Comparative Study of Graph Neural Networks Acceleration Techniques

Aditya Gupta
Indian Institute of Science
Bengaluru, India
adityapg@iisc.ac.in

Armaan Khetarpaul
Indian Institute of Science
Bengaluru, India
armaank@iisc.ac.in

Shankaradithyaa Venkateswaran
Indian Institute of Science
Bengaluru, India
shankaradith@iisc.ac.in

Umang Majumder
Indian Institute of Science
Bengaluru, India
umangm@iisc.ac.in

***Abstract*—Training Graph Neural Networks on large datasets is often a challenge due to limited hardware capacity. To tackle such issues, we apply Sampling and Sparsification methods on the OGBN-Proteins dataset to enable and accelerate GNN training. The results from these methods are then compared, along with their drawbacks and bottlenecks.**

***Index Terms*—Graph Neural Networks, GCN, GraphSAINT, GraphSAGE, DropEdge, NeuralSparse, Scalability, Graph Sampling, Sparse Learning, Node Classification, Large-Scale Graphs**

I. Introduction

Graphs are one of the most versatile data structures, with the capability to model any relationship, even if it is not always the most efficient way to do it in practice. For example, while pixels in an image can be represented as a network of connected nodes, representing the image as an array is more convenient for CNNs. For complex data, however, representation in the form of graphs becomes important. There are multiple real-life use cases of representing data in the form of graphs: representing relationships in social models, representing the structure of molecules and proteins, recreating relationships between objects in an image, etc.

A Graph Neural Network (GNN) is a class of Machine Learning models that predict properties on a dataset of graphs. The property to be predicted may be a graph-level, node-level, or edge-level classification or regression task. Two popular ways to represent graphs are adjacency matrices and adjacency lists. Since most of the graphs we will be dealing with are sparse, adjacency lists are the better storage method.

In this project, we aim to compare various techniques to accelerate the process of GNN training on large datasets. Some of these methods are commonly used, while others have been adapted to our use case and tested. The goal of this project is to identify the pros and cons of these methods to help someone choose the most optimal method of training for a given dataset.

II. Related Work

The most basic algorithm to perform Graph Level Predictions is the GCN [1], which generalizes convolution to graphs by aggregating and transforming information from a node's local neighborhood.

Graph networks can be extremely large and scale very fast in complexity as the number of nodes and edges increase, so optimization of their training becomes important. There are two major ways to perform GNN Acceleration are Sampling and sparsification

A. Sampling

GCNs are executed in a full batch manner. Model weights are updated once per epoch and slow down training convergence. Graph Sampling methods select nodes from a target node's neighborhood to acquire subgraphs for subsequent training. Node-wise sampling methods focus on each node and its neighbors in a training graph. They consider all neighbors a fixed number of hops away to perform sampling, eg. GraphSAGE [2], VR-GCN [3]. Layer-wise Sampling methods sample a fixed number of nodes in each layer based on a precomputed probability, eg. FastGCN [4], AS-GCN [5]. Subgraph-based sampling methods generate subgraphs for training in a two-step manner, by sampling nodes and constructing edges, eg. GraphSAINT [6]. Heterogeneous sampling methods are designed to accelerate training for heterogeneous graphs, eg. HetGNN [7]

B. Sparsification

Graph Sparsification typically removes edges from the graph either randomly or through a specific optimization goal. Sparsifying graphs before they are fed into a GNN makes computation efficient and reduces memory access for model training. Heuristic Sparsification methods like DropEdge [8] randomly remove edges in each training epoch, which aims to solve the over-smoothing issue in deep GNN training. Learnable Sparsification methods like NeuralSparse [9] use a DNN to learn a sparsification strategy, while other approaches like SGCN [10] casts sparsification as an optimization problem and resolve it via an alternating direction method of approach.

III. Methodology

A. Dataset

The OGBN-Proteins dataset is an undirected, weighted, and typed graph with 132,534 nodes and 39,561,252 edges, each represented by an 8-dimensional feature vector, where each dimension represents the approximate confidence of a single association type and takes values between 0 and 1. The task is to predict 112 independent binary targets on each node. The protein nodes are split into training/validation/test sets according to the species from which the proteins come. This enables the evaluation of the generalization performance of the model across different species.

