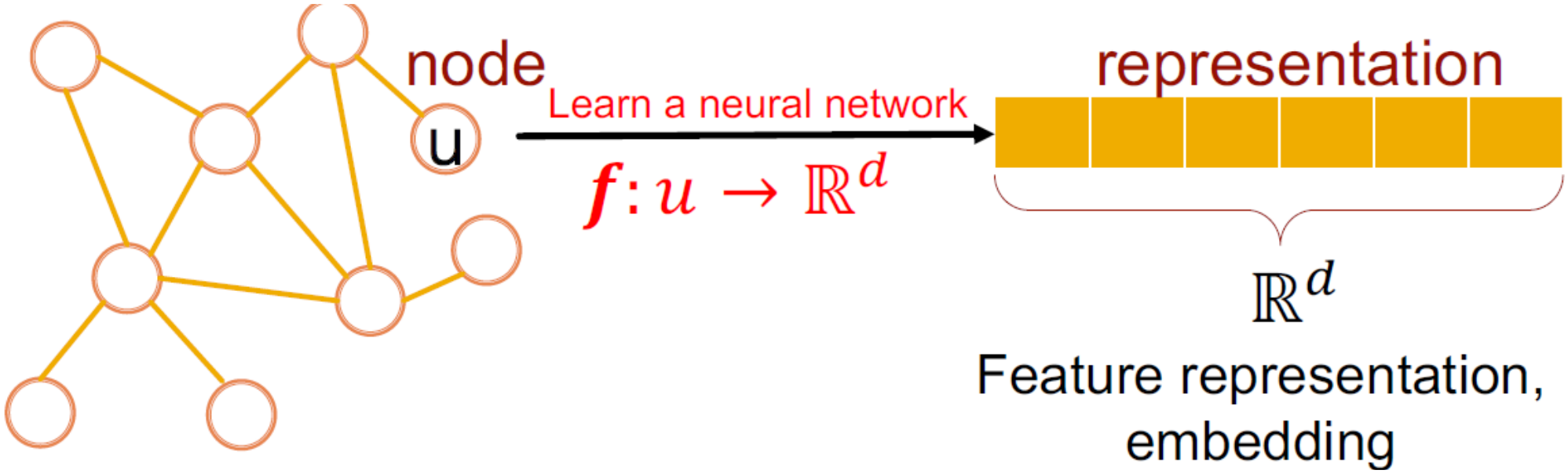
DS 503: Advanced Data Analytics

Lecture 18: Graph Learning

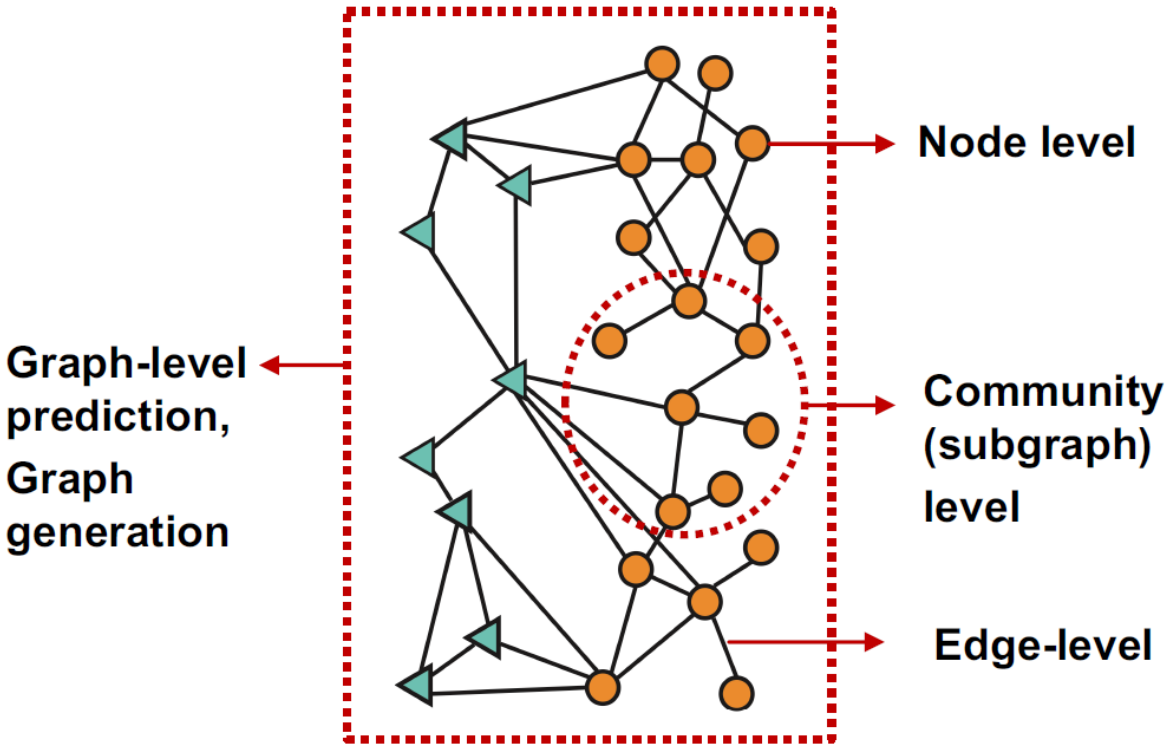
Gagan Raj Gupta

Recap: Representation Learning

- (Supervised) Machine Learning Lifecycle: Retrain for every new feature. Every single time!
- Representation Learning – Automatically learn the features
- Map nodes to d-dimensional **embeddings** such that similar nodes in the network are embedded close together



Graph ML Tasks



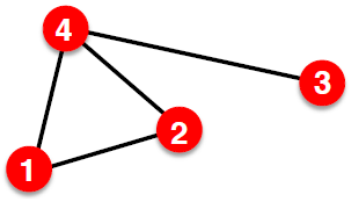
- **Node classification**: Predict a property of a node
  - **Example:** Categorize online users / items
- **Link prediction**: Predict whether there are missing links between two nodes
  - **Example:** Knowledge graph completion
- **Graph classification**: Categorize different graphs
  - **Example:** Molecule property prediction
- **Clustering**: Detect if nodes form a community
  - **Example:** Social circle detection
- **Other tasks**:
  - **Graph generation**: Drug discovery
  - **Graph evolution**: Physical simulation

Motivation

- Graph Representation Learning alleviates the need to do feature engineering every single time
- Today, we will look at two different ways to learn representations of nodes
  - Node Embeddings (based on Random Walks)
    - Preserve similarity of nodes
  - Graph Neural Networks (based on message passing)

