HyFER: A Framework for Making Hypergraph Learning Easy, Scalable and Benchmarkable

Hyunjin Hwang*
KAIST EE
Daejeon, South Korea
hyunjinhwang@kaist.ac.kr

Seungwoo Lee*
KAIST EE
Daejeon, South Korea
ksalsw1996@kaist.ac.kr

Kijung Shin
KAIST AI & EE
Daejeon, South Korea
kijungs@kaist.ac.kr

ABSTRACT

Interests in hypergraphs, which are a generalization of graphs, have emerged due to their expressiveness. This expressiveness causes some difficulties in applying deep learning techniques to hypergraphs, and a number of Hypergrpah Neural Networks (hyperGNNs) have overcome or bypassed such difficulties in their own way. Up until now, there is no standard way of doing so, and as a consequence, it takes much effort to directly compare different hyperGNNs even with their open-sourced implementation. In order to address this issue, we propose HyFER, an easy-to-use and efficient framework for implementing and evaluating hyperGNNs. Using HyFER, which is well modularized for easy adaptation to new datasets, models, and tasks, we could directly compare three hyperGNNs in two tasks on four datasets under the same settings.

1 INTRODUCTION

Graphs have been used extensively for representing and analyzing complicated and massive datasets. To realize its full potential, applying neural network technology to graph data has become one of the popular research areas. Neural network models considering the structural characteristics of graphs are required to this end, and *Graph Neural Networks (GNNs)*, such as Graph Convolution Network [15], have been actively studied [9, 15, 17, 19, 20, 24, 25].

While GNNs have become sophisticated, at the same time, interests in hypergraphs, an advanced form of graphs, have also increased [1–7, 10–12, 16, 21, 23]. Hypergraphs are capable of representing group interactions, while graphs only can represent pairwise interactions. To be specific, in hypergraphs, edges are extended to hyperedges, which contain an arbitrary number of nodes. Figure 1 shows an example of a hypergraph.

As a trade-off of this high capacity, however, hypergraphs are tricky to handle. First, among one-hop neighbors, nodes co-appearing at different hyperedges often need to be dealt with differently. Moreover, as hyperedges can contain any number of nodes, a hypergraph with $|V|$ nodes has $2^{|V|}$ potential hyperedges, and thus it is too costly to compute the likelihood of each potential hyperedge. Most *Hypergraph Neural Networks (hyperGNNs)* do have their own way of bypassing such difficulties, but there are no obvious standards up until now. Details, including data format, data splitting schemes, learning process, and evaluation metrics, vary a lot in their open source implementations, making it difficult to directly compare their performances or unify their source code. Additionally, when applying the same model to various hypergraph learning tasks, additional steps are required but not standardized.

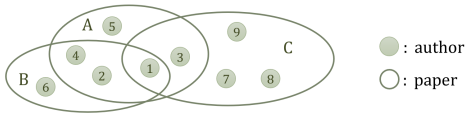

Figure 1: An example hypergraph: coauthorship hypergraph with 3 papers (hyperedges) and 9 authors (nodes).

To address such difficulties, we propose HyFER (**Hy**pergraph Neural Network **F**ramework for **E**fficiency and **R**eproducibility), a framework for implementing and evaluating hyperGNNs. Our framework is well-modularized, dividing the entire learning process into three parts, each of which can be replaced or customized, depending on datasets, hyperGNN models, tasks, etc. Moreover, to address the scalability problem that most existing implementations suffer from, HyFER is built on top of *Deep Graph Library (DGL) [18]*, which is a highly-efficient open-sourced library for GNNs. While providing convenience and modularity, HyFER is as efficient as most efficient open-sourced implementations in terms of memory and inference time. We summarize our contributions as follows:

- **Modularized Framework:** We modularize the proposed framework into three parts: (1) Data module, (2) Model module, and (3) Task module. This modularization helps users readily perform experiments on various datasets, models, and tasks.
- **Efficient Implementation:** We implement our framework on top of DGL, which is a highly optimized framework for GNNs. By taking advantage of its message-passing facilities, our framework achieves high speed and memory efficiency.
- **Direct Comparison:** Using our framework, we compare three hyperGNN models in two tasks in a direct and fair way under the same settings.

