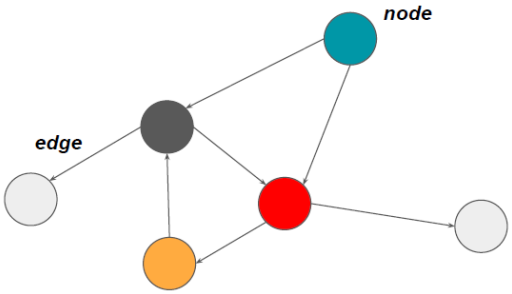
Chapter 10 – Graph Neural Networks

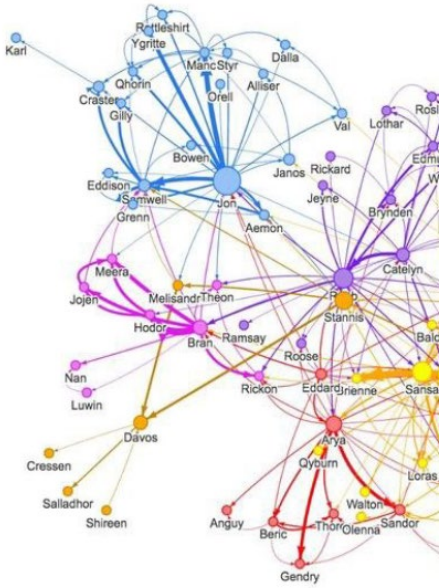
Author: Gianmarco Scarano

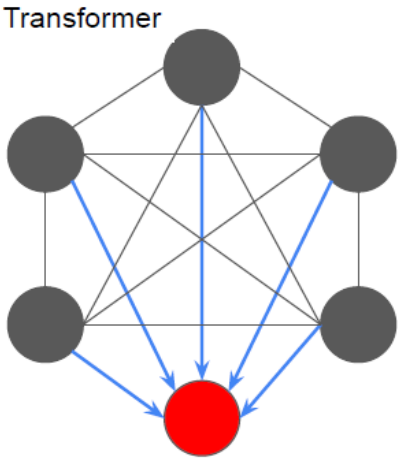
gianmarcoscarano@gmail.com

1. Introduction

Actually, everything can be seen as a Graph, which is a combination of Nodes and Edges:







As per notations, a Graph can be defined as $G = (V, E)$ where V are **Vertexes** and E are **Edges**.

We also define $W \in \mathbb{R}^{|V| \, x \, |V|}$ as the **Adjacency Matrix**.

The Goal for a Graph is to learn low-dimensional vector representations (which are actually embeddings) for nodes in the graph, such that very important properties of the graph itself are preserved in the embedding space *(an example could be: Two nodes which have similar connections in G would be close in the embedding space)*.

For this reason, we define $Z \in \mathbb{R}^{|V| \, x \, d}$ as the **Node Embedding Matrix** $(d \ll |V|)$.

Graphs may have node attributes (for example *gender, age, etc.*) which can be referred as *Node Features*, defined as $x \in \mathbb{R}^{|V| \, x \, d_0}$ where d_0 is the input feature dimension. Of course, we can collect this data into a matrix which we'll call **Node Feature Matrix.**

Given an *Adjacency Matrix* and a *Node Feature Matrix*, the final goal is, then, to **learn** a mapping $\boldsymbol{W}, \boldsymbol{X} \rightarrow \boldsymbol{Z}$ where $\boldsymbol{Z}$ is the *Node Embedding Matrix*.

