Stanford CS224W: Course Project Instructions

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

Course Project Overview

- We ask you to develop **a tutorial/case-study** on **applying state-of-the-art graph ML to a real-world problem using PyG.**
  - Someone can then follow your tutorial and learn how to apply ML to a real-world problem.
- The project is **open-ended**: You can choose models and problems to work on.
  - In the **instruction doc**, we provide some examples of models and datasets.
- The final product will be **a draft blog post** that you share with us privately.
  - With your permission we would publish best ones at PyG.org.
- **Groups of 3 students** are strongly recommended; groups of 1-2 also permitted.

Why Blog Posts?

- A great exercise for you to understand and implement **graph ML models applied to real-world problems**.
- A great lasting resource **for the broader community to study graph ML.**
  - Blog posts are more accessible than technical reports.
  - We will publicize selected blog posts from you!

What are Good Blog Posts?

- **Good blog posts should include**
  - **Step-by-step explanation of graph ML techniques**
    - Assume your readers are
      - familiar with ML, deep learning, and Pytorch
      - not familiar with graph ML and PyG
  - **Visualization**
    - To explain techniques and results, Gifs > Images > Text
    - The more visualization, the better.
  - **Code snippets of PyG/Pytorch**
  - **Link to Google Colab to reproduce your results**
    - Your Colab should be readable and include enough documentations.

Application Domains

- **Application domains of graph ML includes**
  - Recommender systems
  - Molecule classification
  - Paper classification in citation networks
  - Knowledge graph completion
  - Product classification in co-purchasing graphs
  - Fraud detection in transaction networks
  - Protein function prediction in protein-protein interaction networks
  - Friend recommendation in social networks

Finding Graph ML Models

- **OGB Leaderboard**
  - https://ogb.stanford.edu/docs/leader_overview/
- **Top ML conference papers:**
  - KDD
  - ICLR
  - ICML
  - NeurIPS
  - WWW
  - WSDM

Tips: Narrow down relevant papers by searching titles (e.g., containing “graph”).

Next Step: Project Proposal

- **By October 19, 11:59pm PT**
- The proposal should include the following:
  - **Application domain**
    - Which dataset are you planning to use?
    - Describe the dataset, prediction tasks, and metric.
    - Why did you choose the dataset?
  - **Graph ML technique that you want to apply**
    - Graph ML model you plan to use
    - Describe the model (using figures and equations)
    - Why the model is appropriate for the dataset?

Next Step: Project Proposal

- **Special OH this week dedicated to project**
  - Jure's OH: 1-3pm on Wed 10/13
    - 10 min slots: https://calendly.com/cs224w-oh/jure-project
  - Weihua's OH: 10am-12pm on Thu 10/14
    - 15 min slots: https://calendly.com/cs224w-oh/weihua-project
    - This will be recurring **every Thursday**
- **How to sign up**
  - **One person** from the group should sign up and add their group members under "Guest emails"
  - Zoom link will be in the invite – you will be let off the **waiting room** when it is your turn, **be on time**!

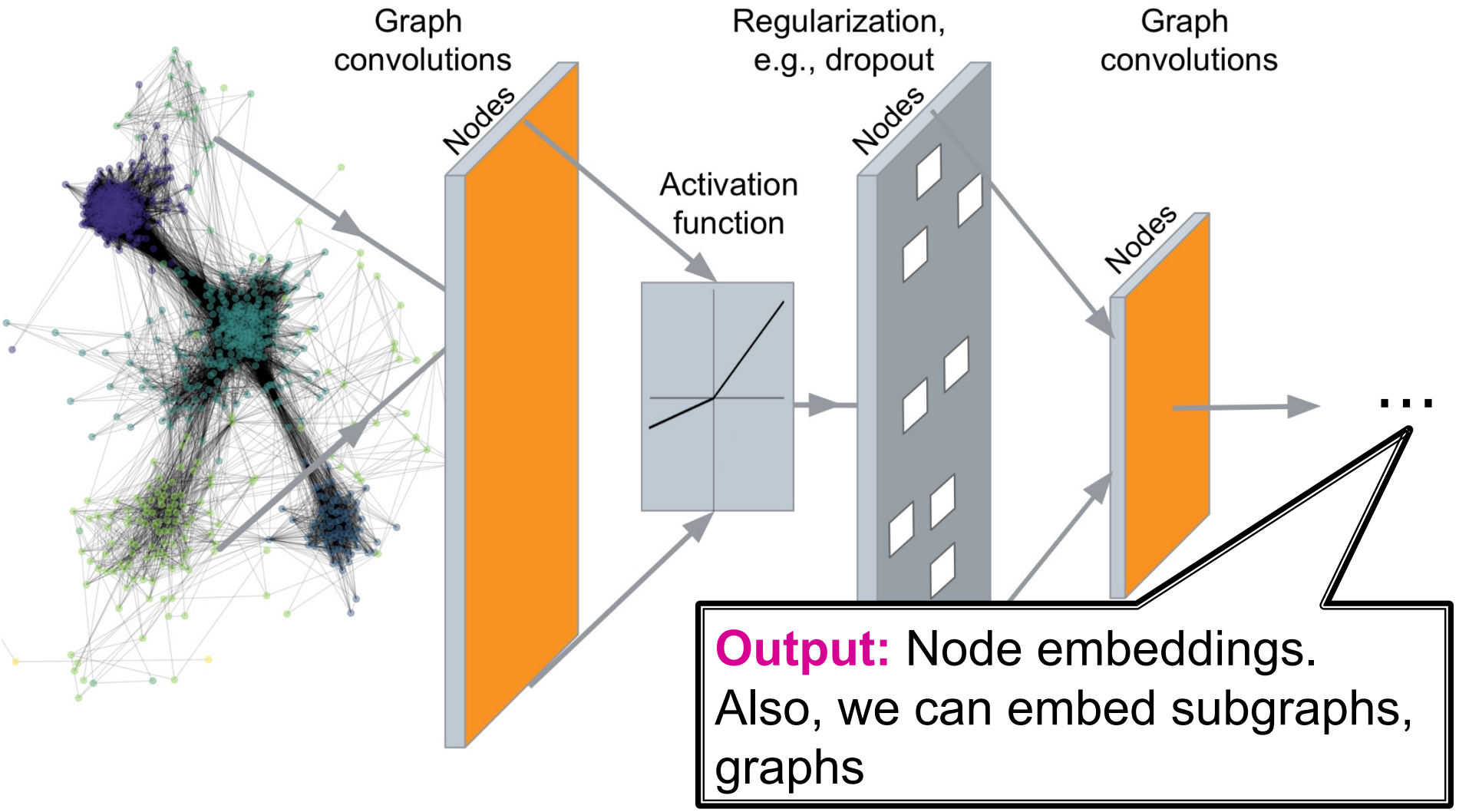
Stanford CS224W: A General Perspective on Graph Neural Networks

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

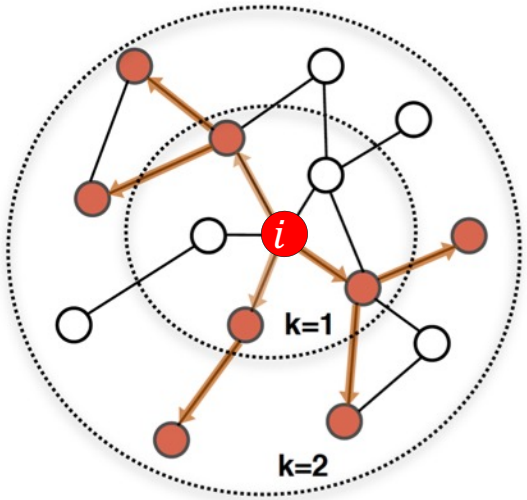
Recap: Deep Graph Encoders



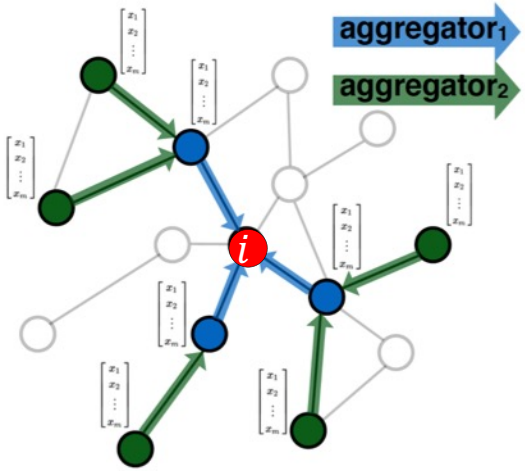
Output: Node embeddings. Also, we can embed subgraphs, graphs

Recap: Graph Neural Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

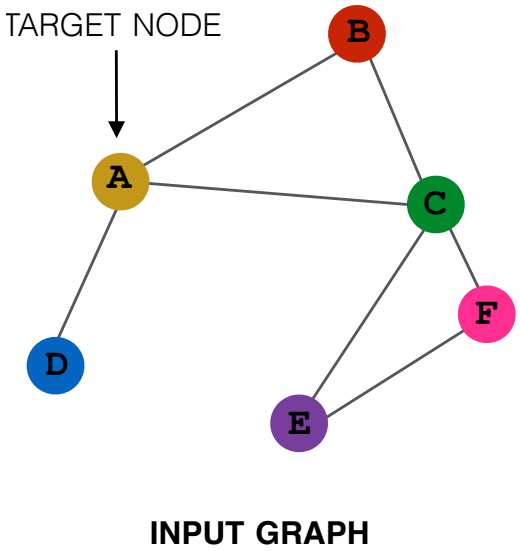


Propagate and transform information

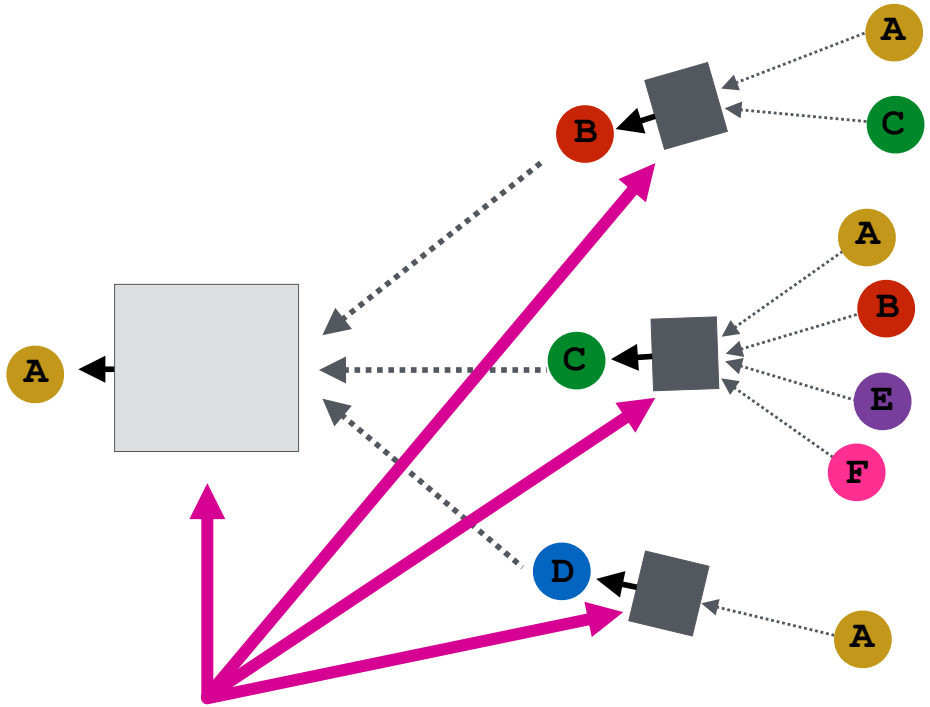
Learn how to propagate information across the graph to compute node features

Recap: Aggregate from Neighbors

- **Intuition:** Nodes aggregate information from their neighbors using neural networks



