OPT2020: 12th Annual Workshop on Optimization for Machine Learning

Decoupled Greedy Learning of Graph Neural Networks

Yewen Wang WYW10804@CS.UCLA.COM
Jian Tang JIAN.TANG@HEC.CA
Yizhou Sun YZSUN@CS.UCLA.EDU
Guy Wolf WOLFGUY@MILA.QUEBEC

Abstract

Graph Neural Networks (GNNs) become very popular for graph-related applications due to their superior performance. However, they have been shown to be computationally expensive in large scale settings due to the recursive computation while obtaining node embeddings. To address this issue, in this work, we introduce a decoupled greedy learning method for GNNs (DGL-GNN) that decouples the GNN into smaller modules and associates each module with greedy auxiliary objectives. Our approach allows GNN layers to be updated during the training process without waiting for feedback from successor layers, thus making parallel GNN training possible. Our method achieves improved efficiency without significantly compromising model performances, which would be important for time or memory limited applications. Further, we propose a lazy-update scheme during training to further improve its efficiency. We empirically analyse our proposed DGL-GNN model, and demonstrate its effectiveness and superior efficiency through a range of experiments. Compared to the sampling-based acceleration, our model is more stable, and we do not have to trade-off between efficiency and accuracy. Finally, we note that while here we focus on comparing the decoupled approach as an alternative to other methods, it can also be regarded as complementary, for example, to sampling and other scalability-enhancing improvements of GNN training.

1. Introduction

Graph Neural Networks (GNN) have been shown to be highly effective in graph-related tasks, such as node classification [13], graph classification [20], and graph matching [1]. Given a graph of arbitrary size and attributes, GNNs obtain informative node embeddings by first aggregating information from the neighbors of each node, then transforming the aggregated information. As a result, GNNs can fuse together the topological structure and node features of a graph, and have thus became dominant models for graph-based applications.

Despite its superior representation power, the graph convolution operation has been shown to be expensive when GNNs become deep and wide [6]. Therefore, training a deep GNN model is challenging for large and dense graphs. Since deep and wide GNNs are becoming increasingly important with the emergence of classification tasks on large graphs, such as the newly proposed OGB datasets [11], and semantic segmentation tasks in [14], we focus here on studying methods for alleviating computational burdens associated with large-scale GNN training. Several sampling-based strategies have been proposed during the past years to alleviate this computation issue of large-scale GNNs such as [6–9, 22]. However, these methods need us to balance the computation cost and the down-stream task performance.

In addition to the inefficiency brought by the graph convolution operation, as discussed in [3], the sequential nature of standard backpropagation also leads to inefficiency. As pointed out in [12], backpropagation for deep neural networks suffers an update-locking problem, which means each layer heavily relies on upper layers' feedback to update itself, and thus, it must wait for the information to propagate through the whole network before updating. This would be a great obstacle for GNN layers to be trained in parallel to alleviate computation pressure under time and memory constraint, and would prohibit the GNN training to be trained in an asynchronous setting.

In this work, using semi-supervised node classification as an example, we show that the greedy learning would help to decouple the optimization of each layer in GNNs and enable GNNs to achieve update-unlocking . By using this decoupled greedy learning for GNNs, we can achieve parallelization of the network layers, which would make the model training much more efficient and would be very important for time or memory limited applications. Moreover, we propose to use a lazy-update scheme during training, which is to exchange information between layers after a certain number of epochs instead of every epoch, this will further improve the efficiency while not sacrificing much performance. We run a set of experiments to justify our model, and show its great efficiency on all benchmark datasets.

Our main contributions can be summarized as follows. **First**, we introduce a decoupled greedy learning algorithm for GNNs that achieves update-unlocking and enables GNN layer to be trained in parallel. **Next**, we propose to leverage a lazy-update scheme to improve the training efficiency. We evaluate our proposed training strategy thoroughly on benchmark datasets, and demonstrate it has superior efficiency while not sacrificing much performance. **Finally**, our method is not limited to the GCN and the node classification task, but can be combined with other scalability-enhancing GNNs and can be applied to other graph-related tasks.

2. Related Work

Before discussing our proposed approach, we review related work on efficient training strategies for GNNs. The details of each model and their computational complexities are given in Appendix A.

2.1. Deep Graph Convolutional Network (DeepGCN)

Graph convolutional network [GCN, 13] is one of the most popular models for graph-related tasks. Given an undirected graph $\mathcal{G}$ with node feature matrix $\boldsymbol{X} \in \mathbb{R}^{N\times D}$ and adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ where N is node number and D is feature dimension, let $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$, $\tilde{\boldsymbol{D}}$ be a diagonal matrix satisfying $\tilde{\boldsymbol{D}}_{i,i} = \sum_{j=1}^{N} \tilde{\boldsymbol{A}}_{i,j}$, and $\boldsymbol{F} = \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$ be the normalized $\tilde{\boldsymbol{A}}$, then, the l-th GCN layer will have the output $\boldsymbol{H}^{(l)}$ as $\boldsymbol{H}^{(l)} = \sigma(\boldsymbol{F}\boldsymbol{H}^{(l-1)}\boldsymbol{W}^{(l)})$, where σ is the non-linear transformation, and $\boldsymbol{W}^{(l)}$ is the trainable weight matrix at layer l.

