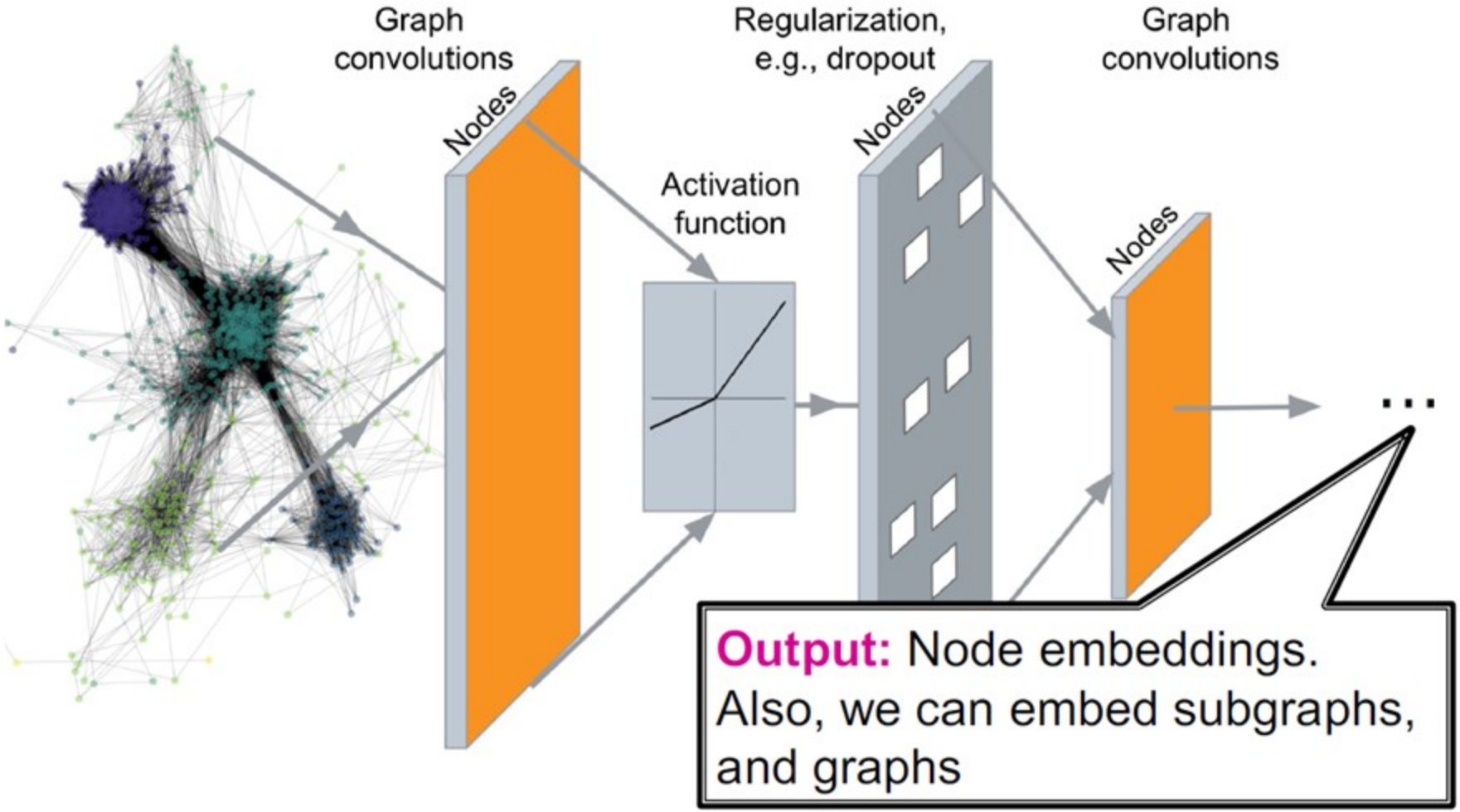
Graph Neural Network

Graph Neural Networks

- Introduction
- Deep Learning for graphs
- Graph Convolutional Networks (GCN)

Introduction



Deep Learning for graphs

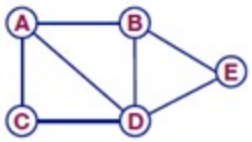
- Content
  - **Local network neighborhoods:**
    - Describe aggregation strategies
    - Define computation graphs
  - **Stacking multiple layers:**
    - Describe the model, parameters, training
    - How to fit the model?
    - Simple example for unsupervised and supervised training

Deep Learning for graphs

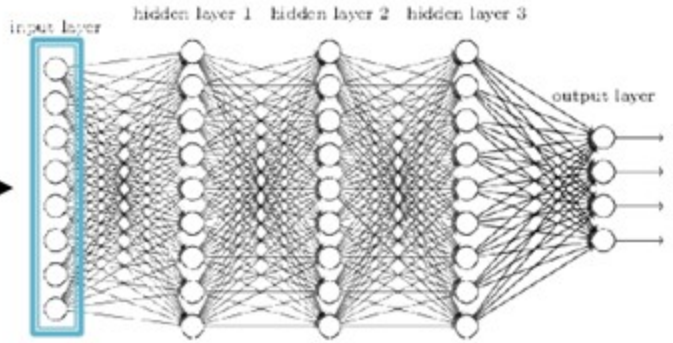
- Setup
  - **Assume we have a graph G:**
    - V is the **vertex set**
    - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
    - $\boldsymbol{X} \in \mathbb{R}^{m \times |V|}$ is a matrix of **node features**
    - v: a node in V; $N(v)$: the set of neighbors of v.
    - **Node features:**
      - Social networks: User profile, User image
      - Biological networks: Gene expression profiles, gene functional information
      - When there is no node feature in the graph dataset:
        - Indicator vectors (one-hot encoding of a node)
        - Vector of constant 1: [1, 1, ..., 1]

Deep Learning for graphs

- Naïve Approach
  - Join adjacency matrix and features
  - Feed them into a deep neural net:



	A	B	C	D	E	Feat	
A	0	1	1	1	0	1	0
B	1	0	0	1	1	0	0
C	1	0	0	1	0	0	1
D	1	1	1	0	1	1	1
E	0	1	0	1	0	1	0



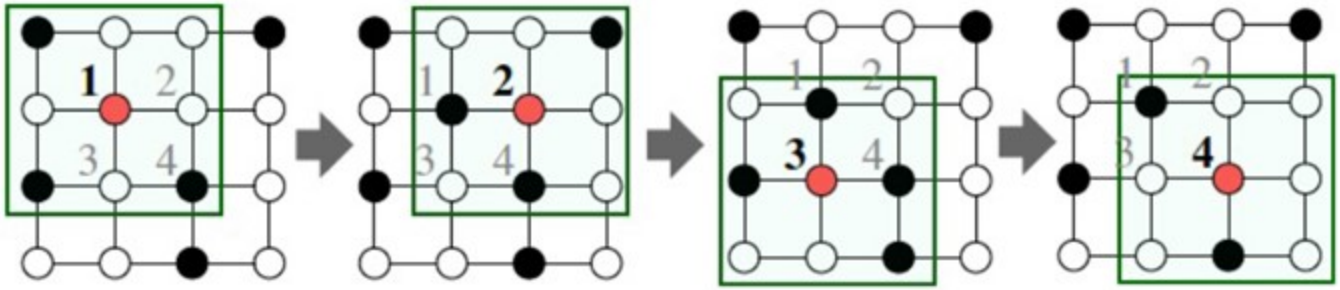
?

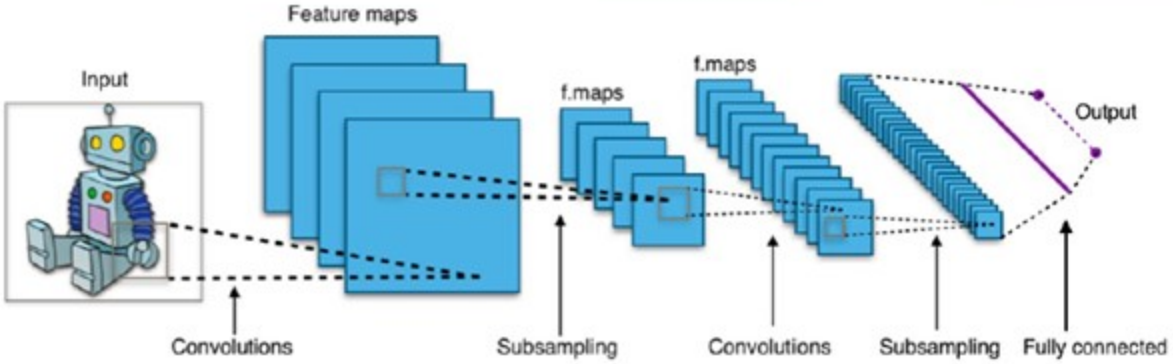
  - Issues with this idea:
    - $O(|V|)$ parameters
    - Not applicable to graphs of different sizes
    - Sensitive to node ordering

Deep Learning for graphs

- Convolutional Networks

  CNN on an image:





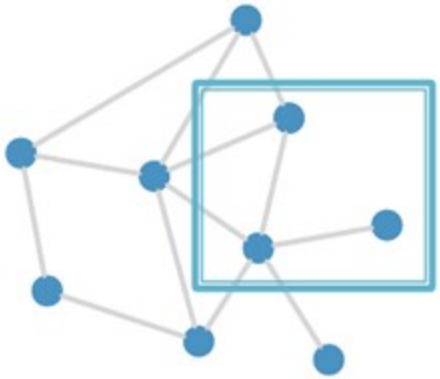
Goal is to generalize convolutions beyond simple lattices

Leverage node features/attributes (e.g., text, images)

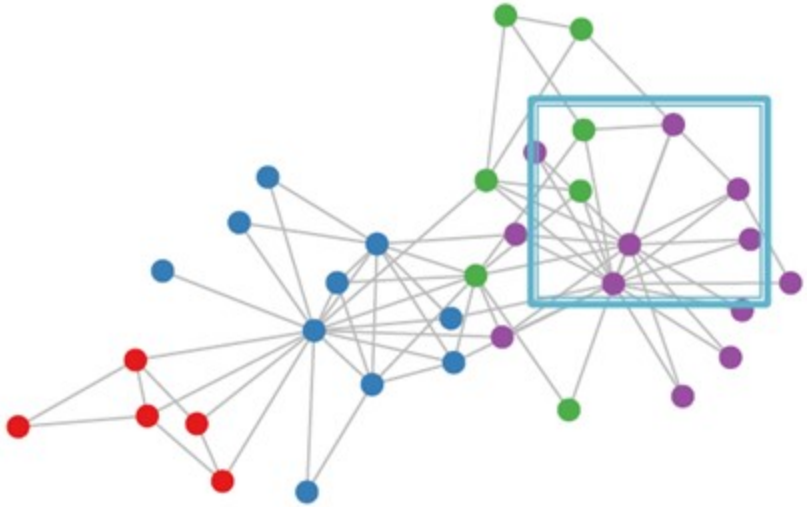
Deep Learning for graphs

- Real-World Graphs

  But our graphs look like this:



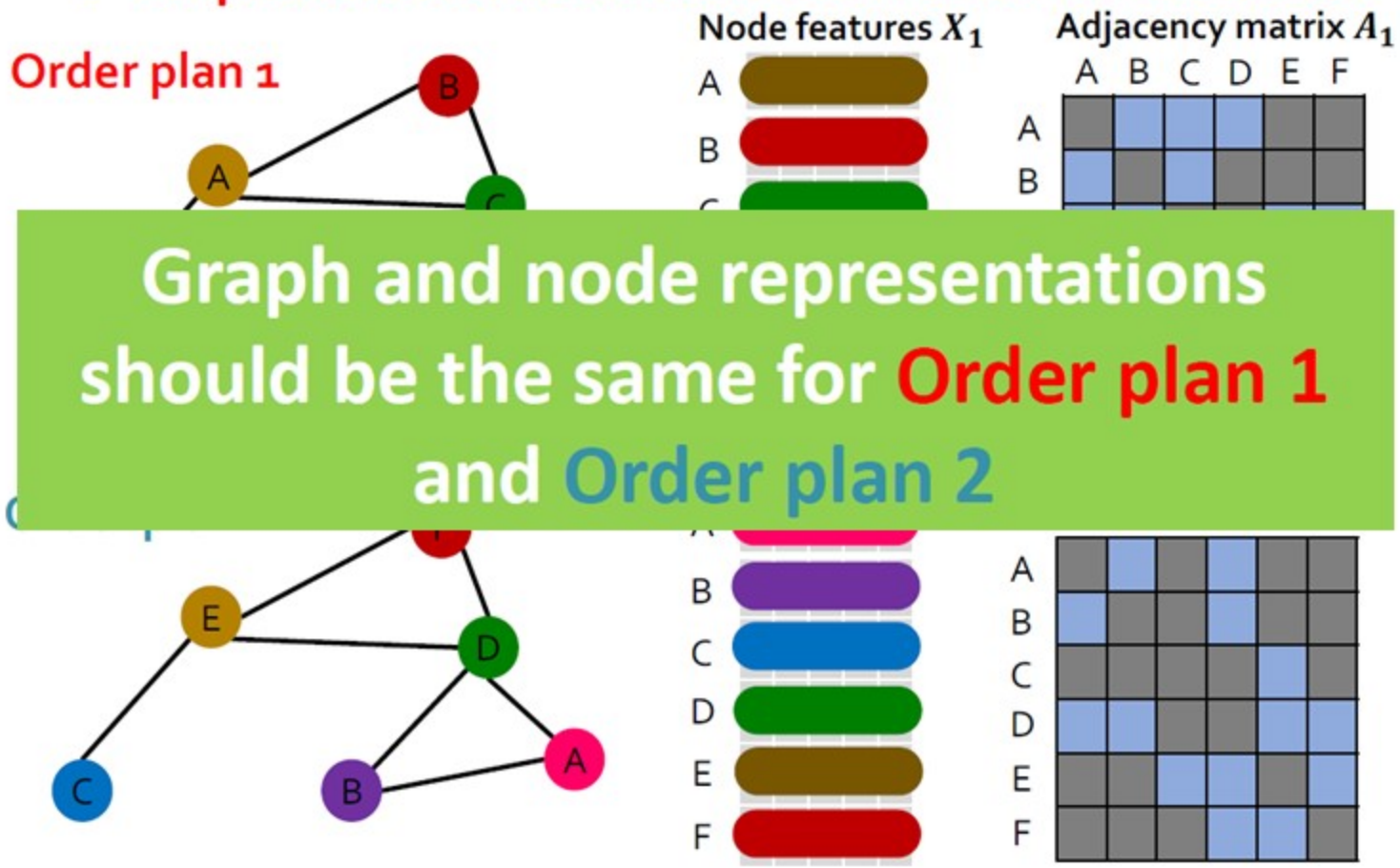
or this:



- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

Deep Learning for graphs

- Permutation Invariance
  - Graph does not have a canonical order of the nodes!



