Supernode Graph Neural Networks

Samuele Angheben, 240268

Steve Azzolin, Francesco Ferrini

Keywords: Graph neural network, Concepts, Supernodes

1 Introduction

This project aims to implement and analyze the Supernode Graph Neural Network, which consist of a new preprocessing operation and consequently custom graph neural network that tries to increase the expressive power of naive graph neural networks.
The idea is based on *concepts* that are pattern that can be found in the graph, once a list of concepts is selected we transform the original graph adding for each concept realization a supernode that is connected to the proper original nodes.
This technique is between high order GNN and naive GNN, the intention is to give to the message passing scheme more power with more node visibility and use learnable convolutional layer to boost naive GNN performance.

2 Preliminaries

In this section, we introduce the definition of concepts and supernodes in a graph and then briefly recall the topic of graph classification with graph neural networks and some basics theoretical results.

2.1 Concepts and Supernodes

With Supernode Graph Neural Networks we want to enhance the expressive power of GNN by adding additional nodes to the graph (supernodes) that represent the manifestation of patterns (concepts) within the graph and so are connected to the original graph's nodes that correspond to the particular realization.

Definition (Concept). A concept C is a pattern that can be found in a graph.
Given a graph $G = (V, E)$ and a concept C we can extract each realization R_{C_i} of C in G as a list of subsets of nodes:

$$R_C = [R_{C_1}, \ldots, R_{C_n}] \qquad where\ R_{C_i} \subseteq V$$

Example: `max_cliques`, `cycle_basis`, `line_paths`, `star`

Once we have defined our concepts and we can extract them from the graph, we can transform the original graph to add the supernodes in three different ways:

- homogeneous transformation:
  each supernodes will be of the same type of the original nodes of the graph.
- heterogeneous transformation:
  all the supernodes will be of the same type that is different from the type of the nodes of the original graph.
- heterogeneous multi transformation:
  for each concepts there are different type of supernodes and each of them are different from the type of the nodes of the original graph.

More formally we can define the transformation as follows:

Transformation (Supernodes homogeneous). Given a graph $G = (V, E)$ and a list of concepts $L = [C^1, C^2, \dots, C^n]$ we can create a list R_L that contains all the realization:

$$R_L = [\ \forall C^j \in L.\ \forall R_{C^j_i}.\ R_{C^j_i}\ ]$$

Then transform the original graph:

$$G' = (V + S, E + E_S).$$

where $S = \{S_k\}$ and $E_S = \{(S_k, n_{S_k})\}$ with $n_{S_k} \in R_L[k]$ for $k = 1, \dots, len(R_L)$.

Transformation (Supernodes heterogeneous). Given a graph $G = (V, E)$ and a list of concepts $L = [C^1, C^2, \dots, C^n]$ we can create a list R_L that contains all the realization:

$$R_L = [\ \forall C^j \in L.\ \forall R_{C^j_i}.\ R_{C^j_i}\ ]$$

Then create an heterogeneous graph from the original homogeneous graph:

$$G' = (H_V, H_E).$$

$$\begin{aligned}
H_V = \quad & \{\text{normal} : V,\ \text{supernodes} : S\} \\
H_E = \quad & \{(\text{normal, orig, normal}) : E \\
& \ (\text{supernodes, toNor, normal}) : E_S \\
& \ (\text{normal, toSup, supernodes}) : flip(E_S) \\
& \ (\text{normal, identity, normal}) : I_V \\
& \ (\text{supernodes, identity, supernodes}) : I_S\}
\end{aligned}$$

where $S = \{S_k\}$ and $E_S = \{(S_k, n_{S_k})\}$ with $n_{S_k} \in R_L[k]$ for $k = 1, \dots, len(R_L)$ and $I_V = \{(n_i, n_i), \forall n_i \in V\}$ and $I_S = \{(n_s, n_s), \forall n_s \in S\}$.

