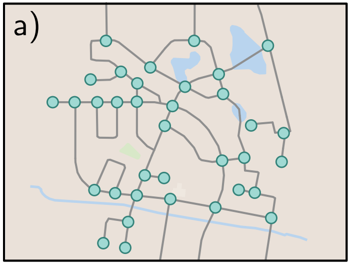
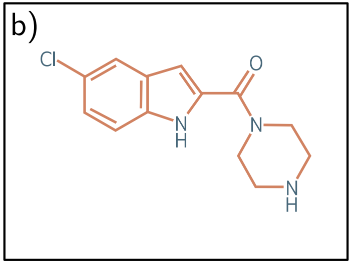
Graph Neural Networks

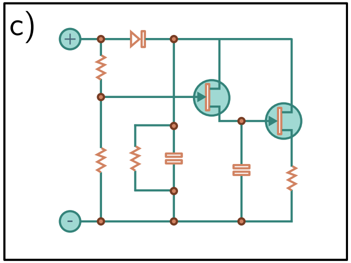
CS 4277 Deep Learning

Kennesaw State University

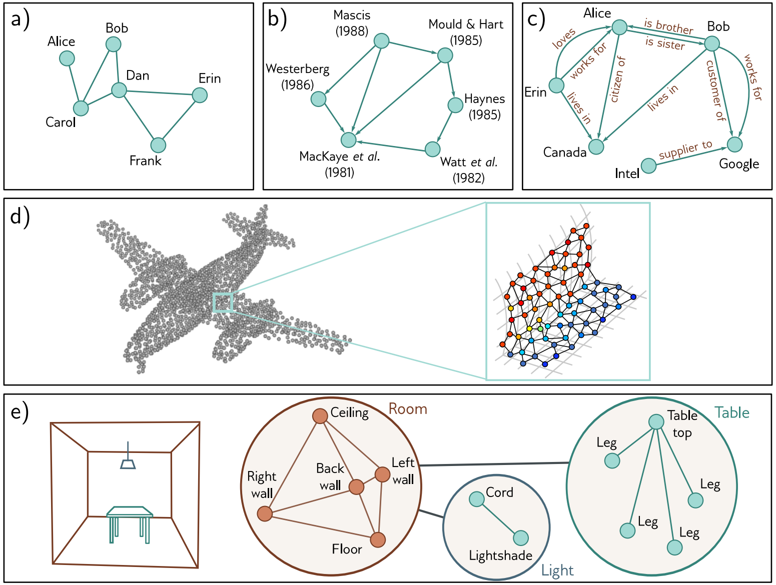
Graphs in the Real World



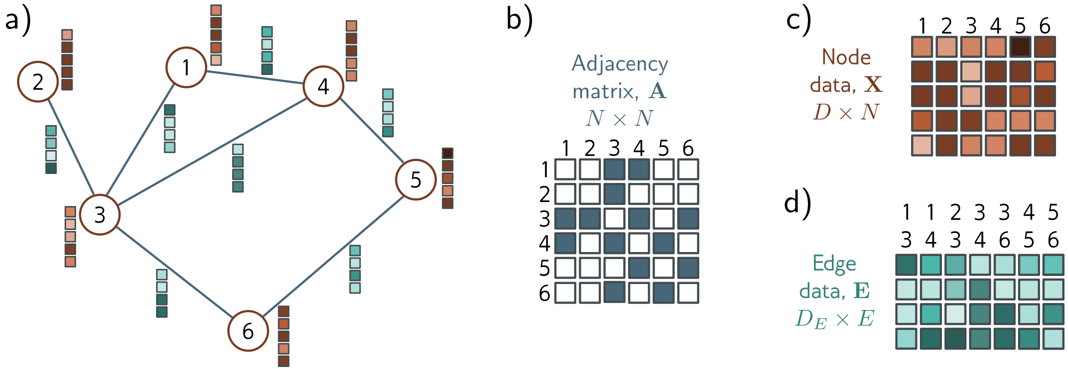




Types of Graphs

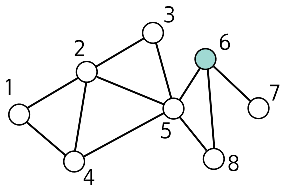


Graph Represenatation



Adjacency Matrices

a)



b)

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

c)

