Totally Dynamic Hypergraph Neural Network

Peng Zhou[1], **Zongqian Wu**[1], **Xiangxiang Zeng**[3], **Guoqiu Wen**[1*], **Junbo Ma**[1], **Xiaofeng Zhu**[1,2*]

[1]Guangxi Key Lab of Multi-Source Information Mining and Security, Guangxi Normal University, Guilin 541004, China

[2] University of Electronic Science and Technology of China

[3]Hunan University

Abstract

Recent dynamic hypergraph neural networks (DHGNNs) are designed to adaptively optimize the hypergraph structure to avoid the dependence on the initial hypergraph structure, thus capturing more hidden information for representation learning. However, most existing DHGNNs cannot adjust the hyperedge number and thus fail to fully explore the underlying hypergraph structure. This paper proposes a new method, namely, totally hypergraph neural network (TDHNN), to adjust the hyperedge number for optimizing the hypergraph structure. Specifically, the proposed method first captures hyperedge feature distribution to obtain dynamical hyperedge features rather than fixed ones, by conducting the sampling from the learned distribution. The hypergraph is then constructed based on the attention coefficients of both sampled hyperedges and nodes. The node features are dynamically updated by designing a simple hypergraph convolution algorithm. Experimental results on real datasets demonstrate the effectiveness of the proposed method, compared to SOTA methods. The source code can be accessed via https://github.com/HHW-zhou/TDHNN.

1 Introduction

Graphs are widely used in real applications including social networking [Berahmand *et al.*, 2021], web search [Wang *et al.*, 2021], and recommendation systems [Wu *et al.*, 2022] since they can efficiently capture relationships among data. However, the construction of the regular graph is based on pairwise relations, which makes it difficult to describe complex relationships. Hypergraphs can handle this problem naturally. The hypergraph is a good alternative to address the above issue by connecting every hyperedge with an arbitrary number of nodes. As a result, the hypergraph is able to capture complex relationships among nodes, and thus having more expressive ability than regular graphs.

*Corresponding author (seanzhuxf@gmail.com)

This work was supported in part by the National Key Research and Development Program of China under Grant No. 2022YFA1004100.

Previous hypergraph methods can be divided into two categories, *i.e.,* static hypergraph neural networks (SHGNNs) and dynamic hypergraph neural networks (DHGNNs). SHGNNs conduct representation learning by using the initialized hypergraph structure to capture the relatuionships among nodes. To do this, HGNN [Feng *et al.*, 2019] employed the convolution method and HGNN^{+} [Gao *et al.*, 2022] introduced a spatial-based hypergraph convolution. However, SHGNNs are highly dependent on the initialized hypergraph structure, which usually has redundant information and cannot discover hidden relationships among nodes. To address these issues, DHGNNs were proposed to learn latent connections from features and can mine more useful information. For instance, [Jiang *et al.*, 2019] proposed to reconstruct the hypergraph using both kNN and k-means. [Bai *et al.*, 2021] proposed using continuous values to construct the incidence matrix and the attention mechanism to learn the connection weights. DeepHGSL [Zhang *et al.*, 2022] uses hidden representations in multiple hypergraph convolutional layers to construct the hypergraph. HSL [Cai *et al.*, 2022] samples hyperedges from the initial hypergraph structure (*i.e.,* removing redundant hyperedges) and utilizes attention mechanisms to capture more relationships among nodes and hyperedges for the hypergraph construction.

Previous DHGNNs ignore the adjustment of the number of hyperedges, *i.e.,* hyperedge number, resulting in being unavailable to correctly explore the hypergraph structure. That is, no matter how the nodes are allocated, the hyperedge number is always the same. To address this issue, t-DHL [Gao *et al.*, 2020] attempts to adaptively adjust the hyperedge number by projecting the hyperedge space onto a binary tensor. However, it is a traditional machine learning method and cannot be used in an end-to-end way, thus difficultly exploring the relationship among nodes. Besides, we observe that it is essential for considering the hyperedge features to adjust the hyperedge number. Actually, a hyperedge connects a set of nodes, so that the common information among these nodes (the hyperedge features for short) can be used for characterizing this set of nodes. In contrast, the hyperedge is not necessary if it cannot represent the common characteristics of a set of nodes by a specific measurement, *e.g.,* the attention coefficients between the hyperedge and the nodes in this paper. In this way, the hyperedge number is updated with the hyperedge features. However, the hyperedge features are unavail-

