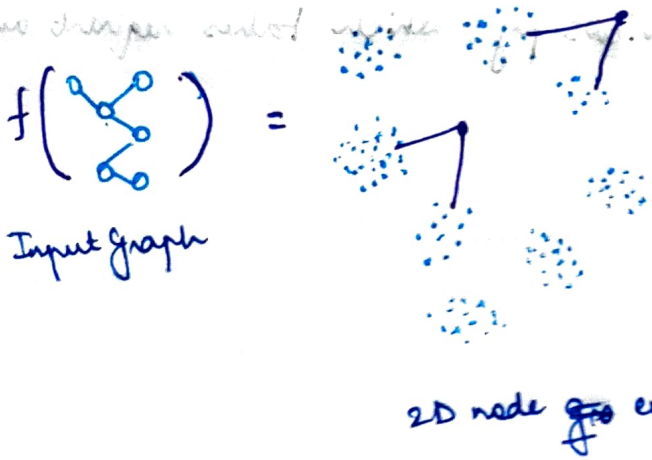
Node Embeddings

Intuition: Map nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together.



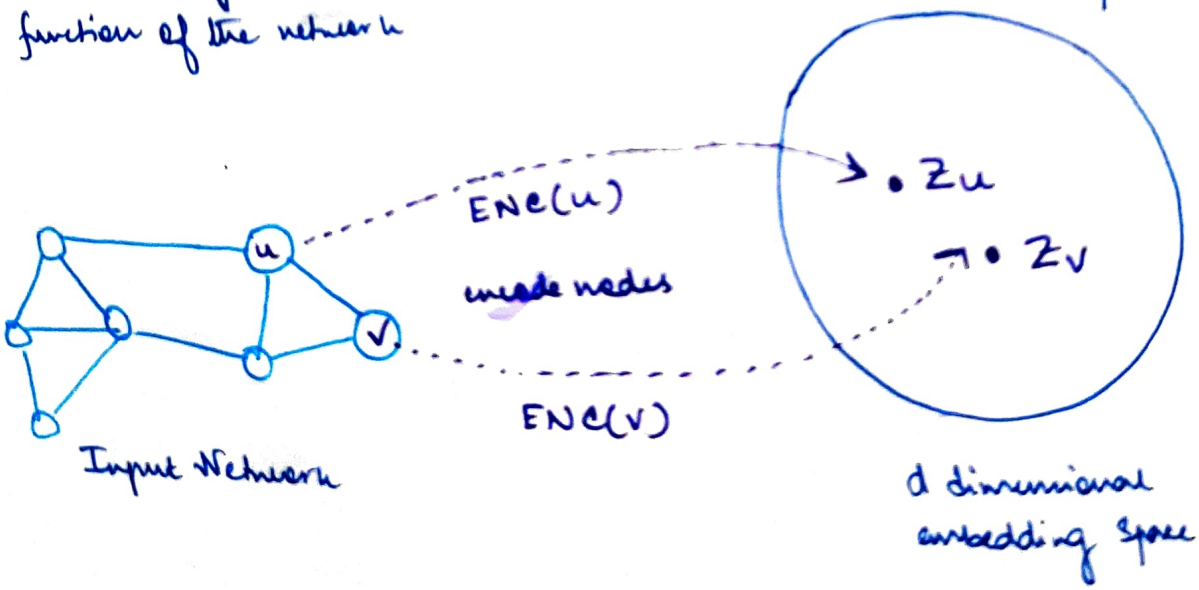
The idea is to encode nodes in such a way that

similarity $(u, v) \approx z_v^T z_u$

$$\text{similarity}(u, v) \approx z_v^T z_u$$

similarity function of the network

similarity function of the vector space



so, if two nodes are connected by edges, then they should be very close together in the embedding space.

In GNN, for a node v

ENC(v) = multiple layers of non-linear transformations based on graph structure

Tasks on Network
- Node Classification
- Link Prediction
- Community detection
- Network Similarity

Deep learning for Graphs

From what we have in classical deep learning, the question comes

HOW TO GENERALIZE CLASSICAL DEEP LEARNING TO WORK WITH GRAPHS?

Set Up: Assume we have a graph G(V, E).

V: Vertex Set

E: $E \subseteq V \times V$ is the set of edges

A: $A_{|V| \times |V|}$ is the adjacency matrix

X: $X \in \mathbb{R}^{|V| \times d}$ is the matrix of node features.

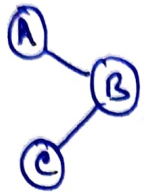
v: a given node of G.

N_v: (or) $N(v)$ is the ~~set of~~ neighbourhood set of the node v.

The most important question is:

HOW DO WE INPUT A GRAPH INTO A NEURAL NETWORK?

