GraphSAINT: Graph SAmpling Based INductive Learning Technique for Large Scale Graphs

Anonymous author(s)

ABSTRACT

Graph Convolutional Networks are powerful tools for learning representations of attributed graphs, thus facilitating classification and clustering tasks. To train GCNs on large scale graphs, state-of-the-art algorithms construct minibatches based on neighbor sampling *across GCN layers*. By changing the view point, we propose a fundamentally different approach for GCN minibatch construction. We claim that the sampling to construct a minibatch should only depend on the nature of the underlying graph data, and can be decoupled from any specific GCN model design. Thus, in our minibatch training, we use appropriate *graph sampling* (rather than layer sampling) algorithms to identify representative subgraphs of the training graph, and build *complete* GCNs on the small sampled subgraphs. The proposed minibatch construction technique can be applied to most GCN variants in the literature. More importantly, it solves the "neighbor explosion" challenge seen in previous works, and enables efficient training of deeper and higher order GCNs. By extensive experiments on large scale graphs, we show superior performance in training accuracy, speed and model design flexibility. Especially for inductive node-classification tasks on large graphs, our method achieves higher accuracy with 2× to 11× training time reduction, compared with state-of-the-art.

KEYWORDS

Deep learning, graph convolutional networks, graph sampling

1 INTRODUCTION

Recently, representation learning on graphs has attracted much attention in the deep learning community. These representations of unstructured data such as graphs, images and texts facilitate classification and clustering. Inspired by Convolutional Neural Networks (CNNs) on grid-like data (i.e., images), Graph Convolutional Networks (GCNs) [11, 16] have been proposed as extensions of CNNs to attributed graphs – graphs with features on every node along with labels. Although state-of-the-art CNNs can be scaled to large sizes (tens or hundreds of layers, hundreds of thousands of input values, with millions of training images) [14], current works on GCNs [9, 10, 12, 13] have only demonstrated success on graphs with training shallow GCN models (2 layers) with relatively small graphs compared to web-scale networks. Scaling GCNs to larger graphs and deeper layers still requires fast alternate training techniques.

For GCNs, the data that need to be gathered to compute one output node comes from its neighbors in the previous layer, each of which in turn, gathers their output from the previous layer, and so on. It is easy to see that the deeper we back track, the more support nodes we need. Essentially, the number of support nodes (and thus the training complexity) can grow exponentially with the GCN depth. To mitigate such "neighbor explosion", state-of-the-art methods [9, 10, 13, 18] perform various *layer sampling* techniques to reduce the number of support nodes. The works in [13, 18] ensure that only a small number of neighbors (typically 10 to 50) are selected to support one node in the next layer. By reusing historical activation values, the training algorithm in [10] further reduces the required number of support nodes in the previous layer to 2. The work in [9] proposes an importance sampling algorithm that makes training empirically faster than [13]. However, the required sampling size in deeper layers to avoid accuracy loss remains unclear. Although these various layer sampling techniques significantly speed up GCN training, for large scale graphs, the computation complexity to evaluate one minibatch may still be exponential with the GCN depth. This makes the development of deep GCN models very expensive.

Present work. We present GraphSAINT (Graph SAmpling based INductive learning Technique), an approach for efficient training of deep GCNs. GraphSAINT is based on a fundamentally different way of minibatch construction. Instead of building a GCN on the complete training graph and sampling across the layers, we sample the training graph first and then build a complete GCN on the sampled subgraph. Our minibatches are thus *graph sampling* based. Intuitively, the graph sampler should select nodes that have strong influence on each other into the same minibatch. This way, when propagating through the GCN layers, the sampled nodes can "support" each other without the need to gather information from nodes outside the batch. The most important benefit of our graph sampling based approach is that we no longer face the challenge of "neighbor explosion" in training, as every GCN we build during training is a small and *complete* GCN.

Several interesting questions emerge with the idea of graph sampling based training. We address each of them in this paper through detailed analysis and experiments.

- **What is an ideal graph sampling algorithm?**
  The sampler should identify small induced subgraphs that capture the characteristics of the training graph. We evaluate several simple yet effective sampling algorithms, and provide intuitions for designing good samplers in the context of GCN training.
- **How should we design the loss function?**
  The minibatch loss over the sampled graph should be redefined, since the GCN output nodes are not i.i.d. sampled. We propose a technique to normalize the minibatch loss, which improves accuracy and robustness of training.
- **What are the variations of GCN models (depth, connections) that can be exploited?**
  The graph sampling based minibatch algorithm makes it feasible to train deep GCN models. We propose a generalized GCN model that captures the flavor of previously proposed GCN variants. We experimentally evaluate the training characteristics of such deep models on large datasets.

Our experimental results on four large real world graphs show that the proposed GraphSAINT framework outperforms strong baselines in both accuracy and training time. Specifically, on the Reddit dataset our training algorithm achieves highest test accuracy with 2.9× less time using a 2-layer GCN, and with 11.1× less time using a 3-layer GCN. We also demonstrate that our method enables large scale training of deeper GCNs. GraphSAINT obtains more accurate 3-layer and 4-layer GCN models without significant increase in training time.

2 RELATED WORK

The proposed GraphSAINT framework is designed to achieve high accuracy while significantly decreasing the training time compared to state-of-the-art, a summary of which is presented below.

A neural network model that extends convolution operation to graph domain was first proposed by the pioneer work in [8]. The works in [11, 16] further speed up graph convolution computation by fast localized filters derived from Chebyshev expansion. However, they target relatively small datasets and thus the training proceeds in the full batch fashion. In order to scale GCN models to large graphs, various layer sampling techniques [9, 10, 12, 13, 18] have been proposed to enable minibatch training. All of these methods follow the three meta steps: (1) Construct a complete GCN on the full training graph. (2) Construct minibatches by sampling nodes in each layer. (3) Propagate forward and backward among the sampled GCN nodes. Step (1) is executed once before training starts, and steps (2) and (3) proceed iteratively to update the weights using stochastic gradient descend. The layer sampling algorithm proposed by GraphSAGE [13] performs uniform node sampling on the neighbor nodes of the previous layer. It enforces a pre-defined budget to sample neighbors, so as to bound the complexity of minibatch evaluation. The work in [18] enhances the layer sampling algorithm of [13] by introducing an importance score to each neighbor node. The algorithm presumably leads to less information loss due to the bias towards influential neighbors. To further restrict the neighbor size when propagating down the layers, the techniques proposed by [10] guarantee that only two neighbors are needed to support the computation of next layer activations. The idea is to use the historical activation values in the previous layer to avoid re-evaluation of these neighbor nodes. The work in [9] performs sampling from another perspective. Instead of neighbors of nodes between the inter-layer connections to identify the sampling population, node sampling is performed independently for each layer, and then the inter-layer edges are connected among the selected nodes. Empirically, their method leads to less support nodes (and thus faster training) than [13]. It is still unclear what the required sampling size is for deeper layers to avoid accuracy loss. On the other had, we propose a different perspective by sampling the graph first and then constructing full GCN based on it.

There are other efforts that speeds up large scale training by optimizing GPU/CPU performance. [18] proposes a producer-consumer model that balances the CPU and GPU workload, so as to maximize the overall GPU utilization. [19] proposes using graph sampling instead of layer sampling, their focus is to design parallelization techniques to improve training performance on tightly coupled GPU/multi-core CPUs. [12] proposes a "subgraph training" method which ensures that the minibatch data fits into GPU memory. Note that their notion of "subgraph" differs from that presented here as these subgraphs are essentially induced from all the support nodes for a given minibatch *after layer sampling*. Since such sampling algorithm is similar to [13], the problem of "neighbor explosion" is still likely to happen, and so the subgraph size is limited by a predefined threshold. On the other hand, in our approach, all nodes in the subgraph are considered to be in the minibatch and the support nodes are constructed from within the subgraph. Further, we also perform a detailed analysis of the choice of subgraph sampling which has not been addressed in the literature.

Apart from the above techniques to optimize scalability, there have been several successful exploration in improving the GCN model. Based on the localized spectral filter designs [11, 16] which average the neighbor activations, [13] proposes two other types of aggregators that perform various non-linear mappings on the neighbor activations. [12] proposes a variant of max-pooling operator which selects the k-top values of the neighbor features. Such pooling operation leads to a GCN design which incorporates regular convolution on structured data.

