xGCN: An Extreme Graph Convolutional Network for Large-scale Social Link Prediction

Xiran Song
Huazhong University of Science and Technology
Wuhan, China
xiransong@hust.edu.cn

Jianxun Lian
Microsoft Research Asia
Beijing, China
jianxun.lian@outlook.com

Hong Huang*
Zihan Luo
Wei Zhou
Xue Lin
honghuang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Mingqi Wu
Microsoft Gaming
Redmond, United States

Chaozhuo Li
Xing Xie
Microsoft Research Asia
Beijing, China

Hai Jin
Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

Graph neural networks (GNNs) have seen widespread usage across multiple real-world applications, yet in transductive learning, they still face challenges in accuracy, efficiency, and scalability, due to the extensive number of trainable parameters in the embedding table and the paradigm of stacking neighborhood aggregations. This paper presents a novel model called xGCN for large-scale network embedding, which is a practical solution for link predictions. xGCN addresses these issues by encoding graph-structure data in an extreme convolutional manner, and has the potential to push the performance of network embedding-based link predictions to a new record. Specifically, instead of assigning each node with a directly learnable embedding vector, xGCN regards node embeddings as static features. It uses a propagation operation to smooth node embeddings and relies on a *Refinement neural Network* (RefNet) to transform the coarse embeddings derived from the unsupervised propagation into new ones that optimize a training objective. The output of RefNet, which are well-refined embeddings, will replace the original node embeddings. This process is repeated iteratively until the model converges to a satisfying status. Experiments on three social network datasets with link prediction tasks show that xGCN not only achieves the best accuracy compared with a series of competitive baselines but also is highly efficient and scalable.

*Hong Huang is the corresponding author. Xiran Song, Hong Huang, Zihan Luo, Wei Zhou, Xue Lin and Hai Jin are affiliated with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology.

CCS CONCEPTS

• **Information systems → Recommender systems**; • **Computing methodologies → Learning latent representations**.

KEYWORDS

Link prediction, graph neural network, social recommendation

ACM Reference Format:
Xiran Song, Jianxun Lian, Hong Huang, Zihan Luo, Wei Zhou, Xue Lin, Mingqi Wu, Chaozhuo Li, Xing Xie, and Hai Jin. 2023. xGCN: An Extreme Graph Convolutional Network for Large-scale Social Link Prediction. In *Proceedings of the ACM Web Conference 2023 (WWW '23), April 30–May 04, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543507.3583340

1 INTRODUCTION

Graph structure data, such as social networks, knowledge graphs, and molecular graphs, is prevalent in modern life. Graph embedding [8] has been shown to be an effective technique for representing graph structure data by encoding each node with a low-dimensional vector. In recent years, research interests have shifted from shallow graph embeddings [9, 21, 23] towards *graph neural networks* (GNNs) [29] due to their superior ability to explicitly encode useful patterns from the high-order neighborhood [26, 27]. In this paper, we examine the case of embedding social networks, where a user's neighborhood on the graph plays a crucial role in representing the user, and its application in social link prediction.

In the inductive graph representation learning tasks [10], the nodes are associated with attributes, and all trainable parameters come from the graph neural networks: $\Theta = \{\Theta_W\}$. However, in classical network embedding tasks, each node is associated with a d-dimensional embedding vector which is trainable, so the parameter set becomes $\Theta = \{\Theta_W, \Theta_E\}$. Θ_E is called the embedding table and denoted by $\mathbf{E}$ hereinafter. Mainstream methods of GNNs usually follow a general paradigm: aggregating messages from neighbors, performing some transformation, stacking these two steps multiple times to acquire high-order neighborhood information, and learning all the parameters by *stochastic gradient descent* (SGD). Potential

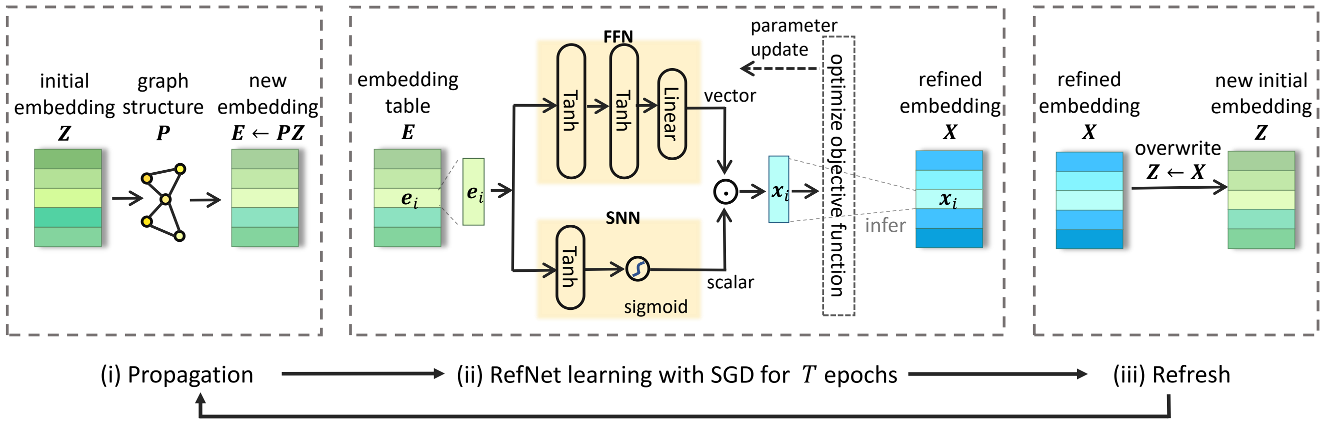


Figure 1: An overview of the key components in xGCN

drawbacks of this paradigm are three-fold: (1) The neighborhood size increases exponentially with the hop distance, which can easily cause the over-smoothness problem and scalability issue; (2) For a graph with N nodes, the embedding table $\mathbf{E}$ alone has $O(Nd)$ learnable parameters, which makes GNNs hard to parallelize (because the communication cost will dominate the computational cost); (3) Parameters in Θ_W and in Θ_E have different properties (e.g., Θ_W is dense while Θ_E is sparse), however, both of them are updated by gradient back-propagation in a unified framework. It together with the existence of gradient vanishing and gradient explosion issues, may lead to sub-optimal performance in both the training efficiency and final accuracy of GNNs. Thus, in some prior studies, researchers find that removing Θ_W and retaining only Θ_E can yield better performance [12] for link predictions.

In this paper, we propose a brand-new GNN named xGCN, which is short for *extreme graph convolutional network*, for social link predictions. Our motivations come from a series of prior studies: (1) RandNE [34] demonstrates that the network structure information can be preserved with iteratively embedding propagation without any trainable parameters; (2) LightGCN [12] indicates that in the embedding propagation framework, when the node embedding is trainable with some supervised labels such as link predictions, the quality of node embeddings can further be improved; (3) [6, 15] demonstrate that the *Feed Forward Network* (FFN) plays a key role in memorizing knowledge and performing essential information transformation in the Transformer architecture. Thus, we abandon the classical paradigm of GNN, which is denoted as *[neighbors aggregation, transformation, stacking, SGD(Θ_W, Θ_E)]* for simplicity. Instead, we propose a new paradigm of iterative *[Propagation, Refinement, SGD(Θ_W), Refresh]*, which integrates the motivations of message propagation, controllable embeddings, and message distillation. An overview of this process is illustrated in Figure 1.

Similar to RandNE, the node embeddings in xGCN are not trainable, thus, we can get rid of the $O(Nd)$ embedding table and the model is feasible for parallelization. We first perform a step of embedding propagation to encode the network structure information into node embeddings (Figure 1-(i)). We argue that as long as a node embedding carries a certain amount of graph structure information, an FFN module can perform information transformation so that node embeddings are refined to a better status (Figure 1-(ii)). Trainable parameters are located only in the FFN module, and they are updated by SGD. After the FFN is optimized, we refresh the embedding table with the output of FFN (Figure 1-(iii)). In this way, the embedding table gets updated in one shot rather than in a slow, iterative manner with SGD (such as the mechanism in LightGCN [12]).

