Graph Neural Network

Deep Learning (ENGG*6600*01)

School of Engineering,
University of Guelph, ON, Canada

Course Instructor: Benyamin Ghojogh

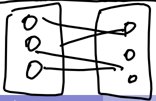
Summer 2023

Introduction

Introduction

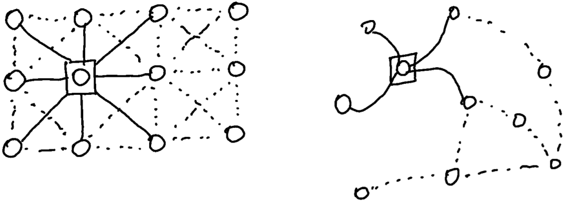
- Many real-world datasets are in the form of graphs. Some examples are social networks, protein interaction networks, the internet (World Wide Web), molecules, etc.
- Image data can be considered as graph. Every image is a graph where each pixel represents a node (vertex) connected by edges to its adjacent pixels.
- Text data can also be considered as graph. Every token (word) can be a node connected by an edge to its next token (word).
- Tasks in graph processing:
  - Graph-level task: predict the property of an entire graph. Example: predict whether an antibody protein binds to an antigen protein or not.
  - Node-level task: predict the identity or role of every node in the graph. Example: Every node has some features and there is a label for every node. For instance, if the nodes correspond to people, the label can be whether the person lives in a specific city or not.
  - Edge-level task: predict the identity or role of every edge in the graph. Example: In recommender systems for movie suggestion to users, some nodes are the users and some nodes are the movies. An edge between a user and a movie exists if the user has rated that movie and the label of the edge is the rating score. It is possible to predict the label (score) of non-existing edges between a user and a movie.

bipartite graph



Introduction

- As was said, images are special cases of graphs. The graph of an image is called the Euclidean graph or a grid graph. But what if there is a graph with some arbitrary structure or irregular shape.
- In Convolutional Neural Network (CNN) [1], there is convolution of a filter kernel with the image. The question is how to define convolution of a filter kernel with the arbitrary graph.



Graph Fourier Transform

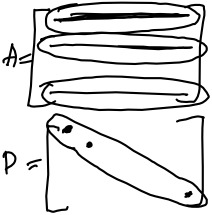
Laplacian of Graph

- Consider a graph $\boxed{\mathcal{G}(\mathcal{V}, \mathcal{E})}$ with nodes (vertices) $\mathcal{V}$ and edges $\mathcal{E}$.
- Let the number of nodes be n. The adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is a matrix whose (i,j)-th element is one if the node i is connected to the node j and is zero otherwise.
- The degree matrix of the matrix $\boldsymbol{A}$ is a diagonal matrix whose (i,i)-th element is the summation of the i-th row of the matrix $\boldsymbol{A}$, i.e.:

$$\boxed{D(i,i) := \sum_{j=1}^{n} \boldsymbol{A}(i,j),} \tag{1}$$

  where $\boldsymbol{A}(i,j)$ denotes the (i,j)-th element of $\boldsymbol{A}$.
- The Laplacian matrix of the graph $\mathcal{G}$ is defined as:

$$\boxed{\mathbb{R}^{n\times n} \ni \boldsymbol{L} := \boldsymbol{D} - \boldsymbol{A}.} \tag{2}$$



- It is noteworthy that there exist some other variants of Laplacian matrix such as [2, 3]:

$$\boxed{\boldsymbol{L} \leftarrow \boldsymbol{D}^{-\alpha}\boldsymbol{A}\boldsymbol{D}^{-\alpha},} \tag{3}$$

  where $\alpha \geq 0$ is a parameter. A common value for this parameter is $\alpha = 0.5$:

$$\boxed{\boldsymbol{L} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}.} \tag{4}$$

  This matrix is also referred to as the normalized Laplacian matrix.
- Here, the normalized Laplacian is used.

Fourier Functions

- Consider the eigenvalue decomposition of the normalized Laplacian matrix [4]:

$$\boxed{\boldsymbol{L} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top,} \tag{5}$$

where $\boldsymbol{U} = [\boldsymbol{u}_1, \dots, \boldsymbol{u}_n] \in \mathbb{R}^{n \times n}$ and $\boldsymbol{\Lambda} = \textbf{diag}([\lambda_1, \dots, \lambda_n]^\top) \in \mathbb{R}^{n \times n}$ contain the eigenvectors and eigenvalues of the normalized Laplacian matrix, respectively.

- The eigenvectors of the (normalized) Laplacian, i.e., $\boldsymbol{u}_1, \dots, \boldsymbol{u}_n$, are called the **Fourier functions**.
- The **Fourier transform** is projecting a signal $\boldsymbol{x}$ on the Fourier functions.
- The result is the coefficients of the **Fourier series**.

$$u^T x$$

$$U^T \enclose{circle}{x}$$

Graph Fourier Transform

$$U = [u_1, \dots, u_n] \qquad U^\top x$$

- **Graph Fourier transform** projects the input graph signal to a space whose orthonormal bases are the eigenvectors of the normalized Laplacian of the graph.
- For now, assume that every node of graph has a scalar feature value. Let $\mathbb{R}^n \ni \boldsymbol{x} = [x_1, \dots, x_n]^\top$ be the vector of features of all nodes in the graph, where $x_i \in \mathbb{R}$ is the feature vector of the i-th node.
- The graph Fourier transform of $\boldsymbol{x}$ is its projection onto the column space of the matrix $\boldsymbol{U}$:

$$\boxed{f(\boldsymbol{x}) = \widehat{\boldsymbol{x}} = \boldsymbol{U}^\top \boldsymbol{x}.} \tag{6}$$

