Dynamic GNN Inference as-a-Service via Accuracy-Performance Trade-offs

No Author Given

No Institute Given

Abstract. Recently, dynamic Machine Learning as-a-Service (MLaaS) systems have been proposed. These systems automatically switch between available ML model variants when serving requests in order to react to fluctuating workloads or meet changing client performance demands. The available ML model variants need to exhibit accuracy and performance trade-offs over a wide range: higher complexity variants can fulfill requests with high accuracy but at high performance costs (e.g., latency), while lower complexity variants will serve at lower accuracy for decreased performance costs. The spectrum of model variant performance determines the effective range of the dynamic MLaaS system, thus, the techniques to generate model variants are critical. Currently, dynamic MLaaS systems support applications in image processing and speech recognition; in this paper, we extend such systems to graph machine learning applications. We develop an analytical performance model for Graph Neural Network (GNN) inference to identify the bottlenecks, and utilize input feature pruning as a technique to generate GNN model variants for dynamic MLaaS. We evaluate our technique using different GNN architectures on large datasets and produce pruned model variants which can reduce inference latency by up to 80% with an accuracy loss less than 5% compared to non-pruned models, showing that the technique effectively generates the wide trade-off range necessary for dynamic MLaaS. We are the first to develop techniques for trading off accuracy and performance to generate GNN model variants for dynamic MLaaS systems.

Keywords: Machine learning systems, graph neural network, accuracy performance trade-off, feature pruning

1 Introduction

Machine Learning (ML) sits at the forefront of modern-day computing technology. In recent years, many ML applications have scaled beyond what can be handled on most local setups - models can exceed 100 billion parameters [2] and inference task workloads may number in the trillions [11]. Due to such high complexities and workloads, ML inference is often offloaded to specialized ML-as-a-Service (MLaaS) systems like AWS SageMaker [1], which provide the infrastructure needed to perform at scale. Such systems present a decoupling of the ML task - clients have limited monitoring and control of their application

between the time that they make a request and the time that they receive a prediction.

Recently, several works have proposed dynamic MLaaS systems [16] [5] [25] which are robust to fluctuating workloads and changing demands by automatically adapting internal service. These systems take advantage of the fact that the same inference task can be served by many models, differing in parameters such as model architecture, accelerator platform, and optimization degree [15]. Generally, a more complex model variant can provide a better prediction but will have higher latency and resource costs, whereas a simpler variant may lose inference accuracy but will have lower response costs. Dynamic systems host many model variants which present a spectrum of accuracy and performance, and at runtime, they decide which variant is best suited to serve each request. So far, proposed dynamic MLaaS systems have studied applications in image processing and speech recognition [16] [5] [25].

Graph Neural Networks (GNN) are a subset of ML models which perform analysis on graph-structured data. GNN inference is used in a variety of real-world applications such as recommendation [13] [27], traffic prediction [3], and fraud detection [12]. While traditional MLaaS systems such as SageMaker [1] offer support for GNN applications, there have not yet been techniques to adapt them to dynamic systems. In this work, we aim to extend dynamic inference service to GNNs by developing a method for generating the model variants that expose the accuracy and performance range needed by the system.

Our contributions can be summarized as follows:

- We develop a theoretical model of GNN inference characteristics and show that communication is a significant bottleneck which can be optimized to improve performance.
- We develop different techniques for input feature pruning in order to decrease communication costs for inference on heterogeneous platforms.
- We evaluate our pruning techniques in the scope of model variant generation for dynamic MLaaS systems using 2 GNN architectures and 5 datasets. On different datasets, we produce variants which exhibit 14.87ms to 8.58ms (42%), 51.86ms to 9.50ms (81%), and 15.66ms to 4.57ms (71%) reductions in latency for accuracy losses of less than 1%, 2%, and 5%, respectively.
- We briefly comment on the potential of other recent GNN work for exposing an accuracy/performance trade-off for dynamic GNN inference service. We are the first to study the intersection between dynamic MLaaS systems and GNN applications.

2 Background

2.1 ML-as-a-Service Systems

ML has greatly scaled in both model complexity and in workload. State of the art models can contain billions of parameters [2] and single companies can serve trillions of predictions per day [11]. At such scales, it becomes impractical for

each user to host their own computing and hardware resources for ML tasks. Instead, dedicated MLaaS systems are used to allow them to offload their tasks to specialized infrastructure. MLaaS can be offered as a client-facing business, such as AWS SageMaker [1], but they are also deployed internally to one company, for instance Facebook serves its predictions through FBLearner [7].