Reproducibility: The source code and datasets used in this work are available at https://github.com/ksalsw1996DM/HyFER.

In Section 2, we introduce several hyperGNN models. In Section 3, we present our framework. Next, we apply our framework to two concrete tasks in Sections 4 and 5. Lastly, we summarize our contributions in Section 6.

2 PRELIMINARIES AND RELATED WORK

In this section, we introduce some concepts and existing hyperGNN models, which are used throughout this study.

*Equal Contribution.

GLB '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

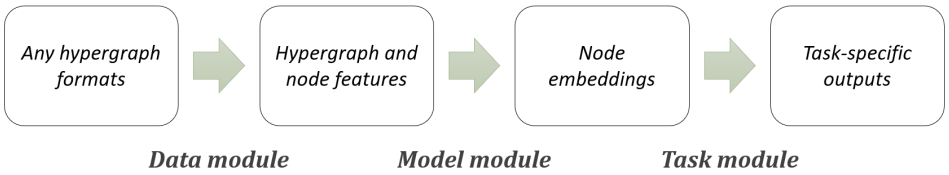

Figure 2: Three stages in HyFER, our proposed framework for hypergraph learning.

2.1 Functional Form of HyperGNNs

A *hypergraph* $G = (V, E)$ is defined as a set of nodes $V = \{v_1, ..., v_N\}$ and a set of hyperedges $E = \{e_1, ...e_M\}$, where each $e_i \in E$ is a subset of V. The *incidence matrix* $A \in \mathbb{R}^{|V|\times|E|}$ of a G is defined as:

$$A_{ij} = \begin{cases} 1 & \text{if } v_i \in e_j \\ 0 & \text{otherwise} \end{cases}$$

HyperGNNs obtain node embeddings (i.e., vector representations of nodes) from given node features and their connectivity. Given d node features as a matrix $H \in \mathbb{R}^{|V|\times d}$ and connectivity in the from of an incidence matrix $A \in \mathbb{R}^{|V|\times|E|}$, node embeddings in each l-th layer of hyperGNNs can be written as:

$$H^{(l+1)} = F_w(H^{(l)}, A), \quad H^{(0)} = H \tag{1}$$

where F_w is a function that varies with models.

2.2 Example HyperGNN Models

HGNN [7]. A general way of handling hypergraphs is to convert them into ordinary graphs, for example, by clique expansion or star expansion. HGNN is a hyperGNN model that uses star expansion. To be specific, it repeats (1) aggregating the embeddings of nodes belonging to each hyperedge, and (2) aggregating again those aggregated for the hyperedges that each node belongs to. This process results in node embeddings that summarize the features of neighboring nodes. If we let $D_e \in \mathbb{R}^{|E|\times|E|}$ and $D_v \in \mathbb{R}^{|V|\times|V|}$ as hyperedge and node degree matrices, which are diagonal, each l-th layer of HGNN can be written as:

$$H^{(l+1)} = \sigma(D_v^{-1/2} A D_e^{-1} A^T D_v^{-1/2} H^{(l)} W) \tag{2}$$

where W is a learnable parameter and σ is an activation function.

HNHN [6]. In HGNN, it is hard to treat nodes differently while aggregating embeddings. In addition to what a layer of HGNN has, each layer of HNHN has a normalization term with two hyperparameters α and β, which decides weights in aggregation. Specifically, the larger α (or β) is, the larger the effects of high-degree nodes (or hyperedges) in the aggregation are.

HAT. HNHN requires additional hyperparameters (i.e., α and β) that need to be tuned. Replacing them with self attention [13, 17] may lead to even better aggregation without any hyperparameter. However, using attention in hypergraphs in a straightforward way is memory intensive, and to bypass this problem, recent studies [3, 23] proposed simplified attention mechanisms, relying on some assumptions. In this work, we consider a hyperGNN model with a memory-efficient self-attention mechanism, which we call HAT. HAT achieves efficiency by applying self-attention to aggregation at hyperedges and aggregation at nodes separately. HAT can be implemented using our framework with affordable memory overhead, as shown in Section 4.3.

