Graph Machine Learning

Lecture 9 : Graph Transformer

Oct 17 2026

Xavier Bresson

https://x.com/xbresson

Department of Computer Science

National University of Singapore (NUS)



Course lectures

- Introduction to Graph Machine Learning
- Part 1: GML without feature learning (before 2014)
  - Introduction to Graph Science
  - Graph Analysis Techniques without Feature Learning
    - Graph clustering
    - Graph SVM
    - Recommendation on graphs
    - Graph-based visualization
- Part 2 : GML with shallow feature learning (2014-2016)
  - Shallow graph feature learning
- Part 3 : GML with deep feature learning, a.k.a. GNNs (after 2016)
  - Graph Convolutional Networks (spectral and spatial)
  - Weisfeiler-Lehman GNNs
  - Graph Transformer
  - Graph generation & molecular science
  - Material design
  - Integrating GNNs and LLMs
  - Deep Learning for Combinatorial optimization

Outline

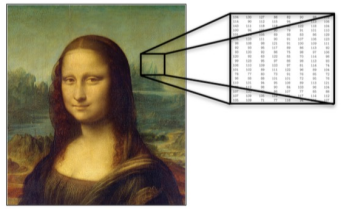
- Review of graph network architectures
- Graph positional encoding
- Message-passing and over-squashing
- Graph Transformer
- Graph ViT / Hierarchical Transformer
- Conclusion

Outline

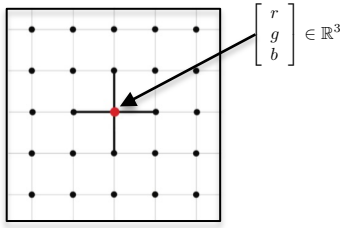
- Review of graph network architectures
- Graph positional encoding
- Message-passing and over-squashing
- Graph Transformer
- Graph ViT / Hierarchical Transformer
- Conclusion

Deep learning across data structures

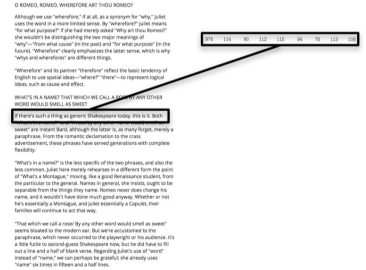
- To enable DL + X, we need three key ingredients :
  - Data, compute / GPU and neural network
- Different architectures exploit different data structures :
  - CNNs for regular grids, RNNs and Transformers for sequences and sets.
  - DL libraries PyTorch[1], TensorFlow[2] are GPU-optimized for dense matrix operations.
  - Graph Neural Networks (GNNs) are more universal and broadly applicable architectures.
  - Graph computations are sparse and irregular, making GPU execution less efficient / GPUs are not optimized for sparse matrix operations.



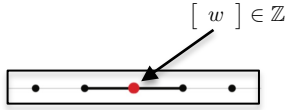
Computer Vision (CV)



grid



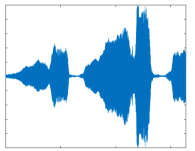
Natural Language Processing (NLP)



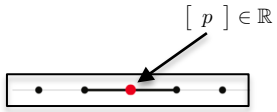
sequence



Graph-structure data



Speech Recognition (SR)



sequence

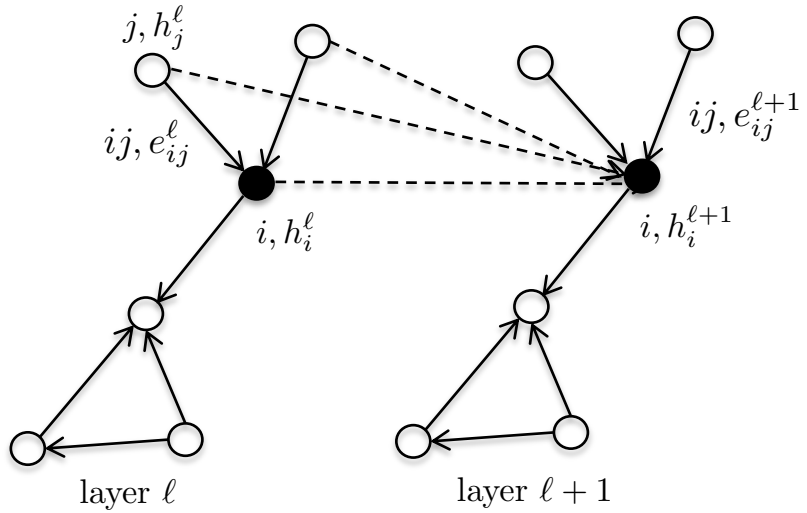
[1] Paszke et-al, Automatic differentiation in pytorch, 2017
[2] Abadi et-al, TensorFlow: a system for Large-Scale machine learning, 2016

