D-VAE: A Variational Autoencoder for Directed Acyclic Graphs

Muhan Zhang, Shali Jiang, Zhicheng Cui, Roman Garnett, Yixin Chen
Department of Computer Science and Engineering
Washington University in St. Louis
`{muhan, jiang.s, z.cui, garnett}@wustl.edu`, `chen@cse.wustl.edu`

Abstract

Graph structured data are abundant in the real world. Among different graph types, directed acyclic graphs (DAGs) are of particular interest to machine learning researchers, as many machine learning models are realized as computations on DAGs, including neural networks and Bayesian networks. In this paper, we study deep generative models for DAGs, and propose a novel DAG variational autoencoder (D-VAE). To encode DAGs into the latent space, we leverage graph neural networks. We propose an asynchronous message passing scheme that allows encoding the computations on DAGs, rather than using existing simultaneous message passing schemes to encode local graph structures. We demonstrate the effectiveness of our proposed D-VAE through two tasks: neural architecture search and Bayesian network structure learning. Experiments show that our model not only generates novel and valid DAGs, but also produces a smooth latent space that facilitates searching for DAGs with better performance through Bayesian optimization.

1 Introduction

Many real-world problems can be posed as optimizing of a directed acyclic graph (DAG) representing some computational task. For example, the architecture of a neural network is a DAG. The problem of searching optimal neural architectures is essentially a DAG optimization task. Similarly, one critical problem in learning graphical models – optimizing the connection structures of Bayesian networks [1], is also a DAG optimization task. DAG optimization is pervasive in other fields as well. In electronic circuit design, engineers need to optimize DAG circuit blocks not only to realize target functions, but also to meet specifications such as power usage and operating temperature.

DAG optimization is a hard problem. Firstly, the evaluation of a DAG's performance is often time-consuming (e.g., training a neural network). Secondly, state-of-the-art black-box optimization techniques such as simulated annealing and Bayesian optimization primarily operate in a continuous space, thus are not directly applicable to DAG optimization due to the discrete nature of DAGs. In particular, to make Bayesian optimization work for discrete structures, we need a kernel to measure the similarity between discrete structures as well as a method to explore the design space and extrapolate to new points. Principled solutions to these problems are still lacking.

Is there a way to circumvent the trouble from discreteness? The answer is yes. If we can **embed all DAGs to a continuous space** and make the space relatively smooth, we might be able to directly use principled black-box optimization algorithms to optimize DAGs in this space, or even use gradient methods if gradients are available. Recently, there has been increased interest in training generative models for discrete data types such as molecules [2, 3], arithmetic expressions [4], source code [5], undirected graphs [6], etc. In particular, Kusner et al. [3] developed a grammar variational autoencoder (G-VAE) for molecules, which is able to encode and decode molecules into and from a **continuous latent space**, allowing one to optimize molecule properties by searching in this well-

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

behaved space instead of a discrete space. Inspired by this work, we propose to also train a variational autoencoder for DAGs, and optimize DAG structures in the latent space via Bayesian optimization.

To encode DAGs, we leverage graph neural networks (GNNs) [7]. Traditionally, a GNN treats all nodes symmetrically, and extracts local features around nodes by **simultaneously** passing all nodes' neighbors' messages to themselves. However, such a simultaneous message passing scheme is designed to learn local structure features. It might not be suitable for DAGs, since in a DAG: 1) nodes are not symmetric, but intrinsically have some ordering based on its dependency structure; and 2) we are more concerned about the computation represented by the entire graph, not the local structures.

In this paper, we propose an **asynchronous message passing scheme** to encode the computations on DAGs. The message passing no longer happens at all nodes simultaneously, but respects the computation dependencies (the partial order) among the nodes. For example, suppose node A has two predecessors, B and C, in a DAG. Our scheme does not perform feature learning for A until the feature learning on B and C are both finished. Then, the aggregated message from B and C is passed to A to trigger A's feature learning. This means, although the message passing is not simultaneous, it is also not completely unordered – some synchronization is still required. We incorporate this feature learning scheme in both our encoder and decoder, and propose DAG *variational autoencoder* (D-VAE). D-VAE has an excellent theoretical property for modeling DAGs– we prove that D-VAE can **injectively** encode **computations** on DAGs. This means, we can build a mapping from the discrete space to a continuous latent space so that **every** DAG computation has its **unique** embedding in the latent space, which **justifies** performing optimization in the latent space instead of the original design space.

Our contributions in this paper are: 1) We propose D-VAE, a variational autoencoder for DAGs using a novel asynchronous message passing scheme, which is able to injectively encode computations. 2) Based on D-VAE, we propose a new DAG optimization framework which performs Bayesian optimization in a continuous latent space. 3) We apply D-VAE to two problems, neural architecture search and Bayesian network structure learning. Experiments show that D-VAE not only generates novel and valid DAGs, but also learns smooth latent spaces effective for optimizing DAG structures.

2 Related work

Variational autoencoder (VAE) [8, 9] provides a framework to learn both a probabilistic generative model $p_\theta(\mathbf{x}|\mathbf{z})$ (the decoder) as well as an approximated posterior distribution $q_\phi(\mathbf{z}|\mathbf{x})$ (the encoder). VAE is trained through maximizing the evidence lower bound

$$\mathcal{L}(\phi,\theta;\mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \mathrm{KL}[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})]. \tag{1}$$

The posterior approximation $q_\phi(\mathbf{z}|\mathbf{x})$ and the generative model $p_\theta(\mathbf{x}|\mathbf{z})$ can in principle take arbitrary parametric forms whose parameters ϕ and θ are output by the encoder and decoder networks. After learning $p_\theta(\mathbf{x}|\mathbf{z})$, we can generate new data by decoding latent space vectors $\mathbf{z}$ sampled from the prior $p(\mathbf{z})$. For generating discrete data, $p_\theta(\mathbf{x}|\mathbf{z})$ is often decomposed into a series of decision steps.

Deep graph generative models use neural networks to learn distributions over graphs. There are mainly three types: token-based, adjacency-matrix-based, and graph-based. Token-based models [2, 3, 10] represent a graph as a sequence of tokens (e.g., characters, grammar rules) and model these sequences using RNNs. They are less general since task-specific graph grammars such as SMILES for molecules [11] are required. Adjacency-matrix-based models [12, 13, 14, 15, 16] leverage the proxy adjacency matrix representation of a graph, and generate the matrix in one shot or generate the columns/entries sequentially. In contrast, graph-based models [6, 17, 18, 19] seem more natural, since they operate directly on graph structures (instead of proxy matrix representations) by iteratively adding new nodes/edges to a graph based on the existing graph and node states. In addition, the graph and node states are learned by **graph neural networks (GNNs)**, which have already shown their powerful graph representation learning ability on various tasks [20, 21, 22, 23, 24, 25, 26, 27].

Neural architecture search (NAS) aims at automating the design of neural network architectures. It has seen major advances in recent years [28, 29, 30, 31, 32, 33]. See Hutter et al. [34] for an overview. NAS methods can be mainly categorized into: 1) reinforcement learning methods [28, 31, 33] which train controllers to generate architectures with high rewards in terms of validation accuracy, 2) Bayesian optimization based methods [35] which define kernels to measure architecture similarity and extrapolate the architecture space heuristically, 3) evolutionary approaches [29, 36, 37] which use evolutionary algorithms to optimize neural architectures, and 4) differentiable methods

[32, 38, 39] which use continuous relaxation/mapping of neural architectures to enable gradient-based optimization. In Appendix A, we include more detailed discussion on several most related works.

Bayesian network structure learning (BNSL) is to learn the structure of the underlying Bayesian network from observed data [40, 41, 42, 43]. Bayesian network is a probabilistic graphical model encoding conditional dependencies among variables via a DAG [1]. One main approach for BNSL is score-based search, i.e., define some "goodness-of-fit" score for network structures, and search for one with the optimal score in the discrete design space. Commonly used scores include BIC and BDeu, mostly based on marginal likelihood [1]. Due to the NP-hardness [44], however, exact algorithms such as dynamic programming [45] or shortest path approaches [46, 47] can only solve small-scale problems. Thus, people have to resort to heuristic methods such as local search and simulated annealing, etc. [48]. BNSL is still an active research area [41, 43, 49, 50, 51].

3 DAG variational autoencoder (D-VAE)

In this section, we describe our proposed DAG variational autoencoder (D-VAE). D-VAE uses an asynchronous message passing scheme to encode and decode DAGs. In contrast to the simultaneous message passing in traditional GNNs, D-VAE allows encoding *computations* rather than *structures*.

Definition 1. ***(Computation)*** *Given a set of elementary operations $\mathcal{O}$, a computation C is the composition of a finite number of operations $o \in \mathcal{O}$ applied to an input signal x, with the output of each operation being the input to its succeeding operations.*

The set of elementary operations $\mathcal{O}$ depends on specific applications. For example, when we are interested in computations given by a calculator, $\mathcal{O}$ will be the set of all the operations defined on the functional buttons, such as $+$, $-$, $\times$, $\div$, etc. When modeling neural networks, $\mathcal{O}$ can be a predefined set of basic layers, such as 3×3 convolution, 5×5 convolution, 2×2 max pooling, etc. A computation can be represented as a directed acyclic graph (DAG), with directed edges representing signal flow directions among node operations. The graph must be acyclic, since otherwise the input signal will go through an infinite number of operations so that the computation never stops. Figure 1 shows two examples. Note that the two different DAGs in Figure 1 represent the same computation, as the input signal goes through exactly the same operations. We discuss it further in Appendix B.

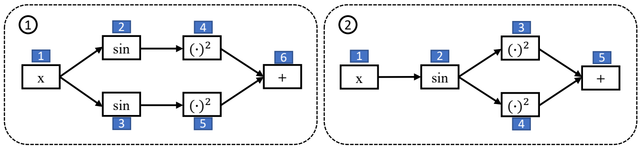

Figure 1: Computations can be represented by DAGs. Note that the left and right DAGs represent the same computation.

3.1 Encoding

We first introduce the encoder of D-VAE, which can be seen as a graph neural network (GNN) using an asynchronous message passing scheme. Given a DAG G, we assume there is a single starting node which does not have any predecessors (e.g., the input layer of a neural architecture). If there are multiple such nodes, we add a virtual starting node connecting to all of them.

Similar to standard GNNs, we use an update function $\mathcal{U}$ to compute the hidden state of each node based on its neighbors' incoming message. The hidden state of node v is given by:

$$\mathbf{h}_v = \mathcal{U}(\mathbf{x}_v, \mathbf{h}_v^{\text{in}}), \tag{2}$$

where $\mathbf{x}_v$ is the one-hot encoding of v's type, and $\mathbf{h}_v^{\text{in}}$ represents the incoming message to v. $\mathbf{h}_v^{\text{in}}$ is given by aggregating the hidden states of v's predecessors using an aggregation function $\mathcal{A}$:

$$\mathbf{h}_v^{\text{in}} = \mathcal{A}(\{\mathbf{h}_u : u \to v\}), \tag{3}$$

where $u \to v$ denotes there is a directed edge from u to v, and $\{\mathbf{h}_u : u \to v\}$ represents a multiset of v's predecessors' hidden states. If an empty set is input to $\mathcal{A}$ (corresponding to the case for the starting node without any predecessors), we let $\mathcal{A}$ output an all-zero vector.

