L^2-GCN: Layer-Wise and Learned Efficient Training of Graph Convolutional Networks

Yuning You,* Tianlong Chen,* Zhangyang Wang, Yang Shen
Texas A&M University
{yuning.you,wiwjp619,atlaswang,yshen}@tamu.edu

Abstract

Graph convolution networks (GCN) are increasingly popular in many applications, yet remain notoriously hard to train over large graph datasets. They need to compute node representations recursively from their neighbors. Current GCN training algorithms suffer from either high computational costs that grow exponentially with the number of layers, or high memory usage for loading the entire graph and node embeddings. In this paper, we propose a novel efficient layer-wise training framework for GCN (L-GCN), that disentangles feature aggregation and feature transformation during training, hence greatly reducing time and memory complexities. We present theoretical analysis for L-GCN under the graph isomorphism framework, that L-GCN leads to as powerful GCNs as the more costly conventional training algorithm does, under mild conditions. We further propose L^2-GCN, which learns a controller for each layer that can automatically adjust the training epochs per layer in L-GCN. Experiments show that L-GCN is faster than state-of-the-arts by at least an order of magnitude, with a consistent of memory usage not dependent on dataset size, while maintaining comparable prediction performance. With the learned controller, L^2-GCN can further cut the training time in half. Our codes are available at https://github.com/Shen-Lab/L2-GCN.

1. Introduction

Graph convolution networks (GCN) [13] generalize convolutional neural networks (CNN) [14] to graph data. Given a node in a graph, a GCN first *aggregates* the node embedding with its neighbor node embeddings, and then *transforms* the embedding through (hierarchical) feed-forward propagation. The two core operations, *i.e.*, aggregating and transforming node embeddings, take advantage of the graph structure and outperform structure-unaware alternatives [17, 18, 9]. GCNs hence demonstrate prevailing success in many graph-based applications, including node classifica-
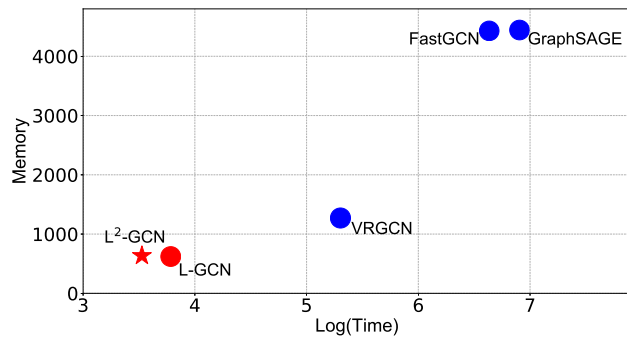
---
*Equal Contribution.

Figure 1: Summary of our achieved performance and efficiency on Reddit. **The lower left corner indicates the desired lowest complexity in time (training time) and memory consumption (GPU memory usage).** The size of markers represents F1 scores. Blue circles (●) are state-of-the-art mini-batch training algorithms, red circle (●) is L-GCN, and red star (★) is L^2-GCN. The corresponding F1 scores are: GraphSAGE (●, 93.4), FastGCN (●, 92.6), VRGCN (●, 96.0), L-GCN (●, 94.2) and L^2-GCN(★, 94.0).

tion [13], link prediction [26] and graph classification [24].

However, the training of GCNs has been a headache, and a hurdle to scale up GCNs further. How to train CNNs more efficiently has recently become a popular topic of explosive interest, by bypassing unnecessary data or reducing expensive operations [19, 25, 12]. For GCNs, as the graph dataset grows, the large number of nodes and the potentially dense adjacency matrix prohibit fitting them all into the memory, thus putting *full-batch* training algorithms (*i.e.*, those requiring the full data and holistic adjacency matrix to perform) in jeopardy. That motivates the development of *mini-batch* training algorithms, *i.e.*, treating each node as a data point and updating locally. In each mini-batch, the embedding of a node at the lth layer is computed from the neighborhood node embeddings at the $(l-1)$-th layer through the graph convolution operation. As the computation is performed recursively through all layers, the mini-batch complexity will increase exponentially with respect to the layer number. To mitigate the complexity explosion, several sampling-based strategies have been adopted, *e.g.* GraphSAGE [10] and Fast-

Table 1: Time and memory complexities: we consider time complexity for the feature propagation in the network, and memory complexity for storing node embeddings in each layer (since given the same network architecture, the weight storage costs the same memory for all compared algorithms c). L is the layer number, D the feature dimension, N the node number, S_{NEI} the neighborhood size, B the minibatch size, S the training sample size, S_{VR} the reduced sample size, and N_{BAT} the minibatch number.

	GCN [13]	Vanilla Mini-Batch	GraphSAGE [10]	FastGCN [4]	VRGCN [5]	**L-GCN**
Time	$O(L\|\|\hat{A}\|\|_0 D + LND^2)$	$O(S_{\text{NEI}}^L BD^2)$	$O(S^L BD^2)$	$> O(SLBD^2)$	$> O(S_{\text{VR}}^L BD^2)$	$O(L\frac{\|\|\hat{A}\|\|_0}{N_{\text{BAT}}}D + BD^2)$
Memory	$O(LND)$	$O(S_{\text{NEI}}^L BD)$	$O(S^L BD)$	$> O(SLBD)$	$O(LND)$	$O(BD)$

GCN [4], yet with few performance guarantees. VRGCN [5] reduces the sample size through variance reduction, and guarantees its performance convergence to the full-sample approach, but it requires to store the full-batch node embeddings of each layer in the memory, limiting its efficiency gain. Cluster-GCN [7] used graph clustering to partition the large graph into subgraphs, and performs subgraph-level mini-batch training, yet again being only empirical.

In this paper, we propose a novel layer-wise training algorithm for GCNs, called (**L-GCN**). The key idea is to decouple the two key operations in the per-layer feedforward graph convolution: feature aggregation (FA) and feature transformation (FT), whose concatenation and cascade result in the exponentially growing complexity. Surprisingly, the resulting greedy algorithm will not necessarily compromise the network representation capability, as shown by our theoretical analysis inspired by [23] using a graph isomorphism framework. To bypass extra hyper-parameter tuning, we then introduce layer-wise and learned GCN training (**L²-GCN**), which learns a controller for each layer that can automatically adjust the training epochs per layer in L-GCN. Table 1 compares the training complexity between L-GCN, L²-GCN and existing competitive algorithms, demonstrating our approaches' remarkable advantage in reducing both time and memory complexities. More experiments show that our proposed algorithms are significantly faster than state-of-the-arts, with a consistent usage of GPU memory not dependent on dataset size, while maintaining comparable prediction performance. Our contributions can be summarized below:

- A layer-wise training algorithm for GCNs with much lower time and memory complexities;
- Theoretical justification that under some sufficient conditions the greedy algorithm does not compromise in the graph-representative power;
- Learned controllers that automatically configure layer-wise training epoch numbers, in place of manual hyper-parameter tuning;
- State-of-the-art performance achieved in addition to the light weight, on extensive applications.

