A Neural Framework for Learning DAG to DAG Translation

M. Clara De Paolis Kaluza
Northeastern University
Boston, MA 02115
clara@ccs.neu.edu

Saeed Amizadeh
Microsoft Inc.
Redmond, WA 98052
saamizad@microsoft.com

Rose Yu
Northeastern University
Boston, MA 02115
roseyu@northeastern.edu

Abstract

Learning representations of graph data is essential to many real-world machine learning problems, and therefore, there has been a recent push in representation learning community to adapt deep learning techniques for graph-structured data, either as input or output of the model. However, almost all of these techniques only focus on either the input or the output graph space and not both. In this paper, we propose a neural model for learning deep functions on the space of directed acyclic graphs (DAGs) where *both* the domain and the range of the target function are DAG spaces, and develop a DAG-to-DAG translation model. We consider a supervised learning mechanism to efficiently train our DAG-to-DAG simplification model. Our experiments show the ability of the model to learn the structural signals and patterns in both input and output spaces, showing improved syntax measures in generated outputs by over 20% and edge classification by 44%.

1 Introduction

Graphs play a crucial role in modeling complex interactions and relations among the components of a given problem and as a result, in many real-world machine learning applications, the data naturally appear in graph forms, in both the input and the output spaces. Therefore, both understanding (*i.e. embedding*) and generating (*i.e. synthesis*) graph-structured data are essential to many problems; examples of such problems include computation graph optimization [18], graph summarization [16], SQL query optimization [25], program synthesis [4, 1, 3], and logical expression simplification [20, 26]. For understanding graph-structured data, traditional techniques incorporate (unsupervised) graph embeddings or fixed feature extractors to map graphs into real vector spaces [6]. More recent frameworks learn such embeddings directly using a variety of graph neural networks[5, 14, 22, 9, 21, 12, 19]. For synthesis, classical methods typically use rule-based, grammar-driven techniques [10, 11], while the more recent techniques try to learn the synthesis process directly via machine learning [15, 23, 27, 4, 1, 3]. Mathematically, these neural methodologies for graph-structured data can be seen as function approximation frameworks where either the domain or the range of the target function is a graph space and therefore embedding and synthesis are treated separately.

In this paper, we bring the embedding and synthesis methodologies under one unified framework such that one can learn functions from one graph space onto another graph space without the strong assumption of independence between the embedding and generative process. In particular, we focus on learning functions on the space of directed acyclic graphs (DAG), and refer to the learned function as *DAG-function*. The applications of learning such DAG-functions range from program/query optimization to circuit simplification where both the inputs and the outputs are represented as DAGs. DAGs can also be seen as the generalization of sequences where each element in the sequence (node in the DAG) can have more than one direct predecessor. As a result, our proposed neural DAG-to-DAG learning framework generalizes the sequence-to-sequence (Seq2Seq) learning framework used for machine translation [24] by proving support for general DAG structures rather than the single-predecessor chain structure of Seq2Seq. Furthermore, we propose several differentiable structural loss functions to efficiently train DAG-functions in the supervised setting. In summary, the main contributions of this paper are: (a) We study the problem of DAG-to-DAG simplification as a

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

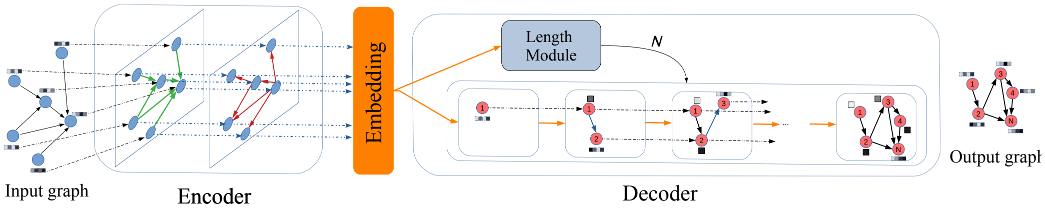

Figure 1: The encoder embeds of the input graph. The decoder generates an output graph conditioned on this embedding by first determining the number of nodes N, then generating N nodes sequentially. At each step n, edges from existing nodes to the new node v_i are determined as well as the node feature x_{v_i}.

