ABSTRACT

Graph Neural Networks (GNNs) are proven to be powerful models to generate node embedding for downstream applications. However, due to the high computation complexity of GNN inference, it is hard to deploy GNNs for large-scale or real-time applications. In this paper, we propose to accelerate GNN inference by pruning the dimensions in each layer with negligible accuracy loss. Our pruning framework uses a novel LASSO regression formulation for GNNs to identify feature dimensions (channels) that have high influence on the output activation. We identify two inference scenarios and design pruning schemes based on their computation and memory usage for each. To further reduce the inference complexity, we effectively store and reuse hidden features of visited nodes, which significantly reduces the number of supporting nodes needed to compute the target embedding. We evaluate the proposed method with the node classification problem on five popular datasets and a real-time spam detection application. We demonstrate that the pruned GNN models greatly reduce computation and memory usage with little accuracy loss. For full inference, the proposed method achieves an average of $3.27 \times$ speedup with only 0.002 drop in F1-Micro on GPU. For batched inference, the proposed method achieves an average of $6.67 \times$ speedup with only 0.003 drop in F1-Micro on CPU. To the best of our knowledge, we are the first to accelerate large scale real-time GNN inference through channel pruning.

PVLDB Reference Format:

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tedzhouhk/GCNP.

1 INTRODUCTION

Recently, Graph Neural Networks (GNNs) have attracted the attention of many AI researchers due to the high expressive power and generalizability of graphs in many applications. The node embedding generated from GNNs outperforms other graph representation learning methods when fed into downstream applications like node classification, edge prediction, and graph classification. Table 1 shows some popular applications of GNNs on various size graphs with different latency requirements. The knowledge graphs used in few-shot learning could only contain around one hundred of nodes and hundreds of edges, while the social network graphs could have billions of nodes and trillions of edges. Most of these GNN applications are latency sensitive at inference. For example, the applications related to Computer Vision need to perform streaming real-time inference on the data captured by the cameras. The applications related to fraud and spam detection need to identify malicious posts and transactions as fast as possible to avoid the property loss of the victim users. In addition to latency, some vision applications that utilize GNNs need to perform inference on edge devices with limited computing power and memory, such as self-driving cars with 3D-cameras and radars.

Compared with traditional graph analytics algorithms, GNNs have high computation cost as one node needs to gather and aggregate feature vectors from all the neighbors in its receptive field to compute a forward pass. To accelerate the training of GNNs, many works [4, 5, 13, 42] adopt stochastic node sampling techniques to reduce the number of supporting neighbors. GraphNorm [3] normalizes the node attributes to speedup the convergence. As a result, GNNs training scales well with graph size. It only takes seconds to minutes to train on a graph with millions of nodes. However, many GNN applications struggle at inference when deployed to...
production environment. Performing the full forward pass with all the neighbors at inference leads to high memory usage and latency. The node sampling techniques, when applied to inference, struggle to maintain high accuracy on every sample. In consequence, GNN applications either turn to traditional graph analytics algorithms with lower complexity, or rely on obsolete (not updated recently) embedding. For example, Youtube [12] turns to label propagation to detect abusive videos. Pinterest [39] has to use obsolete embedding generated with the MapReduce framework in an offline process. Taobao [19] runs the GNN based malicious account detection daily, instead of immediately after one transaction pops. Even with the compromise of offline inference, GNN inference is still expensive on large graphs. It is reported that a cluster with 378 computing nodes still needs one day to generate embedding for 3 billion nodes [39]. In addition, GraphBERT [45] shows that pre-trained GNN models could be directly (or with light fine-tuning) transferred to address new tasks, which makes accelerating GNN inference more important.

