Graph Generation with VGAE and Normalizing Flows

Alinejad Kévin, Moutet Maxime

January 15, 2025

1 Introduction

The goal of this challenge is to train a model to generate graphs. We are provided with the code from [2], which is composed of a variational graph autoencoder and denoising diffusion model. The VGAE aims to learn a latent representation of the graphs, while the denoising model uses this latent representation to reconstruct the adjacency matrix. The denoising model is conditioned by some characteristic features described in a textual file associated with each graph sample.

The performance of the VGAE is monitored by three different metrics. We optimize the reconstruction loss (MAE between the adjacency matrix in our case), and the Kullback-Leibler divergence. For a graph G with adjacency matrix A and node features x, we denote by A_θ and x_θ the adjacency matrix and node representation of the generated graph G_θ. The overall loss is given by:

$$\mathcal{L}(G, G_\theta) = ||A - A_\theta||_1 + \beta D_{KL}(x, x_\theta) \qquad (1)$$

The $\beta \in \mathbb{R}$ parameter realizes a trade off between the reconstruction loss and the Kullback-Leibler divergence.

For the diffusion model part, we only optimize the Huber loss between the real noise and its reconstruction.

Our approach is divided into three different parts. First, we extensively studied the data distributions, and how to extract the most informative features from the files. In particular, we used a sentence encoder to construct an embedding of the textual descriptions and conditioned the denoising model with this additional features. Then, we tried to improve the VGAE structure, by using different encoders/decoders architectures. After, we replaced the generative diffusion model by a matching flows approach. Finally, we refined the training with different schedulers and optimizers.

The GitHub for this project, with our code and results, can be found at: `https://github.com/MoutetMaxime/altegrad-challenge`.

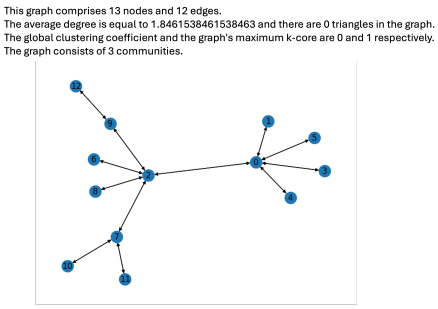
2 Data

The dataset is composed of a train, test, and validation sets. The validation and training tests are composed of two types of data:

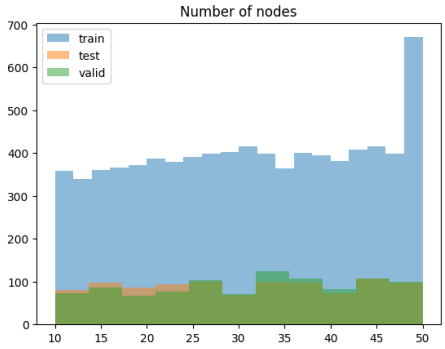
- Graphs: This folder contains both edgelist and graphml formats.
- Descriptions: Each graph file has a textual description describing the number of nodes, edges as well as other characteristics of the graph structure.

The validation set is similar to the training set, but has only 1000 samples compared to 8000 samples for the training set. Finally, the test set is composed of 1000 textual descriptions of the graphs to generate. All the descriptions are very similar and have the same format. Thus, the extraction of the statistics of each graph is made with a simple regex formulae.

We quickly studied the training and test distribution of each statistic described in the textual files. The led us to undersample the training set, because there was a little shift in the distributions of the number of nodes, see Figure 1b.



(a) Sample associated with the textual description.



(b) We can see that there is a very large amount of graphs with a high number of nodes in the training set. The other statistics had very similar distributions between training and test.

Figure 1: Data Exploration

3 Method

We describe here all the different methods we implemented. Some of them did not improve the overall performance of the algorithm, but we detail here everything we tried.

3.1 Encoders

3.1.1 Graph Isomorphism Networks

In the baseline, the encoder used is a Graph Isomorphism Network (GIN) [5]. We briefly describe its behavior. GINs are designed to effectively capture graph structures while ensuring maximum discriminative power between non-isomorphic graphs. Unlike other models, GINs employ a sum-based aggregation scheme that generalizes the Weisfeiler-Lehman graph isomorphism test.

