Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Using Graph Neural Networks for approximating DAG posteriors

Scientific Students' Association Report

Author:

Zsombor Bánfi

Advisor:

András Formanek

2014

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Bánfi Zsombor*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2022. november 1.

Bánfi Zsombor
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

Deep generative models have achieved remarkable success in various data domains, including images, time series, and natural languages. However, there remain substantial challenges for combinatorial structures, including graphs, despite them being abundant in the world.

Among different graph types, directed acyclic graphs (DAGs) are of particular interest to machine learning researchers, as in many problems data is represented in that form, including Neural Architecture Search, Social Network Profiling, or Bayesian Network Structure learning. Our work concerns the latter problem. We aim to create a model which approximates well the posterior of Bayesian Networks and can also be sampled effectively so that it is suitable for generative purposes.

Markov Chain Monte Carlo (MCMC) algorithms, the ubiquitous tool for sampling from a high-dimensional, multimodal probability distributions, typically rely on random local updates to propagate configurations of a given system in a way that ensures that generated configurations will be distributed according to a target probability distribution asymptotically. Despite MCMC algorithms being able to generate unique and novel DAGs, them lacking speed can create a bottleneck.

Our contribution is to try to solve this problem by replacing the present MCMC algorithm with a neural network-based generative model. We used state-of-the-art models adjusted specifically to our problem. This includes leveraging a Variational Autoencoder (VAE) as our generative model with Graph Neural Networks (GNNs) incorporated as encoder and decoder. In addition to generation, numerous examinations were conducted in the latent space regarding its smoothness and also the interpolation and extrapolation possibilities.

Chapter 1

Introduction

In recent years deep generative models have undergone rapid progression for a wide variety of data domains, such as continuous data (e.g., images) and sequences (e.g., time series and natural language sentences). Representative methods, including generative adversarial networks (GAN) and variational autoencoders (VAE), learn a distribution, parameterized by deep neural networks from a set of training examples. Amid the tremendous progress, combinatorial structures, particularly graphs, remain substantial challenges for deep generative models.

Graphs are essential data structures that concisely encapsulate the flow of information in many important real-world domains. For example, relations between entities in knowledge graphs and social networks are well captured by graphs, and they are also good for modeling the physical world, e.g. molecular structure and the interactions between objects in physical systems. Thus, the ability to capture the distribution of a particular family of graphs has many applications. [21] For instance, by sampling graphical models, new configurations can be discovered that have certain global properties that are needed, for example, in molecule discovery. Obtaining the sentence structural information in text sequence which can be exploited to augment original sequence data by incorporating the task-specific knowledge, requires the ability to model graph distributions. [30] Distributions on graphs can also provide priors for Bayesian structure learning of graphical models.

Probabilistic models of graphs have been studied extensively from multiple perspectives. One approach is based on random graph models, which refer to the Erdős-Rényi (ER) or G(n,m), where all graphs with n nodes and m edges are assigned equal probabilities. [9] These have proven effective at modeling domains such as social networks as they are designed to capture certain robust graph properties, such as degree distribution and diameter. On the other hand, they struggle with more richly structured domains where small structural differences can be functionally significant, such as in chemistry or representing meaning in natural language, as they oversimplify the underlying distributions of graphs.

With the growing amount of graph-structured data, there is an increasing demand for developing deep models that are capable of complex graph generation.

To name a few, GraphRNN treats graph generation as a sequential generation problem and generates nodes and edges step by step; MoFlow [34] designs an invertible mapping between the input graph and the latent space and generates the graph (node feature and edge feature matrices) in one single step; MolGAN [8] designs a GAN-based graph generative model where a discriminator is used to ensure the properties of the generated graphs; GDSS [18] designs a score-based graph generative model which adds Gaussian noise to both node features and structures and reconstructs from Gaussian noise to obtain gener-

ated graphs during inference and DVAE[35] uses an asynchronous message passing scheme to encode and synthesize graphs. Additionally, many other graph-generative methods are utilized for deep graph generation. [37]

In our work, we adopted DVAE, one of the state-of-the-art models for DAG learning, in a Bayesian Network Structure learning task.

Our article is structured as follows:

- Chapter 2 summarizes the most important concepts for deep graph generation, including:
  - Graphs
  - Neural networks
  - Graph neural networks
  - Graph generation approaches
  - Bayesian networks
  - Markov Chain Monte Carlo methods
- Chapter 3 explains two applications of DAG generation: Neural Architecture Search and Bayesian Network Structure Learning
- Chapter 4 summarizes our work, and proposes solutions to existing problems for future works

Chapter 2

Research

We conducted a comprehensive survey on the current graph generative methods, examining both neural network- and random graph model-based methods. In the next section, we summarise the most relevant concepts.

2.1 Graphs

A graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". A graph is represented as $G = (V, E)$ where V is the set of vertices or nodes, and E is the set of edges. Let $v_i \in V$ to denote a node and $e_{ij} = (v_i, v_j) \in E$ to denote an edge pointing from v_j to v_i. The neighborhood of a node v is defined as $N(v) = \{u \in V | (v, u) \in E\}$. The adjacency matrix $\mathbf{A}$ is a $n \times n$ matrix with $A_{ij} = 1$ if $e_{ij} \in E$ and $A_{ij} = 0$ if $e_{ij} \notin E$. A graph may have node attributes $\mathbf{X}$, where $\mathbf{X} \in \mathbf{R}^{n \times d}$ is a node feature matrix with $x_v \in \mathbf{R}^d$ representing the feature vector of a node v. Meanwhile, a graph may have edge attributes $\mathbf{X}^e$, where $\mathbf{X}^e \in \mathbf{R}^{m \times c}$ is an edge feature matrix with $x^e_{v,u} \in \mathbf{R}^c$ representing the feature vector of an edge (v, u). [31]

2.1.1 Directed graph

A directed graph is a graph with all edges directed from one node to another. An undirected graph is considered as a special case of directed graphs where there is a pair of edges with inverse directions if two nodes are connected. A graph is undirected if and only if the adjacency matrix is symmetric.

2.2 Neural network

An artificial neural network (or simply neural network) consists of an input layer of neurons (or nodes, units), one or two (or even three) hidden layers of neurons, and a final layer of output neurons. Figure 5.1 shows a typical architecture, where lines connecting neurons are also shown. Each connection is associated with a numeric number called *weight*. The output, h_i, of neuron i in the hidden layer is,

$$h_i = \sigma\left(\sum_{j=1}^{N} V_{ij}x_j + T_i^{hid}\right), \tag{2.1}$$

where $\sigma(\cdot)$ is called activation (or transfer) function, N the number of input neurons, V_{ij} the weights, x_j inputs to the input neurons, and T_i^{hid} the threshold terms of the hidden neurons. The purpose of the activation function is, besides introducing nonlinearity into the neural network, to bound the value of the neuron so that the neural network is not paralyzed by divergent neurons.[28] A common example of the activation function is the sigmoid (or logistic) function defined as

$$\sigma(u) = \frac{1}{1+\exp(-u)}. \tag{2.2}$$

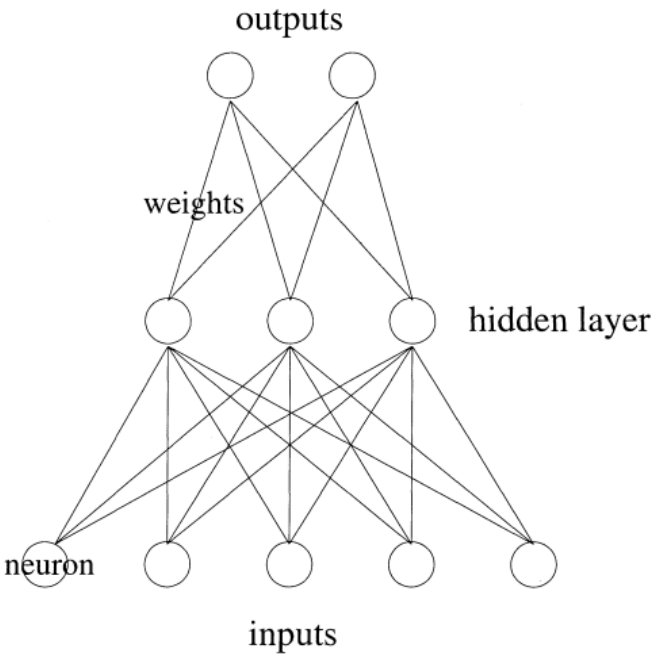

Figure 2.1: Architecture of a neural network [28]

2.3 Graph neural networks

Graph neural networks were proposed by Gori et al [16] in 2005 to construct a framework for neural networks that are capable of processing graph-structured data.

The intuitive idea underlying the proposed approach is that nodes in a graph represent objects or concepts, and edges represent their relationships. Each concept is naturally defined by its features and related concepts. Thus, we can attach a state to each node that is based on the information contained in the neighborhood. The state contains a representation of the concept denoted by and can be used to produce an output, i.e., a decision about the concept.[26] The representation of the target node is learned by

iteratively updating the state of the node using the information propagation mechanism of the graph until a global equilibrium state is attained.[31] [36]

The related notions are defined as follows: let the input graph be $G = (V, E, \mathbf{X}_V, \mathbf{X}_E)$, $V = v_1, v_2, ..., v_n$ represents the set of nodes, and $E = \{(i,j)|$ when v_i is adjacent to $v_j\}$ is the set of edges. x_i denotes the feature vector of node v_i, and $\mathbf{X}_V = \{x_1, x_2, ..., x_n\}$ is the set of feature vectors of all nodes. $x_(i,j)$ denotes the feature vector of edge (i,j), and $\mathbf{X}_E = \{x(i,j)|(i,j) \in E\}$ is the set of feature vectors of all edges. The input graph G is converted into a dynamic graph $G^t = (V, E, \mathbf{X}_V, \mathbf{X}_E, H^t)$ in the graph neural network model, where $t = 1, 2, ..., T$ represents time and $H^t = (h_1^{(t)}, h_2^{(t)}, ..., h_n^{(t)})$, $h(t)_i$ represents the state vector of node v_i at time t, which depends on the graph G^{t-1} at time $t-1$. The equation of $h_i^{(t)}$ is as follows:

$$h_i^{(t)} = f_w(x_i, x_{co(i)}, h_{ne(i)}^{t-1}, x_{ne(i)}) \tag{2.3}$$

where $f_w(\cdot)$ denotes the local transformation function with parameter w, $x_{ne(i)}$ is the set of feature vectors of all nodes adjacent to node v_i, $x_{co(i)}$ is the set of feature vectors of all edges connected to node v_i, and $h_{ne(i)}^{(t)}$ is the set of state vectors of all nodes adjacent to node v_i at time t. GNN updates the node status in an iterative manner.[36]

These early studies fall into the category of recurrent graph neural networks (RecGNNs). Amid the high computational cost, RecGNNs are conceptually important and inspired later research on convolutional graph neural networks. In particular, the idea of message passing is inherited by spatial based convolutional graph neural networks. [31]

2.3.1 Convolutional graph neural networks

Convolutional neural networks achieved remarkable results in image processing, and this success motivated the idea of generalizing the operation of convolution to non-Euclidian data. The idea of ConvGNNs is to build a node's representation by aggregating its own and its neighbor's features. Similarly to CNNs ConvGNNs stack multiple layers of filters to extract higher-level features. Graph convolutional networks can be categorized into two groups: Spectral-based and Spatial-based ConvGNNs.

