IT5429E-1-24 (24.1A01)(Fall 2024): Graph Analytics for Big Data

Week 3: Graph Neural Nets

Instructor: Thanh H. Nguyen

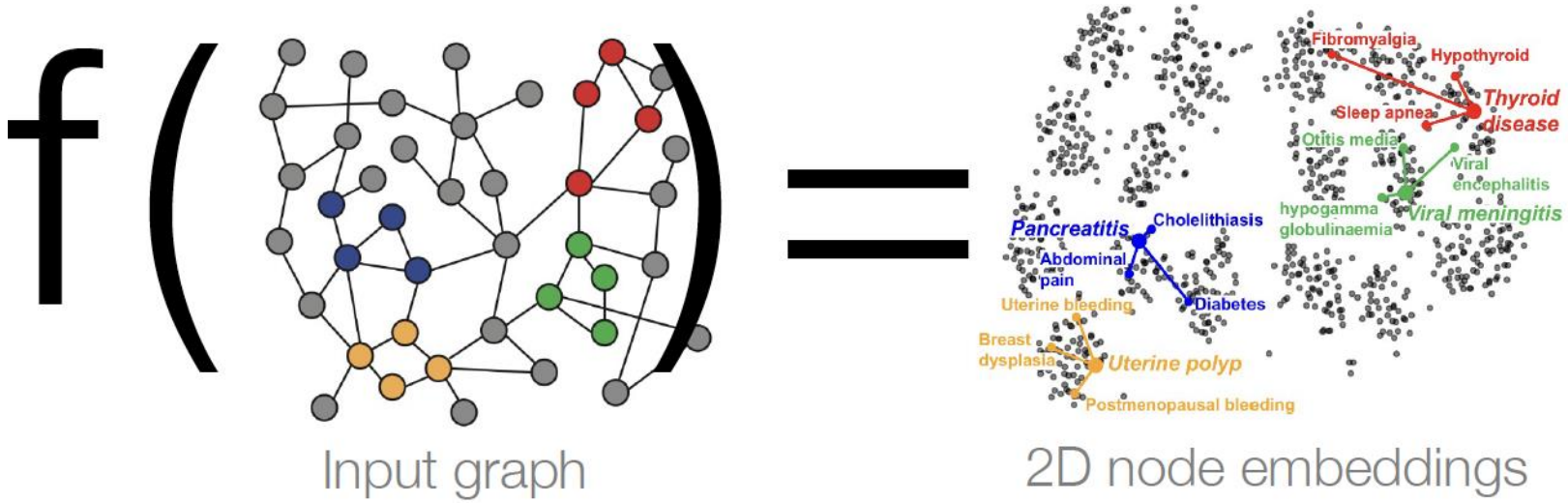
Many slides are adapted from https://web.stanford.edu/class/cs224w/

Reminder

- Class project
  - Project proposal (deadline: end of week 4)

Recap: Node Embeddings

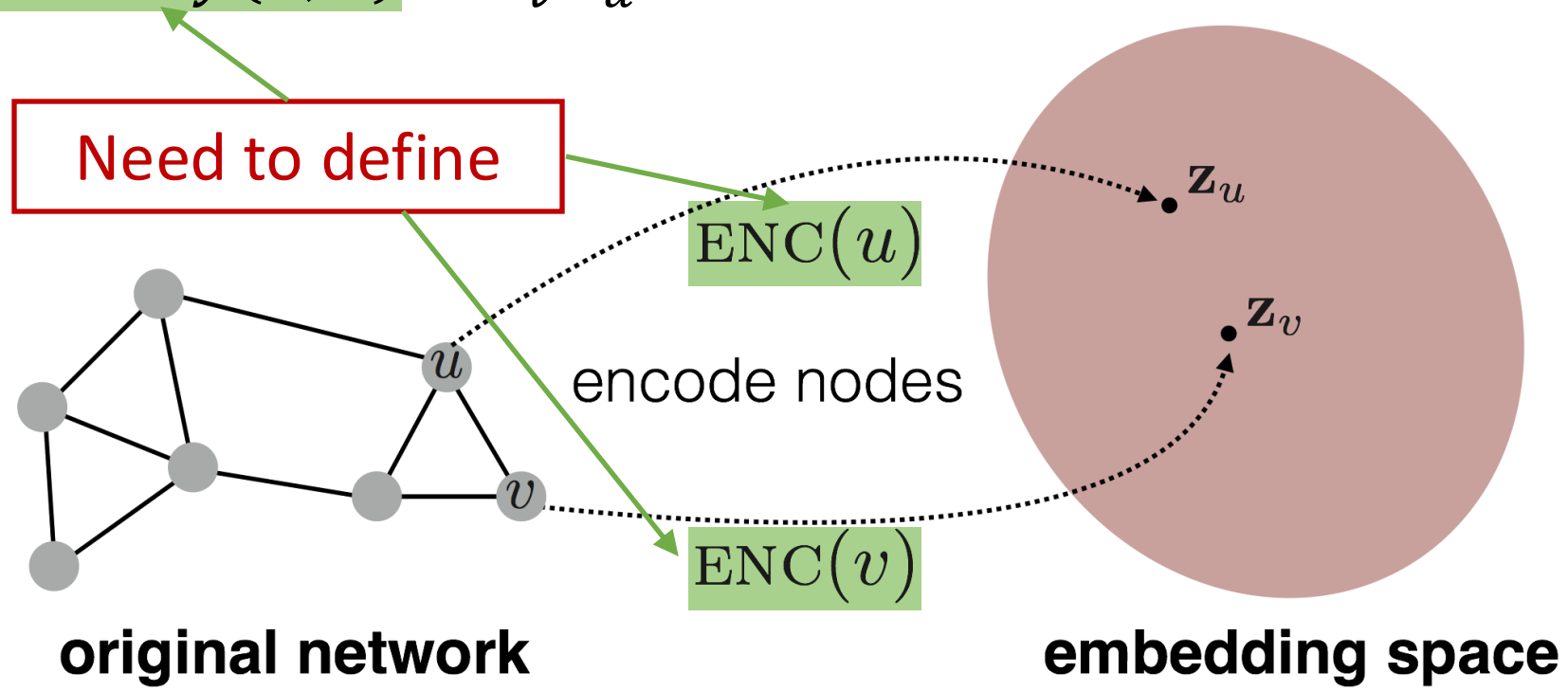
- Intuition: Map nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together



- Question: How to learn mapping function f?

Recap: Node Embeddings

- Goal: $similarity(u, v) \approx z_v^T z_u$



Recap: Two Key Components

- Encoder: Map each node to a low dimensional vector

$$ENC(v) = z_v$$

- Similarity function: Specify how relationships in embedding space map to the original space

$$similarity(u, v) \approx z_v^T z_u$$

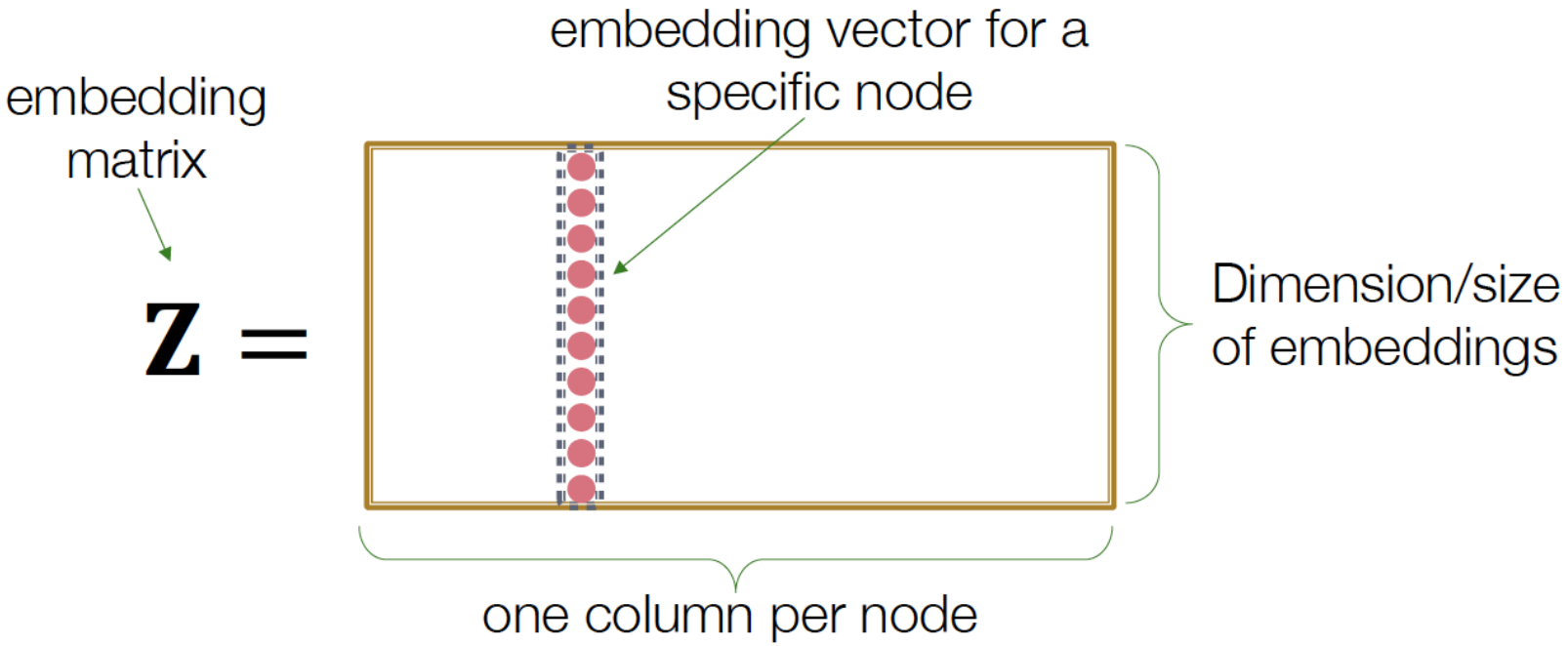
Decoder

Similarity of u and v in original network

Dot product between node embeddings

Recap: Shallow Encoding

- Simplest encoding approach: Encoder is an embedding lookup



Recap: Shallow Encoders

- Limitations:
  - $O(|V|d)$ parameters are needed
    - No parameter sharing between nodes
    - Every node has a unique embedding
  - Inherently "transductive"
    - Cannot generate embedding for nodes no seen during training
  - Do not incorporate node features
    - Nodes in many graphs have features that we can/should leverage

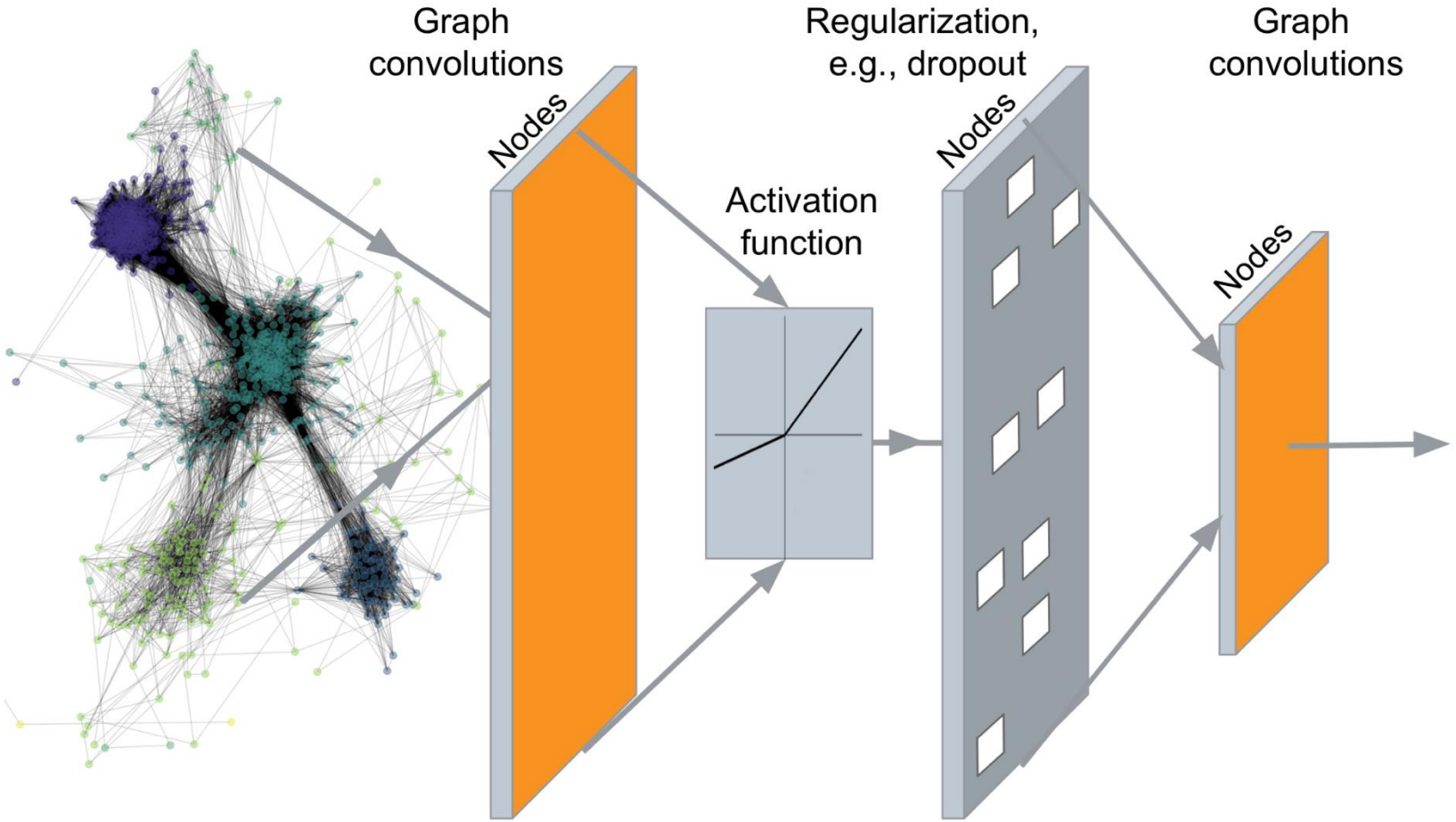
Today: Deep Graph Encoders

- Graph Neural Nets (GNNs)

$ENC(v) =$ Multiple layers of non-linear transformations based on graph structures

- All these deep encoders can be combined with node similarity functions defined previously

Deep Graph Encoders

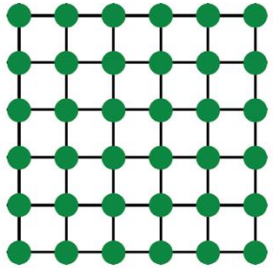


… Output: node embeddings. We can also embed sub-graphs, and graphs.

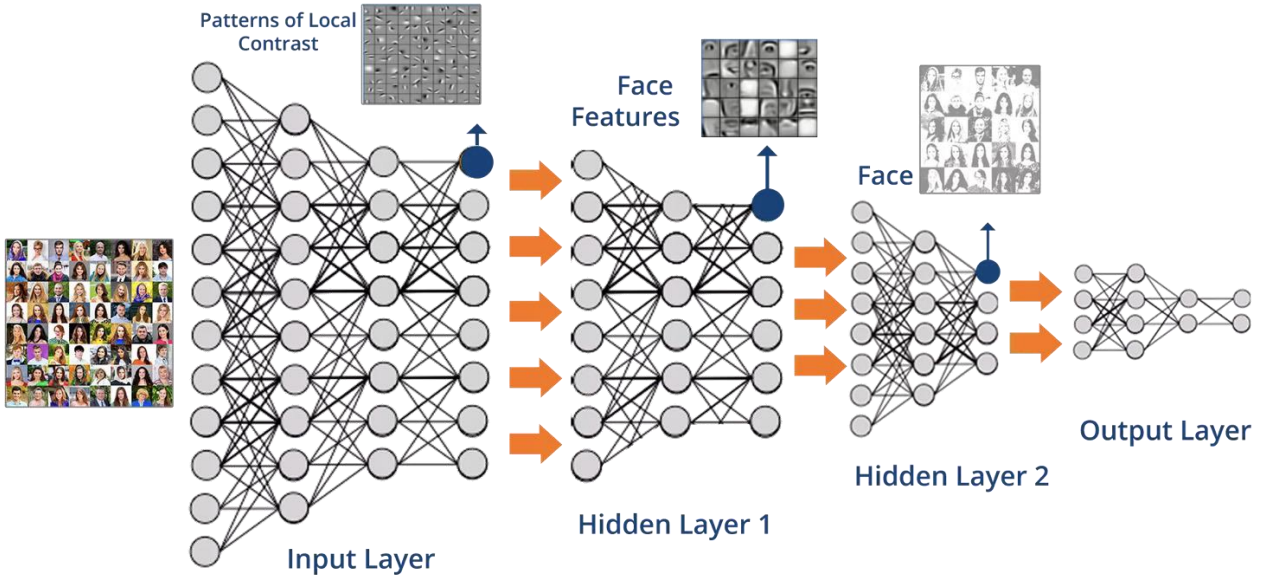
Tasks on Networks

- Node classification
  - Predict type of a given node
- Link prediction
  - Predict whether two nodes are linked
- Community detection
  - Identify densely linked clusters of nodes
- Network similarity
  - How similar are two (sub-)networks

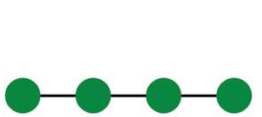
Modern ML Toolbox

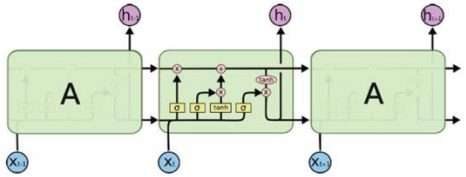


Images



Text/Speech



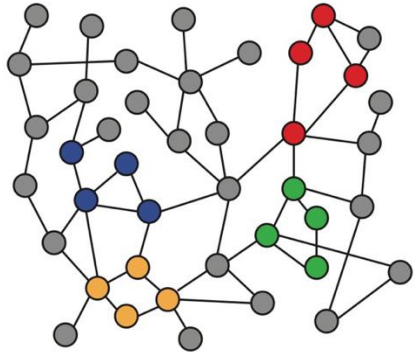


Modern deep learning toolbox is designed for simple sequences and grids

Why is Graph Deep Learning Hard?

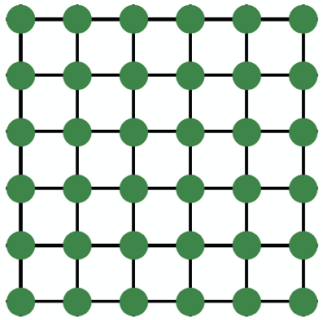
Networks are complex

- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)



Networks

vs



Images

Texts

- No fixed node ordering or reference point
- Often dynamic and have multimodal features

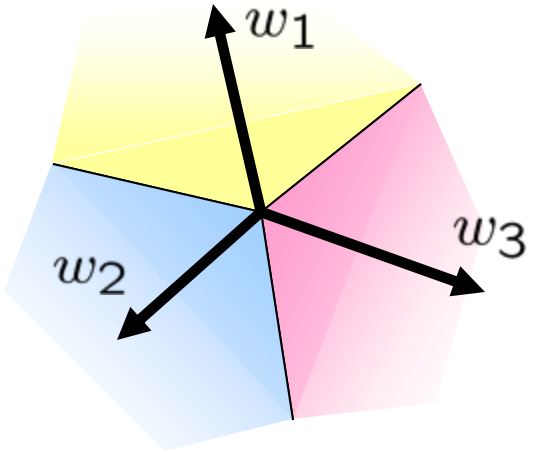
Outline

1. Basics of deep learning
2. Deep learning for graphs
3. Graph Convolutional Networks (GCNs)
4. GNNs versus CNNs

Basics of Deep Learning

Multiclass Logistic Regression

- Intermediate score:
  - A weight vector for each class: w_y
  - Score (activation) of a class y: $z_y = w_y \cdot x$

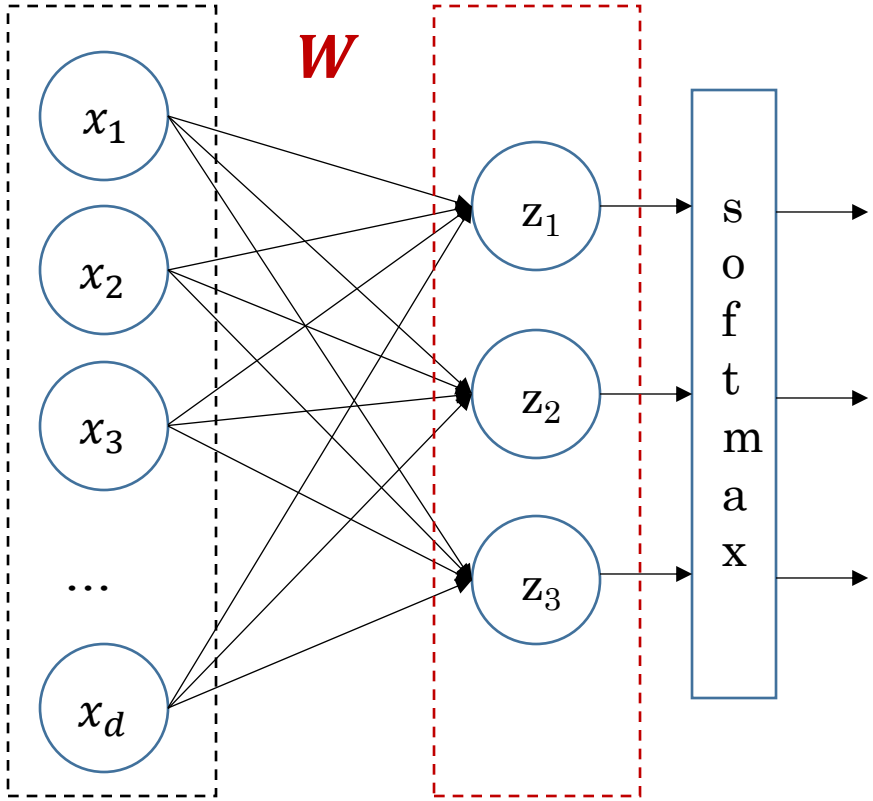


- Convert score into probabilities?
  - Soft-max function:

$$\underbrace{z_1, z_2, z_3}_{\text{original score}} \rightarrow \underbrace{\frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}}, \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}}, \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}}}_{\text{Softmax function}}$$

Multi-class Logistic Regression

- = special case of neural network



$$P(y_1) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_2) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_3) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

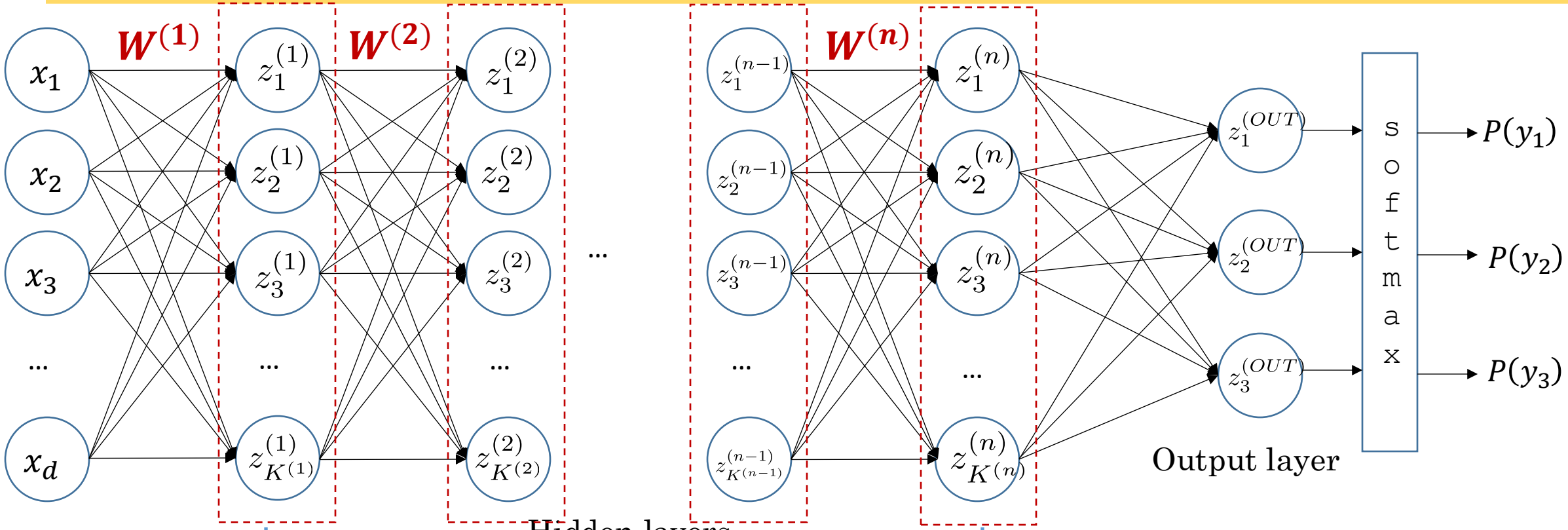
Input layer: $x = [x_1, x_2, \cdots, x_d]^T$

Intermediate (hidden) layer: $z = [z_1, z_2, z_3] = Wx$

Weight matrix: $W = \{w_{ij}\}$

Capture how much each input feature x_i influence each intermediate value z_j

Deep Neural Network = Learn the Features!



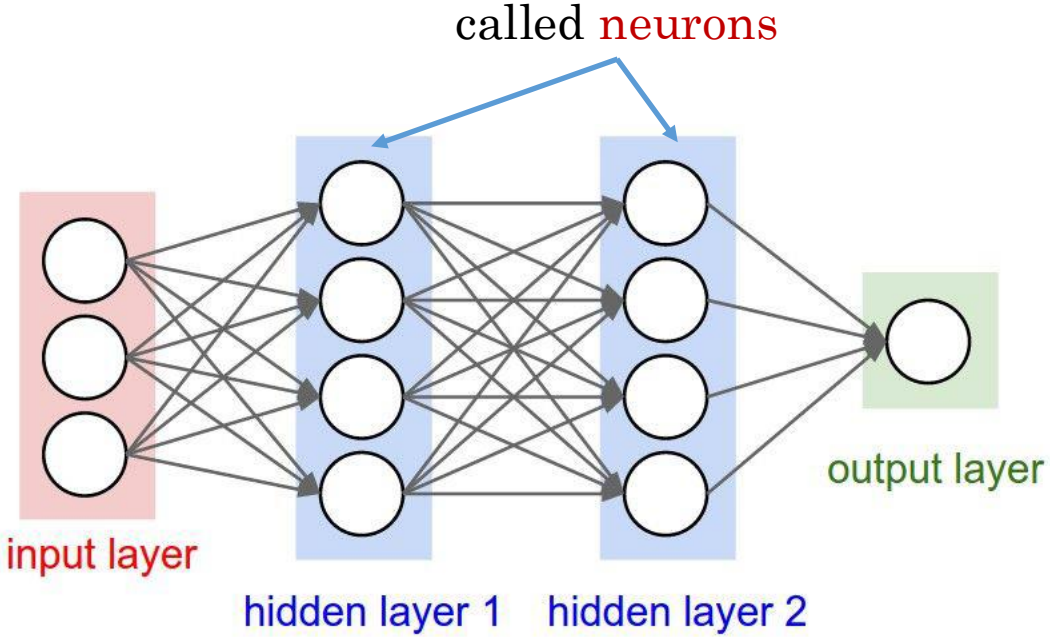
i^{th} hidden layer: $z^{(i)} = \left(z_1^{(i)}, z_2^{(i)}, \cdots, z_{K^{(i)}}^{(i)}\right)$ has hidden size $K^{(i)}$

$$z^{(i)} = \max\left\{0, W^{(i)} z^{(i-1)}\right\}$$

Element-wise max operator, called ReLu activation function

Train a Neural Net

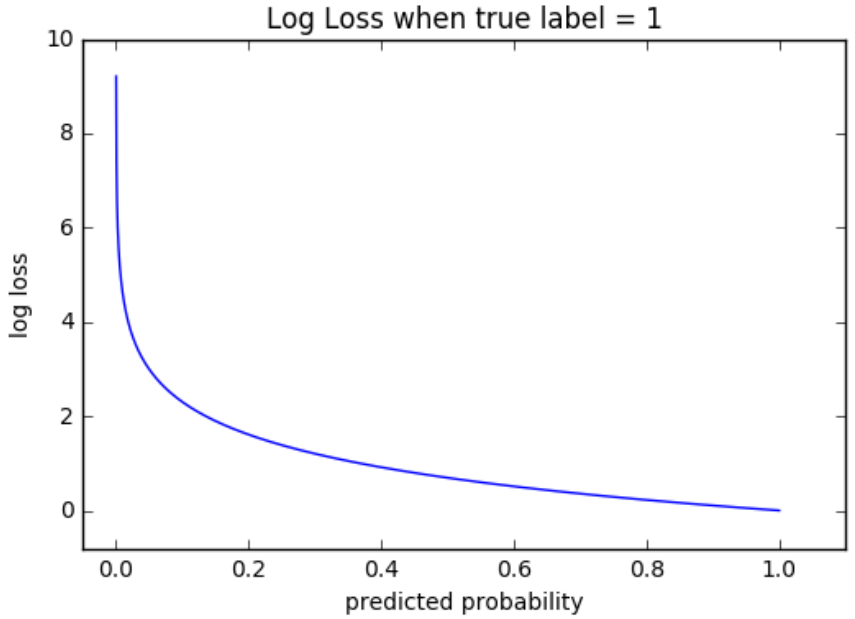
- Train a neural networks: gradient descent
  - Forward pass: input is passed forward through the network to produce a prediction output. A loss metric is computed based on the difference between prediction and target (true output).
  - Backward pass: derivatives of this loss metric are calculated and propagated back through the network using a technique called backpropagation.
- We will talk more about this!!!



Common Loss Functions

Cross Entropy

- Measures the performance of a classification model whose output is a probability value between 0 and 1
- Cross-entropy loss increases as the predicted probability diverges from the actual label.
- A perfect model would have a log loss of 0.