translation problem, which enjoys wide application in computation optimization problems and graph summarization. (b) We develop a general encoder-decoder framework for learning functions from a DAG space onto another DAG space - aka *DAG-to-DAG translation*. (c) We empirically investigate the effectiveness of several differentiable structural loss functions in order to set up a unified supervised learning framework for training DAG-function and adapt our supervised DAG-to-DAG translation framework to learn a DAG-function for the Boolean circuit simplification problem. We present experiments on the problem of logical circuit simplification and show superior performance compared to baselines in structural and syntactical measures.

2 Related Work

Several deep graph-structured generative models have been developed for graph synthesis that mostly focus on producing random samples from a learned distribution *e.g.* DeepGMG [15], GraphVAE [23], GraphRNNs [27]. These models have found applications in synthesizing structures that obey some grammar or distribution, such as unobserved but plausible molecular structures (*e.g.* [23], [13], and others). In contrast to these models, our goal is to generate outputs conditioned on single inputs, much like a translation model. There has been several deep models proposed for embedding graph-structured data, *e.g.* [22, 5, 14, 22, 9, 21, 12, 19, 2]. In this paper, we employ the model from [2] since it specially considers DAG structures. Similar to our proposed work, the domain-specific tree-to-tree model for the program translation introduced in [7] generates an output structure conditioned on a single input with the goal of matching the semantic meaning of the input graph. However, our model can work on more general graph structures as long as a node ordering is given.

3 DAG-to-DAG Recursive Neural Network

In this section, we develop a new framework for DAG-to-DAG learning that encodes the structure of inputs and, conditioned on these inputs, generates DAG outputs and we specify a supervised approach for learning on this framework.

Formulation Let $G_{in} = (V_{in}, E_{in})$ and $G_{out} = (V_{out}, E_{out})$ be two DAGs such that $G_{out} = \mathcal{F}(G_{in})$ for some DAG-function $\mathcal{F}$. The nodes $v \in V_{in}, V_{out}$ are described by feature vectors $\boldsymbol{x}_v$ that encode node properties. For DAGs, a natural node order is given by the topological sort of the graph. For each node v in a DAG G, predecessor nodes $u \in \pi(v)$ are the set of nodes such that there is an edge $e_{u \to v} \in E_G$. The goal is to learn function $\mathcal{F}$ given a training set of paired input and output graphs. For example, in the case of logical circuit simplification, G_{in} is a logical circuit, the DAG-function $\mathcal{F}$ represents a simplification algorithm, and G_{out} is a semantically equivalent simplified form of the expression represented by G_{in}. Features $\boldsymbol{x}_v$ encode the type of gate or variable corresponding to each node. An example of logical expression simplification is depicted in Figure 2

The D2DRNN Model We propose an encoder-decoder design using recursive neural networks to model the function $\mathcal{F}(\cdot)$; we refer to this model as `D2DRNN`. In particular, the model is composed of an encoder $\mathcal{E}_\alpha$ with model parameters α which computes a fixed-size embedding of the input graph G_{in}, and a decoder $\mathcal{D}_\beta$ with parameters β which takes as input the embedding and produces an output graph $\hat{G}_{out}$. Thus, we define the DAG-function as $\mathcal{F}_\theta(G_{in}) := \mathcal{D}_\beta(\mathcal{E}_\alpha(G_{in}))$ where $\theta := \langle \alpha, \beta \rangle$.

