

# Semi-supervised Methods for Graph Representation

[Modeling Data With Networks + Network Embedding:  
Problems, Methodologies and Frontiers](#)

Ivan Brugere (University of Illinois at Chicago)  
Peng Cui (Tsinghua University)  
Bryan Perozzi (Google)  
Wenwu Zhu (Tsinghua University)  
Tanya Berger-Wolf (University of Illinois at Chicago)  
Jian Pei (Simon Fraser University)

Bryan Perozzi  
[bperozzi@acm.org](mailto:bperozzi@acm.org)

# Why Supervision?

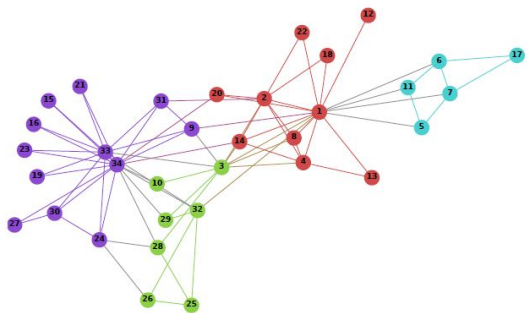
Supervision allows us to tailor the network representation to the task we actually care about!

E.g: Are you going to use the representations for labeling?

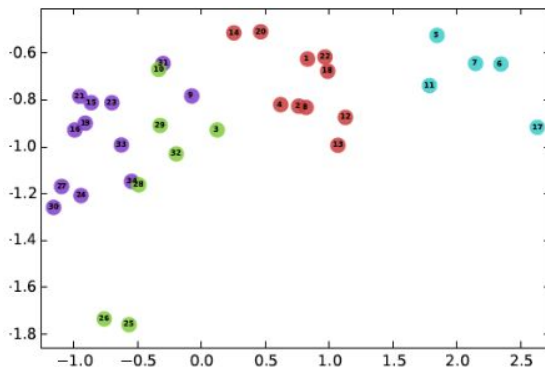
=> Include training labels

# Power of Supervision

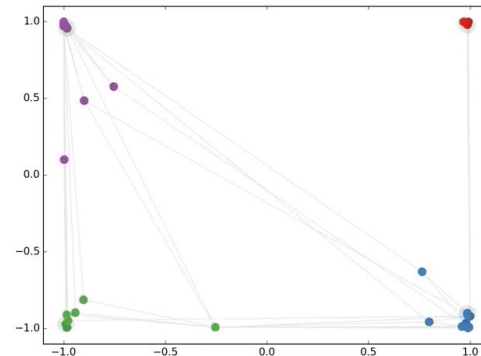
Graph: Zachary's Karate Club  
Labels: Community Ids



**Graph Layout**

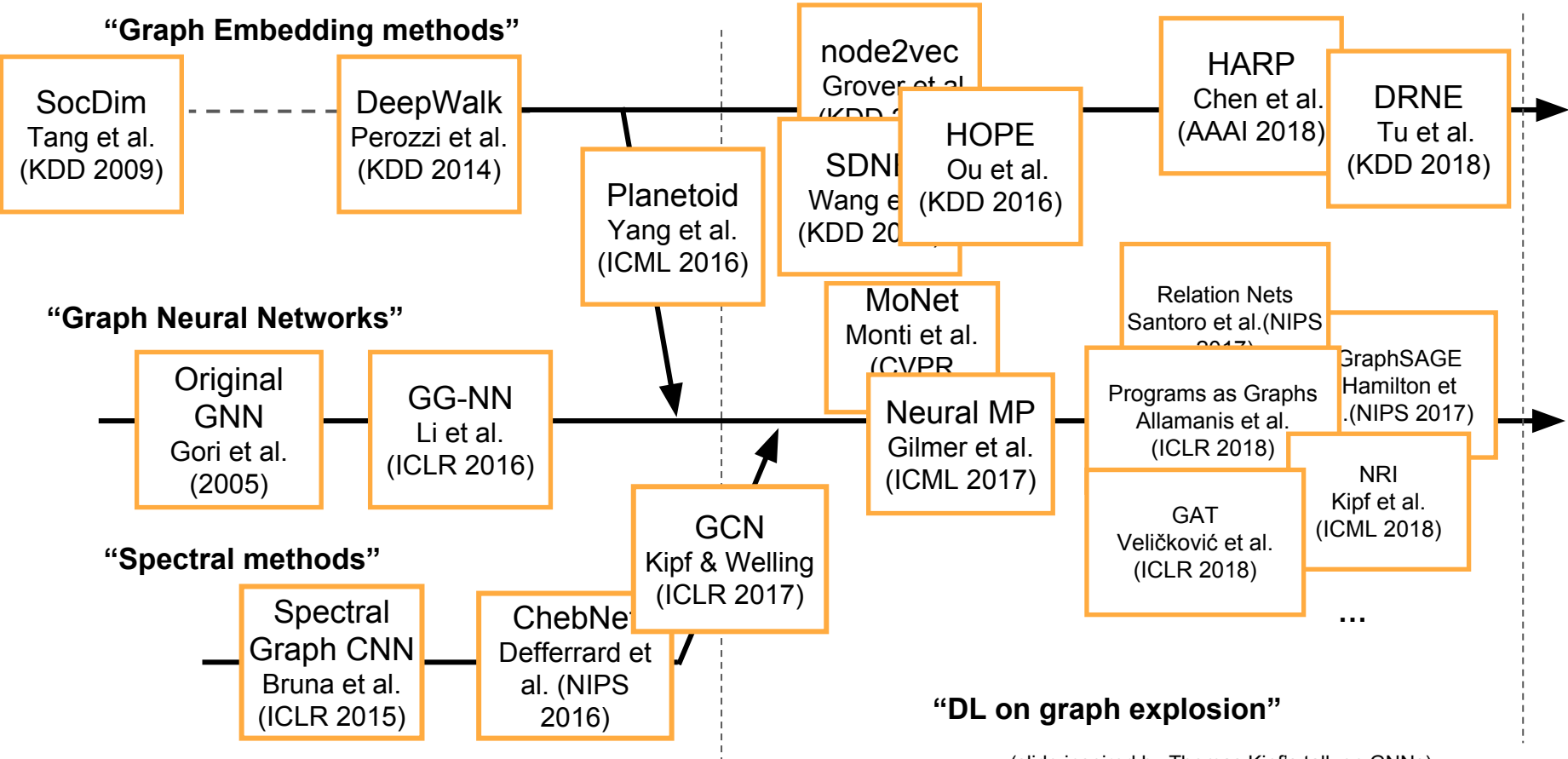


**Unsupervised  
Representation  
(DeepWalk)**



**Supervised  
Representation  
(GCN)**

# A brief history of Graphs and Neural Networks



# In this Section

## 1. **Semi-Supervised Learning w/ Graph Embeddings**

- a. Unsupervised Embeddings + Training Data
- b. Graphs as Regularizers
- c. Graph Convolutional Approaches
  - i. Overview
  - ii. GCNs
  - iii. Extensions

## 2. Supervision as Inspiration

- a. The Graph as Supervision
- b. Watch Your Step

# Semi-supervised Learning on Graphs

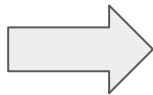
Problem Definition:

- INPUT:
  - Adjacency Matrix,  $A$
  - Features,  $X$
  - Partially Labeled Nodes
- OUTPUT:
  - Labels for all Nodes,  $Y$

# Semi-supervised Graph Representation Learning

## Standard SSL Definition:

- INPUT:
  - Adjacency Matrix,  $A$
  - Features,  $X$
  - Partially Labeled Nodes
- OUTPUT:
  - Labels for all Nodes,  $Y$



## Methods We'll Focus On:

- INPUT:
  - Adjacency Matrix,  $A$
  - Features,  $X$
  - Partially Labeled Nodes
- OUTPUT:
  - Representations  $\Phi$  for each Node
    - Correlated with the labels
  - Labels for all Nodes,  $Y$

# Adding Supervision to Unsupervised Embeddings



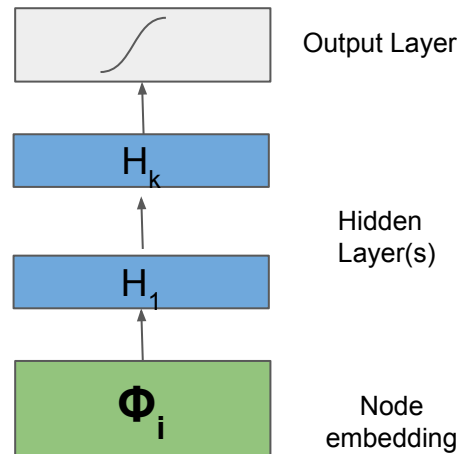
# The Straightforward Approach

**Given:** We already know how to create good embeddings for a graph.

**Idea:** Why not “put a DNN on it”?

1. Embed  $A \rightarrow \Phi$
2. Pass through 0 or more hidden layers
3. Add Output Layer

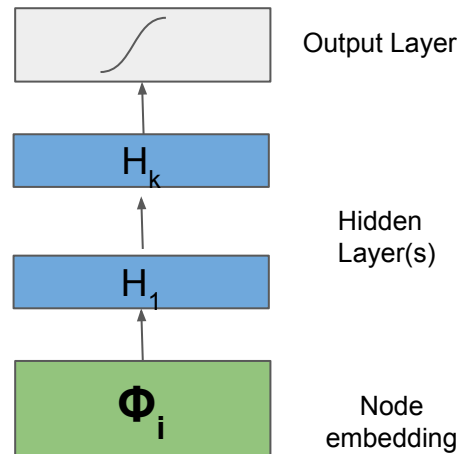
The internal layers of this DNN *are* semi-supervised graph representations.



# Challenges with “Put a DNN on it”

**Problem 1:** Maybe something went wrong with the embedding process (bad initialization, hyper-parameters, etc).

We’ve already thrown out the graph!

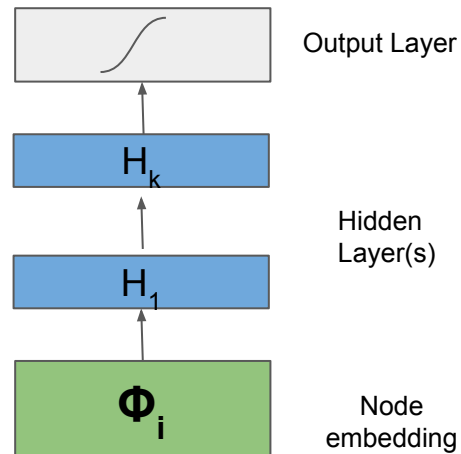


# Potential Problem: Generalization

**Problem 2:** How can we loosen the assumption on  $\Phi$ 's quality.

One way to do this is might be to fine-tune each node's  $\Phi_i$  as we train.

(Doesn't generalize well, and can easily overfit - testing data never receive  $\Phi_i$  updates. )

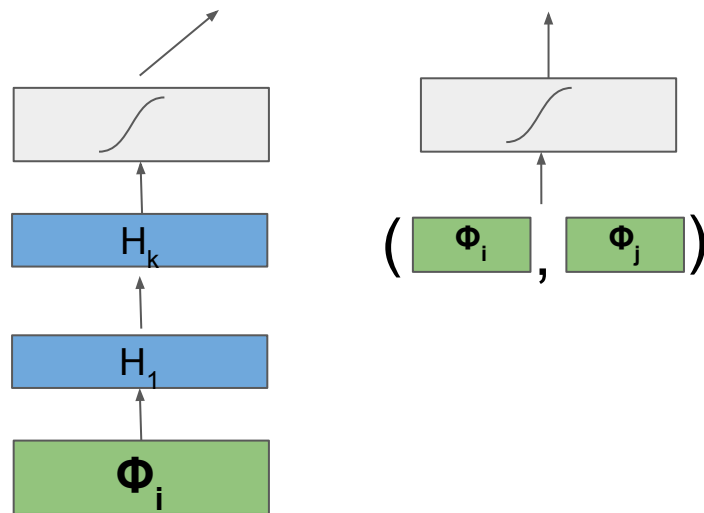


# Joint Loss Models

**Idea:** NN are flexible, can we combine?

1. Require the embeddings stay good
  - a. (Similar to a reconstruction loss)
2. Share representations between loss terms

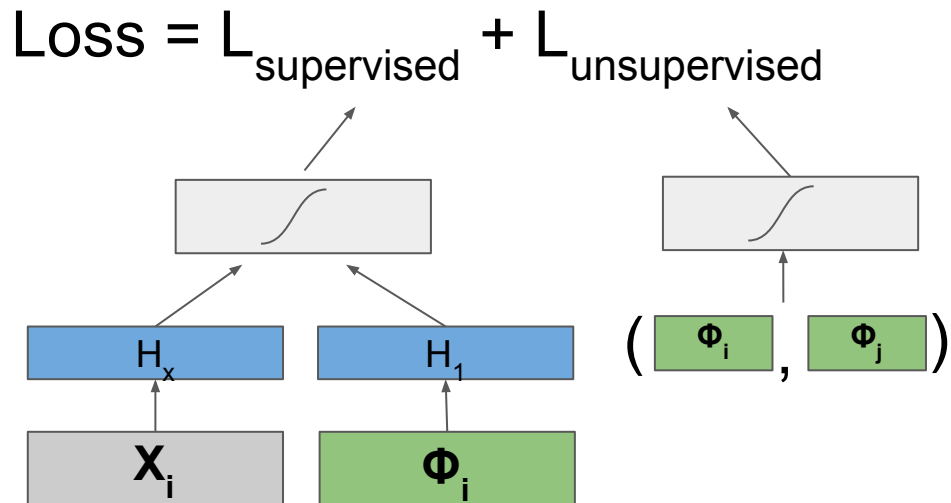
$$\text{Loss} = L_{\text{supervised}} + L_{\text{unsupervised}}$$



# Planetoid-T Model

Application of what we've discussed so far.

