Robust Graph Representation Learning via Neural Sparsification

Cheng Zheng [1] **Bo Zong** [2] **Wei Cheng** [2] **Dongjin Song** [2] **Jingchao Ni** [2] **Wenchao Yu** [2] **Haifeng Chen** [2] **Wei Wang** [1]

Abstract

Graph representation learning serves as the core of important prediction tasks, ranging from product recommendation to fraud detection. Real-life graphs usually have complex information in the local neighborhood, where each node is described by a rich set of features and connects to dozens or even hundreds of neighbors. Despite the success of neighborhood aggregation in graph neural networks, task-irrelevant information is mixed into nodes' neighborhood, making learned models suffer from sub-optimal generalization performance. In this paper, we present NeuralSparse, a supervised graph sparsification technique that improves generalization power by learning to remove potentially task-irrelevant edges from input graphs. Our method takes both structural and non-structural information as input, utilizes deep neural networks to parameterize sparsification processes, and optimizes the parameters by feedback signals from downstream tasks. Under the NeuralSparse framework, supervised graph sparsification could seamlessly connect with existing graph neural networks for more robust performance. Experimental results on both benchmark and private datasets show that NeuralSparse can yield up to 7.2% improvement in testing accuracy when working with existing graph neural networks on node classification tasks.

1. Introduction

Representation learning has been in the center of many machine learning tasks on graphs, such as name disambiguation in citation networks (Zhang et al., 2018), spam detection in social networks (Akoglu et al., 2015), recommendations in online marketing (Ying et al., 2018), and many others (Yu et al., 2018; Li et al., 2018). As a class of models that can simultaneously utilize non-structural (*e.g.*, node and edge features) and structural information in graphs, Graph Neural Networks (GNNs) construct effective representations for downstream tasks by iteratively aggregating neighborhood information (Li et al., 2016; Hamilton et al., 2017; Kipf & Welling, 2017). Such methods have demonstrated state-of-the-art performance in classification and prediction tasks on graph data (Veličković et al., 2018; Chen et al., 2018; Xu et al., 2019; Ying et al., 2019).

Meanwhile, the underlying motivation why two nodes get connected may have no relation to a target downstream task, and such task-irrelevant edges could hurt neighborhood aggregation as well as the performance of GNNs. Consider the following example shown in Figure 1. Blue and Red are two classes of nodes, whose two-dimensional features are generated following two independent Gaussian distributions, respectively. As shown in Figure 1(a), the overlap between their feature distributions makes it difficult to find a good boundary that well separates the Blue and Red nodes by node features only. Blue and Red nodes are also inter-connected forming a graph. For each node (either Blue or Red), it randomly selects 10 nodes as its one-hop neighbors, and the resulting edges may not be related to node labels. On such a graph, we train a two-layer GCN (Kipf & Welling, 2017), and the node representations output from the two-layer GCN is illustrated in Figure 1(b). When task-irrelevant edges are mixed into neighborhood aggregation, the trained GCN fails to learn better representations, and it becomes difficult to learn a subsequent classifier with strong generalization power.

In this paper, we study how to utilize supervision signals to remove task-irrelevant edges in an inductive manner to achieve robust graph representation learning. Conventional methods, such as graph sparsification (Liu et al., 2018; Zhang & Patone, 2017; Leskovec & Faloutsos, 2006; Sadhanala et al., 2016), are unsupervised such that the resulting sparsified graphs may not favor downstream tasks. Several works (Zeng et al., 2020; Hamilton et al., 2017; Chen et al., 2018) focus on downsampling under predefined dis-

[1]Department of Computer Science, University of California, Los Angeles, CA, USA [2]NEC Laboratories America, Princeton, NJ, USA. Correspondence to: Cheng Zheng <chengzheng@cs.ucla.edu>, Wei Wang <weiwang@cs.ucla.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.

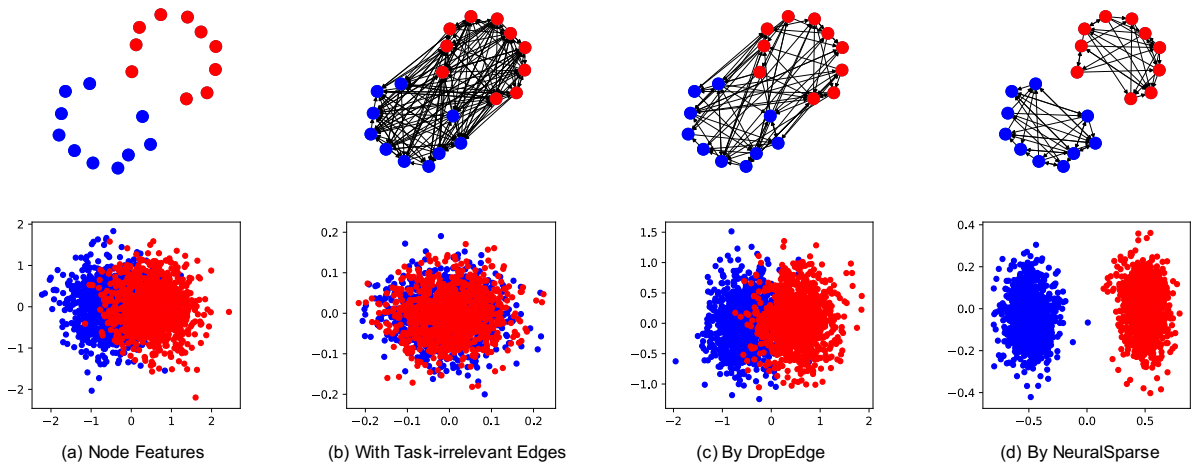


Figure 1. Top row: Small samples of (sparsified) graphs for illustration. Bottom row: Visualization of (learned) node representations. (a) Node representations are input two-dimensional node features. (b) Node representations are learned from a two-layer GCN on top of graphs with task irrelevant edges. (c) Node representations are learned from DropEdge (with a two-layer GCN). (d) Node representations are learned from NeuralSparse (with a two-layer GCN).

