Graph Neural Networks Demystified

Only the essential concepts in Stanford CS224W to get up-to-speed with GNNs
Or just to better understand the material in CS224W

Ng Yen Kaow

Embeddings

- Relatively small vectors associated with each object where similar objects have similar embeddings
- Using the embeddings of graph elements, various tasks can be performed
  - Cluster nodes in a graph
  - Predict properties of a node
  - Predict if two nodes may be connected
  - Classify entire graphs
- To perform each task, use the embedding with a suitable ML method
  - e.g. clustering can be performed with k-means

Obtaining embeddings

- Embeddings can be formed with or learned from features
  - Node-level features
    - Degree
    - Centrality (eigenvector/ betweenness/ closeness)
    - Clustering coefficient
    - Graphlets
    - Structure-based features
  - Link-level features
    - Distance-based features
    - Local/global neighborhood overlap
  - Graph-level features
    - Graph kernels
- Task-independent embeddings can be learned from unsupervised learning

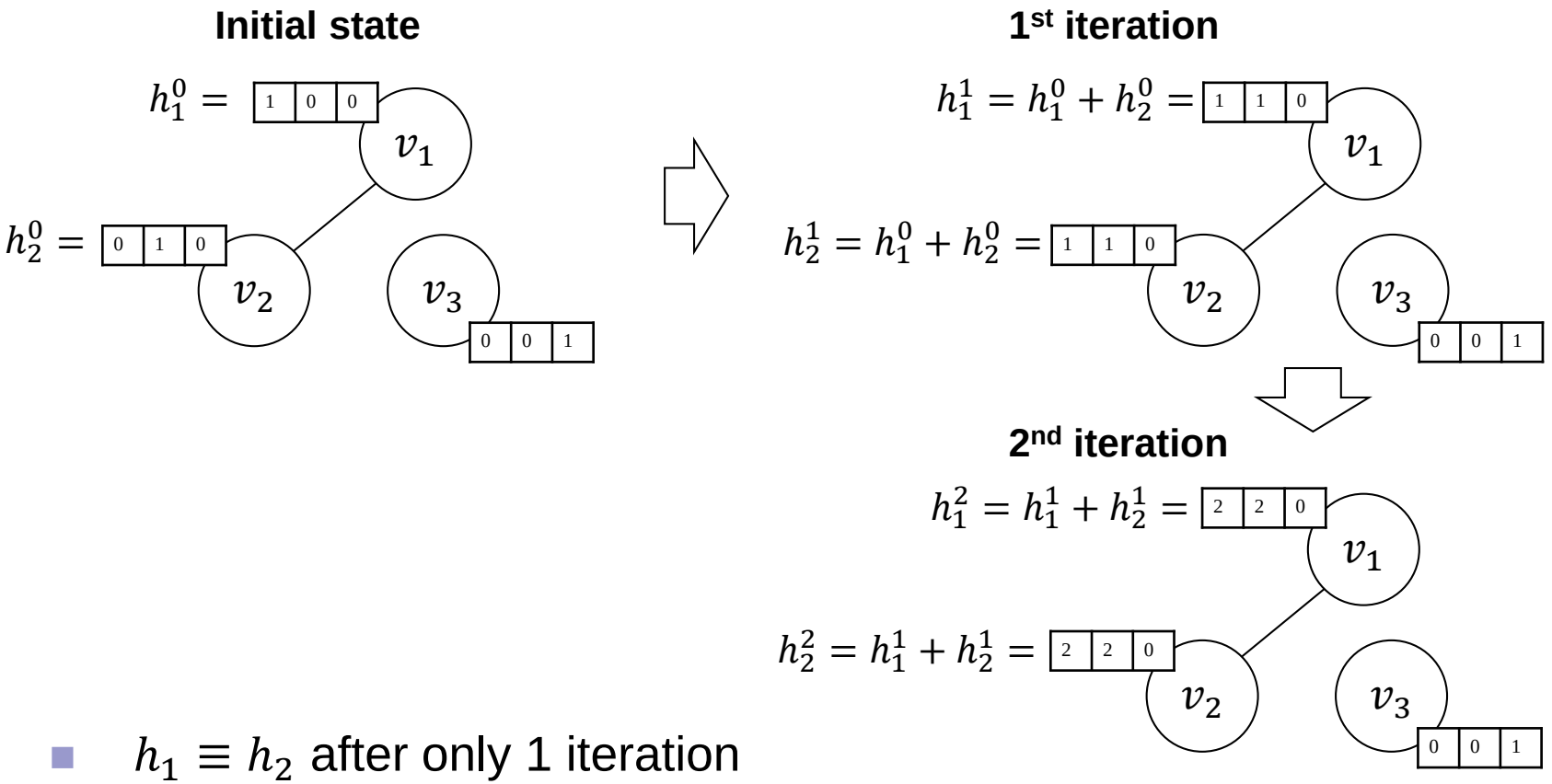
Task-independent embeddings

- Unsupervised extraction by random walks
