SubMix: Learning to Mix Graph Sampling Heuristics

Abstract

Sampling subgraphs for training Graph Neural Networks (GNNs) is receiving much attention from the GNN community. While a variety of methods have been proposed, each method samples the graph according to its own heuristic. However, there has been little work in mixing these heuristics in an end-to-end trainable manner. In this work, we design a generative framework for graph sampling. Our method, SubMix, parameterizes graph sampling as a convex combination of heuristics. We show that a continuous relaxation of the discrete sampling process allows us to efficiently obtain analytical gradients for training the sampling parameters. Our experimental results illustrate the usefulness of learning graph sampling in three scenarios: (1) robust training of GNNs by automatically learning to discard noisy edge sources; (2) improving model performance by trainable and online edge subset selection; and (3) by integrating our framework into state-of-the-art (SOTA) decoupled GNN models, for homogeneous OGBN datasets. Our method raises the SOTA on challenging ogbn-arxiv and ogbn-products, respectively, by over 4 and 0.5 percentage points.

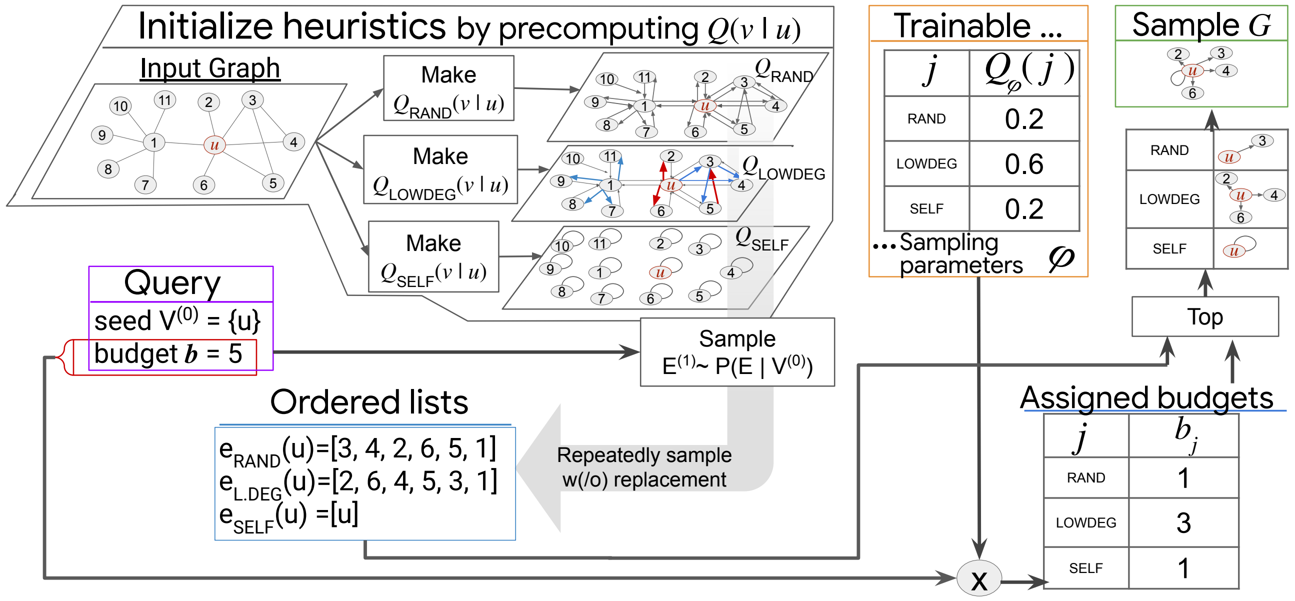
1 INTRODUCTION

Graph Neural Networks (GNNs) are frequently used as machine learning models for relational data, *e.g.*, in predicting node classes or relationships between nodes (*a.k.a.* edges). The node and edge types are domain-specific. Application domains include social, biochemical, and computational networks, *e.g.*, where GNNs are used for recommendation, predicting protein-protein interactions, or job scheduling [Bruna et al., 2014, Kipf and Welling, 2017, Monti et al., 2017, Veličković et al., 2018, Hamilton et al., 2017, Battaglia et al., 2018, Chami et al., 2022].

Early GNNs were conceptually formulated and implemented on the graph as a whole [Bruna et al., 2014, Kipf and Welling, 2017]. As such, the entire graph is processed at every training step. For training on larger graphs, researchers resorted to a variety of directions, including (D.1) **subgraph sampling** and (D.2) **graph-decoupled training**, among others (§6).

Direction (D.1) repeatedly samples *subgraphs* from the (larger) input graph. Each subgraph sample serves as one training example for the GNN. The process of sampling subgraphs–how to select a meaningful neighborhood for a set of nodes–is usually based on a heuristic and is not trainable. For instance, Chiang et al. [2019] partitions the input graph into smaller subgraphs via an algorithm for graph clustering such as METIS [Karypis and Kumar, 1998]. Once the subgraphs are created, they are repeatedly used throughout training (and inference). Zeng et al. [2020] try three samplers: node-level, edge-level, and random walk-based samplers, respectively, similar to [Chen et al., 2018], [Zou et al., 2019] and [Hamilton et al., 2017, Markowitz et al., 2021]. These methods all employ sampling heuristics, *e.g.*, choose nodes with probability proportional to their degree [Chen et al., 2018], or a few edges for every sampled node [Hamilton et al., 2017, Ying et al., 2018, Markowitz et al., 2021]. While these heuristics stem from reasonable inductive biases, learning to combine them is under-explored.

Contributions: We propose a generative model of subgraphs (§3), which can be written as a mixture distribution of sampling heuristics, where the mixture weights are learned. Given a node, each heuristic assigns a probability distribution on its neighbors. We train parameters of the mixture model to optimize the supervision objective, *e.g.*, cross-entropy for node classification. To obtain analytical gradients, we relax the discrete process of subgraph sampling into the continuous domain. We evaluate our method in three setups. First, if we use an adversary sampling heuristic, always yielding non-edges, we find that our method



Initialize heuristics by precomputing $Q(v \mid u)$
Input Graph
Make $Q_{\text{RAND}}(v \mid u)$
Make $Q_{\text{LOWDEG}}(v \mid u)$
Make $Q_{\text{SELF}}(v \mid u)$
Q_{RAND}
Q_{LOWDEG}
Q_{SELF}
Trainable ...
j
$Q_\varphi(j)$
RAND 0.2
LOWDEG 0.6
SELF 0.2
...Sampling parameters φ
Sample G
Query
seed $V^{(0)} = \{u\}$
budget $b = 5$
Sample $E^{(1)} \sim P(E \mid V^{(0)})$
Ordered lists
$e_{\text{RAND}}(u)=[3, 4, 2, 6, 5, 1]$
$e_{\text{L.DEG}}(u)=[2, 6, 4, 5, 3, 1]$
$e_{\text{SELF}}(u) =[u]$
Repeatedly sample w(/o) replacement
Assigned budgets
j b_j
RAND 1
LOWDEG 3
SELF 1
Top
RAND
LOWDEG
SELF

Figure 1: **Summary of our Method**. **(top-left)**. The input graph initializes our utilized heuristics only once. Each heuristic maintains distribution $Q(\cdot \mid u)$ per node u. We show relative mass outgoing a node by edge color: red and blue, for Q_{LowDeg}, depict larger mass than grey edges. Once initialized, **(left)**, user can request 5 edges outgoing from node u. Each heuristic j then yields an **ordered list** of neighbors, sampled from $Q_j(\cdot \mid u)$ with (or without) replacement. The **(Trainable)** $Q_\phi(j)$ allocates the total budget of 5 among the heuristics where heuristic j can select $b_j = Q_\phi(j) \times 5$. **(top right)** Finally, the sampled edges get combined into sampled subgraph. The discrete process is relaxed in §3.4.

assigns it weight ≈ 0. Second, we cluster edges using features of its connecting endpoint nodes. Our method can choose which cluster to sample more frequently, generally increasing model performance. Finally, we show how our framework can be integrated into full-graph decoupled GNN methods (D.2) This allows us to raise the SOTA performance on ogbn-arxiv from 77.98% to 82.26% and performance on ogbn-products from 87.98% to 88.59%, when comparing only models that do not use external features. The performance gains introduce no additional GNN parameters.

