# Inductive Representation Learning on Large Graphs

**William L. Hamilton**[*]
wleif@stanford.edu

**Rex Ying**[*]
rexying@stanford.edu

**Jure Leskovec**
jure@cs.stanford.edu

Department of Computer Science
Stanford University
Stanford, CA, 94305

## Abstract

Low-dimensional embeddings of nodes in large graphs have proved extremely useful in a variety of prediction tasks, from content recommendation to identifying protein functions. However, most existing approaches require that all nodes in the graph are present during training of the embeddings; these previous approaches are inherently *transductive* and do not naturally generalize to unseen nodes. Here we present GraphSAGE, a general *inductive* framework that leverages node feature information (e.g., text attributes) to efficiently generate node embeddings for previously unseen data. Instead of training individual embeddings for each node, we learn a function that generates embeddings by sampling and aggregating features from a node's local neighborhood. Our algorithm outperforms strong baselines on three inductive node-classification benchmarks: we classify the category of unseen nodes in evolving information graphs based on citation and Reddit post data, and we show that our algorithm generalizes to completely unseen graphs using a multi-graph dataset of protein-protein interactions.

## 1 Introduction

Low-dimensional vector embeddings of nodes in large graphs[1] have proved extremely useful as feature inputs for a wide variety of prediction and graph analysis tasks [5, 11, 28, 35, 36]. The basic idea behind node embedding approaches is to use dimensionality reduction techniques to distill the high-dimensional information about a node's neighborhood into a dense vector embedding. These node embeddings can then be fed to downstream machine learning systems and aid in tasks such as node classification, clustering, and link prediction [11, 28, 35].

However, previous works have focused on embedding nodes from a single fixed graph, and many real-world applications require embeddings to be quickly generated for unseen nodes, or entirely new (sub)graphs. This inductive capability is essential for high-throughput, production machine learning systems, which operate on evolving graphs and constantly encounter unseen nodes (e.g., posts on Reddit, users and videos on Youtube). An inductive approach to generating node embeddings also facilitates generalization across graphs with the same form of features: for example, one could train an embedding generator on protein-protein interaction graphs derived from a model organism, and then easily produce node embeddings for data collected on new organisms using the trained model.

The inductive node embedding problem is especially difficult, compared to the transductive setting, because generalizing to unseen nodes requires "aligning" newly observed subgraphs to the node embeddings that the algorithm has already optimized on. An inductive framework must learn to

[*]The two first authors made equal contributions.

[1]While it is common to refer to these data structures as social or biological *networks*, we use the term *graph* to avoid ambiguity with neural network terminology.

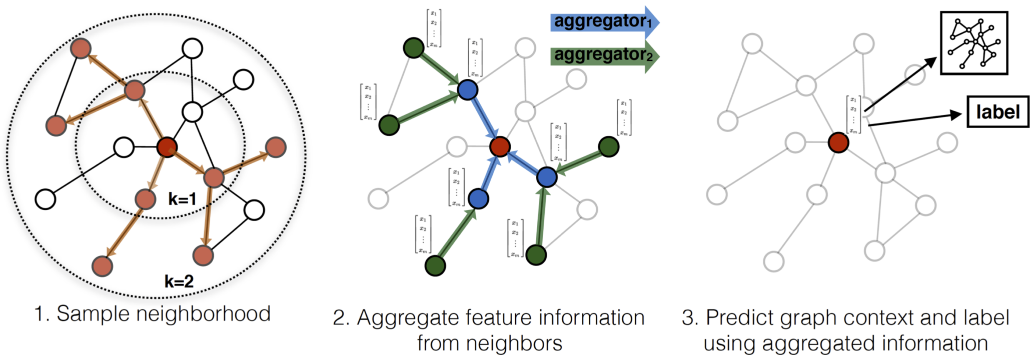
k=1
k=2
1. Sample neighborhood
aggregator1
aggregator2
2. Aggregate feature information from neighbors
label
3. Predict graph context and label using aggregated information

Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

recognize structural properties of a node's neighborhood that reveal both the node's local role in the graph, as well as its global position.

Most existing approaches to generating node embeddings are inherently transductive. The majority of these approaches directly optimize the embeddings for each node using matrix-factorization-based objectives, and do not naturally generalize to unseen data, since they make predictions on nodes in a single, fixed graph [5, 11, 23, 28, 35, 36, 37, 39]. These approaches can be modified to operate in an inductive setting (e.g., [28]), but these modifications tend to be computationally expensive, requiring additional rounds of gradient descent before new predictions can be made. There are also recent approaches to learning over graph structures using convolution operators that offer promise as an embedding methodology [17]. So far, graph convolutional networks (GCNs) have only been applied in the transductive setting with fixed graphs [17, 18]. In this work we both extend GCNs to the task of inductive unsupervised learning and propose a framework that generalizes the GCN approach to use trainable aggregation functions (beyond simple convolutions).

**Present work**. We propose a general framework, called GraphSAGE (SAmple and aggreGatE), for inductive node embedding. Unlike embedding approaches that are based on matrix factorization, we leverage node features (e.g., text attributes, node profile information, node degrees) in order to learn an embedding function that generalizes to unseen nodes. By incorporating node features in the learning algorithm, we simultaneously learn the topological structure of each node's neighborhood as well as the distribution of node features in the neighborhood. While we focus on feature-rich graphs (e.g., citation data with text attributes, biological data with functional/molecular markers), our approach can also make use of structural features that are present in all graphs (e.g., node degrees). Thus, our algorithm can also be applied to graphs without node features.

Instead of training a distinct embedding vector for each node, we train a set of *aggregator functions* that learn to aggregate feature information from a node's local neighborhood (Figure 1). Each aggregator function aggregates information from a different number of hops, or search depth, away from a given node. At test, or inference time, we use our trained system to generate embeddings for entirely unseen nodes by applying the learned aggregation functions. Following previous work on generating node embeddings, we design an unsupervised loss function that allows GraphSAGE to be trained without task-specific supervision. We also show that GraphSAGE can be trained in a fully supervised manner.

We evaluate our algorithm on three node-classification benchmarks, which test GraphSAGE's ability to generate useful embeddings on unseen data. We use two evolving document graphs based on citation data and Reddit post data (predicting paper and post categories, respectively), and a multi-graph generalization experiment based on a dataset of protein-protein interactions (predicting protein functions). Using these benchmarks, we show that our approach is able to effectively generate representations for unseen nodes and outperform relevant baselines by a significant margin: across domains, our supervised approach improves classification F1-scores by an average of 51% compared to using node features alone and GraphSAGE consistently outperforms a strong, transductive baseline [28], despite this baseline taking $\sim$100$\times$ longer to run on unseen nodes. We also show that the new aggregator architectures we propose provide significant gains (7.4% on average) compared to an aggregator inspired by graph convolutional networks [17]. Lastly, we probe the expressive capability of our approach and show, through theoretical analysis, that GraphSAGE is capable of learning structural information about a node's role in a graph, despite the fact that it is inherently based on features (Section 5).

## 2 Related work

Our algorithm is conceptually related to previous node embedding approaches, general supervised approaches to learning over graphs, and recent advancements in applying convolutional neural networks to graph-structured data.[2]

**Factorization-based embedding approaches**. There are a number of recent node embedding approaches that learn low-dimensional embeddings using random walk statistics and matrix factorization-based learning objectives [5, 11, 28, 35, 36]. These methods also bear close relationships to more classic approaches to spectral clustering [23], multi-dimensional scaling [19], as well as the PageRank algorithm [25]. Since these embedding algorithms directly train node embeddings for individual nodes, they are inherently transductive and, at the very least, require expensive additional training (e.g., via stochastic gradient descent) to make predictions on new nodes. In addition, for many of these approaches (e.g., [11, 28, 35, 36]) the objective function is invariant to orthogonal transformations of the embeddings, which means that the embedding space does not naturally generalize between graphs and can drift during re-training. One notable exception to this trend is the Planetoid-I algorithm introduced by Yang et al. [40], which is an inductive, embedding-based approach to semi-supervised learning. However, Planetoid-I does not use any graph structural information during inference; instead, it uses the graph structure as a form of regularization during training. Unlike these previous approaches, we leverage feature information in order to train a model to produce embeddings for unseen nodes.

**Supervised learning over graphs**. Beyond node embedding approaches, there is a rich literature on supervised learning over graph-structured data. This includes a wide variety of kernel-based approaches, where feature vectors for graphs are derived from various graph kernels (see [32] and references therein). There are also a number of recent neural network approaches to supervised learning over graph structures [7, 10, 21, 31]. Our approach is conceptually inspired by a number of these algorithms. However, whereas these previous approaches attempt to classify entire graphs (or subgraphs), the focus of this work is generating useful representations for individual nodes.