The proposed GraphSAINT framework consists of innovations along two orthogonal directions. It investigates an alternative way of minibatch construction based on graph sampling. This enables fast training of deep GCN, and thus leads to a wealth of exploration in GCN architecture as well.

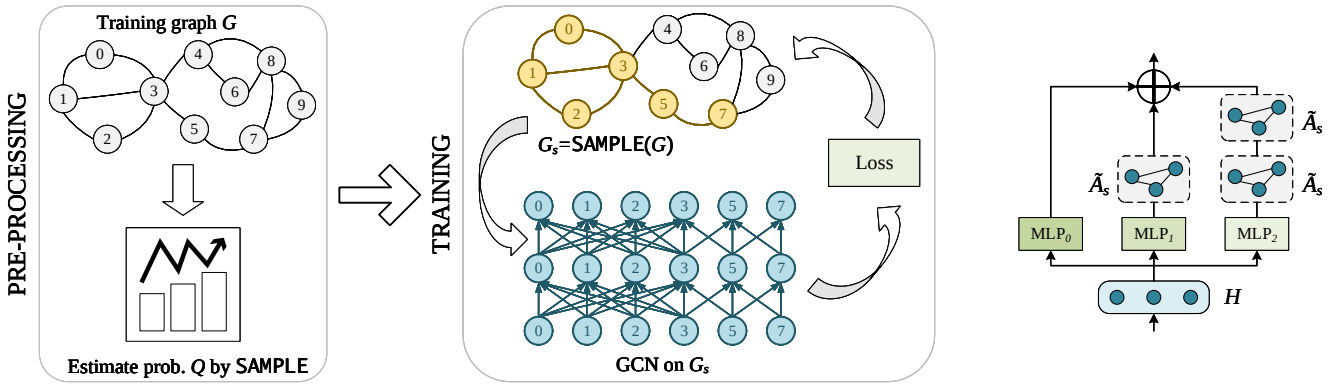
3 METHOD: GRAPHSAINT

3.1 Problem Setup

GCN embeds the input graph into low dimensional vector space by learning interactions between graph nodes and their attributes. The input to a GCN is a node attributed graph $\mathcal{G}$, where each node is associated with a feature vector. The GCN output is a matrix, whose row vectors represent the embedding vectors of the graph nodes. Using the output matrix, we can easily perform downstream tasks like node classification and clustering.

A GCN is built by stacking multiple graph convolution layers. One node in a GCN layer corresponds to one node in graph $\mathcal{G}$. Inter-layer connections are added based on the node connections in $\mathcal{G}$. In other words, a node v in layer ℓ may propagate its activation to node u in layer $\ell + 1$, if v and u are connected in $\mathcal{G}$ (directly or indirectly). Each node in layer $\ell + 1$ gathers features (activations) from nodes in layer ℓ along the inter-layer connections, and then applies some aggregation functions on its neighbor features. We show in Section 3.4 details of the feature propagation rule as well as the design of neighbor aggregators.

GCNs can be applied in the inductive as well as transductive settings. While the proposed GraphSAINT is applicable to both, in this paper, our evaluation is focused on inductive learning. It has been shown that inductive learning is especially difficult [13] — during training, information of nodes (including node attributes and connections) in the test set is not present. Thus, a good inductive model has to generalize to completely unseen graphs. We focus on supervised learning of GCNs. Each training graph node is provided with a ground-truth label. Prediction of the labels is made by an MLP whose inputs are the node embeddings generated by the GCN.



(a) GraphSAINT training algorithm

(b) GraphSAINT GCN layer (order-2)

Figure 1: Illustration of GraphSAINT

Below we define the notations used in the paper. Let $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$ be the training graph with attribute $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times f^{(0)}}$. Let $\boldsymbol{A}$ be the adjacency matrix of $\mathcal{G}$, and $\widetilde{\boldsymbol{A}}$ be the normalized adjacency matrix (See [11, 16] for various normalization methods). Let $\boldsymbol{h}_v^{(\ell)} \in \mathbb{R}^{f^{(\ell)}}$ denote the activation of a node v in GCN layer ℓ. Let $\boldsymbol{H}^{(\ell)} \in \mathbb{R}^{|\mathcal{V}| \times f^{(\ell)}}$ be the activation matrix consisting of $\boldsymbol{h}_v^{(\ell)}$. Note that we regard the input layer as layer 0, and thus $\boldsymbol{H}^{(0)} = \boldsymbol{X}$. Further, let $\boldsymbol{y}_v$ be the predicted label of node v, and $\overline{\boldsymbol{y}}_v$ be the ground truth. Let $\boldsymbol{Y}$ and $\overline{\boldsymbol{Y}}$ be the label matrix constructed from the corresponding label vectors. Lastly, let the set of weight matrices associated with layer ℓ be $\left\{ \boldsymbol{W}^{(\ell)} \right\}$. See Section 3.4 for usage of these weights. Unless otherwise specified, symbols with subscript s denote parameters corresponding to a sampled subgraph of $\mathcal{G}$ (e.g., $\mathcal{G}_s$ is a sampled graph from $\mathcal{G}$, and $\widetilde{\boldsymbol{A}}_s$ is the normalized adjacency matrix of $\mathcal{G}_s$). Parameters within parentheses "$(\cdot)$" in superscript denote GCN layer index, and those without parentheses denote power of a matrix.

3.2 Minibatch Construction

The core design philosophy of GraphSAINT is that, the sampling process in minibatch training should take place on the data itself, before the GCN is constructed. We believe that using proper graph sampling algorithms, we can extract small representative subgraphs from the large training graph, such that accurate label prediction can be made by propagating information *within the subgraphs*.

Figure 1a and Algorithm 1 illustrate the overall training algorithm. Before training starts, we perform light weight pre-processing on $\mathcal{G}$ with the given sampler `SAMPLE`. The pre-processing estimates a probability distribution over $\mathcal{V}$ which is used later to normalize the minibatch loss (Section 3.3). Then training proceeds by iterative weight updates via SGD. Each iteration starts with an independently sampled $\mathcal{G}_s$. We then build A *complete* GCN on the small $\mathcal{G}_s$ (see Section 3.4 for GCN model) which makes predictions for every $v \in \mathcal{V}_s$. We calculate loss by regarding every $v \in \mathcal{V}_s$ as the minibatch node, and use back-propagation for parameter update.

This raises the question of how to design the function `SAMPLE`. Intuitively, there are two requirements: (1) Nodes that have high influence on each other should be sampled in the same subgraph. (2) Each node (even isolated nodes without connection) should have non-zero probability to be sampled in the minibatch. For requirement (1), an ideal `SAMPLE` should consider the joint information from node connections as well as attributes. However, the resulting algorithm may have a high complexity as it would need to infer the relationships between features. For simplicity, we define "influence" only from the graph connectivity perspective and propose traversal based samplers. Requirement (2) leads to better generalization since the learning algorithm has chance to look at any $v \in \mathcal{V}$, and thus non-zero probability of exploring the whole feature space and label space. Note that `SAMPLE` satisfying requirement (1) almost inevitably introduces bias towards "high centrality" nodes. Such bias can be addressed by properly normalizing the minibatch loss, as shown in Section 3.3.

Algorithm 2 presents four simple graph sampling algorithms based on the above two requirements. Each sampler comes with a budget. For NodeSampler, FrontierSampler and RandomWalkSampler, we specify the node budget $|\mathcal{V}_s| \leq N$, while for EdgeSampler, we specify the edge budget $|\mathcal{E}_s| \leq M$. We define the function $\text{neigh}(v, k)$ to return the number of k-hop neighbors of node v. So $\text{neigh}(v, 0) = 1$, and $\text{neigh}(v, 1)$ equals to the degree of v. For NodeSampler, we perform random node sampling based on the distribution defined by the number of k-hop neighbors. Typical, we pick $k \in \{0, 1, 2\}$. For EdgeSampler, we perform random edge sampling from $\mathcal{E}$. We add edges (v, v) to form self-loops, so that even isolated nodes have chance to be sampled in the subgraph. For FrontierSampler, the sampler maintains a fixed size (n) set of frontier nodes. The sampler randomly pops out a frontier node and replaces it with one of its neighbors. Our algorithm design is mostly similar to [17]. The only difference is that when popping out frontier nodes, we follow a distribution defined by number of k-hop neighbors. For RandomWalkSampler, we randomly pick R nodes as the roots, and perform length-D random walks from the roots. For all samplers, we obtain the induced subgraph from the nodes encountered in the sampling process.