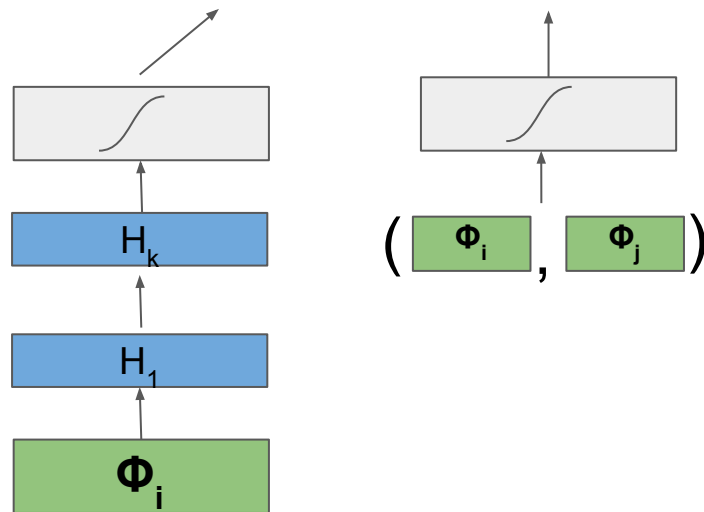
1. Features and Node Embeddings combined for a supervised loss.
2. Unsupervised loss keeps the embeddings good



# Challenges with Joint Loss Models

**Problem 1:** How to balance the two terms of the loss?

$$\text{Loss} = L_{\text{supervised}} + L_{\text{unsupervised}}$$

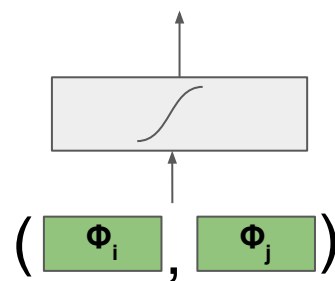
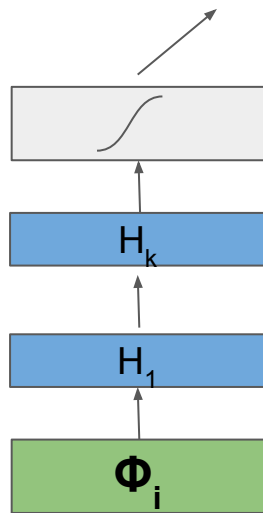


# Challenges with Joint Loss Models

**Problem 1:** How to balance the two terms of the loss?

**Problem 2:** Model only has single global hyper-parameter to control combination for all nodes.

$$\text{Loss} = L_{\text{supervised}} + L_{\text{unsupervised}}$$



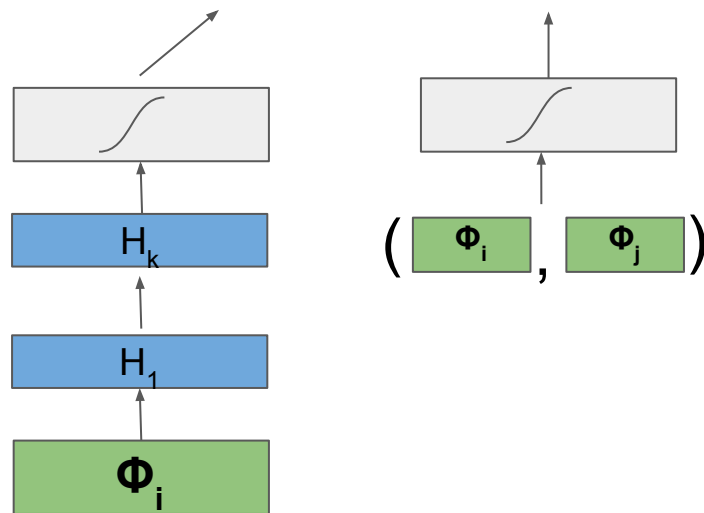
# Challenges with Joint Loss Models

**Problem 1:** How to balance the two terms of the loss?

**Problem 2:** Model only has single global hyper-parameter to control combination for all nodes.

**Problem 3:** We're still throwing away the graph...

$$\text{Loss} = L_{\text{supervised}} + L_{\text{unsupervised}}$$





# Graphs as a Regularizer (aside)

# Graph Regularization for Label Assignment

Significant body of work on adding a graph regularization to an existing model.

E.g. [Zhu, et al, 2003] combine a classifier loss  $L_0$  with a graph-regularizer loss  $L_{\text{reg}}$  that encourages similar output for connected nodes

$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{\text{reg}}, \quad \text{with} \quad \mathcal{L}_{\text{reg}} = \sum_{i,j} A_{ij} \|f(X_i) - f(X_j)\|^2$$

Another example is Neural Graph Machines [Bui et al, WSDM'18], which uses graph similarity as a regularization in a hidden (latent) layer

This is outside the scope of today's tutorial, but it's good to be aware of.

# Outline

1. Semi-Supervised Learning w/ Graph Embeddings
  - a. Unsupervised Embeddings + Training Data
  - b. Graphs as Regularizers
  - c. **Graph Convolutional Approaches**
    - i. Overview
    - ii. GCNs
    - iii. Extensions
2. Supervision as Inspiration
  - a. The Graph as Supervision
  - b. Watch Your Step

# Graph Convolutional Methods

# The Big Idea

1. Formulate a joint objective over the graph and the labels
2. Jointly learn representation and label predictions.

Methods here explicitly spread information over the graph while learning their representations.

# The success story of deep learning

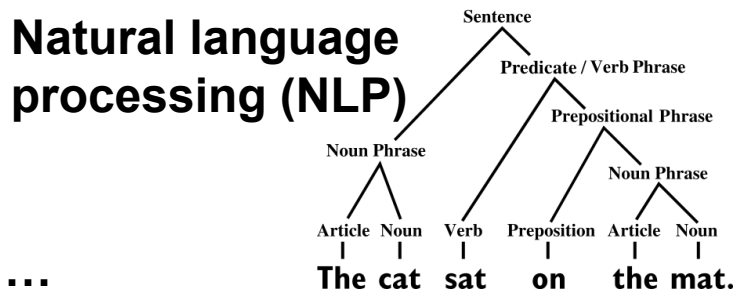
IMAGENET



Speech data

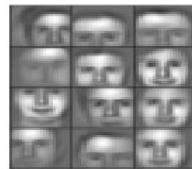
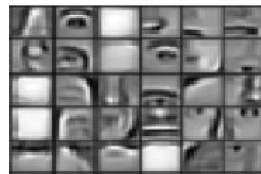
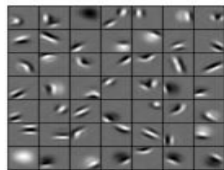


Natural language processing (NLP)

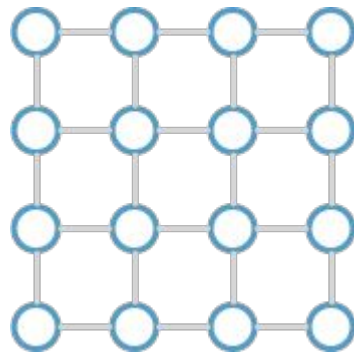
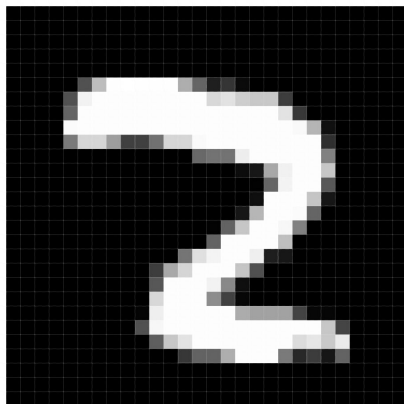


Deep neural nets that exploit:

- translation invariance (weight sharing)
- hierarchical compositionality



# Recap: Deep learning on Euclidean data



2D grid

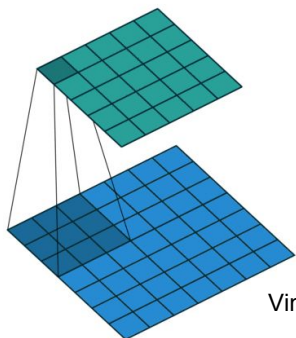


1D grid

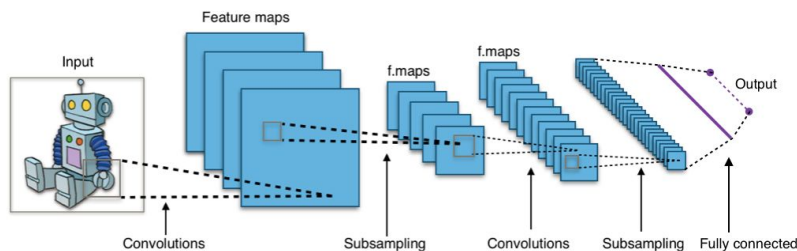
[original slide: Thomas Kipf, used w/ permission]

# Recap: Deep learning on Euclidean data

## Convolutional neural networks (CNNs)

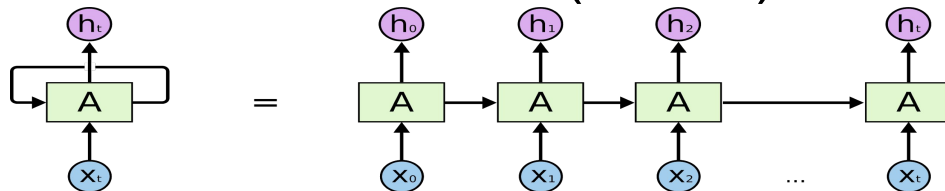


(Animation by Vincent Dumoulin)



(Source: Wikipedia)

## or recurrent neural networks (RNNs)

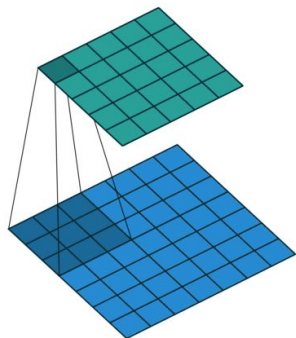


(Source: Christopher Olah's blog)

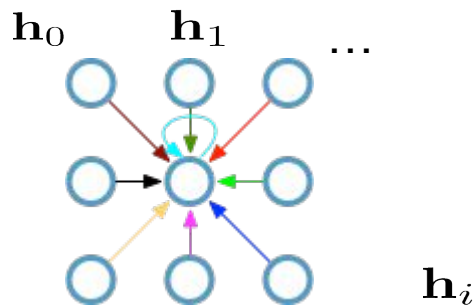


# Convolutional neural networks (on grids)

Single CNN layer with 3x3 filter:



(Animation by  
Vincent Dumoulin)



**Update for a single pixel:**

- Transform neighbors individually  $\mathbf{W}_i \mathbf{h}_i$
- Add everything up  $\sum_i \mathbf{W}_i \mathbf{h}_i$

Full update: 
$$\mathbf{h}_4^{(l+1)} = \sigma \left( \mathbf{W}_0^{(l)} \mathbf{h}_0^{(l)} + \mathbf{W}_1^{(l)} \mathbf{h}_1^{(l)} + \dots + \mathbf{W}_8^{(l)} \mathbf{h}_8^{(l)} \right)$$

[original slide: Thomas Kipf, used w/ permission]

# Graph Convolutional Networks

# Spectral graph convolutions

## Main idea:

Use **convolution theorem** to generalize convolution to graphs.

