

Geometric deep learning

P. Pala and S. Berretti

University of Florence - Italy

Outline of course content

Prerequisites: Python programming, some basic knowledge of Deep Neural Networks

Adopted tools: Pytorch geometric, Colab

Topics

- Geometric learning: intro and motivations
- Graph Convolutional Networks: Spatial and Spectral approaches
- GNN message passing model
- Attention in GNN
- Graph pooling
- Generalized convolutions: point cloud and meshes

Schedule

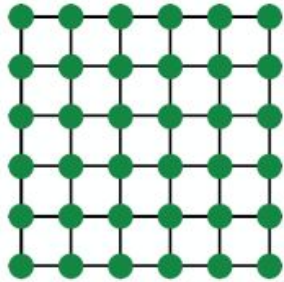
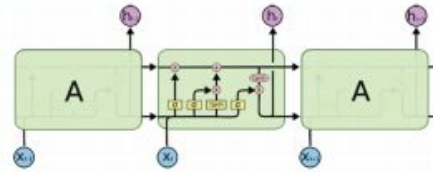
- July 11, 10.00 - 13.00 (P. Pala)
- July 12, 10.00 - 13.00 (S. Berretti)

Credits

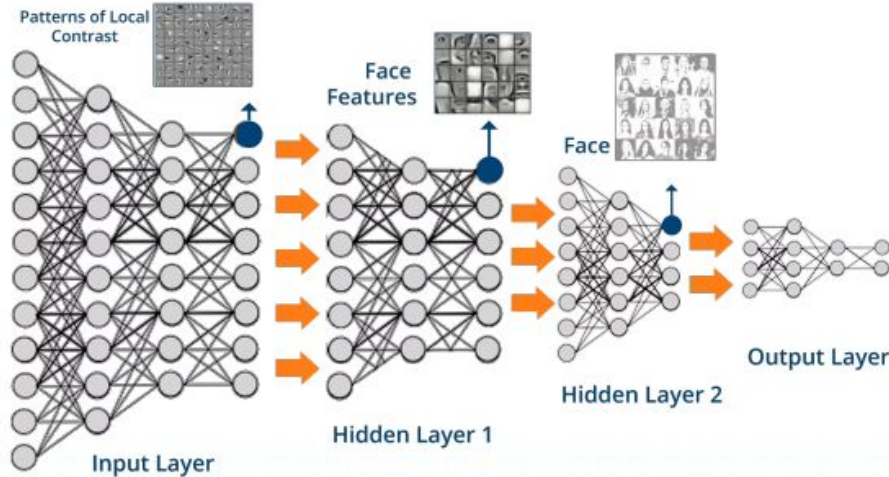
Course: Machine Learning with Graphs
Jure Leskovec, Stanford University
Book: Graph Representation Learning
William L. Hamilton, McGill University Ed.

Beyond grid structured data

Text/Speech

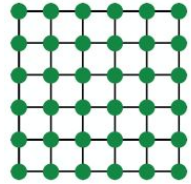
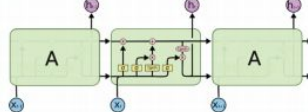
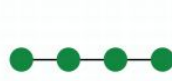


Images

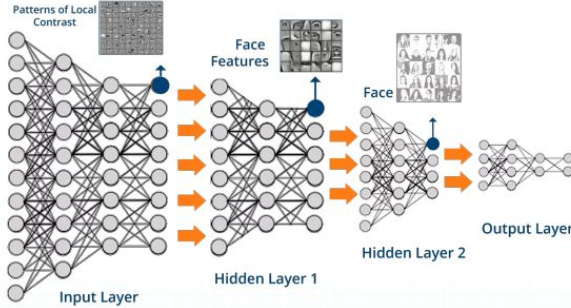


Beyond grid structured data

Text/Speech

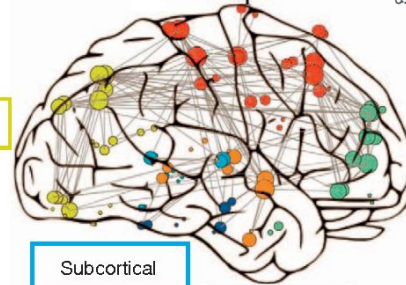


Images



Occipital

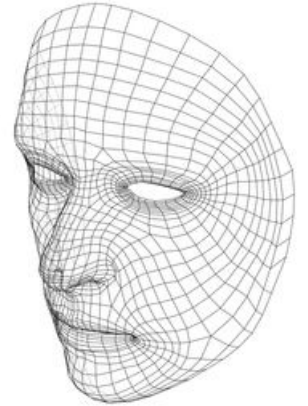
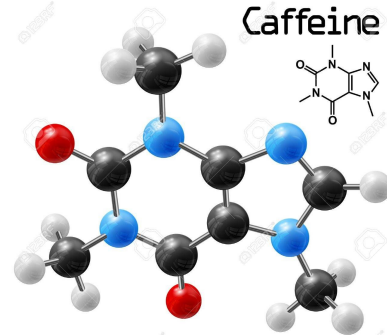
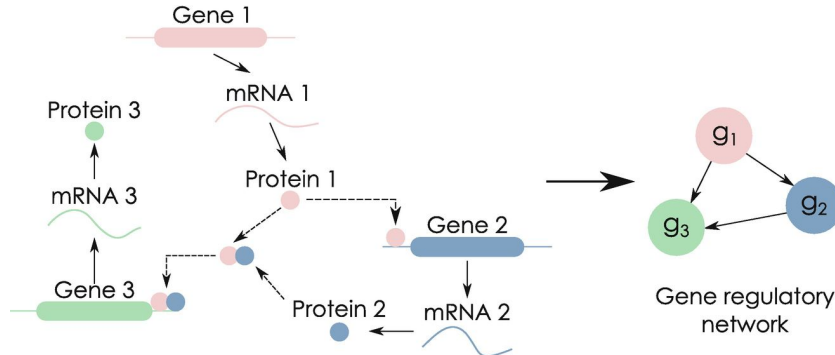
S/F RMI



Parietal

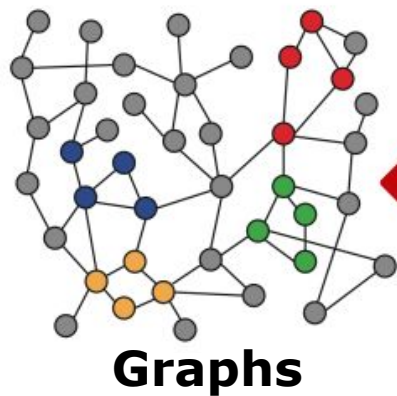
Frontal

Temporal

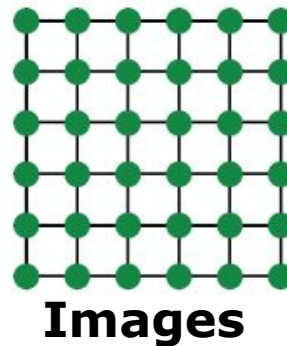


Graphs (Networks)

Arbitrary size, Complex topological structure, No fixed ordering or reference point.

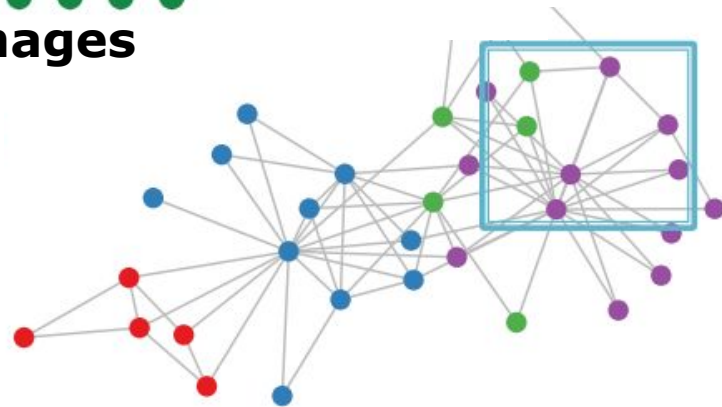


VS.



Text / Audio

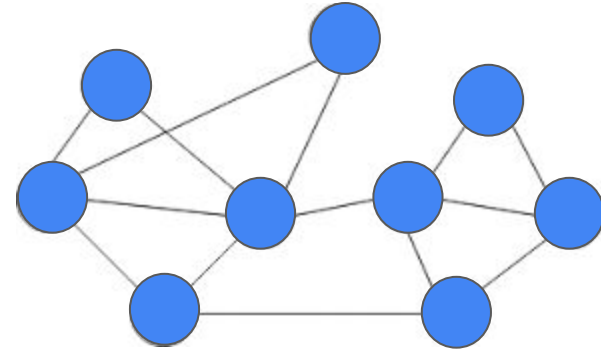
On a graph there is no fixed notion of locality or “sliding window”



Graph notation and representation structures

Graphs - Notation and properties

Formally, a graph $G=(V, E)$ is defined by a set of **nodes** V (sometimes referred to as **vertices**) and a set of **edges** E between these nodes.



We denote an edge going from node $u \in V$ to node $v \in V$ as $(u, v) \in E$.

The **degree** of a node $u \in V$ is the number of edges incident with the node u .

Simple graphs:

- there is one type of nodes and one type of edges
- there is at most one edge between each pair of nodes,
- no edges between a node and itself,
- all edges are undirected, i.e., $(u, v) \in E \leftrightarrow (v, u) \in E$.

Graphs - Notation and properties

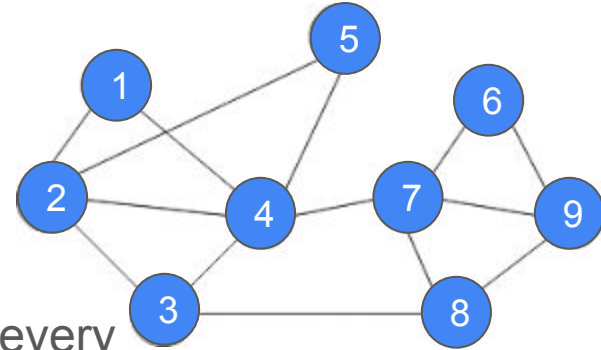
A convenient way to represent simple graphs is through an adjacency matrix $A \in \mathbb{R}^{|V| \times |V|}$.

For this purpose, we order the nodes in the graph so that every node indexes a particular row and column in the adjacency matrix.

- This ordering is totally arbitrary and different orderings yield different adjacency matrices

We can represent the presence of edges as entries in this matrix: $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise.

If the graph contains only undirected edges then A is a symmetric matrix, but if the graph is directed then A is not necessarily symmetric.

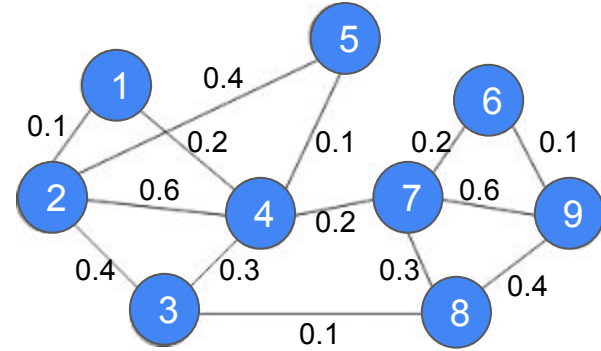


$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 &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

Graphs - Notation and properties

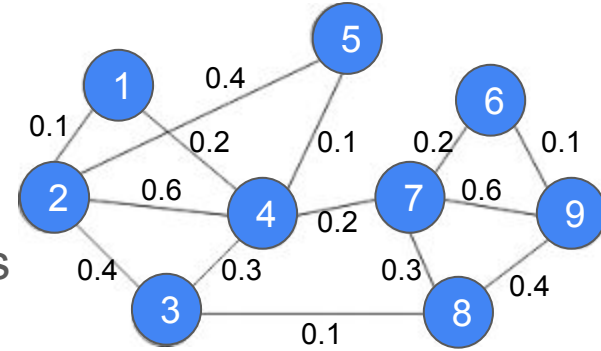
Some graphs can also have weighted edges where the entries in the adjacency matrix are arbitrary real-values rather than $\{0,1\}$.

For instance, a weighted edge in a protein-protein interaction graph might represent the strength of the association between two proteins.



Graphs - Notation and properties

Beyond the distinction between undirected, directed and weighted edges, graphs that have different types of edges can be considered.



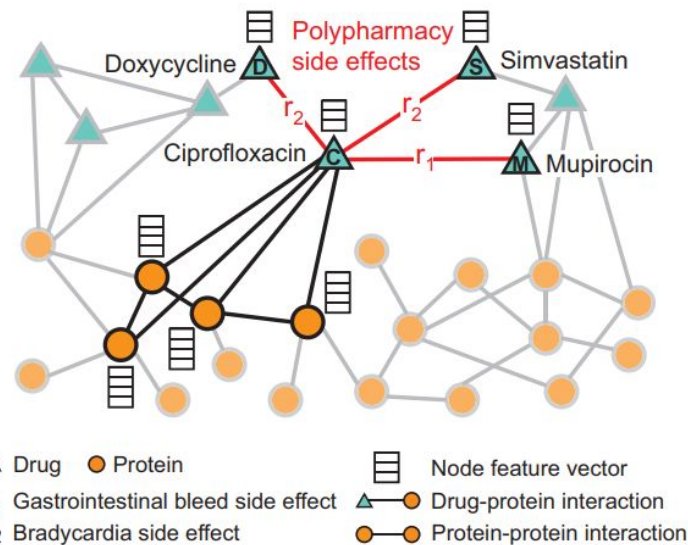
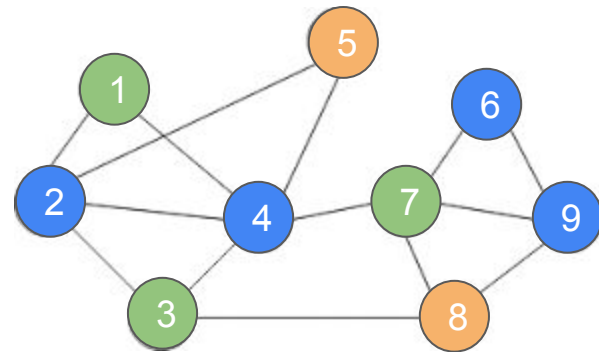
In these cases we can extend the edge notation to include an edge or relation type τ , e.g., $(u, v, \tau) \in E$, and we can define one adjacency matrix A_τ per edge type.

Multi-relational, $A \in \mathbb{R}^{|V| \times |V| \times |T|}$, where T is the set of edge types.

Graphs - Notation and properties

Some heterogeneous graphs may have nodes of different types, meaning that we can partition the set of nodes into disjoint sets $V = V_1 \cup V_2 \cup \dots \cup V_k$ where $V_i \cap V_j = \emptyset, \forall i \neq j$

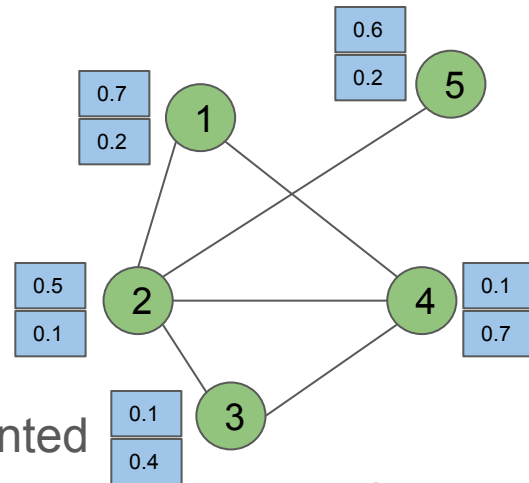
Edges in heterogeneous graphs generally satisfy constraints according to the node types, most commonly the constraint that certain edges only connect nodes of certain types



Graphs - Notation and properties

Last but not the least, typically we also have **feature information** associated with a graph.

Most often these are node-level attributes that are represented using a real-valued matrix $X \in \mathbb{R}^{|V| \times m}$, where we assume that the ordering of the nodes is consistent with the ordering in the adjacency matrix.



A =

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

X =

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

$$(A, X) \equiv (PA, PX)$$

P permutation matrix

Graphs - Notation and properties

Unnormalized Laplacian matrix:

$$\mathbf{L} = \mathbf{D} - \mathbf{A}$$

where \mathbf{A} is the adjacency matrix and \mathbf{D} is the **degree** matrix.

