KAN We Tensorize GraphSAGE

Richard Gao
UCSB
rwgao@ucsb.edu

Charlie Jyu
UCSB
cjyu@ucsb.edu

Sanjukta Krishnagopal
UCSB
sanjukta@ucsb.edu

Abstract

A common task in graph machine learning is to find low-dimensional embeddings of nodes. After embeddings are generated, they can be applied to downstream tasks such as recommendation systems and protein classification. These nodes can be trained through unsupervised, semi-supervised, and supervised objectives. In this project, we build upon previous work in two directions: introducing a new aggregator and parameter reduction.

1 Introduction

Embedding graph nodes in low-dimensional vector space is a widely explored subject in graph machine learning. Node embeddings seek to encode node-level features, neighborhood information, and global information that can be later used for tasks such as node classification and clustering. GraphSAGE [4] opts to learn aggregator functions to generate node embeddings. These learned aggregator functions can then be applied to completely unseen nodes during inference on large graphs.

For this project, we modify two aspects of GraphSAGE:

1. GraphSAGE incorporates node features from the dataset into calculating embeddings. If this node input feature dimension is high, the model size can become very large. To encourage training on larger feature spaces, we parameterize the linear layers of GraphSAGE using tensor-train (TT) decomposition [10, 11], greatly reducing the number of trainable parameters.
2. GraphSAGE introduces multiple trainable aggregators. To this end, we explore a new aggregator parameterized by a Kolmogorov-Arnold Network (KAN) [9]. Instead of trainable weights, KANs offer trainable activations parameterized by splines. Previous works [6, 18] have explored the efficacy of KAN layers in Graph Convolutional Networks (GCN) [7] and Graph Neural Networks (GNN) [15]. We seek to extend this analysis to GraphSAGE.

2 Related Works

KAN in graph models KANs have been sparsely explored in the space of graph machine learning. Kiamari, et al. modify GCNs to create two new KAN models [6]. The difference between the two lies in placing a KAN layer either before or after the aggregation step. In their first proposed architecture, a KAN layer is applied post-aggregation; in their second, a KAN layer is applied to the embeddings before aggregation. Zhang and Zhang divide the message-passing framework for node representation into an aggregation and extraction step [18]. The extraction step is when a node update function is used to update the next layer's node features and is often parameterized by an MLP. Both papers demonstrate improved performance over traditional methods. However, Zhang and Zhang point out a dramatic increase in training time from using KAN. Our work operates in a similar setting; however, we explore results on unsupervised and inductive node classification problems.

TT in graph models Previous work that applies TT-decomposition to graph machine learning is also sparse. Both Yin, et al. and Qu, et al. propose using TT-decomposition on embedding tables for memory reduction [17, 13, 5]. Our work explores directly applying TT-decomposition to the trainable weights of GraphSAGE.

3 Background

3.1 GraphSAGE

GraphSAGE is an inductive learning framework that samples neighbors and aggregates them from a local neighborhood up to depth K [4]. At each depth $k \in \{1, \ldots, K\}$, for every node, GraphSAGE samples a fixed number of neighbors S_k, and retrieves their hidden representations generated from the previous depth of $k-1$. These hidden representations are then aggregated using a learnable aggregator function that produces a neighborhood hidden representation. This representation is then combined with the original node's own hidden representation and passed through a learnable weight and nonlinearity. The algorithm flow is detailed below 1.

Algorithm 1 GraphSAGE embedding generation

```
Input: Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth $K$; weight matrices $\mathbf{W}^k, \forall k \in \{1, \ldots, K\}$; non-linearity $\sigma$; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, \ldots, K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{|\mathcal{V}|}$
Output: Vector representations $\mathbf{z}_v$, for all $v \in \mathcal{V}$
1: $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$
2: for $k = 1, \ldots, K$ do
3:     for $v \in \mathcal{V}$ do
4:         $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$
5:         $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
6:     end for
7:     $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
8: end for
9: $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$
```

The original GraphSAGE paper provides multiple aggregator architectures. Their improvements over each other in GraphSAGE's results, however, are marginal. Thus, we focus on modifying the pooling aggregator (1) for our project:

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}} \mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\}), \tag{1}$$

