A survey on graph embeddings

Andrew Liu, Yuhan Xia, You Zhou, Shuai Shao

December 2022

1 Introduction

Graphs are a ubiquitous domain for modeling real-world data and research in graph ML has recently witnessed an explosion in popularity. In this survey, we conduct a study of the field of graph embeddings, focusing on matrix factorization/spectral methods, sequence-based approaches, hyperbolic embeddings, and most-recent graph neural network (GNN) approaches. Unlike other surveys about graph embeddings, our work explores both the state-of-the-art and provides an in-depth explanation for key theoretical results.

2 Matrix-factorization based methods

Matrix factorization is a matrix decomposition method that factorize a matrix to the product of matrices. It is widely used in many machine learning areas for dimension reduction and it is also applied in graph embedding methods. Here the matrices used to represent the graph could be adjacent matrix, graph Laplacian matrix, node transition probability matrix, Katz similarity matrix, or other similarity matrices. Most early studies in graph embedding are based on this approach.

2.1 Linear graph embedding

To utilize matrix factorization for graph embedding, a straightforward idea is to use classical linear models such as Multidimensional Scaling (MDS) [27], Principal Component Analysis (PCA) [47] and Linear Discriminate Analysis (LDA) [29] to factorize the graph matrix. Specifically, the pioneer study MDS directly decompose the graph adjacent matrix A through minimizing the mean squared error (MSE) between a_{ij} and the Euclidean distance between two feature vectors X_i and X_j on the manifold to find the optimal embedding.

Later, latent semantic indexing LSI [12] utilized Singular Value Decomposition (SVD) [19] for matrix factorization to achieve automatic indexing and retrieval. Specifically, for any rectangular matrix, X, LSI decomposes it into the product of three other matrices:

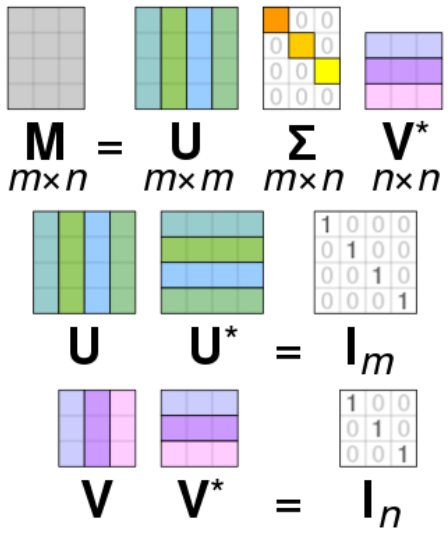


Figure 1: A simple illustration for SVD

$$X = T_0 S_0 D_0^{'}$$

Then it preserves the top k singular values and corresponding rows and columns in T_0 and D_0 to reach an adjustable representational richness. So the result is a reduced model:

$$X \approx \hat{X} = TSD^{'}$$

which is a rank-k model with the best possible least-squares-fit to X that we use to approximate the data.

Other linear decomposition models, such as PCA and LDA, are also applied in this area. PCA conduct linearly transformation and try to finds new axes that maximize the variance in the data. LDA tries to maximize the inter-class variance while minimize the intra-class variance to get a desired embedding.

However, these linear embedding models have some problems. One of the problems is that the low dimension embeddings are related to the input vectors through a linear transformation, which mean that it only performs well when the initial data lies on, or near, a low dimensional manifold. When the data does not follow the distribution of the low dimensional manifold, these models are hard to find a good embedding and do not perform well.

Another problem is that these models did not consider the neighborhood relationships of nodes. They regard all training pairs (i to j) as connected. And this will lose the information of graph connection property. To improve this, the follow-up studies first construct a K-Nearest-Neighbor (KNN) graph from the nodes, where each node is only connected to its top k similar neighbors. Then, researchers utilize different methods to calculate the similarity matrix.

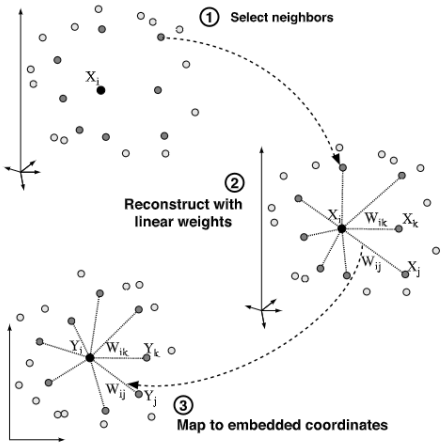


Figure 2: Illustration for LLE procedure

2.2 Nonlinear transformation

To get a better low dimensional representation of the graphs, more and more nonlinear graph embedding methods were proposed in recent years. Here we will introduce some of the popular works in this area.

2.2.1 Locally Linear Embedding

Locally Linear Embedding (LLE) [39] model improves the estimation of pairwise distances in MDS. It recovers the global nonlinear structure from locally linear fits.

Specifically, the LLE algorithm expect each data point and its neighbors to lie on, or close to a local linear patch of the manifold. Then, based on the local linear geometry of these patches, the algorithm can reconstruct each data point from its neighbors with linear coefficients. The reconstruction error can be measured by:

$$\varepsilon(W) = \sum_{\mathrm{i}} \left| \vec{X}_{\mathrm{i}} - \Sigma_{\mathrm{j}} W_{\mathrm{ij}} \vec{X}_{\mathrm{j}} \right|^2$$

which is the sum of the squared distances between all the data points and their reconstructions.

LLE first compute the weight W_{ij} for data reconstruction. The weights W_{ij} represents the contribution of the j-th data to the i-th data reconstruction. Since LLE forces that each data should be reconstructed from its neighbors, the neighborhood relationship of the graph in the weights W_{ij} could be preserved by setting $W_{ij} = 0$ if X_j does not belong to the set of neighbors of X_i. Meanwhile, the sum of weights should follow $\Sigma_{\mathrm{j}} W_{\mathrm{ij}} = 1$.

It is easy to notice that the reconstruction weights W_{ij} reflect the intrinsic geometric properties of the data. And LLE hope to find embeddings which still follow the geometric relationship. In particular, the embedded data should still be available to be reconstructed with the same weights W_{ij}.

In the end, each high dimensional input X_i will be mapped to a low dimensional vector Y_i by choosing d-dimensional coordinates Y_i to minimize the embedding cost function:

$$\Phi(Y) = \sum_{\mathrm{i}} \left| \vec{Y_{\mathrm{i}}} - \Sigma_{\mathrm{j}} W_{\mathrm{ij}} \vec{Y_{\mathrm{j}}} \right|^2$$

Then we can get a neighborhood-preserving mapping with LLE.

2.2.2 Isomap

The Isomap algorithm [43] is built on classical MDS method, but tries to preserve the neighborhood relationship as well. It approximate the distance to faraway points by adding up a sequence of "short hops" between neighboring points.

Specifically, the Isomap takes the distances $d_X(i,j)$ between nodes i, j as input. Isomap first determines each points' neighbors on manifold M. There are two types of Isomap: $\epsilon - Isomap$ and $K - Isomap$. $\epsilon - Isomap$ defines neighbors based on a maximum distance, for each node i, if node j and i is closer thanϵ, then they are regarded as node i's neighbor. For $K - Isomap$, j is picked with the K nearest neighbors. After find the neighbors for each node, connect them and their neighbors with edge of length $d_X(i,j)$, and set others as infinite.

Then, Isomap tries to estimate the geodesic distances $d_M(i,j)$ between point pairs on manifold M.

It computes their shortest path distances $d_G(i,j)$ by updating

$$d_G(i,j) = \min\left\{d_G(i,j), d_G(i,k) + d_G(k,j)\right\}$$

for all $k = 1, 2, ..., N$

Finally, Isomap applies classical MDS to the matrix of graph distances $D_G = d_G(i,j)$, to compute an embedding of the data in a d-dimensional Euclidean space Y by minimizing the cost function

$$E = \left\|\tau\left(D_G\right) - \tau\left(D_Y\right)\right\|_{L^2}$$

where D_Y denotes the matrix of Euclidean distances $\{d_Y(i,j) = \|\mathbf{y}_i - \mathbf{y}_j\|\}$ and $\|A\|_{L^2}$ is the L_2 matrix norm $\sqrt{\Sigma_{i,j} A_{ij}^2}$. The τ operator computes the inner products. Then we can get the embedding by setting the coordinates $\mathbf{y}_i$ to the top d eigenvectors of the matrix $\tau\left(D_G\right)$ (13).

Both Isomap and LLE introduces the information of neighborhood and preserves more information about graph, and achieve good result for nonlinear graph embeddings.

2.2.3 Laplaican Eigenmaps

The Laplaican Eigenmaps method [5] is similar with LLE and Isomap. First, it construct the graph with the same approach as Isomap. Then, it utilizes two ways for weighting the edges: using heat kernel

$$W_{ij} = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{t}}$$

for the edge of connected nodes i and j. Or set $W_{ij} = 1$ for the edge of connected nodes i and j. The latter setting could avoid the necessity of chossing t.

Finally, compute the eigenvalues and eigenvectors for the generalized eigenvector problem

$$L\mathbf{y} = \lambda D\mathbf{y}$$

where D is diagonal weight matrix, its entries are column summation of the edge's weight W: $D_{ii} = \sum_j W_{ji}$. And $L = D - W$ is the graph Laplacian matrix. It is a symmetric and positive semidefinite matrix.

Let $\mathbf{y}_0, \ldots, \mathbf{y}_{k-1}$ be the solutions of the eigenvalue decomposition with ascent order. Then the embedding into the lower dimensional space $\mathbb{R}^m$ is given by $(\mathbf{y}_1(i), \ldots, \mathbf{y}_m(i))$

In fact, Laplaican Eigenmaps is trying to minimize

$$\begin{aligned}\phi(Y) &= \frac{1}{2}\sum_{i,j} |Y_i - Y_j|^2 W_{ij} \\ &= \operatorname{tr}\left(Y^T L Y\right),\end{aligned}$$

where L is the Laplacian of graph G.

2.2.4 Graph Factorization

Graph Factorization [2] is a framework for large-scale graph decomposition and inference. Given a graph adjacency matrix G, it factorizes the matrix G through minimizing

$$f(Y, Z, \lambda) = \frac{1}{2}\sum_{(i,j)\in E} \left(Y_{ij} - \langle Z_i, Z_j\rangle\right)^2 + \frac{\lambda}{2}\sum_i \|Z_i\|^2$$

where $Y \in \mathbb{R}^{n\times n}$ and Y_{ij} is the weight on edge between i and j. $Z \in \mathbb{R}^{n\times r}$ is the factor matrix we aim to find and Z_i is the factor vector for node i in matrix Z. Here n is the number of nodes, λ is a regularization coefficient, and r is a given number for factorization that $r \ll n$. Our goal is to find a Z that ZZ^T is close to Y.

To the best of our knowledge, Graph Factorization is the first graph embedding method to obtain a complexity that is complexityis linear in the number of iterations and edges m in G, which is $O(|E|d)$, while the previous mentioned methods have a complexity of $O(|E|d^2)$.

2.2.5 GraRep

GraRep [7] employs a noise contrastive estimation (NCE) [21] to define the objective function. Specifically, it first define a non-negative adjacency matrix S for a graph by setting $S_{i,j} = 1$ if there exists an edge from v_i to v_j and $S_{i,j} = 0$ otherwise. For weighted graphs $S_{i,j}$ is equal to the weight of the edges. Then, it computes the degree matrix D for the graph with adjacency matrix S :

$$D_{ij} = \begin{cases} \sum_p S_{ip}, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}$$

After that, it calculates the following (1-step) probability transition matrix A

$$A = D^{-1}S$$

where $A_{i,j}$ is the transition probability from v_i to v_j in one step. We can find that the A matrix can be considered as a re-scaled S matrix whose rows are normalized.

Then, for each k from 1 to the maximum transition step K, it computes the positive log probability matrix by

$$X_{i,j}^k = \log\left(\frac{A_{i,j}^k}{\sum_t A_{t,j}^k}\right) - \log(\beta)$$

and conduct SVD for X to get W

$$\left[U^k, \Sigma^k, \left(V^k\right)^T\right] = SVD\left(X^k\right)$$
$$W^k = U_d^k \left(\Sigma_d^k\right)^{\frac{1}{2}}$$

The concatenated W for each of the k-step representation is the output matrix representation. Since A contains the global information of the graph, we can say that we get a graph representations with global structural information. However, a problem of GraRep is that it suffers from a high time complexity $O(|V|3)$ due to the computation of the power of a matrix and the SVD.

2.2.6 High-Order Proximity preserved Embedding

High-Order Proximity preserved Embedding (HOPE) [33] focuses on preserving high-order proximities of large scale graphs and it is capable of capturing the asymmetric transitivity. Specifically, it tries to minimize a L2-norm of the gap between similarity matrix and as the loss function:

$$\min \left\| \mathbf{S} - \mathbf{U}^s \cdot \mathbf{U}^{t^\top} \right\|_F^2$$

It finds that many high-order proximity measurements in graph can reflect the asymmetric transitivity. And they represents the similarity matrix S as:

$$\mathbf{S} = \mathbf{M}_g^{-1} \cdot \mathbf{M}_l$$

where $\mathbf{M}_g$ and $\mathbf{M}_l$ are both polynomial of matrices.

HOPE utilized different types of $\mathbf{S}$ such as Katz Index [25], Rooted Page Rank (RPR), Common Neighbors (CN) and Adamic-Adar (AA) for approximation, and conduct generalized SVD with $\mathbf{M}_g$ and $\mathbf{M}_l$ to get the singular value and corresponding singular vectors.

$$\mathbf{S} = \sum_{i=1}^{N} \sigma_i \mathbf{v}_i^s \mathbf{v}_i^{t^\top}$$

And finally, we can get the optimal embedding vectors as:

$$\mathbf{U}^s = [\sqrt{\sigma_1} \cdot \mathbf{v}_1^s, \cdots, \sqrt{\sigma_K} \cdot \mathbf{v}_K^s]$$
$$\mathbf{U}^t = \left[\sqrt{\sigma_1} \cdot \mathbf{v}_1^t, \cdots, \sqrt{\sigma_K} \cdot \mathbf{v}_K^t\right]$$

The time complexity of HOPE is $O\left(m \cdot K^2 \cdot L\right)$, where m is the number of edges, K is the embedding dimension and L is the iteration number. So this model complexity is linear with the volume of data, which means that it is scalable for large scale graphs.

