SLAPS: Self-Supervision Improves Structure Learning for Graph Neural Networks

Bahare Fatemi [*1] **Layla El Asri** [2] **Seyed Mehran Kazemi** [2]

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 a task-specific 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 task-specific supervision 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 through self-supervision. 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).

The performance of GNNs highly depends on the quality of the input graph structure and deteriorates when the graph structure is noisy (see Zügner et al., 2018; Dai et al., 2018; Fox & Rajamanickam, 2019). The need for 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 an 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 develop a model that learns the GNN parameters and an adjacency matrix simultaneously. Our goal is to learn a structure that maximizes the GNN performance on the downstream task. This is different from the works that aim at discovering the node relations or dependencies, e.g., in probabilistic graphical models. 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 well-motivated self-supervised task that helps learn a high-quality adjacency matrix. The self-supervised task is generic and can be combined with several existing approaches. It works by masking some input features (or adding noise to them) and training a separate GNN aiming at updating the adjacency matrix in such a way that it can recover the masked (or noisy) features.

We experiment with several 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

[*]This work was done during an internship at Borealis AI. [1]University of British Columbia [2]Borealis AI. Correspondence to: Bahare Fatemi <bfatemi@cs.ubc.ca>.

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; Belkin et al., 2006). 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). Moreover, designing an appropriate similarity metric may not always be straightforward.

Fully-connected Graph: Another approach is to start with a fully-connected graph and assign edges weights using the available meta-data or employ GNN variants such as graph attention networks (Veličković et al., 2018; Zhang et al., 2018) which provide weights for each edge via an attention mechanism. 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). The complexity of this approach grows rapidly making it applicable only to small-sized graphs. Zhang et al. (2020) propose to define local neighborhoods for each node and only assume that these local neighborhoods are fully connected. Their approach, however, relies on an initial graph structure to define the local neighborhoods.

Learnable Graph: Instead of a similarity graph based 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 bilinear similarity function with learnable parameters. Franceschi et al. (2019) sample graph structures from a learnable fully-connected structure and employ a bi-level optimization setup for simultaneously learning the GNN parameters and the structure. 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. A common approach in this category is to learn a projection of 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 for projection which uses the initial node features as well as an initial graph structure, aiming at providing a revised graph structure to the task-specific GNN. In our experiments, we compare with several approaches from this category.

Leveraging Domain Knowledge: In some applications, one may leverage domain knowledge 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 attacks by learning a cleaned version of the input structure. They use the domain knowledge that clean adjacency matrices are often sparse and low-rank and exhibit feature smoothness along connected nodes. In our paper, we experiment with general-purpose datasets without access to domain knowledge.

Proposed Method: Our model falls within the learnable graph category. We supplement the training with a self-supervised objective to increase the amount of supervision in learning a structure. Our self-supervised task is inspired by, and similar to, the pre-training strategies for GNNs (Hu et al., 2020b;c; Jin et al., 2020a; You et al., 2020; Zhu et al., 2020) (specifically, we adopt the multi-task learning framework of You et al. (2020)), but it 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.

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 i^{th} element as $\boldsymbol{v}_i$. For a matrix $\boldsymbol{M}$, we represent the i^{th} row as $\boldsymbol{M}_i$ and the element at the i^{th} 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 $\mathcal{V} = \{v_1, \dots, v_n\}$ is a set of nodes, $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is an 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.

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 (Nair & Hinton, 2010), 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. Here, $\boldsymbol{D}$ is a (diagonal) degree matrix for $(\boldsymbol{A} + \boldsymbol{I})$ defined as $\boldsymbol{D}_{ii} = 1 + \sum_j \boldsymbol{A}_{ij}$.

4. Proposed Method: SLAPS

SLAPS consists of four components: 1) generator, 2) adjacency processor, 3) classifier, and 4) self-supervision. Figure 1 illustrates these components. We describe each component in more detail and motivate the need for self-supervision.

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 $\boldsymbol{X} \in \mathbb{R}^{n \times f}$ as input and produces a (perhaps sparse, non-normalized, and non-symmetric) matrix $\tilde{\boldsymbol{A}} \in \mathbb{R}^{n \times n}$ as output. We consider the following two generators and leave experimenting with more sophisticated graph generators (e.g., (You et al., 2018; Liu et al., 2019)) and models with tractable adjacency computations (e.g., (Choromanski et al., 2020)) as future work.

