Nimble GNN Embedding with Tensor-Train Decomposition

Chunxing Yin*
Georgia Institute of Technology
USA
cyin9@gatech.edu

Da Zheng
Amazon
USA
dzzhen@amazon.com

Israt Nisa
Amazon
USA
nisisrat@amazon.com

Christos Faloutsos
Amazon
USA
faloutso@amazon.com

George Karypis
Amazon
USA
gkarypis@amazon.com

Richard Vuduc
Georgia Institute of Technology
USA
richie@cc.gatech.edu

ABSTRACT

This paper describes a new method for representing embedding tables of graph neural networks (GNNs) more compactly via tensor-train (TT) decomposition. We consider the scenario where (a) the graph data that *lack* node features, thereby requiring the learning of embeddings during training; and (b) we wish to exploit GPU platforms, where smaller tables are needed to reduce host-to-GPU communication even for large-memory GPUs. The use of TT enables a compact parameterization of the embedding, rendering it small enough to fit entirely on modern GPUs even for massive graphs. When combined with judicious schemes for initialization and hierarchical graph partitioning, this approach can reduce the size of node embedding vectors by 1,659× to 81,362× on large publicly available benchmark datasets, achieving comparable or better accuracy and significant speedups on multi-GPU systems. In some cases, our model without explicit node features on input can even match the accuracy of models that use node features.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**.

KEYWORDS

graph neural networks, tensor-train decomposition, embedding

ACM Reference Format:
Chunxing Yin, Da Zheng, Israt Nisa, Christos Faloutsos, George Karypis, and Richard Vuduc. 2022. Nimble GNN Embedding with Tensor-Train Decomposition. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD'22).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3534678.3539423

1 INTRODUCTION

This paper is motivated by a need to improve the scalability of graph neural networks (GNNs), which has become an important part of the modern graph modeling toolkit in an era when graphs with many billions of nodes and edges are commonplace. For instance, Facebook has billions of users; Amazon has billions of users and billions of items; the knowledge graph Freebase [1] has 1.9 billion triples and thousands of relations; and Google's Knowledge Graph has over 500 billion facts spanning 5 billion entities.[1]

For GNNs, a significant scaling challenge occurs when the graph does not come with explicit node features. In this case, one typically associates a trainable embedding vector with each node. The GNN learns these embeddings via message passing, and these trainable node embeddings become part of the overall model's parameterization, with the parameters updated via back propagation. The problem is the memory requirement for these embeddings, which strains modern GPU-accelerated systems. Because the embedding storage significantly exceeds the capacity of even large-memory

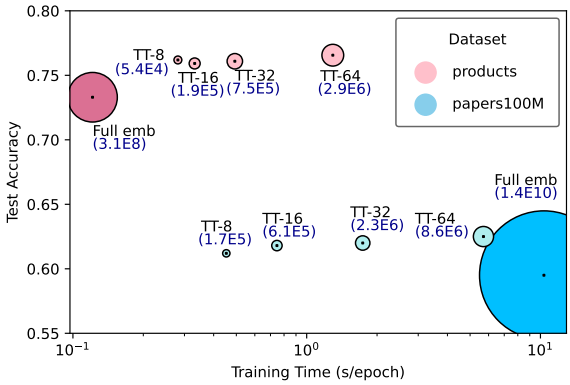

Figure 1: The trade-offs among memory consumption, training time, and accuracy of two OGB node property prediction tasks. The marker area is proportional to the model size, along with the model configuration and number of parameters labeled next to the data points. TT embedding significantly compresses the node embedding in both datasets without sacrificing the model accuracy. Our method accelerates training for large graphs by up to 20×, but introduces overheads to smaller graphs.

*The majority of this work was done when the author was an intern at Amazon.

SIGKDD'22, August 14-18, 2022, Washington, DC, USA

ACM ISBN 978-1-4503-9385-0/22/08…$15.00
https://doi.org/10.1145/3534678.3539423

[1] https://blog.google/products/search/about-knowledge-graph-and-knowledge-panels/

GPUs, a typical strategy is to keep the initial node embeddings on the host CPU and perform mini-batch computations on the GPU [27]. But in this scenario, severe performance bottlenecks arise due to the cost of doing node embedding updates on the CPU as well as host-to-GPU communication.

There are several methods to accelerate training for models with a large number of trainable embeddings [10, 17, 19, 24] (Section 2). Many use hashing and have been applied mainly in recommendation models. Recently, a position-based hash method has been developed to compress the trainable node embeddings for GNNs [9]. However, even this method's compression factor of 10× does not overcome the memory capacity limits of a GPU.

A recent alternative to reduce embedding storage is the tensor-train (TT) decomposition, which was shown to be effective for recommendation models [25]. One can think of TT as a compact parameterization of an embedding table that can be much smaller than naïvely storing the embedding vectors. Therefore, it is natural to consider this method for GNNs, as we do in this paper. However, adopting it is nontrivial: *a priori*, it is unknown how TT will affect accuracy and speed—which TT can actually *worsen*, even if only slightly—as it reduces parameter storage [25].

For GNNs, we can overcome these risks through additional innovations (Section 4). One innovation is in the TT-parameter initialization scheme. Tensorization implicitly increases the depth of the network and results in slower convergence. To address this issue, we consider orthogonal random initialization, which ensures that columns of the initial embedding constructed by the tensor-train method are orthogonal. The other innovation, also motivated directly by the existence of graph structure, is to reorder the nodes in the graph based on a hierarchical graph partition algorithm, namely, METIS [11]. This reordering heuristically groups nodes that are topologically close to each other, which leads to a greater ability of homophilic nodes to share parameters in the TT representation.

Combining TT with the novel initialization techniques and node reordering based on graph partition can actually *improve* model accuracy on standard OGB benchmarks [7] (Section 6). We achieved 1% higher accuracy for the ogbn-arxiv graph, and 3% for ogbn-products using fewer parameters of 415× than full embedding table, and 12.2× than the state-of-the-art position-based embedding [9]. Moreover, our method can train a GNN model on ogbn-products *without* using explicit *a priori* node features to achieve an accuracy comparable to a baseline GNN model that is given such node features. Meanwhile, because TT can compress the node embeddings by a factor of 424× to 81362× on ogbn-papers100M, as we also show, we can fit very large models *entirely on the GPU*. This capability speeds up training by a factor of 23.9× and 19.3× for ogbn-papers100M, which has 111 million nodes, using Graphsage and GAT, respectively.

The essential highlights and tradeoffs of using our TT-based representation versus a full embedding are summarized in Figure 1, which compares accuracy, training time, and the size of the embedding table parameterization for two of the datasets. The TT method, which is associated with the so-called TT-rank hyperparameter, produces significantly smaller representations in both datasets, while maintaining comparable accuracy. However, training time can speed up significantly (ogbn-papers100) or slow down (ogbn-products). These observations both indicate the strong potential of the TT method and raise several questions for future work about how to characterize its efficacy (Section 7).

