EmbedX: A Versatile, Efficient and Scalable Platform to Embed Both Graphs and High-Dimensional Sparse Data

Yuanhang Zou*
Tencent
yuanhangzou@tencent.com

Zhihao Ding*
Hong Kong Polytechnic University
22040455r@connect.polyu.hk

Jieming Shi†
Hong Kong Polytechnic University
jieming.shi@polyu.edu.hk

Shuting Guo
Tencent
tinkleguo@tencent.com

Chunchen Su
Tencent
hillsu@tencent.com

Yafei Zhang
Tencent
kimmyzhang@tencent.com

ABSTRACT

In modern online services, it is of growing importance to process web-scale *graph data* and *high-dimensional sparse data* together into embeddings for downstream tasks, such as recommendation, advertisement, prediction, and classification. There exist learning methods and systems for either high-dimensional sparse data or graphs, but not both.

There is an urgent need in industry to have a system to efficiently process both types of data for higher business value, which however, is challenging. The data in Tencent contains billions of samples with sparse features in very high dimensions, and graphs are also with billions of nodes and edges. Moreover, learning models often perform expensive operations with high computational costs. It is difficult to store, manage, and retrieve massive sparse data and graph data together, since they exhibit different characteristics.

We present EmbedX, an industrial distributed learning framework from Tencent, which is versatile and efficient to support embedding on both graphs and high-dimensional sparse data. EmbedX consists of distributed server layers for graph and sparse data management, and optimized parameter and graph operators, to efficiently support 4 categories of methods, including deep learning models on high-dimensional sparse data, network embedding methods, graph neural networks, and in-house developed joint learning models on both types of data. Extensive experiments on massive Tencent data and public data demonstrate the superiority of EmbedX. For instance, on a Tencent dataset with 1.3 billion nodes, 35 billion edges, and 2.8 billion samples with sparse features in 1.6 billion dimension, EmbedX performs an order of magnitude faster for training and our joint models achieve superior effectiveness. EmbedX is deployed in Tencent. A/B test on real use cases further validates the power of EmbedX. EmbedX is implemented in C++ and open-sourced at https://github.com/Tencent/embedx.

PVLDB Reference Format:
Yuanhang Zou, Zhihao Ding, Jieming Shi, Shuting Guo, Chunchen Su, and Yafei Zhang. EmbedX: A Versatile, Efficient and Scalable Platform to Embed Both Graphs and High-Dimensional Sparse Data. PVLDB, 16(12): 3543 - 3556, 2023.
doi:10.14778/3611540.3611546

*Co-primary authors.
†Corresponding author.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611546

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Tencent/embedx.

1 INTRODUCTION

High-dimensional sparse data and *graph data* are two prominent types of data in real-world applications. At Wechat, an instant messaging platform from Tencent, there are billions of users and items (*e.g.*, images, articles, videos, and addresses) with very high-dimensional sparse features. Meanwhile, the interactions among users and items are modeled as graphs. Massive user log data are continuously generated in Petabytes. It is of paramount importance to efficiently process such web-scale data in industry.

One classic methodology is to adopt *deep learning models on high-dimensional sparse data (DLS)* to learn embeddings of users and items. For instance, a user can have a sparse feature vector in high dimension to indicate the places where the user has visited by one-hot or multi-hot values in the vector, while all the other values are zero [2]. As another example, a sparse feature vector could depict the images published by a user. DLS models, particularly recommendation systems, have been proposed [12, 52] to learn effective representations of entities to facilitate downstream applications, such as recommendation, searching, advertisement and marketing in Tencent.

Meanwhile, another popular way is to build sophisticated graphs to model the interactions among users and items, where nodes are the entities and edges are the interactions. For instance, a payment transaction from a user to another user indicates a transaction edge between the users. There exist numerous graph representation learning methods, including *network embedding (NE)* [10, 11, 41, 42] and *graph neural network (GNN)* methods [14, 23, 46, 48, 49], to learn effective node embeddings. Graph data in Tencent are heterogeneous with node and edge types, and are attributed. The graphs are with billions of nodes and edges, which are massive, valuable, but also challenging to efficiently process. NE and GNN methods are useful in industry for various applications, *e.g.*, evaluating the default rate of loans, measuring the security of user identity, and recommending music.

In production, it is preferable to apply learning models on both graphs and high-dimensional sparse data together for higher business value. However, in literature, there exist systems only for graph data [30, 38, 56, 58, 59] or high-dimensional sparse data [20, 29, 36], but not both. Also there are general machine learning platforms, such as TensorFlow [1], PyTorch [40], MxNet [6], and Caffe [19], which still require significant efforts for customized development of dedicated models on specific types of data.

To our knowledge, there is a lack of industrial-level scalable frameworks to seamlessly support models on both graphs and high-dimensional sparse data. The **challenges** are threefold.

- *How to efficiently process web-scale high-dimensional sparse data and graph data for offline training and online inference?* In industry, the number of data samples, feature dimensions, and the number of nodes and edges are all massive. Existing learning models usually incur immense overheads with expensive operations, *e.g.*, sampling and gradient updates. Moreover, services require timely online inference to provide prompt responses to end users, which is rather challenging for big models.
- *How to efficiently store, manage, and retrieve high-dimensional sparse data and graph data together, which exhibit different characteristics?* The underlying system architecture is crucial for the efficiency and scalability on sparse data and graph data that cannot be simply handled together, since they usually require different formats of storage and involve different operations. For instance, sparse features are often projected to dense embeddings by neural networks, while sampling techniques, *e.g.*, random walks, are common graph operations.
- *How to develop joint models to learn more expressive embeddings by leveraging the two perspectives from high-dimensional sparse data and graphs?* As explained, while it is with great business value to generate embeddings by considering the rich semantics of both sparse and graph data, this is insufficiently explored yet, especially on system developments. It is challenging to develop new models since models for sparse data and graph data are with different design philosophies over radically different data models.

To address these challenges, we present EmbedX, an industrial distributed learning framework that is versatile to support 4 categories of methods for both graphs and high-dimensional sparse data, as listed in Table 1. EmbedX is efficient, scalable and effective to handle web-scale data for offline training and online inference. We summarize the **contributions** of EmbedX as follows.

Server Infrastructure. To support both high-dimensional sparse data and graph data, in EmbedX, we build a server layer that consists of (i) graph servers, parameter servers, and training workers for offline training, and (ii) KV stores and serving workers for online inference. The server layer is optimized by several techniques for the efficient processing of massive data, including a *bit-level type encoding* scheme to save storage costs, an *aggressive asynchronous communication* mechanism, a *parallel request processing* technique, as well as *online inference optimization*.

Operators. EmbedX provides two sets of efficient operators, including *parameter operators* and *graph operators*. The parameter operators can efficiently retrieve embeddings and support parameter updates during both online and offline stages. The graph operators are highly efficient with dedicated techniques for efficient *random walk sampling*, *negative sampling*, and *neighbor sampling*, which are the key operations in graph learning models.

Table 1: Four Categories of Models built in EmbedX

Category	Method	Sparse	Graph	In-house
Deep Learning Models on High-dimensional Sparse Data (**DLS**)	YouTubeDNN [8]	✓		
	DSSM [17]	✓		
	DeepFM [12]	✓		
	Self-Training DSSM	✓		✓
	Online DSSM	✓		✓
	Submodel-DSSM	✓		✓
Network Embedding Methods (**NE**)	DeepWalk [41]		✓	
	Node2vec [11]		✓	
	Struct2vec [42]		✓	
	Metapath2vec [10]		✓	
	EGES [47]		✓	
Graph Neural Networks (**GNN**)	GraphSAGE [14]		✓	
	PinSAGE [53]		✓	
	GAT [46]		✓	
	StrucGraphSAGE		✓	✓
	MetaGraphSAGE		✓	✓
	Self-Training GraphSAGE		✓	✓
	Joint-Training GraphSAGE		✓	✓
Joint Learning Models on Sparse and Graph Data (**JLSG**)	GraphDeepFM	✓	✓	✓
	BipartiteGraphDeepFM	✓	✓	✓
	GerlDeepFM	✓	✓	✓
	GraphEsmmDFM	✓	✓	✓
	GraphDTN	✓	✓	✓
	GraphDSSM	✓	✓	✓

Algorithms. Built on top of the server and operator layers, EmbedX is versatile to support *4 categories of learning algorithms*, including deep learning models for sparse data (DLS), network embedding (NE) methods, graph neural networks (GNNs), and in-house joint learning models on both sparse and graph data (JLSG). All the new models developed in-house are ticked in the 3rd column of Table 1. Particularly, we develop all JLSG models, such as GraphDeepFM and GraphDSSM, while in literature, models on both high-dimensional sparse data and graph data are under-explored. For DLS, NE, and GNN models, EmbedX has built-in models either from existing studies or developed in-house.

Deployment and Evaluation. We conduct extensive experiments on real-world billion-scale Tencent datasets and public datasets to demonstrate the superior performance of EmbedX in Section 7. Moreover, EmbedX is deployed in multiple business sectors in Tencent. We provide A/B test results of use cases in production for News feed, music recommendation, and malicious account discovery, which further validates the power of EmbedX as an industrial-level platform.

2 PRELIMINARIES

We introduce the data models and the general embedding workflows of high-dimensional sparse data and graph data (Figure 1).

