Sketch-GNN: Scalable Graph Neural Networks with Sublinear Training Complexity

Mucong Ding*
Department of Computer Science
University of Maryland, College Park
College Park, MD 20742
mding@cs.umd.edu

Tahseen Rabbani
Department of Computer Science
University of Maryland, College Park
College Park, MD 20742
trabbani@cs.umd.edu

Bang An
Department of Computer Science
University of Maryland, College Park
College Park, MD 20742
bangan@cs.umd.edu

Evan Wang
Department of Computer Science
University of Maryland, College Park
College Park, MD 20742
ezw@terpmail.umd.edu

Furong Huang
Department of Computer Science
University of Maryland, College Park
College Park, MD 20742
furongh@cs.umd.edu

Abstract

Graph Neural Networks (GNNs) are widely applied to graph learning problems such as node classification. When scaling up the underlying graphs of GNNs to a larger size, we are forced to either train on the complete graph and keep the full graph adjacency and node embeddings in memory (which is often infeasible) or mini-batch sample the graph (which results in exponentially growing computational complexities with respect to the number of GNN layers). Various sampling-based and historical-embedding-based methods are proposed to avoid this exponential growth of complexities. However, none of these solutions eliminates the linear dependence on graph size. This paper proposes a sketch-based algorithm whose training time and memory grow sublinearly with respect to graph size by training GNNs atop a few compact sketches of graph adjacency and node embeddings. Based on polynomial tensor-sketch (PTS) theory, our framework provides a novel protocol for sketching non-linear activations and graph convolution matrices in GNNs, as opposed to existing methods that sketch linear weights or gradients in neural networks. In addition, we develop a locality sensitive hashing (LSH) technique that can be trained to improve the quality of sketches. Experiments on large-graph benchmarks demonstrate the scalability and competitive performance of our Sketch-GNNs versus their full-size GNN counterparts.

1 Introduction

Graph Neural Networks (GNNs) have achieved the state-of-the-art graph learning in numerous applications, including classification [26], clustering [3], recommendation systems [42], social networks [16] and more, through representation learning of target nodes using information aggregated from neighborhoods in the graph. The manner in which GNNs utilize graph topology, however, makes it challenging to scale learning to larger graphs or deeper models with desirable computational

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

and memory efficiency. Full-batch training that stores the Laplacian of the complete graph suffers from a memory complexity of $\mathcal{O}(m + ndL + d^2L)$ on an n-node, m-edge graph with node features of dimension d when employing an L-layer graph convolutional network (GCN). This linear memory complexity dependence on n and the limited memory capacity of GPUs make it difficult to train on large graphs with millions of nodes or more. As an example, the *MAG240M-LSC* dataset [21] is a node classification benchmark with over 240 million nodes that takes over 202 GB of GPU memory when fully loaded.

To address the memory constraints, two major lines of research are proposed: (1) Sampling-based approaches [18, 11, 12, 14, 44] based on the idea of implementing message passing only between the neighbors within a sampled mini-batch; (2) Historical-embedding based techniques, such as GNNAutoScale [17] and VQ-GNN [15]), which maintain the expressive power of GNNs on sampled subgraphs using historical embeddings. However, all of these methods require the number of mini-batches to be proportional to the size of the graph for fixed memory consumption. In other words, they significantly increase computational time complexity in exchange for memory efficiency when scaling up to large graphs. For example, training a 4-layer GCN with just 333K parameters (1.3 MB) for 500 epochs on *ogbn-papers100M* can take more than 2 days on a powerful AWS p4d.24x large instance [21].

We seek to achieve efficient training of GNNs with time and memory complexities sublinear in graph size without significant accuracy degradation. Despite the difficulty of this goal, it should be achievable given that (1) the number of learnable parameters in GNNs is independent of the graph size, and (2) training may not require a traversal of all local neighborhoods on a graph, but rather only the most representative ones (thus sublinear in graph size) as some neighborhoods may be very similar. In addition, commonly-used GNNs are typically small and shallow with limited model capacity and expressive power, indicating that a modest proportion of data may suffice.

This paper presents *Sketch-GNN*, a framework for training GNNs with sublinear time and memory complexity with respect to graph size. Using the idea of sketching, which maps high-dimensional data structures to a lower dimension through entry hashing, we sketch the $n \times n$ adjacency matrix and the $n \times d$ node feature matrix to a few $c \times c$ and $c \times d$ sketches respectively before training, where c is the sketch dimension. While most existing literature focuses on sketching linear weights or gradients, we introduce a method for sketching non-linear activation units using polynomial tensor sketch theory [19]. This preserves prediction accuracy while avoiding the need to "unsketch" back to the original high dimensional graph-node space n, thereby eliminating the dependence of training complexity on the underlying graph size n. Moreover, we propose to learn and update the sketches in an online manner using learnable locality sensitive hashing (LSH) [9]. This reduces the performance loss by adaptively enhancing the sketch quality while incurring minor overhead sublinear in graph size. In practice, we find that the sketch-ratio c/n required to maintain "full-graph" model performance drops as n increases; as a result, our Sketch-GNN enjoys sublinear training scalability.

Sketch-GNN applies sketching techniques to GNNs to achieve training complexity sublinear to the *data* size. This is fundamentally different from the few existing works which sketch the weights or gradients [29, 13, 25, 28, 36] to reduce the memory footprint of the model and speed up optimization. Our approach is flexible to architecture and has the potential to be generalized to other neural networks and data types, e.g., CNNs on gigapixel images. To the best of our knowledge, *Sketch-GNN* is the first sub-linear complexity training algorithm for GNNs based on LSH and tensor sketching. The sublinear efficiency obtained applies not only to GNNs with a fixed convolution matrix, such as GCN [26] and GraphSAGE [18], but also to GNNs with learnable convolution matrices, such as GAT [38].

Experiments on several large graph datasets, such as *ogbn-product* [21] with 2.45M nodes, demonstrate that *Sketch-GNNs* can match the performance of the standard model trained on the complete graph, while requiring significantly reduced computations and memory for both fixed (GCN, GraphSAGE) and learnable convolution (GAT) models. For instance, SketchGCN on *ogbn-arxiv* [21] is 72% and 55% faster than the corresponding full-graph and sampling-based (GraphSAINT [44]) baselines, while the pre-processing time is just 14% of overall reduction (when running 500 epochs).

2 Preliminaries

Basic Notations. Consider a graph with n nodes and m edges. Connectivity is given by the adjacency matrix $A \in \{0, 1\}^{n \times n}$ and features on nodes are represented by the matrix $X \in \mathbb{R}^{n \times d}$, where d is the

number of features. Given a matrix C, let $C_{i,j}$, $C_{i,:}$, and $C_{:,j}$ denote its (i,j)-th entry, i-th row, and j-th column, respectively. $\odot$ denotes the element-wise (Hadamard) product, whereas $C^{\odot k}$ represents the k-th order element-wise power. $\|\cdot\|_F$ is the symbol for the Frobenius norm. $I_n \in \mathbb{R}^{n\times n}$ denotes the identity matrix, whereas $\mathbf{1}_n \in \mathbb{R}^n$ is the vector whose elements are all ones. $\mathsf{Med}\{\cdot\}$ represents the element-wise median over a set of matrices. Superscripts are used to indicate multiple instances of the same kind of variable; for instance, $X^{(l)} \in \mathbb{R}^{n\times d_l}$ are the node representations on layer l.

Unified Framework of GNNs. A *Graph Neural Network (GNN)* layer receives the node representation of the preceding layer $X^{(l)} \in \mathbb{R}^{n\times d}$ as input and outputs a new representation $X^{(l+1)} \in \mathbb{R}^{n\times d}$, where $X = X^{(0)} \in \mathbb{R}^{n\times d}$ are the input features. Although GNNs are designed following different guiding principles, such as neighborhood aggregation (GraphSAGE), spatial convolution (GCN), self-attention (GAT), and Weisfeiler-Lehman (WL) alignment (GIN [43]), the great majority of GNNs can be interpreted as performing message passing on node features, followed by feature transformation and an activation function. The update rule of these GNNs can be summarized as [15]

$$X^{(l+1)} = \sigma\Big(\textstyle\sum_q C^{(q)} X^{(l)} W^{(l,q)}\Big). \tag{1}$$

Where $C^{(q)} \in \mathbb{R}^{n\times n}$ denotes the q-th convolution matrix that defines the message passing operator, $q \in \mathbb{Z}_+$ is index of convolution, $\sigma(\cdot)$ is some choice of nonlinear activation function, and $W^{(l,q)} \in \mathbb{R}^{d_l \times d_{l+1}}$ denotes the learnable linear weight matrix for the l-th layer and q-th filter. GNNs under this paradigm differ from each other by their choice of convolution matrices $C^{(q)}$, which can be either fixed (GCN and GraphSAGE) or learnable (GAT). In Appendix A.1, we re-formulate a number of well-known GNNs under this framework. Unless otherwise specified, we assume $q = 1$ and $d = d_l$ for every layer $l \in [L]$ for notational convenience.

Count Sketch and Tensor Sketch. **(1)** *Count sketch* [7, 40] is an efficient dimensionality reduction method that projects an n-dimensional vector $\boldsymbol{u}$ into a smaller c-dimensional space using a random hash table $h : [n] \to [c]$ and a binary Rademacher variable $s : [n] \to \{\pm 1\}$, where $[n] = \{1, \ldots, n\}$. Count sketch is defined as $\mathsf{CS}(\boldsymbol{u})_i = \sum_{h(j)=i} s(j)\boldsymbol{u}_j$, which is a linear transformation of $\boldsymbol{u}$, i.e., $\mathsf{CS}(\boldsymbol{u}) = R\boldsymbol{u}$. Here, $R \in \mathbb{R}^{c\times n}$ denotes the so-called *count sketch matrix*, which has exactly one non-zero element per column. **(2)** *Tensor sketch* [31] is proposed as a generalization of count sketch to the tensor product of vectors. Given $\boldsymbol{z} \in \mathbb{R}^n$ and an order k, consider a k number of i.i.d. hash tables $h^{(1)}, \ldots, h^{(k)} : [n] \to [c]$ and i.i.d. binary Rademacher variables $s^{(1)}, \ldots, s^{(k)} : [n] \to \{\pm 1\}$. Tensor sketch also projects vector $\boldsymbol{z} \in \mathbb{R}^n$ into $\mathbb{R}^c$, and is defined as $\mathsf{TS}_k(\boldsymbol{z})_i = \sum_{h(j_1,\cdots,j_k)=i} s^{(1)}(j_1)\cdots s^{(k)}(j_k)\boldsymbol{z}_{j_1}\cdots\boldsymbol{z}_{j_k}$, where $h(j_1, \cdots, j_k) = (h^{(1)}(j_1) + \cdots + h^{(k)}(j_k)) \bmod c$. By definition, a tensor sketch of order $k = 1$ degenerates to count sketch; $\mathsf{TS}_1(\cdot) = \mathsf{CS}(\cdot)$. **(3)** We define *count sketch of a matrix* $U \in \mathbb{R}^{d\times n}$ as the count sketch of each row vector individually, i.e., $\mathsf{CS}(U) \in \mathbb{R}^{d\times c}$ where $[\mathsf{CS}(U)]_{i,:} = \mathsf{CS}(U_{i,:})$. The *tensor sketch of a matrix* is defined in the same way. Pham and Pagh [31] devise a fast computation of tensor sketch of $U \in \mathbb{R}^{d\times n}$ (sketch dimension c and order k) using count sketches and the Fast Fourier Transform (FFT):

$$\mathsf{TS}_k(U) = \mathsf{FFT}^{-1}\Big(\textstyle\bigodot_{p=1}^{k} \mathsf{FFT}\big(\mathsf{CS}^{(p)}(U)\big)\Big), \tag{2}$$

where $\mathsf{CS}^{(p)}(\cdot)$ is the count sketch with hash function $h^{(p)}$ and Rademacher variable $s^{(p)}$. $\mathsf{FFT}(\cdot)$ and $\mathsf{FFT}^{-1}(\cdot)$ are the FFT and its inverse applied to each row of a matrix.

