Analyzing Performance Bottleneck in Graph Neural Network Training: An Experimental View

Zhaokang Wang, Yunpan Wang, Chunfeng Yuan, Yihua Huang*

State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China

Abstract

This is the abstract.

Keywords: Keyword 1

1. Introduction

This is a survey [1].

2. Review of Graph Neural Networks

In this section, we introduce the concepts related to the graph neural network (GNN, for short) and breifly survey typical graph neural networks. We denote a simple graph $\mathcal{G}$ as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are the vertex set and the edge set of $\mathcal{G}$, respectively. Let $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ as the number of vertices/edges. We use v_i $(0 \leq i < n)$ to denote a vertex and $e_{i,j} = (v_i, v_j)$ to denote the edge pointing from v_i to v_j. The adjacency set of v_i is $\mathcal{N}(v_i) = \{v|(v_i, v) \in \mathcal{E}\}$. We denote a *vector* with a bold lower case letter like $\boldsymbol{x}$ and a *matri*x with a bold upper case letter like $\boldsymbol{X}$.

2.1. General Structure of Graph Neural Networks

As illustracted in Figure 1, a typical GNN can be decomposed into three parts: an input layer + several GNN layers + a prediction layer.

*Corresponding author

Email address: {wangzhaokang, wangyp}@smail.nju.edu.cn, {cfyuan, yhuang}@nju.edu.cn (Zhaokang Wang, Yunpan Wang, Chunfeng Yuan, Yihua Huang)

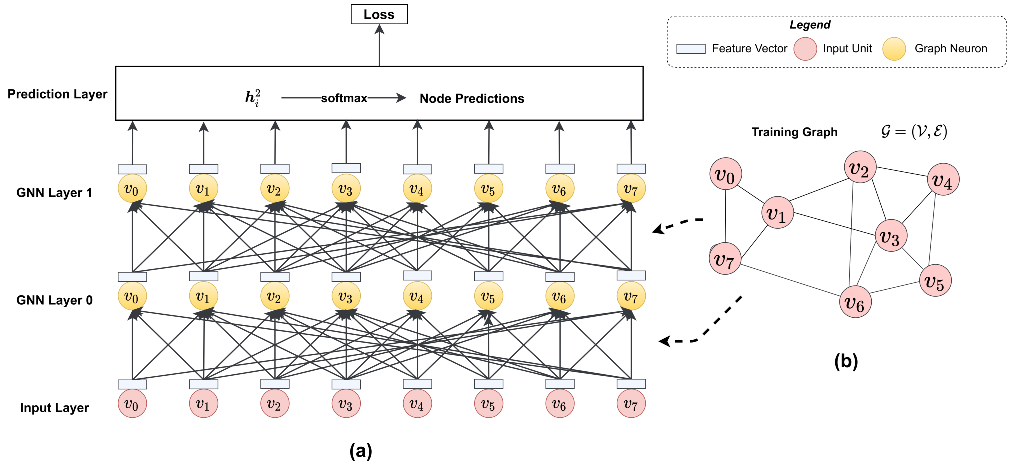
Loss
Legend
Feature Vector
Input Unit
Graph Neuron
Prediction Layer
h_i^2 softmax Node Predictions
Training Graph
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$
GNN Layer 1
GNN Layer 0
Input Layer
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
(a)
(b)

Figure 1: Structure of a typical graph neural network. (a) Demo GNN, (b) Demo graph. The target application is the node classification. The demo GNN has two GNN layers.

A GNN receives a graph $\mathcal{G}$ as the input. Every vertex v_i in $\mathcal{G}$ is attached with a feature vector $\boldsymbol{x}_i$ to describe the properties of the vertex. The edges of $\mathcal{G}$ may also be attached with feature vectors $\boldsymbol{e}_{i,j}$ The input layer of a GNN receives feature vectors from all vertices and passes them to GNN layers.

