Graph Neural Networks

11785 Deep Learning
Fall 2024

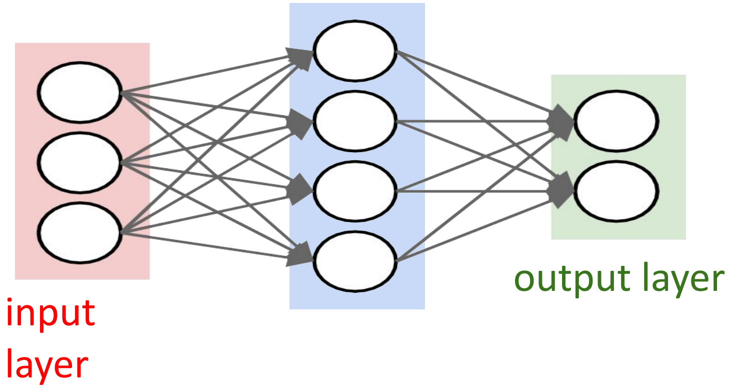
Gabrial Zencha & Carmel SAGBO
11-785, Fall 2024

Models so far

- **MLPs are universal function approximators**
  - Boolean functions, classifiers, and regressions
- **MLPs can be trained through variations of gradient descent**
  - Gradients can be computed by backpropagation

MLP Model

Or, more generally a vector input

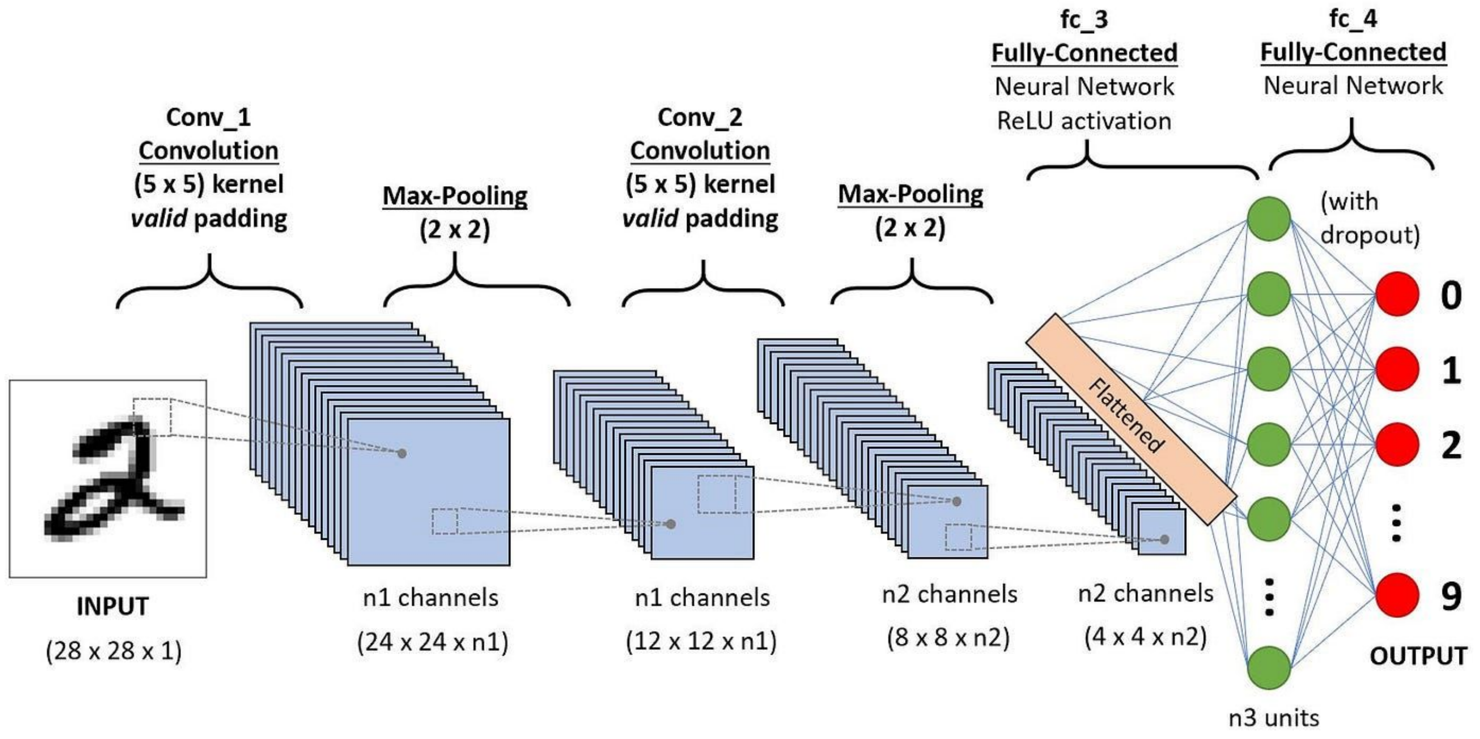


- Can recognize patterns in data
  - E.g. digits
  - Or any other vector data

Models so far

- **CNNs designed for image and spatial data**
  - Convolutional layers learn spatial patterns (e.g., edges, textures).
  - Pooling layers reduce spatial dimensions while retaining key features.
- **CNNs can be trained through variations of gradient descent**
  - Gradients can be computed by backpropagation

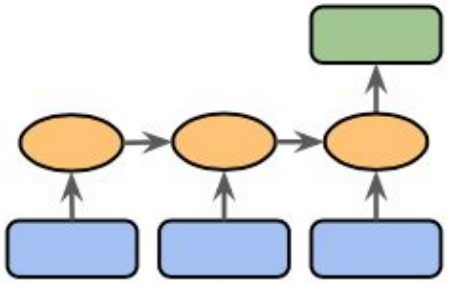
CNN Model

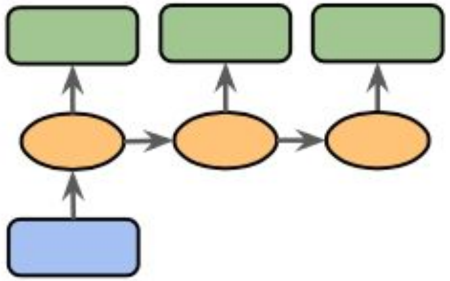

fc_3
Fully-Connected
Neural Network
ReLU activation
fc_4
Fully-Connected
Neural Network
Conv_1
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
Conv_2
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
(with dropout)
0
1
2
9
Flattened
INPUT
(28 x 28 x 1)
n1 channels
(24 x 24 x n1)
n1 channels
(12 x 12 x n1)
n2 channels
(8 x 8 x n2)
n2 channels
(4 x 4 x n2)
OUTPUT
n3 units

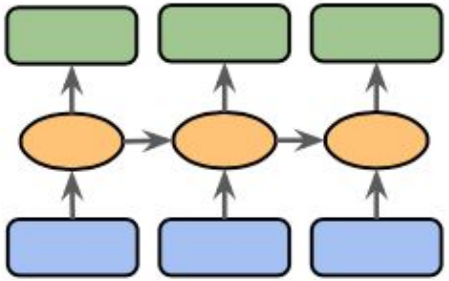
Models so far

- **Sequence-to-Sequence Models: sequential data.**
  - RNNs, LSTMS, Transformers
  - Encode input sequence and decode the encoded sequence.
- **RNNs, LSTMS, Transformers can be trained through variations of gradient descent**
  - Gradients can be computed by backpropagation

Sequence-to-Sequence Model


many-to-one


one-to-many


many-to-many

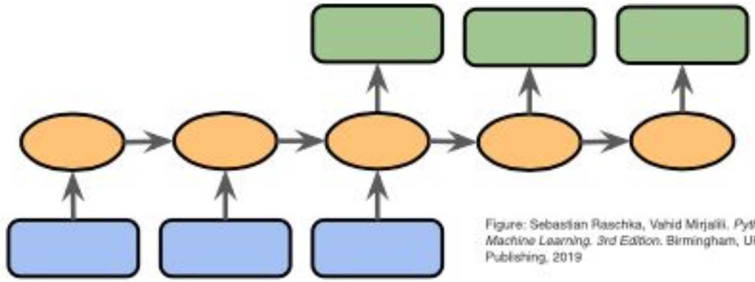

many-to-many

Figure: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Birmingham, UK: Packt Publishing, 2019

Data seen so far (Euclidean Data)

Data that resides in structured, grid-like spaces with well-defined dimensions and coordinate systems

- Tabular Data (MLPs): Rows and columns.
- Images (CNNs): 2D grids of pixel intensities.
- Videos (3D CNNs): Sequential frames forming a spatiotemporal grid.
- Sequences (RNNs, LSTMs Transformers): 1D ordered data like text or time-series.

Non Euclidean Data

Data that resides in irregular, non-grid-like structures where relationships are not confined to regular Euclidean spaces.

- Graphs: Nodes and edges representing entities and relationships.
    - Social networks: People connected by friendships.
    - Molecules: Atoms connected by chemical bonds.
    - Knowledge graphs: Entities linked by relationships.
- Manifolds: Curved surfaces, e.g., 3D shapes or mesh data.
- Point Clouds: Sets of points in 3D space without a grid structure (e.g., LiDAR data).