Deep Learning Models on High-Dimensional Sparse Data (DLS). In online services, there are many types of entities, such as users, articles, images, etc. These entities have sparse features and dense features. For instance, in the lower part of Figure 1, a user u has dense features $\mathbf{x}^d$, *e.g.*, age and gender, and has sparse features $\mathbf{x}^s$ depicting his/her online behaviors. The sparse features

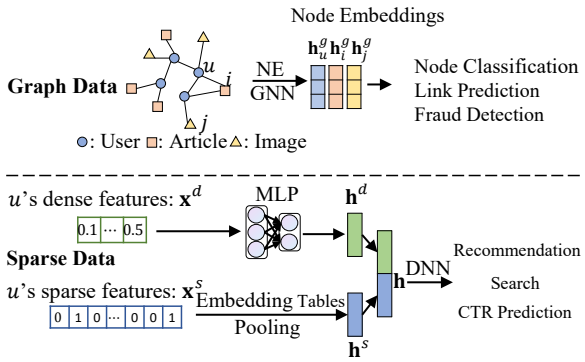


Figure 1: Embedding workflows on graph data (upper) and high-dimensional sparse data (lower).

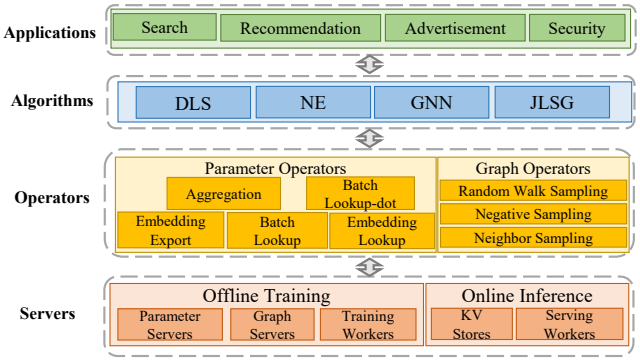


Figure 2: EmbedX Architecture.

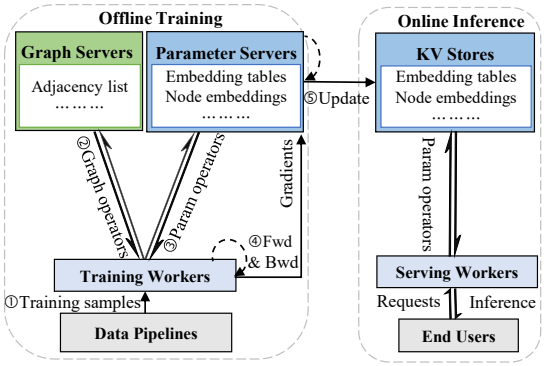


Figure 3: EmbedX Offline and Online Workflows.

are usually in the form of high-dimensional one-hot or multi-hot vectors, which may indicate the places where u has visited and the articles that u has read. The dimension of $\mathbf{x}^s$ can be high, *e.g.*, in billions in production. As show in Figure 1, DLS models usually adopt embedding tables and pooling techniques to train and convert sparse vector $\mathbf{x}^s$ to dense embedding $\mathbf{h}^s$. Embedding $\mathbf{h}^s$ for sparse features are usually integrated with the embedding $\mathbf{h}^d$ of dense features $\mathbf{x}^d$ obtained by neural networks, *e.g.*, Multilayer perceptron (MLP) in Figure 1, to get the final representation $\mathbf{h}$ of an entity. Then the embeddings of two entities may be then fed into deep neural networks to train for various tasks, *e.g.*, recommendation.

Network Embedding (NE) and Graph Neural Networks (GNN). The interactions among different entities can also be modeled as graphs, in which nodes and edges are with types and attributes, *e.g.*, the upper part of Figure 1. For instance, two users as nodes are connected by an friendship edge. NE and GNN methods [11, 14, 23, 41, 46] are popular and important to consider high-order (multi-hop) relationships between entities, to learn meaningful node embeddings. The node embedding $\mathbf{h}_u^g$ of a user u and the node embedding $\mathbf{h}_i^g$ of an item i in a graph can be used to train models for link prediction, node classification, fraud detection, etc.

3 EMBEDX OVERVIEW

EmbedX is implemented with the design principles to support a rich collection of learning models on billion-scale high-dimensional sparse data and graph data in an efficient and effective manner for offline training and online inference. In this section, we provide the whole picture of EmbedX with 4 layers, and explain its offline and online workflows. We present the technical designs of each layer in subsequent sections.

Architecture. Figure 2 depicts the architecture of EmbedX with 4 layers, namely server layer, operator layer, algorithm layer and application layer. In a nutshell, the server layer provides the storage of sparse data and graph data, the management of models parameters, and the computational workers for offline training and online inference. Specifically, offline training is supported by a set of parameter servers, graph servers, and training workers; online inference adopts online KV stores and serving workers. On top of the server layer, EmbedX provides a rich set of popular and important operations in the operator layer for both high-dimensional sparse data and graph data. Specifically, the parameter operators include embedding lookup, aggregation, batch lookup, batch lookup-dot, and embedding export operations. Representative graph operators include various types of random walk sampling, negative sampling, and neighborhood sampling. At the algorithm layer, EmbedX implements 4 categories of methods, including DLS, NE, GNN, and JLSG methods as listed Table 1. Note that plenty of these methods are newly developed in-house at Tencent. The application layer of EmbedX integrates with the wide application domains in Tencent for search, recommendation, advertisement, security, etc.

The workflow of EmbedX is illustrated in Figure 3.

Offline Training Workflow:

(1) Training workers fetch batches of training samples in the form of ⟨userID, itemID⟩ pairs from data pipelines (*e.g.*, Kafka and Hadoop) for batch training.
(2) Training workers then send requests to graph servers to perform specific graph operators (*e.g.*, random walk sampling, negative sampling, and neighbor sampling) for the training samples. Graph data is distributed in multiple graph servers.
(3) Meanwhile, training workers also send requests to parameter servers to lookup the related high-dimensional sparse features, embedding tables, and model parameters of the training samples in parameter servers via the parameter operators.
(4) Having all the necessary training high-dimensional sparse data and graph data of the training samples, training workers train models by performing forward and backward model execution, and then calculate and send the gradients to parameter servers.

(5) Parameter servers update model parameters, embedding tables, *etc.*, based on the results received from training workers. The updated parameters are persisted as checkpoints. Offline parameter servers also perform incremental updates to online KV stores periodically to keep online models up to date.

Online Inference Workflow. Note that EmbedX supports various tasks. We use recommendation as an example to explain the online inference workflow. As shown on the right of Figure 3, when receiving recommendation requests from end users, serving workers will retrieve the corresponding model parameters, embeddings of sparse features, node embeddings of the user, and candidate items from the online KV stores. Then serving workers invoke the deployed models for online inference.

In the following, we elaborate the system designs and optimizations for the server layer in Section 4, present the algorithmic designs of operators in Section 5, develop the built-in models in Section 6, and conduct extensive experiments in Section 7. Note that EmbedX also contains standard configurations in modern systems, such as caching and backup. In this paper, we focus on explaining the optimizations specifically designed in EmbedX.

4 SERVER LAYER

The server layer has two parts, one for offline training and the other for online inference. Offline training is conducted on massive historical training data and involves expensive iterative model forward and backward optimizations. Online inference is required to be performed in near real-time over the up-to-date online data. Hence, the server-layer designs for offline training and online inference are quite different. The designs of offline training and optimizations are presented in Sections 4.1 and 4.2, respectively, while the designs of online inference are elaborated in Section 4.3.

4.1 Offline Training

The offline part is responsible for the storage and management of graph data (graph servers) and high-dimensional sparse data and model parameters (parameter servers), and also provide the computational workhorses for training (training workers).

Parameter Servers. EmbedX has a group of parameter servers that store and update model parameters based on the computational results obtained from training workers during the offline training stage. Parameter servers also disseminate the updated parameters to the clients in online server layer for online inference. Remark that the model parameters include the parameters of learning models (*e.g.*, GNNs) and embedding tables that convert sparse data into dense embeddings. The model parameters are stored and retrieved in key-value storage in a distributed manner, since the embedding tables for high-dimensional sparse data are in massive volume in production. We partition embedding tables into shards by row ids in the tables. Then the embedding shards are stored in the distributed parameter servers. Moreover, parameter servers consist of computational graphs and optimizers, *e.g.*, Adam[22] and SGD[4], for different models, to efficiently perform gradient descent.

Graph Servers. The graph data at Tencent contain billions of nodes and hundreds of billions of edges, which is far beyond the capacity of a commodity machine. Thus, EmbedX stores a graph in a distributed way across multiple graph servers and provides efficient operators on graphs, in order to facilitate the data requests from training workers. EmbedX provides flexible ways to store various types of graphs, including homogeneous graphs, heterogeneous graphs, and attributed graphs [11, 16, 23, 41]. To split massive graph data into partitions, we implement a series of graph partitioning methods in EmbedX, including random shuffling by ids, Metis[21], edge cut and node cut partitions [49]. Clients have the flexibility to choose a partitioning method to split a graph into several parts for distributed storage. For instance, a fast way is to decide the graph server to store a node v's adjacency list is to modulo the node id of v over the number of graph servers. In practice, this way is often robust to various types of graphs and efficient to handle large-scale data, with minimum partitioning cost.