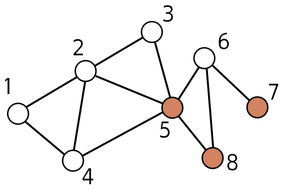
$$\mathbf{A}^2 = \begin{bmatrix} 2 & 1 & 1 & 1 & 2 & 0 & 0 & 0 \\ 1 & 4 & 1 & 2 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 2 & 1 & 1 & 0 & 1 \\ 1 & 2 & 2 & 3 & 1 & 1 & 0 & 1 \\ 2 & 2 & 1 & 1 & 5 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 3 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 2 \end{bmatrix}$$

d)

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

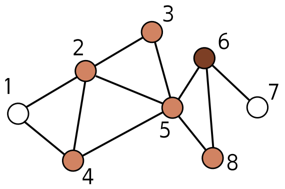
e)

$$\mathbf{Ax} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

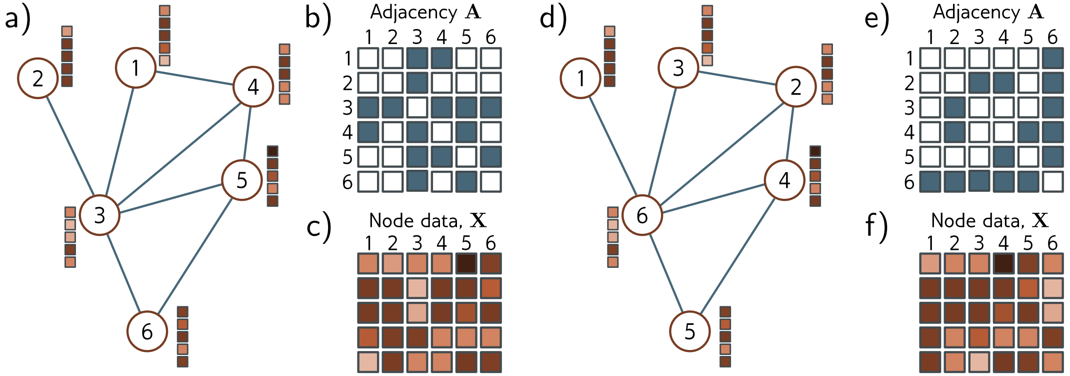


f)

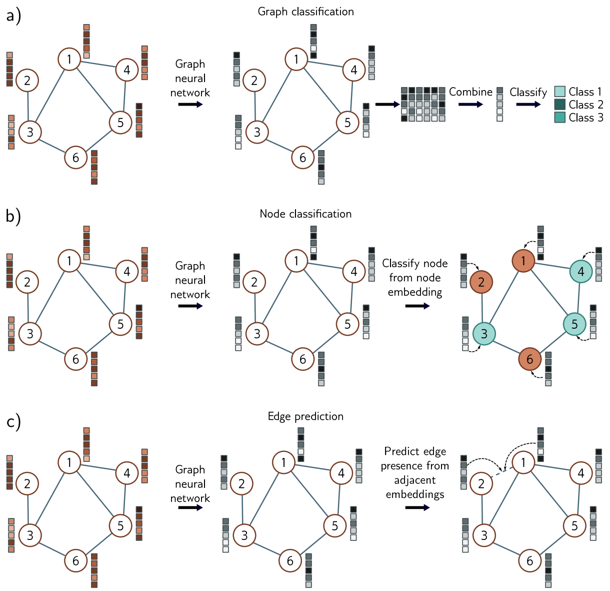
$$\mathbf{A}^2\mathbf{x} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 3 \\ 0 \\ 1 \end{bmatrix}$$



Permuting Node Indices



Graph Network Tasks



Loss Functions for Supervised Graph Problems

Supervised graph problems typically in three categories:

1. Graph-level tasks: assign label or estimate a value from the whole graph.

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

2. Node-level tasks: assign a label or value to each node in the graph.

Loss functions are defined in the same way as for graph-level tasks, except that now this is done independently at each node n:

$$Pr(y^{(n)} = 1|\boldsymbol{X}, \boldsymbol{A}) = \text{sig}(\beta_K + \boldsymbol{\omega}_K \boldsymbol{H}_K^{(n)})$$

3. Edge-prediction tasks: predict the probability that an edge should exist between nodes.

Like binary classification, map node embeddings to single number representing the probability that nodes should be connected.