**Graph convolutional networks**. In recent years, several convolutional neural network architectures for learning over graphs have been proposed (e.g., [4, 9, 8, 17, 24]). The majority of these methods do not scale to large graphs or are designed for whole-graph classification (or both) [4, 9, 8, 24]. However, our approach is closely related to the graph convolutional network (GCN), introduced by Kipf et al. [17, 18]. The original GCN algorithm [17] is designed for semi-supervised learning in a transductive setting, and the exact algorithm requires that the full graph Laplacian is known during training. A simple variant of our algorithm can be viewed as an extension of the GCN framework to the inductive setting, a point which we revisit in Section 3.3.

## 3 Proposed method: GraphSAGE

The key idea behind our approach is that we learn how to aggregate feature information from a node's local neighborhood (e.g., the degrees or text attributes of nearby nodes). We first describe the GraphSAGE embedding generation (i.e., forward propagation) algorithm, which generates embeddings for nodes assuming that the GraphSAGE model parameters are already learned (Section 3.1). We then describe how the GraphSAGE model parameters can be learned using standard stochastic gradient descent and backpropagation techniques (Section 3.2).

### 3.1 Embedding generation (i.e., forward propagation) algorithm

In this section, we describe the embedding generation, or forward propagation algorithm (Algorithm 1), which assumes that the model has already been trained and that the parameters are fixed. In particular, we assume that we have learned the parameters of $K$ aggregator functions (denoted $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$), which aggregate information from node neighbors, as well as a set of weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$, which are used to propagate information between different layers of the model or "search depths". Section 3.2 describes how we train these parameters.

[2]In the time between this papers original submission to NIPS 2017 and the submission of the final, accepted (i.e., "camera-ready") version, there have been a number of closely related (e.g., follow-up) works published on pre-print servers. For temporal clarity, we do not review or compare against these papers in detail.

**Algorithm 1:** GraphSAGE embedding generation (i.e., forward propagation) algorithm

```
Input  : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth $K$; weight matrices
         $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity $\sigma$; differentiable aggregator functions
         $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \to 2^{\mathcal{V}}$
Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$
1 $\mathbf{h}^0_v \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;
2 for $k = 1...K$ do
3     for $v \in \mathcal{V}$ do
4         $\mathbf{h}^k_{\mathcal{N}(v)} \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}^{k-1}_u, \forall u \in \mathcal{N}(v)\})$;
5         $\mathbf{h}^k_v \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}^{k-1}_v, \mathbf{h}^k_{\mathcal{N}(v)})\right)$
6     end
7     $\mathbf{h}^k_v \leftarrow \mathbf{h}^k_v / \|\mathbf{h}^k_v\|_2, \forall v \in \mathcal{V}$
8 end
9 $\mathbf{z}_v \leftarrow \mathbf{h}^K_v, \forall v \in \mathcal{V}$
```

The intuition behind Algorithm 1 is that at each iteration, or search depth, nodes aggregate information from their local neighbors, and as this process iterates, nodes incrementally gain more and more information from further reaches of the graph.

Algorithm 1 describes the embedding generation process in the case where the entire graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and features for all nodes $\mathbf{x}_v, \forall v \in \mathcal{V}$, are provided as input. We describe how to generalize this to the minibatch setting below. Each step in the outer loop of Algorithm 1 proceeds as follows, where $k$ denotes the current step in the outer loop (or the depth of the search) and $\mathbf{h}^k$ denotes a node's representation at this step: First, each node $v \in \mathcal{V}$ aggregates the representations of the nodes in its immediate neighborhood, $\{\mathbf{h}^{k-1}_u, \forall u \in \mathcal{N}(v)\}$, into a single vector $\mathbf{h}^{k-1}_{\mathcal{N}(v)}$. Note that this aggregation step depends on the representations generated at the previous iteration of the outer loop (i.e., $k - 1$), and the $k = 0$ ("base case") representations are defined as the input node features. After aggregating the neighboring feature vectors, GraphSAGE then concatenates the node's current representation, $\mathbf{h}^{k-1}_v$, with the aggregated neighborhood vector, $\mathbf{h}^{k-1}_{\mathcal{N}(v)}$, and this concatenated vector is fed through a fully connected layer with nonlinear activation function $\sigma$, which transforms the representations to be used at the next step of the algorithm (i.e., $\mathbf{h}^k_v, \forall v \in \mathcal{V}$). For notational convenience, we denote the final representations output at depth $K$ as $\mathbf{z}_v \equiv \mathbf{h}^K_v, \forall v \in \mathcal{V}$. The aggregation of the neighbor representations can be done by a variety of aggregator architectures (denoted by the AGGREGATE placeholder in Algorithm 1), and we discuss different architecture choices in Section 3.3 below.

To extend Algorithm 1 to the minibatch setting, given a set of input nodes, we first forward sample the required neighborhood sets (up to depth $K$) and then we run the inner loop (line 3 in Algorithm 1), but instead of iterating over all nodes, we compute only the representations that are necessary to satisfy the recursion at each depth (Appendix **??** contains complete minibatch pseudocode).

**Relation to the Weisfeiler-Lehman Isomorphism Test**. The GraphSAGE algorithm is conceptually inspired by a classic algorithm for testing graph isomorphism. If, in Algorithm 1, we (i) set $K = |\mathcal{V}|$, (ii) set the weight matrices as the identity, and (iii) use an appropriate hash function as an aggregator (with no non-linearity), then Algorithm 1 is an instance of the Weisfeiler-Lehman (WL) isomorphism test, also known as "naive vertex refinement" [32]. If the set of representations $\{\mathbf{z}_v, \forall v \in \mathcal{V}\}$ output by Algorithm 1 for two subgraphs are identical then the WL test declares the two subgraphs to be isomorphic. This test is known to fail in some cases, but is valid for a broad class of graphs [32]. GraphSAGE is a continuous approximation to the WL test, where we replace the hash function with trainable neural network aggregators. Of course, we use GraphSAGE to generate useful node representations–not to test graph isomorphism. Nevertheless, the connection between GraphSAGE and the classic WL test provides theoretical context for our algorithm design to learn the topological structure of node neighborhoods.

**Neighborhood definition**. In this work, we uniformly sample a fixed-size set of neighbors, instead of using full neighborhood sets in Algorithm 1, in order to keep the computational footprint of each batch fixed.[3] That is, using overloaded notation, we define $\mathcal{N}(v)$ as a fixed-size, uniform draw from the set $\{u \in \mathcal{V} : (u, v) \in \mathcal{E}\}$, and we draw different uniform samples at each iteration, $k$, in Algorithm 1. Without this sampling the memory and expected runtime of a single batch is unpredictable and in the worst case $O(|\mathcal{V}|)$. In contrast, the per-batch space and time complexity for GraphSAGE is fixed at $O(\prod_{i=1}^{K} S_i)$, where $S_i, i \in \{1, ..., K\}$ and $K$ are user-specified constants. Practically speaking we found that our approach could achieve high performance with $K = 2$ and $S_1 \cdot S_2 \leq 500$ (see Section 4.4 for details).

### 3.2 Learning the parameters of GraphSAGE

In order to learn useful, predictive representations in a fully unsupervised setting, we apply a graph-based loss function to the output representations, $\mathbf{z}_u, \forall u \in \mathcal{V}$, and tune the weight matrices, $\mathbf{W}^k, \forall k \in \{1, ..., K\}$, and parameters of the aggregator functions via stochastic gradient descent. The graph-based loss function encourages nearby nodes to have similar representations, while enforcing that the representations of disparate nodes are highly distinct:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right), \tag{1}$$

where $v$ is a node that co-occurs near $u$ on fixed-length random walk, $\sigma$ is the sigmoid function, $P_n$ is a negative sampling distribution, and $Q$ defines the number of negative samples. Importantly, unlike previous embedding approaches, the representations $\mathbf{z}_u$ that we feed into this loss function are generated from the features contained within a node's local neighborhood, rather than training a unique embedding for each node (via an embedding look-up).

This unsupervised setting emulates situations where node features are provided to downstream machine learning applications, as a service or in a static repository. In cases where representations are to be used only on a specific downstream task, the unsupervised loss (Equation 1) can simply be replaced, or augmented, by a task-specific objective (e.g., cross-entropy loss).

