A Framework for Accelerating Graph Convolutional Networks on Massive Datasets

Xiang Li[1](✉), Ruoming Jin[2](✉), Rajiv Ramnath[1](✉), and Gagan Agrawal[3](✉)

[1] Ohio State University, Columbus, OH 43210, USA
{li.3880,ramnath.6}@osu.edu

[2] Kent State University, Kent, OH, USA
jin@cs.kent.edu

[3] Augusta University, Augusta, GA, USA
gagrawal@augusta.edu

Abstract. In recent years, there has been much interest in Graph Convolutional Networks (GCNs). There are several challenges associated with training GCNs. Particularly among them, because of massive scale of graphs, there is not only a large computation time, but also the need for partitioning and loading data multiple times. This paper presents a different framework in which existing GCN methods can be accelerated for execution on large graphs. Building on top of ideas from meta-learning we present an optimization strategy. This strategy is applied to three existing frameworks, resulting in new methods that we refer to as GraphSage++, ClusterGCN++, and GraphSaint++. Using graphs with order of 100 million edges, we demonstrate that we reduce the overall training time by up to 30%, while not having a noticeable reduction in F1 scores in most cases.

1 Introduction

In recent years, there has been much interest in Graph Convolutional Networks (GCNs) [2,15]. There are several challenges associated with training GCNs. One of them is the *neighborhood explosion* when training a k-deep GCN, where the value at each node needs to be computed as an aggregation from its k-hop neighborhood. For graphs with large degrees, this computation is often intractable. To address this challenge, a number of sampling methods have been developed [4,5,9,10,16,18].

Another problem in GCN when applied to very large graphs is the need for partitioning the data and loading them multiple times because all of the data may not fit in the memory of the GPU. To explain the issue, consider the following summary of a typical training process [18]. *"1. Construct a complete GCN on the full training graph. 2. Sample nodes or edges of each layer to form mini-batches. 3. Perform forward and backward propagation among the sampled GCN. Steps (2) and (3) proceed iteratively"*. Now, if the training graph in the

D. Mohaisen and R. Jin (Eds.): CSoNet 2021, LNCS 13116, pp. 79–92, 2021.
https://doi.org/10.1007/978-3-030-91434-9_8

step 1 can fit into the memory of a single GPU, steps (2) and (3) can be applied without the need for loading or unloading the data. However, GPUs often do not have sufficient memory to allow a full training graph to be loaded. Only a limited amount of work to date has considered this problem [6,17]. These works build minibatches from subgraphs, and thus do not require the entire training graph to fit into the GPU memory. However, the problem with this approach is the high cost of frequently loading subgraphs into the GPU during each iteration.

This paper presents a different framework in which different GCN methods can be accelerated for execution on large graphs. Our work draws its inspiration from the idea of *meta-learning* [14], In meta-learning, we assume there is a large number of tasks over the same dataset, and our goal is to optimize these tasks all together. The correspondence we can draw is that training of a GCN using a single large graph can be viewed as a collection of training tasks over subgraphs or partitions, each of which fits into GPU memory.

Based on this idea, we develop an overall framework for accelerating GCN training over large graphs. The main idea is that by focusing on training of GCN over each subgraph that has already been loaded into memory, we can reduce the data loading times as compared to a normal implementation. We apply this idea to three recent algorithms for GCN training, GraphSaint [18], GraphSage [9], and ClusterGCN [6], resulting in new algorithms GraphSaint++, GraphSage++, and ClusterGCN++, respectively. We show mathematical analysis denoting why these methods are still able to converge, while reducing data loading costs.

We have carried out a detailed experimental evaluation of our three new algorithms using four graph datasets. We demonstrate how we are able to obtain comparable convergence and final F1 scores while reducing the data loading time by up to 90% and total training time up to 30%.

2 Technical Details

This section provides important backgrounds on GCNs, followed by discussion of existing methods, with an emphasis on the memory requirements and associated data loading costs.

2.1 Background

Consider a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges $E \subseteq V \times V$ represented by an adjacency matrix M, where an entry $M(i, j)$ denotes an edge between nodes i and j. Associated with every node in the graph are F features. Thus $X \in R^{|V| \times F}$ captures the F features for all nodes in the graph.