tributions. As the predefined distributions may not well adapt to subsequent tasks, such methods could suffer sub-optimal prediction performance. Multiple recent efforts strive to make use of supervision signals to remove noise edges (Wang et al., 2019). However, the proposed methods are either transductive with difficulty to scale (Franceschi et al., 2019) or of high gradient variance bringing increased training difficulty (Rong et al., 2020).

Present work. We propose Neural Sparsification (NeuralSparse), a general framework that simultaneously learns to select task-relevant edges and graph representations by feedback signals from downstream tasks. The NeuralSparse consists of two major components: sparsification network and GNN. For the sparsification network, we utilize a deep neural network to parameterize the sparsification process: how to select edges from the one-hop neighborhood given a fixed budget. In the training phase, the network learns to optimize a sparsification strategy that favors downstream tasks. In the testing phase, the network sparsifies input graphs following the learned strategy, instead of sampling subgraphs from a predefined distribution. Unlike conventional sparsification techniques, our technique takes both structural and non-structural information as input and optimizes the sparsification strategy by feedback from downstream tasks, instead of using (possibly irrelevant) heuristics. For the GNN component, the NeuralSparse feeds the sparsified graphs to GNNs and learns graph representations for subsequent prediction tasks. Under the NeuralSparse framework, by the standard stochastic gradient descent and backpropagation techniques, we can simultaneously optimize graph sparsification and representations. As shown in Figure 1(d), with task-irrelevant edges automatically excluded, the node representations learned from the NeuralSparse suggest a clearer boundary between Blue and Red with promising generalization power, and the sparsification learned by NeuralSparse could be more effective than the regularization provided by layer-wise random edge dropping (Rong et al., 2020) shown in Figure1(c).

Experimental results on both public and private datasets demonstrate that NeuralSparse consistently provides improved performance for existing GNNs on node classification tasks, yielding up to 7.2% improvement.

2. Related Work

Our work is related to two lines of research: graph sparsification and graph representation learning.

Graph sparsification. The goal of graph sparsification is to find small subgraphs from input large graphs that best preserve desired properties. Existing techniques are mainly unsupervised and deal with simple graphs without node/edge features for preserving predefined graph metrics (Hübler et al., 2008), information propagation traces (Mathioudakis et al., 2011), graph spectrum (Calandriello et al., 2018; Chakeri et al., 2016; Adhikari et al., 2018), node degree distribution (Eden et al., 2018; Voudigari et al., 2016), node distance distribution (Leskovec & Faloutsos, 2006), or clustering coefficient (Maiya & Berger-Wolf, 2010). Importance based edge sampling has also been studied in a scenario where we could predefine edge importance (Zhao, 2015; Chen et al., 2018).

Unlike existing methods that mainly work with simple

graphs without node/edge features in an unsupervised manner, our method takes node/edge features as parts of input and optimizes graph sparsification by supervision signals from errors made in downstream tasks.

Graph representation learning. Graph neural networks (GNNs) are the most popular techniques that enable vector representation learning for large graphs with complex node/edge features. All existing GNNs share a common spirit: extracting local structural features by neighborhood aggregation. Scarselli et al. (2009) explore how to extract multi-hop features by iterative neighborhood aggregation. Inspired by the success of convolutional neural networks, multiple studies (Defferrard et al., 2016; Bruna et al., 2014) investigate how to learn convolutional filters in the graph spectral domain under transductive settings. To enable inductive learning, convolutional filters in the graph domain are proposed (Simonovsky & Komodakis, 2017; Niepert et al., 2016; Kipf & Welling, 2017; Veličković et al., 2018; Xu et al., 2018), and a few studies (Hamilton et al., 2017; Lee et al., 2018) explore how to differentiate neighborhood filtering by sequential models. Multiple recent works (Xu et al., 2019; Abu-El-Haija et al., 2019) investigate the expressive power of GNNs, and Ying et al. (2019) propose to identify critical subgraph structure with trained GNNs. In addition, Franceschi et al. (2019) study how to sample high-quality subgraphs from a transductive setting by learning Bernoulli variables on individual edges. Recent efforts also attempt to sample subgraphs from predefined distributions (Zeng et al., 2020; Hamilton et al., 2017), and regularize graph learning by random edge dropping (Rong et al., 2020).

Our work contributes from a unique angle: by inductively learning to select task-relevant edges from downstream supervision signal, our technique can further boost generalization performance for existing GNNs.

3. Proposed Method: NeuralSparse

In this section, we introduce the core idea of our method. We start with the notations that are frequently used in this paper. We then describe the theoretical justification behind NeuralSparse and our architecture to tackle the supervised node classification problem.

