Stanford CS224W: A General Perspective on Graph Neural Networks

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
Recap: Deep Graph Encoders

Output: Node embeddings. Also, we can embed subgraphs, graphs.
Idea: Node’s neighborhood defines a computation graph

Determine node computation graph

Propagate and transform information

Learn how to propagate information across the graph to compute node features
**Intuition:** Nodes aggregate information from their neighbors using neural networks.

![Diagram](image-url)
**Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!
Stanford CS224W: A General Perspective on Graph Neural Networks

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
A General GNN Framework (1)

GNN Layer = Message + Aggregation

- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020
Connect GNN layers into a GNN

- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity

GNN Layer 1

GNN Layer 2
A General GNN Framework (3)

Idea: Raw input graph ≠ computational graph
- Graph feature augmentation
- Graph structure augmentation

(4) Graph augmentation
How do we train a GNN

- Supervised/Unsupervised objectives
- Node/Edge/Graph level objectives
A General GNN Framework (5)

(5) Learning objective

(2) Aggregation

(1) Message

(3) Layer connectivity

GNN Layer 1

GNN Layer 2

(4) Graph augmentation

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020
Stanford CS224W: A Single Layer of a GNN
A GNN Layer

GNN Layer = Message + Aggregation

- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...

\[ \text{GNN Layer} = \text{Message} + \text{Aggregation} \]

(1) Message

(2) Aggregation

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020
Idea of a GNN Layer:

- Compress a set of vectors into a single vector

Two step process:

- (1) Message
- (2) Aggregation

Input node embedding $h_v^{(l-1)}$, $h_u^{(l-1)}_{u \in N(v)}$

Output node embedding $h_v^{(l)}$
Message Computation

- **(1) Message computation**
  - **Message function:** \( m_u^{(l)} = \text{MSG}^{(l)} \left( h_u^{(l-1)} \right) \)
    - **Intuition:** Each node will create a message, which will be sent to other nodes later
    - **Example:** A Linear layer \( m_u^{(l)} = W^{(l)} h_u^{(l-1)} \)
      - Multiply node features with weight matrix \( W^{(l)} \)

- **(2) Aggregation**
(2) Aggregation

Intuition: Each node will aggregate the messages from node \( v \)'s neighbors

\[
h_v^{(l)} = \text{AGG}^{(l)} \left( \left\{ m_u^{(l)}, u \in N(v) \right\} \right)
\]

Example: Sum(\( \cdot \)), Mean(\( \cdot \)) or Max(\( \cdot \)) aggregator

\[
h_v^{(l)} = \text{Sum}(\{m_u^{(l)}, u \in N(v)\})
\]
**Message Aggregation: Issue**

- **Issue**: Information from node $v$ itself **could get lost**
  - Computation of $h^{(l)}_v$ does not directly depend on $h^{(l-1)}_v$
- **Solution**: Include $h^{(l-1)}_v$ when computing $h^{(l)}_v$
  - (1) **Message**: compute message from node $v$ itself
    - Usually, a **different message computation** will be performed
      \[
      m^{(l)}_u = W^{(l)} h^{(l-1)}_u \quad \text{and} \quad m^{(l)}_v = B^{(l)} h^{(l-1)}_v
      \]
  - (2) **Aggregation**: After aggregating from neighbors, we can aggregate the message from node $v$ itself
    - Via **concatenation** or **summation**

[Then aggregate from node itself]

\[
\begin{align*}
 h^{(l)}_v &= \text{CONCAT} \left( \text{AGG} \left( \{ m^{(l)}_u, u \in N(v) \} \right), m^{(l)}_v \right) \\
\text{First aggregate from neighbors}
\end{align*}
\]
Putting things together:

(1) **Message**: each node computes a message

\[ m_u^{(l)} = MSG^{(l)} \left( h_u^{(l-1)} \right), \ u \in \{ N(v) \cup v \} \]

(2) **Aggregation**: aggregate messages from neighbors

\[ h_v^{(l)} = AGG^{(l)} \left( \left\{ m_u^{(l)}, \ u \in N(v) \right\}, m_v^{(l)} \right) \]