3.2 Tensor-Train (TT) Decomposition

3.2.1 TT-Format

For notation, we will denote a tensor $\boldsymbol{\mathcal{A}}$ by bold calligraphic uppercase letters and a matrix $\mathbf{A}$ by just boldface uppercase letters. A tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ is said to be of order d, and $I_1, I_2, \ldots, I_d$ are called its modes. We can index $\boldsymbol{\mathcal{A}}$ using an index $\boldsymbol{i} \in I_1 \times I_2 \times \cdots \times I_d$ such that $\boldsymbol{\mathcal{A}}(\boldsymbol{i}) = \boldsymbol{\mathcal{A}}(i_1, i_2, \ldots, i_d)$.

A d-way tensor $\boldsymbol{\mathcal{A}}$ is in TT-format if for all modes $k \in \{1, 2, \ldots, d\}$, and for all indices of the k-th mode $j_k \in \{1, 2, \ldots, I_k\}$, there exists a matrix $\mathbf{G}_k[j_k]$ such that we can obtain the entries of $\boldsymbol{\mathcal{A}}$ like so [11]:

$$\boldsymbol{\mathcal{A}}(j_1, j_2, \ldots, j_d) = \prod_{k=1}^{d} \mathbf{G}_k[j_k] \tag{2}$$

Together, the sets of matrices $\{\mathbf{G}_k[j_k]\}_{j_k=1}^{I_k}$ for all modes k are called cores [1]. The sizes of each matrix in a core are determined by a sequence $\{r_k\}_{k=0}^{d}$ referred to as the TT-ranks. Specifically, the matrix $\mathbf{G}_k[j_k]$ has size $r_{k-1} \times r_k$. Furthermore, $r_0 = r_d = 1$ in order to keep our matrix product (2) corresponding to an entry in $\boldsymbol{\mathcal{A}}$ a scalar.

Thus, storing the full tensor $\boldsymbol{\mathcal{A}}$ requires $\prod_{k=1}^{d} I_k$ elements, while storing it in TT-format requires only $\sum_{k=1}^{d} I_k r_{k-1} r_k$ elements. If we choose small TT-ranks, we can greatly reduce the amount of elements needed to represent $\boldsymbol{\mathcal{A}}$ and perform operations using tensorized representations.

3.2.2 Tensorizing Linear Layers

To take advantage of the compression introduced by using TT-format, it can be applied to a linear layer of the form $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{W} \in \mathbb{R}^{M \times N}$, $\mathbf{b} \in \mathbb{R}^{M}$.

Novikov, et al. start by tensorizing $\mathbf{b} \in \mathbb{R}^{M}$ into $\mathcal{B} \in \mathbb{R}^{m_1 \times m_2 \times \cdots \times m_d}$: Assume M has d factors such that $M = \prod_{k=1}^{d} m_k$. Define a bijection $\boldsymbol{\mu}$ that maps a coordinate $l \in \{1, 2, \ldots M\}$ of $\mathbf{b}$ to a vector index $\boldsymbol{\mu}(l) = (\mu_1(l), \mu_2(l), \ldots, \mu_d(l))$ of $\boldsymbol{\mathcal{B}}$. For every mode k, $\mu_k(l) \in \{1, 2, \ldots, m_k\}$. Thus, the entries of b are defined as $\boldsymbol{\mathcal{B}}(\boldsymbol{\mu}(l)) = \mathbf{b}_l$, called the TT-vector.

Novikov, et al. tensorize $\mathbf{W} \in \mathbb{R}^{M \times N}$ into $\boldsymbol{\mathcal{W}}$ in a similar fashion by factoring N into d factors, $\prod_{k=1}^{d} n_k$, like we did M. Define a similar bijection $\boldsymbol{\nu}$ for N that maps $t \in \{1, 2, \ldots, N\}$ to a vector index $\boldsymbol{\nu}(t) = (\nu_1(t), \nu_2(t), \ldots, \nu_d(t))$. Construct an order d tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{m_1 n_1 \times \cdots \times m_d n_d}$. Indexing $\mathbf{W}$ with (t, l) is equivalent to indexing $\boldsymbol{\mathcal{W}}$ with $(\boldsymbol{\nu}(t), \boldsymbol{\mu}(l))$, with the advantage that we can represent $\boldsymbol{\mathcal{W}}$ in TT-format:

$$\mathbf{W}_{t,l} = \boldsymbol{\mathcal{W}}((\nu_1(t), \mu_1(l)), \ldots, (\nu_d(t), \mu_d(l)) = \prod_{k=1}^{d} \mathbf{G}_k[(\nu_k(t), \mu_k(l)]. \tag{3}$$