A GCN framework is composed of a number of layers (say, L). At each layer, the GCN computes a latent representation, using representation from the previous layer. For simplicity of presentation, we assume that the latent representation has the same dimension F as that of node feature. Thus, we denote the representation computed at the layer l by X^l, and $X^0 = X$. Now, the computation at each layer can be denoted as $X^{l+1} = A \times X^l \times W^l$. Here, W^l is a *feature transformation matrix*, $W^l \in R^{F \times F}$. The goal of the training process is to learn these matrices. In an inductive supervised learning based on GCNs, the goal is to learn the L weight transformation matrices while minimizing a *loss function*.

2.2 Existing Methods, Memory Requirements, and Data Loading Costs

Consider the original GCN method [13]. This method evaluates embeddings for each node in the graph for each layer yielding memory requirement as $|V| \times F \times L$. In addition, the process needs to maintain the matrix A and the current values of W^l for each l. Thus, the total memory requirement will be $|V| \times F \times L + |A| + L \times F^2$. For large graphs, this can easily exceed the available memory on a single GPU. In this work, the authors have not discussed any method for partitioning the problem that will allow us to work on different parts of the graph. Besides large memory requirements, this method also suffers from large computational time cost.

Since the original GCN method was presented, several researchers have developed methods for improving the efficiency of the process [4–6,9,18] Most of these approaches involve the use of *mini-batches*, possibly together with sampling of the neighborhood. Unfortunately, these approaches do not sufficiently reduce memory requirements for most graphs, especially when the number of layers is large. In the mini-batch approach, consider a batch size of b. If the average degree of a node is d, then with L layers, there are $b \times d^L$ nodes for which embeddings need to be computed. Depending upon the value of b, d, and L, this number can easily approach $|V|$, resulting in memory requirements comparable to the original GCN method. Some reduction in the exponential growth of the number of layers can be achieved with sampling of the neighbors. For example, GraphSAGE [9] takes a fixed number of neighbors for each node. If this number is s ($s < d$), then the number of nodes for which embeddings need to be calculated reduces to $b \times s^L$. Note, however, that, as different nodes are selected to be part of the mini-batch for each epoch, we have one of the two possibilities. First, we store all nodes and their features on the device (such as the GPU). This limits the size of the graph that can be processed. The second possibility is to load the set of nodes that are part of the mini-batch and their neighborhood for each epoch. This, however, means high cost of reloading data for each epoch.

Two new efforts have specifically focused on the need of processing large graphs – ClusterGCN [6] and GraphSAINT [18]. We now describe these approaches with an emphasis of examining the data loading costs associated with them. ClusterGCN is an approach based on partitioning the graph, and subsequently, choosing a mini-batch from within a partition. The advantage of this approach is that nodes within a mini-batch are more likely to have common neighbors, thus allowing greater reuse of computations done on some of the nodes. Because of partitioning, this approach can also deal with very large graphs, which others approaches may not be able to handle. However, a hidden cost associated with this method in dealing with large graphs is that of data loading. If n partitions are created from the graph and the training is conducted over m epochs, each partition needs to be loaded m times during training.

GraphSAINT [18] can also handle very large graphs, but takes a different approach. Instead of choosing nodes that form a given mini-batch, it samples a smaller graph from the larger graph. Each epoch of the method works with one such sampled graph. Because the size of the sampled graph can be quite small, this method can also train very large graphs. However, there is a cost of sampling and loading the sampled graph for each epoch.

3 Overall Approach and Implementations

We discussed how the cost of loading either a partition or a k-step neighborhood of mini-batch vertices, or sampled subgraphs, can be quite high. To address this problem, we draw motivation from the previous work on *meta-learning* [14].