Transformation (Supernodes heterogeneous multi). Given a graph $G = (V, E)$ and a list of concepts $L = [C^1, C^2, \dots, C^n]$ we can create a dict of list R_L that for each concepts contains all the realization of it:

$$D_L = \{name(C^j) : R_{C^j}.\ \forall C^j \in L\}$$

Then create an heterogeneous graph from the original homogeneous graph:

$$G' = (H_V, H_E).$$

$$\begin{aligned}
H_V = \quad & \{\text{normal} : V,\ \text{C}^j\_name : S_{C^j}.\ \forall C^j \in D_L\} \\
H_E = \quad & \{(\text{normal, orig, normal}) : E \\
& \ (C^j\_name, \text{toNor, normal}) : E_{S_{Cj}} \\
& \ (\text{normal, toSup, C}^j\_name) : flip(E_{S_{Cj}}) \\
& \ (\text{normal, identity, normal}) : I_V \\
& \ C^j\_name, \text{identity, C}^j\_name) : I_{S_{Cj}}.\ \forall C^j \in D_L\}
\end{aligned}$$

where $S_{C^j} = \{S_{Ck}\}$ and $E_{S_{Cj}} = \{(S_{Ck}, n_{S_{Ck}})\}$ with $n_{S_{Ck}} \in D_L(C^j)[k]$ for $k = 1, \dots, len(D_L(C^j))$ and $I_V = \{(n_i, n_i), \forall n_i \in V\}$ and $I_{S_{Cj}} = \{(n_s, n_s), \forall n_s \in S_{C^j}\}, \forall C^j \in D_L$.

2.2 Graph classification with GNN

The usual structure of GNN models for graph classification is the following:

1. Node embeddings: a sequence of graph convolutional layers that compute node embeddings.
2. Readout: a function that aggregates node embeddings into a graph embedding.
3. Classifier: a function that takes the graph embedding and computes the output.

We follow this structure for all the models that we implemented, but we added a new step at the beginning to enable a flexible initialization of the supernodes:

0. Supernode init: a graph convolutional layers that compute supernode's initial features.

Depending on the type of transformation that we want to apply to the graph the models will threaten the data in different ways and so the layers will be different.
From the implementation point of view, to perform convolutional operation only on supernodes in homogeneous graphs we used masks on nodes and edges created during the transformation meanwhile in heterogeneous graphs we used the `HeteroConv` from torch_geometric that allow to specify the edge type to operate on, we added the `identity` type edge to keep the nodes at the same value during the update of other type of nodes.

We know recall some common graph convolutional and pooling layers that we will use for our models:
Definition (MessagePassing). Message passing layers follow the form

$$\mathbf{x}_i' = \gamma_{\mathbf{\Theta}} \left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} \phi_{\mathbf{\Theta}} \left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i} \right) \right),$$

where $\bigoplus$ denotes a differentiable, permutation invariant function, e.g., sum, mean, min, max or mul, and $\gamma_{\mathbf{\Theta}}$ and $\phi_{\mathbf{\Theta}}$ denote differentiable functions such as MLPs

MessagePassing (SimpleConv) A simple message passing operator that performs (non-trainable) propagation.

$$\mathbf{x}_i' = \bigoplus_{j \in \mathcal{N}(i)} e_{ji} \cdot \mathbf{x}_j.$$

where $\bigoplus$ defines a custom aggregation scheme (eg: `add`, `sum`, `mean`, `min`, `max`, `mul`)

MessagePassing (GINConv) The graph isomorphism operator from the "How Powerful are Graph Neural Networks?" paper.

$$\mathbf{x}_i' = h_{\mathbf{\Theta}} \left((1 + \epsilon) \cdot \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j \right).$$

here $h_{\mathbf{\Theta}}$ denotes a neural network, .i.e. an MLP.

MessagePassing (GCNConv) The graph convolutional operator from the "Semi-supervised Classification with Graph Convolutional Networks" paper. Its node-wise formulation is given by:

$$\mathbf{x}_i' = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j.$$

With $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$, where $e_{j,i}$ denotes the edge weight from source node j to the target node i.