The Naive Approach is to use the adjacency matrix of the graph, concatenate it with corresponding node features and feed it to the ~~graph~~ network.

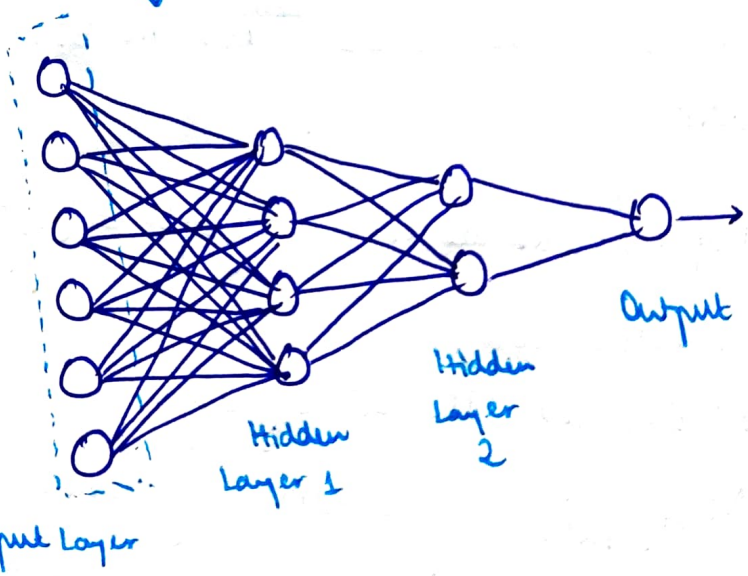


one hot encoded features

A → [1 0 0] B → [0 1 0] C → [0 0 1]

	A	B	C			
A	0	1	0	1	0	0
B	1	0	1	0	1	0
C	0	1	0	0	0	1

→ [Adj(A), feat[A]]



Issues: • Since, the input is a combination of the adjacency entry of a node and its features the number of parameters is $O(|V|)$ and since, we have one training example per node

the number of parameters would be greater than the training examples resulting in unstable training and overfitting.

- This way of ~~encoding into~~ inputting a graph in a neural network will not be applicable for graphs with different sizes since, that would have different adjacency matrices.

- This method is sensitive to node ordering, as the adjacency matrix would change if we label the nodes in a different way. This is result in a different input being passed to the neural network and thereby confusing the model.

Idea: Convolutional Networks

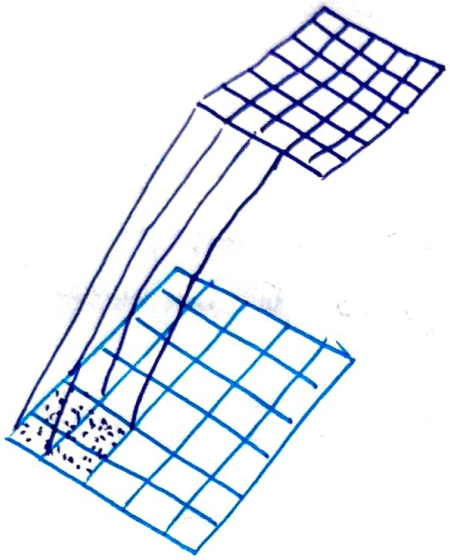
The idea is to generalise convolutions beyond simple lattices and use it on graphs leveraging the node features / attributes.

Problems with using the same convolutional operator (as used in images) in graphs:

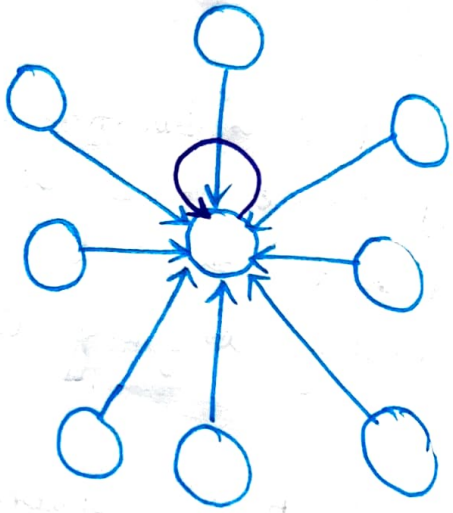
- No fixed notion ~~of~~ of location or sliding window on the graph.

- Graph is permutation invariant i.e the properties of graphs doesn't change to permuting the node ordering. So, we need a function / model that doesn't depend on the arbitrary ordering of the adjacency matrix.

From Images to Graphs



Image



Graphs.

The idea of a convolution is, it takes a submatrix of pixel applies some transformation on it to create a new pixel and then we slide this operator over the entire image.

In graphs, to incorporate this technique we have a central node which will borrow the information from its neighbours and create new information using them as well its own information.

