Graph Neural Networks Demystified

An overview of the essential concepts in Stanford CS224W (Lectures 1~9) with only oversimplified examples

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

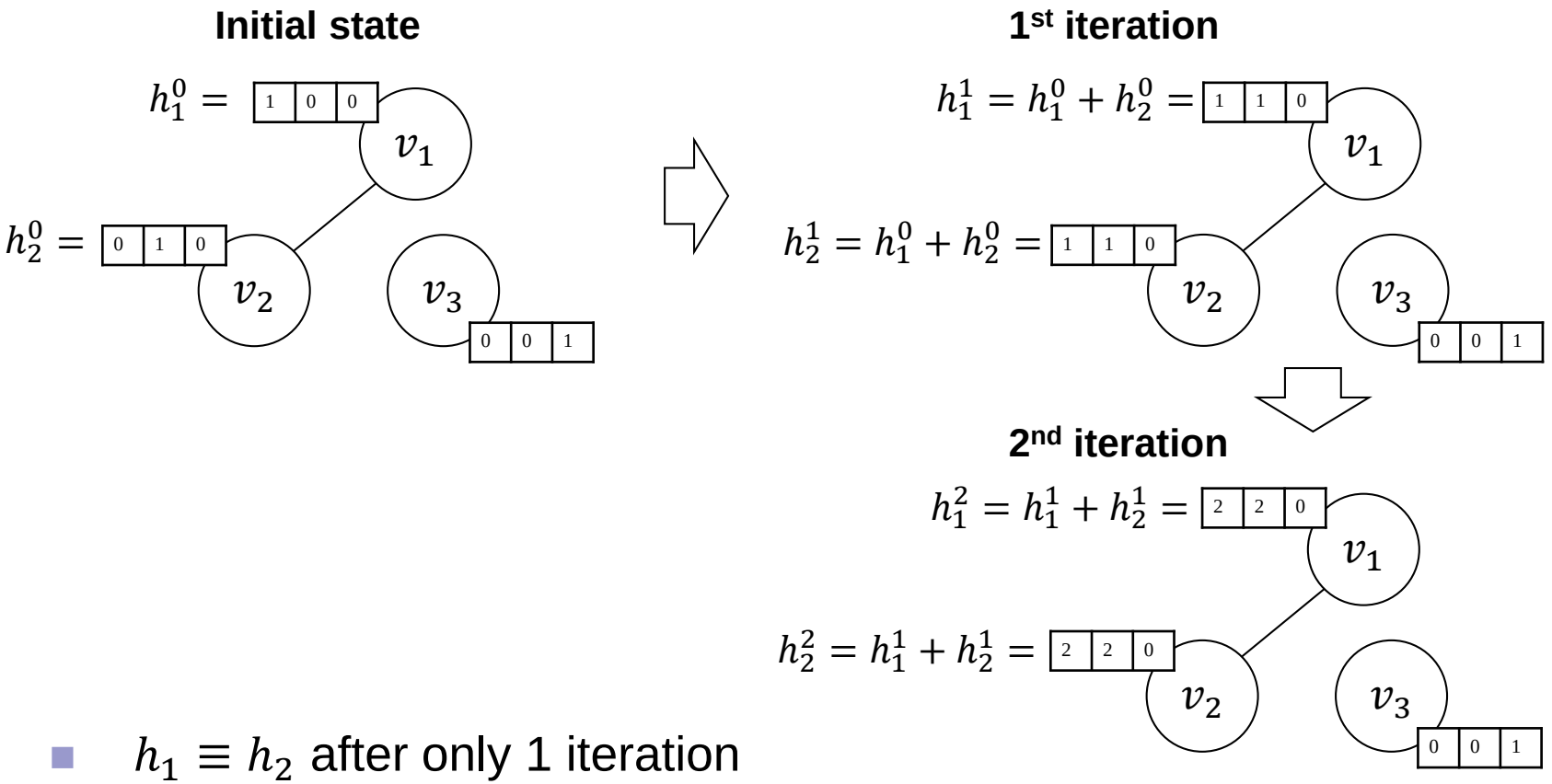
Task-independent embeddings

- 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

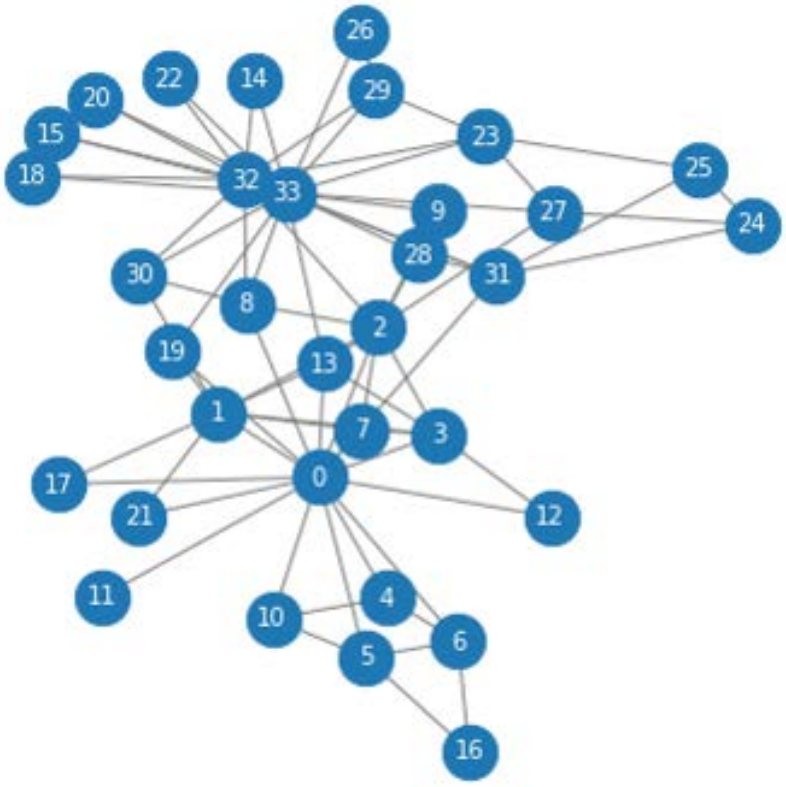
Example: Let h_i^j denote features of node i at iteration j and let $h_1^0 = (1 \quad 0 \quad 0)$, $h_2^0 = (0 \quad 1 \quad 0)$, and $h_3^0 = (0 \quad 0 \quad 1)$



- $h_1 \equiv h_2$ 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?



