GNN: A Survey on Architectures and Optimizations

Saksham Goel, Aneesh Shetty, Anubhav Goel
{saksham,aneeshks,anubhavg}@cs.utexas.edu
The University of Texas at Austin

Abstract

Graph Neural Networks have achieved unprecedented success in modeling graph-structured problems. However, they are computationally expensive to train and use in production. Practitioners are forced to compromise on the size of their training graph and the depth & power of their neural network. Consequently, a large number of approximations and optimization techniques have been developed over the last several years to overcome these limitations. This survey is an attempt to systematically classify these techniques, helping practitioners evaluate holistically and apply them in a context-sensitive manner. We compare and contrast key ideas from nine recent representative papers that describe various optimized GNN architectures, neighborhood sampling techniques, and computational models for efficient training and deployment. In particular, this survey is organized as follows:

- Section 1 defines the general node classification problem that GNNs aim to solve. It then describes three popular GNN architectures along with their limitations.
- Section 2 introduces the idea of neighborhood sampling. It categorizes these techniques into four distinct classes, and describes the key ideas in each class through several recent papers.
- Finally, Section 3 describes a novel computational model to improve end-to-end GNN training performance.

1 Architectures

Graph Neural Networks (GNNs) tackle the semi-supervised problem of classifying nodes in an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. This task has two inputs. First is the feature matrix $X \in \mathbb{R}^{N \times C}$, where the row vector X_i denotes the features of the i$^{\text{th}}$ node. C is the number of features per node, and $N = |\mathcal{V}|$ is the number of nodes in the graph. The second input is the true labels of only *some* nodes, where the label for the j^{th} node is represented as another vector $Z_j \in \mathbb{R}^B$. The goal is to learn the labels of *all* nodes, i.e., $Z \in \mathbb{R}^{N \times B}$. More concretely, we wish to learn a function f such that

$$Z = f(X, \mathcal{G}) \tag{1}$$

using the given true labels. A well studied instantiation of this problem is image classification, where the pixels are arranged in a lattice. Graph node classification generalizes this problem for structured, non-Euclidean data, with the underlying belief that this structure contains additional information that is not captured by the per-node input features. Examples include social networks, citation networks, web graphs, telecom networks, and more.

The fundamental idea behind GNNs is to build a deep neural network model to learn f. We now discuss three prominent classes of architectures to design such a neural network.

1.1 Graph Convolution Networks

Graph Convolutional Network (GCN) is a GNN variant that attempts to generalize Convolutional Neural Networks (CNNs), which have achieved phenomenal success in image-related tasks [1–3]. The core building block of CNNs is the discrete convolution operator, that is repeatedly used to apply a learned 'filter' on input features. Discrete convolution exploits spatial locality between pixels, but it is only defined for lattice graphs. To extend this to general graphs, [4] defines a graph Fourier Transform, and [5] uses the convolution theorem to define a graph convolution operator $*_{\mathcal{G}}$ such that:

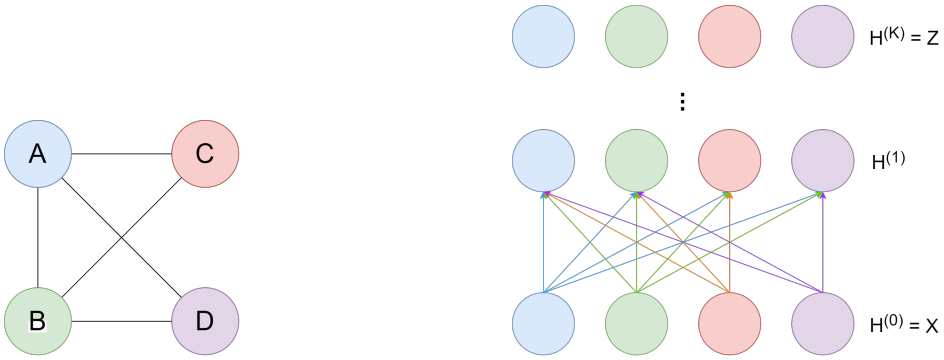
$$g_\theta *_{\mathcal{G}} x = U g_\theta U^\top x \tag{2}$$

where $g_\theta = diag(\theta)$ is a filter parameterized by $\theta \in \mathbb{R}^N$, $x \in \mathbb{R}^N$ is a per-node signal (or feature), and U is the eigenvector matrix of the normalized laplacian of $\mathcal{G}$[1]. Calculating the eigen decomposition of the laplacian, and multiplying U in Equation 2 is computationally prohibitive for large graphs. As a result, [6] and [7] expand graph convolution using Chebyshev polynomials and perform a series of first-order approximations to show that:

$$g_\theta *_{\mathcal{G}} x \approx w(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}) x \tag{3}$$

where $\tilde{A} = A + I_N$, i.e., the adjacency matrix with self loops added. Similarly $\tilde{D} = D + I_N$. $w \in \mathbb{R}$ is a single 'weight' that approximates the effect of θ. Equation 3 only depends on the neighbors of a node to compute its convolution with a filter. Thus, it is inexpensive to compute and can be easily parallelized. We omit the details of this approximation for brevity, and refer the reader to the original texts [4–7] for a more comprehensive treatment.

[1] The laplacian of a graph is defined as $D - A$, where D and A are the degree and adjacency matrix respectively.



(a) An example undirected graph

(b) Neural Network for the example graph with K layers. Each layer has a node for every vertex, and connects to its neighbors in the previous layer.