- The inverse graph Fourier transform reconstructs the signal back from projection:

$$f^{-1}(\widehat{\boldsymbol{x}}) = \boldsymbol{U} f(\boldsymbol{x}) = \boldsymbol{U}\boldsymbol{U}^\top \boldsymbol{x}. \tag{7}$$

Graph Convolution

- The **graph convolution** of the input signal $\boldsymbol{x}$ with the filter $\boldsymbol{g} \in \mathbb{R}^n$ is defined as:

$$\boldsymbol{x} * \boldsymbol{g} = f^{-1}(f(\boldsymbol{x})f(\boldsymbol{g})) \overset{(6)}{=} f^{-1}(\boldsymbol{U}^\top \boldsymbol{x} \boldsymbol{U}^\top \boldsymbol{g}) \overset{(7)}{=} \boldsymbol{U}(\boldsymbol{U}^\top \boldsymbol{x} \boldsymbol{U}^\top \boldsymbol{g}). \quad \star \tag{8}$$

- We define:

$$\mathbb{R}^{n\times n} \ni \boldsymbol{G} := \mathbf{diag}(\overbrace{\overset{n\times n}{\boldsymbol{U}^\top} \overset{n\times 1}{\boldsymbol{g}}}^{n\times 1}) = \begin{bmatrix} \boldsymbol{u}_1^\top \boldsymbol{g} & 0 & \cdots & 0 \\ 0 & \boldsymbol{u}_2^\top \boldsymbol{g} & \cdots & 0 \\ & & \ddots & \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \boldsymbol{u}_n^\top \boldsymbol{g} \end{bmatrix}. \tag{9}$$

- Hence, the graph convolution can be stated as:

$$\mathbb{R}^n \ni \boldsymbol{x} * \boldsymbol{g} = \boldsymbol{U}\boldsymbol{G}\boldsymbol{U}^\top \boldsymbol{x}. \quad \star \tag{10}$$

- If every node has a feature vector rather than a feature value, the features become a matrix $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ where every row is the d-dimensional feature vector of a node. Then, the graph convolution becomes:

$$\boxed{\mathbb{R}^{n\times d} \ni \boldsymbol{X} * \boldsymbol{g} = \boldsymbol{U}\boldsymbol{G}\boldsymbol{U}^\top \boldsymbol{X}.} \tag{11}$$

ChebNet

ChebNet

- Convolutional graph neural networks have been built upon two main approaches:
  - **spectral** methods which have a graph signal processing perspective. ★
  - **spatial** methods which define graph convolution by information propagation. ★
- **Graph Convolutional Network (GCN)** [5] bridged the gap between spectral and spatial approaches.
- Recall Eq. (11). If the input of the ℓ-th layer is denoted by $\boldsymbol{H}^{(\ell-1)}$ and the output of the ℓ-th layer be $\boldsymbol{H}^{(\ell)}$, then Eq. (11) becomes:

★
$$\boxed{\boldsymbol{H}^{(\ell)} = \sigma(\overbrace{\boldsymbol{U}\boldsymbol{G}\boldsymbol{U}^\top} \boldsymbol{H}^{(\ell-1)}),} \tag{12}$$

where the activation function $\sigma(.)$ has been applied on the result of graph convolution. The first layer accepts the data features as input:

$$\boxed{\boldsymbol{H}^{(0)} = \boldsymbol{X}.} \tag{13}$$

ChebNet

- A big limitation with Eq. (12) is that $\boldsymbol{U}$ in that equation is the matrix of eigenvectors of the Laplacian of its input graph. The computational complexity of the eigenvalue decomposition of the $n \times n$ Laplacian matrix is $\mathcal{O}(n^3)$.
- **ChebNet** (2016) [6] improves the computational complexity of the convolutional neural network. It approximates the filter $\boldsymbol{g}$ by Chebyshev polynomials of the diagonal matrix of eigenvalues $\boldsymbol{\Lambda}$.
- The **Chebyshev polynomials** are:

$$\begin{cases} \boxed{T_0(x) = 1,} \\ T_1(x) = x, \\ T_i(x) = 2xT_{i-1}(x) - T_{i-2}(x). \quad \longleftarrow \end{cases} \tag{14}$$

The domain of input x for Chebyshev polynomials is $[-1, 1]$. for example, the Chebyshev polynomials are widely used for cosine expressions:

$$\boxed{\cos(i\alpha) = T_i(\cos(\alpha)).}$$

$$\cos(2\alpha) = T_2(\cos\alpha) = 2\cos\alpha\cos\alpha - 1 = \boxed{2\cos^2\alpha - 1}$$

$$\cos(0\alpha) = T_0(\cos\alpha) = 1$$

$$\cos(\alpha) = T_1(\cos\alpha) = \cos\alpha$$

ChebNet

- ChebNet approximates the filter $\boldsymbol{G}$ by a linear combination of Chebyshev polynomials of the eigenvalues $\boldsymbol{\Lambda}$:

$$\boxed{\boldsymbol{G} = \sum_{i=0}^{k} \theta_i \widetilde{T_i(\boldsymbol{\Lambda})}, \longrightarrow}$$

  where k is the order of Chebyshev polynomials.

- However, there is a problem with the domain of the Chebyshev polynomials in this equation. The eigenvalues, i.e., the diagonal elements of $\boldsymbol{\Lambda}$ are between zero and the largest eigenvalue $\lambda_{\max}$. Therefore, the eigenvalues need to be normalized as:

$$\begin{bmatrix}\lambda_{max} \\ 0\end{bmatrix} \frac{2}{\lambda_{max}} \begin{bmatrix}2 \\ 0\end{bmatrix} \xrightarrow{-1} \begin{bmatrix}1 \\ -1\end{bmatrix} \quad \underline{\mathbb{R}^{n\times n}} \ni \underbrace{\widetilde{\boldsymbol{\Lambda}} := \underbrace{\frac{2}{\lambda_{\max}}\boldsymbol{\Lambda} - \boldsymbol{I}_n,}} \longrightarrow [-1, +1] \tag{15}$$

  where $\boldsymbol{I}_n$ is the $n \times n$ identity matrix. The values in the normalized eigenvalue matrix are in range $[-1, 1]$ as required by the domain of Chebyshev polynomials.