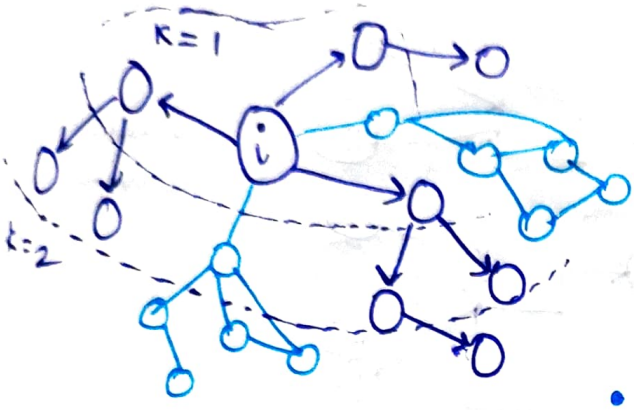
Idea: Transform information from neighbours and combine it:

- Transform messages h_i from neighbours: $W_i h_i$
- Add them up $\sum_i W_i h_i$

~~How to feed a~~

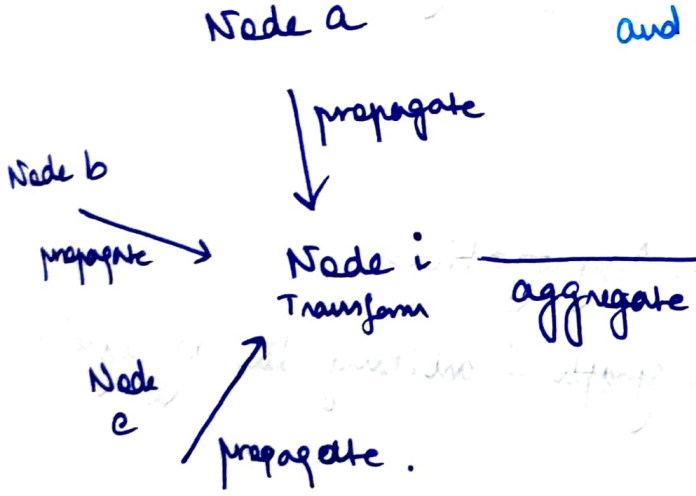
How Graph Convolutional Networks work?

In GCNs every node has its own neural network defined by its neighbourhood.



So, in order to ~~post~~ predict something about node i, then the node i will take information from its neighbours, which will take information from its neighbours and learn to

- Propagate this information across the graph
- Aggregate this information
- Create a new message which the next node can propagate, transform and aggregate.

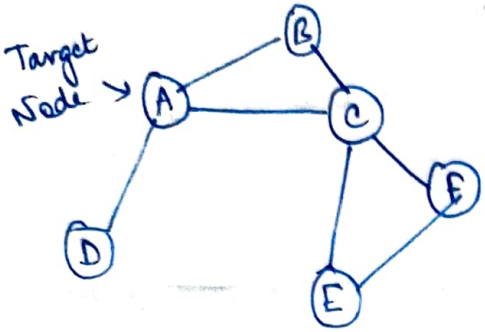


So, the idea in a GCN is:

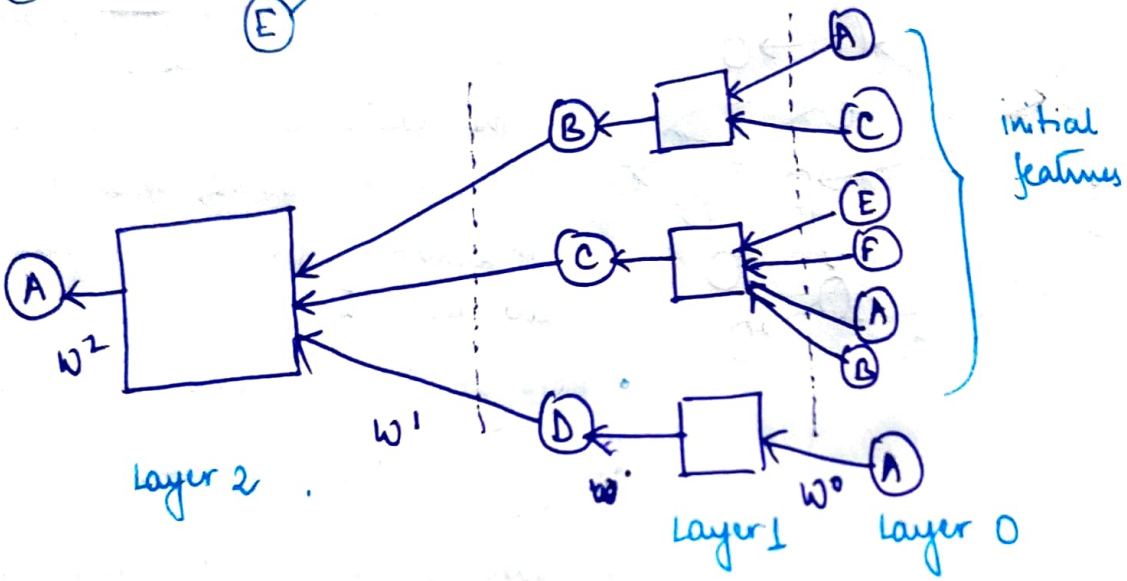
- Determine the node computation graph (local neighbourhood)
- Propagate and transform this information over the computation graph.