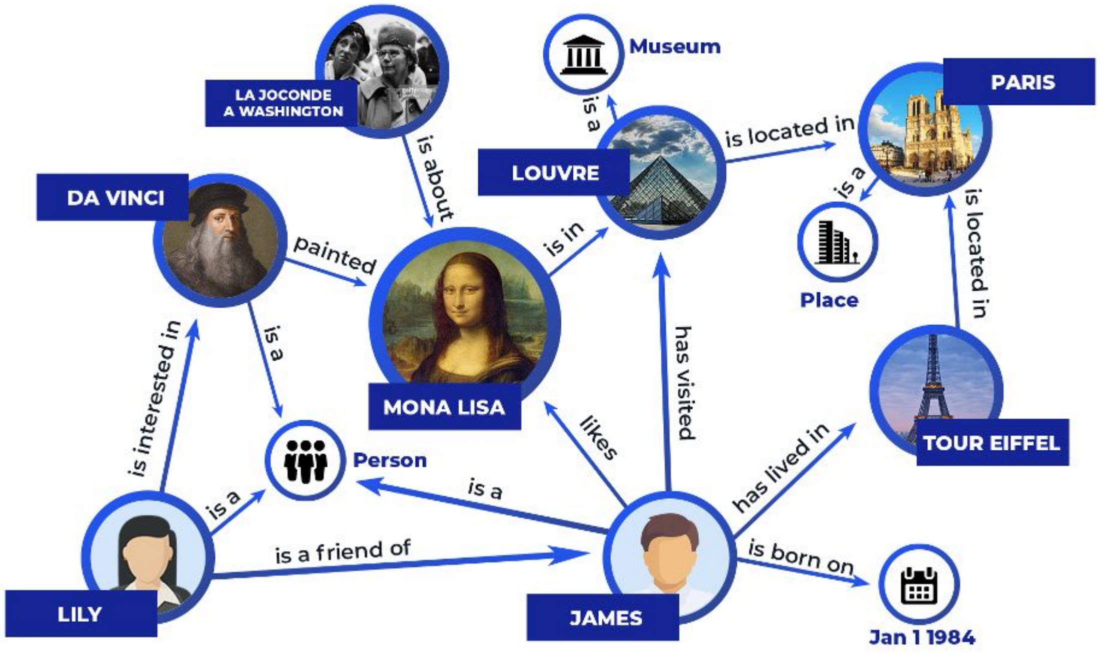
Real-World Data is Often Non-Euclidean

Traffic Networks

Real-World Data is Often Non-Euclidean

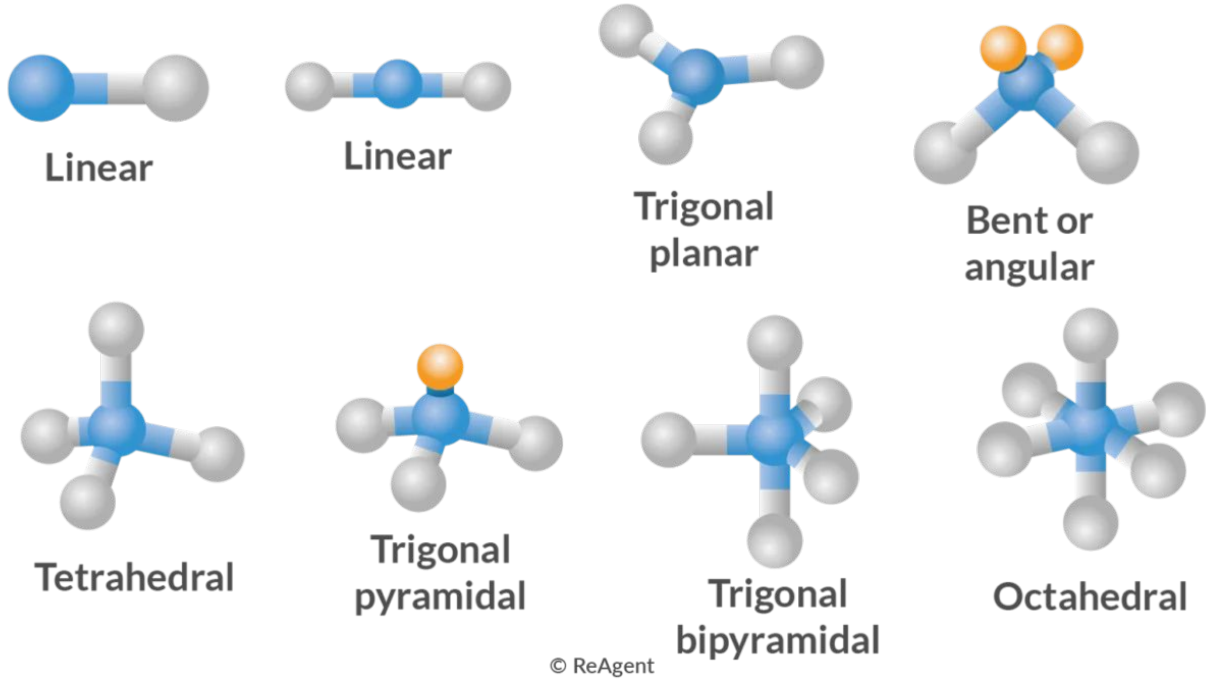
Social Networks

Real-World Data is Often Non-Euclidean



Knowledge Graphs

Real-World Data is Often Non-Euclidean



Complex relationships

Challenges in handling Non-Euclidean Data

– Fixed Grid Assumptions (MLPs, CNNs, RNNs)

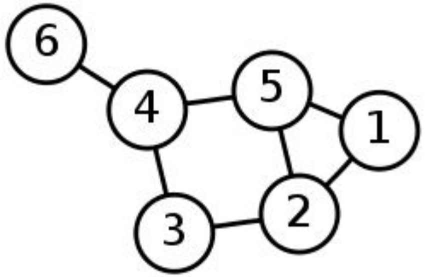
- Assume regular, structured data (e.g., grids or sequences).
- Cannot directly handle irregular neighborhoods or variable node connectivity in graphs or other non-Euclidean structures.

Non-Euclidean data lacks the regular grid structure required for traditional convolution or recurrent processing.

Challenges in handling Non-Euclidean Data

Irregular Neighborhoods:

- Varying numbers of neighbors per node.



- No uniform notion of proximity or direction.

Standard convolution filters (which operate on fixed local neighborhoods) fail to adapt to these variable structures.

Challenges in handling Non-Euclidean Data

Lack of Spatial Regularity:

- The concept of "locality" is not fixed and varies across the structure.

Order Sensitivity:

- Non-Euclidean data like point clouds, graphs (undirected) is unordered.

Defining meaningful filters or operations without losing structural information is non-trivial.

We need a permutation invariant / equivariant

Why it Matters ?

- Enabling Novel Applications
  - Drug discovery: Predict molecule effectiveness or toxicity.
  - Social network analysis: Detect influencers or communities.
  - Recommender systems: Suggest products or content using knowledge graphs.
- Capturing Complex Relationships
  - Many problems require understanding relationships, not just data points.
- Improved Performance in Existing Tasks
  - Models that consider the graph of road networks outperform grid-based approaches by understanding connectivity.

Poll 1

True or False

1. Euclidean data refers to data that lies in a space where the distance between points is calculated using the Euclidean distance formula, while non-Euclidean data involves spaces where the concept of distance may follow different rules, such as hyperbolic or graph-based distances.
2. CNNs and MLPs are specifically designed to handle non-Euclidean data, such as graphs and hyperbolic spaces, without any modifications.

Poll 1

True or False

1. Euclidean data refers to data that lies in a space where the distance between points is calculated using the Euclidean distance formula, while non-Euclidean data involves spaces where the concept of distance may follow different rules, such as hyperbolic or graph-based distances. **(True)**
2. CNNs and MLPs are specifically designed to handle non-Euclidean data, such as graphs and hyperbolic spaces, without any modifications. **(False)**

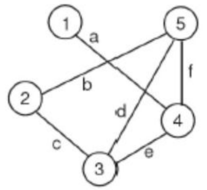
How to solve challenges faced by other models (MLPs, CNNS, Seq-Seq) with Non-Euclidean data

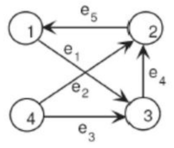
Graph Neural Networks

What is a Graph ?

– In one restricted but very common sense of the term, a graph is an ordered pair **G = (V, E)** comprising :

- **V** a set of vertices (also called nodes or points)
- **E** $\subseteq$ {{x, y}|x, y $\in$ **V** and x ≠ y} a set of edges (also called links or lines), which are unordered pairs of vertices (that is, an edge is associated with two distinct vertices).





$$A_1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Graph Representation

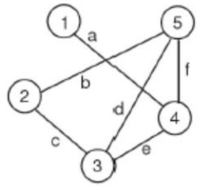
A Graph is generally represented using these different forms:

- The adjacency Matrix A :

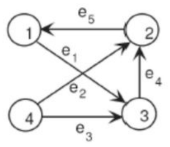
  It is a ***n* × *n*** matrix in which:

  - **n** in the number of vertices
  - **A(i, j) = 1** only if there is a link from i to j and
  - **A(i, j) =** 0 if not.

- Other common representation is based of Edge Features or Node Features.
-



$$A_1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$



$$A_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Graph Node Embeddings

- Motivation for Graph Node Embedding

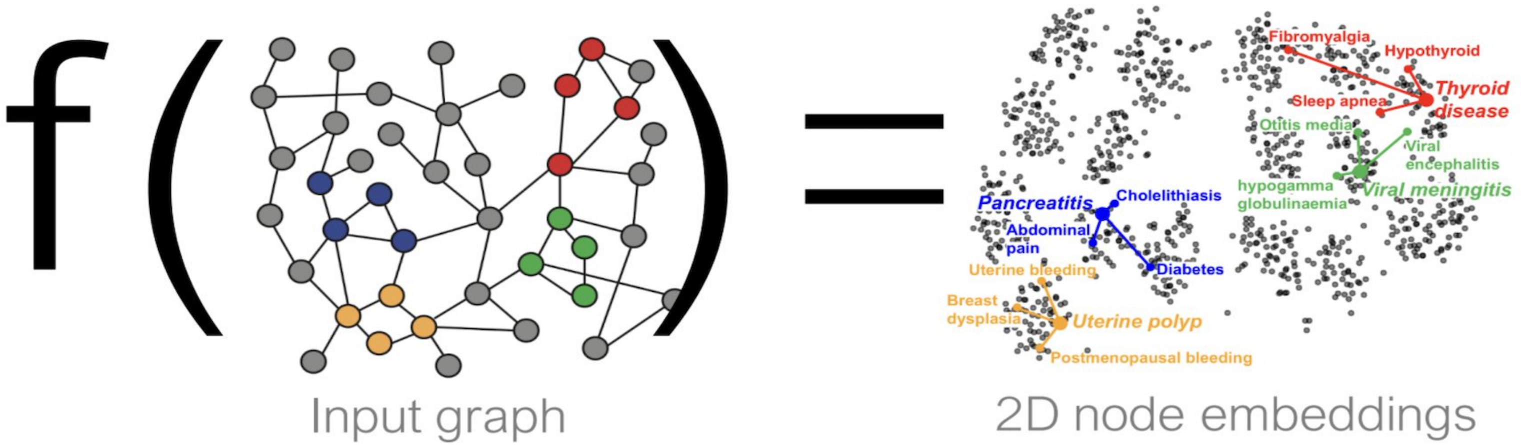
- GloVe co-occurrence graph → word embedding → NLP
- Hyperlinked websites → page embedding → websites classification
- Citation graphs → article embedding → literature classification
- Co-author graphs → author embedding → community detection
- Molecular structure graph → atom embedding → AI for science

- Unified view

- **Nodes** can be any objects (words, documents, authors, atoms, proteins, etc.)
- **Links** represent the interactions or dependencies among nodes.
- **Embedding Vectors**
  - Capturing the latent features of nodes based on graph structures
  - Supporting down-stream prediction tasks (node/graph classification, community detection, dense retrieval, etc.)