We can easily see that complexity of computing one minibatch ($\mathcal{G}_s$) increases linearly with the depth of GCN. This is a direct conclusion from the fact that: (1) Each GCN layer performs identical operations to propagate node features, and (2) The number of nodes sampled in each layer is exactly $|\mathcal{V}_s|$. Note that it has been argued

Algorithm 1 GraphSAINT training algorithm

Input: Training graph $\mathcal{G}(\mathcal{V},\mathcal{E},\boldsymbol{X})$; Labels $\overline{Y}$; Sampler SAMPLE;
Output: GCN model with trained weights

1: Pre-processing ▷ Section 3.3
2: **while** not terminate **do**
3: $\mathcal{G}_s \leftarrow$ SAMPLE $(\mathcal{G})$ ▷ Section 3.2
4: GCN construction on $\mathcal{G}_s$ ▷ Section 3.4
5: $\{\ \boldsymbol{y}_v \mid v \in \mathcal{V}_s\ \} \leftarrow$ Forward propagation of $\{\ \boldsymbol{x}_v \mid v \in \mathcal{V}_s\ \}$
6: Loss calculation by $\boldsymbol{y}_v$ and $\overline{\boldsymbol{y}}_v$, $v \in \mathcal{V}_s$ ▷ Section 3.3
7: Weight update by backward propagation
8: **end while**

Algorithm 2 Graph samplers of GraphSAINT

Input: Training graph $\mathcal{G}(\mathcal{V},\mathcal{E})$; Sampling parameters
Output: Sampled graph $\mathcal{G}_s\ (\mathcal{V}_s,\mathcal{E}_s)$

1: **function** NodeSampler($\mathcal{G}$,N,k)
2: $P(v) := \text{neigh}(v,k)\,/\sum_{v'\in\mathcal{V}} \text{neigh}(v',k)$
3: $\mathcal{V}_s \leftarrow N$ nodes randomly sampled from $\mathcal{V}$ according to P
4: $\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
5: **end function**
6: **function** EdgeSampler($\mathcal{G}$,M)
7: $\mathcal{E}_s \leftarrow M$ edges sampled uniformly at random from $\mathcal{E}$
8: $\mathcal{V}_s \leftarrow$ Set of nodes that are end-points of edges in $\mathcal{E}_s$
9: $\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
10: **end function**
11: **function** FrontierSampler($\mathcal{G}$,N,n,k)
12: $\mathcal{V}_{\text{FS}} \leftarrow n$ nodes sampled uniformly at random from $\mathcal{V}$
13: $\mathcal{V}_s \leftarrow \mathcal{V}_{\text{FS}}$
14: **for** $i = n$ to N **do**
15: $P(v) := \text{neigh}(v,k)\,/\sum_{v'\in\mathcal{V}_{\text{FS}}} \text{neigh}(v',k)$
16: $u \leftarrow$ Node sampled from $\mathcal{V}_{\text{FS}}$ according to P
17: $u' \leftarrow$ Node randomly sampled from u's neighbor
18: $\mathcal{V}_{\text{FS}} \leftarrow (\mathcal{V}_{\text{FS}} \setminus \{\ u\ \}) \cup \{\ u'\ \}$
19: $\mathcal{V}_s \leftarrow \mathcal{V}_s \cup \{\ u\ \}$
20: **end for**
21: $\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
22: **end function**
23: **function** RandomWalkSampler($\mathcal{G}$,R,D)
24: $\mathcal{V}_{\text{root}} \leftarrow R$ nodes sampled uniformly at random from $\mathcal{V}$
25: $\mathcal{V}_s \leftarrow \mathcal{V}_{\text{root}}$
26: **for** $v \in \mathcal{V}_{\text{root}}$ **do**
27: $u \leftarrow v$
28: **for** $d = 0$ to D **do**
29: $u \leftarrow$ Node randomly sampled from u's neighbor
30: $\mathcal{V}_s \leftarrow \mathcal{V}_s \cup \{\ u\ \}$
31: **end for**
32: **end for**
33: $\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
34: **end function**

in [10] that training complexity can potentially grow exponentially with GCN depth for layer sampling based methods. Thus, the minibatch algorithm in GraphSAINT leads to more efficient training.

3.3 Loss Function

Empirical risk for full-batched GCN is given by an estimation of $\mathbb{E}_{v\sim P}\left[L\left(\boldsymbol{y}_v,\overline{\boldsymbol{y}}_v\right)\right]$ as $R_{\text{GCN}} = \sum_{v\in\mathcal{V}} L\left(\boldsymbol{y}_v,\overline{\boldsymbol{y}}_v\right)\cdot P(v)$, where $\boldsymbol{y}_v$ is the label of v evaluated in the full-batched manner (i.e., a node aggregates features from *all* neighbors), $\overline{\boldsymbol{y}}_v$ is the ground-truth label, and L is the loss function. Typically, $P(v) = \frac{1}{|\mathcal{V}|}, \forall v$.

Let probability of a node v occurring in a sampled graph $\mathcal{G}_s\ (\mathcal{V}_s,\mathcal{E}_s)$ be estimated by $Q(v)$ as the fraction of sampled subgraphs that contain v. Assuming that the function SAMPLE ensures a non-zero probability of selecting any given node, $Q(v) > 0$. Further, assume $L_{\mathcal{G}_s}\left(\boldsymbol{y}'_v,\overline{\boldsymbol{y}}_v\right)$ represents the loss calculated on the subgraph $\mathcal{G}_s$ ($\boldsymbol{y}'_v$ is the label predicted for v in the subgraph). We assume that $\mathcal{G}_s$ maintains the essential information from the original graph necessary for training, and so $\left|L_{\mathcal{G}_s}(\boldsymbol{y}'_v,\overline{\boldsymbol{y}}_v) - L(\boldsymbol{y}_v,\overline{\boldsymbol{y}}_v)\right| \le \epsilon$, for some small $\epsilon \ge 0$. Now, we define our minibatch loss as:

$$L^{\text{mini}}_{\mathcal{G}_s} = \sum_{v\in\mathcal{G}_s} L_{\mathcal{G}_s}(\boldsymbol{y}'_v,\overline{\boldsymbol{y}}_v)\cdot\frac{P(v)}{Q(v)}.$$

Finally, using large number of sampled graphs, expected minibatch loss can be calculated as:

$$\mathbb{E}_{\mathcal{G}_s\sim\text{SAMPLE}}\left(L^{\text{mini}}_{\mathcal{G}_s}\right) = \frac{1}{|\mathbb{G}|}\sum_{\mathcal{G}_s\in\mathbb{G}} L^{\text{mini}}_{\mathcal{G}_s}\left(\boldsymbol{y}'_v,\overline{\boldsymbol{y}}_v\right),$$

where $\mathbb{G}$ is the set of all sampled graphs.

Proposition 3.1. *Under the assumption of ideal sampler, i.e.,* $\left|L_{\mathcal{G}_s}\left(\boldsymbol{y}'_v,\overline{\boldsymbol{y}}_v\right) - L\left(\boldsymbol{y}_v,\overline{\boldsymbol{y}}_v\right)\right| \le \epsilon$ *, the expected minibatch loss of GraphSAINT approaches the empirical risk in the full batch GCN,*

$$\left|\mathbb{E}\left(L^{mini}_{\mathcal{G}_s}\right) - R_{GCN}\right| \le \epsilon$$

