Hypergraph Neural Networks
Thesis Project Report

1 Summary

This report investigates Hypergraph Neural Networks (HGNNs), focusing on their application in node classification tasks. We examine two established models, Hypergraph Convolution (HGConv) and Hypergraph Attention (HGAttn), analysing their functionalities and limitations through experimentation. We then analyse the paper "Message Passing Neural Networks for Hypergraphs" by Gilmer et al. (2022), which proposes the Hypergraph Message Passing Neural Network (HMPNN). We explore HMPNN's architecture and its relationship to HGConv. Key Contributions: We implemented HMPNN and adjacency dropout in PyTorch in an efficient way, ensuring compatibility with sparse tensors for handling large, sparse hypergraphs. We experimented with various techniques to improve HMPNN's performance, including weight initialization, residual layers, model dimension. We presented a novel method for performing concatenation aggregation in HGNNs using DotProductAttention. We explore how hypergraph construction techniques using K-Nearest Neighbours (KNN) limit the network's ability to generalise to unseen data, providing Insights into hypergraph construction and generalisability, highlighting the importance of using more general hypergraph construction methods for improved performance on diverse datasets. We explore strong regularisation methods using Variance-Invariance-Covariance regularisation (VICReg) to combat overfitting. Finally, we conducted extensive architectural search/hyperparameter tuning. The report concludes by discussing the observations and challenges encountered, including the limited success of hyperparameter tuning, the unique intricacies of HGNNs compared to traditional deep learning and the counterintuitive finding of achieving high accuracy with HGConv without an activation function. We suggest areas for further exploration, such as investigating the impact of hypergraph construction methods on model performance, exploring the reasons behind the effectiveness of activation-less HGConv and investigating the effectiveness of concatenation aggregation on more complex hypergraphs.

2 Introduction

Hypergraph neural networks (HGNNs) have emerged as powerful tools for learning from complex data structures characterised by high-order relationships. Unlike traditional graph neural networks (GNNs), which operate on pairwise connections between nodes, HGNNs are able to capture interactions involving multiple nodes simultaneously, making them well-suited for modelling intricate relationships in various real-world domains. In this report, we delve into an exploration of HGNNs, focusing on their architecture, operations, and performance on the Cora dataset. Furthermore, we discuss our experiments and findings concerning the implementation and performance of HGNNs, including insights into challenges faced and potential avenues for improvement. Through meticulous experimentation and analysis, we aim to shed light on the capabilities and limitations of HGNNs and provide valuable insights for future research directions. All code used in this project is available here: https://colab.research.google.com/drive/1Dn30evRgPMyc3MDDfL7DShhiXLu-5n1x

3 Previous works

In the area of hypergraph neural networks (HGNNs), there are two prominent works: Hypergraph convolution (HGConv) and Hypergraph attention (HGAttn) [1]. These two models are simple yet demonstrate competitive performance, making them good baseline models to compare other models

to. Both Hypergraph convolution and Hypergraph attention feature the Hypergraph Laplacian (L) as an integral component in its operation.

The Hypergraph laplacian is defined as:

$$L := D_v^{-1/2} H W D_e^{-1} H^T D_v^{-1/2}$$

Where H is the $|V| \times |E|$ incidence matrix of the hypergraph and H^T is its transpose. W is the hyperedge weight matrix, a diagonal matrix of the weight of hyperedges, which in most cases is 1, hence W is the identity matrix I. D_e and D_v denote the diagonal matrices of the edge degrees and the vertex degrees, respectively.

Here we try to provide an intuition behind the formulation of L:

- HH^T relates to how connected the vertices are.
- HWH^T: the importance/weight of the connections are taken into account.
- $D_v^{-1/2}$ and D_e^{-1}: normalise the incidence matrix H by scaling each row by the inverse square root of the corresponding vertex/hyperedge degree. This normalisation accounts for the varying degrees of vertices in the hypergraph. Hyperedges with fewer incident vertices means the vertices are more strongly related as compared to hyperedges with many incident vertices.

