Scalability and Efficiency of Graph Neural Networks

Yuhao Zhang
University of California, San Diego
yuz870@eng.ucsd.edu

ABSTRACT

The past decade saw the rise of deep learning, which has revolutionized many domains such as computer vision and natural language processing. The data of many of these applications are represented in Euclidean space and assumed to be i.i.d. However, for many other applications such as citation networks, social network analysis, and drug discovery, data is represented as graphs to capture the complex interconnections. Traditional deep neural networks like CNN and RNN cannot be naively used in this domain; thus, there have been numerous efforts to apply existing and develop new techniques to work on graph data. These works are known as Graph Neural Networks (GNN). The complexity of graph data not only poses challenges to learning algorithms, but also causes a series of system issues such as low scalability, low runtime efficiency, and high memory footprint. This survey will give a brief overview of GNNs, including Recurrent GNN, Spectral-based and Spatial-based Convolutional GNN, and the emerging systems built to support GNN training and inference. This survey will focus on scalability, efficiency, and other system issues of GNN models and systems.

1 INTRODUCTION

Deep neural networks, or deep learning, have revolutionized many domains like machine translation, computer vision, and natural language processing. They have proven to be highly effective in extracting complex and implicit features from the data. In many classical applications, the data is naturally represented as regular grids, such as images with their RGB pixel values. Techniques such as CNN [5, 28, 39] are effective in capturing the local and shift-invariant information within an image.

However, not all data has a simple grid or sequence shape; many data are naturally represented as graphs, while many other data can also be transformed into graphs [44]. Examples of graph data range from social and citation network [36, 38], protein-protein interaction [48], to point clouds [42]. Unfortunately, applying classical deep learning methods on graph data is not trivial, as graph data can have a rather irregular topology, and each node may have a different number of neighbors. There may not be ordering among the neighbors. Furthermore, the neighbors could be very distinct from each other, and there can also be information associated with the edges. These characteristics would make deep learning primitives such as convolution hard to define. As Figure 1 presents, natural operations such as translation and convolution on a grid are not trivial to define for graph data.

Apart from the structural complexity, another distinctive characteristic of graphs is the absence of the i.i.d. assumption. In a classical machine learning setting, data points are usually assumed to be independently sampled from their actual distribution. However, data embedded in a graph as nodes are often explicitly connected, and these data are inherently non-i.i.d. This has complex implications when it comes to sub-sampling of graph data.

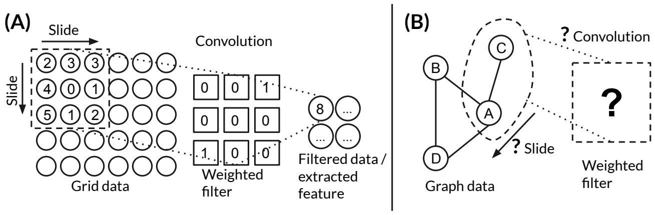

Figure 1: (A): Illustration of convolution on grid data such as image. (B): Natural operations such as translation and convolution on grid are non-trivial to define on graph data.

As such, there have been two directions to bring deep learning to graph data. The first route is by converting the graph data into the regular form to utilize existing deep learning methods; a prominent example is [33]. The second route is developing new deep learning techniques or repurposing old techniques to work on graph data; these newly proposed techniques are known as Graph Neural Networks (GNN). This field has gained lots of attention recently and is the main topic of this paper.

Past efforts on GNNs can be roughly categorized into three divisions: the recurrent GNNs, spatial-based convolutional GNNs, and spectral-based convolutional GNNs [43]. The recurrent GNN and spatial-based convolutional GNN share very much in common, and they both explicitly leverage the structure and spatial localities of graphs. Hence, in some literature, they are also called spatial methods, while spectral-based convolutional GNNs are called spectral methods. Unlike spatial methods derived from deep learning research, spectral methods have a rather different lineage as they rise from the graph signal processing community. We will dive into the details later.

GNN research has not been solely focused on improving the accuracy performance of benchmark datasets. Lots of works focus on the system issues such as memory footprint and scalability [9, 18]. Apart from the algorithmic research and various techniques to bring down the computational cost, GNNs have also received attention from the system community. Just as the classical deep learning techniques need to be repurposed for GNNs, the underlying systems used for training/inference also need to change. The majority of the deep learning systems [1, 32] had very little design consideration for graph data. As a result, their interfaces are unnatural for programming GNNs [41], and their runtime performance is not optimal.

Recently a new branch of system research has emerged: GNN systems [41]. These systems mainly focus on GNN model training and inference and optimize the execution to save time and avoid OOM errors. They are specifically tailored for graph data and GNN workloads, and they often offer orders of magnitude acceleration [14, 41] over the classical deep learning systems.

Many surveys on GNNs exist [3, 43, 47] in the literature. However, none of them emphasizes the system issues and includes the newest advances in the GNN system domain. This paper is a short survey of GNN focusing on research related to the system issues of GNN. In addition to the algorithmic advancements, this paper will also go over some of the state-of-arts of GNN systems and provide potential future research directions.

This paper is organized as follows: Section 2 goes over some necessary backgrounds. Section 3 covers various GNN architectures and the system issues and trade-offs associated with them. Section 4 briefly introduces the emerging research field of GNN systems. Finally, Section 5 concludes this survey and discusses future research directions.

2 BACKGROUND AND PRELIMINARIES

2.1 Deep Neural Networks

Deep neural networks, or deep learning, have revolutionized many domains. To summarize its capability: one writes a set of simple goals (loss functions). Then, the program can automatically learn a way to represent the data and generate predictions for new data. Some prominent examples include CNNs[39] and RNNs[20].

Recently other architectures such as Generative Adversarial Networks (GANs) [17] and pure attention mechanism-based methods such as Transformers [40] have also greatly shaped the landscape.

However, these techniques were majorly developed for regular-shaped data, such as tabular data, images, and time series. Naively applying deep learning to graph data is not viable. The model now needs to consider the underlying topology of the graph and must be capable of following the data relationship defined in the graph; the i.i.d. assumption commonly found in a lot of datasets may not hold for graph data. Therefore, graph data requires special treatment, and the DNN techniques usually need to be tweaked before applying to graph data.