Setup

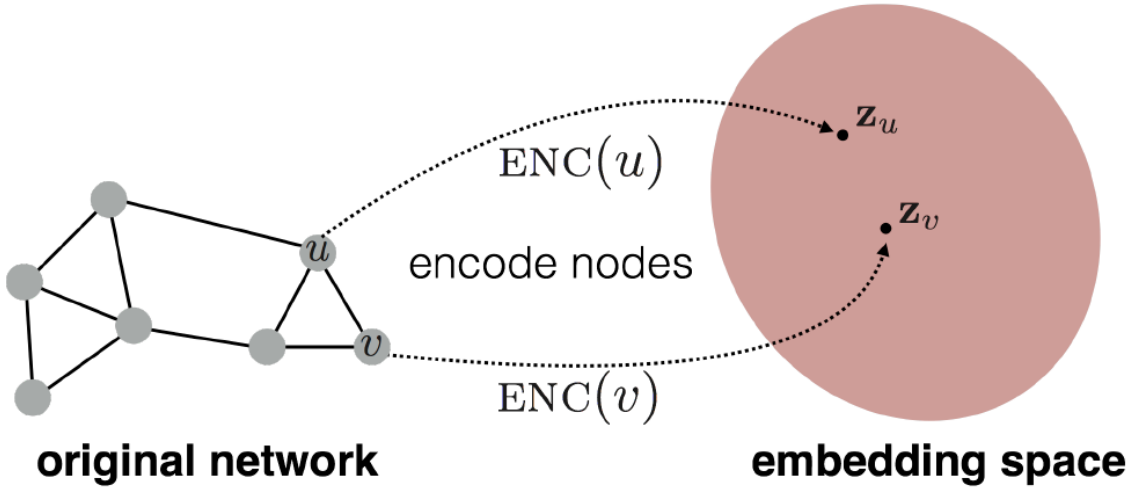
- Assume we have a Graph G and adjacency matrix A



V: {1, 2, 3, 4}

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

- Goal is to encode nodes so that similarity in embedding space, approximates similarity in the graph



$$\underbrace{\text{similarity}(u, v)}_{\text{in the original network}} \approx \underbrace{\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u}_{\text{Similarity of the embedding}}$$

Defining Node Similarity

- Should two nodes have a similar embedding if they…
  - are linked?
  - share neighbors?
  - have similar "structural roles"?
- **Unsupervised/self-supervised** way of learning node embeddings
  - We are **not** utilizing node labels or features
  - The goal is to directly estimate a set of coordinates (i.e., the embedding) of a node so that some aspect of the network structure (captured by DEC) is preserved
- These embeddings are **task independent**
  - They are not trained for a specific task but can be used for any task.
- We will now learn node similarity definition that uses **random walks**, and how to optimize embeddings for such a similarity measure

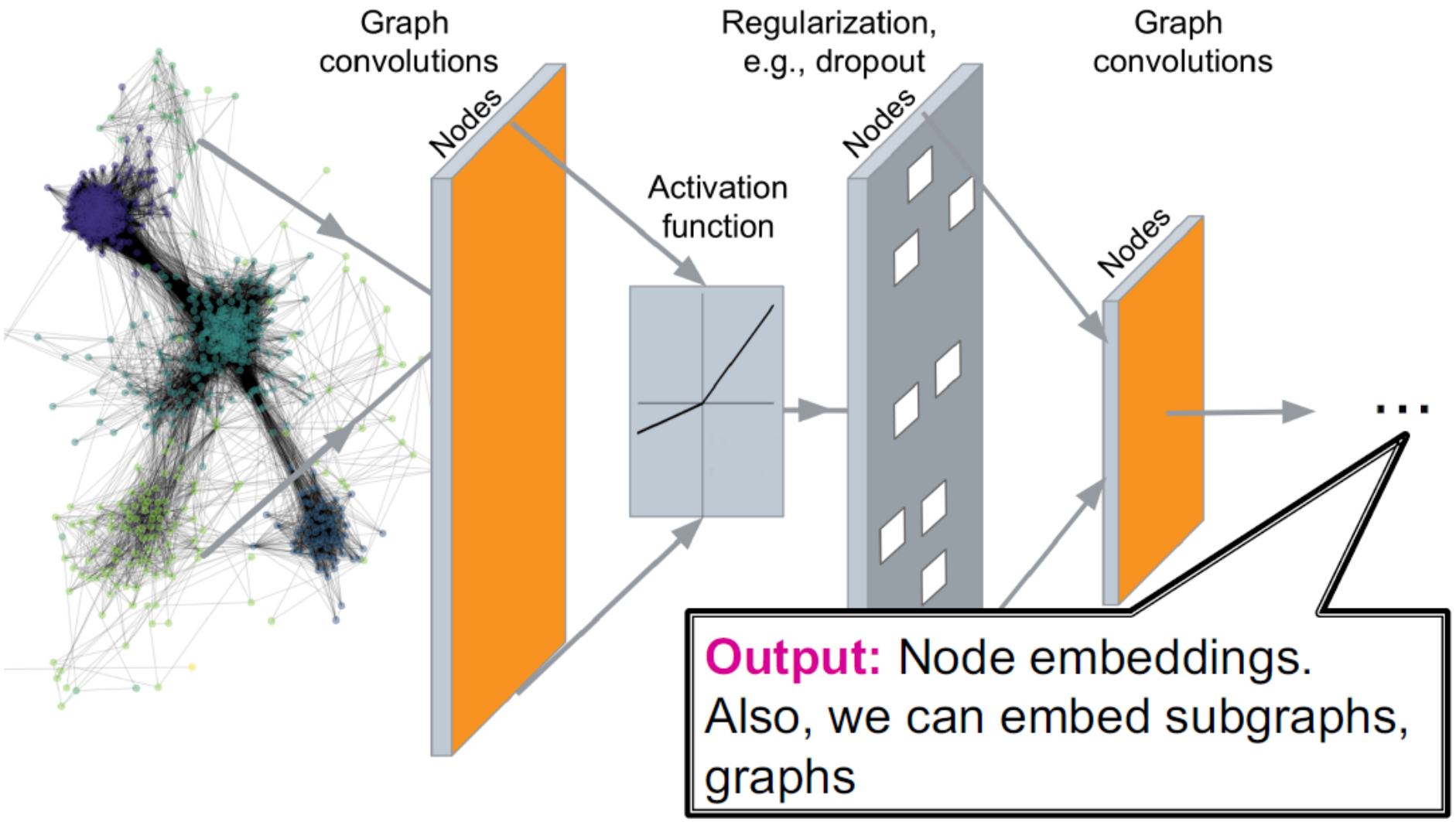
Random walk based Node Embeddings

- Lec 3 (pages 20-47)