**Nonlinearity (activation)**: Adds expressiveness

- Often written as \( \sigma(\cdot) \): ReLU(\cdot), Sigmoid(\cdot), ...
- Can be added to message or aggregation
(1) **Graph Convolutional Networks (GCN)**

\[
h^{(l)}_v = \sigma \left( W^{(l)} \sum_{u \in N(v)} \frac{h^{(l-1)}_u}{|N(v)|} \right)
\]

- **How to write this as Message + Aggregation?**

\[
h^{(l)}_v = \sigma \left( \sum_{u \in N(v)} W^{(l)} \frac{h^{(l-1)}_u}{|N(v)|} \right)
\]
Classical GNN Layers: GCN (2)

- **(1) Graph Convolutional Networks (GCN)**

\[
\mathbf{h}^{(l)}_v = \sigma \left( \sum_{u \in N(v)} W^{(l)} \frac{\mathbf{h}^{(l-1)}_u}{|N(v)|} \right)
\]

- **Message:**
  - Each Neighbor: \( \mathbf{m}^{(l)}_u = \frac{1}{|N(v)|} W^{(l)} \mathbf{h}^{(l-1)}_u \)

- **Aggregation:**
  - **Sum** over messages from neighbors, then apply activation
  - \( \mathbf{h}^{(l)}_v = \sigma \left( \text{Sum} \left( \left\{ \mathbf{m}^{(l)}_u, u \in N(v) \right\} \right) \right) \)

**Normalized by node degree**
(In the GCN paper they use a slightly different normalization)
(2) GraphSAGE

\[ h_v^{(l)} = \sigma \left( W^{(l)} \cdot \operatorname{CONCAT} \left( h_v^{(l-1)}, \operatorname{AGG} \left( \{ h_u^{(l-1)}, \forall u \in N(v) \} \right) \right) \right) \]

How to write this as Message + Aggregation?

- **Message** is computed within the \( \operatorname{AGG}(\cdot) \)
- **Two-stage aggregation**
  - **Stage 1:** Aggregate from node neighbors
    \[ h_{N(v)}^{(l)} \leftarrow \operatorname{AGG} \left( \{ h_u^{(l-1)}, \forall u \in N(v) \} \right) \]
  - **Stage 2:** Further aggregate over the node itself
    \[ h_v^{(l)} \leftarrow \sigma \left( W^{(l)} \cdot \operatorname{CONCAT}(h_v^{(l-1)}, h_{N(v)}^{(l)}) \right) \]
**GraphSAGE Neighbor Aggregation**

- **Mean:** Take a weighted average of neighbors

\[
AGG = \frac{\sum_{u \in N(v)} h_u^{(l-1)}}{|N(v)|}
\]

- **Pool:** Transform neighbor vectors and apply symmetric vector function $\text{Mean}(\cdot)$ or $\text{Max}(\cdot)$

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

- **LSTM:** Apply LSTM to reshuffled of neighbors

\[
AGG = \text{LSTM}([h_u^{(l-1)}, \forall u \in \pi(N(v))])
\]
GraphSAGE: L2 Normalization

- **L₂ Normalization:**
  - **Optional:** Apply L₂ normalization to \( h^{(l)}_v \) at every layer
  - \( h^{(l)}_v \leftarrow \frac{h^{(l)}_v}{\|h^{(l)}_v\|_2} \quad \forall v \in V \) where \( \|u\|_2 = \sqrt{\sum_i u_i^2} \) (L₂-norm)
  - Without L₂ normalization, the embedding vectors have different scales (L₂-norm) for vectors
  - In some cases (not always), normalization of embedding results in performance improvement
  - After L₂ normalization, all vectors will have the same L₂-norm
(3) Graph Attention Networks

\[ h^{(l)}_v = \sigma \left( \sum_{u \in N(v)} \alpha_{vu} W^{(l)} h^{(l-1)}_u \right) \]

Attention weights

In GCN / GraphSAGE

- \( \alpha_{vu} = \frac{1}{|N(v)|} \) is the **weighting factor (importance)** of node \( u \)'s message to node \( v \)
- \( \Rightarrow \) \( \alpha_{vu} \) is defined **explicitly** based on the structural properties of the graph (node degree)
- \( \Rightarrow \) All neighbors \( u \in N(v) \) are equally important to node \( v \)
(3) **Graph Attention Networks**

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

**Attention weights**

Not all node’s neighbors are equally important

- **Attention** is inspired by cognitive attention.
- The **attention** $\alpha_{vu}$ focuses on the important parts of the input data and fades out the rest.
  - **Idea**: the NN should devote more computing power on that small but important part of the data.
  - Which part of the data is more important depends on the context and is learned through training.
Can we do better than simple neighborhood aggregation?

