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 defined by 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. In machine learning, deep neural networks are DAGs. Although they have achieved remarkable performance on a wide range of learning tasks, tremendous efforts need to be devoted to designing their architectures, which is essentially a DAG optimization task. Similarly, optimizing the connection structures of Bayesian networks is also a critical problem in learning graphical models [1]. DAG optimization is pervasive in other fields as well. For example, in electronic circuit design, engineers need to optimize directed network structures not only to realize target functions, but also to meet specifications such as power usage and operating temperature.

DAGs, as well as other discrete structures, cannot be optimized directly with traditional gradient-based techniques, as gradients are not available. Bayesian optimization, a state-of-the-art black-box optimization technique, requires 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. 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], general graphs [6], etc. In particular, Kusner et al. [3] developed a grammar variational autoencoder (GVAE) 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-behaved space instead of a discrete space. Inspired by this work, we propose to train variational autoencoders for DAGs, and optimize DAG structures in the latent space based on Bayesian optimization.

Existing graph generative models can be classified into three categories: token-based, adjacency-matrix-based, and graph-based approaches. Token-based graph generative models [2, 3, 7] represent a graph as a sequence of tokens (e.g., characters, grammar production rules) and model these sequences based on established RNN modules such as gated recurrent unit (GRU) [8]. Adjacency-matrix-based models [9, 10, 11, 12, 13] generate columns/entries of the adjacency matrix of a graph sequentially or generate the entire adjacency matrix in one shot. Graph-based models [6, 14, 15, 16] iteratively add new nodes or new edges to a graph based on the the existing graph state and node hidden states.

Among the three types, token-based models require task-specific graph grammars such as SMILES for molecules [17], and thus is less general. Adjacency-matrix-based models leverage proxy matrix representations of graphs, and generate graphs through multi-layer perceptrons (MLPs) or RNNs which could be less expressive and less suitable for graphs. On the other hand, graph-based models seem to be more general and natural, since they operate directly on graph structures instead of proxy representations. In addition, the iterative process is driven by the current graph and nodes' states as represented by **graph neural networks (GNNs)**, which have already shown their powerful graph feature learning ability on various tasks [18, 19, 20, 21, 22, 23, 24].

A GNN extracts local features around nodes by passing neighboring nodes' messages to the center. Traditionally, such message passing happens at all nodes **simultaneously** to extract local substructure features for all nodes [25]. Then, these feature vectors are summed to be the graph embedding. Although such symmetric message passing works for undirected graphs, it fails to capture the computation dependencies defined by DAGs – nodes within a DAG naturally have some ordering based on its dependency structures, which is ignored by existing GNNs but crucial for performing the computation on the DAG. If we encode a DAG using existing GNNs, the embedding may only capture its local **structures** but fail to encode the **computation**.

To encode the computation on a DAG, we propose an **asynchronous** message passing scheme, which no longer passes messages for all nodes simultaneously, but respects the computation dependencies among the nodes. For example, suppose node A has two parent nodes, 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.

We incorporate this feature learning scheme in both our encoder and decoder, and propose the *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 the 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.

We validate the proposed D-VAE on neural networks and Bayesian networks. We show that our model not only generates novel, realistic, and valid DAGs, but also produces smooth latent spaces effective for searching better neural architectures and Bayesian networks through Bayesian optimization.

Our contributions in this paper are: 1) We propose D-VAE, a variational autoencoder for DAGs with a novel asynchronous message passing scheme. 2) We for the first time apply graph generative models to optimizing architectures of learning algorithms. 3) We theoretically prove D-VAE encodes computations injectively, thus justifying optimizing discrete architectures in a continuous latent space.

2 Related Work

2.1 Variational Autoencoder

Variational autoencoder (VAE) [26, 27] 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 (ELBO)

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

The prior distribution $p(\mathbf{z})$ is typically taken to be $\mathcal{N}(\mathbf{0}, \mathbf{I})$. The posterior approximation $q_\phi(\mathbf{z}|\mathbf{x})$ is usually a multivariate Gaussian distributions whose mean and covariance matrix are parameterized by the encoder network and the covariance matrix is often constrained to be diagonal. The generative model $p_\theta(\mathbf{x}|\mathbf{z})$ can in principle take arbitrary parametric forms whose parameters are output by the decoder network. After learning $p_\theta(\mathbf{x}|\mathbf{z})$, we can generate new data by decoding latent vectors $\mathbf{z}$ sampled from the prior distribution $p(\mathbf{z})$. For generating discrete data types, $p_\theta(\mathbf{x}|\mathbf{z})$ is often decomposed into a series of decision steps.

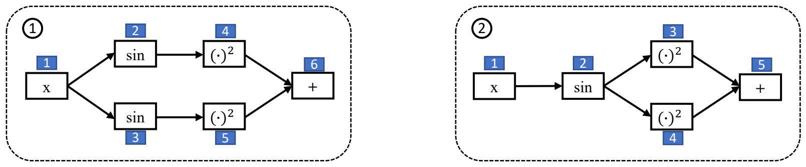


Figure 1: Examples of computations on DAGs.

