Lecture 4: A General Perspective on Graph Neural Networks

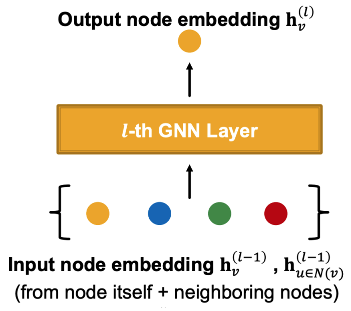
GNNs generalize neural network to graph-structured data instead of grid-like data. GNNs operate on nodes connected through edges in a graph, and they learn how to propagate information across the graph to compute node features. In these graphs, we have

- V is the set of nodes (or vertices).
- E is the set of edges (representing connections).

Each node $v \in V$ has an associated feature vector h_v, and the goal of the GNN is to learn node embeddings that capture both node features and the graph structure. It does this by taking advantage of the computation graph that is based on a node's neighborhood.

1 General Structure of a GNN Layer

The basic GNN layer involves two main operations: **Message Computation/Passing** and **Aggregation**. There are different ways of doing this, like the approaches in GraphSage, GCN, GAT etc.


Output node embedding $\mathbf{h}_v^{(l)}$
l-th GNN Layer
Input node embedding $\mathbf{h}_v^{(l-1)}$, $\mathbf{h}_{u \in N(v)}^{(l-1)}$
(from node itself + neighboring nodes)

When completing these steps, it is important that we consider the node's own embeddings.

1.1 Message Computation/Passing

The general framework for computing a message is:

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

For example, if we used a linear layer, the message passing function computes messages $m_u^{(l)}$ for each node's neighbors based on their features from the previous layer like so:

$$m_u^{(l)} = W^{(l)} h_u^{(l-1)}$$
$$m_v^{(l)} = B^{(l)} h_v^{(l-1)}$$

where:

- $W^{(l)}$ is a trainable weight matrix specific to layer l.
- $B^{(l)}$ is a trainable weight matrix specific to layer l and node the desired node
- $h_u^{(l-1)}$ is the feature vector of neighbor node u from the previous layer $l-1$.
- $h_v^{(l-1)}$ is the feature vector of node v from the previous layer $l-1$.

1.2 Aggregation Operation

After computing messages from neighbors, these messages are aggregated to update the target node's representation. Note that we include the aggregate from the node itself:

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

where:

- $N(v)$ represents the set of neighbors of node v.
- $\text{AGG}(\cdot)$ is an aggregation function such as summation, mean, or max.

1.2.1 Choice of Aggregation Function

- **Sum**: The most expressive and common choice.

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

  This method accumulates information, maintaining an invariant transformation under the permutation of neighbors.

- **Mean**: Normalizes by the degree to handle nodes with varying neighbor counts.

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

  This method is beneficial when graphs have high degree variance but may reduce the expressive power compared to summation.

- **Max**: Takes the element-wise maximum across neighbors, highlighting the most prominent feature.

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

1.3 Nonlinearity

We can add a nonlinearity to either the message or aggregation parts to expressiveness. This can often be written as sigma. Examples of such none nonlinearity include ReLU and Sigmoid. For example,

$$h_v^{(l)} = \mathbf{ReLU}(\mathbf{CONCAT}\left(\text{AGG}\left(\{m_u^{(l)} : u \in N(v)\}\right), m_v^{(l)}\right))$$

We apply activation to i-th dimension of embedding x. ReLU takes the max of a value and 0, and is the most commonly used activation function. Sigmoid is used only when you want to restrict the range of your embeddings. Parametric ReLU is like ReLU but has a min term and has a trainable parameter. Its equation is as following (a_i is a trainable parameter):

$$\mathbf{PReLU}(x_i) = max(x_i, 0) + \alpha_i min(x_i, 0)$$

2 Graph Convolutional Networks (GCNs)

GCNs extend the above structure by incorporating node degree normalization to stabilize training:

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

Please note, in the GCN paper, they use a slightly different normalization.

3 GraphSAGE (Sample and Aggregation)

GraphSAGE uses a sampling-based aggregation approach to scale better with large graphs.

