Towards Sparsification of Graph Neural Networks

Hongwu Peng*§, Deniz Gurevin*§, Shaoyi Huang*, Tong Geng †, Weiwen Jiang ‡, Omer Khan*,
and Caiwen Ding*

*University of Connecticut, CT, USA. †University of Rochester, NY, USA. ‡George Mason University, VA, USA.
*{hongwu.peng, deniz.gurevin, shaoyi.huang, khan, caiwen.ding}@uconn.edu,
†tgeng@ur.rochester.edu, ‡wjiang8@gmu.edu

Abstract—As real-world graphs expand in size, larger GNN models with billions of parameters are deployed. High parameter count in such models makes training and inference on graphs expensive and challenging. To reduce the computational and memory costs of GNNs, optimization methods such as pruning the redundant nodes and edges in input graphs have been commonly adopted. However, *model compression*, which directly targets the sparsification of model layers, has been mostly limited to traditional Deep Neural Networks (DNNs) used for tasks such as image classification and object detection. In this paper, we utilize two state-of-the-art model compression methods (1) *train and prune* and (2) *sparse training* for the sparsification of weight layers in GNNs. We evaluate and compare the efficiency of both methods in terms of accuracy, training sparsity, and training FLOPs on real-world graphs. Our experimental results show that on the ia-email, wiki-talk, and stackoverflow datasets for link prediction, sparse training with much lower training FLOPs achieves a comparable accuracy with the train and prune method. On the brain dataset for node classification, sparse training uses a lower number FLOPs (less than 1/7 FLOPs of train and prune method) and preserves a much better accuracy performance under extreme model sparsity. Our model sparsification code is publicly available on GitHub[1].

Index Terms—graph, GNN, sparsification, model compression, sparse training, Surrogate Lagrangian Relaxation (SLR)

I. INTRODUCTION

Graph learning is an emerging branch in deep learning research that aims to reduce human effort in making tactical real-time decisions in applications, such as computer vision [1], traffic forecasting [2], autonomous systems [3], drug discovery [4], and social influence [5]. Graph learning architectures that combine the node embeddings of a graph into neural network models have been studied and proposed, such as GNNs [6], Graph Convolutional Networks (GCN) [7], GraphSAGE [8], and Graph Attention Networks (GAT) [9]. An example of GCN is given in Fig. 1, the input of GCN is the graph structure and embedding. Each layer of GCN will aggregate the node's adjacent embedding and conduct a linear transformation with non-linear activation. The GCN final output is the prediction result for tasks.

Increasing real-world graph sizes lead to the deployment of large GNN models with billions of parameters [10], [11]. For example, Pinterest's PinSAGE [12] and Alibaba's AliGraph [13] operate on graphs with billions of user/item embeddings (e.g., 492 million vertices, 6.8 billion edges for AliGraph). As model sizes continue to grow, GNN training has an outsize

Fig. 1: Graph convolution network structure.

computational cost. Training such large-scale GNNs require high-end servers with expensive GPUs. that are difficult to maintain. The computational challenges of training massive GNNs with billions of parameters on large-scale graphs is an important and emerging problem in the machine learning community. Sparsifying parameters in such large GNN models can reduce the computational and memory cost in the training and inference stages.

There are currently two main approaches for reducing GNN training and inference complexity by sparsification: *simplifying the input graph* and *sparsifying the model*. The first approach, which utilizes pruning or sampling of the nodes or edges in input graphs, has been explored extensively [14]–[17]. On the other hand, while model compression (or sparsification) is well-studied for traditional Deep Neural Networks (DNNs) [18]–[35], it is an under-explored area in the context of GNNs. To the best of our knowledge, recent work by Chen *et al.* [36] was the first to propose a framework for GNNs, called Unified GNN Sparsification (UGS), that pruned the input graph as well as the model weights using the well-studied *lottery ticket hypothesis* weight pruning method [19]. In this paper,

Fig. 2: Overview of two sparsification methods for neural networks: (1) *train and prune* and (2) *sparse training*.

---

we explore GNN sparsification using state-of-the-art model compression techniques for model weights, and evaluate their performance with both sparse *and* dense node embeddings.

In general, two classes of model sparsification methods (shown in Figure 2) are frequently employed to achieve high scalability, performance, and energy efficiency for neural networks: (1) *train and prune* (green line) and (2) *sparse training* (red line). The train and prune method [20], [21], [23], [25]–[27], [37], [38] first trains a dense model until it converges (step 1) and uses top-k weight pruning (step 2). Since the model accuracy usually drops after this top-k pruning stage, other optimization techniques, such as iterative pruning with fine-tuning and masked retraining on the sparsified model, have been employed (step 3) to recover model accuracy [39], [40]. Model parameters are dense in the first training step and sparse only in the retraining stage (one-third of the overall training time [23]). Although the final masked-retraining process is known to increase the overall run-time cost of the weight pruning pipeline [23], the initial dense training in this approach can lead to higher accuracy due to the availability of more model parameters.

The second model sparsification method is sparse training, which starts the sparsification process of the neural network layers directly from the beginning of training, using even fewer training iterations compared to dense training. Sparse training fixes the model sparsity at the beginning of the training and uses *drop and grow* policy to explore the sparse model architecture that yields the highest accuracy. It is the core of sparse training as a large number of weights are switched between zero and non-zero. Such frequent memory write and read operations heavily impact system performance. A "proper" drop and grow policy could significantly improve the temporal and spatial locality. Moreover, having a fixed sparsity throughout the training allows sparse training to reduce the computation and memory footprint of both training and inference stages, i.e., the weight parameters are sparse throughout the training process. The application and evaluation of the sparse training method have so far been limited only to classical DNNs for image classification tasks.

