Graph Convolutional Networks

Duc Pham

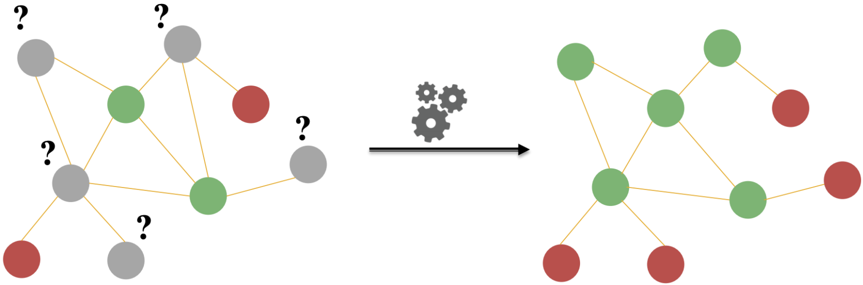
November 19, 2020

Table of Contents

Table of Contents

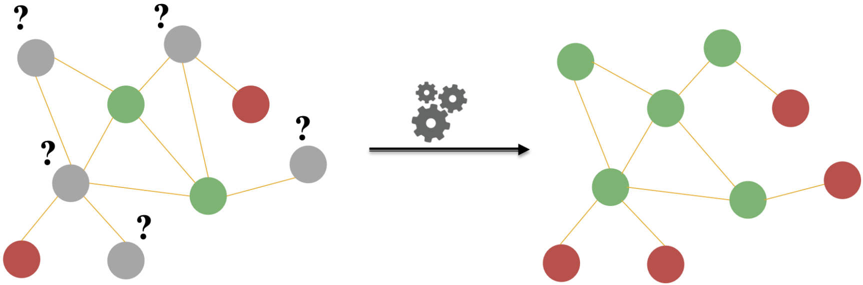
Problem: Node classification

- Only a subset of nodes has an associated label

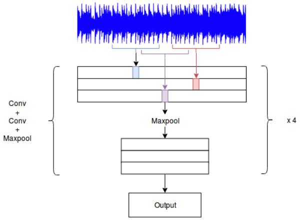


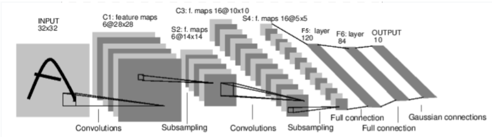
Problem: Node classification

- Only a subset of nodes has an associated label
- Each node can also have a feature vector



Convolution Operator





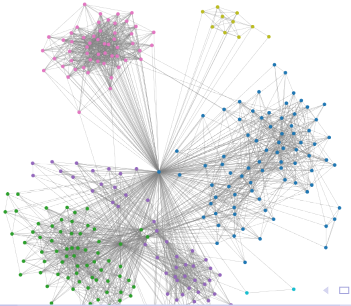


Table of Contents

Graph representations

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of nodes: $n = |\mathcal{V}|$.
All matrices below are $\in \mathbb{R}^{n \times n}$:

- Adjacency matrix A, $A_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$

Graph representations

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of nodes: $n = |\mathcal{V}|$.
All matrices below are $\in \mathbb{R}^{n \times n}$:

- Adjacency matrix A, $A_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$
- Degree matrix D, $D_{ii} = \sum_j A_{ij}$

Graph representations

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of nodes: $n = |\mathcal{V}|$.
All matrices below are $\in \mathbb{R}^{n \times n}$:

- Adjacency matrix A, $A_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$
- Degree matrix D, $D_{ii} = \sum_j A_{ij}$
- Unnormalized Laplacian $L = D - A$

Graph representations

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of nodes: $n = |\mathcal{V}|$.
All matrices below are $\in \mathbb{R}^{n \times n}$:

- Adjacency matrix A, $A_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$
- Degree matrix D, $D_{ii} = \sum_j A_{ij}$
- Unnormalized Laplacian $L = D - A$
- Symmetric normalized Laplacian:

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

Message Passing Framework

- Node u has a hidden embedding $\mathbf{h}_u^{(k)}$ at the k^{th} message passing iteration

Message Passing Framework

- Node u has a hidden embedding $\mathbf{h}_u^{(k)}$ at the k^{th} message passing iteration
- The set of neighboring nodes of u is denoted as $\mathcal{N}(u)$

Message Passing Framework

- Node u has a hidden embedding $\mathbf{h}_u^{(k)}$ at the k^{th} message passing iteration
- The set of neighboring nodes of u is denoted as $\mathcal{N}(u)$
- At each iteration, perform these two steps for each node:

Message Passing Framework

- Node u has a hidden embedding $\mathbf{h}_u^{(k)}$ at the k^{th} message passing iteration
- The set of neighboring nodes of u is denoted as $\mathcal{N}(u)$
- At each iteration, perform these two steps for each node:
  1. $\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$

Message Passing Framework

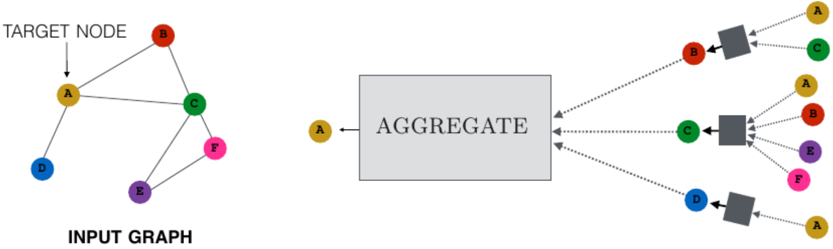
- Node u has a hidden embedding $\mathbf{h}_u^{(k)}$ at the k^{th} message passing iteration
- The set of neighboring nodes of u is denoted as $\mathcal{N}(u)$
- At each iteration, perform these two steps for each node:
    1. $\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$
    2. $\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)})$

Message Passing Framework

- Node u has a hidden embedding $\mathbf{h}_u^{(k)}$ at the k^{th} message passing iteration
- The set of neighboring nodes of u is denoted as $\mathcal{N}(u)$
- At each iteration, perform these two steps for each node:
  1. $\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$
  2. $\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)})$

Message Passing Framework

- Node u has a hidden embedding $\mathbf{h}_u^{(k)}$ at the k^{th} message passing iteration
- The set of neighboring nodes of u is denoted as $\mathcal{N}(u)$
- At each iteration, perform these two steps for each node:
  1. $\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$
  2. $\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)})$



Message passing in GCN

- GCN's update rule (graph-level equation):

$$H^{(k+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(k)}W^{(k)})$$

where $\tilde{A} = A + I$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ (adding self-loops)

Message passing in GCN

- GCN's update rule (graph-level equation):

$$H^{(k+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(k)}W^{(k)})$$

  where $\tilde{A} = A + I$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ (adding self-loops)
- Translating to message passing framework (node-level equations):

$$\begin{aligned}\mathbf{m}^{(k)}_{\mathcal{N}(u)} &= \text{AGGREGATE}^{(k)}(\{\mathbf{h}^{(k)}_v, \forall v \in \mathcal{N}(u) \cup \{u\}\}) \\ &= \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\end{aligned} \quad (1)$$

$$\mathbf{h}^{(k+1)}_u = \text{UPDATE}^{(k)}(\mathbf{m}^{(k)}_{\mathcal{N}(u)}) = \sigma(W^{(k)}\mathbf{m}^{(k)}_{\mathcal{N}(u)}) \quad (2)$$

Table of Contents

Laplace operator vs. Graph Laplacian

- **Laplace operator**:

Laplace operator vs. Graph Laplacian

- **Laplace operator**:
  - Applying to a twice-differentiable function $f, \mathbb{R}^n \rightarrow \mathbb{R}$

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

Laplace operator vs. Graph Laplacian

- **Laplace operator**:
  - Applying to a twice-differentiable function $f, \mathbb{R}^n \to \mathbb{R}$

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

  - Its eigenfunctions are the complex exponentials:

$$\Delta(e^{j\omega t}) = \frac{\partial^2 (e^{j\omega t})}{\partial x^2} = -\omega^2 e^{j\omega t}$$

    which are used in the inverse Fourier Transform.

Laplace operator vs. Graph Laplacian

- **Laplace operator**:
  - Applying to a twice-differentiable function $f, \mathbb{R}^n \to \mathbb{R}$

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

  - Its eigenfunctions are the complex exponentials:

$$\Delta(e^{j\omega t}) = \frac{\partial^2 (e^{j\omega t})}{\partial x^2} = -\omega^2 e^{j\omega t}$$

    which are used in the inverse Fourier Transform.
- **Graph Laplacian**:

Laplace operator vs. Graph Laplacian

- **Laplace operator**:
  - Applying to a twice-differentiable function $f, \mathbb{R}^n \to \mathbb{R}$

