SLAPS: SELF-SUPERVISION IMPROVES STRUCTURE LEARNING FOR GRAPH NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Graph neural networks (GNNs) work well when the graph structure is provided. However, this structure may not always be available in real-world applications. One solution to this problem is to infer the latent structure and then apply a GNN to the inferred graph. Unfortunately, the space of possible graph structures grows super-exponentially with the number of nodes and so the available node labels may be insufficient for learning both the structure and the GNN parameters. In this work, we propose the **S**imultaneous **L**earning of **A**djacency and GNN **P**arameters with **S**elf-supervision, or SLAPS, a method that provides more supervision for inferring a graph structure. This approach consists of training a denoising autoencoder GNN in parallel with the task-specific GNN. The autoencoder is trained to reconstruct the initial node features given noisy node features as well as a structure provided by a learnable graph generator. We explore the design space of SLAPS by comparing different graph generation and symmetrization approaches. A comprehensive experimental study demonstrates that SLAPS scales to large graphs with hundreds of thousands of nodes and outperforms several models that have been proposed to learn a task-specific graph structure on established benchmarks.

1 INTRODUCTION

Graph representation learning has grown rapidly and found applications in domains where data points define a graph (Chami et al., 2020; Kazemi et al., 2020). Graph neural networks (GNNs) (Scarselli et al., 2008) have been a key component to the success of the research in this area. Following the success of graph convolutional networks (GCNs) (Kipf & Welling, 2017) on semi-supervised node classification, several other GNN variants have been proposed for different prediction tasks on graphs (Hamilton et al., 2017; Veličković et al., 2018; Gilmer et al., 2017; Battaglia et al., 2018) and the power of these models has been studied theoretically (Xu et al., 2019; Sato, 2020).

GNNs take as input a set of node features and an adjacency matrix corresponding to the graph structure, and, for each node, output an embedding that captures not only the initial features of the node but also the features and embeddings of its neighbors. The performance of GNNs highly depends on the quality of the input graph structure and deteriorates substantially when the graph structure is noisy (see Zügner et al., 2018; Dai et al., 2018; Fox & Rajamanickam, 2019). The need for both node features and a clean graph structure impedes the applicability of GNNs to domains where one has access to a set of nodes and their features but not to their underlying graph structure, or only has access to a noisy structure. Examples of such domains include brain signal classification (Jang et al., 2019), computer-aided diagnosis (Cosmo et al., 2020), analysis of computer programs (Johnson et al., 2020), and particle reconstruction (Qasim et al., 2019).

In this paper, we address this limitation by developing a model that learns both the GNN parameters as well as an adjacency matrix simultaneously. Since the number of possible graph structures grows super-exponentially with the number of nodes (Stanley, 1973) and obtaining node labels is typically costly, the number of available labels may not be enough for learning both the GNN parameters and an adjacency matrix–especially for semi-supervised node classification. Our main contribution is to supplement the classification task with a self-supervised task that helps learn a high-quality adjacency matrix. Our self-supervision approach masks some input features (or adds noise to them) and trains a separate GNN aiming at updating the adjacency matrix in such a way that it can recover

the masked (or noisy) features. Introducing this self-supervision adds the inductive bias that a graph structure suitable for predicting the node features is also suitable for predicting the node labels.

We experiment with several classification datasets. For datasets with a graph structure, we only feed the node features to our model. The model operates on the node features and an adjacency that is learned simultaneously from data. We compare our model with different classes of methods: some which do not use the graph structure for predicting labels, some which use a fixed k-Nearest Neighbors (kNN) graph built based on a chosen similarity metric, and some which initialize the graph with kNN but then revise it throughout the training. We show that our model consistently outperforms these methods. We also show that the self-supervised task is key to the high performance of our model. As an additional contribution, we provide an implementation for simultaneous structure and parameter learning that scales to graphs with hundreds of thousands of nodes.

2 Related Work

Existing methods that relate to this work can be grouped into the following categories.

Similarity Graph: One approach for inferring a graph structure is to select a similarity metric and set the edge weight between two nodes to be their similarity (Roweis & Saul, 2000; Tenenbaum et al., 2000). To obtain a sparse structure, one may create a kNN similarity graph, only connect pairs of nodes whose similarity surpasses some predefined threshold, or do sampling. As an example, Gidaris & Komodakis (2019) create a (fixed) kNN graph using the cosine similarity of the node features. Wang et al. (2019b) extend this idea by creating a fresh graph in each layer of the GNN based on the node embedding similarities in that layer as opposed to fixing a graph solely based on the initial features. Instead of choosing a single similarity metric, Halcrow et al. (2020) fuse several (potentially weak) measures of similarity. The quality of the predictions of these methods depends heavily on the choice of the similarity metric(s) and the value of k for the kNN graph, or the threshold on similarity. Furthermore, designing an appropriate similarity metric may not be straightforward in some applications.

Fully-connected Graph: Another approach is to assume a fully-connected graph and employ GNN variants such as graph attention networks (Veličković et al., 2018; Zhang et al., 2018) or the transformer (Vaswani et al., 2017) which infer the graph structure via an attention mechanism, or infer the graph structure using additional information. This approach has been used in computer vision (e.g., Suhail & Sigal, 2019), natural language processing (e.g., Zhu et al., 2019), and few-shot learning (e.g., Garcia & Bruna, 2017), where there are not many nodes. The complexity of this approach, however, grows rapidly making it applicable only to small-sized graphs with a few thousand nodes and not scalable to the datasets we use in our experiments.

Learnable Graph: Instead of computing a similarity graph on the initial features, one may use a graph generator with learnable parameters. Li et al. (2018b) create a fully-connected graph based on a biliear similarity function with learnable parameters. A common approach is to learn to project the nodes to a latent space where node similarities correspond to edge weights. Wu et al. (2018) project the nodes to a latent space by learning weights for each of the input features. Cosmo et al. (2020) and Qasim et al. (2019) use a multi-layer perceptron for projection. Yu et al. (2020) use a GNN that projects the nodes into a latent space using the initial node features as well as an initial graph structure, aiming at providing a revised graph structure to the task-specific GNN. Franceschi et al. (2019) propose a model named *LDS* with a bi-level optimization setup for simultaneously learning the GNN parameters and a full adjacency matrix. Yang et al. (2019) update the input adjacency matrix based on the inductive bias that nodes belonging to the same class should be connected to each other and nodes belonging to different classes should be disconnected. Chen et al. (2020) propose an iterative approach that iterates over projecting the nodes to a latent space and constructing an adjacency matrix from the latent representations multiple times. In our experiments, we compare with several approaches from this category.