2 PRELIMINARIES

Let $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{A}, \mathbf{X})$ denote a graph with node set $\mathbf{V}$, edge set $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$, adjacency matrix $\mathbf{A} \in \{0, 1\}^{|\mathbf{V}| \times |\mathbf{V}|}$, and node feature matrix $\mathbf{X} \in \mathbb{R}^{|\mathbf{V}| \times d}$ containing a d-dimensional feature vector for node $u \in \mathbf{V}$ at row $\mathbf{X}_u$. If the directed edge $u \to v$ exists then $(u, v) \in \mathbf{E}$ and $\mathbf{A}_{uv} = 1$. We denote the degree of u as $\delta_u = \sum_v \mathbf{A}_{uv}$. Let $\mathbf{D}$ be the degree matrix with entries $\mathbf{D}_{uu} = \delta_u$. If the graph is undirected, *i.e.*, $\mathbf{A}$ is symmetric, then let $\widehat{\mathbf{A}} \in \mathbb{R}^{|\mathbf{V}| \times |\mathbf{V}|}$ denote the symmetrically normalized $\mathbf{A}$ [Kipf and Welling, 2017]: $\widehat{\mathbf{A}} = (\mathbf{D}+\mathbf{I})^{-\frac{1}{2}}(\mathbf{A}+\mathbf{I})(\mathbf{D}+\mathbf{I})^{-\frac{1}{2}}$. Let $\mathcal{N}(u)$ denote the set of out-neighbors of u with $\mathcal{N}(u) = \{v \mid (u, v) \in \mathbf{E}\}$. We apply our method to node classification tasks. Let $\mathbf{V}_{\text{tr}} \subset \mathbf{V}$ denote the set of training labeled nodes. For each $u \in \mathbf{V}_{\text{tr}}$, let y_u denote its label. Finally, we denote the output of a GNN on graph $\mathbf{G}$ as $h_\theta(\mathbf{G}) \triangleq h_\theta(\mathbf{A}, \mathbf{X})$.

2.1 GRAPH NEURAL NETWORKS

Graph Neural Networks (GNNs), as the name suggests, are neural networks that operate on graphs. In their earliest inceptions, they were written down in terms of the full graph. There are many variants of GNNs, including ones that incorporate multiple hops [Abu-El-Haija et al., 2019], encode distances [Li et al., 2020, Zhang and Chen, 2018], add skip-connection between layers [Xu et al., 2018, Pham et al., 2017, Chen et al., 2020], use multilayer perceptrons (MLPs) between hops [Xu et al., 2019], among many others—see [Chami et al., 2022] for a comprehensive survey. Nonetheless, we emphasize that while GNNs are central to our work, they are not our main contribution: we use them as a "black box". It suffices to know that GNNs compute

$$h_\theta : \mathbb{R}^{|\mathbf{V}| \times |\mathbf{V}|} \times \mathbb{R}^{|\mathbf{V}| \times d} \to \mathbb{R}^{|\mathbf{V}| \times z}. \quad (1)$$

Specifically, a GNN has trainable parameters θ and can map adjacency $\mathbf{A}$ and node-features $\mathbf{X}$, as $h_\theta(\mathbf{A}, \mathbf{X})$, onto z dimensions per node. For example, in node classification tasks z can be the number of classes. In our work, we use two GNN variants: the *Graph Convolutional Network* (GCN) of Kipf and Welling [2017] and EnGCN by Duan et al. [2022].

GCN with 3 layers can be written as

$$h_\theta(\mathbf{A}, \mathbf{X}) = \sigma\left(\widehat{\mathbf{A}}\sigma\left(\widehat{\mathbf{A}}\sigma\left(\widehat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(1)}\right)\mathbf{W}^{(2)}\right)\mathbf{W}^{(3)}\right) \tag{2}$$

with $\theta = \left\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(3)}\right\}$ and $\sigma(\cdot)$ is the element-wise non-linearity.

2.2 LEVEL-BASED SUBGRAPH SAMPLING

Rather than invoking training and inference on the graph as a whole, as described by earlier GNN formulations [Kipf and Welling, 2017, Veličković et al., 2018, Abu-El-Haija et al., 2019, Xu et al., 2019, , *etc*], instead, many sampling-based methods feed *subgraph samples* into the GNN (§2.1) for training and inference. The sampling can be described as a generative process. First, sample seed nodes $V^{(0)} = \{v_1^{(0)}, v_2^{(0)}, \dots\}$ with $V^{(0)} \subseteq \mathbf{V}_{\text{tr}}$ and initialize $E^{(0)} = \{\}$ as an empty set. Then, sample neighbors for every node in $V^{(0)}$. Let $V^{(1)}$ denote the set of all sampled neighbors. The process can be repeated to construct $V^{(2)}$ containing sampled neighbors for every $v \in V^{(1)}$, and so on, until a desired sampling depth is reached. This can be described by an iterative process:

$$\textit{(sample seed)} \quad V^{(0)} \leftarrow s(\mathbf{V}_{\text{tr}}), \tag{3}$$

$$\textit{(sample edges)} \quad E^{(i+1)} \leftarrow \bigcup_{v \in V^{(i)}} s(\mathcal{N}(v)), \tag{4}$$

$$\textit{(collect nodes)} \quad V^{(i+1)} \leftarrow \left\{v \mid (u, v) \in E^{(i+1)}\right\}, \tag{5}$$

$$\textit{(subgraph sample)} \quad \widetilde{\mathbf{G}} \leftarrow \left(\bigcup_i V^{(i)}, \bigcup_i E^{(i)}\right), \tag{6}$$

where Eqs. (4) & (5) are repeated, until a desired sampling depth is reached. Function $s(\cdot)$ outputs a random subset of its argument and $\widetilde{\mathbf{G}}$ denotes the final subgraph sample. Then, one can run $h_\theta(\widetilde{\mathbf{G}})$ for training or inference on $\widetilde{\mathbf{G}}$.

Since s makes random choices, $\widetilde{\mathbf{G}}$ is a random variable $\widetilde{\mathbf{G}} \sim \mathbb{P}(\mathcal{G})$. The work of Hamilton et al. [2017], Ying et al. [2018], Markowitz et al. [2021], Ferludin et al. [2022] produce tree-samples, with each tree rooted at a seed node $\in V^{(0)}$. The choice of $s(\cdot)$ varies per method. For instance, Markowitz et al. [2021] set $s(\cdot)$ to uniform (all neighbors have equal probability to be sampled) while Chen et al. [2018], Zeng et al. [2020], Zou et al. [2019] define $s(\cdot)$ such that each node will be sampled with probability related to its degree.