Compared to the traditional simultaneous message passing, in D-VAE the message passing for a node must wait until all of its predecessors' hidden states have already been computed. This simulates how a computation is really performed – to execute some operation, we also need to wait until all its input signals are ready. So how to make sure all the predecessor states are available when a new node comes? One solution is that we can sequentially perform message passing for nodes following a *topological ordering* of the DAG. We illustrate this encoding process in Figure 2.

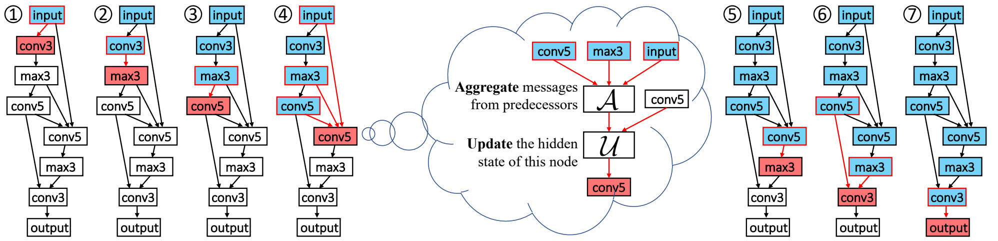
①
②
③
④
⑤
⑥
⑦
input
conv3
max3
conv5
conv5
max3
conv3
output
Aggregate messages from predecessors
$\mathcal{A}$
Update the hidden state of this node
$\mathcal{U}$

Figure 2: An illustration of the encoding procedure for a neural architecture. Following a topological ordering, we iteratively compute the hidden state for each node (red) by feeding in its predecessors' hidden states (blue). This simulates how an input signal goes through a computation, with $\mathbf{h}_v$ simulating the output signal at node v.

After all nodes' hidden states are computed, we use $\mathbf{h}_{v_n}$, the hidden state of the ending node v_n without any successors, as the output of the encoder. Then we feed $\mathbf{h}_{v_n}$ to two MLPs to get the mean and variance parameters of the posterior approximation $q_\phi(\mathbf{z}|G)$ in (1). If there are multiple nodes without successors, we again add a virtual ending node connecting from all of them.

Note that although topological orderings are usually not unique for a DAG, we can take any one of them as the message passing order while ensuring the encoder output is always the same, revealed by the following theorem. We include all theorem proofs in the appendix.

Theorem 1. *The* D-VAE *encoder is invariant to node permutations of the input* DAG *if the aggregation function $\mathcal{A}$ is invariant to the order of its inputs.*

Theorem 1 means isomorphic DAGs are always encoded the same, no matter how we index the nodes. It also indicates that so long as we encode a DAG complying with its partial order, we can perform message passing in arbitrary order (even parallelly for some nodes) with the same encoding result.

The next theorem shows another property of D-VAE that is crucial for its success in modeling DAGs, i.e., it is able to injectively encode computations on DAGs.

Theorem 2. *Let G be any* DAG *representing some computation C. Let $v_1, \ldots, v_n$ be its nodes following a topological order each representing some operation $o_i, 1 \leq i \leq n$, where v_n is the ending node. Then, the encoder of* D-VAE *maps C to $\mathbf{h}_{v_n}$ injectively if $\mathcal{A}$ is injective and $\mathcal{U}$ is injective.*

The significance of Theorem 2 is that it provides a way to injectively encode computations on DAGs, so that every computation has a unique embedding in the latent space. Therefore, instead of performing optimization in the original discrete space, we may alternatively perform optimization in the **continuous latent space**. In this well-behaved Euclidean space, distance is well defined, and principled Bayesian optimization can be applied to search for latent points with high performance scores, which transforms the discrete optimization problem into an easier continuous problem.

Note that Theorem 2 states D-VAE injectively encodes computations on graph structures, rather than graph structures themselves. Being able to injectively encode graph structures is a very strong condition, as it implies an efficient algorithm to solve the challenging graph isomorphism (GI) problem. Luckily, here what we really care about are computations instead of structures, since we do not want to differentiate two different structures G_1 and G_2 as long as they represent the **same computation**. Figure 1 shows such an example. Our D-VAE can identify that the two DAGs in Figure 1 actually represent the same computation by encoding them to the same vector, while those encoders focusing on encoding structures might fail to capture the underlying computation and output different vectors. We discuss more advantages of Theorem 2 in optimizing DAGs in Appendix G.

To model and learn the injective functions $\mathcal{A}$ and $\mathcal{U}$, we resort to neural networks thanks to the universal approximation theorem [52]. For example, we can let $\mathcal{A}$ be a gated sum:

$$\mathbf{h}_v^{\text{in}} = \sum\nolimits_{u \to v} g(\mathbf{h}_u) \odot m(\mathbf{h}_u), \tag{4}$$

where m is a mapping network and g is a gating network. Such a gated sum can model injective multiset functions [53], and is invariant to input order. To model the injective update function $\mathcal{U}$, we can use a gated recurrent unit (GRU) [54], with $\mathbf{h}_v^{\text{in}}$ treated as the input hidden state:

$$\mathbf{h}_v = \text{GRU}_e(\mathbf{x}_v, \mathbf{h}_v^{\text{in}}). \tag{5}$$

Here the subscript e denotes "encoding". Using a GRU also allows reducing our framework to traditional sequence to sequence modeling frameworks [55], as discussed in 3.4.

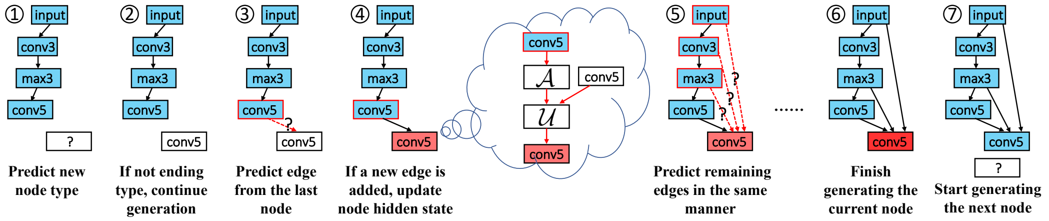
①
input
conv3
max3
conv5
?
Predict new node type
②
If not ending type, continue generation
③
?
Predict edge from the last node
④
If a new edge is added, update node hidden state
$\mathcal{A}$
$\mathcal{U}$
⑤
Predict remaining edges in the same manner
......
⑥
Finish generating the current node
⑦
Start generating the next node

Figure 3: An illustration of the steps for generating a new node.

The above aggregation and update functions can be used to encode general computation graphs. For neural architectures, depending on how the outputs of multiple previous layers are aggregated as the input to a next layer, we will make a modification to (4), which is discussed in Appendix E. For Bayesian networks, we also make some modifications to their encoding due to the special d-separation properties of Bayesian networks, which is discussed in Appendix F.

3.2 Decoding

We now describe how D-VAE decodes latent vectors to DAGs (the generative part). The D-VAE decoder uses the same asynchronous message passing scheme as in the encoder to learn intermediate node and graph states. Similar to (5), the decoder uses another GRU, denoted by GRU_d, to update node hidden states during the generation. Given the latent vector $\mathbf{z}$ to decode, we first use an MLP to map $\mathbf{z}$ to $\mathbf{h_0}$ as the initial hidden state to be fed to GRU_d. Then, the decoder constructs a DAG node by node. For the i^{th} generated node v_i, the following steps are performed:

1. Compute v_i's type distribution using an MLP $f_{\text{add\_vertex}}$ (followed by a softmax) based on the current graph state $\mathbf{h}_G := \mathbf{h}_{v_{i-1}}$.
2. Sample v_i's type. If the sampled type is the ending type, stop the decoding, connect all loose ends (nodes without successors) to v_i, and output the DAG; otherwise, continue the generation.
3. Update v_i's hidden state by $\mathbf{h}_{v_i} = \mathrm{GRU}_d(\mathbf{x}_{v_i}, \mathbf{h}^{\text{in}}_{v_i})$, where $\mathbf{h}^{\text{in}}_{v_i} = \mathbf{h_0}$ if $i = 1$; otherwise, $\mathbf{h}^{\text{in}}_{v_i}$ is the aggregated message from its predecessors' hidden states given by equation (4).
4. For $j = i-1, i-2, \ldots, 1$: (a) compute the edge probability of (v_j, v_i) using an MLP $f_{\text{add\_edge}}$ based on $\mathbf{h}_{v_j}$ and $\mathbf{h}_{v_i}$; (b) sample the edge; and (c) if a new edge is added, update $\mathbf{h}_{v_i}$ using step 3.

The above steps are iteratively applied to each new generated node, until step 2 samples the ending type. For every new node, we first predict its node type based on the current graph state, and then sequentially predict whether each existing node has a directed edge to it based on the existing and current nodes' hidden states. Figure 3 illustrates this process. Since edges always point to new nodes, the generated graph is guaranteed to be acyclic. Note that we maintain hidden states for both the current node and existing nodes, and keep updating them during the generation. For example, whenever step 4 samples a new edge between v_j and v_i, we will update $\mathbf{h}_{v_i}$ to reflect the change of its predecessors and thus the change of the computation so far. Then, we will use the new $\mathbf{h}_{v_i}$ for the next prediction. Such a dynamic updating scheme is flexible, computation-aware, and always uses the up-to-date state of each node to predict next steps. In contrast, methods based on RNNs [3, 13] do not maintain states for old nodes, and only use the current RNN state to predict the next step.

In step 4, when sequentially predicting incoming edges from previous nodes, we choose the reversed order $i-1, \ldots, 1$ instead of $1, \ldots, i-1$ or any other order. This is based on the prior knowledge that a new node v_i is more likely to firstly connect from the node v_{i-1} immediately before it. For example, in neural architecture design, when adding a new layer, we often first connect it from the last added layer, and then decide whether there should be skip connections from other previous layers. Note that however, such an order is not fixed and can be flexible according to specific applications.

3.3 Training

During the training phase, we use teacher forcing [17] to measure the reconstruction loss: following the topological order with which the input DAG's nodes are consumed, we sum the negative log-likelihood of each decoding step by forcing them to generate the ground truth node type or edge at each step. This ensures that the model makes predictions based on the correct histories. Then, we optimize the VAE loss (the negative of (1)) using mini-batch gradient descent following [17]. Note that teacher forcing is only used in training. During generation, we sample a node type or edge at each step according to the decoding distributions described in Section 3.2 and calculate subsequent decoding distributions based on the sampled results.