- Hence, the approximation of the filter $\boldsymbol{g}$ is:

$$\boxed{\boldsymbol{G} = \sum_{i=0}^{k} \theta_i T_i(\widetilde{\boldsymbol{\Lambda}}).} \tag{16}$$

ChebNet

- Recall Eq. (10):

$$\boldsymbol{x} * \boldsymbol{g} = \boldsymbol{U}\boldsymbol{G}\boldsymbol{U}^\top \boldsymbol{x} \overset{(16)}{=} \boldsymbol{U}\Big(\sum_{i=0}^{k} \theta_i T_i(\widetilde{\boldsymbol{\Lambda}})\Big)\boldsymbol{U}^\top \boldsymbol{x} = \sum_{i=0}^{k} \theta_i \boldsymbol{U} T_i(\widetilde{\boldsymbol{\Lambda}})\boldsymbol{U}^\top \boldsymbol{x}. \tag{17}$$

- The matrix $\boldsymbol{U}$ is orthogonal, i.e., its columns are orthonormal, because it is the matrix of eigenvectors. For an orthonormal transformation, the following holds:

$$\boxed{\boldsymbol{U} T_i(\widetilde{\boldsymbol{\Lambda}})\boldsymbol{U}^\top = T_i(\boldsymbol{U}\widetilde{\boldsymbol{\Lambda}}\boldsymbol{U}^\top).} \tag{18}$$

- Similar to Eq. (15), we define:

$$\boxed{\widetilde{\boldsymbol{L}} := \frac{2}{\lambda_{\max}}\boldsymbol{L} - \boldsymbol{I}_n,} \tag{19}$$

where $\lambda_{\max}$ is largest eigenvalue of the normalized Laplacian $\boldsymbol{L}$. Then, according to Eq. (15) and similar to Eq. (5), the eigenvalue decomposition of $\widetilde{\boldsymbol{L}}$ becomes:

$$L = U\Lambda U^\top \implies \boxed{\widetilde{\boldsymbol{L}} = \boldsymbol{U}\widetilde{\boldsymbol{\Lambda}}\boldsymbol{U}^\top.} \tag{20}$$

- Combining Eqs. (18) and (20) gives:

$$\boxed{\boldsymbol{U} T_i(\widetilde{\boldsymbol{\Lambda}})\boldsymbol{U}^\top = T_i(\widetilde{\boldsymbol{L}}).} \tag{21}$$

- Putting Eq. (21) in Eq. (17) gives:

$$\boxed{\boldsymbol{x} * \boldsymbol{g} = \sum_{i=0}^{k} \theta_i T_i(\widetilde{\boldsymbol{L}})\boldsymbol{x}.} \tag{22}$$

ChebNet

- Comparing Eqs. (12) and (22):

$$\boxed{\boldsymbol{H}^{(\ell)} = \sigma(\boldsymbol{U}\boldsymbol{G}\boldsymbol{U}^\top \boldsymbol{H}^{(\ell-1)}),} \rightarrow \text{regular GC}$$

$$\boxed{\boldsymbol{x} * \boldsymbol{g} = \sum_{i=0}^{k} \theta_i \boxed{T_i(\widetilde{\boldsymbol{L}})} \boldsymbol{x},} \dashrightarrow \text{ChebNet}$$

  shows that ChebNet resolves the limitation of eigenvalue decomposition of the Laplacian. In fact, it uses the approximation of Chebyshev polynomials and does not perform eigenvalue decomposition.

Graph Convolutional Network

Graph Convolutional Network

- **Graph Convolutional Network (GCN)** (2017) [5] is the first-order approximation of the ChebNet. In Eq. (22), it approximates the Chebyshev polynomials to its first order ($k = 1$):

$$\sum_{i=1}^{k} \boxed{T_i(\widetilde{\boldsymbol{L}}) \approx T_0(\widetilde{\boldsymbol{L}}) + T_1(\widetilde{\boldsymbol{L}}).} \tag{23}$$

In other words:

$$k=1 \leftarrow \quad \boldsymbol{x} * \boldsymbol{g} \approx \underbrace{\sum_{i=0}^{1} \theta_i T_i(\widetilde{\boldsymbol{L}})\boldsymbol{x}} = \underbrace{\theta_0 \overbrace{T_0(\widetilde{\boldsymbol{L}})\boldsymbol{x}}^{1} + \theta_1 \overbrace{T_1(\widetilde{\boldsymbol{L}})\boldsymbol{x}}^{\widetilde{L}}} \overset{(14)}{=} \underline{\theta_0 \boldsymbol{x} + \theta_1 \widetilde{\boldsymbol{L}}\boldsymbol{x}}.$$

- More number of learnable parameters may result in overfitting [7]. To reduce the number of parameters and to avoid overfitting, it is assumed that $\theta_0 = \theta_1 = \theta$, so:

$$\boldsymbol{x} * \boldsymbol{g} = \underbrace{\theta \boldsymbol{x} + \theta \widetilde{\boldsymbol{L}}\boldsymbol{x}} = \underbrace{\theta(\boldsymbol{I} + \widetilde{\boldsymbol{L}})\boldsymbol{x}} \overset{(19)}{=} \theta(\underbrace{\cancel{\boldsymbol{I}} + \frac{2}{\lambda_{\max}}\boldsymbol{L} - \cancel{\boldsymbol{I}}})\boldsymbol{x} = \underbrace{\theta \frac{2}{\lambda_{\max}} \boldsymbol{L}\boldsymbol{x}}.$$