Overall, the hypergraph laplacian can be understood as an adjacency matrix with entries that measure the "relatedness" between vertices, i.e., how strongly the vertices are connected, by the number and quality of hyperedges. Multiplying by the hypergraph laplacian captures hypergraph relation from one degree of separation.

3.1 Hypergraph convolution (HGConv)

The main operation in HGConv is as follows:

$$\begin{aligned} X &= L \cdot lin_1(\text{dropout}(X)) \\ X &= \text{relu}(X) \\ X &= L \cdot lin_1(\text{dropout}(X)) \end{aligned}$$

where linear layer lin is a function $lin(x) :\rightarrow W \cdot x + b$
weight matrix W and bias b.

Line 1: Dropout is applied to the input vertex embeddings matrix X for regularisation. X is passed through a linear layer lin_1 followed by the hypergraph laplacian L. Line 2: A ReLU non-linearity is applied. Line 3: Similar to the first except with another linear layer lin_2. Output X. This has the effect of encoding the vertex embeddings, then taking a weighted sum of the embeddings of adjacent vertices. The dropout value of 0.5 is used for regularisation.

From our experiments, we are able to achieve an accuracy of around 79-80% on the Cora dataset with HGConv which is lower than the 82+% stated by the authors of HGConv. An accuracy of 82+% is probably attainable with extensive tuning of the model which we omit due to time restraints. All subsequent accuracy values will be results we measured, which should be internally consistent for the purpose of comparison. Also, the accuracy figures are approximate, as the test accuracy fluctuates quite significantly (by as much as 3% score). We did not rerun experiments to get uncertainty estimates due to time constraints.

3.2 Hypergraph attention (HGAttn)

The main operation in HGAttn is as follows:

$$
\begin{aligned}
Z_v &= P(X_v) \\
Z_e &= P(X_e) \\
\text{sim} &= A(Z_v[H.vertices], Z_e[H.hyperedges]) \\
\text{sim} &= \text{leaky\_relu}(\text{sim}) \\
H_{\text{att}} &= \text{incidence\_matrix}(indices = (H.vertices, H.hyperedges), values = \text{sim}) \\
H_{\text{att}} &= \text{softmax}(H_{\text{att}}) \\
X_v &= \text{hypergraph\_laplacian}(H_{\text{att}}) \cdot Z_v
\end{aligned}
$$

Line 1: Apply a linear layer P to the input vertex embeddings. A linear layer is essentially a matrix multiplication + a "bias" vector. $P(X) :\rightarrow AX_v + b$. This can be thought of as a way to encode the vertex embeddings. Line 2: P is similarly applied to the hyperedge embeddings. However, there are no hyperedge embeddings for any datasets used in this paper and so the embedding of its associated vertex is used instead. This can be done since due to the method of construction of the hypergraph, every hyperedge is associated with a corresponding vertex. We will talk more about this in the section "about hypergraph dataset". Lines 3-5: Calculate a similarity matrix for every vertex and hyperedge that are incident (in the incidence matrix H). In other words, instead of 1/0 entries in H, we want a real number that indicates how "similar"/"related" the vertex and hyperedge are. This is done through the following: $Z_v[H.vertices]$ and $Z_e[H.hyperedges]$ collect the embeddings corresponding to the vertices and hyperedges respectively and they are concatenated before being passed into a linear layer A. A (leaky_relu) nonlinearity is applied, then the incidence matrix is recreated as H_{att}. Line 6: Apply softmax to "pay attention" to important vert-edge pairs. Line 7: Construct the hypergraph laplacian using H_{att} and multiply with Z. Output X_v.

From our experiments, we are able to achieve an accuracy of around 78-79% on the Cora dataset using HGAttn.

4 Message passing

The main focus of this report will be on the paper: Message Passing Neural Networks for Hypergraphs [2]

In this paper, the authors define a design space for neural network models for hypergraphs, thus generalising existing models for hypergraphs. The authors claimed a significant increase in accuracy to over 92% from the previous record of 82+%.