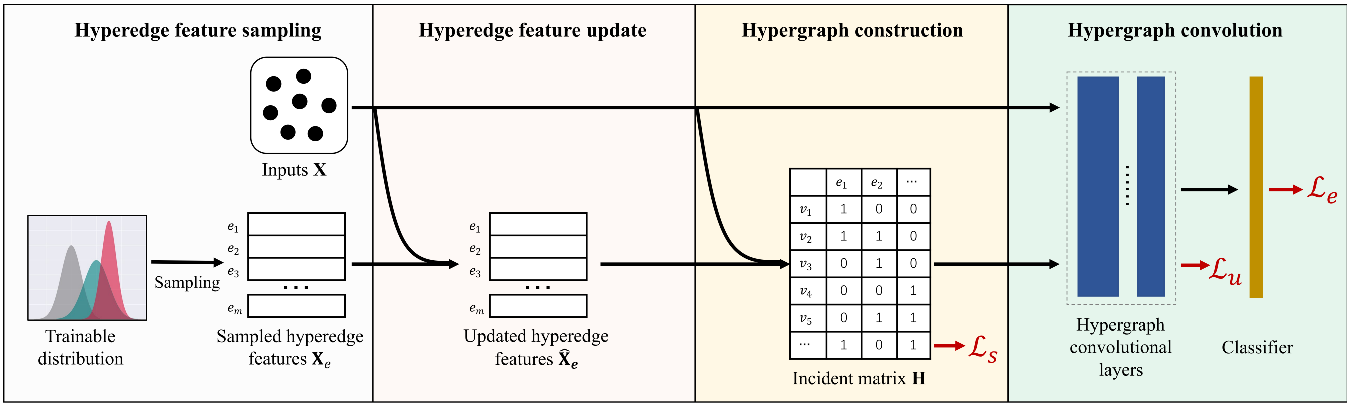


Figure 1: The architecture of the proposed TDHNN involving four steps, *i.e.,* (1) Hyperedge feature sampling randomly samples m hyperedges from a trainable hyperedge feature distribution; (2) Hyperedge feature update renews hyperedge features by the attention coefficients of the sampled hyperedge features and the node features; (3) Hypergraph construction builds the hypergraph by assigning nodes to the hyperedges; (4) Hypergraph convolution updates node features by a simple hypergraph convolutional layer.

able for most datasets and few studies focused on considering the hyperedge features for hypergraph neural networks.

To address the above issues, in this paper, we propose a new method, namely, Totally Dynamic Hypergraph Neural Network (TDHNN) shown in Figure 1, to learn dynamical hyperedge features for updating the hyperedge number, including four steps, *i.e.,* hyperedge feature sampling, hyperedge feature update, hypergraph construction, and hypergraph convolution. The first two steps generate the hyperedge features and the third step adjusts the hyperedge number. Hence, the aforementioned issues in previous methods have been explored. In particular, both of the hyperedge features and the hyperedge number are adaptively adjusted with the updated node features. As a result, our method avoids the influence of the low quality of the initial hypergraph.

Different from previous methods, the main contributions of our proposed method are summarized as follows:

- We propose a new end-to-end dynamic hypergraph framework, which can dynamically adjust the hypergraph structure and the number of hyperedges.
- We propose a simple hypergraph convolution algorithm based on the learned hyperedge features.
- We propose a supervised constraint loss and an unsupervised constraint loss to improve the learned hypergraph.

2 Methodology

A hypergraph is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{H})$. $\mathcal{V} = \{v_1; v_2; ...; v_n\}$ is the set of all nodes accompanied by nodes feature matrix $\mathbf{X}_v \in \mathbb{R}^{n \times d_n}$ in which feature dimension is d_n; $\mathcal{E} = \{e_1; e_2; ...; e_m\}$ is the set of all hyperedges, with each hyperedge $e \subset \mathcal{E}$ constitutes a subset of $\mathcal{V}$ [Arya *et al.*, 2020]. Similar to $\mathcal{V}$, $\mathcal{E}$ should also have a feature matrix $\mathbf{X}_e \in \mathbb{R}^{m \times d_e}$, although this feature matrix is unavailable in most datasets. $\mathbf{H} \in \mathbb{R}^{n \times m}$ is the incident matrix, which implies the topology of the hypergraph. In incident matrix $\mathbf{H}$, each row $\mathbf{h}_i$ represents the relationship between the i-th node and all hyperedges, $\mathbf{h}_{i,j} = 1$ means there is a connection between the i-th node and the j-th hyperedge, otherwise the opposite. $\mathbf{h}_{i,j}$ can also be a continuous value, indicating the connection strength between the i-th node and the j-th hyperedge. Since $\mathbf{H}$ implies the topology of a hypergraph, we also use $\mathbf{H}$ to denote the hypergraph for simplicity. The degree of a hyperedge e_i indicates the number of nodes contained in this hyperedge, denoted by $\lceil(e_i)$. In this paper, given a set of nodes $\mathcal{V}$ and its feature matrix $\mathbf{X}_v$, our goal is to learn a hyperedge feature distribution $\mathcal{P}(\mathbf{X}_e|\mathbf{X}_v)$ and use this distribution to sample a suitable number of hyperedges to construct a hypergraph $\mathbf{H}$, then use hypergraph convolution to update node features.

2.1 Hyperedge Feature Sampling

Common methods for constructing hypergraphs, such as kNN [Huang *et al.*, 2009], $l1$-hypergraph [Wang *et al.*, 2015], or constructing hypergraph based on existing graph structures [Fang *et al.*, 2014], all construct hyperedges centered at nodes, which means the number of hyperedges is equal to the number of nodes. As the sample size increases, the number of hyperedges also becomes very large. Not only will it cause many nodes to appear on multiple hyperedges, but it will also reduce computational efficiency. If we have the features of hyperedges in advance and construct hypergraphs centered on hyperedges, we can break the above limitations. However, hyperedge features are unavailable for most datasets, which poses a challenge. To solve this problem, we propose to sample hyperedges from a trainable distribution $\mathcal{P}(\mathbf{X}_e|\mathbf{X}_v)$, and use the relationship between the sampled hyperedges and training samples to update hyperedge features and this distribution. Specifically, we assume that each hyperedge's feature dimension is independent and obeys a trainable Gaussian distribution $(\mathbf{X}_e)_{i,j} \sim N(\boldsymbol{\mu}^j, diag(\boldsymbol{\sigma}^j))$, where $\boldsymbol{\mu} \in \mathbb{R}^{1 \times d_e}$ and $\boldsymbol{\sigma} \in \mathbb{R}^{1 \times d_e}$. At the very beginning, we randomly initialize this distribution and sample m times to get the initial hyperedge feature matrix $\mathbf{X}_e \in \mathbb{R}^{m \times d_e}$. Since the sampling process is discrete and cannot produce gradients, we use reparameterization skill [Kingma and Welling, 2013] to make $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ trainable, *i.e.,* we first sample $\mathbf{Q} \in \mathbb{R}^{m \times d_e}$

from $N(\mathbf{0}, \mathbf{1})$, and then use the following formula to get $\mathbf{X}_e$:

$$(\mathbf{X}_e)_i = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \mathbf{Q}_i, \tag{1}$$

where $\odot$ represents the Hadamard multiplication.