DAG Encoder We use as an encoder function the recently developed Deep-Gated DAG Recursive Neural Network (`DG-DAGRNN`)[2] This model generalizes stacked RNNs on sequences to DAG structures. Each layer of the `DG-DAGRNN` is comprised of gated recurrent units (GRUs) [8], repeated for each node $v \in G_{in}$. The GRU corresponding to node v has as input an aggregated representation

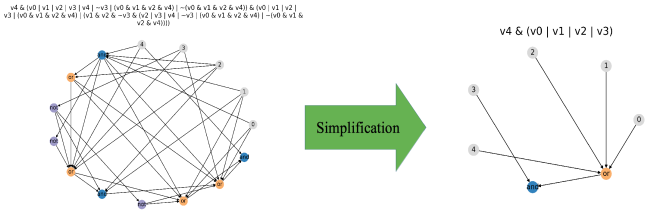
v4 & (v0 | v1 | v2 | v3 | v4 | ~v3 | (v0 & v1 & v2 & v4) | ~(v0 & v1 & v2 & v4)) & (v0 | v1 | v2 | v3 | (v0 & v1 & v2 & v4) | (v1 & v2 & ~v3 & (v2 | v3 | v4 | ~v3 | (v0 & v1 & v2 & v4) | ~(v0 & v1 & v2 & v4))))
and
or
not
Simplification
v4 & (v0 | v1 | v2 | v3)
4
3
2
1
0

Figure 2: Logical expression simplification as a graph optimization problem

of the hidden states of the units corresponding to the predecessors $\pi(v)$. We use a sum operator as the aggregation function. For an aggregation function $\mathcal{A}$, the hidden state $\boldsymbol{h}_v$ is given by

$$\boldsymbol{h}_v = GRU(\boldsymbol{x}_v, \boldsymbol{h}'_v), \quad \text{where } \boldsymbol{h}'_v = \mathcal{A}\left(\{\boldsymbol{h}_u | u \in \pi(v)\}\right). \tag{1}$$

Since the topological sort of G_{in} defines an ordering of nodes, all the hidden states $\boldsymbol{h}_v$ can be computed with a single forward pass along a layer of the DG-DAGRNN, computing equation 1 for each node in topological order. This differs from many existing graph embedding models that require loopy message passing to compute node representations [5, 14]. The encoder contains multiple layers, each layer passing hidden states to the recurrent units in subsequent layer corresponding to the same node. Inspired by the findings in [24] that sequence learning benefits from processing some sequences in backwards order, some layers in the encoder process the DAG nodes in the reverse order.

DAG Decoder The encoder provides an embedding $\boldsymbol{H}_{in} = \mathcal{E}_\alpha(G_{in})$, which serves as the input for the DAG decoder. We generate an output graph iteratively. First, the size of the output graph (*i.e.* the number of nodes) is calculated by the Length module f_{Length}. For this function, we use a multi-layer perceptron (MLP) with Poisson regressor output layer, which takes the input graph embedding $\boldsymbol{H}_{in}$ and outputs the mean N of a Poisson distribution describing the output length.

Once the length N is determined, for $n = 1, \ldots, N$, a node v_n is added to the output graph. Whether we should add an edge e_{u,v_n} for all nodes $u \in \{v_1, \ldots, v_{n-1}\}$ already in the graph is determined by the NodeScore module, which is a MLP. Since the output nodes are generated in the topological order as well, edges are always directed from nodes added earlier to nodes added later in the process.

For each generated node v, we compute the hidden state $\boldsymbol{h}_v$ using a similar mechanism as the encoder: we aggregate the hidden states of its predecessors and feed it to a GRU. The other input of the GRU cell is set to the aggregated states of all the *sink* nodes generated so far. For the first node, the hidden state is initialized based on the encoder's output. Next, the output node features are generated based on its hidden state using the Feature module $f_{Feature}$, which is another MLP. Finally, once the last node is generated, edges are introduced with probability 1 for sinks in the graph to ensure a connected graph with only one sink node as an output. This process is detailed in Algorithm 0 in Appendix A.