2.3.1.1 Spectral based ConvGNNs

Spectral GNNs can be summarized into a general form: first transforming the spatial signal X through an MLP, then applying spectral filters parameterized by a polynomial of the normalized Laplacian $\hat{\mathbf{L}}$, and finally applying another MLP to the filtered signal. By designing/learning the polynomial coefficients, spectral GNNs can simulate a wide range of filters (low-pass, band-pass, high-pass) in the spectral domain, enabling GNNs to work on not only homophilic but also heterophilic graphs [29]

Spectral ConvGNNs are based on a branch of mathematics called graph signal processing. They use the normalized Laplacian representation of graphs which is defined as $\hat{\mathbf{L}} = \mathbf{I_n} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ where $\mathbf{D}$ is the diagonal matrix of node degrees $\mathbf{D}_{ii} = \sum_j(\mathbf{A}_{i,j})$. The normalized graph Laplacian is a real symmetric positive semidefinite, which allows it to be factored as $\hat{\mathbf{L}} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$ where $\mathbf{U} = [\mathbf{u_0}, \mathbf{u_1}, ..., \mathbf{u_{n-1}}] \in \mathbf{R}^{n \times n}$ is eigenvector matrix ordered sorted by eigenvalues in ascending order and $\mathbf{\Lambda}$ is the spectrum, i.e the diagonal matrix of eigenvalues. The eigenvalues of this matrix form an orthonormal space, $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ A

graph signal is vector of features of all the nodes of a graph where the i-th component is the value of the i-th node.

The graph Fourier transform performs a projection between the input graph signal and the orthonormal space whose basis is derived by the eigenvectors of the normalized Graph Laplacian. It is given by the equation $\mathcal{F}(x) = \mathbf{U}^T\mathbf{x}$, and its inverse $\mathcal{F}^{-1}(\hat{\mathbf{x}}) = \mathbf{U}^T\hat{\mathbf{x}}$, where $\hat{\mathbf{x}}$ represents the resulted signal from the graph fourier transform.

In this orthonormal space graph filters are defined on signals as $\mathbf{x} = \sum_i \hat{x}_i \mathbf{u}_i$ These filters then can be used to extract information about the graph struture and its features

Now the graph convolution of the input signal x with a filter $g \in \mathbf{R}^n$ is defined as

$$\mathbf{x} *_G g = \mathcal{F}^{-1}(\mathcal{F}(x) \odot \mathcal{F}(g)) = \mathbf{U}(\mathbf{U}^T x \odot \mathbf{U}^T g) \tag{2.4}$$

where $\odot$ denotes the element-wise product. If we denote a filter as $g_\theta = diag(\mathbf{U}^T g)$, then the spectral graph convolution is simplified as

$$\mathbf{x} *_G \mathbf{g}_\theta = \mathbf{U}\mathbf{g}_\theta\mathbf{U}^T\mathbf{x} \tag{2.5}$$

Though various models have emerged, spectral GNNs' expressive power is still under-researched. Moreover, these models differ mainly in the basis choices of the spectral filters, and it was proven by Wang et al. [29] that that even without nonlinearity, spectral GNNs can be universal under mild conditions. They proposed a novel spectral GNN called JacobiConv, that used Jacobi basis. JacobiConv outperforms the previous state-of-theart method on real-world datasets without using nonlinearity.

2.3.1.2 Spatial based ConvGNNs

On the other hand spatial based ConvGNNS does not operate in the signal space. Instead they define graph convolution based on the node's spatial relations. [31]

Images can be seen as special graphs each pixel acts as a node and is connected to its nearby pixels. Applying a filter is taking the weighted average of a pixel and its neighbors across multiple channels. Similarly, spatial-based graph convolution updates the central node by convolving its representation with its neighbors. From another point of view, they rely on information propagation similarly to RecGNNs.

The basic assumption of spatial based graph convolution is that each node v_i is additionally identified by its coordinates $\mathbf{p}_i \in \mathbb{R}^t$. In the case of images, $\mathbf{p}_i$ is the vector of two dimensional pixel coordinates, while for chemical compounds, it denotes location of the atom in two or three dimensional space (depending on the representation of chemical compound). In contrast to standard features $\mathbf{x}_i$, $\mathbf{p}_i$ is not changed across layers, but only used to construct a better graph representation. For this purpose the aggregation function is defined as:

$$\bar{\mathbf{h}}_i = \sum_{j \in N_i} \text{ReLU}(\mathbf{U}^T(\mathbf{p}_j - \mathbf{p}_i) + \mathbf{b}) \odot \mathbf{h}_j \tag{2.6}$$

where $\mathbf{U} \in \mathbb{R}^{t \times d}, \mathbf{b} \in \mathbb{R}^d$ are trainable parameters, d is the dimension of $\mathbf{h}_j$ and $\odot$ is element-wise multiplication. The pair $\mathbf{U}, \mathbf{b}$ plays a role of a convolutional filter which operates on the neighborhood of v_i . The relative positions in the neighborhood are transformed using a linear operation combined with nonlinear ReLU function. This scalar is used to weigh the feature vectors $\mathbf{h}_j$ in a neighborhood. By the analogy with classical convolution, this transformation can be extended to multiple filters. $\bar{\mathbf{U}} = [\mathbf{U}^{(1)}, ..., \mathbf{U}^{(k)}]$ and $\mathbf{B} = [\mathbf{b}^{(1)}, ..., \mathbf{b}^{(k)}]$ define k filters. The intermediate representation $\bar{\mathbf{h}}_i$ is then a vector defined by:

$$\bar{\mathbf{h}}_i(\bar{\mathbf{U}}, \mathbf{B}) = \bar{\mathbf{h}}_i(\mathbf{U}^{(1)}, \mathbf{b}^{(1)}) \oplus ... \oplus \bar{\mathbf{h}}_i(\mathbf{U}^{(k)}, \mathbf{b}^{(k)}) \tag{2.7}$$

where $\oplus$ denotes the vector concatenation. Finally, MLP transformation is applied to transform these feature vectors into new representation. [7] Spatial models are preferred over spectral models due to efficiency, generality, and flexibility issues. Spectral models are less efficient than spatial models, as they either need to perform eigenvector computation or handle the whole graph at the same time. Spatial models are more scalable to large graphs as they directly perform convolutions in the graph domain via information propagation. Another complication with spectral models is that they rely on a graph Fourier basis and generalize poorly to new graphs. They assume a fixed graph, meaning that perturbations to a graph would result in a change of eigenbasis. Spatialbased models, on the other hand, perform graph convolutions locally on each node where weights can be easily shared across different locations and structures. Spatial-based models can also operate on undirected graphs, which makes them more suitable for NAS and BNSL tasks. [31]

2.3.2 Latent variable approaches

In latent variable architectures, graphs are transformed into a low-dimensional latent representation, which follows a stochastic distribution. Then a sample is drawn from this latent distribution and decoded into a graph structure. A variety of latent variable approaches have been proposed, but most of them follow the encoder-sampler-decoder pipeline.

2.3.2.1 Encoder

The encoding function maps discrete graph objects to continuous, latent vectors. For an encoder to be capable of creating new, meaningful samples, the latent space must satisfy two important conditions: continuity and completeness.

- **Continuity:** continuity means that two vectors close to each other in latent space must have similar properties after decoding
- **Completeness:** the network has to decode the sampled latent vectors into "meaningful" objects

To fulfill these reuqirements probabilistic generative models are employed as encoders. Formally, the encoder outputs the parameters of a stochastic distribuition folowing a prior distribution.

Despite probabilistic models, without the regularization of the latent space, they can also significantly overfit the data and can learn distributions with close-to-zero variance and

deviating expected value, therefore acting as a normal autoencoder. Different models enforce this structural condition in different ways.

2.3.2.2 Sampler

The following step in the generation process is to sample a latent representation from the learned distribution. The sampler usually follows one of the following strategies: random sampling, controllable sampling

Random sampling During random sampling, latent code is randomly selected from the learned distribution.

Controllable sampling On the other hand, controllable sampling attempts to sample a vector that possesses the desired properties. The downside of controllable sampling is that it requires additional optimization of the latent space. Controllable generation modifies the randomly sampled or the encoded vector in a way that, after decoded, the generated graph possesses the desired properties. There are three frequently used approaches:

- **Disentangled sampling** separates the dimensions of the latent space and associates each variable with one property, forcing the latent variables to be disentangled from each other. With this approach, a desired property change of the generated graph can be achieved by modifying its latent representation.
- **Conditional sampling** concatenates a conditional code to the latent vector, which can explicitly control the properties of the generated graphs.
- **Traverse-based sampling** uses direct optimization in the continuous latent space to obtain samples witch the required properties. It can also control the generated graph properties by using heuristic-based search approaches (e.g interpolation, extrapolation) in the latent space.

2.3.2.3 Decoder

The decoder receives the sampled latent vector as its input and reconstructs a graph structure from it. Decoders can be divided into two categories: sequential generators and one-hot generators.

Sequential generators iteratively reconstruct the graphs, usually node by node, edge by edge. To use this approach graph nodes have to follow an ordering for the generation, in the case of DAGs, this can be their topological order. Sequential generation benefits from the flexibility that the number of nodes can be unknown beforehand. Another advantage of this approach is that after each step, constraint checks can be performed to ensure that the generated graph obeys certain restrictions. On the downside when generating large graphs with long sequences, the accumulated error can build up, resulting in a diversion from the original graph.

In contrast, one-shot generation regenerates/generates the graph as an adjacency matrix with optional node and edge features in a single step. The latent representation is fed into a neural network which outputs the adjacency and feature matrices. Many different neural networks can be utilized for this task, such as Convolutional Neural Networks (CNN), Graph Neural Networks (GNN), and multi-layer perceptron (MLP). One-shot generation

does not suffer from accumulating error during generation, but lacks flexibility as the number of nodes has to be known in advance. It also suffers from scalability issues as it scales as $\mathcal{O}(n^2)$ with respect to the number of nodes n, in the graph.

2.3.2.4 Deep generative models

Normalizing flows Normalizing flows are a series of invertible functions which create a mapping between the latent variables and the graphs via the change of variable theorem. The change of variables formula describes how to evaluate the densities of a random variable that is a deterministic transformation from another variable.

Formally, in a normalizing flow model, the mapping between the latent variables, Z and the observed variables $\mathbf{X}$ given by $f_\theta : \mathbb{R}^n \to \mathbb{R}^n$, is deterministic and invertible such that $X = f_\theta(Z)$ and $Z = f_\theta^{-1}(\mathbf{X})$.

