Input Feature Pruning for Accelerating GNN Inference on Heterogeneous Platforms

Jason Yik
Harvard University
jyik@g.harvard.edu

Sanmukh R. Kuppannagari
Case Western Reserve University
sanmukh.kuppannagari@case.edu

Hanqing Zeng
Meta AI
zengh@meta.com

Viktor K. Prasanna
University of Southern California
prasanna@usc.edu

***Abstract*—Graph Neural Networks (GNNs) are an emerging class of machine learning models which utilize structured graph information and node features to reduce high-dimensional input data to low-dimensional embeddings, from which predictions can be made. Due to the compounding effect of aggregating neighbor information, GNN inferences require raw data from many times more nodes than are targeted for prediction. Thus, on heterogeneous compute platforms, inference latency can be largely subject to the inter-device communication cost of transferring input feature data to the GPU/accelerator before computation has even begun. In this paper, we analyze the trade-off effect of pruning input features from GNN models, reducing the volume of raw data that the model works with to lower communication latency at the expense of an expected decrease in the overall model accuracy. We develop greedy and regression-based algorithms to determine which features to retain for optimal prediction accuracy. We evaluate pruned model variants and find that they can reduce inference latency by up to 80% with an accuracy loss of less than 5% compared to non-pruned models. Furthermore, we show that the latency reductions from input feature pruning can be extended under different system variables such as batch size and floating point precision.**

***Index Terms*—data science algorithms, graph neural network, accuracy/performance trade-off, input feature pruning**

I. Introduction

Graph Neural Networks (GNNs) have recently attracted attention for their ability to learn and make predictions on graph-structured data. State-of-the-art models have been deployed on web-scale graphs in a diverse set of real-world applications such as recommendation [1] [2], traffic prediction [3], and fraud detection [4]. As is the case for traditional neural network models, GNNs operate on large tensors and have high computation workloads, so to achieve reasonable execution times GNNs will execute on heterogeneous systems consisting of a host CPU and an accelerator device. In addition to GPU-based acceleration, FPGA [5] and ASIC [6] accelerators for GNN execution have been proposed.

To express the structural information of its graph, GNNs rely on aggregating previous-layer features from the neighborhood of target nodes to generate features at the current layer. As the depth of a GNN increases, such aggregation causes an exponential increase in the number of sampled nodes, referred to as the ”neighbor explosion” effect [7]. This explosion effect is particularly impactful at the initial GNN layer, in which the raw input features are aggregated. For heterogeneous compute platforms, the communication latency of transferring this large volume of input features before computation begins can heavily impact the overall inference latency.

While several works have focused on optimizing the computation aspects of GNNs [8], optimizing for inter-device communication has not been well explored. A key challenge in communication optimization is that any reduction in the transferred data removes information from which the model can use for inference, leading to an adverse non-trivial impact on the accuracy of the model. Hence, detailed trade-off analysis of the accuracy and performance of model variants generated with reduced data volumes is required.

In this work, we study the effect of pruning input features from GNN nodes to accelerate inference by reducing inter-device communication. A high level analogy to a non-GNN neural network approach would be to reduce the resolution of an input video stream. For example, it is clear that down-sampling a 1080p, 60 Hz video to 360p, 24 Hz would allow for optimized memory transfer, potentially leading to important latency reductions in both real-time and offline inference settings. However, the corresponding idea of reducing the input data dimensionality for GNNs has not been studied. Since GNN inputs are embeddings of node features by nature, pruning techniques that have historically been applied to intermediate model channels [9] can also be applied to raw GNN input data.

We explore the trade-off in our pruned model variants' inference accuracy and make suggestions as to how they can be used in GNN inference service systems. Our contributions can be summarized as follows:

- We highlight how neighborhood aggregation leads to an inter-device communication bottleneck during GNN inference.
- We develop heuristic and regression-based algorithms for pruning input features from GNN nodes.
- We evaluate our pruning techniques on widely-used, large datasets and different GNN architectures. On different datasets, we produce variants which exhibit 11.88ms to 4.24ms (64%), 15.58ms to 2.54ms (83%), and 50.03ms to 8.85ms (82%) reductions in latency for accuracy losses of less than 5% in all cases.

The current work was supported by the U.S. National Science Foundation under award numbers PPoSS-2119816, SaTC-2104264, and OAC-1911229. Completed while authors were affiliated with the University of Southern California.

- We show that the latency reductions from input feature pruning can be extended under system variables such as batch size and floating point precision.
- We show how pruned model variants can be incorporated into GNN inference as-a-service cloud systems.

II. Background

A. Graph Neural Networks

Graph neural networks are a class of models which perform inference using neighbor message passing over graph-structured connections, turning a node's high-dimensional input features into a low-dimensional embedding that can be used in downstream tasks such as classification [10]. A GNN model is usually constructed from multiple layers, each of which transforms the features (or intermediate embeddings) from the previous layer. The forward pass of each layer can be generalized into two steps, *feature aggregation* and *feature transformation*.

1) Feature aggregation: For a given layer l, the first step is to aggregate each target node's neighbor features from the previous layer, X^{l-1} (X^0 denotes the input features, in the case of $l = 1$). Aggregation functions differ by GNN architecture, having distinct ways of combining the self features of a node with its neighbors' features. Generally, for a neighborhood and aggregation scheme modeled by A^l, feature aggregation takes the form

$$Z^l = (A^l X^{l-1})$$

In practice, since a node can have an arbitrarily large number of neighbors, neighbor sampling algorithms are typically used to limit the number of features aggregated for each node. For instance, the GraphSAGE [11] sampling algorithm defines per-layer budgets and randomly samples at most that many neighbors for each node that requires feature transformation in the layer. Thus, in the above equation X^{l-1} has the size of the sampled neighborhood and A^l defines which neighbors are sampled.