$$Pr(y^{(mn)} = 1|\boldsymbol{X}, \boldsymbol{A}) = \text{sig}(\boldsymbol{h}^{(m)T} \boldsymbol{h}^{(n)})$$

Graph-Level Loss Functions

$$Pr(y = 1|\boldsymbol{X}, \boldsymbol{A}) = \mathsf{sig}(\beta_K + \frac{\boldsymbol{\omega}_K \boldsymbol{H}_K \mathbf{1}}{N})$$

Node-Level Loss Functions

Loss functions are defined in the same way as for graph-level tasks, except that now this is done independently at each node n:

$$Pr(y^{(n)} = 1|\boldsymbol{X}, \boldsymbol{A}) = \text{sig}(\beta_K + \boldsymbol{\omega}_K \boldsymbol{H}_K^{(n)})$$

Edge Prediction Loss Functions

Like binary classification, map node embeddings to single number representing the probability that nodes should be connected.

$$Pr(y^{(mn)} = 1|\boldsymbol{X}, \boldsymbol{A}) = \text{sig}(\boldsymbol{h}^{(m)T}\boldsymbol{h}^{(n)})$$

Graph Convolutional Networks

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

Equivariance and Invariance

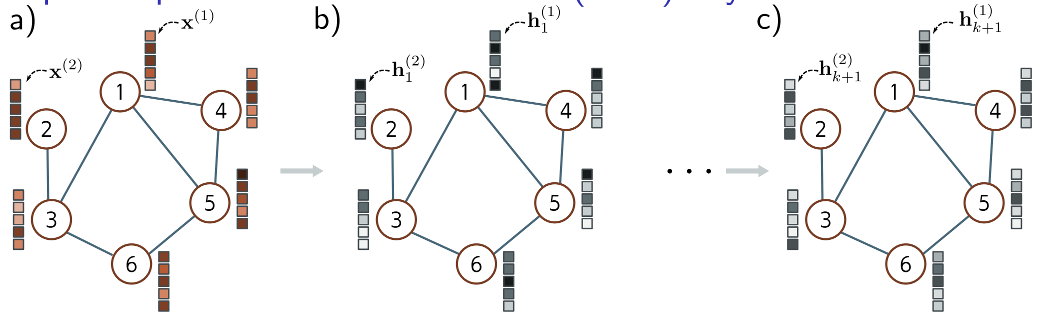
$$\boldsymbol{H}_{k+1}\boldsymbol{P} = \boldsymbol{F}(\boldsymbol{H}_k\boldsymbol{P}, \boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}, \phi_k)$$

$$y = \mathsf{sig}(\beta_K + \frac{\boldsymbol{\omega}_K\boldsymbol{H}_K\boldsymbol{1}}{N}) = \mathsf{sig}(\beta_K + \frac{\boldsymbol{\omega}_K\boldsymbol{H}_K\boldsymbol{P}\boldsymbol{1}}{N})$$

Parameter Sharing

- Convolutional layers for images process each pixel identically
- Convolution updates variables by taking weighted sum of information from neighbors
- Could do similar for graph networks by using same parameters at every node
- Challenge: nodes have differeing numbers of neighbors

Example Graph Convolutional Network (GCN) Layer



$$\mathbf{h}_1^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_1^{(n)} + \boldsymbol{\Omega}_0 \mathbf{agg}[n]\right] \qquad \mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k^{(n)} + \boldsymbol{\Omega}_k \mathbf{agg}[n]\right]$$

Aggregate information from neighboring nodes by summing their embeddings:

$$\mathbf{agg}(n,k) = \sum_{m \in \mathrm{ne}(n)} \boldsymbol{h}_k^{(m)}$$

where $\mathrm{ne}(n)$ is the set of indices of the nieghbors of node n.

Then apply linear transform $\boldsymbol{\Omega}_k$ to embedding $\boldsymbol{h}_k^{(n)}$, add bias $\boldsymbol{\beta}_k$ and pass through nonlinear activation function.