Loosely speaking:

*A convolution corresponds to a multiplication in the Fourier domain.*

**Graph Fourier transform:** [Hammond, Vandergheynst, Gribonval, 2009]

$$\mathcal{F}[\mathbf{x}] = \Phi^T \mathbf{x} \quad \Phi: \text{eigenvectors of graph Laplacian } \mathbf{L}$$

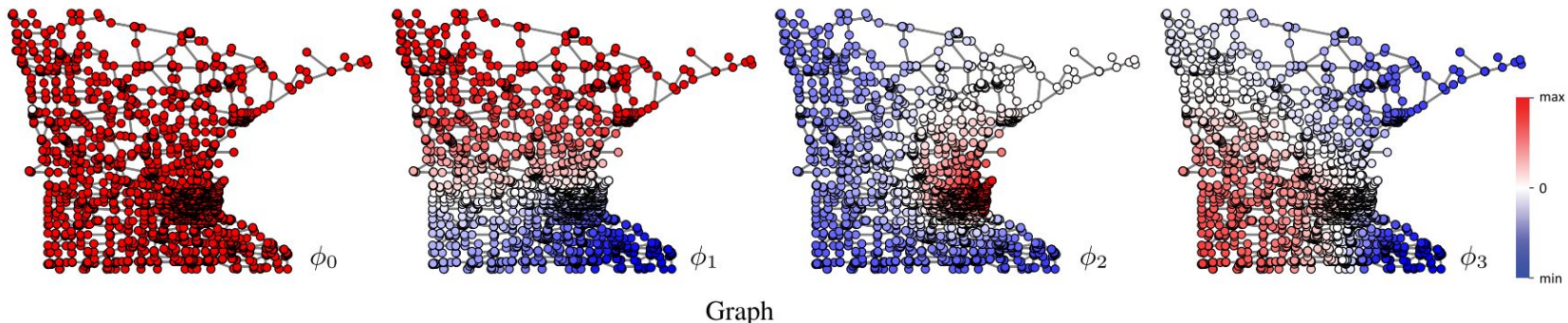
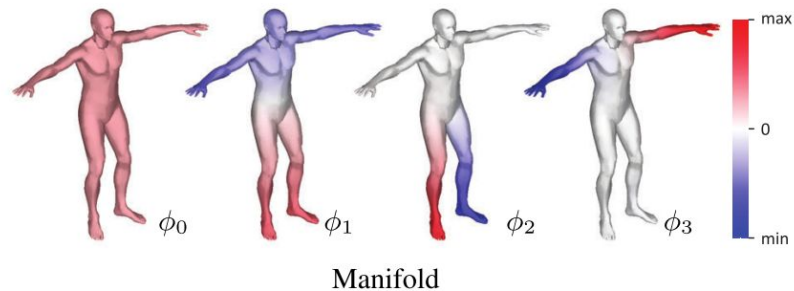
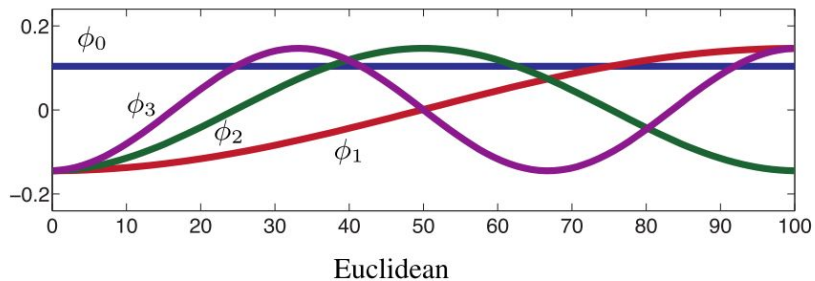
with  $\mathbf{L} = \mathbf{I}_N - \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}}$  (normalized graph Laplacian)

and  $\mathbf{L} = \Phi \Lambda \Phi^T$  (its eigen-decomposition)

$\mathbf{D}$ : degree matrix

$$D_{ii} = \sum_j A_{ij}$$

# Eigenvectors of the graph Laplacian



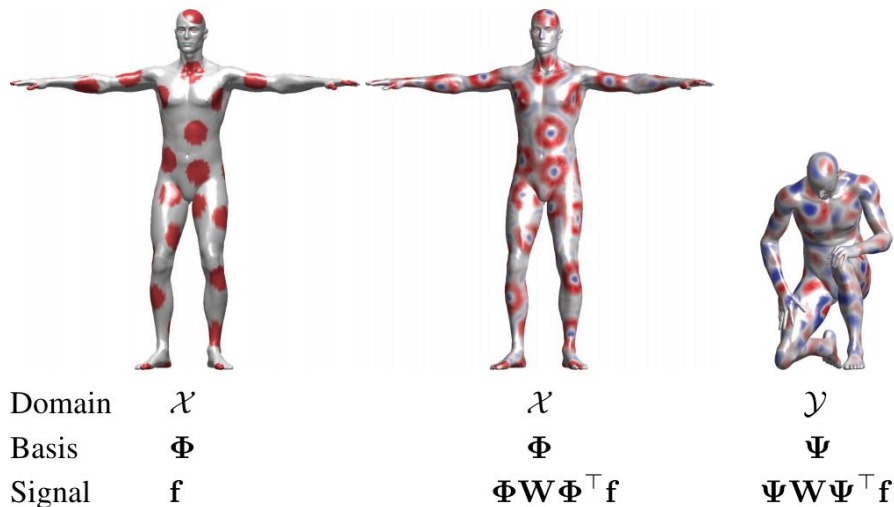
[FIGS3] Example of the first four Laplacian eigenfunctions  $\phi_0, \dots, \phi_3$  on a Euclidean domain (1D line, top left) and non-Euclidean domains

$\phi_i$ : eigenvectors of *graph Laplacian*  $\mathbf{L}$

[Figure: Bronstein et al., 2016]

[original slide: Thomas Kipf, used w/ permission]

# CNNs with spectral graph convolutions



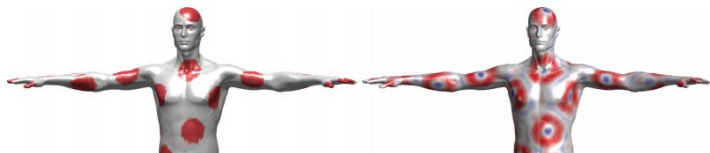
[Figure: Bronstein et al., 2016]

**Recipe for CNN on graphs** [Bruna et al., 2014]:

Stack multiple layers of spectral graph convolutions + non-linearities

[original slide: Thomas Kipf, used w/ permission]

# CNNs with spectral graph convolutions



## Limitations:

- Calculating  $\Phi$  is expensive  $\mathcal{O}(N^3)$
- Evaluating  $\Phi^T \mathbf{x}$  is  $\mathcal{O}(N^2)$
- Spectral filters are graph-specific => difficult to transfer

[Figure: Bronstein et al., 2016]

**Recipe for CNN on graphs** [Bruna et al., 2014]:

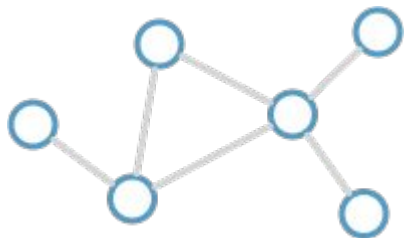
Stack multiple layers of spectral graph convolutions + non-linearities

[original slide: Thomas Kipf, used w/ permission]

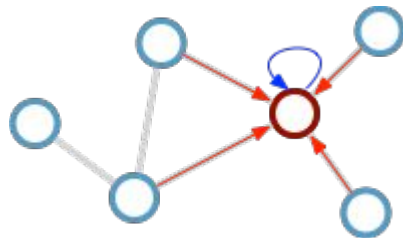
(first proposed in [Scarselli et al. 2009])

# CNNs on graphs with spatial filters

Consider this undirected graph:



Calculate update for node in red:



**Update rule:** 
$$\mathbf{h}_i^{(l+1)} = \sigma \left( \mathbf{h}_i^{(l)} \mathbf{W}_0^{(l)} + \sum_{j \in \mathcal{N}_i} \frac{1}{c_{ij}} \mathbf{h}_j^{(l)} \mathbf{W}_1^{(l)} \right)$$

$\mathcal{N}_i$  : neighbor indices  
 $c_{ij}$  : norm. constant (per edge)

How is this related to spectral CNNs on graphs?

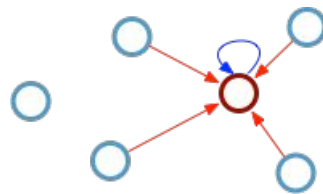
➔ Localized 1st-order approximation of spectral filters [Kipf & Welling, ICLR 2017]

[original slide: Thomas Kipf, used w/ permission]

# Vectorized form

$$\mathbf{H}^{(l+1)} = \sigma \left( \mathbf{H}^{(l)} \mathbf{W}_0^{(l)} + \tilde{\mathbf{A}} \mathbf{H}^{(l)} \mathbf{W}_1^{(l)} \right)$$

$$\text{with } \tilde{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}}$$

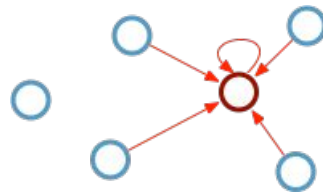


$$\mathbf{H}^{(l)} = [\mathbf{h}_1^{(l)T}, \dots, \mathbf{h}_N^{(l)T}]^T$$

Or treat self-connection in the same way:

$$\mathbf{H}^{(l+1)} = \sigma \left( \hat{\mathbf{A}} \mathbf{H}^{(l)} \mathbf{W}_1^{(l)} \right)$$

$$\text{with } \hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} (\mathbf{A} + \mathbf{I}_N) \tilde{\mathbf{D}}^{-\frac{1}{2}}$$



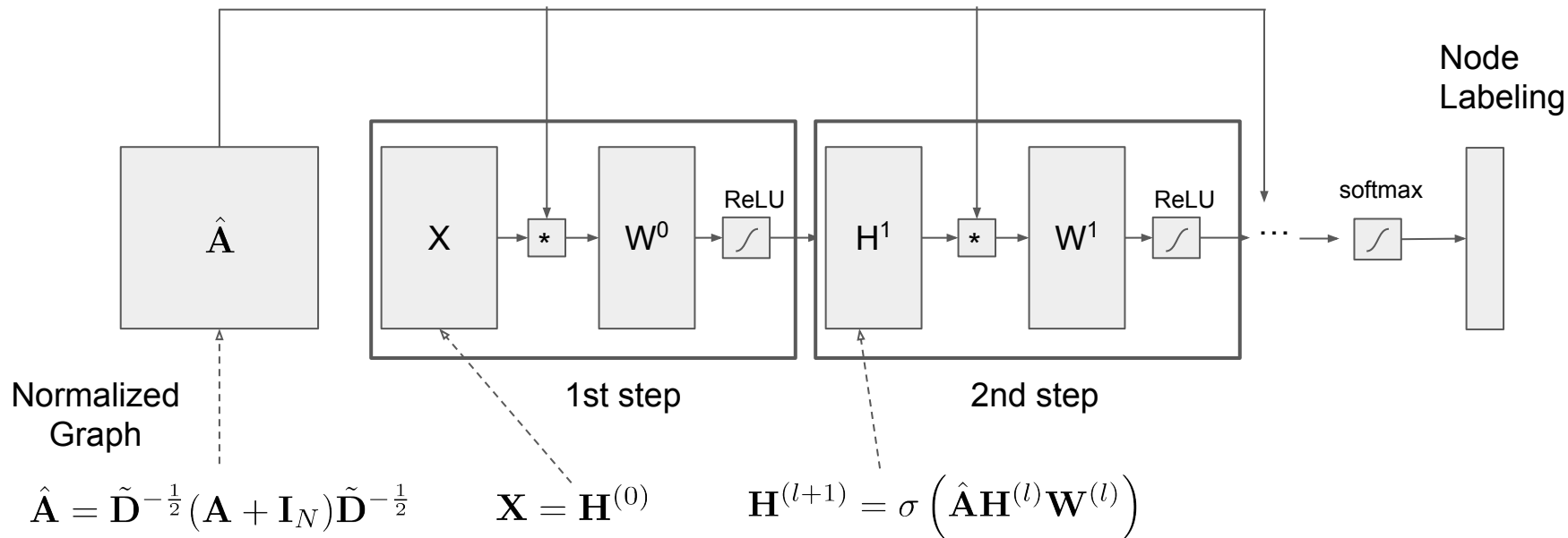
$$\tilde{D}_{ii} = \sum_j (A_{ij} + \delta_{ij})$$

$\mathbf{A}$  is typically **sparse**

- ➔ We can use sparse matrix multiplications!
- ➔ Efficient  $\mathcal{O}(|\mathcal{E}|)$  implementation in Theano or TensorFlow

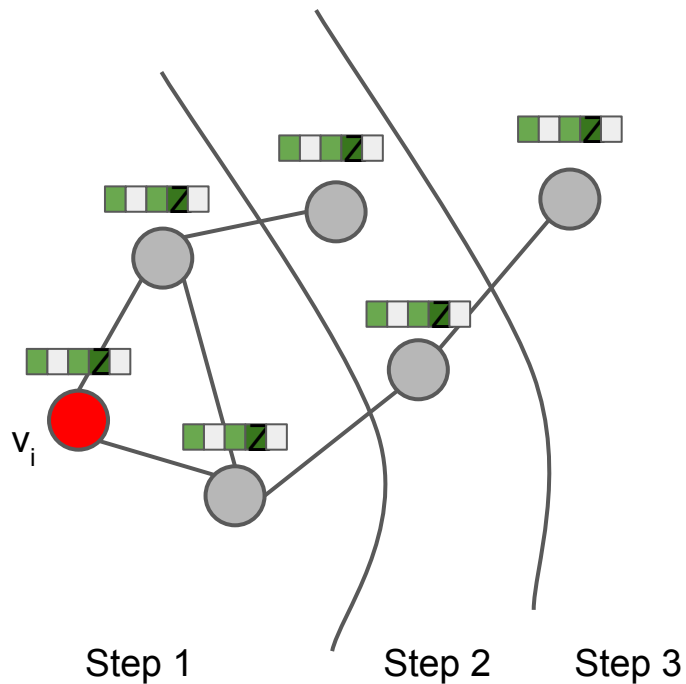


# Block View of the GCN Model



# Graph View of GCN Model

At every iteration, the model aggregates information from one hop deeper.