2) Feature transformation: In this step, at layer-l the aggregated features Z^l are transformed by the learned weight matrix W^l and a nonlinear function σ to generate the next layer's features. This step thus takes the form

$$X^l = \sigma(Z^l W^l)$$

After the aggregation and transformation across all layers in the GNN model, each node is left with a low-dimensional embedding from which a prediction can be made. In this paper, we study node classification, in which nodes' final embeddings are subject to a softmax layer that predicts one or multiple classifications for the node.

GNN architectures such as GCN [10], GraphSAGE [11], and GIN [12] differ in aggregation schemes and transformation methods. A GNN model's architecture is not dependent on its neighbor sampling algorithm [13], i.e. which neighbors it chooses to aggregate in each layer. For simplicity, in this paper we use the aforementioned GraphSAGE sampling algorithm, though our pruning methods can be applied regardless of which sampling algorithm is used.

B. Related Work

A recent survey on algorithms for GNN acceleration [8] categorizes current methods into two optimization categories - graph-level and model-level. The graph-level optimization methods focus primarily on breaking up graph data into more manageable pieces via sampling, sparsification, or partitioning. Such methods work only on the graph itself and do not account for node features. Model-level optimization methods aim for efficient computation by simplifying or compressing the GNN model itself. Efficient communication methods which optimize on inter-device data transfers have not been well-studied, especially for GNN inference tasks. In the following section we discuss related works relevant to efficient communication, our methods, and practical use cases for our work.

Quantization. Several works have proposed quantization methods for GNNs which reduce the precision of feature and model data to compress and accelerate model execution. Degree-Quant [14] uses node protection and percentile tracking at training time to generate high-accuracy quantized models, and BGN [15] goes to the limit of quantization by reducing model weights and embeddings to single bits. While the potential of reducing the precision of input features for use in quantized GNNs offers to solve the same memory transfer bottleneck as our input feature pruning, quantized GNN work has focused more on CPU-only inference service, especially for use on small edge CPUs. As noted in Degree-Quant, the massive parallel computation of GPUs limits the compute benefits of lower precision and offers little speedup over non-quantized model inference [14]. In our evaluation, we perform tests with reduced-precision models to show how they compare and combine with input feature pruning, see Section IV-D.

Neighborhood Sampling. GNN models such as Graph Attention Networks [16] aggregate features from all neighbors, assigning each with an attention coefficient dictating how impactful that neighbor's features should be. However, for very large, web-scale graphs, aggregating all neighbors for every target node quickly becomes intractable. Thus, bounding the sampled neighborhood to some constant size is required for scaling GNN inference. Recently, shaDowGNN [13] has proposed to decouple neighborhood sampling from GNN architecture, identifying a fixed number of important nodes from around the immediate neighborhood of a target and using only those nodes for aggregation in all layers. By reducing the number of sampled nodes, these methods also can accelerate GNN execution by reducing initial communication volume. Rather than studying graph sampling, in this work we examine the orthogonal dimension of node features, and we note that our method of input feature pruning can be combined with sampling algorithms for greater potential performance improvements.

Channel Pruning. Pruning neural networks generally refers to reducing model sizes by removing internal parameters [17], and our techniques for GNN input feature pruning are adapted from GNN channel pruning methods [9]. While reducing model parameters, channel pruning maintains all raw input feature data. Rather than focusing on decreasing feature communication time, it lowers computation complexity and compresses models. Generally, small and shallow GNN models already achieve state-of-the-art performance [18], so our work optimizes around the size of data rather than the size of the model.

Accuracy/Performance Trade-offs in ML as-a-Service. Recently, accuracy/performance trade-offs have been explored for machine learning service systems [19] [20] [21], hereafter referred to as dynamic service systems. Such systems take advantage of the fact that the same inference task can be served by many different model variants, differing in parameters such as model architecture, accelerator platform, and optimization degree. Having many different model variants on hand, dynamic systems will switch service automatically between high-accuracy/high-latency variants to lower-accuracy/lower-latency variants in reaction to changing user demands or workloads. Our technique of input feature pruning generates model variants that serve the same inference tasks at varying accuracy and latency, so GNN model variants that have been input feature-pruned can be used practically in such dynamic systems. We study this use case in Section V. So far, dynamic systems have studied applications in image processing [21] and speech recognition [19] [20] using traditional convolutional neural networks - to our knowledge we are the first to study the intersection of GNN models with dynamic systems.

III. Approach

We consider heterogeneous GNN inference platforms with a host CPU and an accelerator device (GPU, FPGA [5], ASIC [6]). The host receives inference requests and communicates input data from its memory to the accelerator memory over PCIe. In our analysis, we consider that the inference requests follow back-to-back and thus the model can reside within accelerator memory instead of being transferred on every request.

A. Communication Bottleneck

Feature aggregation in GNNs necessitates raw input data from many more nodes than are targeted for inference. In cases where graphs are too large to fit in accelerator memory, or the same accelerator serves multiple client tasks, transferring this raw data during each inference request can be a latency bottleneck.

