Graph Neural Network, ChebNet, Graph Convolutional Network, and Graph Autoencoder: Tutorial and Survey

Benyamin Ghojogh, Ali Ghodsi {BGHOJOGH, ALI.GHODSI}@UWATERLOO.CA
Waterloo, Ontario, Canada

Abstract

This is a tutorial paper on graph neural networks including ChebNet, graph convolutional network, graph attention network, and graph autoencoder. It starts with Laplacian of graph, graph Fourier transform, and graph convolution. Then, it is explained how Chebyshev polynomials are used in graph networks to have ChebNet. Afterwards, graph convolutional network and its general framework are introduced. Then, graph attention network is explained as a combination of attention mechanism and graph neural networks. Finally, graph reconstruction autoencoder and graph variational autoencoder are introduced.

1. Introduction

Many real-world datasets are in the form of graphs. Some examples of graph data are social networks, protein interaction networks, the internet (World Wide Web), and molecules. Image data can also be considered as graphs. Every image is a graph where each pixel represents a node (vertex) connected by edges to its adjacent pixels (Cheung et al., 2018; Sudderth & Freeman, 2008). Moreover, text data can be considered as graphs (Koncel-Kedziorski et al., 2019). Every token (word) can be a node connected by an edge to its next token (word). Another example is use of graphs in biology by modeling proteins and antibodies as graphs with amino acids as the nodes. For instance, GearNet (Zhang et al., 2023) and PECAN (Pittala & Bailey-Kellogg, 2020) are graph representations of proteins in neural networks.

There are different tasks in graph processing:

- Graph-level task: it predicts the property of the entire graph. For example, it predicts whether an antibody protein binds to an antigen protein or not (Myung et al., 2022).
- Node-level task: it predicts the identity or role of every node in the graph. In this task, 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: it predicts the identity or role of every edge in the graph. For example, in recommender systems for movie suggestion to users, some nodes are the users and some nodes are the movies (Wang et al., 2021). 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.

As mentioned, images are special cases of graphs. The graph of an image is called the Euclidean graph or a grid graph (see Fig. 1-a). In Convolutional Neural Network (CNN) (LeCun et al., 1998), there is convolution of a filter kernel with the Euclidean graph of the image. However, what about a graph with some arbitrary structure or irregular shape (see Fig. 1-b)? The question is how to define convolution of a filter kernel with the arbitrary graph. This question is answered in the following section.

2. Graph Fourier Transform

2.1. Laplacian of Graph

Consider a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ 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.:

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

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 (Ghojogh et al., 2023b):

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

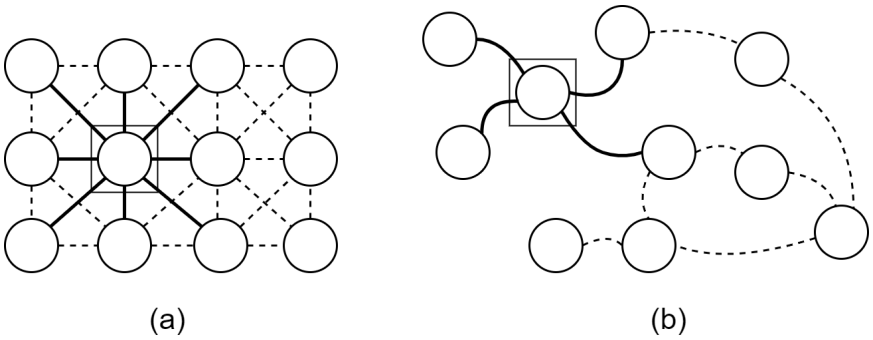

Figure 1. (a) Euclidean graph (grid graph) such as pixels of image, (b) arbitrary graph with irregular shape.

It is noteworthy that there exist some other variants of Laplacian matrix such as (Weiss, 1999; Ng et al., 2001):

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

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

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

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

2.2. Graph Fourier Transform

Consider the eigenvalue decomposition of the normalized Laplacian matrix (Ghojogh et al., 2019):

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

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 (Trigub & Belinsky, 2012).