As pointed out in [15], when GCN becomes deep, it will suffer severe over-smoothing problem, which mean the nodes will become not distinguishable after stacking too many network layers. However, for applications such as semantic segmentation [14] or classification tasks on large datasets [11], we do need deeper GCN models. Therefore, we follow the work of [14], alleviating over-smoothing problem by adding residual links between GCN layers and obtain the deepGCN model. The l-th layer of our network model will be $\boldsymbol{H}^{(l)} = \sigma(\boldsymbol{F}\boldsymbol{H}^{(l-1)}\boldsymbol{W}^{(l)}) + \boldsymbol{H}^{(l-1)}$.

2.2. Efficient GNN Training

Several methods have been proposed to alleviate this computation issue of large-scale GNNs. GraphSAGE [9] took the first step to leverage a neighborhood sampling strategy for GNNs training, which only aggregates a sampled subset of neighbors of each node in the graph convolution operation. However, though this sampling method helps reduce memory and time cost for shallow GNNs, it computes the representation of a node recursively, and the node's receptive field grows exponentially with the number of GNN layers, which may make the memory and time cost even goes larger for deeper GNNs when the sample number is big. The work of [6, 7, 22] developed sampling-based stochastic training methods to train GNNs more efficiently and avoid this exponential growth problem. [8] proposed a batch learning algorithm by exploiting the graph clustering structure.

In addition to these batch-learning methods, the efficiency of GNN training can be improved with a layer-wise strategy. Layerwise learning for neural networks was first discussed in [5, 10] and was applied to CNNs and achieved impressive results in [3, 4]. Recently, [21] explored it for GNN training. The key idea is to train GNNs layer by layer sequentially. For a L-layer GNN, we first train its first layer with an auxiliary classifier until it get fully converged, then we fix this layer and start to optimize next layer, we do the same things for all L layers. This layerwise training saves us a lot of memory, since it only requires us to focus on one layer and store this layer's activation results. This layerwise training scheme also saves us a lot of time, because it decouples the two components in the per-layer feed-forward graph convolution: aggregation and transformation, and it only need to conduct the aggregation step once for each layer, which greatly reduce the time cost.

3. Proposed Approach

3.1. Model Architecture

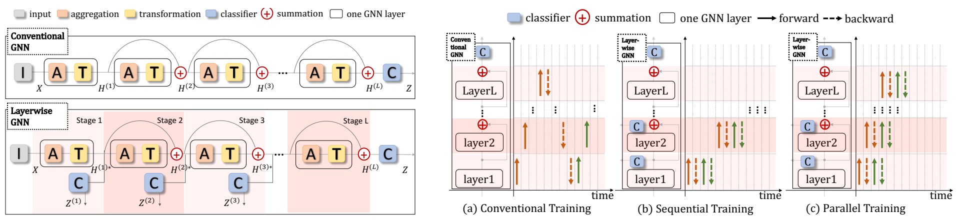


Figure 1: **(Left)** High level framework of conventional deepGCN (upper) and layerwise deepGCN (lower). The aggregation step (A) corresponds to $\boldsymbol{F}\boldsymbol{H}^{(l-1)}$ operation and the transformation step corresponds to $\sigma(\cdot\boldsymbol{W}^{(l)})$ operation. **(Right)** Signal propagation process for 3 GNN training methods. Arrows of different colors represents different batches of data.

As mentioned in section 2.1, we introduce our proposed algorithm with deepGCN model since the residual link would help to alleviate GCN's over-smoothing problem when it goes deep . There exists two ways to train such GCN model: conventional training and sequential layerwise training. We illustrate these two strategies with the high-level framework shown in Figure 1. For conventional training, we jointly optimize the learnable parameters in all layers and in the classifier. For layerwise training, we break the training for a L-layer GNN into L sequential stages, each stage has to wait all its previous layers to get fully converged to start training. Note that, the sequential layerwise training can save time and memory while not compromising too much performance, this suggests its promising applications in large scale models under hardware and time constraints. We now consider, *whether we can extend it to a parallel version, so that the efficiency can be further improved?* Interestingly, as shown in the following sections, we find the answer is affirmative.