Locality Sensitive Hashing. The definition of count sketch and tensor sketch is based on hash table(s) that only requires a data independent uniformity, i.e., with high probability the hash-buckets are of similar size. In contrast, locality sensitive hashing (LSH) is a hashing scheme that uses locality-sensitive hash function $H : \mathbb{R}^d \to [c]$ to ensure that nearby vectors are hashed into the same bucket (out of c buckets in total) with high probability while distant ones are not. *SimHash* achieves the locality-sensitive property by employing random projections [8]. Given a random matrix $P \in \mathbb{R}^{c/2\times d}$, SimHash defines a locality-sensitive hash function

$$H(\boldsymbol{u}) = \arg\max\left([P\boldsymbol{u} \parallel -P\boldsymbol{u}]\right), \tag{3}$$

where $[\cdot \parallel \cdot]$ denotes concatenation of two vectors and $\arg\max$ returns the index of the largest element. SimHash is efficient for large batches of vectors [1]. In this paper, we apply a learnable version of SimHash that is proposed by Chen et al. [9], in which the projection matrix P is updated using gradient descent; see Section 3.3 for details.

3 Sketch-GNN Framework via Polynomial Tensor Sketch

Problem and Insights. We intend to develop a "sketched counterpart" of GNNs, where training is based solely on (dimensionality-reduced) compact sketches of the convolution and node feature matrices, the sizes of which can be set independently of the graph size n. In each layer, Sketch-GNN receives some sketches of the convolution matrix C and node representation matrix $X^{(l)}$ and outputs some sketches of the node representations $X^{(l+1)}$. As a result, the memory and time complexities are inherently independent of n. The bottleneck of this problem is estimating the nonlinear activated product $\sigma(CX^{(l)}W^{(l)})$, where $W^{(l)}$ is the learnable weight of the l-th layer.

Before considering the nonlinear activation, as a first step, we approximate the linear product $CX^{(l)}W^{(l)}$, using dimensionality reduction techniques such as random projections and low-rank decompositions. As a direct corollary of the (distributional) Johnson–Lindenstrauss (JL) lemma [24], there exists a projection matrix $R \in \mathbb{R}^{c\times n}$ such that $CX^{(l)}W^{(l)} \approx \big(CR^\mathsf{T}\big)\big(RX^{(l)}W^{(l)}\big)$ [15]. Tensor sketch is one of the techniques that can achieve the JL bound [2]; for an error bound, see Lemma 1 in Appendix B.

Count sketch offers a good estimation of a matrix product, $CX^{(l)}W^{(l)} \approx \mathsf{CS}(C)\mathsf{CS}((X^{(l)}W^{(l)})^\mathsf{T})^\mathsf{T}$. While tensor sketch can be used to approximate the power of matrix product, i.e., $(CX^{(l)}W^{(l)})^{\odot k} \approx \mathsf{TS}_k(C)\mathsf{TS}_k((X^{(l)}W^{(l)})^\mathsf{T})^\mathsf{T}$, where $(\cdot)^{\odot k}$ is the k-th order element-wise power. If we combine the estimators of element-wise powers of $CX^{(l)}W^{(l)}$, we can approximate the (element-wise) activation $\sigma(\cdot)$ on $CX^{(l)}W^{(l)}$. This technique is known as a *polynomial tensor sketch (PTS)* and is discussed in [19]. In this paper, we apply PTS to sketch the message passing of GNNs, including the nonlinear activations.

3.1 Sketch-GNN: Approximated Update Rules

Polynomial Tensor Sketch. Our goal is to approximate the update rule of GNNs (Eq. (1)) in each layer. We first expand the element-wise non-linearity σ as a power series, and then approximate the powers using count/tensor sketch, i.e.,

$$X^{(l+1)} = \sigma(CX^{(l)}W^{(l)}) \approx \sum\nolimits_{k=1}^{r} c_k \left(CX^{(l)}W^{(l)}\right)^{\odot k} \approx \sum\nolimits_{k=1}^{r} c_k\, \mathsf{TS}_k(C)\, \mathsf{TS}_k\big((X^{(l)}W^{(l)})^\mathsf{T}\big)^\mathsf{T}, \quad (4)$$

where the $k = 0$ term always evaluates to zero as $\sigma(0) = 0$. In Eq. (4), coefficients c_k are introduced to enable learning or data-driven selection of the weights when combing the terms of different order k. This allows for the approximation of a variety of nonlinear activation functions, such as sigmoid and ReLU. The error of this approximation relies on the precise estimation of the coefficients $\{c_k\}_{k=1}^r$. To identify the coefficients, Han et al. [19] design a coreset-based regression algorithm, which requires at least $O(n)$ additional time and memory. Since the coefficients $\{c_k\}_{k=1}^r$ that achieve the best performance for the classification tasks do not necessarily approximate a known activation, we propose learning the coefficients $\{c_k\}_{k=1}^r$ to optimize the classification loss directly using gradient descent with simple L_2 regularization. Experiments indicate that the learned coefficients can approximate the sigmoid activation with relative errors comparable to those of the coreset-based method; see Fig. 1a in Section 5.

Approximated Update Rules. The remaining step is to approximate the operations of GNNs using PTS (Eq. (4)) on sketches of convolution matrix C and node representation matrix $X^{(l)}$. Consider r pairwise-independent count sketches $\{\mathsf{CS}^{(k)}(\cdot)\}_{k=1}^r$ with sketch dimension c, associated with hash tables $h^{(1)}, \ldots, h^{(r)}$ and binary Rademacher variables $s^{(1)}, \ldots, s^{(r)}$, defined prior to training an L-layer *Sketch-GNN*. Using these hash tables and Rademacher variables, we may also construct tensor sketches $\{\mathsf{TS}_k(\cdot)\}_{k=2}^r$ up to the maximum order r.

In *Sketch-GNN*, sketches of node representations (instead of the $O(n)$ standard representation) are propagated between layers. To get rid of the dependence on n, we count sketch both sides of Eq. (4)

$$\begin{aligned} S_X^{(l+1,k')} &:= \mathsf{CS}^{(k')}\big((X^{(l+1)})^\mathsf{T}\big) \approx \mathsf{CS}^{(k')}\big(\textstyle\sum_{k=1}^{r} c_k^{(l)} \mathsf{TS}_k\big((X^{(l)}W^{(l)})^\mathsf{T}\big)\mathsf{TS}_k(C)^\mathsf{T}\big) \\ &= \textstyle\sum_{k=1}^{r} c_k^{(l)}\, \mathsf{TS}_k\big((X^{(l)}W^{(l)})^\mathsf{T}\big)\mathsf{CS}^{(k')}\big(\mathsf{TS}_k(C)^\mathsf{T}\big) \\ &= \textstyle\sum_{k=1}^{r} c_k^{(l)} \mathsf{FFT}^{-1}\Big(\bigodot_{p=1}^{k} \mathsf{FFT}\big((W^{(l)})^\mathsf{T} S_X^{(l,p)}\big)\Big)\, S_C^{(l,k,k')}, \end{aligned} \quad (5)$$

where $S_X^{(l+1,k')} = \mathsf{CS}^{(k')}((X^{(l+1)})^\mathsf{T}) \in \mathbb{R}^{d\times c}$ is the transpose of column-wise count sketch of $X^{(l+1)}$, and the superscripts of $S_X^{(l+1,k')}$ indicate that it is the k'-th count sketch of $X^{(l+1)}$ (i.e., sketched by $CS^{(k)}(\cdot)$). In the second line of Eq. (5), we can move the matrix, $c_k^{(l)}\mathsf{TS}_k((X^{(l)}W^{(l)})^\mathsf{T})$, multiplied on the left to $\mathsf{TS}_k(C)^\mathsf{T}$ out of the count sketch function $\mathsf{CS}^{(k')}(\cdot)$, since the operation of

row-wise count sketch $\mathsf{CS}^{(k')}(\cdot)$ is equivalent to multiplying the associated count sketch matrix $R^{(k')}$ on the right, i.e., for any $U \in \mathbb{R}^{n\times n}$, $\mathsf{CS}^{(k')}(U) = UR^{(k')}$. In the third line of Eq. (5), we denote the "two-sided sketch" of the convolution matrix as $S_C^{(l,k,k')} := \mathsf{CS}^{(k')}(\mathsf{TS}_k(C)^\mathsf{T}) \in \mathbb{R}^{c\times c}$ and expand the tensor sketch $\mathsf{TS}_k((X^{(l)}W^{(l)})^\mathsf{T})$ using the FFT-based formula (Eq. (2)).

Eq. (5) is the (recursive) **update rule** of *Sketch-GNN*, which approximates the operation of the original GNN and learns the sketches of representations. Looking at the both ends of Eq. (5), we obtain a formula that approximates the sketches of $X^{(l+1)}$ using the sketches of $X^{(l)}$ and C, with learnable weights $W^{(l)} \in \mathbb{R}^{d\times d}$ and coefficients $\{c_k^{(l)} \in \mathbb{R}\}_{k=1}^r$. The forward-pass and backward-propagation between the input sketches $\{S_X^{(0,k)}\}_{k=1}^r$ and the sketches of the final layer representations $\{S^{(L,k)}\}_{k=1}^r$ take $O(c)$ time and memory (see Section 3.3 for complexity details).

3.2 Error Bound on Estimated Representation

Based on Lemma 1 and the results in [19], we establish an error bound on the estimated final layer representation $\widetilde{X}^{(L)}$ for GCN; see Appendix B for the proof and discussions.