2.3 Conclusion

Matrix-factorization-based graph embedding methods achieved a great success in early years. They are easy to implement and with high interpretability. However, they are still suffered from high computational complexity for large graphs and most of them only capture a small-order proximity. These makes those approaches hard to be widely used in the era of big data. But their idea and mathematical models are still worthy for every researcher to study.

3 Sequence-based models

3.1 Graph Embedding paradigm

Given an unweighted graph $G = (V, E)$, its adjacency matrix $\mathbf{A} \in \{0, 1\}^{|V| \times |V|}$ may be built using $A_{vu} = \mathbb{1}[(v, u) \in E]$, where $\mathbb{1}[.]$ is an indicator function that evaluates to 1 iff boolean argument is true. In general, as defined in [1] graph embedding approaches aim to achieve the following goal:

$$\min_{\mathbf{Y}} \mathcal{L}(f(\mathbf{A}), g(\mathbf{Y}))$$

where $\mathbf{Y} \in \mathbb{R}^{|V| \times d}$ represents a d-dimensional node embedding dictionar; $f : \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^{|V| \times |V|}$ marks a transformation of the adjacency matrix; $g : \mathbb{R}^{|V| \times d} \rightarrow \mathbb{R}^{|V| \times |V|}$ is a pairwise edge function; and $\mathcal{L} : \mathbb{R}^{|V| \times |V|} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}$ is a loss function.

3.2 Markov-chain model of Random Walk

For an edge connecting vertexes v_i and v_j, a weight w_{ij} exists to represent how important the relation between v_i and v_j is. Elements of adjacency matrix A are set to the weight as $a_{ij} = w_{ij}$. Random Walk is defined as a Markov-chain describing the sequence of nodes visted. Denote $s(t)$ as the current random state of the markov chain at time t, the value of it is assigned as the node number. Then traversing from the current state $s(t) = i$ to $s(t+1) = j$, the probability is defined as $p_{ij} = P(s(t+1) = j|s(t) = i) = \frac{a_{ij}}{a_i}$ where $z_i = \sum_{j=1}^{n} a_{ij}$.

Denoting the probability of being at state i at time t as $\pi_i(t) = P(s(t) = i)$ and $\mathbf{P}$ as transition matrix with $p_{ij} = P(s(t+1) = j|s(t) = i)$. Then once the initial distribution $\pi(0) = \pi^0$ is known, Markov-chain evolution is calculated $\pi(t+1) = \mathbf{P}^\top \pi(t)$ where $\pi(t) = [\pi_1(t), \pi_2(t), ..., \pi_n(t)]^\top$.

3.3 Quality of interest

Random walk is original used to characterize the similarity between nodes of a weighted and undirected graph. Based on Markov-chain, the similarity is calculated through a set of quality of interest defined in [13]:

1. **Average first-passage time**: $m(k|i)$ is the average number of steps required by a random walker to reach state k for the first time after beginning in state i. Denote $T_{ik} = \min(t \geq 0 \mid s(t) = k \text{ and } s(0) = i)$ as the shortest time between beginning from vertex v_i and hitting v_k.

   $$m(k|i) = E[T_{ik}|s(0) = i]$$

   which can be calculated recurrently

   $$\begin{cases} m(k|k) = 0 \\ m(k|i) = 1 + \sum_{j=1}^{n} p_{ij} m(k|j), \quad \text{for} \quad i \neq k, \end{cases}$$

2. **Average commute time**: $n(i,j) = m(j|i) + m(i|j)$ is defined as the average number of steps it takes for a random walker to traverse from vertex v_i to v_j. This may be calculated as the distance between any two states, and so is a graph metric using the following reasoning:

   (a) $n(i,j) \geq 0$,

   (b) $n(i,j) = 0$ if and only if $i = j$,

   (c) $n(i,j) = n(j,i)$,

   (d) $n(i,j) \leq n(i,k) + n(k,j)$.

   With the definition of pseudoinverse of the Laplacian matrix ($\mathbf{L}^+$) and volume of the graph $V_G = \sum_{k=1}^{n} d_{kk}$, $n(i,j)$ can be written as

   $$n(i,j) = V_G \left(l_{ii}^+ + l_{jj}^+ - 2l_{ij}^+\right)$$

in matrix form it is

$$n(i,j) = V_G \left(\mathbf{e}_i - \mathbf{e}_j\right)^{\mathrm{T}} \mathbf{L}^{+} \left(\mathbf{e}_i - \mathbf{e}_j\right),$$

where $\mathbf{e}_i$ is defined as

$$\mathbf{e}_i = \left[\underset{1}{0}, \ldots, \underset{i-1}{0}, \underset{i}{1}, \underset{i+1}{0}, \ldots, \underset{n}{0}\right]^{\mathrm{T}}$$

3. **Average first-passage cost**: $o(k|i)$, is the factor used to generalize average first-passage time by costing each transition. When $c(j|i) = 1$ for all i, j, $m(k|i)$ is a special case of $o(k|i)$. As a result, $o(k|i)$ may be calculated as

$$\begin{cases} o(k|k) = 0 \\ o(k|i) = \sum_{j=1}^{n} p_{ij} c(j|i) + \sum_{j=1}^{n} p_{ij} o(k|j), \quad \text{for } i \neq k. \end{cases}$$

4. **Pseudoinverse of the Laplacian matrix**: $\mathbf{L}^{+}$ the Moore-Penrose pseudoinverse of Laplacian matrix L, whose elements are the inner products of the node vectors in Euclidean space while retaining the Euclidean Commute Time Distance(ECTD). $\mathbf{L}^{+}$ is also a Gram matrix and hence a valid kernel. Specifically, $\mathbf{L}^{+}$ can be calculated with the formula

$$\mathbf{L}^{+} = \left(\mathbf{L} - \mathbf{e}\mathbf{e}^{\mathrm{T}}/n\right)^{-1} + \mathbf{e}\mathbf{e}^{\mathrm{T}}/n$$

5. **Euclidean Commute Time Distance**(ECTD): $[n(i,j)]^{1/2}$, which is also a distance because $\mathbf{L}^{+}$ is semi-deinite and is also Euclidean.

The attributes listed above give similarity metrics between any node in a linked network. Experiment findings also showed that inner-product-based quantities, particularly the pseudoinverse of the Laplacian matrix, outperform and produce more reliable results than typical scoring methods. However, problems exist as the approach does not scale well for huge datasets and lacks generality for directed graphs.

3.4 Random Walk based betweenness centrality

Betweenness is a measure of a node's centrality in the network, or the extent to which a vertex is located on the paths that connect other nodes. Betweenness may also be regarded of as a measure of an anchor's effect on information travelling between others. Simply betweenness [14] and flow betweenness [15] are two previously suggested measuring approaches.

The simply betweenness of vertex i is defined as the proportion of shortest routes in a network between pairs of vertices that travel through i.

$$b_i = \frac{\sum_{s<t} g_i^{(st)}/n_{st}}{(1/2)n(n-1)}$$

where $g_i^{(st)}$ represents the number of geodesic pathways that pass through i from vertex s to vertex t; n_{st} is the total number of geodesic paths from s to t and n is the number of vertices in the network.

The flow betweenness of vertex i is defined as the amount of flow sent from s to t when the maximum flow is transmitted, averaged over all s and t.

As defined in [31], the random walk betweenness of vertex i is intuitively equivalent to the number of times a random walk starting at s and finishing at t passes through i averaged across all s and t. This new method is formally defined with the current flow analogy.

3.4.1 Current Flow Analogy

The current flow betweenness of vertex i is defined to be the amount of current that flows through i averaged through all and t.

Let V_i be the voltage at vertex i, A is the adjacency matrix, and δ_{ij} is the Kronecker δ

$$\delta_{ij} = \begin{cases} 1, \text{ if } i = j \\ 0, \text{ otherwise} \end{cases}$$

$$\sum_j A_{ij} \left(V_i - V_j\right) = \delta_{is} - \delta_{it}$$

is derived because of the Kiirchhoff's law of current conservation. This equation can be written as

$$(\boldsymbol{D} - \boldsymbol{A}) \cdot \boldsymbol{V} = \boldsymbol{s}$$

where k_i denotes the degree of v_i and $\mathbf{D}$ is formed as a diagonal matrix with $D_{ii} = k_i$; $\mathbf{s}$ is the source vector, where

$$s_i = \begin{cases} 1, \text{ for } i = s \\ -1, \text{ for } i = t \\ 0, \text{ otherwise} \end{cases}$$

Since the $\mathbf{D} - \mathbf{A}$ is a graph laplacian and is singular, remove the vth equation in accordance to vertex v that voltage relate to, and the output is noted as $\mathbf{D}_v - \mathbf{A}_v$, and V is calculated:

$$\boldsymbol{V} = \left(\boldsymbol{D}_v - \boldsymbol{A}_v\right)^{-1} \cdot \boldsymbol{s}$$

Add 0 back to $\left(\boldsymbol{D}_v - \boldsymbol{A}_v\right)^{-1}$ and denote the result as $\mathbf{T}$.

$$V_i^{(st)} = T_{is} - T_{it}$$

$V_i^{(st)}$ means the voltage at vertex v_i for source s and target t, it can be written in the form of T as above. Taking half of sum of the absolute values of the currents flowing on v_i, the current on v_i is gained.

$$I_i^{(st)} = \frac{1}{2} \sum_j A_{ij} \left| V_i^{(st)} - V_j^{(st)} \right| = \frac{1}{2} \sum_j A_{ij} \left| T_{is} - T_{it} - T_{js} + T_{jt} \right|, \quad \text{for } i \neq s, t$$

Also, the current on start vertex s and end vertex t are defined as 1. Finally the betweenness is calculated as the average on the current flow over all s-t pairs:

$$b_i = \frac{\sum_{s<t} I_i^{(st)}}{(1/2)n(n-1)}$$

3.4.2 Random Walk betweenness

The betweenness of vertex i is defined to be the net number of times a walk goes through i, where net indicates two walks that pass through the same vertex from different directions are canceled out and do not contribute to betweenness. Taking Absorbing Random Walk into account, the probability of being at v_i on the next step given now on v_j is:

$$M_{ij} = \frac{A_{ij}}{k_j}, \quad \text{for } j \neq t$$

So,

$$\boldsymbol{M} = \boldsymbol{A} \cdot \boldsymbol{D}^{-1}$$

with the same notation as the present betweenness. It is worth noting that, according to the absorbing random walk feature, $\mathbb{M}_{it} = 0$ for each i. As previously stated, column and row t of the matrix are removed to produce $\boldsymbol{M}_t = \boldsymbol{A}_t \cdot \boldsymbol{D}_t^{-1}$. So, if signifies $[\boldsymbol{M}_t^r]_{js}$ as the probability of being at v_j after r steps from the initial position s, then the likelihood of being on an adjacent vertex v_i on the following step is $k_j^{-1}[\boldsymbol{M}_t^r]_{js}$. And the average chance of switching from v_j to v_i is $k_j^{-1}\left[(\boldsymbol{I} - \boldsymbol{M}_t)^{-1}\right]_{js}$, which may be expressed as vectors:

$$\boldsymbol{V} = \boldsymbol{D}_t^{-1} \cdot (\boldsymbol{I} - \boldsymbol{M}_t)^{-1} \cdot \boldsymbol{s} = (\boldsymbol{D}_t - \boldsymbol{A}_t)^{-1} \cdot \boldsymbol{s}$$

With the equation above, $V_i - V_j$ is the net flow of random walk along the edge from j to i, and the net flow if random walks through v_i is the same with Eq.(1), that is

$$b_i = \frac{\sum_{s<t} \sum_j A_{ij} \mid V_i^{(st)} - V_j^{(st)} \mid}{n(n-1)}$$

The Random Walk betweenness overcomes the problem that the preceding definition of betweenness is unrealistic, in that only a fraction of all potential pathways are computed under optimality assumptions. The metric is especially effective for locating high-centrality vertices that do not happen to be on geodesic lines or paths produced by maximum-flow cut-sets.

3.5 Two-step learning method

The two-step method for embedding learning is first to do random walk co-occurrence sampling, and then followed by node representation learning.

Random walks along E begin at a random node $v_0 \in \text{sample}(V)$, and sample an edge repeatedly to transition to the next node as $v_{i+1} := \text{sample}(\mathcal{N}[v_i])$.

Here $\mathcal{N}[v_i]$ is a set containing outgoing edges from v_i. Then, the random walks $v_0 \to v_1 \to v_2 \to ..$ are fed into word2vec algorithm, which learns embeddings by randomly selecting each node v_i from the sequence. As a result, the embedding representation of this anchor node v_i is closer to the embeddings of its context neighbors $\{v_{i+1}, v_{i+2}, \ldots, v_{i+c}\}$. In practice, the size of the context window c is chosen from a distribution such as uniform $\mathcal{U}\{1, C\}$.

Let $\mathbf{D} \in \mathbb{R}^{|V|\times|V|}$ be the random walks co-occurrence matrix with each entry D_{vu} containing the number of times nodes v and u are co-visited within context distance $c \sim \mathcal{U}\{1, C\}$ in all simulated random walks.

The random walks-based graph embedding approach may also be used in conjunction with the broader graph embedding framework with $f(\mathbf{A}) = \mathbf{D}$, $g(\mathbf{Y}) = \mathbf{Y} \times \mathbf{Y}^\top$ and loss function

$$\min_{\mathbf{Y}} \left[\log Z - \sum_{v,u \in V} D_{vu} \left(Y_v^\top Y_u \right) \right]$$

$Z = \sum_{v,u} \exp\left(Y_v^\top Y_u\right)$ denotes partition function, it can be estimated with negative sampling.

3.6 DeepWalk

DeepWalk [36] learns the social representation of a graph's vertices by modeling a sequence of brief random walks. Furthermore, the model learns structural regularities found in short random walks.

3.6.1 Language Modeling

Because identifying associations in a social network is akin to predicting letter sequences in NLP, the language model is refined and used to graph vertex embedding.

