GraphStorm: All-in-one Graph Machine Learning Framework for Industry Applications

Da Zheng
Amazon AWS AI
Santa Clara, CA, USA
dzzhen@amazon.com

Xiang Song
Amazon AWS AI
Santa Clara, CA, USA
xiangsx@amazon.com

Qi Zhu
Amazon AWS AI
Santa Clara, CA, USA
qzhuamzn@amazon.com

Jian Zhang
Amazon AWS AI
Santa Clara, CA, USA
jamezhan@amazon.com

Theodore Vasiloudis
Amazon AWS AI
Santa Clara, CA, USA
thvasilo@amazon.com

Runjie Ma
Amazon AWS AI
Santa Clara, CA, USA
runjie@amazon.com

Houyu Zhang
Amazon Search AI
Seattle, WA, USA
zhanhouy@amazon.com

Zichen Wang
Amazon AWS AI
Santa Clara, CA, USA
zichewan@amazon.com

Soji Adeshina
Amazon AWS AI
Santa Clara, CA, USA
adesojia@amazon.com

Israt Nisa
Amazon AWS AI
Santa Clara, CA, USA
nisisrat@amazon.com

Alejandro Mottini
Amazon Search AI
Seattle, WA, USA
amottini@amazon.com

Qingjun Cui
Amazon Search AI
Palo Alto, CA, USA
qingjunc@amazon.com

Huzefa Rangwala
Amazon AWS AI
Santa Clara, CA, USA
rhuzefa@amazon.com

Belinda Zeng
Amazon SP
Seattle, WA, USA
zengb@amazon.com

Christos Faloutsos
Amazon AWS AI
Seattle, WA, USA
faloutso@amazon.com

George Karypis
Amazon AWS AI
Seattle, WA, USA
gkarypis@amazon.com

ABSTRACT

Graph machine learning (GML) is effective in many business applications. However, making GML easy to use and applicable to industry applications with massive datasets remain challenging. We developed GraphStorm, which provides an end-to-end solution for scalable graph construction, graph model training and inference. GraphStorm has the following desirable properties: (a) **Easy to use**: it can perform graph construction and model training and inference with just a single command; (b) **Expert-friendly**: GraphStorm contains many advanced GML modeling techniques to handle complex graph data and improve model performance; (c) **Scalable**: every component in GraphStorm can operate on graphs with billions of nodes and can scale model training and inference to different hardware without changing any code. GraphStorm has been used and deployed for over a dozen billion-scale industry applications after its release in May 2023. It is open-sourced in Github: https://github.com/awslabs/graphstorm.

CCS CONCEPTS

• **Information systems** → **Data mining**; **Computing platforms**.

KEYWORDS

Graph machine learning, Industry scale

ACM Reference Format:
Da Zheng, Xiang Song, Qi Zhu, Jian Zhang, Theodore Vasiloudis, Runjie Ma, Houyu Zhang, Zichen Wang, Soji Adeshina, Israt Nisa, Alejandro Mottini, Qingjun Cui, Huzefa Rangwala, Belinda Zeng, Christos Faloutsos, and George Karypis. 2024. GraphStorm: All-in-one Graph Machine Learning Framework for Industry Applications. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24), August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3637528.3671603

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

ACM ISBN 979-8-4007-0490-1/24/08
https://doi.org/10.1145/3637528.3671603

1 INTRODUCTION

Recent research has demonstrated the value of GML across a range of applications and domains, such as social networks and e-commerce. However, deploying such GML solutions to solve real business problems remains challenging for three reasons. First, industry graphs are massive, usually in the order of many millions or even billions of nodes and edges. Second, industry graphs are complex. They are usually heterogeneous with multiple node types and edge types. Some nodes and edges are associated with diverse features, such as numerical, categorical and text/image features, while some other nodes or edges have no features. Third, many applications do not store data in a graph format. To apply GML to these data, we need to first construct a graph. Defining a graph schema is part of graph modeling and often requires multiple rounds of trials and errors.

Today users need to manually develop complex code with libraries such as DGL [21] or Pytorch-Geometric [7]. This usually requires users to have significant modeling expertise in GML. Furthermore, even if a user has the required expertise, it requires

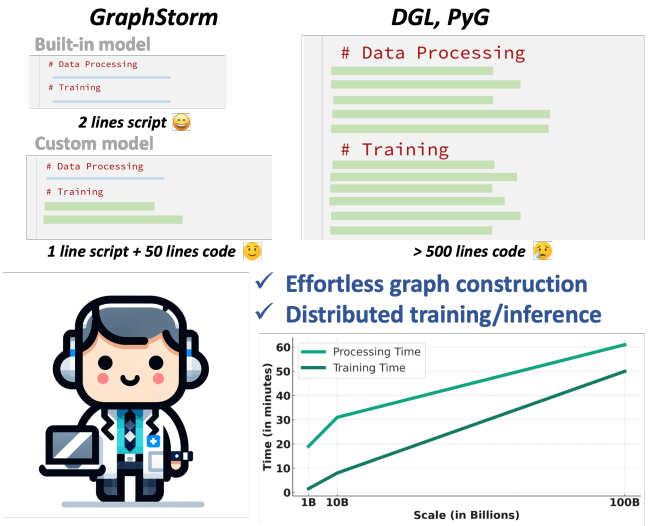

Figure 1: Easy and Scalable GML with GraphStorm.

significant efforts to develop techniques to achieve good performance and scale model training and inference to large graphs on a cluster of CPU/GPU machines. These challenges prevent GML from becoming a commonly used technique in industry.

To address these challenges, we developed GraphStorm, which is a no-code/low-code framework for scientists in the industry to develop, train, and deploy GML models for solving real business problems. It provides end-to-end pipelines for graph construction, model training and deployment to simplify every step of developing GML models. It balances the usability, flexibility, modeling capability, training efficiency and scalability so that users who are not familiar with GML can easily apply it to their applications. As illustrated in Figure 1, GraphStorm allows users to develop GML models and scale them to industry graphs with billions of edges by using a few lines of code. To scale to graphs with hundreds of billions of edges, GraphStorm is built on top of the distributed GNN system DistDGL [28], adopts scalable algorithms and provides efficient implementations for these algorithms. To help users quickly prototype a GML model, GraphStorm provides a command line interface that allows users to run model training and inference with a single command. Once a GML model prototype is ready, GraphStorm provides two options to help users further improve model performance. It provides many built-in modeling techniques to improve performance; it supports custom model API to allow users to develop their own model implementations. GraphStorm allows model training/inference to run on a single machine and scale it to a cluster of machines without changing any line of code.