Theorem 1. *For a Sketch-GNN with L layers, the estimated final layer representation is $\widetilde{X}^{(L)} = \mathsf{Med}\{R^{(k)}S_X^{(L,k)} \mid k = 1,\cdots,r\}$, where the sketches are recursively computed using Eq. (5). For $\Gamma^{(l)} = \max\{5\|X^{(l)}W^{(l)}\|_F^2, (2+3^r)\sum_i(\sum_j[X^{(l)}W^{(l)}]_{i,j})^r\}$, it holds that $\mathbf{E}(\|X^{(L)} - \widetilde{X}^{(L)}\|_F^2)/\|X^{(L)}\|_F^2 \le \prod_{l=1}^L(1 + 2/(1 + c{\lambda^{(l)}}^2/nr\Gamma^{(l)})) - 1$, where $\lambda^{(l)} \ge 0$ is the smallest singular value of the matrix $Z \in \mathbb{R}^{nd\times r}$ and $Z_{:,k}$ is the vectorization of $(CX^{(l)}W^{(l)})^{\odot k}$. Moreover, if $(c(\lambda^{(l)})^2/nr\Gamma^{(l)}) \gg 1$ holds true for every layer, the relative error is $O(L(n/c))$, which is proportional to the depth of the model, and inversely proportional to the sketch ratio (c/n).*

Remarks. Despite the fact that in Theorem 1 the error bound grows for smaller sketch ratios c/n, we observe in experiments that the sketch-ratio required for competitive performance decreases as n increases; see Section 5. As for the number of independent sketches r, we know from Lemma 1 that the dependence of r on n is $r = \Omega(3^{\log_c n})$ which is negligible when n is not too small; thus, in practice $r = 3$ is used.

The theoretical framework may not completely correspond to reality. Experimentally, the coefficients $\{\{c_k^{(l)}\}_{k=1}^r\}_{l=1}^L$ with the highest performance do not necessarily approximate a known activation. We defer the challenging problem of bounding the error of sketches and coefficients learned by gradients to future studies. Although the error bound is in expectation, we do not train over different sketches per iteration due to the instability caused by randomness. Instead, we introduce learnable locality sensitive hashing (LSH) in the next section to counteract the approximation limitations caused by the fixed number of sketches.

3.3 A Practical Implementation: Learning Sketches using LSH

Motivations of Learnable Sketches. In Section 3, we apply polynomial tensor sketch (PTS) to approximate the operations of GNNs on sketches of the convolution and feature matrices. Nonetheless, the pre-computed sketches are fixed during training, resulting in **two major drawbacks**: **(1)** The performance is limited by the quality of the initial sketches. For example, if the randomly-generated hash tables $\{h^{(k)}\}_{k=1}^r$ have unevenly distributed buckets, there will be more hash collisions and consequently worse sketch representations. The performance will suffer because only sketches are used in training. **(2)** More importantly, when multiple Sketch-GNN layers are stacked, the input representation $X^{(l)}$ changes during training (starting from the second layer). Fixed hash tables are not tailored to the "changing" hidden representations.

We seek a method for efficiently constructing high-quality hash tables tailored for each hidden embedding. Locality sensitive hashing (LSH) is a suitable tool since it is data-dependent and preserves data similarity by hashing similar vectors into the same bucket. This can significantly improve the quality of sketches by reducing the errors due to hash collisions.

Combining LSH with Sketching. At the time of sketching, the hash table $h^{(k)} : [n] \to [c]$ is replaced with an LSH function $H^{(k)} : \mathbb{R}^d \to [c]$, for any $k \in [r]$. Specifically, in the l-th layer of a Sketch-GNN, we hash the i-th node to the $H^{(k)}(X_{i,:}^{(l)})$-th bucket for every $i \in [n]$, where $X_{i,:}^{(l)}$ is the

embedding vector of node i. As a result, we define a data-dependent hash table

$$h^{(l,k)}(i) = H^{(k)}(X^{(l)}_{i,:}) \tag{6}$$

that can be used for computing the sketches of $S_X^{(l,k)}$ and $S_C^{(l,k,k')}$. This LSH-based sketching can be directly applied to sketch the fixed convolution matrix and the input feature matrix. If SimHash is used, i.e., $H^{(k)}(\boldsymbol{u}) = \arg\max\left(\left[P^{(k)}\boldsymbol{u} \parallel -P^{(k)}\boldsymbol{u}\right]\right)$ (Eq. (3)), an additional $O(ncr(\log c + d))$ computational overhead is introduced to hash the n nodes for the r hash tables during preprocessing; see Appendix F more information. SimHash(es) are implemented as simple matrix multiplications that are practically very fast.

In order to employ LSH-based hash functions customized to each layer to sketch the hidden representations of a Sketch-GNN (i.e., $l = 2, \ldots, L-1$), we face **two major challenges**: **(1)** Unless we explicitly unsketch in each layer, the estimated hidden representations $\widetilde{X}^{(l)}(l = 2, \ldots, L-1)$ cannot be accessed and used to compute the hash tables. However, unsketching any hidden representation, i.e., $\widetilde{X}^{(l)} = \mathsf{Med}\{R^{(k)}S_X^{(l,k)} \mid k = 1, \cdots, r\}$, requires $O(n)$ memory and time. We need to come up with an efficient algorithm that updates the hash tables without having to unsketch the complete representation. **(2)** It's unclear how to change the underlying hash table of a sketch across layers without unsketching to the n-dimensional space, even if we know the most up-to-date hash tables suited to each layer.

The **challenge (2)**, i.e., changing the underlying hash table of across layers, can be solved by maintaining a sparse $c \times c$ matrix $T^{(l,k)} := R^{(l,k)}(R^{(l+1,k)})^\mathsf{T}$ for each $k \in [r]$, which only requires $O(cr)$ memory and time overhead; see Appendix C for more information and detailed discussions. We focus on **challenge (1)** for the remainder of this section.

Online Learning of Sketches. To learn a hash table tailored for a hidden layer using LSH without unsketching, we develop an efficient algorithm to update the LSH function using only a size-$|B|$ subset of the length-n unsketched representations, where B denotes a subset of nodes we select. This algorithm, which we term *online learning of sketches*, is made up of two key parts: *(Part 1)* select a subset of nodes $B \subseteq [n]$ to effectively update the hash table, and *(Part 2)* update the LSH function $H(\cdot)$ with a triplet loss computed using this subset.

(1) Selection of subset B: Because model parameters are updated slowly during neural network training, the data-dependent LSH hash tables also changes slowly (this behavior was detailed in [9]). The amount of updates to the hash table drops very fast along with training, empirically verified in Fig. 1b (left) in Section 5. Based on this insight, we only need to update a small fraction of the hash table during training. To identify this subset $B \in [n]$ of nodes, gradient signals can be used. Intuitively, a node representation vector hashed into the wrong bucket will be aggregated with distant vectors and lead to larger errors and subsequently larger gradient signals. Specifically, we propose finding the candidate set B of nodes by taking the union of the several buckets with the largest gradients, i.e., $B = \{i \mid h^{(l,k)}(i) = \arg\max_j [S_X^{(l,k)}]_{j,:} \text{ for some } k\}$. The memory and overhead required to update the entries in B in the hash table is $O(|B|)$.

(2) Update of LSH function: In order to update the projection matrix P that defines a SimHash $H^{(k)} : \mathbb{R}^d \to [c]$ (Eq. (3)), instead of the $O(n)$ full triplet loss introduced by [9], we consider a sampled version of the triplet loss on the candidate set B with $O(|B|)$ complexity, namely

$$\mathcal{L}(H, \mathcal{P}_+, \mathcal{P}_-) = \max\left\{0, \textstyle\sum_{(\boldsymbol{u},\boldsymbol{v})\in\mathcal{P}_-} \cos(H(\boldsymbol{u}), H(\boldsymbol{v})) - \sum_{(\boldsymbol{u},\boldsymbol{v})\in\mathcal{P}_+} \cos(H(\boldsymbol{u}), H(\boldsymbol{v})) + \alpha\right\}, \tag{7}$$

where $\mathcal{P}_+ = \{(\widetilde{X}_{i,:}, \widetilde{X}_{j,:}) \mid i, j \in B, \langle \widetilde{X}_{i,:}, \widetilde{X}_{j,:}\rangle > t_+\}$ and $\mathcal{P}_- = \{(\widetilde{X}_{:,i}, \widetilde{X}_{:,j}) \mid i, j \in B, \langle \widetilde{X}_{:,i}, \widetilde{X}_{:,j}\rangle < t_-\}$ are the similar and dissimilar node-pairs in the subset B; $t_+ > t_-$ and $\alpha > 0$ are hyper-parameters. This triplet loss $\mathcal{L}(H, \mathcal{P}_+, \mathcal{P}_-)$ is used to update P using gradient descent, as described in [9], with a $O(c|B|d + |B|^2)$ overhead. Experimental validation of this LSH update mechanism can be found in Fig. 1b in Section 5.

Avoiding $O(n)$ in Loss Evaluation. We can estimate the final layer representation using the r sketches $\{S^{(L,k)}\}_{k=1}^r$, i.e., $\widetilde{X}^{(L)} = \mathsf{Med}\{R^{(k)}S_X^{(L,k)} \mid k = 1, \cdots, r\}$ and compute the losses of all nodes for node classification (or some node pairs for link prediction). However, the complexity of loss evaluation is $O(n)$, proportional to the number of ground-truth labels. In order to avoid $O(n)$ complexity completely, rather than un-sketching the node representation for all labeled nodes, we employ the locality sensitive hashing (LSH) technique again for loss calculation so that only a subset

of node losses are evaluated based on a set of hash tables. Specifically, we construct an LSH hash table for each class in a node classification problem, which indexes all of the labeled nodes of this class and can be utilized to choose the nodes with poor predictions by leveraging the locality property. This technique, introduced in [10], is known as sparse forward-pass and back-propagation, and we defer the descriptions to Appendix C.

One-time Preprocessing. If the convolution matrix C is fixed (GCN, GraphSAGE), the "two-sided sketch" $S_C^{(l,k,k')} = \mathsf{CS}^{(k')}(\mathsf{TS}_k(C)^\mathsf{T}) \in \mathbb{R}^{c\times c}$ is the same in each layer and may be denoted as $S_C^{(k,k')}$. In addition, all of the r^2 sketches of C, i.e., $\{\{S_C^{(k,k')} \in \mathbb{R}^{c\times c}\}_{k=1}^r\}_{k'=1}^r$ can be computed during the preprocessing phase. If the convolution matrix C is sparse (which is true for most GNNs following Eq. (1) on a sparse graph), we can use the sparse matrix representations for the sketches $\{\{S_C^{(k,k')} \in \mathbb{R}^{c\times c}\}_{k=1}^r\}_{k'=1}^r$, and the total memory taken by the r^2 sketches is $O(r^2c(m/n))$ where $(2m/n)$ is the average node degree (see Appendix F for details). We also need to compute the r count sketches of the input node feature matrix $X = X^{(0)}$, i.e., $\{S_X^{(0,k)}\}_{k=1}^r$ during preprocessing, which requires $O(rcd)$ memory in total. In this regard, we have substituted the input data with compact graph-size independent sketches (i.e., $O(c)$ memory). Although the preprocessing time required to compute these sketches is $O(n)$, it is a one-time cost prior to training, and it is widely known that sketching is practically very fast.