Leveraging Domain Knowledge: In applications where specific domain knowledge is available, one may leverage this to guide the model toward learning specific structures. For example, Johnson et al. (2020) leverage abstract syntax trees and regular languages in learning graph structures of Python programs that aid reasoning for downstream tasks. Jin et al. (2020b) train GNNs that are robust to adversarial attack by learning a cleaned version of the input poisoned adjacency matrix

using the domain knowledge that clean adjacency matrices are often sparse and low-rank and exhibit feature smoothness along connected nodes.

Proposed Method: Our model falls within the learnable graph category in which we use graph generators with learnable parameters to infer the adjacency matrix. We supplement the training with a self-supervised objective to increase the amount of supervision in learning a graph structure. The self-supervised objective is generic and can be combined with many of the models described above. It is inspired by the successful training procedures of several recent language models such as BERT (Devlin et al., 2018) and RoBERTa (Liu et al., 2019b). Similar self-supervision techniques have also been employed for GNNs (Hu et al., 2020b;c) (Jin et al., 2020a; You et al., 2020; Zhu et al., 2020). While we employ similar self-supervision techniques, our work differs from this line of work as we use self-supervision for learning a graph structure whereas the above methods use it to learn better (and, in some cases, transferable) GNN parameters. Our self-supervised task is also in the same vein as the auxiliary tasks in the context of multi-task learning used in computer vision, natural language processing, and reinforcement learning (see, e.g., Jaderberg et al., 2016; Liebel & Körner, 2018; Alonso & Plank, 2016).

3 Background and Notation

We use lowercase letters to denote scalars, bold lowercase letters to denote vectors and bold uppercase letters to denote matrices. $\boldsymbol{I}$ represents an identity matrix. For a vector $\boldsymbol{v}$, we represent its ith element as $\boldsymbol{v}_i$. For a matrix $\boldsymbol{M}$, we represent the ith row as $\boldsymbol{M}_i$ and the element at the ith row and j^{th} column as $\boldsymbol{M}_{ij}$. For an attributed graph, we use n, m and f to represent the number of nodes, edges, and features respectively, and denote the graph as $\mathcal{G} = \{\mathcal{V}, \boldsymbol{A}, \boldsymbol{X}\}$ where $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a set of nodes, $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is a (sparse) adjacency matrix with $\boldsymbol{A}_{ij}$ indicating the weight of the edge from v_i to v_j ($\boldsymbol{A}_{ij} = 0$ implies no edge), and $\boldsymbol{X} \in \mathbb{R}^{n \times f}$ is a matrix whose rows correspond to node features or attributes. A degree matrix $\boldsymbol{D}$ for a graph $\mathcal{G}$ is a diagonal matrix where $\boldsymbol{D}_{ii} = \sum_j \boldsymbol{A}_{ij}$.

Graph convolutional networks (GCNs) are a powerful variant of GNNs. For a graph $\mathcal{G} = \{\mathcal{V}, \boldsymbol{A}, \boldsymbol{X}\}$ with a degree matrix $\boldsymbol{D}$, layer l of the GCN architecture can be defined as $\boldsymbol{H}^{(l)} = \sigma(\hat{\boldsymbol{A}}\boldsymbol{H}^{(l-1)}\boldsymbol{W}^{(l)})$ where $\hat{\boldsymbol{A}}$ represents a normalized adjacency matrix, $\boldsymbol{H}^{(l-1)} \in \mathbb{R}^{n \times d_{l-1}}$ represents the node representations in layer *l-1* with $\boldsymbol{H}^{(0)} = \boldsymbol{X}$, $\boldsymbol{W}^{(l)} \in \mathbb{R}^{d_{l-1} \times d_l}$ is a weight matrix, σ is an activation function such as ReLU, and $\boldsymbol{H}^{(l)} \in \mathbb{R}^{n \times d_l}$ is the updated node embeddings. For undirected graphs where the adjacency is symmetric, $\hat{\boldsymbol{A}} = \boldsymbol{D}^{-\frac{1}{2}}(\boldsymbol{A} + \boldsymbol{I})\boldsymbol{D}^{-\frac{1}{2}}$ corresponds to a row-and-column normalized adjacency with self-loops, and for directed graphs where the adjacency is not necessarily symmetric, $\hat{\boldsymbol{A}} = \boldsymbol{D}^{-1}(\boldsymbol{A} + \boldsymbol{I})$ corresponds to a row normalized adjacency matrix with self-loops.

4 SLAPS: Simultaneous Learning of Adjacency matrix and GNN Parameters with Self-supervision

We break our model into four components: 1) generator, 2) adjacency processor, 3) classifier, and 4) self-supervision. The generator takes the node features as input and generates a (perhaps sparse, non-normalized, and non-symmetric) matrix $\tilde{\boldsymbol{A}} \in \mathbb{R}^{n \times n}$. $\tilde{\boldsymbol{A}}$ is then fed into the adjacency processor which outputs $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ corresponding to a normalized, and in some cases symmetric version of $\tilde{\boldsymbol{A}}$. The classifier is a GNN that receives $\boldsymbol{A}$ as well as the node features as input and classifies the nodes into a set of predefined classes. The self-supervision component is a GNN that receives noisy features and the generated adjacency as input and aims at denoising the features. Figure 1 illustrates the different components and in what follows, we describe each component in more detail.

4.1 Generator

The generator is a function $\mathsf{G} : \mathbb{R}^{n \times f} \to \mathbb{R}^{n \times n}$ with parameters θ_{G} which takes the node features $\boldsymbol{X}$ as input and produces $\tilde{\boldsymbol{A}} \in \mathbb{R}^{n \times n}$ as output. We consider the following two generators.