Using change of variables, the marginal likelihood $p(x)$ is given by

$$p_X(x;\theta) = p_Z(f_\theta^{-1}(x)) \left| \det \left(\frac{\partial f_\theta^{-1}(x)}{\partial x} \right) \right| \tag{2.8}$$

Normalizing-flow-based models are usually trained by minimizing the negative log-likelihood over the training data.

Generative adverserial networks A generative adversarial network[15] is a kind of machine learning framework which consists of two main components the generator and the discriminator, embodied by neural networks. These two components play a zero-sum game, where the generator generates realistic objects (e.g graphs) and the discriminator tries to distinguish between real and artificial objects.

$$\min_{f_G} \max_{f_D} \mathcal{L}_{\text{GAN}}(f_G, f_D) = \mathbb{E}_{G\sim p(G)}[log f_D(G)] + \mathbb{E}_{z\sim p(z)}[log(1 - f_D(f_G(z)))] \tag{2.9}$$

GANs are especially popular in the computer vision domain, although they often suffer from various phenomena, such as posterior collapse or the vanishing gradient problem.

Variational autoencoder Variational autoencoder [19] (VAE) estimates the distributions of graphs $p(G)$ by maximizing the Evidence Lower Bound (ELBO) as follows:

$$\mathcal{L}_{\text{VAE}} = \mathbb{E}_{z\sim q_\phi(\mathbf{z}|G)} \log(p_\theta(G|\mathbf{z})) - D_{\text{KL}}(q_\phi(\mathbf{z}|G) \parallel p_\theta(\mathbf{z}))) \tag{2.10}$$

The first term is the reconstruction loss, a measure of similarity between the input and the reconstructed graph. The latter is a regularization term, that enhances the disentanglement of the latent space. This regularization appears in the form of KL divergence, which is an asymmetric measure of distribution similarity, it drives the learned distribution to the prior, usually Gaussian distribution.

The problem with VAE is that the sampling from a distribution that is parameterized by our model is not differentiable, in other words, to implement an encoder and decoder

as a neural network, backpropagation has to flow through random sampling and that is not possible. The reparameterization trick is a way to rewrite the expectation so that the distribution with respect to which we take the gradient is independent of parameter θ. To achieve this, we need to make the stochastic element in q independent of θ.

This eliminates the need to backpropagate through a sampling node by treating the sampling as noise.

$$\mathbf{z} = \mu + \sigma \odot \epsilon \tag{2.11}$$

Latent variable approaches have had a big impact on generative tasks, especially in the image domain. For example, Ramesh et al. developed a model called Dall-e [22], which uses a VAE-based architecture for text-to-image generative tasks. For the same problem, Yu et al. developed the Google Parti [33] an autoregressive model using a GAN encoding architecture. The success of these models suggests that latent variable approaches may be well suited for non-Euclidean data generation tasks.

2.3.3 Graph analytics tasks

With the graph structure and node content information as inputs, the outputs of GNNs can focus on different graph analytics tasks with one of the following mechanisms:

2.3.3.1 Node-level

Outputs tackle node regression and classification problems. Recurrent and Convolutional GNNs can produce high-level node features by information propagation or graph convolution. With the help of MLPs or a softmax output layer, GNNs are capable of performing node-level tasks in an end-to-end manner.

2.3.3.2 Edge-level

Edge-level tasks include edge classification and link prediction. With the utilization of a similarity function or a neural network label or connection, strength prediction can be performed on GNN outputs.

2.3.3.3 Graph-level

Graph-level outputs relate to the graph classification or generation tasks. To achieve a compact representation on the graph level, GNNs are often combined with pooling and readout operations. Graph generative tasks can be classified into two categories: graph structure learning and generative sampling.

- **Graph structure learning** simultaneously learns an optimized graph structure along with representations for downstream tasks. Unlike graph generation that aims to generate new graphs, the purpose of graph structure learning is to improve the given noisy or incomplete graphs.[37]
- **Generative sampling** learns to generate subsets of nodes and edges from a large graph. As most graph generative models do not scale to large single-graph datasets

such as citation networks, graph generative sampling could serve as an alternative approach to generate large-scale graphs by sampling subgraphs from a large graph and reconstructing a new graph. [37]

2.4 Graph generation

Many real-world systems may be considered networks of discrete object networks. These networks that exhibit non-regular topological features are called complex networks and are subject to intense research in a variety of fields including social science, biological, and information science.

The analysis of these networks includes answering questions about the properties of their nodes, the types of connections, their topology, and so on. These complex networks can be realistically represented as graphs, on which different operations can be performed to understand their underlying structure and mechanisms. Despite their commonality, scientists often suffered from a lack of actual data available for proper analysis, which led to the need for scalable, synthetic data.

With the emergence of machine learning and neural networks, many new models have been developed that outperform classical algorithms.

The concepts appearing in graph modeling can be categorized into three main groups: Generative, Feature-driven, and Domain-specific approaches.

- Generative: the generative class represents mechanisms that qualitatively explain graph patterns. After a graph has been constructed according to the specified rules, begins a discovery phase to find out what features it has. Then its feature similarity to real-world graph patterns can be analyzed and the generative rules can be modified according to the results.
- Feture-driven: feature-driven class aims to design a model, which quantitatively fit the required graph features. The development process is to design or tune a model, that satisfies the given the set of desired features.
- Domain-specific class includes graph generating methods with additional network attributes, such as community structure or edge weights.
- Latent attributes: the concept assumes that connections between nodes depend on some deep-rooted properties of the nodes represented by their attributes.

2.4.1 Generative approach

Based on the ER model which is the most widely used and yet the least realistic model of a random graph, designers of RG models developed many algorithms trying to explain the phenomenas apperaing in real world complex networks. These algorithms can be grouped into many different categories from which we will comprise the following: Classic, Local rules, Recursion and Latent attributes

2.4.1.1 Classsic

The naive interpretation of randomness is to connect each pair of nodes independently. The ER model, which is possibly the most widely known model, connects each pair of nodes

on a set of n nodes, with a constant probability p. Although the ER model has unrealistic properties, it is rich with theoretical results. Another well-known construction, named the small-world model, aims to achieve both low diameter together and high clustering. The Watts-Strogatz model starts with a regular lattice, where each node has an equal amount of neighbours. Each edge is then replaced with a random edge with probability p. [9].

2.4.1.2 Local rules

Local rules mean that the graph's growth is governed by rules affecting a node and its neighbors. An example of these rules is called triadic closure, which is well observable in social networks. It states that if the nodes u and v have a common neighbor the probability of the edge (u, v) is higher than in the same they don't share common connection.

2.4.1.3 Recursion

Recursive algorithms grew from the observation that networks are formed by larger communities composed of smaller communities and so on, forming a scale-free, self-similar, hierarcical structure. One of the most influential model, R-MAT [3] produces a graph by recursively sampling edges from a partitioned adjacency-matrix.

2.4.1.4 Latent attributes

The motivation of this idea a sociological theory called homophily, which claims similarities attract. For example people of close age, interest, geographical location are more likely to be connected in the network. This concecpt is incorporated in the model by expressing edge probability as a function of node attributes. $P_{ij} = f(\overrightarrow{a}_i, \overrightarrow{a}_j)$

Geometric approaches In geometric approaches[6] node attributes are interpreted as geometrical coordinates, and it proved to be productive in modeling ad hoc wireless networks, sensor-actuator networks, and the Internet, where physical distance between the nodes directly influences their connectivity. The attempts to adapt complex networks for geometric framework led to the assumption that hyperbolic geometry underlies their structure. For example, power-law exponent is a function of the space curvature. In other words, a more-relevant distance metric on graphs is based on the shortest path (geodesic line), and it is rather hyperbolic than Euclidean. Moreover, hierarchical structure and treelike patterns, common in real networks, better fit into hyperbolic space.

In the Embedding based random graph model (ERGG), each node of a directed graph is associated with a vector $\mathbf{r_i}$ being a triple u_i, v_i , and Z_i . Link probability is based on a directed softmax model, where the conditional probability of the edge $i \to j$ is $P(j|i) = \exp(u_i \cdot v_j - Z_i)$, with Z_i being a normalization coefficient. At the construction phase, edge $i \to j$ is created if $P(j|i)$ is above a threshold t_G. Representations $\{r_i\}$ and the threshold are learned to fit best to a given graph G.

As a resume, we note that the selection of graph geometry could be treated as the selection of metric in the node vectors space. The simplest geometry is Euclidean, but the hyperbolic metric is more sophisticated and efficient.

Node labeling The core assumption in the concept of node labeling is that edge probability is defined by the similarity of node labels. In Random typing graphs (RTG), a

random typing process is used to generate character sequences terminating with "space.". Each node represents a unique word. At each algorithm step, two label nodes are created by generating one letter l. Each letter has its typing probability pl. If a connection exists between the created nodes the edge weight is incremented, otherwise a connection is made.

2.4.2 Feature-driven class

2.4.2.1 Analytical solution

Carefully designed algorithms can generate graphs that satisfy the desired constraints and characteristics. Such models are mathematically tractable and allow precise control of the graph features, making them useful for analysis. The disadvantage of these models is that they are poorly representative of real networks with complex properties such as subgraph distribution. These models also suffer from low accuracy when several feature constraints are formulated. A common method is to modify the graph in an iterative madder, satisfying the desired charateristics one-by-one, however several non-trivial problems have not been solved using analytical methods such as implementing a target degree sequence and connected components together. Nevertheless, they are still widely used because they have been studied extensively and can serve as null models. Also, in practice approximating the given constants is a sufficient trade-off for the ability to control a large number of parameters. These RG models can serve as generators for benchmark graphs, one of the most common algorithm in this category is called LFR , which is capable of generating complex, directed weighted graphs with overlapping community structure.

2.4.2.2 Graph editing

Edge switching The conventional approach to graph randomizing is edge switching or edge rewiring. It is recurrently performed on the Graph in a way that the set of Constraints C remains content. Pairwise edge switching is the most common edge-switching operation, since it does not change the node degrees: a pair of edges $i \to j, k \to l$ is rewired into $i \to l, k \to j$. Markov chain's are often used by this class of graph generators. Given an input graph G_0, at each step the algorithm picks a random pair of edges to be switched. This way the chain has a stationary distribution uniform over all graphs with the same node degrees and it is also irreducible. In the case of more-elaborate constraints C, a standard Monte Carlo sampling techniques are employed to achieve a Markov chain with a wished stationary distribution corresponding to C. For example Ying Xiaowei and Wu Xintao [32] use the Metropolis-Hastings algorithm to generate graphs with a target distribution of features $g(S)$. Unfortunately, MCMC following comlex requirements suffer from two problems: the method may not be ergodic, i.e not all valid states could be reachable via allowed steps and an increase in chain convergence time.

Representation editing: Another way of graph editing is rather than modifying graph G itself, modify its representation $R(G)$ given that it reflects the graph features properly. Then the problem is shifted to finding adequate representations and operations of transforming G to $R(G)$ and backwards.

It was an important observation by Gutfraind et al [17] that the properties of real networks that should be preserved during generation are not only those measured at the finest resolution but also those that can be measured at the coarse resolutions. Their model called

Musketeer used a hierarchical construction of aggregators that was capable of replicating several real-world original networks, but without the guarantee of planarity.

