EFFICIENT DYNAMIC GRAPH REPRESENTATION LEARNING AT SCALE

A PREPRINT

Xinshi Chen[1], **Yan Zhu**[2], **Haowen Xu**[1], **Mengyang Liu**[1], **Liang Xiong**[2], **Muhan Zhang**[2], **Le Song**[1,3]
[1]Georgia Institute of Technology, [2]Facebook, [3]MBZUAI

December 15, 2021

ABSTRACT

Dynamic graphs with ordered sequences of events between nodes are prevalent in real-world industrial applications such as e-commerce and social platforms. However, representation learning for dynamic graphs has posed great computational challenges due to the time and structure dependency and irregular nature of the data, preventing such models from being deployed to real-world applications. To tackle this challenge, we propose an efficient algorithm, **E**fficient **D**ynamic **G**raph l**E**arning (**EDGE**), which selectively expresses certain temporal dependency via training loss to improve the parallelism in computations. We show that **EDGE** can scale to dynamic graphs with millions of nodes and hundreds of millions of temporal events and achieve new state-of-the-art (SOTA) performance.

1 Introduction

Graph representation learning is becoming increasingly popular in addressing many graph-related applications, such as social networks, recommendation systems, knowledge graphs, biology, and so on [1, 2, 3]. A substantial body of works has focused on *static* graph neural networks (GNNs) [4, 5, 6, 7, 8, 9], but many real-world interactions are temporal, making *dynamic* graph a more preferable model. Dynamic graphs explicitly model both the time- and node-dependency of interactions, allowing them to better represent temporal evolutions over graphs while also respecting the dynamic nature of the data. However, unlike the active line of research on efficient algorithms for training static GNNs [10, 11, 12, 13], existing works on dynamic graph models could only be applied to small graphs.

In some sense, dynamic graphs combine the properties of sequence models and GNNs. While enjoying the representation power of both, it suffers from significantly more computational challenges. On the one hand, unlike sequence models that treat the interaction events between different nodes independently, dynamic graph models, however, introduce node dependencies, which prevents us from processing events on different nodes parallelly. On the other hand, static GNNs perform synchronized updates over the nodes, which is not allowed in the dynamic counterpart due to the time ordering constraints.

To be more specific, many dynamic graph models represent the state of a node $u \in \mathcal{V}$ at time t using a low-dimensional vector $\texttt{emb}(u,t) \in \mathbb{R}^d$. To model temporal evolutions, whenever an interaction event $e = (u_1, u_2, t, r)$ occurs between nodes u_1 and u_2 at time t, with label r, the embeddings of involved nodes u_1 and u_2 will be updated by a neural operator $\mathcal{F}_\theta$ in the following form:

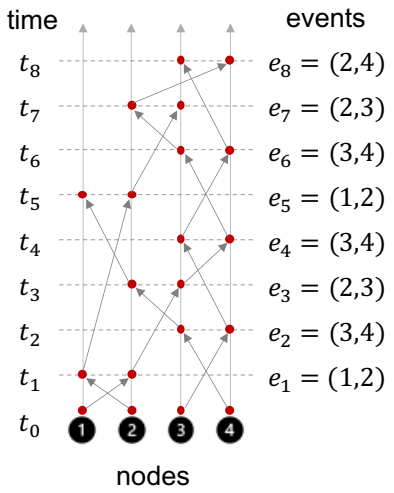


Figure 1: A simple example: the computational DAG of a dynamic graph model. Each red dot •, which corresponds to a node $u \in \{1,2,3,4\}$ and a time $t \in \{t_0, \cdots, t_8\}$, indicates the computation of the temporal node embedding $\texttt{emb}(u,t)$. The edges $\longrightarrow$ indicate the dependencies of the computations.

$$\texttt{emb}(u_1,t), \texttt{emb}(u_2,t) \leftarrow \mathcal{F}_\theta\left(\texttt{emb}(u_1,t^-), \texttt{emb}(u_2,t^-), \text{feature}(e)\right), \tag{1}$$

where the notation $\texttt{emb}(u, t^-)$ represents the most recent state of the node u before time t. Given a sequence of interaction events, **what is the computational complexity of computing all the temporal embeddings** $\texttt{emb}(u, t)$**?**

As an example, Fig. 1 shows the computational graph of the embedding updates for 8 interaction events that occurred between 4 nodes. To avoid ambiguity in terminology, we refer to a computational graph as a *computational DAG* (directed acyclic graph) throughout this paper because it must be directed acyclic.

A naive approach can compute Eq. 1 sequentially according to the time ordering of interactions, which will require the *number of interactions* many computational steps. To improve the computational complexity, a recent advance JODIE [14] has proposed to parallelize independent computations. For example, the interaction events $e_1 = (1, 2)$ and $e_2 = (3, 4)$ in Fig. 1 are independent since their involved nodes do not overlap, so JODIE will perform the updates for e_1 and e_2 parallelly (please compare Fig. 2 (left) to Fig. 1). However, the effectiveness of this strategy heavily depends on the dataset's property - to what the interaction events can be parallelized, which reveals to be far from satisfactory on real-world datasets due to the presence of many actively interacted and long-lived nodes.

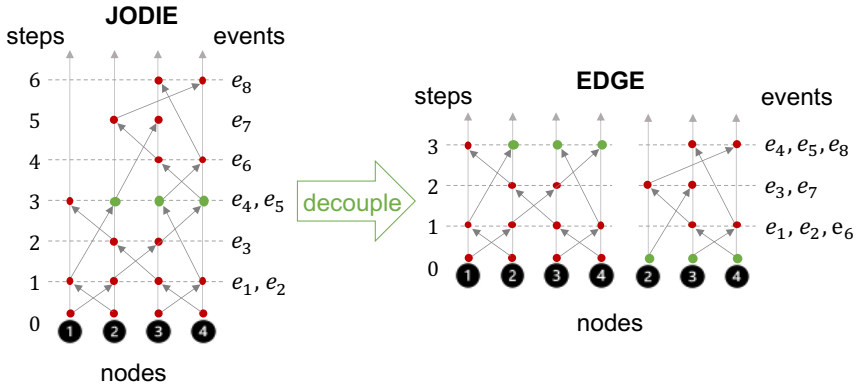


Figure 2: Parallelization for the example in Fig 1. JODIE takes 6 steps for the 8 events in Fig 1. EDGE decouples 3 temporal nodes (green) from the computational graph, and treat the decoupled nodes as independent of their ancestors. Then it takes 3 steps in total to compute all temporal embeddings.

To address the computational barriers that prevent the use of dynamic graph models from industrial-scale datasets, in this paper, we propose **EDGE**, which stands for **E**fficient **D**ynamic **G**raph l**E**arning.

