Modification of the Graph Generation Process in NetGAN using Node Attributes

(Guided Research)

René Filip
rene.filip@tum.de

Aleksandar Bojchevski
aleksandar.bojchevski@in.tum.de

Daniel Zuegner
zuegnerd@in.tum.de

Stephan Günnemann
guennemann@in.tum.de

***Abstract*—Graph generation plays an important role in biology, sociology and computer science as many problems can be modeled as a graph. Explicit generation methods usually succeed in capturing one specific property of a graph but fail to capture the others. To tackle this problem, "NetGAN" introduced an implicit method that can learn multiple properties of a graph by using the "generative adversarial network" architecture. Even though NetGAN performs well in comparison to state of the art methods, there is still room for improvements. In this guided research project we first give further insights into the generation process of NetGAN by visualizing how random walks are generated. Then, we propose improvements for the generator's architecture to generate graphs with different sizes and without retraining.**

I. Introduction

NetGAN [1] introduces a new method to generate big graphs and networks. Many methods have been introduced to generate new graphs with certain properties [2]. However, these methods usually work in an explicit way and one needs to state explicitly which properties a new graph should have. Additionally, explicit methods usually only focus on one or a few graph properties and it is not clear how to combine multiple methods together. In contrast, NetGAN introduces an *implicit* generation process that learns graph properties of an input graph and outputs a new, different graph with similar properties. NetGAN builds on the Wasserstein GAN architecture [3] which is widely used in the computer vision community to generate new images from a desired distribution.

A. Architecture

Like any other Generative Adversarial Network [4], NetGAN consists of a generator and a discriminator. Instead of generating the graph directly, we generate random walks instead. A random walk is a sequence of vertices and edges with a fixed size and once we generated a sufficiently large amount of random walks, we use them to construct the new desired graph. The generator tries to generate plausible random walks while the discriminator has to judge whether a random walk has been generated artificially or is a real one from the input graph. Both the generator and the discriminator use a Long Short-term Memory (LSTM) structure [5] as a model.

Figure 1 summarizes the generation process more in detail. We will now highlight important components that play a role for us later, but we refer to the original paper [1] for more details.

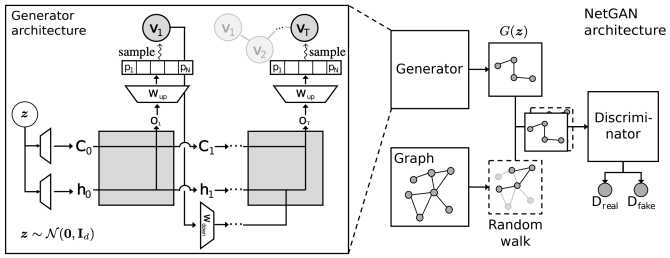


Fig. 1: NetGAN architecture

Let N be the number of nodes of the input graph (V, E) and let T be the length of a random walk. Each new node v_{i+1} of the walk is generated by an LSTM-cell that receives the previously sampled node v_i, the cell state C_i and the hidden state h_i as an input. The cell outputs a vector $p \in \mathbb{R}^N$ from a categorical distribution and the new node v_{i+1} will be then sampled from it.

However, because we are dealing with large graphs, N can become very large and it can become very costly to output p directly. Instead, the cells output a smaller vector $o \in \mathbb{R}^K$ where $K \ll N$. Then, to sample from it we perform an up-projection using the matrix $W_{\text{up}} \in \mathbb{R}^{K \times N}$. Similarly, after sampling the N-dimensional vector it needs to be reduced to K again by down-projecting it with the matrix $W_{\text{down}} \in \mathbb{R}^{N \times K}$. Both W_{up} and W_{down} will be learned during training.

To sample from a categorical distribution we perform the Gumbel-max trick [6][7]: Let z be sampled from a $\text{Gumbel}(0, 1)$ distribution. Then, the new node is determined by using the arg-max operation

$$v_{i+1} = \underset{j \in \{1,\dots,N\}}{\arg\max}\ p_j + z_j$$

Obviously, we cannot differentiate this step. Instead, during the backward pass we perform a Softmax operation which can be thought of as a smooth approximation of the arg-max operation and which is indeed differentiable.

$$p_i = \frac{\exp\left((\log \pi_i + g_i)/\tau\right)}{\sum_{j=1}^{k} \exp\left((\log \pi_j + g_j)/\tau\right)} \tag{1}$$

The parameter τ controls the competing goals between a good approximation but bad gradients and vice versa.

B. Limitations

Due to its architecture there are currently two limitations that we are going to focus on: First, our random walks can only generate a graph with the same number of nodes as the input graph, namely N nodes (or smaller, if some nodes are never sampled). However, certain graph properties are independent of size and we would be interested in generating graphs with different sizes. Second, once our input graph changes, we cannot propagate these new information into an existing, already trained model. At the moment, we would have to retrain the whole model to account for a tiny graph change.