Graph Node Embedding

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



How to learn the mapping function f

Poll 2

True or False

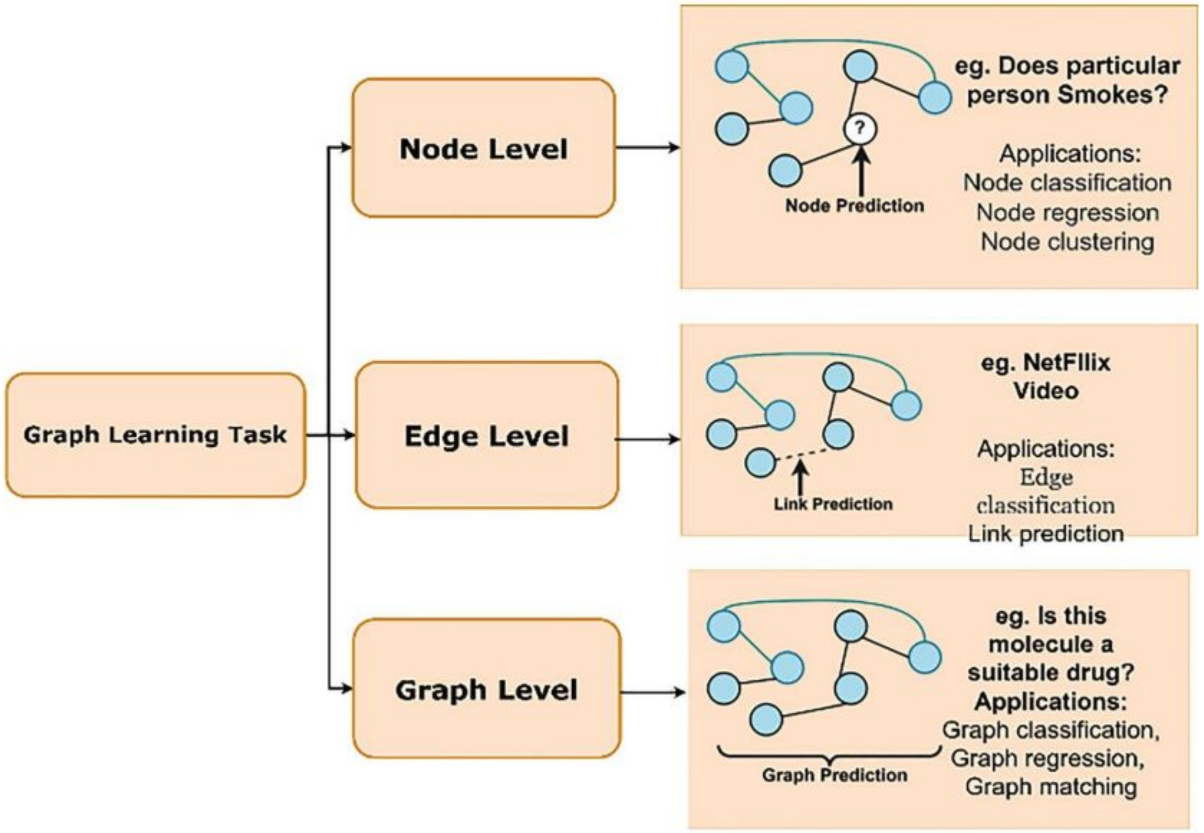
Node embeddings aim to map nodes in a graph to a continuous vector space while preserving their structural and semantic properties.

Poll 2

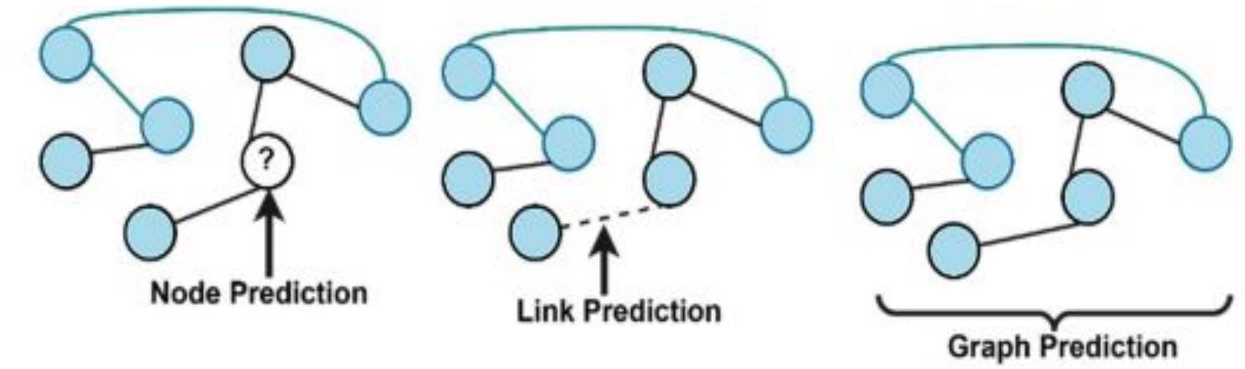
True or False

Node embeddings aim to map nodes in a graph to a continuous vector space while preserving their structural and semantic properties. **(True)**