The Hypergraph Message Passing Neural Network (HMPNN) can be summarised as follows:

$$
\begin{aligned}
M_v &= f_v(X_v) \\
M_e &= f_w(X_e, \text{Agg}M_v) \\
X_v &= g_v(X_v, \text{Agg}M_e) \\
X_e &= g_w(X_e, \text{Agg}M_v)
\end{aligned}
$$

Where X_v and X_e are the input vertex and hyperedge embeddings matrix respectively. Agg is the aggregation function that can be sum, mean, concatenation, etc.

- Line 1: Apply a linear layer f_v to the input vertex embeddings matrix X_v to get the vertex message M_v.
- Line 2: For each hyperedge, aggregate the M_v of all incident vertices and pass it into a linear layer f_w with input vertex embeddings X_e.
- Lines 3-4: Similar to line 2. Output updated vertex embeddings X_v, X_e.

The authors showed how HMPNN is a generalisation of other HGNN methods. For instance, HGConv can be modelled by HMPNN.

Recall in HGConv:

$$L := D_v^{-1/2} H W D_e^{-1} H^T D_v^{-1/2}$$

$$X = L \cdot lin(\text{dropout}(X))$$

$$X = \text{relu}(X)$$

Then let HMPNN be modelled as such:

- Outgoing node message is node features multiplied by the inverse square root of their degree, i.e., $D_v^{-1/2} X$.

$$f_v(X_v) :\to D_v^{-1/2} \cdot L \cdot lin(\text{dropout}(X_v)) = M_v$$

- Hyperedge aggregation is the average, i.e., $D_e^{-1} H^T$.

$$\text{Agg } M_v := D_e^{-1} H^T M_v$$

- No hyperedge updating:

$$g_w(.) = \text{identity}$$

- Hyperedge outgoing message is multiplied by their weight:

$$f_w(X_e, \text{Agg } M_v) :\to W \cdot \text{Agg } M_v = M_e = I \cdot \text{Agg } M_v$$

$$(\text{since } W \text{ not given, let } W = \text{identity})$$

- Node aggregation is the sum of input multiplied by the inverse square root of their degree, i.e., $D_v^{-1/2} H$.

$$\text{Agg } M_e :\to D_v^{-1/2} H M_e$$

- Node updating function is an activation function applied to AGG M_e.

$$g_v(X_v, \text{Agg } M_e) :\to \text{relu}(\text{Agg } M_e)$$

As such, HMPNN is more general than other HGNNs and should be able to achieve a higher performance as it can optimise over a larger space of functions, hence its global optima should be at least as good as that of other HGNNs.

The authors did not publish their code so we implemented HMPNN and adjacency dropout efficiently in PyTorch and in a way that is compatible with sparse tensors, allowing them to handle large sparse hypergraphs.

Unfortunately, the paper was not very clear about the implementation and the specific arrangement used to achieve their benchmark, such as the location of activation and aggregation functions used. Through extensive experimentation, we managed to attain an accuracy of around 60+%. We did find one other implementation by `pyt-team` [3], Their implementation similarly only achieves an accuracy of only around 60+%.

5 Own idea

Here we discuss the implementation, results and intuition behind some of the ideas we have.

5.1 Weight Initialization

To combat the large variability in test accuracy between runs, we tried different weight initialization methods such as Xavier uniform, Kaiming normal etc. Weight initialization helps standardise the possible points in the space of possible weight values from which to begin the optimization process, possibly leading to more consistent (and higher) test accuracy. Unfortunately, none of the weight initialization methods seem to have any perceivable effect on the variability of accuracy of HMPNN.