3 PROBABILISTIC MODELS OF EDGES

We would like to no longer assume that the process for generating $\widetilde{\mathbf{G}}$ is fixed. Instead, we set it to a learnable convex combination of heuristics. Section §3.1 shows how we parameterize the process and how we sample discrete subgraphs $\widetilde{\mathbf{G}}$. Then, §3.4 shows how to learn the subgraph sampling process.

We position our contribution by recalling empirical risk minimization (ERM):

$$\min_\theta \mathcal{R}(\theta) = \min_\theta \mathop{\mathbb{E}}_{\widetilde{\mathbf{G}} \sim \mathbb{P}(\mathcal{G})} \mathcal{L}\left(h_\theta(\widetilde{\mathbf{G}}), y(\widetilde{\mathbf{G}})\right), \tag{7}$$

where $\mathcal{L}$ is a loss function (we use cross-entropy for classification) between desired labels y and output of GNN h with current parameters θ. $\mathbb{P}$ can be characterized per Eqs. 3–6.

Our primary goal is to parameterize the sampling procedure $\mathbb{P}$ and learn it. We write down our modified ERM:

$$\min_{\theta,\phi} \mathcal{R}(\theta, \phi) = \min_{\theta,\phi} \mathop{\mathbb{E}}_{\widetilde{\mathbf{G}} \sim \mathbb{P}_\phi(\mathcal{G})} \mathcal{L}\left(h_\theta(\widetilde{\mathbf{G}}), y(\widetilde{\mathbf{G}})\right) \tag{8}$$

where ϕ denotes sampling parameters. Note that ϕ is not part of the GNN model h, but only part of the subgraph sampling procedure.

3.1 SUBGRAPH SAMPLING ($\widetilde{\mathbf{G}}$)

Given the sampling process described in §2.2 (Equations 3–6), $\mathbb{P}$ can be written as product across sampling step i:

$$\mathbb{P}_\phi(\mathcal{G}) = P(V^{(0)}) \prod_i P_\phi(E^{(i+1)} \mid V^{(i)}) P(V^{(i)} \mid E^{(i)})$$

$$\triangleq P(V^{(0)}) \prod_i P_\phi(E^{(i+1)} \mid V^{(i)}). \tag{9}$$

Note that we intentionally skip all $P(V^{(i)} \mid E^{(i)})$: once $E^{(i)}$ is sampled, then $V^{(i)}$ becomes deterministic. Further, we emphasize that in our work, $P(V^{(0)})$ is not parameterized. Rather, it is set to the uniform distribution, *i.e.*, we sample batches of seed nodes uniformly at random.

Equations 4 and 5 sample $V^{(i+1)}$ given $V^{(i)}$. Therefore, the choice of $s(\cdot)$ specifies the conditional $P(E^{(i+1)} \mid V^{(i)})$. While earlier subgraph sampling methods implicitly set $P(E^{(i+1)} \mid V^{(i)})$ to a fixed distribution[1], we define it as a mixture:

$$P_\phi(E^{(i+1)} \mid V^{(i)}) = \sum_{j \le J} Q_\phi(E^{(i+1)}, C^{(i)}{=}j \mid V^{(i)}) \tag{10}$$

$$= \sum_{j \le J} Q(E^{^{(i+1)}} \mid V^{(i)}, C^{(i)}{=}j) Q_\phi(C^{(i)}{=}j \mid V^{(i)}) \tag{11}$$

$$= \mathbb{E}_{j \sim Q_\phi}\left[Q_j(E^{^{(i+1)}} \mid V^{(i)})\right], \tag{12}$$

where j indexes J mixture components. The first term of Eq. 11 denotes the *component*, abbrev: $Q_j(E^{^{(i+1)}} \mid V^{(i)}) = Q(E^{^{(i+1)}} \mid V^{(i)}, C^{(i)} = j)$, and the second term is the component *weight*. In our present work, only Q_ϕ contains

[1]One can write down $P(E^{(i+1)} \mid V^{(i)})$ for Hamilton et al. [2017], Markowitz et al. [2021]. Suppose $V^{(i)} = \{u_1\}$ and if s is uniformly sample $k \le |\mathcal{N}(u_1)|$ entries with equal probability, then $P(E^{(i+1)} \mid V^{(i)}) = \frac{1}{\binom{|\mathcal{N}(u_1)|}{k}}$ iff $E^{(i+1)} \subseteq \mathcal{N}(u_1)$ and $|E^{(i+1)}| = k$, *i.e.*, k-sized subset of u_1's neighbors. Equivalently, each edge (u_1, v) appears with probability $= \frac{k}{|\mathcal{N}(u_1)|}$.

trainable parameters ϕ but Q_j's are fixed heuristics. Sections §3.2 and §3.3 give examples for these.

We consider a few heuristics Q_j. Most of these can be factorized w.r.t. all input nodes as

$$Q_j(E^{(i+1)} \mid V^{(i)}) = \prod_{(u,v)\in E^{(i+1)}} Q_j(v \mid u), \qquad (13)$$

where $Q_j(v \mid u)$ equals the probability of choosing neighbor v from $\mathcal{N}(u)$, as assigned by the jth heuristic.

The Appendix also considers the case where Q_j's only factorize on $V^{(i)}$ but not on $E^{(i+1)}$, yielding Q_j with **support on subsets** $\subseteq \bigcup_{u\in V^{(i)}} \mathcal{N}(u)$.

3.2 PARAMETERIZING WEIGHTS Q_ϕ

To parameterize Q_ϕ, we try-out two versions, each making its own assumptions. The first version assumes independence: The distribution Q_ϕ is independent from node u that we seek samples for. The second version makes the Markov-chain assumption: Q_ϕ is conditioned on the heuristic that gave rise to u, but not u itself. With these assumptions, we parameterize Q_ϕ as:

$$Q_\phi(C^{(0)}{=}j) = \text{softmax}(\mathbf{c})_j \triangleq c_j \qquad (14)$$

$$Q_\phi(C^{(i+1)}{=}j \mid C^{(i)}{=}k) = \text{softmax}(\mathbf{c}_k)_j \triangleq c_{jk} \qquad (15)$$

where $\mathbf{c}, \mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_J \in \mathbb{R}^J$ are the trainable distribution parameters, and J denotes the number of components. In this case sampling parameters ϕ are:

$$\phi = \{\mathbf{c}, \mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_J\}. \qquad (16)$$

When operating under the independence assumption, $Q_\phi(C^{(i)}) = Q_\phi(C^{(0)}) = \mathbf{c}$ and therefore $\phi = \{\mathbf{c}\}$. Modeling parameters in Eq. 15 are needed when making Markov-chain assumption. In general, nth order Markov-chain would require $\mathcal{O}(J^{1+n})$ parameters.

