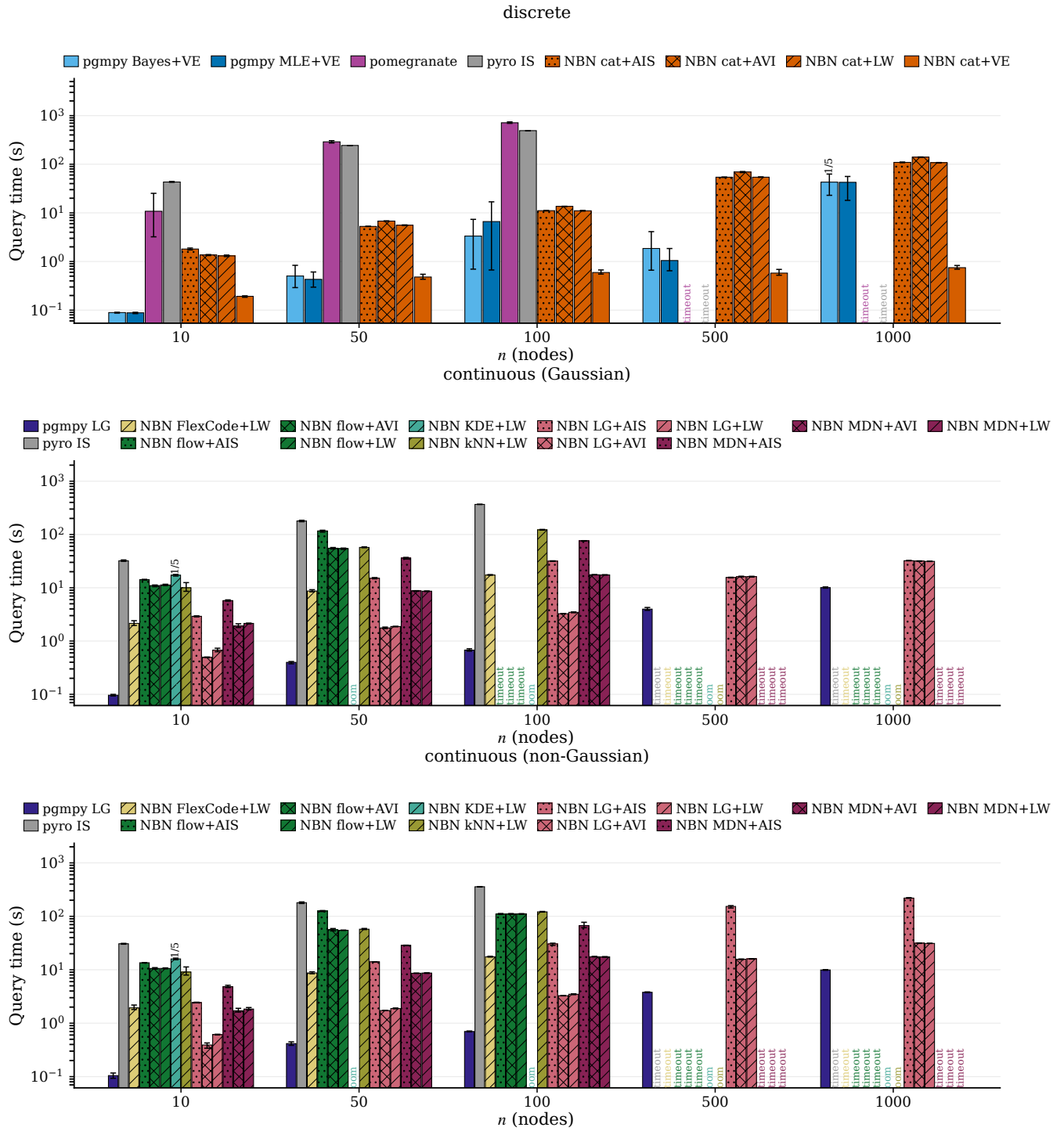


Figure 6: Inference accuracy, all methods: total query time (s) against network size.

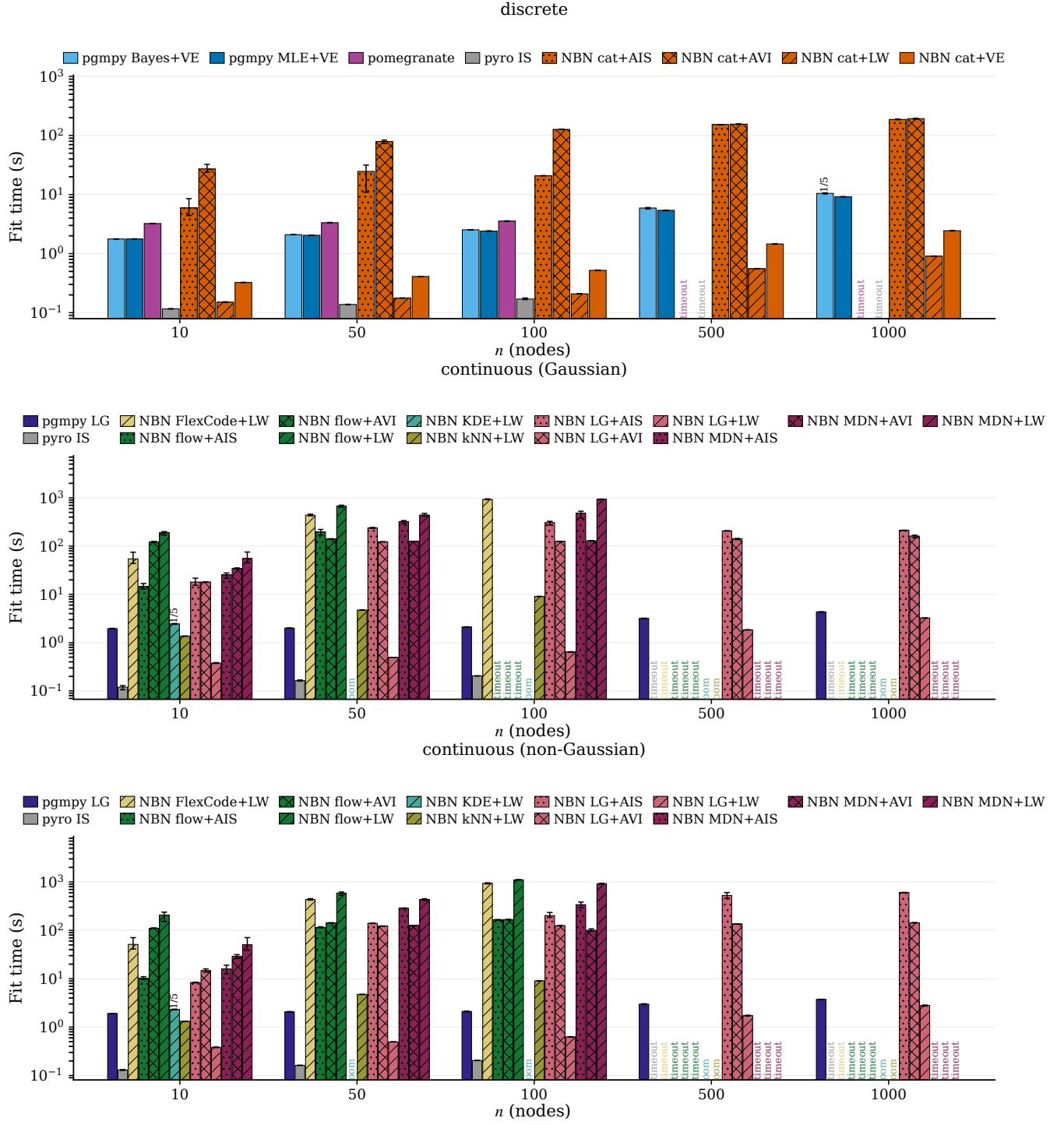


Figure 7: Inference accuracy, all methods: fit time (s) against network size.

Table 5: Inference accuracy, discrete networks: TV (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-bayes-ve	0.061±0.0083	0.0766±0.0059	0.0783±0.0031	0.0783±0.0035	0.078±0.0014 (1/5)
pgmpy-mle-ve	0.061±0.0083	0.0766±0.0059	0.0783±0.0031	0.0783±0.0035	0.0782±0.00035
pomegranate-discrete-ve	0.0678±0.0054 (2/5)	nan	nan	timeout	timeout
pyro-empirical-importance	0.151±0.013	0.201±0.013	0.215±0.0054	timeout	timeout
nbn-cat-ais	0.0625±0.0085	0.0812±0.0052	0.0836±0.0032	0.0838±0.0018	0.0845±0.0014
nbn-cat-avi	0.0534±0.0026	0.0845±0.0051	0.086±0.0012	0.082±0.0071	0.0875±0.0043
nbn-cat-lw	0.0624±0.0082	0.0799±0.0053	0.0842±0.003	0.0825±0.0027	0.0831±0.0024
nbn-cat-ve	0.061±0.0083	0.0766±0.0059	0.0783±0.0031	0.0783±0.0035	0.0782±0.00035

Table 6: Inference accuracy, discrete networks: total query time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-bayes-ve	0.0893±0.0018	0.504±0.27	3.33±3.3	1.85±1.7	42.9±39 (1/5)
pgmpy-mle-ve	0.0882±0.0028	0.43±0.16	6.63±8.1	1.05±0.6	42.5±19
pomegranate-discrete-ve	10.8±11	289±16	711±24	timeout	timeout
pyro-empirical-importance	43.1±0.81	243±0.31	490±3.4	timeout	timeout
nbn-cat-ais	1.81±0.081	5.27±0.026	11.1±0.13	54.2±0.54	109±1.7
nbn-cat-avi	1.36±0.031	6.78±0.055	13.6±0.075	69.3±1.8	140±0.92
nbn-cat-lw	1.31±0.046	5.59±0.066	11±0.18	54.2±0.57	108±0.79
nbn-cat-ve	0.193±0.0065	0.48±0.058	0.594±0.059	0.578±0.083	0.747±0.067

Table 7: Inference accuracy, discrete networks: fit time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-bayes-ve	1.77±0.012	2.09±0.0073	2.53±0.02	5.91±0.16	10.5±0.33 (1/5)
pgmpy-mle-ve	1.77±0.0069	2.04±0.0044	2.4±0.026	5.41±0.052	9.17±0.053
pomegranate-discrete-ve	3.23±0.024	3.34±0.026	3.56±0.042	timeout	timeout
pyro-empirical-importance	0.116±0.001	0.137±0.0011	0.17±0.0055	timeout	timeout
nbn-cat-ais	5.93±2.1	24.6±10	20.8±0.022	153±0.59	187±2.5
nbn-cat-avi	27.1±4.3	79.1±5	127±0.29	156±1.6	193±3.1
nbn-cat-lw	0.151±0.0015	0.177±0.00027	0.209±0.0014	0.556±0.0032	0.907±0.0044
nbn-cat-ve	0.324±0.0012	0.41±0.0018	0.521±0.0062	1.46±0.013	2.44±0.017

Table 8: Inference accuracy, linear-Gaussian networks: W1 (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg-predict	0.0359±0.0053	0.0495±0.0051	0.0617±0.011	0.0676±0.0043	0.0394±0.01
pyro-empirical-importance	0.17±0.029	0.321±0.071	0.449±0.027	timeout	timeout
nbn-flexcode-lw	0.0489±0.015	0.104±0.013	0.129±0.018	timeout	timeout
nbn-flow-ais	0.0534±0.01	0.0763±0.0093	timeout	timeout	timeout
nbn-flow-avi	0.048±0.001	0.0729±0.011	timeout	timeout	timeout
nbn-flow-lw	0.0485±0.0033	0.0712±0.011	timeout	timeout	timeout
nbn-kde-lw	0.0408±0.0087 (1/5)	oom	oom	oom	oom
nbn-knn-lw	0.0506±0.0059	0.0864±0.004	0.096±0.0058	oom	oom
nbn-lg-ais	0.0384±0.0073	0.0699±0.0082	0.0947±0.0085	0.117±0.018	0.0723±0.012
nbn-lg-avi	0.0477±0.016	0.0659±0.01	0.0983±0.0093	0.121±0.021	0.0703±0.015
nbn-lg-lw	0.0353±0.0053	0.0659±0.0092	0.0937±0.0054	0.116±0.015	0.0744±0.018
nbn-mdn-ais	0.037±0.0051	0.0665±0.011	0.0981±0.0029	timeout	timeout
nbn-mdn-avi	0.0481±0.016	0.0643±0.01	0.099±0.0073	timeout	timeout
nbn-mdn-lw	0.0376±0.0046	0.0661±0.013	0.0971±0.00068	timeout	timeout

Table 9: Inference accuracy, linear-Gaussian networks: total query time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg-predict	0.0963±0.0032	0.399±0.019	0.676±0.035	3.98±0.24	10.1±0.25
pyro-empirical-importance	32.1±0.98	179±5.5	367±2	timeout	timeout
nbn-flexcode-lw	2.15±0.22	8.69±0.4	17.4±0.25	timeout	timeout
nbn-flow-ais	14.2±0.21	117±4.7	timeout	timeout	timeout
nbn-flow-avi	11±0.4	55.3±1.7	timeout	timeout	timeout
nbn-flow-lw	11.3±0.32	54.3±1.5	timeout	timeout	timeout
nbn-kde-lw	17.2±2.2 (1/5)	oom	oom	oom	oom
nbn-knn-lw	10±2	57.2±1.1	122±1.8	oom	oom
nbn-lg-ais	2.92±0.064	15.1±0.39	31.7±0.31	15.6±0.2	32.2±0.23
nbn-lg-avi	0.495±0.0075	1.76±0.057	3.26±0.024	16.2±0.23	31.7±0.36
nbn-lg-lw	0.675±0.051	1.88±0.02	3.46±0.063	16.2±0.25	31.3±0.16
nbn-mdn-ais	5.75±0.16	36.1±1.3	76.1±0.71	timeout	timeout
nbn-mdn-avi	1.92±0.16	8.74±0.1	17.5±0.062	timeout	timeout
nbn-mdn-lw	2.15±0.034	8.64±0.11	17.4±0.16	timeout	timeout

Table 10: Inference accuracy, linear-Gaussian networks: fit time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg-predict	1.95±0.028	2±0.0094	2.12±0.016	3.18±0.026	4.34±0.04
pyro-empirical-importance	0.12±0.012	0.164±0.0047	0.205±0.00058	timeout	timeout
nbn-flexcode-lw	54.4±16	437±17	928±18	timeout	timeout
nbn-flow-ais	14.6±1.9	196±26	timeout	timeout	timeout
nbn-flow-avi	122±3.1	141±1.4	timeout	timeout	timeout
nbn-flow-lw	189±15	678±36	timeout	timeout	timeout
nbn-kde-lw	2.45±0.12 (1/5)	oom	oom	oom	oom
nbn-knn-lw	1.37±0.0097	4.73±0.013	9.06±0.022	oom	oom
nbn-lg-ais	18±3.1	238±5.1	305±27	207±1.1	212±1.2
nbn-lg-avi	18±0.19	123±0.055	125±0.69	142±3.7	162±9.7
nbn-lg-lw	0.377±0.0072	0.496±0.0013	0.644±0.0067	1.84±0.015	3.24±0.012
nbn-mdn-ais	25.4±3.2	322±21	479±73	timeout	timeout
nbn-mdn-avi	34.6±0.76	125±0.35	129±0.072	timeout	timeout
nbn-mdn-lw	55.3±15	441±28	930±11	timeout	timeout