2 RELATED WORK

Among compression techniques for reducing training and inference costs, magnitude pruning [28], variational dropout [15], and l_0 regularization [13] are the best known. However, these have not been proven to be effective solutions for embedding layer compression. Thus, we focus our summary on techniques most relevant to embedding-layer compression and tensor decomposition.

Embedding Table Compression Techniques: Kalantzi et al. explores constructing the node embeddings in GNNs through a composition of a positional-based shared vector and a node specific vector, each from a small pool of embeddings [9]. This work improves accuracy while reducing memory consumption by up to 34×. However, this level of compression is not enough for the embeddings to fit on a GPU.

An alternative is the use of hashing-based methods, especially for personalization and recommendation. Feature hashing uses a single hash function to map multiple elements to the same embedding vector [24]. However, hash collisions yield intolerable accuracy degradation [26]. The accuracy loss and limited compression ratios appear to be the general issue with unstructured hashing-based methods as well [18]. The application of algebraic methods, such as low-rank approximation on the embedding tables, also suffer unacceptable accuracy losses [4, 6].

Tensorization: Tensor methods, including Tucker decomposition [2], tensor ring (TR) decomposition [23], and tensor train (TT) decomposition [3], have been extensively studied for compressing fully-connected and convolutional layers. In particular, TT and TR offer a structural way approximate full tensor weights and thus are shown to be effective in preserving the DNN weights and accuracy. Tensorized embeddings can produce unique and compact representations, potentially avoiding loss of information caused by mapping uncorrelated items to a same embedding as occurs with hashing. Compressing large embedding layers through TT decomposition has been shown to be effective for recommendation systems [25], where the model can be more than 100× smaller with marginal losses of accuracy. However, this work has not explored the explainability of tensorized networks and can also incur accuracy losses.

3 BACKGROUND

GNNs and trainable embeddings. Graph Neural Networks (GNNs) are deep learning models capable of incorporating topological graph structures and learning node/edge features. They are often modeled using a message-passing framework, where each node aggregates the messages received from its neighboring nodes. Given a graph $G(V, E)$ with vertex set V and edges E, we denote the **node embedding** of $v \in V$ at layer l as $h_v^{(l)}$ and the neighborhood of v as $\mathcal{N}(v)$. A GNN model updates the node embedding via

$$h_v^{(l+1)} = g\left(h_v^{(l)}, \sum_{u \in \mathcal{N}(v)} f\left(h_v^{(l)}, h_u^{(l)}, e_{uv}\right)\right), \tag{1}$$

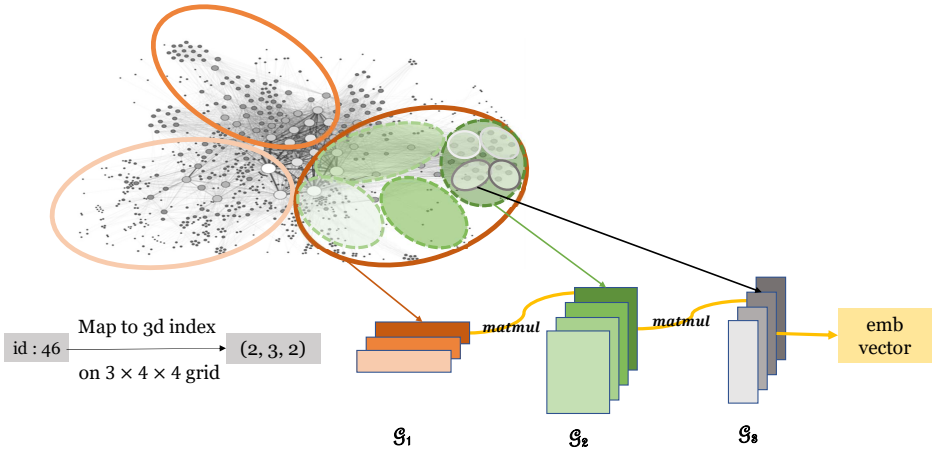

Figure 2: Overview of TT embedding vector construction in GNNs. Embedding vectors are constructed via shared parameters in the corresponding TT cores if the nodes belong to the same cluster in the hierarchical partition.

where e_{uv} is the edge feature associated with the edge $(u, v) \in E$, and $f(\cdot)$ and $g(\cdot)$ are learnable functions that calculate and update the messages, respectively.

In Equation (1), $h_v^{(0)}$ represents the input feature of vertex v. If the input features are unavailable or of low-quality, it is common to place a trainable node embedding module in the GNN and use a one-hot encoding for input features. Let $\mathbf{W} \in \mathbb{R}^{M \times N}$ be the embedding matrix, where M equals the number of vertices and N is the embedding dimension. The embedding vector $\mathbf{w}_i$ of node i is

$$\mathbf{w}_i^T = \mathbf{e}_i^T \mathbf{W}, \tag{2}$$

where $\mathbf{e}_i$ is the one-hot encoding with $i-$th position to be 1, and 0 everywhere else.

Tensor-train decompositions. Let $\mathcal{A}$ be a d-dimensional tensor of shape $N_1 \times N_2 \times \dots N_d$. Its **tensor train (TT)** is the factorization,

$$\mathcal{A}(i_1, i_2, \dots, i_d) = \mathcal{G}_1(:, i_1, :)\mathcal{G}_2(:, i_2, :) \dots \mathcal{G}_d(:, i_d, :), \tag{3}$$

where each three-dimensional tensor $\mathcal{G}_k$ is a **TT core**, of shape $R_{k-1} \times N_k \times R_k$. The values $\{R_k\}_{k=0}^d$ are the **TT ranks**, where $R_0 = R_d = 1$ so that the first and last operands are row and column vectors, respectively.

A TT decomposition can be applied to a matrix, referred to as **TT matrix format (TTM)**. We formalize it as follows. Let $\mathbf{W} \in \mathbb{R}^{M \times N}$ be a matrix and factor each of M and N into any product of integers,

$$M = \prod_{k=1}^{d} m_k \quad \text{and} \quad N = \prod_{k=1}^{d} n_k.$$

Then reshape $\mathbf{W}$ into a $2d$-dimensional tensor of shape $m_1 \times m_2 \times \dots \times m_d \times n_1 \times n_2 \times \dots \times n_d$ and shuffle the axes of the tensor to form $\mathcal{W} \in \mathbb{R}^{(m_1 \times n_1) \times (m_2 \times n_2) \times \dots \times (m_d \times n_d)}$, where

$$\begin{aligned} &\mathcal{W}((i_1, j_1), (i_2, j_2), \dots, (i_d, j_d)) \\ &\quad = \mathcal{G}_1(:, i_1, j_1, :)\mathcal{G}_2(:, i_2, j_2, :) \dots \mathcal{G}_d(:, i_d, j_d, :) \end{aligned} \tag{4}$$

and each 4-d tensor $\mathcal{G}_k \in \mathbb{R}^{R_{k-1} \times m_k \times n_k \times R_k}$ with $R_0 = R_d = 1$. Although $\mathcal{G}_1$ and $\mathcal{G}_d$ are 3-d tensors,we will treat them as 4-d tensors for notional consistency, e.g., $\mathcal{G}_1(0, i_1, j_1, r)$. Equation (4) is our compressed representation of the embedding table $\mathbf{W}$, which we refer to as **TT-emb**.