  - **DeepWalk**
    - Estimate pairwise distance between nodes (hence their co-occurrence probability)
      - Usable for finding product relatedness in recommender
    - Node embeddings
      1. Estimate node distances with random walks
      2. Train a neural network (with node input and embedding output) such that distances between embeddings agree with estimated distances
  - **Anonymous Walk**
    - Embeddings for entire graphs
  - **Simpler method: just add up neighbors**

Embeddings by adding neighbors

- Sum up the features of (self and) neighbor nodes
  - Features of nodes in close proximity will become similar

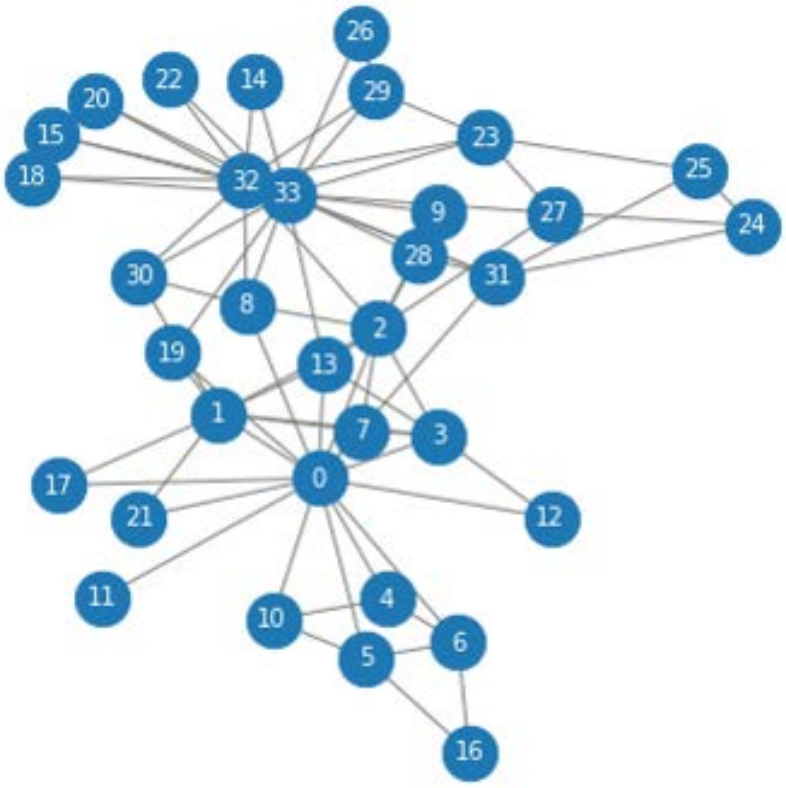
Example: Let h_i^j denote features of node i at iteration j and let $h_1^0 = (1 \quad 0 \quad 0)$, $h_2^0 = (0 \quad 1 \quad 0)$, and $h_3^0 = (0 \quad 0 \quad 1)$



- $h_1 \equiv h_2$ after only 1 iteration

Embeddings by adding neighbors

- To cluster nodes in a graph, will it work if we
  1. Start with a unique feature for each node, and
  2. Repeatedly add up neighboring features, and
  3. Finally, cluster the resultant features with some method like *k*-means?