Aggregation (Global_add_pool) Returns batch-wise graph-level-outputs by averaging node features across the node dimension. For a single graph, its output is computed by

$$\mathbf{r}_i = \sum_{n=1}^{N_i} \mathbf{x}_n.$$

Aggregation (Global_mean_pool) Returns batch-wise graph-level-outputs by averaging node features across the node dimension. For a single graph, its output is computed by

$$\mathbf{r}_i = \frac{1}{N_i} \sum_{n=1}^{N_i} \mathbf{x}_n.$$

2.3 Representational power of GNN

From theoretical results we know that no first order GNN can have a higher representational power than the Weisfeiler-Lehman test of isomorphism.

3 Concepts and Models

In this section we will describe the concepts that we have build and the models types based on the type of transformation.

3.1 Concepts

We have built a list of concepts that we think are useful to represent the structure of a graph, in particular we have focused on the following concepts:

Concepts	
name	description
maxcliques	the cliques with at least 3 nodes
cyclebasis(max_num)	the first max_num biggest cycle basis
linepaths	the linepahts: the chains of nodes that have only 2 edges
k_edge_comp	k-edge-connected component
star2	for each node the 2-neighbours distance
maxlines	find all the nodes with biggest degree and extract for each couple of them the shortest path
minlines	find all the nodes with least degree and extract for each couple of them the shortest path
k_core	group each node with the same k-core value
degree_centrality	group each node with the same degree centrality value
comm_modul	communities based on modularity
maxcliques_cyclebasis	maxcliques + cyclebasis
maxcliques_cyclebasis _star2	maxcliques + cyclebasis + star2
cycb_maxcliq_star2 _minl_maxl	maxcliques + cyclebasis + star2 + minlines + maxlines

For all the transformation we first convert the graph to `networkx`, then extract the specified list of concepts and then create the new graph with supernodes. Since we can specify a list we can either take a single concept or all the combinations possible of them, in particular we tested the combinations at the bottom of the table.

Remark We can't create concepts that are based on randomness, because we are interested in graph classification or graph isomorphism test so we would have to extract the same nodes for isomorphic graphs.

3.1.1 Concept analysis

To perform a first analysis on the dataset that we want to operate on we have created a script that for each concept calculate the mean, variance, max, min of realization. This is useful to have a better understanding of the data and discard concept that are not relevant.
We believe that adding to much supernodes can lead to oversmoothing, to few can be irrelevant. Moreover concepts that are local by construction can have a different behavior than the ones that connects more distant nodes.

3.2 Models type

In our experiments we considered four different type of models:

- normal (type 0):
  models that operates on the original graph.
- supernode homogeneous (type 1):
  models that operates on graphs after the supernode homogeneous transformation.
- supernode heterogeneous (type 2):
  models that operates on graphs after the supernode heterogeneous transformation.
- supernode heterogeneous multi (type 3):
  models that operates on graphs after the supernode heterogeneous multi transformation.

Models that works on heterogeneous graphs are built with `HeteroConv`, this means that they have a different embedding space for each type of node and to update a node embeddings we perform the add aggregation of each space after the message passing. This create the difference between supernode homogeneous and supernode heterogeneous, in the first supernodes and original nodes lives in the same space, for this reason the model can't distinguish original nodes from supernodes. The main difference between supernode heterogeneous and supernode heterogeneous multi is that in the latter we can specify a convolution for each concepts, this means that if we use a learnable layer the network can threat each concept differently.

4 Tree-cycle experiment

For the first experiment we decided to synthesize our dataset, it is composed of graphs with and without cycles, in particular the class of each graph is given by this property.
The reason it to check if applying the supernode preprocessing phase to the graphs with the concept `cycle_basis`, that indeed identify cycles, will increase the performance of the models.

Remark In this case the preprocessing computation alone will be able to correctly classify the dataset, since to add a supernode we need to find the cycles, but the goal is understand if the model will benefit from this preprocessing on the graph to then apply this techniques in more complex settings.