Table 1: TT-emb related Notations.

Notation	Description
$\mathbf{W} \in \mathbb{R}^{M \times N}$	$M \times N$ trainable embedding table
$\mathcal{W}$	$\mathbf{W}$ reshaped as a $2d$-way tensor
m_i, n_i	the i-th factor of M and N, resp.
d	number of TT cores
$R_i \in \mathbb{R}$	predefined TT ranks
$\mathcal{G}_i \in \mathbb{R}^{R_{i-1} \times M_i \times N_i \times R_k}$	TT-core for matrix decomposition

The choice of factorizations of M and N is arbitrary, and tensor reshaping can be performed in the same way with zero padding. To minimize the number of parameters of the TT-cores, we factor M and N using $m_k = \lceil M^{1/d} \rceil$, and $n_k = \lceil N^{1/d} \rceil$. TTM format reduces the number of parameters for storing a matrix from $O(MN)$ to $O(dR^2(MN)^{1/d})$. Table 7 in Section 6.3 documents the exact embedding table sizes and our parameter choices.

Given the TT-cores $\{\mathcal{G}_k : \mathcal{G}_k \in \mathbb{R}^{R_{k-1} \times m_k \times n_k \times R_k}\}$, we construct an embedding vector for vertex i, which is the $i-$th row of $\mathbf{W}$, via

$$\mathbf{w}_i(j) = W(i, j) = \mathcal{G}_1(:, i_1, j_1, :)\mathcal{G}_2(:, i_2, j_2, :) \dots \mathcal{G}_d(:, i_d, j_d, :) \tag{5}$$

where $i = \sum_{p=1}^{d} i_p \prod_{q=p+1}^{d} m_q$, and $j = \sum_{p=1}^{d} j_p \prod_{q=p+1}^{d} n_q$ for $j = 1, 2, \dots, N$. An entire row of $\mathbf{W}$ can be reconstructed all at once (rather than elementwise) using $d - 1$ matrix multiplications [25]. Unfold each 3-d subtensor of $\mathcal{G}_k(:, i_k, :, :)$ into a matrix of shape $R_{k-1} \times n_k R_k$. Let $\mathbf{w}_i^{(k)} \in \mathbf{R}^{\prod_1^{k-1} n_i \times n_k R_k}$ be the partial product of the first k unfolding matrices. We reshape $\mathbf{w}_i^{(k)}$ to be of shape $\prod_1^k n_i \times R_k$ and proceed with this chain of matrix multiplications, which in the end calculates the entire row $w_i^{(d)} \in \mathbf{R}^{N \times 1}$.

This pipeline for embedding lookup with TTM is illustrated in Figure 2. As an example, assume the graph has $N = 48$ nodes, and the node with id 46 is queried. First, we map the node id to a 3-d

coordinate $(i_1, i_2, i_3) \in \mathbb{Z}_+^{m_1 \times m_2 \times m_3}$, namely $(2, 3, 2)$ in this case. To reconstruct the embedding vector for node 46, we multiply the i_k^{th} slice from TT-core $\mathcal{G}_i$.

4 METHODS

Our method is motivated by two principles. First, we wish to learn a *unique* embedding vector for each node. Doing so intends to reduce information loss or accuracy degradation that might arise when mapping distinct nodes into the same vector, as with hashing (Section 2). Second, we seek node embeddings that reflect *graph homophily* [14]. That is the property where topologically close nodes tend to share similar representations and labels, which might result in similar node embeddings.

The method itself has two major components. We apply TT decomposition to reduce the embedding table size in GNNs and leverage graph homophily by aligning the graph's topology with the hierarchy of the tensor decomposition. To promote convergence, we propose orthogonal weight initialization algorithms for TT-emb in GNNs (Section 4.1). We then incorporate the graph topological information into tensor decomposition by preprocessing the graph through recursive partitioning and rearranging nodes by clusters, which would enforce more similar embedding construction among neighboring nodes. Our use of graph partitioning tries to maximize TT-core weight sharing in the embedding construction (Section 4.2).

4.1 Orthogonal Initialization of TT-cores

Previous studies have suggested thats, comparing to Gaussian initialization, initializing network weights from the orthogonal group can accelerate convergence and overcome the dependence between network width and the width needed for efficient convergence to a global minimum [8]. Therefore, our algorithm for TT-core initialization ensures that the product embedding matrix $\mathbf{W}$ is orthogonal.

For simplicity, we describe and analyze the algorithm assuming a 3-d TT decomposition, which can be easily extended to higher dimensions. Let $\mathbf{W} \in \mathbb{R}^{M \times N}$ be a matrix which we decompose as

$$\mathcal{W}((i_1, j_1), (i_2, j_2), (i_3, j_3)) = \mathcal{G}_1(:, i_1, j_1, :)\mathcal{G}_2(:, i_2, j_2, :)\mathcal{G}_3(:, i_3, j_3, :),$$

where $\mathcal{G}_1 \in \mathbb{R}^{1 \times m_1 \times n_1 \times R}$, $\mathcal{G}_2 \in \mathbb{R}^{R \times m_2 \times n_2 \times R}$, and $\mathcal{G}_3 \in \mathbb{R}^{R \times m_3 \times n_3 \times 1}$.

One intuitive approach is to decompose a random orthogonal matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$. Algorithm 1 describes the process of decomposing a matrix into its TTM format, adapted from the TT-SVD decomposition algorithm [16]. This method ensures the orthogonality of the constructed embedding matrix but not of each single TT-core.

However, constructing a large orthogonal matrix and its TTM decomposition can be slow. We show an efficient algorithm to initialize unfolding matrices of each $\mathcal{G}_i$ from small orthogonal matrices, in a way that ensures the orthogonality of the product embedding matrix $\mathbf{W}$. Formally, we want to initialize the tensor cores so that $\mathbf{W}$ is uniformly random from the space of scaled semi-orthogonal matrix, i.e., those having $\mathbf{W}^T\mathbf{W} = \alpha\mathbf{I}$. Our algorithm iteratively initializes each TT-core as shown in Algorithm 2.

