ALTEGRAD 2024 - Data Challenge Report
CGM Team

Channdeth Sok
M2DS
channdeth.sok@gmail.com

Gleb Goncharov
M2DS
gigoncharov@outlook.com

Marceau Leclerc
M2DS
marceau.leclerc@outlook.fr

Abstract

This report is part of the work our team produced in the framework of the ALTEGRAD 2024 Data Challenge. We will present most of our experiments that ended up being interesting, either because their implementation was insightful in some way or because the results they gave at the task at hand are interesting to report. Our goal in this document is to explain our reasoning and how we advanced through this project. Overall, we achieved 0.112 with a generative approach and 0.099 with a simple MLP model. The project's Github repository can be found here.

1. Introduction

The task we aim to tackle is a down-scaled version of the one presented in [2], to which end was introduced the architecture of the Neural Graph Generator (NGG). The goal is to generate graphs presenting the key features (e.g. the number of nodes, average degree, etc.) prompted by the user. In the code we were provided, these features indeed were given in a prompt format and were exactly 7, unlike in [2] where there are up to 15 features. To accomplish this, the provided baseline was a latent-diffusion model, an architecture that combines a variational autoencoder (VAE) that can be trained to reconstruct graphs while producing a latent space in the bottleneck, and a diffusion model (DM) which can be trained to sample realistic latent space vectors using the conditioning, i.e. the features we want to recover.

In this report we will present the data we were handed and how we came to use it like we did, as well as our evaluation metrics. From there we will go through the different models, tweaks and strategies we used to try and better accomplish the task, as well as report their respective performances. Finally we will discuss our results and the lessons they yielded.

2. Related Work

We will here briefly present the sources of our inspirations when navigating this project. We already introduced the framework of NGG [2].

Contrastive Learning. To try and improve upon NGG, we experimented with a contrastive training objective. CLIP [6] was introduced to align visual and textual spaces with great success. Following it, GraphCLIP [7] was introduced to align the "graph modality" with the textual one. We decided to base our contrastive experiments on this philosophy, adapting the method to our (less complex) task where it was relevant, i.e. when training NGG's VAE.

Language Models. The data features provided were in the form of prompts, it then seemed natural to try and encode this language data using LLMs, specifically trying BERT [1] that has proven to produce sentence embeddings of high quality. Still on the topic of LLMs, we had the idea to use a finetuned LLM to directly predict the graphs' edgelists out of the prompts. To operate such finetuning in practice with hardware available to us, we chose to train a LoRA [3] for the memory-efficiency of this method.

Graph embedding. We experimented with a different method of graph embedding than the one (spectral) provided. Deepwalk [5] is a method that embeds graphs by running a fixed number of fixed-length random walks on all the nodes. This per-node representation is then mapped to a learned embedding space using a method such as [4].

Generative Approaches. Also, since our task is a generative one, we studied and discussed different generative approaches like GANS, VAE, diffusion models, and Bayesian Networks. However, we didn't have enough time (and initiatives as well) to try and implement all of them.

3. Approach and results

We make the choice of presenting the methods we used along with their results. In our opinion, this clarifies our exploratory approach and helps to navigate this report. To that end, we will start by introducing the metric we came up with to evaluate our progress and models along with the

baseline we had, and then go through our experiments, starting with supervised learning approaches and following with the generative ones.