Thus, Novikov, et al. show that computing any linear layer using TT-format is equivalent to the following:

$$\boldsymbol{\mathcal{Y}}(i_1, \ldots, i_d) = \sum_{j_1, \ldots, j_d} \prod_{k=1}^{d} \mathbf{G}_k[i_k, j_k] \boldsymbol{\mathcal{X}}(j_1, \ldots, j_d) + \boldsymbol{\mathcal{B}}(i_1, \ldots, i_d), \tag{4}$$

where $\boldsymbol{\mathcal{X}}$ is obtained by reshaping $\mathbf{x}$. If $r = \max_k r_k$ is the maximal TT-rank, Novikov, et al. show that the computational complexity of (4) is $\mathcal{O}(dr^2 \max\{M, N\})$ [10].

3.3 Kolmogorov-Arnold Networks

3.3.1 Kolmogorov-Arnold Representation Theorem

The Kolmogorov-Arnold Representation Theorem establishes that for any multivariate function f defined on a bounded domain, f can be decomposed into the sum of continuous univariate functions. Formally, a function $f : [0, 1]^n \to \mathbb{R}$ can be written as,

$$f(\mathbf{x}) = f(x_1, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right), \tag{5}$$

where for all p and q, $\phi_{q,p} : [0, 1] \to \mathbb{R}$ and $\Phi_q : \mathbb{R} \to \mathbb{R}$. This representation establishes that any multivariate function only requires composition through summation of single variable functions.

Kolmogorov-Arnold Representation Theorem is not a well-explored topic in machine learning because while $\phi_{q,p}$ are continuous, they are generally not smooth [9]. Liu et al. [9] propose a new neural network architecture inspired by (5), showing that the representation theorem is an instantiation of their network with a specific width and depth.

[1] In other literature, cores are expressed as order-3 tensors $\boldsymbol{\mathcal{G}}_k$. We can obtain the same formulation by simply stacking the set of matrices $\{\mathbf{G}_k[j_k]\}_{j_k=1}^{I_k}$ to obtain $\mathcal{G}_k$, for some mode k.

3.3.2 Kolmogorov-Arnold Network (KAN) Architecture

In traditional deep learning, a L-layer multi-layer perception (MLP) is a composition of affine transformations with nonlinear activations:

$$\text{MLP}(\mathbf{x}) = (\mathbf{W}_{L-1} \circ \sigma \circ \mathbf{W}_{L-2} \circ \sigma \circ \cdots \circ \mathbf{W}_1 \circ \sigma \circ \mathbf{W}_0)\mathbf{x}. \tag{6}$$

The weight matrices $\{\mathbf{W}_l\}_{l=0}^{L-1}$ are then optimized over some loss function. Liu, et al. prototype a KAN architecture that fixes weights and instead learns activation functions. More formally, a depth L KAN with shape $[n_0, n_1, \ldots, n_L]$ has n_l nodes in its l-th layer. The l-th layer maps input shape n_l to output shape n_{l+1}, and is parameterized by a matrix of univariate functions:

$$\mathbf{\Phi}_l = \begin{pmatrix} \phi_{l,1,1} & \cdots & \phi_{l,1,n_l} \\ \vdots & \ddots & \vdots \\ \phi_{l,n_{l+1},1} & \cdots & \phi_{l,n_{l+1},n_l} \end{pmatrix}. \tag{7}$$

Where each function $\phi_{l,q,p}$ is parameterized by trainable B-splines. The post-activation $\mathbf{x}_{l+1}$ is then computed as

$$\mathbf{x}_{l+1} = \begin{pmatrix} \phi_{l,1,1}(\cdot) & \cdots & \phi_{l,1,n_l}(\cdot) \\ \vdots & \ddots & \vdots \\ \phi_{l,n_{l+1},1}(\cdot) & \cdots & \phi_{l,n_{l+1},n_l}(\cdot) \end{pmatrix} \mathbf{x}_l. \tag{8}$$