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 the 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 $\boldsymbol{U}$ (Ghojogh et al., 2023a):

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

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

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

2.3. Graph Convolution

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

$$\begin{aligned}\boldsymbol{x} * \boldsymbol{g} &:= f^{-1}\big(f(\boldsymbol{x}) f(\boldsymbol{g})\big) \overset{(6)}{=} f^{-1}\big(\boldsymbol{U}^\top \boldsymbol{x} \boldsymbol{U}^\top \boldsymbol{g}\big) \\ &\overset{(7)}{=} \boldsymbol{U}\big(\boldsymbol{U}^\top \boldsymbol{x} \boldsymbol{U}^\top \boldsymbol{g}\big).\end{aligned} \tag{8}$$

We define:

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

Hence, the graph convolution, Eq. (8), can be stated as:

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

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:

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

3. ChebNet

Convolutional graph neural networks have been built upon two main approaches:

- spectral methods which have a graph signal processing perspective, and
- spatial methods which define graph convolution by information propagation.

Graph Convolutional Network (GCN) (Kipf & Welling, 2017) 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 is $\boldsymbol{H}^{(\ell)}$, then Eq. (11) becomes:

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

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

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

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 $\mathcal{O}(n^3)$ (Golub & Van Loan, 2013). Chebyshev Network (ChebNet) (Defferrard et al., 2016) improves the computational complexity of the convolutional neural network. It approximates the filter $\boldsymbol{g}$ by Chebyshev polynomials of the diagonal matrix of eigenvalues, i.e., $\boldsymbol{\Lambda}$.

The Chebyshev polynomials are (Mason & Handscomb, 2002):

$$\begin{aligned} &T_0(x) = 1, \\ &T_1(x) = x, \\ &T_i(x) = 2xT_{i-1}(x) - T_{i-2}(x). \end{aligned} \tag{14}$$

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

$$\cos(i\alpha) = T_i\big(\cos(\alpha)\big).$$

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

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

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:

$$\mathbb{R}^{n\times n} \ni \widetilde{\boldsymbol{\Lambda}} := \frac{2}{\lambda_{\max}}\boldsymbol{\Lambda} - \boldsymbol{I}_n, \tag{15}$$

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:

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

Recall Eq. (10):

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

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:

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

Similar to Eq. (15), we define:

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

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:

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

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

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

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

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

Comparing Eqs. (12) and (22) shows that ChebNet resolves the limitation of eigenvalue decomposition of the Laplacian. This is because it uses the approximation of Chebyshev polynomials and does not perform eigenvalue decomposition.

4. Graph Convolutional Network

Graph Convolutional Network (GCN) (Kipf & Welling, 2017) is the first-order approximation of ChebNet. In Eq. (22), GCN approximates the Chebyshev polynomials to its first order ($k = 1$):

$$T_i(\widetilde{\boldsymbol{L}}) \approx T_0(\widetilde{\boldsymbol{L}}) + T_1(\widetilde{\boldsymbol{L}}). \tag{23}$$

In other words:

$$\begin{aligned} \boldsymbol{x} * \boldsymbol{g} &\approx \sum_{i=0}^{1} \theta_i T_i(\widetilde{\boldsymbol{L}})\boldsymbol{x} = \theta_0 T_0(\widetilde{\boldsymbol{L}})\boldsymbol{x} + \theta_1 T_1(\widetilde{\boldsymbol{L}})\boldsymbol{x} \\ &\overset{(14)}{=} \theta_0 \boldsymbol{x} + \theta_1 \widetilde{\boldsymbol{L}}\boldsymbol{x}. \end{aligned}$$

More number of learnable parameters may result in overfitting (Ghojogh & Crowley, 2019). To reduce the number

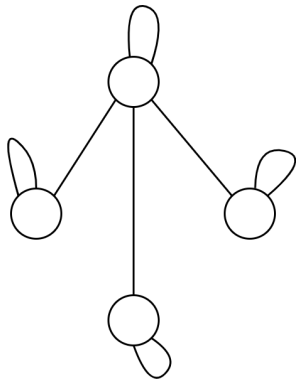

Figure 2. Self-loops of nodes of graph in GCN.

of parameters and to avoid overfitting, it is assumed that $\theta_0 = \theta_1 = \theta$, so:

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

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

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

It has been empirically observed that this results in some instability in training of GCN (Kipf & Welling, 2017). Therefore, an additional assumption is added to have self-loops on the nodes meaning that every node has an edge from it to itself (see Fig. 2).