3 PROPOSED FRAMEWORK: CORE MODULES

Our framework HyFER (**Hy**pergraph Neural Network **F**ramework for **E**fficiency and **R**eproducibility) consists of three modules, which are executed consecutively, as shown in Figure 2. In this section, we describe each module, or equivalently each step, with required inputs and outputs. Each module can be customized to suit different purposes, independently without affecting the other modules.

3.1 Data Module

In the data module, input data are transformed in the required format. One of two required outcomes is *node features*. As discussed in Section 2.1, in hyperGNNs, node features are passed from each node to its neighbors. One may use external nodes features if they are available. Otherwise, nodes features can be generated using any transductive node embedding method, such as node2vec [8]. Once d features are given or generated, our framework requires the them to be stored in the form of a real-valued matrix $H \in \mathbb{R}^{|V|\times d}$, where each i-th row corresponds to the feature vector of the i-th node.

The other required output is a hypergraph stored in a data structure that efficiently supports the following two operations in parallel: (1) passing information at each node to hyperedges containing the node, and (2) passing information at each hyperedge to nodes contained in it. As discussed in Section 2, typical hyperGNNs are based on these two types of interaction. Our framework requires the *DGLGraph* format [18], which is a data structure satisfying the above conditions. Specifically, a hypergraph $G = (V, E)$ is represented as a bipartite graph $G' = (V', E')$, where $V' = V \cup E$ and $E' = \{(v, e) \in V \times E : v \in e\}$, and stored in the *DGLGraph* format. For convenience, our framework provides an example of converting a hypergraph in the form of a list of hyperedges, each of which is a list of node ids, into a *DGLGraph*.

- **Input:** a hypergraph dataset,
- **Outputs:** (1) node features H in the form of a matrix, and (2) a hypergraph G in the form of a *DGLGraph*.

3.2 Model Module

This step is where nodes and hyperedges pass messages to each other, using Eq. (1), and as a result, node embeddings are obtained. Recall that, in the previous step, the hypergraph G is stored in a data structure that supports such message passing efficiently. Users need to define the neighborhood aggregation function F_w in Eq. (1), or they need to choose one among those of HGNN, HNHN, and HAT (see Section 2.2), which our framework provides. Output node embeddings need to be stored in the form of a real-valued matrix $H' \in \mathbb{R}^{|V|\times d'}$, where the embedding dimension d' is a hyperparameter. Each i-th row of H' corresponds to the embedding of the i-th node in G.

- **Input:** (1) node features H in the form of a matrix, and (2) a hypergraph G in the form of a *DGLGraph*,

Algorithm 1: HyFER: Node Classification

Input : (1) hypergraph $G = (V, E)$, (2) node features $H \in \mathbb{R}^{|V| \times d}$, (3) training epochs T, (4) train_idx, (5) train_labels, (6) test_idx

Output : test labels

```
/* training */
pred_labels ← empty list with length |train_idx|
for t = 1, ..., T do
    for i = 1, ..., |train_idx| do
        H' ← HyperGNN(G = (V, E), H)
        pred_labels[i] ← Classifier(H'[train_idx[i]])
        loss = Loss(pred_label[i], train_label[i])
        update HyperGNN weights with loss
/* test */
test_labels ← empty list with length N − |train_idx|
H' ← HyperGNN(G = (V, E), H)
for i = 1, ..., |test_idx| do
    test_labels[i] ← Classifier(H'[test_idx[i]])
return test_labels
```

- **Outputs:** node embeddings H' in the form of a matrix.

3.3 Task Module

In this module, node embeddings from the previous module are applied to a downstream task that users have in mind. Users need to define (1) task-specific functions that the node embeddings are fed into, and (2) a loss function that they aim to minimize. The outputs of the task-specific functions are usually considered as the final outputs, and the loss function typically depends on them.

- **Input:** node embeddings H' in the form of a matrix,
- **Outputs:** task-specific outputs.

4 NODE CLASSIFICATION

In this section, we describe how our framework is used for node classification, with some experimental results.

4.1 Problem Definition

We consider a semi-supervised setting, as formalized in Problem 1.