2. Related Work

We follow [7] to categorize existing GCN training algorithms into *full-batch* and *mini-batch* (stochastic) algorithms,

and compare their pros and cons.

2.1. Full-Batch GCN Training

The original GCN [13] adopted the full-batch gradient descent algorithm. Let's define an undirected graph as $G = (V, E)$, where $V = \{v_1, ..., v_N\}$ represents the vertex set with N nodes, and $E = \{e_1, ..., e_{N_E}\}$ represents the edge set with N_E edges: $e_n = (v_i, v_j)$ indicates an edge between vertices v_i and v_j. $F \in \mathbb{R}^{N \times D}$ is the feature matrix with the feature dimension D, and $A \in \mathbb{R}^{N \times N}$ is the adjacency matrix where $a_{ij} = a_{ji} = \left\{ \begin{smallmatrix} 1, \text{if } (v_i,v_j) \text{ or } (v_j,v_i) \in E \\ 0, \text{otherwise} \end{smallmatrix} \right.$. By constructing an L-layer GCN, we express the output $X^{(l)} \in \mathbb{R}^{N \times D^{(l)}}$ of the lth layer and the network loss as:

$$X^{(l)} = \sigma(\hat{A}X^{(l-1)}W^{(l)}), \text{Loss} = Loss(X^{(L)}, \Theta, Y), \quad (1)$$

where $\hat{A}$ is the regularized adjacency matrix, $X^{(0)} = F$, $W^{(l)} \in \mathbb{R}^{D^{(l-1)} \times D^{(l)}}$ is the weight matrix, $D^{(0)} = D$, $\sigma(\cdot)$ is a nonlinear function, $\Theta \in \mathbb{R}^{D^{(L)} \times D_{CLA}}$ is the linear classification matrix, Y the training labels, and $Loss(\cdot)$ the loss function. For simplicity and without affecting the analysis, we set $D^{(1)} = ... = D^{(L)} = D$.

For the time complexity of the network propagation in (1), $\hat{X}^{(l)} = \hat{A}X^{(l-1)}$ costs $O(\|\|\hat{A}\|\|_0 D)$ in time and $\hat{X}^{(l)}W^{(l)}$ costs $O(ND^2)$ in time, which in total leads to $O(L\|\|\hat{A}\|\|_0 D + LND^2)$ time consumed for the entire network. For the memory complexity, storing the L-layer embeddings $X^{(l)}, l = 1, ..., L$ requires $O(LND)$ in memory. Both time and memory complexities are proportional to N, which cannot scale up well for large graphs.

2.2. Mini-Batch SGD Algorithms

The vanilla mini-batch SGD algorithm propagates the vertex representations in a minibatch, rather than for all nodes. We rewrite the network propagation (1) for the ith node in the lth layer as:

$$x_i^{(l)} = \sigma((\hat{a}_{ii}x_i^{(l-1)} + \sum_{j=1,...,N, \text{ s.t. } \hat{a}_{ij} \neq 0} a_{ij}x_j^{(l-1)}), W^{(l)}),$$

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N} Loss(x_i^{(L)}, \Theta, Y),$$
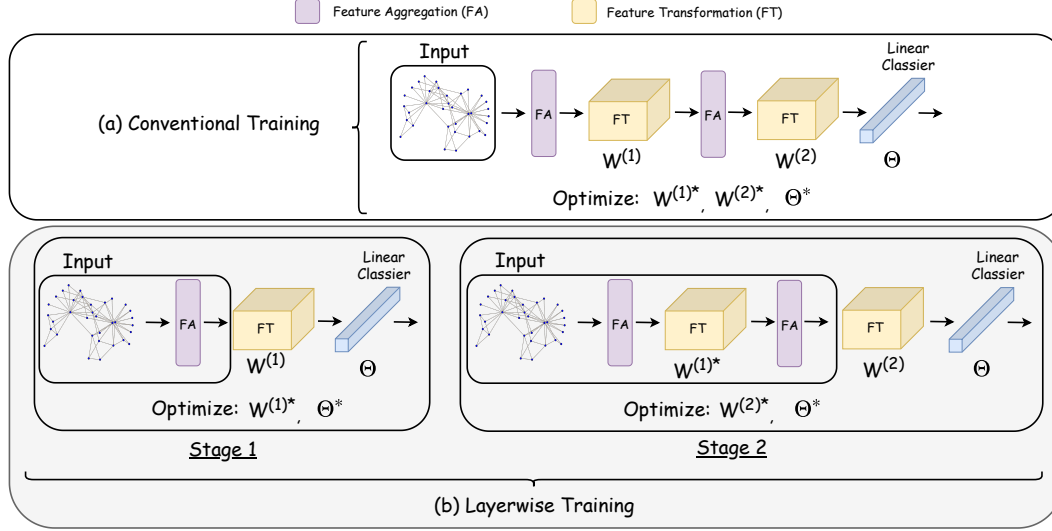
$$(2)$$

Figure 2: Conventional training vs layer-wise training for a two-layer GCN. (a) In conventional training, the optimizer jointly optimizes the weight matrix $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}$ as $\boldsymbol{W}^{(1)*}, \boldsymbol{W}^{(2)*}$ and the linear classifier $\boldsymbol{\Theta}$ as $\boldsymbol{\Theta}^*$, saving $\boldsymbol{W}^{(1)*}, \boldsymbol{W}^{(2)*}, \boldsymbol{\Theta}^*$ for the network. (b) In layer-wise training, two layers are trained in two sequential stages: in the first stage the optimizer optimizes the weight matrix $\boldsymbol{W}^{(1)}$ as $\boldsymbol{W}^{(1)*}$ and the linear classifier $\boldsymbol{\Theta}^{(1)}$ as $\boldsymbol{\Theta}^{(1)*}$, saving $\boldsymbol{W}^{(1)*}$ for the second stage and dropping $\boldsymbol{\Theta}^{(1)*}$; then in the second stage the optimizer optimizes the weight matrix $\boldsymbol{W}^{(2)}$ as $\boldsymbol{W}^{(2)*}$ and the linear classifier $\boldsymbol{\Theta}^{(2)}$ as $\boldsymbol{\Theta}^*$, saving $\boldsymbol{W}^{(2)*}, \boldsymbol{\Theta}^*$ for the network.