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

$$\begin{aligned}&\widetilde{\boldsymbol{A}} := \boldsymbol{A} + \boldsymbol{I}, \\ &\widetilde{\boldsymbol{D}}(i,j) := \sum_{j=1}^{n}\widetilde{\boldsymbol{A}}(i,j), \\ &\bar{\boldsymbol{L}} := \widetilde{\boldsymbol{D}}^{-1/2}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-1/2}.\end{aligned} \tag{25}$$

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

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

In matrix form, if every row of $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ is the d-dimensional feature vector of a node, this equation becomes $\boldsymbol{x} * \boldsymbol{g} = \bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\theta}$ 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 $\boldsymbol{x} * \boldsymbol{g} = \bar{\boldsymbol{L}}\boldsymbol{X}\boldsymbol{\Theta}$ 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 is $\boldsymbol{H}^{(\ell)}$, then Eq. (11) becomes:

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

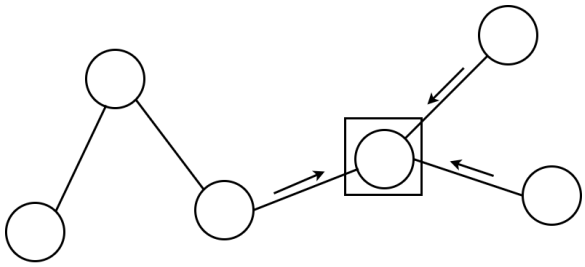

Figure 3. A node being impacted by its connected nodes before linear transformation in a graph neural network.

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

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):

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

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.

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

$$\boldsymbol{H}^{(\ell)} = \sigma(\boldsymbol{H}^{(\ell-1)}\boldsymbol{\Theta}). \tag{27}$$

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

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

Comparing Eqs. (27) and (28) shows the relation of GCN and feedforward neural 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 a 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 Fig. 3).

5. More General Frameworks of Graph Convolutional Network

5.1. The General Framework

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

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

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 $\widetilde{\boldsymbol{D}}^{-1}$:

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

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

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

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 mean pooling.

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

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

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

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

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).

5.2. Machine Learning Tasks in Graph Neural Networks

There exist different machine learning tasks in graph neural networks:

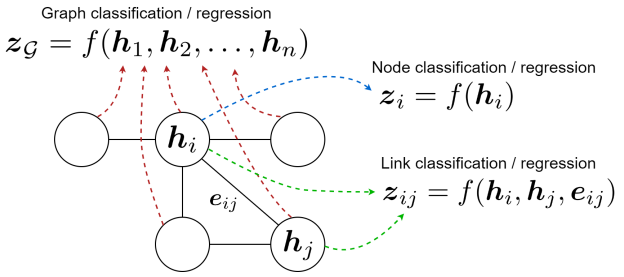

Figure 4. Use of features for machine learning tasks in graph neural networks

- 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 $\boldsymbol{e}_{ij}$ of the last layer are used in the loss function for the link classification or regression.

The use of features in these tasks is depicted in Fig. 4.