We evaluate the scalability and modeling capability of GraphStorm with two public graph data with hundreds of millions of nodes and synthetic graph data ranging from 1 billion edges to 100 billion edges. GraphStorm is scalable and efficient in processing and modeling graphs with complex features. It finishes every step needed for training a GML model on billion-scale graphs, such as MAG and Amazon Review within hours. Even on a synthetic graph with 100 billion edges, we can process the data and train a GNN model with a few hours. We further show that GraphStorm provides many practical and effective techniques to help users improve graph model performance in the benchmark datasets. For example, GraphStorm allows users to freely try out different graph schemas and a proper graph schema can improve model performance by up to 15% on our benchmark datasets without changing the model architecture and model training procedure; BERT+GNN fine-tuning can improve model performance on text-rich graphs by up to 17.6% in our benchmarks; GNN distillation can improve the performance of a BERT model by 8%. The scalability and practical modeling capablities are essential for industry production. We have successfully developed multiple GML models with GraphStorm that outperform production models and are deployed in production.

Our contributions are summarized as below:

- **Easy to use**: GraphStorm provides users an easy solution from graph construction to GML model training and inference with a single command line for real-world applications.
- **Expert-friendly**: GraphStorm provides many built-in modeling techniques to help scientists improve model performance in complex industry graphs without writing any code. It also provides a custom model API for users to develop and train their own models.
- **Scalable**: GraphStorm ensures that every step for GML model development and deployment can scale to graphs with billions of nodes and hundreds of billions of edges and can scale model training and inference in different hardware (e.g., from single GPU to multi-machine multi-GPU) without any code modification.
- **In production**: GraphStorm has been deployed in production on billion-scale graphs, for over 6 months.

2 RELATED WORKS

There are many existing GML frameworks. DGL [21] and Pytorch-Geometric [7] are two popular frameworks that provide low-level API for writing graph neural network models (GNNs). Users need to write complex code to address many practical problems in graph applications. On top of these low-level GML frameworks, people developed higher-level frameworks specialized for one type of GML problems. For example, DGL-KE [29] and Pytorch-BigGraph [14] contain a set of knowledge graph embedding models and provide scalable training solutions. TGL [31] provides a set of models for continuous-time temporal graphs and Pytorch-Geometric Temporal [16] contains a set of models for spatiotemporal models. These high-level frameworks provide command-line interfaces to train models without writing code, which simplifies model development.

AliGraph [33], Euler [1], AGL [25], PGL [2] and TensorFlow-GNN [6] are industry GNN frameworks. They are built for distributed training and inference on large heterogeneous graphs. AliGraph, Euler and TensorFlow-GNN adds GNN-related functionalities to TensorFlow, such as graph data structure, message passing operations, mini-batch sampling on graph data. AGL is built on top of MapReduce for distributed training and inference of GNN. PGL is a GNN framework built on top of Paddle-Paddle [2]. One of the key design choices is how to perform mini-batch sampling for GNN training. TensorFlow-GNN and AGL choose the option of preprocessing the input graph to generate mini-batch graphs and save

them to disks, while AliGraph, Euler and PGL choose on-the-fly sampling. The main benefit of on-the-fly sampling is that it allows users to easily change some hyperparameters of GNN, such as the number of GNN layers and fanout, to tune model performance. GraphStorm is an industry GML framework built on top of DGL and provides high-level GML functionalities, such as end-to-end training/inference pipelines and advanced GML techniques. To support fast prototyping, GraphStorm adopts on-the-fly sampling.

CogDL [4] is a high-level GML library that provides many built-in GML methods and benchmark datasets. CogDL is built for researchers to reproduce model performance and set up standard benchmarks for comparing the performance of different models. GraphStorm is built for scientists to apply GML to industry applications. Therefore, it does not contain built-in benchmark datasets but provides an easy way of loading tabular data into GraphStorm for model training and inference. GraphStorm is much more scalable than CogDL.

3 DESIGN

GraphStorm is designed as a general framework to accelerate GML adoption in the industry. It provides functionalities to help every step in the journey of developing and deploying GML models in the industry, including graph construction, model prototyping, model tuning and model deployment. Typically, graph modeling starts by defining the right graph schema. GraphStorm accepts data in tabular format and provides a tool to construct a graph based on the graph schema defined by users. After graph construction, a user applies GML methods to the data to train an initial model. For model development, GraphStorm provides end-to-end pipelines to train a model with a single command line. To improve model performance, GraphStorm provides many built-in techniques to tackle common problems in industry graph applications; for more advanced users, GraphStorm provides custom model APIs for trying new modeling ideas. Once a model meets the deployment criteria, GraphStorm allows users to deploy the model without any code modification on the ML service platforms.

To cover a large number of industry applications, GraphStorm provides modularized implementations to support diverse graph data and applications. Figure 2 shows the abstractions required for model training, including datasets, models, data loaders, tasks, training algorithms and evaluations. GraphStorm currently supports four types of graph data: homogeneous graphs, heterogeneous graphs, temporal graphs and textual graphs. For different types of graph data, GraphStorm provides different modeling techniques, such as RGCN [18] and HGT [9] for heterogeneous graphs and TGAT [5] for temporal graphs. GraphStorm provides task-specific data loaders, such as node data loaders for node-level tasks, edge data loaders for edge-level tasks. GraphStorm explicitly provides separate data loaders for predicting an attribute of an edge (*EdgeDataLoader*) and for predicting the existence of an edge (*LinkPredictionDataLoader*) for the sake of efficiency because *LinkPredictionDataLoader* not only samples positive edges from a graph but also needs to construct negative edges for efficient training (see Section 3.3.4). GraphStorm supports different training strategies, such as single-task training, multi-task training and LM-GNN co-training. GraphStorm currently supports seven graph tasks and provides corresponding evaluation metrics for model evaluation. By combining different components, we can have a complete solution for a graph application. For example, as illustrated in Figure 2, if we need to perform a link prediction task on a heterogeneous graph with rich text features, we can take *RGAT* as a graph model, use *LinkPredictionDataLoader* as the data loader and *LM+GNN co-training* as the training algorithm to train a model for *link prediction*. We can use *MRR* for model evaluation.

3.1 Architecture

To realize the modularized implementations, GraphStorm is designed with four layers: the distributed graph engine, pipelines for graph construction, training and inference, a general model zoo and service integration, as illustrated in Figure 3. GraphStorm ensures that every layer is scalable in order to achieve scalability.

3.1.1 Distributed graph engine. The first layer is the distributed graph engine built on top of DistDGL [28], which enables distributed training and inference on billion-scale graphs. It provides the abstractions of datasets and data loaders to help users access different types of graph data (e.g. homogeneous graphs and heterogeneous graphs) stored in different hardware (e.g., in CPU memory or distributed memory). It provides distributed tensors and distributed embedding layers to help users store and access node/edge features and learnable node embeddings in a cluster of machines. The distributed graph engine provides the same interface on different hardware to hide the complexity of handling different hardware. GraphStorm adopts on-the-fly mini-batch sampling on a distributed graph. The benefit of on-the-fly sampling is to allow sampling mini-batch data with different configurations without preprocessing the input data again. This is particularly useful for tuning hyperparameters, such as the batch size and the number of GNN layers.