Deep Learning for graphs

- Permutation Invariance

  What does it mean by "graph representation is same for two order plans"?

  - Consider we learn a function f that maps a graph $G = (\boldsymbol{A}, \boldsymbol{X})$ to a vector $\mathbb{R}^d$.

    $\boldsymbol{A}$ is the adjacency matrix
    $\boldsymbol{X}$ is the node feature matrix

  - Then, if $f(\boldsymbol{A}_i, \boldsymbol{X}_i) = f(\boldsymbol{A}_j, \boldsymbol{X}_j)$ for any order plan i and j, we formally say f is a **permutation invariant function**.

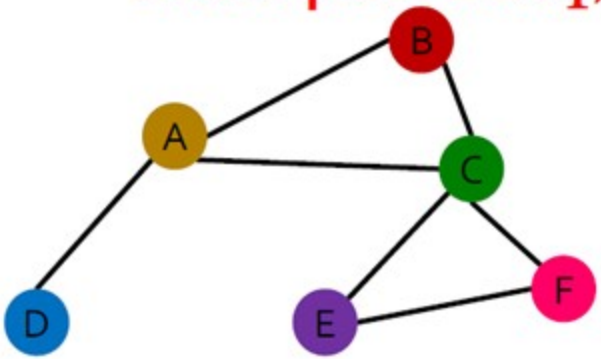
For a graph with m nodes, there are $m!$ different order plans.

Deep Learning for graphs

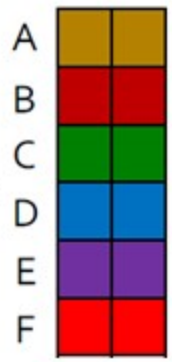
- Permutation Equivariance

  Similarly for node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

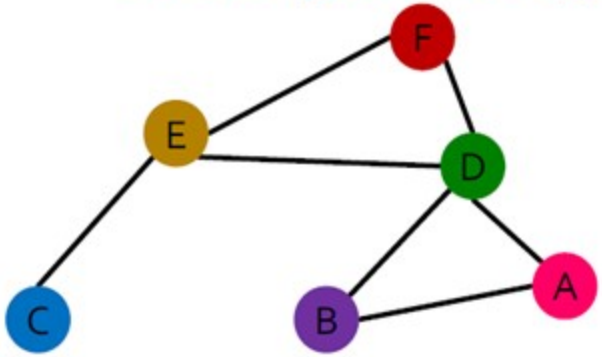
Order plan 1: A_1, X_1



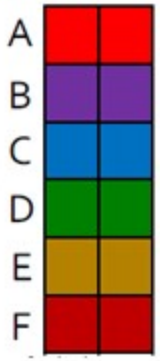
$$f(A_1, X_1) =$$



Order plan 2: A_2, X_2



$$f(A_2, X_2) =$$

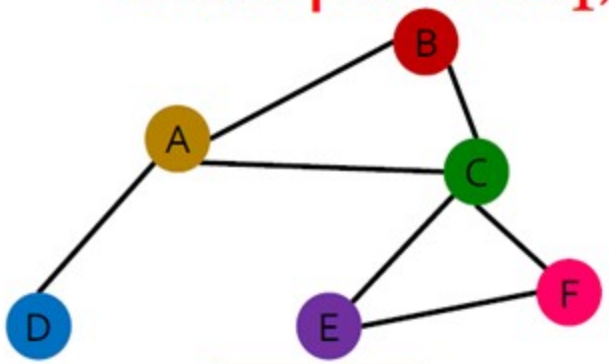


Deep Learning for graphs

- Permutation Equivariance

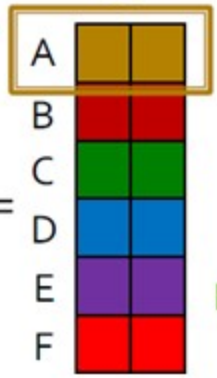
Similarly for node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: A_1, X_1

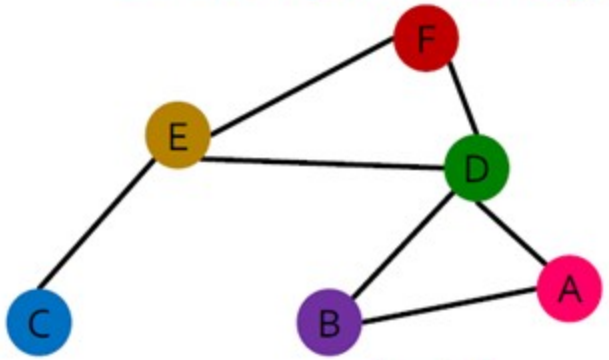


Representation vector of the brown node A

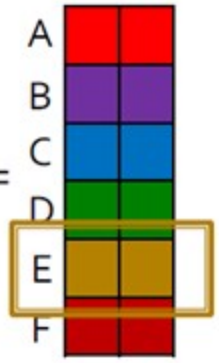
$$f(A_1, X_1) =$$



Order plan 2: A_2, X_2



$$f(A_2, X_2) =$$



Representation vector of the brown node E

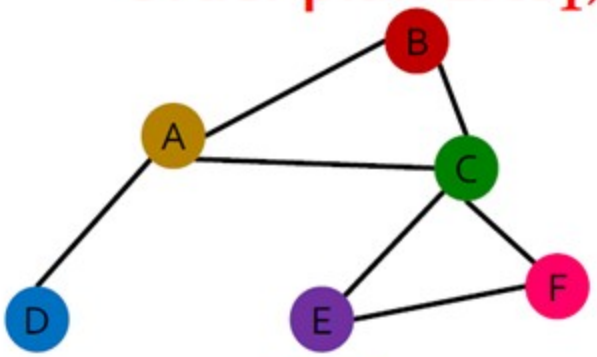
For two order plans, the vector of node at the same position is the same!

Deep Learning for graphs

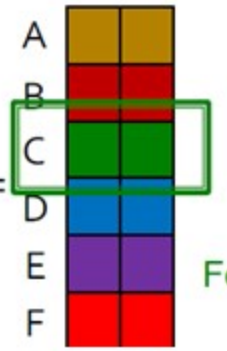
- Permutation Equivariance

Similarly for node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: A_1, X_1

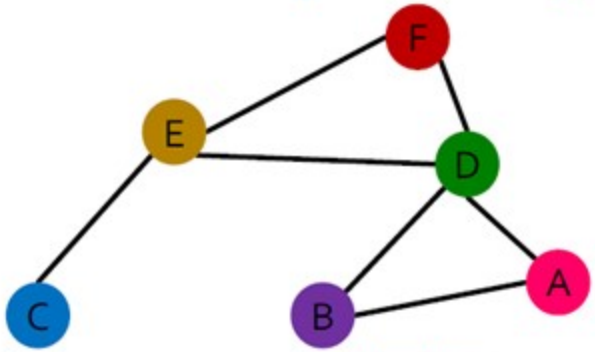


$f(A_1, X_1) =$

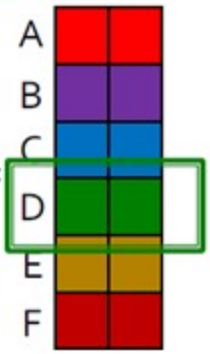


Representation vector of the brown node C

Order plan 2: A_2, X_2



$f(A_2, X_2) =$



Representation vector of the brown node D

For two order plans, the vector of node at the same position is the same!

Deep Learning for graphs

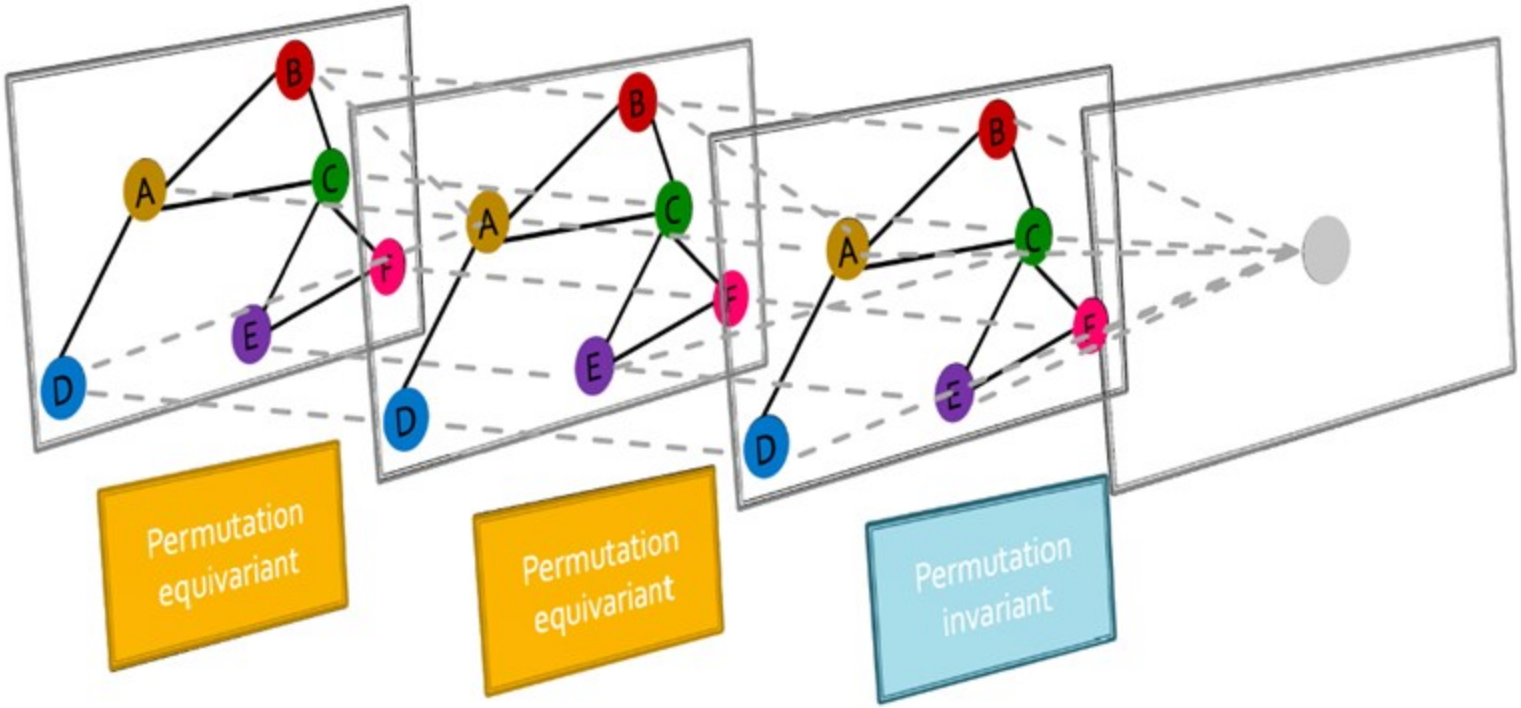
- Permutation Equivariance

  For node representation

  - Consider we learn a function f that maps a graph $G = (\boldsymbol{A}, \boldsymbol{X})$ to a matrix $\mathbb{R}^{m \times d}$
    - graph has m nodes, each row is the embedding of a node.
  - Similarly, if this property holds for any pair of order plan i and j, we say f is a **permutation equivariant function**.

Deep Learning for graphs

- Graph Neural Networks
  - **Graph neural networks consist of multiple permutation equivariant / invariant functions.**



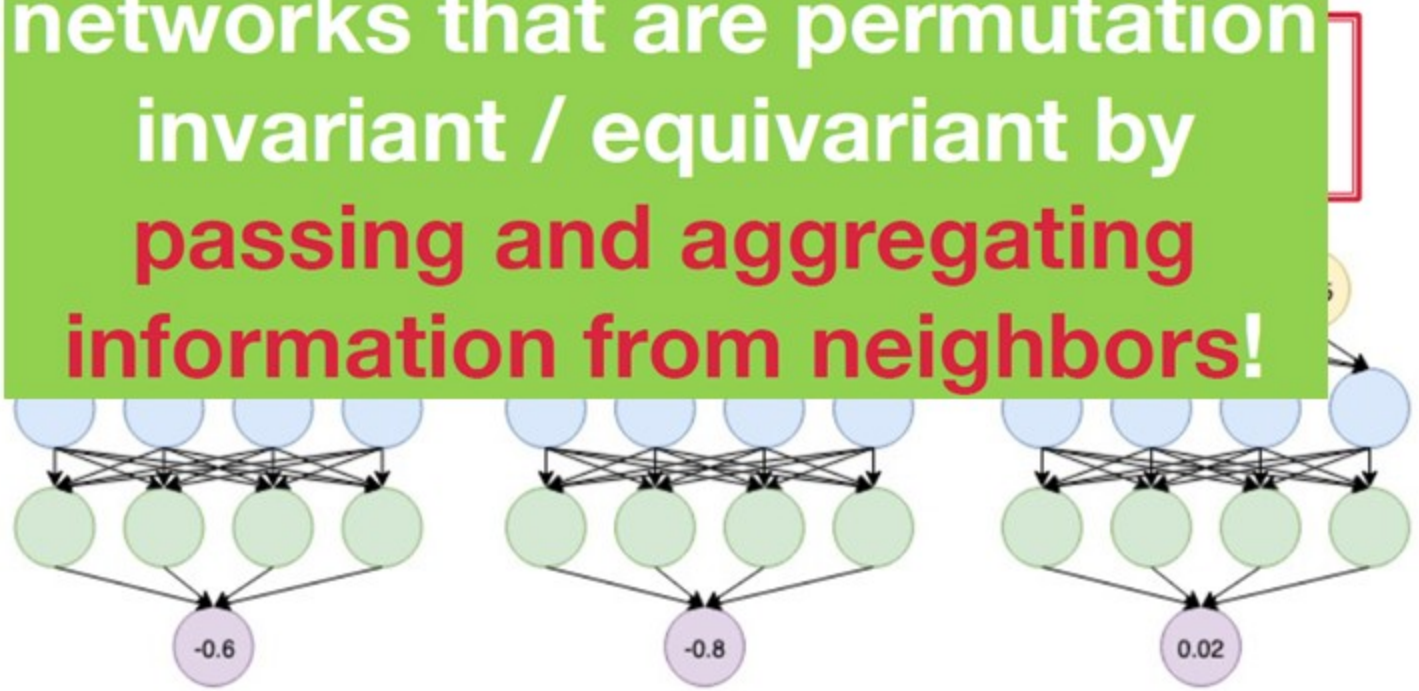
Deep Learning for graphs

- Graph Neural Networks

  Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

  - **N**

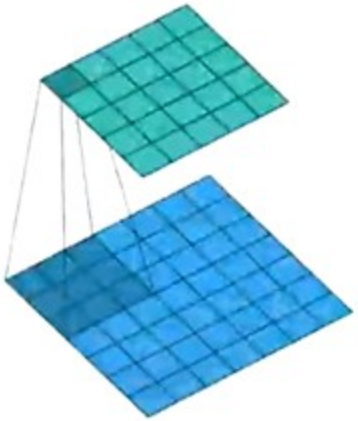
Next: Design graph neural networks that are permutation invariant / equivariant by passing and aggregating information from neighbors!