Loss Function We propose a supervised learning framework for DAG-to-DAG translation. Given a training dataset $D = \{G_{in}^{(i)}, G_{target}^{(i)}\}$ and loss $\mathcal{L}$, we optimize the model parameters $\boldsymbol{\theta} = \langle \boldsymbol{\alpha}, \boldsymbol{\beta} \rangle$ to minimize the loss using end-to-end backpropagation. Our loss is defined as:

$$\mathcal{L} = \mathcal{L}_{length} + \mathcal{L}_{nodes} + \mathcal{L}_{structure}, \tag{2}$$

where $\mathcal{L}_{length} = \text{Poisson-NLL-Loss}(f_{Length}(\boldsymbol{H}_{in}), |V_{target}|) \approx |V_{out}| - |V_{target}| \log(|V_{out}|)$ measures the difference between the lengths of the output and the target graphs while $\mathcal{L}_{nodes} = \text{Cross-Entropy}(V_{out}, V_{target})$ quantifies the difference between the node feature vectors of the output and target. The structure loss $\mathcal{L}_{structure}$ measures the difference between the output and the target in terms of graph structure. For the structure loss, we have empirically tested four different options: (1) the binary cross-entropy (BCE) loss, where the presence/absence of an edge is treated as binary classification, (2) the Frobenius norm of the difference between the output and the target adjacency matrices, (3) the ℓ_1-norm of the same quantity, and (4) the diffusion loss, where the ℓ_2-distance between the diffusion results of a random unit vector over the output and the target graphs is calculated. An empirical comparison of these loss functions on the validation set is demonstrated in Appendix B.

4 Experiments

In order to validate our method, we test on the DAG-to-DAG problem of circuit simplification [20, 26], a well-suited problem for our model since it involves encoding and synthesizing DAGs.

Circuit Simplification In circuit simplification, a logical expression is reduced to a logically-equivalent expression with fewer operations, as depicted in Figure 2. We construct a dataset of 12,500 random syntactically-valid circuits with 15-30 nodes, each of one of four types: variables, `NOT`-gates, `OR`-gates, and `AND`-gates. Ground truth simplified graphs were generated by simplifying generated expressions using the Sympy Python library [17] and converting these expression to their graph representation. See Appendix C for details of the dataset. Note that in this setup, we do *not* explicitly enforce any syntactical or semantical (*i.e.* logical equivalence) constraints during training. Even though incorporating such constraints are essential for learning a practical circuit simplifier, in this paper, we only study the extent the pure representational capacity of our model can learn these constraints *implicitly*. Nevertheless, enforcing such constraints explicitly is a topic of our future work.

Baselines We consider translation models that condition the output graph on a single input rather than sample from an unconditional distribution of graphs. The `Seq2Seq` model [24] considers topologically-sorted nodes in a graph as a sequence and generates an output sequence of nodes. This model does not generate edges or consider the structure of the input graph, only the order of nodes. We also consider extensions of this baseline, sequence-to-DAG (`Seq2DAG`) and DAG-to-sequence (`DAG2Seq`) models. In the former, the encoder consumes the topologically ordered nodes and generates a DAG-structured output using the same decoder as the full `D2DRNN` model. Conversely, the latter uses the same DG-DAGRNN encoder as the full `D2DRNN` to produce an embedding of the input graph, but generates only a sequence of nodes rather than the full structure of a DAG.

Evaluation Metrics In our experiments, error measures are calculated on the held-out test set of 2,500 circuits. The *sequence edit distance* measures the Levenshtein edit distance between the output and target sequences. This measure gives an error on generated node types as well as the order in which they were generated. Next, we calculate the average difference in the number of nodes and the number of edges between the generated and target circuits. We also consider several syntax-related error measures. The *node syntax error* measures the percentage of generated nodes that violate the fan-in constraint for each node type. For circuit simplification, the number of output nodes is never greater than that of the input circuit, so we measure the percentage of outputs that violates this constraint. Finally, we measure the total percentage of graphs do not violate any of the above syntactical constraints. The results are listed in Table 1. In the case of the `Seq2Seq` and `DAG2Seq` models, since no graph structure is generated, we cannot evaluate the graph-based error measures. As shown in the table, the D2DRNN model where both the encoder and the decoder operate on DAGs outperforms the baselines especially regarding the structural metrics. On the other hand the Seq2Seq model which treats DAGs as mere sequences does not outperform any of the DAG-aware methods, which in turn shows the necessity of taking the DAG structure into consideration.