Important properties:

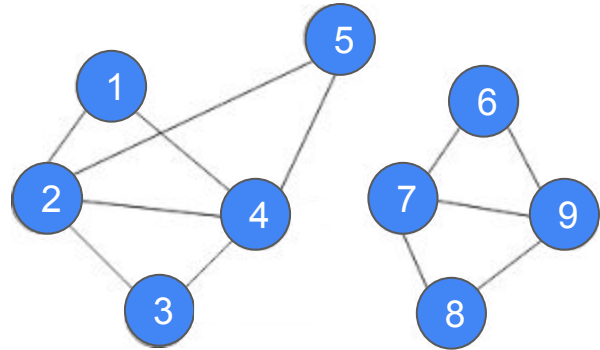
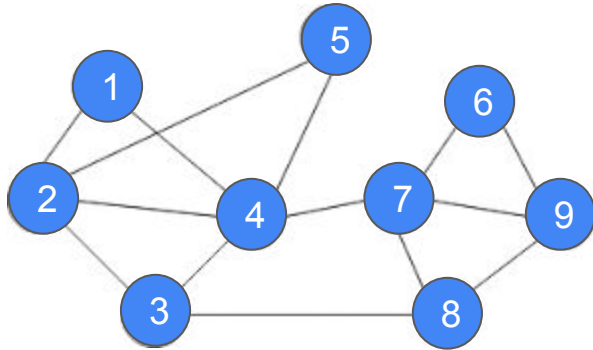
- It is symmetric and positive semi-definite
- At least one of the eigenvalues of \mathbf{L} is zero
- The following vector identity holds

$$\forall \mathbf{x} \in \mathbb{R}^{|V|}$$

$$\begin{aligned}\mathbf{x}^\top \mathbf{L} \mathbf{x} &= \frac{1}{2} \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \mathbf{A}[u, v] (\mathbf{x}[u] - \mathbf{x}[v])^2 \\ &= \sum_{(u, v) \in \mathcal{E}} (\mathbf{x}[u] - \mathbf{x}[v])^2\end{aligned}$$

Graphs - Notation and properties

Theorem: The geometric multiplicity of the 0 eigenvalue of the Laplacian L corresponds to the number of connected components in the graph.



Graphs - Notation and properties

Symmetric normalized Laplacian:

$$L_{\text{sym}} = D^{-1/2} L D^{-1/2} = I - D^{-1/2} A D^{-1/2}$$

Random walk Laplacian:

$$L_{\text{RW}} = D^{-1} L = I - D^{-1} A$$

Learning on graphs: Tasks

Node level classification/prediction

Edge level classification/prediction

Clustering and community detection

Graph level classification/prediction

Graph Representation Learning

- Graph Convolutional Networks
- Graph Neural Networks
- Graph Attention Networks

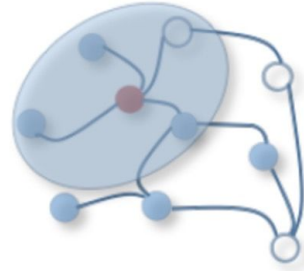
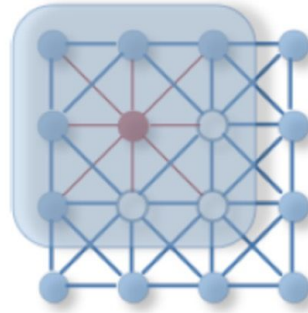
Graph Convolutional Networks

To begin with, we can try to generalize the CNN approach to operate on a graph structure.

Graph Convolution Networks (GCNs), draw on the idea of Convolution Neural Networks re-defining them for the non-euclidean data domain.

A regular Convolutional Neural Network captures the surrounding information of each pixel of an image.

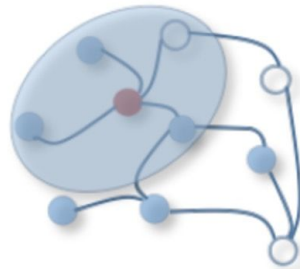
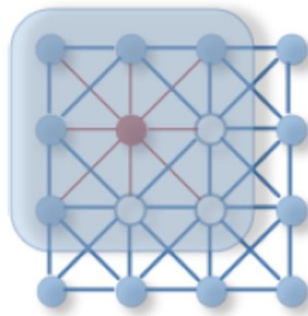
Similar to euclidean data like images, the convolution framework here aims to capture neighbourhood information for non-euclidean spaces like graph nodes.



Graph Convolutional Networks

Two types of approaches to GCNs:

- **Spatial GCNs:** Formulate graph convolutions as aggregating feature information from neighbours.
- **Spectral GCNs:** Define graph convolutions by introducing filters from the perspective of graph signal processing based on graph spectral theory.



Spatial-GCN

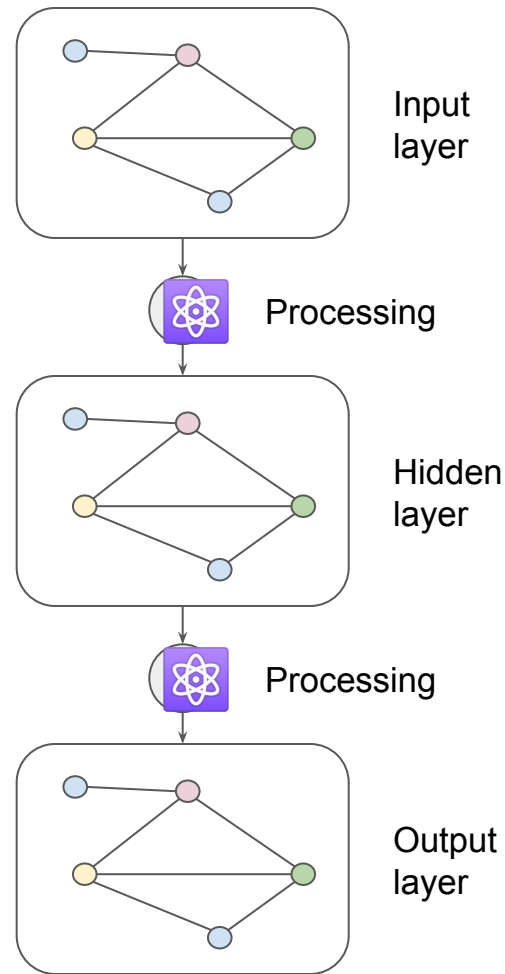
Graph Convolutional Networks

In a CNN, we apply a filter on the original image to get the representation in the next layer.

Similarly, in GCN, we apply a filter which creates node representations at the next layer.

- The next layer has the same topology of the first one (same nodes same edges)
- What changes is the feature information associated with each node

How to compute this new feature information?



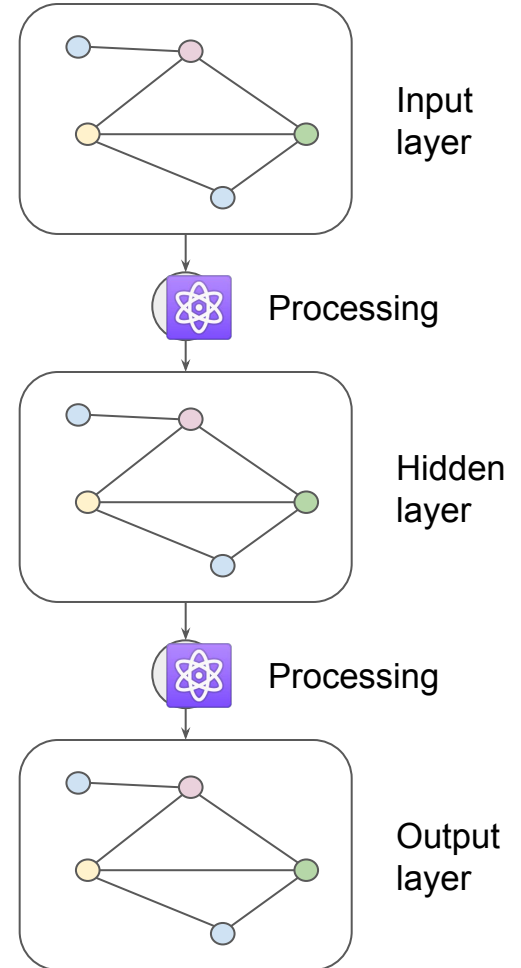
Graph Convolutional Networks

In a CNN, we apply a filter on the original image to get the representation in the next layer.

Similarly, in GCN, we apply a filter which creates node representations at the next layer.

$$H^{i+1} = f(H^i, A)$$

Adjacency matrix
Node features at layer i
Non linear function
Node features at layer $i+1$



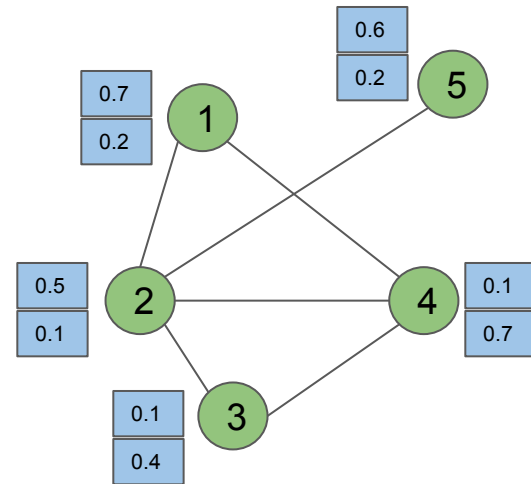
Graph Convolutional Networks

A =

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

X =

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2



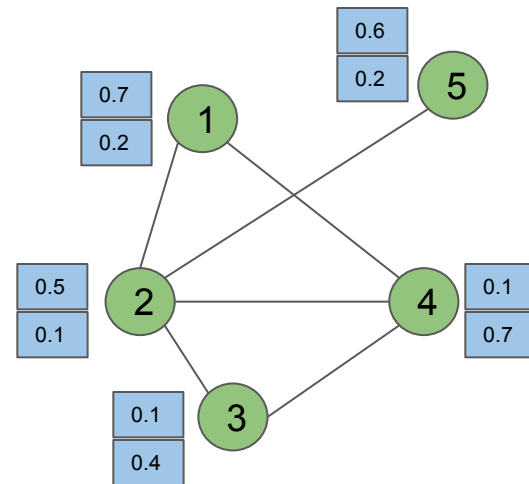
Graph Convolutional Networks

$A =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

$X =$

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2



$AX =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

$=$

0.6	

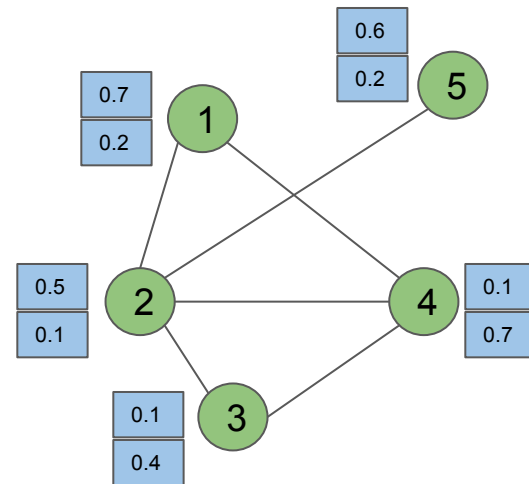
Graph Convolutional Networks

$A =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

$X =$

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2



$AX =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

$=$

0.6	0.8

Graph Convolutional Networks

$A =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

$X =$

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

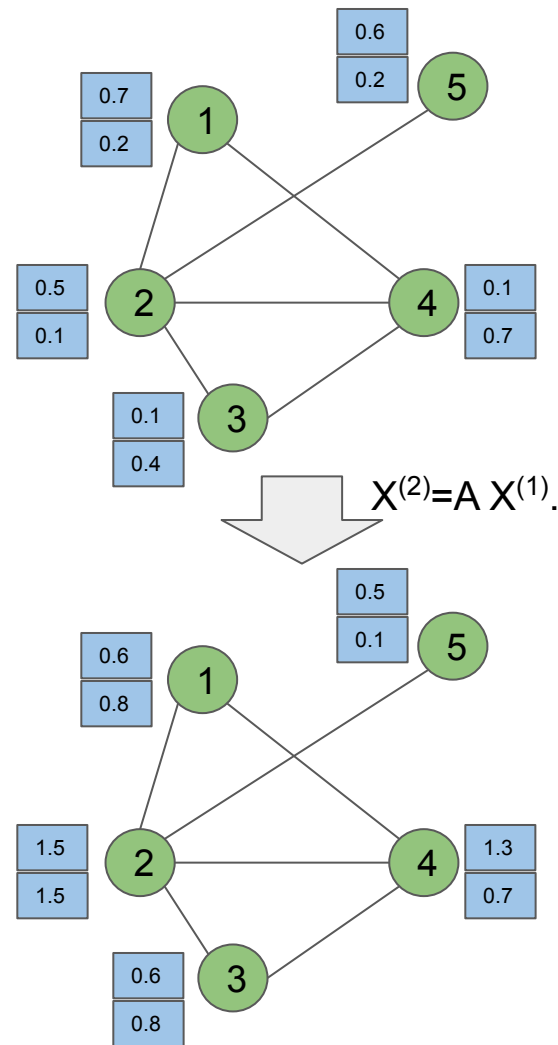
$AX =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

$=$

0.6	0.8
1.5	1.5
0.6	0.8
1.3	0.7
0.5	0.1



Graph Convolutional Networks

$A =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

$X =$

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

1	1	1
1	0	1
1	1	1

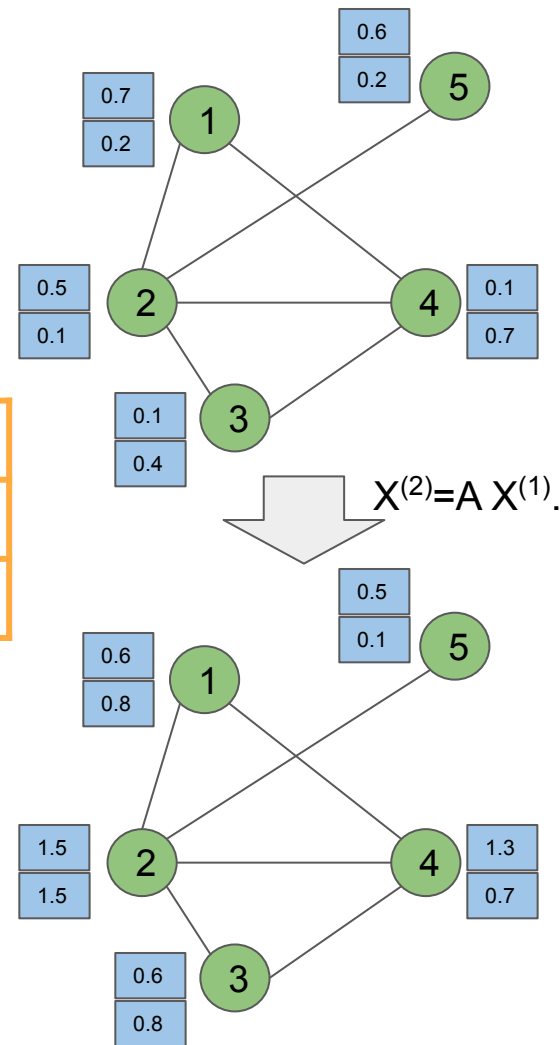
$AX =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

$=$

0.6	0.8
1.5	1.5
0.6	0.8
1.3	0.7
0.5	0.1



Graph Convolutional Networks

Noticeable issues:

1. The new representation of a node just accounts for neighboring nodes but not for the node itself
2. Nodes that have a large number of neighbours (higher degree) will have larger new values (e.g. node 2)

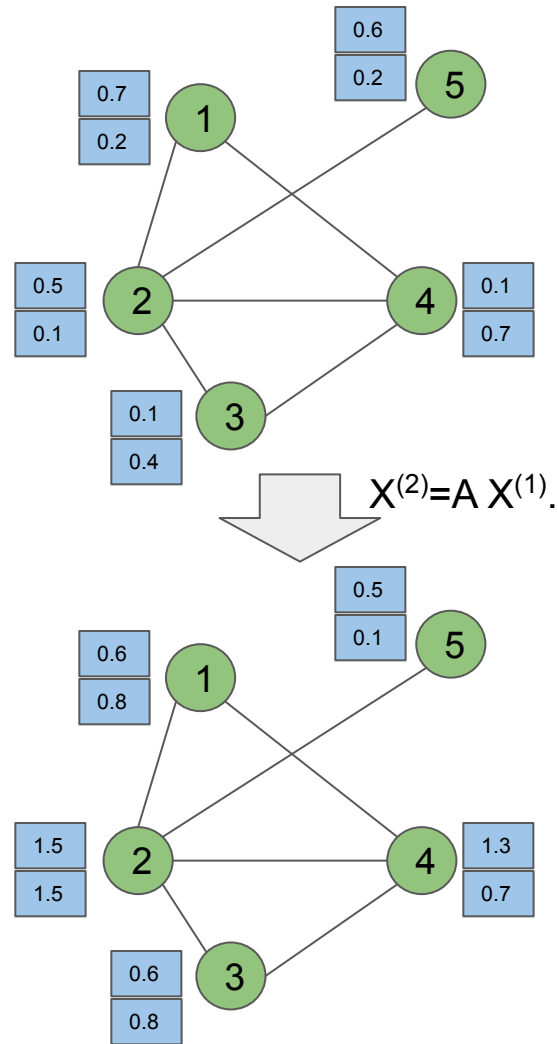
$AX =$

0	1	0	1	0
1	0	1	1	1
0	1	0	1	0
1	1	1	0	0
0	1	0	0	0

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

$=$

0.6	0.8
1.5	1.5
0.6	0.8
1.3	0.7
0.5	0.1



Graph Convolutional Networks

Noticeable issues:

1. The new representation of a node just accounts for neighboring nodes but not for the node itself
2. Nodes that have a large number of neighbours (higher degree) will have larger new values

Solution to 1

Self-loops are added.