3.4 Discussion and model extensions

Relation with RNNs. The D-VAE encoder and decoder can be reduced to ordinary RNNs when the input DAG is reduced to a chain of nodes. Although we propose D-VAE from a GNN's perspective, our model can also be seen as a generalization of traditional sequence modeling frameworks [55, 56] where a timestamp depends only on the timestamp immediately before it, to the DAG case where a timestamp has multiple previous dependencies. As special DAGs, similar ideas have been explored for trees [57, 17], where a node can have multiple incoming edges yet only one outgoing edge.

Bidirectional encoding. D-VAE's encoding process can be seen as simulating how an input signal goes through a DAG, with $\mathbf{h}_v$ simulating the output signal at each node v. This is also known as *forward propagation* in neural networks. Inspired by the bidirectional RNN [58], we can also use another GRU to reversely encode a DAG (i.e., reverse all edge directions and encode the DAG again), thus simulating the *backward propagation* too. After reverse encoding, we get two ending states, which are concatenated and linearly mapped to their original size as the final output state. We find this bidirectional encoding can increase the performance and convergence speed on neural architectures.

Incorporating vertex semantics. Note that D-VAE currently uses one-hot encoding of node types as $\mathbf{x}_v$, which does not consider the semantic meanings of different node types. For example, a 3×3 convolution layer might be functionally very similar to a 5×5 convolution layer, while being functionally distinct from a max pooling layer. We expect incorporating such semantic meanings of node types to be able to further improve D-VAE's performance. For example, we can use pretrained embeddings of node types to replace the one-hot encoding. We leave it for future work.

4 Experiments

We validate the proposed DAG variational autoencoder (D-VAE) on two DAG optimization tasks:

- **Neural architecture search.** Our neural network dataset contains 19,020 neural architectures from the ENAS software [33]. Each neural architecture has 6 layers (excluding input and output layers) sampled from: 3×3 and 5×5 convolutions, 3×3 and 5×5 depthwise-separable convolutions [59], 3×3 max pooling, and 3×3 average pooling. We evaluate each neural architecture's weight-sharing accuracy [33] (a proxy of the true accuracy) on CIFAR-10 [60] as its performance measure. We split the dataset into 90% training and 10% held-out test sets. We use the training set for VAE training, and use the test set only for evaluation.
- **Bayesian network structure learning.** Our Bayesian network dataset contains 200,000 random 8-node Bayesian networks from the `bnlearn` package [61] in R. For each network, we compute the Bayesian Information Criterion (BIC) score to measure the performance of the network structure for fitting the Asia dataset [62]. We split the Bayesian networks into 90% training and 10% test sets. For more details, please refer to Appendix I.

Following [3], we do four experiments for each task:

- **Basic abilities of VAE models.** In this experiment, we perform standard tests to evaluate the reconstructive and generative abilities of a VAE model for DAGs, including reconstruction accuracy, prior validity, uniqueness and novelty.
- **Predictive performance of latent representation.** We test how well we can use the latent embeddings of neural architectures and Bayesian networks to predict their performances.
- **Bayesian optimization.** This is the motivating application of D-VAE. We test how well the learned latent space can be used for searching for high-performance DAGs through Bayesian optimization.
- **Latent space visualization.** We visualize the latent space to qualitatively evaluate its smoothness.

Since there is little previous work on DAG generation, we compare D-VAE with four generative baselines adapted for DAGs: S-VAE, GraphRNN, GCN and DeepGMG. Among them, S-VAE [56] and GraphRNN [13] are adjacency-matrix-based methods; GCN [22] and DeepGMG [6] are graph-based methods which use simultaneous message passing to embed DAGs. We include more details about these baselines and discuss D-VAE's advantages over them in Appendix J. The training details are in Appendix K. All the code and data are available at `https://github.com/muhanzhang/D-VAE`.

Table 1: Reconstruction accuracy, prior validity, uniqueness and novelty (%).

	Neural architectures				Bayesian networks			
Methods	Accuracy	Validity	Uniqueness	Novelty	Accuracy	Validity	Uniqueness	Novelty
D-VAE	99.96	100.00	37.26	100.00	99.94	98.84	38.98	98.01
S-VAE	99.98	100.00	37.03	99.99	99.99	100.00	35.51	99.70
GraphRNN	99.85	99.84	29.77	100.00	96.71	100.00	27.30	98.57
GCN	98.70	99.53	34.00	100.00	99.81	99.02	32.84	99.40
DeepGMG	94.98	98.66	46.37	99.93	47.74	98.86	57.27	98.49

4.1 Reconstruction accuracy, prior validity, uniqueness and novelty

Being able to accurately reconstruct input examples and generate valid new examples are basic requirements for VAE models. In this experiment, we evaluate the models by measuring 1) how often they can reconstruct input DAGs perfectly (Accuracy), 2) how often they can generate valid neural architectures or Bayesian networks from the prior distribution (Validity), 3) the proportion of unique DAGs out of the valid generations (Uniqueness), and 4) the proportion of valid generations that are never seen in the training set (Novelty).

We first evaluate each model's reconstruction accuracy on the test sets. Following previous work [3, 17], we regard the encoding as a stochastic process. That is, after getting the mean and variance parameters of the posterior approximation $q_\phi(\mathbf{z}|G)$, we sample a $\mathbf{z}$ from it as G's latent vector. To estimate the reconstruction accuracy, we sample $\mathbf{z}$ 10 times for each G, and decode each $\mathbf{z}$ 10 times too. Then we report the average proportion of the 100 decoded DAGs that are identical to the input. To calculate prior validity, we sample 1,000 latent vectors $\mathbf{z}$ from the prior distribution $p(\mathbf{z})$ and decode each latent vector 10 times. Then we report the proportion of valid DAGs in these 10,000 generations. A generated DAG is valid if it can be read by the original software which generated the training data. More details about the validity experiment are in Appendix M.1.

We show the results in Table 1. Among all the models, D-VAE and S-VAE generally perform the best. We find that D-VAE, S-VAE and GraphRNN all have near perfect reconstruction accuracy, prior validity and novelty. However, D-VAE and S-VAE show higher uniqueness, meaning that they generate more diverse examples. GCN and DeepGMG have worse reconstruction accuracies for neural architectures due to nonzero training losses. This is because the simultaneous message passing scheme in them focus more on learning local graph structures, but fail to encode the computation represented by the entire neural network. Besides, the sum pooling after the message passing might also lose some global topology information which is important for the reconstruction. The nonzero training loss of DeepGMG acts like an early stopping regularizer, making DeepGMG generate more unique graphs. Nevertheless, reconstruction accuracy is much more important than uniqueness in our tasks, since we want our embeddings to accurately remap to their original structures after latent space optimization.

4.2 Predictive performance of latent representation.

In this experiment, we evaluate how well the learned latent embeddings can predict the corresponding DAGs' performances, which tests a VAE's unsupervised representation learning ability. Being able to accurately predict a latent point's performance also makes it much easier to search for high-performance points in this latent space. Thus, the experiment is also an indirect way to evaluate a VAE latent space's amenability for DAG optimization. Following [3], we train a sparse Gaussian process (SGP) model [63] with 500 inducing points on the embeddings of training data to predict the performance of unseen test data. We include the SGP training details in Appendix L.

Table 2: Predictive performance of encoded means.

	Neural architectures		Bayesian networks	
Methods	RMSE	Pearson's r	RMSE	Pearson's r
D-VAE	**0.384±0.002**	**0.920±0.001**	**0.300±0.004**	**0.959±0.001**
S-VAE	0.478±0.002	0.873±0.001	0.369±0.003	0.933±0.001
GraphRNN	0.726±0.002	0.669±0.001	0.774±0.007	0.641±0.002
GCN	0.485±0.006	0.870±0.001	0.557±0.006	0.836±0.002
DeepGMG	0.433±0.002	0.897±0.001	0.788±0.007	0.625±0.002

We use two metrics to evaluate the predictive performance of the latent embeddings (given by the mean of the posterior approximations $q_\phi(\mathbf{z}|G)$). One is the RMSE between the SGP predictions and the true performances. The other is the Pearson correlation coefficient (or Pearson's r), measuring how well the prediction and real performance tend to go up and down together. A small RMSE and a large Pearson's r indicate a better predictive performance.

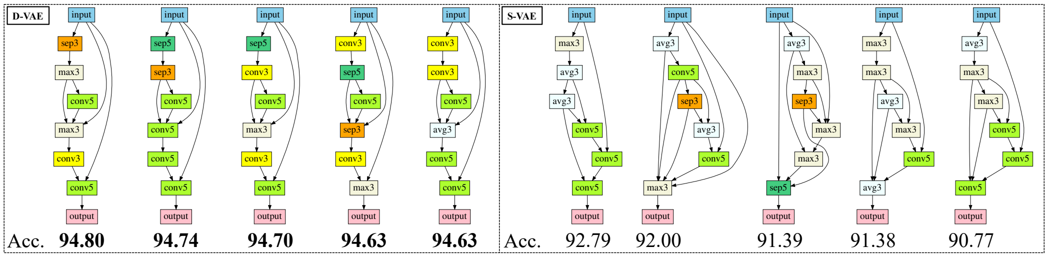
D-VAE
Acc. 94.80 94.74 94.70 94.63 94.63
S-VAE
Acc. 92.79 92.00 91.39 91.38 90.77
input
sep3
sep5
conv3
conv5
max3
avg3
output

Figure 4: Top 5 neural architectures found by each model and their true test accuracies.

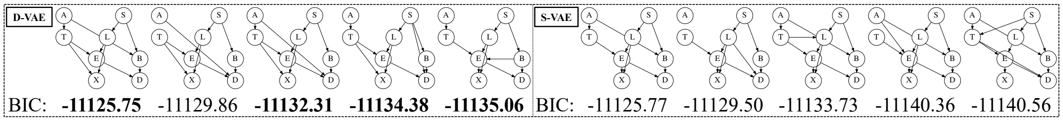
D-VAE
BIC: -11125.75 -11129.86 -11132.31 -11134.38 -11135.06
S-VAE
BIC: -11125.77 -11129.50 -11133.73 -11140.36 -11140.56
A
S
T
L
E
B
X
D

Figure 5: Top 5 Bayesian networks found by each model and their BIC scores (higher the better).

All the experiments are repeated 10 times and the means and standard deviations are reported. Table 2 shows the results. We find that both the RMSE and Pearson's r of D-VAE are significantly better than those of the other models. A possible explanation is that D-VAE encodes the computation, while a DAG's performance is primarily determined by its computation. Therefore, D-VAE's latent embeddings are more informative about performance. In comparison, adjacency-matrix-based methods (S-VAE and GraphRNN) and graph-based methods with simultaneous message passing (GCN and DeepGMG) both only encode (local) graph structures without specifically modeling computations on DAG structures. The better predictive power of D-VAE favors using a predictive model in its latent space to guide the search for high performance graphs.