3.1.2 Graph construction pipeline. Enterprise data are usually stored in RDBMS and NoSQL data stores. None of them can be directly used by GML frameworks to conduct ML tasks. GraphStorm provides a graph processing pipeline that takes tabular data as input and constructs a graph for model training and inference. GraphStorm provides a single-machine implementation to enable easly model prototyping and a distributed Spark-based [24] implementation for scalable deployment. Both implementations take the input data and output graphs in the same format, making it easy for users to move from small-scale experiments to large-scale deployments.

The graph construction pipeline takes multiple steps to transform data and eventually converts them into a graph format. The first step is feature transformation. To stay within a single library for data processing, model training and inference, GraphStorm provides implementations of many feature encoding methods that accept numerical, categorical and text data and apply appropriate transformations on the features at a billion scale in a distributed manner. If the input data use strings to identify nodes and edges, GraphStorm also converts them into integers because GraphStorm model training and inference requires nodes and edges are identified by integers. GraphStorm provides a distributed ID mapping mechanism that creates massive mapping tables from strings to integers and applies them to all string IDs on nodes and edges. After

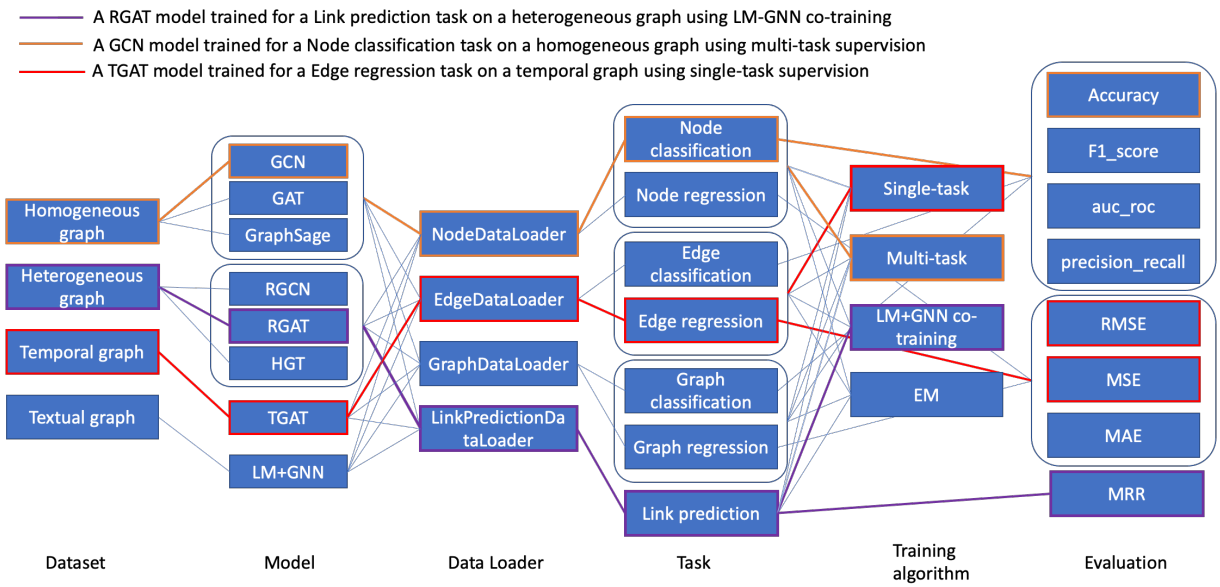

Figure 2: The functionalities of GraphStorm. The colored lines show examples of constructing a complete model solution in GraphStorm.

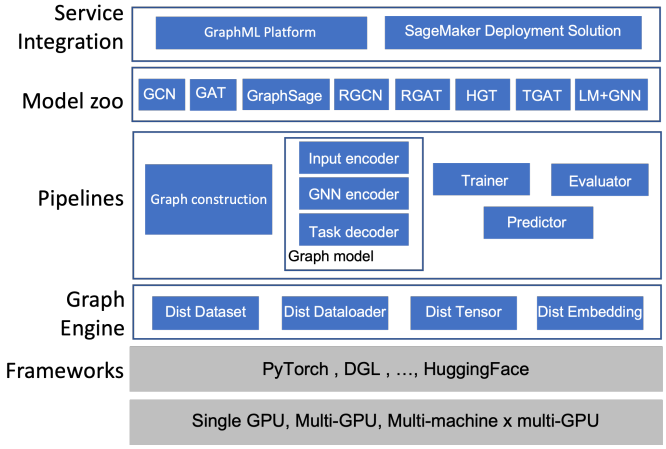

Figure 3: GraphStorm architecture

feature transformation and ID mapping, GraphStorm provides a distributed graph partitioning framework with various edge-cut algorithms, such as random and METIS partitioning [11, 12], which assigns nodes and edges to partitions. The graph partititon algorithms in GraphStorm support massive graphs with billions of nodes and edges. The implementation of graph partitioning algorithm is decoupled from other components, making it straightforward to introduce new distributed partitioning algorithms. After the graph partition algorithm, the pipeline shuffles node data and edge data in a distributed manner, constructs graph partition objects on each machine in parallel and saves them in the DistDGL format, which is used by GraphStorm for model training and inference.

3.1.3 Training and inference pipeline. The training/inference pipeline exposes GML model templates and supports different graph tasks (node-level predictions, edge-level predictions and graph-level predictions). GraphStorm splits a GML model into three components: node/edge input encoders, graph encoders and task decoders. The node/edge input encoders handle node/edge features (e.g., compute BERT embeddings from text features or read learnable embeddings on featureless nodes). The graph encoders use GNN to encode the graph structure and node/edge features together to generate embeddings of nodes. The task decoder is usually designed specifically for a task. The pipelines provide trainers, predictors, and evaluators to train, infer and evaluate a task-specific GML model that is implemented with the graph model interface (shown in Section 3.2.2). Because the models in the GraphStorm model zoo are implemented with the same model interface, the same pipelines are used to train/infer/evaluate GraphStorm's built-in models as well as users' custom models.

3.1.4 Model zoo. To lower the effort of developing GML models and improving performance, GraphStorm provides a model zoo with a diverse set of GML model implementations. This includes RGCN [18], RGAT [3] and HGT [9] for heterogeneous graphs, GCN [13], GAT [20] and GraphSage [8] for homogeneous graphs, TGAT [5] for temporal graphs, LM+GNN [10] for text-rich graphs. These model implementations can be used with different task decoders to handle different graph tasks.

3.2 User interface

GraphStorm is designed to lower the bar of developing GML models for beginner users as well as supporting advanced GML scientists to experiment with new methods on industry-scale graph data. To support these diverse requirements, GraphStorm provides two types of interfaces. The command line interface allows quick prototyping

of a GML model on application data and model performance tuning with builtin GML techniques in GraphStorm. The programming interface allows advanced users to develop their custom models to further improve model performance.