While training ML models is an intensive development step, the large workload of using trained models for inference makes it the dominant component of ML service - 90% of ML infrastructure on AWS is dedicated to inference while the other 10% is used for training [9]. Recently, the inference services of existing MLaaS systems have been under scrutiny. Deployed systems like SageMaker [1] require clients to specify the model used for each inference request; consequently users must perform extensive offline testing to profile different model variants to figure which best meets their runtime demands. Furthermore, at runtime the users are decoupled from the system environment and cannot react quickly to spiking loads, which can lead to uncertain latency response times. Recognizing these issues, recent projects [16] [5] [25] have described MLaaS systems which store inference-serving model variants and internally switch service between them at runtime to react to system load and client demands; we categorize these as **dynamic MLaaS systems**. Instead of requiring users to specify the model which serves their request, dynamic systems query service requirements and choose the best model variant for inference internally. For instance, if a user requests higher latency (to reduce costs) or many users submit requests at the same time, then the system may switch to a model variant which is not as accurate, but can make faster responses through lowered computation costs. Thus, the effective range of the dynamic system is dependent on the accuracy/performance trade-offs of its model variants - with a greater trade-off range the dynamic system can offer more service options to the user and react to greater workload fluctuations.

The general framework of a dynamic MLaaS system is shown in Fig. 1. A client makes a query with performance objectives. Based on the objectives, the request is routed to a worker, which uses a variant from the model variant repository to serve it. Finally, the response is sent to the client. Asynchronously, the model variant repository is filled by a variant generator, which takes input data to train a variant and profiles it for the system's selection policy. Model variant generation may be orchestrated by the system itself [16] or offline [5]. Compared with traditional MLaaS systems, dynamic systems offer higher predictability, finer client control, and more efficient development and scaling.

2.2 Graph Neural Networks

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

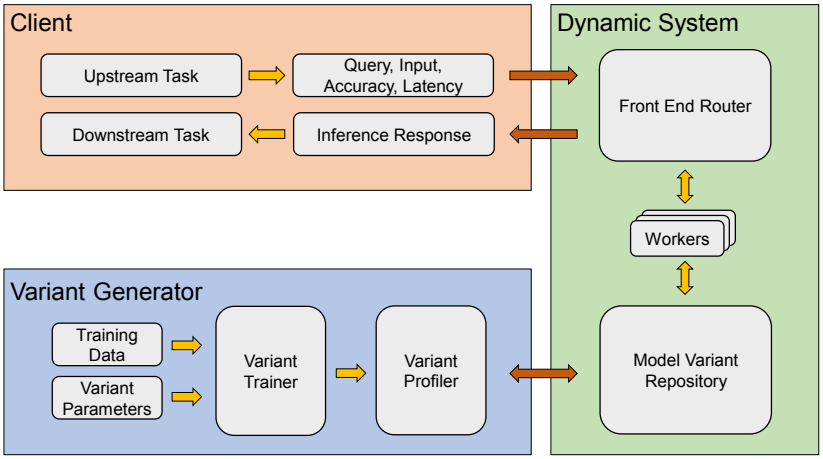

Fig. 1. General framework of a dynamic MLaaS system.

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 instance, the GraphSAGE [6] sampling algorithm defines per-layer constants 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 edges are sampled.

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

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

GNN architectures such as GCN [10], GraphSAGE [6], and GIN [20] differ in aggregation schemes and transformation methods. A GNN model's architecture is not dependent on its neighbor sampling algorithm [22], i.e. which neighbors it chooses to aggregate in each layer. For simplicity, in this paper we adhere to the aforementioned GraphSAGE sampling algorithm for describing our techniques.

3 Approach

To identify opportunities for reducing GNN inference latency, we first develop an analytical performance model in Section 3.1 of the communication and computation of GNN inference. Our model shows that for GNN architectures used in real world applications, communication is a major bottleneck. Hence, in Section 3.2, we develop an approach to generate model variants with greatly reduced latency by reducing communication.

3.1 GNN Inference Performance Model

Because of the large graphs and tensor computations used in GNN inference, the computation is offloaded to GPU [26], FPGA [24], or ASIC [21] accelerators. Thus, we analyze the communication and computation time of GNN inference on a heterogeneous CPU-Accelerator platform, using the parameters:

- t: Number of target nodes for inference
- L: Number of layers in the GNN model
- f_k: Feature dimension of the k-th GNN layer, where f_0 denotes input feature dimension and f_L denotes output embedding dimension
- s_k: Number of neighbors sampled for aggregation in layer k
- b_f: Bytes per feature
- BW: Bandwidth for communication between CPU and accelerator
- OPS: Accelerator compute power, in operations per second