where $\boldsymbol{x}_i^{(0)} = F[i, :]$. With (2), we can feed the feature matrix $\boldsymbol{F}$ in a mini-batch dataloader and run the stocastic gradient descent (SGD) optimizer. Suppose B is the minibatch size and S_{NEI} the neighborhood size, the time complexity for the propagation per mini-batch is $O(S_{\mathrm{NEI}}^L BD^2)$ and the memory complexity is $O(S_{\mathrm{NEI}}^L BD)$. We next discuss a few variants on top of the vanilla minim-batch algorithm:

- **GraphSAGE & FastGCN.** [10, 4] Both adopted sampling scheme to reduce complexities. GraphSAGE proposes to use fixed-size sampling for the neighborhood in each layer. It yet suffers from the "neighborhood expansion" problem, making its time and memory complexities grow exponentially with the layer number. FastGCN proposes global importance sampling rather than local neighborhood sampling, alleviating the complexity growth issue. Suppose $S \leq S_{\mathrm{NEI}}$ is the sample size, the time and memory complexities are $O(S^L BD^2)$ and $O(S^L BD)$ for GraphSAGE, and $O(SLBD^2)$ and $O(SLBD)$ for FastGCN, respectively. Besides, for FastGCN, there is extra complexity requirement for importance weight computation.

- **VRGCN.** [5] proposes to use variance reduction to reduce the sample size in each layer, which managed to achieve good performance with smaller graphs. Unfortunately, it requires to store all the vertex intermediate embeddings during training, which leads to its memory complexity coming close to the full-batch training. Suppose $S_{\mathrm{VR}} \leq S$ is the reduced sample size, the time and memory complexities of VRGCN are $O(S_{\mathrm{VR}}^L BD^2)$

and $O(LND)$, respectively (plus some overhead for computing variance reduction).

- **Cluster-GCN.** [7] Instead of feeding nodes and their neighbors directly, [7] first uses a graph clustering algorithm to partition subgraphs, and then runs the SGD optimizer over each subgraph. The performance of this approach heavily hinges on the chosen graph clustering algorithm. It is further difficult to ensure training stability, *e.g.*, w.r.t different clustering settings.

3. Proposed Algorithm

To discuss the bottleneck of graph convolutional network (GCN) training algorithms, we first analyze the propagation of GCN following [22] and factorize the propagation (1) into *feature aggregation (FA)* and *feature transformation (FT)*.

Feature aggregation. To learn the node representation $\boldsymbol{X}^{(l)}$ of the lth layer, in the first step GCN follows the neighborhood aggregation strategy, where in the lth layer it updates the representation of each node by aggregating the representations of its neighbors, and at the same time the representation of itself is aggregated by the representations of its neighbors, which is written as:

$$\hat{\boldsymbol{X}}^{(l)} = \hat{\boldsymbol{A}} \boldsymbol{X}^{(l-1)}. \tag{3}$$

With (3), the time and memory complexity is highly dependent on the edge number, and in the mini-batch SGD algorithm it is highly dependent on the sample size. Since during mini-batch SGD training for an L-layer network, L

times of FA for each node requires its L-th order neighbor nodes' representations, which results in sampling a large number of neighbor nodes. FA is the main barrier for reducing the time and memory complexity of GCN in the mini-batch SGD algorithm.

Feature transformation. After FA, in the second step GCN conducts FT in the lth layer, which consists of linear and nonlinear transformations:

$$\boldsymbol{X}^{(l)} = \sigma(\hat{\boldsymbol{X}}^{(l)}\boldsymbol{W}^{(l)}). \tag{4}$$

With (4), the complexity is mainly relevant to the feature dimension. L times FT for a node only requires its own representation in each layer. Given the supervised node labels $\boldsymbol{Y}$, the conventional training process for a GCN is formulated as:

$$(\boldsymbol{W}^{(1)*}, ..., \boldsymbol{W}^{(L)*}, \boldsymbol{\Theta}^*) = \\ \min_{\boldsymbol{W}^{(1)},...,\boldsymbol{W}^{(L)},\boldsymbol{\Theta}} Loss(\boldsymbol{X}^{(L)}, \boldsymbol{\Theta}, \boldsymbol{Y}), \ s.t. \ (3), (4). \tag{5}$$

For the entire propagation of a mini-batch SGD over an L-layer GCN, there are L times of FA and FT in each batch as shown in Figure 2(a) . Without FT, L times of FA can aggregate the structure information, which lacks representation learning and is still time- and memory-consuming. Without FA, L times of FT is no more than a multi-layer perceptron (MLP), which efficiently learns the representation but lacks structure information.

3.1. L-GCN: Layer-wise GCN Training

As described earlier, the one-batch propagation of the conventional training for an L-layer GCN consists of L times of feature aggregation (FA) and feature transformation (FT). Both FA and FT are necessary for capturing graph structures and learning data representations but the coupling between the two leads to inefficient training. We therefore propose a *layer-wise training algorithm* (L-GCN) to properly separate the FA and FT processes while training GCN layer by layer.

We illustrate the L-GCN algorithm in Figure 2(b). For training the lth layer, we do FA once for all the $(l-1)$th vertex representations, aggregating its lth order structure information, and then feed the vertex embeddings into a single layer perceptron and run the mini-batch SGD optimizer for batches. The lth layer is trained by solving

$$(\boldsymbol{W}^{(l)*}, \boldsymbol{\Theta}^*) = \\ \min_{\boldsymbol{W}^{(l)},\boldsymbol{\Theta}} Loss\left(\sigma(\hat{\boldsymbol{A}}\boldsymbol{X}^{(l-1)}\boldsymbol{W}^{(l)}), \boldsymbol{\Theta}, \boldsymbol{Y}\right). \tag{6}$$

Note that $\boldsymbol{X}^{(l-1)}$ depends on $(\boldsymbol{W}^{(1)*}, \ldots, \boldsymbol{W}^{(l-1)*})$. After finishing the lth layer training, we save the weight matrix $\boldsymbol{W}^{(l)*}$ between the current input layer and hidden layer as the weight matrix of the lth layer, drop the weight

matrices between hidden layer and output layer unless $l = L$, and calculate the lth-layer representations. This process is repeated until all layers are trained.