INPUT GRAPH

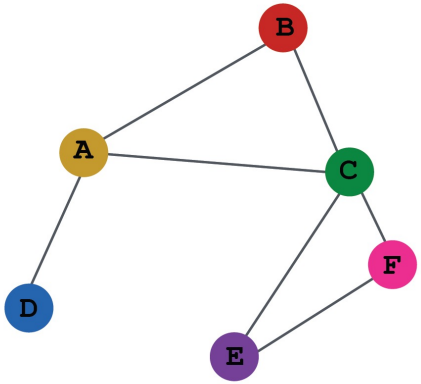


Neural networks

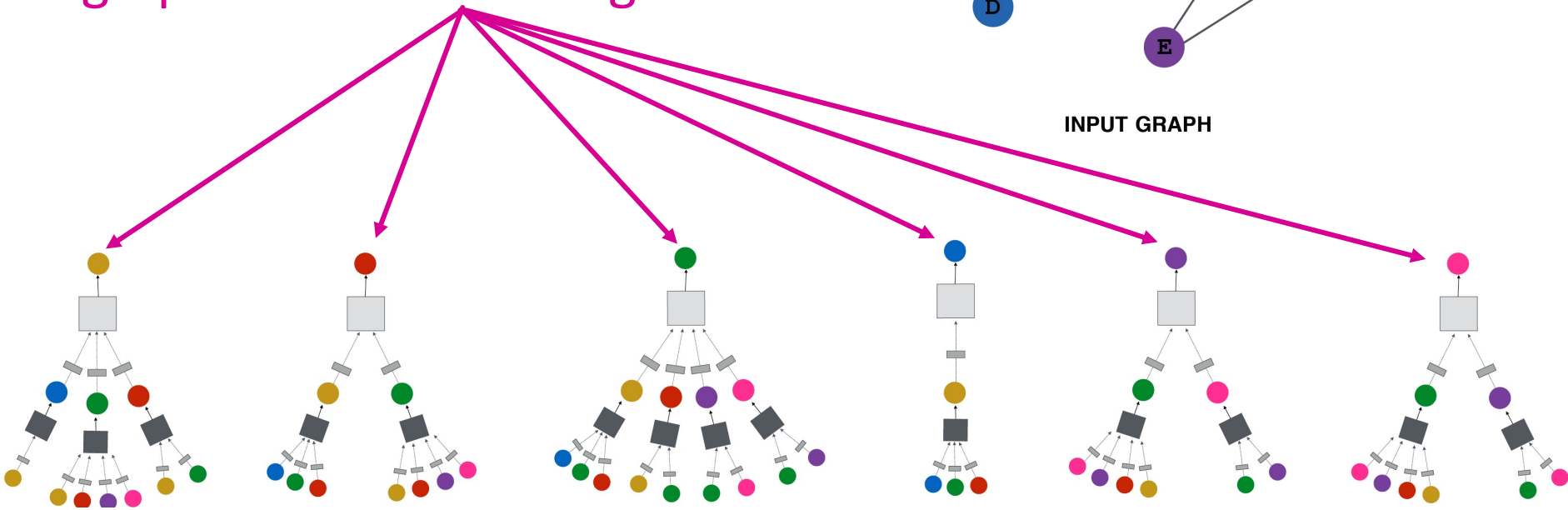
Recap: Aggregate Neighbors

- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!



INPUT GRAPH



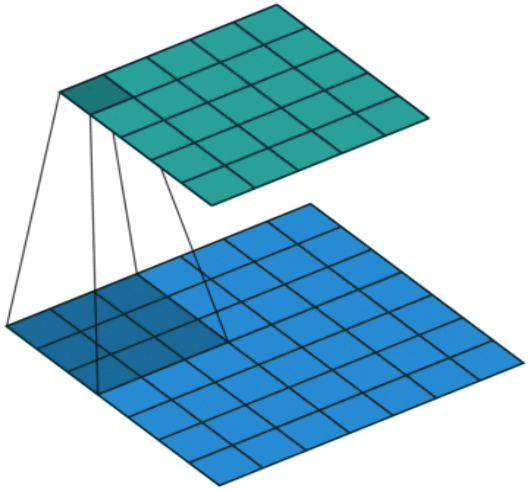
Why GNNs generalize other NNs?

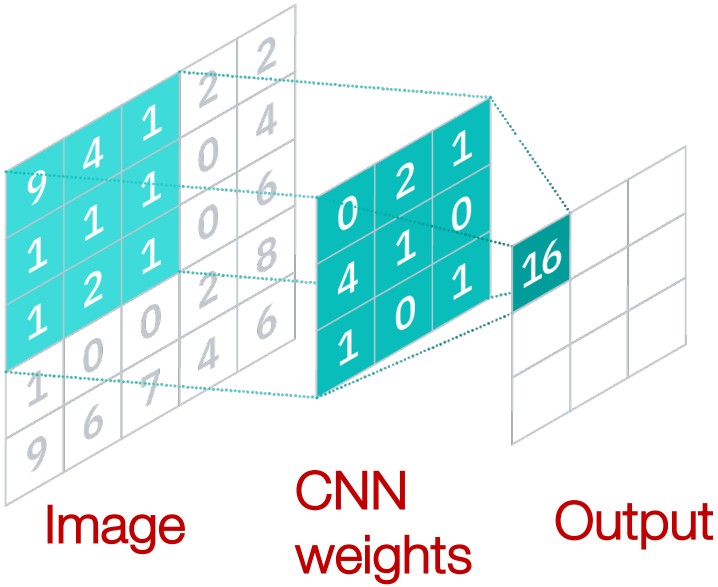
- **Defined notions of permutation invariance and equivariance.**

- **How does GNNs compare to prominent architectures such as Convolutional Neural Nets, and Transformers?**

Convolutional Neural Network

Convolutional neural network (CNN) layer with 3x3 filter:



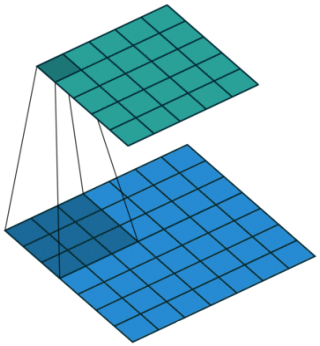


CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v) \cup \{v\}} \mathrm{W}_l^u \mathrm{h}_u^{(l)}), \quad \forall l \in \{0, \ldots, L-1\}$

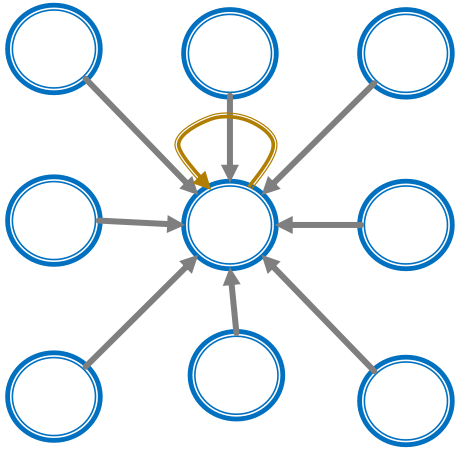
$N(v)$ represents the 8 neighbor pixels of v.

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



Image



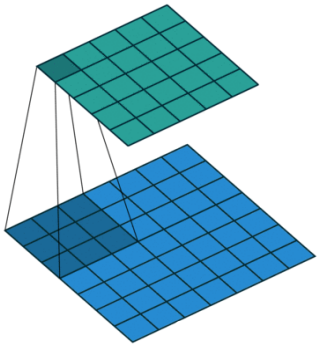
Graph

- GNN formulation (previous slide): $\mathrm{h}_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$
- CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v) \cup \{v\}} \mathrm{W}_l^u \mathrm{h}_u^{(l)}), \forall l \in \{0, \ldots, L-1\}$

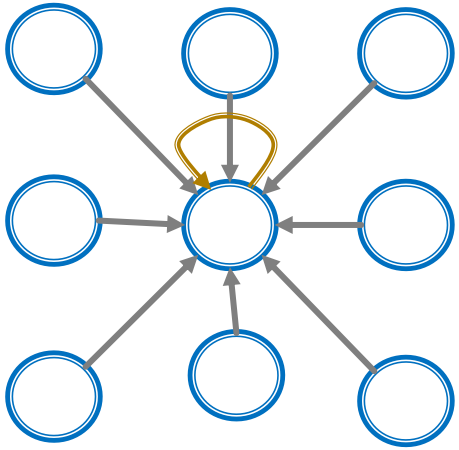
  if we rewrite: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v)} \mathbf{W}_l^u \mathrm{h}_u^{(l)} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



Image



Graph

GNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$

CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v)} \mathbf{W}_l^{u} \mathrm{h}_u^{(l)} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$

Key difference: We can learn different W_l^u for different "neighbor" u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: {(-1,-1). (-1,0), (-1, 1), …, (1, 1)}

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:

- **CNN can be seen as a special GNN with fixed neighbor size and ordering:**
  - **The size of the filter is pre-defined for a CNN.**
  - **The advantage of GNN is it processes arbitrary graphs with different degrees for each node.**

Key difference: We can learn different W_l^u for different "neighbor" u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: {(-1,-1). (-1,0), (-1, 1), …, (1, 1)}

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:

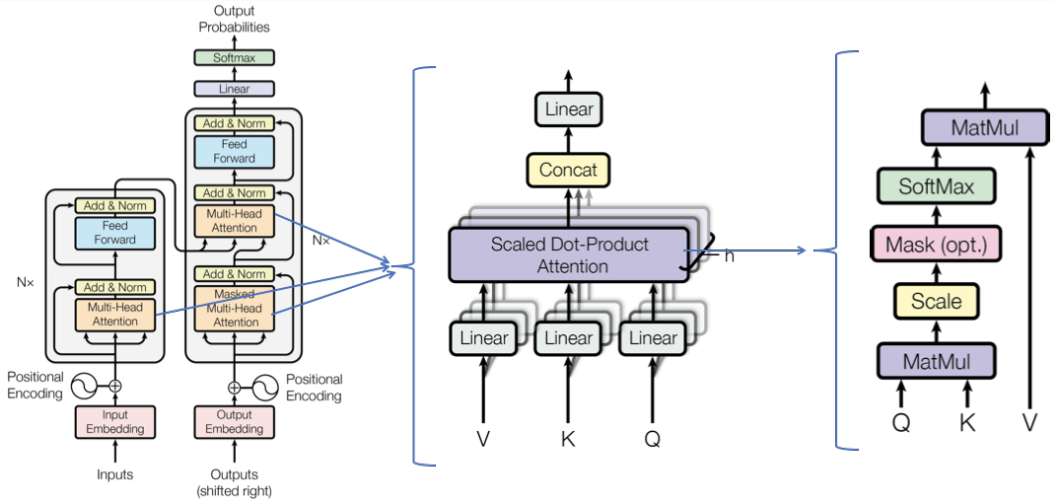
- **CNN can be seen as a special GNN with fixed neighbor size and ordering.**
- **CNN is not permutation equivariant.**
  - **Switching the order of pixels will leads to different outputs.**

Key difference: We can learn different W_l^u for different "neighbor" u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: {(-1,-1). (-1,0), (-1, 1), …, (1, 1)}

[Attention is all you need. Vaswani et al., NeurIPS 2017]

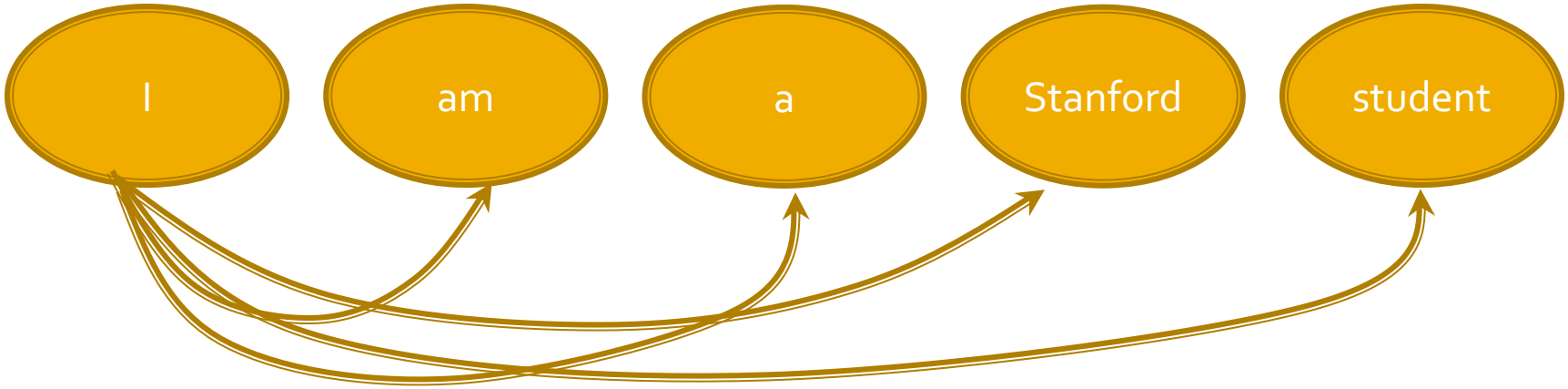
Transformer

Transformer is one of the most popular architectures that achieves great performance in many sequence modeling tasks.



Key component: self-attention

- Every token/word attends to all the other tokens/words via matrix calculation.