Full Parameterization (FP): For this generator, $\theta_{\mathsf{G}} \in \mathbb{R}^{n \times n}$ 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. 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. The advantages of this generator include its simplicity and flexibility for learning any adjacency matrix. Its disadvantages include adding n^2 parameters to the model, which limits scalability and makes the model susceptible to overfitting.

MLP-kNN: Here, 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 $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ with $\boldsymbol{S}_{ij} = \mathsf{Sim}(\boldsymbol{X}'_i, \boldsymbol{X}'_j)$ for some differentiable similarity function Sim (we used cosine). Then $\tilde{\boldsymbol{A}} = \mathsf{kNN}(\boldsymbol{X}') = \boldsymbol{M} \odot \boldsymbol{S}$ where $\odot$ represents the Hadamard (element-wise) product. 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 $\boldsymbol{M}_{ij} = 1$. In our experiments, we compute exact k-nearest neighbors; one can approximate it using locality-sensitive hashing approaches for larger graphs (see, e.g., (Halcrow et al., 2020; Kitaev et al., 2020)). 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. 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 $\boldsymbol{S}$.

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 $\boldsymbol{A}^{kNN}$. For MLP-kNN, we consider two variants. 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 are positive and make the adjacency symmetric and normalized, we let:

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

Here P is a function with a non-negative range applied element-wise on its input. In our experiments, when using an MLP generator, we let P be the ReLU function applied elements-wise on $\tilde{\boldsymbol{A}}$. When using the fully-parameterized

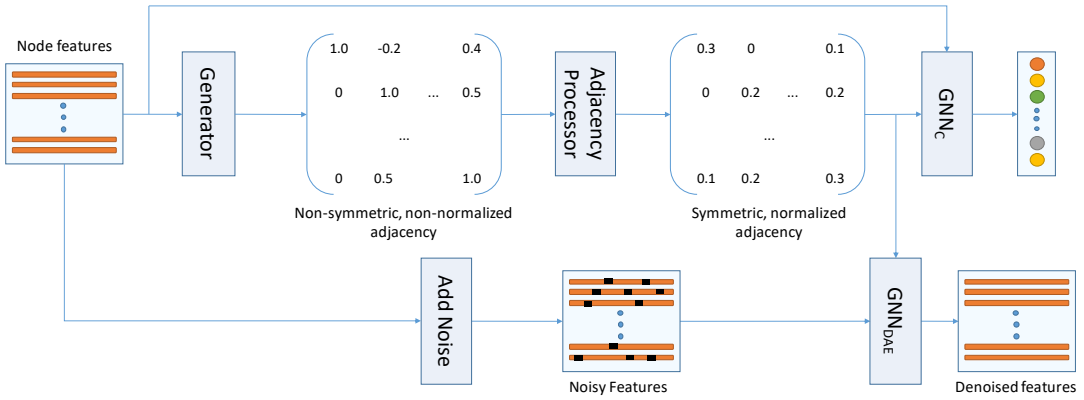

Figure 1. Overview of SLAPS. At the top, a generator receives the node features and produces a non-symmetric, non-normalized adjacency having (possibly) both positive and negative values (Section 4.1). The adjacency processor makes the values positive, symmetrizes and normalizes the adjacency (Section 4.2). The resulting adjacency and the node features go into $\mathsf{GNN_C}$ 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 $\mathsf{GNN_{DAE}}$ which then denoises the features (Section 4.5).

(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 (Clevert et al., 2015) 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 $\mathsf{G_{MLP}}$. 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 $\mathsf{GNN_C} : \mathbb{R}^{n\times f} \times \mathbb{R}^{n\times n} \rightarrow \mathbb{R}^{n\times|\mathcal{C}|}$ with parameters $\theta_{\mathsf{GNN_C}}$. 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 $\theta_{\mathsf{GNN_C}} = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}\}$ and define our classifier as $\mathsf{GNN_C}(\boldsymbol{A}, \boldsymbol{X}; \theta_{\mathsf{GNN_C}}) = \boldsymbol{A}\mathsf{ReLU}(\boldsymbol{A}\boldsymbol{X}\boldsymbol{W}^{(1)})\boldsymbol{W}^{(2)}$ 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. Why not just the first three components?