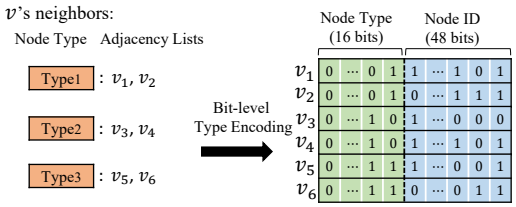


Figure 4: Bit-level Type Encoding.

Training Workers. The training workers in EmbedX work in parallel to fetch data from graph servers, embeddings of sparse data and model parameters from parameter servers, and perform forward and backward execution over the models under training. In EmbedX, training workers and parameter servers are designed to communicate asynchronously, as explained in Section 4.2.

4.2 Optimizations of Offline Training

In EmbedX, we develop novel server-level optimization techniques to facilitate efficient processing and save storage costs. In particular, we will introduce three representative server optimizations, namely bit-level type encoding scheme, aggressive asynchronous communication, and parallel requests to parameter servers.

Bit-level Type Encoding. We need to store different types of entities (*e.g.*, users, documents, videos, images, songs, *et al.*,) and all kinds of interactions between entities as heterogeneous graphs. At the scale of billions of nodes with hundreds of node types, it is space-consuming to solely store the type information of a node's adjacent nodes. A common way is to group the neighbors of a node by types and then maintain the neighbors of the same type into a type-specific adjacency list. For example, on the left side of Figure 4, it shows the three type-specific adjacency lists of node v. However, this way may cost hundreds of GB to store the type information on massive graphs. For instance, given a graph with 1 billion nodes and 100 node types, each node needs to store up to 100 node types of its neighbors in integers, and assume that an integer takes 4 bytes. Then the total space required to store the neighboring node type information of all nodes is about up to 400GB. If the integer takes 8 bytes, the space cost is up to 800GB. Another challenge is how to efficiently perform neighbor sampling with node types on large graphs, which is a common procedure required in graph learning algorithms. To address these challenges, we propose a bit-level type

encoding scheme to integrate node type into node id integer and thus save the storage cost for type information. As shown on the right side of Figure 4, given a 64-bit integer, the encoding scheme uses the lower 48 bits to store a node id and the upper 16 bits to encode the corresponding node type. Then given a node v, we store its neighbors represented by the type-encoded integers into a single adjacency list that is sorted based on the integer values, *e.g.*, Figure 4. Naturally, the neighbors are grouped based on their types in the list, since the upper 16 bits of the integers represent node type information. The encoding scheme stores node ids and their types together, and thus can save hundreds of GB. The bit encoding scheme also enables fast neighbor sampling via bit operations.

Aggressive Asynchronous Communication. In traditional parameter servers [27], the process of model training typically consists of 4 steps as shown in Figure 5(a). When a training worker gets a batch of training data as the current task, it first issues a Pull request to parameter servers to get the model parameters to be updated in the current task. Second, the training worker performs forward and backward executions to compute gradients. Third, the training worker issues a Push request to push the gradients to parameter servers. Fourth, the training worker waits for acknowledgement from parameter servers and will mark the task as completed until acknowledgement received. If no acknowledgement is received, the worker will have to issue another Push request to parameter servers. This traditional synchronous communication process, especially the fourth step where training worker waits in idle for acknowledgement, slows down the training process on web-scale data. In EmbedX, we propose to perform aggressive asynchronous communication from training workers to parameter servers, such that the fourth step is skipped as shown in Figure 5(b), in order to let training workers immediately commit to new training tasks after Push without waiting for acknowledgement. Compared with the traditional way, EmbedX experiences higher utilization rate of training workers. Note that there is a trade-off between training efficiency and model convergence. The skip of the fourth step may have the risk of losing gradient updates of certain batches in parameter servers. We find that the risk is negligible in practice, especially when training data are abundant. It is practically infrequent to lose gradient updates. Thus, it is tolerable to have few gradient losses happen, which is insignificant on model effectiveness, but improves the efficiency, as validated in experiments.

Parallel Requests to Parameter Servers. In our application scenarios, a parameter server needs to handle a huge number of pull/push requests from training workers. Conventionally, when a parameter server is handling a request, other requests are in queue, which is inefficient. Therefore, we configure and enable a parameter server to handle multiple requests simultaneously by multiple threads in EmbedX. As the key-value store in parameter servers uses hash-map as low-level data structure, which forbids concurrent read and write operations, we modify the key-value store with read/write locks to support parallel requests to parameter servers. This optimization makes the parameter servers more efficient than conventional ones in production.

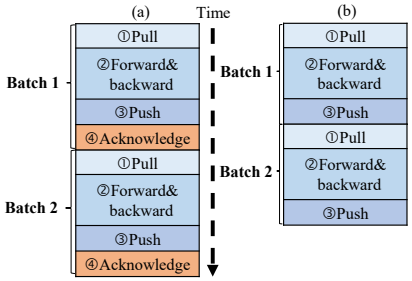


Figure 5: (a): Traditional synchronous communication; (b): Aggressive asynchronous communication.

4.3 Online Inference and Optimization

The server design for online inference is different from that of offline training. Specifically, EmbedX provides KV stores and serving workers to facilitate efficient online inference.

KV Stores. The online KV stores store parameters of models, embedding tables, and computational graphs. KV stores *incrementally* receive latest parameters pushed from offline parameter servers. Different from offline parameter servers, KV stores do not need to perform operations to update model parameters. In other words, KV stores can be regarded as a light-weight version of parameter servers by discarding training optimizers.

Serving Workers. Servering workers are the main computational power to perform online inference. As mentioned, serving workers obtain model parameters, embeddings of the user and item from the online KV stores. Then serving workers run the deployed models for online inference tasks, *e.g.*, recommendation, ranking, classification.

Online Inference Optimization. In industry, it is crucial to provide timely online inference results to users. For instance, in WeChat messaging platform, it is preferable to handle online recommendation in 30ms (milliseconds) per recommendation request. However, this is challenging since online inference involves feature fetching of users and candidate items from very large feature databases. Even worse, new data is generated dynamically online. If the latest data (*e.g.*, online graphs) are used for online inference, the models need to perform expensive forward executions on the fly, which greatly slows down online inference. To mitigate the situation, we exploit a trade-off between online efficiency and data timeliness, to match the fast online inference requirements in milliseconds. In particular, we maintain the graph data in offline storage to be as latest as possible compared with online data, with negligible differences, so that the offline representation of a node in offline graphs does not change a lot compared with its online counterpart. Then we can safely use the node embeddings obtained from offline graph data for online inference. In such a way, we avoid the expensive online execution of graph learning models and improve online efficiency. Specifically, in the parameter servers, we maintain a node embedding table containing the representations of all nodes, and the table will be incrementally synchronized to online KV stores for inference. This optimization can not only reduce the time cost of online inference, but also relieve EmbedX from maintaining online graph servers, which saves significant computational resources.

5 OPERATOR LAYER

As shown in Figure 2, EmbedX provides efficient parameter operators to manage massive high-dimensional sparse data and model parameters (Section 5.1). Moreover, EmbedX has dedicated graph operators to efficiently process graphs (Section 5.2).

5.1 Parameter Operators and Optimizations

The parameter operators aim to efficiently access embedding tables and support frequent operations on model parameters during both offline training and online inference stages. In EmbedX, we provide highly optimized parameter operators implemented in C++.

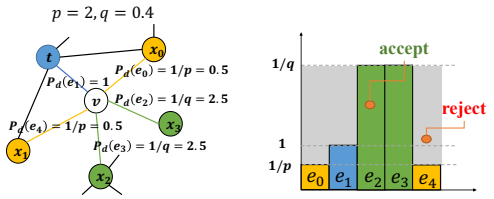
- **Embedding lookup operator** is an elementary operation that retrieves dense embeddings according to IDs from embedding tables in parameter servers.
- **Batch dot operator** performs dot product calculation between two batches of users and items. Dot product is frequently used in machine learning models, and naive dot product over high-dimensional vectors can be rather expensive if triggered many times. The batch dot operator performs in batches to reduce memory consumption and improve efficiency by parallelism.
- **Batch lookup-dot operator** combines the operations of embedding lookup and batch dot together in one encapsulated operation. We observe that usually embedding lookup and dot product operators often appear one after another. In EmbedX, to avoid frequent invocations of fine-grained operators that may require frequent memory allocation, we develop this croase-grained batch lookup-dot operator for system efficiency.
- **Aggregation operator** in EmbedX combines multiple embeddings into a single embedding vector. The aggregation operator supports various functions, including mean, sum, and max pooling, and concatenation. Aggregation is commonly utilized in all types of models.
- **Embedding export operator** enables saving updated model parameters as checkpoints efficiently in offline parameter servers, so that they can be further disseminated to online KV stores for inference. As model parameters are often in large size, we optimize the operator to reduce the latency of model saving.

In addition to the operators above, EmbedX includes more operators which are available in its code base. All the operators are implemented as APIs in EmbedX and can be called independently, so that clients can build their models in a flexible way.

5.2 Graph Operators and Optimizations

As sampling techniques are fundamental operations in many NE and GNN models, EmbedX supports a rich collection of graph operations. Specifically, EmbedX supports *three categories of graph sampling operations* on large-scale graphs as listed below. Note that for each category, there are *multiple variants* supported in EmbedX for different NE and GNN models. Vanilla implementation of these operations incur prohibitive costs, especially on billion-scale graphs. We optimize them in EmbedX for efficiency.