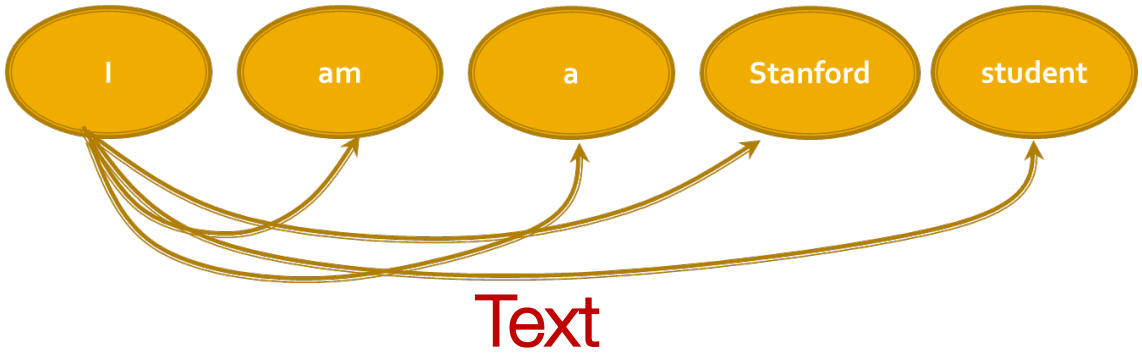


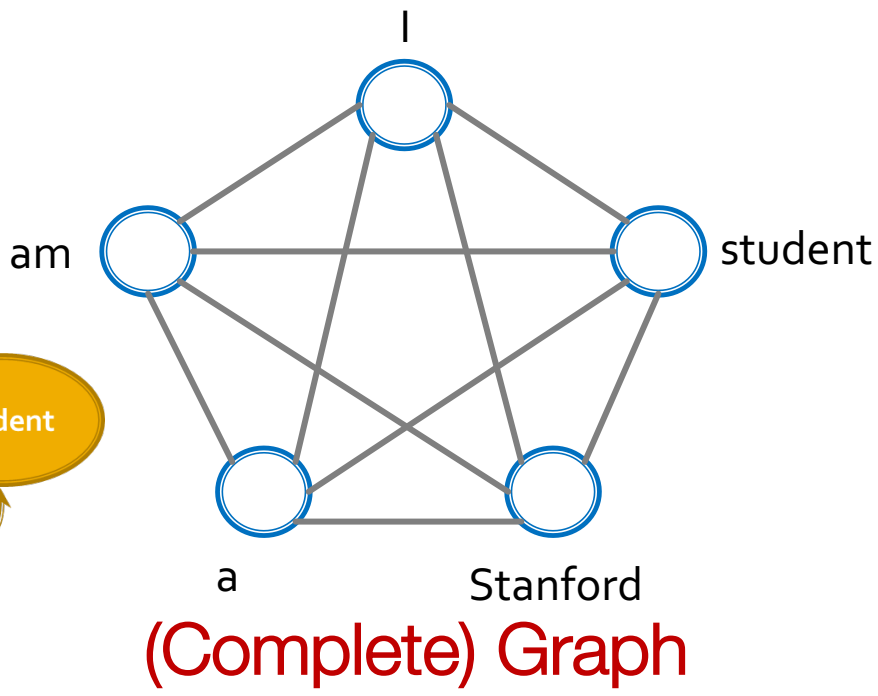
A nice blog plot for this: https://towardsdatascience.com/transformers-are-graph-neural-networks-bca9f75412aa

GNN vs. Transformer

Transformer layer can be seen as a special GNN that runs on a fully-connected "word" graph!

Since each word attends to **all the other words**, **the computation graph** of a transformer layer is identical to that of a GNN on the **fully-connected "word" graph**.





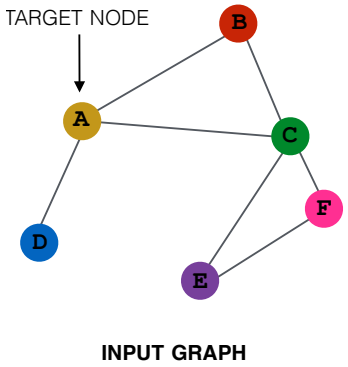
Stanford CS224W: A General Perspective on Graph Neural Networks

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

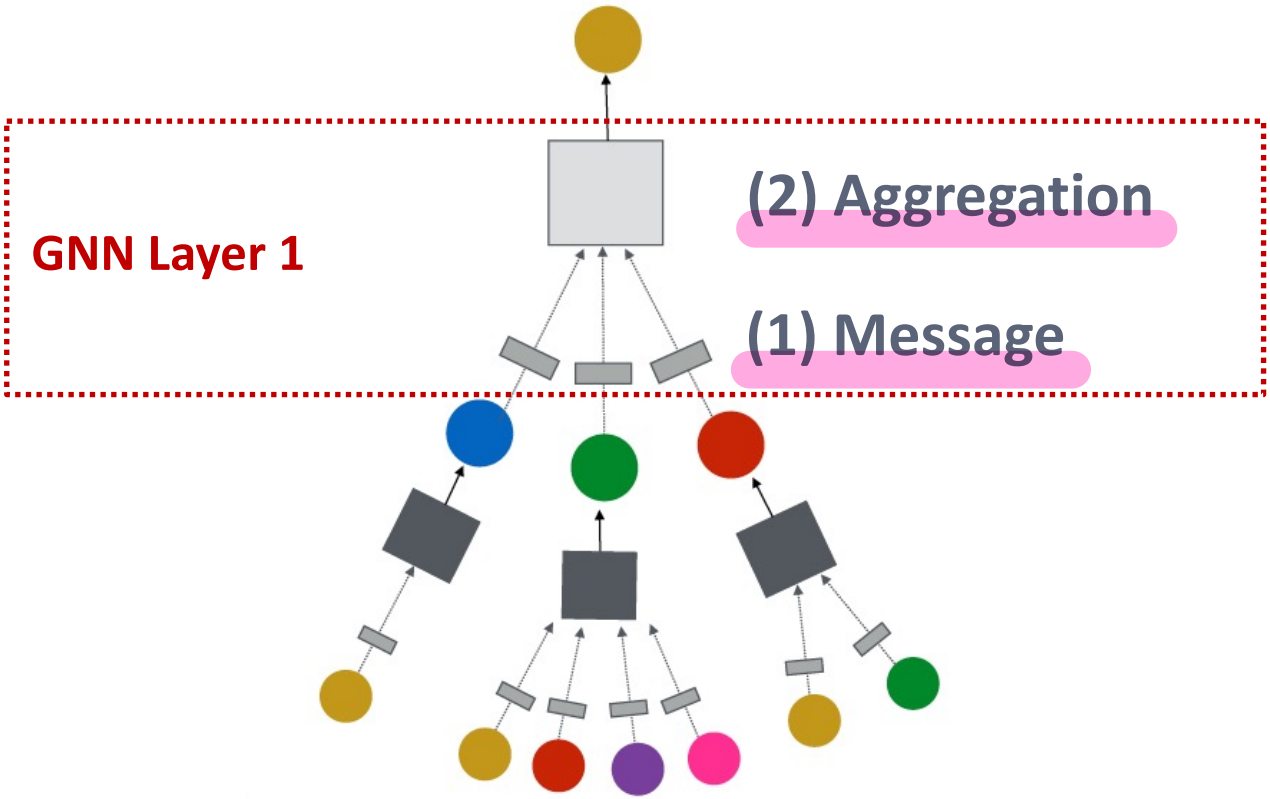
A General GNN Framework (1)



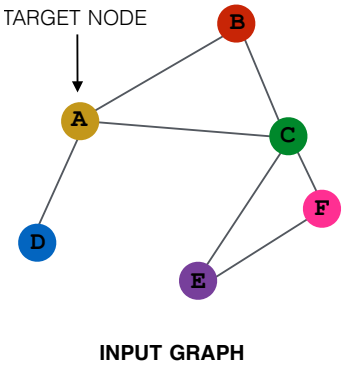
INPUT GRAPH

GNN Layer = Message + Aggregation

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



A General GNN Framework (2)

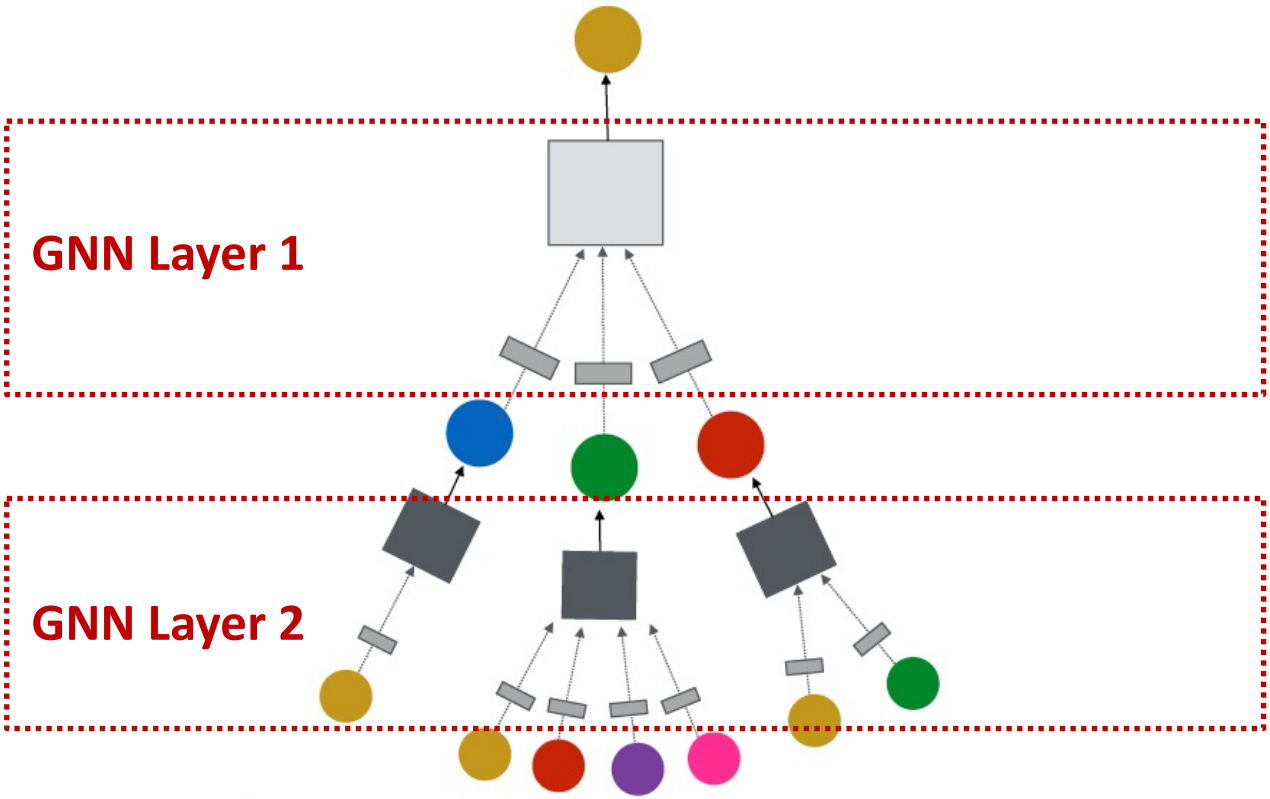


INPUT GRAPH

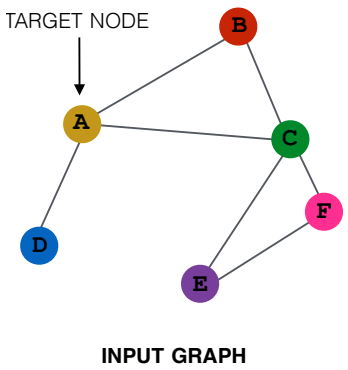
Connect GNN layers into a GNN

- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity



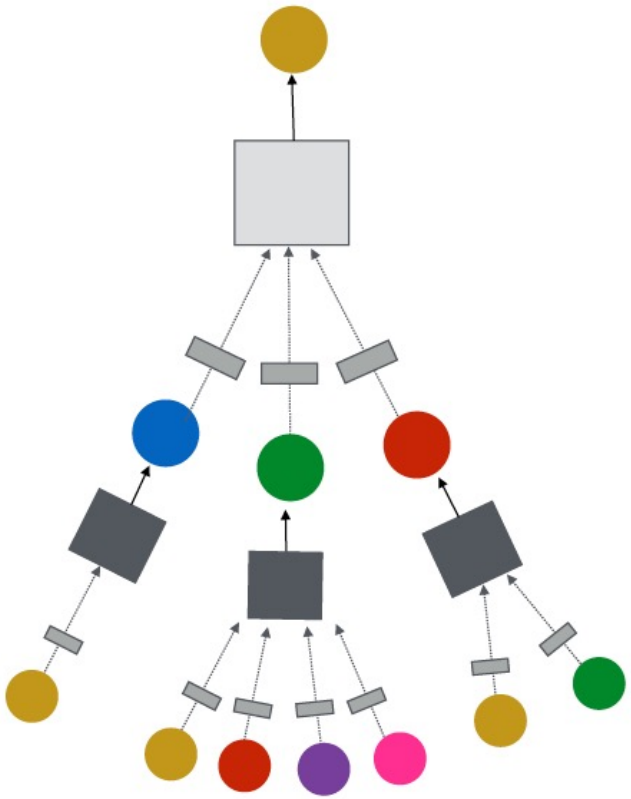
A General GNN Framework (3)