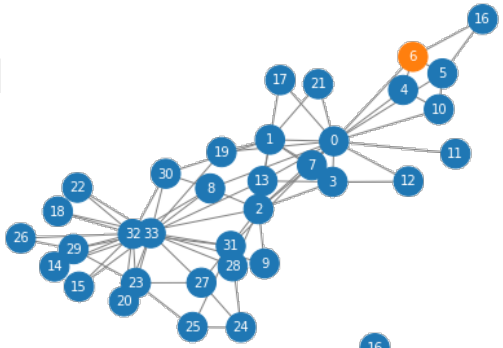
Let's try with karate club network

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

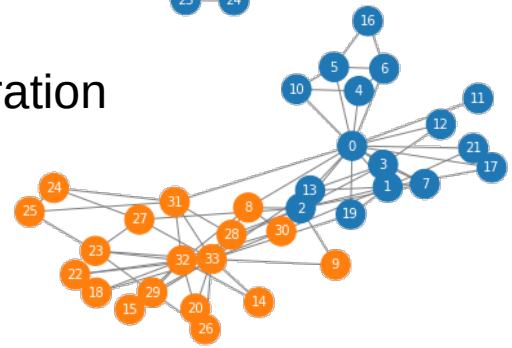
Embeddings by adding neighbors

- Karate club

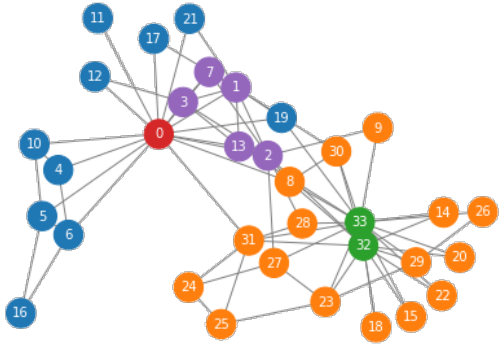
Initial



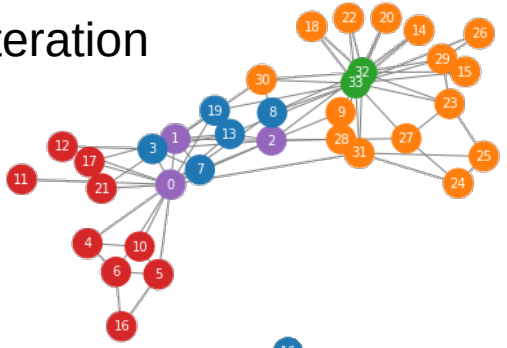
1st iteration



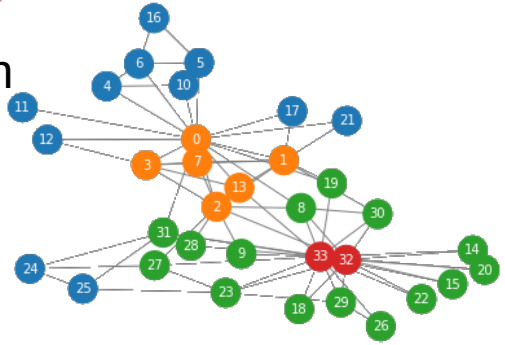
2nd iteration



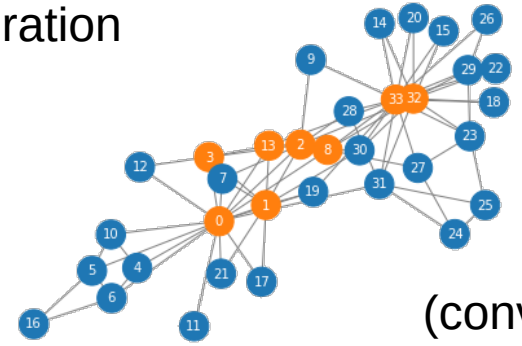
3rd iteration



4th iteration



5th iteration



(converged)

(number of clusters determined with silhouette method)

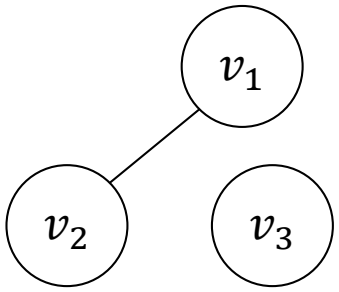
Adding neighbors w/ linear algebra

- 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 $\hat{A}H$

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

Permutation invariant so that the outcome is the same regardless of node order within matrix

e.g.



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

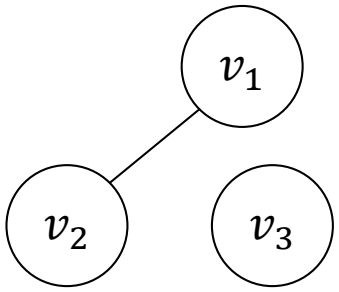
Adding neighbors w/ linear algebra

- 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)
      ```

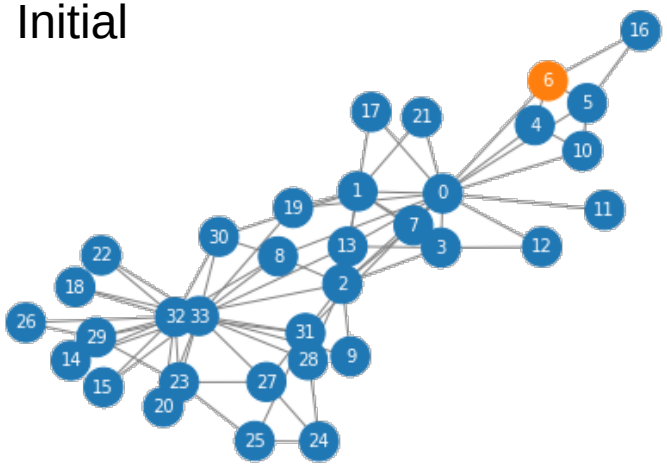
Normalized $\hat{A}$ is in general **not symmetric**

