GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings

Matthias Fey [1] **Jan Eric Lenssen** [1] **Frank Weichert** [1] **Jure Leskovec** [2]

Abstract

We present *GNNAutoScale* (GAS), a framework for scaling arbitrary message-passing GNNs to large graphs. GAS prunes entire sub-trees of the computation graph by utilizing historical embeddings from prior training iterations, leading to constant GPU memory consumption in respect to input node size without dropping any data. While existing solutions weaken the expressive power of message passing due to sub-sampling of edges or non-trainable propagations, our approach is provably able to maintain the expressive power of the original GNN. We achieve this by providing approximation error bounds of historical embeddings and show how to tighten them in practice. Empirically, we show that the practical realization of our framework, *PyGAS*, an easy-to-use extension for PyTorch Geometric, is both fast and memory-efficient, learns expressive node representations, closely resembles the performance of their non-scaling counterparts, and reaches state-of-the-art performance on large-scale graphs.

1. Introduction

Graph Neural Networks (GNNs) capture local graph structure and feature information in a trainable fashion to derive powerful node representations suitable for a given task at hand (Hamilton, 2020; Ma & Tang, 2020). As such, numerous GNNs have been proposed in the past that integrate ideas such as maximal expressiveness (Xu et al., 2019), anisotropy and attention (Veličković et al., 2018), non-linearities (Wang et al., 2019), or multiple aggregations (Corso et al., 2020) into their message passing formulation. However, one of the challenges that have so far precluded their wide adoption in industrial and social applications is the difficulty to scale them to large graphs (Frasca et al., 2020).

While the full-gradient in a GNN is straightforward to compute, assuming one has access to *all* hidden node embeddings in *all* layers, this is not feasible in large-scale graphs due to GPU memory limitations (Ma & Tang, 2020). Therefore, it is desirable to approximate its full-batch gradient stochastically by considering only a mini-batch $\mathcal{B} \subseteq \mathcal{V}$ of nodes for loss computation. However, this stochastic gradient is still expensive to obtain due to the exponentially increasing dependency of nodes over layers; a phenomenon framed as *neighbor explosion* (Hamilton et al., 2017). Due to neighbor explosion and since the whole computation graph needs to be stored on the GPU, deeper architectures can not be applied to large graphs. Therefore, a scalable solution needs to make the memory consumption constant or sub-linear in respect to the number of input nodes.

Recent works aim to alleviate this problem by proposing various sampling techniques based on the concept of dropping edges (Ma & Tang, 2020; Rong et al., 2020): *Node-wise sampling* (Hamilton et al., 2017; Chen et al., 2018b; Markowitz et al., 2021) recursively samples a fixed number of 1-hop neighbors; *Layer-wise sampling* techniques independently sample nodes for each layer, leading to a constant sample size in each layer (Chen et al., 2018a; Zou et al., 2019; Huang et al., 2018); In *subgraph sampling* (Chiang et al., 2019; Zeng et al., 2020b;a), a full GNN is run on an entire subgraph $\mathcal{G}[\mathcal{B}]$ induced by a sampled batch of nodes $\mathcal{B} \subseteq \mathcal{V}$. These techniques get rid of the neighbor explosion problem by sampling the graph but may fail to preserve the edges that present a meaningful topological structure. Further, existing approaches are either still restricted to shallow networks, non-exchangeable GNN operators or operators with reduced expressiveness. In particular, they consider only specific GNN operators and it is an open question whether these techniques can be successfully applied to the wide range of GNN architectures available (Veličković et al., 2018; Xu et al., 2019; Corso et al., 2020; Chen et al., 2020b). Another line of work is based on the idea of decoupling propagations from predictions, either as a pre- (Wu et al., 2019; Klicpera et al., 2019a; Frasca et al., 2020; Yu et al., 2020) or post-processing step (Huang et al., 2021). While this scheme enjoys fast training and inference time, it cannot be applied to any GNN, in particular because the propagation is non-trainable, and therefore reduces model

[1]Department of Computer Science, TU Dortmund University [2]Department of Computer Science, Stanford University. Correspondence to: Matthias Fey <matthias.fey@udo.edu>.

Proceedings of the 38th *International Conference on Machine Learning*, PMLR 139, 2021. Copyright 2021 by the author(s).

expressiveness. A different scalability technique is based on the idea of training each GNN layer in isolation (You et al., 2020). While this scheme resolves the neighbor explosion problem and accounts for all edges, it cannot infer complex interactions across consecutive layers.

Here, we propose the *GNNAutoScale* (GAS) framework that disentangles the scalability aspect of GNNs from their underlying message passing implementation. GAS revisits and generalizes the idea of *historical* embeddings (Chen et al., 2018b), which are defined as node embeddings acquired in previous iterations of training, *cf.* Figure 1. For a given mini-batch of nodes, GAS prunes the GNN computation graph so that only nodes inside the current mini-batch and their direct 1-hop neighbors are retained, *independent* of GNN depth. Historical embeddings act as an offline storage and are used to accurately fill in the inter-dependency information of out-of-mini-batch nodes, *cf.* Figure 1c. Through constant memory consumption in respect to input node size, GAS is able to scale the training of GNNs to large graphs, while still accounting for *all* available neighborhood information.

We show that approximation errors induced by historical information are solely caused by the staleness of the history and the Lipschitz continuity of the learned function, and propose solutions for tightening the proven bounds in practice. Furthermore, we connect scalability with expressiveness and theoretically show under which conditions historical embeddings allow to learn expressive node representations on large graphs. As a result, GAS is the first scalable solution that is able to keep the existing expressivity properties of the used GNN, which exist for a wide range of models (Xu et al., 2019; Morris et al., 2019; Corso et al., 2020).

We implement our framework practically as *PyGAS*[1], an extension for the PyTorch Geometric library (Fey & Lenssen, 2019), which makes it easy to convert common and custom GNN models into their scalable variants and to apply them to large-scale graphs. Experiments show that GNNs utilizing GAS achieve the same performances as their (non-scalable) full-batch equivalents (while requiring orders of magnitude less GPU memory), and are able to learn expressive node representations. Furthermore, GAS allows the application of expressive and hard-to-scale-up models on large graphs, leading to state-of-the-art results on several large-scale graph benchmark datasets.

2. Scalable GNNs via Historical Embeddings

Background. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ or $\boldsymbol{A} \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ denote a *graph* with node feature vectors $\boldsymbol{x}_v$ for all $v \in \mathcal{V}$. In this work, we are mostly interested in the task of *node classification*, where each node $v \in \mathcal{V}$ is associated with a label y_v, and the goal is to learn a representation $\boldsymbol{h}_v$ from which y_v can be easily predicted. To derive such a representation, GNNs follow a *neural message passing scheme* (Gilmer et al., 2017). Formally, the $(\ell+1)$-th layer of a GNN is defined as (omitting edge features for simplicity)

$$\begin{aligned}\boldsymbol{h}_v^{(\ell+1)} &= \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_v^{(\ell)}, \left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} \right\}\!\!\right\}_{w \in \mathcal{N}(v)}\right) \\ &= \textsc{Update}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_v^{(\ell)}, \bigoplus_{w \in \mathcal{N}(v)} \textsc{Msg}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_w^{(\ell)}, \boldsymbol{h}_v^{(\ell)}\right)\right)\end{aligned} \tag{1}$$

where $\boldsymbol{h}_v^{(\ell)}$ represents the embedding of node v obtained in layer ℓ and $\mathcal{N}(v)$ defines the neighborhood set of v. We initialize $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$. Here, $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}$ operates on *multisets* $\{\!\{\ldots\}\!\}$ and can be decomposed into differentiable $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ functions parametrized by weights $\boldsymbol{\theta}$, as well as permutation-invariant aggregation functions $\bigoplus$, *e.g.* taking the sum, mean or maximum of features (Fey & Lenssen, 2019; Gilmer et al., 2017; Qi et al., 2017; Wang et al., 2019; Xu et al., 2019; Kipf & Welling, 2017; Veličković et al., 2018; Hamilton et al., 2017; Klicpera et al., 2019a; Chen et al., 2020b; Xu et al., 2018). Our following scalability framework is based on the general message passing formulation given in Equation (1) and thus is applicable to this wide range of different GNN operators.

