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, Energy Delay Product**

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 neighbourhood.

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 neighbourhood to acquire subgraphs for subsequent training. Node-wise sampling methods focus on each node and its neighbours in a training graph. They consider all neighbours 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 [13] 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)

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

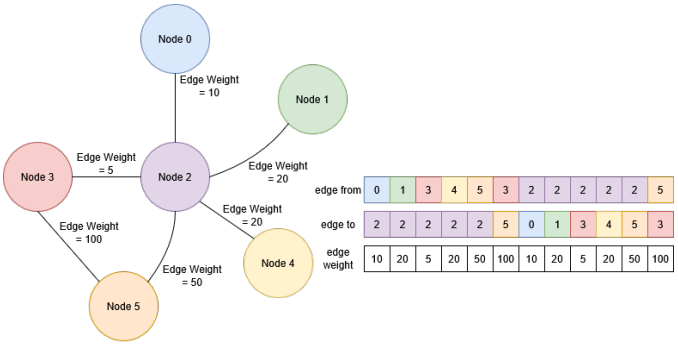

Fig. 1: Initial Graph with Edge weights

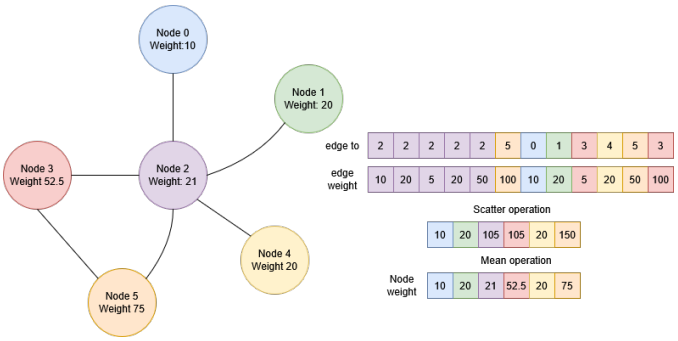

Fig. 2: Graph with node weights

B. Hardware Specifications

The sparsification and sampling methods were tested on a Ryzen 7 8845HS CPU and an NVIDIA RTX 4060 GPU. We focused on resource utilization, total run times and test accuracy while conducting our experiments. 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. We will be looking at two sampling and two sparsification methods which are commonly used to improve GNN training.

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 neighbours 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 neighbours. 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 neighbour aggregation. Once again, we used two such methods on the given dataset.

The first Sparsification method is called DropEdge [8], 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 [9], 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,

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

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. Results

We have discussed sampling and sparsification methods in the sections above. Since both these methods perform a different kind of operation on the graph, we can also use them together to accelerate our training. In this section, we will discuss various experiments performed on the OGBN-Proteins dataset to test out all combinations of these acceleration techniques. All experiments were conducted several time to ensure consistency in the results.

A. Experiments

We have two sampling methods, namely GraphSAGE and GraphSAINT, apart from the normal method of randomly sampling nodes from a uniform distribution. We also have two sparsification methods available, DropEdge and Neural Sparse, but we can proceed with training even without any sparsification. Using every possible combination of sampling and sparsification, we can perform 9 different experiments on the dataset. This should help us study the pros and cons of each method.

The training was performed by dividing the full dataset into train, validation and test splits. After every epoch of training on the train set, an inference is run on the validation set to check the resulting ROC AUC. If the validation accuracy keeps decreasing for a given number of epochs, then we stop the training to prevent overfitting. For our experiments, we set this threshold to be 10. Once the training is complete, the resulting model can be tested on the test split to obtain the final ROC AUC, which reflects the quality of the trained model.

Another thing to note about these methods is that both of them accelerate training by reducing the effective size of the graph to be operated on. This helps a lot is tackling large scale graphs with millions of nodes and edges. However, for smaller graphs, using both these methods simultaneously might result in the graph getting so small that training becomes ineffective. This issue was faced by us when we tried out these methods on the PROTEINS [12] dataset, due to which we ended up not using it for any further experiments. All the results listed in this section are from experiments performed solely on the OGBN-Proteins dataset.

B. Metrics

While performing these experiments, we focused on measuring important system related parameters like the CPU and GPU usage, RAM and vRAM occupied, and CPU and GPU power draw. These are essential metrics which can be used to evaluate the burden on the system while performing the training with a given technique. These measurements were made on a Windows laptop using the HWiNFO application, which exposes hardware level monitoring for a machine.

As we aim to accelerate model training, we will also be looking at the time it takes for the model to converge for each technique we test. This will be split into the time taken per epoch and the total number of epochs. Since we are performing a multi-class classification task here, the performance of the model will be evaluated using the ROC Area Under Curve (AUC) metric, which measures the probability that the model will rank a randomly chosen positive example higher than a randomly chosen negative example. This is a common metric used to measure the performance in classification tasks.