2.2 Machine Learning Systems for Efficient Model Training and Inference

Because of the booming of DNN and their massive adoption across academia and industry, the demand for reliable, easy-to-use, and efficient software also skyrocketed [1, 32, 45]. There are many aspects of the machine learning life cycle, from data sourcing, cleaning to model training and inference. Among these topics, model training and inference are arguably among the ones that face the most scalability and efficiency challenges; the complexity of the models and the amount of data has been rapidly growing up [4, 23]. It is vital for the model training and inference systems to keep up with the pace to enable such workloads at scale. Again, due to the unique nature of graphs and GNNs, existing systems need to be tweaked and specialized for GNN workloads.

2.3 Notations and Definitions

We now summarize the set of notations and definitions used throughout the paper. Table 1 presents all the notations. We now give the definitions of several concepts.

Table 1: Notations used in this paper.

Notation	Description
$G(V, E)$	A graph G with nodes set V and edges set E
v, e	Node and edge
$\mathbf{x}_v, \mathbf{x}_e$	Node and edge features of $\mathbf{v}$ and $\mathbf{e}$, respectively
$\mathbf{X}_V, \mathbf{X}_E$	Node/edge features stacked up as columns formed matrices
$\mathbf{h}_v$	The hidden states/node embedding of v
$\mathbf{H}_V$	The column-stacked matrix of all $\mathbf{h}_v$
D_v, D_e, D_h	Dimension of node and edge features, and hidden states
$w, \mathbf{w}, \mathbf{W}$	scalar weight, weight vector, and weight matrix
$\mathcal{N}(v)$	Function that returns the neighbors of v, including v
$\hat{\mathcal{N}}(v)$	Same as above, excluding v
$\mathbf{A}$	The adjacency matrix
$\mathbf{D}$	The degree matrix
$\mathbf{L}$	The graph Laplacian

(1) **Graph.** A graph $G(V, E)$ is defined as a collection of nodes V and edges E connecting the nodes. The graph can be directed or undirected, cyclic or acyclic.

(2) **Node and edge features.** Each node $v \in V$ can have a D_v-dimensional node feature vector $\mathbf{x}_v \in \mathbb{R}^{D_v}$. Similarly, each edge $e \in E$ can also have a D_e-dimensional edge feature $\mathbf{x}_e \in \mathbb{R}^{D_n}$. Stack these features as column-vectors to form matrix, we have matrix $\mathbf{X}_n \in \mathbb{R}^{D_v \times |V|}$ and $\mathbf{X}_e \in \mathbb{R}^{D_e \times |E|}$.

(3) **Weights and model parameters.** Throughout the paper, the learnable weights of neural networks are typically denoted as w (scalar), $\mathbf{w}$ (vector), or $\mathbf{W}$ (matrix). When it comes to weight matrices composed of weight vectors, they are usually row-based, in contrast to feature vectors and matrices.

(4) **Node neighbors.** Given a node v, the function $\mathcal{N}$ returns the set of all 1-hop neighbors of v, including the node itself. Similarly, define $\hat{\mathcal{N}}$ to be the function that returns all neighbors of a node, excluding itself. This includes both the nodes with edges pointing to v, and nodes pointed by v.

(5) **Node embeddings and hidden states.** Node embeddings are representations of nodes in a Euclidean space through a usually learned mapping. They contain information about the node. In this paper, we use hidden states and node embeddings interchangeably. We use $\mathbf{h_v}$ throughout the paper to denote the node embeddings. The dimensionality D_h is determined by the graph neural network and is usually a hyperparameter.

(6) **Graph Laplacian.** Graph Laplacian is a matrix representation of the graph. It is very important in spectral graph theory and spectral-based convolutional GNNs, please refer to Section 3.3 for details.

3 GRAPH NEURAL NETWORKS

We now go over some representative graph neural network architectures. Roughly speaking, there have been three major branches of GNN: the recurrent GNN, the spatial-based convolutional GNN, and the spectral-based convolutional GNN. The first two types exploit spatial locality information in a graph. Therefore they are also collectively called the spatial method. In contrast, the latter one is called the spectral method, as it conducts graph convolution through a point-wise product in the graph Fourier space, based on

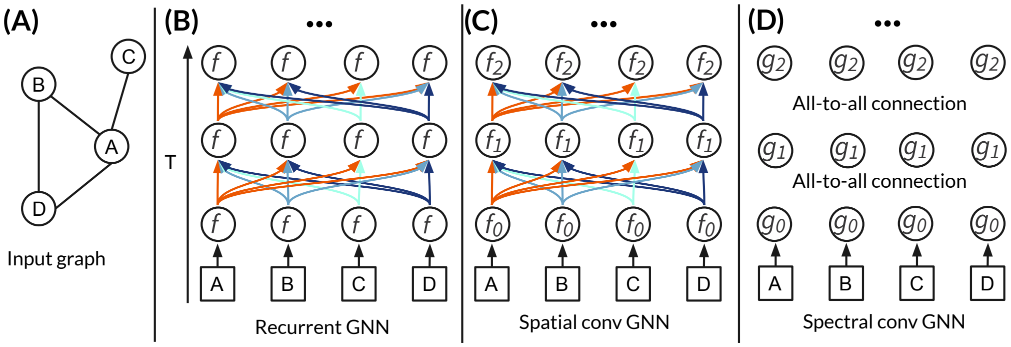

Figure 2: Conceptual comparison of GNN architectures. (A): Input graph. (B): Recurrent GNN. Same function is applied recursively through time. (C): Spatial-based convolution. Different filters are applied in different layers. (D): Spectral-based convolution. Filters operate on the entire graph, unlike locally in (B) and (C).

the convolutional theorem. Such Fourier transformation requires global information of the graph. Figure 2 shows a conceptual comparison between these methods.

Beyond spatial and spectral methods, there also exist other GNN architectures such as graph generative adversarial networks (Graph Autoencoders and GANs) [7, 8] and spatial-temporal GNNs [25, 46]. However, there are overlaps between them and spatial/spectral methods, and many of these architectures rely on primitives such as graph convolution. Due to space limits, this paper will focus on the spatial and spectral methods, as they are deemed more fundamental.

3.1 Recurrent GNNs

Recurrent GNNs are analogous to the Recurrent Neural Networks (RNNs) in the sense that they both contain a recursive neural network layer and apply the same set of parameters to different parts of the input data. Many of the first published works of GNN [35] falls into this category, and there have been numerous subsequent works [12, 15, 24]. In the original paper, the architecture is simply named GNN, and in this survey, we will call it GNN-0 to differ from other GNNs.