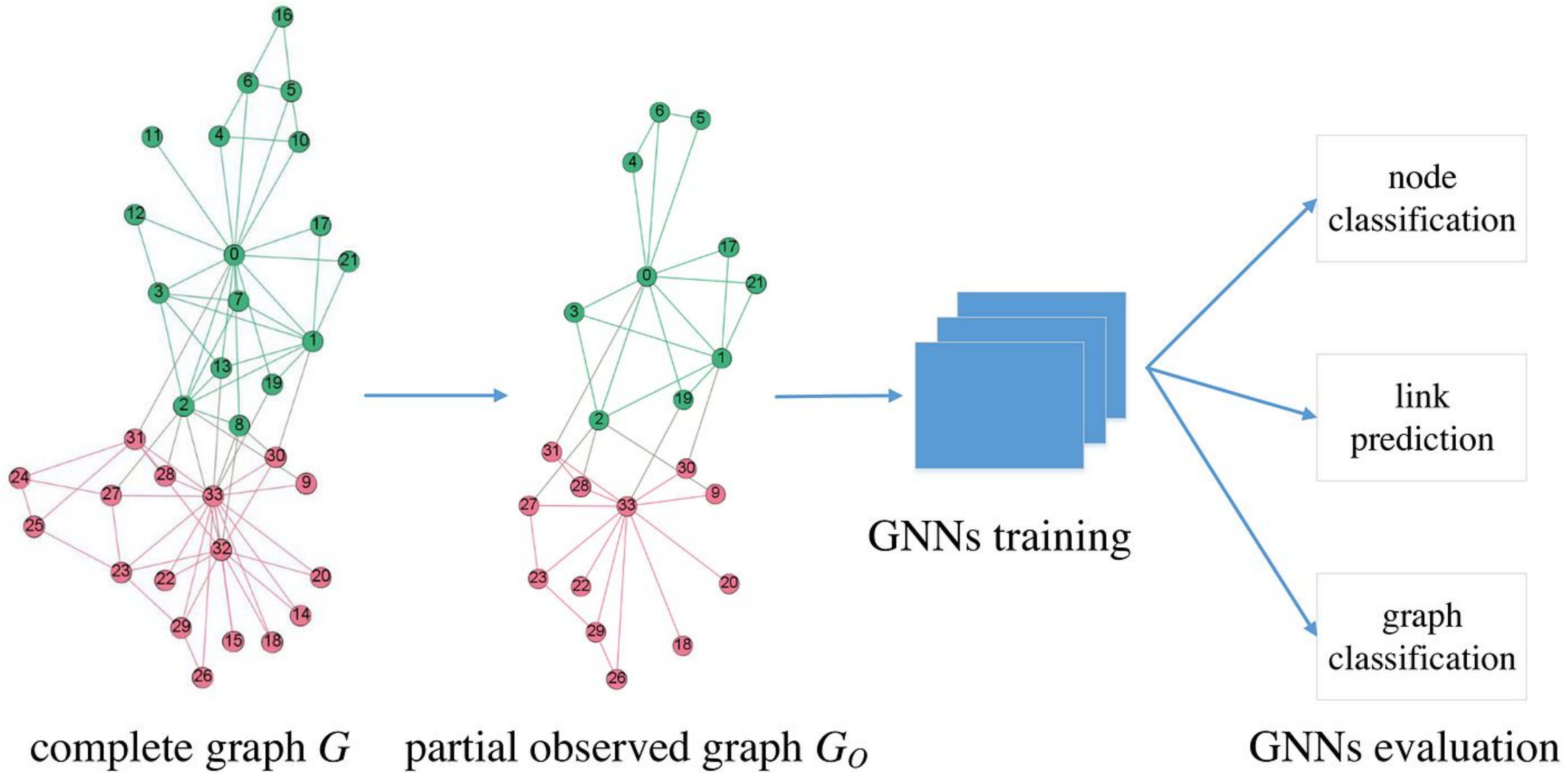
Graph Learning Task



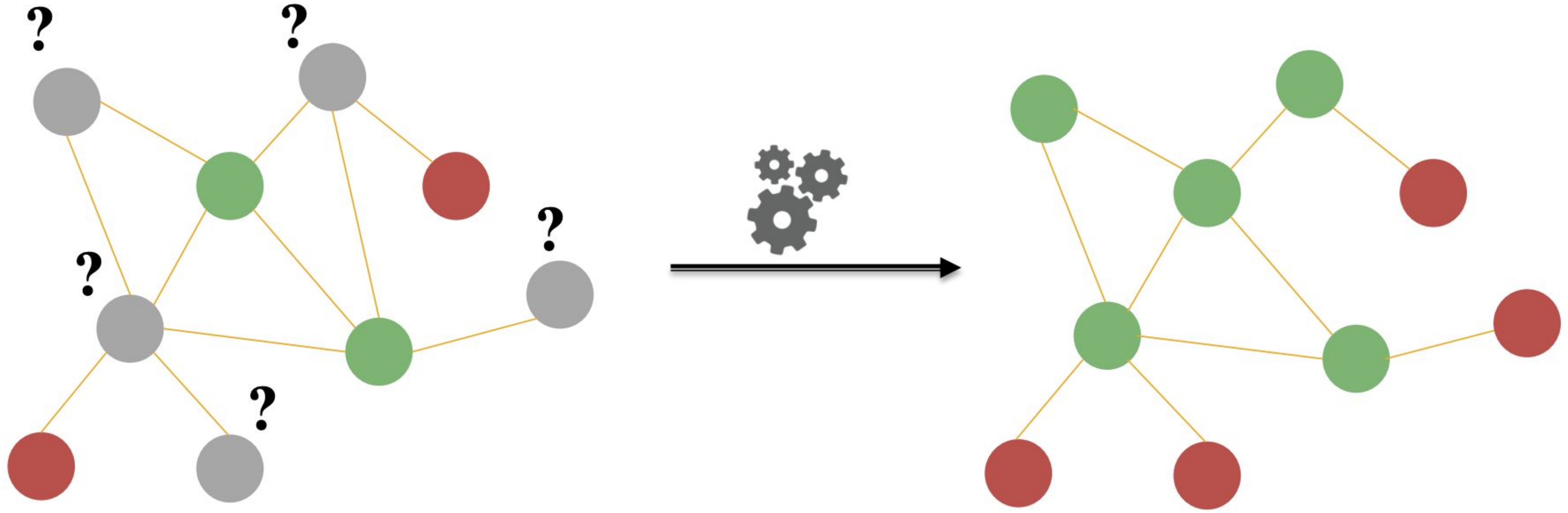
Graph Learning Task



Overview

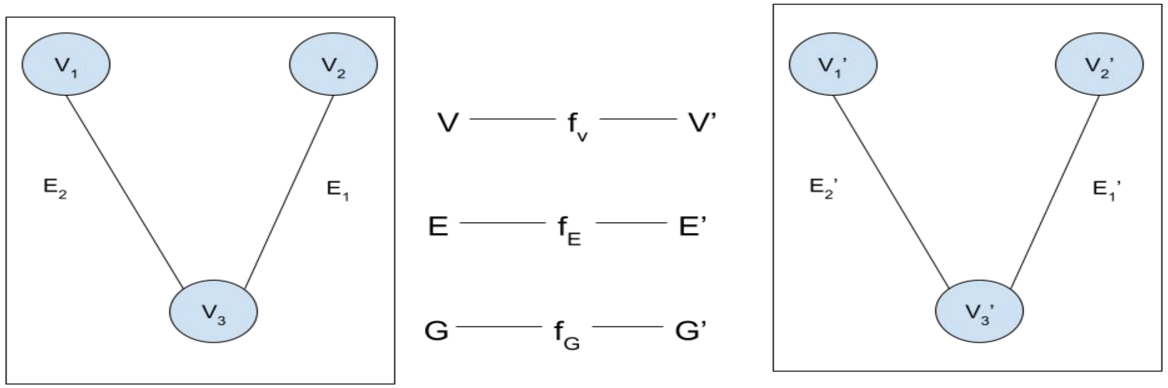

complete graph G
partial observed graph G_O
GNNs training
node classification
link prediction
graph classification
GNNs evaluation

Node Classification



Using an MLP Node Level Classification

V = Nodes, E = Edges, G = Graph



$f_V = MLP(V)$, $f_E = MLP(E)$, $f_G = MLP(G)$

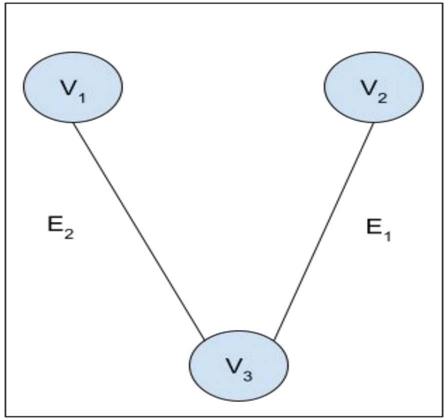
$$\begin{tikzcd} G(V,E) \arrow[r, dash] & \sigma[f(G(V,E))] \arrow[r, dash] & G(V',E') \end{tikzcd}$$

Apply a linear classifier to the embeddings (node, edge,graph)

Train the classifier using variation of SGD, with gradients calculated using backpropagation

Using an MLP Node Level Classification

Information Stored in Nodes, we want to classify V_1, V_2, V_3



$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} = f \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix}$$

F = MLP (Linear Classifier)

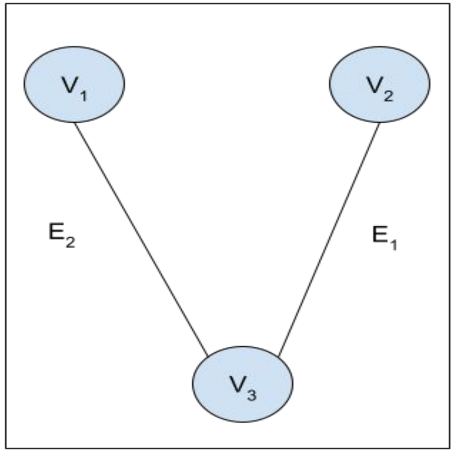
Apply a linear classifier to the embeddings (node)

Train the classifier using variation of SGD

Gradients calculated with backpropagation

Using an MLP Node Level Classification

Information Stored in **Edges**, we still want to classify V_1, V_2, V_3



1. Pool and aggregate information from edges to form node embeddings

$$V'_n = \rho_{E_n \rightarrow V_n}$$

$$\rho_{E_n \rightarrow V_n} = \text{AGG}\left(\forall\ E_i \in E(V_i)\right)$$

E (v) = Edges connected to node, v

Example, $E(v_3) = (E_1, E_2)$

AGG = Sum, Mean, Max, Min, etc)

2. Now apply linear classifier to V_n' to determine classes

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = f \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Train the classifier using variation of SGD
Gradients calculated with backpropagation

Edge Level Prediction

If Information stored in Edges

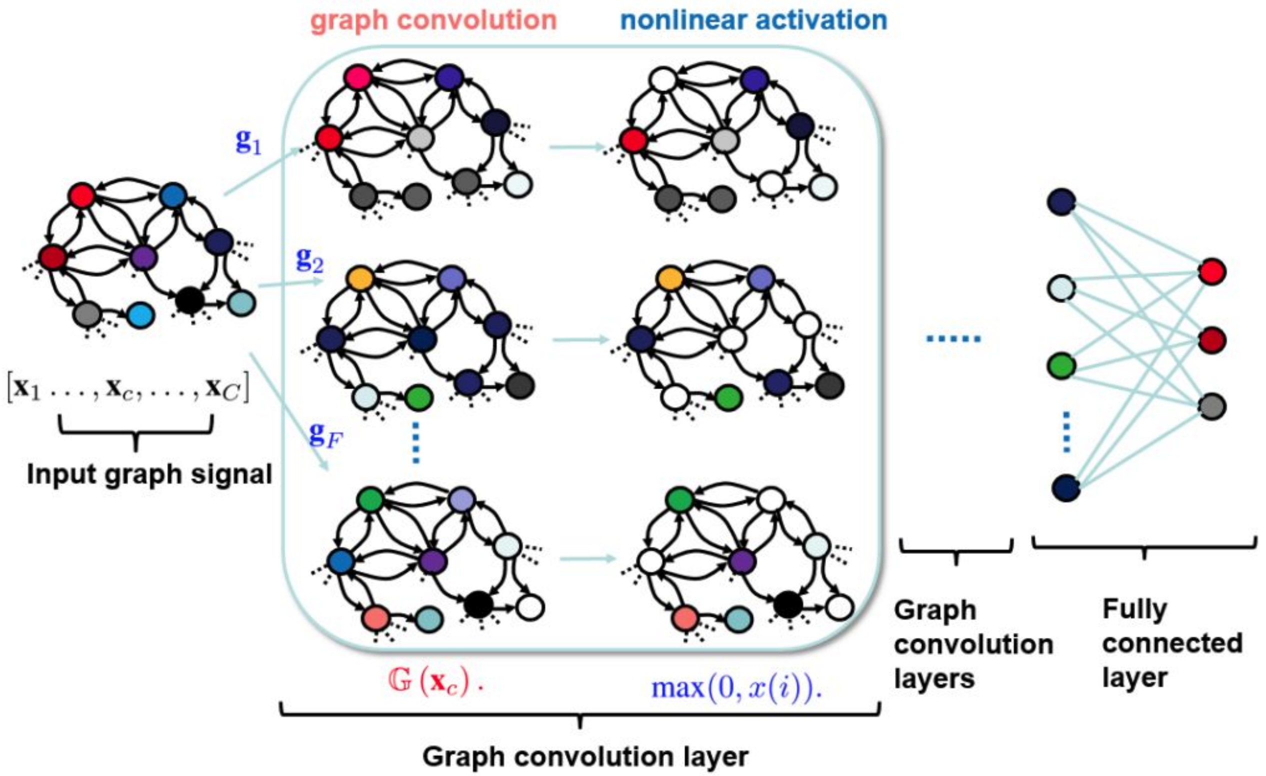
- Use MLP on edge embeddings

If Information is stored in Nodes

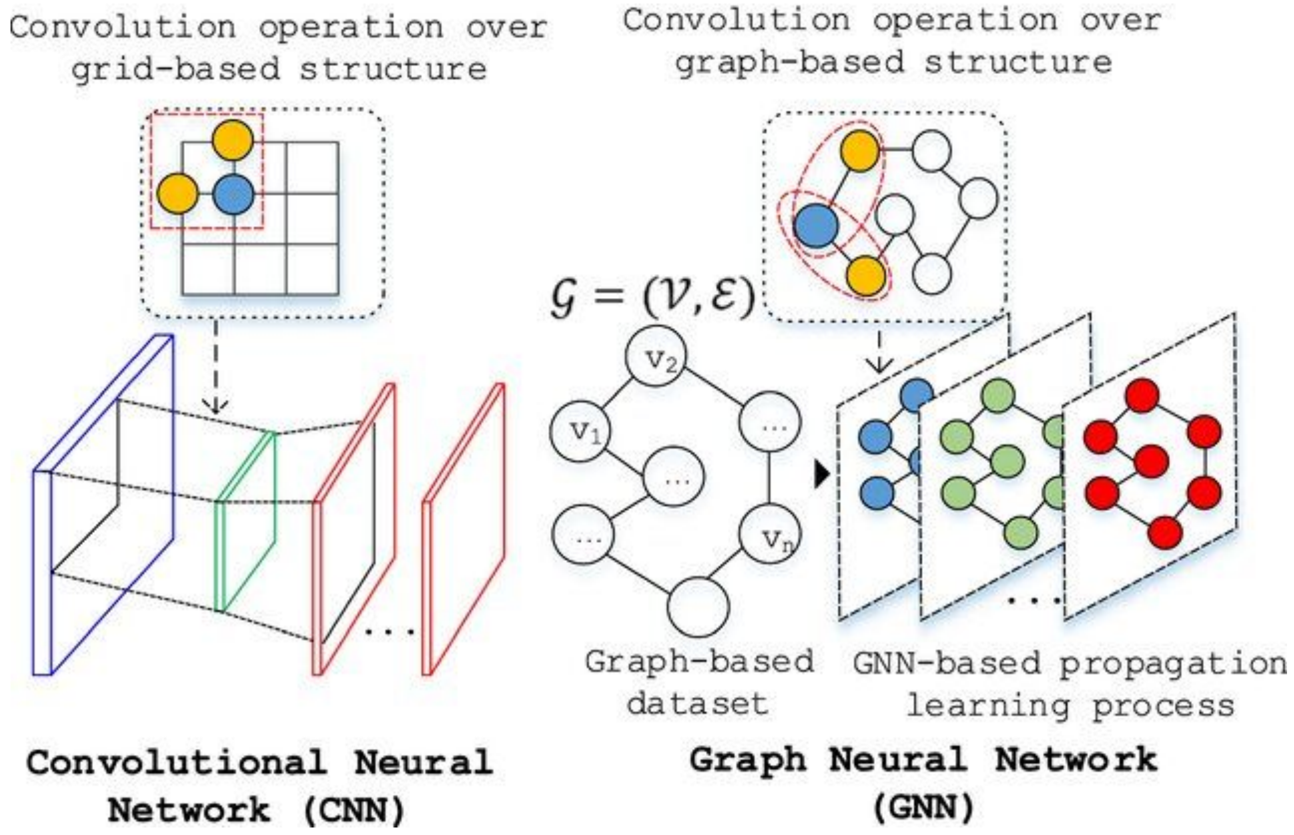
- Pool neighboring node embeddings
- Aggregate them to form new edge embeddings
- Use MLP on new edge embeddings

Can we generalize this ?