### 3.3 Aggregator Architectures

Unlike machine learning over N-D lattices (e.g., sentences, images, or 3-D volumes), a node's neighbors have no natural ordering; thus, the aggregator functions in Algorithm 1 must operate over an unordered set of vectors. Ideally, an aggregator function would be symmetric (i.e., invariant to permutations of its inputs) while still being trainable and maintaining high representational capacity. The symmetry property of the aggregation function ensures that our neural network model can be trained and applied to arbitrarily ordered node neighborhood feature sets. We examined three candidate aggregator functions:

**Mean aggregator**. Our first candidate aggregator function is the mean operator, where we simply take the elementwise mean of the vectors in $\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}$. The mean aggregator is nearly equivalent to the convolutional propagation rule used in the transductive GCN framework [17]. In particular, we can derive an inductive variant of the GCN approach by replacing lines 4 and 5 in Algorithm 1 with the following:[4]

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}). \tag{2}$$

We call this modified mean-based aggregator *convolutional* since it is a rough, linear approximation of a localized spectral convolution [17]. An important distinction between this convolutional aggregator and our other proposed aggregators is that it does not perform the concatenation operation in line 5 of Algorithm 1—i.e., the convolutional aggregator does concatenate the node's previous layer representation $\mathbf{h}_v^{k-1}$ with the aggregated neighborhood vector $\mathbf{h}_{\mathcal{N}(v)}^k$. This concatenation can be viewed as a simple form of a "skip connection" [13] between the different "search depths", or "layers" of the GraphSAGE algorithm, and it leads to significant gains in performance (Section 4).

**LSTM aggregator**. We also examined a more complex aggregator based on an LSTM architecture [14]. Compared to the mean aggregator, LSTMs have the advantage of larger expressive capability. However, it is important to note that LSTMs are not inherently symmetric (i.e., they are not permutation invariant), since they process their inputs in a sequential manner. We adapt LSTMs to operate on an unordered set by simply applying the LSTMs to a random permutation of the node's neighbors.

[3]Exploring non-uniform samplers is an important direction for future work.

[4]Note that this differs from Kipf et al's exact equation by a minor normalization constant [17].

**Pooling aggregator**. The final aggregator we examine is both symmetric and trainable. In this *pooling* approach, each neighbor's vector is independently fed through a fully-connected neural network; following this transformation, an elementwise max-pooling operation is applied to aggregate information across the neighbor set:

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\}), \tag{3}$$

where $\max$ denotes the element-wise max operator and $\sigma$ is a nonlinear activation function. In principle, the function applied before the max pooling can be an arbitrarily deep multi-layer perceptron, but we focus on simple single-layer architectures in this work. This approach is inspired by recent advancements in applying neural network architectures to learn over general point sets [29]. Intuitively, the multi-layer perceptron can be thought of as a set of functions that compute features for each of the node representations in the neighbor set. By applying the max-pooling operator to each of the computed features, the model effectively captures different aspects of the neighborhood set. Note also that, in principle, any symmetric vector function could be used in place of the $\max$ operator (e.g., an element-wise mean). We found no significant difference between max- and mean-pooling in developments test and thus focused on max-pooling for the rest of our experiments.

## 4 Experiments

We test the performance of GraphSAGE on three benchmark tasks: (i) classifying academic papers into different subjects using the Web of Science citation dataset, (ii) classifying Reddit posts as belonging to different communities, and (iii) classifying protein functions across various biological protein-protein interaction (PPI) graphs. Sections 4.1 and 4.2 summarize the datasets, and the supplementary material contains additional information. In all these experiments, we perform predictions on nodes that are not seen during training, and, in the case of the PPI dataset, we test on entirely unseen graphs.

**Experimental set-up**. To contextualize the empirical results on our inductive benchmarks, we compare against four baselines: a random classifer, a logistic regression feature-based classifier (that ignores graph structure), the DeepWalk algorithm [28] as a representative factorization-based approach, and a concatenation of the raw features and DeepWalk embeddings. We also compare four variants of GraphSAGE that use the different aggregator functions (Section 3.3). Since, the "convolutional" variant of GraphSAGE is an extended, inductive version of Kipf et al's semi-supervised GCN [17], we term this variant GraphSAGE-GCN. We test unsupervised variants of GraphSAGE trained according to the loss in Equation (1), as well as supervised variants that are trained directly on classification cross-entropy loss. For all the GraphSAGE variants we used rectified linear units as the non-linearity and set $K = 2$ with neighborhood sample sizes $S_1 = 25$ and $S_2 = 10$ (see Section 4.4 for sensitivity analyses).

For the Reddit and citation datasets, we use "online" training for DeepWalk as described in Perozzi et al. [28], where we run a new round of SGD optimization to embed the new test nodes before making predictions (see the Appendix for details). In the multi-graph setting, we cannot apply DeepWalk, since the embedding spaces generated by running the DeepWalk algorithm on different disjoint graphs can be arbitrarily rotated with respect to each other (Appendix **??**).

All models were implemented in TensorFlow [1] with the Adam optimizer [16] (except DeepWalk, which performed better with the vanilla gradient descent optimizer). We designed our experiments with the goals of (i) verifying the improvement of GraphSAGE over the baseline approaches (i.e., raw features and DeepWalk) and (ii) providing a rigorous comparison of the different GraphSAGE aggregator architectures. In order to provide a fair comparison, all models share an identical implementation of their minibatch iterators, loss function and neighborhood sampler (when applicable). Moreover, in order to guard against unintentional "hyperparameter hacking" in the comparisons between GraphSAGE aggregators, we sweep over the same set of hyperparameters for all GraphSAGE variants (choosing the best setting for each variant according to performance on a validation set). The set of possible hyperparameter values was determined on early validation tests using subsets of the citation and Reddit data that we then discarded from our analyses. The appendix contains further implementation details.[5]

[5]Code and links to the datasets: `http://snap.stanford.edu/graphsage/`

Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

| | Citation | | Reddit | | PPI | |
|---|---|---|---|---|---|---|
| Name | Unsup. F1 | Sup. F1 | Unsup. F1 | Sup. F1 | Unsup. F1 | Sup. F1 |
| Random | 0.206 | 0.206 | 0.043 | 0.042 | 0.396 | 0.396 |
| Raw features | 0.575 | 0.575 | 0.585 | 0.585 | 0.422 | 0.422 |
| DeepWalk | 0.565 | 0.565 | 0.324 | 0.324 | — | — |
| DeepWalk + features | 0.701 | 0.701 | 0.691 | 0.691 | — | — |
| GraphSAGE-GCN | 0.742 | 0.772 | **0.908** | 0.930 | 0.465 | 0.500 |
| GraphSAGE-mean | 0.778 | 0.820 | 0.897 | 0.950 | 0.486 | 0.598 |
| GraphSAGE-LSTM | 0.788 | 0.832 | **0.907** | **0.954** | 0.482 | **0.612** |
| GraphSAGE-pool | **0.798** | **0.839** | 0.892 | 0.948 | **0.502** | 0.600 |
| % gain over feat. | 39% | 46% | 55% | 63% | 19% | 45% |

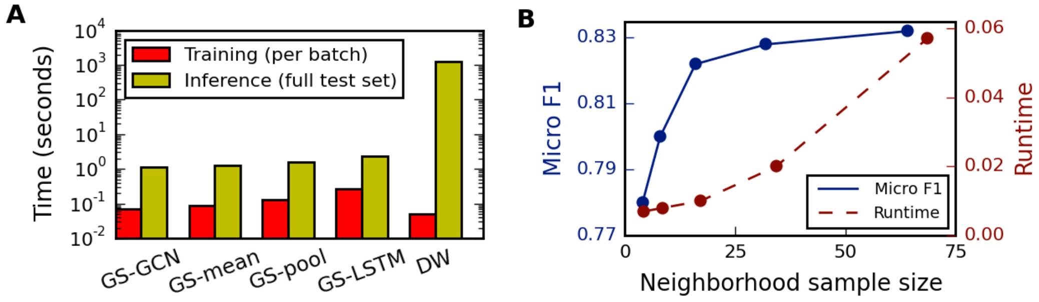

Figure 2: **A**: Timing experiments on Reddit data, with training batches of size 512 and inference on the full test set (79,534 nodes). **B**: Model performance with respect to the size of the sampled neighborhood, where the "neighborhood sample size" refers to the number of neighbors sampled at each depth for $K = 2$ with $S_1 = S_2$ (on the citation data using GraphSAGE-mean).

### 4.1 Inductive learning on evolving graphs: Citation and Reddit data

Our first two experiments are on classifying nodes in evolving information graphs, a task that is especially relevant to high-throughput production systems, which constantly encounter unseen data.

**Citation data**. Our first task is predicting paper subject categories on a large citation dataset. We use an undirected citation graph dataset derived from the Thomson Reuters Web of Science Core Collection, corresponding to all papers in six biology-related fields for the years 2000-2005. The node labels for this dataset correspond to the six different field labels. In total, this is dataset contains 302,424 nodes with an average degree of 9.15. We train all the algorithms on the 2000-2004 data and use the 2005 data for testing (with 30% used for validation). For features, we used node degrees and processed the paper abstracts according Arora et al.'s [2] sentence embedding approach, with 300-dimensional word vectors trained using the GenSim word2vec implementation [30].