- **Random-Walk Sampling** simulates stochastic processes on graph topology to get sequences of nodes. EmbedX supports various types of random walks, including first-order random walk, second-order random walk, and meta-path random walks.



(a) Sample graph (b) Rejection sampling at v

Figure 6: Node2vec and Vanilla Rejection Sampling.

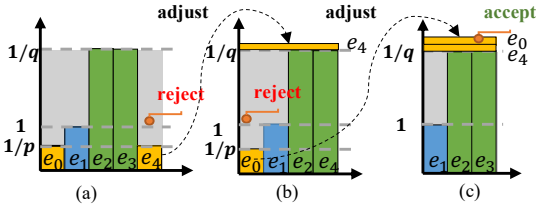


Figure 7: Dynamic Rejection Sampling (DRJ).

- **Negative Sampling** generates negative samples with respect to certain nodes, which can be used to speed up training convergence. The sampling is customized with the consideration of graph topology, attributes, and heterogeneous information.
- **Neighbor Sampling** samples subsets of nodes from the neighborhoods of target nodes for training. The neighbor sampling may consider node and edge types, attributes, etc.

As graph sampling plays a crucial role in NE and GNN methods, we implement them with dedicated optimizations. First, many NE methods heavily rely on random walk sampling [11, 41]. Sophisticated random walk sampling strategies on massive graphs incur immense computational costs and slow down the training process, *e.g.*, taking up to 98.8% of the total execution time [57]. To boost efficiency, EmbedX provides a collection of sampling techniques including *Inverse Transform Sampling (ITS)* [50], *alias method* [25], and *rejection sampling* [50]. For negative sampling, we implement it with a hierarchical sampling technique [43] to solve the inconsistency between sampling in a single graph server and sampling in distributed graph servers. We further develop a new *dynamic rejection sampling* (DRJ) below to reduce time costs in second-order random walks for NE methods, *e.g.*, Node2vec [11] and EGES [47].

Dynamic Rejection Sampling (DRJ). Second-order random walks on graphs depend not only on the current state but also the previous state [11, 44]. As shown in [50], rejection sampling is preferable for the second-order random walks, compared with ITS and alias methods that require high pre-computation costs. In EmbedX, we further develop the dynamic rejection sampling (DRJ) to enhance the efficiency of rejection sampling. We use the second-order random walks in Node2vec [11] as an example to elaborate DRJ. In Node2vec, suppose that a random walk just traversed node t and now resides at node v. The transition probability of traversing edge $e = (v, x)$ to node x as next move is $P(e) = P_s(e) \cdot P_d(t, v, x)$, where $P_s(e)$ is the static transition probability of edge e, *e.g.*, edge weight of e, and $P_d(t, v, x)$ is the dynamic transition probability depending on the shortest path distance d_{tx} between t and x as defined below. The computation of $P_d(t, v, x)$ is expensive and cannot be

pre-computed since it depends on nodes t, v, and x.

$$P_d(t,v,x) = \begin{cases} 1/p & \text{if } d_{tx} = 0, \\ 1 & \text{if } d_{tx} = 1, \\ 1/q & \text{if } d_{tx} = 2, \end{cases} \quad (1)$$

where p is a return parameter and q is an in-out-parameter, and p and q together enable Node2vec to interpolate between Breadth-first Sampling (BFS) and Depth-first Sampling (DFS) [11].

An illustration for Node2vec is shown in Figure 6 for an unweighted graph with $P_s(e) = 1$. At node v, to decide next edge e based on the dynamic transition probability P_d, rejection sampling first uniformly samples one edge e_i from e_0, e_1, e_2, e_3, and then randomly samples a number y in range $(0, Q(v))$, where $Q(v) = \max(\frac{1}{q}, 1, \frac{1}{p})$. If $y \leq P_d(e_i)$, we accept e_i as a successful sample and next move is the other end node of e_i; otherwise e_i is rejected, and we need to start another sampling trial, until an edge is accepted. This process can be seen as throwing a dart within that rectangle area consists of bars shown in Figure 6(b). The rectangle area has width $\sum_{e \in E_v} P_s(e)$ and height $Q(v)$, where E_v is the set of edges with v as staring node. Each bar represents an edge e with width $P_s(e)$ and height $P_d(e)$. The total area of bars is acceptance area and the area outside the bars in the rectangle is rejection area. When the coordinate of dart falls in the bar representing edge e, we accept e; otherwise we reject e and dart again. The expected number of trials required to sample an edge is the reverse of the ratio of acceptance area in the rectangle: $\frac{Q(v) \cdot \sum_{e \in E_v} P_s(e)}{\sum_{e \in E_v} P_s(e) \cdot P_d(e)}$.

It is obvious that the efficiency of rejection sampling is highly dependent on the ratio of acceptance area. If we could increase the ratio of acceptance area, then the sampling efficiency will be improved. However, in real-world scenarios, the ratio of accept area is dependent on graph structure and hyper-parameter settings (p,q in Node2vec). Under undesirable situations with large rejection area, *e.g.*, the grey area in Figure 6(b), due to the large $1/q$, the probability to reject a sample (*e.g.*, e_0, e_1 and e_4) is quite high, which subsequently slows down second-order random walk sampling. Hence, we propose DRJ to dynamically adjust the areas for acceptance and rejection, to improve the efficiency. However, the challenge is that it is infeasible to iterate all edges of a node v beforehand to get the distribution as shown in Figure 6(b) (*i.e.*, Figure 7(a)). What we know are just the parameters p and q. To address the challenge, DRJ is carefully designed to *only* use the samples that have been inspected to dynamically increase acceptance area and shrink rejection area, without iterating all edges. DRJ is illustrated in Figure 7. Suppose that edge e_4 with low acceptance bar in yellow is randomly sampled and rejected in the first dart (Figure 7(a)). Then DRJ adjusts the yellow bar of e_4 into a *horizontal bar* with the same area, but with width across the whole horizontal range and it is placed on top of the highest value (*i.e.*, $1/q$), as shown in Figure 7(b). In other words, we distribute the acceptance area e_4 on top of all the remaining vertical bars. Obviously, the grey area of rejection shrinks compared with the original one in Figure 7(a). Specifically, assume the width of each vertical bar is 1, *i.e.*, unweighted graphs ($P_s(e) = 1$), and the height of the original bar of e_4 is $1/p$. Then the grey area is reduced by $(1/q - 1/p)$. The height of the newly created horizontal bar of e_4 is $1/(p \cdot (\sum_{e \in E_v \setminus \{e_4\}} P_s(e)))$.

After making dynamic adjustments for multiple rejected samples (*e.g.*, e_4 and e_0 in Figure 7(c)), we can have a pile of horizontal bars on the top. Let $\hat{E}_v$ be the set of edges that were rejected in previous trials. Then all the horizontal bars are with the same width $\sum_{e' \in E_v \setminus \hat{E}_v} P_s(e')$. The height of the horizontal bar for edge e can be obtained based on the fact of its unchanged acceptance area, *i.e.*, $\frac{P_s(e) \cdot P_d(e)}{\sum_{e' \in E_v \setminus \hat{E}_v} P_s(e')}$. And the total height of the pile is $\sum_{e \in \hat{E}_v} \frac{P_s(e) \cdot P_d(e)}{\sum_{e' \in E_v \setminus \hat{E}_v} P_s(e')}$. Consequently, in DRJ, we virtually maintain two parts: the lower part consists of the original vertical bars of edges in $E_v \setminus \hat{E}_v$; the upper part contains a pile of horizontal bars that are dynamically created for previously rejected samples in $\hat{E}_v$. Then in the next trial, DRJ samples a 2-dimensional coordinate in x-axis range $[0, \sum_{e' \in E_v \setminus \hat{E}_v} P_s(e')]$ and y-axis range $[0, Q(v) + \sum_{e \in \hat{E}_v} \frac{P_s(e) \cdot P_d(e)}{\sum_{e' \in E_v \setminus \hat{E}_v} P_s(e')}]$. If the coordinate falls into the lower part (*i.e.*, with y-axis coordinate $\leq Q(v)$), we follow the original rejection sampling scheme to decide if the corresponding sample should be accepted or not. If the sample is rejected, DRJ performs the dynamic adjustment above. Otherwise, the coordinate falls into the upper area; in this case, DRJ locates the corresponding horizontal bar which the coordinate falls into, and then accepts the corresponding sample edge for next move, *e.g.*, e_0 in Figure 7(c). Note that the relative size ratio of the bars in colors is unchanged during the dynamic adjustment of DRJ, and thus, the correctness of DRJ is guaranteed.

6 ALGORITHMS

EmbedX supports a large collection of algorithms in 4 categories, as summarized in Table 1. There are plenty of new models developed in-house, such as all the JLSG models that make full use of sparse data and graph data for better performance in the applications of Tencent. In what follows, we elaborate the representative models.

6.1 Support of State-of-the-Art Models

In this section, we provide the technical summary of the state-of-the-art models built in EmbedX, which will be used later to elaborate the new models developed in-house in Section 6.2.