This computation graph determines the underlying structure of the neural network.

Suppose we have a graph G as follows:



So, A will have its own neural network architecture (clearly) which would look as follows.



Network Architecture for layer A.

□ := Neural Network:

To learn: Message Transformation, Aggregation.

Now, since the ~~ordering~~ ordering of a graph is arbitary the aggregation should be order-invariant.

NEIGHBORHOOD AGGREGATION SHOULD BE PERMUTATION INVARIANT

?

A simply neighborhood aggregation would be

- Average the ~~information from~~ messages from the neighbors

• Apply the neural network.

Mathematically,

h_v^l := embedding of node v in layer l

W_l := weight matrix in layer l for neighbours

B_l := weight matrix in layer l for the original node (v).

$h_v^0 = x_v \rightarrow$ node feature of node v

trainable parameters

embedding of v at layer l

$$h_v^{(l+1)} = \sigma\left(\frac{1}{|N(v)|}\sum_{u \in N(v)} W_l \cdot h_u^{(l)} + B_l h_v^{(l)}\right)$$

$l \in \{0, 1, \ldots, L-1\}$

$z_v = h_v^{(L)} \rightarrow$ final node embedding

σ := transformation function (non linear e.g ReLU).

Now, if every node has its own architecture and transformation operator, the question comes

HOW DO WE TRAIN THE MODEL?

The parameters are W_l and B_l $\forall\, l \in \{0, 1, \ldots, L-1\}$ are trainable and can be trained using SGD by feeding these embeddings in any loss function.

i.e let $\hat{y} := f(h_v^L)$

y := target label for node v.

Then, $$W^* = \underset{W^0, W^1, \ldots W^{L-1}}{\operatorname{argmin}} \mathcal{L}(\hat{y}, y)$$ can be solved using SGD.

Matrix formulation:

Let $H^{(l)} = [h_1^{(l)} \; h_2^{(l)} \; \dots \; h_{|V|}^{(l)}]^T$

Now, $\sum_{u \in N(v)} h_u^{(l)}$ is basically $A_v H^{(l)}$

Let $D \in \mathbb{R}^{|V| \times |V|}$ be a diagonal matrix s.t

$$D_{v,v} = \deg(v)$$

$$D_{u,v} = 0 \quad \text{if } u \neq v.$$

Then,

$$D D^{-1} = I$$

$\therefore D^{-1}$ is also a diagonal matrix with entries:

$$D^{-1}_{v,v} = \frac{1}{\deg(v)}$$

$$D^{-1}_{u,v} = 0 \quad \text{if } u \neq v$$

So, $\frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(l)}$ is $\sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|}$

which is: $D^{-1}_{v,v} A_v H^{(l)}_v$

∴ Generalizing this

$$\frac{1}{|N(v)|}\sum_{u \in N(v)} h_u^{(l)} = D^{-1} A H^{(l)}$$

Rewriting the update function in matrix form:

$$H^{(l+1)} = \sigma\left(D^{-1} A H^{(l)} W_l^T + H^{(l)} B_l^T \right)$$

$$H^{(l+1)} = \sigma\left(\tilde{A} H^{(l)} W_l^T + H^{(l)} B_l^T \right) \quad \text{where } \tilde{A} = D^{-1} A$$

$\tilde{A} H^{(l)} W_l^T$: neighborhood aggregation

$H^{(l)} B_l^T$: self transformation

How to train a GNN?

Supervised setting : Suppose we are interested in node classification and so, in this supervised setting we minimise the loss

~~$\min_{\theta} \mathcal{L}(y, f(z_v))$~~

y := node label of v

z_v := embedding of node v

$$\min_{\theta} \mathcal{L}(y, f(z_v))$$

We can use SGD to come to the solution.

Unsupervised setting : In the unsupervised setting the graph structure can be used as supervision.

We can define the notion of similarity as the dot product ~~of two nodes should~~ between the two node embeddings should correspond to their similarity in the network

Now, we can use Deep Encoder ~~as the~~ to learn the embeddings.

