ALTEGRAD Report

Antoine Vialle **antoine.vialle@student-cs.fr**
Gabrielle Caillaud **gabrielle.clld@gmail.com**
Paul Tabbara **paul.tabbara@student-cs.fr**

kaggle : Les Trois Mousquetaires

1 Introduction

Graphs are a powerful tool for representing relationships and structures across diverse domains, from social networks to molecular biology. Deep learning has revolutionized graph generation, enabling the creation of graphs that capture complex patterns and structures. These advances have unlocked new applications in areas like drug discovery, network analysis, and transportation systems.
Conditional graph generation takes this a step further by focusing on generating graphs that satisfy specific constraints, such as connectivity patterns, degree distributions, or domain-specific properties. However, generating graphs under multiple simultaneous constraints is a significant challenge. Traditional approaches, like the Erdős-Rényi model, work well for simple constraints like number of nodes or edges but fail to handle more complex constraints.
This report outlines our exploration of deep learning techniques to address the challenge of generating graphs with specific desired properties.

2 Background

Machine learning and deep learning techniques have been developed and adapted to handle the complexities of graph-structured data [1, 2]. One particularly challenging and fascinating task in this domain is graph generation, which involves creating graphs that satisfy specific structural or functional requirements. This has led to the development of diverse methodologies tailored to address various applications and constraints [3, 4, 5, 6].

In our work, we focus on the challenge of generating graphs with specified properties building on advances in conditioned neural graph generation [6].

The proposed methodology integrates the strengths of graph variational autoencoders (VAEs) and conditioned diffusion processes. The autoencoder learns to encode graphs into a low-dimensional latent vector space and decode them back into graph structures. The conditioned diffusion process incorporates specific constraints to generate latent vectors that, when decoded, produce graphs aligning with desired properties. This approach offers several advantages.

First, performing diffusion in a low-dimensional latent space enhances computational efficiency. Operating directly in the high-dimensional graph space would involve greater complexity and increased training difficulty, especially given the often limited availability of graph data.
Second, if the autoencoder's Kullback-Leibler (KL) divergence loss is substantial, indicating a divergence from a Gaussian distribution, the denoising process can mitigate this discrepancy. It effectively adapts the inherently non-Gaussian latent space to align with the white noise space, ensuring more accurate graph generation when sampling.
Third, by conditioning the diffusion process on specific constraints, our methodology allows for precise control over the properties of the generated graphs.

3 Contributions

We dive in this section into the different contributions and experiment we did for this challenge.

- We used differentiable feature computations to create a specific MSE and MAE loss designed for our targeted conditioning.
- We developed two baselines to compare our models with, one based on random graph sampling until having the right amount of triangles. The second is based on gradient descent to generate the adjacency matrices.
- We tested several encoder architectures (GIN, GAT, and PNA)
- We tested several decoder arhictectures, including the DecoderWithstats, which takes as input the latent representation from the encoder and the desired statistics.
- We embedded the textual statistical description of the graphs with different models for the conditional denoiser and compared its performances to a denoiser using a vector statistical description.
- We tried a constrastive learning approach but we did not get good results at all.

3.1 Varying the objective loss

nodes	edges	Mean Degree	triangles	Global clustering coefficient	max-k-core	communities

Table 1: Graph metrics summary

To build graphs with specific features, we focus on the seven properties shown above, five of which can be differentiated with respect to the adjacency matrix: the number of nodes, edges, triangles, mean degree, and global clustering coefficient. These properties are used in loss functions for our models, evaluated with Mean Absolute Error (MAE) or Mean Squared Error (MSE). They also remain consistent when using soft adjacency matrices, which represent edge probabilities instead of hard adjacency matrices with binary values (0 or 1). To compute our losses on all of the five features, we normalize them using the standard deviation from our dataset to ensure all properties are on a similar scale. In the following, we denote these losses as NMAE (renormalized MAE) and NMSE (renormalized MSE). In practice, we add this loss as a regularization term in the training of our VAE and our diffusion model.

3.2 Baselines

To establish a baseline for evaluating our models' inference, we employed two distinct methods that do not rely on the classic deep learning training and inference framework. We tried an Iterative Edge Generation, starting from an empty graph, we iteratively added edges until the desired number of triangles was achieved using independent bernoulli variables for each not connected edges (Algo 1.). We also tried a gradient descent on adjacency matrix logits weights. We leveraged the differentiable computation of graph properties with respect to the coefficient of the adjacency matrix. Using Gumbel sampling, we generated graphs based on these coefficients. The Mean Absolute Error (MAE) across the first five components was computed in a differentiable manner to train the weights (Algo 2.) .