One may create a model using only the three components described so far corresponding to the top part of Figure 1. As we will explain here, however, this model may suffer severely from supervision starvation. The same problem also applies to many existing approaches for the problem studied in this work, as they can be formulated as a combination of variants of these three components.

Consider a scenario in training the model described above where two unlabeled nodes v_i and v_j are not directly connected to any labeled nodes according to the generated structure. Then, since a two-layer GCN makes predictions for the nodes based on their two-hop neighbors, the classification loss (i.e. $\mathcal{L}_C$) is not affected by the edge between v_i and v_j and this edge receives no supervision[1]. Figure 2 provides an example of such a scenario. Let us call the edges that do not affect the loss function $\mathcal{L}_C$ (and consequently do not receive supervision) as *no-supervision edges*. These edges are clearly problematic because although they may not affect the training loss, the predictions at the test time depend on these edges and if their values are learned without enough supervision, the model may make poor predictions at the test time. A natural question with regard to the extent of the problem caused by such edges is the proportion of no-supervision edges. The following theorem formally establishes the extent of the problem for Erdős-Rényi graphs (Erdős & Rényi, 1959). An *Erdős-Rényi* graph with n nodes and m edges is a graph chosen uniformly at random from the collection of all graphs which have n nodes and m edges.

[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).

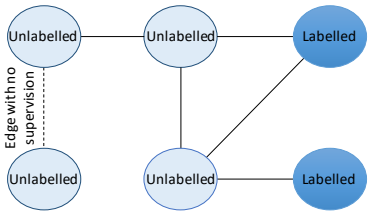

Figure 2. Using a two-layer GCN, the predictions made for the labeled nodes are not affected by the dashed edge.

Theorem 1 *Let $\mathcal{G}(n, m)$ be an Erdős-Rényi graph with n nodes and m edges. Assume we have labels for q nodes selected uniformly at random. The probability of an edge being a no-supervision edge with a two-layer GCN is equal to* $(1 - \frac{q}{n})(1 - \frac{q}{n-1}) \prod_{i=1}^{2q}(1 - \frac{m-1}{\binom{n}{2}-i})$.

We defer the proof to Appendix A. To put the numbers from the theorem in perspective, let us consider three established benchmarks for semi-supervised node classification namely *Cora*, *Citeseer*, and *Pubmed* (the statistics for these datasets can be found in Appendix B). For an Erdős-Rényi graph with similar statistics as the Cora dataset ($n = 2708$, $m = 5429$, $q = 140$), the probability of an edge being a no-supervision edge is 59.4% according to the above theorem. For Citeseer, and Pubmed, this number is 75.7% and 96.7% respectively.

While Theorem 1 is stated for Erdős-Rényi graphs where the labeled nodes have been selected uniformly at random, in real-world applications the problem may be even more severe as, e.g., the labels may not be distributed evenly in different parts of the graph.

4.5. Self-Supervision

To increase the amount of supervision for learning the structure and remedy the problem pointed out in Section 4.4, we propose a self-supervised approach based on denoising autoencoders (Vincent et al., 2008). Let $\mathsf{GNN}_{\mathsf{DAE}} : \mathbb{R}^{n\times f} \times \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times f}$ be a GNN with parameters $\theta_{\mathsf{GNN}_{\mathsf{DAE}}}$ that takes node features and a normalized adjacency produced by a generator as input and provides updated node features with the same dimension as output. 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. During training, we minimize:

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

where $\boldsymbol{A}$ is the generated adjacency matrix and L is a loss function. For datasets where the features consist of binary vectors, idx consists of r percent of the indices of $\boldsymbol{X}$ whose values are 1 and $r\eta$ percent of the indices whose values are 0, 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 1s in the selected mask to 0s 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 0 or by adding independent Gaussian noises to each of the features. In this case, L is the mean-squared error loss.

This self-supervision uses the intuition that the node features are correlated with the node labels and helps by incorporating the inductive bias that a graph structure that is appropriate for predicting the node features is also appropriate for predicting the node labels. Although some edges may not receive supervision from the main task (i.e. from $\mathsf{GCN}_{\mathsf{C}}$ – see Section 4.4), the supervision provided by this task (i.e. from $\mathsf{GCN}_{\mathsf{DAE}}$) helps learn an appropriate weight for them.