In this paper, we apply and compare (1) train and prune and (2) sparse training model sparsification methods in the context of graph learning. In our GNN sparsification framework, we sparsify the weights of representative Feed-forward Neural Network architecture and Graph Convolutional Networks (GCN) which propagate external node embeddings through its layers for link prediction and node classification tasks on graphs. We combine and evaluate weight pruning with both dense and sparse input embeddings. In our evaluation, we compare both of the train and prune and sparse training methods in terms of accuracy, achieved sparsity, and performance.

In summary, our contributions are as follows

- We formulate two sparsification frameworks for GNNs based on (1) train and prune and (2) sparse training.
- We evaluate and compare the trade-offs of the two evaluated sparsification methods in terms of accuracy, sparsity, and training FLOPs.

- To the best of our knowledge, to date, this is the first attempt that applies sparse training on graphs.
- For the brain, Cora, and CiteSeer dataset, we achieve a much higher accuracy using the sparse training method with much lower training FLOPs compared to the train and prune method.

II. BACKGROUND

A. Graph Learning

Graphs are ubiquitous data structures that describe complex systems with entities (nodes) and their interactions (edges). Large-scale and complex graph data makes machine learning tasks on graphs challenging due to having inefficient representations and requiring task-specific domain expertise.

Graph representation learning (GRL) aims to address these challenges by encoding graph structure into a low-dimensional embedding space. GRL translates the similarity between nodes in the original graph into closeness in the embedding space. This way, graph data is represented in a lower dimensional space that reflects the underlying graph structure efficiently.

One such GRL technique is based on performing random walks on a graph [41], [42]. Random walks capture the node properties by randomly visiting adjacent nodes. Random walks are then fed to word2vec's skip-gram model [43], which is a natural language processing (NLP) technique, to capture node embeddings. These node embeddings are fed into downstream graph learning tasks such as link prediction or node classification.

Graph learning is a well studied problem and there are many different techniques for it. GNNs [6], GCN [7], GraphSAGE [8], and Graph Attention Networks (GAT) [9] are some of the techniques for learning inductive node embeddings that combine external node features into neural network models.

Fig. 3: Sparsification of a 2-layer FNN. Given a graph, the GRL algorithm computes the embeddings of node u and feeds it to the input layer of the FNN. The FNN then propagates the node embeddings through its layers to output a prediction L_u. We specifically focus on weight matrices sparsification.

Different architectures can be employed for graph prediction tasks. In this paper, we consider a commonly deployed Feed-forward Neural Network (FNN) and GCN as representative networks for graph learning. We focus on compression, or *sparsification*, of the weight matrices of an FNN architecture that propagates node embeddings through its layers as shown in Figure 3. However, our proposed sparsification framework can be applied to other GNN architectures.

B. Sparsification Methods

The machine learning community has recently investigated many model compression methods for Deep Neural Networks (DNNs). These methods include weight pruning, quantization, sparsity regularization, and clustering [18]–[21], [23], [25], [26], [44]. The model compression techniques can reduce the learning noise and even increase the prediction accuracy [45]. The sparsified model may also increase the model robustness and has the potential to defend against adversarial attacks [46].

Generally, there are two major types of model sparsification methods. The first is to train the model until it converges and then prune, in which the model is pruned using top-k (threshold-based) weight pruning [20], [21]. This method has also been optimized by employing iterative pruning with fine-tuning for weight dropping and accuracy retraining [39], [40]. The second one, sparse training [47], gives up the hypothesis that the dense model could guide the sparsification process [19] and directly trains a model with fixed sparsity.

In this paper, we focus on these two methods of model sparsification: (1) the train and prune and (2) sparse training.

1) Train and Prune: Weight pruning is one of the most common model compression methods. Several prior works have observed that a portion of weights in neural networks are redundant. Weight pruning aims to remove the redundant components in the model and achieve similar accuracy with the original model [25], [26], [44], [48].

Earlier work in weight pruning is mostly based on heuristic approaches [49], [50]. Later, to overcome the heuristic nature, a systematic DNN weight pruning framework based on the Alternating Direction Methods of Multipliers (ADMM) technique [51] has been proposed in [21]. This work formulated the DNN weight pruning problem as a mathematical optimization problem and improved weight pruning by achieving $21\times$ compression on AlexNet and $71.2\times$ on LeNet-5.

However, ADMM does not guarantee the satisfaction of all constraints because of the non-convex objective function [52]. For this reason, ADMM-based weight pruning usually follows a final masked retraining process to further improve the model accuracy since the accuracy dramatically degrades after pruning. However, the retraining phase significantly increases the overall run-time cost of the pipeline.

To partially overcome this problem, a systematic weight pruning optimization approach based on Surrogate Lagrangian Relaxation (SLR) [53] has been proposed [23]. Within the SLR-based method, Lagrangian multipliers approach their optimal values faster as compared to those in the ADMM technique, and therefore, provide faster convergence during the training step and reduce final retraining cost. SLR weight pruning technique has so far been limited to classical DNNs for image classification and object detection tasks.

2) Sparse Training: The train and prune method aims to reduce the computation and memory footprint at inference stage [54] (e.g., for a typical three-stage (training-pruning-retraining) pruning process, the weight parameters are dense in the first two stages (two third of the training time [23])) and are sparse in the retraining stage. Sparse training reduces

the computation and memory footprint in both ***training*** and ***inference*** stages, i.e., the weight parameters are sparse (with fixed mask tensors) throughout the training process.