Complexities of Sketch-GCN. The theoretical complexities of Sketch-GNN is summarized as follows, where for simplicity we assume bounded maximum node degree, i.e., $m = O(n)$. **(1) Training Complexity**: *(1a) Forward and backward propagation*: $O(Lcrd(\log(c) + d + m/n)) = O(c)$ time and $O(Lr(cd + rm/n)) = O(c)$ memory. *(1b) Hash and sketch update*: $O(Lr(c + |B|d)) = O(c)$ time and memory. **(2) Preprocessing**: $O(r(rm + n + c)) = O(n)$ time and $O(rc(d + rm/n)) = O(c)$ memory. **(3) Inference**: $O(Ld(m + nd)) = O(n)$ time and $O(m + Ld(n + d)) = O(n)$ memory (the same as a standard GCN). We defer a detailed summary of the theoretical complexities of Sketch-GNN to Appendix F.

We generalize Sketch-GNN to more GNN models in Appendix D and the pseudo-code which outlines the complete workflow of Sketch-GNN can be find in Appendix E.

4 Related Work

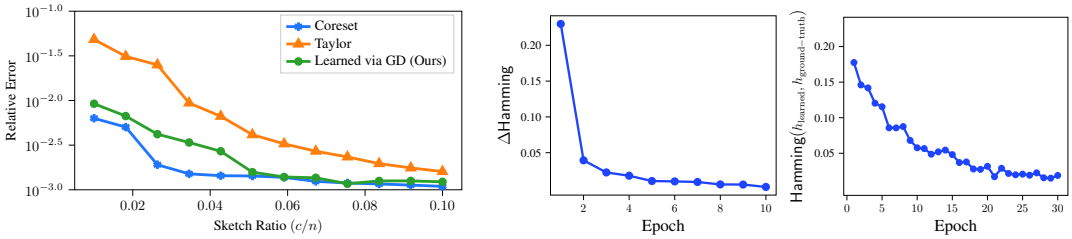
Scalable methods for GNNs can be categorized into four classes, all of them still require linear training complexities. **(A)** On a large sparse graph with n nodes and m edges, the "full-graph" training of a L-layer GCN with d-dimensional (hidden) features per layer requires $O(m+ndL+d^2L)$ memory and $O(mdL + nd^2L)$ epoch time. **(B)** Sampling-based methods sample mini-batches from the complete graph following three schemes: (1) node-wisely sample a subset of neighbors in each layer to reduce the neighborhood size; (2) layer-wisely sample a set of nodes independently in each layer; (3) subgraph-wisely sample a subgraph directly and simply forward-pass and back-propagate on that subgraph. **(B.1)** GraphSAGE [18] samples r neighbors for each node while ignoring messages from other neighbors. $O(br^L)$ nodes are sampled in a mini-batch (where b is the mini-batch size), and the epoch time is $O(ndr^L)$; therefore, GraphSAGE is impractical for deep GNNs on a large graph. FastGCN [12] and LADIES [46] are layer-sampling methods that apply importance sampling to reduce variance. **(B.2)** The subgraph-wise scheme has the best performance and is most prevalent. Cluster-GCN [14] partitions the graph into many densely connected subgraphs and samples a subset of subgraphs (with edges between subgraphs added back) for training per iteration. GraphSAINT [44] samples a set of nodes and uses the induced subgraph for mini-batch training. Both Cluster-GCN and GraphSAINT require $O(mdL + nd^2L)$ epoch time, which is the same as "full-graph" training, although Cluster-GCN also needs $O(m)$ pre-processing time. **(C)** Apart from sampling strategies, historical-embedding-based methods propose mitigating sampling errors and improving performance using some stored embeddings. GNNAutoScale [17] keeps a snapshot of all embeddings in CPU memory, leading to a large $O(ndL)$ memory overhead. VQ-GNN [15] maintains a vector quantized data structure for the historical embeddings, whose size is independent of n. **(D)** Linearized GNNs [41, 4, 32] replace the message passing operation in each layer with a one-time message passing during preprocessing. They are practically efficient, but the theoretical complexities remain $O(n)$. Linearized models usually over-simplify the corresponding GNN and limit its expressive power.

Towards sublinear GNNs. Nearly all existing scalable methods focus on mini-batching the large graph and resolving the memory bottleneck of GNNs, without reducing the epoch training time.

Few recent work focus on graph compression [22, 23] can also achieve sublinear training time by coarsening (e.g., using [30]) the graph during preprocessing and training GNNs on the coarsened graph with fewer nodes and edges. Nevertheless, this strategy suffers from **two issues**: **(1)** Although graph coarsening is a one-time cost, the memory and time overheads are often worse than $O(n)$ and can be prohibitively large on graphs with over 100K nodes. Even the fastest graph coarsening algorithm used by [22] takes more than 68 minutes to process the 233K-node *Reddit* graph [44]; see Table 1. The long preprocessing time renders any training speedups meaningless. **(2)** The test performance of a model trained on the coarsened graph highly depends on the GNN type. Although the performance of [22] on GCN is good, significant performance degradations are observed on GraphSAGE and GAT; see Section 5.

We defer discussion of more scalable GNN papers and the broad literature of sketching and LHS for neural networks to Appendix G.

5 Experiments



(a) Error of PTS (b) Learnable LSH

Figure 1: **Figure 1a** Relative errors when applying polynomial tensor sketch (PTS) to the nonlinear unit $\sigma(CXW)$ following Eq. (4). The dataset used is Cora [33]. σ is the sigmoid activation. We set $r = 5$ and test on a GCN with fixed $W = I_d \in \mathbb{R}^{d\times d}$. The coefficients $\{c_k\}_{k=1}^r$ can be computed by a coreset regression [19] (blue), by a Taylor expansion of $\sigma(\cdot)$ (orange), or learned from gradient descent proposed by us (green). **Figure 1b** The left plot shows the Hamming distance changes of the hash table in the 2nd layer during the training of a 2-layer *Sketch-GCN*, where the hash table is constructed from the unsketched representation $\widetilde{X}^{(1)}$ using SimHash. The right plot shows the Hamming distances between the hash table learned using our algorithm and the hash table constructed directly from $\widetilde{X}^{(1)}$.

Table 1: Time and memory efficiencies of Sketch-GNN versus other scalable methods.

Benchmark	*ogbn-arxiv*			*Reddit*		
Efficiency Measure	Preprocessing Time	Epoch Time	Train Memory	Preprocessing Time	Epoch Time	Train Memory
"Full-Graph" (oracle)	—	0.49 s	983 MB	—	OOM[1]	OOM
GraphSAINT	—	0.30 s	31.4 MB	—	2.09 s	977 MB
VQ-GNN	—	0.37 s	48.9 MB	—	2.16 s	1281 MB
Coarsening	358 s	0.20 s	22.1 MB	4123 s	1.04 s	530 MB
Sketch-GNN (ours)	27 s	0.13 s	38.7 MB	141 s	0.81 s	748 MB

[1] "OOM" refers to "out of memory".

Table 2: Performance of Sketch-GNN in comparison to Graph Coarsening [22] on *ogbn-arxiv*.

Benchmark	*ogbn-arxiv*								
GNN Model	GCN			GraphSAGE			GAT		
"Full-Graph" (oracle)	.7174 ± .0029			.7149 ± .0027			.7233 ± .0045		
Sketch Ratio (c/n)	0.1	0.2	0.4	0.1	0.2	0.4	0.1	0.2	0.4
Coarsening	.6508 ± .0091	.6665 ± .0010	.6892 ± .0035	.5264 ± .0251	.5996 ± .0134	.6609 ± .0061	.5177 ± .0028	.5946 ± .0027	.6307 ± .0041
Sketch-GNN (ours)	.6913 ± .0154	.7004 ± .0096	.7028 ± .0087	.6929 ± .0194	.6963 ± .0056	.7048 ± .0080	.6967 ± .0067	.6910 ± .0135	.7053 ± .0034

Table 3: Performance of Sketch-GNN versus SGC [41], GraphSAINT [44], and VQ-GNN [15].

Benchmark	*ogbn-arxiv*			*Reddit*			*ogbn-product*		
SGC	.6944 ± .0005			.9464 ± .0011			.6683 ± .0029		
GNN Model	GCN	GraphSAGE	GAT	GCN	GraphSAGE	GAT	GCN	GraphSAGE	GAT
"Full-Graph" (oracle)	.7174 ± .0029	.7149 ± .0027	.7233 ± .0045	OOM	OOM +/	OOM	OOM	OOM	OOM
GraphSAINT	.7079 ± .0057	.6987 ± .0039	.7117 ± .0032	.9225 ± .0057	.9581 ± .0074	.9431 ± .0067	.7602 ± .0021	.7908 ± .0024	.7971 ± .0042
VQ-GNN	.7055 ± .0033	.7028 ± .0047	.7043 ± .0034	.9399 ± .0021	.9449 ± .0024	.9438 ± .0059	.7524 ± .0032	.7809 ± .0019	.7823 ± .0049
Sketch Ratio (c/n)	0.4			0.3			0.2		
Sketch-GNN (ours)	.7028 ± .0087	.7048 ± .0080	.7053 ± .0034	.9280 ± .0034	.9485 ± .0061	.9326 ± .0063	.7659 ± .0086	.7851 ± .0071	.7797 ± .0101

In this section, we evaluate the proposed *Sketch-GNN* algorithm and compare it with the (oracle) "full-graph" training baseline, a graph-coarsening based method (**Coarsening** [22]) which has sublinear training time, and other scalable methods including: a sampling-based method (**GraphSAINT** [44]), a historical-embedding based method (**VQ-GNN** [15]), and a linearized GNN (**SGC** [41]). We test

on several large graph benchmarks including *ogbn-arxiv* (169K nodes, 1.2M edges), *Reddit* (233K nodes, 11.6M edges), and *ogbn-products* (2.4M nodes, 61.9M edges) from [20, 44]. See Appendix H for the implementation details.

Proof-of-Concept Experiments: (1) Errors of gradient-learned PTS coefficients: In Fig. 1a, we train the PTS coefficients to approximate the sigmoid activated $\sigma(CXW)$ to evaluate its approximation power to the ground-truth activation. The relative errors are comparable to those of the coreset-based method. **(2) Slow-change phenomenon of LSH hash tables**: In Fig. 1b (left), we count the changes of the hash table constructed from an unsketched hidden representation for each epoch, characterized by the Hamming distances between consecutive updates. The changes drop rapidly as training progresses, indicating that apart from the beginning of training, the hash codes of most nodes do not change at each update. **(3) Sampled triplet loss for learnable LSH**: In Fig. 1b (right), we verify the effectiveness of our update mechanism for LSH hash functions, as the learned hash table gradually approaches the "ground truth", i.e., the hash table constructed from the unsketched hidden representation.