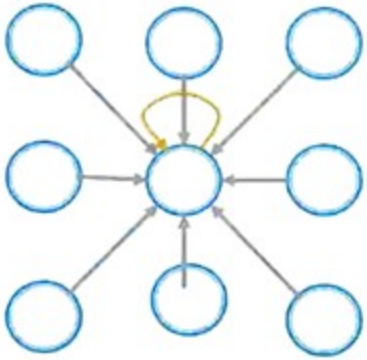
Graph Convolutional Networks

- From Images to Graphs

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



Image



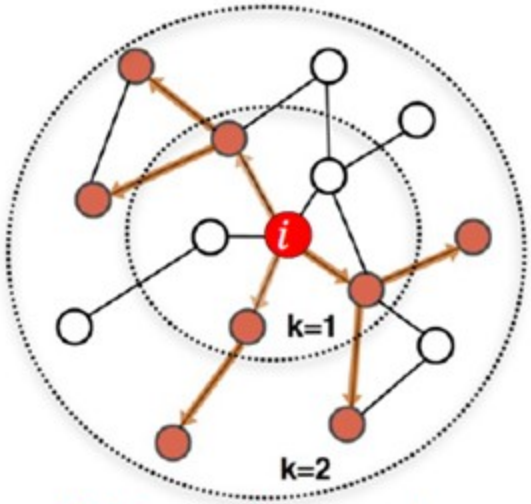
Graph

Idea: transform information at the neighbors and combine it:

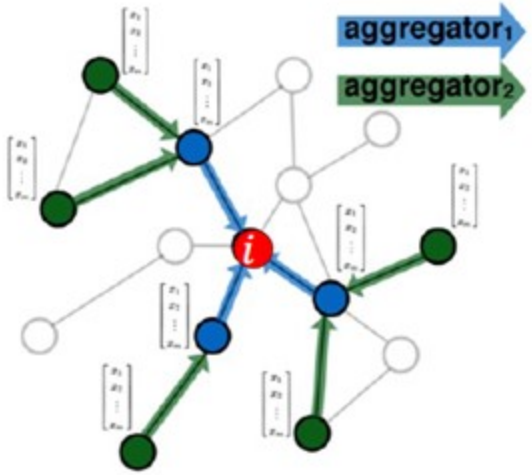
- Transform "messages" h_i from neighbors: $W_i\ h_i$
- Add them up: $\sum_i W_i\ h_i$

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

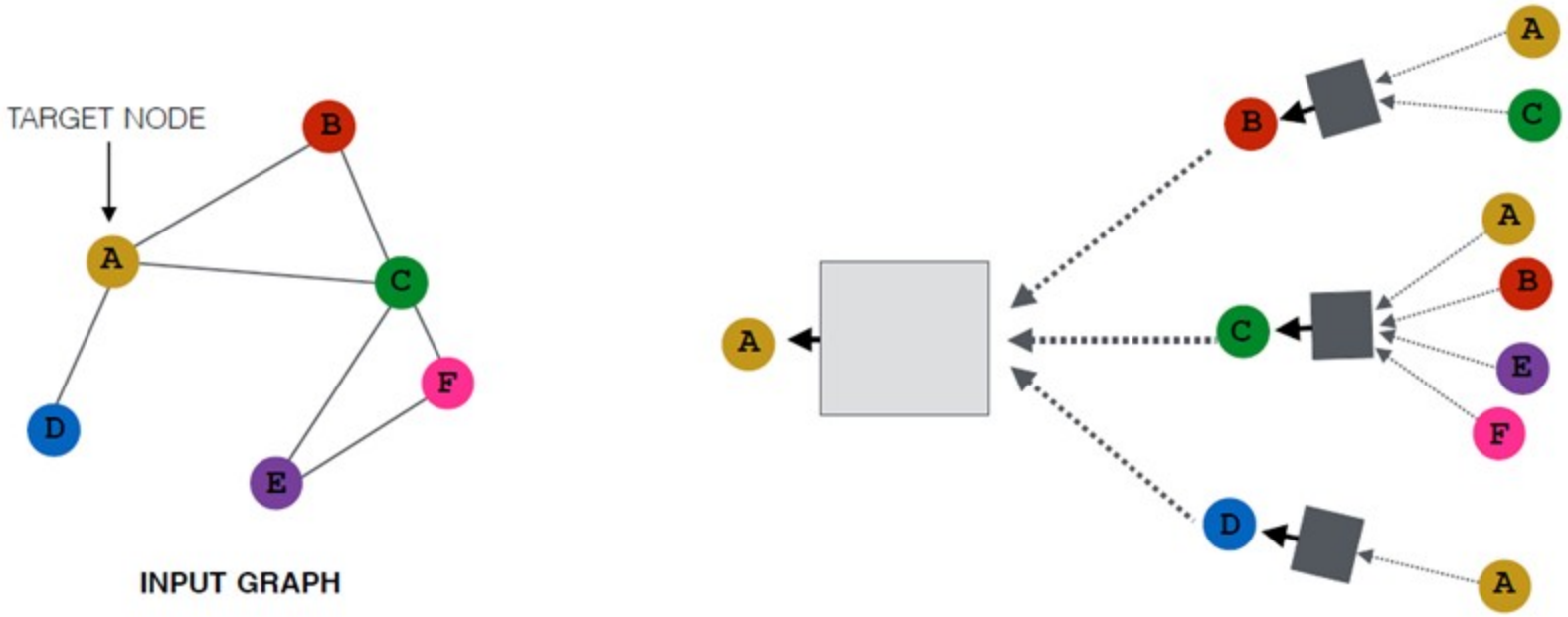


Propagate and transform information

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

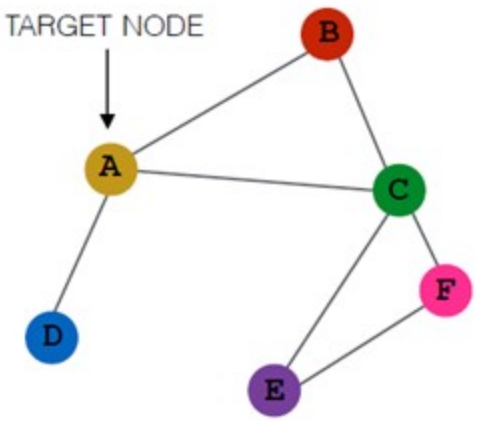
Graph Convolutional Networks

- Idea : Aggregate Neighbors
  - **Key idea:** Generate node embeddings based on **local network neighborhoods**

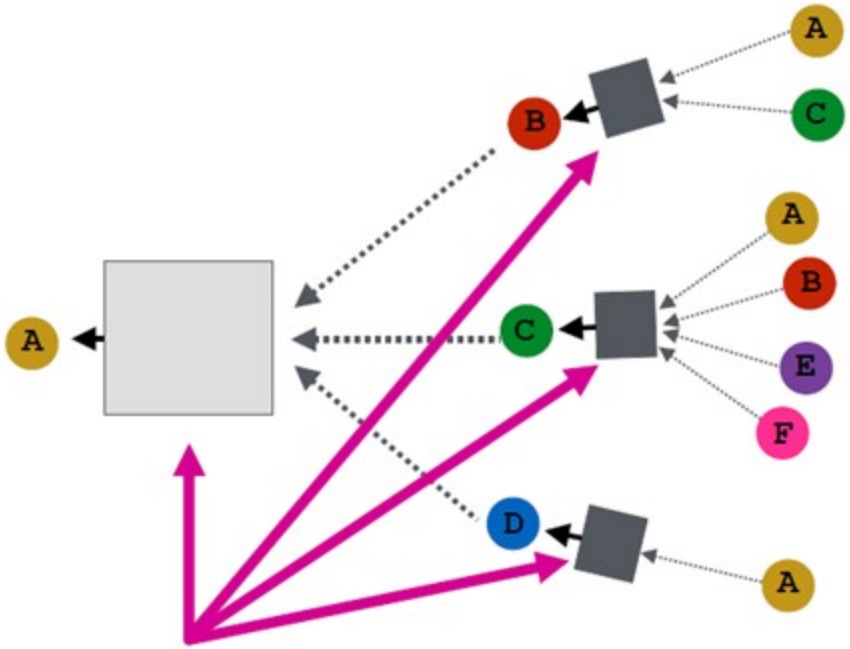


Graph Convolutional Networks

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



INPUT GRAPH

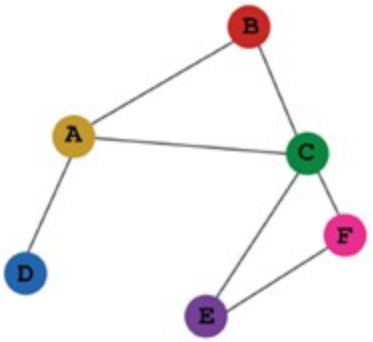


Neural networks

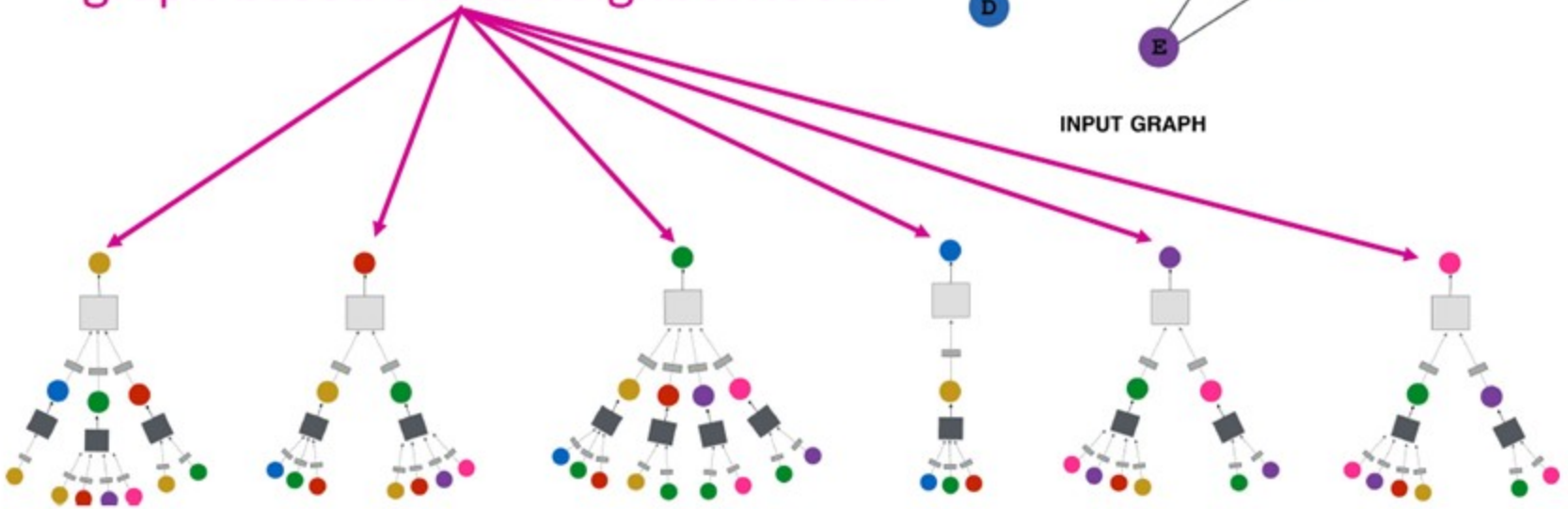
Graph Convolutional Networks

- Idea : Aggregate Neighbors
  - **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!