Problem 1 (Node Classification). *Given (1) a hypergraph $G = (V, E)$, (2) node features $H \in \mathbb{R}^{|V| \times d}$, and (3) the labels of a small fraction of nodes, the goal is to infer the labels of other nodes.*

4.2 Implementation on HyFER

We describe how we use HyFER to apply three hyperGNN models to Problem 1 for the comparison in the next subsection. We define four Data modules for four real-world datasets (i.e., Citeseer, Cora, Pubmed, and DBLP), respectively. All the datasets, which are described in Appendix A, have external node features, and they are passed through the Data modules. We also define three Model modules where we implement the neighborhood aggregation schemes in HGNN, HNHN, and HAT (see Section 2.2), respectively. Lastly, we define a Task module, where task-specific functions and a loss function need to be specified. We use a fully connected layer followed by the softmax function as the former and the cross entropy function as the latter. In addition to defining the three types of modules, as an additional functionality in the Data modules, we need to define how to split nodes with labels into training, validation, and test sets. We define the functionality so that 20 randomly chosen nodes with each label are used for training, and the same number of randomly chosen nodes with each label are used for validation. The other nodes are used for testing. We provide these 4 Data modules, 3 Model modules, and 1 Task module as a part of HyFER, and all the modules and the data split functionality can be replaced.

The process of HyFER for node classification is described in Algorithm 1, where (1) **HyperGNN** is either HGNN, HNHN, or HAT, (2) **Classifier** is a fully connected layer followed by the softmax function, and (3) **Loss** is the cross entropy function. Due to space limits, we discuss the validation stage separately in Appendix B.

Table 1: Inference time (in seconds).

		Citeseer	Cora	Pubmed	DBLP
HGNN	Open Source	1.901 ± 0.092	1.602 ± 0.052	O.O.M	O.O.M
	HyFER	**0.119 ± 0.026**	**0.096 ± 0.004**	**0.284 ± 0.004**	**0.206 ± 0.008**
HNHN	Open Source	**0.002 ± 0.0004**	**0.002 ± 0.002**	0.176 ± 0.017	0.179 ± 0.016
	HyFER	0.066 ± 0.027	0.057 ± 0.006	**0.135 ± 0.00**	**0.112 ± 0.008**
HAT	Open Source	N/A	N/A	N/A	N/A
	HyFER	0.071 ± 0.026	0.059 ± 0.003	0.143 ± 0.003	0.115 ± 0.009

Table 2: GPU memory usage (in MBs).

		Citeseer	Cora	Pubmed	DBLP
HGNN	Open Source	1995	1595	O.O.M.	O.O.M.
	HyFER	**1141**	**1127**	**1245**	**2367**
HNHN	Open Source	**1135**	**1091**	2042	**2507**
	HyFER	1209	1185	**1941**	3477
HAT	Open Source	N/A	N/A	N/A	N/A
	HyFER	1239	1229	1919	4539

Table 3: Node classification accuracy.

	Citeseer	Cora	Pubmed	DBLP
HGNN	**0.678 ± 0.028**	**0.710 ± 0.027**	**0.778 ± 0.022**	**0.884 ± 0.002**
HNHN	0.636 ± 0.029	0.660 ± 0.026	0.771 ± 0.023	0.859 ± 0.005
HAT	0.644 ± 0.029	0.607 ± 0.044	0.764 ± 0.012	0.868 ± 0.002

4.3 Experimental Results

Using the implementations described in the previous subsection, we compare HGNN, HNHN, and HAT, in terms of inference time, GPU memory usage, and node classification accuracy. Additionally, we evaluate the efficiency of HyFER by comparing the implementations on HyFER with open-sourced ones. We repeated all experiments 20 times; and means and standard deviations are reported.

Inference Time and Memory Usage. As seen in Tables 1 and 2, HNHN and HAT were about two times faster than HGNN, while HGNN was most memory efficient followed by HNHN and then HAT. The implementation of HGNN on HyFER was significantly faster and memory-efficient than the open-sourced implementation of HGNN. The implementation of HNHN on HyFER was significantly faster than the open-sourced one in large datasets (i.e., Pubmed and DBLP), while in overall, they were comparable in terms of speed and memory efficiency.