Efficiency of Sketch-GNNs. For efficiency measures, we are interested in the comparison to Coarsening, as both approaches achieve sublinear training time at the cost of some preprocessing overheads. We use a 3-layer GCN as the backbone and set the sketch ratios (c/n, ratio of sketch dimension c to graph size n) of both algorithms to $c/n = 0.1$, meaning that the coarsened graph contains $n/10$ nodes. We measure their preprocessing time, average epoch training time, and peak training memory, as reported in Table 1. Although not rigorously comparable, we also set the mini-batch size of GraphSAINT and VQ-GNN to $b = n/10$. We report the average epoch training time and peak training memory for each method and the "full-graph" training baseline. In addition to Table 1, the following are also recorded: (1) Coarsening requires 980 MB to preprocess *ogbn-arxiv*, whereas Sketch-GNN only requires 539 MB. (2) Our preprocessing on the largest dataset, *ogbn-product* (2.4M nodes), takes only 414s. (3) The wallclock time for the validation accuracy to reach 99% of its best is 88 ± 8s for SketchGCN, which is shorter than VQ-GNN's 103 ± 11s and GraphSAINT's 120 ± 4s.

From Table 1 and the aforementioned results, we can draw **four important conclusions**: **(1)** Sketch-GNN achieves the fastest average epoch time. The coarsened graph is typically much denser and increases the time required for message passing. **(2)** Sketch-GNN usually converges faster than GraphSAINT and VQ-GNN. **(3)** Our preprocessing time is significantly less than that of Coarsening. Coarsening suffers from an extremely long preprocessing time, rendering the training speed-ups meaningless. Moreover, our preprocessing time scales well with graph size and sparsity. **(4)** We also require less preprocessing memory as sketching is linear/multi-linear operation and usually preserves sparsity. **(5)** Sketch-GNN often requires more training memory than Coarsening in order to maintain the copies of sketches and additional data structures, although these memory overheads are small.

Performance of Sketch-GNNs. We first compare the performance of *Sketch-GNN* with Coarsening under various sketch ratios to understand how their performance is affected by the memory bottleneck. In Table 2, we report the test accuracy of both approaches on *ogbn-arxiv*, with a 3-layer GCN, GraphSAGE, or GAT as the backbone and a sketch ratio of 0.1, 0.2, or 0.4. We see there are significant performance degradations when applying Coarsening to GraphSAGE and GAT, even under sketch ratio 0.4, indicating that Coarsening may be compatible only with specific GNNs (GCN and APPNP as explained in [22]). In contrast, the performance drops of Sketch-GNN are always small across all architectures, even when the sketch ratio is 0.1. Therefore, our approach generalizes to more GNN architectures and consistently outperforms the Coarsening method.

We move on to compare Sketch-GNN with linearized GNNs (SGC), sampling-based (GraphSAINT), and historical-embedding-based (VQ-GNN) methods. In Table 3, we report the performance of SGC, the "full-graph" training (oracle), GraphSAINT and VQ-GNN with mini-batch size 50K (their performance is not affected by the choice of mini-batch size if it is not too small), and Sketch-GNN with appropriate sketch ratios (0.4 on *ogbn-arxiv*, 0.3 on *Reddit*, and 0.2 on *ogbn-product*). From Table 3, we confirm that, with an appropriate sketch ratio, the performance of *Sketch-GNN* is always close to the "full-graph" oracle and competitive with the other scalable approaches. Impressively, the needed sketch ratio c/n for Sketch-GNN to achieve competitive performance reduces as graph size grows. This further illustrates that, as previously indicated, the required training complexities (to get acceptable performance) are sublinear to the graph size.

Ablation Studies: (1) Dependence of sketch dimension c on graph size n. Although the theoretical approximation error increases under smaller sketch ratio c/n, we observe competitive experimental

results with smaller c/n especially on large graphs. In practice, the sketch-ratio required to maintain "full-graph" model performance decreases with n, as verified in Table 3: $c/n = 0.4$ is needed on *ogbn-arxiv* with 169K nodes but $c/n = 0.2$ is adequate on *ogbn-product* with 2.45M nodes. **(2) Learned Sketches versus Fixed Sketches.** We find that learned sketches can improve the performance of all models and on all datasets. Under sketch-ratio $c/n = 0.2$, the Sketch-GCN with learned sketches achieves 0.7004 ± 0.0096 accuracy on *ogbn-arxiv* while fixed randomized sketches degrade performance to 0.6649 ± 0.0106.

6 Conclusion

We present *Sketch-GNN*, a sketch-based GNN training framework with sublinear training time and memory complexities. Our main contributions are (1) approximating nonlinear operations in GNNs using polynomial tensor sketch (PTS) and (2) updating sketches using learnable locality sensitive hashing (LSH). Our novel framework has the potential to be applied to other architectures and applications where the amount of data makes training even simple models impractical.

References

[1] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. *Advances in neural information processing systems*, 28, 2015.

[2] Haim Avron, Huy Nguyen, and David Woodruff. Subspace embeddings for the polynomial kernel. *Advances in neural information processing systems*, 27, 2014.

[3] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Spectral clustering with graph neural networks for graph pooling. In *International Conference on Machine Learning*, pages 874–883. PMLR, 2020.

[4] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. Scaling graph neural networks with approximate pagerank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2464–2473, 2020.

[5] Daniele Calandriello, Alessandro Lazaric, Ioannis Koutis, and Michal Valko. Improved large-scale graph learning through ridge spectral sparsification. In *International Conference on Machine Learning*, pages 688–697. PMLR, 2018.

[6] Sarath Chandar, Sungjin Ahn, Hugo Larochelle, Pascal Vincent, Gerald Tesauro, and Yoshua Bengio. Hierarchical memory networks. *arXiv preprint arXiv:1605.07427*, 2016.

[7] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[8] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[9] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. Mongoose: A learnable lsh framework for efficient neural network training. In *International Conference on Learning Representations*, 2020.

[10] Beidi Chen, Tharun Medini, James Farwell, Charlie Tai, Anshumali Shrivastava, et al. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *Proceedings of Machine Learning and Systems*, 2:291–306, 2020.

[11] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. In *International Conference on Machine Learning*, pages 942–950. PMLR, 2018.

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

[13] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International conference on machine learning*, pages 2285–2294. PMLR, 2015.

[14] 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 *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 257–266, 2019.

[15] Mucong Ding, Kezhi Kong, Jingling Li, Chen Zhu, John Dickerson, Furong Huang, and Tom Goldstein. Vq-gnn: A universal framework to scale up graph neural networks using vector quantization. *Advances in Neural Information Processing Systems*, 34, 2021.

[16] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The World Wide Web Conference*, pages 417–426, 2019.

[17] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning*, pages 3294–3304. PMLR, 2021.

[18] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[19] Insu Han, Haim Avron, and Jinwoo Shin. Polynomial tensor sketch for element-wise function of low-rank matrix. In *International Conference on Machine Learning*, pages 3984–3993. PMLR, 2020.

[20] 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. *Advances in neural information processing systems*, 33:22118–22133, 2020.

[21] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. Ogb-lsc: A large-scale challenge for machine learning on graphs. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[22] Zengfeng Huang, Shengzhong Zhang, Chong Xi, Tang Liu, and Min Zhou. Scaling up graph neural networks via graph coarsening. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 675–684, 2021.

[23] Wei Jin, Lingxiao Zhao, Shichang Zhang, Yozen Liu, Jiliang Tang, and Neil Shah. Graph condensation for graph neural networks. In *International Conference on Learning Representations*, 2022.

[24] William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary Mathematics*, 26(189-206):1, 1984.

[25] Shiva Prasad Kasiviswanathan, Nina Narodytska, and Hongxia Jin. Network approximation using tensor sketching. In *International Joint Conference on Artificial Intelligence*, pages 2319–2325, 2018.

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

[27] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2019.

[28] Yibo Lin, Zhao Song, and Lin F Yang. Towards a theoretical understanding of hashing-based neural nets. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 127–137. PMLR, 2019.

[29] Zirui Liu, Kaixiong Zhou, Fan Yang, Li Li, Rui Chen, and Xia Hu. Exact: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*, 2021.

[30] Andreas Loukas. Graph reduction with spectral and cut guarantees. *Journal of Machine Learning Research*, 20:1–42, 2019.

[31] Ninh Pham and Rasmus Pagh. Fast and scalable polynomial kernels via explicit feature maps. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 239–247, 2013.

[32] Emanuele Rossi, Fabrizio Frasca, Ben Chamberlain, Davide Eynard, Michael Bronstein, and Federico Monti. Sign: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198*, 2020.

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

[34] Yang Shi and Animashree Anandkumar. Higher-order count sketch: Dimensionality reduction that retains efficient tensor operations. In *2020 Data Compression Conference (DCC)*, pages 394–394. IEEE, 2020.

[35] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454, 2017.

[36] Ryan Spring, Anastasios Kyrillidis, Vijai Mohan, and Anshumali Shrivastava. Compressing gradient optimizers via count-sketches. In *International Conference on Machine Learning*, pages 5946–5955. PMLR, 2019.

[37] Rakshith S Srinivasa, Cao Xiao, Lucas Glass, Justin Romberg, and Jimeng Sun. Fast graph attention networks using effective resistance based graph sparsification. *arXiv preprint arXiv:2006.08796*, 2020.

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

[39] Yining Wang, Hsiao-Yu Tung, Alexander J Smola, and Anima Anandkumar. Fast and guaranteed tensor decomposition via sketching. *Advances in neural information processing systems*, 28, 2015.

[40] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 1113–1120, 2009.

[41] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, pages 6861–6871. PMLR, 2019.

[42] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)*, 2020.

[43] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2018.

[44] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2019.

[45] Cheng Zheng, Bo Zong, Wei Cheng, Dongjin Song, Jingchao Ni, Wenchao Yu, Haifeng Chen, and Wei Wang. Robust graph representation learning via neural sparsification. In *International Conference on Machine Learning*, pages 11458–11468. PMLR, 2020.

[46] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in Neural Information Processing Systems*, 32:11249–11259, 2019.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] See Section 1.
   (b) Did you describe the limitations of your work? [Yes] Currently, our work has two major limitations: (1) our theoretical assumptions and results may not perfectly correspond to the reality; see the theoretical remarks in Section 3, and (2) our implementation is not fully-optimized with the more advanced libraries; see the efficiency discussions in Section 5.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] We see our work as a theoretical and methodological contribution toward more resource-efficient graph representation learning. Our methodological advances may enable larger-scale network analysis for societal good. However, progress in graph embedding learning may potentially inspire other hostile social network studies, such as monitoring fine-grained user interactions.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] See Lemma 1 and Theorem 1.
   (b) Did you include complete proofs of all theoretical results? [Yes] See Appendix B.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See Appendix H.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix H.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See Section 5.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix H.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See Appendix H.
   (b) Did you mention the license of the assets? [Yes] See Appendix H.
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Supplementary Material for Sketch-GNN: Scalable Graph Neural Networks with Sublinear Training Complexity

A More Preliminaries

In this appendix, further preliminary information and relevant discussions are provided.