3.1 Background: Meta-learning Approach

In meta-learning, we assume there is a large number of tasks over the same dataset, and our goal is to optimize these tasks all together. In [14], a remarkably simple algorithm *Reptile* is proposed. We summarize the approach as Algorithm 1. Here τ denotes a task (line 2) and $U_\tau^k(\phi)$ (line 3) denotes the function that performs k gradient updates from the training algorithm on sampled (mini-batched) data starting with ϕ. This training is performed using Stochastic Gradient Descent (SGD) or Adam [12]. In line 4, we update ϕ, treating $\phi - \tilde{\phi}$ as the gradient. For this update, a parameter ϵ is used as the step size.

Algorithm 1. Reptile (serial version)

```
    Initialize $\phi$ (the vector of initial weights)
1:  for iteration $i = 1, 2, \ldots$ do
2:      Sample task $\tau$ with loss $\tilde{\phi}$
3:      Compute $\tilde{\phi} = U_\tau^k(\phi)$, denoting k steps (SGD or Adam)
4:      Update $\phi \leftarrow \phi + \epsilon(\tilde{\phi} - \phi)$
5:  end for
```

In [14], it has been argued that the Reptile converges towards a solution ϕ that is close (in Euclidean distance) to each task τ's manifold of the optimal solutions. As stated above, meta-learning is concerned with a large number of tasks that are being optimized together.

3.2 Our Approach

Algorithm 2. Large-Scale GCN training framework

```
    Input: GCN model, Graph $G(V, E)$, feature $X$, label $\bar{Y}$
    Output: GCN model with trained weights;

 1: $\Phi$: initialization parameters
 2: for $macro\_epoch = 1, 2, \ldots, T_{macro}$ do
 3:     Shuffle training nodes
 4:     for $s = 1, 2, \ldots$ do
 5:         Load or Generate $G_s(V_s, E_s)$
 6:         for $mini\_epoch = 1, 2, \cdots T_{micro}$ do
 7:             $\tilde{\phi} = U_s(\phi)$ , denoting a full-batch train on $G_s$
 8:         end for
 9:     end for
10: end for
```

Based on the discussion above, we can consider training a GCN on each subgraph as a learning task (denoted as τ_1, τ_2, ...) and then training the GCN on the original graph (denoted as G) as the meta-learning task. By doing this, we can perform more computation/training using one subgraph that is already loaded into the GPU memory, and thus saving the loading cost from CPU main memory, or disk, or even remote storage (through network). For each training task, we go through a specific number of training epochs T_{Total} before the convergence is reached. The total training epoch duration is divided into two parameters, *macro_epoch* and *micro_epoch*, such that

$$T_{Total} = T_{macro} \times T_{micro} \tag{1}$$

During each macro_epoch, one subgraph will be generated, uploaded on to GPU and trained for T_{micro} epochs. In this way, each subgraph data only need to be generated and uploaded on to GPU for a total of T_{macro} times. The overall framework is shown as Algorithm 2. This method involves loops over *macro_epochs*. Each iteration starts with loading or generating a subgraph G_s (line 3). This subgraph can be pre-computed or be constructed on the fly, as we will explain later. The size of the each subgraph will be adjusted to be able to fit GPU memory – since the GCN model is trained by aggregating node features from a subgraph, a larger subgraph is expected to provide more information for the learning task. Therefore, during each micro_epoch, each subgraph will be full-batch trained to utilize all of its information for a better prediction performance and $U_s(\phi)$ (line 7) denotes the function that performs one-step gradient full-batch training. Overall, the update in line 6–8 corresponds to T_{micro} steps full-batch training on the entire subgraph G_s. This is also the reason why we do not use the parameter ϵ (line 4 from Algorithm 1) in line 7 of Algorithm 2.

Our training technique can be applied to multiple GCN learning frameworks since it is orthogonal to both graph sampling/partition methods and

GCN architecture. Three different GCN training algorithm have been adopted under our framework to illustrate the effectiveness of our proposed training strategy: GraphSaint [18], GraphSage [9] and ClusterGCN [6]. The main difference between these GCN training algorithm is their distinct strategies of constructing subgraphs. We mainly applied GCN architecture from the previous GraphSaint work with all the hyper-parameters carefully tuned for each benchmark dataset [18]. We refer to the resulting algorithms as GraphSaint++, GraphSage++, and ClusterGCN++, respectively.