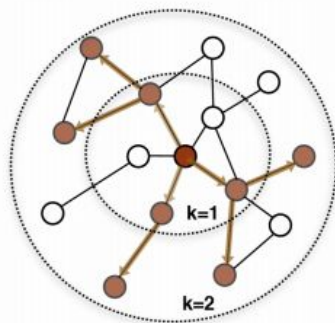
# Potential Limitations of the GCN Model

1. Fixed Aggregation Function
2. Scalability
3. Treats all Graph Edges equally
4. Effective Depth

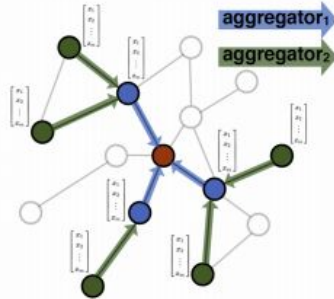
# GCN Extensions

# GraphSAGE

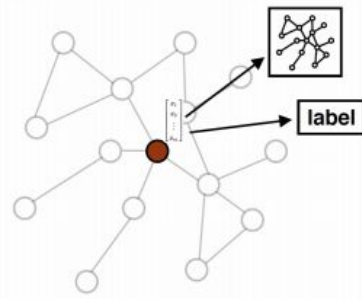
SAGE = **S**Ample Aggre**GatE**.



1. Sample neighborhood



2. Aggregate feature information from neighbors



3. Predict graph context and label using aggregated information

Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

Versus GCN:

Stochastic sampling

Flexible aggregation  
(commonly mean still)

Skip Layers

L2 norm layer

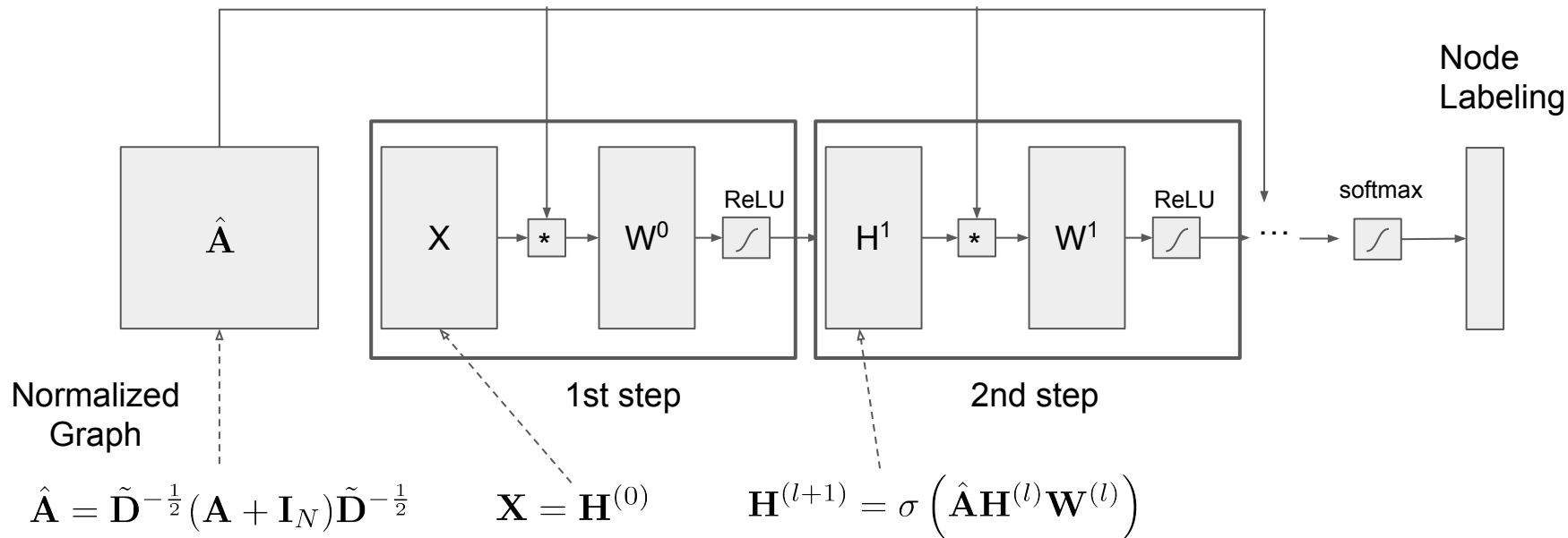
**Algorithm 1:** GraphSAGE embedding generation (i.e., forward propagation) algorithm

**Input** : Graph  $\mathcal{G}(\mathcal{V}, \mathcal{E})$ ; input features  $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$ ; depth  $K$ ; weight matrices  $\mathbf{W}^k, \forall k \in \{1, \dots, K\}$ ; non-linearity  $\sigma$ ; differentiable aggregator functions  $\text{AGGREGATE}_k, \forall k \in \{1, \dots, K\}$ ; neighborhood function  $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

**Output** : Vector representations  $\mathbf{z}_v$  for all  $v \in \mathcal{V}$

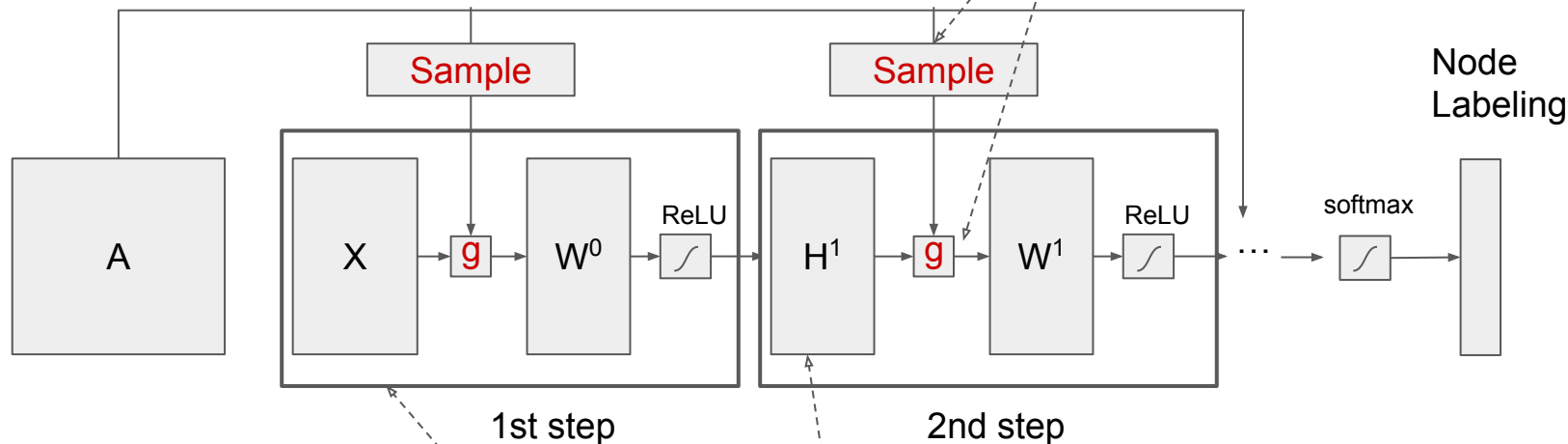
```
1  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;  
2 for  $k = 1 \dots K$  do  
3   for  $v \in \mathcal{V}$  do  
4      $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ ;  
5      $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))$   
6   end  
7  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$   
8 end  
9  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$ 
```

# Block View of the GCN Model



# Block View of the **GraphSAGE** Model

Introduces **Sample** and **Aggregate**  
 $g()$  can be Average or LSTM



$$\mathbf{X} = \mathbf{H}^{(0)}$$

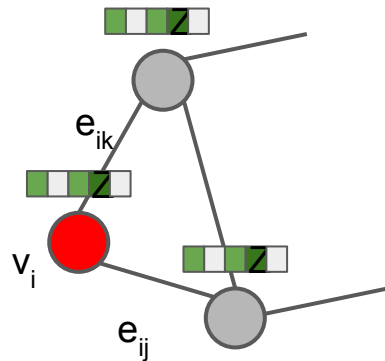
$$\mathbf{H}^{(l+1)} = \sigma \left( \hat{\mathbf{A}} \mathbf{H}^{(l)} \mathbf{W}^{(l)} \right)$$

# Graph Attention Networks (GAT)

**Problem:** GCNs assume the importance of the edges in the graph are equal.

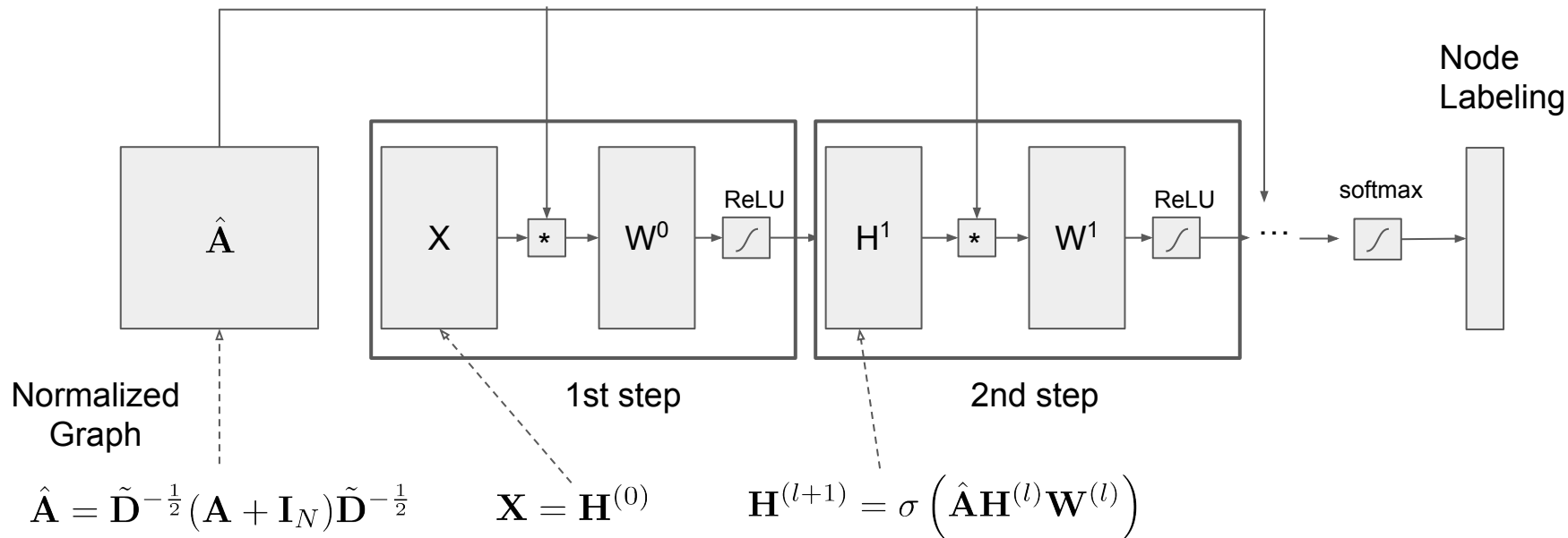
**Idea:** What if we add a learnable attention weight for each edge  $e_{ij}$ ?

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$
$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T [\mathbf{W}\vec{h}_i \parallel \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T [\mathbf{W}\vec{h}_i \parallel \mathbf{W}\vec{h}_k]\right)\right)}$$