Although it has not caught much attention of researchers, accelerating GNN inference is as important as accelerating GNN training. Based on these GNN applications, we define two inference scenarios – full inference where the target nodes are all the nodes (or a large portion of nodes, i.e., the test set) in the graph, and batched inference where the target nodes are a few nodes. Full inference applies to GNN applications that operate on small to medium size graphs, or perform offline inference on large graphs. Batched inference applies to GNN applications that have strict requirements on latency, or need to be executed on edge devices such as embedded systems and FPGAs. Full inference performs forward propagation on all the nodes in the graph, while batched inference only propagates from the selected supporting nodes of the target nodes. For batched inference, the number of supporting nodes grows exponentially with the number of GNN layers, which is referred to as the “neighbor explosion” problem. In this work, we propose to accelerate GNN inference by reducing the input feature dimensions in each GNN layer and reusing the hidden features for visited nodes.

Our pruning framework works on most GNN architectures and can significantly improve their inference throughput with little or no loss in accuracy. The main contributions of this work are

- We develop a novel LASSO regression formulation to prune input channels for GNN layers, which outperforms random and greedy pruning methods.
- We design different pruning schemes for full inference and batched inference addressing their computation complexity and memory usage.
- We develop a novel technique to store and reuse the hidden features of visited nodes for batched inference, which mitigates the “neighbor explosion” problem.
- We evaluate the performance of the pruned models on five popular datasets and a real-time spam detection application. The pruned GNN models greatly reduce the complexity and memory usage with negligible accuracy loss. For full/batched inference, the pruned models reduce the computation to 0.19×/0.10× and memory requirements to 0.43×/0.18× with only 0.002/0.003 F1-Macro drop on average. The pruned models achieve an average of 3.27×/6.67× speedup for full/batched inference on GPU/CPU.

### 2 BACKGROUND

#### 2.1 Graph Neural Networks

For a graph $G(V, E)$ where each node $v \in V$ has node attributes $h(v) \in \mathbb{R}^d$, GNNs iteratively gather and aggregate information from neighbors to compute node embedding. Denote the matrix of all the output features $h^{(i)}(v)$ stacked horizontally in layer-$i$ by $h^{(i)}$. Let $A$ be the normalized adjacency matrix. In general, the output features $h^{(i)}$ of layer-$i$ is computed by

$$h^{(i)} = \sigma \left( \bigoplus_{k=K',}^K A^k h^{(i-1)} W_k^{(i)} \right)$$

where $\bigoplus$ denotes the horizontal concatenation operation, $W_k^{(i)}$ is the learnable weight matrix of order $k$ in layer-$i$. And $\sigma(\cdot)$ denotes the ReLU activation. We stack multiple layers and let the input of the first layer $h^{(0)} = h$ to compute the node embedding. For $K' = K = 1$, Equation 1 shows the forward propagation of vanilla Graph Convolutional Network [16]. For $K' = 0, K = 1$, Equation 1 is the GraphSAGE [13] architecture. For $K' = 0, K > 1$, Equation 1 is the MixHop [1] architecture. For other variants of GNNs [28, 33, 34], Equation 1 could be adapted by adding residue connections or alternating the normalized adjacency matrix.

#### 2.2 Case Study: GraphSAGE Inference

We perform a case study to analyze the complexity and memory usage for both inference scenarios on the widely used GraphSAGE architecture. We choose to analyze the GraphSAGE architecture as it achieves top tier accuracy with relatively high throughput (see Figure 1). For the GraphSAGE architecture, $K' = 0, K = 1$ and the adjacency matrix $A$ is normalized by $A = D^{-1/2}A$. For $D$ is the diagonal degree matrix.
2.2.1 Full Inference. To perform full inference that computes node embedding for all the nodes in the graph, we batch the node-wise aggregation and compute sparse-dense matrix multiplication $\tilde{A}k^{(i-1)}$. Denote the input and output feature dimensions of the weight matrices $W_k^{(i)}$ by $f_k^{\text{in}(i)}$ and $f_k^{\text{out}(i)}$ (all input feature dimensions are equal in each layer). Let $|V|$ be the number of nodes in the graph. Assume the average degree of the whole graph is $d$. The average complexity per node $C_{\text{full}}^{(i)}$ and total memory consumption $M_{\text{full}}^{(i)}$ of full inference are

$$C_{\text{full}}^{(i)} = O \left( d \min(f_1^{\text{in}(i)}, f_1^{\text{out}(i)}) + \sum_{k=0}^{1} f_k^{\text{in}(i)} f_k^{\text{out}(i)} \right)$$

$$M_{\text{full}}^{(i)} = |V| \left( f_0^{\text{in}(i)} + f_0^{\text{out}(i)} + f_1^{\text{out}(i)} \right) + \sum_{k=0}^{1} f_k^{\text{in}(i)} f_k^{\text{out}(i)}$$