Graph neural networks

- What is a graph neural network (GNN) ?
- A learnable compositional function that maps a graph and its features to continuous vector representations for downstream tasks with permutation-equivariant outputs for node-/edge-level prediction, and permutation-invariant outputs for graph-level prediction (defined on the next slide).
- Typically formulated as message passing[1], and equivalently, we can view it as a discrete nonlinear diffusion process[2] on graphs :

$$\text{Node-update}: \quad h_i^{\ell+1} = f_{\text{node}}\big(h_i^\ell, \{h_j^\ell, e_{ij}^\ell : j \in \mathcal{N}_i\}\big) \in \mathbb{R}^d$$

$$\text{Edge-update}: \quad e_{ij}^{\ell+1} = f_{\text{edge}}\big(e_{ij}^\ell, h_i^\ell, h_j^\ell\big) \in \mathbb{R}^d$$



[1] Gilmer-etal, Neural message passing for quantum chemistry, 2017
[2] Bresson-Laurent, Residual gated graph convnets, 2017

Permutation equivariance and invariance

- **Symmetry : GNN layers are permutation-equivariant, graph-level readouts are permutation-invariant.**
- **Definition of a permutation-equivariant function :**

$$f_{\text{equi}}(Ph, PAP^\top) = P\, f_{\text{equi}}(h, A),\ h \in \mathbb{R}^{n\times d},\ A \in \mathbb{R}^{n\times n},$$
$$\forall P \in \mathcal{P}_n \ \text{ permutation matrix}$$

meaning that relabeling nodes before applying f_{equi} is equivalent to re-labeling the output afterward.

- **Definition of a permutation-invariant function :**

$$f_{\text{inv}}(Ph, PAP^\top) = f_{\text{inv}}(h, A),\ \forall P \in \mathcal{P}_n$$

meaning that the output is unchanged by any re-labeling of the nodes.

Permutation equivariance

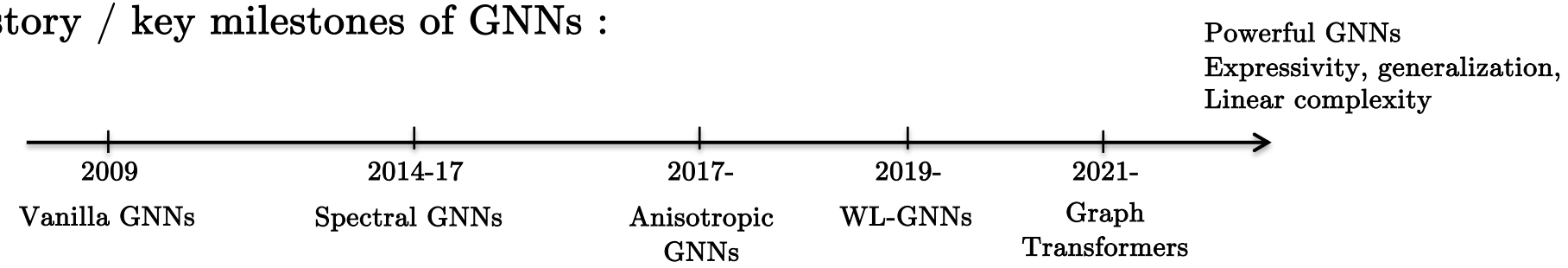
$$\begin{array}{ccc} h & \xrightarrow{f_{\text{equi}}} & f_{\text{equi}}(h) \\ \downarrow P & & \downarrow P \\ Ph & \xrightarrow{f_{\text{equi}}} & P f_{\text{equi}}(h) \end{array}$$