Proof. By definition,

$$\begin{aligned}
\mathbb{E}\left(L^{\text{mini}}_{\mathcal{G}_s}\right) &= \frac{1}{|\mathbb{G}|}\sum_{\mathcal{G}_s\in\mathbb{G}}\sum_{v\in\mathcal{V}_s} L_{\mathcal{G}_s}(\boldsymbol{y}'_v,\overline{\boldsymbol{y}}_v)\cdot\frac{P(v)}{Q(v)} \\
&= \frac{1}{|\mathbb{G}|}\sum_{v\in\mathcal{V}}\sum_{\mathcal{G}_s\in\mathbb{G},v\in\mathcal{V}_s} L_{\mathcal{G}_s}(\boldsymbol{y}'_v,\overline{\boldsymbol{y}}_v)\cdot\frac{P(v)}{Q(v)} \\
&\le \frac{1}{|\mathbb{G}|}\sum_{v\in\mathcal{V}}\sum_{\mathcal{G}_s\in\mathbb{G},v\in\mathcal{G}_s}\left(L(\boldsymbol{y}_v,\overline{\boldsymbol{y}}_v)+\epsilon\right)\cdot\frac{P(v)}{Q(v)} \\
&= \sum_{v\in\mathcal{V}}\left(L(\boldsymbol{y}_v,\overline{\boldsymbol{y}}_v)+\epsilon\right)\cdot\frac{P(v)}{Q(v)}\frac{\sum_{\mathcal{G}_s}\mathbb{I}_{\mathcal{G}_s}(v)}{|\mathbb{G}|} \\
&= \sum_{v\in\mathcal{V}}\left(L(\boldsymbol{y}_v,\overline{\boldsymbol{y}}_v)+\epsilon\right)\cdot\frac{P(v)}{Q(v)}\cdot Q(v) \\
&= \sum_{v\in\mathcal{V}}\left(L(\boldsymbol{y}_v,\overline{\boldsymbol{y}}_v)+\epsilon\right)\cdot P(v) = R_{\text{GCN}}+\epsilon. \qquad (1)
\end{aligned}$$

Here, $\mathbb{I}$ is the indicator function. Similarly, we can show $\mathbb{E}\left(L^{\text{mini}}_{\mathcal{G}_s}\right) \ge R_{\text{GCN}} - \epsilon$. □

Our minibatch loss calculation requires an estimate of the probability of a node v being sampled by SAMPLE (defined by Q). This is done by repeatedly run SAMPLE many times, and count the number of occurrence of each $v \in \mathcal{V}$. This pre-processing step for Q estimation incurs near-zero overhead, since the subgraphs returned by SAMPLE are reused as minibatches in training. Algorithm 3 shows the pre-processing (corresponding to Algorithm 1, line 1).

Algorithm 3 Pre-processing to estimate Q

Input: Training graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; Graph sampler SAMPLE;
Output: Estimated Q; Set of sampled subgraphs $\mathbb{G}$

$\boldsymbol{c} \leftarrow$ Counter vector of size $|\mathcal{V}|$, initialized to all 0
$\mathbb{G} \leftarrow \emptyset$
for $i = 1$ to N **do** ▷ N is some given constant
  $\mathcal{G}_{s,i}\left(\mathcal{V}_{s,i}, \mathcal{E}_{s,i}\right) \leftarrow \texttt{SAMPLE}(\mathcal{G})$
  $\mathbb{G} \leftarrow \mathbb{G} \cup \left\{ \mathcal{G}_{s,i} \right\}$
  for v in $\mathcal{V}_{s,i}$ **do**
    $\boldsymbol{c}_v \leftarrow \boldsymbol{c}_v + 1$
  end for
end for
$Q(v) := \boldsymbol{c}_v / N$

3.4 GCN Model

In this section, we first generalize the existing GCN layer designs [9–13, 16, 18] using two fundamental propagation rules. Then we propose a deep GCN model whose efficient training is enabled by the minibatch construction of GraphSAINT. Various GCN models proposed in the literature share common propagation characteristics. In the most general case, each GCN layer performs two aggregation steps to propagate the previous layer activation to the next layer. Equation 2 specifies the basic rules:

$$\begin{aligned} \boldsymbol{M}_k^{(\ell)} &= \widetilde{\boldsymbol{A}}^k \bullet \phi_k\left(\boldsymbol{H}^{(\ell)}\right) \\ \boldsymbol{H}^{(\ell+1)} &= \sigma\left(\bigoplus_{k=0}^{K} \psi_k\left(\boldsymbol{M}_k^{(\ell)}\right)\right) \end{aligned} \tag{2}$$

where $\boldsymbol{M}_k^{(\ell)}$ stores the intermediate results; ϕ and ψ are some non-linear mapping on each row of $\boldsymbol{H}^{(\ell)}$ and $\boldsymbol{M}_k^{(\ell)}$; "$\bullet$" specifies some reduction operation along columns of $\phi_k(\cdot)$, weighted by rows of $\widetilde{\boldsymbol{A}}^k$ (e.g., matrix dot product); "$\bigoplus$" performs another reduction operation on the output matrices of ψ_k; σ is the layer activation function (e.g., ReLU). The design of the second propagation step is inspired by the approximation of spectral graph convolution using Chebyshev polynomials, as detailed in [11, 16]. We refer to K as the "*order*" of a GCN layer. Table 1 summarizes the choices of the propagation parameters by state-of-the-art methods.

Table 1: Propagation rules for various GCN models.

Model	Training	K	ϕ	$\bullet$	ψ	$\bigoplus$
[11, 16]	F.B.	≥ 1	Lin.	Sum	Id.	Sum
[9, 10]	L.S.	1	Lin.	Sum	Id.	Sum
[18]	L.S.	1	MLP	Sum	MLP	Concat.
[13]	L.S.	1	MLP	Pool	Id.	Concat.
[13]	L.S.	1	Id.	LSTM	Id.	Concat.
[12]	L.S.	1	Id.	Pool	CNN	Concat.
Ours	G.S.	≥ 0	MLP	Sum	Id.	Sum/Concat.

F.B.: **F**ull **B**atch training, L.S.: training by **L**ayer **S**ampling, G.S.: training by **G**raph **S**ampling; Id.: identity mapping ($\phi(\boldsymbol{H}) = \boldsymbol{H}$), Lin.: linear mapping ($\phi(\boldsymbol{H}) = \boldsymbol{H} \cdot \boldsymbol{W}$)

Note that a straight-forward way to realize an order-K GCN layer is by stacking K number of order-1 layer together, as below. For instance, for the GCN model of [9, 10] with order 1, Equation 2 simplifies to: $\boldsymbol{H}^{(\ell+1)} = \sigma\left(\sum_{k=0}^{1} \widetilde{\boldsymbol{A}}^k \cdot \boldsymbol{H}^{(\ell)} \cdot \boldsymbol{W}_k^{(\ell)}\right)$. Now stacking two such layers together (without the non-linear activation σ in between) gives: $\boldsymbol{H}^{(\ell+2)} = \sigma\left(\sum_{k=0}^{2} \widetilde{\boldsymbol{A}}^k \cdot \boldsymbol{H}^{(\ell)} \cdot \boldsymbol{W}_k'\right)$, where $\boldsymbol{W}_0' = \boldsymbol{W}_0^{(\ell)} \cdot \boldsymbol{W}_0^{(\ell+1)}$, $\boldsymbol{W}_1' = \boldsymbol{W}_1^{(\ell)} \cdot \boldsymbol{W}_0^{(\ell+1)} + \boldsymbol{W}_0^{(\ell)} \cdot \boldsymbol{W}_1^{(\ell+1)}$ and $\boldsymbol{W}_2' = \boldsymbol{W}_1^{(\ell)} \cdot \boldsymbol{W}_1^{(\ell+1)}$. Therefore, an order-$K$ layer can be realized by K number of order 1 layers. On the positive side, this implementation enables the existing layer sampling techniques to be applied directly to order-K layers. On the other hand, however, such implementation results in increased model size and depth — a single order-K layer is of depth 1 and contains $K+1$ weight matrices, while K number of order-1 layers stacked together have depth K and contain $2K$ weight matrices. Expanding model size and depth may cause difficulties in learning (e.g., gradient vanishing).

Owing to this observation and to allow flexibility in modeling, we design a GCN layer of GraphSAINT, specified by the last row of Table 1. A single GraphSAINT layer supports any order $K \geq 0$, where $K = 0$ corresponds to a normal MLP layer without any information exchange along graph edges. We use MLP to perform mapping ϕ. We observe that there is not much gain in including an additional mapping ψ after ϕ — an order-K GraphSAINT layer with ψ as MLP is almost the same as an order-K layer followed by an order-0 layer with ψ as identity mapping.