As the output features of all nodes are computed in every layer, the computation and memory consumption distribute evenly in each layer. Each branch in one layer also contributes to a non-negligible portion of the computation and memory usage.

2.2.2 Batched Inference. For batched inference, the GraphSAGE architecture aggregates from $L$-hop neighbors. Denote the set of target nodes to infer by $V_t$. In layer-$i$, the average number of supporting nodes is $|V_t| \sum_{l=0}^{L-i+1} d_l$, which leads to the average complexity per node dominated by the complexity in the last layer

$$C_{\text{batched}} = \sum_{i=1}^{L} C_{\text{batched}}^{(i)} = \sum_{i=1}^{L} \sum_{l=0}^{L-i} d_l C_{\text{full}}^{(i)} = O(d^{L-1}C_{\text{full}}^{(1)})$$

Similarly, the memory consumption also peaks in the first layer with the most supporting neighbors.

2.3 Related Work

There are many existing works on channel pruning in Deep Neural Networks. The works [14, 31] prune the channels in the convolution layer by applying penalized regression on the input channels. ThiNet [20] prunes the channels based on statistics from the next layer. The work [38] forces some channels to freeze during the training and remove them at inference. Unlike performing inference on texts or images where each instance is independent with the others, inference of nodes depends on the graph structure and attributes of other supporting nodes. The computation pattern is also different for different inference scenarios. These two challenges make it hard to directly apply the existing channel pruning techniques on GNNs. Recently, several works [22, 32] have tried to accelerate training and inference of GNNs by removing the nonlinearity of internal layers and pre-computing the feature aggregation ($A^k h$). PPRGo [2] accelerates inference by performing less aggregation as in training. These methods require pre-processing on either the node attributes or the adjacency matrix, which do not apply to evolving graphs. TinyGNN [36] speeds up inference by training a shallow student GNN supervised by a teacher GNN. Recently, several works have tried to accelerate the full batch propagation in Equation 1 through matrix partitioning [41], node re-ordering [43], and runtime scheduling [26]. Others have developed hardware accelerators [37, 40] and in-memory processors [25]. These hardware-specific optimization techniques do not address the basic problem – high computation complexity of GNNs. On the other hand, although not aiming at rapid inference, some works [35, 46] propose to prune the edges to reduce the noise aggregated from neighbors. However, they are limited to knowledge graphs with specific inference queries.

In contrast, we propose a general method to reduce the inference complexity by directly pruning the input channels. Our method
works for all types of graphs and most GNN architectures. Combined with edge pruning methods and architecture simplification methods, our method has the potential to further speed up the inference. In addition, our pruning method does not incur extra sparse operations. The dimensions of the matrix operations are also lower, which makes it easier to design hardware accelerators and in-memory processors.

3 APPROACH

In GNNs, channels refer to columns in the hidden features matrix \( \mathbf{h}^{(i)} \). We propose to solve the channel pruning problem by applying LASSO regression [27] directly on the input channels. For a pre-trained GNN model, we prune the channels reversely from the output layer to the input layer. Figure 2 shows one pruned GNN layer with multiple branches. In this section, we first introduce the formulation and optimization of channel pruning in a single layer. Then, we discuss the pruning schemes for full inference and batched inference. For batched inference, we further propose a novel technique that stores the hidden features for visited nodes and aggregates directly from them during inference.