The time and memory complexities are significantly lower compared to the conventional training and the state of the arts, as shown in Table 1. For the time complexity, L-GCN only conducts FA L times in the entire training process and FT does once per batch, whereas the conventional mini-batch training conducts FA L times and FT L times in each batch. Suppose that the total training batch number is N_{BAT}, the time complexity of L-GCN is $O(L\frac{||\hat{\boldsymbol{A}}||_0}{N_{BAT}}D + BD^2)$. The memory complexity is $O(BD)$ since L-GCN only trains a single layer perceptron in each batch.

3.2. Theoretical Justification of L-GCN

We set out to answer the following question theoretically for L-GCN: *How close could the performance of layer-wise trained GCN be compared with conventionally trained GCN?* To establish the theoretical background of our layer-wise training algorithm, we follow Xu and coworker's work [23] and show that a layer-wise trained GCN can be as powerful as a conventionally trained GCN under certain conditions.

In [23], the representation power of an aggregation-based graph neural network (GNN) is evaluated, when input feature space is countable, as the ability to map any two different nodes into different embeddings. The evaluation of the representation power is extended to the ability to map any two non-isomorphic graphs into non-isomophic embeddings, where the graphs are generated as the rooted subtrees of the corresponding nodes. An L-layer GNN $\mathcal{A} : \mathbb{G} \to \mathbb{R}^D$ (excluding the linear classifier described earlier) can be represented [23] as:

$$\mathcal{A} = \mathcal{R} \circ \mathcal{L}^{(L)} \circ ... \circ \mathcal{L}^{(1)}, \tag{7}$$

where $\mathcal{L}^{(l)} : \mathbb{R}^D \times \mathbb{M}^D \to \mathbb{R}^D, (l = 1, \ldots, L)$ is the vertex-wise aggregating mapping , $\mathbb{M}^D$ is the multiset of dimension D, and $\mathcal{R} : \mathbb{M}^D \to \mathbb{R}^D$ is the readout mapping as:

$$\boldsymbol{x}_i^{(l)} = \mathcal{L}^{(l)}(\boldsymbol{x}_i^{(l-1)}, \{\boldsymbol{x}_j^{(l-1)} : j \in \mathcal{N}(i)\}), l \in L, \\ \text{Output} = \mathcal{R}(\{\boldsymbol{x}_i^{(L)} : i \in N\}), \tag{8}$$

where $\mathcal{N}(i)$ is the set of node neighbors for the ith node.

Since GCN belongs to aggregation-based GNN, we use the same graph isomorphism framework for our analysis. Xu et al. [23] provided the upper-bound power of GNN as Weisfeiler-Lehman graph isomorphism test (WL test) [20], and proved sufficient conditions for GNN to be as powerful as the WL test, which is described in the following lemma and theorem.

Lemma 1. [23] Let G_1 and G_2 be any two non-isomorphic graphs, i.e. $G_1 \ncong G_2$. If a GNN $\mathcal{A}$ maps G_1 and G_2 into different embeddings, the WL test also decide G_1 and G_2 are not isomorphic.

Theorem 2. [23] Let $\mathcal{A} = \mathcal{R} \circ \mathcal{L}^{(L)} \circ ... \circ \mathcal{L}^{(1)}$ be a GNN with sufficient number of GNN layers, $\mathcal{A}$ maps any graphs G_1 and G_2 that the WL test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold: a) The mappings $\mathcal{L}^{(l)}, l \in L$ are injective. b) The readout mapping $\mathcal{R}$ is injective.

We further propose to use the graph isomorphism framework to characterize the "power" of a GNN. In this framework, we observe the fact that for an aggregation-based GNN $\mathcal{A}$ (such as GCN), with any pair of isomorphic graphs G_1 and G_2, we always have $\mathcal{A}(G_1) = \mathcal{A}(G_2)$ due to the identical input and aggregation-based mapping. In contrast, for any pair of non-isomorphic graphs G_1 and G_2, there exists certain probability ϵ that $\mathcal{A}$ wrongly maps them into identical embeddings, i.e. $\mathcal{A}(G_1) = \mathcal{A}(G_2)$, as shown in Table 2. Therefore, to further analyze our algorithm, we first define a specific metric to evaluate the capacity of a GNN, as the probability of mapping any non-isomorphic graphs into different embeddings.

Table 2: Conditional probabilities of GNN $\mathcal{A}$ identifying two graphs G_1 and G_2 given their (non)isomorphism.

	$G_1 \cong G_2$	$G_1 \ncong G_2$
$\mathcal{A}(G_1) = \mathcal{A}(G_2)$	1	ϵ
$\mathcal{A}(G_1) \neq \mathcal{A}(G_2)$	0	$1 - \epsilon$

Definition 3. Let $\mathcal{A}$ be a GNN; G_1 and G_2 are i.i.d. The capacity of $\mathcal{A}$, $C_{\mathcal{A}}$, is defined as the probability to map G_1 and G_2 into different embeddings if they are non-isomorphic:

$$C_{\mathcal{A}} \equiv \text{Prob}\{\mathcal{A}(G_1) \neq \mathcal{A}(G_2) | G_1 \ncong G_2\}. \quad (9)$$

Higher capacity of a GNN indicates its stronger distinguishing capability between non-isomorphic graphs, which corresponds to more power in graph isomorphism framework. In other words, not so powerful network will have a higher probability to map non-isomorphic graphs into the same embeddings and fail to distinguish them. With Theorem 2 and Definition 3, we have $C_{\mathcal{A}} \leq C_{WL}$, i.e. the capacity of WL test is the upper bound of the capacity of any aggregation-based GNN. Intuitively, with the metric to evaluate the network power, we further define the training process as the problem of optimizing the network capacity.