4.6. SLAPS

Our final model 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 $\mathsf{GNN}_{\mathsf{DAE}}$ 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 $\mathsf{GNN}_{\mathsf{DAE}}$ model by minimizing $\mathcal{L}_{DAE}$ described in in Equation 2. Recall that $\mathcal{L}_{DAE}$ depends on the parameters θ_{G} of the generator and the parameters $\theta_{\mathsf{GNN}_{\mathsf{DAE}}}$ 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 $\mathsf{GNN}_{\mathsf{DAE}}$.

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. We also compare against MLP-GAM* (Stretcu et al., 2019) which learns a fully-connected graph structure and uses this structure to supplement the loss function of the MLP toward predicting similar labels for neighboring nodes. Our third baseline is label propagation (LP) (Zhu & Ghahramani, 2002), a well-known model for semi-supervised learning. Similar to Franceschi et al. (2019), we also consider a baseline named *kNN-GCN* where we create a kNN graph

Table 1. Results of SLAPS and the baselines on established node classification benchmarks. † indicates results have been taken from Franceschi et al. (2019). ‡ indicates results have been taken from Stretcu 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	$\underline{54.7 \pm 0.1}$
*MLP-GAM**		$70.7^{\ddagger}$	$70.3^{\ddagger}$	–	–	$71.9^{\ddagger}$	–
LP		37.6 ± 0.0	23.2 ± 0.0	36.2 ± 0.0	29.1 ± 0.0	41.3 ± 0.0	OOM
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

based on the node features and feed this graph to a GCN. The graph structure remains fixed in this 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 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 (non-graph) datasets available in scikit-learn (Pedregosa et al., 2011) including Wine, Cancer, Digits, and 20News. Dataset statistics can be found in Appendix B. For Cora and Citeseer, the LDS model uses the train data for learning the parameters of their classification GCN, half of the validation for learning the parameters of the adjacency matrix (in their bi-level optimization setup, these are considered as hyperparameters), and the other half of the validation set for early stopping and tuning the other hyperparameters. Besides experimenting with the original setups of these two datasets, we also consider a setup that is closer to that of LDS: we use the train set and half of the validation set for training and the other half of validation for early stopping and hyperparameter tuning. We name the modified versions Cora390 and Citeseer370 respectively where the number proceeding the dataset name corresponds to the number of labels used for training. We also follow a similar procedure for the scikit-learn datasets.

Implementation: We defer the implementation details and the best hyperparameter settings for our model on all the datasets to Appendix C. Code and data is available at https://github.com/BorealisAI/SLAPS-GNN.

5.1. Comparative results

The results of SLAPS and the baselines on the node classification benchmarks are in Table 1. Considering the baselines first, we see that learning a fully-connected graph in MLP-GAM* makes it outperform MLP. 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; a low-quality structure can harm model performance. LDS outperforms MLP but 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.

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, the baselines that learn a graph structure fail on ogbn-arxiv; our implementation, on the other hand, scales 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 (9,607 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.

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	$\mathbf{96.6 \pm 0.2}$	$\underline{93.2 \pm 0.6}$	$\underline{49.8 \pm 0.9}$

5.2. The Effectiveness of Self-supervision

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 4.6 that in SLAPS$_{2s}$, the adjacency is learned only based on the self-supervision task and the node labels are only used for early stopping, hyperparameter tuning, and training $\mathsf{GCN_C}$. 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 $\mathsf{GCN_C}$ 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.

The effect of the training set size: According to Theorem 1, a smaller q (corresponding to the training set size) results in more no-supervision edges in each epoch. To explore the effect of self-supervision as a function of q, we compared SLAPS with and without supervision on Cora and Citeseer while reducing the number of labeled nodes per class from 20 to 5. We used the FP generator for this experiment. With 5 labeled nodes per class, adding self-supervision provides 16.7% and 22.0% improvements on Cora and Citeseer respectively, which is substantially higher than the corresponding numbers when using 20 labeled nodes per class (10.0% and 7.0% respectively). This provides empirical evidence for Theorem 1.

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

5.3. Analyses of kNN and Symmetrization

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.

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.