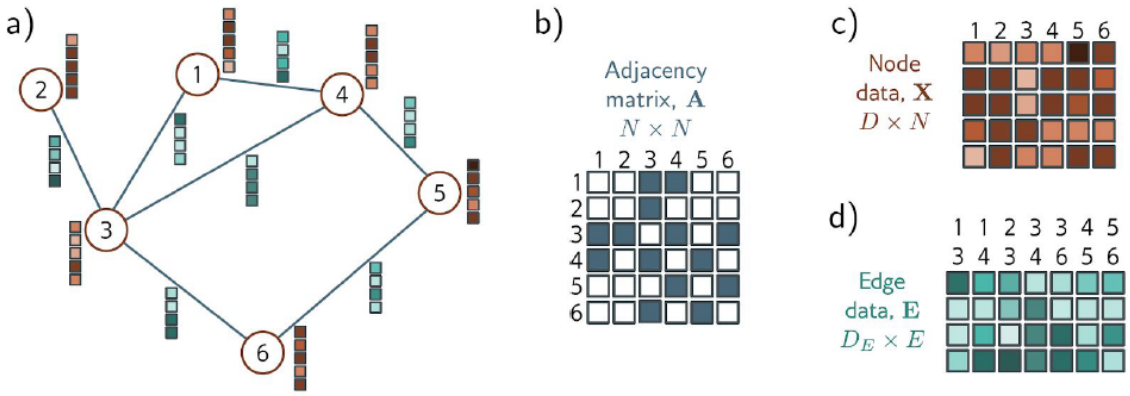


Figure 13.3 Graph representation. a) Example graph with six nodes and seven edges. Each node has an associated embedding of length five (brown vectors). Each edge has an associated embedding of length four (blue vectors). This graph can be represented by three matrices. b) The adjacency matrix is a binary matrix where element (m, n) is set to one if node m connects to node n. c) The node data matrix $\mathbf{X}$ contains the concatenated node embeddings. d) The edge data matrix $\mathbf{E}$ contains the edge embeddings.

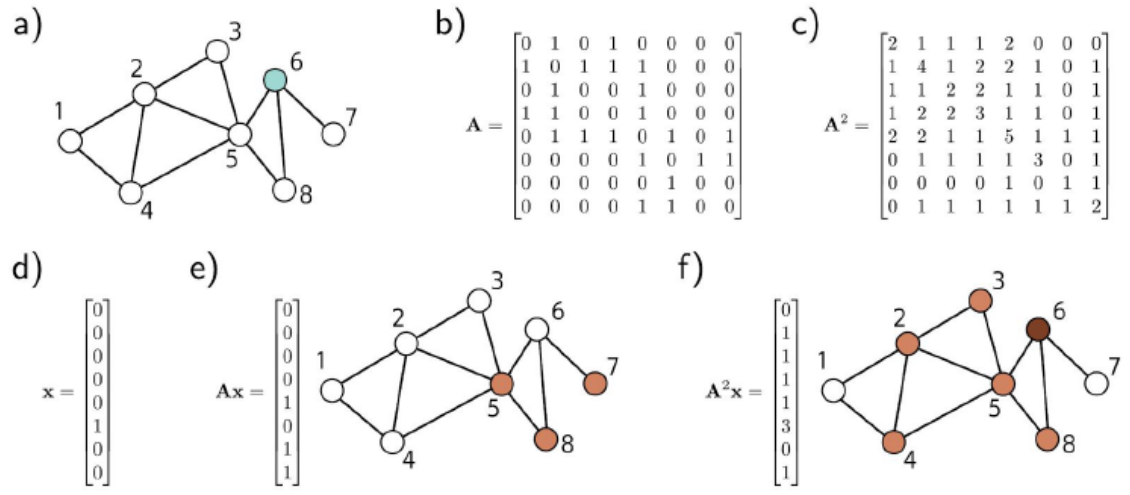


Figure 13.4 Properties of the adjacency matrix. a) Example graph. b) Position (m, n) of the adjacency matrix $\mathbf{A}$ contains the number of walks of length one from node m to node n. c) Position (m, n) of the squared adjacency matrix $\mathbf{A}^2$ contains the number of walks of length two from node n to node m. d) One hot vector representing node six, which was highlighted in panel (a). e) When we pre-multiply this vector by $\mathbf{A}$, the result contains the number of walks of length one from node six to each node; we can reach nodes five, seven, and eight in one move. f) When we pre-multiply this vector by $\mathbf{A}^2$, the resulting vector contains the number of walks of length two from node six to each node; we can reach nodes two, three, four, five, and eight in two moves, and we can return to the original node in three different ways (via nodes five, seven, and eight).