Node Classification Accuracy. As seen in Table 3, despite its simplest functional form (see Section 2.2), HGNN was consistently most accurate for the node classification task. There was no clear winner between HNHN and HAT.

Table 4: Hyperedge prediction accuracy.

	Dataset	Citeseer		Cora		Pubmed		DBLP	
Model	Aggregator	Acc (%)	AUROC	Acc (%)	AUROC	Acc (%)	AUROC	Acc (%)	AUROC
HGNN	mean	0.534 ± 0.053	0.538 ± 0.051	0.504 ± 0.015	0.509 ± 0.013	0.578 ± 0.039	0.580 ± 0.038	0.627 ± 0.017	0.627 ± 0.017
	attention	**0.573 ± 0.056**	**0.571 ± 0.058**	0.497 ± 0.019	0.498 ± 0.019	0.565 ± 0.035	0.565 ± 0.036	**0.687 ± 0.049**	**0.686 ± 0.050**
	maxmin	0.556 ± 0.062	0.556 ± 0.060	**0.515 ± 0.031**	**0.521 ± 0.029**	**0.600 ± 0.032**	**0.602 ± 0.032**	0.648 ± 0.043	0.647 ± 0.043
	SAGNN	0.539 ± 0.044	0.539 ± 0.043	0.507 ± 0.020	0.509 ± 0.018	0.565 ± 0.052	0.564 ± 0.053	0.648 ± 0.034	0.648 ± 0.034
HNHN	mean	0.521 ± 0.020	0.521 ± 0.020	0.501 ± 0.011	0.506 ± 0.011	0.562 ± 0.049	0.562 ± 0.049	0.590 ± 0.025	0.589 ± 0.025
	attention	0.507 ± 0.015	0.507 ± 0.015	0.503 ± 0.018	0.506 ± 0.016	0.555 ± 0.054	0.555 ± 0.054	0.630 ± 0.051	0.630 ± 0.052
	maxmin	0.529 ± 0.056	0.532 ± 0.054	0.513 ± 0.033	0.516 ± 0.033	0.559 ± 0.031	0.561 ± 0.031	0.623 ± 0.018	0.623 ± 0.018
	SAGNN	0.539 ± 0.042	0.541 ± 0.043	0.507 ± 0.013	0.506 ± 0.015	0.498 ± 0.008	0.496 ± 0.008	0.651 ± 0.046	0.651 ± 0.046
HAT	mean	0.547 ± 0.033	0.546 ± 0.032	0.499 ± 0.012	0.504 ± 0.0124	0.542 ± 0.042	0.541 ± 0.042	0.652 ± 0.033	0.652 ± 0.033
	attention	0.569 ± 0.039	0.569 ± 0.038	0.503 ± 0.026	0.508 ± 0.025	0.584 ± 0.051	0.584 ± 0.051	0.668 ± 0.036	0.669 ± 0.036
	maxmin	0.534 ± 0.053	0.539 ± 0.051	0.505 ± 0.020	0.510 ± 0.018	0.529 ± 0.031	0.532 ± 0.030	0.642 ± 0.045	0.643 ± 0.045
	SAGNN	0.554 ± 0.032	0.553 ± 0.032	0.506 ± 0.021	0.510 ± 0.019	0.550 ± 0.062	0.550 ± 0.063	0.663 ± 0.056	0.663 ± 0.056

Algorithm 2: HyFER: Hyperedge Prediction

Input : (1) hypergraph $G = (V, E)$, (2) node features $H \in \mathbb{R}^{|V| \times d}$, (3) training epochs T, (4) ground hyperedges E', (5) train candidates C_{tr} (6) test candidates C_{ts}

Output: test labels

```
/* training */
pred_labels ← empty list with length |C_tr|
for t = 1, ..., T do
    for c_i ∈ C_tr do
        H' ← HyperGNN(G' = (V, E' ∪ {c_i}), H)
        rep_vec ← Aggregator(H'[c_i])
        pred_labels[i] ← Classifier(rep_vec)
        loss = Loss(pred_labels[i], c_i ∈ E)
        update HyperGNN weights with loss
/* test */
test_labels ← empty list with length |C_ts|
for c_i ∈ C_ts do
    H' ← HyperGNN(G' = (V, E' ∪ {c_i}), H)
    rep_vec ← Aggregator(H'[c_i])
    test_labels[i] ← Classifier(rep_vec)
return test_labels
```

5 HYPEREDGE PREDICTION

In this section, we demonstrate how we use our framework for hyperedge prediction, with some experimental results.