We conduct link prediction experiments on three real-world social network datasets. xGCN consistently outperforms a set of competitive baselines such as GAMLP and PPRGo. This demonstrates that the new GNN framework can learn high-quality embeddings for various social networks. Besides the accuracy advantage, we also conduct training efficiency studies and verify that xGCN converges much faster than classical GNN models. At last, to test the scalability, we train xGCN on a 100 million Xbox graph with a single machine, using only 92 GB RAM and 11 hours to converge, and it can outperform node2vec by a large margin. To summarize,

- We propose a novel model xGCN for social link prediction, which gets rid of the traditional GNN paradigm and achieves better accuracy, efficiency, and scalability with less trainable parameters.
- We design three core components, including propagation, the refinement network, and a refresh control mechanism, to make xGCN effective and robust across different social networks.
- We conduct experiments on three datasets to demonstrate the superiority of xGCN on effectiveness, efficiency, and scalability. Our code is released at https://github.com/CGCL-codes/xGCN.

2 METHODOLOGIES

2.1 Task Definition

Graph embedding for link predictions. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ containing $|\mathcal{V}| = N$ nodes and $|\mathcal{E}| = M$ edges. The edges of $\mathcal{G}$ can also be formulated as an adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, with $\mathbf{A}_{uv} = 1$ indicating an edge from node u to node v. A diagonal matrix $\mathbf{D}$ stores the degree of each node: $\mathbf{D}_{uu} = \sum_{k=1}^{N} \mathbf{A}_{uk}$. Our goal is to learn an embedding model f which can represent each node $v \in \mathcal{V}$ with a d-dimensional embedding vector $\mathbf{x}_v = f(v|\mathcal{G})$, $\mathbf{x}_v \in \mathbb{R}^d$, so that the occurrence probability of an edge between two nodes u and v can be measured by their dot-product $\hat{y}_{uv} = \mathbf{x}_u^T \mathbf{x}_v$.

2.2 The Framework of xGCN

Mainstream graph embedding models usually allocate a learnable embedding table $\mathbf{E} \in \mathbb{R}^{N\times d}$. It together with some additional graph neural network parameters Θ_W, constitutes the trainable parameter set of f, i.e., $\Theta = \{\Theta_W, \Theta_E\}$. However, when the size of the graph is large, which is the common case for real-world social networks, the trainable embedding table $\mathbf{E}$ becomes the bottleneck causing training efficiency and scalability problems. In xGCN, we propose a totally different approach, in which there are three key operations, including *embedding propagation*, *embedding refinement*, and *embedding refresh*. These three operations are executed in a chain and will be repeated for multiple iterations until convergence, with the fundamental goal of learning graph-structure-aware node embeddings. In contrast to existing GNNs, the embedding table in xGCN is not the trainable parameter, and all trainable parameters lie in a refinement neural network, i.e., $\Theta = \{\Theta_W\}$. To distinguish from a trainable embedding table, we use $\mathbf{Z}$ to denote the base embedding table of xGCN. We initialize $\mathbf{Z}$ randomly and then perform a graph convolutional operation to smooth nearby nodes' embedding as well as propagate node information along the graph structure. Next, we train a Refinement neural Network (denoted as *RefNet*) to transform the current embeddings into new embeddings $\mathbf{X}$, with the goal to preserve useful signals and filter out noises. The parameters of RefNet will be updated by normal gradient descent methods such as SGD. Third, when the RefNet is well trained, which means that it can output higher quality embeddings, we replace $\mathbf{Z}$ with $\mathbf{X}$, which we refer to as the *embedding refresh* operation. These three operations are repeated with multiple iterations until the model converges to a satisfying status. The overview of xGCN is illustrated in Figure 1 and Algorithm 1. Details for each key operation are as follows.

2.3 Key Components

2.3.1 ***Embedding propagation.*** We assume that the unique information for each node is stored in the corresponding embedding vector in $\mathbf{Z}$. At the very beginning, $\mathbf{Z}$ is initialized as a random matrix. Since the local neighborhood is important to depict a node, we aggregate neighbors to derive the node's new representation, so that the network structure information is strengthened:

$$\mathbf{E} \leftarrow \mathbf{PZ} \tag{1}$$

where $\mathbf{P}$ is the propagation matrix of the graph, it stands for the direction of information to be smoothed and can have different implementations, such as the normalized adjacency matrix $\tilde{\mathbf{A}}$ of $\mathcal{G}$ (i.e., information is propagated to the first order neighborhood): $\tilde{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$, or the top-$k$ PPR neighbors matrix $\tilde{\mathbf{\Pi}}$ (i.e., propagate to the most influential neighbors for the center node), or multiplication of normalized adjacency matrix: $\tilde{\mathbf{A}}^2$ (i.e., propagate to the second order neighborhood). We empirically find that using $\tilde{\mathbf{A}}$ can achieve the best performance.

2.3.2 ***RefNet learning.*** Embedding propagation is an unsupervised operation. Although it can encode graph structure information, unfortunately, it also brings a lot of noise. To extract useful information and filter out noise, we design a RefNet component to learn to transform relatively lower-quality embeddings into ones that better encode the graph structure. RefNet is composed of a *Feed-Forward Network* (FFN) and a *Scaling Neural Network* (SNN). The last hidden layer of FFN does not include an activation function, while the rest of the hidden layers use Tanh as an activation function. A design principle for FFN is that the middle layers need a significantly larger dimension than the input vector. E.g., a 2-layer FFN with an input embedding size being 64 is $[Linear(64, 1024), Tanh, Linear(1024, 64)]$, where *Linear* indicates multiplying a parameter matrix and then adding a bias vector. The SNN is a smaller neural network that outputs a single scalar between $(0, 1)$, which shapes the magnitude of FFN's output vectors to a proper level. We empirically find that the normalization of the last layer of FFN is important and SNN performs much better than others such as Tanh and L2-normalization. The SNN's structure is $[Linear(64, 32), Tanh, Linear(32, 1), Sigmoid]$. The output of RefNet is:

$$\mathbf{X} = SNN(\mathbf{E}) \cdot FFN(\mathbf{E}) \tag{2}$$

We adopt the pair-wise ranking loss function – BPR [12] – to optimize the parameters in the RefNet:

$$\mathcal{L} = \frac{1}{|\mathcal{E}|} \sum_{\langle u,v\rangle \in \mathcal{E}, \langle u,\hat{v}\rangle \notin \mathcal{E}} softplus(y_{u,\hat{v}} - y_{u,v}) \tag{3}$$

where $y_{u,v}$ is the scorer to estimate edge probability according to the node embeddings. Without loss of generality, in this paper, we use dot product, $y_{u,v} = \mathbf{x}_u^T \mathbf{x}_v$, as the scorer, but it can be easily extended to other types of scorers such as Logistic Regression or Deep Neural Networks. For each positive edge $(u, v) \in \mathcal{E}$, we randomly sample a pair of nodes $(u, \hat{v})$ which does not exist in $\mathcal{E}$ as a negative instance.

2.3.3 ***Embedding refresh.*** Note that the embedding table $\mathbf{E}$ is not trainable during the learning process of RefNet. After the RefNet is well trained, the resulting embeddings can represent a better state of the node representations, so we replace the embedding table with RefNet's output:

$$\mathbf{Z} \leftarrow \mathbf{X} \tag{4}$$

2.3.4 ***Training strategy.*** During the training of xGCN, the three operations – embedding propagation, RefNet learning, and embedding refresh – are repeatedly executed. One challenge is how to coordinate between RefNet learning and embedding refresh. Since these two components are not optimized with derivable parameters under an end-to-end framework, if the refresh operation is performed at an improper time, RefNet's optimization may be severely impacted (see experiments in Section 3.5, setting K=0 and K=Inf, and Section 3.6). To address this challenge, we design a simple yet effective *refresh controlling mechanism*: during a warm-up stage, the representation refresh operation and propagation operation are performed after the RefNet is updated for T epochs; there are in total K times refresh/propagation operations in the warm-up stage, where both T and K are hyper-parameters; after the warm-up stage, embeddings will not be refreshed at a regular frequency; instead, we use the validation set to detect the best epoch of embeddings up to now. If the RefNet can no longer improve the metrics on the validation set for T_{tol} epochs, which means the refinement for the current input embedding table is converged, we refresh the embedding table with the current best state of the embedding table, and then perform propagation, with the hope that RefNet can continue to refine the embeddings at a new start. The detailed training algorithm is shown in Algorithm 1.