Figure 1. Graph with its corresponding GNN

With the graph convolution operator defined, a neural network convolutional layer can be defined as follows:

$$\begin{aligned} H^{(k+1)} &= \sigma\left(g_{\theta^{(k+1)}} *_{\mathcal{G}} H^{(k)}\right) \\ &\approx \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(k)}W^{(k+1)}\right) \end{aligned} \tag{4}$$

or equivalently, as a per-node equation,

$$h_v^{(k+1)} \approx \sigma\left(\sum_{u\in\mathcal{N}(v)\cup\{v\}} \frac{h_u^{(k)}}{\sqrt{|\mathcal{N}(v)|\cdot|\mathcal{N}(u)|}} W^{(k+1)}\right) \tag{5}$$

This extends Equation 3 to work with multi-dimensional features and multiple filters. Here, $H^{(k)} = [h_v^{(k)}] \in \mathbb{R}^{N\times O_k}$, and $h_v^{(k)}$ is the activation (or features) of node v at the k^{th} layer. $\sigma(\cdot)$ is a non-linear function (for example ReLU or softmax). $W^{(k+1)} \in \mathbb{R}^{O_k\times O_{k+1}}$ is the weight matrix between k^{th} and $(k+1)^{\text{th}}$ layers that represents O_{k+1} number of filters. Each filter convolves all O_k input features of neighboring nodes to produce a single output feature. $\mathcal{N}(v)$ denotes the neighborhood set of node v i.e., $\mathcal{N}(v) = \{v'|(v',v) \in \mathcal{E}\}$. A variable number of such layers, say K, can be stacked together with the first and the last layer matching the input and output dimensions respectively. This means $h_i^{(0)} = X_i$ (and $O_0 = C$) and $h_i^{(K)} = Z_i$ (and $O_K = B$) for every node i. It is worth reiterating that features of a node at layer $k+1$ only depends on layer k nodes that are a single hop away in the graph. As a result, more layers allow the network to incorporate the effect of nodes that are farther away. Figure 1b shows the schematic diagram of these layers for an example graph.

Similar to any other neural network, the $W^{(k)}$ weight filter matrix at each layer can be learned by gradient descent. [7] defines a cross-entropy loss using only the given true labels, indexed by the set Y_L, as follows:

$$\mathcal{L} = -\sum_{i\in Y_L}\sum_{b=1}^{B} Z_{ib}^{\text{true}} \log(Z_{ib}) \tag{6}$$

where Z^{true} and Z are the true and predicted labels respectively. Similarly, [8] defines a loss function for the unsupervised setting, i.e., when no true labels are available. It encourages nearby nodes to have similar representations and penalizes similarity for disparate nodes.

Real world graphs grow dynamically and new unlabeled nodes and edges may be added to a graph after the weights matrices have been learned. Retraining weights periodically from scratch using these new connections may be infeasible. [8] shows that this may not be needed; weights once learned on the original graph perform reasonably well in predicting the labels of new nodes that were not present in the training set.

1.2 Simple GCN

Simple GCN [9] (SGC) is the spiritual precursor of GCN, although with the benefit of hindsight. It argues that the neural network proposed by GCN is far too complex, and a simpler architecture may be sufficient to learn f in most cases. Specifically, it proposes removing the non-linearity function from all layers except the last. As a result, using Equation 4, the activation at the output layer can be directly expressed as a simple function of the input features:

$$H^{(K)} \approx \sigma\left((\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}})^K X W^{(1)}W^{(2)}\cdots W^{(K)}\right)$$

Moreover, the product of all the weight matrices can be learned together as a single matrix $W = W^{(1)}W^{(2)}\cdots W^{(K)}$. This simplifies the expression to:

$$H^{(K)} \approx \sigma\left((\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}})^K X W\right) \tag{7}$$

This simplified equation has several benefits. First, $(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}})^K$ can be precomputed; forward propagation reduces to exactly two sparse-matrix multiplications (SpMM) along with a final non-linear function such as softmax. This is significantly more efficient than Equation 4, which requires matrix multiplications for every layer. Second, Equation 7 can be interpreted as logistic regression on preprocessed input $\tilde{X} = (\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}})^K X$. The preprocessing step incorporates features from nodes within K-hops. Logistic regression is a convex optimization problem with a global optima. It can be solved accurately and quickly using Stochastic Gradient Descent. Together, these simplifications result in a much lower memory footprint and computational complexity.

Experiments validate this claim. SGC improves end-to-end training and inference time by atleast an order of magnitude on most major graph datasets [10] when compared to GCN or even its optimized versions [11] (discussed later). SGC is also remarkably competitive, performing within 1% of other more complex architectures in terms of test accuracy.

1.3 Other Architectures

Several recent papers have focused on designing other alternative neural network architectures for the node classification problem. The most prominent of these is GraphSAGE [8], which templatizes Equation 5 as follows:

$$h_v^{(k+1)} = \sigma(\textsc{Con}(h_v^{(k)}, \textsc{Agg}(\{h_u^{(k)} | u \in \mathcal{N}(v)\})) \cdot W^{(k+1)}) \tag{8}$$

Here, $\textsc{Agg}(\cdot)$ and $\textsc{Con}(\cdot)$ represent any generic aggregate and concatenate functions respectively. $\textsc{Agg}(\cdot)$ aggregates the activations of neighbors of a node. $\textsc{Con}(\cdot)$ concatenates the aggregation with the node's own previous activation. Specific instantiations of $\textsc{Agg}(\cdot)$ and $\textsc{Con}(\cdot)$ include a variety of differentiable operators, beyond the one given in Equation 5. Table 1 shows a few examples. In practice, $h_v^{(k+1)}$ is also normalized to a unit vector to prevent the exploding or vanishing gradient problems.

Architectures	$\textsc{Agg}(h_1, \cdots, h_n)$	$\textsc{Con}(h_v, x)$
GCN	$\sum_i \frac{h_i}{\sqrt{\mathcal{N}(i)}}$	$\frac{h_v}{\mathcal{N}(v)} + \frac{x}{\sqrt{\mathcal{N}(v)}}$
GraphSAGE-mean	$\sum_i h_i$	$\frac{h_v + x}{\mathcal{N}(v)}$
GraphSAGE-pool	$\max_i (h_i)$	$h_v \| x$
GraphSAGE-LSTM	$\text{LSTM}(h_1, \cdots, h_n)$	$h_v \| x$

Table 1. Different aggregate and concatenate functions

Experiments show that these new aggregation and concatenate functions improve prediction accuracy on most datasets, presumably because of their higher expressive capabilities. Out of all different aggregators presented, the LSTM aggregator performs the best. The max pooling operator also results in comparable performance while being computationally inexpensive.