2.2 Neural Architecture Search

Neural architecture search (NAS) aims at automating the design of neural network architectures by searching candidate architectures in a search space automatically. It has seen major advances in recent years [28, 29, 30, 31, 32, 33]. See Hutter et al. [34] for an overview. Methods for NAS can be categorized according to their search space, search strategy, and performance estimation strategy etc., which all greatly affect the performance of the final architecture. Based on search strategy, there are 1) reinforcement learning based 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 similarities and extrapolate the architecture space heuristically, 3) evolutionary approaches [29, 36, 37] which use evolutionary algorithms for optimizing the neural architectures, and 4) differentiable methods [32, 38] which use continuous relaxation/embedding of the search space to enable gradient-based optimization. In Appendix A we include more discussion on several most related works. In this paper, we introduce a new NAS method different from existing approaches. Based on the proposed D-VAE, we learn a compact encoding space and apply principled Bayesian optimization methods over this continuous latent space to select latent points that are most promising to decode to better neural architectures.

2.3 Bayesian Network Structure Learning

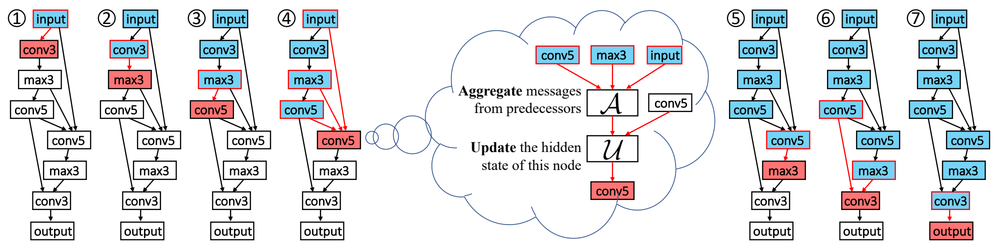
Bayesian network (BN) is an important type of graphical model with wide applications [1]. Bayesian network structure learning (BNSL) is to learn the structure of the underlying Bayesian network from observed data. BNSL dates at least back to Chow and Liu [39] where the Chow-Liu tree algorithm was developed for learning tree structured models. Much effort has been devoted to this area in recent decades (see Chapter 18 of Koller and Friedman [1] for a detailed discussion), and there is still a great deal of research on this topic [40, 41, 42]. One of the main approaches for BNSL is score-based search. That is, we define some "goodness-of-fit" score for a given network structure, and search for one with the optimal score. Commonly used scores include BIC and BDeu etc., mostly based on marginal likelihood [1]. Which score to use is by itself an ongoing research topic [43]. It is well known that finding an optimally scored BN with indegree at most d is NP-hard for $d \geq 2$ [44], so exact algorithms such as dynamic programming [45] or shortest path approaches [46, 47] can only solve small-scale problems. We have to resort to heuristic methods such as local search, simulated annealing etc. [48]. All these approaches optimize the network structure in the discrete space. In this paper, we propose to embed the discrete space into a continuous Euclidean space, and show that model scores such as BIC can be modeled smoothly in the embedded space, and hence approximate the NP-hard combinatorial search problem to a continuous optimization problem.

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 respecting the computational dependencies. In contrast to the simultaneous message passing in traditional GNNs, D-VAE allows encoding computations rather than encoding local structures. We first define *computation* below.

Definition 1. *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 following operations.*

The set of elementary operations $\mathcal{O}$ depends on specific applications. For example, when we are interested in computations given by a calculator, the elementary operation set will be all the operations defined on the functional buttons of the calculator, such as $+$, $-$, $\times$, $\div$ etc. When modeling neural


Aggregate messages from predecessors
Update the hidden state of this node
input
conv3
max3
conv5
output
Figure 2: An illustration of the encoding procedure for neural architectures. A node only updates its hidden state until all its predecessors' hidden states have been computed. Nodes in blue boxes have already updated their hidden states, and nodes in red boxes are the ones being updated.

networks, the set of elementary operations can be a predefined set of basic layers, such as 3×3 convolution, 5×5 convolution, 7×7 convolution, 2×2 max pooling etc. Note that a computation is not restricted to be a line of operations with the last operation's output being the current operation's input. It allows an operation to take arbitrary previous operations' outputs as input, with these operations forming a directed acyclic graph (DAG). The graph must be acyclic, since otherwise it will represent an infinite number of operations so that the computation never stops. Figure 1 shows two examples.

Having defined computation, we are ready to introduce how D-VAE encodes and decodes DAGs.

3.1 Encoding

We first describe how the D-VAE encoder works. The D-VAE encoder can be seen as a graph neural network (GNN) using an asynchronous message passing scheme. Given a DAG, we assume there is a single starting node which does not have any predecessors (if there are multiple such nodes, we add a virtual starting node connecting to all of them). In a neural architecture, this starting node corresponds to the input layer. We use a function $\mathcal{U}$ to update the hidden state of each node based on its neighbors' hidden states. The update function $\mathcal{U}$ takes two input variables: 1) $\mathbf{x}_v$, the one-hot encoding of the current vertex v's type, and 2) $\mathbf{h}_v^{\text{in}}$, the aggregated message from v's predecessors, given by:

$$\mathbf{h}_v^{\text{in}} = \mathcal{A}(\{\mathbf{h}_u : u \to v\}), \tag{2}$$

where $u \to v$ denotes there is a directed edge from u to v, $\mathcal{A}$ is an aggregation function on the (ordered) multiset of v's predecessors' hidden states. For the starting node without predecessors, $\mathcal{A}$ outputs an all-zero initialization vector. Having $\mathbf{x}_v$ and $\mathbf{h}_v^{\text{in}}$, the hidden state of v is updated by