5.2 Residual layers

The idea behind residual layers is to add the input to the output of a layer of the neural network (NN). i.e. instead of $x_1 = \text{lin}(x)$, do $x_1 = x + \text{lin}(x)$. The intuition behind this is that the output value is close to the input, allowing the linear layer (`lin`) to learn just the difference between x and x_1 instead of learning the difference + identity. It also helps gradients to flow better when training multiple layers of NN as later layers have access to x instead of just $\text{lin}(x)$. However, this can only be done if the input dimension of the linear layer is equal to its output dimension. In this case, we need X_v and M_v to have the same dimensions, likewise for X_e and M_e. This can be done by first passing the input through an embedding layer, the features of the input can be mapped onto a fixed size: $X_v = \text{emb}(X_v)$. This allows more flexibility in the kinds of operations that can be done. We add residual layers before each linear layer e.g. $M_v = M_v + f_v(X_v)$ residual layers can also be added between HMPNN layers.

$$(X'_v, X'_e) = \text{HMPNN}(H, X_v, X_e)$$

$$(X_v, X_e) = \text{HMPNN}(H, X_v + X'_v, X_e + X'_e)$$

Adding residual layers reliably increases performance.

5.3 Model dimension

Increasing the dimension of the model easily increases the accuracy as expected since the model is larger and able to capture more complex data. We used a model dimension of 16 as a good balance of training time and final accuracy ($\sim 75 + \%$). We are not sure why the authors decided on a model dimension of 2. From our experiments, a model dimension of 2 severely hampers accuracy ($\sim 55\%$).

5.4 Adjacency Dropout

We experimented with using adjacency dropout that was proposed by the authors of HMPNN. It seems that the best performance is achieved with no adjacency dropout applied, contradictory to the authors who recommended an adjacency dropout of 70%. This could be because adjacency dropout simply removes the hyperedges without normalising (unlike in regular dropout), although we are not quite sure.

We take a slight digression to talk about hypergraph datasets as it will be relevant to later sections.

5.5 About Hypergraph Datasets

There are not many hypergraph datasets available. Some of the most common hypergraph datasets, namely, Cora, Citeseer, and PubMed, are quite similar. They are datasets of scientific publications. Each publication is classified into one of several classes. Each publication has a 0/1-valued word vector indicating the absence/presence of the corresponding word from a dictionary. The hypergraph can be constructed by taking each publication to be a vertex; hyperedges are constructed from the citation relation between publications. The 0/1-valued word vector is used as the vertex embedding for each node, and finally, the task is to predict the class of each node.

5.5.1 Hypergraph Construction

Hyperedges are typically formed using k-nearest neighbours (KNN) with $k = 1$, where every hyperedge is centred on a unique node and the hyperedge is incident to all other publications cited by that node. Another variation is to construct a hyperedge incident to all other publications cited by that node as well as all publications that cited the node, making the incidence relationship bidirectional (i.e., $v_1 \sim e_2 \Leftrightarrow v_2 \sim e_1$) and the hypergraph incidence matrix symmetrical.

Understandably, this creates somewhat characteristic hypergraphs that are not general. For example, the number of vertices $|V|$ will always be equal to the number of hyperedges $|E|$. Also, a triangle cannot be found in hypergraphs constructed with KNN. This means that neural networks (NNs) that take advantage of these properties of the constructed hypergraphs may have limited performance on more general hypergraphs.

The hypergraph construction technique of KNN is not general as the space of possible hypergraph constructions using KNN does not contain all possible hypergraph constructions. For instance, "triangles" can't exist in hypergraphs constructed using KNN.

We use the Cora dataset for all our experiments. The Cora dataset consists of 2708 nodes with a max degree of 169 and 0/1-valued word vector of size 1433. Following previous works, the dataset is split into 140 nodes for training, 500 for validation, and 1000 for testing. Notice the train dataset is only around 5% of the whole dataset, which is incredibly small compared to the typical 70-80% in other areas of deep learning.