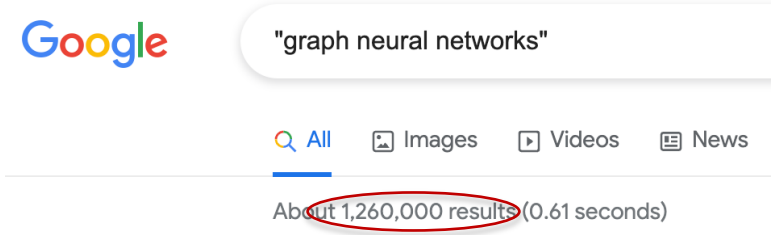
Permutation invariance

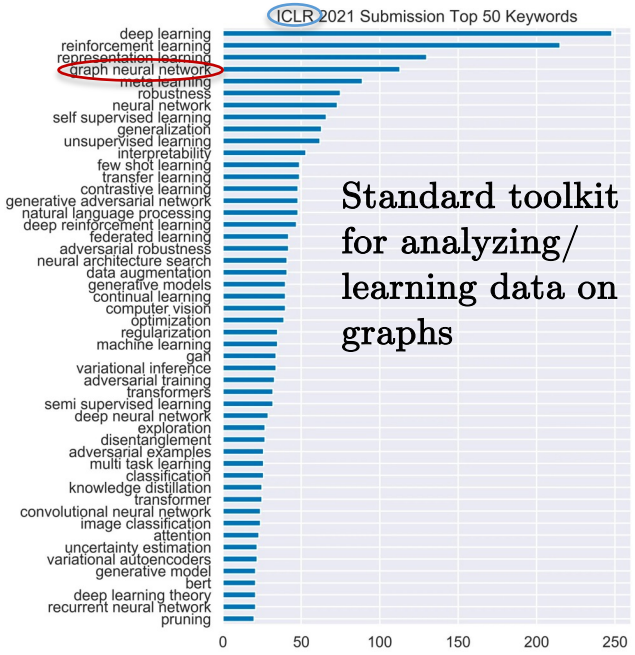
$$\begin{array}{ccc} h & \xrightarrow{f_{\text{inv}}} & f_{\text{inv}}(h) \\ \downarrow P & & \downarrow P \\ Ph & \xrightarrow{f_{\text{inv}}} & f_{\text{inv}}(h) \end{array}$$

Evolution of graph networks

- A brief history / key milestones of GNNs :



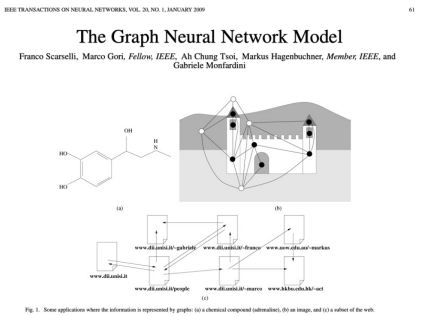




Standard toolkit for analyzing/ learning data on graphs

Early GNNs : Recurrent fixed-point model

- Contributions
  - First neural architecture for graph-structured data[1] : Shared local transition function with permutation equivariant-/invariant layer, independent of graph size, local receptive fields and weight sharing.
  - Linear cost per propagation step: $O(n+n_E)$.
  - Simplified update equation : $h_i^{\ell+1} = f_{\text{GNN}}\big(h_i^\ell, \{h_j^\ell : j \in \mathcal{N}_i\}\big) \in \mathbb{R}^d$
    $$= \sum_{j \in \mathcal{N}_i} \sigma(U x_i + V h_j^\ell) \in \mathbb{R}^d,$$


IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 20, NO. 1, JANUARY 2009 61

The Graph Neural Network Model

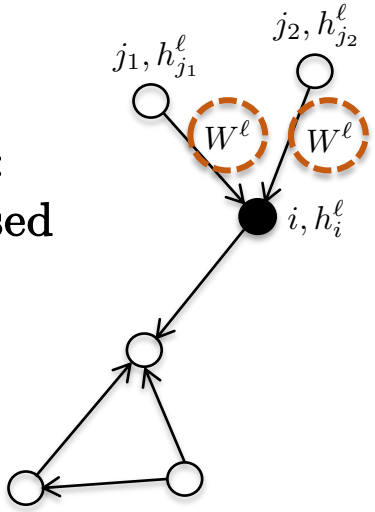
Franco Scarselli, Marco Gori, *Fellow, IEEE*, Ah Chung Tsoi, Markus Hagenbuchner, *Member, IEEE*, and Gabriele Monfardini

- Limitations
  - Basically a recurrent neural network on graphs : Each node state is recursively updated from its neighbors on the graph.
  - Recurrent fixed-point model requiring iterative convergence.
  - Vanishing gradients and contractive dynamics limit long-range information propagation.