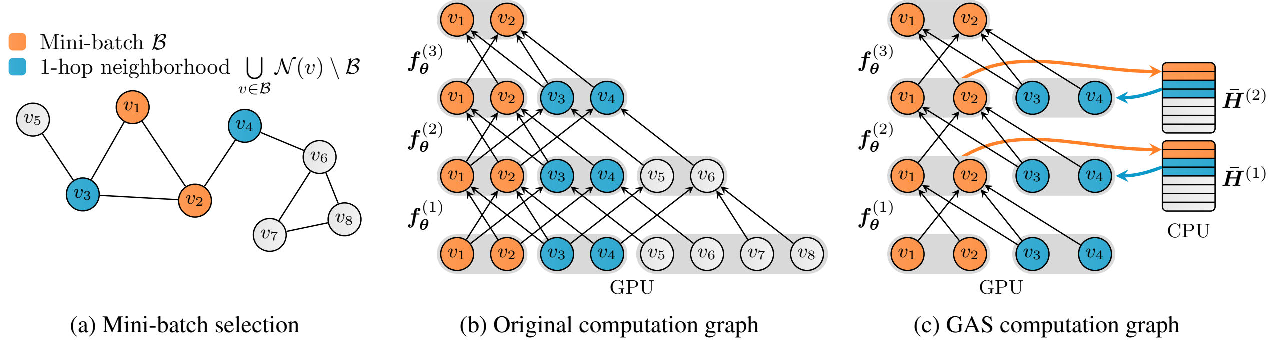
Historical Embeddings. Let $\boldsymbol{h}_v^{(\ell)}$ denote the node embedding in layer ℓ of a node $v \in \mathcal{B}$ in a mini-batch $\mathcal{B} \subseteq \mathcal{V}$. For the general message scheme given in Equation (1), the execution of $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}$ can be formulated as:

$$\begin{aligned}\boldsymbol{h}_v^{(\ell+1)} &= \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_v^{(\ell)}, \left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} \right\}\!\!\right\}_{w \in \mathcal{N}(v)}\right) \\ &= \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_v^{(\ell)}, \left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} \right\}\!\!\right\}_{w \in \mathcal{N}(v) \cap \mathcal{B}} \cup \left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} \right\}\!\!\right\}_{w \in \mathcal{N}(v) \setminus \mathcal{B}}\right) \\ &\approx \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_v^{(\ell)}, \left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} \right\}\!\!\right\}_{w \in \mathcal{N}(v) \cap \mathcal{B}} \cup \underbrace{\left\{\!\!\left\{ \bar{\boldsymbol{h}}_w^{(\ell)} \right\}\!\!\right\}_{w \in \mathcal{N}(v) \setminus \mathcal{B}}}_{\text{Historical embeddings}}\right)\end{aligned} \tag{2}$$

Here, we separate the neighborhood information of the multiset into *two* parts: **(1)** the local information of neighbors $\mathcal{N}(v)$ which are part of the current mini-batch $\mathcal{B}$, and **(2)** the information of neighbors which are not included in the current mini-batch. For out-of-mini-batch nodes, we approximate their embeddings via historical embeddings acquired in previous iterations of training (Chen et al., 2018b), denoted by $\bar{\boldsymbol{h}}_w^{(\ell)}$. After each step of training, the newly computed embeddings $\boldsymbol{h}_v^{(\ell+1)}$ are pushed to the history and serve as historical embeddings $\bar{\boldsymbol{h}}_w^{(\ell+1)}$ in future iterations. The separation of in-mini-batch nodes and out-of-mini-batch nodes, and their approximation via historical embeddings represent the foundation of our GAS framework.

A high-level illustration of its computation flow is visualized in Figure 1. Figure 1b shows the original data flow without

[1] https://github.com/rusty1s/pyg_autoscale



(a) Mini-batch selection (b) Original computation graph (c) GAS computation graph

Figure 1. **Mini-batch processing of GNNs with historical embeddings.** ■ denotes the nodes in the current mini-batch and ■ represents their direct 1-hop neighbors. For a given mini-batch (a), GPU memory and computation costs exponentially increase with GNN depth (b). The usage of historical embeddings avoids this problem as it allows to *prune* entire sub-trees of the computation graph, which leads to constant GPU memory consumption in respect to input node size (c). Here, nodes in the current mini-batch *push* their updated embeddings to the history $\bar{\boldsymbol{H}}^{(\ell)}$, while their direct neighbors *pull* their most recent historical embeddings from $\bar{\boldsymbol{H}}^{(\ell)}$ for further processing.

historical embeddings. The required GPU memory increases as the model gets deeper. After a few layers, embeddings for the entire input graph need to be stored, even if only a mini-batch of nodes is considered for loss computation. In contrast, historical embeddings eliminate this problem by approximating entire sub-trees of the computation graph, *cf.* Figure 1c. The required historical embeddings are pulled from an offline storage, instead of being re-computed in each iteration, which keeps the required information for each batch local. For a single batch $\mathcal{B} \subseteq \mathcal{V}$, the GPU memory footprint for one training step is given by $\mathcal{O}(|\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \cup \{v\}| \cdot L)$ and thus only scales linearly with the number of layers L. The majority of data (the histories) can be stored in RAM or hard drive storage rather than GPU memory.

In the following, we are going to use $\tilde{\boldsymbol{h}}_v^{(\ell)}$ to denote embeddings estimated via GAS (line 3 of Equation (2)) to differentiate them from the exact embeddings obtained without historical approximation (line 1 of Equation (2)). In contrast to existing scaling solutions based on sub-sampling edges, the usage of historical embeddings as utilized in GAS provides the following additional advantages:

(1) GAS trains over all the data: In GAS, a GNN will make use of all available graph information, *i.e.* *no* edges are dropped, which results in lower variance and more accurate estimations (since $\|\bar{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\| \ll \|\boldsymbol{h}_v^{(\ell)}\|$). Importantly, for a single epoch and layer, each edge is still only processed once, putting its time complexity $\mathcal{O}(|\mathcal{E}|)$ on par with its full-batch counterpart. Notably, more accurate estimations will further strengthen gradient estimation during backpropagation. Specifically, the model parameters will be updated based on the node embeddings of *all* neighbors since $\partial \tilde{\boldsymbol{h}}_v^{(\ell+1)} / \partial \boldsymbol{\theta}$ also depends on $\{\!\{ \bar{\boldsymbol{h}}_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B} \}\!\}$.