Table 11: Inference accuracy, non-Gaussian networks: W1 (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg-predict	0.131±0.07	0.223±0.036	0.0901±0.02	0.0982±0.049	0.156±0.012
pyro-empirical-importance	0.228±0.058	0.343±0.04	0.228±0.016	timeout	timeout
nbn-flexcode-lw	0.068±0.037	0.175±0.083	0.131±0.071	timeout	timeout
nbn-flow-ais	0.0599±0.019	0.0801±0.022	0.0516±0.0028	timeout	timeout
nbn-flow-avi	0.0517±0.0056	0.083±0.023	0.0539±0.002	timeout	timeout
nbn-flow-lw	0.051±0.0026	0.0809±0.024	0.0508±0.0043	timeout	timeout
nbn-kde-lw	0.0454±0.0079 (1/5)	oom	oom	oom	oom
nbn-knn-lw	0.068±0.028	0.11±0.012	0.0681±0.0079	oom	oom
nbn-lg-ais	0.13±0.07	0.224±0.035	0.0917±0.017	0.0998±0.051	0.155±0.013
nbn-lg-avi	0.131±0.07	0.232±0.024	0.0903±0.019	0.0991±0.05	0.155±0.012
nbn-lg-lw	0.129±0.07	0.226±0.036	0.0878±0.016	0.102±0.055	0.155±0.011
nbn-mdn-ais	0.0483±0.007	0.0791±0.017	0.0627±0.0054	timeout	timeout
nbn-mdn-avi	0.0499±0.0084	0.0707±0.015	0.0513±0.0053	timeout	timeout
nbn-mdn-lw	0.0475±0.008	0.0715±0.016	0.0522±0.007	timeout	timeout

Table 12: Inference accuracy, non-Gaussian networks: total query time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg-predict	0.105±0.012	0.416±0.031	0.7±0.013	3.79±0.061	9.92±0.14
pyro-empirical-importance	30.7±0.55	180±6	358±3.2	timeout	timeout
nbn-flexcode-lw	1.93±0.19	8.72±0.37	17.4±0.33	timeout	timeout
nbn-flow-ais	13.5±0.14	126±2.6	110±2.1	timeout	timeout
nbn-flow-avi	10.6±0.34	56±2.9	110±1.6	timeout	timeout
nbn-flow-lw	10.6±0.22	54.8±0.25	111±1.5	timeout	timeout
nbn-kde-lw	16±2.1 (1/5)	oom	oom	oom	oom
nbn-knn-lw	9.07±1.7	57.4±1.4	121±1.5	oom	oom
nbn-lg-ais	2.44±0.029	14.1±0.027	30.5±1.4	153±7.7	220±3.1
nbn-lg-avi	0.391±0.044	1.73±0.0097	3.27±0.022	15.7±0.2	31.5±0.28
nbn-lg-lw	0.612±0.0093	1.88±0.033	3.48±0.063	16±0.14	31.1±0.0079
nbn-mdn-ais	4.82±0.24	28.6±0.22	67±8.1	timeout	timeout
nbn-mdn-avi	1.71±0.14	8.61±0.032	17.5±0.22	timeout	timeout
nbn-mdn-lw	1.84±0.098	8.69±0.099	17.4±0.23	timeout	timeout

Table 13: Inference accuracy, non-Gaussian networks: fit time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg-predict	1.92±0.0021	2.07±0.027	2.11±0.025	2.98±0.063	3.75±0.021
pyro-empirical-importance	0.131±0.0025	0.162±0.00045	0.204±0.0004	timeout	timeout
nbn-flexcode-lw	51.2±15	433±15	932±24	timeout	timeout
nbn-flow-ais	10.2±0.7	115±2.7	166±3	timeout	timeout
nbn-flow-avi	111±1.2	142±2.1	166±2.1	timeout	timeout
nbn-flow-lw	207±43	588±44	1.11e+03±15	timeout	timeout
nbn-kde-lw	2.31±0.11 (1/5)	oom	oom	oom	oom
nbn-knn-lw	1.32±0.0052	4.74±0.0095	9.02±0.015	oom	oom
nbn-lg-ais	8.35±0.14	140±0.58	202±24	525±73	606±0.84
nbn-lg-avi	14.7±1.2	123±0.025	125±0.73	136±1	143±1.3
nbn-lg-lw	0.383±0.0055	0.495±0.0022	0.627±0.0054	1.73±0.032	2.79±0.068
nbn-mdn-ais	15.8±3.2	288±0.45	339±45	timeout	timeout
nbn-mdn-avi	29.1±2.6	126±0.64	100±5.4	timeout	timeout
nbn-mdn-lw	50.1±16	431±16	920±19	timeout	timeout

B.2 Scalability

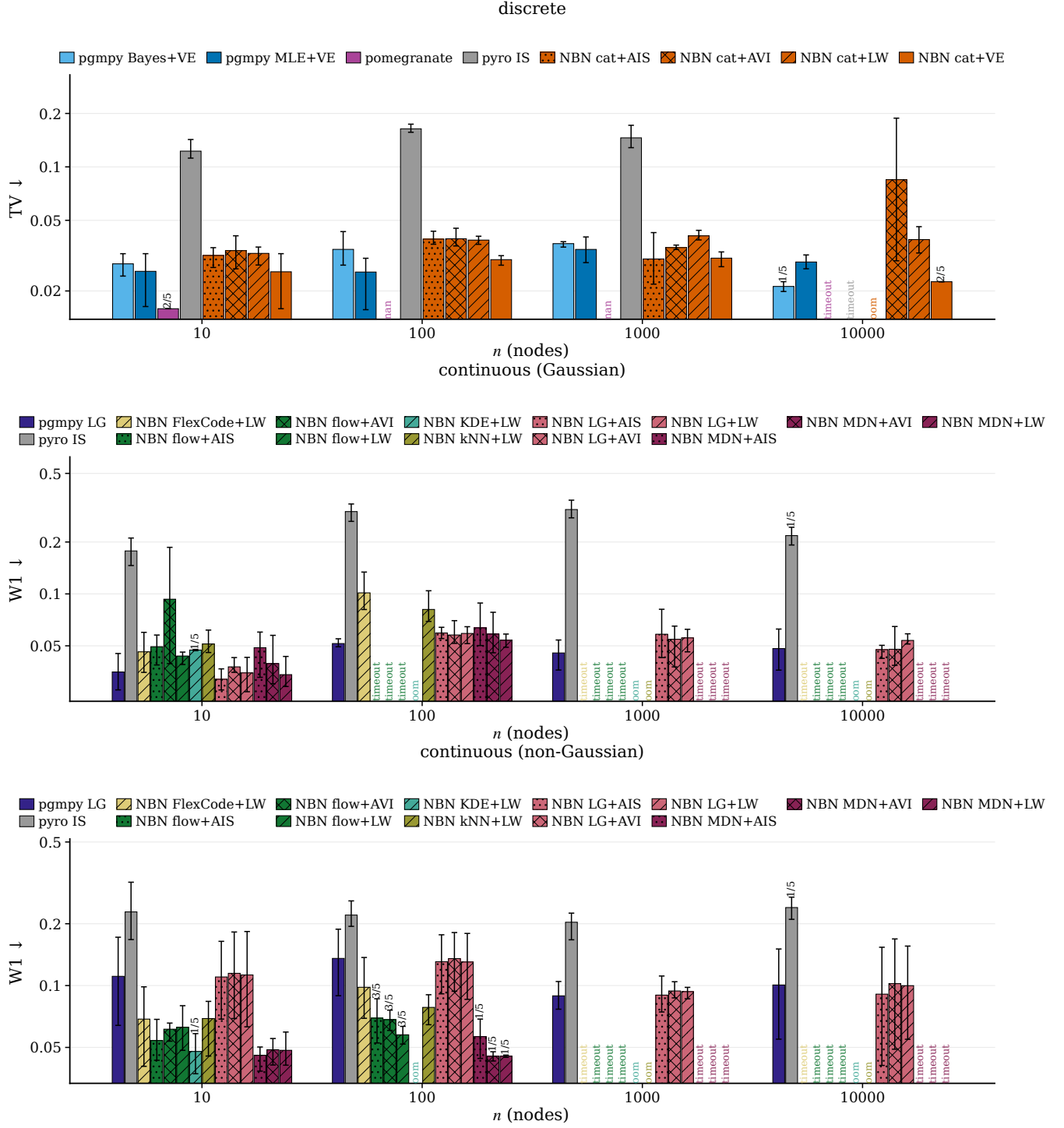


Figure 8: Scalability, all methods: accuracy (TV on discrete targets, W1 on continuous ones) against network size.

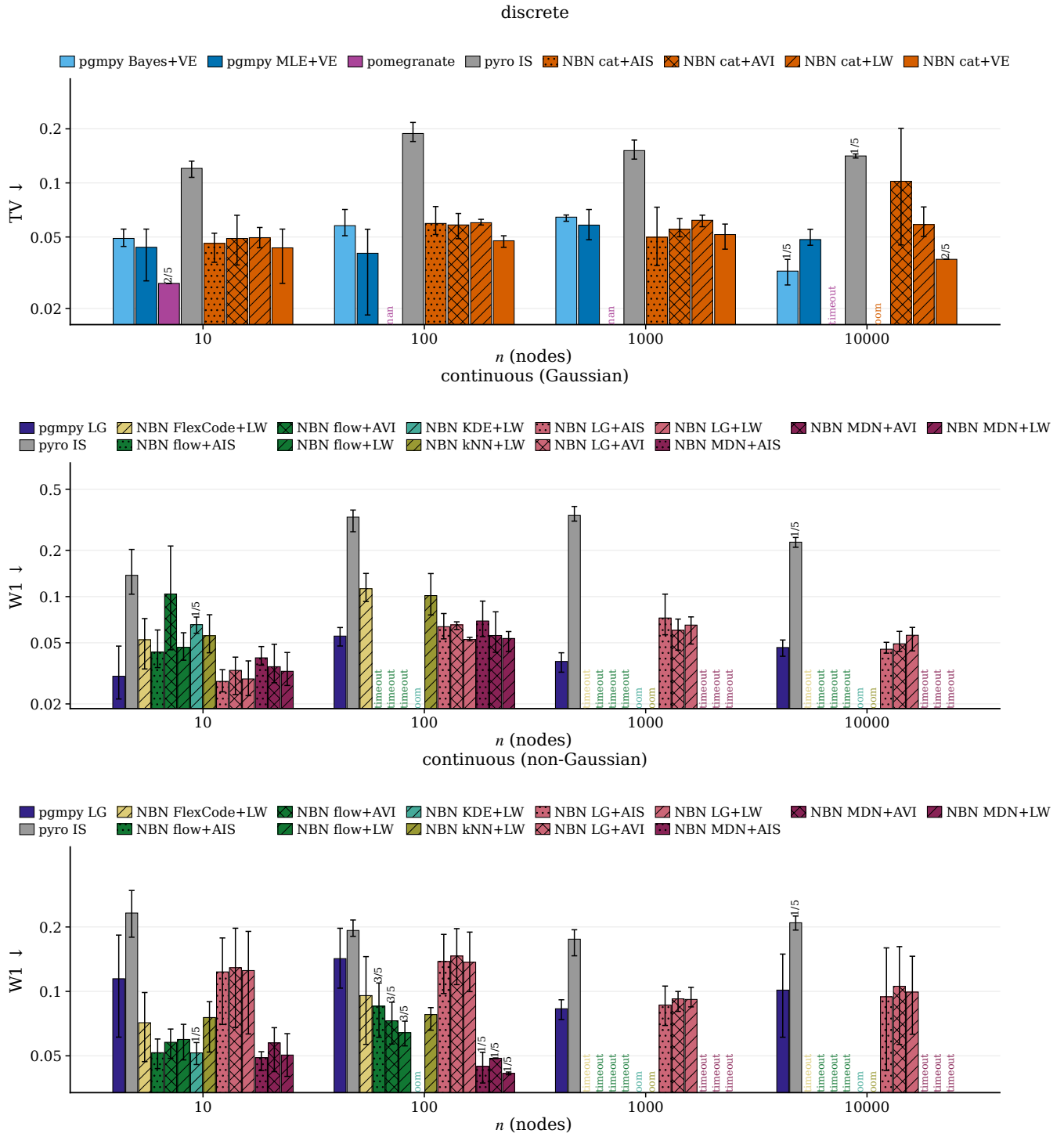


Figure 9: Scalability, all methods: accuracy with full evidence against network size.

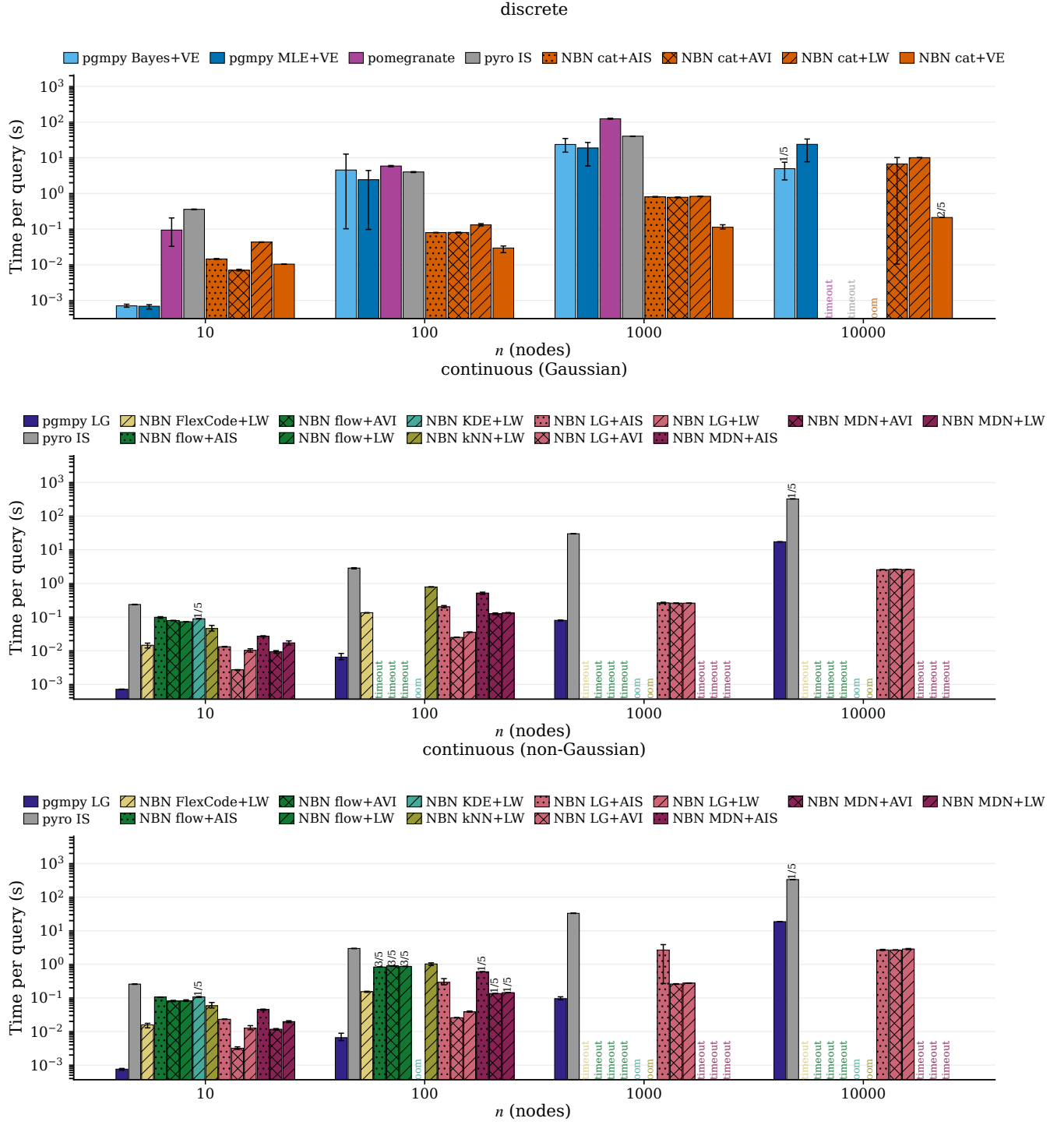


Figure 10: Scalability, all methods: time per query (s), both evidence modes against network size.