Claim 4.1. *The embedding matrix W initialized by Algorithm 2 is column orthogonal.*

Algorithm 1: TTM Decomposition

Input: Embedding matrix $\mathbf{X} \in \mathbb{R}^{M \times N}$ consists of normalized eigenvectors of L
Output: TT-cores $\mathcal{G}_1, \ldots, \mathcal{G}_d$ of X as defined in Eqn 4

```
1 X = reshape(X, [m_1, m_2 ... m_d, n_1, n_2, ..., n_d]);
2 Y = transpose(X, [m_1, n_1, m_2, n_2 ..., m_d, n_d]);
3 for k = 1 to d-1 do
4     Y = reshape(Y, [R_k * m_k * N_k, :]);
5     Compute truncated SVD Y = UΣV^T + E, rank(U) ≤ R_k;
6     G_k = reshape(U, [R_k, m_k, n_k, R_{k+1}]);
7     Y = ΣV^T;
8 G_d = reshape(Y, [R_d, m_d, n_d, 1]);
```

Algorithm 2: Orthogonal Initialization of TT-cores.

Input: Matrix shapes M, N factorized as $M = \prod_1^d m_i$, $N = \prod_1^d n_i$, and TT-ranks $\{R_0, R_1, \ldots, R_d\}$
Output: TT-cores $\{\mathcal{G}_i\}_{i=1}^d$ s.t. the unfolding matrix of $\mathcal{G}_i$ and product $W \in \mathbb{R}^{M \times N}$ are semi-orthogonal

```
1 for i = 1 to d do
2     Generate n_i R_{i-1} orthogonal vectors {v_1, v_2, ... v_{n_i R_{i-1}}},
        where v_j ∈ R^{m_i R_i} ;
3     for r = 1 to R_{i-1} do
4         for j = 1 to n_i do
5             G_i(r, :, j, :) = reshape(v_{n_i r + j}, [m_i, R_i]);
```

Proof. We denote the factorization of a row index $0 \le p \le M$ as $p = \sum_{\alpha=1}^3 p_\alpha m_\alpha$, and the factorization of a column index $0 \le q \le N$ as $j = \sum_{\alpha=1}^3 q_\alpha n_\alpha$. To show the orthogonality of $\mathbf{W}$, we compute the entry (i, j) in $\mathbf{W}^T\mathbf{W}$

$$\mathbf{W}^T\mathbf{W}(i,j) = \sum_{k=1}^{M} \mathbf{W}^T(i,k)\mathbf{W}(k,j) = \sum_{k=1}^{M} \mathbf{W}(k,i)\mathbf{W}(k,j)$$
$$= \sum_{k_1,k_2,k_3} \mathcal{G}_1(0,k_1,i_1,:)\mathcal{G}_2(:,k_2,i_2,:)(\mathcal{G}_3(:,k_3,i_3,0)\mathcal{G}_3(:,k_3,j_3,0)^T)$$
$$\mathcal{G}_2(:,k_2,j_2,:)^T\mathcal{G}_1(0,k_1,j_1,:)^T. \tag{6}$$

We initialize $\mathcal{G}_3$ as line 1-2 in Algorithm 2. We show that $\mathbf{A} = \sum_{k_3} \mathcal{G}_3(:,k_3,i_3,0)\mathcal{G}_3(:,k_3,j_3,0)^T$ is orthogonal. If $i_3 = j_3$,

$$\mathbf{W}(p,q) = \begin{cases} \sum_{k_3} \mathcal{G}_3(p,k_3,i_3,0)^2 = 1 & \text{, if } p = q, \\ \sum_{k_3} \mathcal{G}_3(p,k_3,i_3,0)\mathcal{G}_3(q,k_3,i_3,0) = 0 & \text{, if } p \neq q. \end{cases} \tag{7}$$

Similarly, if $i_3 \neq j_3$, we can verify that $A(p,q)$ is always 0. Therefore,

$$\mathbf{A} = \sum_{k_3} \mathcal{G}_3(:,k_3,i_3,0)\mathcal{G}_3(:,k_3,j_3,0)^T = \begin{cases} \mathbf{I}, & \text{if } i_3 = j_3 \\ 0, & \text{if } i_3 \neq j_3 \end{cases} \tag{8}$$

If $i_3 \neq j_3$, then Equation (6) is 0. Otherwise, $i_3 = j_3$ and

$$\mathbf{W}^T\mathbf{W}(i,j) = \sum_{k_1,k_2} \mathcal{G}_1(0,k_1,i_1,:)\mathcal{G}_2(:,k_2,i_2,:)\mathcal{G}_2(:,k_2,j_2,:)^T\mathcal{G}_1(0,k_1,j_1,:)^T.$$

Similarly, we can show that lines 3-6 of Algorithm 2 yield

$$\sum_k \mathcal{G}_l(:,k,i_l,:)\mathcal{G}_l(:,k,j_l,:)^T = \begin{cases}\alpha\mathbf{I} & \text{, if } i_l = j_l \\ 0 & \text{, if } i_l \neq j_l\end{cases}, \text{ for } l = 1,2$$

This completes the proof that $\mathbf{W}$ is column orthogonal. □

The shape of the embedding matrix will satisfy $m \gg n$, ensuring that $R_{i-1}n_i < R_i m_i$ for $i = 2,3$. Therefore, it is always feasible to initialize the tensor cores with Rn_α orthogonal vectors. But it is not guaranteed that $R_2 n_3 \geq m_3$ in $\mathcal{G}_3$. For instance, suppose $m_3 = \lceil m^{1/3} \rceil = 126$ and $n = \lceil n^{1/3} \rceil = 5$ when $m = 10^8$ and $n = 128$. Then $Rn_3 \geq m_3$ is only satisfied when $R \leq 25$. Thus, we prefer to order the factors of the matrix dimensions m, n such that $m_3 = \max(\text{factor}(m))$ and $n_3 = \min(\text{factor}(n))$. Although this algorithm might be infeasible for large TT-ranks, we show in Section 6 that small TT-ranks are sufficient for many real-world graphs.

4.2 Homophily-informed TT Alignment

The embedding similarity between neighboring nodes can be inherited from TT format given an appropriate node ordering. Let i be the index of a node. Its embedding vector is constructed from

$$W[i,:] = \mathcal{G}_1(0,i_1,:,:)\mathcal{G}_2(:,i_2,:,:)\mathcal{G}_3(:,i_3,:,0),$$

where

$$i_1 = i/(m_2 * m_3), i_2 = (i/m_3) \bmod m_2, \text{ and}$$
$$i_3 = i \bmod m_3.$$

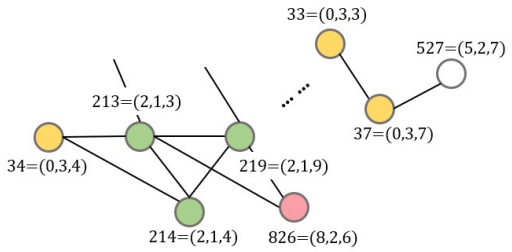

Figure 3: 3D coordinate mapping of an example graph with 1,000 nodes, assuming $m_1 = m_2 = m_3 = 10$. Nodes of the same color share the same $\mathcal{G}_1$ and $\mathcal{G}_2$ subtensors during embedding construction.