The spectral variant is symmetric, but **not normalized**

Adding neighbors w/ linear algebra

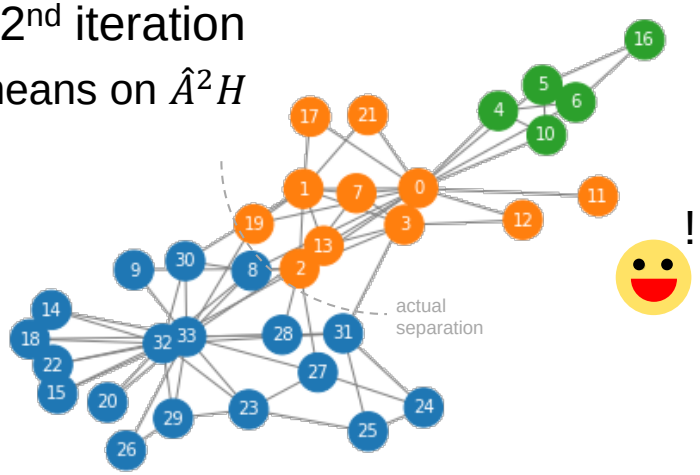
- Redo karate club with normalized $\hat{A} \leftarrow D^{-1}\hat{A}$

Initial



2nd iteration

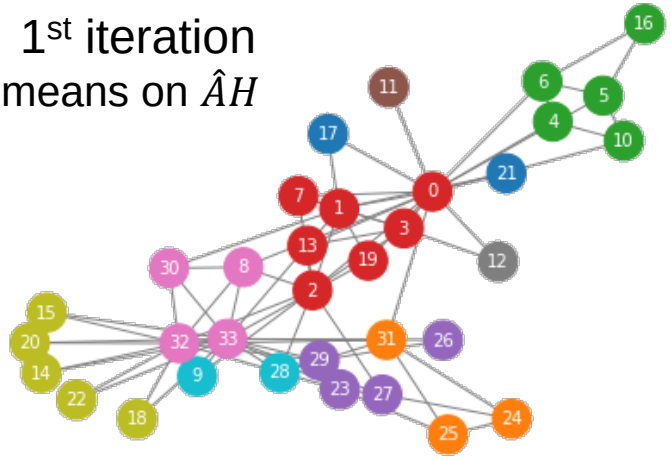
k-means on $\hat{A}^2 H$



!

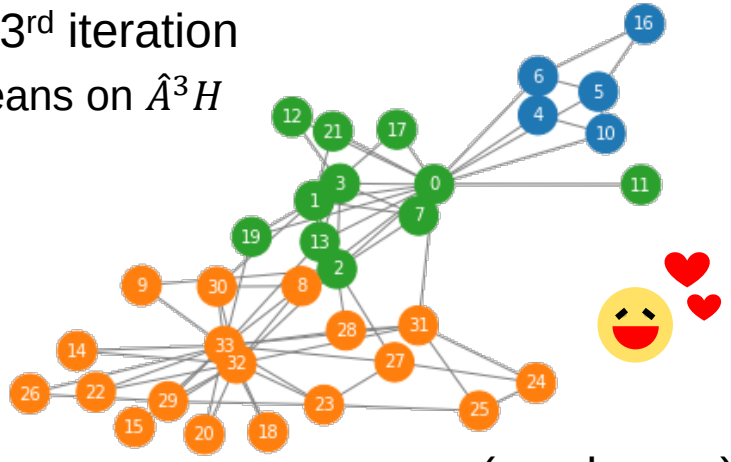
1st iteration

k-means on $\hat{A}H$



3rd iteration

k-means on $\hat{A}^3 H$



(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?

  Early (RNN-like) GNNs are iterated until convergence but they quickly ran out of favor to Graph Convolutional Networks (GCNs) where the number of iterations is fixed as defined by the number of convolutional layers
  - Each iteration would “bunch up” neighboring features of 1 hop away (receptive field)
  - We should determine the number of iterations by the **nature of the graph** (later)

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

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

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

Nature of the graph

- The **Cheeger constant** (or expansion constant) of an **unweighted** graph $G(E,V)$ is

$$h(G) = \min_{S \subseteq V} \frac{|\{(u,v)|u \in S, v \in \bar{S}\}|}{\min(|S|,|\bar{S}|)}$$

  - $|\{(u,v)|u \in S, v \in \bar{S}\}|$ indicates how well vertices in S are connected to vertices in $\bar{S}$
  - $\min(|S|,|\bar{S}|)$ favors S where $|S| \approx |\bar{S}|$
  - For weighted graphs, a similar measure called conductance can be defined with edge weights (a_{uv})

$$\phi(G) = \min_{S \subseteq V} \frac{\sum_{v \in S, u \in \bar{S}} a_{vu}}{\min\left(\sum_{v \in S, u \in V} a_{vu}, \sum_{v \in \bar{S}, u \in V} a_{vu}\right)}$$

Based on https://dspace.mit.edu/handle/1721.1/121660

Nature of the graph

- The **Cheeger constant** (or expansion constant) of an **unweighted** graph $G(E,V)$ is

$$h(G) = \min_{S \subseteq V} \frac{|\{(u,v)|u \in S, v \in \bar{S}\}|}{\min(|S|,|\bar{S}|)}$$

- A large $h(G)$ indicates a **highly-connected graph**
  - A feature in a highly-connected graph will propagate in the graph very quickly
  - A random walk in a highly-connected graph converges in $O(\log|V|)$ steps to an almost uniform distribution (mixing time)
    - Upon which the embedding of every node is influenced almost equally by any other node