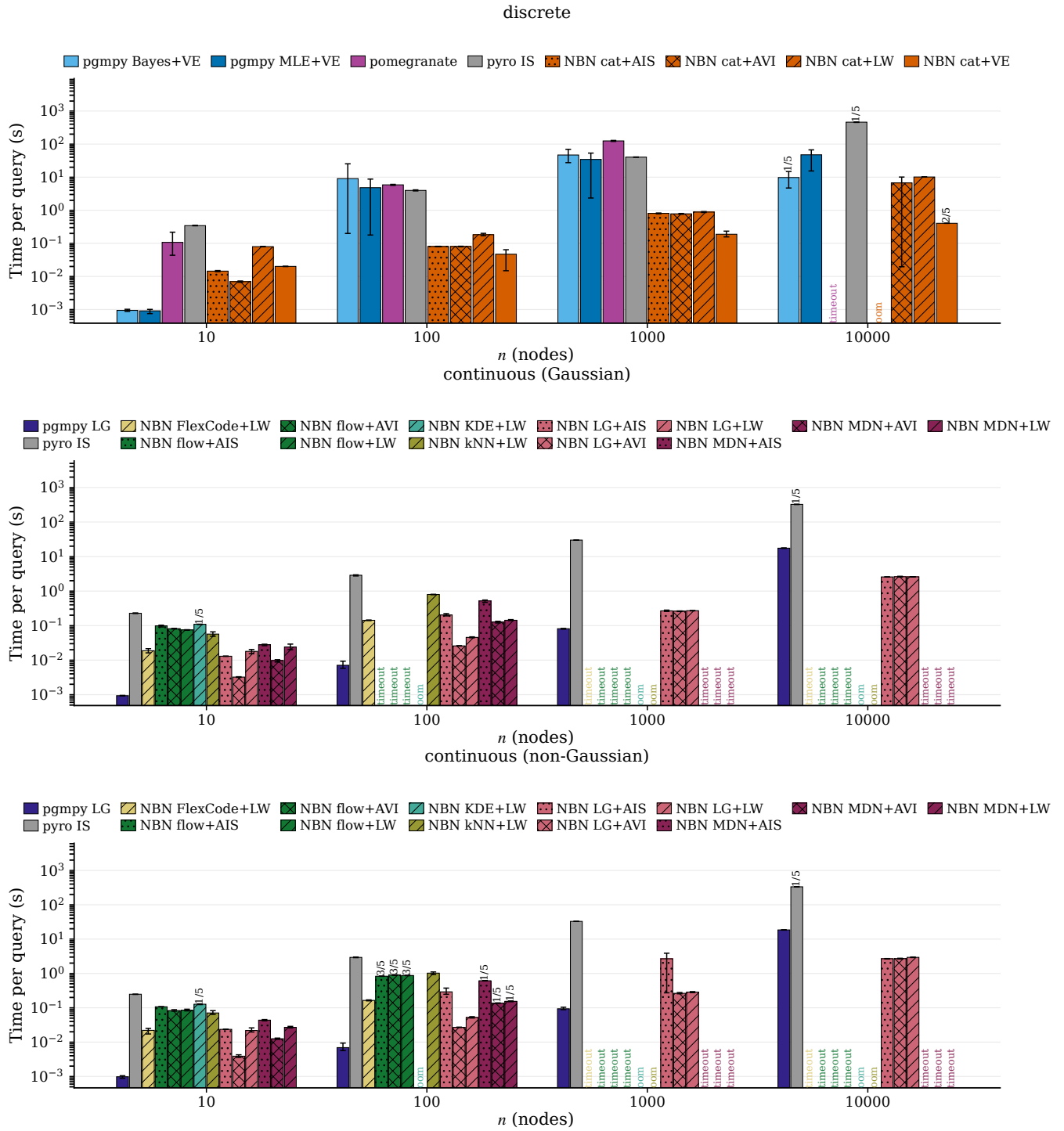


Figure 11: Scalability, all methods: time per query (s), full evidence against network size.

Table 14: Scalability, discrete networks: TV (lower is better), both evidence modes, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-bayes-ve	0.0284±0.0041	0.0343±0.0076	0.037±0.0013	0.0212±0.0029 (1/5)
pgmpy-mle-ve	0.0258±0.008	0.0256±0.0074	0.0343±0.0057	0.0291±0.0026
pomegranate-discrete-ve	0.0159±0.0034 (2/5)	nan	nan	timeout
pyro-empirical-importance	0.123±0.015	0.164±0.0087	0.146±0.022	timeout
nbn-cat-ais	0.0317±0.004	0.0392±0.0034	0.0303±0.01	oom
nbn-cat-avi	0.0337±0.0071	0.0394±0.0046	0.0351±0.001	0.0847±0.079
nbn-cat-lw	0.0326±0.0037	0.0387±0.002	0.0409±0.0026	0.0389±0.0067
nbn-cat-ve	0.0256±0.0083	0.03±0.0018	0.0306±0.0029	0.0225±0.0028 (2/5)

Table 15: Scalability, discrete networks: TV (lower is better), full evidence, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-bayes-ve	0.0491±0.0055	0.0578±0.01	0.0645±0.0026	0.0323±0.0082 (1/5)
pgmpy-mle-ve	0.0438±0.013	0.0406±0.018	0.0582±0.011	0.0484±0.0051
pomegranate-discrete-ve	0.0276±0.0087 (2/5)	nan	nan	timeout
pyro-empirical-importance	0.12±0.012	0.188±0.024	0.151±0.019	0.141±0.046 (1/5)
nbn-cat-ais	0.046±0.0082	0.0595±0.011	0.0498±0.019	oom
nbn-cat-avi	0.049±0.016	0.0583±0.0093	0.0551±0.0066	0.102±0.078
nbn-cat-lw	0.0495±0.0065	0.06±0.0022	0.0619±0.0046	0.0586±0.012
nbn-cat-ve	0.0435±0.014	0.0476±0.0035	0.0516±0.0081	0.0376±0.0067 (2/5)

Table 16: Scalability, discrete networks: time per query in seconds, both evidence modes, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-bayes-ve	0.000709±7e-05	4.54±6.3	23.6±10	4.96±20 (1/5)
pgmpy-mle-ve	0.000689±9.3e-05	2.43±2.2	18.9±11	23.8±13
pomegranate-discrete-ve	0.094±0.087	5.87±0.23	124±3.9	timeout
pyro-empirical-importance	0.359±0.0058	4.01±0.12	40.4±0.34	timeout
nbn-cat-ais	0.0146±0.0004	0.08±0.00099	0.808±0.019	oom
nbn-cat-avi	0.00704±0.00036	0.0806±0.0018	0.782±0.018	6.73±5.1
nbn-cat-lw	0.0435±0.00027	0.131±0.0095	0.831±0.015	10.1±0.035
nbn-cat-ve	0.0104±0.00012	0.0296±0.0059	0.113±0.016	0.212±0.024 (2/5)

Table 17: Scalability, discrete networks: time per query in seconds, full evidence, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-bayes-ve	0.000942±7e-05	9.05±13	46.8±21	9.78±41 (1/5)
pgmpy-mle-ve	0.000912±0.00013	4.83±4.3	34.5±26	47.5±26
pomegranate-discrete-ve	0.107±0.087	5.88±0.21	125±4.3	timeout
pyro-empirical-importance	0.344±0.0056	3.98±0.15	40.4±0.4	461±10 (1/5)
nbn-cat-ais	0.0145±0.00059	0.0802±0.0013	0.807±0.022	oom
nbn-cat-avi	0.00697±0.00031	0.0805±0.0016	0.782±0.022	6.72±5
nbn-cat-lw	0.0796±0.0011	0.183±0.015	0.884±0.027	10.1±0.052
nbn-cat-ve	0.0201±0.00021	0.047±0.025	0.189±0.038	0.404±0.051 (2/5)

Table 18: Scalability, linear-Gaussian networks: W1 (lower is better), both evidence modes, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-lg-predict	0.0353±0.0087	0.0515±0.0028	0.0455±0.0089	0.0482±0.013
pyro-empirical-importance	0.178±0.032	0.3±0.034	0.309±0.037	0.218±0.031 (1/5)
nbn-flexcode-lw	0.0462±0.012	0.101±0.026	timeout	timeout
nbn-flow-ais	0.0494±0.0095	timeout	timeout	timeout
nbn-flow-avi	0.0932±0.073	timeout	timeout	timeout
nbn-flow-lw	0.0436±0.0035	timeout	timeout	timeout
nbn-kde-lw	0.0473±0.0058 (1/5)	oom	oom	oom
nbn-knn-lw	0.0513±0.008	0.0813±0.018	oom	oom
nbn-lg-ais	0.032±0.0045	0.0594±0.0045	0.0584±0.019	0.0475±0.004
nbn-lg-avi	0.0378±0.0038	0.0578±0.0091	0.0546±0.014	0.0478±0.013
nbn-lg-lw	0.0348±0.0079	0.059±0.0065	0.0556±0.0081	0.0537±0.0037
nbn-mdn-ais	0.0488±0.014	0.0636±0.019	timeout	timeout
nbn-mdn-avi	0.0395±0.014	0.0585±0.016	timeout	timeout
nbn-mdn-lw	0.034±0.0072	0.0539±0.0047	timeout	timeout

Table 19: Scalability, linear-Gaussian networks: W1 (lower is better), full evidence, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-lg-predict	0.0303±0.013	0.0553±0.0076	0.0378±0.0054	0.0466±0.0056
pyro-empirical-importance	0.138±0.05	0.33±0.051	0.338±0.038	0.226±0.022 (1/5)
nbn-flexcode-lw	0.0524±0.019	0.113±0.024	timeout	timeout
nbn-flow-ais	0.0435±0.013	timeout	timeout	timeout
nbn-flow-avi	0.104±0.084	timeout	timeout	timeout
nbn-flow-lw	0.0467±0.0098	timeout	timeout	timeout
nbn-kde-lw	0.0656±0.011 (1/5)	oom	oom	oom
nbn-knn-lw	0.0557±0.017	0.102±0.033	oom	oom
nbn-lg-ais	0.028±0.0047	0.0637±0.012	0.0726±0.024	0.0454±0.0038
nbn-lg-avi	0.0331±0.0087	0.0654±0.0037	0.0603±0.013	0.0493±0.0077
nbn-lg-lw	0.0291±0.0077	0.0526±0.0014	0.0652±0.012	0.056±0.0094
nbn-mdn-ais	0.0399±0.0057	0.0694±0.019	timeout	timeout
nbn-mdn-avi	0.035±0.011	0.0558±0.018	timeout	timeout
nbn-mdn-lw	0.0326±0.0084	0.0534±0.0077	timeout	timeout

Table 20: Scalability, linear-Gaussian networks: time per query in seconds, both evidence modes, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-lg-predict	0.000719±1.3e-05	0.00648±0.0014	0.0803±0.0019	17.3±0.16
pyro-empirical-importance	0.237±0.0036	2.86±0.11	30±0.29	324±5.4 (1/5)
nbn-flexcode-lw	0.0146±0.0024	0.135±0.002	timeout	timeout
nbn-flow-ais	0.0989±0.0055	timeout	timeout	timeout
nbn-flow-avi	0.0793±0.00092	timeout	timeout	timeout
nbn-flow-lw	0.0728±0.00074	timeout	timeout	timeout
nbn-kde-lw	0.0893±0.011 (1/5)	oom	oom	oom
nbn-knn-lw	0.0457±0.0091	0.788±0.011	oom	oom
nbn-lg-ais	0.0132±0.00029	0.202±0.015	0.264±0.01	2.56±0.0042
nbn-lg-avi	0.00272±4.5e-05	0.025±0.00018	0.26±0.0051	2.64±0.04
nbn-lg-lw	0.0103±0.0012	0.0357±0.00072	0.26±0.0022	2.57±0.014
nbn-mdn-ais	0.0272±0.00096	0.519±0.039	timeout	timeout
nbn-mdn-avi	0.00938±0.00079	0.126±0.0073	timeout	timeout
nbn-mdn-lw	0.0172±0.0024	0.134±0.0027	timeout	timeout

Table 21: Scalability, linear-Gaussian networks: time per query in seconds, full evidence, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-lg-predict	0.000936±1.7e-05	0.0071±0.0018	0.0812±0.002	17.5±0.27
pyro-empirical-importance	0.228±0.0047	2.88±0.11	30±0.26	324±5.5 (1/5)
nbn-flexcode-lw	0.0187±0.0026	0.143±0.0028	timeout	timeout
nbn-flow-ais	0.0987±0.0057	timeout	timeout	timeout
nbn-flow-avi	0.0816±0.0016	timeout	timeout	timeout
nbn-flow-lw	0.075±0.0012	timeout	timeout	timeout
nbn-kde-lw	0.109±0.0095 (1/5)	oom	oom	oom
nbn-knn-lw	0.0568±0.0085	0.798±0.0098	oom	oom
nbn-lg-ais	0.013±0.00013	0.204±0.016	0.268±0.012	2.57±0.0098
nbn-lg-avi	0.00323±6.9e-05	0.0259±0.00027	0.262±0.005	2.61±0.059
nbn-lg-lw	0.018±0.0025	0.0457±0.0015	0.271±0.0031	2.59±0.025
nbn-mdn-ais	0.0279±0.0013	0.519±0.04	timeout	timeout
nbn-mdn-avi	0.00972±0.00074	0.127±0.0072	timeout	timeout
nbn-mdn-lw	0.024±0.0045	0.143±0.0048	timeout	timeout

Table 22: Scalability, non-Gaussian networks: W1 (lower is better), both evidence modes, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-lg-predict	0.111±0.054	0.136±0.049	0.0891±0.014	0.101±0.048
pyro-empirical-importance	0.228±0.076	0.22±0.032	0.203±0.029	0.24±0.046 (1/5)
nbn-flexcode-lw	0.0687±0.029	0.0981±0.034	timeout	timeout
nbn-flow-ais	0.054±0.013	0.0694±0.0084 (3/5)	timeout	timeout
nbn-flow-avi	0.0613±0.006	0.0682±0.0039 (3/5)	timeout	timeout
nbn-flow-lw	0.0627±0.016	0.0576±0.0028 (3/5)	timeout	timeout
nbn-kde-lw	0.0478±0.012 (1/5)	oom	oom	oom
nbn-knn-lw	0.0691±0.019	0.0783±0.013	oom	oom
nbn-lg-ais	0.11±0.048	0.131±0.043	0.0899±0.018	0.0908±0.056
nbn-lg-avi	0.115±0.057	0.135±0.044	0.0942±0.0087	0.102±0.06
nbn-lg-lw	0.113±0.06	0.13±0.047	0.0934±0.0059	0.0999±0.05
nbn-mdn-ais	0.0458±0.006	0.0564±0.016 (1/5)	timeout	timeout
nbn-mdn-avi	0.0487±0.0071	0.0453±0.006 (1/5)	timeout	timeout
nbn-mdn-lw	0.0484±0.0092	0.0453±0.004 (1/5)	timeout	timeout