3.2. Decoupled Greedy Learning Algorithm

To enable parallel GNN training, the most challenging problem is update-locking. Before updating one layer, we have to wait after the signal has been passed through all its successors, which would bring inefficiency. To alleviate this problem, we decouple the GNN model into different layers, associating each layer with an auxiliary classifier, which is a MLP layer with softmax activation, and assigning a per-layer greedy objective. Then, with the output activation of a given layer, we can leverage the auxiliary classifier to optimize the per-layer objective and therefore can update the current layer without any feedback from its successors while the rest layers are still in the forward process. We name our training strategy as **D**ecoupled **G**reedy **L**earning of **GNN**s (DGL-GNN).

With our DGL-GNN, we achieve update-**un**locking, and therefore can enable parallel training for layerwise GNNs. For clarity, we provide Figure 1 to compare the signal propagation process of the conventionally trained GNN, sequentially trained layerwise GNN, and the parallel trained GNN. With this illustration, we can know that the key difference between the sequentially training method and our method would be whether the signal from previous layers are fully converged or not, and the parallel training of layerwise GNN can avoid the case in which one layer is forwarding or back-propagating the signal while other layers are idle. Therefore, given same number of batches of data, the parallel version would converge earlier than the conventional and the sequential version.

We then empirically observed that, without passing the signal to next layer immediately after the forward process, we still get same-level performance. Thus, we find that the efficiency of DGL-GNN can be further improved by leveraging an **L**azy **U**pdate scheme (LU-DGL-GNN). Instead of using the up-to-date activation output from its predecessor, one layer can use the history activation to learn its parameters and only update the history activation a few times during the overall training process. Then, same as sequential trained layerwise GNN, we only need to conduct the aggregation step for one time after each update, this saves us a lot time.

For clarity purpose, we show our algorithms in a more formal way in Appendix A.1. And to justify the rationality of our proposed model, we present an analogy of our approach to the classic Block Coordinate Descent (BCD) optimization strategy and its variants [17–19] in Appendix A.3.

4. Empirical results

We evaluate our proposed algorithms with the multi-class node classification task. However, it should be noted that the decoupled greedy learning method can also be applied in other graph-related tasks and is not limited to node classification. We use three public datasets for evaluation: cora, citeseer, and pubmed [16] . We introduce these datasets and summarize their statistic in Appendix A.4. We compare our method against several baseline models introduced in Section 2: Full-Batch GCN [13], FastGCN [7], LADIES [22], and LGCN [21]. For all the methods, we use the same deepGCN model architecture and set all the hidden-dimension as 128. We follow the public implementations and their parameter settings of all the baselines. Further implementation details are in Appendix A.4. We evaluate the performance of different methods with the following evaluation metrics: **Accuracy** (%): The micro F1-score of the test data at the convergence point. **Memory**

Table 1: Comparison of LU-DGL-GCN with baseline methods on benchmark datasets. We set the number of layers as 5, and set $T_{lazy} = 50$ here for LU-DGL-GCN. We do mini-batch training for all these methods, and set the batch number as 10 and batch size as 512.

Dataset	Method	Accuracy(%)	Total Time(s)	Mem(MiB)
Cora	GCN	77.8 ± 1.3	42.2 ± 1.0	31.7
	LADIES	78.8 ± 0.8	31.5 ± 0.8	3.1
	FastGCN	55.3 ± 4.8	36.8 ± 2.1	3.1
	LGCN	80.4 ± 0.9	$119.611.5$	6.9
	LU-DGL-GCN	78.0 ± 1.3	14.1 ± 0.4	6.9
Citeseer	GCN	65.5 ± 2.4	33.1 ± 1.2	67.9
	LADIES	66.6 ± 1.2	32.5 ± 1.3	5.9
	FastGCN	35.9 ± 1.0	34.5 ± 1.7	5.9
	LGCN	67.1 ± 1.7	107.9 ± 5.8	14.7
	LU-DGL-GCN	64.8 ± 6.3	13.9 ± 0.2	14.7
Pubmed	GCN	74.8 ± 2.6	74.8 ± 2.6	137.9
	LADIES	77.9 ± 2.4	33.9 ± 1.5	1.9
	FastGCN	41.2 ± 0.5	34.6 ± 1.4	1.9
	LGCN	76.2 ± 1.6	141.8 ± 12.8	29.1
	LU-DGL-GCN	76.9 ± 5.3	14.9 ± 1.0	29.2

(MiB): The maximum per-GPU memory cost during training. **Total Running Time (s)**: The total training time (exclude validation) before convergence.

We summarize the classification performance results in Table 4, which demonstrates the efficacy of our approach. We can see that, our LU-DGL-GCN is the fastest among all the baselines. We can also find that our proposed method has a clear advantage which is to save the per-GPU memory. Compared to sampling-based baselines (LADIES and FastGCN), our LU-DGL-GCN has a better accuracy and is faster, while only need slightly more per-GPU memory. Compared to LGCN which is a sequantially-trained layerwise GCN, our LU-DGL-GCN has a clear advantage on the time cost. Our experimental results together with our analysis in previous sections indicate that our LU-DGL-GNN would be very helpful for time and memory limited applications, especially for the scenarios in which we need deep models.