A.1 Common GNNs in the Unified Framework

Here we list the common GNNs that can be re-formulated into the unified framework, which is introduced in Section 2. The majority of GNNs can be interpreted as performing message passing on node features, followed by feature transformation and an activation function, a process known as "generalized graph convolution" (Eq. (1)). Within this common framework, different types of GNNs differ from each other by their choice of convolution matrices $C^{(q)}$, which can be either fixed or learnable. A learnable convolution matrix depends on the inputs and learnable parameters and can be different in each layer (thus denoted as $C^{(l,q)}$),

$$C_{i,j}^{(l,q)} = \underbrace{\mathfrak{C}_{i,j}^{(q)}}_{\text{fixed}} \cdot \underbrace{h_{\theta^{(l,q)}}^{(q)}(X_{i,:}^{(l)}, X_{j,:}^{(l)})}_{\text{learnable}}, \tag{8}$$

where $\mathfrak{C}^{(q)}$ denotes the fixed mask of the q-th learnable convolution, which may depend on the adjacency matrix A and input edge features $E_{i,j}$. While $h^{(q)}(\cdot,\cdot) : \mathbb{R}^{f_l} \times \mathbb{R}^{f_l} \to \mathbb{R}$ can be any learnable model parametrized by $\theta^{(l,q)}$. Sometimes a learnable convolution matrix may be further row-wise normalized as $C_{i,j}^{(l,q)} \leftarrow C_{i,j}^{(l,q)} / \sum_j C_{i,j}^{(l,q)}$, for example Graph Attention Network (GAT [38]). According to [15], we list some well-known GNN models that fall inside this framework in Table 4.

Table 4: Summary of GNNs re-formulated as generalized graph convolution [15].

Model Name	Design Idea	Conv. Matrix Type	# of Conv.	Convolution Matrix
GCN[1] [26]	Spatial Conv.	Fixed	1	$C = \widetilde{D}^{-1/2}\widetilde{A}\widetilde{D}^{-1/2}$
GIN[1] [43]	WL-Test	Fixed + Learnable	2	$\begin{cases} C^{(1)} = A \\ \mathfrak{C}^{(2)} = I_n \text{ and } h_{\epsilon^{(l)}}^{(2)} = 1 + \epsilon^{(l)} \end{cases}$
SAGE[2] [18]	Message Passing	Fixed	2	$\begin{cases} C^{(1)} = I_n \\ C^{(2)} = D^{-1}A \end{cases}$
GAT[3] [38]	Self-Attention	Learnable	# of heads	$\begin{cases} \mathfrak{C}^{(q)} = A + I_n \text{ and} \\ h_{\boldsymbol{a}^{(l,q)}}^{(q)}(X_{i,:}^{(l)}, X_{j,:}^{(l)}) = \exp\big(\text{LeakyReLU}(\\ \quad (X_{i,:}^{(l)}W^{(l,q)} \parallel X_{j,:}^{(l)}W^{(l,q)}) \cdot \boldsymbol{a}^{(l,q)})\big) \end{cases}$

[1] Where $\widetilde{A} = A + I_n$, $\widetilde{D} = D + I_n$. [2] $C^{(2)}$ represents mean aggregator. Weight matrix in [18] is $W^{(l)} = W^{(l,1)} \parallel W^{(l,2)}$.
[3] Need row-wise normalization. $C_{i,j}^{(l,q)}$ is non-zero if and only if $A_{i,j} = 1$, thus GAT follows direct-neighbor aggregation.

A.2 Definition of Locality Sensitivity Hashing

The definitions of count sketch and tensor sketch are based on the hash table(s) that merely require data-independent uniformity, i.e., a high likelihood that the hash-buckets are of comparable size. In contrast, locality sensitive hashing (LSH) is a hashing scheme with a locality-sensitive hash function $H : \mathbb{R}^d \to [c]$ that assures close vectors are hashed into the same bucket with a high probability while distant ones are not. Consider a locality-sensitive hash function $H : \mathbb{R}^d \to [c]$ that maps vectors in $\mathbb{R}^d$ to the buckets $\{1, \ldots, c\}$. A family of LSH functions $\mathcal{H}$ is (D, tD, p_1, p_2)-sensitive if and only if for any $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$ and any H selected uniformly at random from $\mathcal{H}$, it satisfies

$$\begin{aligned} &\text{if} \quad \mathsf{Sim}(\boldsymbol{u}, \boldsymbol{v}) \geq D \quad \text{then} \quad \mathbb{P}[H(\boldsymbol{u}) = H(\boldsymbol{v})] \geq p_1, \\ &\text{if} \quad \mathsf{Sim}(\boldsymbol{u}, \boldsymbol{v}) \leq tD \quad \text{then} \quad \mathbb{P}[H(\boldsymbol{u}) = H(\boldsymbol{v})] \leq p_2, \end{aligned} \tag{9}$$

where $\mathsf{Sim}(\cdot,\cdot)$ is a similarity metric defined on $\mathbb{R}^d$.

B Polynomial Tensor Sketch and Error Bounds

In this appendix, we provide additional theoretical details regarding the concentration guarantees of sketching the linear part in each GNN layer (Lemma 1), and the proof of our multi-layer error bound (Theorem 1).

B.1 Error Bound for Sketching Linear Products

Here, we discuss the problem of approximating the linear product $CX^{(l)}W^{(l)}$ using count/tensor sketch. Since we rely on count/tensor sketch to compress the individual components C and $X^{(l)}W^{(l)}$ of the intermediate product $CX^{(l)}W^{(l)}$ before we sketch the nonlinear activation, it is useful to know how closely sketching approximates the product. We have the following result:

Lemma 1. *Given matrices $C \in \mathbb{R}^{n\times n}$ and $(X^{(l)}W^{(l)})^{\mathsf{T}} \in \mathbb{R}^{d\times n}$, consider a randomly selected cuont sketch matrix $R \in \mathbb{R}^{c\times n}$ (defined in Section 2), where c is the sketch dimension, and it is formed using $r = \sqrt[j]{n}$ underlying hash functions drawn from a 3-wise independent hash family $\mathcal{H}$ for some $j \geq 1$. If $c \geq (2+3^j)/(\varepsilon^2\delta)$, we have*

$$\Pr\left(\left\|(CR_k^{\mathsf{T}})(R_kX^{(l)}W^{(l)}) - CX^{(l)}W^{(l)}\right\|_F^2 > \varepsilon^2\|C\|_F^2\|X^{(l)}W^{(l)}\|_F^2\right) \leq \delta. \tag{10}$$

Proof. The proof follows immediately from the Theorem 1 of [2]. □

For $j \geq 1$ fulfilling $c \geq (2+3^j)/(\varepsilon^2\delta)$, we have $j = O(\log_3 c)$, and consequently $r = (n)^{1/j} = \Omega(3^{\log_c n})$. In practice, when n is not too small, $\log_c n \approx 1$ since c grows sublinearly with respect to n. In this sense, the dependence of r on n is negligible.

B.2 Proof of Error Bound for Final-Layer Representation (Theorem 1).

Proof. For fixed degree r of a polynomial tensor sketch (PTS), by the Theorem 5 of [19], for $\Gamma^{(1)} = \max\left\{5\|X^{(l)}W^{(l)}\|_F^2, (2+3^r)\sum_i(\sum_j[X^{(l)}W^{(l)}]_{i,j})^r\right\}$, it holds that

$$\mathbf{E}(\|\sigma(CX^{(l)}W^{(l)}) - \widetilde{X}^{(l+1)}\|_F^2) \leq \left(\frac{2}{1+\frac{c\lambda^{(l)2}}{nr\Gamma^{(l)}}}\right)\|\sigma(CX^{(l)}W^{(l)})\|_F^2, \tag{11}$$

where $\lambda^{(1)} \geq 0$ is the smallest singular value of the matrix $Z \in \mathbb{R}^{nd\times r}$, each column, $Z_{:,k}$, being the vectorization of $(CX^{(1)}W^{(1)})^{\odot k}$. This is the error bound for sketching a single layer, including the non-linear activation units.

Consider starting from the first layer ($l = 1$), for simplicity, let us denote the upper bound when $l = 1$ as E_1. The error in the second layer ($l - 2$), including the propagated error from the first layer E_1, is expressible as $\|\sigma(CX^{(2)}W^{(2)}) - \widetilde{X}^{(3)} + E_1\|_F^2$, which by sub-multiplicativity and the inequality $(a+b)^2 \leq 2a^2 + 2b^2$ gives

$$\|\sigma(CX^{(2)}W^{(2)}) - \widetilde{X}^{(3)} + E_1\|_F^2 \leq 2\|\sigma(CX^{(2)}W^{(2)}) - \widetilde{X}^{(3)}\|_F^2 + 2||E_1||. \tag{12}$$

By repeatedly invoking the update rule/recurrence in Eq. (1) and the Theorem 5 in [19] up to the final layer $l = L$, we obtain the overall upper bound on the total error as claimed. □

C Learnable Sketches and LSH

C.1 Learning of the Polynomial Tensor Sketch Coefficients.

We propose to learn the coefficients $\{c_k\}_{k=1}^r$ using gradient descent with an L_2 regularization, $\lambda\sum_{k=1}^r c_k^2$. For a node classification task, the coefficients in all layers are directly optimized to minimize the classification loss. Experimentally, the coefficients that obtain the best classification accuracy do not necessarily correspond to a known activation.

For the proof of concept experiment (Fig. 1a in Section 5), the coefficients $\{c_k\}_{k=1}^r$ in the first layer are learned to approximate the sigmoid activated hidden embeddings $\sigma(CX^{(1)}W^{(1)})$. The relative errors are evaluated relative to the "sigmoid activated ground-truth". We find in our experiments that the relative errors are comparable to the coreset-based approach.

C.2 Change the Hash Table of Count Sketches

Here we provide more information regarding the solution to the challenge (2) in Section 3.3. Since the hash tables utilized by each layer is different, we have to change the underlying hash table of the sketched representations when propagating through Sketch-GNN.

Consider the Sketch-GNN forward-pass described by Eq. (5), while the count sketch functions are now different in each layer. We denote the k'-th count sketch function in the l-th layer by $\mathsf{CS}^{(l,k')}(\cdot)$ (adding the superscript (l)), and denote its underlying hash table by $h^{(l,k)}$. Since the hash table used to count sketch $S_C^{(l,k,k')}$ is $h^{(l,k')}$, what we obtain using Eq. (5) is $\mathsf{CS}^{(l,k')}((X^{(l+1)})^\mathsf{T})$. However, we actually need $S_X^{(l+1,k')} = \mathsf{CS}^{(l+1,k')}((X^{(l+1)})^\mathsf{T})$ as the input to the subsequent layer.