Algorithm 1 xGCN Training Process

```
Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, Hyper-parameters $K$, $T$, $T_{tol}$
Output: Node embeddings $\mathbf{X}$
 1: Randomly initialize E and RefNet.
 2: Do Propagation according to Eq.(1)
 3: $total\_refresh\_times \leftarrow 0$
 4: for $epoch = 1$ to $max\_epoch$ do
 5:     update RefNet with SGD to minimize Eq.(3)
 6:     if $total\_refresh\_times < K$ then
 7:         /* In the warm-up stage */
 8:         if  epoch % T == 0 then
 9:             Do Refresh according to Eq.(4)
10:             Do Propagation according to Eq.(1)
11:             $total\_refresh\_times$ += 1
12:         end if
13:     else
14:         if  no improvement for $T_{tol}$ epochs then
15:             Do Refresh according to Eq.(4)
16:             Do Propagation according to Eq.(1)
17:         end if
18:     end if
19: end for
```

2.4 Discussions

xGCN is a new style of GNN, which includes two decoupled steps: step-1 is to conduct message propagation, so the graph-structure signals can be distributed to node embeddings. There are two purposes designed for this step: providing graph signals and reducing repeated subgraph computation cost. Step-2 is designed for information refinement (RefNet) since both useful and useless signals will be passed in step-1. Prior works such as [1] reveal that the bottleneck of message-passing style GNNs lies in the over-squashing issues, especially for long-range dependencies, which motivates us to leverage a more powerful refinement network (RefNet) to distill useful information from the squashed embedding.

Classical models allocate a learnable embedding vector for each node, which will result in massive learnable parameters (for example, 100 million nodes with 32-dimensional embedding vectors). Too many learnable parameters usually make models hard to optimize, prone to overfitting, and poor in generalization. xGCN puts all the learnable parameters in RefNet, which contains fewer parameters than the embedding table; meanwhile, RefNet is shared by all the nodes, and its parameters do not belong to sparse parameters, so the generalization ability is better.

2.5 Theoretical Analysis

The framework and optimization of xGCN can be formulated and explained by the EM algorithm [20]. For notation simplicity, let $x_{(k)}$ denote the k-th data sample (which represents a pair of nodes) and $z_{(k)}$ denote the latent variables associated with $x_{(k)}$. Let $\theta(t)$ denote the learned parameters at time step t. The EM algorithm takes the following form:

Table 1: Basic statistics of datasets

	# nodes	# edges	ave. degree	density
Pokec	1,632,803	27,560,308	16.9	$2.06e{-}5$
LiveJournal	4,847,571	62,094,395	12.8	$5.28e{-}6$
Xbox-3m	3,000,000	80,194,576	26.7	$1.78e{-}5$

E-step: Given the estimated parameter $\theta(t)$ at iteration t, compute the expectation of latent variables:

$$Q(z_{(k)}) = p(z_{(k)}|x_{(k)}; \theta(t)) \tag{5}$$

M-step: Update θ to maximize the expected likelihood of the observed data (which is also called the Q-function):

$$Q(\theta|\theta(t)) = \arg\max_{\theta} \mathbb{E}_{z_{(k)} \sim Q(z_{(k)})} [log\ p(x_{(k)}, z_{(k)}|\theta)] \tag{6}$$

The M-step is represented by Line 5 in Algorithm 1 which optimizes model parameters to maximize a value function, while the E-step is represented by Lines 6 to 18 with an implementation trick - the probability mass function $Q(z_{(k)})$ follows a sharp distribution, allowing us to find the maximum value instead of calculating expectations. Detailed theoretical analysis can be found in Appendix A.1. As a result, both the feasibility of the model and convergence of the optimizer can be guaranteed by leveraging the theory of the EM algorithm.

3 EXPERIMENTS

3.1 Experiment Setup

3.1.1 ***Datasets.*** We use two biggest social network datasets from SNAP[1], **Pokec** and **LiveJournal**, and one industrial dataset, **Xbox**, which is a gaming social network provided by Xbox Gaming Corp. The complete Xbox dataset contains about 100 million nodes, for a comprehensive comparison with baselines, we sample a medium-size subgraph including 3 million nodes and denote it as Xbox-3m. Basic statistics are listed in Table 1. We evaluate model performance on the node recommendation task, with details on task settings introduced in Appendix A.2.

3.1.2 ***Evaluation metrics.*** Two widely used evaluation metrics are adopted - Recall and *Normalized Discounted Cumulative Gain* (NDCG) [12, 33]. Recall@K measures the ability to retrieve positive target nodes among top K recommendations, which is defined as the number of retrieved positive nodes divided by the number of total ground-truth positive nodes. We average all users' individual Recall@K as the overall Recall@K indicator. NDCG measures how well positive target nodes can be ranked to the top positions compared with an ideal ranking order. In the experiments, for conciseness, we report Recall@50, Recall@100, and NDCG@100, which are abbreviated as R@50, R@100, and N@100, respectively.

3.1.3 ***Baselines.*** We compare xGCN against a variety of methods, including pure propagation method RandNE [34]; shallow graph embedding methods: node2vec [9], SimpleX [17], and UltraGCN [18] (though named after "GCN", it does not perform graph convolution operation explicitly); competitive GNNs: GraphSAGE [10], GAT

[1]http://snap.stanford.edu/data/index.html

[25] and GIN [30], LightGCN [12], SGC [28], S^{2}GC [35], SIGN [7], GBP [3], GAMLP [32], and PPRGo [2]. Detailed hyper-parameters can be found in Appendix A.3.

3.2 Overall Performance

The overall accuracy performance comparison is summarized in Table 2. For all the datasets, we repeat each model 5 times and report the average scores with standard deviations. We have the following observations: (1) On all three datasets, xGCN outperforms baseline models by a large margin. E.g., when compared with the best baseline in terms of Recall@100, xGCN achieves a performance gain of 15.34% on Pokec, 16.20% on LiveJournal, and 48.18% on Xbox-3m. (2) GNN methods, such as GraphSAGE, GAT, and GIN, are generally better than shallow graph embedding methods, which is in line with the intuition. GNNs explicitly encode meaningful information from the neighborhood to strengthen a node's representation, in theory, they have stronger expressiveness than shallow embedding methods. (3) In most cases, simplified GNNs (such as LightGCN and PPRGo) outperform non-simplified ones (GraphSAGE, GAT, and GIN). On Pokec and LiveJournal, the performance of non-simplified GNNs is significantly much worse than PPRGo, even when their base embedding tables are warmed-up with a well-trained node2vec model, while PPRGo is trained from scratch. These observations are consistent with some related works [12, 28]. In the scenario of node recommendations, embedding transformation, and nonlinear activation may bring difficulty for model training, and thus, degrade models' performance. Different from GraphSAGE, GAT, and GIN, xGCN moves embedding transformation and nonlinear activation from graph convolutions to a refinement network, leaving the message propagation process parameter-free.

To investigate whether xGCN exhibits unfairness, such as its high overall accuracy being achieved by diminishing the performance of some nodes to greatly enhance the performance of others, we plot the accuracy by group in Figure 2. We compare node2vec and PPRGo since node2vec is the most classical node embedding method and PPRGo is the best baseline method on Pokec and LiveJournal. We sort nodes by their degree in ascending order and then split nodes into 10 percentiles. E.g., 0 on the x-axis in Figure 2 means the nodes whose degree belongs to the $0 - 10\%$ percentile. We can observe that xGCN consistently outperforms baselines in all the node groups, which demonstrates that the superiority of xGCN comes from its true power rather than playing some distribution tricks. Another interesting observation is that nodes with lower degrees have higher accuracy values. This phenomenon indicates that users' earlier social relationships are easier to predict.

3.3 Training Efficiency

Next, we examine the training efficiency of xGCN, compared with LightGCN and PPRGo. All models are run with 100 epochs using the same hardware configuration: GPU is *Tesla P100, 16GB* and CPU is *Intel Xeon CPU E5-2690 v4 @ 2.60GHz*. For each epoch, we print the Recall@100 score on the validation set. Figure 3 depicts the training curves of the three models. On all three datasets, xGCN uses far less time to converge to a satisfying status. Since xGCN is much more lightweight than LightGCN and PPRGo, training one epoch of xGCN takes less time than the other two GNNs. For

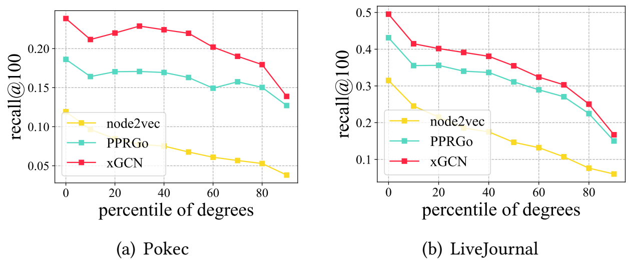


Figure 2: Group-level performance of xGCN, PPRGo, and node2vec. Nodes are evenly split into 10 groups according to the degree in ascending order.