Algorithm 1 Iterative Edge Generation

function ITERATIVEEDGEGENERATION(num_nodes, target_triangles)
  Initialize $A \leftarrow$ zeros(num_nodes, num_nodes)
  current_triangle_count $\leftarrow 0$
  while current_triangle_count $<$ target_triangles **do**
    for all pairs of unconnected nodes (u, v) **do**
      Add edge (u, v) with probability p (Bernoulli variable)
    end for
  end while
  return A
end function

The results of our baseline methods are summarized in the following table:
Based on this table, we conclude that the main bottleneck of our trained adjacency matrix generation is the lack of max_k_core and communities differentiable metric.
Both methods can be done fast in parallel leveraging modern GPU's computation speed. Iterative Edge Generation takes less then 20 seconds and Gradient Descent method less then 4 seconds to generate the 1000 test graphs.

Algorithm 2 Gradient Descent on Adjacency Matrix Logits

function GRADIENTDESCENTADJACENCY(num_nodes, target_properties, learning_rate, num_iterations)
  Initialize $W \leftarrow \mathcal{N}_{0,1}(\text{num\_nodes}, \text{num\_nodes})$
  for iteration $\in \{1, \ldots, \text{num\_iterations}\}$ **do**
    $A \leftarrow$ sample from Gumbel(W, $tau = 1$)
    current_properties $\leftarrow$ ComputeProperties(A)
    loss $\leftarrow$ MAE(current_properties, target_properties)
    gradient $\leftarrow$ ComputeGradient(loss, W)
    $W \leftarrow W -$ learning_rate $\times$ gradient
  end for
  return A
end function

Method	MAE Normalized	MAE per Component						
		Nodes	Edges	Degree	Triangles	Clustering Coefficient	max_k_core	Communities
Edge Generation	1.1022	0.0000	22.0140	1.2253	7.4460	0.0764	1.4330	0.7360
Trained Adjacency	0.6710	0.0	5.4520	0.3606	36.4710	0.0139	0.797	0.6710

Table 2: Baseline results for graph generation methods.

As we will see in what follows, the Trained Adjacency actually gave us our best results: 0.07 kaggle score whereas our best deep learning model was 0.10 kaggle score.

3.3 Autoencoder

We experimented with various autoencoders architectures to improve the quality of the reconstructed graphs.

3.3.1 Encoder

Other than the base GIN encoder, we also implemented encoders that relies on GAT Conv layers and encoders that relies on PNA Layers.

Graph Isomorphism Networks (GIN) is a sound choice for the embedding task. Indeed, the authors of [7] draw inspiration from the Weisfeiler-Lehman (WL) graph isomorphism test to build a structure that have a high discriminative and representation power. The aggregation to update nodes representations they proposed is as follows:

$$h_v^{(k)} = \text{MLP}^{(k)}\left(\left(1 + \varepsilon^{(k)}\right) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}\right),$$

We also experimented with Graph Attention Networks (GAT) [8]. The key aspect of GATs is that an attention mechanism is used to adapt weights given to neigbhoring nodes for aggregation. We thought this adaptive balancing may yield interesting results and different representations of the graph than those obtained via GIN layers, and may capture more complex interactions.

Lastly, we also experimented with Principal Neighborhood Agregator (PNA), that combines several agregators and scalers like so:

$$\bigoplus := \begin{bmatrix} I \\ S(D, \alpha = 1) \\ S(D, \alpha = -1) \end{bmatrix} \otimes \begin{bmatrix} \mu \\ \sigma \\ \max \\ \min \end{bmatrix}$$

It can be used in a message passing GNN layer: $X_i^{(t+1)} = U\left(X_i^{(t)}, \bigoplus_{(j,i)\in E} M\left(X_i^{(t)}, E_{j\to i}, X_j^{(t)}\right)\right)$ [9]. Intuitively, this allows the model to capture more graph properties and thus more complex structures and patterns. This structure also suits the case where graphs have very diverse local structures which is the case for our synthetised graph datasest. In our code, we used `aggregators = ["mean", "min", "max", "std"]` and `scalers = ["identity", "amplification", "attenuation"]`.