To see how weight sharing among nodes is determined by node ids, consider the illustration in Figure 3. If the nodes are randomly ordered, constructing an embedding vector is equivalent to drawing one slice $\mathcal{G}_l(:,i_l,:,:)$ uniformly at random for $l = 1,2,3$ and composing the weights. Consequently, distant vertices may share weights. Instead, ordering the nodes based on graph topology would better promote weight sharing among topologically neighboring nodes and thereby produce homophilic representations.

Formally, for any node index $j \in [i - i_3, i + m_3 - i_3]$, the row $\mathbf{W}[j,:]$ is constructed using the shared weights $\mathcal{G}_1(0,i_1,:,:)\mathcal{G}_2(:,i_2,:,:) \in \mathbb{R}^{m_1 m_2 \times r}$. The two rows $\mathbf{W}[i,:]$ and $\mathbf{W}[j,:]$ are formed by the same linear transformation of different set of vectors $\mathcal{G}_3(:,i_3,:,0)$ and $\mathcal{G}_3(:,j_3,:,0)$ respectively. Similarly, all the nodes with indices in range $[i - i_3 - i_2 m_3, i + m_3 - i_3 + (m_2 - i_2)m_3]$ share at least one factor matrix $\mathcal{G}_1(0,i_1,:,:)$ in embedding vector construction. The graph's topology influences the weight sharing and embedding vectors can be more similar when the node indices are closer. Hence, we need to preprocess the graph in a way that the topological closeness between nodes is proportional to the difference of their indices.

We perform hierarchical graph partitioning to discover clusters and reorder the nodes accordingly. We denote the original graph as level-0; to form the level $i+1$ partitioning, we recursively apply a p_i−way graph partitioning on each of the communities at level i. In this way, there will be p_1 partitions at level 1, $p_1 p_2$ partitions at level 2, and so on so forth. Let L be the total number of partition levels and $0 \leq p \leq p_1 p_2 \ldots p_L$ be a partition id at the L-th level partition. We reassign the indices to nodes in the partition from range $[np/(p_1 p_2 \ldots p_L), n(p+1)/(p_1 p_2 \ldots p_L)]$, where n is the number of nodes in the graph. We will assess this approach in Section 6.2.

5 EXPERIMENTAL SETUP

5.1 Datasets

We use three node-property prediction datasets from Open Graph Benchmark [7], summarized in Table 2. We add reversed edges to the ogbn-arxiv and ogbn-papers100M datasets to make the associated graphs undirected.

We train all learnable embedding models using one-hot encoding and without the original node features provided by OGB. We also train each model using the original node features given by OGB as a baseline to understand the quality of our design. The embedding dimensions of our embeddings are set to be the same as the datasets' original node features.

Table 2: Dataset statistics. We add reversed edges to ogbn-arxiv and ogbn-papers100M to make the graph undirected.

Dataset	#Nodes	#Edges	Emb. Dim.
ogbn-arxiv	169,343	2,332,486	128
ogbn-products	2,449,029	61,859,140	100
ogbn-papers100M	111,059,956	3,231,371,744	128

5.2 Hardware Platforms

The machine configurations we used are summarized in Table 3. We train all the models using the AWS EC2 g4dn.16xlarge instance and evaluate the training efficiency on EC2 multi-GPU instances p3.16xlarge and g4dn.metal. The r5dn.24xlarge instance is only used for preprocessing ogbn-papers100M dataset.

5.3 GNN Models and Baseline

For each dataset, we choose two different GNN models implemented with Deep Graph Library (DGL) [22]. For ogbn-arxiv, we use GCN [12] and GAT [20] models. For the two larger datasets, we

Table 3: Hardware platform details.

EC2 Type	Hardware specification
r5dn.24xlarge	2×24 cores, 700 GB RAM, 100 Gbit/s network
p3.16xlarge	2×16 cores, 500 GB RAM, 8 V100 GPUs
g4dn.metal	2×24 cores, 384 GB RAM, 8 T4 GPUs
g4dn.16xlarge	2×16 cores, 256 GB RAM, 1 T4 GPUs

use GraphSage [5] and GAT. The hyperparameters for each model are set to those tuned by the DGL team.

We consider the full embedding table models (Full-Emb) and position-based hash embedding (Pos-Emb) [9] as our baselines. All models are trained using one-hot encoding and no additional node features for each node, so that the model learns a unique embedding vector per node as the input to the GNNs.

6 RESULTS

We evaluate the performance of our proposed algorithms, varying the weight initialization, hierarchical graph partition, and training cost. We use *TT-Emb(r)* to refer to the GNN model with a trainable tensorized embedding with TT rank r. We compare our models to three other baseline options:

- *Orig-Feat*: Use the original features given by the OGB dataset as the non-trainable node feature. The features are preprocessed text associated with the nodes, such as paper abstract and product description.
- *Full-Emb*: Train a full embedding table only with one-hot encoding for each node.
- *Pos-Emb*: A positional-based hash embedding trained with one-hot encoding for each node.

Table 4: Performance summary of smaller graphs.

	ogbn-arxiv				ogbn-products			
Method	GCN	GAT	CR*	Time+	Graphsage	GAT	CR*	Time+
Full-Emb	0.671	0.677	1×	0.49	0.733	0.755	1×	33.4
Orig-Feats	0.724	0.736	1×	0.41	0.782	0.786	1×	31.5
Pos-Emb	0.674	0.676	12×	-	0.758	0.767	34×	-
TT-Emb	**0.681**	**0.682**	1,117×	0.97	**0.761**	**0.784**	415×	75.8

∗ Compression ratio of embedding layer compared to FullEmb and Orig-Feats.
+ Embedding layer training time (sec) per epoch for ogbn-arxiv with GCN and ogbn-products with GraphSage on g4dn.16xlarge instance.

Table 5: Accuracy, training time per batch, and compression ratio (CR) of ogbn-papers100M using different TT ranks.