Figure 1b shows an order-2 GraphSAINT layer, where $\widetilde{\boldsymbol{A}}_s$ represents the (normalized) adjacency matrix of subgraph $\mathcal{G}_s$. Note:

(1) GraphSAINT easily supports minibatch training of order-K GCNs — we only need to replace $\widetilde{\boldsymbol{A}}$ with $\widetilde{\boldsymbol{A}}_s$ in Equation 2.
(2) Each order-K layer has K parallel branches. This design in principle is similar to the "skip-connection" design popular in deep CNNs [14]. If a node k hops away does not propagate useful information, the training algorithm can easily learn weights of MLP_k as 0, thus skipping the whole branch k.
(3) We use k stages of $\widetilde{\boldsymbol{A}}_s$ rather than a single stage of $\widetilde{\boldsymbol{A}}_s^k$ to exchange information among the MLP_k outputs. This is a simple trick that saves a lot of computation in training as $\widetilde{\boldsymbol{A}}_s^k$ can be dense. Notice that $\left(\widetilde{\boldsymbol{A}}_s^2\right) \cdot \phi_k\left(\boldsymbol{H}^{(\ell)}\right) = \widetilde{\boldsymbol{A}}_s \cdot \left(\widetilde{\boldsymbol{A}}_s \cdot \phi_k\left(\boldsymbol{H}^{(\ell)}\right)\right)$.

4 EXPERIMENTS

We conduct extensive experiments to evaluate GraphSAINT in terms of subgraph sampling based minibatch construction and GCN model design. We also demonstrate the advantage of GraphSAINT in terms of both training accuracy and time through comparison against state-of-the-art methods on four large graph datasets.

4.1 Experimental Setup

As specified in Section 3.1, we focus on inductive, supervised learning. We use four datasets for evaluation as summarized in Table 2. (1) *Protein-Protein Interaction (PPI) graph* [3]: This graph specifies interactions between proteins of human tissues. The features correspond to positional gene sets, motif gene sets and immunological

signatures [20]. The labels are gene ontology sets. (2) *Flickr graph* [1]: This is an image-image graph. A node represents an image on Flickr, and an edge connects two nodes if they share the same location, gallery, etc. The node features contain the bag of words representation of the image description. The labels categorizes the type of the image, like portrait or nature scene. (3) *Reddit graph* [3]: This is a post-post graph. A node represents a post on Reddit, and an edge between two nodes is formed if they shared a common user in their comments. The node features are the concatenation of 300-dimensional word vectors from their title and comments.[2] The labels are the community (on Reddit) of the posts. (4) *Yelp graph* [6]: This is a user-user graph. A node represents a user of Yelp. An edge represents that the linked users are friends of each other. The node features are word vectors from the reviews posted by the users. The labels indicate the type of business that the users often review. All four datasets follow "fixed-partition" splits. More details of the datasets can be found at Appendix A.2.

We implement graph samplers in Python and the training algorithm with Tensorflow [7] and Adam [15] optimizer (source code released anonymously at [1]). We run all the experiments on a NVIDIA Tesla P100 GPU (see Appendix A.1 for hardware specification).

The metrics of interest are the F1-micro score and the total training time. When calculating training time (for GraphSAINT as well as baselines), we accumulate the time to perform forward and backward propagation on minibatches, and exclude the time to generate predictions on the validation / test set. The reason is that different methods use different algorithms to generate validation predictions (for example, [10] uses a fast algorithm to estimate the predictions on the validation set and a slower yet more accurate algorithm to calculate the exact predictions on the test set). To make fair timing comparison, we exclude the pre-processing time of the baseline [10] (to aggregate node attributes for all $v \in \mathcal{V}$ and setup the historical value table) as well as GraphSAINT (to estimate distribution Q, Algorithm 3, which is relatively negligible). Before training, we repeatedly perform independent graph sampling. This step is trivially parallelized by launching one CPU core for one independent sampler. See Appendix B for pre-processing time analysis.

Table 2: Dataset Statistics

Dataset	Nodes	Edges	Feat.	Classes	Train/Val/Test
PPI	14,755	225,270	50	m-121	0.66/0.12/0.22
Flickr	89,250	899,756	500	s-7	0.50/0.25/0.25
Reddit	232,965	11,606,919	602	s-41	0.66/0.10/0.24
Yelp	716,847	6,977,410	300	m-100	0.75/0.15/0.10

"m" stands for **m**ulti-class classification, and "s" stands for **s**ingle-class.

One "epoch" in training is conventionally defined as the full traversal of nodes in the output layer. In layer sampling based methods [10, 13, 16], if the minibatch size is B and we sample the output layer uniformly without replacement, all the $|\mathcal{V}|$ output nodes can be covered by $|\mathcal{V}|/B$ number of minibatches. We abuse the notion of "epoch" in the case of GraphSAINT by defining one epoch as $|\mathcal{V}|/|\mathcal{V}_s|$ number of minibatches. It should be noted, however, that there is no guarantee all output nodes are covered exactly once over these minibatches (subgraphs).

[1] https://github.com/GraphSAINT/GraphSAINT

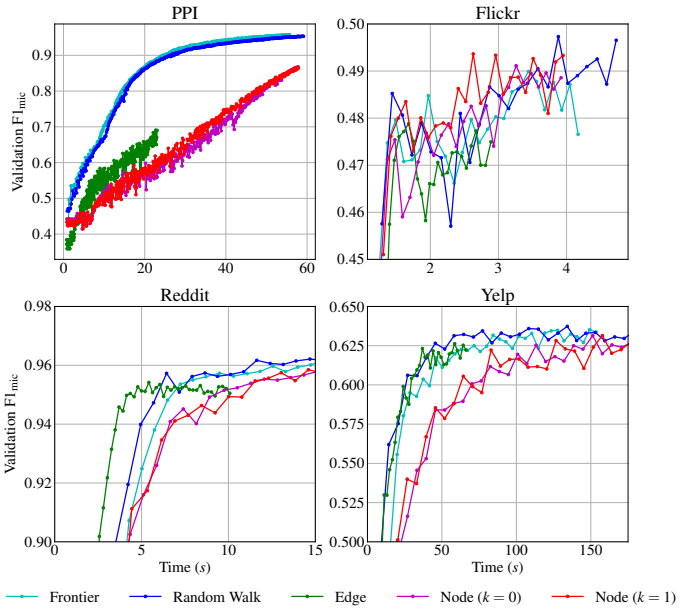


Figure 2: Performance comparison of various samplers

4.2 Evaluation on GraphSAINT: Minibatch Construction

We first evaluate how the choice of graph sampler influences the quality of learning. We test the four samplers listed in Algorithm 2. Figure 2 shows the convergence plots for various graph samplers. For this set of experiments, we use a 2-layer model. Given the dataset, we train using each graph sampler for the same number of epochs, and we keep the subgraph size $|\mathcal{V}_s|$ the same across samplers ($|\mathcal{V}_s| = 8000$). Other parameters are as follows:

- EdgeSampler: Edge budget $M = 10,000$, terminate if number of nodes sampled is approximately 8000.
- NodeSampler: Number of hops $k \in \{0, 1\}$.
- FrontierSampler: Number of hops $k = 1$; frontier $m = 1000$.
- RandomWalkSampler: Number of roots $R = 3000$; walk length $D = 3$.

Overall, graph traversal based samplers (RandomWalkSampler, FrontierSampler) achieve higher F1_{mic}. Edge sampler tends to return sparser subgraphs, and thus training with edge sampler converges in the least time. However, due to the loss of neighbor information in such sparse subgraphs, the final accuracy is low. Node sampling in many cases leads to poor accuracy as well. Note that for PPI, the convergence under NodeSampler ($k = 0$ and $k = 1$) is significantly slower. This is due to the fact that the batch size of $|\mathcal{V}_s| = 8000$ is too large for small graphs such as PPI. Also, we note that all samplers seem to converge to approximately same F1_{mic}, however, at different rates. Given a sampler, the choice of its own parameters also affects training quality. By additional experiments, we find out that for RandomWalkSampler, keeping the walk length small ($3 \leq$

$D \leq 5$) helps with accuracy and convergence. For FrontierSampler, 1000 is a good empirical value for frontier size m. All the above findings are consistent with the intuition developed in Section 3.2. A good sampler should discover connectivity patterns in the original training graph $\mathcal{G}$, and also be able to explore all nodes in $\mathcal{G}$ without being trapped in a small, highly connected region.

4.3 Evaluation on GraphSAINT: GCN Model

Here we evaluate the GCN model presented in Section 3.4. Although most state-of-the-art methods focus on networks with only two graph convolution layers, we explore the effect of further increasing the GCN depth. We also design experiments to reveal the potential of using high order layers to improve training quality.