Static mask sparse training Single-Shot Network Pruning (SNIP) [55] was the first static mask training method to train sparse sub-networks at initialization. Later, GraSP [56] considered the weights less important if removing them would result in the least drop in the gradient norm. Saliency criteria [57] is proposed to help decide weight importance and employed to increase the accuracy of a sparse neural network. SynFlow [58] observed that the SNIP pruning method may lead to a layer collapse phenomenon and adopts gradient-based score to avoid layer collapse. The mask static training methods explores unstructured sparse training, which is restricted to be deployed on hardware platform and get acceleration. Taking advantage of hardware-aware design, Pixelated Butterfly [59] integrated the structured fixed sparsity butterfly format and low-rank decomposition to capture the global and local information. However, with the static masks, there is limited flexibility to preserve the important weights, so the accuracy is restricted.

Dynamic mask sparse training Dynamic sparse training is the process of training with a fixed number of nonzero elements in each neural network layer. Every ΔT iteration (ΔT is the drop-and-grow frequency), a proportion of weights with least magnitude values will be dropped or set to zero, and then new weights will be randomly or greedily added to the layer in the same amount as the previously removed. Different sparse training methods usually use the same dropping method (i.e., magnitude dropping), while the growth method vary. Sparse Evolutionary Training (SET) [60] randomly grew back the previously dropped weights. RigL [47] grew back the weights with top-k largest gradients. SNFS [61] utilized momentum to find the important weights and layers. ITOP [62], [63] found that the benefit of dynamic mask training come from its ability to cover all possible parameter positions.

III. SPARSIFICATION FRAMEWORKS FOR GNNS

In this section, we formulate the model sparsification for GNNs using (1) train and prune and (2) sparse training. Specifically, for the train and prune, we use the SLR-based weight pruning method. For sparse training, we follow a similar drop and grow method that was proposed in RigL [47] to explore the sparse model architecture during the sparse training process.

A. Weight Pruning Using SLR

Consider a GNN with N layers, where the weights at layer n are denoted by $\mathbf{W}_n$ for $n \in \{1, 2, ..., N\}$. In the ADMM and SLR training, the loss function can be defined as $f(\mathbf{W}_n) + \sum_{n=1}^{N} g_n(\mathbf{W}_n)$, for each layer n. The first term represents the nonlinear smooth loss function and the second term represents the non-differentiable "cardinality" penalty term [21] which ensures that the number of nonzero weights are less than or equal to the predefined number l_n within each layer n.

The objective of SLR training is to minimize the loss function. However, because the loss function is subject to

constraints on the cardinality of weights, it cannot be solved in its entirety. SLR technique decomposes the loss problem into 2 smaller subproblems by introducing duplicate variables $\mathbf{W}_n = \mathbf{Z}_n$ and rewriting the problem as $\min_{\mathbf{W}_n} f(\mathbf{W}_n) + \sum_{n=1}^{N} g_n(\mathbf{Z}_n)$. The *Augmented* Lagrangian function [21], [51] of this problem can be written as:

$$
\begin{aligned}
L_\rho(\mathbf{W}_n, \mathbf{Z}_n, \mathbf{\Lambda}_n) = & f(\mathbf{W}_n) + \sum_{n=1}^{N} g_n(\mathbf{Z}_n) \\
& + \sum_{n=1}^{N} \mathrm{tr}[\mathbf{\Lambda}_n^T(\mathbf{W}_n - \mathbf{Z}_n)] + \sum_{n=1}^{N} \frac{\rho}{2} \|\mathbf{W}_n - \mathbf{Z}_n\|_F^2,
\end{aligned}
\tag{1}
$$

where $\mathbf{\Lambda}_n$ are dual variables corresponding to constraints $\mathbf{W}_n = \mathbf{Z}_n$. The positive scalar ρ is the penalty coefficient, $\mathrm{tr}(\cdot)$ denotes the trace, and $\|\cdot\|_F^2$ denotes the Frobenius norm.

After the decomposition of the problem, the subproblems are solved iteratively in 2 steps:

1) Solve "Loss Function" subproblem for $\mathbf{W}_n$ by using stochastic gradient descent.
2) Solve "Cardinality" subproblem for $\mathbf{Z}_n$ through pruning by using projections onto discrete subspace.

At iteration k, for given values of multipliers $\mathbf{\Lambda}_n^k$, the first subproblem tries to minimize the Lagrangian function, while keeping $\mathbf{Z}_n$ at previously obtained values $\mathbf{Z}_n^{k-1}$ as

$$
\min_{\mathbf{W}_n} L_\rho(\mathbf{W}_n, \mathbf{Z}_n^{k-1}, \mathbf{\Lambda}_n).
\tag{2}
$$

the subproblem can be solved by stochastic gradient descent (SGD) since the loss function of the FNN and the regularizer are differentiable .

However, an additional "surrogate" optimality condition [53] for updating the multipliers is used as follows

$$
L_\rho(\mathbf{W}_n^k, \mathbf{Z}_n^{k-1}, \mathbf{\Lambda}_n^k) < L_\rho(\mathbf{W}_n^{k-1}, \mathbf{Z}_n^{k-1}, \mathbf{\Lambda}_n^k)
\tag{3}
$$

If this condition is satisfied, multipliers are updated as $\mathbf{\Lambda}_n'^{k+1} := \mathbf{\Lambda}_n^k + s'^k(\mathbf{W}_n^k - \mathbf{Z}_n^{k-1})$, with an additional stepsize parameter s_k [23]:

$$
s'^k = \alpha^k \frac{s^{k-1}\|\mathbf{W}^{k-1} - \mathbf{Z}^{k-1}\|}{\|\mathbf{W}^k - \mathbf{Z}^{k-1}\|}.
$$

The stepsize parameter can be defined as $\alpha^k = 1 - (1/(M \times k^{(1-\frac{1}{k^r})}))$ for $M > 1$, $0 < r < 1$.