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

  - Its eigenfunctions are the complex exponentials:

$$\Delta(e^{j\omega t}) = \frac{\partial^2 (e^{j\omega t})}{\partial x^2} = -\omega^2 e^{j\omega t}$$

    which are used in the inverse Fourier Transform.
- **Graph Laplacian**:
  - Applying to a graph signal $\mathbf{x} \in \mathbb{R}^n$,

$$(L\mathbf{x})[i] = \sum_{\mathcal{N}_i} (\mathbf{x}[i] - \mathbf{x}[j])$$

Laplace operator vs. Graph Laplacian

- **Laplace operator**:
  - Applying to a twice-differentiable function $f, \mathbb{R}^n \to \mathbb{R}$

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

  - Its eigenfunctions are the complex exponentials:

$$\Delta(e^{j\omega t}) = \frac{\partial^2(e^{j\omega t})}{\partial x^2} = -\omega^2 e^{j\omega t}$$

    which are used in the inverse Fourier Transform.
- **Graph Laplacian**:
  - Applying to a graph signal $\mathbf{x} \in \mathbb{R}^n$,

$$(L\mathbf{x})[i] = \sum_{\mathcal{N}_i} (\mathbf{x}[i] - \mathbf{x}[j])$$

  - It has orthogonal eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_n$. Graph Fourier Transform?

Graph Fourier Transform

- Continuous inverse Fourier Transform:

$$f(t) = \int_{-\infty}^{\infty} F(\omega) e^{j\omega t} d\omega$$

Graph Fourier Transform

- Continuous inverse Fourier Transform:

$$f(t) = \int_{-\infty}^{\infty} F(\omega) e^{j\omega t} d\omega$$

- Given the eigendecomposition of L (note that we are using the symmetric normalized version):

$$L = U \Lambda U^T$$

The inverse Fourier transform is computed as

$$\mathbf{f} = U\mathbf{s}$$

and the forward Fourier transform is computed as

$$\mathbf{s} = U^{-1}\mathbf{f} = U^T\mathbf{f}.$$

Graph Fourier Transform

- Continuous inverse Fourier Transform:

$$f(t) = \int_{-\infty}^{\infty} F(\omega) e^{j\omega t} d\omega$$

- Given the eigendecomposition of L (note that we are using the symmetric normalized version):

$$L = U \Lambda U^T$$

The inverse Fourier transform is computed as

$$\mathbf{f} = U\mathbf{s}$$

and the forward Fourier transform is computed as

$$\mathbf{s} = U^{-1}\mathbf{f} = U^T\mathbf{f}.$$

- We can think of the eigenvalues as "frequency" values.

Convolution in Spectral Domain

- Convolution theorem:

$$\mathbf{g} * \mathbf{x} \longleftrightarrow \mathbf{G} \odot \mathbf{X}$$

Convolution in Spectral Domain

- Convolution theorem:

$$\mathbf{g} * \mathbf{x} \longleftrightarrow \mathbf{G} \odot \mathbf{X}$$

- Convolution in spectral domain:

$$\begin{aligned}
\mathbf{g} * \mathbf{x} &= \mathcal{F}^{-1}(\mathcal{F}(\mathbf{g}) \odot \mathcal{F}(\mathbf{x})) \\
&= U(\underbrace{U^T\mathbf{g}} \odot U^T\mathbf{x}) \\
&= U(\quad \boldsymbol{\theta} \odot U^T\mathbf{x}) \\
&= U \underbrace{\mathrm{diag}(\boldsymbol{\theta})} U^T\mathbf{x} \\
&= U \quad g_{\boldsymbol{\theta}} \quad U^T\mathbf{x}
\end{aligned}$$

where g_θ is the convolutional filter to be learned.

Learning the convolutional filter

$$\mathbf{g} * \mathbf{x} = U g_{\boldsymbol{\theta}} U^T \mathbf{x}$$

Learn the filter $g_{\boldsymbol{\theta}}$ directly?

- Can be arbitrary, does not depend on the graph structure (i.e. not spatially localized)

Learning the convolutional filter

$$\mathbf{g} * \mathbf{x} = U g_{\boldsymbol{\theta}} U^T \mathbf{x}$$

Learn the filter $g_{\boldsymbol{\theta}}$ directly?

- Can be arbitrary, does not depend on the graph structure (i.e. not spatially localized)
- Have to learn $O(n)$ parameters (since $\boldsymbol{\theta} \in \mathbb{R}^n$)

Learning the convolutional filter

$$\mathbf{g} * \mathbf{x} = U g_{\boldsymbol{\theta}} U^T \mathbf{x}$$

Learn the filter $g_{\boldsymbol{\theta}}$ directly?

