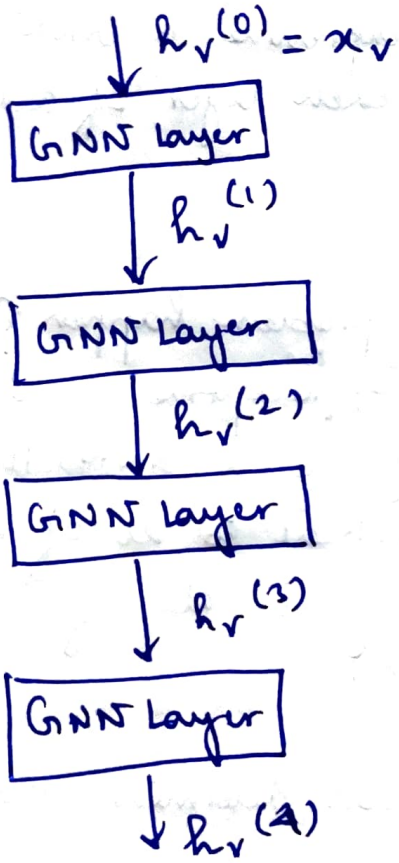
Stacking GNN Layers

How to connect GNN layers into a GNN.

- Stack layers sequentially
- Ways of adding skip connections

Stacking layers sequentially:

This is the standard way.



Input: Initial raw node feature x_v

Output: Node embedding $h_v^{(L)}$ after L GNN layers

The Over Smoothing Problem

GNNs tend to suffer from the over smoothing problem if too many layers are stacked up.

Over smoothing problem result in node embeddings converging to the same value which is a problem as we want to use node embeddings to differentiate them.

~~Why does the over-smoothing problem happen?~~

One key idea to take away from Graph Neural Networks is:
The depth of a Graph Neural Network in terms of layers IS DIFFERENT from the depth of a Convolutional Neural Network or Recurrent Neural Network.

In GNN, the depth tells us HOW MANY HOPS WE SHOULD TRAVERSE TO COLLECT INFORMATION for a specific node?

So, it doesn't say how complex or expressive a GNN is as that depends on what we have inside each layer of a GNN.

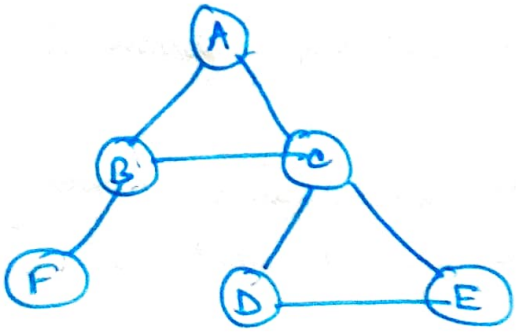
Now,

Why does the over-smoothing problem happen?

The reason is pretty intuitive. If the receptive field (or the nodes covered by hops) is too big for a GNN, then every node tend to have an identical set of nodes to collect information from resulting in all of the node embeddings being almost similar.

Receptive Field: A set of nodes that determine the embedding of a node of interest. For example,

Consider the graph below



1 hop ~~receptive~~ receptive field of node A
$= \{B, C\}$

2 hop receptive field of node A
$= \{B, C, F, D, E\}$
which is the entire graph.

2 hop receptive field of node B
$= \{A, F, C, D, E\}$

So, as the number of hops increases, the receptive field of a node increases and tend to overlap with other nodes. This happens especially when the underlying graph structure has a lot of edges i.e its densely connected.

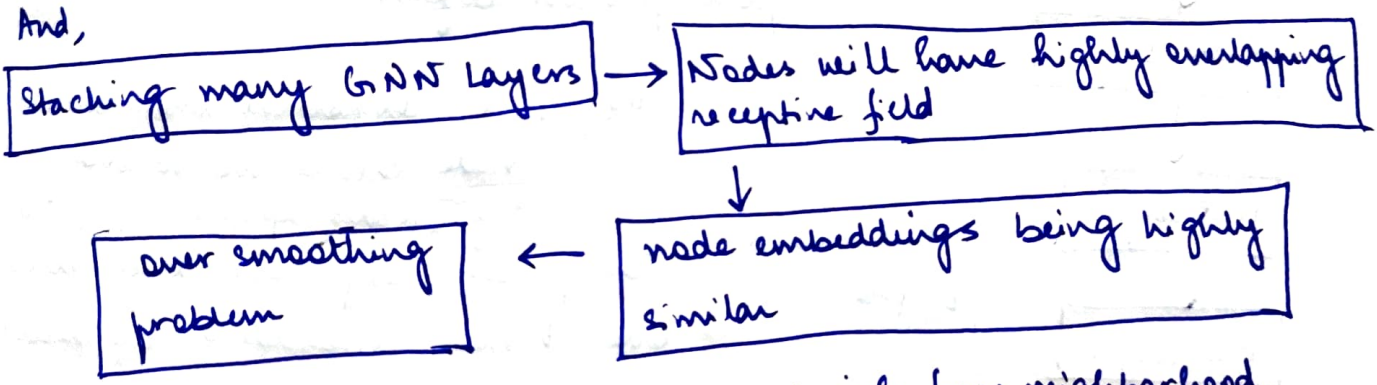
For e.g. if we have a 2-layer GNN to model the graph depicted previous, clearly the

embedding of node A will be similar to the embedding of node B as, both of their receptive fields are almost identical.

Thus,

If two nodes have highly overlapping receptive fields, then their embeddings are highly similar.