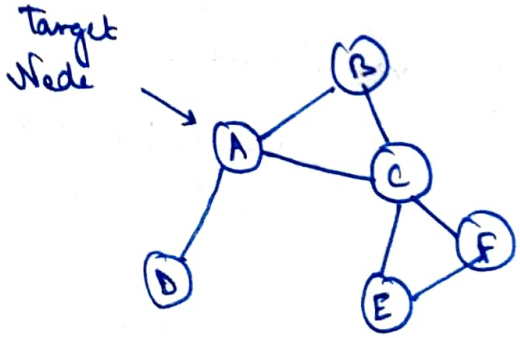
$$\mathcal{L} = \sum_{z_u, z_v} CE\left(y_{u,v}, DEC(z_u, z_v)\right).$$

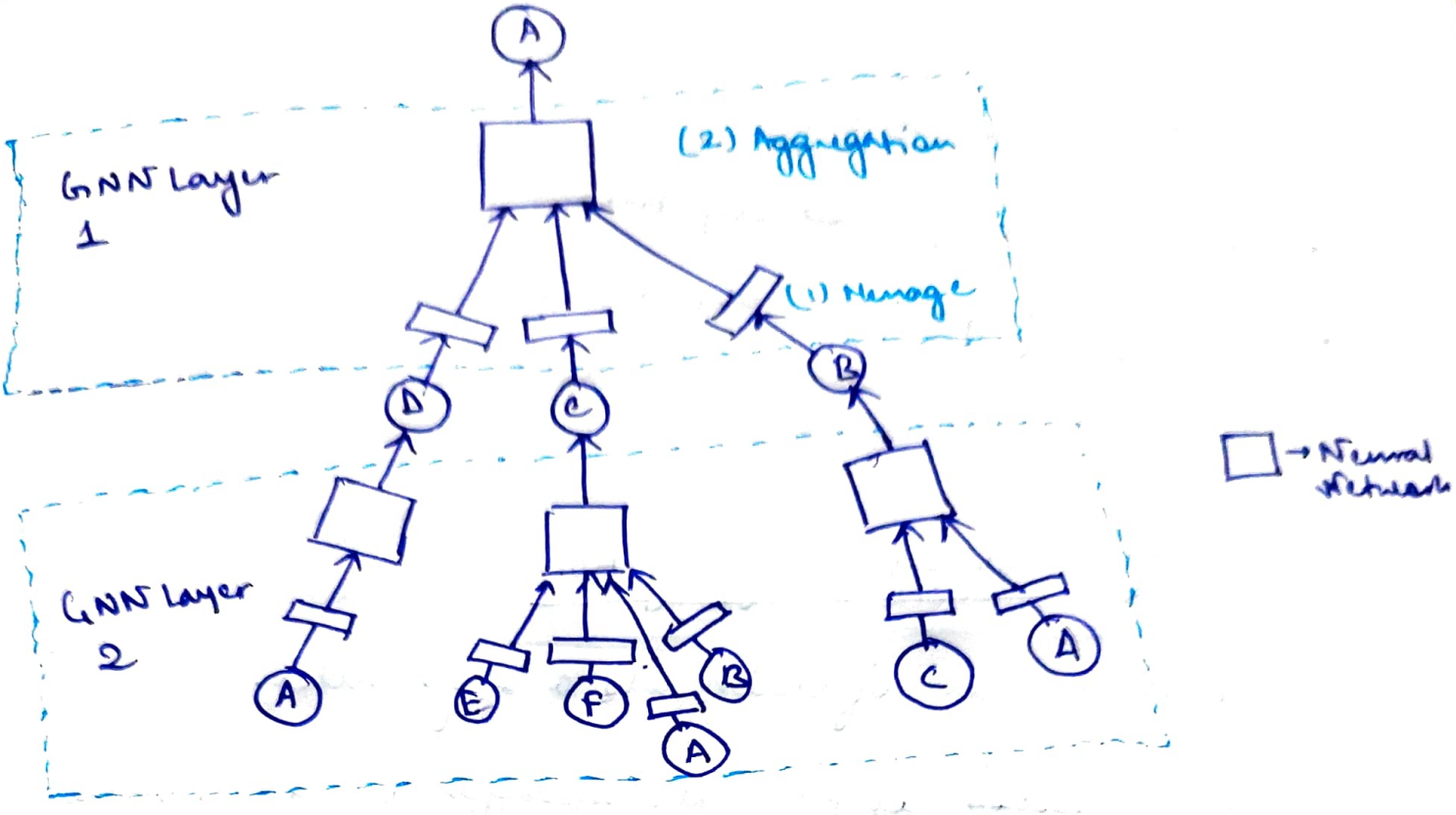
$y_{u,v} = 1$ if u and v are similar.

$DEC(u,v)$ = decoder such as the ~~innerproduct~~ inner product or cosine similarity.