3.3 PARAMETERIZING COMPONENTS Q_j

While component weights are trainable (§3.2), the components Q_j's are **fixed sampling heuristics**:

- Random sampler (Q_{RAND}): given node u, return subset of its neighbors $\mathcal{N}(u)$ uniformly at random.
- Top-degree sampler (Q_{TOPDEG}): samples neighbor with probability increasing with its degree.
$$Q_{\text{TOPDEG}}(v \mid u) \propto \mathbf{A}_{uv} \log(\alpha + \delta_v). \qquad (17)$$
Here and below, we tried $\alpha = 1$ and $\alpha = 2$, without noticing any significant difference in performance. In practice, we tried sampling with and without replacement, and the results are almost identical.
- Least-degree sampler (Q_{LOWDEG}): samples neighbor with probability decreasing with its degree.
$$Q_{\text{LOWDEG}}(v \mid u) \propto \mathbf{A}_{uv} \log(\alpha + \delta_{\max} - \delta_v), \qquad (18)$$
where the maximum node degree $\delta_{\max} = \max_{u\in\mathbf{V}} \delta_u$.
- PageRank (Q_{PAGERANK}): samples neighbor with probability increasing with its PageRank [Page et al., 1999].
$$Q_{\text{PAGERANK}}(v \mid u) \propto \mathbf{A}_{uv} \log(\alpha + \text{PageRank}(v)). \qquad (19)$$
- Self-sampler (Q_{SELF}) that samples only itself:
$$Q_{\text{SELF}}(v \mid u) = 1 \ \ \textbf{iff} \ \ v = u. \qquad (20)$$
- Subset-based samplers, ones assigning probability to edge-sets, rather than one-node-at-a-time, including:
$$Q_{\text{MAXCOVERX}}(E^{(i+1)} \mid V^{(i)})$$
$$Q_{\text{MAXCOVERA}}(E^{(i+1)} \mid V^{(i)})$$
$$Q_{\text{MAXCOVERAX}}(E^{(i+1)} \mid V^{(i)})$$
for choosing edge-sets with destination nodes that are far-apart, respectively, in the feature-space, the graph topology, or a mixture of the two. For details see the Appendix.

Now that we have specified how we design Q_ϕ and Q_j, the next steps include determining how to obtain $\nabla_\phi \mathcal{R}$.

3.4 LEARNING ϕ WHEN SUBGRAPH SAMPLING

A routine for level-based subgraph sampling typically takes two sets of hyperparameters:

- Seed nodes $V^{(0)} \subseteq \mathbf{V}_{\text{tr}}$, as discussed earlier.
- Sampling budgets $b^{(1)}, b^{(2)}, \dots \in \mathbb{Z}_+$.

Then, the routine should, for each $u \in V^{(0)}$, sample $b^{(1)}$ of its neighbors, and for each of those neighbors, sample $b^{(2)}$ of its neighbors, and so on, until graph $\widetilde{\mathbf{G}}$ is composed.

For the sake of demonstration, consider only two samplers $[Q_{\text{LowDeg}}, Q_{\text{TopDeg}}]$ and a first hop sampling budget of $b^{(1)} = 20$. Also, suppose that the sampling budget is split with proportion $\mathbf{c} = [0.2, 0.8]$ (defined in Eq.14). Then, to sample $E^{(1)} \sim P_\phi(E^{(1)} \mid V^{(0)})$, one can:

sample 4=0.2(20) $= c_1 \times b^{(1)} \triangleq b_1^{(1)}$ edges from Q_{LowDeg};

sample 16=0.8(20) $= c_2 \times b^{(1)} \triangleq b_2^{(1)}$ edges from Q_{TopDeg}.

In general, sampler Q_j gets a first-hop budget of:

$$b_j^{(1)} = c_j \times b^{(1)}. \qquad (21)$$

Sampler Q_j can double its own $b_j^{(1)}$ by doubling the learnable $c_j = \text{softmax}(\mathbf{c})_j$, where $\mathbf{c} \in \phi$, *cf.* Eq.14. Subsequently, at the ith-hop, when sampling edges for $u \in V^{(i)}$, Q_j receives budget

$$b_j^{(i)} = c_{jk} \times b^{(i)} \qquad (22)$$

where $k \in \{1, \dots, J\}$ is the **mixture identity** that sampled

$u \in V^{(i-1)}$ *i.e.*, u was sampled from Q_k (according to Markovian assumption #2, Appendix §**??**)

In summary, training ϕ determines the partitioning of (integral[2]) budgets, among J samplers, which compete for budget via softmax transformation.

3.4.1 How are $b_j^{(i)}$ utilized in GNN h?

From our parameters ϕ we can obtain $b_j^{(i)}$. On the other hand, the objective function $\mathcal{L}$ is measured on GNN h. To reason about how ϕ can be learned, we'd like to assess the influence[3] of $b_j^{(i)}$ on $\mathcal{L}$. The GNN model §2.1 can operate on the adjacency $\ddot{\mathbf{A}}$ corresponding to sampled subgraph $\widetilde{\mathbf{G}}$. One can express $\ddot{\mathbf{A}}$ in terms of $b_j^{(i)}$. Let $e_j^{(i)}(u)$ be a re-ordering of $\mathcal{N}(u)$, that was sampled at step i, by repeatedly selecting from $\sim Q_j(\cdot \mid u)$, with or without replacement, until $\mathcal{N}(u)$ is exhausted. With this, we can write:

$$E^{(i)} = \bigcup_{j \in [J]} e_j^{(i)}(u)_{[:b_j^{(i)}]} \tag{23}$$

Where notation $[: b_j^{(j)}]$ selects the first $b_j^{(j)}$ of the ordered sequence. Entry (u, v) of the adjacency matrix can now be defined piece-wise:

$$\ddot{\mathbf{A}}_{uv} = \begin{cases} 1, & \textbf{if } (u,v) \in \bigcup_i E^{(i)} \\ 0, & \textbf{otherwise.} \end{cases} \tag{24}$$

The order of elements in $e_j^{(i)}(u)$ is always respected – only the first elements are chosen. This allows for expressing sophisticated sampling heuristics that cannot be factorized per edge, for example based on maximum coverage, mutual information or other monotone submodular functions, a class of functions widely used for data sampling and summarization. For this class of functions, the well known Greedy algorithm (Nemhauser and Wolsey [1978], Minoux [2005], Mirzasoleiman et al. [2015]) produces an ordering in which every prefix of size k $(1 - 1/e)$-approximates the size-k subset of *maximum* objective value. Therefore, by producing an ordering based on such greedy algorithms we can capture a variety of data sampling methods with implicit $Q_j(\cdot \mid u)$.

3.4.2 Backpropagation from h to ϕ

GNN h is a function of $\ddot{\mathbf{A}}$. The information flows from ϕ to the GNN h as $\phi \rightarrow \{b_j^{(i)}\}_{i,j} \rightarrow \{E^{(i)}\}_i \rightarrow \ddot{\mathbf{A}} \rightarrow h$. Therefore, then it is natural to assess how to compute $\frac{\partial \mathcal{L}}{\partial \phi}$ from $\frac{\partial \mathcal{L}}{\partial h}$, *e.g.*, by backpropagation. However, this is not trivial. In particular, $\{E^{(i)}\}_i$ appear only in the branching condition for piecewise $\ddot{\mathbf{A}}$ (Eq. 24). As such, $\ddot{\mathbf{A}}$ is not continuously differentiable *w.r.t.* $b^{(i)} \in \mathbb{Z}_+$. Hence, the Jacobians $\left\{\frac{\partial A}{\partial \mathbf{c}}\right\}_{\mathbf{c} \in \phi}$ and therefore $\nabla_\phi \mathcal{R}$ are uninformative to compute.

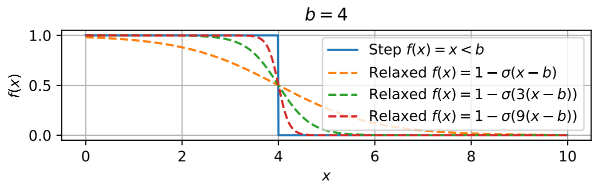


Figure 2: **Continous Relaxation**. For budget $b = 4$, solid line shows discrete step function, whereas dashed-lines show our continuous relaxation (Eq. 25) each dashed line with a choice of sharpness constant β. As β increases, the continuous relaxation becomes closer to the discrete step function.

Table 1: **Datasets**.