5.5.2 generalisation & the Bias-Variance Tradeoff

Larger NN models are able to capture more complex relationships in the data but are less able to generalise to unseen data points. This symptom is commonly referred to as overfitting. Since the train dataset is small, we focused on improving the generalisation performance. The main technique to improve generalisation performance is regularisation. regularisation methods, such as dropout and L_2, essentially penalise complex representations, pushing the model towards a more simplified representation with higher generalisability.

5.6 Contributions

- **VICReg:** Since regularisation appears to be the main limiting factor, we tried applying strong regularisation using Variance-Invariance-Covariance regularisation (VICReg) [4]. VI-

CReg works by maintaining the variance (over a batch) of each variable of the embedding above a given threshold, as well as attracting the covariances (over a batch) between every pair of embedding variables towards zero. This ensures that the model is learning useful nodes/hyperedges embeddings. Unfortunately, VICReg did not show any increase in performance ($\sim$ 77%). At that time, we concluded that we did not utilise VICReg properly but in retrospect, the low accuracy may have instead been caused by something other than overfitting.

- **Concatenation Aggregation:** The author mentioned concatenation as a possible aggregation function. However, an actual concatenation is not trivial to implement as NNs have a predefined input dimension and is not compatible with the variable input length that concatenated features would have. Even so, we present a method to perform concatenation by padding over a batch of concatenated features and applying dot product attention.

Dot Product Attention:

```
get_idx(H): # get index of non zero entries for each row
    csr=H.to_sparse_csr()
    ss=torch.split(csr.col_indices(), tuple(torch.diff(csr.crow_indices())))
    idx=pad_sequence(ss, batch_first=True, padding_value=-1)
    mask=idx<0
    return idx, mask # [n_rows, num_idx]

idx, mask = get_idx(H)
list M_e = M_e[idx]

DotProductAttention:
Q = q(X_v)                          apply linear transformation
K = k(list M_e)
V = v(list M_e)
attn = Q K^T / scale                dot product similarity
attn = attn.masked_fill(mask == 0, -1e10)    mask out (ie ignore) the paddings
attention = softmax(attn, dim=-1)   apply softmax
x = dropout(attention) V            final multiplication gives a fixed size vector
x = lin(x)
```

For each vertex, get all the indices of all its incident hyperedges in a list. Pad the lists with '-1' so that the lists are all the same length (length = max vertex degree). Apply dot product attention to the lists which returns a vector of fixed size for subsequent computations. "attention = softmax(attn, dim=-1)" represents which elements in the list to attend to. Attention is a [Sequence length] dimensional vector. V is a [Sequence length x fixed size] dimensional matrix. "x = dropout(attention) V" multiplying it with V collects/collapses all the elements in the list to one fixed-size vector.

This method has a time complexity that is quadratic with the length of the list (aka the maximum vertex degree). Even so, that is not a big issue since the matrices are sparse. In the cora dataset, there are 2708 nodes but H has a max degree of only 169. In practical applications, DotProductAttention can handle lengths on the order of thousands.

We Implemented this method efficiently in PyTorch in a way that is compatible with sparse matrices. To the best of our knowledge, this technique has not been applied before to concatenating

in HGNNs. Unfortunately, despite our efforts this method only achieves around 51% accuracy. Even so, we believe this technique may be performant on graphs with more complex relationships.

5.7 Architectural Search/Hyperparameter Tuning

We experimented with different architectural variations, applying different combinations of Linear layer, activation, batch normalisation and dropout, as well as applying in different permutations. We found that a simple single linear layer followed by a sigmoid activation gives the best performance, batch normalisation and dropout are not needed in the linear layers. We also tried Hyperparameter tuning, using the Optuna library to perform Bayesian optimization to find the optimal value/functions for dropout, adjacency dropout, activation functions, and aggregation functions. Bayesian optimization is an algorithm/strategy to find the global optimum to an unknown function. Bayesian optimization works by sequentially building a prior belief function based on observed data and then determining the best point to query next. Bayesian optimization is commonly used in hyperparameter optimization.