By definition, we can change the underlying hash table like $S_X^{(l+1,k')} = \mathsf{CS}^{(l+1,k')}((X^{(l+1)})^\mathsf{T}) = \mathsf{CS}^{(l,k')}((X^{(l+1)})^\mathsf{T})R^{(l,k')}(R^{(l+1,k')})^\mathsf{T}$, where $R^{(l,k')}$ is the count sketch matrix of $\mathsf{CS}^{(l,k')}(\cdot)$. In fact, we only need to right multiply a $c \times c$ matrix $T^{(l,k')} := R^{(l,k')}(R^{(l+1,k')})^\mathsf{T}$, which is $O(c^2)$ and can be efficiently computed by

$$[T^{(l,k')}]_{i,j} = \sum_{a=1}^{n} s_a^{(l+1,k')} s_a^{(l,k')} \mathbb{1}\{h^{(l,k')}(a) = i\} \mathbb{1}\{h^{(l+1,k')}(a) = j\}. \tag{13}$$

We can maintain this $c \times c$ matrix $T^{(l,k')}$ as a signature of both hash tables $h^{(l,k')}$ and $h^{(l+1,k')}$. We are able to update $T^{(l,k')}$ efficiently when we update the hash tables on a subset B of entries (see Section 3.3). We can also compute the sizes of buckets for both hash functions from $T^{(l,k')}$, which is useful to sketch the attention units in GAT; see Appendix D.

C.3 Sparse Forward-Pass and Back-Propagation for Loss Evaluation

Here we provide more details on using the sparse forward-pass and back-propagation technique in [10] to avoid $O(n)$ complexity in loss evaluation. For a node classification task, we construct an LSH hash table for each class, which indexes all the labeled nodes in the training split that belong to this class. These LSH hash tables can be used to select the nodes with bad predictions in constant time, i.e., nodes whose predicted class scores have a small inner product with respect to the ground truth (one-hot encoded) label. Consequently, we only evaluate the loss on the selected nodes, avoiding the $O(n)$ complexity. The LSH hash tables are updated using the same method described in challenge (1) in Section 3.3.

D Generalize to More GNNs

This appendix briefly describes how to generalizing Sketch-GNN from GCN to some other GNN architectures, including GraphSAGE [18] and GAT [38].

D.1 Sketch-GraphSAGE: Sketching Multiple Fixed Convolutions

The update rule (Eq. (5)) of Sketch-GNN can be directly applied to GNNs with only one fixed convolution matrix, such as GCN by setting $C = \widetilde{D}^{-1/2}\widetilde{A}\widetilde{D}^{-1/2}$. Here we seek to generalize Sketch-GNN to GNNs with multiple fixed convolutions, for example, GraphSAGE with $C^{(1)} = I_n$ and $C^{(2)} = D^{-1}A$. This can be accomplished by rewriting the update rule of GraphSAGE $X^{(l+1)} = \sigma(X^{(l)}W^{(l,1)} + D^{-1}AX^{(l)}W^{(l,2)})$ as a form resembling $\sigma(UV^\mathsf{T})$, so that the polynomial tensor sketch technique may still be used.

Therefore, we replace the update rule (Eq. (5)) with the following for GraphSAGE,

$$\begin{aligned}
&\sigma(X^{(l)}W^{(l,1)} + D^{-1}AX^{(l)}W^{(l,2)}) = \sigma\Big(\Big[I_n \parallel (D^{-1}A)^\mathsf{T}\Big]^\mathsf{T}\Big[X^{(l)}W^{(l,1)} \parallel X^{(l)}W^{(l,2)}\Big]\Big) \\
&\approx \sum_{k=1}^{r} c_k \mathsf{TS}_k\Big(\Big[I_n \| (D^{-1}A)^\mathsf{T}\Big]^\mathsf{T}\Big)\mathsf{TS}_k\Big(\Big[X^{(l)}W^{(l,1)} \| X^{(l)}W^{(l,2)}\Big]^\mathsf{T}\Big)^\mathsf{T}.
\end{aligned} \tag{14}$$

D.2 Sketch-GAT: Sketching Self-Attention Units

GAT employs self-attention to learn the convolution matrix $C^{(l)}$ (superscript (l) denotes the convolution matrices learned are different in each layer). For the sake of simplicity, we assume single-headed attention while we can generalize to multiple heads using the same method as for GraphSAGE. The convolution matrix of GAT is defined as $C^{(l)} = (A+I_n) \odot ((\exp^{\odot}(Z^{(l)})\mathbf{1}_n)^{\mathsf{T}})^{-1} \exp^{\odot}(Z^{(l)})$, where $\mathbf{1}_n \in \mathbb{R}^n$ is a vector of ones, $Z^{(l)} \in \mathbb{R}^{n \times n}$ is the raw attention scores in the l-th layer, defined as $Z^{(l)}_{i,j} = \mathrm{LeakyReLU}([X^{(l)}_{i,:}W^{(l)} \parallel X^{(l)}_{j,:}W^{(l)}] \cdot \boldsymbol{a}^{(l)})$, with $\boldsymbol{a}^{(l)} \in \mathbb{R}^{2n}$ being the learnable parameter vector.

Our goal is to approximate the sketches of the convolution matrix $S_C^{(l,k,k')}$ using the sketches of node representations $S_X^{(l,k)}$ and the learnable weights $W^{(l)}, \boldsymbol{a}^{(l)}$. We accomplish this by utilizing the locality-sensitive property of the sketches and by assuming that the random Rademacher variables $s^{(l,1)}, \cdots, s^{(l,k)}$ are fixed to $+1$. We find that setting all Rademacher variables to $+1$ has no discernible effect on the performance of Sketch-GAT.

With this additional assumption, each vector of node representation can be approximated by the average of vectors hashed into the same bucket, i.e., $X^{(l)}_{i,:} \approx \sum_j \mathbb{1}\{h^{(l,k)}(i) = h^{(l,k)}(j)\}X^{(l)}_{j,:} / \sum_j \mathbb{1}\{h^{(l,k)}(i) = h^{(l,k)}(j)\}$ for any $k \in [r]$. More specifically, the numerator is exactly the $h^{(l,k)}(i)$-th column vector of the sketch $S_X^{(l,k)}$, i.e., $\sum_j \mathbb{1}\{h^{(l,k)}(i) = h^{(l,k)}(j)\}X^{(l)}_{j,:} = [S_X^{(l,k)}]_{:,h^{(l,k)}(i)}$. Using only the sketch $S_X^{(l,k)}$ and the bucket sizes in the hash table $h^{(l,k)}$, we can approximate any $X^{(l)}_{i,:}$ as a function of $h^{(l,k)}(i)$ (instead of i), and thus approximate the entries of this $n \times n$ matrix $Z^{(l)}$ with c^2 distinct values only. Even after the element-wise exponential and row-wise normalization, any attention score $[((\exp^{\odot}(Z^{(l)})\mathbf{1}_n)^{\mathsf{T}})^{-1} \exp^{\odot}(Z^{(l)})]_{i,j}$ can still be estimated as a function of the tuple $(h^{(l,k)}(i), h^{(l,k)}(j))$, where $Z^{(l)}_{i,j} = \langle X^{(l)}_{i,:}, X^{(l)}_{j,:} \rangle$. This means we can approximate the attention scores $[((\exp^{\odot}(Z^{(l)})\mathbf{1}_n)^{\mathsf{T}})^{-1} \exp^{\odot}(Z^{(l)})]$ using the sketched representation $S_X^{(l,k)}$, using the fact that $Z^{(l)}_{i,j} = \langle X^{(l)}_{i,:}, X^{(l)}_{j,:} \rangle \approx \langle [S^{(l)}_X]_{:,h^{(l)}(i)} / |\{a|h^{(l)}(a) = h^{(l)}(i)\}| [S^{(l)}_X]_{:,h^{(l)}(j)} / |\{a|h^{(l)}(a) = h^{(l)}(j)\}| \rangle$, where $\{a|h^{(l)}(a) = h^{(l)}(i)\}|$ is the bucket size of $h^{(l)}(i)$-th hash bucket.

We can see that computing the sketches of $C^{(l)}$ (the sketch functions are defined by the same hash table $h^{(l,k)}(\cdot)$) only requires **(1)** the c^2 distinct estimations of the entries in $((\exp^{\odot}(Z^{(l)})\mathbf{1}_n)^{\mathsf{T}})^{-1} \exp^{\odot}(Z^{(l)})$, and **(2)** an "averaged $c \times c$ version" of the mask $(A + I_n)$, which is exactly the two-sided count sketch of $(A + I_n)$ defined by the hash table $h^{(i,j)}$. In conclusion, we find a $O(c^2)$ algorithm to estimate the sketches of the convolution matrix $S_C^{(l,k,k')}$ using the sketches of node representations $S_X^{(l,k)}$ and a pre-computed two-sided count sketch of the mask matrix $(A + I_n)$.

E The Complete Pseudo-Code

The following is the pseudo-code outlining the workflow of Sketch-GNN (assuming GCN backbone).

F Summary of Theoretical Complexities

In this appendix, we provide more details on the theoretical complexities of Sketch-GNN with a GCN backbone. For simplicity, we assume bounded maximum node degree, i.e., $m = \theta(n)$.

Preprocessing. The r sketches of the node feature matrix take $O(r(n+c)d)$ time and occupy $O(rdc)$ memory. And the r^2 sketches of the convolution matrix require $O(r(m + c) + r^2m)$ time (the LSH hash tables are determined by the node feature vectors already) and $O(r^2cm/n)$ memory. The total preprocessing time is $O(r^2m + rm + r(n + c)d) = O(n)$ and the memory taken by the sketches is $O(rc(d + rm/n)) = O(c)$.

