LiGNN: Graph Neural Networks at LinkedIn

Fedor Borisyuk, Shihai He, Yunbo Ouyang, Morteza Ramezani, Peng Du, Xiaochen Hou, Chengming Jiang, Nitin Pasumarthy, Priya Bannur, Birjodh Tiwana, Ping Liu, Siddharth Dangi, Daqi Sun, Zhoutao Pei, Xiao Shi, Sirou Zhu, Qianqi Shen, Kuang-Hsuan Lee, David Stein*, Baolei Li*, Haichao Wei, Amol Ghoting, Souvik Ghosh
LinkedIn Inc.

ABSTRACT

In this paper, we present *LiGNN*, a deployed large-scale Graph Neural Networks (GNNs) Framework. We share our insight on developing and deployment of GNNs at large scale at LinkedIn. We present a set of algorithmic improvements to the quality of GNN representation learning including temporal graph architectures with long term losses, effective cold start solutions via graph densification, ID embeddings and multi-hop neighbor sampling. We explain how we built and sped up by 7x our large-scale training on LinkedIn graphs with adaptive sampling of neighbors, grouping and slicing of training data batches, specialized shared-memory queue and local gradient optimization. We summarize our deployment lessons and learnings gathered from A/B test experiments. The techniques presented in this work have contributed to an approximate relative improvements of 1% of Job application hearing back rate, 2% Ads CTR lift, 0.5% of Feed engaged daily active users, 0.2% session lift and 0.1% weekly active user lift from people recommendation. We believe that this work can provide practical solutions and insights for engineers who are interested in applying Graph neural networks at large scale.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Human-centered computing** → **Social networking sites**; • **Information systems** → **Recommender systems**.

KEYWORDS

Graph Neural Networks, GNN, Recommender Systems

ACM Reference Format:
Fedor Borisyuk, Shihai He, Yunbo Ouyang, Morteza Ramezani, Peng Du, Xiaochen Hou,, Chengming Jiang, Nitin Pasumarthy, Priya Bannur, Birjodh Tiwana, Ping Liu, Siddharth Dangi,, Daqi Sun, Zhoutao Pei, Xiao Shi, Sirou Zhu, Qianqi Shen, Kuang-Hsuan Lee, David Stein*, Baolei Li*, Haichao Wei, Amol Ghoting, Souvik Ghosh. 2018. LiGNN: Graph Neural Networks at LinkedIn. In *Proceedings of (KDD'24).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

KDD'24, August 25–29, 2024, Barcelona, Spain

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

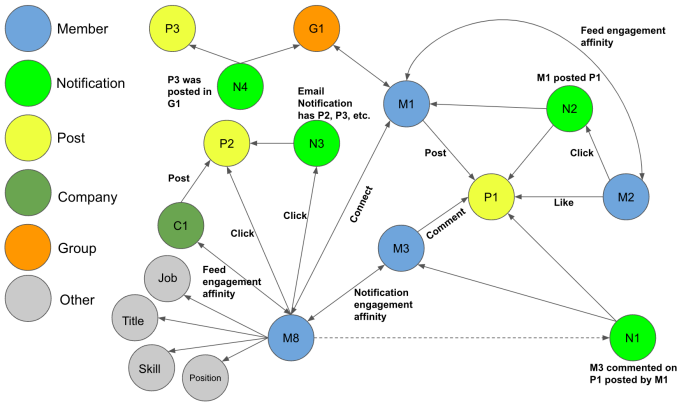

Figure 1: Schematic representation of LinkedIn Graph. Members engaging with Posts, Jobs, Groups, Companies and other members.

1 INTRODUCTION

LinkedIn is the world's largest professional network with more than 1 billion members in more than 200 countries and territories worldwide. LinkedIn's ecosystem encompasses members, companies, universities, students, and groups, all forming connections within the professional graph. Hundreds of millions of LinkedIn members actively engage to seek opportunities and connect with professionals. Integrating all these entities into the graph is an intuitive step. Our graph boasts up to a hundred billion nodes and several hundred billion edges. Graph edges symbolize various activities on the LinkedIn app, such as job applications, post engagements, and networking interactions (see Figure 1).

Developing *LiGNN* at scale presented several challenges:

- GNN training at scale: Unlike traditional DNN training, GNN training has unique training scalability issues due to graph hosting requirements (§3.2).
- Diverse entities: our goal was to create a unified graph embedding space for various entities like posts, members, companies, and jobs (§3.1).
- Cold start: infrequent visits by some LinkedIn members result in limited preference data (§3.5).
- Dynamic system: the dynamic, temporal nature of the LinkedIn ecosystem limited the capabilities of GNN models.

*Work done while at LinkedIn.

Our technological advancements in Graph Neural Networks address these challenges. The **contribution of the paper** consists of:

- GNN training at scale: section §3.2 outlines the GNN training infrastructure, while §4 discusses scaling LinkedIn's GNN training. This involves integrating Microsoft's real-time DeepGNN graph engine [25] with GPU node training jobs. We propose novel techniques using adaptive sampling and node grouping to expedite training, and share our approaches on effective data processing with shared memory queue. We also share our path on how we achieved high GNN training stability (§4.1).
- Diverse entities: in §3.1 we share our experience of building and optimizing a graph for multi-task environments at LinkedIn scale, integrating different LinkedIn entities into a singular embedding space.
- Cold start: to improve the experience for less active members, we propose methods for graph densification (§3.5) and share our experience on speeding up multi-hop graph sampling (§3.6).
- Dynamic system: to keep up with LinkedIn's dynamic ecosystem, we integrated Temporal Graphs with long-term optimization (§3.4) and implemented near-line graph serving (§5). Temporal modeling helps GNNs discern historically significant member interactions, while the near-line graph infrastructure swiftly reflects interactions in member and item representations. Our solution to incorporate temporal aspects in graphs is simpler and more scalable, making it suitable for production, compared to conventional temporal graph architectures [24, 28].

2 RELATED WORK

Graph Neural Networks (GNNs) are effective for modeling graphs [7] and relational data [5]. Much research has focused on enhancing GNN model architectures [7, 11, 24, 27]. Our work builds upon the SAGE [7] architecture, integrating sequential temporal modeling with transformer-based sequence modeling and long-term losses [21, 23], tailored to the GNN domain.