Deep Learning Models for High-Dimensional Sparse Data (DLS). Given a target entity (*e.g.*, a user) and a database containing billions of entities (*e.g.*, items), a typical workflow for many tasks, such as recommendation, follows a two-step procedure: *retrieval* and *ranking* [8]. In particular, retrieval as the initial step is to quickly retrieve a small candidate pool containing hundreds of candidate entities that are highly relevant to the target based on their sparse and dense features. Then ranking is applied over the small candidate pool to calculate finer ranking scores of the candidates with respect to the target. The top-ranked candidates are finally returned for the target. There exist collections of retrieval models and ranking models [8, 12, 17, 38, 58]. EmbedX is flexible to support these models, such as YouTubeDNN [8], DSSM [17], and DeepFM [12] in Table 1. In the following, we explain the classic DSSM and DeepFM, which will be used in Section 6.2.

Retrieval model DSSM follows a popular two-tower design with two encoders, user tower and item tower, which learn the embeddings of users and items respectively [52]. Given a user-item pair

$\langle u, i \rangle$, the user tower f_u contains an embedding table that converts the sparse feature vector $\mathbf{x}_u^s$ of user u to dense embedding $\mathbf{h}_u^s$ and MLPs (multi-layer perceptrons) to transform the dense features $\mathbf{x}_u^d$ of u to dense embedding $\mathbf{h}_u^d$. The embedding $\mathbf{h}_u$ of user u is obtained by concatenating $\mathbf{h}_u^s$ and $\mathbf{h}_u^d$, $\mathbf{h}_u = \mathbf{h}_u^s \parallel \mathbf{h}_u^s$. The item tower f_i follows a similar structure with its embedding table and MLPs to obtain the embeddings $\mathbf{h}_i^s$ and $\mathbf{h}_i^d$ of the sparse and dense features $\mathbf{x}_i^s$ and $\mathbf{x}_i^d$ respectively, and the item embedding $\mathbf{h}_i = \mathbf{h}_i^s \parallel \mathbf{h}_i^d$. Then DSSM adopts cosine similarity between the user embedding and item embedding, $\text{cosine}(\mathbf{h}_u, \mathbf{h}_i)$, to measure the relevance between the user and item. DSSM is trained over a pair-wise loss with negative sampling. Then in the inference stage for retrieval, candidate items are efficiently retrieved by nearest search algorithms. Other retrieval models follow a similar procedure [8, 15, 26].

Meanwhile, a representative ranking model DeepFM [12] adopts Click-Through-Rate (CTR) as ranking score. Given a user-item pair $\langle u, i \rangle$, DeepFM first obtains an embedding $\mathbf{h}$ of the pair via embedding layers f_{emb}, which considers the sparse and dense features of both the user and item, including $\mathbf{x}_u^s$, $\mathbf{x}_u^d$, $\mathbf{x}_i^s$, and $\mathbf{x}_i^d$. Then in DeepFM, factorization machines f_{FM} and MLPs f_{MLP} are used to model low-order and high-order feature interactions respectively with embedding $\mathbf{h}$ as input to predict the CTR score $\hat{y}$ as follows. DeepFM is trained by CTR binary cross entropy loss [55].

$$\hat{y} = \sigma(f_{FM}(\mathbf{h}) + f_{MLP}(\mathbf{h})), \tag{2}$$

where σ is the sigmoid function, and $\hat{y}$ is the predicted CTR score for a user-item pair.

Network Embedding Methods (NE). An important category of NE methods employs random walk sampling to consider deep graph topology to learn node embeddings [10, 11, 41, 42, 47]. A typical procedure is shown in Eq. (3). Given a node v, a number of random walks $RW(v)$ are sampled starting from v, and every random walk represents a sequence of nodes in the multi-hop vicinity of v. The sampled random walk sequences are then used in the training of maximizing NE objectives, such as SkipGram [34] in Eq. (3), to generate node embeddings $\mathbf{h}_v$ of node v.

$$\begin{aligned} RW(v) &= \text{RandomWalk}(v), \\ \max Pr(RW(v)|v) &= \prod_{u \in RW(v)} Pr(u|v) = \frac{\exp(\mathbf{h}_u \cdot \mathbf{h}_v)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{h}_k \cdot \mathbf{h}_v)} \end{aligned} \tag{3}$$

where $\mathcal{V}$ is node set of a graph G.

The objective in (3) aims to maximize the probability of observing the nodes u in node v's random walks $RW(v)$ on conditional to the embedding $\mathbf{h}_v$. Intuitively, the random walks preserve the high-order graph topological features, and nodes that frequently co-occur in the random walk sequences should be structurally close to each other and thus with similar embeddings. Notice that directly maximizing Eq. (3) is inefficient [41]. Negative sampling is often adopted to speed up training convergence. A major difference among random-walk-based NE methods lies in the random walk variants employed. DeepWalk [41] uses truncated random walks which select the next node uniformly from the adjacent nodes of current node. Node2vec [11] performs second-order random walks with dynamic transition probability as explained in Section 5.2. To capture high-order structure similarity, Struc2vec [42] performs multi-layer random walk sampling. EGES [47] constructs an item graph from users' behavior history, and item embeddings can be learned from the item graph by random walks. Heterogeneous NE methods [10, 18, 45], such as Metapath2Vec [10], learn embeddings in heterogeneous graphs via meta-path sampling techniques.

As introduced in Section 5.2, EmbedX provides efficient and scalable graph operators to meet the needs of different NE models listed in Table 1. For instance, the dynamic rejection sampling DRJ in Section 5.2 speeds up Node2vec [11], EGES [47], and other NE methods relying on high-order random walks. Further, the parameter updates of NE methods are carried out efficiently using the parameter operators in EmbedX as presented in Section 5.1.

Graph Neural Networks (GNNs). GNNs [14, 23, 46] can be formulated under a general message passing framework [14], which typically consists of three major operations: neighbor sampling, neighbor aggregation, and embedding update, as shown in Eq. (4). At each ℓ-th layer of GNNs, for a node v, GNNs first perform certain neighborhood sampling to get its neighborhood $\mathcal{N}_s(v)$. Then the embeddings $\mathbf{h}_u^{(\ell-1)}$ of the nodes $u \in \mathcal{N}_s(v)$ of the previous $(\ell-1)$-th layer are aggregated to get intermediate embedding $\mathbf{a}^{(\ell)}(v)$ that is then together with the previous embedding $\mathbf{h}_v^{(\ell-1)}$ of v to get the updated embedding $\mathbf{h}_v^{(\ell)}$ of the current ℓ-th layer.

$$\begin{aligned} &\mathcal{N}_s(v) = \text{NeighborSample}(v), \\ &\mathbf{a}^{(\ell)}(v) = \text{Aggregate}^{(\ell)}(\{\mathbf{h}_u^{(\ell-1)} : u \in \mathcal{N}_s(v)\}), \\ &\mathbf{h}_v^{(\ell)} = \text{Update}^{(\ell)}(\mathbf{h}_v^{(\ell-1)}, \mathbf{a}^{(\ell)}(v)), \end{aligned} \tag{4}$$

Different GNNs are usually with different designs of the three operations. EmbedX provides scalable and efficient neighbor sampling, neighbor aggregation, and update operators for implementing different GNNs, such as GraphSAGE [14], GAT [46], and PinSAGE [53] in Table 1. In particular, GraphSAGE first adopts uniform random neighbor sampling to sample a fixed-size subset of neighbors. Then, mean pooling is adopted as neighbor aggregation function to get intermediate $\mathbf{a}^{(\ell)}(v)$. For the update operation, GraphSAGE performs concatenation on $\mathbf{a}^{(\ell)}(v)$ and $\mathbf{h}_u^{(\ell-1)}$ followed by linear projection and non-linear activation to get the updated embedding $\mathbf{h}_v^{(\ell)}$. GAT [46] applies attention-based weighted sum for aggregation, and $\mathbf{a}^{(\ell)}(v)$ and $\mathbf{h}_u^{(\ell-1)}$ are projected and summed to get $\mathbf{h}_v^{(\ell)}$. To adapt GNNs to web-scale recommendation systems, PinSAGE [53] employs the aforementioned random walks and regards frequently visited nodes by random walks as the sampled neighborhood. Then, PinSAGE adopts weighted-mean as neighbor aggregation with $L1$-normalized visit counts of nodes as the neighbor weights. With the server-level and operator-level designs in Section 4 and 5, various GNNs are seamlessly supported in EmbedX.

6.2 New Models Developed In-house

As explained, existing methods are developed either for sparse data or graph data, but models handling both types of data are underexplored. On top of the server and operator layers in EmbedX, we develop new joint learning models (JLSG) that use both types of data for better performance in real-world scenarios.

GraphDSSM. As introduced in Section 6.1, DSSM is a retrieval model that utilizes high-dimensional sparse features. Apparently the multi-hop interactions among entities, which are modeled as

graphs, are neglected in DSSM. It is promising to further leverage the embeddings of users and items as nodes in graphs to enhance performance. Given a user-item pair $\langle u, i \rangle$, a straightforward way is to first get the embeddings of user and item by existing unsupervised NE methods, *e.g.*, [11, 14], and then regard the embeddings as extra input features of u and i into DSSM. This trivial way may yield sub-optimal performance, since embeddings are obtained in an unsupervised way without considering the training objective.