To work with GNNs and test out different acceleration techniques, we require graphs with node features. Thus, we employ the scattering operation to convert edge features to node features. (Figure 1 and Figure 2)

$$V_i = \frac{\sum_{j \in N(i)} e_{i \to j}}{|N(i)|}$$

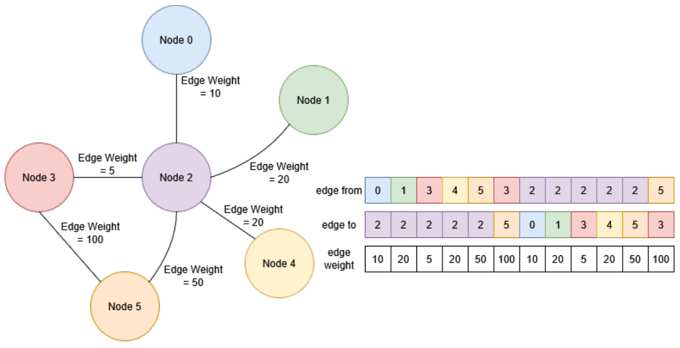

Fig. 1. Initial Graph with Edge weights

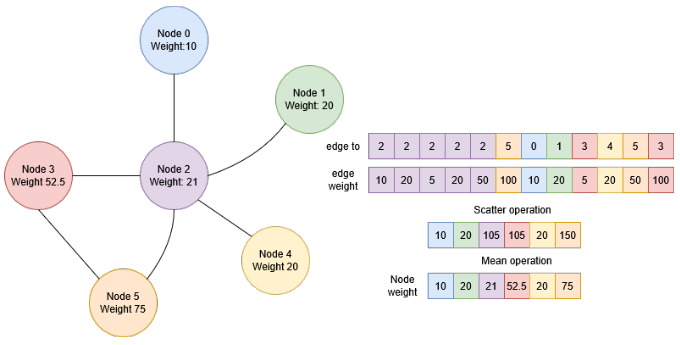

Fig. 2. Graph with node weights

B. Hardware Specifications

The sparsification and sampling methods were tested on an NVIDIA RTX 4090 GPU. We mainly focused on the total run time and processor utilization. The experiments were run multiple times to ensure consistency in the outputs.

C. Baseline Model and Implementation Details

All these methods were coded in Python using `torch-2.3.0` and `torch_geometric` libraries.

The baseline model for the experiments was a simple GNN model. This model contains two `GCNConv` layers followed by a fully connected MLP layer. The data was parsed using `RandomNodeLoader`.

To improve upon this baseline, we employed Sampling and Sparsification methods to get faster training times and increased model accuracy.

D. Sampling Methods

Instead of dividing the dataset into mini-batches randomly, Sampling Methods aim to intelligently choose the nodes for a mini-batch to help accelerate model convergence. We used two such methods on the given dataset.

The first Sampling method we use is called GraphSAGE [2], a combination of SAmple and aggreGatE, hence the name. It proposes a simplified way to sample nodes as follows: Given a node, GraphSAGE samples a fixed size set of neighbors defined as $N(v)$ which uniformly draws from the set $u \in V : (u, v) \in E$ (as a uniform distribution). These uniform distributions are not the same because each node can have a varying number of neighbors. GraphSAGE proposes to resample this fixed-size set (defined for each node of the graph) at each iteration k at which it is run. This is not to be confused with the epochs, as this iteration k is a parameter of the GraphSAGE algorithm itself. To further simplify the process, this sampling is done only for the nodes being processed in each mini-batch.

Another Sampling method we test is called GraphSAINT [6]. This method samples nodes by assigning probabilities to each one based on their features. It first builds an adjacency matrix A containing node features and then normalizes it across a row to give a matrix $\tilde{A}$. This is used to calculate the probability of choosing a node $P(u) \propto ||\tilde{A_{:,u}}||^2$. Nodes with more connections and stronger features are naturally assigned a high probability. This ensures important nodes participate in training repeatedly, while leaving room for other smaller nodes to show up in some iterations.

Although calculating these probabilities for sampling presents an additional overhead compared to random sampling, it should be offset by the fact that it can help the model converge in fewer epochs. This would reduce the overall time required for training and could potentially even give us a better model.