Table 23: Scalability, non-Gaussian networks: W1 (lower is better), full evidence, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-lg-predict	0.115±0.061	0.142±0.047	0.083±0.0088	0.101±0.044
pyro-empirical-importance	0.232±0.059	0.193±0.017	0.175±0.024	0.209±0.036 (1/5)
nbn-flexcode-lw	0.0714±0.026	0.0956±0.044	timeout	timeout
nbn-flow-ais	0.0517±0.0082	0.0853±0.012 (3/5)	timeout	timeout
nbn-flow-avi	0.0578±0.009	0.0729±0.0081 (3/5)	timeout	timeout
nbn-flow-lw	0.0596±0.011	0.0641±0.0042 (3/5)	timeout	timeout
nbn-kde-lw	0.0516±0.0076 (1/5)	oom	oom	oom
nbn-knn-lw	0.0755±0.019	0.0779±0.009	oom	oom
nbn-lg-ais	0.123±0.054	0.138±0.043	0.0864±0.018	0.0944±0.058
nbn-lg-avi	0.129±0.065	0.146±0.044	0.0923±0.0097	0.106±0.053
nbn-lg-lw	0.125±0.064	0.137±0.045	0.0917±0.0099	0.0992±0.041
nbn-mdn-ais	0.049±0.0048	0.0446±0.014 (1/5)	timeout	timeout
nbn-mdn-avi	0.0575±0.013	0.0487±0.0053 (1/5)	timeout	timeout
nbn-mdn-lw	0.0504±0.012	0.0414±0.0046 (1/5)	timeout	timeout

Table 24: Scalability, non-Gaussian networks: time per query in seconds, both evidence modes, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-lg-predict	0.00076±4.5e-05	0.00659±0.0018	0.0954±0.01	18.6±0.13
pyro-empirical-importance	0.258±0.0039	2.97±0.031	33.1±0.39	334±5.7 (1/5)
nbn-flexcode-lw	0.0158±0.0023	0.154±0.0053	timeout	timeout
nbn-flow-ais	0.106±0.00086	0.835±0.0039 (3/5)	timeout	timeout
nbn-flow-avi	0.0806±0.0033	0.895±0.0087 (3/5)	timeout	timeout
nbn-flow-lw	0.0818±0.0044	0.87±0.0025 (3/5)	timeout	timeout
nbn-kde-lw	0.107±0.014 (1/5)	oom	oom	oom
nbn-knn-lw	0.0584±0.011	1.03±0.091	oom	oom
nbn-lg-ais	0.0232±0.00029	0.292±0.064	2.66±1.8	2.69±0.097
nbn-lg-avi	0.00319±0.00029	0.0257±0.00028	0.26±0.0053	2.67±0.033
nbn-lg-lw	0.0126±0.0018	0.0387±0.0016	0.277±0.0039	2.87±0.11
nbn-mdn-ais	0.045±0.0018	0.602±0.057 (1/5)	timeout	timeout
nbn-mdn-avi	0.0117±0.00058	0.135±0.013 (1/5)	timeout	timeout
nbn-mdn-lw	0.0198±0.0012	0.144±0.0046 (1/5)	timeout	timeout

Table 25: Scalability, non-Gaussian networks: time per query in seconds, full evidence, against network size.

Method	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
pgmpy-lg-predict	0.000981±8.2e-05	0.00693±0.0019	0.0935±0.0078	18.5±0.2
pyro-empirical-importance	0.248±0.0022	2.93±0.063	32.8±0.026	334±5.7 (1/5)
nbn-flexcode-lw	0.022±0.0039	0.164±0.0052	timeout	timeout
nbn-flow-ais	0.107±0.00053	0.831±0.0018 (3/5)	timeout	timeout
nbn-flow-avi	0.083±0.0041	0.901±0.011 (3/5)	timeout	timeout
nbn-flow-lw	0.0847±0.0047	0.873±0.0048 (3/5)	timeout	timeout
nbn-kde-lw	0.127±0.014 (1/5)	oom	oom	oom
nbn-knn-lw	0.0707±0.0091	1.02±0.089	oom	oom
nbn-lg-ais	0.0236±0.00015	0.291±0.064	2.67±1.8	2.68±0.024
nbn-lg-avi	0.00389±0.00034	0.027±0.00029	0.264±0.011	2.7±0.059
nbn-lg-lw	0.0218±0.0031	0.0521±0.0023	0.287±0.0095	2.94±0.053
nbn-mdn-ais	0.0437±0.0018	0.613±0.066 (1/5)	timeout	timeout
nbn-mdn-avi	0.0126±0.00052	0.137±0.014 (1/5)	timeout	timeout
nbn-mdn-lw	0.0269±0.0014	0.154±0.0071 (1/5)	timeout	timeout

B.3 Batched queries

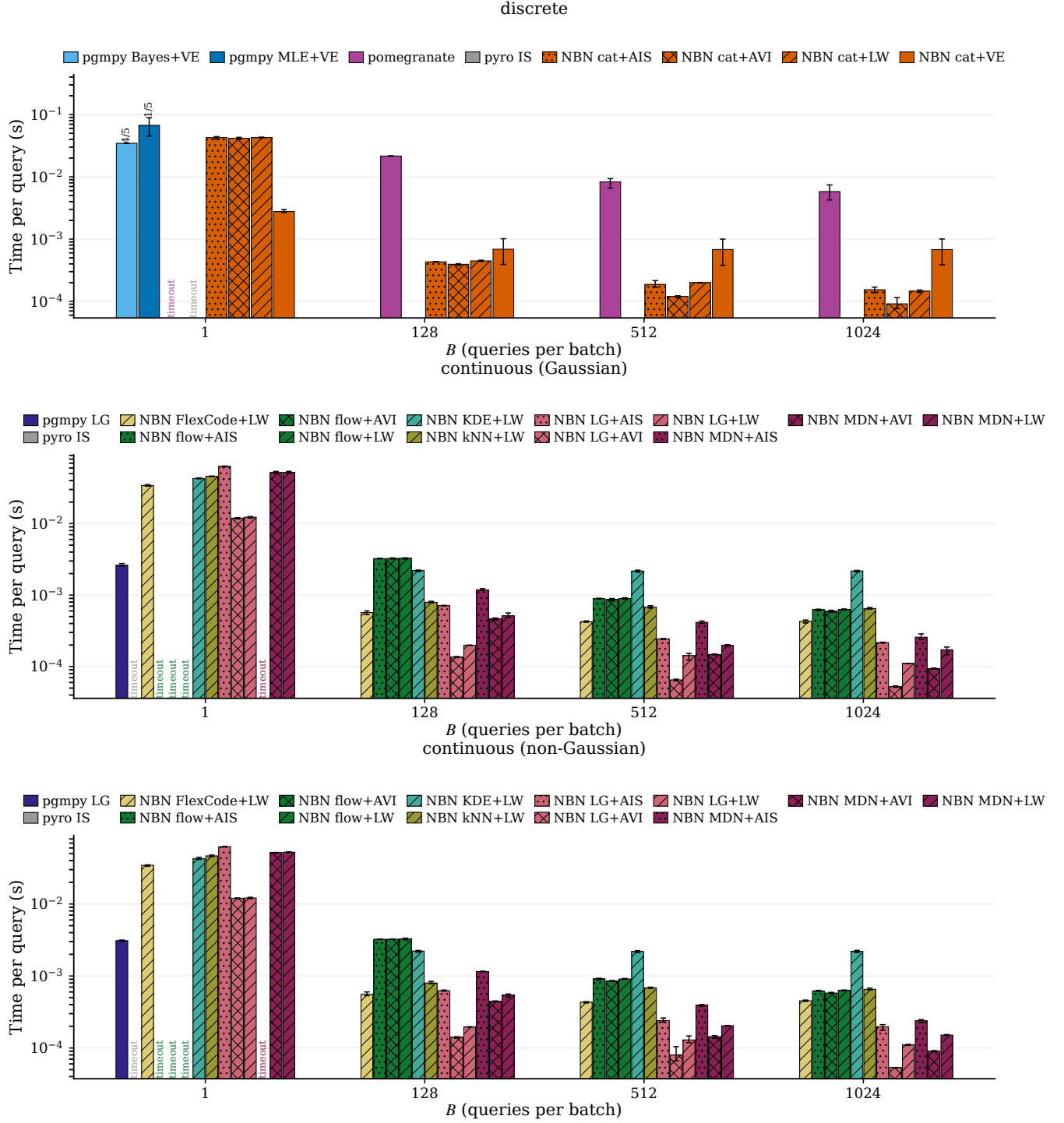


Figure 12: Batched queries, all methods: time per query (s), full evidence against query batch size B .

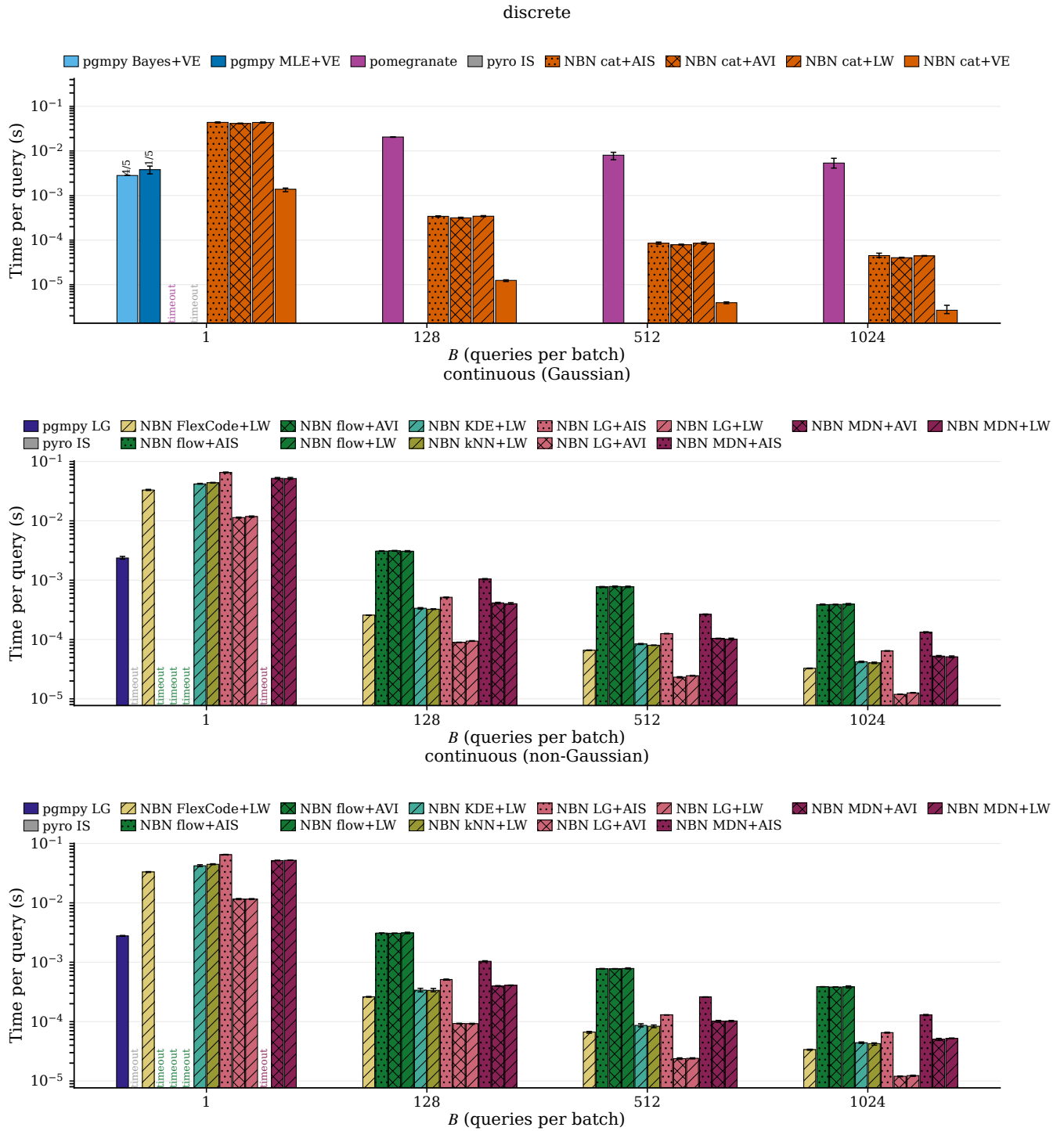


Figure 13: Batched queries, all methods: time per query (s), empty evidence against query batch size B .

Table 26: Batched queries, discrete networks: time per query in seconds, full evidence, against query batch size B .

Method	$B = 1$	$B = 128$	$B = 512$	$B = 1024$
pgmpy-bayes-ve	0.0346±0 (4/5)	–	–	–
pgmpy-mle-ve	0.0671±0.025 (1/5)	–	–	–
pomegranate-discrete-ve	timeout	0.0216±0.00012	0.00829±0.0014	0.0058±0.0016
pyro-empirical-importance	timeout	–	–	–
nbn-cat-ais	0.0425±0.0019	0.000431±4.3e-06	0.000187±2.4e-05	0.000154±1.6e-05
nbn-cat-avi	0.0417±0.0017	0.000394±1.2e-05	0.00012±4.5e-06	9.14e-05±1.9e-05
nbn-cat-lw	0.0425±0.0007	0.000446±9.4e-06	0.0002±1.2e-06	0.000146±6.4e-06
nbn-cat-ve	0.00278±0.00016	0.000688±0.00031	0.000681±0.00031	0.000679±0.00031

Table 27: Batched queries, discrete networks: time per query in seconds, empty evidence, against query batch size B .

Method	$B = 1$	$B = 128$	$B = 512$	$B = 1024$
pgmpy-bayes-ve	0.00281±0 (4/5)	–	–	–
pgmpy-mle-ve	0.00381±0.0012 (1/5)	–	–	–
pomegranate-discrete-ve	timeout	0.0206±0.00026	0.00802±0.0015	0.00535±0.0014
pyro-empirical-importance	timeout	–	–	–
nbn-cat-ais	0.0437±0.0012	0.000341±9.6e-06	8.5e-05±4.7e-06	4.49e-05±5e-06
nbn-cat-avi	0.0413±0.0007	0.000316±9e-06	7.87e-05±1.8e-06	4.02e-05±7.7e-07
nbn-cat-lw	0.0436±0.0011	0.000343±9.5e-06	8.5e-05±4.6e-06	4.42e-05±6.7e-07
nbn-cat-ve	0.00138±0.00013	1.25e-05±5.3e-07	3.94e-06±1.8e-07	2.66e-06±6e-07

Table 28: Batched queries, linear-Gaussian networks: time per query in seconds, full evidence, against query batch size B .

Method	$B = 1$	$B = 128$	$B = 512$	$B = 1024$
pgmpy-lg-predict	0.00262±0.00011	–	–	–
pyro-empirical-importance	timeout	–	–	–
nbn-flexcode-lw	0.0341±0.00083	0.000569±3.4e-05	0.000425±8.9e-06	0.000428±2.1e-05
nbn-flow-ais	timeout	0.00324±2.5e-05	0.000894±9.4e-06	0.000625±1.5e-05
nbn-flow-avi	timeout	0.00325±5.4e-05	0.000867±3e-05	0.000593±2e-05
nbn-flow-lw	timeout	0.00326±6.9e-05	0.000896±2.2e-05	0.000627±1e-05
nbn-kde-lw	0.0428±0.00068	0.0022±4.9e-05	0.00217±6.7e-05	0.00217±5.4e-05
nbn-knn-lw	0.0459±0.00036	0.000792±2.1e-05	0.000682±2.6e-05	0.000649±1.7e-05
nbn-lg-ais	0.0631±0.00075	0.00071±6.8e-06	0.000244±3.4e-06	0.000217±2.6e-06
nbn-lg-avi	0.0119±0.00025	0.000136±2.1e-06	6.51e-05±1.8e-06	5.25e-05±1.2e-06
nbn-lg-lw	0.0123±0.0003	0.000199±7.8e-07	0.000142±1.5e-05	0.000111±4.6e-07
nbn-mdn-ais	timeout	0.00118±3.7e-05	0.000416±1.6e-05	0.000258±2.3e-05
nbn-mdn-avi	0.052±0.0015	0.000467±1.6e-05	0.000148±1.7e-06	9.37e-05±1.5e-06
nbn-mdn-lw	0.052±0.0015	0.000515±3.6e-05	0.000197±4.5e-06	0.000172±2.1e-05

Table 29: Batched queries, linear-Gaussian networks: time per query in seconds, empty evidence, against query batch size B .