To fully integrate graph data and sparse data together, we propose a new joint model GraphDSSM. GraphDSSM consists of *four modules*: user tower, item tower, user network embedding module, and item network embedding module. The user and item towers are the same as DSSM. The user and item network embedding modules learn node embeddings of user and item when they are nodes in a graph. More importantly, the embedding outputs of the four modules are trained under the same loss function in an end-to-end manner for effectiveness. Specifically, we adopt GraphSAGE [14] for the user network embedding module f_u^g and the item network embedding module f_i^g to generate the node embedding $\mathbf{h}_u^g$ of user u and the node embedding $\mathbf{h}_i^g$ of item i in the input graph G. The embedding $\mathbf{h}_u^s$ (resp. $\mathbf{h}_u^s$) of user u (resp. item i) with respect to its high-dimensional sparse data, as well as their embeddings $\mathbf{h}_u^d$ and $\mathbf{h}_i^d$ of their dense features, are obtained via a similar way as explained in Section 6.1. Then the final user embedding concatenates $\mathbf{h}_u^g$, $\mathbf{h}_u^s$, and $\mathbf{h}_u^d$. The final item embedding concatenates $\mathbf{h}_i^g$, $\mathbf{h}_i^s$, and $\mathbf{h}_i^d$. The cosine similarity between user u and item i is calculated as

$$sim(u, i) = cosine(\mathbf{h}_u^g \parallel \mathbf{h}_u^s \parallel \mathbf{h}_u^d, \mathbf{h}_i^g \parallel \mathbf{h}_i^s \parallel \mathbf{h}_i^d).$$

Then, GraphDSSM, including the network embedding modules f_u^g, f_i^g and the two-tower neural networks f_u, f_i, is trained under a binary cross entropy objective to minimize loss

$$\mathcal{L}_{GraphDSSM} = -\log \prod_{(u,i^+)} \frac{\exp(\gamma sim(u, i^+))}{\gamma \sum_{i' \in \mathbf{I}} \exp(sim(u, i'))}, \tag{5}$$

where i^+ indicates a positive item that the user has clicked or purchased, $\mathbf{I}$ denotes the set of candidate items, and γ is a smoothing factor in the softmax function.

GraphDeepFM. We develop a new ranking model GraphDeepFM that takes into consideration not only high-dimensional sparse data but also multi-hop graph features for effectively ranking candidate items with respect to a user. In a nutshell, GraphDeepFM improves DeepFM by extending the input feature fields with graph features extracted from high-order interactions between users and items in a graph. GraphDeepFM consists of two components: the vanilla DeepFM and a new network embedding module f^g. The network embedding module f^g is responsible to generate the embedding $\mathbf{h}^g$ of a user-item pair over the input graph G, while the embedding layer f_{emb} considers the sparse and dense features of the user-item pair to generate embedding $\mathbf{h}$. Then GraphDeepFM applies factorization machines and MLPs over the concatenated embeddings of $\mathbf{h}^g$ and $\mathbf{h}$ to predict the CTR score:

$$\hat{y} = \sigma(f_{FM}(\mathbf{h}^g \parallel \mathbf{h}) + f_{MLP}(\mathbf{h}^g \parallel \mathbf{h})). \tag{6}$$

Further, GraphDeepFM is trained over a loss function $\mathcal{L}$ combining both a CTR loss $\mathcal{L}_{CTR}$ as well as a multi-hop neighbor-similarity based loss $\mathcal{L}_G$ for extracting correlations between users and items,

$$\mathcal{L}_{GraphDeepFM} = \mathcal{L}_{CTR} + \mathcal{L}_G. \tag{7}$$

Specifically, the CTR loss is a binary cross entropy loss [55]:

$$\mathcal{L}_{CTR} = -y \log(\sigma(\hat{y})) - (1 - y) \log(\sigma(1 - \hat{y})), \tag{8}$$

where y is the ground-truth score and $\hat{y}$ is the predicted score, and σ is sigmoid activation function.

Loss $\mathcal{L}_G$ assumes that the learned embeddings $\mathbf{h}^g$ should be similar to their k-hop neighbors in graph G. We extract k-hop neighbors of nodes via DeepWalk-based sampling, and $\mathcal{L}_G$ is formalized by

$$\mathcal{L}_G = -\sum_{p \in P} \sum_{v \in p} (\log(\sigma(\mathbf{h}_u^{g\top} \mathbf{h}_v^g))), \tag{9}$$

where P is a set of k-hop random walk paths, p is a path starts from node u, and σ is the sigmoid function.

More New JLSG Models. We further develop more joint models catered for various settings using both sparse data and graph data. In particular, **BipartiteGraphDeepFM** extends GraphDeepFM to specific bipartite graphs. **GerlDeepFM** extends GraphDeepFM by considering node types when aggregating and generating node embeddings. In particular, GerlDeepFM aggregates the neighbors of different types separately for a node. Then GerlDeepFM employs self-attention to learn the weights of different types of neighbors during the training process. **GraphEsmmDFM** extends GraphDeepFM with multi-task learning that co-optimizes CTR rate and post-click conversion (CVR) rate by replacing the CTR loss with the objective in ESSM [31]. We also design **GraphDTN** to improve DTN [58] by implementing the embedding layer in DTN with GNNs instead of MLPs so that the multi-hop interactions are considered.

New GNNs. We develop a series of new GNNs over GraphSAGE. **StrucGraphSAGE** integrates structural similarity into the neighbor aggregation of vanilla GraphSage. Specifically, StrucGraphSAGE constructs a structural similarity graph by [42], and performs random walks on the graph. **MetaGraphSAGE** extends GraphSAGE to heterogeneous graphs with node types and edge types via meta-path sampling and aggregation. **Self-Training GraphSAGE** adopts pseudo labeling techniques [28] to relieve the scarcity of training labels. **Joint-Training GraphSAGE** is trained under task-related objective and structure-preserving objective to encode both task and structure information into embeddings.

New DLS models. In EmbedX, we develop several new DLS models on high-dimensional sparse data. **Self-training DSSM** adopts pseudo labels and enhanced negative sampling to improve the effectiveness of DSSM, **Online DSSM** employs importance sampling and is trained with real-time online user feedback, to capture online user interests. In **Submodel-DSSM**, we enhance the robustness of the embeddings in DSSM by auxiliary contrastive learning models.

7 EXPERIMENTS

We conduct extensive experiments to evaluate the proposed EmbedX. In Section 7.2, we report the system efficiency on real web-scale graph and high-dimensional sparse data, and evaluate the key optimizations in EmbedX. In Section 7.3, we evaluate the effectiveness of the algorithms, *e.g.*, the new JLSG models, on sparse and graph data. In Section 7.4, we present the A/B test results of real cases in production, to demonstrate the power of EmbedX.

Table 2: Data Statistics (M: million. B: billion. K: thousand).

Dataset	Feature dimension	# Samples	# Nodes	# Edges	Sparse	Graph
Tencent-1	320M	240M	205M	6B	✓	✓
Tencent-2	1.6B	2.8B	1.3B	35B	✓	✓
Criteo	33M	45M	-	-	✓	
Cora	-	-	2.7K	10.5K		✓
Wiki	-	-	10.3K	333.9K		✓

7.1 Experimental Setup

Datasets. We adopt both Tencent datasets and public datasets to conduct the experiments. Table 2 lists the statistics of the 5 datasets. Tencent-1 and Tencent-2 are two web-scale datasets in Tencent. Both Tencent-1 and Tencent-2 contain not only graph data but also high-dimensional sparse data. In particular, Tencent-1 contains 205 million nodes and 6 billion edges in its graph, and 240 million samples with sparse features in 320 million dimension. The sparse feature dimension only includes the node ID space but also many other feature dimensions in Tencent to profile users and items. Tencent-2 is a much larger dataset with 1.3 billion nodes and 35 billion edges, and 2.8 billion data samples with 1.6 billion sparse feature dimension. Criteo [9] is a publicly available dataset containing high-dimensional sparse data for recommendation. The dataset comprises of 45 million feedback records of ad clicks within a week for predicting CTR. The features in Criteo are numerical and categorical features, such as the advertiser ID, the website ID, the user's IP address, *etc.* Cora [51] and Wiki [32] are two public graph datasets, widely used to evaluate NE and GNN methods.

Setup. We evaluate EmbedX over a cluster of 50 parameter servers, 50 graph servers, and 280 training workers, unless otherwise specified. A parameter server is equipped with 100GB RAM and a 10-core CPU. A graph server has 256GB RAM and a 10-core CPU. A training worker is virtually equipped with 64GB RAM and a 8-core CPU.

Evaluation Metrics. Two metrics are used for evaluating system efficiency: *training throughput*, which measures the number of training examples processed per second by the system, and *running time*. We evaluate effectiveness by various metrics [12, 15]. Specifically, *AUC* measures the area under the ROC curve. *Hit Ratio at K (HR@K)* measures the percentage of users for whom the clicked item is among the top K items recommended by the model. *Retention time* is a business metric which measures the average time per person spent in an application. *Recall* measures the ratio of true positives to the total number of positive instances. For all these effectiveness metrics, the higher the better.

7.2 System Efficiency

Here we evaluate the system efficiency of EmbedX with server-level and operator-level optimizations developed in Section 4 and Section 5 respectively, on Tencent-1 and Tencent-2 datasets that consist of large-scale high-dimensional sparse data and graph data.

Overall Efficiency. We compare EmbedX with a baseline system that is previously developed in Tencent without the system optimizations in EmbedX. Both systems are evaluated over the same hardware specifications and under the same configurations. For each system, we train a collection of methods in Table 1, and evaluate the training time per epoch. EmbedX is significantly faster than the baseline system, often by up to orders of magnitude. Specifically, in Figure 8, we report the average training time per epoch

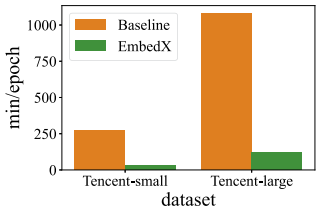


Figure 8: EmbedX Efficiency.