Without going into details, we can have a **Permutation Matrix** P which simply changes the order of the nodes in the graph:

$$X' = X \cdot P$$

$$A = P^T \cdot A \cdot P$$

2. Graph Neural Networks

Given a Graph with **Adjacency matrix** A and **Data Matrix** X we pass it through a series of **GNN Layers.**

Each Layer creates a **Hidden** representation of G that we define as $\boldsymbol{H_k}$ for $k = 1, \dots, K$.

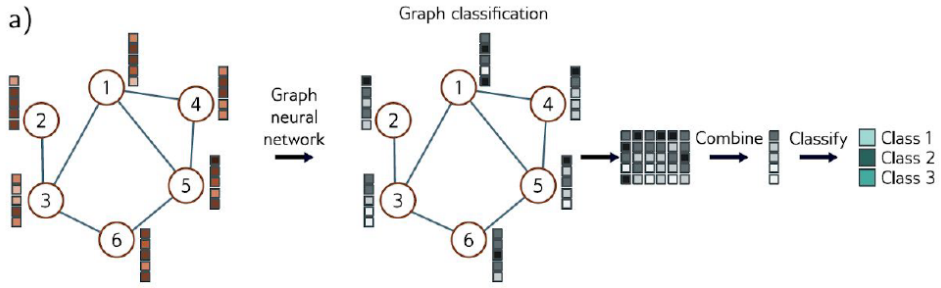
Each column of $\boldsymbol{H_k}$ contains information about that specific node $\boldsymbol{k}$ and its context.

In the context of a GNN, the 3 possible tasks are the following:

- Graph Classification
  - In this case, the network assigns a label exploiting both the structure and node embeddings. One example could be predicting the temperature at which a molecule becomes liquid or whether a molecule is poisonous to human beings or not. Formally speaking:

$$Pr(y = 1|\mathbf{X}, \mathbf{A}) = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{1}/N\right]$$

    Where the parameter β and ω are learned parameters from GNN which then get multiplied by the embedding matrix $H_k \cdot 1/N$ (combine as can be seen below)

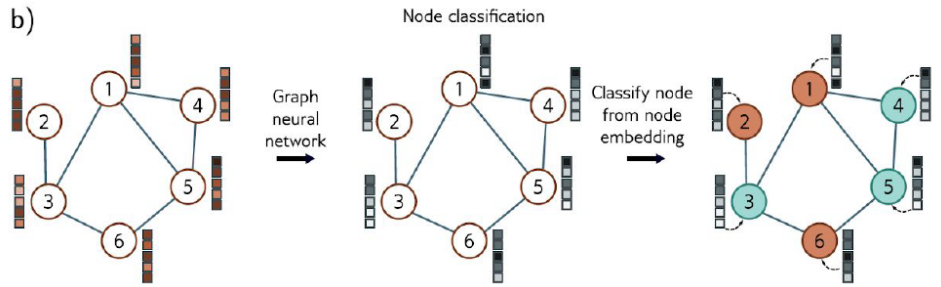


- Node Classification

  - For example, given a graph constructed from a 3D Point Cloud, the goal might be to classify the nodes according to whether they belong to any part of the object in the cloud.

$$Pr(y^{(n)} = 1|\mathbf{X}, \mathbf{A}) = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{h}_K^{(n)}\right]$$

    In this case, we can see that we treat only the h_k value and not the whole Embedding Matrix H_k.