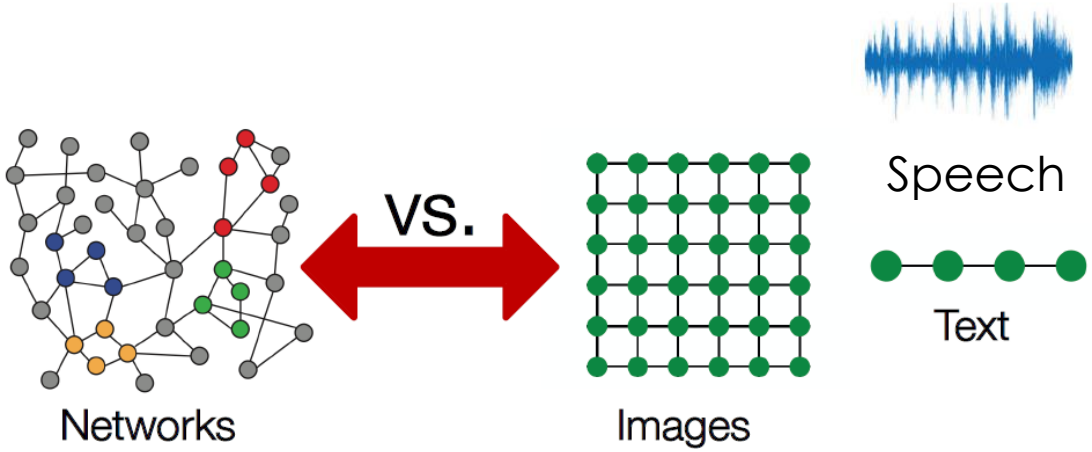
Limitations

- **Limitations** of shallow embedding methods:
- **$O(|V|)$ parameters are needed**:
  - No sharing of parameters between nodes
  - Every node has its own unique embedding
- **Inherently "transductive"**:
  - Cannot generate embeddings for nodes that are not seen during training
- **Do not incorporate node features**:
  - Many graphs have features that we can and should leverage

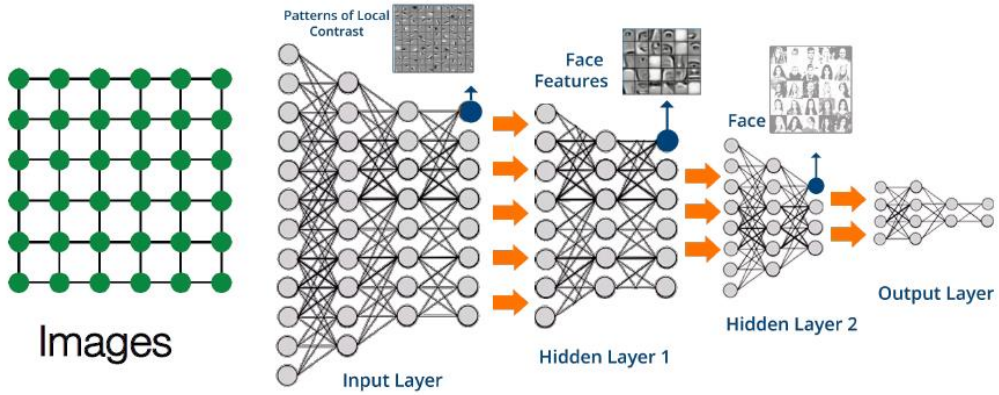
Deep Graph Encoders



Graph structured data

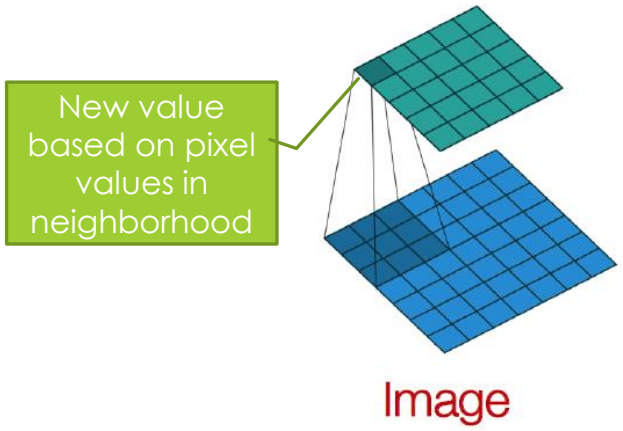


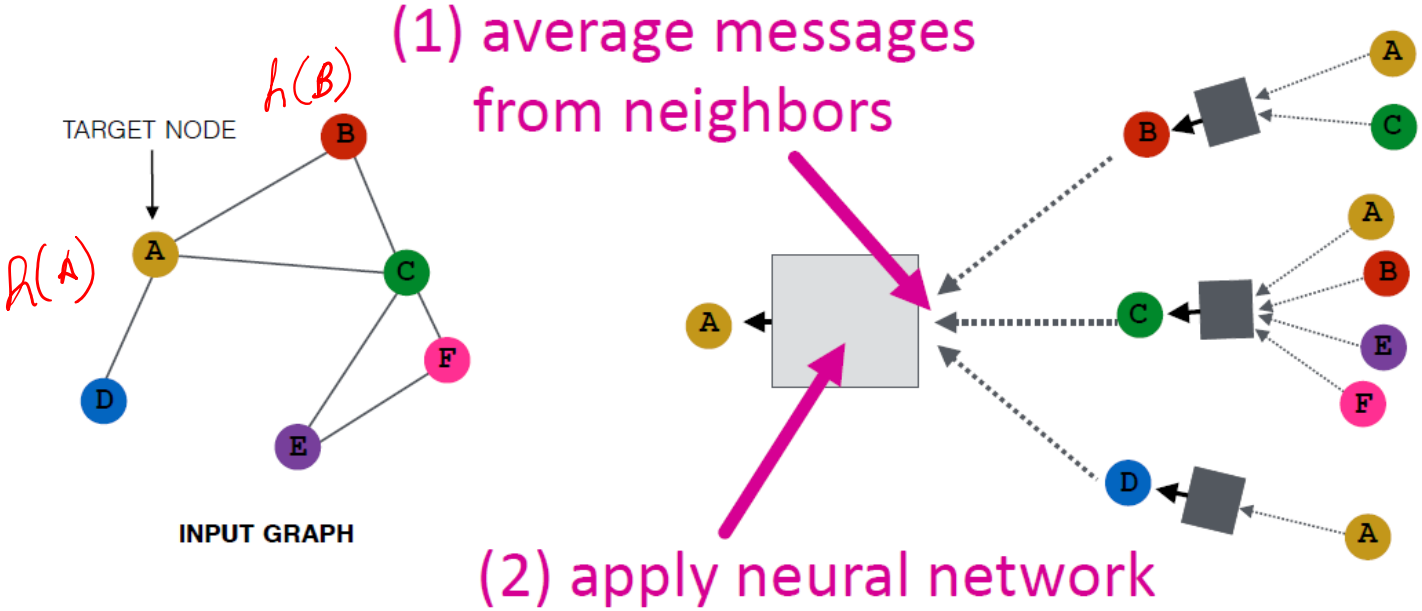
Modern ML Toolbox



- Modern deep learning toolbox is designed for sequences and grids.
- Graph data {V, E} is far more complex with arbitrary size and complex topological structure (i.e. no spatial locality like grids).
- There is no fixed ordering or reference point. Even simple looking problem of comparing two graphs for equality is NP hard.
- Graph data is often dynamic and have multi-modal features.
- Eg. 3D meshes, social networks, telecommunication, networks, biological networks or brain connectomes.