Definition 4. Let a GNN $\mathcal{A} = \mathcal{R} \circ \mathcal{L}^{(L)} \circ ... \circ \mathcal{L}^{(1)}$ with a fixed injective readout function $\mathcal{R}$, G_1 and G_2 are i.i.d. The training process for $\mathcal{A}$ is formulated as:

$$\mathcal{L}^{(L)*}, ..., \mathcal{L}^{(1)*} =$$
$$\max_{\mathcal{L}^{(L)}, ..., \mathcal{L}^{(1)}} \text{Prob}\{\mathcal{R} \circ \mathcal{L}^{(L)} \circ ... \circ \mathcal{L}^{(1)}(G_1) \quad (10)$$
$$\neq \mathcal{R} \circ \mathcal{L}^{(L)} \circ ... \circ \mathcal{L}^{(1)}(G_2) | G_1 \ncong G_2\}.$$

Therefore, when training the network, the optimizer tries to find the best layer mapping for GNN to map non-isomorphic graphs into different embeddings as much as possible. With training process in Definition 4, we formulate the greedy layer-wise training for $\mathcal{A}$ as:

$$\mathcal{L}^{(1)*} = \max_{\mathcal{L}^{(1)}} \text{Prob}\{\mathcal{R} \circ \mathcal{L}^{(1)}(G_1) \neq \mathcal{R} \circ \mathcal{L}^{(1)}(G_2) | G_1 \ncong G_2\},$$

$$\mathcal{L}^{(2)*} = \max_{\mathcal{L}^{(2)}} \text{Prob}\{\mathcal{R} \circ \mathcal{L}^{(2)} \circ \mathcal{L}^{(1)*}(G_1)$$
$$\neq \mathcal{R} \circ \mathcal{L}^{(2)} \circ \mathcal{L}^{(1)*}(G_2) | G_1 \ncong G_2\},$$

...

$$\mathcal{L}^{(L)*} = \max_{\mathcal{L}^{(L)}} \text{Prob}\{\mathcal{R} \circ \mathcal{L}^{(L)} \circ \mathcal{L}^{(L-1)*} \circ ... \circ \mathcal{L}^{(1)*}(G_1)$$
$$\neq \mathcal{R} \circ \mathcal{L}^{(L)} \circ \mathcal{L}^{(L-1)*} \circ ... \circ \mathcal{L}^{(1)*}(G_2) | G_1 \ncong G_2\},$$

$$(11)$$

In the following theorem, we provide a sufficient condition for a network trained layer-wise (11) to achieve the same capacity, as one trained from end to end (10).

Theorem 5. Let $\mathcal{A} = \mathcal{R} \circ \mathcal{L}^{(L)} \circ ... \circ \mathcal{L}^{(1)}$ be a GNN with a fixed injective readout function $\mathcal{R}$. If $\mathcal{A}$ can be conventionally trained by solving the optimization problem (10) and the resulting $\mathcal{A}_{Con} = \mathcal{R} \circ \mathcal{L}_{Con}^{(L)} \circ ... \circ \mathcal{L}_{Con}^{(1)}$ is as powerful as the WL test given the conditions in Theorem 2, then $\mathcal{A}$ can also be layer-wise trained by solving the optimization problem (11) with the resulting $\mathcal{A}_{Lay} = \mathcal{R} \circ \mathcal{L}_{Lay}^{(L)} \circ ... \circ \mathcal{L}_{Lay}^{(1)}$ achieving the same capacity.

We provide the proof in the appendix. For the network architecture which is originally powerful enough through conventional training, we can train it to achieve the same capacity through layer-wise training. The idea of the proof is that: if there exists the injective mapping for each layer as the conditions in Theorem 2 satisfied, we can prove to find the injective mapping with layer-wise optimization problem as (11). Otherwise, when the network architecture can not be powerful enough through conventional training, the following theorem establishes that the layer-wise trained network has non-decreasing capacity as the layer number increases.

Theorem 6. Let GNN $\mathcal{A} = \mathcal{R} \circ \mathcal{L}^{(L)} \circ ... \circ \mathcal{L}^{(1)}$ with a fixed injective readout function $\mathcal{R}$, G_1 and G_2 are i.i.d., and $\boldsymbol{x}_i^{(l)} = \mathcal{L}^{(l)}(\boldsymbol{x}_i^{(l-1)}, \{\boldsymbol{x}_j^{(l-1)} : j \in \mathcal{N}_i\}) : \mathbb{R}^D \times \mathbb{M}^D \to \mathbb{R}^D$. With layer-wise training, if $\mathcal{L}_{Lay}^{(l)}$ is not guaranteed to be injective for $(\boldsymbol{x}_i^{(l-1)}, \{\boldsymbol{x}_j^{(l-1)} : j \in \mathcal{N}_i\})$, but it still can distinguish different $\boldsymbol{x}_i^{(l-1)}$, i.e. if $\boldsymbol{x}_i^{(l-1)} \neq \boldsymbol{x}_k^{(l-1)}$, then $\mathcal{L}_{Lay}^{(l)}(\boldsymbol{x}_i^{(l-1)}, \{\boldsymbol{x}_j^{(l-1)} : j \in \mathcal{N}_i\}) \neq \mathcal{L}_{Lay}^{(l)}(\boldsymbol{x}_k^{(l-1)}, \{\boldsymbol{x}_j^{(l-1)} : j \in \mathcal{N}_k\})$, then we have that the capacity of the network is monotonically non-decreasing
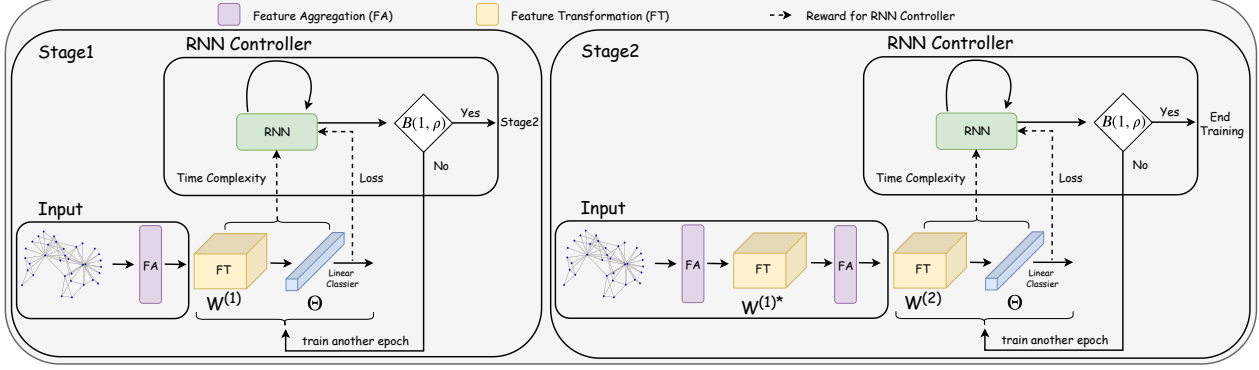
Figure 3: Learning to optimize in layerwise training algorithm for a two-layer GCN. Two layers are trained in two stage: in each stage the layer-wise trained network generates the training loss, the later index as the input for the RNN controller, and the RNN controller outputs the hidden stage for the next epoch, and the stopping probability ρ for the current epoch, and sees whether the stopping probability ρ is greater than the threshold probability ρ_{Thres}. If $\rho > \rho_{\mathrm{Thres}}$ then the layer-wise training finishes, entering the next stage or ending the whole train (if it is the last layer), otherwise $\rho \leq \rho_{\mathrm{Thres}}$ then training for another epoch. When one iteration of training process finishes, the reward of performance and efficiency will be fed back to the RNN controller, updating the weight and starting a new round of training.