5.1 Problem Definition

Due to a vast number of potential hyperedges, it is prohibitive to compute and rank their likelihood. Thus, as in [22], we formulate hyperedge prediction as the task of classifying real and fake hyperedges, which we call *positive and negative candidates*, based on known hyperedges, which we call *ground hyperedges*.

Problem 2 (Hyperedge Prediction). *Given (1) ground hyperedges $E' \subset E$ in a hypergraph $G = (V, E)$, (2) node features $H \in \mathbb{R}^{|V| \times d}$, and (3) candidates $C \in 2^V$ where $C \cap E' = \emptyset$, the goal is to classify whether each candidate in C belongs to E or not.*

5.2 Implementation on HyFER

We present how we use HyFER to apply three hyperGNNs to Problem 2 for the comparison in the next subsection. As in Section 4, we define four Data modules and three Model modules. For the new task in Definition 2, as additional functionalities in the Data modules, we need to define (1) how to create negative candidates, (2) how to divide hyperedges into ground hyperedges and positive candidates, and (3) how to divide candidates into training, validation, and test sets. We define the functionalities so that we use (1) MNS [14] for negative sampling, (2) random 70/30 ground hyperedges/positive candidates splits, and (3) random 50/50 training/test splits.[1] For negative sampling, HyFER provides three more predefined options: UNS, CNS, and SNS (see Appendix C).

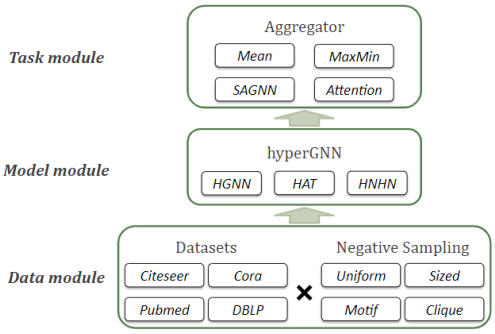

Figure 3: Predefined modules and functionalities provided by HyFER. For details, see Section 2.2 and Appendices C-D.

Then, we define Task modules, where task-specific functions and a loss function need to be specified. As the former, we use one among four aggregators, which aggregates the embeddings of the nodes in each candidate, and a fully connected layer followed by the sigmoid function, together which act as a classifier. As the latter, we use the binary cross entropy loss. For aggregation, HyFER provides four predefined options: Mean, MaxMin, Attention, and SAGNN (see Appendix D). They are all permutation invariant (i.e., the order of input embeddings does not affect the output embedding) and capable of aggregating any number of embeddings.

HyFER processes the modules as described in Algorithm 2, where (1) **HyperGNN** is either HGNN, HNHN, or HAT, (2) **Aggregator** is one among mean, maxmin, attention, and SAGNN, (3) **Classifier** is a fully connected layer followed by the sigmoid function, and (4) **Loss** is the binary cross entropy function.

5.3 Experimental Results

Using the implementations described in the previous subsection, we measured the accuracy and AUROC in Table 4. Despite its simplicity, HGNN was consistently most accurate when it is equipped with a proper aggregator. There was no clear winner between aggregators.

[1] We use hyperparameter values obtained in Section 4 (See Appendix B for details).

6 CONCLUSION

In this work, we present HyFER, an easy-to-use and efficient framework for deep learning on hypergraphs. It is implemented on top of DGL for efficiency, and it is modularized into three parts for easy adaptation to new datasets, models, and tasks. Using HyFER, we could directly compare three hyperGNNs in two tasks on four datasets under the same settings.

Reproducibility: The source code and datasets used in this work are available at https://github.com/ksalsw1996DM/HyFER.

Acknowledgements

This work was supported by National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1C1C1008296) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00075, Artificial Intelligence Graduate School Program (KAIST)).