Example: Graph Classification

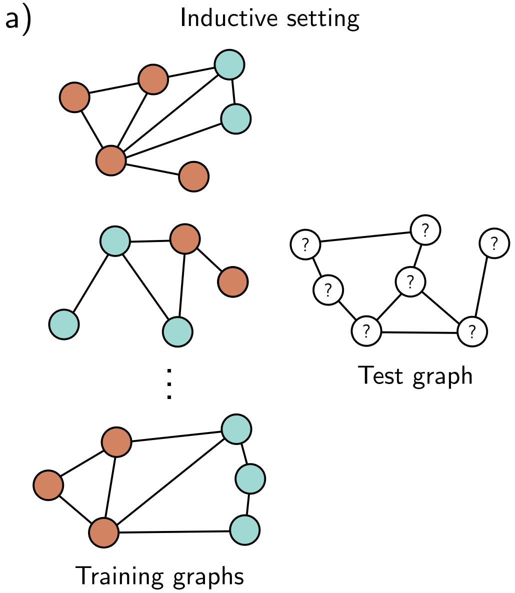
$$
\begin{aligned}
\boldsymbol{H}_1 &= \boldsymbol{a}\big(\beta_0 \mathbf{1}^T + \boldsymbol{\Omega}_0 \boldsymbol{X}(\boldsymbol{A}+\boldsymbol{I})\big)\\
\boldsymbol{H}_2 &= \boldsymbol{a}\big(\beta_1 \mathbf{1}^T + \boldsymbol{\Omega}_0 \boldsymbol{H}_1(\boldsymbol{A}+\boldsymbol{I})\big)\\
\vdots &= \vdots\\
\boldsymbol{H}_K &= \boldsymbol{a}\big(\beta_{K-1} \mathbf{1}^T + \boldsymbol{\Omega}_{K-1} \boldsymbol{H}_{K-1}(\boldsymbol{A}+\boldsymbol{I})\big)\\
f(\boldsymbol{X}, \boldsymbol{A}, \boldsymbol{\Phi}) &= \mathsf{sig}\big(\beta_K + \frac{\boldsymbol{\omega}_K \boldsymbol{H}_K \mathbf{1}}{N}\big)
\end{aligned}
$$

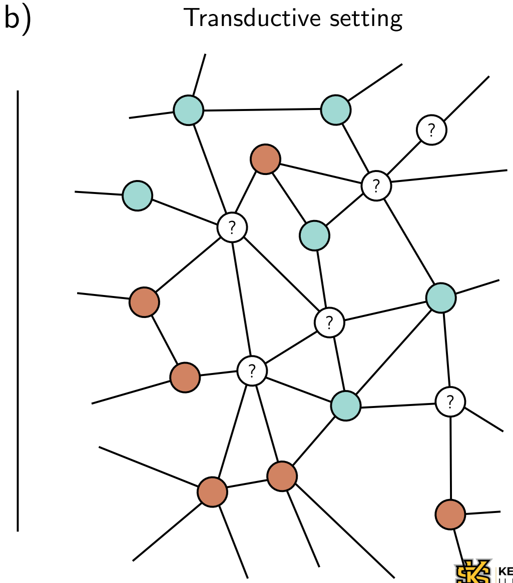
Training with Batches

Given I training graphs $\{\boldsymbol{X}_i, \boldsymbol{A}_i\}$ and their labels y_i, the parameters $\boldsymbol{\Phi} = \{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}_{k=0}^{K}$ can be learned using SGD and the binary cross-entropy loss (equation 5.19). Fully connected networks, convolutional networks, and transformers all exploit the parallelism of modern hardware to process an entire batch of training examples concurrently. To this end, the batch elements are concatenated into a higher-dimensional tensor. However, each graph may have a different number of nodes. Hence, the matrices Xi and Ai have different sizes, and there is no way to concatenate them into 3D tensors.

Luckily, a simple trick allows us to process the whole batch in parallel. The graphs in the batch are treated as disjoint components of a single large graph. The network can then be run as a single instance of the network equations. The mean pooling is carried out only over the individual graphs to make a single representation per graph that can be fed into the loss function.