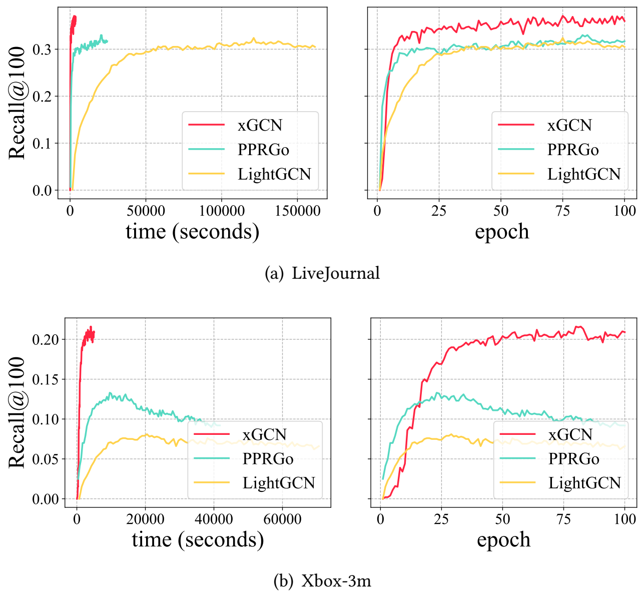


Figure 3: Training efficiency study. The curves are Recall@100 scores on the validation set.

example, on the Xbox-3m dataset, training one epoch costs around 50 seconds for xGCN, 2000 seconds for LightGCN, and 1800 seconds for PPRGo. Besides, on Pokec and LiveJournal, xGCN converges with fewer epochs, thus, the total training time of xGCN is much less than PPRGo and LightGCN. On the Xbox-3m dataset, although xGCN takes a few more epochs to converge, the absolute time cost of xGCN is still much less than the other two models.

3.4 Large-scale Graph with 100 Million Nodes

We test the scalability of xGCN with a real-world Xbox social network that contains 100 million nodes. Our goal is to explore how we can learn large-scale graph embeddings easily with a single normal machine, so in this section, all the models in comparison are run on a CPU device with large RAM. Due to the time limit, we allow all the models to run for at most 72 hours. If a model does not converge after 72 hours, we stop it and use its best model snapshot to perform an evaluation. We tune a few key parameters such as

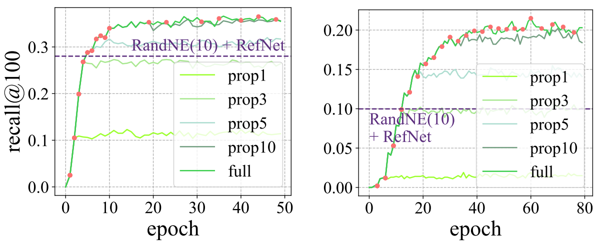
Table 2: Overall performance comparison of different models on three datasets. Numbers are in percentage (%).

	Pokec			LiveJournal			Xbox-3m		
	R@50	R@100	N@100	R@50	R@100	N@100	R@50	R@100	N@100
RandNE [34]	1.04$_{\pm 0.05}$	1.69$_{\pm 0.08}$	0.44$_{\pm 0.02}$	0.30$_{\pm 0.02}$	0.37$_{\pm 0.02}$	0.14$_{\pm 0.01}$	0.60$_{\pm 0.01}$	0.76$_{\pm 0.02}$	0.26$_{\pm 0.01}$
node2vec [9]	4.51$_{\pm 0.06}$	7.45$_{\pm 0.10}$	1.74$_{\pm 0.04}$	12.89$_{\pm 0.25}$	16.93$_{\pm 0.29}$	5.26$_{\pm 0.09}$	8.52$_{\pm 0.23}$	10.56$_{\pm 0.26}$	3.39$_{\pm 0.10}$
UltraGCN [18]	5.62$_{\pm 0.42}$	7.74$_{\pm 0.49}$	2.16$_{\pm 0.15}$	11.14$_{\pm 0.65}$	14.27$_{\pm 0.74}$	4.39$_{\pm 0.26}$	2.30$_{\pm 0.13}$	3.04$_{\pm 0.15}$	0.89$_{\pm 0.05}$
SimpleX [17]	1.01$_{\pm 0.06}$	1.53$_{\pm 0.10}$	0.44$_{\pm 0.03}$	6.82$_{\pm 0.32}$	8.70$_{\pm 0.34}$	2.64$_{\pm 0.12}$	0.77$_{\pm 0.01}$	1.01$_{\pm 0.06}$	0.30$_{\pm 0.01}$
GraphSAGE [10]	7.22$_{\pm 0.11}$	10.87$_{\pm 0.16}$	2.76$_{\pm 0.05}$	19.84$_{\pm 0.25}$	24.63$_{\pm 0.30}$	8.17$_{\pm 0.08}$	10.26$_{\pm 0.39}$	12.39$_{\pm 0.49}$	4.14$_{\pm 0.18}$
GAT [25]	4.65$_{\pm 0.27}$	7.60$_{\pm 0.35}$	1.72$_{\pm 0.08}$	17.59$_{\pm 0.44}$	22.66$_{\pm 0.47}$	6.87$_{\pm 0.18}$	5.17$_{\pm 0.17}$	7.04$_{\pm 0.16}$	1.83$_{\pm 0.05}$
GIN [30]	7.62$_{\pm 0.13}$	11.24$_{\pm 0.18}$	2.92$_{\pm 0.04}$	21.44$_{\pm 0.20}$	25.79$_{\pm 0.18}$	8.99$_{\pm 0.10}$	10.24$_{\pm 0.08}$	12.00$_{\pm 0.11}$	4.38$_{\pm 0.04}$
SGC [28]	6.01$_{\pm 0.00}$	10.24$_{\pm 0.00}$	2.18$_{\pm 0.00}$	12.43$_{\pm 0.10}$	16.52$_{\pm 0.10}$	473$_{\pm 0.07}$	5.18$_{\pm 0.04}$	7.20$_{\pm 0.05}$	2.03$_{\pm 0.02}$
S^{2}GC [35]	7.15$_{\pm 0.00}$	10.72$_{\pm 0.00}$	2.59$_{\pm 0.00}$	14.67$_{\pm 0.34}$	20.30$_{\pm 0.26}$	4.82$_{\pm 0.13}$	7.64$_{\pm 0.07}$	10.21$_{\pm 0.06}$	2.89$_{\pm 0.03}$
SIGN [7]	5.58$_{\pm 0.76}$	9.38$_{\pm 0.99}$	2.00$_{\pm 0.24}$	15.84$_{\pm 0.40}$	19.09$_{\pm 0.55}$	6.55$_{\pm 0.14}$	7.64$_{\pm 0.06}$	9.12$_{\pm 0.11}$	3.25$_{\pm 0.04}$
GBP [3]	9.24$_{\pm 0.15}$	13.23$_{\pm 0.20}$	3.49$_{\pm 0.04}$	22.65$_{\pm 0.32}$	27.36$_{\pm 0.27}$	9.55$_{\pm 0.09}$	$\underline{12.07}_{\pm 0.24}$	$\underline{14.30}_{\pm 0.26}$	$\underline{4.92}_{\pm 0.11}$
GAMLP [32]	12.50$_{\pm 0.25}$	17.22$_{\pm 0.33}$	4.58±0.11	24.77$_{\pm 0.39}$	30.01$_{\pm 0.35}$	$\underline{10.22}_{\pm 0.19}$	11.39$_{\pm 0.26}$	13.48$_{\pm 0.26}$	4.68$_{\pm 0.12}$
LightGCN [12]	12.26$_{\pm 0.63}$	17.55$_{\pm 0.72}$	4.78$_{\pm 0.23}$	21.74$_{\pm 0.35}$	27.49$_{\pm 0.39}$	8.26$_{\pm 0.14}$	5.91$_{\pm 0.11}$	7.98$_{\pm 0.11}$	2.21$_{\pm 0.05}$
PPRGo [2]	$\underline{13.99}_{\pm 0.19}$	$\underline{18.58}_{\pm 0.20}$	$\underline{5.30}_{\pm 0.08}$	$\underline{25.48}_{\pm 0.58}$	$\underline{31.30}_{\pm 0.59}$	9.54$_{\pm 0.16}$	10.64$_{\pm 0.08}$	12.27$_{\pm 0.09}$	4.22$_{\pm 0.04}$
xGCN [ours]	**16.07**$_{\pm 0.21}$	**21.43**$_{\pm 0.24}$	**6.25**$_{\pm 0.06}$	**31.44**$_{\pm 0.09}$	**36.37**$_{\pm 0.14}$	**13.41**$_{\pm 0.09}$	**18.52**$_{\pm 0.18}$	**21.19**$_{\pm 0.09}$	**7.61**$_{\pm 0.09}$
Improv.	+14.87%	+15.34%	+17.92%	+23.39%	+16.20%	+31.21%	+53.43%	+48.18%	+54.67%