**Reddit data**. In our second task, we predict which community different Reddit posts belong to. Reddit is a large online discussion forum where users post and comment on content in different topical communities. We constructed a graph dataset from Reddit posts made in the month of September, 2014. The node label in this case is the community, or "subreddit", that a post belongs to. We sampled 50 large communities and built a post-to-post graph, connecting posts if the same user comments on both. In total this dataset contains 232,965 posts with an average degree of 492. We use the first 20 days for training and the remaining days for testing (with 30% used for validation). For features, we use off-the-shelf 300-dimensional GloVe CommonCrawl word vectors [27]; for each post, we concatenated (i) the average embedding of the post title, (ii) the average embedding of all the post's comments (iii) the post's score, and (iv) the number of comments made on the post.

The first four columns of Table 1 summarize the performance of GraphSAGE as well as the baseline approaches on these two datasets. We find that GraphSAGE outperforms all the baselines by a significant margin, and the trainable, neural network aggregators provide significant gains compared to the GCN approach. For example, the unsupervised variant GraphSAGE-pool outperforms the concatenation of the DeepWalk embeddings and the raw features by 13.8% on the citation data and 29.1% on the Reddit data, while the supervised version provides a gain of 19.7% and 37.2%, respectively. Interestingly, the LSTM based aggregator shows strong performance, despite the fact that it is designed for sequential data and not unordered sets. Lastly, we see that the performance of unsupervised GraphSAGE is reasonably competitive with the fully supervised version, indicating that our framework can achieve strong performance without task-specific fine-tuning.

### 4.2 Generalizing across graphs: Protein-protein interactions

We now consider the task of generalizing across graphs, which requires learning about node roles rather than community structure. We classify protein roles—in terms of their cellular functions from gene ontology—in various protein-protein interaction (PPI) graphs, with each graph corresponding to a different human tissue [41]. We use positional gene sets, motif gene sets and immunological signatures as features and gene ontology sets as labels (121 in total), collected from the Molecular Signatures Database [34]. The average graph contains 2373 nodes, with an average degree of 28.8. We train all algorithms on 20 graphs and then average prediction F1 scores on two test graphs (with two other graphs used for validation).

The final two columns of Table 1 summarize the accuracies of the various approaches on this data. Again we see that GraphSAGE significantly outperforms the baseline approaches, with the LSTM- and pooling-based aggregators providing substantial gains over the mean- and GCN-based aggregators.[6]

### 4.3 Runtime and parameter sensitivity

Figure 2.A summarizes the training and test runtimes for the different approaches. The training time for the methods are comparable (with GraphSAGE-LSTM being the slowest). However, the need to sample new random walks and run new rounds of SGD to embed unseen nodes makes DeepWalk 100-500$\times$ slower at test time.

For the GraphSAGE variants, we found that setting $K = 2$ provided a consistent boost in accuracy of around 10-15%, on average, compared to $K = 1$; however, increasing $K$ beyond 2 gave marginal returns in performance (0-5%) while increasing the runtime by a prohibitively large factor of 10-100$\times$, depending on the neighborhood sample size. We also found diminishing returns for sampling large neighborhoods (Figure 2.B). Thus, despite the higher variance induced by sub-sampling neighborhoods, GraphSAGE is still able to maintain strong predictive accuracy, while significantly improving the runtime.

### 4.4 Summary comparison between the different aggregator architectures

Overall, we found that the LSTM- and pool-based aggregators performed the best, in terms of both average performance and number of experimental settings where they were the top-performing method (Table 1). To give more quantitative insight into these trends, we consider each of the six different experimental settings (i.e., (3 datasets) $\times$ (unsupervised vs. supervised)) as trials and consider what performance trends are likely to generalize. In particular, we use the non-parametric Wilcoxon Signed-Rank Test [33] to quantify the differences between the different aggregators across trials, reporting the $T$-statistic and $p$-value where applicable. Note that this method is rank-based and essentially tests whether we would expect one particular approach to outperform another in a new experimental setting. Given our small sample size of only 6 different settings, this significance test is somewhat underpowered; nonetheless, the $T$-statistic and associated $p$-values are useful quantitative measures to assess the aggregators' relative performances.

We see that LSTM-, pool- and mean-based aggregators all provide statistically significant gains over the GCN-based approach ($T = 1.0$, $p = 0.02$ for all three). However, the gains of the LSTM and pool approaches over the mean-based aggregator are more marginal ($T = 1.5$, $p = 0.03$, comparing

[6]Note that in very recent follow-up work Chen and Zhu [6] achieve superior performance by optimizing the GraphSAGE hyperparameters specifically for the PPI task and implementing new training techniques (e.g., dropout, layer normalization, and a new sampling scheme). We refer the reader to their work for the current state-of-the-art numbers on the PPI dataset that are possible using a variant of the GraphSAGE approach.

LSTM to mean; $T = 4.5$, $p = 0.10$, comparing pool to mean). There is no significant difference between the LSTM and pool approaches ($T = 10.0$, $p = 0.46$). However, GraphSAGE-LSTM is significantly slower than GraphSAGE-pool (by a factor of $\approx 2\times$), perhaps giving the pooling-based aggregator a slight edge overall.

## 5 Theoretical analysis

In this section, we probe the expressive capabilities of GraphSAGE in order to provide insight into how GraphSAGE can learn about graph structure, even though it is inherently based on features. As a case-study, we consider whether GraphSAGE can learn to predict the clustering coefficient of a node, i.e., the proportion of triangles that are closed within the node's 1-hop neighborhood [38]. The clustering coefficient is a popular measure of how clustered a node's local neighborhood is, and it serves as a building block for many more complicated structural motifs [3]. We can show that Algorithm 1 is capable of approximating clustering coefficients to an arbitrary degree of precision:

**Theorem 1.** *Let $\mathbf{x}_v \in U, \forall v \in \mathcal{V}$ denote the feature inputs for Algorithm 1 on graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $U$ is any compact subset of $\mathbb{R}^d$. Suppose that there exists a fixed positive constant $C \in \mathbb{R}^+$ such that $\|\mathbf{x}_v - \mathbf{x}_{v'}\|_2 > C$ for all pairs of nodes. Then we have that $\forall \epsilon > 0$ there exists a parameter setting $\mathbf{\Theta}^*$ for Algorithm 1 such that after $K = 4$ iterations*

$$|z_v - c_v| < \epsilon, \forall v \in \mathcal{V},$$

*where $z_v \in \mathbb{R}$ are final output values generated by Algorithm 1 and $c_v$ are node clustering coefficients.*

Theorem 1 states that for any graph there exists a parameter setting for Algorithm 1 such that it can approximate clustering coefficients in that graph to an arbitrary precision, if the features for every node are distinct (and if the model is sufficiently high-dimensional). The full proof of Theorem 1 is in the Appendix. Note that as a corollary of Theorem 1, GraphSAGE can learn about local graph structure, even when the node feature inputs are sampled from an absolutely continuous random distribution (see the Appendix for details). The basic idea behind the proof is that if each node has a unique feature representation, then we can learn to map nodes to indicator vectors and identify node neighborhoods. The proof of Theorem 1 relies on some properties of the pooling aggregator, which also provides insight into why GraphSAGE-pool outperforms the GCN and mean-based aggregators.

## 6 Conclusion

We introduced a novel approach that allows embeddings to be efficiently generated for unseen nodes. GraphSAGE consistently outperforms state-of-the-art baselines, effectively trades off performance and runtime by sampling node neighborhoods, and our theoretical analysis provides insight into how our approach can learn about local graph structures. A number of extensions and potential improvements are possible, such as extending GraphSAGE to incorporate directed or multi-modal graphs. A particularly interesting direction for future work is exploring non-uniform neighborhood sampling functions, and perhaps even learning these functions as part of the GraphSAGE optimization.

**Acknowledgments**

The authors thank Austin Benson, Aditya Grover, Bryan He, Dan Jurafsky, Alex Ratner, Marinka Zitnik, and Daniel Selsam for their helpful discussions and comments on early drafts. The authors would also like to thank Ben Johnson for his many useful questions and comments on our code. This research has been supported in part by NSF IIS-1149837, DARPA SIMPLEX, Stanford Data Science Initiative, Huawei, and Chan Zuckerberg Biohub. W.L.H. was also supported by the SAP Stanford Graduate Fellowship and an NSERC PGS-D grant. The views and conclusions expressed in this material are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above funding agencies, corporations, or the U.S. and Canadian governments.

## References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint* , 2016.

[2] S. Arora, Y. Liang, and T. Ma. A simple but tough-to-beat baseline for sentence embeddings. In *ICLR*, 2017.