GNNs propagate signals across graphs, where neighbor sampling is crucial. We found Personalized PageRank (PPR) [2, 7, 30, 31] sampling to be more effective than random sampling. Our implementation uses two-hop sampling with forward push and random walks [2], showing scalability on LinkedIn data.

Several studies have aimed to accelerate GNN training jobs at industry scale such as MLPinit [8], GraphStorm [33], BigGraph [15], HUGE [20]. We introduce over three novel techniques in §4 that achieved a significant reduction in training time on large-scale production data.

GNNs are widely used in the industry for various applications like people recommendations [26], Anti-abuse [6], weather forecasting [14], Ads [19]. Our system, evaluated within LinkedIn, shows promising results in domains of People recommendations, Ads, Job Recommendations, and Feed post recommendations.

Numerous studies have explored cold start solutions, such as student-teacher consistency learning [17], attribute edges [22], self-supervised pre-training [10], meta-learning [9, 10], and similarity-based embeddings with LSH bucketing [6]. Our approach introduces artificial nearest neighbor edges to cold start nodes, leveraging content embeddings, which has shown quality improvements in various production applications at LinkedIn.

3 GNN MODELING AND TRAINING

3.1 Graph Construction

The graph used for the LinkedIn GNN models is a heterogeneous graph, which contains tens of node types and edge types, as shown in Figure 1. To densify the graph, we combine the subgraphs from different domains together, such as feed recommendations, job recommendions, notifications. Each domain can train their GNN models using its owned subgraph, or leveraging the combined graph. Over all, the graph contains 3 types of edges: (1) engagement edges, (2) affinity edges and (3) attribute edges. The engagement edges represent the engagements between LinkedIn's members and the contents on the LinkedIn platform, such as "member M2 liked post P1" is represented by an edge between M2 and P1. The affinity edges capture the historical engagements between LinkedIn's members and the creator of the contents, such as "member M2 has engaged with contents posted on LinkedIn by member M1" is represented by an edge between M2 and M1. The attribute edges capture the HAS-A relationships between two nodes such as "member M8 has a software engineer job" is represented by an edge between M8 and the corresponding job node. The edges are weighted by the strength of the affinity or engagements between two entities, except all the attribute edges use 1.0 as their edge weights. Currently, the combined graph contains up to a hundred billion nodes and several hundred billion edges.

3.2 Training infra

In LinkedIn, all GNN training and inference jobs are executed in the in-house Kubernetes (K8S) [13] based cluster, which has access to a Hadoop File System (HDFS). Each job needs to deploy a Graph Engine (GE, usually on CPU nodes) and a GNN Trainer (usually on GPU nodes) in the K8S cluster. The lightweight high-performance Microsoft DeepGNN [25] was chosen as the Graph Engine to provide fast real-time graph sampling with a variety of sampling strategies. As shown in Figure 2, the Graph Data Preparation step collects the graph data including edges and node features, and writes the data to HDFS. During training or inference, the GE loads the graph data into distributed memory and serves the data in real-time. Then, the GNN Trainer makes gRPC calls to the GE to fetch the sampled compute graphs and use the data for GNN training or inference. The GE can be deployed independently or as a part of specific training or inference job. Depending on the size of graph, one can launch one or more instances (pod[*]) to serve a portion of the partitioned graph. During training (or inference) the DeepGNN client queries the GEs with a given setup, which consists of the sampling algorithm and configuration, over gRPC. The resulting data is consumed by the underlying deep learning framework (Tensorflow). Figure 2 summarizes the GNN pipelines on K8S cluster at Linkedin.

[*]Pods are the smallest deployable units of computing in Kubernetes.

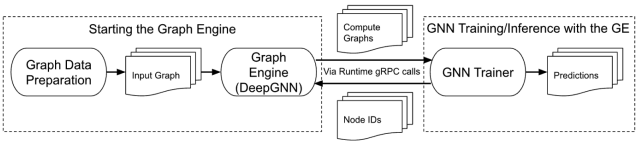

Figure 2: High level view of GNN pipelines.

3.3 GNN architecture

Considering the complexity of using GNN models to replace the existing machine learning models in LinkedIn, we adopted the the encoder-decoder architecture for the GNN models as shown in Figure 3. In this way, we can only take the trained encoder to generate the node embeddings and apply the embeddings in the downstream application models as new features. To handle the large scale LinkedIn graph and carry out inductive learning, the encoder adopts the GraphSAGE-style framework [7], which inductively generates the node embeddings based on graph sampling and neighborhood aggregation. The graph sampling is provided by the DeepGNN GE, including multi-hop random sampling, weighted sampling, Personalized PageRank (PPR) sampling. The neighborhood aggregation mainly uses the mean aggregation or the attention-based aggregation. The decoder of the GNN model takes the embeddings generated from the encoder as its input and computes the predictions. Currently we support Multilayer Perceptron (MLP) decoder, cosine decoder and in-batch negative sampling decoder [18] for link prediction tasks. The cosine decoder computes the cosine similarity between the source node embedding and the destination node embedding, and makes predictions on it. The in-batch negative sampling decoder treat all other samples in the batch as negative samples and makes predictions based on the dot products of each sample pairs. In the next section we discuss how we add transformer based architectures to *LiGNN*.

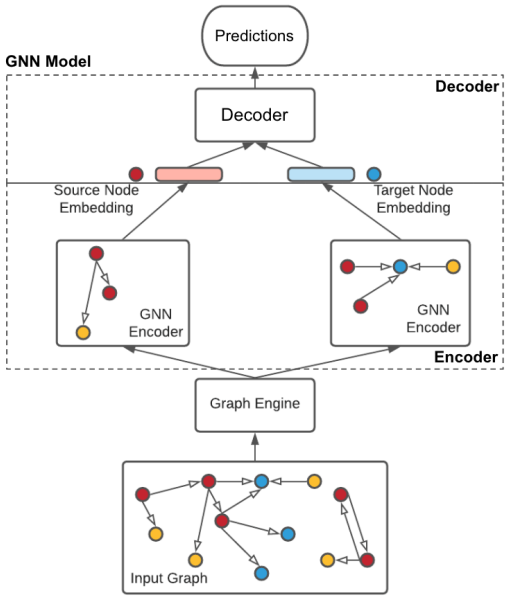

Figure 3: GNN model architecture.