General form of GNNs

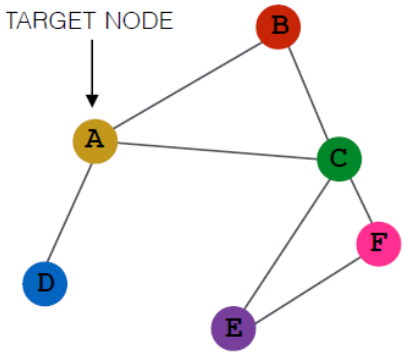




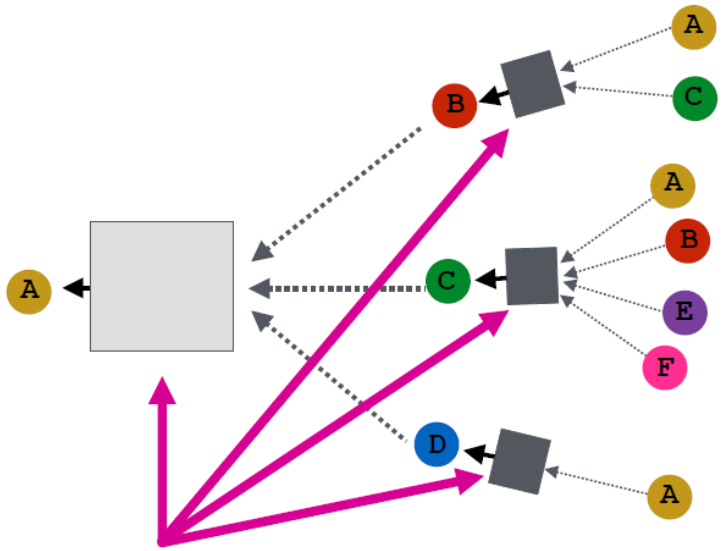
Graph neural networks are based on aggregating information (messages) from the node's neighbors and passing them through a neural network to compute/update the embedding of the node

Key Ideas

- **Key idea:** Generate node embeddings based on **local network neighborhoods**
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



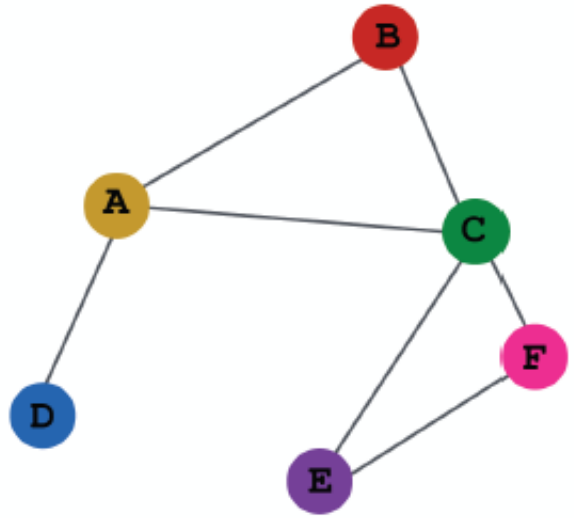
INPUT GRAPH



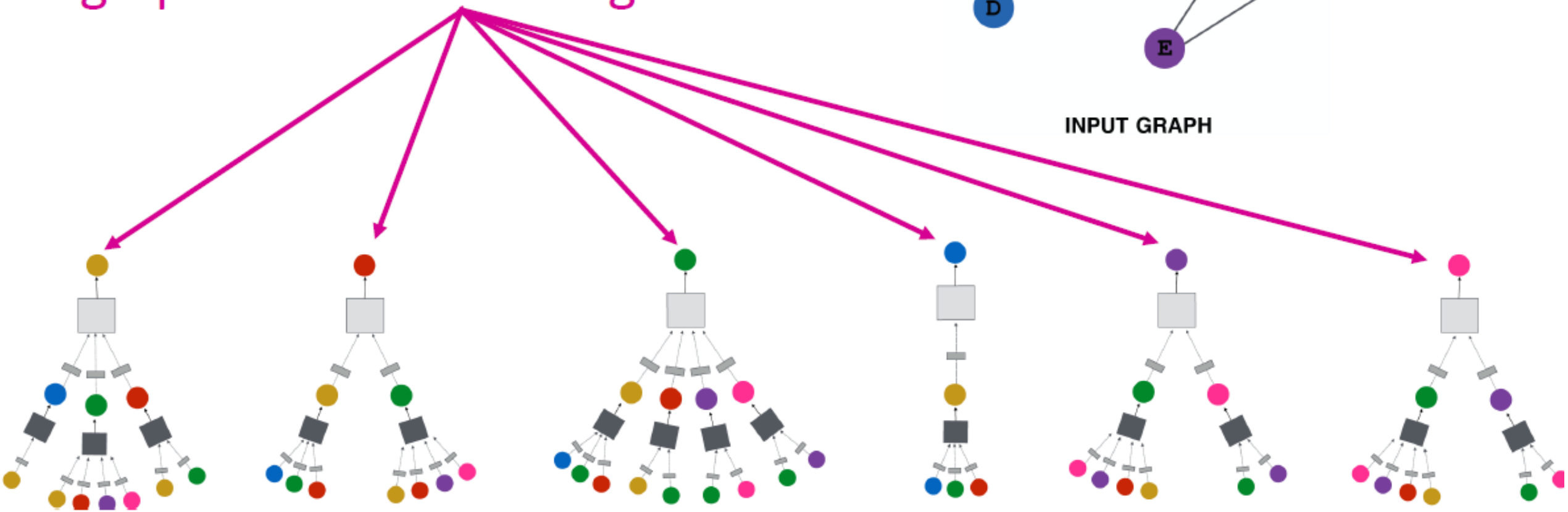
Neural networks

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

Every node defines a computation graph based on its neighborhood!