The language model predicts the likelihood of a certain word sequence $W_1^n = (w_0, w_1, ..., w_n)$ appearing in a corpus and maximize $\Pr(w_n | w_0, w_1, ..., w_{n-1})$. Consider random walk as a word with a set length, with mapping function $\Phi : v \in V \mapsto \mathbb{R}^{|V|\times d}$ that gain latent representation of vertex v, estimate likelihood:

$$\Pr\left(v_i \mid \left(\Phi\left(v_1\right), \Phi\left(v_2\right), \cdots, \Phi\left(v_{i-1}\right)\right)\right)$$

To minimize training parameters, the model must instead maximize the likelihood of any word occurring in the context without knowing its offset from the provided word. The objection function of DeepWalk is as below:

$$\underset{\Phi}{\text{minimize}} - \log \Pr\left(\{v_{i-w}, \cdots, v_{i-1}, v_{i+1}, \cdots, v_{i+w}\} \mid \Phi\left(v_i\right)\right)$$

3.6.2 DeepWalk Algorithm

The algorithm is carried out in two phases. First, until the maximum length t is achieved, a random walk generator samples uniformly a root vertex v_i from

V and samples neighbors of the last vertex visited. To explore the vertices, a random ordering is established at the start of each walk. The total number of times a random walk must be started is γ, and iteration over all vertices is required in the inner loop. SkipGram is used to update parameters based on the goal function.

Algorithm 1 DEEPWALK(G, w, d, γ, t)

Input: graph $G(V, E)$
w: window size
d: embedding size
γ: walks per vertex
t: walk length
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$
Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
Build a binary Tree T from V
for $i = 0$ to γ **do**
  $\mathcal{O} = \text{Shuffle}(V)$
  for $v_i \in \mathcal{O}$ **do**
    $\mathcal{W}_{v_i} = \text{RandomWalk}(G, v_i, \text{t})$
    $\text{SkipGram}(\Phi, \mathcal{W}_{v_i}, w)$
  end for
end for

3.6.3 SkipGram Algorithm

SkipGram optimizes the likelihood of v_i's neighbors in the walk by maximizing the co-occurrence probability among entities in a window.

Algorithm 2 SkipGram($\phi, \mathcal{W}_{v_i}, w$)

for $v_j \in \mathcal{W}_{v_i}$ **do**
  for $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ **do**
    $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
    $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
  end for
end for

It has been demonstrated that when DeepWalk is executed utilizing several threads, it is highly scalable, parallelizable, and efficient in very large scale machine learning. However, the approach is still constrained by the efficiency of the downstream word embbedding model, the selection of random walk hyperparameters, and the requirement for an extensive theoretical proof.

3.7 DeepWalk with Graph attention

Different from previous methods, DeepWalk with graph attention proposed in [1] includes attention parameters in training loss to guide random walk sampling focus on the most helpful data.

Moreover, because hyper-parameter settings have a substantial impact on the effectiveness of unsupervised graph embedding, for example, the duration of random walks and the training window length in word2vec, DeepWalk with Graph attention method also aims to replace fixed hyper-parameters with trainable parameters.

According to the graph embbeding objectives framework, the method sets $g(\mathbf{Y}) = g([\mathbf{L} \mid \mathbf{R}]) = \mathbf{L} \times \mathbf{R}^\top$ with respect to SVD and $f(\mathbf{A}) = \mathbb{E}[\mathbf{D}]$.

3.7.1 Expectation on the co-occurrence matrix

$\mathbb{E}(\mathbf{D})$ above denotes the expectation on co-occurrence matrix generated from random walk simulations, which helps join the two steps of Random Walk. Since $\mathbb{E}(\mathbf{D})$ is formulated directly without knowing $\mathbf{D}$ (obtained through random walk simulation and co-occurrence sampling), sampling parameters are tuned naturally.

Let $\mathcal{T}$ be the graph G's transition matrix; it may be constructed by normalizing rows of $\mathbf{A}$:

$$\mathcal{T} = \operatorname{diag}\left(\mathbf{A} \times \mathbf{1_n}\right)^{-1} \times \mathbf{A}$$

Given a random surfer's starting probability distribution $p^{(0)} \in \mathbb{R}^{|V|}$, the surfer's distribution after k steps may be represented using $\mathcal{T}$, as $p^{(k)} = {p^{(0)}}^\top (\mathcal{T})^k$. The expectation on $\mathbf{D} \in \mathbb{R}^{|V| \times |V|}$ is written as:

$$\mathbb{E}\left[\mathbf{D}^{\text{DEEPWALK}}; C\right] = \tilde{\mathbf{P}}^{(0)} \sum_{k=1}^{C} \left[1 - \frac{k-1}{C}\right] (\mathcal{T})^k$$

where C indicates the upper bound of random walk length, $c_i \sim \mathcal{U}\{1, C\}$ is sampled individually for each anchor v_i; and $\tilde{\mathbf{P}}^{(0)} \in \mathbb{R}^{|V| \times |V|}$ is a diagonal matrix with the number of walks beginning at node v_i as the ith element on the diagonal. The expectation expression is as below when running GloVe [35] embedding algorithm:

$$\mathbb{E}\left[\mathbf{D}^{\text{GloVe}}; C\right] = \tilde{\mathbf{P}}^{(0)} \sum_{k=1}^{C} \left[\frac{1}{k}\right] (\mathcal{T})^k$$

3.7.2 Graph Attention Models

The context distribution, indicated as Q, is the hyper-parameter that controls the likelihood of sampling a node-pair when it is visited within a certain distance. Factorizing $Q = (Q_1, Q_2, ..., Q_C)$ where $Q_k \geq 0$ and $\sum_k Q_k = 1$, Q_k can be

assigned as the co-efficient to $(T)^k$.

$$\mathbb{E}\left[\mathbf{D}; Q_1, Q_2, \ldots Q_C\right] = \tilde{\mathbf{P}}^{(0)} \sum_{k=1}^{C} Q_k (\mathcal{T})^k = \tilde{\mathbf{P}}^{(0)} \mathop{\mathbb{E}}_{k \sim Q} \left[(\mathcal{T})^k\right]$$

Q_k is respectively $[1 - \frac{k-1}{C}]$ or $\propto \frac{1}{k}$ when training through word2vec or GloVe.

The Graph Attention Model represents a node's context distribution Q as the softmax output, and train the q_k through learning process together with node embedding training. Leveraging large value of C, every graph learns a specific attention. The expectation with the infinite power series of transition matrix is defined as belows:

$$\mathbb{E}\left[\mathbf{D}^{\text{softmax }[\infty]}; q_1, q_2, q_3, \ldots\right] = \tilde{\mathbf{P}}^{(0)} \lim_{C \to \infty} \sum_{k=1}^{C} \text{softmax}\left(q_1, q_2, q_3, \ldots\right)_k (\mathcal{T})^k$$

3.7.3 Training Objectives

The training loss with Graph Attention Models is modified on the basis of DeepWalk loss function:

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

With this strategy, the context hyper-parameters are replaced with trainable models. Aside from improved efficiency, the model eliminates the need for manual grid searches over random walk length and context distribution form. Experiments have shown that the model is robust to its hyperparameters, and that varying the length of the random walk has diverse impacts on the information preservation of different graphs.

4 Hyperbolic embeddings

To understand a graph topology more intuitively, it is usually helpful to embed the graph into a metric space so that the structure can be visualized. Although Euclidean space is efficient in most graph embedding tasks, it faces many difficulties when handling tree-like structures whose size grows exponentially such as social networks.

The following section will introduce two popular hyperbolic model, then analyze their advantages and disadvantages. The section after that will take the Graph Neural Network as an example to show how to apply the hyperbolic metric on a method based Euclidean space. Finally, some experiments are introduced to compare the Euclidean model and hyperbolic models.

4.1 Hyperbolic Models

Different from Euclidean space's zero curvature, hyperbolic space is a space with a constant negative curvature. Also, it is a smooth Riemannian manifold and as such locally Euclidean space.

4.1.1 Lorentz Model

According to Peng et al. (2015) [34], the Lorentz model $\mathbb{L}^n$ of an n dimensional hyperbolic space is a manifold embedded in the $n+1$ dimensional Minkowski space. The Lorentz model is defined as the upper sheet of a two-sheeted n-dimensional hyperbola with the metric $\mathfrak{g}^L$ (Nickle et al. 2018) [32], which is:

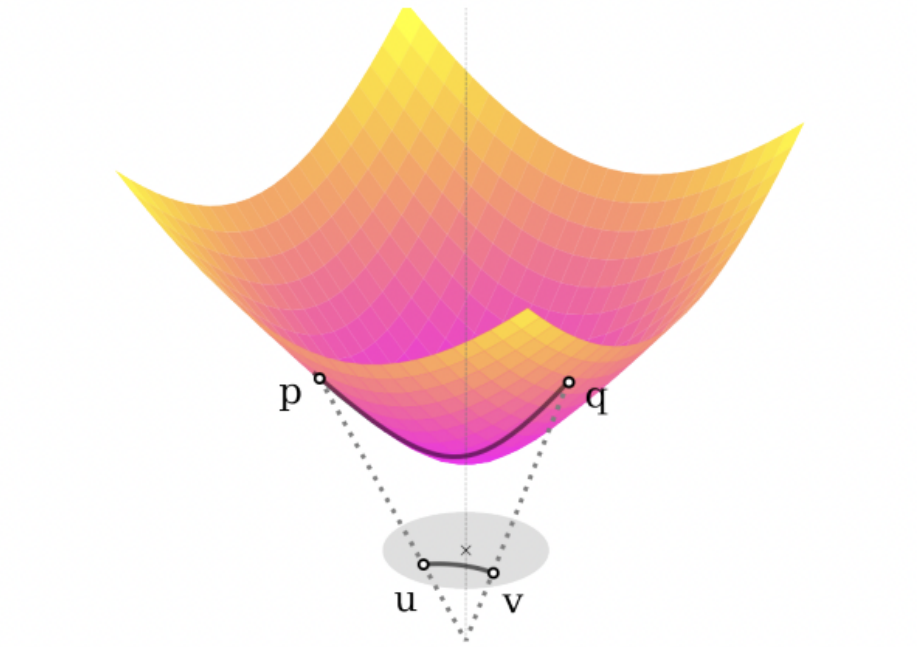
$$\mathbb{L}^n = \left\{ x = (x_0, \ldots, x_n) \in \mathbb{R}^{n+1} :< x, x >_{\mathbb{L}} = -1, x_0 > 0 \right\},$$

where the $<,>_{\mathbb{L}}$ represents the Lorentzian inner product, which is defined as

$$< x, y >_{\mathbb{L}} = x^T \mathfrak{g}^L y = -x_0 y_0 + \sum_{i=1}^{n} x_i y_i,$$

x and $y \in \mathbb{R}^{n+1}$ where $\mathfrak{g}^L$ is a diagonal matrix with entries of 1s, except for the first element being -1. For any $x \in \mathbb{L}^n$, we can get that $x_0 = \sqrt{1 + \sum_{i=1}^{n+1} x_i^2}$. The distance in the Lorentz Model is defined as

$$d(x, y) = \operatorname{arcosh}\left(- < x, y >_{\mathbb{L}}\right).$$



(b) Lorentz model of hyperbolic geometry.

The main advantage for the Lorentz Model is that it has simple distance formula, making it easier to do the optimization calculations. Moreover, different from next Poincaré Model, there is no singularity in the formula of Lorentz Model, which avoids the numerical instability during the iterative algorithm.

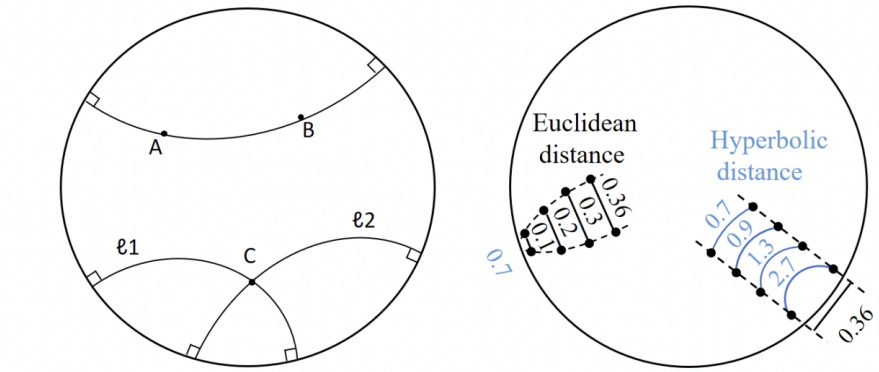
4.1.2 Poincaré Model

The Poincaré model is given by projecting each point of $\mathbb{L}^n$ onto the hyperplane $x_0 = 0$, using the rays emanating from $(-1, 0, \ldots, 0)$. The Poincaré model $\mathbb{B}$ is a manifold equipped with a Riemannian metric $\mathfrak{g}^B$. This metric is conformal to the Euclidean metric $\mathfrak{g}^E$ with the conformal factor $\lambda_x = \frac{2}{1-\|x\|^2}$, and $\mathfrak{g}^B = \lambda_x^2 \mathfrak{g}^E$ [34]. Formally, an n dimensional Poincaré unit ball (manifold) is defined as

$$\mathbb{B}^n = \{x \in \mathbb{R}^n : \|x\| < 1\},$$

where $\|\cdot\|$ denotes the Euclidean norm. Formally, the distance between $x, y \in \mathcal{M}$ is defined as:

$$d(x, y) = \operatorname{arcosh}\left(1 + 2\frac{\|x - y\|^2}{(1 - \|x\|^2)(1 - \|y\|^2)}\right).$$



The Poincaré model can be understand as a disk whose point density grow to infinity when a point getting close to the disk boundary. The main advantage of the Poincaré Model is that it is on a hyperplane, which makes it easier to visualize the hyperbolic structure. In the above figure, the left disk shows the "straight" lines in the Poincaré disk, which are required to be perpendicular to the boundary by definition. The right picture provides a intuitive comparison of distance between Euclidean model and Poincaré model.