3.1. Preliminaries

3.1.1 Evaluation metric

We will further discuss the original training metrics when elaborating on NGG in Section 3.1.3. To enable us to compare our models without completely relying on Kaggle submissions, we came up with an evaluation script that can score a sequence of graph edglelists (following the structure of the target output to submit). To that end we simply took the description of the evaluation in the provided instruction sheet. The submissions were described as being scored according to a mean average error (MAE) directly on the features of the output graphs, compared to the desired features inputted. We simply reproduced that with the obvious `NetworkX` Python library utilitaries. The only choice we made was regarding the number of communities feature, for which the computing algorithm incorporates stochasticity. To try and circumvent the noise induced as a consequence without adding too much computing time, we ran 10 Louvain community computations and averaged their outputs. We also reversed-engineered the fact that the features were standard-scaled in order not to bias the score towards the features that inherently have high numerical values, *e.g*. the number of triangles or number of edges.
The method described above was implemented in a python script we could run either on demand or at the end of a training procedure when generating the output graph. It was not used as a validation metric inside a training procedure. We abused of the test set in that way, even though it is the opposite of rigorous practice. We motivate that choice by the fact that this project is part of a Kaggle competition, and we did want to score as high as possible. This evaluation script was somewhat reliable, in particular of high MAE values, but did lack precision when obtaining better scores, all in all limiting our ability to mindlessly overfit using it. We did notice at a late stage of the project that a very similar metric was handed in the Github associated to [2]. We did not use it at all and entirely complied with the "no external code" rule of the project.

3.1.2 Handed data

The data provided was formatted as follows. We had a given prompt that described the graph to predict expected features. These were the number of nodes, edges, average degree, number of triangles, global clustering coefficient, maximum k-core and number of communities. These 7 features were always present and always in this exact order, sprinkled in one of two prompt templates. Apart for the test set, these prompts were paired with corresponding graphs. The train/validation/test split was 8000/1000/1000. When running our custom experiments, we chose to standard-scale the features in order to ensure consistency in our approach.

3.1.3 NGG baseline

The handed NGG [2] architecture is a latent diffusion model adapted for graphs which is comprised of a VAE that encodes a graph's embedding to a latent space, and decodes the latent representation to graph adjacency matrix, that aims to recover a graph with corresponding features. To make use of this latent space, a DM denoises a standard vector conditionally to the features after these are embedded via a simple MLP. The general base training workflow goes as follows:

1. The features are extracted directly using the consistent prompt format.

2. The graphs go through a spectral embedding to retrieve the 10 most important spectral features.

3. The VAE is trained first using two objectives:

   - A reconstruction objective on the decoded adjacency matrix.
   - A transport objective in the form of the minimization of a Kullback-Leibler divergence in the latent space.

   These losses are summed with a weighting of 0.05 to the KLD loss to define the VAE's loss.

4. The DM and feature MLP are then trained simultaneously to recover the associated latent representation using a simple L1 loss.

5. The then trained model can be used to sample a latent representation of the requested features that is then decoded to predict an associated graph's adjacency matrix, that can be used to recover an edgelist.

This original model typically produces - after 200 and 100 epochs for the VAE and DM respectively and no early stopping procedure, which is enough for the models' losses to converge on the validation set - outputs that score an MAE of around **0.88621**, translating to a score of around 0.89554 with our evaluation metric. This was our provided baseline.

3.2. Non-generative approaches

We will here go through our experiments that did not involve generative models. These were not the core of our work, and used either as baseline or interesting follow-up.

3.2.1 1-NN

Since the problem is part of a Kaggle challenge and we wanted to score quite well, we started with a very simple but intuitive idea just to see how well it scored. What can be better for predicting a desired graph than it's closest neighbor from the training set in the feature space ? We made it out first baseline, a 1-NN model. It showed promising results with an MAE of 0.172 on the Kaggle leaderboard.