$$\mathbf{h}_v = \mathcal{U}(\mathbf{x}_v, \mathbf{h}_v^{\text{in}}). \tag{3}$$

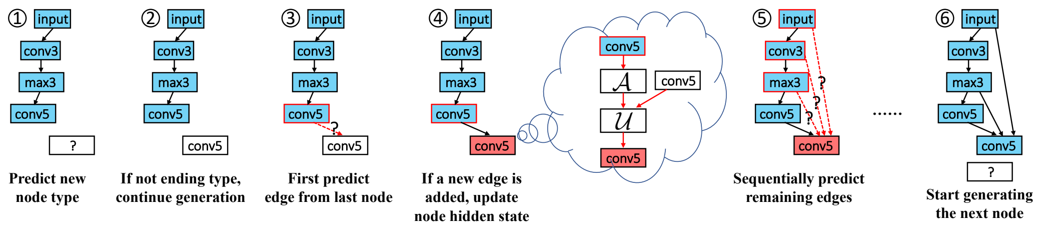
Compared to the traditional simultaneous message passing, the message passing for a node in D-VAE must wait until all of its predecessors' hidden states have already been computed. To make sure all the predecessors' hidden states are available when a new node comes, we can feed in nodes following a *topological ordering* of the DAG.

Finally, after all nodes' hidden states are computed, we use $\mathbf{h}_{v_n}$, the hidden state of v_n (the ending node without any successors) as the output of the encoder. In Figure 2, we use a real neural architecture to illustrate the encoding process. Then we feed $\mathbf{h}_{v_n}$ to two multi-layer perceptrons (MLPs) to get the mean and variance parameters of q_ϕ in (1). If there are multiple nodes without successors, we again add a virtual ending node connecting from all these "loose ends".

Note that although topological orderings are usually not unique for a DAG, we can take either one of them as the input node order while ensuring the encoding result is always the same, formalized by the following theorem. We include all theorem proofs in the appendix.

Theorem 1. *If the aggregation function $\mathcal{A}$ is invariant to the order of its inputs, then the D-VAE encoder is permutation-invariant.*

Theorem 1 means that we can get the same encoding result for isomorphic DAGs, no matter what node ordering/indexing is used. This property is much desirable since real world DAGs do not often have a canonical indexing – we do not want a DAG to be encoded into a different latent vector after permuting its nodes. The next theorem shows another property of D-VAE that is crucial for its success in modeling computation graphs, i.e., it is able to injectively encode computations on DAGs.



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

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

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 can **equivalently** 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 high-performance latent points. By performing optimization in this space, we can transform the hard discrete optimization problem into a much easier problem where principled methods are applicable. Finally, we decode the found latent points back to DAGs to finish the optimization.

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 might provide an efficient algorithm to solve the challenging graph isomorphism (GI) problem. In fact, graph neural networks are proved to be at most as powerful as the Weisfeiler-Lehman (WL) test [49], which itself still cannot solve the GI problem. Luckily, here what we really want to injectively encode are computations instead of structures, since we do not need to differentiate two different structures G_1 and G_2 as long as they represent the same computation. The two DAGs in Figure 1 show such an example. Although our D-VAE does not injectively encode structures (so does not any GNN), it is able to differentiate all distinct computations. Our D-VAE can identify that the two DAGs in Figure 1 actually represent the same computation by giving them the same encoding.

To model and learn the functions $\mathcal{A}$ and $\mathcal{U}$, we resort to neural networks thanks to the universal approximation theorem [50]. For example, we can let $\mathcal{A}$ be a gated sum:

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

where m is a mapping network and g is a gating network. Such a gated sum can be used to model injective multiset functions [49, Lemma 5].

To model the injective update function $\mathcal{U}$, we can use a gated recurrent unit (GRU):

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

were the subscript e denotes "encoding". Using a GRU also allows reducing our framework to traditional RNNs for sequences, as discussed in 3.3.

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 generates DAGs from the latent space. The D-VAE decoder uses the same asynchronous message passing scheme as in the encoder to learn intermediate node and graph states. 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 fed to GRU_d. Then, the decoder constructs a DAG node by node based on existing graph's state. 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 to v_i, and output the DAG; otherwise, continue the generation.
3. Update v_i's hidden state by $\mathbf{h}_{v_i} = \text{GRU}_d(\mathbf{x}_{v_i}, \mathbf{h}^{\text{in}}_{v_i})$, 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 $j = i-1, i-2, \dots, 1$: (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 each generated 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 node's hidden state and the current node's hidden state. Figure 3 illustrates this process. 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 $\mathbf{h}_{v_i}$ to predict the remaining edges or next node's type. Such a dynamic updating scheme is very flexible and always uses the up-to-date state of each node to generate next steps. In contrast, a non-graph-based generative model mostly only maintain the current state of its RNN [3, 10], and use it alone to generate the next step.

In step 4, when sequentially predicting incoming edges from previous nodes, we choose the reversed order $i-1, \dots, 1$ instead of $1, \dots, i-1$ 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 Discussion

Reduction to RNN. The D-VAE encoder and decoder can be reduced to ordinary RNNs when the input DAGs are reduced to linked lists. Although we propose D-VAE from a GNN's perspective, our model can also be seen as a generalization of the traditional sequence modeling framework [51] where a timestamp depends only on the timestamp immediately before it, to the DAG cases where a timestamp has multiple previous dependencies.