In terms of communication volume, the worst-case scenario is that every node aggregates from every one of its neighbors, a sampling method used by some GNN architectures [16]. In this case, embeddings for target nodes at the last layer are generated by aggregating all intermediate features from the second-to-last layer, which are generated by aggregating all features from the third-to-last layer, ... until the input features are reached. Thus, there is an exponential growth in aggregation size backwards through the GNN layers. For inference on t target nodes in a graph with degree d, an L-layer GNN would need to aggregate $O(d^L t)$ input feature vectors.

In practice, aggregating features from every neighbor of each target node can become computationally intractable. Thus, sampling budgets may be imposed at each layer to limit the number of neighbor features aggregated. For a neighbor sampling method with sampling budget n_i for each layer i, the size of the neighborhood N_0 from which input feature vectors must be aggregated is

$$N_0 = c_0 \cdot t \cdot \prod_{i=0}^{L-1} n_i$$

Since the same node may be sampled by multiple different neighbors in the next layer, the number of unique nodes is some fraction c_0 of the product. Generally, the size of the aggregated neighborhood in the i-th layer of the GNN is

$$N_i = c_i \cdot t \cdot \prod_{i}^{L-1} n_i$$

where c_i is the unique node factor for that layer.

The compounding effect of sampling neighbors makes GNN inference intensive in communication, a property not seen in other neural network types. Because of the exponential aggregation trend, N_i is multiplicatively larger than N_{i+1}, thus the size of N_0 is dominant within the GNN execution. Also, due to the high-dimensional nature of raw data, the feature vectors f_0 for each node are larger than intermediate and final feature vectors. The input feature matrix X^0, with dimensions $(N_0 \times f_0)$, is therefore the largest matrix computed by the GNN and determines a significant portion of the computation workload. Since X^0 is the input to the GNN inference, the entire matrix must be transferred from host memory to accelerator memory, so it is also the dominant portion of the communication workload. Therefore, the complexity of computation and communication in GNN inference is comparable, unlike in the case for CNN architectures in which sliding windows may compute many times over the same input data. It is well known that communication performance and scaling lags far behind computation performance and scaling [22], so GNN execution can strongly suffer from a communication bottleneck. We validate this theory in our evaluations, see Table V.

B. Input Feature Pruning

Motivated by the idea of an inter-device communication bottleneck, we propose to increase the performance of GNN inference on heterogeneous platforms by pruning input features. If feature communication is a large factor in overall inference latency, reducing the volume of transferred data can significantly decrease the response time. However, pruning also creates lower-complexity GNN models - as more features are pruned there is less raw data to inform the model to generate its embeddings. Thus, by varying the degree of pruning, our technique can generate different GNN model variants that exhibit a range of accuracy and performance.

Pruning input features shrinks the input feature dimension by removing certain features, creating new input vectors for each node which include only important features. "Importance" of features refers to their relative usefulness in a GNN model for reaching the correct output embeddings: if a similar embedding can be reached in the absence of a certain feature then the feature is not important and can be pruned.

In terms of the input feature matrix X^0 with f_0 original features, we are identifying the important features as the subset S with elements $[1, f_0]$, and we generate pruned input feature matrix X^0_S with the columns of X^0 corresponding to the elements of S.

To generate different model variants we vary pruning amount by changing the size of X^0_S. For ease of choosing different pruning degrees, we order the features with an importance ranking and prune away lower-ranked features to construct the subset S. We study model variants generated using two heuristic-based methods and one regression-based method, adapted from methods which were first proposed to prune the hidden channels of GNN models for the sake of reducing computation and model size [9]. In this work, we extend them to GNN input feature pruning for the primary purpose of reducing inter-device communication. We analyze the different methods in Section IV-A.

C. Ranking Algorithms

1) Transforming Weights (TransWt): Using a trained GNN model, the features are ranked based on the L1-norm of the row corresponding to each feature in the first-layer weight matrix W^1 of the model. W^1 transforms f_0 features to f_1 features, so its dimensions are $(f_0 \times f_1)$. The ranking of feature i, R_i is

$$R_i = \sum_{j=0}^{f_1-1} W^1[i][j]$$

Intuitively, smaller values in the weight matrix decrease a feature's impact on transformation into the next layer's features, thus that feature can be deemed less important. In practice, this approach can fall short due to its inability to account for the relative average magnitudes of the features, for instance a feature with generally low magnitude would have greater transforming weights than a feature with generally high magnitude, for both to have similar impact on the next layer's features.

2) Feature Magnitude (FeatMag): Unlike the previous algorithm which requires a trained GNN model, this uses the training data itself to rank features. For input features of T training nodes, X^0_T (dimension $(T \times f_0))$), the ranking of feature i is

$$R_i = \frac{\sum_{j=0}^{T-1} |X^0_T[j][i]|}{T}$$

Features which have greater average magnitudes over the available training data are ranked higher than those with lower average magnitude. For features centered around 0, features are ranked by their overall variance. This technique attempts to circumvent the previous technique's issue with assigning more importance to low-magnitude features, though it is subject to failing to discern between unprocessed features that may have widely differing magnitude ranges. For example, some features may be very expressive near 0 in the range $(-1, 1)$ while others may be expressive in the range $(0, 100)$.