And,



If you collect too much info from neighborhood, resulting in everyone collecting too much and thus same information.

Now that we know why over smoothing happens,

How to overcome over smoothing ?

Lesson1: Be cautious when adding GNN layers.

- Adding more GNN layers doesn't always help and cause more problems
- Analyze the necessary receptive field e.g. compute the diameter.
- Set the number of GNN layers a bit more than the receptive field.

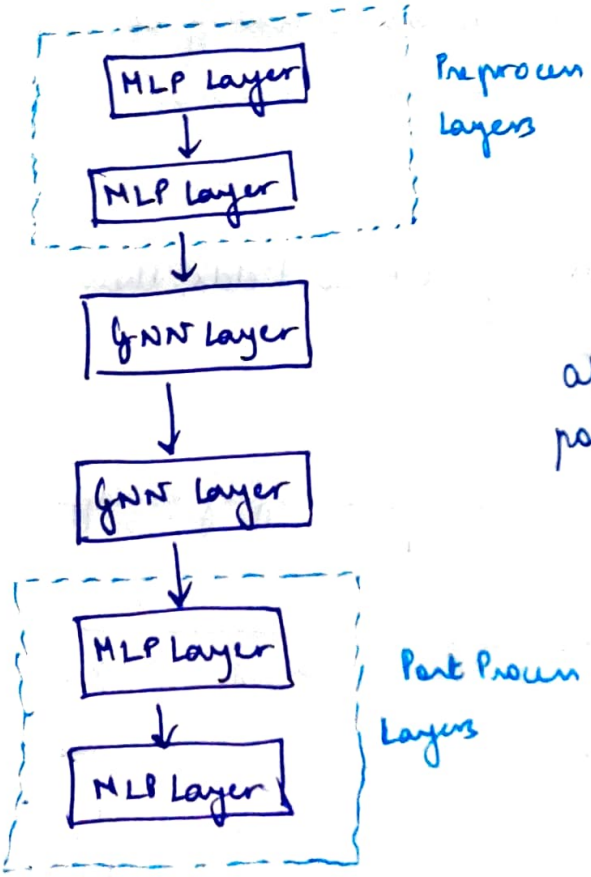
Now, if we have a shallow GNN, how do we make it expressive.

How to make shallow GNNs more expressive?

- Increase the expressive power within the layer.

  We can make Aggregation/Transformation a neural network

- Add layers that don't pass messages



~~A GNN Layer doesn't necessarily contain~~

A GNN doesn't necessarily contain only GNN layers.

We can add MLP layers before and after GNN layers as preprocess and post process layers.

Preprocess: Important when encoding node features are necessary

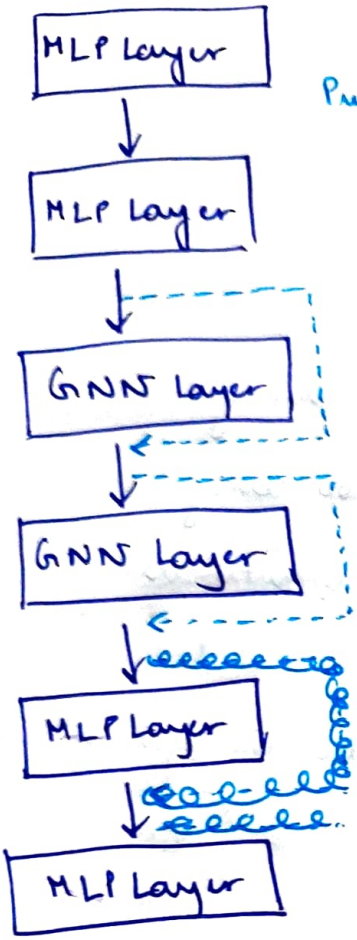
E.g. when Nodes represent images/text

Post process: Important when reasoning over ↑ node embeddings are necessary

E.g. Graph classification, Knowledge graphs.

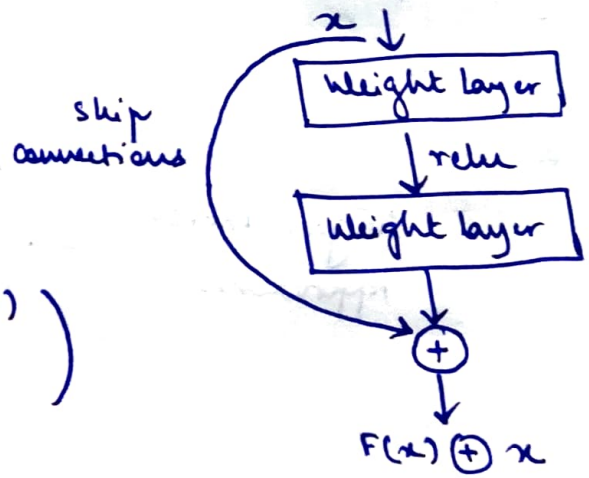
Lemon 2: Add skip connections

Since, when the number of layers are less, GNN can better differentiate nodes, so why not increase their weightage by adding shortcuts to the ~~the~~ GNN architecture?



Why do skip connections work?