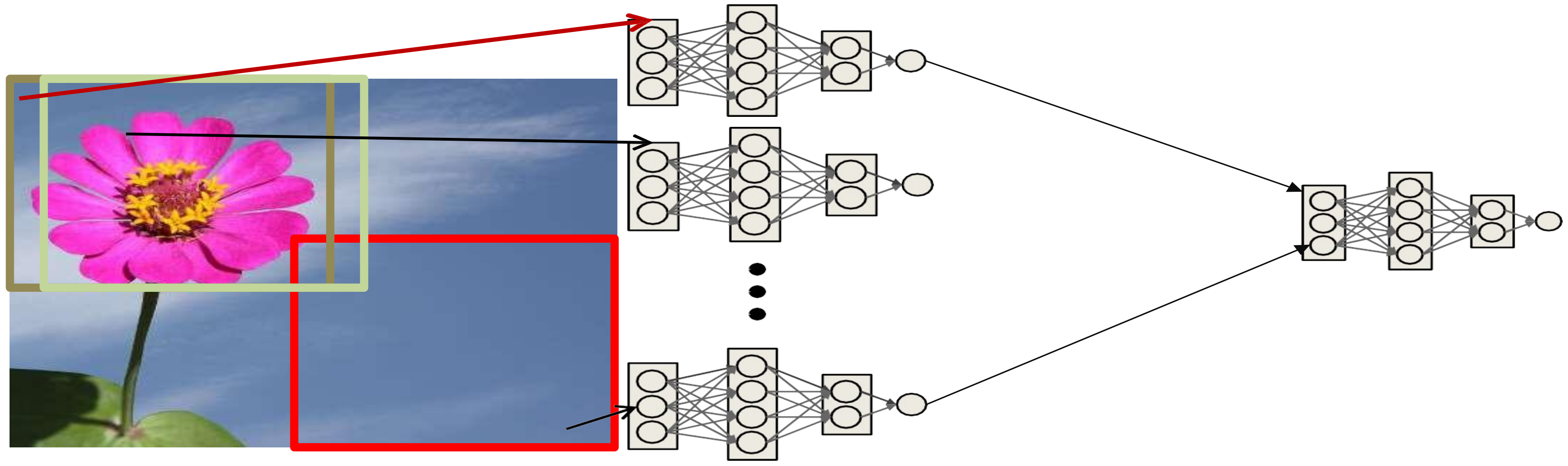
Graph Covulutional Network



Graph Convolution Vs CNN


Convolution operation over grid-based structure
Convolution operation over graph-based structure
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$
v_2
v_1
v_n
Graph-based dataset
GNN-based propagation learning process
Convolutional Neural Network (CNN)
Graph Neural Network (GNN)

A step back at CNNs



Convolutions process data by aggregating information from a fixed local neighborhood of pixels using filters (kernels).

Assumption: Data lies on a regular Euclidean grid, where neighboring pixels are equidistant and uniformly connected.

Graph Convolution

Three stage process.

1. Message Passing: Each node sends its features to its neighboring nodes, as defined by the graph's edges.

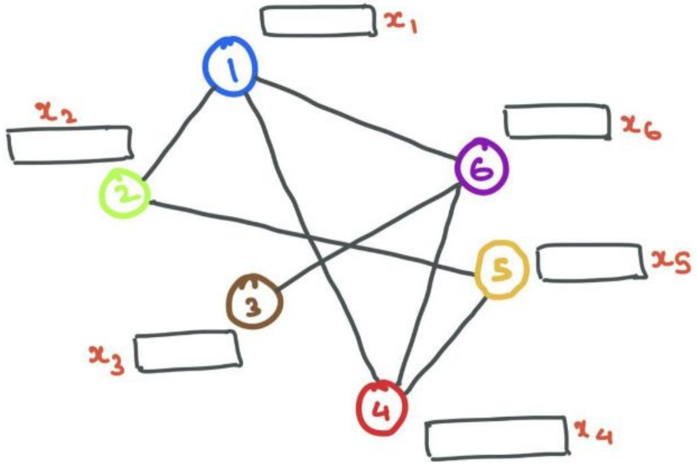
2. Aggregation: Each node collects and combines the features received from its neighbors (e.g., via sum, mean, or max).

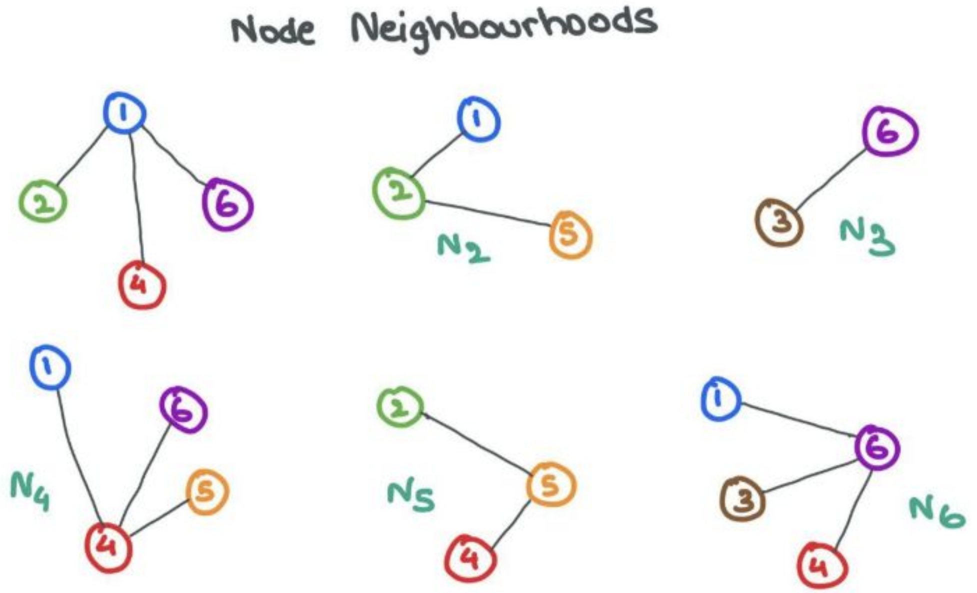
3. Update: Each node updates its feature representation by applying a transformation (e.g., using a neural network layer) to the aggregated features.

1: Message Passing

The Neighbourhood N_i of a node i is defined as the set of nodes j connected to i by an edge. Formally,

$$N_i = \{j : e_{ij} \in E\}.$$

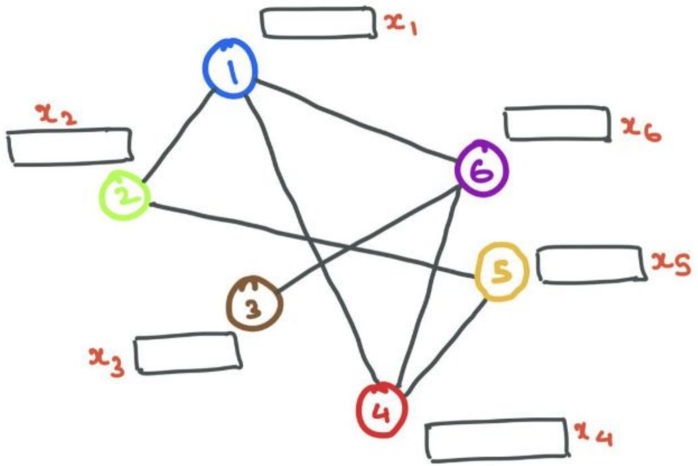


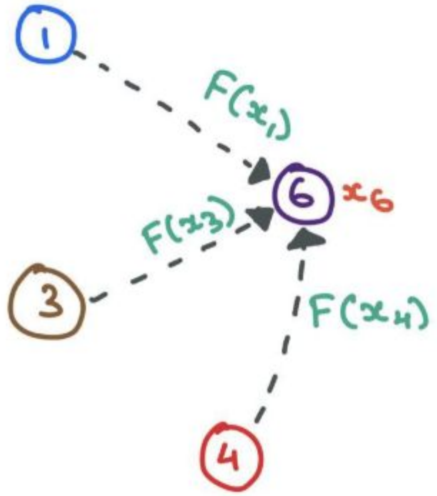


1: Message Passing (Message Creation)

Zooming to Node 6 with neighbors $\{1, 3, 4\}$, we transform each of the node features using a function F, which could just be an MLP or an affine transform:

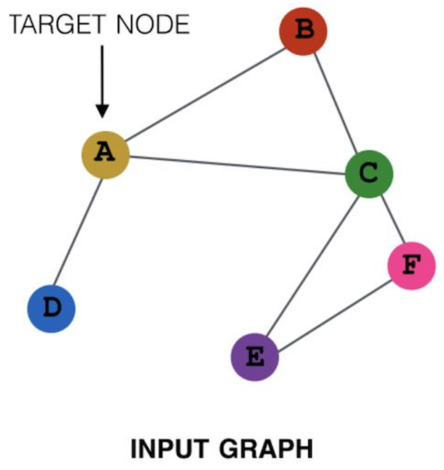
$$F(x_j) = W_j \cdot x_j + b.$$

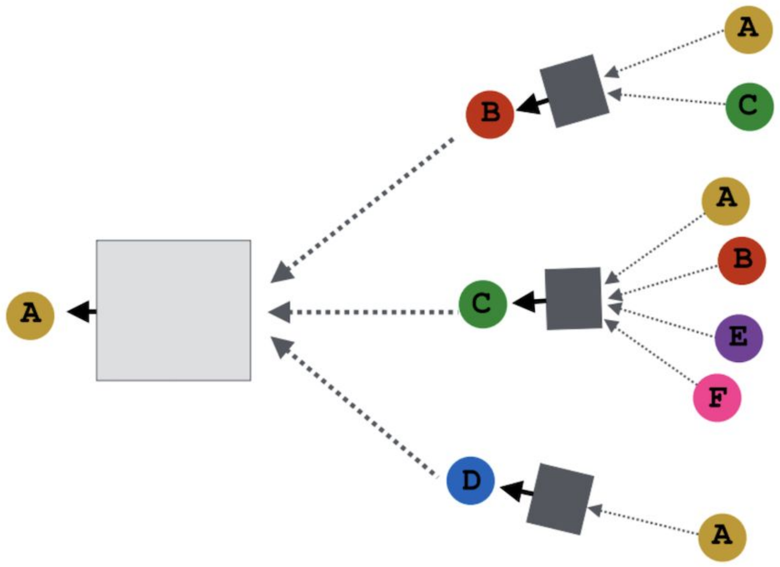




2: Aggregation Step

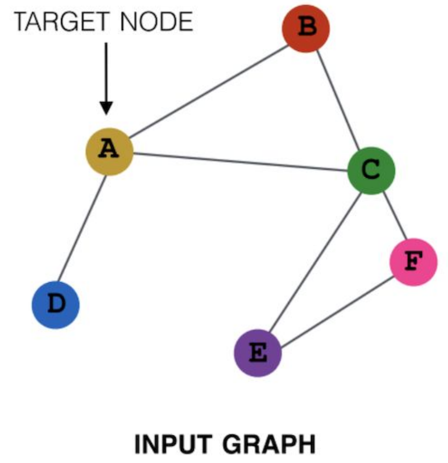
- Generate **node embeddings** based on local network **neighborhoods.**

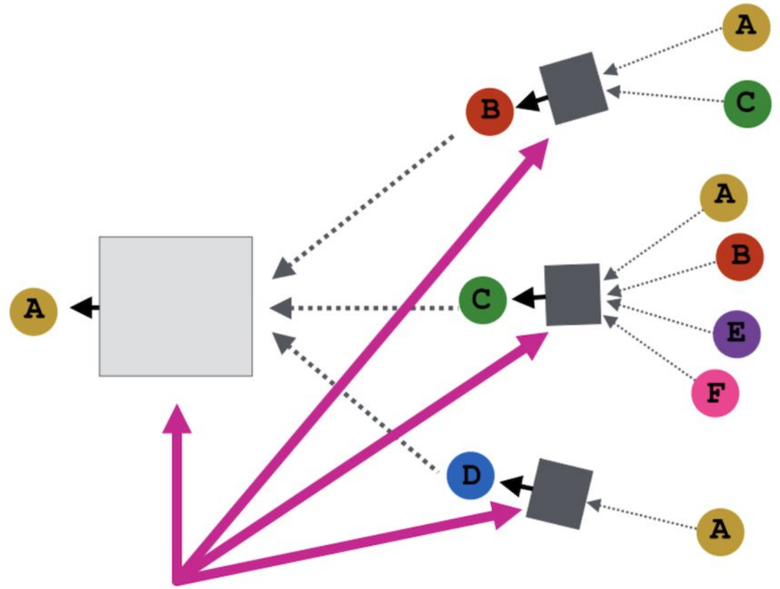




2: Aggregation Step

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

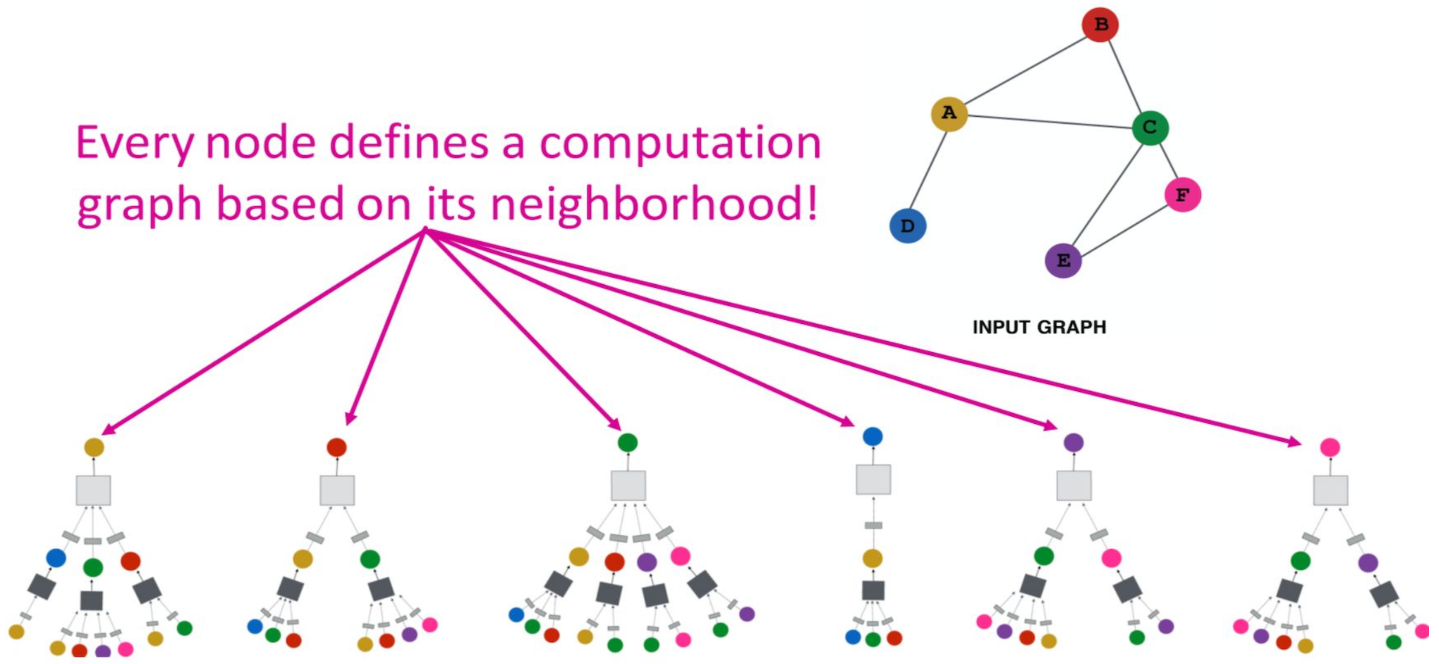




Neural networks

2: Aggregation Step

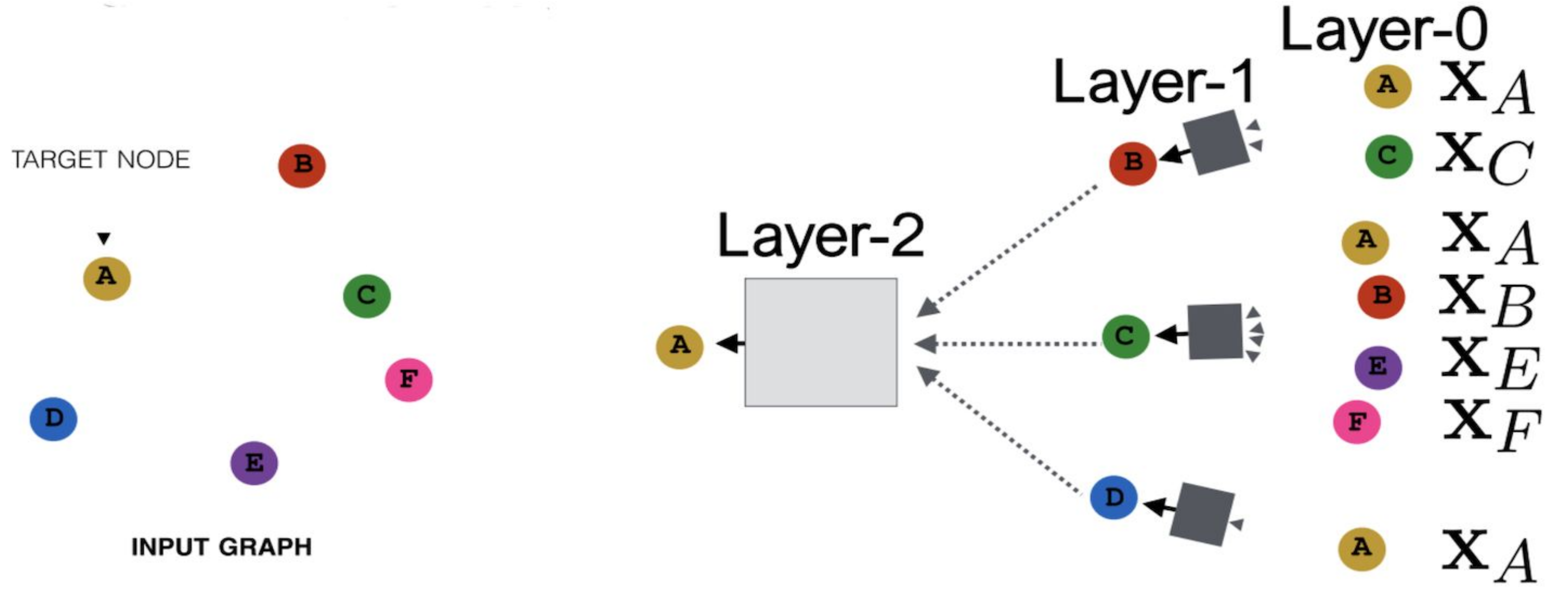
Network neighborhood defines a computation graph


Every node defines a computation graph based on its neighborhood!
B
A
C
F
D
E
INPUT GRAPH