2.2 Hyperedge Feature Update

Subsequently, we propose to use the attention coefficient to represent the correlation between each hyperedge and all nodes. Nevertheless, attention can only be computed if the hyperedge features and node features belong to the same feature space [Bai *et al.*, 2021], so we first use a mapping function $f(\cdot)$ to map the input features $\mathbf{X}_v$ onto the feature space of the hyperedge features:

$$\hat{\mathbf{X}}_v = f(\mathbf{X}_v) \in \mathbb{R}^{n \times d_e}. \tag{2}$$

Then we adopt the same attention calculation method as Transformer [Vaswani *et al.*, 2017], including $q(\cdot)$, $k(\cdot)$ and $v(\cdot)$ three trainable mapping function:

$$\boldsymbol{\alpha}_{e_i,v_j} = \frac{e^{q((\mathbf{X}_e)_i)\cdot k((\hat{\mathbf{X}}_v)_j^\top)}}{\sum_l e^{q((\mathbf{X}_e)_i)\cdot k((\hat{\mathbf{X}}_v)_l^\top)}}, \tag{3}$$

where $\mathbf{A}_e \in \mathbb{R}^{m \times n}$ is the attention matrix of hyperedges, $\boldsymbol{\alpha}_{e_i,v_j}$ is the attention coefficient of hyperedge e_i and node v_j. After that, we aggregate the features of the top k_n nodes with the highest attention coefficient into the hyperedge:

$$\hat{\mathbf{X}}_e = MLP(concate(\mathbf{X}_e, top_{k_n}(\mathbf{A}_e) \cdot v(\hat{\mathbf{X}}_v))), \tag{4}$$

where $MLP(\cdot)$ is Multi-Layer Perceptron, $concate(\cdot)$ means concatenation. The $top_{k_n}(\cdot)$ here means that for each row of $\mathbf{A}_e$, keep the first k_n values and set the rest to zero. In this way, we connect $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ with the input features $\mathbf{X}_v$. Thus we can use backpropagation to update them. Note that we resample hyperedges at each iteration.

2.3 Hypergraph Construction

Now we have the features of m hyperedges. As mentioned before, building a hypergraph is actually a clustering process, and each hyperedge is equivalent to a cluster centroid. So, we have to put each node into appropriate clusters. From the perspective of clustering, we have the features of the cluster centroids and the features of the samples; the easiest way is to calculate the distance between each sample and each cluster centroid and then divide each sample into the nearest cluster. Nevertheless, from the hypergraph structure and application perspective, a node can be connected to an arbitrary number of hyperedges. For example, in a co-author network, each author can associate multiple works simultaneously. Here we still use the Transformer-like attention method to calculate the attention coefficient between nodes and hyperedges instead of their distances. However, unlike using the hyperedges queries $q(\mathbf{X}_e)$ to match the keywords of nodes $k(\hat{\mathbf{X}}_v)$ in the previous step, in this step, we use the nodes queries $q(\hat{\mathbf{X}}_v)$ to match the hyperedges keywords $k(\hat{\mathbf{X}}_e)$:

$$\boldsymbol{\alpha}_{v_i,e_j} = \frac{e^{q((\hat{\mathbf{X}}_v)_i)\cdot k((\hat{\mathbf{X}}_e)_j^\top)}}{\sum_l e^{q((\hat{\mathbf{X}}_v)_i)\cdot k((\hat{\mathbf{X}}_e)_l^\top)}}, \tag{5}$$

where $\mathbf{A}_v \in \mathbb{R}^{n \times m}$ is the attention matrix of hyperedges, $\boldsymbol{\alpha}_{v_i,e_j}$ is the attention coefficient of node v_i and hyperedge e_j. This reverse is because we need to ensure there are no isolated nodes since isolated nodes cannot exchange information from the constructed hypergraph, which may affect the performance of downstream tasks. Then we put each node into the k_e hyperedges with the highest attention coefficient:

$$\mathbf{H} = top_{k_e}(\mathbf{A}_v), \tag{6}$$

the $top_{k_e}(\cdot)$ here means that for each row of $\mathbf{A}_v$, keep the first k_e values, and set the rest to zero.

Supervised Constraint

To improve the learned hypergraph structure, we designed a supervised constraint function and an unsupervised constraint function. As mentioned above, each hyperedge can be regarded as the centroid of a cluster, and constructing a hypergraph is to divide nodes with the same label into the same cluster. In other words, for any pair of nodes v_i and v_j with the same label, we hope them to be connected to the same hyperedges. Since the i-th row of the incident matrix $\mathbf{H}$ is the connection between the i-node and hyperedges and the $\mathbf{H}$ we learned in Eq. (6) is continuous, what we have to do is to minimize the distance between $\mathbf{h}_i$ and $\mathbf{h}_j$, *i.e.,* $\min d(\mathbf{h}_i, \mathbf{h}_j)$. From this, we can get the loss function:

$$\mathcal{L}_s = \sum_{c \in \mathcal{C}} \sum_{v_i \in c; v_j \in c} d(\mathbf{h}_i, \mathbf{h}_j), \tag{7}$$

where $\mathcal{C}$ represents the set of categories of nodes in a dataset, $c \in \mathcal{C}$ represents a specific category and $d(\cdot)$ is the distance function. Eq. (7) can also be regarded as a kind of contrastive learning using only positive samples.