5.4. Experiments with Noisy Graphs

So far, we have shown that self-supervision helps learn a better graph structure for GNNs. Here, we verify whether self-supervision is also helpful when a noisy structure is provided as input. Toward this goal, we experiment with Cora

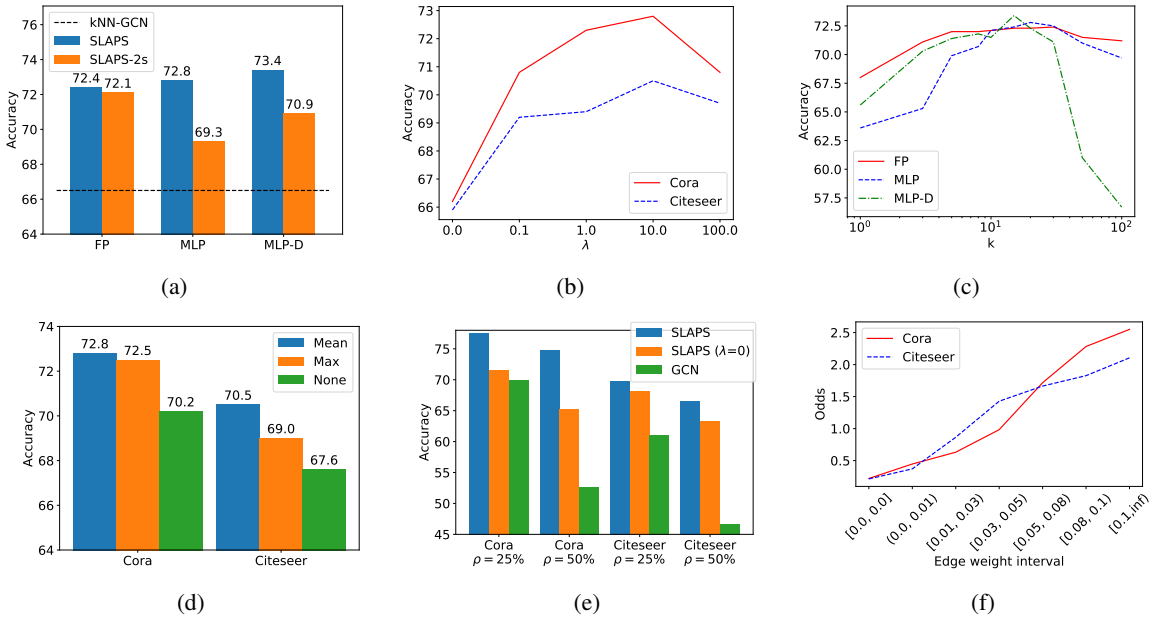

Figure 3. The performance of SLAPS (a) compared to SLAPS_{2s} on Cora with different generators, (b) with MLP graph generator on Cora and Citeseer as a function of λ, (c) with different graph generators on Cora as a function of k in kNN, and (d) on Cora and Citeseer with different adjacency symmetrizations, (e) compared to SLAPS with $\lambda = 0$ and GCN when noisy graphs are provided as input (ρ indicates the percentage of perturbations). (f) The odds of two nodes in the test set sharing the same label as a function of the edge weights learned by SLAPS.

and Citeseer and provide noisy versions of the input graph as input. We perturb the graph structure by replacing ρ percent of the edges in the original structure (selected uniformly at random) with random edges. Figure 3(e) shows the performance of SLAPS with and wihtout self-supervision ($\lambda = 0$ corresponds to no supervision). We also report the results of vanilla GCN on these perturbed graphs for comparison. It can be viewed that self-supervision consistently provides a boost in performance especially for higher values of ρ.

5.5. Analyses of the Learned Adjacency

Noisy graphs: Following the experiment in Section 5.4, we compared the learned and original structures by measuring the number of random edges added during perturbation but removed by the model and the number of edges removed during the perturbation but recovered by the model. For Cora, SLAPS removed 76.2% and 70.4% of the noisy edges and recovered 58.3% and 44.5% of the removed edges for $\rho = 25\%$ and $\rho = 50\%$ respectively while SLAPS with $\lambda = 0$ only removed 62.8% and 54.9% of the noisy edges and recovered 51.4% and 35.8% of the removed edges. This provides evidence on self-supervision being helpful for structure learning.