3.1.1 *The Graph Neural Network Model (GNN-0).*

GNN-0 [35] is commonly known as one of the first works on graph neural networks. It calculates the hidden states of each node through a message passing/diffusion model, in which the hidden states of nodes are treated as information that can freely flow within the graph. Then each node's hidden state would finally reach equilibrium. Mathematically, the update rule for node v at step t can be written as:

$$\mathbf{h}_v^t = \sum_{u \in \hat{\mathcal{N}}(v)} f(\mathbf{x_v}, \mathbf{x_u}, \mathbf{x_e}, \mathbf{h}_u^{t-1}), \tag{1}$$

where f is a learnable function and implemented using a fully connected neural network. For simplicity, define a single variate function to be:

$$f_w(z) = \sum_{u \in \hat{\mathcal{N}}(v)} f(\mathbf{x_v}, \mathbf{x_u}, \mathbf{x_e}, z) \tag{2}$$

The graph's hidden states/node embeddings are said to be in equilibrium if:

$$\mathbf{h}_v^t = f_w(\mathbf{h}_v^{t-1}), \tag{3}$$

which indicates that the neural network does not further alter the hidden states of the node and that $\mathbf{h}_v^t$ is the fixed point of the function f_w. Additionally, another fully connected layer, denoted as function g, is added at the end of propagation to generate predictions based on the hidden states. The loss is then calculated based on the predictions and can be backpropagated to both g and f_w during training. The choice of using a multilayered neural network to implement f and g is convenient. However, this fixed point does not always uniquely exist unless if the function f_w is a contraction map (Banach fixed-point theorem), a characteristic that a plain fully connected neural network does not possess. Therefore, to guarantee the uniqueness and existence of the fixed point, a correction term needs to be added to the loss function. This correction term would put constraints on the Jacobian of the neural network and consequently force it to be a contraction mapping.

GNN-0 resembles the form of a recurrent neural network, except it has a graph shape, while RNNs are list-like. Recurrent GNN has very similar characteristics with RNNs and shares the same training paradigm. A common practice is to unroll the GNN-0 in time, just like RNN, and Figure 2 (B) shows the unrolled network. The structure of the graph determines the connections between each layer. To train such a GNN model, common methods such as backpropagation through time (BPTT) can be adopted. However, the BPTT algorithm requires storing all the intermediate hidden states during the forward propagation, which drastically increases the memory footprint as the algorithm scales linearly with the number of layers.

To overcome the memory issue of BPTT, GNN-0 adopts Recurrent Back-propagation (RBP), also known as the Almeida-Pineda algorithm [2, 34]. RBP is a method for recurrent neural network training, and it is applicable when the hidden states of the recurrent neural network converge to a fixed point, which GNN-0 exactly does. RBP has a significant memory footprint reduction compared to BPTT as it stores only the hidden states of the last layer of recursion, meaning constant space complexity on a given graph and independent of the recursion depth. RBP works as follows iteratively: it first conducts a recursive forward propagation until the network reaches equilibrium. Then instead of unrolling the network in time, it conducts a recursive backpropagation until the gradients also become stable. It then applies the gradients and repeats for the next iteration.

Even with RBP to control the memory footprint, GNN-0 still faces various issues: as a recurrent neural network, it has the same problems RNNs face. It also conducts gradient descent (GD) that needs the whole graph and all hidden states for each iteration, while the de-facto deep learning training framework is stochastic gradient descent (SGD). Existing systems are mostly optimized for SGD training [1, 31, 32, 37]. There are attempts to apply SGD to recurrent GNN [12], but it is unknown if it still theoretically guarantees convergence to a fixed point. Furthermore, the constraint of contraction map GNN-0 was pointed out that it may hinder the expressive power of the neural network and may cause dropped performance for long-range dependencies [24].

3.1.2 Gated Graph Neural Network (GG-NN).

GG-NN [24] is a follow-up work to GNN-0. It applies a gated recurrent unit (GRU) to construct the update rule of hidden states. Because of this, GG-NN does not guarantee a fixed point anymore and there is no need for the neural network to be contraction mapping. For the same reason, GG-NN requires an initialization the node embeddings, which are chosen to be the node features. GNN-0 does not need such initialization since the exponentially quick convergence is guranteed. The update rule of GG-NN can be written as:

$$\mathbf{h}_v^0 = \mathbf{x}_v, \tag{4}$$

$$\mathbf{a}_v^t = \sum_{u \in \hat{\mathcal{N}}(v)} w_{e_{uv}} \mathbf{h}_u^{t-1}, \tag{5}$$

$$\mathbf{h}_v^t = GRU(\mathbf{a}_v^t), \tag{6}$$

where $w_{e_{uv}}$ represents the parameter associated with the edge e_{uv}, this parameter is also learnable and depends on the type and direction of the edge. GG-NN updates the node embedding by first aggregating the embeddings of neighbors and then conducts a regular GRU operation on the aggregated results, defined as:

$$\mathbf{z}^t = \sigma(\mathbf{W}_z \mathbf{a}_v^t + \mathbf{U}_z \mathbf{h}_v^{t-1}), \tag{7}$$

$$\mathbf{r}^t = \sigma(\mathbf{W}_r \mathbf{a}_v^t + \mathbf{U}_r \mathbf{h}_v^{t-1}), \tag{8}$$

$$\mathbf{c}^t = \sigma(\mathbf{W}_c \mathbf{a}_v^t + \mathbf{U}_c(\mathbf{r}^t \odot \mathbf{h}_v^{t-1})), \tag{9}$$

$$GRU(\mathbf{a}_v^t) = (1 - \mathbf{z}^t) \odot \mathbf{h}_v^{t-1} + \mathbf{z}^t \odot \mathbf{c}_v^t. \tag{10}$$

Equation (7), (8), and (9) correspond to the update gate, reset gate, and current memory content of a GRU [11], respectively. *sigma* is the sigmoid function; $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}_c, \mathbf{U}_z, \mathbf{U}_r, \mathbf{U}_c$ are weights associated with each gate. This grants GG-NN the capability of capturing long-range dependencies, and it relaxes the contraction mapping constraint. It still shares the same form as GNN-0. GG-NN then adopts truncated BPTT for training. Truncated BPTT can also avoid the potentially uncontrolled number of recursion steps as encountered in GNN-0.