The design of **EDGE** is motivated by our key finding that the computational complexity is restricted by the **longest path** in the computational DAG - no matter how we optimize the parallelization of updates within a step, it takes at least the length of the longest path many sequential steps in the end. Section 3 will go over this in greater detail. As a simple example, the computational DAG in Fig. 2 (left) has a longest path of length 6, so JODIE cannot take less than 6 steps to finish the computations. The longest path in the computational DAG represents the limit that JODIE and any other methods can reach, so EDGE is designed to carefully reduce its length from two aspects, as briefly summarized below.

First, EDGE selectively expresses some computational dependencies via the training loss to achieve better parallelism in computation. To be more specific, EDGE decouples some nodes (called **d-nodes**) from the computational DAG to remove certain computational dependencies; and these dependencies will be added back via the training loss. For instance, the d-nodes (green-colored ●) in Fig. 2 are divided into two nodes each, which separates a d-node's descendants from its ancestors, shortening the longest path in the computational DAG. However, a naive decoupling strategy could lead to a sub-optimal model due to the ignored dependencies. In Section 3, we will explain how we carefully design the decoupling strategy by answering the following two questions, which are critical to its success.

- How to select the most effective set of d-nodes for shortening the longest path?
- How to modify the training loss to compensate for the removed computational dependencies caused by the d-nodes?

Second, EDGE assumes the ***convergent states*** *of actively interacted nodes and models them by static embeddings.* As visualized in Fig. 3, many realistic datasets contain scale-free interaction networks [15], where most nodes have a small number of interactions but a few high-degree nodes are actively interacted. Examples include celebrities on Twitter who have lots of followers, and popular items on Amazon which are frequently clicked. While these active nodes contribute a large amount of edges to the graph, their embedding states tend to be static. The reasoning behind is mainly two-fold. First, an active node's interactions with other nodes give us *diminishing* returns in terms of their information value in understanding this node's characteristics. Second, from the optimization perspective, a recurrent operator such as $\mathcal{F}_\theta$ in Eq. 1 often has *smaller derivatives* in deeper layers, which is also the cause of the gradient vanishing problem [16]. With the operator applied repeatedly, embeddings of active nodes will eventually evolve slowly. With this motivation, EDGE represents active nodes by node-specific *static embeddings*

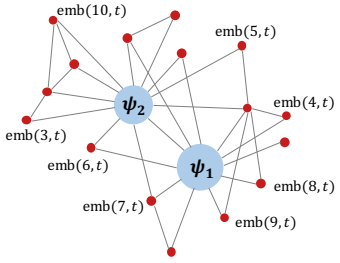


Figure 3: Scale-free network.

$\boldsymbol{\psi}_u$ during training, while keeping other nodes dynamic. The effect from other nodes to active nodes are implicitly incorporated through the gradient updates during training. After being trained, the operator $\mathcal{F}_\theta$ could still be applied to active nodes during testing phase. This static treatment of active nodes proves to be very effective on realistic datasets.

In summary, these two designs jointly reduce the path length in the training computational graph, making EDGE scalable to realistic datasets. We conduct comprehensive experiments on several user-item interaction datasets, including two large-scale datasets, to compare EDGE to a wide range of baselines, showing the SOTA performances of EDGE in both accuracy and prediction efficiency.

Related work. Although learning on dynamic graphs is relatively recent [17, 18, 19, 20, 21], various models were actively proposed to represent temporal patterns via the time ordering of interaction events [22, 23, 14] or from temporal aligned graph snapshots [18, 19]. A large subset of such models that updates node-wise embeddings through a neural network when new interactions appear [14, 22, 23, 24, 25] is most relevant to this paper. They vary from each other mainly by designing different architectures and incorporating different information. The method proposed in this paper could be generally applied to improve the training efficiency of these models. For a more comprehensive summary of other dynamic models, we refer the audience to [20] which summarizes different models via a general framework.

2 Model

In this paper, we focus on modeling the interactions between two groups of nodes, $\mathcal{V}_1$ and $\mathcal{V}_2$, but the technique could easily be applied to other scenarios involving fewer or more nodes in each temporal event.

We denote an interaction event between nodes $u_1 \in \mathcal{V}_1$ and $u_2 \in \mathcal{V}_2$ by $e = (u_1, u_2, t, r)$, where t is the time and r is a label, which can indicate click/non-click in user-item interaction networks, for example. In this section, we will illustrate how a dynamic graph model is used to describe the evolution of node embeddings over time when a sequence of interaction events are observed.

2.1 Embedding Evolution

In many dynamic graph models, two crucial components for modeling the evolution of node embeddings are

1. the initial embeddings $\texttt{emb}(u, 0)$ of the nodes; and
2. the neural operator $\mathcal{F}_\theta$ for updating them (via Eq. 1).

In EDGE, a key consideration is the convergent states of active nodes. Therefore, it brings in a new component:

3. **the node-specific static embeddings for active nodes**.

To be more specific, Given a degree threshold n_1^*, we say a node u_1 is active if it has more than n_1^* interactions in the training set. We can denote the set of active nodes by $\mathcal{V}_1^{act} \subseteq \mathcal{V}_1$ and define $\mathcal{V}_2^{act} \subseteq \mathcal{V}_2$ in a similary way based on a threshold n_2^*. In EDGE, these active nodes' embeddings are modeled by static embeddings

$$\texttt{emb}(u, t) = \boldsymbol{\psi}_u, \quad \forall t \leq T_{train}, \quad \forall u \in \mathcal{V}_1^{act} \cup \mathcal{V}_2^{act}, \tag{2}$$

where each $\boldsymbol{\psi}_u$ is a learnable vector, and T_{train} is the final time-stamp in the training dataset. The other 'inactive' nodes are evolved by the neural operator $\mathcal{F}_\theta$ when the interactions $e = (u_1, u_2, t, r)$ occur as follows.

$$\forall t \leq T_{train}, \quad \forall u_1 \in \mathcal{V}_1 \setminus \mathcal{V}_1^{act}$$

$$\texttt{emb}(u_1, 0) = \boldsymbol{\xi}_1 \tag{3}$$

$$\texttt{emb}(u_1, t) \leftarrow \mathcal{F}_{\theta_1}\left(\texttt{emb}(u_1, t^-), \texttt{emb}(u_2, t^-), \text{feature}(e)\right) \tag{4}$$

$$\forall t \leq T_{train}, \quad \forall u_2 \in \mathcal{V}_2 \setminus \mathcal{V}_2^{act}$$

$$\texttt{emb}(u_2, 0) = \boldsymbol{\xi}_2 \tag{5}$$

$$\texttt{emb}(u_2, t) \leftarrow \mathcal{F}_{\theta_2}\left(\texttt{emb}(u_2, t^-), \texttt{emb}(u_1, t^-), \text{feature}(e)\right), \tag{6}$$

where $\boldsymbol{\xi}_1, \boldsymbol{\xi}_2$ are learnable vectors that define the common initializations for inactive nodes in $\mathcal{V}_1$ and $\mathcal{V}_2$, and we use different parameters θ_1 and θ_2 in the operator $\mathcal{F}$ that updates these two groups of nodes. The architecture of $\mathcal{F}$ is not the main focus of this paper, so we simply adopt a gated recurrent unit (GRU) [26] based architecture. The notion $\text{feature}(e)$ can be any available event features. Examples include contents of posts on Twitter, static features of items on shopping platforms, etc. Lastly, the notation t^- in (u, t^-) refers to the time-stamp of the last event occured to u before t.