3.2.1 Command line interface. GraphStorm provides the command-line interface for graph construction, model training, performance tuning and model inference. For graph construction, GraphStorm provides *gconstruct.construct_graph*, which takes the application data stored in tabular format (e.g., CSV and Parquet) and constructs a graph with the GraphStorm format for training and inference based on the graph schema defined by users. For model development, GraphStorm provides a module for each graph task. For example, GraphStorm provides *run.gs_node_classification* to train and run inference of node classification models. GraphStorm allows users to configure model training and inference to enable many built-in techniques (shown in Section 3.3) by using the command arguments. Appendix B shows examples of GraphStorm commands.

3.2.2 Programming interface. GraphStorm provides API for developing custom models to further improve model performance. GraphStorm provides task-specific trainers, predictors, evaluators as well as model templates. Figure 4 shows a GraphStorm training script to train an RGCN node classification model for illustration. At a minimum, a user only needs 8 lines of code in the training script to train and evaluate a model. Appendix C shows an example of defining a graph model with GraphStorm's custom model API. With GraphStorm's API, a user only needs to focus on model development and GraphStorm scales the model training and inference because the same training script can run on a single GPU, multiple GPUs and multiple machines without modifying any code. This greatly reduces the effort of prototyping models on the subset of application data and scaling the model training and deployment on the actual data.

```
import graphstorm as gs

gs.initialize()
data = gs.dataloading.GSgnnData(part_config_file,
        node_feat_field='feat',
        label_field='label')
model = RGCNNCModel(train_data.g,
        num_gnn_layers=2, num_hidden=128, num_classes=2)
evaluator = gs.eval.GSgnnAccEvaluator(multilabel=False)
dataloader = gs.dataloading.GSgnnNodeDataLoader(data,
        train_idxs, fanout=[5, 5], batch_size=1024)
val_dataloader = gs.dataloading.GSgnnNodeDataLoader(data,
        eval_idxs, fanout=[5, 5], batch_size=1024)
trainer = gs.trainer.GSgnnNodeTrainer(model,
        eval=evaluator)
trainer.fit(train_dataloader=dataloader,
        val_dataloader=val_dataloader,
        num_epochs=10)
```

Figure 4: GraphStorm training script for node classification.

3.3 Modeling techniques

Industry graphs are usually heterogeneous with multiple node types and edge types. There are many modeling challenges associated with them. Below are a few common problems that we encounter when modeling industry graphs.

- Some nodes/edges are associated with multi-modal features, such as text and images. How to jointly model text/image data with graph data together?
- Some node types do not have node features. How to model these nodes efficiently and effectively?
- It is common that some nodes in a dataset have no neighbors or few neighbors. How to improve the model performance on these isolated nodes with graph modeling?
- Link prediction is a common task in industry applications. However, the training set of link prediction is usually large (billions of edges). How to effectively and efficiently train a GML model with link prediction on a billion-scale graph?

GraphStorm provides built-in techniques to address these modeling issues to improve model performance. GraphStorm only adopts methods that can scale to billion-scale graphs and also ensures that their implementations are scalable.

3.3.1 Joint modeling text and graph data. Many industry graph data have rich text features on nodes and edges. For example, the Amazon search graph has rich text features on queries (keywords) and products (product descriptions) [22]. This requires jointly modeling graph and text data together. By default, GraphStorm runs language models (LM), such as BERT, on the text features and runs a GNN model on the graph structure in a cascading manner.

GraphStorm implements some efficient methods to train LM models and GNN models for downstream tasks [10, 27]. For example, Ioannidis et al. [10] devises a three-stage training method that first performs graph-aware fine-tuning of LM, trains the GNN model with the fixed LM and follows with the end-to-end fine-tuning; GLEM [27] trains the LM model and the GNN model iteratively on the downstream application. The original GELM paper was designed for homogeneous graphs and GraphStorm extends it to heterogeneous graphs.

3.3.2 Efficient training large embedding tables for featureless nodes. In an industry graph, some node types do not have any node features. For example, the Microsoft academic graph (MAG) does not have node features on author nodes. When training a GNN model on the graph, GraphStorm by default adds learnable embeddings on author nodes, which results in a massive learnable embedding table. There are two problems of using learnable embeddings for featureless nodes. First, the learnable embedding table is usually massive because a graph can have many featureless nodes (e.g., there are 200 million authors in the MAG graph). Secondly, adding learnable embeddings significantly increases the model size, which causes overfitting.

GraphStorm provides methods that construct node features of the featureless nodes with the neighbors that have features.

$$F'_{v_i} = f(F_{v_j}, \forall j \in N_i) \tag{1}$$

With the constructed node features, a user can use a normal GNN model to generate GNN embeddings. We can have different choices

for f. We can use non-learnable function (e.g., average) or learnable functions (e.g., Transformer) to construct node features.

3.3.3 GNN distillation for isolated nodes. In many industry graphs, there exist nodes that do not have neighbors, known as *isolated* nodes. For example, in e-commerce applications, a search graph is constructed from customer logs over some period of time (e.g., purchases and clicks connecting the search queries and the products) for model training. When the model is deployed, it is common that new products and queries are added. These new nodes are not covered by the graph during model training and sometimes do not connect with any nodes in the graph. Simply applying the trained GNN model on these isolated nodes is not effective. To address this issue, GraphStorm provides the capability of distilling a GNN model into another model without graph dependency (e.g., MLP or DistilBERT [17]) [26, 30]. GraphStorm provides multiple options for distillation (e.g., using soft labels or using embeddings directly).

3.3.4 Optimizations for link prediction. Link prediction is widely used in the industry as a pre-training method to produce high-quality entity representations. However, performing link prediction training on large graphs is non-trivial both in terms of model performance and efficiency. GraphStorm provides many techniques to optimize link prediction.

GraphStorm incorporates three ways of improving the model performance of link prediction. Firstly, to avoid information leak in model training, GraphStorm by default excludes validation and testing edges from training graphs in the model training [32]. GraphStorm also excludes training target edges from message passing to avoid model overfitting. Secondly, to better handle heterogeneous graphs, GraphStorm provides two ways to compute link prediction scores: dot product and DistMult [23]. Dot product works when there is only one training edge type, while DistMult works when there are multiple training edge types. Thirdly, GraphStorm provides two options to compute training losses, i.e., cross entropy loss, and contrastive loss. The cross entropy loss turns a link prediction task into a binary classification task. The contrastive loss compels the representations of connected nodes to be similar while forcing the representations of disconnected nodes to be dissimilar. Contrastive loss is more stable to different numbers of negative samples and converges faster. In contrast, cross entropy loss only works with small number of negative samples and requires more time to converge. However, cross entropy loss allows users to define the importance, i.e., a weight, for each positive edge, providing extra flexibility when modeling positive samples.