Method	$B = 1$	$B = 128$	$B = 512$	$B = 1024$
pgmpy-lg-predict	0.00236±0.00013	–	–	–
pyro-empirical-importance	timeout	–	–	–
nbn-flexcode-lw	0.033±0.00071	0.000257±1.5e-06	6.6e-05±5.6e-07	3.27e-05±4.3e-07
nbn-flow-ais	timeout	0.00309±3.8e-05	0.000773±1.3e-05	0.000388±6.6e-06
nbn-flow-avi	timeout	0.00313±4.1e-05	0.000784±1.6e-05	0.00039±3.6e-06
nbn-flow-lw	timeout	0.00308±6.7e-05	0.000773±2e-05	0.000393±1.2e-05
nbn-kde-lw	0.0419±0.00065	0.000336±9.2e-06	8.42e-05±1.7e-06	4.21e-05±1.1e-06
nbn-knn-lw	0.0439±0.00045	0.000324±4.1e-06	8.01e-05±5.9e-07	4.03e-05±1.1e-06
nbn-lg-ais	0.0654±0.0011	0.000515±6.7e-06	0.000126±8.5e-07	6.42e-05±6.5e-07
nbn-lg-avi	0.0114±0.00021	8.98e-05±5.8e-07	2.33e-05±5.4e-07	1.19e-05±4.7e-08
nbn-lg-lw	0.0118±0.00029	9.4e-05±1.1e-06	2.46e-05±2.8e-07	1.26e-05±1.5e-07
nbn-mdn-ais	timeout	0.00105±2.3e-05	0.000266±3.7e-06	0.000133±2.3e-06
nbn-mdn-avi	0.052±0.0014	0.000414±1.2e-05	0.000104±1.2e-06	5.25e-05±1.6e-06
nbn-mdn-lw	0.0517±0.0019	0.000402±1.3e-05	0.000101±3.3e-06	5.12e-05±1.6e-06

Table 30: Batched queries, non-Gaussian networks: time per query in seconds, full evidence, against query batch size B .

Method	$B = 1$	$B = 128$	$B = 512$	$B = 1024$
pgmpy-lg-predict	0.0031±6.8e-05	–	–	–
pyro-empirical-importance	timeout	–	–	–
nbn-flexcode-lw	0.0344±0.00079	0.000556±3.4e-05	0.000432±9.1e-06	0.000452±9.4e-06
nbn-flow-ais	timeout	0.00324±6.6e-06	0.000911±2.3e-05	0.00062±9e-06
nbn-flow-avi	timeout	0.00324±4.9e-06	0.000846±1.4e-05	0.000579±9.9e-06
nbn-flow-lw	timeout	0.00328±9e-05	0.000908±1.3e-05	0.000629±8.9e-06
nbn-kde-lw	0.0426±0.0015	0.00222±5.9e-05	0.00219±6.6e-05	0.0022±7.7e-05
nbn-knn-lw	0.0466±0.0011	0.0008±3e-05	0.000684±1.3e-05	0.000653±1.9e-05
nbn-lg-ais	0.0625±0.00057	0.000625±1.1e-05	0.00024±1.6e-05	0.000197±1.7e-05
nbn-lg-avi	0.012±0.00012	0.000141±3.7e-06	7.99e-05±1.9e-05	5.34e-05±2e-07
nbn-lg-lw	0.0121±0.00026	0.000196±1.2e-06	0.000129±1.4e-05	0.00011±1.6e-06
nbn-mdn-ais	timeout	0.00115±1.3e-05	0.000392±9.5e-06	0.000239±1.1e-05
nbn-mdn-avi	0.0515±0.00044	0.000444±5.3e-06	0.000144±5.8e-06	9.06e-05±1.9e-06
nbn-mdn-lw	0.0523±0.00056	0.000541±2.6e-05	0.000203±1.3e-06	0.000151±2.3e-06

Table 31: Batched queries, non-Gaussian networks: time per query in seconds, empty evidence, against query batch size B .

Method	$B = 1$	$B = 128$	$B = 512$	$B = 1024$
pgmpy-lg-predict	0.00278±4.3e-05	–	–	–
pyro-empirical-importance	timeout	–	–	–
nbn-flexcode-lw	0.0332±0.00053	0.000261±4.1e-06	6.67e-05±2.2e-06	3.38e-05±6.7e-07
nbn-flow-ais	timeout	0.0031±9.2e-06	0.000775±9.2e-06	0.000387±4.8e-07
nbn-flow-avi	timeout	0.00307±8.2e-06	0.000775±4e-06	0.000383±1e-06
nbn-flow-lw	timeout	0.00311±8.4e-05	0.000785±1.7e-05	0.000385±1.5e-05
nbn-kde-lw	0.0421±0.0015	0.000338±2.4e-05	8.64e-05±5.3e-06	4.41e-05±1.2e-06
nbn-knn-lw	0.0446±0.00078	0.000336±2.2e-05	8.36e-05±4.3e-06	4.26e-05±1.9e-06
nbn-lg-ais	0.0647±0.00064	0.000512±1e-05	0.00013±5e-07	6.52e-05±1.1e-06
nbn-lg-avi	0.0116±0.00025	9.3e-05±1.1e-06	2.38e-05±6.9e-07	1.2e-05±1.8e-07
nbn-lg-lw	0.0116±0.0002	9.22e-05±1.7e-06	2.41e-05±5e-07	1.23e-05±2.8e-07
nbn-mdn-ais	timeout	0.00104±2.1e-05	0.00026±9.3e-07	0.00013±2.7e-06
nbn-mdn-avi	0.0515±0.00046	0.000397±5.5e-06	0.000102±3.6e-06	5.05e-05±2e-06
nbn-mdn-lw	0.0521±0.00017	0.000409±4.6e-06	0.000103±2.6e-06	5.22e-05±5.6e-07

B.4 Real networks

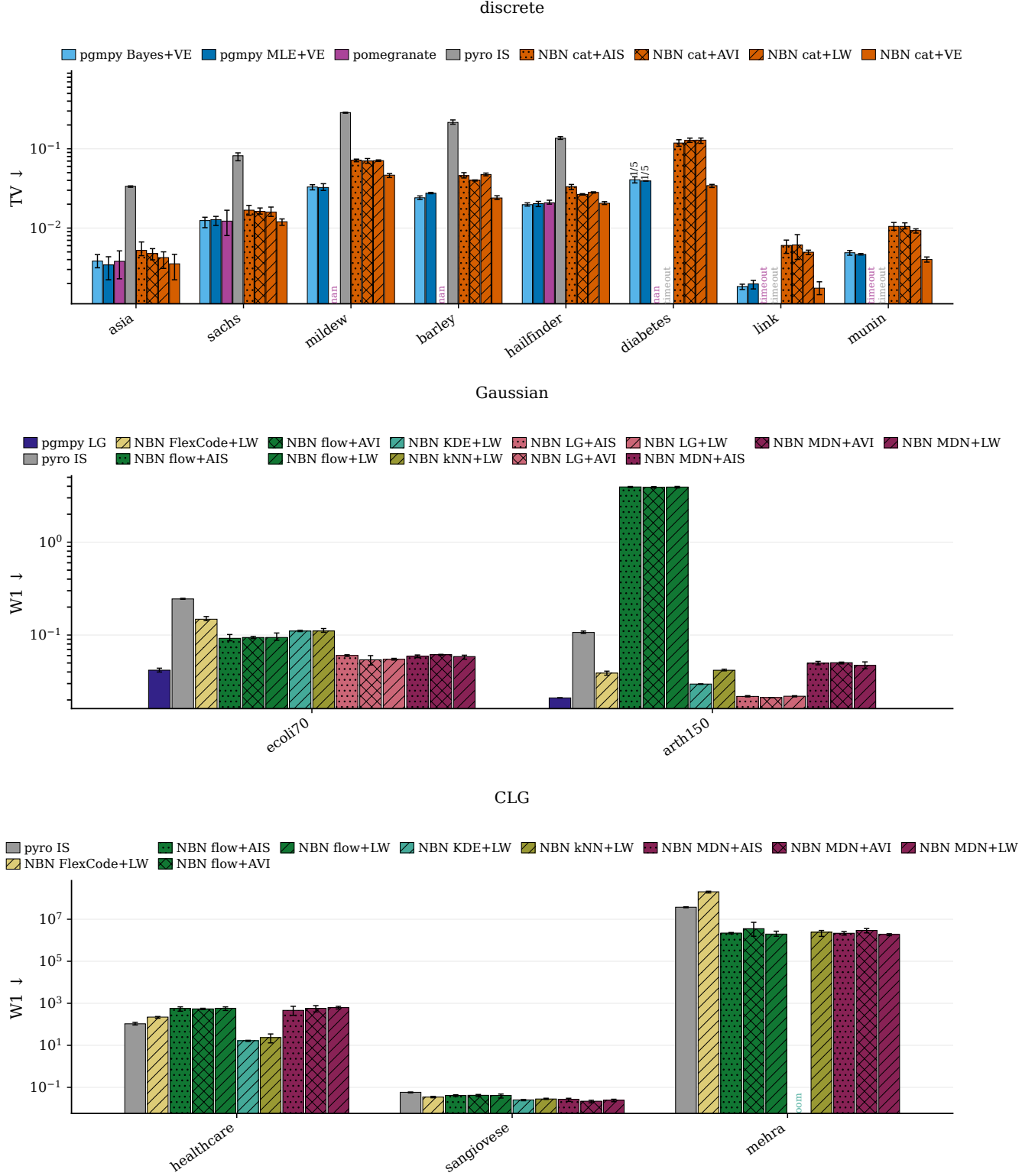


Figure 14: Real networks, all methods: accuracy (TV on discrete targets, W1 on continuous ones) against network.

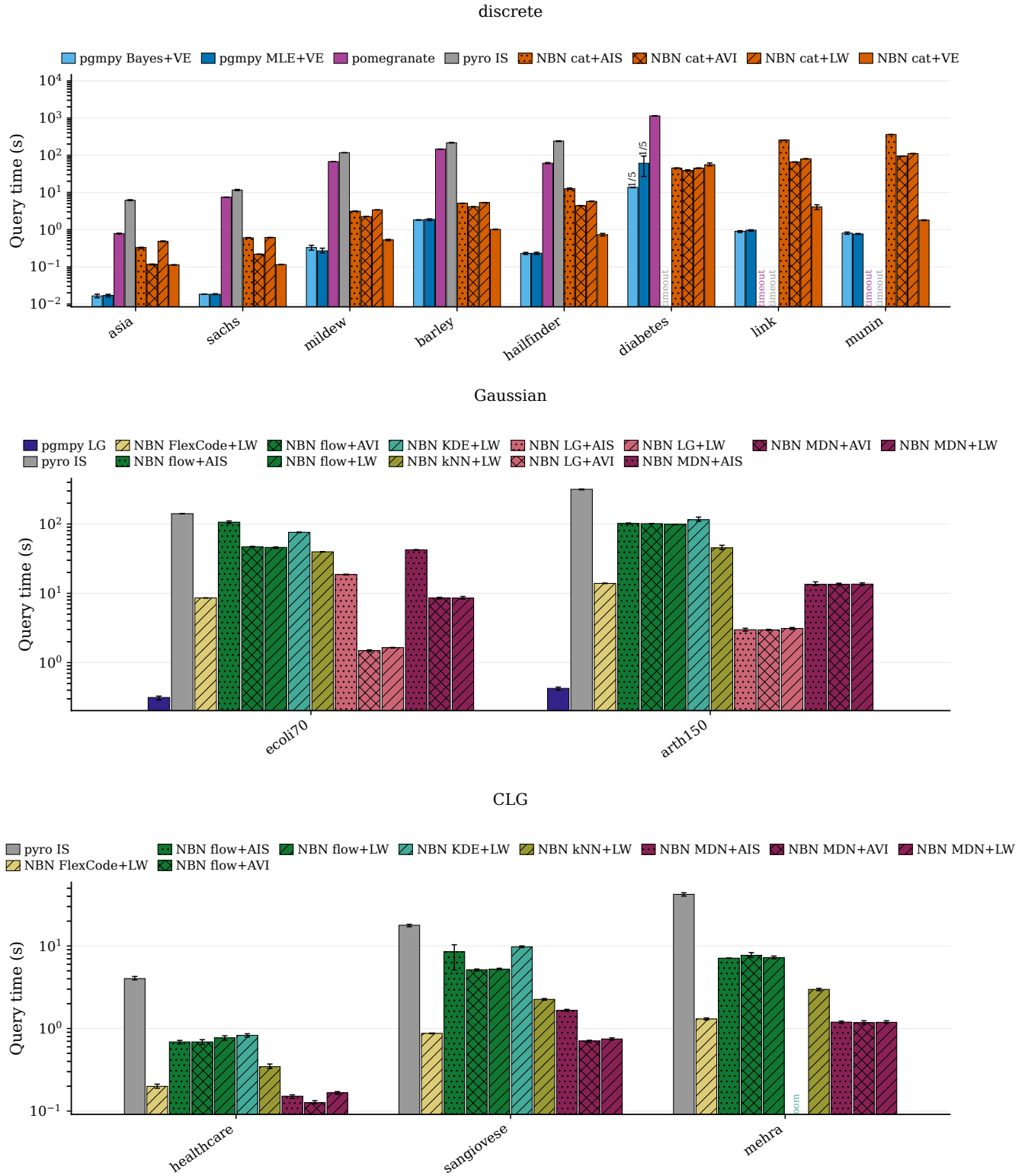


Figure 15: Real networks, all methods: total query time (s) against network.

Table 32: Real networks, discrete networks: TV (lower is better), against network.

Method	asia	sachs	mildew	barley	hailfinder	diabetes	link	munin
pgmpy-bayes-ve	0.00383±0.00074	0.0124±0.0018	0.033±0.0024	0.0241±0.0013	0.0198±0.00092	0.0407±0.0049 (1/5)	0.00183±0.00014	0.00488±0.00033
pgmpy-mle-ve	0.00343±0.0011	0.0128±0.0016	0.0325±0.0032	0.0275±0.00051	0.0203±0.0015	0.0393±0.0011 (1/5)	0.00197±0.00024	0.00464±9.7e-05
pomegranate-discrete-ve	0.0038±0.0014	0.0122±0.0044	nan	nan	0.0208±0.0012	nan	timeout	timeout
pyro-empirical-importance	0.0336±0.00069	0.0818±0.0091	0.287±0.0041	0.215±0.014	0.136±0.0055	timeout	timeout	timeout
nbn-cat-ais	0.00524±0.0011	0.0168±0.0024	0.072±0.0027	0.0459±0.0036	0.0333±0.0025	0.118±0.011	0.00604±0.0011	0.0105±0.0012
nbn-cat-avi	0.00478±0.00076	0.0162±0.0015	0.0703±0.0048	0.0401±0.00064	0.0266±0.00054	0.128±0.008	0.00612±0.0021	0.0105±0.00093
nbn-cat-lw	0.00424±0.00096	0.0159±0.0022	0.0709±0.0016	0.0474±0.0019	0.0281±0.00052	0.129±0.0099	0.00496±0.00032	0.00929±0.00056
nbn-cat-ve	0.00354±0.0012	0.012±0.0011	0.0465±0.0027	0.024±0.0013	0.0206±0.00087	0.0343±0.0016	0.00174±0.00033	0.00401±0.0003

Table 33: Real networks, discrete networks: total query time in seconds, against network.