Notations. We represent an input graph of n nodes as $G = (V, E, \mathbf{A})$: (1) $V \in \mathbb{R}^{n \times d_n}$ includes node features with dimensionality d_n; (2) $E \in \mathbb{R}^{n \times n}$ is a binary matrix where $E(u, v) = 1$ if there is an edge between node u and node v; (3) $\mathbf{A} \in \mathbb{R}^{n \times n \times d_e}$ encodes input edge features of dimensionality d_e. Besides, we use Y to denote the prediction target in downstream tasks (*e.g.,* $Y \in \mathbb{R}^{n \times d_l}$ if we are dealing with a node classification problem with d_l classes).

Theoretical justification. From the perspective of statistical learning, the key of a defined prediction task is to learn $P(Y \mid G)$, where Y is the prediction target and G is an input graph. Instead of directly working with original graphs, we would like to leverage sparsified subgraphs to remove task-irrelevant information. In other words, we are interested in the following variant,

$$P(Y \mid G) \approx \sum_{g \in \mathbb{S}_G} P(Y \mid g) P(g \mid G), \tag{1}$$

where g is a sparsified subgraph, and $\mathbb{S}_G$ is a class of sparsified subgraphs of G.

In general, because of the combinatorial complexity in graphs, it is intractable to enumerate all possible g as well as estimate the exact values of $P(Y \mid g)$ and $P(g \mid G)$. Therefore, we approximate the distributions by tractable functions,

$$\sum_{g \in \mathbb{S}_G} P(Y \mid g) P(g \mid G) \approx \sum_{g \in \mathbb{S}_G} Q_\theta(Y \mid g) Q_\phi(g \mid G) \tag{2}$$

where Q_θ and Q_ϕ are approximation functions for $P(Y \mid g)$ and $P(g \mid G)$ parameterized by θ and ϕ, respectively.

Moreover, to make the above graph sparsification process differentiable, we employ reparameterization tricks (Jang et al., 2017) to make $Q_\phi(g \mid G)$ directly generate differentiable samples, such that

$$\sum_{g \in \mathbb{S}_G} Q_\theta(Y \mid g) Q_\phi(g \mid G) \propto \sum_{g' \sim Q_\phi(g|G)} Q_\theta(Y \mid g') \tag{3}$$

where $g' \sim Q_\phi(g \mid G)$ means g' is a random sample drawn from $Q_\phi(g \mid G)$.

To this end, the key is how to find appropriate approximation functions $Q_\phi(g \mid G)$ and $Q_\theta(Y \mid g)$.

Architecture. In this paper, we propose Neural Sparsification (NeuralSparse) to implement the theoretical framework discussed in Equation 3. As shown in Figure 2, NeuralSparse consists of two major components: the sparsification network and GNNs.

- The sparsification network is a multi-layer neural network that implements $Q_\phi(g \mid G)$: Taking G as input, it generates a random sparsified subgraph of G drawn from a learned distribution.
- GNNs implement $Q_\theta(Y \mid g)$ that takes the sparsified subgraph as input, extracts node representations, and makes predictions for downstream tasks.

As the sparsified subgraph samples are differentiable, the two components can be jointly trained using the gradient descent based backpropagation techniques from a supervised loss function, as illustrated in Algorithm 1. While the

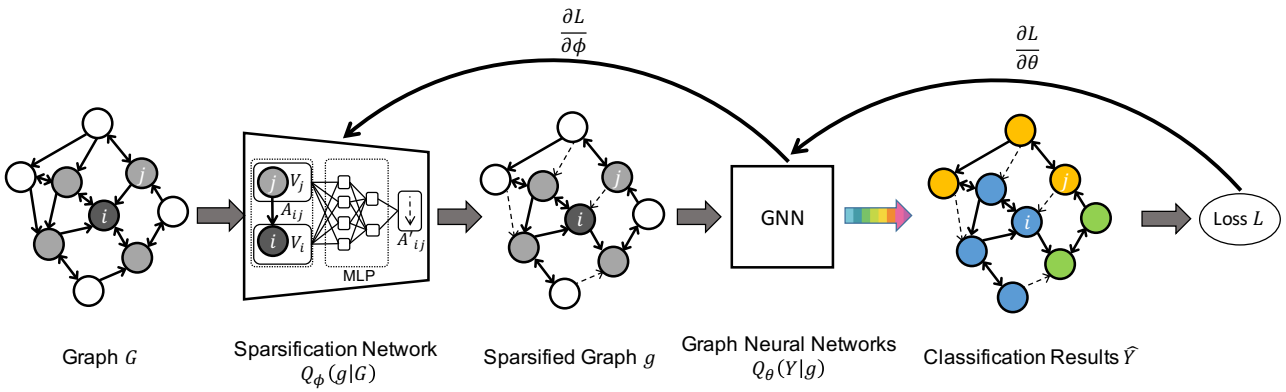


Figure 2. The overview of NeuralSparse

Algorithm 1 Training algorithm for NeuralSparse

1: **Input:** graph $G = (V, E, \mathbf{A})$, integer l, and training labels Y.
2: **while** stop criterion is not met **do**
3: Generate sparsified subgraphs $\{g_1, g_2, \cdots, g_l\}$ by sparsification network (Section 4);
4: Produce prediction $\{\hat{Y}_1, \hat{Y}_2, \cdots, \hat{Y}_l\}$ by feeding $\{g_1, g_2, \cdots, g_l\}$ into GNNs;
5: Calculate loss function J;
6: Update ϕ and θ by descending J
7: **end while**

GNNs have been widely investigated in recent works (Kipf & Welling, 2017; Hamilton et al., 2017; Veličković et al., 2018), we focus on the practical implementation for the sparsification network in the remaining of this paper.