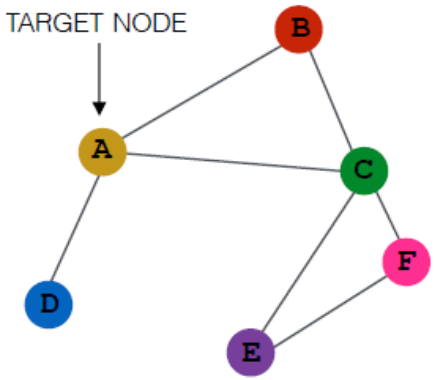


INPUT GRAPH

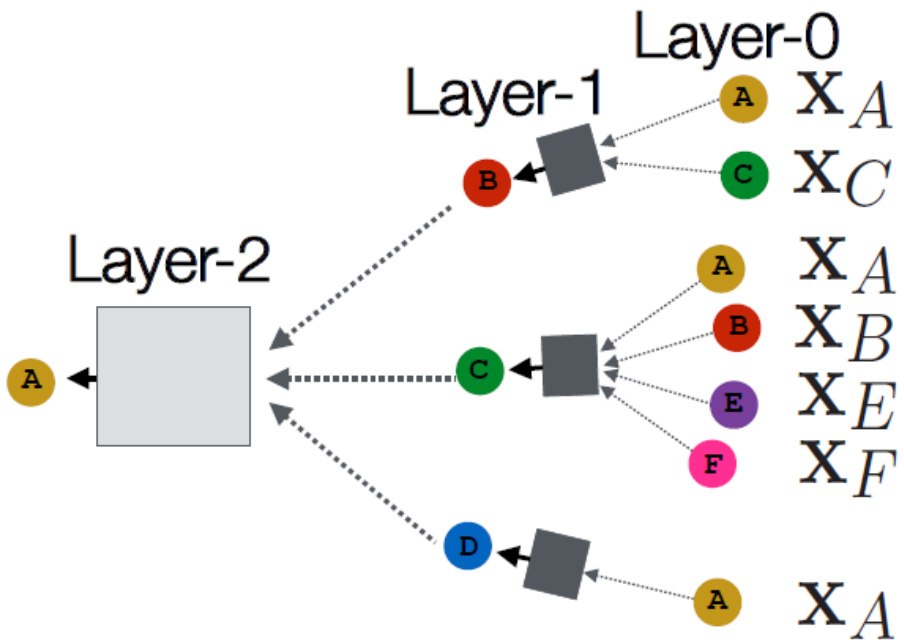


Deep Models

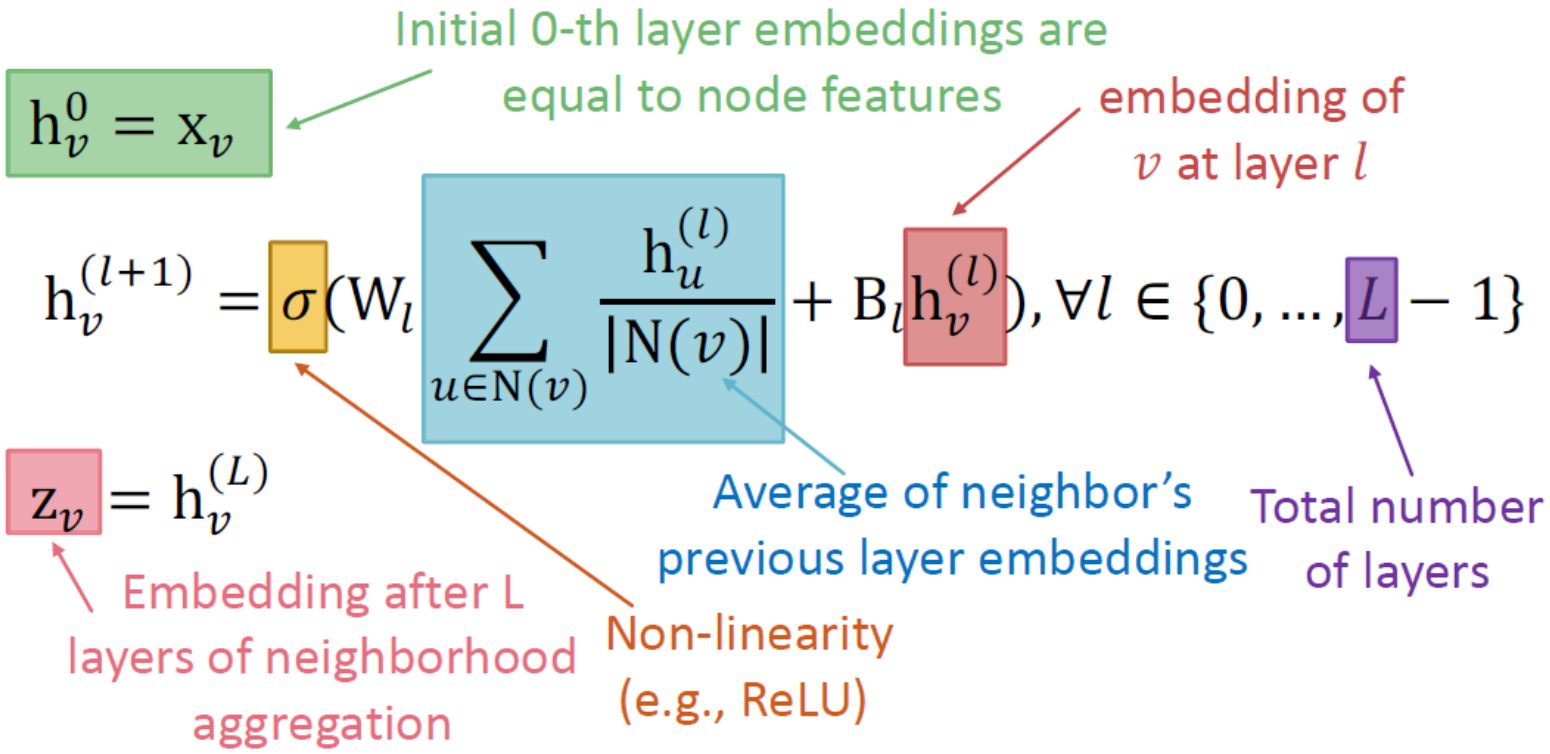
- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node u is its input feature, x_u
  - Layer-k embedding gets information from nodes that are K hops away