3.4 Temporal Graphs

Experiment set up	AUC	Relative Lift
Baseline: SAGE encoder + BCE loss	0.71978	–
Temporal Encoder (TempEnc)	0.75204	4.48%
TempEnc + Positional encoding	0.75277	4.58%
TempEnc + timestamp encoding	0.75143	4.40%
TempEnc + Regular Causal mask	0.74991	4.19%
TempEnc + Prefix Causal Mask	0.75316	4.64%
TempEnc + use dst neighbors in src	0.75978	5.56%
TempEnc + long term loss, future history len = 40	0.75157	4.42%
TempEnc + long term loss, future history len = 20	0.75099	4.34%
TempEnc + long term loss, future history len = 10	0.75193	4.47%
all above combined (future history length = 10)	0.76176	5.83%

Table 1: Temporal model experiment results on Feed data.

The GNNs that we discussed so far are static, which lack temporal dynamics that is critical for professional social networks like LinkedIn, where interactions are time-sensitive. The rise of transformer models in recommendation systems, particularly for sequential data, suggests a natural fusion of GNNs with temporal sequence modeling [1, 21, 23]. Although there is research on event-driven and message-passing in dynamic GNNs [24, 28], their real-world applicability is limited. We redefine "temporal graphs" to focus on temporal sequence modeling within GNNs, as shown in Figure 4.

Our design modifies the neighbor sampling method of a SAGE [32] encoder by adding time-based node sampling to capture the last N (e.g. 100) activities of member before a certain time. We expand the SAGE encoder's output to multi-head dimensions with head number H (e.g. 4) by using a larger dimension equals $H * d$, reshaping it into a sequence with length H and dimension d. After that we encode the N activities with a node encoding module and concatenate them with the outputs of the SAGE encoder and eventually get a sequence of length $H + N$ and dimension d. Then we feed this sequence to the transformer's encoder to get an output of length $H + N$ and dimension d. We also add positional encoding [29] and prefix causal masking where for the first H tokens, we have full attention and for each token in the last N positions, they can only attend all the H tokens and tokens before itself in the sub-sequence of the N activity embeddings [16]. We combined binary cross entropy loss and a long term losses [21, 23] during training. We can cut the the length-N sequence into two parts: first part length equal N_1 and second part length equal N_2, where $N_1 + N_2 = N$. Long Term Loss extends prediction to N_2 future events, where we can use the output embedding at positions N_1 to predict embeddings from N_1 to N.

3.5 Graph Densification

The degree distribution in social network graphs often follows a power law, with most nodes having few interactions. This presents a challenge for neighborhood aggregation in GNNs, particularly for nodes with low out-degrees. To combat this, *LiGNN* implements graph densification by adding artificial edges based on auxiliary information. When a low-out-degree node is similar to a high-out-degree node, an artificial edge is added between them, leveraging

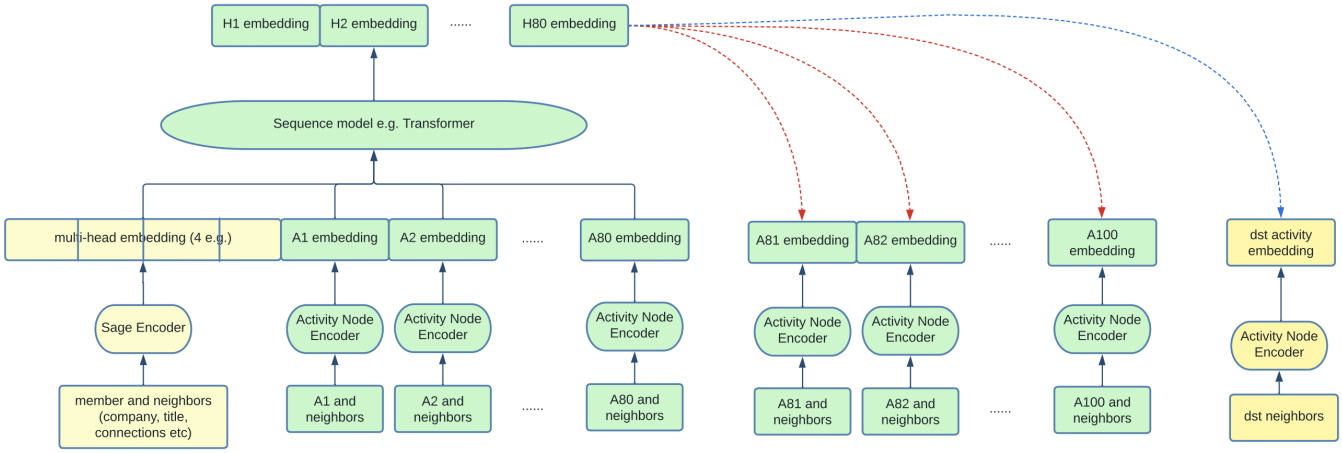

Figure 4: Temporal model includes: (1) static SAGE-encoder depicted in Yellow, (2) transformer based temporal sequence model in Green. Red lines depict long term loss and blue lines depict cosine similarity loss described in §3.3.

external content embeddings to gauge the similarity. The graph densification algorithm is outlined in Algorithm 1.

Algorithm 1 consists of three main functions. The Query function retrieves the embedding for a node. the approximate_knn function identifies the top k similar high-out-degree nodes for a low-out-degree node, using embedding similarity. For scalability in handling numerous nodes, we use an in-house approximate nearest neighbor search solution, based on HNSW [3]. The create_edge function forms artificial edges between a low-out-degree node and its top k similar high-out-degree counterparts. This method facilitates information flow from active nodes to less active nodes, mitigating cold start issues.

In LinkedIn's production scenarios, based on node out-degree quantiles we set degree_upper_bound at the 90th percentile, and degree_lower_bound set at the 30th percentile, with optimal results when k is around 50. We utilize different external embeddings for varying node types, such as profile LLM embeddings for member nodes, derived from member profile data, and content embeddings for item nodes, based on text and image content.