3.2.2 Supervised approaches

Given the previous experiment as well as later results in Section 3.4.1, we came to think a supervised approach such as an MLP directly mapping the features to an adjacency matrix would score a low MAE.
Naturally we started looking for the best architecture of such simple MLP. We experimented with various depths and widths with multiple runs. A depth of 5 hidden layers of growing widths (16, 64, 256, 1024, 2048) consistently produced the best results, with MAEs of 0.21794 (ours) and 0.12541 (Kaggle). Even though this was slightly worse than our best conditional (see Section 3.4.1), it still seemed worth trying to improve it. To that end, we tried two varying architecture tricks to try and better the score.
1st to 2nd to last layer skip connection. Skip connections have proven to better some deep learning architectures (typically on complex computer vision and NLP tasks). We still wanted to try it in our MLP as it wasn't too hard to implement. We connected the output of the first layer of our best MLP architecture to its 2nd-to-last with a simple concatenation. The results obtained simply were on par with the straight feedforward architecture.
Using 2 MLPs. Another trick that seemed promising was splitting the work between 2 MLPs (same architectures as the best MLP above) according to their number of nodes and a fixed threshold, in order to specialize a network on "smaller" graphs and the other on "bigger graphs". This could be implemented with a simple mask. We experimented with a handful of thresholds, and it somewhat expectedly turned out that the best was around 30 nodes i.e. the average number of nodes in the dataset. This also yielded the best MAEs so far : 0.21655 (ours) translating to 0.11602 (Kaggle).
We chose to run several training procedures of that last architecture (mix of MLPs) to try and lower the metric as much as possible. We observed that using an early stopping procedure on the reconstruction loss prevented the model to learn its best, i.e. overfitting this metric did not translate in a worse MAE. We of course reran our other performant experiments (simple MLP, conditional VAE 3.4.1) to ensure that this wasn't the sole reason they did not reach such a low MAE. Getting rid of that early stopping procedure opened the gate to simply merging the training and validation dataset, saving a prediction csv on the test set every 500 epochs. From there we evaluated each of these csv files with our MAE script. The best was a model trained as so over 3000 epochs, reaching MAEs of 0.20246 (ours) and **0.099014** (Kaggle). This ended up being our best result.
We did try a similar architecture but by splitting according to the mean number of edges instead of nodes, with on-par results.
It is hard to interpret this result. It may have been that the mix of 2 MLPs according to the graph size indeed was the better procedure, taking advantage of 2 specialized networks. However, another reasonable explanation could very well be that having models trained on limited data (4500 examples each) induced some noise we took advantage of when running and trying several procedures. At the end of the day, it still was our best score and we did not beat it using the same metric-abusing technique on the conditional VAE with and without denoiser (see Section 3.4.3 on the denoiser ablation to make sense of it) or the simple feedforward MLP.

3.3. Generative approaches

3.3.1 Base NGG tuning and data synthesis

We did try varying the depths and widths of the various bricks of NGG to try and see if we could improve the MAE. These experiments were unfruitful. It does seem indeed that the default hyperparameters of the handed NGG model were close the optimal one for the base set up (including parameters and datasets). However we did get a somewhat slight improvement when generating new data.
Data augmentation. We did want to generate more data from the get go in order to see if it would allow NGG to learn further, as well as to have them for the other architectures we would try. Upon exploring the data and reading [2] in details we figured it wouldn't be efficient to simply reverse engineer the exact synthesis process, there being more than 15 graph generation algorithms. To circumvent that difficulty, we chose to simply slightly modify the graphs in the original training and validation sets. For each graph G in those sets, we chose to generate 6 new graphs as follows. Define m to be an integer sampled uniformly at random between 1 and $\lfloor 0.20 * n \rfloor$ where n is the number of nodes in G.

- 2 were generated using $G's$ adjacency matrix A by simply "flipping" values at random, *e.g.* a 0 would become a 1 and conversely. This was of course done symmetrically m times.
- 2 were generated deleting m nodes in G, and their corresponding edges.
- 2 were generating adding m nodes in G, and connecting them uniformly at random to p other existing nodes

in G, sampling p as the degree of an existing node in G selected uniformly at random as well. This way, added nodes are connected to other nodes in a way that isn't dramatically different as the other existing nodes.

After these graphs were generated, we could pair them with their prompts by simply computing their 7 features of interest and placing them in one of the 2 prompt templates that were present in the original dataset.
Running NGG on this enhanced dataset (the combination of both the original dataset and the generated one) led us to our first noticeable improvement, yielding a MAEs of **0.84613** (Kaggle) and **0.86101** (our script). We still needed to make drastic changes to better these scores.

3.4. Conditioning through the whole pipeline

In the provided baseline the conditioning was implemented only in the latent diffusion part. Theoretically it is possible to add it also to the encoder and decoder parts. We implemented a first version with conditioning just in the encoder, which did not improve the base NGG model.