Can we let weighting factors $\alpha_{vu}$ to be learned?

- **Goal:** Specify *arbitrary importance* to different neighbors of each node in the graph
- **Idea:** Compute embedding $h_v^{(l)}$ of each node in the graph following an *attention strategy*:
  - Nodes attend over their neighborhoods’ message
  - Implicitly specifying different weights to different nodes in a neighborhood

[Velickovic et al., ICLR 2018; Vaswani et al., NIPS 2017]
Let $\alpha_{vu}$ be computed as a byproduct of an attention mechanism $a$:

1. Let $a$ compute **attention coefficients** $e_{vu}$ across pairs of nodes $u, v$ based on their messages:

$$e_{vu} = a(W^{(l)}h^{(l-1)}_u, W^{(l)}h^{(l-1)}_v)$$

- $e_{vu}$ indicates the importance of $u$'s message to node $v$

$$e_{AB} = a(W^{(l)}h^{(l-1)}_A, W^{(l)}h^{(l-1)}_B)$$
Attention Mechanism (2)

- **Normalize** $e_{vu}$ into the **final attention weight** $\alpha_{vu}$
  - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:
    \[
    \alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}
    \]

- **Weighted sum** based on the **final attention weight** $\alpha_{vu}$
    \[
    h_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} W^{(l)} h_u^{(l-1)}\right)
    \]

**Weighted sum using** $\alpha_{AB}$, $\alpha_{AC}$, $\alpha_{AD}$:

\[
\begin{align*}
    h_A^{(l)} &= \sigma\left(\alpha_{AB} W^{(l)} h_B^{(l-1)} + \alpha_{AC} W^{(l)} h_C^{(l-1)} + \\
                &\quad \quad + \alpha_{AD} W^{(l)} h_D^{(l-1)}\right)
\end{align*}
\]
What is the form of attention mechanism $a$?

- The approach is agnostic to the choice of $a$
  - E.g., use a simple single-layer neural network
    - $a$ have trainable parameters (weights in the Linear layer)

Parameters of $a$ are trained jointly:

- Learn the parameters together with weight matrices (i.e., other parameter of the neural net $W^{(l)}$) in an end-to-end fashion
Multi-head attention: Stabilizes the learning process of attention mechanism

- Create multiple attention scores (each replica with a different set of parameters):
  \[
  h_v^{(l)}[1] = \sigma \left( \sum_{u \in N(v)} \alpha_{vu}^1 W^{(l)} h_u^{(l-1)} \right) \\
  h_v^{(l)}[2] = \sigma \left( \sum_{u \in N(v)} \alpha_{vu}^2 W^{(l)} h_u^{(l-1)} \right) \\
  h_v^{(l)}[3] = \sigma \left( \sum_{u \in N(v)} \alpha_{vu}^3 W^{(l)} h_u^{(l-1)} \right)
  \]

- Outputs are aggregated:
  - By concatenation or summation
  - \[ h_v^{(l)} = \text{AGG}(h_v^{(l)}[1], h_v^{(l)}[2], h_v^{(l)}[3]) \]
Benefits of Attention Mechanism

- **Key benefit:** Allows for (implicitly) specifying different importance values $\alpha_{vu}$ to different neighbors

- **Computationally efficient:**
  - Computation of attentional coefficients can be parallelized across all edges of the graph
  - Aggregation may be parallelized across all nodes

- **Storage efficient:**
  - Sparse matrix operations do not require more than $O(V + E)$ entries to be stored
  - Fixed number of parameters, irrespective of graph size

- **Localized:**
  - Only attends over local network neighborhoods

- **Inductive capability:**
  - It is a shared \textit{edge-wise} mechanism
  - It does not depend on the global graph structure
Attention mechanism can be used with many different graph neural network models

In many cases, attention leads to performance gains

- **t-SNE plot of GAT-based node embeddings:**
  - Node color: 7 publication classes
  - Edge thickness: Normalized attention coefficients between nodes $i$ and $j$, across eight attention heads, $\sum_k(\alpha^k_{ij} + \alpha^k_{ji})$
In practice, these classic GNN layers are a great starting point.