Communication To begin, the input features must be communicated to the accelerator. For each target node, s_L neighbors are aggregated, and each of those neighbors must aggregate s_{L-1} neighbors, and so on, creating a compounding neighbor explosion effect. Thus, the number of unique nodes N_0 to be sampled for input features is at most the product of neighbors sampled.

$$N_0 \leq s_1 \cdot s_2 \cdot ... \cdot s_L \cdot t = t \prod_1^L s_k$$

Many of the neighbors sampled will be repeated since the same node can simultaneously be in the neighborhoods of many other nodes. To simplify, we assume a fraction c of the maximum are unique nodes, while the rest are overlapped.

$$N_0 = c \left(t \prod_1^L s_k \right)$$

For each of these nodes, we must transfer f_0 features, each of size b_f. Considering communication bandwidth, transfer time is thus

$$T = \frac{c \left(t \prod_1^L s_k \right) \cdot f_0 \cdot b_f}{BW}$$

Once the accelerator has completed computation, the final target node embeddings are transmitted back to the CPU, for the toal communication time

$$T_{comm} = \frac{c\left(t\prod_1^L s_k\right)\cdot f_0 \cdot b_f}{BW} + \frac{t \cdot f_L \cdot b_f}{BW}$$

Given that the general goal of machine learning inference is to reduce the dimension of raw data to lower-dimensional embeddings, generally $f_0 > f_L$. Combined with the neighbor explosion effect of sampling across the layers, communication volume of input features at the beginning of inference can be orders of magnitude greater than communication volume of target node embeddings after computation.

Computation Once input features have been transferred to the accelerator, it is ready for inference computation. In each layer, computation performs *aggregation*, *transformation*, and *activation*. The number of nodes which are targets for hidden feature generation in layer l, N_l, is dependent on the sampling of each layer that comes afterwards. Again for simplicity, to account for repeated neighbors we assume the fraction c of these nodes are unique.

$$N_l = c(s_{l+1} \cdot s_{l+2} \cdot ... \cdot s_L \cdot t) = c\left(t \prod_{k=l+1}^{L} s_k\right)$$

For aggregation, in layer l all nodes which need embeddings sample and aggregate previous layer's features for s_l of their neighbors, for number of operations

$$AggregateOps_l = N_l \cdot s_l \cdot f_{l-1}$$

The aggregated features are transformed to the new feature dimension by multiplying by the weight matrix, for number of operations

$$TransformOps_l = N_l \cdot f_{l-1} \cdot f_l$$

Finally, the transformed features are passed through an activation function, which we assume is a constant operation

$$ActivationOps_l = N_l \cdot f_l$$

Adding the total operations for each layer of the GNN and considering compute power of the accelerator gives total compute time

$$T_{comp} = \frac{\sum_{l=1}^{L}\left(c\left(t\prod_{k=l+1}^{L} s_k\right)\cdot((s_l \cdot f_{l-1}) + (f_{l-1} \cdot f_l) + f_l)\right)}{OPS}$$

Analysis: We analyze our performance model by simulating a real-world inference case, using a 2-layer GNN model performing inference for a batch size of 1024 nodes. All feature/embedding dimensions are 128, and the number of

neighbors sampled in every layer is 25, with unique nodes factor $c = 0.5$. Each feature is 4 bytes and the accelerator for computation is a modern server GPU [18], which has a maximum bandwidth of 30 GB/s (PCIe 4.0 x 16). For compute power, based on the number of cores and frequency in the GPU, we estimate max power to be 14 GFLOPS. Considering the limitations in our computation analysis, such as simple operation counting and that not all cores may be active at a given time, we conservatively estimate to utilize 5% of maximum compute power, which is 0.7 GFLOPS.

Under these parameters, we find input feature transfer time of **5.46ms**, computation time of **0.35ms**, and transfer of final target embeddings **0.01ms**. The large input neighborhood caused by the compounding neighbor effect results in a high transfer volume for input features, which causes input feature transfer time outweighs computation latency during GNN inference. In Section 4.2, we show evidence of the communication bottleneck empirically, and in most cases we observe an order-of-magnitude difference between communication time and computation time in practice as well. Given this, we identify input feature transfer as an important area for optimization to reduce the latency of GNN inference.

3.2 Input Feature Pruning

Motivated by the transfer-time bottleneck, we propose to increase the performance of GNN inference on heterogeneous platforms by pruning input features. Given the large impact of feature communication on overall latency, reducing the volume of transferred data can significantly decrease 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 exhibits the accuracy/performance trade-off for generating high- and low-complexity model variants to be used in dynamic MLaaS systems.