Full Parameterization (FP): In this case, θ_{G} is a single matrix in $\mathbb{R}^{n \times n}$ and the generator function is defined as $\tilde{\boldsymbol{A}} = \mathsf{G}_{FP}(\boldsymbol{X}; \theta_{\mathsf{G}}) = \theta_{\mathsf{G}}$. That is, the generator ignores the input node features and directly optimizes the adjacency matrix. The disadvantages of this generator include adding n^2

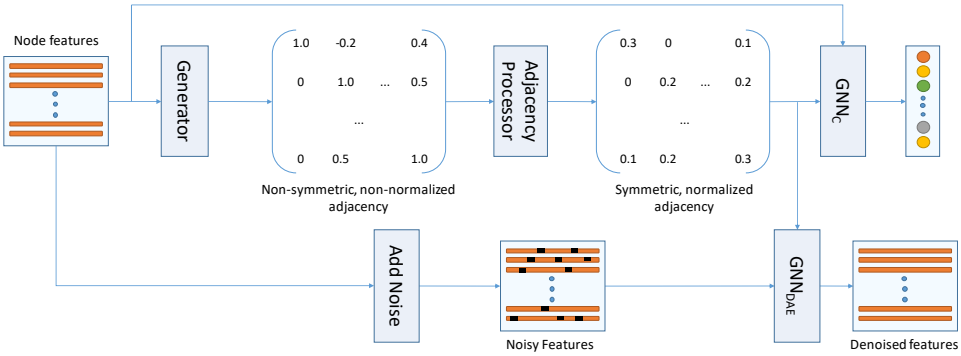

Figure 1: Overview of SLAPS. At the top, a generator receives the node features and produces a non-symmetric, non-normalized adjacency having (potentially) both positive and negative values (Section 4.1). The adjacency processor makes the values positive, symmetrizes and normalizes the adjacency (Section 4.2). The resulting adjacency and the node features go into GNN_C which predicts the node classes (Section 4.3). At the bottom, some noise is added to the node features. The resulting noisy features and the generated adjacency go into $\mathsf{GNN}_\mathsf{DAE}$ which then denoises the features (Section 4.4).

parameters to the model, which limits scalability and makes the model susceptible to overfitting, and not being applicable to inductive settings where during test time predictions are to be made for nodes unseen during training. This generator is similar to the one proposed by Franceschi et al. (2019) except that they treat each element of $\tilde{\boldsymbol{A}}$ as the parameter of a Bernoulli distribution and sample graph structures from these Bernoulli distributions.

MLP-kNN: In this case, θ_G corresponds to the weights of a multi-layer perceptron (MLP) and $\tilde{\boldsymbol{A}} = \mathsf{G}_\mathsf{MLP}(\boldsymbol{X}; \theta_\mathsf{G}) = \mathsf{kNN}(\mathsf{MLP}(\boldsymbol{X}))$, where $\mathsf{MLP} : \mathbb{R}^{n \times f} \to \mathbb{R}^{n \times f'}$ is an MLP that produces a matrix with updated node representations $\boldsymbol{X}'$; $\mathsf{kNN} : \mathbb{R}^{n \times f'} \to \mathbb{R}^{n \times n}$ produces a sparse matrix. Let $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ with $\boldsymbol{M}_{ij} = 1$ if v_j is among the top k similar nodes to v_i and 0 otherwise, and let $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ such that $\boldsymbol{S}_{ij} = \mathsf{Sim}(\boldsymbol{X}'_i, \boldsymbol{X}'_j)$ for some differentiable similarity function Sim (we used cosine in our experiments). Then $\tilde{\boldsymbol{A}} = \mathsf{kNN}(\boldsymbol{X}') = \boldsymbol{M} \odot \boldsymbol{S}$ where $\odot$ represents the Hadamard (element-wise) product. Since $\boldsymbol{S}$ is computed based on $\boldsymbol{X}'$, the gradients flow to the elements in $\boldsymbol{X}'$ (and consequently to the weights of the MLP) through $\boldsymbol{S}$. In the backward phase of our model, we compute the gradients only with respect to those elements in $\boldsymbol{S}$ whose corresponding value in $\boldsymbol{M}$ is 1 (i.e. those elements $\boldsymbol{S}_{ij}$ such that $\boldsymbol{M}_{ij} = 1$); the gradient with respect to the other elements is 0. With this formulation, in the forward phase of the network, one can first compute the matrix $\boldsymbol{M}$ using an off-the-shelf k-nearest neighbors algorithm and then compute the similarities in $\boldsymbol{S}$ only for pairs of nodes where $\boldsymbol{M}_{ij} = 1$. Unlike FP, this generator can be used for the inductive setting.

Smart initialization: In our experiments, we found the initialization of the generator parameters (i.e. θ_G) to be important. Let $\boldsymbol{A}^{kNN}$ represent an adjacency matrix created by applying a kNN function on the initial node features. One smart initialization for θ_G is to initialize them in a way that the generator generates $\boldsymbol{A}^{kNN}$ before training starts (i.e. $\tilde{\boldsymbol{A}} = \boldsymbol{A}^{kNN}$ before training starts). Such an initialization can be trivially done for the FP generator by initializing θ_G to $\boldsymbol{A}^{kNN}$, but may not be straightforward for the MLP-kNN generator. To enable initializing the parameters of the MLP-kNN generator in a way that it generates $\boldsymbol{A}^{kNN}$ before training starts, we consider two variants of this generator. In one, hereafter referred to simply as MLP, we keep the input dimension the same throughout the layers. In the other, hereafter referred to as MLP-D, we consider MLPs with diagonal weight matrices (i.e., except the main diagonal, all other parameters in the weight matrices are zero). For both variants, we initialize the weight matrices in θ_G with the identity matrix to ensure that the output of the MLP is initially the same as its input and the kNN graph created on these outputs is equivalent to $\boldsymbol{A}^{kNN}$. MLP-D can be thought of as assigning different weights to different features and then computing node similarities. Note that, alternatively, one may use other MLP variants but pre-train the weights to output $\boldsymbol{A}^{kNN}$ before the main training starts.