The second subproblem for cardinality is solved with respect to $\mathbf{Z}_n$ while fixing other variables at values $\mathbf{W}_n^k$ as

$$
\min_{\mathbf{Z}_n} L_\rho(\mathbf{W}_n^k, \mathbf{Z}_n, \mathbf{\Lambda}_n'^{k+1}).
\tag{4}
$$

The globally optimal solution of this can be derived using the Euclidean projection of $\mathbf{W}_n^{k+1}$ and $\mathbf{U}_n^k$ onto the set $\mathbf{S}_n = \{\mathbf{W}_n \mid \mathrm{card}(\mathbf{W}_n) \leq l_n\}, n = 1, \ldots, N$. This achieved through pruning using the Euclidean projection of $\mathbf{W}_n^k$ and $\mathbf{\Lambda}_n'^{k+1}$ onto discrete subspace. Again, in order to ensure that the multipliers' updating directions are proper, another "surrogate" optimality condition needs to be satisfied:

$$
L_\rho(\mathbf{W}_n^k, \mathbf{Z}_n^k, \mathbf{\Lambda}_n'^{k+1}) < L_\rho(\mathbf{W}_n^k, \mathbf{Z}_n^{k-1}, \mathbf{\Lambda}_n'^{k+1})
\tag{5}
$$

If this condition is not satisfied, then both subproblems are solved again by using the latest available values for $\mathbf{W}_n$ and $\mathbf{Z}_n$. However, if the condition is satisfied, multipliers are updated as $\mathbf{\Lambda}_n^{k+1} := \mathbf{\Lambda}_n'^{k+1} + s^k(\mathbf{W}_n^k - \mathbf{Z}_n^k)$ where the stepsizes are calculated as follows

$$
s^k = \alpha^k \frac{s'^k\|\mathbf{W}^{k-1} - \mathbf{Z}^{k-1}\|}{\|\mathbf{W}^k - \mathbf{Z}^k\|},
$$

Overall, SLR allows efficient subproblem solution coordination using (1) stepsizes approaching zero and (2) the satisfaction of surrogate optimality conditions ensuring updates to multipliers are assigned along correct directions. Since it supports independent and systematic adjustment of the penalty coefficient and stepsizes, model parameters obtained by SLR are much closer to their optimal values compared to ADMM, which does not support the adjustment of stepsizes without leading to slower convergence. The SLR weight pruning method has been shown to have faster convergence compared to ADMM and therefore, reduce the overall training time [23].

After this SLR training process which adjusts the weights, the redundant weights whose values are closer to zero are pruned using Top-k pruning. This can be followed by a fine-tuning step, which masks the zero weights while training, for accuracy optimizations.

B. Sparse Training

Drop-and-grow Schedule. Dynamic sparse training is the process of training with fixed number of nonzero weights in each neural network layer. In Fig. 4, we use a toy example to illustrate the sparse training dataflow. For simplicity, we use the matrix with a size of 4×4 to represent a weight tensor in the neural network. The sparse training is comprised of 4 steps as follows. ❶ The weight tensor is random sparsified as W^0 at a given sparsity $S = 0.5$, which means 50% of weights will be deactivated (set as zeros) and others remain activate (non-zero). ❷ The sparsified tensor will be trained $\Delta T - 1$ iterations, where ΔT is the drop-and-grow frequency. During the $\Delta T - 1$ epochs, the non-zero elements in weight tensor are updated following the standard training process, while the zero elements will remain as zero. At the i-th iteration, the weight tensor is denoted as W^i, while the gradient is denoted as G^i. ❸ At the ΔT-th ietration, we first drop k weights that are closed to zero or set the weights that have the least k absolute magnitude as zeros ($k = 2$). Then, ❹ we grow the weights with the highest k absolute gradients back to nonzero (updating the weights with the highest k absolute gradients to nonzero in the following weights updating iteration). During the process, the number of activated weights are kept the same, i.e., the newly activated (non-zero) weights are the same amount as the previously deactivated (zero) weights. ❷❸❹ will be repeated till the end of the training.

Sparse Training Forward Propagation and Back Propagation. Consider a GNN with L layers. At training step t and activation a, the collection of weights parameters of the l-th layer is denoted by $\mathbf{W}^l$, respectively. The aim of sparse training is to keep the sparsity S of weight parameters as

Fig. 4: Iterative drop & grow based sparse training process.

$S \in [0,1]$ during the whole training process. We drop the α percent of weights that are closest to zero (i.e., smallest positive weights and the largest negative weights).

The forward propagation could be formulated as

$$a^l = \sigma(z^l) = \sigma(\beta^l \circledast a^{l-1} + b^l), \quad (6)$$

where z and b represent output and biases before activation. $\circledast$ is convolution operation. $\sigma(\cdot)$ is the activation function and a is the activation. At each step, we define β^l as a subset of weights from $\mathbf{W}^l$, and set the rest with zeros.

$$\beta^t = \begin{cases} \mathbf{W}^l & \text{if } i \in A^l, \\ 0 & \text{otherwise.} \end{cases}$$

where we define A^l as the indices of active parameters in a sparse subset. The initial selection of A^l element could be a random process [47] or restricted to the top-K proportion of weights by magnitude [64].