3.1 Single Branch Pruning

To prune the channels, we aim to generate the same output features in a branch before activation \( \mathbf{h}^{(i)} \) with fewer input dimensions. We focus on \( \mathbf{h}^{(i)} \) instead of \( \mathbf{h}^{(i)} \) to keep all operations linear. For branch \( k (1 \leq k \leq K) \) in layer-\( i \), let \( c_k \leftarrow \mathbf{c}_k^{(i)} \) be the number of channels in the original GNN. We formulate the channel pruning problem with budget \( \eta_k^{(i)} \) as the following optimization problem

\[
\begin{align*}
\arg \min_{\mathbf{\hat{h}}_k^{(i)}, \tilde{\mathbf{W}}_k^{(i)}} & \| \mathbf{Y}_k^{(i)} - \mathbf{\hat{h}}_k^{(i)} \mathbf{\hat{W}}_k^{(i)} \|_2^2 \\
\text{subject to} & \| \mathbf{\hat{h}}_k^{(i)} \|_0 \leq \eta_k^{(i)} c_k \\
& \| \mathbf{Y}_k^{(i)} - \mathbf{\hat{h}}_k^{(i)} \mathbf{\hat{W}}_k^{(i)} \|_2 \leq \eta_k^{(i)} c_k
\end{align*}
\]

where \( \mathbf{Y}_k^{(i)} \leftarrow \mathbf{A}^k \mathbf{h}^{(i-1)} \mathbf{W}_k^{(i)} \) is the target output features. \( \| \cdot \|_2 \) is the L2-norm and \( \| \cdot \|_0 \) is the L0-norm measuring the number of non-zero elements. \( \mathbf{\hat{h}}_k^{(i)} \in \mathbb{R}^{c_k \times n} \) is the coefficient vector acting as masks for each channel. \( \odot \) denotes element-wise multiplication on each row of the matrix. If \( \mathbf{\hat{h}}_k^{(i)} (j) = 0 \), then the \( j \)th channel of the input features \( \mathbf{h}^{(i-1)} \) can be removed. To solve the optimization problem, we first relax the L0-norm to L1-norm and add a penalty term with penalty factor \( \lambda \).

\[
\begin{align*}
\arg \min_{\mathbf{\hat{h}}_k^{(i)}, \tilde{\mathbf{W}}_k^{(i)}} & \| \mathbf{Y}_k^{(i)} - \mathbf{\hat{h}}_k^{(i)} \mathbf{\hat{W}}_k^{(i)} \|_2^2 + \lambda \| \mathbf{\hat{h}}_k^{(i)} \|_1 \\
& \| \mathbf{Y}_k^{(i)} - \mathbf{\hat{h}}_k^{(i)} \mathbf{\hat{W}}_k^{(i)} \|_2 \leq \eta_k^{(i)} c_k
\end{align*}
\]

We separate the optimization of \( \mathbf{\hat{h}}_k^{(i)} \) and \( \tilde{\mathbf{W}}_k^{(i)} \) into two sub-problems and optimize on the sub-problems iteratively to find the global minimum. Initially, \( \tilde{\mathbf{W}}_k^{(i)} = \mathbf{W}_k^{(i)} \) and \( \mathbf{\hat{h}}_k^{(i)} = 1 \). We optimize both sub-problems on the hidden features of the training nodes. We use the training graph as the normalized adjacency matrix during optimization to avoid information leak.

3.1.1 Optimization on \( \mathbf{\hat{h}}_k^{(i)} \). To optimize \( \mathbf{\hat{h}}_k^{(i)} \), \( \tilde{\mathbf{W}}_k^{(i)} \) is fixed. The problem becomes a classic LASSO regression problem with "large n, small \( p \). We solve the LASSO regression with Stochastic Gradient Descent (SGD). Due to the L1-norm term in the constraint, some mask values in the solution of \( \mathbf{\hat{h}}_k^{(i)} \) would shrink to zero, leading to the removal of the corresponding channels.