4.3 Bayesian optimization

We perform Bayesian optimization (BO) using the two best models, D-VAE and S-VAE, validated by previous experiments. Based on the SGP model from the last experiment, we perform 10 iterations of batch BO, and average results across 10 trials. Following Kusner et al. [3], in each iteration, a batch of 50 points are proposed by sequentially maximizing the expected improvement (EI) acquisition function, using Kriging Believer [64] to assume labels for previously chosen points in the batch. For each batch of selected points, we evaluate their decoded DAGs' real performances and add them back to the SGP to select the next batch. Finally, we check the best-performing DAGs found by each model to evaluate its DAG optimization performance.

Neural architectures. For neural architectures, we select the top 15 found architectures in terms of their weight-sharing accuracies, and fully train them on CIFAR-10's train set to evaluate their true test accuracies. More details can be found in Appendix H. We show the 5 architectures with the highest true test accuracies in Figure 4. As we can see, D-VAE in general found much better neural architectures than S-VAE. Among the selected architectures, D-VAE achieved a highest accuracy of 94.80%, while S-VAE's highest accuracy was only 92.79%. In addition, all the 5 architectures of D-VAE have accuracies higher than 94%, indicating that D-VAE's latent space can stably find many high-performance architectures. More details about our NAS experiments are in Appendix H.

Bayesian networks. We similarly report the top 5 Bayesian networks found by each model ranked by their BIC scores in Figure 5. D-VAE generally found better Bayesian networks than S-VAE. The best Bayesian network found by D-VAE achieved a BIC of -11125.75, which is better than the best network in the training set with a BIC of -11141.89 (a higher BIC score is better). Note that BIC is in log scale, thus the probability of our found network to explain the data is actually 1E7 times larger than that of the best training network. For reference, the true Bayesian network used to generate the Asia data has a BIC of -11109.74. Although we did not exactly find the true network, our found network was close to it and outperformed all 180,000 training networks. Our experiments show that searching in an embedding space is a promising direction for Bayesian network structure learning.

4.4 Latent space visualization

In this experiment, we visualize the latent spaces of the VAE models to get a sense of their smoothness.

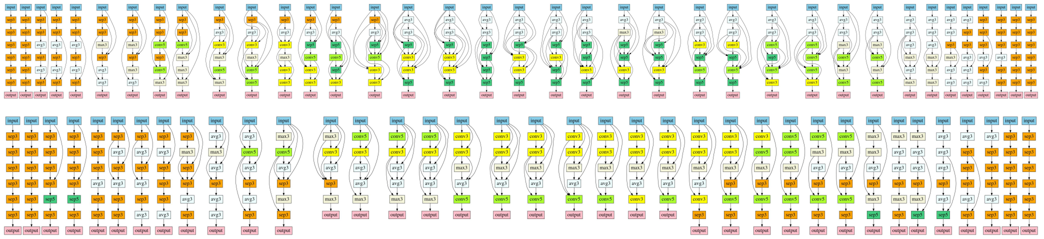

Figure 6: Great circle interpolation starting from a point and returning to itself. Upper: D-VAE. Lower: S-VAE.

For neural architectures, we visualize the decoded architectures from points along a great circle in the latent space [65] (slerp). We start from the latent embedding of a straight network without skip connections. Imagine this latent embedding as a point on the surface of a sphere (visualize the earth). We randomly pick a great circle starting from this point and returning to itself around the sphere. Along this circle, we evenly pick 35 points and visualize their decoded neural architectures in Figure 6. As we can see, both D-VAE and S-VAE show relatively smooth interpolations by changing only a few node types or edges each time. Visually speaking, S-VAE's structural changes are even smoother. This is because S-VAE treats DAGs as strings, thus tending to embed DAGs with few differences in string representations to similar regions of the latent space without considering their computational differences (see Appendix J for more discussion of this problem). In contrast, D-VAE models computations, and focuses more on the smoothness w.r.t. computation rather than structure.

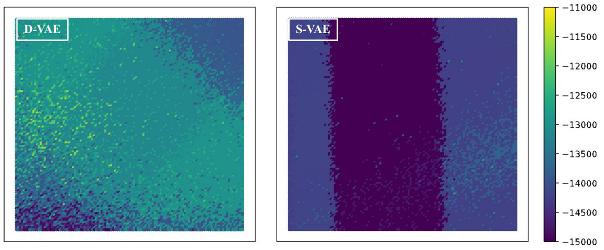

Figure 7: Visualizing a principal 2-D subspace of the latent space.

For Bayesian networks, we directly visualize the BIC score distribution of the latent space. To do so, we reduce its dimensionality by choosing a 2-D subspace spanned by the first two principal components of the training data's embeddings. In this low-dimensional subspace, we compute the BIC scores of all the points evenly spaced within a $[-0.3, 0.3]$ grid and visualize the scores using a colormap in Figure 7. As we can see, D-VAE seems to better differentiate high-score points from low-score ones and shows more smoothly changing BIC scores. In comparison, S-VAE shows sharp boundaries and seems to mix high-score and low-score points more severely. The smoother latent space might be the key reason for the better Bayesian optimization performance with D-VAE. Furthermore, we notice that D-VAE's 2-D latent space is brighter; one explanation is the two principal components of D-VAE explain more variance (59%) of training data than those of S-VAE (17%). Thus, along the two principal components of S-VAE we will see less points from the training distribution. These out-of-distribution points tend to decode to not very good Bayesian networks, thus are darker. This also indicates that D-VAE learns a more compact latent space.

5 Conclusion

In this paper, we have proposed D-VAE, a GNN-based deep generative model for DAGs. D-VAE uses a novel asynchronous message passing scheme to encode a DAG respecting its partial order, which explicitly models the computations on DAGs. By performing Bayesian optimization in D-VAE's latent spaces, we offer promising new directions to two important problems, neural architecture search and Bayesian network structure learning. We hope D-VAE can inspire more research on studying DAGs and their applications in the real world.

Acknowledgments

MZ, ZC and YC were supported by the National Science Foundation (NSF) under award numbers III-1526012 and SCH-1622678, and by the National Institute of Health under award number 1R21HS024581. SJ and RG were supported by the NSF under award numbers IIA–1355406, IIS–1845434, and OAC–1940224. The authors would like to thank Liran Wang for the helpful discussions.

References

[1] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[2] Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

[3] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *International Conference on Machine Learning*, pages 1945–1954, 2017.

[4] Matt J Kusner and José Miguel Hernández-Lobato. GANs for sequences of discrete elements with the Gumbel-softmax distribution. *arXiv preprint arXiv:1611.04051*, 2016.

[5] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. TerpreT: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

[6] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

[7] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[8] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[9] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082*, 2014.

[10] Hanjun Dai, Yingtao Tian, Bo Dai, Steven Skiena, and Le Song. Syntax-Directed Variational Autoencoder for Structured Data. *arXiv preprint arXiv:1802.08786*, 2018.

[11] David Weininger. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988.

[12] Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders. *arXiv preprint arXiv:1802.03480*, 2018.

[13] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. In *International Conference on Machine Learning*, pages 5694–5703, 2018.

[14] Nicola De Cao and Thomas Kipf. MolGAN: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

[15] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. NetGAN: Generating Graphs via Random Walks. *arXiv preprint arXiv:1803.00816*, 2018.

[16] Tengfei Ma, Jie Chen, and Cao Xiao. Constrained generation of semantically valid graphs via regularizing variational autoencoders. In *Advances in Neural Information Processing Systems*, pages 7113–7124, 2018.

[17] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *Proceedings of the 35th International Conference on Machine Learning*, pages 2323–2332, 2018.

[18] Qi Liu, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. Constrained graph variational autoencoders for molecule design. *arXiv preprint arXiv:1805.09076*, 2018.

[19] Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *Advances in Neural Information Processing Systems*, pages 6412–6422, 2018.

[20] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232, 2015.

[21] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[22] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[23] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023, 2016.

[24] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[25] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[26] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, pages 5165–5175, 2018.

[27] Muhan Zhang and Yixin Chen. Inductive matrix completion based on graph neural networks. *arXiv preprint arXiv:1904.12058*, 2019.

[28] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[29] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[30] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

[31] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.

[32] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[33] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[34] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.

[35] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric Xing. Neural architecture search with Bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, 2018.

[36] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[37] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[38] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[39] Renqian Luo, Fei Tian, Tao Qin, En-Hong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, 2018.

[40] C Chow and Cong Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

[41] Tian Gao, Kshitij Fadnis, and Murray Campbell. Local-to-global Bayesian network structure learning. In *International Conference on Machine Learning*, pages 1193–1202, 2017.

[42] Tian Gao and Dennis Wei. Parallel Bayesian network structure learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1685–1694, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/gao18b.html`.

[43] Dominik Linzner and Heinz Koeppl. Cluster Variational Approximations for Structure Learning of Continuous-Time Bayesian Networks from Incomplete Data. In *Advances in Neural Information Processing Systems*, pages 7891–7901, 2018.

[44] David Maxwell Chickering. Learning Bayesian networks is NP-complete. In *Learning from data*, pages 121–130. Springer, 1996.

[45] Ajit P. Singh and Andrew W. Moore. Finding Optimal Bayesian Networks by Dynamic Programming, 2005.

[46] Changhe Yuan, Brandon Malone, and Xiaojian Wu. Learning Optimal Bayesian Networks Using A* Search. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Three*, IJCAI'11, pages 2186–2191. AAAI Press, 2011. ISBN 978-1-57735-515-1. doi: 10.5591/978-1-57735-516-8/IJCAI11-364. URL `http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-364`.

[47] Changhe Yuan and Brandon Malone. Learning Optimal Bayesian Networks: A Shortest Path Perspective. *Journal of Artificial Intelligence Research*, 48(1):23–65, October 2013. ISSN 1076-9757. URL `http://dl.acm.org/citation.cfm?id=2591248.2591250`.

[48] Do Chickering, Dan Geiger, and David Heckerman. Learning Bayesian networks: Search methods and experimental results. In *Proceedings of Fifth Conference on Artificial Intelligence and Statistics*, pages 112–128, 1995.

[49] Tomi Silander, Janne Leppä-aho, Elias Jääsaari, and Teemu Roos. Quotient Normalized Maximum Likelihood Criterion for Learning Bayesian Network Structures. In *International Conference on Artificial Intelligence and Statistics*, pages 948–957, 2018.

[50] Xun Zheng, Bryon Aragam, Pradeep K Ravikumar, and Eric P Xing. DAGs with NO TEARS: Continuous optimization for structure learning. In *Advances in Neural Information Processing Systems*, pages 9472–9483, 2018.

[51] Yue Yu, Jie Chen, Tian Gao, and Mo Yu. DAG-GNN: DAG Structure Learning with Graph Neural Networks. *arXiv preprint arXiv:1904.10098*, 2019.

[52] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[53] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[54] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[55] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[56] Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

[57] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[58] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[59] François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, pages 1610–02357, 2017.