3.4.1 Conditioning NGG's encoder and decoder

We then conditioned the whole VAE in NGG, incorporating feature conditioning both in the encoder an decoder via simple MLPs. This architecture can be visualized in Fig. 1.

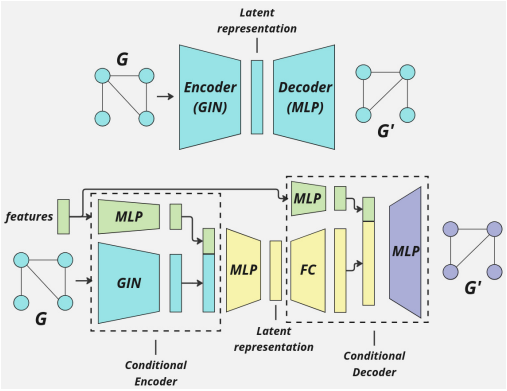

Figure 1. Base NGG (above) and Conditional VAE architecture (below)

Surprisingly for us it lead to remarkable results, our best generative for generative approaches with **0.11173** on Kaggle. To reach this MAE we did thoroughly tune the architecture. We proceeded in the following way : we picked a hyperparameter *e.g.* the hidden size of the decoder's layers, found the best one (using each time an early stopping criterium), and moved on to another hyperparameter, and so on and so forth. The best architecture is presented in Table 1. We also reported another very interesting result that we will make sense of.

Hyperparameter	Best CondVAE	Small Latent Dim
Encoder hid dim	32	32
Decoder hid dim	768	768
Conditioning hid dim	48	48
Latent dim	64	2
# layers encoder	4	4
# layers decoder	4	4
MAE (ours)	0.21641	0.24495

Table 1. Conditional VAE architecture tuning

One thing that has to be noted is that if this "best" architecture produced the lowest MAE, it is likely due to luck. Indeed when trying out architectures in a "sequential search" fashion, the MAE wouldn't vary much, only improved when increasing the hidden size of the feature MLPs and that of the decoder, suggesting these two bricks were solely responsible for the score being better. This was confirmed when setting the latent dimension to the ridiculously low value of 2 and obtaining a result that was still competitive. Note that when our MAE was of values between 0.22 and 0.25, the Kaggle MAE was around 0.12 to 0.14. The best model we produced attained an MAE of **0.11173**. It is important to note that this score was obtained without the use of the denoiser, simply drawing a normal vector at random in the latent space and feeding it to the decoder. We make sense of this choice further down in Section 3.4.3 about the ablation of the denoiser.
The lesson from this tuning was rather clear : the features alone may enable us to predict good enough graphs. This lead to experimenting with MLPs, see Section 3.2.2.

3.4.2 Style transfer approach baseline

Impressed by significant improvements using only a conditional VAE model we decided to enhance it in an interesting way. We tried to combine KNN and VAE. The intuition behind the approach was quite simple. We knew that 1-NN is capable to produce quite satisfying results but this is not a generative approach. The only thing it lacks is an ability to generate slightly modified graphs that are somehow close to the input one. So, we tried to find the closest neighbor of a graph and encode it with our best conditional VAE encoder. Then, given the latent representation, decode a new graph conditionally on the target graph features. Unfortunately, this did not enhance the results of conditional VAE greatly.

3.4.3 Ablation of the denoiser

The previous experiments were enhanced when removing the denoiser. We indeed tried feeding the latent vector drawn at random directly to our decoder, and it turned out

to improve our MAEs.
This was a surprise at first, but we needed to investigate it further. We tried various denoiser architectures. We tried removing the MLP used to embed the features, noticing to significant deterioration. We tried varying the depth, width of the DM as well as the number of timesteps. No architecture distinguished itself whatsoever indicating this part was outright useless. In fact, not using the denoiser yielded slightly better results, but in the order of 0.01 to 0.1 MAE at most.
Again this was a little mind-boggling, and further results (Section 3.2.2) would help make sense of that observation. For what follows, we will indicate wether a denoiser was trained and used, but bear in mind it is not a crucial part.