- Skip connections create a mixture of models.
- Suppose we have a n layered GNN, with skip connections then, we essentially have 2^n possible paths and thus 2^n Models.
- Thus, we get a mixture of Shallow GNNs + Deep GNNs



E.g. GCN with skip connections:

Standard GCN layer:

$$h_v^{(l)} = \sigma\left(W^{(l)} \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(l-1)}\right)$$

GCN with skip connections:

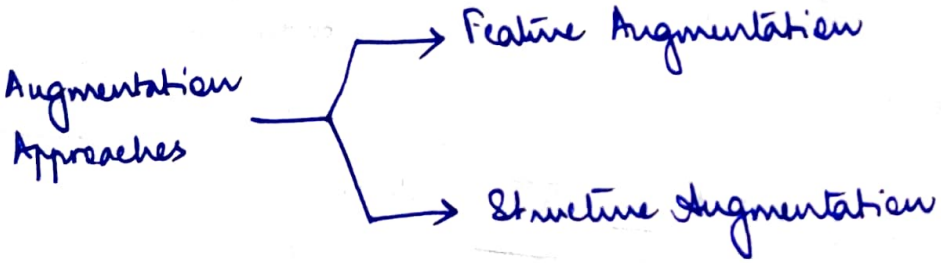
$$h_v^{(l)} = \sigma\Big(\underbrace{W^{(l)} \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(l-1)}}_{F(x)} + \underbrace{h_v^{(l-1)}}_{x}\Big)$$

We can also add skip connections to more than one layers.

GNN Augmentation

Why augment graphs?

- Features: The input graph lacks features
- Graph Structure:
  - The graph is too sparse → inefficient message passing
  - The graph is too dense → message passing too costly
  - The graph is too large → cannot fit the computational graph into GPU.
- Its unlikely that the input graph happens to be the optimal computation graph for embeddings.



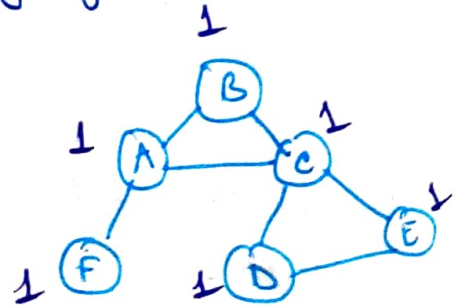
Feature Augmentation:

Why feature augmentation?

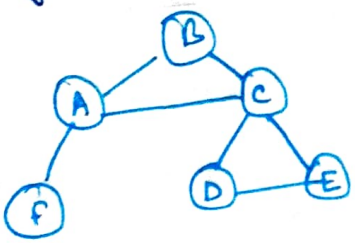
- Input Graph might not have enough features.

Approaches:

- Assign constant values to nodes



- Assign unique IDs to nodes



Node A: [1|0|0|0|0|0]

Node B: [0|1|0|0|0|0]

⋮

and so on.

The unique id's are converted into one hot encoded vectors

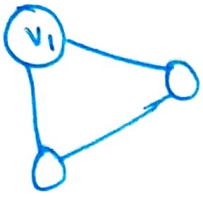
Feature augmentation: Constant vs one hot

	Constant Node Feature	One hot Node Feature
Expressive power:	Medium: All nodes are identical, but GNN can still learn from graph structure.	High. Each node has a unique ID, so node specific information can be stored.
Inductive learning:	High: Simple to generalise to new nodes as we assign constant feature to them and then apply GNN.	Low: Cannot generalize to new nodes as new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs.
Computational cost:	Low: Only 1 dimensional feature	High: $O(\|V\|)$ dimensional feature. Cannot apply to large graphs.
Use Cases:	Any graph, inductive settings as it can generalize to new nodes	Small graph transductive settings

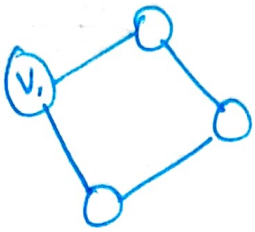
- Certain structures are hard to learn by the GNN.

E.g. Cycle count feature

Consider the two graphs



v_1 in a 3 cycle

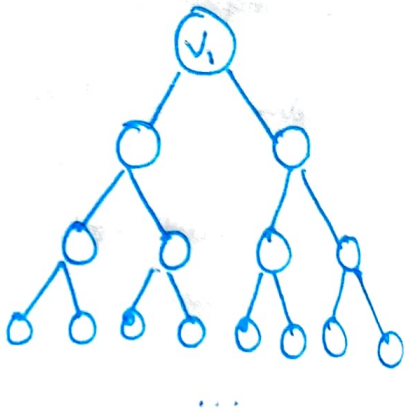


v_1 in a 4-cycle.

Can GNN learn the length of the cycle v_1 resides in?

Unfortunately, no. Because,

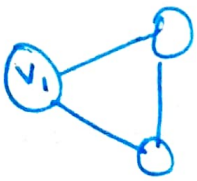
- All nodes will have a degree of 2
- The computational graph for both the nodes in the two different graphs will be the same binary tree.



...

Even if, v_1 resides in an infinite cycle the computational graph would be the same as above

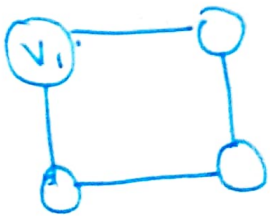
So, we can use node feature augmentation and use cycle-count as augmented node feature.



Augmented node feature for v_1

$[0, 0, 0, 1, 0, 0]$
$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$

$\Rightarrow$ v_1 is in a 3-cycle so the 3rd entry is 1.



Augmented node feature for v_1

$[0, 0, 0, 0, 1, 0]$
$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$

$\Rightarrow$ v_1 is in a 4-cycle so the 4th entry is 1.