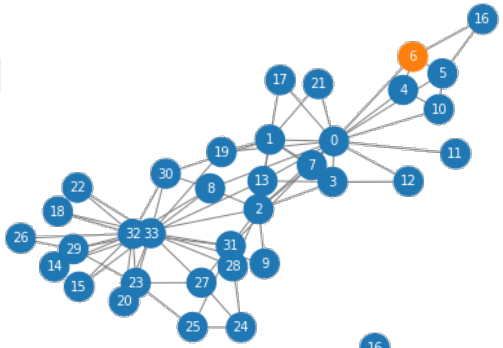
Let's try with karate club network

Based on https://tkipf.github.io/graph-convolutional-networks/

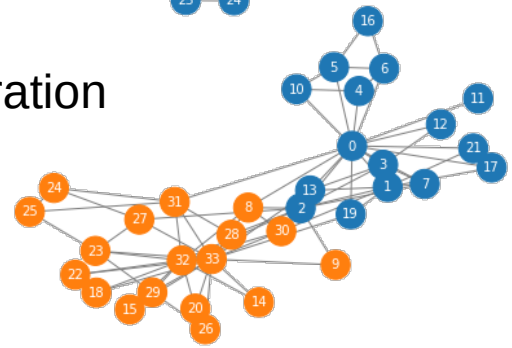
Embeddings by adding neighbors

- Karate club

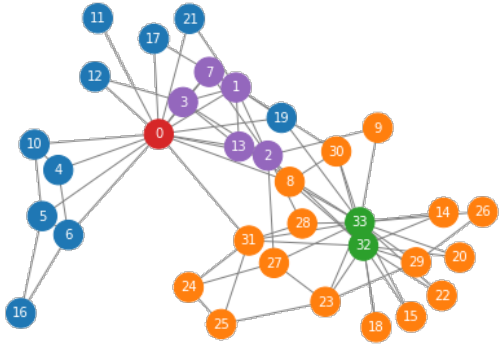
Initial



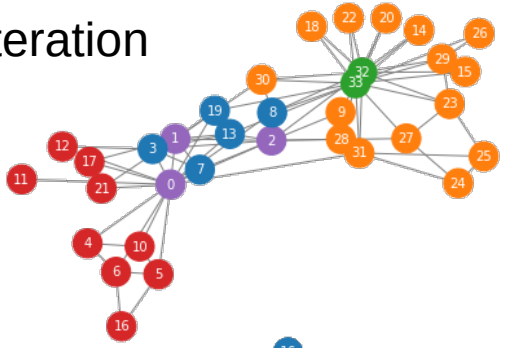
1st iteration



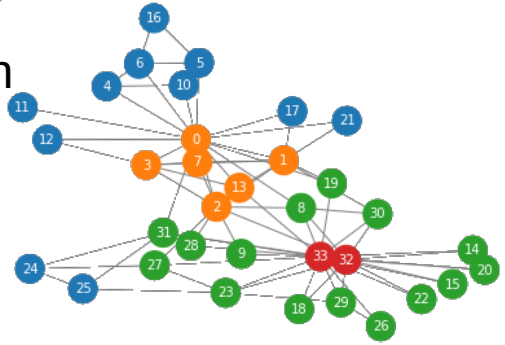
2nd iteration



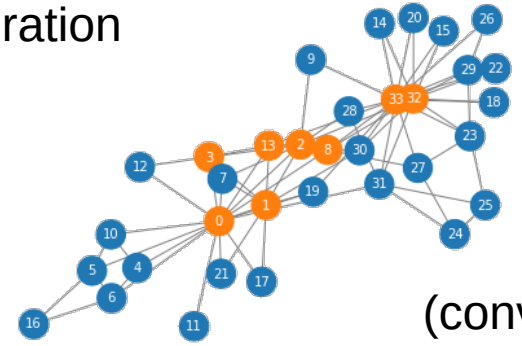
3rd iteration



4th iteration



5th iteration



(converged)

(number of clusters determined with silhouette method)

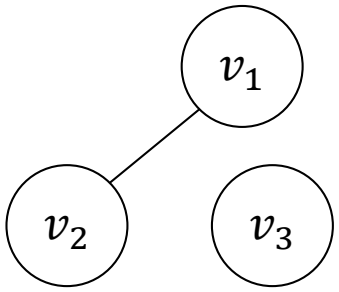
Adding neighbors w/ linear algebra

- Let matrix H be a matrix where each row is a node and each column is a feature
  - H have dim $|V| \times d$
- Let A be an adjacency matrix
  - Let $\hat{A} = A + I$ where I is the identify matrix
- Then, **sum** is simply $\hat{A}H$

Note that node order is not important

$$\begin{pmatrix} a & b & c \\ & \dots & \\ & \dots & \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} ah_1 + bh_2 + ch_3 \\ \dots \\ \dots \end{pmatrix}$$

e.g.