- Edge Classification
  - The network predicts whether or not there should be an edge between nodes n and m.
  - This is common in Social Network, since the network might predict whether two people know and like each other and suggest that they should connect if that is the case:

    $$Pr(y^{(mn)} = 1|\mathbf{X}, \mathbf{A}) = \text{sig}\left[\mathbf{h}^{(m)T}\mathbf{h}^{(n)}\right]$$

    As we can see from this formula, we take the single h-value from node m and n.

Convolutions in Graphs aim at updating each node by aggregating information from nearby nodes (in such a way that they prioritize information from neighbors) using the original graph structure.

Each Layer of GCN is a function $F[\cdot]$ with parameters Φ that takes the node embeddings H and adjacency matrix A and outputs new node embeddings.

$$\begin{aligned}
\mathbf{H}_1 &= \mathbf{F}[\mathbf{X}, \mathbf{A}, \phi_0] \\
\mathbf{H}_2 &= \mathbf{F}[\mathbf{H}_1, \mathbf{A}, \phi_1] \\
\mathbf{H}_3 &= \mathbf{F}[\mathbf{H}_2, \mathbf{A}, \phi_2] \\
\vdots &= \vdots \\
\mathbf{H}_K &= \mathbf{F}[\mathbf{H}_{K-1}, \mathbf{A}, \phi_{K-1}]
\end{aligned}$$

- **X** is the input
- **A** is the adjacency matrix
- $\mathbf{H}_k$ is the k-th layer node embedding
- ϕ_k denotes the parameters that map from layer k to layer k+1

$$\mathbf{agg}[n, k] = \sum_{m \in \text{ne}[n]} \mathbf{h}_k^{(m)}$$

ne[n] == neighbors of node n

Then, we apply a linear transformation Ω_k transformation to the embedding $h_k^{(n)}$ at the current node and further applying a bias term β_k. Finally, we simply pass this result through a non-linear activation function $\boldsymbol{a}[\cdot]$.

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \cdot \mathbf{h}_k^{(n)} + \boldsymbol{\Omega}_k \cdot \mathbf{agg}[n, k]\right]$$

Furthermore, we can collect node embeddings into the matrix H_k and post-multiply by the adjacency matrix A, having $\Omega_k H_k A$, where the n-th column is exactly $agg[n, k]$.

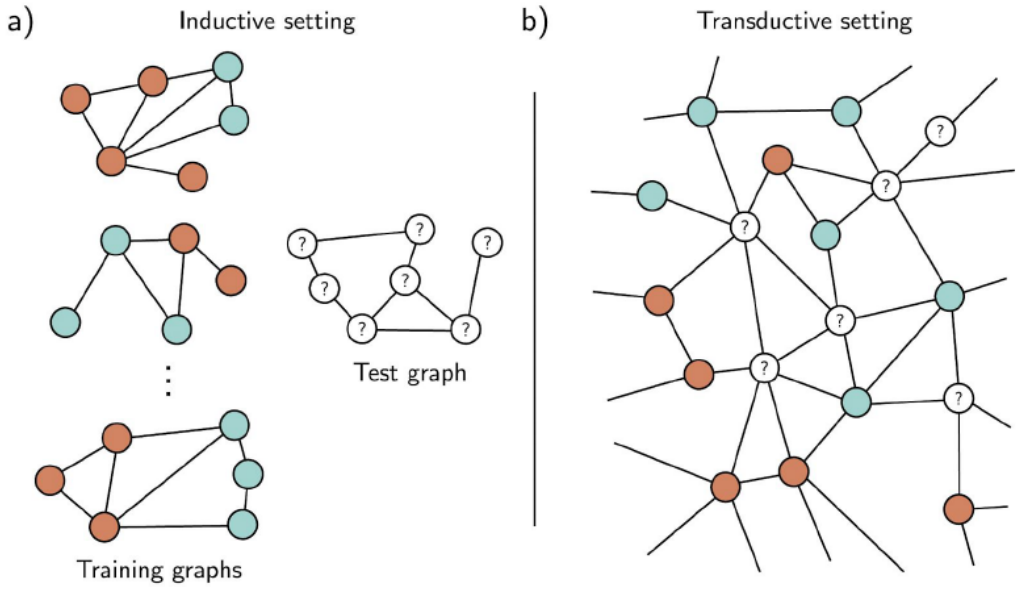
$$\begin{aligned}
\mathbf{H}_{k+1} &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k + \boldsymbol{\Omega}_k \mathbf{H}_k \mathbf{A}\right] \\
&= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k (\mathbf{A} + \mathbf{I})\right],
\end{aligned}$$