5. Conclusions

In this paper, we focus on the efficiency issue of GNN training in large-scale applications and present a decoupled greedy GNN learning strategy. Our proposed DGL-GNN and LU-DGL-GCN model achieves update-**un**locking by introducing greedy auxiliary objectives during training, and enables parallelization by decoupling the GNN into smaller modules. The LU-DGL-GNN leverages a lazy-update scheme during training and further improve the model efficiency. We note that while we introduce our proposed methods with deepGCN model and use the semi-supervised node classification task as an example, our methods are not limited to this setting, and can be applied to other GNNs and graph-related downstream tasks. Further, while here we focus on comparing the decoupled approach as an alternative to other sampling-based methods respect to their accuracy and efficiency, these approaches can be regarded as complementary to each other. By combining the decoupled greedy learning method with other scalability-enhancing improvements of GNN training, the computation cost would be further reduced, which poses a promising direction for future work.

6. Acknowledgement

This work was partially funded by IVADO Professor startup & operational funds, IVADO Fundamental Research Project grant PRF-2019-3583139727 [*G.W.*], NSF III-1705169, NSF CAREER Award 1741634, NSF 1937599, DARPA HR00112090027, Okawa Foundation Grant, and Amazon Research Award [*Y.S.*].

References

[1] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 384–392, 2019.

[2] Amir Beck and Luba Tetruashvili. On the convergence of block coordinate descent type methods. *SIAM journal on Optimization*, 23(4):2037–2060, 2013.

[3] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Decoupled greedy learning of cnns. *arXiv preprint arXiv:1901.08164*, 2019.

[4] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Greedy layerwise learning can scale to imagenet. In *International conference on machine learning*, pages 583–593. PMLR, 2019.

[5] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

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

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

[8] 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. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 257–266, 2019.

[9] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[10] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[11] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[12] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *International Conference on Machine Learning*, pages 1627–1635. PMLR, 2017.

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

[14] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In *Proceedings of the IEEE International Conference on Computer Vision*, pages 9267–9276, 2019.

[15] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. *arXiv preprint arXiv:1801.07606*, 2018.

[16] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[17] Hao-Jun Michael Shi, Shenyinying Tu, Yangyang Xu, and Wotao Yin. A primer on coordinate descent algorithms. *arXiv preprint arXiv:1610.00040*, 2016.

[18] Stephen J Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

[19] Tong Tong Wu, Kenneth Lange, et al. Coordinate descent algorithms for lasso penalized regression. *The Annals of Applied Statistics*, 2(1):224–244, 2008.

[20] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in neural information processing systems*, pages 4800–4810, 2018.

[21] Yuning You, Tianlong Chen, Zhangyang Wang, and Yang Shen. L2-gcn: Layer-wise and learned efficient training of graph convolutional networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2127–2135, 2020.

[22] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *Advances in Neural Information Processing Systems*, pages 11249–11259, 2019.

Appendix A. Appendix

A.1. Algorithms

Following our notations in section 2, we denote by $\boldsymbol{F}$ the normalized adjacency matrix, $\boldsymbol{H}^{(l)}$ the output activation of l-th layer, and $\boldsymbol{W}^{(l)}$ the learnable parameters for l-th layer. Plus, let $\boldsymbol{Y}$ be the labels, $\Theta^{(l)}$ be the parameters for l-th layer's classifier, and $loss$ be the cross-entropy loss. Then, we have the per-layer objective function: $loss_{(\boldsymbol{W}^{(l)},\Theta^{(l)})}(\boldsymbol{Y}, \boldsymbol{H}^{(l-1)}, \boldsymbol{F}; \boldsymbol{W}^{(l)}, \Theta^{(l)})$. We now formally define our DGL-GNN training method in Algorithm 1 in Appendix A.1. Note that, the inner for-loop can be done in a parallel manner, i.e., when the $l-$th layer is working on the backward process

Algorithm 1 Decoupled Greedy Learning (DGL) of GNNs

```
Require: Normalized Adjacency Matrix $\boldsymbol{F}$; Feature Matrix $\boldsymbol{X}$; Labels $\boldsymbol{Y}$; Total Number of Iterations $T$; Total Number of Layers $L$.
1: Initialize: $\boldsymbol{H}^{(0)} = \boldsymbol{X}$;
2: for $t = 1$ to $T$ do
3:   for $l = 1$ to $L$ do
4:     $\boldsymbol{H}^{(l)} = \sigma(\boldsymbol{F}\boldsymbol{H}^{(l-1)}\boldsymbol{W}^{(l)})$   // Get node embeddings and store them as $\boldsymbol{H}^{(l)}$.
5:     $(\boldsymbol{W}^{(l)}, \Theta^{(l)}) \leftarrow$ Update with $\nabla loss_{(\boldsymbol{W}^{(l)},\Theta^{(l)})}(\boldsymbol{Y}, \boldsymbol{H}^{(l-1)}, \boldsymbol{F}; \boldsymbol{W}^{(l)}, \Theta^{(l)})$ // Update parameters.
6:   end for
7: end for
```