Algorithm 1 Graph Densification

Require: degree_lower_bound, degree_upper_bound, external_embeddings, k

function GRAPH_DENSIFICATION(degree_lower_bound, degree_upper_bound, external_embedding, k)
  low_degree_node_set ← {x|x's out-degree ≤ degree_lower_bound}
  high_degree_node_set ← {x|x's out-degree ≥ degree_upper_bound}
  high_degree_embedding_set ← {query(external_embeddings, node) for node in high_degree_node_set}
  for node **in** low_degree_node_set **do**
    node_embedding ← query(external_embeddings, node)
    top_k_set ← approximate_knn(high_degree_embedding_set, k, node_embedding)
    create_edge(top_k_set, node)
  end for
end function

3.6 Multi-hop Graph Sampling

For *LiGNN* to surpass traditional deep learning methods, effective sampling is key. Simple one-hop sampling falls short in capturing the complex graph topology, hence *LiGNN* adopts multi-hop sampling for deeper graph analysis. Three multi-hop sampling techniques were explored:

- **Multi-hop random/weighted sampling**: This method allows for either random sampling or user-configurable weighted sampling, where weights are adjustable for different edge types.
- **Multi-hop Personalized PageRank (PPR) Sampling**: Integral to *LiGNN*, PPR is a prominent tool in large-scale graph mining. It locates neighbors with the top k PPR scores relative to a source node, identifying key topological nodes. Despite PPR's slower pace compared to random/weighted sampling, efficiency is enhanced through approximate calculations using the Forward Push Algorithm and system-level optimizations. To accelerate PPR sampling for a batch of nodes, we consolidate sampling requests in each iteration of Forward Push into a single batch, reducing overhead.
- **Two-hop Personalized PageRank (PPR) Sampling**: Tailored for nearline serving, which currently only supports 2-hop methods, this approach returns neighbors within a 2-hop radius with the top k PPR scores. It utilizes a fast 2-hop random walk algorithm for PPR computation, offering quicker sampling than multi-hop PPR.

Subsequent sections will demonstrate the superiority of PPR sampling over multi-hop random/weighted sampling in terms of effectiveness. In experiments in Follow Feed and People recommendations, 2-hop PPR sampling contributes around 90% of gains and accelerate the sampling speed by 3 times, therefore we choose 2-hop PPR sampling as the default sampling strategy.

4 TRAINING STABILITY AND SPEED

In this section we will cover how we improved the stability and speed of GNN model training at LinkedIn.

Algorithm 2 Adaptive_Neighbor_Sampling

Require: starting_neighbor_count, final_neighbor_count, metric, tolerance, stride, tolerance_decay, min_update_freq, last_metric = 0.0

function ADAPTIVE_NEIGHBOR_SAMPLING(starting_neighbor_count, final_neighbor_count, metric, tolerance, last_metric, stride)
  current_neighbor_count ← starting_neighbor_count
  for epoch **in** epochs **do**
    for step **in** steps **do**
      train_data ← query(current_neighbor_count)
      train(model, train_data)
    end for
    current_metrics ← evaluate(model, val_data)
    if current_metric ≤ last_metric + tolerance or epoch % min_update_freq == 0 **then**
      current_neighbor_count ← min(current_neighbor_count + stride, final_neighbor_count)
    end if
    current_metric ←last_metric
    tolerance ← tolerance * tolerance_decay
  end for
  return model
end function

Algorithm 3 Grouping and Slicing

Require: training_data, group_size, gradient_step

function GROUPING_SLICING_SAMPLING(training_data, group_size, gradient_step)
  grouped_training_data ← group(training_data, key="member_id")
  sliced_padded_data ← slice_and_pad(grouped_training_data, group_size) ▷ pad dummy item ids and labels if count is smaller than group_size
  for epoch **in** epochs **do**
    for step **in** steps **do**
      training_batch ← query(sliced_padded_data) ▷ each row in training_batch contains one member and group_size items
      local_item_size ← group_size % gradient_step
      for i **in** gradient_step **do**
        member_prediction ← model(training_batch["member"])
        item_data, mask ← training_batch["item"][$i \times local\_item\_size : (i+1) \times local\_item\_size$] ▷ mask is to tell what items are padded
        item_prediction ← model(item_data)
        loss ← cal_loss(member_prediction, item_prediction, label, mask)
        update_model(model, loss)
      end for
    end for
  end for
  return model
end function

4.1 Training stability

Training GNN models, unlike conventional deep learning models, demands real-time graph sampling from the GE and retrieving labeled data from HDFS, intensifying network strain and affecting training stability. We implemented several techniques, boosting the training success rate from 30% to over 90%, as detailed in Table 2.

gRPC Retry: GNN training workers often fetch GBs of data from the GE for each batch via gRPC calls, straining the data transmission between workers and the GE server. With distributed training employing 6 to 24 workers, connection losses to the GE were common. By modifying the default gRPC retry policy to maximize "max_attempts" and "max_backoff", we effectively resolved the connection issue, enhancing the training success rate by 15%.

Horovod Training: Besides connection problems with the GE server, many job failures stemmed from worker-to-worker communication breakdowns. Transitioning from TensorFlow's MultiWorkerMirroredStrategy to Horovod distributed training, which utilizes NVIDIA's NCCL 2 and ring allreduce operation, significantly improved training stability, increasing success rates by 35%.

Memory Leak: Parallel data fetching in training workers, involving multiple prefetchers for graph data and storing batch data in a queue with a typical size of 10, usually consumed tens of GBs of memory. We observed delayed garbage collection, leading to memory leaks and out-of-memory failures. Adopting TensorFlow's GeneratorEnqueuer resolved this memory leak issue, further enhancing training stability by 10%.

4.2 Training speed

Increasing model size typically boosts performance, but it also lengthens training time. Thus, we focus on techniques to accelerate training, enabling swift iteration even as the model grows. We've also observed that GNN jobs are often data-bound, implying that optimizing neighbor collection from the graph engine can significantly impact training speed. Overall during development of GNNs at LinkedIn the training time reduced from 24 hours, when we started, to 3.3 hours on the latest training jobs, with largest contributions from Adaptive Neighbor sampling, Grouping and Slicing and Share-Memory Queue.

Reduce average step time: Typically, each training step is comprised of three components: data loading, forward and backward pass. If data-parallel distributed training is used, gradients need to be communicated across all workers through an AllReduce operation after backward pass. To reduce average step time, we can focus on optimization of the most time consuming components. Local gradient aggregation is a technique to reduce the frequency of gradient communication. Gradients will be aggregated locally on each worker for N mini-batches before they are sent to other workers through AllReduce. Note that local gradient aggregation is effectively increasing batch size by N times, and utilizing techniques like learning rate scaling [12] is important for large-batch training.