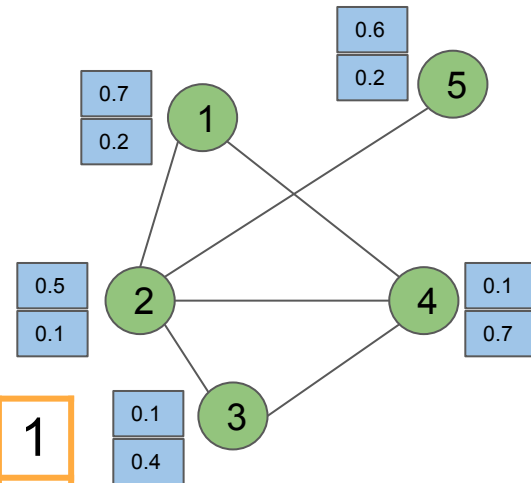
Mathematically, self-loops are expressed by adding the identity matrix to the adjacency matrix

$$\hat{A} = A + I$$

$\hat{A} =$

1	1	0	1	0
1	1	1	1	1
0	1	1	1	0
1	1	1	1	0
0	1	0	0	1

1	1	1
1	1	1
1	1	1



Graph Convolutional Networks

Noticeable issues:

1. The new representation of a node just accounts for neighboring nodes but not for the node itself
2. Nodes that have a large number of neighbours (higher degree) will have larger new values

Solution to 2

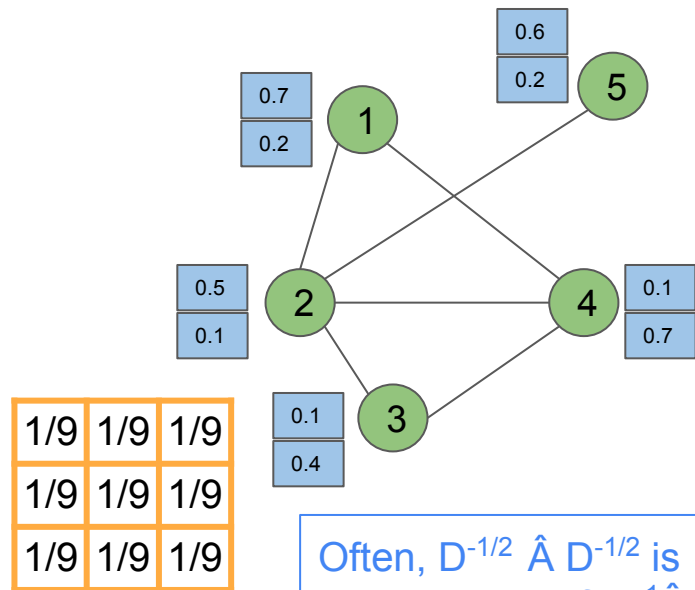
Use **normalisation**.

Normalise \hat{A} such that all rows sum up to one, i.e. $D^{-1}\hat{A}$, where D is the diagonal node degree matrix (of \hat{A}) $\hat{A} =$
Use of $D^{-1}\hat{A}$ corresponds to taking the average instead of the sum of neighboring node features.

1	1	0	1	0
1	1	1	1	1
0	1	1	1	0
1	1	1	1	0
0	1	0	0	1

$D^{-1} =$

1/3	0	0	0	0
0	1/5	0	0	0
0	0	1/3	0	0
0	0	0	1/4	0
0	0	0	0	1/2



Graph Convolutional Networks

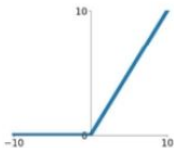
In a CNN, we apply a filter on the original image to get the representation in the next layer.

Similarly, in GCN, we apply a filter which creates the next layer representation.

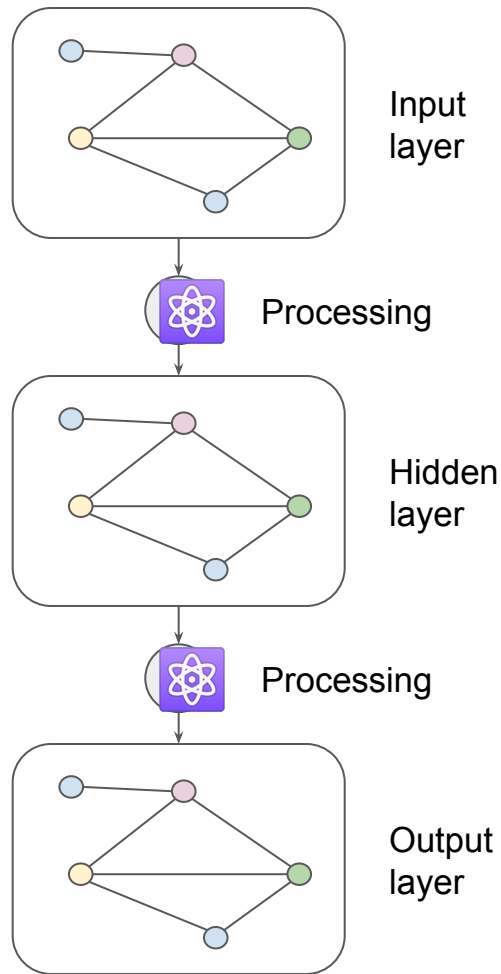
$$H^{i+1} = f(H^i, A)$$

For instance: $f(H^i, A) = \sigma(A H^i W^i)$

ReLU
 $\max(0, x)$



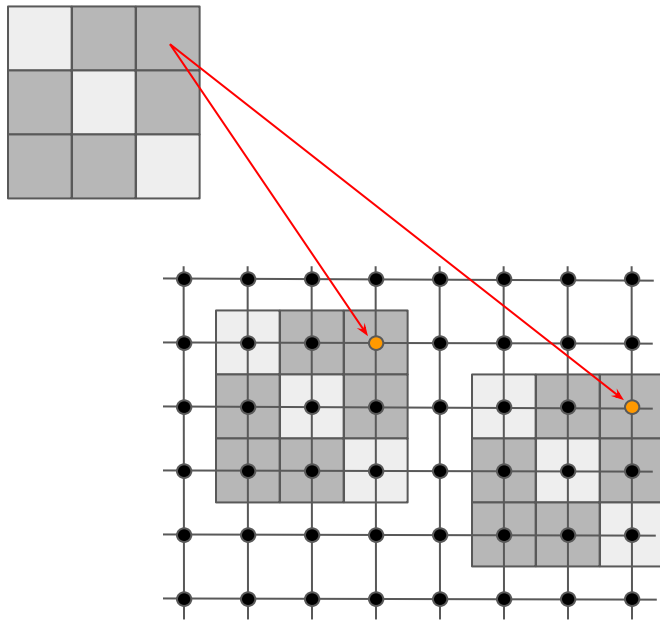
Trainable weight tensor for layer i
ReLU function



Spectral - GCN

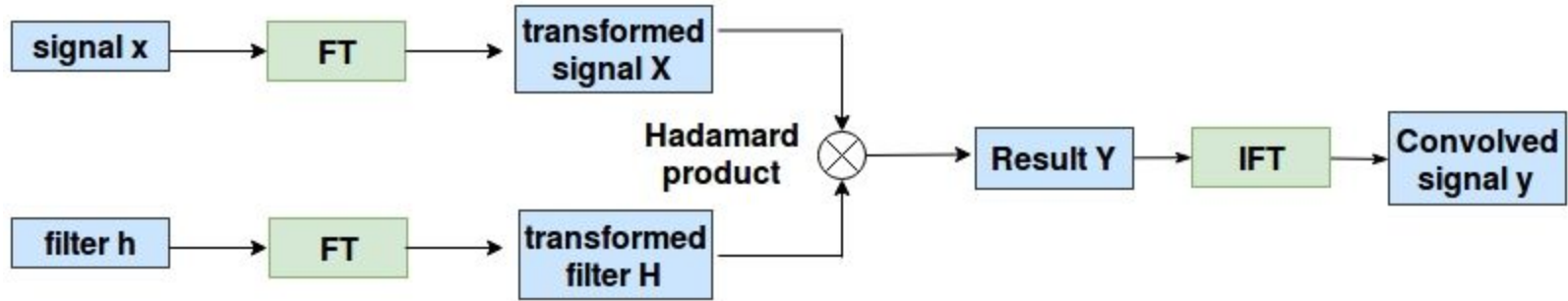
Spectral - GCN

Can we extend the convolution operator to act on graphs instead of grid-structured data?



Spectral - GCN

Can we extend the convolution operator to act on graphs instead of grid-structured data?



How can we define the Fourier transform in the domain of graphs?

Spectral - GCN

The **Laplace operator** of a function $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is defined as the divergence of the gradient of f


$$\begin{aligned}\Delta f(\mathbf{x}) &= \nabla^2 f(\mathbf{x}) \\ &= \sum_{i=1}^n \frac{\partial^2 f}{\partial x_i^2}\end{aligned}$$

Property of the Laplace operator:

$$-\Delta(e^{2\pi i s t}) = -\frac{\partial^2 (e^{2\pi i s t})}{\partial t^2} = (2\pi s)^2 e^{2\pi i s t}$$

The eigenfunctions of Δ are the same complex exponentials that make up the modes of the frequency domain in the Fourier transform.

Laplacian


$$\begin{aligned}\mathbf{x}^\top \mathbf{L} \mathbf{x} &= \frac{1}{2} \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \mathbf{A}[u, v] (\mathbf{x}[u] - \mathbf{x}[v])^2 \\ &= \sum_{(u, v) \in \mathcal{E}} (\mathbf{x}[u] - \mathbf{x}[v])^2\end{aligned}$$

Spectral - GCN

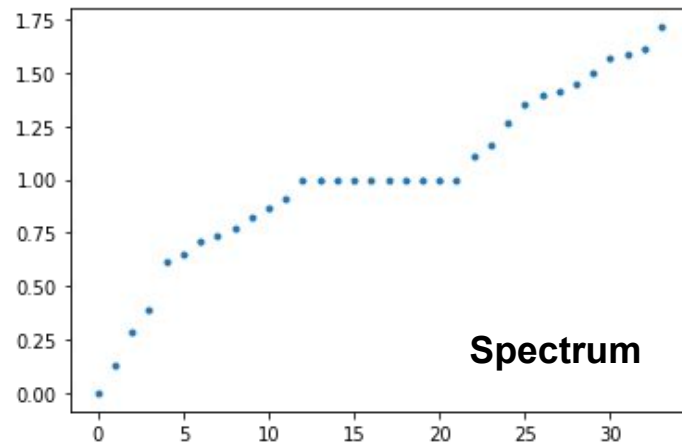
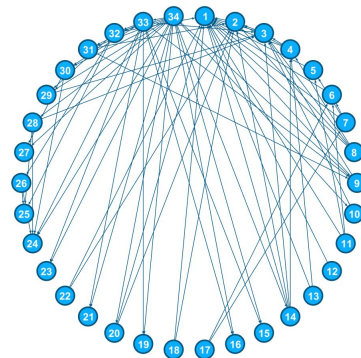
$$\mathbf{L}_{\text{sym}} = \mathbf{D}^{-1/2} \mathbf{L} \mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2} \mathbf{A} \mathbf{D}^{-1/2}$$

Eigendecomposition of the graph Laplacian (symmetric):

$$\mathbf{L} = \mathbf{\Phi}^T \mathbf{\Lambda} \mathbf{\Phi}$$

Graph **Fourier modes**: eigenvectors of the Laplacian (columns of $\mathbf{\Phi}$)

Graph **Spectrum**: eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \dots \lambda_n \leq 2$



Spectral - GCN

Transformation of a graph signal x to the Fourier domain: $\hat{x} = \Phi^T x$

Inverse Fourier transform: $x = \Phi \hat{x}$

Graph convolutions in the spectral domain are defined via point-wise products in the transformed Fourier space.

$$x *_G g = \mathcal{F}^{-1}(\mathcal{F}(x) \odot \mathcal{F}(g)) = \Phi(\underbrace{\underbrace{\Phi^T x}_{n \times 1} \odot \underbrace{\Phi^T g}_{n \times 1}}_{n \times 1})$$

$n \times 1$

Spectral - GCN

Transformation of a graph signal x to the Fourier domain: $\hat{x} = \Phi^T x$

Inverse Fourier transform: $x = \Phi \hat{x}$

Graph convolutions in the spectral domain are defined via point-wise products in the transformed Fourier space.

$$x *_G g = \mathcal{F}^{-1}(\mathcal{F}(x) \odot \mathcal{F}(g)) = \Phi(\underbrace{\underbrace{\Phi^T x}_{n \times 1} \odot \underbrace{\Phi^T g}_{n \times 1}}_{n \times 1})$$

$$\mathfrak{g} = \text{diag}(\vartheta_1, \dots, \vartheta_n) = \text{diag}(\Phi^T g)$$

Graph convolution simplifies to $\Phi \mathfrak{g} \Phi^T x$

- optimize \mathfrak{g} by backpropagation

Spectral - GCN - Challenges

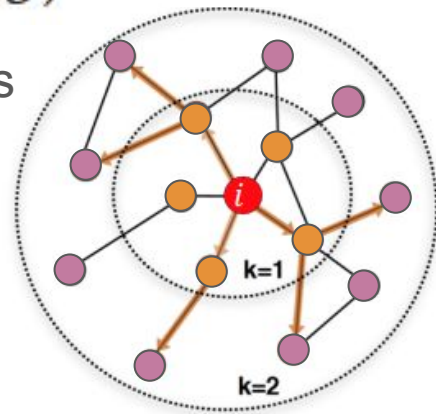
$$x *_G g = \mathcal{F}^{-1}(\mathcal{F}(x) \odot \mathcal{F}(g)) = \Phi(\Phi^T x \odot \Phi^T g) \quad , \quad \Phi \Phi^T x$$

1. The number of filter parameters to learn depends on the number of nodes of the graph.
2. The filters are not localized and refer to the entire graph
3. The algorithm needs to calculate the eigen-decomposition explicitly and multiply signal with Fourier basis. There is no Fast Fourier Transform algorithm defined for graphs, hence the computation is $O(n^2)$.

Spectral - GCN - Improvements

$$x *_G g = \mathcal{F}^{-1}(\mathcal{F}(x) \odot \mathcal{F}(g)) = \Phi(\Phi^T x \odot \Phi^T g) \quad \Phi \mathfrak{g} \Phi^T x$$

1. Instead of parameterizing the filter by n parameters, express it as a weighted combination of K smooth components
2. Instead of computing the eigendecomposition explicitly, the filter is expressed as a polynomial function computed recursively from the rescaled Laplacian



$$\Phi g(\bar{\Lambda}) \Phi^T = \sum_{k=0}^{K-1} \theta_k T_k(\bar{\Delta})$$

Some references on graph spectral filtering

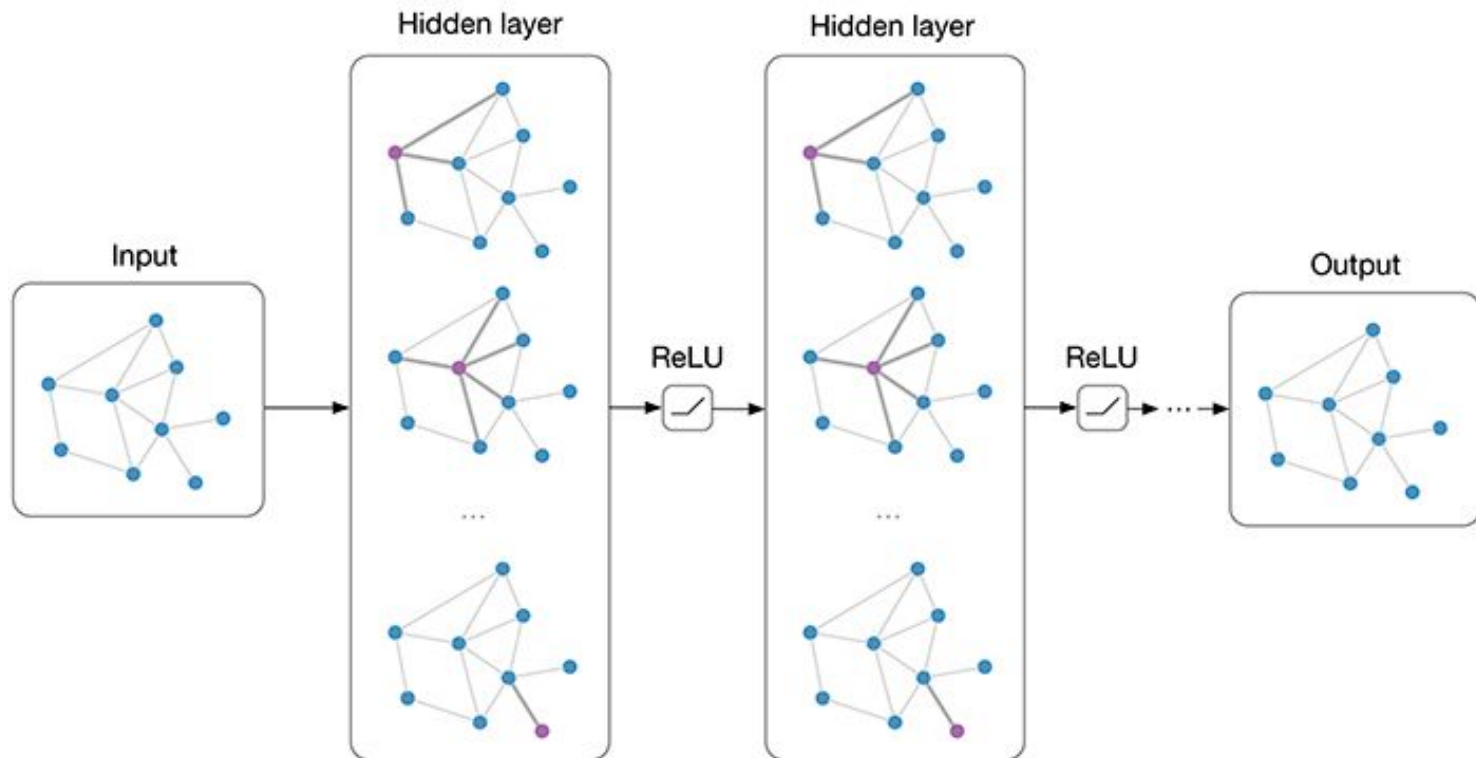
2014 - Joan Bruna, Wojciech Zaremba, Arthur Szlam, Yann LeCun:
Spectral Networks and Locally Connected Networks on Graphs. ICLR