- Can be arbitrary, does not depend on the graph structure (i.e. not spatially localized)
- Have to learn $O(n)$ parameters (since $\boldsymbol{\theta} \in \mathbb{R}^n$)
- Computation is expensive (eigendecompostion, multiplication of dense matrices)

Learning the convolutional filter

Parameterize $g_{\boldsymbol{\theta}}$ as a polynomial of the Laplacian's eigenvalues:

$$g_{\boldsymbol{\theta}}(\Lambda) = \sum_{k=0}^{K} \theta_k \Lambda^k$$

Since $(U\Lambda U^T)^k = U\Lambda^k U^T$, we can write:

$$\mathbf{g} * \mathbf{x} = U(\sum_{k=0}^{K} \theta_k \Lambda^k)U^T\mathbf{x} = (\sum_{k=0}^{K} \theta_k L^k)\mathbf{x}$$

- Guarantees spatial locality (k-hop neighborhood) (?)

Learning the convolutional filter

Parameterize $g_{\boldsymbol{\theta}}$ as a polynomial of the Laplacian's eigenvalues:

$$g_{\boldsymbol{\theta}}(\Lambda) = \sum_{k=0}^{K} \theta_k \Lambda^k$$

Since $(U \Lambda U^T)^k = U \Lambda^k U^T$, we can write:

$$\mathbf{g} * \mathbf{x} = U(\sum_{k=0}^{K} \theta_k \Lambda^k) U^T \mathbf{x} = (\sum_{k=0}^{K} \theta_k L^k)\mathbf{x}$$

- Guarantees spatial locality (k-hop neighborhood) (?)
- Only needs to learn K parameters per layer

Learning the convolutional filter

Parameterize $g_{\boldsymbol{\theta}}$ as a polynomial of the Laplacian's eigenvalues:

$$g_{\boldsymbol{\theta}}(\Lambda) = \sum_{k=0}^{K} \theta_k \Lambda^k$$

Since $(U\Lambda U^T)^k = U\Lambda^k U^T$, we can write:

$$\mathbf{g} * \mathbf{x} = U(\sum_{k=0}^{K} \theta_k \Lambda^k)U^T\mathbf{x} = (\sum_{k=0}^{K} \theta_k L^k)\mathbf{x}$$

- Guarantees spatial locality (k-hop neighborhood) (?)
- Only needs to learn K parameters per layer
- Removes the need of eigendecomposition; cost of matrix multiplication becomes $O(|\mathcal{E}|)$ as L is sparse

Learning the convolutional filter

Parameterize $g_{\boldsymbol{\theta}}$ as a polynomial of the Laplacian's eigenvalues:

$$g_{\boldsymbol{\theta}}(\Lambda) = \sum_{k=0}^{K} \theta_k \Lambda^k$$

Since $(U\Lambda U^T)^k = U\Lambda^k U^T$, we can write:

$$\mathbf{g} * \mathbf{x} = U(\sum_{k=0}^{K} \theta_k \Lambda^k) U^T \mathbf{x} = (\sum_{k=0}^{K} \theta_k L^k)\mathbf{x}$$

- Guarantees spatial locality (k-hop neighborhood) (?)
- Only needs to learn K parameters per layer
- Removes the need of eigendecomposition; cost of matrix multiplication becomes $O(|\mathcal{E}|)$ as L is sparse
- The matrix powers are unstable $\rightarrow$ difficult to optimize

Learning the convolutional filter

Use the approximated Chebyshev polynomial:

$$\sum_{k=0}^{K} \theta'_k T_k(\tilde{L}) \approx \sum_{k=0}^{K} \theta_k L^k$$

in which:

$$\begin{aligned}
\tilde{L} &= \frac{2}{\lambda_{\mathsf{max}}} L - I \\
T_k(\tilde{L}) &= 2\tilde{L}T_{k-1}(\tilde{L}) - T_{k-2}(\tilde{L}) \\
T_0 &= I \\
T_1 &= \tilde{L}
\end{aligned}$$

- No matrix powers $\rightarrow$ stable under perturbation of the coefficients

Table of Contents

Linear convolutional layer

- Final equation for spectral convolution:

$$\mathbf{g} * \mathbf{x} \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L})\mathbf{x}$$

Linear convolutional layer

- Final equation for spectral convolution:

$$\mathbf{g} * \mathbf{x} \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L})\mathbf{x}$$

- Limit to $K = 1$ (1-hop neighbor):

$$\begin{aligned} \mathbf{g} * \mathbf{x} &\approx \theta'_0 \mathbf{x} + \theta'_1 \frac{2}{\lambda_{\text{max}}}(L - I)\mathbf{x} \\ &\approx \theta'_0 \mathbf{x} - \theta'_1 D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \mathbf{x} \end{aligned}$$