\[
\begin{align*}
\arg \min_{\mathbf{\hat{h}}_k^{(i)}} & \| \mathbf{Y}_k^{(i)} - \mathbf{Z}_k^{(i)} \otimes \mathbf{\hat{h}}_k^{(i)} \|_2 + \lambda \| \mathbf{\hat{h}}_k^{(i)} \|_1 \\
\text{where} & \mathbf{Z}_k^{(i)} = \mathbf{A}^k \mathbf{h}^{(i-1)} \tilde{\mathbf{W}}_k^{(i)}
\end{align*}
\]

3.1.2 Optimization on \( \tilde{\mathbf{W}}_k^{(i)} \). To optimize \( \tilde{\mathbf{W}}_k^{(i)} \), \( \mathbf{\hat{h}}_k^{(i)} \) is fixed. The problem becomes a quadratic programming problem.

\[
\begin{align*}
\arg \min_{\tilde{\mathbf{W}}_k^{(i)}} & \| \mathbf{Y}_k^{(i)} - \mathbf{X}_k^{(i)} \tilde{\mathbf{W}}_k^{(i)} \|_2^2 \\
\text{where} & \mathbf{X}_k^{(i)} = \mathbf{A}^k \mathbf{h}^{(i-1)} \otimes \mathbf{\hat{h}}_k^{(i)} \text{ and } \mathbf{\hat{W}}_k^{(i)} = (\mathbf{X}_k^{(i)T} \otimes \mathbf{X}_k^{(i)})^{-1} \mathbf{X}_k^{(i)T} \mathbf{Y}_k^{(i)}
\end{align*}
\]

3.2 Single Layer Pruning

To prune the input channels for one layer with multiple branches, we need to ensure each branch shares the same pruned channels in \( \mathbf{h}^{(i-1)} \) so that these channels could be removed in the output of the previous layer. We enforce that the same channels are pruned in each branch by applying a shared coefficient vector \( \mathbf{\hat{h}}_k^{(i)} \) and jointly optimizing on all the branches.

\[
\begin{align*}
\arg \min_{\mathbf{\hat{h}}_k^{(i)}, \tilde{\mathbf{W}}_k^{(i)}} & \sum_{k=1}^{K} \| \mathbf{Y}_k^{(i)} - \mathbf{\hat{h}}_k^{(i)} \odot \mathbf{\hat{W}}_k^{(i)} \|_2^2 + \lambda \| \mathbf{\hat{h}}_k^{(i)} \|_1 \\
\text{where} & \mathbf{Y}_k^{(i)} \leftarrow \mathbf{A}^k \mathbf{h}^{(i-1)} \tilde{\mathbf{W}}_k^{(i)} \text{ and } \mathbf{\hat{h}}_k^{(i)} \text{ is stacked horizontally.}
\end{align*}
\]

3.3 End-to-End GNN Pruning

We design the pruning schemes for different inference scenarios based on the computation complexity and memory consumption.
We need to visit the node attributes for an exponential amount of time in the case study in Section 2.2. The channel pruning in layer-i not only ignores some input channels in \( h^{(i-1)} \), but also leads to the reduction of output channels in the weight matrices \( W_k^{(i)} \) of the previous layer. Thus, when pruning the whole network, we prune reversely from the output layer to the input layer. Dense layers are treated as GNN layers with \( K' = K = 0 \).

### 3.3.1 Pruned Full Inference

For full inference, we simply prune each layer with a constant budget \( \eta \) except the input layer as the computation and memory distribute evenly in each layer. We do not remove any dimensions of the raw node attributes in layer-0. The complexity per node and memory usage of the pruned models range in \((\eta^2, \eta)\) and \((\eta, 1)\), compared with the original model. For other GNN architectures that follow similar forward propagation in Equation 1 like JK [34] and SIGN [22], our pruning method could be directly applied with constant budget.