Eq. 2 to Eq. 6 jointly define the overall embedding evolution model in EDGE. A few remarks and important discussions for this model are presented in the following.

Efficiency gain. The reduction of computational dependencies achieved by representing active nodes using static embeddings is significant, because the active nodes contribute a substantial number of edges to the network. The only computational sacrifice here is the memory cost for introducing the additional set of learnable vectors $\{\boldsymbol{\psi}_u\}$ for the active nodes. However, in our experiments, this does not cause any problems. Every transductive graph neural network, by comparison, will require a memory cost greater than that.

Accuracy. The static treatment of active nodes does damage the model's performance, which will be verified by extensive experiments in Section 4. This static strategy is well-thought-out and supported by a number of arguments below.

1. If we view the embedding evolution as a refresh of our understanding about the features of a node, the information value of a new interaction for an active node is minimal.
2. Optimization is as important as the model. With the reduced computational dependencies, the training optimization becomes easier, which might also be the reason why we often observe an improvement in model performance in our experiments.
3. From the view of variance-bias trade-off, it is reasonable to use a transductive (i.e., node-specific) embedding for active nodes and use an inductive model for inactive nodes.
4. During the test phase, we could still apply the learned operator $\mathcal{F}_\theta$ to evolve the embeddings of active nodes according to their newly observed events.

2.2 Prediction Model and Loss Function

After computing the node embeddings, a predictive model extras useful features to archieve the desired prediction. Our experiments are mainly about predicting the next interacted node in $\mathcal{V}_2$ for the nodes in $\mathcal{V}_1$, so we employ a neural layer $\mathcal{H}_\theta$ to predict the embedding of the next interacted node. Here we abuse the notation θ to generally represent parameters in neural networks. Inspired by the design in [14], we employ a prediction layer in the following form:

$$\boldsymbol{h}_{u_1,t} \leftarrow \mathcal{H}_\theta(\texttt{emb}(u_1,t), \text{feature}(u_1), \text{feature}(\texttt{node}(u_1,t^-))),$$

where $\texttt{node}(u_1,t^-) \in \mathcal{V}_2$ is the most recently interacted node of u_1 before time t. Here feature$(\cdot)$ denotes any available node features. In our experiments, $\mathcal{H}_\theta$ is simply a multi-layer perceptron (MLP).

With the prediction embedding $\boldsymbol{h}_{u_1,t}$, one can then measure its similarity to embeddings of nodes in $\mathcal{V}_2$. In our experiments, we measure its similarity $s_t(u_1,u_2)$ to a node u_2 using inner product, and use cross-entropy loss (equivalently, the softmax loss) for each event e in the training set $\mathcal{D}_{train}$:

$$\mathcal{L}(e=(u_1,u_2,t,r)\,;\Theta) = -\log\left(\frac{\exp\left(s_t(u_1,u_2)\right)}{\sum_{u_2'\in\mathcal{O}(u_1,t)}\exp\left(s_t(u_1,u_2')\right)}\right) \tag{7}$$

where $\Theta := \{\theta_1,\theta_2,\boldsymbol{\xi}_1,\boldsymbol{\xi}_2,\{\boldsymbol{\psi}_u : u\in\mathcal{V}_1^{act}\cup\mathcal{V}_2^{act}\}\}$ includes all parameters. $\mathcal{O}(u_1,t)$ is a set of 'negative' nodes that u_1 did not interact with at t. In our experiments, this set is randomly sampled from $\mathcal{V}_2$.

3 Efficient Training Algorithm

The bottleneck of previous training methods is the under-utilization of GPUs since the computation is hard to parallelize. We find that the computational complexity is limited by the longest path in the computational DAG, which motivates our design of *d-nodes* for shortening it. We will describe the details in this section.

3.1 Why Longest Path?

Why is the computational complexity limited by the longest path in the computational DAG? To answer this question, we would like to refer the audience to the literature of parallel algorithm analysis. Briefly speaking,

- The so-called *work-depth model* of parallel computation was originally introduced by [27] in 1982, which has been used for many years to describe the performance of parallel algorithms.
- *Depth* is defined as the longest chain of sequential dependencies in the computation [28].
- Minimizing the depth is important in designing parallel algorithms, because the depth determines the *shortest possible execution time* [29].

When the computations of a dynamic graph are viewed as the operations in a parallel algorithm, then the "longest path in the computational DAG" corresponds to the "depth" in the work-depth model, which determines the shortest possible execution time.

3.2 A Simplified Illustration of D-nodes

Before going into the details of the **d-nodes**, we would like to first explain the high-level idea through a highly simplified example. Instead of the very complex dynamic graph model, consider a 6-layer feed-forward network $\sigma(w_6\,\sigma(w_5\,\sigma(w_4\,\sigma(w_3\,\sigma(w_2\,\sigma(w_1 x))))))$.
Clearly, its computational DAG is a path

$$x \to h_1 \to h_2 \to h_3 \to h_4 \to h_5 \to h_6$$

where $h_i = \sigma(w_i h_{i-1})$ is the i-th hidden layer output. Clearly, it requires 6 sequential steps to finish the computations layer by layer. However, we can speed it up to 3 sequential steps by decoupling this path through the node h_3.

More precisely, we **select h_3 as the d-node and add a learnable vector ϕ_3 for it**. Then we can perform the computations $h_3 = \sigma(w_3\,\sigma(w_2\,\sigma(w_1 x)))$ and $h_6 = \sigma(w_6\,\sigma(w_5\,\sigma(w_4\phi_3)))$ parallelly. The new computational DAG becomes 2 independent paths:

$$x \to h_1 \to h_2 \to h_3$$
$$\phi_3 \to h_4 \to h_5 \to h_6$$

and their lengths are 3. The key is that ϕ_3 is a learnable vector that does not depend on any other nodes, so we can start to use ϕ_3 for computing h_4, h_5, h_6 before we obtain h_3.

The final problem is, with this decoupling, it seems the model becomes different from the original 6-layer feed-forward network. Fortunately, to maintain the consistency, we only need to enforce the constraint $h_3 = \phi_3$ during the optimization. Apparently, if $h_3 = \phi_3$ is satisfied, the models before and after the decoupling are equivalent. Therefore, we can perform gradient updates to optimize both the network parameters $\boldsymbol{w} = (w_1, \cdots, w_6)$ and ϕ_3 by solving the following constrained optimization

$$\min_{\boldsymbol{w},\phi_3} \text{loss}(h_6) \quad \text{subject to} \quad h_3 = \phi_3. \tag{8}$$

Experimentally, we enforce the constraint softly by:

$$\min_{\boldsymbol{w},\phi_3} \text{loss}(h_6) + \frac{\alpha}{2}\|h_3 - \phi_3\|_2^2. \tag{9}$$

To summarize, the key idea of d-nodes is to **selectively express some dependencies in computation by dependencies in optimization (via training loss)**.