The following sections are outlined as followed: Before we modify NetGAN, we want to get a better understanding of how random walks are generated in the first place. To do that, we show a simple visualization technique in II. Then, we modify NetGAN's architecture to solve the two problems above in section III. To measure its performance, we introduce an additional evaluation metric in section IV and perform experiments in section V. At the end, we summarize our results and give a short outlook in VI.

II. Visualizing Random Walks

Before we modify the up and down sampling process in the generator, we first want to better understand it. To do so, we visualize this process by using a t-SNE [8] plot of the up and down projection matrix W and highlight one instance of a random walk. This allows us to better understand how random walks are build step by step.

Let $W_{\mathrm{up}} \in \mathbb{R}^{K \times N}$ and $W_{\mathrm{down}} \in \mathbb{R}^{N \times K}$ be the matrix responsible for the up and down sampling respectively. We will focus our analysis on W_{up} but the same technique can be applied on W_{down} as well. The up projection is a simple vector-matrix multiplication or a scalar product and during training we can think of it as a similarity measure. Each of the N columns of W_{up} represents a K-dimensional vector of a node and we can visualize their relationship to each other by using t-SNE.

Figure 2a and 2b shows such a plot for the data set "Cora ML" [9] and "Citeseer" [10]. The coloring matches each node's label. We notice that the nodes exhibit certain structure within the graph. Not only they form clusters but the clusters also matches their labels. This is interesting, because we did not give NetGAN access to the label data. One could follow that nodes within one class are more "similar" than labels from another. Of course, by looking at the data set this is not surprising but it is good to see this behaviour reflecting in the graph.

Now, in order to visualize a random walk of length T, we are going to modify NetGAN slightly: Each LSTM cell should not only sample the next node for the walk, but it should also output the N-dimensional probability vector which has been used for the sampling. We then use these T vectors and generate T plots each showing which node has been sampled before (visualized by a black line) and which nodes might be sampled next (indicated by the blue transparent spheres where the transparency is set by the node's probability to be sampled next). Remember the two "modes" of our sampling procedure. We use the "hard", arg-max mode to sample the next node and we use the "smooth", Softmax mode to visualize alternatives. For τ we chose 0.5 as this value is already small but still helps us to show some alternative nodes. However, the "hard" sampling is independent of it.

Figure 3 and 4 show one random walk for Cora ML and Citeseer respectively and in the Appendix VII we listed a few more. In general, once a node jumps into one cluster, it stays there and only samples nodes around him. This makes sense for Cora ML and Citeseer, because papers refers to other papers from the same research community much more often than from other communities. Additionally, a random walk might also consists of one node multiple times because of that. The same can be said for the offered alternatives highlighted by blue circles. Nevertheless, it also happens that the random walk changes a cluster. In figure 3 the random walk sampled one node from another one but then returned back to the old cluster. In the Appendix there are also graphs where jumping occurs more often.