$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} h_1 + h_2 \\ h_1 + h_2 \\ h_3 \end{pmatrix}$$

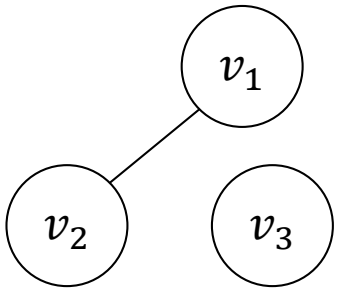
Adding neighbors w/ linear algebra

- Let matrix H be a matrix where each row is a node and each column is a feature
  - H have dim $|V| \times d$
- Let A be an adjacency matrix
  - Let $\hat{A} = A + I$ where I is the identify matrix
- Further **normalize** each row of $\hat{A}$ to sum to 1

$$\begin{pmatrix} 1/3 & 1/3 & 1/3 \\ \dots & & \\ \dots & & \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} (h_1 + h_2 + h_3)/3 \\ \dots \\ \dots \end{pmatrix}$$

Note that **normalize** does the same thing as **mean**

e.g.



$$\begin{pmatrix} .5 & .5 & 0 \\ .5 & .5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} (h_1+h_2)/2 \\ (h_1+h_2)/2 \\ h_3 \end{pmatrix}$$

Adding neighbors w/ linear algebra

- Let matrix H be a matrix where each row is a node and each column is a feature
  - H have dim $|V| \times d$
- Let A be an adjacency matrix
  - Let $\hat{A} = A + I$ where I is the identify matrix
- Further **normalize** each row of $\hat{A}$ to sum to 1
  - To perform this normalization, it suffices that we let $\hat{A} \leftarrow D^{-1}\hat{A}$ where D is the diagonal node degree matrix
    - In PyTorch, use `torch.nn.functional.normalize(A, p=1, dim=1)`
  - Or, use $\hat{A} \leftarrow D^{-\frac{1}{2}}\hat{A}D^{-\frac{1}{2}}$, the spectral variant
    - In PyTorch, use

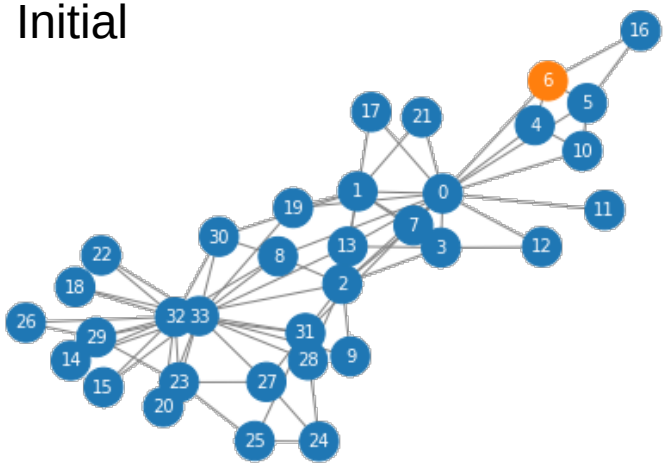
```
D = torch.diag(torch.sum(A, 1)).inverse().sqrt()
D = torch.mm(torch.mm(D, A), D)
```