The ERGG model can be classified not only as a geometric latent variable approach but also as a feature driven, graph editing solution. In ERGG the editing occurs at the level of vector representation of its nodes. After embedding the input graph in the vector space, the new graphs are constructed by samplingprobability distribution approximating the distribution of the node vectors with a small Gaussian noise mixed into the samples. Finally, new nodes are connected with edges giving a new graph. Expirements show that besides reproducing main graph features, these models provide variability of graphs, that are generated from one input graph, close to natural variability within a domain.[10] [9]

2.4.3 Domain specific class

The above RG models usually work on simple, directed graphs, however many real world applications require the graphs to possess more complex features, such as a given community structure, or node and edge properties, like weights. The domain specific class includes all RG concepts that are not covered by the above RG approaches. Although this category is rather broad, most problems concern either community structures or weighted edges.

2.4.3.1 Weighted edges

Weighted edge graphs appear in many complex network. Weight could represent the strength of molecular bonds, the cost of travelling between nodes or the measure of "friendship" in social networks. Many algorithms form weighted edges by summing the multi-edges connecting two nodes, and are generalized to process such graphs.

2.4.3.2 Community structured

Many networks have been found to exhibit a "community structure", i.e. they are naturally partitioned into communities or modules, with dense connections within communities but sparser connections between communities. Communities are of particular interest both in their own terms, as functional building blocks of networks, and for their insights into the underlying mechanisms of how networks are formed. [20] Both generative and feature-driven methods exist on this topic, but their essential concept is the same in most cases, they try to attach a label to each nodes indicating which community the node belongs to.

With the development of deep neural networks, many of these fundamental concepts have been adopted and used in the development of deep graph generation methods. For example, VAEs both use the latent variable approach to encode graphs in lower dimensional latent space and use representation editing to ensure that the generated graphs exhibit the desired characteristics.

2.5 Bayesian networks

Bayesian networks (BNs) represent systems as a network of interactions between variables from primary cause to final outcome, with all cause-effect assumptions made explicit.[4]

Formally Bayesian networks are defined as as a pair $(\mathcal{G}, \mathcal{O})$ that encodes a joint probability distribution over a finite set $\mathcal{X} = X_1, ..., X_n$ of categorical variables. It is composed of:

1. A directed acyclic graph (DAG) $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ whose nodes V correspond to the variables in X and arcs E represent direct dependencies between variables
2. A collection of conditional mass functions $\mathcal{O}$ that define the behavior of each variable X_i given its parents Π_i in the graph[25]

Although the representation of the full joint table $P(X)$ takes exponential space in the number of variables n. This complexity can be avoided thanks to the so-called Markov condition, which states that in a Bayesian network every variable is conditionally independent of its non-descendant non-parents given its parents. By using this condition, it is possible to represent the joint table in a compact form, as a multiplication of local mass functions:

$$p(\mathbf{x}) = p(x_1, ..., x_n) = \prod_i p(x_i|\pi_i) \tag{2.12}$$

, where x is an instantiation of all the variables in $\mathcal{X}$, x_i is the value of variable X_i in $\mathbf{x}$, π_i is an instantiation of all the variables in the parent set Π_i compatible with $\mathbf{x}$. The representation now takes still exponential space but only in the size of the largest parent set.[25]

The ability to integrate multiple issues, interactions, and outcomes, combined with the potential to investigate tradeoffs make BNs suitable for modeling environmental systems. Furthermore, they are apt for utilising data and knowledge from different sources and handling missing data. BNs readily incorporate and explicitly represent uncertain information, and this uncertainty is propagated through to and expressed in the model outputs. BNs are based on a relatively simple causal graphical structure, meaning they can be built without highly technical modelling skills and be understood by non-technical users and stakeholders. In BNs, variables are represented by nodes, which are linked by arcs that symbolise dependent relationships between variables.[4]

2.6 Markov Chain Monte Carlo

The process through which probabilistic model quantities can be calculated is referred as inference.

Due to the nature of probabilistic models exact inference is likely intractable, so we have to resort to some form of approximation. This intractablility is caused by the summation of a discrete distribution of many random variables or integration of a continuous distribution of many variables during inference.

$$s = \int p(\mathbf{x}) f(\mathbf{x}) d\mathbf{x} = E_p \left[f(\mathbf{x}] \right. \tag{2.13}$$

To avoid these unmanageable calculations usually Monte Carlo sampling is used, which draws independent samples from the probability distribution, then repeats this process until the desired quantity is well approximated.

However, Monte Carlo sampling does not behave well in high-dimensions. Firstly because as the number of parameters increases, the volume of the sample space exponentially grows, this is also known as the curse of dimensionality.

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{x}^{(i)}) \tag{2.14}$$

Second, and perhaps most critically, random samples taken from the target distribution with Monte Carlo Sampling are assumed to be independent and drawn independently. This is typically not the case for inference with Bayesian structured or graphical probabilistic models.

Markov chain is a method for systematically generating a sequence of random variables where the current value probabilistically depends on the value of the prior variable. Specifically, the subsequent variable selection depends only on the last variable in the chain.

$$P(X_{n+1} = k | X_n = k_n, X_{n-1} = k_{n-1}, ..., X_1 = k_1) = P(X_{n+1} = k | X_n = k_n) \tag{2.15}$$

MCMC is essentially Monte Carlo integration using Markov chains. Monte Carlo integration draws samples from the required distribution and then forms sample averages to approximate expectations. Markov chain Monte Carlo draws these samples by running a cleverly constructed Markov chain for a long time. [13]

In cases when the next state probability distribution cannot be sampled directly the Metropolis-Hastings algorithm can be a convenient solution.

They are based on a Markov chain whose dependence on the predecessor is split into two parts: a proposal and an acceptance of the proposal. The proposals suggest an arbitrary next step in the trajectory of the chain and the acceptance makes sure the appropriate limiting direction is maintained by rejecting unwanted moves of the chain. [12]

More formally the Metropolis–Hastings algorithm generates a collection of states according to a desired distribution $P(x)$. To accomplish this, the algorithm uses a Markov process, which asymptotically reaches a unique stationary distribution $\pi(x)$ such that $\pi(x) = P(x)$ [23] A Markov process is uniquely defined by its transition probabilities $P(x'|x)$, the probability of transitioning from any given state x to any other given state x'. It has a unique stationary distribution $\pi(x)$ when the following two conditions are met: [23]

1. Existence of stationary distribution: there must exist a stationary distribution $\pi(x)$. Detailed balance is a sufficient condition for the existence of $\pi(x)$, which requires that $x \to x'$ is reversible, i.e for every pair of states x, x', the probability of being in state x and transitioning to state x' must be equal to the probability of being in state x' and transitioning to state x, $\pi(x)P(x'|x) = \pi(x')P(x|x')$.
2. Uniqueness of stationary distribution: the stationary distribution $\pi(x)$ must be unique. This is guaranteed by ergodicity of the Markov process, which requires that every state must satisfy the following constraints:
    - **Aperiodicity** — the system does not return to the same state at fixed intervals
    - **Positive recurrence** —the expected number of steps for returning to the same state is finite

Provided that specified conditions are met, the empirical distribution of saved states $x_0, ...x_T$ will approach $P(x)$. The number of iterations (T) required to effectively estimate $P(x)$ depends on the number of factors, including the relationship between $P(x)$ and the proposal distribution and the desired accuracy of estimation.

Markov chain Monte Carlo (MCMC) methods are powerful tools for inferring Bayesian network (BN) structures from data [1]. BNMCMC is software which implements MCMC methods for this specific problem.

To infer BN structures from data, the samplers in BNMCMC explore the discrete space of candidate BNs. At each sampling iteration, these samplers iterate through all possible neighbourhoods of a particular BN and sample the best scored one among them- The neighbourhood of a particular BN consists of all networks that can be obtained by modifying a pair of nodes by adding, deleting or reversing an edge. Experimental data is used for learning the conditional probability tables (CPTs) of nodes of a candidate BN and a score is evaluated using those CPTs by applying a probabilistic model which refers to the posterior probability of the network given data. [1]

Chapter 3

Applications

In the next section, we briefly describe the two main research areas of DAG generation. Neural architecture search and Bayesian network structure learning both aims to automate the design of models for specific datasets, thereby reducing the time and expert knowledge required to solve these problems.

3.1 Neural Architecture search

For a given data set, finding a neural architecture that gives decent results in a relatively short time is a difficult problem that requires expert knowledge and considerable hand-tuning. These tasks are time consuming and the cost of data scientists is very high, even compared to the cost of high-end GPUs in public clouds. In addition, human availability is generally low and prone to errors. This problem is a really hot topic in today's machine learning community, and a whole new branch of ai, called Neural Architecture Search, has grown up to solve it.

Neural Architecture Search (NAS) is the process of automating architecture engineering and the logical next step in automating machine learning. Already by now, NAS methods have outperformed manually designed architectures on some tasks such as image classification, object detection or semantic segmentation. NAS can be seen as subfield of AutoML and has significant overlap with hyperparameter optimization and meta-learning. We categorize methods for NAS according to three dimensions: search space, search strategy, and performance estimation strategy [11]

- **Search Space:** The search space defines which architectures can be represented in principle. Due to the substantial size of the search space, the search process can be extremely time consuming. To address this problem, prior knowledge of typical properties of architectures, well suited to the task, can be incorporated, thereby reducing the size of the search space. However, human involvement can introduce bias into the process, potentially preventing the discovery of new architectural building blocks.
- **Search Strategy:** The exploration process of the search space (which is often exponentially large or even unbounded) is defined by the search strategy. It has to take into account the exploration-exploitation trade-off, i.e. it has to find well-performing architectures in a limited time while avoiding premature convergence to a region of suboptimal architectures. Seach strategies can be mainly categorized into:

  - Reinforcement learning methods which train controllers to generate architectures with high rewards in terms of validation accuracy
  - Bayesian optimization based methods,which define kernels to measure architecture similarity and extrapolate the architecture space heuristically
  - evolutionary approaches which use evolutionary algorithms to optimize neural architectures
  - differentiable methods which use continuous relaxation/mapping of neural architectures to enable gradient-based optimization. [35]

- **Performance Estimation Strategy:** The NAS typically seeks to find architectures that yield high predictive performance on unseen data. Performance estimation is the estimation of this performance: the most trivial approach is to test the architecture against data using standard training and validation, unfortunately this method is computationally expensive and limits the number of architectures that can be evaluated. Much recent research has therefore focused on developing methods that reduce the cost of these performance estimates. [11]

3.2 Bayesian Network Structure Learning

Although Bayesian networks are a great tool for Bayesian inference, constructing the network structure which truly represents the underlying causal mechanism based solely on the observational data is far from trivial. Several researchers have studied this application, which has been proven to be NP-Hard, therefore analytical algorithms, such as dynamic programming or shortest path approaches, are only applicable for small-scale probems. Also considering that the available observed data is usually incomplete, there may exist several similarly well performing network structures. These problems are tackled by Bayesian network structure learning (BNSL), i.e the learning of a Bayesian network from observational data. BNSL has been studied extensively during the last decade, one reason for this is because it can be used to automatically construct Bayesian networks for inferring possible causal relations. [27] Bayesian network learning has been widely used in many scientific regions, including bioinformatics for the interpreting and discovering gene regulatory pathways, variable selection for classification and algorithm design for optimally solving the problems under certain conditions

The problem of learning the structure of a Bayesian network from a complete dataset of d datapoints $\mathbf{D} = D_1, ..., D_d$ corresponds to determining the set of directed arcs E for the DAG $G = (X, E)$, using some criterion that specifies the quality of a structure. This can also be stated as choosing for each variable X_i its parent set i . The usual assumption is for the data to be complete. The case of missing data currently represents a bottleneck for structure learning, as few methods can properly manage it.