2015 - Mikael Henaff, Joan Bruna, Yann LeCun:
Deep Convolutional Networks on Graph-Structured Data. CoRR abs/1506.05163

2016 - Michaël Defferrard, Xavier Bresson, Pierre Vandergheynst:
Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. NIPS

2019 - Ron Levie, Federico Monti, Xavier Bresson, Michael M. Bronstein:
CayleyNets: Graph Convolutional Neural Networks With Complex Rational Spectral Filters

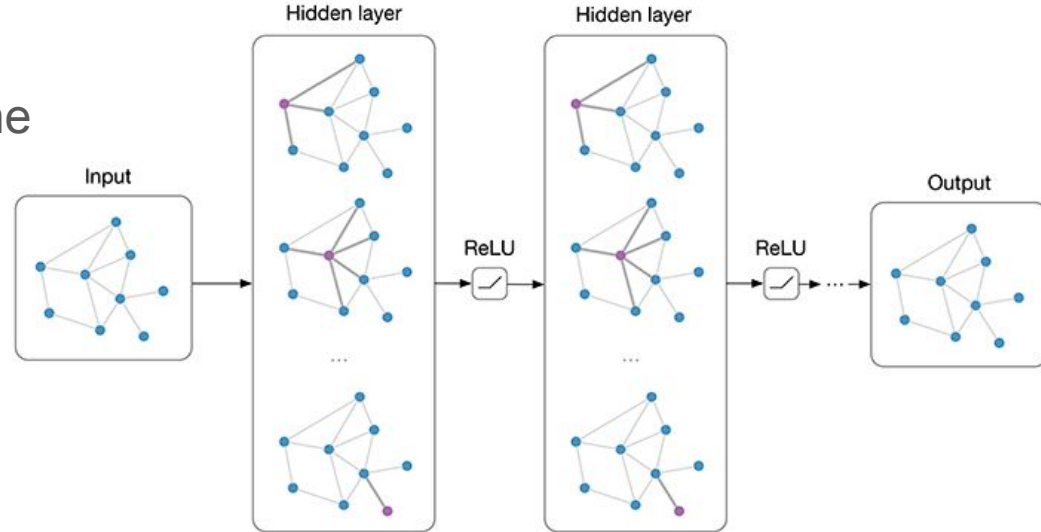
Graph Convolutional Networks - Remarks



Graph Convolutional Networks - Remarks

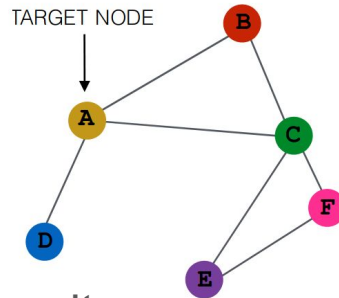
What if the graph is not static?

- A new node is added: what is the embedding of the new node?
- An edge is removed or added: how do the embeddings of the nodes change?

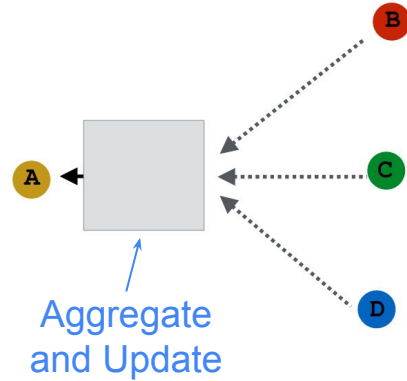


Graph Convolutional Networks - Remarks

To compute the embedding of node A at layer k, embeddings of neighboring nodes at the previous layer are **aggregated**.



It is as if each node receives a message from its neighboring nodes and updates its status (embedding) by aggregating these messages.



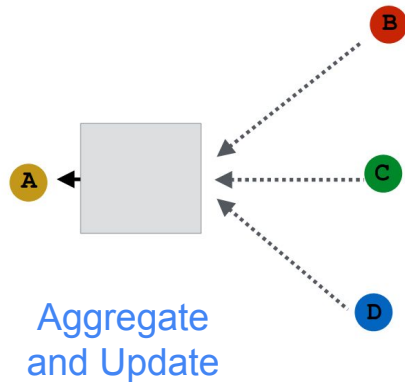
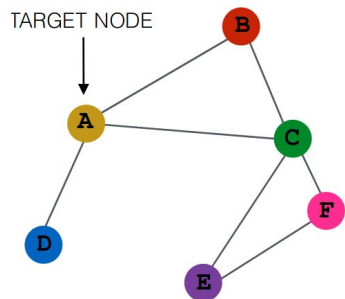
The Graph Neural Network model

Neural message passing

The key feature of a GNN is that it uses a form of neural message passing in which vector messages are exchanged between nodes and updated using neural networks.

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)} \left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\}) \right)$$

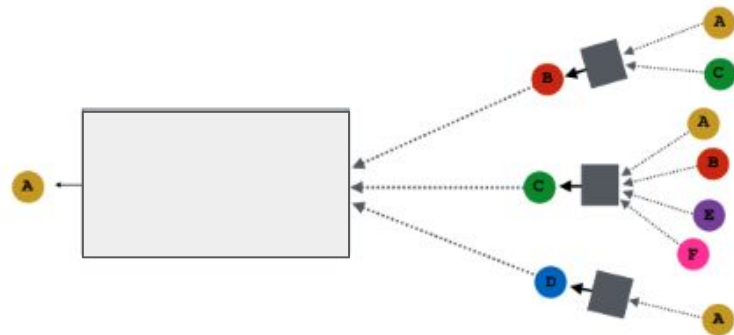
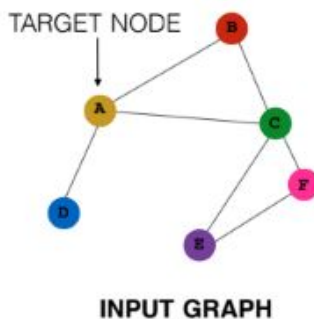
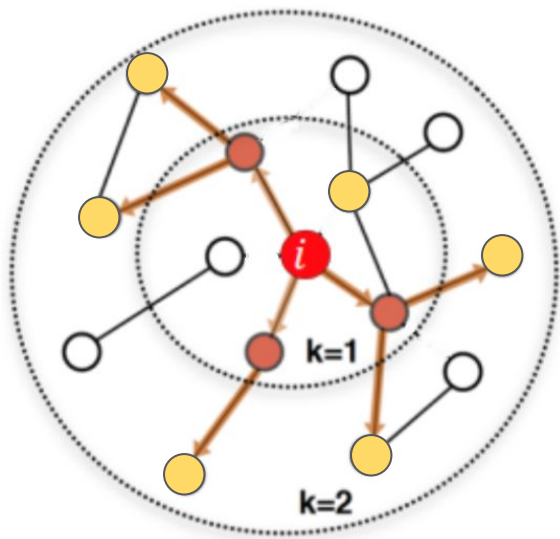
Every node defines a computation graph based on its neighborhood!



Neural message passing

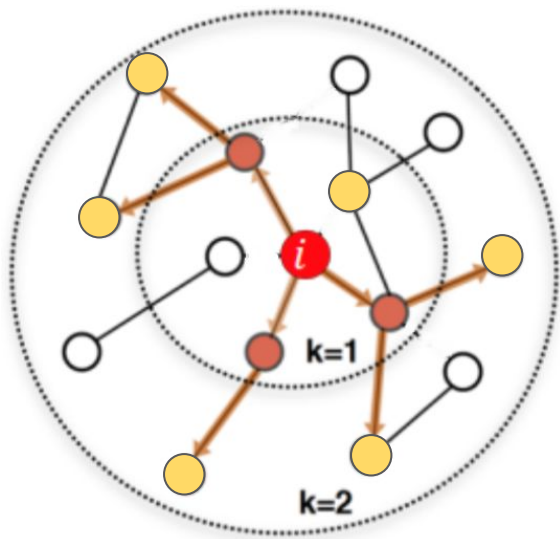
At each iteration, every node aggregates information from its local neighborhood, and as these iterations progress each node embedding contains more and more information from further reaches of the graph.

- Structural information
- Feature information

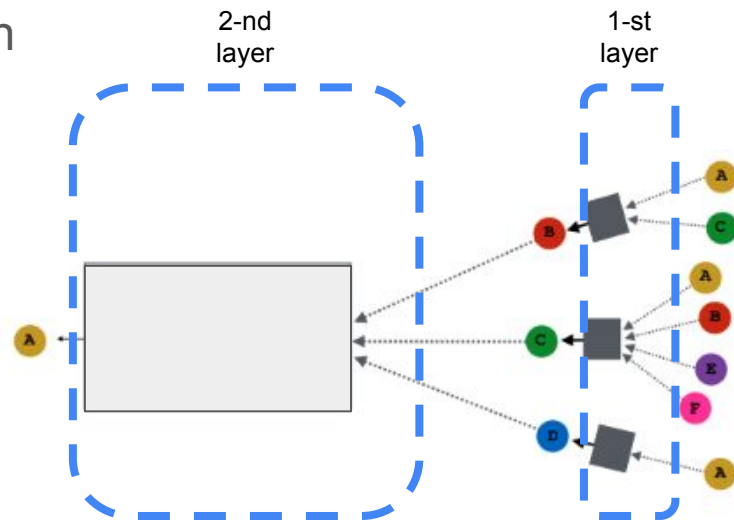
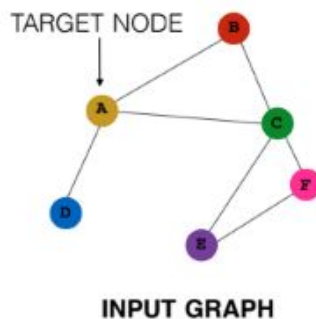


Neural message passing

At each iteration, every node aggregates information from its local neighborhood, and as these iterations progress each node embedding contains more and more information from further reaches of the graph.



- Structural information
- Feature information



Neural message passing

The new embedding of node u is computed by aggregating messages from neighboring nodes $v \in N(u)$

$$\mathbf{W}_{\text{neigh}}^{(k)} \sum_{v \in N(u)} \mathbf{h}_v^{(k-1)}$$

To avoid loss of information about node u its new embedding should also preserve a message from node u itself

$$\mathbf{W}_{\text{self}}^{(k)} \mathbf{h}_u^{(k-1)} .$$

Basic message passing

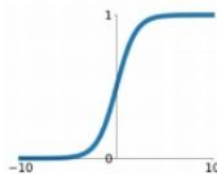
$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}_{\text{self}}^{(k)} \mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)} \right), \quad (5.7)$$

$$\mathbf{W}_{\text{self}}^{(k)}, \mathbf{W}_{\text{neigh}}^{(k)} \in \mathbb{R}^{d(k) \times d(k-1)}$$

σ is an elementwise non-linearity

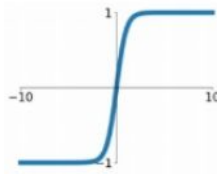
Sigmoid

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



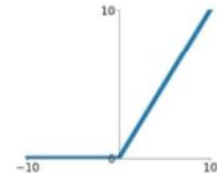
tanh

$$\tanh(x)$$



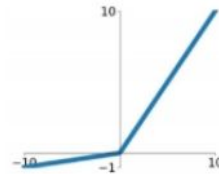
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

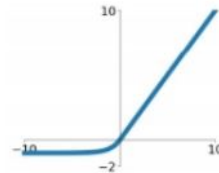


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

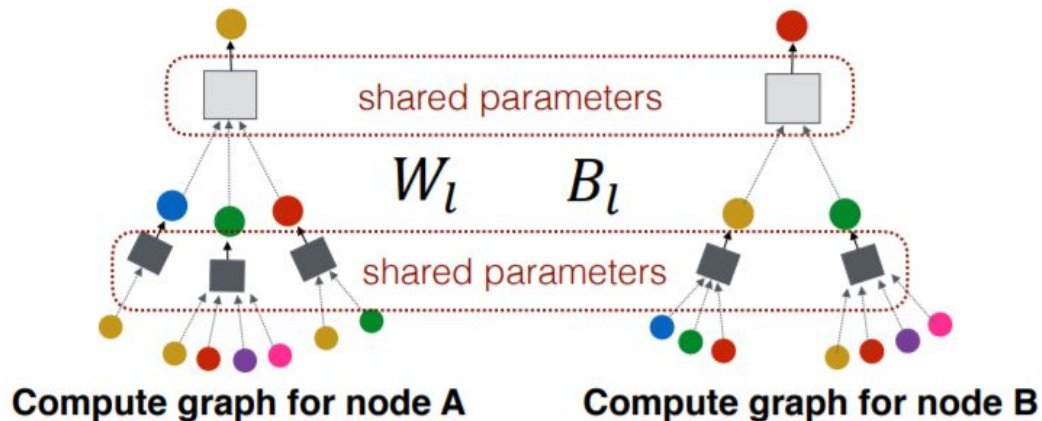
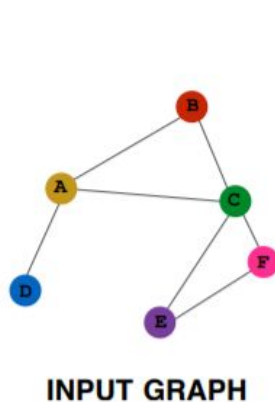
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Basic message passing

The same aggregation parameters are shared for all nodes of the same layer

- The number of model parameters is sublinear in $|V|$
- The model supports inference on new nodes



Inductive vs transductive capability

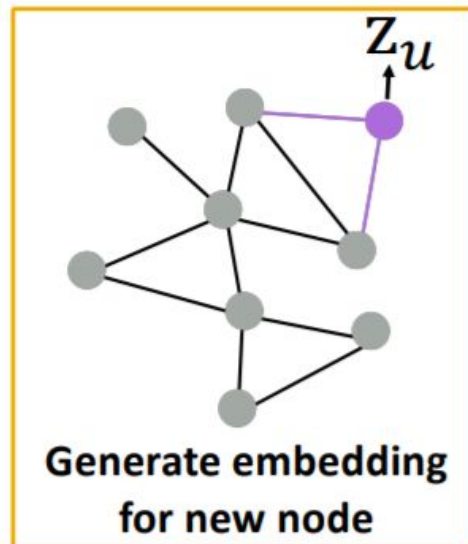
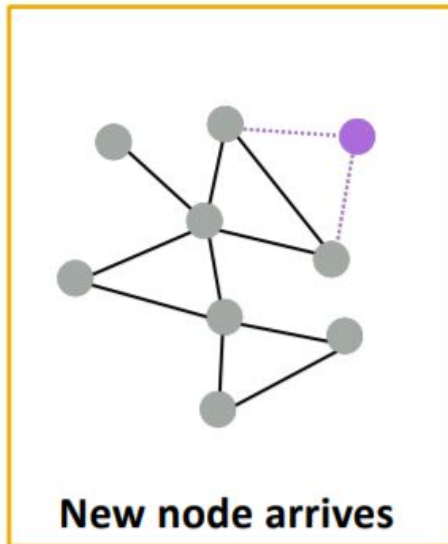
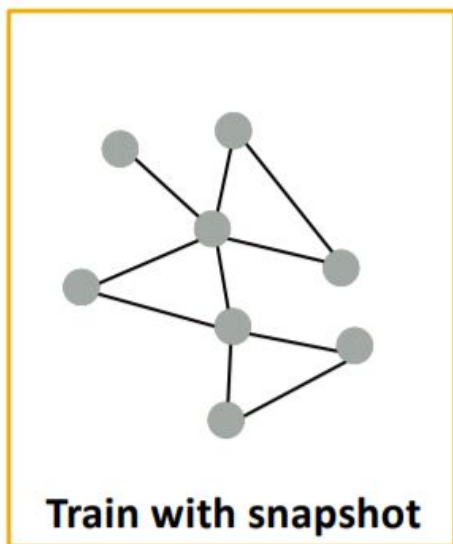
Transductive methods can only generate embeddings for nodes that were present during the training phase.

This restriction prevents these methods from being used on **inductive** applications, which involve generalizing after training to new nodes or brand new graphs.