Adjusting the Number of Hyperedges

Since we learn the distribution of hyperedge features instead of fixed hyperedge features, we can change the number of hyperedges by adjusting the number of samples m. The challenge is that the number of samples is non-differentiable, and we cannot adjust it from backpropagation. From the graph theory perspective, we can solve this challenge if we know what kind of hyperedge and node relationship constitutes an optimal hypergraph. However, this is also a very challenging topic, and there is currently no relevant theoretical research. In order to allow the model to find a better sampling number adaptively, we designed a simple but effective evaluation criterion to judge whether the number of learned hyperedges is appropriate. We define the saturation score of a hypergraph:

$$S_{\mathbf{H}} = 1 - \frac{|\mathcal{E}_{empty}|}{|\mathcal{E}|}, \tag{8}$$

where $\mathcal{E}_{empty} = \{e | e \in \mathcal{E}, \lceil_e = 0\}$ is the set of empty hyperedges in the hypergraph, and $S_{\mathbf{H}}$ is the saturation score of the hypergraph $\mathbf{H}$, *i.e.,* the proportion of non-empty hyperedges to all hyperedges. This idea is very intuitive: if there are too many empty hyperedges in a hypergraph, which proves that these hyperedges are redundant, we then reduce the number of samples; at the same time, due to the extreme situation of sampling, it is possible to sample outliers, so we should also allow a few numbers of empty hyperedges. In order to achieve this, we set two hyperparameters $\beta \in [0,1]$ and $\gamma \in [0,1]$, which represent the lower and upper limits of saturation score, respectively. After each iteration, we adjust the number of samples according to the saturation score:

$$m = \begin{cases} m-1, S_{\mathbf{H}} < \beta; \\ m+1, S_{\mathbf{H}} > \gamma. \end{cases} \tag{9}$$

2.4 Hypergraph Convolution

Simple Hypergraph Convolutional Layer

We then use the learned hyperedge features and the constructed hypergraph to update the nodes' features. [Feng *et al.*, 2019] proposed the first hypergraph convolution formula:

$$\mathbf{X}_v^{(t+1)} = \sigma(\mathbf{D}_v^{-\frac{1}{2}}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^\top\mathbf{D}_v^{-\frac{1}{2}}\mathbf{X}_v^{(t)}\mathbf{\Theta}^{(t)}). \tag{10}$$

Although Eq. (10) is derived from the spectral domain based on the Fourier transform, we can still interpret it from the perspective of the spatial domain where $\mathbf{D}_v^{-\frac{1}{2}}$, $\mathbf{W}$ and $\mathbf{D}_e^{-1}$ are all diagonal matrices that are equivalent to regularization terms. If we remove these regularization terms, then we have:

$$\mathbf{X}_v^{(t+1)} = \sigma(\mathbf{H}\mathbf{H}^\top\mathbf{X}_v^{(t)}\mathbf{\Theta}^{(t)}). \tag{11}$$

In Eq. (11), $\mathbf{H}^\top\mathbf{X}_v^{(t)}\mathbf{\Theta}^{(t)}$ is to multiply the feature matrix of nodes with a weight matrix and then convert it into the feature matrix of hyperedges according to the hypergraph structure. Then it is multiplied by $\mathbf{H}$, which assigns the hyperedge features to each corresponding node again. Since the hyperedge features are sampled from a trainable distribution, we do not need to convert node features to hyperedge features in Eq. (11). Thus the convolution formula can be simplified as:

$$\mathbf{X}_v^{(t+1)} = \sigma(\mathbf{H}^{(t)}\mathbf{X}_e^{(t)}\mathbf{\Theta}^{(t)}). \tag{12}$$

We first multiply the learned features of hyperedges by a weight matrix $\mathbf{\Theta} \in \mathbb{R}^{d_e \times d}$ and then assign the hyperedge features to each relevant node according to the learned hypergraph structure. Note that at each layer, we reconstruct $\mathbf{H}$ according to the current layer's hyperedge and node features.

A previous work [Huang and Yang, 2021] pointed out that self-loops are very important for hypergraph convolution. In a hypergraph, self-loops are hyperedges that contain only one node. If self-loops are not introduced, the representation of a node will only be affected by the features of its neighbor nodes in the previous layer and lose its own features. Therefore, Eq. (12) can be modified as:

$$\mathbf{X}_v^{(t+1)} = \sigma(\tilde{\mathbf{H}}^{(t)}\tilde{\mathbf{X}}_e^{(t)}\mathbf{\Theta}^{(t)}), \tag{13}$$

where $\tilde{\mathbf{H}}^{(t)} = concat(\mathbf{H}^{(t)}, \mathbf{H}^{'}) \in \mathbb{R}^{n\times(m+n)}$, $\mathbf{H}^{'} \in \mathbb{R}^{n\times n}$ is a diagonal incident matrix in which each node v_i belongs to only one hyperedge e_i with degree $\lceil(e_i) = 1$. Since there is only one node in the hyperedge of the self-loop, we can directly regard the features of the node as the hyperedge features, *i.e.,* $\tilde{\mathbf{X}}_e^{(t)} = stack(\mathbf{X}_e^{(t)}, \hat{\mathbf{X}}_v^{(t)}) \in \mathbb{R}^{(m+n)\times d_e}$. However, instead of taking the form of Eq. (13), our final convolution formula is:

$$\mathbf{X}_v^{(t+1)} = \sigma(w\hat{\mathbf{X}}_v^{(t)} + \mathbf{H}^{(t)}\mathbf{X}_e^{(t)}\mathbf{\Theta}^{(t)}), \tag{14}$$

where w is a trainable parameter. That is, we add the features of nodes themselves on the basis of Eq. (12). In fact, Eq. (14) is equally valid and more computationally efficient than Eq. (13).