4. Sparsification Network

Following the theory discussed above, the goal of the sparsification network is to generate sparsified subgraphs for input graphs, serving as the approximation function $Q_\phi(g \mid G)$. Therefore, we need to answer the following three questions in the sparsification network. **i**). What is $\mathbb{S}_G$ in Equation 1, the class of subgraphs we focus on? **ii**). How to sample sparsified subgraphs? **iii**). How to make the sparsified subgraph sampling process differentiable for the end-to-end training? In the following, we address the questions one by one.

k-neighbor subgraphs. We focus on k-neighbor subgraphs for $\mathbb{S}_G$ (Sadhanala et al., 2016): Given an input graph, a k-neighbor subgraph shares the same set of nodes with the input graph, and each node in the subgraph can select no more than k edges from its one-hop neighborhood. Although the concept of the sparsification network is not limited to a specific class of subgraphs, we choose k-neighbor subgraphs for the following reasons.

- We are able to adjust the estimation on the amount of task-relevant graph data by tuning the hyper-parameter k. Intuitively, when k is an under-estimate, the amount of task-relevant graph data accessed by GNNs could be inadequate, leading to inferior performance. When k is an over-estimate, the downstream GNNs may overfit the introduced noise or irrelevant graph data, resulting in sub-optimal performance. It could be difficult to set a golden hyper-parameter that works all the time, but one has the freedom to choose the k that is the best fit for a specific task.

- k-neighbor subgraphs are friendly to parallel computation. As each node selects its edges independently from its neighborhood, we can utilize tensor operations in existing deep learning frameworks, such as tensorflow (Abadi et al., 2016), to speed up the sparsification process for k-neighbor subgraphs.

Sampling k-neighbor subgraphs. Given k and an input graph $G = (V, E, \mathbf{A})$, we obtain a k-neighbor subgraph by repeatedly sampling edges for each node in the original graph. Without loss of generality, we sketch this sampling process by focusing on a specific node u in graph G. Let $\mathbb{N}_u$ be the set of one-hop neighbors of the node u.

1. $v \sim f_\phi(V(u), V(\mathbb{N}_u), \mathbf{A}(u))$, where $f_\phi(\cdot)$ is a function that generates a one-hop neighbor v from the learned distribution based on the node u's attributes, node attributes of u's neighbors $V(\mathbb{N}_u)$, and their edge attributes $\mathbf{A}(u)$. In particular, the learned distribution is encoded by parameters ϕ.

2. Edge $E(u, v)$ is selected for the node u.

3. The above two steps are repeated k times.

Note that the above process performs sampling without replacement. Given a node u, each of its adjacent edges is selected at most once. Moreover, the sampling function $f_\phi(\cdot)$ is shared among nodes; therefore, the number of parameters ϕ is independent of the input graph size.

Making samples differentiable. While conventional methods are able to generate discrete samples (Sadhanala et al., 2016), these samples are not differentiable such that it is difficult to utilize them to optimize sample generation. To make samples differentiable, we propose a Gumbel-Softmax based multi-layer neural network to implement the sampling function $f_\phi(\cdot)$ discussed above.

To make the discussion self-contained, we briefly discuss the idea of Gumbel-Softmax. Gumbel-Softmax is a reparameterization trick used to generate differentiable discrete samples (Jang et al., 2017; Maddison et al., 2017). Under appropriate hyper-parameter settings, Gumbel-Softmax is able to generate continuous vectors that are as "sharp" as one-hot vectors widely used to encode discrete data.

Without loss of generality, we focus on a specific node u in a graph $G = (V, E, \mathbf{A})$. Let $\mathbb{N}_u$ be the set of one-hop neighbors of the node u. We implement $f_\phi(\cdot)$ as follows.

1. $\forall v \in \mathbb{N}_u$,

$$z_{u,v} = \text{MLP}_\phi(V(u), V(v), \mathbf{A}(u, v)), \tag{4}$$

   where MLP_ϕ is a multi-layer neural network with parameters ϕ.

2. $\forall v \in \mathbb{N}_u$, we employ a softmax function to compute the probability to sample the edge,

$$\pi_{u,v} = \frac{\exp(z_{u,v})}{\sum_{w \in \mathbb{N}_u} \exp(z_{u,w})} \tag{5}$$

3. Using Gumbel-Softmax, we generate differentiable samples

$$x_{u,v} = \frac{\exp((\log(\pi_{u,v}) + \epsilon_v)/\tau)}{\sum_{w \in \mathbb{N}_u} \exp((\log(\pi_{u,w}) + \epsilon_w)/\tau)} \tag{6}$$

   where $x_{u,v}$ is a scalar, $\epsilon_v = -\log(-\log(s))$ with s randomly drawn from Uniform$(0, 1)$, and τ is a hyper-parameter called *temperature* which controls the interpolation between the discrete distribution and continuous categorical densities.

Note that when we sample k edges, the computation for $z_{u,v}$ and $\pi_{u,v}$ only needs to be performed once. For the hyper-parameter τ, we discuss how to tune it as follows.