Deep Model: Many Layers

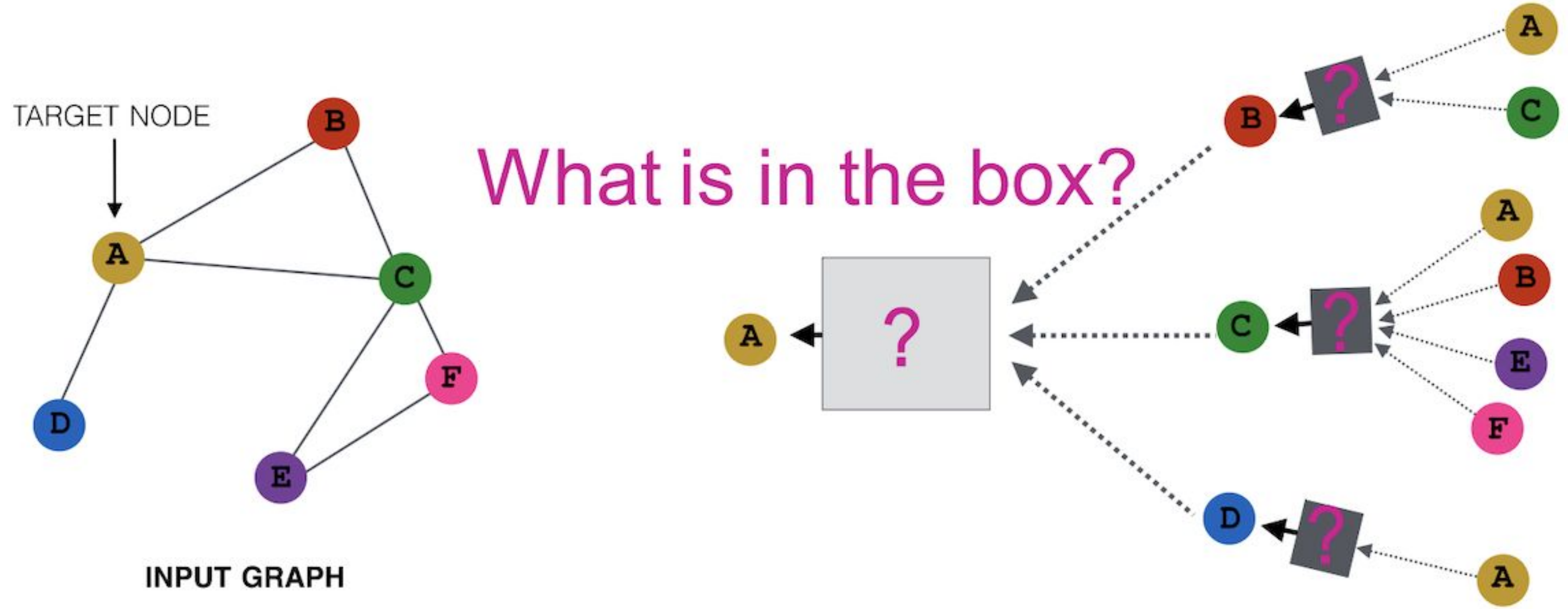
Model can be of arbitrary depth:

- 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



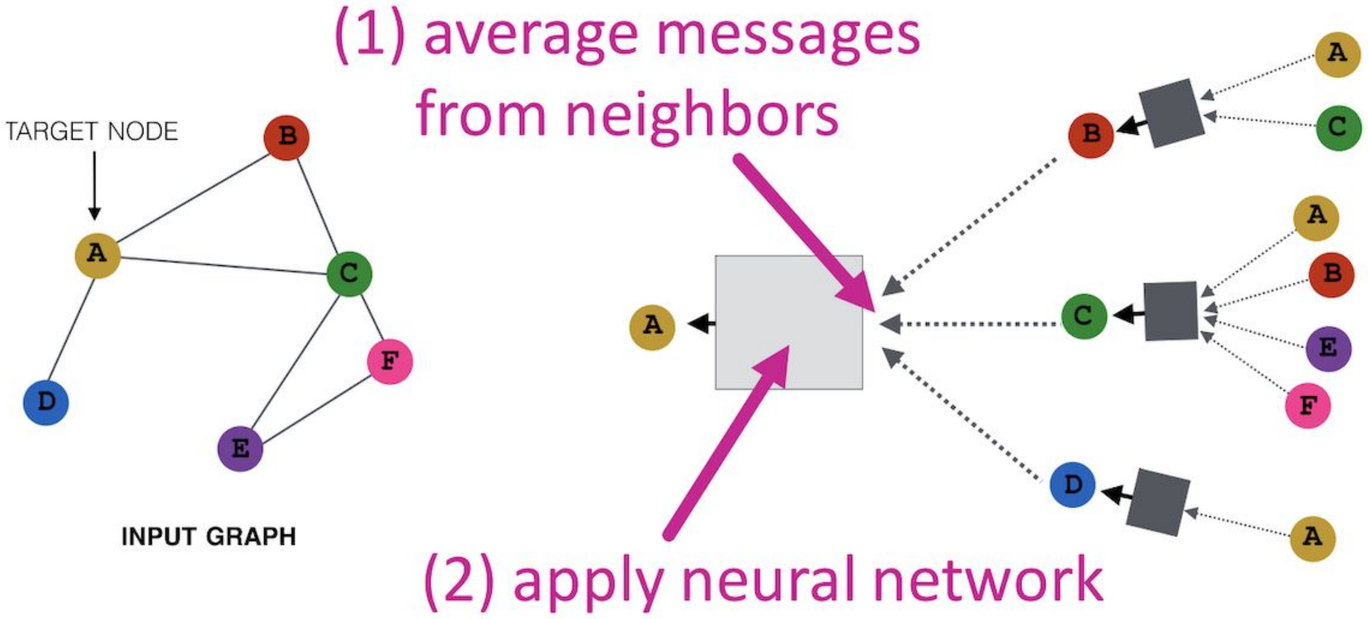
2: Aggregation Step

Neighborhood aggregation: Key distinctions are in how different approaches aggregate



2: Aggregation Step

Basic approach: Average information from neighbors and apply a neural network



3: Update Step

Basic approach: Average information from neighbors and apply a neural network

Initial 0-th layer embeddings are equal to node features

$$\boxed{\mathrm{h}_v^0 = \mathrm{x}_v}$$

embedding of v at layer k

$$\mathrm{h}_v^{(k+1)} = \boxed{\sigma}\left(\mathrm{W}_k \boxed{\sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(k)}}{|\mathrm{N}(v)|}} + \mathrm{B}_k \boxed{\mathrm{h}_v^{(k)}}\right), \forall k \in \{0, \ldots, \boxed{K} - 1\}$$

Total number of layers

$$\boxed{\mathrm{z}_v} = \mathrm{h}_v^{(K)}$$

Average of neighbor's previous layer embeddings

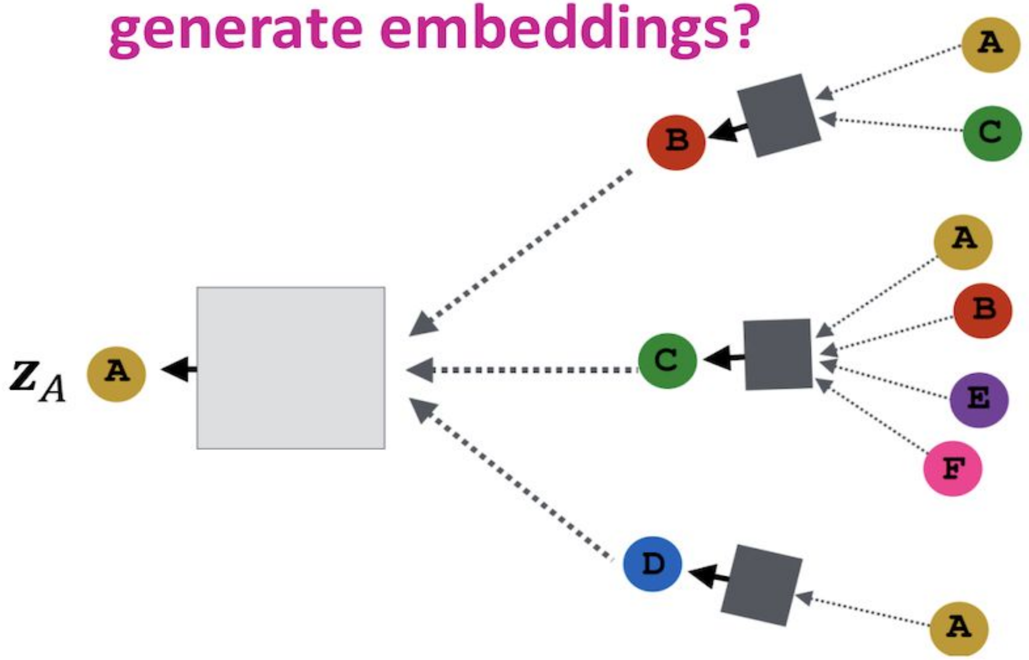
Embedding after K layers of neighborhood aggregation

Non-linearity (e.g., ReLU)

Notice summation is a permutation invariant pooling/aggregation.

Model Training

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings.

Model Training

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

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

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

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

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

Model Training

Node embedding z_v is a function of input graph

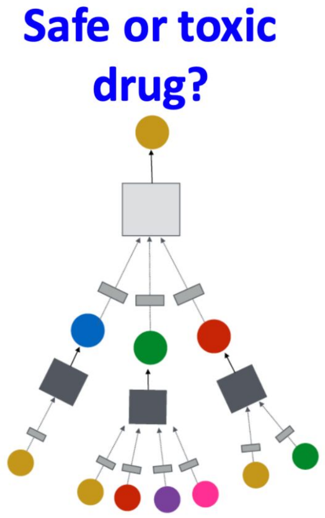
Supervised setting: We want to minimize the loss

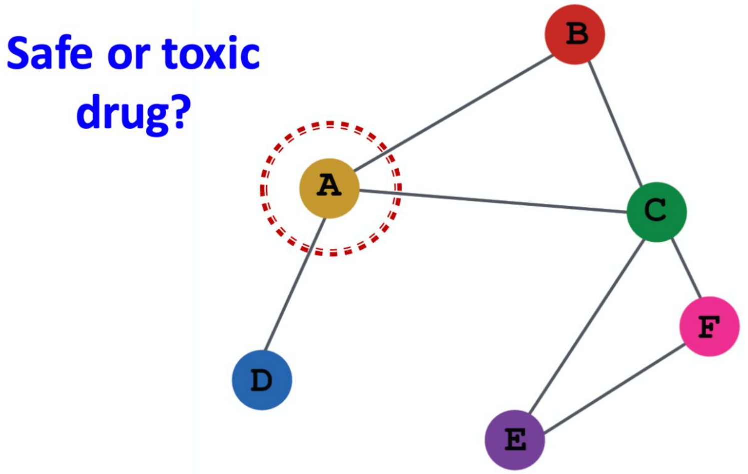
$$\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{z}_v))$$

- $\boldsymbol{y}$: node label
- $\mathscr{L}$ could be L2 if $\boldsymbol{y}$ is real number, or cross entropy if $\boldsymbol{y}$ is categorical

Model Training

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





E.g., a drug-drug interaction network

Model Training

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

Use cross entropy loss

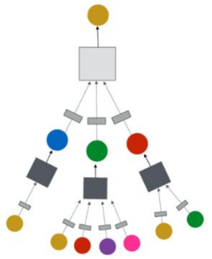
$$\mathcal{L} = -\sum_{v \in V} y_v \log(\sigma(\mathrm{z}_v^{\mathrm{T}} \theta)) + (1 - y_v) \log(1 - \sigma(\mathrm{z}_v^{\mathrm{T}} \theta))$$

Encoder output: node embedding

Classification weights

Node class label

Safe or toxic drug?