The reconstruction losses for autoencoders trained with those different encoders are plotted in figure 2 in Appendix. Overall, we did not observe significant changes by using PNA or GAT layers compared to using GIN layers.

3.3.2 Decoder

We tried a new deocoder archicteture, which we call DecoderWithStats. Since after the denoiser, we generate graphs based on their latent representation using the decoder, we decided it was important to have a good and accurate decoder. In order to have a more accurate decoder, we concatenate the input with a latent representation of the graph's statistics at each layer, as it is described in figure 1. This is a conditionning of the decoder based on the statistics we want our graphs to have. Using this decoder provided us with better results than the classic decoder. The figure 3 in Appendix shows that the DecoderWithStats clearly reconstructs the graphs better than the Classic Decoder, and that it quicker achieves convergence. Moreover, we were able to achieve even better results with a regularization NMAE loss.

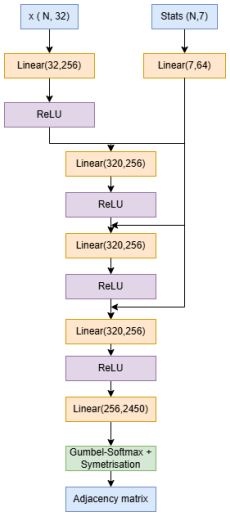

Figure 1: DecoderWithStats architecture.

After training 3000 epochs with the same hyperparameters, the validation Kullback-Leibler divergence loss is 392.2006 for the classic decoder, and 0.0073 for the DecoderWithStats. This means that the DecoderWithStats is able to take a Gaussian white noise as input and, thanks to the conditional decoding based on the statistics, can produce a graph respecting most desired properties. In this context, having an encoder and a denoiser does not bring more information. Therefore, we decided to make an inference on the test set using the DecoderWithStats only. Our best kaggle score with this decoder is 0.10412, which we obtained with a decoder of 7 layers, a hidden dimension of 256 and a latent dimension for the statistics of 64, and a NMAE regularization loss. This score was better than those of models with autoencoder and denoiser combined. With this experiment, we showed that using a conditionnal decoder only is sufficient for our task, while also having far less parameters than a stack of autoencoders and denoiser.

We noticed that the DecoderWithStats gave better results when we added the NMAE regularization loss described in section 3.1 to the classic sum of the Reconstruction and KLD loss. We made the weight of this NMAE regularization vary by multiplying it with a hyperparameter λ_{NMAE} . The table 3 shows the impact of this hyperparameter. Our best kaggle submission score using the DecoderWithStats model was obtained with $\lambda_{\text{NMAE}} = 0.01$. Using large values of λ_{NMAE} does not imply good values on inference on the test set, as we show in the last column of the table. It is indeed also important to have a low KLD and reconstruction losses.

NMAE regularization weight	None	0.0001	0.01	0.05	0.1
NMAE score on test set	1.08	1.01	0.778	0.797	1.81

Table 3: Normalized Mean Absolute Error (NMAE) on test set for the 7 target features of the graphs. This show the impact of λ_{NMAE}, the hyperparameter that controls the weight given to the regularization. The model trained is a DecoderWithStats, and its performances are tested on inference with the decoder only, using white noise as input.

3.4 Using Word embeddings

We tried three different types of word embeddings.

1. The SentenceTransformer library with the *all-MiniLM-L6-v*. We first chose this model because of its small size and because it was specifically trained to embed text and it is easy to use. The embeddings are a dimension of 384. Though, we did not get good results using these embeddings.
   The *all-MiniLM-L6-v2* embedder was trained on short sentences or one sentence at a time, and we used it to embed our graph's representation that is a couple of sentences long. So we used it on a text that was probably out-of-distribution for the model. Furthermore, the text description were automatically generated and present the same structure, only the numbers vary. We think this embedding model was too small to capture the differences between multiple input texts.

2. Bert encoding. [10] We used the pre-trained model 'bert-base-uncased' that is available on the transformers library. It is a classic embedder, so we wanted to test its performance on our data.

3. Jina Embeddings V3: [11] We chose this pretrained embedder, that is available on the transformers library, because it was trained to handle long input sequences up to 8192 tokens. It is based on the Jina-XLM-RoBERTa architecture, so it is built on BERT. One of its specificity is that it provides task-specific

embeddings (retrieval, separation, classification, text-matching), so we thought that it could me more adapted to our task. Moreover, this model supports Matryoshka Embeddings, so the embeddings can be truncated to different sizes, which can be useful to reduce computation while keeping the most important information in the embeddings. We first wanted to see if this embedding was able to hold any useful information at all, so we used the full embedding size of 1024.