Consider an undirected graph G with N nodes denoted as $[1, N]$, where each node i is associated with an initial representation $h_i^{(0)} \in \mathbb{R}^d$. At each layer l, the node embeddings are updated according to the following equation:

$$h_i^{(l+1)} = \text{MLP}\left((1+\epsilon)\cdot h_i^{(l)} + \sum_{j\in\mathcal{N}_i} h_j^{(l)}\right) \qquad (2)$$

where $\mathcal{N}_i$ denotes the set of neighbors of node i, ϵ is a parameter (either learnable or fixed) that controls the contribution of the node's own representation, and MLP is a multi-layer perceptron used to capture nonlinear interactions.

This sum-based aggregation mechanism ensures that the embeddings are injective under certain conditions, enabling GIN to distinguish between non-isomorphic graphs. By iterating this process over multiple layers, the node representations gradually incorporate structural and semantic information from their neighborhoods.

In order to improve the latent representation of the graph generated by this encoder, we tried to add the information of the textual data to the encoder. We first did it by concatenating the extracted features to the input graph spectral embedding provided when loading the data. To do this in a more elegant way, we implemented a cross attention head and computed the representation of each graph, with the keys and values coming from the description features.

3.1.2 Graph Attention Networks

Graphical Attention Networks (GANs) [4] have been shown to be effective in a number of tasks. Like other graphical neural networks, GATs aggregate information from each node's neighbors in order to calculate its integration. Their main difference lies in the use of an attention mechanism that weights the contributions of each node's neighbors.

Consider an undirected graph G composed of N nodes, denoted $[1, N]$. Let d be the dimension of the node embeddings and $h_1, ..., h_N \in \mathbb{R}^d$ be the corresponding embeddings. Let also $W \in \mathbb{R}^{d' \times d}$ be a transformation matrix and $a \in \mathbb{R}^{2d'}$ a weight vector. The attention weights are defined as follows:

$$e(h_i, h_j) = a^T \text{LeakyReLU}([Wh_i|Wh_j]) \quad (3)$$

where $|$ denotes the concatenation operation. The attention weights are then normalized using the softmax operator:

$$\alpha_{ij} = \frac{\exp(e(h_i, h_j))}{\sum_{k \in \mathcal{N}_i} \exp(e(h_i, h_k))} \quad (4)$$

where $\mathcal{N}_i$ represents the set of neighbours of node i in graph G. This mechanism allows graphs of different sizes to be processed efficiently in batches. The final integration of a node i is then computed as:

$$h'_i = \text{LeakyReLU}\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} W h_j\right) \quad (5)$$

In general, we use multi-head attention, with K heads, $a^{(1)}, \ldots, a^{(K)} \in \mathbb{R}^{2d'/K}$ and $W^{(1)}, \ldots, W^{(K)} \in \mathbb{R}^{d'/K \times d}$:

$$h'_i = \text{LeakyReLU}\left(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(k)} W^{(k)} h_j\right) \quad (6)$$

3.2 Decoder

We did not really dived into the decoder. We kept the same decoder as in the baseline, which is composed of several MLP layers. In the same way as for the encoder, we augmented the latent representation given to the decoder with the features statistics from the textual description. This led to a huge improvement in the reconstruction loss. We did it first by concatenation before passing in the mlp layers. Then we tried to use again cross attention heads to take into account this information in a more subtle way. This unfortunately did not yield better results than concatenation and increased the computation time.

3.3 Generative models

We were particularly interested in the last part of the architecture, which takes the encoding of the VGAE, and trains a denoising model to reconstruct the adjacency matrix. We tried to implement another generative model based on normalizing flows.