The most considerable modification of GG-NN to GNN-0 is the adoption of GRU with the size-fixed layer unrolling. This work removes the confinement for contraction mapping and mitigates the long-range dependency problems. However, without the theoretical guarantee of equilibrium, RBP becomes unsuitable for the task, and BPTT is used instead. This then increases this method's memory footprint, and one can see a subtle trade-off between GG-NN and GNN-0.

3.2 Spatial-based Convolutional GNNs

Analogous to image convolutions, it is possible to generalize the definition of convolution to the graph domain. Spatial-based convolutional GNNs share a lot in common with recurrent GNNs. They mainly differ as the spatial methods use **different** filters in each layer, while in the recurrent GNNs, the same layer of neural networks are applied to the graph until an equilibrium (GNN-0) or a limited number of steps (GG-NN) is reached. This difference is illustrated as in Figure 2 (B) and (C).

Convolutional neural networks (CNNs) have proven successful in capturing shift-invariant local information and features of images. Suppose we treat an image as a graph and see each spatial location as a graph node and the pixel values (or the multi-channel feature vector, as in the intermediate layers) as the node embedding. In that case, a convolution on it can be seen as an aggregation of neighbor embeddings. We now give an example of how one could generalize image convolution to graph convolution.

Image convolution. The convolution on an image $I(x, y)$ by a weighted filter $\omega(x, y)$ with size $a \times b$ is defined as:

$$I'(x, y) = \omega * I(x, y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx, dy) I(x + dx, y + dy), \tag{11}$$

where $I'(x, y)$ is the filtered image. This operation can be thought as first flipping the image both horizontally and vertically, then 'sliding' the filter on the image to conduct point-wise product followed by sum. We now give an example. Let $I(x, y)$ to be an 5×5 single-channel image. We first flip it and it results in $\tilde{I}(x, y)$ and we explicit write out the pixel values of the $[2, 2]$ subimage of $\tilde{I}(x, y)$:

$$\tilde{I}(x, y) = \begin{pmatrix} p_0 & p_1 & p_2 & \text{pixel} & \text{pixel} \\ p_3 & p_4 & p_5 & \text{pixel} & \text{pixel} \\ p_6 & p_7 & p_8 & \text{pixel} & \text{pixel} \\ \text{pixel} & \text{pixel} & \text{pixel} & \text{pixel} & \text{pixel} \\ \text{pixel} & \text{pixel} & \text{pixel} & \text{pixel} & \text{pixel} \end{pmatrix}, \tag{12}$$

and a 3×3 filter $\omega(x, y)$ be written as:

$$\omega(x, y) = \begin{pmatrix} w_0 & w_1 & w_2 \\ w_3 & w_4 & w_5 \\ w_6 & w_7 & w_8 \end{pmatrix}. \tag{13}$$

Then after the filtering, the value at the location $(2, 2)$ of the filtered image is:

$$I'(2, 2) = p_0 w_0 + p_1 w_1 + p_2 w_2 + \ldots = \sum_{i=0}^{8} p_i w_i. \tag{14}$$

Note this is exactly a weighted sum of the pixel values surrounding spatial location p_4. The shape and size of the filter defines the scope of this summation. Now if we consider the image to be a graph, and each spatial location represents a node with the pixel value being the node feature, and add an edge from each node to its surrounding nodes (including diagonal and anti diagonal), we can rewrite Equation 14 to be:

$$I'(2,2) = \sum_{p_i \in \mathcal{N}(p_4)} p_i w_i, \tag{15}$$

or in general:

$$I'(x,y) = \sum_{p_i \in \mathcal{N}(\tilde{I}(x,y))} p_i w_i. \tag{16}$$

If the image is multi-channel, meaning each spatial location has, instead of a scalar, a vector of pixel values. The convolution can be written as:

$$I'(x,y) = \sum_{p_i \in \mathcal{N}(\tilde{I}(x,y))} \mathbf{p}_i \mathbf{w}_i. \tag{17}$$

With this convenient form, we may now move on to define the spatial graph convolution in analogy. However, there are two distinctions between an image and a graph: 1. image nodes have a fixed number of neighbors, defined by the filter. 2. image node neighbors have an implicit ordering, also defined by the ordering of the filter. In a graph, the number of neighbors could vary from node to node, and the neighbors may be unordered (not positional).

For graphs where we can safely assume a maximum degree and the ordering of the neighbors can be defined, we may be able to define graph convolution similarly to image convolution, but this may not work for general graphs. Moreover, in some applications, compared to the ordering of neighbors, one may care more about the types of edges connecting these neighbors. Hence, edge features may serve as a way to distinguish between neighbors, and naturally, we would want to share the filter weights for the same type of edges throughout the graph.

3.2.1 *Neural Network for Graph (NN4G).*

NN4G [30] was among the first works to introduce convolution operation on a graph. When it comes to the definition of graph convolution, it is tempting to try to use the definition in Equation 17. However, as mentioned above, this definition would require the graph to be positional, and each node has a fixed degree. If we relax the requirements of a fixed degree but assume that there is a finite set of edges (the edges could be distinguished by the edge features, their directions, or by order in positional graphs), we can then share the weights within each edge type. This is called the stationary assumption. The definition in this scenario is as follows.

Spatial graph convolution with stationary assumption. The convolution of a filter ω on a graph G is defined as:

$$(\omega * \mathbf{X}_V)(v) = \sum_{u \in \mathcal{N}(v)} \mathbf{w}_{e_{uv}} \mathbf{h}_u, \tag{18}$$

where $\mathbf{w}_{e_{uv}}$ is a learnable per-edge type vector of weights. Note this form is very similar to GG-NN's way of handling positional information. However, this definition of convolution is not applicable to general graphs where the edges are indistinguishable from each other. Therefore, NN4G uses a simplified but more general definition for convolution.

General spatial graph convolution. Without the stationary assumption, convolution is defined as:

$$(\omega * \mathbf{X}_V)(v) = \mathbf{w} \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u, \tag{19}$$

where $\mathbf{w}$ is a learnable weights vector shared for all edges and nodes. The contributions from the neighbors are summed together directly, as the edges are assumed to be indistinguishable.