Thus, the entire forward pass of a L-layer KAN can be written as a composition similar to (6):

$$\text{KAN}(\mathbf{x}) = (\mathbf{\Phi}_{L-1} \circ \mathbf{\Phi}_{L-2} \circ \cdots \circ \mathbf{\Phi}_0)\mathbf{x}. \tag{9}$$

Using this formulation, Liu, et al. show that the Kolmogorov-Arnold Representation Theorem is equivalent to a 2-layer KAN with shape $[n, 2n+1, 1]$. Furthermore, Liu, et al. hypothesize that while the Kolmogorov-Arnold Representation often admits non-smooth functions, a deep KAN architecture may relax this behavior, encouraging smoother univariate functions.

4 Proposed Aggregator Architectures

In this section, we detail how we incorporate TT-layers and KANs into training GraphSAGE.

4.1 TT-Aggregator

The GraphSAGE paper highlights the importance of incorporating rich node features into training. However, a greater node feature dimension also leads to an explosion in trainable parameters. Let d be the dimensionality of each node's feature space. Assuming we use depth k max-pooling aggregators (1), GraphSAGE has approximately $3kd^2 + kd$ parameters. The dimensionality d can often be large and can become a limiting factor if we want to train a wider network. For example, if the nodes of a graph encode text embeddings, we may wish to train on a larger feature space. In situations like this, we can try tensorizing the weight matrices in GraphSAGE to reduce the number of parameters. Let $\text{TT}(\mathbf{W})$ denote the operation of tensorizing a weight matrix shown as shown in (3). We modify line 5 of Algorithm 1 by tensorizing $\mathbf{W}^k$ and reshaping the concatenation to the appropriate TT shapes:

$$\mathbf{h}_v^k \leftarrow \sigma\left(\text{TT}(\mathbf{W}^k) \cdot \text{RESHAPE}(\text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))\right). \tag{10}$$

We apply a similar modification to the max-pool aggregator (1):

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\text{TT}(\mathbf{W}_{\text{pool}}) \cdot \text{RESHAPE}(\mathbf{h}_{u_i}^k) + \text{TT}(\mathbf{b})\right), \forall u_i \in \mathcal{N}(v)\}). \tag{11}$$

Parameter Count To analyze how this changes the number of parameters, assume that we can factor d and $2d$ into $d = \prod_{c=1}^{C} n_c$ and $2d = \prod_{c=1}^{C} m_c$. Given prior determined TT-ranks $\{r_c\}_{c=0}^{C}$, the total number parameters in a tensorized depth k GraphSAGE model is approximately given by,

$$\#\text{params} = k \sum_{c=1}^{C} r_{c-1} r_c n_c (n_c + m_c). \tag{12}$$

It is difficult to control the factorizations of d, so most of the burden in parameter reduction falls upon the TT-ranks $\{r_c\}_{c=0}^{C}$ that we choose. We use automatic rank determination using the TT-SVD algorithm [11]. The core idea of TT-SVD is to reshape a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ into a matrix $A \in \mathbb{R}^{I_1 \times I_2 I_3 \cdots I_d}$, and apply singular-value decomposition (SVD) to obtain TT-core $\boldsymbol{\mathcal{G}}_1$. This process of iteratively unfolding each mode of $\boldsymbol{\mathcal{A}}$ and applying SVD is repeated until all TT-cores are obtained:

4.2 KAN-Aggregator

Since KANs directly learn non-linear activations as opposed to linear transformations composed with MLPs (6), we can directly replace a fully connected layer with a KAN. However, we found that directly initializing a KAN network did not work well and was difficult to train: given input and output dimensions d_{in} and d_{out}, a linear layer with a bias term yields $d_{in} \times d_{out} + d_{out}$ parameters. To parameterize a KAN layer, we require the input and output dimensions, as well as spline order k on G intervals. This yields $d_{in} \times d_{out} \times (G + k + 3) + d_{out}$ total parameters. We can see that even for very small values of G and k, a KAN layer with the same input and output dimensions as an MLP instantiates significantly more parameters.