Method	asia	sachs	mildew	barley	hailfinder	diabetes	link	munin
pgmpy-bayes-ve	0.0165±0.0018	0.0183±8.6e-05	0.331±0.05	1.82±0.027	0.229±0.015	13.6±1.3 (1/5)	0.9±0.05	0.804±0.056
pgmpy-mle-ve	0.0171±0.0014	0.0183±0.00051	0.269±0.041	1.88±0.1	0.232±0.016	60.8±1.1e+02 (1/5)	0.965±0.04	0.763±0.02
pomegranate-discrete-ve	0.785±0.022	7.47±0.036	67.5±0.32	146±0.85	60.9±2.3	1.14e+03±8.4	timeout	timeout
pyro-empirical-importance	6.15±0.18	11.6±0.41	117±0.47	217±4.8	240±5	timeout	timeout	timeout
nbn-cat-ais	0.328±0.012	0.602±0.011	3.12±0.03	5.13±0.033	12.5±0.58	45.2±0.68	255±0.64	360±1.6
nbn-cat-avi	0.116±0.0033	0.219±0.0025	2.25±0.015	4.13±0.02	4.39±0.065	38.7±1.7	65.6±1.1	94.7±0.55
nbn-cat-lw	0.482±0.014	0.614±0.0071	3.39±0.02	5.35±0.015	5.74±0.1	44.8±1.1	79.8±1.1	111±0.29
nbn-cat-ve	0.113±0.0021	0.115±0.0013	0.531±0.023	1.01±0.02	0.733±0.058	55.9±5.2	4.09±0.6	1.8±0.05

Table 34: Real networks, Gaussian networks: W1 (lower is better), against network.

Method	ecoli70	arth150
pgmpy-lg-predict	0.0418±0.002	0.021±7.5e-05
pyro-empirical-importance	0.245±0.0029	0.107±0.0031
nbn-flexcode-lw	0.148±0.0077	0.0389±0.0019
nbn-flow-ais	0.0923±0.0073	3.94±0.062
nbn-flow-avi	0.0942±0.0025	3.91±0.073
nbn-flow-lw	0.0942±0.0087	3.92±0.075
nbn-kde-lw	0.11±0.0018	0.0297±0.0002
nbn-knn-lw	0.111±0.0052	0.0418±0.00083
nbn-lg-ais	0.0603±0.0012	0.0218±0.00034
nbn-lg-avi	0.0541±0.0062	0.0212±1.1e-05
nbn-lg-lw	0.0547±0.0012	0.0219±0.0003
nbn-mdn-ais	0.059±0.002	0.0498±0.0021
nbn-mdn-avi	0.0615±0.00074	0.0503±0.0011
nbn-mdn-lw	0.0586±0.0029	0.0471±0.004

Table 35: Real networks, Gaussian networks: total query time in seconds, against network.

Method	ecoli70	arth150
pgmpy-lg-predict	0.314±0.02	0.425±0.021
pyro-empirical-importance	141±0.82	317±4.6
nbn-flexcode-lw	8.58±0.047	13.9±0.11
nbn-flow-ais	106±4.2	102±1.9
nbn-flow-avi	47±0.58	101±0.25
nbn-flow-lw	45.9±1.1	99±0.95
nbn-kde-lw	76±0.66	115±8.3
nbn-knn-lw	39.7±0.22	45.2±3.3
nbn-lg-ais	18.6±0.26	2.98±0.14
nbn-lg-avi	1.49±0.045	2.97±0.05
nbn-lg-lw	1.64±0.018	3.11±0.1
nbn-mdn-ais	42.4±0.4	13.5±0.85
nbn-mdn-avi	8.6±0.23	13.5±0.41
nbn-mdn-lw	8.55±0.37	13.6±0.6

Table 36: Real networks, CLG networks: W1 (lower is better), against network.

Method	healthcare	sangiovese	mehra
pyro-empirical-importance	106±15	0.0583±0.0027	3.74e+07±1.9e+06
nbn-flexcode-lw	221±21	0.0344±0.0027	2.03e+08±1.9e+07
nbn-flow-ais	572±1.3e+02	0.0412±0.004	2.21e+06±1.6e+05
nbn-flow-avi	543±39	0.042±0.005	3.56e+06±2.8e+06
nbn-flow-lw	581±1.1e+02	0.0412±0.0084	1.97e+06±5.7e+05
nbn-kde-lw	16.6±1	0.0252±0.0017	oom
nbn-knn-lw	23.4±11	0.028±0.0019	2.41e+06±7.1e+05
nbn-mdn-ais	461±2.3e+02	0.0271±0.0044	2.17e+06±4.1e+05
nbn-mdn-avi	576±1.9e+02	0.022±0.0029	2.97e+06±6.8e+05
nbn-mdn-lw	625±90	0.0242±0.0032	1.89e+06±2e+05

Table 37: Real networks, CLG networks: total query time in seconds, against network.

Method	healthcare	sangiovese	mehra
pyro-empirical-importance	4.03±0.19	17.8±0.6	42.2±1.9
nbn-flexcode-lw	0.199±0.011	0.873±0.0092	1.3±0.037
nbn-flow-ais	0.688±0.032	8.52±2.6	7.12±0.023
nbn-flow-avi	0.688±0.046	5.16±0.15	7.71±0.51
nbn-flow-lw	0.771±0.048	5.24±0.11	7.24±0.26
nbn-kde-lw	0.827±0.037	9.78±0.23	oom
nbn-knn-lw	0.345±0.021	2.25±0.052	2.98±0.098
nbn-mdn-ais	0.152±0.0077	1.67±0.047	1.2±0.041
nbn-mdn-avi	0.126±0.0056	0.705±0.02	1.18±0.055
nbn-mdn-lw	0.166±0.0064	0.747±0.023	1.19±0.042

B.5 Parameter learning

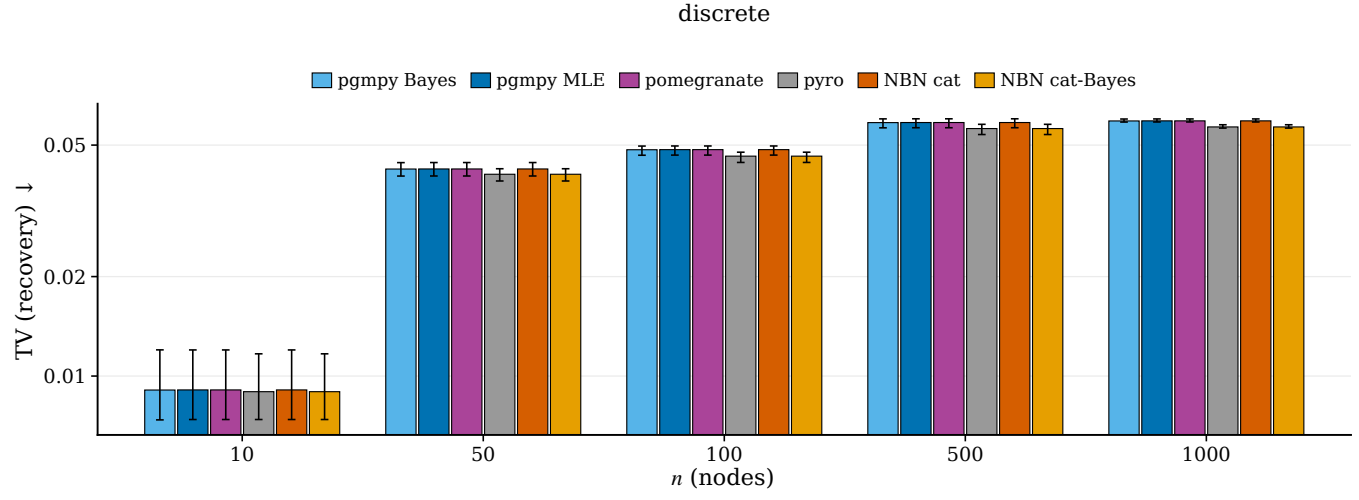


Figure 16: Parameter learning, all methods: parameter-recovery TV against network size.

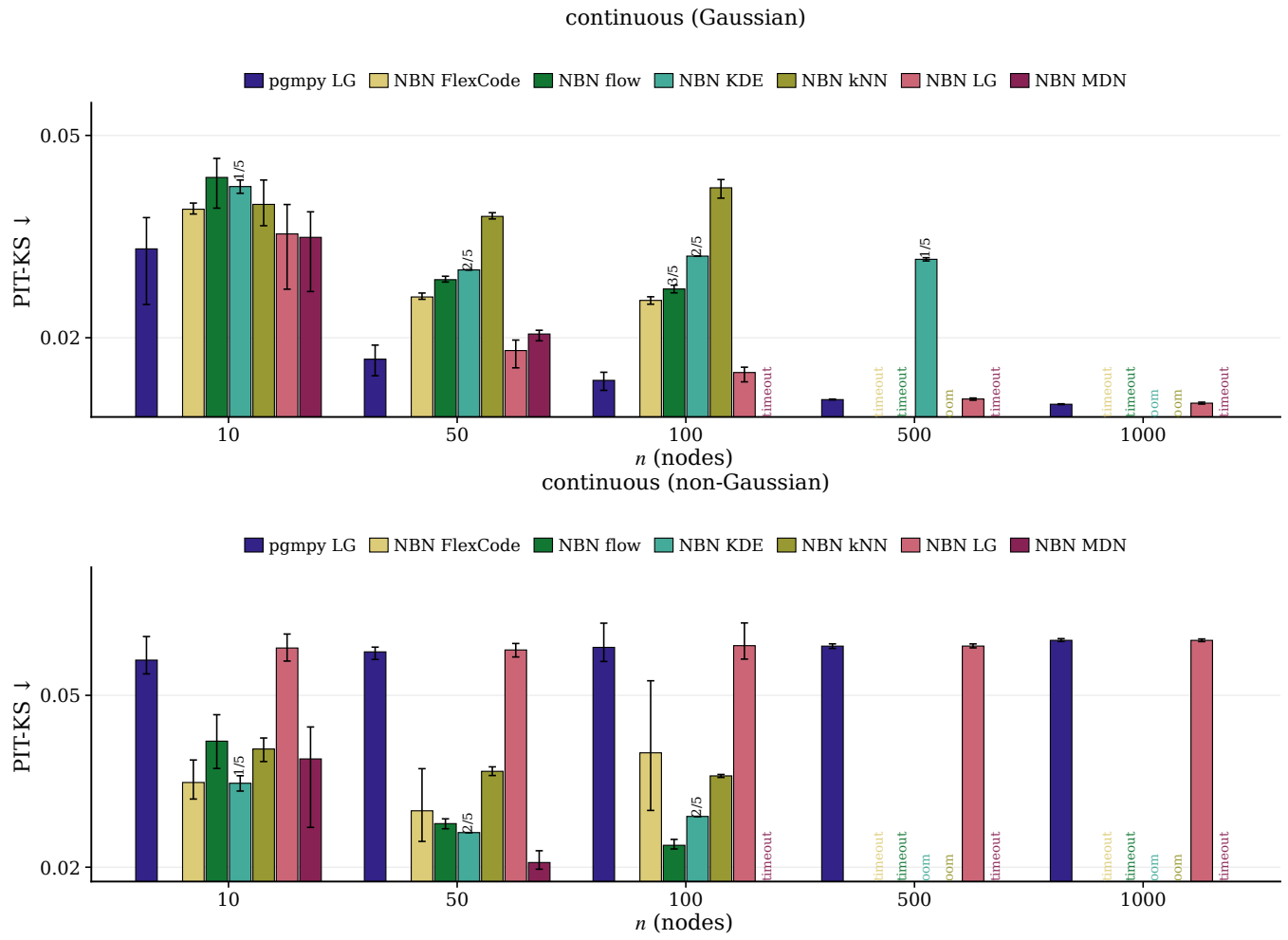


Figure 17: Parameter learning, all methods: calibration error (PIT-KS) against network size.

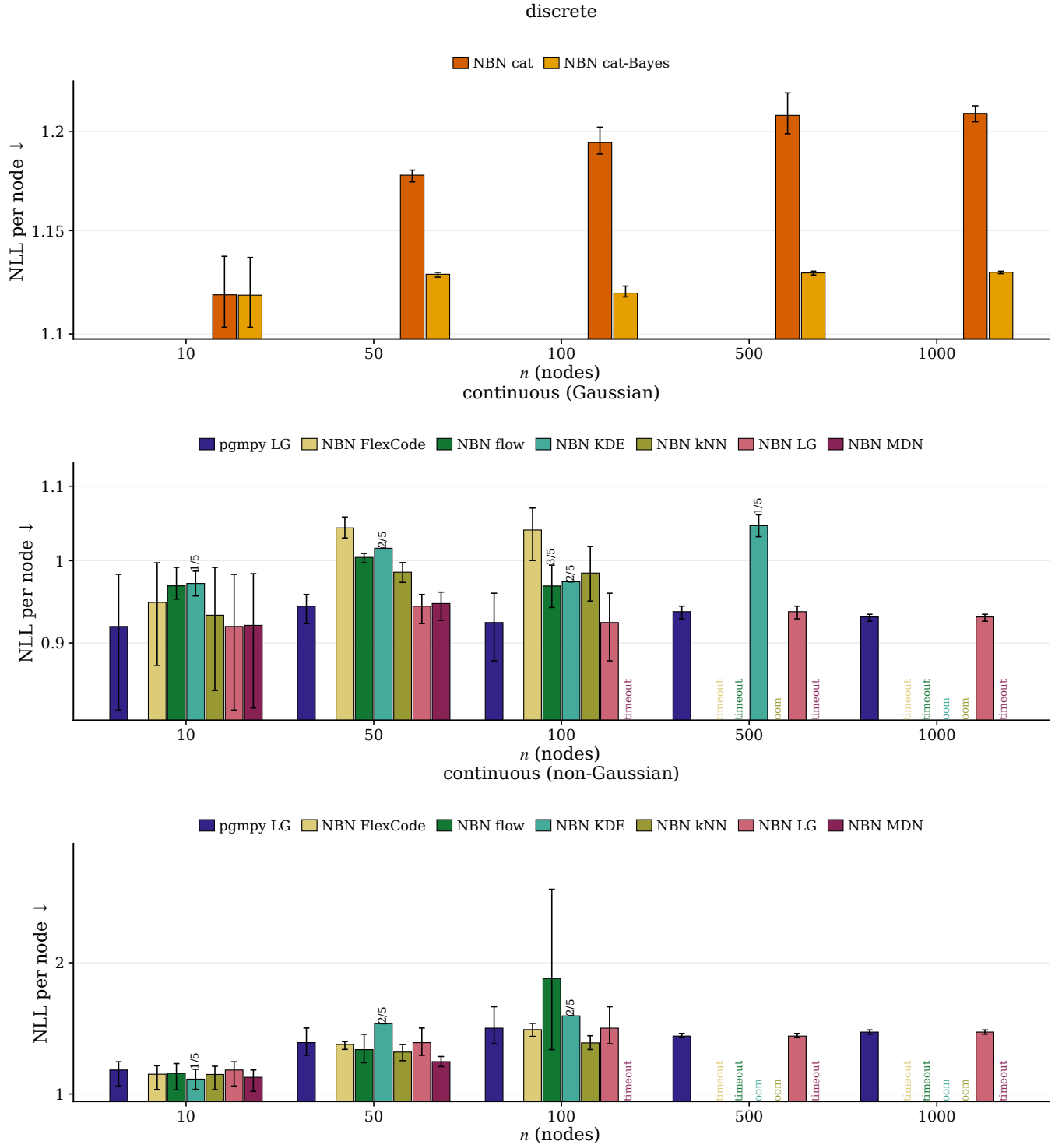


Figure 18: Parameter learning, all methods: held-out negative log-likelihood per node against network size.