However, GraphSAGE pays for this with a loss in interpretability. It moves away from the spectral theory roots: it isn't clear if any general aggregate function is related to the approximation of the graph convolution operator defined in Equation 3. Further, it is not clear why non-symmetric aggregation functions like LSTM perform better, even though neighbors of a node have no inherent order.

2 Stochastic Sampling

The GNN architectures described above do not scale to real-world graphs with a large number of nodes and edges. This scaling problem can be attributed to two main reasons. First, the space and time complexity of GNN training is proportional to the size of the graph. In particular, the forward pass algorithm (Equation 8) depends on the entire neighborhood of a node to compute its activation. This leads to the worst-case time complexity of $O(|\mathcal{V}|^2)$ per layer, or $O(K * |\mathcal{V}|^2)$ for the entire network. Further, backpropagation to compute gradients using the loss function defined in Equation 6 requires storing the activations of all neighbors of nodes in the previous layer. In the worst case, this has a space complexity of $O(K * |\mathcal{V}|)$.

Secondly, there is usually significant variance in the sizes of neighborhoods of different nodes. This variance is reflected in the time it takes to compute their activations and gradients, making it hard to effectively parallelize GNN training on GPUs.

To mitigate these training challenges, a series of papers have introduced and extended the idea of "sampling". Sampling techniques approximate the effect of the full neighborhood of a node with a small, randomized subset. The size of this subset is usually bounded, resulting in a much lower computational complexity. These techniques can be broadly classified into three categories: node-wise, layer-wise, and subgraph-wise. We now discuss a few representative papers that highlight each of these classes.

2.1 Node-Wise Sampling

2.1.1 GraphSAGE. GraphSAGE [8] introduces the idea of node-wise neighborhood sampling. It suggests *uniformly* sampling a subset from the set of neighbors of each node at every layer in the network. The size of this set is fixed for every node at a particular layer. Figure 2 shows example subsets of neighborhoods of different nodes.. Let S_k denote the size of the fixed-size subset that is sampled at for the neighborhood of every node at layer k. Then, the space and time complexity is of the order

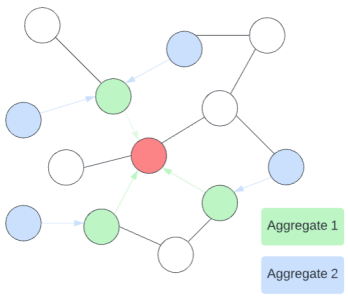


Figure 2. To compute the activation of the red node, only some selected neighbors (green) are sampled and aggregated. Similarly, for green nodes, only the blue nodes are sampled.

$O(\Pi_{k=1}^{K} S_k)$ per batch, i.e. for a single pass of forward and backward propagation. $S_i, i \in \{1, 2, \cdots, K\}$ are hyperparameters that can be chosen by the user. The values of S_i are chosen to be small enough so that $\Pi_{k=1}^{K} S_k << |\mathcal{V}|^2$. As a result, training GNNs using this sampling technique is significantly more efficient.

Results and Limitations. For the experiments presented in the paper, the authors show that increasing values of S_i yield decreasing marginal benefits in accuracy. However, the total training time grows super-linearly with S_i. For a two-layer network ($K = 2$), they choose S_1 and S_2 such that $S_1 \times S_2 \leq 500$ for all datasets [10].

A major limitation of this work is that it does not explore sampling techniques beyond uniform sampling. Further, it does not experiment with different-sized subsets of neighbors for each node. A degree-dependent sampling strategy could be a possible substitution for uniform sampling.

2.1.2 PinSage. PinSage [12] is a large-scale deep recommendation engine that was developed and deployed at Pinterest. PinSage combines the ideas of random walk statistics and neighborhood-based aggregation to extract important information from both of them. PinSage is included as a part of this study due to the numerous improvements that it makes over GraphSAGE as well as its demonstration of using the discussed methods in a practical setting, namely web-based recommender systems.

PinSage leverages some key ideas to improve the training of GCNs on large graphs which can be summarized as follows:

1. On-the-fly convolutions: This idea is borrowed from GraphSAGE wherein instead of using the entire Laplacian of the graph during the convolution operation, a smaller neighborhood (obtained from node-wise sampling) is used.
2. Producer-consumer minibatch construction: This is one of the main contributions of this paper which ensures maximum GPU utilization. A large memory, CPU-based producer can store the entire graph and efficiently sample node neighborhoods as well as fetch features required to define the convolution whereas a GPU-based consumer model takes as input the computation graphs formulated by the producer and performs loss backpropagation.
3. Efficient MapReduce inference: This distributes the trained model in such a way that generating inference minimizes computation.

The problem is organized as follows: users interact with *pins*, which are visual bookmarks to online content on Pinterest, such that can be organized into *boards*, on the basis of thematic relationship. The recommendation task is as follows: given a pin that has been marked by a user, identify a closely related pin, that is, find related pins to a given pin. The problem is modeled as a bipartite graph with one set of nodes being the pins and the other set being the boards. The edges connect pins which are linked to various boards. The proposed algorithm aims to find *vector embeddings*, i.e., activation of the final layer, for each node and then recommendations can be made using nearest-neighbor lookup algorithms in the embedding space.

The forward pass for the algorithm looks the same as Equation 8 with the exception that the neighborhood set $\mathcal{N}(v)$ is not sampled uniformly as in GraphSAGE. The concept of importance-based neighborhoods is used instead, that is, selecting nodes from the neighborhood of a node that exert the most influence on it. To determine this, the authors perform random walks starting at a given node u and compute the L_1-normalized visit count of the nodes visited during the random walk.

For this paper, the authors consider the top T neighbors for a given node in every layer (in contrast to S_k for the k^{th} layer in GraphSAGE). Keeping track of the normalized counts of the nodes visited during a random walk allows the algorithm to aggregate their embeddings using a weighted average (normalized counts) which takes into account the influence of each neighbor. The authors refer to this as *importance pooling*.

As mentioned earlier, the producer-consumer minibatch construction scheme is used to make maximum utilization of CPU and GPU memory. The adjacency list and the feature matrices for huge graphs are stored in CPU memory where the random walks are performed. At the beginning of each iteration, the features as well as the weights associated with each minibatch are fed to the GPU memory where the training happens.