INPUT GRAPH

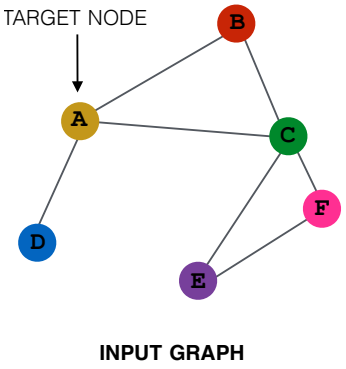
Idea: Raw input graph ≠ computational graph

- Graph feature augmentation
- Graph structure augmentation



(4) Graph augmentation

A General GNN Framework (4)

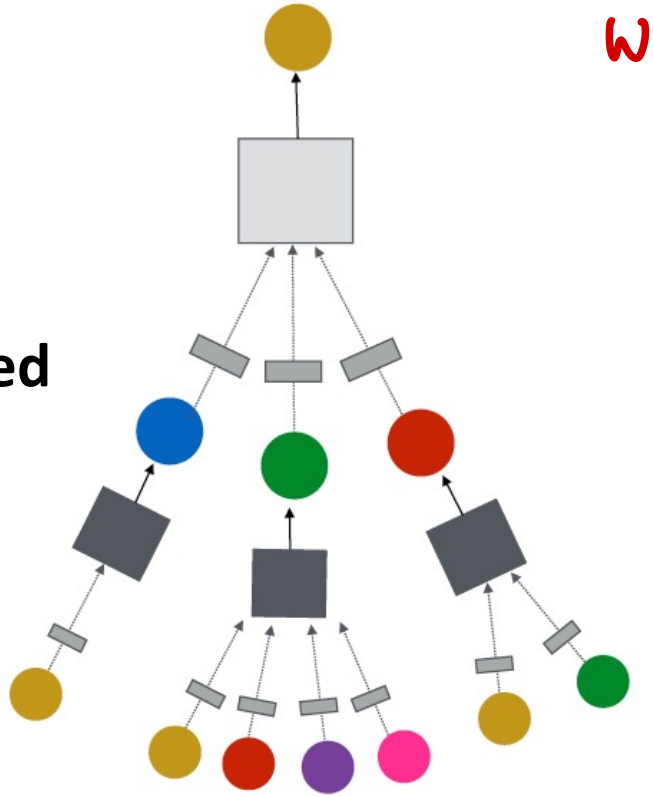


INPUT GRAPH

(5) Learning objective

What Objective Function

What task?

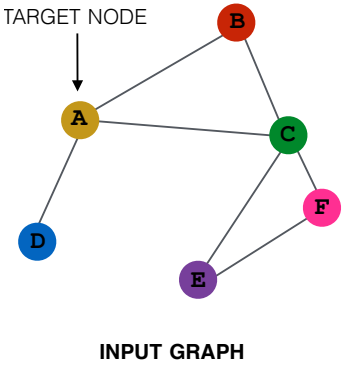


How do we train a GNN

- **Supervised/Unsupervised objectives**
- **Node/Edge/Graph level objectives**

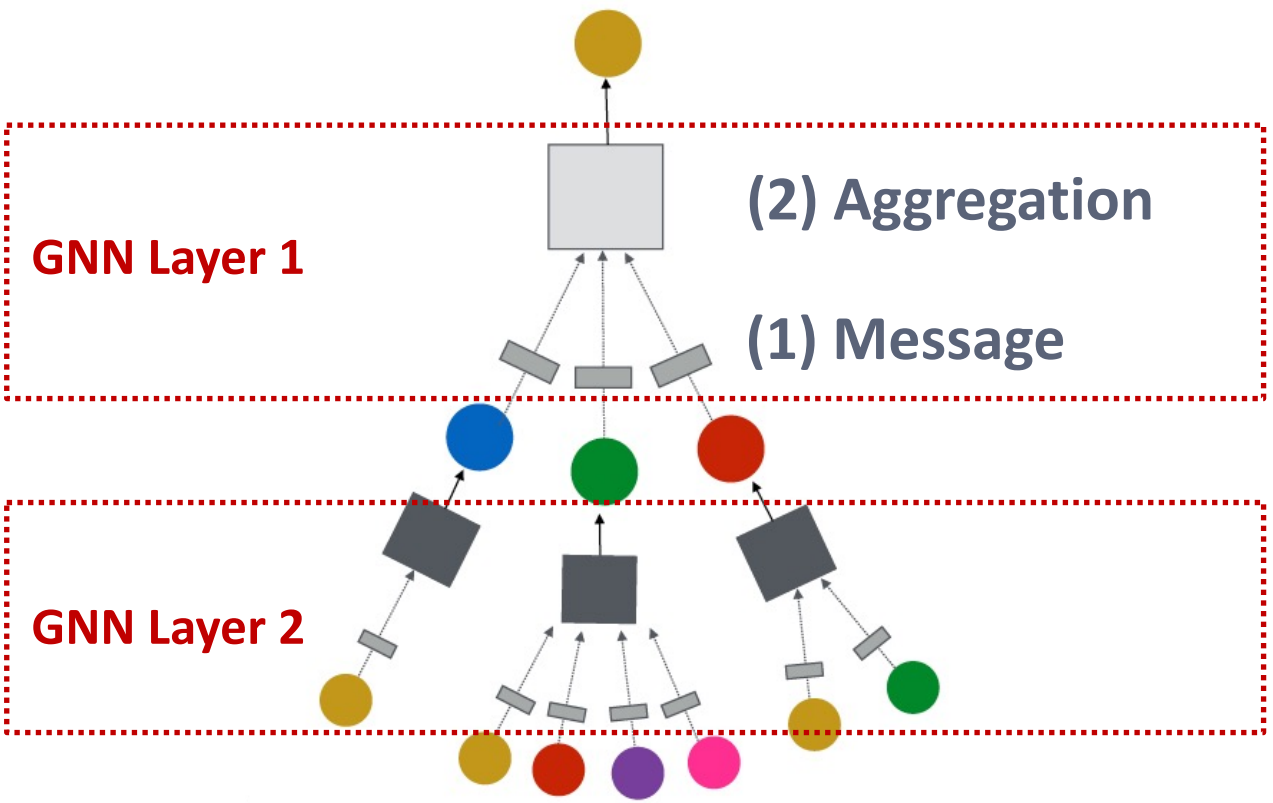
(We will discuss all of these later in class)

GNN Framework: Summary



INPUT GRAPH

(5) Learning objective



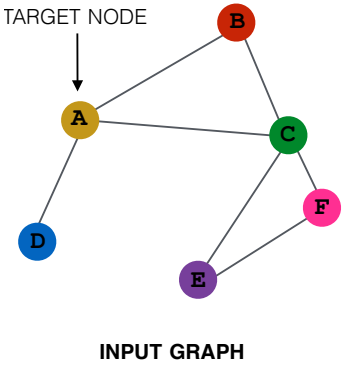
(3) Layer connectivity

(4) Graph augmentation

Stanford CS224W: A Single Layer of a GNN

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

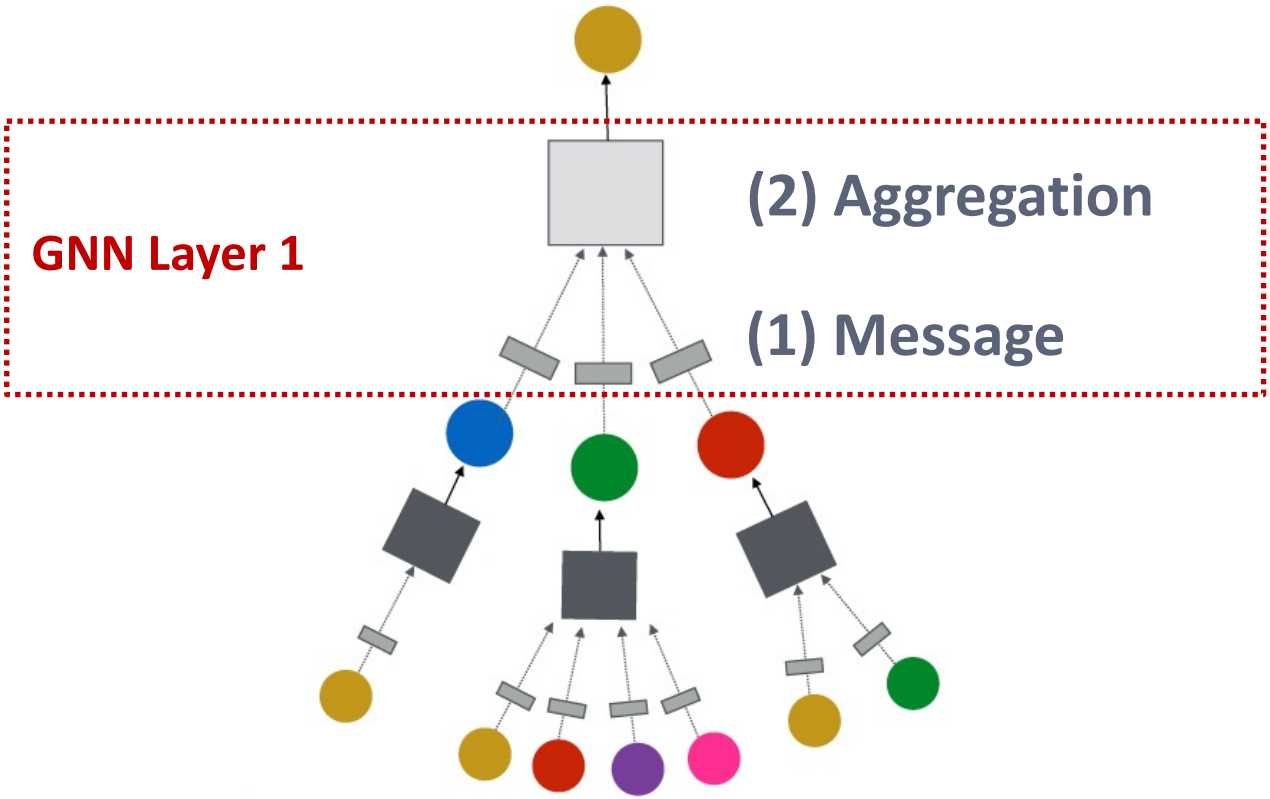
A GNN Layer



INPUT GRAPH

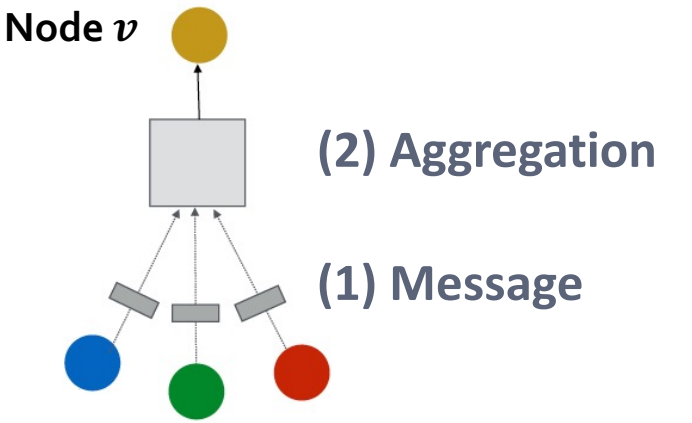
GNN Layer = Message + Aggregation

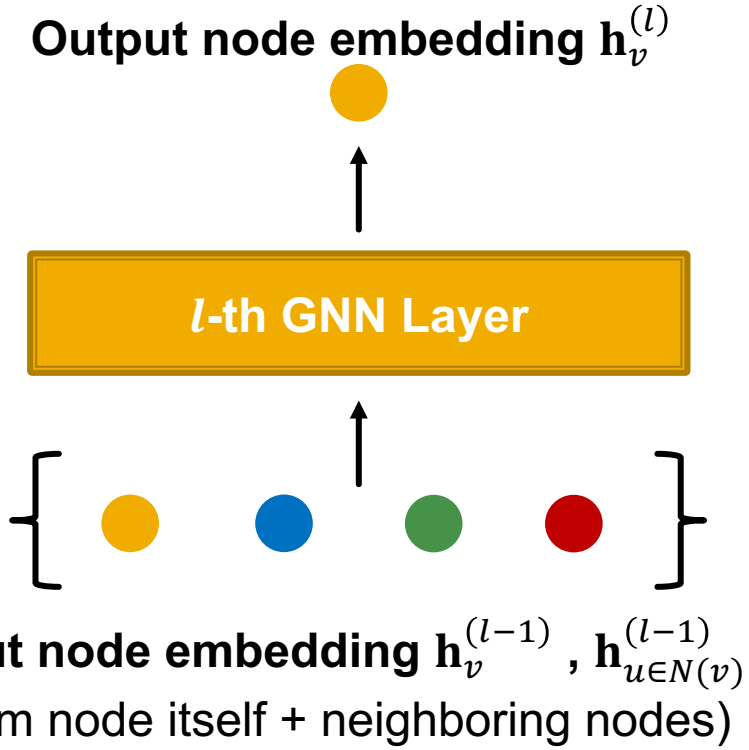
- **Different instantiations under this perspective**
- **GCN, GraphSAGE, GAT, …**