A GNN layer consists of n graph neurons, where n is the number of vertices in $\mathcal{G}$. Each graph neuron corresponds to a vertex in $\mathcal{G}$. In the first GNN layer (Layer 0), the graph neuron of the vertex v_i collects input feature vectors of itself and the vertices $\boldsymbol{x}_j$ that are adjacent to v_i in $\mathcal{G}$ (i.e., $v_j \in \mathcal{N}(v_i)$) from the input layer. After aggregating input feature vectors and applying non-linear transformation, the graph neuron outputs a hidden feature vector $\boldsymbol{h}_i^1$ for v_i. Take the demo DNN in Figure 1(a) as the example. Since $\mathcal{N}(v_3) = \{v_1, v_2, v_4, v_5, v_6\}$, the graph neuron of v_1 at layer 0 collects the feature vectors $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4, \boldsymbol{x}_5, \boldsymbol{x}_6\}$ from the input layer and outputs $\boldsymbol{h}_1^1$. Different GNNs mainly differ in the graph neurons that they use. We elaborate on their details later.

The connection between the input layer and the first GNN layer is determined by the topology of $\mathcal{G}$. In the traditional neural networks, neurons of neighboring layers are fully connected. In GNNs, two graph neurons are connected only if their corresponding vertices have an edge between them in $\mathcal{G}$. Most real-world graphs are very *sparse*, i.e. $|\mathcal{E}| \ll |\mathcal{V}|^2$.

In the next GNN layer (Layer 1), the graph neuron of v_i collects the hidden feature vectors of itself $\boldsymbol{h}_i^1$ and its neighbors ($\boldsymbol{h}_j^1$ with $v_j \in \mathcal{N}(v_i)$) from the *previous* GNN layer. Based on the collected hidden vectors, the graph neuron in Layer 1 outputs a new hidden feature vector $\boldsymbol{h}_i^2$ for v_i. Though there are only two GNN layers in Figure 1, a GNN allows to stack more GNN layers to support deeper graph analysis.

Assume there are L GNN layers. The last GNN layer (Layer $L-1$) outputs a hidden feature vector $\boldsymbol{h}_i^L$ for every vertex v_i. As an embedding vector, $\boldsymbol{h}_i^L$ encodoes the knowledge learned from the input layer and all the previous GNN layers. Since $\boldsymbol{h}_i^L$ is affected by v_i and the vertices in the L-hop neighborhood of v_i, analyzing a graph with a *deeper* GNN means analyzing each vertex with a *wider* scope.

The hidden feature vectors $\boldsymbol{h}_i^L$ of the last GNN layer are fed to the prediction layer to generate the output of the whole GNN. The prediction layer is a standard nerual network. The structure of the prediction layer depends on the prediction task of the GNN. Take the node classification task as the example, as shown in Figure 1. The node classification predicts a label for every vertex in $\mathcal{G}$. In this case, the prediction layer can be a simple softmax layer with $\boldsymbol{h}_i^L$ as the input and a vector of probabilities as the output. If the prediction task is edge prediction, the hidden feature vectors of two vertices are concatenated and fed into a softmax layer. If we need to predict a label for the whole graph, a pooling (max/mean/...) layer is added to generate an embedding vector for the whole graph and the embedding vector is used to produce the final prediction.

Supporting end-to-end training is a prominent advantage of GNN, compared with other graph-based machine learning methods. We can calculate the gradients of the loss function on the model parameters from the prediction layer directly. With the help of the back proporgation technique, the gradient is propogated from the prediction layer back to the previous GNN layers layer by layer. The model parameters are updated with a gradient descent optimizer like