3) LASSO Regression (Lasso): We use lasso regression on the input features to identify their importance. Given a trained GNN model and training data, we learn a coefficients mask which is applied to input features before they are aggregated and transformed. The objective is to minimize the difference between first-layer intermediate features (X^1) generated from masked input features and first-layer intermediate features generated from original input features, using the aggregation scheme and weight matrix of the first layer of the GNN model for both. We add a penalty based on the combined magnitude of the coefficients, which forces the coefficients of certain features to decrease. A small coefficient for a given feature decreases its impact towards generating first-layer hidden features, making it a better candidate to be pruned. Thus, after the coefficients mask is learned, we rank features based off of the magnitude of their coefficients in the mask.

Formally, for a GNN model with first-layer aggregation neighborhood A^1 and weight matrix W^1, input features X^0_T from nodes in the training set, and penalty factor λ we optimize the coefficient mask β for the problem

$$\arg\min_{\beta} \left(\|A^1 X^0_T W^1 - A^1(\beta \odot X^0_T) W^1\|_2^2 + \lambda \|\beta\|_1 \right)$$

where $\odot$ denotes element-wise multiplication for each row of the matrix.

To begin, all coefficients in β are 1. In one epoch for the lasso regression, we calculate the above function as loss and backpropagate on β. To force features to decrease, at each epoch we increase the penalty factor λ by a set amount. After regression, features are ranked based on the corresponding magnitudes in β

$$R_i = \beta[i]$$

Since features are ranked based on the magnitude of their coefficient in β, we do not require any quota of features to be near zero and instead stop optimization after a number of epochs.

D. Generating Model Variants

Once the features are ranked, they can be used to create pruned input feature matrices X^0_S to train a model variant using only the pruned features. The ranking allows for fine control over how many features should be used to train pruned models since the subset S of saved features are taken from the top of the ranking, though in practice we find that a broad performance range is possible with only a few pruned model variants.

The process of ranking and training pruned model variants can operate with or without the supervision of the system client, asynchronous to inference service. A user may choose to perform the training and separately upload the model variants into a machine learning as-a-service system, or the user may upload a master model trained with all input features and the system can automatically perform ranking and pruned model variant training internally. At runtime, based on the model variant selected for service, the system will transfer only the pruned feature set to its accelerator to serve inference requests.

In our evaluation, we observe that the time for ranking features, even for Lasso, is negligible compared to pruned model variant training time. Since the decreased feature communication time is also relevant to forward propagation in training and pruned models have fewer weights, training pruned model variants is tractable. In practice, we observe a rough correlation between prune amount and training time reduction. The combined training time for 2×, 4×, and 8× pruned models is on the same order of the training time of the unpruned model, which would suggest linear scaling when applying input feature pruning to larger GNN datasets. This is consistent with our hypothesis of the general GNN communication bottleneck (Section III-A) and our empirical results of inference latency reduction (Section IV-B).

IV. Evaluation

TABLE I
DATASET STATISTICS. (S) AND (M) DENOTE SINGLE- AND MULTI-LABEL CLASSIFICATION.

Dataset	Nodes	Edges	Features	Classes
Flickr [23]	89,250	899,756	500	7(s)
Arxiv [24]	169,343	1,166,243	128	40(s)
Reddit [23]	232,965	11,606,919	602	41(s)
Yelp [23]	716,847	6,977,410	300	100(m)
Products [24]	2,449,029	61,859,140	100	47(s)

In this section, we evaluate the accuracy/latency trade-offs of the model variants generated by input feature pruning. Unless otherwise specified, all experiments are run on a machine equipped with an AMD Ryzen 3990X CPU and NVIDIA A6000 GPU, connected using PCIe 4.0.

We choose five datasets for GNN node classification from Pytorch Geometric [23] and Open Graph Benchmark [24], described in Table I. We run experiments under inductive settings [11], in which test nodes are unseen during training, which represents classification tasks where a GNN model is trained on a subset of all nodes and deployed for inference on the rest of the nodes or new nodes. All are single-classification tasks except Yelp, which assigns multiple ground-truth labels per node. Thus "Accuracy" refers to the F1-micro score for Yelp evaluation.

Our models are implemented using Python3 and Pytorch Geometric [23]. All models have two layers (one hidden feature transformation between input and output features). We train model variants with a batch size of 1024 and test the total accuracy and average latency values with 1024 inference batch size using the testing node set. We use GraphSAGE neighbor sampling with sampling budgets of 10 and 25 in the first and second layers, respectively.

In our workflow, we first train a model using all original input features, then rank features using the trained model as necessary for the ranking technique. To express a wide pruning range with concision we use an exponential pruning scheme, generating model variants with half, a quarter, and an eighth of original input features, referred to as 2×, 4×, 8× in our results. The un-pruned original model is referred to as 1×. Hyperparameters were tuned for the original model using the validation set, and all pruned variants are trained with the same hyperparameters as the original model they are generated from.

A. Comparison of Ranking Techniques

TABLE II
COMPARISON OF MODEL ACCURACY FOR DIFFERENT RANKING TECHNIQUES

Dataset	Prune Method	1×	2×	4×	8×
Flickr	TransWt	51.52	**51.27**	**50.81**	49.93
	FeatMag	51.52	50.79	50.28	**50.08**
	Lasso	51.52	51.15	50.69	49.69
Arxiv	TransWt	69.50	**67.85**	**66.09**	**60.22**
	FeatMag	69.50	66.93	63.61	56.71
	Lasso	69.50	67.03	63.43	58.10
Reddit	TransWt	94.72	**95.18**	**94.58**	**93.26**
	FeatMag	94.72	93.69	92.26	90.60
	Lasso	94.72	93.76	93.17	91.65
Yelp	TransWt	63.34	**61.77**	**59.44**	54.68
	FeatMag	63.34	60.99	58.45	**55.93**
	Lasso	63.34	61.17	58.81	55.36
Products	TransWt	74.90	71.72	61.89	47.75
	FeatMag	74.90	**74.24**	**71.65**	**66.25**
	Lasso	74.90	74.18	70.62	65.50

In Table II, we compare the accuracy of GraphSAGE model variants generated using the three different ranking algorithms outlined in Section III (Transforming Weights, Feature Magnitude, Lasso Regression). We do not report latency values here since the models differ only in which features are pruned and otherwise have the same dimensions. Highest accuracy model variants per pruning amount are bolded. The un-pruned 1× model is the same for each of the ranking techniques.