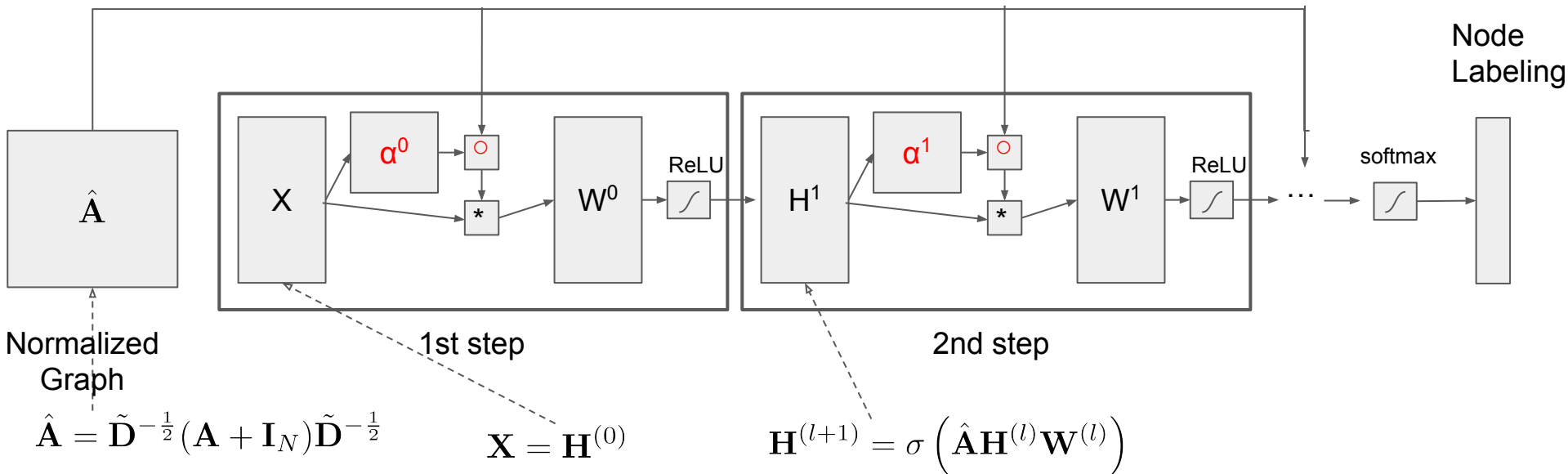




# Block View of the GCN Model



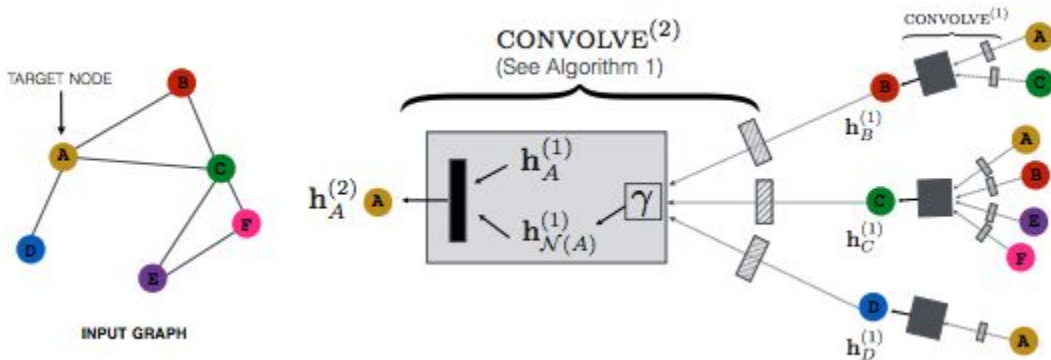
# Block View of the GAT Model



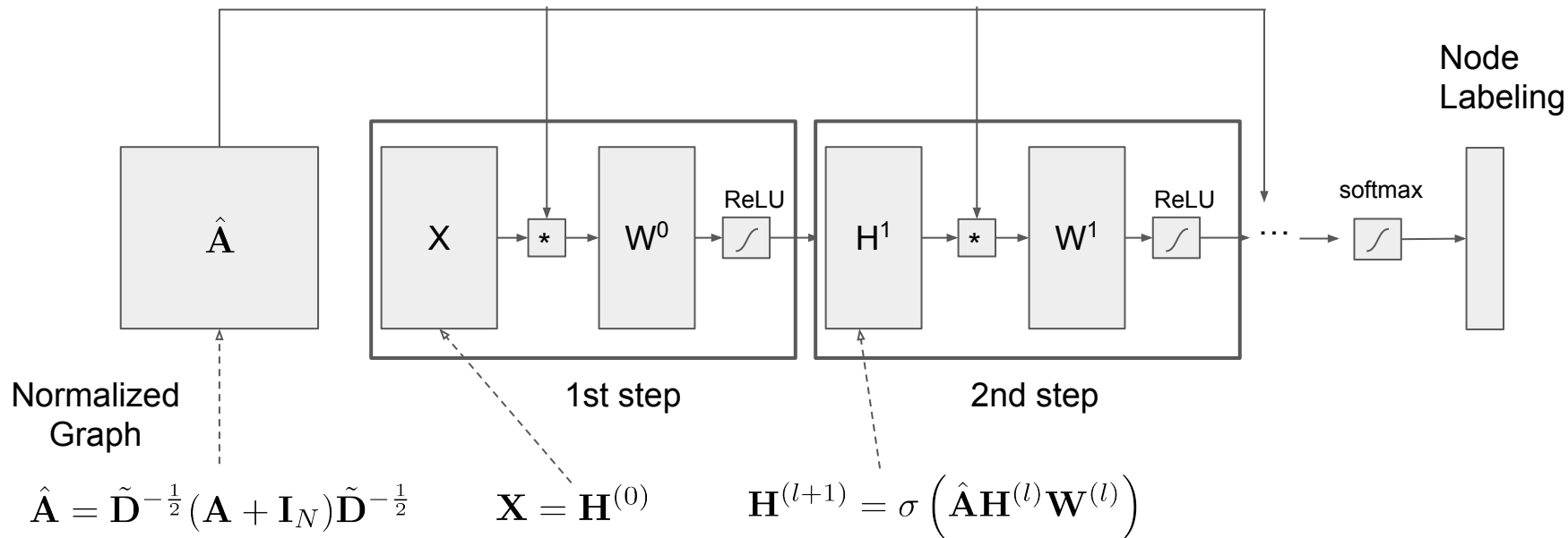
# Scalable GCNs

**Problem:** How to distribute a GCN over billion node graphs?

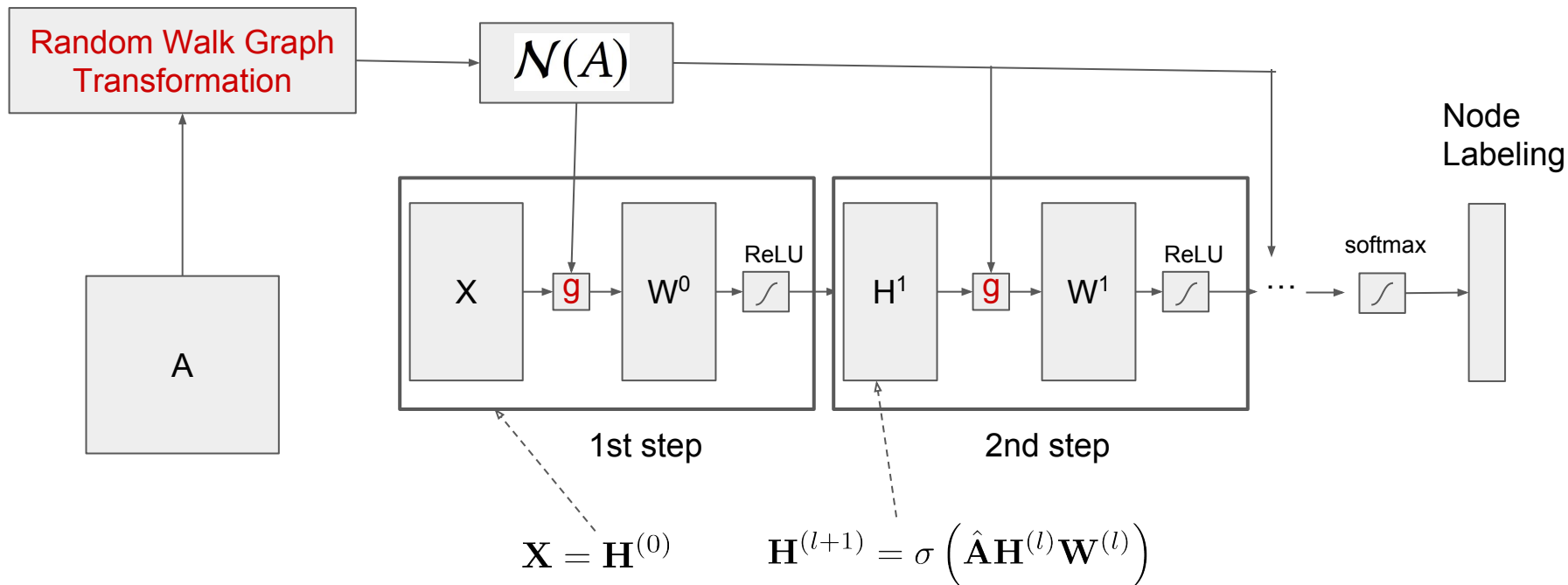
**Idea:** Sampling neighbors, local convolution, and M/R node embeddings.



# Block View of the GCN Model

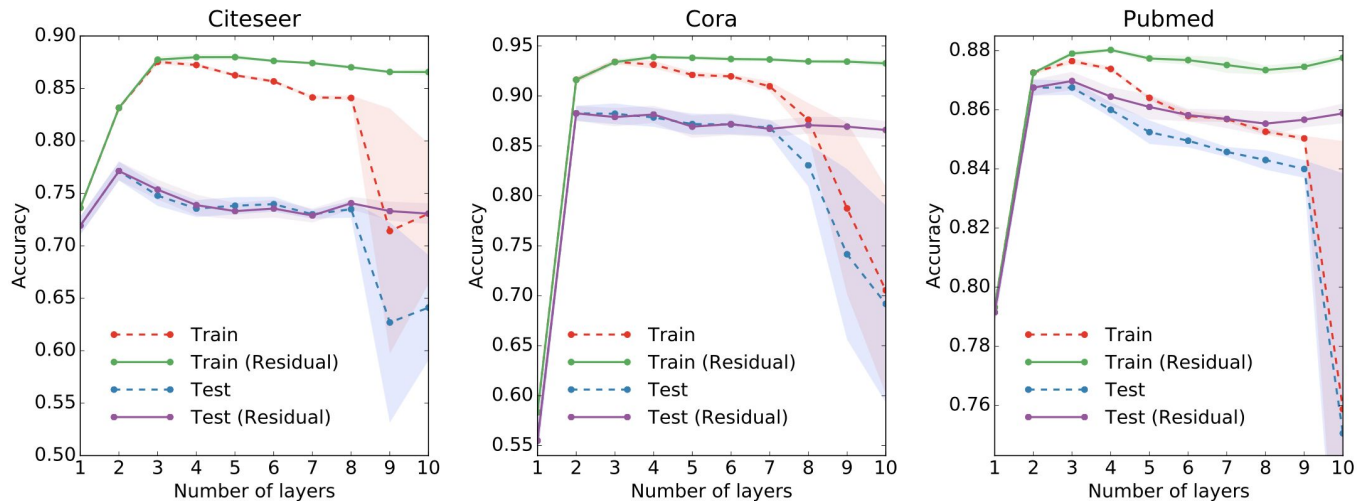


# Block View of the Scalable GCN Model



After training, inference via MapReduce pipeline.

# GCN: How deep is deep enough?



Residual connection

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}\right) + \boxed{H^{(l)}}$$

# Graph Convolution VS Random Walks

$$\text{GCN}_{2\text{-layer}}(\hat{A}, X; \theta) = \text{softmax} \left( \hat{A} \sigma(\hat{A} X W^{(0)}) W^{(1)} \right)$$

Consider special case if  $\sigma$  is “identity” and  $W^{(0)}$  is identity matrix:

$$\text{GCN}_{2\text{-layer-special}}(\hat{A}, X) = \text{softmax} \left( \hat{A} \hat{A} X W^{(1)} \right)$$

Becomes equivalent to 1-step random walk. In general:

$$\hat{A}^k = D^{-\frac{1}{2}} A T^{k-1} D^{-\frac{1}{2}}$$

What if we explicitly feed-in Random Walk statistics into GCNs?

# N-GCN: Multi-scale Graph Convolution for Semi-supervised Node Classification

- Make many instantiations of GCN modules.
- Feed each some power of Adjacency matrix.
- Concatenate output of all GCN instantiations, feed into fully-connected layers, producing node labels.

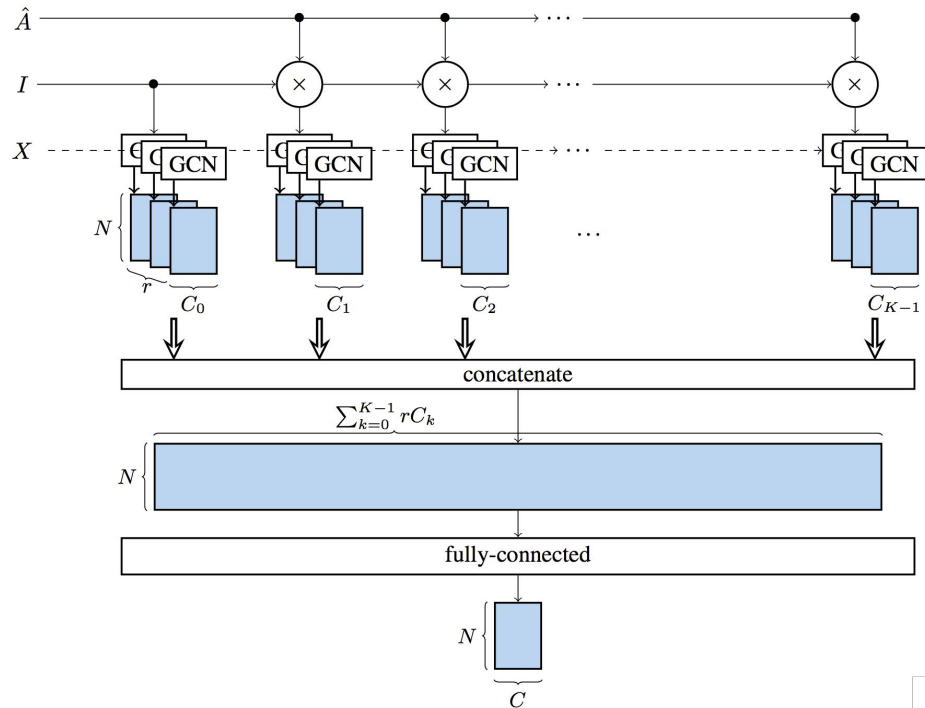


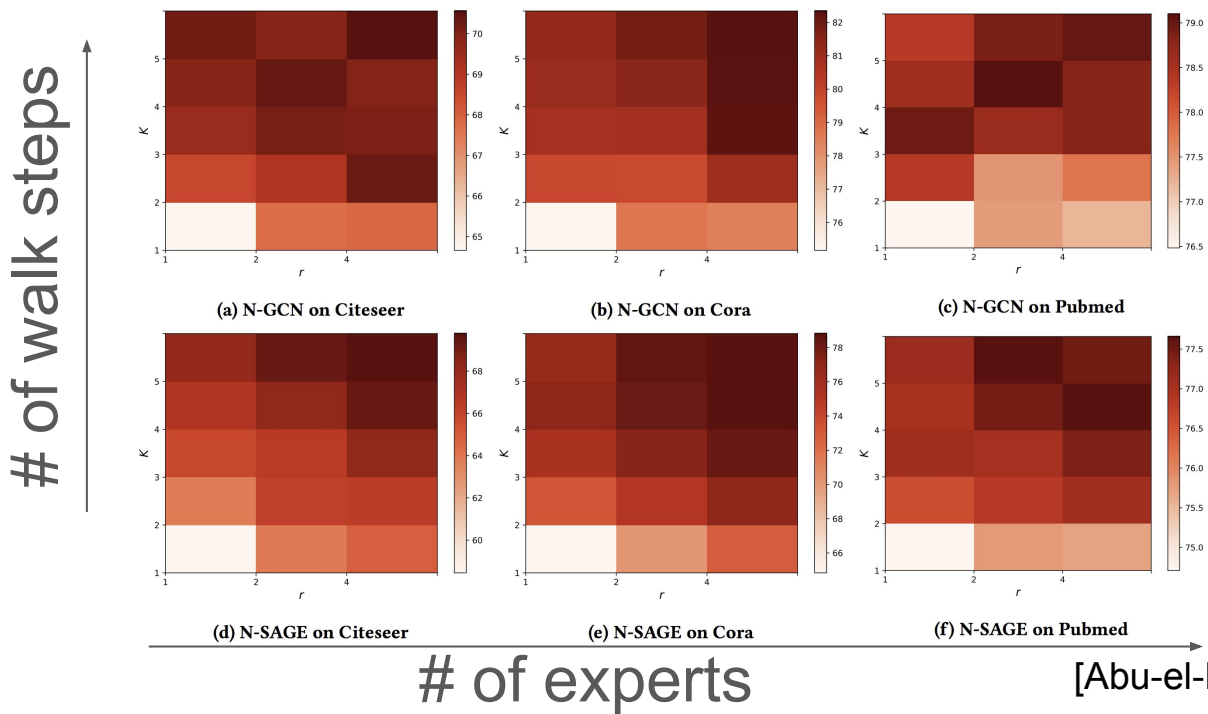
Fig: Architecture of Network of GCNs (N-GCN)