The task is computationally non-trivial due to the enormous size of the space of possible graphs G, growing super-exponentially in the number of nodes n.There are two major approaches for tackling this problem: score-based and constraint-based.

- **Score-based:** in the score-based approach, an evaluation score metric is defined on the Bayesian networks, measuring how well the networks fit the data. Then the method searches over the space of DAGs, aiming for the maximal scoring architecture. Commonly used scores include BIC and BDeu, mostly based on marginal likelihood.

- **The constraint-based:** Constraint-based algorithms identify conditional independence constraints by using hypothesis tests to learn independences among the variables in the model. Following these constraints, the DAG is in turn built. Their performance is critically determined by the adopted hypothesis test. [25] [14] This approach works well with some other prior (expert) knowledge of structure but requires lots of data samples to guarantee testing power. So it is less reliable when the number of sample is small.

Due to the NP-hardness, however, exact algorithms such as dynamic programming or shortest path approaches can only solve small-scale problems.

As both neural and Bayesian networks can be considered as DAGs, deep graph generative methods can be applied to solve these problems. For example, NASGEM[5] uses an encoder-decoder architecture to encode neural networks in lower dimensional space, similarly DVAE[35] uses a latent variable approach but with an asynchronous message passing scheme. Both studies have achieved remarkable success, although this is still a very nascent and unexplored area of machine learning.

Chapter 4

Learning of graph posteriors

After an exhaustive scientific research in graph generation, we found that for learning Bayesian Network structures a Variational Autoencoder model based on DVAE could provide the best results.

4.1 D-VAE

D-VAE[35] is GNN architecture which,uses an asynchronous message passing scheme to encode and decode DAGs. In contrast to the simultaneous message passing in traditional GNNs, D-VAE allows encoding computations rather than structures.

4.1.1 Encoding

The encoder of D-VAE can be considered a graph neural network, which uses an asynchronous message passing scheme. Every DAG fed into the network is expected to have a single starting node. In case that there are multiple such nodes, a pseudo starting node is added to the graph, connecting to all of them.

D-VAE uses an Update function $\mathcal{U}$ to update a node's hidden state by its neighbours messages. The hidden state therefore is given by:

$$\mathbf{h_v} = \mathcal{U}(x_v, \mathbf{h}_v^{\text{in}}) \tag{4.1}$$

where x_v is the encoding of the node's type and $\mathbf{h}_v^{\text{in}}$ is the aggregate of the incoming messages from v's neighbours. $\mathbf{h}_v^{\text{in}}$ is calculated by the aggregation function $\mathcal{A}$:

$$\mathbf{h}_v^{\text{in}} = \mathcal{A}(\mathbf{h}_u : u \to v) \tag{4.2}$$

where $u \to v$ is a directed edge from u to v and $\{\mathbf{h}_u : u \to v\}$ is the multiset of v's predecessors hidden states

Unlike in simultaneous message passing, in D-VAE the message passing for a node must wait until all of it's predecessors's hidden states have been computed. To ensure that

a node can start it's message passsing when the algorithm reaches it D-VAE iterates on nodes following a topological order of the DAG.

Once the hidden state of all nodes has been calculated, the hidden state of the terminating node, the one without any successors, is used as the output of the encoder. This graph state is then fed into two MLPs to obtain the mean and the variance parameters for the posterior approximation. If there exists several nodes without successors, we again add a virtual end node connecting all of them.

Although topological order is not necessarily unique for DAGs it was proven by Zhang et al. [35] that all of them produces the same output under the condition that the aggregation function is invariant to the order of its inputs. It was also shown that the DVAE encoder maps its input injectively to the latent space if the aggregation and update functions are injective. An important aspect of DVAE is that it encodes computations on graph structures, and not the structures themself. Injectively encoding graph structures would require solving the graph isomorphism problem, which is categorized as an NP hard problem. Fortunately, here what we are really interested in are the calculations, not the structures, as we do not aim to distinguish between two different G1 and G2 structures, as long as they represent the same calculation.

For modeling and learning injective functions, we rely on neural networks, thanks to the universal approximation theorem. We used gated sum as our A function:

$$\mathbf{h}_v^{\text{in}} = \sum_{u \to v} g(\mathbf{h}_u) \odot m(\mathbf{h}_u) \tag{4.3}$$

where m denotes a mapping network and g a gating network. The update function was modelled by a gated recurrent unit (GRU), which treated in_v as its input:

$$\mathbf{h}_v = \text{GRU}_e(\mathbf{x}_v, \mathbf{h}_v^{\text{in}}) \tag{4.4}$$

4.1.2 Decoding

Similar to the encoder, the encoder uses an asynchronous message-passing protocol to learn intermediate nodes and graph states. The decoder generates the hidden states of nodes by updating them using another GRU, denoted by GRU_d. For the latent vector z to be decoded, we first apply an MLP to map z to h_0 as the initial hidden state to be fed to GRU_d .

1. Compute v_i's type distribution using an MLP f_{add_vertex} (followed by a softmax) based on the current graph state $\mathbf{h}_G := \mathbf{h}_{v_{i-1}}$.

2. Sample v_i's type. If the sampled type is the ending type, stop the decoding, connect all loose ends (nodes without successors) to v_i , and output the DAG; otherwise, continue the generation.

3. Update v_i's hidden state by $\mathbf{h}_{v_i} = \text{GRU}_d(\mathbf{x}_{v_i}, \mathbf{h}_{v_i}^{\text{in}})$, where $\text{in}_{v_i} = h_0$ if $i = 1$; otherwise, in_{v_i} is the aggregated message from its predecessors' hidden states given by equation (2.4).

4. For $j = i-1, i-2, ..., 1$:

   4.1 compute the edge probability of (v_j , v_i) using an MLP f_add_vertex based on $\mathbf{h}_{v_j}$ and $\mathbf{h}_{v_i}$

   4.2 sample the edge;

4.3 if a new edge is added, update hvi using step 3

The above process iteratively samples new node types and then sequentially predicts whether to connect the new node to the existing ones or not based on the current node's state. This algorithm requires the maintenance of hidden states for both the new and the already existing nodes, unlike in RNNs where states for old nodes are not maintained. Sampled edges always point from previous nodes towards the new vertex, therefore the generated graph is guaranteed to be acyclic.

The generating process stops when in step 2 an end-type node is sampled.

Teacher forcing Teacher forcing is a training algorithm for recurrent neural networks. Its core idea is that instead of feeding the previous output of the network into the network in the next training step, the observed data is given as input thus forcing the RNN to stay close to the ground-truth sequence.

As DVAE generates nodes sequentially this idea could be incorporated during training. The loss consisted of 3 parts:

1. Negative log likelihood between the generated and the true node types
2. Cross entropy between the edge probabilities and the ground truth edges
3. KL divergence of the models mean and variance and a standard normal distribution

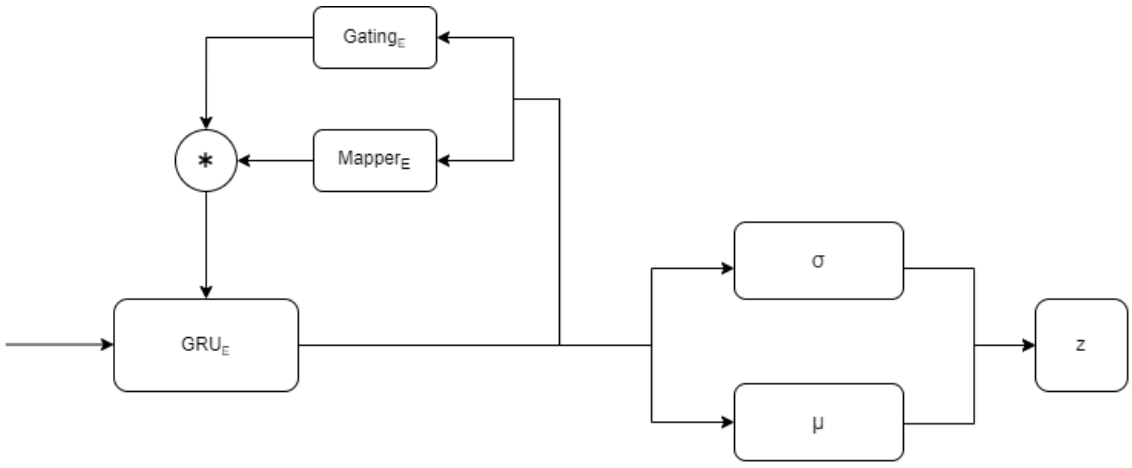

Figure 4.1: The encoder architecture

4.2 Posterior learning of small scale DAGs

First we wanted to test the model's expressive power, so we constructed a smaller scale problem. We investigated whether the model was capable of generating samples from a smaller graph class with uniform distribution. To conduct this survey first we had to choose a class of graphs with manageable size and generate all possible samples in that class. The number of DAGs as a function of the number of vertices, $G(n)$, is super-exponential in n and is given by the following recursive equation:[24]