[60] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[61] Marco Scutari. Learning Bayesian Networks with the bnlearn R Package. *Journal of Statistical Software, Articles*, 35(3):1–22, 2010. ISSN 1548-7660. doi: 10.18637/jss.v035.i03. URL `https://www.jstatsoft.org/v035/i03`.

[62] Steffen L Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 157–224, 1988.

[63] Edward Snelson and Zoubin Ghahramani. Sparse Gaussian processes using pseudo-inputs. In *Advances in neural information processing systems*, pages 1257–1264, 2006.

[64] David Ginsbourger, Rodolphe Le Riche, and Laurent Carraro. Kriging is well-suited to parallelize optimization. In *Computational intelligence in expensive optimization problems*, pages 131–162. Springer, 2010.

[65] Tom White. Sampling generative networks. *arXiv preprint arXiv:1609.04468*, 2016.

[66] Marc-André Zöller and Marco F Huber. Survey on automated machine learning. *arXiv preprint arXiv:1904.12054*, 2019.

[67] Jonas Mueller, David Gifford, and Tommi Jaakkola. Sequence to better sequence: continuous revision of combinatorial structures. In *International Conference on Machine Learning*, pages 2536–2544, 2017.

[68] Nicolo Fusi, Rishit Sheth, and Melih Elibol. Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*, pages 3352–3361, 2018.

[69] Benjamin Yackley and Terran Lane. Smoothness and Structure Learning by Proxy. In *International Conference on Machine Learning*, 2012.

[70] Blake Anderson and Terran Lane. Fast Bayesian network structure search using Gaussian processes. 2009. Available at https://www.cs.unm.edu/ treport/tr/09-06/paper.pdf.

[71] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Appendices

A More Related Work

Both neural architecture search (NAS) and Bayesian network structure learning (BNSL) are subfields of AutoML. See Zöller and Huber [66] for a survey. We have given a brief overview of NAS and BNSL in Section 2. Below we discuss several works most related to our work in more detail.

Luo et al. [39] proposed a novel NAS approach called Neural Architecture Optimization (NAO). The basic idea is to jointly learn an encoder-decoder between networks and a *continuous* space, and also a performance predictor f that maps the continuous representation of a network to its performance on a given dataset; then they perform two or three iterations of gradient descent on f to find better architectures in the continuous space, which are then decoded to real networks to evaluate. This methodology is similar to that of Gómez-Bombarelli et al. [2] and Jin et al. [17] for molecule optimization; also similar to Mueller et al. [67] for slightly revising a sentence.

There are several key differences comparing to our approach. First, NAO uses strings (e.g. "node-2 conv 3x3 node1 max-pooling 3x3") to represent neural architectures, whereas we directly use graph representations, which is more natural and generally applicable to other graphs such as Bayesian network structures. Second, NAO uses supervised learning instead of unsupervised learning, which means it needs to first evaluate a considerable amount of randomly sampled graphs on a typically large dataset (e.g. train many neural networks), and use these results to supervise the training of the autoencoder. Given a new dataset, the autoencoder needs to be completely retrained. In contrast, we train our variational autoencoder in a fully unsupervised manner, so the model is of general purposes.

Fusi et al. [68] proposed a novel AutoML algorithm also using model embedding, but with a matrix factorization approach. They first construct a matrix of performances of thousands of ML pipelines on hundreds of datasets; then they use a probabilistic matrix factorization to get the latent representations of the pipelines. Given a new dataset, Bayesian optimization with the expected improvement heuristic is used to find the best pipeline. This approach only allows us to choose from predefined off-the-shelf ML models, hence its flexibility is somewhat limited.

Kandasamy et al. [35] use Bayesian optimization for NAS; they define a kernel that measures the similarities between networks by solving an optimal transport problem, and in each iteration, they use some evolutionary heuristics to generate a set of candidate networks based on making small modifications to existing networks, and use expected improvement to choose the next one to evaluate. This work is similar to ours in terms of the application of Bayesian optimization. However, defining a kernel to measure the similarities between discrete structures is a non-trivial problem. In addition, the discrete search space is heuristically extrapolated near existing architectures, which makes the search essentially local. In contrast, we directly fit a Gaussian process over the entire continuous latent space, enabling more global optimization.

Using Gaussian process (GP) for Bayesian network structure learning has also been studied before. Yackley and Lane [69] analyzed the smoothness of BDe score, showing that a local change (e.g. adding an edge) can change the score by at most $\mathcal{O}(\log n)$, where n is the number of training points. They proposed to use GP as a proxy for the score to accelerate the search. Anderson and Lane [70] used GP to model the BDe score, and showed that the probability of improvement is better than that of using hill climbing to guide the local search. However, these methods still heuristically and locally operate in the discrete space, whereas our latent space makes both local and global methods such as gradient descent and Bayesian optimization applicable in a principled manner.

Recently, Zheng et al. [50] also proposed a continuous optimization approach for BNSL, where the decision variable is the adjacency matrix of the DAG and the objective function is least square loss based on linear structural equation modeling (SEM); acyclicity is ensured by a novel equality constraint. Yu et al. [51] generalize this approach to nonlinear SEM using VAE. We highlight several key differences from our approach: 1) they directly optimize the adjacency matrix, but we optimize a learned latent representation of the DAGs; 2) they ensure acyclicity by enforcing an equality constraint, but for our method acyclicity is automatically guaranteed by the decoding process; 3) They use gradient-based optimization, but we use global black-box optimization. 4) Their methods are specific to BNSL, but ours applies to general DAG optimization; 5) The usage of VAE is totally different: they use VAE as a generative model for the data (sampled from the DAG), but we use VAE as a generative model for DAGs.

B Graph Structure vs. Computation vs. Function

In Section 3 we defined computation. Here we discuss the differences among DAG structure, computation and function. A DAG structure with operations on nodes define a computation, and two DAGs can define the same computation, which are illustrated in Figure 1. A computation defines a function, and two computations can define the same function. For example, computation $C_1 := x + 1 - 1$ defines a function $f(x) = x$, while computations $C_2 := x - 1 + 1$ and $C_3 := x$ also define the function $f(x) = x$. However, C_1, C_2 and C_3 are different computations. In other words, a computation is (informally speaking) a process which focuses on the course of how the input is processed into the output, while a function is a mapping from input to output which does not care about the process.

Sometimes, the same computation can also define different functions, e.g., two identical neural architectures will represent different functions given they are trained differently (since the weights of their layers will be different). In D-VAE, we model computations instead of functions, since 1) modeling functions is much harder than modeling computations (requires understanding the semantic meaning of each operation, such as the cancelling out of $+$ and $-$), and 2) modeling functions additionally requires knowing the parameters of some operations, which are unknown before training.

Note also that in Definition 1, we only allow one single input signal. But in real world a computation sometimes has multiple initial input signals. However, the case of multiple input signals can be reduced to the single input case by adding an initial assignment operation that assigns the combined input signal to their corresponding next-level operations. For ease of presentation, we uniformly assume single input throughout the paper.

C Proof of Theorem 1

Proof. Let v_1 be the starting node with no predecessors. By assumption, v_1 is the single starting node no matter how we permute the nodes of the input DAG. For v_1, the aggregation function $\mathcal{A}$ always outputs a zero vector. Thus, $\mathbf{h}_{v_1}^{\text{in}}$ is invariant to node permutations. Subsequently, the hidden state $\mathbf{h}_{v_1} = \mathcal{U}(\mathbf{x}_{v_1}, \mathbf{h}_{v_1}^{\text{in}})$ is also invariant to node permutations.

Now we prove the theorem by structural induction. Consider node v. Suppose for every predecessor u of v, the hidden state $\mathbf{h}_u$ is invariant to node permutations. We will show that $\mathbf{h}_v$ is also invariant to node permutations. Notice that in (3), the output $\mathbf{h}_v^{\text{in}}$ by $\mathcal{A}$ is invariant to node permutations, since $\mathcal{A}$ is invariant to the order of its inputs $\mathbf{h}_u$, and all $\mathbf{h}_u$ are invariant to node permutations. Subsequently, node v's hidden state $\mathbf{h}_v = \mathcal{U}(\mathbf{x}_v, \mathbf{h}_v^{\text{in}})$ is invariant to node permutations. By induction, we know that every node's hidden state is invariant to node permutations, including the ending node's hidden state. Thus, the D-VAE encoder is invariant to node permutations. □

D Proof of Theorem 2

Proof. Suppose there is an arbitrary input signal x fed to the starting node v_1. For convenience, we will use $C_i(x)$ to denote the output signal at vertex v_i, where C_i represents the composition of all the operations along the paths from v_1 to v_i.

For the starting node v_1, remember we feed a fixed $\mathbf{h}_{v_1}^{\text{in}} = \mathbf{0}$ to (2), thus $\mathbf{h}_{v_1}$ is also fixed. Since C_1 also represents a fixed input operation, we know that the mapping from C_1 to $\mathbf{h}_{v_1}$ is injective. Now we prove the theorem by induction. Assume the mapping from C_j to $\mathbf{h}_{v_j}$ is injective for all $1 \leq j < i$. We will prove that the mapping from C_i to $\mathbf{h}_{v_i}$ is also injective.

Let $\phi_j(C_j) = \mathbf{h}_{v_j}$ where ϕ_j is injective. Consider the output signal $C_i(x)$, which is given by feeding $\{C_j(x) : v_j \to v_i\}$ to o_i. Thus,

$$C_i(x) = o_i(\{C_j(x) : v_j \to v_i\}). \tag{6}$$

In other words, we can write C_i as

$$C_i = \psi(o_i, \{C_j : v_j \to v_i\}), \tag{7}$$

where ψ is an injective function used for defining the composite computation C_i based upon o_i and $\{C_j : v_j \to v_i\}$. Note that $\{C_j : v_j \to v_i\}$ can be either unordered or ordered depending on the operation o_i. For example, if o_i is some symmetric operations such as adding or multiplication, then $\{C_j : v_j \to v_i\}$ can be unordered. If o_i is some operation like subtraction or division, then $\{C_j : v_j \to v_i\}$ must be ordered.