Table 3: Results of training on 100m Xbox social graph

	R@100	N@100	#. param	Training Time
RandNE	0.18	0.09	0	16 minutes
node2vec	2.50	0.54	3.2e9	72 hours
LightGCN	0.42	0.14	3.2e9	72 hours
PPRGo	3.68	1.19	3.2e9	72 hours
GraphSAGE	4.54	1.43	3.2e9 + 1e4	72 + 30 hours
GBP	4.54	1.48	3.2e9 + 6.7e4	72 + 15 hours
xGCN	**6.10**	**1.85**	1.1e6	11 hours

the learning rate and GCN layers so that each model can achieve its best performance with the hard 72 hours time constraint. For node2vec, the traditional implementations cannot scale to large graphs due to the existence of the transition probability matrix. To address issues, we especially implement a highly efficient and scalable version of node2vec by ourselves. Technical details can be referred to in Appendix A.4. Now one epoch of node2vec costs about 24 hours. We find that directly training GraphSAGE with each node assigning a learnable embedding vector will lead to very poor performance. Thus, we use node2vec's results to initialize GraphSAGE's embedding table, and we denote the *Total Training Time* of GraphSAGE as *72+30 hours*, with 72 hours of node2vec training and 30 hours of GraphSAGE training. For xGCN, one epoch costs about 51 minutes, and it converges at the 9th epoch, so the total training time is less than 11 hours (because we spare some tolerance epochs before early stopping). The CPU memory consumption of xGCN is 92 GB. The accuracy comparison is shown in Table 3, from which



(a) LiveJournal (b) Xbox-3m

Figure 4: Comparisons of learning curves of xGCN with different propagation times. Each red point signifies the epoch time at which propagation is initiated.

we can see that xGCN uses less time than all the learnable baselines and achieves much better accuracy performance.

3.5 Ablation Study

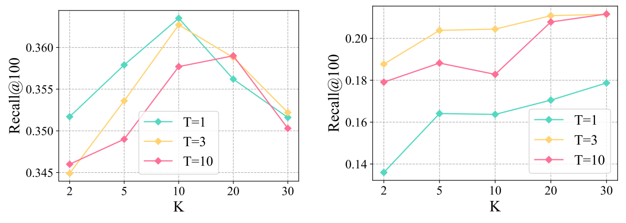
We perform ablation studies to justify the necessity of some key components: (1) the iterative refresh-then-propagate training framework, (2) the *scaling neural network* (SNN) in the RefNet, and (3) the warm-up training stage. The results are reported in Table 4 (due to the space limitation, we move the results on the Pokec dataset to the appendix), where we can have the following conclusions: (i) shows that purely propagating on the randomly initialized embeddings is hard to encode any useful information. (ii.a) and (ii.b) indicate that RefNet has a certain ability to transform relatively low-quality embeddings into more representative ones, however,

Table 4: Ablation studies for xGCN. Notation explanations: (i) Pure-prop: only do propagation for multiple times. (ii.a) Random-RefNet: randomly initialize the embedding table, then learn RefNet, do not perform refresh or do propagation. (ii.b) Prop-RefNet: first do propagation (based on (i)), then learn RefNet. (ii.c) Prop-RefNet-refresh: first do propagation (based on (i)), then learn RefNet, perform refresh but do no more propagation. (iii.a) Full K=0: set $K = 0$, which means removing the warm-up training stage. (iii.b) Full K=Inf: set $K = \infty$, which means the training process only contains the warm-up stage. Group (iv) analyses the effectiveness of SSN. For example, "2L-FFN-SSN" denotes the RefNet is composed of a 2-layer FFN and an SSN. "Res" means replacing the SSN with a common residual connection. "3L-FFN-SSN (Full)" equals to the final xGCN.

	LiveJournal			**Xbox-3m**		
	R@50	R@100	N@100	R@50	R@100	N@100
(i) Pure-prop	0.13	0.26	0.05	0.04	0.06	0.01
(ii.a) Random-RefNet	0.19	0.32	0.07	0.59	0.73	0.21
(ii.b) Prop-RefNet	24.28	28.29	9.61	7.11	9.16	2.37
(ii.c) Prop-RefNet-refresh	24.36	28.35	9.45	9.32	10.53	3.70
(iii.a) Full K=0	29.07	32.47	12.32	14.55	16.65	5.95
(iii.b) Full K=Inf	28.96	34.61	12.24	18.05	20.81	7.29
(iv.a) 2L-FFN-Res	23.46	29.96	7.55	13.86	17.77	4.89
(iv.b) 2L-FFN-SSN	24.65	30.05	8.75	16.88	19.93	6.10
(iv.c) 3L-FFN-Res	18.93	24.17	6.47	9.95	13.67	3.30
(iv.d) 3L-FFN-SSN (Full)	**31.44**	**36.37**	**13.41**	**18.52**	**21.19**	**7.61**

only learning a RefNet is still far from achieving satisfying performance. The comparison of (iv.d) and (ii.c) suggests that without the propagation operation, the iterative refresh-then-training process cannot continuously improve the quality of the embeddings. (iii.a) and (iii.b) indicate the necessity of the two-stage training strategy as well as the refresh controlling mechanism. (iv.a) - (iv.d) demonstrate the necessity of the SNN, when it is replaced with a classical ResNet [11], the performance is greatly affected, and the impact of SNN on the 3-layer FFN is greater than it on the 2-layer FFN.

To gain a clearer understanding of the benefits of propagation for xGCN, we fixed the maximum propagation times to 1, 3, 5, and 10 and plotted the accuracy curves on the validation set in Figure 4. E.g., *prop3* means that when the propagation time reaches 3, we stop further propagation of xGCN and only train the RefNet in the subsequent epochs. *full* indicates a normal setting of xGCN. Each red point in the figure indicates an epoch time when propagation is triggered. Figure 4 demonstrates that propagation is critical for xGCN to converge to a satisfying state. Inadequate times of propagation lead to poor performance. On the other hand, the required maximum number of propagations is not large, i.e., usually, 10 is close to the best performance. Propagation and RefNet learning should be placed under the iterative learning framework, as a counter-example, in Figure 4, the transverse line *RandNE(10)+RefNet* means that we directly propagate the embedding for 10 times and then start training the RefNet until it converges. Its performance is much worse than the corresponding *prop10* version, which indicates the necessity of iterative update of propagation and RefNet.



(a) LiveJournal (b) Xbox-3m

Figure 5: Hyper-parameter study on T and K

Table 5: Hyper-parameter studies: comparisons of FFN structures and graph propagation methods

	LiveJournal			**Xbox-3m**		
	R@50	R@100	N@100	R@50	R@100	N@100
Results of different FFN structures						
2-128	18.01	26.25	5.50	14.03	16.87	5.25
2-1024	24.65	30.05	8.75	16.88	19.93	6.10
3-128	20.66	28.13	6.26	13.79	16.88	4.96
3-1024	**31.44**	**36.37**	**13.41**	**18.52**	**21.19**	**7.61**
Results of different graph propagation methods						
$\tilde{\mathbf{A}}$	**31.44**	**36.37**	**13.41**	**18.52**	**21.19**	**7.61**
$\tilde{\mathbf{A}}^2$	29.26	34.86	11.84	16.97	19.83	6.63
$\tilde{\mathbf{\Pi}}$	23.15	29.50	8.33	15.83	18.32	6.16

3.6 Hyper-parameter Sensitivity

Here we study how xGCN is impacted by key hyper-parameters (we report the results on the Pokec dataset in the appendix). First, in Figure 5 we can observe that a proper setting of T and K in the refresh controller is important, and the best setting differs in different datasets. On Pokec and Xbox-3m, the model is more sensitive with T, while on LiveJournal, the model is more sensitive with K. In general, a setting of $[T = 3, K = 10]$ can lead to a decent performance. Moreover, Table 5 shows the impact of the FFN structure in RefNet and graph propagation methods. As in expectation, a larger dimension size for the middle layers leads to better performance, which is verified by comparing *2-1024* with *2-128* and comparing *3-1024* with *3-128*. Meanwhile, a deeper structure can boost the performance, e.g., *3-1024* can consistently outperform *2-1024* on three datasets. As for graph propagation methods, we compare three candidates for the propagation matrix $\mathbf{P}$ in the embedding propagation step: first-order neighborhood $\tilde{\mathbf{A}}$, second-order neighborhood $\tilde{\mathbf{A}}^2$, and the top-k PPR neighbors $\tilde{\mathbf{\Pi}}$. The simplest one, $\tilde{\mathbf{A}}$, achieves the best performance consistently. A possible reason is that the recurrent *[propagation, refinement, refresh]* process already conveys messages from the high-order neighborhood.