Nature of the graph

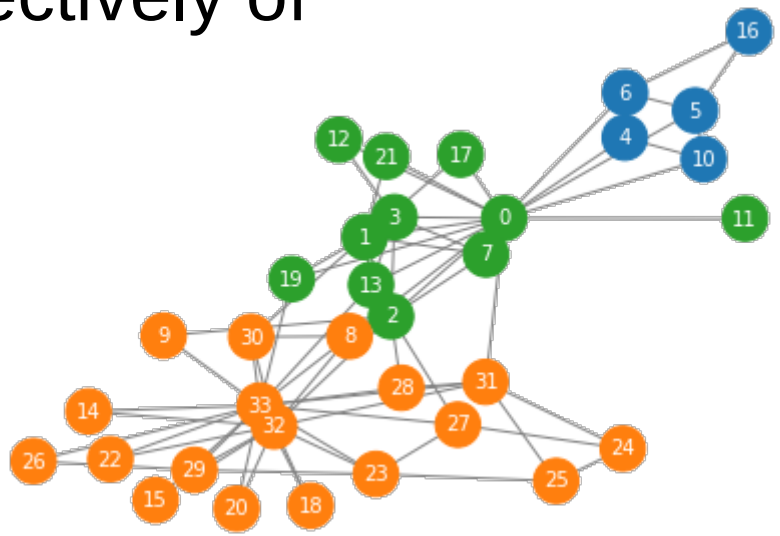
- Examine the number of steps required for the clusters in the karate club to mix

- The clusters are respectively of sizes 18, 11, 5

$$\log(18) = 4.17$$

$$\log(11) = 3.46$$

$$\log(5) = 2.32$$

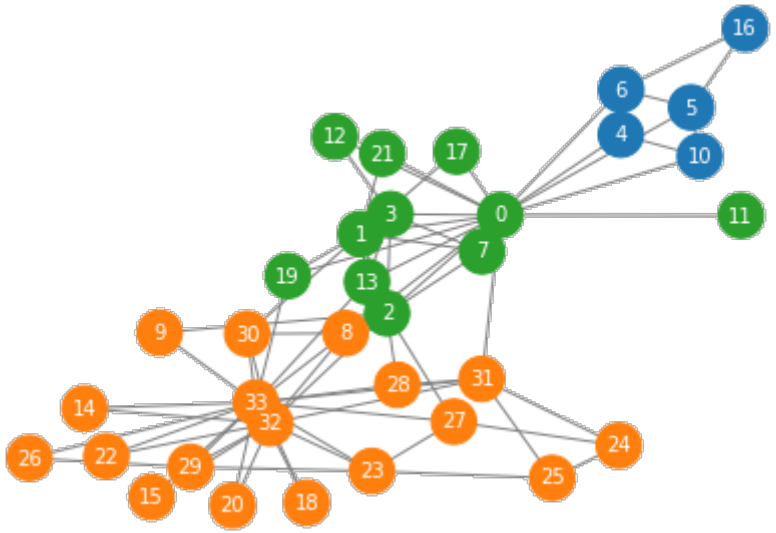


- 5 iterations/ steps suffice for nodes in the respective clusters to influence each other equally

Nature of the graph

- Examine the number of steps required for the clusters in the karate club to mix

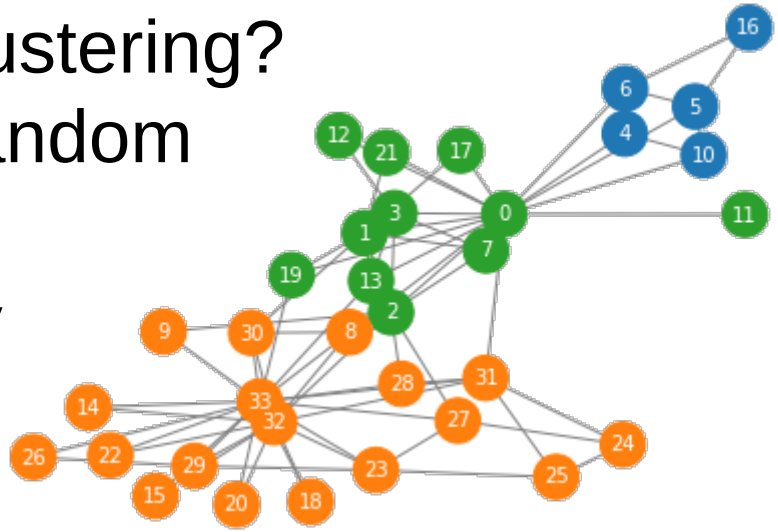
- Will increasing the number of iterations eventually spread the features uniformly across the entire karate club graph?
  - Depends on whether the limiting distribution x^* (i.e. $\hat{A}x^* = x^*$) is everywhere constant



Nature of the graph

- Examine the number of steps required for the clusters in the karate club to mix

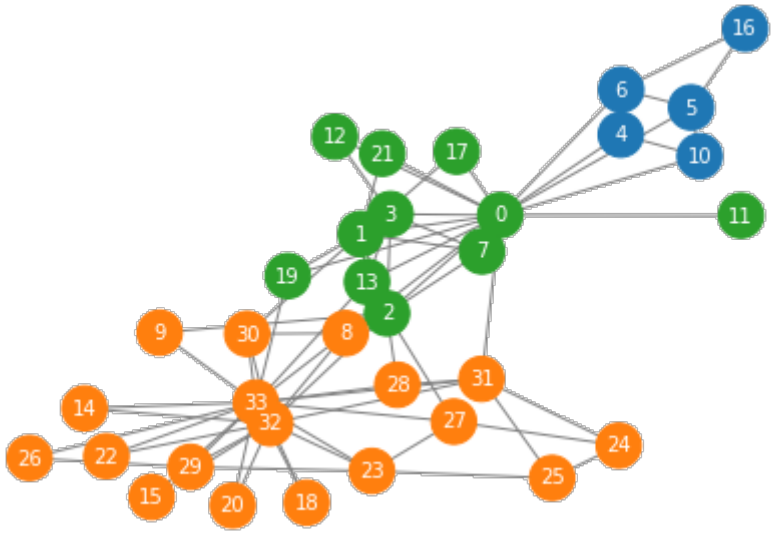
- Is this clustering the same as (normalized Laplacian) spectral clustering? Since they are both random walks?
  - They would be exactly the same if
    - This is repeated to convergence, and
    - We managed to use multi-valued features in the spectral clustering (and also use the eigenvector of $\lambda = 0$)



Nature of the graph

- Examine the number of steps required for the clusters in the karate club to mix

- Will the clustering results differ between $D^{-1}L$ and $D^{-1/2}LD^{-1/2}$?
  - This question is deep