(2) GAS enables constant inference time complexity: The time complexity of model inference is reduced to a constant factor, since we can directly use the historical embeddings of the last layer to derive predictions for test nodes.

(3) GAS is simple to implement: Our scheme does not need to maintain recursive layer-wise computation graphs, which makes its overall implementation straightforward and comparable to full-batch training. Only minor modifications are required to *pull* information from and *push* information to the histories, *cf.* our training algorithm in the appendix.

(4) GAS provides theoretical guarantees: In particular, if the model weights are kept fixed, $\tilde{\boldsymbol{h}}_v^{(\ell)}$ eventually equals $\boldsymbol{h}_v^{(\ell)}$ after a fixed amount of iterations (Chen et al., 2018b).

3. Approximation Error and Expressiveness

The advantages of utilizing historical embeddings $\bar{\boldsymbol{h}}_v^{(\ell)}$ to compute an approximation $\tilde{\boldsymbol{h}}_v^{(\ell)}$ of the exact embedding $\boldsymbol{h}_v^{(\ell)}$ come at the cost of an approximation error $\|\tilde{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\|$, which can be decomposed into two sources of variance: **(1)** The *closeness* of estimated inputs to their exact values, *i.e.* $\|\tilde{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\| \geq 0$, and **(2)** the *staleness* of historical embeddings, *i.e.* $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \tilde{\boldsymbol{h}}_v^{(\ell-1)}\| \geq 0$. In the following, we show concrete bounds for this error, which can be then tightened using specific procedures. Here, our analysis focuses on arbitrary $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$ GNN layers as described in Equation (1), but we restrict both $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ to model k-Lipschitz continuous functions due to their potentially highly non-linear nature. Proofs of all lemmas and theorems can be found in the appendix.

Lemma 1 *Let* $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ *and* $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ *be Lipschitz continuous functions with Lipschitz constants* k_1 *and* k_2, *respectively. If, for all* $v \in \mathcal{V}$, *the inputs are close to the exact input,* i.e. $\|\tilde{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\| \leq \delta$, *and the historical embeddings do not run too stale,* i.e. $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \tilde{\boldsymbol{h}}_v^{(\ell-1)}\| \leq \epsilon$,

then the output error is bounded by

$$\|\tilde{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\| \leq \delta\, k_2 + (\delta + \epsilon)\, k_1\, k_2\, |\mathcal{N}(v)|.$$

Due to the behavior of Lipschitz constants in a series of function compositions, we obtain an upper bound that is dependent on k_1, k_2 and $|\mathcal{N}(v)|$, as well as dependent on the errors δ and ϵ of the inputs. Interestingly, sum aggregation, the most expressive aggregation function (Xu et al., 2019), introduces a factor of $|\mathcal{N}(v)|$ to the upper bound, while we can obtain a much tighter upper bound for mean or max aggregation, *cf.* its proof. Next, we take a look at the final output error produced by a L-layered GNN:

Theorem 2 *Let $\boldsymbol{f}_{\boldsymbol{\theta}}^{(L)}$ be a L-layered GNN, containing only Lipschitz continuous* $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ *and* $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ *functions with Lipschitz constants k_1 and k_2, respectively. If, for all $v \in \mathcal{V}$ and all $\ell \in \{1, \ldots, L-1\}$, the historical embeddings do not run too stale,* i.e. $\|\bar{\boldsymbol{h}}_v^{(\ell)} - \tilde{\boldsymbol{h}}_v^{(\ell)}\| \leq \epsilon^{(\ell)}$*, then the final output error is bounded by*

$$\|\tilde{\boldsymbol{h}}_{v,j}^{(L)} - \boldsymbol{h}_{v,j}^{(L)}\| \leq \sum_{\ell=1}^{L-1} \epsilon^{(\ell)}\, k_1^{L-\ell}\, k_2^{L-\ell}\, |\mathcal{N}(v)|^{L-\ell}.$$

Notably, this upper bound does not longer depend on $\|\tilde{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\| \leq \delta^{(\ell)}$, and is instead solely conditioned on the staleness of histories $\|\bar{\boldsymbol{h}}_v^{(\ell)} - \tilde{\boldsymbol{h}}_v^{(\ell)}\| \leq \epsilon^{(\ell)}$. However, it depends *exponentially* on the Lipschitz constants k_1 and k_2 as well as $|\mathcal{N}(v)|$ with respect to the number of layers. In particular, each additional layer introduces a less restrictive bound since the errors made in the first layers get immediately propagated to later ones, leading to potentially high inaccuracies for histories in deeper GNNs. We will later propose solutions for tightening the proven bound in practice, allowing the application of GAS to deep and non-linear GNNs. Furthermore, Theorem 2 lets us immediately derive an upper error bound of gradients as well, *i.e.*

$$\|\nabla_{\boldsymbol{\theta}} \mathcal{L}(\tilde{\boldsymbol{h}}_v^{(L)}) - \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{h}_v^{(L)})\| \leq \lambda \|\tilde{\boldsymbol{h}}_v^{(L)} - \boldsymbol{h}_v^{(L)}\|$$

in case $\mathcal{L}$ is λ-Lipschitz continuous. As such, GAS encourages low variance and bias in the learning signal as well. However, parameters are not guaranteed to converge to the same optimum since we explicitely consider arbitrary GNNs solving non-convex problems (Cong et al., 2020).

It is well known that the most powerful GNNs adhere to the same representational power as the *Weisfeiler-Lehman (WL) test* (Weisfeiler & Lehman, 1968) in distinguishing non-isomorphic structures, *i.e.* $\boldsymbol{h}_v^{(L)} \neq \boldsymbol{h}_w^{(L)}$ in case $c_v^{(L)} \neq c_w^{(L)}$ (Xu et al., 2019; Morris et al., 2019), where $c_v^{(L)}$ denotes a node's coloring after L rounds of color refinement. However, in order to leverage such expressiveness, a GNN needs to be able to reason about structural differences across neighborhoods directly *during* training. We now show that GNNs that scale by sampling edges are not capable of doing so:

Proposition 3 *Let $\boldsymbol{f}_{\boldsymbol{\theta}}^{(L)} \colon \mathcal{V} \to \mathbb{R}^d$ be a L-layered GNN as expressive as the WL test in distinguishing the L-hop neighborhood around each node $v \in \mathcal{V}$. Then, there exists a graph $\boldsymbol{A} \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ for which $\boldsymbol{f}_{\boldsymbol{\theta}}^{(L)}$ operating on a sampled variant $\tilde{\boldsymbol{A}}$, $\tilde{a}_{v,w} = \begin{cases} \frac{|\mathcal{N}(v)|}{|\tilde{\mathcal{N}}(v)|}, & \text{if } w \in \tilde{\mathcal{N}}(v) \\ 0, & \text{otherwise} \end{cases}$, produces a non-equivalent coloring,* i.e. $\tilde{\boldsymbol{h}}_v^{(L)} \neq \tilde{\boldsymbol{h}}_w^{(L)}$ *while $c_v^{(L)} = c_w^{(L)}$ for nodes $v, w \in \mathcal{V}$.*