	Graphsage		GAT		CR
	Accuracy	Time	Accuracy	Time	
FullEmb	0.595	10.34	N/A*	7.16	-
TT-Emb(8)	0.612	0.39	0.632	0.37	81,362
TT-Emb(16)	0.618	0.45	0.631	0.45	23,479
TT-Emb(32)	0.620	0.51	0.633	0.68	6,360
TT-Emb(64)	0.625	1.74	0.634	1.67	1,659

∗ Out of memory on the experiment platform.
+ The graph is preprocessed with 6,400 partitions. TT-emb(64) is initialized using Algorithm 1, and others using Algorithm 2.

Table 4 summarizes the accuracy, compression ratio, and training time of the baseline models and our optimal configuration for the two smaller datasets. Table 5 shows the performance on ogbn-papers100M with detailed configuration of TT-emb. For each dataset, we need to choose the TT rank, initialization algorithm, and the graph partitioning granularity. Our design is able to achieve the best accuracy among all models trained with one-hot encoding with order of magnitude fewer parameters. In particular, on ogbn-products, the quality of our compact embedding is comparable to the original node feature.

We study the accuracy of tensorized models initialized from different algorithms in Section 6.1. Then in Section 6.2, we analyze how graph partitioning affects the performance of the model. We discuss the compression and scalability of our model in Section 6.3.

6.1 Weight Initialization

We assess the impact of TT ranks and initialization algorithms from Section 4.1 on the model accuracy. Taking ogbn-products as an example, Table 6 summarizes the accuracy of models trained while varying the three most important hyperparameters: TT rank, initialization algorithm, and graph partitioning granularity. We focus on TT ranks and weight initialization in this section and leave graph partitioning to Section 6.2.

The three horizontal groups show the model accuracy as TT rank ranges from 8 to 32. Regardless of the initialization algorithm, Table 6 shows that, for GraphSage and GAT, larger TT ranks generally yield higher model accuracies. Increasing TT rank significantly improves model performance, especially with a coarse partitioning.

The three rows under each *TT-Emb(r)* group in Table 6 show that the accuracy of the models differ only by initialization algorithm. We refer to Algorithm 1 as decomp-ortho and Algorithm 2 as ortho-core in the table. Comparing to initializing TT-Emb from Gaussian distribution, the two orthogonal initialization algorithms improve the test accuracy in a similar way across all configurations and models. However, the decomposed orthogonal initialization (Algorithm 1) comes with a much higher computational cost. It takes 31 min for ogbn-papers100M on a r5dn.24xlarge instance, where 25 % of the time is spent on generating a random orthogonal matrix of size $110M \times 128$ using Scipy's QR decomposition [21]. By contrast, Algorithm 2 only takes a few seconds to generate orthogonal TT-cores using the Gram-Schmidt method for a graph of the same size.

In summary, for smaller TT ranks, the orthogonal TT-core initialization provides higher efficiency; for large TT ranks, the TTM decomposition of orthogonal matrices yields better scalability.

6.2 Graph Partitioning

To partition the graph, we use METIS, an algorithm based on the multilevel recursive-bisection and multilevel k-way partitioning scheme [11]. We reorder the nodes in the graph based on the partition results such that nodes in the same partition are indexed continuously (Section 4.2). This reordering encourages neighboring nodes to be constructed with shared parameters in their TT cores, thus inheriting a common embedding representation in each level of the partition.

Table 6: Accuracy of ogbn-products for different TT ranks, initialization algorithms, and graph partitioning.

		Graphsage							GAT						
# of partitions		0	4	16	256	800	1,600	3,200	0	4	16	256	800	1,600	3,200
TT-Emb(8)	decomp-ortho	0.6077	0.7188	0.7382	0.7352	0.7475	0.7508	0.7539	0.7386	0.7416	0.7393	0.7434	0.7505	0.7666	0.7718
	ortho-core	0.6321	0.6941	0.7333	0.7426	0.7517	0.7435	**0.7599**	0.7426	0.7541	0.7515	0.754	**0.7746**	0.7642	0.7689
	Gaussian	0.6189	0.6859	0.7230	0.7282	0.7188	0.7187	0.7190	0.6951	0.7147	0.7484	0.7451	0.7641	0.7554	0.7613
TT-Emb(16)	decomp-ortho	0.6688	0.7314	0.7489	0.7368	0.7484	**0.7606**	0.7556	0.7544	0.7440	0.7508	0.7512	**0.7812**	0.7726	0.7746
	ortho-core	0.6773	0.7039	0.7362	0.7379	0.7416	0.7434	0.7580	0.7464	0.7610	0.7603	0.7719	0.7803	0.7638	0.7719
	Gaussian	0.6716	0.6958	0.7305	0.7333	0.7385	0.7275	0.7487	0.7100	0.7559	0.7247	0.7531	0.7618	0.7637	0.7583
TT-Emb(32)	decomp-ortho	0.6701	0.7292	0.7414	0.7430	0.7378	**0.7613**	0.7566	0.7285	0.7671	0.7611	0.7747	0.7835	0.7756	**0.7843**
	ortho-core*	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	Gaussian	0.6712	0.7123	0.7396	0.7388	0.7485	0.7422	0.7455	0.7464	0.7607	0.7157	0.7719	0.7672	0.7521	0.7636

∗ The orthogonal initialization applies when factors of embedding dimension is sufficiently larger than TT rank. See Section 4.1 for more details.

To avoid correlation between node ids and topology from the default node ordering, we randomly shuffle the node orders before training or performing other techniques. Doing so allows us to study the effects of graph partitioning. Again, we refer to Table 6 when discussing the accuracy of GraphSage and GAT trained with different TT ranks and numbers of partitions. Using even a small number of partitions helps to improve TT-emb's accuracy significantly. Partitioning the graph into four clusters improves the accuracy of TT-emb Graphsage by up to 6.7 % and for GAT by up to 3.8 %.

However, since the nodes within a partition are randomly ordered, the locality of node indexing is not guaranteed. With a coarse partition, TT-emb does not capture the shared property, which results in lower accuracy. As we increase the number of partitions to 3,200, the accuracy of Graphsage improves by an average of 10.5 % and of GAT by an average of 3.2 % with orthogonal initialization. On the other hand, an overly fine-grained graph partition is unnecessary as TT-emb is robust to the bottom-level node permutation within the METIS partition hierarchy.

For a fixed TT rank, the test accuracy generally improves as the number of partitions increases. The minimal number of partitions to reach the best for GraphSage and GAT are 3,200 and 800, respectively. With the optimal partition configuration, our design outperforms the other two embedding models using any TT rank comparing the the baselines in Table 4.

Moreover, We observe that graph partition is particularly effective in homophily graphs with small TT ranks as shown in Table 5. In ogbn-papers100M, compared to using default node ordering, the test accuracy of TT-emb(8) improves by 6 % with 6,400 partitions, while that improvement decreases to 3 % with TT rank 128.