Classical GNN Layers: GraphSAGE

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

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

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

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

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

GraphSAGE Neighbor Aggregation

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

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

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

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

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

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

GraphSAGE: L2 Normalization

ℓ_2 Normalization:

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

Poll 3

True or False

A Graph Neural Network (GNN) using graph convolution can still be trained for edge-level prediction even if there is no information in the nodes

Poll 3

True or False

A Graph Neural Network (GNN) using graph convolution can still be trained for edge-level prediction even if there is no information in the nodes **(True)**

GAT: Graph Attention Networks

- (3) Graph Attention Networks

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

Attention weights

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

Classical GNN Layers : GAT

Graph Attention Networks

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

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.

Graph Attention Network

Can we do better than simple neighborhood aggregation?

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

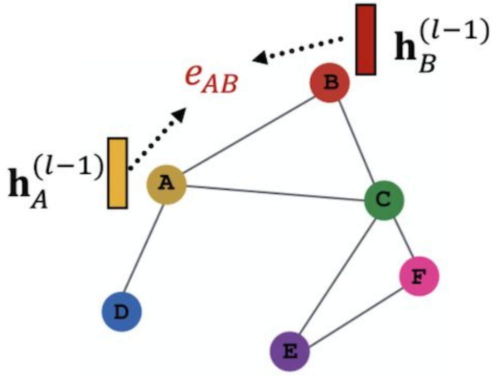
- **Goal:** Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea:** Compute embedding $\boldsymbol{h}_v^{(l)}$ of each node in the graph following an **attention strategy**:
  - Nodes attend over their neighborhoods' message
  - Implicitly specifying different weights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

Attention Mechanism

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

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

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



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

Attention Mechanism

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

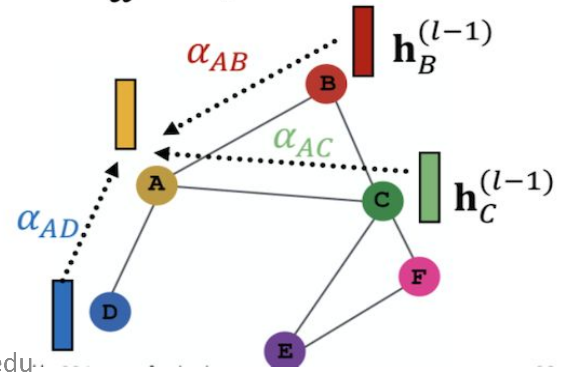
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

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

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

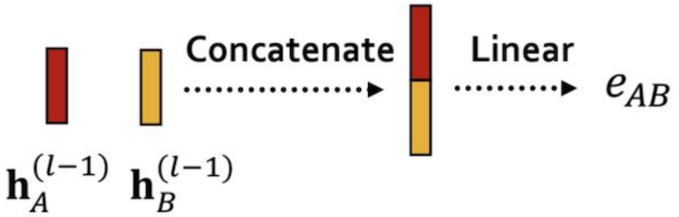
Weighted sum using α_{AB}, α_{AC}, α_{AD}:

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



Attention Mechanism

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



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

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

Attention Mechanism

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

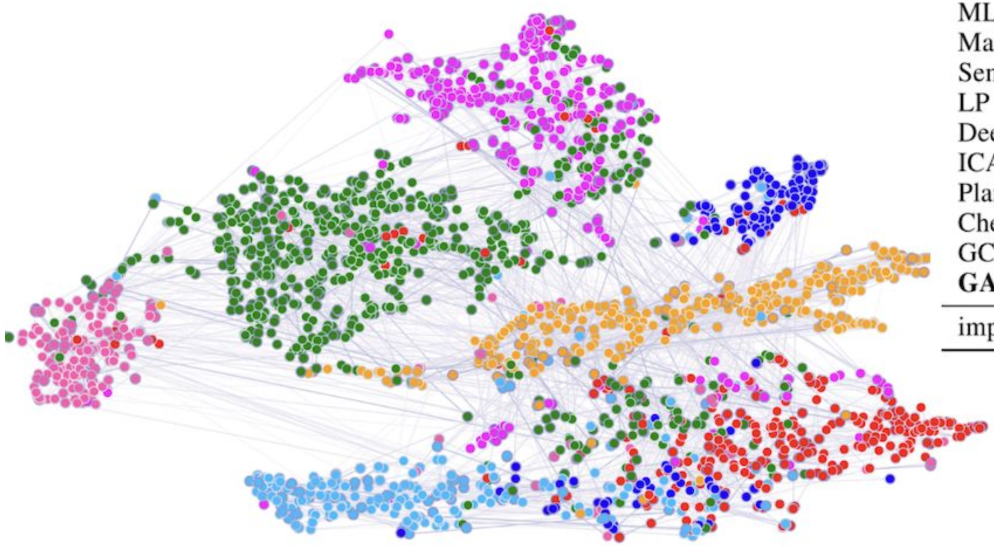
$$\mathbf{h}_v^{(l)}[1] = \sigma(\textstyle\sum_{u \in N(v)} \alpha_{vu}^{1} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$
$$\mathbf{h}_v^{(l)}[2] = \sigma(\textstyle\sum_{u \in N(v)} \alpha_{vu}^{2} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$
$$\mathbf{h}_v^{(l)}[3] = \sigma(\textstyle\sum_{u \in N(v)} \alpha_{vu}^{3} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

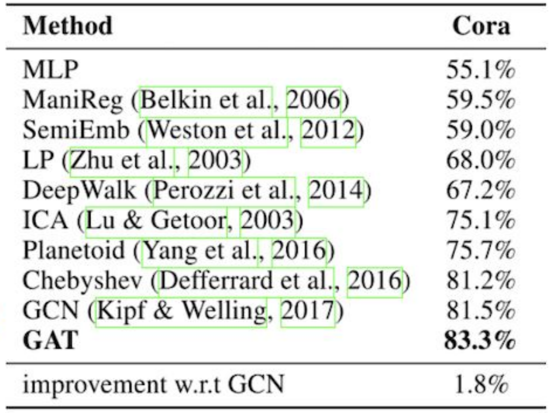
  - **Outputs are aggregated:**
    - By concatenation or summation
    - $\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$

Benefit 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

GAT Exemple: Core Citation Net





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%

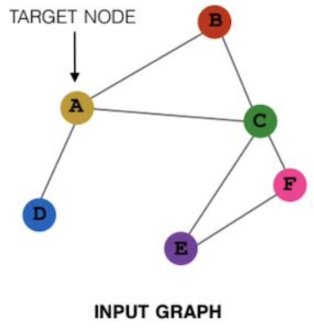
Attention mechanism can be used with many different graph neural network models

In many cases, attention leads to performance gains

t-SNE plot of GAT-based node embeddings:

- Node color: 7 publication classes
- Edge thickness: Normalized attention coefficients between nodes i and j, across eight attention heads, $\sum_k(\alpha_{ij}^k + \alpha_{ji}^k)$

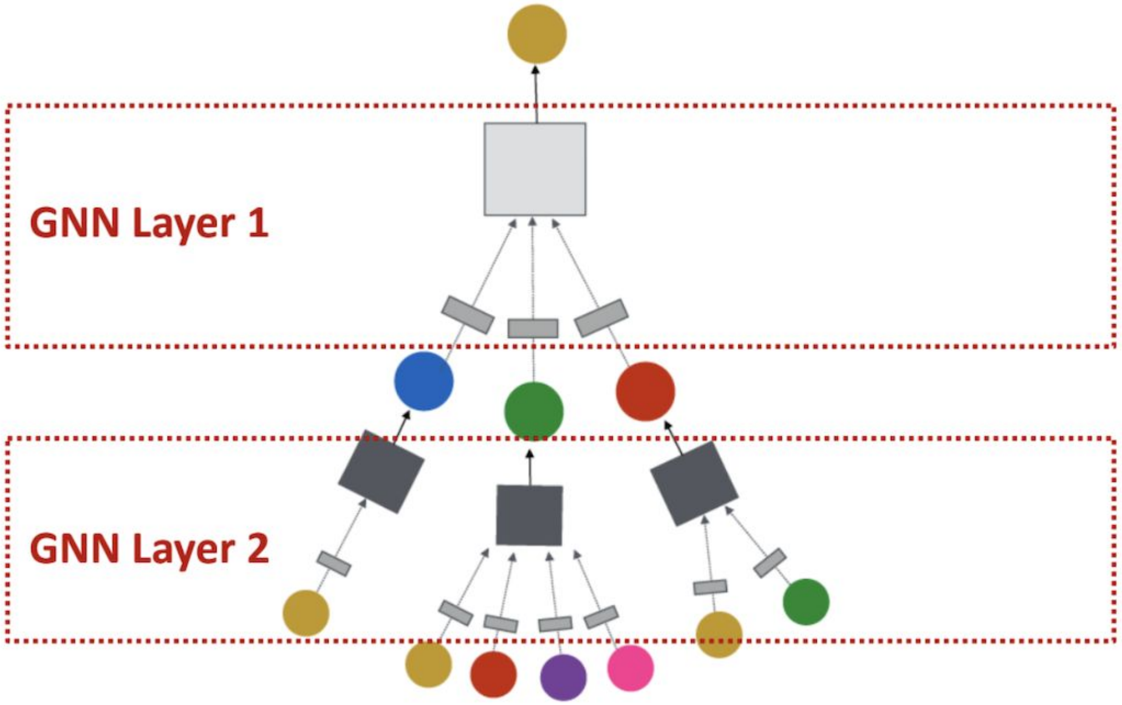
Stacking GNN Layers



How to connect GNN layers into a GNN?

- Stack layers sequentially
- Ways of adding skip connections

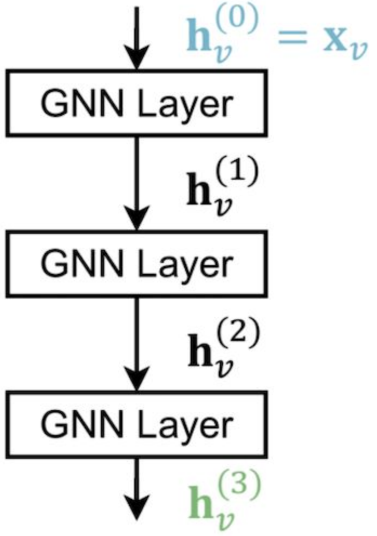
(3) Layer connectivity



Stacking GNN Layers

How to construct a Graph Neural Network?

- **The standard way:** Stack GNN layers sequentially
- **Input:** Initial raw node feature $\mathbf{x}_v$
- **Output:** Node embeddings $\mathbf{h}_v^{(L)}$ after L GNN layers



In summary

- Traditional Neural Networks types can be used in various learning tasks,
- However it does not work well for all types of data,
- Graph Neural Networks can help in such a situation where we rely on relationships between entities (eg: Social Network, Drug Discovery),
- **GNN, GCN, GraphSAGE, GAT** etc
- General techniques for model training are for GNN
    - Dropout, Feature Augmentation or Structure Augmentation (Virtual Nodes or edges, Sample neighbors when, doing message passing etc)