Discussion on temperature tuning. The behavior of Gumbel-Softmax is governed by a hyper-parameter τ called temperature. In general, when τ is small, the Gumbel-Softmax distribution resembles the discrete distribution, which induces strong sparsity; however, small τ also introduces high-variance gradients that block effective backpropagation. A high value of τ cannot produce expected sparsification effect. Following the practice in (Jang et al., 2017), we adopt the strategy by starting the training with a high temperature and anneal to a small value with a guided schedule.

Sparsification algorithm and its complexity. As shown in Algorithm 2, given hyper-parameter k, the sparsification network visits each node's one-hop neighbors k times. Let m be the total number of edges in the graph. The complexity of sampling subgraphs by the sparsification network is $O(km)$. When k is small in practice, the overall complexity is $O(m)$.

Algorithm 2 Sampling subgraphs by sparsification network

Input: graph $G = (V, E, \mathbf{A})$ and integer k.
Edge set $\mathbb{H} = \emptyset$
for $u \in \mathbb{V}$ **do**
  for $v \in \mathbb{N}_u$ **do**
    $z_{u,v} \leftarrow \text{MLP}_\phi(V(u), V(v), \mathbf{A}(u, v))$
  end for
  for $v \in \mathbb{N}_u$ **do**
    $\pi_{u,v} \leftarrow \exp(z_{u,v}) / \sum_{w \in \mathbb{N}_u} \exp(z_{u,w})$
  end for
  for $j = 1, \cdots, k$ **do**
    for $v \in \mathbb{N}_u$ **do**
      $x_{u,v} \leftarrow \frac{\exp((\log(\pi_{u,v}) + \epsilon_v)/\tau)}{\sum_{w \in \mathbb{N}_u} \exp((\log(\pi_{u,w}) + \epsilon_w)/\tau)}$
    end for
    Add the edge represented by vector $[x_{u,v}]$ into $\mathbb{H}$
  end for
end for

Comparison with multiple related methods. Unlike FastGCN (Chen et al., 2018), GraphSAINT (Zeng et al., 2020) and DropEdge (Rong et al., 2020) that incorporate layer-wise node samplers to reduce the complexity of GNNs, NeuralSparse samples subgraphs before applying GNNs. As for the computation complexity, the sparsification in NeuralSparse is more friendly to parallel computation than the layer-conditioned approaches such as AS-GCN. Compared with the graph attentional models (Veličković et al., 2018), the NeuralSparse can produce sparser neighborhoods, which effectively remove task-irrelevant information on original graphs. Unlike LDS (Franceschi et al., 2019), NeuralSparse learns graph sparsification under inductive setting, and its graph sampling is constrained by input graph topology.

Table 1. Dataset statistics

	Reddit	PPI	Transaction	Cora	Citeseer
Task	Inductive	Inductive	Inductive	Transductive	Transductive
Nodes	232,965	56,944	95,544	2,708	3,327
Edges	11,606,919	818,716	963,468	5,429	4,732
Features	602	50	120	1,433	3,703
Classes	41	121	2	7	6
Training Nodes	152,410	44,906	47,772	140	120
Validation Nodes	23,699	6,514	9,554	500	500
Testing Nodes	55,334	5,524	38,218	1,000	1,000

5. Experimental Study

In this section, we evaluate our proposed NeuralSparse on the node classification task with both inductive and transductive settings. The experimental results demonstrate that NeuralSparse achieves superior classification performance over state-of-the-art GNN models. Moreover, we provide a case study to demonstrate how the sparsified subgraphs generated by NeuralSparse could improve classification compared against other sparsification baselines. The supplementary material contains more experimental details.

5.1. Datasets

We employ five datasets from various domains and conduct the node classification task following the settings as described in Hamilton et al. (2017) and Kipf & Welling (2017). The dataset statistics are summarized in Table 1.

Inductive datasets. We utilize the Reddit and PPI datasets and follow the same setting in Hamilton et al. (2017). The Reddit dataset contains a post-to-post graph with word vectors as node features. The node labels represent which community Reddit posts belong to. The protein-protein interaction (PPI) dataset contains graphs corresponding to different human tissues. The node features are positional gene sets, motif gene sets, and immunological signatures. The nodes are multi-labeled by gene ontology.

Graphs in the Transaction dataset contains real transactions between organizations in two years, with the first year for training and the second year for validation/testing. Each node represents an organization and each edge indicates a transaction between two organizations. Node attributes are side information about the organizations such as account balance, cash reserve, etc. On this dataset, the objective is to classify organizations into two categories: *promising* or *others* for investment in the near future. More details on the Transaction dataset can be found in Supplementary S1.

In the inductive setting, models can only access training nodes' attributes, edges, and labels during training. In the PPI and Transaction datasets, the models have to generalize to completely unseen graphs.

Transductive datasets. We use two citation benchmark datasets in Yang et al. (2016) and Kipf & Welling (2017) with the transductive experimental setting. The citation graphs contain nodes corresponding to documents and edges as citations. Node features are the sparse bag-of-words representations of the documents and node labels indicate the topic class of the documents. In transductive learning, the training methods have access to all node features and edges, with a limited subset of node labels.