Cluster assumption: 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 learned adjacency, 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(f) represents the odds for different weight intervals (recall that $\boldsymbol{A}$ is row and column normalized). 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. Specifically, when $\boldsymbol{A}_{ij} \geq 0.1$, v_i and v_j are almost 2.5 and 2.0 times more likely to share the same label on Cora and Citeseer respectively. Note that SLAPS may connect nodes based on a different criterion than the one used in the original datasets and so the learned adjacencies in this experiment do not necessarily resemble the original structures.

6. Conclusion

We proposed SLAPS which is a model for learning the parameters of a graph neural network and the graph structure of the nodes simultaneously based on self-supervision. We explored the design space of SLAPS by comparing different graph generation and symmetrization approaches. We showed the effectiveness of our model using a comprehensive set of experiments and analyses.

References

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Belkin, M., Niyogi, P., and Sindhwani, V. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *JMLRR*, 7(Nov):2399–2434, 2006.

Chami, I., Abu-El-Haija, S., Perozzi, B., Ré, C., and Murphy, K. Machine learning on graphs: A model and comprehensive taxonomy. *arXiv preprint arXiv:2005.03675*, 2020.

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

Chen, Y., Wu, L., and Zaki, M. J. 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.

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

Cosmo, L., Kazi, A., Ahmadi, S.-A., Navab, N., and Bronstein, M. Latent patient network learning for automatic diagnosis. *arXiv preprint arXiv:2003.13620*, 2020.

Dai, H., Li, H., Tian, T., Huang, X., Wang, L., Zhu, J., and Song, L. Adversarial attack on graph structured data. *arXiv preprint arXiv:1806.02371*, 2018.

Erdős, P. and Rényi, A. On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

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

Franceschi, L., Niepert, M., Pontil, M., and He, X. Learning discrete structures for graph neural networks. In *ICML*, 2019.

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

Gidaris, S. and Komodakis, N. 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.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *ICML*, pp. 1263–1272, 2017.

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

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *NeurIPS*, 2017.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020a.

Hu, W., Liu, B., Gomes, J., Zitnik, M., Liang, P., Pande, V., and Leskovec, J. Strategies for pre-training graph neural networks. In *ICLR*, 2020b.

Hu, Z., Dong, Y., Wang, K., Chang, K.-W., and Sun, Y. 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.

Jang, S., Moon, S.-E., and Lee, J.-S. Brain signal classification via learning connectivity structure. *arXiv preprint arXiv:1905.11678*, 2019.

Jin, W., Derr, T., Liu, H., Wang, Y., Wang, S., Liu, Z., and Tang, J. Self-supervised learning on graphs: Deep insights and new direction. *arXiv preprint arXiv:2006.10141*, 2020a.

Jin, W., Ma, Y., Liu, X., Tang, X., Wang, S., and Tang, J. Graph structure learning for robust graph neural networks. *arXiv preprint arXiv:2005.10203*, 2020b.

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

Kazemi, S. M., Goel, R., Jain, K., Kobyzev, I., Sethi, A., Forsyth, P., and Poupart, P. Representation learning for dynamic graphs: A survey. *JMLR*, 2020.

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

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

Kitaev, N., Kaiser, Ł., and Levskaya, A. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Li, Q., Han, Z., and Wu, X.-M. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI*, 2018a.

Li, R., Wang, S., Zhu, F., and Huang, J. Adaptive graph convolutional neural networks. *arXiv preprint arXiv:1801.03226*, 2018b.

Liu, J., Kumar, A., Ba, J., Kiros, J., and Swersky, K. Graph normalizing flows. In *NeurIPS*, pp. 13556–13566, 2019.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

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

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. Scikit-learn: Machine learning in python. *JMLR*, 12:2825–2830, 2011.

Qasim, S. R., Kieseler, J., Iiyama, Y., and Pierini, M. Learning representations of irregular particle-detector geometry with distance-weighted graph networks. *The European Physical Journal C*, 79(7):1–11, 2019.

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

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

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., and Eliassi-Rad, T. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

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

Stretcu, O., Viswanathan, K., Movshovitz-Attias, D., Platanios, E., Ravi, S., and Tomkins, A. Graph agreement models for semi-supervised learning. In *NeurIPS*, pp. 8713–8723, 2019.