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 $\left(\hat{A}\right)_v$

$$h_v \leftarrow \Big(\underbrace{\text{CONCAT}\Big(\overbrace{\text{AGG}\Big(\left(\hat{A}\right)_v H \Big)}^{\text{Aggregate neighbors}}, \overbrace{h_v}^{\text{Self}} \Big)}_{\text{Concatenate self \& aggregated neighbors}} \Big) W$$

Transform

AGG can be one of many options including MLP, LSTM, *etc.*

(Why use these? See Graph Isomorphism Network)

⇒ AGG is learnable

Frameworks

- **Graph Attention Networks** (GAN)
  - Instead of learning AGG, learn $\hat{A}$
    - Generalize the adjacency matrix $\hat{A}$ to **attention weights** $\Lambda = (\alpha_{vu})$

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

where $\alpha_{vu} = \dfrac{\exp(e_{vu})}{\sum_{x \in N(v)} \exp(e_{vx})}$, and

e_{vu} is a measure of how related u and v are

- e_{vu} is usually computed as $\mathrm{LINEAR}\left(\mathrm{CONCAT}(h_v W, h_u W)\right)$

  - Do not confuse with Generative Adversarial Networks which is for generating anime pics

- Implemented in PyTorch Geometric (PyG) as GCNConv (GCN), SAGEConv (GraphSAGE), and GATConv (GAN)
  - See https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html

Frameworks

- **Message Passing Neural Network** (MPNN)
  - Involve $N(v)$ in the transformation W for v

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

    $$\Rightarrow h_v \leftarrow H \bigoplus_{u \in N(v)} \phi(h_v, h_u)$$

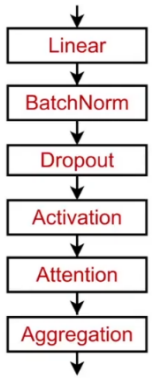
    This change allows us to incorporate edge features in the embedding

    $$\Rightarrow h_v \leftarrow H \bigoplus_{u \in N(v)} \phi(h_v, h_u, e_{vu})$$

  - How to compute $\phi(h_v, h_u, e_{vu})$ algebraically?
    - Let edge features be in a 3D matrix E
    - Then, $\left(\hat{A}\right)_v H$ and $\left(\hat{A}\right)_v (E)_v$ gives us two matrices with matching rows (each row corresponding to h_u and e_{vu} respectively)
    - Concatenate $\left(\hat{A}\right)_v H$ and $\left(\hat{A}\right)_v (E)_v$ and give as input to an NN
- A similar framework, Principal Neighborhood Aggregation (PNAConv), is implemented in PyG (these frameworks are not discussed in CS224W)

In practical use

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

In practical use

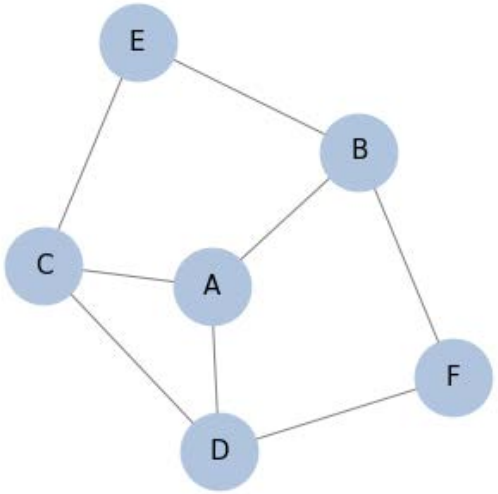
- Adding graph elements
  - Features
    - Similar to feature engineering
  - Virtual nodes
    - Connecting all the nodes in a sparse but apparent subgraph to a virtual node will allow those nodes to better communicate
  - Virtual edges
    - Create new graph by systematically adding edges
    - Example: Given a bipartite graph, breaking the graph into two of only nodes of the same type is good for some analyses
      - Let A be the adjacency matrix of the bipartite graph G
      - A^2 then gives the number of paths of distance 2 between nodes in G
        ⇒ an adjacency matrix between nodes of the same type
        ⇒ allows us to separate G into two graphs, each of same node type
      - $A + A^2$ can form an adjacency matrix with heterogeneous edges

Training GNNs

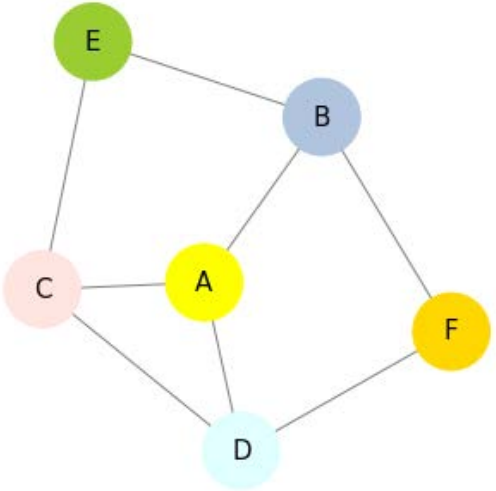
- Using node embeddings as input to a prediction function
  - Embedding of 1 node can be used directly
  - Embeddings of 2 nodes can be
    - Concatenated to form an **edge embedding**
    - Projected on each other to get their similarity
  - Embeddings of nodes of the entire graph can be
    - Summed, averaged, searched for max/min, *etc.*
    - Clustered, then the clusters summed, average, *etc.*, in a hierarchical fashion
- **Edge embeddings from edge features** are also possible, though not discussed in CS224W
  - The framework **Node and Edge features in graph Neural Networks** (NENN) (not yet in PyG)

Distinguishing node embeddings

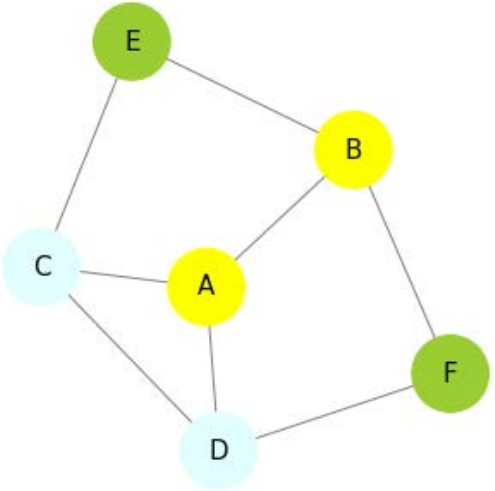
- Should C and D have the same embedding in the following graphs? Given that features are given by the colors and mutually exclusive (orthogonal)



Yes?