4.1.3 Transform of the Two Models

The origin in Euclidean space and the Poincaré disk, can be transformed to $(1, 0, \ldots, 0)$ in the Lorentz model. the following are the general transformation formula:

$$p_{\mathcal{L}\to\mathcal{B}}\left(x_0, x_1, \ldots, x_n\right) = \frac{\left(x_1, \ldots, x_n\right)}{x_0 + 1}$$

$$p_{\mathcal{B}\to\mathcal{L}}\left(x_0, x_1, \ldots, x_n\right) = \frac{\left(1 + \|\mathbf{x}\|^2, 2x_1, \ldots, 2x_n\right)}{1 - \|\mathbf{x}\|^2}$$

Remark: Liu et al. (2019) [28] point out that "activation functions such as ReLU and leaky ReLU are not manifold-preserving in the Lorentz model." This is because the Lorentz model is just the upper sheet of the hyperbola, and ReLU activation function will vanish all the values on the manifold. Hence, it is necessary to map the Lorentz model to Poincaré disk before the activation, then mapping the value back to the Lorentz Model after the activation.

4.2 Generalized Euclidean Operations

In this part, many frequently used operations on the Euclidean Space will be generalized on the hyperbolic space.

4.2.1 Basic Arithmetic Operations

The following generalization is based on applying Möbius transformation on Euclidean space to get the Gyrovector space. Specifically, for a model $\mathbb{B} :=$

$\{x \in \mathbb{R}^n\ , ||x|| < 1\}$, the Gyrovector space provides a non-associatitative algebraic formulation for studying hyperbolic geometry, in analogy to the way vector spaces are used in Euclidean geometry (Peng et al. 2015) [34].

In the Gyrovector space, the Möbius addition $\oplus$ for x and y in model B is defined as

$$x \oplus y = \frac{\left(1 + 2\langle x, y\rangle + \|y\|^2\right) x + \left(1 - \|x\|^2\right) y}{1 + 2\langle x, y\rangle + \|x\|^2\|y\|^2}$$

The Möbius subtraction is simply defined as: $x \ominus y = x \oplus (-y)$
The Möbius scalar multiplication $\otimes$ is defined as

$$r \otimes x = \begin{cases} \tanh\left(r \operatorname{artanh}(\|x\|)\frac{x}{\|x\|}, \right. & x \in \mathbb{B}^n \\ 0, & x = 0, \end{cases}$$

where r is a scalar factor [20].

4.2.2 Exponential Map and Logarithm Map

In fact, all the above formula can be derived by the exponential map and logarithm map. Logarithm map can project a point on the hyperbolic manifold to its tangent space while exponential map can project it back to the hyperbolic manifold. Therefore, for any operation $\cdot$ with $a \cdot b$ on Euclidean space, its corresponding term in the hyperbolic space should be: $x \odot y = Exp_v(Log_v(x) \cdot Log_v(y))$.

Poincaré Model:
For a vector $v \in \mathcal{T}_x\mathcal{M}$ in the tangent space, the exponential map is defined as

$$\exp_x(v) = x \oplus \left(\tanh\left(\frac{\lambda_x\|v\|}{2}\right)\frac{v}{\|v\|}\right),$$

and as the inverse operation of the exponential map, for a point $y \in \mathbb{B}$ on the manifold, the logarithmic map is defined as

$$\log_x(y) = \frac{2}{\lambda_x}\ \operatorname{artanh}\ (\|-x \oplus y\|)\frac{-x \oplus y}{\|-x \oplus y\|}.$$

In which the λ_x is the conformal factor [17].

Lorentz Model:
The exponential map $\exp_{\mathbf{x}} : \mathcal{T}_{\mathbf{x}}\mathcal{L} \to \mathcal{L}$ and the logarithmic map $\log_{\mathbf{x}} : \mathcal{L} \to \mathcal{T}_{\mathbf{x}}\mathcal{L}$ are defined as:

$$\exp_{\mathbf{x}}(\mathbf{v}) = \cosh\left(\|\mathbf{v}\|_{\mathcal{L}}\right)\mathbf{x} + \sinh\left(\|\mathbf{v}\|_{\mathcal{L}}\right)\frac{\mathbf{v}}{\|\mathbf{v}\|_{\mathcal{L}}}$$

$$\log_{\mathbf{x}}(\mathbf{y}) = \frac{\operatorname{arcosh}\left(-\langle \mathbf{x}, \mathbf{y}\rangle_{\mathcal{L}}\right)}{\sqrt{\langle \mathbf{x}, \mathbf{y}\rangle_{\mathcal{L}}^2 - 1}}\left(\mathbf{y} + \langle \mathbf{x}, \mathbf{y}\rangle_{\mathcal{L}}\mathbf{x}\right)$$

where $\|\mathbf{v}\|_{\mathcal{L}} = \sqrt{\langle \mathbf{v}, \mathbf{v}\rangle_{\mathcal{L}}}$.

4.3 Hyperbolic Graph Neural Network

This section takes Graph Neural Network (GNN) as an instance to show how to generalize a Euclidean based method to hyperbolic space.

4.3.1 GNN on Euclidean Space

Graph neural networks can be interpreted as performing message passing between nodes (Liu et al. 2019) [28]. The framework is based on graph convolutional networks where node representations are computed by aggregating messages from direct neighbors over multiple steps. That is, the message from node v to its receiving neighbor u is computed as $\mathbf{m}_v^{k+1} = \mathbf{W}^k \tilde{\mathbf{A}}_{uv} \mathbf{h}_v^k$. Here $\mathbf{h}_v^k$ is the representation of node v at step $k, \mathbf{W}^k \in \mathbb{R}^{h \times h}$ constitutes the trainable parameters for step k (i.e., the k-th layer), and $\tilde{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-\frac{1}{2}}$ captures the connectivity of the graph. To get $\tilde{\mathbf{A}}$, the identity matrix $\mathbf{I}$ is added to the adjacency matrix $\mathbf{A}$ to obtain self-loops for each node, and the resultant matrix is normalized using the diagonal degree matrix $\left(\mathbf{D}_{ii} = \sum_j (\mathbf{A}_{ij} + \mathbf{I}_{ij})\right)$. We then obtain a new representation of u at step $k+1$ by summing up all the messages from its neighbors before applying the activation function $\sigma : \mathbf{h}_u^{k+1} = \sigma\left(\sum_{v \in \mathcal{I}(u)} \mathbf{m}_v^{t+1}\right)$, where $\mathcal{I}(u)$ is the set of in-neighbors of u, i.e. $v \in \mathcal{I}(u)$ if and if only v has an edge pointing to u. Thus, in a more compact notation, the information propagates on the graph as:

$$\mathbf{h}_u^{k+1} = \sigma\left(\sum_{v \in \mathcal{I}(u)} \tilde{\mathbf{A}}_{uv} \mathbf{W}^k \mathbf{h}_v^k\right).$$

More details about GNN will be introduced in section 5.

4.3.2 Generalized GNN

Similar to how to generalize operations on Euclidean space to hyperbolic space, the generalization of GNN on hyperbolic space also utilize the exponential and logarithm maps. The propagation rule for each node $u \in V$ is calculated as:

$$\mathbf{h}_u^{k+1} = \sigma\left(\exp_{\mathbf{x}'}\left(\sum_{v \in \mathcal{I}(u)} \tilde{\mathbf{A}}_{uv} \mathbf{W}^k \log_{\mathbf{x}'}\left(\mathbf{h}_v^k\right)\right)\right).$$

At layer k, we map each node representation $\mathbf{h}_v^k \in \mathcal{M}$, where $v \in \mathcal{I}(u)$ is a neighbor of u, to the tangent space of a chosen point $\mathbf{x}' \in \mathcal{M}$ using the logarithmic map $\log_{\mathbf{x}'}$. Here $\tilde{\mathbf{A}}$ and $\mathbf{W}^k$ are the normalized adjacency matrix and the trainable parameters, respectively.
Remark: Liu et al. (2019) explain that the activation σ is applied after the exponential map to prevent exponential map cancelled by the logarithm map

[28]: if the activation was applied before the exponential map, i.e. $\mathbf{h}_u^{k+1} = \exp_{\mathbf{x}'}\left(\sigma\left(\sum_{v\in\mathcal{I}(u)}\tilde{\mathbf{A}}_{uv}\mathbf{W}^k\log_{\mathbf{x}'}\left(\mathbf{h}_v^k\right)\right)\right)$, the exponential map $\exp_{\mathbf{x}'}$ at step k would have been cancelled by the logarithmic map $\log_{\mathbf{x}'}$ at step $k+1$ as $\log_{\mathbf{x}'}\left(\exp_{\mathbf{x}'}(\mathbf{h})\right) = \mathbf{h}$.

4.3.3 Centroid-Based Regression and Classification

Just as above, a generalized classification and regression method for GNN is needed on hyperbolic space.
The key idea is to build a differentiable function $\psi : \mathcal{M} \to \mathbb{R}^d$ that can collect the structural information of the graph. In detail, we first introduce a list of centroids $\mathcal{C} = \left[\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{|\mathcal{C}|}\right]$, where each $\mathbf{c}_i \in \mathcal{M}$. The centroids are learned jointly with the GNN using backpropagation. The pairwise distance between $\mathbf{c}_i$ and $\mathbf{h}_j^K$ is calculated as: $\psi_{ij} = d\left(\mathbf{c}_i, \mathbf{h}_j^K\right)$. Next, we concatenate all distances $\left(\psi_{1j}, \ldots, \psi_{|\mathcal{C}|j}\right) \in \mathbb{R}^{|\mathcal{C}|}$ to summarize the position of $\mathbf{h}_j^K$ relative to the centroids. For node-level regression,

$$\hat{y} = \mathbf{w}_o^T\left(\psi_{1j}, \ldots, \psi_{|\mathcal{C}|j}\right),$$

where $\mathbf{w}_o \in \mathbb{R}^{|C|}$, and for node-level classification,

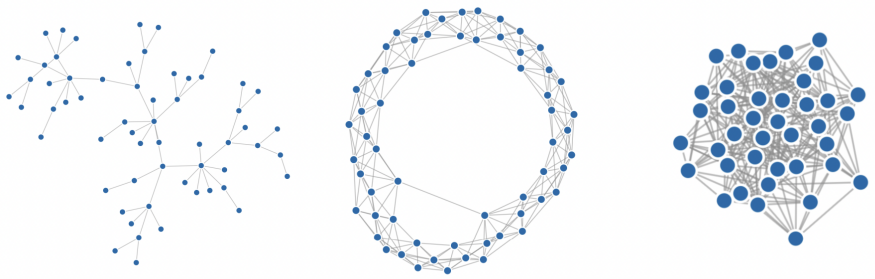
$$p\left(y_j\right) = \text{softmax}\left(\mathbf{W}_o\left(\psi_{1j}, \ldots, \psi_{|\mathcal{C}|j}\right)\right)$$

where $\mathbf{W}_o \in \mathbb{R}^{c\times|C|}$ and c denotes the number of classes. For graph-level predictions, we first use average pooling to combine the distances of different nodes, obtaining $\left(\psi_1, \ldots, \psi_{|\mathcal{C}|}\right)$, where $\psi_i = \sum_{j=1}^{|V|}\psi_{ij}/|V|$, before feeding $\left(\psi_1, \ldots, \psi_{|\mathcal{C}|}\right)$ into fully connected networks. Standard cross entropy and mean square error are used as loss functions for classification and regression, respectively.

4.4 Graph Structure Classification Experiment

This section will introduce a experiment using the hyperbolic GNN, and compare the performance of Euclidean Model, Lorentz Model and Poincaré Model.

This experiment is designed to check whether hyperbolic space do better at capturing the structural information of graphs.



(a) Barabási-Albert (b) Watts-Strogatz (c) Erdős-Rényi

The above three graphs are created by three distinct graph generator: Erdos-Rényi, Barabási-Albert and Watts-Strogatz. And the task is to classify the corret generator a graph belongs to.

For each graph generator we uniformly sample a number of nodes between 100 and 500 and subsequently apply the generation algorithm on the nodes. For Barabási-Albert graphs, we set the number of edges to attach from a new node to existing nodes to a random number between 1 and 100. For Erdős-Rényi, the probability for edge creation is set to $0.1 - 1$. For Watts-Strogatz, each node is connected to $1 - 100$ nearest neighbors in the ring topology, and the probability of rewiring each edge is set to $0.1 - 1$ [28].

4.5 Results and Conclusion

	Dimensionality				
	3	5	10	20	256
Euclidean	77.2 ± 0.12	90.0 ± 0.21	90.6 ± 0.17	94.8 ± 0.25	95.3 ± 0.17
Poincare	93.0 ± 0.05	95.6 ± 0.14	95.9 ± 0.14	96.2 ± 0.06	93.7 ± 0.05
Lorentz	94.1 ± 0.03	95.1 ± 0.25	96.4 ± 0.23	96.6 ± 0.22	95.3 ± 0.28

Table 1: F1 (macro) score and standard deviation of classifying synthetically generated graphs according to the underlying graph generation algorithm (high is good).

The above table shows the result of the experiment. It can be observed that both of the hyperbolic models performs much better than the Euclidean model, especially in low dimensional space. It can be inferred that hyperbolic space is strong enough to collect structural information despite the low-dimension. This is a big advantage for hyperbolic space because it can hugely reduce the memory complexity. Furthermore, we can notice that Lorentz model performs slightly better than the Poincaré Model but only one case. This result is consistent with above analysis of Lorentz model in section 4.1.1: it do better at numerical optimization than the Poincaré Model.

5 Graph Neural Networks (GNN)

Whereas previously discussed methods create graph embeddings using only graph topology (which encodes similarity between nodes), it is often desirable to consider functions on graphs that also consider node features. For example, in the case of social media analysis where edges between nodes may represent interactions between users, it is not always the case that nodes connected with an edge are similar, since interacting users on social media may disagree with each other. As such, in these cases it is necessary to also consider features of each individual user/node in the final graph representation. Graph neural networks (GNNs) provide a natural way of accounting for both graph topology and node features through message passing - a local neighborhood feature aggregation procedure [30].

The general message passing paradigm is as follows: We are given an undirected graph $G = (V, E)$ with the set of edges $E \subset V \times V$ and node features $x_u \in \mathbb{R}^d, \forall u \in V$. We stack the node features into a single nd matrix X. As usual, the graph topology is represented using a (potentially weighted) adjacency matrix A.