- It is possible to absorb the constant $2/\lambda_{\max}$ into the learnable parameters and simply the graph convolution as:

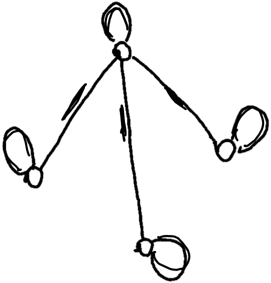
$$\boxed{\boldsymbol{x} * \boldsymbol{g} = \underbrace{\theta \boldsymbol{L}\boldsymbol{x}} \overset{(4)}{=} \theta \underbrace{\boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}}\boldsymbol{x}.} \tag{24}$$

Graph Convolutional Network

- We found:

$$\boxed{\boldsymbol{x} * \boldsymbol{g} = \theta \boldsymbol{D}^{-1/2} \boldsymbol{A} \boldsymbol{D}^{-1/2} \boldsymbol{x}.}$$

- It has been empirically observed that this results in some instability in training of GCN. Therefore, an additional assumption is added to have self-loops on the nodes meaning that every node has an edge from it to itself.



Graph Convolutional Network

- Mathematically, it means that the main diagonal of the adjacency matrix should become one by adding the identity matrix to it. Therefore, we define:

$$\widetilde{\boldsymbol{A}} := \boldsymbol{A} + \boldsymbol{I},$$

$$\underbrace{\widetilde{\boldsymbol{D}}(i,j)} := \sum_{j=1}^{n} \widetilde{\boldsymbol{A}}(i,j), \tag{25}$$

$$\underbrace{\bar{\boldsymbol{L}}} := \underbrace{\widetilde{\boldsymbol{D}}^{-1/2}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-1/2}}.$$

- As a result, the Eq. (24) is replaced by:

$$\underbrace{\boldsymbol{x} * \boldsymbol{g} = \theta \overbrace{\widetilde{\boldsymbol{D}}^{-1/2}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-1/2}}}\boldsymbol{x} = \underbrace{\theta\bar{\boldsymbol{L}}\boldsymbol{x}}.$$

  In matrix form, if every row of $\underline{\boldsymbol{X} \in \mathbb{R}^{n\times d}}$ is the d-dimensional feature vector of a node, this equation becomes $\underline{\boldsymbol{x} * \boldsymbol{g} = \bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\theta}}$ where $\boldsymbol{\theta} \in \mathbb{R}^{d}$. If there is a need to have f feature maps after the convolution, then this equation can become $\boxed{\boldsymbol{x} * \boldsymbol{g} = \bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\Theta}}$ where $\boldsymbol{\Theta} \in \mathbb{R}^{d\times f}$.

- As a result, if the input of the ℓ-th layer is denoted by $\boldsymbol{H}^{(\ell-1)}$ and the output of the ℓ-th layer be $\boldsymbol{H}^{(\ell)}$, then Eq. (11) becomes:

$$\boxed{\underbrace{\boldsymbol{H}^{(\ell)}} = \sigma(\bar{\boldsymbol{L}}\underbrace{\overbrace{\boldsymbol{H}^{(\ell-1)}}\boldsymbol{\Theta}}),} \tag{26}$$

  where the activation function $\sigma(.)$ has been applied on the result of graph convolution. The first layer accepts the data features as input, as stated in Eq. (13).

Graph Convolutional Network

- Eq. (26) is the graph convolution performed in every layer of GCN where $\boldsymbol{\Theta}$ is the matrix of learnable weights in the layer.
- Comparing Eqs. (12) and (26):

$$\boxed{\begin{aligned} \boldsymbol{H}^{(\ell)} &= \sigma(\boldsymbol{U}\boldsymbol{G}\boldsymbol{U}^\top \boldsymbol{H}^{(\ell-1)}), \\ \boldsymbol{H}^{(\ell)} &= \sigma(\bar{\boldsymbol{L}}\boldsymbol{H}^{(\ell-1)}\boldsymbol{\Theta}), \end{aligned}}$$

shows that GCN resolves the limitation of eigenvalue decomposition of the Laplacian. It makes use of the approximation of Chebyshev polynomials and does not perform eigenvalue decomposition.

Graph Convolutional Network vs. Feedforward Network

- In the fully connected layer of a feedforward neural network, the operation of the layer is:

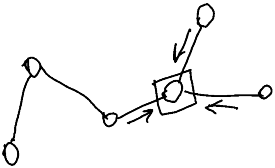
★
$$\boxed{\boldsymbol{H}^{(\ell)} = \sigma(\underbrace{\boldsymbol{H}^{(\ell-1)}}\boldsymbol{\Theta}).} \qquad \cancel{\sigma}\, H^{(\ell-1)}\Theta \tag{27}$$

However, according to Eqs. (25) and (26), the operation of convolution in a layer of GCN is:

★
$$\boxed{\boldsymbol{H}^{(\ell)} = \sigma(\boxed{\widetilde{\boldsymbol{D}}^{-1/2}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-1/2}}\boldsymbol{H}^{(\ell-1)}\boldsymbol{\Theta}).} \tag{28}$$

- Comparing Eqs. (27) and (28) shows the relation of GCN and feedforward network. In a fully connected layer of feedforward network, all the features of previous layer $\boldsymbol{H}^{(\ell-1)}$ are fed to the next layer through a linear transformation by the weight matrix $\boldsymbol{\Theta}$ followed by a nonlinear activation function. However, in graph neural network, firstly the adjacency matrix defines which nodes (or features) are connected to each other, and then the linear transformation by the weight matrix $\boldsymbol{\Theta}$ is performed followed by a nonlinear activation function. In other words, the adjacency matrix determines which nodes should impact the features of every node (see this this figure).

when complete graph (all nodes are connected) reduces to fully conned layer



when not connected to a node $\tilde{A} = 0 \rightarrow H^{(\ell)} = 0$ not have impact from that node

More General Frameworks of Graph Convolutional Network

More General Frameworks of GCN

- The update rule of every layer, i.e., Eq. (28), can be restated as:

$$\boxed{\boldsymbol{h}_i^{(\ell)} = \sigma\Big(\sum_{j\in\mathcal{N}_i} \boldsymbol{\Theta}\boldsymbol{h}_j^{(\ell-1)}\Big),} \tag{29}$$

  for the i-th neuron in the ℓ-th layer, where $\mathcal{N}_i$ denotes the neighbors of the i-th node (or neuron) in the input of the layer. This update rule is called **sum pooling** because it sums over the neighbors.