3.5. Contrastive learning objective in NGG

We tried improving NGG's latent space by incorporating a contrastive objective instead of the KLD loss. Inspiring ourselves from [6] and [7], we chose to try and make the graph encoder (the GIN) in NGG learn a latent space that minimizes the contrastive loss $\mathcal{L}_c$ defined below. To that end, we left the VAE untouched but incorporated an MLP to embed the features of a given graph G. A sketch of the workflow is provided in Fig. 2.

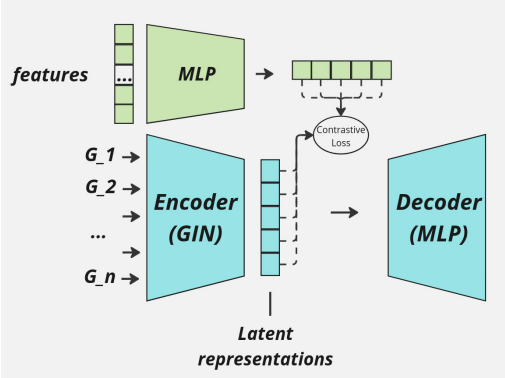

Figure 2. Addition of a feature MLP to NGG to get a contrastive loss.

Contrastive loss. Given a batch of n graphs $\{G_1, ..., G_n\}$ and its label vector $l = \{1, ..., n\}$:

1. Embed the graph and feature batches in their respective encoders to get their representations E_G and E_f of sizes $n \times d$ and $n \times d$ (resp.).

2. Compute $S = \frac{1}{\alpha}(E_G E_f^T)$the similarity matrix of size $n \times n$. Here α is a temperature parameter fixed at 0.07.

3. Compute the loss, CE denoting the cross-entropy function:

$$\mathcal{L}_c = \frac{1}{2}(\text{CE}(S, l) + \text{CE}(S^T, l)) \quad (1)$$

Now we can train this CLIP-ized NGG in two different ways.

- We can either do it sequentially, first minimizing this contrastive loss and training both encoders, leaving the decoder untouched, before then freezing the graph encoder to produce latent representations from which we can train the decoder minimizing the original reconstruction loss. The upside of such sequential training is that we do indeed force the latent space to encode in some way information about the features, from which the decoder can then learn to reconstruct.

- The alternative way is to train this modified NGG "jointly", meaning that all the bricks are trained at once minimizing a loss that is the sum of the reconstruction and contrastive losses. For the presented results below we don't weigh these losses and add them as is. A few explorations showed adding a weighting only impacted the first training steps. The upside of this so-called joint training is that the decoder can prevent the latent space from being too particular about the contrastive task.

We tried both these procedures on the base NGG (with an added feature encoder such as in Fig. 2) as well as with the conditional VAE discussed in Section 3.4.1 that had a feature encoder as a core feature. The results in Table 2 were obtained using batches of size 1024 for the whole chain when learning jointly and 1024 for the contrastive task while 256 for the reconstruction task in the sequential workflow. In both cases, the original dataset was used. The denoiser was trained afterwards and used to sample new data, slightly worsening the results as dicussed in Section 3.4.3.

Procedure	Model	MAE (Ours)	MAE (Kaggle)
Sequential	NGG	0.79587	0.80624
Joint	NGG	0.80754	0.79519
Sequential	C. VAE	N/A	0.80418
Joint	C. VAE	N/A	0.77733

Table 2. MAEs of NGG and the conditionnal VAE integrating the contrastive procedures

Both workflows ended up yielding similar results. These were a slight improvement from the base NGG, but did not perform nearly as well as the conditional VAE presented in the Section 3.4.1. In fact, for this model when incorporating this contrastive learning approach, the results were much worse than when training by minimizing the base KLD and reconstruction losses. We did not continue in the direction of contrastive learning.
Small remark, we did try to further use and freeze the contrastively learnt feature encoder in the denoiser instead of

having it train its own MLP. A few runs showed it did not improve it, confirming further the results in Section 3.4.3.