We want to look for a function F that is local at each step. In other words, the latent representation h_u for each node u depends only on the features of nodes in the neighborhood of u (X_{N_u}). Additionally, since graphs do not have a natural order (permuting the nodes should not affect the latent representation of each node), we require F to be permutation-equivariant, in the sense that F commutes with the permutation operator P (which simply permutes the rows in an adjacency matrix):

$$F(PX, PAP^T) = PF(X, A) \quad \forall P$$

A natural way of achieving these requirements is to assume the following form for F:

$$h = F(X, A) = \begin{bmatrix} \phi(x_1, X_{N_1}) \\ \phi(x_2, X_{N_2}) \\ \vdots \\ \phi(x_n, X_{N_n}) \end{bmatrix}$$

Here, ϕ is a local aggregation function that is permutation-invariant, i.e. $\phi(x_i, X_{N_u}) = \phi(x_i, PX_{N_u})$. An aggregation that is permutation invariant can be easily achieved through, for example, a sum, average, or max over neighborhood node features. Below, we consider the three general classes of GNNs.

5.1 Three flavors of message-passing GNNs

According to Bronstein et al. (2021), graph neural networks can be broadly classified into three flavors, the most general being the message-passing neural network (MPNN), followed by graph attentional networks (GAT), and graph convolutional networks (GCN).

First, in the convolutional flavor, only the sender creates messages and each message is weighted using a fixed coefficient c_{uv} which is determined completely by the graph topology (the adjacency matrix A).

$$h_u = \phi\left(x_u, \bigoplus_{v \in N_u} c_{uv}\psi(x_v)\right)$$

Note that the $\bigoplus$ denotes a permutation-invariant aggregation and ϕ, ψ are learnable functions (e.g. MLPs).

In the attentional case, the message depends only on the sender, but the weights associated to each message are neural networks that depend on both parties:

$$h_u = \phi\left(x_u, \bigoplus_{v \in N_u} a(x_u, x_v)\psi(x_v)\right)$$

Finally, for general MPNNs, the message sent depends on both the sender and receiver:

$$h_u = \phi\left(x_u, \bigoplus_{v \in N_u} \psi(x_u, x_v)\right)$$

It is evident that the only difference between these flavors is the degree to which they parametrize their message vectors using neural networks ψ. GCNs assume that the graph has a homophilous structure, i.e that neighboring nodes are similar and thus the weight of their interactions c_{uv} can be captured with graph topology alone. Because of their dependence on just the adjacency matrix A, GCNs can compute their latent states using sparse matrix multiplication and are thus highly scalable.

In the case of GATs, the weights themselves become parametrized and dependent on node features. In this case, we drop the assumption that proximity implies similarity. Nonetheless, GATs still compute only a scalar per edge and are thus the middle-ground between GCNs and MPNNs in terms of scalability, scale, capacity, and interpretability. It is worth noting that the widely celebrated Transformer [4] architecture is in fact equivalent to a GAT instantiated on a fully connected graph.

Finally, general MPNNs compute arbitrary vectors to be sent across edges and are ideal for applications in computational chemistry and physics simulation tasks. They do, of course, less scalable due to their large parameter count.

In the following two subsections, we perform an in-depth study of the inner-workings of GCNs and GATs. We then take a look at an alternative way of implicitly incorporating graph structure via structural encoding with Graphormer.

5.1.1 Graph convolutional networks (GCN)

The aggregation used in GCN is a generalization of the convolution operation to the graph domain [45]. Recall that the Fourier basis diagonalizes

the convolution operation (computing the convolution is equivalent to FFT, element-wise multiplication by a diagonal matrix, followed by IFFT) and that, furthermore, the Fourier bases are eigenfunctions of the Laplacian operator. Hence, we can consider the eigenvectors of the normalized graph Laplacian $L = I_N - D^{-1/2}AD^{-1/2} = U\Lambda U^T$ as the Fourier basis vectors (recall that L is PSD and thus diagonalizable). The convolution between any filter f_θ and a one-dimensional signal x can then be computed as:

$$f_\theta * x = U^T g_\theta U x$$

Where g_θ is the diagonal matrix of the filter f_θ expressed in the (complete) Fourier eigenbasis. In practice, we perform computations purely in the spectral domain. However, $U^T g_\theta U x$ is expensive to compute since the matrix multiplication has cost $O(N^2)$ and computing U requires an eigendecomposition, which is costly for large graphs. To circumvent this problem, Hammond et al. (2011) [11] proposes that g_θ can be approximated by a truncated sum of Chebyshev polynomials T_k:

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

Where $\tilde{\Lambda} = \frac{2}{\lambda_{max}} L - I_N$, where λ_{max} is the largest eigenvalue of L. This gives rise to:

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

With $L \approx \frac{2}{\lambda_{max}} L - I_N$. This step arises because $L^k = (U\Lambda U^T)^k = U\Lambda^k U^T$.

This allows us to avoid computing the eigen-decomposition explicitly, achieving complexity of $O(|E|)$. Notice that since this is a K-order polynomial in L, the aggregation is over nodes in the K-hop neighborhood of the central node.

GCN simplifies the graph convolution further by considering only the first order approximation (K=1) and assumes that $\lambda_{max} \approx 2$ (parameters of the neural network will offer rescaling during training). Furthermore, it reduces the number of parameters to just one: $\theta = \theta_0' = -\theta_1'$. Hence:

$$f_\theta * x \approx \theta \left(I + D^{-1/2}AD^{-1/2} \right) x$$

According to the paper, the point of using just the first order approximation is to avoid overfitting on local neighborhood structures for graphs with wide node degree distributions. We achieve aggregation over larger neighborhoods by simply stacking layers (and element-wise non-linearities). Note, however, that $I + D^{-1/2}AD^{-1/2}$ has eigenvalues in the interval $[0, 2]$ and thus repeatedly multiplying by $I + D^{-1/2}AD^{-1/2}$ can cause exploding gradients. This is mitigated using a renormalization trick $I + D^{-1/2}AD^{-1/2} \rightarrow \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$, where $\tilde{A} = A + I_N$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. For higher dimensional node features, the operation generalizes to:

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

Where $\Theta \in R^{C\times F}, X \in R^{N\times C}$. The operation can be efficiently computed with complexity $O(|E|FC)$. However, limitations include the fact that the aggregation is determined purely by graph topology, thus assuming homophilous graph structure. Furthermore, GCNs are non-inductive in the sense that they cannot readily generalize to different graph structures (since a different graph adjacency matrix requires a complete retraining).

5.1.2 Graph attention networks (GAT)

Given node features of N nodes $h = h_1, \ldots, h_N$, where $h_i \in R^F$, GAT [37] employs the self-attention mechanism to calculate coefficients $e_ij = a(Wh_i, Wh_j)$, where $a(.,.)$ is a feedforward layer (e.g. MLP with ReLU activation). These coefficients are then normalized:

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k\in N_i}\exp(e_{ik})}$$

Recall that α_{ij} serves as the weighting of messages used to update the node representation of node i, i.e.:

$$h_i' = \sigma\left(\sum_{j\in N_i}\alpha_{ij}Wh_j\right)$$

Where σ is some non-linearity. The following are some crucial differences between GAT and GCN:

1. For GAT, e_{ij} depends on both topology and node features, hence being more flexible than GCN.

2. GAT maintains a complete graph over the nodes. The graph structure is injected later via masked attention, computing e_{ij} only for nodes $j \in N_i$. Hence, GAT models are independent of global graph structure, allowing them to be readily applied in inductive learning applications (i.e. when one has new graph structures during test time).

3. Self-attention is parallelizable across edges. Computation of output features is parallelizable across nodes.

4. No eigendecomposition of Laplacian is required (since self-attention mechanism is independent of graph structure).

5.1.3 Structural encoding using Graphormer

Instead of explicitly accounting for graph structure using spectral properties (e.g. GCNs) or masking out nodes according to the adjacency matrix (as in the case of GAT), Graphormer [8] uses a transformer and treats the graph as

being fully connected. Structural information is then injected into the attention mechanism via structural encoding, allowing the model to account for graph structure implicitly.

More concretely, Graphormer uses a combination of centrality encoding, spatial encoding, and edge encoding. Degree centrality encoding accounts for node centrality – i.e. the indegree and outdegree of a node – which in turn provides information about the importance of the node in the graph. This is particularly important, for example, in social media networks, whereby group dynamics are heavily dominated by a few celebrities. The implementation is as trivial as adding the indegree and outdegree to the initial node features:

$$h_i^{(0)} = x_i + z^-_{\deg^-(v_i)} + z^+_{\deg^+(v_i)}$$

Spatial encoding accounts for the connectivity structure determined by the adjacency matrix, and is achieved with a function $\phi(v_i, v_j) : V \times V \to \mathbb{R}$ to measure the spatial relations between nodes. A learnable bias $b_\phi(v_i, v_j)$ is then added to the attention matrix.

Finally, edge features are encoded by similarly adding another bias term given by $c_{ij} = \frac{1}{N}\sum_{n=1}^{N} x_{e_n}(w_n^E)^T$. The resultant attention matrix is thus given by:

$$A_{ij} = \frac{(h_i W_Q)(h_j W_K)^T}{\sqrt{d}} + b_{\phi(v_i,v_j)} + c_{ij}$$

Despite the trivial implementation of Graphormer, the architecture has proven competitive against GCNs and GATs on several molecular modeling datasets (MolHIV, MolPCBA and ZINC).

The obvious benefit of using a fully-connected graph is that the receptive field captures the entire graph, i.e. features can freely propagate between each and every node and edge on the graph while being softly modulated by structural information (via structural encoding). The disadvantage, of course, is that this yields a $O\left(|V|^2\right)$ cost in computation and memory and is thus not scalable to extremely large graphs.

5.2 Types of problems addressed by GNNs

It is clear that there are in general three types of problems that GNNs try to solve: node feature prediction, link prediction/latent graph inference, and graph prediction. To generate latent representations of node features, it suffices to stack multiple layers of permutation-equivariant functions F with other standard neural network components such as non-linearities, pooling layers, and normalization. In the case of graph prediction, where one tries to generate a graph-wide output (e.g. the total energy of a molecule represented by the graph), one simply uses a permutation-invariant global pooling layer as the final layer of the network.

In the link prediction/graph inference problem, one is given an incomplete graph and is asked to create more edges to complete the graph.

The Neural Relational Inference [44] solution to the problem involves assuming upfront a prior distribution over the edges. A GNN is then applied over the fully connected graph to obtain the latent edge features, followed by a decoder that generates a posterior over the edges. Edges are the sampled from this posterior. These edges are then used for a second GNN for the original downstream task. Note that since the action of choosing an edge is an inherently discontinuous function (0 if the edge is not chosen and 1 if it is), it is necessary to refrain from making hard decisions at train time so as to guarantee backpropagation works. Instead, one applies the Gumbel trick, whereby one takes a little of every edge weighted by its posterior probabilities.

Dynamic graph CNNs [48] and Pointer graph networks [38] adopt a very similar yet somewhat simpler approach of using dot product as measures of similarities between nodes and either creating edges for the k largest similarities or letting the probability of edges existing being proportional to these similarities.

Finally, the Differentiable Graph Module [3] combines the similarity-weighted approach mentioned above with a reinforcement learning agent. At every GNN layer, the probability of an edge existing (computed using dot products of node representations) is used as a stochastic policy for an RL agent that selects k edges for every node. The agent is then rewarded based on downstream performance (e.g. accuracy measure) and the policy is optimized using policy gradient methods (e.g. REINFORCE, TRPO, PPO).

5.3 Fundamental limitations of GNNs and generalizations

The message propagation mechanism in GNNs can be viewed as local diffusion processes on graphs. This physical analogy gives rise to several key limitations of GNNs, namely limited expressivity (in terms of capturing graph structure), inability to accommodate long-range dependencies (LRD), oversmoothing, and oversquashing of information.

We begin by noting that since GNNs perform local message passing, each layer of the GNN can only propagate messages across 1-neighborhoods (i.e. directly adjacent nodes). Hence, learning interactions between distal nodes (say nodes k-steps apart) necessitates a deep network (of depth at least k). Increasing depth of neural networks typically leads to overfitting. For GNNs, this also leads to the additional problem of oversmoothing, whereby latent node representations generated by deep layers tend to be very similar to each other. Intuitively, this is because since GNNs are diffusion processes, excessive diffusion leads to homogenization of latent features.

Additionally, Alon & Yahav (2021) [46] identified the problem of oversquashing, where messages become distorted when propagating between distal nodes. This occurs specifically at bottlenecks of the graph – regions where there are (exponentially) many incoming messages that must be compressed into a fixed size vector representation. This poses challenge for capturing long-range dependencies, especially when the graph has exponentially many long-range nodes (e.g. in "small-world graphs"/social networks, receptive field grows exponentially with

distance).

Finally, Xu et al. (2019) [26] noticed a striking similarity between the message passing mechanism and the Weisfeiler Lehman graph isomorphism test, which is known to fail at distinguishing certain non-isomorphic graphs. In their paper, they formally demonstrate the equivalence of the two and thus the limited expressivity of GNNs for capturing graph structure.

In the following subsections, we explore the formalism for each problem and the methods proposed to address them.