A Single GNN Layer

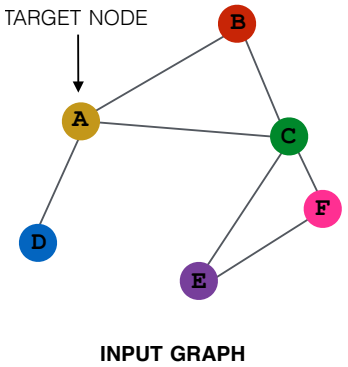
- **Idea of a GNN Layer:**
  - Compress a set of vectors into a single vector
  - **Two-step process:**
    - **(1) Message**
    - **(2) Aggregation**

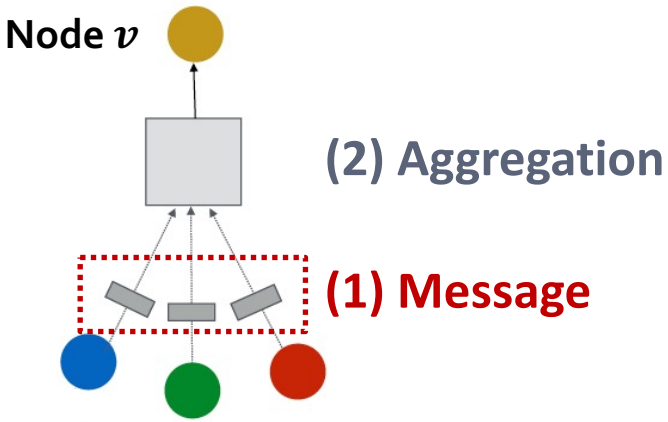




Message Computation

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





Message Aggregation

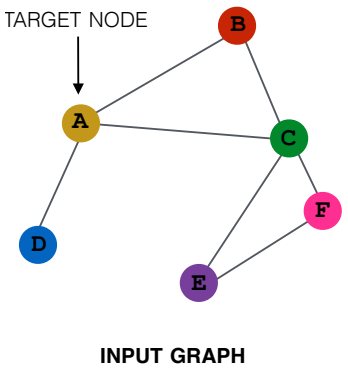
- **(2) Aggregation**

order invarient.

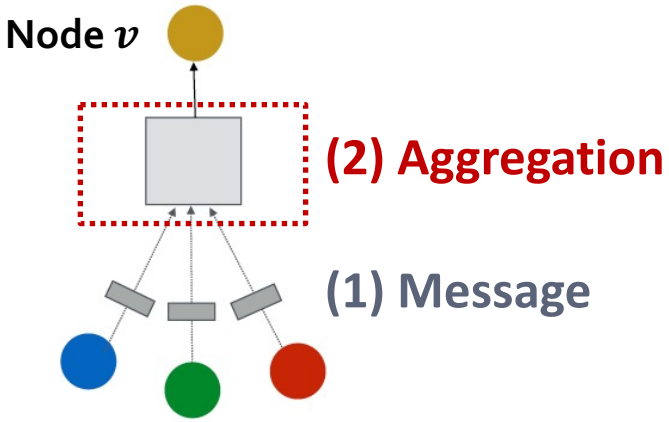
  - **Intuition:** Each node will aggregate the messages from node v's neighbors

$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)$$

  - **Example:** $\mathrm{Sum}(\cdot)$, $\mathrm{Mean}(\cdot)$ or $\mathrm{Max}(\cdot)$ aggregator
    - $\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$



INPUT GRAPH



Message Aggregation: Issue

과거 본인의 대한 정보 손실 가능성

- **Issue:** Information from node v itself **could get lost**
  - Computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$
- **Solution:** Include $\mathbf{h}_v^{(l-1)}$ when computing $\mathbf{h}_v^{(l)}$
  - **(1) Message: compute message from node v itself**
    - Usually, a **different message computation** will be performed

$$\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)} \qquad \mathbf{m}_v^{(l)} = \mathbf{B}^{(l)}\mathbf{h}_v^{(l-1)}$$

  - **(2) Aggregation:** After aggregating from neighbors, we can **aggregate the message from node v itself**
    - Via concatenation **or** summation

Then aggregate from node itself

$$\mathbf{h}_v^{(l)} = \text{CONCAT}\left(\text{AGG}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right), \mathbf{m}_v^{(l)}\right)$$

First aggregate from neighbors

A Single GNN Layer

- **Putting things together:**
  - **(1) Message**: each node computes a message

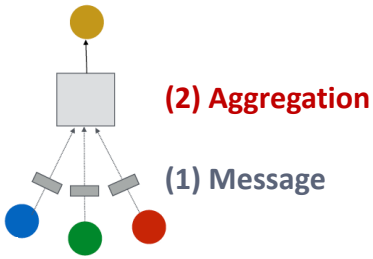
$$\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$$

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

$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}, \mathbf{m}_v^{(l)}\right)$$

Self message + concat

  - **Nonlinearity (activation):** Adds expressiveness
    - Often written as $\sigma(\cdot)$: $\mathrm{ReLU}(\cdot)$, $\mathrm{Sigmoid}(\cdot)$, ...
    - Can be added to **message or aggregation**



Classical GNN Layers: GCN (1)

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

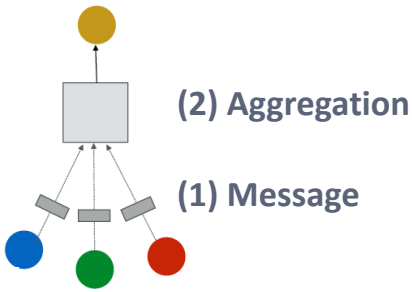
$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

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

Message

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

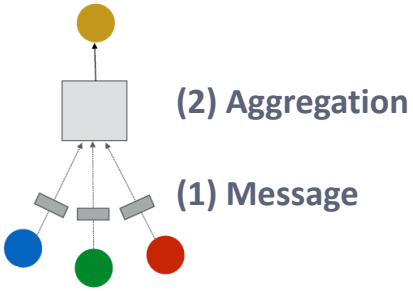
Aggregation



Classical GNN Layers: GCN (2)

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

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



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

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

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

In GCN graph is assumed to have self-edges that are included in the summation.

Classical GNN Layers: GraphSAGE

- **(2) GraphSAGE**

$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \cdot \text{CONCAT}\left(\mathbf{h}_v^{(l-1)}, \text{AGG}\left(\left\{\mathbf{h}_u^{(l-1)}, \forall u \in N(v)\right\}\right)\right)\right)$$

- **How to write this as Message + Aggregation?**
  - **Message** is computed within the $\text{AGG}(\cdot)$
  - **Two-stage aggregation**
    - **Stage 1:** Aggregate from node neighbors

$$\mathbf{h}_{N(v)}^{(l)} \leftarrow \text{AGG}\left(\left\{\mathbf{h}_u^{(l-1)}, \forall u \in N(v)\right\}\right)$$

    - **Stage 2:** Further aggregate over the node itself

$$\mathbf{h}_v^{(l)} \leftarrow \sigma\left(\mathbf{W}^{(l)} \cdot \text{CONCAT}(\mathbf{h}_v^{(l-1)}, \mathbf{h}_{N(v)}^{(l)})\right)$$

GraphSAGE Neighbor Aggregation

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

$$\text{AGG} = \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}$$

Aggregation

Message computation

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

MLP 씌우는 것.

$$\text{AGG} = \text{Mean}(\{\text{MLP}(\mathbf{h}_u^{(l-1)}), \forall u \in N(v)\})$$

Aggregation

Message computation

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

Sequence Model 가능!!

$$\text{AGG} = \text{LSTM}([\mathbf{h}_u^{(l-1)}, \forall u \in \pi(N(v))])$$

Aggregation

GraphSAGE: L2 Normalization

- **ℓ_2 Normalization:**
  - **Optional:** Apply ℓ_2 normalization to $\mathbf{h}_v^{(l)}$ at every layer
  - $\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\left\|\mathbf{h}_v^{(l)}\right\|_2} \; \forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ (ℓ_2-norm)
  - Without ℓ_2 normalization, the embedding vectors have different scales (ℓ_2-norm) for vectors
  - In some cases (not always), normalization of embedding results in performance improvement
  - After ℓ_2 normalization, all vectors will have the same ℓ_2-norm

Classical GNN Layers: GAT (1)

- **(3) Graph Attention Networks**

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

Attention weights ⇒ weight associated with every neighbor.

- **In GCN / GraphSAGE**
  - $\alpha_{vu} = \frac{1}{|N(v)|}$ is the **weighting factor (importance)** of node u's message to node v
  - $\Longrightarrow$ α_{vu} is defined **explicitly** based on the structural properties of the graph (node degree)
  - $\Longrightarrow$ All neighbors $u \in N(v)$ are equally important to node v

implicitly defined. / or learned.

Classical GNN Layers: GAT (2)

We want to learn message importances.

- **(3) Graph Attention Networks**

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

Attention weights

Not all node's neighbors are equally important

- **Attention** is inspired by cognitive attention.
- The **attention** α_{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.

[Velickovic et al., ICLR 2018; Vaswani et al., NIPS 2017]

Graph Attention Networks

Can we do better than simple neighborhood aggregation?

Can we let weighting factors α_{vu} to be learned?

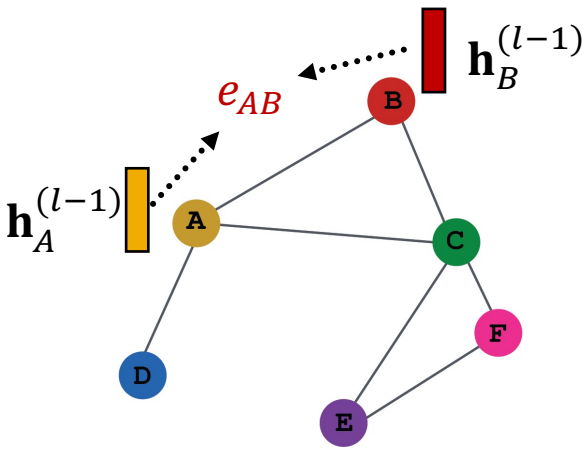
- **Goal:** Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea:** Compute embedding $\boldsymbol{h}_v^{(l)}$ of each node in the graph following an **attention strategy:**
  - Nodes attend over their neighborhoods' message
  - Implicitly specifying different weights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

Attention Mechanism (1)

- Let α_{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:

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

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



$$e_{AB} = a(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)})$$

Attention Mechanism (2)

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

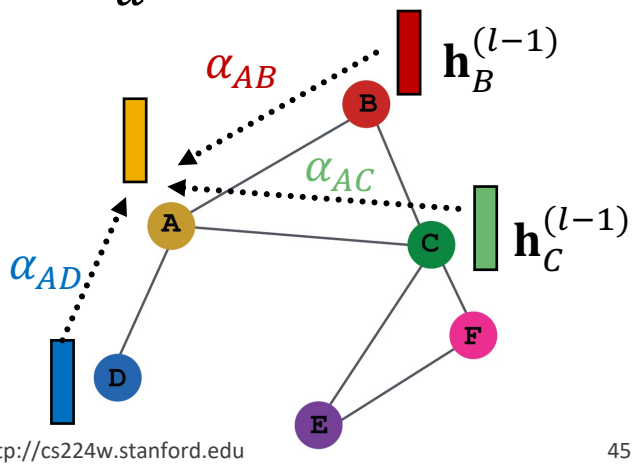
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the **final attention weight** α_{vu}