Link prediction requires constructing numerous negative edges to train a model. In distributed training, constructing negative edges naively may result in accessing a large number of nodes from remote partitions. This would cause a large amount of data movement between machines and slow training speed. GraphStorm provides multiple negative sampling methods, such as uniform negative sampling, joint negative sampling and in-batch negative sampling, to trade-off training efficiency and model performance. For example, uniform sampling may allow the best model performance but requires a mini-batch to involve numerous nodes from remote partitions, which results in slow training speed; in-batch negative sampling only uses the nodes within the mini-batch to construct negative edges, which minimize the number of nodes in a mini-batch to speed up training, but may have some model performance degradation. GraphStorm provides all of these options to give users more options to train their models (see details in Appendix A).

4 EVALUATION

4.1 Datasets

We evaluate GraphStorm with two large heterogeneous graphs: MAG [19] and Amazon review [15] (shown in Table 1). We construct two tasks on the benchmark datasets. For node classification, we predict the paper venue on the MAG data and predict the brand of a product on the Amazon review dataset; for link prediction, we predict what papers a paper cites on the MAG dataset and predict which products are co-purchased with a given product on the Amazon review dataset.

4.2 GML on Industrial-Scale Graphs

We benchmark two main methods (pre-trained BERT+GNN and fine-tuned BERT+GNN) in GraphStorm on the two large datasets. For pre-trained BERT+GNN, we first use the pre-trained BERT model to compute BERT embeddings from node text features and train a GNN model for prediction. The alternative solution is to fine-tune the BERT models on the graph data before using the fine-tuned BERT model to compute BERT embeddings and train GNN models for prediction. We adopt different ways to fine-tune the BERT models for different tasks. For node classification, we fine-tune the BERT model on the training set with the node classification tasks; for link prediction, we fine-tune the BERT model with the link prediction tasks.

Table 2 shows the overall model performance of the two methods and the overall computation time of the whole pipeline starting from data processing and graph construction. In this experiment, we use 8 r5.24xlarge instances for data processing and use 4 g5.48xlarge instances for model training and inference. Overall, GraphStorm enables efficient graph construction and model training on large text-rich graphs with hundreds of millions of nodes. We can see that for most cases, GraphStorm can finish all steps, even the expensive BERT computations, within a few hours. The only exception is fine-tuning BERT model with link prediction and training a GNN link prediction model on the MAG dataset, which take 2-3 days. The reason is that the training set of the dataset for link prediction has billions of edges and we fine-tune BERT and train GNN model on the entire training set. For pre-trained BERT+GNN, it takes 3.5 hours to compute BERT embeddings on 240 million paper nodes in MAG and takes 8 hours on 240 million nodes in the Amazon review data. To improve model performance, we fine-tune the BERT model on the training set of the downstream tasks, which leads to a significant performance boost. Its performance improvement is also significant. For example, fine-tuning BERT on MAG can improve BERT+GNN by 11% for node classification and by 40% for link prediction.

We also benchmark GraphStorm on large synthetic graphs to further illustrate its scalability. We generate three synthetic graphs with one billion, 10 billion and 100 billion edges respectively. The average degree of each graph is 100 and the node feature dimension is 64. We train a GCN model for node classification on each

Table 1: The statistics of benchmark datasets.

Dataset	#nodes	#edges	#node/edge types	NC training set	LP training set	text-feature nodes
Amazon Review	286,462,374	1,053,940,310	3/4	1,826,784	25,984,064	242,967,461
MAG	484,511,504	7,520,311,838	4/4	28,679,392	1,313,781,772	240,955,156

Table 2: The overall performance and computation time of GraphStorm.

Dataset	Task	Data process	Target	Pre-trained BERT + GNN			Fine-tuned BERT + GNN		
				LM Time Cost	Epoch Duration	Metric	LM Time Cost	Epoch Duration	Metric
MAG	NC	9:13:00	venue	3:26:00	2:14:33	Acc:0.5715	23:42:35	2:16:42	Acc: 0.6333
	LP		cite	3:18:19	36:34:52	Mrr: 0.4873	75:08:22	36:12:17	Mrr: 0.6841
AR	NC	3:35:00	brand	8:03:23	0:02:47	Acc: 0.8407	8:09:54	0:02:48	Acc: 0.8963
	LP		co-purchase	7:28:12	0:44:12	MRR: 0.9602	7:42:18	0:33:19	MRR: 0.9710

Table 3: Scalability of GraphStorm on synthetic graphs with 1B, 10B and 100B edges. The corresponding training set sizes are 8 million, 80 million and 800 million, respectively.

Graph	Data pre-process		Graph Partition		Model Training	
Size	# instances	Time	# instances	Time	# instances	Time
1B	4	19 min	8	8 min	8	1.5 min
10B	8	31 min	16	41 min	16	8 min
100B	16	61 min	32	416 min	32	50 min

Table 4: Performance on Amazon Review Graph varying graph schemas.

Schema	node types	featureless	LP	NC
Homogenous	item	No	MRR:0.937	Acc:0.640
Heterogenous-v1	+review	No	MRR:0.944	Acc:0.742
Heterogenous-v2	+customer	"customer"	MRR:0.960	Acc:0.725

graph. In this experiment, we use r5.24xlarges instances for data pre-processing, graph partition and model training. For graph partition, we use a random partition instead of METIS partition. For model training, we take 80% of nodes as training nodes and run the training for 10 epochs. Table 3 shows the computation time of graph pre-processing, graph partition and model training. Overall, GraphStorm enables graph construction and model training on 100 billion scale graphs within hours and shows good scalability. For data pre-processing, when increasing the graph size from 1 billion to 100 billion edges, the overall computation cost (measured by instance-minutes) only increases by 13×. For model training, when increasing both the graph size and training set by 100X, the overall training cost only increases by 133×. For graph partition, when increasing the graph size by 100×, the overall computation cost increases by 208×.

4.3 Modeling graph data from data preprocessing

GML leverages the inductive bias inherent in graph structures, making the graph schema crucial for the successful application of GML. GraphStorm offers an efficient graph construction pipeline to power scientists to experiment GML modeling starting from the graph construction stage. We use the Amazon Review graph to illustrate the importance of graph schema. We gradually increase the heterogeneity of the graph schema and observe the corresponding variations in model performance. As illustrated in Table 4, adding review nodes and the (item, *receives*, review) edge enhances performance in both co-purchase prediction and brand classification tasks. Moreover, incorporating featureless customer nodes and the (customer, *writes*, review) edge further improves co-purchase prediction performance, but not node classification. This improvement is likely because items reviewed by the same customer have a higher likelihood of being purchased together, although this factor does not significantly influence the determination of an item's brand. This indicates the importance of defining the right graph schema to improve performance and that users need to try out different graph schemas to determine the best schema for a given task.