Unsupervised Constraint

We use the labeled nodes to get the supervised constraint and hope that the nodes with the same label would be divided into the same hyperedge; similarly, for the unsupervised constraint, we hope that the nodes divided into the same hyperedge would be more similar. In the supervised stage, our constraint loss was directly applied to $\mathbf{H}$, and in the unsupervised stage, we added a constraint on node features after the convolutional layer:

$$\mathcal{L}_u = \sum_{e\in\mathcal{E}}\sum_{v_i\in e, v_j\in e} d((\mathbf{X}_v)_i, (\mathbf{X}_v)_j). \tag{15}$$

Finally, we evaluate our model on the node classification task, so the final loss function is:

$$\mathcal{L} = \mathcal{L}_e + \lambda_1\mathcal{L}_s + \lambda_2\mathcal{L}_u, \tag{16}$$

where $\mathcal{L}_e$ is the empirical loss of node classification, and λ_1 and λ_2 are trade-off hyperparameters.

2.5 The Connection and Difference with k-means

The way we construct a hypergraph is similar to k-means. The primary process of k-means is (1) randomly selecting cluster centroids, (2) clustering, (3) updating the cluster centroids, then repeating (2) and (3) until convergence. The pipeline of the proposed TDHNN is (1) sampling hyperedges, (2) assigning nodes to each hyperedge, (3) updating the feature distribution of hyperedges, then repeating (2) and (3) until convergence. The main differences between our method and k-means are as follows:

- k-means learns features of cluster centroids, while TDHNN learns the feature distribution of hyperedges;
- k-means selects fixed k cluster centroids, while TDHNN dynamically adjusts the number of samples according to the saturation score of the learned hypergraph;
- Each cluster in k-means is disjoint, while TDHNN has intersections between hyperedges.

3 Experiments

3.1 Experimental Settings

Datasets

Following the work of the first hypergraph convolutional neural network [Feng *et al.*, 2019], we use two visual object classification datasets (*i.e.,* Princeton ModelNet40 [Wu *et al.*, 2015] and National Taiwan University 3D model (NTU for short)). Among them, ModelNet40 contains $12,311$ objects with 40 types, and NTU contains $2,012$ objects with a total of 67 types of 3D shapes. Like [Feng *et al.*, 2019], we use Group-View Convolutional Neural Network (GVCNN) [Feng *et al.*, 2018] and Multi-view Convolutional Neural Network (MVCNN) [Su *et al.*, 2015] for feature extraction and consider the case of using one set of features separately and using two sets of features at the same time. We adopted the same split standard for ModelNet40 and NTU, *i.e.,* 80% as the training set and 20% as the testing set. Since the standard split of a dataset uses fixed training samples, a method may be affected by the fixed data distribution. For better comparison,

	ModelNet40			NTU		
	GVCNN	MVCNN	GV+MV	GVCNN	MVCNN	GV+MV
GCN	91.80±0.46	91.50±1.80	94.85±1.75	78.80±0.92	78.72±1.97	80.43±1.09
GAT	91.65±0.25	90.07±0.41	95.75±0.14	79.60±0.03	78.50±1.17	80.16±1.08
HGNN	91.80±1.73	91.00±0.66	96.96±1.43	82.50±1.62	79.10±0.90	83.64±0.37
HGNN$^+$	92.50±0.08	90.60±1.68	96.92±1.81	82.80±1.11	76.40±1.17	84.18±0.82
HNHN	92.10±1.76	91.10±1.84	93.80±1.84	83.10±1.89	79.60±0.79	80.60±0.95
DHGNN	92.13±1.55	85.53±0.83	96.99±1.46	82.30±0.98	77.60±1.55	85.13±0.26
HyperGCN	92.20±0.80	90.20±0.28	96.10±0.63	79.90±1.78	78.10±0.83	79.90±0.91
DeepHGSL	89.32±0.71	88.62±0.93	90.33±0.66	76.28±1.45	72.30±1.57	78.67±0.77
HSL	93.17±0.25	91.44±0.42	96.92±0.41	81.82±1.30	75.68±1.41	82.26±1.20
TDHNN	**93.81±1.04**	**92.33±1.67**	**97.52±0.80**	**83.69±0.45**	**79.62±0.53**	**86.05±1.10**

Table 1: Visual object classification accuracy on ModelNet40 and NTU. GVCNN and MVCNN indicate that the features extracted by GVCNN and MVCNN are used as input, respectively; GV+MV indicates that the two sets of features are concatenated as input. We report the mean and standard deviation over 20 runs.

we use the same method as [Jiang *et al.*, 2019] to randomly sample different proportions of the data on Cora [Veličković *et al.*, 2017] as the training set. Specifically, in addition to the standard split, we respectively select 2%, 5.2%, 10%, 20%, 30%, and 44% of the data to train.

Comparison Methods

Our comparison methods include two classic graph models, *i.e.,* GCN [Kipf and Welling, 2016] and GAT [Veličković *et al.*, 2017]; three hypergraph convolutional neural networks, *i.e.,* HGNN, HGNN$^+$, and HNHN [Dong *et al.*, 2020]; and two dynamic hypergraph networks, *i.e.,* DHGNN, HyperGCN [Yadati *et al.*, 2019], DeepHGSL [Zhang *et al.*, 2022] and HSL [Cai *et al.*, 2022]. We implement GCN and GAT by the open tool PyTorch Geometric [Fey and Lenssen, 2019], and we implement HGNN, HGNN$^+$, HNHN, HyperGCN and DeepHGSL by the open tool DHG (DeepHypergraph) [Gao *et al.*, 2022]. For DHGNN and HSL, we use their source code for experiments.

Setting-up

We uniformly set the feature dimension of the hyperedge d_e to 128 and the initial sampling number of the hyperedge m to 100. The number of nodes used to update the hyperedge features and the number of hyperedges each node belongs to are set to 10. For the hypergraph saturation score, we set the lower limit β to 0.9, and the upper limit γ is set to 0.95. We used dropout [Srivastava *et al.*, 2014] to prevent overfitting and set the drop rate to 0.2. The optimizer we use is Adam [Kingma and Ba, 2014], and the learning rate is 0.001.