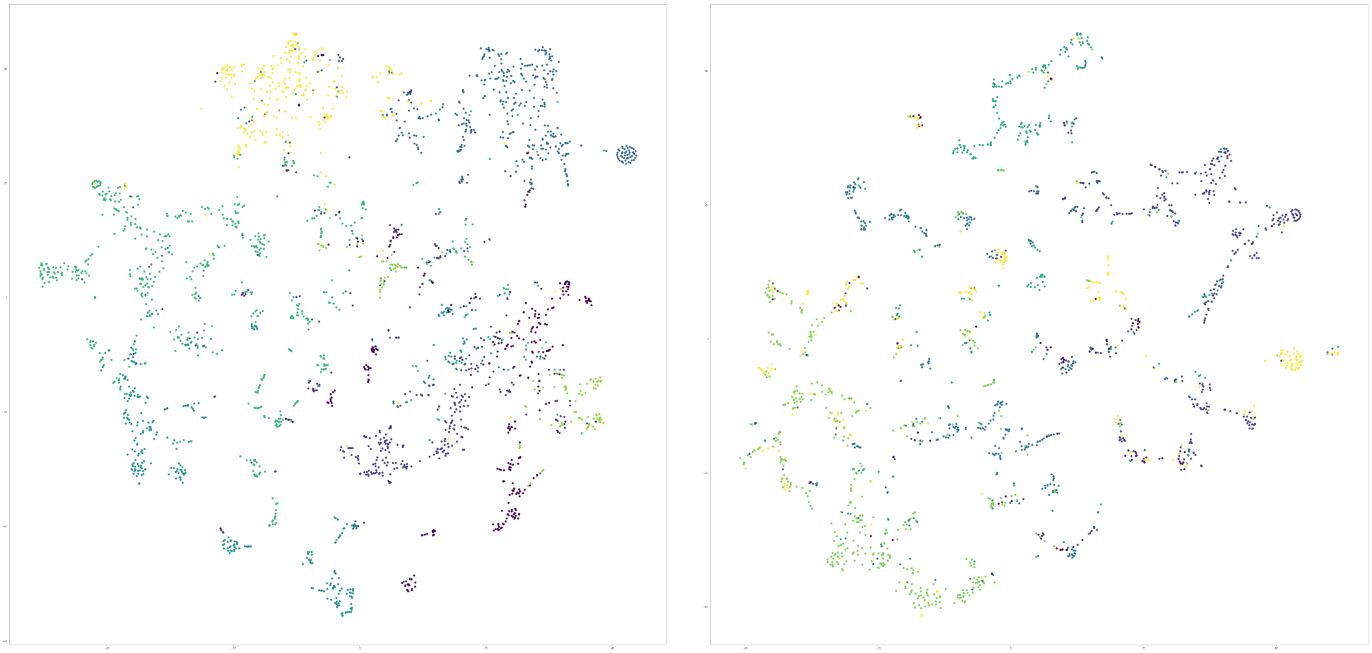
III. NN

We would like to have a model that can generate graphs with different node sizes. One obvious idea would be to extend W_{up} and W_{down} to sample from a probability vector with N' elements instead of N. However, it is not clear how one would extend W with meaningful data. In order to solve this problem, we will first highlight the second problem with NetGAN: Assume your input graph is not fixed but grows (in terms of nodes) over time. Currently, for any new node we would have to fully re-train NetGAN again to make usage of the new information. This can become very time consuming and inefficient.

A. Neural Network Method

Many graph data sets do not only contain the structure of the graph but also provide additional node attributes, i.e. each node comes with an additional vector of a certain dimensionality that can contain any kind of data. Can we use this information for our problem? Our change to NetGAN uses these node attributes to "identify" a node and to generate the up and down projection for it. Instead of learning W directly during the training, NetGAN learns a neural network f_W which takes node attributes $X \in \mathbb{R}^{N \times D}$ as input and outputs two matrices $W_{\mathrm{up}} \in \mathbb{R}^{K \times N}$ and $W_{\mathrm{down}} \in \mathbb{R}^{N \times K}$. Notice that the dimensions of W depend on the input dimension X, i.e. if X grows in size, then also does W. Or in other words, if the graph changes over time, we do not have to retrain NetGAN again because we can reuse the weights of f_W to generate our new projection matrices.

Figure 5 shows how the change fits into the existing structure of NetGAN. It comes with the consequence that two nodes with similar node attributes also generate similar columns (or rows) in W. One need to evaluate whether this behaviour is meaningful for a given data set. In addition, the node attributes need to be expressive enough to correctly