Mixed precision training [4] combines the use of half precision and single precision numerical formats in math operations. With GPU support, it brought 8% speedup in forward and backward pass of our GNN model training. Numerical underflow and overflow issues can occur under float16 computation. To avoid it, operations

with large reductions should be carried out in float32, model's output layer should also use float32 to guarantee accurate calculation of loss. In our experiments, keeping the last layer of node aggregator in float32 is also crucial to maintaining model accuracy under mixed precision training.

Increase convergence speed: We explored MLPinit [8], which trains node encoder weights from the node features in two tower style link-prediction matching without querying the GE. We observed 16.25% speedup from using MLPinit. Next we will show how we generalized MLPinit using Adaptive Neighbor Sampling strategy to decrease training speed even further.

Adaptive Neighbor Sampling: Since the I/O (reading data from GE) is the bottleneck of GNN training, we proposed several techniques to speeding up GNN by tackling the I/O part, one of which is to adaptively increase the number of neighbors to be sampled during training. We sample a small number of neighbors at the beginning and adaptively increase the neighbor count by monitoring the model performance. If the metric (e.g., AUC) keeps increasing with a small number of neighbors, we do not sample more neighbors. We only sample more neighbors when the metrics are not improved by a certain threshold. Since the number of neighbors and I/O time are correlated, starting with a small number of neighbors to learn a model can help save a large amount of training time (Algorithm 2).

Grouping and Slicing: The training dataset comprises millions of triplets including member IDs, item IDs (like follow feed posts or jobs), and labels, showing interactions between members and items. Notably, active members often interact with numerous items, confirmed by feed dataset analysis. Given the I/O constraints of GNN, traditional feature generation by querying neighbors for each member-item pair is inefficient due to repeated queries for active members. To optimize, we group training records by members, slicing grouped items and labels at a set threshold, then querying once for the member and grouped items. For instance, if a member has 10 interactions and the group size is 5, we create two data records for this member with 5 items each, cutting GE queries from 10 to 2, albeit each query being slightly more extensive.

Once we get the grouped data, e.g., one member with 5 items, there are two training approaches: (A) generate member and item embeddings together, compute average loss of the 5 pairs, backpropagate once, or (B) forward and backward passes for each pair, updating the model 5 times. While A is generally faster, it may underperform compared to B. However, with large model sizes, A can be a good way to reduce training time. We made the number of training passes configurable: it can be any divisible number between (A) and the group size (B). Experiments on LinkedIn data showed that using an intermediate number performs more effectively without reducing model quality and leads to a 69.9% reduction in training time. For more details, refer to Algorithm 3 for full details. See Table 3, Figure 5 and Figure 6 in appendix for results of experiments on Follow Feed and Job recommendations.

Python Multi-Processing with Shared Memory Queue: DeepGNN leverages C++ for computation-heavy tasks like sampling and querying node and edge features, but LinkedIn's models and sampling algorithms are primarily in Python. Python's operations, constrained by the GIL, involve considerable data processing and generation. To manage DeepGNN client's extensive data processing during training, we adopted Python Multi-Processing for parallel prefetching and pre-processing of data batches. However, using multiple Python processes entails overhead from data copying between parent and child processes. To minimize this, we crafted a shared-memory queue in native Python, employing the multiprocessing package to simultaneously query the DeepGNN Graph Engine across multiple processes. This approach efficiently prefetches and preprocesses the necessary training data. Our experiments demonstrated that this multi-processing with a shared-memory queue can reduce training times by as much as 68%.

GPU co-location: We also explored GPU co-location used in other graph engines [33], where we locate Graph Engine on the CPUs of GPU machines to save some TF-to-GE communication. However we didn't observe improvement in training speed. We observed that currently our training jobs are more constrained on TF-to-TF communication in comparison to TF-to-GE communication, leading to diminishing returns from co-location. The situation can change depending on the models we develop in the future.

Technique	Train Success Improvement
gRPC Retry	15%
Horovod Training	35%
Data Generator with GeneratorEnqueuer	10%

Table 2: Training Stability

Technique	Training time reduction
MLPinit	16.25%
Adaptive neighbor sampling	24.2%
Grouping and Slicing	69.9%
Mixed Precision	8.0%
LGA	35.2%
GPU Co-location	0.0%
Shared-Memory Queue	68.04%

Table 3: Training Time reduction techniques measured on one of our largest Follow Feed dataset. Training time reduced from 24 hours, when we started, to 3.3 hours.

5 NEAR-LINE INFERENCE

The online production system at LinkedIn places high importance on feature freshness, particularly when members interact with posts. To avoid stale recommendations from outdated GNN embeddings, a GNN model inference pipeline is used for near real-time generation of member/item GNN embeddings. The term "Item" encompasses recommendations like Posts, Jobs, Ads, and People.

This section covers two areas: the tech stack of the nearline pipeline and its architecture. LinkedIn's nearline pipeline utilizes Apache Beam. The Managed-beam team and the Machine Learning Infrastructure team at LinkedIn have contributed valuable components, such as SourceComponent, SinkComponent, and InferenceComponent, to assist AI engineers in minimizing development costs. However, challenges like the lack of batch feature fetchers, data converters for 2D tensors, and certain sampling functions were noted. To improve the pipeline for GNN embeddings, LinkedIn developed specific Beam components, aiming for wide applicability in GNN use cases. Integrating these with LinkedIn's ecosystem,

particularly the Frame framework and ML infra data types, was a significant challenge.

The GNN nearline pipeline, depicted in Figure 7, starts with an Item Creation event via Kafka, when member interacts with an Item (e.g., clicks, connects, applies). It performs joins to collect features for the GNN model, which then conducts inference. Outputs are stored in Venice feature storage or a Kafka topic for other Beam pipelines. This process also applies to member updates, with an added feature of tracking members interacting with the item.

6 EXPERIMENTS

In this section we present variety of vertical applications to demonstrate how GNNs can be applied to production. We will show ablation studies and online A/B test impact.

6.1 Experiments in Follow Feed

The LinkedIn Follow Feed recommends posts from a member's professional network. GNN embeddings are used in the Embedding-Based Retrieval (EBR) model of the Follow Feed recommendation system. These embeddings effectively capture the viewer's relationship with the post creator and interest in the post content. In the GNN model, the Follow Feed recommendation issue is treated as a link prediction task, determining the likelihood of a member interacting with a post. This model employs a SAGE encoder for generating member and post embeddings and a cosine decoder to calculate the similarity between these embeddings, predicting interaction probability. The cosine similarity thus indicates the relevance between member and post. In the EBR model, the ranking score for topK candidate selection combines post recency and relevance (cosine similarity). An offline experiment showed the GNN-based EBR model achieves a relative 9.6% improvement in recall compared to the existing rule-based candidate selection model. In the online A/B test we observe an relative improvement of 0.5% in Feed Engaged Daily Active Users.