By measuring the power drawn by the system and the total time taken for training, we can calculate another important metric called the Energy Delay Product (EDP). EDP is a metric used to evaluate the efficiency of a system. More formally, it is defined as:

$$\text{EDP} = \text{Power} \times \text{Time}^2$$

A system with a lower EDP is considered more efficient for a given task. This allows us to recognize the combination of methods which helps us accelerate GNN training in the most efficient manner.

With these metrics defined, we can move on to analyze all the experiments mentioned above and study their pros and cons for the OGBN-Proteins dataset.

Sparsification Method	Sampling Method	CPU		GPU		RAM (MB)	vRAM (MB)
		Usage (%)	Power (W)	Usage (%)	Power (W)		
None	Uniformly Random	29.6	10.2	6.3	4.3	4212	1450
None	GraphSAGE	40.9	18.6	15.6	5.0	6481	433
None	GraphSAINT	45.0	20.1	23.7	9.8	7121	1055
DropEdge	Uniformly Random	40.3	18.0	42.3	20.0	3141	3388
DropEdge	GraphSAGE	44.0	19.5	41.8	19.3	6778	2560
DropEdge	GraphSAINT	47.2	21.2	23.9	10.3	7391	849
NeuralSparse	Uniformly Random	36.5	14.2	27.4	16.4	3074	1639
NeuralSparse	GraphSAGE	38.9	16.0	24.9	13.4	6802	2955
NeuralSparse	GraphSAINT	36.8	14.6	22.9	9.1	7713	1943

TABLE I: System parameters while training

C. System parameters

We first ran all 9 experiments for a few epochs each to record the relevant systems parameters, which are CPU and GPU usage, CPU and GPU power, and RAM and vRAM used. All the results can be seen in Table I. The values shown are the ones obtained after subtracting the relevant idle usage.

Let us start by looking at the CPU usage and CPU power draw. We can notice a pattern here that the power draw increases with an increase in utilization. This is natural since a higher number of computations happening in the CPU would require a proportionally higher power. The lowest power is used by the baseline case, where we do not use any sparsification and sample randomly. All other methods have a higher usage due to the overhead of setting up the methods. The GraphSAINT sampling method has a very high usage due to the intensive calculations performed by it across the entire adjacency matrix of the graph. Pairing it up with the DropEdge sparsification method further increases this due to the additional overhead of sparsification. However using GraphSAINT with NeuralSparse actually reduces its average usage. This happens since NeuralSparse takes much more time to execute, which reduces the overall contribution of GraphSAINT in the average usage.

Next we look at the GPU usage and GPU power draw for the methods. We notice that most of these numbers are much lower than the CPU usages, which might feel surprising for the training of neural networks which work primarily on GPUs. However we must realize that the neural network we use here is a simple two layer GCN, which does not have a high compute requirement. Even though we are working with a large enough dataset, we seem to be bottlenecked by the data transfer happening from the CPU. GPUs in general also have a higher power requirement, but due to the low workload here, they are working at a lower power limit. The average GPU usage here would depend on the amount of time the program uses the GPU and the number of data points it is operating on in parallel.

For the case where we do not use any sparsification, the GPU usage is the highest for GraphSAINT, which is most probably the result of the GPU being used for a higher proportion of the total time. This changes for other sparsification methods like DropEdge and NeuralSparse, where a lot more time is possibly spent on the CPU part for computing the sparsified graph. However both of these methods still result in an overall higher GPU usage for all samplers since they reduce the neighbours for every node, which allows us to go through the entire dataset faster. Since less time is spent for moving data relevant to each node, we can utilize the GPU more, resulting in a higher usage.

Finally we take a look at the CPU RAM and the GPU vRAM used by these methods. We notice that the RAM usage is much higher than vRAM for all the methods tested. This happens since most preprocessing required is done on the CPU, where a large chunk of the dataset might need to be loaded to perform operations. For example, GraphSAINT has the highest RAM usage for all sparsification methods since it operates on the entire adjacency matrix to calculate the relevant nodes. The average vRAM usage seems relatively random since it depends on all the previous factors such as the fraction of total time spent by the GPU, the size of the batch passed to the GPU, and the number of edges per node for the batch.

Overall we can see how different methods affect the system parameters for GNN training. All the methods discussed result in an increased CPU and GPU usage, which is generally a good sign since they are able to utilize the available resources properly. The RAM and vRAM usages are not affected much, since they are mostly a property of the dataset itself. Depending on the size of the dataset, the significance of the overheads would differ. In our case, since our dataset is big enough, the overheads do not make a large difference.

Now that we have measured how the system is affected by all these methods, we will proceed to run the full training loop to see if we get better overall results in terms of ROC AUC and EDP.