3.3.1 Continuous Normalizing flow : reminders

we remind the reader of the following notations:

- $\mathbb{R}^d$ denotes the latent space with latent points $\mathbf{z} = (z^1, \ldots, z^d) \in \mathbb{R}^d$.
- $\mathbf{v}\colon [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$ is the time-dependent vector field.
- $\boldsymbol{\phi}\colon [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$ is a flow. A flow is associated with a vector field $\mathbf{v}_t$ by the following equations:

$$\frac{d}{dt}\,\phi_t(\mathbf{z}) \;=\; \mathbf{v}_t\big(\phi_t(\mathbf{z})\big), \quad \phi_0(\mathbf{z}) \;=\; \mathbf{z}. \quad (7)$$

The main idea is then to use a neural network to model the vector field $\mathbf{v}_t$, which implies that ϕ_t will also be a parametric model; we call it a *Continuous Normalizing Flow (CNF)*. In the context of generative modeling, the CNF allows us to transform a simple prior distribution (*e.g.* Gaussian) p_0 into a more complex one p_1 (which we want close to the data distribution q) by using the push-forward equation:

$$p_t \;=\; [\phi_t]_{\#}\, p_0. \quad (8)$$

Here,

$$\forall\, \mathbf{z} \in \mathbb{R}^d, \quad [\phi_t]_{\#}\, p_0(\mathbf{z}) \;=\; p_0\big(\phi_t^{-1}(\mathbf{z})\big)\, \det\Big[\tfrac{\partial(\phi_t^{-1})}{\partial \mathbf{z}}\Big].$$

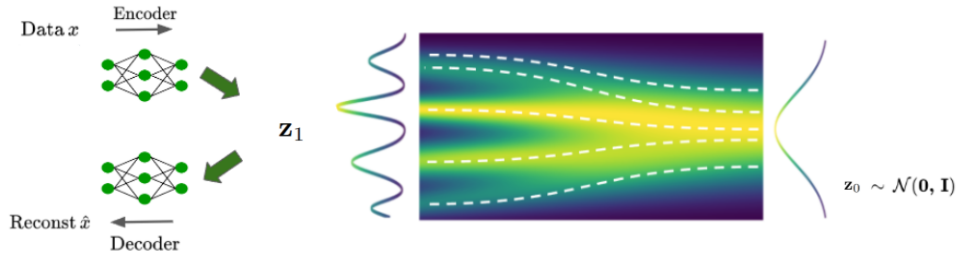


Figure 2: Illustration of a generative model adapted from [1]. The original image is from an image-based generative model, where an encoder maps data x to a latent space (z_0), and a decoder reconstructs the data ($\hat{x}$). In this work, the concept is applied to graph generative modeling with similar principles, differing primarily in the data type.

But this is intractable. Thus, the *Flow Matching* aims at learning the true vector field $\mathbf{u}_t$, via:

$$\mathcal{L}_{\mathrm{FM}}(\theta) \;=\; \mathbb{E}_{t,p_t(\mathbf{z})}\Big[\,\big\|\mathbf{v}_t(\mathbf{z}) - \mathbf{u}_t(\mathbf{z})\big\|^2\Big]. \tag{9}$$

However, this objective is also intractable, so the article [3] introduces the *Conditional Flow Matching* objective, where $\mathbf{z}_1 \sim q(\mathbf{z}_1)$ approximates $p_1(\mathbf{z}_1)$:

$$\mathcal{L}_{\mathrm{CFM}}(\theta) \;=\; \mathbb{E}_{t,q(\mathbf{z}_1),\,p_t(\mathbf{z}|\mathbf{z}_1)}\Big[\,\big\|\mathbf{v}_t(\mathbf{z}) - \mathbf{u}_t\big(\mathbf{z}\,\big|\mathbf{z}_1\big)\big\|^2\Big]. \tag{10}$$

3.3.2 Flow Matching with Optimal Transport Conditional Vector Fields

In the context of the challenge, we consider *Flow Matching* in a latent space of dimension d. Let us denote by $\mathbf{z} \in \mathbb{R}^d$ a latent vector, which in our case comes from encoding a graph $\mathcal{G}$ via a variational autoencoder (VAE). We aim to learn a time-dependent **velocity field**

$$\mathbf{v}_\theta : [0,1] \times \mathbb{R}^d \times \mathbb{R}^m \;\longrightarrow\; \mathbb{R}^d,$$

where $[0,1]$ is the time interval, $\mathbf{z} \in \mathbb{R}^d$ is the latent code, and $\mathbf{c} \in \mathbb{R}^m$ is a *condition* (e.g., graph statistics). Intuitively, we want to continuously transform a simple *base* distribution (like a standard Gaussian) into the latent distribution that we observe from real data.