No?

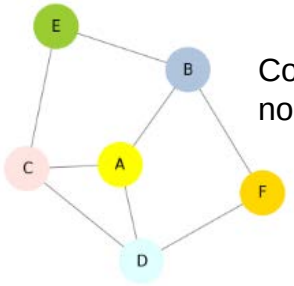


Should?

- How about A and B?
- **Idea**: Two nodes should have the same **embedding** if they have the same **feature** and **neighborhood structure**, and vice versa

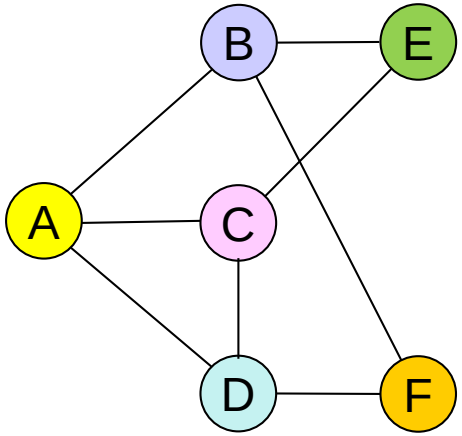
Distinguishing node embeddings

- Given a GCN of 2 layers, the embedding of A is computed as follows

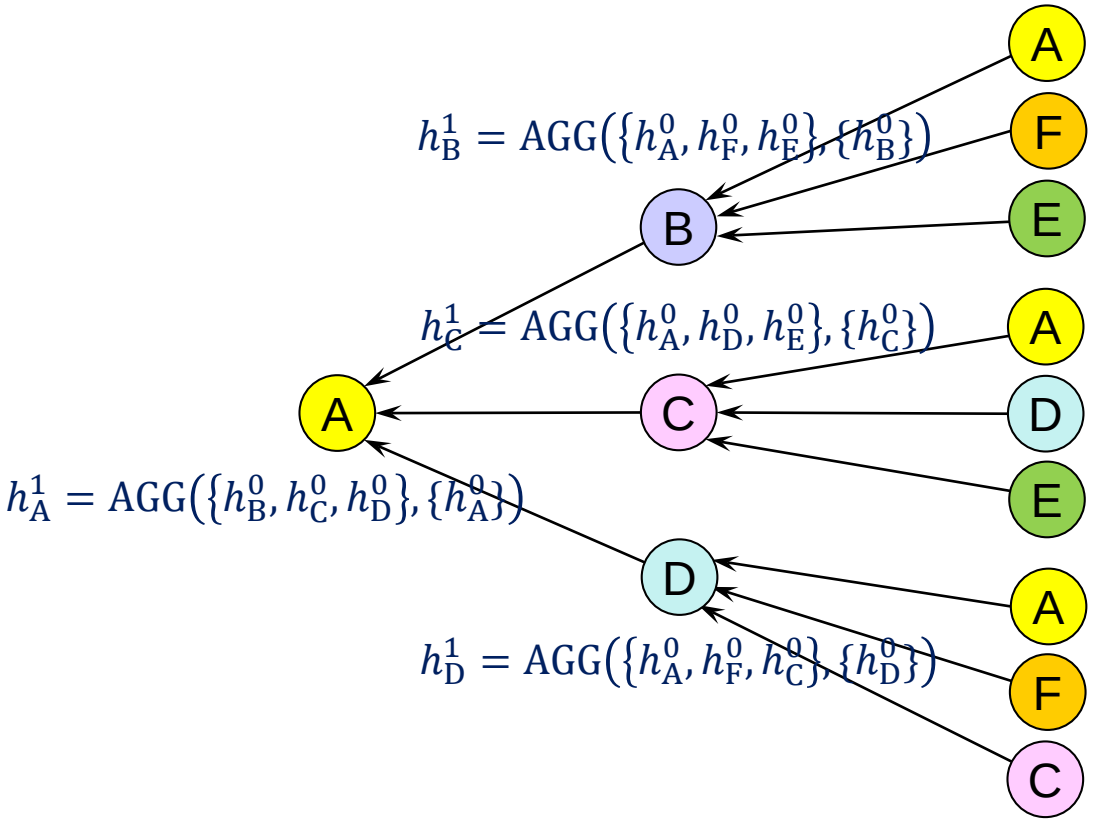


Consider the case where nodes have distinct features

Rearranged w.r.t. distance from A



Computation graph of A's embedding



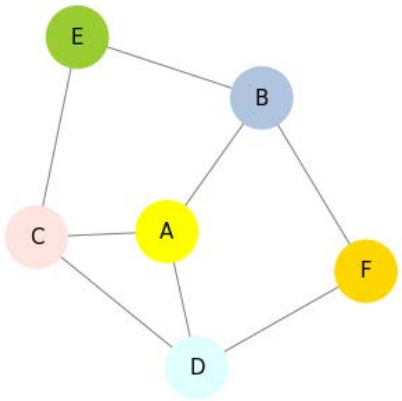
Distinguishing node embeddings

- Given a GCN of 2 layers, the embedding of A is computed as follows
  - Let h_A^0=|1|0|0|0|0|0|, h_B^0=|0|1|0|0|0|0|, h_C^0=|0|0|1|0|0|0|, h_D^0=|0|0|0|1|0|0|, h_D^0=|0|0|0|0|1|0|, h_D^0=|0|0|0|0|0|1|, and let AGG be **addition**. Then
    - $h_B^1 = \text{AGG}(\{h_A^0, h_F^0, h_E^0\}, \{h_B^0\}) = 1|1|0|0|1|1$
    - $h_C^1 = \text{AGG}(\{h_A^0, h_D^0, h_E^0\}, \{h_C^0\}) = 1|0|1|1|1|0$
    - $h_D^1 = \text{AGG}(\{h_A^0, h_F^0, h_C^0\}, \{h_D^0\}) = 1|0|1|1|0|1$
    - $h_A^1 = \text{AGG}(\{h_B^0, h_C^0, h_D^0\}, \{h_A^0\}) = 1|1|1|1|0|0$