The ablation study for the Follow Feed GNN model offers insights for how to construct the GNN model to achieve the best model performance. Table 4 highlights that including node ID embeddings significantly enhances model efficacy by an +15.3% in validation AUC. The graph sampling strategy plays a crucial role, with performance generally improving as more neighbors are sampled; a jump from 20 to 200 neighbors results in an 3.2% AUC increase. Different aggregators were evaluated, showing the attention aggregator outperforms the mean and self-attention (where each node attends to itself) aggregators with an +0.9% AUC increase. For link prediction tasks, using dual encoders (one for source and one for destination nodes) is more effective than a single encoder approach leading to an +2.5% AUC. The sparsity of the Follow Feed graph shows that adding cold start edges, as outlined in (§3.5), leads to an 0.5% improvement in validation AUC. The efficiency of sampling algorithms ranks as follows: 2-hop PPR Sampling > Random Sampling > Weighted Sampling. Implementing temporal modeling (§3.4) in the Follow Feed application resulted in an 5.8% AUC lift.

Experiment Setup	AUC Lift
Baseline: SAGE, 20 neighbors, Mean Aggregator	-
SAGE, 200 neighbors	+3.2%
+ Attention Aggregator	+0.9%
+ Dual Encoder	+2.5%
+ ID embeddings	+15.3%
+ Graph Densification	+0.5%
+ 2-hop PPR Sampling	+0.6%
+ Temporal Graph	+5.8%

Table 4: Follow Feed validation AUC for different GNN configurations. Techniques mentioned in the table are ordered in timeline of development and show incremental value of improvements on top of prior rows in the table.

6.2 Experiments in Out-Of-Network Feed

Beyond the in-network Follow Feed service, we've extended GNN applications to Out-Of-Network (OON) content recommendations. This feature allows LinkedIn members to discover content beyond their immediate network connections, tailored to their interests. OON posts are displayed on members' Feed pages or Notifications, based on engagement likelihood. We constructed the OON graph incorporating member-post engagement edges, member-creator affinity edges, and cold start edges (§3.1, §3.5). The OON GNN model employs a SAGE-encoder and cosine decoder. The trained SAGE-encoder produces GNN embeddings for members and posts, which are then integrated into the Embedding-Based Retrieval (EBR) system as an additional candidate generator for OON recommendations. Our online evaluations indicated significant improvements in key metrics, including an approximate relative increase of 0.2% in Daily Active Users (DAU) engaging with professional content.

6.3 Experiments in Job Recommendations

GNN member embeddings are utilized in LinkedIn's Top Applicant Jobs (TAJ), a premium feature that suggests jobs to members with higher likelihood of being accepted. TAJ's ranking problem is framed as a link prediction task, using a heterogeneous graph with nodes representing members, jobs, skills, and positions (company-title pairs). The graph, enriched by diverse edge types, and connects members to jobs based on their application history. The GNN model, trained on this rich graph, produces member embeddings for integration into TAJ's ranking models. GNN embeddings which are added on top of the existing two-tower models (member-job) have shown substantial improvements in key metrics, both offline and online, as detailed in Table 5. Notable relative achievements include a 0.3% increase in premium member subscription renewal, a 1% rise in the hearing back rate (applications receiving a positive response within 7 days), and a 1.8% growth in company follows. Additionally, temporal model in job recommendations yielded a 6.8% AUC lift in the Job Recommendation Ranking model, leading to relative increases of 0.4% in job viewers and 0.4% in total qualified applicants.

Evaluation	Metric	Lift
Offline	AUC	+1.1%
Online	Renewal Rate	+0.3%
	Positive Hearing Back Rate	+1.0%
	Company Follows	+1.8%

Table 5: Job Recommendation offline and online relative improvements.

Evaluation	Metric	Lift
Offline	Recall	+29.6%
Online	Sessions	+0.2%
	Weekly Active User	+0.1%
	New Member Connections	+2.4%

Table 7: Relative improvements in online A/B experiments within People Recommendation EBR with GNN embeddings

6.4 Experiments in People Recommendations

LinkedIn's people recommendation service suggests potential connections to members. GNN embeddings have been integrated into its retrieval phase. To train the GNN model, a substantial graph was constructed, comprising up to one billion member nodes and billions of connection edges. The weight of each edge between members u and v is determined by the formula:

$$\frac{\#\text{ of common connections between } u \text{ and } v}{\sqrt{\#\text{ of } u\text{'s connections}} \times \sqrt{\#\text{ of } v\text{'s connections}}}.$$

This use case has been formulated as a link prediction task. We compared 3 different multi-hop sampling strategies in Table 6. PPR family strategies outperform weighted sampling by at least 2%. It is expected that multi-hop PPR sampling is slightly better than 2-hop PPR sampling but the improvement margin is relatively small.

Sampling Method	GNN Validation AUC Lift
2-hop Weighted Sampling	-
Multi-Hop PPR Sampling	+2.3%
2-hop PPR Sampling	+2.1%

Table 6: GNN sampling in People Recommendations

We choose 2-hop PPR sampling as the ultimate production sampling strategy due to the balance between metric enhancements and computational complexity. In particular, 2-hop PPR sampling contributes around 90% of gains and accelerate the sampling speed by 3 times. 500M members' embeddings are generated and they are integrated as new features in downstream models for Embedding-Based-Retrieval (EBR) tasks. A summary of the offline recall and online metrics is presented in Table 7. We observe significant offline and online metric lift after incorporating GNN embeddings with 2-hop PPR sampling.