A General GNN Framework

A General GNN Framework has three components to talk about. Consider we have a graph





The first component ~~is~~ of a GNN framework is:

- Message + Aggregation which is the content of every layer of a GNN

The second component is about:

- Stacking layers up in a GNN (Connect GNN layers)
  - We can stack layers sequentially
  - We can add skip connections

The third component is:

- Defining the computational graph. Ideas like Raw Input Graph ≠ Computational graph

Ideas like:

- Graph feature augmentation
- Graph structure manipulation

The final component is:

How to train the GNN?

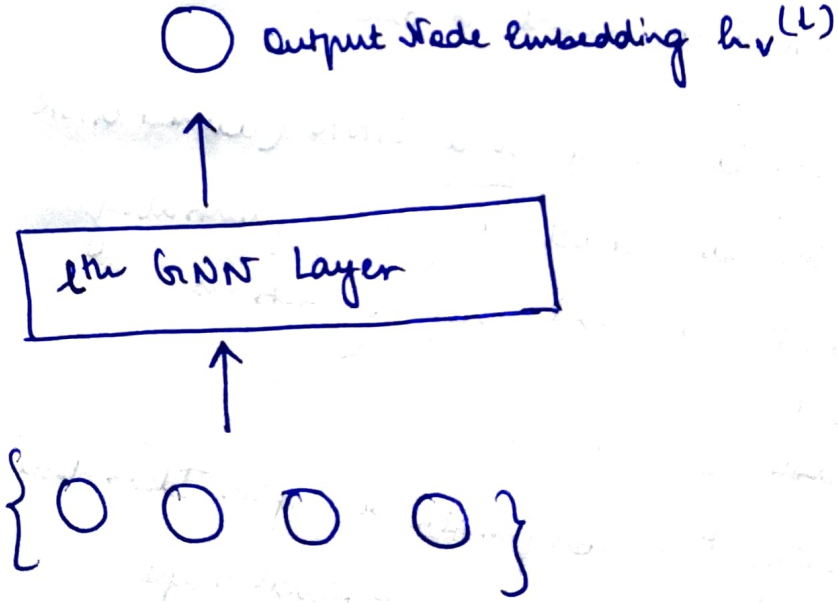
- Supervised / Unsupervised objectives
- Node / Edge / Graph level objective.

A GNN Layer : Message + Aggregation

Idea: Compress a set of vectors into a single vector.

We want to ~~compress~~ compress the messages coming from the neighbors and we do this by a two step process:

- Message Transformation
- Message Aggregation



Input: $h_v^{(l-1)}$ and $h_u^{(l-1)}$; $u \in N(v)$

- **Message Computation**: In this step, we take the output from the previous layer and somehow transform it to create a message to be sent to other nodes layer.

$$m_u^{(l)} = MSG\left(h_u^{(l-1)}\right)$$

Typically this message computation is a linear layer described as follows

$$m_u^{(l)} = W_l h_u^{(l-1)}$$

- **Aggregation**: This is the step, where the transformed messages are aggregated into a single message.

$$h_v^{(l)} = AGG\left(\left\{m_u^{(l)} \mid u \in N(v)\right\}\right)$$

The AGG function can be the $Sum(\cdot)$, $Max(\cdot)$ or $Mean(\cdot)$ aggregator.

Issue: The issue with is that the information from the own node could get lost as we are just using the messages from the neighbors.

Solution: Include $h_v(l-1)$ while computing $h_v(l)$

- **Message**:

$m_u = W_l h_u^{(l-1)}$ — message from neighbors

$m_v = B_l h_v^{(l-1)}$ — message from own node

• Aggregation: After aggregating the messages from neighbors we aggregate the message from v itself via summation or concatenation.

~~$h_v^{(l)} = \text{CONCAT}\left(\text{AGG}\left(\{h_u^{(l-1)} \mid u \in N(v)\}\right),\right.$~~

$$h_v^{(l)} = \text{CONCAT}\left(\text{AGG}\left(\{m_u^{(l-1)} \mid u \in N(v)\}\right), m_v^{(l-1)}\right)$$

↑ messages from neighbors

↑ message from itself

So, putting things together,

(1) Message: Each node computes a message,

$$m_u^{(l)} = \text{MSG}\left(m_u^{(l-1)}\right) \quad u \in N(v) \cup \{v\}$$

(2) Aggregation: Aggregate messages from neighbors

$$z_v^{(l)} = \text{AGG}\left(\{m_u^{(l)} \mid u \in N(v)\}, m_v^{(l)}\right)$$

(3) Transformation: Finally, we can transform the output to add expressiveness to our model.

$$h_v^{(l)} = \sigma\left(z_v^{(l)}\right)$$

σ : ReLU, Sigmoid, etc.