To synthesize this dataset we exploit the fact that a graph without cycles is a tree and therefore we first constructed random trees and then to half of them we added `cycle_level`-times random edges. In this way we can generate this type of dataset quickly and flexibly in respect to number of graphs, graphs size, cycle level and proportions.

Remark We set each node features to [1], in this way we force the model to reason on the structure of the graph.

4.1 Settings and results

Dataset				
type	graph number	proportions	node number	cycle level
Dataset_tree_cycle	10000	0.5	40	10

Models					
model name	type	supernode init	node embeddings	readout	classifier
GCN	0	-	3 * (GCNConv(32) + relu)	global_add_pool	MLP(3L,32)
GIN	0	-	3 * (GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)
GCNS	1	SimpleConv(Add)	3 * (GCNConv(32) + relu)	global_add_pool	MLP(3L,32)
GINS	1	SimpleConv(Add)	3 * (GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)

For training all the models we used as criterion `CrossEntropyLoss` and as optimizer `Adam`, furthermore the dataloader uses a `batch_size` of 60 graphs.

Results			
model	concepts	number of epoch	test accuracy
GCN	cycle_basis	50	0.5000
GCNS	cycle_basis	10	1.0000
GIN	cycle_basis	10	1.0000
GINS	cycle_basis	10	0.9995

Considerations
The results show that the normal GIN model on the original graph is enough to correctly classify the dataset, but we can notice that for the GCN models the supernode preprocessing is necessary to be able to classify the dataset.

5 BREC experiments

For the second experiment to understand the effectiveness of the supernode preprocessing we decided to use the BREC dataset from *Towards Better Evaluation of GNN Expressiveness with BREC Dataset*.
In summary it includes 400 pairs of non-isomorphic graphs with difficulty up to 4-WL-indistinguishable, divided in 4 category: Basic, Regular, Extension, CFI. Furthermore in the paper they tested different models which give us a good measure for comparison.
They also provide a base template code to implement new models, since the task is to distinguish pair of graphs the loss function is the following:

$$L(f, G, H) = Max(0, \frac{f(G) \cdot f(H)}{\mid f(G) \mid\mid f(H) \mid} - \gamma).$$

where the GNN model $f : G \to \mathbb{R}^d$, for our test we used $d = 16$, G and H are two non-isomorphic graphs, and $\gamma = 0$. The loss function aims to promote a cosine similarity value lower than γ, thereby encouraging a greater separation between the two graph embeddings.
As evaluation methods we used their method called Reliable Pairwise Comparison.

5.1 Settings and results

Since each graph of the dataset doesn't have any node features with apply one of the two following transformation:

- constant 1 (type 0)
  Assign to each node the same constant value 1.

- vector type (type 1)
  Assign to each node a vector of the same length of the number of concepts plus one, where each element is 0 except the dimension that represent the concepts that the node represent.