The GraphSaint is a graph sampling based algorithm. To generate representative subgraphs for efficient information aggregation during training, it uses samplers that aggregate nodes with high influence on each other and also sample edges [18]. Several samplers have been used, such as random node sampler, random edge sampler, and random walk based sampler[18]. In applying our approach here, in each macro_epoch, we first shuffle all the train nodes. A user selected sampler will be executed to sample train nodes to construct each subgraph. The data that is required to be uploaded on GPU for the training process includes the adjacency matrix of subgraph, node features, edge weights, and the node labels. A series of T_{micro} epochs full-batch training is performed to update the model weights. The data uploading and full-batch training in the *micro_epoch* phase will be similar in all three GCN training algorithms.

The GraphSage is an inductive framework that learns node embeddings with good generalization performance by utilizing a node's neighborhood information. More specifically, features of a node's neighborhood will be sampled and aggregated [9]. The topological structure of each node's neighborhood and the node features distribution will be learnt simultaneously [9]. Nodes can get information from its neighbors at multiple hops. The number of hops and the number of neighbor nodes on each hop are user defined parameters and can be specified by the Neighbor Number List $L = [S_1, S_2, \ldots]$. For example, $L = [10, 5]$ means we include two-hop neighbors with 10 neighbor train nodes on the first hop and 5 on the second. In applying our approach, we first randomly sample a specific number of train nodes at the beginning of each *macro_epoch*. We further expand the sampled nodes by further including their neighbor train nodes based on the array L to construct the subgraph. The other details are identical to the previous method.

The ClusterGCN framework exploits the graph clustering structure for SGD-based training for an improved memory and computation efficiency [6]. A graph clustering algorithm will be applied to partition the whole graph into disjoint clusters. These clusters will later be randomly recombined into multiple isolated subgraphs. During training, each train node can only utilize features of nodes which locate in the same isolated subgraph [6]. We first partition the training Graph into isolated clusters using METIS [11]. During each macro_epoch, clusters will be shuffled to reduce bias and a subgraph will be constructed by combining a specific number of clusters. Both the number of clusters and subgraphs are user defined hyper-parameters and the subgraph size should be chosen so it fits in GPU memory. The other details are the same as in other methods.

4 Experimental Results

Table 1. Dataset Details ("s" for single class and "m" for multiclass classification)

Dataset	Nodes	Edges	Feature	Classes	Train/Val/Test
Flickr	$89,250$	$899,756$	500	7 (s)	0.50/0.25/0.25
Reddit	$232,965$	$11,606,919$	602	41 (s)	0.66/0.10/0.24
Yelp	$716,847$	$6,977,410$	300	100 (m)	0.75/0.10/0.15
Amazon	$1,598,960$	$132,169,734$	200	107 (m)	0.85/0.05/0.10

4.1 Implementations and Setup

Our framework implementations are based on the GraphSaint architecture [18] with three major components: sampler, GCN model and subgraph generator. Inside each part, we incorporate implementation for three GCN training methods (GraphSaint [18], ClusterGCN [6] and GraphSage [9]) and we apply our strategy by separating the training process into two phases as described in Sect. 3.2, resulting in GraphSaint++, ClusterGCN++, and GraphSage++, respectively. The hyper-parameters obtained after the tuning process are listed in the Table 2. We use Adam [12] optimizer with learning rates carefully tuned for all our experiments. Dropout regularization is applied. GCN architecture is specified as $L \times F$, where L is the GCN depth and F is the *hidden dimension*, i.e. the dimension of latent representation in the GCN model.

The last column of Table 2 depends on a specific GCN method as explained below. For GraphSaint, we use the edge sampler and each edge will be sampled into a subgraph based on an independent decision [18]. The Edge budget in Table 2 is given as a sampling parameter to specify the expected number of edges in each subgraph [18]. For GraphSage neighborhood sampling, the previous work [9] shows that high learning performance can be obtained by including a neighbor number list $L = [S_1, S_2](S_1 \cdot S_2 \leq 500)$ and we are using $L = [S_1, S_2](S_1 \cdot S_2 \leq 500)$ in our work. The Node budget in Table 2 is to enforce a pre-defined budget on the subgraph size [9]. For ClusterGCN, we applied the strategy of stochastic multiple partitions [6] to reduce the bias and the diagonal enhancement technique to further improve the performance. In the subgraph generation procedure, isolated clusters are formed by using METIS clustering algorithm [11] and later recombined into subgraphs randomly without replacement. The number of isolated clusters N and the number of subgraphs K are user-defined sampling parameters as given in the table.