Dataset	Nodes	Edges
Citeseer	3,327 articles	4,732 citations
Cora	2,708 articles	5,429 citations
Pubmed	19,717 articles	44,338 citations
ogbn-arxiv	169,343 papers	1.2M citations
ogbn-products	2.5M products	61.9M co-purchases

We derive a continuous relaxation $\dddot{\mathbf{A}}$ of $\ddot{\mathbf{A}}$, such that the **values** of $\dddot{\mathbf{A}}$ are a function of b's and therefore $\mathbf{c}$'s. Moreover, $\dddot{\mathbf{A}}$ looks similar to piecewise $\ddot{\mathbf{A}}$. Row $\dddot{\mathbf{A}}_u^{(j)}$ of Eq. 24, when re-ordered by $e_j(u)$ can be plotted as a step function, see Fig. 2. To obtain a differentiable function, we propose to relax the stepwise selection with:

$$\dddot{\mathbf{A}}_{uv}^{(j)} = 1 - \sigma(\beta(r_{juv} - b_1^{(j)})), \tag{25}$$

where r_{juv} equals the position r for which $q_j(u)_r = v$, $\sigma(x) = (1 + e^{-x})^{-1}$, and β is a sharpness hyperparameter that we set $\beta = 2$. Re-ordering rows of $\dddot{\mathbf{A}}_{uv}^{(j)}$ according to $e_j(u)$ gives a reasonable approximation of the step function that is continuously-differentiable *w.r.t.* $\mathbf{c}$. See Fig. 2 for visual comparison between Eq. 24 & 25. Subsequently, we can calculate $\nabla_\phi \mathcal{R}$. Constructing $\dddot{\mathbf{A}}$ is detailed in Alg. 1.

4 EXPERIMENTAL EVALUATION

We conduct experiments using our method on five popular node classification datasets, summarized in Table 1. We utilize three datasets open-sourced by Yang et al. [2016]: **citeseer**, **cora** and **pubmed**, partially-popularized by Kipf and Welling [2017]. However, we use a larger training set (we include both training and validation into training) – as such we train our baselines. In addition, we use two datasets by OGB [Hu et al., 2020] for node classification: **ogbn-arxiv** and **ogbn-products**. For OGB datasets, we use the official train:validate:test partitions.

[2] $b_j^{(i)}$'s are not integral but one can do randomized rounding, *e.g.*, draw $\mathbf{Y} \sim$ Bernoulli$(b_j^{(i)} - \lfloor b_j^{(i)} \rfloor)$ then $b_j^{(i)} \leftarrow \lfloor b_j^{(i)} + \mathbf{Y} \rfloor$.

[3] Influence should increase with $\frac{\partial \mathcal{L}}{\partial b_j^{(i)}}$.

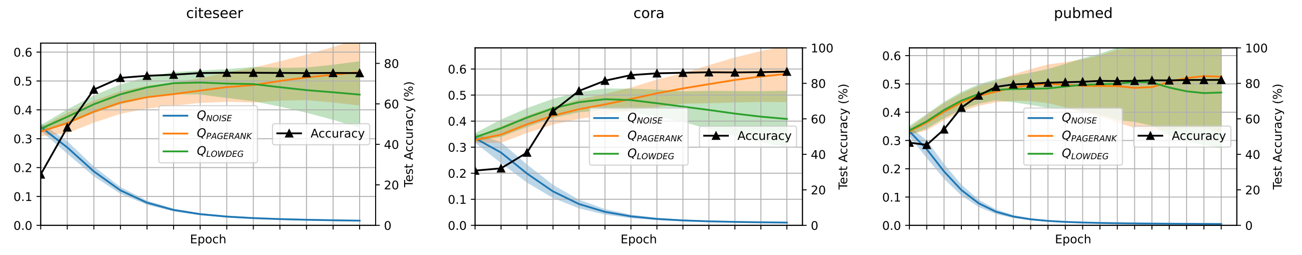


Figure 3: **Robust against noise edge sources**. Heuristic sampling negative edges gets ignored when learning ϕ (see §4.1).

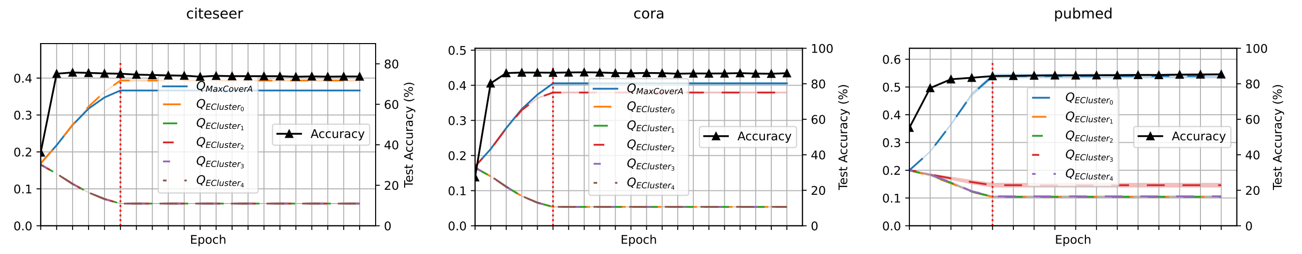


Figure 4: Q_ϕ distribution among edge clusters: each Q_{RAND} contains about $\frac{1}{5}$ of the edges. We freeze the sampling parameters ϕ after 25% of training epochs. For **pubmed**, it is sufficient to train on only $\frac{1}{5}$ of the edges while for **citeseer** and **cora** we supplemented with all graph edges but sampled according to greedy algorithm for max graph coverage, per Appendix.

Table 2: Test accuracy of GNN models trained with **(left)** subgraphs sampled uniformly at random, against **(right)** subgraphs from clustered edge sources, per §4.2

	GCN($\widetilde{\mathbf{G}}$)	
Dataset	**baseline**: $\widetilde{\mathbf{G}} \sim \mathbb{P}$	**our-**§4.2: $\widetilde{\mathbf{G}} \sim \mathbb{P}_\theta$
Citeseer	74.08 ± 0.47	73.73 ± 0.50
Cora	84.74 ± 0.51	85.79 ± 0.64
Pubmed	83.90 ± 0.16	85.26 ± 0.29

Table 3: **Node Classification on OGBN leaderboard**. Methods (a) use external data: text of abstract or product description, resp., for ogbn-arxiv and ogbn-products. Methods (b) use only provided feature vectors. Last line is ours.

	Method	ogbn-arxiv	ogbn-products
(a)	GLEM+EnGCN	79.66 ± 0.06	90.14 ± 0.12
	GIANT++	76.37 ± 0.11	86.84 ± 0.05
(b)	EnGCN	77.98 ± 0.07	87.98 ± 0.04
	DRGAT	74.16 ± 0.07	82.30
	DRGAT++	76.33 ± 0.08	–
	GAMPL++	–	85.20 ± 0.08
	Ours: EnGCN($\ddot{\mathbf{A}}$)	82.26 ± 0.09	88.59 ± 0.07