Normalized $\hat{A}$ is in general **not symmetric**

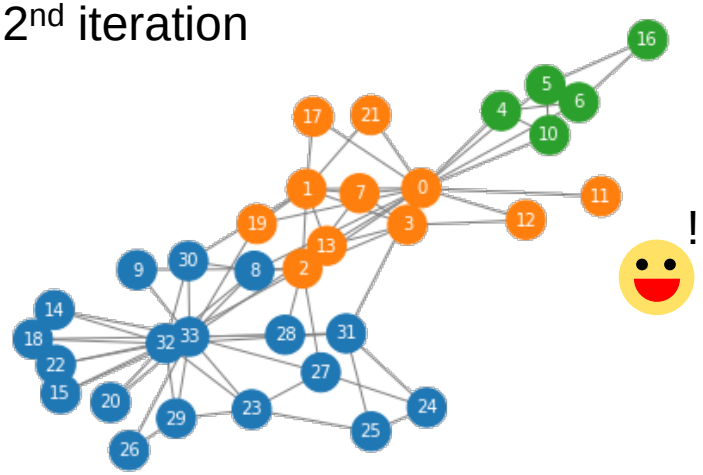
Adding neighbors w/ linear algebra

- Redo karate club with normalized $\hat{A} \leftarrow D^{-1}\hat{A}$

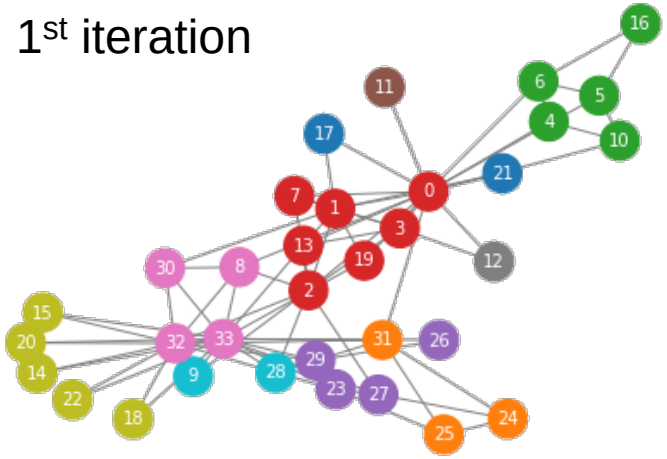
Initial



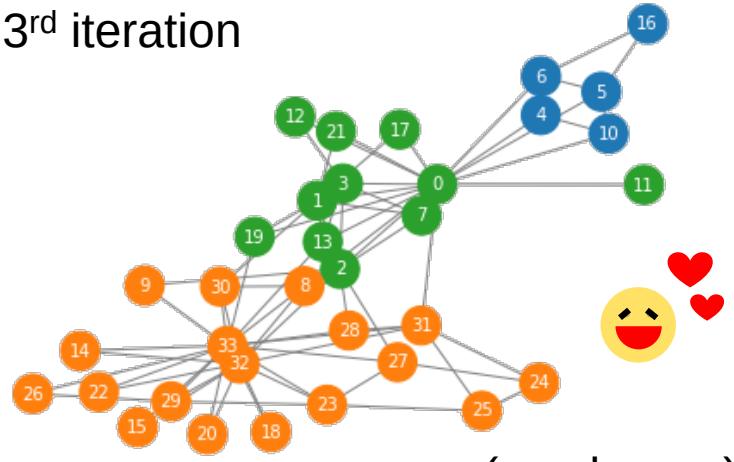
2nd iteration



1st iteration



3rd iteration



(no change)

(number of clusters determined with silhouette method)

Adding neighbors: evaluation

- Why do we need normalization
  - Without normalization, feature values for the nodes of high centrality would quickly add up, making them distinct from the nodes of low centrality
- How many iterations should be used?
  - Each iteration would “bunch up” neighboring features of 1 hop away (receptive field)
  - With that in mind, we should determine the number of iterations by the nature of the graph
  - The earliest (RNN-like) GNNs are iterated until convergence but these ideas were quickly replaced by (CNN-like) GNNs where the number of iterations is fixed as defined by the number of layers

Relationship to PageRank

For simplicity consider eigenvector centrality problem, that is, the undirected version of PageRank

- Problem Statement. Suppose each node v corresponds to a value x_v which is the sum of its neighbors' values

$$x_v = \frac{1}{\lambda} \sum_{u \in N(v)} x_u$$

where λ is some given constant and positive factor

Given adjacency matrix A of a graph G, solve x_v for all $v \in G$

- Problem is equivalent to that of finding vector $\boldsymbol{x}$ such that

$$\lambda \boldsymbol{x} = A\boldsymbol{x}$$

  - Solutions are all the eigenvectors that maximize $\boldsymbol{x}^\top A\boldsymbol{x}$

- Problem is also equivalent to that of **adding up all neighboring single-valued features**, but **excluding that of self**, **until convergence**

Adding neighbors: evaluation

- Benefits of strategy
  - Simplicity
  - Efficiently computed with adjacency matrix
- Disadvantage of strategy
  - Embeddings produced are of size of the number of nodes in the graph

    ⇒ Learn a **transformation matrix**

    $W: R^{|V|} \rightarrow R^{d}$ for some smaller d

Transformation matrix W

- W is typically a linear transformation layer of size $|V| \times d$ where d is the target dimensionality of the embeddings
- Combined with the adjacency matrix $\hat{A}$, we now have a complete matrix formulation for computing embedding h_v of a node v from (itself and) its neighbors, in the form of

$$h_v \leftarrow \left(\hat{A}\right)_v HW$$

where

  - $\left(\hat{A}\right)_v$ is the row in $\hat{A}$ for the node v, and
  - H is a matrix containing the features/embeddings of all the nodes (of course, only the rows in H with non-zero entries in $\left(\hat{A}\right)_v$ are needed for computing h_v)