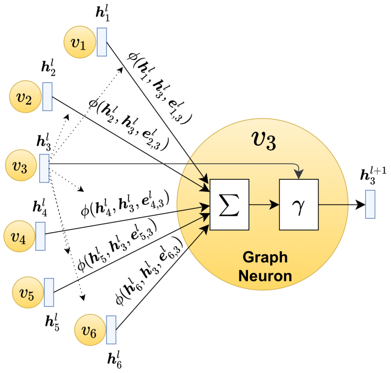
$\boldsymbol{h}_1^l$
v_1
$\phi(\boldsymbol{h}_1^l, \boldsymbol{h}_3^l, \boldsymbol{e}_{1,3}^l)$
$\boldsymbol{h}_2^l$
v_2
$\phi(\boldsymbol{h}_2^l, \boldsymbol{h}_3^l, \boldsymbol{e}_{2,3}^l)$
v_3
$\boldsymbol{h}_3^l$
v_3
$\boldsymbol{h}_3^{l+1}$
Σ
γ
$\boldsymbol{h}_4^l$
$\phi(\boldsymbol{h}_4^l, \boldsymbol{h}_3^l, \boldsymbol{e}_{4,3}^l)$
v_4
$\phi(\boldsymbol{h}_5^l, \boldsymbol{h}_3^l, \boldsymbol{e}_{5,3}^l)$
Graph Neuron
$\phi(\boldsymbol{h}_6^l, \boldsymbol{h}_3^l, \boldsymbol{e}_{6,3}^l)$
v_5
$\boldsymbol{h}_5^l$
v_6
$\boldsymbol{h}_6^l$

Figure 2: Graph neuron of v_3 at the GNN layer l with the graph $\mathcal{G}$ in Figure 1(b). $\phi/\Sigma/\gamma$ are the message/aggregation/vertex update functions in the message-passing model, respecitvely.

Adam. Except for the input feature vector, there is no need to conduct handworked feature extraction. In a fully parameterized way, the GNN automatically extracts an embedding vector for each vertex from its L-hop neighborhood. The parameters are tuned according to the specific prediction task, leading to high prediction accuracy.

2.2. Graph Neuron and Message-passing Model

Graph neurons are building blocks of a GNN. A GNN layer consists of $|\mathcal{V}|$ graph neurons. Each vertex corresponds to a graph neuron. A graph neuron as shown in Figure 2 is a small neural network. The graph neuron of v_i at layer l receives hidden feature vectors $\boldsymbol{h}_j^l$ from the graph neurons of v_i and its neighbors ($v_j \in \{v_i\} \cup \mathcal{N}(v_i)$) at the previous GNN layer [1]. The graph neuron aggregates the received hidden feature vectors, applies non-linear transformations, and outputs a new hidden feature vector $\boldsymbol{h}_i^{l+1}$.

We follow the message-passing model [2] to formally define a graph neuron.

[1] For the GNN layer 0, graph neurons receive input feature vectors, i.e., $\boldsymbol{h}_i^0 = \boldsymbol{x}_i$

The message-passing model is widely used in the cutting-edge GNN training systems like PyTorch Geometric (PyG) [3] and Deep Graph Library (DGL) [4]. Figure 2 shows the structure of a graph neuron in the message-passing model. Graph neurons at layer l are made of three *differentiable* functions: ϕ^l, Σ^l and γ^l. The graph neuron calculates the output hidden vector $\boldsymbol{h}_i^{l+1}$ by

$$\boldsymbol{h}_i^{l+1} = \gamma^l(\boldsymbol{h}_i^l, \Sigma^l_{v_j \in \mathcal{N}(v_i)} \phi^l(\boldsymbol{h}_i^l, \boldsymbol{h}_j^l, \boldsymbol{e}_{j,i})).$$