Algorithm 1 Samples sparse $\ddot{\mathbf{A}}$ with continuous relaxation

```
1: Input: Seed: $V^{(0)}$; sampler heuristics $\{q_j\}_{j \le J}$; sampling depth: $B$, per-step sampling budgets: $[b_1, \dots, b_S]$;
2: Output: Adjacency $\ddot{\mathbf{A}}$ of sampled subgraph.
3: $\widetilde{V}^{(0)} \leftarrow [(\texttt{null}, u)$ for $u \in V^{(0)}]$
4: for $i \leftarrow 1$ to S do
5:     $\widetilde{V}^{(i)} \leftarrow [\,]$
6:     for $(k, u)$ in $\widetilde{V}^{(i-1)}$ do
7:         for $j \leftarrow 1$ to $J$ do
8:             for $r \leftarrow 1$ to $b_i$ do
9:                 $v \leftarrow q_j(u)[r]$
10:                $\widetilde{V}^{(i)}\texttt{.append}((j, v))$
11:                $b_j^{(i)} \leftarrow c_{kj} \times b^{(i)}$
12:                $\ddot{\mathbf{A}}_{uv} \leftarrow \ddot{\mathbf{A}}_{uv} + 1 - \sigma(\beta(r - b_i^{(j)}))$
13:            end for
14:        end for
15:    end for
16: end for
```

4.1 AGAINST ADVERSARIAL EDGE SOURCES

In these experiments, we assume that we have one $Q_{\text{NOISE}}(\cdot|u)$ that has uniform probability over all V, *i.e.*, it is very likely that $v \notin \mathcal{N}(u)$ for $v \sim Q_{\text{NOISE}}(\cdot|u)$. Since Q_{NOISE} favors negative edges, we would hope that our learning algorithm is able to down-weigh Q_{NOISE}. Specifically, we hope to learn $\phi = \{\mathbf{c}\}$ with $c_{\text{NOISE}} \approx 0$.

Setup: We train GCN of [Kipf and Welling, 2017], with training examples being subgraphs sampled according to §3.1&§3.4, with sampling budgets $b^{(1)} = 20, b^{(2)} = 10$. We use: $\{Q_j\}_j = \{Q_{\text{PageRank}}, Q_{\text{LowDeg}}, Q_{\text{Noise}}\}$. We attempt Datasets: **citeseer**, **cora**, **pubmed**.

Discussion: We see that, as summarized in Fig. 3, subgraph sampling parameters ϕ learn to ignore edges coming from Q_{Noise}. In all cases, c_{Noise} goes towards 0. We run each experiment ten times, and plot the average Q_ϕ through time, shading up-to the standard deviation.

Hyperparameters: We use GCN model of [Kipf and Welling, 2017] with default hyperparameters, *i.e.*, two GCN layers, hidden dimension of 32, trained with dropout and Adam optimizer using learning rate of 0.01 (but 0.005 for **pubmed**). The learning rate for ϕ parameters is set to 0.05.

4.2 EMPHASIZING FAVORITE EDGES

We test if our method can choose edges that are preferred for the node classification task. Specifically, we:

1. Partition edges $\mathbf{E}$ into $\mathbf{E}^{\text{p}}_1, \mathbf{E}^{\text{p}}_2, \mathbf{E}^{\text{p}}_3, \mathbf{E}^{\text{p}}_4, \mathbf{E}^{\text{p}}_5$ – we experiment only with 5 partitions. The partition is based on the features of edge endpoints (details below).
2. We instantiate Q_{ECluster_m} for partition $\mathbf{E}^{\text{p}}_m$ with:
$$Q_{\text{ECluster}_m}(v \mid u) = \frac{1}{|(\tilde{v}, u) \in \mathbf{E}^{\text{p}}_m|} \mathbb{1}[(u, v) \in \mathbf{E}^{\text{p}}_m],$$
where $\mathbb{1}[\cdot]$ is the indicator function, evaluating to 1 if its argument is true and otherwise to 0. Our learning algorithm should pick among the 5 edge clusters: if $\phi = \{\mathbf{c}\}$ was learned, such that, $c_m > c_\ell$, then, $\mathbf{E}^{\text{p}}_m$ contains **better edges** than $\mathbf{E}^{\text{p}}_\ell$, according to the objective function.

Partitioning algorithm: For each edge $(u, v) \in \mathbf{E}$, calculate edge feature $\mathbf{X}_{vu} = \mathbf{X}_{uv} \in \mathbb{R}^d$ as the Hadamard product of node features: $\mathbf{X}_{uv} = \mathbf{X}_u \circ \mathbf{X}_v$. Then, we run k-means on $\{\mathbf{X}_{uv}\}_{(u,v)\in\mathbf{E}}$ with $k = 5$. If k-means assigns cluster m to feature vector $\mathbf{X}_{uv}$, then edge (u, v) will be in $\mathbf{E}^{\text{p}}_m$. Note that $\mathbf{E} = \cup_m \mathbf{E}^{\text{p}}_m$ and that $\mathbf{E}^{\text{p}}_m \cap \mathbf{E}^{\text{p}}_\ell = \emptyset$ for all $m \neq \ell$.

Datasets and Hyperparameters: same as §4.1.

Setup We compare two GCNs, both of them are trained with sampling budgets $b^{(1)} = 20, b^{(2)} = 10$. The first GCN is **baseline**, where training examples are subgraphs sampled uniformly at random *i.e.*, only Q_{ECluster} is used: given a seed node, a random subtree is sampled by recursively choosing from neighbors with equal probability. The second GCN is **our-**§4.2, where training examples are subgraphs sampled according to §3.1&§3.4. For **pubmed**, we use $\{Q_j\}_j = \{Q_{\text{Rand}^{\text{p}}_m}\}_{m=1}^5$. For **cora** and **citeseer**, we additionally use $Q_{\text{MaxCoverA}}$, which samples edges due to greedy algorithm, as detailed in Appendix.

Discussion: According to Table 2, we see that over-emphasizing edges, based on the cluster of the endpoint features, helps for two of the three datasets. In addition, Fig. 4 summarizes the learned $\mathbf{c} \in \phi$. We see that each graph indeed chooses some edges to be over-repeated (assigning remainder edge sources to ≈ 0). For **pubmed**, we can get reasonable results while **ignoring most graph edges**, *i.e.*, ϕ learns to discard all but one $\mathbf{E}^{\text{p}}$.

5 MELD INTO DECOUPLED GNN

Many methods train node-wise multi-layer perceptron MLP($\mathbf{X}$) : $\mathbb{R}^d \rightarrow \mathbb{R}^z$ mapping node features into classes, where the trainable transformation does **not** use the graph. Instead, the graph is used in a non-trainable **propagation** [Wu et al., 2019, Frasca et al., 2020, Gasteiger et al., 2019a, Duan et al., 2022]. Most recently, EnGCN [Duan et al., 2022] proposes to run the (full-graph) propagation only a few times during training, e.g., once every 100 epochs on node-wise MLP. Our generative framework can be utilized in decoupled GNNs. Since the **propagation** step is allowed to consider the full graph, we define a full-graph adjacency $\widetilde{\mathbf{A}} \in \mathbb{R}^{|V|\times|V|}$, based on §3, as:

$$\widetilde{\mathbf{A}}_{uv} = \mathbb{E}_{j\sim Q_\phi}\left[Q_j(v \mid u)\right] \triangleq \mathbb{E}_{j\sim Q_\phi}\left[\mathbf{A}_{(j)uv}\right] \qquad (26)$$

where $\mathbf{A}_{(j)}$ reshapes Q_j into a matrix.