Inductive capability for new nodes

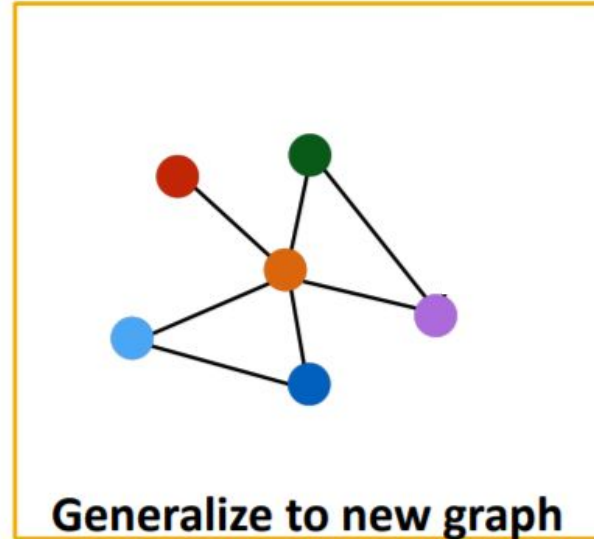
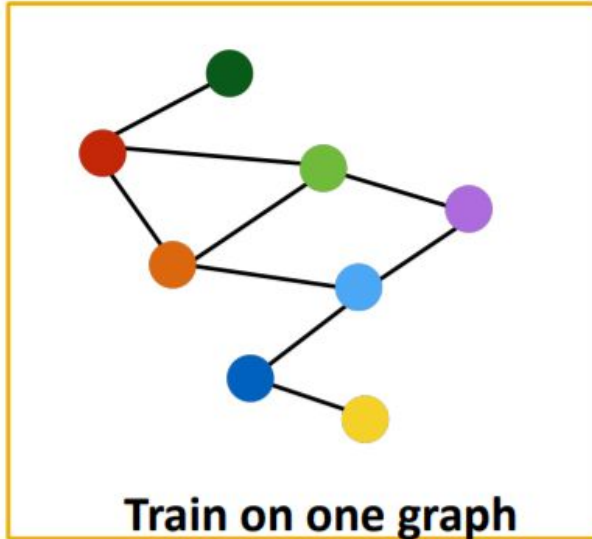
In many application contexts the graph is not static

- New nodes can be added and it is necessary to generate on the fly the embeddings for the new nodes without re-training the network



Inductive capability

-



Summary

From GCN to basic message passing model Generate node embeddings by aggregating neighborhood information

- We saw a basic variant of this idea
- Key distinctions are in how different approaches aggregate information across the layers

Next:

- Some limitations of the basic message passing approach
- The GraphSAGE (**S**Amples and aggre**G**ate**E**) model

Geometric deep learning

PART II

Basic message passing and layer stacking

Initialise(h_v^0) $\forall v \in V$

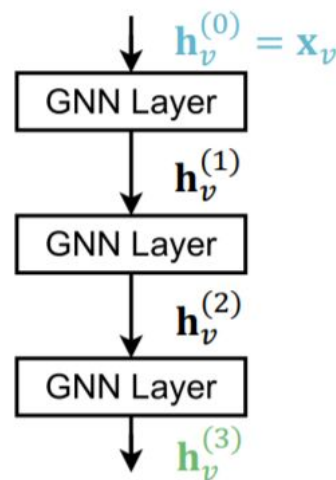
for $k = 1..K$ do

for $v \in V$ do

$$h_v^k = \text{AGGREGATE}_k(h_v^{k-1}, \{h_u^{k-1} \forall u \in N(v)\})$$

h_v^k will now be containing the embeddings

Learnable weights W

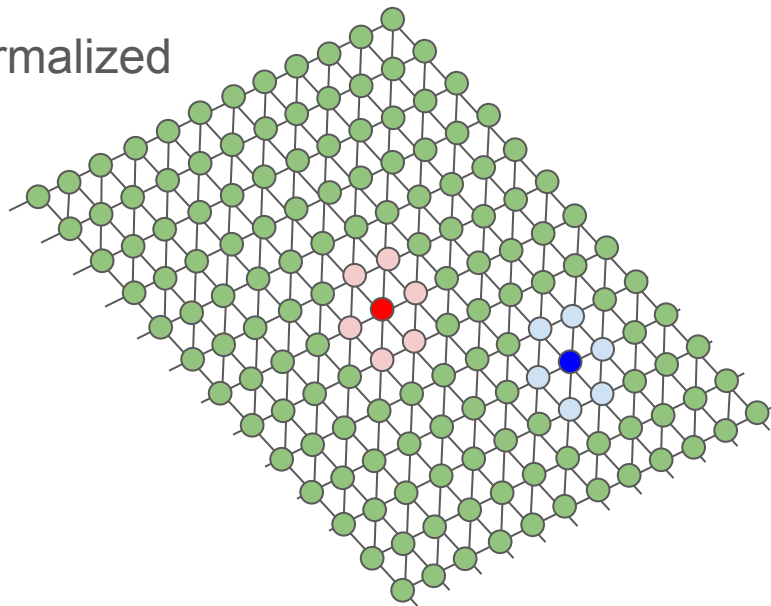


Problem: If the message from the current node (v) is summed with messages of its neighbors $N(v)$, information from the current node is not adequately preserved.

This way of updating nodes embeddings may yield to **oversmoothing**

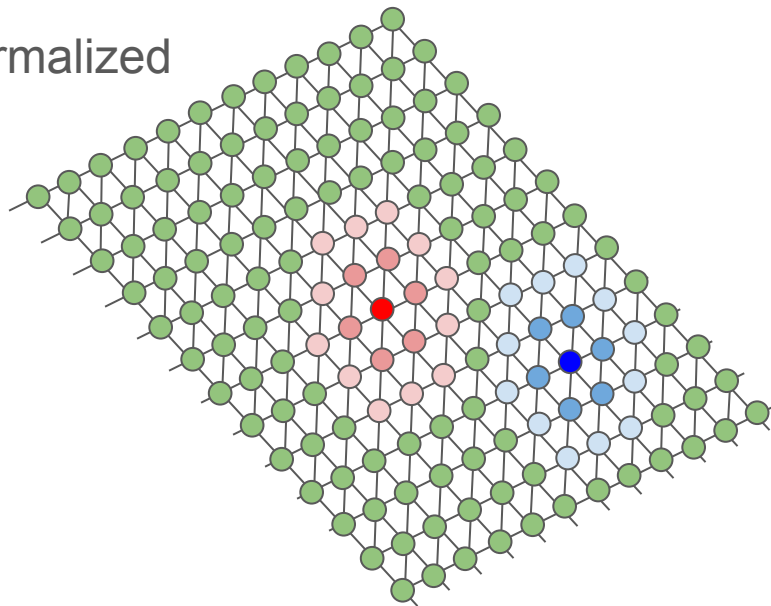
Oversmoothing

This issue of over-smoothing in GNNs can be formalized by defining the influence of each node's input feature $h_u^{(0)}$ on the final layer embedding of all the other nodes in the graph.



Oversmoothing and receptive field

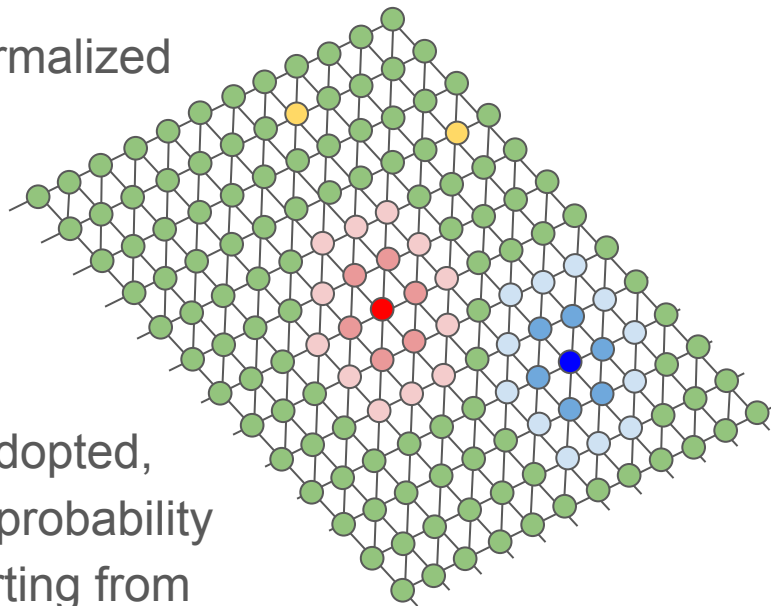
This issue of over-smoothing in GNNs can be formalized by defining the influence of each node's input feature $h_u^{(0)}$ on the final layer embedding of all the other nodes in the graph.



Oversmoothing and receptive field

This issue of over-smoothing in GNNs can be formalized by defining the influence of each node's input feature $h^{(0)}_u$ on the final layer embedding of all the other nodes in the graph.

Theorem: If a K-layer GNN-style base model is adopted, the influence of $h^{(0)}_u$ on $h^{(K)}_v$ is proportional the probability of reaching node v on a K-step random walk starting from node u .



Concatenation and skip-connections

Base update model $\text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)})$

$\mathbf{W}_{\text{self}}, \mathbf{W}_{\text{neigh}} \in \mathbb{R}^{d \times d}$

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

One of the simplest skip connection updates employs a concatenation to preserve more node-level information during message passing:

$$\text{UPDATE}_{\text{concat}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = [\text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) \oplus \mathbf{h}_u]$$

Concatenation and skip-connections

We can also employ other forms of skip-connections, such as a linear interpolation

$$\text{UPDATE}_{\text{interpolate}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \boldsymbol{\alpha}_1 \odot \text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) + \boldsymbol{\alpha}_2 \odot \mathbf{h}_u$$

where $\alpha_1, \alpha_2 \in [0, 1]^d$ are gating vectors with $\alpha_2 = 1 - \alpha_1$ and \odot denotes element-wise multiplication.

GraphSAGE

Inductive Representation Learning on Large Graphs

William L. Hamilton*
wleif@stanford.edu

Rex Ying*
rexying@stanford.edu

Jure Leskovec
jure@cs.stanford.edu

Department of Computer Science
Stanford University
Stanford, CA, 94305

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

GraphSAGE

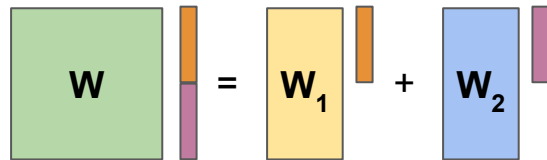
To avoid loss of information about each node, the embedding of the current node is **concatenated** with aggregation of embeddings of the neighboring nodes

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

Concatenation of two input vectors followed by a linear layer is identical to adding up two linear layer outputs for each input:

$$\mathbf{W} [x_1 \parallel x_2] = \mathbf{W}_1 x_1 + \mathbf{W}_2 x_2$$

If $x_1, x_2 \in \mathbb{R}^d$ and $\mathbf{W} \in \mathbb{R}^{2d \times 2d}$ then $\mathbf{W}_{\text{self}}, \mathbf{W}_{\text{neigh}} \in \mathbb{R}^{2d \times d}$.



GraphSAGE

Furthermore, several strategies are explored for the aggregation operator. In all cases these are permutation invariant

- Mean aggregator
- LSTM aggregator
- Pool aggregator

GraphSAGE - Mean aggregator

Elementwise mean of the vectors \mathbf{h}_u^{k-1} , $\forall u \in \{N(v) \cup v\}$.

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}))$$

Janossy pooling

Let π_i denote a permutation function that takes the unordered set of neighbor embeddings and places these embeddings in a sequence based on some arbitrary ordering.

The Janossy pooling approach performs neighborhood aggregation by

$$\mathbf{m}_{\mathcal{N}(u)} = \text{MLP}_{\theta} \left(\frac{1}{|\Pi|} \sum_{\pi_i \in \Pi} \rho_{\phi}(\mathbf{h}_{v_1}, \mathbf{h}_{v_2}, \dots, \mathbf{h}_{v_{|\mathcal{N}(u)|}})_{\pi_i} \right)$$

Permutation sensitive function that operates on sequences, e.g. an LSTM



GraphSAGE - LSTM aggregator

Use a Long Short Term Memory network to learn how to aggregate the neighbours.

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

GraphSAGE - Pool aggregator

Transform the embeddings of all nodes in the neighbourhood using a non linear operator (such as a Perceptron) and apply elementwise min or max pooling

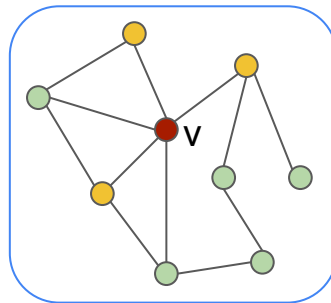
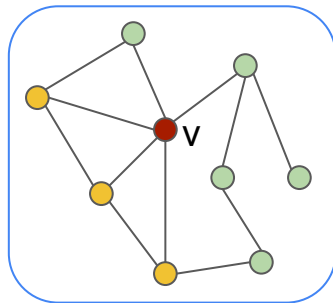
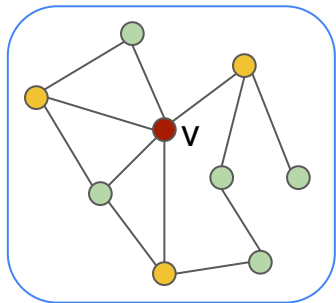
$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma(\mathbf{W}_{\text{pool}} \mathbf{h}_{u_i}^k + \mathbf{b}), \forall u_i \in \mathcal{N}(v)\})$$

GraphSAGE - Neighborhood

The neighborhood of node v , $N(v)$ is a fixed-size, uniform draw from the set

$$\{u \in V : (u, v) \in E\}$$

Different uniform samples are drawn at each iteration



GraphSAGE - Loss function

To address a fully unsupervised learning task, the graph-based loss function encourages nearby nodes to have similar representations, while enforcing that the representations of disparate nodes are highly distinct:


$$J_G(\mathbf{z}_u) = -\log(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n}))$$

v is a node that co-occurs near u

σ is the sigmoid function,

P_n is a negative sampling distribution,

Q defines the number of negative samples



This selects Q nodes v_n that **are not close** to node v

This unsupervised setting emulates situations where node features are provided to downstream machine learning applications

To normalize or not to normalize

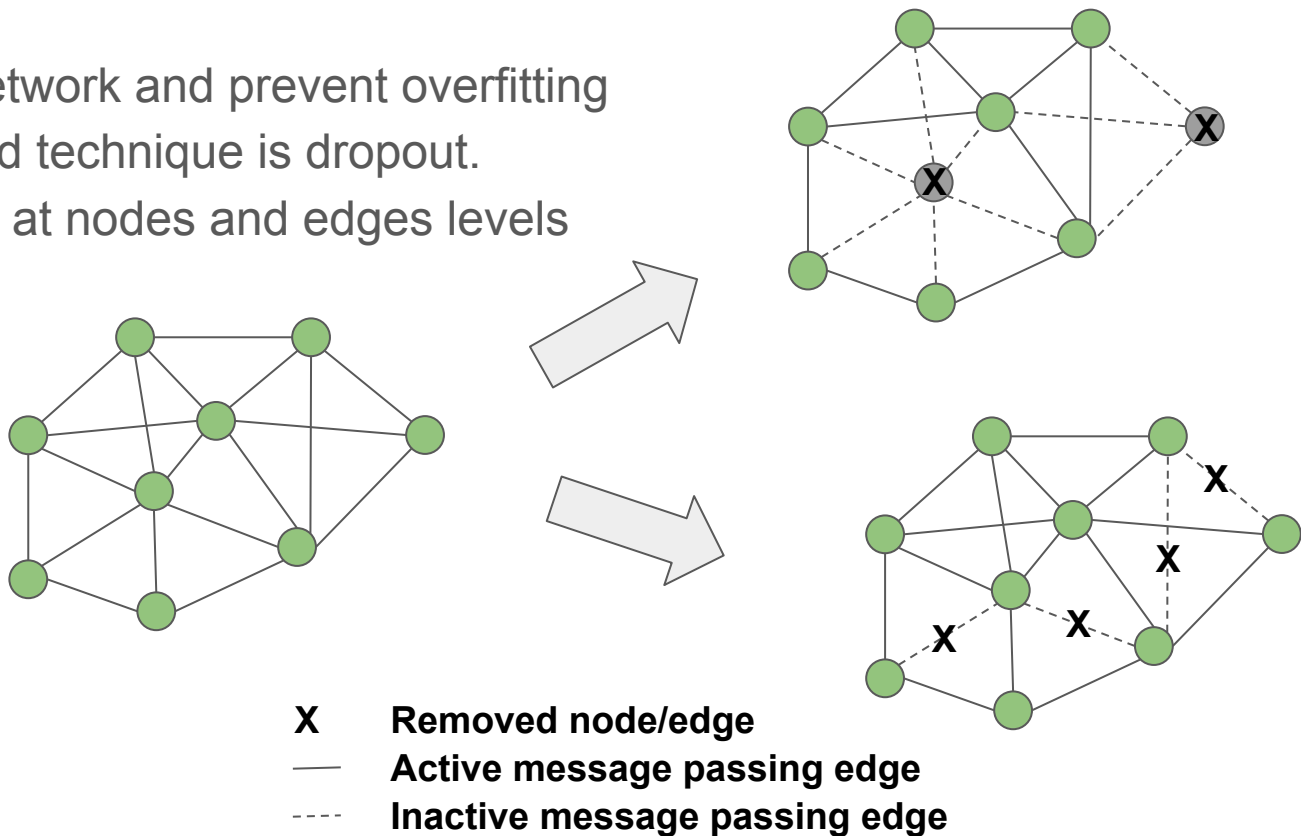
Proper normalization can be essential to achieve stable and strong performance when using a GNN. However, **normalization can also lead to a loss of information.**

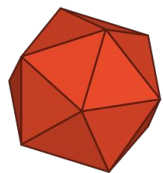
The use of normalization is thus an application-specific question. **Usually, normalization is most helpful in tasks where node feature information is far more useful than structural information**, or where there is a very wide range of node degrees that can lead to instabilities during optimization.