With (2) and (3), we can write the hidden state $\mathbf{h}_{v_i}$ as follows:

$$\begin{aligned}\mathbf{h}_{v_i} &= \mathcal{U}(\mathbf{x}_{v_i}, \mathcal{A}(\{\mathbf{h}_{v_j} : v_j \to v_i\})) \\ &= \mathcal{U}(O(o_i), \mathcal{A}(\{\phi_j(C_j) : v_j \to v_i\})), \end{aligned} \tag{8}$$

where O is the injective one-hot encoding function mapping o_i to $\mathbf{x}_{v_i}$. In the above equation, $\mathcal{U}, O, \mathcal{A}, \phi_j$ are all injective. Since the composition of injective functions is injective, there exists an injective function φ so that

$$\mathbf{h}_{v_i} = \varphi(o_i, \{C_j : v_j \to v_i\}). \tag{9}$$

Then combining (7) we have:

$$\begin{aligned}\mathbf{h}_{v_i} &= \varphi \circ \psi^{-1}\psi(o_i, \{C_j : v_j \to v_i\}) \\ &= \varphi \circ \psi^{-1}(C_i). \end{aligned} \tag{10}$$

$\varphi \circ \psi^{-1}$ is injective since the composition of injective functions is injective. Thus, we have proved that the mapping from C_i to $\mathbf{h}_{v_i}$ is injective. $\square$

E Modifications for Encoding Neural Architectures

According to Theorem 2, to ensure D-VAE injectively encodes computations, we need the aggregation function $\mathcal{A}$ to be injective. Remember $\mathcal{A}$ takes the multiset $\{\mathbf{h}_u : u \to v\})$ as input. If the order of its elements does not matter, then the gated sum in (4) can model this injective multiset function without issues. However, if the order matters (i.e., permuting the elements of $\{\mathbf{h}_u : u \to v\}$ makes $\mathcal{A}$ output different results), we need a different aggregation function that can encode such orders.

Whether the order should matter for $\mathcal{A}$ depends on whether the input order matters for the operations o (see the proof for Theorem 2 for more details). For example, if multiple previous layers' outputs are summed or averaged as the input to a next layer in the neural networks, then $\mathcal{A}$ can be modeled by the gated sum in (4) as the order of inputs does not matter. However, if these outputs are concatenated as the next layer's input, then the order does matter. In our experiments, the neural architectures use the second way to aggregate outputs from previous layers. The order of concatenation depends on a global order of the layers in a neural architecture. For example, if layer-2 and layer-4's outputs are input to layer-5, then layer-2's output will be before layer-4's output in their concatenation.

Since the gated sum in (4) can only handle the unordered case, we can slightly modify (4) in order to make it order-aware thus more suitable for our neural architectures. Our scheme is as follows:

$$\mathbf{h}_v^{\text{in}} = \sum_{u \to v} g(\text{Concat}(\mathbf{h}_u, \mathbf{x}_{\text{uid}})) \odot m(\text{Concat}(\mathbf{h}_u, \mathbf{x}_{\text{uid}})), \tag{11}$$

where $\mathbf{x}_{\text{uid}}$ is the one-hot encoding of layer u's global ID (1,2,3,...). Such an aggregation function respects the concatenation order of the layers. We empirically observed that this aggregation function can increase D-VAE's performance on neural architectures compared to the plain aggregation function (4). However, even using (4) still outperformed all baselines.

F Modifications for Encoding Bayesian Networks

We also make some modifications when encoding Bayesian networks. One modification is that the aggregation function (4) is changed to:

$$\mathbf{h}_v^{\text{in}} = \sum_{u \to v} g(\mathbf{x}_u) \odot m(\mathbf{x}_u). \tag{12}$$

Compared to (4), we replace $\mathbf{h}_u$ with the node type feature $\mathbf{x}_u$. This is due to the differences between computations on a neural architecture and on a Bayesian network. In a neural network, the signal flow follows the network architecture, where the output signal of a layer is fed as the input signals to its succeeding layers. Also in a neural network, what we are interested in is the result output by the final layer. In contrast, for a Bayesian network, the graph represents a set of conditional dependencies among variables instead of a computational flow. In particular, for Bayesian network structure learning, we are often concerned about computing the (log) marginal likelihood score of a dataset given a graph structure, which is often decomposed into individual variables given their parents (see Definition 18.2 in Koller and Friedman [1]). For example, in Figure 8, the overall score can be decomposed into $s(X_1) + s(X_2) + s(X_3 \mid X_1, X_2) + s(X_4) + s(X_5 \mid X_3, X_4)$. To compute the score $s(X_5 \mid X_3, X_4)$ for X_5, we only need the values of X_3 and X_4; its grandparents X_1 and X_2 should have no influence on X_5. Based on this intuition, when computing the hidden state of a node, we use the features $\mathbf{x}_u$ of its parents u instead of $\mathbf{h}_u$, which "d-separates" the node from its grandparents. For the update function, we still use (5).

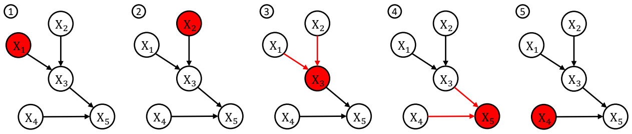

Figure 8: An example Bayesian network and its encoding.

Also based on the decomposibility of the score, we make another modification for encoding Bayesian networks by using the sum of all node states as the final output state instead of only using the ending node state. Similarly, when decoding Bayesian networks, the graph state $\mathbf{h}_G := \sum_{j=1,\dots,i-1} \mathbf{h}_{v_j}$.

Note that the combination of (12) and (5) can injectively model the conditional dependence between v and its parents u. In addition, using summing can model injective set functions [53, Lemma 5]. Therefore, the above encoding scheme is able to **injectively encode** the complete **conditional dependencies** of a Bayesian network, thus also the overall score function s of the network.

G Advantages of Encoding Computations in DAG Optimization

Here we discuss why D-VAE's ability to injectively encode computations (Theorem 2) is of great benefit to performing DAG optimization in the latent space. Firstly, our target is to find a DAG that achieves high performance (e.g., accuracy of neural network, BIC score of Bayesian network) on a given dataset. The performance of a DAG is directly related to its computation. For example, given the same set of layer parameters, two neural networks with the same computation will have the same performance on a given test set. Since D-VAE encodes computations instead of structures, it allows **embedding DAGs with similar performances to the same regions** in the latent space, rather than embedding DAGs with merely similar structure patterns to the same regions. Subsequently, the latent space can be **smooth w.r.t.** *performance* instead of *structure*. Such smoothness can greatly facilitate searching for high-performance DAGs in the latent space, since similar-performance DAGs tend to locate near each other in the latent space instead of locating randomly, and modeling a smoothly-changing performance surface is much easier.

Note that Theorem 2 is a necessary condition for the latent space to be smooth w.r.t. performance, because if D-VAE cannot injectively encode computations, it might map two DAGs representing completely different computations to the same encoding, making this point of the latent space arbitrarily unsmooth. Although there yet is no theoretical guarantee that the latent space must be smooth w.r.t. DAGs' performances, we do empirically observe that the predictive performance and Bayesian optimization performance of D-VAE's latent space are significantly better than those of baselines, which is indirect evidence that D-VAE's latent space is smoother w.r.t. performance. Our visualization results also confirm the smoothness. See Section 4.2, 4.3, 4.4 for details.

H More Details about Neural Architecture Search

We use the efficient neural architecture search (ENAS)'s software [33] to generate the training and testing neural architectures. With these seed architectures, we can train a VAE model and thus search for new high-performance architectures in the latent space.

ENAS alternately trains two components: 1) an RNN-based controller which is used to propose new architectures, and 2) the shared weights of the proposed architectures. It uses a weight-sharing scheme to obtain a quick but rough estimate of how good an architecture is. That is, it forces all the proposed architectures to use the same set of shared weights, instead of fully training each neural network individually. It assumes that an architecture with a high validation accuracy using the shared weights (i.e., the weight-sharing accuracy) is more likely to have a high test accuracy after fully retraining its weights from scratch.

We first run ENAS in the macro space (Section 2.3 of [33]) for 1000 epochs with 20 architectures proposed in each epoch. For all the proposed architectures excluding the first 1000 burn-in ones, we evaluate their weight-sharing accuracies using the shared weights from the last epoch. We further split the data into 90% training and 10% held-out test sets. Then our task becomes to train a VAE on the training neural architectures, and then generate new high-performance architectures from the latent space based on Bayesian optimization. Note that our target performance measure here is the weight-sharing accuracy, not the true validation/test accuracy after fully retraining the architecture. This is because the weight-sharing accuracy takes around 0.5 second to evaluate, while fully training a network takes over 12 hours. In consideration of our limited computational resources, we choose the weight-sharing accuracy as our optimization target in the Bayesian optimization experiments.

After the Bayesian optimization finds a final set of architectures with high weight-sharing accuracies, we will fully train them to evaluate their true test accuracies on CIFAR-10. To fully train an architecture, we follow the original setting of [33] to train each architecture on CIFAR-10's training set for 310 epochs, and report the last epoch's net's test accuracy. See [33, Section 3.2] for details.

Due to our constrained computational resources, we choose not to perform Bayesian optimization to optimize the true validation accuracy (obtained by fully training a neural network), which would be a more principled way for searching neural architectures. Nevertheless, we describe its procedure here for future explorations: After training the D-VAE, we have no architectures at all to initialize a Gaussian process regression on the true validation accuracy. Thus, we need to randomly pick up some points in the latent space, decode them into neural architectures, and get their true validation accuracies after full training. Then with these initial points, we start the Bayesian optimization similarly to Section 4.3, with the optimization target replaced by the true validation accuracy. Finally, we will find a set of architectures with the highest true validation accuracies, and report their true test accuracies. This experiment will take much longer time (possibly months of GPU time). Thus, it is very necessary to train multiple models parallelly on many machines, like [28] does.

One might wonder why we train another generative model after we already have ENAS. Firstly, ENAS is a task-specific supervised model. It leverages the validation accuracy signals of the target task to guide the generation of new architectures based on reinforcement learning. For any new NAS task, ENAS needs to be completely retrained. In contrast, D-VAE is unsupervised. Once trained, it can be applied to NAS tasks targeting different datasets. For example, although we use the neural architectures generated by the ENAS targeting CIFAR-10 to train our D-VAE, once trained, we can use D-VAE's latent space to search neural architectures suitable for CV tasks other than CIFAR-10. In contrast, the trained ENAS is not applicable to other tasks since it uses supervised signals from CIFAR-10. In other words, ENAS is only used to generate a set of seed architectures for training D-VAE, and is not necessary for D-VAE. For example, we may also train D-VAE using the recent NAS-Bench-101 dataset[1], which we leave for future work. Another exclusive advantage of D-VAE is that it provides a way to learn neural architecture embeddings, which can be used for downstream tasks such as visualization and classification, etc.

In the Bayesian optimization experiments (Section 4.3), the best architecture found by D-VAE achieves a test accuracy of 94.80% on CIFAR-10. Although not outperforming state-of-the-art NAS techniques such as NAONET which has an error rate of 2.11%, our architecture only contains 3 million parameters compared to NAONET + CUTOUT which has 128 million parameters [39]. In addition, the search space is different between the two approaches: we directly search 6-layer CNNs, while NAONET searches 5-layer CNN cells and stacks the found cell for 6 times to construct a CNN, thus having much deeper final networks. Finally, NAONET used 200 GPUs to fully train 1,000 architectures for 1 day, and added 4 times more filters after optimization. In comparison, we only used 1 GPU to evaluate the weight-sharing accuracy, and did not add filters to boost the performance. We emphasize that the main purpose of the paper is to introduce a DAG generative model that is capable of DAG optimization, rather than to break NAS records.