Accuracy differences are generally subtle between the different ranking techniques, which can suggest that structural information gained through neighbor aggregation may play a more significant role in inference accuracy than which specific node features are chosen. In general, we find that TransWt is slightly more accurate than Lasso, and both are slightly more accurate than FeatMag. For most of the datasets, TransWt outperforms the other ranking techniques across all pruning levels, sometimes by significant margins up to 2.5%. FeatMag models generally perform 1% less accurate than Lasso models.

The notable exception to the general trend is the Products dataset, for which TransWt models are far less accurate - around 3%, 10%, and 20% less than the FeatMag and Lasso models at 2×, 4×, and 8× pruned, respectively. Surprisingly, simply ranking features based off of their average magnitude across the training data produces better pruned models than the regression-based Lasso models, though the difference is marginal. In our tests, we find that the TransWt algorithm ranks low-magnitude features highest for the Products dataset, as expected given the technique's affinity for low-magnitude features as described in Section III-B. FeatMag, on the other hand, ranks high-magnitude features highest, and we also find that the features chosen by Lasso ranking also generally have higher magnitude.

Of the datasets tested, Arxiv [24], Reddit [11], and Yelp [7] all use averages of embeddings from individual words in the associated node's text as their node features, and Flickr [7] uses unprocessed bag-of-words counts as node features. Unlike the other datasets, Products [24] uses principal component analysis to reduce bag-of-words counts from product descriptions to a 100-dimensional embedding. Principal component analysis, by its nature, produces non-uniform embeddings: the first principal component exposes the highest variance across the data points (i.e. all nodes), the second component exposes the next highest variance when the first is disregarded, and so on. Thus, by using the FeatMag ranking technique on the Products dataset, we essentially are ranking the features according to the given PCA-based order, since higher-variance features have larger average magnitudes than the lower-variance features. For the Products dataset and other datasets with similarly generated features, the FeatMag ranking technique or even simply pruning features from the tail-ends of the original input feature vectors generates the strongest pruned model variants.

Though it did not achieve the highest accuracy in our results, Lasso generally outperforms FeatMag or is behind within a small error margin ($<1\%$). Due to its consistently high performance (as opposed to the catastrophic fall-off for TransWt on Products) and the theoretical robustness of regression, for the rest of our experiments we use the Lasso algorithm for all model variants.

B. Lasso Model Variant Performance

In Tables III and IV we show results for input feature pruning for all 5 datasets using the Lasso ranking technique for SAGE [11] and GIN [12] architectures, respectively. The GraphSAGE architecture learns two transforming weight matrices per layer, one which is applied to a node's self-features and the other which is applied to the mean of the node's sampled neighbor features. The GIN architecture scales self-features by a learned value, adds them to sampled neighbor features, and passes it through an MLP to generate the next layer's features. For the un-pruned original model and the three pruned model variants, we show accuracy, latency, serialized model size, and GPU memory usage. Latency measures the time beginning when feature data starts transferring to the GPU and ending once GPU computations are complete.

In all results, we notice a significant reduction in inference latency, positively correlated to the number of features pruned. As expected, datasets with a larger number of original input features see a larger latency reduction. Arxiv and Products, with the lowest number of original features 128 and 100, achieve around 50% latency reduction from 1× to 8× pruned models, while the other three datasets with a larger number of features (300, 500, 602) achieve around 80% latency reduction for the same. Notably, for a large number of features like in the Reddit dataset, latency improvements are nearly linear to the pruning amount, which validates the technique of extrapolating the number of features to prune given some target inference latency constraint.

For both the GIN and SAGE architectures, accuracy results between the same graph under the same pruning are similar, but between the datasets the accuracy results are variable. For some datasets, there is a small drop in accuracy, while others exhibit large accuracy degradations as more features are pruned. The relative drop in accuracy is not as predictable as in latency, but we observe that it is also correlated to the number of original input features. Flickr and Reddit, with 500 and 602

TABLE III
EXPERIMENTAL RESULTS FOR SAGE VARIANTS WITH LASSO RANKING.

Dataset	Prune	Accuracy	Latency (ms)	Size (KB)	GPU Mem (MB)
Flickr	1×	51.52	15.58	1021	174
	2×	51.15	8.82	523	90
	4×	50.69	3.92	272	46
	8×	49.69	2.54	145	25
Arxiv	1×	69.50	4.40	680	86
	2×	67.03	3.36	424	46
	4×	63.43	2.79	296	25
	8×	58.10	1.81	232	15
Reddit	1×	94.72	48.74	1293	541
	2×	93.76	26.63	693	274
	4×	93.17	14.63	390	140
	8×	91.65	9.00	240	71
Yelp	1×	63.34	15.06	1608	1100
	2×	61.17	8.91	1009	697
	4×	58.81	4.17	709	493
	8×	55.36	3.03	556	390
Products	1×	74.90	11.88	301	954
	2×	74.18	6.03	201	488
	4×	70.62	4.24	151	253
	8×	65.50	3.32	125	132

TABLE IV
EXPERIMENTAL RESULTS FOR GIN VARIANTS WITH LASSO RANKING.