ϕ^l is the *message* function. For every incident edge (v_j, v_i) of v_i, ϕ receives the output hidden feature vectors $\boldsymbol{h}_i^l$ and $\boldsymbol{h}_j^l$ of the previous GNN layer and the edge feature vector $\boldsymbol{e}_{j,i}$ as the input. ϕ^l outputs a message vector $\boldsymbol{m}_{j,i}^l$ for every edge (v_j, v_i) at layer l, i.e., $\boldsymbol{m}_{j,i}^l = \phi^l(\boldsymbol{h}_i^l, \boldsymbol{h}_j^l, \boldsymbol{e}_{j,i})$. For v_i, the message vectors $\boldsymbol{m}_{x,j}^l$ with $v_x \in \mathcal{N}(v_i)$ are aggregated by the *aggregation* function Σ^l to produce an aggregated vector $\boldsymbol{s}_i^l$, i.e., $\boldsymbol{s}_i^l = \Sigma^l_{v_j \in \mathcal{N}(v_i)} \boldsymbol{m}_{j,i}^l$. v_i's aggregated vector $\boldsymbol{s}_i^l$ and its hidden vector $\boldsymbol{h}_i^l$ from the previous GNN layer are fed into the *vertex update* function γ^l to calculate the output hidden vector $\boldsymbol{h}_i^{l+1}$ of the current layer l, i.e., $\boldsymbol{h}_i^{l+1} = \gamma^l(\boldsymbol{h}_i^l, \boldsymbol{s}_i^l)$ The end-to-end training requires ϕ^l and γ^l (like multi layer perceptrons and GRU) and Σ_l (like mean, sum, element-wise min/max) are *differentiable* to make the whole GNN differentialble.

Different GNNs adopt different kinds of graph neurons and have different definitions of the three functions. ϕ and Σ are the *edge computation* functions. They are conducted over every edge in $\mathcal{G}$. γ is the *vertex computation* function. It is conducted over every vertex in $\mathcal{G}$. Table 1 and Table 2 list the edge functions and the vertex functions of typical GNNs, respectively. For ChebNet, we report its GNN layer in the tables [2].

[2] A layer of ChebNet consists of K GNN sub-layers and a summation layer, i.e., $\boldsymbol{H}^{l+1} = \sum_{k=1}^{K} \boldsymbol{Z}^{(k)} \boldsymbol{W}^{(k)}$ with GNN layers $\boldsymbol{Z}^{(1)} = \boldsymbol{H}$, $\boldsymbol{Z}^{(2)} = \hat{\boldsymbol{L}}\boldsymbol{H}$, and $\boldsymbol{Z}^{(k)} = 2\hat{\boldsymbol{L}}\boldsymbol{Z}^{(k-1)} - \boldsymbol{Z}^{(k-2)}$. $\boldsymbol{H}^l$ is the matrix of output hidden feature vectors of the layer $l-1$. In the table, we report the GNN sub-layer of ChebNet that calculates $\boldsymbol{Z}^{(k)}$