|     | <b>Method</b>                     | <b>Citeseer</b> | <b>Cora</b> | <b>Pubmed</b> | <b>PPI</b>  |
|-----|-----------------------------------|-----------------|-------------|---------------|-------------|
| (a) | ManiReg (Belkin et al., 2006b)    | 60.1            | 59.5        | 70.7          | –           |
| (b) | SemiEmb (Weston et al., 2012)     | 59.6            | 59.0        | 71.1          | –           |
| (c) | LP (Zhu et al., 2003)             | 45.3            | 68.0        | 63.0          | –           |
| (d) | DeepWalk (Perozzi et al., 2014)   | 43.2            | 67.2        | 65.3          | –           |
| (e) | ICA (Lu & Getoor, 2003)           | 69.1            | 75.1        | 73.9          | –           |
| (f) | Planetoid (Yang et al., 2016)     | 64.7            | 75.7        | 77.2          | –           |
| (g) | GCN (Kipf & Welling, 2017)        | 70.3            | 81.5        | 79.0          | –           |
| (h) | SAGE-LSTM (Hamilton et al., 2017) | –               | –           | –             | 61.2        |
| (i) | SAGE (Hamilton et al., 2017)      | –               | –           | –             | 60.0        |
| (j) | DCNN (our implementation)         | 71.1            | 81.3        | 79.3          | 44.0        |
| (k) | GCN (our implementation)          | 71.2            | 81.0        | 78.8          | 46.2        |
| (l) | SAGE (our implementation)         | 63.5            | 77.4        | 77.6          | 59.8        |
| (m) | N-GCN (ours)                      | <b>72.2</b>     | <b>83.0</b> | <b>79.5</b>   | 46.8        |
| (n) | N-SAGE (ours)                     | 71.0            | 81.8        | 79.4          | <b>65.0</b> |

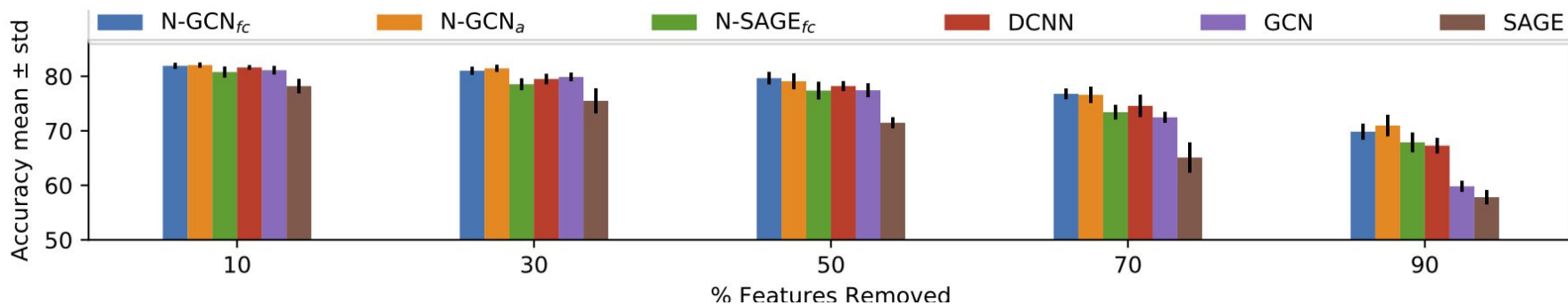
# Sensitivity Analysis

- Accuracy VS random walk steps (K) VS replication factor (r)

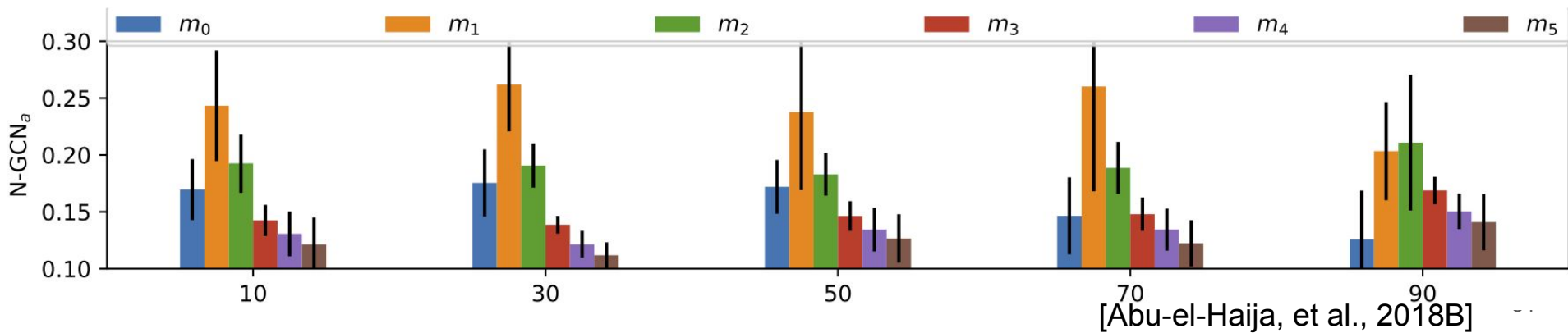


# Experiment: Input Perturbations [c]

Removing features degrades performance of baselines more than NGCN



N-GCN assigns more weight to further nodes when features are removed



# In this Section

## 1. Semi-Supervised Learning w/ Graph Embeddings

- a. Unsupervised Embeddings + Training Data
- b. Graphs as Regularizers
- c. Graph Convolutional Approaches
  - i. Overview
  - ii. GCNs
  - iii. Extensions

## 2. **Supervision as Inspiration**

- a. The Graph as Supervision
- b. Watch Your Step

Looking Forward: Graphs as Supervision

# Graph as Supervision: Loss Function

One can think of *the known edges* as *supervision*.

E.g., in Link Prediction, we assume that graph is *partially* observed, with goal of completing it: ranking missing (hidden) positive edges above negative ones.

It is possible to train unsupervised embeddings using an edge function  $g(u, v) \in \{0, 1\}$ , which outputs 1 if an edge exists and 0 otherwise.

One such example is the Graph Likelihood objective, which is a “supervised” loss function [Abu-el-Haija, et al., CIKM 2017].

# Probabilistic Graph Likelihood: Derivation

- Assume  $g : V \times V \rightarrow [0, 1]$  is an edge estimator.
- If  $g()$  is “accurate”, then likelihood below will equal to 1 when evaluated on  $A$ :

$$\Pr(G) = \prod_{(u,v) \in A} g(u, v) \prod_{(u,v) \notin A} 1 - g(u, v)$$

# Probabilistic Graph Likelihood: Derivation

- Assume  $g : V \times V \rightarrow [0, 1]$  is an edge estimator.
- If  $g()$  is “accurate”, then likelihood below will equal to 1 when evaluated on  $A$ :

$$\Pr(G) = \prod_{(u,v) \in A} g(u, v) \prod_{(u,v) \notin A} 1 - g(u, v)$$

- Equation above can be written as:


$$\Pr(G) = \prod_{(u,v)} g(u, v)^{\mathbf{1}[(u,v) \in A]} (1 - g(u, v))^{\mathbf{1}[(u,v) \notin A]}$$



# Probabilistic Graph Likelihood: Derivation

$$\begin{aligned}\Pr(G) &= \prod_{(u,v) \in A} g(u, v) \prod_{(u,v) \notin A} 1 - g(u, v) \\ &= \prod_{(u,v)} g(u, v)^{\mathbf{1}[(u,v) \in A]} (1 - g(u, v))^{\mathbf{1}[(u,v) \notin A]}\end{aligned}$$

# Probabilistic Graph Likelihood: Derivation

$$\begin{aligned}\Pr(G) &= \prod_{(u,v) \in A} g(u,v) \prod_{(u,v) \notin A} 1 - g(u,v) \\ &= \prod_{(u,v)} g(u,v)^{\mathbf{1}[(u,v) \in A]} (1 - g(u,v))^{\mathbf{1}[(u,v) \notin A]}\end{aligned}$$


## Graph Likelihood

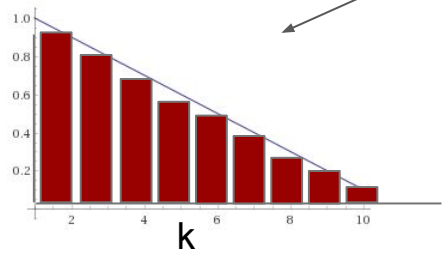
$$\Pr(G) = \prod_{(u,v)} g(u,v)^{\mathcal{D}_{u,v}} (1 - g(u,v))^{\mathbf{1}[(u,v) \notin A]}$$

$\mathcal{D}_{u,v}$  is frequency of  $u$  and  $v$  are co-visited in random walks.  
i.e. is # of times that  $u$  is sampled in  $v$ 's context

# Context Distribution: Sampling Window

- What is  $\mathcal{D}_{u,v}$  really ?

$$\mathbb{E} [\mathcal{D}^{\text{DEEPWALK}}] \propto \sum_{k=1}^C \left[ 1 - \frac{k-1}{C} \right] \mathcal{T}^k$$



Random Walk  
Transition Matrix

Fixed Context Distribution

# Context Distributions

Training with DeepWalk yields (in expectation) co-visit statistics matrix:

$$\mathbb{E} [\mathcal{D}^{\text{DEEPWALK}}] \propto \sum_{k=1}^C \left[ 1 - \frac{k-1}{C} \right] \mathcal{T}^k$$

Training with GloVe [Pennington, et al., EMNLP 2014] yields (in expectation) co-visit statistics matrix:

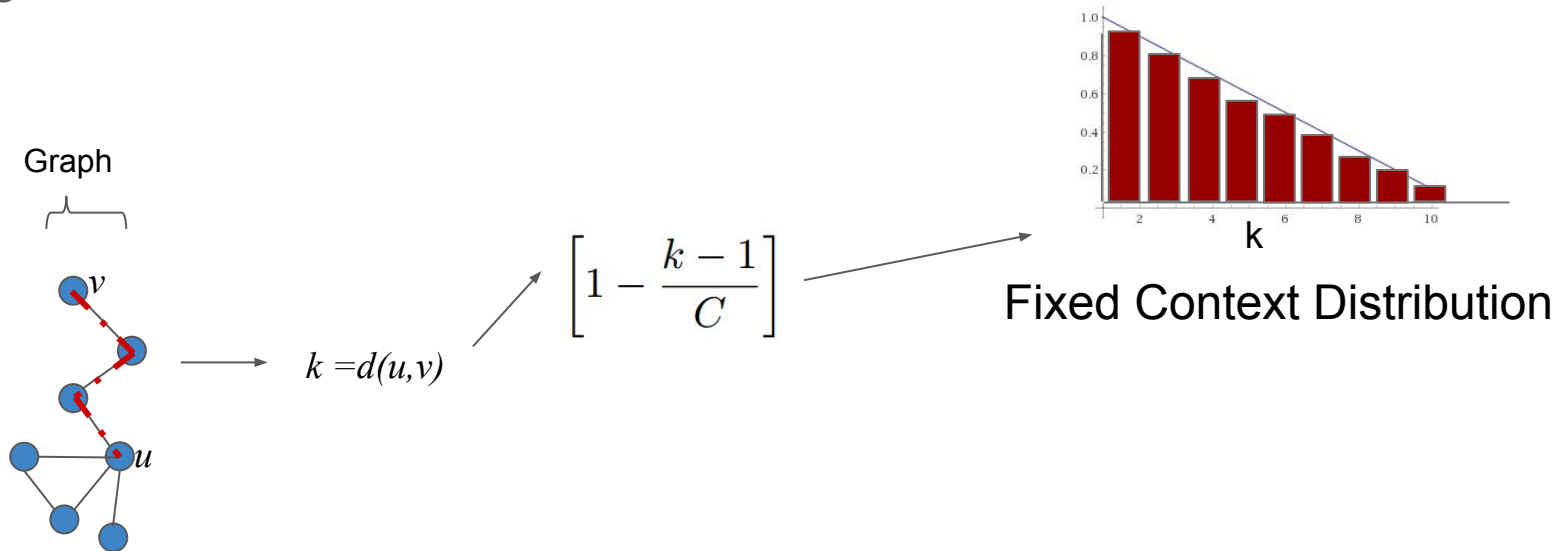
$$\mathbb{E} [\mathcal{D}^{\text{GloVe}}] \propto \sum_{k=1}^C \frac{1}{k} \mathcal{T}^k$$

Which one is better? What do we choose for the context distribution?