3.6. Other experiments

3.6.1 Graph embedding with Deepwalk

We wanted to experiment with a different embedding than the spectral one to see if it would improve the latent representation. To that end we used the Deepwalk [5] method. To find a good compromise between a fine representation and a fast obtention, we chose to run 10 walks of size 12 per node. The generated embeddings were set to be of dimension 10, in order to fairly compare them with the spectral embeddings. From there we could run the base NGG workflow described in 3.1.3 that yielded an MAE (ours) of **0.89538**. As this was not a significant improvement, we chose to stick with a spectral embedding of the graphs for all the other approaches.

3.6.2 Prompt embedding

Similarly, we wanted to try embedding the features in a more complex space than the base 7-dimension representation. To that end we tried directly embedding the prompts with BERT [1] using a simple summing aggregation to get a 768-dimensional representation. This led to an a MAE (ours) of **0.89470**. To try and better it, we also concatenated this vector with the 7 features themselves, leading to score similar to the ones without any prompt embedding. As a consequence, we did not chose to embed the prompts in that way and kept going with the 7 features only.

3.6.3 Finetuning LLMs

An exotic idea that seemed relevant for us to give it a chance was to just finetune (using a LoRA [3]) a language model to directly spit out a graph's edgelist from a prompt. To that end a first attempt was done in the following setting. First we generated short very standard prompts (in order not to waste context on it) as well as the expected output in the form of the corresponding graph's edgelist. A LoRA of rank 16 was trained along Microsoft's Phi3 Mini Instruct quantized in 4 bits. This allowed training of said adapter on a single 8GB laptop GPU with batch sizes of 1. We chose for this attempt to sacrifice all the efficiency we could to maximize the yielding model's potential.
Training was ran in under 8 hours. However the results were very disappointing. We did not expect them to be perfect - as this version of Phi3 had a maximum context length of 4096 tokens, which was not enough to cover our prompt-answer pairs that could go up to 6000 tokens for the most heavily connected graphs. But in fact this attempt yielded nonsensical results that did not comply with the template we trained it on.

We still want to try this idea thanks to the Unsloth library, which allows us to optimize and fine-tune the Llama 3 8B model in under 3 hours while maintaining a reasonable level of model output. The context length of Llama 3 8B is 8000 tokens, which might be sufficient for our needs. We managed to fine-tune this model on a single 16GB Nvidia Tesla T4 GPU.
To achieve this, we utilized the Unsloth library, which significantly speeds up the fine-tuning process and reduces memory usage. Unsloth makes Llama 3.1 (8B) fine-tuning 2 times faster and uses 60% less memory. This library allowed us to finish the fine-tuning process quickly and effectively on the available hardware. We were initially pleased to see the output we wanted. However, after running the model on a test dataset we were disappointed with some results. It could not generate complete the full edge lists and sometimes duplicated edges, which is why we could not evaluate the results with the desired metric.

4. Discussion

Even though we did get competitive results with our MLP implementation, it was somewhat disappointing. We wanted a more intricate generative method to do better, but we just could not make it happen. To take a step back, we need to underline the fact that we had access to a small training-validation set of 9000 examples, where NGG [2] was trained on 1M such examples. The fact that the best NGG was improved by training it on generated slightly-off distribution graphs does corroborate this theory. It also has to be acknowledged that our best results were systematically found taking advantage of the best random state. This does not change the general outline of our results, but it has to be said that results were very close for our best models. All in all, one of the many lessons we drew from this project was to never forget the simple things, provided here we were working in a framework with various limitations, especially regarding the size of the dataset we had access to.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. 1, 6

[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. 1, 2, 3, 6

[3] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. 1, 6

[4] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. 1

[5] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '14, page 701–710. ACM, Aug. 2014. 1, 6

[6] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021. 1, 5

[7] Yun Zhu, Haizhou Shi, Xiaotang Wang, Yongchao Liu, Yaoke Wang, Boci Peng, Chuntao Hong, and Siliang Tang. Graph-clip: Enhancing transferability in graph foundation models for text-attributed graphs, 2024. 1, 5