GNN	Type	Σ	ϕ	Complexity
ChebNet [5]	Spectral	sum	$\boldsymbol{m}_{j,i}^{(k)} = e_{j,i}\boldsymbol{z}_j^{(k-1)}$	$O(d_{in})$
GCN [6]	Spectral	sum	$\boldsymbol{m}_{j,i}^l = e_{j,i}\boldsymbol{h}_j^l$	$O(d_{in})$
AGCN	Spectral	sum	$\boldsymbol{m}_{j,i}^l = \tilde{e}_{j,i}^l\boldsymbol{h}_j^l$	$O(d_{in})$
GraphSAGE	Non-spectral	mean/LSTM	$\boldsymbol{m}_{j,i}^l = \boldsymbol{h}_j^l$	$O(1)$
GraphSAGE-pool	Non-spectral	max	$\boldsymbol{m}_{j,i}^l = \delta(\boldsymbol{W}^l{}_{pool}\boldsymbol{h}_j^l + b^l)$	$O(d_{in} * d_{out})$
Neural FPs	Non-spectral	sum	$\boldsymbol{m}_{j,i}^l = \boldsymbol{h}_j^l$	$O(1)$
SSE	Recurrent	sum	$\boldsymbol{m}_{j,i}^l = [\boldsymbol{h}_i^l \parallel \boldsymbol{h}_j^l]$	$O(d_{in})$
GGNN	Gated	sum	$\boldsymbol{m}_{j,i} = \boldsymbol{W}^l\boldsymbol{h}_j^l$	$O(d_{in} * d_{out})$
GAT	Attention	sum	$\alpha_{j,i}^k = \frac{\exp(LeakyReLU(\boldsymbol{a}^T[\boldsymbol{W}^{l,k}\boldsymbol{h}_j^l \parallel \boldsymbol{W}^{l,k}\boldsymbol{h}_j^l]))}{\sum_{k\in\mathcal{N}(i)}\exp(LeakyReLU(\boldsymbol{a}^T[\boldsymbol{W}^{l,k}\boldsymbol{h}_j^l \parallel \boldsymbol{W}^{l,k}\boldsymbol{h}_k^l]))}$	$O(K * d_{in} * d_{out})$
			Multi-head concatenation : $\boldsymbol{m}_{j,i}^l = \parallel_{k=1}^K \delta(\alpha_{j,i}^k\boldsymbol{W}^{l,k}\boldsymbol{h}_j^l)$	
			Multi-head average : $\boldsymbol{m}_{j,i}^l = \frac{1}{K}\sum_{k=1}^K \delta(\alpha_{j,i}^k\boldsymbol{W}^{l,k}\boldsymbol{h}_j^l)$	
GaAN	Attention	sum,max,mean	$\alpha_{j,i}^k = \frac{\exp(\boldsymbol{a}^T[\boldsymbol{W}_{xa}^{l,k}\boldsymbol{h}_j^l \parallel \boldsymbol{W}_{ya}^{l,k}\boldsymbol{h}_i^l])}{\sum_{k\in\mathcal{N}(j)}\exp(\boldsymbol{a}^T[\boldsymbol{W}_{xa}^{l,k}\boldsymbol{h}_j^l \parallel \boldsymbol{W}_{ya}^{l,k}\boldsymbol{h}_k^l])}$	$O(max(d_a, d_m, d_v) * K * d_{in})$
			$\boldsymbol{m}_{j,i,1}^l = \parallel_{k=1}^K \delta(\alpha_{j,i}^k\boldsymbol{W}_v^{l,k}\boldsymbol{h}_j^l)$	
			$\boldsymbol{m}_{j,i,2}^l = \boldsymbol{W}^l{}_m\boldsymbol{h}_j^l$	
			$\boldsymbol{m}_{j,i,3}^l = \boldsymbol{h}_j^l$	

Table 1: Typical graph neural networks and their edge computation functions.d_{in} and d_{out} are dimensions of the input and output hidden feature vectors, respectively. Blue variables are model parameters to learn. For ChebNet, we report its GNN sub-layer. Because two sum operator are interchangeable and sum operator and concatenation operator are interchangeable, we do it for GAT to explain more clearly in edge computations functions.