Sparsification Method	Sampling Method	Time per Epoch (s)	Number of Epochs	ROC AUC (%)	EDP (W min^2)
None	Uniformly Random	18.1	49	66.56	3605.21
None	GraphSAGE	14.7	29	68.27	1726.45
None	GraphSAINT	1.4	45	72.23	48.29
DropEdge	Uniformly Random	36.4	17	64.15	6413.80
DropEdge	GraphSAGE	28.2	38	74.76	19553.45
DropEdge	GraphSAINT	1.8	31	70.53	**39.01**
NeuralSparse	Uniformly Random	106.2	25	76.46	81455.40
NeuralSparse	GraphSAGE	84.1	19	77.34	29008.15
NeuralSparse	GraphSAINT	57.1	37	**79.17**	46494.81

TABLE II: Training results

D. Training results

Now that we have seen how the system parameters are affected by different methods, we can proceed with training the entire model to see how they perform on the test dataset. We measure the total time taken for training by checking the time taken per epoch and the total number of epochs required for convergence. The performance is then measured with ROC AUC. The CPU and GPU power measured above will also be used to calculate EDP. All the results can be seen in Table II.

Starting with the time taken per epoch, we can immediately see that GraphSAINT significantly reduces the training time for all sparsification methods. This happens because GraphSAINT samples a few nodes and removes all other nodes from the graph before passing it to the GPU for processing. Since each batch only acts on this small set of nodes, training becomes extremely fast. When paired with NeuralSparse, the training time is high due to NeuralSparse itself being a very heavy operation, but GraphSAINT still manages to reduce the training time the most compared to other sampling methods. Overall we can see that both sparsification methods increase the training time compared to the baseline since they need to go through every edge to decide which edges to drop. This overhead, which needs to be performed before the start of each epoch, results in a higher time per epoch.

Coming to the number of epochs required, the pattern is relatively random since it depends a lot on the dataset. These numbers can vary a lot depending on the random seed used for shuffling and the threshold set for stopping training. For our dataset, we can see that every method ends up requiring less epochs than the baseline. Intelligent sparsification and sampling has helped us reach convergence faster, which showcases the viability of these methods. Using sparsification methods resulted in a lower number of epochs required, since selecting the most relevant edges helps the model learn faster without getting biased by unimportant connections. Sampling also provides similar benefits by choosing relevant nodes.

Next we look at the ROC AUC obtained from all these methods on the test set. We can see that almost all the methods result in an improved ROC AUC over the baseline. In particular, the best improvement is obtained with NeuralSparse sparsification and GraphSAINT sampling. NeuralSparse in general seems to provide the best results, where all sampling methods paired with it result in a significant boost over the baseline. GraphSAINT also provides the best improvement among the sampling methods. Overall, we can see that all the methods we tried have proved useful in improving the performance.

Finally, we take a look at the EDP for all these experiments. Since the EDP is proportional to the square of the total time, methods with a higher training time will have an exponentially higher EDP. GraphSAINT, which resulted in a very short training time shows a significantly lower EDP compared to more intensive methods like NeuralSparse. The best EDP was obtained by combining DropEdge and GraphSAINT, which is much lower than the baseline. The power draw does not play a very significant role here since the training time becomes the deciding factor in most cases. We can see from Table I that GraphSAINT did indeed have the highest power draw, but it compensates for it by providing the fastest training time. However this improvement in EDP does come at the cost of a reduced ROC AUC.

To summarize, we see a tradeoff between the ROC AUC and EDP for GNN training. A higher ROC AUC often requires a higher EDP for training on the dataset tested. GraphSAINT provides the best reductions in EDP due to its amazingly fast training times, while NeuralSparse provides the best ROC AUC at the cost of slower training. DropEdge gave us a lower ROC AUC than the baseline for some cases, without any reduction in EDP. GraphSAGE also provides marginal improvements in the ROC AUC without heavily increasing the EDP. Every method has its own pros and cons. The choice of the best method depends on the dataset to be trained on.

V. Conclusion

In this project we looked at different methods to accelerate GNN training. We attempted to obtain improved performance on the OGBN-Proteins dataset. For this we used two sparsification methods called DropEdge and NeuralSparse, and two sampling methods called GraphSAGE and GraphSAINT.

From our experiments, we saw that all these methods were able to give us a significant boost in performance metrics like ROC AUC and EDP. In particular, NeuralSparse gave us the best improvement in ROC AUC while GraphSAINT gave the lowest EDP while still improving the ROC AUC from the baseline. In general, the best method to use would depend on the dataset in hand.

There are a lot more experiments that can be conducted with these methods by changing the dataset and the choice of hyperparameters used. For some datasets like PROTEINS, we even see a loss of performance due to the overheads involved with each method. To obtain the maximum value out of these techniques, we need appropriate datasets which should be large enough to make the overheads insignificant over the benefits obtained. Real world graphs with billions of edges would greatly benefit from these methods, given appropriate hardware to run them.

The python implementation of all these methods, which we used for conducting our experiments, can be found at https://github.com/adityapgupta/GNN_Acceleration

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.