Inductive vs. Transductive Models

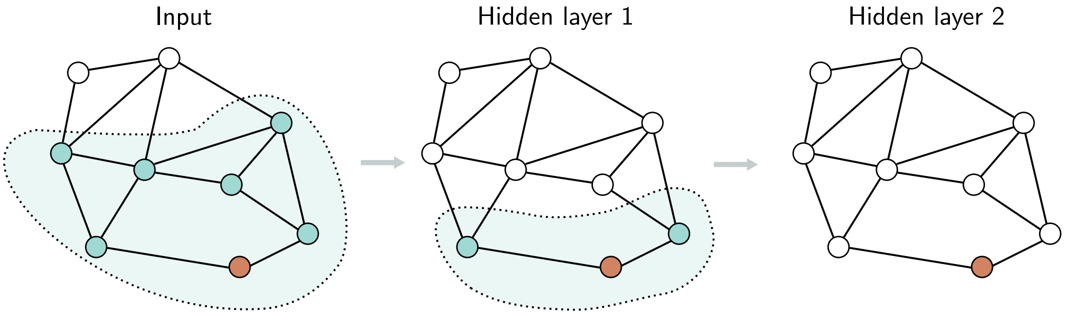




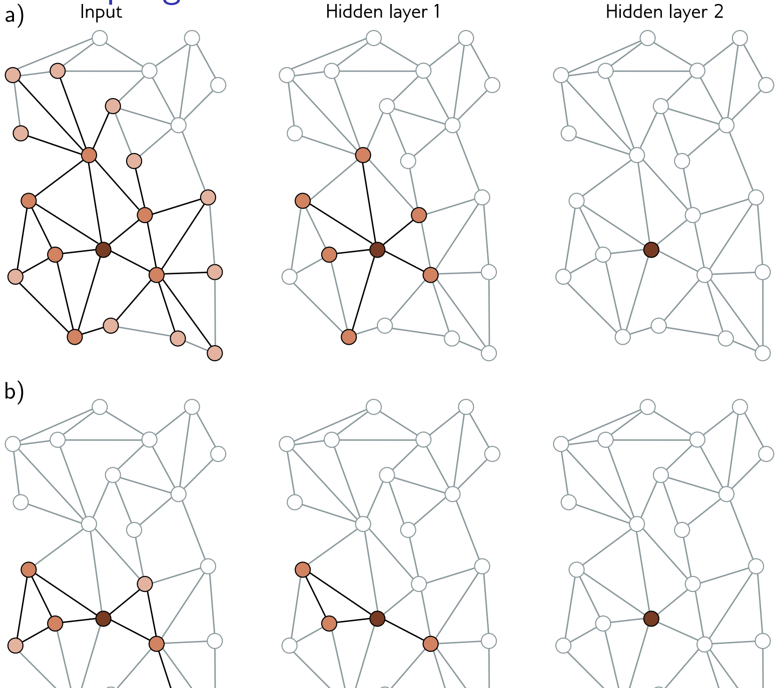
Example: Node Classification

$$f(\boldsymbol{X}, \boldsymbol{A}, \boldsymbol{\Phi}) = \mathsf{sig}\big(\beta_K \mathbf{1}^T + \boldsymbol{\omega}_K \boldsymbol{H}_K\big)$$

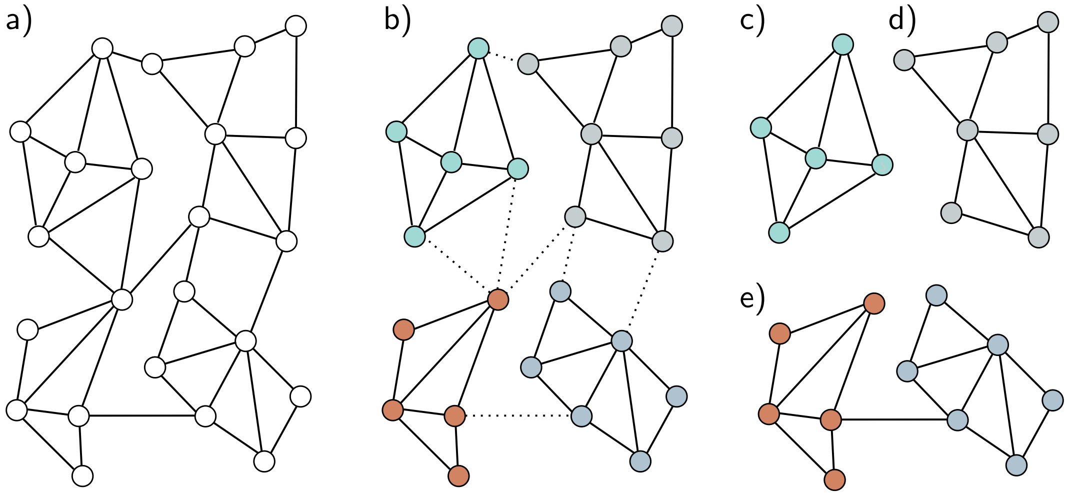
Choosing Bitches



Neighborhood Sampling



Graph Partitioning



Layers for GCNs

Approaches to both (i) the combination of the current embedding with the aggregated neighbors and (ii) the aggregation process itself.