4.4 Performance of GraphStorm techniques

Besides the aforementioned GNN and LM+GNN models, GraphStorm supports many techniques to improve model performance on industry graph data. In this section, we evaluate some of the techniques described in Section 3.3 on our benchmark datasets.

4.4.1 Joint text and graph modeling. We benchmark GraphStorm's capability of jointly modeling text and graph data. We use different methods to train BERT and GNN on the full MAG data to illustrate the effectiveness of different methods (Figure 5). Here we compare four different methods: fine-tune BERT to predict the venue, train GNN with BERT embeddings from pre-trained BERT model (pre-trained BERT+GNN), train GNN with BERT embeddings generated from a BERT model fine-tuned with link prediction (FTLP BERT+GNN), train GNN with BERT embeddings generated from a BERT model fine-tuned with venue prediction (FTNC BERT+GNN). First, BERT+GNN is much more effective in predicting paper venues than BERT alone by up to 54%. It also shows the importance of BERT fine-tuning in the prediction task. Even though fine-tuning BERT with link prediction can improve the prediction accuracy (by up to 7.6% over pre-trained BERT+GNN), the best accuracy is

Table 5: Performance of GNN Embeddings Distillation on MAG Dataset.

Settings	Acc
DistilBERT fine-tuned with venue labels (768 dim)	41.17%
DistilBERT with GNN distillation (128 dim)	44.53%

achieved by first fine-tuning the BERT model with venue prediction and training GNN with the fine-tuned BERT model with venue prediction (by 17.6%).

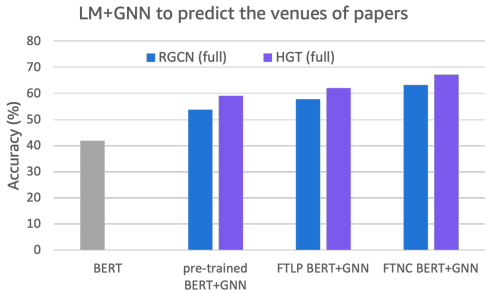

Figure 5: Jointly modeling text and graph data on Microsoft Academic Graphs

4.4.2 GNN distillation. We evaluate the GNN distillation capability in GraphStorm. In this experiment, we train a GNN model to predict venues of papers in MAG and distill it to a language model; we compare this distilled model with a language model directly fine-tuned to predict venues. We choose pre-trained 70MM HuggingFace DistilBERT [17] for the BERT model architecture and use its initial weights for both the baseline and the distilled model. For the baseline, we fine-tune the DistilBERT using the venue labels. For the distilled model, we conduct the distillation to minimize the distance between the embeddings from a GNN teacher model and the embeddings from a DistilBERT student model. MSE loss is used to supervise the training. After the model training, we use them to generate embeddings on paper nodes separately. We evaluate the performance of the generated embeddings by training separate MLP decoders for the embeddings from baseline DistlBERT and GNN-distilled DistilBERT. As shown in Table 5, the GNN-distilled DistilBERT embeddings outperform the baseline DistilBERT embeddings by 8.2%, demonstrating the structure knowledge from the GNN teacher model are transferred to the GNN-distilled DistilBERT.

4.4.3 Link prediction. We benchmark GraphStorm's capability of training link prediction models on Amazon Review dataset. We conduct link prediction on the (item, also-buy, item) edges. We compare the model performance and training time with different loss functions and negative sampling settings. We experiment both contrastive loss and cross entropy loss. The negative sampling methods include in-batch negative sampling, joint negative sampling with the number of negative edges of 1024, 32 and 4 (denoted as joint-1024, joint-32 and joint-4, respectively), and uniform negative sampling with the number of negative edges of 1024 and 32 (denoted as uniform-1024 and uniform-32, respectively). We set the local batch size to 1024 and the maximum training epochs to 20.

Table 6: Performance of link prediction on Amazon Review Graph with varying training techniques.

Loss func	Neg-Sample	epoch time	#epochs	Metric
contrastive	in-batch	1340.90s	5	MRR:0.951
contrastive	joint-1024	1344.65s	8	MRR:0.956
contrastive	joint-32	1286.64s	8	MRR:0.958
contrastive	joint-4	1289.9s	10	MRR:0.956
contrastive	uniform-32	1726.19s	8	MRR: 0.957
contrastive	uniform-1024		OOM	
cross-entropy	in-batch	1343.94s	20	MRR: 0.250
cross-entropy	joint-1024	1330.50s	18	MRR:0.334
cross-entropy	joint-32	1290.72s	15	MRR:0.380
cross-entropy	joint-4	1288.53s	16	MRR:0.645
cross-entropy	uniform-32	1746.68s	20	MRR:0.377
cross-entropy	uniform-1024		OOM	

Table 6 shows the result. Overall, the performance of contrastive loss is much better than cross entropy loss. Contrastive loss is more robust to the variance of the number of negative edges. Cross entropy loss works much better when the number of negative edges is small, e.g., 4, but its performance decreases when the number of negative edges is larger. Furthermore, contrastive loss converges much faster than cross entropy loss. In general, uniform negative sampling is more computationally expensive and consumes more GPU memory than the other two methods because it samples many more negative nodes. For example, with batch size of 1024, both in-batch and joint-32 samples 1024 nodes to construct negative edges, while uniform-32 has to sample 32,768 nodes. So the epoch time of uniform sampling is larger than the other two.

5 DISCUSSION AND CONCLUSIONS

GraphStorm is a general no-code/low-code GML framework designed for industry applications. It provides end-to-end pipelines for graph construction, model training and inference for many different graph tasks and scales to industry graphs with billions of nodes efficiently. This helps scientists develop GML models starting from graph schema definition and GML model prototyping on large industry-scale graphs without writing code. Based on our experience working with scientists in the industry, this significantly reduces the overhead of developing a new GML model and tuning its performance for industry applications. GraphStorm provides many advanced GML techniques to handle problems commonly encountered in industry applications and we have fully verified their effectiveness and scalability on public and private datasets. With GraphStorm, scientists can quickly prototype GML models and use built-in techniques to improve performance and outperform production models. Users can also develop their own custom models in GraphStorm and scale their model training to industry-scale graphs with a cluster of machines. Currently, GraphStorm has been used to develop and deploy GML models for over a dozen industry applications.

REFERENCES

[1] Euler github. https://github.com/alibaba/euler, September 2020.

[2] Pgl github. https://github.com/PaddlePaddle/PGL, September 2020.

[3] D. Busbridge, D. Sherburn, P. Cavallo, and N. Y. Hammerla. Relational graph attention networks. 2019.

[4] Y. Cen, Z. Hou, Y. Wang, Q. Chen, Y. Luo, Z. Yu, H. Zhang, X. Yao, A. Zeng, S. Guo, Y. Dong, Y. Yang, P. Zhang, G. Dai, Y. Wang, C. Zhou, H. Yang, and J. Tang. Cogdl: A comprehensive library for graph deep learning. In Proceedings of the ACM Web Conference 2023 (WWW'23), 2023.