---

[1] https://github.com/google-research/nasbench

I More Details about Bayesian Network Structure Learning

We consider a small synthetic problem called Asia [62] as our target Bayesian network structure learning problem. The Asia dataset is composed of 5,000 samples, each is generated by a true network with 8 binary variables[2]. Bayesian Information Criteria (BIC) score is used to evaluate how well a Bayesian network fits the 5,000 samples. To train a VAE model to generate Bayesian network structures, we sample 200,000 random 8-node Bayesian networks from the `bnlearn` package [61] in R, which are split into 90% training and 10% testing sets. Our task is to train a VAE model on the training Bayesian networks, and search in the latent space for Bayesian networks with high BIC scores using Bayesian optimization. In this task, we consider a simplified case where the topological order of the true network is known – we let the sampled training and test Bayesian networks have topological orders consistent with the true network of Asia. This is a reasonable assumption for many practical applications, e.g., when the variables have a temporal order [1]. When sampling a network, the probability of a node having an edge with a previous node (as specified by the order) is set to the default option $2/(k-1)$, where $k = 8$ is the number of nodes, which results in sparse graphs where the number of edges is in the same order of the number of nodes.

J Baselines

As discussed in the related work, there are other types of graph generative models that can potentially work for DAGs. We explore three possible approaches and contrast them with D-VAE.

S-VAE. The S-VAE baseline treats a DAG as a sequence of node strings, which we call string-based variational autoencoder (S-VAE). In S-VAE, each node is represented as the one-hot encoding of its type number concatenated with a 0/1 indicator vector indicating which previous nodes have directed edges to it (i.e., a column of the adjacency matrix). For example, suppose there are two node types and five nodes, then node 4's string "0 1, 0 1 1 0 0" means this node has type 2, and has directed edges from previous nodes 2 and 3. S-VAE leverages a standard GRU-based RNN variational autoencoder [56] on the topologically sorted node sequences, with each node's string treated as its input bit vector.

GraphRNN. One similar generative model is GraphRNN [13]. Different from S-VAE, it further decomposes an adjacency column into entries and generates the entries one by one using another edge-level GRU. GraphRNN is a pure generative model which does not have an encoder, thus cannot optimize DAG performance in a latent space. To compare with GraphRNN, we equip it with S-VAE's encoder and use it as another baseline. Note that the original GraphRNN feeds nodes using a BFS order (for undirected graphs), yet we find that it is much worse than using a topological order here. Note also that although GraphRNN seems more expressive than S-VAE, we find that in our applications GraphRNN tends to have more severe overfitting and generates less diverse DAGs.

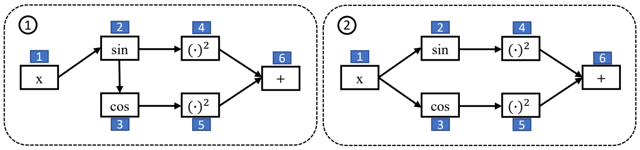

Figure 9: Two bits of change in the string representations can completely change the computational purpose.

Both GraphRNN and S-VAE treat DAGs as bit strings and use RNNs to model them. This representation has several drawbacks. Firstly, since the topological ordering is often not unique for a DAG, there might be multiple string representations for the same DAG, which all result in different encoded representations. This will violate the permutation invariance in Theorem 1. Secondly, the string representations can be very brittle in terms of modeling DAGs' computational purposes. In Figure 9, the left and right DAGs' string representations are only different by two bits, i.e., the edge (2,3) in the left is changed to the edge (1,3) in the right. However, the two bits of change in structure greatly changes the signal flow, which makes the right DAG always output 1. In S-VAE and GraphRNN, since the bit representations of the left and right DAGs are very similar, they are highly likely to be encoded to similar latent vectors. In particular, the only difference between encoding the left and right DAGs is that, for node 3, the encoder RNN will read an adjacency column of [0, 1, 0, 0, 0, 0] in the left, and read [1, 0, 0, 0, 0, 0] in the right, while all the remaining encoding is exactly the same. By embedding two DAGs serving very different computational purposes to the same region of the latent space, S-VAE and GraphRNN tend to have less smooth latent spaces which make optimization

---

[2] http://www.bnlearn.com/documentation/man/asia.html

on them more difficult. In contrast, D-VAE can better differentiate such subtle differences, as the change of edge (2,3) to (1,3) completely changes what aggregated message node 3 receives in D-VAE (hidden state of node 2 vs. hidden state of node 1), which greatly affects node 3 and all its successors' feature learning.

GCN. The graph convolutional network (GCN) [22] is one representative graph neural network with a simultaneous message passing scheme. In GCN, all the nodes take their neighbors' incoming messages to update their own states simultaneously instead of following an order. After message passing, the summed node states is used as the graph state. We include GCN as the third baseline. Since GCN can only encode graphs, we equip GCN with D-VAE's decoder to make it a VAE model. For neural architectures, we searched the number of message passing layers from 1 to 5. We found that if we only use 1 message passing layer, the reconstruction accuracy is only around 5%. And if we use 2 or more layers, the reconstruction accuracy gets around 97% stably but never reaches nearly 100% like other models. This demonstrates GCN's limitation of only encoding local substructures for neural architectures. The final GCN model uses 3 message passing layers. For Bayesian networks, we find 1 layer is enough to reach a 99% reconstruction accuracy, which is reasonable since Bayesian networks are naturally local. We report a GCN model using 2 message passing layers.

Using GCN as the encoder can ensure permutation invariance, since node ordering does not matter in GCN. However, GCN's message passing focuses on propagating the neighboring nodes' features to each center node to encode the **local substructure pattern** around each node. In comparison, D-VAE's message passing simulates how the computation is performed along the directed paths of a DAG and focuses on encoding the computation. Although learning local structural features is essential for GCN's successes in node classification and graph classification, here in our tasks, modeling the computation represented by the entire graph is much more important than modeling the local features. Encoding only local substructures may also lose important information about the global DAG topology, making it more difficult to reconstruct the DAG.

DeepGMG. DeepGMG [6] is a graph-based graph generative model that uses a simultaneous message passing to learn intermediate node/graph states and uses a similar decoding scheme to D-VAE to generate nodes/edges of a graph sequentially. DeepGMG is originally designed for generating general (undirected) graphs. Several modifications are made to adapt it to our tasks. First, we make it a VAE by equipping it with a 3-layer message passing network as the encoder using its own message passing functions, and use the original generative model as the decoder. Second, we feed in nodes using a topo-order instead of the original random order (and see much improvement). Third, we make the sampled edges in the decoding phase only point to new nodes to ensure acyclicity.

Similar to GCN, DeepGMG's training loss never reaches near zero even with extensive hyperparameter tuning, which again reveals the limitation of simultatenous message passing for encoding DAGs. In comparison, D-VAE can be perfectly trained to near zero loss.

We omit other possible approaches such as GraphVAE [12] and some recent graph-based models [17, 18, 19] etc., either because they lack official code or they target specific graphs (such as molecules) only.

K VAE Training Details

We use the same settings and hyperparameters (where applicable) for all the four models to be as pair as possible. Many hyperparameters are inherited from Kusner et al. [3]. Single-layer GRUs are used in all models requiring recurrent units, with the same hidden state size of 501. We set the dimension of the latent space to be 56 for all models. All VAE models use $\mathcal{N}(\mathbf{0}, \mathbf{I})$ as the prior distribution $p(\mathbf{z})$, and take $q_\phi(\mathbf{z}|G)$ (G denotes the input DAG) to be a normal distribution with a diagonal covariance matrix, whose mean and variance parameters are output by the encoder. The two MLPs used to output the mean and variance parameters are all implemented as single linear layer networks.

For the decoder network of D-VAE, we let $f_{\text{add\_vertex}}$ and $f_{\text{add\_edge}}$ be two-layer MLPs with ReLU nonlinearities, where the hidden layer sizes are set to two times of the input sizes. Softmax activation is used after $f_{\text{add\_vertex}}$, and sigmoid activation is used after $f_{\text{add\_edge}}$. For the gating network g, we use a single linear layer with sigmoid activation. For the mapping function m, we use a linear mapping without activation. The bidirectional encoding discussed in Section 3.4 is enabled for D-VAE on neural architectures, and disabled for D-VAE on Bayesian networks and other models where it gets no better results.

When optimizing the VAE loss, we use ReconstructLoss $+ \alpha$KLDivergence as the loss function. In the original VAE framework, α is set to 1. However, we found that it led to poor reconstruction accuracies, similar to the findings of previous work [3, 10, 17]. Following the implementation of Jin et al. [17], we set $\alpha = 0.005$. Mini-batch SGD with Adam optimizer [71] is used for all models. For neural architectures, we use a batch size of 32 and train all models except DeepGMG for 300 epochs. For Bayesian networks, we use a batch size of 128 and train all models except DeepGMG for 100 epochs. For DeepGMG, we early stop the training at epoch 30 and epoch 5 for neural architectures and Bayesian networks, respectively, in order to avoid numerical instabilities. We use an initial learning rate of 1E-4, and multiply the learning rate by 0.1 whenever the training loss does not decrease for 10 epochs. We use PyTorch to implement all the models.

L SGP Training Details

We use sparse Gaussian process (SGP) regression as the predictive model. We use the open sourced SGP implementation in [3]. Both the training and testing data's performances are standardized according to the mean and std of the training data's performances before feeding to the SGP. And the RMSE and Pearson's r in Table 2 are also calculated on the standardized performances. We use the default Adam optimizer to train the SGP for 100 epochs constantly with a mini-batch size of 1,000 and learning rate of 5E-4.

For neural architectures, we use all the training data to train the SGP. For Bayesian networks, we randomly sample 5,000 training examples each time, due to two reasons: 1) using all the 180,000 examples to train the SGP might not be realistic for a typical scenario where network/dataset is large and evaluating a network is expensive; and 2) we found using a smaller sample of training data results in more stable BO performance due to the less probability of duplicate rows which might result in ill conditioned matrices. Note also that, when training the variational autoencoders, all the training data are used, since the VAE training is purely unsupervised.

M More Experimental Results

M.1 More details on the piror validity experiment

Since different models can have different levels of convergence w.r.t. the KLD loss in (1), their posterior distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$ may have different degrees of alignment with the prior distribution $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. If we evaluate prior validity by sampling from $p(\mathbf{z})$ for all models, we will favor those models that have a higher-level of KLD convergence. To remove such effects and focus purely on models' intrinsic ability to generate valid DAGs, when evaluating prior validity, we apply $\mathbf{z} = \mathbf{z} \odot \text{std}(\mathbf{Z}_{\text{train}}) + \text{mean}(\mathbf{Z}_{\text{train}})$ for each model (where $\mathbf{Z}_{\text{train}}$ are encoded means of the training data by the model), so that the latent vectors are scaled and shifted to the center of the training data's embeddings. If we do not apply such transformations, we find that we can easily control the prior validity results by optimizing for more or less epochs or putting more or less weight on the KLD loss.