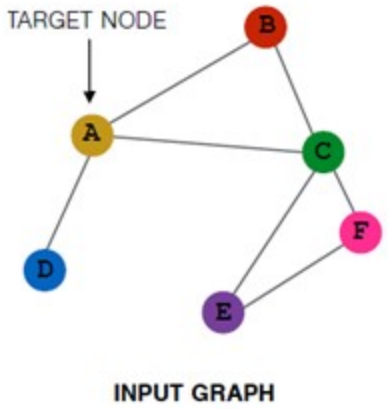


INPUT GRAPH

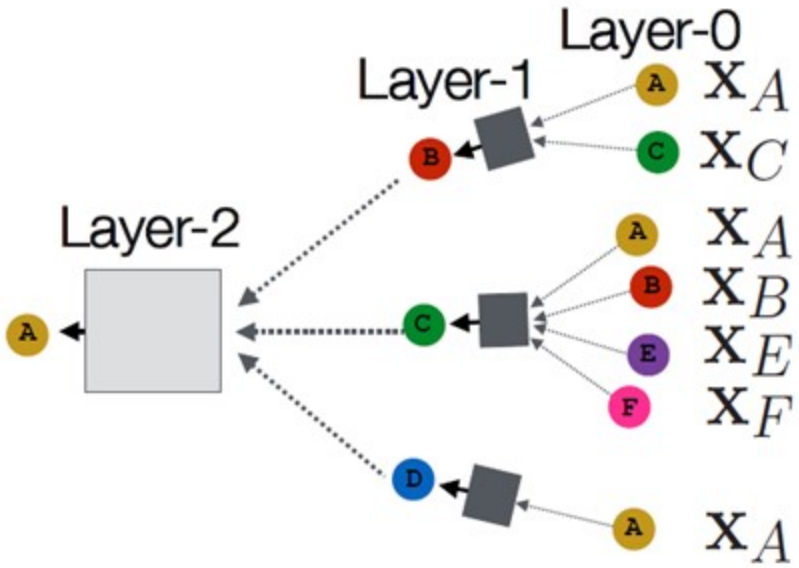


Graph Convolutional Networks

- Deep Model : Many Layers
  - Model can be of arbitrary depth:
    - Nodes have embeddings at each layer
    - Layer-0 embedding of node v is its input feature, x_v
    - Layer-k embedding gets information from nodes that are k hops away

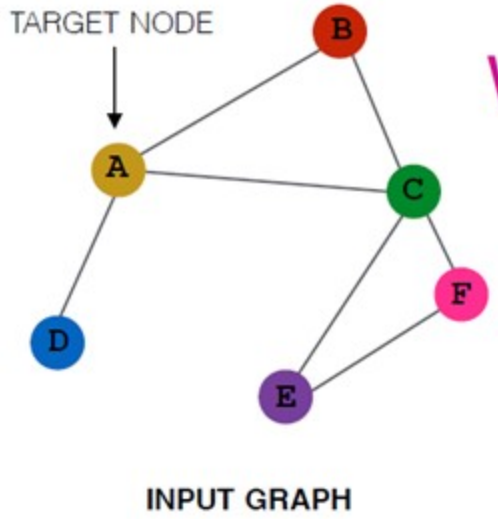


INPUT GRAPH

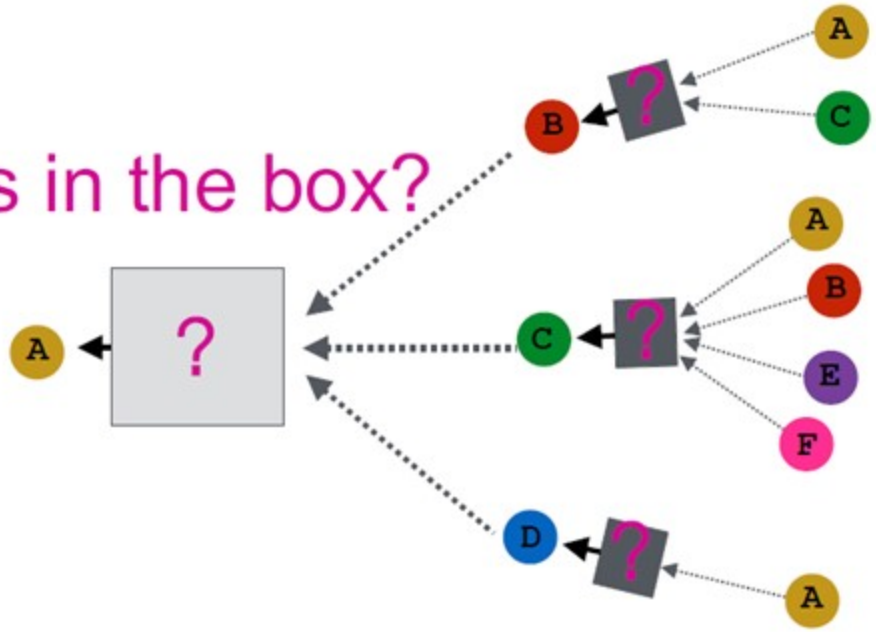


Graph Convolutional Networks

- Neighborhood Aggregation
  - **Neighborhood aggregation:** Key distinctions are in how different approaches aggregate information across the layers

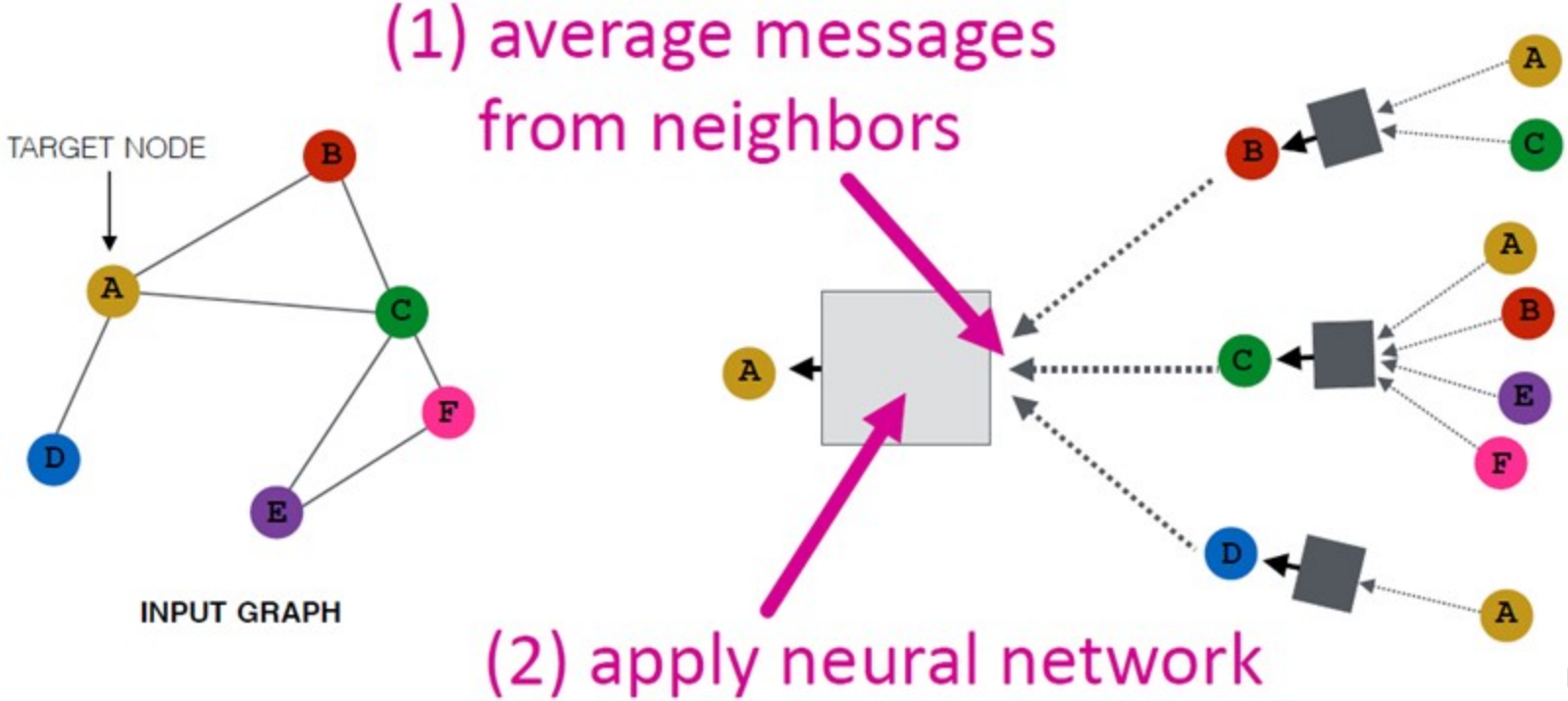


What is in the box?



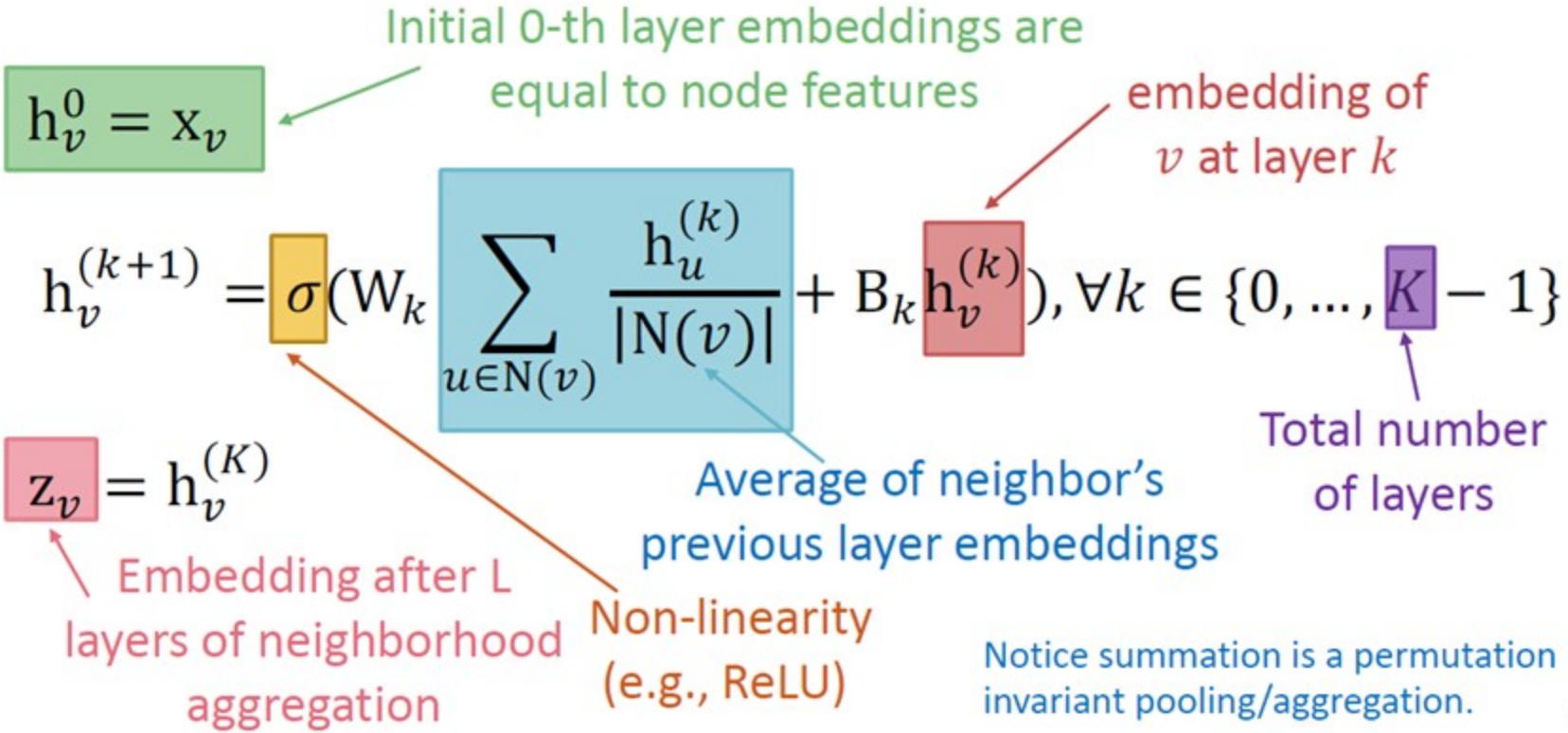
Graph Convolutional Networks

- Neighborhood Aggregation
  - **Basic approach:** Average information from neighbors and apply a neural network



Graph Convolutional Networks

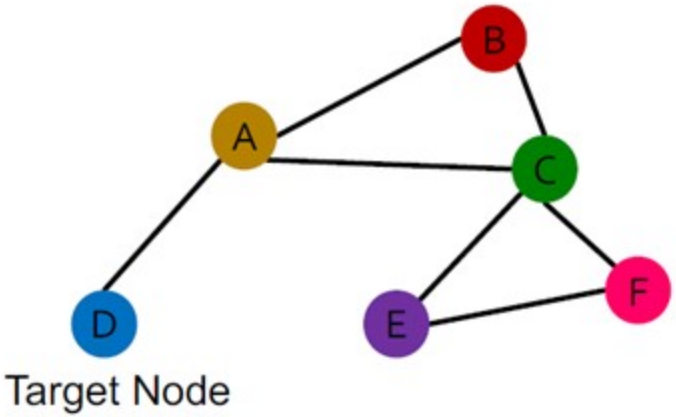
- Deep Encoder
  - **Basic approach:** Average neighbor messages and apply a neural network

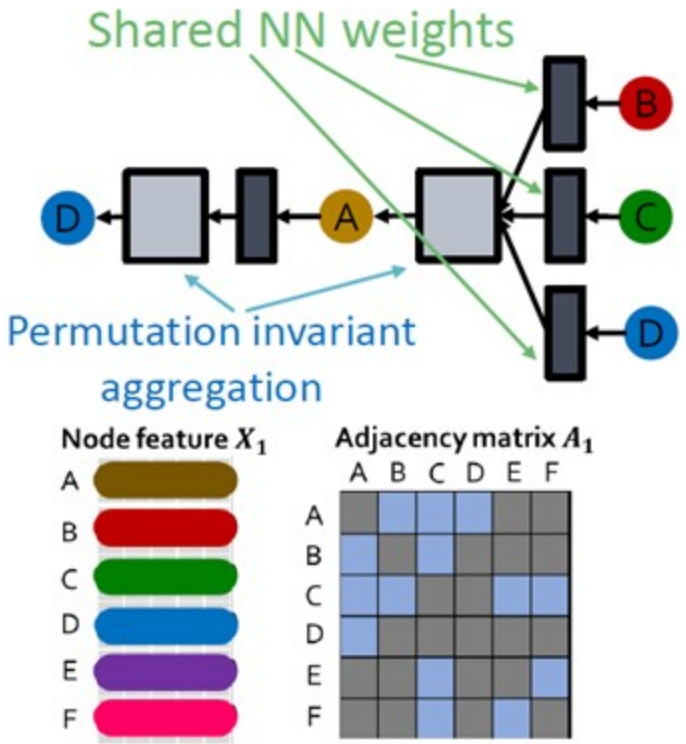


Graph Convolutional Networks

- Equivariant Property

  Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.