Linear convolutional layer

- Final equation for spectral convolution:

$$\mathbf{g} * \mathbf{x} \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L})\mathbf{x}$$

- Limit to $K = 1$ (1-hop neighbor):

$$\begin{aligned} \mathbf{g} * \mathbf{x} &\approx \theta'_0 \mathbf{x} + \theta'_1 \frac{2}{\lambda_{\text{max}}}(L - I)\mathbf{x} \\ &\approx \theta'_0 \mathbf{x} - \theta'_1 D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \mathbf{x} \end{aligned}$$

- Share parameter for both terms (set $\theta'_0 = -\theta'_1 = \theta$):

$$\mathbf{g} * \mathbf{x} \approx \theta(I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}})\mathbf{x}$$

Linear convolutional layer

- Final equation for spectral convolution:

$$\mathbf{g} * \mathbf{x} \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L})\mathbf{x}$$

- Limit to $K = 1$ (1-hop neighbor):

$$\begin{aligned} \mathbf{g} * \mathbf{x} &\approx \theta'_0 \mathbf{x} + \theta'_1 \frac{2}{\lambda_{\mathsf{max}}}(L - I)\mathbf{x} \\ &\approx \theta'_0 \mathbf{x} - \theta'_1 D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \mathbf{x} \end{aligned}$$

- Share parameter for both terms (set $\theta'_0 = -\theta'_1 = \theta$):

$$\mathbf{g} * \mathbf{x} \approx \theta(I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}})\mathbf{x}$$

- Renormalize to reduce the eigenvalue range from $[0, 2]$ to $[0, 1]$:

$$\mathbf{g} * \mathbf{x} \approx \theta(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}})\mathbf{x}$$

Linear convolutional layer

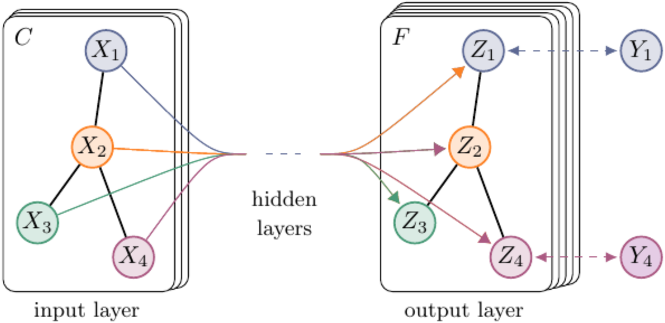
- Expand to learning F filters simultaneously and signals with C channels:

$$Z = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta$$

  where $X \in \mathbb{R}^{N \times C}$, $\Theta \in \mathbb{R}^{C \times F}$ and $Z \in \mathbb{R}^{N \times F}$.

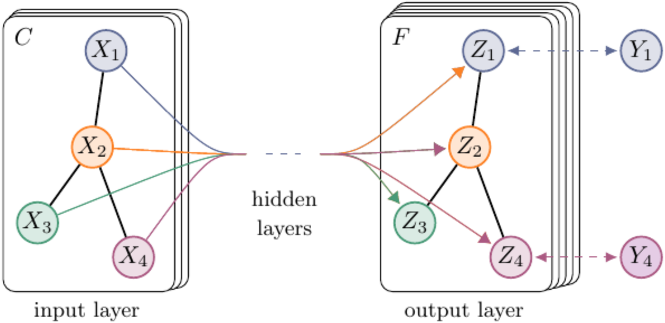
Semi-supervised node classification

- Stack multiple GCN layers



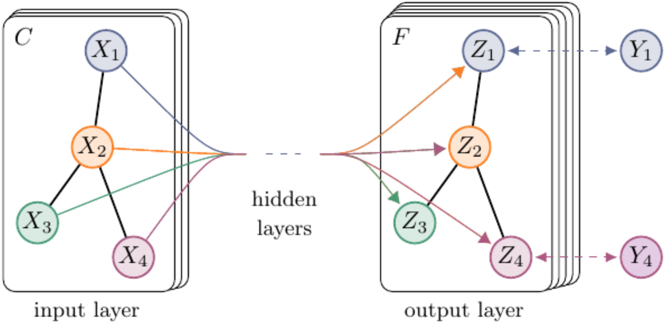
Semi-supervised node classification

- Stack multiple GCN layers
- Apply a softmax function on the final embeddings



Semi-supervised node classification

- Stack multiple GCN layers
- Apply a softmax function on the final embeddings
- Use cross-entropy loss function (only on nodes with available labels)



- Stack multiple GCN layers
- Apply a softmax function on the final embeddings
- Use cross-entropy loss function (only on nodes with available labels)
- Use batch SGD to train the model