- There is a problem with sum pooling. Summing the contents of the neighboring nodes (or neurons) increases the scale of the output feature gradually over multiple layers.
- To resolve this issue, it is possible to normalize the input of the activation function by $\widetilde{\boldsymbol{D}}^{-1}$:

$$\star \quad \boldsymbol{H}^{(\ell)} = \sigma\big(\widetilde{\boldsymbol{D}}^{-1}\widetilde{\boldsymbol{A}}\boldsymbol{H}^{(\ell-1)}\boldsymbol{\Theta}\big), \tag{30}$$

  where $\widetilde{\boldsymbol{D}}$ is defined in Eq. (25). Eq. (30) can be stated for every node i:

$\tilde{D}(i,i) = |N_i|$

$\tilde{D}(i,i)^{-1} = \frac{1}{|N_i|}$

$$\boxed{\boldsymbol{h}_i^{(\ell)} = \sigma\Big(\sum_{j\in\mathcal{N}_i} \frac{1}{|\mathcal{N}_i|}\boldsymbol{\Theta}\boldsymbol{h}_j^{(\ell-1)}\Big),} \tag{31}$$

  where $|\mathcal{N}_i|$ denotes the number of neighbors for the i-th node. This is because the degree matrix counts the number of neighbors for nodes.

- The update rule in Eq. (30) or (31) is called the **mean pooling**.

More General Frameworks of GCN

- Rather than Eq. (30), it is possible to use symmetric normalization in mean pooling:

$$\boxed{\boldsymbol{H}^{(\ell)} = \sigma\Big(\widetilde{\boldsymbol{D}}^{-1/2}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-1/2}\boldsymbol{H}^{(\ell-1)}\boldsymbol{\Theta}\Big).} \tag{32}$$

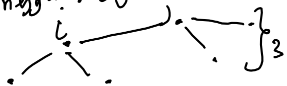
- Eq. (32) can be stated for every node i:

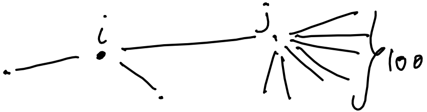
$$\star \quad \boxed{\boldsymbol{h}_i^{(\ell)} = \sigma\Big(\sum_{j\in\mathcal{N}_i}\frac{1}{\sqrt{|\mathcal{N}_i||\mathcal{N}_j|}}\boldsymbol{\Theta}\boldsymbol{h}_j^{(\ell-1)}\Big),} \tag{33}$$

  which is called **mean pooling** with symmetric normalization.

- Comparing Eqs. (28) and (32) shows that the original GCN uses mean pooling with symmetric normalization.
- Eq. (33) means that for every node i, if the node j is a neighbor, its impact on the node i should be more if the node j has few number of neighbors ($|\mathcal{N}_j|$ is small). However, its impact on the node i should be less if the node j has large number of neighbors ($|\mathcal{N}_j|$ is large) because the node i would be one of the many neighbors of node j. Note that this impact is not considered in Eq. (31).

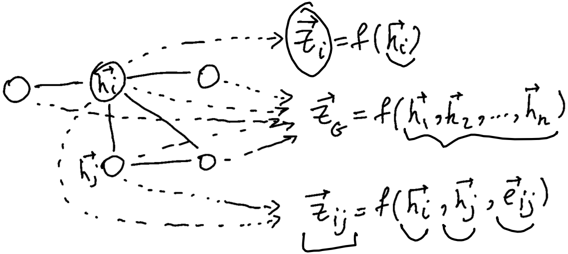
i one of few neighbors of j





More General Frameworks of GCN

- Different tasks:
  - Node classification/regression: after the multiple layers of convolution, the $\boldsymbol{h}_i$'s of the last layer are used in the loss function for the node classification or regression.
  - Graph classification/regression: after the multiple layers of convolution, all the $\boldsymbol{h}_i$'s of the last layer are aggregated and used in the loss function for the graph classification or regression.
  - Link classification/regression: after the multiple layers of convolution, the $\boldsymbol{h}_i$'s and the edges of the last layer are used in the loss function for the link classification or regression.


$\vec{h}_i$
$\vec{h}_j$
$\vec{z}_i = f(\vec{h}_i)$
$\vec{z}_G = f(\underbrace{\vec{h}_1, \vec{h}_2, \dots, \vec{h}_n})$
$\vec{z}_{ij} = f(\vec{h}_i, \vec{h}_j, \vec{e}_{ij})$

Graph Attention Network

Graph Attention Network

- As was seen in Eqs. (29), (31), and (33), the linear combination in pooling can have weights.
- **Graph Attention Network (GAT)** (2017) [8] adopts attention mechanisms to learn the relative weights between two connected nodes. In the pooling operation, the weights of attention are added:

$$\boxed{\boldsymbol{h}_i^{(\ell)} = \sigma\Big(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \boldsymbol{h}_j^{(\ell-1)}\Big),} \tag{34}$$

where the attention weight α_{ij} measures the influence of node j to node i.