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

To generate different model variants we vary prune degree by changing the size of X_S^0. For ease of choosing different pruning degrees, we order the features into an importance ranking and choose from the top of the ranking to construct the subset S, pruning away lower-ranked features. We study model variants generated using three methods of feature ranking - *Random*, heuristic-based *Greedy*, and regression-based *Lasso*. These methods were first proposed to prune the hidden channels of GNN models for the sake of reduced computation [26]; in

this work, we extend them to GNN input feature pruning for the sake of reduced communication.

Random The features are ranked randomly.

Greedy 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. 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.

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 hidden features (X^1) generated from masked input features and first-layer hidden 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 from nodes in the training set, and penalty factor λ we optimize the coefficient mask β for the problem

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

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. 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.

3.3 Generating Model Variants

Once the features are ranked, they can be used to create pruned input feature matrices X_S^0 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 dynamic MLaaS 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. 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.

4 Evaluation

Table 1. Dataset statistics. (s) and (m) denote single- and multi-label classification.

Dataset	Nodes	Edges	Features	Classes
Flickr [4]	89,250	899,756	500	7(s)
Arxiv [8]	169,343	1,166,243	128	40(s)
Reddit [4]	232,965	11,606,919	602	41(s)
Yelp [4]	716,847	6,977,410	300	100(m)
Products [8]	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. All experiments are run on a machine equipped with an AMD Ryzen 3990X CPU and NVIDIA A6000 GPU, with a reported bandwidth of 13.5 GB/s from host to device.

We choose five datasets for GNN node classification from Pytorch Geometric [4] and Open Graph Benchmark [8], described in Table 1, and run experiments under inductive settings [6] (test nodes are unseen during training). All are single-classification except Yelp, which assigns multiple classifications per node, thus "Accuracy" refers to the F1-micro score for Yelp evaluation.

Our models are implemented using Python3 and Pytorch Geometric [4]. All models have two layers (one hidden feature transformation between input and output features). We consider two GNN architectures, GIN [20] and GraphSAGE [6]. We train all model variants with a batch size of 1024 and test the accuracy and latency values with batched inference of 1024 using the testing node set. We use GraphSAGE neighbor sampling (see Section 2.2) with constants 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. From the feature ranking we generate model variants with half, a quarter, and an eighth of original input features, referred to as 2×, 4×, 8× in our results. The unpruned 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.

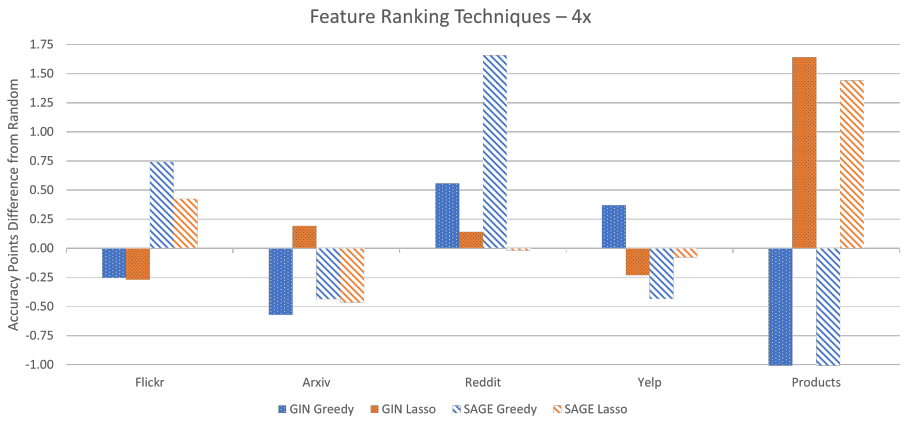

Fig. 2. Accuracy comparison of 4× pruned model variants produced with the different feature ranking techniques. GIN Greedy/Lasso are compared to GIN Random, SAGE Greedy/Lasso to SAGE Random.

4.1 Comparison of Ranking Techniques

We first compare the model variants generated by the three ranking techniques for pruning, as shown in Figure 2. For each of the GNN architectures, we use Random ranking accuracy as a baseline and show the accuracy points difference above or below using Greedy and Lasso techniques. We choose to show 4× models as the intermediate pruning amount. Note that the ranking techniques have no effect on inference latency, since communication volume is dependent on prune amount.

For most of the datasets, there is a negligible difference between the accuracy of the models generated by the different ranking techniques, evident by the short bars. Surprisingly, the Random ranking can often generate model variants as good or slightly better than Greedy and Lasso ranking, which suggests that for many datasets, there is low variation in the importance of input features, such that choosing any of them randomly yields enough information for nearly the highest quality predictions.