6.5 Experiments in Ads

CTR prediction forms the cornerstone of LinkedIn's ads recommendation system, where GNN models are used to integrate graph topology into the prediction process. The ads graph includes member nodes, creative (ads) nodes, campaign nodes, and company nodes, connected by member interaction and creative attribute edges. However, the sparsity of the ads graph presents challenges. To mitigate this, we assumed members with similar tastes in Feed posts or similar connections might also share ad interaction patterns. Therefore, we added Feed affinity edges and member connection edges (with downsampling) to the graph, expanding it to billions of nodes and edges. In the post-GNN model training stage, we generated node embeddings and incorporated them into downstream CTR models. Offline metrics, summarized in Table 8, show the impact of different edge types and GNN embeddings on CTR prediction. The results indicate that adding member GNN embeddings alone improves AUC by 0.17%. The inclusion of Feed edges and member connection edges further increases AUC by 0.29%. The most effective CTR model, showing an 0.39% AUC lift, used all types of GNN embeddings, underscoring the synergy between member and item embeddings. When implemented online, this comprehensive CTR model, with member, creative, and campaign embeddings, achieved an approximate relative 2% online CTR lift. We are exploring boosting model performance without using Feed affinity and member connection. Graph densification becomes especially useful in this scenario for Ads. After adding artificial member-to-member edges via graph densification, we have seen an 0.28% AUC lift. Billions of cross-domain edges can be safely replaced with adding artificial edges between members.

Edges	Output GNN Embeddings	AUC lift
Ads	member	+0.17%
Ads with graph densification	member	+0.28%
Ads, Feed affinity, member connection	member	+0.29%
Ads, Feed affinity, member connection	member, creative, campaign	+0.39%

Table 8: Offline Metrics for Ads CTR Models with GNN embeddings

7 DEPLOYMENT LESSONS

Over the development of GNNs at LinkedIn we experimented with variety of applications and training infrastructures. Here we share some of our deployment lessons we learnt during the development.

7.1 Impression Discount Before Retrieval

In Follow Feed experiments, even though models exhibit high metric improvement in offline assessment. However, when we deployed these models online, the initial positive effects fade away and sometimes turned negative after a few days. As this behavior is model agnostic, we examined the system-wise reasons for this behavior. We discovered that the impression discount component, which filters out the updates that members have already viewed, is located after the retrieval layer. Under this setting, the impression discount component will continuously discard the relevant updates chosen from the retrieval layer, since the relevance scores are relatively stable. The previous retrieval model does not face this issue, as it retrieves the most recent updates, which are dynamic. Online metrics improve steadily after we position the impression discount component before the retrieval model.

7.2 Graph Engine scales up GNN training

GNN models rely on sampled neighbors, or the "compute graph," for training. Initially, Spark jobs were used to pre-compute these graphs and store them on HDFS, but this approach had significant drawbacks. First, the precomputation process, especially self-joins on large graphs, was slow, taking 20 hours for about 500M nodes, and couldn't scale for billions of nodes. Second, training speeds were hampered by heavy disk I/O, as the precomputed graph data

on HDFS was over 10 times larger than the original graph due to repeated nodes in the label data. Third, model iteration was sluggish since any change in neighbor sampling strategy required regenerating the compute graph. Additionally, the static nature of these precomputed graphs limited model generalization, affecting performance. To overcome these issues, we switched to using a Graph Engine for real-time compute graph sampling. This change eliminated the need for precomputation and addressed slow disk I/O by directly serving graph data from memory. With the graph data in GE, we can experiment with various sampling strategies or model architectures without altering the underlying graph, enhancing model iteration by 10X. Moreover, GE allows training jobs to dynamically request compute graphs in real-time for each training instance. Introducing randomness in sampling means compute graphs for the same node vary with each request, leading to better model performance through enhanced generalization.

8 CONCLUSION

In this paper we presented *LiGNN*, large scale GNN framework at LinkedIn. We shared set of approaches to train GNN model effectively reducing training speed by 7x, and improving quality of baseline GNN model by large margin. The lessons we share in the paper can be useful to industry practitioners. *LiGNN* has been deployed to variety of applications at LinkedIn including Feed, Jobs, people recommendation and Ads domains with significant production impact.

REFERENCES

[1] Alaa Awad, Denisa Roberts, Eden Dolev, Andrea Heyman, Zahra Ebrahimzadeh, Zoe Weil, Marcin Mejran, Vaibhav Malpani, and Mahir Yavuz. 2023. adSformers: Personalization from Short-Term Sequences and Diversity of Representations in Etsy Ads. *arXiv preprint arXiv:2302.01255* (2023).

[2] Aleksandar Bojchevski, Johannes Gasteiger, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling graph neural networks with approximate pagerank. In *KDD*.

[3] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. 2020. LANNS: a web-scale approximate nearest neighbor lookup system. *arXiv preprint arXiv:2010.09426* (2020).

[4] Paulius Micikevicius et al. 2018. Mixed precision training. https://doi.org/10.48550/arXiv.1710.03740

[5] Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. 2023. Relational Deep Learning: Graph Representation Learning on Relational Databases. arXiv:cs.LG/2312.04615

[6] Jonathan Halcrow, Alexandru Mosoi, Sam Ruth, and Bryan Perozzi. 2020. Grale: Designing Networks for Graph Learning. KDD.

[7] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. NIPS.

[8] Xiaotian Han, Tong Zhao, Yozen Liu, Xia Hu, and Neil Shah. 2023. MLPInit: Embarrassingly Simple GNN Training Acceleration with MLP Initialization. arXiv:cs.LG/2210.00102

[9] Bowen Hao, Hongzhi Yin, Jing Zhang, Cuiping Li, and Hong Chen. 2023. A Multi-Strategy-Based Pre-Training Method for Cold-Start Recommendation. *ACM Trans. Inf. Syst.* (2023).

[10] Bowen Hao, Jing Zhang, Hongzhi Yin, Cuiping Li, and Hong Chen. 2020. Pre-Training Graph Neural Networks for Cold-Start Users and Items Representation. arXiv:cs.IR/2012.07064

[11] Xu Keyulu, Hu Weihua, Leskovec Jure, and Jegelka Stefanie. 2019. How Powerful are Graph Neural Networks? ICLR.

[12] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. https://doi.org/10.48550/arXiv.1404.5997

[13] Kubernetes. 2014. an open-source system for automating deployment. https://kubernetes.io.

[14] Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Alexander Pritzel, Suman Ravuri, Timo Ewalds, Ferran Alet, Zach Eaton-Rosen, Weihua Hu, Alexander Merose, Stephan Hoyer, George Holland, Jacklynn Stott, Oriol Vinyals, Shakir Mohamed, and Peter Battaglia. 2022. GraphCast: Learning skillful medium-range global weather forecasting.

[15] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. arXiv:cs.LG/1903.12287

[16] Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. 2018. Generating Wikipedia by Summarizing Long Sequences. In *International Conference on Learning Representations*.