Both these sampling methods were implemented by using them to initiate the respective Dataloaders, an object that returns samples from the training set to train the model on.

E. Sparsification Methods

One more way to accelerate GNN training is to reduce the number of edges per node, resulting in faster neighbor aggregation. Once again, we used two such methods on the given dataset.

The first Sparsification method is called DropEdge [6], a regularization technique that helps mitigate the problem of over-smoothing and over-fitting. Over-smoothing causes node representations to converge to a stationary point as depth increases, while over-fitting reduces generalization, especially on small datasets. To deal with this, DropEdge randomly drops a fraction p of edges in the graph every epoch, giving a perturbed adjacency matrix $A_{\text{Edge Dropped}} = A - A'$. This perturbed matrix is then passed to the GNN. This reduces the spectral gap of A, thereby increasing the number of layers before node embeddings converge to a stationary point.

This was implemented before passing the input to the GNN. That is, at the start of every epoch, edges were dropped from each induced subgraph before passing it to the GNN for the forward pass.

Another Sparsification method we use is called Neural Sparse [7], which is an edge sampling-based sparsification method. The model uses Neural Networks to generate a probability distribution over candidate vertices, and use these probability distributions to sample a subgraph over the same nodes. This sparsified subgraph is then used for prediction/classification tasks. More formally,

$$\begin{aligned}
&\forall\, v \text{ in batch:}\\
N_v &= \text{Candidate neighbors of } v\\
E_v &= \text{Candidate edges}\\
z &= Softmax(MLP_\phi(\mathbb{V}(v), \mathbb{V}(N_v), \mathbb{E}(E_v)))\\
\pi_u &= \frac{e^{(\log z_u + \epsilon_u)/\tau}}{\sum_{u' \in N_v} e^{(\log z_{u'} + \epsilon_{u'})/\tau}}\\
&u_1, \dots, u_k \sim \pi\\
(v, u_1), \dots, (v, u_k) &\text{ are used to make the}\\
&\text{sparsified subgraph.}
\end{aligned}$$

This was implemented by using the above method to generate a subgraph at every iteration, for every batch, and then that subgraph was used to forward pass into the GNN, generate predictions, and training.

IV. Preliminary Results and Discussion

The aim of these methods was to improve and accelerate GNN training on hardware, without sacrificing performance. Our preliminary results show the strength of these methods in improving training quality and time. For each method, we present its comparison with the baseline, and the improvement observed, if any. The metric we use for this is the ROC Area Under Curve, since this is a multiclass detection problem.

A. GraphSage

TABLE I
GraphSAGE Comparison with Baseline

Model	Test ROC AUC	Best ROC AUC
Baseline	70.13	74.22
GraphSAGE	66.71	69.52

Here in table I, we can see that the ROC AUC for the GraphSAGE algorithm is less than the baseline. This is due to the naive node sampling method that GraphSAGE employs, which can cause drops in the final evaluation.

There is also another factor that comes into play, which is the aggregate function that is deployed. Here, instead of the regular aggregate function, which is the mean/average aggregate function, GraphSAGE proposes a `maxpool` aggregator function, which we have implemented. Here, the `maxpool` operator passes the node features of each neighbor through a Multi Layer Perceptron, after which an element-wise maximum is taken across all these outputs to get the final aggregated node feature vector.

This learnable aggregate operator is used to inductively run the algorithm to generate embeddings for unseen nodes. All this causes a slight drop in ROC AUC values as compared to the baseline GNN, but as a tradeoff, we get a learned model that can be used to generate reasonable embeddings for unseen nodes.

B. GraphSaint

TABLE II
Graph Saint Comparison with baseline

Mode	Test ROC AUC	Best ROC AUC	Epochs
Baseline	70.13	74.22	42
GraphSaint	72.69	78.43	21

Table II contrasts baseline training using `RandomNodeLoader` to the one using `GraphSaintNodeSampler`. The training time was roughly the same for the two, as the backend model was the same. The difference however is that GraphSaint node sampling produced significantly better results than the baseline training. Moreover, the training converged (early stopping reached with patience = 10) much earlier for GraphSaint than for the baseline. This verifies the effectiveness of this method in robust training.