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

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

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. In *ICLR*, 2018.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. Extracting and composing robust features with denoising autoencoders. In *ICML*, pp. 1096–1103, 2008.

Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019a.

Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., and Solomon, J. M. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)*, 38 (5):1–12, 2019b.

Wu, X., Zhao, L., and Akoglu, L. A quest for structure: Jointly learning the graph structure and semi-supervised classification. In *CIKM*, pp. 87–96, 2018.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *ICLR*, 2019.

Yang, L., Kang, Z., Cao, X., Jin, D., Yang, B., and Guo, Y. Topology optimization based graph convolutional network. In *IJCAI*, pp. 4054–4061, 2019.

You, J., Ying, R., Ren, X., Hamilton, W. L., and Leskovec, J. Graphrnn: Generating realistic graphs with deep auto-regressive models. *arXiv preprint arXiv:1802.08773*, 2018.

You, Y., Chen, T., Wang, Z., and Shen, Y. When does self-supervision help graph convolutional networks? *arXiv preprint arXiv:2006.09136*, 2020.

Yu, D., Zhang, R., Jiang, Z., Wu, Y., and Yang, Y. Graph-revised convolutional network. In *ECML PKDD*, 2020.

Zhang, J., Shi, X., Xie, J., Ma, H., King, I., and Yeung, D.-Y. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.

Zhang, J., Zhang, H., Xia, C., and Sun, L. Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140*, 2020.

Zhu, H., Lin, Y., Liu, Z., Fu, J., Chua, T.-s., and Sun, M. Graph neural networks with generated parameters for relation extraction. *arXiv preprint arXiv:1902.00756*, 2019.

Zhu, Q., Du, B., and Yan, P. Self-supervised training of graph convolutional networks. *arXiv preprint arXiv:2006.02380*, 2020.

Zhu, X. and Ghahramani, Z. Learning from labeled and unlabeled data with label propagation. 2002.

Zhu, X., Ghahramani, Z., and Lafferty, J. D. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pp. 912–919, 2003.

Zügner, D., Akbarnejad, A., and Günnemann, S. 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. Proof of Theorem 1

Theorem 1 *Let $\mathcal{G}(n, m)$ be an Erdős-Rényi graph with n nodes and m edges. Assume we have labels for q nodes selected uniformly at random. The probability of an edge being a no-supervision edge with a two-layer GCN is equal to* $(1-\frac{q}{n})(1-\frac{q}{n-1})\prod_{i=1}^{2q}(1-\frac{m-1}{\binom{n}{2}-i})$.

Proof. To compute the probability of an edge being a no-supervision edge, we first compute the probability of the two nodes of the edge being unlabeled themselves and then the probability of the two nodes not being connected to any labeled nodes. Let v and u represent two nodes connected by an edge.

With n nodes and q labels, the probability of a node being labeled is $\frac{q}{n}$. Therefore, $Pr(v\ is\ unlabeled) = (1-\frac{q}{n})$ and $Pr(u\ is\ unlabeled \mid v\ is\ unlabeled) = (1-\frac{q}{n-1})$. Therefore, $Pr(v\ is\ unlabeled,\ u\ is\ unlabeled) = (1-\frac{q}{n})(1-\frac{q}{n-1})$.

Since there is an edge between v and v, there are $m-1$ edges remaining. Also, there are $\binom{n}{2}-1$ pairs of nodes that can potentially have an edge between them. Therefore, the probability of v being disconnected from the first labeled node is $1-\frac{m-1}{\binom{n}{2}-1}$. If v is disconnected from the first labeled node, there are still $m-1$ edges remaining and there are now $\binom{n}{2}-2$ pairs of nodes that can potentially have an edge between them. So the probability of v being disconnected from the second node given that it is disconnected from the first labeled node is $1-\frac{m-1}{\binom{n}{2}-2}$. With similar reasoning, we can see that the probability of v being disconnected from the i-th labeled node given that it is disconnected from the first $i-1$ labeled nodes is $1-\frac{m-1}{\binom{n}{2}-i}$.

We can follow similar reasoning for u. The probability of u being disconnected from the first labeled node given that v is disconnected from all q labeled nodes is $1-\frac{m-1}{\binom{n}{2}-q-1}$. That is because there are still $m-1$ edges remaining and $\binom{n}{2}-q-1$ pairs of nodes that can potentially be connected with an edge. We can also see that the probability of u being disconnected from the i-th labeled node given that it is disconnected from the first $i-1$ labeled nodes and that v is disconnected from all q labeled nodes is $1-\frac{m-1}{\binom{n}{2}-q-i}$.