[5] da Xu, chuanwei ruan, evren korpeoglu, sushant kumar, and kannan achan. Inductive representation learning on temporal graphs. In International Conference on Learning Representations (ICLR), 2020.

[6] O. Ferludin, A. Eigenwillig, M. Blais, D. Zelle, J. Pfeifer, A. Sanchez-Gonzalez, W. L. S. Li, S. Abu-El-Haija, P. Battaglia, N. Bulut, J. Halcrow, F. M. G. de Almeida, P. Gonnet, L. Jiang, P. Kothari, S. Lattanzi, A. Linhares, B. Mayer, V. Mirrokni, J. Palowitch, M. Paradkar, J. She, A. Tsitsulin, K. Villela, L. Wang, D. Wong, and B. Perozzi. TF-GNN: graph neural networks in tensorflow. CoRR, abs/2207.03522, 2023.

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

[8] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. Advances in neural information processing systems, 30, 2017.

[9] Z. Hu, Y. Dong, K. Wang, and Y. Sun. Heterogeneous graph transformer. In WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020, pages 2704–2710. ACM / IW3C2, 2020.

[10] V. N. Ioannidis, X. Song, D. Zheng, H. Zhang, J. Ma, Y. Xu, B. Zeng, T. Chilimbi, and G. Karypis. Efficient and effective training of language and graph neural network models, 2022.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput., 20(1):359–392, jan 1998.

[12] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. Technical report, Department of Computer Science, University of Minnesota, 2011.

[13] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 2016.

[14] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. PyTorch-BigGraph: A Large-scale Graph Embedding System. In Proceedings of the 2nd SysML Conference, Palo Alto, CA, USA, 2019.

[15] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel. Image-based recommendations on styles and substitutes. In Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval, pages 43–52, 2015.

[16] B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, G. Lopez, N. Collignon, and R. Sarkar. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In Proceedings of the 30th ACM International Conference on Information and Knowledge Management, page 4564–4573, 2021.

[17] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108, 2019.

[18] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks, 2017.

[19] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. Hsu, and K. Wang. An overview of microsoft academic service (mas) and applications. In Proceedings of the 24th international conference on world wide web, pages 243–246, 2015.

[20] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. arXiv preprint arXiv:1710.10903, 2017.

[21] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315, 2019.

[22] H. Xie, D. Zheng, J. Ma, H. Zhang, V. N. Ioannidis, X. Song, Q. Ping, S. Wang, C. Yang, Y. Xu, B. Zeng, and T. Chilimbi. Graph-aware language model pre-training on a large graph corpus can help multiple graph applications. In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD '23, page 5270–5281, New York, NY, USA, 2023. Association for Computing Machinery.

[23] B. Yang, S. W.-t. Yih, X. He, J. Gao, and L. Deng. Embedding entities and relations for learning and inference in knowledge bases. In Proceedings of the International Conference on Learning Representations (ICLR) 2015, 2015.

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10), Boston, MA, June 2010. USENIX Association.

[25] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi. Agl: A scalable system for industrial-purpose graph machine learning. Proceedings of the VLDB Endowment, 13(12).

[26] S. Zhang, Y. Liu, Y. Sun, and N. Shah. Graph-less neural networks: Teaching old MLPs new tricks via distillation. In International Conference on Learning Representations, 2022.

[27] J. Zhao, M. Qu, C. Li, H. Yan, Q. Liu, R. Li, X. Xie, and J. Tang. Learning on large-scale text-attributed graphs via variational inference. In The Eleventh International Conference on Learning Representations, 2023.

[28] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. CoRR, abs/2010.05337, 2020.

[29] D. Zheng, X. Song, C. Ma, Z. Tan, Z. Ye, J. Dong, H. Xiong, Z. Zhang, and G. Karypis. Dgl-ke: Training knowledge graph embeddings at scale. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '20, page 739–748, New York, NY, USA, 2020. Association for Computing Machinery.

[30] W. Zheng, E. W. Huang, N. Rao, S. Katariya, Z. Wang, and K. Subbian. Cold brew: Distilling graph node representations with incomplete or missing neighborhoods. In International Conference on Learning Representations, 2022.

[31] H. Zhou, D. Zheng, I. Nisa, V. Ioannidis, X. Song, and G. Karypis. TGL: A general framework for temporal gnn training on billion-scale graphs. Proc. VLDB Endow., 15(8), 2022.

[32] J. Zhu, Y. Zhou, V. N. Ioannidis, S. Qian, W. Ai, X. Song, and D. Koutra. Spottarget: Rethinking the effect of target edges for link prediction in graph neural networks. arXiv e-prints, pages arXiv–2306, 2023.

[33] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. Aligraph: a comprehensive graph neural network platform. Proceedings of the VLDB Endowment, 12(12):2094–2105, 2019.

A LINK PREDICTION SCORE FUNCTIONS AND LOSS FUNCTIONS

A.1 Score functions

GraphStorm provides two score functions:

- **Dot Product**:

$$score(v_i, v_j) = \sum_{i=0}^{n-1} emb_{v_i} * emb_{v_j} \tag{2}$$

  where v_i and v_j are two nodes, emb_{v_i} is the embedding of v_i, emb_{v_j} is the embedding of v_j, and the dimension of emb_{v_i} and emb_{v_j} is n.
- **DistMult**:

$$score(v_i, v_j) = \sum_{i=0}^{n-1} emb_{v_i} * emb_{rel_{ij}} * emb_{v_j} \tag{3}$$

  where where v_i and v_j are two nodes, emb_{v_i} is the embedding of v_i, emb_{v_j} is the embedding of v_j, emb_{rel_ij} is the embedding regarding to the edge type of the edge between v_i and v_j, and the dimension of emb_{v_i} and emb_{v_j} is n.

A.2 Loss functions

GraphStorm provides three loss functions:

- **Cross entropy**: The cross entropy loss turns a link prediction task into a binary classification task. We treat positive edges as 1 and negative edges as 0. The loss of edge e is as following:

$$loss = -y \cdot \log score + (1 - y) \cdot \log(1 - score) \tag{4}$$

  where y is 1 when e is a positive edge and 0 when it is a negative edge. $score$ is the score number of e computed by the score function.
- **Weighted cross entropy**: The weighted cross entropy loss is similar to **Cross entropy** loss except that it allows users to set a weight for each positive edge. The loss function of e is as following:

$$loss = -w_e \left[y \cdot \log score + (1 - y) \cdot \log(1 - score)\right] \tag{5}$$

  where y is 1 when e is a positive edge and 0 when it is a negative edge. $score$ is the score number of e computed by the score function, w_e is the weight of e and is defined as

$$w_e = \begin{cases} 1, & \text{if } e \in G, \\ 0, & \text{if } e \notin G \end{cases} \tag{6}$$

  where G is the training graph.
- **Contrastive loss**: The contrastive loss compels the representations of connected nodes to be similar, while forcing the representations of disconnected nodes remain dissimilar. In the implementation, we use the score computed by the score function to represent the distance between nodes. When computing the loss, we group one positive edge with the N negative edges corresponding to it. The loss function is as following:

$$loss = -log(\frac{exp(pos\_score)}{\sum_{1=0}^{N} exp(score_i)}) \tag{7}$$

  where pos_score is the score of the positive edge. $score_i$ is the score of the i-th edge. In total, there are $N + 1$ edges, within which there is 1 positive edge and N negative edges.