Once the training is complete, fetching embeddings for nodes during inference for huge graphs might still introduce latency, especially in the case of the bipartite graph in question, which requires computing embeddings

for the two sets of nodes in different ways. The authors develop an inference procedure specific to this problem (which uses *pins* and *boards*) which works in the following steps:

1. The first step involves *mapping all pins* to the embedding space using the trained model.
2. The second step involves *aggregating (reducing) the mapping of pins associated with a board to find out the embedding of the board*.

The authors introduce one final optimization idea to improve training which is *Curriculum Training*. The idea is that the model is fed harder and harder examples in each epoch while learning so as to improve the learning curve trajectory (in this case, a hard-to-learn example would be a pair of nodes that are not significantly further apart in the graph but should not be close in the embedding space).

Results and Limitations. The authors define metrics for measuring the performance of recommendation systems, *hit-rate* and *Mean Reciprocal Rank*. PinSage achieves a significantly higher performance for both of these metrics against all of the measured baselines as well as the performed ablation studies (using aggregation schemes that do not make use of neighborhood weights in contrast to importance pooling).

Additionally, the authors perform head-to-head user studies against baselines where a user is presented with two examples from PinSage and a baseline in addition to a reference pin, and the user to asked to pick a recommendation based on the reference or assign them an equal score. PinSage exhibits a higher win percentage against all the baselines.

One major limitation of this work is that it focuses on baselines such as visual and annotation embeddings which make no use of the graph structure and hence, it may not be fair to use them for comparison. The only graph-based method that they use is Pixie which does not use any neural network over the graph structure and relies on random walks only. The other major limitation is that even though the method makes use of CPU and GPU runtimes, the authors report times for the GPU section of the pipeline, and no CPU times are reported. The CPU memory requirements (which are expected to be quite high since the entire graph along with features and node-wise neighborhood weights need to be stored) are not mentioned as well.

2.2 Layer-Wise Sampling

While the node-wise sampling idea cut some cost, it still suffers from the exponential expansion of neighbors. FastGCN [11] was the first to present the idea to view graph convolutions and the final loss as an integral transform of the activation from one layer to another, and of sampling a certain number of nodes in one sampling step. To introduce layer-wise sampling, we recall the forward propagation for graph convolutions from Equation 5 restated as an integral:

$$h^{(k+1)}(v) = \sigma\left(\int_{u\in\mathcal{V}'} \hat{A}(u,v).h^{(k)}(u).W^{(k+1)}.dP(u)\right) \quad (9)$$

Where $\hat{A}(u,v) = \frac{A(u,v)}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}$ is the normalized Laplacian relation, and the activations at each layer are interpreted as functions of the nodes. This integral is imagined to be over an *infinite* graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$. The actual graph data $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is assumed to be induced from this infinite graph after sampling $\mathcal{V}$ from $\mathcal{V}'$ using some probability measure $(\mathcal{V}', \mathcal{F}, P)$. Each node is assumed to be drawn from an iid random variable. Thus, using some bootstrapped sampling distribution Q over $\mathcal{V}$, the integral becomes:

$$h^{(k+1)}(v) = \sigma\left(\mu_Q^{(k+1)}(v)W^{(k+1)}\right) \quad (10)$$

where

$$\mu_Q^{(k+1)}(v) = \mathbb{E}_{u\sim Q}\left[\hat{A}(u,v).h^{(k)}(u).\frac{dP(u)}{dQ(u)}\right] \quad (11)$$

In general, $Q(u)$ denotes the probability of sampling node u for layer k conditioned on the sampled nodes $v_1, \cdots, v_{t_{k+1}}$ in layer $k+1$. Without loss of generality, assume that the nodes are sampled starting from the top layers to the bottom. To speed up forward propagation of Equation 10, the paper uses Monte-Carlo approximation to compute $\mu^{(k+1)}$:

$$\hat{\mu}_Q^{(k+1)}(v_i) = \frac{1}{t_k}\sum_{j=1}^{t_k} \frac{\hat{A}(u_j,v_i).h^{(k)}(u_j)}{Q(u_j|v_1,\cdots,v_{t_{k+1}})} \quad (12)$$

Here, $\hat{\mu}_Q^{(k+1)}(v_i)$ is the approximate expectation for layer $k+1$ for node v_i, where $u_j \sim Q(u_j|v_1,\cdots,v_{t_{k+1}})$ is the probability of sampling node u_j in layer k given nodes $v_1, \cdots, v_{t_{k+1}}$ in layer $k+1$. A variety of sampling methods have been developed by analyzing different choice of Q. Typically, they aim to reduce the variance of the estimator $\hat{\mu}_Q^{(k+1)}(v_i)$, i.e., find a sampling probability distribution Q that minimizes $\mathrm{Var}_Q\left[\hat{\mu}_Q^{(k+1)}(v_i)\right]$. Other methods work on different objectives, such as reducing the sparseness of forward computation defined in this way.

2.2.1 FastGCN.

FastGCN [11] is a method for training GCN that uses independent sampling of nodes in each layer to reduce expensive computation. By interpreting the convolution operation in GCN as an integral transformation of the embedding function, FastGCN estimates node embeddings using the Monte Carlo approach, which involves approximating embeddings by sampling

a certain number of nodes in each layer. This inductive learning process achieves a significant speedup for GCN training.

In FastGCN, the sampling is done independently for each layer in a batched manner, and even within each layer, the nodes are sampled independently. This makes the sampling fast and efficient. To reduce the variance of the estimator, FastGCN suggests an importance sampling technique to alter the probability distribution. While the optimal distribution to minimize $\mathrm{Var}_Q\left[\hat{\mu}_Q^{(k+1)}(v_i)\right]$ is expensive to compute since it depends on the parameters, the paper suggests the sampling probability distribution:

$$Q(u) = \frac{\sum_{v\in\mathcal{V}} \hat{A}(v,u)^2}{\sum_{u\in\mathcal{V}}\sum_{v\in\mathcal{V}} \hat{A}(v,u)^2}, \quad u \in \mathcal{V} \tag{13}$$

works well in practice in comparision against uniform sampling distribution. Intuitively, the importance sampling denotes that a node with higher in-degree has a higher probability of being sampled. The sampling distribution is also independent of the layer, as well as the nodes sampled in the layer till now. The algorithm samples t_k nodes for each layer k of the GCN according to Q, and the inter-layer connections are reconstructed using the graph $\mathcal{G}$ to compute the forward pass for the sampled nodes. This paper formally proves that the constructed Monte-Carlo estimator is consistent, i.e., with increasing the number of samples, the estimator $\hat{\mu}_Q^{(k+1)}(v_i)$ converges to the expected value.