- Option 1: Hyper-parameter approach (change with each dataset)
- Option 2: Learn them from the graph!

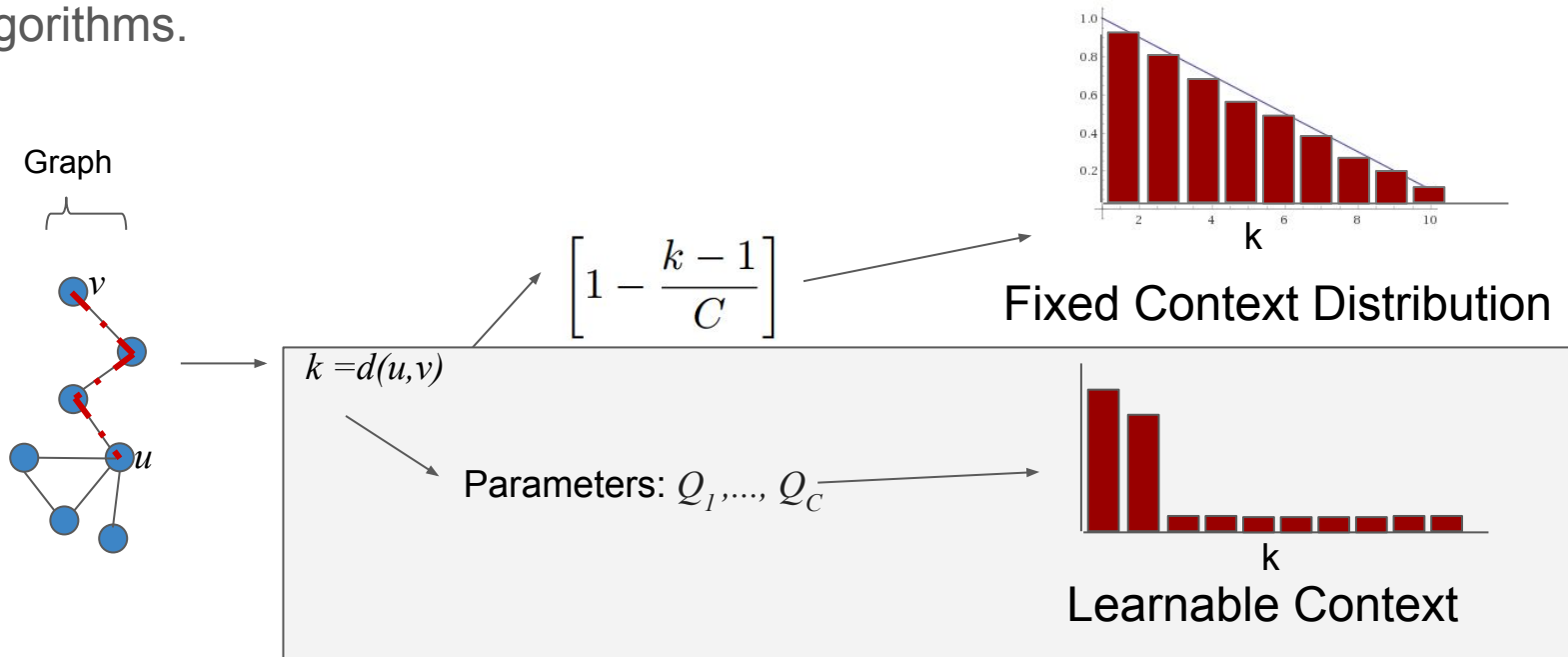
# Context Distribution is a Hidden Hyper-parameter!

Problem: Distance in random walk (# hops) hardcoded as importance in many algorithms.



# Watch Your Step: Learning Graph Embeddings Through Attention

Problem: Distance in random walk (# hops) hardcoded as importance in many algorithms.



# Learnable Context Distributions

Instead of hardcoding context distribution like previous work:

$$\mathbb{E} [\mathcal{D}^{\text{DEEPWALK}}] \propto \sum_{k=1}^C \left[ 1 - \frac{k-1}{C} \right] \mathcal{T}^k$$

Let's parametrize the expectation with a real positive vector  $Q$ :

$$\mathbb{E} [\mathbf{D}; Q_1, Q_2, \dots, Q_C] = \tilde{\mathbf{P}}^{(0)} \sum_{k=1}^C Q_k (\mathcal{T})^k$$

Under constraints:  $Q = (Q_1, Q_2, \dots, Q_C)$  with  $Q_k \geq 0$

E.g. we can set:  $(Q_1, Q_2, Q_3, \dots) = \text{softmax}((q_1, q_2, q_3, \dots))$ ,

# Learnable Context Distributions

Objective function:

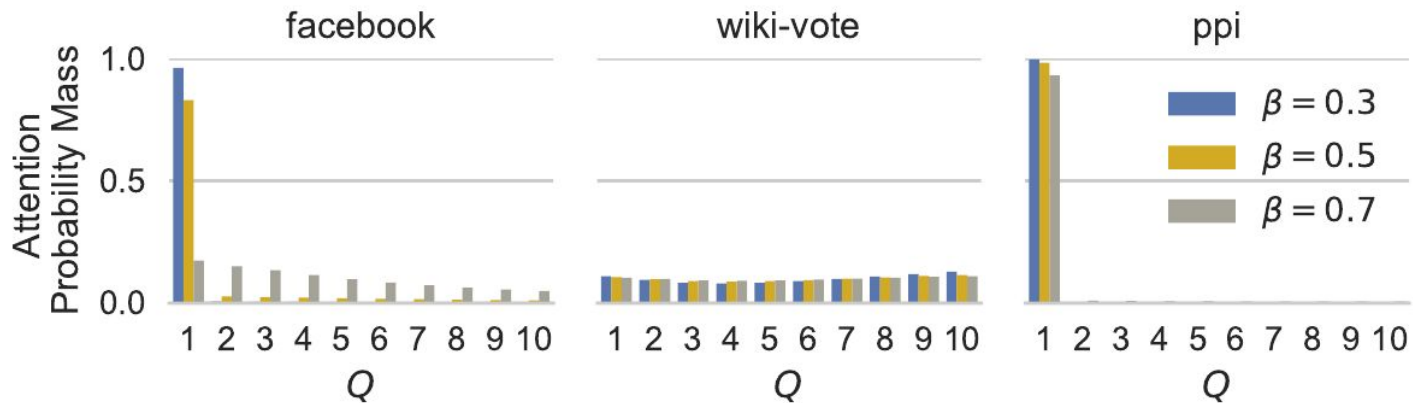
$$\min_{\mathbf{L}, \mathbf{R}, \mathbf{q}} \beta \|\mathbf{q}\|_2^2 + \left\| -\mathbb{E}[\mathbf{D}; \mathbf{q}] \circ \log(\sigma(\mathbf{L} \times \mathbf{R}^T)) - \mathbb{1}[\mathbf{A} = 0] \circ \log(1 - \sigma(\mathbf{L} \times \mathbf{R}^T)) \right\|_1$$

Where the attention parameters in vector  $\mathbf{q}=(q_1 \ q_2 \ \dots )$  is only used for training -- not inference (its not part of the node embeddings  $\mathbf{L}, \mathbf{R}$ ).

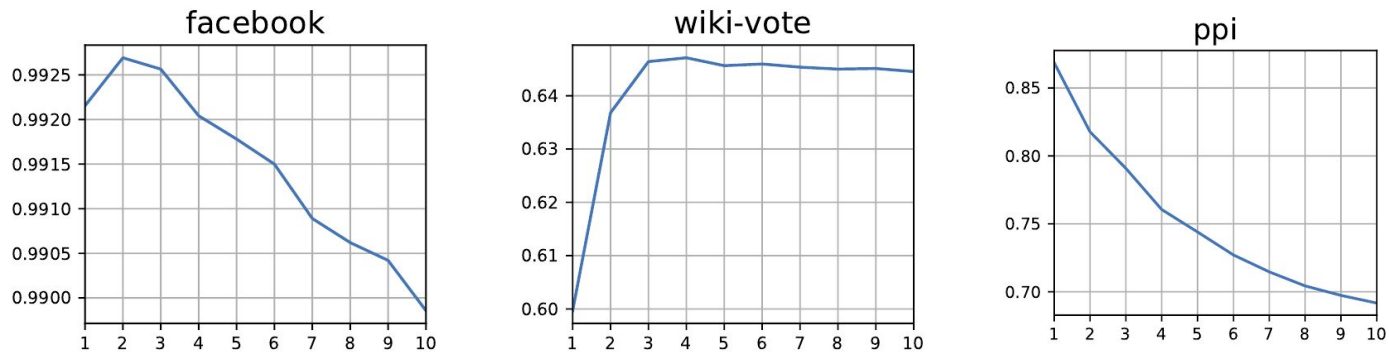


# Context Distributions: What does Q learn?

- Different distribution for every graph



- Distributions agree with optimal, if we sweep *window\_size* with node2vec.



# Learning Context Distribution: Link Prediction Results

| Dataset      | dim | D by Simulation     |                     |              | Attention Walks ( <i>ours</i> )   |                                   |
|--------------|-----|---------------------|---------------------|--------------|-----------------------------------|-----------------------------------|
|              |     | node2vec<br>$W = 2$ | node2vec<br>$W = 5$ | Asym<br>Proj | $\lambda$ -decay (13)             | softmax (11)                      |
| wiki-vote    | 64  | 64.4                | 63.6                | 91.7         | <b>93.5 <math>\pm</math> 0.62</b> | <b>93.8 <math>\pm</math> 0.13</b> |
|              | 128 | 63.7                | 64.6                | 91.7         | 92.9 $\pm$ 0.73                   | <b>93.8 <math>\pm</math> 0.05</b> |
| ego-Facebook | 64  | 99.1                | 99.0                | 97.4         | <b>99.3 <math>\pm</math> 0.02</b> | <b>99.4 <math>\pm</math> 0.10</b> |
|              | 128 | 99.3                | 99.2                | 97.3         | 99.3 $\pm$ 0.03                   | <b>99.5 <math>\pm</math> 0.03</b> |
| ca-AstroPh   | 64  | 97.4                | 96.9                | 95.7         | <b>98.6 <math>\pm</math> 0.03</b> | 97.9 $\pm$ 0.21                   |
|              | 128 | 97.7                | 97.5                | 95.7         | <b>98.6 <math>\pm</math> 0.03</b> | 98.1 $\pm$ 0.49                   |
| ca-HepTh     | 64  | 90.6                | 91.8                | 90.3         | 91.4 $\pm$ 0.17                   | <b>93.6 <math>\pm</math> 0.06</b> |
|              | 128 | 90.1                | 92.0                | 90.3         | 92.2 $\pm$ 0.18                   | <b>93.9 <math>\pm</math> 0.05</b> |
| PPI          | 64  | 79.7                | 70.6                | 82.4         | <b>90.0 <math>\pm</math> 0.03</b> | <b>89.8 <math>\pm</math> 1.05</b> |
|              | 128 | 81.8                | 74.4                | 83.9         | 90.4 $\pm$ 0.06                   | <b>91.0 <math>\pm</math> 0.28</b> |

# Takeaways

1. Supervision can create embeddings that are good for a downstream task (e.g. node labeling)
2. Very active field of research
3. Supervision can provide cool inspiration for better unsupervised methods

# References

[\[Abu-el-Haija, et al., CIKM 2017\]](#)

[\[Abu-el-Haija, et al., 2018A\]](#)

[\[Abu-el-Haija, et al., 2018B\]](#)

[\[Bronstein et al., 2016\]](#)

[\[Bruna et al., 2014\]](#)

[\[Bui et al, WSDM'18\]](#)

[\[Hamilton, et al., NIPS 2017\]](#)

[\[Hammond, et al., 2009\]](#)

[\[Kipf & Welling, ICLR 2017\]](#)

[\[Pennington, et al., EMNLP 2014\]](#)

[\[Perozzi, et al., KDD 2014\]](#)

[\[Scarselli et al. 2009\]](#)

[\[Veličković, et al., ICLR 2018\]](#)

[\[Yang, et al. ICML'16\]](#)

[\[Ying, et al., KDD 2018\]](#)

[\[Zhu, et al., ICML 2003\]](#)

# Whole Graph Representation Learning

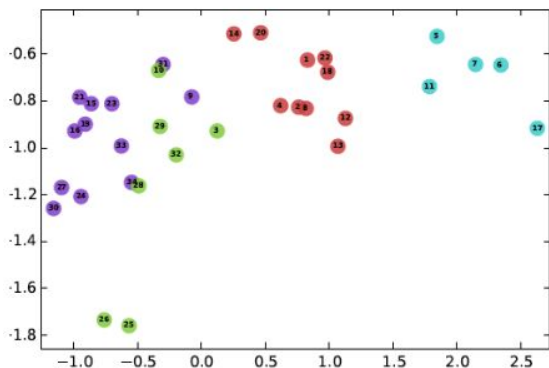
[Modeling Data With Networks + Network Embedding:  
Problems, Methodologies and Frontiers](#)

Ivan Brugere (University of Illinois at Chicago)  
Peng Cui (Tsinghua University)  
Bryan Perozzi (Google)  
Wenwu Zhu (Tsinghua University)  
Tanya Berger-Wolf (University of Illinois at Chicago)  
Jian Pei (Simon Fraser University)

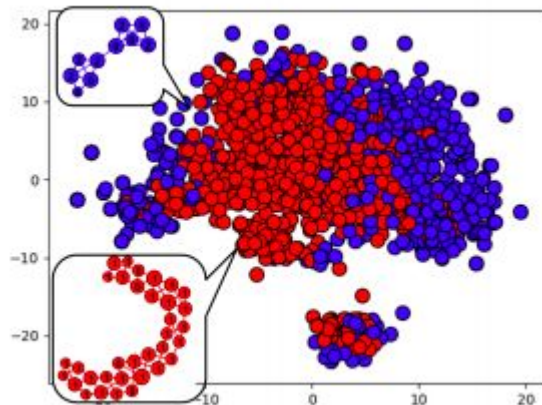
Bryan Perozzi  
[bperozzi@acm.org](mailto:bperozzi@acm.org)

# What does it mean to represent a whole graph?

Moving from a representation over nodes to an embedding for an entire graph.