with deeper layers:

$$
\begin{aligned}
&\mathrm{Prob}\{\mathcal{R} \circ \mathcal{L}_{Lay}^{(l-1)} \circ ... \circ \mathcal{L}_{Lay}^{(1)}(G_1) \\
&\neq \mathcal{R} \circ \mathcal{L}_{Lay}^{(l-1)} \circ ... \circ \mathcal{L}_{Lay}^{(1)}(G_2)|G_1 \ncong G_2\} \\
&\leq \mathrm{Prob}\{\mathcal{R} \circ \mathcal{L}_{Lay}^{(l)} \circ \mathcal{L}_{Lay}^{(l-1)} \circ ... \circ \mathcal{L}_{Lay}^{(1)}(G_1) \\
&\neq \mathcal{R} \circ \mathcal{L}_{Lay}^{(l)} \circ \mathcal{L}_{Lay}^{(l-1)} \circ ... \circ \mathcal{L}_{Lay}^{(1)}(G_2)|G_1 \ncong G_2\}.
\end{aligned}
\tag{12}
$$

We again direct readers to the appendix for the proof. The theorem indicates that, if the network architecture is not powerful enough through conventional training , we can try to increase its capacity through training a deeper network. Layer-wise training can also train deeper GCNs more efficiently compared to state-of-the-arts.

What remains challenging is that the network capacity $C_{\mathcal{A}}$ is not available in an analytical form with regards to network parameters. In this study, we use the cross entropy as the loss function in classification tasks. More development in loss functions would be needed in future.

3.3. L²-GCN: Training with Learn Controllers

One challenge to apply the layerwise training algorithm to graph convolutional networks (L-GCN) is that one may need to manually adjust the training epochs for each layer. A possible solution is early stopping, nevertheless it does not intuitively work well in L-GCN since the training loss in each layer is not comparable with the final validation loss. Motivated by learning to optimize [2, 6, 15, 3], we propose L²-GCN, training a learned RNN controller to decide when to stop in each layer's training via policy-based REINFORCE [21]. The algorithm is illustrated in Figure 3.

Specifically, we model the training process for L-GCN as a Markov Decision Process (MDP) defined as follows: i) **Action.** The action a_t at time t for the RNN controller is

making the decision on whether to stop at the current-layer training or not. ii) **State.** The state s_t at time t is the loss in the current epoch, the layer index, and the hidden state of the RNN controller at time $t - 1$. iii) **Reward.** The purpose of the RNN controller is to train the network efficiently with competitive performance, and therefore the non-zero reward is only received at the end of the MDP as the weighted sum of final loss and total training epochs (Time Complexity). iv) **Terminal.** Once the L-GCN finishes the L-layer training, the process terminates.

Given the above settings, a sample trajectory from MDP will be: $(s_1, a_1, r_1, ..., s_t, a_t, r_t)$. The detailed architechture of RNN controller is shown in Figure 4. For each time step, the RNN will output a hidden vector, which will be decoded and classied by its corresponding softmax classier. The RNN controller works in an autogressive way, where the output of the last step will be fed into the next step. L-GCN will be sampled for each time step's output to decide whether to stop or not. When terminated, a final reward will be fed to the controller to update the weight.

4. Experiments

In this section, we evaluate predictive performance, training time, and GPU memory usages of our proposed L-GCN and L²-GCN on single- and multi-class classification tasks for six increasingly larger datasets: Cora & PubMed [13], PPI & Reddit [10], and Amazon-670K & Amazon-3M [7], as summarized in appendix. For Amazon-670K & Amazon-3M, we use principal component analysis [11] to reduce the feature dimension down to 100, and use the top-level categories as the class labels. The train/validate/test split is following the conventional setting for the inductive supervised learning scenario. We implemented our proposed
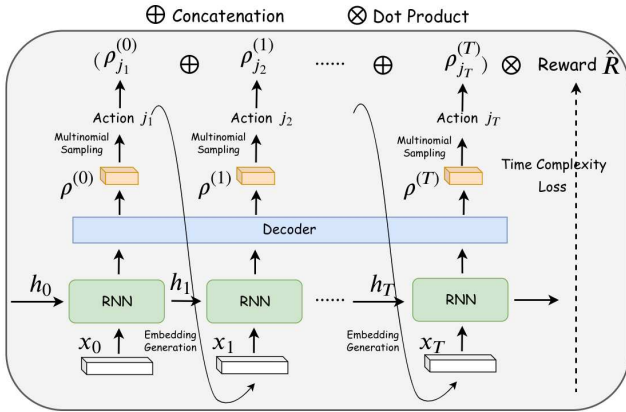
Figure 4: For each time step t, the RNN controller will first take the previous hidden vector h_t and the generated embedding x_t as input . Embedding x_t is a concatenation of action embeddings, loss in the current epoch and the layer index. (Following [8], we generate embeddings randomly from a multinomial distribution for each action.) Then the RNN will output a hidden vector h_{t+1}, which will be decoded and classified by its corresponding softmax classifier. After obtaining probability vector $\rho^{(t)}$, it performs multinomial sampling to pick the action j_{t+1} according to $\rho^{(t)}$. The relative probability $\rho^{(t)}_{t+1}$ will be further concatenated and recorded for reward calculation. When terminal, a final reward which is the dot product between picked actions probability vector and $\hat{R}$ ($\hat{R}$ is the weighted sum of the final loss and total training epochs), will be generated for the controller to update its weights.

algorithm in PyTorch [16]: for layer-wise training, we use the Adam optimizer with learning rate of 0.001 for Cora & PubMed, and 0.0001 for PPI, Reddit, Amazon-670K and Amazon-3M; for RNN controller, we set the controller to make a stopping-or-not decision each 10 epochs (5 for Cora and 50 for PPI), use the controller architecture as in [8] and the Adam optimizer with the learning rate of 0.05. All the experiments are conducted on a machine with GeForce GTX 1080 Ti GPU (11 GB memory), 8-core Intel i7-9800X CPU (3.80 GHz) and 16 GB of RAM.