During backward propagation, we obtain the gradient of the active parameters update the weights [65].

$$\delta^l = \delta^{l+1} \circledast 180°\text{rotation}(\beta^{l+1}) \odot \sigma'(z^l), \quad (7)$$
$$G^l = a^{l-1} \circledast \delta^l \quad (8)$$

where $180°$rotation represents to rotate the weight tensor β^{l+1} 180 degrees. δ^l is the error in the l-th layer. G^l are gradients. $\circledast$ are dot-product. σ' represents the derivative of activation.

C. Training FLOPs Analysis

We first evaluate the case where the input is dense and the model is dense or sparsified. We assume the forward path for a dense model has f_D total number of float point operations (FLOPs) for a single epoch, and the sparsified model has f_S total number of FLOPs. f_S and f_D can be the connected throughput sparsity p: $f_S = f_D * (1 - p)$. Then, for each training epoch, the dense model consumes $3f_D$ FLOPs for forward and backward path [47], sparse model consumes $3f_S$ FLOPs for forward and backward path. For the SLR training process, assuming there are T_1 epochs for dense training, T_2 epochs for re-weight training, and T_3 epochs for sparsified training. The total training process of SLR training will have

$T_1 * f_D + T_2 * f_D + T_3 * f_S$ FLOPs. Assuming there are T_s epochs for the sparse training process, the training process will have $T_s * f_S$ FLOPs.

For some of the tasks, the input embedding can also be sparsified to reduce the total number of FLOPs further. The sparse embedding introduces a SpGEMM operation [66] into the DNN model if the weight matrix is also sparse. Assuming the sparsity of embedding is p_e and weight sparsity if p. For SpGEMM operation, the probability of index matching (both locations of embedding and weight matrix have element) is $(1 - p_e) \cdot (1 - p)$. The FLOPs of the first FC layer is scaled by $(1 - p_e) \cdot (1 - p)$ times compared to the dense counterpart. Assuming the embedding dimension is d_{emb}, then the probability of a location of output matrix of first layer has element is $p_{o1} = (1 - (1 - p_e) \cdot (1 - p))^{d_{emb}}$, which corresponds to the sparsity of SpGEMM output matrix. With the same derivation, the second layer FLOPs is scaled by $(1 - p_{o1}) \cdot (1 - p)$ times compared to the dense case. The output sparsity of second layer is given as $p_{o2} = (1 - (1 - p_e) \cdot (1 - p))^{d_{hid2}}$, and the d_{hid2} is the hidden dimension of second layer.

IV. METHODOLOGY

A. Experimental Setup

We conduct our DNN training on an Intel Xeon Gold 5218 machine at 2.30 GHz with Ubuntu 18.04 using an Nvidia Quadro RTX 6000 GPU with 24 GB GPU memory.

In our experiments, we firstly evaluate FNN architectures for 2 ML tasks on graphs: link prediction and node classification. We use real-world temporal graph datasets. For the link prediction task, we use wiki-talk [67]–[69], ia-email [70], [71] and stackoverflow [67], [68] datasets. For the node classification task, we use brain dataset [72], [73]. The details of these datasets can be seen in Table I. We also evaluate 2-layer GCN architectures with 16 hidden dimension for node classification task on graph. Cora [74], Pubmed [75], and CiteSeer [76] datasets are used for evaluating the GCN performance.

B. Graph Learning

For the graph learning tasks, we use an open-source C++ implementation[2] by Talati et al. [77]. We use this framework

[2]https://github.com/talnish/iiswc21_rwalk

Fig. 5: GNN accuracy v.s. sparsity on different datasets. –Blue line: SLR & dense embedding. –Yellow line: sparse training & dense embedding. –Green line: SLR & sparse embedding. –Red line: sparse training & sparse embedding.

TABLE I: Parameters of the datasets used for experiments.

Task	Dataset	#Nodes	#Edges
Link Prediction	ia-email [70], [71]	87,274	1,148,072
	wiki-talk [67]–[69]	1,140,149	7,833,140
	stackoverflow [67], [68]	6,024,271	63,497,050
Node Classification	brain [72], [73]	5,000	1,955,488

for link prediction and node classification tasks on temporal graphs. The purpose of the node classification task is to classify a previously unseen node in a correct label/category. The link prediction task aims to predict the presence/absence of a previously unseen edge formed between 2 nodes. The link prediction is performed as a classification task: the edges present in the graph are classified as positive edges, and the edges absent in the graph are classified as negative edges.

2-layer FNN with 128 hidden dimension is used for link prediction. For node classification, a 3-layer FNN with hidden dimensions as 256 and 128 is used. Node embeddings are inputs for link prediction and node classification tasks. We use dense and sparse embedding for training as input. For the sparse embedding, we select the top-$k\%$ values in the embedding matrix [78] and prune the rest. The k selection considers the trade-off between embedding sparsity and information loss. We set $k = 1.38, 3.28, 26.2, 46.9$ for wiki-talk, ia-email, stackoverflow and brain datasets, respectively.

For the 2-layer GCN on the node classification datasets, we set the hidden dimension as 16 to evaluate the performance. All datasets are divided as 60%, 20%, and 20% for training, validation, and testing, respectively.

C. Model Sparsification Setup

The SLR training follows (1) the standard dense training, (2) re-weighted training through SLR and (3) sparsified training. The re-weighted training utilizes SLR algorithms to find the proper sparse model architecture from the dense model. After the sparse model architecture is determined, the sparsified training is applied to further retain the model accuracy for a given sparse model.