**Node Representation  
(DeepWalk)**



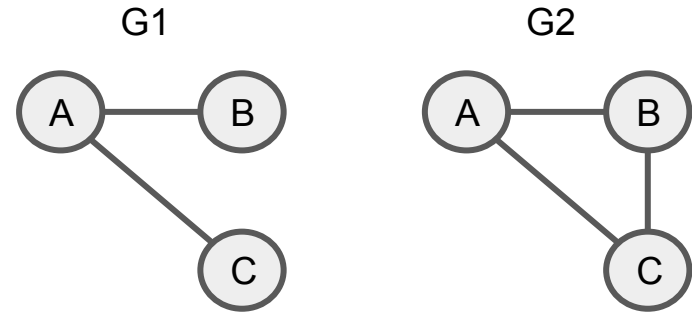
**Graph Representation  
[Taheri, Gimpel, Berger-Wolf -  
KDD'18]**

# Traditional Method 1: Graph Matching

Many similarity methods defined over graphs, e.g.

## Graph Edit Distance:

How many {node,edge} {insertions,deletions} are needed to transform one graph into another?



Edit distance (G1,G2) = 1

# Traditional Method 2: Graph Kernels

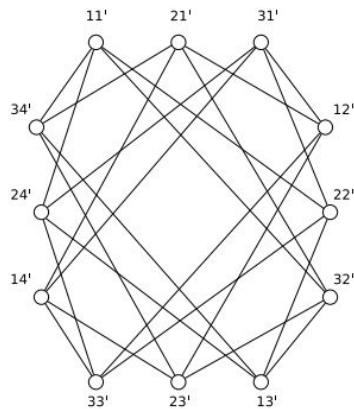
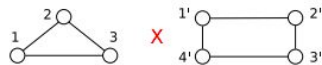
Idea: Define a kernel between graphs that captures their similarity.

Example: Random Walk Graph Kernel:

Given a pair of graphs, perform random walks on both (at once), and count the number of matching walks.

Pros: Elegant mathematical formalism

Cons: Scalability (even efficient methods  $O(N^3)$ )



direct product graph



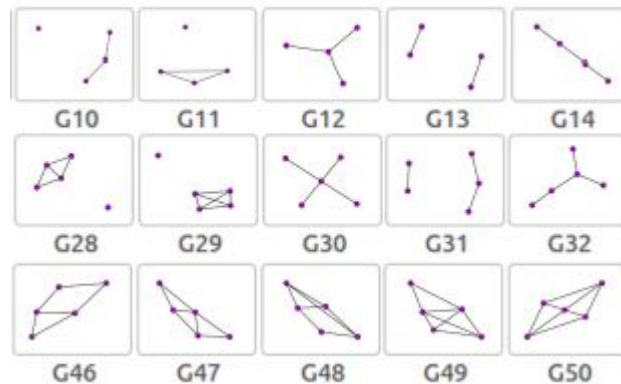
# Deep Graph Kernel

Idea:

Decompose graph into list of discrete substructures. Learn similarity between substructures using Skipgram.

Structure considered:

1. Graphlets (i.e. Motifs)
2. Shortest Paths
3. Weisfeiler-Lehman Kernels



Learned substructure similarity matrix

$$\mathcal{K}(\mathcal{G}, \mathcal{G}') = \phi(\mathcal{G})^T \mathcal{M} \phi(\mathcal{G}')$$

$\phi(\mathcal{G})$

Counts of graph substructures

# PATCHY-SAN

Idea: Linearize local graph structure, so traditional convolution can be applied.

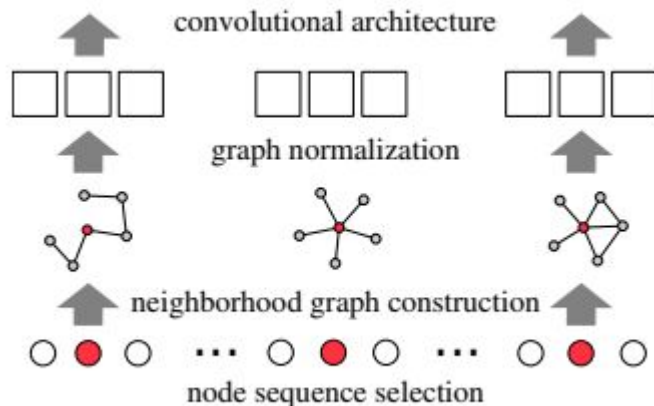
Convolutional architecture to predict graph's label (e.g. just like an image's label).

Pros:

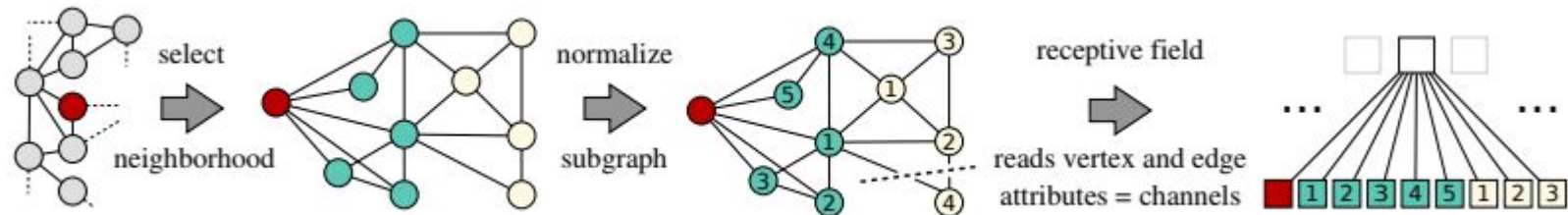
Leverage architecture from image classification

Cons:

Requires external graph linearization routine



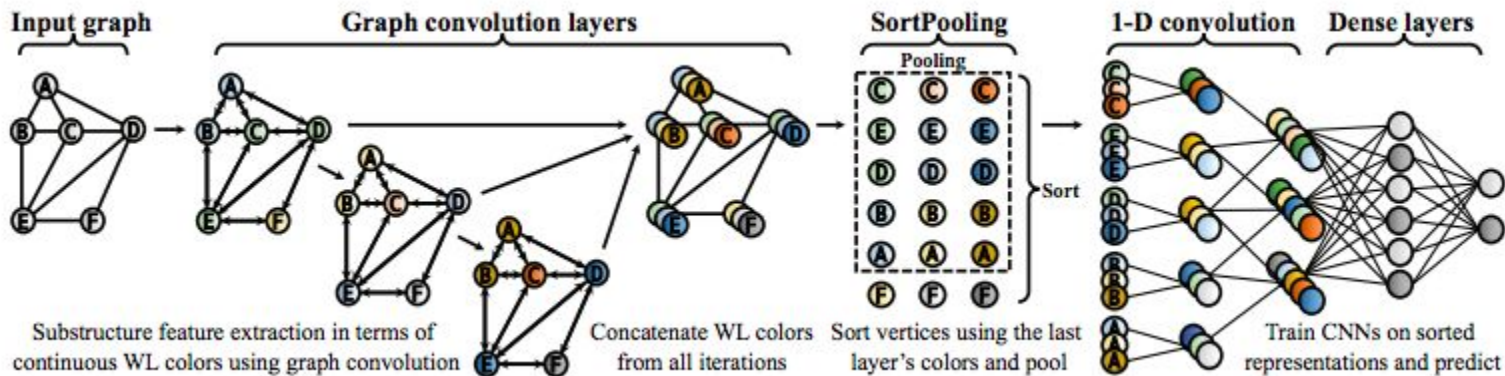
# PATCHY-SAN: Algorithm Overview



1. Order nodes
2. Select Neighborhood
3. Linearize Neighborhood
  - a. 1-dimensional Weisfeiler-Lehman routine (heuristic)
4. Apply standard convolutional architecture

# DGCNN

Combine information from the Weisfeiler-Lehman kernel, with a pooling layer inspired by PATCHY-SAN.

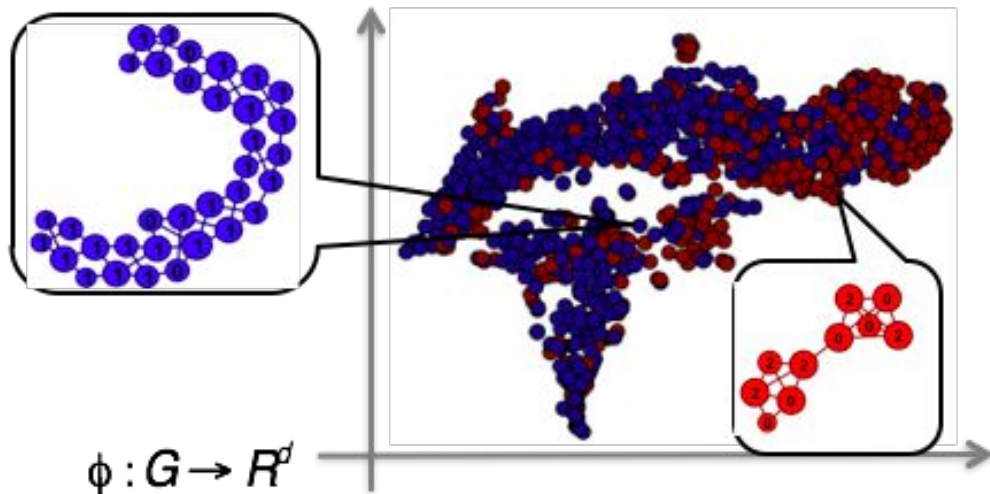


# Sequence Modeling for Graphs

Use an LSTM to encode observed similarity pairs from a graph.

1. Random Walks
2. Shortest Paths
3. Breadth First Search

The latent space of these models is a graph representation.



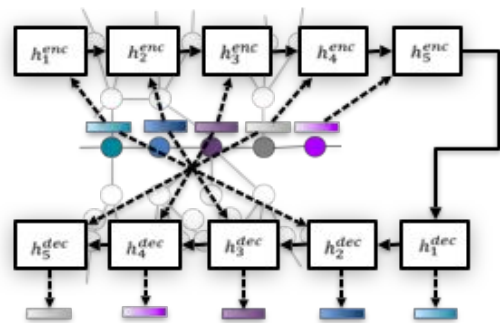
Learned graph representations in Proteins [Borgwardt 2005] dataset, which has two classes: enzyme (red) and non-enzyme (blue).

# LSTM Sequence Encoding

Choice of subgraph structure



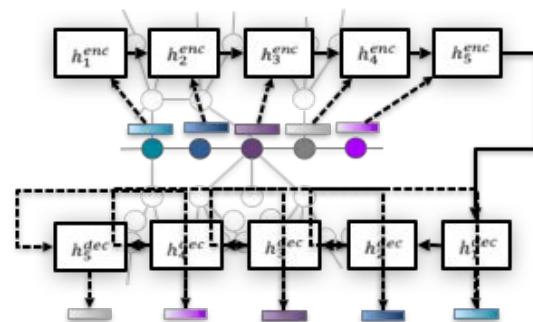
S2S Autoencoder



$$h_t^{enc} = LSTM_{enc}(Emb(v_t), h_{t-1}^{enc})$$

$$h_t^{dec} = LSTM_{dec}(Emb(v_{t-1}), h_{t-1}^{enc})$$

S2S Autoencoder Previously Predicted



$$h_t^{dec} = LSTM_{dec}(Emb(\bar{v}_{t-1}), h_{t-1}^{enc})$$

# References

[\[A. Sanfeliu; K-S. Fu \(1983\)\]](#)

[\[Niepert et al, ICML 2016\]](#)

[\[P. Yanardag and S. Vishwanathan, KDD'15\]](#)

[\[Taheri, Gimpel, Berger-Wolf, KDD'18\]](#)

[\[Vishwanathan et al, 2010\]](#)

[\[Zhang et al, AAAI'18\]](#)