- Binary classification: $Loss = -(y \log p + (1-y)\log(1-p))$
- Multi-classification: $Loss = -\sum_l y_l \log p_l$
  - $y_l = 1$ if the true label is l, and $y_l = 0$, otherwise
  - p_l: predicted probability of having label l

Mean Square Error (MSE)

- Commonly used for regression loss
- Measure the average of squared difference between predictions and actual observations

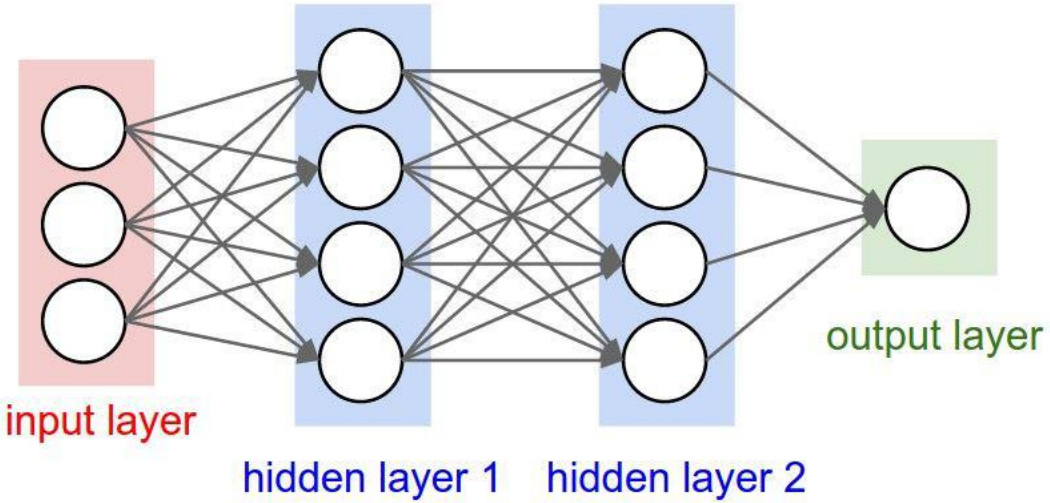
$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Summary: Fully Connected Layer

- Neural networks receive an input (a single vector), and transform it through a series of *hidden layers*

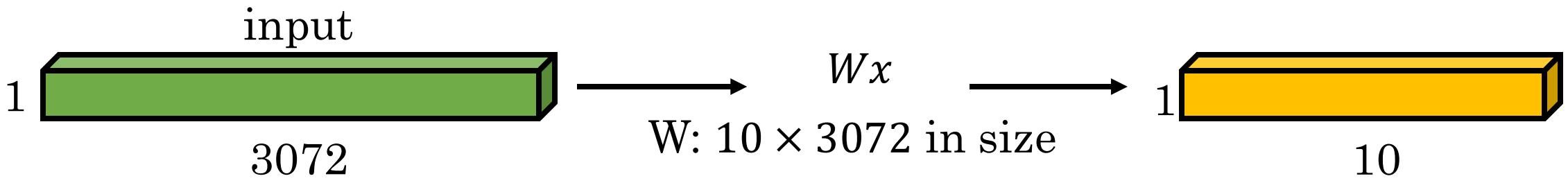
- Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer

- The last fully-connected layer is called the "output layer" and in classification settings it represents the class scores.



Fully Connected Layer with Images

- $32 \times 32 \times 3$ image is stretched to 3072×1



- Regular neural networks do not scale well to full images

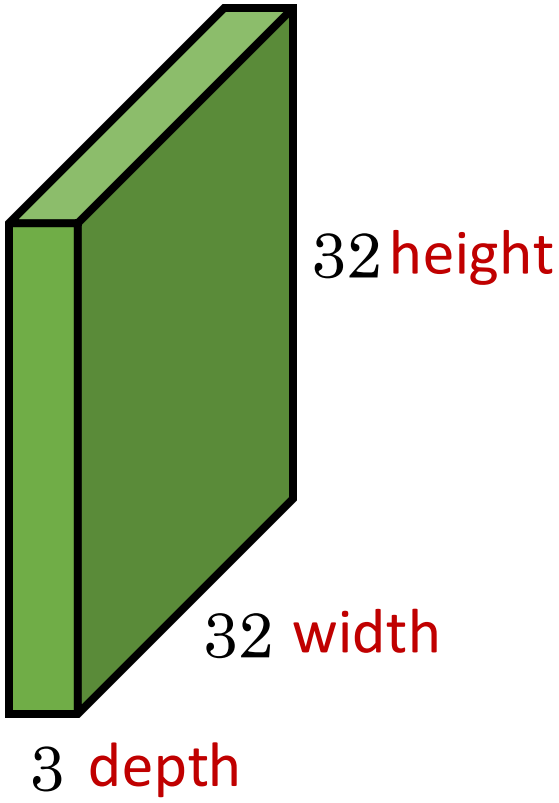
Convolutional Neural Networks (ConvNet)

- Layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth
  - *Depth* refers to the third dimension of an activation volume, not to the total number of layers in a network
  - Example: the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively)

- Main layers:
  - Convolutional layers
  - Pooling layers
  - Fully-connected layers

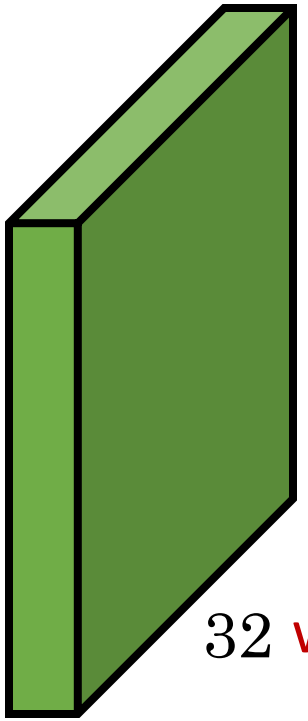
Convolutional Layer

- $32 \times 32 \times 3$ image is preserved spatial structure



Convolutional Layer

$32 \times 32 \times 3$

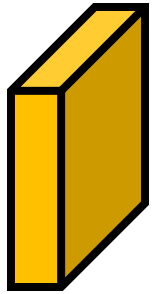


32 height

32 width

3 depth

$5 \times 5 \times 3$ filter (tensor) of learnable weights w

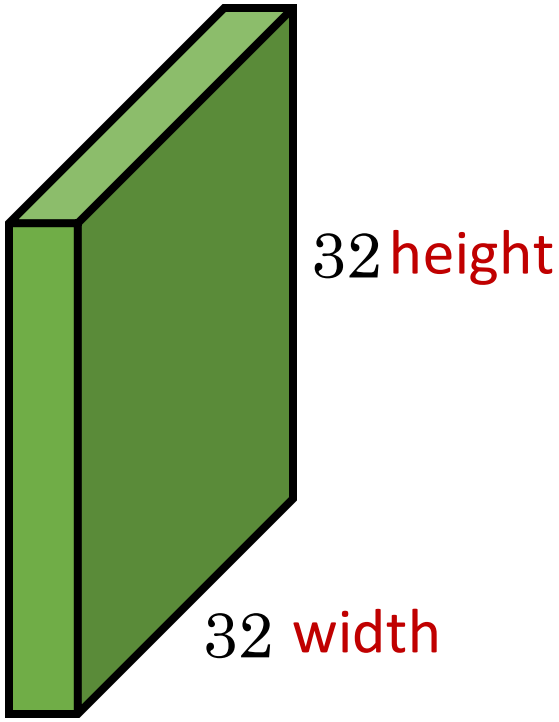


- Convolve the filter with the image, i.e., "slide over the image spatially, computing dot products"

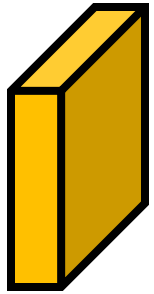
Convolutional Layer

Filters always extend the full depth of the input volume

$32 \times 32 \times 3$

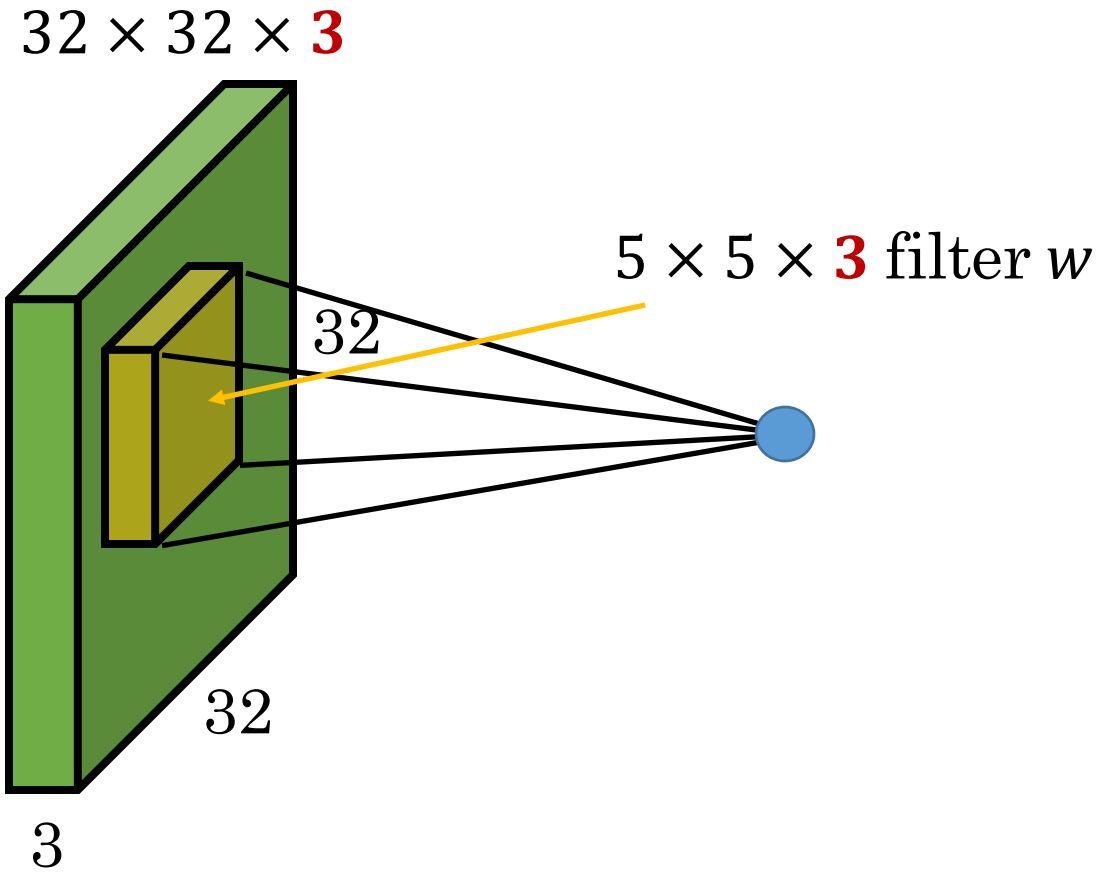


$5 \times 5 \times 3$ filter w



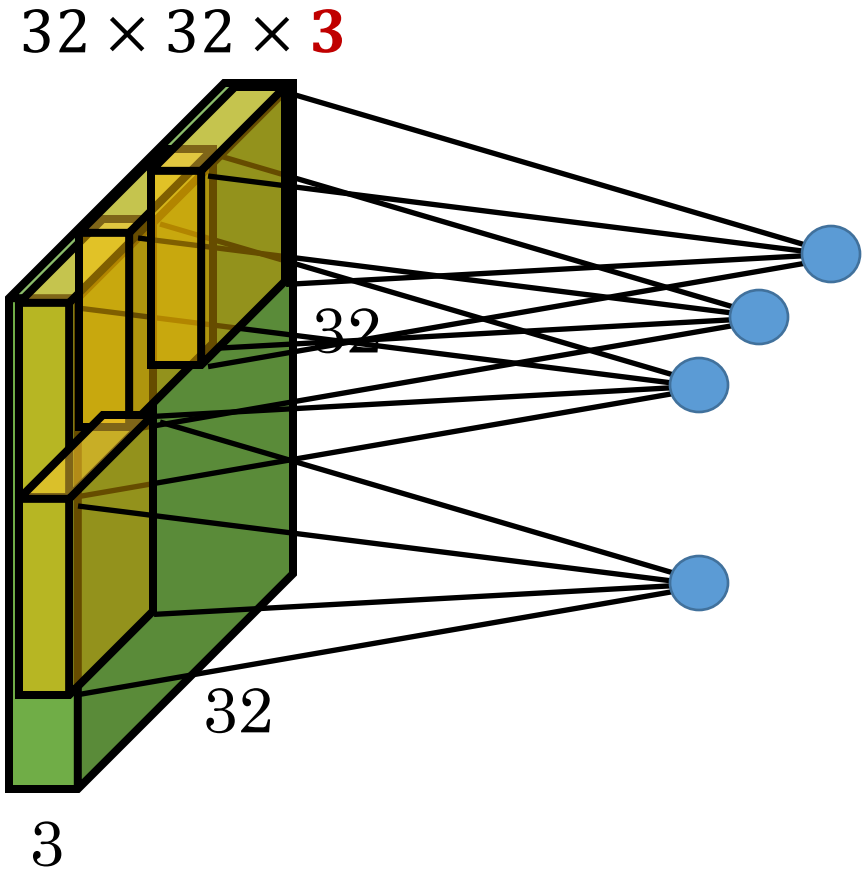
- Convolve the filter with the image, i.e., "slide over the image spatially, computing dot products"

Convolutional Layer

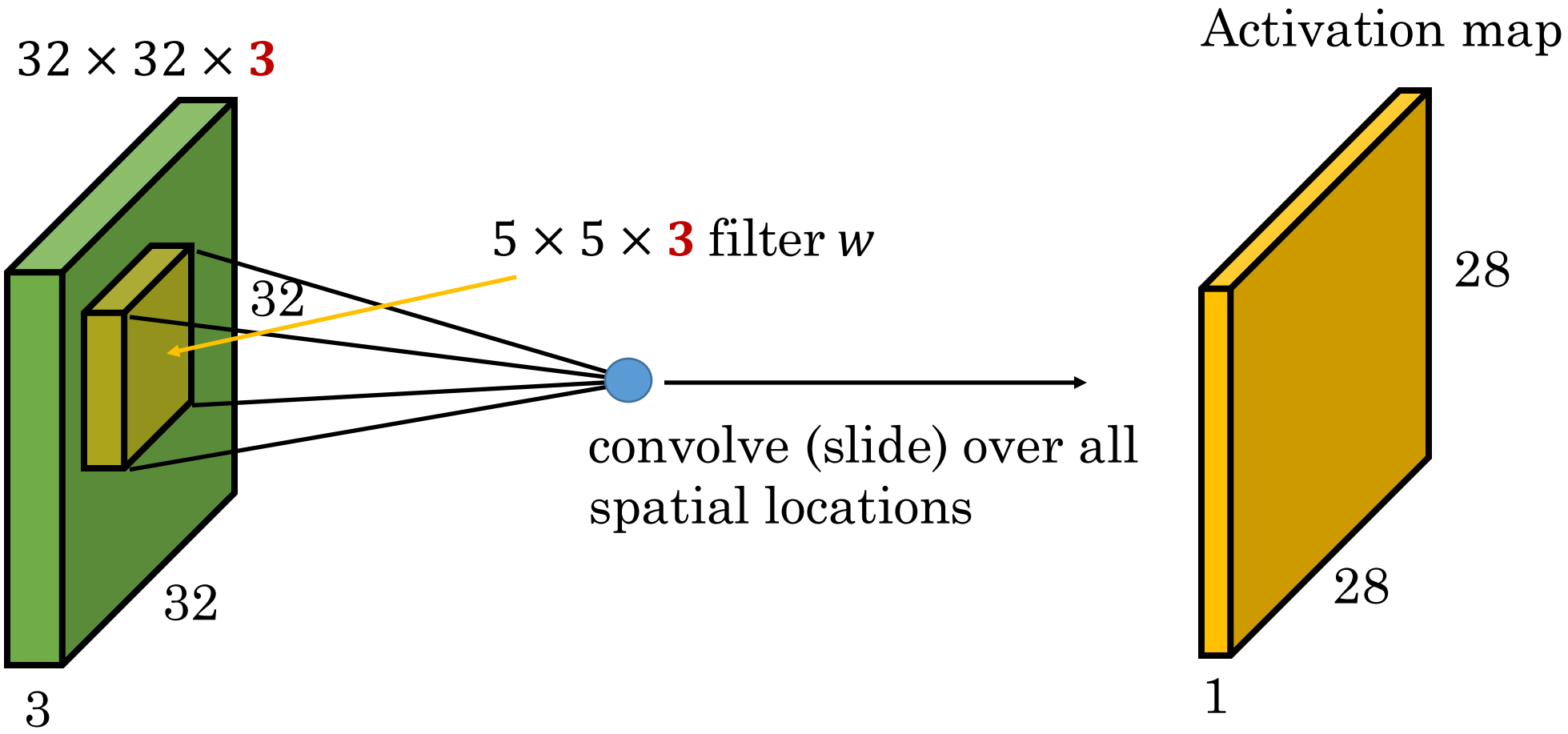


- **1** number: the result of taking a dot product between the filter and a small $5 \times 5 \times 3$ chunk of the image
- i.e., $5 \times 5 \times 3 = 75$-dimensional dot product + bias ($wx + b$)

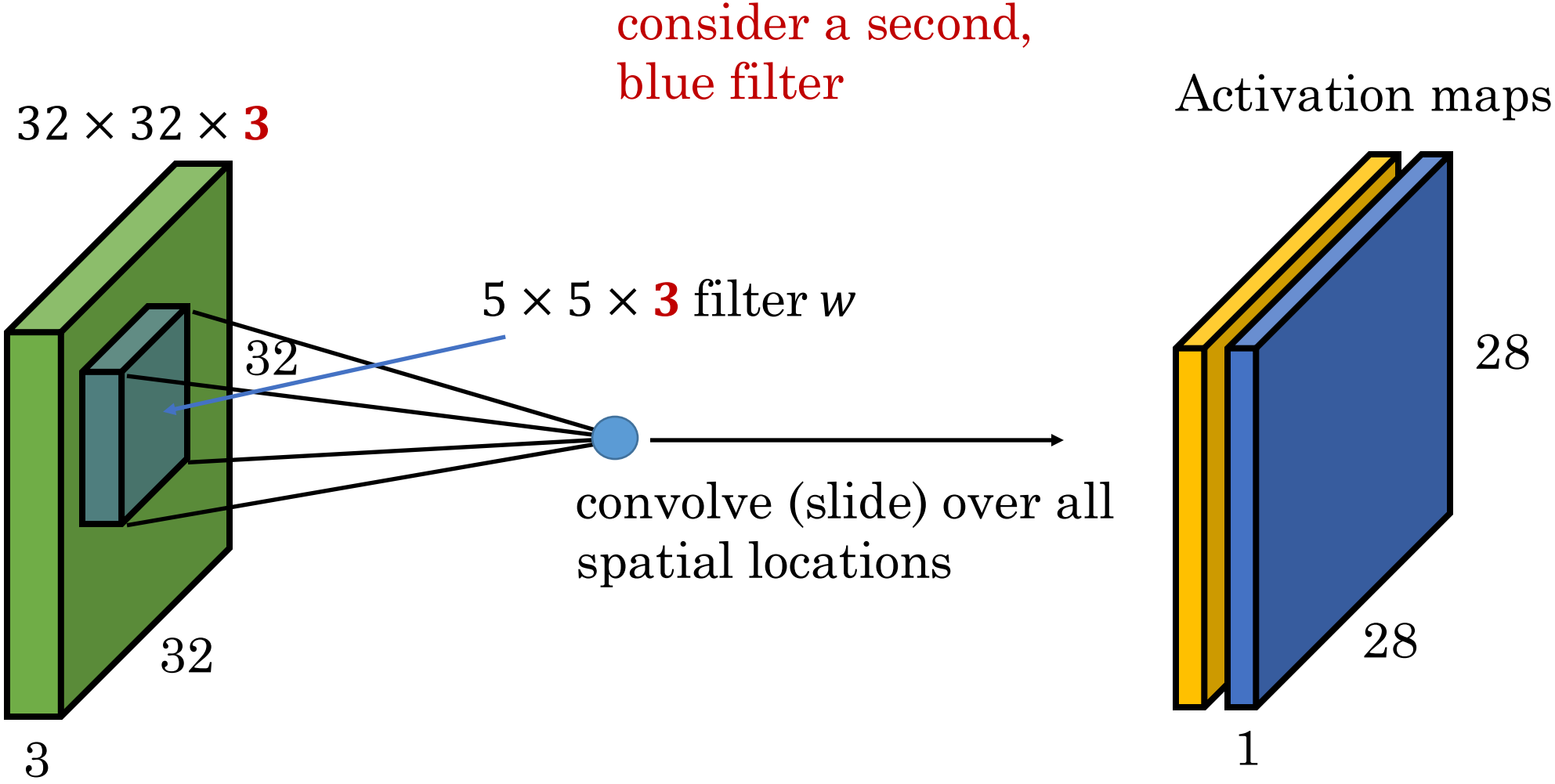
Convolutional Layer



Convolutional Layer

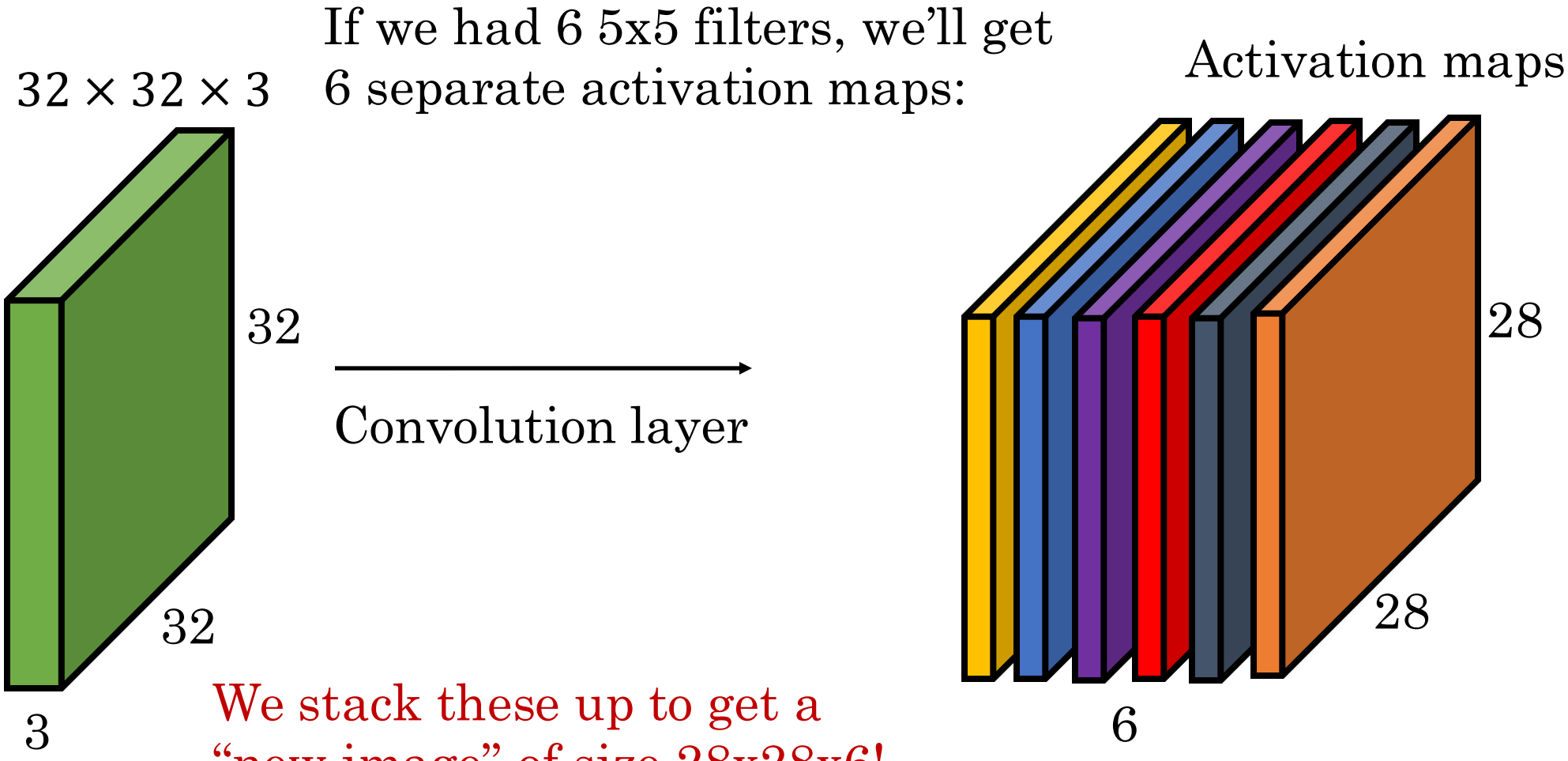


Convolutional Layer



Convolutional Layer

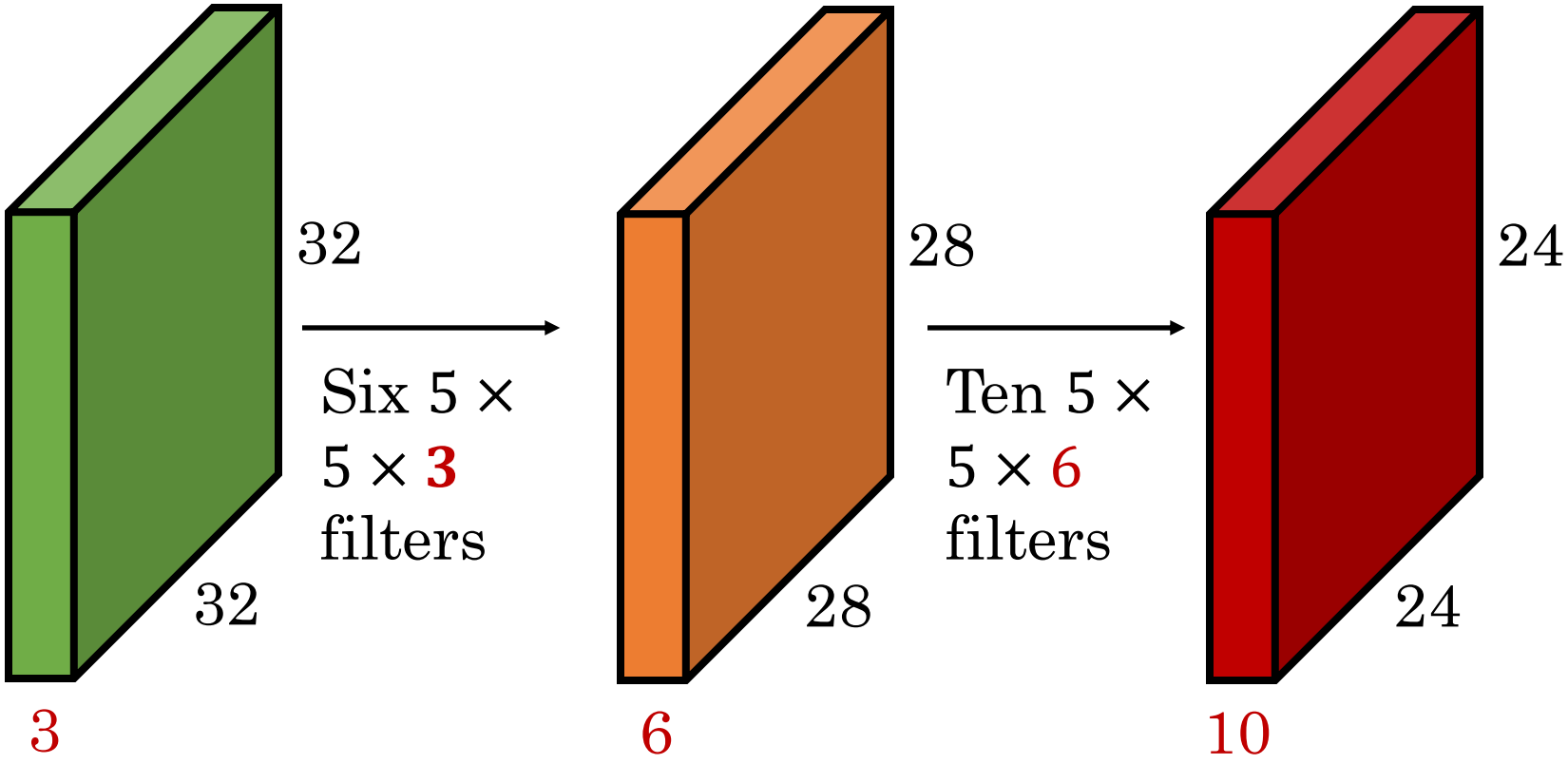
If we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