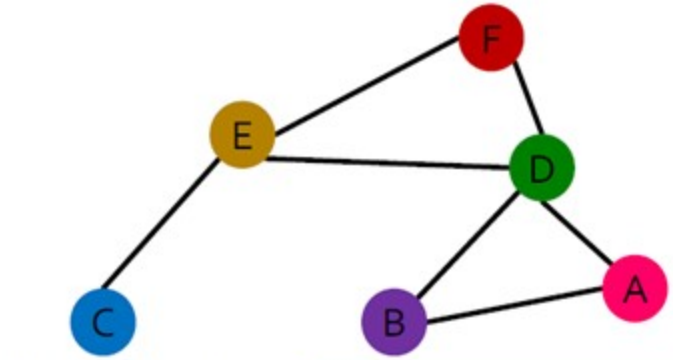




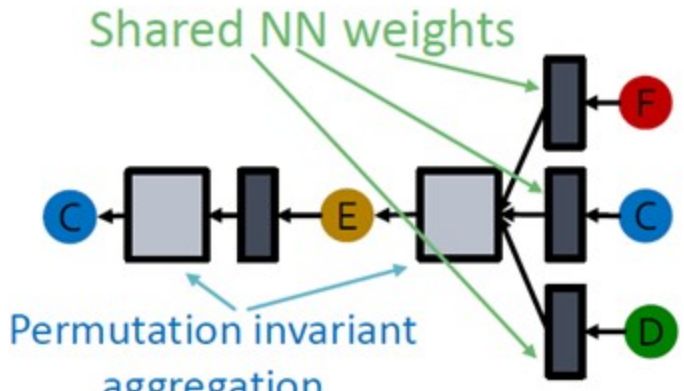
Graph Convolutional Networks

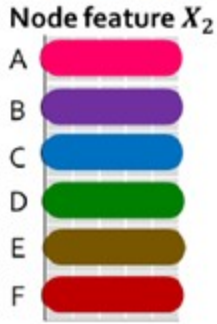
- Equivariant Property

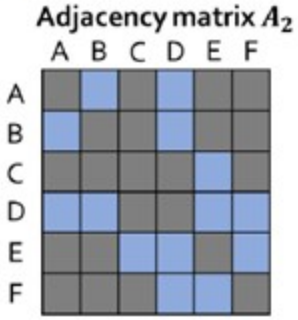
Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.



The target node (blue) has the same computation graph for different order plans



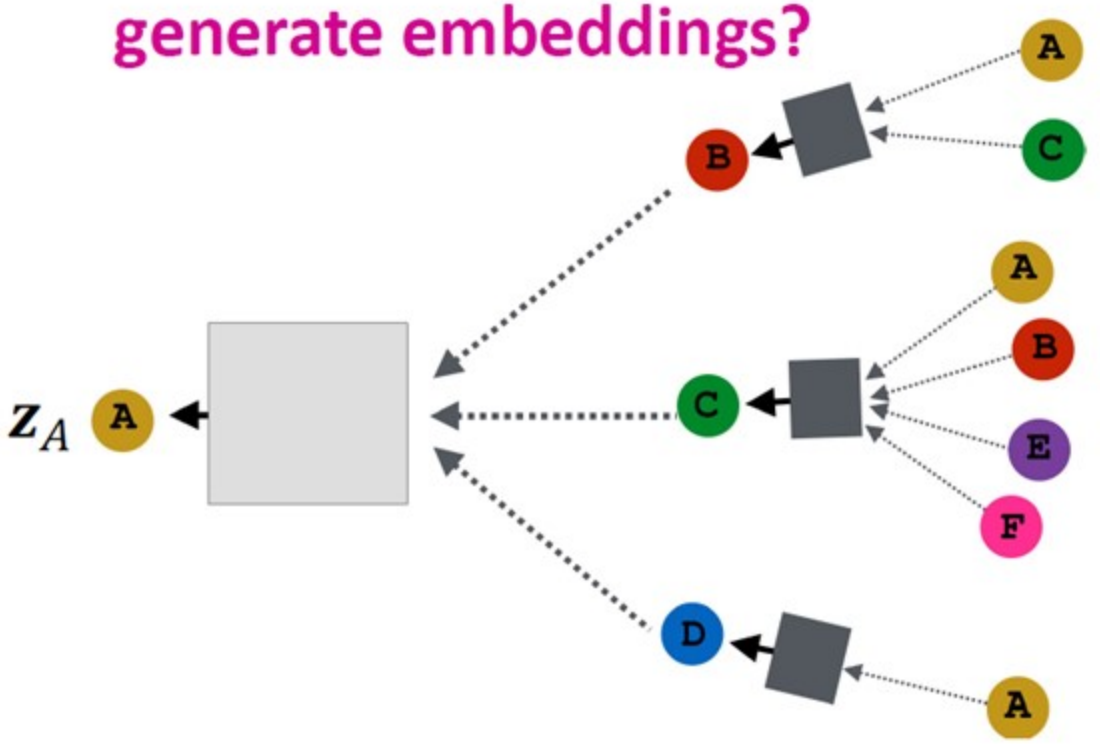




Graph Convolutional Networks

- Training the Model

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings.

Graph Convolutional Networks

- Model Parameters

Trainable weight matrices
(i.e., what we learn)

$$h_v^{(0)} = x_v$$

$$h_v^{(k+1)} = \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}\right), \forall k \in \{0..K-1\}$$

$$z_v = h_v^{(K)}$$

Final node embedding

We can feed these **embeddings into any loss function** and run SGD to **train the weight parameters**

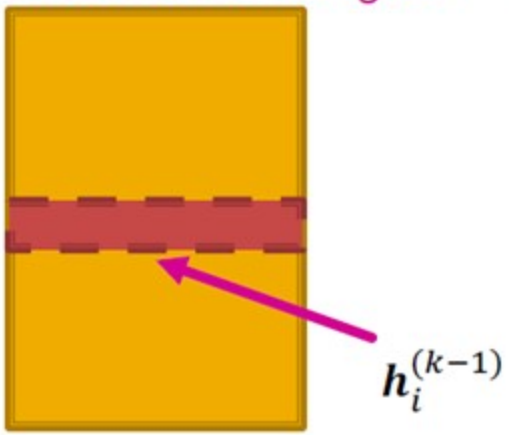
h_v^k: the hidden representation of node v at layer k

- W_k: weight matrix for neighborhood aggregation
- B_k: weight matrix for transforming hidden vector of self

Graph Convolutional Networks

- Matrix Formulation
  - **Many aggregations can be performed efficiently by (sparse) matrix operations**
    - Let $H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^{\mathrm{T}}$
    - Then: $\sum_{u \in N_v} h_u^{(k)} = \mathrm{A}_{v,:}\mathrm{H}^{(k)}$
    - Let D be diagonal matrix where $D_{v,v} = \mathrm{Deg}(v) = |N(v)|$
      - The inverse of D: D^{-1} is also diagonal: $D_{v,v}^{-1} = 1/|N(v)|$
    - **Therefore,**

Matrix of hidden embeddings $H^{(k-1)}$



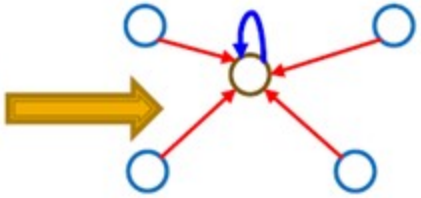
$$\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|} \Rightarrow H^{(k+1)} = D^{-1}AH^{(k)}$$

Graph Convolutional Networks

- Matrix Formulation
  - Re-writing update function in matrix form:

$$H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^{\mathrm{T}} + H^{(k)}B_k^{\mathrm{T}})$$

where $\tilde{A} = D^{-1}A$



$H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$

    - Red: neighborhood aggregation
    - Blue: self transformation

  - In practice, this implies that efficient sparse matrix multiplication can be used ($\tilde{A}$ is sparse)

  - **Note**: not all GNNs can be expressed in matrix form, when aggregation function is complex

Graph Convolutional Networks

- How to Train a GNN
  - Node embedding $\boldsymbol{z}_v$ is a function of input graph
  - **Supervised setting**: we want to minimize the loss $\mathcal{L}$

$$\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{z}_v))$$

    - $\boldsymbol{y}$: node label
    - $\mathcal{L}$ could be L2 if $\boldsymbol{y}$ is real number, or cross entropy if $\boldsymbol{y}$ is categorical
  - **Unsupervised setting:**
    - No node label available
    - **Use the graph structure as the supervision!**

Graph Convolutional Networks

- How to Train a GNN : **Unsupervised Training**
  - **"Similar" nodes have similar embeddings**

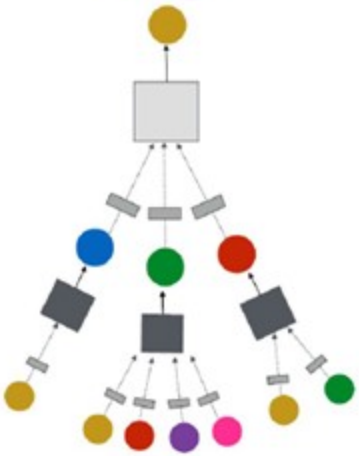
$$\mathcal{L} = \sum_{z_u, z_v} \mathrm{CE}(y_{u,v}, \mathrm{DEC}(z_u, z_v))$$

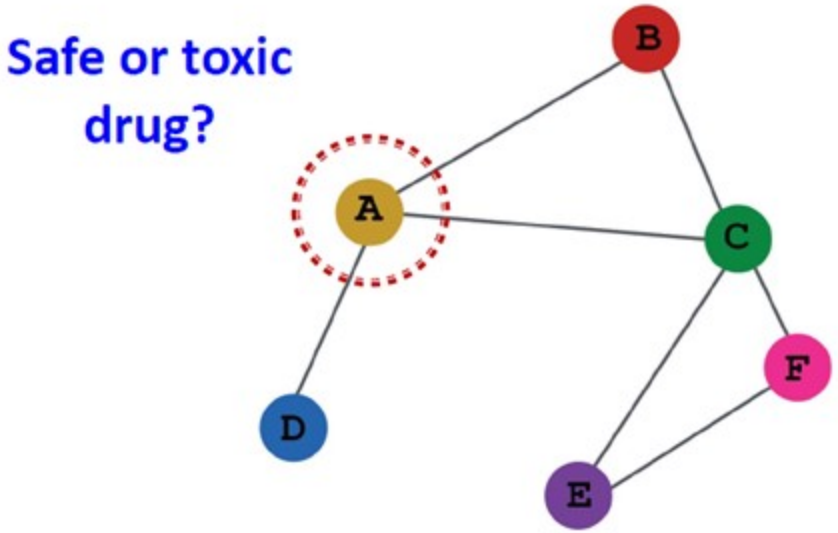
    - Where $y_{u,v} = 1$ when node u and v are **similar**
    - CE is the cross entropy
    - DEC is the decoder such as inner product
  - **Node similarity** can be anything from Lecture 3, e.g., a loss based on:
    - **Random walks** (node2vec, DeepWalk, struc2vec)
    - **Matrix factorization**
    - **Node proximity in the graph**

Graph Convolutional Networks

- How to Train a GNN : **Supervised Training**

  Directly train the model for a supervised task (e.g., node classification)





E.g., a drug-drug interaction network

Graph Convolutional Networks

- How to Train a GNN : **Supervised Training**

  Directly train the model for a supervised task (e.g., **node classification**)
  - Use cross entropy loss

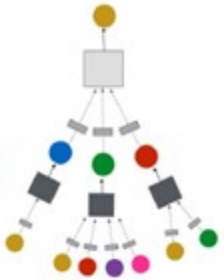
$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))$$

Encoder output: node embedding

Classification weights

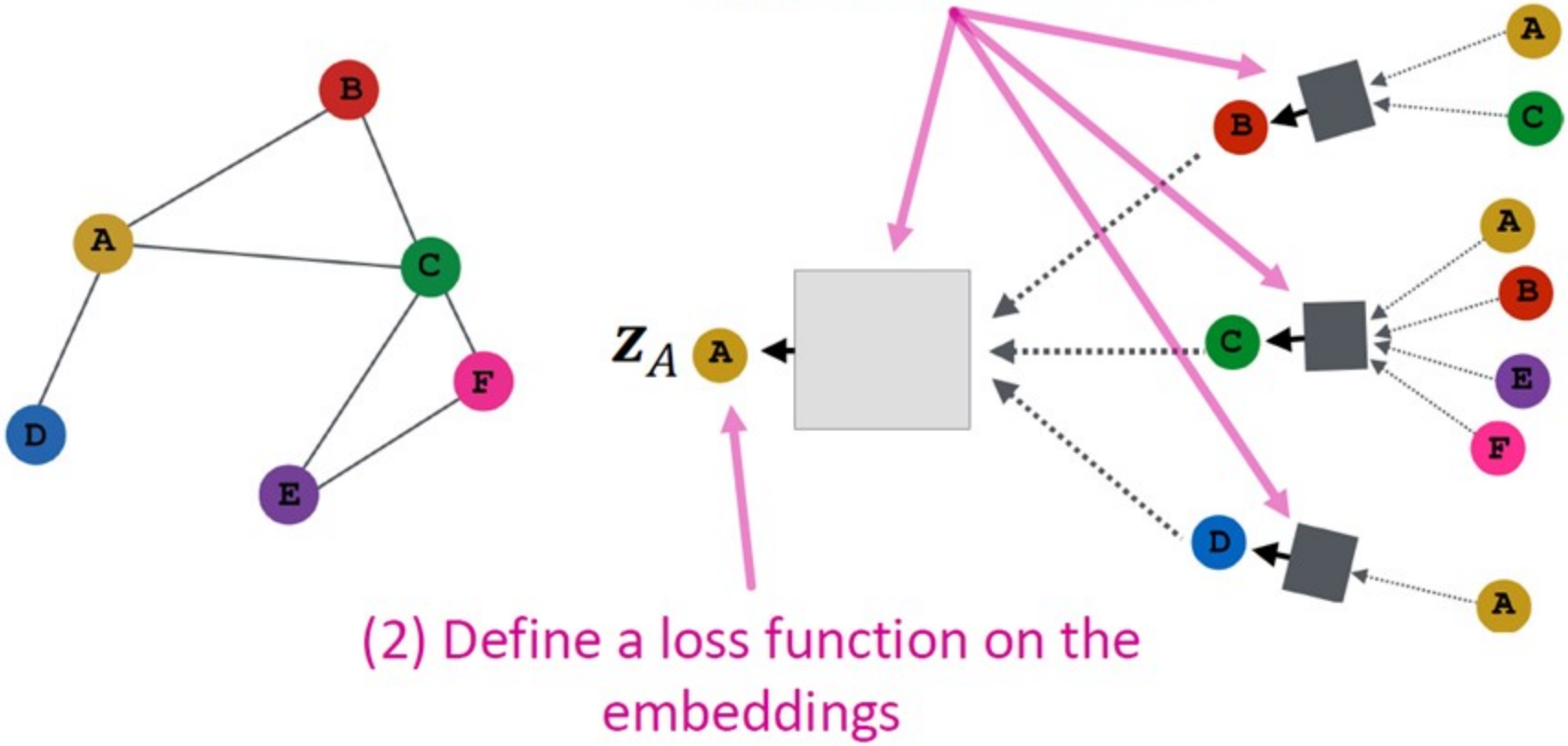
Node class label

Safe or toxic drug?