Algorithm 2 Decoupled Greedy Learning (DGL) of GNNs with Lazy Update Scheme

```
Require: Normalized Adjacency Matrix $\boldsymbol{F}$; Feature Matrix $\boldsymbol{X}$; Labels $\boldsymbol{Y}$; Total Number of Iterations $T$; Total Number of Layers $L$; Waiting time $T_{lazy}$.
1: Initialize: $\hat{\boldsymbol{H}}^{(0)} = \boldsymbol{F}\boldsymbol{X}$;
2: for $t = 1$ to $T$ do
3:   for $l = 1$ to $L$ do
4:     $\boldsymbol{H}^{(l)} = \sigma(\hat{\boldsymbol{H}}^{(l-1)}\boldsymbol{W}^{(l)})$   // Get node embeddings.
5:     $(\boldsymbol{W}^{(l)}, \Theta^{(l)}) \leftarrow$ Update with $\nabla loss_{(\boldsymbol{W}^{(l)},\Theta^{(l)})}(\boldsymbol{Y}, \hat{\boldsymbol{H}}^{(l-1)}; \boldsymbol{W}^{(l)}, \Theta^{(l)})$ // Update parameters.
6:     if ($t$ mod $T_{lazy} == 0$) then
7:       $\hat{\boldsymbol{H}}^{(l)} = \boldsymbol{F}\boldsymbol{H}^{(l)}$ // Get propagated node embeddings and store them as $\hat{\boldsymbol{H}}^{(l)}$.
8:     end if
9:   end for
10: end for
```

as given in line 5, the $(l+1)-$th layer can start forward propagation as given in line 4. Therefore, we claim our DGL-GNN algorithm can achieve update-**un**locking.

We denote by $\hat{\boldsymbol{H}}^{(l)}$ the aggregated stored history activation for layer l. We now formally define the LU-DGL-GNN method in Algorithm 2 in Appendix A.1.

A.2. Efficient GNN Training Methods

A.2.1. RELATED WORKS

To alleviate the expensive computation issue of GNN introduced in previous section, a lot of literature has proposed sampling-based batch-learning algorithms to train GNNs more efficiently. We introduce those methods in the following and summarize their complexities in Table 2. We refer the reader to Appendix A.2.2 for detailed computation.

GraphSAGE [9] introduced a node sampling strategy (NS), which is to randomly sample s neighbors for each node at each layer, then, for each node, instead of aggregating embeddings of all its neighbors, we only aggregate the sampled ones. VRGCN [6] also followed this NS strategy, but it further proposed to leverage history activation to reduce the variance of the estimator. Though NS

Table 2: Summary of Complexity. Here $\bar{D}$ denotes the average degree, b denotes the batch size, s_{node} and s_{layer} are the number of sampled neighbors in NS and IS respectively, K is the dimension of embedding vectors (for simplicity, assume it is the same across all layers), L is the number of layers, N is the number of nodes in the graph, $\boldsymbol{A}$ is the adjacency matrix, T is the number of iterations, T_{wait} is the waiting time for LU-DGL-GCN.

Methods	Memory (per GPU)	Time
Full-Batch GCN [13]	$\mathcal{O}(LNK + LK^2)$	$\mathcal{O}(TL\|\boldsymbol{A}\|_0 K + TLNK^2)$
GraphSage [9]	$\mathcal{O}(bKs_{node}^{L-1} + LK^2)$	$\mathcal{O}(bTKs_{node}^{L} + bTK^2 s_{node}^{L-1})$
VR-GCN [6]	$\mathcal{O}(LNK + LK^2)$	$\mathcal{O}(b\bar{D}TKs_{node}^{L-1} + bTK^2 s_{node}^{L-1})$
FastGCN [7]	$\mathcal{O}(LKs_{layer} + LK^2)$	$\mathcal{O}(TLKs_{layer}^2 + TLK^2 s_{layer})$
LADIES [22]	$\mathcal{O}(LKs_{layer} + LK^2)$	$\mathcal{O}(TLKs_{layer}^2 + TLK^2 s_{layer})$
ClusterGCN [8]	$\mathcal{O}(bLK + LK^2)$	$\mathcal{O}(TL\|\boldsymbol{A}\|_0 K + TLNK^2)$
L2GCN [21]	$\mathcal{O}(NK + 2K^2)$	$\mathcal{O}(L\|\boldsymbol{A}\|_0 K + 2TLNK^2)$
LU-DGL-GCN (ours)	$\mathcal{O}(NK + 2K^2)$	$\mathcal{O}(T\|\boldsymbol{A}\|_0 K/T_{wait} + 2TNK^2)$

scheme has smaller complexity compared to full-batch GNN, there exists redundant computation and the complexity grows exponentially with the layer number.