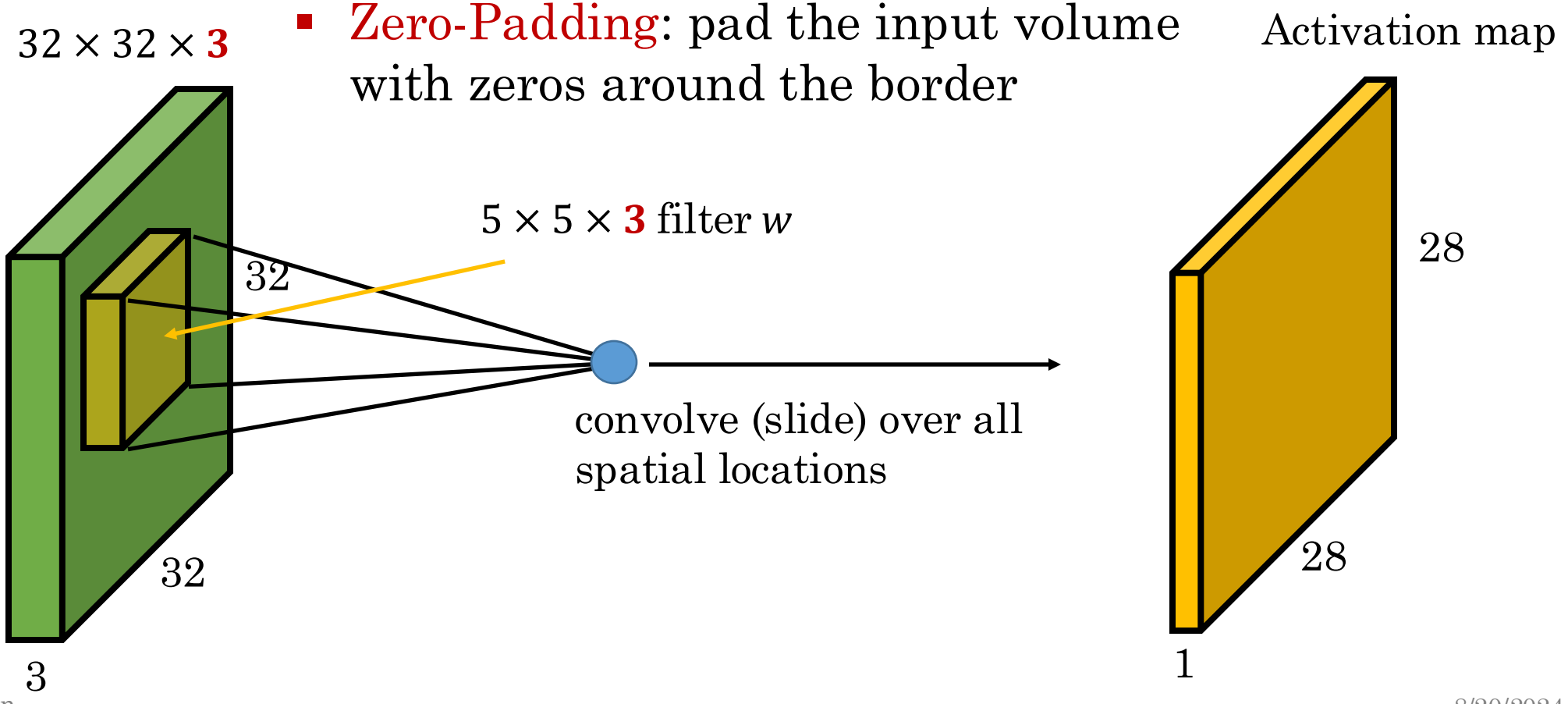
Convolutional Neural Networks

- ConvNet is a sequence of Convolution Layers, interspersed with activation functions



A Closer Look at Spatial Dimensions

- Stride: number of pixels with which we slide the filter
- Zero-Padding: pad the input volume with zeros around the border



Deep Learning for Graphs

Content

- Local network neighborhoods
  - Describe aggregation strategies
  - Define computational graphs

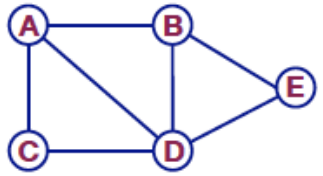
- Stacking multiple layers
  - Describe model, parameters, training
  - How to fit the model?
  - Simple examples for supervised and unsupervised training

Setup

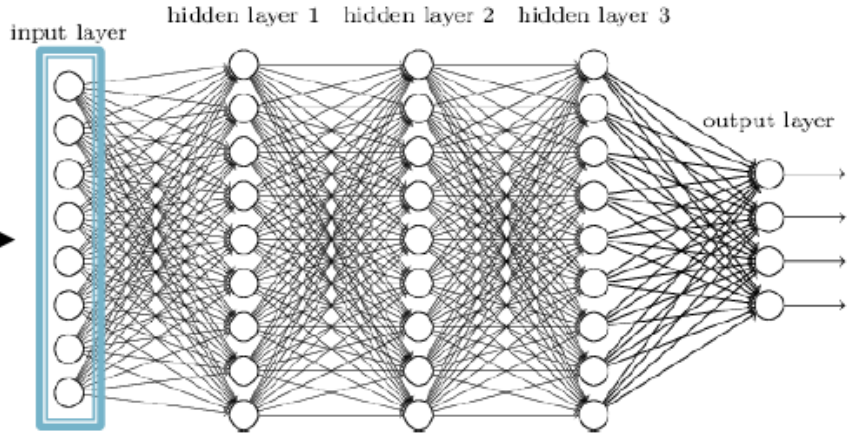
- A graph G
  - V: vertex set
  - A: adjacency matrix (assume binary)
  - $X \in \mathbb{R}^{|V| \times m}$: a matrix of node features
  - v: a node in V, $N(v)$: the neighbor set of v
- Node features:
  - Social networks: user profile, user images
  - Biological networks: gene expression profiles, gene functional information
  - When there is no node feature:
    - Indicator vectors (one-hot encoding of a node)
    - Vector of constant 1: $[1, 1, \cdots, 1]$

A Naïve Approach

- Join adjacency matrix and features
- Feed them into a deep neural network



	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

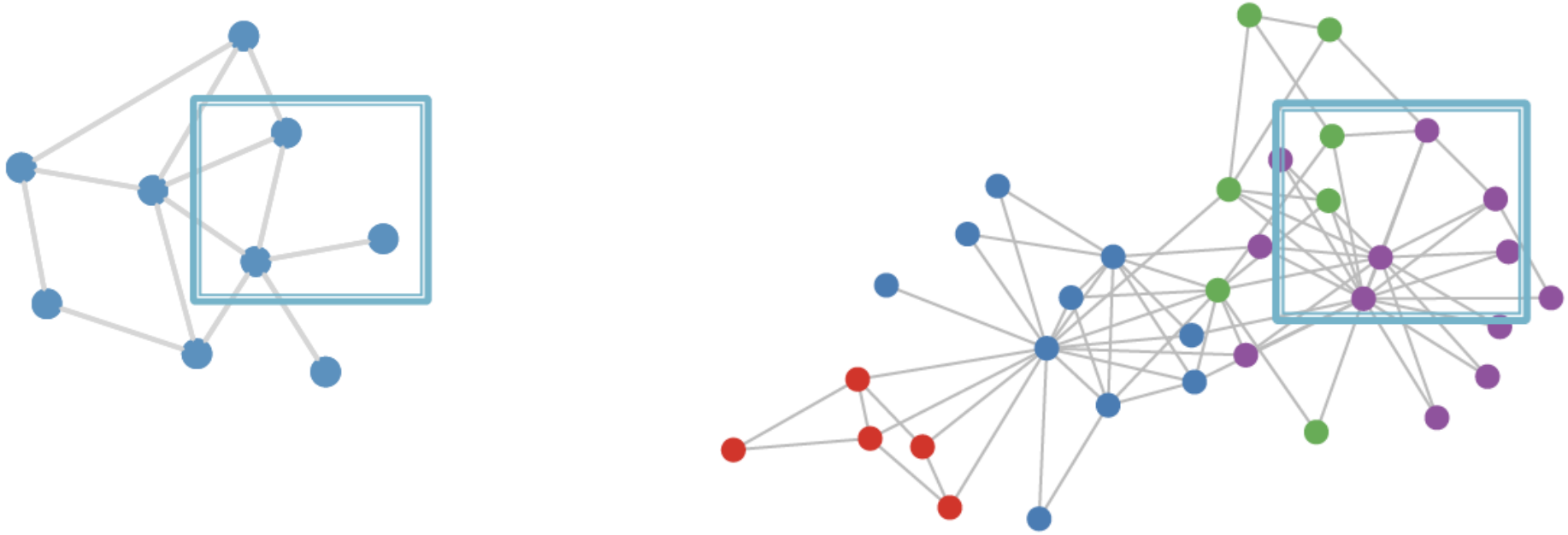


?

- Issues:
  - $O(|V|)$ parameters
  - Not applicable to graphs of different sizes
  - Sensitive to node orderings

How About Convolutional Neural Nets?

Real-world Graphs



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

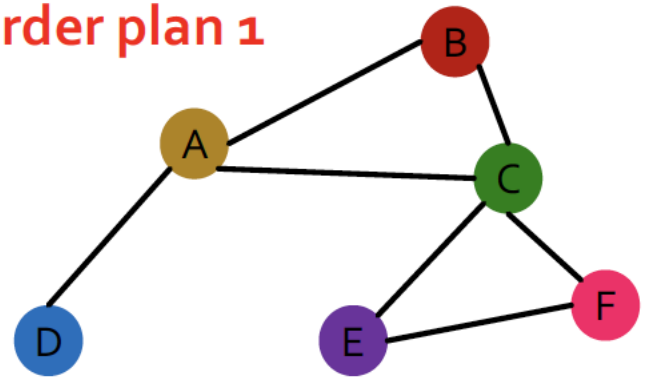
Permutation Invariance

- Graph does not have a canonical order of the nodes

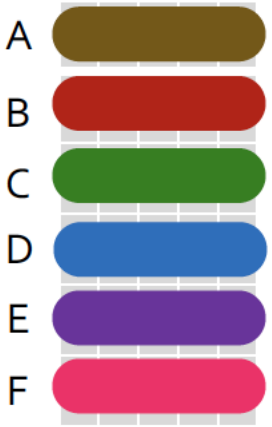
- We can have many different order plans

Permutation Invariance

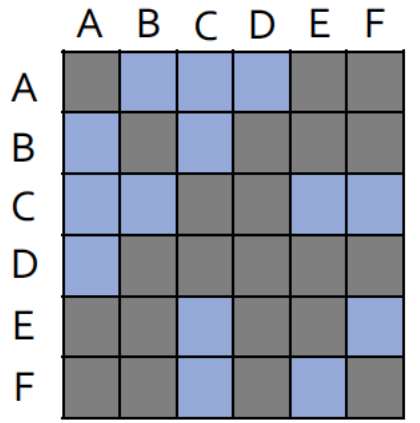
Order plan 1



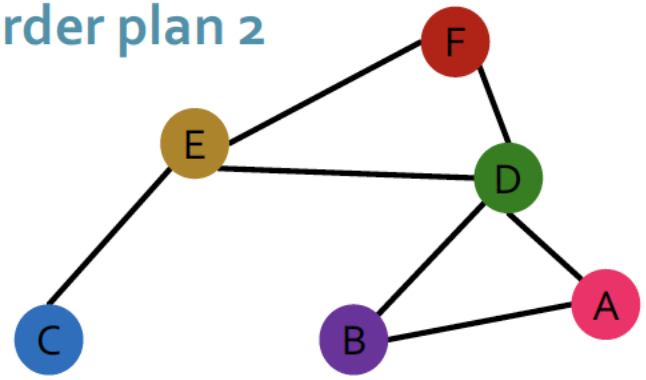
Node features X_1



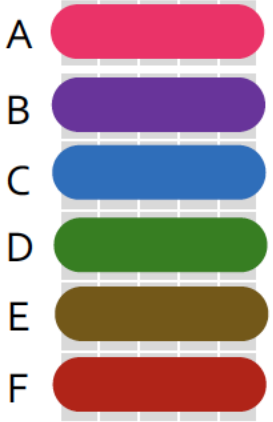
Adjacency matrix A_1



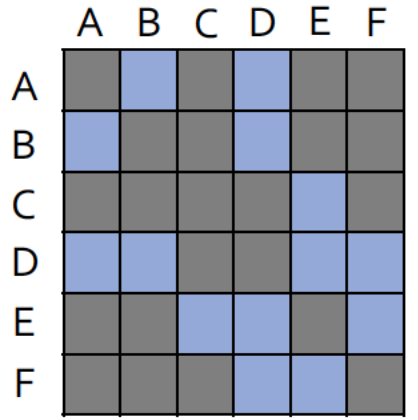
Order plan 2



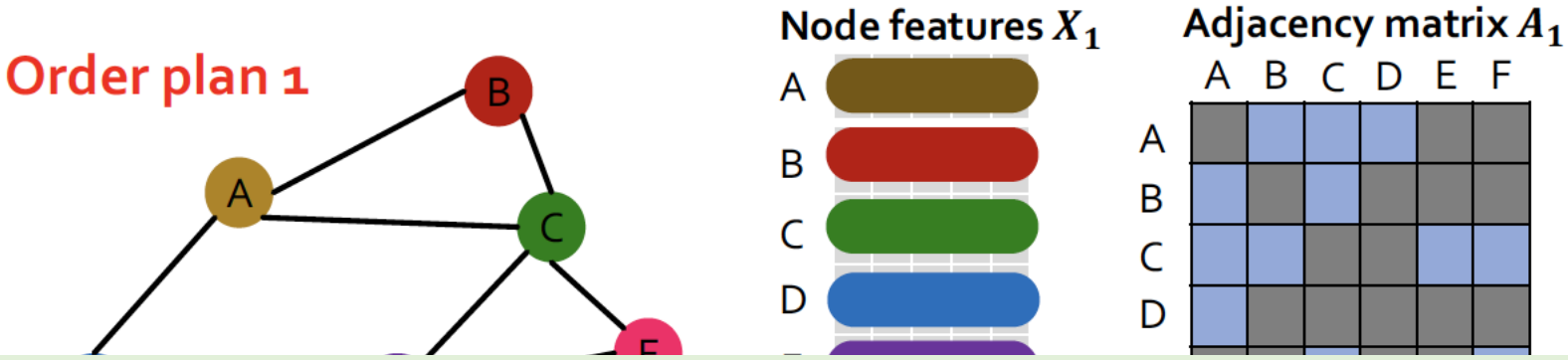
Node features X_2



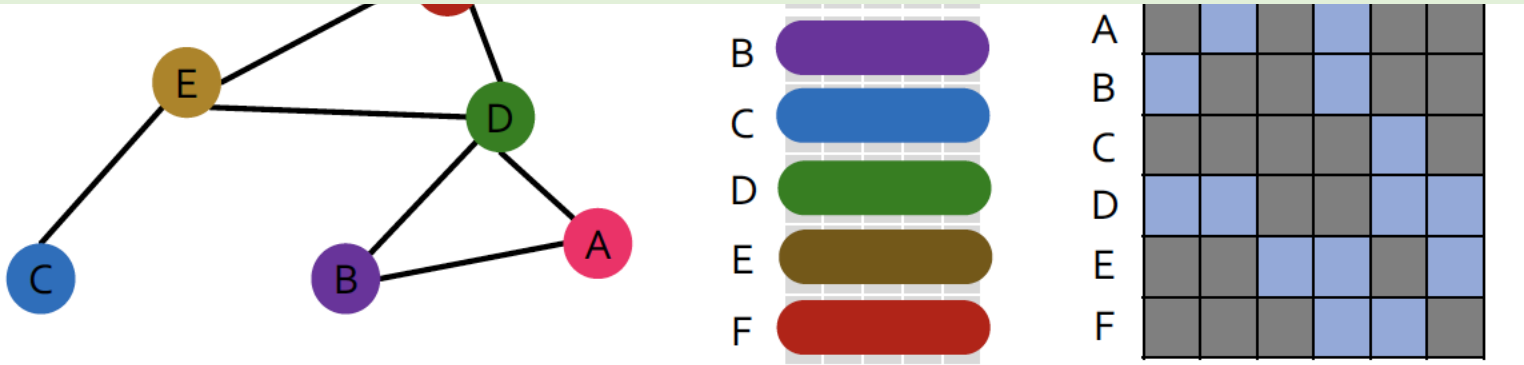
Adjacency matrix A_2



Permutation Invariance



Graph and node representations should be the same for Order plan 1 and Order plan 2

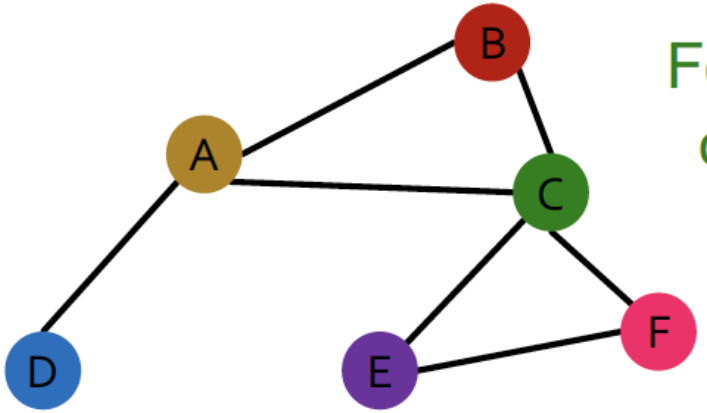


Permutation Invariance

- Representation: given a graph $G = (A, X)$ where A is adjacency matrix and X is node feature matrix, then:

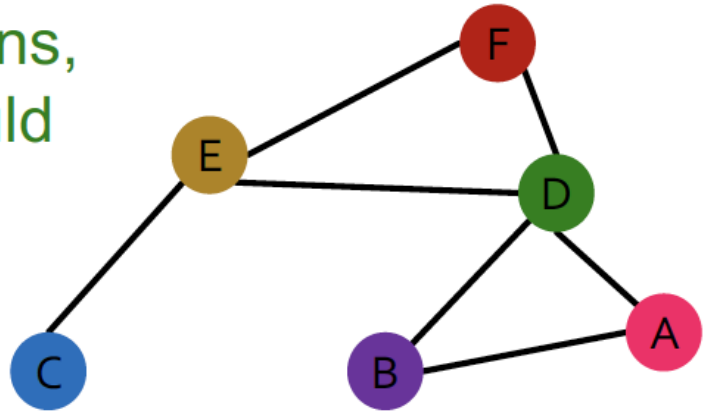
$$f(A_1, X_1) = f(A_2, X_2)$$

Order plan 1: A_1, X_1



For two order plans, output of f should be the same!

Order plan 2: A_2, X_2



Permutation Invariance

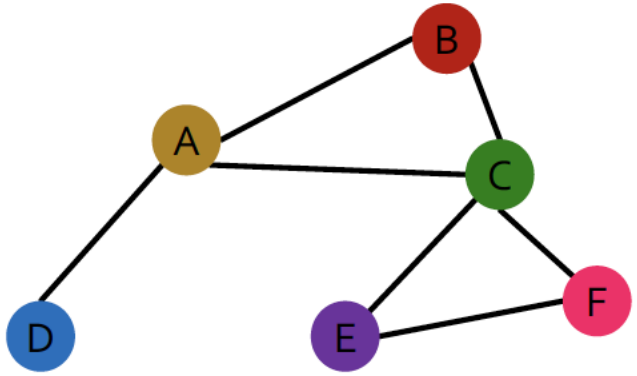
- Consider we learn a function f that maps a graph $G = (A, X)$ to a vector $\mathbb{R}^d$
- If $f(A_i, X_i) = f(A_j, X_j)$ for any order plan i and j, we formally say f is permutation invariant.

- Definition: For any graph function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^d$, f is permutation invariant if $f(A, X) = f(PAP^T, PX)$ for any permutation P
  - Permutation P: a shuffle of the node order

Permutation Equivariance

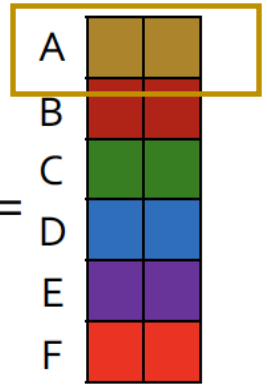
- For node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{|V|\times d}$

Order plan 1: A_1, X_1



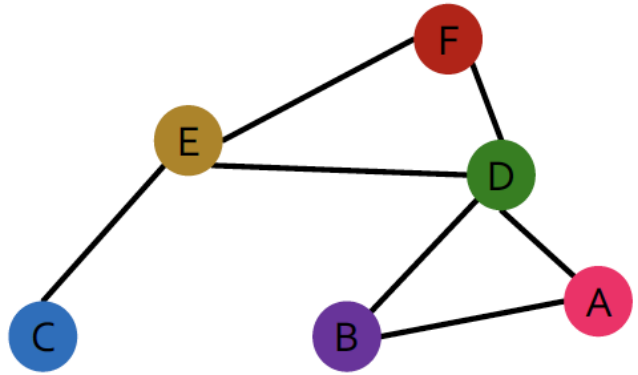
Representation vector of brown node A

$f(\boldsymbol{A}_1, \boldsymbol{X}_1) =$

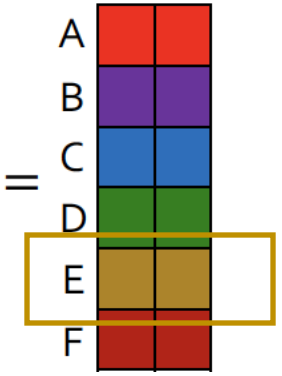


For two order plans, the vector of node at the same position in the graph is the same

Order plan 2: A_2, X_2



$f(\boldsymbol{A}_2, \boldsymbol{X}_2) =$



Representation vector of brown node E

Permutation Equivariant

- For node representation
  - Consider we learn a function f that maps a graph $G = (A, X)$ to a vector $\mathbb{R}^d$
  - If the output vector of a node at the same position in the graph remains unchanged for any order plan, we say f is permutation equivariant.

- Definition: For any node function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \to \mathbb{R}^{|V| \times d}$, f is permutation-equivariant if $Pf(A, X) = f(PAP^T, PX)$ for any permutation P

Summary: Invariance and Equivariance

- Permutation invariant
  - Permute the input, the output stay the same

$$f(A, X) = f(PAP^T, PX)$$

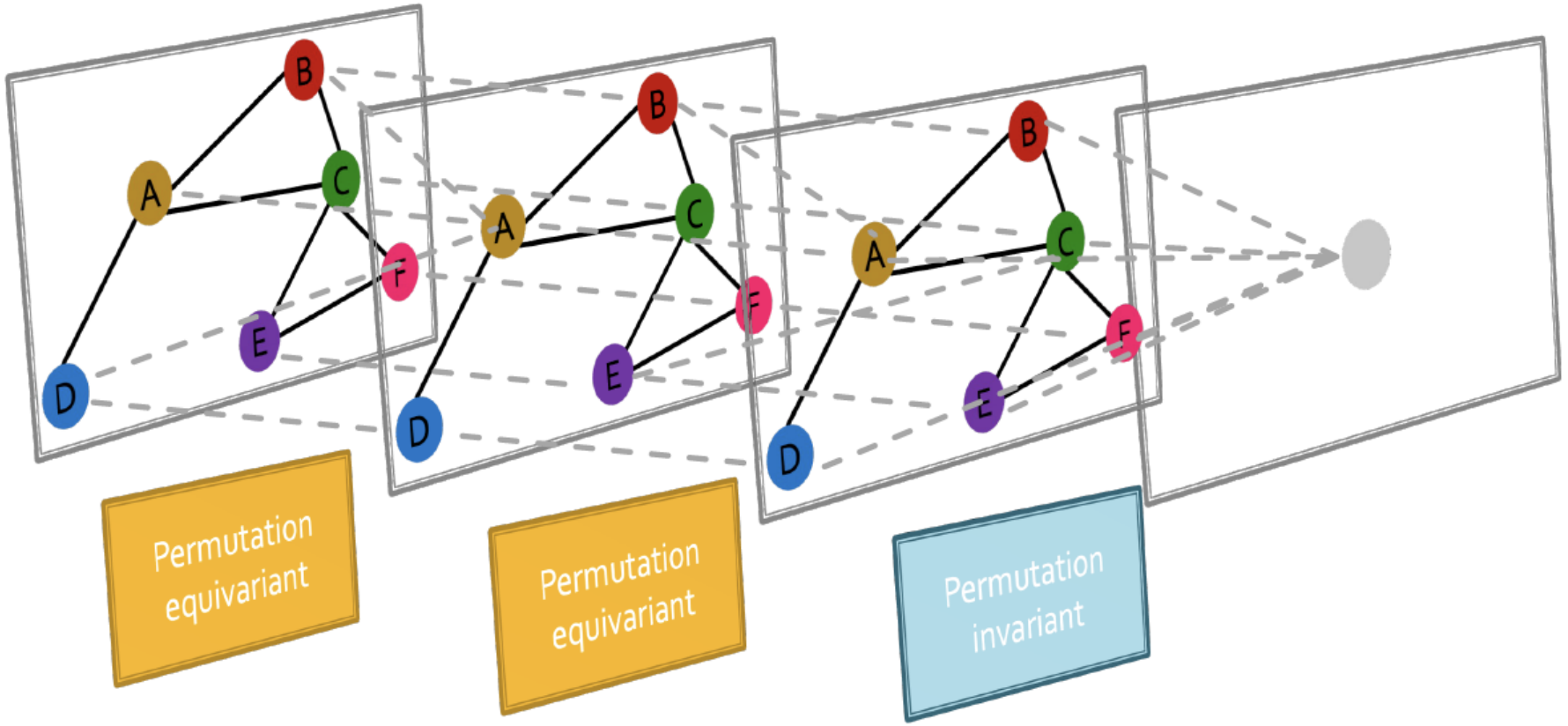
- Permutation equivariant
  - Permute the input, output also permutes accordingly

$$Pf(A, X) = f(PAP^T, PX)$$

- Example:
  - $f(A, X) = 1^T X$: permutation invariant
  - $f(A, X) = X$: permutation equivariant
  - $f(A, X) = AX$: permutation equivariant

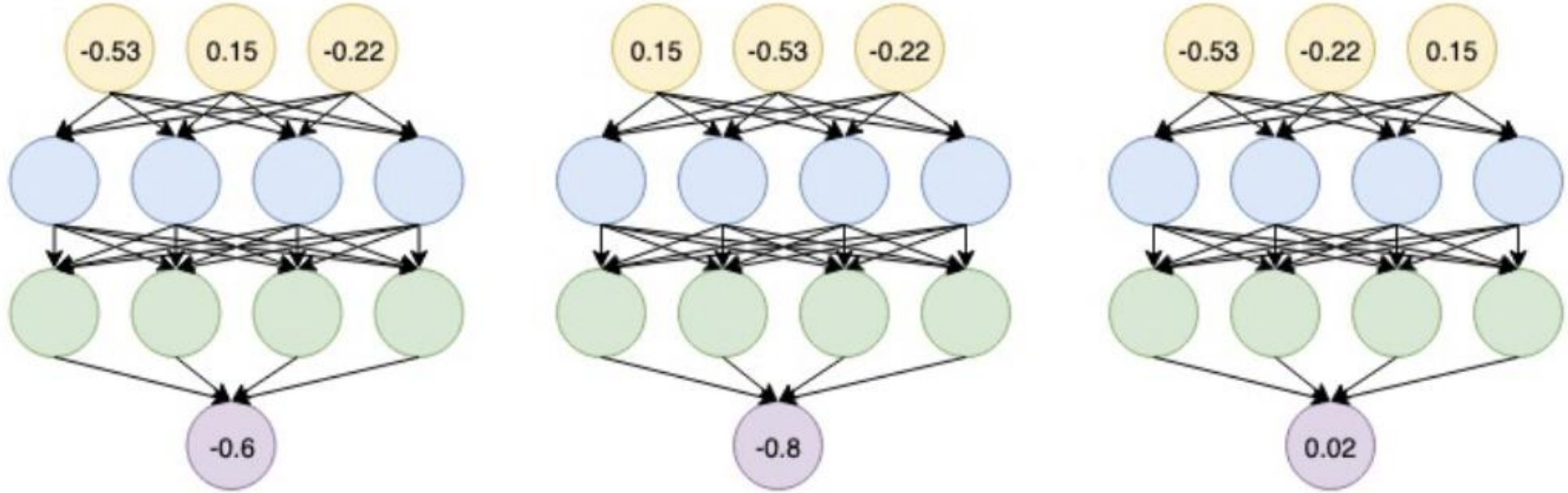
Overview: Graph Neural Networks

- GNNs consist of multiple permutation equivariant / invariant functions



Overview: Graph Neural Networks

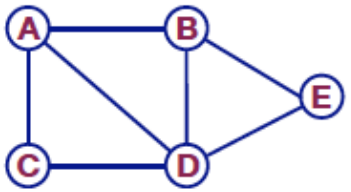
- Are other neural net architectures, e.g., MLPs, permutation invariant/equivariant?
  - No



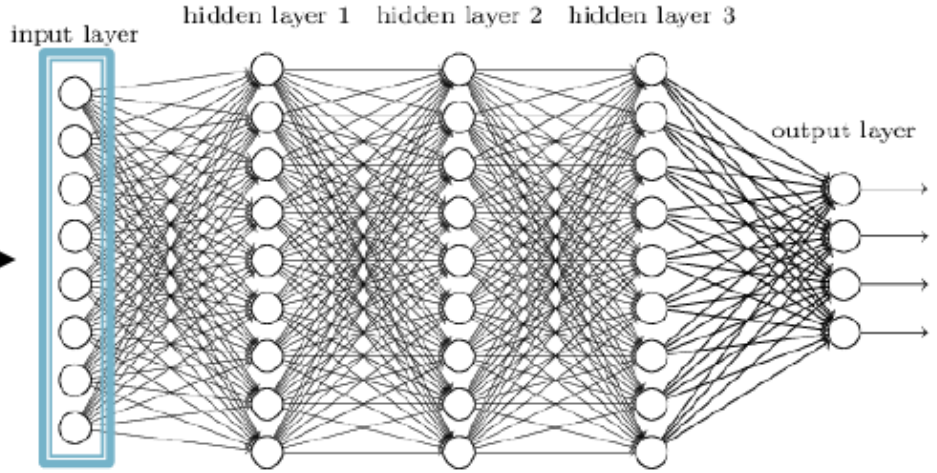
Switching order of the input leads to different outputs

Overview: Graph Neural Networks

- Are other neural net architectures, e.g., MLPs, permutation invariant/equivariant?
  - No



	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



?