5.2. Experimental Setup

Baseline models. We incorporate four state-of-the-art methods as the base GNN components, including GCN (Kipf & Welling, 2017), GraphSAGE (Hamilton et al., 2017), GAT (Veličković et al., 2018), and GIN (Xu et al., 2019). Besides evaluating the effectiveness and efficiency of NeuralSparse against base GNNs, we leverage three other categories of methods in the experiments: (1) We incorporate the two unsupervised graph sparsification models, the spectral sparsifier (SS, Sadhanala et al., 2016) and the Rank Degree (RD, Voudigari et al., 2016). The input graphs are sparsified before sent to the base GNNs for node classification. (2) We compare against the random layer-wise sampler DropEdge (Rong et al., 2020). Similar to the Dropout trick (Hinton et al., 2012), DropEdge randomly removes connections among node neighborhood in each GNN layer. (3) We also incorporate LDS (Franceschi et al., 2019), which works under a transductive setting and learns Bernoulli variables associated with individual edges.

Temperature tuning. We anneal the temperature with the schedule $\tau = \max(0.05, \exp(-rp))$, where p is the training epoch and $r \in 10^{\{-5,-4,-3,-2,-1\}}$. τ is updated every N steps and $N \in \{50, 100, ..., 500\}$. Compared with the MNIST VAE model in Jang et al. (2017), smaller hyperparameter τ fits NeuralSparse better in practice. More details on the experimental settings and implementation can

Table 2. Node classification performance

Sparsifier	Method	Reddit	PPI	Transaction	Cora	Citeseer
		Micro-F1	Micro-F1	AUC	Accuracy	Accuracy
N/A	GCN	0.922 ± 0.041	0.532 ± 0.024	0.564 ± 0.018	0.810 ± 0.027	0.694 ± 0.020
	GraphSAGE	0.938 ± 0.029	0.600 ± 0.027	0.574 ± 0.029	0.825 ± 0.033	0.710 ± 0.020
	GAT	-	0.973 ± 0.030	0.616 ± 0.022	0.821 ± 0.043	0.721 ± 0.037
	GIN	0.928 ± 0.022	0.703 ± 0.028	0.607 ± 0.031	0.816 ± 0.020	0.709 ± 0.037
SS/ RD	GCN	0.912 ± 0.022	0.521 ± 0.024	0.562 ± 0.035	0.780 ± 0.045	0.684 ± 0.033
	GraphSAGE	0.907 ± 0.018	0.576 ± 0.022	0.565 ± 0.042	0.806 ± 0.032	0.701 ± 0.027
	GAT	-	0.889 ± 0.034	0.614 ± 0.044	0.807 ± 0.047	0.686 ± 0.034
	GIN	0.901 ± 0.021	0.693 ± 0.019	0.593 ± 0.038	0.785 ± 0.041	0.706 ± 0.043
DropEdge	GCN	0.961 ± 0.040	0.548 ± 0.041	0.591 ± 0.040	0.828 ± 0.035	0.723 ± 0.043
	GraphSAGE	0.963 ± 0.043	0.632 ± 0.031	0.598 ± 0.043	0.821 ± 0.048	0.712 ± 0.032
	GAT	-	0.851 ± 0.030	0.604 ± 0.043	0.789 ± 0.039	0.691 ± 0.039
	GIN	0.931 ± 0.031	0.783 ± 0.037	0.625 ± 0.035	0.818 ± 0.044	0.715 ± 0.039
LDS	GCN	-	-	-	0.831 ± 0.017	0.727 ± 0.021
Neural Sparse	GCN	0.966 ± 0.020	0.651 ± 0.014	0.610 ± 0.022	0.837 ± 0.014	$\mathbf{0.741} \pm 0.014$
	GraphSAGE	$\mathbf{0.967} \pm 0.015$	0.696 ± 0.023	0.649 ± 0.018	0.841 ± 0.024	0.736 ± 0.013
	GAT	-	$\mathbf{0.986} \pm 0.015$	$\mathbf{0.671} \pm 0.018$	$\mathbf{0.842} \pm 0.015$	0.736 ± 0.026
	GIN	0.959 ± 0.027	0.892 ± 0.015	0.634 ± 0.023	0.838 ± 0.027	0.738 ± 0.015

be found in Supplementary S2 and S3.

Metrics. We evaluate the performance on the transductive datasets with accuracy (Kipf & Welling, 2017). For inductive tasks on the Reddit and PPI datasets, we report micro-averaged F1 scores (Hamilton et al., 2017). Due to the imbalanced classes in the Transaction dataset, models are evaluated with AUC value (Huang & Ling, 2005). The results show the average of 10 runs.

5.3. Classification Performance

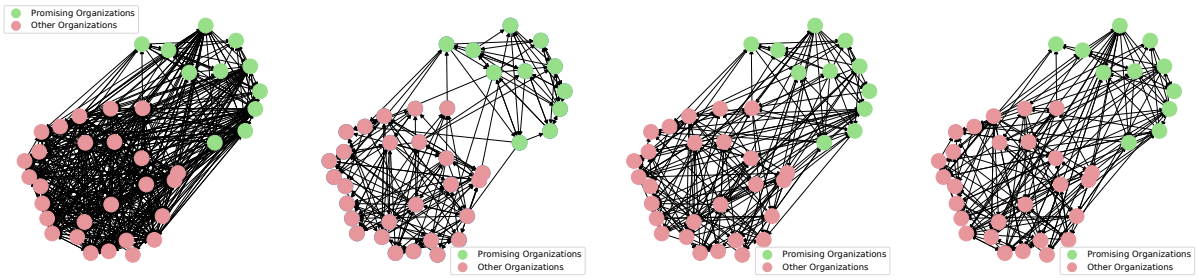
Table 2 summarizes the classification performance of NeuralSparse and the baseline methods on all datasets. For Reddit, PPI, Transaction, Cora, and Citeseer, the hyper-parameter k is set as 30, 15, 10, 5, and 3 respectively. The hyper-parameter l is set as 1. Note that the result of GAT on Reddit is missing due to the out-of-memory error and LDS only works under the transductive setting. For simplicity, we only report the better performance with SS or RD sparsifiers.