$$\underbrace{\alpha_{ij}} = \underbrace{\text{attention}(\underbrace{\boldsymbol{h}_i^{(\ell-1)}}, \underbrace{\boldsymbol{h}_j^{(\ell-1)}}).} \tag{35}$$

- The attention weight can be computed by a attention function $a(.)$ between $\boldsymbol{h}_i^{(\ell-1)}$ and $\boldsymbol{h}_j^{(\ell-1)}$:

$$\boxed{a_{ij} = a(\boldsymbol{h}_i^{(\ell-1)}, \boldsymbol{h}_j^{(\ell-1)}).} \tag{36}$$

This attention function may also consider the edge between the nodes i and j:

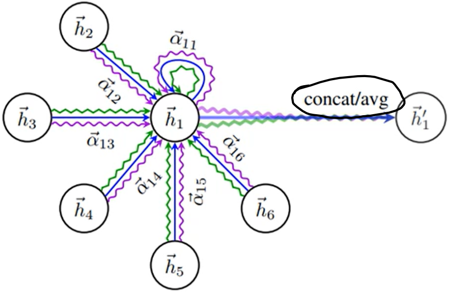
$$\boxed{a_{ij} = a(\boldsymbol{h}_i^{(\ell-1)}, \boldsymbol{h}_j^{(\ell-1)}, \underline{e_{ij}}).} \tag{37}$$

Graph Attention Network

- This attention function $a(.)$ can be a transformer autoencoder [9].
- However, GAT models the attention function $a(.)$ as a single-layer feedforward neural network. This single-layer neural network calculates the attention between nodes.
- Finally, the attention values of every node are normalized in a softmax form to obtain the attention weights:

$$\boxed{\alpha_{ij} = \frac{e^{a_{ij}}}{\sum_{k \in \mathcal{N}_i} e^{a_{ik}}},} \tag{38}$$

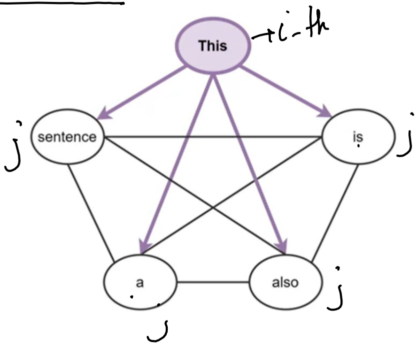
where the summation in the denominator is over the neighbors of the i-th node.



credit of image: [9]

Graph Attention Network

- Transformers [9] are special cases of graph attention neural networks.
- In fact, every sentence or sequence can be considered as a graph where GAT can calculate the attention between the tokens in the sequence. For example, the graph for the sentence "This is also a sentence" is depicted below.



Graph Attention Network

- In the following, GAT and transformer are compared.
- In GAT, the attention is $a_{ij} = a(\boldsymbol{h}_i^{(\ell-1)}, \boldsymbol{h}_j^{(\ell-1)})$ where the $\boldsymbol{h}_i^{(\ell-1)}$ and $\boldsymbol{h}_j^{(\ell-1)}$ are passed through a single-layer network with some weight $\boldsymbol{W}$. Therefore, the attention is calculated between $\boldsymbol{W}^\top \boldsymbol{h}_i^{(\ell-1)}$ and $\boldsymbol{W}^\top \boldsymbol{h}_j^{(\ell-1)}$ after feeding to the single layer of network. In transformer, on the other hand, the attention is $a_{ij} = a(\boldsymbol{q}_i, \boldsymbol{k}_j)$ where the query $\boldsymbol{q}_i$ and key $\boldsymbol{k}_j$ are different linear transformations of the same tokens, i.e., $\boldsymbol{q}_i = \boldsymbol{W}_Q^\top \boldsymbol{x}$ and $\boldsymbol{k}_i = \boldsymbol{W}_K^\top \boldsymbol{x}$ [10]. Therefore, the difference of GAT and transformer is that GAT uses the shared learnable wight for the query and key but transformer uses different learnable weights for them.
- Another difference between GAT and transformer is that GAT uses a single-layer feedforward neural network as the attention function $a(.)$. However, in transformer, this function is [10]:

$$\boxed{a(\boldsymbol{q}_i, \boldsymbol{k}_j) = \frac{1}{\sqrt{p}} \boldsymbol{q}_i^\top \boldsymbol{k}_j,}$$

  where p is the dimensionality of query and key.
- The last difference of GAT and transformer is the softmax form. GAT sums over the neighbors in the denominator of the softmax form (see Eq. (38)). However, transformer sums over all tokens in the sequence:

$$\alpha_{ij} = \frac{e^{a_{ij}}}{\sum_{k=1}^{n} e^{a_{ik}}}.$$

Graph Autoencoder

Graph Autoencoder

- Consider the following autoencoder where the encoder has two layers. This autoencoder accepts a graph as its input; hence, its name is **Graph Autoencoder (GAE)** (2016) [11].
- According to Eq. (26), the first layer of the encoder is:

$$\boxed{\boldsymbol{H}^{(1)} = \sigma(\bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\Theta}_1),} \tag{39}$$

  where $\boldsymbol{\Theta}_1$ is the learnable weight matrix of the first layer, $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ is the feature vectors of nodes stacked row-wise, $\bar{\boldsymbol{L}}$ is defined in Eq. (25) based on the adjacency matrix of the graph, $\sigma(.)$ is usually the ReLU activation function [12], and $\boldsymbol{H}^{(1)}$ is the output of the first layer.
- Again, according to Eq. (26), the second layer of the encoder is:

$$\boxed{\boldsymbol{H}^{(2)} = \bar{\boldsymbol{L}}\boldsymbol{H}^{(1)}\boldsymbol{\Theta}_2,} \tag{40}$$

  where $\boldsymbol{\Theta}_2$ is the learnable weight matrix of the second layer, $\boldsymbol{H}^{(2)}$ is the output of the second layer, and the second layer is assumed not to have an actvation function.
- Putting Eq. (39) in Eq. (40) gives the following function which we denote by $\text{GCN}(\boldsymbol{X}, \boldsymbol{A}; \boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2)$:

$$\boxed{\text{GCN}(\boldsymbol{X}, \boldsymbol{A}; \boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2) := \boxed{\bar{\boldsymbol{L}}\,\sigma(\bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\Theta}_1)\boldsymbol{\Theta}_2.}} \tag{41}$$

- There are two types of GAE, i.e., graph reconstruction autoencoder and graph variational autoencoder [11]. These autoencoders are introduced in the following.