In this example, the computational DAG simply consists of paths. In the case of a dynamic graph, the computational DAG is far more complex. However, the algorithm follows the same logic. We will present the details in the following sub-sections.

3.3 Step 1: Selection of D-nodes

In the example in Section 3.2, the computational DAG is a path, so selecting the mid-point h_3 as the d-node is very effective in shortening the path. However, in the case of dynamic graph models, the computational DAG is a lot more complex due to the node dependency and time dependency (e.g., Fig. 1).

Ideally, we should choose a small number of d-nodes that can effectively shorten the longest path in the computational DAG. Therefore, in this section, we explore the answer of:

Given a limited quota K, how to choose the most effective collection of K d-nodes?

To begin with, we construct the computational DAG for computing all temporal node embeddings $\texttt{emb}(u,t)$ through the outlined steps in Algorithm 1 and denote it by $\texttt{CompG} = (\mathcal{V}_{\text{CG}}, \mathcal{E}_{\text{CG}})$. The goal is to decouple a subset of K nodes $\mathcal{V}_{\text{CG}}^D \subseteq \mathcal{V}_{\text{CG}}$ from the computational DAG to shorten the longest path.

Mathematically, given the computational DAG, $\texttt{CompG} = (\mathcal{V}_{\text{CG}}, \mathcal{E}_{\text{CG}})$, the d-node selection problem can be formulated as the following minmax integer programming (IP):

$$\min_{\{d_i\in\{0,1\},\ell_i\in\mathbb{N}\}_{i\in\mathcal{V}_{\text{CG}}}} \left\{\max_{i\in\mathcal{V}_{\text{CG}}} \ell_i\right\}$$
$$\text{s.t.} \begin{cases} \ell_i \geq (1-d_j)\ell_j + 1 & \forall (j,i) \in \mathcal{E}_{\text{CG}} \\ \sum_{i\in\mathcal{V}_{\text{CG}}} d_i \leq K \end{cases},$$

Algorithm 1: Build computational DAG

$\mathcal{V}_{\text{CG}} = \mathcal{V}_1^{act} \cup \mathcal{V}_2^{act} \cup \{(u,0) : u \in \mathcal{V}_1 \setminus \mathcal{V}_1^{act} \cup \mathcal{V}_2 \setminus \mathcal{V}_2^{act}\}$;
$\mathcal{E}_{\text{CG}} = \emptyset$;
for $e = (u_1, u_2, t, r) \in \mathcal{D}_{train}$ **do**
    Execute for both $i = 1$ and $i = 2$:
    if $u_i \in \mathcal{V}_i \setminus \mathcal{V}_i^{act}$ **then**
        Add the node (u_i, t) to $\mathcal{V}_{\text{CG}}$;
        Add the edge $((u_i, t^-), (u_i, t))$ to $\mathcal{E}_{\text{CG}}$;
        $j = 3 - i$;
        if $u_j \in \mathcal{V}_j \setminus \mathcal{V}_j^{act}$ **then**
            Add the edge $((u_j, t^-), (u_i, t))$ to $\mathcal{E}_{\text{CG}}$;
        else
            Add the edge $(u_j, (u_i, t))$ to $\mathcal{E}_{\text{CG}}$;

Output: $\texttt{CompG} = (\mathcal{V}_{\text{CG}}, \mathcal{E}_{\text{CG}})$;

Algorithm 2: Select K d-nodes from `CompG`

Input: $\mathcal{V}_{\text{CG}}, \mathcal{E}_{\text{CG}}, K$;
Initialize $\mathcal{V}_{\text{CG}}^D = \emptyset$;
while $k < K$, k++ **do**
    Find a longest path p in the DAG $(\mathcal{V}_{\text{CG}}, \mathcal{E}_{\text{CG}})$;
    Find the center node i in the path p;
    Add i to $\mathcal{V}_{\text{CG}}^D$;
    Add a new node i' to $\mathcal{V}_{\text{CG}}$;
    for $j \in \{j | (i,j) \in \mathcal{E}_{\text{CG}}\}$ **do**
        Remove edge (i,j) from $\mathcal{E}_{\text{CG}}$;
        Add edge (i', j) to $\mathcal{E}_{\text{CG}}$;

Output: $\mathcal{V}_{\text{CG}}^D$;
Output: New graph $\texttt{D-CompG} = (\mathcal{V}_{\text{CG}}, \mathcal{E}_{\text{CG}})$;

where the binary variable d_i indicates whether the node $i \in \mathcal{V}_{\text{CG}}$ is selected as a d-node, and ℓ_i measures the longest distance between node i and root nodes in the decoupled graph.

Given the extremely large problem size of this IP (the number of nodes in $\mathcal{V}_{\text{CG}}$ is proportional to the number of events in the dataset), we cannot use existing packages such as Gurobi [30] to solve it. Therefore, we design a greedy heuristic to give an approximate solution, which reveals to be effective in experiments.

The algorithm steps are outlined in Algorithm 2. Briefly speaking, it repeatedly finds the middle-point of the longest path in the computational DAG, takes it as a d-node, and then adjusts edges from this node, until K d-nodes are found. The idea of this algorithm is similar to alternative minimization maximization, but solving a sub-problem each time sequentially for $k = 1, \cdots, K$:

(i) *Optimize with fixed* $\{d_i\}$: Suppose values of $\{d_i\}$ are given by current selections $\{i^1, \cdots, i^{k-1}\}$. Then the node that optimizes the inter maximization $i^* = \arg\max_{i \in \mathcal{V}_{\text{CG}}} \ell_i$ is apparently in the longest path p in Algorithm 2. Furthermore, for all edges in this path, the corresponding constraints in the IP are binding (i.e., equality $\ell_i = (1 - d_j)\ell_j + 1$ holds for all $(j,i) \in p$), and removing all other constraints does not change the optimal value ℓ_{i^*}. Therefore, we select the next d-node by solving a sub-problem identified by this longest path p.

(ii) *Select* i^k *on sub-problem*: This step selects a d-node i^k and makes $d_{i^k} = 1$, which is accomplished by solving the IP with the subset of constraints identified by edges in the longest path p. As pointed out in Algorithm 2, the i^k that optimizes the sub-problem is the center node in the path p.

It is notable that the longest path of a DAG can be found in linear time, and Algorithm 2 only needs to run once as a pre-processing step.

3.4 Step 2: Express Dependency by Constraints

From now on, we index each selected d-node by $(u,t) \in \mathcal{V}_{\text{CG}}^D$ since it has a unique correspondence to a temporal state $\texttt{emb}(u,t)$.