However, Greedy and Lasso can outperform Random by significant margins of more than 1%. Between the two, Lasso more consistently chooses important features, especially considering the Products dataset, in which both Greedy models perform worse than Random and further below Lasso. As previously mentioned, the heuristic of adding the magnitudes from W^1 corresponding to each feature used in Greedy ranking can fall short when features have differing average magnitudes. The features of Products nodes exhibit a wide range - the maximum standard deviation of the features is 192. The other datasets' features do not show such wide variance, for instance Arxiv has a maximum standard deviation of 0.47. While Greedy ranking may be a suitable heuristic for ranking datasets with known low-variance features, for instance normalized features, due to their inconsistency with high-variance features we opt to study Lasso model variants in further evaluation.

Table 2. Experimental results for GIN variants with Lasso ranking.

Dataset	Prune	Accuracy	Lat (ms)	Comm/Comp	Size (KB)	Mem (MB)
Flickr	1×	50.85	13.44	15.08	268	171
	2×	50.77	7.52	8.01	209	88
	4×	49.78	3.71	5.01	176	44
	8×	49.36	2.40	3.33	160	23
Arxiv	1×	68.90	4.15	4.14	184	84
	2×	68.57	2.79	3.68	169	44
	4×	66.56	2.01	2.84	161	23
	8×	60.97	1.60	2.30	157	13
Reddit	1×	91.59	43.69	53.43	432	538
	2×	89.45	24.46	29.25	359	272
	4×	93.88	13.65	15.83	320	138
	8×	93.26	8.28	9.84	301	71
Yelp	1×	62.90	13.77	16.19	730	1097
	2×	61.25	7.96	8.48	657	688
	4×	58.14	4.29	3.95	618	484
	8×	54.75	2.96	3.10	599	382
Products	1×	71.31	11.42	13.19	308	954
	2×	69.39	5.84	5.82	296	488
	4×	66.56	4.19	4.68	290	254
	8×	60.06	3.15	4.10	286	133

4.2 Results with Lasso Ranking

In Tables 2 and 3 we show results for input feature pruning for all 5 datasets using the Lasso ranking technique for GIN and SAGE architectures, respec-

Table 3. Experimental results for SAGE variants with Lasso ranking.

Dataset	Prune	Accuracy	Lat (ms)	Comm/Comp	Size (KB)	Mem (MB)
Flickr	1×	50.44	14.87	9.97	1021	174
	2×	49.81	8.58	7.49	523	90
	4×	49.52	4.19	3.86	272	46
	8×	48.44	2.78	3.18	145	25
Arxiv	1×	69.40	4.42	3.13	680	86
	2×	67.29	3.07	2.55	424	46
	4×	65.67	2.23	1.77	296	25
	8×	59.93	1.93	1.52	232	15
Reddit	1×	94.42	51.86	12.64	1293	541
	2×	94.27	28.03	11.22	693	274
	4×	94.68	15.69	8.81	390	140
	8×	93.30	9.50	6.61	240	71
Yelp	1×	63.32	15.66	9.35	1608	1100
	2×	61.33	8.82	6.44	1009	697
	4×	58.82	4.57	3.10	709	493
	8×	56.37	3.32	2.29	556	390
Products	1×	72.43	12.40	9.19	301	954
	2×	71.60	6.31	4.80	201	488
	4×	68.57	4.59	3.77	151	253
	8×	63.58	3.60	3.24	125	132

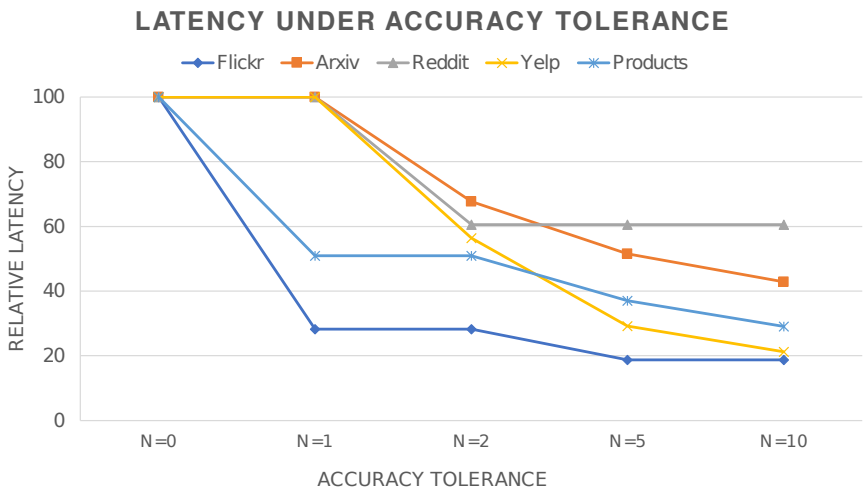

Fig. 3. Minimum latency under different accuracy tolerances for SAGE Lasso variants.

tively. For the un-pruned original model and the three pruned model variants, we show accuracy, latency, the communication latency to computation latency ratio, serialized model size, and GPU memory usage. Latency measures the time beginning when input features are transferred to the GPU and ending once GPU computations are complete.