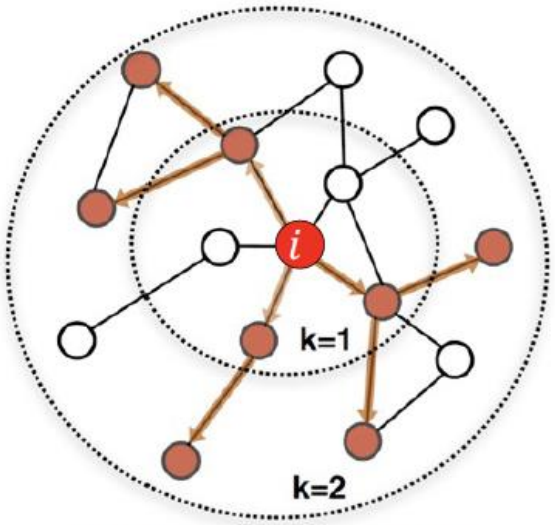
That's why naïve MLP approach fails for graphs

Overview: Graph Neural Networks

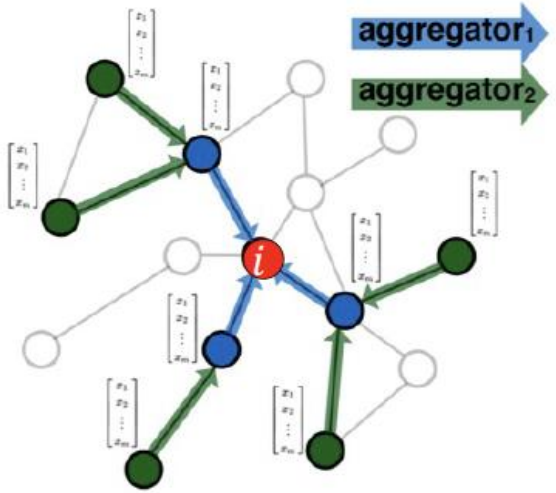
Next: Design graph neural networks that are permutation invariant / equivariant by passing and aggregating information from neighbors

Graph Convolutional Networks

Idea: Node's neighborhood defines a computational graph



Determine node computation graph

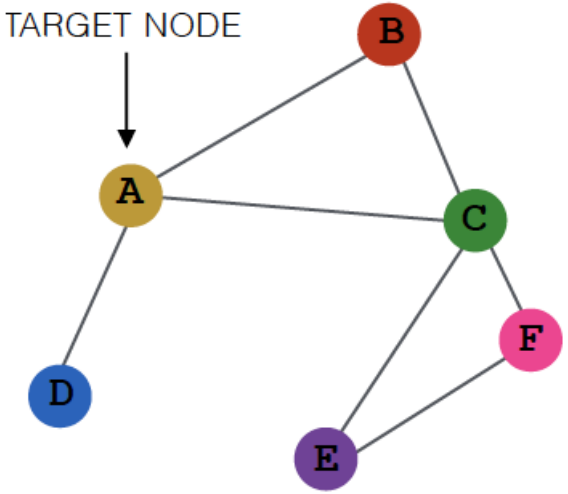


Propagate and transform information

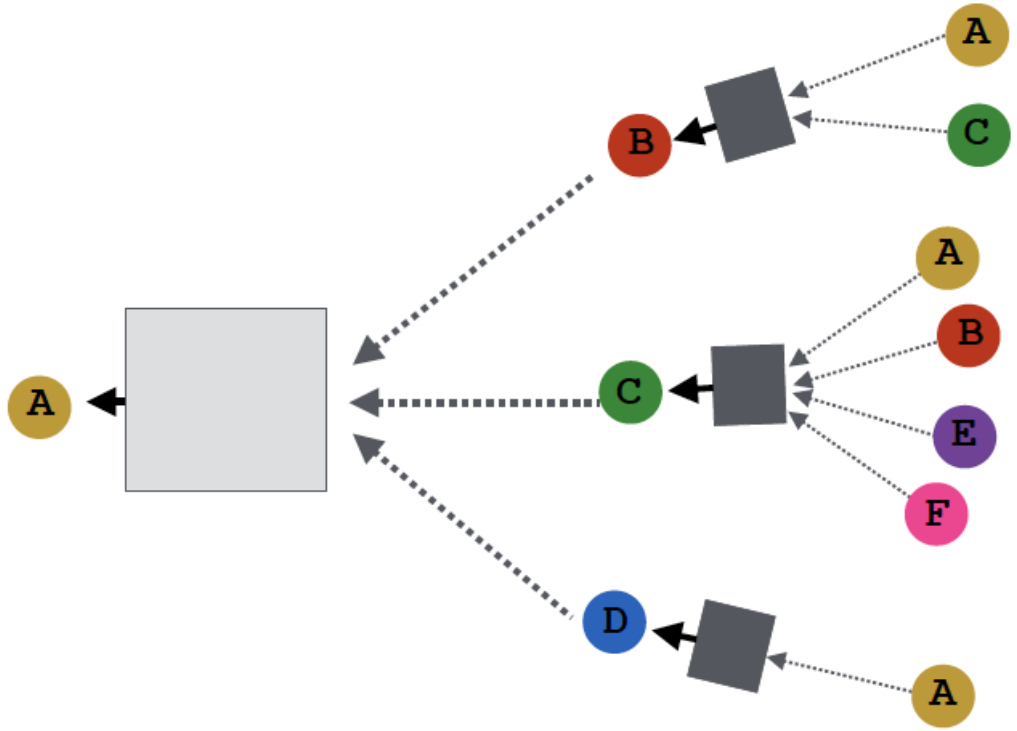
Learn how to propagate information across the graph to compute node features

Idea: Aggregate Neighbors

- Key idea: Generate node embeddings based on local network neighborhoods

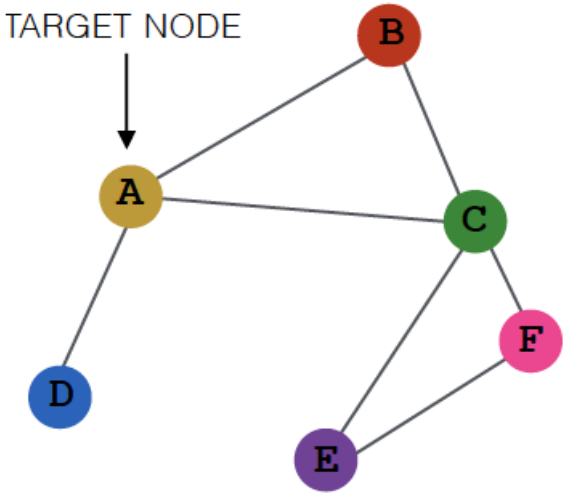


INPUT GRAPH

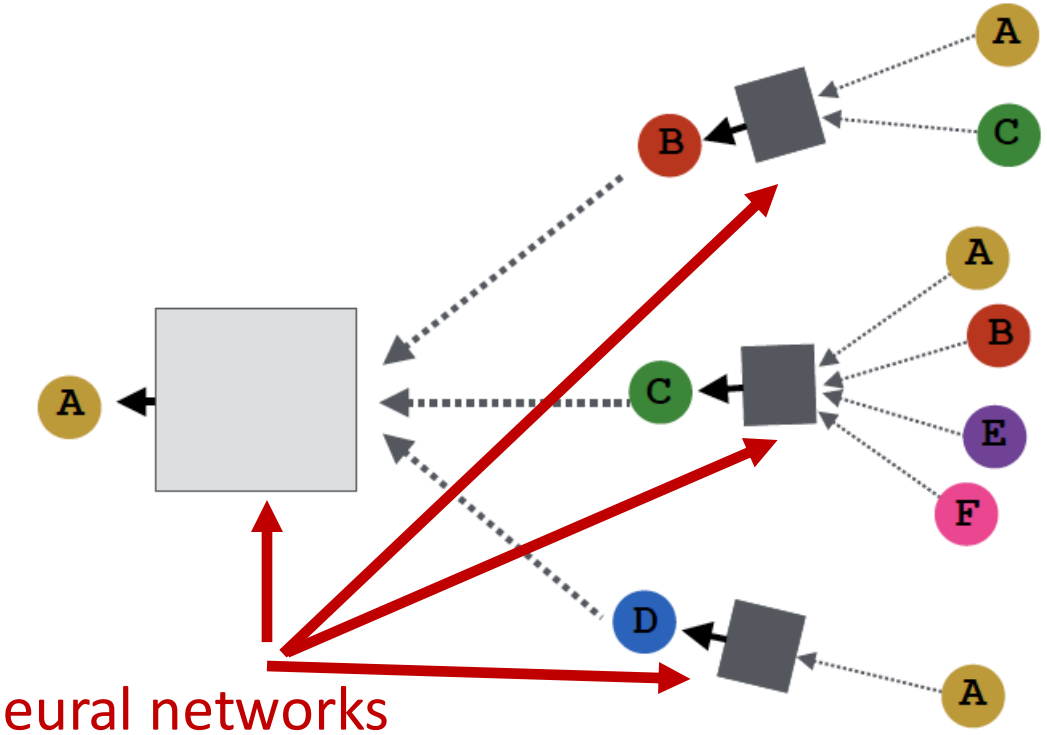


Idea: Aggregate Neighbors

- Intuition: Node aggregate information from their neighbors using neural networks

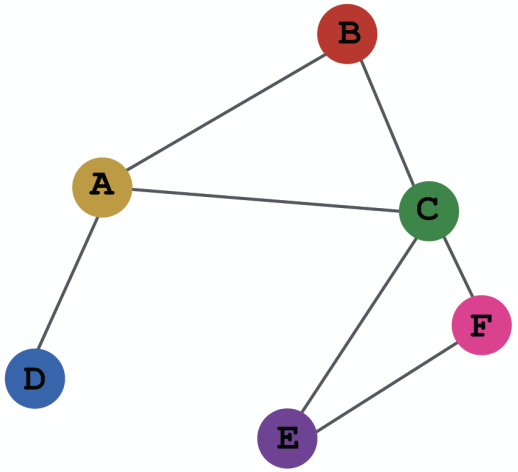


INPUT GRAPH



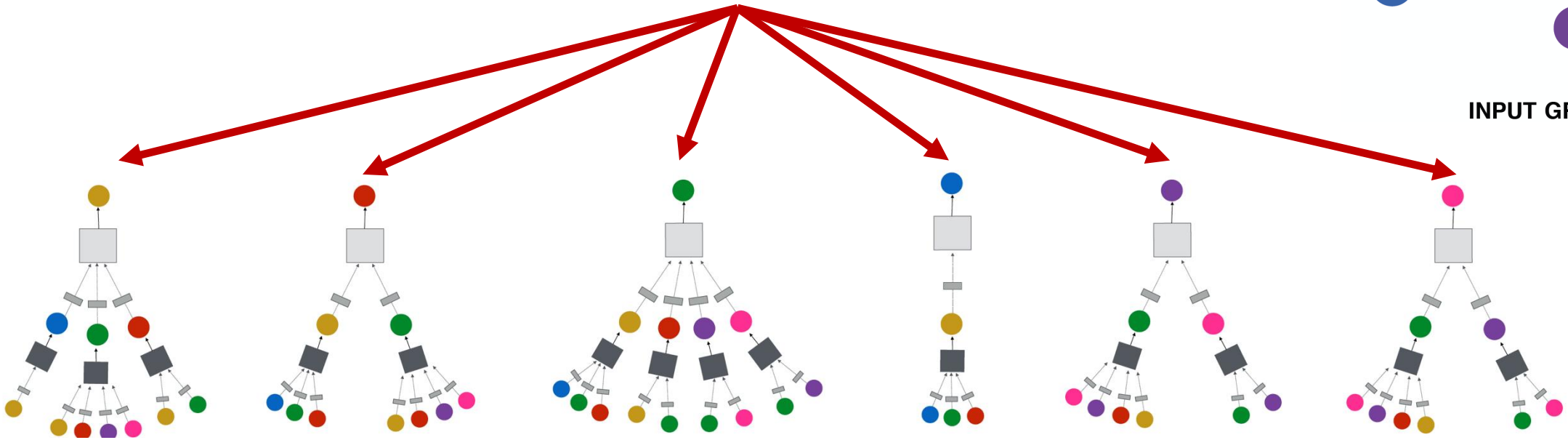
Idea: Aggregate Neighbors

- Intuition: Network neighborhood defines a computational graph



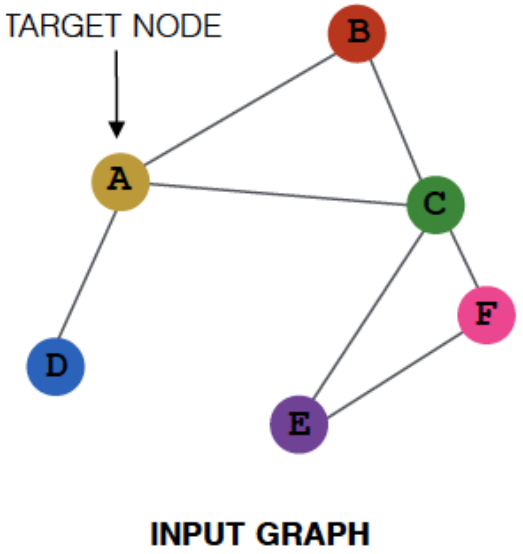
INPUT GRAPH

Every node defines a computational graph based on its neighborhood

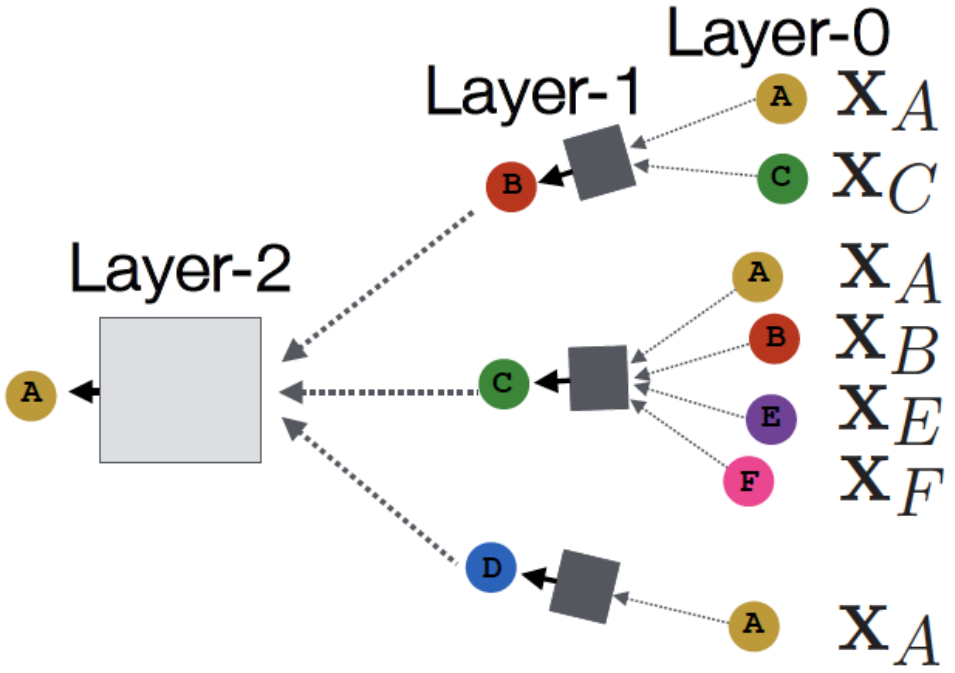


Deep Model: Many Layers

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

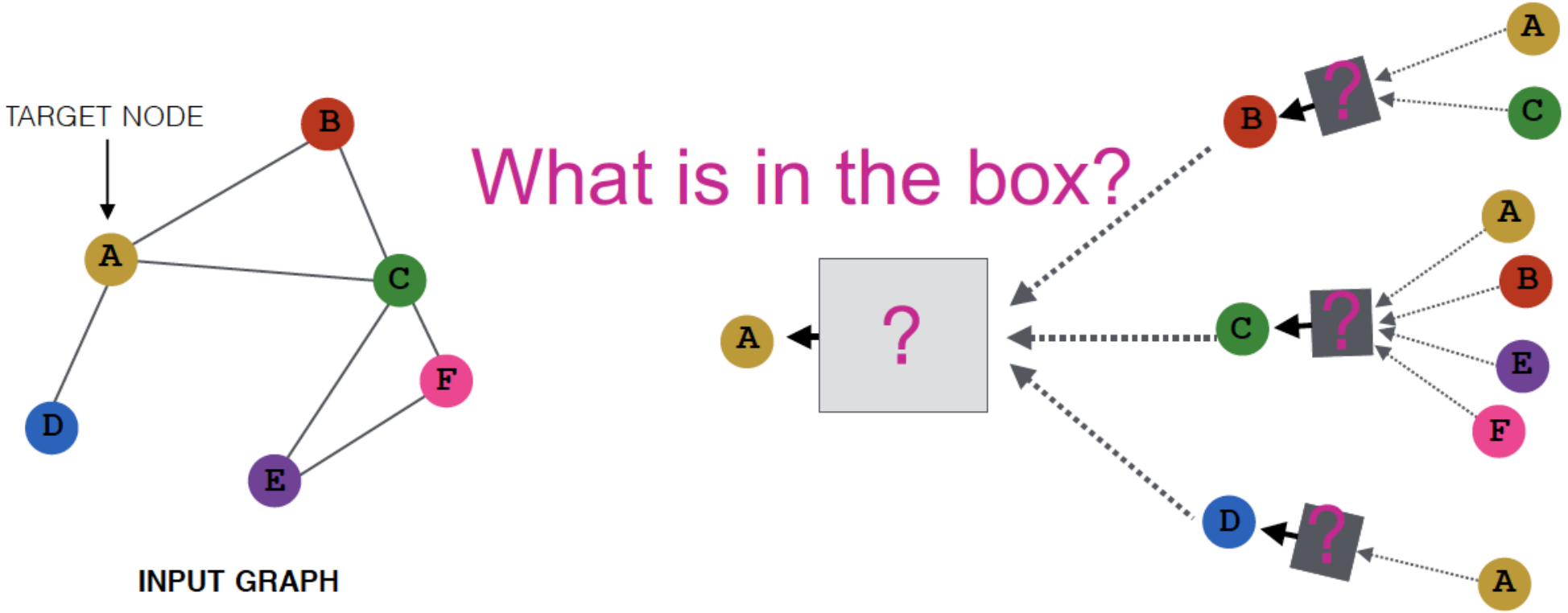


INPUT GRAPH



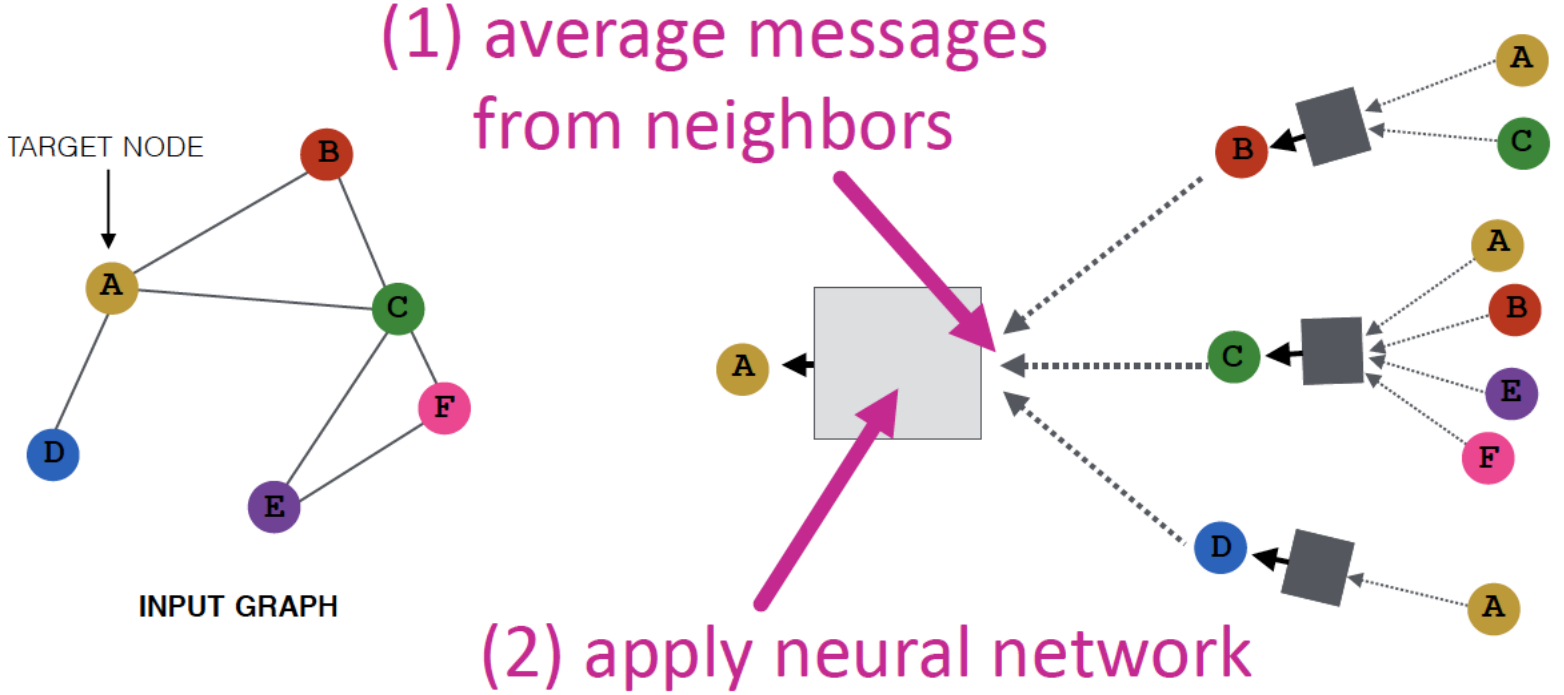
Neighborhood Aggregation

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



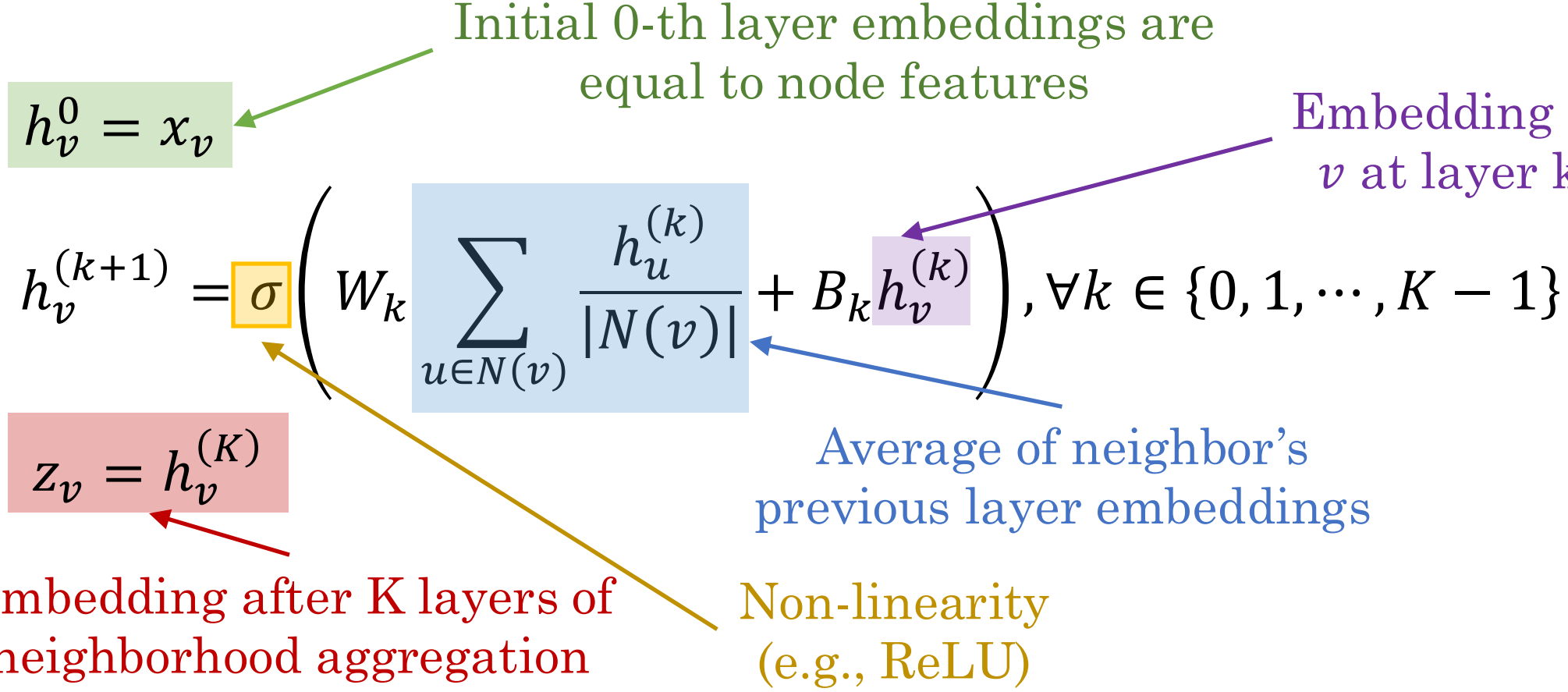
Neighborhood Aggregation

- Basic approach: average information from neighbors and apply a neural network



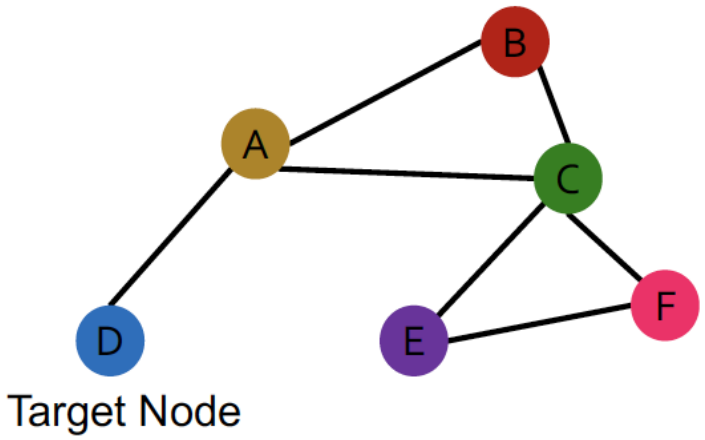
The Math: Deep Encoder

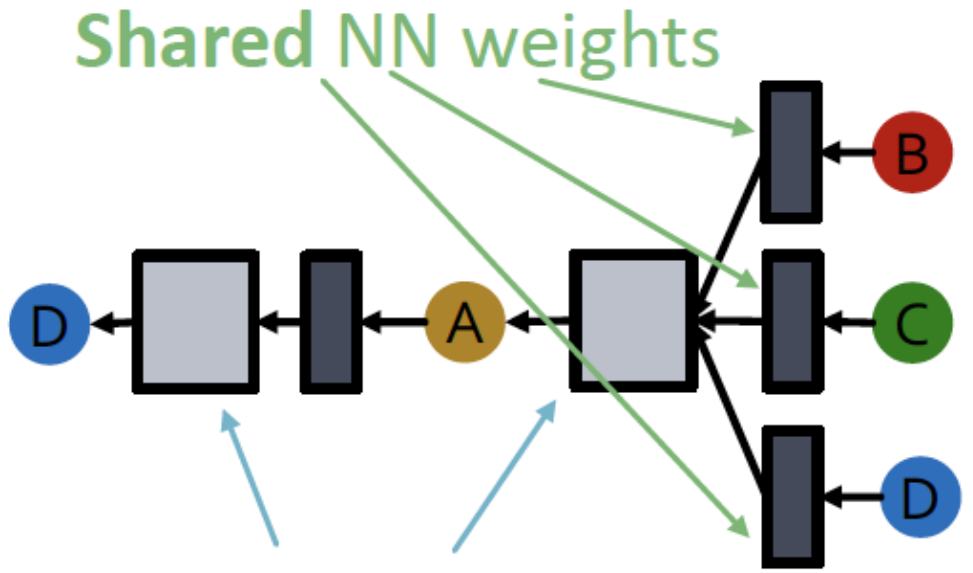
- Basic approach: average information from neighbors and apply a neural network



GCN: Invariance and Equivariance

- What are the invariance and equivariance properties for a GCN?
  - Given a node, GCN that computes its embedding is permute invariant

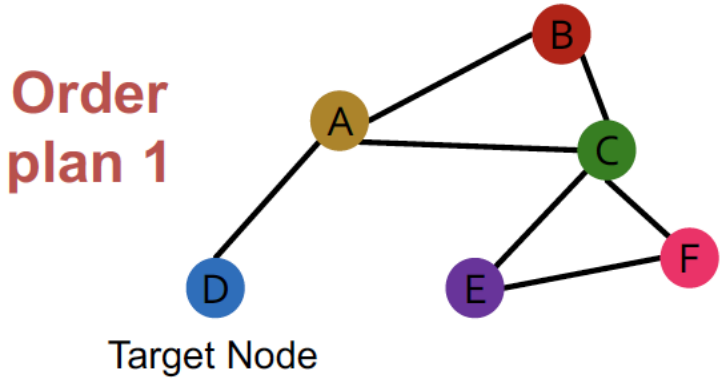


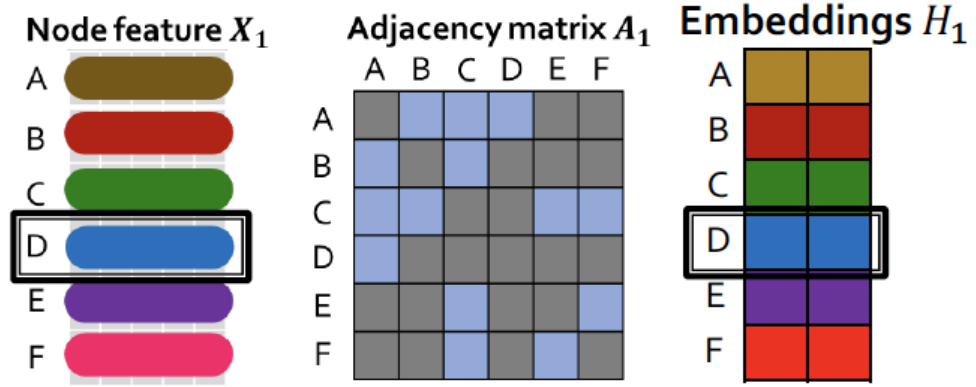


Average of neighbor's previous layer embeddings - **Permutation invariant**

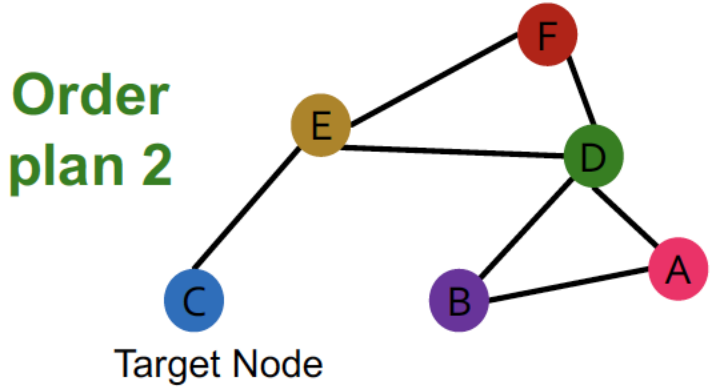
GCN: Invariance and Equivariance

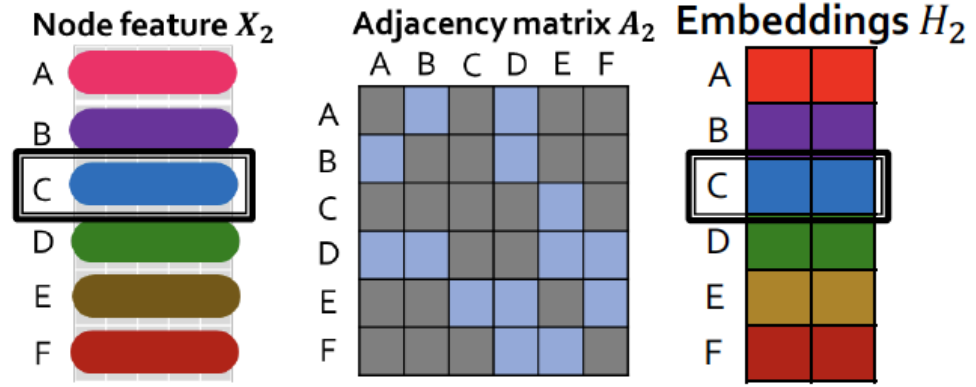
- Consider all nodes in a graph, GCN computation is permutation equivariance.