Each d-node (u,t) plays a role of detaching the events happened to u after t from those before t. Similar to adding the vector ϕ_3 for h_3 in Section 3.2, the decoupling is realized by creating an *additional embedding* $\boldsymbol{\phi}_{u,t}$ for each d-node, which will be used as a replacement of $\texttt{emb}(u,t)$ for any events after t that involve the state of (u,t). An apparent benefit is $\boldsymbol{\phi}_{u,t}$ can be used for computing future evolutions before obtaining the embedding $\texttt{emb}(u,t)$.

It is important that the decoupling of the d-nodes should not break the original dependencies in the model. Fortunately, it is easy to observe that involving the d-nodes will not change the obtained node embeddings if its associated two embeddings are equal:

$$\boldsymbol{\phi}_{u,t} = \texttt{emb}(u,t), \quad \forall (u,t) \in \mathcal{V}_{\text{CG}}^D.$$

Therefore, during the training phase, EDGE solves a constrained optimization:

$$\min_{\{\boldsymbol{\phi}_{u,t}\},\Theta} \frac{1}{|\mathcal{D}_{train}|} \sum_{e \in \mathcal{D}_{train}} \mathcal{L}(e\,; \Theta, \{\boldsymbol{\phi}_{u,t}\})$$
$$\text{subject to} \quad \boldsymbol{\phi}_{u,t} = \texttt{emb}(u,t), \quad \forall (u,t) \in \mathcal{V}_{\text{CG}}^D.$$

Various algorithms are available for solving such a constrained optimization problem. For example, one can use Lagrangian method with quadratic penalty, and solve it by alternative primal-dual updates. In our experiments, we enforce the constraints softly, by adding a quadratic penalty $\frac{\alpha}{2}\|\boldsymbol{\phi}_{u,t} - \texttt{emb}(u,t)\|_2^2$ to the loss whenever the node $\texttt{emb}(u,t)$ is reached in the computational DAG.

4 Experiments

We conduct experiments on four public datasets. Two of them are large datasets close to industrial scale, on which we show the scalability and prediction accuracy of EDGE, by comparing to a diverse set of SOTA methods. The other two are comparatively smaller datasets, on which all dynamic graph baselines are runnable, so they are used for comparison to them. Implementation of EDGE will be publicly available upon acceptance.

Datasets include Taobao [31], MovieLens-25M (ML-25M) [32], lastfm-1K [33], and Reddit [34]. Dataset statistics are summarized in Table 1. Taobao and ML-25M are large-scale. Especially, Taobao contains about 100 million interactions. All datasets provide timed user-item interaction data where for each interaction, user ID, item ID, timestamp, item feature, and event feature (if available) are given. For Reddit, subreddits are considered as items. For ML-25M, we ignore the ratings and simply use it as interaction data. For Taobao and ML-25M, we sort the interactions by timestamp, and use a (0.7,0.1,0.2) data-split for training, validation, and testing, respectively. For Lastfm-1K and Reddit, we use the filtered version given by [14] and follow their data-split of (0.8,0.1,0.1). More details and downloadable links to the datasets are provided in Appendix A.

Table 1: Dataset Statistics

Dataset	#users	#items	#interactions
lastfm-1K	1,000	980	1,293,103
Reddit	10,000	984	672,447
ML-25M	162,538	59,048	24,999,849
Taobao	987,975	4,111,798	96,678,667

Baselines. We compare EDGE to a wide range of baselines spanning 3 categories: *(i) Dynamic graph models* include JODIE [14], dynAERNN [18], TGAT [21], CTDNE [35], and DeepCoevolve [23]. The first three were proposed recently and more advanced. *(ii) GNNs*: GCMC [7] is a SOTA graph-based architecture for link prediction in user-item interaction network. GCMC-SAGE is its stochastic variant which is more efficient, using techniques in GraphSAGE [21]. GCMC-GAT is another variant based on graph attention networks [36]. *(iii) Deep sequence models*: SumPooling is a simple yet effective model widely used in industry. GRU4REC [37] is a representative of RNN-based models. SASRec [38] is a 2-layer transformer decoder-like model. DIEN [39] is an advanced attention-based sequence model. MIMN [31] is a memory-enhanced RNN for better capturing long sequential user behaviors. It is a strong baseline and the Taobao dataset were publicized by the authors.

Table 2: Overall performance. EDGE uses 160 and 20 d-nodes per batch on Taobao and ML-25M respectively. * indicates that the released implementation is improved by us to make the method either runnable on these large datasets or better adapted to the evaluation metric, so the results are expected to be better than the original version. Modifications are described in Appendix B. '-oom-' stands for out of memory.

	method	Taobao		ML-25M	
		MRR	Rec@10	MRR	Rec@10
GNNs	GCMC*	0.303	0.542	0.171	0.378
	GCMC-SAGE*	0.149	0.366	0.109	0.264
	GCMC-GAT*	0.230	0.494	0.185	0.404
Sequence Models	SumPooling	0.415	0.664	0.302	0.591
	GRU4REC*	0.546	0.777	0.364	0.657
	DIEN*	0.605	0.834	0.356	0.638
	MIMN*	0.607	0.828	0.363	0.653
	SASRec*	0.488	0.702	0.360	0.653
Dynamic Graphs	dynAERNN	-oom-	-oom-	0.249	0.509
	TGAT	0.016	0.025	0.114	0.219
	JODIE*	0.454	0.680	0.354	0.634
	EDGE	0.675	0.841	0.397	0.673

Configuration. In all experiments, both node embedding dimension and feature embedding dimension are 64. Only Reddit contains event features. Other datasets may provide categorical item features. We experimentally find that including item ID as an additional item feature is generally helpful. To conduct stochastic optimization, we sort the events by time and divide them into 300 batches, so that each batch contains a sub-graph within a continuous time window. To use d-nodes, we choose K d-nodes per batch. The threshold n^* for active nodes is chosen to be 200 for users and 100 for items on both Taobao and ML-25M. The other two datasets are filtered and small, so threshold is not necessary. We use $n^* = \infty$ for both datasets (which means no static embedding is used for any nodes). Besides, we also tried using $n^* = 400$ for Reddit and $n^* = 500$ for lastfm-1K.

Evaluation metric. We measure the performance of different models in terms of the mean reciprocal rank (MRR), which is the average of the reciprocal rank, and recall@10. For both metrics, higher values are better. On Taobao and ML-25M, the ranking of the truly interacted item in each event is calculated among 500 items where the other 499 negative items are uniformly sampled from the set of all items. On Reddit and lastfm-1K, ranks are over all items. In the tables in this section, we mark the best performing method by red color and the second best by blue color.

Result 1: On large datasets.

(i) MRR and Rec@10: The overall performances on Taobao and ML-25M are summarized in Table 2. EDGE has consistent improvements compared to all baselines. Improvements of 11.2% and 9.1% over the second best method are achieved in terms of MRR. In fact, the released implementations of many baselines cannot run or give reasonable performances on these large datasets. As indicated by symbols * in Table 2, we improve the baselines to allow them to have more advantages. Please refer to Appendix B for details.