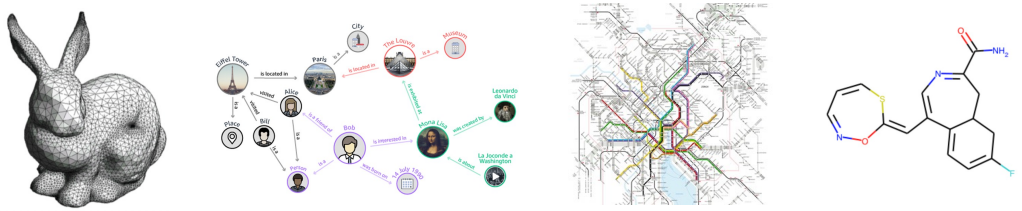
[1] Scarselli-etal, The Graph Neural Network Model, 2009

Spectral GNNs : Graph Fourier filters

- Contributions[1,2,3]
  - Leverage graph spectral theory to define convolution on graphs.
  - Polynomial filters provide stable, robust to perturbation, exact K-hop receptive fields.
  - Linear complexity O(K(n+n_E)) = O(n).
  - GCN[3] is the most popular GNN technique.
- Limitation
  - Isotropic aggregation: Neighbors are processed by the same kernel.



$$
\begin{aligned}
h^{\ell+1} &= \eta(w^\ell *_G h^\ell) \in \mathbb{R}^{n\times d} \\
&= \eta(\Phi \hat{w}^\ell(\Lambda)\Phi^T h^\ell),\ L = \Phi\Lambda\Phi^T \in \mathbb{R}^{n\times n} \\
&= \eta(\hat{w}^\ell(L)h^\ell) \quad \text{spectral convolution} \\
&= \eta(\sum_{k=0}^{K-1} T_k(L)h^\ell W_k^\ell) \quad \text{Chebyshev layer}^{[2]} \\
&= \eta(D^{-1/2}AD^{-1/2}h^\ell W^\ell) \quad \text{1st order approximation} \\
h_i^{\ell+1} &= \eta(\ \frac{1}{\sqrt{d_i}} \sum_{j\in\mathcal{N}_i} \frac{1}{\sqrt{d_j}} W^\ell h_j^\ell\ ) \in \mathbb{R}^d \quad \text{GCN layer}^{[3]}
\end{aligned}
$$



Most real-world data is anisotropic.

[1] Bruna-Zaremba-Szlam-LeCun, Spectral networks and locally connected networks on graphs, 2013
[2] Defferrard-Bresson-Vandergheynst, Convolutional neural networks on graphs with fast localized spectral filtering, 2016
[3] Kipf-Welling, Semi-supervised classification with graph convolutional networks, 2017

Anisotropic GNNs

- Contributions
  - Design anisotropic, permutation-equivariant aggregation mechanisms that treat neighbors differently.
  - GatedGCN[1] learns edge gates inspired by anisotropic diffusion[2], GAT[3] uses softmax attention[4], and gauge-equivariant mesh CNN[5] defines anisotropic directional kernels.
  - Stable, robust, interpretable, linear complexity $O(n+n_E)$ cost per layer.
- Limitation : Bounded expressivity of local message passing and over-squashing (explained later).

$$e_{ij}^{\ell+1} = e_{ij}^{\ell} + \mathrm{ReLU}\Big(\mathrm{BN}\big(V_1^{\ell} h_i^{\ell-1} + V_2^{\ell} h_j^{\ell-1} + V_3^{\ell} e_{ij}^{\ell}\big)\Big) \in \mathbb{R}^d$$

$$w_{ij}^{\ell+1} = \frac{\sigma(e_{ij}^{\ell+1})}{\sum_{j' \in \mathcal{N}_i} \sigma(e_{ij'}^{\ell+1}) + \varepsilon} \in \mathbb{R}^d$$

$$h_i^{\ell+1} = h_i^{\ell} + \mathrm{ReLU}\Big(\mathrm{BN}\Big(W_1^{\ell} h_i^{\ell} + \sum_{j \in \mathcal{N}_i} w_{ij}^{\ell+1} \odot W_2^{\ell} h_j^{\ell}\Big)\Big) \in \mathbb{R}^d$$