3.7 Visualization of Node Embeddings

At last, we visualize node embeddings with the t-SNE package [24] to see if the learned embeddings have community patterns. We first partition a social graph into 100 clusters with METIS, then

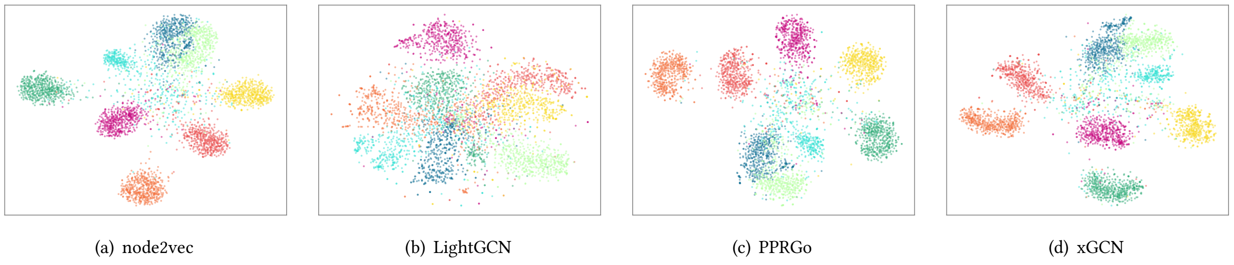


Figure 6: Visualization of embeddings on the Pokec dataset. Embeddings are mapped to a 2-D space with the t-SNE toolkit. We partition the Pokec graph by metis, then sample 8 clusters and assign each of them a dedicated color. Each point represents a node sampled from a cluster.

tag each node with its affiliated cluster ID as its label. For better conciseness, we randomly select 8 clusters and assign each cluster a dedicated color. Figure 6 shows the t-SNE plots on the Pokec dataset of four embedding models. Just like other classical models, xGCN can also generate meaningful embeddings from which the community information can be distinguished and the graph layout can be clearly displayed.

4 RELATED WORK

Shallow graph embedding. Factorization-based methods, such as Word2Vec [19] and Matrix Factorization [14], have been successfully applied for sparse high-dimensional data. Motivated by them, researchers try to factorize nodes on a graph with embedding vectors. Perozzi et al. [21] introduce the method DeepWalk, which draws random walks over a graph and learns embedding vectors to measure co-occurring nodes within a window size. Grover et al. [9] argue that the random walk process should consider the second order of graph structure, instead of only randomly selecting a next step from the first order neighborhood. Yao et al. [31] propose a unified framework for random walk-based graph embedding models with an efficient Metropolis-Hasting sampling method. Instead of using random walks, Tang et al. [23] design some objective functions to encode both first-order and second-order graph proximities. As for scalable implementations for shallow graph embedding methods, PBG [16] divides nodes into n buckets, so that the adjacency matrix is decomposed into $n \times n$ non-overlapping blocks. Multiple processes can simultaneously train edges from different blocks with minimum data synchronization. GraphVite [36] uses a similar approach to partition the graph, and it further introduces some efficient collaboration strategies for acceleration with GPUs.
Graph neural networks. GNNs explicitly model a node's neighborhood information into its embedding vector with neural networks, they demonstrate superior performance compared with shallow graph embedding models. Kipf and Welling [13] propose the *graph convolutional network* (GCN) for semi-supervised classification. Velickovic et al. [25] propose the *graph attention networks* (GATs), which use self-attentional layers for specifying different weights to different neighbors. Xu et al. [30] introduce a theoretical framework to help analyze the expressive power of GNNs, especially on what types of graph structures can or cannot be distinguished by some popular GNNs. He et al. [12] observe that for the link prediction task in recommender systems, feature transformation, and nonlinear activation are useless and even reduce the performance. Thus, they propose LightGCN, a simplified structure of GCN. The aggregation of the neighborhood in GNNs makes them hard to scale to large graphs. To address this issue, Hamilton et al. [10] sample a fixed size of nodes in each hop of the neighborhood, so that computational cost will not grow exponentially with the number of hops. Chiang et al. [5] partition an original big graph into subgraphs. At each mini-batch training step, the neighborhood for convolutional operations is restricted in a selected subgraph. Bojchevski et al. [2] introduce PPRGo, which breaks the classical message-passing scheme of GNNs. PPRGo uses approximated *Personalized Page Rank* (PPR) to simplify the information diffusion on the graph. It can bring significant speed gains while achieving competitive accuracy performance, and particularly achieves state-of-the-art performance on social-network-related tasks.

5 CONCLUSIONS

In this paper, we present xGCN, a novel GNN model that improves the accuracy, efficiency, and scalability of graph-based embeddings for link prediction tasks. Unlike traditional GNNs that stack multiple layers of neighborhood aggregation and optimize all parameters through end-to-end gradient back-propagation, xGCN learns graph structure information in a progressive *[propagation, refinement, refresh]* manner, reducing the bottleneck caused by heavy trainable embedding tables. Our experiments on three large-scale social network datasets demonstrate the superiority of xGCN for link predictions. This research opens up new possibilities for large-scale graph-based embeddings in various applications such as recommendation systems, social network analysis, and knowledge graph embeddings. Future work will focus on extending xGCN to other tasks such as node classification and designing mechanisms that can steer the dynamics and diversity of link recommendations to mitigate radicalization and polarization [22] in social networks.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.62172174, 62127808).

REFERENCES

[1] Uri Alon and Eran Yahav. 2021. On the Bottleneck of Graph Neural Networks and its Practical Implications. In *Proceedings of the 9th International Conference on Learning Representations*. 1–12.

[2] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling Graph Neural Networks with Approximate PageRank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2464–2473.

[3] Ming Chen, Zhewei Wei, Bolin Ding, Yaliang Li, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2020. Scalable Graph Neural Networks via Bidirectional Propagation. In *Proceedings of the Annual Conference on Neural Information Processing Systems 2020*. 14556–14566.

[4] Yihua Chen and Maya R. Gupta. 2010. EM Demystified: An Expectation-Maximization Tutorial. *UWEE Technical Report* UWEETR-2010-0002 (2010), 1–26.

[5] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 257–266.

[6] Yuxin Fang, Li Dong, Hangbo Bao, Xinggang Wang, and Furu Wei. 2023. Corrupted Image Modeling for Self-Supervised Visual Pre-Training. In *Proceedings of the 11th International Conference on Learning Representations*. 1–12.

[7] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Ben Chamberlain, Michael Bronstein, and Federico Monti. 2020. SIGN: Scalable Inception Graph Neural Networks. In *Proceedings of Graph Representation Learning and Beyond Workshop at the 37th International Conference on Machine Learning*. 1–9.

[8] Palash Goyal and Emilio Ferrara. 2018. Graph Embedding Techniques, Applications, and Performance: A Survey. *Knowledge Based Systems* 151 (2018), 78–94.

[9] 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.

[10] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31th International Conference on Neural Information Processing Systems*. 1024–1034.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[12] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yong-Dong Zhang, and Meng Wang. 2020. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 639–648.

[13] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*. 1–14.

[14] Yehuda Koren, Robert M. Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer* 42, 8 (2009), 30–37.

[15] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *Proceedings of the 9th International Conference on Learning Representations*. 1–14.

[16] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alexander Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Proceedings of Machine Learning and Systems 2019*. 120–131.

[17] Kelong Mao, Jieming Zhu, Jinpeng Wang, Quanyu Dai, Zhenhua Dong, Xi Xiao, and Xiuqiang He. 2021. SimpleX: A Simple and Strong Baseline for Collaborative Filtering. In *Proceedings of the 21st ACM Conference on Information and Knowledge Management*. 1243–1252.