The final and total formula then, is given by:

$$\text{f}[\mathbf{X}, \mathbf{A}, \boldsymbol{\Phi}] = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{1}/N\right]$$

2.2 Batching & Computational problems

If we consider batching graphs of different shapes and sizes, we'd have a single graph made up of B connected components (in this case we wouldn't have any pooling):



There are some drawbacks though, such as being logistically difficult to train a GNN of such a big size (due to the fact that we store node embeddings at every network layer) and that we only have a single graph so it's not obvious how to properly perform stochastic gradient descent.

Before looking at other two approaches, let's define the **Receptive Field**:

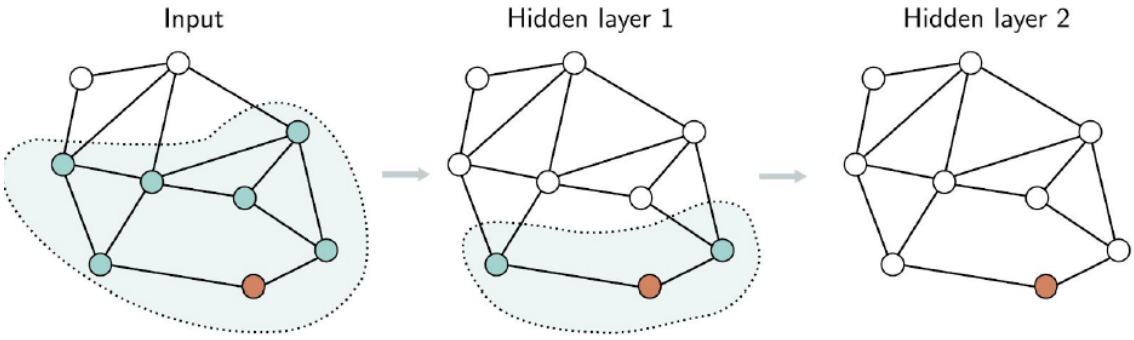


Figure 13.9 Receptive fields in graph neural networks. Consider the orange node in hidden layer two (right). This receives input from the nodes in the 1-hop neighborhood in hidden layer one (shaded region in center). These nodes in hidden layer one receive inputs from their neighbors in turn, and the orange node in layer two receives inputs from all the input nodes in the 2-hop neighborhood (shaded area on left). The region of the graph that contributes to a given node is equivalent to the notion of a receptive field in convolutional neural networks.

Other two approaches could be:

- Neighborhood sampling:
  - We start with the batch nodes and randomly sample a fixed number of their neighbors (the ones present in the previous layer). This process gets iterated such that we have a graph that increases with each layer but in a much-controlled way.