To compare the three word embeddings and the baseline (the seven parsed statistics), we used the same already trained autoencoder. The results are presented in the table 4. As we can see, the results are not improved when using any of the three word embeddings. We got better results by simply parsing the relevant numbers from the text input. Among the word embedders, the Jina- Embeddings V3 showed the best results on the val loss, but the denoiser on the parsed statistics presents a validation loss that is twice lower. This is explained by the specificity of our data: since there is only two types of text, and each type contains the same exact sentences except for the numbers, using a word embedding on our text data doesn't add more information. All useful information is already contained in a clearer and more concise way in the *data.stats* tensor, so we were not suprised that using text embedding did not yield better results. However, if we had more diverse text inputs, on which a parser could not be applied, using text embedding could produce reasonable results. The plots of the train and validation losses for the Bert and Jina V3 embedding in the figure 4 in Appendix also show that such word embeddings are difficult to use for the denoiser: the validation results show significant variations for both word embeddings during training.
Therefore, in most of our experiments, we stick to the conditional denoiser on the parsed statistics.

Models	Train Loss after 1000 epochs	Val Loss after 1000 epochs	Best Train loss	Best Val Loss
BERT embedding	0.0179	0.0204	0.0159	0.0127
Jina Embedding	0.0162	0.0153	0.0155	0.0111
Sentence-Transformers pre-trained	0.0547	0.0494	0.0517	0.0384
Parsed Statistics (dim = 7)	0.0108	0.0069	0.0108	**0.00561**

Table 4: Results after training denoising model for 1000 epochs with the same hyperparameter, such as 3 layers in the decoder with a hidden dimension of 512

3.5 Contrastive Learning

We tried contrastive learning techniques, by implementing an Information Noise-Contrastive Estimation (InfoNCE) loss. The idea is to maximize similarity between "close" datapoints. To do so we added a loss term to maximize the cosine similarity between a latent representation z_i of a graph and the latent z_j of an augmented version of the same graph z_i, $cosine\_sim(z_i, z_j) = z_i z_j^T / ||z_i z_j||$. However, using InfoNCE did not lead to significant improvements on the reconstruction loss for the autoencoder or in the global performances for the latent diffusion process. We hypothesize that this can be because the latent space in our case has to encompass precise information about the number of nodes or edges for example. When generating augmented data, by dropping edges or nodes for example, a lot of crucial information is lost and, to some extent, we do not want too much similarity between the latent representations of a graph and its augmented version.

4 Conclusion and Future approaches

In conclusion, we got our best results (0.07741 kaggle score) with a baseline approach that has only has a few parameters to train. We also showed that using a conditional decoder provides better results (0.10412 kaggle score) than an using a diffusion model on the latent space of an autoencoder. Due to the specificity of our machine-generated data, using word embeddings do not improve our results with diffusion models.
If we had more time, we would have liked to perform extensive data augmentation with different techniques, as we think this could improve the training of deep models. We also would have liked to try the WGAN implementation, and the Discrete Graph Denoising Diffusion (DiGress) architecture.

References

[1] Ines Chami et al. *Machine Learning on Graphs: A Model and Comprehensive Taxonomy*. 2022. arXiv: 2005.03675 [cs.LG]. URL: https://arxiv.org/abs/2005.03675.

[2] Ziwei Zhang, Peng Cui, and Wenwu Zhu. *Deep Learning on Graphs: A Survey*. 2020. arXiv: 1812.04202 [cs.LG]. URL: https://arxiv.org/abs/1812.04202.

[3] Xiaojie Guo and Liang Zhao. *A Systematic Survey on Deep Generative Models for Graph Generation*. 2022. arXiv: 2007.06686 [cs.LG]. URL: https://arxiv.org/abs/2007.06686.

[4] Yanqiao Zhu et al. *A Survey on Deep Graph Generation: Methods and Applications*. 2022. arXiv: 2203.06714 [cs.LG]. URL: https://arxiv.org/abs/2203.06714.