1. **Aggregation of Neighbor Information**:

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

2. **Combination with Node's Own Information**:

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

We use many types of neighbor aggregation. For example, we can use Mean, Pool, and LSTM. Pool transforms neighbor vectors and applies symmetric vector function Mean(*) or Max(*). It transforms it by using an MLP:

$$\text{AGG} = \text{Mean}(\text{MLP}(h_u^{(l-1)}), \forall u \in N(v))$$

We can also apply LSTM to reshuffled of neighbors:

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

In GraphSAGE, we can also apply an optional L2 normalization to h_v^l at every layer. Without l2 normalization the embedding vectors have different scales for vectors, so after this normalization, they will have the same l2 norm. The normalization looks like:

$$h_v^{(l)} \leftarrow \frac{h_v^l}{||h_v^l||_2} \forall (v) \in V \text{where} ||u||_2 = \sqrt{\sum_i u_i^2}$$

4 Graph Attention Networks (GATs)

GATs introduce attention weights α_{vu} for neighbor messages. We define this new parameter a_{vu} for nodes u and v which helps us gain any idea of attention. In GCN and GraphSAGE, we can think of a_{vu} as equal to $\frac{1}{|N(v)|}$. In GAT, we understand that not all node embeddings are equally important, so we can have changing a_{vu} value based on the nodes. How do we get these attention values? We implicitly specify different weights to different nodes in a neighborhood.

If we have an attention mechanism a, we can compute attention coefficients e_{vu} via:

$$e_{vu} = a(m_u^{(l)}, m_v^{(l)})$$

The attention score e_{vu} is passed through a softmax to obtain α_{vu}:

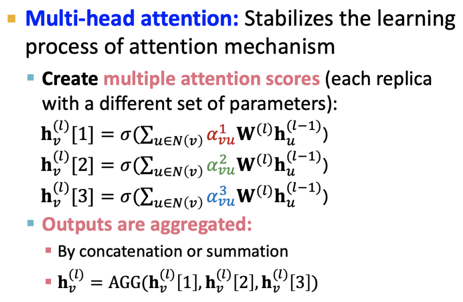
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

The node update equation is:

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

This approach is agnostic to the form of the attention mechanism. We know that parameters of a are trained jointly.

We also have an idea of Multi-head attention that stabilizes the learning process of attention.



- **Multi-head attention:** Stabilizes the learning process of attention mechanism
  - **Create multiple attention scores** (each replica with a different set of parameters):
    $\mathbf{h}_v^{(l)}[1] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^1 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$
    $\mathbf{h}_v^{(l)}[2] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^2 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$
    $\mathbf{h}_v^{(l)}[3] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^3 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$
  - **Outputs are aggregated:**
    - By concatenation or summation
    - $\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$

Benefits of Attention Mechanism:

1. Allows for implicitly specifying different importance values to different neighbors.

2. Computationally efficient

3. Storage efficient

4. Localized

5. Inductive Capability

5 GNN Layers in Practice

Many of the modern deep learning modules can be incorporated into a GNN layer.

We can use Batch Normalisation to stabilize neural network training. Specifically, given a batch of inputs, we can re-center the node embeddings into zero mean and re-scale the variance into unit variance.

We can use Dropout to prevent overfitting. During training, we some probability p, randomly set neurons to zero. During testing, we must use all the neurons for computation.

We can also use Attention/Gating and any other useful deep learning modules.

6 Putting it all Together: Stacking of the Layers

Now that we know what one layer looks like, we can just put layers together sequentially to make a network. We can also add skip connections.

6.1 The Over-Smoothing Problem

The issue of stacking many GNN layers is that over-smoothing may occur. That is when all the node embeddings converge to the same value. We can use Receptive Field Analysis to see this. The receptive field of a node in a GNN is the set of nodes whose information influences the node after k layers. For a k-layer GNN, the receptive field includes all nodes within k hops. So, if we stack many GNN layers, nodes have a higher overlapped receptive field, and then node embeddings will be similar, and we will suffer from over-smoothing. How do we circumvent this issue?

First, we can make the expressive power within each GNN layer as high as possible. This could look like making aggregation/transformation a deep neural network. Or, we can use Skip Connections.

6.2 Skip Connections

Skip Connections increase the impact of earlier layers on the final node embeddings, by adding shortcuts in GNNs. This helps as node embeddings in earlier GNN layers can sometimes better differentiate nodes. This way, we get a mixture of shallow and deep GNNs. Here is an example skip connection equation:

$$h_v^{(l)} = f(h_v^{(l-1)}) + h_v^{(0)}$$

7 Graph Manipulation Techniques

So far we have only thought of our raw input graph to be our computational graph. But this isn't necessarily the best. Input graphs lack features, and can be too sparse/dense or too large. Thus, we have a couple of ways of augmenting our input graphs to make them better to run GNNs on.

- **Virtual Edges**: Connect 2-hop neighbors via virtual edges or use adjacency matrix A + A^2.
- **Virtual Nodes**: Introduce nodes that connect to all nodes, effectively reducing graph diameter and enhancing message passing.
- **Neighbor Sampling**: For dense graphs, a subset of neighbors is selected to optimize computational efficiency.