For link prediction, we set $\rho = 0.01$, $s = 0.01$, $r = 0.1$, $M = 200$ for SLR training. For node classification, we set $\rho = 0.02$ $s = 0.02$, $r = 0.1$, $M = 200$ for SLR training. The initial learning rate is 0.05 for link prediction and 0.005 for node classification. We set the dense training to 20 epochs

with an exponentially decayed learning rate, and the re-weight training has 10 epochs for sparse model convergence. The final sparsified training has 10 epochs. The batch size is set as 1024.

For sparse training, we set the total number of epochs to 40 to match the total number of epochs with SLR training. We use the cosine annealing learning rate scheduler with the initial rate of 0.1. The batch size is set as 128 for link prediction and 512 for node classification. We set the death rate as 0.5 and drop-and-grow frequency as 1000 batch iterations.

V. EXPERIMENTAL RESULTS

A. Training FLOPs Evaluation

We set the model parameter sparsity as 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 0.906, 0.938, 0.969 to evaluate the performance of the SLR and sparse training method. We first evaluate the FLOPs of those two methods based on section III-C. By using dense training as the base, the normalized training FLOPs of SLR and sparse training can be found in Table II. When the model sparsity is 0.906, the SLR requires more than $8 \times$ more FLOPs than the sparse training.

We also evaluate the embedding sparsification influence on training FLOPs in Table II. For the link prediction tasks, the model is a 2-layer FNN, and the first layer occupies 94.1% of total FLOPs, and thus the embedding sparsification can significantly reduce the total training FLOPs. For ia-email and wiki-talk datasets with link prediction task, the embedding sparsification brings more than $10 \times$ training FLOPs reduction. However, for the brain dataset with node classification task, the model is 3-layer FNN, and the first layer contributes to 32.5% of the total FLOPs. In this case, the FLOPs reduction using embedding sparsification is not significant.

TABLE II: Normalized training FLOPs (sparsity = 0.906)

Training FLOPs	SLR		Sparse training	
Embedding	Dense	Sparse	Dense	Sparse
ia-email	0.773 $\times$	0.056 $\times$	0.094 $\times$	0.0067 $\times$
wiki-talk	0.773 $\times$	0.069 $\times$	0.094 $\times$	0.0084 $\times$
stackoverflow	0.773 $\times$	0.236 $\times$	0.094 $\times$	0.0286 $\times$
brain	0.773 $\times$	0.640 $\times$	0.094 $\times$	0.0776 $\times$

B. Accuracy Evaluation

We further evaluate the accuracy performance of the SLR training and sparse training for different model sparsity and

Fig. 6: GCN accuracy vs. sparsity on Cora [74], Pubmed [75], and CiteSeer [76] datasets. –Blue line: SLR training. –Yellow line: Sparse training. Sparse embeddings are used in the experiments.

embedding sparsity setups. The full comparison is given in Fig. 5 and Fig. 6. Most of the accuracy-sparsity curve has the Occam's Hill [79] property where the accuracy first increases with increasing sparsity and then decreases. The learned noise can be reduced with proper sparsity, which further enhances the model performance.

For both ia-email, wiki-talk, and stackoverflow datasets which are not sensitive to model parameter sparsity, sparse training has a comparable performance to the SLR method in terms of accuracy. However, for a more complex task such as node classification on the brain dataset, the sparse training will have much higher accuracy under extreme sparsity.

For ia-email dataset, the embedding sparsification with 98.62% decreases the model accuracy by 0.5% with the SLR method and sparse training evaluation. As shown in Table II, the embedding sparsification for ia-email dataset contributes to more than 10 × total FLOPs reduction for both training methods. For the wiki-talk dataset, the sparsified embedding has a positive impact on training accuracy and concurrently reduces training FLOPs more than 10 ×. For stackoverflow dataset, the sparse embedding reduces FLOPs more than 3 ×, with less than 1% accuracy drop on average. For brain dataset with node classification task, the embedding sparsification only brings approximately 1.1 × FLOPs reduction and causes 3% accuracy degradation on average. Thus, the embedding sparsification is not favorable for this specific task. For most of the evaluated datasets and tasks, the embeddings sparsification technique provides a significant training FLOPs reduction benefit and has little impact on accuracy.

The GCN sparsification performance on Cora [74], Pubmed [75], and CiteSeer [76] is similar to the FNN performance. In most cases, the sparse training-based sparsification method has a better accuracy under high sparsity. The 2-layer GCN only has 16 hidden dimensions, which makes the pruning unstable under high sparsity.

We further provide the accuracy-epoch evaluation for stackoverflow and brain datasets under 0.906 model sparsity with dense embedding. The comparison is shown in Fig. 7. There is a significant accuracy drop at epoch 20 for the SLR algorithm as the re-weighted training starts and the parameters are pruned. For the stackoverflow dataset, the SLR-based training converges at a higher accuracy than the sparse training. However, the stackoverflow dataset has a much faster convergence rate as the accuracy remains stable for the last 20 epochs. The sparse training has a better convergence rate and accuracy than the SLR method on the brain dataset.

VI. CONCLUSION

In this work, we explore the reduction of the computational and memory costs of GNNs. We compare two types of model sparsification methods: (1) the train and prune method and (2) the sparse training method. For the train and prune, we utilize the SLR optimization method to select the proper sparse model architecture. We randomly initialize the sparse model for the sparse training approach and explore the architecture during training. The experimental results show that the sparse training method preserves much lower total FLOPs than the train and prune method, and has comparable accuracy to the train and prune method as well as a much higher accuracy under extreme sparsity. Sparse training method requires less training time than the SLR method. In the future, we will explore other methods for DNN sparsification, such as LTH. And we'll also explore the system speed-up of various sparsification methods for training and inference tasks.

Fig. 7: Accuracy v.s. epoch. –: SLR method. –: Sparse training.