5.3.1 Oversmoothing and Graph Coupled Oscillator Networks

Rusch et al. (2022) [42] formalizes the limitation of oversmoothing (homogenization of latent vector representations) using the Dirichlet energy:

$$E(X) = \frac{1}{|V|} \sum_{i \in V} \sum_{j \in N_i} ||X_i - X_j||^2$$

The Dirichlet energy measures the average similarity of each node with its 1-neighborhood. We say oversmoothing occurs if the Dirichlet energy exponentially decays with the layer number/diffusion timestep n in the GNN, i.e. $E(X^n) \leq C_1 e^{-C_2 n}$. To better analyze this problem, one considers the continuous relaxation ($n \to t \in \mathbb{R}$) and models the diffusion process as a dynamical system. In particular, the authors propose a coupled oscillator ODE (GraphCON) given by:

$$X'' = \sigma(F_\theta(X, t)) - \gamma X - \alpha X'$$

Where $(F_\theta(X, t))_i = F_\theta(X_i(t), X_j(t), t), \forall (i, j) \in E$ is the diffusion/message passing operation (e.g. attentional, convolutional), σ is a non-linearity, and γ and α are constants. This second order ODE can be written as a system of first-order ODEs:

$$Y' = \sigma(F_\theta(X, t)) - \gamma X - \alpha Y$$
$$X' = Y$$

In this representation the coupled nature of the two equations is made explicit. To actually perform message passing, one needs to discretize the above system (e.g. using IMEX), giving rise to:

$$Y^n = Y^{n-1} + \Delta t \left[\sigma(F_\theta(X^{n-1}, t^{n-1})) - \gamma X^{n-1} - \alpha Y^{n-1}\right]$$
$$X^n = X^{n-1} + \Delta t Y^n$$

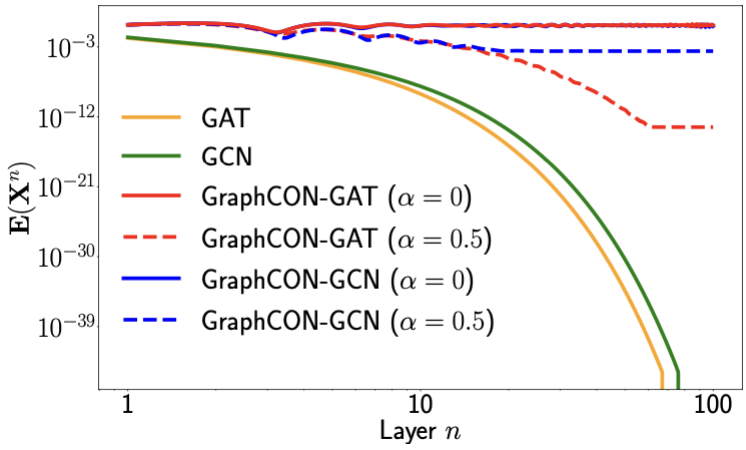
Where $t^n = n\Delta t$, $n = 1, \ldots, N$, with N being the depth of the GraphCON network. One notes that GraphCON is a generalization of GNNs, since GNNs can be recovered as the steady state solutions (which means when $X' = 0$, $Y' = 0$) X^*,Y^* of GraphCON ODEs with autonomous coupling function $F_\theta = F_\theta(X)$:

$$Y^* = 0$$
$$X^* = \frac{\Delta t}{\gamma}\sigma(F_\theta(X^*))$$

The second equation is solved using fixed point iteration, giving rise to the standard GNN architecture:

$$X^n = \sigma(F_\theta(X^{n-1}))$$

Furthermore, GraphCON, by construction, solves the oversmoothing problem. First, it can be trivially proven that oversmoothing occurs if and only if $(X^*, Y^*) = (c, 0)$ are exponentially stable steady states (fixed points) of the ODE. The primary proof of the GraphCON paper is for the following theorem: For GraphCON, assuming the activation σ is ReLU, for any $c \in \mathbb{R}^m$ such that each entry is non-negative, the hidden state $(c, 0)$ is a steady state. However, if $\alpha \geq 1/2$, this fixed point is not exponentially stable, thereby preventing oversmoothing. Experimental results show that GraphCON drastically improves the oversmoothing phenomenon compared to baselines (vanilla GAT and GCN). Moreover, GraphCON also achieved state-of-the-art results for numerous downstream node classification tasks.



(GraphCON wrapper alleviates oversmoothing)

Homophily level	**Cora** **0.81**	**Citeseer** **0.74**	**Pubmed** **0.80**
GAT-ppr	81.6 ± 0.3	68.5 ± 0.2	76.7 ± 0.3
MoNet	81.3 ± 1.3	71.2 ± 2.0	78.6 ± 2.3
GraphSage-mean	79.2 ± 7.7	71.6 ± 1.9	77.4 ± 2.2
GraphSage-maxpool	76.6 ± 1.9	67.5 ± 2.3	76.1 ± 2.3
CGNN	81.4 ± 1.6	66.9 ± 1.8	66.6 ± 4.4
GDE	78.7 ± 2.2	71.8 ± 1.1	73.9 ± 3.7
GCN	81.5 ± 1.3	71.9 ± 1.9	77.8 ± 2.9
GraphCON-GCN	81.9 ± 1.7	72.9 ± 2.1	$\mathbf{78.8 \pm 2.6}$
GAT	81.8 ± 1.3	71.4 ± 1.9	78.7 ± 2.3
GraphCON-GAT	$\mathbf{83.2 \pm 1.4}$	$\mathbf{73.2 \pm 1.8}$	$\mathbf{79.5 \pm 1.8}$
GRAND	$\mathbf{83.6 \pm 1.0}$	$\mathbf{73.4 \pm 0.5}$	$\mathbf{78.8 \pm 1.7}$
GraphCON-Tran	$\mathbf{84.2 \pm 1.3}$	$\mathbf{74.2 \pm 1.7}$	$\mathbf{79.4 \pm 1.3}$

(Downstream performance on homophilic graphs)

Homophily level	**Texas** **0.11**	**Wisconsin** **0.21**	**Cornell** **0.30**
GPRGNN	78.4 ± 4.4	82.9 ± 4.2	80.3 ± 8.1
H2GCN	$\mathbf{84.9 \pm 7.2}$	$\mathbf{87.7 \pm 5.0}$	$\mathbf{82.7 \pm 5.3}$
GCNII	77.6 ± 3.8	80.4 ± 3.4	77.9 ± 3.8
Geom-GCN	66.8 ± 2.7	64.5 ± 3.7	60.5 ± 3.7
PairNorm	60.3 ± 4.3	48.4 ± 6.1	58.9 ± 3.2
GraphSAGE	$\mathbf{82.4 \pm 6.1}$	81.2 ± 5.6	76.0 ± 5.0
MLP	80.8 ± 4.8	85.3 ± 3.3	81.9 ± 6.4
GAT	52.2 ± 6.6	49.4 ± 4.1	61.9 ± 5.1
GraphCON-GAT	82.2 ± 4.7	$\mathbf{85.7 \pm 3.6}$	$\mathbf{83.2 \pm 7.0}$
GCN	55.1 ± 5.2	51.8 ± 3.1	60.5 ± 5.3
GraphCON-GCN	$\mathbf{85.4 \pm 4.2}$	$\mathbf{87.8 \pm 3.3}$	$\mathbf{84.3 \pm 4.8}$

(Downstream performance on heterophilic graphs)
(Images from Rusch et al. 2022)

We note that modeling GNNs as dynamical systems and introducing generalized wrappers (as in the case of GraphCON) seems to be a promising approach for solving many of the GNN bottlenecks, as one can equate performance of GNNs using stability properties of solutions to ODEs, which is a very well-studied problem (especially compared to black box neural networks). We will see later that a similar approach, namely graph neural diffusion models, is used for dynamic graph rewiring (for solving the bottleneck/oversquashing problem).

5.3.2 Long range dependencies, oversquashing, and bottlenecks

Alon & Yahav (2021) [46] identified bottlenecks – "bridge-like" topologies of the graph – as the root causes for the oversquashing phenomenon, whereby long-range message propagation is hindered due to exponentially many messages being compressed into fixed-size node representations. This is especially problematic when the receptive field grows too quickly with the distance.

Topping et al. (2022) [22] provides a theoretical framework to rigorously define oversquashing based on curvature of graphs. The authors propose using the Jacobian of node representations as a measure of oversquashing and shows how bottlenecks cause oversquashing. Bottlenecks, on the other hand, can be formalized as regions of the graph with negative curvature. This thereby enables one to identify and surgically remove bottlenecks without drastically changing the overall graph topology.

We recall the general form of an MPNN/GNN:

$$h_i^{(l+1)} = \phi_l \left(h_i^{(l)}, \sum_{j=1}^{n} \hat{A}_{ij} \psi_l(h_i^{(l)}, h_j^{(l)}) \right)$$

Concretely, oversquashing can be interpreted as the node representation $h_i^{(l)}$ of node i being insensitive to feature x_s of node s at distance r away. Recall that the latent representation h_i gets updated by x_s only after $r+1$ layers in the GNN (since message passing is local). So we can explicitly measure oversquashing using the Jacobian $\left| \frac{\partial h_i^{(r+1)}}{\partial x_s} \right|$

A smaller Jacobian implies smaller sensitivity to distal nodes and hence significant oversquashing. The authors place an explicit bound on this quantity in terms of the graph topology. In particular, given a MPNN, let $i, s \in V$ with $s \in S_{r+1}(i)$. If $|\nabla \phi_l| \leq \alpha$ and $|\nabla \psi_l| \leq \beta$ for $0 \leq l \leq r$, then:

$$\left| \frac{\partial h_i^{(r+1)}}{\partial x_s} \right| \leq (\alpha\beta)^{r+1} \left(\hat{A}^{r+1} \right)_{is}$$

For example, let d_G be the shortest distance between two nodes. If $d_G(i,s) = r+1$ and the subgraph induced by the ball $B_{r+1}(i) = \{v \in V | d_G(v,i) < r+1\}$ is a binary tree (which involves receptive field exponentially increasing with distance), then $\left(\hat{A}^{r+1} \right)_{is} = 2^{-1}3^{-r}$, i.e. the sensitivity exponentially decays with respect to distance.

Next, the authors show that exponentially decaying upper bounds on $\left| \frac{\partial h_i^{(r+1)}}{\partial x_s} \right|$ are attributed to negative curvature. Here, they define the Balanced Forman curvature:

$$Ric(i,j) := 2/d_i + 2/d_j - 2 + 2\frac{|\#_\Delta(i,j)|}{max(d_i,d_j)} + \frac{|\#_\Delta(i,j)|}{min(d_i,d_j)} + \frac{(\gamma_{max})^{-1}}{max(d_i,d_j)} \left(|\#_\square^i| + |\#_\square^j| \right)$$

Where $\#_\Delta(i,j) := S_1(i) \cap S_1(j)$ are triangles based at the edge $(i,j) \in E$, $\#^i_\square(i,j) := \{k \in S_1(i) \backslash S_1(j), k \neq j : \exists w \in (S_1(k) \cap S_1(j)) \backslash S_1(i)\}$ are neighbors of i forming a 4-cycle based on $(i,j) \in E$ without diagonals inside, and $\gamma_{max}(i,j)$ is the maximal number of 4-cycles based on $(i,j) \in E$ traversing a common node.

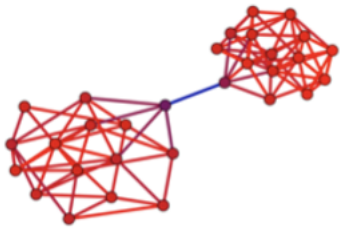
Details aside, several important properties of the Balanced Forman curvature exist. First, for positive curvature $Ric(i,j) \geq k > 0$ for any edge $(i,j) \in E$, the receptive field is bounded by a polynomial in distance $|B_r(i)| \leq P(r)$, which means the bottleneck is not significant.

Furthermore, given an MPNN, let $(i,j) \in E$ and $d_i \leq d_j$ and assume $|\nabla \phi_l| \leq \alpha$, $|\nabla \psi_l| \leq \beta$ for all layers $0 \leq l \leq L-1$ (where L is the depth) of the MPNN and that $\exists \delta$ such that $0 < \delta < (max(d_i, d_j))^{-1/2}$, $\delta < \gamma^{-1}_{max}$, and $Ric(i,j) \leq -2 + \delta$. Then there exists $Q_j \subset S_2(i)$ with $|Q_j| > \delta^{-1}$ and for all layers $0 \leq l_0 \leq L-2$ we have:

$$\frac{1}{|Q_j|} \sum_{k \in Q_j} \left| \frac{\partial h^{(l_0+2)}}{\partial h_i^{(l_0)}} \right| < (\alpha\beta)^2 \delta^{1/4}$$

This result demonstrates that negative curvatures give rise to bottlenecks.

Note that bottlenecks in graph topology can also be related to spectral properties of the graph. In particular, consider the min-cut problem whereby one is to partition the graph into two subgraphs such that the number of edges cut is minimized. We expect that if there are two communities of nodes with a small number of edges acting as "bridges" across the two communities, then the min-cut would be small and correspondingly the bottleneck would be significant.



(Image from Topping et al. 2022)

Hence, graph curvature (in the differential geometry sense) can be related to the eigengap (smallest non-zero eigenvalue) of the graph Laplacian. With this in mind, one defines the Cheeger constant as:

$$h_G := \min_{S \subset V} h_S$$

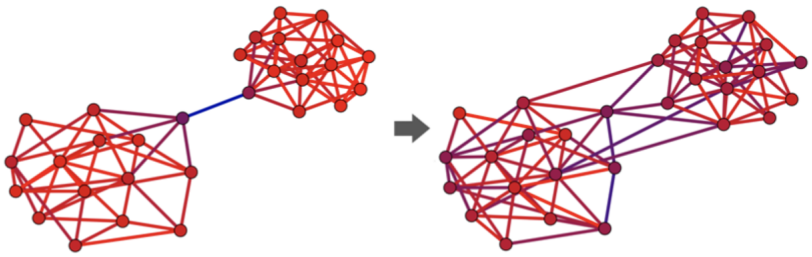
$$h_S := \min_{S \subset V} \frac{|\partial S|}{\min(vol(S), vol(V \backslash S))}$$

Where $\partial S = \{(i,j) : i \in S, j \in V \backslash S\}$ and $vol(S) = \sum_{i \in S} d_i$. The Cheeger constant relates to the smallest non-zero eigenvalue λ_1 of the Laplacian with the following inequality: $2h_G \geq \lambda_1 \geq h_G^2/2$. Hence, the smaller the Cheeger constant, the more significant the bottleneck and subsequent oversquashing.

Furthermore, as one might expect, if $Ric(i,j) \geq k > 0$ for all $(i,j) \in E$, then $\lambda_1/2 \geq h_G \geq k/2$. In other words, keeping the curvature positive places a lower bound on the Cheeger constant.

With the notion of curvature and bottleneckedness rigorously defined, the algorithm for precise graph rewiring – known as Stochastic Discrete Ricci Flow (SDRF) – simply involves probabilistically adding edges to the graph, where the probability is weighted by how much the edge would improve the edge with minimal curvature.