INPUT GRAPH



GNN and Graph Sage



Model parameters

W_k: Weight matrix for neighbors
B_k: Weight matrix for self

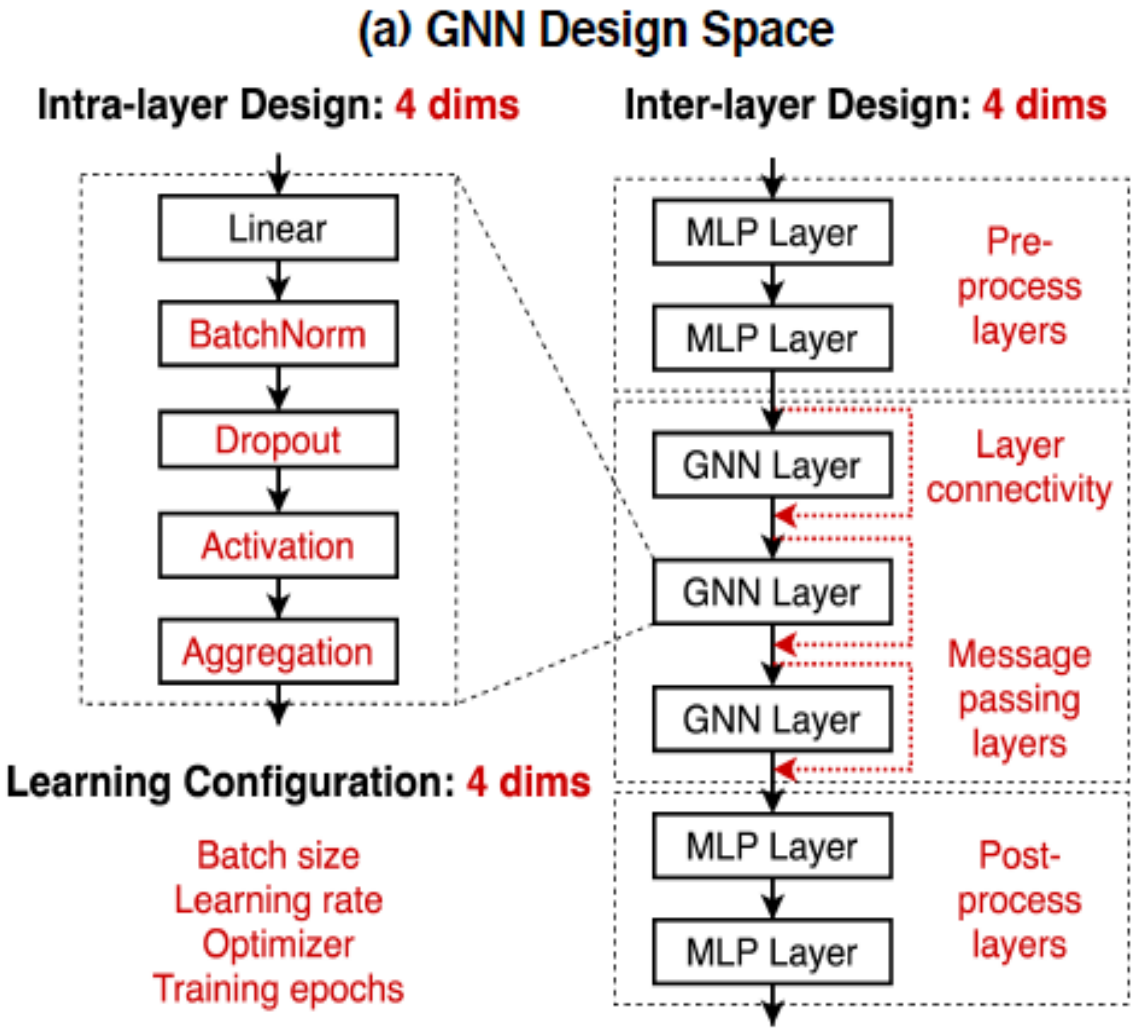
Same parameters are shared for all nodes, allowing it to be generalized to unseen nodes

Loss function

Closeness in original graph
Train on a supervised task, eg, classification using embeddings

Graph SAGE modifies this averaging function to any differentiable function that maps a set of vectors in N(v) to a single vector, thus generalizing neighborhood aggregation. Also, h(v) is concatenated to avoid over-smoothing.

Design Space



- Each Layer uses the Inter-Layer design
- MLPs in Pre and Post processing layer add *'depth'*
- *Skip* connections try to avoid over-smoothing

Update rule inside each layer

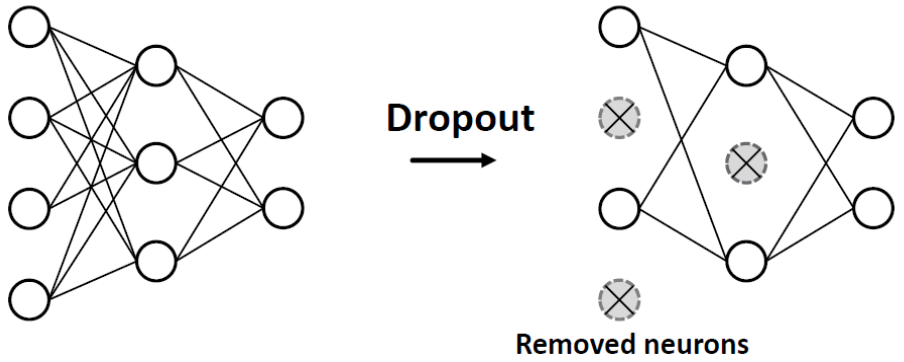
$$\mathbf{h}_v^{(k+1)} = \text{AGG}\left(\left\{\text{ACT}\left(\text{DROPOUT}\left(\text{BN}(\mathbf{W}^{(k)}\mathbf{h}_u^{(k)} + \mathbf{b}^{(k)})\right)\right), u \in \mathcal{N}(v)\right\}\right)$$

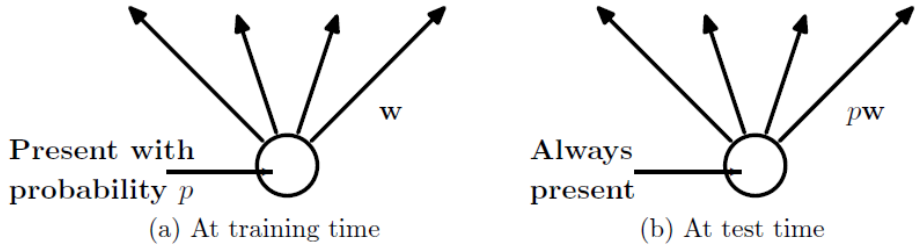
Batch Normalization	Dropout	Activation	Aggregation
True, False	False, 0.3, 0.6	RELU, PRELU, SWISH	MEAN, MAX, SUM

- Batch normalization improves stability (Reduces internal covariate shift)
- Drop-out is a way to regularize (BN=true may reduce its need)
- Activation functions
- SUM Aggregation is considered the best (most expressive) theoretically

Dropout

- **Goal**: Avoid over-fitting by model combination
- **Idea**: Randomly drop units along with connections during training
- This prevents units from co-adapting too much.
- During training, dropout samples from an exponential number of different “thinned” networks.
- At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single un-thinned network that has smaller weights
- Usual p used in the paper. P=0.8 for input layer and 0.5 for hidden layers
- **Idea**: For GNN it means we only aggregate messages from some of the neighbors





https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Other methods: L1/L2 regularization with penalty terms, early-stopping

Batch Normalization

- **Goal**: Stabilize neural networks training
- **Idea**: Given a batch of inputs (node embeddings)
  - Re-center the node embeddings into zero mean
  - Re-scale the variance into unit variance
- Input: x_i 's in the batch. Batch size m
- Trainable parameters: ϒ and β
- Output: y_i 's for the batch
- Parameters of the affine transformation needs to be learnt for each "neuron"/activation.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

https://arxiv.org/pdf/1502.03167v3.pdf

Increase learning rate, Remove dropout,
Reduce L2 weight regularization,
Accelerate learning rate decay,

Activation

$$\text{swish}(x) := x \times \text{sigmoid}(\beta x) = \frac{x}{1 + e^{-\beta x}}$$

A smoothing function which **nonlinearly interpolates between a linear and the ReLU function**.[2]

- For β=1, the function becomes equivalent to the *Sigmoid-weighted Linear Unit* (SiL) function
- β=0, the functions turns into the scaled linear function f(x)=x/2.
- With β→∞, the sigmoid component approaches a 0-1 function, so swish becomes like the ReLU function.