Bidirectional encoding. D-VAE encoder simulates how an input signal goes through a DAG, which is also known as *forward propagation* in neural networks. Inspired by the bidirectional RNN [52], we can also use another GRU to reversely encode a DAG, thus simulating the *backward propagation* process 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 functional similarities 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, which is left for future work.

4 Experiments

We validate the proposed DAG variational autoencoder (D-VAE) through two DAG modeling 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 or 5×5 convolution, 3×3 or 5×5 depthwise-separable convolution [53], 3×3 max pooling, and 3×3 average pooling. We evaluate each neural architecture's weight-sharing (WS) accuracy [33] on CIFAR-10 [54] as its performance measure. We split the dataset into 90% train and 10% held-out test sets. We use the train set to train the VAE models, and use the test set for evaluation. More details are in Appendix G.

Table 1: 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	0.832±0.001	0.527±0.001	0.421±0.004	0.914±0.001

- **Bayesian network structure learning.** Our Bayesian network dataset contains 200,000 random 8-node Bayesian networks from the `bnlearn` package [55] 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 [56]. We split the Bayesian networks into 90% train and 10% test sets. For more details, please refer to Appendix H.

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

- **Basic abilities of VAE models.** In this experiment, we perform standard tests to evaluate the basic abilities of the VAE models for modeling discrete objects, including reconstruction accuracy, prior validity, uniqueness and novelty. We move the results of this part to Appendix L.1.
- **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.
- **DAG 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 three generative baselines adapted for DAGs: S-VAE, GraphRNN and GCN. Among them, S-VAE [51] and GraphRNN [10] are adjacency-matrix-based methods, and GCN [20] uses simultaneous message passing to encode DAGs. We discuss D-VAE's advantages over these baselines in Appendix I. The training details are in Appendix J. All the code and data are available at `https://github.com/muhanzhang/DVAE`.

4.1 Predictive performance of latent representation.

In this experiment, we evaluate how well the latent embeddings learned by each model can predict the corresponding DAGs' performances, which is an important criterion to evaluate a VAE latent space's suitability for DAG optimization. This is because, if we can predict a latent point's performance with small error, searching for high-performance points in this latent space will be much easier.

Following [3], we train a sparse Gaussian Process (SGP) regression model [57] with 500 inducing points on the training data's embeddings to predict the performances of the unseen testing data's embeddings. We include the SGP training details in Appendix K.

We use two metrics to evaluate the predictive performance of the latent embeddings (given by the posterior means). 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. Table 1 shows the results. All the experiments are repeated 10 times and the means and standard deviations are reported.

From Table 1, we find that both the RMSE and Pearson's r of D-VAE are significantly better than other models. This verifies D-VAE's advantages of encoding the computations on DAGs. S-VAE follows closely by achieving the second best performance. GraphRNN and GCN have less satisfying performance in this experiment. The better predictive power of D-VAE's latent space means performing Bayesian optimization in this space is more likely to find high-performance points.

4.2 Bayesian Optimization

We perform Bayesian optimization 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 Bayesian optimization with batch size = 50 using the expected improvement (EI) heuristic [58]. Concretely speaking, we start from the training data's embeddings, and iteratively propose new points

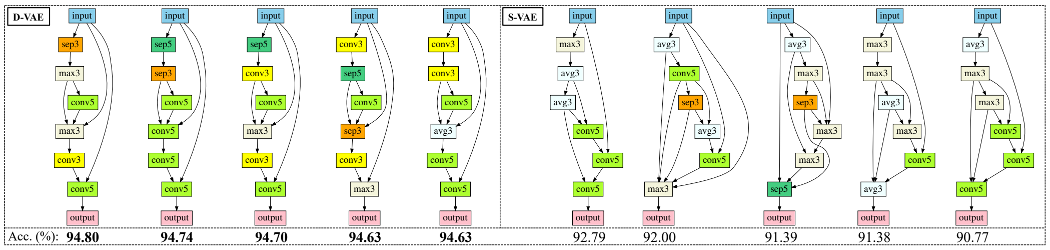


Figure 4: Top 5 neural architectures found by each model with their true test accuracies.

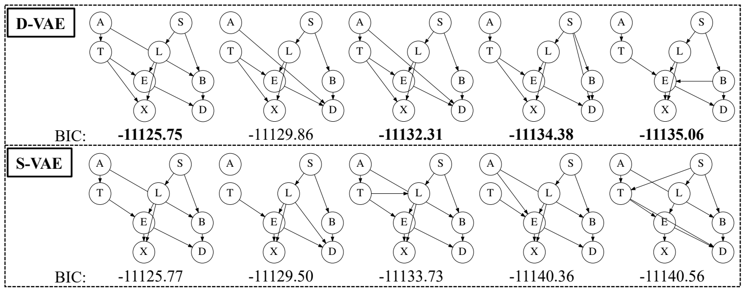


Figure 5: Top 5 Bayesian networks found by each model and their BIC scores (higher the better).

from the latent space that are most promising to be decoded to better-performance DAGs. We then evaluate these decoded DAGs' true performances and add them back to the SGP to select the next batch of points. Finally, we check the best-performing DAGs found by each model. We ensemble the results over 10 trials, and present the results for each task in the following.