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

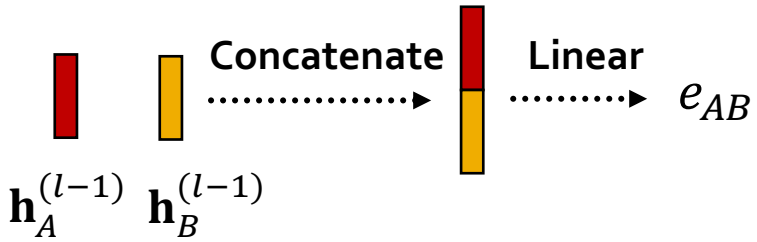
Weighted sum using α_{AB}, α_{AC}, α_{AD}:

$$\mathbf{h}_A^{(l)} = \sigma\left(\alpha_{AB}\mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)} + \alpha_{AC}\mathbf{W}^{(l)}\mathbf{h}_C^{(l-1)} + \alpha_{AD}\mathbf{W}^{(l)}\mathbf{h}_D^{(l-1)}\right)$$



Attention Mechanism (3)

- **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)



$$e_{AB} = a\left(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}\right)$$
$$= \text{Linear}\left(\text{Concat}\left(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}\right)\right)$$

  - Parameters of a are trained jointly:
    - Learn the parameters together with weight matrices (i.e., other parameter of the neural net $\mathbf{W}^{(l)}$) in an end-to-end fashion

Attention Mechanism (4)

- **Multi-head attention:** Stabilizes the learning process of attention mechanism
  - **Create multiple attention scores** (each replica with a different set of parameters):

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

$$\mathbf{h}_v^{(l)}[2] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^{2} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

$$\mathbf{h}_v^{(l)}[3] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^{3} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

  - **Outputs are aggregated:**
    - By concatenation or summation
    - $\mathbf{h}_v^{(l)} = \mathrm{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$

Benefits of Attention Mechanism

- **Key benefit:** Allows for (implicitly) specifying **different importance values (α_{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 *edge-wise* mechanism
  - It does not depend on the global graph structure

Stanford CS224W: GNN Layers in Practice

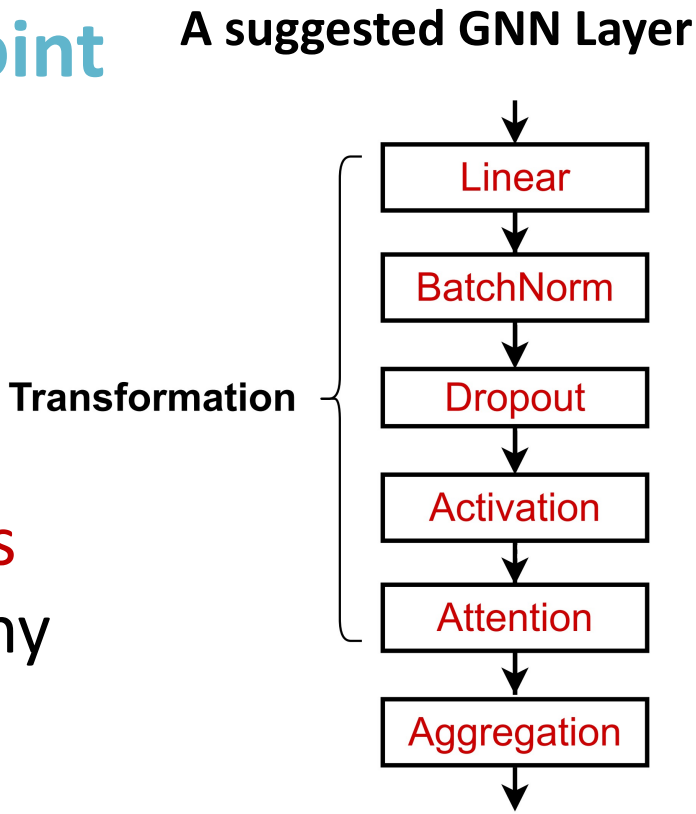
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

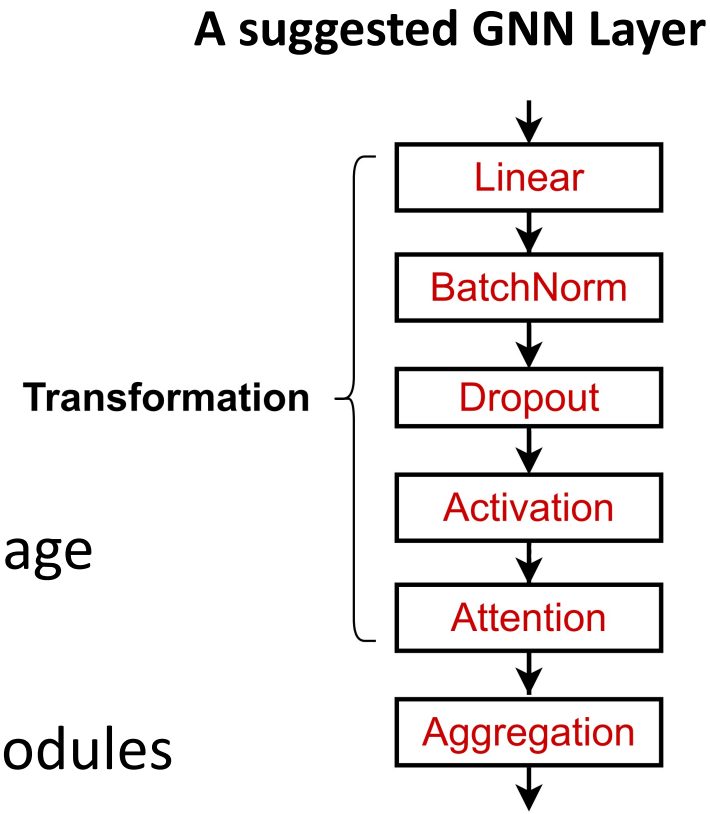
GNN Layer in Practice

- **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



GNN Layer in Practice

- **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



S. Loffe, C.Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

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: $\mathbf{X} \in \mathbb{R}^{N \times D}$
N node embeddings

Trainable Parameters:
$\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^{D}$

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

Step 1:
Compute the mean and variance over N embeddings

$$\boldsymbol{\mu}_j = \frac{1}{N}\sum_{i=1}^{N} \mathbf{X}_{i,j}$$

$$\boldsymbol{\sigma}_j^2 = \frac{1}{N}\sum_{i=1}^{N} \left(\mathbf{X}_{i,j} - \boldsymbol{\mu}_j\right)^2$$

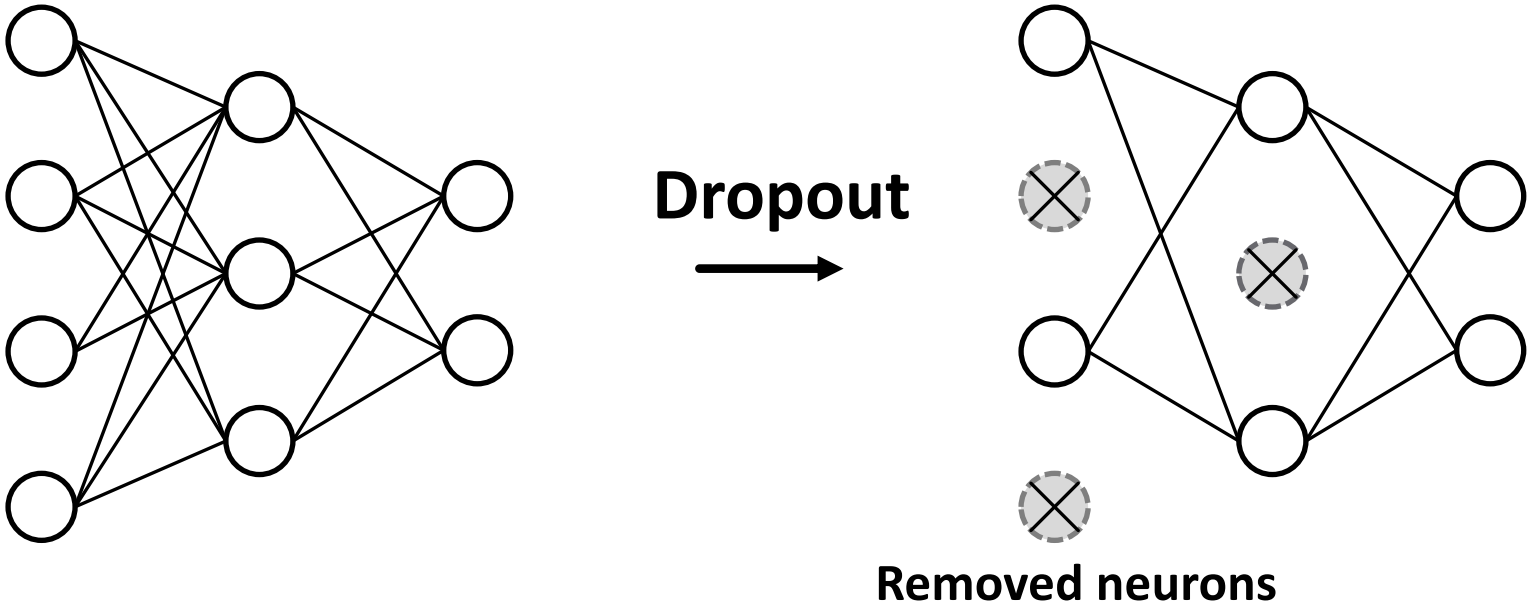
Step 2:
Normalize the feature using computed mean and variance

$$\widehat{\mathbf{X}}_{i,j} = \frac{\mathbf{X}_{i,j} - \boldsymbol{\mu}_j}{\sqrt{\boldsymbol{\sigma}_j^2 + \epsilon}}$$

$$\mathbf{Y}_{i,j} = \boldsymbol{\gamma}_j \widehat{\mathbf{X}}_{i,j} + \boldsymbol{\beta}_j$$

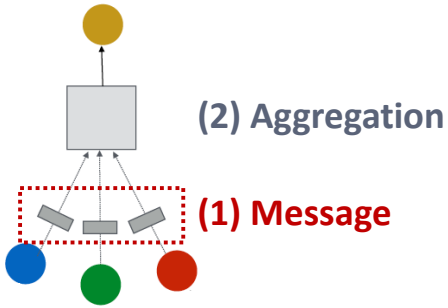
Dropout

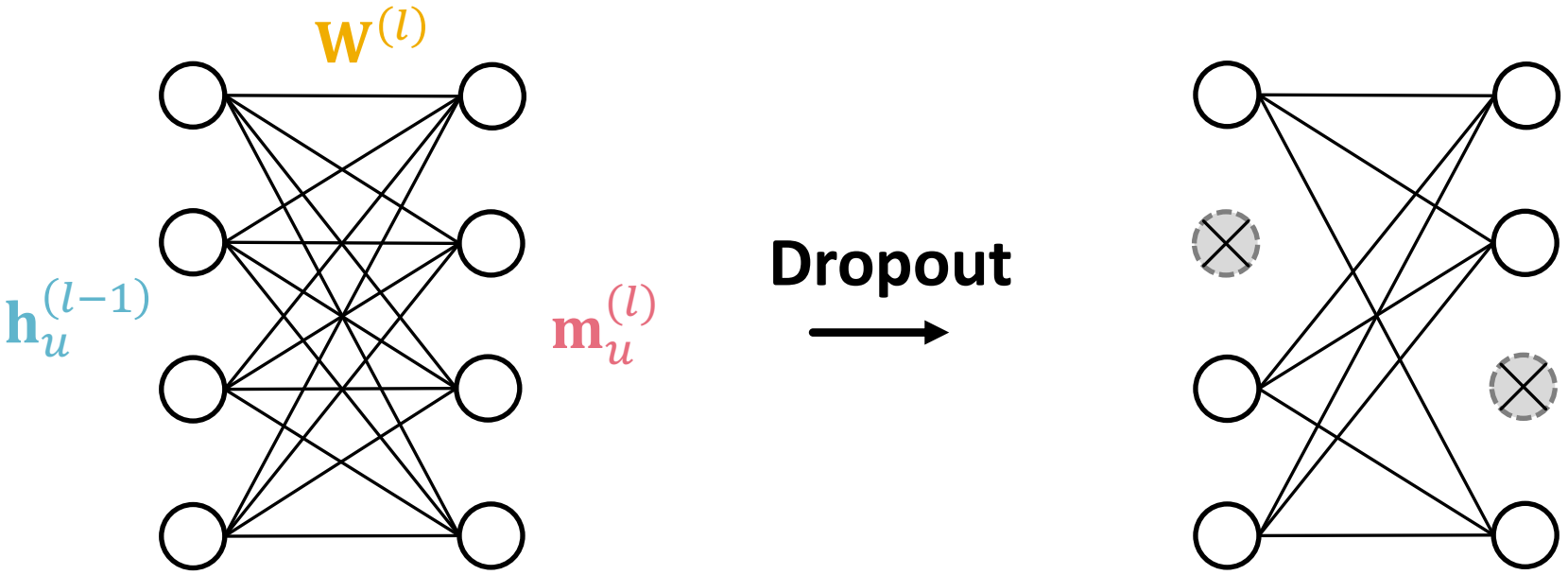
- **Goal**: Regularize a neural net to prevent overfitting.
- **Idea**:
  - **During training**: with some probability p, randomly set neurons to zero (turn off)
  - **During testing:** Use all the neurons for computation