[17] Taichi Liu, Chen Gao, Zhenyu Wang, Dong Li, Jianye Hao, Depeng Jin, and Yong Li. 2023. Uncertainty-Aware Consistency Learning for Cold-Start Item Recommendation. In *SIGIR*.

[18] Yiqun Liu, Kaushik Rangadurai, Yunzhong He, Siddarth Malreddy, Xunlong Gui, Xiaoyi Liu, and Fedor Borisyuk. 2021. Que2Search: Fast and Accurate Query and Document Understanding for Search at Facebook. KDD.

[19] Zongtao Liu, Bin Ma, Quan Liu, Jian Xu, and Bo Zheng. 2021. Heterogeneous Graph Neural Networks for Large-Scale Bid Keyword Matching. CIKM.

[20] Brandon A. Mayer, Anton Tsitsulin, Hendrik Fichtenberger, Jonathan Halcrow, and Bryan Perozzi. 2023. HUGE: Huge Unsupervised Graph Embeddings with TPUs. In *KDD*.

[21] Nikil Pancha, Andrew Zhai, Jure Leskovec, and Charles Rosenberg. 2022. PinnerFormer: Sequence Modeling for User Representation at Pinterest. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3702–3712.

[22] Tieyun Qian, Yile Liang, Qing Li, and Hui Xiong. 2022. Attribute Graph Neural Networks for Strict Cold Start Recommendation. *IEEE Transactions on Knowledge and Data Engineering* (2022).

[23] Kaushik Rangadurai, Yiqun Liu, Siddarth Malreddy, Xiaoyi Liu, Piyush Maheshwari, Vishwanath Sangale, and Fedor Borisyuk. 2022. Nxtpost: User to post recommendations in facebook groups. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3792–3800.

[24] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637* (2020).

[25] Alex Samylkin. 2022. DeepGNN is a framework for training machine learning models on large scale graph data. https://github.com/microsoft/DeepGNN

[26] Aravind Sankar, Yozen Liu, Jun Yu, and Neil Shah. 2021. Graph Neural Networks for Friend Ranking in Large-Scale Social Platforms. KDD.

[27] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web*.

[28] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. 2021. Foundations and modeling of dynamic networks using dynamic graph neural networks: A survey. *IEEE Access* 9 (2021), 79143–79168.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[30] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. KDD.

[31] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Shuo Shang, and Ji-Rong Wen. 2018. TopPPR: Top-k Personalized PageRank Queries with Precision Guarantees on Large Graphs. SIGMOD.

[32] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[33] Da Zheng and Florian Saupe. 2023. Fast-track graph ML with GraphStorm: A new way to solve problems on enterprise-scale graphs. https://aws.amazon.com/blogs/machine-learning/fast-track-graph-ml-with-graphstorm-a-new-way-to-solve-problems-on-enterprise-scale-graphs/

A INFORMATION FOR REPRODUCIBILITY

A.1 Parameter selection for training speed up using Adaptive Neighbor Sampling and Grouping & Slicing

We provide additional information on parameters of Adaptive Neighbor Sampling and Grouping & Slicing strategies. From convergence plots on Figure 5 and Figure 6 we observe significant speed up of training due to Adaptive Neighbor Sampling and Grouping & Slicing techniques described in §4.2. We can see that convergence speed is improved in comparison to baseline. Adaptive Neighbor Sampling and Grouping & Slicing require parameters tuning to observe improvement. Here we set the parameter to group size equal 4 and update step equal 1 for the Follow Feed case, and same parameters perform well in Job Recommendations. For Adaptive neighbor sampling we start with neighborhood size of 2 neighbors and increase the number of neighbors sampled in stride of 20. Same parameters settings were used across applications for Adaptive neighbor sampling.

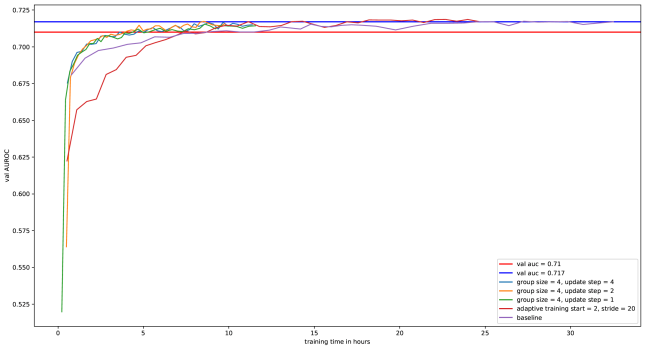

Figure 5: Follow Feed with Adaptive Neighbor Sampling and Grouping & Slicing

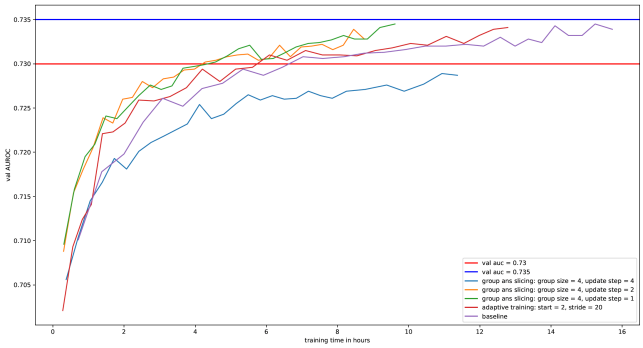

Figure 6: Job recommendation with Adaptive Neighbor Sampling and Grouping & Slicing

A.2 Nearline inference for GNNs

Here on Figure 7 we present details on nearline pipeline developed for Job Recommendations Engine. Due to deployment of the pipeline we could enable fresh job recommendations within LinkedIn.

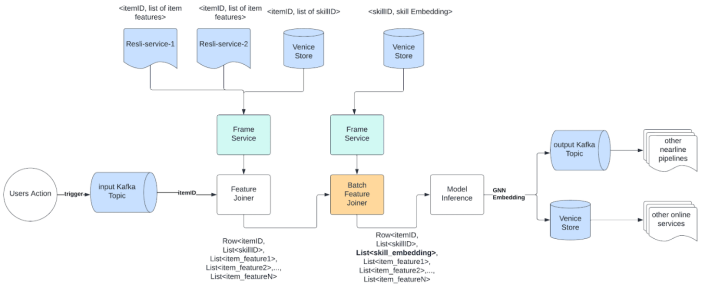

Figure 7: GNN nearline pipeline example