	D2DRNN	Seq2DAG	DAG2Seq	Seq2Seq
Length loss	0.062	**0.060**	–	–
Node loss	1.717	**1.647**	0.908	0.924
Seq. edit distance	0.499	0.492	**0.452**	0.447
Num. nodes	2.567	2.206	**2.0**	3.0
Num. edges	10.60	**2.557**	–	–
Edge classification	**0.139**	0.365	–	–
Node syntax	**0.221**	0.666	–	–
Num. vars	**0**	**0**	0.121	0.0158
Graph syntax	**0.793**	0.994	–	–

Table 1: Errors for full model (DAG2DAG) and baselines. In all cases, lower is better Though all models can learn some aspects of the sequence of nodes, the full model outperforms the baselines in syntax measures

5 Conclusion

In this paper, we proposed a novel neural graph synthesis framework that learns to summarize a DAG while preserving the syntactic structure. Both our proposed encoder and decoder are structure-aware, which in turn enables them to outperform the baselines that treat DAGs as mere sequences, as our experiments showed. Furthermore, for the task of logical circuit simplification, compared to baselines, our model were able to implicitly pick up syntactical constraints without actually employing any explicit syntactical or semantical loss terms during training. This further shows the excellent representational power of the proposed model. As future work, we intend to incorporate Reinforcement Learning to directly encode syntactical and semantical constraints into the learning process of DAG-to-DAG models.

References

[1] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Anonymous. Learning to solve circuit-sat: An unsupervised differentiable approach. In *Submitted to International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=BJxgz2R9t7`. under review.

[3] Anonymous. Learning a meta-solver for syntax-guided program synthesis. In *Submitted to International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=Syl8Sn0cK7`. under review.

[4] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

[5] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[6] H. Cai, V. W. Zheng, and K. Chang. A comprehensive survey of graph embedding: problems, techniques and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[7] X. Chen, C. Liu, and D. Song. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018. URL `http://arxiv.org/abs/1802.03691`.

[8] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[9] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

[10] S. Gulwani, O. Polozov, R. Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[11] E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming*, pages 50–73. Springer, 2009.

[12] R. Kondor, H. T. Son, H. Pan, B. Anderson, and S. Trivedi. Covariant compositional networks for learning graphs. *arXiv preprint arXiv:1801.02144*, 2018.

[13] Q. Li, Z. Han, and X.-M. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. *arXiv preprint arXiv:1801.07606*, 2018.

[14] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[15] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

[16] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: A survey. *ACM Computing Surveys (CSUR)*, 51(3):62, 2018.

[17] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, Jan. 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL `https://doi.org/10.7717/peerj-cs.103`.

[18] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

[19] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proc. CVPR*, volume 1, page 3, 2017.

[20] S. Sapra, M. Theobald, and E. Clarke. Sat-based algorithms for logic minimization. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 510–517. IEEE, 2003.

[21] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[22] B. Shuai, Z. Zuo, B. Wang, and G. Wang. Dag-recurrent neural networks for scene labeling. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3620–3629, 2016.

[23] M. Simonovsky and N. Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. *arXiv preprint arXiv:1802.03480*, 2018.

[24] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[25] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, page to appear, 2019.

[26] C. Yang and M. Ciesielski. Bds: A bdd-based logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(7):866–876, 2002.

[27] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec. Graphrnn: A deep generative model for graphs. *arXiv preprint arXiv:1802.08773*, 2018.