[3] A. R. Benson, D. F. Gleich, and J. Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

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

[5] S. Cao, W. Lu, and Q. Xu. Grarep: Learning graph representations with global structural information. In *KDD*, 2015.

[6] J. Chen and J. Zhu. Stochastic training of graph convolutional networks. *arXiv preprint arXiv:1710.10568*, 2017.

[7] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *ICML*, 2016.

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

[9] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, 2015.

[10] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks*, volume 2, pages 729–734, 2005.

[11] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, 2016.

[12] W. L. Hamilton, J. Leskovec, and D. Jurafsky. Diachronic word embeddings reveal statistical laws of semantic change. In *ACL*, 2016.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *EACV*, 2016.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

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

[16] D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[17] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2016.

[18] T. N. Kipf and M. Welling. Variational graph auto-encoders. In *NIPS Workshop on Bayesian Deep Learning*, 2016.

[19] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[20] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. In *NIPS*, 2014.

[21] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *ICLR*, 2015.

[22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[23] A. Y. Ng, M. I. Jordan, Y. Weiss, et al. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2001.

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

[25] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[28] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *KDD*, 2014.

[29] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*, 2017.

[30] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *LREC*, 2010.

[31] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[32] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[33] S. Siegal. *Nonparametric statistics for the behavioral sciences*. McGraw-hill, 1956.

[34] A. Subramanian, P. Tamayo, V. K. Mootha, S. Mukherjee, B. L. Ebert, M. A. Gillette, A. Paulovich, S. L. Pomeroy, T. R. Golub, E. S. Lander, et al. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102(43):15545–15550, 2005.

[35] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. Line: Large-scale information network embedding. In *WWW*, 2015.

[36] D. Wang, P. Cui, and W. Zhu. Structural deep network embedding. In *KDD*, 2016.

[37] X. Wang, P. Cui, J. Wang, J. Pei, W. Zhu, and S. Yang. Community preserving network embedding. In *AAAI*, 2017.

[38] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[39] L. Xu, X. Wei, J. Cao, and P. S. Yu. Embedding identity and interest for social networks. In *WWW*, 2017.

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

[41] M. Zitnik and J. Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):190–198, 2017.

# Appendices

## A Minibatch pseudocode

In order to use stochastic gradient descent, we adapt our algorithm to allow forward and backward propagation for minibatches of nodes and edges. Here we focus on the minibatch forward propagation algorithm, analogous to Algorithm 1. In the forward propagation of GraphSAGE the minibatch $\mathcal{B}$ contains nodes that we want to generate representations for. Algorithm 2 gives the pseudocode for the minibatch approach.

**Algorithm 2:** GraphSAGE minibatch forward propagation algorithm

```
Input  : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$;
         input features $\{\mathbf{x}_v, \forall v \in \mathcal{B}\}$;
         depth $K$; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$;
         non-linearity $\sigma$;
         differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$;
         neighborhood sampling functions, $\mathcal{N}_k : v \rightarrow 2^{\mathcal{V}}, \forall k \in \{1, ..., K\}$
Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{B}$
1  $\mathcal{B}^K \leftarrow \mathcal{B}$;
2  for $k = K...1$ do
3      $B^{k-1} \leftarrow \mathcal{B}^k$ ;
4      for $u \in \mathcal{B}^k$ do
5          $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$;
6      end
7  end
8  $\mathbf{h}^0_u \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$ ;
9  for $k = 1...K$ do
10     for $u \in \mathcal{B}^k$ do
11         $\mathbf{h}^k_{\mathcal{N}(u)} \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}^{k-1}_{u'}, \forall u' \in \mathcal{N}_k(u)\})$;
12         $\mathbf{h}^k_u \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}^{k-1}_u, \mathbf{h}^k_{\mathcal{N}(u)})\right)$;
13         $\mathbf{h}^k_u \leftarrow \mathbf{h}^k_u / \|\mathbf{h}^k_u\|_2$;
14     end
15 end
16 $\mathbf{z}_u \leftarrow \mathbf{h}^K_u, \forall u \in \mathcal{B}$
```

The main idea is to sample all the nodes needed for the computation first. Lines 2-7 of Algorithm 2 correspond to the sampling stage. Each set $\mathcal{B}^k$ contains the nodes that are needed to compute the representations of nodes $v \in \mathcal{B}^{k+1}$, i.e., the nodes in the $(k+1)$-st iteration, or "layer", of Algorithm 1. Lines 9-15 correspond to the aggregation stage, which is almost identical to the batch inference algorithm. Note that in Lines 12 and 13, the representation at iteration $k$ of any node in set $\mathcal{B}^k$ can be computed, because its representation at iteration $k-1$ and the representations of its sampled neighbors at iteration $k-1$ have already been computed in the previous loop. The algorithm thus avoids computing the representations for nodes that are not in the current minibatch and not used during the current iteration of stochastic gradient descent. We use the notation $\mathcal{N}_k(u)$ to denote a deterministic function which specifies a random sample of a node's neighborhood (i.e., the randomness is assumed to be pre-computed in the mappings). We index this function by $k$ to denote the fact that the random samples are independent across iterations over $k$. We use a uniform sampling function in this work and sample with replacement in cases where the sample size is larger than the node's degree.

Note that the sampling process in Algorithm 2 is conceptually reversed compared to the iterations over $k$ in Algorithm 1: we start with the "layer-K" nodes (i.e., the nodes in $\mathcal{B}$) that we want to generate representations for; then we sample their neighbors (i.e., the nodes at "layer-K-1" of the algorithm) and so on. One consequence of this is that the definition of neighborhood sampling sizes can be somewhat counterintuitive. In particular, if we use $K = 2$ total iterations with sample sizes $S_1$

and $S_2$ then this means that we sample $S_1$ nodes during iteration $k = 1$ of Algorithm 1 and $S_2$ nodes during iteration $k = 2$, and—from the perspective of the "target" nodes in $\mathcal{B}$ that we want to generate representations for after iteration $k = 2$—this amounts to sampling $S_2$ of their immediate neighbors and $S_1 \cdot S_2$ of their 2-hop neighbors.

## B Additional Dataset Details

In this section, we provide some additional, relevant dataset details. The full PPI and Reddit datasets are available at: `http://snap.stanford.edu/graphsage/`. The Web of Science dataset (WoS) is licensed by Thomson Reuters and can be made available to groups with valid WoS licenses.

**Reddit data** To sample communities, we ranked communities by their total number of comments in 2014 and selected the communities with ranks [11,50] (inclusive). We omitted the largest communities because they are large, generic default communities that substantially skew the class distribution. We selected the largest connected component of the graph defined over the union of these communities. We performed early validation experiments and model development on data from October and November, 2014.

Details on the source of the Reddit data are at: `https://archive.org/details/FullRedditSubmissionCorpus2006ThruAugust2015` and `https://archive.org/details/2015_reddit_comments_corpus`.

**WoS data** We selected the following subfields manually, based on them being of relatively equal size and all biology-related fields. We performed early validation and model development on the neuroscience subfield (code=RU, which is excluded from our final set). We did not run any experiments on any other subsets of the WoS data. We took the largest connected component of the graph defined over the union of these fields.

- Immunology (code: NI, number of documents: 77356)
- Ecology (code: GU, number of documents: 37935)
- Biophysics (code: DA, number of documents: 36688)
- Endocrinology and Metabolism (code: IA, number of documents: 52225).
- Cell Biology (code: DR, number of documents: 84231)
- Biology (other) (code: CU, number of documents: 13988)

**PPI Tissue Data** For training, we randomly selected 20 PPI networks that had at least 15,000 edges. For testing and validation, we selected 4 large networks (2 for validation, 2 for testing, each with at least 35,000 edges). All experiments for model design and development were performed on the same 2 validation networks, and we used the same random training set in all experiments.

We selected features that included at least 10% of the proteins that appear in any of the PPI graphs. Note that the feature data is very sparse for dataset (42% of nodes have no non-zero feature values), which makes leveraging neighborhood information critical.

## C Details on the Experimental Setup and Hyperparameter Tuning

**Random walks for the unsupervised objective** For all settings, we ran 50 random walks of length 5 from each node in order to obtain the pairs needed for the unsupervised loss (Equation 1). Our implementation of the random walks is in pure Python and is based directly on Python code provided by Perozzi et al. [28].

**Logistic regression model** For the feature only model and to make predictions on the embeddings output from the unsupervised models, we used the logistic SGDClassifier from the scikit-learn Python package [26], with all default settings. Note that this model is always optimized only on the training nodes and it is not fine-tuned on the embeddings that are generated for the test data.