Results and Limitations. This method alleviates the overhead of exponential expansion of neighborhood in node-wise sampling method which had a complexity of $O(\Pi_{k=1}^{K} S_k)$ to a linear trend of $O(\sum_{k=1}^{K} S_k)$ per batch. This saves computation costs in both memory and time to sample. This paper is the first to introduce a scalable mini-batch procedure, and also formally shows the converge theoretically and in practice on most common graph datasets. FastGCN also first describes a close to optimal importance sampling method using the in-degree of the node, for sampling nodes independently in each layer, and across layers.

The paper evaluates the effective speed-up by comparing the total training time until convergence, against the original GCN, batched GCN and GraphSAGE. They find that FastGCN has orders of magnitude improvement, particularly so for larger and dense graphs, e.g. on Reddit FastGCN has the 5x speedup in training time compared to GraphSAGE, and 100x speedup compared to batched GCN implementation, while the original GCN implementation goes out of memory. FastGCN had a 15x speedup to GraphSAGE on CORA and PubMed dataset.

They also report only a marginal difference in micro F1 scores in the downstream tasks for these datasets, with a maximum deviation of only 0.01 from the best model which was usually GraphSAGE.

A core limitation of FastGCN is that since the inter-layer sampling is also independent, it may cause the situation that the sampled nodes are not connected in two consecutive layers. While importance sampling might mitigate this effect a little, in general this can result in very sparsely connected layers during forward propagation and gradient computation. This can deteriorate training and model performance.

2.2.2 LADIES. Layer Dependent Importance Sampling (LADIES) builds directly on top of FastGCN to fix its core limitation. It proposes a layer-dependent importance sampling approach to reduce sparsity in the inter-layer connections introduced by independent sampling, while also maintaining the linear trend in the memory consumption.

To solve the above challenge, LADIES performs layer-dependent sampling in a top-down manner. Given the sampled vertices $\mathcal{V}^{(k+1)} = \{v_1, \cdots, v_{t_{k+1}}\}$ for layer $k+1$, the probability of sampling u in layer k is given by:

$$Q^{(k)}(u|\mathcal{V}^{(k+1)}) = \frac{\sum_{v\in\mathcal{V}^{(k+1)}} \hat{A}(u,v)^2}{\sum_{u\in\mathcal{V}}\sum_{v\in\mathcal{V}^{(k+1)}} \hat{A}(u,v)^2}, \quad u \in \mathcal{V} \tag{14}$$

Observe that if u is not the neighbor of any of the nodes sampled in the next layer $k+1$, then $Q^{(k)}(u|\mathcal{V}^{(k+1)}) = 0$. This means that LADIES only samples a node in layer k, if it is a neighbor of some node in layer $k+1$. Moreover, the importance sampling here intuitively denotes that a node u which connects to more nodes in layer $k+1$ has a higher probability of being sampled for layer k. This makes the computation graph for the batch dense, since with high probability, a large number of edges will be connected between layers.

Results and Limitations. LADIES maintains the linear trend of $O(\sum_{k=1}^{K} S_k)$ memory per batch, with a little increasing in computation time due to dependence in sampling from top layer to the next. This also makes the sampled computation graph denser (more edges across layer), thereby solving a major drawback of FastGCN. This improves the rate of convergence of mini-batch gradient descent and the training procedure.

The authors evaluate LADIES on the same datasets as the others (CORA, PubMed, and Reddit) [10] and report Micro F1 for downstream tasks, and the total training time, per-batch computation time and memory usage. They observe that layer-sampling methods like LADIES and FastGCN are more robust to noisy and stochastic data in the graphs compared to full batch methods. LADIES has a time and memory complexity similar to FastGCN, which are improvements over GCN and GraphSAGE. The authors observe that for dense graphs like Reddit, their

Micro F1 score is a consistent improvement over FastGCN under the observation that FastGCN can have many sparse computation graphs due to its sampling procedure not being neighborhood aware.

2.3 Subgraph-Based Sampling

When considering any node-wise or layer-wise sampling method to select a subset of the neighborhood graph on which to perform the training, it is important to note that the nodes which are sampled for each mini-batch are reused during every epoch of backpropagation. This motivates the selection of a mini-batch in such a fashion that each mini-batch contains closely-connected subgraphs of the entire graph.

The above insight can be formalized by the notion of Embedding Utilization. Consider a GNN with K layers. Subsequently, consider a node i at layer k whose activation is used u times for the computation of all node activations at layer $k+1$. Then, the embedding utilization of this node at layer k is defined as u. If we sample a mini-batch by performing random sampling, we expect the value of u to be extremely small since graphs are usually large and sparse.

ClusterGCN [13] is a framework that aims to sample nodes such that embedding utilization is maximized. This, in turn, reduces computations per batch. This problem is solved by connecting embedding utilization to a clustering objective. Consider a batch of nodes $\mathcal{B}$ that is used across all the layers of the neural network. Denote the subgraph formed by the nodes in $\mathcal{B}$ as $\mathcal{G}_{\mathcal{B}}$. Since $\mathcal{G}_{\mathcal{B}}$ is used to compute the activations in every layer, the embedding utilization is equal to the number of edges present in $\mathcal{B}$. This motivates the design of $\mathcal{B}$ so as to maximize the number of edges within the batch and minimize the number of edges between each batch.