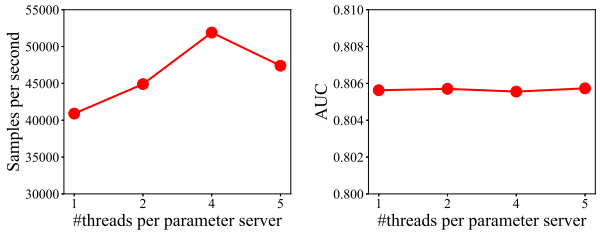
Table 3: Training throughput of EmbedX with aggressive asynchronous communication, compared with PS-Lite.

System	Training throughput (samples/s)	AUC(%)
PS-Lite	38,520	80.52
EmbedX	40,900	80.56
Rel. Imp.	+6.18%	+0.05%

of GraphDSSM on Tencent-1 and the training time per epoch of GraphDeepFM on Tencent-2 dataset of EmbedX and the baseline system. The average training time per epoch of EmbedX is 30 minutes on Tencent-1, and EmbedX is about 9 times faster than the baseline system that takes 280 minutes per training epoch. On Tencent-2 that is much larger, EmbedX takes 109 minutes per epoch to train GraphDeepFM, while the baseline system requires 1100 minutes per epoch, which indicates that EmbedX is about 10 times faster. The results demonstrate the superior efficiency and scalability of EmbedX to handle web-scale high-dimensional sparse data and large-scale graphs in industry. Note that GraphDSSM and GraphDeepFM are two new joint models developed in-house in Tencent to handle both high-dimensional sparse data and graph data. Their effectiveness improvements are reported in Section 7.3.

The superiority of EmbedX is achieved by the systematic designs of server-level optimizations in Section 4 and the novel designs of parameter operators and graph operators developed in Section 5. In the following, we specifically evaluate these key optimizations.

Aggressive Asynchronous Communication. As presented in Section 4.2, we propose to let training workers to work asynchronously to handle next training task, rather than waiting in idle to get the acknowledgement from parameter servers on whether the parameter updates are successful or not. Here we study the efficiency improvement by the technique, and its impact on effectiveness. Specifically, we train DeepFM using the parameter servers in EmbedX, and compare it with the base parameter server configuration in PS-Lite [27] that does not have our proposed technique, while all the other configurations are the same. The experiment is conducted on Criteo dataset that contains high-dimensional sparse data as listed in Table 2. In terms of model parameters, we follow the suggested settings in [12]. For a fair comparison, the number of threads per parameter server is set as 1. Table 3 reports the training throughput of EmbedX with aggressive asynchronous communication and PS-Lite (*samples/s*), as well as the AUC of the trained DeepFM model for recommendation. Observe that EmbedX achieves higher training throughput than PS-Lite, with an relative improvement (Rel. Imp.) of 6.18%, due to the proposed aggressive asynchronous communication. Meanwhile, the effectiveness of the DeepFM model trained by EmbedX is almost the same as the DeepFM model trained by the baseline, with negligible



(a) Training throughput (b) Model performance

Figure 9: Training throughput and effectiveness when varying the number of threads per parameter server in EmbedX.

Table 4: Evaluation on dynamic rejection sampling (DRJ).

Dataset	Setting	Time cost (s)			Average sampling trials	
		Vanilla	DRJ	Speedup Ratio	Vanilla	DRJ
Cora	$q = 1/1024$	3.66	**0.90**	4.07	1.27	**1.12**
	$q = 1024$	7.67	**4.82**	1.59	4.93	**3.45**
Wiki	$q = 1/1024$	33.22	**1.54**	21.57	45.74	**1.62**
	$q = 1024$	7.90	**3.36**	2.35	7.98	**2.68**

improvement of AUC in Table 3. This validates that it is feasible to tolerate few losses of parameter updates caused by the aggressive asynchronous communication technique for higher efficiency.

Parallel Requests to Parameter servers. We then evaluate the throughput of EmbedX when varying the number of threads per parameter server from 1 to 5, and report the throughput of EmbedX in Figure 9(a). The training throughput increases as the number of threads per parameter server increases and reaches its peak when the number of threads is 4, with an efficiency improvement of 26.89%, compared to single-threaded parameter servers. However, the throughput drops when the number of threads per parameter server is 5. We speculate that this is due to the network bandwidth limitation, since the data volume of Pull and Push requests to parameter servers from training workers is massive on large datasets. Figure 9(b) reports the corresponding AUC performance of the trained DeepFM, which is relatively stable when the number of threads per parameter server changes. The results demonstrate the efficiency of the optimization in EmbedX.

Dynamic Rejection Sampling (DRJ). To evaluate the performance of the proposed DRJ for high-order random walks, we train Node2vec [11] with *vanilla* rejection sampling and our proposed *DRJ* respectively on public graph datasets Cora and Wiki, and report the efficiency results in Table 4. Specifically, we train Node2vec for 50 epochs on each dataset with random walks of fixed length 80. While fixing $p = 1$, we vary q parameter in Node2vec with 1024 and 1/1024 to test two extreme cases. In Table 4, the training time in seconds and the average number of sampling trials required per acceptance are reported for vanilla Node2vec and Node2vec with DRJ. Observe that Node2vec with DRJ is significantly faster than vanilla Node2vec, by up to orders of magnitude. For instance, on Wiki dataset with $q = 1/1024$, the training time with DRJ is 1.54 seconds, more than 20 times faster than vanilla Node2vec. The average sampling trials by DRJ is also fewer than vanilla Node2vec, which explains the efficiency improvement achieved by DRJ.

Table 5: GraphDSSM Evaluation.

Method	HR@50(%)	HR@100(%)
DSSM	20.84	30.51
GraphDSSM	22.87	33.14
Rel. Imp.	+9.74%	+8.62%

Table 6: GraphDeepFM Evaluation.

Method	AUC(%)	Rel. Imp.
DeepFM	78.01	0.00
DeepFM+node embeddings	78.03	+0.025%
GraphDeepFM	**78.42**	+0.525%

7.3 Algorithm Evaluation

We have built all the models in 4 categories in Table 1 into EmbedX. In this section, we provide the experimental evaluation on two representative in-house models GraphDSSM and GraphDeepFM, which are new JLSG models to handle both high-dimensional sparse data and graph data (Section 6). In experiments, we randomly extract 90% of the data for training and use the remaining data as test data. We set the dimension of embedding vectors as 128.

GraphDSSM. We compare the effectiveness of GraphDSSM with DSSM on Tencent-1 dataset. We use Hit Ratio at K (HR@K) as the evaluation metrics which measures the percentage of users for whom the clicked item is among the top K items recommended by the model. As shown in Table 6, compared with DSSM that only utilizes high-dimensional sparse data, the relative improvements achieved by GraphDSSM is 9.74% on HR@50 and 8.62% on HR@100. By leveraging both graph data and sparse data and trained in an end-to-end manner, GraphDSSM achieves superior performance in retrieving, which validates the importance of using both types of data in production for higher business values.

GraphDeepFM. We compare our proposed GraphDeepFM with vanilla DeepFM, as well as DeepFM incorporated with pre-trained node embeddings by GraphSAGE (DeepFM+node embeddings) on Tencent-2 dataset. The AUC scores of the three methods are reported in Table 6. Observe that GraphDeepFM that holistically integrates high-dimensional sparse data and graph data achieves the highest AUC score 78.42%. GraphDeepFM achieves better AUC than DeepFM by relative improvement 0.525%. DeepFM that simply appends node embeddings generated from unsupervised GNNs achieves AUC 78.03% with limited relative improvement. The improvement of GraphDeepFM is mainly attributed to the multi-task learning design of GraphDeepFM which co-trains DeepFM and network embedding module in Section 6.2.

7.4 Use Cases In Production

We have deployed EmbedX in production for four years. It has achieved superior performance in multiple business sectors in Tencent. We select 3 representative applications supported by EmbedX and report the A/B test of EmbedX in Table 7.

(1) **News feed** continuously updates stream of news that is tailored to the interests of users. As a ranking task, the evaluation metric is AUC in production. EmbedX improves AUC by 3.06% with the adaptation of our JLSG model GraphDeepFM which makes use of both sparse data and graph data.

Table 7: A/B tests on 3 applications in Tencent.

Application	Model	Metrics	Rel. Imp.
News Feed	GraphDeepFM	AUC	+3.06%
Music Recommendation	DeepFM	Retention Time	+6.09%
Malicious accounts	GraphSAGE	Recall	+21.9%

(2) **Music recommendation** aims at recommending songs that users are likely to listen. The key business metric is the average time per person spent on listening (Average Retention Time). EmbedX improves the metric by 6.09%, while reducing the percentage of users who skip a song before it finishes playing.
(3) **Malicious accounts discovery** refers to the identification of user accounts associated with illegal or unethical activities, such as cybercrime, illegal gambling, or counterfeit goods production in Tencent. Recall is the key metric to evaluate this service, and it measures the percentage of true malicious accounts that are correctly identified by the model. EmbedX improves Recall by 21.9% by leveraging graph data for detection.