[18] Kelong Mao, Jieming Zhu, Xi Xiao, Biao Lu, Zhaowei Wang, and Xiuqiang He. 2021. UltraGCN: Ultra Simplification of Graph Convolutional Networks for Recommendation. In *Proceedings of the 21st ACM Conference on Information and Knowledge Management*. 1253–1262.

[19] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the 1st International Conference on Learning Representations*. 1–12.

[20] Todd K. Moon. 1996. The Expectation-maximization Algorithm. *IEEE Signal Processing Magazine* 13, 6 (1996), 47–60.

[21] 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.

[22] Fernando P. Santos, Yphtach Lelkes, and Simon A. Levin. 2021. Link Recommendation Algorithms and Dynamics of Polarization in Online Social Networks. *Proceedings of the National Academy of Sciences* 118, 50 (2021), e2102141118.

[23] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-Scale Information Network Embedding. In *Proceedings of the 24th International Conference on World Wide Web*. 1067–1077.

[24] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data Using t-SNE. *Journal of Machine Learning Research* 9, 86 (2008), 2579–2605.

[25] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations*. 1–12.

[26] Yiqi Wang, Chaozhuo Li, Mingzheng Li, Wei Jin, Yuming Liu, Hao Sun, Xing Xie, and Jiliang Tang. 2022. Localized graph collaborative filtering. In *Proceedings of the 2022 SIAM International Conference on Data Mining (SDM)*. SIAM, 540–548.

[27] Yiqi Wang, Chaozhuo Li, Zheng Liu, Mingzheng Li, Jiliang Tang, Xing Xie, Lei Chen, and Philip S Yu. 2022. An Adaptive Graph Pre-training Framework for Localized Collaborative Filtering. *ACM Transactions on Information Systems* 41, 2 (2022), 1–27.

[28] Felix Wu, Amauri H. Souza Jr., Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. 2019. Simplifying Graph Convolutional Networks. In *Proceedings of the 36th International Conference on Machine Learning*. 6861–6871.

[29] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24.

[30] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *Proceedings of the 7th International Conference on Learning Representations*. 1–17.

[31] Xingyu Yao, Yingxia Shao, Bin Cui, and Lei Chen. 2021. UniNet: Scalable Network Representation Learning with Metropolis-Hastings Sampling. In *Proceedings of the 37th IEEE International Conference on Data Engineering*. 516–527.

[32] Wentao Zhang, Ziqi Yin, Zeang Sheng, Yang Li, Wen Ouyang, Xiaosen Li, Yangyu Tao, Zhi Yang, and Bin Cui. 2022. Graph Attention Multi-Layer Perceptron. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4560–4570.

[33] Yiding Zhang, Chaozhuo Li, Xing Xie, Xiao Wang, Chuan Shi, Yuming Liu, Hao Sun, Liangjie Zhang, Weiwei Deng, and Qi Zhang. 2022. Geometric Disentangled Collaborative Filtering. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 80–90.

[34] Ziwei Zhang, Peng Cui, Haoyang Li, Xiao Wang, and Wenwu Zhu. 2018. Billion-Scale Network Embedding with Iterative Random Projection. In *Proceedings of the 2018 IEEE International Conference on Data Mining*. 787–796.

[35] Hao Zhu and Piotr Koniusz. 2021. Simple Spectral Graph Convolution. In *Proceedings of the 9th International Conference on Learning Representations*. 1–12.

[36] Zhaocheng Zhu, Shizhen Xu, Jian Tang, and Meng Qu. 2019. GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding. In *Proceedings of The Web Conference 2019*. 2494–2504.

A APPENDIX

A.1 Theoretical Analysis

A.1.1 EM Formulation. The framework and optimization of xGCN can be formulated and explained by the EM algorithm [20]. As a generative process, each node i's base embedding vector $\mathbf{z}_i$ is sampled from a standard Gaussian prior $\mathcal{N}(\mathbf{0}, \mathbf{I}_d)$. The base embedding vector $\mathbf{z}_i$ is transformed via a decoder $Dec(\cdot)$ to generate a distribution probability $\boldsymbol{\pi}(\mathbf{z}_i)$ over N nodes indicating the edge existence probability. In this paper, we consider the task of dense link prediction, so we first transform $\mathbf{z}_i$ with the refinement network, denoted by $\mathbf{z}'_i = f(\mathbf{z}_i\ ; \theta, \mathcal{G})$, then use the similarity (such as dot product or cosine similarity) of refined representations to determine the likelihood of an edge $A_{i,j}$:

$$\boldsymbol{\pi}(\mathbf{z}_i)_j \propto exp(Dec(\mathbf{z}_i; \theta, \mathcal{G})_j) = exp(f(\mathbf{z}_i\ ; \theta, \mathcal{G}) \cdot f(\mathbf{z}_j\ ; \theta, \mathcal{G})) \quad (7)$$

$$A_{i,j} \sim Bern(\boldsymbol{\pi}(\mathbf{z}_i)_j) \quad (8)$$

where $\mathcal{G}$ denotes the graph structure, exp denotes a softmax transformation and $Bern$ is the Bernoulli distribution. Unlike traditional models which usually make $\mathbf{z}_*$ as learnable embedding parameters or use another graph encoder $Enc(\cdot)$ to derive $\mathbf{z}_*$ then train all the parameters in an end-to-end manner, in xGCN, we take $\mathbf{z}_*$ as unobserved latent variables and use the EM algorithm to optimize the model.

For notation simplicity, let $x_{(k)}$ denote the k-th data sample (which is a pair of nodes such as $A_{i,j}$) and $z_{(k)}$ denote the latent variables associated with $x_{(k)}$ (which is $\mathbf{z}_i$ and $\mathbf{z}_j$). Let $\theta(t)$ denote the learned parameters at time step t. The EM algorithm takes the following form:

E-step: Given the estimated parameter $\theta(t)$ at iteration t, compute the expectation of latent variables:

$$Q(z_{(k)}) = p(z_{(k)}|x_{(k)}; \theta(t)) \quad (9)$$

M-step: Update θ to maximize the expected likelihood of the observed data (which is also called the Q-function):

$$Q(\theta|\theta(t)) = \arg\max_{\theta}\ \mathbb{E}_{z_{(k)} \sim Q(z_{(k)})}[log\ p(x_{(k)}, z_{(k)}|\theta)] \quad (10)$$

We assume the probability mass function $Q(z_{(k)})$ follows a sharp distribution, such as $\mathcal{N}(g(x_{(k)}; \theta(t)), \sigma^2\mathbf{I})$, where $g(x_{(k)}; \theta(t)) = \arg\max_{z_{(k)}} p(z_{(k)}|x_{(k)}; \theta(t))$ and $\sigma \longrightarrow 0$. Then, Eq. (10) can be simplified to:

$$Q(\theta|\theta(t)) = \arg\max_{\theta}\ log\ [\ p(x_{(k)}|g(x_{(k)}; \theta(t)), \theta) \cdot p(g(x_{(k)}; \theta(t))\ ] \quad (11)$$

The next question is how to determine $g(x_{(k)}; \theta(t))$. Here, we use the proof by contradiction to demonstrate that $g(x_{(k)}; \theta(t))$ equals to the output of the refinement network.

Proof. Assume that $g(x_{(k)}; \theta(t))$ does not equal to the output of the refinement network which is denoted as $z'_{(k)} = f(z_{(k)}(t); \theta(t), \mathcal{G})$. Then, there exists another $z''_{(k)} = g(x_{(k)}; \theta(t))$ and $z'_{(k)} \neq z''_{(k)}$. According to Bayes' theorem,

$$p(z_{(k)}|x_{(k)}; \theta(t)) = \frac{p(x_{(k)}|z_{(k)}; \theta(t)) \cdot p(z_{(k)}; \theta(t))}{p(x_{(k)}; \theta(t))}$$
$$\propto p(x_{(k)}|z_{(k)}; \theta(t)) \cdot p(z_{(k)}) \quad (12)$$

which indicates that $p(x_{(k)}|z''_{(k)}; \theta(t)) \cdot p(z''_{(k)}) > p(x_{(k)}|z'_{(k)}; \theta(t)) \cdot p(z'_{(k)})$. But this cannot be true, because the purpose of the M-step is via optimizing the parameters $\theta(t)$, so that the refinement network $f(z_{(k)}(t); \theta(t), \mathcal{G})$ can output a value $z'_{(k)}$ to maximize Eq.(12). Hence we have a contradiction and so $g(x_{(k)}; \theta(t))$ equals to the output of the refinement network. □

Once $g(x_{(k)}; \theta(t))$ is determined, Eq. (11) can be directly optimized by SGD.