4.2 ADJACENCY PROCESSOR

The output $\tilde{\boldsymbol{A}}$ of the generator may have both positive and negative values, may be non-symmetric and non-normalized. To ensure all values of the adjacency are positive and make the adjacency symmetric and normalized, we apply the following function to $\tilde{\boldsymbol{A}}$:

$$\boldsymbol{A} = \boldsymbol{D}^{-\frac{1}{2}}\Big(\frac{\mathsf{P}(\tilde{\boldsymbol{A}}) + \mathsf{P}(\tilde{\boldsymbol{A}})^T}{2}\Big)\boldsymbol{D}^{-\frac{1}{2}} \tag{1}$$

Here P is a function with a non-negative range. In our experiments, when using an MLP generator, we apply the ReLU function to the elements of $\tilde{\boldsymbol{A}}$. When using the fully-parameterized (FP) generator, applying ReLU results in a gradient flow problem as any edge whose corresponding value in $\tilde{\boldsymbol{A}}$ becomes less than or equal to zero stops receiving gradient updates. For this reason, for FP we apply the ELU function to the elements of $\tilde{\boldsymbol{A}}$ and then add a value of 1. The sub-expression $\frac{\mathsf{P}(\tilde{\boldsymbol{A}})+\mathsf{P}(\tilde{\boldsymbol{A}})^T}{2}$ makes the resulting matrix $\mathsf{P}(\tilde{\boldsymbol{A}})$ symmetric. To understand the reason for taking the mean of $\mathsf{P}(\tilde{\boldsymbol{A}})$ and $\mathsf{P}(\tilde{\boldsymbol{A}})^T$, assume $\tilde{\boldsymbol{A}}$ is generated by $\mathsf{G}_{\mathsf{MLP}}$. If v_j is among the k most similar nodes to v_i and vice versa, then the strength of the connection between v_i and v_j will remain the same. However, if, say, v_j is among the k most similar nodes to v_i but v_i is not among the top k for v_j, then taking the average of the similarities reduces the strength of the connection between v_i and v_j. Finally, once we have a symmetric adjacency with non-negative values, we compute the degree matrix $\boldsymbol{D}$ for $\frac{\mathsf{P}(\tilde{\boldsymbol{A}})+\mathsf{P}(\tilde{\boldsymbol{A}})^T}{2}$ and normalize $\frac{\mathsf{P}(\tilde{\boldsymbol{A}})+\mathsf{P}(\tilde{\boldsymbol{A}})^T}{2}$ by multiplying it left and right with $\boldsymbol{D}^{-\frac{1}{2}}$.

4.3 CLASSIFIER

The classifier is a function $\mathsf{GNN}_{\mathsf{C}} : \mathbb{R}^{n\times f} \times \mathbb{R}^{n\times n} \rightarrow \mathbb{R}^{n\times|\mathcal{C}|}$ with parameters $\theta_{\mathsf{GNN}_{\mathsf{C}}}$. It takes the node features $\boldsymbol{X}$ and the generated adjacency $\boldsymbol{A}$ as input and provides for each node the logits for each class. $\mathcal{C}$ corresponds to the classes and $|\mathcal{C}|$ corresponds to the number of classes. We use a two-layer GCN for which $\theta_{\mathsf{GNN}_{\mathsf{C}}} = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}\}$ and define our classifier as $\mathsf{GNN}_{\mathsf{C}}(\boldsymbol{A}, \boldsymbol{X}; \theta_{\mathsf{GNN}_{\mathsf{C}}}) = \boldsymbol{A}\mathsf{ReLU}(\boldsymbol{A}\boldsymbol{X}\boldsymbol{W}^{(1)})\boldsymbol{W}^{(2)}$ but other GNN variants can be used as well (recall that $\boldsymbol{A}$ is normalized). The training loss $\mathcal{L}_C$ for the classification task is computed by taking the softmax of the logits to produce a probability distribution for each node and then computing the cross-entropy loss.

4.4 ADDING SELF-SUPERVISION

As explained in Section 1, in many domains, the number of labeled nodes may be insufficient for learning both the structure and the GNN parameters from data. To increase the amount of supervision for learning the structure, we propose a self-supervised approach based on denoising autoencoders (Vincent et al., 2008). Let $\mathsf{GNN}_{\mathsf{DAE}} : \mathbb{R}^{n\times f} \times \mathbb{R}^{n\times n} \rightarrow \mathbb{R}^{n\times f}$ be a GNN that takes node features as well as a normalized adjacency produced by a generator as input and provides updated node features with the same dimension as input. We train $\mathsf{GNN}_{\mathsf{DAE}}$ such that it receives a noisy version $\tilde{\boldsymbol{X}}$ of the features $\boldsymbol{X}$ as input and produces the denoised features $\boldsymbol{X}$ as output. Let *idx* represent the indices corresponding to the elements of $\boldsymbol{X}$ to which we have added noise, and $\boldsymbol{X}_{idx}$ represent the values at these indices. The aim of the training procedure is to minimize:

$$\mathcal{L}_{DAE} = \mathsf{L}(\boldsymbol{X}_{idx}, \mathsf{GNN}_{\mathsf{DAE}}(\tilde{\boldsymbol{X}}, \boldsymbol{A}; \theta_{\mathsf{GNN}_{\mathsf{DAE}}})_{idx}) \tag{2}$$

where $\boldsymbol{A}$ is the generated adjacency matrix and L is a loss function. To add noise to the input features for datasets where features consist of binary vectors, in each iteration, idx consists of r percent of the indices of $\boldsymbol{X}$ whose values are ones and $r\eta$ percent of the indices whose values are zeros, both selected uniformly at random in each epoch. Both r and η (corresponding to the negative ratio) are hyperparameters. In this case, we add noise by setting the ones in the selected mask to zeros and L is the binary cross-entropy loss. For datasets where the input features are continuous numbers, idx consists of r percent of the indices of $\boldsymbol{X}$ selected uniformly at random in each epoch. We add noise by either replacing the values at idx with zeros or by adding independent Gaussian noises to each of the features. In this case, L is the mean-squared error loss.

To understand the cruciality of the proposed self-supervision, let us consider a scenario for training the model in Figure 1 (or a model with a similar architecture) but without the self-supervised task. As training proceeds, assume that two unlabeled nodes v_i

Table 1: Results of SLAPS and the baselines on established node classification benchmarks. † indicates results have been taken from Franceschi et al. (2019). Bold and underlined values indicate best and second-best mean performances respectively. OOM indicates out of memory.