ClusterGCN works as follows: for a graph $\mathcal{G}$, partition its nodes into c groups $\mathcal{V} = [\mathcal{V}_1, \mathcal{V}_2, \cdots, \mathcal{V}_c]$ where $\mathcal{V}_i$ consists of the nodes in the i^{th} partition. This will give us c subgraphs

$$\bar{\mathcal{G}} = [\mathcal{G}_1, \cdots, \mathcal{G}_c] = [(\mathcal{V}_1, \mathcal{E}_1), \cdots, (\mathcal{V}_c, \mathcal{E}_c)],$$

where each $\mathcal{E}_i$ consists of the edges only between the nodes in $\mathcal{V}_i$. Using this, we can rewrite the adjacency matrix (denoted as A) as a decomposition of c^2 submatrices as follows:

$$A = \bar{A} + \Delta = \begin{bmatrix} A_{11} & \dots & A_{1c} \\ \vdots & \ddots & \vdots \\ A_{c1} & \dots & A_{cc} \end{bmatrix} \tag{15}$$

where

$$\bar{A} = \begin{bmatrix} A_{11} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & A_{cc} \end{bmatrix}, \Delta = \begin{bmatrix} 0 & \dots & A_{1c} \\ \vdots & \ddots & \vdots \\ A_{c1} & \dots & 0 \end{bmatrix} \tag{16}$$

where each diagonal block A_{ii} of size $|\mathcal{V}_i| \times |\mathcal{V}_i|$ contains the links within $\mathcal{G}_i$. Δ contains the remaining off-diagonal entries of A. Similarly, the feature vectors X and labels Z can be partitioned as $[X_1, \cdots, X_c]$ and $[Z_1, \cdots, Z_c]$ respectively according to the clusters. The block-diagonal approximation results in the loss function being decomposable into batches. Using $\bar{A}$ as the approximation of A allows the forward and the backpropagation steps to be written as matrix multiplications instead of neighborhood search procedures.

Graph clustering methods such as Metis [14] and Graclus [15] can be used to perform the clustering step to create the batches as mentioned above. These methods capture the community structure of the graph and promote in-cluster links, which is beneficial for two reasons:

1. The embedding utilization is equivalent to the number of edges in each cluster which are maximized by the very construction of these methods.
2. If we approximate A by $\bar{A}$, then the error term comes from the Δ matrix which denotes the edges between the clusters which are minimized during the cluster construction.

One consequence of using subgraph as the batch is that the computation is essentially now broken down to a subgraph level wherein a single batch can be loaded into the memory and processed independently of the other batches, leading to a more memory-efficient algorithm.

The above scheme completely disregards the edges between the clusters (ignores Δ), and the authors refer to it as the vanilla cluster-GCN scheme. Removing these edges can lead to increased approximation error in the gradients. However, there is a more fundamental problem with using clustering methods to obtain the subgraphs. Clustering algorithms group similar nodes together, which can lead to the distribution of the clusters being different from the original graph, which can lead to a bias in the estimate of the gradients. This is confirmed by looking at the entropy distribution for each cluster which in many cases, is not representative of the overall entropy distribution, which leads to convergence issues.

The workaround is as follows: cluster the graph into p clusters keeping the value of p very large. The batch $\mathcal{B}$ can be constructed by sampling q such clusters

$$\mathcal{B} = \mathcal{V}_{i_1} \cup \mathcal{V}_{i_2} \cup \cdots \cup \mathcal{V}_{i_q}$$

where $\mathcal{V}_{i_j}$ is the j^{th} cluster (out of q clusters which have been sampled) for the i^{th} batch. Additionally, once this sampling is complete, the edges between the different clusters are added back.

Results and Limitations. Using the scheme where a batch is composed of multiple clusters results in smaller variance across the batches and reduces the bias in the

gradient computation. The overall effect is fast convergence, low memory requirement, and low runtime. Additionally, with this setup, the authors train deeper networks on huge graphs which was previously not possible since the computation would result in an out-of-memory error on the GPU. The deeper networks exhibit improved performance.

One limitation of this work is that it is not applicable to graphs when each node in the graph has a similar degree. In this case, the clustering process will not be possible and even if the clusters are obtained by random sampling, the approximation error due to ignoring the edges between the batches will be very high which will result in convergence issues.

2.4 Heterogenous Sampling Method

The heterogeneous sampling method [16] (HetGNN) is a new approach for handling heterogeneity in graphs and improving training speed. This method focuses on efficiently sampling from various types of nodes in a heterogeneous graph, which contains nodes and edges of different types. A heterogeneous graph extends the definition of a regular graph such that $\mathcal{G} = (\mathcal{V}, \mathcal{E}, O_{\mathcal{V}}, \mathcal{R}_{\mathcal{E}})$, where $O_{\mathcal{V}}$ and $\mathcal{R}_{\mathcal{E}}$ represent node and edge types. The previously described Graph Neural Network approaches are not directly applicable to heterogeneous graphs due to two key properties: heterogeneity and semantics. Since these graphs contain multiple types of nodes and relations, the aggregation function cannot be directly applied to them. Additionally, they contain semantic information that is implicitly captured by higher-order relations, called meta-paths. Different meta-paths reveal different semantics, serving as a way to observe the target node's local structure. In general, the relationships between nodes in $\mathcal{G}$ are complex and imbalanced, resulting in nodes having different numbers of neighbors and different types of neighbors with unbalanced numbers. This makes sampling neighbors and capturing neighborhood representation a significant challenge.

GNNs typically aggregate features from a node's direct neighbors, as seen in §1. Concretely, directly extending previous methods to heterogeneous graphs can cause the following issues:

- They cannot capture equitable information from different types of neighbors, resulting in insufficient or incorrect representation
  Suppose there is a heterogenous graphs of Authors, their Publications, the Conferences of the papers and Places (where the authors live as well as the conferences). Directly applying a standard GNN here will treat the author-location edges same as author-publication edges and might also mix their embeddings while aggregation. This can result in a noisy learning with no grounding based on the actual attribute of the node.
  Moreover, just a neighbor aggregation might not help to learn representation that encodes information about k-hop neighbors where different hops result in different types of nodes.
- These approaches are weakened by varying neighborhood sizes and cannot aggregate content features from heterogeneous neighbors. This leads to noisy representation for nodes with a large degree and many different kinds of neighbors and insufficient representation for nodes with a low degree and mostly homogeouns neighbors.