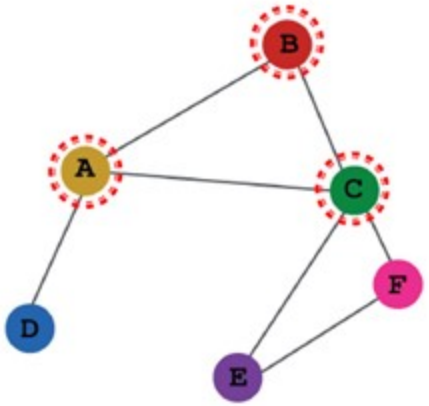
Graph Convolutional Networks

- Model Design Overview



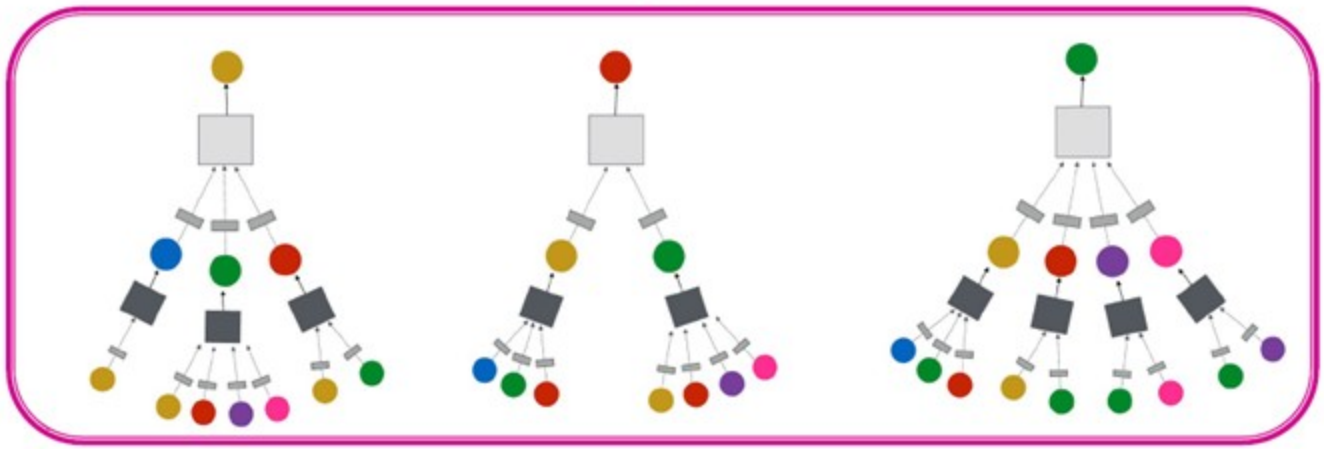
Graph Convolutional Networks

- Model Design Overview



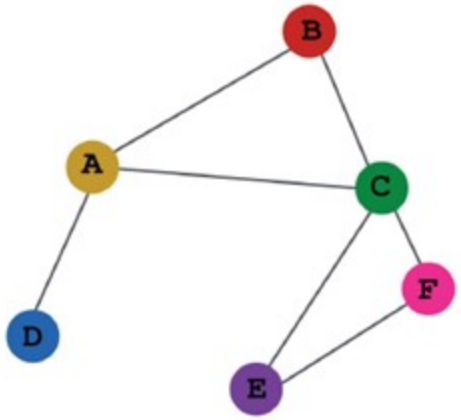
INPUT GRAPH

(3) Train on a set of nodes, i.e., a batch of compute graphs



Graph Convolutional Networks

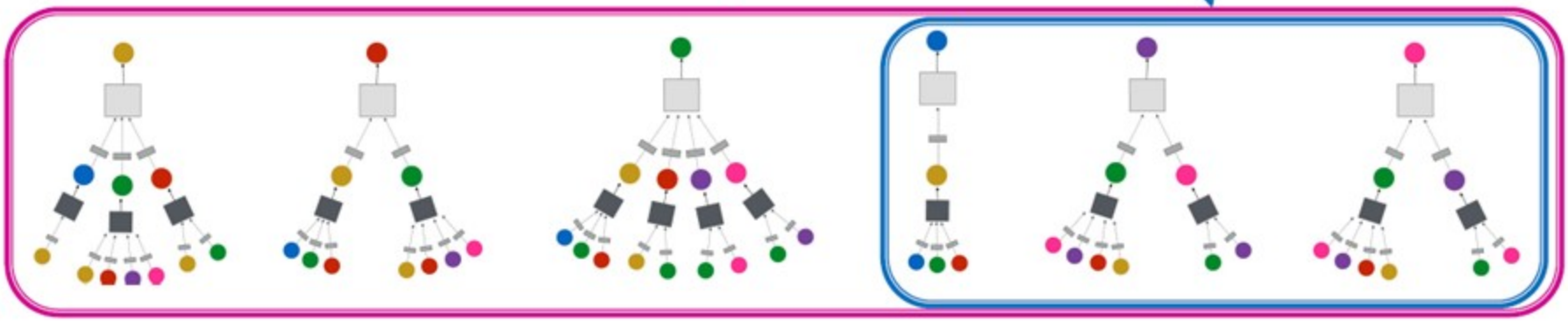
- Model Design Overview



INPUT GRAPH

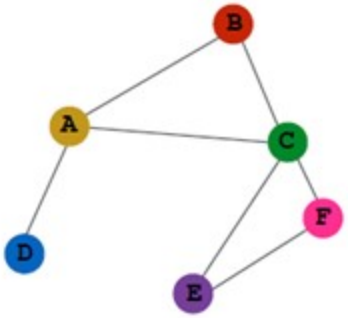
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

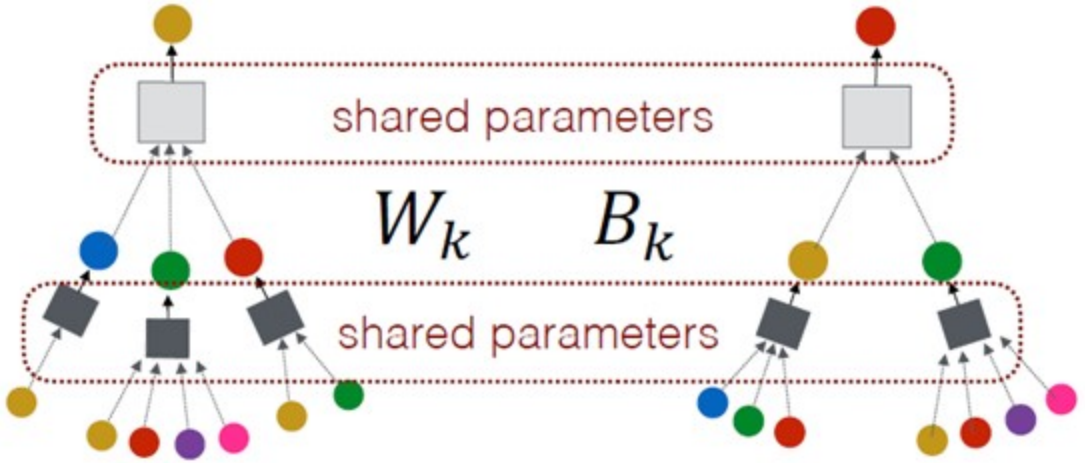


Graph Convolutional Networks

- Inductive Capability
- **The same aggregation parameters are shared for all nodes:**
  - The number of model parameters is sublinear in $|V|$ and we can **generalize to unseen nodes**!



INPUT GRAPH

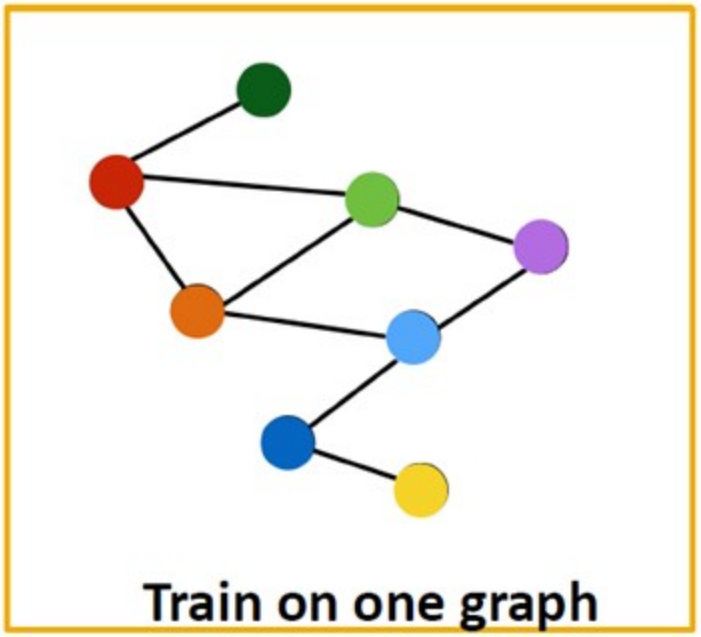


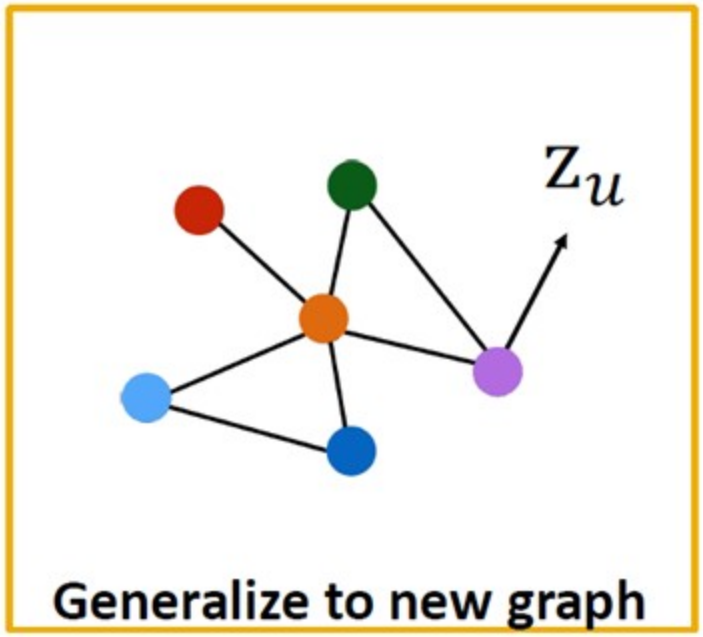
Compute graph for node A

Compute graph for node B

Graph Convolutional Networks

- Inductive Capability : New Graphs



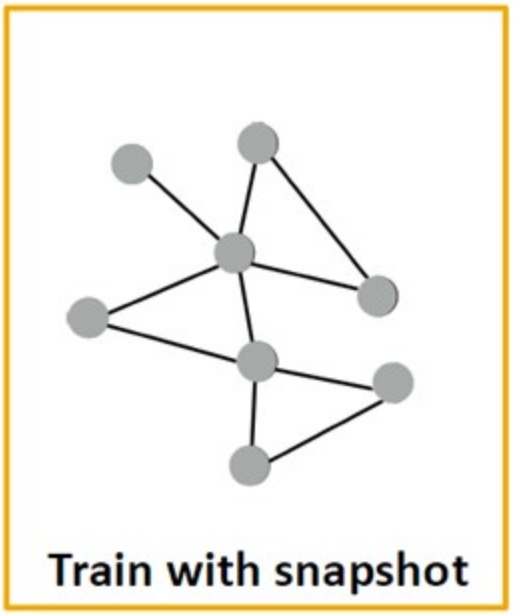


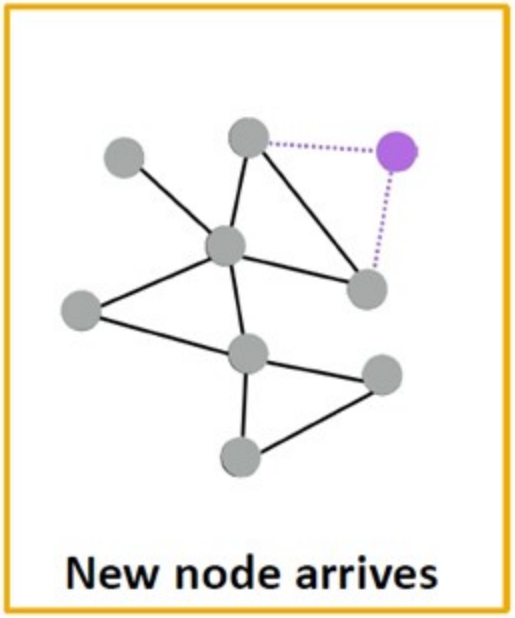
Inductive node embedding → Generalize to entirely unseen graphs

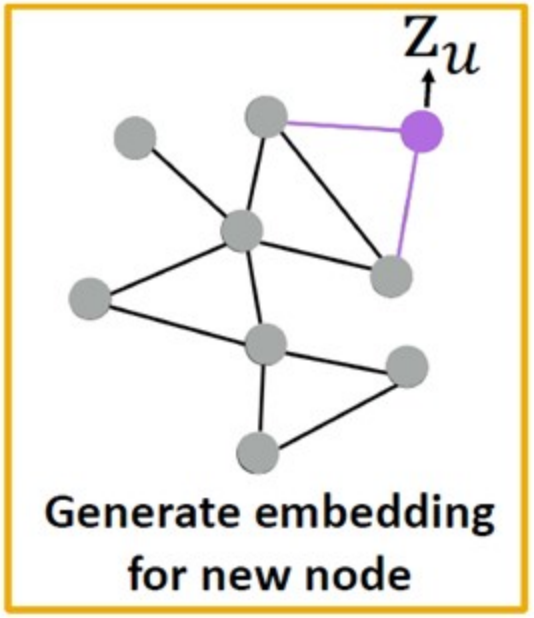
E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

Graph Convolutional Networks

- Inductive Capability : New Nodes

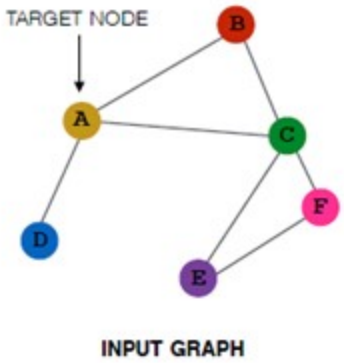






- Many application settings constantly encounter previously unseen nodes:
  - E.g., Reddit, YouTube, Google Scholar
- Need to generate new embeddings "on the fly"