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 get their true test accuracies. More details can be found in Appendix G. 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, the highest accuracy 94.80% of D-VAE is 2.01% higher than the highest accuracy of S-VAE. In addition, even the 5th best architecture of D-VAE still achieves an accuracy of 94.63%, indicating that D-VAE's latent space is more suitable for neural architecture search. Although not outperforming state-of-the-art NAS techniques such as NAONet [38] (2.11% error rate on CIFAR-10), our search space is much smaller, and we did not apply any data augmentation techniques nor did we copy multiple folds or add more filters after finding the architecture.

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 train example with a BIC of -11141.89. 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 is very close to it and outperforms all training data. Our experiments show that searching in an embedding space is a promising direction for Bayesian network structure learning.

4.3 Latent Space Visualization

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

For neural architectures, we visualize the decoded architectures from points along a great circle in the latent space. Each point is decoded 500 times and the most common decoded DAG is used. We start from the latent embedding of a flat network comprising of only 3×3 depthwise-separable convolution (orange) layers. Then, imagine this point 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 surface. Along this circle, we evenly pick 35 points and visualize their decoded nets 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

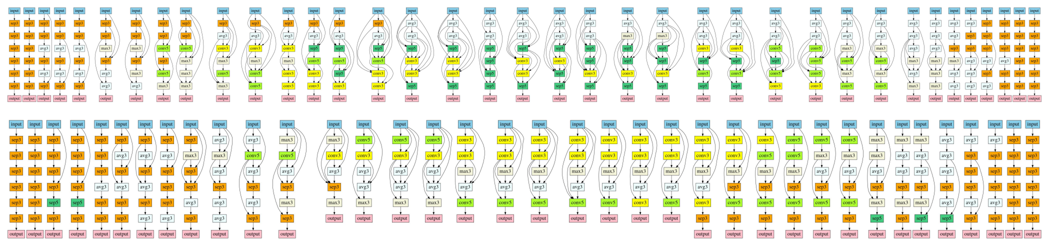


Figure 6: Great circle interpolation starting from a point and returning to itself. Upper: D-VAE. Lower: S-VAE.

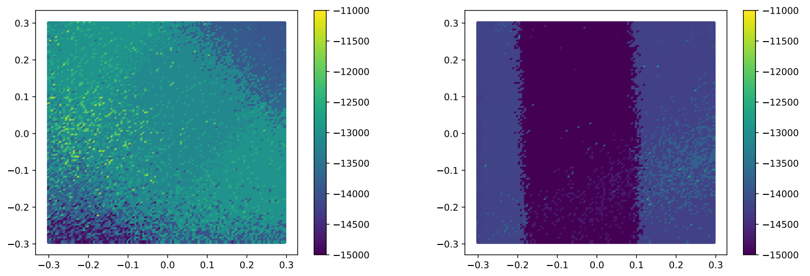


Figure 7: 2D visualization of the principal 2-D subspace of the latent space. Left: D-VAE. Right: S-VAE.

each time. Visually speaking, S-VAE's interpolations change even more smoothly by changing only zero or one node/edge between the middle interpolations. This is because S-VAE focuses on modeling the bit string representations, thus tending to embed two DAGs with few structural differences to similar latent vectors although they might have very different computational purposes. In contrast, D-VAE does not focus on structural smoothness but on smoothness of performance.

For Bayesian networks, we aim to directly visualize the BIC score distribution of the latent space. To do so, we reduce its dimensionality by choosing a 2-D subspace of the latent space 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 locally smooth regions. We suspect this helps Bayesian optimization find high-performance Bayesian networks more easily in D-VAE.

5 Conclusion

In this paper, we have proposed D-VAE, a variational autoencoder for directed acyclic graphs, and demonstrated its great potential for neural architecture search and Bayesian network structure learning, two important tasks that have never been explored by graph generative models. We believe D-VAE will be broadly useful for modeling other DAGs representing computation tasks. Graph generative models often lack real world applications. We believe our work will inspire more research on using graph generative models to optimize computation graph structures.

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] Hanjun Dai, Yingtao Tian, Bo Dai, Steven Skiena, and Le Song. Syntax-directed variational autoencoder for structured data. *arXiv preprint arXiv:1802.08786*, 2018.

[8] 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.

[9] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. *arXiv preprint arXiv:1802.03480*, 2018.

[10] 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.

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

[12] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Netgan: Generating graphs via random walks. *arXiv preprint arXiv:1803.00816*, 2018.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

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

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

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

[23] 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.

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

[25] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

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

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

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

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

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

[41] 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`.

[42] 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.

[43] 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.

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

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

[51] 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.

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

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

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

[55] 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`.

[56] 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.

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

[58] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

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

[60] 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.

[61] 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.

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

[63] 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.

Appendices

A More Related Work

Both neural architecture search (NAS) and Bayesian network structure learning (BNSL) are subfields of AutoML. See Zöller and Huber [59] for a survey. Below we introduce several works most related to our work.

Luo et al. [38] proposed a novel 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 very similar to that of Gómez-Bombarelli et al. [2], Jin et al. [14] for molecule optimization; also similar to Mueller et al. [60] for slightly revising a sentence.