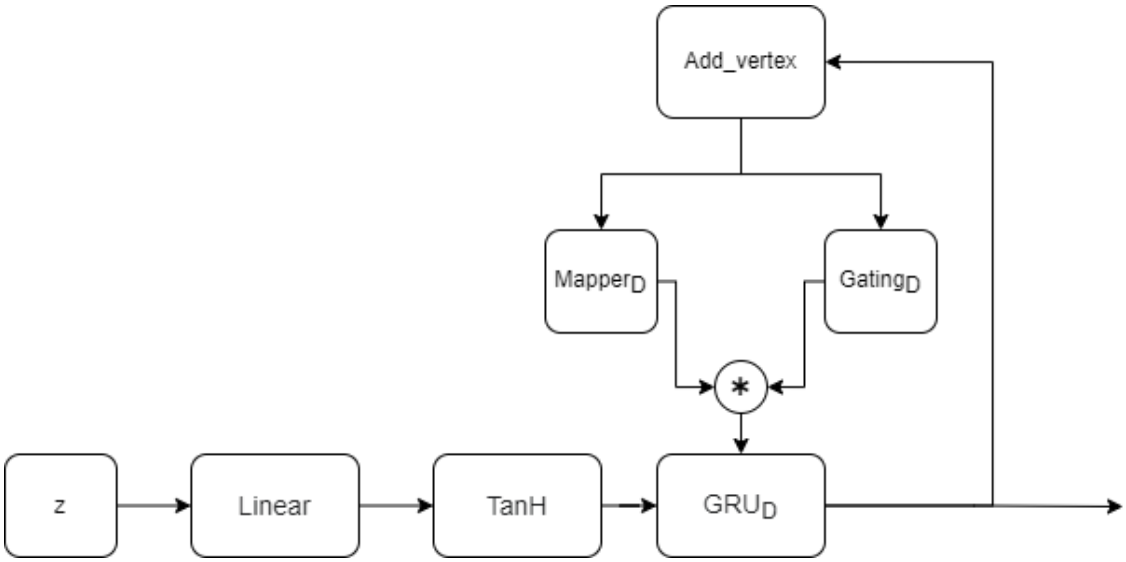

Figure 4.2: The vertex decoder

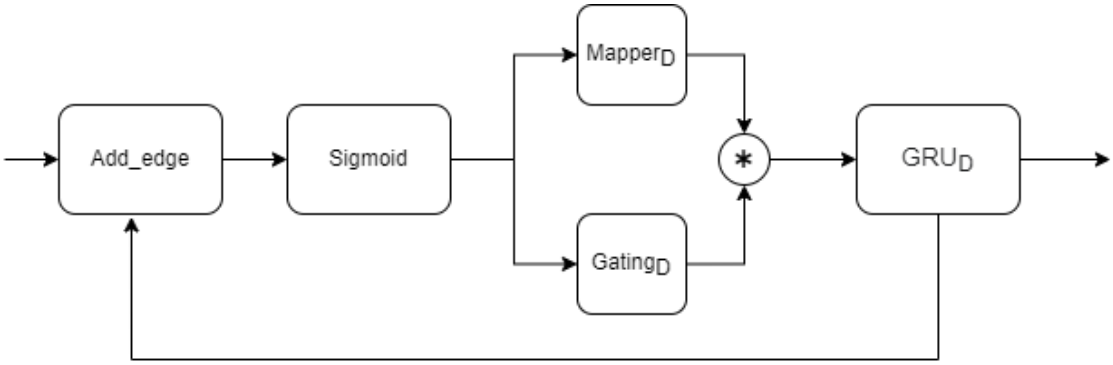

Figure 4.3: The edge decoder

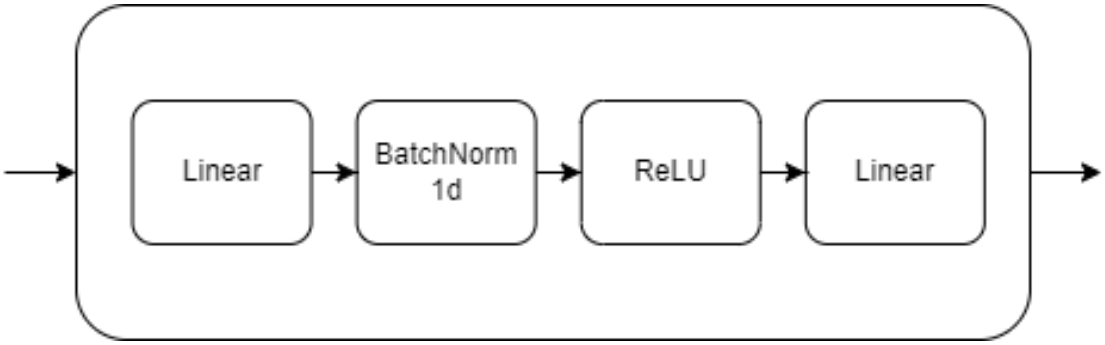

Figure 4.4: Add_vertex and add_edge block architectures

$$a_n = \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} 2^{k(n-k)} a_{n-k} \tag{4.5}$$

,with $a_0 = 1$.

n	number of DAGs
1	1
2	3
3	25
4	543
5	29.281
6	3.781.503

Table 4.1: Number of possible DAGs as a function of vertex number

Based on the values of the function and required training time we chose the DAGs containing 4 vertices.

4.2.1 Generating 4 vertex DAGs

An important attribute of DAGs is that their adjacency-matrix is upper-triangular. We used this property for generating all possible DAGs. We started with a DAG containing the maximal number of edges and iteratively removed edge combinations until we reached 0 edges. During each iteration the node labels were permuted and the resulting graphs were appended to a list. Duplications were filtered each iteration. Our algorithm was far from optimal and but it was capable of generating 4 and 5 node graphs in reasonable time.

```
   Input: n - number of vertices in graph
   Output: results - list containing all possible DAGs with n vertices
 1 maxEdges ← n(n−1)/2
 2 fullGraph ← graph with upper-triangular adjacency-matrix
 3 for i ← 0 to maxEdges do
 4     combinations ← generateCombinations(possibleEdges, i)
 5     for combination in combinations do
 6         currentGraph ← removeEdges(fullGraph, combination)
 7         permutedGraphs ← permuteLabels(currentGraph)
 8         for permutedGraph in permutedGraphs do
 9             if results not contains permutedGraph then
10                 results.append(permutedGraph)
```

4.2.2 Model

By default DVAE inserts a virtual starting and ending node during generation, and connects nodes without predecessors to the start node and nodes without successors to the end node. It was recommended that the training dataset also contained these virtual nodes and edges. For these reasons we extended the DAGs with the necessary nodes and edges. The available source for DVAE contained multiple models, each tailored to a specific problem type, such as NAS or BNSL. We used DVAE_BN as each graph in the dataset contained each nodetype only once, therefore acted as a Bayesian Network. DVAE_BNs encoder differs from the standard DVAE encoder.

One modification is that the aggregation function is changed to:

$$\mathbf{h}_v^{in} = \sum_{u \rightarrow v} g(\mathbf{x}_u) \odot m(\mathbf{x}_u) \tag{4.6}$$

Compared to the original update function, h_u is replaced by the node type feature x_u. This is because of differences between the calculations on the neural architectures and on Bayesian networks. In a neural network, the signal flows through the network, in a way that the output of one layer is used as the input of the subsequent layers.

While in a neural network the focus is on the output of the final layer, in a Bayesian network the graph represents the set of conditional relations between variables instead of the computational process.

In DVAE_BN the graph state calculations is also modified, due to the decomposibility of the score. Instead of using the ending node state, the graph state is calculated as the sum of every individual node state.

4.2.3 Results

We ran the training with many different configurations, though we highlight only three with interesting results. For the first configuration we used a 2 dimensional latent space, with 64 GRU size. The second configuration was quite similar, but we added a Batch Normalization layer to the add_edge and add_vertex components. Although the second configuration was able to achieve lower train loss, on the test dataset it produced far worse results. For our last configuration we removed the BatchNorm layer, doubledthe latent dimensions and GRU hidden size to 4 and 128. As you can see the last configuration was able to reconstruct the graphs with the greatest accuracy. We were able to achieve near 0 KL-divergence in all three cases.

Parameter	Configuration 1	Configuration 2	
Learning rate	10^{-4}	10^{-4}	10^{-4}
Batch size	16	16	16
GRU size	64	64	128
Latent dimensions	2	2	4
Batch normalization	No	Yes	No

Table 4.2: Configurations

We also conducted experiments in the latent space of the models. First we inspected the locations of encoded graphs in the latent space for Configurations 2 and 3. To be able to display the 4 dimensional space we used PCA on the latent variables, therefore reducing its dimension to two. We also applied this transformation to the 2 dimensional case.

In the 4 dimensional case our model could achieve a much more symmetric encoding. In our last experiment we sampled 1000 points from the latent space, decoded and each point 10 times, resulting in 10.000 sampled graphs. Unfortunately the results did not show order in the latent space regarding the graph that the sampled was decoded to.

We also examined the fequency of the decoded graphs. With a uniform sampling distribution each possible graph should have been sampled approximately $19 - 20$ times. The following table summarizes the statistics of the random generation process.

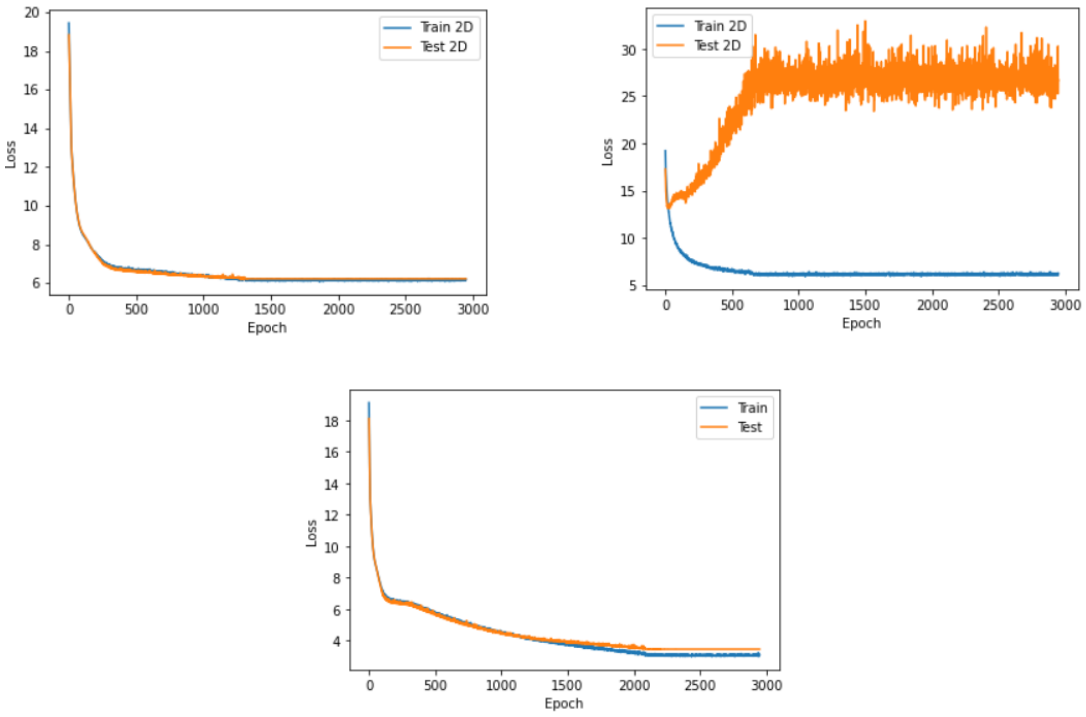

Figure 4.5: Train and test results for Configurations.