- **Rectified linear unit (ReLU)**

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

  - Most commonly used
- **Sigmoid**

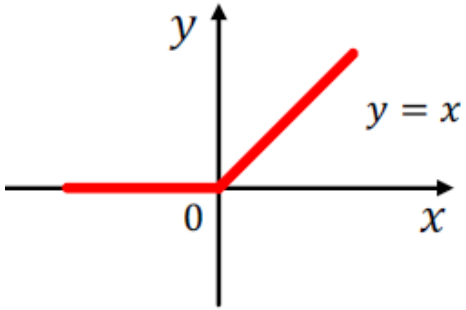
  $\sigma(\mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{x}_i}}$

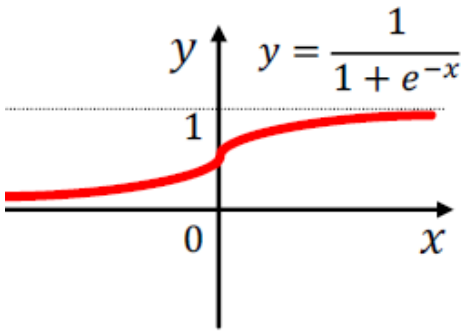
  - Used only when you want to restrict the range of your embeddings
- **Parametric ReLU**

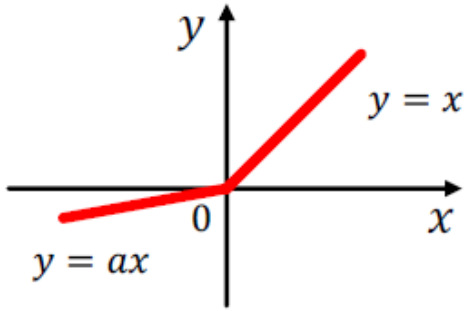
  $\text{PReLU}(\mathbf{x}_i) = \max(\mathbf{x}_i, 0) + a_i \min(\mathbf{x}_i, 0)$

  a_i is a trainable parameter

  - Empirically performs better than ReLU







- Each dimension of the embedding passes through a non-linear function
- Which activation should I use in my GNN?

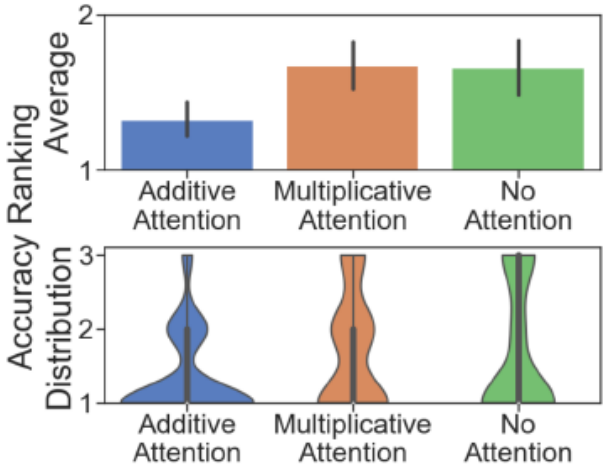
https://github.com/snap-stanford/GraphGym

Attention

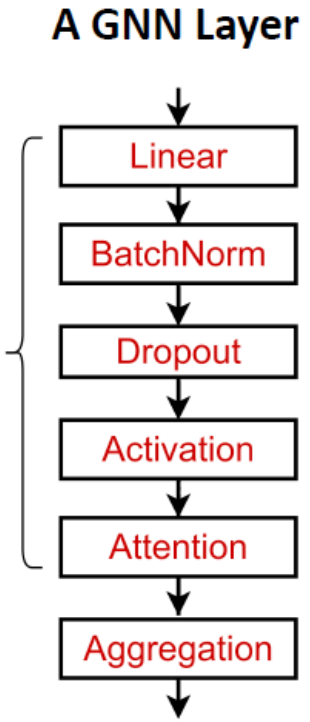
$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

Attention weights

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



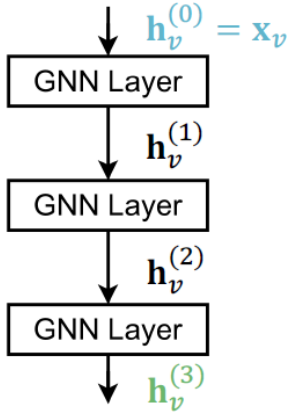
New design dimension

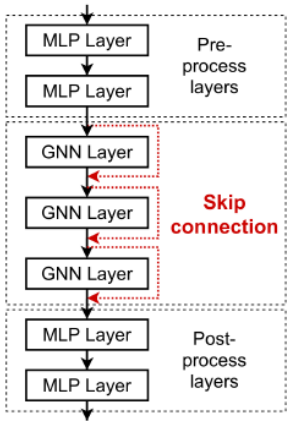


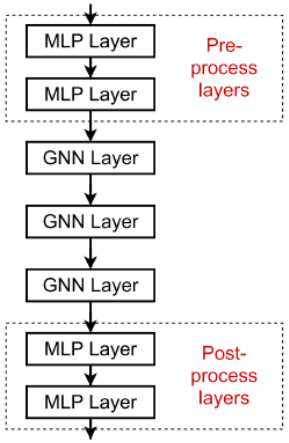
- Additive attention is always helpful!
- Do you have more ideas you want to incorporate intra layer?
  - Sampling (similar to drop-out)
- Designing novel GNN layers is an active research frontier!

Stacking Layers of a GNN

Layer connectivity	Pre-process layers	Message passing layers	Post-precess layers
STACK, SKIP-SUM, SKIP-CAT	1, 2, 3	2, 4, 6, 8	1, 2, 3



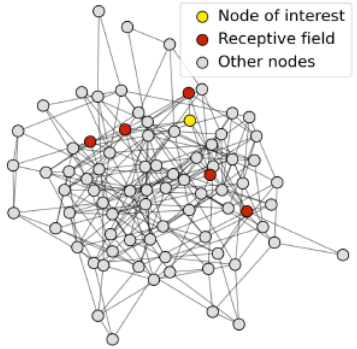




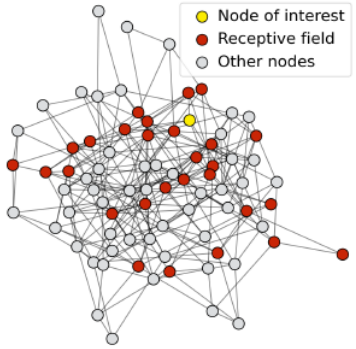
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