REFERENCES

[1] Sameer Agarwal, Kristin Branson, and Serge Belongie. 2006. Higher order learning with graphs. In *ICML*.

[2] Ilya Amburg, Nate Veldt, and Austin Benson. 2020. Clustering in graphs and hypergraphs with categorical edge labels. In *WWW*.

[3] Song Bai, Feihu Zhang, and Philip HS Torr. 2021. Hypergraph convolution and hypergraph attention. *Pattern Recognition* 110 (2021), 107637.

[4] Austin R Benson, Rediet Abebe, Michael T Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *PNAS* 115, 48 (2018), E11221–E11230.

[5] Austin R Benson, Ravi Kumar, and Andrew Tomkins. 2018. Sequences of sets. In *KDD*.

[6] Yihe Dong, Will Sawin, and Yoshua Bengio. 2020. HNHN: Hypergraph networks with hyperedge neurons. *arXiv preprint arXiv:2006.12278* (2020).

[7] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. 2019. Hypergraph neural networks. In *AAAI*.

[8] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *KDD*.

[9] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*.

[10] Geon Lee, Minyoung Choe, and Kijung Shin. 2021. How Do Hyperedges Overlap in Real-World Hypergraphs?–Patterns, Measures, and Generators. In *WWW*.

[11] Geon Lee, Jihoon Ko, and Kijung Shin. 2020. Hypergraph Motifs: Concepts, Algorithms, and Discoveries. *PVLDB* 13, 11 (2020).

[12] Pan Li and Olgica Milenkovic. 2017. Inhomogeneous hypergraph clustering with applications. *NeurIPS*.

[13] Zhouhan Lin, Minwei Feng, Cícero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A Structured Self-Attentive Sentence Embedding. In *ICLR*.

[14] Prasanna Patil, Govind Sharma, and M Narasimha Murty. 2020. Negative sampling for hyperlink prediction in networks. In *PAKDD*.

[15] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *TNN* 20, 1 (2008), 61–80.

[16] Julian Shun. 2020. Practical parallel hypergraph algorithms. In *PPoPP*.

[17] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.

[18] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[19] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *arXiv preprint arXiv:1901.00596* (2019).

[20] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks?. In *ICLR*.

[21] Naganand Yadati, Vikram Nitin, Madhav Nimishakavi, Prateek Yadav, Anand Louis, and Partha Talukdar. 2020. NHP: Neural Hypergraph Link Prediction. In *CIKM*.

[22] Se-eun Yoon, Hyungseok Song, Kijung Shin, and Yung Yi. 2020. How Much and When Do We Need Higher-order Information in Hypergraphs? A Case Study on Hyperedge Prediction. In *WWW*.

[23] Ruochi Zhang, Yuesong Zou, and Jian Ma. 2019. Hyper-SAGNN: a self-attention based graph neural network for hypergraphs. *ICLR*.

[24] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2018. Deep Learning on Graphs: A Survey. *arXiv preprint arXiv:1812.04202* (2018).

[25] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2018. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434* (2018).

APPENDIX

A DATASETS

As stated in the main paper, we used four hypergraph datasets that are frequently used. Citeseer, Cora, and Pubmed datasets originate from co-citation networks. Each node is a paper, and each hyperedge consists of a set of papers cited by the same paper. In these datasets, we excluded papers that are cited exactly once, which correspond to isolated nodes. The DBLP dataset originates from a co-authorship dataset. Each node is a paper, and each hyperedge consists of the papers written by the same author. Some statistics of the datasets are provided in Table 7.

Table 5: Hypergraph datasets.

	Citeseer	Cora	Pubmed	DBLP
# of Nodes |V|	3312	2708	19717	43413
# of Hyperedges |E|	1079	1579	7963	22535
Average Size of Hyperedges	3.2	3.0	4.3	4.7
Input Feature Dimension	3703	1433	500	1425
# of Labels	6	7	3	6

B HYPERPARAMETER SEARCH

In Table 6, we provide the hyperparameter search space. We used grid search to find the best combination of hyperparameter values for each model in the node classification tasks. We tested each combination on 10 different splits and different random seeds.