(ii) Training efficiency: In Fig. 4, the x-axis is the training time per epoch, which is a measure of training efficiency, and the y-axis is the MRR performance on test set. While achieving good performances in MRR, EDGE is very efficient. It has a similar efficiency as the sequence models, and is a lot faster than both GNN-based models and exisiting dynamic

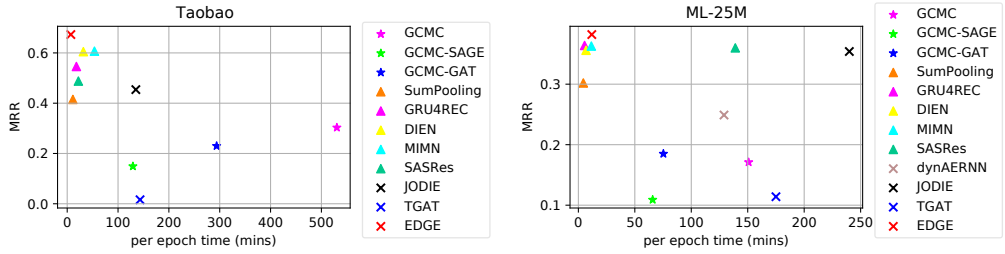


Figure 4: Training time per epoch and MRR performance on test set. This figure shows the comparison in both efficiency and accuracy.

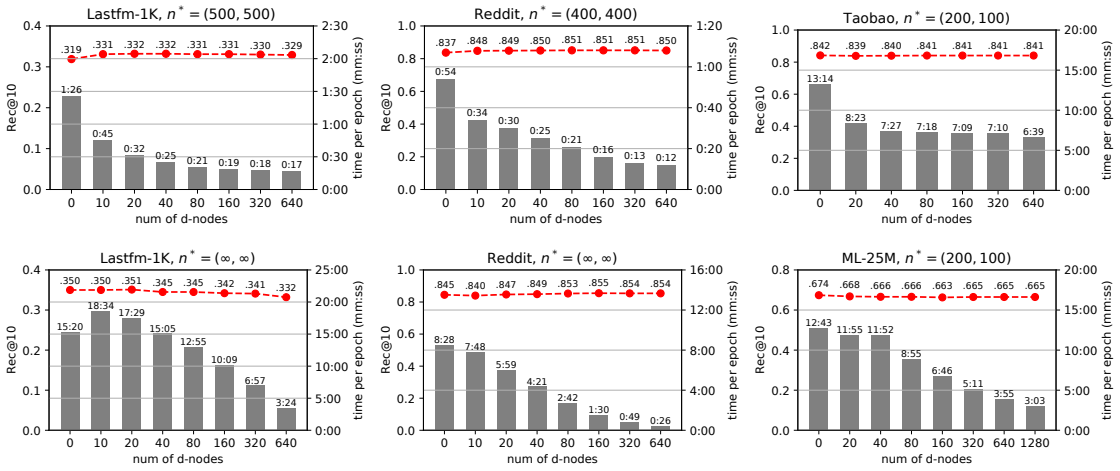


Figure 5: Bar plots indicate the training time (mins:seconds) per epoch. Red scatter points are the Rec@10 performance. Similar figures with MRR performance are given in Appendix C.

graph models, which take 2 hours or more to run on a single epoch. Furthermore, we observe that EDGE actually takes fewer number of epochs to converge compared to the sequence models. The #(d-nodes) and active threshold n^* are the same as that in Table 2. Results with different configurations will be compared later.

Table 3: Comparison to dynamic graph models.

method	lastfm-1K		Reddit	
	MRR	Rec@10	MRR	Rec@10
CTDNE	0.01	0.01	0.165	0.257
DeepCoevolve	0.019	0.039	0.171	0.275
JODIE	0.195	0.307	0.726	0.852
dynAERNN	0.021	0.038	0.157	0.301
TGAT	0.015	0.023	0.602	0.775
EDGE	0.199	0.332	0.725	0.855

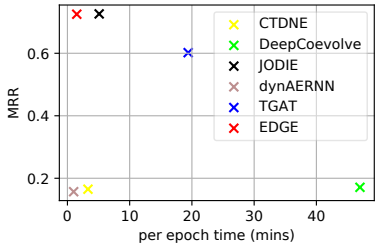


Figure 6: Run-time comparison on Reddit.

Result 2: On small datasets. Lastfm-1K and Reddit are datasets pre-processed by JODIE which are relatively small so that all dynamic graph models can scale to them. *(i) MRR and Rec@10* are summarized in Table 3. Overall, EDGE has similar performances as JODIE, which is expected, because the architecture we use in EDGE is similar to that in JODIE. The advantage of EDGE is in efficient training, but on small datasets, JODIE can be well trained, too. *(ii) Runtime:* However, EDGE is more efficient than JODIE. Fig. 6 shows the training time of EDGE when #(d-nodes) per batch is 160 and the threshold for active nodes is $n^* = \infty$, so all its speed-up comes from d-nodes.

Result 3: Ablation study on the d-nodes. We validate the effectiveness of EDGE, especially the speed-up from d-nodes, by a series of ablation experiments. Fig. 5 summarizes the results. By gradually increasing the number of d-nodes, we can observe the decrease in training time from the bar plots. The d-nodes are very effective in reducing the computational cost in the sense that they constitute a small portion of the computational nodes, especially on large

datasets, and do not have much effect on the prediction performances. We can even observe some improvement in prediction accuracy when using more d-nodes. This might be credited to the easiness of training on shorter sequences.

Table 4: Longest path comparison on ML-25M.

Algorithm 2 (ours)	number of d-nodes	0	10	20	40	80	160	320	640
	longest path length	209.75	164.05	142.81	115.01	89.63	66.12	45.47	29.43
cut-by-time	number of parts	1	2	4	8	16	32	64	128
	number of d-nodes	0	134	250	346	429	506	588	701
	longest path length	209.75	202.6	198.95	196.29	193.02	190.84	187.39	179.51

Result 4: Reduction in the path length. To verify that our selection algorithm has actually reduce the longest path in the computational DAG, we explicitly compute its length after decoupling the selected d-nodes. For comparison, we also implement a comparatively simpler approach, called 'cut-by-time'. It selects the d-nodes to divide the DAG into multiple disconnected parts based on the time intervals. Results are reported in Table 4, which reveals a few advantages of our heuristic. First, we can easily control the number of d-nodes. However, in the case of cut-by-time or other algorithms that cut the DAG into separate parts, the number of d-nodes is determined by the number of parts. Second, our heuristic is a lot more effective, in terms of reducing the length of the longest path.