### 3.3.2 Pruned Batched Inference

The major challenge in batched inference is the “neighbor explosion” problem where the number of supporting nodes grows exponentially as the network goes deeper. We need to visit the node attributes for an exponential amount of nodes to compute the embedding for one target node. Therefore, we focus on reducing the computation and memory usage in the first layer by reducing the channels in the first layer and the second layer. In the first layer, we focus on the branches that have more neighbors than others. For the GraphSAGE architecture with pruning budget \( \eta \), we prune the \( k = 1 \) branch in layer-1 and the whole layer-2 with budget \( \eta \), which reduces the dominant terms in the computation and memory usage by \( \eta \).

In addition to channel pruning, we store the hidden features \( h^{(i)} \) of visited nodes in the middle layers. Their neighbors, when aggregating from them, directly aggregate from the stored hidden features, instead of iteratively looking at farther neighbors. Figure 3 shows the supporting nodes in each layer with stored hidden features. Ideally, if we store the hidden features for all visited nodes, the batched inference would have exactly the same complexity as full inference (i.e., \( d = 1 \) in Equation 3). However, indexing and storing the hidden features incur extra data transfer which increases the latency. On evolving graphs, out-dated hidden features also affect accuracy. The portion of hidden features to store in each batch could be dynamically determined by the task-specific target latency and accuracy. Applications with high latency tolerance could potentially save more hidden features to increase throughput. For out-dated hidden features, we can set a threshold and discard them when the accuracy drop reaches the threshold. In practice, we find storing the hidden features for the root nodes at inference is a good balance point for the datasets we use. In addition, the root nodes usually have the most up-to-date hidden features in batched inference.

### 3.3.3 Detailed Optimization Procedure

In the experiment, we perform one iteration on each sub-problem instead of multiple iterations [14]. For the sub-problem of \( \tilde{W} \), instead of the least square solution, we also apply SGD as the size of \( X \) could be large. We partition the matrix \( A_k h^{(i-1)} \) and \( X_k^{(i)} \) row-wise to form mini-batches. Define one epoch as performing SGD on the whole matrix once. To optimize the whole problem, we first optimize several epochs on the sub-problem of \( \tilde{\beta} \). At the end of each epoch, we slightly increase the penalty factor \( \lambda \) until pruning budget is met or over-penalized (all values in \( \beta \) are decreasing). Note that as the mask values converge to zero, some mask values may be exactly zero while the others are close to zero. We clip the masks with small values to zero according to the pruning budget to make sure the corresponding channels are completely removed. Then, we optimize the sub-problem for \( \tilde{W}_k^{(i)} \) until converge. The final weights of the pruned layer are obtained by applying the mask \( \tilde{\beta}_k^{(i)} \) to the weights \( \tilde{W}_k^{(i)} \).

## 4 EXPERIMENTS

We evaluate the performance of the proposed method with the node classification problem on five popular datasets: 1. **Flickr** [42] classifying the types of user-uploaded images, 2. **Arxiv** [15] classifying subject areas of Arxiv CS papers, 3. **Reddit** [13] classifying communities of Reddit posts, 4. **Yelp** [42] classifying types of businesses on Yelp, 5. **Products** [15] classifying categories of products on Amazon. For batched inference, we also evaluate with a real world spam detection application on the YelpCHIII [21] dataset that identifies spam reviews on Yelp. We adopt supervised and inductive settings on all datasets.

For the models to prune, we use the 2-layer GraphSAGE [13] architecture (Equation 1 with \( K' = 0, K = 1 \)) with the common

Table 2: Dataset statistics. The Attr. column shows the dimension of the node attributes. (s) in Classes denotes multi-class single-label classification problem while (m) denotes multi-class multi-label classification problem. The Test% column shows the percentage of test nodes.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
<th>Attr.</th>
<th>Classes</th>
<th>Test%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flickr</td>
<td>89,250</td>
<td>899,756</td>
<td>500</td>
<td>7(s)</td>
<td>25%</td>
</tr>
<tr>
<td>Arxiv</td>
<td>169,343</td>
<td>1,166,243</td>
<td>128</td>
<td>40(s)</td>
<td>29%</td>
</tr>
<tr>
<td>Reddit</td>
<td>232,965</td>
<td>11,606,919</td>
<td>602</td>
<td>41(s)</td>
<td>24%</td>
</tr>
<tr>
<td>Yelp</td>
<td>716,847</td>
<td>6,977,410</td>
<td>300</td>
<td>100(m)</td>
<td>10%</td>
</tr>
<tr>
<td>Products</td>
<td>2,449,029</td>
<td>61,859,140</td>
<td>300</td>
<td>100(m)</td>
<td>10%</td>
</tr>
</tbody>
</table>
Table 3: Pruned full inference results on GPU. The * nodes in the plots denote the results of the reference models (no pruning).