Dropout for GNNs

- In GNN, Dropout is applied to **the linear layer in the message function**
  - **A simple message function with linear layer:** $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$





Visualization of a linear layer

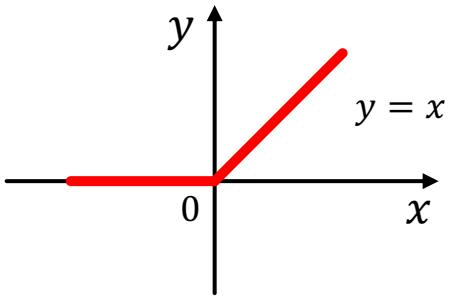
Activation (Non-linearity)

Apply activation to i-th dimension of embedding $\mathbf{x}$

- **Rectified linear unit (ReLU)**

  $$\text{ReLU}(\mathbf{x}_i) = \max(\mathbf{x}_i, 0)$$

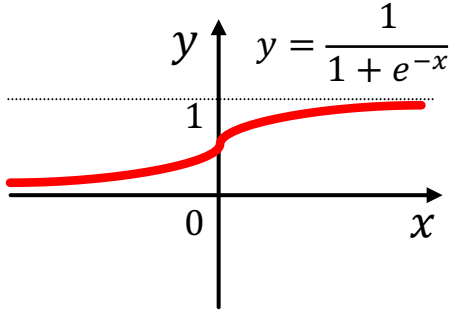
  - Most commonly used



- **Sigmoid**

  $$\sigma(\mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{x}_i}}$$

  - Used only when you want to restrict the range of your embeddings

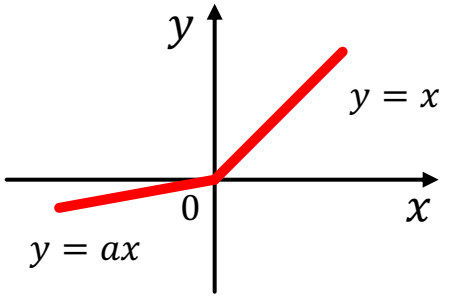


- **Parametric ReLU**

  $$\text{PReLU}(\mathbf{x}_i) = \max(\mathbf{x}_i, 0) + a_i \min(\mathbf{x}_i, 0)$$

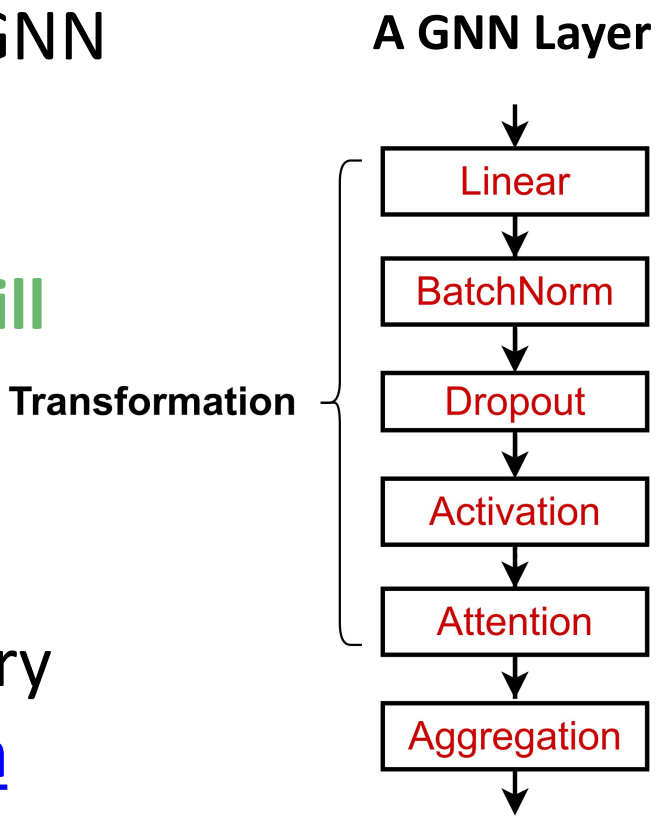
  a_i is a trainable parameter

  - Empirically performs better than ReLU



GNN Layer in Practice

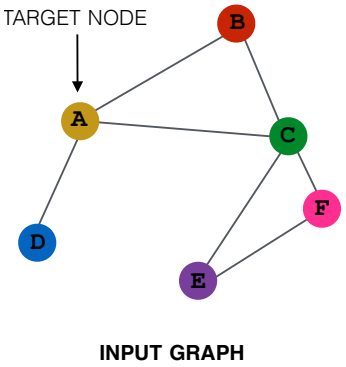
- **Summary:** Modern deep learning modules can be included into a GNN layer for better performance
- **Designing novel GNN layers is still an active research frontier!**
- **Suggested resources:** You can explore diverse GNN designs or try out your own ideas in **GraphGym**



Stanford CS224W: Stacking Layers of a GNN

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

Stacking GNN Layers

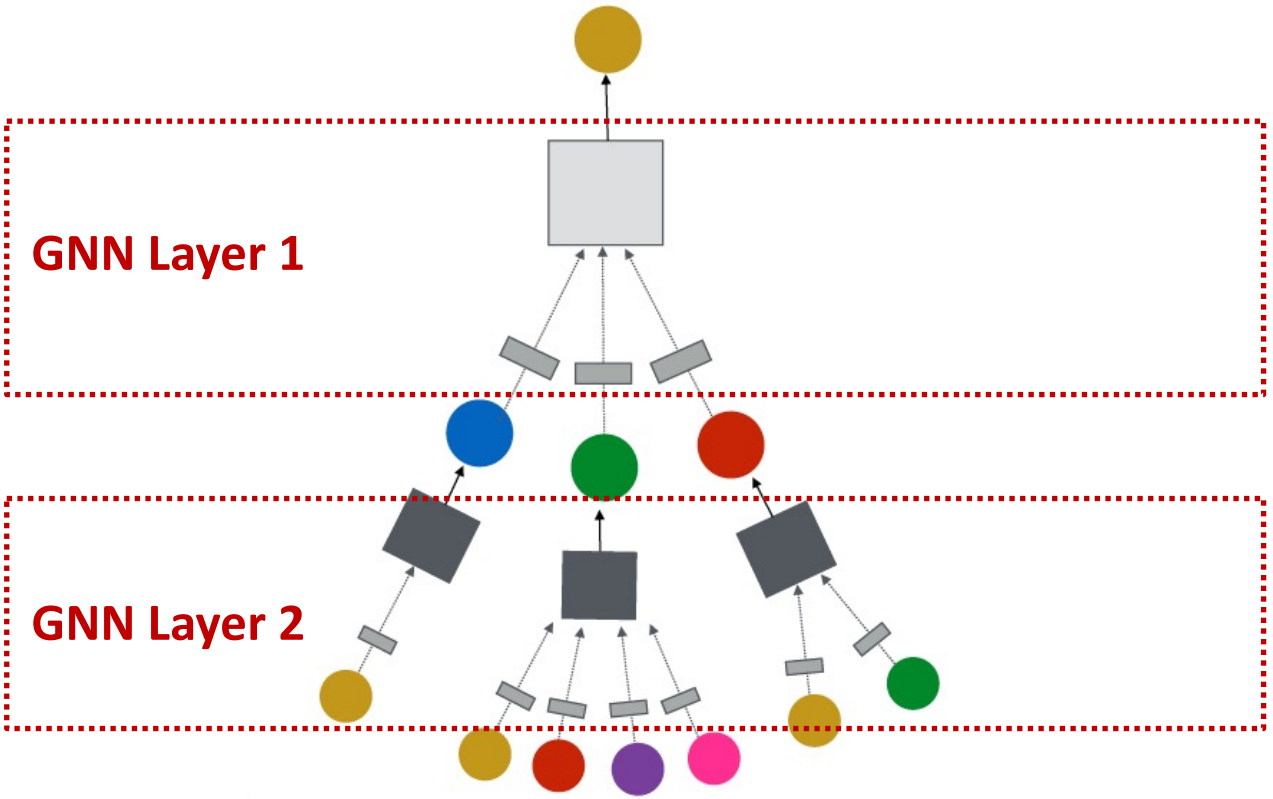


INPUT GRAPH

How to connect GNN layers into a GNN?

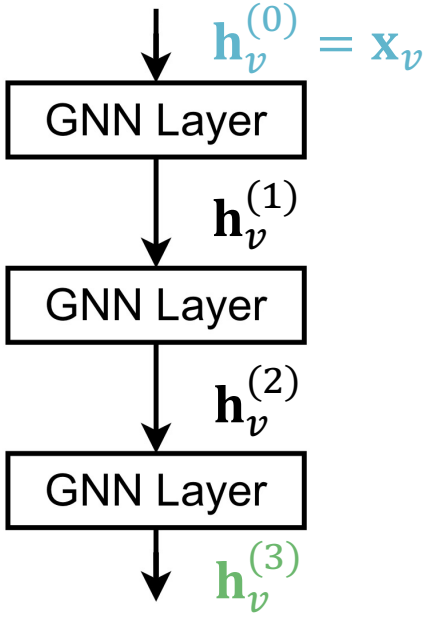
- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity



Stacking GNN Layers

- **How to construct a Graph Neural Network?**
  - **The standard way:** Stack GNN layers sequentially
  - **Input:** Initial raw node feature $\mathbf{x}_v$
  - **Output:** Node embeddings $\mathbf{h}_v^{(L)}$ after L GNN layers



The Over-smoothing Problem

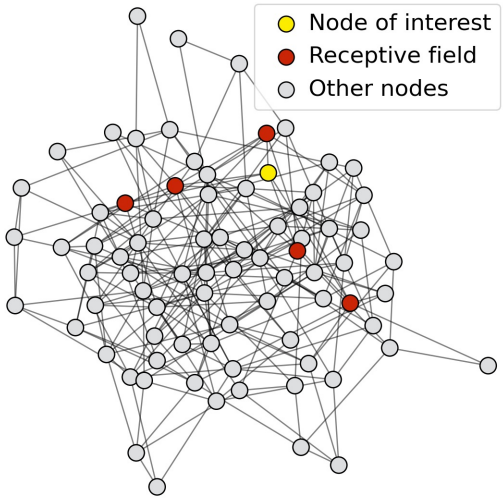
Collect info from many hops

- **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?**

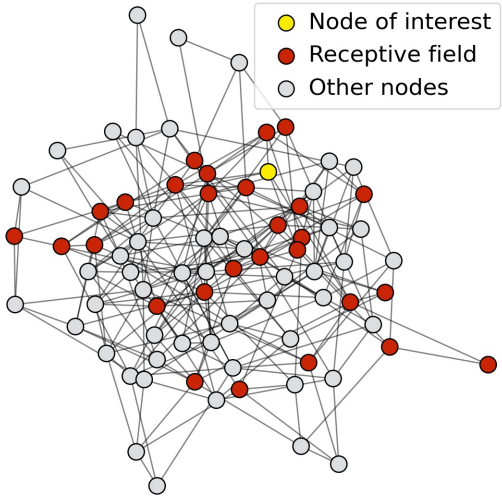
Receptive Field of a GNN

- **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**

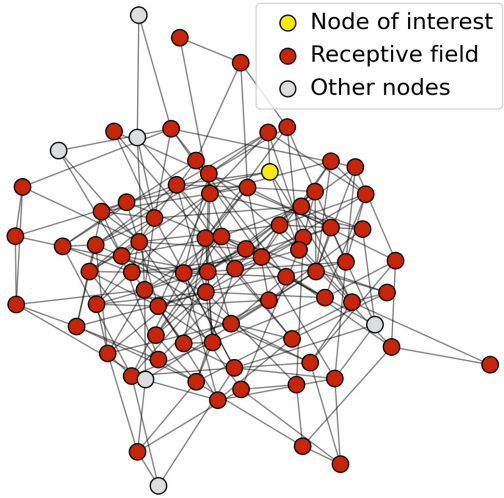
Receptive field for 1-layer GNN



Receptive field for 2-layer GNN



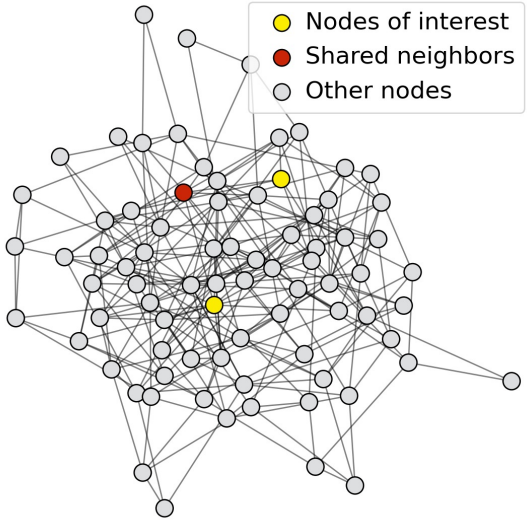
Receptive field for 3-layer GNN



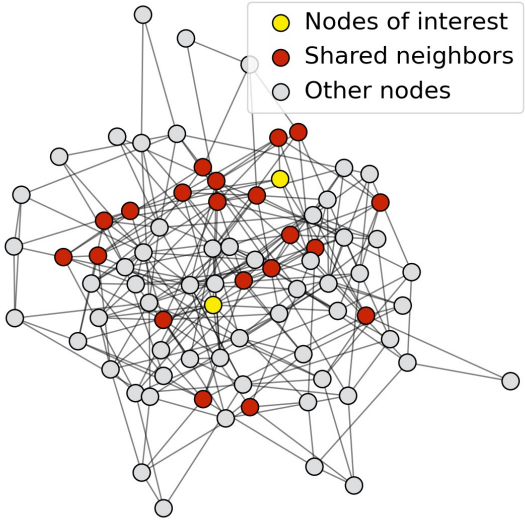
Receptive Field of a GNN

- **Receptive field overlap** for two nodes
  - **The shared neighbors quickly grows** when we increase the number of hops (num of GNN layers)