In all results, we notice a strong trend of high communication to computation latency ratios. Nearly all un-pruned $1\times$ models show around an order-of-magnitude difference, confirming the results from our analytical model in Section 3.1. When the communication bottleneck is large, denoted by a large Comm/Comp ratio, pruning half of the input features can nearly cut latency in half as well. For almost all datasets, the $8\times$ pruned model can achieve around an 80% reduction in latency from the $1\times$ original models. The exception is Arxiv, which already exhibits a small Comm/Comp ratio due to its relatively low degree.

For all other datasets, node degree is higher and the Comm/Comp ratio can stay high even as many features are pruned. Given the nearly linear latency improvements with respect to pruning amount noted when communication is a bottleneck, one could extrapolate the number of features to prune to generate a model variant with around a certain latency, which can be useful if a system is searching for a model variant to fulfill a specific latency objective.

In both the GIN and SAGE architectures, accuracy results between the same datasets are similar, but against other 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 correlated to the dimension of input features in the dataset. Arxiv and Products have the fewest features, 128 and 100 respectively, and model variants for these datasets exhibit the largest accuracy loss with respect to pruned feature amount. Using only 16 and 12 features, $8\times$ pruned models for Arxiv and Products struggle to learn classification to apply to thousands or millions of unseen nodes, unlike for a graph with more features like Reddit, which still has 75 features in its $8\times$ pruned models.

On the other end of the spectrum, for Reddit in both SAGE and GIN we see increased accuracy for pruned models than for models using a larger number of input features. Nevertheless, the pruned models in these cases do not significantly outperform the other models, so we attribute these results to alleviation of overfitting. The datasets, having large input feature dimensions, can have too much raw information which can overfit its un-pruned models to the training data. As input features are pruned away, increased sparsity can generate more accurate models since there is less learned noise [14] and thus better prediction generalization.

Pruning models not only affects their accuracy and latency, but also their size and GPU memory usage at runtime. 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.

4.3 Latency Under Accuracy Tolerance

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 [5]. For this metric, we first identify the model which serves inference at 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 figure 4.1 for the SAGE Lasso model variants.

The accuracy tolerance graph represents the range of generated model variants, of which we have only four per dataset, thus there can be at most four distinct latency values. The lowest distinct accuracy tolerances can serve as options for the user to choose between in their dynamic system requests. For instance, for the Flickr dataset, a user may be presented with $N = 0$, 1, 5 as their tolerance choices, with relative latency 100, 30, and 20 respectively.

For a greater number and finer granularity of accuracy tolerances, the system must search more pruned models and host more model variants. However, as shown in figure 4.1, the simple exponential pruning scheme that we implemented often generates a wide enough range of model variants for dynamic inference service. For a 2% tolerance, inference latency can be decreased by 30% to 70%, and extending to a 5% tolerance can decrease latency by another 10 to 30 points.

4.4 Model Variant Profiling

In a system operating under an inductive setting [6], 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.

Figure 4 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.

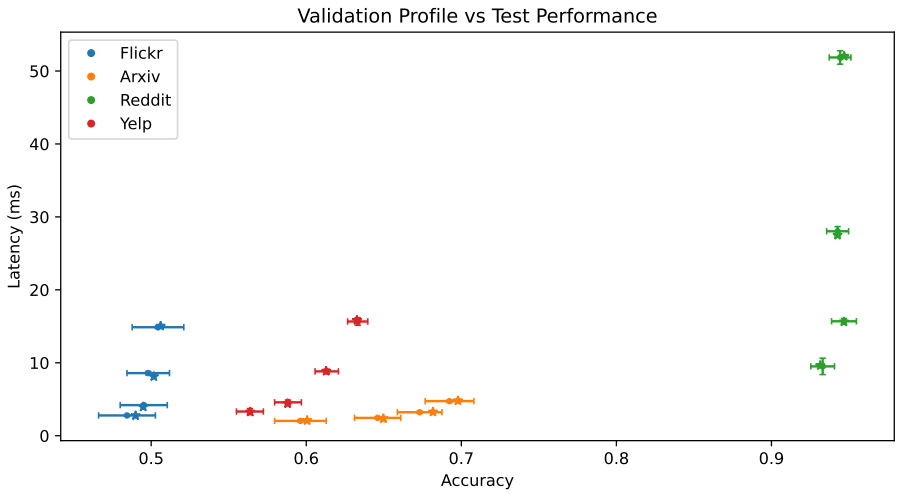

Fig. 4. Static performance modeling for SAGE variants under Lasso ranking, for all datasets except Products. Stars represent expected performance based on validation set, crosses represent test set performance with 1 std.