Graph Reconstruction Autoencoder

- In the graph reconstruction autoencoder, also called the non-probabilistic GAE, the encoder is Eq. (41) with two layers. The p-dimensional latent embeddings of nodes, denoted by $\boxed{\boldsymbol{Z} \in \mathbb{R}^{n \times p}}$, are obtained as:

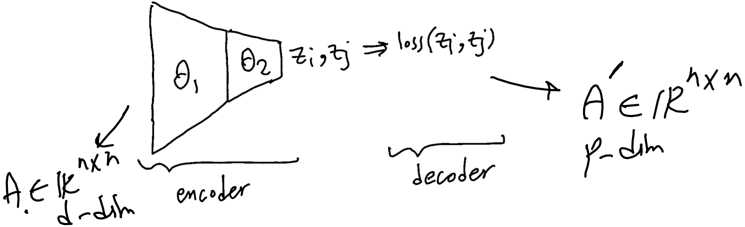
$$\boxed{\boldsymbol{Z}} = \mathrm{GCN}(\boldsymbol{X}, \boldsymbol{A}; \boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2) := \underbrace{\bar{\boldsymbol{L}}\,\sigma(\bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\Theta}_1)\boldsymbol{\Theta}_2.}$$

- The decoder of graph reconstruction autoencoder does not contain any layers but models measuring similarity between the embedding vectors of the nodes (see this figure). It is the sigmoid function of $\boldsymbol{z}_i^\top \boldsymbol{z}_j$ to show the score of similarity (inner product) of the latent variables $\boldsymbol{z}_i$ and $\boldsymbol{z}_j$. In other words, it reconstructs the adjacency matrix but with the latent embeddings of nodes rather than the nodes directly:

$$\left\{\begin{aligned} &\widehat{\boldsymbol{A}} = \overbrace{\text{sigmoid}}(\boxed{\boldsymbol{Z}\boldsymbol{Z}^\top}), \text{ or} && (42)\\ &\widehat{\boldsymbol{A}}(i,j) = \boxed{\frac{1}{1+e^{-\boldsymbol{z}_i^\top \boldsymbol{z}_j}}.} && (43)\end{aligned}\right.$$

idea of MDS

- The graph reconstruction autoencoder is depicted in this figure.



Θ_1
Θ_2
$z_i, z_j \Rightarrow \mathrm{loss}(z_i, z_j)$
$A' \in \mathbb{R}^{n \times n}$
p-dim
$A \in \mathbb{R}^{n \times n}$
d-dim
encoder
decoder

Graph Reconstruction Autoencoder

- The loss is the mean squared error between the adjacency matrix and the reconstructed adjacency matrix:

$$\boxed{\underset{\boldsymbol{\theta}}{\text{minimize}} \quad \|\widehat{\boldsymbol{A}} - \boldsymbol{A}\|_F^2,} \tag{44}$$

where $\|.\|_F$ denotes the Frobenius norm and $\boldsymbol{\theta} := \{\boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2\}$ is the learnable parameters. This loss function is minimized by backpropagation [13].

no decoder

decoder : calculating $\hat{A}$

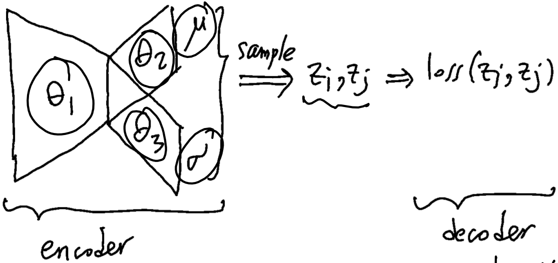
Graph Variational Autoencoder

- Graph variational autoencoder uses these two GCN modules for estimating the mean and variance in the latent space by the encoder:

$$\star\ \boxed{\mathrm{GCN}_{\boldsymbol{\mu}}(\boldsymbol{X}, \boldsymbol{A}; \boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2) := \bar{\boldsymbol{L}}\,\sigma(\bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\Theta}_1)\boldsymbol{\Theta}_2,} \tag{45}$$

$$\star\ \boxed{\mathrm{GCN}_{\boldsymbol{\sigma}}(\boldsymbol{X}, \boldsymbol{A}; \boldsymbol{\Theta}_1, \boldsymbol{\Theta}_3) := \bar{\boldsymbol{L}}\,\sigma(\bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\Theta}_1)\boldsymbol{\Theta}_3,} \tag{46}$$

where the first layer is shared between them as shown in this figure.



$$x_i \Rightarrow \theta_1, \theta_2 \Rightarrow \mu_i \mapsto N(\mu_i, \sigma_i) \to z_i$$
$$\theta_1, \theta_3 \to \sigma_i$$
$$x_j \Rightarrow \theta_1, \theta_2 \to \mu \mapsto N(\mu_j, \sigma_j) \to z_j$$
$$\theta_1, \theta_3 \to -$$