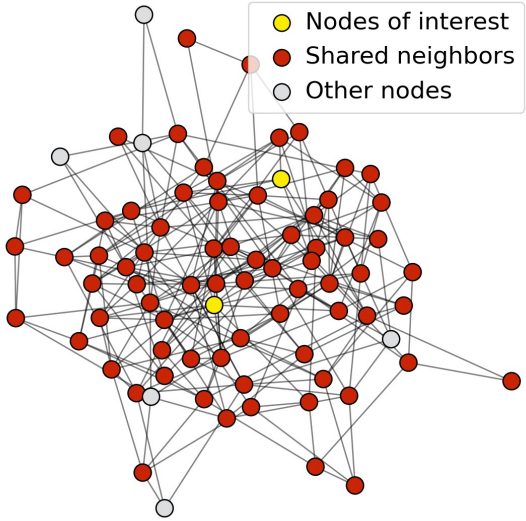
1-hop neighbor overlap
Only 1 node



2-hop neighbor overlap
About 20 nodes



3-hop neighbor overlap
Almost all the nodes!



Receptive Field & Over-smoothing

- **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?

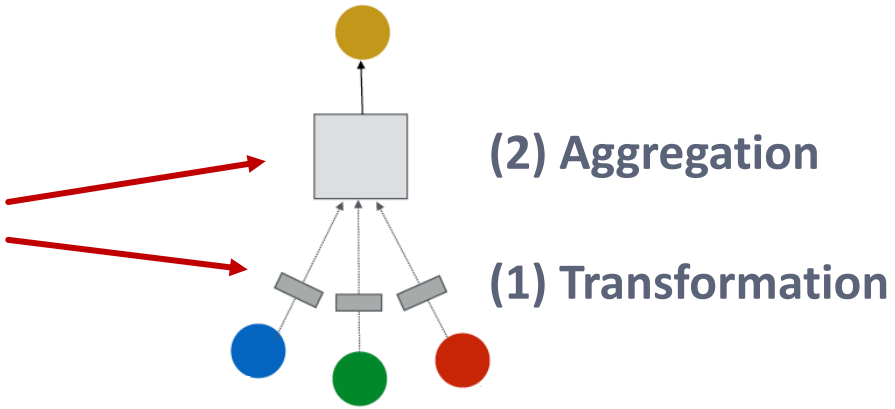
Design GNN Layer Connectivity

- **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?

Expressive Power for Shallow GNNs

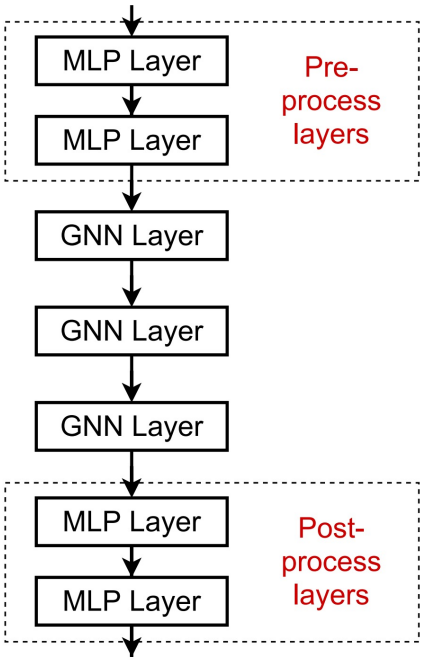
- **How to make a shallow GNN more expressive?**
- **Solution 1:** Increase the expressive power **within each GNN layer**
  - In our previous examples, each transformation or aggregation function only include one linear layer
  - We can **make aggregation / transformation become a deep neural network**!

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



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**



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

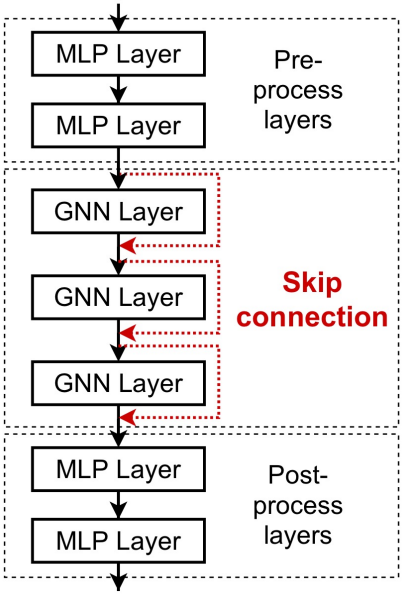
Post-processing layers: Important when reasoning / transformation over node embeddings are needed
E.g., graph classification, knowledge graphs

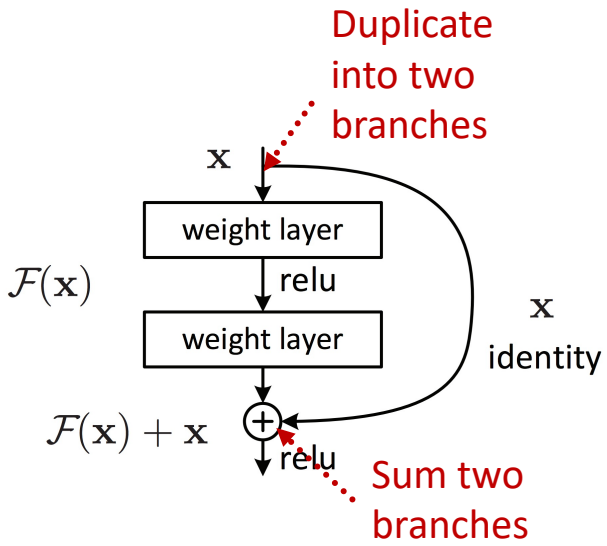
In practice, adding these layers works great!

He et al. Deep Residual Learning for Image Recognition, CVPR 2015

Design GNN Layer Connectivity

- **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 GNN**





Idea of skip connections:

Before adding shortcuts:

$$\boldsymbol{F}(\mathbf{x})$$

After adding shortcuts:

$$\boldsymbol{F}(\mathbf{x}) + \mathbf{x}$$

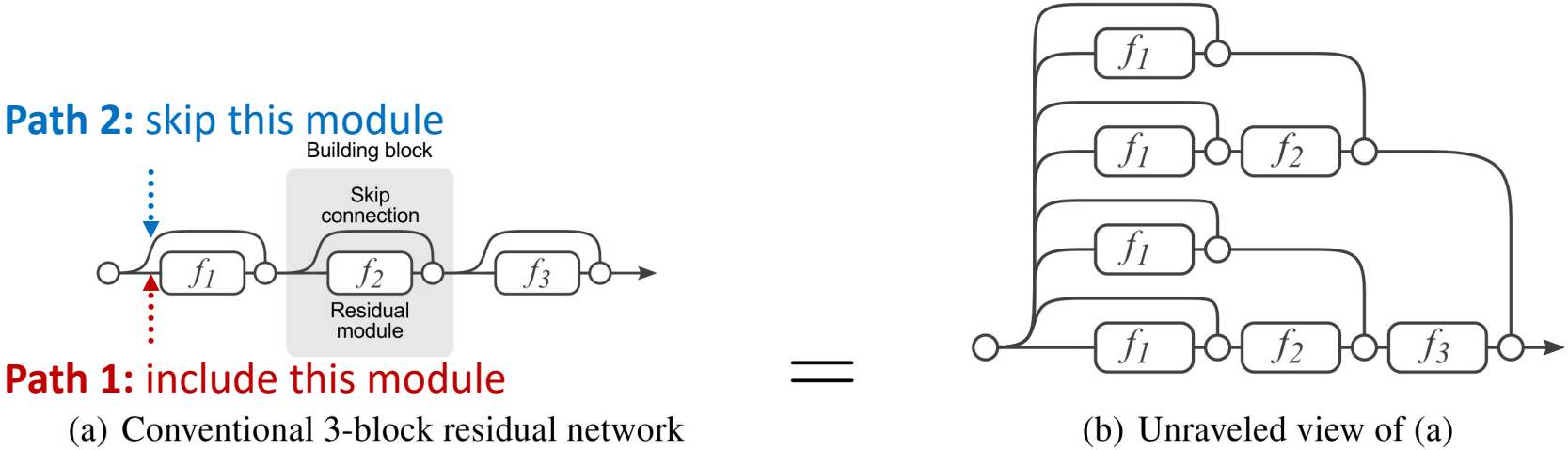
Idea of Skip Connections

- **Why do skip connections work?**
  - **Intuition:** Skip connections create **a mixture of models**
  - N skip connections → 2^N possible paths
  - Each path could have up to N modules
  - We automatically get **a mixture of shallow GNNs and deep GNNs**

↳ weighted combination of a prev. layer & current layer message.

All the possible paths:

$$2 * 2 * 2 = 2^3 = 8$$



(a) Conventional 3-block residual network

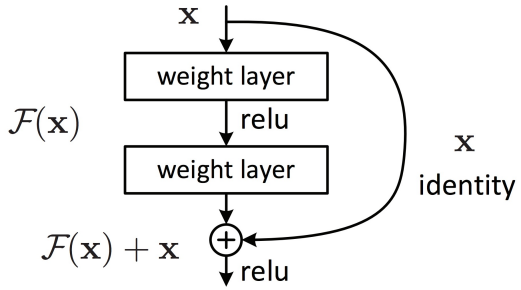
(b) Unraveled view of (a)

Veit et al. Residual Networks Behave Like Ensembles of Relatively Shallow Networks, ArXiv 2016

Example: GCN with Skip Connections

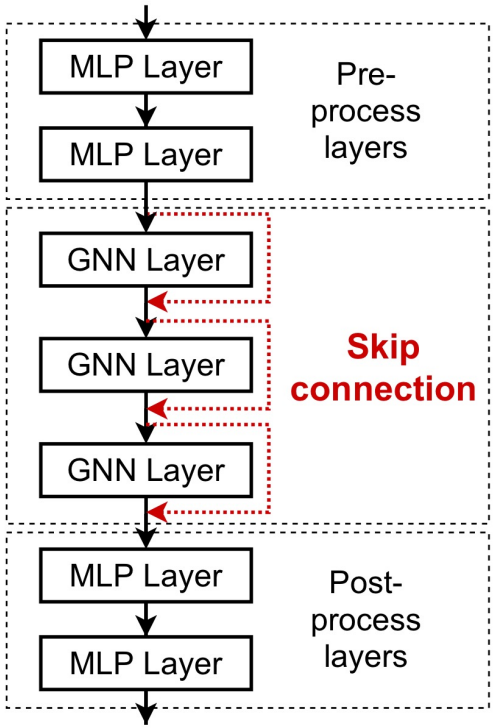
- **A standard GCN layer**
- $\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$

This is our $F(\mathbf{x})$



- **A GCN layer with skip connection**
- $\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)}\right)$

$F(\mathbf{x})$ + $\mathbf{x}$



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