Based on the general spatial graph convolution, we can now go on to define the update rule of each node's hidden states:

$$\mathbf{h}_v^k = \mathbf{W}_V^k \; \mathbf{x}_v + \mathbf{W}_U^k \sum_{u \in \hat{\mathcal{N}}(v)} \mathbf{h}_u^{k-1}, \tag{20}$$

where $\mathbf{W}_V^k$ is the k-th layer weight matrix associated with the node features while $\mathbf{W}_U^k$ is the weight matrix associated with the aggregated neighbor features.

Commonly, the update rule can be re-written in matrix form to express the convolution on the entire graph:

$$\mathbf{H}_V^k = \mathbf{W}_V^k \; \mathbf{X}_V + \mathbf{W}_U^k \mathbf{H}_V^{k-1} \mathbf{A}, \tag{21}$$

where $\mathbf{H}_V^k$, $\mathbf{H}_V^{k-1}$, and $\mathbf{X}_V$ are stacked up horizontally (each vector forms a column in the matrx) by $\mathbf{h}_v^k$, $\mathbf{x}_v$, and $\mathbf{h}_u^{k-1}$, respectively. $\mathbf{A}$ is the adjacency matrix.

Similar to recurrent GNNs, NN4G uses another neural network to generate predictions based on the extracted node embeddings, the loss of which is then backpropagated for training. Unlike the recurrent GNNs, NN4G contains only feed-forward neural networks, and neither BPTT nor RBP is needed. However, like GG-NN, it contains a fixed number of layers, and it needs to keep all intermediate hidden states for backpropagation, and the memory footprint could be high.

NN4G and other spatial-based convolutional GNN are closely related to the recurrent GNNs. Collectively, they are called spatial methods as they directly work on the graph locally and extract the spatial information, in contrast to the spectral methods, which we will go over later. They are also designed to capture the shift-invariant features through weight sharing at different spatial locations. The major difference between them is spatial convolutional GNNs use multiple layers of different filters to gradually extract, ideally, higher-level features, whereas recurrent GNNs apply the same layer recurrently with the assumption of information propagation and equilibrium.

3.2.2 *GraphSage.*

So far, all methods we covered relied on batch gradient descent, in a sense that the entire graph must be iterated before one step of updates can be made to the model. This can hinder the training efficiency and scalability dramatically as the graph could be excessively large. More importantly, it could lead to many memory-related issues as the intermediate results of the entire graph need to be present before backpropagation.

GraphSage [18] can mitigate these challenges. It proposes a batched training scheme through fix-sized sampling of the neighbors during each aggregation. The update rule for a spatial convolutional GNN can be written as:

$$\mathbf{h}_v^k = \mathbf{W}_V^k \; \mathbf{x}_v + \mathbf{W}_U^k \sum_{u \in \mathcal{S}(\hat{\mathcal{N}}(v))} \mathbf{h}_u^{k-1}. \tag{22}$$

The summation can be substituted with any commutative and associative aggregation.

It then goes on to adopt the mini-batch SGD for training. During a training iteration, it first samples a batch of nodes to serve as the root nodes and fetch all of their k-hop neighbors, from which the node embedding will be extracted, and losses will be calculated. Then during the forward propagation, it keeps each aggregation size fixed by sampling the neighbors. With this tweak to the update rule, GraphSage has constant time and space complexity on a fixed graph architecture.

Neighbor explosion problem. Through sampling, GraphSage can outperform prior arts by up to 100x [18]. However, the challenge has not been fully solved. GraphSage still requires storing all of the related neighbors of the root node during training for each batch. With just one convolution layer, a fix-sized sample of all the 1-hop neighbors' features needs to be stored. This is a relatively small and constant amount of data and can be handled easily. Then if we propagate to the second convolution layer, we will then need to calculate each 1-hop neighbor's node embedding generated by the first layer, which will then require storing their 1-hop neighbors as well. This indicates we will need to store in total the k-hop neighbors of one root node to calculate its embedding after the k-th layer. The size of memory consumption grows exponentially with the number of layers and can hinder the scalability. This issue is referred to as the neighbor explosion problem [10].

3.2.3 *Fast Learning with Graph Convolutional Network (FastGCN).*

Sampling proves to be a somewhat effective tool when handling scalability issues. However, GraphSage's neighbor sampling is still subject to the neighbor explosion problem. As the depth of the GNN increases, the sampled subgraph exponentially grows, and it can quickly become a large portion of the original graph, diminishing the goal of reducing the memory footprint. This problem stems from the fact that the graph data is non-i.i.d. A sampling of vertices will also involve their neighbors.

FastGCN [9] proposes to improve the efficiency and scalability of spatial convolutional GNN by introducing an i.i.d. assumption and then samples the original graph nodes as if they were independently drawn from a distribution. It draws random sampling in each convolutional layer and this can be seen as an approximation to convolution. The update rule is written as:

$$\mathbf{h}_v^k = \mathbf{W}_V^k \; \mathbf{x}_v + \mathbf{W}_U^k \sum_{u \in \hat{\mathcal{N}}(v) \wedge u \in \mathcal{S}^{k-1}(V)} \mathbf{h}_u^{k-1}, if \; v \in \mathcal{S}^k(V), \tag{23}$$

where $\mathcal{S}^{k-1}(V)$ is the sampled nodes in the k-th layer. If $v \notin \mathcal{S}^k(V)$, $\mathbf{h}_v^k$ will not be calculated.

With this sampling of nodes, time and space complexity now become the summation of sample sizes of every layer and therefore linear, instead of exponential, to the number of layers. On the other hand, this sampling is radical, and the inter-connections between layers could be scarce as each node has an independent sample of nodes, which may or may be connected with the previous and next layer's sample.

3.3 Spectral-based Convolutional GNNs

Spectral-based convolutional GNNs, or simply spectral GNNs, are stemmed from the world of graph signal processing and have many theoretical backgrounds. They usually assume the graph to be undirected. Instead of defining graph convolution analogous to image convolutions, these GNNs conduct convolution relying on the convolution theorem. It has been shown in [22] that after a series of approximation and normalization, spectral-based methods can also be reduced to spatial-based methods.