**Hyperparameter selection** In all settings, we performed hyperparameter selection on the learning rate and the model dimension. With the exception of DeepWalk, we performed a parameter sweep on initial learning rates $\{0.01, 0.001, 0.0001\}$ for the supervised models and $\{2 \times 10^{-6}, 2 \times 10^{-7}, 2 \times 10^{-8}\}$ for the unsupervised models.[7] When applicable, we tested a "big" and "small" version of each model, where we tried to keep the overall model sizes comparable. For the pooling aggregator, the "big" model had a pooling dimension of 1024, while the "small" model had a dimension of 512. For the LSTM aggregator, the "big" model had a hidden dimension of 256, while the "small" model had a hidden dimension of 128; note that the actual parameter count for the LSTM is roughly $4\times$ this number, due to weights for the different gates. In all experiments and for all models we specify the output dimension of the $\mathbf{h}_i^k$ vectors at every depth $k$ of the recursion to be $256$. All models use rectified linear units as a non-linear activation function. All the unsupervised GraphSAGE models and DeepWalk used 20 negative samples with context distribution smoothing over node degrees using a smoothing parameter of $0.75$, following [11, 22, 28]. Initial experiments revealed that DeepWalk performed much better with large learning rates, so we swept over rates in the set $\{0.2, 0.4, 0.8\}$. For the supervised GraphSAGE methods, we ran 10 epochs for all models. All methods except DeepWalk use batch sizes of 512. We found that DeepWalk achieved faster wall-clock convergence with a smaller batch size of 64.

**Hardware** Except for DeepWalk, we ran experiments single a machine with 4 NVIDIA Titan X Pascal GPUs (12Gb of RAM at 10Gbps speed), 16 Intel Xeon CPUs (E5-2623 v4 @ 2.60GHz), and 256Gb of RAM. DeepWalk was faster on a CPU intensive machine with 144 Intel Xeon CPUs (E7-8890 v3 @ 2.50GHz) and 2Tb of RAM. Overall, our experiments took about 3 days in a shared resource setting. We expect that a consumer-grade single-GPU machine (e.g., with a Titan X GPU) could complete our full set of experiments in 4-7 days, if its full resources were dedicated.

**Notes on the DeepWalk implementation** Existing DeepWalk implementations [28, 11] are simply wrappers around dedicated word2vec code, and they do not easily support embedding new nodes and other variations. Moreover, this makes it difficult to compare runtimes and other statistics for these approaches. For this reason, we reimplemented DeepWalk in pure TensorFlow, using the vector initializations etc that are described in the TensorFlow word2vec tutorial.[8]

We found that DeepWalk was much slower to converge than the other methods, and since it is 2-5X faster at training, we gave it 5 passes over the random walk data, instead of one. To update the DeepWalk method on new data, we ran 50 random walks of length 5 (as described above) and performed updates on the embeddings for the new nodes while holding the already trained embeddings fixed. We also tested two variants, one where we restricted the sampled random walk "context nodes" to only be from the set of already trained nodes (which alleviates statistical drift) and an approach without this restriction. We always selected the better performing variant. Note that despite DeepWalk's poor performance on the inductive task, it is far more competitive when tested in the transductive setting, where it can be extensively trained on a single, fixed graph. (That said, Kipf et al [17][18] found that GCN-based approach consistently outperformed DeepWalk, even in the transductive setting on link prediction, a task that theoretically favors DeepWalk.) We did observe DeepWalk's performance *could* improve with further training, and in some cases it could become competitive with the unsupervised GraphSAGE approaches (but not the supervised approaches) if we let it run for $>1000\times$ longer than the other approaches (in terms of wall clock time for prediction on the test set); however, we did not deem this to be a meaningful comparison for the inductive task.

Note that DeepWalk is also equivalent to the node2vec model [11] with $p = q = 1$.

**Notes on neighborhood sampling** Due to the heavy-tailed nature of degree distributions we downsample the edges in all graphs before feeding them into the GraphSAGE algorithm. In particular, we subsample edges so that no node has degree larger than 128. Since we only sample at most 25 neighbors per node, this is a reasonable tradeoff. This downsampling allows us to store neighborhood information as dense adjacency lists, which drastically improves computational efficiency. For the Reddit data we also downsampled the edges of the original graph as a pre-processing step, since the

[7]Note that these values differ from our previous reported pre-print values because they are corrected to account for an extraneous normalization by the batch size. We thank Ben Johnson for pointing out this discrepancy.

[8]`https://github.com/tensorflow/models/blob/master/tutorials/embedding/word2vec.py`

original graph is extremely dense. All experiments are on the downsampled version, but we release the full version on the project website for reference.

## D Alignment Issues and Orthogonal Invariance for DeepWalk and Related Approaches

DeepWalk [28], node2vec [11], and other recent successful node embedding approaches employ objective functions of the form:

$$\alpha \sum_{i,j \in \mathcal{A}} f(\mathbf{z}_i^\top \mathbf{z}_j) + \beta \sum_{i,j \in \mathcal{B}} g(\mathbf{z}_i^\top \mathbf{z}_j) \tag{4}$$

where $f$, $g$ are smooth, continuous functions, $\mathbf{z}_i$ are the node representations that are being directly optimized (i.e., via embedding look-ups), and $\mathcal{A}, \mathcal{B}$ are sets of pairs of nodes. Note that in many cases, in the actual code implementations used by the authors of these approaches, nodes are associated with two unique embedding vectors and the arguments to the dot products in $f$ and $g$ are drawn for distinct embedding look-ups (e.g., [11, 28]); however, this does not fundamentally alter the learning algorithm. The majority of approaches also normalize the learned embeddings to unit length, so we assume this post-processing as well.

By connection to word embedding approaches and the arguments of [20], these approaches can also be viewed as stochastic, implicit matrix factorizations where we are trying to learn a matrix $\mathbf{Z} \in \mathbb{R}^{|\mathcal{V}| \times d}$ such that

$$\mathbf{Z}\mathbf{Z}^\top \approx \mathbf{M}, \tag{5}$$

where $\mathbf{M}$ is some matrix containing random walk statistics.

An important consequence of this structure is that the embeddings can be rotated by an arbitrary orthogonal matrix, without impacting the objective:

$$\mathbf{Z}\mathbf{Q}^\top \mathbf{Q}\mathbf{Z}^\top = \mathbf{Z}\mathbf{Z}^\top, \tag{6}$$

where $\mathbf{Q} \in \mathbb{R}^{d \times d}$ is any orthogonal matrix. Since the embeddings are otherwise unconstrained and the only error signal comes from the orthogonally-invariant objective (**??**), the entire embedding space is free to arbitrarily rotate during training.

Two clear consequences of this are:

1. Suppose we run an embedding approach based on (**??**) on two separate graphs A and B using the same output dimension. Without some explicit penalty enforcing alignment, the learned embeddings spaces for the two graphs will be arbitrarily rotated with respect to each other after training. Thus, for any node classification method that is trained on individual embeddings from graph A, inputting the embeddings from graph B will be essentially random. This fact is also simply true by virtue of the fact that the $\mathbf{M}$ matrices of these graphs are completely disjoint. Of course, if we had a way to match "similar" nodes between the graphs, then it could be possible to use an alignment procedure to share information between the graphs, such as the procedure proposed by [12] for aligning the output of word embedding algorithms. Investigating such alignment procedures is an interesting direction for future work; though these approaches will inevitably be slow run on new data, compared to approaches like GraphSAGE that can simply generate embeddings for new nodes without any additional training or alignment.
2. Suppose that we run an embedding approach based on (**??**) on graph C at time $t$ and train a classifier on the learned embeddings. Then at time $t+1$ we add more nodes to C and run a new round of SGD and update all embeddings. Two issues arise: First by analogy to point 1 above, if the new nodes are only connected to a very small number of the old nodes, then the embedding space for the new nodes can essentially become rotated with respect to the original embedding space. Moreover, if we update all embeddings during training (not just for the new nodes), as suggested by [28]'s streaming approach to DeepWalk, then the embedding space can arbitrarily rotate compared to the embedding space that we trained our classifier on, which only further exasperates the problem.

Note that this rotational invariance is not problematic for tasks that only rely on pairwise node distances (e.g., link prediction via dot products). Moreover, some reasonable approaches to alleviate this issue of statistical drift are to (1) not update the already trained embeddings when optimizing the embeddings for new test nodes and (2) to only keep existing nodes as "context nodes" in the sampled random walks, i.e. to ensure that every dot-product in the skip-gram objective is the product of an already-trained node and a new/test node. We tried both of these approaches in this work and always selected the best performing DeepWalk variant.