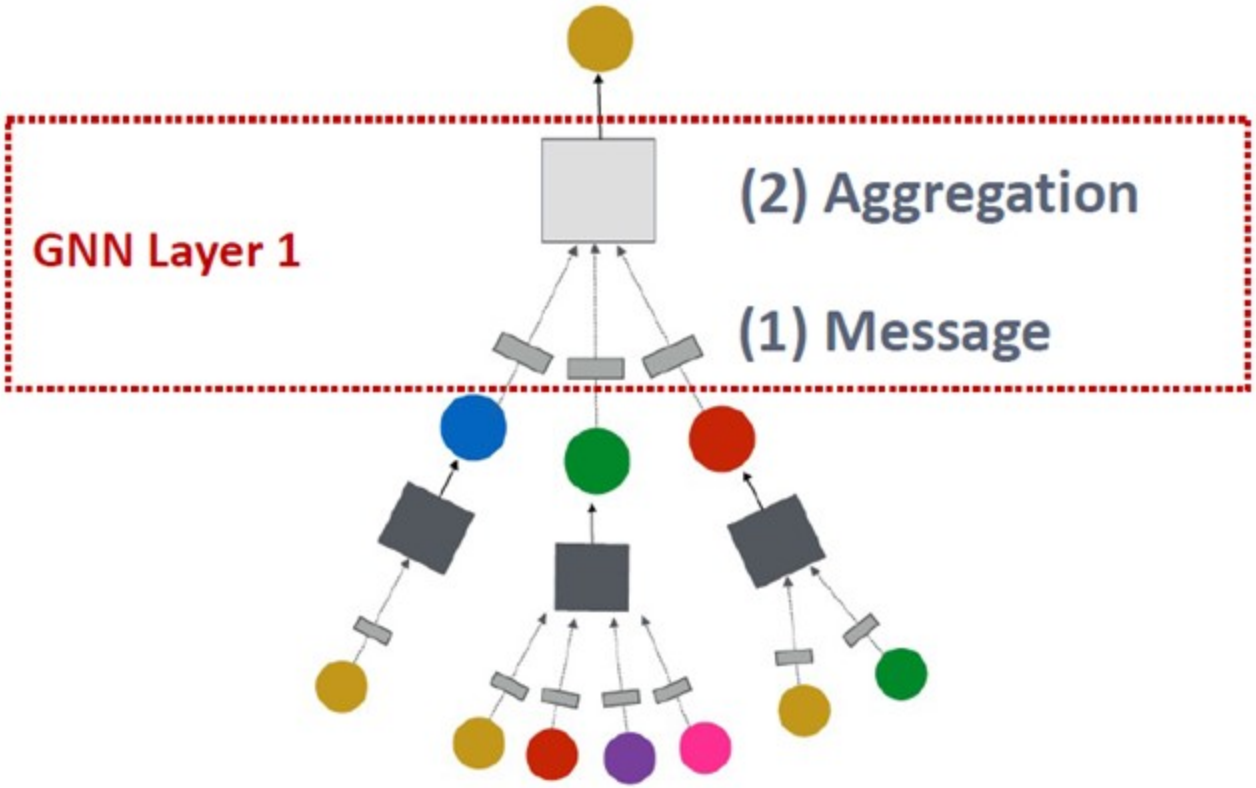
GNN Framework



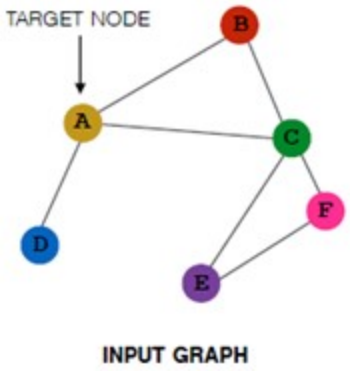
INPUT GRAPH

GNN Layer = Message + Aggregation

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



GNN Framework

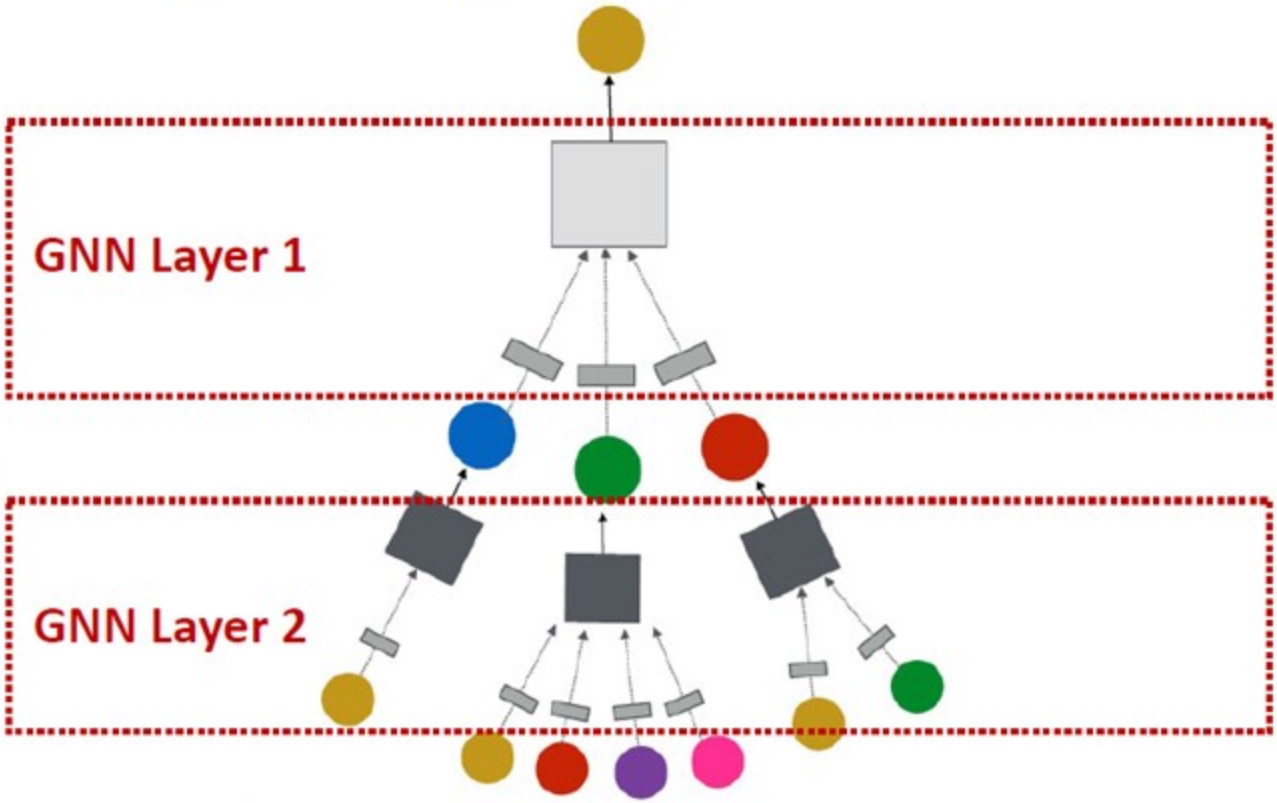


INPUT GRAPH

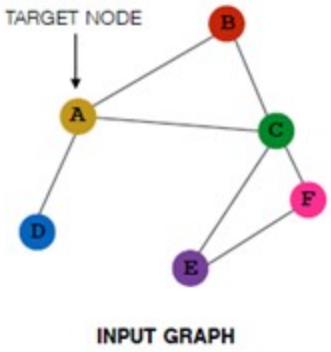
Connect GNN layers into a GNN

- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity

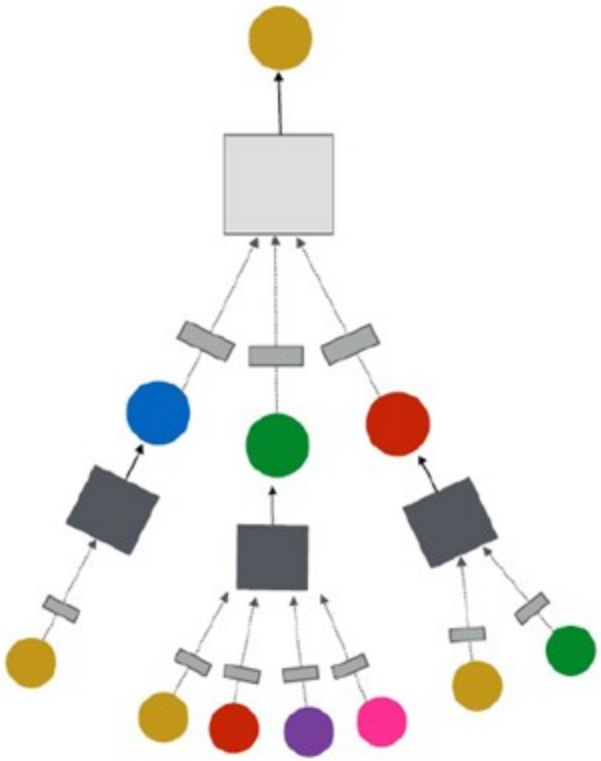


GNN Framework



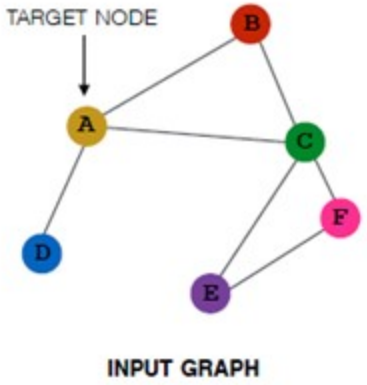
Idea: Raw input graph ≠ computational graph