Other commonly used features:

- Node degree
- Clustering coefficient
- Page Rank
- Centrality.

Structure Augmentation:

Add Virtual Nodes / Edges:

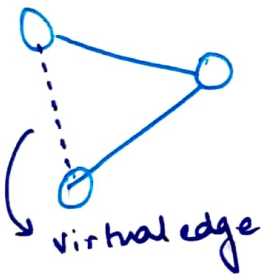
Motivation: Augment sparse graphs

Add virtual edges:

Approach: In sparse graphs, in order to increase the number of edges, we ~~add~~ connect the 2-hop neighbors via virtual edges.

Intuition: Instead of using adjacency matrix A use $A+A^2$ for GNN computation.

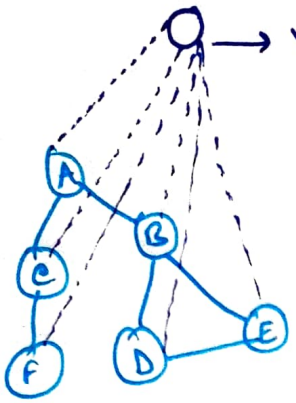
Use Cases: Bipartite Graphs



Add Virtual Nodes:

Motivation: In a ~~too~~ sparse graph two nodes may have a shortest path distance of 10 ie they are very far away and so message passing needs a lot of layers.

In such ~~a~~ cases we add a virtual node and connect these two nodes which will make the shortest path to 2.



Node A — Virtual Node — Node B.

Benefits: Improves message passing in sparse graphs.

Node Neighborhood Sampling:

Previously, all the nodes were used for message passing. But, if the graph is huge or too dense then nodes may have too many neighbors which would make message passing computationally highly expensive.

So, for message passing we can randomly sample a node's neighborhood

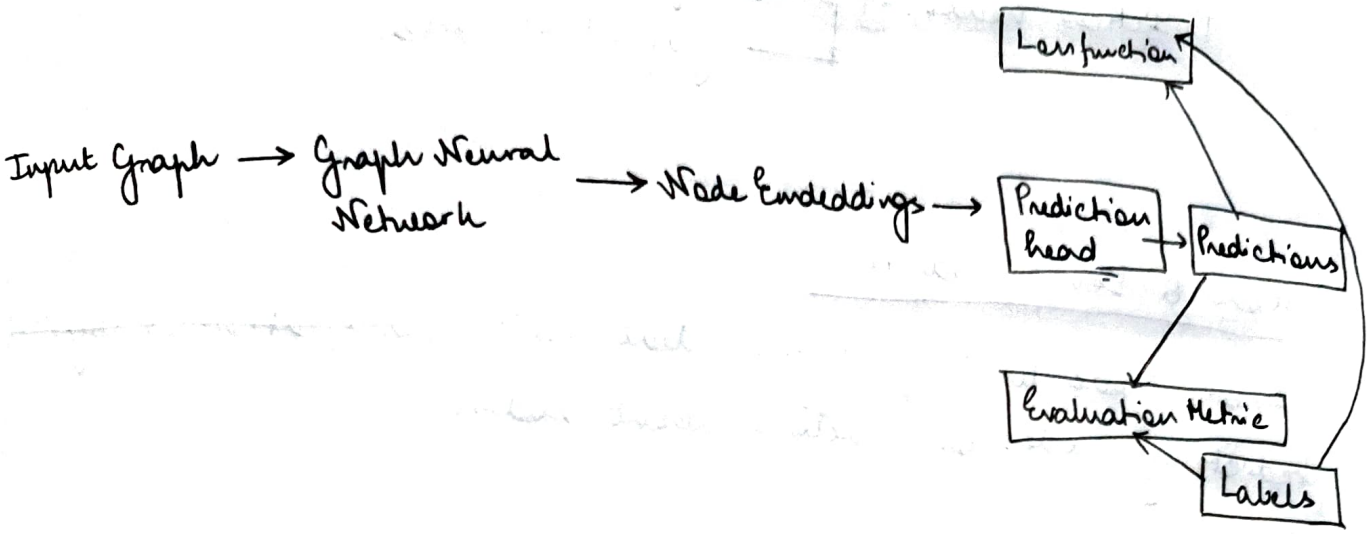
~~Training Graph Neural Networks~~

Benefits:
- Greatly reduces computational cost
- Allows scaling for large graphs
- And in practice, works great.

Training Graph Neural Networks

Now, that we're done with the bits and pieces and intricacies of the layers & and connections, let us discuss about how to train ~~these~~ these networks.

GNN Training Pipeline:



Output of a GNN: So, after a pass through a GNN what we have is ~~a~~ are embeddings for each node.

Output of a GNN: $\{ h_v^{(L)} : v \in V \}$ L is the number of layers in the GNN.

So far, we've discussed how to learn node embeddings using a GNN, now lets talk about the ~~two~~ second part which is defining a prediction head, choosing the loss function based on the task, setting up the evaluation metrics.

Prediction Heads:

Prediction heads are the type of tasks we want to define after learning the node embeddings.

Prediction heads: → Node-level tasks
→ Edge-level tasks
→ Graph level tasks

Node Level Prediction

- ~~The~~ Node level prediction deal with ~~classification or regression~~ predicting nodes or factors about nodes.

Since, we've learnt node embeddings from the GNN, we can directly make prediction about each node from there node embeddings

After GNN computation, suppose we have learnt ~~for~~ d-dimensional embeddings:

$$H := \{ h_v^{(L)} : v \in V,\ h_v^{(L)} \in \mathbb{R}^d \}$$

Now, suppose we want to make a k-way prediction

- Classification: classify among k-categories
- Regression: regress on k targets.