Projection Layers To alleviate this issue, we propose downward and upward projection layers $\mathbf{W}_{in}$ and $\mathbf{W}_{out}$ to stitch higher-dimensional inputs and outputs with lower-dimensional KAN inputs and outputs. Reducing the parameters of KANs is an open problem, and Liu, et al. speculate that KANs require much fewer parameters than their MLP counterparts. Empirical works on KANs and graph neural networks suggest that feeding KAN a latent representation improves performance. Additionally, previous work on KANs with graph data [9] hypothesizes that LayerNorm [1] layers improve the stability of KAN training, so we include them in our work as well. We apply these changes to all weights + non-linearities in GraphSAGE. In line 5 of Algorithm 1:

$$\mathbf{h}_v^k \leftarrow (\mathbf{W}_{out}^k \circ \text{KAN}^k \circ \text{LN}^k \circ \mathbf{W}_{in}^k)(\text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)). \tag{13}$$

We define the KAN-aggregator in a similar fashion:

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{(\mathbf{W}_{out}^{\text{pool}} \circ \text{KAN}^{\text{pool}} \circ \text{LN}^{\text{pool}} \circ \mathbf{W}_{in}^{\text{pool}})(\mathbf{h}_{u_i}^k), \forall u_i \in \mathcal{N}(v)\}), \tag{14}$$

5 Training

We test GraphSAGE in a fully unsupervised setting and use the unsupervised loss function proposed in the original GraphSAGE paper:

$$J_G(\mathbf{z}_u) = -\log(\sigma(\mathbf{z}_u^\intercal \mathbf{z}_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \left[-\log(\sigma(\mathbf{z}_u^\intercal \mathbf{z}_{v_n})) \right]. \tag{15}$$

This loss function is applied to the embedding outputs $\mathbf{z}_u$ for every node u. The loss function encourages similar representations between nearby nodes through a nearby node v that co-occurs with u on a fixed-length random walk. To guide distant nodes to have different representations, the loss functions samples Q negative samples from a negative sampling distribution P_n.

6 Experiments

6.1 Real World Data

We conduct experiments on three datasets: protein-protein interaction (PPI) [19] and Cora [3]. We summarize the attributes of each dataset in Table 1. Note that the feature space of Cora is prime, so

Table 1: Datasets

Name	Features	Nodes	Edges	Classes
PPI (20 graphs)	50	$\sim 2,245.3$	$\sim 61,318.4$	121
Cora	1,433 + 1	2,708	10,556	7

Table 2: F1 score

	PPI		Cora	
Name	Parameters	Unsup. F1	Parameters	Unsup. F1
MLP-Pool (baseline)	64866	0.4658	2989374	0.727
TT-Pool	7123	0.4583	1172346	0.709
KAN-Pool	80610	0.4620	2843418	0.764

we pad each node feature by one dimension, denoted as "+1" in Table 1. This is necessary because the dimensions of tensorized linear layers depend on the factorization of the original weight matrix's dimensions (3).

PPI The PPI dataset contains 24 protein-protein interaction graphs, where each graph is a distinct human tissue. We reserve 20 for training, two for validation, and two for testing.

Cora The Cora dataset is a bibliographic dataset composed of Machine Learning papers classified into seven classes. We split the data into training, validation, and test sets.

6.1.1 Experimental set-up

We try to adhere to as many of the original hyperparameter choices in GraphSAGE. We conduct all experiments with rectified linear units as non-linearity, $K = 2$ aggregators/search depth, and neighborhood sample sizes $S_1 = 25$ and $S_2 = 10$ for each search depth. We compare our proposed methods against a baseline MLP-based pooling aggregator 1 from the original GraphSAGE paper. For each model, we perform stochastic gradient descent on unsupervised loss (15) using the Adam optimizer over multiple epochs (10 for PPI, 50 for Cora). For each model, we report the number of trainable parameters as well as their unsupervised F1 score 2. The F1 scores are obtained by the logistic SGDClassifier from the scikit-learn [12]. This model is never fine-tuned on the validation and testing data. For each model, we perform hyperparameter tuning on the learning rate for Adam. The results for each model are detailed in Table 2.