4.1. Comparison with State of the Arts

To demonstrate the efficiency and performance of our proposed algorithms, we compare them with state-of-the-arts in Table 3. We compare L-GCN and L^2-GCN to the state-of-the-art GCN mini-batch training algorithms as GraphSAGE [10], FastGCN [4] and VRGCN [5], using their originally released codes and published settings, except that the batchsize and the embedding dimension of hidden layers are kept the same in all methods to ensure fair comparisons. Specifically, we set the batchsize at 256 for Cora and 1024 for others; and we did the embedding dimension of hidden layers at 16 for Cora & PubMed, 512 for PPI and 128 for others. We do not compare the controller with other hyper-parameter tuning methods since the controller is widely used in many fields such as neural architecture search [8].

On four common datasets Cora, PubMed, PPI and Reddit, we demonstrate that our proposed algorithm L-GCN is significantly faster than state-of-the-arts, with a consistent usage of GPU memory not dependent on dataset size, while maintaining comparable prediction performance. With a learned controller to make the stopping decision, L^2-GCN can further reduce the training time (here we do not include search time) by half with tiny performance loss compared to L-GCN. For super large datasets, GraphSAGE and FastGCN fail to converge on Amazon-670K, and exceed the time limit on Amazon-3M in our experiment, whereas VRGCN achieves good performances after long training. Our proposed algorithms still stably achieve comparable performances efficiently on both Amazon-670K and Amazon-3M.

We did not include in Table 1 the time spent on hyper-parameter tuning (search) for any algorithm. Such a comparison was impossible as search time was not accessible for pre-trained state-of-the-arts. Although a typical controller learning can be expensive (as reported in Table 4), RNN controllers in L^2-GCN learned over (especially large) datasets can be transferrable (shown next); and L^2-GCN without controller retraining actually saves time compared to dataset-specific manual tuning. As to the memory usage, the trends in practical GPU memory usages during training did not entirely agree with those in the theoretical analyses (Table 1). We contemplate that it is more likely in implementation: other models were implemented on TensorFlow and ours on PyTorch; and possible CPU memory usage of some models was unclear.

4.2. Ablation Study

Transferability. We explore the transferability of the learned controller. Results in Table 4 show that the controller learned from larger datasets could be reused for smaller ones (with similar loss functions) and thus save search time.

Epoch configuration. We consider the influence of different epoch configurations in layer-wise training on performance on six datasets. Table 5 shows that training under different epoch numbers in different layers will affect the final performance. For layer-wise training (L-GCN), we configure different numbers of epochs for the two layers of our GCN as reported in Table 5. For layer-wise training with learning to optimize (L^2-GCN), we let the RNN controller to learn the epoch configuration from randomly sampled subgraphs as training data and report the automatically learned epoch numbers. Experimental results show that, trained with more epochs for each layer, L-GCN improves perfoemance except for Cora. Moreover, with learning to optimize, the RNN controller in L^2-GCN automatically learns epoch configurations with tiny performance loss but much less epochs. Figure 5 compares the loss curves of layer-wise training under various configurations and over various datasets.

Deeper networks. We evaluate the necessity of train-

Table 3: Comparison with state-of-the-art on performance, training time and GPU memory usage (GPU memory usage during training). The best results for each row / dataset are highlighted in red.

	GraphSAGE [10]			FastGCN [4]			VRGCN [5]			L-GCN			L²-GCN		
	F1 (%)	Time	Memory	F1 (%)	Time	Memory	F1 (%)	Time	Memory	F1 (%)	Time	Memory	F1 (%)	Time	Memory
Cora	85.0	18s	655M	85.5	6.02s	659M	85.4	5.47s	253M	84.7	0.45s	619M	84.1	0.38s	619M
PubMed	86.5	483s	675M	87.4	32s	851M	86.4	118s	375M	86.8	2.93s	619M	85.8	1.50s	631M
PPI	68.8	402s	849M	-	-	-	98.6	63s	759M	97.2	49s	629M	96.8	26s	631M
Reddit	93.4	998s	4343M	92.6	761s	4429M	96.0	201s	1271M	94.2	44s	621M	94.0	34s	635M
Amazon-670K	83.1	2153s	849M	76.1	548s	1621M	92.7	534s	625M	91.6	54s	601M	91.2	30s	613M
Amazon-3M	-	-	-	-	-	-	88.3	2165s	625M	88.4	203s	601M	88.4	125s	613M

Table 4: The transferability of the learned contorllers.

	Cora			PubMed		
	F1 (%)	Train	Search	F1 (%)	Train	Search
Controller-Cora	84.1	0.38s	16s	-	-	-
Controller-PubMed	84.3	0.36s	0s	85.8	1.50s	125s
Controller-Amazon-3M	84.8	0.43s	0s	86.3	2.43s	0s

Table 5: The influence of epoch configuration.