Then, the prediction head is ~~diff~~ defined by:

$$\hat{y}_v = \text{Head}_{\text{node}}\left(h_v^{(L)}\right) = \left(W^{(H)}\right)^T h_v^{(L)}$$

$h_v^{(L)} \in \mathbb{R}^d$ ~~$W^{(H)} \in \mathbb{R}^{K \times d}$~~ $W^{(H)} \in \mathbb{R}^{d \times K}$

So, we map the embeddings from $\mathbb{R}^d$ to $\mathbb{R}^K$ so that we can now compute the loss.

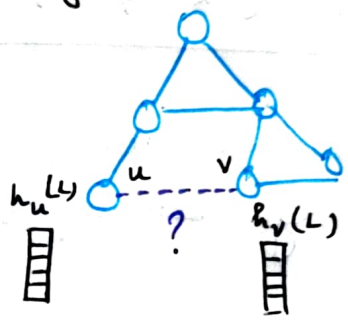
Based on the loss (task chosen) we can transform this $W^{(H)T} h_v^{(L)}$ using e.g. softmax in case of classification.

Edge Level Prediction

In edge level prediction, we use pairs of node embeddings to predict edges.

For e.g. suppose we want to make a k-way prediction, then

$$\hat{y}_{uv} = \text{Head}_{\text{edge}}(h_u^{(L)}, h_v^{(L)})$$



There are options for defining this $\text{Head}_{\text{edge}}(h_u^{(L)}, h_v^{(L)})$:

- Concatenation + Linear

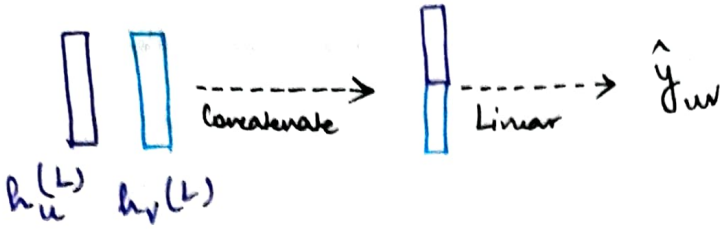
This clearly a obvious way where we concatenate the two embeddings and transform it to a k-dimensional space to make predictions.

$$\hat{y}_{uv} = (W^{(H)})^T \cdot (h_u^{(L)} \,\|\, h_v^{(L)})$$

$\|$ denotes concatenation.

$h_u^{(L)}, h_v^{(L)} \in \mathbb{R}^d$

$W^{(H)} \in \mathbb{R}^{2d \times k}$



- **Dot Product**

$$\hat{y}_{uv} = h_u^{(L)T} h_v^{(L)}$$

$$\boxed{\hat{y}_{uv} = \left(h_u^{(L)}\right)^T h_v^{(L)}}$$

We can use dot product to make predictions on the edge between u and v.

~~However~~

However, this is applicable for only one-way prediction which is basically link prediction.

Generalizing to k-way predictions

We have $W^{(1)}, \ldots, W^{(k)}$ trainable weights s.t

$$\hat{y}_{uv}^{(1)} = \left(h_u^{(L)}\right)^T \cdot W^{(1)} \cdot h_v^{(L)}$$

$$\hat{y}_{uv}^{(2)} = \left(h_u^{(L)}\right)^T \cdot W^{(2)} h_v^{(L)}$$

$$\vdots$$

$$\hat{y}_{uv}^{(k)} = \left(h_u^{(L)}\right)^T \cdot W^{(k)} h_v^{(L)}$$

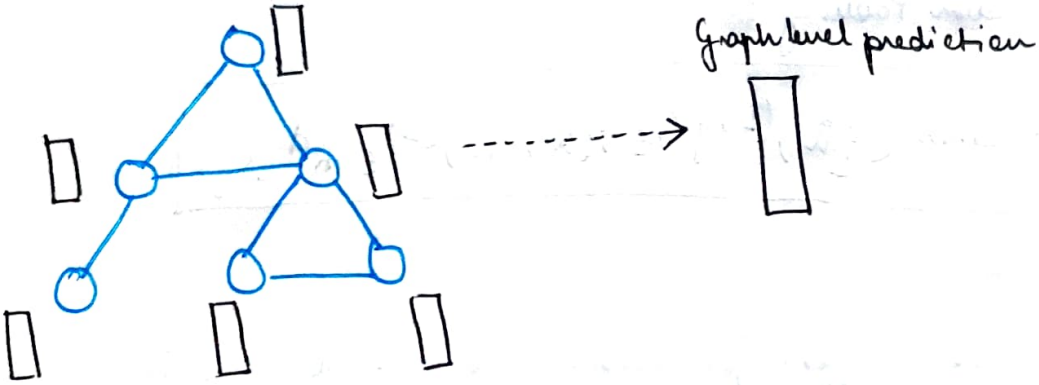
$$\boxed{\hat{y}_{uv} = \Big\|_{i=1}^{k} \left(h_u^{(L)}\right)^T W^{(i)} \cdot h_v^{(L)}}$$

$$\boxed{\hat{y}_{uv} \in \mathbb{R}^k}$$

Graph level Prediction

In this task we want to make prediction using all the node embeddings in our graph.

We want to learn one embedding for the entire graph and then make predictions.