3.2 Result and Discussion

Visual Object Classification

The experimental results of visual object classification are shown in Table 1. The proposed TDHNN achieves the best results among all comparison methods, regardless of the set of features used, both on ModelNet40 and NTU. Compared with graph algorithms GCN and GAT, TDHNN has an average increase of 2.82% and 2.88%, respectively. Compared with the static hypergraph algorithms HGNN, HGNN$^+$, and HNHN, TDHNN has an average improvement of 1.33%, 1.6%, and 2.12%, respectively. Compared with dynamic hypergraph algorithms DHGNN, HyperGCN, DeepHGSL and HSL, our classification accuracy has averagely increased by 2.22%, 2.77%, 6.25% and 1.95% respectively. In general, hypergraph algorithms outperform graph algorithms on ModelNet40 and NTU. However, some algorithms show instability in the experimental results. For example, when HNHN uses the features of GVCNN and MVCNN on ModelNet40, respectively, it shows high accuracy (92.10% and 91.10%, respectively). However, the accuracy improvement is not obvious when fusing these two sets of features simultaneously (93.80%). Compared with other methods, it is even lower, and a similar situation occurs at NTU. When DHGNN uses the features of MVCNN alone, the accuracy is relatively low, whether it is on ModelNet40 or NTU, but it achieves good grades when fusing these two sets of features. In sharp contrast to these two methods, TDHNN has achieved better results, showing better stability and compatibility, whether using a specific feature alone or fusing two sets of features simultaneously.

Overall, the proposed TDHNN has three advantages: first, by applying the hypergraph structure, we can better mine the multi-relationships in the data; second, we dynamically learn the hypergraph structure, which can fully mine the potential relationships in the data; third, we can dynamically adjust the number of hyperedges to help learn a more reasonable hypergraph structure.

Citation Network Classification

The experimental results of experiments on Cora using different proportion samples are shown in Table 2. As is shown, TDHNN has achieved the best results except for the 2% split, which is 76.11% second to DHGNN (76.90%) with a gap of 0.8%. Compared with graph algorithms GCN and GAT, TDHNN has an average increase of 4.11% and 1.57%, respectively. Compared with the static hypergraph algorithms HGNN, HGNN$^+$, and HNHN, TDHNN has an average improvement of 3.14%, 4.37%, and 6.18%, respectively. Compared with dynamic hypergraph algorithms DHGNN, Hyper-

lr	GCN	GAT	HGNN	HGNN$^+$	HNHN	DHGNN	HyperGCN	DeepHGSL	HSL	TDHNN
std	81.50±1.59	83.00±0.37	81.80±0.79	79.60±1.59	76.80±1.02	82.50±0.69	62.22±1.92	82.16±0.76	79.45±1.92	**83.60±0.62**
2%	69.60±0.19	74.80±0.81	75.40±1.17	71.40±1.41	66.50±2.06	**76.90±1.71**	46.21±1.53	74.52±1.74	74.86±4.19	76.11±1.14
5.2%	77.80±0.43	79.40±0.47	79.70±0.75	76.20±1.32	73.10±1.52	80.20±0.20	54.25±0.84	78.66±2.12	77.91±0.80	**80.41±1.50**
10%	79.90±0.66	81.50±1.90	80.00±0.28	78.20±0.87	76.50±1.94	81.60±0.12	64.93±0.42	79.29±1.32	79.18±1.55	**84.53±1.69**
20%	81.40±0.57	83.50±1.67	80.10±1.08	81.40±1.43	80.90±1.83	83.60±1.79	72.51±0.02	80.32±1.18	81.69±1.39	**85.04±1.87**
30%	81.90±1.82	84.50±0.15	82.00±1.67	82.50±0.17	82.50±1.04	85.00±0.41	78.82±1.83	83.22±1.16	83.15±1.70	**85.86±1.81**
44%	82.00±0.71	85.20±0.23	81.90±0.09	83.00±0.35	83.30±0.96	85.60±0.69	82.44±0.62	83.65±0.89	84.98±1.69	**87.34±1.24**

Table 2: Node classification results on Cora. In addition to the standard division, we randomly select 2%, 5.2%, 10%, 20%, 30%, and 44% of the data, respectively as the training set.

GCN, DeepHGSL, and HSL, our classification accuracy has averagely increased by 1.07%, 17.38%, 2.86% and 3.09% respectively. Specifically, under the split of 2% and 5.2%, HGNN, HGNN$^+$, and our TDHNN have achieved relatively similar performance. However, with the increase of the training ratio, the advantage of TDHNN is revealed, which might be attributed to the supervised constraint loss $\mathcal{L}_s$.

3.3 Ablation Study

The proposed TDHNN has three main components, a hyperedge updater (HU for short), supervised constraints (C1 for short), and unsupervised constraints (C2 for short). To demonstrate the effectiveness of each part, we tested different combinations of these components. As shown in Table 3, first, TDHNN has an average increase of 1.22%, 3.75%, and 3.13% on ModelNet40, NTU, and Cora compared to using only a single component, respectively. In the case of only using HU, the performance of the model at each dataset is acceptable, especially on ModelNet40 (97.28%), which is close to using all components. In the case of using only C1, the overall performance drops a lot on ModelNet40 (94.24%) and NTU (77.74%), but it is slightly better on Cora than using only HU (increased by 0.3%). When only using C2, ModelNet40 also performed very well (97.36%), and it was also better than using only HU or C1 on Cora. When any two components are used in combination, the overall performance is not significantly improved compared with using a single component. Specifically, when combining C1 and C2, the model cannot even converge on ModelNet40 and NTU (the classification accuracy is 7.86% and 6.43%, respectively). However, when combining all these components, the model's performance achieves the best, and the case of non-convergence does not exist anymore.