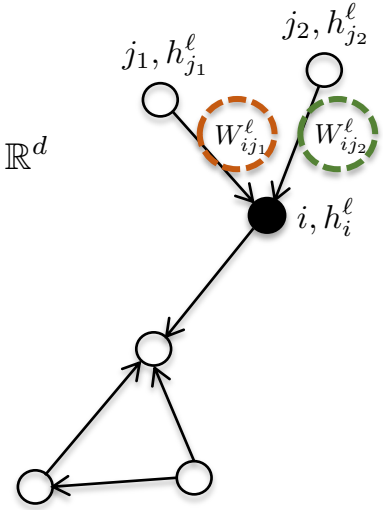
GatedGCNs[1]

$$e_{ij}^{k,\ell} = \mathrm{LeakyReLU}\left((a^{k,\ell})^{\top}\left[W^{k,\ell} h_i^{\ell} \| W^{k,\ell} h_j^{\ell}\right]\right) \in \mathbb{R}$$

$$\alpha_{ij}^{k,\ell} = \frac{\exp(e_{ij}^{k,\ell})}{\sum_{j' \in \mathcal{N}_i \cup \{i\}} \exp(e_{ij'}^{k,\ell})} \in \mathbb{R}_+$$

$$h_i^{\ell+1} = \|_{k=1}^{K} \eta\Big(\sum_{j \in \mathcal{N}_i \cup \{i\}} \alpha_{ij}^{k,\ell} W^{k,\ell} h_j^{\ell}\Big) \in \mathbb{R}^d$$

GAT[1]



[1] Bresson-Laurent, Residual gated graph convnets, 2017
[2] Velickovic-Bengio-etal, Graph Attention Networks, 2017
[3] DeHaan-Weiler-Cohen-Welling, Gauge equivariant mesh CNNs: Anisotropic convolutions on geometric graphs, 2020

Expressivity limitation

- Standard GNNs s.a. GCN, GAT and GatedGCN have fundamental expressivity limits :
  - They fail to distinguish simple non-isomorphic graphs[1,2], i.e. at most as powerful as 1-WL[3].
  - They are unable to count elementary sub-structures like motifs, cycles and cliques[4].

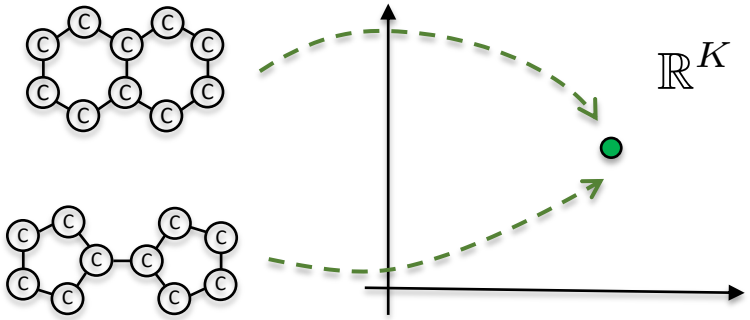


Fig 1. Vectorial graph representation

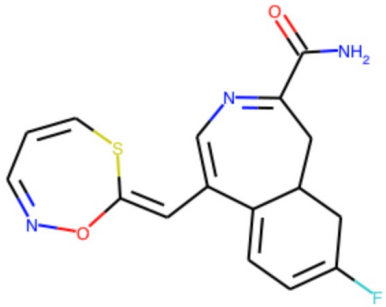


Fig 2. Graph with cycles (rings)

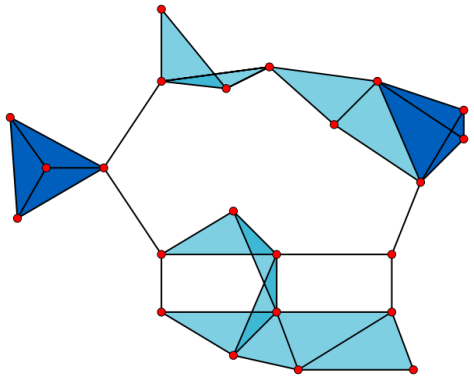


Fig 3. Graph with 19 × 3-vertex cliques (light and dark blue triangles) and 2 × 4-vertex cliques (dark blue areas)

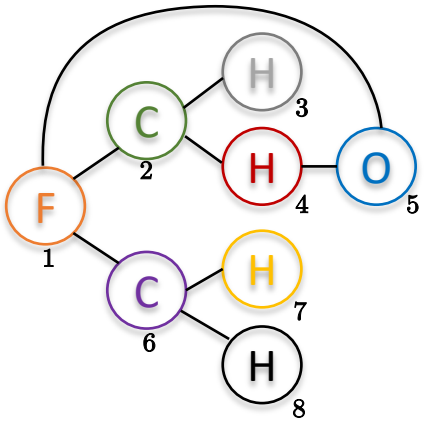
[1] Xu-etal, How powerful are graph neural networks? 2019
[2] Morris-etal, Weisfeiler and leman go neural: Higher-order graph networks, 2019
[3] Weisfeiler-Lehman, A reduction of a graph to a canonical form and an algebra arising during this reduction, 1968
[4] Chen-etal, Can graph neural networks count substructures? 2020