As the probability of the two nodes being unlabeled and not being connected to any labeled nodes in the graph are independent, their joint probability is the multiplication of their probabilities computed above and it is equal to $(1-\frac{q}{n})(1-\frac{q}{n-1})\prod_{i=1}^{2q}(1-\frac{m-1}{\binom{n}{2}-i})$. □

B. Dataset statistics

The statistics of the datasets used in the experiments can be found in Table 4.

C. Implementation Details

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 $\mathsf{GNN_C}$ and $\mathsf{GNN_{DAE}}$ 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 $\mathsf{GCN_C}$ as lr_C and one for the other parameters of the models as lr_{DAE}. 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

Table 3. Best set of hyperparameters for different datasets chosen on validation set.

Dataset	Generator	lr_C	lr_{DAE}	$dropout_c$	$dropout_{DAE}$	k	λ	r	η
Cora	FP	0.001	0.01	0.5	0.25	30	10	10	5
Cora	MLP	0.01	0.001	0.25	0.5	20	10	10	5
Cora	MLP-D	0.01	0.001	0.25	0.5	15	10	10	5
Citeseer	FP	0.01	0.01	0.5	0.5	30	1	10	1
Citeseer	MLP	0.01	0.001	0.25	0.5	30	10	10	5
Citeseer	MLP-D	0.001	0.01	0.5	0.5	20	10	10	5
Cora390	FP	0.01	0.01	0.25	0.5	20	100	10	5
Cora390	MLP	0.01	0.001	0.25	0.5	20	10	10	5
Cora390	MLP-D	0.001	0.001	0.25	0.5	20	10	10	5
Citeseer370	FP	0.01	0.01	0.5	0.5	30	1	10	1
Citeseer370	MLP	0.01	0.001	0.25	0.5	30	10	10	5
Citeseer370	MLP-D	0.01	0.01	0.25	0.5	20	10	10	5
Pubmed	MLP	0.01	0.01	0.5	0.5	15	10	10	5
Pubmed	MLP-D	0.01	0.01	0.25	0.25	15	100	5	5
ogbn-arxiv	MLP	0.01	0.001	0.25	0.5	15	10	1	5
ogbn-arxiv	MLP-D	0.01	0.001	0.5	0.25	15	10	1	5
Wine	FP	0.01	0.001	0.5	0.5	20	0.1	5	5
Wine	MLP	0.01	0.001	0.5	0.25	20	0.1	5	5
Wine	MLP-D	0.01	0.01	0.25	0.5	10	1	5	5
Cancer	FP	0.01	0.001	0.5	0.25	20	0.1	5	5
Cancer	MLP	0.01	0.001	0.5	0.5	20	1.0	5	5
Cancer	MLP-D	0.01	0.01	0.5	0.5	20	0.1	5	5
Digits	FP	0.01	0.001	0.25	0.5	20	0.1	5	5
Digits	MLP	0.01	0.001	0.25	0.5	20	10	5	5
Digits	MLP-D	0.01	0.01	0.25	0.25	20	0.1	5	5
20news	FP	0.01	0.01	0.5	0.5	20	500	5	5
20news	MLP	0.001	0.001	0.25	0.5	20	500	5	5
20news	MLP-D	0.01	0.01	0.25	0.25	20	100	5	5

Table 4. 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
Pubmed	19,717	44,338	3	500	0.003
ogbn-arxiv	169,343	1,166,243	40	128	0.537
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

for both GNN_C and $\mathsf{GNN}_\mathsf{DAE}$ as $dropout_C$ $dropout_{DAE}$ respectively 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 $\{10, 15, 20, 30\}$. We tuned λ (λ controls the relative importance of the two losses) from the set $\{0.1, 1, 10, 100, 500\}$. We tuned r and η from the sets $\{1, 5, 10\}$ and $\{1, 5\}$ respectively. The best set of hyperparameters for each dataset chosen on the validation set is in table 3. 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. For LP (Zhu et al., 2003), we used scikit-learn python package (Pedregosa et al., 2011).

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