Input: graph G, temperature $\tau > 0$, max number of iterations, optional Ric upper-bound C^+
Repeat
  1) For edge $i \sim j$ with minimal Ricci curvature $\text{Ric}(i,j)$:
    Calculate vector $\boldsymbol{x}$ where $x_{kl} = \text{Ric}_{kl}(i,j) - \text{Ric}(i,j)$, the improvement to $\text{Ric}(i,j)$ from adding edge $k \sim l$ where $k \in B_1(i), l \in B_1(j)$;
    Sample index k, l with probability $\text{softmax}(\tau \boldsymbol{x})_{kl}$ and add edge $k \sim l$ to G.
  2) Remove edge $i \sim j$ with maximal Ricci curvature $\text{Ric}(i,j)$ if $\text{Ric}(i,j) > C^+$.
Until convergence, or max iterations reached;



(Image from Topping et al. 2022)

Note that to prevent the graph topology from changing drastically, one must keep the degree distribution and curvature distribution balanced by removing the maximal Ricci curvature edge. The consequence of this scheme is that the graph edit distance is at most 2x the number of iterations.

SDRF vs DIGL (Diffusion Improves Graph Learning): Note that prior works in graph rewiring exist. For example, a diffusion-based rewiring method proposed by Gasteiger et al. (2022) known as DIGL smooths out the adjacency matrix and thus improves connection between nodes at short diffusion distance [24]. Since bottlenecks are more prominent for nodes at long diffusion distance, such random-walk procedures are limited in effectiveness.

Additionally, in terms of preserving graph structure, given a maximum graph edit distance, SDRF surgically addresses the lowest curvature edges, whereas diffusion-based methods are less precise (and thus require a larger graph edit distance to be effective). Furthermore, since $Ric(i,j) < -2 + \delta$ implies $min(d_i, d_j) > 2/\delta$, this means that modifying edges with very negative curvature implies adding edges to nodes with high degree, which leaves the degree distribution of the graph relatively unchanged.

It is worth noting though, that since DIGL adds connections for nodes at short-diffusion distances, DIGL typically performs better for homophilous graphs and worse for non-homophilous graphs compared to SDRF.

The authors ran experiments comparing SDRF vs DIGL graph rewiring on downstream node classification tasks. Note that a standard GCN was used as a base model. $H(G)$ denotes the level of homophily in each dataset.

	Cornell	Texas	Wisconsin	Chameleon	Squirrel	Actor	Cora	Citeseer	Pubmed
$\mathcal{H}(G)$	0.11	0.06	0.16	0.25	0.22	0.24	0.83	0.71	0.79
None	52.69 ± 0.21	61.19 ± 0.49	54.60 ± 0.86	41.80 ± 0.41	39.83 ± 0.14	28.70 ± 0.09	81.89 ± 0.79	72.31 ± 0.17	78.16 ± 0.23
Undirected	53.20 ± 0.53	63.38 ± 0.87	51.37 ± 1.15	42.63 ± 0.30	40.77 ± 0.16	28.10 ± 0.11	-	-	-
+FA	$\mathbf{58.29 \pm 0.49}$	$\mathbf{64.82 \pm 0.29}$	55.48 ± 0.62	42.33 ± 0.17	40.74 ± 0.13	28.68 ± 0.16	81.65 ± 0.18	70.47 ± 0.18	$\mathbf{79.48 \pm 0.12}$
DIGL (PPR)	58.26 ± 0.50	62.03 ± 0.43	49.53 ± 0.27	42.02 ± 0.13	34.38 ± 0.11	$\mathbf{30.79 \pm 0.10}$	$\mathbf{83.21 \pm 0.27}$	$\mathbf{73.29 \pm 0.17}$	78.84 ± 0.08
DIGL + Undirected	$\mathbf{59.54 \pm 0.64}$	63.54 ± 0.38	52.23 ± 0.54	42.68 ± 0.12	33.36 ± 0.21	29.71 ± 0.11	-	-	-
SDRF	54.60 ± 0.39	64.46 ± 0.38	$\mathbf{55.51 \pm 0.27}$	$\mathbf{43.75 \pm 0.31}$	$\mathbf{40.97 \pm 0.14}$	29.70 ± 0.13	$\mathbf{82.76 \pm 0.23}$	$\mathbf{72.58 \pm 0.20}$	$\mathbf{79.10 \pm 0.11}$
SDRF + Undirected	57.54 ± 0.34	$\mathbf{70.35 \pm 0.60}$	$\mathbf{61.55 \pm 0.86}$	$\mathbf{44.46 \pm 0.17}$	$\mathbf{41.47 \pm 0.21}$	$\mathbf{29.85 \pm 0.07}$	-	-	-

5.3.3 Limited (structural) expressivity

The Weisfeiler Lehman graph isomorphism test (WL-1) is as follows: To decide whether two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic, for all nodes $v \in V_1$, assign a color c_v^0. Iteratively refine the color of each node by aggregating neighbor information until the color distribution converges by follow the update rule:

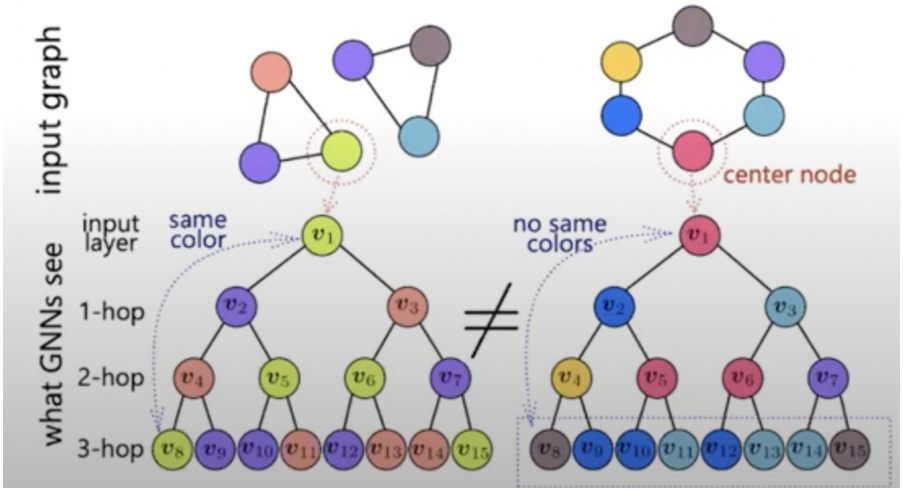
$$c_v^{t+1} = HASH\left(c_v^t, \{c_u^t\}_{u \in N_v}\right)$$

Repeat the process for G_2 and compare the color histograms. If the histograms are different, the two graphs are not isomorphic, whereas if they are the same, isomorphism is not guaranteed.

Xu et al. (2019) [26] demonstrated that the most expressive class of GNNs (MPNNs) are in fact limited in expressivity since they are as powerful as the WL-1, which is known to be unable to distinguish non-isomorphisms between simple graph structures (such as closed triangles). Moreover, over a discrete node features, maximal expressivity is achieved using injective aggregations (e.g. sums). In the case of continuous features, it has been proven that there is no best aggregator [16] and that to discriminate between multisets of size n, at least n aggregators are needed. The paper proposes combining the mean, standard deviation, maximum, and minimum aggregators and concatenating the results.

Higher order WL tests: To perform better at identifying graph isomorphisms, one can use higher-order k-WL tests (that operate on tuples of nodes) and, accordingly, generalize GNNs to higher-order GNNs. This can be done via analyzing the failure cases of 1-WL and modifying features, modifying message passing rules, or modifying the graph structure.

For example, since 1-WL and GNNs cannot detect closed triangles, one can augment nodes with random features/colors [41]. This enables a node to "see" itself in later hops and thereby distinguish a graph structure from another, as seen below:



(Image from https://geometricdeeplearning.com/)

Other feature augmentations approaches include positional encoding, as in the case of RP-GNNs [40] and P-GNNs [23], and counting subgraphs (e.g. rings) [18].

A particularly interesting line of work involves decoupling the original input graph from the computational graph and using higher-order structures to increase expressivity of GNNs. 1-2-3-GNNs [9] and topological/cellular message passing [10] are examples of such approaches. In both papers, instead of maintaining features purely for nodes, one introduces features to higher-dimensional structures such as edges and faces, and more generally, k-dimensional simplices. We explore topological/cellular message passing in more detail in the following section.

Topological/cellular message passing: Recall that standard GNNs suffer from capturing LRDs. Furthermore, they can only explicitly model pairwise interactions between nodes. The real world, however, typically involves higher-order interactions, such as the interactions between groups of people or the interactions between molecular rings. Topological message passing simplicial networks (MPSN) achieve this by augmenting graphs with higher-order topological structures known as simplicial complexes.

The authors demonstrate that MPSN is strictly more powerful than WL, and are as powerful as their newly defined Simplicial Weisfeiler Lehman (SWL) test.

For a vertex set V, a simplicial complex K is a collection of nonempty subsets of V that contains all singletons of V and is closed under the operation of taking subsets. A member $\sigma = \{v_0, \ldots, v_k\} \in K$ of cardinality $k+1$ is called a k-dimensional simplex (or just k-simplex). We note that 0-simplicies are vertices, 1-simplices are edges, and 2-simplices are triangles.

Using the notion of k-simplices, we can define boundary incidence relations, which describe which simplices are on the boundary of another simplex. For

example, $\{v_1\}$ and $\{v_2\}$ are on the boundary of edge $\{v_1, v_2\}$, so $\{v_1\} \prec \{v_1, v_2\}$ and $\{v_2\} \prec \{v_1, v_2\}$. Moreover, we can assign orientations to k-simplices by assigning a particular order $(v_0, \ldots, v_k)$.

The Simplicial WL Test is then similar to 1-WL test, except acting on all simplices $\sigma \in K$ rather than just acting on vertices. It is not hard to see that this is strictly more powerful than the 1-WL test. A key difference to note, however, is that in the case of SWL we may have different types of adjacent simplices. The four types are given as follows:

1. Boundary simplices $B(\sigma) = \{\tau | \tau \prec \sigma\}$
2. Co-boundary simplices $C(\sigma) = \{\tau | \sigma \prec \tau\}$
3. Lower-adjacencies $N_{\downarrow}(\sigma) = \{\tau | \exists \delta, \delta \prec \tau \wedge \delta \prec \sigma\}$
4. Upper-adjacencies $N_{\uparrow}(\sigma) = \{\tau | \exists \delta, \tau \prec \delta \wedge \sigma \prec \delta\}$

We can then define a multiset of colors for each type of adjacency: $c_B^t(\sigma), c_C^t(\sigma), c_{\downarrow}^t(\sigma), c_{\uparrow}^t(\sigma)$. The update rule for SWL is then:

$$c_{\sigma}^{(t+1)} = HASH\{c_{\sigma}^t, c_B^t(\sigma), c_C^t(\sigma), c_{\downarrow}^t(\sigma), c_{\uparrow}^t(\sigma)\}$$

The corresponding MPSN mechanism is given by:

$$m_B^{(t+1)}(\sigma) = AGG_{\tau \in B(\sigma)}(M_B(h_{\sigma}^t, h_{\tau}^t))$$

$$m_C^{(t+1)}(\sigma) = AGG_{\tau \in C(\sigma)}(M_C(h_{\sigma}^t, h_{\tau}^t))$$

$$m_{\downarrow}^{(t+1)}(\sigma) = AGG_{\tau \in N_{\downarrow}(\sigma)}(M_{\downarrow}(h_{\sigma}^t, h_{\tau}^t, h_{\sigma \cap \tau}^t))$$

$$m_{\uparrow}^{(t+1)}(\sigma) = AGG_{\tau \in N_{\uparrow}(\sigma)}(M_{\uparrow}(h_{\sigma}^t, h_{\tau}^t, h_{\sigma \cup \tau}^t))$$

$$h_{\sigma}^{(t+1)} = U\left(h_{\sigma}^t, m_B^{(t+1)}(\sigma), m_C^{(t+1)}(\sigma), m_{\downarrow}^{(t+1)}(\sigma), m_{\uparrow}^{(t+1)}(\sigma)\right)$$

Note that despite the increased expressivity of such models, the authors note that this does not readily translate into better downstream classification performance.

5.3.4 Diffusion-based generalizations of GNNs

In the spirit of modeling GNNs as diffusion processes on node features, yet unlike GraphCON, Graph Neural Diffusion (GRAND) considers a discretization of a partial differential equation in both time (layers of the GNN) and 2D space (layout of the graph) [6]. The goal here is to simultaneously tackle GNN depth, oversmoothing, and bottlenecks, while being stable to perturbations in the graph topology. Learnable spatial discretization corresponds to dynamic graph rewiring (as opposed to a fixed rewiring scheme e.g. in the case of SDRF), whereas time discretization gives rise to the neural network's layers.

GRAND starts with a generalization of the continuity equation – a partial differential equation in physics that expresses conservation law of a particular quantity (e.g. mass, energy, electric charge). This is a sufficient equation to determine the dynamics of any diffusion process. On the graph domain, the generalized continuity equation is given by (up to an overall minus sign):

$$\frac{\partial x}{\partial t} = div\left[G(x(t), t)\nabla x\right]$$

With initial condition $x(0)$ and where G is the diffusivity function. For simplicity, assume no boundary conditions. The role of the diffusivity function is to introduce anisotropy in the diffusion process. In particular, one can denote $G = diag(a(x_i(t), x_j(t), t))$, where a is a function denoting similarity between nodes i and j. In GRAND, the authors propose to share parameters between the layers to mitigate overfitting and decrease model size, so a does not explicitly depend on time, i.e. $a = a(x_i, x_j)$. After applying a finite difference discretization for the ∇ operator ($(\nabla x)_{ij} = x_j - x_i$), one obtains the following:

$$\frac{\partial}{\partial t}x(t) = \left(A(x(t)) - I\right)x(t) = \bar{A}(x(t))x(t)$$

Where $A(x) = (a(x_i, x_j))$ is the attention matrix. In GRAND, one chooses a to be the scaled dot product attention:

$$a(X_i, X_j) = softmax\left(\frac{(W_K X_i)^T W_Q X_j}{d_k}\right)$$

Note that the stability of this PDE corresponds to the GNN model's robustness to perturbations in the graph topology. A solution x(t) to a PDE is stable if $\forall\epsilon > 0, \exists\delta > 0$ such that for any solution $\hat{x}(t)$ such that $|x(0) - \hat{x}(0)| < \delta$ then $|x(t) - \hat{x}(t)| \leq \epsilon, \forall t \geq 0$. In other words, small changes in initial condition should always lead to small changes in time evolution. To satisfy the stability condition, consider two cases. If $\bar{A}$ is constant, then the solution is simply $x = x_0 e^{\bar{A}t}$, so it suffices to make $\bar{A}$ negative-semidefinite (so the solution does not blow up). If $\bar{A}(x(t))$ is not constant, the ODE is non-linear, which requires for stability that $\max_i x_i(0) \geq x_i(t) \geq \min_i x_i(0)$ for all $t \geq 0$ and that $\bar{A}(x)x$ is continuous in x. In other words, the largest component of the solution x_i does not increase in time and the smallest component does not decrease in time. These properties are automatically satisfied since $\bar{A}$ is Lipschitz and right-stochastic.