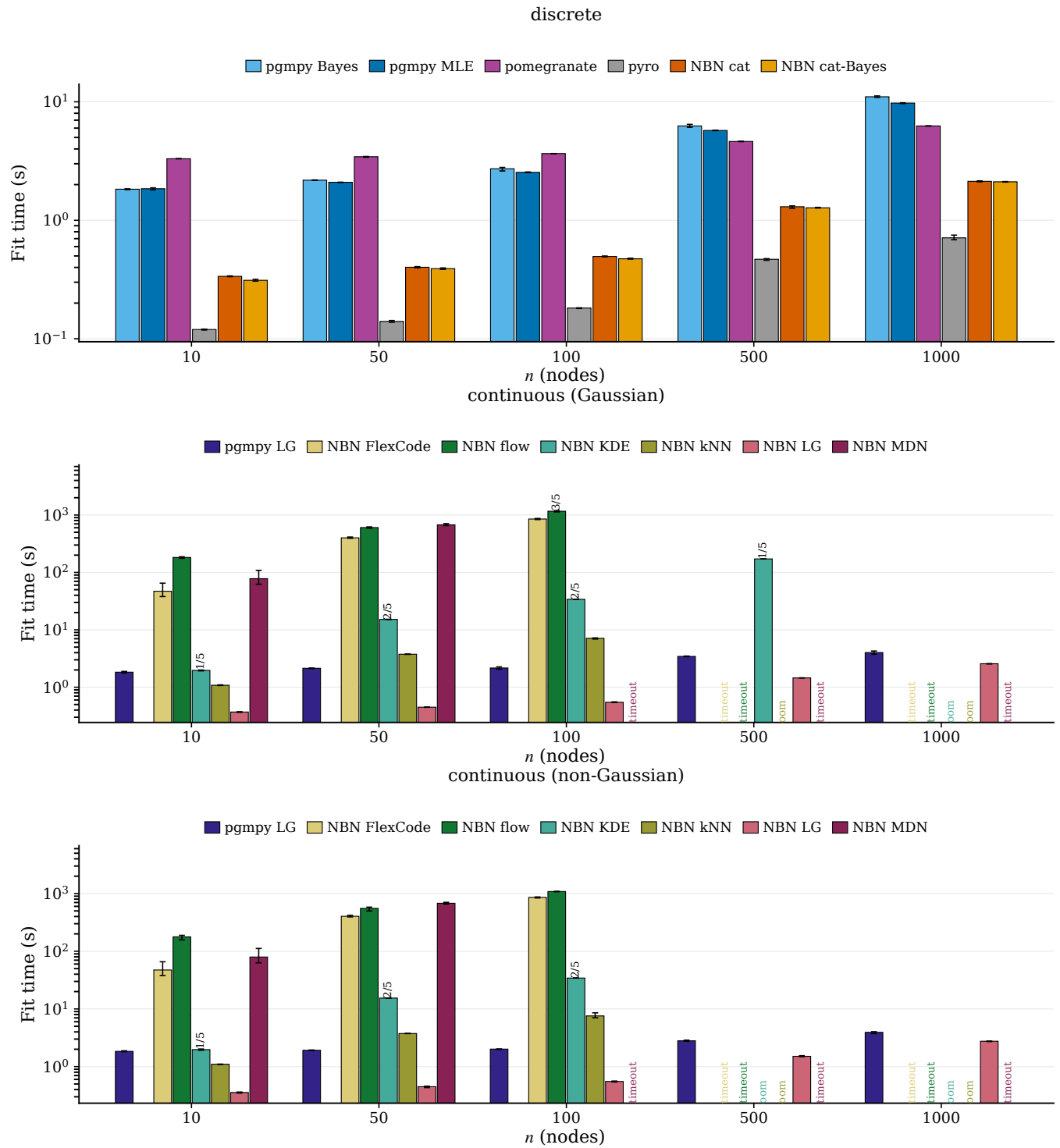


Figure 19: Parameter learning, all methods: fit time (s) against network size.

Table 38: Parameter learning, discrete networks: parameter-recovery TV (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-bayes	0.00907±0.0023	0.0423±0.002	0.0484±0.0015	0.0585±0.0018	0.0592±0.00063
pgmpy-mle	0.00908±0.0023	0.0423±0.002	0.0484±0.0015	0.0585±0.0018	0.0592±0.00063
pomegranate-discrete-ve	0.00908±0.0023	0.0423±0.002	0.0484±0.0015	0.0585±0.0018	0.0592±0.00063
pyro-empirical	0.00897±0.0021	0.0408±0.0017	0.0463±0.0016	0.0561±0.002	0.0567±0.00066
nbn-cat	0.00908±0.0023	0.0423±0.002	0.0484±0.0015	0.0585±0.0018	0.0592±0.00063
nbn-cat-bayes	0.00897±0.0021	0.0408±0.0017	0.0463±0.0016	0.0561±0.002	0.0567±0.00066

Table 39: Parameter learning, discrete networks: held-out negative log-likelihood per node (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
nbn-cat	1.12±0.017	1.18±0.003	1.19±0.0069	1.21±0.011	1.21±0.0042
nbn-cat-bayes	1.12±0.017	1.13±0.0012	1.12±0.0026	1.13±0.00093	1.13±0.00059

Table 40: Parameter learning, discrete networks: fit time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-bayes	1.83±0.015	2.18±0.0091	2.72±0.092	6.26±0.18	11±0.16
pgmpy-mle	1.85±0.037	2.09±0.0058	2.54±0.016	5.73±0.023	9.75±0.079
pomegranate-discrete-ve	3.31±0.0011	3.44±0.025	3.66±0.0033	4.63±0.016	6.26±0.011
pyro-empirical	0.12±0.0011	0.14±0.0024	0.182±0.0012	0.469±0.0074	0.711±0.032
nbn-cat	0.336±0.0024	0.401±0.0046	0.495±0.0039	1.3±0.03	2.13±0.02
nbn-cat-bayes	0.311±0.0056	0.391±0.0055	0.473±0.0041	1.27±0.0083	2.11±0.011

Table 41: Parameter learning, linear-Gaussian networks: PIT-KS (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg	0.0299±0.0056	0.0181±0.0013	0.0165±0.00067	0.0151±2.8e-05	0.0148±2.9e-05
nbn-flexcode	0.0358±0.00089	0.0241±0.00034	0.0237±0.0004	timeout	timeout
nbn-flow	0.0413±0.0045	0.026±0.00034	0.0249±0.00022 (3/5)	timeout	timeout
nbn-kde	0.0397±0.0018 (1/5)	0.0272±0.00022 (2/5)	0.0289±0.0019 (2/5)	0.0285±0.0043 (1/5)	oom
nbn-knn	0.0366±0.0038	0.0346±0.00049	0.0394±0.0017	oom	oom
nbn-lg	0.032±0.0058	0.0189±0.0012	0.0171±0.00056	0.0152±7.5e-05	0.0149±6.4e-05
nbn-mdn	0.0315±0.0054	0.0203±0.00048	timeout	timeout	timeout

Table 42: Parameter learning, linear-Gaussian networks: held-out negative log-likelihood per node (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg	0.919±0.078	0.943±0.018	0.924±0.04	0.937±0.0076	0.931±0.0041
nbn-flexcode	0.948±0.061	1.04±0.014	1.04±0.035	timeout	timeout
nbn-flow	0.968±0.02	1±0.006	0.968±0.013 (3/5)	timeout	timeout
nbn-kde	0.971±0.039 (1/5)	1.02±0.018 (2/5)	0.973±0.013 (2/5)	1.05±0.024 (1/5)	oom
nbn-knn	0.933±0.072	0.985±0.012	0.984±0.034	oom	oom
nbn-lg	0.919±0.078	0.943±0.018	0.924±0.04	0.937±0.0076	0.931±0.0041
nbn-mdn	0.921±0.078	0.947±0.017	timeout	timeout	timeout

Table 43: Parameter learning, linear-Gaussian networks: fit time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg	1.83±0.06	2.14±0.025	2.16±0.095	3.44±0.04	4.02±0.25
nbn-flexcode	47±14	400±10	852±19	timeout	timeout
nbn-flow	182±5.5	605±16	1.16e+03±16 (3/5)	timeout	timeout
nbn-kde	1.96±0.095 (1/5)	15.2±0.23 (2/5)	34±0.77 (2/5)	172±1.3 (1/5)	oom
nbn-knn	1.08±0.013	3.76±0.028	7.11±0.16	oom	oom
nbn-lg	0.37±0.0049	0.449±0.0022	0.547±0.0073	1.45±0.0014	2.57±0.018
nbn-mdn	77.8±23	674±27	timeout	timeout	timeout

Table 44: Parameter learning, non-Gaussian networks: PIT-KS (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg	0.0603±0.0062	0.063±0.0021	0.0646±0.0068	0.065±0.00079	0.067±0.00051
nbn-flexcode	0.0314±0.0033	0.027±0.0054	0.0368±0.013	timeout	timeout
nbn-flow	0.0392±0.0056	0.0252±0.00067	0.0225±0.00058	timeout	timeout
nbn-kde	0.0313±0.0024 (1/5)	0.0241±0.00052 (2/5)	0.0262±0.00022 (2/5)	oom	oom
nbn-knn	0.0376±0.0024	0.0333±0.00077	0.0325±0.00026	oom	oom
nbn-lg	0.0644±0.0047	0.0637±0.0023	0.0652±0.0064	0.065±0.00068	0.067±0.00047
nbn-mdn	0.0356±0.0088	0.0205±0.001	timeout	timeout	timeout

Table 45: Parameter learning, non-Gaussian networks: held-out negative log-likelihood per node (lower is better), against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg	1.13±0.071	1.31±0.095	1.42±0.14	1.36±0.015	1.39±0.016
nbn-flexcode	1.11±0.068	1.3±0.027	1.41±0.048	timeout	timeout
nbn-flow	1.11±0.076	1.26±0.096	1.84±0.84	timeout	timeout
nbn-kde	1.08±0.087 (1/5)	1.45±3.4 (2/5)	1.51±0.032 (2/5)	oom	oom
nbn-knn	1.11±0.067	1.25±0.053	1.31±0.048	oom	oom
nbn-lg	1.13±0.071	1.31±0.095	1.42±0.14	1.36±0.015	1.39±0.016
nbn-mdn	1.09±0.06	1.19±0.031	timeout	timeout	timeout

Table 46: Parameter learning, non-Gaussian networks: fit time in seconds, against network size.

Method	$n = 10$	$n = 50$	$n = 100$	$n = 500$	$n = 1000$
pgmpy-lg	1.84±0.033	1.92±0.02	2.01±0.016	2.8±0.059	3.93±0.15
nbn-flexcode	47.3±14	404±14	858±18	timeout	timeout
nbn-flow	178±16	553±41	1.08e+03±10	timeout	timeout
nbn-kde	1.97±0.1 (1/5)	15.4±0.23 (2/5)	34.3±2 (2/5)	oom	oom
nbn-knn	1.1±0.0083	3.75±0.0084	7.57±0.77	oom	oom
nbn-lg	0.356±0.0075	0.449±0.013	0.553±0.01	1.51±0.031	2.76±0.028
nbn-mdn	79.4±24	679±24	timeout	timeout	timeout

B.6 Sample efficiency

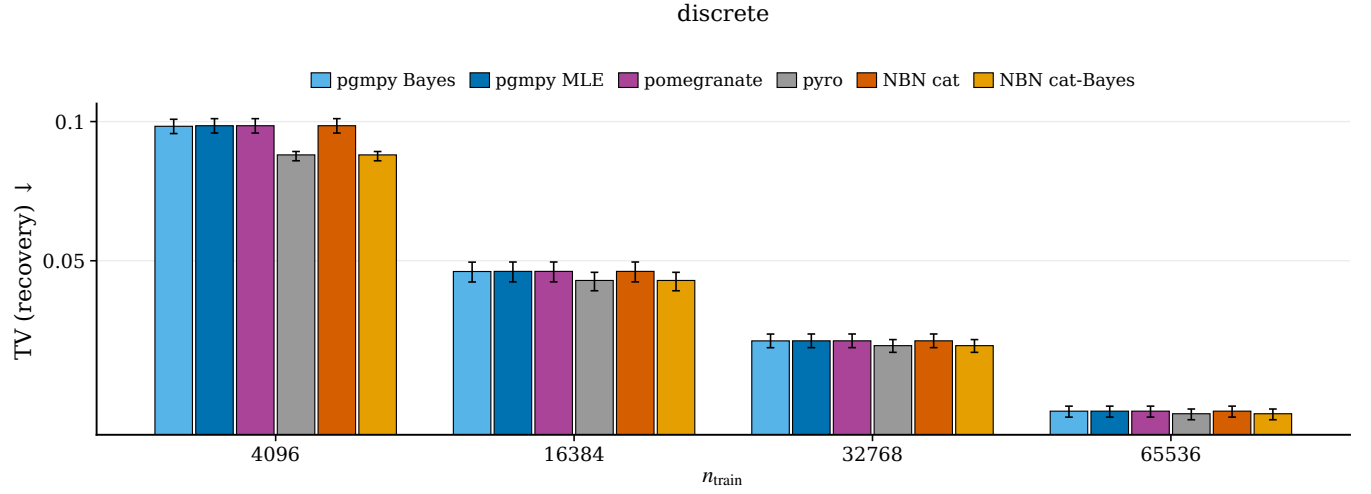


Figure 20: Sample efficiency, all methods: parameter-recovery TV against training-set size.

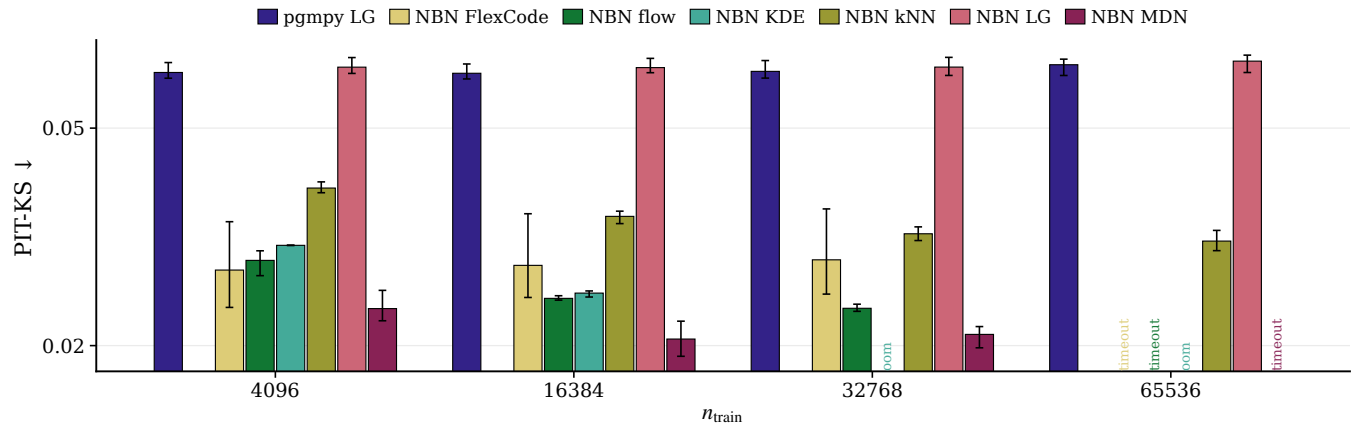
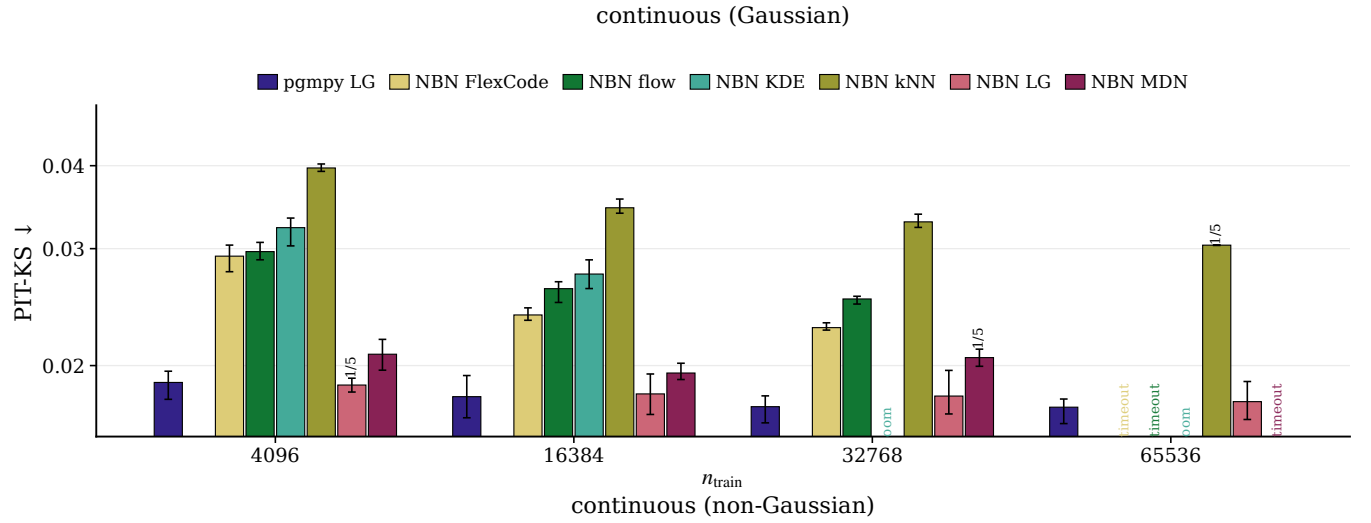


Figure 21: Sample efficiency, all methods: calibration error (PIT-KS) against training-set size.