REFERENCES

[1] W. Shi and R. Rajkumar, "Point-gnn: Graph neural network for 3d object detection in a point cloud," in *CVPR*, 2020, pp. 1711–1719.
[2] W. Jiang and J. Luo, "Graph neural network for traffic forecasting: A survey," *Expert Systems with Applications*, p. 117921, 2022.
[3] X. Sun *et al.*, "Formal verification of neural network controlled autonomous systems," in *HSCC*, 2019, pp. 147–156.
[4] P. Bongini *et al.*, "Molecular generative graph neural networks for drug discovery," *Neurocomputing*, vol. 450, pp. 242–252, 2021.
[5] Z. Guo and H. Wang, "A deep graph neural network-based mechanism for social recommendations," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2776–2783, 2020.
[6] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, pp. 61–80, 2009.
[7] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *ArXiv*, vol. abs/1609.02907, 2017.

[8] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NIPS*, 2017.

[9] P. Velickovic *et al.*, "Graph attention networks," *ArXiv*, vol. abs/1710.10903, 2018.

[10] A. Sriram *et al.*, "Towards training billion parameter graph neural networks for atomic simulations," *ArXiv*, vol. abs/2203.09697, 2022.

[11] D. Manu *et al.*, "Co-exploration of graph neural network and network-on-chip design using automl," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021, pp. 175–180.

[12] R. Ying *et al.*, "Graph convolutional neural networks for web-scale recommender systems," *KDD '18*, 2018.

[13] R. Zhu *et al.*, "Aligraph: A comprehensive graph neural network platform," *KDD '19*, 2019.

[14] J. Chen *et al.*, "Fastgcn: Fast learning with graph convolutional networks via importance sampling," *ArXiv*, vol. abs/1801.10247, 2018.

[15] S. Yu, A. Mazaheri, and A. Jannesari, "Gnn-rl compression: Topology-aware network pruning using multi-stage graph embedding and reinforcement learning," *ArXiv*, vol. abs/2102.03214, 2021.

[16] Y. Cao, Z. Liu, C. Li, Z. Liu, J.-Z. Li, and T.-S. Chua, "Multi-channel graph neural network for entity alignment," in *ACL*, 2019.

[17] C. Chen *et al.*, "Dygnn: Algorithm and architecture support of dynamic pruning for graph neural networks," *DAC '21*, pp. 1201–1206, 2021.

[18] C. Ding *et al.*, "Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *MICRO-50*, 2017, pp. 395–408.

[19] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *ICLR*, 2018.

[20] A. Kusupati *et al.*, "Soft threshold weight reparameterization for learnable sparsity," in *ICML*. PMLR, 2020, pp. 5544–5555.

[21] T. Zhang *et al.*, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *ECCV*, 2018, pp. 184–199.

[22] S. Huang *et al.*, "An automatic and efficient bert pruning for edge ai systems," in *ISQED '22*. IEEE, 2022, pp. 1–6.

[23] D. Gurevin *et al.*, "Enabling retrain-free deep neural network pruning using surrogate lagrangian relaxation," in *IJCAI*, 2021.

[24] H. Peng *et al.*, "Optimizing fpga-based accelerator design for large-scale molecular similarity search (special session paper)," in *ICCAD '21*. IEEE, 2021, pp. 1–7.

[25] T. Chen *et al.*, "A unified lottery ticket hypothesis for graph neural networks," in *ICML*. PMLR, 2021, pp. 1695–1706.

[26] H. Tessier *et al.*, "Rethinking weight decay for efficient neural network pruning," *Journal of Imaging*, vol. 8, no. 3, p. 64, 2022.

[27] S. Huang *et al.*, "Sparse progressive distillation: Resolving overfitting under pretrain-and-finetune paradigm," in *ACL*, 2022, pp. 190–200.

[28] H. Peng *et al.*, "A length adaptive algorithm-hardware co-design of transformer on fpga through sparse attention and dynamic pipelining," in *DAC '22*. IEEE, 2022.

[29] P. Qi *et al.*, "Accelerating framework of transformer by hardware design and model compression co-optimization," in *ICCAD '21*. IEEE, 2021, pp. 1–9.

[30] S. Huang *et al.*, "Hmc-tran: A tensor-core inspired hierarchical model compression for transformer-based dnns on gpu," in *GLSVLSI*, 2021, pp. 169–174.

[31] H. Peng *et al.*, "Accelerating transformer-based deep learning models on fpgas using column balanced block pruning," in *ISQED '21*. IEEE, 2021, pp. 142–148.

[32] D. Xu *et al.*, "Rethinking network pruning–under the pre-train and fine-tune paradigm," in *NAACL '21*, 2021, pp. 2376–2382.

[33] T. Chen *et al.*, "Coarsening the granularity: Towards structurally sparse lottery tickets," *ICML*, 2022.

[34] J. Li *et al.*, "Towards acceleration of deep convolutional neural networks using stochastic computing," in *ASP-DAC*. IEEE, 2017, pp. 115–120.

[35] A. Ren *et al.*, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," *ASPLOS*, 2017.

[36] T. Chen, Y. Sui, X. Chen, A. Zhang, and Z. Wang, "A unified lottery ticket hypothesis for graph neural networks," in *ICML*, 2021.

[37] H. Peng *et al.*, "Binary complex neural network acceleration on fpga," in *ASAP '21*. IEEE, 2021, pp. 85–92.

[38] P. Qi *et al.*, "Accommodating transformer onto fpga: Coupling the balanced model compression and fpga-implementation optimization," in *GLSVLSI*, 2021, pp. 163–168.