Models

Models original						
model name	f	type	supernode init	node embeddings	readout	classifier
GAT0	0	0	-	4 * (GATConv(32) + relu)	global_add_pool	MLP(3L,32)
GIN	0	0	-	4 * (GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)
Models supernode homogeneous						
GIN_Sadd	0	1	SimpleConv('add')	4 * (normal: GINConv(MLP,32), supernode:SimpleConv('add') + relu)	global_add_pool	MLP(3L,32)
GAT_Sadd	0	1	SimpleConv('add')	4 * (normal: GATConv(32), supernode:SimpleConv('add')+ relu)	global_add_pool	MLP(3L,32)
GIN_SGIN	0	1	SimpleConv('add')	4 * (normal: GINConv(MLP,32), supernode:GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)
GIN_SGIN_noSINIT	0	1	-	4 * (normal: GINConv(MLP,32), supernode:GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)
GIN_SGIN_typef	1	1	-	4 * (normal: GINConv(MLP,32), supernode:GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)
Models supernode heterogeneous						
HGAT_simple	0	2	SimpleConv('add')	4 * ((HeteroConv(('normal', 'toSup', 'supernodes'): SimpleConv('add'), ('normal', 'orig', 'normal'): GATConv(32), ('supernodes', 'toNor', 'normal'): GATConv(32), , aggr='sum'))+ relu)	add(global_add_pool of each type)	MLP(3L,32)
HGIN_simple	0	2	SimpleConv('add')	4 * ((HeteroConv(('normal', 'toSup', 'supernodes'): SimpleConv('add'), ('normal', 'orig', 'normal'): GINConv(MLP,32), ('supernodes', 'toNor', 'normal'): GINConv(MLP,32), , aggr='sum'))+ relu)	add(global_add_pool of each type)	MLP(3L,32)
Models supernode multi heterogeneous						
HGAT_m_simple	0	3	SimpleConv('add')	4 * ((HeteroConv(('normal', 'orig', 'normal'): GATConv(32), ('normal', 'toSup', $\mathcal{C}$): SimpleConv('add'), ($\mathcal{C}$, 'toNor', 'normal'): GATConv(32), , aggr='sum'))+ relu) $\forall\mathcal{C}$	add(global_add_pool of each type)	MLP(3L,32)
HGIN_m_simple	0	3	SimpleConv('add')	4 * ((HeteroConv(('normal', 'orig', 'normal'): GINConv(MLP,32), ('normal', 'toSup', $\mathcal{C}$): SimpleConv('add'), ($\mathcal{C}$, 'toNor', 'normal'): GINConv(MLP, 32), , aggr='sum'))+ relu) $\forall\mathcal{C}$	add(global_add_pool of each type)	MLP(3L,32)
HGIN_m_all	0	3	SimpleConv('add')	4 * ((HeteroConv(('normal', 'orig', 'normal'): GINConv(MLP,32), ('normal', 'toSup', $\mathcal{C}$): GINConv(MLP, 32), ($\mathcal{C}$, 'toNor', 'normal'): GINConv(MLP, 32), , aggr='sum'))+ relu) $\forall\mathcal{C}$	add(global_add_pool of each type)	MLP(3L,32)
HGT_multi	0	3	-	4 * (HGTConv(32, heads=4) + relu)	add(global_add_pool of each type)	MLP(3L,32)

Results

Results different models										
	Basic(60)		Regular(140)		Extension(100)		CFI(100)		Total(400)	
Model	Number	Accuracy	Number	Accuracy	Number	Accuracy	Number	Accuracy	Number	Accuracy
Non GNN										
2-WL	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
3-WL	60	100.0%	50	35.7%	100	100.0%	-	-	210	52.5%
Original										
GAT0	0	00.0%	0	00.0%	0	00.0%	0	00.0%	0	00.0%
GIN0	0	00.0%	0	00.0%	0	00.0%	0	00.0%	0	00.0%
Supernode homogeneous, concepts: cyclebasis_maxcliques										
GIN_Sadd	17	28.3%	40	28.6%	10	10.0%	3	03.0%	70	17.5%
GAT_Sadd	11	05.4%	32	22.9%	8	08.0%	3	03.0%	54	13.5%
GIN_SGIN	31	51.7%	51	36.4%	18	18.0%	3	3.0%	103	25.8%
GIN_SGIN _noSINIT	50	83.3%	81	57.9%	43	43.0%	3	3.0%	177	44.2%
GIN_SGIN_typef	48	80.0%	92	65.7%	31	31.0%	3	3.0%	174	43.5%
Supernode heterogeneous, concepts: cyclebasis_maxcliques										
HGAT_simple	41	68.3%	93	66.4%	35	35.0%	3	3.0%	172	43.0%
HGIN_simple	29	48.3%	50	35.7%	17	17.0%	3	3.0%	99	24.8%
Supernode heterogeneous multi, concepts: cyclebasis_maxcliques										
HGAT_m_simple	51	85.0%	99	70.7%	41	41.0%	3	3.0%	194	48.5%
HGIN_m_simple	51	85.0%	115	82.1%	31	31.0%	3	3.0%	200	50.0%
HGIN_m_all	50	83.3%	106	75.7%	33	33.0%	3	3.0%	192	48.0%
HGT_multi	52	86.7%	111	79.3%	31	31.0%	3	3.0%	197	49.2%

After testing several models we have selected the best: GIN_SGIN_noSINIT for homogeneous supernodes and HGIN_m_simple for heterogeneous multi supernodes. On this two models we run the test on all our concepts and some combinations of them.