Following (7) We model the time-evolving state $\boldsymbol{\phi}_t(\mathbf{z}) \in \mathbb{R}^d$ by solving

$$\frac{d}{dt}\,\boldsymbol{\phi}_t(\mathbf{z}) \;=\; \mathbf{v}_\theta\big(t,\,\boldsymbol{\phi}_t(\mathbf{z}),\,\mathbf{c}\big), \quad \boldsymbol{\phi}_0(\mathbf{z}) \;=\; \mathbf{z}.$$

When t goes from 0 to 1, $\boldsymbol{\phi}_t$ pushes a simpler distribution $\mathbf{z}_0 \sim p_0$ toward a more complex distribution p_1, which we desire to match the real latents $\mathbf{z}_1 \sim q(\mathbf{z}_1)$.

Flow Matching Objective. A direct approach would be to match $p_1(\mathbf{z})$ exactly, but computing Jacobians and exact likelihood can be costly. Instead, we exploit the *flow matching* or *trajectory matching* principle [3], which only requires us to *match the velocity* $\mathbf{v}_\theta$ along a chosen reference path. To achieve that, we propose an implementation of the *Optimal Transport conditional Vector Fields*. Please refer to the article [3] for more details about the motivations behind this concept.

Let $\mathbf{z}_1 \sim q(\mathbf{z}_1)$, *i.e.* a point from the dataset, and we can define the *conditional flow* ψ_t (the conditional version of ϕ_t, meaning it corresponds to $\mathbf{u}_t\big(\mathbf{z}\,\big|\mathbf{z}_1\big)$) to be the Optimal Transport *displacement map* between two Gaussians $p_0(\mathbf{z} \mid \mathbf{z}_1)$ and $p_1(\mathbf{z} \mid \mathbf{z}_1)$ (one can prove that ψ_t satisfies an appropriate condition, analogous to eq. (2) above). One practical

choice for ψ_t is a *linear path* between $\mathbf{z}_0$ and $\mathbf{z}_1$, potentially including a small parameter $\sigma_{\min} > 0$ for better stability. We define

$$\psi_t(\mathbf{z}) \;=\; \big[1 - (1 - \sigma_{\min})\, t\big]\, \mathbf{z} \;+\; t\, \mathbf{z}_1. \tag{11}$$

We call this an *optimal transport* line-based path, because when $\sigma_{\min} = 0$, it is a direct linear interpolation from $\mathbf{z}_0$ to $\mathbf{z}_1$. The presence of $\sigma_{\min} > 0$ ensures we never fully lose the base distribution.

Formally, we define a function ψ_t (a *reference trajectory* from base $\mathbf{z}_0$ to target $\mathbf{z}_1$) and require that

$$\mathbf{v}_\theta\big(\psi_t(\mathbf{z}_0),\, t,\, \mathbf{c}\big) \;\approx\; \frac{d}{dt}\, \psi_t(\mathbf{z}_0).$$

In the context of *Conditional Flow Matching* (CFM), we write an expectation over pairs $(\mathbf{z}_1, \mathbf{c})$ and random $(\mathbf{z}_0, t)$. Hence the training loss is

$$\mathcal{L}_{\mathrm{CFM}}(\theta) \;=\; \mathbb{E}_{\substack{\mathbf{z}_1 \sim q \\ \mathbf{z}_0 \sim p_0 \\ t \sim \mathrm{U}[0,1]}} \Big\| \mathbf{v}_\theta\big(\psi_t(\mathbf{z}_0),\, t,\, \mathbf{c}\big) - \tfrac{d}{dt}\, \psi_t(\mathbf{z}_0) \Big\|^2.$$