Result 5: Ablation study on the active nodes. What if active nodes do not use static embeddings? The main purpose of using static embeddings is to improve the efficiency and ease of training, which is especially important for large datasets. To demonstrate this, we gradually increase the node splitting threshold n^* to see the change in run-time and accuracy. Note that when n^* increases, fewer nodes are categorized as active nodes to use static embeddings. For the extreme case when $n^* = \infty$, all nodes are treated as inactive and do not use static embedding. It is observed in Table 5 that using more active nodes can effectively improve training efficiency. Especially, on Movielesns-25M, the efficiency will be unacceptable without using static embeddings for active nodes. In terms of accuracy, on Taobao, the case of (n_1^*, n_2^*)=(200,100) has achieved as good performance as other thresholds. On movielens-25M, it seems (n_1^*, n_2^*)=(400,200) is a better choice for accuracy and efficiency trade-off.

Table 5: Results of using fewer nodes as active nodes.

Taobao

(n_1^*, n_2^*)	time per epoch	MRR	Rec@10
(200,100)	13.14 mins	0.675	0.642
(400,200)	13.84 mins	0.674	0.841
(800,400)	14.52 mins	0.674	0.841
(1600,800)	15.57 mins	0.675	0.841
(∞, ∞)	27.50 mins	0.674	0.841

ML-25M

(n_1^*, n_2^*)	time per epoch	MRR	Rec@10
(200,100)	12.72 mins	0.397	0.674
(400,200)	22.79 mins	0.404	0.683
(800,400)	1 hr 4 mins	0.409	0.687
(1600,800)	3 hr 1 mins	0.406	0.688
(∞, ∞)	20 hr 54 mins	-	-

5 Conclusion and discussion

In this paper, we have proposed an efficient framework, EDGE, to address the computational challenges of learning large-scale dynamic graphs. We evaluated EDGE on large-scale item ranking tasks, and showed that its performances outperform several classes of SOTA methods. The models discussed in this paper are mainly based on node-wise updates by an recurrent operator when new interactions appear, which lack a GNN-like aggregation from a node's neighbors. Future works include adaptation of this efficient algorithm to more sophisticated aggregations, and also exploration on effective training algorithm for constrained optimization.

References

[1] Donald J Jacobs, Andrew J Rader, Leslie A Kuhn, and Michael F Thorpe. Protein flexibility predictions using graph theory. *Proteins: Structure, Function, and Bioinformatics*, 44(2):150–165, 2001.

[2] Evelien Otte and Ronald Rousseau. Social network analysis: a powerful strategy, also for the information sciences. *Journal of information Science*, 28(6):441–453, 2002.

[3] Nikos Manouselis, Hendrik Drachsler, Riina Vuorikari, Hans Hummel, and Rob Koper. Recommender systems in technology enhanced learning. In *Recommender systems handbook*, pages 387–415. Springer, 2011.

[4] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 2702–2711, 2016.

[5] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[6] Federico Monti, Michael Bronstein, and Xavier Bresson. Geometric matrix completion with recurrent multi-graph neural networks. In *Advances in Neural Information Processing Systems*, pages 3700–3710, 2017.

[7] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*, 2017.

[8] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983. ACM, 2018.

[9] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. Revisiting graph neural networks for link prediction. *arXiv preprint arXiv:2010.16103*, 2020.

[10] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[11] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, 2018.

[12] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. In *International Conference on Machine Learning*, pages 942–950. PMLR, 2018.

[13] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *Advances in Neural Information Processing Systems*, 31:4558–4567, 2018.

[14] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1269–1278, 2019.

[15] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187, 2005.

[16] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[17] Mahmudur Rahman, Tanay Kumar Saha, Mohammad Al Hasan, Kevin S Xu, and Chandan K Reddy. Dylink2vec: Effective feature representation for link prediction in dynamic networks. *arXiv preprint arXiv:1804.05755*, 2018.

[18] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowledge-Based Systems*, 187:104816, 2020.

[19] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 519–527, 2020.

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

[21] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962*, 2020.

[22] Nahla Mohamed Ahmed Ibrahim and Ling Chen. Link prediction in dynamic social networks by integrating different types of information. *Applied Intelligence*, 42(4):738–750, 2015.

[23] Hanjun Dai, Yichen Wang, Rakshit Trivedi, and Le Song. Deep coevolutionary network: Embedding user and item features for recommendation. *arXiv preprint arXiv:1609.03675*, 2016.

[24] Rakshit Trivedi, Hanjun Dai, Yichen Wang, and Le Song. Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *International Conference on Machine Learning*, pages 3462–3471. PMLR, 2017.

[25] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. Streaming graph neural networks. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 719–728, 2020.

[26] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[27] Yossi Shiloach and Uzi Vishkin. An o (n2log n) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[28] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[29] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[30] Incorporate Gurobi Optimization. Gurobi optimizer reference manual, 2018.

[31] Qi Pi, Weijie Bian, Guorui Zhou, Xiaoqiang Zhu, and Kun Gai. Practice on long sequential user behavior modeling for click-through rate prediction. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2671–2679, 2019.

[32] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

[33] O. Celma. *Music Recommendation and Discovery in the Long Tail*. Springer, 2010.

[34] Jason Baumgartner, Savvas Zannettou, Brian Keegan, Megan Squire, and Jeremy Blackburn. The pushshift reddit dataset. In *Proceedings of the international AAAI conference on web and social media*, volume 14, pages 830–839, 2020.

[35] Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. Continuous-time dynamic network embeddings. In *Companion Proceedings of the The Web Conference 2018*, pages 969–976, 2018.

[36] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[37] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based recommendations with recurrent neural networks. *arXiv preprint arXiv:1511.06939*, 2015.

[38] Wang-Cheng Kang and Julian McAuley. Self-attentive sequential recommendation. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 197–206. IEEE, 2018.

[39] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 5941–5948, 2019.

A Dataset details

A.1 Overall Description

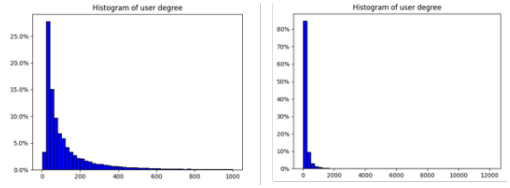
Taobao dataset contains user-item interaction data from November 25 to December 03, 2017 (one week). The details of this dataset can be found at `https://tianchi.aliyun.com/dataset/dataDetail?dataId=649`.

Movielens-25M dataset contains about 25 million rating activities from MovieLens, from January 09, 1995 to November 21, 2019. The details of this dataset can be found at `https://grouplens.org/datasets/movielens/25m/`.

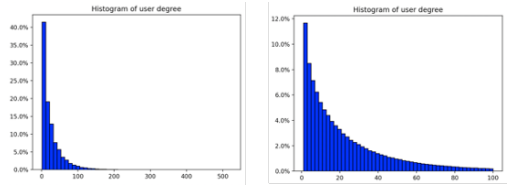
Lastfm-1K dataset contains music listening data of 1000 users. We use the dataset prepared by [14]. The details can be found at `https://github.com/srijankr/jodie` and `http://ocelma.net/MusicRecommendationDataset/lastfm-1K.html`.