    Compute $AGG(X)$ as $\hat{A}H$, where $\hat{A}$ is the adjacency matrix (with self loop), and H is a matrix containing all the embeddings in X
  - Finally the embedding of A is
    - $h_A^2 = \text{AGG}(\{h_B^1, h_C^1, h_D^1\}, \{h_A^1\}) = 4|2|3|3|2|2$
  - Similarly, $h_B^2 = 2|4|2|2|2|2$
    $h_C^2 = 3|2|4|3|2|1$
    $h_D^2 = 3|2|3|4|1|2$
    $h_E^2 = 2|2|2|1|3|1$
    $h_F^2 = 2|2|1|2|1|3$

Distinguishing node embeddings

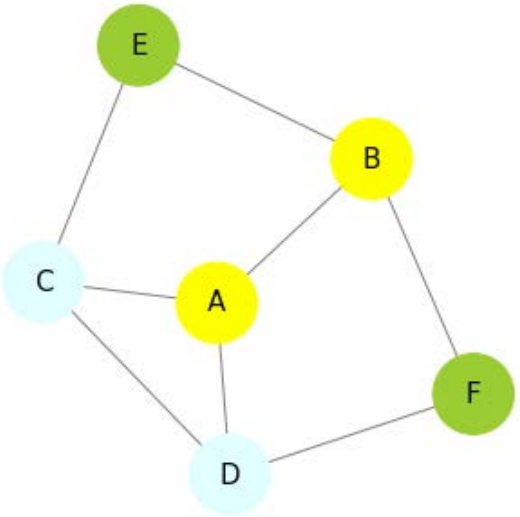
- Given a GCN of 2 layers, the embedding of A is computed as follows
  - Let h_A^0=[1|0|0|0|0|0], h_B^0=[0|1|0|0|0|0], h_C^0=[0|0|1|0|0|0], h_D^0=[0|0|0|1|0|0], h_D^0=[0|0|0|0|1|0], h_D^0=[0|0|0|0|0|1], and let AGG be **addition**. Then
  - $h_A^2 = 4|2|3|3|2|2$
    $h_B^2 = 2|4|2|2|2|2$
    $h_C^2 = 3|2|4|3|2|1$
    $h_D^2 = 3|2|3|4|1|2$
    $h_E^2 = 2|2|2|1|3|1$
    $h_F^2 = 2|2|1|2|1|3$

    By induction they will be distinct for all subsequent iterations
  - For a graph with **distinct node features**, the embeddings will be distinct under **addition** regardless of **neighborhood structure** or **iterations**
    - With the exception of “twin nodes” that are connected only to each other (in which case they will become equal after the first iteration)

Distinguishing node embeddings

- Given a GCN of 2 layers, the embedding of A is computed as follows
  - Let $h_{\mathrm{A}}^0 = h_{\mathrm{B}}^0 = \boxed{1}\boxed{0}\boxed{0}$, $h_{\mathrm{C}}^0 = h_{\mathrm{D}}^0 = \boxed{0}\boxed{1}\boxed{0}$, $h_{\mathrm{E}}^0 = h_{\mathrm{F}}^0 = \boxed{0}\boxed{0}\boxed{1}$, and let AGG be **addition**. Then, for the following graph



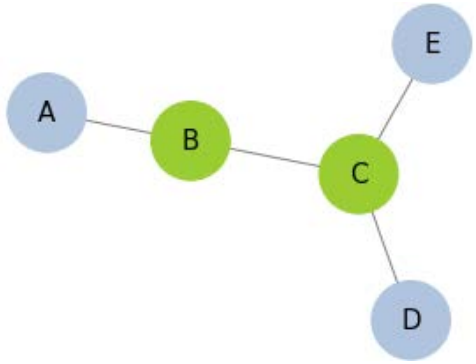
$$h_{\mathrm{A}}^1 = 2|2|0 \qquad h_{\mathrm{A}}^2 = 6|6|4$$
$$h_{\mathrm{B}}^1 = 2|0|2 \qquad h_{\mathrm{B}}^2 = 6|4|4 \qquad h_{\mathrm{A}}^2 \neq h_{\mathrm{B}}^2$$
$$h_{\mathrm{C}}^1 = 1|2|1 \qquad h_{\mathrm{C}}^2 = 5|7|3$$
$$h_{\mathrm{D}}^1 = 1|2|1 \qquad h_{\mathrm{D}}^2 = 5|7|3 \qquad h_{\mathrm{C}}^2 = h_{\mathrm{D}}^2$$
$$h_{\mathrm{E}}^1 = 1|1|1 \qquad h_{\mathrm{E}}^2 = 4|3|4$$
$$h_{\mathrm{F}}^1 = 1|1|1 \qquad h_{\mathrm{F}}^2 = 4|3|4 \qquad h_{\mathrm{E}}^2 = h_{\mathrm{F}}^2$$

  - Two nodes with the **same feature** will always have the same embedding under **addition** if and only if they have the same **neighborhood structure**
    - What about other AGG functions, e.g. **mean**?

Distinguishing node embeddings

- Let $h_A^0 = h_D^0 = h_E^0 = \boxed{1}\boxed{0}$, $h_B^0 = h_C^0 = \boxed{0}\boxed{1}$, and let AGG be **mean**. Then, for the following graph

Compute $AGG(X)$ as $\hat{A}H$ with normalized $\hat{A}$



$$h_A^1 = 0.5 \mid 0.5$$
$$h_B^1 = 0.33 \mid 0.67$$
$$h_C^1 = 0.5 \mid 0.5$$
$$h_D^1 = 0.5 \mid 0.5$$
$$h_E^1 = 0.5 \mid 0.5$$

- As expected, $h_A^1 = h_D^1 = h_E^1$ due to the same feature and neighborhood structure (within 1 hop)
- However, $h_A^1 = h_C^1$ in spite of their differences in both features and neighborhood structure

  ⇒ **mean** cannot get distinct embeddings for distinct nodes

  - Even though this is true only for the first iteration in this example, similar examples can be obtained for any number of layers