Our framework is implemented in Pytorch on CUDA 10.1. The sampling part for the GraphSaint++ and GraphSage++ is implemented in Cython 0.29.21. Our experiments are performed on nodes with Dual Intel Xeon8268s @2.9 GHz CPU and NVIDIA Volta V100 w/32 GB memory GPU and 384 GB DDR4 memory on OSC cluster [3]. Generation of subgraphs is performed in serial with 1 CPU core.

Table 2. Training configuration

GraphSaint++					
Dataset	Learning rate	Dropout	T_{Total}	GCN architecture	Edge budget
Flickr	2×10^{-4}	0.2	30	3×256	6000
Reddit	1×10^{-3}	0.1	100	4×128	6000
Yelp	1×10^{-3}	0.1	100	3×512	2500
Amazon	1×10^{-2}	0.1	30	3×512	2000
GraphSage++					
Dataset	Learning rate	Dropout	T_{Total}	GCN architecture	Node budget
Flickr	5×10^{-5}	0.2	15	2×256	8000
Reddit	1×10^{-3}	0.1	100	2×128	8000
Yelp	1×10^{-3}	0.1	100	2×512	5000
Amazon	2×10^{-4}	0.1	80	2×512	4500
ClusterGCN++					
Dataset	Learning rate	Dropout	T_{Total}	GCN architecture	$K(N)$
Flickr	1×10^{-3}	0.2	15	3×256	16 (64)
Reddit	2×10^{-3}	0.1	100	4×128	64 (256)
Yelp	2×10^{-3}	0.1	100	3×512	64 (256)
Amazon	2×10^{-3}	0.2	100	3×512	64 (256)

4.2 Datasets

We use four benchmark datasets, which have also been used in other recent efforts (for example, GraphSaint [18]). Detailed statistics of all datasets are listed in Table 1. Flickr and Reddit are used for single-class classification task, i.e., each node can only belong to a single class while Yelp and Amazon are for multi-class classification. Each dataset has a specific fraction for the split of training/validation/test data, which is shown in Table 1.

Flickr aims at classifying images based on descriptions and common properties of online images. A node in this graph stands for one image uploaded to Flickr. An edge between two nodes will be established if comment properties exist between two images such as geographic location and comments from the same users. **Reddit** utilize users' comments to generate predictions about online posts communities. Each node is one user and edges will be established based on the friendship between users. This dataset has more than 10 million edges. **Yelp** is about the categorization of business according to customer's reviews and friendship in the open challenge website. One node represents one user and an edge will be created between two nodes if two corresponding users are friends. **Amazon** categorizes types of products by referring to buyers' reviews and activities. A node corresponds to one product on the Amazon website. If two products share the same customer, then an edge will be created between them. This dataset has more than 100 million edges.

Table 3. Test F1 score for different T_{micro} (%)

T_{micro}	Flickr	Reddit	Yelp	Amazon
GraphSaint++				
1	50.19 ± 0.52	96.52 ± 0.03	65.10 ± 0.06	80.70 ± 0.04
5	47.39 ± 0.39	96.50 ± 0.03	64.81 ± 0.07	79.50 ± 0.09
10	–	96.43 ± 0.01	64.43 ± 0.01	78.76 ± 0.13
ClusterGCN++				
1	50.78 ± 0.15	96.01 ± 0.05	64.31 ± 0.09	80.97 ± 0.02
5	49.49 ± 0.42	95.89 ± 0.04	64.47 ± 0.08	80.85 ± 0.03
10	–	95.67 ± 0.11	64.34 ± 0.06	80.70 ± 0.02
GraphSage++				
1	50.53 ± 0.35	96.58 ± 0.04	64.61 ± 0.06	78.12 ± 0.03
5	50.81 ± 0.03	96.47 ± 0.04	64.41 ± 0.04	78.16 ± 0.05
10	–	96.34 ± 0.04	64.20 ± 0.06	78.09 ± 0.07