6. Graph Attention Network

6.1. Formulation of Graph Attention Network

As observed in Eqs. (29), (31), and (33), the linear combination in pooling can have weights. Graph Attention Network (GAT) (Veličković et al., 2017) adopts attention mechanisms to learn the relative weights between two connected nodes. In the pooling operation, the weights of attention are added:

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

where the attention weight α_{ij} measures the influence of node j on node i (Ghojogh & Ghodsi, 2020):

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

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

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

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

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

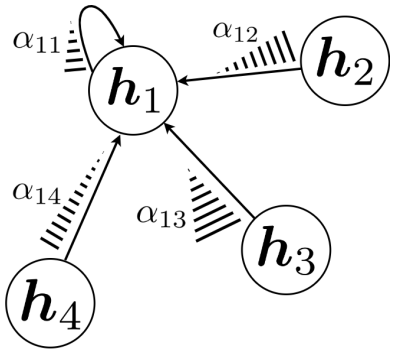

Figure 5. Illustration of attention of nodes in a graph.

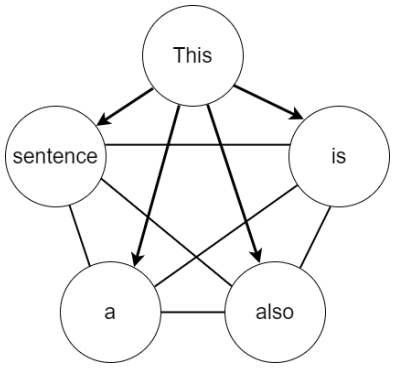

Figure 6. The graph for sentence "This is also a sentence". The arrows show the attention of the word "This" to other words of the sentence.

This attention function $a(.)$ can be a transformer autoencoder (Vaswani et al., 2017). 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:

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

where the summation in the denominator is over the neighbors of the i-th node. Figure 5 shows the attention of nodes in a graph.

It is noteworthy that transformers (Vaswani et al., 2017) are special cases of graph 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 illustrated in Fig. 6.

6.2. Comparison of Graph Attention Network and Transformer

In the following, GAT and transformer are compared. Firstly, on the one hand, in GAT, the attention is $a_{ij} = a(\boldsymbol{h}_i^{(\ell-1)}, \boldsymbol{h}_j^{(\ell-1)})$ where $\boldsymbol{h}_i^{(\ell-1)}$ and $\boldsymbol{h}_j^{(\ell-1)}$ are passed through a single-layer network with some weight matrix $\boldsymbol{W}$. 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 $\boldsymbol{k}_i = \boldsymbol{W}_K^\top \boldsymbol{x}$ (Ghojogh & Ghodsi, 2020). Therefore, the difference of GAT and transformer is that GAT uses the shared learnable weights for the query and key but transformer uses different learnable weights for them.

Secondly, 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 (Ghojogh & Ghodsi, 2020):

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

where p is the dimensionality of the query and the key.

Thirdly, the last difference of GAT and transformer is the softmax form. GAT sums over the neighbors in the denominator of a softmax form (see Eq. (38)). However, transformer sums over all tokens in the sequence:

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

7. Graph Autoencoder

Consider an autoencoder, shown in Fig. 7, where the encoder has two layers. This autoencoder accepts a graph as its input; hence, its name is Graph Autoencoder (GAE) (Kipf & Welling, 2016).

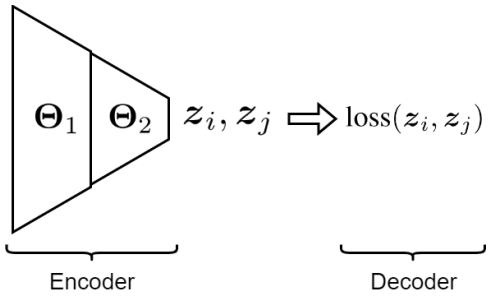

Figure 7. Graph reconstruction autoencoder

According to Eq. (26), the first layer of the encoder is:

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

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 (Nair & Hinton, 2010), and $\boldsymbol{H}^{(1)}$ is the output of the first layer.

Again, according to Eq. (26), the second layer of the encoder is:

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

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 $\text{GCN}(\boldsymbol{X}, \boldsymbol{A}; \boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2)$:

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

There are two types of GAE, i.e., graph reconstruction autoencoder and graph variational autoencoder (Kipf & Welling, 2016). These autoencoders are introduced in the following.