While sampling strategies lose expressive power due to sub-sampling of edges, scalable GNNs based on historical embeddings are leveraging *all* edges during neighborhood aggregation. Therefore, a special interest lies in the question if historical-based GNNs are as expressive as their full-batch counterpart. Here, a maximally powerful *and* scalable GNN needs to fulfill the following two requirements: **(1)** It needs to be as expressive as the WL test in distinguishing non-isomorphic structures, and **(2)** it needs to account for the approximation error $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\|$ induced by the usage of historical embeddings. Since it is known that there exists a wide range of maximally powerful GNNs (Xu et al., 2019; Morris et al., 2019; Corso et al., 2020), we can restrict our analysis to the latter question. Following upon Xu et al. (2019), we focus on the case where input node features are from a countable set $\mathbb{P}^d \subset \mathbb{R}^d$ of bounded size:

Lemma 4 *Let $\{\!\{\boldsymbol{h}_v^{(\ell-1)} \colon v \in \mathcal{V}\}\!\}$ be a countable multiset such that $\|\boldsymbol{h}_v^{(\ell-1)} - \boldsymbol{h}_w^{(\ell-1)}\| > 2(\delta + \epsilon)$ for all $v, w \in \mathcal{V}$, $\boldsymbol{h}_v^{(\ell-1)} \neq \boldsymbol{h}_w^{(\ell-1)}$. If the inputs are close to the exact input,* i.e. $\|\tilde{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\| \leq \delta$*, and the historical embeddings do not run too stale,* i.e. $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \tilde{\boldsymbol{h}}_v^{(\ell-1)}\| \leq \epsilon$*, then there exist* $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ *and* $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ *functions, such that*

$$\|\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}(\tilde{\boldsymbol{h}}_v^{(\ell-1)}) - \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_v^{(\ell-1)})\| \leq \delta + \epsilon$$

and

$$\|\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_v^{(\ell-1)}) - \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_w^{(\ell-1)})\| > 2(\delta + \epsilon + \lambda)$$

for all $v, w \in \mathcal{V}$, $\boldsymbol{h}_v^{(\ell-1)} \neq \boldsymbol{h}_w^{(\ell-1)}$ and all $\lambda > 0$.

Informally, Lemma 4 tells us that if **(1)** exact input embeddings are sufficiently far apart from each other and **(2)** historical embeddings are sufficiently close to the exact embeddings, there exist historical-based GNN operators which can distinguish equal from non-equal inputs. Key to the proof is that $(\delta + \epsilon)$-balls around exact inputs do not intersect each other and are therefore well separated. Notably, we do not require $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$ to model strict injectivity since it is sufficient for $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$ to be $2(\delta + \epsilon)$-injective (Seo et al., 2019).

Following Xu et al. (2019), one can leverage MLPs to model and learn such MESSAGE and UPDATE functions due to the universal approximation theorem (Hornik et al., 1989; Hornik, 1991). However, the theory behind Lemma 4 holds for any maximally powerful GNN operator. Finally, we can use this insight to relate the expressiveness of scalable GNNs to the WL test color refinement procedure:

Theorem 5 *Let $\boldsymbol{f}_{\boldsymbol{\theta}}^{(L)}$ be a L-layered GNN in which all* $\text{MESSAGE}_{\boldsymbol{\theta}}^{(\ell)}$ *and* $\text{UPDATE}_{\boldsymbol{\theta}}^{(\ell)}$ *functions fulfill the conditions of Lemma 4. Then, there exists a map $\phi \colon \mathbb{R}^d \to \Sigma$ so that $\phi(\tilde{\boldsymbol{h}}_v^{(L)}) = c_v^{(L)}$ for all $v \in \mathcal{V}$.*

Theorem 5 extends the insights of Lemma 4 to multi-layered GNNs, and indicates that scalable GNNs using historical embeddings are able to distinguish non-isomorphic structures (that are distinguishable by the WL test) directly during training, which is what makes reasoning about structural properties possible. It should be noted that recent proposals such as DROPEDGE (Rong et al., 2020) are still applicable for data augmentation and message reduction. However, through the given theorem, we disentangle scalability and expressiveness from regularization via edge dropping.

While sampling approaches lose expressiveness compared to their original counterparts (*cf.* Proposition 3), Theorem 5 tells us that, in theory, there exist message passing functions that are as expressive as the WL test in distinguishing non-isomorphic structures while accounting for the effects of approximation in stored embeddings. In practice, we have two degrees of freedom to tighten the upper bounds given by Lemma 1 and Theorem 2, leading to a lower approximation error and higher expressiveness in return: **(1)** Minimizing the *staleness* of historical embeddings, and **(2)** maximizing the *closeness* of estimated inputs to their exact values by controlling the Lipschitz constants of UPDATE and MESSAGE functions. In what follows, we derive a list of procedures to achieve these goals:

Minimizing Inter-Connectivity Between Batches. As formulated in Equation (2) in Section 2, the output embeddings of $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}$ are exact if $|\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \cup \{v\}| = |\mathcal{B}|$, *i.e.* all neighbors of nodes in $\mathcal{B}$ are as well part of $\mathcal{B}$. However, in practice, this can only be guaranteed for full-batch GNNs. Motivated by this observation, we aim to minimize the inter-connectivity between sampled mini-batches, *i.e.* $\min |\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \setminus \mathcal{B}|$, which minimizes history access, and increases closeness and reduces staleness in return.

Similar to CLUSTER-GCN (Chiang et al., 2019), we make use of graph clustering techniques, *e.g.*, METIS (Karypis & Kumar, 1998; Dhillon et al., 2007), to achieve this goal. It aims to construct partitions over the nodes in a graph such that intra-links within clusters occur much more frequently than inter-links between different clusters. Intuitively, this results in a high chance that neighbors of a node are located in the same cluster. Notably, modern graph clustering methods are both fast and scalable with time complexities given by $\mathcal{O}(|\mathcal{E}|)$, and only need to be applied once, which leads to an unremarkable computational overhead in the pre-processing stage. In general, we argue that the METIS clustering technique is highly scalable, as it is in the heart of many large-scale distributed graph storage layers such as (Zhu et al., 2019; Zheng et al., 2020) that are known scale to billion-sized graphs. Furthermore, the additional overhead in the pre-processing stage is quickly compensated by an acceleration of training, since the number of neighbors outside of $\mathcal{B}$ is heavily reduced, and pushing information to the histories now leads to contiguous memory transfers.

Enforcing Local Lipschitz Continuity. To guide our neural network in learning a function with controllable error, we can enforce its intermediate output layers $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$ to be invariant to small input perturbations. In particular, following upon Usama & Chang (2018), we found it useful to apply the auxiliary loss

$$\mathcal{L}_{\text{reg}}^{(\ell)} = \| \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}(\tilde{\boldsymbol{h}}_v^{(\ell-1)}) - \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}(\tilde{\boldsymbol{h}}_v^{(\ell-1)} + \boldsymbol{\epsilon}) \| \tag{3}$$

in highly non-linear message passing phases, *e.g.*, in GIN (Xu et al., 2019). Such regularization enforces equal outputs for small pertubations $\boldsymbol{\epsilon} \sim \mathcal{B}_{\delta}(\mathbf{0})$ inside closed balls of radius δ. Notably, we do not restrict $\text{UPDATE}_{\boldsymbol{\theta}}^{(\ell)}$ and $\text{MESSAGE}_{\boldsymbol{\theta}}^{(\ell)}$ to separately model global k-Lipschitz continuous functions, but rather aim for local Lipschitz continuity at each $\boldsymbol{h}_v^{(\ell-1)}$ for $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$ as a whole. For other message passing GNNs, *e.g.*, in GCN (Kipf & Welling, 2017), L_2 regularization is usually sufficient to ensure closeness of historical embeddings. Further, we found gradient clipping to be an effective method to restrict the parameters from changing too fast, regularizing history changes in return.