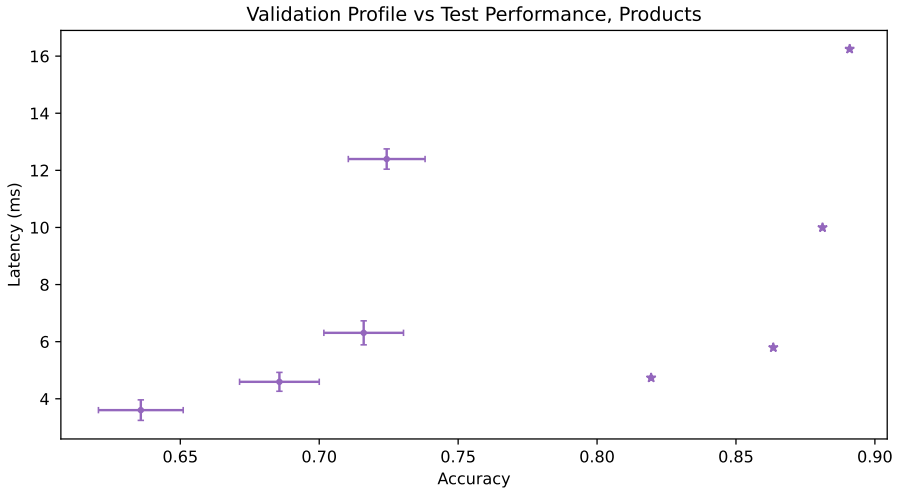

Fig. 5. Static performance modeling for SAGE Products with Lasso ranking.

The exception to the other datasets is Products, for which validation profile vs test performance graph is shown in Figure 5. In this case, not only are validation profiles far too accurate, they also have higher latency expectations. Whereas the other datasets had generally similar structures between validation and testing 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. Given the Products dataset construction of training/validation nodes as popular items while test nodes are unpopular items, and that edges between nodes are constructed when items are purchased together [8], it is consistent with our evaluation.

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.

5 Conclusion

In this paper we extended dynamic MLaaS systems to GNN applications by providing a technique for generating GNN model variants which present an accuracy/performance trade-off range to the system. We analyzed the execution of GNN inference and identified communication as a latency bottleneck, which motivated our approach to prune input features. Over different ranking techniques, GNN architectures, and datasets, we showed the effectiveness of input feature pruning to generate model variants for use in dynamic systems.

We are the first to study the intersection between GNN applications and dynamic MLaaS systems. However, we can identify other GNN works which have potential to be extended to model variant generation: Channel pruning [26] decreases internal feature dimension and thus can create smaller model variants, model variants of low-complexity architectures like SGC [19] can be used in conjunction with high-complexity architecture model variants to form the performance range, GNN models can be quantized [17] to reduce computation complexity via simpler arithmetic operations, and aggregation neighborhoods can be reduced with neighborhood sampling algorithms [23]. Each of these provides a different direction towards which model variants can be generated, and we note that input feature pruning is agnostic to all of them - our technique may be combined with any of these GNN works to generate greater trade-offs. In future work, we will study further into GNN-integrated dynamic systems, for instance in dynamic profiling algorithms and policies for scheduling and selection.

References

1. AWS: Introducing amazon sagemaker support for deep graph library (dgl): Build and train graph neural networks (2019), https://aws.amazon.com/about-