GNN	γ	Complexity
ChebNet [5]	$\boldsymbol{z}_i^{(k)} = 2\boldsymbol{s}_i^{(k)} - \boldsymbol{z}_i^{(k-2)}$	$O(d_{in})$
GCN [6]	$\boldsymbol{h}_i^{l+1} = \boldsymbol{W}^l \boldsymbol{s}_i^l$	$O(d_{in} * d_{out})$
AGCN	$\boldsymbol{h}_i^{l+1} = \boldsymbol{W}^l \boldsymbol{s}_i^l$	$O(d_{in} * d_{out})$
GraphSAGE	$\boldsymbol{h}_i^{l+1} = \delta(\boldsymbol{W}^l [\boldsymbol{s}_i^l \parallel \boldsymbol{h}_i^l])$	$O(d_{in} * d_{out})$
GraphSAGE-pool	$\boldsymbol{h}_i^{l+1} = \boldsymbol{s}_i^l$	$O(1)$
Neural FPs	$\boldsymbol{h}_i^{l+1} = \delta(\boldsymbol{W}^{l,\lvert\mathcal{N}(i)\rvert} (\boldsymbol{h}_i^l + \boldsymbol{s}_i^l))$	$O(d_{in} * d_{out})$
SSE	$\boldsymbol{h}_i^{l+1} = (1-\alpha)\boldsymbol{h}_i^l + \alpha\delta(\boldsymbol{W}_1^l \delta(\boldsymbol{W}_2^l \boldsymbol{s}_i^l))$	$O(d_{in} * d_{out})$
GGNN	$\boldsymbol{z}_i^l = \delta(\boldsymbol{W}^z \boldsymbol{s}_i^l + \boldsymbol{b}^{sz} + \boldsymbol{U}^z \boldsymbol{h}_i^l + \boldsymbol{b}^{hz})$	$O(max(d_{in}, d_{out}) * d_{out})$
	$\boldsymbol{r}_i^l = \delta(\boldsymbol{W}^r \boldsymbol{s}_i^l + \boldsymbol{b}^{sr} + \boldsymbol{U}^r \boldsymbol{h}_i^l + \boldsymbol{b}^{hr})$	
	$\boldsymbol{h}_i^{l+1} = tanh(\boldsymbol{W} \boldsymbol{s}_i^l + \boldsymbol{b}^s + \boldsymbol{U}(\boldsymbol{r}_i^l \odot \boldsymbol{h}_i^l) + \boldsymbol{b}^h))$	
	$\boldsymbol{h}_i^{l+1} = (1 - \boldsymbol{z}_i^l) \odot \boldsymbol{h}_i^l + \boldsymbol{z}_i^l \odot \boldsymbol{h}_i^{l+1}$	
GAT	$\boldsymbol{h}_i^{l+1} = \boldsymbol{s}_i^l$	$O(1)$
GaAN	$\boldsymbol{g}_i = \boldsymbol{W}_g^l [\boldsymbol{h}_i^l \parallel \boldsymbol{s}_{i,2}^l \parallel \boldsymbol{s}_{i,3}^l]$	$O(max(K * d_v + d_{in}, 2 * d_{in} + d_m) * d_{out})$
	$\boldsymbol{h}_i^{l+1} = \boldsymbol{W}_o^l [\boldsymbol{h}_i^l \parallel (\boldsymbol{g}_i \odot \boldsymbol{s}_{i,1}^l)]$	

Table 2: Typical graph neural networks and their vertex computation functions. d_{in} and d_{out} are dimensions of the input and output hidden feature vectors, respectively. Blue variables are model parameters to learn. For ChebNet, we report its GNN sub-layer.For Neural FPs, $\boldsymbol{W}^{l,\lvert\mathcal{N}(i)\rvert}$ is the weight matrix for nodes with degree $\lvert\mathcal{N}(i)\rvert$ at layer l.

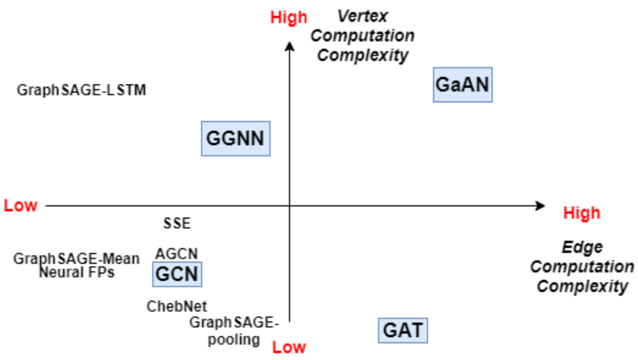

Figure 3: Complexity quadrants of typical GNNs. If two GNNs have the same complexity under the Big-O notation, we further compare their complexity according to how many parameters they need to train.

2.3. Classification of GNNs

Since we focus on analyzing the performance bottleneck in training GNNs, we classify the typical GNNs from the view of computational complexity. The computational complexity of a GNN layer is related to the complexity of its vertex and edge functions, i.e. $O(m * (O_\phi + O_\Sigma) + n * O_\gamma)$, where $O_\phi/O_\Sigma/O_\gamma$ are the computational complexity of the three functions.