[5] Clement Vignac et al. *DiGress: Discrete Denoising diffusion for graph generation*. 2023. arXiv: 2209.14734 [cs.LG]. URL: https://arxiv.org/abs/2209.14734.

[6] Iakovos Evdaimon et al. *Neural Graph Generator: Feature-Conditioned Graph Generation using Latent Diffusion Models*. 2024. arXiv: 2403.01535 [cs.LG]. URL: https://arxiv.org/abs/2403.01535.

[7] Keyulu Xu et al. *How Powerful are Graph Neural Networks?* arXiv:1810.00826 [cs] version: 3. Feb. 2019. DOI: 10.48550/arXiv.1810.00826. URL: http://arxiv.org/abs/1810.00826.

[8] Petar Veličković et al. *Graph Attention Networks*. arXiv:1710.10903 [stat]. Feb. 2018. DOI: 10.48550/arXiv.1710.10903. URL: http://arxiv.org/abs/1710.10903.

[9] Gabriele Corso et al. *Principal Neighbourhood Aggregation for Graph Nets*. arXiv:2004.05718 [cs]. Dec. 2020. DOI: 10.48550/arXiv.2004.05718. URL: http://arxiv.org/abs/2004.05718.

[10] Jacob Devlin. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[11] Saba Sturua et al. *jina-embeddings-v3: Multilingual Embeddings With Task LoRA*. 2024. arXiv: 2409.10173 [cs.CL]. URL: https://arxiv.org/abs/2409.10173.

5 Appendix

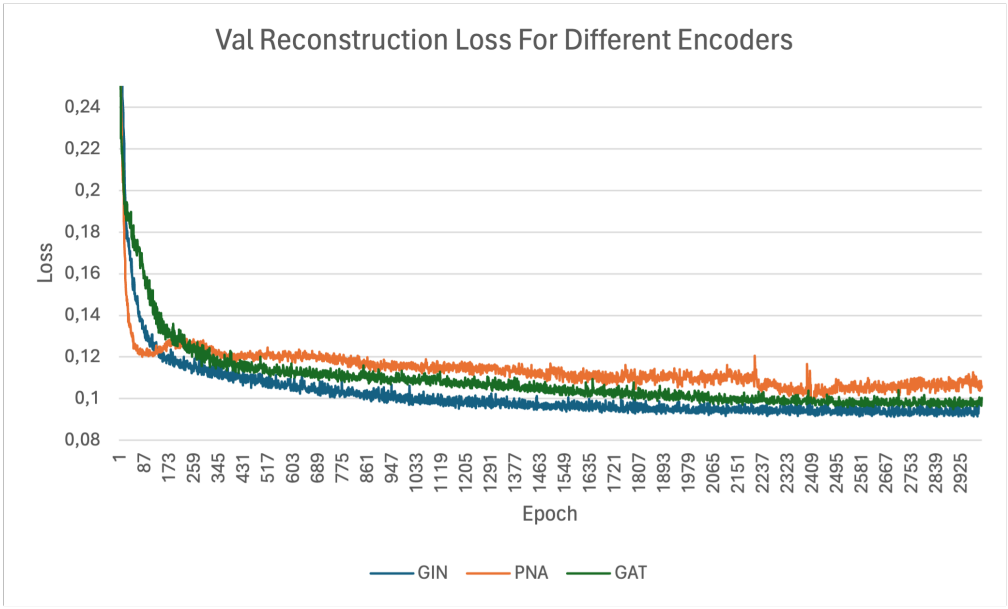

Figure 2: Comparing the validation loss during training for different encoders.