8 RELATED WORK

8.1 High-Dimensional Sparse Data

High-dimensional sparse data is important for many applications, including recommendation systems, advertisement, CTR prediction, etc. For instance, deep learning based recommendation models [7, 8, 12, 52] have attracted much research attention [54]. As explained in Section 2, an entity usually has high-dimensional sparse features and dense features, both of which are trained to convert to low-dimensional embedding vectors via deep learning techniques, to facilitate downstream tasks [20, 29]. YoutubeDNN [8] uses two sub-modules to extract features from video-related data and user-related data respectively, and a third module is adopted to combine the two features for personalized video recommendations. DSSM [17] is a retrieval model follows two-tower architecture which encodes sparse and dense features into user and item embeddings respectively. Relevant items can be retrieved efficiently using nearest search algorithms in inference stage. DeepFM [12] combines factorization machines and MLPs to extract both low-order and high-order feature interactions between different types of features.

With the ever-increasing data scale and model size in modern online services, the offline training and online inference of deep learning models on high-dimensional sparse data (DLS) have become time-consuming. There are existing systems [20, 29, 33, 36] to handle DLS models via dedicated system designs and optimizations, in order to efficiently process the industrial billion-scale sparse data and handle big models with billions of parameters [29, 37]. For instance, XDL [20] is designed to specially handle sparse features and optimized for communications between different types of servers. HET [33] adopts an embedding-cache-enabled architecture to efficiently process frequently updated embeddings, to reduce time overheads. Perisa [29] consists of optimization algorithms with the designs to consider the differences in training sparse embedding layers and dense neural networks, and also includes a distributed system architecture for scalability. Neo [36] is developed with parallelism training for embedding tables and optimized embedding operators for training. Note that these systems are designed only for high-dimensional sparse data, but not for graph data.

8.2 Network Embedding and GNNs

Graphs are ubiquitous to model the multi-hop relationships between entities. It has attracted great attention to develop network embedding (NE) methods [10, 11, 41, 42, 47] and graph neural networks (GNNs) [5, 14, 23, 24, 35, 46, 48, 49]. The objective is to learn low-dimensional representations capturing the underlying structure and semantic information in graphs. The learned representations can be used for node classification, link prediction, etc. Popular NE methods are based on random walks [13]. For example, DeepWalk [41] and Node2vec [11] perform random walks on graphs to train SkipGram model to generate node embeddings. Metapath2vec [10] adopts meta-path-based walks to capture semantics in heterogeneous graphs. EGES [47] and struc2vec [42] first construct item graph and multi-layer graph respectively, and random walk sampling is then performed on the constructed graphs. In terms of GNNs, GCN [23] performs convolutions using graph Laplacian matrix. GraphSAGE adopts message passing framework for aggregate neighborhood representations inductively [14]. GAT [46] further uses self-attention mechanism to give neighbor nodes different weights. PinSAGE [53] samples subgraph using random walk for efficient model training on web-scale graphs. Meanwhile, as the scale of graphs grows to be with billions, even hundreds of billions of nodes and edges, graph learning systems have been developed to scale NE and GNN models on web-scale graphs for training and inference [3, 30, 39, 59]. To our knowledge, these systems are developed to handle graphs, while they are non-trivial to be extended for high-dimensional sparse data.

Summing up, as reviewed in Sections 8.1 and 8.2, there is a lack of industrial-level frameworks to seamlessly support both high-dimensional sparse data and graph data. Our proposed EmbedX fills the gap through thorough architectural developments, system optimizations, and algorithmic designs. EmbedX also provides the opportunity for developing joint learning models to learn more expressive embeddings from the two types of data.

9 CONCLUSION

We present an industrial distributed learning system from Tencent, EmbedX, which is designed to support a versatile collection of embedding models on large-scale high-dimensional sparse data and graph data. EmbedX is able to handle billion-scale online service data in an efficient and scalable manner. To achieve this, EmbedX consists of novel system designs and efficient parameter operators and graph operators. EmbedX is powerful to support 4 categories of algorithms, namely DLS, NE, GNN, and JLSG methods. Extensive experiments on massive Tencent data and public data validate the superiority of EmbedX. EmbedX is currently deployed in Tencent for various business lines. Real use cases with A/B test results in production further demonstrate the power of EmbedX.

ACKNOWLEDGMENTS

This work is supported by Hong Kong RGC ECS No. 25201221, and National Natural Science Foundation of China No. 62202404. This work is also supported by a collaboration grant from Tencent Technology (Shenzhen) Co., Ltd (P0039546). This work is supported by a startup fund (P0033898) from Hong Kong Polytechnic University and project P0036831.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283.

[2] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2021. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 802–814.

[3] Alibaba. 2023. Alibaba/euler: A distributed graph deep learning framework. Retrieved July 5, 2023 from https://github.com/alibaba/euler

[4] Léon Bottou and Olivier Bousquet. 2007. The Tradeoffs of Large Scale Learning. In *Advances in Neural Information Processing Systems 20, Proceedings of the Twenty-First Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 3-6, 2007*. 161–168.

[5] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks?. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[7] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[8] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[9] Criteo. 2023. Download Criteo 1TB click logs dataset. Retrieved July 5, 2023 from https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

[10] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 135–144.

[11] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[12] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 1725–1731.

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Eng. Bull.* 40, 3 (2017), 52–74.

[14] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 1024–1034.

[15] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[16] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*. 2704–2710.

[17] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 2333–2338.

[18] Zhipeng Huang and Nikos Mamoulis. 2017. Heterogeneous information network embedding for meta path based proximity. *arXiv preprint arXiv:1701.05291* (2017).

[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.

[20] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. 2019. XDL: an industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*. 1–9.

[21] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *Technical Report* (1997).

[22] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[23] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[24] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then Propagate: Graph Neural Networks meet Personalized PageRank. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[25] Aaron Q Li, Amr Ahmed, Sujith Ravi, and Alexander J Smola. 2014. Reducing the sampling complexity of topic models. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 891–900.

[26] Chao Li, Zhiyuan Liu, Mengmeng Wu, Yuchi Xu, Huan Zhao, Pipei Huang, Guoliang Kang, Qiwei Chen, Wei Li, and Dik Lun Lee. 2019. Multi-interest network with dynamic routing for recommendation at Tmall. In *Proceedings of the 28th ACM international conference on information and knowledge management*. 2615–2623.

[27] Mu Li, David G. Andersen, Alexander J. Smola, and Kai Yu. 2014. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger (Eds.). 19–27.

[28] Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018. Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 3538–3545.

[29] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. 2022. Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 3288–3298.

[30] Dandan Lin, Shijie Sun, Jingtao Ding, Xuehan Ke, Hao Gu, Xing Huang, Chonggang Song, Xuri Zhang, Lingling Yi, Jie Wen, et al. 2022. PlatoGL: Effective and Scalable Deep Graph Learning System for Graph-enhanced Real-Time Recommendation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 3302–3311.

[31] Xiao Ma, Liqin Zhao, Guan Huang, Zhi Wang, Zelin Hu, Xiaoqiang Zhu, and Kun Gai. 2018. Entire space multi-task model: An effective approach for estimating post-click conversion rate. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. 1137–1140.

[32] Matt Mahoney. 2023. Large text compression benchmark. Retrieved July 5, 2023 from http://www.mattmahoney.net/dc/text.html

[33] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: scaling out huge embedding model training via cache-enabled distributed framework. *Proceedings of the VLDB Endowment* 15, 2 (2021), 312–320.

[34] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[35] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. 2019. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 4602–4609.

[36] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 993–1011.

[37] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, et al. 2021. High-performance, distributed training of large-scale deep learning recommendation models. *arXiv preprint arXiv:2104.05158* (2021).

[38] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*

(2019).

[39] PaddlePaddle. 2023. Paddlepaddle/PGL: Paddle Graph Learning (PGL) is an efficient and flexible graph learning framework based on paddlepaddle. Retrieved July 5, 2023 from https://github.com/PaddlePaddle/PGL

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035.

[41] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[42] Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. 2017. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 385–394.

[43] Stergios Stergiou, Zygimantas Straznickas, Rolina Wu, and Kostas Tsioutsiouliklis. 2017. Distributed Negative Sampling for Word Embeddings. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder Singh and Shaul Markovitch (Eds.). AAAI Press, 2569–2575.

[44] Guolei Sun and Xiangliang Zhang. 2017. Graph embedding with rich information through heterogeneous network. *arXiv preprint arXiv:1710.06879* (2017).

[45] Ke Tu, Peng Cui, Xiao Wang, Fei Wang, and Wenwu Zhu. 2018. Structural Deep Embedding for Hyper-Networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 426–433.

[46] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[47] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 839–848.

[48] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In *International conference on machine learning*. PMLR, 6861–6871.

[49] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[50] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. Knightking: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 524–537.

[51] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. 2016. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*. PMLR, 40–48.

[52] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed Chi. 2019. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 269–277.

[53] 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.

[54] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–38.

[55] Weinan Zhang, Jiarui Qin, Wei Guo, Ruiming Tang, and Xiuqiang He. 2021. Deep Learning for Click-Through Rate Estimation. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Zhi-Hua Zhou (Ed.). ijcai.org, 4695–4703.

[56] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. 2019. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 43–51.

[57] Dongyan Zhou, Songjie Niu, and Shimin Chen. 2018. Efficient graph computation for Node2Vec. *arXiv preprint arXiv:1805.00280* (2018).

[58] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1059–1068.

[59] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105.