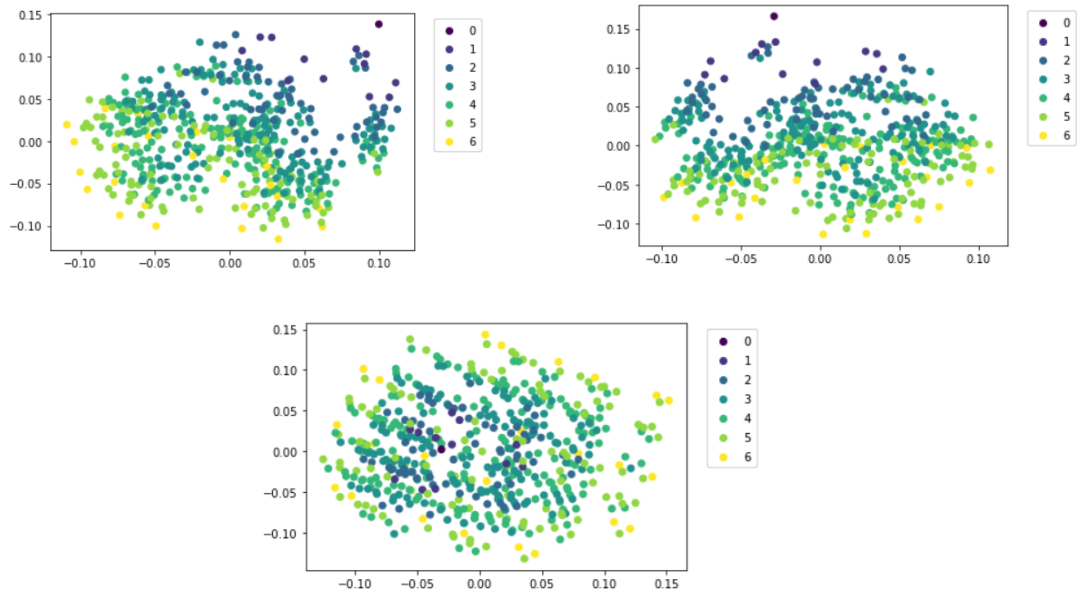

Figure 4.6: The encoded graphs in the latent space. (1) Configuration 2 without PCA (2) Configuration 2 wit PCA (3) Configuration 3 with PCA The colouring is based on the graphs edge count

These experiments showed that, although with Configuration 4 we were able to achieve some degree of order in the encoding of DAGs, the models were not able to generate graphs with uniform distribution.

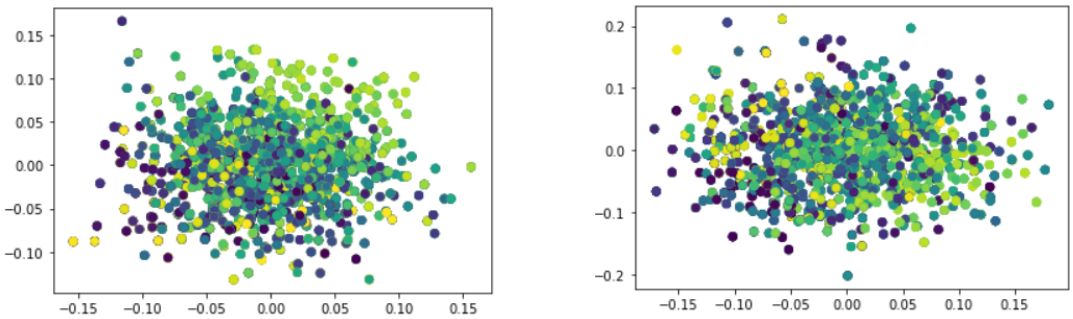

Figure 4.7: The sampled points coloured according to the graphs they were decoded to.

Statistics	Configuration 2	Configuration 3
Generated graphs	10000	10000
Valid graphs	8102	9940
Invalid graphs	1898	60
Average frequency of generated graph	15.11	18.51
Median frequency of generated graph	11	16
Variance of frequencies	12.71	12.42

Table 4.3: Configurations

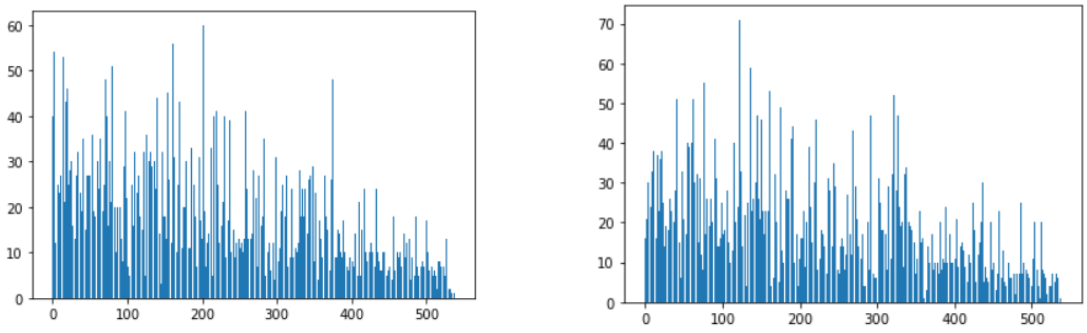

Figure 4.8: Frequencies of the four vertex DAGs decoded from sampled latent point

4.3 Posterior learning of bigger DAGs

As the next step we changed our focus to a bigger scale problem, and wanted to test, if the model was capable of learning BN structures with more nodes. As our dataset we chose the ALARM dataset.

4.3.1 ALARM

The ALARM ("A Logical Alarm Reduction Mechanism") dataset was constructed from expert knowledge to provide an alarm message system for patient monitoring. The domain contains 37 discrete variables taking between 2 and 4 values.

4.3.2 Conversion of Bayesian Network

The available ALARM network is stored in a format, called Bayesian Interchange Format (BIF), however bnmcmc expected a different network format. To solve this problem we developed a transformation script which output the necessarily formatted BN.

4.3.2.1 BIF

The BIF is a vehicle for interoperability of belief network tools and aims at facilitating comparison and discussion of research results. The Interchange Format uses only ASCII symbols and expects one stream to contain a single network. The basic unit of information is a block: a piece of text which starts with a keyword and ends with the end of an attribute list. Arbitrary characters are allowed between blocks. This allows the user to insert arbitrarily long comments outside the blocks, and reserve the //, /* */, comments to be placed inside blocks.

Blocks A block is a unit of information. The general format of a block is:

```
block-type block-name {
   attribute-name  attribute-value;
   attribute-name  attribute-value;
   attribute-name  attribute-value;
 }
```

with as many attributes as necessary. The closing semicolon is mandatory after each attribute. There are three possible blocks: network, variable and probability blocks. The blocks occur in the following order: A network declaration block (one, must be first). A series of variable declaration blocks and probability definition blocks, possibly inter-mixed.

Networks A network block defines the name of the network and lists the properties.

```
network Robot-Planning {
   property version 1.1;
   property author Nobody;
}
```

Variable Variable blocks define the variables in a network.

```
variable Leg {
   type discrete[2] { long, short };
   property temporary yes;
}
```

Probability Probability blocks specify the (conditional) probability tables (CPTs) for these variables, and hence the topology of the network. The block indicates the variables of the probability distribution right after the keyword probability.

```
probability ( Leg | Arm ) {
   table 0.1 0.9 0.9 0.1;
}
```

4.3.2.2 BNMCMC format

BNMCMC expected an xml formatted input. To generate this input we transformed the bif file into another BN repsentation, that we fed into a program called BNF, that converted it into XML format. The transformation script built a BN representation with the following structure:

1. The first row of the file contained the number of nodes in the network
2. The subsequent rows were structured in the following way:
   - Node id
   - Node name
   - List of valid values, delimitered with ","
   - Other options
   - List of parents
   - Flattened list of the CPT rows, delimitered with ","

   The different attributes were seperated by a semicolon (;).

4.3.2.3 Transformation

To transform the bif format we developed the following algorithm. The method first extracts the blocks from the BIF file. Then, depending on the type of block, either the possible values or the parents and CPT were saved in an object representing a node in the network. Once the conversion was complete, the output file was fed into the BNF program, which converted it into an XML file that could be fed into bnmcmc.

```
   Input: BifFile - file containing the BN in bif format
   Output: TranformedFile - file containing the BN in bnmcmc format
 1 bnNodes - list of objects representing vertices in a Bayes Network varIx ← 0
 2 blocks ← extractBlocks(BifFile)
 3 for block in blocks do
 4     type ← blockType(block)
 5     if type is "variable" then
 6         bnNode ← new BnNode
 7         bnNode.values ← extractValues(block)
 8         bnNodes.append(bnNode)
 9     if type is "probability" then
10         bnNode ← bnNodes[varIx]
11         bnNode.parents ← extractParents(block)
12         bnNode.distribution ← extractDistribution(block)
13         varIx ← varIx + 1
14 writeToFile(bnNodes)
```

4.3.3 Running the MCMC model

The available BN was sampled 1000 times, and then these data samples were fed into the bnmcmc program. BNMCMC has many possible configurations but we only used a small subset of them. The applied settings were the following:

- Heat profile: linear
- Burn in: 100
- Parameter prior: bdeu
- vss-bdeu-per-table: 10000

The MCMC method started on a randomized BN structure. We ran the simulation 100000 times each starting from a randomized BN structure and taking 100 steps after burn-in. After each step the resulting structure was logged which we later transformed into a format that could be fed into the neural network.

4.3.4 BNMCMC log processing

During the run bnmcmc logs and saves the visited structures into the specified format. Currently there exist 3 available formats:

- adj: the graph structure is logged as an adjacency matrix
- plist: the struture is represented as $child < parent1 : parent2 : ... >$
- diff_txt: the structure alteration is logged as +/-parent,child, where + represents the addition and - represents the deletion of an edge

For the sake of compactness we used the latter format as it requires significantly less storage than the other two. As the neural network required the input to be in the form of iGraph structures. We developed a transformation script **process_mcmc** which could transform the logs into iGraph objects. Process mcmc:

1. Create a new directed iGraph object
2. Add nodes to the graph: the number of different variables in the original BN +2 for start and end nodes
3. Set node types: the NN can only deal with numerical node types, so each original nodetype is mapped to its index in the graph
4. Add edges:
   - 4.1 Split each line on the operation characters (+ or -)
   - 4.2 Add/remove edge based on operation
   - 4.3 Filter non-acyclic graphs
   - 4.4 Sort nodes by topological order
   - 4.5 Connect nodes without predecessors to START node
   - 4.6 Connect nodes without successors to END node

4.7 Save graph in binary format

As this the process would have taken a significant amount of time, we used paralellization to reduce the runtime.

4.3.5 Results

Despite the vast size of the dataset, we could only use 2000 samples during the training process due to computational limitations. Another problem we have encountered is that the training process is incredibly slow with this graph size. The cause of this problem was that during decoding each generated node was "compared" with all the existing nodes on whether to add a connection between them and then the current node's state was updated. This process takes about $\mathcal{O}(n^2)$ and as n gets higher this significantly slows down the learning process.

For this reason, we could not use DAVE_BN with this dataset, instead, we utilized the encoder of DVAE combined with the decoder of another model called SVAE [2]. SVAEs decoder generates the adjacency matrix in a one-shot manner, therefore the decoding process is notably faster. We ran the training with a latent space size of 64 and 1024 GRU size and $10^{-}4$ learning rate. Similar to previous experiments we tried using batch normalization to drive down training losses. Due to the computational limits, we only ran the training for 500 epochs. It is clearly visible that after the 200 epoch the model stops learning, and the training loss no longer decreases. Currently, we could not achieve better results with this model, but we will continue our investigation with other hyperparameters and model parameter combinations.