aws/whats-new/2019/12/introducing-deep-learning-graphs-with-dgl-sagemaker/
2. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners (2020)
3. Derrow-Pinion, A., She, J., Wong, D., Lange, O., Hester, T., Perez, L., Nunkesser, M., Lee, S., Guo, X., Wiltshire, B., et al.: Eta prediction with graph neural networks in google maps. Proceedings of the 30th ACM International Conference on Information & Knowledge Management (Oct 2021). https://doi.org/10.1145/3459637.3481916, http://dx.doi.org/10.1145/3459637.3481916
4. Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)
5. Halpern, M., Boroujerdian, B., Mummert, T., Duesterwald, E., Janapa Reddi, V.: 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). https://doi.org/10.1109/ispass.2019.00012, http://dx.doi.org/10.1109/ISPASS.2019.00012
6. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: NIPS (2017)
7. Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., Wang, X.: Applied machine learning at facebook: A datacenter infrastructure perspective. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 620–629 (2018). https://doi.org/10.1109/HPCA.2018.00059
8. Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., Leskovec, J.: Open graph benchmark: Datasets for machine learning on graphs. arXiv preprint arXiv:2005.00687 (2020)
9. Hutt, G., Viswanathan, V., Nadolski, A.: Deliver high performance ml inference with aws inferentia (2019), https://www.youtube.com/watch?v=17r1EapAxpk&ab_channel=AWSEvents
10. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (ICLR) (2017)
11. Lee, K., Rao, V., Arnold, W.: Accelerating facebook's infrastructure with application-specific hardware (2019), https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/
12. Liu, Z., Chen, C., Yang, X., Zhou, J., Li, X., Song, L.: Heterogeneous graph neural networks for malicious account detection (2020)
13. Pal, A., Eksombatchai, C., Zhou, Y., Zhao, B., Rosenberg, C., Leskovec, J.: Pinnersage. Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (Jul 2020). https://doi.org/10.1145/3394486.3403280, http://dx.doi.org/10.1145/3394486.3403280
14. Rasmussen, C.E., Ghahramani, Z.: Occam's razor. Advances in neural information processing systems pp. 294–300 (2001)

15. Reddi, V.J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., Chukka, R., Coleman, C., Davis, S., Deng, P., Diamos, G., Duke, J., Fick, D., Gardner, J.S., Hubara, I., Idgunji, S., Jablin, T.B., Jiao, J., John, T.S., Kanwar, P., Lee, D., Liao, J., Lokhmotov, A., Massa, F., Meng, P., Micikevicius, P., Osborne, C., Pekhimenko, G., Rajan, A.T.R., Sequeira, D., Sirasao, A., Sun, F., Tang, H., Thomson, M., Wei, F., Wu, E., Xu, L., Yamada, K., Yu, B., Yuan, G., Zhong, A., Zhang, P., Zhou, Y.: Mlperf inference benchmark (2020)
16. Romero, F., Li, Q., Yadwadkar, N.J., Kozyrakis, C.: Infaas: Automated model-less inference serving. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21). pp. 397–411. USENIX Association (Jul 2021), https://www.usenix.org/conference/atc21/presentation/romero
17. Tailor, S.A., Fernandez-Marques, J., Lane, N.D.: Degree-quant: Quantization-aware training for graph neural networks (2021)
18. TechPowerUp: Nvidia rtx a6000 specs (2020)
19. Wu, F., Zhang, T., de Souza Jr. au2, A.H., Fifty, C., Yu, T., Weinberger, K.Q.: Simplifying graph convolutional networks (2019)
20. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations (2019), https://openreview.net/forum?id=ryGs6iA5Km
21. Yan, M., Deng, L., Hu, X., Liang, L., Feng, Y., Ye, X., Zhang, Z., Fan, D., Xie, Y.: Hygcn: A gcn accelerator with hybrid architecture (2020)
22. Zeng, H., Zhang, M., Xia, Y., Srivastava, A., Malevich, A., Kannan, R., Prasanna, V., Jin, L., Chen, R.: Decoupling the depth and scope of graph neural networks. In: Thirty-Fifth Conference on Neural Information Processing Systems (2021), https://openreview.net/forum?id=d0MtHWY0NZ
23. Zeng, H., Zhang, M., Xia, Y., Srivastava, A., Malevich, A., Kannan, R., Prasanna, V., Jin, L., Chen, R.: Decoupling the depth and scope of graph neural networks. In: Thirty-Fifth Conference on Neural Information Processing Systems (2021)
24. Zhang, B., Zeng, H., Prasanna, V.: Accelerating large scale gcn inference on fpga. In: 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). pp. 241–241 (2020). https://doi.org/10.1109/FCCM48280.2020.00074
25. Zhang, J., Elnikety, S., Zarar, S., Gupta, A., Garg, S.: 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), https://www.usenix.org/conference/hotcloud20/presentation/zhang
26. Zhou, H., Srivastava, A., Zeng, H., Kannan, R., Prasanna, V.: Accelerating large scale real-time gnn inference using channel pruning. Proceedings of the VLDB Endowment **14**(9), 1597–1605 (May 2021). https://doi.org/10.14778/3461535.3461547, http://dx.doi.org/10.14778/3461535.3461547
27. Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., Zhou, J.: Aligraph: A comprehensive graph neural network platform (2019)