7.1. 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 $\boldsymbol{Z} \in \mathbb{R}^{n\times p}$, are obtained as:

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

The decoder of graph reconstruction autoencoder does not contain any layers but it models measuring similarity between the embedding vectors of the nodes (see Fig. 7). 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:

$$\widehat{\boldsymbol{A}} = \text{sigmoid}(\boldsymbol{Z}\boldsymbol{Z}^\top), \text{ or} \tag{42}$$

$$\widehat{\boldsymbol{A}}(i,j) = \frac{1}{1+e^{-\boldsymbol{z}_i^\top \boldsymbol{z}_j}}. \tag{43}$$

The graph reconstruction autoencoder is depicted in Fig. 7.

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

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

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 (Rumelhart et al., 1986).

7.2. Graph Variational Autoencoder

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

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

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

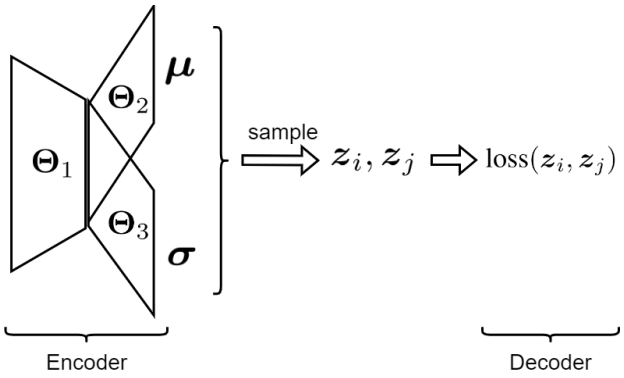

Figure 8. Graph variational autoencoder

where the first layer is shared between them as shown in Fig. 8.

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

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

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:

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

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

The decoder of the autoencoder models the following conditional distribution:

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

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 $\boldsymbol{z}_j$:

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

As a result, the decoder of the graph variational autoencoder does not contain any layers but models measuring similarity between the sampled latent variables in the latent space (see Fig. 8).

The graph variational autoencoder maximizes the Evidence Lower Bound (ELBO) in variational inference (Ghojogh et al., 2021):

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

where KL(.‖.) denotes the Kullback-Leibler (KL) divergence (Kullback & Leibler, 1951), $p(\boldsymbol{Z})$ 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 (Rumelhart et al., 1986). In backpropagation, the loss function should be minimized; therefore, the loss is the ELBO times -1:

$$\begin{aligned}\underset{\boldsymbol{\theta}}{\text{minimize}} \quad & -\mathbb{E}_{q(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{A})}\big[\log(p(\boldsymbol{A}\,|\,\boldsymbol{Z}))\big] \\ & + \mathrm{KL}\big(q(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{A})\|\,p(\boldsymbol{Z})\big). \qquad (52)\end{aligned}$$

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. It is noteworthy that graph autoencoder has been implemented in the Python programming language by the PyTorch Geometric library[1] (Fey & Lenssen, 2019).

8. Conclusion

This tutorial paper covered the theory of different variants of graph neural networks. First, graph Fourier transform and graph convolution were explained. Then, ChebNet was explained followed by graph convolutional network. General frameworks were also introduced for graph convolutional network. Thereafter, graph attention network was covered. Finally, two graph autoencoders, i.e., graph reconstruction autoencoder and graph variational autoencoder, were explained.

Acknowledgement

Some of the materials in this tutorial paper have been partially covered by Prof. Ali Ghodsi's videos (Data Science Courses) and Antonio Longa's videos on YouTube.

References

Cheung, Gene, Magli, Enrico, Tanaka, Yuichi, and Ng, Michael K. Graph spectral image processing. *Proceedings of the IEEE*, 106(5):907–930, 2018.

Defferrard, Michaël, Bresson, Xavier, and Vandergheynst, Pierre. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, volume 29, pp. 3844–3852, 2016.

Fey, Matthias and Lenssen, Jan Eric. Fast graph representation learning with PyTorch Geometric. In *International Conference on Learning Representations (ICLR), RLGM Workshop*, 2019.