A.2.1 Negative sampling methods. GraphStorm provides four negative sampling methods:

- **Uniform negative sampling**: Given N training edges under edge type (src_t, rel_t, dst_t) and the number of negatives set to K, uniform negative sampling randomly samples K nodes from dst_t for each training edge. It corrupts the training edge to form K negative edges by replacing its destination node with sampled negative nodes. In total, it will sample $N * K$ negative nodes.
- **Joint negative sampling**: Given N training edges under edge type (src_t, rel_t, dst_t) and the number of negatives set to K, joint negative sampling randomly samples K nodes from dst_t for every K training edges. For these K training edges, it corrupts each edge to form K negative edges by replacing its destination node with the same set of negative nodes. In total, it only needs to sample N negative nodes. (We suppose N is dividable by K for simplicity.)
- **Local joint negative sampling**: Local joint negative sampling samples negative edges in the same way as joint negative sampling except that all the negative nodes are sampled from the local graph partition.
- **In-batch negative sampling**: In-batch negative sampling creates negative edges by exchanging destination nodes between training edges. For example, suppose there are three training edges (u_1, v_1), (u_2, v_2), (u_3, v_3), In-batch negative sampling will create two negative edges (u_1, v_2) and (u_1, v_3) for (u_1, v_1), two negative edges (u_2, v_1) and (u_2, v_3) for (u_2, v_2) and two negative edges (u_3, v_1) and (u_3, v_2) for (u_3, v_3). If the batch size is smaller than the number of negatives, either of the above three negative sampling methods can be used to sample extra negative edges.

B COMMAND LINE INTERFACE

GraphStorm provides two options for graph construction:

- *graphstorm.gconstruct.construct_graph* runs in a single machine. It is mainly used by the model developers to prototype a GML model on a subset of the application data.
- GraphStorm Distributed Data Processing (GSProcessing) runs on multiple machines. It is mainly designed for industry-scale deployment.

The command below shows how to construct a graph with *graphstorm.gconstruct.construct_graph*. It takes the input data and the graph schema defined by a user in a JSON file.

```
python3 -m graphstorm.gconstruct.construct_graph \
    --num-processes NUM_PROCESSES \
    --output-dir OUTPUT_DIR_PATH \
    --graph-name GRAPH_NAME \
    --num-partitions NUM_PARTITIONS \
    --conf-file GRAPH_SCHEMA_JSON_FILE
```

Figure 6 shows an example of the graph schema configuration in a JSON file that contains the descriptions of the information of nodes and edges.

GraphStorm provides different modules for different graph tasks. For example, GraphStorm provides *graphstorm.run.gs_link_prediction* for link prediction and provides *graphstorm.run.gs_node_classification* for node classification. The same module can be used for both training and inference.

```
python3 -m graphstorm.run.gs_link_prediction \
        --num-trainers NUM_TRAINERS \
        --part-config GRAPH_PARTITION_JSON \
        --ip-config IP_LIST_FILE \
        --cf MODEL_CONF_FILE \
        --save-model-path MODEL_OUTPUT_PATH
```

The main difference between training and inference is that users need to provide $--inference$ to turn the module to the inference mode. It also requires the argument $--restore-model-path$ to load the saved model artifacts and the argument $--save-embed-path$ to save the node embeddings.

```
python3 -m graphstorm.run.gs_link_prediction \
        --inference
        --num_trainers NUM_TRAINERS \
        --part-config GRAPH_PARTITION_JSON \
        --ip-config IP_LIST_FILE \
        --cf MODEL_CONF_FILE \
        --save-embed-path EMBED_OUTPUT_PATH \
        --restore-model-path MODEL_PATH
```

```
{
    "version": "gconstruct-v0.1",
    "nodes": [
        ......
        {
            "node_type": "paper",
            "format": {
                "name": "parquet"
            },
            "files": [
                "nodes/paper.parquet"
            ],
            "node_id_col": "node_id",
            "features": [
                {
                    "feature_col": "feat",
                    "feature_name": "feat"
                }
            ],
            "labels": [
                {
                    "label_col": "label",
                    "task_type": "classification",
                    "split_pct": [0.8, 0.1, 0.1]
                }
            ]
        },
        ......
    ],
    "edges": [
        ......
        {
            "relation": ["paper", "citing", "paper"],
            "format": {
                "name": "parquet"
            },
            "files": [
                "edges/paper_citing_paper.parquet"
            ],
            "source_id_col": "source_id",
            "dest_id_col": "dest_id",
            "labels": [
                {
                    "task_type": "link_prediction",
                    "split_pct": [0.8, 0.1, 0.1]
                }
            ]
        },
    ......
    ]
}
```

Figure 6: A graph schema JSON file example.

C PROGRAMMING INTERFACE

To use GraphStorm's programming interface, users need to implement the forward function to define how to compute the loss function on a mini-batch to train the model, the prediction function to define how to compute the prediction results on each node/edge, the function to create an optimizer, the function to save a model and the function to restore a model from the saved checkpoint.

```
class GSgnnNodeModelBase:
    def forward(self, blocks, node_feats,
        edge_feats, labels, input_nodes)
    def predict(self, blocks, node_feats,
        edge_feats, input_nodes, return_proba):

    def create_optimizer(self):

    def restore_model(self, model_path):

    def save_model(self, model_path):
```

Figure 7: The model template for node prediction tasks.

C.1 Customized Model Example

GraphStorm allows users to develop their own GML models by extending custom model APIs, and run these models in GraphStorm's training and inference pipelines. Here is an example of a customer model in the Graphstorm tutorial (https://graphstorm.readthedocs.io/en/latest/advanced/own-models.html).

```
class HGT(GSgnnNodeModelBase):
    def __init__(self, ......)
    ......
    # use GraphStorm loss function components
    self._loss_fn = ClassifyLossFunc(multilabel=False)

    def forward(self, blocks, node_feats, _, labels, _):
        h = node_feats
        for i in range(self.num_layers):
            h = self.gcs[i](blocks[i], h)
        for ntype, emb in h.items():
            h[ntype] = self.out(emb)
        preds = h[self.target_ntype]
        targets = labels[self.target_ntype]
        loss = self._loss_fn(preds, targets)
        return loss
```

Figure 8: Customized HGT model.