Dataset	Prune	Accuracy	Latency (ms)	Size (KB)	GPU Mem (MB)
Flickr	1×	51.96	14.63	268	171
	2×	51.91	8.15	209	88
	4×	51.51	4.03	176	44
	8×	51.06	2.74	160	23
Arxiv	1×	69.81	4.43	184	84
	2×	69.84	2.85	169	44
	4×	67.98	2.14	161	23
	8×	64.32	1.75	157	13
Reddit	1×	92.04	50.03	432	538
	2×	91.05	26.70	359	272
	4×	89.10	15.16	320	138
	8×	87.64	8.85	301	71
Yelp	1×	63.38	14.93	730	1097
	2×	61.87	8.66	657	688
	4×	59.20	4.11	618	484
	8×	55.75	2.89	599	382
Products	1×	72.82	6.25	308	954
	2×	72.29	4.55	296	488
	4×	70.78	3.41	290	254
	8×	65.73	2.93	286	133

original input features, still achieve high accuracy even when pruned down 8× to 63 or 75 features. On the other hand, Arxiv and Products suffer more accuracy loss with only 16 and 12 input features at 8× prune. In all cases, the accuracy drop between 1× and 8× models is less than 10%.

Generally, we find that 2× models offer significant latency reduction at only slightly reduced accuracy compared to unpruned models. In the case of GIN models on Arxiv, the pruned model even outperforms the original model. In many other cases accuracy differences are less than 1%. Reducing the raw data available to the model, even by as much as half, can positively affect its inference accuracy by reducing noise. As input features are pruned away, increased sparsity can generate more accurate models since there is less learned noise and thus better prediction generalization.

Pruning affects not only the accuracy and latency of the models, but also their sizes and the runtime GPU memory usage. A pruned input feature dimension reduces the number of parameters needed in the first layer weight matrix W^1, and during computation the GPU does not need to save as much raw data or intermediate values. Since inference latency from dynamic systems can be subject to model loading time for cold starts and the system may need to co-host several models on the same device at once, having smaller and more resource-efficient model variants provide more opportunities for system optimization at runtime.

While pruned models are smaller due to reductions in first-layer transforming matrix sizes, we note that the reduction in latency is in general proportional to the pruning ratio, while reduction in model size is not. Thus, we attribute the latency reduction to decreased inter-device communication rather than decreased computation, supporting our hypothesis of a communication bottleneck in Section III.

C. Batch Size

TABLE V
RATIO OF COMMUNICATION TIME TO COMPUTATION TIME AND TOTAL LATENCY FOR DIFFERENT BATCH SIZES FOR SAGE YELP MODELS

	Prune Amount			
Batch Size	1×	2×	4×	8×
1024	**17.29** 15.06ms	**9.34** 8.91ms	**3.90** 4.17ms	**2.89** 3.03ms
512	**9.61** 8.88ms	**3.83** 4.05ms	**2.83** 2.83ms	**2.29** 1.98ms
256	**3.57** 3.96ms	**2.79** 2.57ms	**2.18** 1.87ms	**1.50** 1.37ms
128	**2.65** 2.46ms	**2.00** 1.76ms	**1.43** 1.39ms	**1.10** 1.17ms

To better understand the communication and computation components of total inference latency, in Table V we show the communication time to computation time ratio (in bold) and total latency of SAGE Yelp model variants, subject to different target batch sizes. Communication time measures the latency of inter-device feature transfers from the host to GPU, and computation time measures the latency of GPU computation. Accuracy is not reported since batch size during inference does not affect accuracy.

As the inference batch size decreases, fewer nodes are sampled, and as prune amount increases, fewer features are transferred for each sampled node. Both have the effect of reducing the overall communication volume, which is observed as a decreasing trend in communication/computation ratio both to the right and downwards in Table V. For the same batch size, as prune amount increases the computation load decreases due to the aforementioned reduction in first-layer transforming matrix size. From the decreasing communication/computation ratios, we clearly see that the communication reduction from pruning outpaces the computation reduction.

At smaller batch sizes, communication time begins to reach computation time and performance improvements begin to fall off. At batch size of 128, latency from 1× to 8× model variants decreases by about half for a total of 1.3ms, which is not as much as the 80%, 12ms reduction for the case of 1024 batch size. In our tests, batches smaller than 128 show negligible latency differences as kernel overheads dominate actual feature transfer. For real-time applications during which inference requests arrive individually, input feature pruning of a few nodes at a time will not impact total inference latency. However, for non-real-time web-scale classification tasks on very large graphs (as are modeled by our datasets), millions or even billions of nodes may be targeted for inference and thus larger inference batch sizes are preferable. In such cases, input feature pruning can be effectively used to reduce overall latency.