Model	Generator	Cora	Citeseer	Cora390	Citeseer370	Pubmed	ogbn-arxiv
MLP		$56.1 \pm 1.6^{\dagger}$	$56.7 \pm 1.7^{\dagger}$	65.8 ± 0.4	67.1 ± 0.5	71.4 ± 0.0	$\underline{54.7 \pm 0.1}$
kNN-GCN		$66.5 \pm 0.4^{\dagger}$	$68.3 \pm 1.3^{\dagger}$	72.5 ± 0.5	71.8 ± 0.8	70.4 ± 0.4	49.1 ± 0.3
LDS		-	-	$71.5 \pm 0.8^{\dagger}$	$71.5 \pm 1.1^{\dagger}$	OOM	OOM
GRCN		67.4 ± 0.3	67.3 ± 0.8	71.3 ± 0.9	70.9 ± 0.7	67.3 ± 0.3	OOM
DGCNN		56.5 ± 1.2	55.1 ± 1.4	67.3 ± 0.7	66.6 ± 0.8	70.1 ± 1.3	OOM
IDGL		70.9 ± 0.6	68.2 ± 0.6	73.4 ± 0.5	72.7 ± 0.4	72.3 ± 0.4	OOM
SLAPS	FP	72.4 ± 0.4	$\underline{70.7 \pm 0.4}$	$\mathbf{76.6 \pm 0.4}$	$\underline{73.1 \pm 0.6}$	OOM	OOM
SLAPS	MLP	$\underline{72.8 \pm 0.8}$	70.5 ± 1.1	$\underline{75.3 \pm 1.0}$	73.0 ± 0.9	$\mathbf{74.4 \pm 0.6}$	$\mathbf{56.6 \pm 0.1}$
SLAPS	MLP-D	$\mathbf{73.4 \pm 0.3}$	$\mathbf{72.6 \pm 0.6}$	75.1 ± 0.5	$\mathbf{73.9 \pm 0.4}$	$\underline{73.1 \pm 0.7}$	52.9 ± 0.1

and v_j are not directly connected to any labeled nodes. Then, since a two-layer GCN makes predictions for the nodes based on their two-hop neighbors, the edge between v_i and v_j receives no supervision[1]. Figure 2 provides an example of such a scenario. As a quantitative example, in the original structures and train/validation/test splits of Cora and Citeseer, 80.4% and 89.9% of the nodes, and consequently 64.6% and 80.8% of pairs of nodes, are not connected to any labeled/train nodes. If no supervision is provided for some edges in the graph, after training the existence and weights of these edges may end up being set randomly (or the same as their initialization value), which may be problematic during testing. With the self-supervised task, however, although these edges may not receive supervision from the main task (i.e. from $\mathsf{GCN_C}$), the supervision provided by the self-supervised task (i.e. from $\mathsf{GCN_{DAE}}$) helps learn an appropriate weight for them.

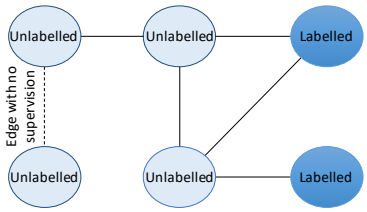

Figure 2: The dashed edge receives no supervision when training a two-layer GCN as it is not in the two-hop neighborhood of any labeled node.

4.5 SLAPS

Our final model, dubbed *SLAPS*, is trained to minimize $\mathcal{L} = \mathcal{L}_C + \lambda \mathcal{L}_{DAE}$ where $\mathcal{L}_C$ is the classification loss, $\mathcal{L}_{DAE}$ is the denoising autoencoder loss (see Equation 2), and λ is a hyperparameter controlling the relative importance of the two losses.

To verify the merit of the $\mathsf{GNN_{DAE}}$ for learning an adjacency matrix in isolation, we also consider a variant of SLAPS named $SLAPS_{2s}$ that is trained in two stages. We first train the $\mathsf{GNN_{DAE}}$ model by minimizing the loss function described in Equation 2. Note that the loss function in Equation 2 depends on the parameters θ_{G} of the generator and the parameters $\theta_{\mathsf{GNN_{DAE}}}$ of the denoising autoencoder. After every t epochs of training, we fix the adjacency matrix, train a classifier with the fixed adjacency matrix, and measure classification accuracy on the validation set. We select the epoch that produces the adjacency providing the best validation accuracy for the classifier. Note that in $SLAPS_{2s}$, the adjacency matrix is trained only based on $\mathsf{GNN_{DAE}}$.

5 Experiments

Baselines: We compare our proposal to several baselines with different properties. The first baseline is a multi-layer perceptron (MLP) which does not take the graph structure into account. Similar to Franceschi et al. (2019), we also consider a baseline named *kNN-GCN* where we create a kNN graph based on the node features and feed this graph to a GCN. The graph structure remains fixed in this

[1]While this problem may be alleviated to some extent by increasing the number of layers of the GCN, deeper GCNs typically provide inferior results due to issues such as oversmoothing (see, e.g., Li et al., 2018a; Oono & Suzuki, 2020).

Table 2: Results on classification datasets. † indicates results have been taken from Franceschi et al. (2019). Bold and underlined values indicate best and second-best mean performances respectively.

Model	Generator	Wine	Cancer	Digits	20news
LDS		$\mathbf{97.3 \pm 0.4}^{\dagger}$	$94.4 \pm 1.9^{\dagger}$	$92.5 \pm 0.7^{\dagger}$	$46.4 \pm 1.6^{\dagger}$
$IDGL$		$\underline{97.0 \pm 0.7}$	94.2 ± 2.3	92.5 ± 1.3	48.5 ± 0.6
$SLAPS$	FP	96.6 ± 0.4	94.6 ± 0.3	$\mathbf{94.4 \pm 0.7}$	44.4 ± 0.8
$SLAPS$	MLP	96.3 ± 1.0	$\underline{96.0 \pm 0.8}$	92.4 ± 0.6	$\mathbf{50.4 \pm 0.7}$
$SLAPS$	MLP-D	96.5 ± 0.8	$\mathbf{96.6 \pm 0.2}$	$\underline{93.2 \pm 0.6}$	$\underline{49.8 \pm 0.9}$

approach. We also compare with baselines that learn the graph structure from data including LDS (Franceschi et al., 2019), GRCN (Yu et al., 2020), DGCNN (Wang et al., 2019b), and IDGL (Chen et al., 2020). We feed a kNN graph to the models requiring an initial graph structure.