- **Graph feature augmentation**
- **Graph structure augmentation**

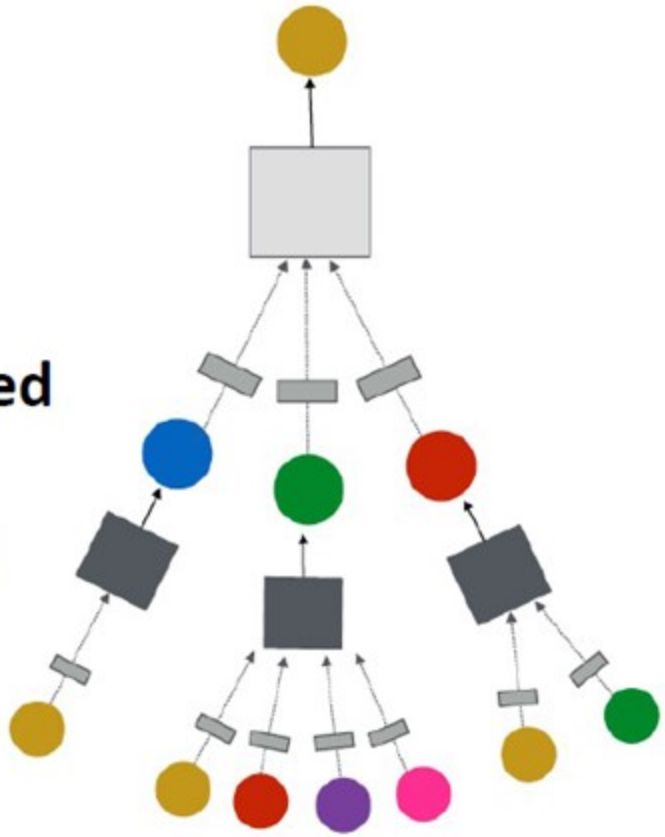


(4) Graph augmentation

GNN Framework



(5) Learning objective

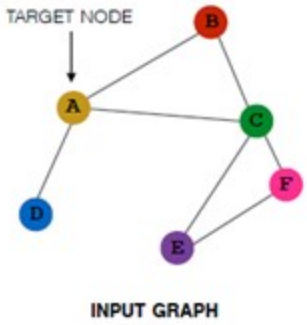


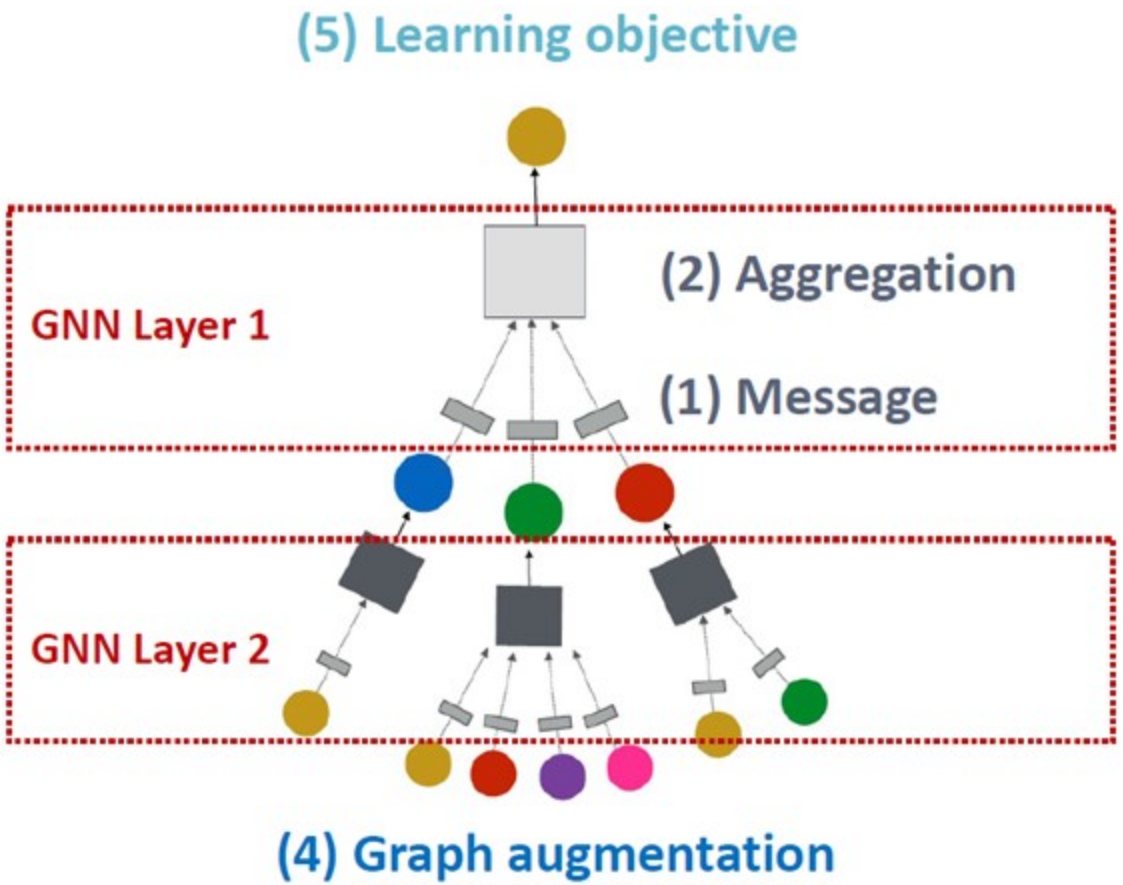
How do we train a GNN

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

GNN Framework

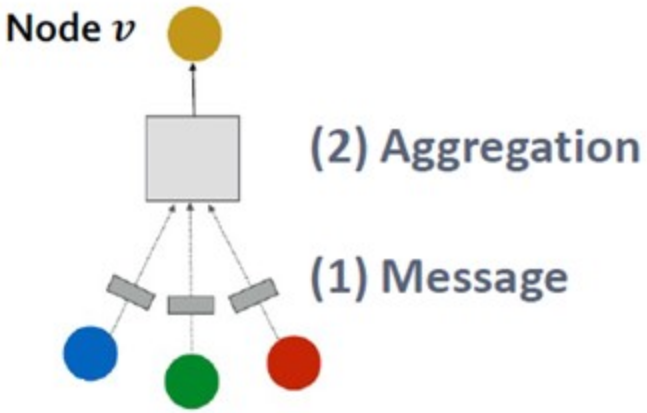
- Summary

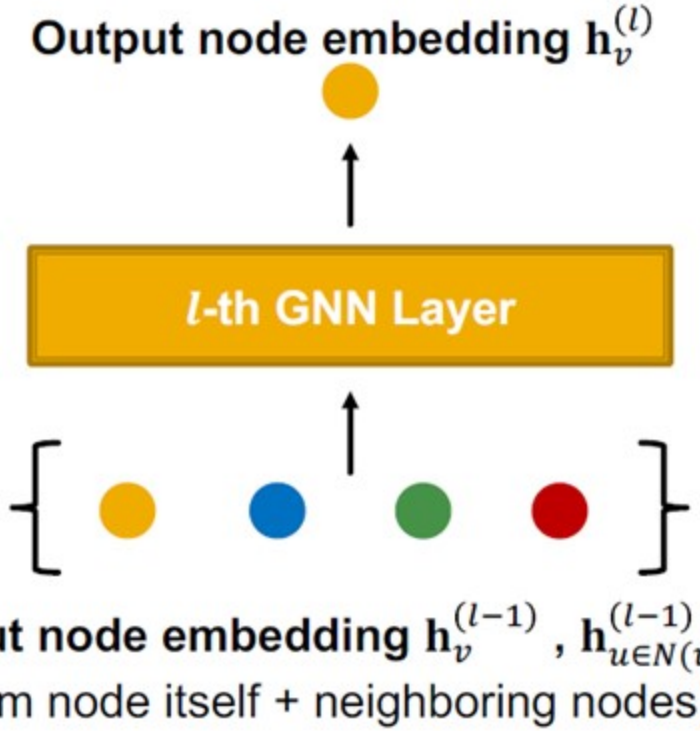




A Single Layer of a GNN

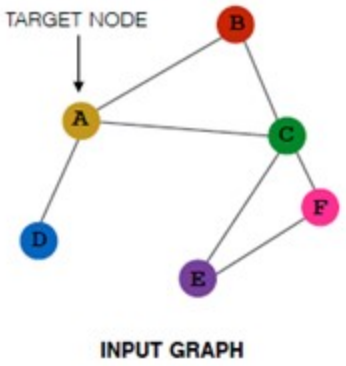
- **Idea of a GNN Layer:**
  - Compress a set of vectors into a single vector
  - **Two-step process:**
    - **(1) Message**
    - **(2) Aggregation**

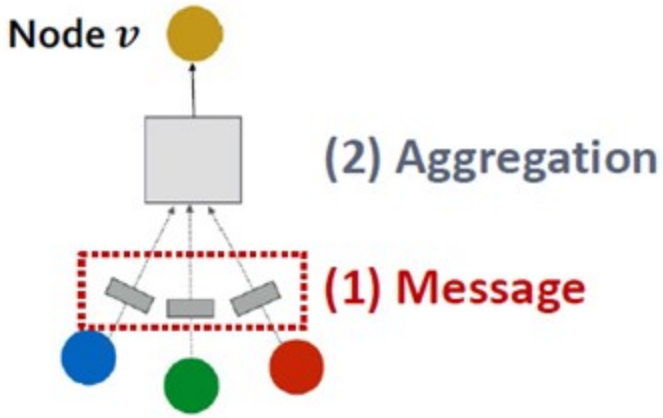




A Single Layer of a GNN

- **(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 $\mathbf{W}^{(l)}$



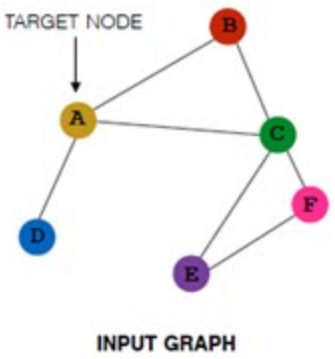


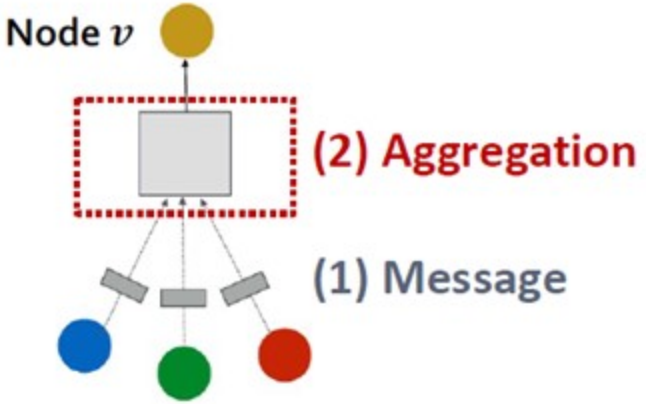
A Single Layer of a GNN

- **(2) Aggregation**
  - **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
    - $\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$





A Single Layer of a GNN

- **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

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

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

  - (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 Layer of a GNN

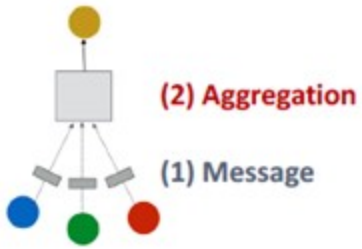
- **Putting things together:**
  - **(1) Message**: each node computes a message

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

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

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

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



A Single Layer of a GNN : GCN

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

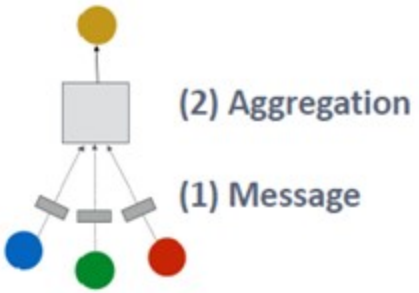
$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

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

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

Message

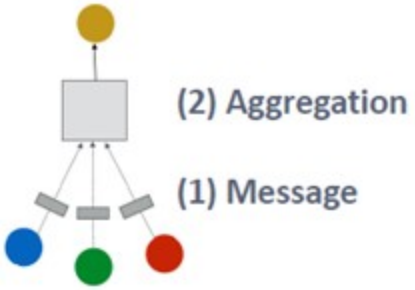
Aggregation



A Single Layer of a GNN : GCN

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

A Single Layer of a GNN : 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

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

A Single Layer of a GNN : GraphSAGE

- **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 $\text{Max}(\cdot)$

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

Aggregation — Message computation

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

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

Aggregation

A Single Layer of a GNN : GraphSAGE

- 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

A Single Layer of a GNN : GAT

- **(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

- **In GCN / GraphSAGE**
  - $\alpha_{vu} = \frac{1}{|N(v)|}$ is the **weighting factor (importance)** of node u's message to node v
  - $\Longrightarrow \alpha_{vu}$ 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

A Single Layer of a GNN : GAT

- **(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.

A Single Layer of a GNN : GAT

Can we do better than simple neighborhood aggregation?

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

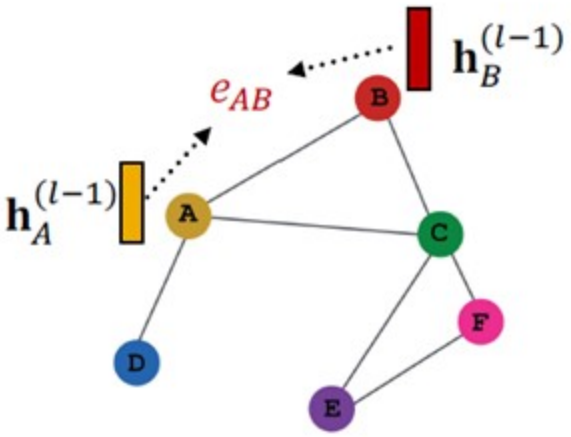
- **Goal:** Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea:** Compute embedding $\boldsymbol{h}_v^{(l)}$ of each node in the graph following an **attention strategy**:
  - Nodes attend over their neighborhoods' message
  - Implicitly specifying different weights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

A Single Layer of a GNN : GAT

- 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**



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

A Single Layer of a GNN : GAT

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

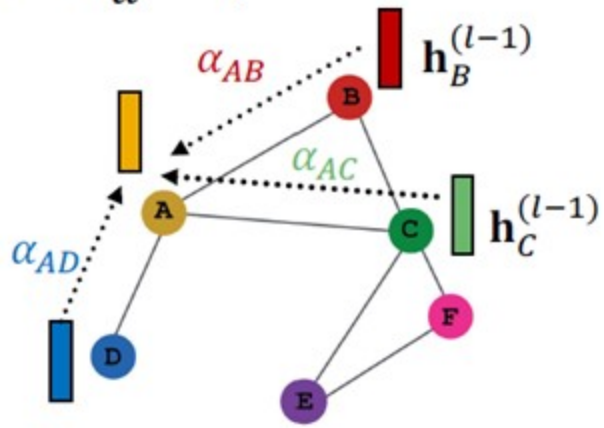
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

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

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

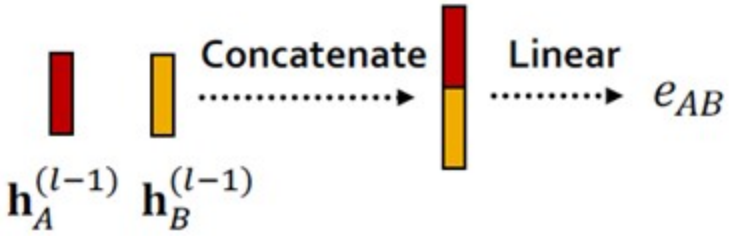
Weighted sum using α_{AB}, α_{AC}, α_{AD}:

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



A Single Layer of a GNN : GAT

- **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)

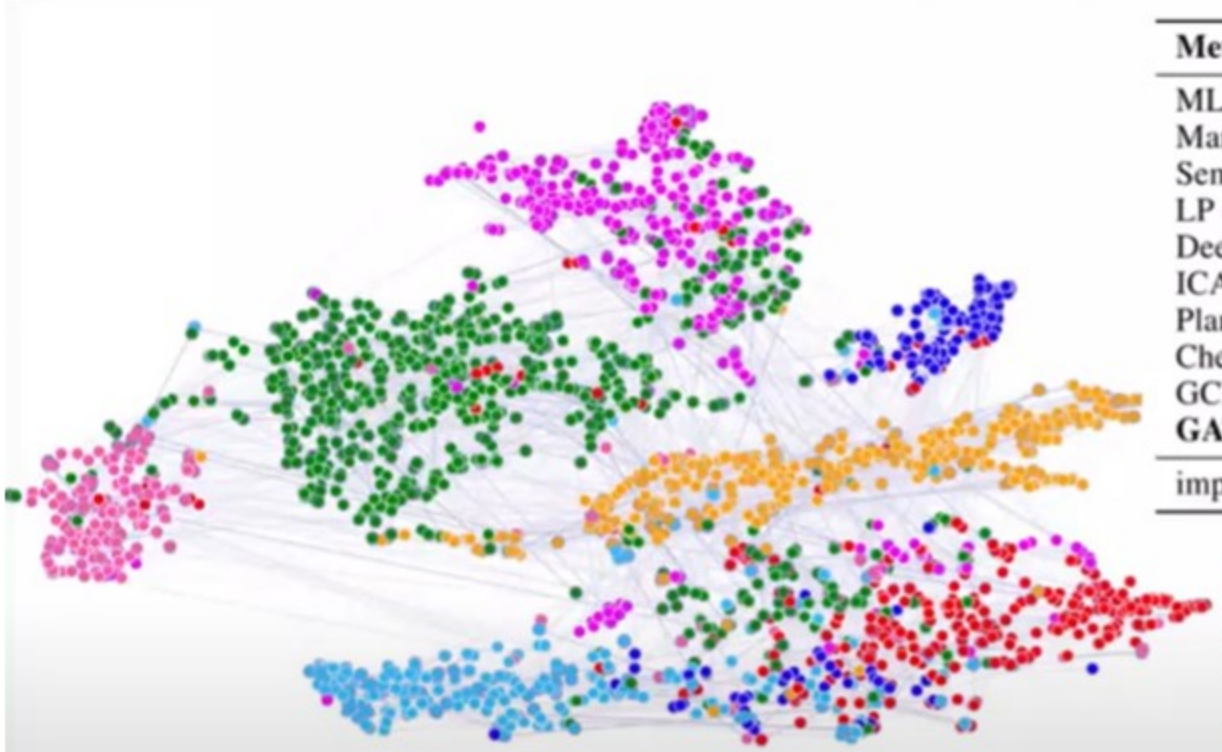


$$e_{AB} = a\left(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}\right)$$
$$= \text{Linear}\left(\text{Concat}\left(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}\right)\right)$$

  - 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

A Single Layer of a GNN : GAT

- Example



Method	Cora
MLP	55.1%
ManiReg (Belkin et al., 2006)	59.5%
SemiEmb (Weston et al., 2012)	59.0%
LP (Zhu et al., 2003)	68.0%
DeepWalk (Perozzi et al., 2014)	67.2%
ICA (Lu & Getoor, 2003)	75.1%
Planetoid (Yang et al., 2016)	75.7%
Chebyshev (Defferrard et al., 2016)	81.2%
GCN (Kipf & Welling, 2017)	81.5%
GAT	**83.3%**
improvement w.r.t GCN	1.8%

Attention mechanism can be used with many different graph neural network models

In many cases, attention leads to performance gains