There are several key differences comparing to our approach. First, they use 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, they use supervised learning instead of unsupervised learning. That means they need 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 embedding; the encoding model needs to be retrained given a new dataset. In contrast, we train our variational autoencoder in a fully unsupervised manner, so the embedding is of general purposes.

Fusi et al. [61] 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 latent representations of the pipelines. Given a new dataset, Bayesian optimization (expected improvement) 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 and use expected improvement to choose the next one to evaluate. This work is similar to ours in the application of Bayesian optimization, but the discrete search space is heuristically extrapolated.

Using Gaussian process (GP) for Bayesian network structure learning has also been studied before. Yackley and Lane [62] 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 [63] used GP to model the BDe score, and showed that the probability of improvement is better than hill climbing to guide the local search. However, these methods still heuristically and locally operate in the discrete space, whereas our embedded space makes both local and global methods such as gradient descent and Bayesian optimization applicable in a principled manner.

B Computation vs. Function

In section 3 we defined computation. Note the difference between a computation and a function. A computation $C_1 := x + 1 - 1$ defines a function $f(x) = x$. However, computations $C_2 := x - 1 + 1$ and $C_3 := x$ also define the same function $f(x) = x$, but 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 which cares about the results. Two same computations can also define different functions, e.g., two identical neural architectures will represent different functions given different training conditions, since the weights of their layers will be different). In D-VAE, we model computations instead of functions, since 1) modeling functions require knowing the semantic meaning of each operation which is left to future work, and 2) a function can additionally require specifying the inner parameters of each operation 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. For the starting node v_1 with no predecessors, the aggregation function $\mathcal{A}$ always outputs a zero vector. Thus, $\mathbf{h}^{\text{in}}_{v_1}$ is invariant to node ordering. Subsequently, the hidden state $\mathbf{h}_{v_1}$ of v_1 output by $\mathcal{U}$ is permutation-invariant.

Now we prove the theorem by induction. Consider node v. Suppose for every predecessor u of v, the hidden state $\mathbf{h}_u$ is permutation-invariant. We will show that $\mathbf{h}_v$ is also permutation-invariant. In (2), the output $\mathbf{h}^{\text{in}}_v$ by $\mathcal{A}$ is permutation-invariant since $\mathcal{A}$ is invariant to the order of its inputs $\mathbf{h}_u$, and $\mathbf{h}_u$ are all permutation-invariant. Subsequently, the output $\mathbf{h}_v$ of (3) is permutation-invariant. By induction, we know that every node's hidden state is invariant to node ordering, including the ending node's hidden state, i.e., the output of D-VAE's encoder. Thus, the D-VAE encoder is permutation-invariant. □

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 $\mathbf{h}^{\text{in}}_{v_1} = \mathbf{0}$. Since (3) is injective, we know the mapping from C_1 to $\mathbf{h}_{v_1}$ is injective. 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 $\{C_j : v_j \to v_i\}$.

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. □

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 w.r.t. its input. Remember $\mathcal{A}$ takes $\{\mathbf{h}_u : u \rightarrow v\})$ as input. Here, $\{\mathbf{h}_u : u \rightarrow v\})$ can be either a multiset (where order of its elements do not matter) or an ordered multiset (where order of its elements matter), depending on how the previous layers' outputs are aggregated as the input to a next layer. For example, if these outputs are summed or averaged as the input to next layer, then $\mathcal{A}$ can take a multiset as input as the order does not matter. However, if these outputs are concatenated as next layer's input, then 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 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 model injective multiset functions, but cannot handle the ordered case, we need to slightly modify (4) in order to make it model an injective ordered multiset function. Our scheme is as follows:

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

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, thus is more suitable for the neural architectures in our experiments. We empirically observed that this aggregation function can further 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:

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

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 next layers. What we are interested in is the computation result output by the final layer. However, for a Bayesian network, the graph represents a set of conditional independences among variables instead of a computational flow. In particular, for 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 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).

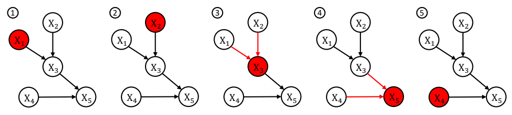


Figure 8: An example Bayesian network and its encoding.

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 $\mathbf{h}_G := \sum_{j=1,\ldots,i-1} \mathbf{h}_{v_j}$.

Note that the combination of (12) and (5) can injectively model the conditional dependence of v and its parents u. In addition, using the sum aggregator can also model injective set functions

[49, Lemma 5]. Therefore, the above encoding scheme is able to **injectively encode** the complete **conditional dependence structure** of a Bayesian network, thus also the overall score function s defined on the structure.

G More Details about Neural Architecture Search

We use the efficient neural architecture search (ENAS)'s software to generate training neural architectures [33]. These ENAS architectures are regarded as the real data which we want our generative models to model. 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 a controller which is used to propose new architectures as well as the shared weights of the proposed architectures. It uses a weight-sharing (WS) scheme to obtain a quick but rough estimate of how good an architecture is. It assumes that an architecture with a high validation accuracy using the shared weights, or weight-sharing accuracy, is more likely to have a high test accuracy after fully retraining its weights from scratch.