- Combining current node and aggregated neighbors
- Residual connections
- Mean aggregation
- Kipf normalization
- Max pooling aggregation
- Aggregation by attention

Combining Current Node and Aggregated Neighbors

In the example GCN layer above, we combined the aggregated neighbors HA with the current nodes H by just summing them:

$$\boldsymbol{H}_{k+1} = \boldsymbol{a}\big(\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \boldsymbol{H}_k(\boldsymbol{A} + \boldsymbol{I})\big) \tag{13.13}$$

Another option is to multiply the current node by a factor of $(1 + \epsilon_k)$, where ϵ_k is a learned scalar that is different for each layer. This is called *diagonal enhancement*:

$$\boldsymbol{H}_{k+1} = \boldsymbol{a}\big(\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \boldsymbol{H}_k(\boldsymbol{A} + (1 + \epsilon_k)\boldsymbol{I})\big) \tag{13.14}$$

A third option is to apply a different linear transform Φ_k to the current node:

$$\begin{aligned}
\boldsymbol{H}_{k+1} &= \boldsymbol{a}(\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \boldsymbol{H}_k \boldsymbol{A} + \boldsymbol{\Phi}_k \boldsymbol{H}_k) \\
\boldsymbol{H}_{k+1} &= \boldsymbol{a}\left(\boldsymbol{\beta}_k \mathbf{1}^T + [\boldsymbol{\Omega}_k \boldsymbol{\Phi}_k] \begin{bmatrix} \boldsymbol{H}_k \boldsymbol{A} \\ \boldsymbol{H}_k \end{bmatrix}\right) \\
\boldsymbol{H}_{k+1} &= \boldsymbol{a}\left(\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}'_k \begin{bmatrix} \boldsymbol{H}_k \boldsymbol{A} \\ \boldsymbol{H}_k \end{bmatrix}\right) \qquad (13.15)
\end{aligned}$$

where $\boldsymbol{\Omega}'_k = \boldsymbol{\Omega}_k \boldsymbol{\Phi}_k$.

Residual Connections

With residual connections, the aggregated representation from the neighbors is transformed and passed through the activation function before summation or concatenation with the current node. For the latter case, the associated network equations are:

$$\boldsymbol{H}_{k+1} = \begin{bmatrix} \boldsymbol{a}(\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \boldsymbol{H}_k \boldsymbol{A}) \\ \boldsymbol{H}_k \end{bmatrix} \tag{13.16}$$

Mean Aggregation

Average of neighbors instead of sum:

$$\mathbf{agg}(n) = \frac{1}{|\mathsf{ne}(n)|} \sum_{m \in \mathsf{ne}(n)} \boldsymbol{h}_m \tag{13.17}$$

If we introduce an $N \times N$ matrix $\boldsymbol{D}$ in which each non-zero element contains the number of neighbors for the associated node, then each diagonal element in $\boldsymbol{D}^{-1}$ contains the denominator from Equation 13.17. Then the new GCN layer is:

$$\boldsymbol{H}_{k+1} = \boldsymbol{a}\big(\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \boldsymbol{H}_k(\boldsymbol{A}\boldsymbol{D}^{-1} + \boldsymbol{I})\big) \tag{13.18}$$

Kipf Normalization

Include the current node with its neighbors. Sum of node representations normalized as:

$$\mathbf{agg}(n) = \sum_{m \in \mathsf{ne}(n)} \frac{\boldsymbol{h}_m}{\sqrt{|\mathsf{ne}(n)||\mathsf{ne}(m)|}} \tag{13.19}$$

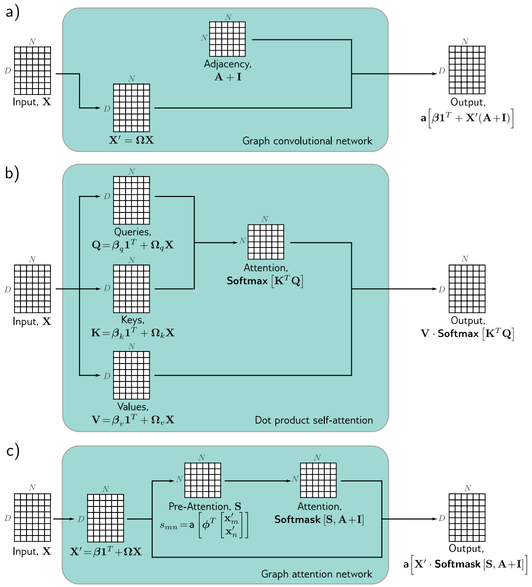
The logic: information from nodes with many neighbors should be down-weighted since connections provide less unique information.