<table>
<thead>
<tr>
<th>Budget</th>
<th>Flickr</th>
<th>Arxiv</th>
<th>Reddit</th>
<th>Yelp</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1-Micro</td>
<td>- 2x</td>
<td>4x</td>
<td>8x</td>
<td>-</td>
</tr>
<tr>
<td>#kMACs/node</td>
<td>545</td>
<td>211</td>
<td>94</td>
<td>48</td>
</tr>
<tr>
<td>Mem. (MB)</td>
<td>531</td>
<td>269</td>
<td>221</td>
<td>199</td>
</tr>
<tr>
<td>Thpt. (mN/s)</td>
<td>2.69</td>
<td>5.28</td>
<td>9.11</td>
<td>15.13</td>
</tr>
<tr>
<td>Thpt. Impr.</td>
<td>-</td>
<td>1.96x</td>
<td>3.39x</td>
<td>5.63x</td>
</tr>
</tbody>
</table>

hidden feature size 256, 512, 128, 512, 512 on the five node classification datasets, respectively. On the YelpCHI dataset, we use 128 as the hidden feature size. We use the standard single floating point precision for both the original models and the pruned models. To obtain trained models to prune, we adopt the sub-graph based training technique from GraphSAINT [42] with the random walk sampler. For each dataset, we prune with three global budgets \( \eta = 0.5, 0.25, 0.125 \) and obtain three pruned models (2x, 4x, 8x) in different sizes. We choose 1024 as the batch size and use the ADAM optimizer for SGD in the two sub-problems. After pruning, we re-train the pruned models until convergence.

To test the speedup of the pruned models, we measure the throughput and latency of full inference on the first four datasets with GPU, and batched inference on all five datasets with CPU and GPU. For batched inference, we form batches randomly from the nodes in the test set until all the nodes in the test set are covered. All accuracy (F1-Micro) results are for the test nodes only. The pruning framework is implemented using PyTorch and Python3. We run all experiments on a machine with 64-core ThreadRipper 2990WX CPU with 256GB of DDR4 RAM, and a single NVIDIA RTX A6000 GPU with 48GB of GDDR6 RAM. All the accuracy results are averages of three runs. For batched inference, we limit the number of hop-2 neighbors to be 32.

4.1 Performance of Single Layer Pruning
We compare the proposed pruning method (LASSO) with pruning the channels with small L1-norm in the corresponding weight matrix (Max Res.) and randomly pruning the channels (Random). Figure 4 shows the loss and F1-Micro curves under different numbers of pruned channels in both branches of layer-2 on the Reddit dataset. We apply layer-wise re-training for all three pruning methods. The proposed pruning method clearly outperforms other pruning methods, especially when the number of pruned channels is more than 30%.