To further understand the robustness of our model against node permutation, we implemented a recursive graph partitioning scheme based on METIS. We recursively partition each subgraph returned by METIS into p_i clusters at the i-th level. This scheme simulates the hierarchical partitioning algorithm of METIS while allowing us to precisely control the branching at each level.

Figure 4 compares the accuracy of models trained with the same recursive partition strategy: first partition the graph into 125 parts, then divide each part into 140 partitions. These two numbers are chosen from the first two TT-core dimensions $m_1 = 125$, $m_2 = 140$, and $m_3 = 150$. We permute the ordering of the partition IDs in the following way:

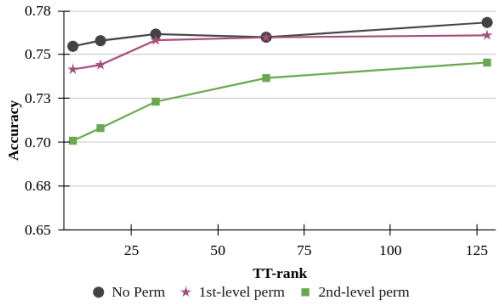

Figure 4: Test accuracy of TT-Emb Graphsage models trained with different graph partition strategies on ogbn-products.

- **No perm**: keep the original partition returned by METIS
- **1st-level perm**: randomly shuffle the ordering of the first level m_1 partitions, each roughly containing $m_2 m_3$ nodes
- **2nd-level perm**: randomly shuffle the ordering of the second level $m_1 m_2$ partitions, each partition contains m_3 nodes on average

The accuracy is modestly affected by a 1st-level permutation, while the accuracy decreases by at least 3 % due to 2nd-level permutation. By Equation (5), after the METIS reordering, each cluster of $m_2 m_3$ nodes are constructed using the same subtensor in $\mathcal{G}_1$, and every m_3 consecutive nodes share the weights from $\mathcal{G}_2$. The 1st-level permutation only interchanges ordering of subtensors $\mathcal{G}_1$, which ensures the graph homophily is preserved. However, the 2nd-level permutation isolates each group of m_2 clusters belonging to a same level-1 partition. Furthermore, every m_3 nodes are constructed with a potentially different subtensor in $\mathcal{G}_1$. Hence, we observe significant accuracy loss due to lack of homophily.

6.3 Model Compression and Training Time

Tensor-train compresses the embedding tables significantly. Table 7 compares the memory requirement between the full embedding table and TT-embedding across different TT ranks for the three datasets. On ogbn-arxiv, to achieve the same level of accuracy

Table 7: The original dimensions of embedding tables in OGB datasets and their respective TT decomposition parameters.

Dataset	Emb. Dimensions		TT-Core Shapes			# of TT Parameters			Memory Reduction		
	# Nodes	Dim.	$\mathcal{G}_1$	$\mathcal{G}_2$	$\mathcal{G}_3$	$R = 16$	$R = 32$	$R = 64$	$R = 16$	$R = 32$	$R = 64$
ogbn-arxiv	169,363	128	$(1, 55, 8, R)$	$(R, 55, 4, R)$	$(R, 56, 4, 1)$	66,944	246,528	943,616	323	88	23
ogbn-products	2,449,029	100	$(1, 125, 4, R)$	$(R, 140, 5, R)$	$(R, 140, 5, 1)$	198,400	755,200	2,944,000	1,580	415	106
ogbn-papers100M	111,059,956	128	$(1, 480, 8, R)$	$(R, 500, 4, R)$	$(R, 500, 4, 1)$	605,440	2,234,880	8,565,760	23,479	6,360	1,659

as Full-Emb, we use a TT rank of at least 64 on GCN and at least 8 on GAT. This configuration results in a maximum compression ratio of 22.9× on GCN and 1,117× on GAT. The memory savings grows as the graph size increases. As shown in Table 6, the minimum TT rank required to outperform the Full-Emb baseline is 8 for both Graphsage and GAT, which results in a model 5,762× smaller than Full-Emb. For the largest dataset, ogbn-papers100M, our model achieves a maximum compression ratio of 81,362× over Full-Emb.

The magnitude of achievable compression can reduce or eliminate the need for CPU-host-to-GPU communication for embeddings, thereby enabling significantly higher GPU utilization. For the 110 million node dataset, the embeddings need just 0.1 GB, allowing the storage and training of the full model on a single GPU. Moreover, in case of multi-GPU training, TT-emb makes it possible to store a copy of embedding table on each GPU, thus reducing communication during training.

Training times are compared Table 8. It shows the training time of the Full-Emb model and TT-Emb compressed models for the three datasets using a single GPU. For the two smaller datasets, the full embedding table can be store on GPU. By using TT compression, each embedding vector is constructed through two back-to-back GEMMs (general dense matrix-matrix multiplications) instead of pure memory access for lookups. Therefore, on ogbn-arxiv and ogbn-products, we observe up to 33.6× slow down with a large TT rank. However, to achieve the same level of accuracy as the baseline model, one only needs to use a TT rank as small as 8 with sufficiently large partition numbers and our proposed initialization. Doing so results in 1.28× and 2.3× training slowdowns for ogbn-arxiv and ogbn-products, respectively.

For ogbn-papers100M, where the full embedding table does not fit in GPU memory, TT-emb outperforms the baseline with a minimum of 19.3× speedup in both models using TT rank 8. Figure 5 breaks down the time spent in each computation phase for ogbn-papers100M training, including forward propagation (fwd), backward propagation (bwd), and parameter update (upd). The full embedding model requires storing and updating embedding parameters on the CPU, which consist of 90 % of the training time and thus limits the potential of such models on large graphs. On the other hand, our model fits entirely on the GPU and thus achieves significant speedup in updating model parameters but spend more time in the embedding construction and gradient computation.

We also train the models on multi-GPUs to study the scaling potential of our method for even larger graphs. Figure 6 shows the training time per epoch of ogbn-papers100M using all TT-emb models and different number of GPUs. The green lines represent for the results collected using a g4dn.metal instance, and the red ones

Table 8: Single GPU training time per epoch on ogbn-arxiv, and per batch on ogbn-products and ogbn-papers100M, benchmarked on p3.16xlarge instance.

	ogbn-arxiv		**ogbn-products**		**ogbn-papers100M**	
	GCN	GAT	GraphSage	GAT	GraphSage	GAT
Full Emb	0.14	0.63	0.12	0.18	10.34	7.16
TT-emb(8)	0.18	0.66	0.28	0.28	0.39	0.37
TT-emb(16)	0.18	0.67	0.33	0.31	0.45	0.45
TT-emb(32)	0.20	0.69	0.49	0.43	0.51	0.68
TT-emb(64)	0.25	0.75	1.29	0.99	1.74	1.67
TT-emb(128)	0.49	0.99	4.65	3.37	5.70	5.37