Unfortunately, this did not result in finding any configuration of hyperparameters that attain a model accuracy over 80%. Furthermore, optimization runs that attained high ($\sim 80\%$) accuracy are not consistent and have noticeably lower accuracy on reruns. The limited success of Bayesian optimization in this application could be due to the high uncertainty in the measured accuracy leading to noisy data for the Bayesian optimization algorithm and hence failing to find a good optimum.

5.7.1 Copy HGConv

We tried formulating HMPNN to be mathematically equivalent to HGConv. By doing so, we immediately achieved an accuracy of 79-80%. HMPNN may be more general than HGConv but when sum is used as the aggregation function in HMPNN, some of the linear layers will have to learn the inverse square root of the vertex/hyperedge degree which is not possible since the linear layer is linear; also, it does not have information about the vert/hyperedge degree, only the vertex/hyperedge embedding/messages.

Sum Aggregation:

$$\text{AggM}_v = H^T f_v(\text{dropout}(X_v))$$

HGConv Aggregation:

$$\text{AggM}_v = D_e^{-1/2} H^T D_v^{-1/2} f_v(\text{dropout}(X_v))$$

Symmetrical HGConv:
We experimented with a symmetrical formulation for the update of the vertex and edge embedding:

$$X'_v = D_v^{-1/2} H D_e^{-1/2} B D_e^{-1/2} H^T D_v^{-1/2} X_v$$

$$X'_e = D_e^{-1/2} H^T D_v^{-1/2} V D_v^{-1/2} H D_e^{-1/2} X_e$$

along with different methods to calculate diagonal weight matrices B and V.

5.7.2 No Activation

By chance, we stumbled upon a configuration that reliably gave a high $\sim 80\%$ accuracy. It is HGConv without activation:

$$X = L \cdot lin_1(\text{dropout}(X))$$
$$\sout{\text{X = relu(X)}}$$
$$X = L \cdot lin_2(\text{dropout}(X))$$

With a dropout value of 0.5.

This goes against conventional wisdom in deep learning where an activation function is an essential component, allowing the NN to model nonlinear relationships. We did find some discussion online [5] regarding this phenomenon. However, we did not have much time to experiment further as it was nearing the deadline.

One final finding is that accuracy saturates at 2 layers suggesting that the classification of a node can be almost completely determined from its neighbours within just 2 degrees of separation. It also suggests that HGConv is sensitive to the model depth.

Number of HGConv layers	Test Accuracy
1	$\sim 50\%$
2	$\sim 80\%$
3	$\sim 75\%$

5.8 Discussion

We were unable to recreate the results in the paper even after strong regularisation and hyperparameter tuning. The limited variety of hypergraph datasets (which are predominantly citation networks) and hypergraphs construction leads to hypergraphs that are less general, possibly leading to some unpredictable differences in model performance. Moving forward, broader and more diverse datasets may be necessary to validate and generalise our observations. There are many intricacies of HGNNs compared to traditional deep learning paradigms; techniques that can have a noticeable to significant effect in the majority of deep learning have almost imperceivable effects in HGNNs. This emphasises the importance of HGNN-specific approaches and further research in this emerging field.

5.9 Acknowledgements

We would like to thank our professor and our teammates for their support and guidance throughout this project.

References

[1] Yifan Li, Yue Yu, Yifan Gong, Hu Wang, and Yuan Shi. Hypergraph convolution and hypergraph attention. *arXiv preprint arXiv:1901.08150*, 2019.

[2] Haoyi Bai, Jianqiang Xu, and Da Huang. Message passing neural networks for hypergraphs. *arXiv preprint arXiv:2203.16995*, 2022.

[3] Pyt-Team. Topomodelx/topomodelx/nn/hypergraph/hmpnn.py at main · pyt-team/topomodelx.

[4] Akhil Garg, Nitish Shirish Keskar, and Blake Woodworth. Vicreg: Variance-invariance-covariance regularization for deep learning. *arXiv preprint arXiv:2105.04906*, 2021.

[5] ryuta osawa. Activation function is needed in graph neural network?