[39] Z. Liu, M. Sun *et al.*, "Rethinking the value of network pruning," in *ICLR*, 2019.

[40] S. P. Singh and D. Alistarh, "Woodfisher: Efficient second-order approximation for neural network compression," *ICLR*, vol. 33, pp. 18 098–18 109, 2020.

[41] B. Perozzi *et al.*, "Deepwalk: online learning of social representations," *KDD '14*, 2014.

[42] G. H. Nguyen *et al.*, "Continuous-time dynamic network embeddings," *WWW '18*, 2018.

[43] T. Mikolov *et al.*, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[44] S. Yu, A. Mazaheri, and A. Jannesari, "Auto graph encoder-decoder for neural network pruning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 6362–6372.

[45] A. Krizhevsky, I. Sutskever, G. E. Hinton, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, "Advances in neural information processing systems," 2012.

[46] Y. Guo, C. Zhang, C. Zhang, and Y. Chen, "Sparse dnns with improved adversarial robustness," *ICLR*, vol. 31, 2018.

[47] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," in *ICML*. PMLR, 2020, pp. 2943–2952.

[48] G. Yuan *et al.*, "Improving dnn fault tolerance using weight pruning and differential crossbar mapping for reram-based edge ai," in *ISQED '21*. IEEE, 2021, pp. 135–141.

[49] Y. He *et al.*, "Channel pruning for accelerating very deep neural networks," *ICCV '17*, pp. 1398–1406, 2017.

[50] H. Mao *et al.*, "Exploring the regularity of sparse structure in convolutional neural networks," *ArXiv*, vol. abs/1705.08922, 2017.

[51] S. P. Boyd *et al.*, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, pp. 1–122, 2011.

[52] H. Li *et al.*, "Admm-based weight pruning for real-time deep learning acceleration on mobile devices," *GLSVLSI*, 2019.

[53] M. A. Bragin *et al.*, "Convergence of the surrogate lagrangian relaxation method," *J. Optim. Theory Appl*, vol. 164, no. 1, pp. 173–201, 2015.

[54] S. Chen *et al.*, "Et: re-thinking self-attention for transformer models on gpus," in *SC '21*, 2021, pp. 1–18.

[55] N. Lee *et al.*, "Snip: Single-shot network pruning based on connection sensitivity," in *ICLR*, 2019.

[56] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow," *ICLR*, 2020.

[57] M. C. Mozer *et al.*, "Skeletonization: A technique for trimming the fat from a network via relevance assessment," *ICLR*, vol. 1, 1988.

[58] H. Tanaka *et al.*, "Pruning neural networks without any data by iteratively conserving synaptic flow," *NeurIPS*, vol. 33, pp. 6377–6389, 2020.

[59] B. Chen, T. Dao, K. Liang, J. Yang, Z. Song, A. Rudra, and C. Re, "Pixelated butterfly: Simple and efficient sparse training for neural network models," *ICLR*, 2022.

[60] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science," *Nature communications*, vol. 9, no. 1, pp. 1–12, 2018.

[61] T. Dettmers and L. Zettlemoyer, "Sparse networks from scratch: Faster training without losing performance," *arXiv preprint arXiv:1907.04840*, 2019.

[62] S. Liu, L. Yin, D. C. Mocanu, and M. Pechenizkiy, "Do we actually need dense over-parameterization? in-time over-parameterization in sparse training," in *ICML*. PMLR, 2021, pp. 6989–7000.

[63] X. Ma *et al.*, "Effective model sparsification by scheduled grow-and-prune methods," in *ICLR*, 2022.

[64] S. Jayakumar, R. Pascanu, J. Rae, S. Osindero, and E. Elsen, "Top-kast: Top-k always sparse training," *ICLR*, vol. 33, pp. 20 744–20 754, 2020.

[65] G. Yuan *et al.*, "Mest: Accurate and fast memory-economic sparse training framework on the edge," *ICLR*, vol. 34, 2021.

[66] N. Srivastava *et al.*, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *MICRO '20*. IEEE, 2020, pp. 766–780.

[67] J. Leskovec and A. Krevl, "{SNAP Datasets}: {Stanford} large network dataset collection," 2014.

[68] A. Paranjape *et al.*, "Motifs in temporal networks," *WSDM '17*, 2017.

[69] S. Cunningham and D. Craig, "Creator governance in social media entertainment," *Social Media + Society*, vol. 5, 2019.

[70] R. A. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[71] J. Shetty and J. Adibi, "The enron email dataset database schema and brief statistical report," 2004.

[72] D. Xu *et al.*, "Spatio-temporal attentive rnn for node classification in temporal attributed graphs," in *IJCAI*, 2019.

[73] M. G. Preti *et al.*, "The dynamic functional connectome: State-of-the-art and perspectives," *NeuroImage*, vol. 160, pp. 41–54, 2017.

[74] A. K. McCallum *et al.*, "Automating the construction of internet portals with machine learning," *Information Retrieval*, vol. 3, no. 2, pp. 127–163, 2000.

[75] P. Sen *et al.*, "Collective classification in network data," *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.

[76] C. L. Giles *et al.*, "Citeseer: An automatic citation indexing system," in *Proceedings of the third ACM conference on Digital libraries*, 1998, pp. 89–98.

[77] N. Talati *et al.*, "A deep dive into understanding the random walk-based temporal graph learning," *IISWC '21*, pp. 87–100, 2021.

[78] N. Tonellotto and C. Macdonald, "Query embedding pruning for dense retrieval," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 3453–3457.

[79] C. Rasmussen and Z. Ghahramani, "Occam's razor," *ICLR*, vol. 13, 2000.