Layer-wise importance sampling strategy (IS) would be a more advanced method for efficient GNN training. FastGCN [7] proposed to sample nodes for each layer with a degree-based sampling probability in order to solve the scalability issue in NS. The work of LADIES [22] leveraged IS idea as well, but it proposed a layer-dependent importance sampling scheme, which enjoys a smaller variance while maintaining same level complexity as FastGCN. Though this IS is better than NS in general, but we may still have to trade-off the complexity with the performance (i.e. accuracy for classification tasks), since using a large sample number would be helpful for the performance and increase the computation cost and vice versa.

Except for ther aforementioned methods, we also has ClusterGCN [8], which proposes to partition the graph into several clusters then randomly select multiple clusters to form a batch to train the GNN. Though this would allows us to train much deeper GCN without much time and memory overhead, the stability of the performance of this approach would be hard to guarantee, since the performance would heavily depends on the graph clustering settings.

A.2.2. COMPLEXITY ANALYSIS

In this section, we explain how we compute the memory and time complexity for baseline models in Table 2.

All the aforementioned baseline methods's memory cost consists two parts: intermediate embedding matrices storage and weight matrices storage. The weight matrices always need $\mathcal{O}(LK^2)$ since it has to store L weight matrices with dimension $K \times K$. The intermediate embedding has different memory cost for different methods. In terms of time complexity, it also consists two parts: the aggregation time cost and transformation time cost. These two parts varies for different methods.

Full-batch GCN stores all the intermediate embedding matrices for all the L layers, and each matrice has N nodes with dimension K, so its memory complexity for intermediate embedding storage would be $\mathcal{O}(LNK)$. Therefore, its total memory complexity is $\mathcal{O}(LNK + LK^2)$. In terms of time complexity, the propagation step which is a sparse-matrix multiplication has time complexity $\mathcal{O}(\|\boldsymbol{A}\|_0 K)$, and the transformation step which is a dense-matrix multiplication has time complexity $\mathcal{O}(NK^2)$. During the training for L layers and T iterations, the total time complexity would be $\mathcal{O}(TL\|\boldsymbol{A}\|_0 K + TLNK^2)$.

For GraphSAGE, it has to store $\mathcal{O}(bs_{node}^{L-1})$ K-dimension node embeddings, so it has a total memory complexity $\mathcal{O}(bKs_{node}^{L-1} + LK^2)$. In terms of time complexity, for each batch, to update one node, we have to update $\mathcal{O}(s_{node}^{L-1})$ activations, each needs $\mathcal{O}(\mathcal{K}\int_{\wr\wr\sqcap})$ for aggregation and K^2 for transformation. Therefore, the total time complexity for GraphSAGE is $\mathcal{O}(bTKs_{node}^{L} + bTK^2 s_{node}^{L-1})$.

For VR-GCN, it stores all historical activations, which takes a memory of $\mathcal{O}(LNK)$ and will lead to the total memory complexity $\mathcal{O}(LNK + LK^2)$. The time complexity can be analyzed similarly as GraphSAGE, which would be $\mathcal{O}(bT\bar{D}Ks_{node}^{L-1} + bK^2 s_{node}^{L-1})$.

For FastGCN, it only has to store b node embeddings in the last layer, and $(L-1)s_{layer}$ node embeddings in the previous $L-1$ layers, so the total memory complexity would be $\mathcal{O}(bK + (L-1)Ks_{layer}) + LK^2)$. In terms of time complexity, we need $\mathcal{O}(bTKs_{layer} + (L-1)TKs_{layer}^2)$ for aggregation and $\mathcal{O}((bTK^2 + (L-1)Ts_{layer})K^2)$ for transformation. Note that, we have $b < s_{layer}$, so we ignore the relatively small terms and get the total memory complexity $\mathcal{O}(LKs_{layer} + LK^2)$, and total time complexity $\mathcal{O}(TLKs_{layer}^2 + TLK^2 s_{layer})$.

For LADIES, its is also a layer-wise sampling method. Though it has different sampling strategy as FastGCN, their memory cost and time cost are the same. There fore, LADIES also has memory complexity $\mathcal{O}(LKs_{layer} + LK^2)$ and time complexity and $\mathcal{O}(TLKs_{layer}^2 + TLK^2 s_{layer})$.