For a generated neural architecture to be read by ENAS, it has to pass the following validity checks: 1) It has one and only one starting node (the input layer); 2) It has one and only one ending type (the output layer); 3) Other than the input node, there are no nodes which do not have any predecessors (no isolated paths); 4) Other than the output node, there are no nodes which do not have any successors (no blocked paths); 5) Each node must have a directed edge from the node immediately before it (the constraint of ENAS), i.e., there is always a main path connecting all the nodes; and 6) It is a DAG.

For a generated Bayesian network to be read by `bnlearn` and evaluated on the Asia dataset, it has to pass the following validity checks: 1) It has exactly 8 nodes; 2) Each type in "ASTLBEXD" appears exactly once; and 3) It is a DAG.

Note that the training graphs generated by the original software all satisfy these validity constraints.

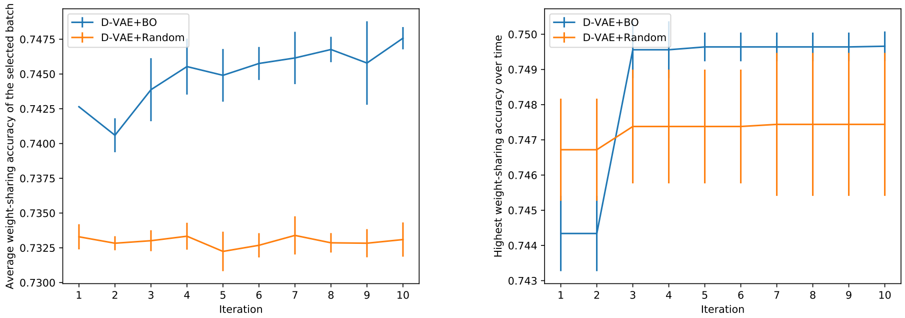

Figure 10: Comparing BO with random search on neural architectures. Left: average weight-sharing accuracy of the selected points in each iteration. Right: highest weight-sharing accuracy of the selected points over time.

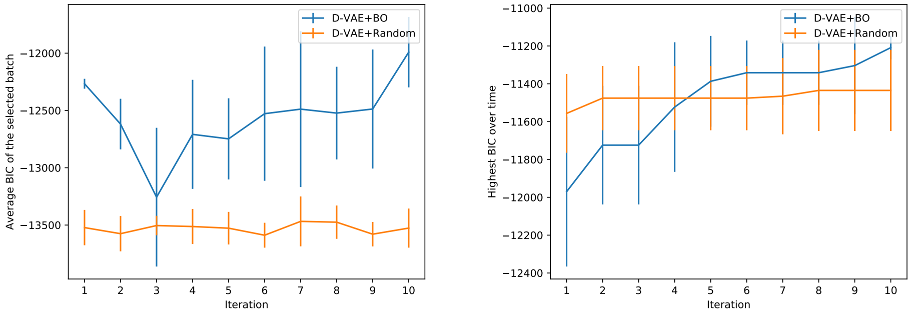

Figure 11: Comparing BO with random search on Bayesian networks. Left: average BIC score of the selected points in each iteration. Right: highest BIC score of the selected points over time.

M.2 Bayesian optimization vs. random search

To validate that Bayesian optimization (BO) in the latent space does provide guidance in searching better DAGs, we compare BO with Random (which randomly samples points from the latent space of D-VAE). Figure 10 and 11 show the results (averaged across 10 trials). In each figure, the left plot shows the average performance of all the points found in each BO round, and the right plot shows the highest performance of all the points found so far. As we can see, BO consistently selects points with better average performance in each round than random search, which is expected. However, for the highest performance results, BO tends to fall behind Random in the initial few rounds. This might be because our batch expected improvement heuristic aims to take advantage of the currently most promising regions by selecting most points of the batch in the same region (exploitation), while Random more evenly explores the entire space (exploration). Nevertheless, BO seems to quickly catch up after a few rounds and shows long-term advantages.

M.3 More D-VAE experiments

D-VAE leverages the proposed asynchronous message passing in both its encoder and decoder. To understand deeper how the asynchronous message passing helps, we add some ablation studies of D-VAE on our neural architecture datasets. Firstly, we replace the asynchronous message passing in D-VAE with simultaneous message passing to construct a D-VAE (SMP) baseline. Secondly, we keep the D-VAE encoder unchanged, and replace the decoder with S-VAE's string-based decoder. The decoder now is a simple RNN with $\mathcal{O}(n)$ complexity instead of the original $\mathcal{O}(n^2)$ complexity, thus is much faster. We name this variant D-VAE (FAST). We compare these two variants with the original D-VAE and S-VAE on our 6-layer neural architectures. The results are shown in Table 3. We can see that D-VAE still in general has the best performance. The D-VAE (SMP) baseline shows inferior reconstruction accuracy due to its nonzero training loss caused by the simultaneous message passing. The D-VAE (FAST) shows similar generative ability to D-VAE. In terms of latent space predictive ability, it is inferior to D-VAE but better than S-VAE. This indicates that, it is beneficial to

Table 3: Generative ability and latent space predictive ability of D-VAE and its variants.

	Generative ability (%)				Predictive ability	
Methods	Accuracy	Validity	Uniqueness	Novelty	RMSE	Pearson's r
D-VAE (SMP)	92.35	99.75	**65.98**	**100.00**	0.455±0.002	0.885±0.001
D-VAE (FAST)	**99.98**	**100.00**	40.53	**100.00**	0.419±0.006	0.905±0.001
D-VAE	**99.96**	**100.00**	37.26	**100.00**	**0.384±0.002**	**0.920±0.001**
S-VAE	**99.98**	**100.00**	37.03	**99.99**	0.478±0.002	0.873±0.001

use asynchronous message passing in both D-VAE's encoder and decoder, rather than only using it to encode DAGs.

Nevertheless, using D-VAE (FAST) allows us to work on deeper neural architectures with much less training time due to its linear decoding complexity. Thus, we repeat our NAS experiments on 12-layer neural architectures. Our final found network has an error rate of 3.88%, comparable to many state-of-the-art NAS results in the macro space such as [33]. We plot our final 12-layer neural architecture in Figure 12.

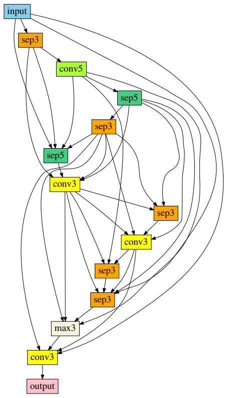

Figure 12: Visualization of the final 12-layer neural architecture found by D-VAE (FAST).

M.4 More visualization results for neural architectures

We randomly pick a neural architecture and use its encoded mean as the starting point. We then generate two random orthogonal directions, and move in the combination of these two directions from the starting point to render a 2-D visualization of the decoded architectures in Figure 13.

M.5 More visualization results for Bayesian networks

We similarly show the 2-D visualization of decoded Bayesian networks in Figure 14. Both D-VAE and S-VAE show smooth latent spaces.

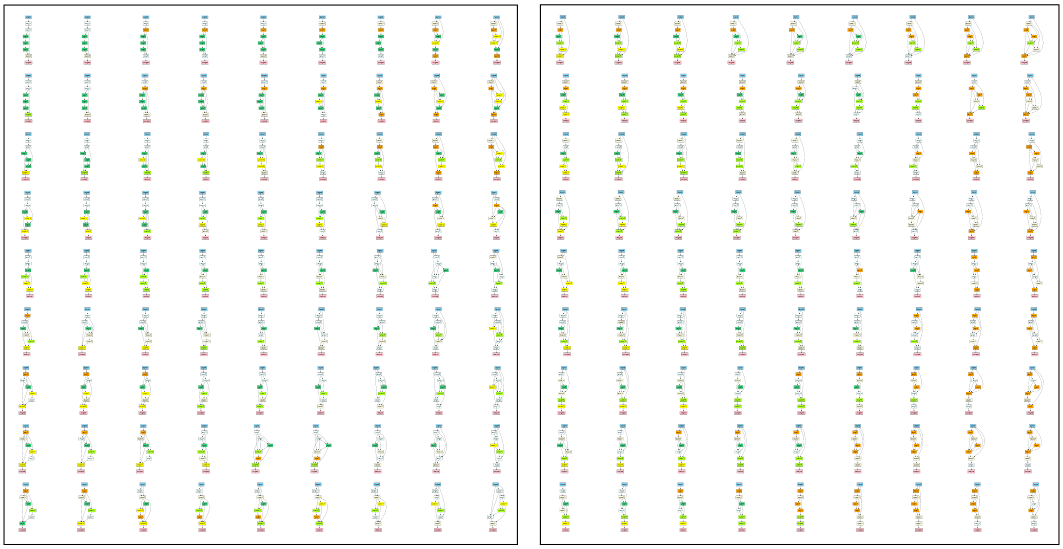

Figure 13: 2-D visualization of decoded neural architectures. Left: D-VAE. Right: S-VAE.

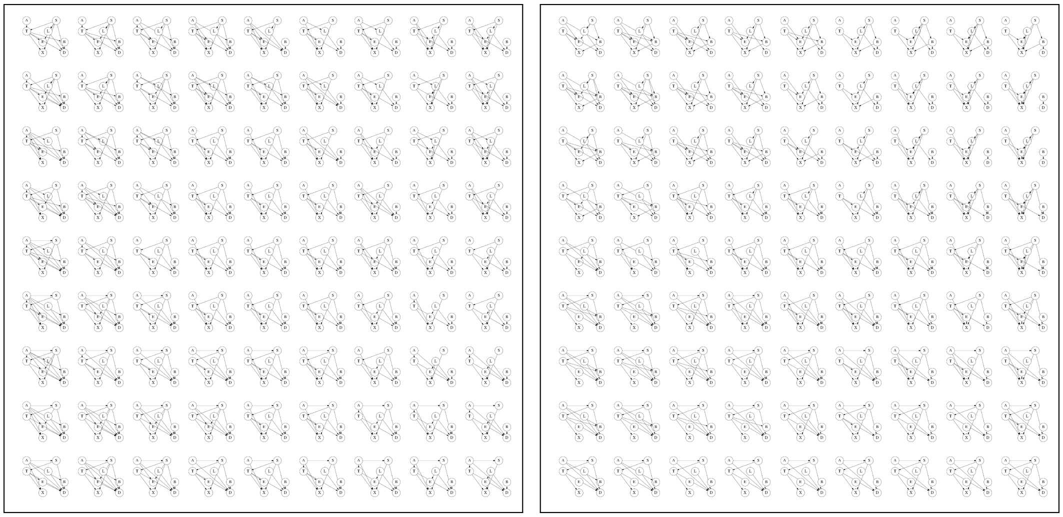

Figure 14: 2-D visualization of decoded Bayesian networks. Left: D-VAE. Right: S-VAE.