4. Related Work

Our GAS framework utilizes historical embeddings as an affordable approximation. The idea of historical embeddings was originally introduced in VR-GCN (Chen et al., 2018b). VR-GCN aims to reduce the variance in estimation during neighbor sampling (Hamilton et al., 2017), and avoids the need to sample a large amount of neighbors in return. Cong et al. (2020) further simplified this scheme into a *one-shot sampling* scenario, where nodes no longer need to recursively explore neighborhoods in each layer. However, these approaches consider only a specific GNN operator which prevent their application to the wide range of GNN architectures available. Furthermore, they only consider shallow architectures and do not account for the increasing approximation error induced by deeper and expressive GNNs, which is well observable in practice, *cf.* Section 6.1.

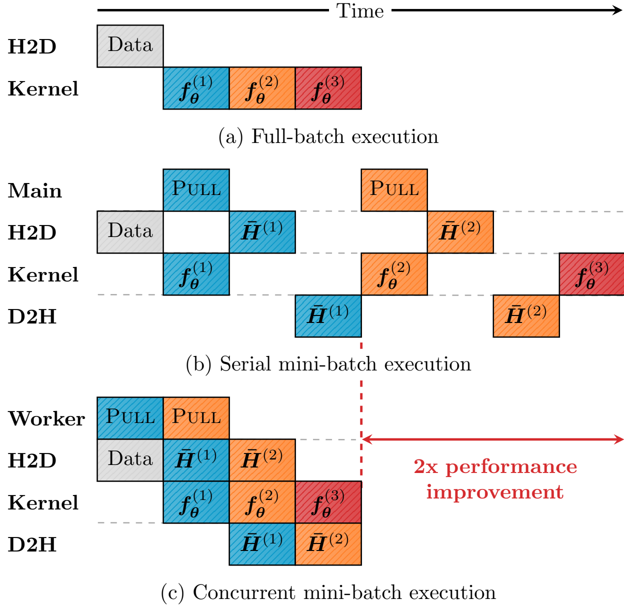


Figure 2. **Illustrative runtime performances of a serial and concurrent mini-batch execution in comparison to a full-batch GNN execution.** In the full-batch approach (a), all necessary data is first transferred to the device via the HOST2DEVICE (H2D) engine, before GNN layers are executed in serial inside the kernel engine. As depicted in (b), a serial mini-batch execution suffers from an I/O bottleneck, in particular because each kernel engine has to wait for memory transfers to complete. The concurrent mini-batch execution (c) avoids this problem by leveraging an additional worker thread and overlapping data transfers, leading to two times performance improvements in comparison to a serial execution, which is on par with the standard full-batch approach.

In order to minimize the inter-connectivity between mini-batches, we utilize graph clustering techniques for mini-batch selection, as first introduced in the subgraph sampling approach CLUSTER-GCN (Chiang et al., 2019). CLUSTER-GCN leverages clustering in order to infer meaningful subgraphs, while we aim to minimize history accesses. Furthermore, CLUSTER-GCN limits message passing to intra-connected nodes, and therefore ignores potentially useful information outside the current mini-batch. This inherently limits the model to learn from nodes nearby. In contrast, our GAS framework makes use of *all* available neighborhood data for aggregation, and therefore avoids this downside.

5. PyGAS: Auto-Scaling GNNs in PyG

We condense our GAS framework and theoretical findings into a tool named *PyGAS* that implements all the presented techniques in practice.[2] *PyGAS* is built upon PYTORCH (Paszke et al., 2019) and utilizes the PYTORCH GEOMETRIC (PyG) library (Fey & Lenssen, 2019). It provides an easy-to-use interface to convert common and custom GNN models from PYTORCH GEOMETRIC into their scalable variants. Furthermore, it provides a fully deterministic test bed for evaluating models on large-scale graphs. An example of the interface is shown in the appendix.

[2] https://github.com/rusty1s/pyg_autoscale

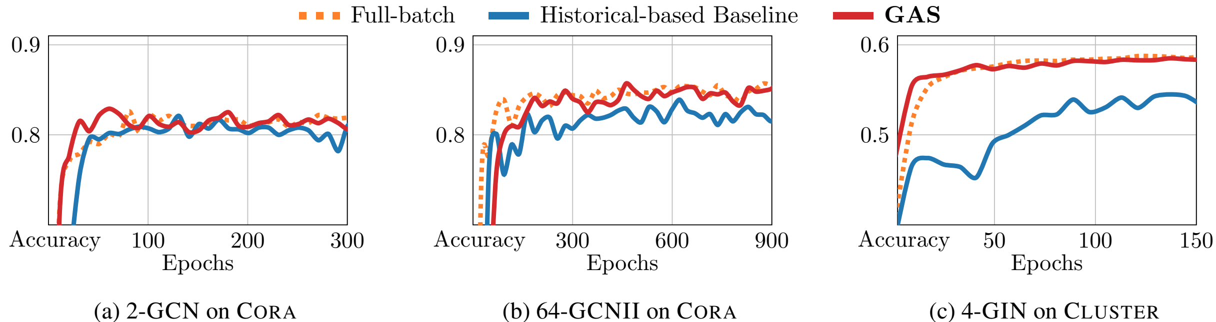
Fast Historical Embeddings. Our approach accesses histories to account for any data outside the current mini-batch, which requires frequent data transfers to and from the GPU. Therefore, *PyGAS* optimizes pulling from and pushing to histories via *non-blocking* device transfers. Specifically, we immediately start pulling historical embeddings for each layer asynchronously at the beginning of each optimization step, which ensures that GPUs do not run idle while waiting for memory transfers to complete. A separate worker thread gathers historical information into one of multiple pinned CPU memory buffers (denoted by PULL), from where it can be transfered to the GPU via the usage of CUDA streams without blocking any CPU or CUDA execution. Synchronization is done by synchronizing the respective CUDA stream before inputting the transferred data into the GNN layer. The same strategy is applied for pushing information to the history. Considering that the device transfer of $\bar{\boldsymbol{H}}^{(\ell-1)}$ is faster than the execution of $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$, this scheme does not lead to any runtime overhead when leveraging historical embeddings and can be twice as fast as its serial non-overlapping counterpart, *cf.* Figure 2. We have implemented our non-blocking transfer scheme with custom C++/CUDA code to avoid Python's global interpreter lock.

6. Experiments

In this section, we evaluate our GAS framework in practice using *PyGAS*, utilizing 6 different GNN operators and 15 datasets. Please refer to the appendix for a detailed description of the used GNN operators and datasets, and to our code for hyperparameter configurations. All models were trained on a single GeForce RTX 2080 Ti (11 GB). In our experiments, we hold all histories in RAM, using a machine with 64GB of CPU memory.