We design GCN models of various depth and orders for the four datasets. Table 3 summarizes the training statistics. We use "$o_1 - o_2 - \ldots - o_n$" to denote a GCN with n graph convolution layers, where o_i defines the order of layer i. The "Dim." column denotes the dimension (length) of each hidden layer activation. We notice that each dataset in this benchmark set has its own unique characteristics, and altogether they lead to the following findings:

Depth and accuracy: Increasing the number of graph convolution layers may help with accuracy improvement. For Flickr, simply stacking order 1 layers consistently improves accuracy. As expected, the number of epochs required to converge also increases. Therefore, the trade-off needs to be considered in increasing the number of layers. For instance, for PPI, accuracy is slightly improved by adding one more layer to a 2-layer model. However, the accuracy achieved by the 2-layer GCN is already high, and since PPI is a relatively small graph, sampling turns out to be less helpful in decreasing execution time.

Order and accuracy: Keeping GCN depth the same, accuracy may also be improved by increasing the order of the layer(s). As shown by the rows of Yelp, for the three 3-layer designs, increasing the order of last layer from 1 to 2 improves $F1_{mic}$ from 0.639 to 0.644, and increasing order of all layers further improves $F1_{mic}$ to 0.647. It is interesting to see that the $1-1-2$ architecture leads to better accuracy as well as faster convergence than $1-1-1$.

Depth and order: As analyzed in Section 3.4, the order of a layer and depth of a model are closely related. For Reddit, model with a single order-2 layer achieves almost identical $F1_{mic}$ as model with two order-1 layers. Comparing the "2" architecture with "$1-1$", the former is shallower and smaller (less weight parameters). Thus, the single layer, order-2 design may converge faster. We also show the convergence curves for the two designs in Figure 3. Together with the observation on Yelp, we conclude that a deep model consisting of a mixture of order-1 and high order layers may lead to the best performance in terms of accuracy and convergence.

Depth and minibatch time: Figure 4 provides a comprehensive evaluation on the training time complexity. We show the relation between per-minibatch GPU computation time and model depth. We increase the number of layers while fixing the hidden layer dimension (256) and sampler configuration (FrontierSampler, $|\mathcal{V}_s| = 6000$).

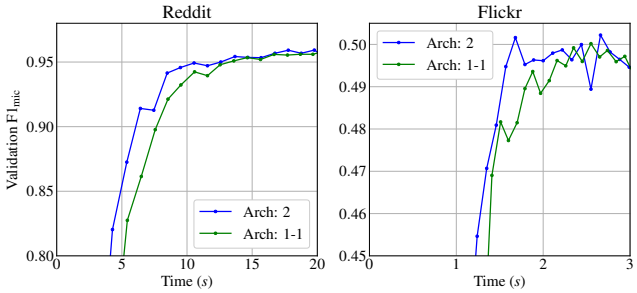


Figure 3: Convergence under two architectures

Table 3: Exploration on GCN models (depth and order)

Dataset	Dim.	Sampler	Arch.	$F1_{mic}$	Epoch	Time (s)
PPI	512	Frontier	$1-1$	0.976	942	141
			$1-1-1$	0.978	890	262
Flickr	512	Frontier	1	0.490	12	2
			$1-1$	0.498	13	5
			$1-1-1$	0.506	32	14
			$1-1-1-1$	0.513	28	23
Reddit	256	Frontier	2	0.962	26	29
			$1-1$	0.963	39	38
Yelp	256	Frontier	$1-1-1$	0.639	32	158
			$1-1-2$	0.644	25	142
			$2-2-2$	0.647	24	206

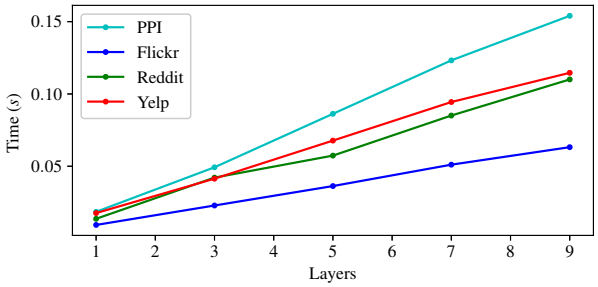


Figure 4: Per-minibatch GPU time *vs.* GCN depth

It is clear that the per-minibatch training time increases approximately linearly with model depth. Section 4.4 shows more details of training time comparison with state-of-the-art methods.

The above model / architecture exploration indicates that our graph sampling based training enables fast exploration of deep GCN models on large scale graphs. One one hand, with the continuous efforts invested in model development, we may likely end up with very deep GCN models to achieve high accuracy. This makes it a critical problem to resolve "neighbor explosion" in training. On the other hand, with the introduction of "order" as an additional design parameter, the design space for GCN models becomes even larger. This means fast training algorithm on large graphs are of utmost importance.

Table 4: Comparison of training time and accuracy with state-of-the-art methods

Layer	Method	PPI			Flickr			Reddit			Yelp		
		$F1_{mic}$ (test)	Epoch	Time (s)	$F1_{mic}$ (test)	Epoch	Time (s)	$F1_{mic}$ (test)	Epoch	Time (s)	$F1_{mic}$ (test)	Epoch	Time (s)
2	**SpGCN** [16]	.515 ± .006	45 ± 3	**13** ± 1	.496 ± .001	7 ± 3	13 ± 4	.933 ± .000	6 ± 1	36 ± 3	.378 ± .001	9 ± 0	170 ± 2
	GraphSAGE [13]	.617 ± .006	48 ± 1	24 ± 3	**.503** ± .000	3 ± 1	6 ± 2	.953 ± .001	4 ± 1	**26** ± 3	.634 ± .006	9 ± 1	161 ± 10
	S-GCN [10]	.954 ± .002	157 ± 7	17 ± 1	.461 ± .003	12 ± 0	7 ± 0	.964 ± .001	32 ± 0	80 ± 9	.638 ± .002	14 ± 1	**122** ± 5
	GraphSAINT	**.959** ± .002	346 ± 3	53 ± 8	.501 ± .000	16 ± 5	**2** ± 1	**.964** ± .001	36 ± 3	28 ± 2	**.639** ± .003	30 ± 1	127 ± 5
3	**GraphSAGE** [13]	.620 ± .006	48 ± 1	56 ± 2	.500 ± .001	3 ± 1	48 ± 13	.956 ± .001	6 ± 2	280 ± 61	.637 ± .005	7 ± 1	808 ± 127
	S-GCN [10]	.921 ± .010	210 ± 15	**46** ± 4	.487 ± .005	20 ± 1	27 ± 1	.965 ± .001	61 ± 5	490 ± 43	.643 ± .001	19 ± 0	279 ± 7
	GraphSAINT	**.959** ± .001	344 ± 5	75 ± 2	**.506** ± .002	32 ± 8	**14** ± 3	**.965** ± .001	42 ± 2	**44** ± 2	**.644** ± .001	25 ± 1	**142** ± 5