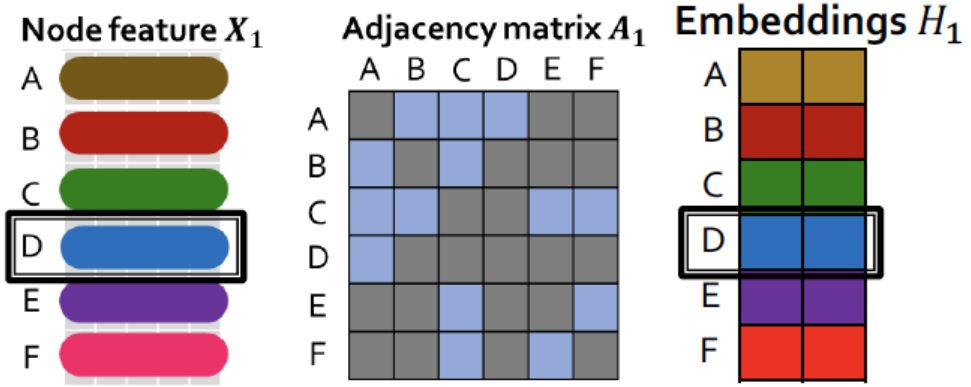
Permute the input, the output also permutes accordingly – permutation equivariant





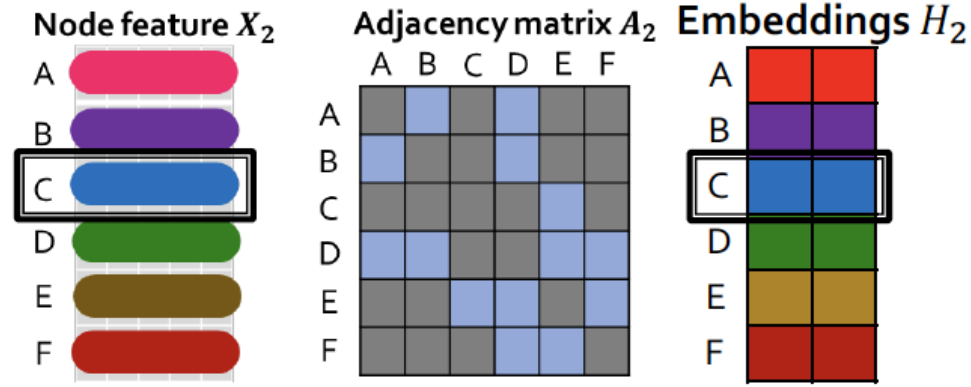
GCN: Invariance and Equivariance

- Consider all nodes in a graph, GCN computation is permutation equivariance.



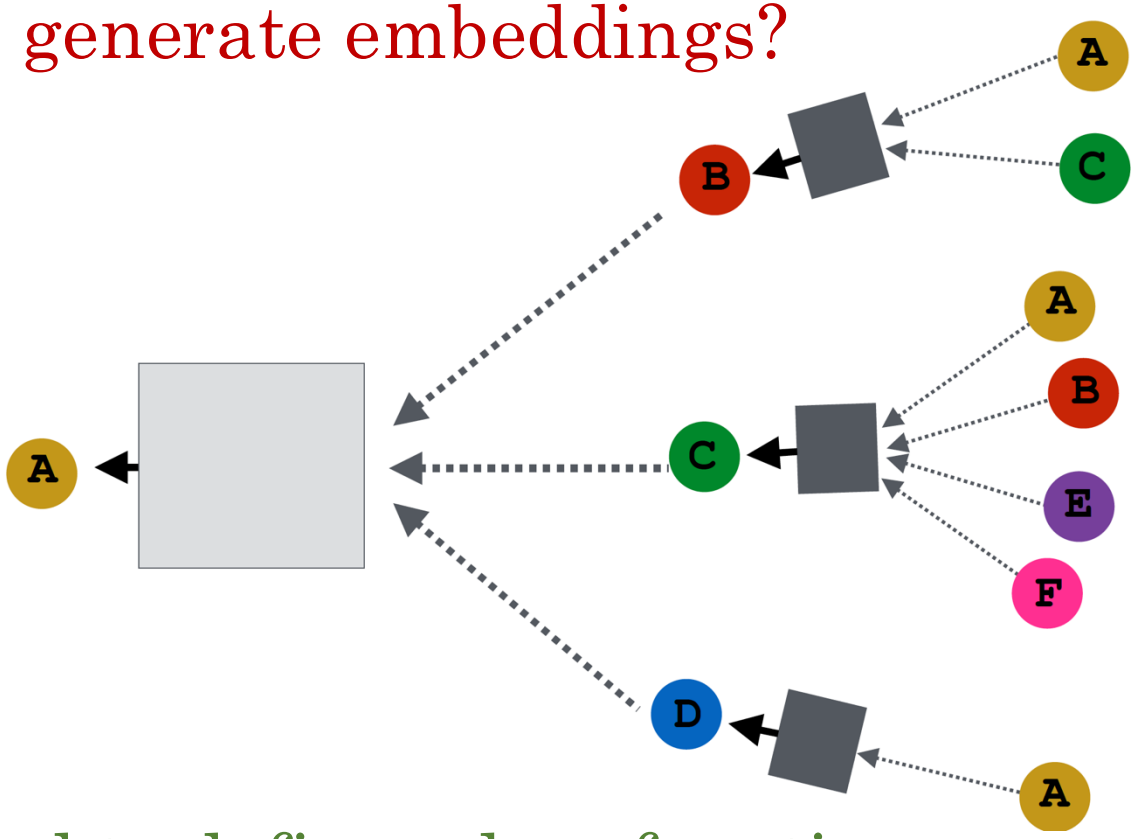
- The rows of input node features and output embeddings are aligned
- We know computing the embedding of a given node with GCN is invariant.
- So, after permutation, the location of a given node in the input node feature matrix is changed, and the output embedding of a given node stays the same (the colors of node feature and embedding are matched)

Permute the input, the output also permutes accordingly – permutation equivariant



Training the Model

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings

Model Parameters

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

$$h_v^0 = x_v$$

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

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

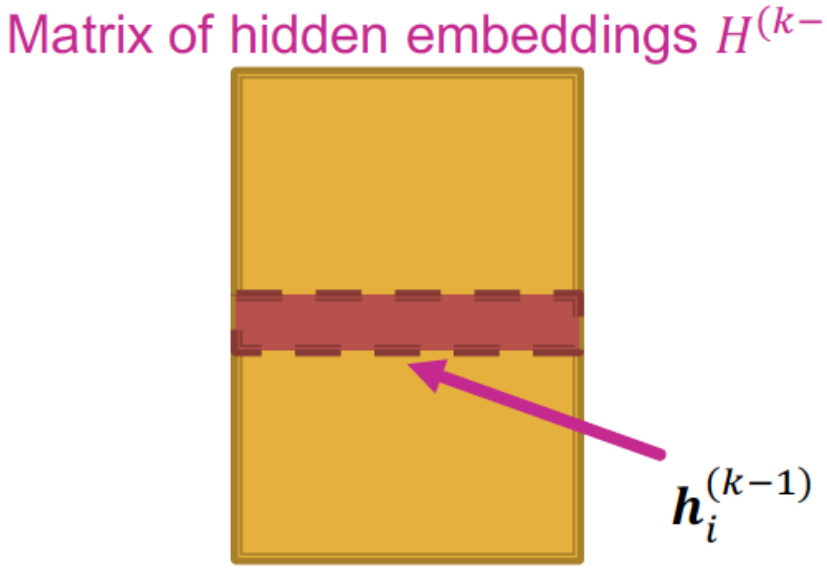
We can feed these embeddings into any loss function and run stochastic gradient descent to train them

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

Matrix Formulation (1)

- Many aggregation can be performed efficiently by (sparse) matrix operations
- Let $H^{(k)} = \left[h_1^{(k)}, h_2^{(k)}, \cdots, h_{|V|}^{(k)}\right]^T$
- Then: $\sum_{u \in N(v)} h_u^{(k)} = A_{v,:} H^{(k)}$
- Let D be diagonal degree matrix
  - $D_{v,v} = Deg(v) = |N(v)|$
  - The inverse of D is also diagonal: $D_{v,v}^{-1} = \frac{1}{|N(v)|}$
- Thus,

$$H^{(k+1)} = D^{-1} A H^{(k)}$$

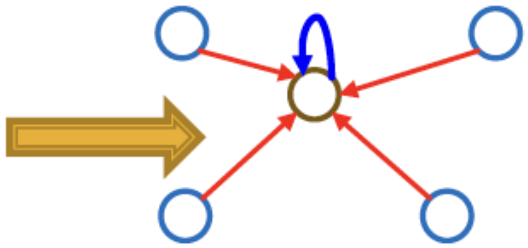


Matrix Formulation (2)

- Rewriting update function in matrix form

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

  - Where $\tilde{A} = D^{-1}A$
  - Red: neighborhood aggregation
  - Blue: self transformation



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

- In practice, this implies that efficient sparse matrix multiplication can be used ($\tilde{A}$ is sparse)
- Note: not all GNNs can be expressed in a simple matrix form, when aggregation function is complex

How to Train a GNN

- Node embedding z_v is a function of input graph
- Supervised setting: We want to minimize loss $\mathcal{L}$:

$$\min_{\Theta} \mathcal{L}\left(y, f_{\Theta}(z_v)\right)$$

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

Unsupervised Training

- One possible idea: "Similar" nodes have similar embeddings.

$$\min_{\Theta} \mathcal{L} = \sum_{z_u, z_v} CE\left(y_{u,v}, DEC(z_u, z_v)\right)$$

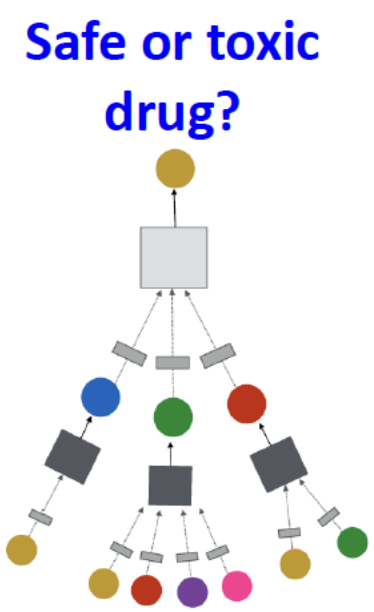
  - Where $y_{u,v} = 1$ when node u and v are similar
  - $z_u = f_\Theta(u)$ and $DEC(\cdot,\cdot)$ is the dot product
- CE is the cross-entropy loss

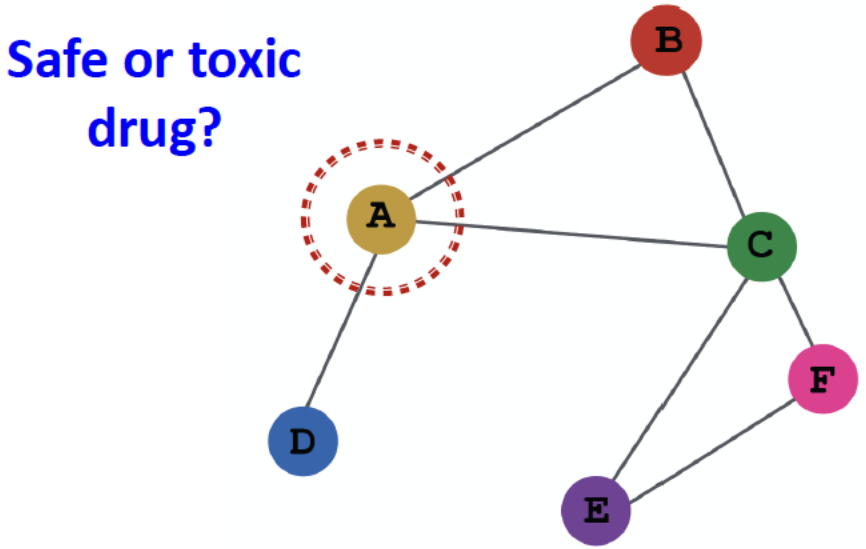
$$CE\big(y, f(x)\big) = -\sum_{i=1}^{C} (y_i \log f_\Theta(x)_i)$$

  - y_i and $f_\Theta(x)_i$ are the actual and predicted values of the i-th class
  - Intuition: the lower the loss, the closer the prediction to one-hot
- Node similarity:
  - Random walks (node2vec, DeepWalk, struc2vec)
  - Matrix factorization

Supervised Training

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

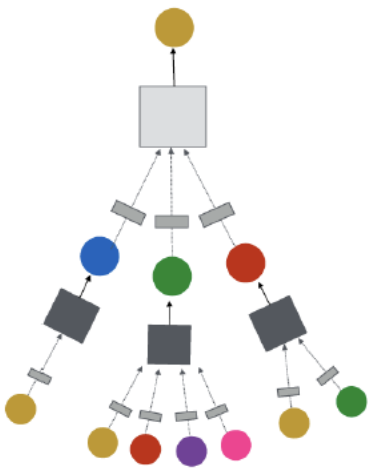




E.g., a drug-drug interaction network

Supervised Training

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



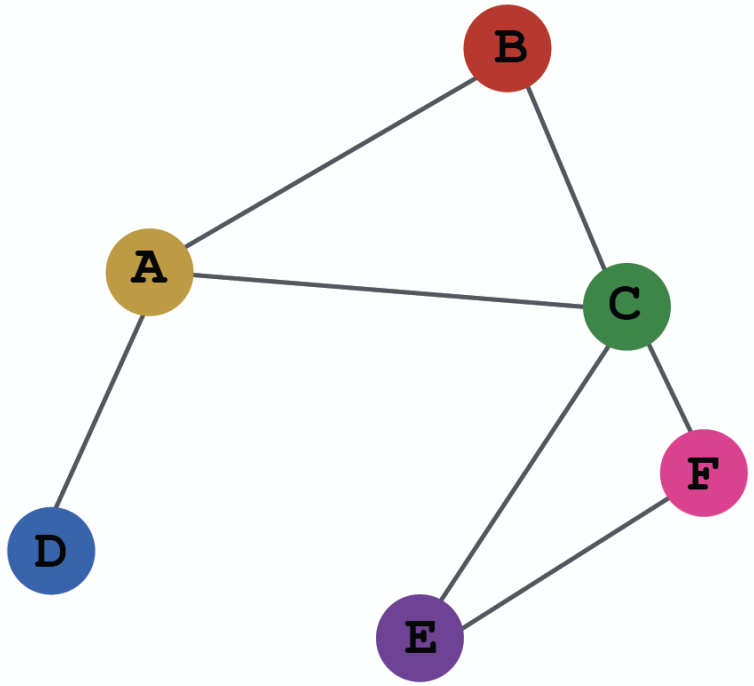
$$\mathcal{L} = -\sum_{v \in V} y_v \log\left(\sigma\left(z_v^T \theta\right)\right) + \left(1 - y_v\right) \log\left(1 - \sigma\left(z_v^T \theta\right)\right)$$

Node class label

Encoder output:
Node embedding

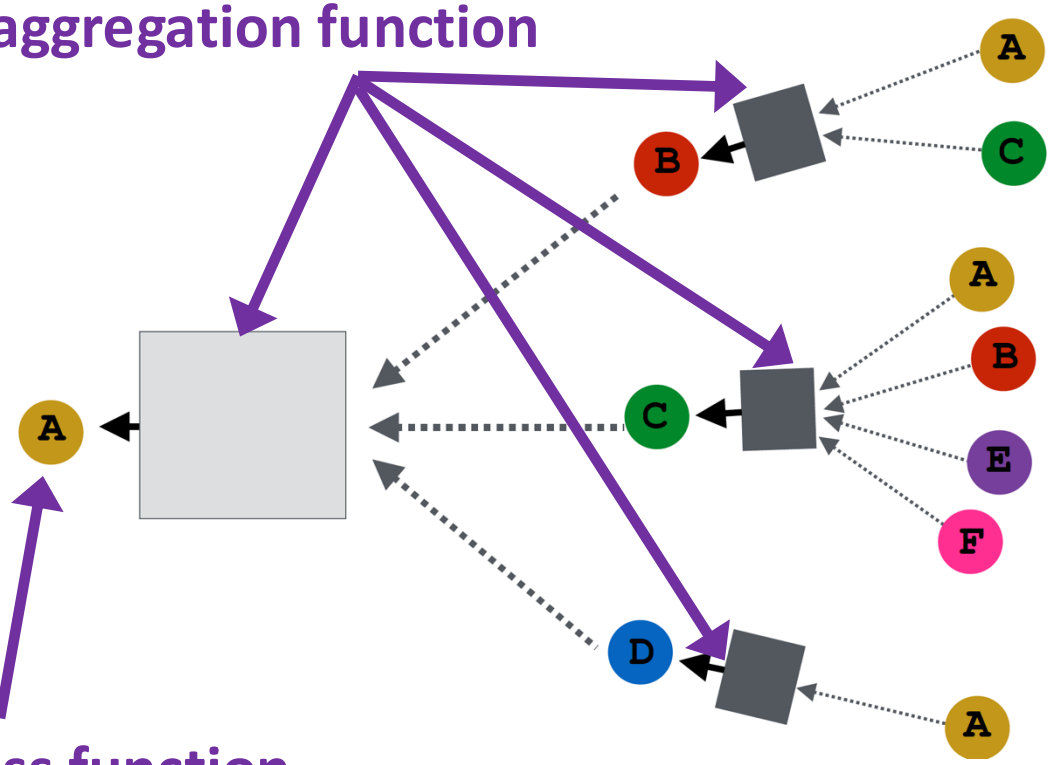
Encoder output:
Node embedding

Model Design: Overview



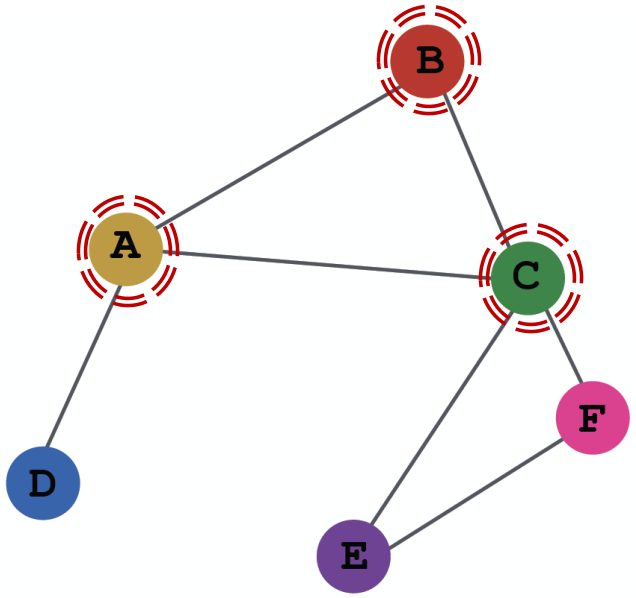
INPUT GRAPH

(1) Define a neighborhood aggregation function



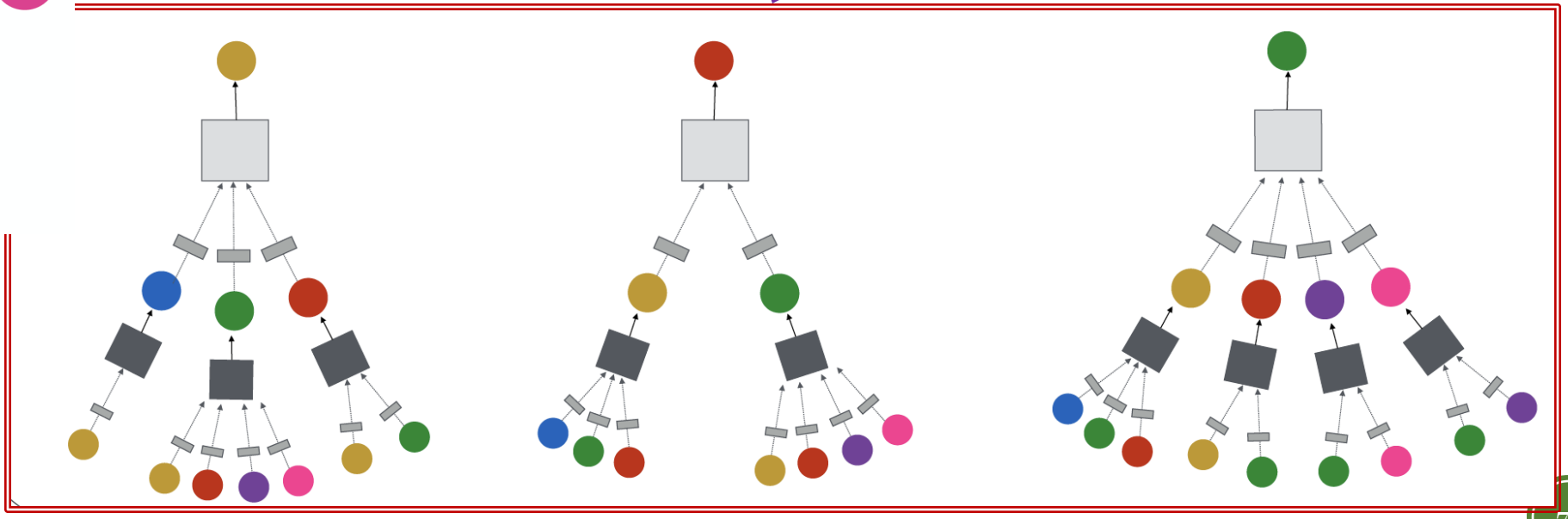
(2) Define a loss function on the embeddings

Model Design

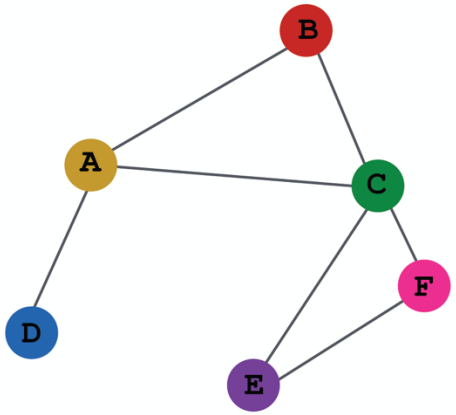


INPUT GRAPH

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



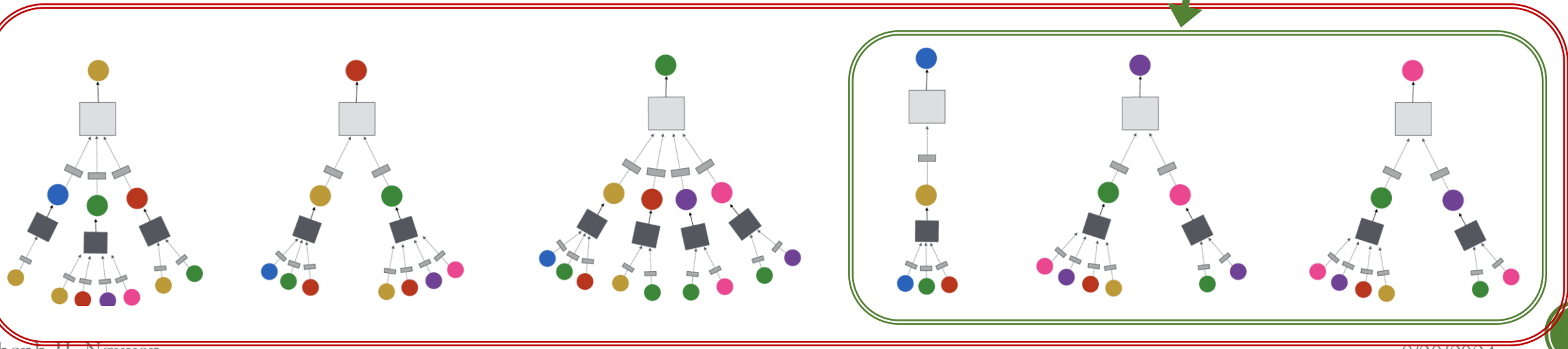
Model Design: Overview



INPUT GRAPH

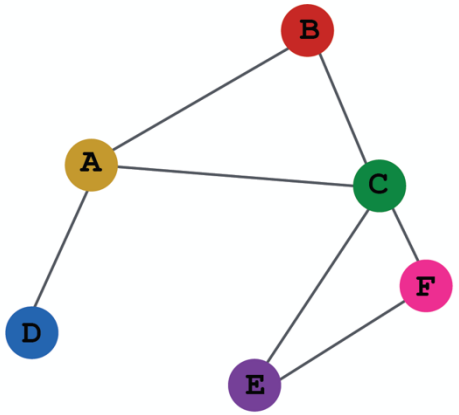
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on

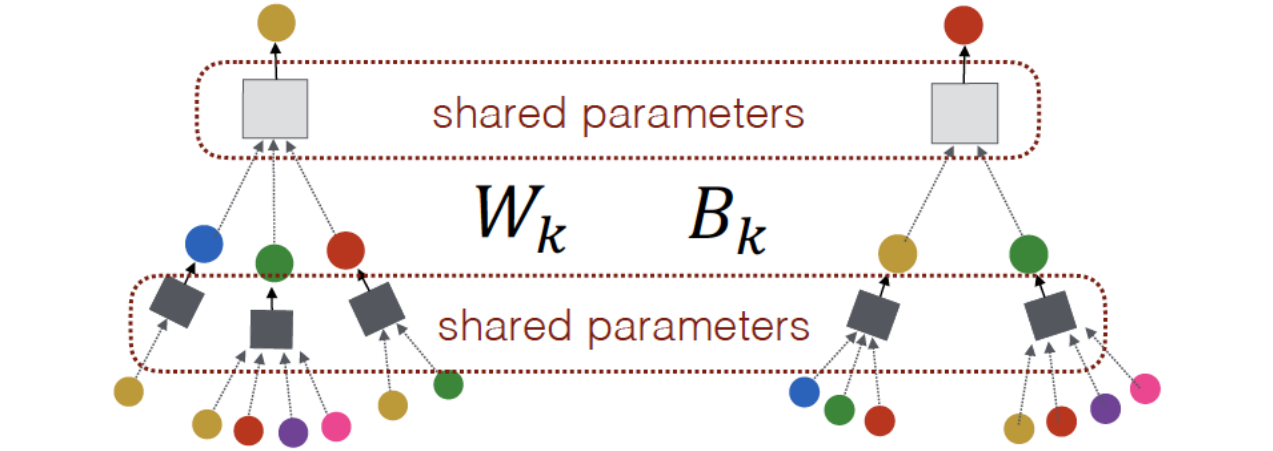


Inductive Capability

- The same aggregation parameters are shared for all nodes
  - #model parameters is sublinear in #nodes;
  - we can generalize to unseen nodes