- We can often get better performance by considering a general GNN layer design.
- Concretely, we can include modern deep learning modules that proved to be useful in many domains.
Many modern deep learning modules can be incorporated into a GNN layer

- **Batch Normalization:**
  - Stabilize neural network training

- **Dropout:**
  - Prevent overfitting

- **Attention/Gating:**
  - Control the importance of a message

- **More:**
  - Any other useful deep learning modules
Batch Normalization

- **Goal**: Stabilize neural networks training
- **Idea**: Given a batch of inputs (node embeddings)
  - Re-center the node embeddings into zero mean
  - Re-scale the variance into unit variance

**Input**: \( X \in \mathbb{R}^{N \times D} \\
N \) node embeddings

**Trainable Parameters**: \( \gamma, \beta \in \mathbb{R}^D \)

**Output**: \( Y \in \mathbb{R}^{N \times D} \\
Normalized node embeddings

**Step 1**: Compute the mean and variance over \( N \) embeddings

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \\
\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
\]

**Step 2**: Normalize the feature using computed mean and variance

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \\
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j
\]
**Goal**: Regularize a neural net to prevent overfitting.

**Idea**:
- **During training**: with some probability $p$, randomly set neurons to zero (turn off)
- **During testing**: Use all the neurons for computation
In GNN, Dropout is applied to the linear layer in the message function.

A simple message function with linear layer:

\[ m_u^{(l)} = W^{(l)} h_u^{(l-1)} \]

Visualization of a linear layer
Activation (Non-linearity)

Apply activation to $i$-th dimension of embedding $x$

- **Rectified linear unit (ReLU)**
  \[
  \text{ReLU}(x_i) = \max(x_i, 0)
  \]
  - Most commonly used

- **Sigmoid**
  \[
  \sigma(x_i) = \frac{1}{1 + e^{-x_i}}
  \]
  - Used only when you want to restrict the range of your embeddings

- **Parametric ReLU**
  \[
  \text{PReLU}(x_i) = \max(x_i, 0) + a_i \min(x_i, 0)
  \]
  - $a_i$ is a trainable parameter
  - Empirically performs better than ReLU
Summary: Modern deep learning modules can be included into a GNN layer for better performance.

Designing novel GNN layers is still an active research frontier!

Suggested resources: You can explore diverse GNN designs or try out your own ideas in GraphGym.
Stanford CS224W: Stacking Layers of a GNN

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
How to connect GNN layers into a GNN?

• Stack layers sequentially
• Ways of adding skip connections

(3) Layer connectivity

GNN Layer 1

GNN Layer 2
Stacking GNN Layers

How to construct a Graph Neural Network?

- **The standard way:** Stack GNN layers sequentially
- **Input:** Initial raw node feature $x_v$
- **Output:** Node embeddings $h_v^{(L)}$ after $L$ GNN layers

$h_v^{(0)} = x_v$

$\begin{align*}
  h_v^{(1)} & \\
  h_v^{(2)} & \\
  h_v^{(3)} & \\
\end{align*}$
The Over-smoothing Problem

- The Issue of stacking many GNN layers
  - GNN suffers from the over-smoothing problem
- The over-smoothing problem: all the node embeddings converge to the same value
  - This is bad because we want to use node embeddings to differentiate nodes
- Why does the over-smoothing problem happen?
Receptive Field of a GNN

- **Receptive field**: the set of nodes that determine the embedding of a node of interest

- In a $K$-layer GNN, each node has a receptive field of $K$-hop neighborhood

Receptive field for 1-layer GNN

Receptive field for 2-layer GNN

Receptive field for 3-layer GNN
Receptive field overlap for two nodes

The shared neighbors quickly grows when we increase the number of hops (num of GNN layers)

1-hop neighbor overlap
Only 1 node

2-hop neighbor overlap
About 20 nodes

3-hop neighbor overlap
Almost all the nodes!
We can explain over-smoothing via the notion of receptive field

- We knew the embedding of a node is determined by its receptive field
  - If two nodes have highly-overlapped receptive fields, then their embeddings are highly similar

- Stack many GNN layers → nodes will have highly-overlapped receptive fields → node embeddings will be highly similar → suffer from the over-smoothing problem

Next: how do we overcome over-smoothing problem?
What do we learn from the over-smoothing problem?