6.1. GAS resembles full-batch performance

First, we analyze how GAS affects the robustness and expressiveness of our method. We compare GAS against two different baselines: a regular full-batch variant and a history baseline, which naively integrates history-based mini-batch training without any of the additional GAS techniques. To evaluate, we make use of a shallow 2-layer GCN (Kipf & Welling, 2017) and two recent state-of-the-art models: a deep GCNII network with 64 layers (Chen et al., 2020b), and a maximally expressive GIN network with 4 layers (Xu et al., 2019). We evaluate those models on tasks for which



(a) 2-GCN on CORA (b) 64-GCNII on CORA (c) 4-GIN on CLUSTER

Figure 3. **Model performance comparison between full-batch, an unoptimized history-based baseline and our GAS approach.** In contrast to the historical-based baseline, GAS reaches the quality of full-batch training, especially for (b) deep and (c) expressive models.

Table 1. **Full-batch vs GAS performance on small transductive graph benchmark datasets across 20 different initializations.** Predictive performance of models trained via GAS closely matches those of full-batch gradient descent on all models for all datasets.
† Results omitted due to unstable performance across different weight initializations, *cf.* Shchur et al. (2018)

Dataset	GCN		GAT		APPNP		GCNII	
	Full	**GAS**	Full	**GAS**	Full	**GAS**	Full	**GAS**
CORA	81.88±0.75	82.29±0.76	82.80±0.47	83.32±0.62	83.28±0.60	83.19±0.58	85.04±0.53	85.52±0.39
CITESEER	70.98±0.66	71.18±0.97	71.72±0.91	71.86±1.00	72.13±0.73	72.63±0.82	73.06±0.81	73.89±0.48
PUBMED	78.73±1.10	79.23±0.62	78.03±0.40	78.42±0.56	80.21±0.20	79.82±0.52	79.72±0.78	80.19±0.49
COAUTHOR-CS	91.08±0.59	91.22±0.45	90.31±0.49	90.38±0.42	92.51±0.47	92.44±0.58	92.45±0.35	92.52±0.31
COAUTHOR-PHYSICS	93.10±0.84	92.98±0.72	92.32±0.86	92.80±0.61	93.40±0.92	93.68±0.61	93.43±0.52	93.61±0.41
AMAZON-COMPUTER	81.17±1.81	80.84±2.26	—†	—†	81.79±2.00	81.66±1.81	83.04±1.81	83.05±1.16
AMAZON-PHOTO	90.25±1.66	90.53±1.40	—†	—†	91.27±1.26	91.23±1.34	91.42±0.81	91.60±0.78
WIKI-CS	79.08±0.50	79.00±0.41	79.44±0.41	79.56±0.47	79.88±0.40	79.75±0.53	79.94±0.67	80.02±0.43
Δ Mean Accuracy	**+0.13**		**+0.29**		**-0.01**		**+0.29**	

they are well suitable: classifying academic papers in a citation network (CORA), and identifying community clusters in Stochastic Block Models (CLUSTER) (Yang et al., 2016; Dwivedi et al., 2020), *cf.* Figure 3. Since CLUSTER is a node classification task containing multiple graphs, we first convert it into a super graph (holding all the nodes of all graphs), and partition this super graph using twice as many partitions as there are initial graphs. It can be seen that especially for deep (64-GCNII, *cf.* Figure 3b) and expressive (4-GIN, *cf.* Figure 3c) architectures, the naive historical-based baseline fails to reach the desired full-batch performance. This can be contributed to the high approximation error induced by deep and expressive models. In contrast, GAS shows far superior performance, reaching the quality of full-batch training in both cases.

In general, we expect the model performances of our GAS mini-batch training to closely resemble the performances of their full-batch counterparts, except for the variance introduced by stochastic optimization (which is, in fact, known to improve generalization (Bottoue & Bousquet, 2007)). To validate, we compare our approach against full-batch performances on small transductive benchmark datasets for which full-batch training is easily feasible. We evaluate on four GNN models that significantly advanced the field of graph representation learning: GCN (Kipf & Welling, 2017), GAT (Veličković et al., 2018), APPNP (Klicpera et al., 2019a) and GCNII (Chen et al., 2020b). For all experiments, we tried to follow the hyperparameter setup of the respective papers as closely as possible and perform an in-depth grid search on datasets for which best performing configurations are not known. We then apply GAS mini-batch training on the *same* set of hyperparameters. As shown in Table 1, all models that utilize GAS training perform as well as their full-batch equivalents (with slight gains overall), confirming the practical effectiveness of our approach. Notably, even for deep GNNs such as APPNP and GCNII, our approach is able to closely resemble the desired performance.

We further conduct an ablation study to highlight the individual performance improvements of our GAS techniques within a GCNII model, *i.e.* minimizing inter-connectivity and applying regularization techniques. Table 2 shows the relative performance improvements of individual GAS techniques in percentage points, compared to the corresponding model performance obtained by full-batch training. Notably, it can be seen that both techniques contribute to resembling full-batch performance, reaching their full strength when used in combination. We include an additional ablation study for training an expressive GIN model in the appendix.

Table 2. **Relative performance improvements of individual GAS techniques within a GCNII model.** The performance improvement is measured in percentage points in relation to the corresponding model performance obtained by full-batch training.

	CORA	CITESEER	PUBMED	COAUTHOR-CS	COAUTHOR-PHYSICS	AMAZON-COMPUTER	AMAZON-PHOTO	WIKI-CS
Baseline	-3.26	-5.66	-3.20	-0.79	-0.50	-5.76	-4.16	-3.19
Regularization	-2.12	-1.03	-1.24	-0.46	-0.24	-3.02	-1.19	-0.74
METIS	-1.57	-3.12	-1.50	-0.47	+0.13	-2.75	-1.02	-0.24
GAS	**+0.48**	**+0.83**	**+0.47**	**+0.07**	**+0.18**	**+0.01**	**+0.18**	**+0.08**

Table 3. **GPU memory consumption (in GB) and the amount of data used (%) across different GNN execution techniques.** GAS consumes low memory while making use of all available neighborhood information during a single optimization step.

	# nodes	717K	169K	2.4M
	# edges	7.9M	1.2M	61.9M
	Method	YELP	ogbn-arxiv	ogbn-products
2-layer	Full-batch	6.64GB/100%	1.44GB/100%	21.96GB/100%
2-layer	GRAPHSAGE	0.76GB/ 9%	0.40GB/ 27%	0.92GB/ 2%
2-layer	CLUSTER-GCN	0.17GB/ 13%	0.15GB/ 40%	0.16GB/ 16%
2-layer	**GAS**	0.51GB/100%	0.22GB/100%	0.36GB/100%
3-layer	Full-batch	9.44GB/100%	2.11GB/100%	31.53GB/100%
3-layer	GRAPHSAGE	2.19GB/ 14%	0.93GB/ 33%	4.34GB/ 5%
3-layer	CLUSTER-GCN	0.23GB/ 13%	0.22GB/ 40%	0.23GB/ 16%
3-layer	**GAS**	0.79GB/100%	0.34GB/100%	0.59GB/100%
4-layer	Full-batch	12.24GB/100%	2.77GB/100%	41.10GB/100%
4-layer	GRAPHSAGE	4.31GB/ 19%	1.55GB/ 37%	11.23GB/ 8%
4-layer	CLUSTER-GCN	0.30GB/ 13%	0.29GB/ 40%	0.29GB/ 16%
4-layer	**GAS**	1.07GB/100%	0.46GB/100%	0.82GB/100%

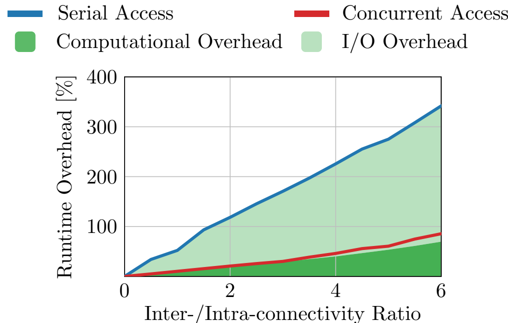


Figure 4. **Runtime overhead in relation to the inter-/intra-connectivity ratio of mini-batches, both for serial and concurrent history access patterns.** The overall runtime overhead is further separated into computational overhead (overhead of aggregating additional messages) and I/O overhead (overhead of pulling from and pushing to histories). Our concurrent memory transfer reduces I/O overhead caused by histories by a wide margin.