Dropout

To regularize the network and prevent overfitting a frequently adopted technique is dropout.

This can take place at nodes and edges levels





PyTorch geometric

<https://pytorch-geometric.readthedocs.io/>

PyTorch Geometric is a geometric deep learning extension library for PyTorch.

It consists of various methods for deep learning on graphs and other irregular structures also known as **geometric deep learning**, from a variety of published papers.

PACKAGE REFERENCE

`torch_geometric`

▢ `torch_geometric.nn`

Convolutional Layers

Dense Convolutional Layers

Normalization Layers

Global Pooling Layers

Pooling Layers

Dense Pooling Layers

Unpooling Layers

Models

Functional

DataParallel Layers

Convolutional Layers

`MessagePassing`

Base class for creating message passing layers of the form

`GCNConv`

The graph convolutional operator from the "Semi-supervised Classification with Graph Convolutional Networks" paper

`ChebConv`

The chebyshev spectral graph convolutional operator from the "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering" paper

`SAGEConv`

The GraphSAGE operator from the "Inductive Representation Learning on Large Graphs" paper

`GraphConv`

The graph neural network operator from the "Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks" paper

`GravNetConv`

The GravNet operator from the "Learning Representations of Irregular Particle-detector Geometry with Distance-weighted Graph Networks" paper, where the graph is dynamically constructed using nearest neighbors.

`GatedGraphConv`

The gated graph convolution operator from the "Gated Graph Sequence Neural Networks" paper

`ResGatedGraphConv`

The residual gated graph convolutional operator from the "Residual Gated Graph ConvNets" paper

`GATConv`

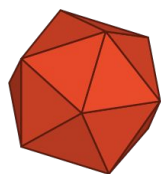
The graph attentional operator from the "Graph Attention Networks" paper

`GATv2Conv`

The GATv2 operator from the "How Attentive are Graph Attention Networks?" paper, which fixes the static attention problem of the standard `GATConv` layer: since the linear layers in the standard GAT are applied right after each other, the ranking of attended nodes is unconditioned on the query node.

`TransformerConv`

The graph transformer operator from the "Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification" paper



PyTorch geometric

It consists of an easy-to-use mini-batch loader for many small and single giant graphs, a large number of common benchmark datasets (based on simple interfaces to create your own), and helpful transforms, both for learning on arbitrary graphs as well as on 3D meshes or point clouds.

PACKAGE REFERENCE

`torch_geometric`

`torch_geometric.nn`

`torch_geometric.data`

`torch_geometric.datasets`

`torch_geometric.transforms`

`torch_geometric.utils`

TORCH_GEOMETRIC.DATASETS

`KarateClub`

Zachary's karate club network from the "[An Information Flow Model for Conflict and Fission in Small Groups](#)" paper, containing 34 nodes, connected by 156 (undirected and unweighted) edges.

`TUDataset`

A variety of graph kernel benchmark datasets, .e.g. "IMDB-BINARY", "REDDIT-BINARY" or "PROTEINS", collected from the [TU Dortmund University](#).

`GNNBenchmarkDataset`

A variety of artificially and semi-artificially generated graph datasets from the "[Benchmarking Graph Neural Networks](#)" paper.

`Planetoid`

The citation network datasets "Cora", "CiteSeer" and "PubMed" from the "[Revisiting Semi-Supervised Learning with Graph Embeddings](#)" paper.

`NELL`

The NELL dataset, a knowledge graph from the "[Toward an Architecture for Never-Ending Language Learning](#)" paper.

`CitationFull`

The full citation network datasets from the "[Deep Gaussian Embedding of Graphs: Unsupervised Inductive Learning via Ranking](#)" paper.

`CoraFull`

Alias for `torch_geometric.dataset.CitationFull` with `name="cora"`.

`Coauthor`

The Coauthor CS and Coauthor Physics networks from the "[Pitfalls of Graph Neural Network Evaluation](#)" paper.

`Amazon`

The Amazon Computers and Amazon Photo networks from the "[Pitfalls of Graph Neural Network Evaluation](#)" paper.

`PPI`

The protein-protein interaction networks from the "[Predicting Multicellular Function through Multi-layer Tissue Networks](#)" paper, containing positional gene sets, motif gene sets and immunological signatures as features (50 in total) and gene ontology sets as labels (121 in total).

GraphSAGE - Implementation

Pytorch geometric implementation of the GraphSAGE model layer

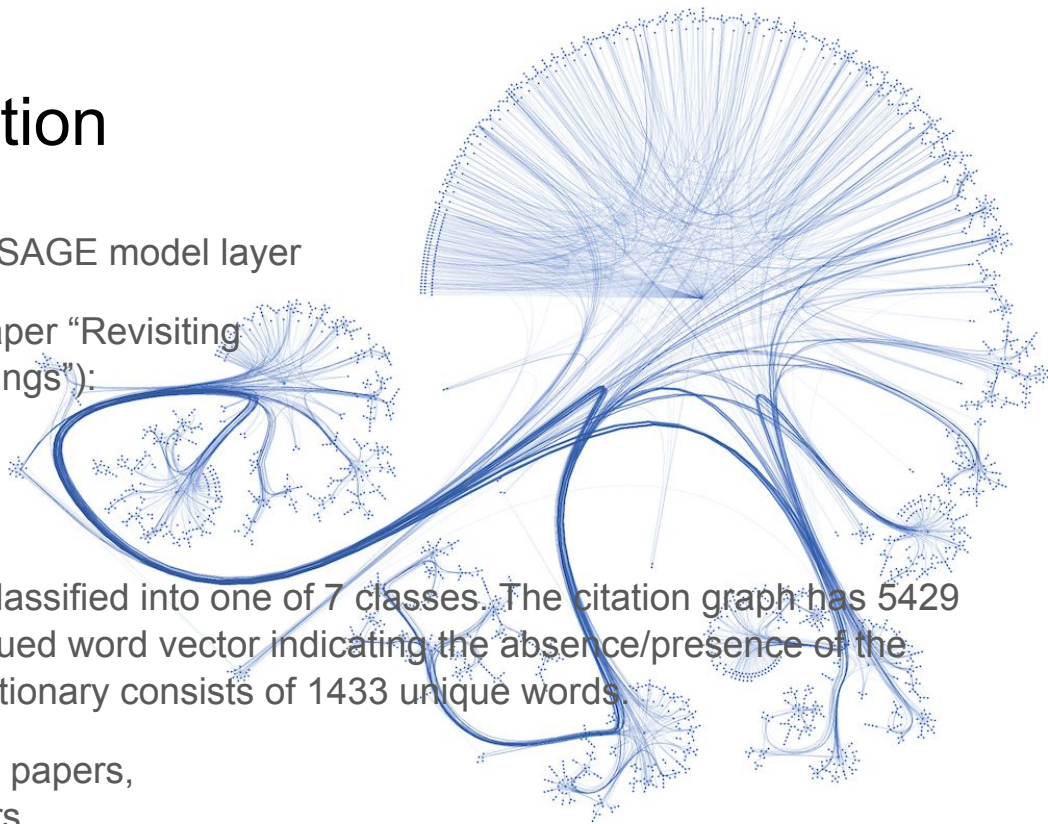
Planetoid dataset of Pytorch geometric (from paper “Revisiting Semi-Supervised Learning with Graph Embeddings”):

- “Cora”,
- “CiteSeer”
- “PubMed”

The Cora dataset: 2708 scientific publications classified into one of 7 classes. The citation graph has 5429 links. Each publication is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary. The dictionary consists of 1433 unique words.

Task: The true class value is known for just 140 papers,
predict the class value for 1000 test papers

For each node, the **input** feature is the 1433-dimensional word vector, the **output** feature is a 7-dimensional vector whose entries are the probabilities for each class



GraphSAGE - Implementation

```
class SAGEConv ( in_channels: Union[int, Tuple[int, int]], out_channels: int, aggr: str = 'mean',  
normalize: bool = False, root_weight: bool = True, project: bool = False, bias: bool = True, **kwargs )  
[source]
```

The GraphSAGE operator from the “Inductive Representation Learning on Large Graphs” paper

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \text{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j$$

If `project = True`, then \mathbf{x}_j will first get projected via

$$\mathbf{x}_j \leftarrow \sigma(\mathbf{W}_3 \mathbf{x}_j + \mathbf{b})$$

as described in Eq. (3) of the paper.

PARAMETERS:

- `in_channels` (*int* or *tuple*) – Size of each input sample, or `-1` to derive the size from the first input(s) to the forward method. A tuple corresponds to the sizes of source and target dimensionalities.
- `out_channels` (*int*) – Size of each output sample.
- `aggr` (*string, optional*) – The aggregation scheme to use (`"mean"`, `"max"`, `"lstm"`). (default: `"mean"`)
- `normalize` (*bool, optional*) – If set to `True`, output features will be ℓ_2 -normalized, i.e., $\frac{\mathbf{x}'_i}{\|\mathbf{x}'_i\|_2}$. (default: `False`)



GraphSAGE - Implementation



+ Code + Text



{x}



```
[ ] import os
import torch
os.environ['TORCH'] = torch.__version__
print(torch.__version__)

!pip install -q torch-scatter -f https://data.pyg.org/whl/torch-${TORCH}.html
!pip install -q torch-sparse -f https://data.pyg.org/whl/torch-${TORCH}.html
!pip install -q git+https://github.com/pyg-team/pytorch_geometric.git
```

```
[ ]
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.data import Data
from torch_geometric.nn import GATConv, GATv2Conv, SAGEConv
from torch_geometric.datasets import Planetoid
import torch_geometric.transforms as T

import matplotlib.pyplot as plt
```

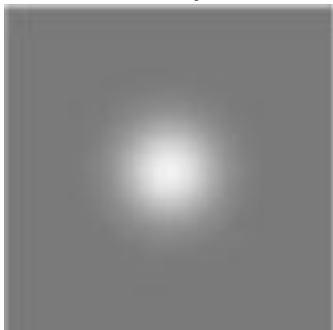
```
[ ] name_data = 'Cora' #'PubMed' #'CiteSeer' #'Cora'
dataset = Planetoid(root= '/tmp/' + name_data, name = name_data)
dataset.transform = T.NormalizeFeatures()
```

Isotropy vs anisotropy

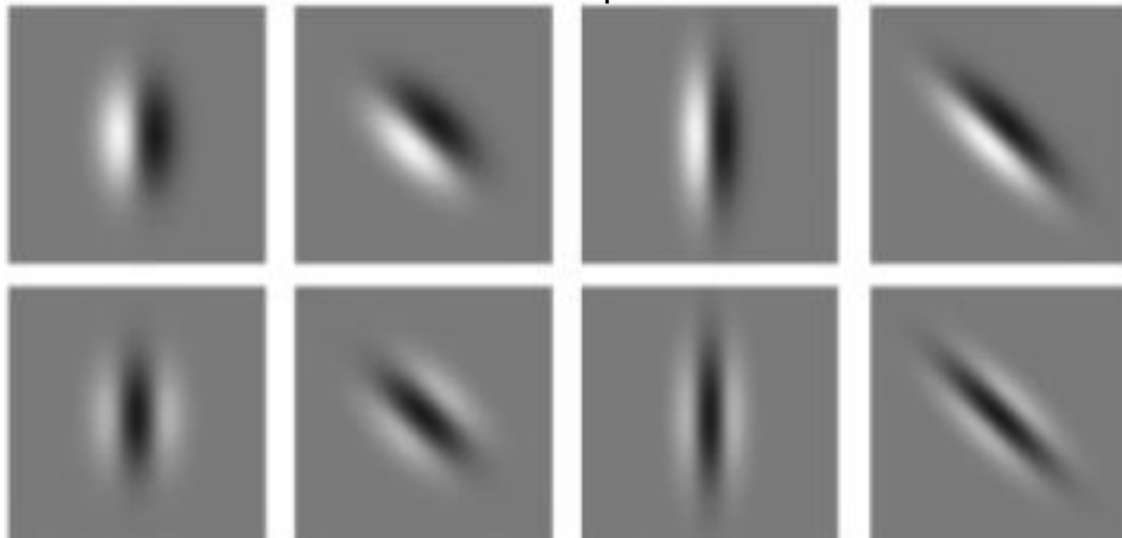
$$h_v^l = \sigma \left(\sum_{u \in N(v)} \frac{1}{|N(v)|} W^l h_u^{l-1} \right)$$

?

Isotropic



Anisotropic



GRAPH ATTENTION NETWORKS

Petar Veličković*

Department of Computer Science and Technology
University of Cambridge
petar.velickovic@cst.cam.ac.uk

Guillem Cucurull*

Centre de Visió per Computador, UAB
gcucurull@gmail.com

Arantxa Casanova*

Centre de Visió per Computador, UAB
ar.casanova.8@gmail.com

Adriana Romero

Montréal Institute for Learning Algorithms
adriana.romero.soriano@umontreal.ca

Pietro Liò

Department of Computer Science and Technology
University of Cambridge
pietro.lio@cst.cam.ac.uk

Yoshua Bengio

Montréal Institute for Learning Algorithms
yoshua.umontreal@gmail.com

Neighbor attention

In GCN/GraphSAGE messages from neighboring nodes are equally important

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

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \text{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j$$

Neighbor attention

In GCN/GraphSAGE messages from neighboring nodes are equally important:

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

However, we could assign a different relevance α_{vu} to each neighboring node

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

Neighbor attention

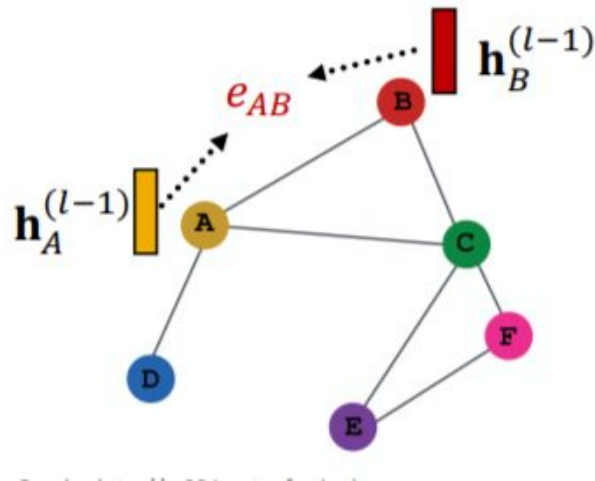
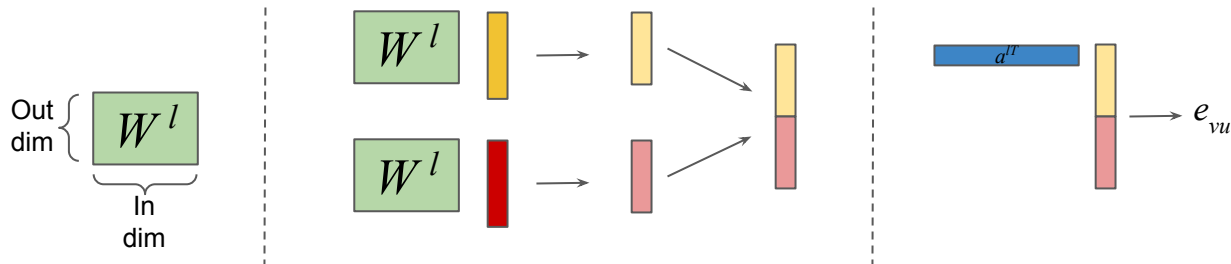
Weights α_{vu} are computed based on **attention coefficients** e_{vu} across pairs of neighboring nodes

$$e_{vu} = (a^l)^T (W^l h_v^{l-1} || W^l h_u^{l-1})$$

Trainable weight matrix

Trainable attention vector

This represents the relevance of u's message to node v



Neighbor attention

Weights α_{vu} are computed based on **attention coefficients** e_{vu} across pairs of neighboring nodes

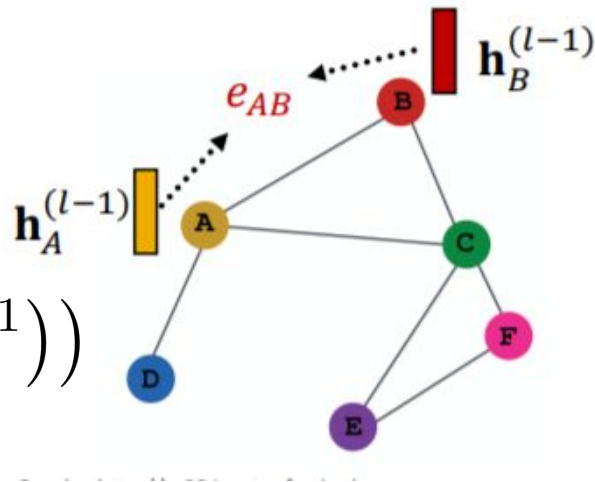
$$e_{vu} = (a^l)^T (W^l h_v^{l-1} || W^l h_u^{l-1})$$

Trainable weight matrix

Trainable attention vector

This represents the relevance of u's message to node v

$$e_{vu} = \text{LeakyReLU} \left((a^l)^T (W^l h_v^{l-1} || W^l h_u^{l-1}) \right)$$



Neighbor attention

To make the attention coefficients easily comparable across different nodes, they are normalized using the softmax function

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

...and used to update the embedding of node v

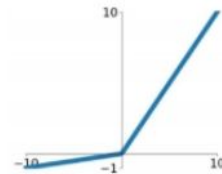
$$h_v^l = \sigma \left(\sum_{u \in N(v)} \alpha_{vu} W^l h_u^{l-1} \right)$$

Neighbor attention

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

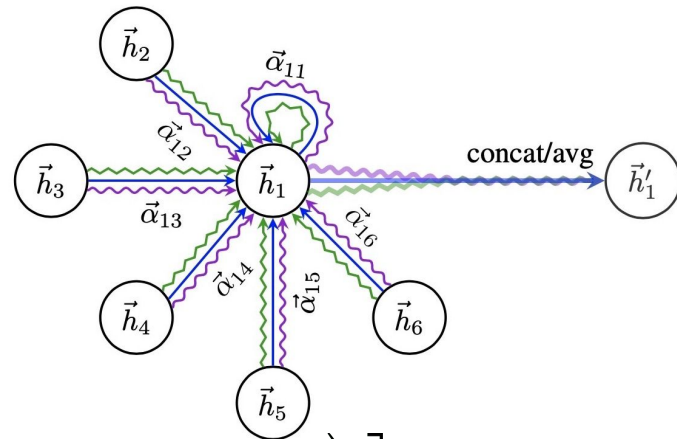
$$h_v^l = \sigma \left(\sum_{u \in N(v)} \alpha_{vu} W^l h_u^{l-1} \right)$$

Leaky ReLU
 $\max(0.2x, x)$



Multi-head attention

To stabilize the learning process of self-attention it's useful to employ multi-head attention. This is obtained by combining K independent attention mechanisms



$$h_v^l = \left[\sigma \left(\sum_{u \in N(v)} \alpha_{vu}^1 W_1^l h_u^{l-1} \right) \parallel \dots \parallel \sigma \left(\sum_{u \in N(v)} \alpha_{vu}^K W_K^l h_u^{l-1} \right) \right]$$

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

Advantages of the graph attentional layer

- Efficient computation
- Different importances improves interpretability,
- The local attention mechanism does not depend on the global graph structure
 - The graph can be directed (we may simply leave out computing α_{ij} if edge $j \rightarrow i$ is not present).
 - The technique is directly applicable to inductive learning

HOW ATTENTIVE ARE GRAPH ATTENTION NETWORKS?