Lesson 1: Be cautious when adding GNN layers

- Unlike neural networks in other domains (CNN for image classification), adding more GNN layers do not always help.
- **Step 1:** Analyze the necessary receptive field to solve your problem. E.g., by computing the diameter of the graph.
- **Step 2:** Set number of GNN layers $L$ to be a bit more than the receptive field we like. Do not set $L$ to be unnecessarily large!

Question: How to enhance the expressive power of a GNN, if the number of GNN layers is small?
How to make a shallow GNN more expressive?

Solution 1: Increase the expressive power within each GNN layer

- In our previous examples, each transformation or aggregation function only include one linear layer
- We can make aggregation / transformation become a deep neural network!

If needed, each box could include a 3-layer MLP

(1) Transformation
(2) Aggregation
Expressive Power for Shallow GNNs

- **How to make a shallow GNN more expressive?**
- **Solution 2**: Add layers that do not pass messages
  - A GNN does not necessarily only contain GNN layers
    - E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process layers** and **post-process layers**

![Diagram of GNN layers with MLP and GNN layers]

**Pre-processing layers**: Important when encoding node features is necessary. E.g., when nodes represent images/text

**Post-processing layers**: Important when reasoning / transformation over node embeddings are needed. E.g., graph classification, knowledge graphs

In practice, adding these layers works great!
What if my problem still requires many GNN layers?

Lesson 2: Add skip connections in GNNs

- **Observation from over-smoothing:** Node embeddings in earlier GNN layers can sometimes better differentiate nodes
- **Solution:** We can increase the impact of earlier layers on the final node embeddings, by adding shortcuts in GNNs

### Idea of skip connections:

Before adding shortcuts: 
$$F(x)$$

After adding shortcuts: 
$$F(x) + x$$
Why do skip connections work?

- **Intuition:** Skip connections create a mixture of models
- $N$ skip connections $\rightarrow 2^N$ possible paths
- Each path could have up to $N$ modules
- We automatically get a mixture of shallow GNNs and deep GNNs

\[
2 \times 2 \times 2 = 2^3 = 8
\]
Example: GCN with Skip Connections

- A standard GCN layer

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

This is our \( F(x) \)

- A GCN layer with skip connection

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

\[ F(x) + x \]
Other Options of Skip Connections

- **Other options**: Directly skip to the last layer
  - The final layer directly aggregates from the all the node embeddings in the previous layers
Stanford CS224W: Graph Manipulation in GNNs

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
General GNN Framework

Idea: Raw input graph ≠ computational graph
- Graph feature augmentation
- Graph structure manipulation

(4) Graph manipulation
Our assumption so far has been

- **Raw input graph = computational graph**

**Reasons for breaking this assumption**

- **Feature level:**
  - The input graph **lacks features** \(\rightarrow\) feature augmentation

- **Structure level:**
  - The graph is **too sparse** \(\rightarrow\) inefficient message passing
  - The graph is **too dense** \(\rightarrow\) message passing is too costly
  - The graph is **too large** \(\rightarrow\) cannot fit the computational graph into a GPU

- It’s just **unlikely that the input graph happens to be the optimal computation graph** for embeddings
Graph Manipulation Approaches

- **Graph Feature manipulation**
  - The input graph lacks features → feature augmentation

- **Graph Structure manipulation**
  - The graph is too sparse → Add virtual nodes / edges
  - The graph is too dense → Sample neighbors when doing message passing
  - The graph is too large → Sample subgraphs to compute embeddings
    - Will cover later in lecture: Scaling up GNNs
Why do we need feature augmentation?

- **(1) Input graph does not have node features**
  - This is common when we only have the adj. matrix

- **Standard approaches:**
  - a) Assign constant values to nodes
Feature Augmentation on Graphs

Why do we need feature augmentation?