The complexity can be decomposed into two parts: the edge computation complexity $O_\phi + O_\Sigma$ and the vertex computation complexity O_γ. In Table 1 and Table 2, we list the edge and vertex computation complexity, respectively. The edge/vertex complexity of a graph neuron are affected by the dimensions of the input/output hidden vectors d_{in} and d_{out} and the dimensions of the model parameters (like the number of heads K in GAT and the dimensions of the view vectors d_a/d_v in GaAN).

We classify the typical GNNs into four quadrants based on their edge/vertex complexity as shown in Figure 3. We pick GCN, GGNN, GAT and GaAN as the representative GNNs of the four quadrants.

GCN [6] (low vertex & edge computational complexity): Graph convolution network (GCN) is the first-order approximation of the spectral-based graph convolutions. It has only one parameter to learn at each layer, i.e. the weight matrix

$\boldsymbol{W}^l$ in γ. A GCN graph neuron can be expressed as $\boldsymbol{h}_i^{l+1} = \boldsymbol{W}^l \sum_{v_j \in \mathcal{N}(v_i)} e_{j,i} \boldsymbol{h}_j^l$, where $e_{j,i}$ is the normalized weight of the edge (v_j, v_i). According to the associative law of the matrix multiplication, $\boldsymbol{h}_i^{l+1} = \sum_{v_j \in \mathcal{N}(v_i)} e_{j,i} \boldsymbol{W}^l \boldsymbol{h}_j^l$. Since the dimension of $\boldsymbol{h}_i^{l+1}$ is usually smaller than $\boldsymbol{h}_i^l$ in practical GCNs, the implementation of GCN [3] chooses to first conduct the vertex computation $\hat{\boldsymbol{h}}_j^l = \boldsymbol{W}^l \boldsymbol{h}_j^l$ for each vertex v_j and then conduct the edge computation $\boldsymbol{h}_i^{l+1} = \sum_{v_j \in \mathcal{N}(v_i)} \hat{\boldsymbol{h}}_j^l$. As $\hat{\boldsymbol{h}}_j^l$ has the same dimension as $\boldsymbol{h}_i^{l+1}$, the implementation significantly reduces the computation cost of the edge computation.

3. Experiment Design

4. Experiment Results and Analysis

5. Insights

6. Related Work

7. Conclusion and Future Work

Acknowledgment

References

[1] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun, Graph neural networks: A review of methods and applications (2018). `arXiv:1812.08434`.
URL https://arxiv.org/abs/1812.08434

[2] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, G. E. Dahl, Neural message passing for quantum chemistry, in: D. Precup, Y. W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, Vol. 70

[3] The implementation of GCN in PyTorch Geometric: https://github.com/rusty1s/pytorch_geometric/blob/1.5.0/torch_geometric/nn/conv/gcn_conv.py

of Proceedings of Machine Learning Research, PMLR, International Convention Centre, Sydney, Australia, 2017, pp. 1263–1272.
URL http://proceedings.mlr.press/v70/gilmer17a.html

[3] M. Fey, J. E. Lenssen, Fast graph representation learning with pytorch geometric (2019). arXiv:1903.02428.
URL https://pytorch-geometric.readthedocs.io/

[4] 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, Z. Zhang, Deep graph library: Towards efficient and scalable deep learning on graphs (2019). arXiv:1909.01315.
URL https://www.dgl.ai/

[5] M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional neural networks on graphs with fast localized spectral filtering, in: D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, R. Garnett (Eds.), Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain, 2016, pp. 3837–3845.
URL http://papers.nips.cc/paper/6081-convolutional-neural-networks-on-graphs-with-fast-l

[6] T. N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, OpenReview.net, 2017.
URL https://openreview.net/forum?id=SJU4ayYgl