(a) Cora ML (b) Citeseer

Fig. 2: t-SNE plot with label information

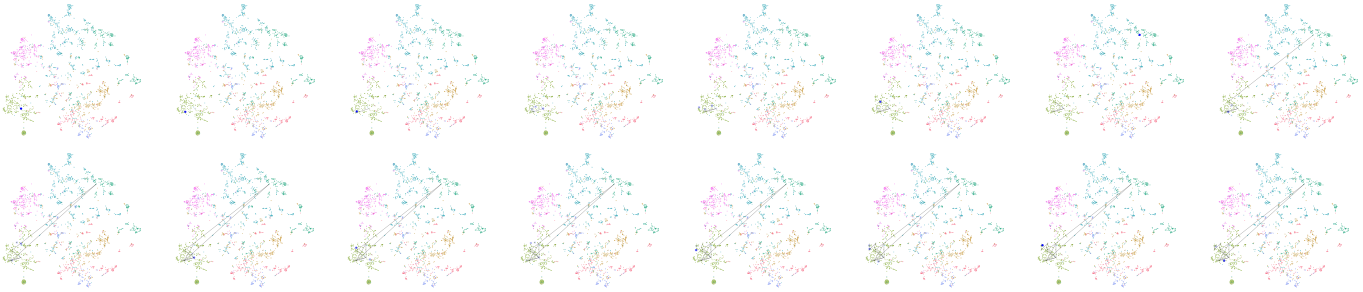


Fig. 3: Random Walk of Cora ML

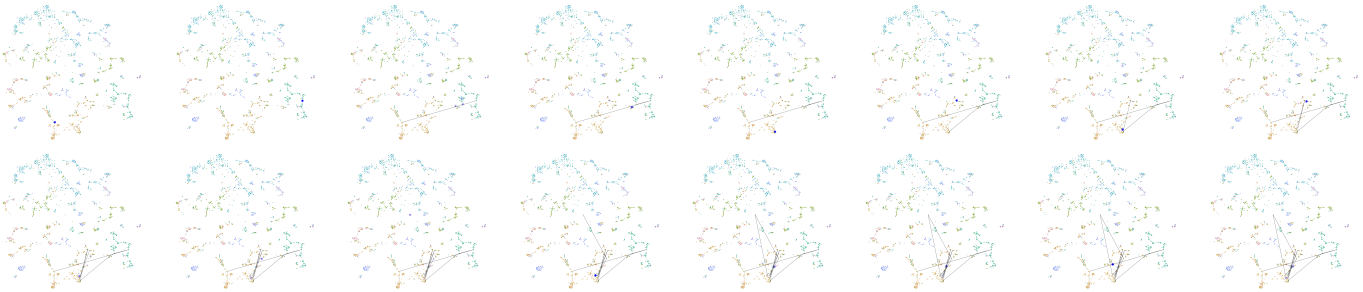


Fig. 4: Random Walk of Citeseer

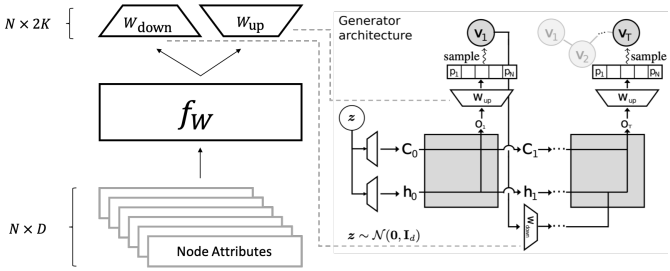


Fig. 5: NetGAN Modification

distinguish clearly different nodes. For instance, if all nodes own the same attributes, then our approach clearly will not work as we would then sample all nodes with an uniform probability distribution in contrast to the real one.

B. 1-Nearest-Neighbour Method

As an alternative, we also introduce a simpler, second method that we will use as a baseline later in experiment V-A. Instead of passing the node attributes through a neural network to generate W, we could also use the node attributes only for identifying the most similar node and then copy its column (or row) from the existing free variable matrix W. In specific, this method consists of following steps:

1) Train the original NetGAN implementation using a graph $G = (V, E, X)$ where $X \in \mathbb{R}^{|V| \times D}$ represents the node attributes of V. Save the up and down projection matrix W_{up} and W_{down}.
2) Obtain a new graph $G' = (V', E', X')$ where $|V'| > |V|$.
3) For each node in G', find its 1-nearest-neighbour in G in respect to their node attributes X' and X. Use the cosine distance as a similarity measure.
4) Each node copies the corresponding column (or row) from W_{up} (or W_{down}) that matches to their 1-nearest-neighbour in G and constructs a new up and down projection matrix W'_{up} and W'_{down} which are now larger in size.

As a similarity measure for 1-NN we use the cosine distance because we are only interested in the orientation of the vectors but not in their lengths.

In contrast to the neural network method, this one does not generate new entries in W but only copies existing entries. On the one hand, this change gives us fewer parameters and hyperparameters in our model which results into faster learning and easier parameter tuning. On the other hand, the model might not perform well when the node attributes are all different altogether. We would expect, that a good neural network should learn the underlying distribution of them and also generalize to node attributes which might differ heavily.

IV. DeepWalk Evaluation

We introduce a new evaluation metric based on DeepWalk helping us to judge our modification from the previous section. The DeepWalk [11] method creates a latent representation of the vertices for a given graph that tries to capture underlying graph properties. A graph (V, E) will be transformed into a collection of d-dimensional vectors of size $|V|$ where each vector represents a vertex of the graph. Revisit the t-SNE plot in figure 2a and 2b that also show representations of vertices in two dimensions. We notice that each node is colored by it's corresponding label and that the nodes build clusters by their labels, even though, we never gave NetGAN access to the label information. Can we use these label information to evaluate a generated graph by NetGAN?

First, we generate a certain number of random walks that represents a new graph. Second, we feed these random walks into DeepWalk to generate a latent representation of the nodes. Third, we combine the latent representation with the label information and train a simple logistic regression classifier using the classification accuracy as a new evaluation metric for NetGAN. If we generate a graph that captures the label information well, then the classification score will be high. Otherwise, the generation process is flawed and the classification score will be low.

Because randomness influences the evaluation score, we try to lower its variance by sampling sufficiently many random walks, using k-fold cross validation for the classification and by running the evaluation metric (with DeepWalk) multiple times and averaging over it.

V. Experiments

We highlight three experiments that led some insights into the proposed modification.

A. Graph Shrinking

We would like to see how our method performs once new graph data is available. To do that we would train a model with the current graph, wait until the graph changes and then reinitialize NetGAN using its old weights but new node attributes to obtain a different generation process. However, because our graph data sets do not grow in their node size, we emulate this growth by doing it the other way round, namely by shrinking it to a smaller size and then adding nodes back. First, we shrink the graph. Second, we train NetGAN using the smaller graph. Third, we use the model of the smaller graph but reinitialize NetGAN with the *full* graph. Last but not least, we evaluate the DeepWalk accuracy of the full graph.

For shrinking, we use the following algorithm: Start with your full graph. Randomly remove 1% of its nodes and assert that the new graph is still connected. If it is not connected, undo and repeat the removal step until a suitable set of nodes is found. Repeat the whole procedure until the graph consists of 10% of its original number of nodes. Notice that by following this algorithm each smaller graph will be a sub-graph of the previous ones. This is another countermeasure to lower the effects of randomness in the evaluation process later.

We will now describe the experiment more in detail: We use Cora ML as our data set and generate 91 different sub-graphs (10% to 100%). For each sub-graph we learn a new NetGAN model with these hyperparameters

Hyperparameter	Value
Data set	Cora ML
Batch size	128
Max. iterations	20000
Evaluation frequency	2000
W_{down} discriminator size	128
W_{down} generator size	128
W_{up} discriminator size	30
W_{up} generator size	40
f_W hidden layers	[336]
Discriminator iterations	3:1
L^2 generator penalty	10^{-7}
L^2 discriminator penalty	$5 \cdot 10^{-5}$
Learning rate	10^{-4}
Temperature start	5
Random walk length T	16

TABLE I: Hyperparameters

Size	f_W	f_W w/ Batch Norm	1-NN
10%	0.495	0.386	**0.560**
20%	0.537	0.482	**0.587**
30%	0.594	0.524	**0.617**
40%	0.635	0.597	**0.664**
50%	0.691	0.665	**0.701**
60%	0.705	0.695	**0.715**
70%	**0.751**	0.733	0.731
80%	0.759	0.752	**0.772**
90%	0.799	0.792	**0.808**
100%	0.827	**0.832**	0.824

TABLE II: Graph Shrinking Results, DeepWalk Accuracy

Due to time constraints and the amount of models we have to train, we limit the maximum number of iterations to 20000. For W_{up} and W_{down} we use the same dimensions as in the original NetGAN paper. For our neural network, we use a leaky ReLU activation function with only one hidden layer of size $2 \cdot (40 + 128) = 336$. Therefore we reduce the node attributes dimension from 2879 over 336 to 168.

For the evaluation, we perform the DeepWalk 10 times with a graph that has been generated by 600000 random walks. We report the classification accuracy. Sometimes, it could happen that one node might not get sampled ever within the random walks and that the size of the result graph does not match to the size of the labels. In that case, we "artificially" sample it by adding a random walk that consists of 16 instances of that node. Alternatively, we could also remove the nodes' label and then perform the classification. In our experiments both methods performed similar scores so we will only report results for the first method. The next table summarizes the hyperparameters for the evaluation step.

Hyper Parameter	Value
Number of random walks	600000
DeepWalk dimension	64
DeepWalk evaluations	10
Training size	60%
Cross Validation	5 fold

In an earlier experiment the performance was not satisfying so we also experimented whether "Batch Normalization" [12] could boost our performance. Figure 6 and table II present our outcome.

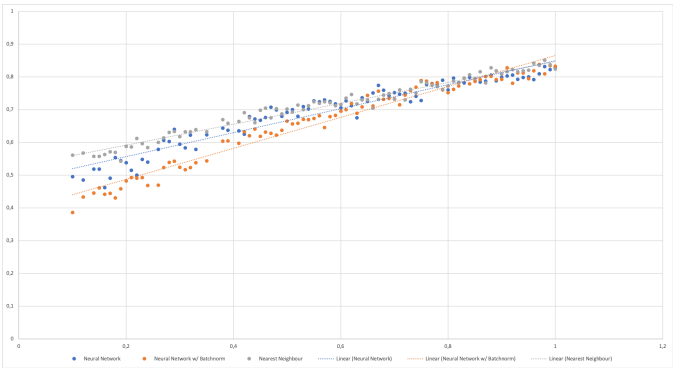


Fig. 6: Shrinking graph experiment results, DeepWalk performance

Surprisingly, the most simple method "1-nearest-neighbour" (1-NN) performed best and batch normalization even made the performance worse. Even though between 90% and 100% results are very similar, the lower percentages show a noticeable difference between the methods. One reason why the performance is not great might be that our neural network method needs a graph with a certain size and that 10% of the original graph could be too low as it does not capture enough information. However, we would still expect to beat the simple baseline more often.

B. Hyper-Parameter Search

Another possible reason why we could not do better is that we used the wrong hyperparameters. The neural network f_W only used one hidden layer with a more or less arbitrary size and it is not clear whether this is the problem for the bad performance. In our search we explore other 1-layer networks with more neurons, multiple hidden layers and whether normalization of the node attributes makes sense. Here, we left out batch normalization again because it slowed down training and made results worse in the previous experiment. Besides the layer of f_W, we used the same values from table III.

Here, we use a higher maximum of iterations to rule out the possibility that our training was too short. In addition, we also report the Link Prediction ROC results that we get by the evaluation steps during training. For DeepWalk we use the same evaluation parameters but change the metric from a simple accuracy to F_1 scores. The results are summarized in table III. The numbers in the table refer *only* to the hidden layer sizes. For example, "336 - 336" would refer to the following dimensionality reduction: $2879 \rightarrow 336 \rightarrow 336 \rightarrow 168 = 40 + 128$

By looking at the results, normalization of the node attributes seem to make the performance slightly worse. Also, increasing the size of neurons in the 1-hidden layer architecture worsens our results, even though this could also be influenced by random noise. For the two- and three-layer architectures the results are clearly worse. Because we modified the generator and added many new weights, the L^2 penalty factor might be off. Therefore, we fixed the "336 - 168" network and modified

1	2	3	Norm?	ROC	DeepWalk F_1
672	0	0	yes	0.917	0.744
672	0	0	no	0.924	0.779
504	0	0	yes	0.917	0.786
504	0	0	no	0.913	0.783
336	0	0	yes	0.919	0.773
336	0	0	no	0.915	0.783
168	0	0	yes	0.916	0.778
168	0	0	no	**0.927**	**0.79**
336	336	0	no	0.88	0.713
336	168	0	no	0.904	0.703
168	336	0	no	0.881	0.729
168	168	0	no	0.883	0.712
336	168	168	no	0.823	0.482
168	168	168	no	0.857	0.639

TABLE III: Hyperparameter search results

the regularization:

L^2 **Regularization**	**ROC**	**DeepWalk** F_1
10^-3	0.510	0.276
10^-4	0.886	**0.745**
10^-5	0.871	0.713
10^-6	**0.904**	0.719
10^-7	0.888	0.718

A different L^2 regularization improves the results a bit but still does not beat the original NetGAN performance.

C. Interpolation between Free Variables and Neural Network

Due to the results above, we also investigate whether certain degrees of freedom are necessary to restore the original NetGAN performance. Instead of generating the projection matrix W *only* by the node attributes, we allow each nodes to "adjust" by it's own by training an additional free variable projection matrix W^{free}, like we do in the original implementation. Then, we interpolate between the neural network and the free variables. In specific, we perform following operation for the up projection. Let $\alpha \in [0, 1]$,

$$W_{\text{up}} = \alpha f_{W_{\text{up}}} + (1 - \alpha) W_{\text{up}}^{\text{free}}$$

Setting $\alpha = 0$ should restore the original NetGAN architecture while $\alpha = 1$ provides the proposed modification. But, the disadvantage of this method is we cannot generate larger graphs anymore because we are using $W_{\text{up}}^{\text{free}}$. The purpose of this experiment is rather to debug NetGAN. Again, we use Cora ML and 200000 as a maximum of iterations and left all other parameters to the ones from table I. The size of the hidden layer is 336.

Table IV and figure 7 shows our results. Setting $\alpha = 25\%$ already worsens the Link Prediction scores while the DeepWalk classification stays roughly constant, except for $\alpha = 75\%$. We would expect that $\alpha = 0\%$ performs best and as good as our baseline. Interestingly enough, the first is true while the latter is not. Again, this could have been caused due to high variance within our measurement method. Figure 7 shows the

Free, $1 - \alpha$	**Neur. Network,** α	**ROC**	**DeepWalk**
Original	-	**0.949**	**0.783**
100%	0%	**0.924**	0.761
75%	25%	0.872	**0.766**
50%	50%	0.866	0.759
25%	75%	0.874	0.727
0%	100%	0.876	0.765

TABLE IV: Interpolation Results

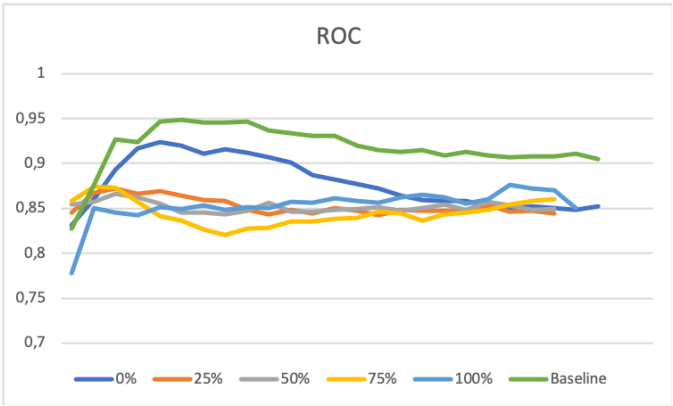


Fig. 7: Link Prediction ROC

link prediction ROC scores during training. We observe that all $\alpha \in \{25\%, 50\%, 75\%, 100\%\}$ archive similar (bad) results.

The graphs in figure 8 visualize for each α the corresponding W_{up}. Naturally, the plot for $\alpha = 0$ looks like the plots we have seen before in section II. Increasing α seems to dissolve the clusters more and more. Nevertheless, the nodes still cluster according to their true label in all figures. The dissolving indicates why the link prediction and DeepWalk performance are not as good as it was expected.

VI. Conclusion

In this guided research paper we presented one method to better understand and debug NetGAN regarding its generation process and we proposed two solutions how we could extend NetGAN to generate graphs with different sizes. Even though we showed that training was in general possible, we could not beat the performance of the original implementation. One possible reason could be that we have to conduct a larger hyperparameter search for our neural network modification, for example we did not experiment with the learning rate. Neither did we try different preprocess techniques like dimensionality reductions for our input graph to make our training better. Also, by adding a new neural network to NetGAN, we increase the size of parameters and more memory and time is necessary. For smaller graphs like Cora ML this is not a huge problem but for larger graphs this can make training more difficult due to a smaller batch size or too small memory on the GPU. The neural network proposal also made training slower. It would be interesting to investigate how to speed it up again.

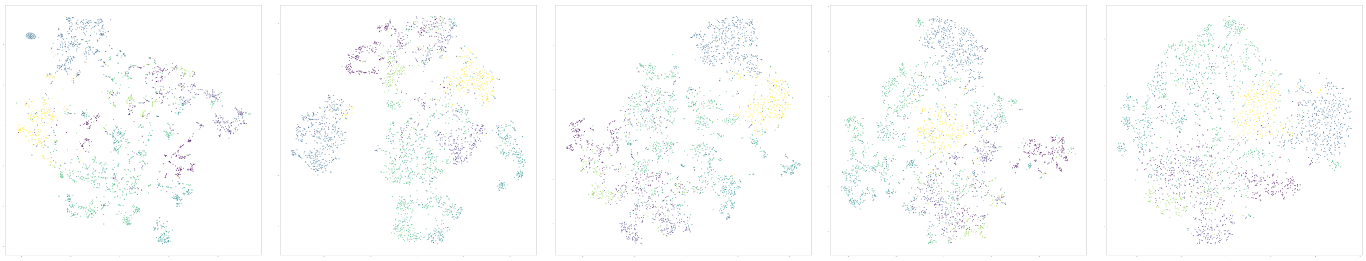


Fig. 8: W Visualization for Interpolation Experiment

REFERENCES

[1] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann, "NetGAN: Generating Graphs via Random Walks," *arXiv e-prints*, p. arXiv:1803.00816, Mar 2018.

[2] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Comput. Surv.*, vol. 38, June 2006.

[3] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv e-prints*, p. arXiv:1701.07875, Jan 2017.

[4] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *arXiv e-prints*, p. arXiv:1406.2661, Jun 2014.

[5] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[6] C. J. Maddison, D. Tarlow, and T. Minka, "A* sampling," in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 3086–3094, Curran Associates, Inc., 2014.

[7] E. Jang, S. Gu, and B. Poole, "Categorical Reparameterization with Gumbel-Softmax," *arXiv e-prints*, p. arXiv:1611.01144, Nov 2016.

[8] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[9] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, "Automating the construction of internet portals with machine learning," *Information Retrieval*, vol. 3, pp. 127–163, Jul 2000.

[10] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad, "Collective classification in network data," tech. rep., 2008.

[11] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," *CoRR*, vol. abs/1403.6652, 2014.

[12] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

VII. APPENDIX

A. Additional Random Walk Visualizations

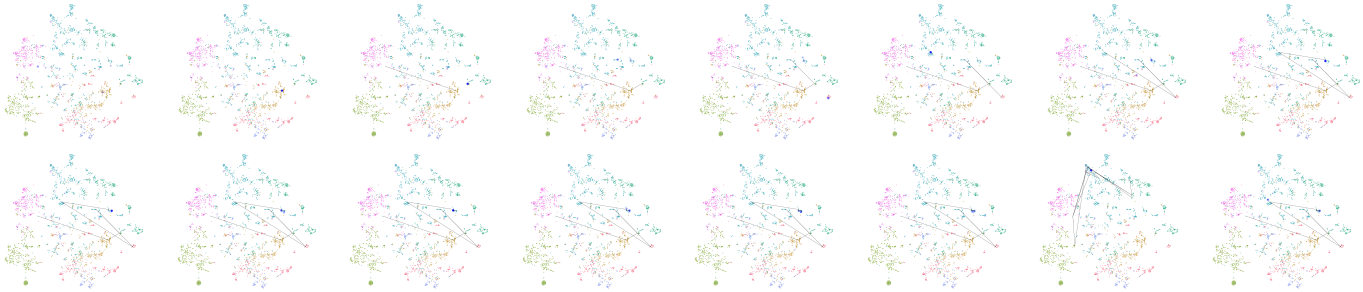


Fig. 9: Cora ML, Additional Plot 1

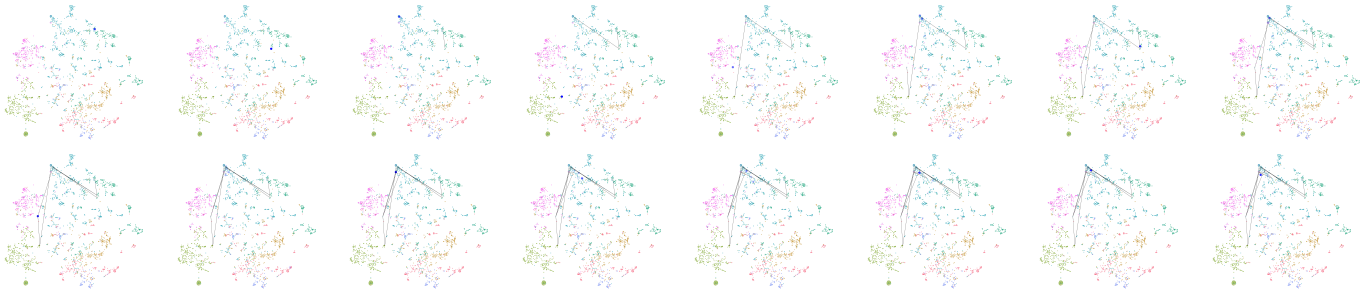


Fig. 10: Cora ML, Additional Plot 2

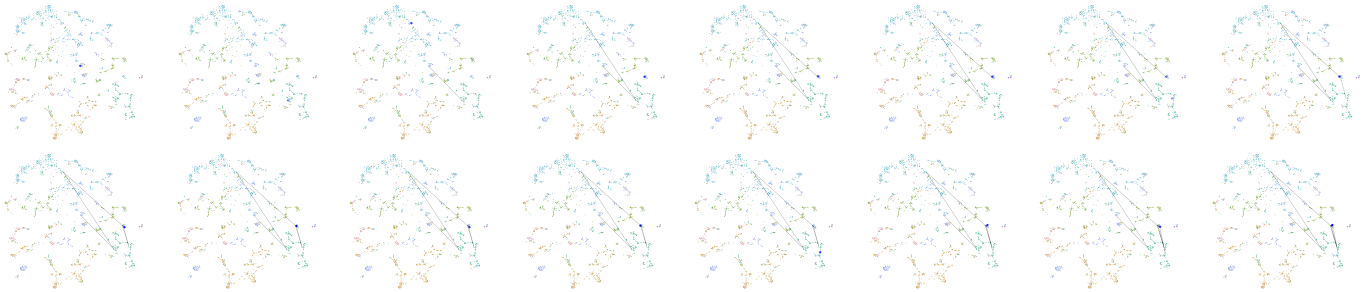


Fig. 11: Citeseer, Additional Plot 1

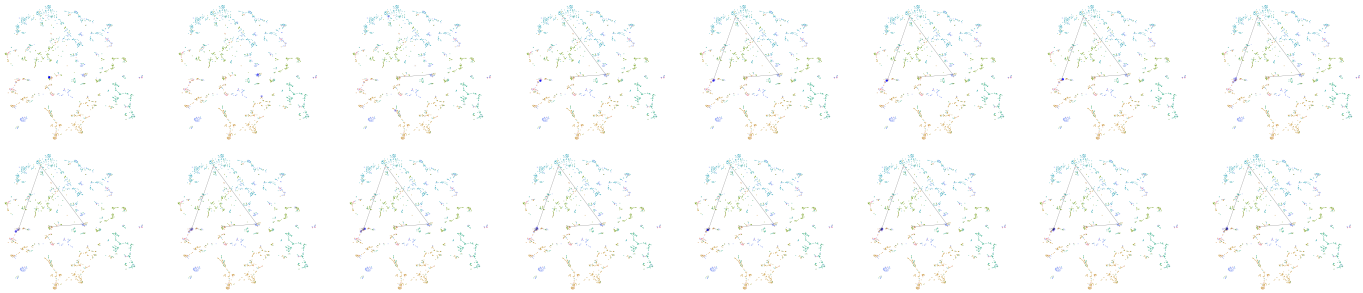


Fig. 12: Citeseer, Additional Plot 2