HU	C1	C2	ModelNet40	NTU	Cora
✓			97.28±0.25	84.71±0.80	79.9±1.25
	✓		94.24±0.16	77.74±0.06	80.2±1.86
		✓	97.36±1.47	84.45±0.91	81.3±1.36
✓	✓		96.88±0.36	83.64±1.21	80.9±0.75
✓		✓	97.36±1.20	84.45±1.30	79.2±0.32
	✓	✓	07.86±0.15	06.43±0.64	79.6±0.53
✓	✓	✓	**97.52±0.80**	**86.05±1.10**	**83.6±0.62**

Table 3: Classification accuracy (mean and standard deviation) of our method with different components on all datasets.

3.4 Visualizations

In order to better show the ability of TDHNN, we use the t-SNE algorithm [Van der Maaten and Hinton, 2008] to reduce the dimensionality of the embedding of the model and visualize it. We also calculate the Silhouette score [Rousseeuw, 1987] of the embedding to evaluate it. We perform the same experiment on HGNN$^+$ and HNHN for comparison. The experimental results are shown in Figure 2. As is shown, in the experimental results of TDHNN, the boundaries of categories are clearer. The Silhouette scores also prove this point, where the Silhouette score of HGNN$^+$ and HNHN are 0.3531 and 0.3344, respectively, while TDHNN is 0.5081, which is 15.50% and 17.34% higher than the previous two methods.

3.5 Running Time

In order to demonstrate the computational efficiency of TDHNN, we compared the time required for each iteration with the latest dynamic hypergraph methods. As shown in the Table 4, TDHNN improved by an average of 9.44 seconds, 3.01 seconds, and 2.31 seconds per epoch compared to DHGNN, HyperGCN, and DeepHGSL on three datasets. Compared to HSL, TDHNN is on average 0.04 seconds slower, that is because HSL samples and obtains new structures from existing hypergraph structures rather than using features for reconstruction. In addition, we did not compare TDHNN with static methods as they did not reconstruct hypergraphs.

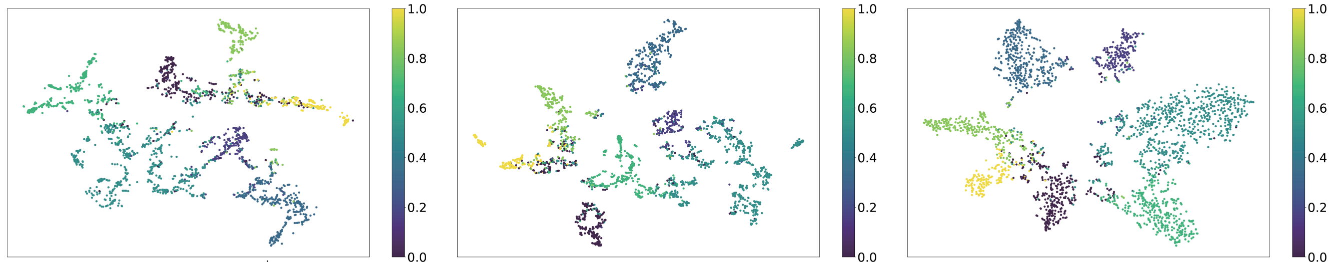
	Time per epoch (seconds)		
	ModelNet40	NTU	Cora
DHGNN	25.04	3.58	0.34
HyperGCN	6.71	1.16	1.81
DeepHGSL	5.33	1.27	0.97
HSL	0.42	0.06	0.02
TDHNN	0.42	0.11	0.11

Table 4: Running time for each epoch.

3.6 Sensitivity Analysis

Effect of Hypergraph Saturation Threshold

We tested the effect of setting the saturation threshold on the model on ModelNet40 and NTU. We set the lower threshold β from 0.1 to 0.9 with a step size of 0.1 and the corresponding upper threshold $\gamma = \beta + 0.05$. According to the



(a) HGNN$^+$ (b) HNHN (c) TDHNN

Figure 2: t-SNE embeddings of HGNN$^+$, HNHN, and the proposed TDHNN on Cora. The Silhouette scores of the embeddings learned by the three methods are 0.3531, 0.3344, and 0.5081, respectively.

experimental results, on ModelNet40, with the change of β and γ, the model's accuracy has almost no change. On NTU, the model's accuracy fluctuates slightly with the change of threshold; the model reaches its best at the point $\beta = 0.9$, but overall the difference is not much. The result shows that the learned hypergraph indeed has a large number of redundant edges. Although these redundant hyperedges do not affect the model's accuracy, they add many unnecessary calculations, reducing the computational efficiency. Thanks to our strategy of dynamically adjusting the number of hyperedges, our model can control the number of redundant hyperedges by setting a threshold, thus reducing the computational overhead. Specifically, considering the convolutional operation $\mathbf{H}^{(t)}\mathbf{X}_e^{(t)}\mathbf{\Theta}^{(t)}$ in Eq. (12), the time complexity is $O(n \times m \times d_e) + O(n \times d_e \times d) = O(nm)$. When using a traditional method such as kNN to generate a hypergraph, the number of hyperedges m is equal to the number of nodes n, then the time complexity is $O(n^2)$. In contrast, the proposed TDHNN sets m as a constant, and the model optimizes this constant at each iteration so that the time complexity of convolution is reduced to $O(n)$.