D. Half Precision

TABLE VI
LATENCY OF HALF PRECISION VS FULL PRECISION SAGE MODELS

Model		Accuracy	Latency (ms)
Reddit	Half, 1×	94.81	27.58
	Half, 2×	93.91	15.62
	Full, 2×	93.76	26.63
	Full, 4×	93.17	14.63
Yelp	Half, 1×	63.36	9.33
	Half, 2×	61.21	4.50
	Full, 2×	61.17	8.91
	Full, 4×	58.81	4.17
Products	Half, 1×	74.91	6.31
	Half, 2×	74.21	4.48
	Full, 2×	74.18	6.03
	Full, 4×	70.62	4.24

Though quantization and reduced precision optimization is not the focus on this paper, we briefly compare our pruning technique against reduced precision model inference. In theory, reducing from full (32-bit) to half (16-bit) floating point precision is equivalent to 2× feature pruning in terms of communication volume. We compare half precision model variants at 1× and 2× pruning to full precision model variants at 2× and 4× pruning in Table VI. Half precision variants are generated by casting pre-trained full precision models.

Overall communication volume, whether at full or half precision, is the determining factor for total inference latency. Half precision 1× models match the latency of full precision 2× models, and similarly half 2× models match full 4× models. All full precision models are slightly faster than the corresponding half precision counterparts, which may be due to the GPU being more optimized for 32-bit operations than 16-bit.

Interestingly, half precision models are generally slightly more accurate than full precision models at the same pruning amount. Post-training quantization in CNNs to 16-bit has been shown to have negligible effect on accuracy [25], and even training GNNs at 16-bit has small effects on accuracy [26]. In our case, the reduction of full precision noise appears as an overall benefit to inference accuracy. Since half precision models achieve significant latency reduction without any accuracy trade-off, in practical use it should be the default. We maintain full precision results for consistency with other research, and note that input feature pruning exhibits a similar accuracy/performance trade-off for both half and full precision models.

V. Input Feature Pruning in a Dynamic System

Model variants produced with input feature pruning exhibit a range of accuracy and performance that can be taken advantage of by a dynamic inference service system. At runtime, the dynamic system automatically switches its inference service based on changing user demands and workloads. For such systems, users interface via service-level objectives of accuracy and latency thresholds. The system has an internal performance profile of each of its model variants so it can deploy inference requests to suitable variants to meet the objectives. In this section, we show the application of our techniques to dynamic systems.

A. Latency Under Accuracy Tolerance

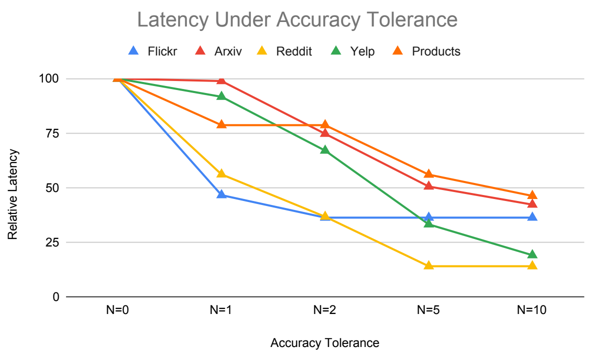

Fig. 1. Minimum latency under different accuracy tolerances for SAGE Lasso variants.

We package the accuracy/performance range of our generated models into an interface that the system or client could use by defining Latency under Accuracy Tolerances, similar to the client API used in Tolerance Tiers [20]. For this metric, we first identify the model variant with the highest possible accuracy, which serves as the baseline. Then, for an accuracy tolerance of N, we provide the relative latency (compared to the baseline) of the lowest-latency model variant which serves inference within N accuracy percent points of the baseline. Here, for highest accuracy inference a user would always choose accuracy tolerance of zero, and as the user relaxes their accuracy tolerance the relative latency will continue to decrease.

A latency under accuracy tolerance graph with multiple tolerances is shown in Fig. 1 for SAGE lasso model variants. For a finer granularity of accuracy tolerances, we report from a set of model variants trained under a linear pruning scheme (90%, 80%, ..., 10% features) instead of the exponential scheme (2×, 4×, 8×).

For a 2-point accuracy tolerance, input feature pruned model variants can achieve 25% to 65% reductions in latency compared to the highest accuracy models. Extending to a 5-point accuracy tolerance can further reduce latency by another 25% to 50%. In some cases, the same model serves at multiple different accuracy tolerances, such as Flickr at $N = 2$, 5, and 10, or Products at $N = 1$ and 2. If finer granularity is required for the accuracy tolerance interface, more model variants can be trained; however, the simple linear-pruned model variant set generates a performance range wide enough for dynamic inference already.

B. Model Variant Profiling

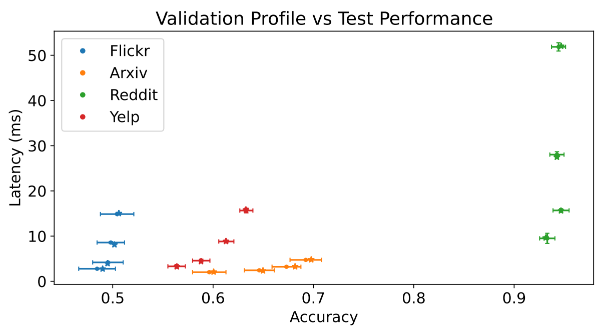

Fig. 2. Static performance modeling for SAGE variants under Lasso ranking, for all datasets except Products. Stars represent expected performance based on validation set, crosses represent test set performance with 1 std.

In a system operating under an inductive setting [11], nodes targeted for inference at runtime are unseen, and there does not exist a test set that generated model variants can be profiled on. As such, generated model variants must be profiled for accuracy and latency characteristics statically before runtime. We simulate static profiling by treating the validation set of our datasets as a test set, performing batched inference on it and logging accuracy and latency results. Then, we test this static profiling method by comparing the validation profile to the results from inference on the test set.