Classical GNN Layers: GCN (Graph Convolutional Networks)

~~For~~ GCN:

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

writing this in the message + aggregation form:

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

GraphSAGE:

$$h_v^{(l)} = \sigma\left(W^{(l)} \cdot \text{CONCAT}\left(h_v^{(l-1)}, \text{AGG}\left(\{h_u^{(l-1)} \mid u \in N(v)\}\right)\right)\right)$$

Writing it as Message + ~~Aggregation~~ form:

- Message is computed ~~through the~~ within AGG(.)

Two stage aggregation:

Stage 1: Aggregate from node neighbors

$$h_{N(v)}^{l} = \text{AGG}\left(\{h_u^{(l-1)} \mid u \in N(v)\}\right)$$

Stage 2: Further aggregate over the node itself

$$h_v^{(l)} = \sigma\left(W^{(l)} \cdot \text{CONCAT}\left(h_v^{(l-1)}, h_{N(v)}^{l}\right)\right)$$

Aggregation Functions:

Mean: Take weighted average of neighbors

$$\text{Aggr} = \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}$$

Pool: We can transform the neighbor vectors and apply Mean(·) or Max(·).

$$\text{Aggr} = \text{Mean}\left(\left\{\text{MLP}\left(h_u^{(l-1)}\right) \mid u \in N(v)\right\}\right)$$

↳ Linear or non linear transformation.

LSTM: We can even apply LSTM to the neighbors. However, since, LSTM's are not order invariant, we must feed it a permutation of the ~~node~~ neighbors everytime so as to teach it not to take the order into consideration.

$$\text{Aggr} = \text{LSTM}\left(\left[h_u^{(l-1)}, \forall u \in \pi(N(v))\right]\right)$$

The choice of the aggregation function ~~do~~ ~~makes~~ decides how expressive our GNN would be.

$$h_v^{(l)} = \sigma\left(W^{(l)} \cdot \text{CONCAT}\left(h_v^{(l-1)}, h_{N(v)}^{l}\right)\right)$$

Aggregation Functions:

Mean: Take weighted average of neighbors

$$\text{Aggr} = \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}$$

Pool: We can transform the neighbor vectors and apply Mean(·) or Max(·).

$$\text{Aggr} = \text{Mean}\left(\left\{\text{MLP}\left(h_u^{(l-1)}\right) \mid u \in N(v)\right\}\right)$$

↳ Linear or non linear transformation.

LSTM: We can even apply LSTM to the neighbors. However, Since, LSTM's are not order invariant, we must feed it a permutation of the ~~most~~ neighbors everytime so as to teach it not to take the order into consideration.

$$\text{Aggr} = \text{LSTM}\left(\left[h_u^{(l-1)}, \forall u \in \pi(N(v))\right]\right)$$

The choice of the aggregation function ~~do~~ ~~makes~~ decides how expressive our GNN would be.

l_2 Normalization:

In GraphSAGE, l_2 normalization is applied at every layer. That is,

$$h_v^{(l)} = \frac{h_v^{(l)}}{\| h_v^{(l)} \|_2} \quad \forall v \in V$$

- Without l_2 normalization, the embedding vectors have different scales.
- In some cases, normalization also result in performance improvement.

Graph Attention Networks

~~$h_v^{(l)} = \sigma\left(\sum_{u \in N(v)} W^{(l)} \alpha_{uv} h_u^{(l-1)} + \alpha_{vv} h_v^{(l-1)} \right)$~~

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

↑ Attention weights

In comparison to GraphSAGE and GCN here attention weights α_{uv} were basically:

$\alpha_{uv} = \frac{1}{|N(v)|}$ and thus were based on the structural properties of graphs like node degree.

Also, clearly in this all neighbours $u \in N(v)$ are equally important

to node v.

Now, the idea in GATs is

> Not all node's neighbors are equally important.

- Attention inspired by cognitive attention focuses on the important parts of the input data and throws out the rest.

- Which part of the data is more important depends on the content and is learned through training.

So,

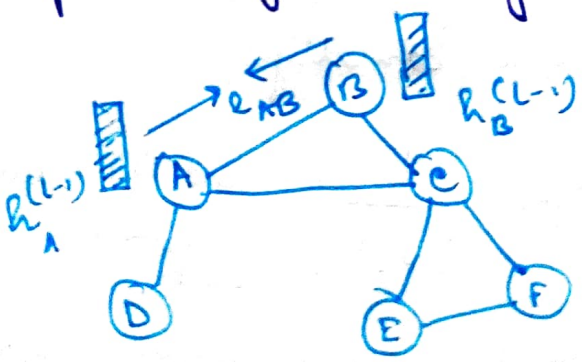
Can we let the weighing factors α_{uv} to be learned?

Then, weighing factors are learned along with all the other parameters of the network.