6.2 Synthetic Data

We also conduct experiments on our models using synthetic graphs with more simple relationships to analyze the embeddings generated by each model. To generate the nodes, we pick random means and standard deviations for each class to create class instances. We then pass each normally distributed data point into a sigmoid transformation to generate nodes. To generate edges, we first pick two random nodes. The probability we place an edge between them depends on two factors: an affinity score and cosine similarity. The affinity score is a uniform random variable for any pair of classes, scaling the likelihood of an edge between nodes from those classes. For cosine similarity, we define the probability density P over $[-1, 1]$ to be $P(x) = \exp(c(x-1))$, where c is a normalizing constant to make P a valid density function. We outline our graph generation flow in Algorithm 2[2][3]. We aim to create a dataset with distinct classes and encapsulate both class-level interactions through affinity scores and node-level interactions through cosine-similarity.

6.2.1 Experimental set-up

We test our models on a synthetic graph with $C = 10$ classes, $N = 500$ nodes, $M = 3000$ edges, and dimensionality $d = 16$. Again, we adhere to the same hyperparameter choices in GraphSAGE

[2]For shorthand, denote [n] as the set $\{1, \ldots, n\} \subseteq \mathbb{N}$

[3]We denote σ as the sigmoid function, and σ_c as a standard deviation value

Algorithm 2 Generate Synthetic Data

```
Input: $C$ classes, $N$ nodes, $M$ edges
Output: Synthetic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$
 1: Sample means from multivariate normal $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_C \sim \mathcal{U}([0,1]^d)$
 2: Sample standard deviations $\sigma_1, \ldots, \sigma_C \sim \mathcal{U}[0,1]$
 3: for $c = 1, \ldots C$ do
 4:     $\mathcal{V}_c \leftarrow \{\sigma(\mathbf{v}_i) \mid \mathbf{v}_i \sim \mathcal{N}(\boldsymbol{\mu}_c, \text{diag}(\sigma_c), i \in [N/C]\}$        ▷ generate nodes for each class
 5: end for
 6: $\mathcal{V} \leftarrow \bigcup_c^C \mathcal{V}_c$
 7: Sample affinities $a_{1,1}, a_{1,2}, \ldots, a_{C,C} \sim \mathcal{U}[0,1]$
 8: $\mathcal{E} \leftarrow \emptyset$
 9: while $|\mathcal{E}| < M$ do
10:     Sample $\mathbf{v}_i$ and $\mathbf{v}_j$ from $\mathcal{V}$
11:     if $p \sim \mathcal{U}[0,1] < a_{i,j} * P(\text{COS-SIM}(\mathbf{v}_i, \mathbf{v}_j))$ then
12:         $\mathcal{E} \leftarrow \mathcal{E} \cup (i, j)$
13:     end if
14: end while
15: return $\mathcal{G}(\mathcal{V}, \mathcal{E})$
```

Table 3: Synthetic Experiment

Name	Parameters	Unsup. F1
MLP-Pool (baseline)	6544	0.56
TT-Pool	1148	0.54
KAN-Pool	6636	0.48

and optimize over 50 epochs. We fine-tune the learning rate of Adam on a validation set. Our test results are reported in Table 3.

6.2.2 Visualization

A big selling point of KANs is that they are "interpretable" and can be visualized. To this end, we plot the learned activations from our KAN model. Specifically, we found that KAN^k (13), which incorporates the neighborhood embedding with the original node embedding, changed the most after training. We plot the activations $\phi_{q,p}$ between each input node and all its outgoing edges for every input node of KAN^1 in Fig. 1. We can see that many input and output pairs learned similarly shaped activations, suggesting there could be redundancy in our architecture.

We also use t-SNE [16] to visualize and compare the embeddings created by each model. We observe that embeddings produced by MLP-based aggregators in Fig. 2 are more similar to ones produced by the TT-based aggregators in Fig. 3. The KAN-based aggregators produced a denser representation where nodes from different classes are very tight, which could be the reason for its lower performance on this dataset.