[1] See https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html

Ghojogh, Benyamin and Crowley, Mark. The theory behind overfitting, cross validation, regularization, bagging, and boosting: tutorial. *arXiv preprint arXiv:1905.12787*, 2019.

Ghojogh, Benyamin and Ghodsi, Ali. Attention mechanism, transformers, BERT, and GPT: tutorial and survey. *Preprint*, 2020.

Ghojogh, Benyamin, Karray, Fakhri, and Crowley, Mark. Eigenvalue and generalized eigenvalue problems: Tutorial. *arXiv preprint arXiv:1903.11240*, 2019.

Ghojogh, Benyamin, Ghodsi, Ali, Karray, Fakhri, and Crowley, Mark. Factor analysis, probabilistic principal component analysis, variational inference, and variational autoencoder: Tutorial and survey. *arXiv preprint arXiv:2101.00734*, 2021.

Ghojogh, Benyamin, Crowley, Mark, Karray, Fakhri, and Ghodsi, Ali. Background on linear algebra. *Elements of Dimensionality Reduction and Manifold Learning*, pp. 17–41, 2023a.

Ghojogh, Benyamin, Crowley, Mark, Karray, Fakhri, and Ghodsi, Ali. Laplacian-based dimensionality reduction. In *Elements of Dimensionality Reduction and Manifold Learning*, pp. 249–284. Springer, 2023b.

Golub, Gene H and Van Loan, Charles F. *Matrix computations*. JHU press, 2013.

Kipf, Thomas N and Welling, Max. Variational graph autoencoders. *arXiv preprint arXiv:1611.07308*, 2016.

Kipf, Thomas N and Welling, Max. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Koncel-Kedziorski, Rik, Bekal, Dhanush, Luan, Yi, Lapata, Mirella, and Hajishirzi, Hannaneh. Text generation from knowledge graphs with graph transformers. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

Kullback, Solomon and Leibler, Richard A. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Mason, John C and Handscomb, David C. *Chebyshev polynomials*. Chapman and Hall/CRC, 2002.

Myung, Yoochan, Pires, Douglas EV, and Ascher, David B. CSM-AB: graph-based antibody–antigen binding affinity prediction and docking scoring function. *Bioinformatics*, 38(4):1141–1143, 2022.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

Ng, Andrew, Jordan, Michael, and Weiss, Yair. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14:849–856, 2001.

Pittala, Srivamshi and Bailey-Kellogg, Chris. Learning context-aware structural representations to predict antigen and antibody binding interfaces. *Bioinformatics*, 36 (13):3996–4003, 2020.

Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Sudderth, Erik B and Freeman, William T. Signal and image processing with belief propagation [DSP applications]. *IEEE Signal Processing Magazine*, 25(2):114–141, 2008.

Trigub, Roald M and Belinsky, Eduard S. *Fourier analysis and approximation of functions*. Springer Science & Business Media, 2012.

Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Łukasz, and Polosukhin, Illia. Attention is all you need. In *Advances in neural information processing systems*, volume 30, 2017.

Veličković, Petar, Cucurull, Guillem, Casanova, Arantxa, Romero, Adriana, Lio, Pietro, and Bengio, Yoshua. Graph attention networks. In *International Conference on Learning Representations*, 2017.

Wang, Shoujin, Hu, Liang, Wang, Yan, He, Xiangnan, Sheng, Quan Z, Orgun, Mehmet A, Cao, Longbing, Ricci, Francesco, and Yu, Philip S. Graph learning based recommender systems: A review. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

Weiss, Yair. Segmentation using eigenvectors: a unifying view. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pp. 975–982. IEEE, 1999.

Zhang, Zuobai, Xu, Minghao, Jamasb, Arian, Chenthamarakshan, Vijil, Lozano, Aurelie, Das, Payel, and Tang, Jian. Protein representation learning by geometric structure pretraining. In *International Conference on Learning Representations (ICLR)*, 2023.