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

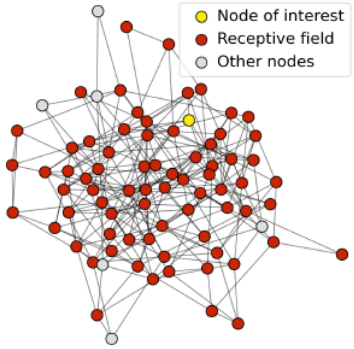
Receptive field for 1-layer GNN



Receptive field for 2-layer GNN



Receptive field for 3-layer GNN

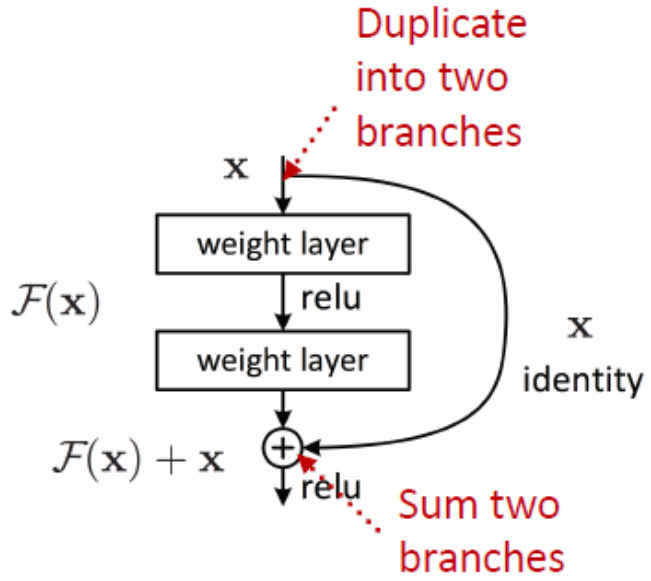


Many stack layers -> highly overlapping receptive fields -> similar node embeddings -> suffer from over-smoothing

SOLUTION: L<Diameter(G)

Skip Connections-I

- **Goal**: Solve under-performing deep networks
- **Idea**: Add skip connections with identity mapping
- Borrow ideas from Deep residual learning framework
- **Observation from over-smoothing:** Node embeddings in earlier GNN layers can sometimes better differentiate nodes
- **Solution:** We can increase the impact of earlier layers on the final node embeddings, **by adding shortcuts in GNN**



Idea of skip connections:

Before adding shortcuts:

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

After adding shortcuts:

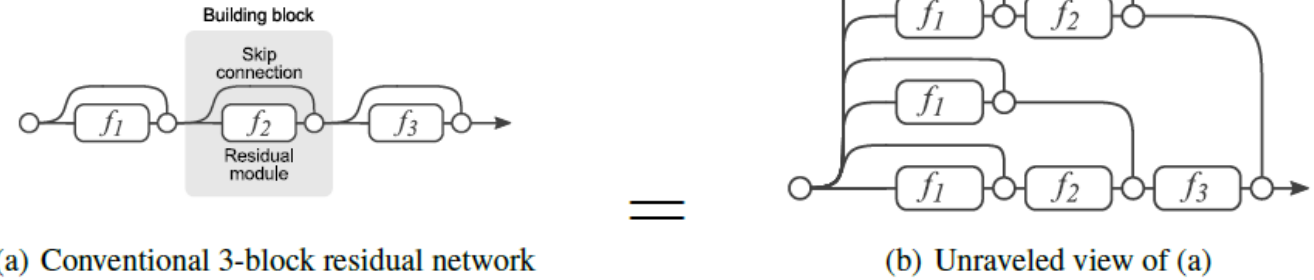
$$\boldsymbol{F}(\mathbf{x}) + \mathbf{x}$$

Resnet Paper: CVPR 2016

Not all systems are easy to optimize

Why do skip connections work?

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



(a) Conventional 3-block residual network

(b) Unraveled view of (a)

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

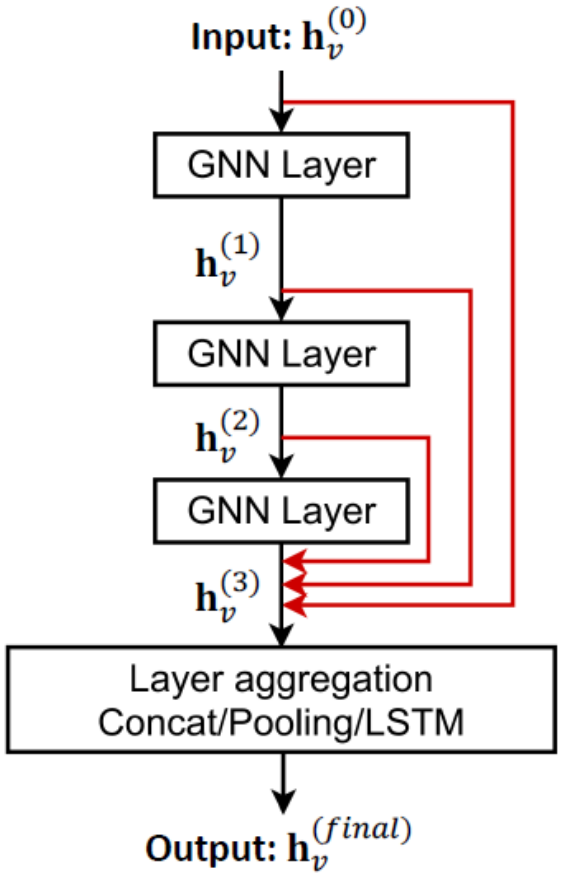
Constructions and Options

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

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



Directly skip to the last layer

- The final layer directly **aggregates from the all the node embeddings** in the previous layers

Xu et al. Representation learning on graphs with jumping knowledge networks, ICML 2018

Training Configurations

Batch size	Learning rate	Optimizer	Training epochs
16, 32, 64	0.1, 0.01, 0.001	SGD, ADAM	100, 200, 400

- Optimizer:
  - SGD: First order method. Early steps of SGD converge more quickly than GD toward the solution, then erratic (need early stopping or averaging)

    https://arxiv.org/abs/1712.06559
  - ADAM: General idea is to use previous gradients (memory) in choosing the next gradient and step size. In particular, exponential moving average is used.
  - https://ruder.io/optimizing-gradient-descent/index.html#otherrecentoptimizers
  - https://johnchenresearch.github.io/demon/

  Active area of theoretical and empirical research

Task Space

Small world: Short average path length, high average clustering coefficient
Scale Free: Degree distribution follows Power law, new nodes keep attaching to nodes with high degrees

- Define 32 diverse illustrative tasks:
  - 12 synthetic* node classification tasks
  - 8 synthetic* graph classification tasks
  - 6 real world node classification: Amazon computers/photo, Citeseer, Coauthor CS/Phy, Cora
  - 6 graph classification tasks: 5 Biology related, IMDB
- The proposed task similarity metric consists of two components:
- Selection of anchor models
  - Sample D random GNN designs from the design space and apply to fixed set of tasks
  - D designs are ranked and evenly sliced into M groups and model with median performance selected
- Measuring task similarity
  - Rank the performance of all M anchor models for each of the tasks
  - Can be generalized to classification, regression and non-predictive tasks as well.
  - Use Kendall rank correlation to measure similarity
  - Computation cost for T tasks = M*T. M=12 is sufficient.

Average Clustering Coefficient C [0.3; 0.6] and Average Path Length L [1.8; 3.0]
*node features: (1) constant scalar, (2) one-hot vectors, (3) node clustering coefficients and (4) node PageRank score
*node-level labels including node clustering coefficient and node PageRank score, and
*graph-level labels, such as average path length