4.2 Full Inference
Table 3 shows the results for full inference on GPU. The computation complexity is measured in thousands of Multiplication-and-Accumulation operations per node (#kMACs/node). The throughput is measured in thousand of target nodes computed per second (kN/s) or million of target nodes computed per second (mN/s). For memory usage, we adopt in-place point-wise operations without storing the intermediate values as we only need to compute forward propagation at inference. The latency is the GPU execution time of a complete forward propagation. The throughput and memory usage are calculated for all the nodes in the graphs. In the F1mic-Thpt row, the x and y axes are the throughput (in mN/s) and F1-micro. On the Flickr dataset, the pruned models achieve higher F1-Micro than the original models, possibly due to better convergence of smaller models. The reduction in computation and memory usage depends on the dimension of the input node attributes. We achieve close to \( \eta^2 \) reduction in computation complexity and \( \eta \) reduction in memory usage on the Arxiv and Yelp dataset with small input node attributes dimensions. We achieve an average of 3.27\(\times\) speedup on GPU with less than 0.006 drop in F1-Micro for all datasets with 4x pruned models. On the Flickr, Arxiv and Reddit datasets, the 8x pruned models still achieve similar accuracy as the original models. The pruned models for full inference reduce the latency on small datasets to meet the requirements for real-time applications and increase the throughput for large datasets. The pruned models also make it possible to run full batch inference of small graphs on edge devices with limited memory. We observe consistent GPU utilization of around 50% for models with different pruning budgets.
On the Flickr, Arxiv, Reddit, and Yelp dataset, our pruning method takes 2.35, 4.34, 6.35, and 32.15 seconds in training and 1.36, 6.38, 10.02, and 346.21 seconds in re-training. Due to the small number of parameters in the pruned models, the re-training of the pruned models takes less time than training the original models.

4.3 Batched Inference

Table 4 shows the results for batched inference on CPU. We calculate the memory usage by the amount of memory needed to compute the forward path of one batch. The attributes and stored hidden features of the supporting nodes in each batch are fed into CPU from DDR4 (peak bandwidth 68GB/s) and GPU from GDDR6 (peak bandwidth 768GB/s) memory. In the F1mic-Thpt. row, the x and y axes are the throughput (in kN/s) and F1-micro. We achieve a reduction in computation complexity and memory usage on all datasets for batched inference without stored hidden features. We store the hidden features of training and validation nodes, and the root nodes in each batch of inference. The storing of hidden features further reduces an average of 33% of supporting nodes in layer-1. We reduce the memory usage from 85-3086MB to 10-147MB, which makes it possible to perform inference on edge devices like mobiles. The memory usage also reflects an upper bound of the amount of input node attributes needed to perform one batch of forward propagation. On all five datasets, the pruned models with stored hidden features achieve less than 30ms (up to 17x improvement) latency on CPU with less than 0.012 F1-Micro drop. The pruned models with stored hidden feature meet the requirements of most real-time applications on CPU. On GPU, our 4x models achieve 4, 16, 4, 8ms latency without stored hidden features and 4, 6, 3, 6ms latency with stored hidden features on the Arxiv, Reddit, Yelp, and Products datasets, respectively. Compared with full inference, batched inference requires less memory and computation for a small number of target nodes. For latency-sensitive or large scale applications, batched inference provides a light-weight and low-latency solution. We observe 100% CPU utilization on all models, and 20% to 50% GPU utilization on GPU depending on the model size.

Figure 5.a shows the latency under different batch sizes on the Reddit dataset on CPU. The latency grows linearly with the batch size, which shows that our pruning method accommodates applications with different inference batch sizes. Figure 5.b shows the trade-off between storing hidden features and extra latency and drop in F1-Micro. Note that the extra latency is mostly caused by the storing of the hidden features, which can be done offline.

4.3.1 Spam Detection Application. To evaluate the performance of the proposed pruning technique on real-time applications, we over-sample the YelpCHI [21] dataset 400 times to create a graph with 27 million nodes which has similar scale as the Yelp website. The nodes in the graph represent reviews for restaurants and hotels in Chicago and are attached with timestamps. The task is to identify spam reviews from the posted reviews between October 2011 and November 2015.

![Figure 5: (a). Latency curves under different batch sizes on the Reddit dataset on CPU. (b). Maximum extra latency and accuracy drop curves with different percentages of stored hidden features.](image)
We use a similar two-layer (or equivalent of two-hop propagation) architecture on all baselines except a one-layer student network (reference), Jumping Knowledge Network (JK) [22], Higher-Order Graph Convolutional Architectures via Sparsified Neighborhood Mixing (Proceedings of Machine Learning Research), Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. 21–29.

REFERENCES