Max Pooling Aggregation

$$\mathbf{agg}(n) = \mathbf{max}_{m \in \mathsf{ne}(n)}(\boldsymbol{h}_m) \tag{13.21}$$

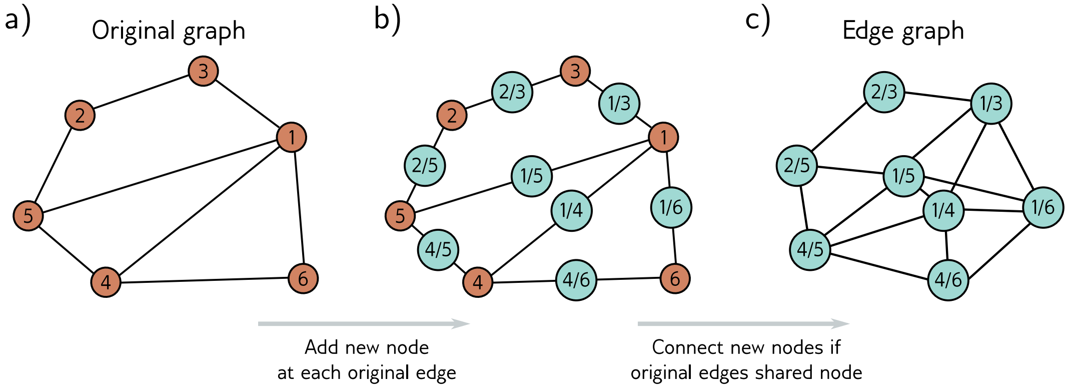
GCNs and Graph Attention Networks

The last form of aggregation we'll discuss is aggregation by attention, summarized here:



a)
Input, $\mathbf{X}$
$\mathbf{X}' = \mathbf{\Omega}\mathbf{X}$
Adjacency, $\mathbf{A}+\mathbf{I}$
Output, $\mathbf{a}\left[\boldsymbol{\beta}\mathbf{1}^T + \mathbf{X}'(\mathbf{A}+\mathbf{I})\right]$
Graph convolutional network
b)
Input, $\mathbf{X}$
Queries, $\mathbf{Q}=\boldsymbol{\beta}_q\mathbf{1}^T+\mathbf{\Omega}_q\mathbf{X}$
Keys, $\mathbf{K}=\boldsymbol{\beta}_k\mathbf{1}^T+\mathbf{\Omega}_k\mathbf{X}$
Values, $\mathbf{V}=\boldsymbol{\beta}_v\mathbf{1}^T+\mathbf{\Omega}_v\mathbf{X}$
Attention, **Softmax** $[\mathbf{K}^T\mathbf{Q}]$
Output, $\mathbf{V}\cdot$ **Softmax** $[\mathbf{K}^T\mathbf{Q}]$
Dot product self-attention
c)
Input, $\mathbf{X}$
$\mathbf{X}'=\boldsymbol{\beta}\mathbf{1}^T+\mathbf{\Omega}\mathbf{X}$
Pre-Attention, $\mathbf{S}$
$s_{mn} = \mathrm{a}\left[\boldsymbol{\phi}^T\begin{bmatrix}\mathbf{x}'_m\\ \mathbf{x}'_n\end{bmatrix}\right]$
Attention, **Softmask** $[\mathbf{S}, \mathbf{A}+\mathbf{I}]$
Output, $\mathbf{a}\left[\mathbf{X}'\cdot\text{Softmask}[\mathbf{S},\mathbf{A}+\mathbf{I}]\right]$
Graph attention network
N
D

Edge Graphs



a)
Original graph
3
2
1
5
4
6
b)
3
2/3
1/3
2
1
2/5
1/5
5
1/4
1/6
4/5
4
4/6
6
c)
Edge graph
2/3
1/3
2/5
1/5
1/4
1/6
4/5
4/6
Add new node
at each original edge
Connect new nodes if
original edges shared node

Closing Thoughts

Boom!