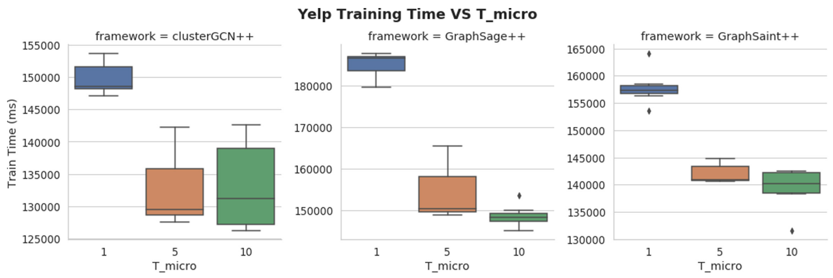

Fig. 1. Training time with different T_{micro} across different frameworks on Yelp

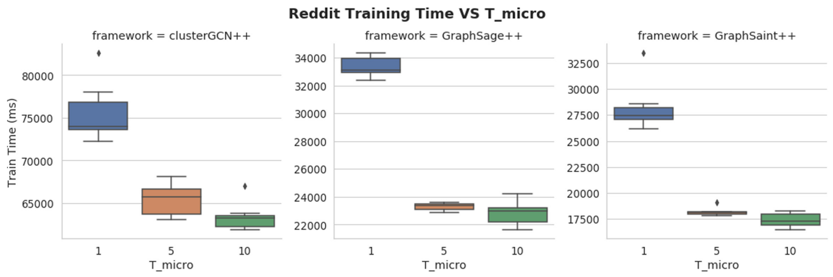

Fig. 2. Training time with different T_{micro} across different frameworks on Reddit

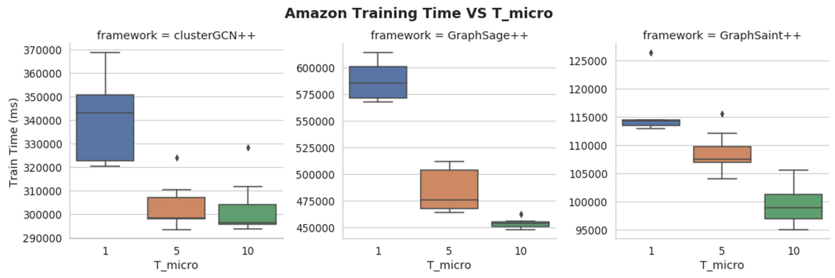

Fig. 3. Training time with different T_{micro} across different frameworks on Amazon

4.3 F1-Score

Our first experiment focused on evaluating the impact of T_{micro} on convergence. It should be noted that when T_{micro} is 1, the computations performed are identical to the original framework, i.e., GraphSage++ is same as GraphSage, and so on (though implementations are different).

In our experiments, as different GCN training algorithms have different convergence rates, we set a different value of T_{Total} for each combination of GCN algorithm and dataset. During the training, we periodically take a snapshot of the model and perform an evaluation on the validation dataset to record the convergence curve. As experiments with different datasets resulted in a very similar behavior, we show results only from the Yelp dataset in this paper. It turns out, as we increase the value of T_{micro}, the convergence did slow down, but only very marginally. Overall, we can see that use of higher values of T_{micro} parameter remains a feasible approach for training GCNs.