The derivative is simply

$$\frac{d}{dt}\, \psi_t(\mathbf{z}_0) \;=\; \mathbf{z}_1 \;-\; (1 - \sigma_{\min})\, \mathbf{z}_0.$$

Thus we obtain the final form of the conditional flow matching loss:

$$\mathcal{L}_{\mathrm{CFM}}(\theta) = \mathbb{E}_{\substack{\mathbf{z}_1 \sim q \\ \mathbf{z}_0 \sim p_0 \\ t \sim \mathrm{U}[0,1]}} \Big\| \mathbf{v}_\theta\big(\psi_t(\mathbf{z}_0),\, t,\, \mathbf{c}\big) - \big(\mathbf{z}_1 \;-\; (1 - \sigma_{\min})\, \mathbf{z}_0\big) \Big\|^2.$$

Architecture. To represent the velocity field $\mathbf{v}_\theta(t, \mathbf{z}, \mathbf{c})$, we parameterize it with a deep neural network:

- A small MLP to embed the condition $\mathbf{c}$ (e.g., graph properties) into a feature vector.
- A time-embedding function that maps $t \in [0, 1]$ to a higher-dimensional sinusoidal feature.
- A final MLP that takes the concatenation of $(\mathbf{z}, \text{cond\_feat}, \text{time\_feat})$ and returns a velocity in $\mathbb{R}^d$.

By minimizing the conditional flow matching loss, the network learns to *transport* standard Gaussian samples $\mathbf{z}_0 \sim p_0$ to the real latent distribution $\mathbf{z}_1 \sim q(\mathbf{z}_1)$ in a time-continuous manner.

Inference / Sampling. Once trained, we can sample new points by integrating the ODE

$$\frac{d}{dt}\, \mathbf{z}(t) \;=\; \mathbf{v}_\theta\big(t, \mathbf{z}(t), \mathbf{c}\big), \quad \mathbf{z}(0) \sim p_0.$$

Using an ODE solver from $t = 0$ to $t = 1$, we obtain $\mathbf{z}(1)$ that approximates the distribution of real latents $\mathbf{z}_1$. We can then decode $\mathbf{z}(1)$ with our VAE decoder to obtain new generated graphs.

3.4 Training

As well as trying to improve the model, we also tried to modify the training. In particular, we tested different optimizers (Adam, AdamW, SGD, ...), as well as different types of scheduler to adapt the learning rate as the training progressed. We also tried using a few warm-up epochs with a very low learning rate before launching the scheduler. Finally, we played with the model's hyperparameters (number of encoder/decoder layers, size of the latent space, ...) to try and optimise our result.

4 Results

We achieved a score on the test set of 0.149. In Table 1, we detail some of our major advancements on the score, and which architecture was used for this. Finally, the best model was the flow matching approach, and a conditional decoder. Our other paths of exploration did not improved a lot the overall result but increased the computation time thus, at the end, we did not used most of it.

Model	MAE on test set	Description
Baseline	0.886	Baseline provided.
Prompt Encoder	0.897	Utilizes a sentence encoder to embed the textual description.
Undersample	0.877	Undersampled large graph to shift the training distribution.
Conditional decoder	0.172	Conditioned the decoder with features description.
OT CFM	0.149	Flow Matching approach and Conditional decoder.

Table 1: Performance metrics and descriptions for different models.

References

[1] Quan Dao, Hao Phung, Binh Nguyen, and Anh Tran. Flow matching in latent space. *arXiv preprint*, arXiv:2307.08698, 2023.

[2] Iakovos Evdaimon, Giannis Nikolentzos, Christos Xypolopoulos, Ahmed Kammoun, Michail Chatzianastasis, Hadi Abdine, and Michalis Vazirgiannis. Neural graph generator: Feature-conditioned graph generation using latent diffusion models, 2024.

[3] Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. *arXiv preprint*, arXiv:2210.02747, 2022.

[4] Amin Salehi and Hasan Davulcu. Graph attention auto-encoders, 2019.

[5] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.