Distinguishing node embeddings

- While our earlier examples did not consider the transformation W or the activation function σ, the arguments are just as valid with them considered
- A function that can distinguish the nodes of distinct feature and neighborhood structure is one that is **injective**
  - **mean** and **max** are not injective
  - On the other hand, **sum** has problems as mentioned
- Theorem (Xu *et al.* 2019). Any injective AGG function can be expressed as $\Phi(\sum_{x \in S} f(x))$ for some non-linear Φ and linear f
- Since MLP is able to approximate any function, we can learn Φ and f with non-linear MLP_Φ and linear MLP_f

$$\mathrm{AGG} = \mathrm{MLP}_\Phi\left(\sum_{x \in S} \mathrm{MLP}_f(x)\right)$$

⇒ **Graph Isomorphism Network (GIN)**

Graph Fourier transform

- GCN has an alternative theoretical basis in terms of graph Fourier transform
- Let U be a eigenbasis of some Laplacian L
- Then $U^\top x$ is a projection of distribution x on eigenbasis U
  - $$U^\top x = \begin{bmatrix} \leftarrow & \mu_1 & \rightarrow \\ \leftarrow & \mu_2 & \rightarrow \\ & \vdots & \end{bmatrix} x = \begin{bmatrix} \mu_1^\top x \\ \mu_2^\top x \\ \vdots \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \end{bmatrix} = \dot{x},$$

    where $a_i = \mu_i x$ is the projection onto μ_i
  - The projected space is $\sum_i a_i \mu_i$

Not in CS224W

Graph Fourier transform

- GCN has an alternative theoretical basis in terms of graph Fourier transform
- Let U be a eigenbasis of some Laplacian L
- An application of U would transform $\dot{x}$ back into x

$$U\dot{x} = \begin{bmatrix} \uparrow & \uparrow & \\ \mu_1 & \mu_2 & \dots \\ \downarrow & \downarrow & \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} \mu_{11}a_1 + \mu_{21}a_2 + \cdots \\ \mu_{12}a_1 + \mu_{22}a_2 + \cdots \\ \vdots \end{bmatrix}$$

$$= \mu_1 a_1 + \mu_2 a_2 + \cdots = \mu_1 \mu_1^\top x + \mu_2 \mu_2^\top x + \cdots$$

$$= \left(\sum_i \mu_i \mu_i^\top\right) x = Ix = x$$

Homework: prove $\sum_i \mu_i \mu_i^\top = I$

Graph Fourier transform

- GCN has an alternative theoretical basis in terms of graph Fourier transform
- Let U be a eigenbasis of some Laplacian L
- Then $U^\top x$ is a projection of distribution x on eigenbasis U, $U^\top x = \dot{x}$
- An application of U would transform $\dot{x}$ back into x, $U(\dot{x}) = U(U^\top x) = x$

  (Also note that $UU^\top = I$, so this is a no brainer)
- Denote $U^\top x$ as $F(x)$ and $U\dot{x}$ as $F^{-1}(\dot{x})$

Graph Fourier transform

- A convolution of x in the Fourier domain of a graph G is
  $x * g = F^{-1}\big(F(x) \odot F(g)\big) = U(U^\top x \odot U^\top g)$
  where g is the filter corresponding to G,
  and $\odot$ is the element-wise (Hadamard) product
- Suppose $U^\top g = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \end{bmatrix}$

  Let $g_\theta = \mathrm{diag}(U^\top g) = \begin{bmatrix} g_1 & 0 & 0 \\ 0 & g_2 & 0 \\ 0 & 0 & \ddots \end{bmatrix}$

  Then we can write $x * g_\theta = U g_\theta U^\top x$

  $\left(U^\top x \odot U^\top g = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \end{bmatrix} \odot \begin{bmatrix} g_1 \\ g_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} a_1 g_1 \\ a_2 g_2 \\ \vdots \end{bmatrix},\quad g_\theta U^\top x = \begin{bmatrix} g_1 & 0 & 0 \\ 0 & g_2 & 0 \\ 0 & 0 & \ddots \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} a_1 g_1 \\ a_2 g_2 \\ \vdots \end{bmatrix}\right)$

  - g_θ can be seen as a function of the eigenvalues
    $g(L) = g(U \Lambda U^\top) = U g_\theta(\Lambda) U^\top$

Graph Fourier transform

- $x * g_\theta = U g_\theta U^\top x$
  - View $g_\theta U^\top$ as a scaling g_1, g_2,… that operates on each element in $x = [x_1 \quad x_2 \quad \ldots]$ after it is "convolved" with its neighbor elements according to the eigenbasis U
    - That is, if eigenvector $\mu_1 = [1 \quad 1 \quad 0]$, then x_1 is "convolved" with x_2 before being scaled by g_1
  - Each g_i scales what is in μ_i

Chebyshev approximation for U

- However, computing U is $O(N^3)$ and computing $U^\top x$ is $O(N^2)$ ⇒ expensive
- Approximate $x * g_\theta$ with Chebyshev polynomials

$$g_{\theta'} * x \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L}) x$$

where $\theta' \in \mathbb{R}^K$ are Chebyshev coefficients and $\tilde{L} = \frac{2}{\lambda_{\max}} L - I_N$ ($\lambda_{\max}$ is the largest eigenvalue)

- For $K = 1$, $g_{\theta'} * x \approx \theta'_0 x + \theta'_1 (L - I_N) x$
  - Furthermore, instead of using the coefficients
    - Let $\lambda_{\max} = 2$
    - Let θ'_0 and θ'_1 be parameters to be learned

Chebyshev approximation for U

- $g_{\theta'} * x \approx \theta'_0 x + \theta'_1 (L - I_N) x$
- Apply this approximation multiple times
  - Each time convolve information from the k nodes away neighbors
  - Each layer learns its own θ'_0 and θ'_1
- This approximation applies to all Laplacians
  - For the normalized Laplacian

$$g_{\theta'} * x = \theta'_0 x - \theta'_1 D^{-\frac{1}{2}} A D^{-\frac{1}{2}} x$$

  - If we constraint the number of parameters by letting $\theta'_0 = \theta'_1 = \theta$

$$g_{\theta'} * x = \theta \left(I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \right) x$$