3.3.1 *Spectral-based Convolutional GNN.*

Spectral-GNN [6] was one of the first spectral-based GNNs. We now have a closer look at how spectral-based methods work. They treat the node features $\mathbf{X}$ as graph signal and define a Fourier transformation based on the structure of the graph. Let $\deg(v)$ be a function to return the degree of node v. Given the adjacency matrix of the graph $\mathbf{A}$, define the node degrees matrix to be a diagonal matrix $\mathbf{D}$:

$$\mathbf{D}_{ii} = \sum_j \mathbf{A}_{ij} = \deg(v_{ij}). \tag{24}$$

Then define the symmetric normalized Laplacian of the graph to be:

$$\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}, \tag{25}$$

where $\mathbf{I}$ is a unit matrix. Graph Laplacian is a matrix representation of the graph. The element of it can be written as:

$$\mathbf{L}_{ij} = \begin{cases} 1 & i = j \text{ and } \deg(v_{ij}) \neq 0, \\ -\dfrac{1}{\sqrt{\deg(v_i)\deg(v_j)}} & i \neq j \text{ and } v_j \in \mathcal{N}(v_i), \\ 0 & \text{otherwise.} \end{cases} \tag{26}$$

Graph Laplacian is a real symmetric positive semidefinite. The eigendecomposition of $\mathbf{L}$ yields:

$$\mathbf{L} = \mathbf{Q} \; \mathbf{\Lambda} \; \mathbf{Q}^\intercal, \tag{27}$$

where $\mathbf{Q}$ is an orthogonal matrix whose columns are the eigenvectors, and $\mathbf{\Lambda}$ is a diagonal matrix composed of the eigenvalues. $\mathbf{Q}$ is then used to define the graph Fourier transformation on the graph signal (node features).

Graph Fourier transformation $\mathcal{F}$ and inverse transformation $\mathcal{F}^{-1}$:

$$\hat{\mathbf{X}}_V = \mathcal{F}(\mathbf{X}_V) = (\mathbf{Q}^\intercal \; \mathbf{X}_V^\intercal)^\intercal = \mathbf{X}_V \; \mathbf{Q}, \tag{28}$$

$$\mathbf{X}_V = \mathcal{F}^{-1}(\hat{\mathbf{X}}_V) = \hat{\mathbf{X}}_V \; \mathbf{Q}^\intercal, \tag{29}$$

where $\hat{\mathbf{X}}_V$ is the transformed node features. Spectral methods then uses the convolution theorem to compute convolution given the Fourier transformation. The spatial convolution on the graph signal can be expressed as a point-wise product in the spectral space. As of now, we start with a simplified single-channel scenario when

$D_v = 1$, then $\mathbf{X}_V \in \mathbb{R}^{1\times|V|}$, i.e., row vector. Assume the filter to be $\mathbf{\Theta} \in \mathbb{R}^{1\times|V|}$, we have the definition of convolution:

Spectral graph convolution:

$$\begin{aligned}\mathbf{\Theta} * \mathbf{X}_V &= \mathcal{F}^{-1}(\mathcal{F}(\Omega) \odot \mathcal{F}(\mathbf{X}_V)) \\ &= ((\mathbf{\Theta}\ \mathbf{Q}) \odot (\mathbf{X}_V\ \mathbf{Q}))\ \mathbf{Q}^\intercal,\end{aligned} \tag{30}$$

the output would also be in $\mathbb{R}^{1\times|V|}$. To simplify it, we can define:

$$\hat{\mathbf{\Theta}} = \text{diag}(\mathbf{\Theta}\ \mathbf{Q}), \tag{31}$$

where diag(.) is function that maps a vector to a diagonal matrix. Then Equation 30 can be re-written as:

$$\mathbf{\Theta} * \mathbf{X}_V = \mathbf{X}_V\ \mathbf{Q}\ \hat{\mathbf{\Theta}}\ \mathbf{Q}^\intercal. \tag{32}$$

This already gives us a way to calculate the convolution on only one of the channels, we now generalize it to multiple channels and filters [6]. The update rule can be written as:

$$\begin{aligned}\mathbf{H}^k[c,:] &= \sum_i \mathbf{\Theta}^{c,i,k} * \mathbf{H}^{k-1}[i,:] \\ &= \sum_i \mathbf{H}^{k-1}[i,:]\ \mathbf{Q}\ \hat{\mathbf{\Theta}}^{c,i,k}\ \mathbf{Q}^\intercal,\end{aligned} \tag{33}$$

where $\hat{\mathbf{\Theta}}^{c,i,k}$ is in the k-th layer, the c-th filter's sub-filter that operates the i-th channel of the previous layer's outputs.

Comparison between spatial and spectral methods: It can be seen that spectral-based convolution is a global operation that is defined on a per-channel basis. For each input channel, there is an associated filter that operates on the entire graph. Filtered results from each previous channel are summed together to form one output channel. Thus, for each convolution, it requires the entire graph to calculate. This is drastically different from the spatial methods, which operate on spatial locations and share the filter weights.

Despite their theoretical importance, spectral methods have many issues that limit their efficiency and scalability. There are three major drawbacks :

1. Requirements for eigendecomposition. Eigendecomposition, generally speaking, is a costly $O(|V|^3)$ operation, and it requires access to the entire graph at the training time, which may not be feasible, sometimes not even possible.
2. Graph Fourier transformation is graph-dependent, meaning knowledge transfer from an existing graph to new nodes or to a completely new graph is non-trivial.
3. Difficult to apply to dynamic graphs. Adding or deleting nodes from the graph would trigger a recalculation of the eigendecomposition and retraining of the GNN.

Due to these constraints, spatial methods are generally more scalable and flexible than spectral methods.

3.3.2 *ChebNet and Graph Convolutional Networks (GCN).*

ChebNet [13], local spectral-GNN [19], and GCN [22] are among the works that attempt to simplify spectral method via approximation, because of the issues mentioned above.