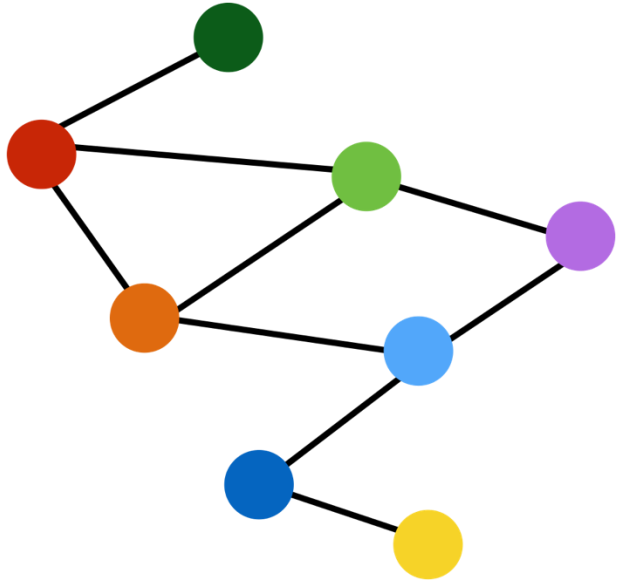
INPUT GRAPH



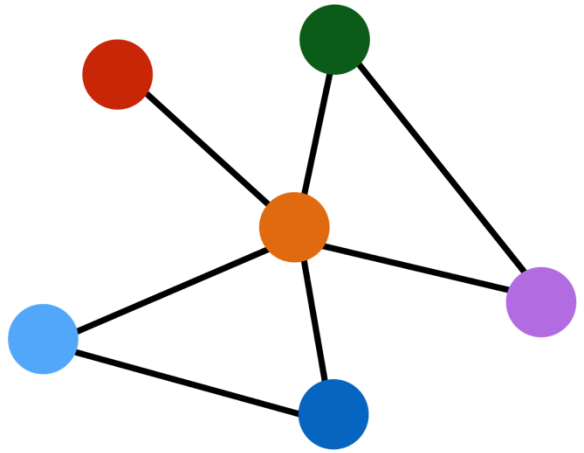
Compute graph for node A

Compute graph for node B

Inductive Capability: New Graphs



Train on one graph

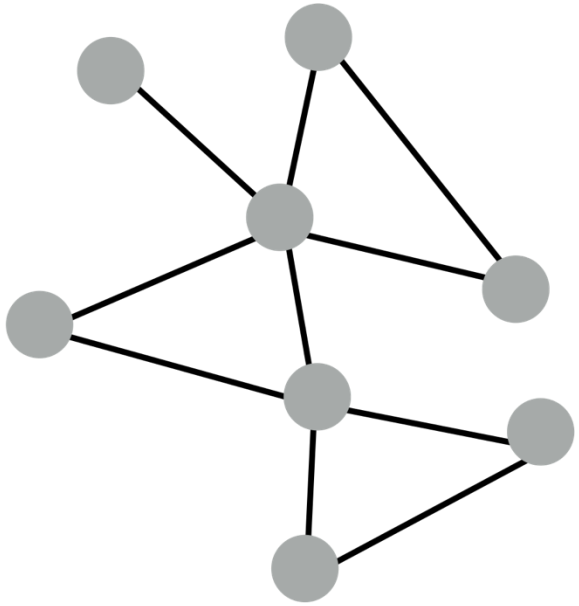


Generalize to new graph

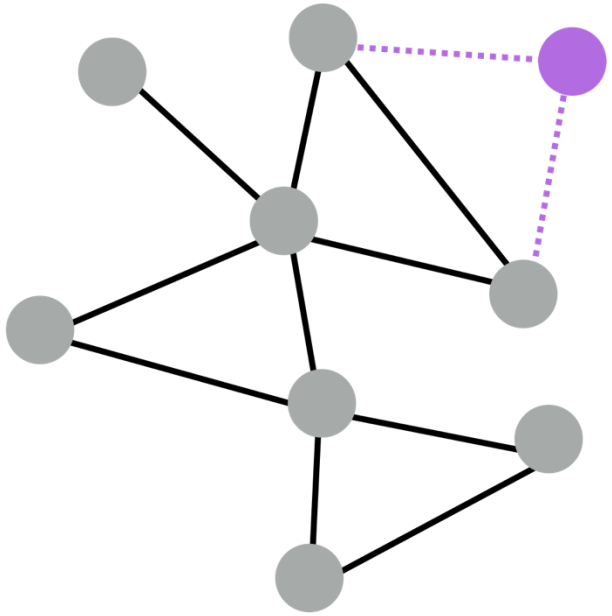
Inductive node embedding

Generalize to entirely unseen graphs

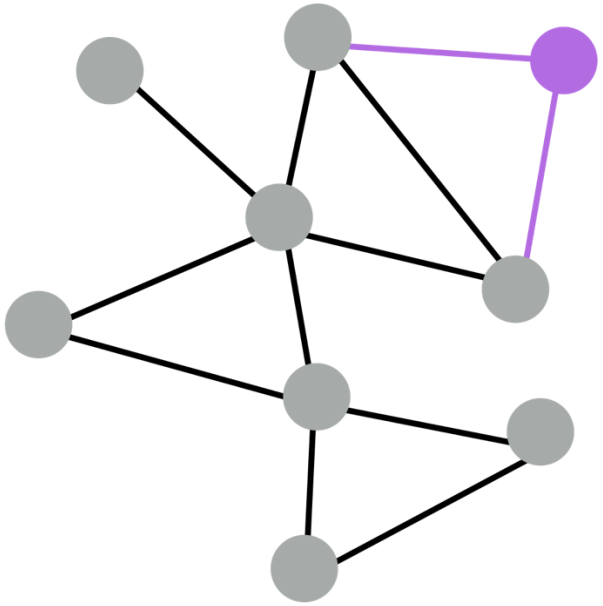
Inductive Capability: New Nodes



Train with snapshot



New node arrives



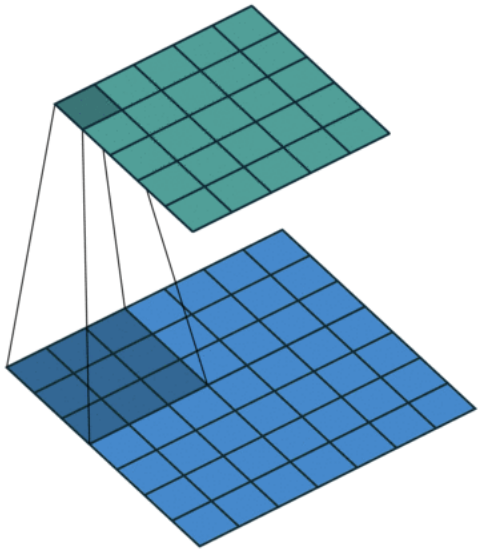
Generate embedding for new node

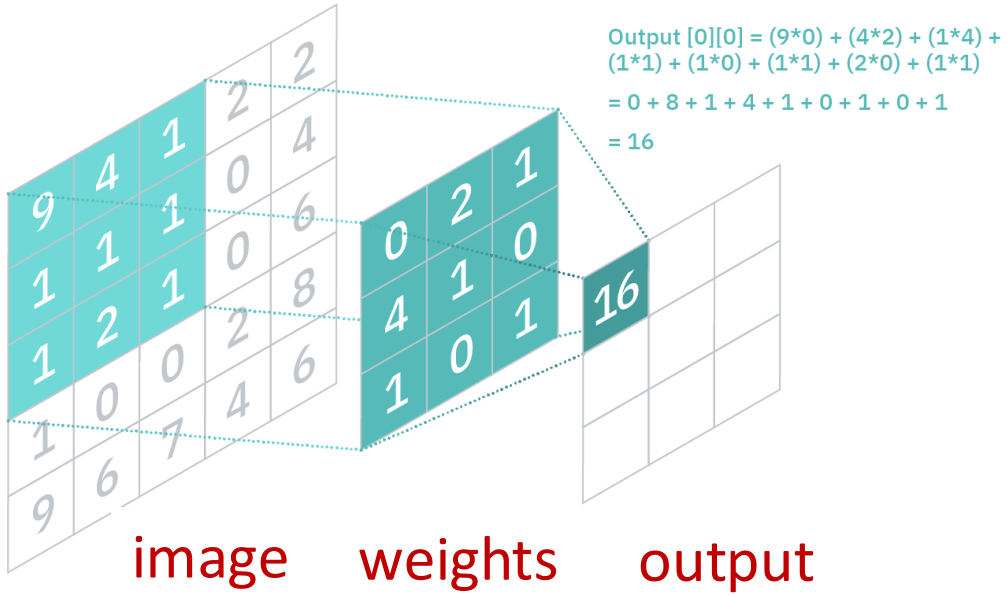
- Applications: Reddit, Youtube, Google scholar
- Generate new embeddings on the fly

GNNs subsume CNNs

Convolutional Neural Networks

- CNN layer with 3x3 filters



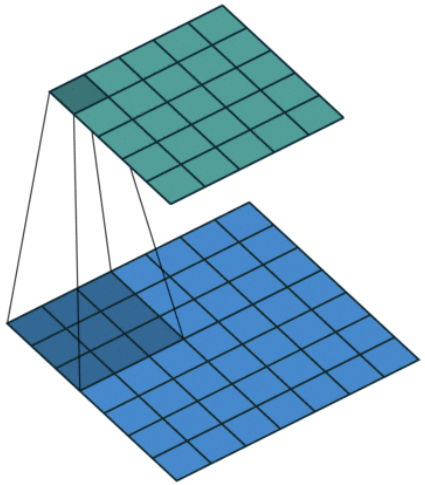


- CNN formulation: $h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}\right), \forall l \in \{0, 1, \cdots, L-1\}$

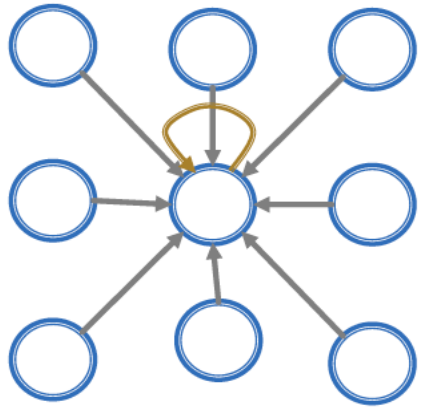
$N(v)$ represents the 8 neighbor pixels of v

GNNs versus CNNs

- CNN layer with 3x3 filters



Image

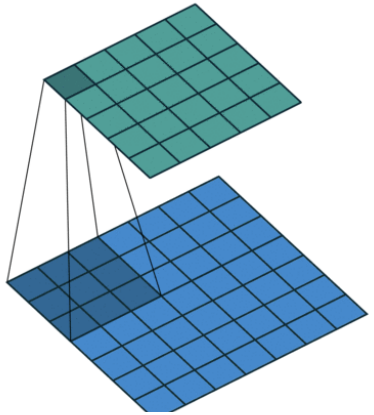


Graph

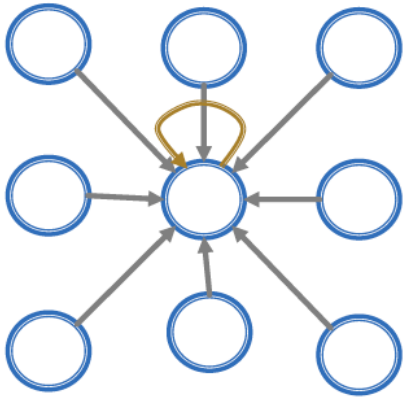
- GNN formulation: $h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{N(u)} + B_l h_v^{(l)}\right), \forall l \in \{0, \cdots, L-1\}$
- CNN formulation: $h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}\right), \forall l \in \{0, 1, \cdots, L-1\}$
- If we rewrite: $h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v)} W_l^u h_u^{(l)} + B_l h_v^{(l)}\right), \forall l \in \{0, \cdots, L-1\}$

GNNs versus CNNs

- CNN layer with 3x3 filters



Image



Graph

- GNN formulation: $h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{N(u)} + B_l h_v^{(l)}\right), \forall l \in \{0, \cdots, L-1\}$
- CNN formulation: $h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v)} W_l^u h_u^{(l)} + B_l h_v^{(l)}\right), \forall l \in \{0, \cdots, L-1\}$

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: $\{(-1,-1),(-1,0),(-1,1),\cdots,(1,1)\}$

GNNs vs CNNs

- 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 of each node

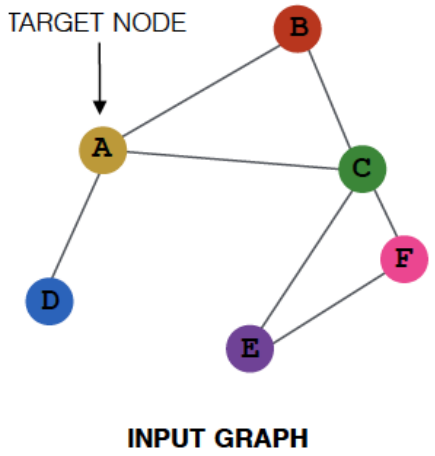
- CNN is not permutation invariant or equivariant
  - Switching the order of pixels leads to different outputs

Summary

- Deep learning for graphs
  - Multiple layers of embedding transformation
  - At each layer, use the embedding at previous layer as the input
  - Aggregation of neighbors and self-embeddings
- Graph convolutional networks
  - Mean aggregation, can be expressed in matrix form
- GNN is a general architecture
  - CNN can be viewed as a special GNN

A General Perspective on GNNs

A General GNN Framework



INPUT GRAPH

(5) Learning objective

(2) Aggregation

GNN Layer 2

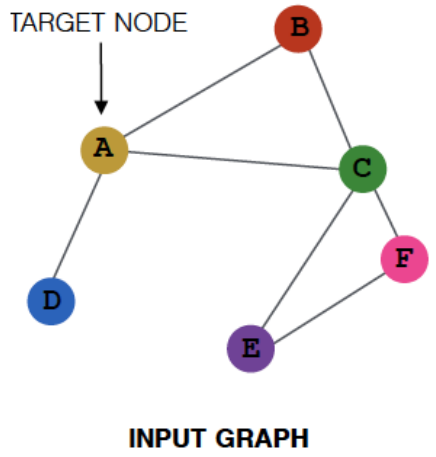
(1) Message

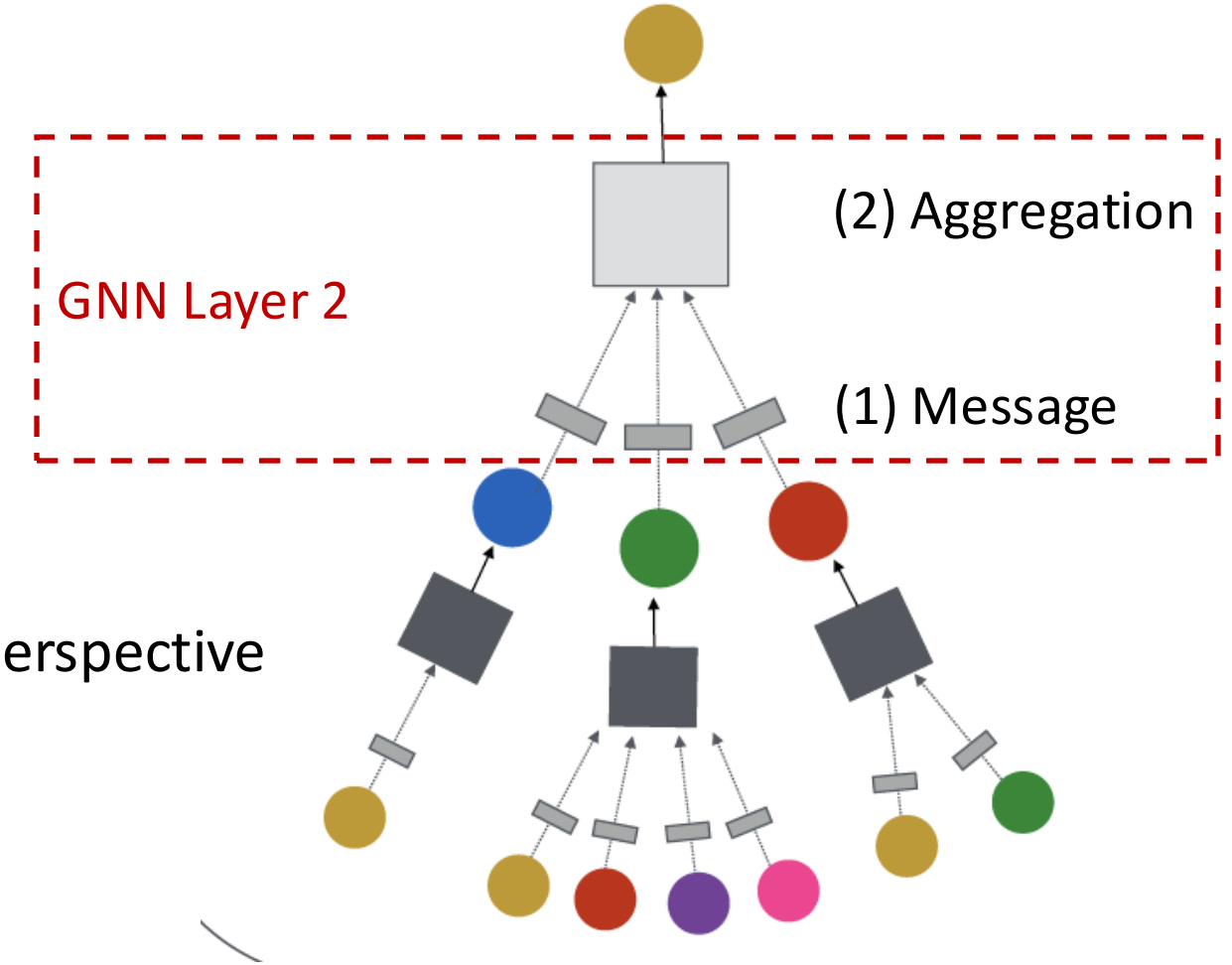
(3) Layer connectivity

GNN Layer 1

(4) Graph augmentation

A GNN Layer



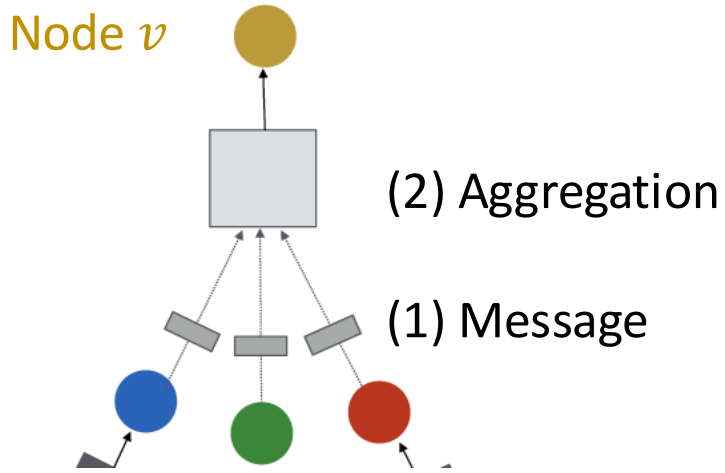


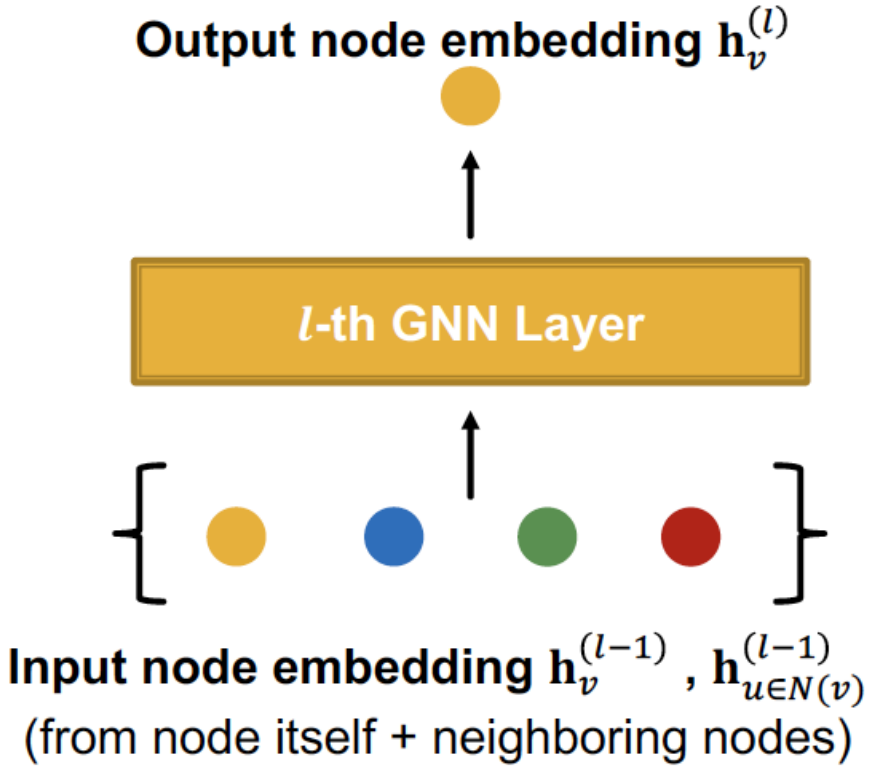
GNN Layer = Message + Aggregation

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

A Single GNN Layer

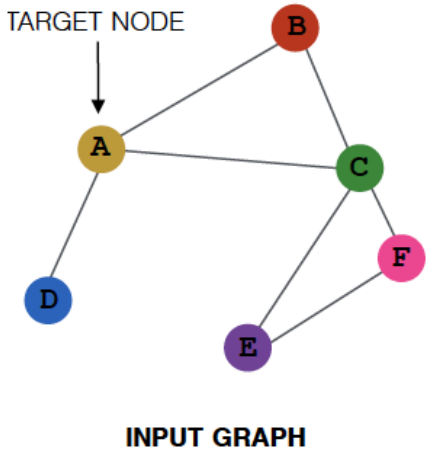
- Idea of a GNN layer:
  - Compress a set of vectors into a single vector
  - Two-step process
    1. Message
    2. Aggregation



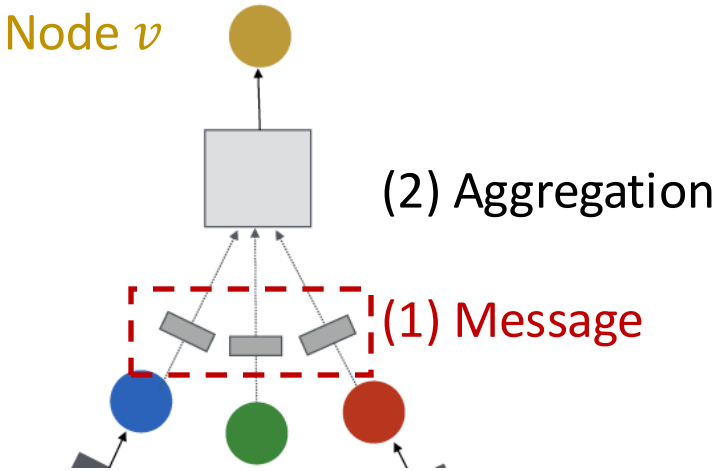


Message Computation

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



INPUT GRAPH

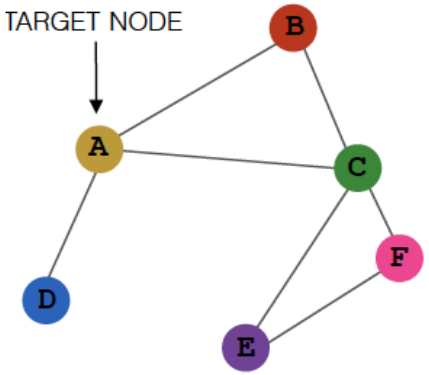


Message Aggregation

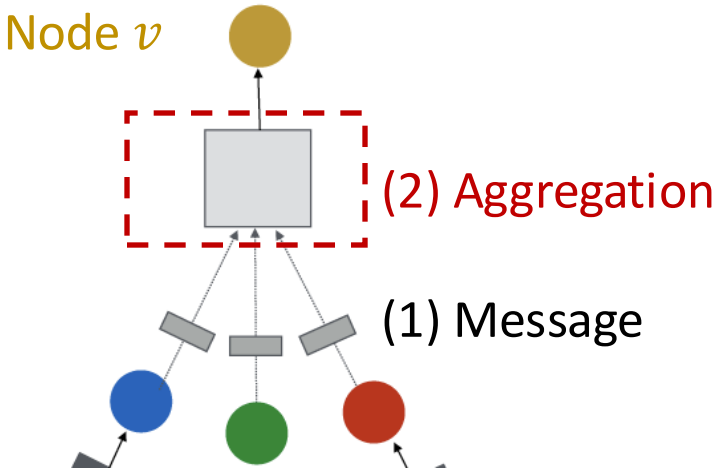
- (2) Aggregation
  - **Intuition**: Node v will aggregate messages from its neighbors u

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

  - **Example:** AGG() can be Sum(), Mean(), or Max() aggregator



INPUT GRAPH



Message Aggregation Issues

- Issue: Information from node v itself could get lost
  - Computation of $h_v^{(l)}$ does not directly depend on $h_v^{(l-1)}$
- Solution: Include $h_v^{(l-1)}$ when computing $h_v^{(l)}$
  - (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

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

First, aggregate from neighbors

A Single GNN Layer

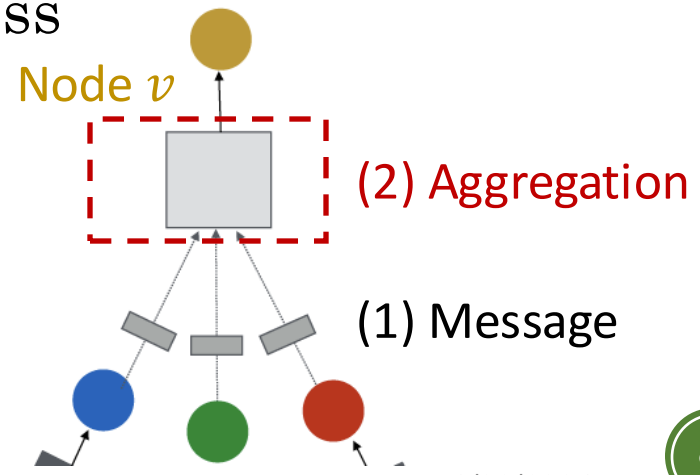
- Putting things together
  - (1) Message: each node computes a message

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

  - (2) Aggregation: aggregate messages from neighbors

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

  - Non-linearity (activation function): Add expressiveness
    - Often write as $\sigma(\cdot)$. Example: ReLU(), Sigmoid(), etc
    - Can be added to message or aggregation



Classical GNN Layers: GCN (1)

- (1) Graph Convolutional Networks (GCN)

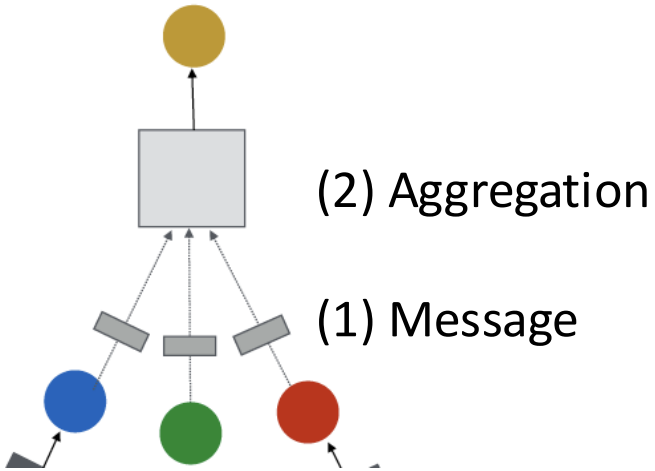
$$h_v^{(l)} = \sigma\left(W^{(l)} \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|} \right)$$

- How to write this as Message + Aggregation?

(1) Message

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

(2) Aggregation

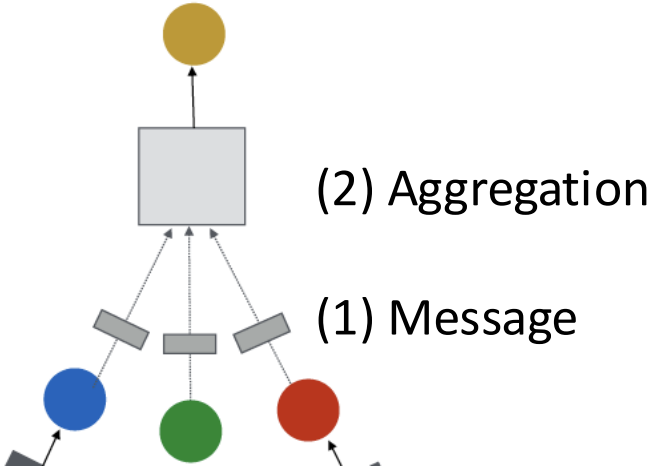


T. Kipf, M. Welling. Semi-Supervised Classification with Graph Convolutional Networks, ICLR 2017

Classical GNN Layers: GCN (2)

- (1) Graph Convolutional Networks (GCN)

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



- Message:
  - Each neighbor: $m_u^{(l)} = \frac{1}{|N(v)|} W^{(l)} h_u^{(l-1)}$

Normalized by node degree

- Aggregation
  - Sum over messages from neighbors, then apply activation
  - $h_v^{(l)} = \sigma\left(Sum\left(\left\{m_u^{(l)}, u \in N(v)\right\}\right)\right)$

Classical GNN Layers: GraphSAGE

- (2) GraphSAGE

$$h_v^{(l)} = \sigma\left(W^{(l)} CONCAT\left(h_v^{(l-1)}, AGG\left(\left\{h_u^{(l-1)}, u \in N(v)\right\}\right)\right)\right)$$

- How to write this as Message + Aggregation
  - Message is computed within the AGG()
  - Two-stage aggregation
    - Stage 1: Aggregate from node neighbors

$$h_{N(v)}^{(l)} = AGG\left(\left\{h_u^{(l-1)}, u \in N(v)\right\}\right)$$

    - Stage 2: Further aggregate over the node itself

$$h_v^{(l)} = \sigma\left(W^{(l)} CONCAT\left(h_v^{(l-1)}, h_{N(v)}^{(l)}\right)\right)$$

Hamilton et al. Inductive Representation Learning on Large Graphs, NeurIPS 2017

GraphSAGE Neighbor Aggregation

- Mean: Take a weighted average of neighbors

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

- Pool: Transform neighbor vectors and apply symmetric vector function Mean() or Max()

$$AGG = Mean\left(\left\{MLP\left(h_u^{(l-1)}\right), \forall u \in N(v)\right\}\right)$$

- LSTM: Apply LSTM to reshuffled of neighbors

$$AGG = LSTM\left(\left[h_u^{(l-1)}, \forall u \in \pi\left(N(v)\right)\right]\right)$$

GraphSAGE: L2 Normalization

- L_2 Normalization:
  - Optional: Apply L_2 normalization to $h_v^{(l)}$ at every layer
  - $h_v^{(l)} \leftarrow \frac{h_v^{(l)}}{\left\| h_v^{(l)} \right\|_2}, \forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$
  - Without L_2 normalization, the embedding vectors have different scales.
  - In some cases (not always), normalization of embedding results in performance improvement

Classical GNN Layers: GAT (1)

- (3) Graph attention networks

Attention weights

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

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

Classical GNN Layers: GAT (2)

- (3) Graph attention networks

Attention weights

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

- Not all node's neighbors are equally important
  - Attention is inspired by cognitive attention
  - The attention α_{vu} focuses on 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 learnt through training

Graph Attention Networks

Can we do better than simple neighborhood aggregation?

Can weighting factors α_{vu} be learnt?