We extend EnGCN [Duan et al., 2022] with our framework. We use our ϕ-parameterized $\widetilde{\mathbf{A}}$ instead of their proposed propagation matrix of $\widehat{\mathbf{A}}$ – Gasteiger et al. [2019b,a] show other plausible choices for propagation matrix. EnGCN uses propagation as a linear operator:

$$\mathbf{X}^{(0)} \leftarrow \mathbf{X}; \qquad \mathbf{X}^{(i+1)} \leftarrow \widetilde{\mathbf{A}}\mathbf{X}^{(i)} \qquad (27)$$

Naïvely feeding $\mathbf{X}^{(i)}$'s (Eq. 27) into MLP GNN, unfortunately, would cast the decoupled GNNs into graph-consuming variants, as one would have to backpropagate through the adjacency values to get $\nabla_\phi \mathcal{R}$. Nonetheless, we maintain decoupling by considering linearity of expectation and of the propagation operation.

$$\mathbf{X}^{(i+1)} = \widetilde{\mathbf{A}}\mathbf{X}^{(i)} = \mathbb{E}_{j\sim Q_\phi}\left[\mathbf{A}_{(j)}\mathbf{X}^{(i)}\right]. \qquad (28)$$

We then batch-compute: $\mathbf{A}_{(1)}\mathbf{X}^{(i-1)}, \ldots, \mathbf{A}_{(J)}\mathbf{X}^{(i-1)}$. For learning layer (i) and node-wise MLP : $\mathbb{R}^{z_i} \rightarrow \mathbb{R}^{z_{i+1}}$, the input features of MLP can be set to:

$$\mathbf{X}^{(i)} = \sum_{j\leq J} \mathbf{A}_{(j)}\mathbf{X}^{(i-1)} c_j \qquad (29)$$

by reading batch-computed values. Now, obtaining $\nabla_\phi \mathcal{R}$ becomes trivial as $\mathbf{c}$ entries appear in the summation. Yet, the graph transformation (left-multiplying by $\mathbf{A}_{(j)}$) happens in a fixed process, maintaining decoupling.

Unfortunately, we incur a cost for increasing the size of the latent space (linear in $\mathcal{O}(J)$). However, for inference *i.e.*, once $\phi = \{\mathbf{c}\}$ is learned and fixed, one need only store one vector per example.

5.1 EVALUATING OUR DECOUPLED GNNS

Datasets. OGB of Hu et al. [2020] open-source many graph tasks. We run our methods on node classification tasks (OGBN), specifically, homogeneous[4] ones: **ogbn-arxiv** and **ogbn-products**. Baselines. We copy baseline numbers from the OGBN public leaderboard. We choose only top performers from each bucket: (a) using external data and (b) not using external data. Table 3 summarizes the results. Details of baselines can be found in [Sun and Wu, 2020, Zhang et al., 2021a,b, 2023, Zhao et al., 2023]

Our model. we modify the code of EnGCN [Duan et al., 2022] to import our contribution. We replace their propagation operation, as described in this section.We inherit all their hyper-parameters, as detailed in [Duan et al., 2022].

Discussion. Propagation function of decoupled GNNs, such as EnGCN, benefits when mixing heuristics. We set the state-of-the-art (SOTA) on ogbn-arxiv, even when competing with methods that use external data. For ogbn-products, our method is SOTA against methods that use no external data.

6 RELATED WORK

We broadly summarize methods from our directions of interest: subgraph sampling and decoupled GNNs.

Subgraph Sampling: Chiang et al. [2019] partition input graphs into subgrahs using algorithm for graph clustering such as METIS [Karypis and Kumar, 1998]. The subgraphs are computed once and re-used for training. Other methods sample graphs on-the-fly. For instance, Chen et al. [2018] sample nodes independently and an edge is only included if both of its endpoints are sampled. Zou et al. [2019] then propose to do conditional sampling: nodes sampled at $V^{(i)}$ would influence nodes sampled at $V^{(i+1)}$. This conditional assumption is also implied by recursive subgraph sampling, such that of [Hamilton et al., 2017, Ying et al., 2018, Markowitz et al., 2021, Ferludin et al., 2022]. While we fit into the last mentioned models, as our sampling is also tree-based, in addition, we learn $\mathrm{P}(v \mid u)$ for all $(u, v) \in \cup_i E^{(i)}$, as convex combination of $\{Q_j(v \mid u)\}_{j \leq J}$. This allows our method to perform online edge subset selection, where subsets can be pre-computed by clustering.

Learnable sampling. Yoon et al. [2021] also propose to learn sampling parameters using gradients. However, they assume (the original) discrete sampling process and they learn using reinforcement-learning policy gradients. Instead, we obtain analytical gradients by relaxing the discrete process. Wang et al. [2021] also trains a sampler. However, they parameterize their sampler in terms of features. In our case, our sampler mixes a number of heuristics. Our heuristics are based on structural properties (e.g., node degree or PageRank) rather than features.

Decoupled GNNs: One of the earliest decoupled GNNs is SimpleGCN [Wu et al., 2019]. Before learning starts, graph propagation is applied as a pre-processing step. Learning only use the graph-transformed information but without using the graph. On the other hand, Gasteiger et al. [2019a] also decouples the learned parameters from the graph propagation, however, by applying the propagation after the node-level model. While these two mentioned methods apply the fixed graph transformation once (after or before node-level model), EnGCN [Duan et al., 2022] interleaves node-level learning with graph propagation. In this work, we modify EnGCN by replacing its fixed propagation function, by a function that conducts a variety of propagations $\{\mathbf{A}_{(j)}\}$ in parallel, each propagation j according to heuristic $j \in [J]$. We learn convex combination of precomputed J graph-transformed features.

For space constraints, we do not discuss other directions for scaling-up learning of GNNs, including utilizing distributed compute [Lerer et al., 2019] or using historical embeddings [Chen et al., 2017, Fey et al., 2021].

7 CONCLUSION

In this work we introduced a method for learning graph sampling while training Graph Neural Networks (GNNs). Our proposed method parameterizes graph sampling as a convex combination of different heuristics. We apply our method in two popular regimes for learning GNNs: (D.1) training on sampled subgraphs, and (D.2) *decoupled GNN*: graph propagation step is fixed. For (1), we propose a continuous relaxation of the discrete process. We evaluate our method in three scenarios. (i) our method can learn to discard edge sources that are noisy; (ii) if edges are clustered using their endpoint features, then our method can learn favorite edge clusters; finally, (iii) integrating our method with decoupled GNNs (D.2) achieves SOTA results on ogbn-arxiv and ogbn-products.

References

Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Hrayr Harutyunyan, Nazanin Alipourfard, Kristina Lerman, Greg Ver Steeg, and Aram Galstyan. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *International Conference on Machine Learning*, ICML, 2019.

Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, H. Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey R. Allen, Charles Nash, Victo-

[4]Extending to heterogeneous settings is left as future work.

ria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew M. Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arxiv/1806.01261*, 2018.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations*, 2014.

Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy. In *Journal on Machine Learning Research*, 2022.

Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, 2018.

Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. In *International Conference on Machine Learning*, ICML, 2020.

Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

Keyu Duan, Zirui Liu, Peihao Wang, Wenqing Zheng, Kaixiong Zhou, Tianlong Chen, Xia Hu, and Zhangyang Wang. A comprehensive study on large-scale graph training: Benchmarking and rethinking. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

Oleksandr Ferludin, Arno Eigenwillig, Martin Blais, Dustin Zelle, Jan Pfeifer, Alvaro Sanchez-Gonzalez, Sibon Li, Sami Abu-El-Haija, Peter Battaglia, Neslihan Bulut, et al. TF-GNN: Graph neural networks in tensorflow. *arXiv preprint arXiv:2207.03522*, 2022.

M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec. GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning (ICML)*, 2021.

Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Benjamin Chamberlain, Michael Bronstein, and Federico Monti. Sign: Scalable inception graph neural networks. In *ICML 2020 Workshop on Graph Representation Learning and Beyond*, 2020.