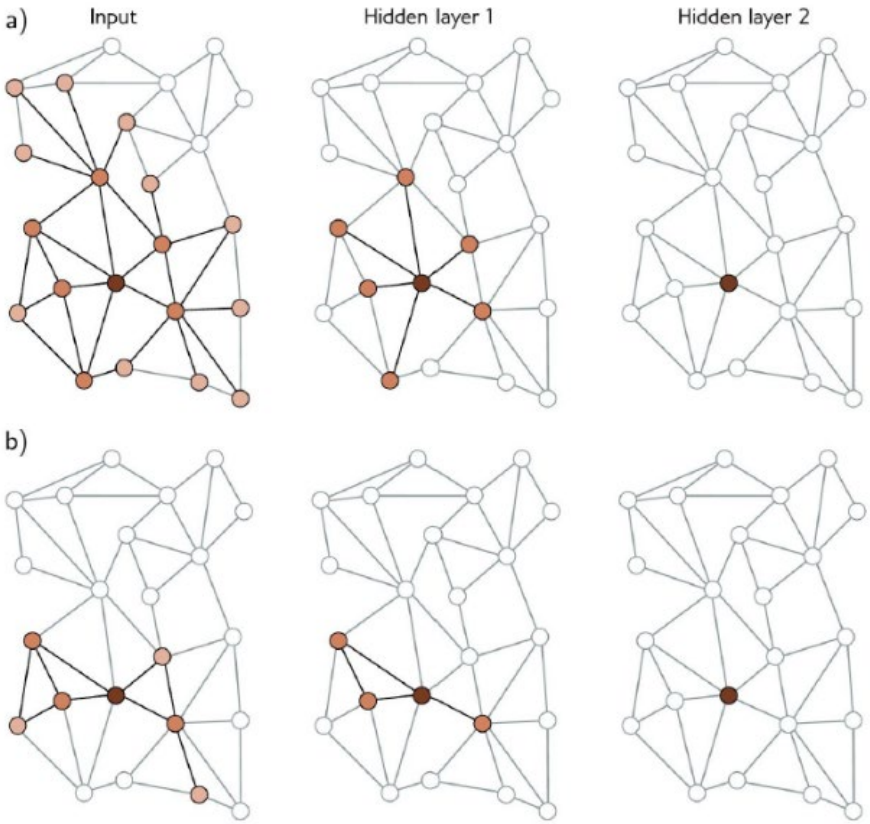
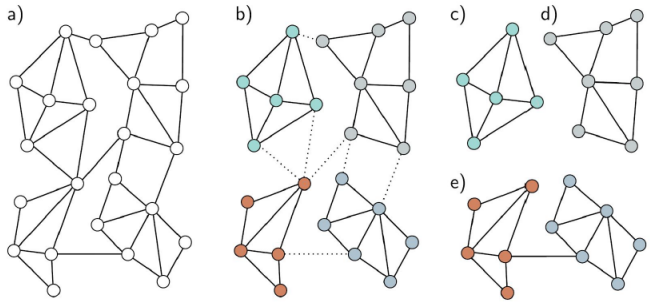


Figure 13.10 Neighborhood sampling. a) One way of forming batches on large graphs is to choose a subset of labeled nodes in the output layer (here, just one node in layer two, right) and then working back to find all of the nodes in the K-hop neighborhood (receptive field). Only this sub-graph is needed to train this batch. Unfortunately, if the graph is densely connected, this may retain a large proportion of the graph. b) One solution is neighborhood sampling. As we work back from the final layer, we select a subset of neighbors (here, three) in the layer before and a subset of the neighbors of these in the layer before that. This restricts the size of the graph for training the batch. In all panels, the brightness represents the distance from the original node.

- Graph partitioning:



2.3 Aggregation

Let's recall that when we defined the GCN Layer, we combined the aggregated neighbors HA with the current nodes H just by summing them up (It's the $(A + I)$ part).

Now, if we add a factor $1 + \epsilon_k$ (which is different for each layer) before summing up, we are actually able to learn this scalar (**Diagonal Enhancement**).

With **Residual Connection**, the aggregated representation from the neighbors is transformed and passed through the activation function before concatenating with the current node.

$$\mathbf{H}_{k+1} = \begin{bmatrix} \mathrm{a}\left[\beta_k 1^T + \Omega_k \mathbf{H}_k \mathrm{A}\right] \\ \mathbf{H}_k \end{bmatrix}$$

Often, it's also convenient taking the average of the neighbors rather than the sum (**Mean Aggregation**).