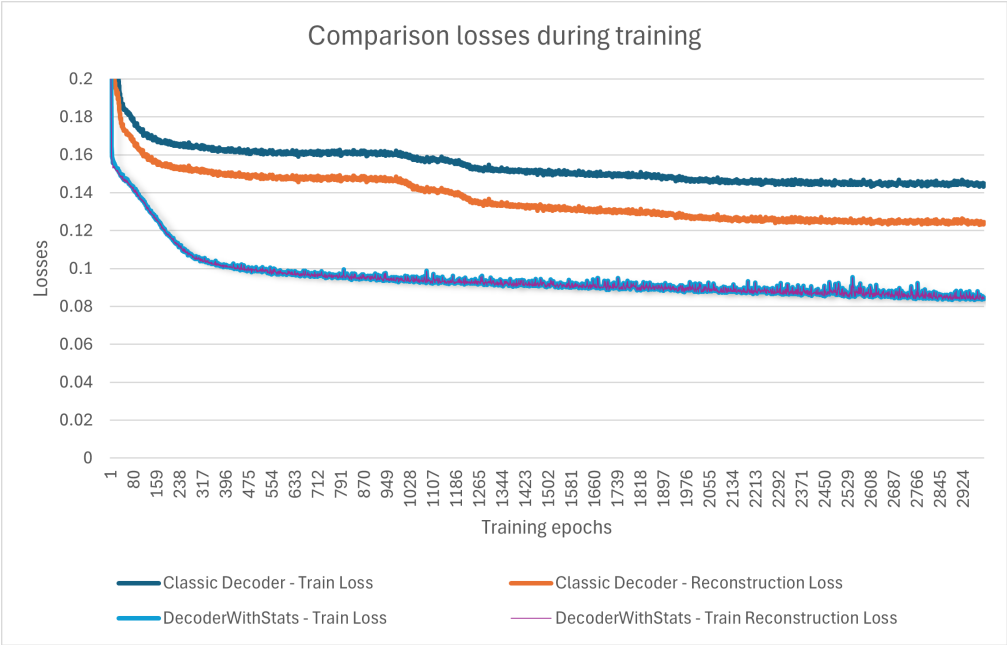

Figure 3: Comparison Training Losses Classic Decoder and DecoderwithStats, all other hyperparameters equal.

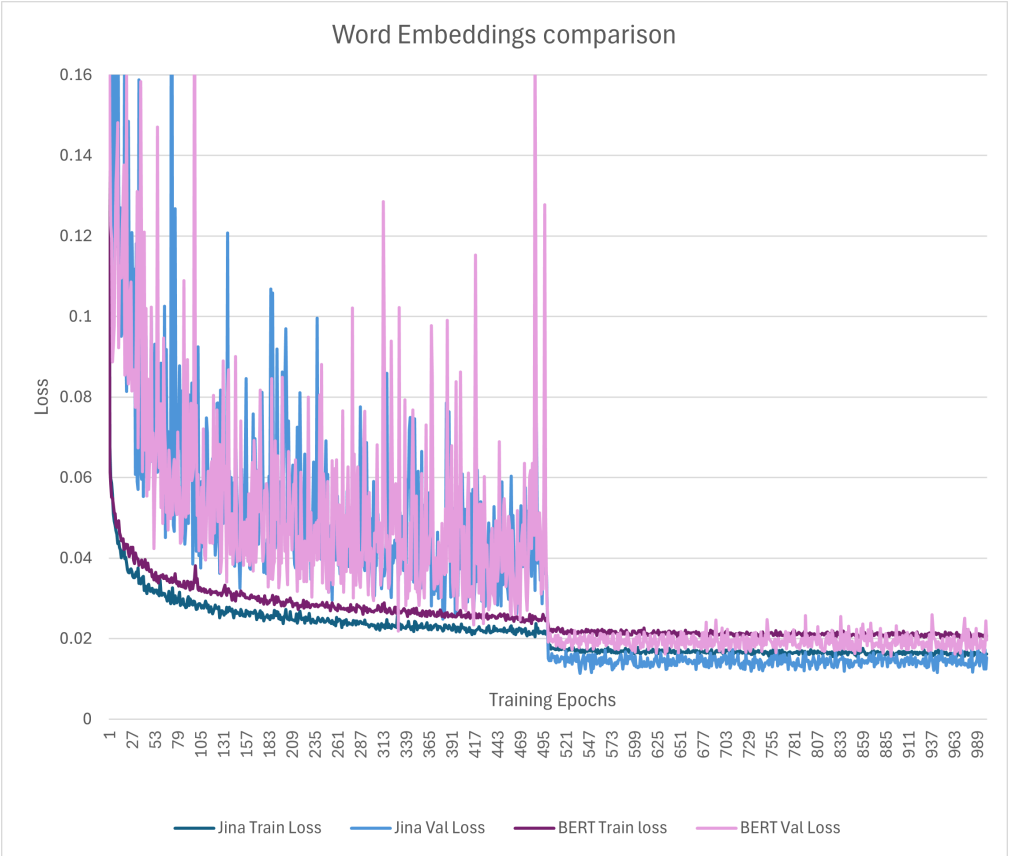

Figure 4: Comparing the train and validation loss during training for different word embeddings.