We can approximatee the global filter by Chebyshev polynomials of $\mathbf{\Lambda}$, an approximation approach introduced in [13, 19], so that the filters are no long global. The convolution can be approximated as K-truncated Chebyshev polynomials:

$$\hat{\mathbf{\Theta}} \approx \sum_i^K \theta_i T_i(\mathbf{\Lambda}), \tag{34}$$

where θ_i is learnable weight and T_i is the Chebyshev polynomial, defined as: $T_i(x) = 2xT_{i-1}(x) - T_{i-2}(x), T_0(x) = 1, T_1(x) = x$. Plug it in Equation 32, we can obtain:

$$\mathbf{\Theta} * \mathbf{X}_V = \mathbf{X}_V\ \sum_i^K \theta_i T_i(\mathbf{L}), \tag{35}$$

where the multiplication with $\mathbf{Q}$ and $\mathbf{Q}^\intercal$ have been absorbed in to L, saving a $O(|V|^2)$ matrix multiplication. Meanwhile, now the polynomial becomes about the Laplacian and only up to order K, this also implies locality, based on a lemma [19]:

Lemma 1. *Let* $\mathbf{L}$ *be the graph Laplacian (normalized or non-normalized) and* $s > 0$ *an integer. For any two nodes* m *and* n, *the length of the shortest path between* m *and* n *is larger than* s, *then* $\mathbf{L}^s_{mn} = 0$.

Therefore, for any node that is more than K-hop away from a given central node of convolution, it does not contribute since the Laplacian is zero, and therefore this convolution becomes local.

Furthermore, if we take only $K = 1$ and assume $\theta_0 = \theta_1 = \theta$, we have [22]:

$$\mathbf{\Theta} * \mathbf{X}_V = \theta \mathbf{X}_V(\mathbf{I} + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}). \tag{36}$$

The scope of convolution will then be limited to only the 1-hop neighbors. Now if we enable multi-channel input and upgrade θ to be a matrix $\bar{\mathbf{\Theta}}$ consisting of multiple filters, and define $\bar{\mathbf{A}} = \mathbf{I} + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$, the update rule can be written as:

$$\mathbf{H}^k_V = \bar{\mathbf{\Theta}}^k\ \mathbf{H}^{k-1}_V \bar{\mathbf{A}}, \tag{37}$$

which shares the same form of Equation 20, except for the normalized adjacency matrix $\bar{\mathbf{A}}$, which can improve the numerical stability of GCN. Finally, we are able to draw the connection between spectral methods and spatial methods and unify them under the same framework. Spatial methods can be seen as a first-order approximation of the spectral methods.

4 GNN SYSTEMS

So far, we have seen that GNNs are largely related to classical deep learning methods such as CNN or RNN. However, the graph data structure adds complexity as graph data can be irregular and connected explicitly. To apply to irregular data and capture a graph's structural information, various GNNs have been proposed.

Now a natural question is: *how do we build software systems to support the GNN-based graph analytics better?* And most importantly, with training/inference being the most computationally intensive part, *how do we build more efficient and scalable systems for GNN training/inference*? In the past, a huge amount of effort has been put into general ML systems for classical deep learning. Most of theese are dataflow systems [1, 32, 45]. Now it is time to consider how to re-use or develop new techniques for scalable and

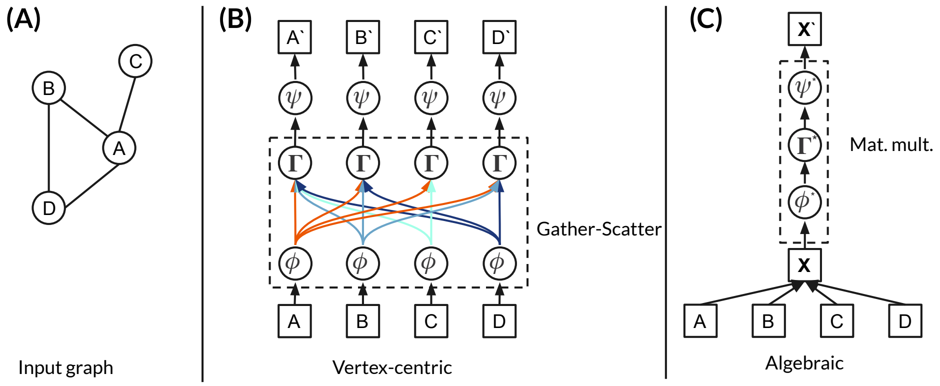

Figure 3: An example of two different approaches for GNN execution. (A): Input graph. (B): Vertex-centric approach. (C): Algebraic approach.

efficient graph deep learning. As an emerging topic, GNN system is gradually gaining popularity.

As of now, most of the GNN system research mainly focus on the spatial methods. Majority of spatial methods can be expressed via a general update rule:

$$\mathbf{h}_v^k = \psi(\mathbf{x}_v^k, \Gamma_{u\in\mathcal{N}(v)} \phi(\mathbf{h}_v^{k-1}, \mathbf{h}_u^{k-1}, \mathbf{x}_{e_{vu}})), \tag{38}$$

where ψ, ϕ, Γ are potentially learnable and differentiable functions, Γ is further required to be commutative and associative. ψ is called the update function, ϕ the message function and Γ the aggregate function.

4.0.1 *PyTorch Geometric (PyG) and NeuGraph.*

Equation 38 fits in the paradigm of vertex-centric programming, which is a common programming model in the graph processing world [16, 27, 29] and very easy to parallelize. PyG [14] follows this pattern and builds GNNs as dataflow graphs on top of PyTorch [32], a dataflow system. Figure 3 (B) shows an example computational graph generated. Each (vectorized) operation is then put on a separate processor to achieve parallelism. Built on top of PyTorch, PyG can easily utilize the features PyTorch provides, such as autograd, GPU acceleration, etc.

NeuGraph [26] is another GNN system built on top of a dataflow system and designed for a single-node multi-GPU setting. It extends the GAS [16] programming model and partitions the computational graph to avoid GPU OOM issues. It further exploits GPU-to-GPU communication to avoid latencies of GPU-to-DRAM.

4.0.2 *Deep Graph Library (DGL).*

DGL [41] takes a different approach for the GNN execution, especially for the message passing (gather-scatter) part. It takes an algebraic approach for the execution by expressing GNN as sparse matrix multiplications (SpMM). This is analogous to those algebraic approaches to general graph analytics tasks [21].

Figure 3 (C) gives an example of the algebraic approach taken by DGL; each step is summarized as linear algebra and executed as SpMM, which is executed via DGL's own specialized kernel. DGL is built upon existing deep learning systems, and the user can also write user-defined functions to extend the built-in functions. DGL describes itself as a framework independent system, that is, independent of the underlying deep learning system. It has full support for several popular frameworks such as Tensorflow or PyTorch as backend. Backpropagation can also be done via another SpMM, and DGL functions are registered in the underlying deep learning framework to take advantage of autograd.