In **Kipf Normalization**, we down-weight nodes with a large number of neighbors. The idea is that if a node has many connections, its influence on the sum should be reduced to avoid overwhelming the unique information. Nodes with numerous connections might provide less unique information because they are highly connected, hence, down-weighting them ensures that each connection contributes in a fair way.

Finally, let's focus on the **Aggregation by Attention**.

We know that the weights of the Graph Attention Layers depend on the data at the nodes. A linear transformation is applied to these node embeddings. Think of it as a way to adjust and mix the information at each node.

For each pair of nodes, we calculate their similarity using a mathematical operation called the dot product. It helps us understand how much the information at one node aligns with the information at another node.

All these similarity values are stored in a matrix S, where each element represents the similarity between every pair of nodes. Since we want to focus on certain nodes more than others, one uses the SoftMax operation on S which normalizes the values.

Moreover, a function called SoftMask is applied which still uses the SoftMax operation but sets attention values to 0 for nodes that are not neighbors.

The attention weights are then applied to the transformed node embeddings, meaning that certain nodes get more focus in the aggregation process. The result is a new set of node embeddings where information from neighboring nodes is given more importance, thanks to the attention mechanism.

2.4 Graph Convolution Framework

We are looking for graph/node/edge embeddings and we would like to build them through this model.

At the beginning, each node is composed by a vector X (input features) , then we update node embeddings by means of Graph Convolutional Layers.

In this framework, these layers are composed by three main objects:

- Patch functions:
  - It's like the filter of CNN. They define the shape of convolutional filters (in Graphs, they specify which nodes interact with each other at every step of the convolution. Shape is VxV which is number of nodes times number of nodes).
- Convolution Filters' Weights
  - During the process, the weights of the convolutions (per each layer) are learnt in order to compute a message (denoted as m_k^{l+1} - *score*) from layer l to layer $l+1$ starting from the aggregation of the neighborhood in a proper way.

  - For each layer ℓ, we have K matrices of dimensions d_ℓ x $d_{\ell+1}$

  $$(\Theta_1^\ell, \ldots, \Theta_K^\ell)$$

  - At every layer, hidden representations H' are convolved with every patch using the convolution filter weights:

  $$m_k^{\ell+1} = f_k(W, H^\ell) H^\ell \Theta_k^\ell \quad \text{for} \quad 1 \le k \le K$$

- Merging functions:
  - From the previous point, we have K messages (1 per filter) and the way we aggregate them is given by the merging function (sum, average, concatenation, etc.).

Once we specified this framework, we can simply specify the shape of these 3 components in order to have different models. For example, the Attention mechanism specified in 2.3 creates the **GAT** model.

<u>Everything that comes after here it's not going to be asked at the exam as said by Professor.</u>

2.4.1 Graph Laplacian

Given a Graph G with nodes V, edges E and Adjacency Matrix A and a degree matrix D (where for each node, D_{ij} is equal to the degree of (v_i) if $i = j$, 0 otherwise), we define the Graph Laplacian as:

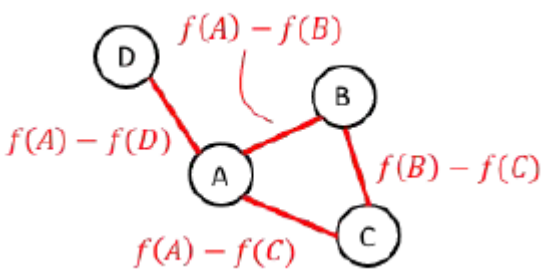
$$L := D - A$$

The Degree Matrix minus the Adjacency Matrix. This simple definition is equivalent to the Laplacian operator on multivariate continuous functions.

2.4.2 Gradients on Graphs

The "magnitude" of a gradient vector defined on a graph, corresponds to the difference in the function's values between the two vertices. Namely, for edge $e_k = (v_i, v_j) \in E$ connecting node v_i to node v_j, its "gradient" can be taken to be:

$$g(e_k) := f(v_i) - f(v_j)$$



The order is important, meaning that if A comes before D the gradient on this edge will be $f(A) - f(D)$

Proof of why L = D – A is at Slide 80/86

(min. 32:00 of lecture)

2.4.3 Spectral Graph Convolution

Spectral methods in graph convolution involve applying convolutions in the "spectral" domain using the Laplacian matrix.