We compare the F1-score performance on test data among distinct values of T_{micro} in Table 3. Each data point is generated by 7 runs under the same hyper-parameter settings. Based on the validation F1-score, we choose the best model snapshot, i.e. the optimal model parameters, to perform evaluation on test data. First, we can see that the state-of-art results as reported from original

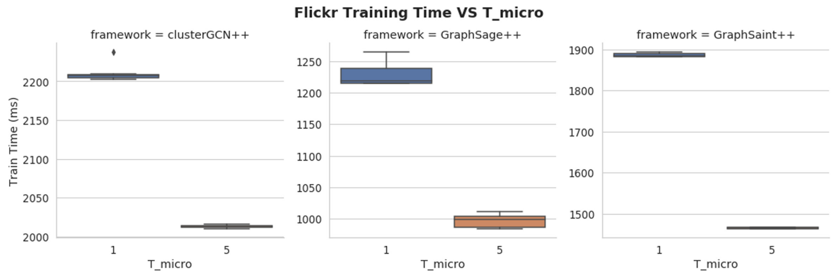

Fig. 4. Training time with different T_{micro} across different frameworks on Flickr

publications of these frameworks have been reproduced when T_{micro} is 1, i.e., when our implementation is simply reproducing original algorithm. Next, as reflected in Table 3, the influence of T_{micro} on test F1 score can vary through different frameworks. There are some cases where $T_{micro} > 1$ can outperform the baseline case ($T_{micro} = 1$). For example, ClusterGCN++ on Yelp obtains its best F1-score with $T_{micro} = 5$. Similar things happen to GraphSage++ on Flickr and Amazon. For the GraphSaint++, a larger T_{micro} does result in some loss of F1 score. Overall, across different combination of GCN training methods and datasets, decrease in F1 score is limited to at most 1–3%, again establishing that use of higher values of T_{micro} parameter remains a feasible approach for training GCNs.

A larger value of T_{micro} implies that we are more focusing on training each subgraph and we will iterate through different subgraphs less frequently. However, compared with the baseline case ($T_{micro} = 1$), we can still achieve good Test F1 score while maintaining a fast convergence speed with a larger T_{micro}. As long as the subgraph is well sampled to be representative of the target graph, we are able to obtain both fast convergence speed as reflected by the validation F1-score convergence curve and good generalization performance as indicated by the test F1-scores.

4.4 Training Times

Finally, we focus on the gains from training times.

The training time excludes all the data pre-processing such as sampling as in GraphSaint++, METIS partitioning as in ClusterGCN++ or neighbor nodes generation as in GraphSage++. It includes time cost of data uploading to the GPU device and the on-GPU computation of training loss and parameters updating. We investigated training time for each dataset using different training methods with multiple T_{micro} values as shown in Figs. 1, 2, 3 and 4. Seven runs are performed for each experimental task to include variations for the training time. The data loading time will shrink T_{micro} times with $T_{micro} > 1$. Significant savings on training time can be achieved especially when the data loading takes a relatively bigger portion in the training time as in Table 4. We see a proportional decrease of training time in Fig. 2. Relatively less train time saving have been achieved for Yelp in Fig. 1 and Amazon (Fig. 3). That is because they only have data loading time with a fraction around $10 - 15\%$ of their training time and GCN computation is their major time cost. That also explains why Flickr in Fig. 4 achieves more time saving for GraphSage++ and GraphSaint++ than it does for ClusterGCN++. Overall, the improvement from $T_{micro} = 5$ to $T_{micro} = 10$ remains limited and only the GraphSaint++ on Amazon as in Fig. 3 shows a relatively obvious difference. That is because a value as $T_{micro} = 5$ will reduce data loading time into a small enough fraction of training time so that further optimization with $T_{micro} = 10$ will not make a substantial difference.

Table 4. Data load time fraction of train time (%)

Framework	Flickr	Reddit	Yelp	Amazon
GraphSaint	27.41 ± 0.19	32.32 ± 3.63	11.76 ± 1.89	10.04 ± 1.58
ClusterGCN	10.34 ± 0.04	11.81 ± 0.95	14.94 ± 3.05	13.59 ± 0.34
GraphSage	23.06 ± 0.77	30.02 ± 1.10	14.58 ± 2.10	14.91 ± 2.39

5 Related Work

Besides GraphSage [9], ClusterGCN [6] and GraphSaint [18] that we have extensively discussed and built on, other prominent efforts are as follows. FastGCN [5] interprets the graph convolution as integral transforms of embedding functions under probability measures and applies importance sampling among graph vertices, though significant overhead could be induced by its sampling algorithm. S-GCN [4] reduces the neighborhood sampling size using a variance reduction technique. These both methods still have scalability issues due to the requirement of keeping all nodes' intermediate embeddings in memory. There has been previous work on developing an efficient out-of-core implementation of Convolution Neural Networks (CNNs) [1]. However, the computation and data access patterns for a CNN is very different from GCNs.