A.1.2 Analysis of Convergence. The convergence of xGCN can be easily proved by leveraging the convergence theory of the EM algorithm. Here we follow the theoretical derivations in [4].

Proof. Let $L(\theta)$ denote the log-likelihood of data x (for notation simplicity, we drop the subscript in $x_{(k)}$) modeled with parameter θ. Convergence can be guaranteed as long as we can prove that $L(\theta(t+1)) > L(\theta(t))$. To this end, we start by

$$\begin{aligned} L(\theta) &= log\ p(x|\theta) \\ &= log \int_{\mathcal{X}(z)} p(x, z|\theta) dz \\ &= log \int_{\mathcal{X}(z)} \frac{p(x, z|\theta)}{p(z|x, \theta(t))} p(z|x, \theta(t)) dz \\ &= log\ \mathbb{E}_{z \sim p(z|x, \theta(t))} \left[\frac{p(x, z|\theta)}{p(z|x, \theta(t))} \right] \quad (13) \\ &\geq \mathbb{E}_{z \sim p(z|x, \theta(t))} \left[log\ \frac{p(x, z|\theta)}{p(z|x, \theta(t))} \right] \quad (14) \\ &= Q(\theta|\theta(t)) + \mathbb{E}_{z \sim p(z|x, \theta(t))}\ [-log\ p(z|x, \theta(t))] \end{aligned}$$

where $\mathcal{X}(z)$ denotes the support of z. From Eq.(13) to Eq.(14) we apply Jensen's inequality because $log(\cdot)$ is a concave function. Let $H(z|x, \theta(t))$ denote $\mathbb{E}_{z \sim p(z|x, \theta(t))}\ [-log\ p(z|x, \theta(t))]$. Then,

$$L(\theta) \geq Q(\theta|\theta(t)) + H(z|x, \theta(t))$$

Note that $H(z|x, \theta(t))$ does not depends on θ.

$$\begin{aligned} &Q(\theta(t)|\theta(t)) + H(z|x, \theta(t)) \\ &= \mathbb{E}_{z \sim p(z|x, \theta(t))}\ [log\ p(x, z|\theta(t))] + \mathbb{E}_{z \sim p(z|x, \theta(t))}\ [-log\ p(z|x, \theta(t))] \\ &= E_{z \sim p(z|x, \theta(t))} log\ p(x|z, \theta(t)) \\ &= log\ p(x|\theta(t) \\ &\triangleq L(\theta(t)) \end{aligned}$$

By definition of $Q(\theta|\theta(t))$ from Eq.(10), we have $Q(\theta(t+1)|\theta(t)) \geq Q(\theta(t)|\theta(t))$, thus we can conclude that:

$$\begin{aligned} &L(\theta(t+1)) - L(\theta(t)) \\ &\geq [Q(\theta(t+1)|\theta(t)) + H(z|x, \theta(t))] - [Q(\theta(t)|\theta(t)) + H(z|x, \theta(t))] \\ &= Q(\theta(t+1)|\theta(t)) - Q(\theta(t)|\theta(t)) \\ &\geq 0 \end{aligned}$$

□

A.1.3 Explanation of Eq.(9). Although Eq.(9) is intuitively understandable, here we provide some theoretical descriptions. The likelihood of a data sample is defined by

$$\begin{aligned} L(\theta) &= log\ p(x|\theta) \\ &= log \int_{\mathcal{X}(z)} p(x, z|\theta) dz \\ &= log \int_{\mathcal{X}(z)} Q(z) \frac{p(x, z|\theta)}{Q(z)} dz \\ &\geq \mathbb{E}_{z \sim Q(z)}\ log \frac{p(x, z|\theta)}{Q(z)} \end{aligned} \tag{15}$$

Since the $log(\cdot)$ function is not linear, the equality of 15 holds if and only if value inside $log(\cdot)$ is a constant c, i.e.,

$$p(x, z|\theta) = c \cdot Q(z)$$

$$p(x|\theta) = \int_z p(x, z|\theta) = \int_z c \cdot Q(z) dz = c$$

So, we can have

$$Q(z) = \frac{p(x, z|\theta)}{c} = \frac{p(x, z|\theta)}{p(x|\theta)} = p(z|x; \theta)$$

A.2 Task Settings and Datasets

We use two biggest social network datasets from SNAP[2], **Pokec** and **LiveJournal**, and one industrial social network dataset, **Xbox**, which is provided by Xbox Gaming Corp., and it is a gaming social network. The complete Xbox dataset contains about 100 million nodes, for a comprehensive comparison with baselines, we sample a medium-size subgraph including 3 million nodes and denote it as Xbox-3m. Some basic statistics are listed in Table 1. We evaluate model performance on the node recommendation task. Nodes in these graphs are users and edges are the *follow* relation between users. We evaluate model performance on the node recommendation task, i.e., given a user u, the task is to find related users that u will follow from the whole social graph. Specifically, a small portion of edges is removed from the original graph to construct the validation set and the test set. Given an edge (u, v), v is considered as a positive node, and the users that are not followed by u are treated as negative nodes. The validation set and the test set contain 1000 and 50000 positive edges respectively (the number of users may be less than the number of edges since a user can have multiple positive nodes). During training, we sample an edge (u, v) from the graph and sample a negative pair $(u, \hat{v})$ to calculate loss by Eq.(3). During the evaluation, we do not perform candidate sampling, the purpose is to retrieve positive nodes from the whole graph.

A.3 Baseline Settings

The embedding dimensions are set to 64 for all models (except for the Xbox-100m dataset, the dimension is set to 32). For GraphSAGE, GAT, GIN, SGC, S^{2}GC, SIGN, GBP, and GAMLP, we find they are hard to train from scratch, so we initialize their base embedding tables with a well-trained node2vec model. In most cases, we find it is better to freeze the base embedding table for the GNNs as static features, and only learn the weight parameters of GCN layers. We use the BPR loss [12] for GraphSAGE, GAT, GIN, SGC, S^{2}GC, SIGN, GBP, GAMLP, LightGCN, PPRGo, and xGCN. For UltraGCN and SimpleX, we use the loss functions in their original papers. In xGCN, the neural architecture of RefNet is: $[Linear(64, 1024), Tanh, Linear(1024, 1024), Tanh, Linear(1024, 64)]$. The structure of SNN is $[Linear(64, 32), Tanh, Linear(32, 1), Sigmoid]$. T_{tol} is set to 3. We search the hyper-parameters for each baseline model, and the code and detailed configurations to reproduce the results can be found at https://github.com/CGCL-codes/xGCN.

[2]http://snap.stanford.edu/data/index.html

Table A1: Efficiency (training time per epoch, in second) and memory (in GB) comparisons of different node2vec implementations. For PyG-GPU version, we list both its CPU memory and GPU memory consumption (CPU memory + GPU memory).

	Pokec		LiveJournal		Xbox-3m	
	time	memory	time	memory	time	memory
SNAP	385	50.49	-	OOM	-	OOM
PyG-CPU	5100	4.62	10, 440	7.26	16, 560	8.03
PyG-GPU	56	3.85 + 6.77	217	5.39 + 6.65	165	4.62 + 8.23
Ours	292	1.65	650	2.75	955	3.74

A.4 Scalable Implementation of node2vec

The traditional implementation of node2vec cannot scale to large graphs due to the existence of the transition probability matrix (including the C-implemented version [3] released by the SNAP and the implementation in PyG [4]). So in Table 3 we report the node2vec results by our own implementation, where there are mainly two key components: the graph walking trajectories and word2vec embedding. For the first component, we adopt the rejection sampling method to generate node2vec trajectories (please refer to [31] for detailed theory) without maintaining a transition probability matrix. We accelerate the trajectories generation process with Numba [5], which is an open-source JIT compiler that translates a subset of Python and NumPy code into fast machine code. For the second part, we adopt Gensim's word2vec module, because it is already accelerated with c/c++ through Cython. The two components are chained under the producer-consumer pipeline, so parallelism is guaranteed. Table A1 is a training efficiency comparison between our node2vec implementation with SNAP's C-implementation and PyG. We strictly make both implementations share the same hyper-parameters for training epochs. SNAP implementation cannot even scale to LiveJournal and Xbox-3m datasets. Although PyG-GPU is the fastest one, it cannot scale to the 100m graph dataset.

[3]https://github.com/snap-stanford/snap/tree/master/examples/node2vec
[4]https://www.pyg.org
[5]https://numba.pydata.org