There are two types: Spectrum-based methods (which compute eigen decomposition), and Spectrum-free methods (which use spectral graph theory without explicitly computing eigenvectors).

- Convolution Theorem:
  - The Convolution Theorem says that graph convolutions can be computed either in the spatial domain (using circulant matrices) or in the spectral domain (using Fourier transform).
- Graph Fourier Transform:
  - For non-Euclidean graphs, we use the Fourier transform on the Laplacian matrix to perform convolution in the spectral domain.
- Spectral CNNs:
  - Spectral CNNs generalize convolutions to graphs by learning filters in the spectral domain of the Laplacian matrix. They use the eigenvectors of the Laplacian and learn filters as multipliers on its eigenvalues.
- Drawbacks of Spectral CNNs:
  - Spectral CNNs have scalability issues and are computationally expensive due to requiring eigen decomposition. Filters are eigen-basis dependent and cannot be easily shared across different graphs.
- Spectrum-Free Methods:
  - Spectrum-free methods use polynomial expansions to approximate spectral filters, making them localized and scalable. These filters depend on the graph's adjacency and degree matrices, providing a k-localized receptive field, in the sense that the receptive field of each filter is k, and only nodes at a distance less than k will interact in the convolution operation.

2.4.4 Graph Convolution Networks (GCN)

In GCN, we consider only 2 filters and then (filter-speaking) we always consider one to be the opposite of the other.

Now, we also assume that the largest Eigenvalue is 2, defining then the GCN:

Therefore, this final form doesn't depend on anything but simply the Degree Matrix and the Identity Matrix.

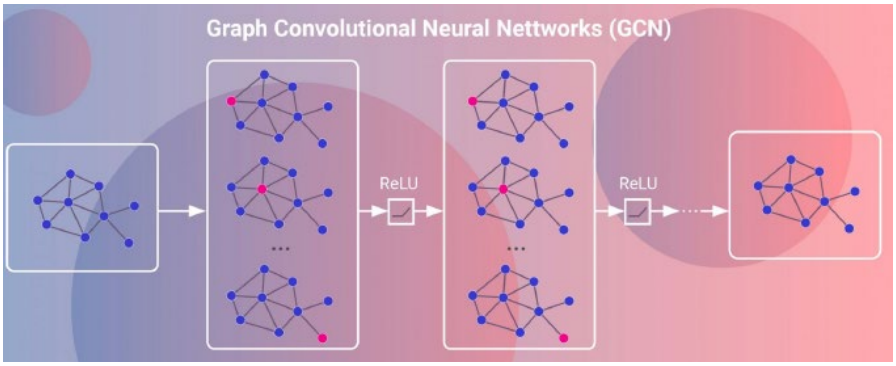
$$H^{\ell+1} = \sigma((2I - \widetilde{L})H^{\ell}\Theta^{\ell})$$
$$= \sigma((I + D^{-1/2}WD^{-1/2})H^{\ell}\Theta^{\ell})$$

Using GCF notation, GCN patch functions can be written as

$$g_1(W, D) = (D + I)^{-1/2}(W + I)(D + I)^{-1/2}$$

and the graph convolution layer is:

$$H^{\ell+1} = \sigma(g_1(W, D)H^{\ell}\Theta^{\ell})$$



We are performing graph convolution where, for each node, we calculate an operator, apply a non-linearity, and repeat the process. As we go through layers, the aggregation process continues. Thanks to the Laplacian matrix, information from neighbors is aggregated to nodes. In each layer, we aggregate information from neighborhoods at increasing distances: 1st layer aggregates from immediate neighbors, 2nd layer from neighbors at distance 2, and so on.