To address these issues, HetGNN designed a new strategy for sampling heterogeneous neighbors. This approach is based on the random walk with restart (RWR) algorithm and comprises of two steps that enable it to collect neighbors of all node types and group neighbors with the same content features:

1. Sampling fixed length RWR. It starts a random walk from node $v \in \mathcal{V}$. The walk iteratively travels to the neighbors of current node or returns to the starting node with a probability p. RWR runs until it successfully collects a fixed number of nodes, denoted as $\text{RWR}(v)$. Note that numbers of different types of nodes in $\text{RWR}(v)$ are constrained to ensure that all node types are sampled for v.
2. Grouping different types of neighbors. For each node type t, we select top k_t nodes from $\text{RWR}(v)$ according to frequency and take them as the set of t-type correlated neighbors of node v, denoted as $\mathcal{N}_t(v)$.

This method avoids the discussed issues since RWR collects all types of neighbors for each node. As a result, it does not suffer from just the first-order approximation as in other GNN methods. The sampled neighborhood size of each node is fixed and only the most frequently visited neighbors are selected, keeping the computation cost low. This can work in tandem with other layer sampling procedures for training since now a new neighborhood set $\mathcal{N}_t(v)$ has been defined for each type t. Neighbors of the same type (having the same content features) are grouped such that type-based aggregation can be deployed.

Results and Limitations. The major advantage of Heterogenous GNN is that it allows the extensive use of context data present in the nodes, jointly along with the graph information. While the experiments done by the paper are extensive over different heterogenous graph settings in Citation and Conferences networks and recommendation networks, a major section of the paper went into motivating architectures for neural networks to create various kinds of encoding and combining the

encoders. The novelty in the paper is in the handling of heterogenous data as neighbors of different kinds using a Random Walk strategy to collect higher hop neighbors which are of different types.

The authors evaluate HetGNN against other models like GraphSAGE and GAT, and observe a relative improvement of 5 - 10% in the efficacy of the generated embeddings for downstream tasks like recommendation, link prediction and inductive node classification in heterogenous graphs.

A limitation is that a major portion of the paper seemed to tackle problems by increasing the capacity of the neural networks employed, and the authors do not note down for a fair comparision with other GNN architectures based on the capacity of the models (number of parameters in the architecture). So it us unclear if the improvement is because of the RWR sampling + type aggregation procedure, or just due to the increase in model capacity.

3 Removing Redundant Computations

Next, we describe a technique to improve the performance of the forward pass algorithm without any loss in accuracy. [17] identifies and describes a method to reduce redundant computations in Equation 8. Their findings are based on two key observations. First, they note that most real-world graphs are clustered, i.e., nodes often have multiple common neighbors. For instance, social networks evolve following the rule of triadic closure: if the connections $u - v$ and $u - w$ exist, there is a tendency for the new connection $v - w$ to be formed. Figure 1a shows an example graph where nodes A and B have two common neighbors – C and D.

Second, the $\textsc{Agg}(\cdot)$ function can often be reduced to repeated application of an *associative* binary operator on neighbors. For instance, this binary operator is sum in the case of GraphSAGE-mean:

$$\begin{aligned}\textsc{Agg}_{\text{mean}}(h_u, h_v, h_w) &= \textsc{Agg}_{\text{mean}}(h_u, \textsc{Agg}_{\text{mean}}(h_v, h_w)) \\ &= h_u + (h_v + h_w) \end{aligned} \tag{17}$$

These two observations can be put together to identify redundant computations in Equation 8. The aggregate over the neighbors of any node can be reduced to the repeated application of a binary operator. Since nodes often have common neighbors, several intermediate operands of this binary operation are repeated. As an example, consider a GraphSAGE-mean based neural network (see Table 1) for the graph in Figure 1a. The $(k+1)^{\text{th}}$ layer activations of node A and B can be computed as follows:

$$h_A^{(k+1)} = \sigma(\textsc{Con}(h_A^{(k)}, h_B^{(k)} + (h_C^{(k)} + h_D^{(k)})) \cdot W^{(k+1)}) \tag{18}$$

$$h_B^{(k+1)} = \sigma(\textsc{Con}(h_B^{(k)}, h_A^{(k)} + (h_C^{(k)} + h_D^{(k)})) \cdot W^{(k+1)}) \tag{19}$$

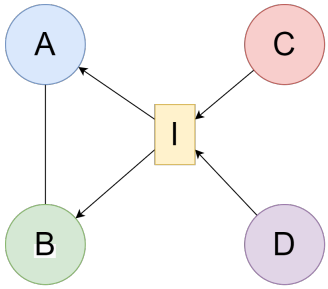


Figure 3. An example HAG for the graph in Figure 1a. I is an intermediate node that stores the aggregation of C and D. This is used twice: to compute the activations of A and B.

Clearly, $(h_C^{(k)} + h_D^{(k)})$ is computed twice and is redundant. In general, the number of redundancies scale with the size of the graph.

[17] removes these redundancies by introducing *intermediate* nodes in the graph. These intermediate nodes store the result of partial aggregates such as $(h_C^{(k)} + h_D^{(k)})$ that can be reused during the computation of activations of nodes. Specifically, it introduces a modified graph $\mathcal{G}' = (\mathcal{V}, \mathcal{V}_A, \mathcal{E}')$ built using the original graph $\mathcal{G}$. $\mathcal{V}_A$ denotes a new set of intermediate nodes, and $\mathcal{E}'$ is the new edge set with edges between both sets of nodes. Edges connected to or from intermediate nodes are directed. This modified graph is called a HAG. For any $v \in \mathcal{V} \bigcup \mathcal{V}_A$, the forward-pass algorithm is modified to work with HAGs as follows:

$$a_v^{(k+1)} = \begin{cases} \textsc{Agg}(\{h_u^{(k)} | u \in \mathcal{N}'(v)\}) & v \in \mathcal{V}_A \\ h_v^{(k)} & v \in \mathcal{V} \end{cases} \tag{20}$$

$$h_v^{(k+1)} = \sigma(\textsc{Con}(h_v^{(k)}, \textsc{Agg}(\{a_u^{(k+1)} | u \in \mathcal{N}'(v)\})) \cdot W^{(k+1)}) \tag{21}$$

where $\mathcal{N}'(v)$ is the new neighborhood function according to $\mathcal{E}'$. Intuitively, $a_v^{(k+1)}$ stores the aggregate of activations of incoming nodes connected to an intermediate node v. This aggregate is then reused to calculate the next layer's activation of every outgoing node connected to v. Figure 3 shows a HAG corresponding to our example graph. A HAG $\mathcal{G}'$ is equivalent to a graph $\mathcal{G}$ iff for every original node $v \in \mathcal{V}$, the expression of activation h_v is equivalent in both the cases.