Johannes Gasteiger, Aleksandar Bojchevski, and Stephan Günnemann. Combining neural networks with personalized pagerank for classification on graphs. In *International Conference on Learning Representations*, 2019a.

Johannes Gasteiger, Stefan Weißenberger, and Stephan Günnemann. Diffusion improves graph learning. In *Conference on Neural Information Processing Systems*, 2019b.

W. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 2017.

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

George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

Thomas Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. In *The Conference on Systems and Machine Learning*, 2019.

Pan Li, Yanbang Wang, and Jure Leskovec. Distance encoding: Design provably more powerful neural networks for graph representation learning. In *Neural Information Processing Systems*, 2020.

Elan Sopher Markowitz, Keshav Balasubramanian, Mehrnoosh Mirtaheri, Sami Abu-El-Haija, Bryan Perozzi, Greg Ver Steeg, and Aram Galstyan. Graph traversal with tensor functionals: A meta-algorithm for scalable learning. In *International Conference on Learning Representations*, ICLR, 2021.

Michel Minoux. Accelerated greedy algorithms for maximizing submodular set functions. In *Optimization Techniques: Proceedings of the 8th IFIP Conference on Optimization Techniques Würzburg, September 5–9, 1977*, pages 234–243. Springer, 2005.

Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, Amin Karbasi, Jan Vondrák, and Andreas Krause. Lazier than lazy greedy. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

Federico Monti, Michael M. Bronstein, and Xavier Bresson. Geometric matrix completion with recurrent multi-graph neural networks. In *International Conference on Machine Learning*, 2017.

George L Nemhauser and Laurence A Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of operations research*, 3(3): 177–188, 1978.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999. URL `http://ilpubs.stanford.edu:8090/422/`.

Trang Pham, Truyen Tran, Dinh Phung, and Svetha Venkatesh. Column networks for collective classification. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 2485–2491, 2017.

Chuxiong Sun and Guoshi Wu. Adaptive graph diffusion networks with hop-wise attention. *arXiv/2012.15024*, 2020.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Yu Wang, Zhiwei Liu, Ziwei Fan, Lichao Sun, and Philip Yu. Dskreg: Differentiable sampling on knowledge graph for recommendation with relational gnn, 08 2021.

Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International Conference on Machine Learning*, 2019.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, ICML, pages 5453–5462, 2018.

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

Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *International Conference on Machine Learning*, 2016.

Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *ACM SIGKDD international conference on Knowledge discovery and data mining*, 2018.

Minji Yoon, Théophile Gervet, Baoxu Shi, Sufeng Niu, Qi He, and Jaewon Yang. Performance-adaptive sampling strategy towards fast and accurate graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery Data Mining*, KDD '21, pages 2046–2056, 2021.

Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020.

Chenhui Zhang, Yufei He, Yukuo Cen, Zhenyu Hou, and Jie Tang. Improving the training of graph neural networks with consistency regularization, 2021a.

Lei Zhang, Xiaodong Yan, Jianshan He, Ruopeng Li, and Wei Chu. Drgcn: Dynamic evolving initial residual for deep graph convolutional networks. *arXiv preprint arXiv:2302.05083*, 2023.

Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Neural Information Processing Systems*, 2018.

Wentao Zhang, Ziqi Yin, Zeang Sheng, Wen Ouyang, Xiaosen Li, Yangyu Tao, Zhi Yang, and Bin Cui. Graph attention multi-layer perceptron. *arXiv preprint arXiv:2108.10097*, 2021b.

Jianan Zhao, Meng Qu, Chaozhuo Li, Hao Yan, Qian Liu, Rui Li, Xing Xie, and Jian Tang. Learning on large-scale text-attributed graphs via variational inference. In *International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=q0nmYciuuZN`.

Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Few-shot representation learning for out-of-vocabulary words. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2019.

A MAX COVERAGE HEURISTICS

A.1 MAX COVERAGE HEURISTICS

- Features maximum coverage ($Q_{\text{MaxCoverX}}$): given a node, prefers neighbors with complimentary features. Specifically, suppose sampled edge set
$$E_u^{(i+1)} \sim Q_{\text{MaxCoverX}}(E_u^{(i+1)} \mid u) \tag{30}$$
contains edges $E_u^{(i+1)} = \{(u, v_1), (u, v_2), (u, v_3), \dots\}$, then, with high probability (per heuristic), $v_1, v_2, v_3, \dots$ are far-apart, in the feature space. In other words, the pairwise Euclidean distances
$$||\mathbf{X}_{v_1} - \mathbf{X}_{v_2}||, ||\mathbf{X}_{v_1} - \mathbf{X}_{v_3}||, \dots, \tag{31}$$
are large.

  While for the previous Q_j nodes can be sampled independently, the MaxCoverX heuristics (this one, and below) are necessarily set-based, *i.e.*, the inclusion of one node affects the probability of sampling another.

- Graph max coverage ($Q_{\text{MaxCoverA}}$): prefers neighbors with complimentary structural positions. Specifically, sampling $E_u^{(i+1)} \sim Q_{\text{MaxCoverA}}(E_u^{(i+1)} \mid u)$ yields edges $E_u^{(i+1)} = \{(u, v_1), (u, v_2), (u, v_3), \dots\}$, such that, the Euclidean distances
$$||\mathbf{U}_{v_1} - \mathbf{U}_{v_2}||, ||\mathbf{U}_{v_1} - \mathbf{U}_{v_3}||, \dots, \tag{32}$$
are large, where $\mathbf{U}$ is the matrix of orthonormal singularvalues of $\widehat{\mathbf{A}}$.

- Graph & features max cov. ($Q_{\text{MaxCoverAX}}$): prefers neighbors with complimentary neighborhood features. Similar to the above two – sampling $E_u^{(i+1)} \sim Q_{\text{MaxCoverA}}(E_u^{(i+1)} \mid u)$ yields edges $E_u^{(i+1)} = \{(u, v_1), (u, v_2), (u, v_3), \dots\}$, such that, the Euclidean distances
$$||(\widehat{\mathbf{A}}\mathbf{X})_{v_1} - \mathbf{U}_{v_2}||, ||(\widehat{\mathbf{A}}\mathbf{X})_{v_1} - (\widehat{\mathbf{A}}\mathbf{X})_{v_3}||, \dots, \tag{33}$$
are large.

The MaxCover heuristics prefer neighbors that increase diversity. Sampling from $Q_{\text{MaxCoverX}}(E \mid u)$ should yield b neighbors with substantially different features. Sampling from $Q_{\text{MaxCoverA}}(E \mid u)$ should yield neighbors $V \subseteq \mathcal{N}(u)$ with $|V| = b$, such that, V has maximum vertex cover. Finally, sampling from $Q_{\text{MaxCoverAX}}(E \mid u)$ should yield b neighbors with substantially different neighboring features.

In general, algorithms for selecting subset that maximize coverage are NP-hard. However, there are many greedy approximations. Our greedy approximation is as follows:

1. (**Define** H) For MaxCoverX, let $H \leftarrow X$. For MaxCoverAX, let $H \leftarrow \widehat{A}X$. Finally, for MaxCoverA, let $H \leftarrow U\Lambda^{\frac{1}{2}}$ were orthonormal matrix U contain the eigenvectors of $\widehat{A}$ and diagonal matrix Λ contain eigenvalues.
2. (**Cluster** H) Run k-means algorithm on rows of H.
3. (**Round-robin**) Sampling from $Q_{\text{MaxCover}^*}(E \mid u)$ should yield target nodes that round-robin the clusters.