The algebraic approach opens possibilities of operator fusion. For instance, consider the update rule for NN4G in Equation 20, it can also be represented as Equation 21 with purely matrix multiplications. Note in this form, the message function can be fused together with the aggregation function, and there is no need to materialize the intermediate messages.

Comparison between PyG and DGL. Generally speaking, the vertex-centric approach has advantages on low-degree graphs, and when the graph data is not coalesced and data access could become a bottleneck for the algebraic approaches. On the contrary, algebraic approaches do not need to materialize the huge amount of messages, leading to system crashes. Algebraic approaches could fuse the message function with the aggregation, and DGL tends to perform better on higher-degree graphs.

5 CONCLUSION AND DISCUSSION

In this paper, we surveyed GNN research, ranging from pioneering works to milestone papers. GNN research is gaining enormous attention, and scalability and efficiency have been one of the major topics. We now discuss the potential research directions on the scalability and efficiency of GNNs.

On the algorithmic research of GNNs. Sampling has proven to be a very effective way of increase GNN efficiency. However, graph sampling is non-trivial because of the non-i.i.d. nature of graph data. Moreover, ignoring the interconnections may lead to degraded accuracy performance. Therefore, sophisticated sampling that considers the connections while still offering a boost in efficiency and scalability is a very important problem. Similarly, graph coarsening, which distills the original graph to be smaller subgraphs, can also significantly improve efficiency.

On the GNN system research. One natural direction for GNN system research is to scale out, i.e., distributed processing, which is receiving attention. On the other hand, they face even more problems like graph partitioning and load balancing. As mentioned before, the graph data is non-i.i.d., thus both sampling and partitioning are difficult to do. Graph partitioning for GNN itself is already a hard problem, let alone considering load-balancing simultaneously. The other important direction is compilers that can generate fused kernels for GNN operations, given a set of arbitrary user-defined functions. Such innovation would be especially useful for algebraic approaches. Meanwhile, existing general-purpose graph analytics systems may have more to offer, especially on the front of scalability and when the system needs to operate on huge graphs.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283. USENIX Association, 2016.

[2] L. B. Almeida. *A Learning Rule for Asynchronous Perceptrons with Feedback in a Combinatorial Environment*, page 102–111. IEEE Press, 1990.

[3] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Process. Mag.*, 34(4):18–42, 2017.

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. 2020.

[5] J. Bruna and S. Mallat. Invariant scattering convolution networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1872–1886, 2013.

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

[7] N. D. Cao and T. Kipf. Molgan: An implicit generative model for small molecular graphs. *CoRR*, abs/1805.11973, 2018.

[8] S. Cao, W. Lu, and Q. Xu. Deep neural networks for learning graph representations. In *AAAI*, pages 1145–1152. AAAI Press, 2016.

[9] J. Chen, T. Ma, and C. Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In *ICLR (Poster)*. OpenReview.net, 2018.

[10] J. Chen, J. Zhu, and L. Song. Stochastic training of graph convolutional networks with variance reduction. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 941–949. PMLR, 2018.

[11] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734. ACL, 2014.

[12] H. Dai, Z. Kozareva, B. Dai, A. J. Smola, and L. Song. Learning steady-states of iterative algorithms over graphs. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 1114–1122. PMLR, 2018.

[13] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3837–3845, 2016.

[14] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

[15] C. Gallicchio and A. Micheli. Graph echo state networks. In *IJCNN*, pages 1–8. IEEE, 2010.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30. USENIX Association, 2012.

[17] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

[18] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[19] D. K. Hammond, P. Vandergheynst, and R. Gribonval. Wavelets on graphs via spectral graph theory. *CoRR*, abs/0912.3848, 2009.

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

[21] J. Kepner and J. R. Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*, volume 22 of *Software, environments, tools*. SIAM, 2011.

[22] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR (Poster)*. OpenReview.net, 2017.

[23] A. Kuznetsova, H. Rom, N. Alldrin, J. R. R. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari. The open images dataset V4. *Int. J. Comput. Vis.*, 128(7):1956–1981, 2020.

[24] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated graph sequence neural networks. In *ICLR (Poster)*, 2016.

[25] Y. Li, R. Yu, C. Shahabi, and Y. Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In *ICLR (Poster)*. OpenReview.net, 2018.

[26] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. Neugraph: Parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference*, pages 443–458. USENIX Association, 2019.

[27] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146. ACM, 2010.

[28] S. Mallat. Group invariant scattering. *Communications on Pure and Applied Mathematics*, 65(10):1331–1398, 2012.

[29] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2), Oct. 2015.

[30] A. Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Trans. Neural Networks*, 20(3):498–511, 2009.

[31] S. Nakandala, Y. Zhang, and A. Kumar. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.*, 13(11):2159–2173, 2020.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

[33] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: online learning of social representations. In *KDD*, pages 701–710. ACM, 2014.

[34] F. J. Pineda. Generalization of back propagation to recurrent and higher order neural networks. In *NIPS*, pages 602–611. American Institue of Physics, 1987.

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

[36] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. *AI Mag.*, 29(3):93–106, 2008.

[37] A. Sergeev and M. D. Balso. Horovod: Fast and Easy Distributed Deep Learning in TF. *arXiv preprint arXiv:1802.05799*, 2018.

[38] L. Tang and H. Liu. Relational learning via latent social dimensions. In *KDD*, pages 817–826. ACM, 2009.

[39] M. Tygert, J. Bruna, S. Chintala, Y. LeCun, S. Piantino, and A. Szlam. A mathematical motivation for complex-valued convolutional networks. *Neural Comput.*, 28(5):815–825, 2016.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[41] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[42] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. Dynamic graph CNN for learning on point clouds. *ACM Trans. Graph.*, 38(5):146:1–146:12, 2019.

[43] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Trans. Neural Networks Learn. Syst.*, 32(1):4–24, 2021.

[44] J. Xu. Representing big data as networks: New methods and insights. *CoRR*, abs/1712.09648, 2017.

[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for in-memory Cluster Computing. In *NSDI*, 2012.

[46] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D. Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. In *UAI*, pages 339–349. AUAI Press, 2018.

[47] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

[48] M. Zitnik and J. Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinform.*, 33(14):i190–i198, 2017.