Trade-off between Two Constraints

In order to make the learned hypergraph better, we set a supervised constraint and an unsupervised constraint. How to set the weight of these two constraints is a question worth discussing. We conducted experiments on ModelNet40 and NTU, respectively, and set λ_1 and λ_2 from 10 to 100 with a step size of 10. It can be concluded that: (1) TDHNN is less sensitive to λ_1 and λ_2 since as λ_1 and λ_2 change, the variance of TDHNN's accuracy within each data distribution tested is relatively low; (2) for different data distributions, the influence of λ_1 and λ_2 is different, *i.e.,* the trade-off of λ_1 and λ_2 depends on the dataset.

4 Conclusion

In this paper, we propose a novel dynamic hypergraph learning framework that can dynamically adjust both the structure of the hypergraph and the number of hyperedges. As far as we know, this may be the first work that constructs hypergraphs centered on the features of hyperedges, and it opens up a new way for the research and learning of hypergraph neural networks. The proposed TDHNN learns the feature distribution of hyperedges and adjusts the number of samples according to the saturation score of the learned hypergraph to dynamically adjust the hypergraph structure. To make the constructed hypergraph more reasonable, we propose two constraints on the graph and features after convolution. Experiments demonstrate the effectiveness of our method. However, this method also has many aspects that can be improved. For example, the measurement of hypergraph saturation and the strategy of adjusting the number of hyperedges may be too simple; the relationship between the number of hyperedges and samples has yet to be further explored and theoretically proved. It is also the direction we need to continue in-depth research and exploration in the future.

Contribution Statement

Zongqian Wu made equal contributions to this work. He was involved in the overall design of the method, conducted comparative experiments, contributed to chart design, and participated in some writing tasks.

References

[Arya *et al.*, 2020] Devanshu Arya, Deepak K Gupta, Stevan Rudinac, and Marcel Worring. Hypersage: Generalizing inductive representation learning on hypergraphs. *arXiv preprint arXiv:2010.04558*, 2020.

[Bai *et al.*, 2021] Song Bai, Feihu Zhang, and Philip HS Torr. Hypergraph convolution and hypergraph attention. *Pattern Recognition*, 110:107637, 2021.

[Berahmand *et al.*, 2021] Kamal Berahmand, Elahe Nasiri, Mehrdad Rostami, and Saman Forouzandeh. A modified deepwalk method for link prediction in attributed social network. *Computing*, 103(10):2227–2249, 2021.

[Cai *et al.*, 2022] Derun Cai, Moxian Song, Chenxi Sun, Baofeng Zhang, Shenda Hong, and Hongyan Li. Hypergraph structure learning for hypergraph neural networks. In *IJCAI*, pages 1923–1929, 2022.

[Dong *et al.*, 2020] Yihe Dong, Will Sawin, and Yoshua Bengio. Hnhn: hypergraph networks with hyperedge neurons. *arXiv preprint arXiv:2006.12278*, 2020.

[Fang *et al.*, 2014] Quan Fang, Jitao Sang, Changsheng Xu, and Yong Rui. Topic-sensitive influencer mining in interest-based social media networks via hypergraph learning. *IEEE Transactions on Multimedia*, 16(3):796–812, 2014.

[Feng *et al.*, 2018] Yifan Feng, Zizhao Zhang, Xibin Zhao, Rongrong Ji, and Yue Gao. Gvcnn: Group-view convolutional neural networks for 3d shape recognition. In *CVPR*, pages 264–272, 2018.

[Feng *et al.*, 2019] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. Hypergraph neural networks. In *AAAI*, pages 3558–3565, 2019.

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

[Gao *et al.*, 2020] Yue Gao, Zizhao Zhang, Haojie Lin, Xibin Zhao, Shaoyi Du, and Changqing Zou. Hypergraph learning: Methods and practices. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[Gao *et al.*, 2022] Yue Gao, Yifan Feng, Shuyi Ji, and Rongrong Ji. Hgnn$^+$: General hypergraph neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[Huang and Yang, 2021] Jing Huang and Jie Yang. Unignn: a unified framework for graph and hypergraph neural networks. *arXiv preprint arXiv:2105.00956*, 2021.

[Huang *et al.*, 2009] Yuchi Huang, Qingshan Liu, and Dimitris Metaxas.] video object segmentation by hypergraph cut. In *CVPR*, pages 1738–1745, 2009.

[Jiang *et al.*, 2019] Jianwen Jiang, Yuxuan Wei, Yifan Feng, Jingxuan Cao, and Yue Gao. Dynamic hypergraph neural networks. In *IJCAI*, pages 2635–2641, 2019.

[Kingma and Ba, 2014] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Kingma and Welling, 2013] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

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

[Rousseeuw, 1987] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[Srivastava *et al.*, 2014] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[Su *et al.*, 2015] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. Multi-view convolutional neural networks for 3d shape recognition. In *ICCV*, pages 945–953, 2015.

[Van der Maaten and Hinton, 2008] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, volume 30, 2017.

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

[Wang *et al.*, 2015] Meng Wang, Xueliang Liu, and Xindong Wu. Visual classification by l1-hypergraph modeling. *IEEE Transactions on Knowledge and Data Engineering*, 27(9):2564–2574, 2015.

[Wang *et al.*, 2021] Meihong Wang, Linling Qiu, and Xiaoli Wang. A survey on knowledge graph embeddings for link prediction. *Symmetry*, 13(3):485, 2021.

[Wu *et al.*, 2015] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *CVPR*, pages 1912–1920, 2015.

[Wu *et al.*, 2022] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys*, 55(5):1–37, 2022.

[Yadati *et al.*, 2019] Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha Talukdar. Hypergcn: A new method for training graph convolutional networks on hypergraphs. In *NeurIPS*, volume 32, 2019.

[Zhang *et al.*, 2022] Zizhao Zhang, Yifan Feng, Shihui Ying, and Yue Gao. Deep hypergraph structure learning. *arXiv preprint arXiv:2208.12547*, 2022.