Also note that empirically DeepWalk performs better on the citation data than the Reddit data (Section 4.1) because this statistical drift is worse in the Reddit data, compared to the citation graph. In particular, the Reddit data has fewer edges from the test set to the train set, which help prevent mis-alignment: 96% of the 2005 citation links connect back to the 2000-2004 data, while only 73% of edges in the Reddit test set connect back to the train data.

## E Proof of Theorem 1

To prove Theorem 1, we first prove three lemmas:

- Lemma 1 states that there exists a continuous function that is guaranteed to only be positive in closed balls around a fixed number of points, with some noise tolerance.
- Lemma 2 notes that we can approximate the function in Lemma 1 to an arbitrary precision using a multilayer perceptron with a single hidden layer.
- Lemma 3 builds off the preceding two lemmas to prove that the pooling architecture can learn to map nodes to unique indicator vectors, assuming that all the input feature vectors are sufficiently distinct.

We also rely on fact that the max-pooling operator (with at least one hidden layer) is capable of approximating any Hausdorff continuous, symmetric function to an arbitrary $\epsilon$ precision [29].

We note that all of the following are essentially *identifiability* arguments. We show that there exists a parameter setting for which Algorithm 1 can learn nodes clustering coefficients, which is non-obvious given that it operates by aggregating feature information. The *efficient learnability* of the functions described is the subject of future work. We also note that these proofs are conservative in the sense that clustering coefficients may be in fact identifiable in fewer iterations, or with less restrictions, than we impose. Moreover, due to our reliance on two universal approximation theorems [15, 29], the required dimensionality is in principle $O(|\mathcal{V}|)$. We can provide a more informative bound on the required output dimension of some particular layers (e..g., Lemma 3); however, in the worst case this identifiability argument relies on having a dimension of $O(|\mathcal{V}|)$. It is worth noting, however, that Kipf et al's "featureless" GCN approach has parameter dimension $O(|\mathcal{V}|)$, so this requirement is not entirely unreasonable [17, 18].

Following Theorem 1, we let $\mathbf{x}_v \in U, \forall v \in \mathcal{V}$ denote the feature inputs for Algorithm 1 on graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $U$ is any compact subset of $\mathbb{R}^d$.

**Lemma 1.** *Let $C \in \mathbb{R}^+$ be a fixed positive constant. Then for any non-empty finite subset of nodes $\mathcal{D} \subseteq \mathcal{V}$, there exists a continuous function $g : U \rightarrow \mathbb{R}$ such that*

$$\begin{cases} g(\mathbf{x}) > \epsilon, & \textit{if } \|\mathbf{x} - \mathbf{x}_v\|_2 = 0 \textit{ for some } v \in \mathcal{D} \\ g(\mathbf{x}) \leq -\epsilon, & \textit{if } \|\mathbf{x} - \mathbf{x}_v\|_2 > C, \forall v \in \mathcal{D}, \end{cases} \tag{7}$$

*where $\epsilon < 0.5$ is a chosen error tolerance.*

*Proof.* Many such functions exist. For concreteness, we provide one construction that satisfies these criteria. Let $\mathbf{x} \in U$ denote an arbitrary input to $g$, let $d_v = \|\mathbf{x} - \mathbf{x}_v\|_2, \forall v \in \mathcal{D}$, and let $g$ be defined as $g(\mathbf{x}) = \sum_{v \in \mathcal{D}} g_v(\mathbf{x})$ with

$$g_v(\mathbf{x}) = \frac{3|\mathcal{D}|\epsilon}{bd_v^2 + 1} - 2\epsilon \tag{8}$$

where $b = \frac{3|\mathcal{D}|-1}{C^2} > 0$. By construction:

1. $g_v$ has a unique maximum of $3|\mathcal{D}|\epsilon - 2\epsilon > 2|\mathcal{D}|\epsilon$ at $d_v = 0$.

2. $\lim_{d_v \to \infty} \left( \frac{3|\mathcal{D}|\epsilon}{bd_v^2+1} - 2\epsilon \right) = -2\epsilon$

3. $\frac{3|\mathcal{D}|\epsilon}{bd_v^2+1} - 2\epsilon \leq -\epsilon$ if $d_v \geq C$.

Note also that $g$ is continuous on its domain ($d_v \in \mathbb{R}^+$) since it is the sum of finite set of continuous functions. Moreover, we have that, for a given input $\mathbf{x} \in U$, if $d_v \geq C$ for all points $v \in \mathcal{D}$ then $g(\mathbf{x}) = \sum_{v\in\mathcal{D}} g_v(\mathbf{a}) \leq -\epsilon$ by property 3 above. And, if $d_v = 0$ for any $v \in \mathcal{D}$, then $g$ is positive by construction, by properties 1 and 2, since in this case,

$$\begin{aligned} g_v(\mathbf{x}) + \sum_{v'\in\mathcal{D}\setminus v} g_{v'}(\mathbf{x}) &\geq g_v(\mathbf{x}) - (|\mathcal{D}|-1)2\epsilon \\ &> g_v(\mathbf{x}) - 2(|\mathcal{D}|)\epsilon \\ &> 2(|\mathcal{D}|)\epsilon - 2(|\mathcal{D}|)\epsilon \\ &> 0, \end{aligned}$$

so we know that $g$ is positive whenever $d_v = 0$ for any node and negative whenever $d_v > C$ for all nodes. □

**Lemma 2.** *The function $g : U \to \mathbb{R}$ can be approximated to an arbitrary degree of precision by standard multilayer perceptron (MLP) with least one hidden layer and a non-constant monotonically increasing activation function (e.g., a rectified linear unit). In precise terms, if we let $f_{\theta_\sigma}$ denote this MLP and $\theta_\sigma$ its parameters, we have that $\forall\epsilon$, $\exists\theta_\sigma$ such that $|f_{\theta_\sigma}(\mathbf{x}) - g(\mathbf{x})| < \epsilon|, \forall \mathbf{x} \in U$.*

*Proof.* This is a direct consequence of Theorem 2 in [15]. □

**Lemma 3.** *Let $\mathbf{A}$ be the adjacency matrix of $G$, let $\mathcal{N}^3(v)$ denote the 3-hop neighborhood of a node, $v$, and define $\chi(G^3)$ as the chromatic number of the graph with adjacency matrix $\mathbf{A}^3$ (ignoring self-loops). Suppose that there exists a fixed positive constant $C \in \mathbb{R}^+$ such that $\|\mathbf{x}_v - \mathbf{x}_{v'}\|_2 > C$ for all pairs of nodes. Then we have that there exists a parameter setting for Algorithm 1, using a pooling aggregator at depth $k = 1$, where this pooling aggregator has $\geq 2$ hidden layers with rectified non-linear units, such that*

$$\mathbf{h}_v^1 \neq \mathbf{h}_{v'}^1, \forall (v, v') \in \{(v, v') : \exists u \in \mathcal{V}, v, v' \in \mathcal{N}^3(u)\}, \mathbf{h}_v^1, \mathbf{h}_{v'}^1 \in \mathcal{E}_I^{\chi(\mathcal{G}^3)}$$

*where $\mathcal{E}_I^{\chi(\mathcal{G}^3)}$ is the set of one-hot indicator vectors of dimension $\chi(\mathcal{G}^3)$.*

*Proof.* By the definition of the chromatic number, we know that we can label every node in $\mathcal{V}$ using $\chi(\mathcal{G}^3)$ unique colors, such that no two nodes that co-occur in any node's 3-hop neighborhood are assigned the same color. Thus, with exactly $\chi(\mathcal{G}^3)$ dimensions we can assign a unique one-hot indicator vector to every node, where no two nodes that co-occur in any 3-hop neighborhood have the same vector. In other words, each color defines a subset of nodes $\mathcal{D} \subseteq \mathcal{V}$ and this subset of nodes can all be mapped to the same indicator vector without introducing conflicts.

By Lemma 1 and 2 and the assumption that $\|\mathbf{x}_v - \mathbf{x}_{v'}\|_2 > C$ for all pairs of nodes, we can choose an $\epsilon < 0.5$ and there exists a single-layer MLP, $f_{\theta_\sigma}$, such that for any subset of nodes $\mathcal{D} \subseteq \mathcal{V}$:

$$\begin{cases} f_{\theta_\sigma}(\mathbf{x}_v) > 0, & \forall v \in \mathcal{D} \\ f_{\theta_\sigma}(\mathbf{x}_v) < 0, & \forall v \in \mathcal{V} \setminus \mathcal{D}. \end{cases} \tag{9}$$

By making this MLP one layer deeper and specifically using a rectified linear activation function, we can return a positive value only for nodes in the subset $\mathcal{D}$ and zero otherwise, and, since we normalize after applying the aggregator layer, this single positive value can be mapped to an indicator vector. Moreover, we can create $\chi(\mathcal{G}^3)$ such MLPs, where each MLP corresponds to a different color/subset; equivalently each MLP corresponds to a different max-pooling dimension in equation 3 of the main text. □

We now restate Theorem 1 and provide a proof.