Then, coefficients α_{uv} are learnt ~~as a~~ as a byproduct of an attention mechanism ~~e_{uv}~~. a.

$$e_{uv} = a\left(W^{(l)} h_u^{(l-1)},\ W^{(l)} h_v^{(l-1)}\right)$$

e_{uv} indicates the importance of u's message to node v.



So, some function a is defined that will take the transformed embedding of u at previous layer, transformed embedding of v at previous layer and return an attention score. e_{uv}

Then, we normalize these weights e_{uv} using the softmax function.

$$\alpha_{uv} = \frac{\exp(e_{uv})}{\sum_{u \in N(v)} \exp(e_{uv})}$$

$$\alpha_{uv} = \frac{\exp(e_{uv})}{\sum_{k \in N(v)} \exp(e_{kv})}$$

Now, the form of attention mechanism is dependent on the choice of a. A simple choice of a would be to concatenate the two followed by a linear layer where the weights of the linear layer would be jointly learnt with the weights of the GNN.

$$\therefore \quad e_{uv} = \text{LeakyReLU}\left(a^T \cdot \left[W^{(l)} h_u^{(l-1)} \,\|\, W^{(l)} h_v^{(l-1)}\right]\right)$$

$a^T \in \mathbb{R}^{2F}$ where, $W^{(l)} h_u^{(l-1)} \in \mathbb{R}^F$

$\|$: concatenation operation $W^{(l)} h_v^{(l-1)} \in \mathbb{R}^F$

$$\alpha_{uv} = \frac{\exp\left(\text{LeakyReLU}\left(a^T \left[W^{(l)} h_u^{(l-1)} \,\|\, W^{(l)} h_v^{(l-1)}\right]\right)\right)}{\sum_{k \in N(v)} \exp\left(\text{LeakyReLU}\left(a^T \left[W^{(l)} h_k^{(l-1)} \,\|\, W^{(l)} h_v^{(l-1)}\right]\right)\right)}$$

Now, working with these attention weights can be tricky as they can be hard to learn or they may be hard to converge.

So, inorder to stabilize learning we introduce Multihead attention. which stabilizes the learning process.

Inorder to do that we obtain multiple attention scores and use it to compute multiple embeddings of the same node. Then, we concatenate them.

$$h_v^{(l)}[1] = \sigma\left(\sum_{u \in N(v)} W^{(l)} \alpha_{uv}^{1} h_u^{(l-1)}\right)$$

$$h_v^{(l)}[2] = \sigma\left(\sum_{u \in N(v)} W^{(l)} \alpha_{uv}^{2} h_u^{(l-1)}\right)$$

$$\vdots$$

$$h_v^{(l)}[k] = \sigma\left(\sum_{u \in N(v)} W^{(l)} \alpha_{uv}^{k} h_u^{(l-1)}\right)$$

$h_u^{(l-1)} \in \mathbb{R}^F$

$$\therefore \boxed{h_v^{(l)} = \Big\Vert_{i=1}^{K} h_v^{(l)}[i]}$$

$\Vert$ represents concatenation.

So, instead of F features we now have KF features of there embeddings.

Benefits of Attention Mechanism:

- Key Benefit: Allows specifying different importance values to different neighbors

- Computationally Efficient: Computation of attention coefficients can be parallelized across all edges.

  Aggregation may be parallelized over all nodes.

Storage efficient:

- Sparse Matrix operations donot require more than $O(V+E)$ entries to be stored.

- Fixed number of parameters irrespective of graph size (as, the weight in the linear layer of attention mechanism depend on the features of node embeddings and not the graph).

Localized:

Only attends to local network neighborhoods.

Inductive capability:

It is shared edge-wise mechanism

Doesnot depend on the global graph structure.

Modifying all of these GNN architectures to include ~~their own~~ self representations of nodes while learning embeddings.

GCN:

$$h_v^0 = x_v \quad \forall v \in V$$

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

GraphSAGE:

~~$h_v^{(l)} = \sigma\left(W^l \cdot \left[\underset{u \in N(v)}{AGG}\left(\{h_u^{(l-1)}\}\right), h_v^{(k-1)}\right.\right.$~~

$$h_v^{(0)} = x_v \quad \forall v \in V$$

$$h_v^{(l)} = \sigma\left(W^{(l)} \cdot \left[\underset{u \in N(v)}{AGG}\left(\{h_u^{(l-1)}\}\right), h_v^{l-1}\right]\right) \quad \forall v \in V.$$

GAT

$$h_v^{(0)} = x_v \quad \forall v \in V.$$

$$h_v^{(l)} = \sigma\left(W^{(l)} \cdot \left[\sum_{u \in N(v)} \alpha_{vu}^{(l)} h_u^{(l-1)} + \alpha_{vv}^{(l)} h_v^{(l-1)}\right]\right)$$