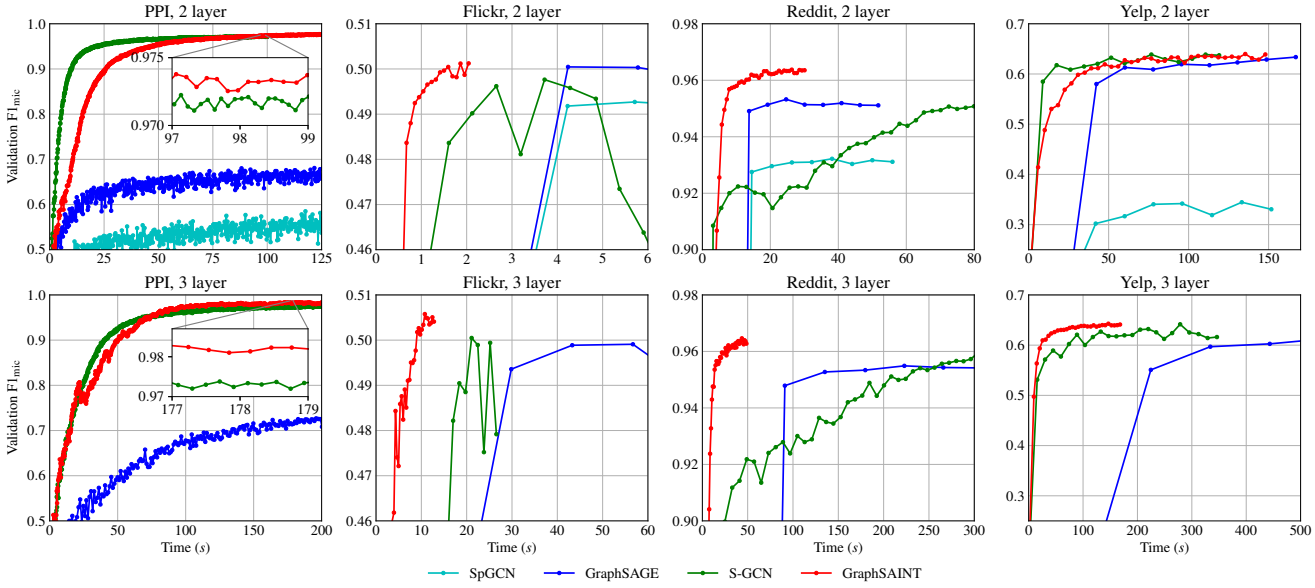


Figure 5: Convergence plots for validation $F1_{mic}$

4.4 Comparison with State-of-the-Art

We compare performance (F1-micro score, convergence time) with strong baselines: Spectral GCN (SpGCN) [16], GraphSAGE [13] and Stochastic training GCN (S-GCN) [10]. All baselines are run with the public source code: [4] for [16] and [13]; [5] for [10]. We run the code 3 times for each configuration of the baselines and GraphSAINT. We report mean and standard deviation of the metrics. Appendix D shows additional accuracy comparison with [9, 12].

For baselines, we follow the termination criteria as specified in the original respective papers / codes. S-GCN uses early stopping termination criteria [10]. Since S-GCN relies on the historical activations in the hidden layers to estimate the last layer output, it does not calculate the exact validation set predictions during training. Thus, its early stopping is judged by the convergence of minibatch loss. Based on the public implementation [4], GraphSAGE and SpGCN terminate after finishing a fixed number of epochs. For GraphSAINT, we also run it for a fixed number of epochs. The total number of epochs are set separately for each dataset based on convergence. Regarding the GCN model, we test GraphSAGE with mean aggregator, and S-GCN with its best performing CVD (Control-Variance with Dropout) technique. GraphSAINT uses the layer design as specified in Section 3.4. For the baselines (SpGCN, GraphSAGE and S-GCN) running PPI and Reddit with 2-layer GCN, we keep all configuration identical to that stated in the original paper / code. For all other experiments, we perform thorough hyper-parameter and model tuning for each method, and report the results corresponding to the configuration with highest $F1_{mic}$. The major parameters that affect $F1_{mic}$ are hidden layer activation length, dropout rate (for S-GCN) and sampler configuration (for GraphSAINT). Appendix C shows details of the configuration (overall GCN model, sampling parameters, etc.) for each dataset.

Table 4 summarizes the performance comparison of the various methods for 2-layer and 3-layer GCNs ($F1_{mic}$ is reported on the test set). Figure 5 shows the convergence plot. Overall, GraphSAINT has the highest training quality. Without loss of accuracy, the training time speedup of GraphSAINT is more significant for deeper GCNs. Note that for S-GCN running PPI, the results in Table 4 correspond to the early stopping criteria specified in the source code [5]. Judging from the trend in validation $F1_{mic}$, we find that

the actual convergence happens at around epoch 1000 (Figure 5). In either case, GraphSAINT is able to achieve higher accuracy than S-GCN, as shown by the convergence curve.

$F1_{mic}$ score: GraphSAINT and S-GCN achieve higher accuracy than GraphSAGE and GCN for PPI, Reddit and Yelp. GraphSAINT and GraphSAGE achieve higher accuracy than S-GCN and GCN for Flickr. Thus, we conclude that due to our choices of samplers (Section 3.2) and normalized loss function design (Section 3.3), the graph sampling process in fact incurs less information loss than the layer sampling process in minibatch training. It should be noted that for PPI, S-GCN has a lower training time compared to GraphSAINT due to early termination. The accuracy of S-GCN, in fact, further increases and becomes almost equal to GraphSAINT when they are both given enough time to execute (Figure 5).

Training time: For larger scale graphs (Flickr, Reddit and Yelp), GraphSAINT uses the least time to converge. The biggest performance gain comes from the fact that the graph sampling method completely resolves the "neighbor explosion" problem in layer sampling based approaches. High speedup is achieved on Reddit. GraphSAINT uses 2.9× less training time than S-GCN to reach the same $F1_{mic}$ score for GCNs with 2 layers. Notice that this speedup is purely due to the difference in minibatch training algorithm, since all GCN layers of GraphSAINT and S-GCN have the same order ($K = 1$) and hidden activation length (128). The speedup of GraphSAINT over S-GCN further increases to 11.1× when increasing the depth to 3. It is also worth noticing that although GraphSAGE converges faster than GraphSAINT on Reddit, it reaches lower accuracy. As shown in Figure 5, GraphSAINT is still significantly faster than GraphSAGE in reaching the same $F1_{mic}$ score for both 2-layer and 3-layer designs. For Yelp, although GraphSAINT is slightly slower than S-GCN (5s out of 127s) on 2-layer models, we achieve 2.0× speedup on 3-layer models compared with S-GCN. For PPI, our performance (convergence curve) is almost identical to S-GCN for 3-layer models. This indicates that small graphs may not gain much benefit from graph sampling based training. This is simply because that the number of support nodes (in layer sampling based methods) is ultimately bound by the training graph size.

Convergence: GraphSAINT usually requires more epochs than the other baselines. The main reason is that the minibatch size of GraphSAINT is larger than that of other layer sampling based methods. SpGCN, GraphSAGE and S-GCN have 512 minibatch nodes (i.e., nodes of the output layer). GraphSAINT has around 2000 to 8000 minibatch nodes (i.e., nodes in the sampled subgraphs). When using traversal based samplers such as FrontierSampler or RandomWalkSampler, we don't expect the minibatch size of GraphSAINT to further increase when using even larger training graphs, since such samplers focus on a small, locally connected region of the original training graph. See Appendix C for details of the minibatch size.

5 CONCLUSION

We have proposed GraphSAINT – a graph sampling based graph convolutional networks (GCNs) for inductive leaning on large graphs. Our technique is a fundamental shift in perspective for training GCNs by performing the sampling on graph and then constructing a GCN on the sampled subgraph, instead of the traditional approach that first constructing the GCN and then sampling on each layer. With proper weighted computation of the loss in each GCN, we have shown that the expected loss approximates the loss of full batch GCN. Enabled by fast training we are able to perform inductive learning with deeper layers and with higher "order" of propagation between layers. We have conducted detailed analysis on several sampling algorithms and shown that Random Walk sampling and Frontier sampling produce the best results. We have also shown that deeper layers increase accuracy, and higher order with fewer layers can perform better compared to lower orders of more layers. While we introduce design parameters, the quick training enables fast exploration of the choice of right model. Finally, we have shown that we are able to achieve higher accuracy compared to state-of-the-art methods in less time. Especially for large graphs and deeper layer, the speed up is extremely high – 2× to 11×.

In the future, we will design and evaluate graph sampling algorithms that can better capture attributed graphs. In addition, we will explore training techniques and alternative GCN models that help improve accuracy of deep GCNs.