Datasets: We use three established benchmarks in the GNN literature namely Cora, Citeseer, and Pubmed (Sen et al., 2008) as well as a newly released dataset for node classification named *ogbn-arxiv* (Hu et al., 2020a) that is orders of magnitude larger than the other three datasets and is more challenging due to the more realistic split of the data into train, validation, and test sets. For all the datasets, we only feed the node features to the models and not the graph structure. Following Franceschi et al. (2019), we also experiment with several classification datasets available in scikit-learn (Pedregosa et al., 2011) including Wine, Cancer, Digits, and 20News. Dataset statistics can be found in the Appendix. For Cora and Citeseer, the LDS model changes the original train and validation splits by using half of the validation set during the training of the bi-level optimization setup and the other half for early stopping and hyperparameter selection. This increases the number of labels used during training from 140 to 390 for Cora and from 120 to 370 for Citeseer. To provide a fair comparison to LDS, we also report our results on the Cora and Citeseer datasets with these modified training and validation sets. We name the modified versions Cora390 and Citeseer370 respectively. We also follow a similar procedure for the scikit-learn datasets.

5.1 Comparative Results

The results of SLAPS and the baselines on the node classification benchmarks are reported in Table 1. Considering only the baselines first, we see that kNN-GCN significantly outperforms MLP on Cora and Citeseer but underperforms on Pubmed and ogbn-arxiv. This shows the importance of the similarity metric and the graph structure that is fed into GCN as a low-quality structure can harm model performance. LDS outperforms MLP but achieves slightly worse results than kNN-GCN under a fair evaluation setting. Moreover, the fully parameterized adjacency matrix of LDS results in memory issues for Pubmed and ogbn-arxiv. As for GRCN, it was shown in the original paper that GRCN can revise a good initial adjacency matrix and provide a substantial boost in performance. However, as evidenced by the results, if the initial graph structure is somewhat poor, GRCN's performance becomes on-par with kNN-GCN. IDGL is the best performing baseline but the iterative nature of it makes it slow to train and test.

SLAPS consistently outperforms the baselines on all datasets, in some cases by large margins. Among the generators, the winner is dataset-dependent with MLP-D mostly outperforming MLP on datasets with many features and MLP outperforming on datasets with small numbers of features. Using the software that was publicly released by the authors, all baselines that learn a graph structure fail on ogbn-arxiv and our implementation is the first that generalizes to such large graphs.

Table 2 reports the results for the scikit-learn datasets and compares with LDS and IDGL. On three out of four datasets, SLAPS outperforms the other two baselines. Among the datasets on which we can train SLAPS with the FP generator, 20news has the largest number of nodes. On this dataset, we observed that an FP generator suffers from overfitting and produces weaker results compared to other generators due to its large number of parameters.

5.2 Further Analysis

SLAPS$_{2s}$: To provide more insight into the value provided by the self-supervision task on the learned adjacency, we conduct experiments with SLAPS$_{2s}$. Recall from Section 4.5 that in SLAPS$_{2s}$, the adjacency is learned only based on the self-supervision task and the node labels are

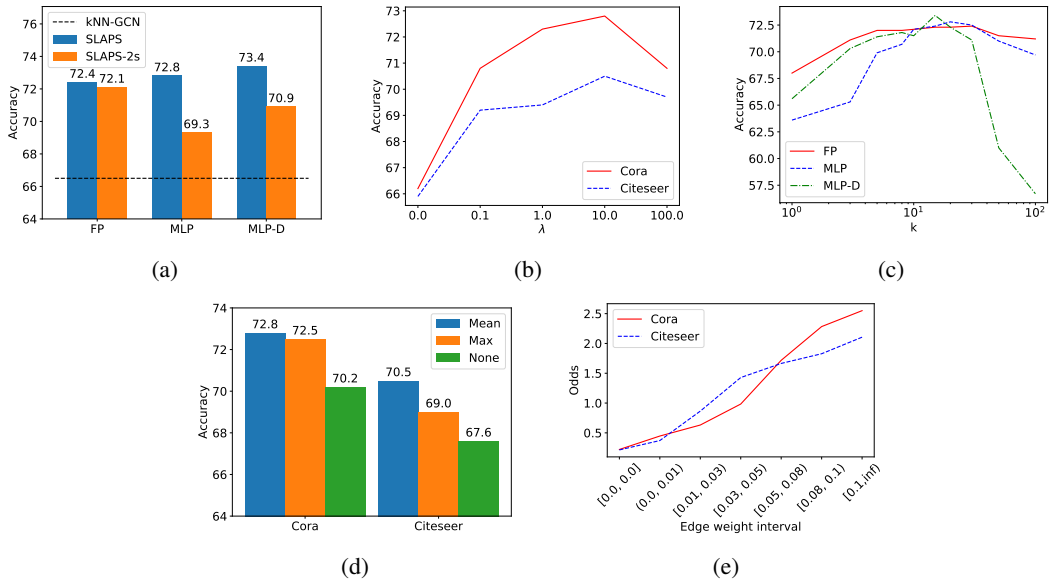

Figure 3: The performance of SLAPS (a) compared to SLAPS_{2s} on Cora with different generators, (b) with MLP graph generator on Cora and Citeseer as a function of λ, (c) with different graph generators on Cora as a function of k in kNN, and (d) on Cora and Citeseer with different adjacency symmetrizations. (e) The odds of two nodes in the test set sharing the same label as a function of the edge weights learned by SLAPS.

only used for early stopping, hyperparameter tuning, and training GCN_C. Figure 3(a) shows the performance of SLAPS and SLAPS_{2s} on Cora and compares them with kNN-GCN. Although SLAPS_{2s} does not use the node labels in learning an adjacency matrix, it outperforms kNN-GCN (8.4% improvement when using an FP generator). With an FP generator, SLAPS_{2s} even achieves competitive performance with SLAPS; this is mainly because FP does not leverage the supervision provided by GCN_C toward learning generalizable patterns that can be used for nodes other than those in the training set. These results corroborate the effectiveness of the self-supervision task for learning an adjacency matrix. Besides, the results show that learning the adjacency using both self-supervision and the task-specific node labels results in higher predictive accuracy.