- Goal:
  - Specify arbitrary importance to different neighbors of each node in graph
- Idea
  - Compute embedding $h_v^{(l)}$ of each node in graph following an attention strategy
  - Node attend over their neighbors' messages
  - Implicitly specifying different weights to different neighbors

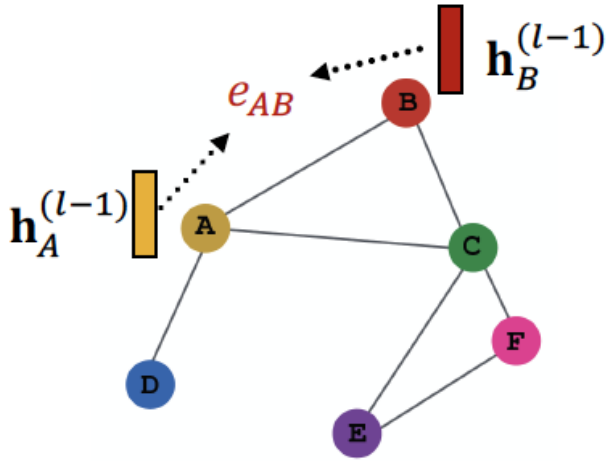
Attention Mechanism (1)

- Let α_{vu} be computed as byproduct of an attention mechanism a
  - Let a compute attention coefficients e_{vu} across pairs of nodes (u, v) based on their messages

$$e_{vu} = a\left(W^{(l)}h_u^{(l-1)}, W^{(l)}h_v^{(l-1)}\right)$$

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

$$e_{AB} = a\left(W^{(l)}h_A^{(l-1)}, W^{(l)}h_B^{(l-1)}\right)$$



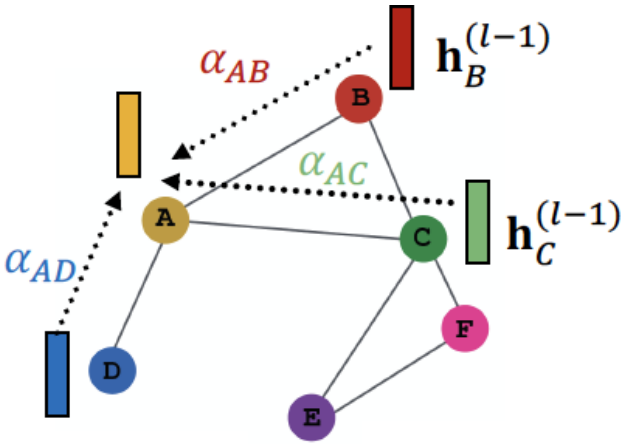
Attention Mechanism (2)

- 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 weights α_{vu}:

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

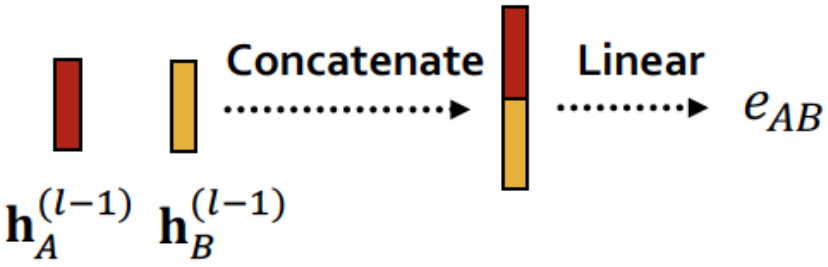


Weighted sum using $\alpha_{AB}, \alpha_{AC}, \alpha_{AD}$:

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

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 has trainable parameters (weights in 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 these parameters with weight matrices (other parameters of neural net $W^{(l)}$) in an end-to-end fashion

Attention Mechanism (4)

- Multi-head attention: Stabilize the learning process of attention mechanism
  - Create multiple attention scores (each replica with a different set of parameters)
  - $h_v^{(l)}[1] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^{1} W^{(l)} h_u^{(l-1)}\right)$
  - $h_v^{(l)}[2] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^{2} W^{(l)} h_u^{(l-1)}\right)$
  - $h_v^{(l)}[3] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^{1} W^{(l)} h_u^{(l-1)}\right)$
  - Outputs are aggregated:
    - By concatenation or summation
    - $h_v^{(l)} = AGG\left(h_v^{(l)}[1], h_v^{(l)}[2], h_v^{(l)}[3]\right)$

Benefits of Attention Mechanism

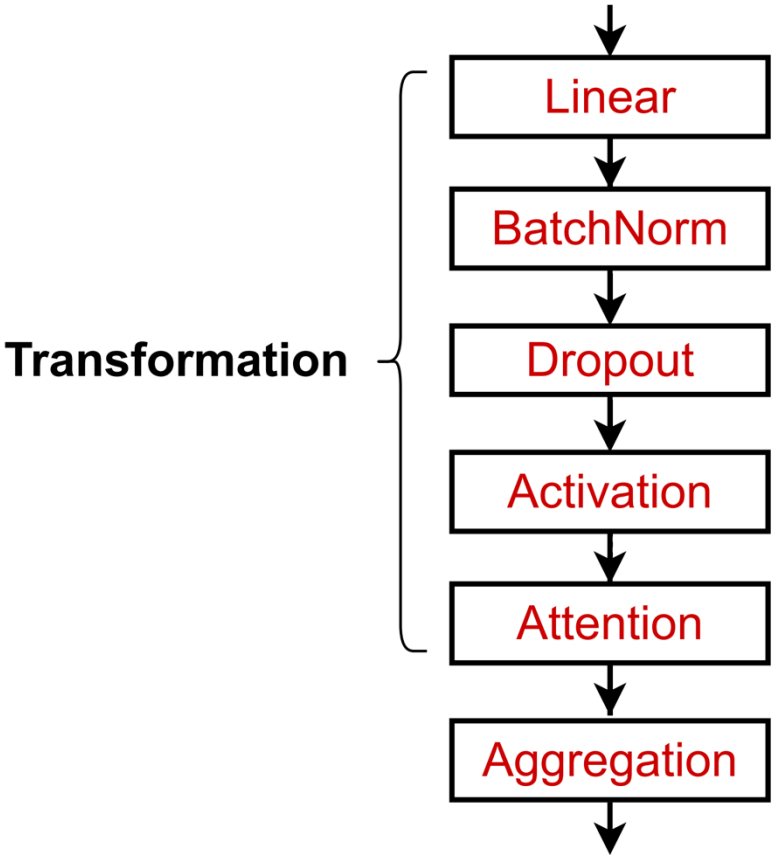
- Key benefit: Allows for (implicitly) specifying different importance values (α_{vu}) to different neighbors
- Computationally efficient
  - Computation of attentional coefficients can be parallelized across all edges of the graph
  - Aggregation may be parallelized across all nodes
- Storage efficient
  - 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

GNN Layers in Practice

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

A suggested GNN Layer

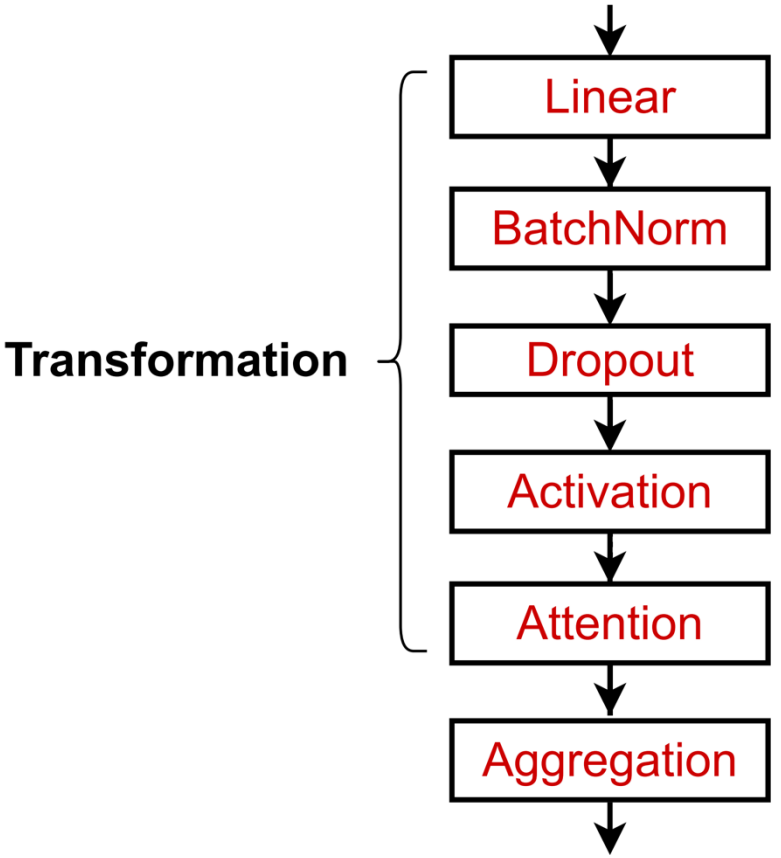


J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

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 messages
  - More:
    - Any useful deep learning modules

A suggested GNN Layer



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

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

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

Step 2:

Normalize the feature using computed mean and variance

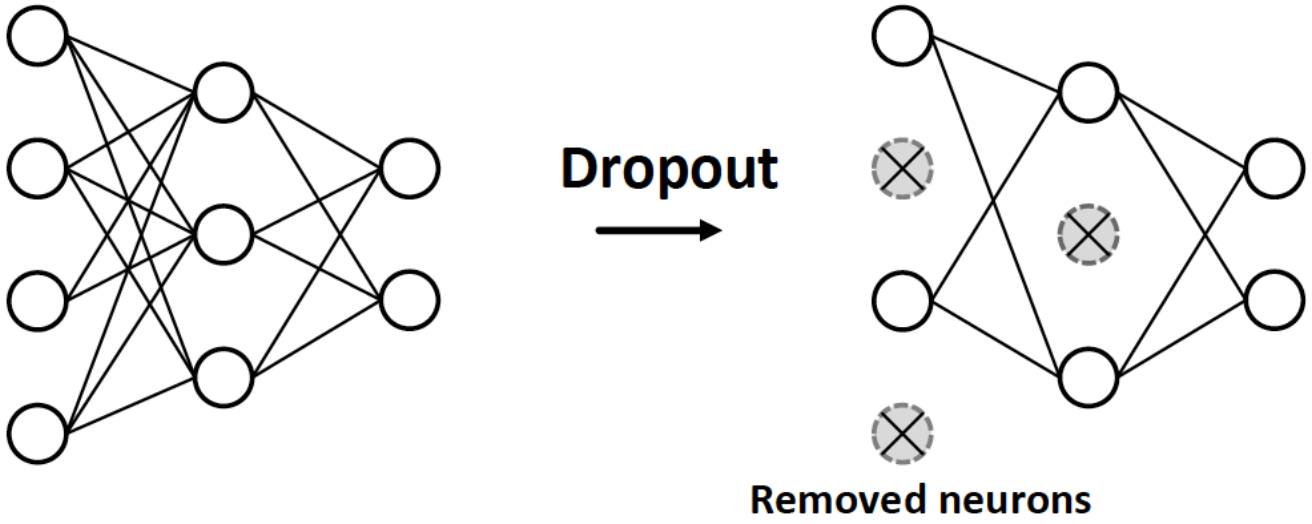
$$\widehat{\mathbf{X}}_{i,j} = \frac{\mathbf{X}_{i,j} - \boldsymbol{\mu}_j}{\sqrt{\boldsymbol{\sigma}_j^2 + \epsilon}}$$

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

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

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

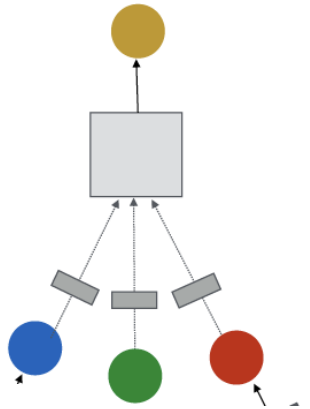


Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

Dropout for GNNs

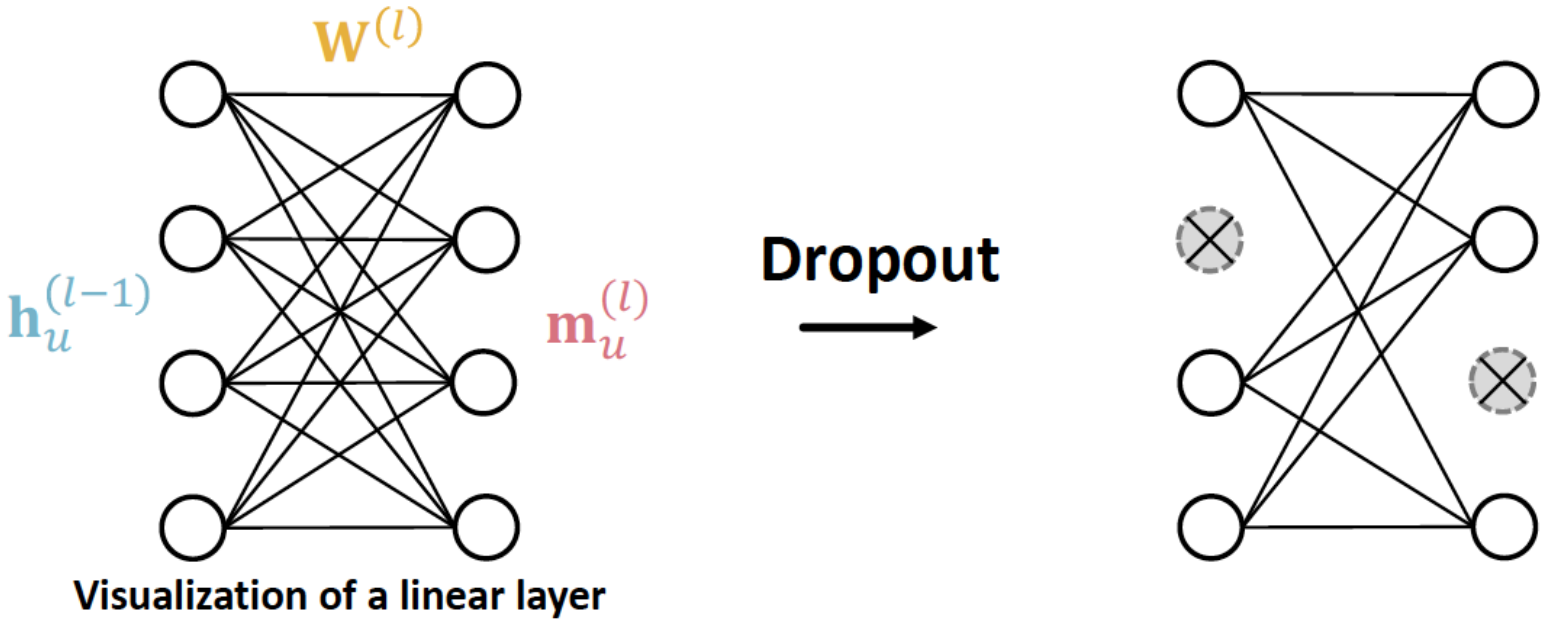
- In GNN, Dropout is applied to the linear layer in the message function
  - A simple message function with linear layer:

$$m_u^{(l)} = W^{(l)} h_u^{(l-1)}$$



(2) Aggregation

(1) Message



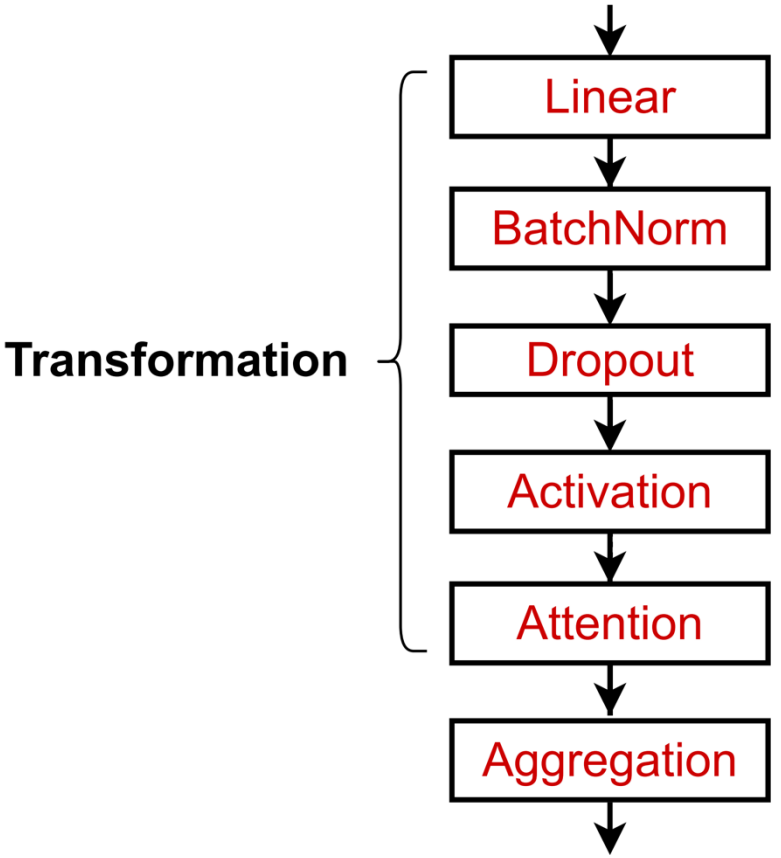
Visualization of a linear layer

GNN Layer in Practice

- Summary: Modern deep learning modules can be included into a GNN layer for better performance

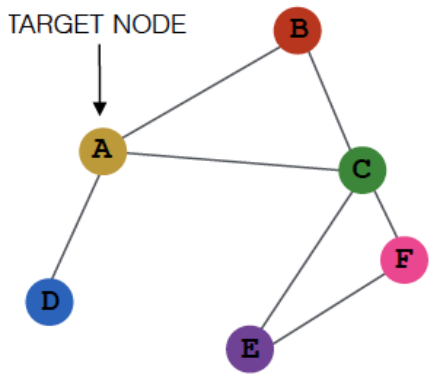
- Designing novel GNN layers is still an active research frontier!

- Suggested resources: You can explore diverse GNN designs or try out your own ideas in GraphGym

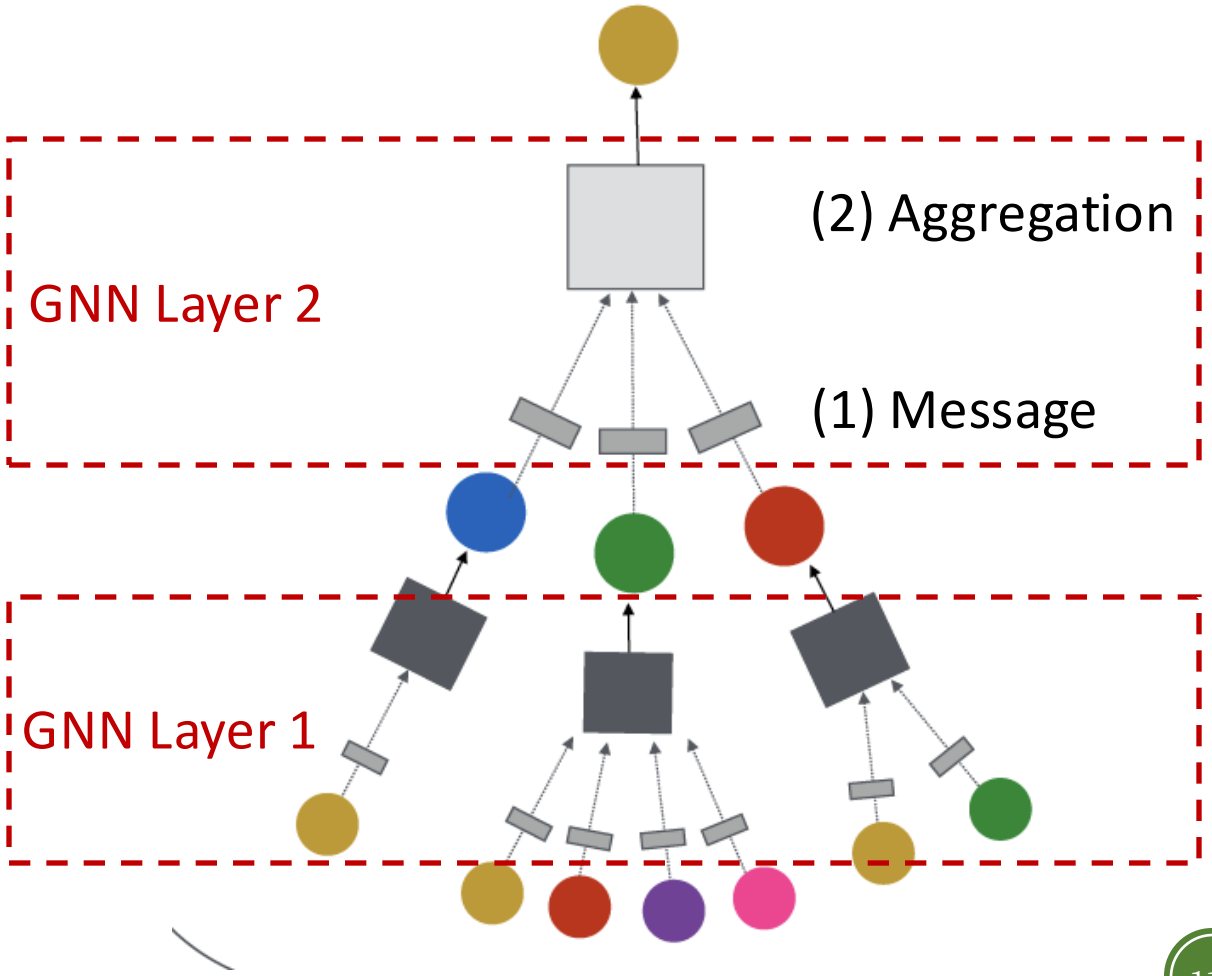


Stacking Layers of a GNN

Stacking GNN Layers



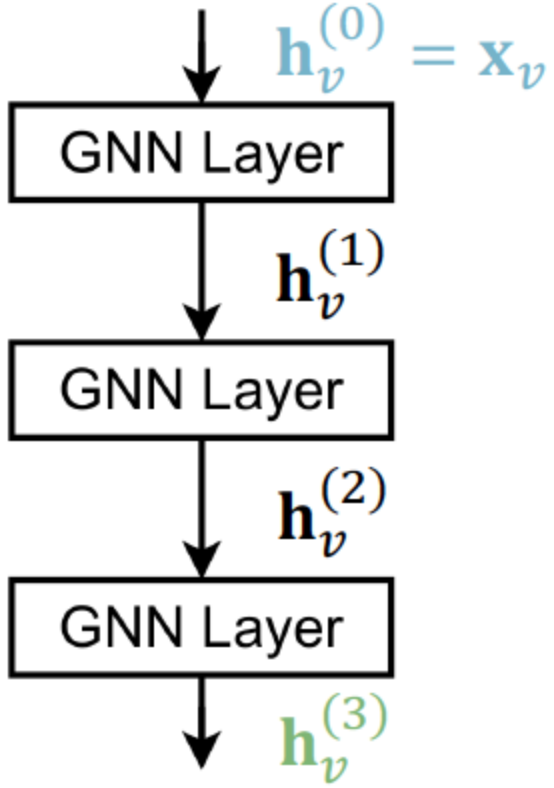
INPUT GRAPH



- How to connect GNN layers into a GNN?
  - Stack layers sequentially
  - Ways of adding skip connections

Stacking GNN Layers

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

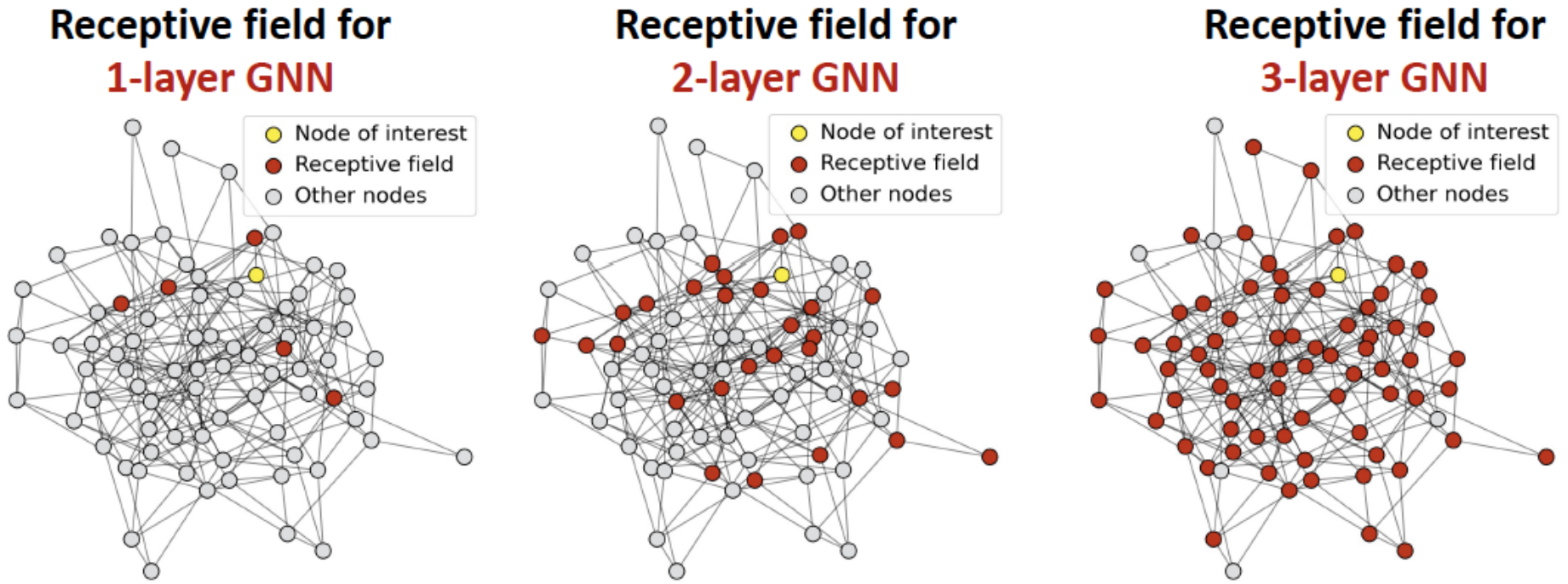


The Over-smoothing Problem

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

Receptive Field of a GNN

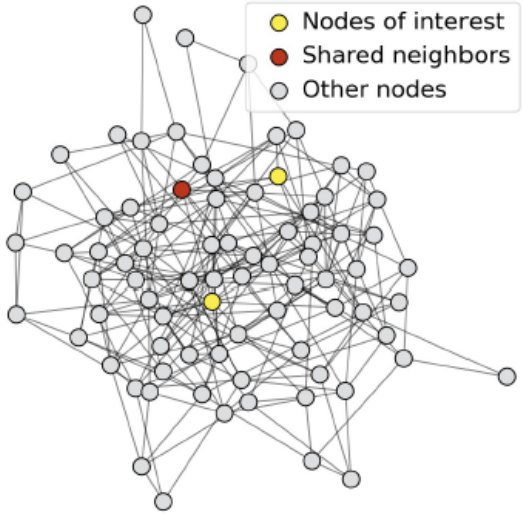
- Receptive field: the set of nodes that determine the embedding of a node of interest
  - In a K-layer GNN, each node has a receptive field of K-hop neighborhood



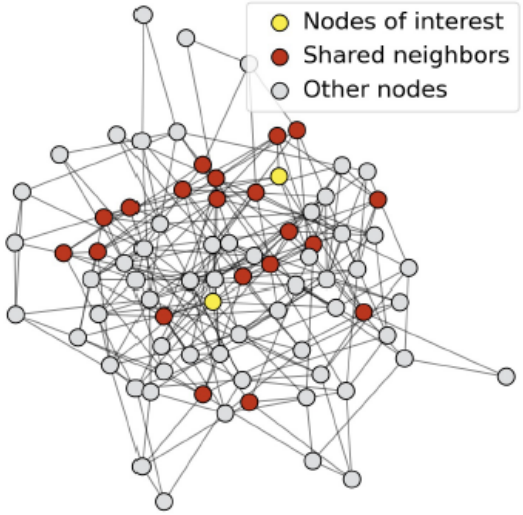
Receptive Field of a GNN

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

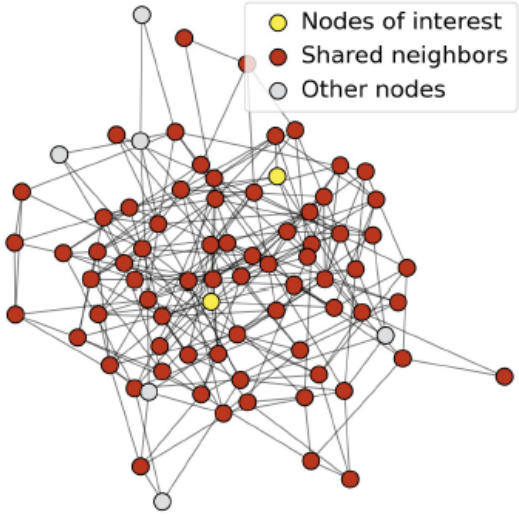
1-hop neighbor overlap
Only 1 node



2-hop neighbor overlap
About 20 nodes



3-hop neighbor overlap
Almost all the nodes!



Receptive Field and Over-smoothing

- We can explain over-smoothing via the notion of the receptive field
  - We know 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 over-smoothing problem
  - How do we overcome over-smoothing problem?

Design GNN Layer Connectivity

- What do we learn from the over-smoothing problem?

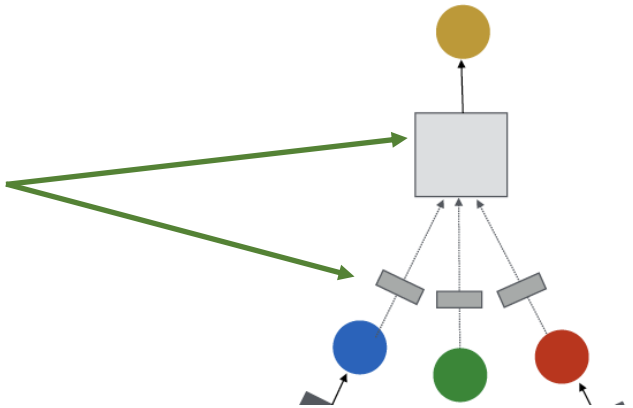
- Lesson 1: Be cautious when adding GNN layers
  - Unlike neural networks in other domains (CNN for image classification), adding more GNN layers do not always help
  - Step 1: Analyze the necessary receptive field to solve your problem. E.g., by computing the diameter of the graph
  - Step 2: Set number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

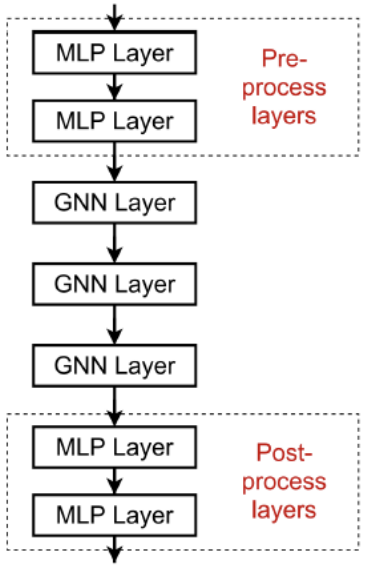
- How to make a shallow GNN more expressive?
- Solution 1: Increase the expressive power within each GNN layer
  - In our previous examples, each transformation or aggregation function only include one linear layer
  - We can make aggregation / transformation become a deep neural network!