3.1 Building a HAG

We briefly cover the algorithm to build an equivalent HAG starting from any general graph. The paper limits the search space of equivalent HAGs by bounding the maximum number of intermediate nodes that can be introduced. In other words, $|\mathcal{V}_A| \leq \mathit{capacity}$, where $\mathit{capacity}$ is a hyperparameter of the HAG building algorithm. This bound is motivated by multiple reasons.

First, each additional intermediate node in the HAG has a memory overhead to store a_v [2]. Further, empirically, the reduction in redundancy has diminishing marginal returns as $capacity$ increases; the additional time to search in a larger space may outweigh the benefit at the frontier.

Finding the *optimal* HAG equivalent to a given graph, i.e., the HAG that reduces the most number of redundant computations with up to $capacity$ intermediate nodes, is NP-hard. This can be proven by a reduction from the well-known maximum coverage problem. Worse, real-world graphs have a large ($\geq 10^7$) number of edges. Any super log-linear algorithm to build a HAG would be computationally infeasible.

To this end, [17] describes a simple $(1-\frac{1}{e})$−approximation algorithm with time complexity $O(capacity \times |\mathcal{V}| + |\mathcal{E}| \times \log(|\mathcal{V}|))$. It initializes a HAG $\mathcal{G}' = (\mathcal{V}, \mathcal{V}_A, \mathcal{E}')$ such that $\mathcal{V}_A = \phi$ and $\mathcal{E}' = \mathcal{E}$. The algorithm then proceeds in two steps. First, it finds the pair of nodes with the maximum overlapping neighborhood sets:

$$(u^*, v^*) = \underset{u,v \in \mathcal{V}}{\operatorname{argmax}}(|\mathcal{N}'(u) \cap \mathcal{N}'(v)|) \tag{22}$$

Then, it introduces a new intermediate node w that captures the partial aggregate of all the common neighboring nodes. The node and edge set can be updated as follows:

$$\mathcal{V}_A \leftarrow \mathcal{V}_A \cup \{w\} \tag{23}$$

$$\mathcal{E}' \leftarrow \mathcal{E}' \cup \{(w, u^*), (w, v^*)\} \cup \{(x, w) | x \in \mathcal{N}(u) \cap \mathcal{N}(v)\} \tag{24}$$

$$\mathcal{E}' \leftarrow \mathcal{E}' - \{(x, u^*), (x, v^*) \mid x \in \mathcal{N}(u) \cap \mathcal{N}(v)\} \tag{25}$$

These two steps are repeated until $capacity$ number of intermediate nodes are added.

3.2 Results and Limitations

This technique is almost universally beneficial. HAGs reduce the number of aggregation operations over the regular graph. Since the result of the computation remains the same, there is no loss in accuracy. Further, the cost to compute a HAG using the linear algorithm described above, and its memory overhead, was found to be negligible in several experiments. Results on standard graph datasets [10] show that the end-to-end performance of training and inference improves by $1.2\times - 3\times$ with $capacity$ set to $\frac{|\mathcal{V}|}{4}$. The exact speedup depends on the sparsity and clustering of the graph.

Notwithstanding these advantages, HAGs suffer from a few limitations. For one, they require that the aggregate function is associative. Recent GNN architectures such as Graph Attention Networks [18] and Temporal GATs [19] use complex non-associative aggregate functions; this technique cannot be extended to such architectures as is.

[2] However, in practice, this memory overhead is negligible since memory can be reused across layers.

Further, the paper provides little guidance on choosing the right $capacity$ hyperparameter. It also assumes that the computational cost to aggregate several activations together is the same as repeated pairwise aggregations. This may not be true in the presence of SIMD vectorization instructions on modern CPUs. A better cost model beyond just reducing the number of aggregations can incorporate such overheads.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] A. Howard, A. Zhmoginov, L.-C. Chen, M. Sandler, and M. Zhu, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," in *CVPR*, 2018.

[4] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst, "Signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular data domains," *CoRR*, vol. abs/1211.0053, 2012. [Online]. Available: http://arxiv.org/abs/1211.0053

[5] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 3844–3852.

[6] D. K. Hammond, P. Vandergheynst, and R. Gribonval, "Wavelets on graphs via spectral graph theory," *Applied and Computational Harmonic Analysis*, vol. 30, no. 2, pp. 129–150, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1063520310000552

[7] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[8] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1025–1035.

[9] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, "Simplifying graph convolutional networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 6861–6871. [Online]. Available: https://proceedings.mlr.press/v97/wu19e.html

[10] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[11] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rytstxWAW

[12] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 974–983. [Online]. Available:

https://doi.org/10.1145/3219819.3219890

[13] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 257–266. [Online]. Available: https://doi.org/10.1145/3292500.3330925

[14] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[15] I. S. Dhillon, Y. Guan, and B. Kulis, "Weighted graph cuts without eigenvectors a multilevel approach," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 11, pp. 1944–1957, 2007.

[16] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 793–803. [Online]. Available: https://doi.org/10.1145/3292500.3330961

[17] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken, "Redundancy-free computation for graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 997–1005. [Online]. Available: https://doi.org/10.1145/3394486.3403142

[18] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[19] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," in *ICML 2020 Workshop on Graph Representation Learning*, 2020.