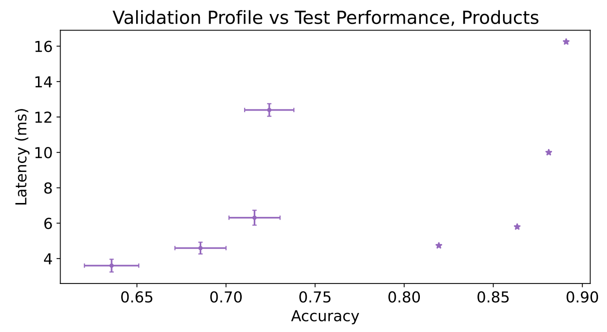

Fig. 3. Static performance modeling for SAGE Products with Lasso ranking.

Figure 2 shows accuracy and latency profiles from validation set inference (stars) versus results from test set inference (error bars ± 1 std.), for SAGE Lasso variants on all datasets except Products. Across the difference dataset sizes, accuracy ranges, and latency ranges, we see that validation profiles are remarkably close to test performance.

The exception to the other datasets is Products, for which validation profile vs test performance graph is shown in Figure 3. In this case, not only are validation profiles far more accurate than test performance, they also have higher latency expectations. Whereas the other datasets have generally similar structures between validation and testing graphs such that validation inference performance could predict testing inference performance well, the results for Products indicate that the validation subgraph is not a good indicator of the testing subgraph. Notably, its accuracy suggests that the validation set's structure and features are a close match to training set, and the lower latency of testing suggest that the test nodes have fewer neighbors to be aggregated. The Products dataset is constructed by ranking items based on sales, and the top 8% is used for training, the next 2% for validation, and the remaining 90% for testing [24]. Such construction is consistent with our evaluation and creates challenges in predicting model performance on unseen nodes.

While static, training-time performance modeling can be extensible to runtime inference in many cases, for certain datasets dynamic profiling techniques are necessary for accurate performance knowledge, and we identify this direction as important future work.

VI. Conclusion

In this paper we studied the effect of pruning input features for accelerated GNN inference on heterogeneous platforms. Our results showed that GNN inference suffers a significant communication bottleneck from inter-device transfer of initial data, which is greatly alleviated by reducing the size of input feature vectors. Over different pruning algorithms, GNN architectures, datasets, and system variables, we showed the effectiveness of input feature pruning to generate lower latency model variants without sacrificing heavily in inference accuracy.

References

[1] A. Pal, C. Eksombatchai, Y. Zhou, B. Zhao, C. Rosenberg, and J. Leskovec, "Pinnersage," *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2020. [Online]. Available: http://dx.doi.org/10.1145/3394486.3403280

[2] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," 2019.

[3] A. Derrow-Pinion, J. She, D. Wong, O. Lange, T. Hester, L. Perez, M. Nunkesser, S. Lee, X. Guo, B. Wiltshire, and et al., "Eta prediction with graph neural networks in google maps," *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, Oct 2021. [Online]. Available: http://dx.doi.org/10.1145/3459637.3481916

[4] Z. Liu, C. Chen, X. Yang, J. Zhou, X. Li, and L. Song, "Heterogeneous graph neural networks for malicious account detection," 2020.

[5] B. Zhang, H. Zeng, and V. Prasanna, "Accelerating large scale gcn inference on fpga," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 241–241.

[6] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," 2020.

[7] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," 2019. [Online]. Available: https://arxiv.org/abs/1907.04931

[8] X. Liu, M. Yan, L. Deng, G. Li, X. Ye, D. Fan, S. Pan, and Y. Xie, "Survey on graph neural network acceleration: An algorithmic perspective," 2022. [Online]. Available: https://arxiv.org/abs/2202.04822

[9] H. Zhou, A. Srivastava, H. Zeng, R. Kannan, and V. Prasanna, "Accelerating large scale real-time gnn inference using channel pruning," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, p. 1597–1605, May 2021. [Online]. Available: http://dx.doi.org/10.14778/3461535.3461547

[10] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

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

[12] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km

[13] H. Zeng, M. Zhang, Y. Xia, A. Srivastava, A. Malevich, R. Kannan, V. Prasanna, L. Jin, and R. Chen, "Decoupling the depth and scope of graph neural networks," in *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021. [Online]. Available: https://openreview.net/forum?id=d0MtHWY0NZ

[14] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," 2021.

[15] M. Bahri, G. Bahl, and S. Zafeiriou, "Binary graph neural networks," 2021.

[16] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[17] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 129–146.

[18] F. Wu, T. Zhang, A. H. de Souza Jr. au2, C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," 2019.

[19] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 397–411. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/romero

[20] M. Halpern, B. Boroujerdian, T. Mummert, E. Duesterwald, and V. Janapa Reddi, "One size does not fit all: Quantifying and exposing the accuracy-latency trade-off in machine learning cloud service apis via tolerance tiers," *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar 2019. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2019.00012

[21] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg, "Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud*

20). USENIX Association, Jul. 2020. [Online]. Available: https://www.usenix.org/conference/hotcloud20/presentation/zhang

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[23] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[24] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[25] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," 2019. [Online]. Available: https://arxiv.org/abs/1906.04721

[26] J. Brennan, S. Bonner, A. Atapour Abarghouei, P. Jackson, B. Obara, and A. McGough, "Not half bad: Exploring half-precision in graph convolutional neural networks," pp. 2725–2734, 12 2020.