Overall, NeuralSparse is able to help GNN techniques achieve competitive generalization performance with sparsified graph data. We make the following observations. (1) Compared with basic GNN models, NeuralSparse can enhance the generalization performance on node classification tasks by utilizing the sparsified subgraphs from the sparsification network, especially in the inductive setting. Indeed, large neighborhood size in the original graphs could increase the chance of introducing noise into the aggregation operations, leading to sub-optimal performance. (2) With different GNN options, the NeuralSparse can consistently achieve comparable or superior performance, while other sparsification approaches tend to favor a certain GNN structure. (3) Compared with DropEdge, NeuralSparse achieves up to 13% of improvement in terms of accuracy with lower variance. In addition, the comparison between NeuralSparse and DropEdge in terms of convergence speed can be found in Supplementary S4. (4) In comparison with the two NeuralSparse variants SS-GNN and RD-GNN, NeuralSparse outperforms because it can effectively leverage the guidance from downstream tasks.

Table 3. Node classification performance with κ-NN graphs

Dataset(κ)	LDS	NeuralSparse
Cora(10)	0.715 ± 0.035	0.723 ± 0.025
Cora(20)	0.703 ± 0.029	0.719 ± 0.021
Citeseer(10)	0.691 ± 0.031	0.723 ± 0.016
Citeseer(20)	0.715 ± 0.026	0.725 ± 0.019

In the following, we discuss the comparison between NeuralSparse and LDS (Franceschi et al., 2019) on the Cora and Citeseer datasets. Note that the row labeled with LDS in Table 2 presents the classification results on original input graphs. In addition, we adopt κ-nearest neighbor (κ-NN) graphs suggested in (Franceschi et al., 2019) for more comprehensive evaluation. In particular, κ-NN graphs are

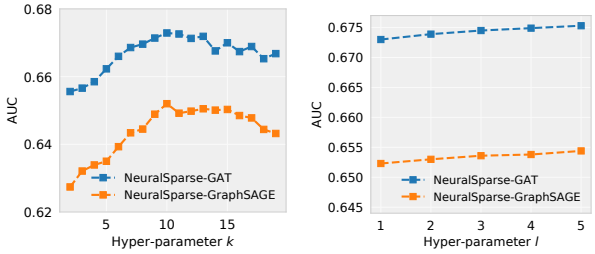


(a) Original (b) NeuralSparse (c) Spectral Sparsifier (d) RD Sparsifier

Figure 3. (a) Original graph from the Transaction dataset and sparsified subgraphs by (b) NeuralSparse, (c) Spectral Sparsifier, and (d) RD Sparsifier.

constructed by connecting individual nodes with their top-κ similar neighbors in terms of node features, and κ is selected from $\{10, 20\}$. In Table 3, we summarize the classification accuracy of LDS (with GCN) and NeuralSparse (with GCN). On both original and κ-NN graphs, NeuralSparse outperforms LDS in terms of classification accuracy. As each edge is associated with a Bernoulli variables, the large number of parameters for graph sparsification could impact the generalization power in LDS. More comparison results between NeuralSparse and LDS can be found in Supplementary S5.

5.4. Sensitivity to Hyper-parameters and the Sparsified Subgraphs



(a) Hyperparameter k (b) Hyperparameter l

Figure 4. Performance vs hyper-parameters

Figure 4(a) demonstrates how classification performance responds when k increases on the Transaction dataset. There exists an optimal k that delivers the best classification AUC score. The similar trend on the validation set is also observed, as shown in Supplementary S6. When k is small, NeuralSparse can only make use of little relevant structural information in feature aggregation, which leads to inferior performance. When k increases, the aggregation convolution involves more complex neighborhood aggregation with a higher chance of overfitting noise data, which negatively impacts the classification performance for unseen testing data. Figure 4(b) shows how hyper-parameter l impacts classification performance on the Transaction dataset. When l increases from 1 to 5, we observe a relatively small improvement in classification AUC score. As the parameters in the sparsification network are shared by all edges in the graph, the estimation variance from random sampling could already be mitigated to some extent by a number of sampled edges in a sparsified subgraph. Thus, when we increase the number of sparsified subgraphs, the incremental gain could be small.

In Figure 3(a), we present a sample of the graph from the Transaction dataset which consists of 38 nodes (promising organizations and other organizations) with an average node degree 15 and node feature dimension 120. As shown in Figure 3(b), the graph sparsified by the NeuralSparse has lower complexity with an average node degree around 5. In Figure 3(c, d), we also present the sparsified graphs output by the two baseline methods, SS and RD. More quantitative evaluations over sparsified graphs from different approaches can be found in Supplementary S7.

By comparing the four plots in Figure 3, we make the following observations: First, the NeuralSparse sparsified graph tends to select edges that connect nodes of identical labels, which favors the downstream classification task. The observed clustering effect could further boost the confidence in decision making. Second, instead of exploring all the neighbors, we can focus on the selected connections/edges, which could make it easier for human experts to perform model interpretation and result visualization.