**Theorem 1.** *Let $\mathbf{x}_v \in \mathbb{R}^d, \forall v \in \mathcal{V}$ denote the feature inputs for Algorithm 1 on graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $U$ is any compact subset of $\mathbb{R}^d$. Suppose that there exists a fixed positive constant $C \in \mathbb{R}^+$ such that $\|\mathbf{x}_v - \mathbf{x}_{v'}\|_2 > C$ for all pairs of nodes. Then we have that $\forall \epsilon > 0$ there exists a parameter setting $\mathbf{\Theta}^*$ for Algorithm 1 such that after $K = 4$ iterations*

$$|z_v - c_v| < \epsilon, \forall v \in \mathcal{V},$$

*where $z_v \in \mathbb{R}$ are final output values generated by Algorithm 1 and $c_v$ are node clustering coefficients, as defined in [38].*

*Proof.* Without loss of generality, we describe how to compute the clustering coefficient for an arbitrary node $v$. For notational convenience we use $\oplus$ to denote vector concatenation and $d_v$ to denote the degree of node $v$. This proof requires 4 iterations of Algorithm 1, where we use the pooling aggregator at all depths. For clarity and we ignore issues related to vector normalization and we use the fact that the pooling aggregator can approximate any Hausdorff continuous function to an arbitrary $\epsilon$ precision [29]. Note that we can always account for normalization constants (line 7 in Algorithm 1) by having aggregators prepend a unit value to all output representations; the normalization constant can then be recovered at later layers by taking the inverse of this prepended value. Note also that almost certainly exist settings where the symmetric functions described below can be computed exactly by the pooling aggregator (or a variant of it), but the symmetric universal approximation theorem of [29] along with Lipschitz continuity arguments suffice for the purposes of proving identifiability of clustering coefficients (up to an arbitrary precision). In particular, the functions described below, that we need approximate to compute clustering coefficients, are all Lipschitz continuous on their domains (assuming we only run on nodes with positive degrees) so the errors introduced by approximation remain bounded by fixed constants (that can be made arbitrarily small).

We assume that the weight matrices, $\mathbf{W}^1, \mathbf{W}^2$ at depths $k = 2$ and $k = 3$ are the identity, and that all non-linearities are rectified linear units. In addition, for the final iteration (i.e, $k = 4$) we completely ignore neighborhood information and simply treat this layers as an MLP with a single hidden layer. Theorem 1 can be equivalently stated as requiring $K = 3$ iterations of Algorithm 1, with the representations then being fed to a single-layer MLP.

By Lemma 3, we can assume that at depth $k = 1$ all nodes in $v$'s 3-hop neighborhood have unique, one-hot indicator vectors, $\mathbf{h}_v^1 \in \mathcal{E}_I$. Thus, at depth $k = 2$ in Algorithm 1, suppose that we sum the unnormalized representations of the neighboring nodes (which is possible by Lemma 4). Then without loss of generality, we will have that $\mathbf{h}_v^2 = \mathbf{h}_v^1 \oplus \mathbf{A}_v$ where $\mathbf{A}$ is the adjacency matrix of the subgraph containing all nodes connected to $v$ in $G^3$ and $\mathbf{A}_v$ is the row of the adjacency matrix corresponding to $v$. Then, at depth $k = 3$, again assume that we sum the neighboring representations (with the weight matrices as the identity), then we will have that

$$\mathbf{h}_v^3 = \mathbf{x}_v \oplus \mathbf{A}_v \oplus \left( \sum_{v \in \mathcal{N}(v)} \mathbf{x}_v \oplus \mathbf{A}_v \right). \tag{10}$$

Letting $m$ denote the dimensionality of the $\mathbf{h}_v^1$ vectors (i.e., $m \equiv \chi(G^3)$ from Lemma 3) and using square brackets to denote vector indexing, we can observe that

- $\mathbf{a} \equiv \mathbf{h}_v^3[0 : m]$ is $v$'s one-hot indicator vector.
- $\mathbf{b} \equiv \mathbf{h}_v^3[m : 2m]$ is $v$'s row in the adjacency matrix, $\mathbf{A}$.
- $\mathbf{c} \equiv \mathbf{h}_v^3[3m : 4m]$ is the sum of the adjacency rows of $v$'s neighbors.

Thus, we have that $\mathbf{b}^\top \mathbf{c}$ is the number of edges in the subgraph containing only $v$ and it's immediate neighbors and $\sum_{i=0}^{m} \mathbf{b}[i] = d_v$. Finally we can compute

$$\frac{2(\mathbf{b}^\top \mathbf{c} - d_v)}{(d_v)(d_v - 1)} = \frac{2|\{e_{v,v'} : v, v' \in \mathcal{N}(v), e_{v,v'} \in \mathcal{E}\}|}{(d_v)(d_v - 1)} \tag{11}$$

$$= c_v, \tag{12}$$

and since this is a continuous function of $\mathbf{h}_v^3$, we can approximate it to an arbitrary $\epsilon$ precision with a single-layer MLP (or equivalently, one more iteration of Algorithm 1, ignoring neighborhood information). Again this last step follows directly from [15]. □

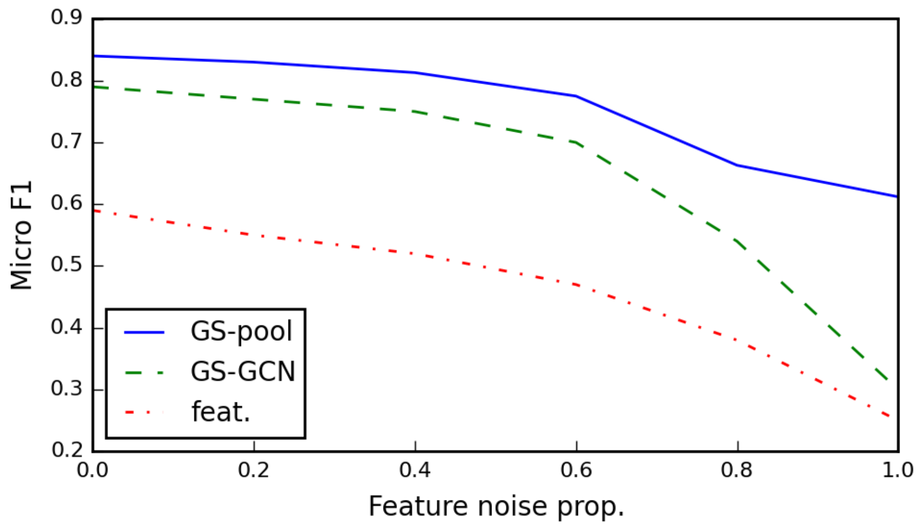

Figure 3: Accuracy (in F1-score) for different approaches on the citation data as the feature matrix is incrementally replaced with random Gaussian noise.

**Corollary 2.** *Suppose we sample nodes features from any probability distribution $\mu$ over $\mathbf{x} \in U$, where $\mu$ is absolutely continuous with respect to the Lebesgue measure. Then the conditions of Theorem 1 are almost surely satisfied with feature inputs $\mathbf{x}_v \sim \mu$.*

Corollary 2 is a direct consequence of Theorem 1 and the fact that, for any probability distribution that is absolutely continuous w.r.t. the Lebesgue measure, the probability of sampling two identical points is zero. Empirically, we found that GraphSAGE-pool was in fact capable of maintaining modest performance by leveraging graph structure, even with completely random feature inputs (see Figure **??**). However, the performance GraphSAGE-GCN was not so robust, which makes intuitive sense given that the Lemmas 1, 2, and 3 rely directly on the universal expressive capability of the pooling aggregator.

Finally, we note that Theorem 1 and Corollary 2 are expressed with respect to a particular given graph and are thus somewhat transductive. For the inductive setting, we can state

**Corollary 3.** *Suppose that for all graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ belonging to some class of graphs $G^*$, we have that $\exists k, d \geq 0, k, d \in \mathbb{Z}$ such that*

$$\mathbf{h}_v^k \neq \mathbf{h}_{v'}^k, \forall (v, v') \in \{(v, v') : \exists u \in \mathcal{V}, v, v' \in \mathcal{N}^3(u)\}, \mathbf{h}_v^k, \mathbf{h}_{v'}^k \in \mathcal{E}_I^d,$$

*then we can approximate clustering coefficients to an arbitrary epsilon after $K = k + 4$ iterations of Algorithm 1.*

Corollary 3 simply states that if after $k$ iterations of Algorithm 1, we can learn to uniquely identify nodes for a class of graphs, then we can also approximate clustering coefficients to an arbitrary precision for this class of graphs.