The value of λ: Figure 3(b) shows the performance of SLAPS[2] on Cora and Citeseer with different values of λ. When $\lambda = 0$, corresponding to removing self-supervision, the model performance is somewhat poor. As soon as λ becomes positive, both models see a large boost in performance showing that self-supervision is crucial to the high performance of SLAPS. Increasing λ further provides larger boosts until it becomes so large that the self-supervision loss dominates the classification loss and the performance deteriorates. Note that with $\lambda = 0$, SLAPS with the MLP generator becomes a variant of the model proposed by Cosmo et al. (2020), but with a different similarity function.

Importance of k in kNN: Figure 3(c) shows the performance of SLAPS on Cora for three graph generators as a function of k in kNN. For all three cases, the value of k plays a major role in model performance. The FP generator is the least sensitive because in FP, k only affects the initialization of the adjacency matrix but then the model can change the number of neighbors of each node. For MLP and MLP-D, however, the number of neighbors of each node remains close to k (but not necessarily equal as the adjacency processor can add or remove some edges) and the two generators become more sensitive to k. For larger values of k, the extra flexibility of the MLP generator enables removing some of the unwanted edges through the function P or reducing the weights of the unwanted edges resulting in MLP being less sensitive to large values of k compared to MLP-D.

[2]The generator used in this experiment is MLP; other generators produced similar results.

Symmetrization: To symmetrize the adjacency, in Equation 1 we took the average of $\mathsf{P}(\tilde{\boldsymbol{A}})$ and $\mathsf{P}(\tilde{\boldsymbol{A}})^T$. Here we also consider two other choices: 1) $\mathsf{max}(\mathsf{P}(\tilde{\boldsymbol{A}}), \mathsf{P}(\tilde{\boldsymbol{A}})^T)$, and 2) not symmetrizing the adjacency (i.e. using $\mathsf{P}(\tilde{\boldsymbol{A}})$). Figure 3(d) compares these three choices on Cora and Citeseer with an MLP generator (other generators produced similar results). On both datasets, symmetrizing the adjacency provides a performance boost. Compared to mean symmetrization, max symmetrization performs slightly worse. This may be because max symmetrization does not distinguish between the case where both v_i and v_j are among the k most similar nodes of each other and the case where only one of them is among the k most similar nodes of the other.

Analysing the learned adjacency: Many graph-based semi-supervised classification models are based on the *cluster assumption* according to which nearby nodes are more likely to share the same label (Chapelle & Zien, 2005). To verify the quality of the adjacency matrix learned using SLAPS, for every pair of nodes in the test set, we compute the odds of the two nodes sharing the same label as a function of the normalized weight of the edge connecting them. Figure 3(e) represents the odds for different weight intervals. For both Cora and Citeseer, nodes connected with higher edge weights are more likely to share the same label compared to nodes with lower or zero edge weights. As a specific example, when $\boldsymbol{A}_{ij} \geq 0.1$, v_i and v_j are almost 2.5 times more likely to share the same label on Cora and almost 2.0 times more likely on Citeseer. Note that SLAPS may connect nodes based on a different criterion than the one used in the original datasets and so the learned adjacencies do not necessarily resemble the original structures.

6 Conclusion

In this paper, we proposed a model for learning the parameters of a graph neural network and the graph structure of the nodes simultaneously. We showed the effectiveness of our model using a comprehensive set of experiments and analyses. In the future, we would like to try more sophisticated graph generation models (e.g., GraphRNN (You et al., 2018) and GNF (Liu et al., 2019a)) and extend our approach to applications with a temporal aspect where node features are observed over time but their connections are not provided as input.

References

Héctor Martínez Alonso and Barbara Plank. When is multitask learning effective? semantic sequence prediction under varying data conditions. *arXiv preprint arXiv:1612.02251*, 2016.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *arXiv preprint arXiv:2005.03675*, 2020.

Olivier Chapelle and Alexander Zien. Semi-supervised classification by low density separation. In *AISTATS*, volume 2005, pp. 57–64. Citeseer, 2005.

Yu Chen, Lingfei Wu, and Mohammed J. Zaki. Deep iterative and adaptive learning for graph neural networks. In *The First International Workshop on Deep Learning on Graphs: Methodologies and Applications (with AAAI)*, February 2020. URL `https://dlg2019.bitbucket.io/aaai20`.

Luca Cosmo, Anees Kazi, Seyed-Ahmad Ahmadi, Nassir Navab, and Michael Bronstein. Latent patient network learning for automatic diagnosis. *arXiv preprint arXiv:2003.13620*, 2020.

Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial attack on graph structured data. *arXiv preprint arXiv:1806.02371*, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

James Fox and Sivasankaran Rajamanickam. How robust are graph neural networks to structural noise? *arXiv preprint arXiv:1912.10206*, 2019.

Luca Franceschi, Mathias Niepert, Massimiliano Pontil, and Xiao He. Learning discrete structures for graph neural networks. In *ICML*, 2019.

Victor Garcia and Joan Bruna. Few-shot learning with graph neural networks. *arXiv preprint arXiv:1711.04043*, 2017.

Spyros Gidaris and Nikos Komodakis. Generating classification weights with gnn denoising autoencoders for few-shot learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 21–30, 2019.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1263–1272. JMLR. org, 2017.

Jonathan Halcrow, Alexandru Mosoi, Sam Ruth, and Bryan Perozzi. Grale: Designing networks for graph learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2523–2532, 2020.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020a.

Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *ICLR*, 2020b.

Ziniu Hu, Yuxiao Dong, Kuansan Wang, Kai-Wei Chang, and Yizhou Sun. Gpt-gnn: Generative pre-training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1857–1867, 2020c.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

Soobeom Jang, Seong-Eun Moon, and Jong-Seok Lee. Brain signal classification via learning connectivity structure. *arXiv preprint arXiv:1905.11678*, 2019.