Suppose we want to make a k-way prediction.

$$\hat{y}_G = \text{Head}_{\text{graph}}\left(\left\{h_v^{(L)} \mid v \in V(G)\right\}\right)$$

Clearly $\text{Head}_{\text{graph}}$ is similar to the $\text{Aggr}(\cdot)$ in GNN.

Options for $\text{Head}_{\text{graph}}\left(\left\{h_v^{(L)} \mid v \in V(G)\right\}\right)$

~~• Global Mean Pooling~~

• Global Mean Pooling

$$\hat{y}_G = \text{Mean}\left(\left\{h_v^{(L)} \mid v \in V(G)\right\}\right)$$

~~$\forall$~~ $h_v^{(L)} \in \mathbb{R}^d$ $\forall\ v \in V$

- Global Max Pooling :

$$\hat{y}_G = \text{Max}\left(\{h_v^{(L)} \mid v \in V(G)\}\right) \qquad h_v^{(L)} \in \mathbb{R}^d \quad \forall v \in V.$$

- Global Sum Pooling :

$$\hat{y}_G = \text{Sum}\left(\{h_v^{(L)} \mid v \in V(G)\ h_v^{(L)} \in \mathbb{R}^d\}\right)$$

Now, these work great for small graphs. However, it may face challenges ~~for~~ while dealing with larger graphs due to :

- Loss of local information.
- Computational complexity
- ~~Invasio~~ Difficulty in capturing long term dependencies.

So, can we do better for large graphs ?

Global pooling over a large graph have a lot of issues. One of them is loss of information. In order to illustrate this let us consider 1-d embeddings.

- 1d embeddings for G_1 : $\{-1, -2, 0, 1, 2\}$
- 1d embeddings for G_2 : $\{-10, -20, 0, 10, 20\}$

Clearly embeddings for G_1 and G_2 are very different and thus their structures must also be very different.

So, if we perform global sum pooling:

- Prediction for G_1: $\hat{y}_{G_1} = \text{Sum}(\{-1, -2, 0, 1, 2\}) = 0$.
- Prediction for G_2: $\hat{y}_{G_2} = \text{Sum}(\{-10, -20, 0, 10, 20\}) = 0$.

$\therefore$ We cannot differentiate G_1 and G_2!

A solution: We aggregate all the nodes hierarchically

Consider the same node embeddings of G_1 and G_2.

G_1: $\{-1, -2, 0, 1, 2\}$

G_2: $\{-10, -20, 0, 10, 20\}$

* We aggregate via the following rule:
  - We first separately aggregate the first 2 nodes and the last 3 nodes
  - Then aggregate again to make the final prediction.

Aka, Aggregate → Sum + ReLU

G_1 node embeddings: $\{\underbrace{-1, -2}_{a}, \underbrace{0, 1, 2}_{b}\}$

Round 1: $\hat{y}_a = \text{ReLU}\{-1 + -2\} = 0$

Round 2: $\hat{y}_b = \text{ReLU}(0 + 1 + 2) = 3$.

$\therefore$ $\boxed{\hat{y}_{G_1} = \text{ReLU}(\hat{y}_a + \hat{y}_b) = 3}$

G_2 node embeddings: $\{\underbrace{-10, -20}_{a}, \underbrace{0, 10, 20}_{b}\}$

Round 1: $\hat{y}_a = \text{ReLU}(-10-20) = 0$.

Round 2: $\hat{y}_b = \text{ReLU}(0+10+20) = 30$

$\therefore$ $\boxed{\hat{y}_{G_2} = \text{ReLU}(\hat{y}_a + \hat{y}_b) = 30}$

$G_1 \rightarrow \hat{y}_{G_1} : \{3\}$ $\quad G_2 \rightarrow \hat{y}_{G_2} : \{30\}$

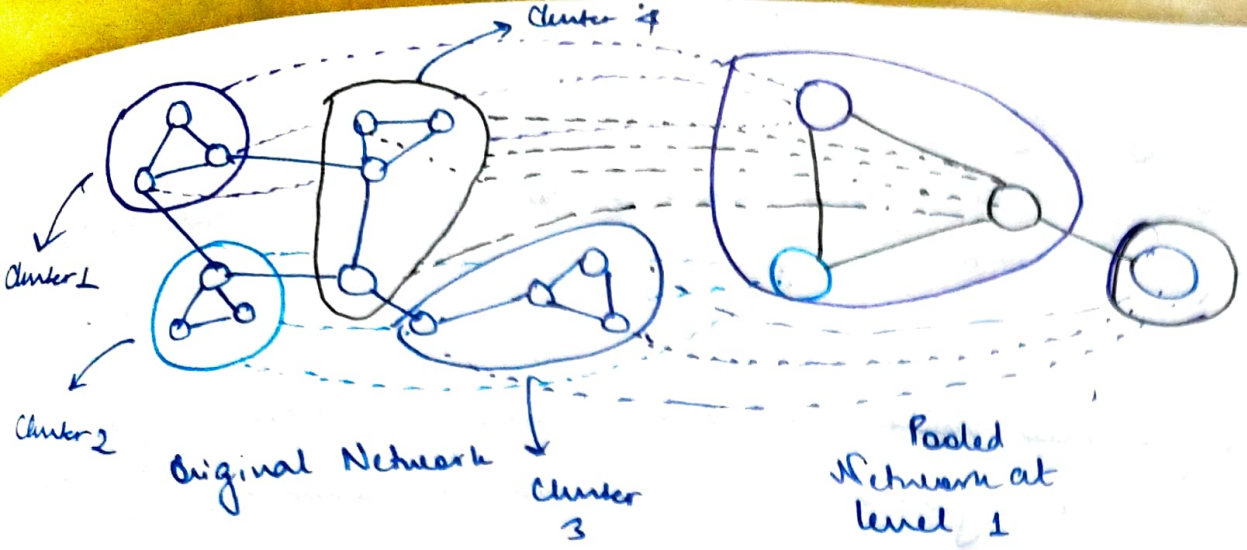
Clearly we can differentiate G_1 and G_2 now!