7 Discussion

Overall, we found that tensorizing the GraphSAGE framework was able to greatly reduce the number of trainable parameters while maintaining comparable F1 scores, suggesting that weights trained on graph data are highly compressible. Furthermore, our results show that KANs can match and even exceed the performance of GraphSAGE. In the synthetic task, however, the KAN model exhibited poorer performance, which may suggest that using projection layers may not be suited for smaller datasets. We refrain from making any claims on any theoretical improvements, as empirical results are often variable and sensitive to more rounds of hyperparameter tuning. Additionally, we only evaluate the performance of each unsupervised model through an extrinsic evaluation metric. Our main goal is to show the viability of applying tensorization and KAN to graph models.

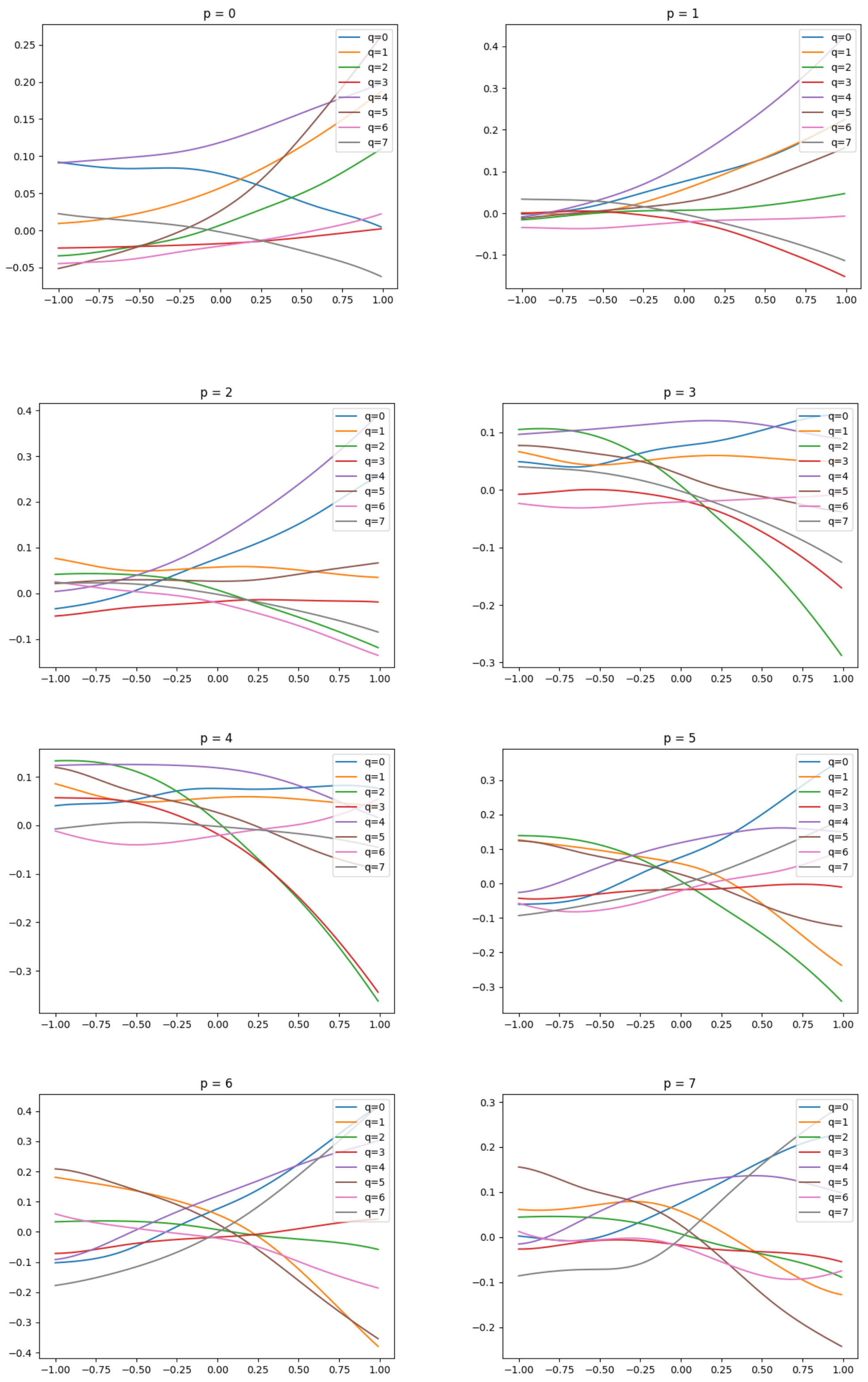

Figure 1: Plots of the B-spline activations learned by KAN on the synthetic task.