Reddit dataset contains one month of posts made by users on subreddits [34]. We use the dataset filtered by [14], which contains the 1,000 most active subreddits as items and the 10,000 most active users. The details can be found at `https://github.com/srijankr/jodie`.

A.2 Node Degree Distribution



(a) ML-25M



(b) Taobao

Figure 7: Distribution of the number of observed interactions in the training dataset.

B Baseline specifications

Some of the compared baseline methods are not directly scalable to large scale interaction graphs, or originally designed for ranking task. To make the baselines runnable on large and sparse graphs and comparable to our proposed method, we have made a few improvements to the released implementations.

- **GRU4Rec & DIEN & MIMN**: We changed the data batching from per-user based to per-example based, to better adapt for time ordered interaction data and to better make the FULL use of the training data. Besides, the implementation of GRU4Rec follows implementation by the authors of MIMN paper, which includes some modifications compared to the original version of GRU4Rec released by their authors, and should be expected to perform better.
- **GCMC & GCMC-SAGE & GCMC-GAT**: We changed the loss function from Softmax over different ratings to Softmax over [true item, 10 random selected items], to better adapt to the ranking task.
- **dynAERNN**: These two methods are originally designed for discrete graph snapshots, while our focused tasks are continues interaction graphs. We manually transformed the interactions graph into 10 snapshots, with equal edge count increments.

  For the downstream ranking task, we followed the evaluation method used in dySat [19]: after the node embeddings are trained, we train a logistic regression classifier to predict link between a pair of user / item node embedding using Hadmard operator. The logits on test set are used for ranking. For Taobao dataset, we are not able to get results of dynAERNN given the memory constraint.

- **JODIE**: we made the following adaptions: 1) replaced the static one-hot representation to 64-dimensional learnable embeddings because the number of nodes are too large that the original implementation will have the out-of-memory issue; 2) add a module to deal with categorical feature via a learnable embedding table since the original implementation is more suitable for continuous feature; 3) used triplet loss with random negative example rather than original MSE loss, which empirically show improvements.
- **TGAT**: We notice that the performance of TGAT is particularly bad on Taobao. We spend efforts on checking the experiments to make sure there is no bug. Finally, we identify two issues of using this method on Taobao dataset. Firstly, TGAT is an inductive model, which could potentially give a better generalization ability for *newly observed* nodes in the test set, but on the other hand it is not expressive enough to make use of the large-scale dataset. Secondly, the original implementation in TGAT does not deal with categorical feature, but in the Taobao dataset, the only feature is a categorical feature for the items. We need to add a module to map the categorical feature to a learnable feature embedding table (similar to how we modify the feature module in JODIE) to achieve a better performance. With this feature embedding module, we were able to increase its performance on Taobao to about 0.086 (MRR) and 0.164 (Rec@10).

C Ablation study results

We present the complete results for the ablation study blow.

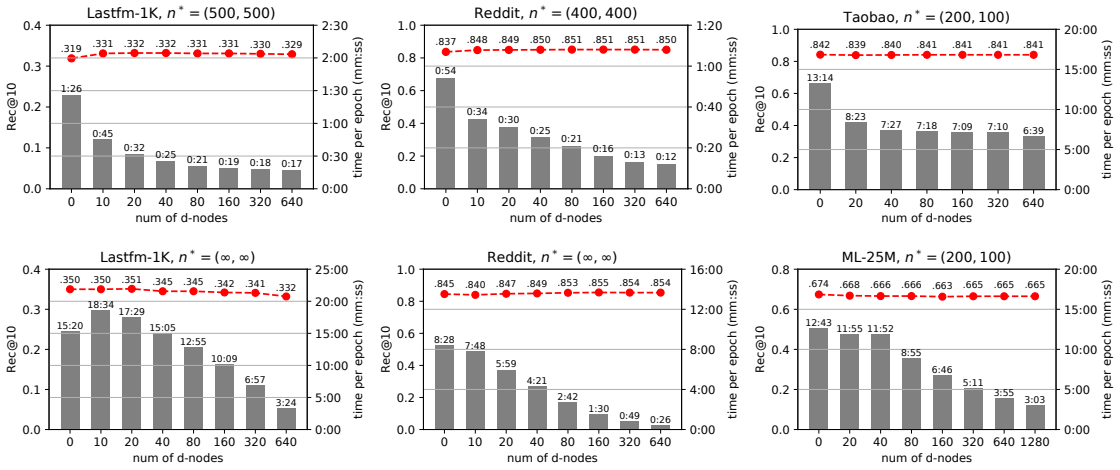


Figure 8: Bar plots indicate the training time (mins:seconds) per epoch. Red scatter points are the **Rec@10** performance.

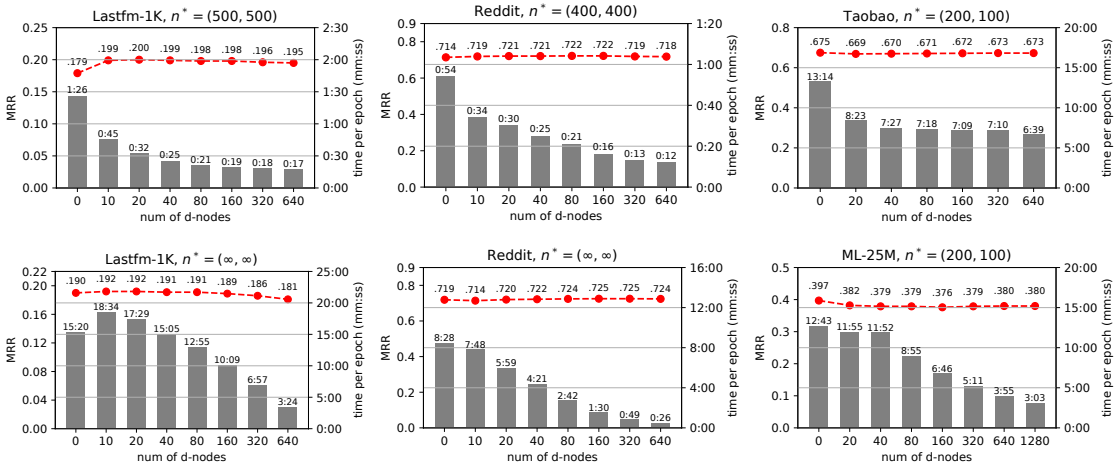


Figure 9: Bar plots indicate the training time (mins:seconds) per epoch. Red scatter points are the **MRR** performance.

D Computing resources

Experiments on large-scale datasets are run on NVIDIA Quadro RTX 6000 (24 GB memory) or on NVIDIA Tesla v100 (16 GB memory). Experiments on small datasets are either run on NVIDIA GeForce 2080 Ti (11 GB memory) or on NVIDIA Tesla v100 (16 GB memory). Each individual job is run on a single GPU.