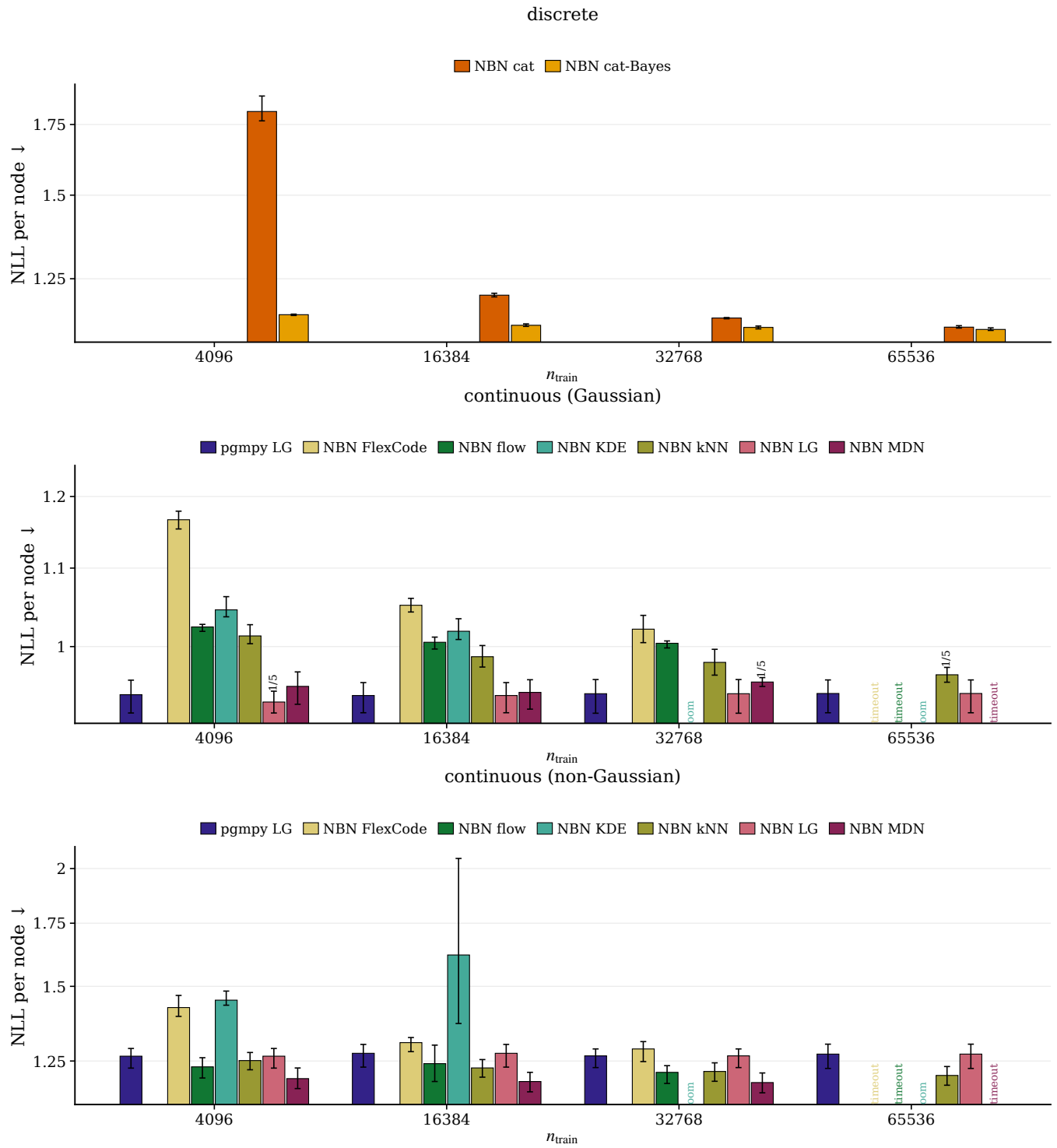


Figure 22: Sample efficiency, all methods: held-out negative log-likelihood per node against training-set size.

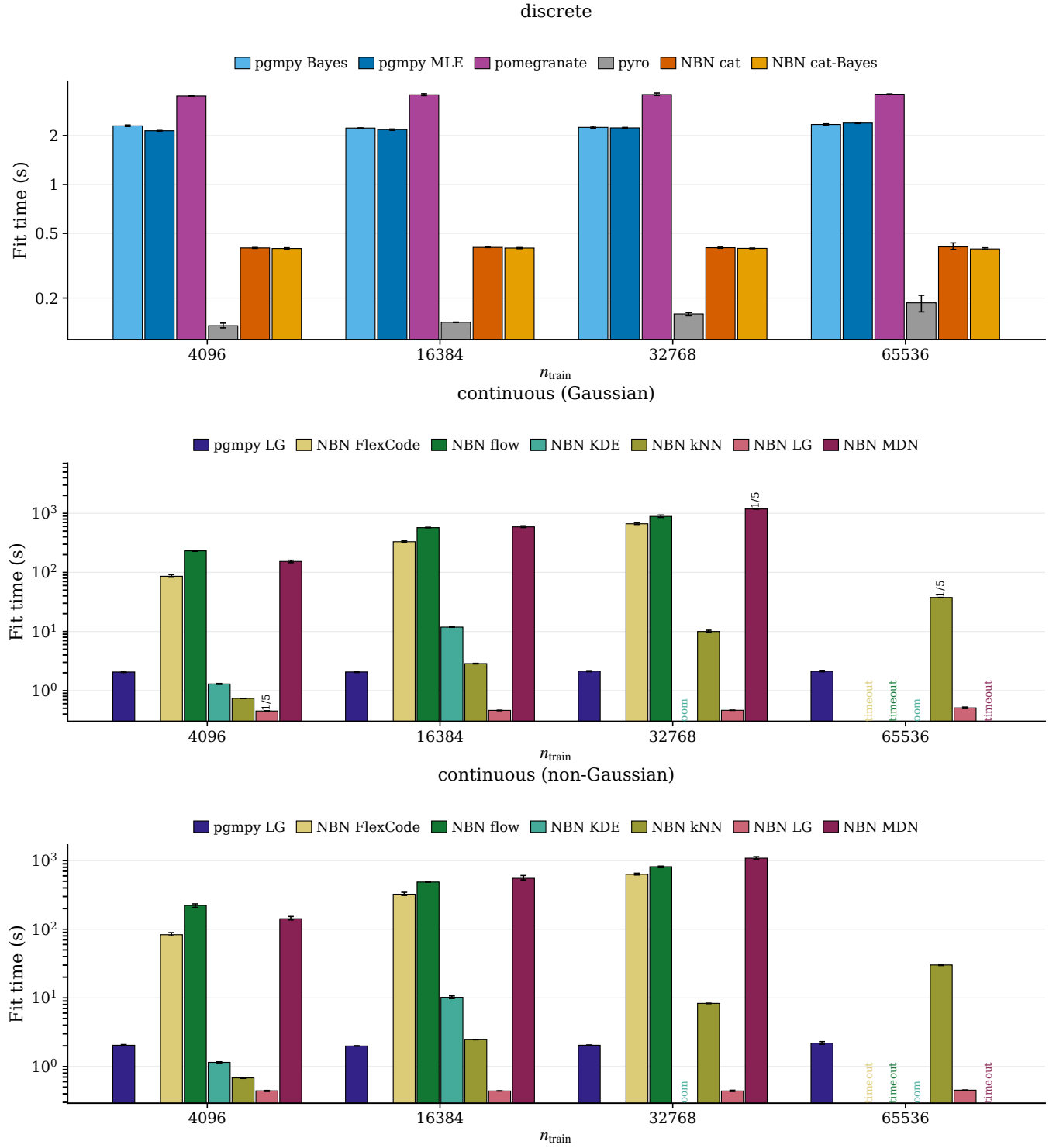


Figure 23: Sample efficiency, all methods: fit time (s) against training-set size.

Table 47: Sample efficiency, discrete networks: parameter-recovery TV (lower is better), against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
pgmpy-bayes	0.0977±0.0035	0.0474±0.0023	0.0335±0.0011	0.0236±0.00064
pgmpy-mle	0.0979±0.0035	0.0474±0.0023	0.0336±0.0011	0.0236±0.00064
pomegranate-discrete-ve	0.0979±0.0035	0.0474±0.0023	0.0336±0.0011	0.0236±0.00064
pyro-empirical	0.0847±0.0019	0.0453±0.0021	0.0328±0.001	0.0233±0.00062
nbn-cat	0.0979±0.0035	0.0474±0.0023	0.0336±0.0011	0.0236±0.00064
nbn-cat-bayes	0.0847±0.0019	0.0453±0.0021	0.0328±0.001	0.0233±0.00062

Table 48: Sample efficiency, discrete networks: held-out negative log-likelihood per node (lower is better), against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
nbn-cat	1.8±0.049	1.21±0.0047	1.15±0.0015	1.12±0.0029
nbn-cat-bayes	1.16±0.0017	1.13±0.0031	1.12±0.0035	1.12±0.0031

Table 49: Sample efficiency, discrete networks: fit time in seconds, against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
pgmpy-bayes	2.29±0.025	2.22±0.0038	2.25±0.035	2.34±0.023
pgmpy-mle	2.14±0.0092	2.18±0.02	2.23±0.013	2.39±0.017
pomegranate-discrete-ve	3.5±0.0068	3.56±0.049	3.57±0.063	3.59±0.021
pyro-empirical	0.135±0.0045	0.142±0.0004	0.16±0.0036	0.187±0.022
nbn-cat	0.407±0.0024	0.41±0.00093	0.408±0.003	0.413±0.019
nbn-cat-bayes	0.404±0.0046	0.406±0.0029	0.404±0.002	0.401±0.0052

Table 50: Sample efficiency, linear-Gaussian networks: PIT-KS (lower is better), against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
pgmpy-lg	0.0189±0.00091	0.018±0.0013	0.0173±0.0008	0.0173±0.00073
nbn-flexcode	0.0292±0.0013	0.0238±0.00052	0.0228±0.00029	timeout
nbn-flow	0.0297±0.0009	0.0261±0.00093	0.0252±0.00034	timeout
nbn-kde	0.0323±0.0015	0.0275±0.0014	oom	oom
nbn-knn	0.0397±0.00051	0.0346±0.00085	0.0329±0.00076	0.0304±0.00049 (1/5)
nbn-lg	0.0187±0.00084 (1/5)	0.0181±0.0013	0.018±0.0014	0.0176±0.0012
nbn-mdn	0.0208±0.0011	0.0195±0.00055	0.0206±0.00096 (1/5)	timeout

Table 51: Sample efficiency, linear-Gaussian networks: held-out negative log-likelihood per node (lower is better), against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
pgmpy-lg	0.943±0.019	0.942±0.017	0.944±0.019	0.945±0.019
nbn-flexcode	1.17±0.013	1.05±0.0087	1.02±0.017	timeout
nbn-flow	1.02±0.0043	1.01±0.0073	1±0.0043	timeout
nbn-kde	1.05±0.013	1.02±0.013	oom	oom
nbn-knn	1.01±0.012	0.988±0.013	0.981±0.015	0.966±0.013 (1/5)
nbn-lg	0.935±0.015 (1/5)	0.942±0.017	0.944±0.019	0.945±0.019
nbn-mdn	0.953±0.019	0.946±0.017	0.958±0.011 (1/5)	timeout

Table 52: Sample efficiency, linear-Gaussian networks: fit time in seconds, against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
pgmpy-lg	2.06±0.049	2.06±0.041	2.14±0.043	2.12±0.057
nbn-flexcode	86±4.3	331±9.1	664±25	timeout
nbn-flow	232±3.9	573±5.5	886±42	timeout
nbn-kde	1.29±0.023	11.9±0.049	oom	oom
nbn-knn	0.737±0.0044	2.86±0.034	9.99±0.39	37.7±0.53 (1/5)
nbn-lg	0.451±0.0072 (1/5)	0.462±0.0039	0.464±0.00023	0.509±0.015
nbn-mdn	153±7.3	589±20	1.18e+03±12 (1/5)	timeout

Table 53: Sample efficiency, non-Gaussian networks: PIT-KS (lower is better), against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
pgmpy-lg	0.0632±0.0021	0.063±0.002	0.0635±0.0024	0.0653±0.0022
nbn-flexcode	0.0275±0.0051	0.028±0.0052	0.0287±0.0054	timeout
nbn-flow	0.0286±0.0015	0.0244±0.00023	0.0234±0.00035	timeout
nbn-kde	0.0305±2.7e-05	0.025±0.00032	oom	oom
nbn-knn	0.0388±0.00089	0.0345±0.0009	0.032±0.00092	0.0311±0.0013
nbn-lg	0.0646±0.0022	0.0645±0.002	0.0647±0.0025	0.0663±0.0024
nbn-mdn	0.0234±0.0015	0.0206±0.0015	0.021±0.00093	timeout

Table 54: Sample efficiency, non-Gaussian networks: held-out negative log-likelihood per node (lower is better), against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
pgmpy-lg	1.26±0.03	1.27±0.035	1.27±0.029	1.27±0.037
nbn-flexcode	1.42±0.037	1.31±0.022	1.29±0.031	timeout
nbn-flow	1.23±0.03	1.24±0.055	1.22±0.026	timeout
nbn-kde	1.45±0.025	1.62±0.34	oom	oom
nbn-knn	1.25±0.026	1.23±0.026	1.22±0.027	1.21±0.028
nbn-lg	1.26±0.03	1.27±0.035	1.27±0.029	1.27±0.037
nbn-mdn	1.2±0.03	1.19±0.028	1.19±0.029	timeout

Table 55: Sample efficiency, non-Gaussian networks: fit time in seconds, against training-set size.

Method	$n_{tr} = 4096$	$n_{tr} = 16384$	$n_{tr} = 32768$	$n_{tr} = 65536$
pgmpy-lg	2.04±0.044	1.99±0.018	2.04±0.016	2.2±0.096
nbn-flexcode	83.4±4.5	324±17	633±19	timeout
nbn-flow	222±13	488±4.8	816±20	timeout
nbn-kde	1.15±0.02	10.2±0.45	oom	oom
nbn-knn	0.681±0.012	2.46±0.0066	8.31±0.054	30.2±0.49
nbn-lg	0.442±0.0072	0.44±0.0015	0.44±0.0089	0.452±0.0021
nbn-mdn	142±8.3	552±42	1.09e+03±44	timeout