Graph Variational Autoencoder

- As the latent variables of the nodes are independent, the encoder of graph variational autoencoder models the following conditional distribution:

  $$\boxed{\underbrace{q(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{A})} = \prod_{i=1}^{n} q(\boldsymbol{z}_i|\boldsymbol{X},\boldsymbol{A}),} \tag{47}$$

  where $\boldsymbol{Z} \in \mathbb{R}^{n \times p}$ contains the p-dimensional latent variables and $\boldsymbol{z}_i \in \mathbb{R}^p$ is the latent variable of the i-th node whose conditional distribution is a Gaussian distribution:

  $$\underbrace{q(\boldsymbol{z}_i|\boldsymbol{X},\boldsymbol{A})} = \underbrace{\mathcal{N}(\boldsymbol{z}_i \mid \boldsymbol{\mu}_i, \mathbf{diag}(\boldsymbol{\sigma}_i^2)),} \tag{48}$$

  where **diag**(.) makes a diagonal matrix with its input as as the diagonal of matrix.
- The latent variables $\{\boldsymbol{z}_i\}_{i=1}^{n}$ are sampled from the multivariate joint distribution in Eq. (47).

$$z_i \sim q(z_i \mid X, A)$$
$$z_j \sim q(z_j \mid x, A)$$

Graph Variational Autoencoder

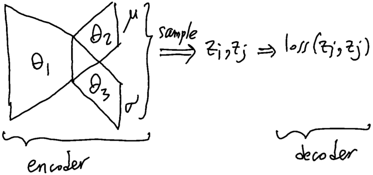
- The decoder of the autoencoder models the following conditional distribution:

$$\underbrace{q(\boldsymbol{A}|\boldsymbol{Z})} = \overbrace{\prod_{i=1}^{n}\prod_{j=1}^{n}} \underbrace{p(\boldsymbol{A}(i,j)|\boldsymbol{z}_i,\boldsymbol{z}_j)}, \tag{49}$$

where $p(\boldsymbol{A}(i,j)|\boldsymbol{z}_i,\boldsymbol{z}_j)$ is the sigmoid function of $\boldsymbol{z}_i^\top \boldsymbol{z}_j$ to show the probability of similarity (inner product) of the latent variables $\boldsymbol{z}_i$ and $\boldsymbol{z}_j$:

$$\boxed{p(\boldsymbol{A}(i,j)=1|\boldsymbol{z}_i,\boldsymbol{z}_j) = \frac{1}{1+e^{-\boldsymbol{z}_i^\top \boldsymbol{z}_j}}.} \bigstar \tag{50}$$

As a result, the decoder of graph variational autoencoder does not contain any layers but models measuring similarity between the sampled latent variables in the latent space (see this figure).



Graph Variational Autoencoder

- The graph variational autoencoder maximizes the Evidence Lower Bound (ELBO) in variational inference [14]:

$$\star\ \boxed{\underset{\boldsymbol{\theta}}{\text{maximize}}}\ \underbrace{\mathbb{E}_{q(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{A})}\big[\log(p(\boldsymbol{A}\,|\,\boldsymbol{Z}))\big]} - \overbrace{\text{KL}\big(q(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{A})\|\,\underline{p(\boldsymbol{Z})}\big)}. \tag{51}$$

  where KL(.‖.) denotes the Kullback-Leibler (KL) divergence [15], $p(\boldsymbol{Z})$ is the desired prior distribution such as some Gaussian distribution, and $\boldsymbol{\theta} := \{\boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2, \boldsymbol{\Theta}_3\}$ is the learnable parameters.

- The graph variational autoencoder is trained by backpropagation [13]. In backpropagation, the loss function should be minimized; therefore, the loss is the ELBO times -1:

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\quad \ominus\,\mathbb{E}_{q(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{A})}\big[\log(p(\boldsymbol{A}\,|\,\boldsymbol{Z}))\big] \oplus \text{KL}\big(q(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{A})\|\,p(\boldsymbol{Z})\big). \tag{52}$$

- minimizing this loss function tries to learn generation of the adjacency matrix $\boldsymbol{A}$ given the sampled latent variables $\boldsymbol{Z}$ while the conditional distribution of the latent variable given the graph and its adjacency matrix becomes similar to the desired prior distribution of the latent space.

Acknowledgment

- Some slides of this slide deck are inspired by teachings of Prof. Ali Ghodsi at University of Waterloo, Department of Statistics.
- Graph neural network in PyTorch Geometric: `https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html`
- Good tutorial on PyTorch Geometric by Antonio Longa: `https://www.youtube.com/playlist?list=PLGMXrbDNfqTzqxB1IGgimuhtfAhGd8lHF`

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] Y. Weiss, "Segmentation using eigenvectors: a unifying view," in *Proceedings of the seventh IEEE international conference on computer vision*, vol. 2, pp. 975–982, IEEE, 1999.

[3] A. Ng, M. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," *Advances in neural information processing systems*, vol. 14, pp. 849–856, 2001.

[4] B. Ghojogh, F. Karray, and M. Crowley, "Eigenvalue and generalized eigenvalue problems: Tutorial," *arXiv preprint arXiv:1903.11240*, 2019.

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[6] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, pp. 3844–3852, 2016.

[7] B. Ghojogh and M. Crowley, "The theory behind overfitting, cross validation, regularization, bagging, and boosting: tutorial," *arXiv preprint arXiv:1905.12787*, 2019.

[8] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2017.

References (cont.)

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, vol. 30, 2017.

[10] B. Ghojogh and A. Ghodsi, "Attention mechanism, transformers, BERT, and GPT: tutorial and survey," 2020.

[11] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.

[12] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[14] B. Ghojogh, A. Ghodsi, F. Karray, and M. Crowley, "Factor analysis, probabilistic principal component analysis, variational inference, and variational autoencoder: Tutorial and survey," *arXiv preprint arXiv:2101.00734*, 2021.

[15] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.