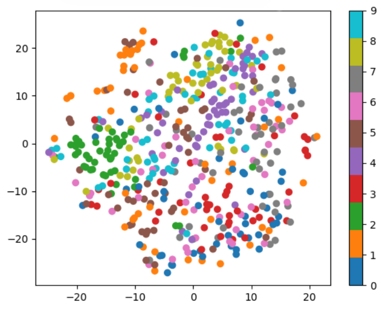

Figure 2: MLP t-SNE Node Embeddings

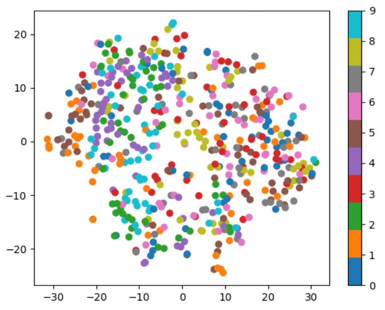

Figure 3: TT t-SNE Node Embeddings

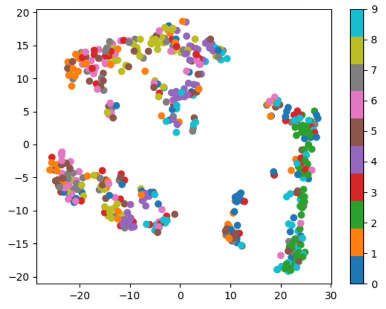

Figure 4: KAN t-SNE Node Embeddings

Future directions Our results are preliminary and do not take into account the computational overhead from forgoing sparse matrix multiplication, which is built into graph machine learning libraries. Our aggregators still rely on the max-pooling operation to remain permutation invariant with respect to graph nodes. Similar to the proposed LSTM-aggregator in the original GraphSAGE paper, we may try to train a KAN-based aggregator that replaces the max-pooling operation entirely using randomly permuted node order during training.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[2] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[3] Lise Getoor. *Link-based Classification*, pages 189–207. Springer London, London, 2005.

[4] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[5] Wei Ju, Zheng Fang, Yiyang Gu, Zequn Liu, Qingqing Long, Ziyue Qiao, Yifang Qin, Jianhao Shen, Fang Sun, Zhiping Xiao, et al. A comprehensive survey on deep graph representation learning. *Neural Networks*, page 106207, 2024.

[6] Mehrdad Kiamari, Mohammad Kiamari, and Bhaskar Krishnamachari. Gkan: Graph kolmogorov-arnold networks, 2024.

[7] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[8] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. Tensorly: Tensor learning in python, 2018.

[9] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y. Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks, 2024.

[10] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks, 2015.

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

[12] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python, 2018.

[13] Zheng Qu, Dimin Niu, Shuangchen Li, Hongzhong Zheng, and Yuan Xie. Tt-gnn: Efficient on-chip graph neural network training via embedding reformation and hardware optimization. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 452–464, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Yangjun Ruan, Yuanhao Xiong, Sashank Reddi, Sanjiv Kumar, and Cho-Jui Hsieh. Learning to learn by zeroth-order oracle, 2019.

[15] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[16] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[17] Chunxing Yin, Da Zheng, Israt Nisa, Christos Faloutos, George Karypis, and Richard Vuduc. Nimble gnn embedding with tensor-train decomposition, 2022.

[18] Fan Zhang and Xin Zhang. Graphkan: Enhancing feature extraction with graph kolmogorov arnold networks, 2024.

[19] Marinka Zitnik and Jure Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198, July 2017.

A Appendix / supplemental material

A.1 Implementation

We build our implementation on top of the GraphSAGE layer from Pytorch Geometric [2]. We use the TensorLy library [8] implementation of TT-SVD for automatically determining ranks during weight tensorization. [14]