The GRAND paper proposes several variants, including the linear version GRAND-L (in which $\bar{A}$ is constant and constrained to be negative semi-definite), the non-linear version GRAND-NL (in which $\bar{A} = \bar{A}(x(t))$) and GRAND-NL-RW (nonlinear with graph rewiring). Note that the graph rewiring here consists of two stages: First, DIGL is used to densify the graph. Then, the attention weights learn which subsets of edges to use, resulting in dynamic graph rewiring.

Digression into numerical integration/ODE solving: Here, we consider a digression into discretization schemes in numerical methods literature. Recall that solving graph diffusion equations involves discretization, which gives

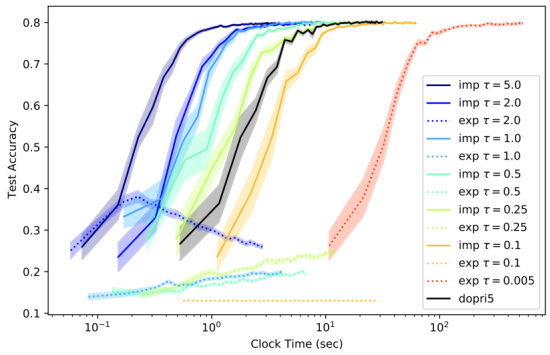
rise to a specific (generalized) GNN architecture. Several discretization schemes are available, including explicit or implicit, single-step or multi-step schemes. These approaches differ in their stability and accuracy of approximating solution of the continuous ODE. Whereas explicit schemes discretize forward in time (e.g. Forward Euler) and thus can be solved directly using a recurrence relation, implicit schemes discretize backward in time (e.g. Backward Euler) and thus require inverting a matrix to time evolve forward (thus being more expensive).

There is generally a tradeoff between the number of iterations K (i.e. depth of the GNN) and the discretization step size τ. Furthermore, explicit schemes are only stable for $0 < \tau < 1$ whereas implicit schemes are stable for any $\tau > 0$.

Additionally, linear multi-step schemes use intermediate fractional time steps to obtain higher-order numerical approximations (e.g. Runge Kutta). Multi-step schemes themselves can be explicit, implicit, or a combination of both.

Finally, there is the option of implementing adaptive step sizes to solve "stiff" ODEs – i.e. those whose solutions have "sharp" or sudden changes. Adaptive methods estimate the iteration error and adjust the step size accordingly.

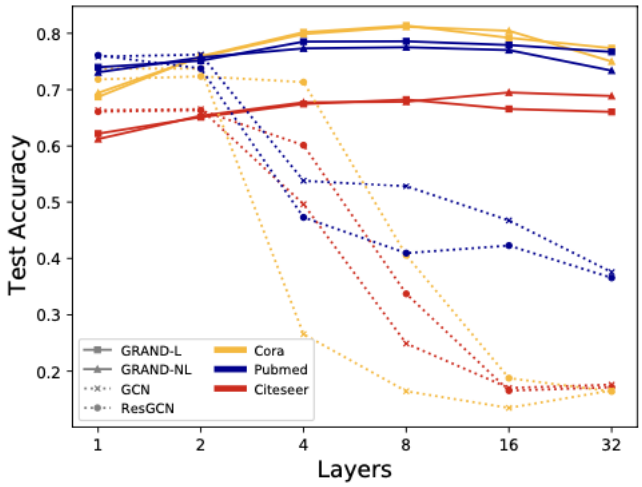
Back to GRAND, the final solution for the node embeddings (after passing the entire GNN) is given by $X(T) = X(0) + \int_0^T \frac{\partial X}{\partial t} dt$, where the dynamics $\frac{\partial X}{\partial t}$ is given by the diffusion equation previously, and $X(0) = \phi(X_{in})$, and the output $Y = \psi(X(T))$. Different types of GRAND architectures are achieved by choice of discretization scheme and choice of diffusivity G (or, equivalently, attention matrix A). For example, standard GNNs are equivalent to the explicit single-step Euler scheme, which is known to be unstable. In experiments, the authors verify that multi-step explicit schemes and implicit schemes yield significant improvements for downstream tasks. Furthermore, although no theory backs this finding, GRAND is shown to solve oversmoothing and perform well with many layers. An interesting line of work would be to compare GRAND with the GraphCON formalism to identify the theoretical basis for these performance benefits.



(Implicit schemes consistently yield greater stability/faster convergence compared to explicit schemes)

Random splits	CORA	CiteSeer	PubMed	Coathor CS	Computer	Photo	ogb-arxiv*
GCN	81.5 ± 1.3	$\mathbf{71.9 \pm 1.9}$	77.8 ± 2.9	91.1 ± 0.5	82.6 ± 2.4	91.2 ± 1.2	$\mathbf{72.17 \pm 0.33}$
GAT	81.8 ± 1.3	71.4 ± 1.9	$\mathbf{78.7 \pm 2.3}$	90.5 ± 0.6	78.0 ± 19.0	85.7 ± 20.3	$\mathbf{73.65 \pm 0.11}^{\dagger}$
GAT-ppr	81.6 ± 0.3	68.5 ± 0.2	76.7 ± 0.3	91.3 ± 0.1	$\mathbf{85.4 \pm 0.3}$	90.9 ± 0.3	N/A
MoNet	81.3 ± 1.3	71.2 ± 2.0	$\mathbf{78.6 \pm 2.3}$	90.8 ± 0.6	83.5 ± 2.2	91.2 ± 2.3	N/A
GS-mean	79.2 ± 7.7	71.6 ± 1.9	77.4 ± 2.2	91.3 ± 2.8	82.4 ± 1.8	91.4 ± 1.3	71.39 ± 0.16
GS-maxpool	76.6 ± 1.9	67.5 ± 2.3	76.1 ± 2.3	85.0 ± 1.1	N/A	90.4 ± 1.3	N/A
CGNN	81.4 ± 1.6	66.9 ± 1.8	66.6 ± 4.4	$\mathbf{92.3 \pm 0.2}$	80.29 ± 2.0	91.39 ± 1.5	58.70 ± 2.5
GDE	78.7 ± 2.2	71.8 ± 1.1	73.9 ± 3.7	91.6 ± 0.1	82.9 ± 0.6	$\mathbf{92.4 \pm 2.0}$	56.66 ± 10.9
GRAND-l (ours)	$\mathbf{83.6 \pm 1.0}$	$\mathbf{73.4 \pm 0.5}$	$\mathbf{78.8 \pm 1.7}$	$\mathbf{92.9 \pm 0.4}$	$\mathbf{83.7 \pm 1.2}$	$\mathbf{92.3 \pm 0.9}$	71.87 ± 0.17
GRAND-nl (ours)	$\mathbf{82.3 \pm 1.6}$	70.9 ± 1.0	77.5 ± 1.8	$\mathbf{92.4 \pm 0.3}$	82.4 ± 2.1	$\mathbf{92.4 \pm 0.8}$	71.2 ± 0.2
GRAND-nl-rw (ours)	$\mathbf{83.3 \pm 1.3}$	$\mathbf{74.1 \pm 1.7}$	78.1 ± 2.1	91.3 ± 0.7	$\mathbf{85.8 \pm 1.5}$	$\mathbf{92.5 \pm 1.0}$	$\mathbf{72.23 \pm 0.20}$

(GRAND achieves SOTA for numerous downstream tasks)



(GRAND overcomes oversmoothing)

6 Conclusion

In this survey we covered some of the important approaches used in graph representation learning - namely matrix factorization, sequence-based methods, hyperbolic embeddings, and graph neural networks. Within said approaches, we identified limitations and promising avenues of future research.

References

[1] Sami Abu-El-Haija, Bryan Perozzi, Rami Al-Rfou, and Alexander A Alemi. Watch your step: Learning node embeddings via graph attention. *Advances in neural information processing systems*, 31, 2018.

[2] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph fac-

torization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48, 2013.

[3] Seyed-Ahmad Ahmadi Nassir Navab Michael Bronstein Anees Kazi, Luca Cosmo. Differentiable graph module (dgm) for graph convolutional networks. 2020.

[4] Niki Parmar Jakob Uszkoreit Llion Jones Aidan N. Gomez Lukasz Kaiser Illia Polosukhin Ashish Vaswani, Noam Shazeer. Attention is all you need.

[5] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. *Advances in neural information processing systems*, 14, 2001.

[6] Maria Gorinova Stefan Webb Emanuele Rossi Michael M. Bronstein Benjamin Paul Chamberlain, James Rowbottom. Grand: Graph neural diffusion. 2021.

[7] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900, 2015.

[8] Shengjie Luo Shuxin Zheng Guolin Ke Di He Yanming Shen Tie-Yan Liu Chengxuan Ying, Tianle Cai. Do transformers really perform bad for graph representation? 2021.

[9] Matthias Fey William L. Hamilton Jan Eric Lenssen Gaurav Rattan Martin Grohe Christopher Morris, Martin Ritzert.

[10] Yu Guang Wang Nina Otter Guido Montúfar Pietro Liò Michael Bronstein Cristian Bodnar, Fabrizio Frasca. Weisfeiler and lehman go topological: Message passing simplicial networks.

[11] Rémi Gribonval David K Hammond, Pierre Vandergheynst. Wavelets on graphs via spectral graph theory. 2009.

[12] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[13] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on knowledge and data engineering*, 19(3):355–369, 2007.

[14] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[15] Linton C Freeman, Stephen P Borgatti, and Douglas R White. Centrality in valued graphs: A measure of betweenness based on network flow. *Social networks*, 13(2):141–154, 1991.

[16] Dominique Beaini Pietro Liò Petar Veličković Gabriele Corso, Luca Cavalleri. Principal neighbourhood aggregation for graph nets. 2020.

[17] Octavian-Eugen Ganea, Gary Bécigneul, and Thomas Hofmann. Hyperbolic neural networks. 2018.

[18] Stefanos Zafeiriou Michael M. Bronstein Giorgos Bouritsas, Fabrizio Frasca. Improving graph neural network expressivity via subgraph isomorphism counting. 2020.

[19] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. In *Linear algebra*, pages 134–151. Springer, 1971.

[20] Caglar Gulcehre, Misha Denil, Mateusz Malinowski, Ali Razavi, Razvan Pascanu, Karl Moritz Hermann, Peter Battaglia, Victor Bapst, David Raposo, Adam Santoro, and Nando de Freitas. Hyperbolic attention networks. 2018.

[21] Michael U Gutmann and Aapo Hyvärinen. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of machine learning research*, 13(2), 2012.

[22] Banjamin Paul Chamberlain Xiaowen Dong Michael M. Bronstein Jake Topping, Francesco Di Giovanni. Understanding over-squashing and bottlenecks on graphs via curvature. 2021.

[23] Jure Leskovic Jiaxuan You, Rex Ying. Position-aware graph neural networks. 2019.

[24] Stephan Günnemann Johannes Gasteiger, Stefan Weißenberger. Diffusion improves graph learning. 2019.

[25] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[26] Jure Leskovec Stefanie Jegelka Keyulu Xu, Weihua Hu. How powerful are graph neural networks. 2019.

[27] Joseph B Kruskal and Myron Wish. *Multidimensional scaling*. Number 11. Sage, 1978.

[28] Qi Liu, Maximilian Nickel, and Douwe Kiela. Hyperbolic graph neural networks. 2019.

[29] Aleix M Martinez and Avinash C Kak. Pca versus lda. *IEEE transactions on pattern analysis and machine intelligence*, 23(2):228–233, 2001.

[30] Taco Cohen Petar Velickovic Michael M. Bronstein, Joan Bruna. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. 2021.

[31] Mark EJ Newman. A measure of betweenness centrality based on random walks. *Social networks*, 27(1):39–54, 2005.

[32] Maximilian Nickel and Douwe Kiela. Learning continuous hierarchies in the lorentz model of hyperbolic geometry. 2018.

[33] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114, 2016.

[34] Wei Peng, Tuomas Varanka, Abdelrahman Mostafa, Henglin Shi, and Guoying Zhao. Hyperbolic deep neural networks: A survey. 2021.

[35] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[36] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[37] Arantxa Casanova Adriana Romero Pietro Lio Yoshua Bengio Petar Velickovic, Guillem Cucurull. Graph attention networks. 2018.

[38] Matthew C. Overlan Razvan Pascanu Oriol Vinyals Charles Blundell Petar Velickovic, Lars Buesing. Pointer graph networks. 2020.

[39] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[40] Vinayak Rao Bruno Ribeiro Ryan L. Murphy, Balasubramaniam Srinivasan. Relational pooling for graph representations. 2019.

[41] Hisashi Kashima Ryoma Sato, Makoto Yamada. Random features strengthen graph neural networks. 2020.

[42] James Rowbottom Siddhartha Mishra Michael M. Bronstein T. Konstantin Rusch, Benjamin P. Chamberlain. Graph-coupled oscillator networks. 2022.

[43] Joshua B Tenenbaum, Vin de Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[44] Kuan-Chieh Wang Max Welling Richard Zemel Thomas Kipf, Ethan Fetaya. Neural relational inference for interacting systems. 2018.

[45] Max Welling Thomas N. Kipf. Semi-supervised classification with graph convolutional networks. 2017.

[46] Eran Yahav Uri Alon. On the bottleneck of graph neural networks and its practical implications. 2020.

[47] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[48] Ziwei Liu Sanjay E. Sarma Michael M. Bronstein Justin M. Solomon Yue Wang, Yongbin Sun. Dynamic graph cnn for learning on point clouds. 2019.