REFERENCES

[1] 2019. Flickr Datasets. https://snap.stanford.edu/data/web-flickr.html.
[2] 2019. NUS-wide Datasets. http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm.
[3] 2019. PPI/Reddit Datasets. http://snap.stanford.edu/graphsage/#datasets.
[4] 2019. Source code (GraphSAGE). https://github.com/williamleif/GraphSAGE.
[5] 2019. Source code (S-GCN). https://github.com/thu-ml/stochastic_gcn.
[6] 2019. Yelp 2018 Challenge. https://www.yelp.com/dataset.
[7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, et al. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. CoRR abs/1603.04467 (2016). arXiv:1603.04467
[8] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2013. Spectral Networks and Locally Connected Networks on Graphs. CoRR abs/1312.6203 (2013). arXiv:1312.6203 http://arxiv.org/abs/1312.6203
[9] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In International Conference on Learning Representations (ICLR).
[10] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction.. In ICML. 941–949.
[11] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In Advances in Neural Information Processing Systems. 3844–3852.
[12] Hongyang Gao, Zhengyang Wang, and Shuiwang Ji. 2018. Large-Scale Learnable Graph Convolutional Networks. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18). ACM, New York, NY, USA, 1416–1424.
[13] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In Advances in Neural Information Processing Systems 30. 1024–1034.
[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. CoRR abs/1512.03385 (2015). arXiv:1512.03385
[15] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. CoRR abs/1412.6980 (2014). arXiv:1412.6980 http://arxiv.org/abs/1412.6980
[16] Thomas Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. In International Conference on Learning Representations (ICLR).
[17] Bruno Ribeiro and Don Towsley. 2010. Estimating and Sampling Graphs with Multidimensional Random Walks. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10). ACM, New York, NY, USA, 390–403. https://doi.org/10.1145/1879141.1879192
[18] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18). 10.
[19] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2018. Accurate, Efficient and Scalable Graph Embedding. CoRR abs/1810.11899 (2018). arXiv:1810.11899 http://arxiv.org/abs/1810.11899
[20] Marinka Zitnik and Jure Leskovec. 2017. Predicting multicellular function through multi-layer tissue networks. CoRR abs/1707.04638 (2017). arXiv:1707.04638

A DETAILED EXPERIMENTAL SETUP

A.1 Hardware Specification

We ran our experiments on a single machine with Dual Intel Xeon CPUs (E5-2698 v4 @ 2.2Ghz), one NVIDIA Tesla P100 GPU (16GB of HBM2 memory) and 512GB of DDR4 memory. The code is written in Python 3.6.8. We use Tensorflow 1.12.0 on CUDA 9.2 with CUDNN 7.2.1 to train the model on GPU and Cython 0.29.2 to sample the subgraphs on 10 cores of the CPU.

A.2 Dataset Details

Here we present the detailed processing procedures of the Yelp and Flickr datasets prepared by ourselves.

Yelp dataset is provided with raw data of businesses, users and reviews. For nodes and edges, we scanned the friends list of each user in the raw data of users. If two users were friends, we created an edge between the nodes representing the two users. We then scanned the all reviews by each user and separate the reviews into words. Each review word is converted to a 300 dimension vectors using the pre-trained GoogleNews [2] model. The word vectors for each node were added and normalized to serve as the node features. For the labels of each node, we scanned the raw businesses data reviewed by that user. If the number of categories of the businesses that a user reviewed was more than some threshold, we assigned this category as the label to the node of this user, which collectively formed a multi-class label for each node.

Flickr dataset originates from NUS-wide [2]. [1] collected four different Flickr datasets including NUS-wide and generated an undirected graph for these four datasets. One node in the graph represents one image uploaded to Flickr. If two images have some connections (same geographic location, same gallery, same user commented, etc.), there is an edge between the nodes of these two images. We removed the nodes of [1] not in the NUS-wide dataset and their edges. We directly used the 500 dimensional bag of words representations of the images provided in the NUS-wide dataset as the node features. For labels, we scanned over the 81 tags of each image and manually merged them to 7 classes. Each image belongs to one of the 7 classes.

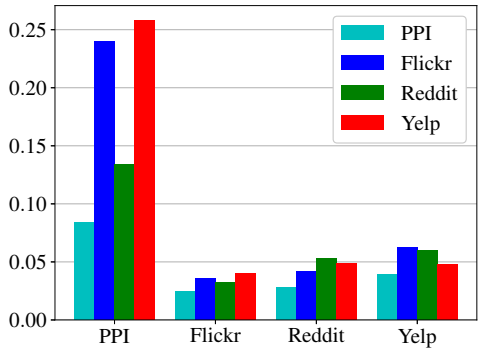


Figure 6: Fraction of total training time spent on sampling

[2] https://code.google.com/archive/p/word2vec/

B PRE-PROCESSING

Before training starts, GraphSAINT performs $N = |\mathcal{S}|$ number of independent graph sampling on the training graph $\mathcal{G}$, in order to estimate $Q(v)$ (Section 3.3). For the four datasets we evaluated, we set N to be $25 \times \frac{|\mathcal{V}|}{|\mathcal{V}_s|} \leq N \leq 50 \times \frac{|Vt|}{|\mathcal{V}_s|}$. In other words, the pre-processing step produces subgraphs that can be later on consumed in $25 \sim 50$ training epochs. The sampler is executed on CPU, and by launching one CPU core to execute one independent sampler, the pre-processing is trivially parallelized. We measured the percentage of total training time spent on executing graph sampler. As shown by Figure 6, graph sampling only accounts for around 5% of the total training time for Flickr, Reddit and Yelp, and less than 25% for the smaller PPI graph.

Table 5: Hyper-parameters for state-of-the-art comparison

Layer	Dataset	Method	Dimension	Dropout
2	PPI	SpGCN [16]	512	0.0
		GraphSAGE [13]	128	0.0
		S-GCN [10]	512	0.2
		GraphSAINT	512	0.0
	Flickr	SpGCN [16]	512	0.0
		GraphSAGE [13]	512	0.0
		S-GCN [10]	256	0.2
		GraphSAINT	256	0.2
	Reddit	SpGCN [16]	128	0.0
		GraphSAGE [13]	128	0.0
		S-GCN [10]	128	0.2
		GraphSAINT	128	0.0
	Yelp	SpGCN [16]	512	0.0
		GraphSAGE [13]	512	0.0
		S-GCN [10]	512	0.0
		GraphSAINT	512	0.0
3	PPI	GraphSAGE [13]	512	0.0
		S-GCN [10]	512	0.2
		GraphSAINT	512	0.0
	Flickr	GraphSAGE [13]	512	0.0
		S-GCN [10]	512	0.2
		GraphSAINT	512	0.2
	Reddit	GraphSAGE [13]	256	0.0
		S-GCN [10]	512	0.2
		GraphSAINT	128	0.0
	Yelp	GraphSAGE [13]	512	0.0
		S-GCN [10]	256	0.0
		GraphSAINT	256	0.0

C HYPER-PARAMETERS AND GCN ARCHITECTURE

Here we show the detailed training parameters and GCN architecture to reproduce the results of Table 4. For baseline models, we

sweep the major design / training parameters to obtain the configuration corresponding to the highest validation $F1_{mic}$. The key parameters that affect accuracy are the dropout rate and the dimension (length) of each hidden activation. To reduce the design space, we keep the dimensions same for all hidden layers. For the two datasets prepared by ourselves (Flickr and Yelp), we train SpGCN [16], GraphSAGE [13] and S-GCN [10] with dimension of 128, 256, 512 and dropout of 0.0, 0.2. For the other two datasets (PPI and Reddit), we use the dimensions and dropout rate as specified in the original source code / paper. All methods (including GraphSAINT) are optimized using Adam Optimizer [15], with a uniform learning rate of 0.01. Table 5 summarizes the dimensions and dropout for each method.

For configuration of GraphSAINT in Table 4, there are two additional parameters: order for the GCN layer and the choice of graph sampler for minibatch construction. For 3-layer Yelp, the result is produced by an "$1-1-2$" architecture (two order-1 layers followed by an order-2 layer). All other GraphSAINT results are produced by architecture consisting of order-1 layers only.

All Table 4 results are obtained by training with minibatches constructed by frontier sampler. The sampler parameters are summarized by Table 6

Table 6: GraphSAINT frontier sampler configuration

Dataset	Frontier size (n)	Subgraph budget (N)	Num. hops (k)
PPI	1800	5000	1
Flickr	3000	8000	1
Reddit	1000	8000	1
Yelp	1000	2000	1

D ADDITIONAL COMPARISON WITH STATE-OF-THE-ART

Here we present additional training accuracy comparison with two other state-of-the-art layer sampling methods: FastGCN [9] and LGCN [12]. Due to difficulties in reusing the available code for different datasets in the case of [9] and [12], we show the comparison with their reported accuracy on PPI and Reddit, respectively. Table 7 summarizes the comparison of $F1_{mic}$ score. GraphSAINT achieves significantly higher accuracy than these two state-of-the-art methods.

Table 7: Additional accuracy comparison with state-of-the-art

Method	Layers	PPI (Test $F1_{mic}$)	Reddit (Test $F1_{mic}$)
FastGCN [9]	2	−−	0.937
LGCN [12]	2	0.772	−−
GraphSAINT	2	0.976	0.964