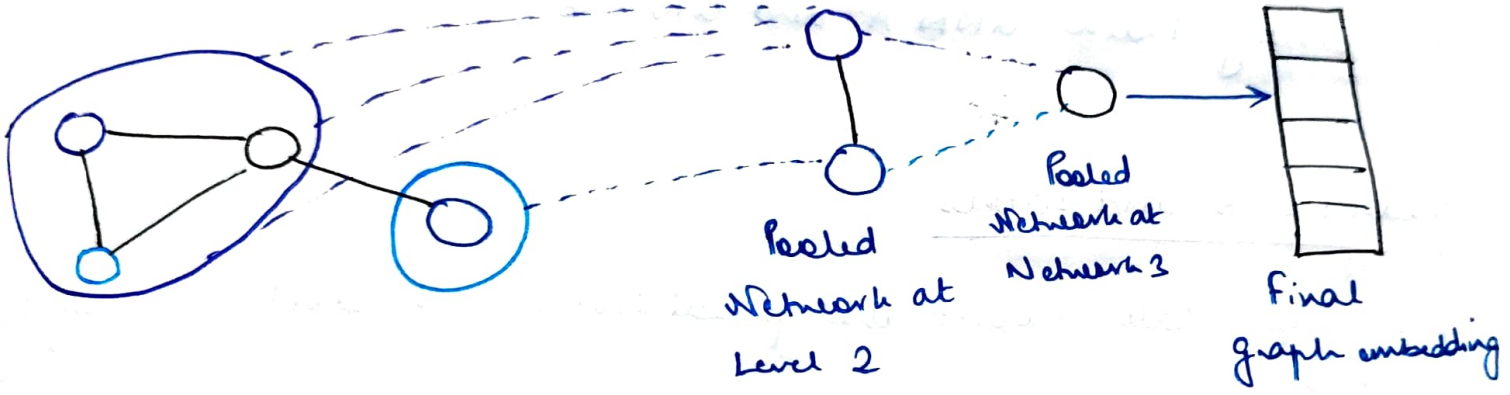
So, Hierarchical Pooling seems to be a better idea, but we have a very reasonable question:

How to decide what to aggregate first and how to hierarchically to aggregate?

Real world graphs do some communities/clusters we to can be made use of while selecting hierarchical aggregation.

So, if we can detect these communities ahead of time we can make use of them to perform aggregation.





One option to learn clusters of these using a standard graph clustering or graph partitioning algorithm.

Other option is to learn these clusters along side learning the node embeddings. This idea is called the **Differential Pooling** which involves: TWO INDEPENDENT GNNs AT EACH LEVEL

GNN A: Compute node embeddings
GNN B: Compute the cluster that a node belongs to

} At each level

GNN A and GNN B at each level can be executed in parallel.

For each Pooling Layer:

- Use cluster assignments from GNN B to aggregate node embeddings generated by GNN A.

- Create a single new node for each cluster, maintaining edges between clusters to generate a new pooled network.

- Jointly train GNN A and GNN B.

Predictions and Labels

Now, lets talk about the predictions and labels part of the pipeline.

⇒ Now, the labels are the target value which our ~~~~ predictions need to attain to reduce the loss.

So, where does the ground truth come from?

- Supervised labels
- Unsupervised signals.

Supervised Learning on Graphs:

The labels come from external sources: E.g. predict drug likeness of a molecular graph.

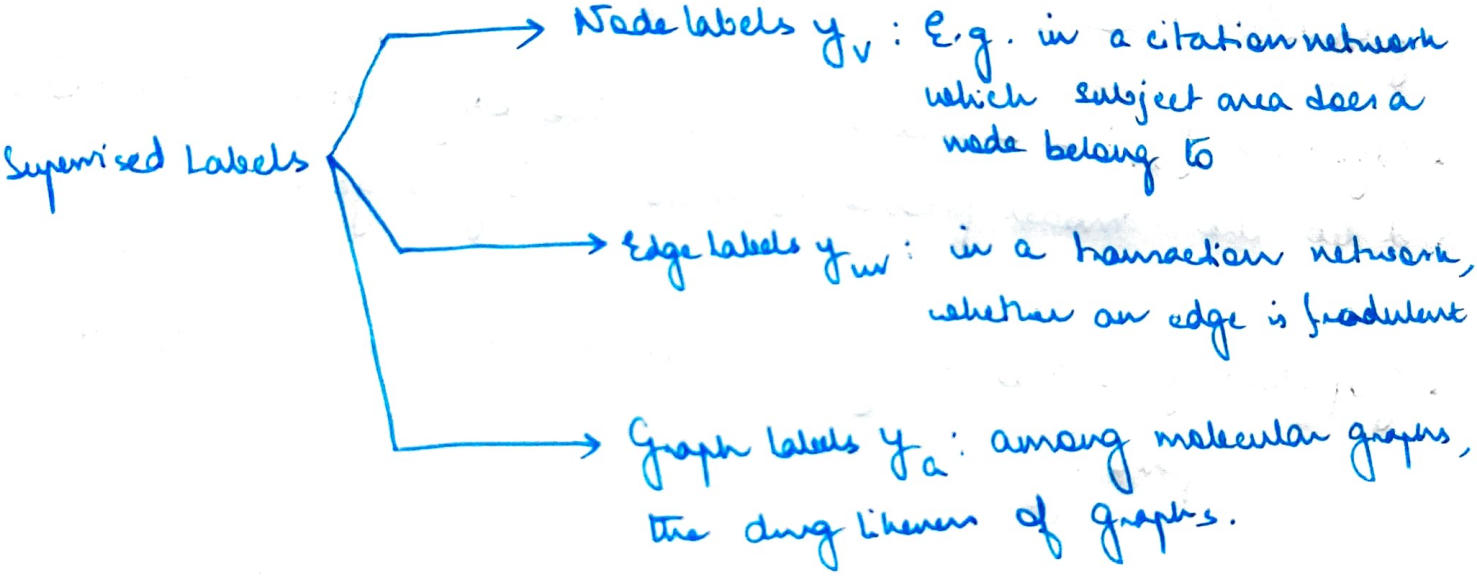
Unsupervised Learning on Graphs:

Signals come from graph themselves: E.g. link prediction

However, we still have "supervision" in unsupervised learning. E.g. train a GNN to predict node clustering coefficient.

This is why another alternate name for unsupervised learning is self-supervised learning.

Supervised Learning:
Supervised Labels come from the specific use cases



Advice: Reduce any supervised graph related task to node/edge/graph labels, as they are easy to work with.

E.g. In the task of node clustering, we can assign some clustering coefficient to each of the nodes and treat the nodes with same label belong to one cluster.

Unsupervised Signals on Graphs

In ~~supervised~~ unsupervised learning we only have a graph without external labels. So, we can make find supervision signals from within the graph ~~to~~ and thus the name ~~supervised~~ self-supervised learning.

For e.g. we can let GNN predict the following:

- Node level y_v: Node statistics such as clustering coefficient, PageRank.
- Edge level y_{uv}: Link prediction; hide the edge between two nodes and let the model predict if there is any edge.
- Graph level y_G: Predict graph statistic like predict if two graphs are isomorphic.

Clearly, these tasks doesn't require any external labels.

Loss Function

Now, we come to the loss function part of the pipeline where we compute the final loss.

~~No~~ The loss function differs from task to task ~~as~~ depending on the type of task.

Final loss → Classification loss
→ Regression loss

~~Sett~~ Setting: Assume we have N data points.

Each data point can be a node / edge / graph

- Node level: prediction: $\hat{y}_v^{(i)}$, label: $y_v^{(i)}$; $i = 1, 2, \ldots, N$
- Edge level: prediction: $\hat{y}_{uv}^{(i)}$, label: $y_{uv}^{(i)}$; $i = 1, 2, \ldots, N$
- Graph level: prediction: $\hat{y}_G^{(i)}$, label: $y_G^{(i)}$; $i = 1, 2, \ldots, N$

So, for each datapoint we have a tuple

$$\left(y^{(i)}, \hat{y}^{(i)} \right) \; ; \; i = 1, 2, \ldots, N$$

($y^{(i)}$: Target, $\hat{y}^{(i)}$: Prediction)

Now, the loss function changes based on whether we are using classification or regression:

Classification e.g. Node classification

Regression: e.g. predict drug ~~likeb~~ likeness of a molecular graph.

Classification Loss: (Cross Entropy Loss)

Suppose we have k class labels, then for i^{th} data point we have

$$CE\left(y^{(i)}, \hat{y}^{(i)} \right) = -\sum_{j=1}^{k} y_j^{(i)} \log \hat{y}_j^{(i)}$$

where $y^{(i)} \in \mathbb{R}^K$ is a one hot encoding of the target

0	0	1	0	0	0

e.g.

$\hat{y}^{(i)} \in \mathbb{R}^K$ is the predict (probability distribution) after softmax(·)

0.2	0.1	0.6	0.04	0.04	0.02

e.g.

Total loss over N training examples

$$\text{Loss} = \sum_{i=1}^{N} CE(y^{(i)}, \hat{y}^{(i)})$$

Regression Loss

For regression loss we often use mean square error a.k.a L2 loss.

Suppose $\hat{y}^{(i)}$ and $y^{(i)} \in \mathbb{R}^K$. Then,

~~$$MSE(y^{(i)}, \hat{y}^{(i)}) = \sum_{j=1}^{K} y_j^{(i)} - \hat{y}^{(i)}$$~~

$$MSE(y^{(i)}, \hat{y}^{(i)}) = \sum_{j=1}^{K} (y_j^{(i)} - \hat{y}_j^{(i)})^2$$

Total loss over all N examples:

$$\text{loss} = \sum_{i=1}^{N} MSE(y^{(i)}, \hat{y}^{(i)})$$

Evaluation Metrics:

Now, we come to the final part of the pipeline which is evaluating the model.

Evaluation Metrics → Classification
→ Regression.

Evaluation Metrics: Regression

- Root Mean Square Error (RMSE)

~~RMSE(y~~

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2}$$

- Mean absolute error (MAE)

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y^{(i)} - \hat{y}^{(i)}|$$

Evaluation Metrics: Classification

Metrics sensitive to classification threshold

- Accuracy
- Precision
- Recall
- F1 Score