Wei Jin, Tyler Derr, Haochen Liu, Yiqi Wang, Suhang Wang, Zitao Liu, and Jiliang Tang. Self-supervised learning on graphs: Deep insights and new direction. *arXiv preprint arXiv:2006.10141*, 2020a.

Wei Jin, Yao Ma, Xiaorui Liu, Xianfeng Tang, Suhang Wang, and Jiliang Tang. Graph structure learning for robust graph neural networks. *arXiv preprint arXiv:2005.10203*, 2020b.

Daniel D Johnson, Hugo Larochelle, and Daniel Tarlow. Learning graph structure with a finite-state automaton layer. *arXiv preprint arXiv:2007.04929*, 2020.

Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *JMLR*, 2020.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI*, 2018a.

Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. Adaptive graph convolutional neural networks. *arXiv preprint arXiv:1801.03226*, 2018b.

Lukas Liebel and Marco Körner. Auxiliary tasks in multi-task learning. *arXiv preprint arXiv:1805.06334*, 2018.

Jenny Liu, Aviral Kumar, Jimmy Ba, Jamie Kiros, and Kevin Swersky. Graph normalizing flows. In *NeurIPS*, pp. 13556–13566, 2019a.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019b.

Kenta Oono and Taiji Suzuki. Graph neural networks exponentially lose expressive power for node classification. In *ICLR*, 2020.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

Shah Rukh Qasim, Jan Kieseler, Yutaro Iiyama, and Maurizio Pierini. Learning representations of irregular particle-detector geometry with distance-weighted graph networks. *The European Physical Journal C*, 79(7):1–11, 2019.

Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

Ryoma Sato. A survey on the expressive power of graph neural networks. *arXiv preprint arXiv:2003.04078*, 2020.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

Richard P Stanley. Acyclic orientations of graphs. *Discrete Mathematics*, 5(2):171–178, 1973.

Mohammed Suhail and Leonid Sigal. Mixture-kernel graph attention network for situation recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 10363–10372, 2019.

Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *ICLR*, 2018.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pp. 1096–1103, 2008.

Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019a.

Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)*, 38(5): 1–12, 2019b.

Xuan Wu, Lingxiao Zhao, and Leman Akoglu. A quest for structure: Jointly learning the graph structure and semi-supervised classification. In *Proceedings of the 27th ACM international conference on information and knowledge management*, pp. 87–96, 2018.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

Liang Yang, Zesheng Kang, Xiaochun Cao, Di Jin, Bo Yang, and Yuanfang Guo. Topology optimization based graph convolutional network. In *IJCAI*, pp. 4054–4061, 2019.

Jiaxuan You, Rex Ying, Xiang Ren, William L Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. *arXiv preprint arXiv:1802.08773*, 2018.

Yuning You, Tianlong Chen, Zhangyang Wang, and Yang Shen. When does self-supervision help graph convolutional networks? *arXiv preprint arXiv:2006.09136*, 2020.

Donghan Yu, Ruohong Zhang, Zhengbao Jiang, Yuexin Wu, and Yiming Yang. Graph-revised convolutional network. In *ECML PKDD*, 2020.

Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.

Hao Zhu, Yankai Lin, Zhiyuan Liu, Jie Fu, Tat-seng Chua, and Maosong Sun. Graph neural networks with generated parameters for relation extraction. *arXiv preprint arXiv:1902.00756*, 2019.

Qikui Zhu, Bo Du, and Pingkun Yan. Self-supervised training of graph convolutional networks. *arXiv preprint arXiv:2006.02380*, 2020.

Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2847–2856, 2018.

A Appendix

Implementation Detail: We implemented our model in PyTorch (Paszke et al., 2017), used deep graph library (DGL) (Wang et al., 2019a) for the sparse operations, and used Adam (Kingma & Ba, 2014) as the optimizer. We performed early stopping and hyperparameter tuning based on the accuracy on the validation set for all datasets except Wine and Cancer. For these two datasets, validation accuracy reached 100 percent with many hyperparameter settings, making it difficult to select the best set of hyperparameters so instead, we used the validation cross-entropy loss. We fixed the maximum number of epochs to 2000. We use two-layer GCNs for both GNN_C and $\mathsf{GNN}_\mathsf{DAE}$ as well as for baselines and two-layer MLPs throughout the paper (for experiments on ogbn-arxiv, although the original paper uses models with three layers and with batch normalization after each layer, to be consistent with our other experiments we used two layers and removed the normalization). We used two learning rates, one for GCN_C and one for the other parameters of the models. We tuned the two learning rates from the set $\{0.01, 0.001\}$. We added dropout layers with dropout probabilities of 0.5 after the first layer of the GNNs. We also added dropout to the adjacency matrix and tuned the values from the set $\{0.25, 0.5\}$. We set the hidden dimension of GNN_C to 32 for all datasets except for ogbn-arxiv for which we set it to 256. We used cosine similarity for building the kNN graphs and tuned the value of k from the set $\{15, 20, 30\}$. We tuned λ (λ controls the relative importance of the two losses) from the set $\{0.1, 1, 10, 100\}$. The code of our experiments will be available upon acceptance of the paper.

For GRCN (Yu et al., 2020), DGCNN (Wang et al., 2019b), and IDGL (Chen et al., 2020), we used the code released by the authors and tuned the hyperparameters as suggested in the original papers. The results of LDS (Franceschi et al., 2019) are directly taken from the original paper. All the

results for our model and the baselines are averaged over 10 runs. We report the mean and standard deviation.

Dataset statistics: The statistics of the datasets used in the experiments can be found in Table 3.

Table 3: Dataset statistics.

Dataset	Nodes	Edges	Classes	Features	Label rate
Cora	2,708	5,429	7	1,433	0.052
Citeseer	3,327	4,732	6	3,703	0.036
Wine	178	0	3	13	0.112
Cancer	569	0	2	30	0.035
Digits	1,797	0	10	64	0.056
20news	9,607	0	10	236	0.021
Pubmed	19,717	44,338	3	500	0.003
ogbn-arxiv	169,343	1,166,243	40	128	0.537