We first run ENAS in macro space (section 2.3 of Pham et al. [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% and 10% train and held-out test set. Then our task becomes to train a VAE on the training neural architectures, and then generate new architectures with high weight-sharing accuracies based on Bayesian optimization. Note that our target variable 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 target variable in the Bayesian optimization experiments.

One might wonder what is the meaning of training another generative model after we already have one. There are two reasons. One is that ENAS is not general-purpose, but task specific. It leverages the validation accuracy signals to train the controller based on reinforcement learning. In contrast, D-VAE is completely unsupervised. After the training is done, D-VAE can be applied to other neural architecture search tasks without retraining. The second reason is that, training a VAE provides a way to embed neural architectures into vectors, which can be used for downstream tasks such as visualization, classification, searching, measuring similarity etc.

Note that in this paper, we only use neural architectures from the ENAS macro space, i.e., each architecture is an end-to-end convolutional neural network instead of a convolutional cell. When we fully train a found architecture, we follow the original setting of ENAS to train on CIFAR-10's train data for 310 epochs and report the test accuracy. We leave the generation of convolutional/RNN cells (ENAS micro space) and the generation of convolutional neural networks with more layers (e.g., 12, 24) to future work.

Due to our constrained computational resources, we choose not to perform Bayesian optimization on the true validation accuracy, which would be a more principled way for searching neural architectures. 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 the same as in section 4.2, with the target value replaced by the true validation accuracy. Note that the evaluation process of each new point found by BO will take much longer time now, which might result in months of GPU time. Thus, making the evaluations parallel is very necessary.

In the experiments, 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 which has an error rate of 2.11%, our architecture only contains 3 million parameters compared to the state-of-the-art NAONet + Cutout which has 128 million parameters [38]. We emphasize that in this paper, our main focus is introducing the D-VAE model for DAG optimization but not achieving state-of-the-art NAS results, since beating state-of-the-art NAS methods typically requires a lot of computation resources which we do not have.

H More details about Bayesian network structure learning

We consider a small synthetic problem called Asia [56] 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[1]. Bayesian Information Criteria (BIC) score is used to evaluate how well a Bayesian network fits the 5,000 samples. 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, which is a reasonable assumption for many practical applications, e.g., when the variables have temporal order [1]. The generated train and test Bayesian networks have topological orders consistent with the true network of Asia. The probability of a node having an edge with a previous node (as specified by the order) is set by the default option $2/(k-1)$, where $k=8$ is the number of nodes, which results in sparse graphs where number of edges is in the same order of the number of nodes.

For the evaluation metric, although we choose the default BIC metric of the package, one could also use BDe score; the two scores have 99.96% corrleation here. Also note that this Asia problem can be solved by hill climbing or exact methods. We study it only for demonstration purposes. We leave the more general setting (when the order is unknown) and learning larger networks as our future work.

I Baselines

As discussed in the introduction, 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 [51] on the topologically sorted node sequences, with each node's string treated as its input vector.

GraphRNN. One similar generative model is GraphRNN [10]. Different from S-VAE, it further decomposes an adjacency column into entries and generates the entries one by one using another edge-level GRU. However, 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.

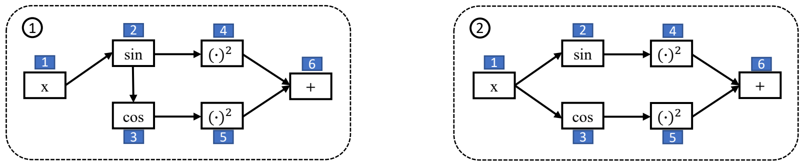


Figure 9: Two bits of change in the string representations can completely change the computational purpose.

Both GraphRNN and S-VAE treat DAGs as bit strings and use RNNs to model them. Here, we use a simple example to show that 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 paths. The result is that the right DAG will always output 1. In S-VAE and GraphRNN, since the bit representations of the left and right DAGs are very similar, it is very likely that they are encoded to similar latent vectors, which makes the latent space less smooth thus making the optimization more difficult. In contrast, D-VAE can better differentiate such subtle differences, as the changing of the signal flow paths also changes how D-VAE encodes the DAG.

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

GCN. The graph convolutional network (GCN) [20] 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 only encodes graphs, we equip GCN with D-VAE's decoder to make it a VAE model.

The main disadvantage of using GCN to encode DAGs is that the simultaneous message passing ignores the computation order of DAG nodes. It only focuses on learning local substructure patterns but fails to encode the computations.

J VAE Training Details

We train all the four models using similar settings to be as fair 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 number of hidden dimensions 501. The MLPs used to output the mean and variance parameters of $q_\phi(\mathbf{z}|\mathbf{X})$ are all implemented as single linear layer networks. We set the dimension of latent space to be 56 for all models. All VAE models use $\mathcal{N}(\mathbf{0}, \mathbf{I})$ as the prior distribution $p(\mathbf{z})$.

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. To measure the reconstruction loss, we use teacher forcing [14]: 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 gradient descent following [14].

When optimizing the VAE loss, we use ReconstructLoss + αKLDivergence as the loss function. In 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, 7, 14]. Following the implementation of Jin et al. [14], we set $\alpha = 0.005$.