Table 6: Hyperparameter search space.

Hyperparameter	Value	Notes
Batch Size	Full Batch (|V|)	
Learning Rate	0.004	
Optimizer	Adam	
Dropout Rates	[0, 0.3, 0.5]	
# of Layers	[1, 2, 3]	
Dim. Embedding	[50, 100, 200, 400, 800, 64, 128]	
Dim. Query	[64, 128]	Used in HAT
α	[-0.5, -0.4, ... , 0.4 , 0.5]	Used in HNHN
β	[-0.5, -0.4, ... , 0.4 , 0.5]	

C NEGATIVE HYPEREDGE SAMPLING

HyFER provides four predefined schemes for choosing negative hyperedges proposed in [14]. Below, we describe how each hyperedge is created in each scheme.

- **Uniform Negative Sampling (UNS).** It first samples a target size k from a uniform distribution, and then it creates a hyperedge of size k by choosing nodes uniformly at random.
- **Sized Negative Sampling (SNS).** It first computes the size distribution of positive hyperedges. Then, it samples a target size k from the obtained size distribution. Lastly, it generates a hyperedge of size k by choosing nodes uniformly at random.

Table 7: Node classification accuracy.

	Dataset	Citeseer	Cora	Pubmed	DBLP
Model	# Layers	Acc (%)	Acc (%)	Acc (%)	Acc (%)
HGNN	1 Layer	**0.678 ± 0.03**	**0.710 ± 0.03**	0.778 ± 0.02	**0.884 ± 0.00**
	2 Layers	0.666 ± 0.02	0.686 ± 0.03	0.785 ± 0.02	0.877 ± 0.00
	3 Layers	0.640 ± 0.04	0.692 ± 0.03	**0.792 ± 0.01**	0.87 ± 0.00
HNHN	1 Layer	**0.636 ± 0.03**	**0.660 ± 0.03**	**0.771 ± 0.02**	0.859 ± 0.00
	2 Layers	0.606 ± 0.02	0.660 ± 0.04	0.76 ± 0.03	**0.864 ± 0.01**
	3 Layers	0.588 ± 0.03	0.632 ± 0.04	0.755 ± 0.05	0.858 ± 0.01
HAT	1 Layer	**0.644 ± 0.03**	0.607 ± 0.04	0.764 ± 0.01	**0.868 ± 0.00**
	2 Layers	0.610 ± 0.02	**0.618 ± 0.03**	**0.779 ± 0.02**	0.859 ± 0.00
	3 Layers	0.598 ± 0.02	0.62 ± 0.04	0.731 ± 0.09	0.843 ± 0.01

- **Motif Negative Sampling (MNS).** It also samples a target size k from the size distribution of positive hyperedges. Then, it samples a set of k nodes that are connected.
- **Clique Negative Sampling (CNS).** It chooses a positive hyperedge uniformly at random and replaces a node chosen uniformly at random within the positive hyperedge with a node chosen uniformly random outside it.

D AGGREGATION METHODS FOR HYPEREDGE PREDICTION

We describe four aggregators, which are the core functionality in the Task module for hyperedge prediction. As discussed in Section 5, they are for generating hyperedge embeddings, and they have two key characteristics: *size independence* and *permutation invariance*.

- **Mean Aggregator.** Given a set of node embeddings, it outputs the element-wise mean as the output hyperedge embedding.
- **Attention Aggregator.** It uses self-attention to obtain hyperedge embeddings. Given a set of nodes and their embeddings, it calculates the attention score for each node within the set by introducing a trainable key vector. Then, it calculates a hyperedge embedding using both the attention scores and then given node embeddings.
- **SAGNN Aggregator [23].** This is another aggregator that uses attention for hyperedge embedding. Given a set of nodes, it first generates a *dynamic embedding* of each node using self-attention and uses a fully connected layer to obtain a *static embedding* of each node. Then, the aggregator calculates the difference between the two embeddings of each node. Lastly, it averages the results over the given set of nodes.
- **MaxMin Aggregator [21].** Given a set of node embeddings, it outputs the element-wise difference between the maximum and minimum values within the set, emphasizing the diversity of embeddings within the set.