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
  - We can make aggregation / transformation become a deep neural network!
    - E.g., we can add MLP layers (applied to each node) before and after
    - GNN layers, as pre-process layers and post-process layers

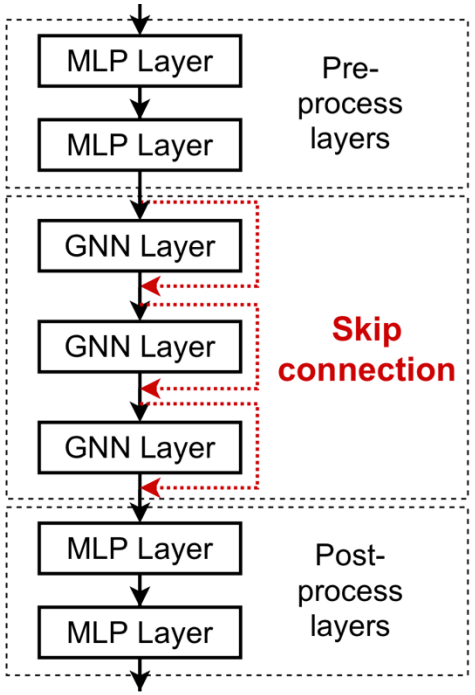


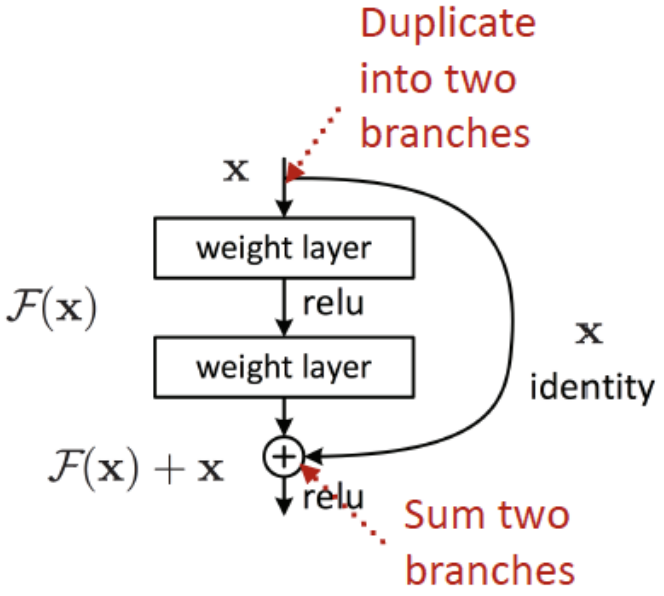
- Pre-processing layers: Important when encoding node features is necessary. E.g., when nodes represent images/text
- Post-processing layers: Important when reasoning / transformation over node embeddings are needed. E.g., graph classification, knowledge graphs
- In practice, adding these layers works great!

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

Design GNN Layer Connectivity

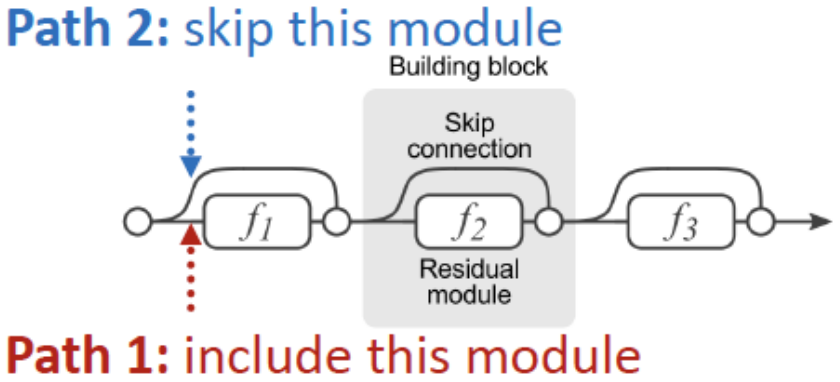
- What if my problem still requires many GNN layers?
- Lesson 2: Add skip connections in GNNs
- Observation from over-smoothing: Node embeddings in earlier GNN layers can sometimes better differentiate nodes
- Solution: We can increase the impact of earlier layers on the final node embeddings, by adding shortcuts in GNN





Idea of Skip Connections

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

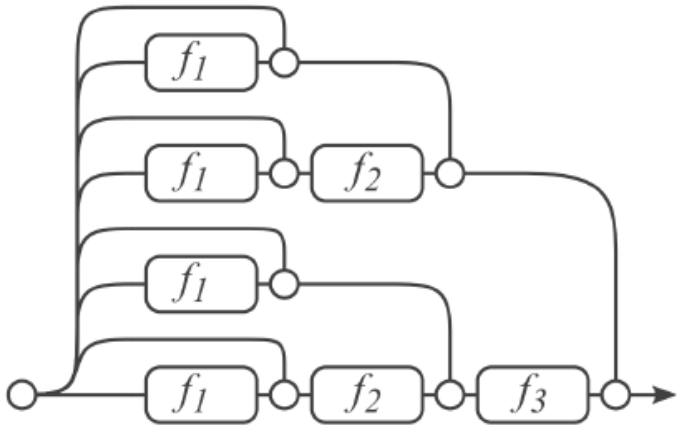


(a) Conventional 3-block residual network

=

All the possible paths:

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



(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

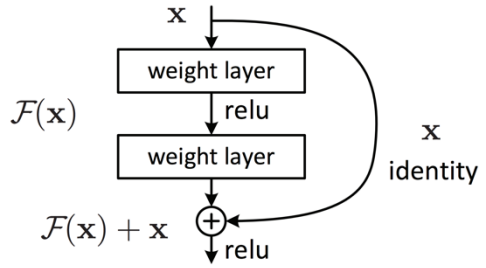
$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

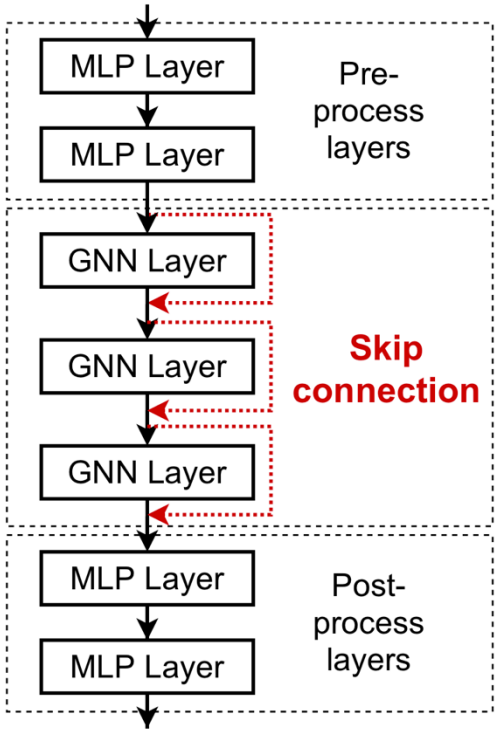
This is our $F(\mathbf{x})$

- A GCN layer with skip connection

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

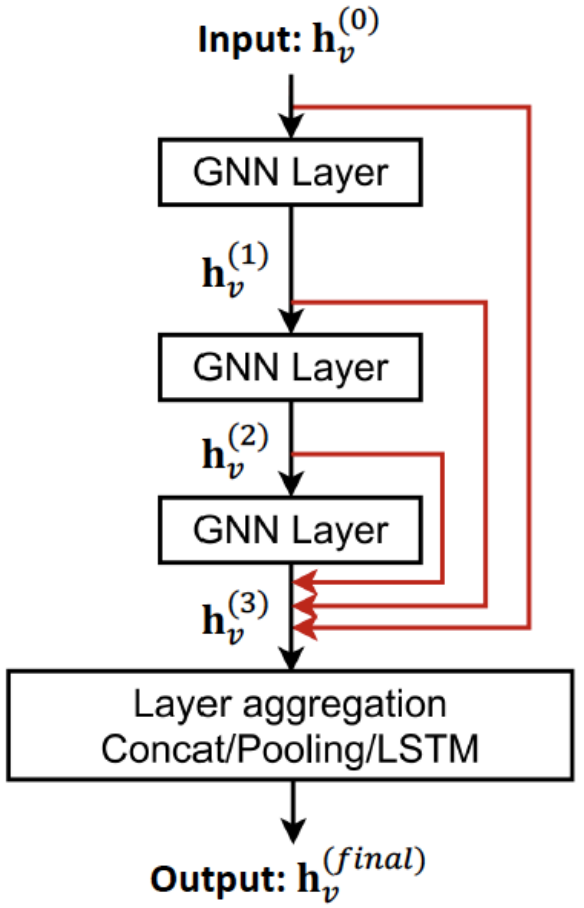
$F(\mathbf{x})$ + $\mathbf{x}$





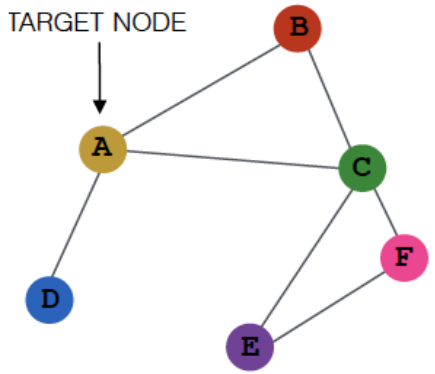
Other Options of Skip Connections

- Other options: Directly skip to the last layer
  - The final layer directly aggregates from the al the node embeddings in the previous layers



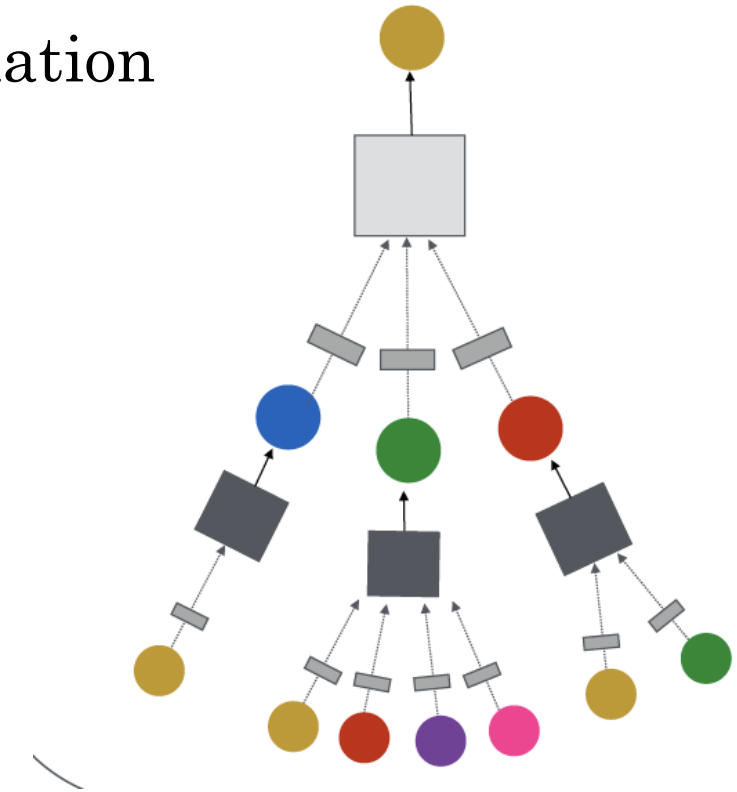
Graph Manipulation in GNNs

General GNN Framework



INPUT GRAPH

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



(4) Graph Manipulation

Why Manipulate Graphs?

- Our assumption so far has been
  - Raw input graph = computational graph
- Reasons for breaking this assumption
  - Feature level
    - The input graph lacks features → feature augmentation
  - Structure level
    - The graph is too sparse → inefficient message passing
    - The graph is too dense → message passing is too costly
    - The graph is too large → cannot fit the computational graph into a GPU
  - It's just unlikely that the input graph happens to be the optimal computation graph for embeddings

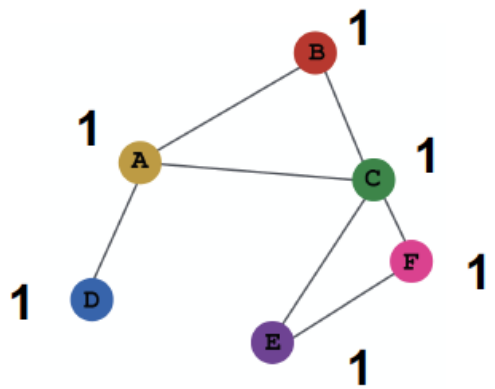
Graph Manipulation Approaches

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

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

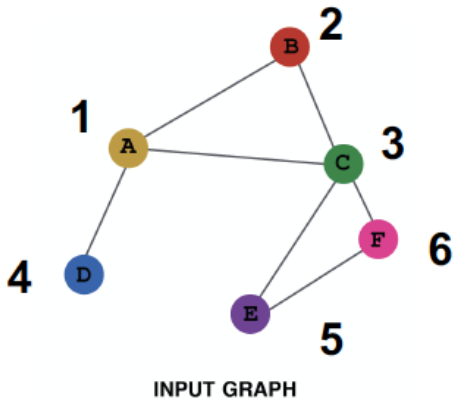
Feature Augmentation on Graphs

- Why do we need feature augmentation?
  - (1) Input graph does not have node features
    - This is common when we only have the adj. matrix
  - Standard approaches:
    - a) Assign constant values to nodes



Feature Augmentation on Graphs

- Why do we need feature augmentation?
  - (1) Input graph does not have node features
    - This is common when we only have the adj. matrix
  - Standard approaches:
    - b) Assign unique IDs to nodes: These IDs are converted into one-hot vectors



INPUT GRAPH

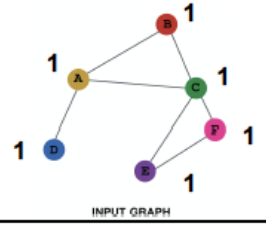
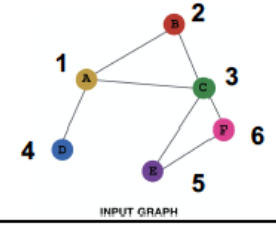
One-hot vector for node with ID=5

ID = 5

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

Total number of IDs = 6

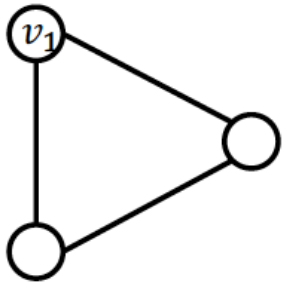
Feature Augmentation on Graphs

	Constant node feature	One-hot node feature
		
Expressive power	**Medium**. All the nodes are identical, but GNN can still learn from the graph structure	**High**. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	**High**. Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN	**Low**. Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs
Computational cost	**Low**. Only 1 dimensional feature	**High**. High dimensional feature, cannot apply to large graphs
Use cases	Any graph, inductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

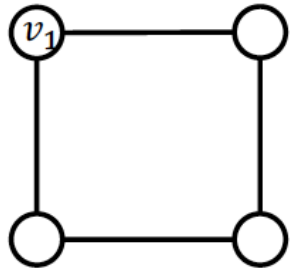
Feature Augmentation on Graphs

- Why do we need feature augmentation?
  - (2) Certain structures are hard to learn by GNN
    - Example: Cycle count feature
    - Can GNN learn the length of a cycle that v_1 resides in?
    - Unfortunately, no.

v_1 resides in a cycle with length 3



v_1 resides in a cycle with length 4



Feature Augmentation on Graphs

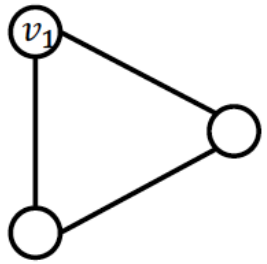
- Why do we need feature augmentation?
  - (2) Certain structures are hard to learn by GNN
  - Solution: We can use cycle count as augmented node features

We start from cycle with length 0

Augmented node feature for v_1

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

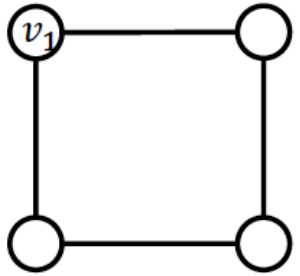
v_1 resides in a cycle with length 3



Augmented node feature for v_1

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

v_1 resides in a cycle with length 4

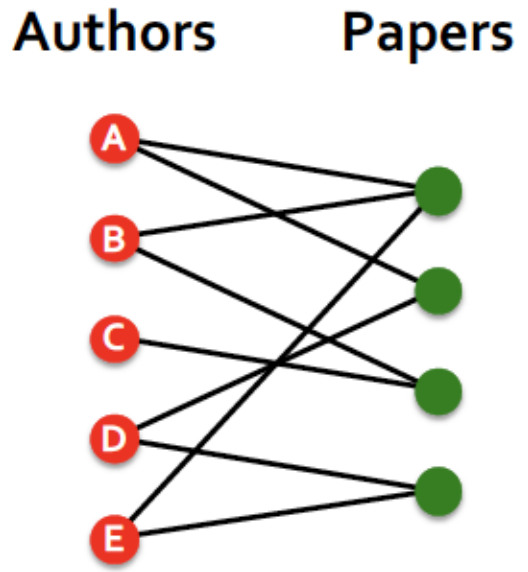


Feature Augmentation on Graphs

- Why do we need feature augmentation?
  - (2) Certain structures are hard to learn by GNN
  - Other commonly used augmented features
    - Clustering coefficient
    - PageRank
    - Centrality
    - …

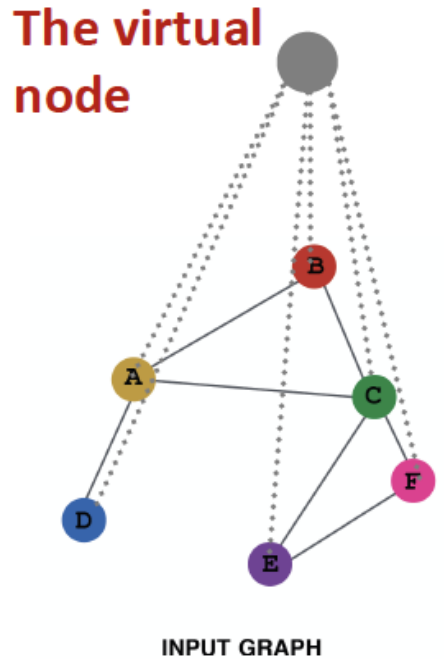
Add Virtual Nodes/Edges

- Motivation: Augment sparse graphs
- (1) Add virtual edges
  - Common approach: Connect 2-hop neighbors via virtual edges
  - Intuition: Instead of using adj. matrix A for GNN computation, use $A + A^2$
- Use cases: Bipartite graphs
  - Author-to-papers (they authored)
  - 2-hop virtual edges make an author-author collaboration graph



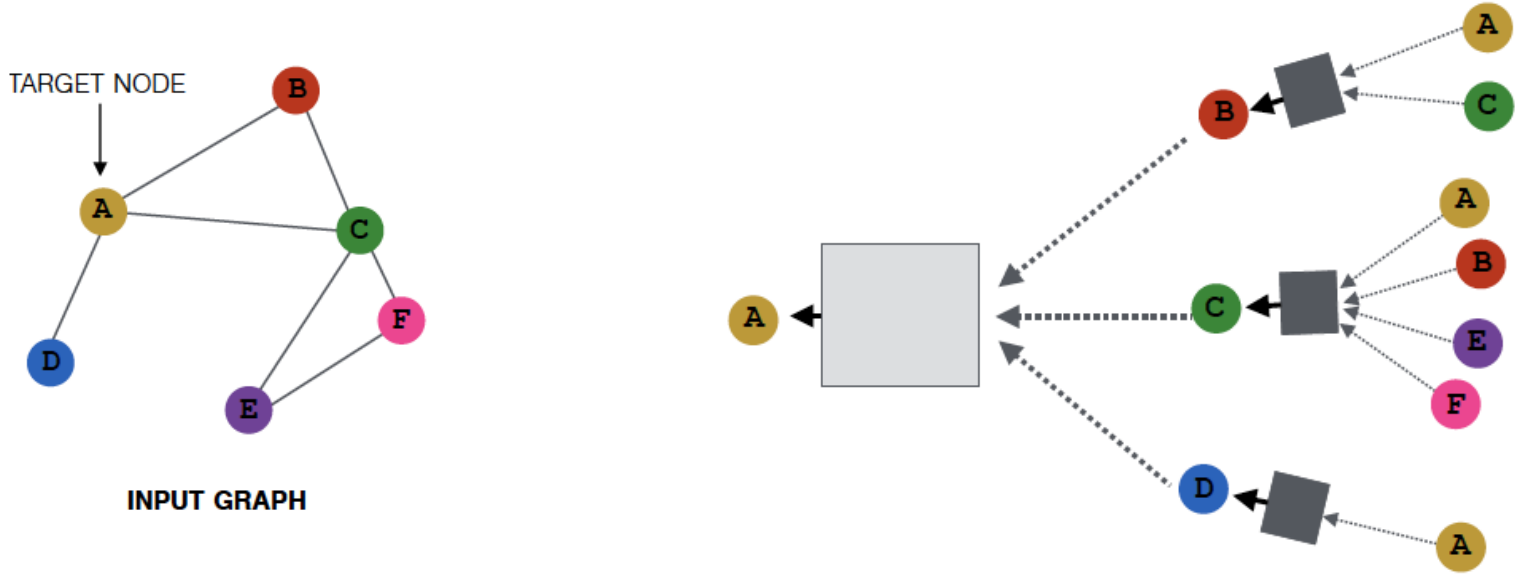
Add Virtual Nodes/Edges

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



Node Neighborhood Sampling

- Our approach so far
  - All neighbors are used for message passing
- Problem: dense/sparse graphs, high-degree nodes

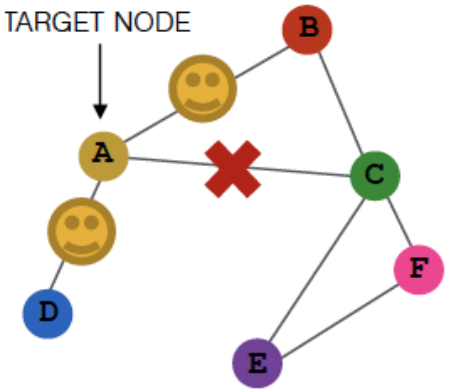


- New idea: (Randomly) determine a node's neighborhood for message passing

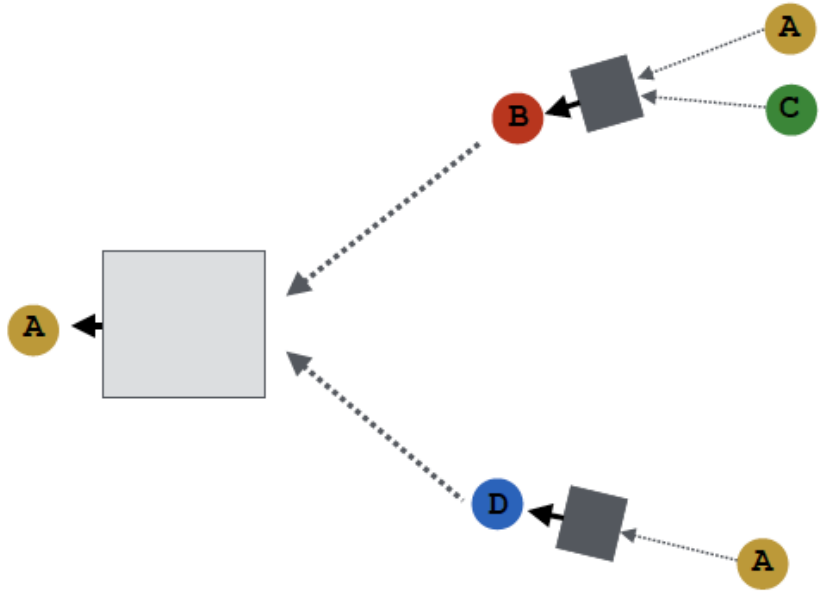
Hamilton et al. Inductive Representation Learning on Large Graphs, NeurIPS 2017

Node Neighboring Sampling Example

- For example, we can randomly choose 2 neighbors to pass messages
  - Only nodes *B* and *D* will pass message to *A*

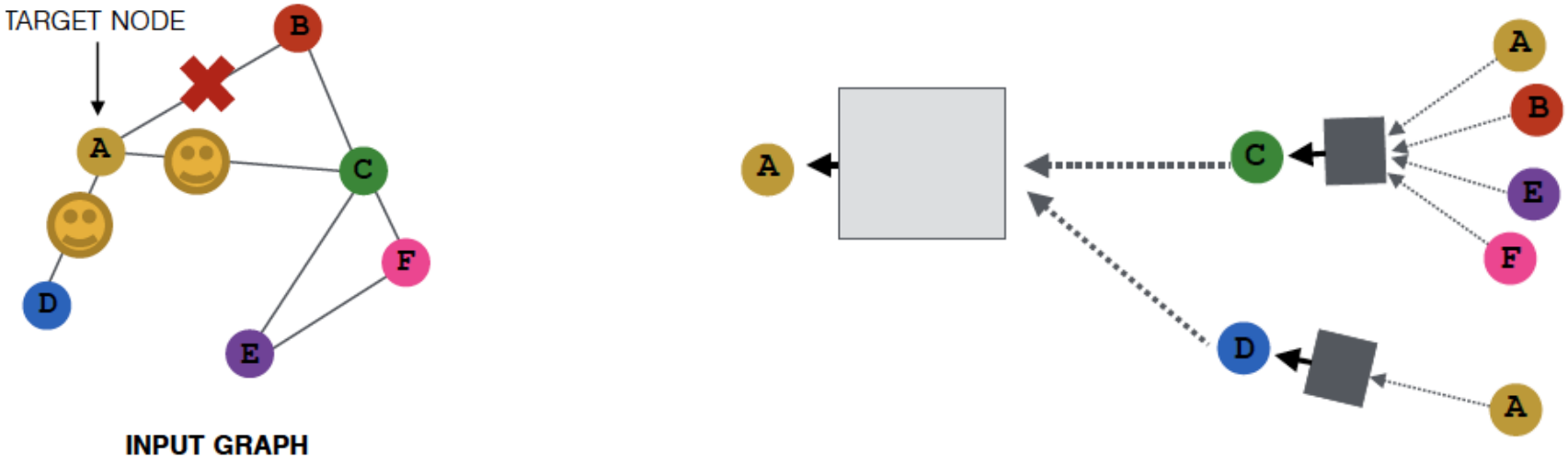


INPUT GRAPH



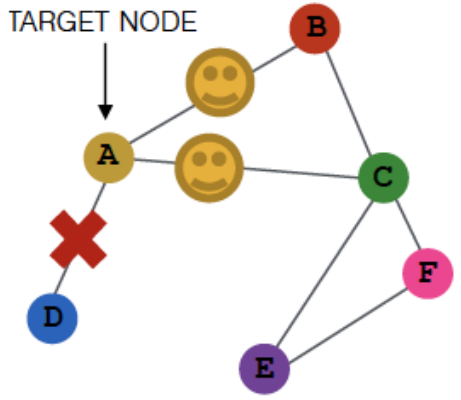
Node Neighboring Sampling Example

- Next time when we compute the embeddings, we can sample different neighbors
  - Only nodes C and *D* will pass message to *A*

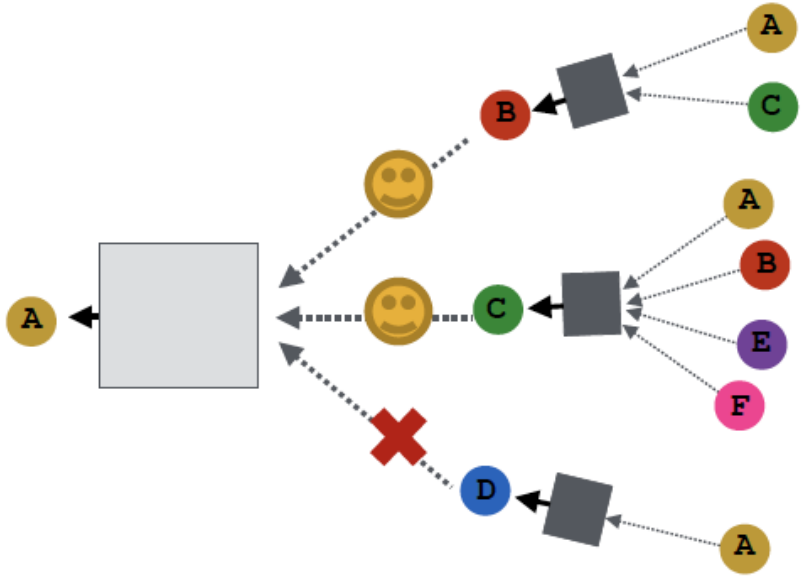


Node Neighboring Sampling Example

- In expectation, we can get embeddings similar to the case where all the neighbors are used
  - Benefits: Greatly reduce computational cost
  - And in practice it works great!



INPUT GRAPH



Summary

- GNN layer
  - Transformation + Aggregation
  - Classic GNN layer: GCN, GraphSAGE, GAT
- Layer connectivity
  - Deciding number of layers
  - Skip connections
- Graph manipulation
  - Feature augmentation
  - Structure manipulation
- Next, GNN objectives, GNN in practice