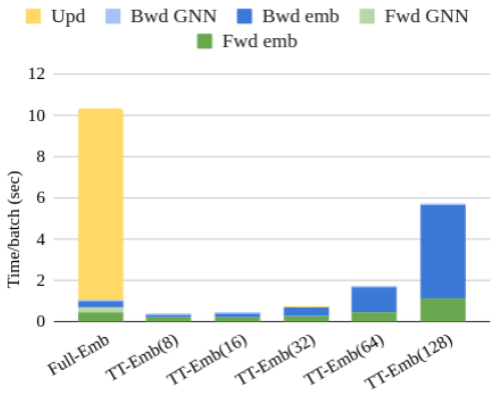

Figure 5: Time spent in each kernel for ogbn-papers100M training. Parameter update of Full-Emb is performed on CPU, whereas all TT-Emb operations are on GPU.

are from a p3.16xlarge instance. Within each color group, the darker shades correspond to model configuration with larger TT ranks. The small TT rank models benefit less from multi-GPU training. Using 8 GPUs, the TT-Emb(8) Graphsage models are less than 2× faster than single-GPU training on both instances. Doubling the TT rank increases the computation for TT-emb by 4×. The embedding computation consist of 77.2 % for TT rank 8, compared to 95.8 % for TT rank 128. Therefore, TT-emb dominates training time with large TT ranks, and multi-GPU distribution achieves a nearly linear speedup for those models.

7 CONCLUSION

Our most significant results lend strong *quantitative* evidence to an intuitive idea: one should incorporate knowledge about the

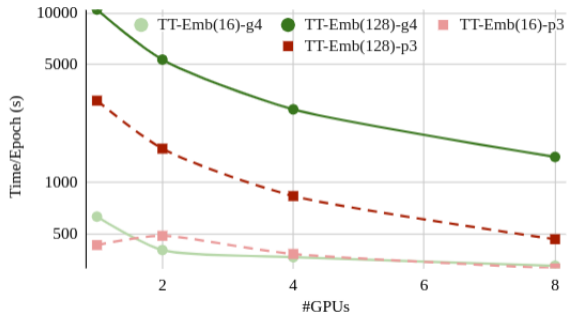

Figure 6: Multi-GPU training time of TT-Emb models for ogbn-papers100M using GraphSage on g4dn.metal instance.

global structure of the input graph *explicitly* into the training process. For example, our proposed algorithm directly uses hierarchical graph structure, discovered via hierarchical graph partitioning, to impose additional structure on the compact parameterization induced by our use of TT decomposition. Intuitively, this approach encourages homophilic node embeddings; empirically, we observed instances of improvements in model accuracy by nearly 5 % compared to a state-of-the-art baseline. It is even possible to learn embeddings *without* using node features while achieving performance competitive with methods that do use such features, which may seem surprising. Given that many real-world graphs lack such features, this result is especially encouraging.

However, partitioning alone is not enough to produce a good overall method. A second critical element of our work is an intentional choice of weight initialization where we orthogonalize the TT-Emb weights. This initialization strategy significantly improves the model accuracy and convergence. Our experiments show that by combining the initialization algorithms and graph partitioning, our overall design outperforms the state-of-the-art trainable embedding methods on all node property prediction tasks, with order of magnitude fewer parameters.

Overall, the combination of TT, graph partitioning, and judicious weight-initialization reduces the size of a node embedding by over 1,659× on an industrial graph with 110M nodes, while maintaining or improving accuracy and speeding up training by as much as 5×. Nevertheless, several important questions remain. For instance, precisely why orthogonalized embedding for GNNs improves classification accuracy is not fully understood. And while our methods are effective on homophilic graphs, the performance on heterophilic and heterogenous graphs remains open.

REFERENCES

[1] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 1247–1250.

[2] Nadav Cohen, Or Sharir, and Amnon Shashua. 2016. On the expressive power of deep learning: A tensor analysis. In *Conference on learning theory*. PMLR, 698–728.

[3] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry Vetrov. 2016. Ultimate tensorization: compressing convolutional and fc layers alike. *arXiv preprint arXiv:1611.03214* (2016).

[4] Benjamin Ghaemmaghami, Zihao Deng, Benjamin Cho, Leo Orshansky, Ashish Kumar Singh, Mattan Erez, and Michael Orshansky. 2020. Training with multi-layer embeddings for model reduction. *arXiv preprint arXiv:2006.05623* (2020).

[5] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[6] Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan Oseledets. 2019. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787* (2019).

[7] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[8] Wei Hu, Lechao Xiao, and Jeffrey Pennington. 2020. Provable benefit of orthogonal initialization in optimizing deep linear networks. *arXiv preprint arXiv:2001.05992* (2020).

[9] Maria Kalantzi and George Karypis. 2021. Position-based Hash Embeddings For Scaling Graph Neural Networks. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 779–789.

[10] Wang-Cheng Kang, Derek Zhiyuan Cheng, Tiansheng Yao, Xinyang Yi, Ting Chen, Lichan Hong, and Ed H Chi. 2020. Deep hash embedding for large-vocab categorical feature representations. *arXiv e-prints* (2020), arXiv–2010.

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

[12] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[13] Christos Louizos, Max Welling, and Diederik P Kingma. 2017. Learning sparse neural networks through L_0 regularization. *arXiv preprint arXiv:1712.01312* (2017).

[14] Miller McPherson, Lynn Smith-Lovin, and James M. Cook. 2001. Birds of a feather: homophily in social networks. *Annual Review of Sociology* 27 (August 2001), 415–444. https://doi.org/10.1146/annurev.soc.27.1.415

[15] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440* (2016).

[16] Ivan V Oseledets. 2011. Tensor-train decomposition. *SIAM Journal on Scientific Computing* 33, 5 (2011), 2295–2317.

[17] Joan Serrà and Alexandros Karatzoglou. 2017. Getting deep recommenders fit: Bloom embeddings for sparse binary input/output networks. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*. 279–287.

[18] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. 2020. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 165–175.

[19] Dan Tito Svenstrup, Jonas Hansen, and Ole Winther. 2017. Hash embeddings for efficient word representations. *Advances in neural information processing systems* 30 (2017).

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

[21] Pauli Virtanen, Ralf Gommers, and et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[22] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. (2019).

[23] Wenqi Wang, Yifan Sun, Brian Eriksson, Wenlin Wang, and Vaneet Aggarwal. 2018. Wide compression: Tensor ring nets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 9329–9338.

[24] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*. 1113–1120.

[25] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. 2021. TT-rec: Tensor train compression for deep learning recommendation models. *Proceedings of Machine Learning and Systems* 3 (2021), 448–462.

[26] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems* 2 (2020), 412–428.

[27] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.

[28] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878* (2017).