Results different concepts with homogeneous transformation										
	Basic(60)		Regular(140)		Extension(100)		CFI(100)		Total(400)	
Concept	Number	Accuracy	Number	Accuracy	Number	Accuracy	Number	Accuracy	Number	Accuracy
Supernode homogeneous, model:GIN_SGIN_noSINIT										
constellation	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
cyclebasis	47	78.3%	46	32.9%	30	30.0%	3	3.0%	126	31.5%
k_edge_comp	0	0.0%	1	0.7%	2	2.0%	3	3.0%	6	1.5%
linepaths	0	0.0%	0	0.0%	1	1.0%	3	3.0%	4	1.0%
maxclique	55	91.7%	119	85.0%	22	22.0%	0	0.0%	196	49.0%
maxlines	0	0.0%	42	30.0%	4	4.0%	4	4.0%	50	12.5%
minlines	10	16.7%	19	13.6%	23	23.0%	4	4.0%	56	14.0%
comm_modul	22	36.7%	3	2.1%	22	22.0%	0	0.0%	47	11.8%
maxcliques _cyclebasis	50	83.3%	81	57.9%	43	43.0%	3	3.0%	177	44.2%

Results different concepts with heterogeneous multi transformation										
	Basic(60)		Regular(140)		Extension(100)		CFI(100)		Total(400)	
Concept	Number	Accuracy	Number	Accuracy	Number	Accuracy	Number	Accuracy	Number	Accuracy
Supernode heterogeneous multi, model:HGIN_m_simple										
constellation	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
cyclebasis	24	40.0%	16	11.4%	13	13.0%	3	3.0%	56	14.0%
k_edge_comp	0	0.0%	1	0.7%	2	2.0%	3	3.0%	6	1.5%
linepaths	0	0.0%	0	0.0%	1	1.0%	3	3.0%	4	1.0%
maxclique	55	91.7%	117	83.6%	22	22.0%	0	0.0%	194	48.5%
maxlines	0	0.0%	38	27.1%	7	7.0%	3	3.0%	48	12.0%
minlines	10	16.7%	4	2.9%	22	22.0%	3	3.0%	39	9.8%
k_core	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
degree_centrality	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
comm_modul	22	36.7%	3	2.1%	19	19.0%	0	0.0%	44	11.0%
star2	16	26.7%	14	10.0%	41	41.0%	0	0.0%	71	17.8%
maxcliques _cyclebasis	52	86.7%	116	82.9%	32	32.0%	3	3.0%	203	50.7%
cycb_maxcliq _star2_minl_maxl	53	88.3%	118	84.3%	56	56.0%	3	3.0%	230	57.5%

Considerations
We can see that models on the original graph that doesn't exploit high order information are not able to distinguish the pair of graphs, this would make us believe that the supernode preprocessing is at least more powerful than 1-WL. We run the 2-WL algorithms provided by the author of the paper and got 0 correct results, this means that all the dataset is at least 2-WL difficult, but this fact is not reported in the paper.
Another consideration on the results is that the homogeneous supernode transformation when applied with multiple concept doesn't increase the performance in respect to taking the best single concept, meanwhile with the heterogeneous multi transformation the models perform better.
Our best model uses the heterogeneous multi transformation with five concepts and reach a total accuracy of 57.5%, which is an interesting achievement looking also at the models tested on the BREC paper.

6 Real world dataset

As last test we decided to try supernode graph neural network on real world datasets. To make our analysis we have choosen `IMDB-BINARY` and `PROTEINS` by `TUDataset`.

Dataset
Both the datasets are well balanced and didn't contain any splitting mask for training, validation and test. We decided to split them 0.6, 0.2 and 0.2 respectively after applying a random shuffle. To make the experiments reproducible we set every random seed to a fix number.