	Cora		PubMed		PPI	
	F1 (%)	Epoch	F1 (%)	Epoch	F1 (%)	Epoch
L-GCN-Config1	83.2	60+60	86.8	100+100	93.7	400+400
L-GCN-Config2	84.7	80+80	86.3	120+120	94.1	500+500
L-GCN-Config3	83.0	100+100	86.4	140+140	94.9	600+600
L²-GCN	84.1	75+75	85.8	30+60	94.1	300+350

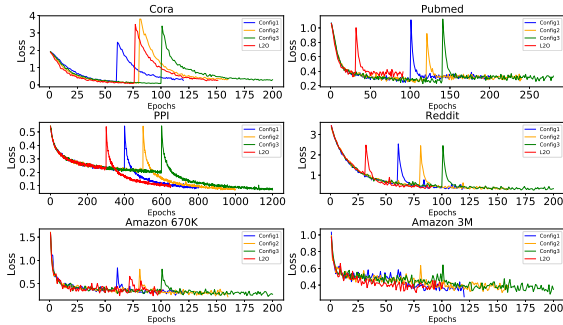|	Reddit		Amazon-670K		Amazon-3M	
	F1 (%)	Epoch	F1 (%)	Epoch	F1 (%)	Epoch
L-GCN-Config1	93.0	60+60	91.4	60+60	88.2	60+60
L-GCN-Config2	93.5	80+80	91.6	80+80	88.4	80+80
L-GCN-Config3	93.8	100+100	91.7	100+100	88.3	100+100
L²-GCN	92.2	30+60	91.2	70+30	88.0	20+80

Figure 5: The loss curves for manual epoch configurations and the learning-to-optimize configuration.

ing a deeper network using layer-wise training. Previous attempts seem to suggest the usefulness of training deeper GCN [13]. However, the datasets used in the experiments there are not large enough to draw a definite conclusion. Here we conduct experiments on 3 large datasets PPI, Reddit and Amazon-3M, with monotonically increasing layer number and total training epochs (each layer is trained for the same number of epochs) as shown in Table 6. Experimental results show that, with more network layers, the prediction performance of layer-wise training gets better. Compared with 2-layer network, 4-layer L-GCN gains performance increase of 4.0, 0.7, and 0.8 (%) on PPI, Reddit and Amazon-3M, respectively. When it comes to learn to optimize, the RNN controller learns a more efficient epoch configuration, while still achieving comparable performances as manually set epoch configurations.

Therefore, in layer-wise training, we have shown that

Table 6: The influence of layer number on final performance.

		PPI		Reddit		Amazon-3M	
		F1 (%)	Epoch	F1 (%)	Epoch	F1 (%)	Epoch
2-layer	L-GCN	93.7	800	93.8	200	88.4	160
	L²-GCN	94.1	750	92.2	90	88.0	100
3-layer	L-GCN	97.2	1200	94.2	300	89.0	240
	L²-GCN	96.8	650	94.0	210	88.7	120
4-layer	L-GCN	97.7	1600	94.5	400	89.2	320
	L²-GCN	97.3	1100	94.3	250	89.0	170

deeper layer networks can have better empirical performances, consistent with the theoretical, non-decreasing network capacity of deeper networks shown in Theorem 6.

Applying layer-wise training to N-GCN. We also apply layer-wise training to N-GCN [1], a recent GCN extension. It consists of several GCNs over multiple scales so layer-wise training is applied to each GCN individually. Results in Table 7 show that with layer-wise training, N-GCN is significantly faster with comparable performance.

Table 7: N-GCN with layer-wise training on Cora.

	Conventional Training		Layer-Wise Training	
	F1 (%)	Time	F1 (%)	Time
N-GCN	83.6	62s	83.1	4s

5. Conclusions

In this paper, we propose novel and efficient layerwise training algorithms for GCN (L-GCN) which separate feature aggregation and feature transformation during training and greatly reduce the complexity. Besides, we analyze theoretical grounds to rationalize the power of L-GCN in the graph isomorphism framework, provide a sufficient condition that L-GCN can be as powerful as conventional training, and prove that L-GCN is increasingly powerful as networks get deeper with more layers. Numerical results further support our theoretical analysis: our proposed algorithm L-GCN is significantly faster than state-of-the-arts, with a consistent usage of GPU memory not dependent on dataset size, while maintaining comparable prediction performance. Finally, motivated by learning to optimize, we propose L²-GCN, designing an RNN controller to make the stopping decision for each-layer training and training it to learn to make the decision rather than manually configure the training epochs. With the learned controller to make the stopping decision, L²-GCN on average further reduces the training time by half with tiny performance loss, compared to L-GCN.

References

[1] Sami Abu-El-Haija, Amol Kapoor, Bryan Perozzi, and Joonseok Lee. N-GCN: Multi-scale graph convolution for semi-supervised node classification. *arXiv preprint arXiv:1802.08888*, 2018. 8

[2] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989, 2016. 6

[3] Yue Cao, Tianlong Chen, Zhangyang Wang, and Yang Shen. Learning to optimize in swarms. In *Advances in Neural Information Processing Systems*, pages 15018–15028, 2019. 6

[4] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018. 2, 3, 7, 8

[5] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017. 2, 3, 7, 8

[6] Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando de Freitas. Learning to learn without gradient descent by gradient descent. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 748–756. JMLR. org, 2017. 6

[7] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. *arXiv preprint arXiv:1905.07953*, 2019. 2, 3, 6

[8] Xinyu Gong, Shiyu Chang, Yifan Jiang, and Zhangyang Wang. AutoGAN: Neural architecture search for generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3224–3234, 2019. 7

[9] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016. 1

[10] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017. 1, 2, 3, 6, 7, 8

[11] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933. 6

[12] Angela H Jiang, Daniel L-K Wong, Giulio Zhou, David G Andersen, Jeffrey Dean, Gregory R Ganger, Gauri Joshi, Michael Kaminksy, Michael Kozuch, Zachary C Lipton, et al. Accelerating deep learning by focusing on the biggest losers. *arXiv preprint arXiv:1910.00762*, 2019. 1

[13] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 1, 2, 6, 8

[14] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995. 1

[15] Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016. 6

[16] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 7

[17] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014. 1

[18] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015. 1

[19] Yue Wang, Ziyu Jiang, Xiaohan Chen, Pengfei Xu, Yang Zhao, Yingyan Lin, and Zhangyang Wang. E2-train: Training state-of-the-art cnns with over 80% energy savings. In *Advances in Neural Information Processing Systems*, pages 5139–5151, 2019. 1

[20] Boris Weisfeiler and Andrei A Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968. 4

[21] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. 6

[22] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr, Christopher Fifty, Tao Yu, and Kilian Q Weinberger. Simplifying graph convolutional networks. *arXiv preprint arXiv:1902.07153*, 2019. 3

[23] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018. 2, 4, 5

[24] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983. ACM, 2018. 1

[25] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Yingyan Lin, Zhangyang Wang, and Richard G Baraniuk. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*, 2019. 1

[26] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, pages 5165–5175, 2018. 1