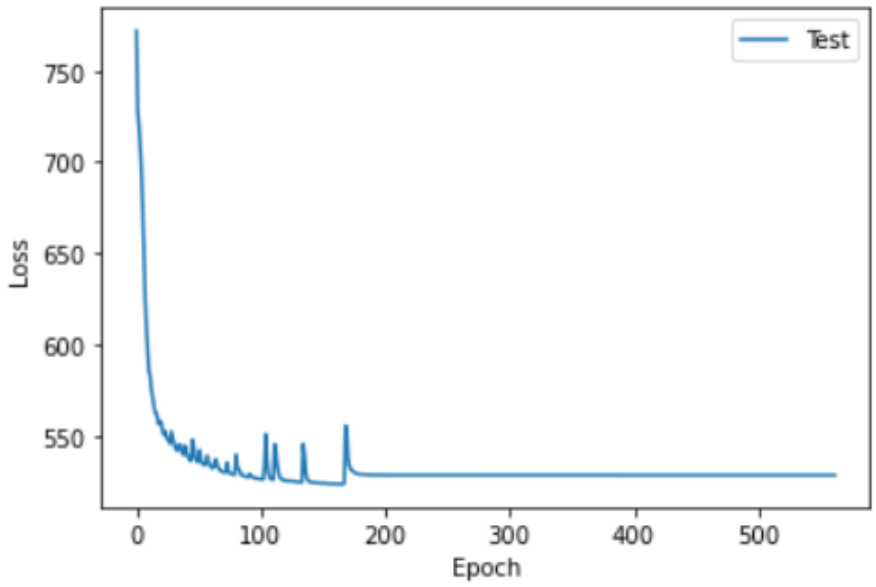

Figure 4.9: Training loss on the ALARM dataset

4.3.6 Future work

The existing solutions for slowness of the decoding process are to generate the nodes or edges in a one-shot way, but that would ignore the node state changes after a connection has been made. We have two proposals for future works to address this problem:

- Batched node generation The nodes of DAGs can be divided into "levels" such that each node in a partition must-have its predecessors from the preceding levels. This partitioning creates subsets of nodes that are not related to each other. If instead of generating nodes one by one, we could generate them partition by partition, it would reduce the decoding time, since we could avoid checking possible connections inside the partitions. The time gained depends to a large extent on the structure of the BN, since the more elements there are in the partitions, the fewer possible connections would need to be checked. In the worst case, each generated batch would contain only 1 node, leading to the original asynchronous message passing.
- Batched edge generation Another option to reduce decoding time would be to generate several edges simultaneously. This could be achieved by encoding the edge probabilities of the newly generated nodes in the current graph state instead of each node. Hence, the adjacency matrix would be generated row by row (or column by column). Even though this would mean that the graph state would be updated only after all the connections to the new node have been established, it could significantly reduce the generation time. Trivially, this method would require the graph state to encode much more complex information about the graph.

Bibliography

[1] A. K. M. Azad, Salem A. Alyami, and Jonathan M. Keith. BNMCMC: a software for inferring and visualizing bayesian networks using MCMC methods. URL https://www.biorxiv.org/content/10.1101/414953v1.

[2] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. URL http://arxiv.org/abs/1511.06349.

[3] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM)*, Proceedings, pages 442–446. Society for Industrial and Applied Mathematics. ISBN 9780898715682. DOI: 10.1137/1.9781611972740.43. URL https://epubs.siam.org/doi/10.1137/1.9781611972740.43.

[4] Serena H. Chen and Carmel A. Pollino. Good practice in bayesian network modelling. *Environmental Modelling Software*, 37:134–145, 2012. ISSN 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2012.03.012. URL https://www.sciencedirect.com/science/article/pii/S1364815212001041.

[5] Hsin-Pai Cheng, Tunhou Zhang, Yixing Zhang, Shiyu Li, Feng Liang, Feng Yan, Meng Li, Vikas Chandra, Hai Li, and Yiran Chen. NASGEM: Neural architecture search via graph embedding method. 35(8):7090–7098. ISSN 2374-3468. DOI: 10.1609/aaai.v35i8.16872. URL https://ojs.aaai.org/index.php/AAAI/article/view/16872.

[6] Jesper Dall and Michael Christensen. Random geometric graphs. 66(1):016121. DOI: 10.1103/PhysRevE.66.016121. URL https://link.aps.org/doi/10.1103/PhysRevE.66.016121.

[7] Tomasz Danel, Przemysław Spurek, Jacek Tabor, Marek Śmieja, Łukasz Struski, Agnieszka Słowik, and Łukasz Maziarka. Spatial graph convolutional networks. URL http://arxiv.org/abs/1909.05310.

[8] Nicola De Cao and Thomas Kipf. MolGAN: An implicit generative model for small molecular graphs. URL http://arxiv.org/abs/1805.11973.

[9] Mikhail Drobyshevskiy and Denis Turdakov. Random graph modeling: A survey of the concepts. 52(6):131:1–131:36. ISSN 0360-0300. DOI: 10.1145/3369782. URL https://doi.org/10.1145/3369782.

[10] Mikhail Drobyshevskiy, Anton Korshunov, and Denis Turdakov. Learning and scaling directed networks via graph embedding. In Michelangelo Ceci, Jaakko Hollmén, Ljupčo Todorovski, Celine Vens, and Sašo Džeroski, editors, *Machine*

Learning and Knowledge Discovery in Databases, Lecture Notes in Computer Science, pages 634–650. Springer International Publishing. ISBN 9783319712499. DOI: 10.1007/978-3-319-71249-9_38.

[11] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. URL http://arxiv.org/abs/1808.05377.

[12] Dani Gamerman and Hedibert F. Lopes. *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference, Second Edition*. CRC Press. ISBN 978-1-58488-587-0. Google-Books-ID: yPvECi_L3bwC.

[13] W. R. Gilks, S. Richardson, and David Spiegelhalter. *Markov Chain Monte Carlo in Practice*. CRC Press. ISBN 978-1-4822-1497-0. Google-Books-ID: T2G1DwAAQBAJ.

[14] Christophe Gonzales, Axel Journe, and Ahmed Mabrouk. Constraint-based bayesian network structure learning using uncertain experts' knowledge. 34. ISSN 2334-0762. DOI: 10.32473/flairs.v34i1.128453. URL https://journals.flvc.org/FLAIRS/article/view/128453.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc. URL https://papers.nips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html.

[16] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942. ISSN: 2161-4407.

[17] Alexander Gutfraind, Lauren Ancel Meyers, and Ilya Safro. Multiscale network generation. URL http://arxiv.org/abs/1207.4266.

[18] Jaehyeong Jo, Seul Lee, and Sung Ju Hwang. Score-based generative modeling of graphs via the system of stochastic differential equations. URL http://arxiv.org/abs/2202.02514.

[19] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. URL http://arxiv.org/abs/1312.6114.

[20] E. A. Leicht and M. E. J. Newman. Community structure in directed networks. 100 (11):118703. ISSN 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.100.118703. URL http://arxiv.org/abs/0709.4500.

[21] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. URL http://arxiv.org/abs/1803.03324.

[22] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. URL http://arxiv.org/abs/2102.12092.

[23] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods [electronic resource]*. New York, NY : Springer New York. ISBN 978-1-4757-4145-2. URL http://archive.org/details/springer_10.1007-978-1-4757-4145-2.

[24] R.W. Robinson. Counting labeled acyclic digraphs. *New Directions in the Theory of Graphs*, page 239 – 273, 1973. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-0001457227&partnerID=40&md5=c7f53d8d2cdbd630f1a699736111f0c4. Cited by: 155.

[25] Mauro Scanagatta, Antonio Salmerón, and Fabio Stella. A survey on bayesian network structure learning from data. 8(4):425–439. ISSN 2192-6360. DOI: 10.1007/s13748-019-00194-y. URL https://doi.org/10.1007/s13748-019-00194-y.

[26] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. 20(1):61–80. ISSN 1941-0093. DOI: 10.1109/TNN.2008.2005605.

[27] Ioannis Tsamardinos, Laura E. Brown, and Constantin F. Aliferis. The max-min hill-climbing bayesian network structure learning algorithm. 65(1):31–78. ISSN 1573-0565. DOI: 10.1007/s10994-006-6889-7. URL https://doi.org/10.1007/s10994-006-6889-7.

[28] Sun-Chong Wang. *Artificial Neural Network*, pages 81–100. Springer US. ISBN 978-1-4613-5046-0 978-1-4615-0377-4. DOI: 10.1007/978-1-4615-0377-4_5. URL http://link.springer.com/10.1007/978-1-4615-0377-4_5.

[29] Xiyuan Wang and Muhan Zhang. How powerful are spectral graph neural networks. In *Proceedings of the 39th International Conference on Machine Learning*, pages 23341–23362. PMLR. URL https://proceedings.mlr.press/v162/wang22am.html. ISSN: 2640-3498.

[30] Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei, and Bo Long. Graph neural networks for natural language processing: A survey, . URL http://arxiv.org/abs/2106.06090.

[31] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. 32(1):4–24, . ISSN 2162-2388. DOI: 10.1109/TNNLS.2020.2978386. Conference Name: IEEE Transactions on Neural Networks and Learning Systems.

[32] Xiaowei Ying and Xintao Wu. Graph generation with prescribed feature constraints. In *Proceedings of the 2009 SIAM International Conference on Data Mining (SDM)*, Proceedings, pages 966–977. Society for Industrial and Applied Mathematics. ISBN 9780898716825. DOI: 10.1137/1.9781611972795.83. URL https://epubs.siam.org/doi/10.1137/1.9781611972795.83.

[33] Jiahui Yu, Yuanzhong Xu, Jing Yu Koh, Thang Luong, Gunjan Baid, Zirui Wang, Vijay Vasudevan, Alexander Ku, Yinfei Yang, Burcu Karagol Ayan, Ben Hutchinson, Wei Han, Zarana Parekh, Xin Li, Han Zhang, Jason Baldridge, and Yonghui Wu. Scaling autoregressive models for content-rich text-to-image generation. URL http://arxiv.org/abs/2206.10789.

[34] Chengxi Zang and Fei Wang. MoFlow: An invertible flow model for generating molecular graphs. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 617–626. DOI: 10.1145/3394486.3403104. URL http://arxiv.org/abs/2006.10137.

[35] Muhan Zhang, Shali Jiang, Zhicheng Cui, Roman Garnett, and Yixin Chen. D-VAE: A variational autoencoder for directed acyclic graphs, . URL http://arxiv.org/abs/1904.11088. type: article.

[36] Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. Graph neural networks and their current applications in bioinformatics. 12, . ISSN 1664-8021. URL https://www.frontiersin.org/articles/10.3389/fgene.2021.690049.

[37] Yanqiao Zhu, Yuanqi Du, Yinkai Wang, Yichen Xu, Jieyu Zhang, Qiang Liu, and Shu Wu. A survey on deep graph generation: Methods and applications. URL http://arxiv.org/abs/2203.06714.