C. DropEdge

TABLE III
DropEdge Comparison with baseline

Mode	Test ROC AUC	Best ROC AUC
Baseline	70.13	74.22
DropEdge	74.87	79.01

Table III compares the baseline training with DropEdge training in terms of test ROC AUC obtained on the test set, and the best ROC AUC achieved throughout the training. The model had roughly the same training time per epoch in either case. However, the performance obtained in the latter case was significantly higher than that obtained in the former case. This suggests that the training was more robust in the latter case.

D. Neural Sparse

TABLE IV
Neural Sparse Comparison with baseline

Mode	Test	Best	Epochs	Train Time/Epoch (s)
Baseline	70.13	74.22	42	26
Neural Sparse	72.69	76.46	25	51

Table IV shows the various metrics that compare baseline training of the model to training with neural sparsification. Specifically, we compare the training qualities on the following metrics: Test ROC AUC on the recommended test set, Best validation ROC AUC achieved throughout the training. Number of epochs taken for training to converge (early stopping with patience = 10 epochs), and average training time per epoch. The results clearly show the effectiveness of this method. The training for the baseline model converged in about the same time (slightly more) than that of training with sparsification. However, there was a slight boost to the performance in case of the latter.

V. Conclusion

From the above preliminary results, we can see that the Sampling and Sparsification methods used are giving us faster model convergence, and in some cases, even better results. This gives us a sanity check about the usefulness of these methods for GNN training acceleration.

All these experiments were conducted on the OGBM-proteins dataset [13], which is a fairly large dataset. We aim to try out these methods on some other datasets like PROTEINS [12] to check if we get any meaningful improvements. There are some overheads involved for each method that need to be studied properly before making a choice for the best method to use on a given dataset.

We can also conduct further experiments into the CPU and GPU utilization in each method to understand the bottlenecks involved. This would help us make better decisions about what hardware and technique is best suited for a given dataset.

Lastly, we can also measure the Energy Delay Product (EDP) for each of these methods on the given hardware to find out the most efficient techniques. While this may not be the best metric to define efficiency, it does give some comparison between the methods, which can be relevant when we want to train larger models.

References

[1] Kipf, Thomas N., and Max Welling. ”Semi-supervised classification with graph convolutional networks.” arXiv preprint arXiv:1609.02907 (2016).

[2] William L. Hamilton, Rex Ying, Jure Leskovec, ”Inductive Representation Learning on Large Graphs”, https://doi.org/10.48550/arXiv.1706.02216

[3] Rui Ye, Xin Li, Yujie Fang, Hongyu Zang, and Mingzhong Wang. 2019. ”A vectorized relational graph convolutional network for multi-relational network alignment.” In Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI’19). AAAI Press, 4135–4141.

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

[5] Li, Maosen, et al. ”Actional-structural graph convolutional networks for skeleton-based action recognition.” Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019.

[6] Zeng, Hanqing, et al. ”Graphsaint: Graph sampling based inductive learning method.” arXiv preprint arXiv:1907.04931 (2019).

[7] Shao, Zezhi, et al. ”Heterogeneous graph neural network with multi-view representation learning.” IEEE Transactions on Knowledge and Data Engineering 35.11 (2022): 11476-11488.

[8] Rong, Yu, et al. ”Dropedge: Towards deep graph convolutional networks on node classification.” arXiv preprint arXiv:1907.10903 (2019).

[9] Zheng, Cheng, et al. ”Robust graph representation learning via neural sparsification.” International Conference on Machine Learning. PMLR, 2020.

[10] Shi, Liushuai, et al. ”SGCN: Sparse graph convolution network for pedestrian trajectory prediction.” Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2021.

[11] Liu, Xin, et al. ”Survey on graph neural network acceleration: An algorithmic perspective.” arXiv preprint arXiv:2202.04822 (2022).

[12] Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alex J. Smola, Hans-Peter Kriegel, ”Protein function prediction via graph kernels”, Bioinformatics, Volume 21, Issue suppl1, June 2005, Pages i47–i56, https://doi.org/10.1093/bioinformatics/bti1007

[13] Hu, Weihua, et al. ”Open graph benchmark: Datasets for machine learning on graphs.” Advances in neural information processing systems 33 (2020): 22118-22133.