Graph isomorphism and WL test

- Graph isomorphism : Two graphs are isomorphic if there exists a node permutation that preserves node neighboors, i.e. $A_2 = PA_1P^T$.
- Determining whether two graphs are isomorphic is NP-intermediate. It is not known if a polynomial time algorithm exists, or the problem is NP-hard.
- The Weisfeiler–Leman (1-WL) test[1] provides a necessary but not sufficient condition for isomorphism, or equivalently, a sufficient test for non-isomorphism.
- Iteratively update each node color using its current color and the multiset of its neighbors' colors :

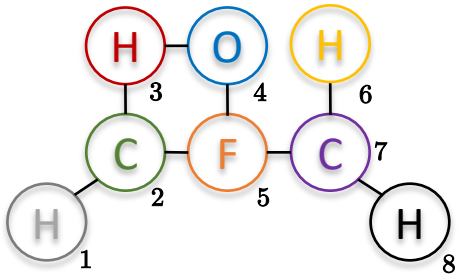
$$c_i^{t+1} = f_{\mathrm{WL}}\left(c_i^t, \{c_j^t\}_{j \in \mathcal{N}_i}\right), \; t = 0, 1, \ldots$$

where $\{a_j\}_j$ is a multiset, i.e. a set of unordered and repetitive elements



Graph 1

⇕ Isomorphic graphs

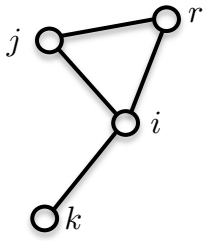


Graph 2

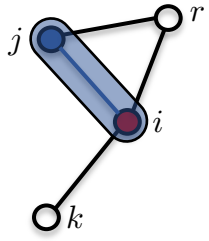
[1] Weisfeiler, Lehman, A reduction of a graph to a canonical form and an algebra arising during this reduction, 1968

Beyond 1-WL : Higher-order GNNs

- GIN[1] matches the discriminative power of 1-WL[2] under injective aggregation and readout.
- Higher-order k-WL : Original test uses pairs/2-tuple of nodes. To improve expressivity power of WL test, higher-order interactions between nodes can be generalized to k-tuples of nodes with $k \geq 3$ to create more colors.
- Higher-order equivariant GNNs[3] can match theoretical higher-order WL tests, but at increased computational cost :
  - These networks require $O(n^k)$ memory/speed complexities, with at least $k = 3$ to be more powerful than GIN, which means $O(n^3)$ and thus not practical.
  - 3-WL/Ring/2-FGNN GNNs[4,5,6] have $O(n^2)$-memory but $O(n^3)$-speed complexities.
  - Expressivity does not necessarily imply generalization[7].

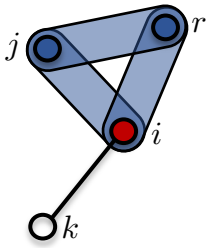


$\mathcal{G} = (V, E)$



Edges $= \{i, j\}, \{i, r\}, \{i, k\}$

2-tuple of nodes



Hyper-edges $= \{i, j, r\}, \{i, k, r\}, \{i, j, k\}$

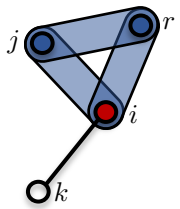
3-tuple of nodes

[1] Xu, Hu, Leskovec, Jegelka, How powerful are graph neural networks?, 2019
[2] Weisfeiler, Lehman, A reduction of a graph to a canonical form and an algebra arising during this reduction, 1968
[3] Maron, Ben-Hamu, Shamir, Lipman, Invariant and equivariant graph networks, 2019
[4] Maron, Ben-Hamu, Serviansky, Lipman, Provably powerful graph networks, 2019
[5] Chen, Villar, Chen, Bruna, On the equivalence between graph isomorphism testing and function approximation with gnns, 2019
[6] Azizian, Lelarge, Expressive power of invariant and equivariant graph neural networks, 2020
[7] Dwivedi, Joshi, Laurent, Bengio, Bresson, Benchmarking graph neural networks, 2020

Outline

- Review of graph network architectures
- Graph positional encoding
- Message-passing and over-squashing
- Graph Transformer
- