"Link or Sink" Navigating the Network with GNNs

Paolo Lapo Cerni[†], Lorenzo Vigorelli[†]

***Abstract*—Complex networks are everywhere, from biology to social science, but the reliability of interaction measures often raises concerns. The *link prediction* problem has been traditionally tackled using several approaches, though often with limited generalizability. Additionally, machine learning frameworks encounter challenges with non-Euclidean geometry and thus the need for a *geometric deep learning*. In this paper, we introduce *graph neural networks* (GNNs) and study their capability to reconstruct a partially observed graph, going beyond the standard cases and architectures. We investigate the role of the *message-passing* and how the training can improve the initial performance, depending on the network topology and the number of training data or node features. Our resulting architecture has been proven robust to changes in the main hyperparameters and effective with a small fraction of training edges and node metadata. However, this framework seems unable to infer the structure of a random graph from the Barabasi ensemble, revealing some possible limitations of this approach. We believe that both industry and academia could benefit from a deeper understanding of GNNs and geometric learning.**

***Index Terms*—Deep Learning, Graph Neural Networks, Graph Autoencoders, Geometric Learning, Link Prediction, Networks Reconstruction**

I. Introduction

The analysis of complex networks has recently become a flourishing research area, attracting scientists with different backgrounds and expertise. Indeed, many different systems can be seen in terms of a set of objects (*nodes*) and their relationships (*edges*), and then studied as graph structures. The possible applications span social sciences, protein-protein interactions, physical systems, computational neuroscience, telecommunication networks, knowledge graphs, and many others. All these examples share the same conceptual model: they can be represented as a network.
However, the reliability of network data is often a source of concern because of the difficulties in measuring and modeling such complex phenomena. For instance, your network realization can come with missing and spurious interactions (i.e. being *corrupted*), or be only partially observed. In this paper, we focus on the second case, attacking the so-called *link prediction* task. We imagine knowing precisely the number of nodes and some information about them but having collected only a subset of the edges, trying to *reconstruct* the full network. This is not merely a theoretical exercise, but it mirrors the real-world challenge of measuring interactions, often coming at a significant cost in terms of time and resources.

[†]Department of Physics and Astronomy, University of Padova

Traditional link prediction strategies rely on node similarity scores, some latent embedding of the node space, or explicit nearest neighbor heuristics [1]. However, every heuristic comes with strong assumptions, limiting its effectiveness and generalizability [2]. Moreover, graphs define a *non-Euclidean geometry*, where most standard machine learning tools encounter significant challenges. Extending deep learning to these domains has been an emerging research area, generally referred to as *geometric deep learning* [3]. In particular, the *graph neural networks* (GNNs) are an attempt to adapt standard learning techniques to network data [4]. We focus on autoencoder structures and convolutional layers to effectively aggregate the information about nodes and reconstruct the partially observed network, providing a comprehensive guide for understanding how such GGNs work and identifying their limitations. We investigate the applicability of these techniques to more general network topologies and learning frameworks, going beyond the standard cases and architectures studied in the literature and trying to unpack the reasons for the performances.
In this paper, we first briefly introduce graph neural networks and their advantages based on the recent literature (Section II). The specific task, network architecture, and learning tools are defined in sections III and IV, and mathematically characterized in section V. Section VI details the performances under different scenarios and configurations. Concluding remarks are provided in Section VII.

II. Graph Neural Networks

Studying graph-structured data with machine-learning techniques requires adapting these methods to handle the unique structure of graphs, including their non-Euclidean nature. Graph-learning tasks are usually divided into three main groups: node-level (e.g. node classification or node clustering), edge-level (e.g. edge classification or link prediction), and graph-level (e.g. graph classification, graph matching, or network regression) [5].
Link prediction is a common problem that has traditionally been tackled using a variety of approaches, which can be categorized into three main types: heuristic methods, latent-feature models, and content-based techniques [1]. Standard heuristics methods use some measure of node similarity as a proxy for the likelihood of links. The simplest ones are based on the local number of common neighbors, but one can define some higher-order metrics that summarize global knowledge about the network (e.g. Kaltz index or Rooted PageRank). On the other hand, latent-feature models extract low-dimensional

node embedding from the graph. Typically this is done with a matrix factorization based on inner products, but one can easily try to enrich these node representations. Finally, content-based methods leverage metadata and explicit features of the nodes to reconstruct the connections between them. Each of these methods is suited to specific contexts, but they often lack generalizability [1], [2].

The deep-learning framework tries to unify these paradigms. Many different architectures have been developed to tackle different challenges, with GNNs emerging as a key approach [5], [6]. A major problem is that the adjacency matrix of a network depends on the arbitrary choice of node ordering, and learning all the possible permutations is practically unfeasible because of the combinatorial complexity. Therefore, it is essential to design your architecture with this consideration in mind. For example, network predictions must be *invariant* to node reordering, while node predictions must be *equivariant* to the same operation [4].

Several architectures, including graph convolutional networks, can be framed in the *message-passing* scheme [4], [7]. Each processing layer has two successive stages: the *aggregation* and the *update*. For each node, we first aggregate information coming from all its neighbors. This step is designed to be equivariant under node reordering. Then, the aggregated knowledge from the neighbors is combined with local information from the node itself and used to update its embedding vector. More formally, for each node n and each layer l, we introduce a column vector $h_n^{(l)}$ for the embedding. This vector is typically initialized using available node metadata. We define the $\texttt{Aggregate}(\cdot)$ function such that the vector

$$z_n^{(l)} = \texttt{Aggregate}(\{h_m^{(l)} : m \in \mathcal{N}(n)\}) \tag{1}$$

respects the symmetry of the system, being $\mathcal{N}(n)$ the neighbors of node n. The simplest aggregate function is the summation. It's important to notice that this step captures key information about the network even before any optimization takes place, summarizing both topological and context-based information. The training plays a role in strengthening the results. Then, another operation is used to update the representation of n, i.e.:

$$h_n^{(l+1)} = \texttt{Update}(h_n^{(l)}, z_n^{(l)}). \tag{2}$$

Both these functions can contain some learnable parameters, as long as they are differentiable with respect to them [4]. For example, a simple form for the latter operator could be $\texttt{Update}(h_n^{(l)}, z_n^{(l)}) = f(W_{\text{self}} h_n^{(l)} + W_{\text{neigh}} z_n^{(l)} + b)$ where $f(\cdot)$ is a nonlinear activation function and W_{self}, W_{neigh}, and b are learnable parameters.

In this paper, we aim to combine information about the nodes and some training edges to learn a latent representation for an undirected graph and reconstruct the full network, i.e. its adjacency matrix. In so doing, we use a non-probabilistic graph auto-encoder (GAE) model as first proposed by Kipft and Welling [8]. This architecture includes convolutional layers and fits indeed in the message-passing framework. It's interesting to notice that this approach has been generalized to tackle the problem of $M \times N$ matrix completion, mapped into the one of link prediction in bipartite graphs [9].

III. Processing Pipeline

The processing pipeline of the edges resembles a standard encoding-decoding procedure, enriched by some node metadata. The idea is to consider an observed network A^O, whose links are a random subset of those in the "true" underlying network A. To simplify the analysis, we consider only *undirected unweighted networks*. The first condition means that $A_{ij} = A_{ji}$ and thus the whole information is contained in the upper triangular part of the matrix. Secondly, being unweighted means that A_{ij} assumes only binary values: $A_{ij} = 1$ if the edge (i, j) is present, $A_{ij} = 0$ if it is absent. Combining the available information, we want to predict whether a link (i, j) was in A, namely the value of A_{ij}.

It is assumed to have collected some information X about the objects of the network A [8]. These features can include any numerical attributes that describe the nodes. For instance, in a social network, these features might represent user attributes like age or number of followers. In a citation network, they might represent properties of a paper such as word count or citation count. Moreover, some could derive from standard network science quantities, including centrality measures (e.g. betweenness, closeness, PageRank, ...), the clustering coefficient, and many others. It is important to highlight that this information possibly refers to the true A. For instance, the degree of a node derives from the analysis of the full set of edges and not only the observed ones. Tuning the number of available features is part of the pipeline assumptions. With no information, X can be substituted with the one-hot encoding matrix [1].

The observed network and the feature vector are then processed by an *encoder*. This module aggregates and summarizes the available information to represent the nodes in a latent space. It produces a low-dimensional embedding, often coming from the neighborhood of each object. In the standard architecture [8], the encoder consists of multiple layers of graph convolutions, framed in the message-passing scheme. However, one can try to enrich the analysis with different modules (e.g. attention-based, recurrent, or skip connection).

The *decoder* in a GAE aims to reconstruct the graph structure from the learned node embeddings. It takes the output of the encoder and uses it to estimate the likelihood of edges between node pairs. In so doing, one can infer the original structure by ranking node pairs according to their predicted edge probabilities. During the training, the output of the decoder is compared with the true edges of A^O to learn a meaningful node representation.

IV. Data and Features

Since the entire pipeline focuses on a single network, the role of training, validation, and testing is slightly different from the usual one. The full set of edges of the "true" network A is divided into these three classes to emulate a

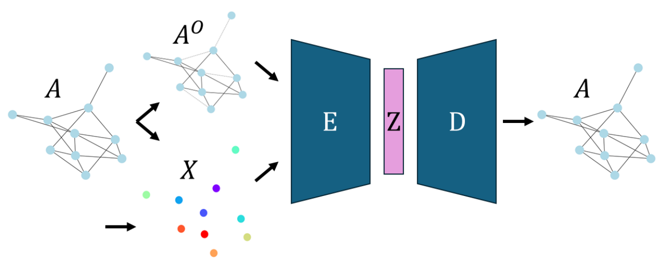

Fig. 1: Information flow described in section III.

real-world scenario, with the encoding and decoding phases having different treatments. This difference arises from the fact that during the encoding one wants to extract information about the edges we know, while the decoding phase requires examples of non-edges, allowing the loss function to take them into account as well. Moreover, the encoding accounts for the fact that we are considering an undirected network, resulting in tensors that are doubled in size due to the inclusion of mirrored versions of the edges [10].

Let's focus first on the training steps. During the encoding, the embedding is produced using the whole set of training edges, and nothing else. Conversely, the decoder uses a 50/50 mix of training edges and non-edges to ensure a fair accuracy evaluation. The non-edges are obtained via *negative sampling*, i.e. randomly chosen from all possible node pairs other than the training edges. This assumes the graph to be *sparse* so that the probability of sampling actual edges is negligible.

During validation, the encoder once again utilizes the same set of training edges, i.e. all the edges that have been already disclosed in the pipeline. The difference occurs in the decoding phase, where the validation edges are employed. The validation decoding set is enriched with an equal number of non-edges. Differently from the training, here the non-edges are assumed to be known.

Finally, the test encoding edges combine both the training and validation sets. The decoding uses the remaining edges, i.e. the full test set, enriched by the same number of known non-edges. Indeed, the "game" is to combine all the available known information to produce an accurate node representation from which to infer the presence/absence of the wanted (test) connections.

The following table describes the number of the (edges, non-edges) in training, validation, and testing for an 85/5/10 split for a graph with 1000 edges. Italic is used to indicate the negative sampling, while $\times 2$ means the mirroring of the tensor for undirected networks.

	Training	Validation	Testing
Encoding	(850, 0)$\times$2	(850, 0)$\times$2	(900, 0)$\times$2
Decoding	(850, *850*)	(50, 50)	(100, 100)

Tab. 1: Splitting example

This paper analyzes two networks and their respective set of features. Firstly, we analyze the PUBMED network from the Planetoid framework [10], [11]. This graph is characterized by 19,717 nodes and 44,324 edges, representing documents and citation links respectively. The data comprehends 500 node features from document metadata and graph analysis. Secondly, we test a much smaller synthetic scale-free network, obtained by the Barabasi-Albert preferential attachment ($m = 1$). This dataset has 2001 nodes (so 2000 edges), and 9 node features only coming from standard network science quantities (e.g. clustering, betweenness, Katz centrality, PageRank, etc). This network ensemble reproduces features typically found in real networks, such as power-law degree distribution and the emergence of hubs.

The features of both datasets have been row-normalized by subtracting the minimum value and dividing by the sum row-wise, making each row sum one. This scaling stabilizes the training and prevents any single feature from dominating.

V. Learning Framework

As mentioned in section III the model consists of an auto-encoder structure. The encoder is formed by two graph convolution network (GCN) layers, while the decoder reconstructs the adjacency matrix with an inner product [12]. A GCN combines node features and the graph structure to iteratively update node representations. It is one of the most popular GNN modules due to its effectiveness in several domains, including link prediction, node classification, and graph classification. In a GCN, the node representations are updated layer by layer using the following propagation rule:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{3}$$

where $H^{(l)}$ is the node representation matrix at the l-th layer, $\tilde{A} = A + I$ is the adjacency matrix with self-loops added (i.e., each node is connected to itself), $\tilde{D}$ is a diagonal matrix where $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ represents the degree of node i in $\tilde{A}$, $W^{(l)}$ is a layer-wise trainable weight matrix, and $\sigma(\cdot)$ is a non-linear activation function, commonly ReLU or the identity function. This propagation rule allows each node to update its representation by aggregating features from its neighbors and incorporating its own features, normalized by their degrees. Using the normalized matrix $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ stabilizes the training process and prevents nodes with high degrees from overly influencing the representation.

As explained in section II, GCNs update node representations through the message-passing framework. Indeed, equation 3 for each node i can be decomposed in the aggregate and update steps. The `AGGREGATE` function collects information from neighboring nodes, with contributions weighted based on the connection strength and normalized by the degrees of both nodes:

$$z_i^{(l)} = \sum_{j \in \mathcal{N}(i)} \frac{\tilde{A}_{ij}}{\sqrt{\tilde{D}_{ii}\tilde{D}_{jj}}} H_j^{(l)} \tag{4}$$

This ensures each node learns from its local neighborhood in a balanced way, regardless of degree variations.

Then, the node updates this aggregated message with its own transformed feature representation:

$$H_i^{(l+1)} = \sigma\left(\left(z_i^{(l)} + \frac{1}{\tilde{D}_{ii}}\tilde{A}_{ii}H_i^{(l)}\right)W^{(l)}\right) \tag{5}$$

In the `UPDATE` step, the node representation is retained along with the aggregated neighborhood information.

GCNs have different strengths that make them effective: (i) they gather information from neighboring nodes, which captures *local graph patterns*; (ii) GCNs are computationally *efficient*, using simple operations like linear transformations and weighted averages, which are faster than more complex methods like graph attention networks (GATs); (iii) They work well across many types of graphs and tasks, making them very *versatile*.

At the last layer, the adjacency matrix is reconstructed using an inner product of the embeddings:

$$\hat{A} = \sigma\left(ZZ^T\right) \tag{6}$$

The sigmoid is used as the output function: it transforms the raw scores (logits) into values that range between 0 and 1. Specifically, the output can be interpreted as the probability that a link exists between each pair of nodes. The sigmoid function is defined as $\sigma(x) = 1/(1 + e^{-x})$. By mapping the logits to the interval [0, 1], we can assess the likelihood of a link's existence, with values above 0.5 indicating the presence of a link.

The optimizer used in this model is *Adam*, which adapts the learning rate for each parameter individually. The chosen loss function is *binary cross-entropy*, well-suitable for classification tasks. This allows us to interpret the raw output scores (logits) from the model, where positive scores indicate a higher probability of a link and negative scores indicate a lower probability.

To evaluate the performance of the algorithm, we primarily used the *Area Under the Curve* (AUC), which represents the area under the *Receiver Operating Characteristic* (ROC) curve. The ROC curve illustrates the trade-off between the true positive rate and the false positive rate at various threshold settings. AUC provides a single metric that summarizes the model's ability to distinguish between the positive and negative classes, making it particularly useful in cases of class imbalance. This choice was made instead of relying solely on classification accuracy. Indeed, the latter relies on the 0.5 threshold and can be misleading when output distributions are unbalanced, even if they are well separated (see figure 5). It's however important to emphasize that the loss is based on the accuracy and not on the AUC.

Several strategies were implemented to mitigate overfitting during training. Firstly, the network structure was modified, as shown in figure 2, including batch normalization layers and dropout. Secondly, an early stopping algorithm was employed to stop training when the validation AUC does not improve after a specified number of epochs (*patience*) by a certain threshold. This approach ensures we obtain the best model based on the highest validation AUC value.

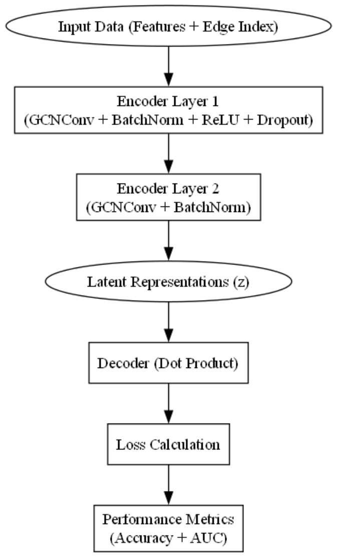

Fig. 2: Architecture of the encoder-decoder model.

To sum up, our model has a total of 34,336 trainable parameters, divided into:

- Weights and bias of the convolutional layers
- Weights and bias of the batch normalization layers

Conversely, the main hyper-parameters are: the optimizer, learning rate, weight decay, loss function, sizes of the convolution layers (i.e. input, hidden, and output channels), dropout percentage, maximum training epochs, patience, and threshold for early stopping.

In addition to using graph convolutional networks, we also implemented other autoencoder models. Firstly we implemented a version of this model with also a recurrent layer (gated recurrent unit) since it can integrate temporal or sequential information that the convolutional layers might not capture. Secondly, we incorporate graph attention networks (GATs). In this approach, the *attention mechanism* allows the model to weigh the importance of different neighboring nodes when aggregating information. This enables the model to focus on the most relevant nodes, improving the graph data representation. Furthermore, we implemented a variational autoencoder (VAE) model as an alternative to the standard autoencoder (AE). VAEs are beneficial because they introduce a probabilistic element, allowing the model to learn a distribution of the input data rather than just a fixed point. This can lead to better generalization and the ability to capture complex structures in the data, making them a promising choice for graph representation learning [5].

VI. Results

A. Best Architectures

As mentioned in the previous section, we developed different encoders to maximize our evaluation metric, the test AUC on the PubMed dataset. The full implementation is publicly available on GitHub, and we can summarize the approaches as follows:

- AE: convolutional layers (GCN)
- AE: GCN + dropout + batch normalization
- AE: graph attention layers (GAT)
- AE: GCN layers + graph recurrent layers (GRNN)
- VAE: based on GCN layers

Table 2 presents the performance of each model, showing the mean and standard deviation over 50 independent training runs with compatible encoder sizes. The best and most stable AUC is achieved using the GCN, enriched by batch normalization and dropout. The GRNN, obtained from the former adding a recurrent layer, does not improve the performance. This is expected because our data is not inherently sequential. However, the GRNN has a much larger set of parameters, being more prone to overfitting. Thus, from now on, we will prefer the less complex GCN+d+bn architecture.

Model	**Accuracy**	**AUC**	**#Params**
GCN	0.717 ± 0.012	0.90 ± 0.02	34144
GCN+d+bn	0.789 ± 0.004	0.9755 ± 0.0009	34336
GAT	0.807 ± 0.005	0.9519 ± 0.0012	68832
GRNN	0.775 ± 0.010	0.9736 ± 0.0014	53152
VAE	0.761 ± 0.008	0.943 ± 0.008	36352

Tab. 2: Performance and parameters of different models

B. Training and validation Curves

Figure 3 shows a typical learning curve for the standard convolutional model without dropout and batch normalization for an 85/5/10 splitting. We recall that, as explained in section V, the loss function tries to maximize the accuracy, even if the AUC is a better performance measure. Indeed, the loss function presents the same "bounces" of the accuracy history, obviously with different slopes. The full convergence requires around 200 epochs, even if the accuracy seems to be stacked in a plateau for a large number of iterations. After the peak, the validation accuracy starts decreasing: a sign of some overfitting. On the other hand, figure 4 reports a typical learning history for the full GCN+d+bn model. The rise is faster and the curves are much smoother, representing a more stable training due to the batch normalization layer.

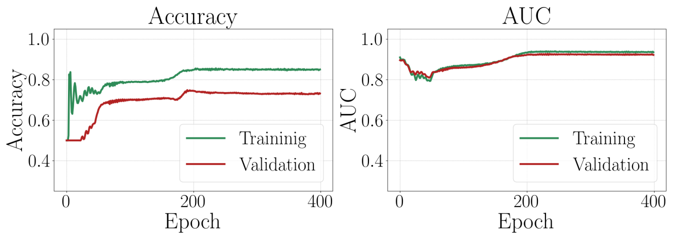

Fig. 3: Learning curves GCN.

We can appreciate that the AUC initially starts at high values, then drops after a few epochs, and eventually rises again, reaching values even higher than the initial ones. This reflects that the message-passing framework can capture edge properties from the training sample even with little or no optimization. Proper training is needed to gain robustness in discriminating between positive and negative examples, as shown in figure 5. At first, the two output distributions both peak around 0.5 with little (but not negligible) discernibility. Conversely, they completely overlap during the AUC drop. The full distribution is properly bimodal only after the training, with the edges having a score peak of 1. However, the non-edges do not peak around 0, as one could expect from a classification problem. This is because they are sampled at each epoch, unlike the edges (see section IV).

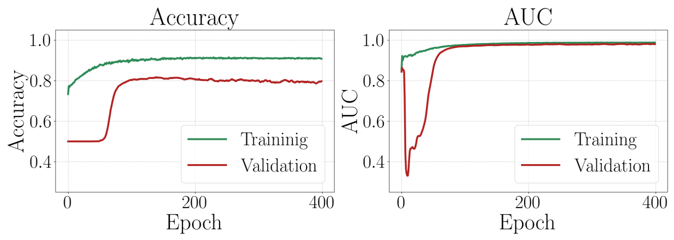

Fig. 4: Learning curves GCN+d+bn.

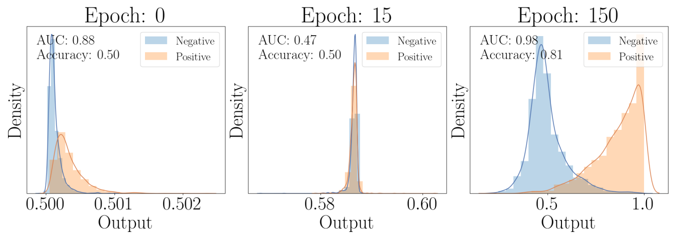

Fig. 5: Output distributions in validation.

C. Robustness of the training

We test the robustness of our GCN+d+bn performance, varying the main hyper-parameters. Firstly, we scan the number of the encoder's hidden and output channels. Our grid search suggests that the results remain stable as long as both embeddings have more than 32 entries. Furthermore, the optimal dropout probability is approximately around 0.5.
The optimizer comes with some hyper-parameters too. The highest AUC scores are obtained using a learning rate of 0.01 (see table 3) and a weight decay of 5e-4. Moreover, we use an early stopping patient of 200 epochs. This is highly conservative but assures to overcome the initial AUC fall. Further implementations could enrich this mechanism, for instance with some epoch-dependent relaxation.

L. r.	0.0001	0.001	0.01	0.1	1
AUC	0.95	0.97	0.97	0.94	0.87

Tab. 3: Role of the learning rate

D. Training Set and Features Sizes

Recalling that the PUBMED network has 44,324 edges and 500 node features, it's interesting to study what is the impact of these numbers on the learning capabilities.
The role of the training set size is shown in figure 6. The splitting is $p/5/(1-p)$, being p the percentage of observed edges. Surprisingly, even with only 10% of the edges, we can reach a discernibility of 93%. This means that this

architecture can infer satisfactorily the structure of a much more dense network than the training one. In other words, we can reconstruct the network starting from a relatively small number of observed interactions.

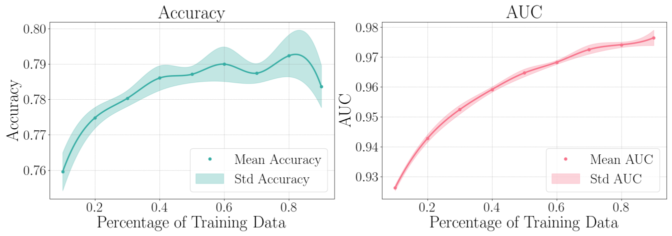

Fig. 6: AUC for different training size percentages

Also the number of node features we assume to know plays an important role. In the PUBMED network, most of them refer to some node metadata. Figure 7 shows that we need much less information to infer the full graph also in this case. This is interesting because it opens the possibility of reconstructing a network without relying too much on side information.

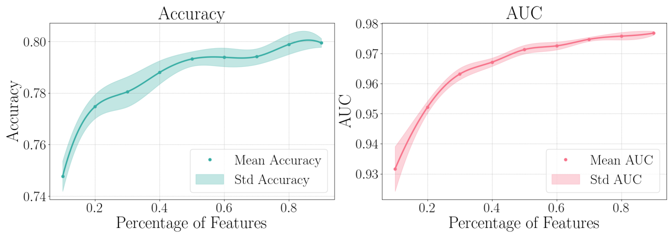

Fig. 7: AUC for different feature size percentages

E. Synthetic network

We explore the capability of these GNNs to reconstruct a synthetic random network, drawn from the Barabasi ensemble. The node features are 9 quantities computed from the network topology only (recall section IV). The learning curves for an 85/5/10 splitting of the 2000 edges are shown in figure 8 (x-axis in log scale). The validation AUC stays consistently at 85% for the first 10 epochs before becoming highly unstable. This reflects that the only useful information comes from the message-passing, while the training is not able to effectively divide the positive and negative output distributions, that remain largely overlapped. Our result seems to indicate that this framework cannot adequately infer the structure of Barabasi random graphs in the absence of nodes' side information. This analysis remains qualitatively equivalent even with slightly larger networks (tested up to 10,000 edges).

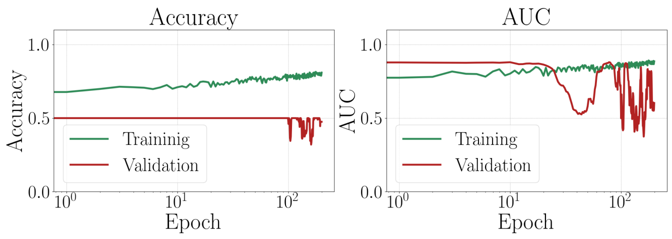

Fig. 8: Learning curve for the Barabasi network

VII. Concluding Remarks

We tested the capability of GNNs to reconstruct partially observed networks. Comparing different architectures, we assessed that the autoencoder with convolution, dropout, and batch normalization delivers the best performance with a simpler structure and that this learning framework is robust to changes in its hyperparameters. Additionally, it remains effective even with significantly fewer training edges and node features. The main limitations are discovered while dealing with the Barabasi ensemble with few features, where the only information comes from the message-passing.
Further studies might investigate several other directions, including generative frameworks. An interesting problem is to reconstruct a partially corrupt network with missing and spurious edges. This requires more studies on the interplay between GNNs, network ensembles, and feature selection.
We believe graph neural networks offer a powerful learning framework for industrial and scientific applications, extending deep learning to non-Euclidean geometries and opening a full new set of problems to be tackled.
Our main challenges centered around conceptualizing the link prediction problem. We shifted our initial focus from studying corrupted networks, due to the complexities of working with ensembles, to a deeper exploration of GNN fundamentals. Additionally, investigating GNN architectures proved difficult, as this is a relatively new and evolving field.

Data Availability: the methods used in this study and supplementary figures are publicly available on GitHub.

References

[1] M. Zhang, "Graph neural networks: Link prediction," in *Graph Neural Networks: Foundations, Frontiers, and Applications* (L. Wu, P. Cui, J. Pei, and L. Zhao, eds.), pp. 195–223, Singapore: Springer Singapore, 2022.
[2] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in neural information processing systems*, vol. 31, 2018.
[3] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.
[4] C. M. Bishop and H. Bishop, "Graph neural networks," in *Deep Learning: Foundations and Concepts*, pp. 407–427, Springer, 2023.
[5] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.
[6] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 249–270, 2020.
[7] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*, pp. 1263–1272, PMLR, 2017.
[8] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.
[9] R. v. d. Berg, T. N. Kipf, and M. Welling, "Graph convolutional matrix completion," *arXiv preprint arXiv:1706.02263*, 2017.
[10] "PyTorch Geometric Documentation." https://pytorch-geometric.readthedocs.io/en/latest/. [Accessed 25-Oct-2024].
[11] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting semi-supervised learning with graph embeddings," in *International conference on machine learning*, pp. 40–48, PMLR, 2016.
[12] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.