In one aspect, the idea of our work is similar to and related to the recent work of data echoing [7] and minibatch persistency [8]. However, these works are based on training using mini-batches, whereas we consider subgraph reuse. Subgraphs are typically much larger and have an internal structures, whereas the standard minibatch consists of randomized data points. In fact, the data echoing and minibatch persistency are mainly used in settings like CNN; to our best of knowledge, it has never been attempt in GCN. Also, we have given mathematical analysis based on the meta-learning, whereas data echoing [7] and minibatch persistency [8] never did.

6 Conclusions

In this paper, we have focused on the problem of training large-scale Graph Convolutional Networks (GCNs), which is becoming increasingly important. Drawing inspiration from the idea of *meta-learning*, we observe that training of a GCN using a single large graph can be viewed as a collection of training tasks over subgraphs or partitions, each of which fits into GPU memory. Based on this idea, we developed both an overall framework as well as three instanciations – converting three recent methods GraphSaint, GraphSage, and ClusterGCN, resulting into new algorithms GraphSaint++, GraphSage++, and ClusterGCN++, respectively. We have also shown mathematical analysis denoting why these methods are still able to converge, while reducing data loading costs.

We have carried out a detailed experimental evaluation of our three new algorithms using four graph datasets. We demonstrate how we are able to obtain comparable convergence and final F1 scores while reducing the data loading time by up to 90% and total training time up to 30%.

References

1. Awan, A.A., Hamidouche, K., Hashmi, J.M., Panda, D.K.: S-caffe: co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, SIGPLAN Notices, vol. 52, no. 8, pp. 193–205, January 2017
2. Cai, H., Zheng, V.W., Chang, K.C.C.: A comprehensive survey of graph embedding: Problems, techniques and applications. CoRR, abs/1709.07604 (2017)
3. Ohio Supercomputer Center. Ohio supercomputer center (1987)
4. Chen, J., Zhu, J., Song, L.: Stochastic training of graph convolutional networks with variance reduction. In: ICML, pp. 941–949 (2018)
5. Chen, J., Ma, T., Xiao, C.: FastGCN: fast learning with graph convolutional networks via importance sampling. In: International Conference on Learning Representations (ICLR) (2018)
6. Chiang, W.L., Liu, X., Si, S., Li, Y., Bengio, S., Hsieh, C.J.: Cluster-GCN. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD), July 2019
7. Choi, D., Passos, A., Shallue, C.J., Dahl, G.E.: Faster neural network training with data echoing. CoRR, abs/1907.05550 (2019)
8. Fischetti, M., Mandatelli, I., Salvagnin, D.: Faster SGD training by minibatch persistency. CoRR, abs/1806.07353 (2018)
9. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems, vol. 30, pp. 1024–1034 (2017)
10. Huang, W., Zhang, T., Rong, Y., Huang, J.: Adaptive sampling towards fast graph representation learning. In: Advances in Neural Information Processing Systems, pp. 4558–4567 (2018)
11. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
12. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: 3rd International Conference for Learning Representations (2015)
13. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (ICLR), abs/1609.02907 (2017)
14. Nichol, A., Schulman, J.: Reptile: a scalable metalearning algorithm. arXiv preprint arXiv:1803.02999, vol. 2, no. 3, p. 4 (2018)
15. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. CoRR, abs/1901.00596 (2019)
16. Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W.L., Leskovec, J.: Graph convolutional neural networks for web-scale recommender systems. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018 (2018)

17. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.: Accurate, efficient and scalable graph embedding. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2019
18. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.: Graphsaint: graph sampling based inductive learning method. In: International Conference on Learning Representations (ICLR) abs/1907.04931 (2020)