- **Variations in this formula lead to various frameworks**

Variations

- Message-aggregation (MSG-AGG)
  - First transform features/embeddings (MSG), then aggregate transformed embeddings (AGG)

$$h_v \leftarrow \underbrace{\left(\hat{A}\right)_v}_{\text{aggregate}} \overbrace{(HW)}^{\text{message}}$$

- Separate computation of self and neighbors
  - Exclude entry for v from $\left(\hat{A}\right)_v$, and let

$$h_v \leftarrow \mathrm{AGG}\left(\overbrace{\left(\hat{A}\right)_v HW}^{\text{Aggregate only neighbors}}, \overbrace{h_v W'}^{\text{Self}}\right)$$

Learn a different transformation for self

Also denoted as B

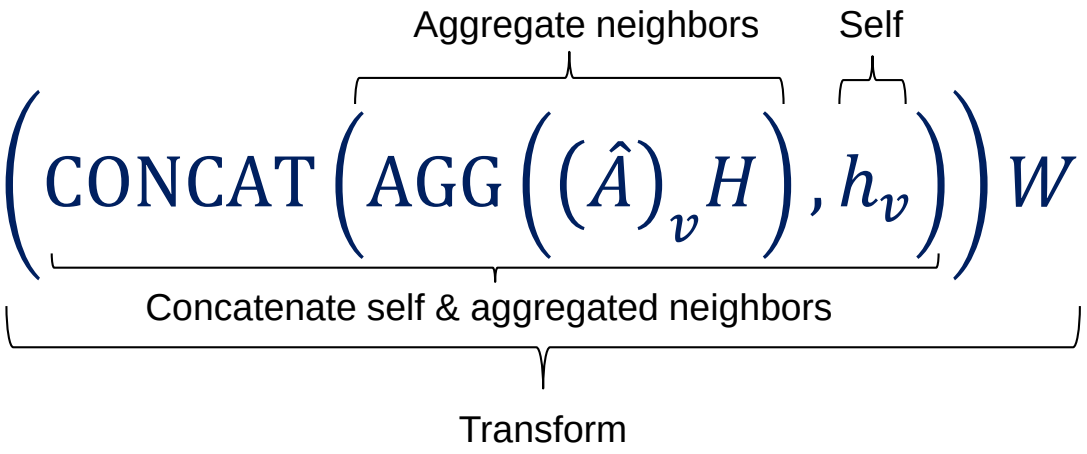
where AGG is, for instance, concatenation

Frameworks

- **Graph Convolutional Network** (GCN)

  $h_v \leftarrow \left(\hat{A}\right)_v (HW)$ (basically just MSG-AGG)

- **GraphSAGE**
  - Exclude entry for v from $\left(\hat{A}\right)_v$



AGG can be one of many options including MLP and LSTM (!)

$\Rightarrow$ AGG is learnable

GraphSAGE

- $h_v \leftarrow \left(\text{CONCAT}\left(\text{AGG}\left(\left(\hat{A}\right)_v H\right), h_v\right)\right) W$
  - As mentioned AGG is learnable
  - $\hat{A}$ is also learnable
    - Generalize adjacency to **attention weights** α_{uv}

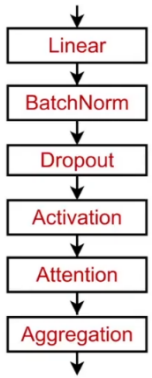
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{x \in N(v)} \exp(e_{vx})}$$

where e_{vu} is a measure of how related u and v are, usually computed as $\text{LINEAR}\left(\text{CONCAT}(h_v W, h_u W)\right)$

- PyTorch Geometric (PyG) has implementations of these frameworks (GCN, GraphSAGE)

In practical use

- At this point we have not mentioned activation function or other elements of DL
  - For activation function just let $\sigma(h_v) \leftarrow h_v$
  - Mix and match as you like



- Embeddings can be used for many downstream tasks
  - We have earlier used k-means for clustering the final output
  - Better performed by constructing a neural network directly with the GNN layers

Augmenting graphs

- Adding features
  - Similar to feature engineering
- Adding virtual nodes
  - Allow nodes in sparse regions of graph to communicate
- Adding virtual edges
  - Bipartite graph
    - Given adjacency matrix A of bipartite graph G
    - A^2 gives the number of paths of distance 2 between nodes in G, and forms an adjacency matrix between nodes of the same type
    - A^2 allows us to separate G into two graph each of the same type
    - $A + A^2$ forms an adjacency matrix with heterogeneous edges