Shaked Brody

Technion

shakedbr@cs.technion.ac.il

Uri Alon

Language Technologies Institute

Carnegie Mellon University

ualon@cs.cmu.edu

Eran Yahav

Technion

yahave@cs.technion.ac.il

GAT-v2

GAT-v2

In the original GAT architecture A scoring function $e : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ computes a score for every edge (j, i) to represent the relevance of the neighbor j to the node i :

$$e(\mathbf{h}_i, \mathbf{h}_j) = \text{LeakyReLU}(\mathbf{a}^\top \cdot [\mathbf{W}\mathbf{h}_i \parallel \mathbf{W}\mathbf{h}_j])$$

where $\mathbf{a} \in \mathbb{R}^{2d'}$, $\mathbf{W} \in \mathbb{R}^{d' \times d}$ are learned, and \parallel denotes vector concatenation.

These attention scores are normalized across all neighbors $j \in \mathcal{N}_i$ using softmax

$$\alpha_{ij} = \text{softmax}_j(e(\mathbf{h}_i, \mathbf{h}_j)) = \frac{\exp(e(\mathbf{h}_i, \mathbf{h}_j))}{\sum_{j' \in \mathcal{N}_i} \exp(e(\mathbf{h}_i, \mathbf{h}_{j'}))}$$

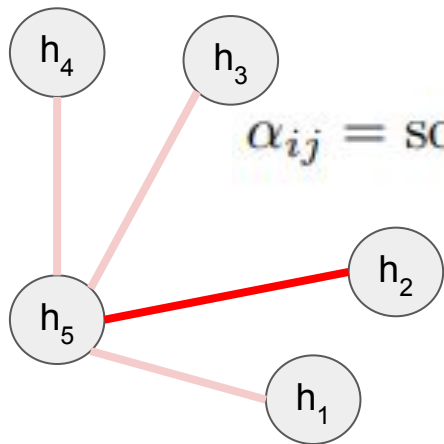
$$\mathbf{h}'_i = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \cdot \mathbf{W}\mathbf{h}_j \right)$$

GAT-v2

$$e(\mathbf{h}_i, \mathbf{h}_j) = \text{LeakyReLU}(\mathbf{a}^\top \cdot [\mathbf{W}\mathbf{h}_i \parallel \mathbf{W}\mathbf{h}_j])$$

$$e(\mathbf{h}_i, \mathbf{h}_j) = \text{LeakyReLU}(\mathbf{a}_1^\top \mathbf{W}\mathbf{h}_i + \mathbf{a}_2^\top \mathbf{W}\mathbf{h}_j) \quad \mathbf{a} = [\mathbf{a}_1 \parallel \mathbf{a}_2] \in \mathbb{R}^{2d'}$$

$\mathbf{a}_2^\top \mathbf{W}\mathbf{h}_{j_{max}}$



$$\alpha_{ij} = \text{softmax}_j(e(\mathbf{h}_i, \mathbf{h}_j)) = \frac{\exp(e(\mathbf{h}_i, \mathbf{h}_j))}{\sum_{j' \in \mathcal{N}_i} \exp(e(\mathbf{h}_i, \mathbf{h}_{j'}))}$$

GAT-v2

$$\text{GAT} \left\{ \begin{array}{l} e(\mathbf{h}_i, \mathbf{h}_j) = \text{LeakyReLU}(\mathbf{a}^\top \cdot [\mathbf{W}\mathbf{h}_i \parallel \mathbf{W}\mathbf{h}_j]) \\ e(\mathbf{h}_i, \mathbf{h}_j) = \text{LeakyReLU}(\mathbf{a}_1^\top \mathbf{W}\mathbf{h}_i + \mathbf{a}_2^\top \mathbf{W}\mathbf{h}_j) \end{array} \right.$$

$$\mathbf{a}_2^\top \mathbf{W}\mathbf{h}_{j_{max}}$$

$$\text{GAT-v2} \left\{ \begin{array}{l} e(\mathbf{h}_i, \mathbf{h}_j) = \mathbf{a}^\top \text{LeakyReLU}(\mathbf{W} \cdot [\mathbf{h}_i \parallel \mathbf{h}_j]) \end{array} \right.$$

$$\alpha_{ij} = \text{softmax}_j(e(\mathbf{h}_i, \mathbf{h}_j)) = \frac{\exp(e(\mathbf{h}_i, \mathbf{h}_j))}{\sum_{j' \in \mathcal{N}_i} \exp(e(\mathbf{h}_i, \mathbf{h}_{j'}))}$$

GAT and GAT-v2 Implementation

Pytorch geometric implementation of the GATConv model layer

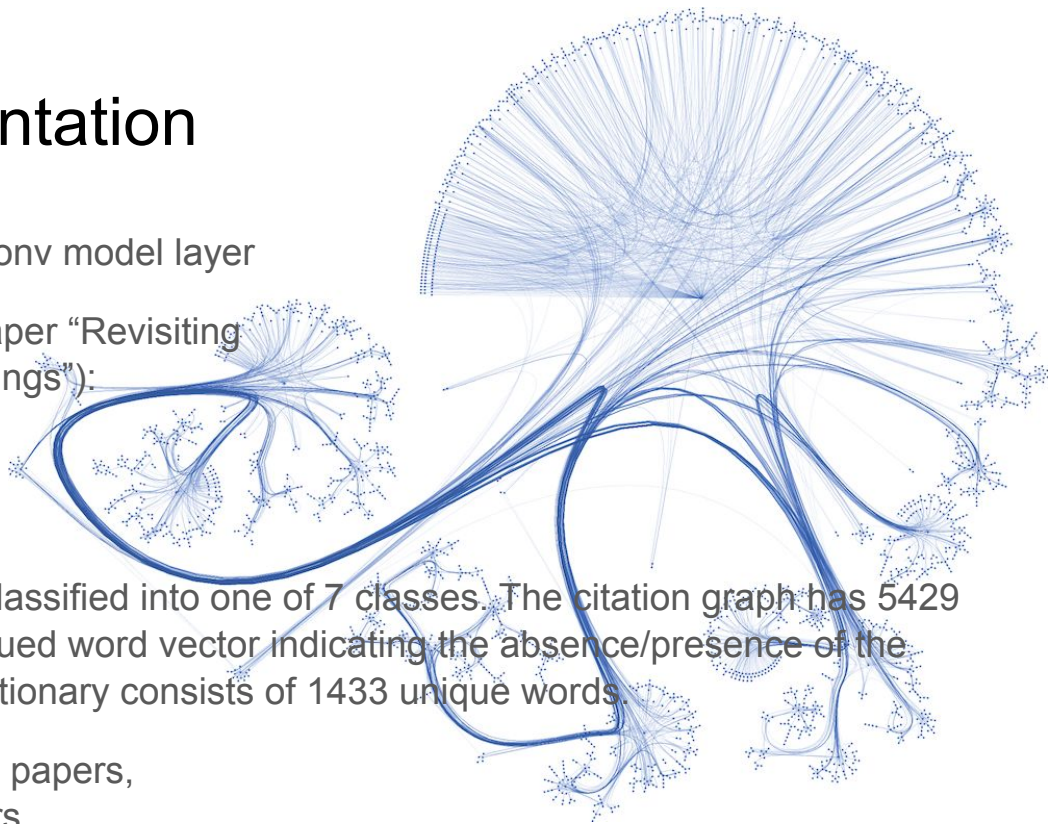
Planetoid dataset of Pytorch geometric (from paper “Revisiting Semi-Supervised Learning with Graph Embeddings”):

- “Cora”,
- “CiteSeer”
- “PubMed”

The Cora dataset: 2708 scientific publications classified into one of 7 classes. The citation graph has 5429 links. Each publication is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary. The dictionary consists of 1433 unique words.

Task: The true class value is known for just 140 papers,
predict the class value for 1000 test papers

For each node, the **input** feature is the 1433-dimensional word vector, the **output** feature is a 7-dimensional vector whose entries are the probabilities for each class



```

class GATConv ( in_channels: Union[int, Tuple[int, int]], out_channels: int, heads: int = 1, concat: bool =
True, negative_slope: float = 0.2, dropout: float = 0.0, add_self_loops: bool = True, bias: bool = True,
**kwargs ) [source]

```

GAT - Implementation

The graph attentional operator from the "Graph Attention Networks" paper

$$\mathbf{x}'_i = \alpha_{i,i} \Theta \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} \Theta \mathbf{x}_j,$$

where the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^\top [\Theta \mathbf{x}_i \parallel \Theta \mathbf{x}_j]))}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp(\text{LeakyReLU}(\mathbf{a}^\top [\Theta \mathbf{x}_i \parallel \Theta \mathbf{x}_k]))}.$$

PARAMETERS:

- **in_channels** (*int or tuple*) – Size of each input sample, or `-1` to derive the size from the first input(s) to the forward method. A tuple corresponds to the sizes of source and target dimensionalities.
- **out_channels** (*int*) – Size of each output sample.
- **heads** (*int, optional*) – Number of multi-head-attentions. (default: `1`)
- **concat** (*bool, optional*) – If set to `False`, the multi-head attentions are averaged instead of concatenated. (default: `True`)
- **negative_slope** (*float, optional*) – LeakyReLU angle of the negative slope. (default: `0.2`)
- **dropout** (*float, optional*) – Dropout probability of the normalized attention coefficients which exposes each node to a stochastically sampled neighborhood during training. (default: `0`)
- **add_self_loops** (*bool, optional*) – If set to `False`, will not add self-loops to the input graph. (default: `True`)
- **bias** (*bool, optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)



GAT and GAT-v2 - Implementation

+ Code + Text

```
[ ] import os
import torch
os.environ['TORCH'] = torch.__version__
print(torch.__version__)

!pip install -q torch-scatter -f https://data.pyg.org/whl/torch-\${TORCH}.html
!pip install -q torch-sparse -f https://data.pyg.org/whl/torch-\${TORCH}.html
!pip install -q git+https://github.com/pyg-team/pytorch\_geometric.git
```

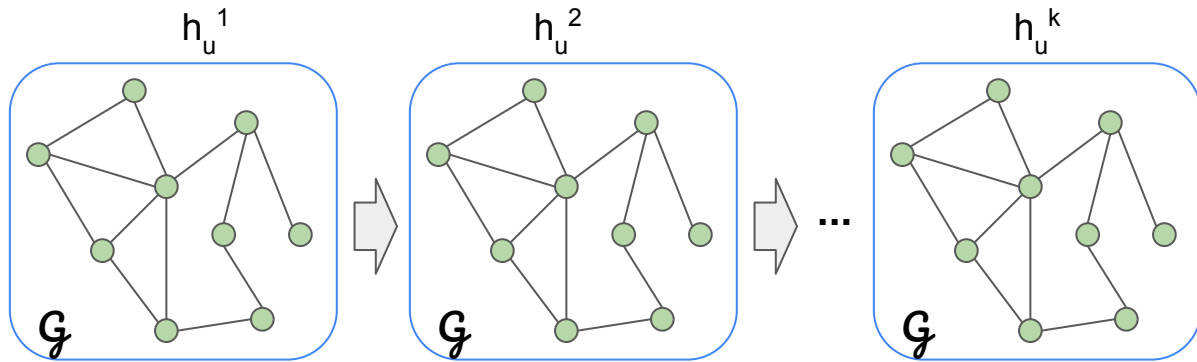
```
[ ]
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.data import Data
from torch_geometric.nn import GATConv, GATv2Conv, SAGEConv
from torch_geometric.datasets import Planetoid
import torch_geometric.transforms as T

import matplotlib.pyplot as plt
```

```
[ ] name_data = 'Cora' #'PubMed' #'CiteSeer' #'Cora'
dataset = Planetoid(root= '/tmp/' + name_data, name = name_data)
dataset.transform = T.NormalizeFeatures()

print(f"Number of Classes in {name_data}:", dataset.num_classes)
print(f"Number of Node Features in {name_data}:", dataset.num_node_features)
```

JK Connections



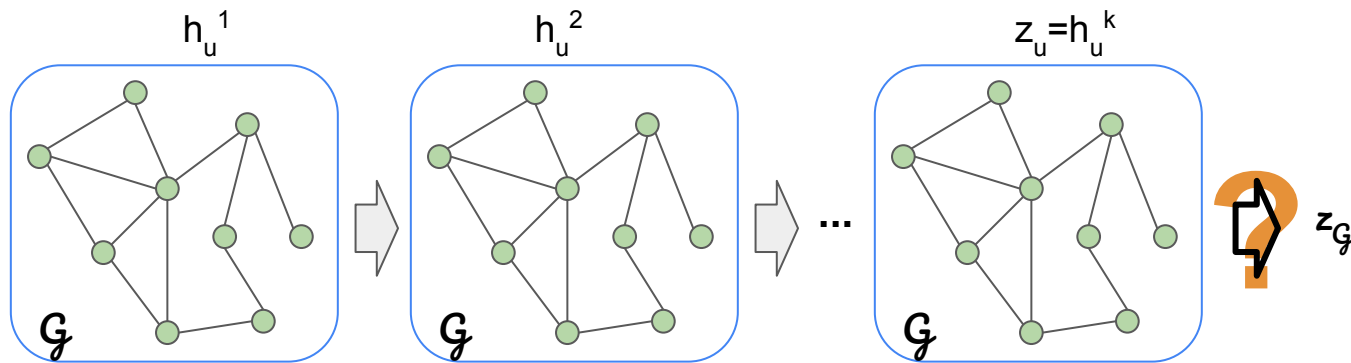
We have assumed that the GNN output corresponds to the **final layer** of the network

However, alternative solutions are possible, such as collecting node representations at each layer

$$z_u = f_{JK}(h_u^{(1)} \oplus h_u^{(2)} \oplus \dots \oplus h_u^{(K)})$$

This strategy is known as adding **jumping knowledge (JK) connections**.

Graph pooling



The neural message passing approach produces a set of node embeddings.

What if we want to make predictions at the graph level?

This task is often referred to as **graph pooling**, since our goal is to pool together the node embeddings in order to learn an embedding of the entire graph.

- **Global pooling**: aggregate all node embeddings
- **Hierarchical pooling**: build a hierarchical representation based on graph coarsening

Global graph pooling

We want to design a pooling function f_p , which maps a set of node embeddings $\{z_1, \dots, z_{|\mathcal{V}|}\}$ to an embedding z_G that represents the full graph

Take a sum (or mean) of the node embeddings

$$\mathbf{z}_G = \frac{\sum_{v \in \mathcal{V}} \mathbf{z}_v}{f_n(|\mathcal{V}|)}$$

f_n being some normalizing function (e.g., the identity function).