6.2. GAS is fast and memory-efficient

For training large-scale GNNs, GPU memory consumption will directly dictate the scalability of the given approach. Here, we show how GAS maintains a low GPU memory footprint while, in contrast to other scalability approaches, accounts for *all* available information inside a GNN's receptive field in a single optimization step. We compare the memory usage of GCN+GAS training with the memory usage of full-batch GCN, and mini-batch GRAPHSAGE (Hamilton et al., 2017) and CLUSTER-GCN (Chiang et al., 2019) training, *cf.* Table 3. Notably, GAS is easily able to fit the required data on the GPU, while memory consumption only increases linearly with the number of layers. Although CLUSTER-GCN maintains an overall lower memory footprint than GAS, it will only utilize a fraction of available information inside its receptive field, *i.e.* ≈23% on average.

We now analyze how GAS enables large-scale training due to fast mini-batch execution. Specifically, we are interested in how our concurrent memory transfer scheme (*cf.* Section 5) reduces the overhead induced by accessing historical embeddings from the offline storage. For this, we evaluate runtimes of a 4-layer GIN model on synthetic graph data, which allows fine-grained control over the ratio between inter- and intra-connected nodes, *cf.* Figure 4. Here, a given mini-batch consists of exactly 4,000 nodes which are randomly intra-connected to 60 other nodes. We vary the number of inter-connections (connections to nodes outside of the batch) by adding out-of-batch nodes that are randomly inter-connected to 60 nodes inside the batch. Notably, the naive serial memory transfer increases runtimes up to 350%, which indicates that frequent history accesses can cause major I/O bottlenecks. In contrast, our concurrent access pattern incurs *almost no* I/O overhead *at all*, and the overhead in execution time is solely explained by the computational overhead of aggregating far more messages during message propagation. Note that in most real-world scenarios, the additional aggregation of history data may only increase runtimes up to 25%, since most real-world datasets contain inter-/intra-connectivity ratios between 0.1 and 2.5, *cf.* appendix. Further, the additional overhead of computing METIS partitions in the pre-processing stage is negligible and is quickly mitigated by faster training times: Computing the partitioning of a graph with 2M nodes takes only about 20–50 seconds (depending on the number of clusters).

Next, we compare runtimes and memory consumption of GAS to the recent GTTF proposal (Markowitz et al., 2021),

Table 4. **Efficiency of GCN with GTTF and GAS.**

Dataset	**Runtime** (s)		**Memory** (MB)	
	GTTF	**GAS**	GTTF	**GAS**
CORA	0.077	**0.006**	18.01	**2.13**
PUBMED	0.071	**0.006**	28.79	**2.19**
PPI	0.976	**0.007**	134.86	**12.37**
FLICKR	1.178	**0.007**	325.97	**16.32**

which utilizes a fast neighbor sampling strategy based on tensor functionals. For this, we make use of a 4-layered GCN model with equal mini-batch and receptive field sizes. As shown in Table 4, GAS is both faster and consumes less memory than GTTF. Although GTTF makes use of a fast vectorized sampling procedure, its underlying recursive neighborhood construction still scales *exponentially* with GNN depth, which explains the observable differences in runtime and memory consumption.

6.3. GAS scales to large graphs

In order to demonstrate the scalability and generality of our approach, we scale various GNN operators to common large-scale graph benchmark datasets. Here, we focus our analysis on GNNs that are notorious hard to scale-up but have the potential to leverage the increased amount of available data to make more accurate predictions. In particular, we benchmark deep GNNs, *i.e.* GCNII (Chen et al., 2020b), and expressive GNNs, *i.e.* PNA (Corso et al., 2020). Note that it is not possible to run those models in full-batch mode on most of these datasets as they will run out of memory on common GPUs. We compare with 10 scalable GNN baselines: GRAPHSAGE (Hamilton et al., 2017), FASTGCN (Chen et al., 2018a), LADIES (Zou et al., 2019), VR-GCN (Chen et al., 2018b), MVS-GNN (Cong et al., 2020), CLUSTER-GCN (Chiang et al., 2019), GRAPHSAINT (Zeng et al., 2020b), SGC (Wu et al., 2019), SIGN (Frasca et al., 2020) and GBP (Chen et al., 2020a). Since results are hard to compare across different approaches due to differences in frameworks, model implementations, weight initializations and optimizers, we additionally report a shallow GCN+GAS baseline. GAS is able to train all models on all datasets on a single GPU, while holding corresponding histories in CPU memory. On the largest dataset, *i.e.* `ogbn-products`, this will consume $\approx L\cdot$ 2GB of storage for L layers, which easily fits in RAM on most modern workstations.

As can be seen in Table 5, the usage of deep and expressive models within our framework advances the state-of-the-art on REDDIT and FLICKR, while it performs equally well for others, *e.g.*, PPI. Notably, our approach outperforms the two historical-based variants VR-GCN and MVS-GNN by a wide margin. Interestingly, our deep and expressive variants reach superior performance than our GCN baseline

Table 5. **Performance on large graph datasets.** GAS is both scalable and general while achieving state-of-the-art performance.

	# nodes	230K	57K	89K	717K	169K	2.4M
	# edges	11.6M	794K	450K	7.9M	1.2M	61.9M
	Method	REDDIT	PPI	FLICKR	YELP	`ogbn-arxiv`	`ogbn-products`
	GRAPHSAGE	95.40	61.20	50.10	63.40	71.49	78.70
	FASTGCN	93.70	—	50.40	—	—	—
	LADIES	92.80	—	—	—	—	—
	VR-GCN	94.50	85.60	—	61.50	—	—
	MVS-GNN	94.90	89.20	—	62.00	—	—
	CLUSTER-GCN	96.60	99.36	48.10	60.90	—	78.97
	GRAPHSAINT	97.00	**99.50**	51.10	65.30	—	79.08
	SGC	96.40	96.30	48.20	64.00	—	—
	SIGN	96.80	97.00	51.40	63.10	—	77.60
	GBP	—	99.30	—	**65.40**	—	—
Full-batch	GCN	95.43	97.58	53.73	OOM	71.64	OOM
	GCNII	OOM	OOM	55.28	OOM	72.83	OOM
	PNA	OOM	OOM	56.23	OOM	72.17	OOM
GAS	GCN	95.45	98.92	54.00	62.94	71.68	76.66
	GCNII	96.77	**99.50**	56.20	65.14	**73.00**	77.24
	PNA	**97.17**	99.44	**56.67**	64.40	72.50	**79.91**

on *all* datasets, which highlights the benefits of evaluating larger models on larger scale.