A The Decoding Algorithm

Algorithm 1 The Decoder Module: $G_{out} = \mathcal{D}(\boldsymbol{H}_{in}, \boldsymbol{\delta})$
$\boldsymbol{H}_{in}$: the embedding of G_{in}
$\boldsymbol{\delta}$: the edge addition threshold

```
$N \leftarrow f_{Length}(\boldsymbol{H}_{in})$    ▷ Compute the size of the output graph
$\boldsymbol{h}_{v_0} \leftarrow f_{init}(\boldsymbol{H}_{in}), v_0 \leftarrow f_{Feature}(\boldsymbol{h}_{v_0})$    ▷ Compute $\boldsymbol{h}_v$ and $v$ for the first node
$\boldsymbol{H}_{Current} \leftarrow \boldsymbol{H}_{in}$    ▷ Initialize the current graph embedding
$V \leftarrow \{v_0\}, E \leftarrow \{\}$
for $i = 1, \ldots, N-1$ do
    for $u \in V$ where $u \neq v_i$ do
        $p_{edge_{u,v_i}} \leftarrow f_{\text{NodeScore}}(\boldsymbol{h}_u, \boldsymbol{H}_{Current})$
        if $p_{edge_{u,v_i}} \geq \boldsymbol{\delta}$ then
            $E \leftarrow E \cup \{e_{u \to v_i}\}$
        end if
    end for
    $\boldsymbol{h}_{v_i} \leftarrow GRU\big(\mathcal{A}(\boldsymbol{h}_u : u \in V_{sink}), \mathcal{A}(\boldsymbol{h}_u : u \in \pi(v_i))\big)$
    $V \leftarrow V \cup \{f_{Feature}(\boldsymbol{h}_{v_i})\}$    ▷ Compute the new node's feature vector
    $\boldsymbol{H}_{Current} \leftarrow g_{agg}(\boldsymbol{h}_v)$    ▷ Update the graph embedding
end for
return $G_{out} := (V, E)$
```

B Comparing Different Structural Losses

Table 2 demonstrates different error metrics on the validation data when the DAG2DAG model was trained using different structure loss functions. As the results show, the Frobenius norm loss and the diffusion loss exhibit superior performance on the validation data compared to the other two losses. As a result, in our experiments, we have chosen the diffusion loss for the structure loss function.

	BCE	$\|\cdot\|_1$	$\|\cdot\|_F$	Diffusion
Length loss	0.049	0.052	0.050	**0.045**
Node loss	2.063	2.453	1.959	**1.544**
Seq. edit dist	0.512	0.516	**0.506**	0.553
Num. nodes error	1.86	2.17	**1.75**	2.58
Num. edges error	**2.06**	2.29	2.12	2.69
Edge classification	0.604	0.591	0.674	**0.362**
Node syntax	0.215	0.229	**0.198**	0.205
Num. vars	0.001	0.102	0.008	**0.00**
Graph syntax	0.732	0.794	0.651	**0.625**

Table 2: Error metrics on the validation data for different choices of structure loss formulations during training.

C Circuit Simplification Dataset

We construct a dataset of random circuits with 15-30 nodes (3-8 variables). Each node in the graph corresponds to one of four node types: variables, `NOT`-gates, `OR`-gates, and `AND`-gates. Node features are given by a one-hot encoding of the four node types. All generated circuits were valid, *i.e.* they have a single sink node, `NOT`-gate nodes have a single input, variable nodes have no inputs, and `OR`- and `AND`-gate nodes have at least two inputs.

To produce ground truth simplifications, the generated logical expressions corresponding to each graphs were simplified using the Sympy Python library [17], which provides an equivalent expression either in conjunctive normal form (CNF) or disjunctive normal form (DNF), depending on which is more compact. Any circuits that simplified to `True` (tautology) or `False` (unsatisfiable) were excluded. We use 10,000 circuit pairs for training and 2,500 pairs for testing. Node features are given by a one-hot encoding of the four node types.