Global graph pooling

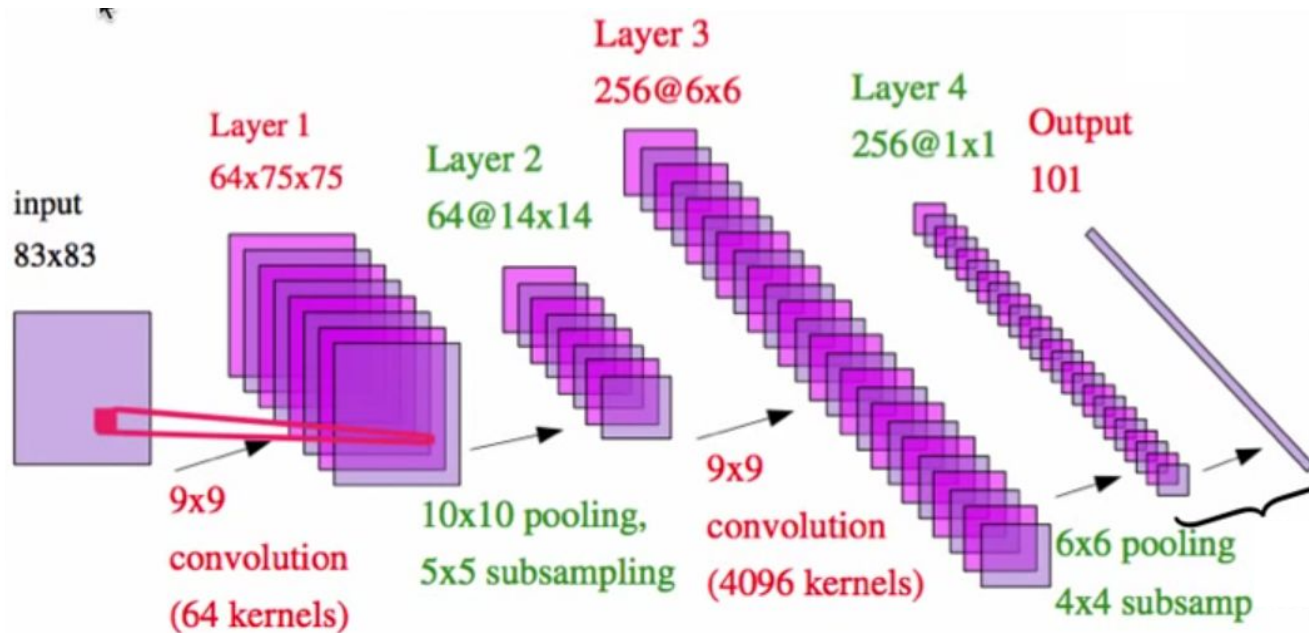
We want to design a pooling function f_p , which maps a set of node embeddings $\{z_1, \dots, z_{|V|}\}$ to an embedding z_G that represents the full graph

Use a combination of LSTMs and attention to pool the node embeddings by iterating over T steps

$$\begin{aligned}\mathbf{q}_t &= \text{LSTM}(\mathbf{o}_{t-1}, \mathbf{q}_{t-1}), & t=1, \dots, T \\ e_{v,t} &= f_a(\mathbf{z}_v, \mathbf{q}_t), \forall v \in \mathcal{V}, \\ a_{v,t} &= \frac{\exp(e_{v,t})}{\sum_{u \in \mathcal{V}} \exp(e_{u,t})}, \forall v \in \mathcal{V}, \\ \mathbf{o}_t &= \sum_{v \in \mathcal{V}} a_{v,t} \mathbf{z}_v. \\ z_G &= [o_1 || \dots || o_T]\end{aligned}$$

Graph coarsening

Graph coarsening



Graph coarsening

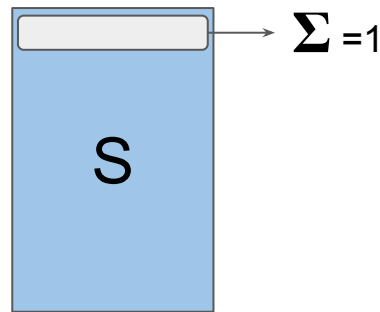
In these style of approaches, we assume that we have some clustering function

$$f_c \rightarrow G \times \mathbb{R}^{|V| \times d} \rightarrow \mathbb{R}^{|V| \times c}$$

which maps all the nodes in the graph to an assignment over c clusters.

This function outputs an assignment matrix $S = f_c(G, Z)$

$S[u, i] \in \mathbb{R}^+$ confidence node u belongs to cluster i .



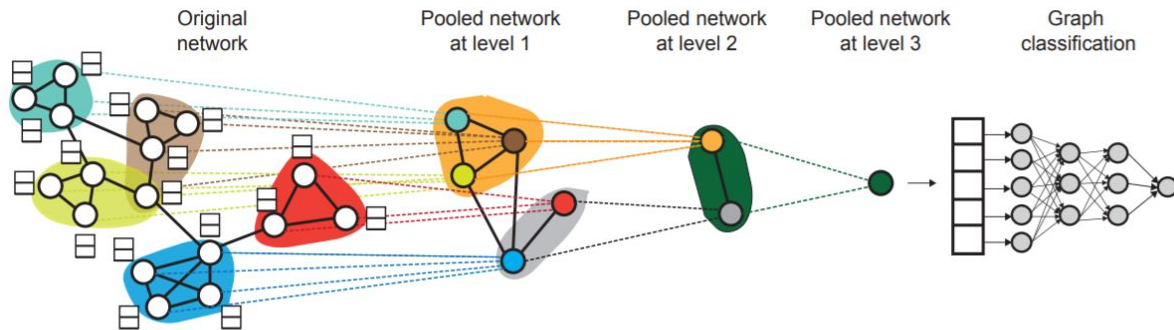
Graph coarsening

The assignment matrix $S \in \mathbb{R}^{|V| \times c}$ is used to compute a new coarsened adjacency matrix

$$A_{\text{new}} = S^T A S \in \mathbb{R}^{c \times c}$$

and a new set of node features

$$X_{\text{new}} = S^T X \in \mathbb{R}^{c \times d}$$



Graph coarsening

A =

0	1	0	1	0
1	0	0	0	1
0	0	0	1	0
1	0	1	0	0
0	1	0	0	0

X =

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

S =

0.9	0.1
0.7	0.3
0.1	0.9
0.1	0.9
0.8	0.2

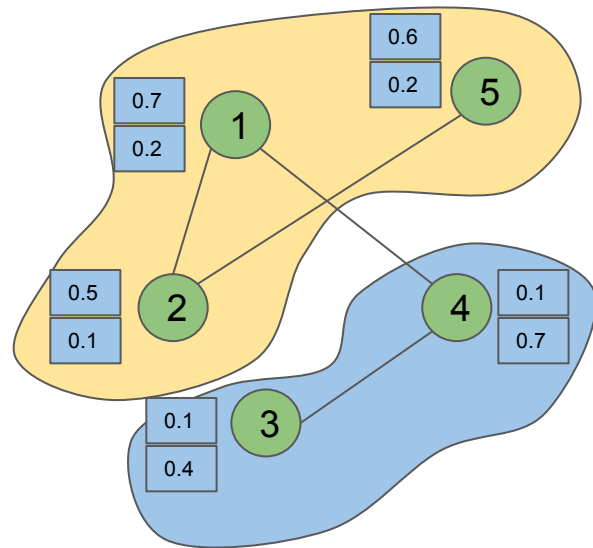
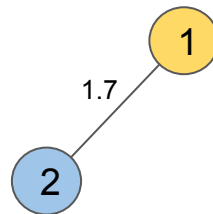
$$A_{\text{new}} = S^T A S \in R^{c \times c}$$

=

$$X_{\text{new}} = S^T X \in R^{c \times d} =$$

2.6	1.7
1.7	2.0

1.5	0.5
0.5	1.1



Graph coarsening

A =

0	1	0	1	0
1	0	0	0	1
0	0	0	1	0
1	0	1	0	0
0	1	0	0	0

X =

0.7	0.2
0.5	0.1
0.1	0.4
0.1	0.7
0.6	0.2

S =

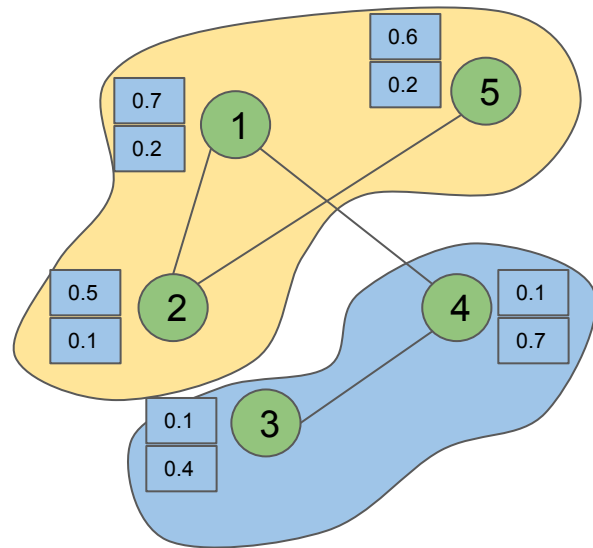
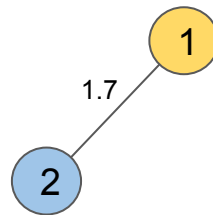
0.9	0.1
0.7	0.3
0.1	0.9
0.1	0.9
0.8	0.2

$$A_{\text{new}} = S^T A S \in \mathbb{R}^{c \times c}$$

$$X_{\text{new}} = S^T X \in \mathbb{R}^{c \times d}$$

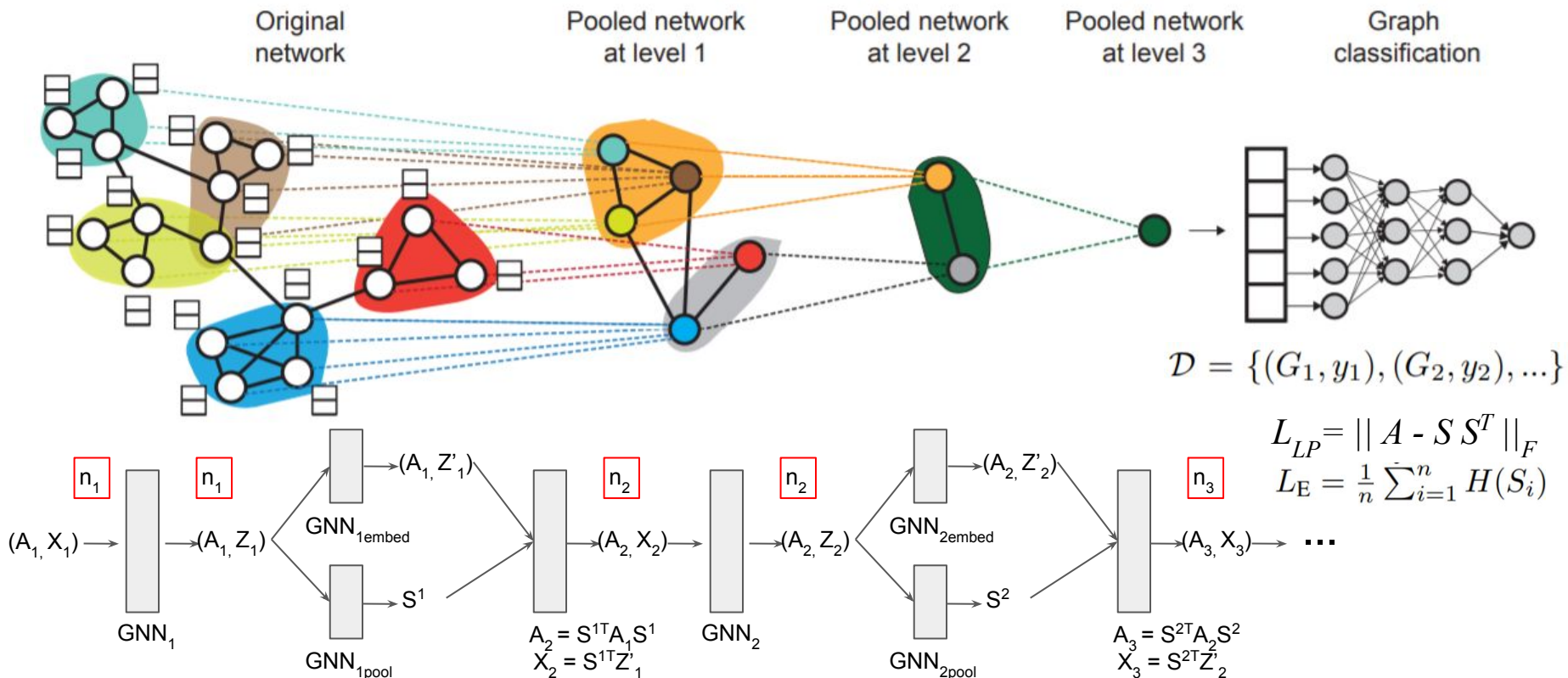
2.6	1.7
1.7	2.0

1.5	0.5
0.5	1.1



What about training a model to learn S and be capable to generalize to new graphs

DIFFPOOL



More on graph coarsening

Published as a conference paper at ICLR 2021

GRAPH COARSENING WITH NEURAL NETWORKS

Chen Cai *

Dingkang Wang [†]

Yusu Wang [‡]

[arXiv:2102.01350](https://arxiv.org/abs/2102.01350)

GNN loss functions

GNNs learning tasks:

- node classification (predicting whether a user is a bot in a social network)
- graph classification (property prediction based on molecular graph structures)
- relation prediction (content recommendation in online platforms).

How do these tasks translate into loss functions?

- $z_u \in \mathbb{R}^d$ node embedding (last layer)
- $z_G \in \mathbb{R}^d$ graph-level embedding (pooling)

GNN loss for node classification

For node classification tasks define the loss using a softmax classification function and negative log-likelihood loss:

$$\mathcal{L} = \sum_{u \in \mathcal{V}_{\text{train}}} -\log(\text{softmax}(\mathbf{z}_u, \mathbf{y}_u))$$

$\mathbf{y}_u \in \mathbb{Z}^c$ is a one-hot vector indicating the class of training node $u \in \mathcal{V}_{\text{train}}$;

$$\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) = \sum_{i=1}^c \mathbf{y}_u[i] \frac{e^{\mathbf{z}_u^\top \mathbf{w}_i}}{\sum_{j=1}^c e^{\mathbf{z}_u^\top \mathbf{w}_j}}$$

$\mathbf{w}_i \in \mathbb{R}^d$, $i = 1, \dots, c$ are trainable parameters

Node classification

Node classification: supervised or semi-supervised?

Three types of nodes can be distinguished:

1. **Training** nodes, V_{train} . These nodes are included in the GNN message passing operations, and they are also used to compute the loss
2. **Transductive test** nodes, V_{trans} . During training participate to message passing but are disregarded for computing the loss
3. **Inductive test** nodes, V_{ind} . Not present in the graph during training

GNN loss for graph classification

Softmax classification loss computed with graph-level embeddings \mathbf{z}_{G_i} over a set of labeled training graphs $T = \{G_1, \dots, G_n\}$

$$\mathcal{L} = \sum_{G_i \in T} -\log \left(\sum_{k=1}^C y_{G_i}[k] \frac{\exp(\mathbf{z}_{G_i}^T \mathbf{W}_k)}{\sum_{j=1}^C \exp(\mathbf{z}_{G_i}^T \mathbf{W}_j)} \right)$$

For **regression tasks** it is standard to employ a squared-error loss

$$\mathcal{L} = \sum_{G_i \in T} \|\text{MLP}(\mathbf{z}_{G_i}) - y_{G_i}\|_2^2$$

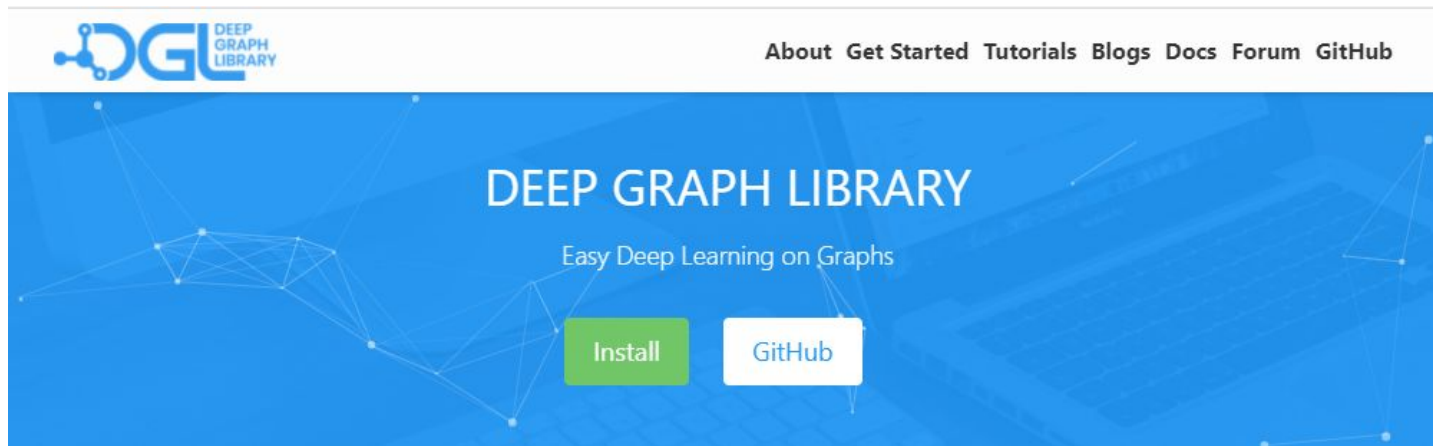
GNN loss for relation prediction

Pairwise node embedding loss functions, to minimize an empirical reconstruction loss \mathcal{L} over a set of training node pairs \mathcal{D} :

$$\mathcal{L} = \sum_{(u,v) \in \mathcal{D}} ||z_v - z_u||^2$$

where z_u and z_v are the output embedding of the two nodes u and v .

Other useful resources



The end