For D-VAE on neural architectures, we use the bidirectional encoding discussed in section 3.3. For Bayesian networks and other models, we find bidirectional encoding to be less useful, sometimes even hurt the performance, thus we only use unidirectional encoding. All DAGs are fed to the models according to their nodes' topological orderings. Although the original GraphRNN feeds nodes using a BFS order (for undirected graphs), we found that it is worse than using a topological order here. Since there is always a path connecting all nodes in the neural architectures generated by ENAS, their topological orderings are unique. For Bayesian networks, we feed nodes by a fixed order "ASTLBEXD" (which is always a topological order).

Mini-batch SGD with Adam optimizer is used for both models. For neural architectures, we use a batch size of 32 and train for 300 epochs. For Bayesian networks, we use a batch size of 128 and train for 100 epochs. An initial learning rate of 1E-4 is constantly used, and we multiply the learning rate by 0.1 if the training loss does not decrease for 10 epochs. We use PyTorch to implement all the models.

K SGP Training Details

We use sparse Gaussian process (SGP) as the predictive model in BO. We use the open sourced SGP implementation in [3]. Both the train and test data are standardized according to the mean and std of the train data before feeding to the SGP. And the predictive performance are also calculated on the standardized data. 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 is not realistic for a typical scenario where network/dataset are large and evaluating a score is expensive; and 2) we found using a smaller sample of training data even 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 it is purely unsupervised.

L More Experimental Results

L.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 portion of unique DAGs out of the valid generations (Uniqueness), and 4) the portion of valid generations that are never seen in the training set (Novelty).

We first evaluate each model's reconstruction accuracy on the test sets. Since both encoding and decoding are stochastic in VAEs, for each test DAG, we encode it 10 times and decode each encoding 10 times too following previous work [3, 14]. Then we report the average portion of the 100 decoded DAGs that are identical to the input.

To calculate prior validity, we sample 1000 latent vectors from the prior distribution and decode each latent vector 10 times. Then we report the portion of these 10,000 generated DAGs that are valid. A generated DAG is valid if it can be read by the original software which generated the training data. More details are in Appendix L.2.

Table 2: 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	5.42	99.37	41.18	100.00	99.07	99.89	30.53	98.26

We show the results in Table 2. Among all the models, D-VAE and S-VAE generally have the highest performance. 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 have higher uniqueness, meaning that they generate more diverse examples. We find that GCN is not suitable for modeling neural architectures as it only reconstructs 5.42% unseen inputs. However, this is not surprising, since the simultaneous message passing scheme in GCN fails to encode the computational dependencies among different layers of a neural network. The use of the summed node hidden states as the graph encoding might also lose important chronological information within the hidden states.

L.2 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 type (the input node); 2) It has one and only one ending type (the output node); 3) Other than the input node, there are no nodes which do not have any predecessors (an isolated path); 4) Other than the output node, there are no nodes which do not have any successors (a blocked path); 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 (no directed cycles).

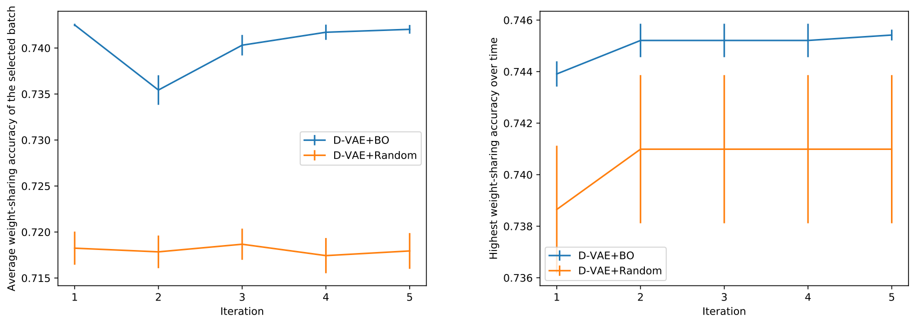


Figure 10: Comparing BO with random search on neural architectures. Left: average weight-sharing accuracy of the selected points in each iteration. Right: highest weight-sharing accuracy of the selected points over time.

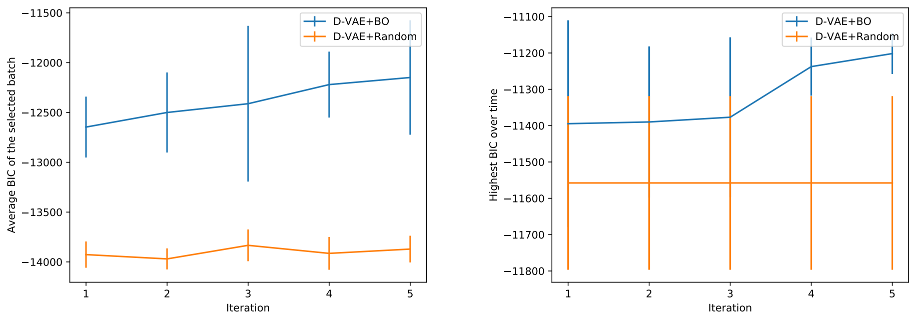


Figure 11: Comparing BO with random search on Bayesian networks. Left: average BIC score of the selected points in each iteration. Right: highest BIC score of the selected points over time.

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.

L.3 Bayesian optimization vs. random search

To validate that Bayesian optimization (BO) in the latent space does provide guidance in searching better DAGs, we compare with a random search baseline which randomly samples points from the latent space of D-VAE. Figure 10 and 11 show the results. As we can see, BO consistently finds better DAGs than random search.