Real world dataset						
dataset name	n. graphs	feature dim.	n. classes	avg. n. nodes	avg. n. edges	avg. node degree
IMDB-BINARY	1000	1	2	19.8	193.1	8.89
PROTEINS	1113	3	2	39.1	145.6	3.73

6.1 Settings and results

Models

For the final test we took our best models.

Models					
model name	type	supernode init	node embeddings	readout	classifier
GIN	0	-	3 * (GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)
GIN_SGIN _noSINIT	1	-	3 * (normal: GINConv(MLP,32), supernode:GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)
HGIN_m_simple	3	SimpleConv('add')	3 * ((HeteroConv(('normal', 'orig', 'normal'): GINConv(MLP,32), ('normal', 'toSup', $\mathcal{C}$): SimpleConv('add'), ($\mathcal{C}$, 'toNor', 'normal'): GINConv(MLP, 32), , aggr='sum'))+ relu) $\forall\mathcal{C}$	add(global _add_pool of each type)	MLP(3L,32)

Results

Results						
model name	type	dataset	concepts	epoch	test accuracy	test ROC AUC
GIN	0	IMDB-BINARY	-	150	0.725	0.799
GIN_SGIN_noSINIT	1	IMDB-BINARY	cyclebasis	150	0.705	0.773
GIN_SGIN_noSINIT	1	IMDB-BINARY	maxclique	150	0.690	0.791
HGIN_m_simple	3	IMDB-BINARY	cyclebasis	150	0.755	0.873
HGIN_m_simple	3	IMDB-BINARY	maxclique	150	0.730	0.849
HGIN_m_simple	3	IMDB-BINARY	cycb_maxcliq_star2 _minl_maxl	150	0.750	0.860
GIN	0	PROTEINS	-	150	0.744	0.811
GIN_SGIN_noSINIT	1	PROTEINS	cyclebasis	150	0.789	0.864
GIN_SGIN_noSINIT	1	PROTEINS	maxclique	150	0.785	0.857
HGIN_m_simple	3	PROTEINS	cyclebasis	150	0.722	0.831
HGIN_m_simple	3	PROTEINS	maxclique	150	0.802	0.885
HGIN_m_simple	3	PROTEINS	cycb_maxcliq_star2 _minl_maxl	150	0.686	0.851

Considerations

In both dataset with the right concepts the supernode models perform better than the base model. Further investigation can be done by tuning more the models.

7 Technical implementation

The first idea of supernode for this project was via the homogeneous transformation, both the original node and the supernodes are of the same type and the convolutional graph operations operate on them equally at the same time.
The first idea was to convert the graph to `networkx`, extract the concepts, create and connect the proper node and then convert back the graph to the `torch_geometric` representation. This was not very flexible because the supernode initialization could be done only during the preprocessing and it was impossible to have a learnable initialization.
At this point we decided to initialize the supernode naively and then implement an ad hoc first layer that works only on supernodes. We achieved this by creating nodes and edges masks that specify the type of the nodes and then in the model we are able to specify which type of node to update, with this solution we had problems with grad and matrices using complex graph convolution operation so we decided to try to use heterogeneous graph.
The implementation of heterogeneous and heterogeneous multi was challenging due to the fact that there aren't many example online but we succeeds to do that. Then implementing the models using `HeteroConv` we realize it didn't correspond to the *pure* algorithm that we had in mind because it

uses different spaces for each node type.
At the end we come back to the mask technique and changed the model operation with simpler operation to not cause error but also to satisfy the original idea.
Given the good results of the heterogeneous transformation we decided to keep testing it with the various datasets.

During the project we implemented different `Dataset` and `Dataloader` to boost the performance and ported the code to `PyTorch Lightning` to facilitate the use of `tensorboard` to keep track and analyze our experiments.

8 Conclusion

With this project we explored the idea of supernode graph neural network and achieve interesting results showing that this technique increase the expressive power of naive GNN.
Further research can be done to ad-hoc dataset with ad-hoc concept built by expert in the particular field.

References

References