A.2.3. COMPLEXITY ANALYSIS FOR LU-DGL-GCN

As shown in Table 2, our proposed methods achieve a lower complexity compared to the conventional training and other baselines. Note that DGL-GCN can be regard as a special case in which $T_{wait} = 1$, i.e. we update the stored activation every epoch. Therefore, we focus on the complexity justification for LU-DGL-GCN.

For time complexity, first, we know that the training process consists two fundamental operations: aggregation and transformation. The time complexity of aggregation is $\mathcal{O}(\|\boldsymbol{A}\|_0 K)$, and the time complexity of the transformation step is $\mathcal{O}(NK^2)$. Then, we note that, for LU-DGL-GCN, during the full learning process, for each layer, we have to do aggregation T/T_{wait} times, and we have to do the transformation $2T$ times because this step should be conducted for both the GNN layer and the auxiliary classifier. Since the computation for each layer is done in parallel, we know that the overall time complexity for LU-DGL-GCN should be $\mathcal{O}(T\|\boldsymbol{A}\|_0 K/T_{wait} + 2TNK^2)$. In practice, if we put different layers on different GPUs and do the training in parallel, there would be some extra non-negligible time cost for GPU communication.

For memory complexity, it also consists two components. We have to store two things for each layer during the training: the history activation and the intermediate learnable weight matrices for GNN and for the auxiliary classifier. The activation takes $\mathcal{O}(NK)$, and the two types of weight matrix take $\mathcal{O}(2K^2)$ space. Again, when we do the training in a parallel fashion and assign the layers to different machines, the per-GPU memory would only be $\mathcal{O}(NK + 2K^2)$, which is significantly reduced compare to most of the existing baselines.

We also oberve that, the DGL-GCN can be analog to the synchronous parallel BCD and the LU-DGL-GCN can be regard as an analogy of asynchronous parallel BCD. Note that our DGL-GCN and LU-DGL-GCN can be implemented in a parallel fashion, and their key difference is whether all the layers share a consistency and up-to-date information. Therefore, if we make the same analogy of learnable parameters and the coordinate blocks as in the above sequential version, then it would be easy to find the similarity between parallel BCD and our decoupled greedy learning methods. With such analogy, it would allow us to leverage existing theorems for BCD optimization to better understand and analyze the DGL-GCN and LU-DGL-GCN.

A.3. Supplementary Discussion of Coordinate and Block Coordinate Descent

A.3.1. COORDINATE DESCENT ALGORITHMS

Coordinate descent (CD) is a classic iterative optimization algorithm that solves an optimization problem by approximately minimizing the objective along each coordinate directions successively. In each iteration, it would choose one variable, fix the other components, then optimizing the objective with respect to only the single variable. By doing so, we only need to solve a lower dimensional minimization problem at each iteration, which would be easier. CD algorithm has been discussed in various literatures and has been used in applications for a long time [17–19].

Block coordinate descent (BCD) is an extension of CD method. The difference of BCD and the conventional CD is that BCD will do the searching along a coordinate hyperplane instead of a single coordinate direction [2], i.e. it groups variables into blocks, and approximately minimizing the objective with respect to only one block of variables at each iteration while fixing the others.

For BCD algorithm, there exists its parallel implementations. As introduced in the work of [18], we can categorize them into two types: synchronous and asynchronous. For synchronous parallel BCD, we partition the computation into pieces and put different pieces on different processors, each processor will update a part of the variables in parallel, then a synchronization step should be conducted to guarantee the consistency of the information shared among all processors before further computation. For asynchronous setting, the difference is we do not have to do the synchronization. As discussed in Section A.3.2, our proposed method can be regarded as analogous to BCD and its aforementioned variants.

A.3.2. ANALOGY TO BLOCK COORDINATE DESCENT

To justify the rationality of the proposed model, we present here an analogy of our decoupling approach to the classic Block Coordinate Descent (BCD) optimization strategy and its variants [17–19], which for completeness are discussed in Appendix A.3. We observe that, the sequential layerwise GNN share similar high-level idea with the BCD method. We regard each layer and its associated auxiliary classifier as a module. Then, for each module, all its parameters can be treated as a coordinate block, we order the coordinate block according to which layer it corresponds to. Note that, in BCD, for each iteration, we choose one coordinate block and optimize the overall training objective with respect to the chosen block. So if we keep choosing the first coordinate block until it fully converged, then keep choosing the second coordinate block, etc., until the last coordinate block fully converge, then this optimization process is the same as the learning process of a sequentially trained layerwise GNN.