- **(1) Input graph does not have node features**
  - This is common when we only have the adj. matrix

- **Standard approaches:**
  - **b) Assign unique IDs to nodes**
  - These IDs are converted into **one-hot vectors**

One-hot vector for node with ID=5

\[ [0, 0, 0, 0, 1, 0] \]

Total number of IDs = 6
Feature Augmentation on Graphs

- Feature augmentation: **constant vs. one-hot**

<table>
<thead>
<tr>
<th></th>
<th>Constant node feature</th>
<th>One-hot node feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressive power</strong></td>
<td><strong>Medium.</strong> All the nodes are identical, but GNN can still learn from the graph structure</td>
<td><strong>High.</strong> Each node has a unique ID, so node-specific information can be stored</td>
</tr>
<tr>
<td><strong>Inductive learning</strong></td>
<td><strong>High.</strong> Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN</td>
<td><strong>Low.</strong> Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn’t know how to embed unseen IDs</td>
</tr>
<tr>
<td><strong>Computational cost</strong></td>
<td><strong>Low.</strong> Only 1 dimensional feature</td>
<td><strong>High.</strong> $O(</td>
</tr>
<tr>
<td><strong>Use cases</strong></td>
<td>Any graph, inductive settings (generalize to new nodes)</td>
<td>Small graph, transductive settings (no new nodes)</td>
</tr>
</tbody>
</table>
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- **Example:** Cycle count feature
  - Can GNN learn the length of a cycle that $v_1$ resides in?
  - Unfortunately, no

$v_1$ resides in a cycle with length 3

$v_1$ resides in a cycle with length 4
- $v_1$ cannot differentiate which graph it resides in
  - Because all the nodes in the graph have degree of 2
  - The computational graphs will be the same binary tree

$v_1$ resides in a cycle with length 3
$v_1$ resides in a cycle with length 4
$v_1$ resides in a cycle with infinite length

The computational graphs for node $v_1$ are always the same
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- **Solution:**
  - We can use cycle count as augmented node features

---

**Augmented node feature for** $v_1$

- $[0, 0, 0, 1, 0, 0]$ - $v_1$ resides in a cycle with length 3

**Augmented node feature for** $v_1$

- $[0, 0, 0, 0, 1, 0]$ - $v_1$ resides in a cycle with length 4
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- Other commonly used augmented features:
  - Clustering coefficient
  - PageRank
  - Centrality
  - ...
- Any feature we have introduced in Lecture 2 can be used!
Add Virtual Nodes / Edges

- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
  - **Common approach:** Connect 2-hop neighbors via virtual edges
  - **Intuition:** Instead of using adj. matrix $A$ for GNN computation, use $A + A^2$

- **Use cases:** Bipartite graphs
  - Author-to-papers (they authored)
  - 2-hop virtual edges make an author-author collaboration graph
**Add Virtual Nodes / Edges**

- **Motivation:** Augment sparse graphs
- **(2) Add virtual nodes**
  - The virtual node will connect to all the nodes in the graph
    - Suppose in a sparse graph, two nodes have shortest path distance of 10
    - After adding the virtual node, all the nodes will have a distance of 2
      - Node A – Virtual node – Node B
  - **Benefits:** Greatly improves message passing in sparse graphs
Node Neighborhood Sampling

- **Previously:**
  - All the nodes are used for message passing

- **New idea:** (Randomly) sample a node’s neighborhood for message passing

Hamilton et al. *Inductive Representation Learning on Large Graphs*, NeurIPS 2017
For example, we can randomly choose 2 neighbors to pass messages

- Only nodes $B$ and $D$ will pass message to $A$
Next time when we compute the embeddings, we can sample different neighbors
- Only nodes $C$ and $D$ will pass message to $A$
In expectation, we can get embeddings similar to the case where all the neighbors are used

- **Benefits:** greatly reduce computational cost
- And in practice it works great!

Ying et al. *Graph Convolutional Neural Networks for Web-Scale Recommender Systems*, KDD 2018
Recap: A general perspective for GNNs

- **GNN Layer:**
  - Transformation + Aggregation
  - Classic GNN layers: GCN, GraphSAGE, GAT

- **Layer connectivity:**
  - Deciding number of layers
  - Skip connections

- **Graph Manipulation:**
  - Feature augmentation
  - Structure manipulation

Next: GNN objectives, GNN in practice