Algorithm 1 *Sketch-GNN*: sketch-based approximate training of GNNs with sublinear complexities

Require: GNN's convolution matrix C, input node features X, ground-truth labels Y

procedure PREPROCESS(C, X)
  Sketch $X = X^{(0)}$ into $\{S_X^{(0,k)}\}_{k=1}^r$ and sketch C into $\{\{S_C^{(k,k')}\}_{k=1}^r\}_{k'=1}^r$
procedure TRAIN($\{\{S_C^{(k,k')}\}_{k=1}^r\}_{k'=1}^r, \{S_X^{(0,k)}\}_{k=1}^r, Y$)
  Initialize weights $\{W^{(l)}\}_{l=1}^L$, coefficients $\{\{c_k^{(l)}\}_{k=1}^r\}_{l=1}^L$, and LSH projections $\{\{P_k^{(l)}\}_{k=1}^r\}_{l=1}^L$.
  for epoch $t = 1, \ldots, T$ **do**
    for layer $l = 1, \ldots, L-1$ **do**
      Forward-pass and compute $S_X^{(l+1,k')}$ via Eq. (5).
    Evaluate losses on a subset B of nodes in buckets with the largest gradients for each class.
    Back-propagate and update weights $\{W^{(l)}\}_{l=1}^L$ and coefficients $\{\{c_k^{(l)}\}_{k=1}^r\}_{l=1}^L$.
    Update the LSH projections $\{\{P_k^{(l)}\}_{k=1}^r\}_{l=1}^L$ with the triplet loss Eq. (7) for every T_{LSH} epoch.
  return Learned weights $\{W^{(l)}\}_{l=1}^L$
procedure INFERENCE($\{W^{(l)}\}_{l=1}^L$)
  Predict via the corresponding standard GNN update rule, using the learned weights $\{W^{(l)}\}_{l=1}^L$

Forward and backward passes. For each sketch in each layer, matrix multiplications take $O(cd(d + m/n))$ time, FFT and its inverse take $O(dc\log(c))$ time, thus the total forward/backward pass time is $O(Lcrd(\log(c) + d + m/n)) = O(c)$. The memory taken by sketches in a Sketch-GNN is just L times the memory of input sketches, i.e., $O(Lrc(d + rm/n)) = O(c)$.

LSH hash updates and loss evaluation. Computing the triplet loss and updating the corresponding part of the hash table requires $O(Lrb(n/c))$ where $b = |B|$ is the number of nodes selected based on the gradients (for each sketch). Updates of the sketches are only performed every T_{LSH} epochs.

Inference is conducted on the standard GCN model with parameters $\{W^{(l)}\}_{l=1}^L$ learned via Sketch-GNN, which takes $O(Ld(m/n + d))$ time on average for a node sample.

Remarks. (1) Sparsity of sketched convolution matrix. The two-sided sketch $\mathsf{CS}(\mathsf{CS}(C)^\mathsf{T}) \in \mathbb{R}^{c\times c}$ maintains sparsity for sparse convolution C, as $\mathsf{CS}(\mathsf{CS}(C)^\mathsf{T}) = RCR^\mathsf{T}$ (a product of 3 sparse matrices) is still sparse, where count-sketch matrix $R \in \mathbb{R}^{c\times n}$ has one non-zero entry per column (by its definition see Section 2). If C has at most s non-zeros per column, there are $\leq s$ non-zeros per column in RC when $c \gg s$ (holds for sparse graphs that real-world data exhibits). Thus, we avoid the $O(c^2)$ memory cost and are strictly $O(c)$. **(2) Overhead of computing the LSH hash tables.** Following Eq. (3) and Eq. (6), we need $O(cd)$ overhead to obtain the LSH hash index of each node, and since we have n nodes in total and we maintain r independent hash tables, the total overhead for computing the LSH hash tables is $O(ncrd)$ during preprocessing.

In conclusion, we achieve sublinear training complexity except for the one-time preprocessing step.

G More Related Work Discussions

G.1 Sketch-GNN v.s. GraphSAINT

GraphSAINT[44] is a graph sampling method that enables training on a mini-batch of subgraphs instead of on the large input graph. GraphSAINT is easily applicable to any graph neural network (GNN), introduces minor overheads, and usually works well in practice. However, GraphSAINT is not a sub-linear training algorithm, it saves memory at the cost of time overhead. We have to iterate through the full batch of subgraphs in an epoch, and the training time complexity is still linear in the graph size. In contrast, our proposed Sketch-GNN is an approximated training algorithm of some GNNs with sub-linear time and memory complexities. Sketch-GNN has the potential to scale better than GraphSAINT on larger graphs. Besides, as a sketching algorithm, Sketch-GNN is suitable for some scenarios, for example, sketching big graphs in an online/streaming fashion. Sketch-GNN can also be combined with subgraph sampling to scale up to extremely large graphs. Sketching the sampled subgraphs (instead of the original graph) avoids the decreasing sketch-ratio when the input graph size grows to extremely large while with a fixed memory constraint.

G.2 Sketching in GNNs

Liu et al. [29] (EXACT) is a recent work which applies random projection to reduce the memory footprint of non-linear activations in GNNs. In this regard, they also applies sketching techniques to scale up the training of GNNs. However there are three important differences between Sketch-GNN and EXACT summarized as follows: (1) Sketch-GNN propagates sketched representations while sketching in EXACT only affects the back-propagation, (2) Sketch-GNN sketches the graph size dimension while EXACT sketches the feature dimension, and (3) Sketch-GNN enjoys sub-linear complexity while EXACT does not. We want to address that Sketch-GNN and EXACT are aiming for very different goals; Sketch-GNN is sketching the graph to achieve sub-linear complexity, while EXACT is sketching to save the memory footprint of non-linear activations

G.3 Sketching Neural Networks

Compression of layers/kernels via sketching methods has been discussed previously, but not on a full-architectural scale. Wang et al. [39] utilize a multi-dimensional count sketch to accelerate the decomposition of a tensorial kernel, at which point the tensor is fully-restored, which is not possible in our memory-limited scenario. Shi and Anandkumar [34] utilize the method of Wang et al. [39] to compute compressed tensorial operations, such as contractions and convolutions, which is more applicable to our setup. Their experiments involve the replacement of a fully-connected layer at the end of a tensor regression network rather than full architectural compression. Furthermore, they guarantee the recovery of a sketched tensor rather than the recovery of tensors passing through a nonlinearity such as a ReLU. Kasiviswanathan et al. [25] propose layer-to-layer compression via sign sketches, albeit with no guarantees, and their back-propagation equations require $O(n^2)$ memory complexity when dealing with the nonlinear activations. In contrast to these prior works, we propose a sketching method for nonlinear activation units, which avoids the need to unsketch back to the high dimensional representation in each layer.

G.4 LSH in Neural Networks

Locality sensitive hashing (LSH) has been widely adopted to address the time and memory bottlenecks of many large-scale neural networks training systems, with applications in computer vision [13], natural language processing [6] and recommender systems [35]. For fully connected neural networks, Chen et al. [10] proposes an algorithm, SLIDE, that retrieves the neurons in each layer with the maximum inner product during the forward pass using an LSH-based data structure. In SLIDE, gradients are only computed for neurons with estimated large gradients during back-propagation. For transformers, Kitaev et al. [27] proposes to mitigate the memory bottleneck of self-attention layers over long sequences using LSH. More recently, Chen et al. [9] has dealt with the update overheads of LSH during the training of NNs. Chen et al. [9] introduces a scheduling algorithm to adaptively perform LSH updates with provable guarantees and a learnable LSH algorithm to improve the query efficiency.

G.5 Graph Sparsification for GNNs

Graph sparsification, i.e., removing task-irrelevant and redundant edges from the large input graph, can be applied to speed up the training of GNNs. Calandriello et al. [5] propose fast and scalable graph sparsification algorithms for graph-Laplacian-based learning on large graphs. Zheng et al. [45] sparsify the graph using neural networks and applied to the training of general GNNs. Srinivasa et al. [37] specifically considered the graph sparsification problem for graph attention (e.g., graph attention networks, GAT). Graph sparsification will not reduce the number of nodes; thus, the memory reduction of node feature representation is limited. However, some carefully designed graph sparsification may enjoy small approximation error (thus smaller performance drops) and improve the robustness of learned models.

H Implementation Details

This appendix lists the implementation details and hyper-parameter setups for the experiments in Section 5.

Datasets. Dataset *ogbn-arxiv* and *ogbn-product* are obtained from the Open Graph Benchmark (OGB)[1]. Dataset *Reddit* is adopted from [44] and downloaded from the PyTorch Geometric library[2], it is a sparser version of the original dataset provided by Hamilton et al. [18]. We conform to the standard data splits defined by OGB or PyTorch Geometric.

Code Frameworks. The implementation of our **Sketch-GNN** is based on the PyTorch library and the PyTorch Sparse library[3]. More specifically, we implement the Fast Fourier Transform (FFT) and its inverse (used in tensor sketch) using PyTorch. We implement count sketch of node features and convolution matrices as sparse-dense or sparse-sparse matrix multiplications, respectively, using PyTorch Sparse. Our implementations of the standard GNNs are based on the PyTorch Geometric library. The implementations of SGC [41] and GraphSAINT [44] are also adopted from PyTorch Geometric, while the implementations of VQ-GNN[4] [15] and Coarsening[5] [22] are adopted from their official repositories, respectively. All of the above-mentioned libraries (except for PyTorch) and code repositories we used are licensed under the MIT license.

Computational Infrastructures. All of the experiments are conducted on Nvidia RTX 2080Ti GPUs with Xeon CPUs.

Repeated Experiments. For the efficiency measures in Section 5, the experiments are repeated two times to check the self-consistency. For the performance measures in Section 5, we run all the experiments five times and report the mean and variance.

Setups of GNNs and Training. On all of the three datasets, unless otherwise specified, we always train 3-layer GNNs with hidden dimensions set to 128 for all scalable methods and for the oracle "full-graph" baseline. The default learning rate is 0.001. We apply batch normalization on *ogbn-arxiv* but not the other two datasets. Dropout is never used. Adam is used as the default optimization algorithm.

Setups of Baseline Methods. For SGC, we set the number of propagation steps k in preprocessing to 3 to be comparable to other 3-layer GNNs. For GraphSAINT, we use the GraphSAINT-RW variant with a random walk length of 3. For VQ-GNN, we set the number of K-means clusters to 256 and use a random walk sampler (walk length is also 3). For Coarsening, we use the Variation Neighborhood graph coarsening method if not otherwise specified. As reported in [22], this coarsening algorithm has the best performance. We use the mean aggregator in GraphSAGE and single-head attention in GAT.

Setups of Sketch-GNN. If not otherwise mentioned, we always set the polynomial order (i.e., the number of sketches) $r = 3$. An L_2 penalty on the learnable coefficients is applied with coefficient λ ranging from 0.01 to 0.1. For the computation of the triplet loss, we always set α to 0.1, but the values of $t_+ > t_- > 0$ are different across datasets. We can find a suitable starting point to tune by finding the smallest inner product of vectors hashed into the same bucket. To get the sampled subset B, we take the union of $0.01c$ buckets with the largest gradient norms for each sketch. The LSH hash functions are updated every time for the first 5 epochs, and then only every $T_{\text{LSH}} = 10$ epochs. We do not traverse through all pairs of vectors in B to populate $\mathcal{P}_+$ and $\mathcal{P}_-$. Instead, we randomly sample pairs until $|\mathcal{P}_+|, |\mathcal{P}_-| > 1000$.

[1] https://ogb.stanford.edu/
[2] https://github.com/pyg-team/pytorch_geometric
[3] https://github.com/rusty1s/pytorch_sparse
[4] https://github.com/devnkong/VQ-GNN
[5] https://github.com/szzhang17/Scaling-Up-Graph-Neural-Networks-Via-Graph-Coarsening