We also oberve that, the DGL-GCN can be analog to the synchronous parallel BCD and the LU-DGL-GCN can be regard as an analogy of asynchronous parallel BCD. Note that our DGL-GCN and LU-DGL-GCN can be implemented in a parallel fashion, and their key difference is whether all the layers share a consistency and up-to-date information. Therefore, if we make the same analogy

Table 3: Statistics of Benchmark Dataset

Dataset	**Cora**	**Citeseer**	**Pubmed**
Nodes	2708	3327	19717
Edges	5429	4732	44338
Classes	7	6	3
Feature	1433	3703	500

of learnable parameters and the coordinate blocks as in the above sequential version, then it would be easy to find the similarity between parallel BCD and our decoupled greedy learning methods. With such analogy, it would allow us to leverage existing theorems for BCD optimization to better understand and analyze the DGL-GCN and LU-DGL-GCN.

A.4. Supplementary for Experiments

Dataset Statistics We use three benchmark datasets: Cora, Citeseer, and Pubmed for the node classification task. The detailed statictics of theses datasets are shown in Table 3.
Hardware We run the experiments on Tesla V100 GPU (16GB).
Baselines Detailed introduction of our baselines can be found in Section 2. As introduced before, as NS methods are in general less competitive than the IS methods, therefore, we only compare against the IS strategies.
Parameter Settings For all the methods and datasets, we conduct training for 10 times and record their mean variance. For small datasets (Cora, Citeseer, Pubmed), we set the epoch number as 200, for OGBN-arxiv, we set the epoch number as 500. We choose the model with the best validation performance as convergence point. For the sampling-based methods: FastGCN and LADIES, we set the sample number as 64, increasing this number would improve the accuracy a little citep, but would increase the memory and time cost.

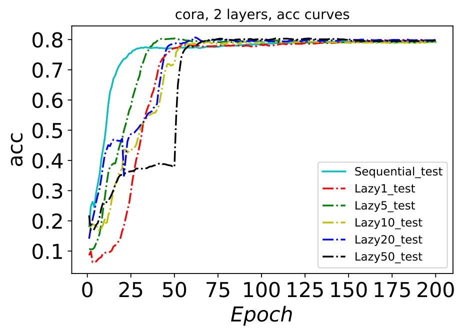
A.5. Ablation Studies

Importance of Lazy Update Scheme. We illustrate the advantage of the Lazy Update Scheme and show how the waiting time T_{lazy} will influence the performance. We use the cora and OGBN-arxiv [11] datasets as examples for small-size and large-size scenarios, we use a 2 layer and 7 layer model for these two scenarios respectively. We run the model for 200 epochs and compare the results. We summarize the results in Table A.5. We find that, with the lazy update scheme, we can greatly reduce the time cost, and a proper waiting time T_{lazy} would help to improve the accuracy a little citep since it can alleviate overfitting. In addition, when comparing the accuracy curves of different T_{layer} shown in Figure 2, we find that large T_{lazy} makes the training more stable.

Sequential Training v.s. Parallel Training. Finally, we briefly compare the sequential training and our parallel training of the greedy objective. Again, we use cora and OGBN-arxiv as examples and follow the above experiment settings. As shown in Figure 2, we find that in terms of accuracy, parallel training can quickly catch up with the sequential training, and is less likely to overfit.

Table 4: Comparison of LU-DGL-GCN with different T_{lazy}. Note that, $T_{lazy} = 1$ corresponds to the DGL-GNN model. Results show that with proper T_{lazy}, we can reduce time cost and improve accuracy while maintaining same memory cost.

Dataset	Performance	$T_{lazy} = 1$	$T_{lazy} = 5$	$T_{lazy} = 10$	$T_{lazy} = 20$	$T_{lazy} = 50$
Cora	Accuracy (%)	78.9 ± 0.8	79.1 ± 0.5	79.0 ± 0.1	79.2 ± 0.6	79.3 ± 1.0
	Total Time (s)	1.7 ± 0.1	1.6 ± 0.07	1.6 ± 0.2	1.4 ± 0.1	1.4 ± 0.1
OGBN-arxiv	Accuracy (%)	69.1 ± 0.3	69.5 ± 0.2	69.4 ± 0.2	69.3 ± 0.3	69.3 ± 0.3
	Total Time (s)	332.5 ± 14.1	85.8 ± 1.4	48.1 ± 0.5	33.1 ± 0.1	23.6 ± 0.1



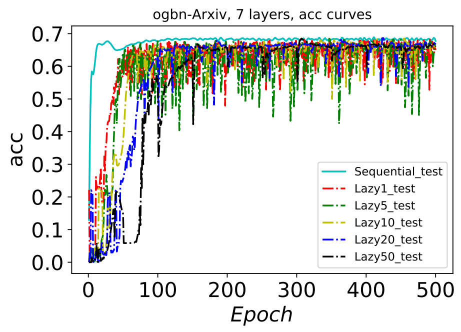


Figure 2: Comparison of sequential and parallel training. Results show that parallel can quickly catches up to sequential training within a few epochs.