7. Conclusion and Future Work

We proposed a general framework for scaling arbitrary message passing GNNs to large graphs without the necessity to sub-sample edges. As we have shown, our approach is able to train both deep and expressive GNNs in a scalable fashion. Notably, our approach is *orthogonal* to many methodological advancements, such as unifying GNNs and label propagation (Shi et al., 2020), graph diffusion (Klicpera et al., 2019b), or random wiring (Valsesia et al., 2020), which we like to investigate further in future works. While our experiments focus on node-level tasks, our work is technically able to scale the training of GNNs for edge-level and graph-level tasks as well. However, this still needs to be verified empirically. Another interesting future direction is the fusion of GAS into a distributed training algorithm (Jia et al., 2020; Ma et al., 2019; Zhu et al., 2016; Tripathy et al., 2020; Wan et al., 2020; Angerd et al., 2020; Zheng et al., 2020), and to extend our framework in accessing histories from disk storage rather than CPU memory. Overall, we hope that our findings lead to the development of sophisticated and expressive GNNs evaluated on large-scale graphs.

Acknowledgements

This work has been supported by the *German Research Association (DFG)* within the Collaborative Research Center SFB 876 *Providing Information by Resource-Constrained Analysis*, projects A6 and B2.

References

Angerd, A., Balasubramanian, K., and Annavaram, M. Distributed training of graph convolutional networks using subgraph approximation. *ICLR submission*, 2020.

Bottoue, L. and Bousquet, O. The tradeoffs of large scale learning. In *NIPS*, 2007.

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

Chen, J., Zhu, J., and Song, L. Stochastic training of graph convolutional networks with variance reduction. In *ICML*, 2018b.

Chen, M., Wei, Z., Ding, B., Li, Y., Yuan, Y., Du, X., and Wen, J. R. Scalable graph neural networks via bidirectional propagation. In *NeurIPS*, 2020a.

Chen, M., Wei, Z., Huang, Z., Ding, B., and Li, Y. Simple and deep graph convolutional networks. In *ICML*, 2020b.

Chiang, W. L., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C. J. Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks. In *KDD*, 2019.

Cong, W., Forsati, R., Kandemir, M., and Mahdavi, M. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *KDD*, 2020.

Corso, G., Cavalleri, L., Beaini, D., Liò, P., and Veličković, P. Principal neighbourhood aggregation for graph nets. In *NeurIPS*, 2020.

Dhillon, I. S., Guan, Y., and Kulis, B. Weighted graph cuts without eigenvectors: A multilevel approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29 (11):1944–1957, 2007.

Dwivedi, V. P., Joshi, C. K., Laurent, T., Bengio, Y., and Bresson, X. Benchmarking graph neural networks. *CoRR*, abs/2003.00982, 2020.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR-W*, 2019.

Frasca, F., Rossi, E., Eynard, D., Chamberlain, B., Bronstein, M. M., and Monti, F. SIGN: Scalable inception graph neural networks. In *ICML-W*, 2020.

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

Hamilton, W. L. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2020.

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

Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

Huang, Q., He, H., Singh, A., Lim, S. N., and Benson, A. R. Combining label propagation and simple models out-performs graph neural networks. In *ICLR*, 2021.

Huang, W., Zhang, T., Rong, Y., and Huang, J. Adaptive sampling towards fast graph representation learning. In *NeurIPS*, 2018.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with ROC. *Proceedings of Machine Learning and Systems*, pp. 187–198, 2020.

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

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

Klicpera, J., Bojchevski, A., and Günnemann, S. Predict then propagate: Graph neural networks meet personalized PageRank. In *ICLR*, 2019a.

Klicpera, J., Weißenberger, S., and Günnemann, S. Diffusion improves graph learning. In *NeurIPS*, 2019b.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. NeuGraph: Parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference*, 2019.

Ma, Y. and Tang, J. *Deep Learning on Graphs*. Cambridge University Press, 2020.

Markowitz, E., Balasubramanian, K., Mirtaheri, M., Abu-El-Haija, S., Perozzi, B., Ver Steeg, G., and Galstyan, A. Graph traversal with tensor functionals: A meta-algorithm for scalable learning. In *ICLR*, 2021.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *AAAI*, 2019.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang,

L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Qi, C. R., Yi, L., Su, H., and Guibas, L. J. PointNet++: Deep hierarchical feature learning on point sets in a metric space. In *NIPS*, 2017.

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

Seo, Y., Loukas, A., and Perraudin, N. Discriminative structural graph classification. *CoRR*, abs/1905.13422, 2019.

Shchur, O., Mumme, M., Bojchevski, A., and Günnemann, S. Pitfalls of graph neural network evaluation. In *NeurIPS-W*, 2018.

Shi, Y., Huang, Z., Wang, W., Zhong, H., Feng, S., and Sun, Y. Masked label prediction: Unified message passing model for semi-supervised classification. *CoRR*, abs/2009.03509, 2020.

Tripathy, A., Yelick, K., and Buluc, A. Reducing communcation in graph neural network training. *CoRR*, abs/2005.03300, 2020.

Usama, M. and Chang, D. E. Towards robust neural networks with lipschitz continuity. *CoRR*, abs/1811.09008, 2018.

Valsesia, D., Fracastoro, G., and Magli, E. Don't stack layers in graph neural networks, wire them randomly. *ICLR submission*, 2020.

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

Wan, C., Li, Y., Kim, N. S., and Lin, Y. BDS-GCN: Efficient full-graph training of graph convolutional nets with partition-parallelism and boundary sampling. *ICLR submission*, 2020.

Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., and Solomon, J. M. Dynamic graph CNN for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 2019.

Weisfeiler, B. and Lehman, A. A. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9), 1968.

Wu, F., Zhang, T., de Souza Jr., A. H., Fifty, C., Yu, T., and Weinberger, K. Q. Simplifying graph convolutional networks. In *ICML*, 2019.

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

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

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

You, Y., Chen, T., Wang, Z., and Shen, Y. L^2-GCN: Layer-wise and learned efficient training of graph convolutional networks. In *CVPR*, 2020.

Yu, L., Shen, J., Li, J., and Lerer, A. Scalable graph neural networks for heterogeneous graphs. *CoRR*, abs/2011.09679, 2020.

Zeng, H., Zhang, M., Xia, Y., Srivastava, A., Kannan, R., Prasanna, V., Jin, L., Malevich, A., and Chen, R. Deep graph neural networks with shallow subgraph samplers. *CoRR*, abs/2012.01.380, 2020a.

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

Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. DistDGL: Distributed graph neural network for training for billion-scale graphs. *CoRR*, abs/2010.05337, 2020.

Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., and Zhou, J. AliGraph: A comprehensive graph neural network platform. In *KDD*, 2019.

Zhu, X., Chen, W., Zheng, W., and Ma, X. Gemini: A computation-centric distributed graph processing system. In *USENIX Symposium on Operating Systems Designand Implementation*, 2016.

Zou, D., Hu, Z., Wang, Y., Jiang, S., Sun, Y., and Gu, Q. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *NeurIPS*, 2019.