6. Conclusion

In this paper, we propose Neural Sparsification (NeuralSparse) to address the noise brought by the task-irrelevant information on real-life large graphs. NeuralSparse consists of two major components: (1) The sparsification network sparsifies input graphs by sampling edges following a learned distribution; (2) GNNs take sparsified subgraphs as input and extract node representa-

tions for downstream tasks. The two components in NeuralSparse can be jointly trained with supervised loss, gradient descent, and backpropagation techniques. The experimental study on real-life datasets shows that the NeuralSparse consistently renders more robust graph representations, and yields up to 7.2% improvement in accuracy over state-of-the-art GNN models.

Acknowledgement

We thank the anonymous reviewers for their careful reading and insightful comments on our manuscript. The work was partially supported by NSF (DGE-1829071, IIS-2031187).

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, 2016.

Abu-El-Haija, S., Perozzi, B., Kapoor, A., Harutyunyan, H., Alipourfard, N., Lerman, K., Steeg, G. V., and Galstyan, A. Mixhop: Higher-order graph convolution architectures via sparsified neighborhood mixing. In *ICML*, 2019.

Adhikari, B., Zhang, Y., Amiri, S. E., Bharadwaj, A., and Prakash, B. A. Propagation-based temporal network summarization. *TKDE*, 2018.

Akoglu, L., Tong, H., and Koutra, D. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 2015.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014.

Calandriello, D., Koutis, I., Lazaric, A., and Valko, M. Improved large-scale graph learning through ridge spectral sparsification. In *ICML*, 2018.

Chakeri, A., Farhidzadeh, H., and Hall, L. O. Spectral sparsification in spectral clustering. In *ICPR*, 2016.

Chen, J., Ma, T., and Xiao, C. Fastgcn: fast learning with graph convolutional networks via importance sampling. In *ICLR*, 2018.

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, 2016.

Eden, T., Jain, S., Pinar, A., Ron, D., and Seshadhri, C. Provable and practical approximations for the degree distribution using sublinear graph samples. In *WWW*, 2018.

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

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

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

Huang, J. and Ling, C. X. Using auc and accuracy in evaluating learning algorithms. *TKDE*, 2005.

Hübler, C., Kriegel, H.-P., Borgwardt, K., and Ghahramani, Z. Metropolis algorithms for representative subgraph sampling. In *ICDM*, 2008.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. In *ICLR*, 2017.

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

Lee, J. B., Rossi, R., and Kong, X. Graph classification using structural attention. In *KDD*, 2018.

Leskovec, J. and Faloutsos, C. Sampling from large graphs. In *KDD*, 2006.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. In *ICLR*, 2016.

Li, Y., Yu, R., Shahabi, C., and Liu, Y. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In *ICLR*, 2018.

Liu, Y., Safavi, T., Dighe, A., and Koutra, D. Graph summarization methods and applications: A survey. *ACM Computing Surveys*, 2018.

Maddison, C. J., Mnih, A., and Teh, Y. W. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR*, 2017.

Maiya, A. S. and Berger-Wolf, T. Y. Sampling community structure. In *WWW*, 2010.

Mathioudakis, M., Bonchi, F., Castillo, C., Gionis, A., and Ukkonen, A. Sparsification of influence networks. In *KDD*, 2011.

Niepert, M., Ahmed, M., and Kutzkov, K. Learning convolutional neural networks for graphs. In *ICML*, 2016.

Rong, Y., Huang, W., Xu, T., and Huang, J. Dropedge: Towards deep graph convolutional networks on node classification. In *ICLR*, 2020.

Sadhanala, V., Wang, Y.-X., and Tibshirani, R. Graph sparsification approaches for laplacian smoothing. In *AISTATS*, 2016.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 2009.

Simonovsky, M. and Komodakis, N. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *CVPR*, 2017.

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

Voudigari, E., Salamanos, N., Papageorgiou, T., and Yannakoudakis, E. J. Rank degree: An efficient algorithm for graph sampling. In *ASONAM*, 2016.

Wang, L., Yu, W., Wang, W., Cheng, W., Zhang, W., Zha, H., He, X., and Chen, H. Learning robust representations with graph denoising policy network. In *ICDM*, 2019.

Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K.-i., and Jegelka, S. Representation learning on graphs with jumping knowledge networks. In *ICML*, 2018.

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

Yang, Z., Cohen, W. W., and Salakhutdinov, R. Revisiting semi-supervised learning with graph embeddings. In *ICML*, 2016.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, 2018.

Ying, R., Bourgeois, D., You, J., Zitnik, M., and Leskovec, J. Gnn explainer: A tool for post-hoc explanation of graph neural networks. In *NIPS*, 2019.

Yu, W., Zheng, C., Cheng, W., Aggarwal, C., Song, D., Zong, B., Chen, H., and Wang, W. Learning deep network representations with adversarially regularized autoencoders. In *KDD*, 2018.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. Graphsaint: Graph sampling based inductive learning method. In *ICLR*, 2020.

Zhang, L.-C. and Patone, M. Graph sampling. *Metron*, 2017.

Zhang, Y., Zhang, F., Yao, P., and Tang, J. Name disambiguation in aminer: Clustering, maintenance, and human in the loop. In *KDD*, 2018.

Zhao, P. gsparsify: Graph motif based sparsification for graph clustering. In *CIKM*, 2015.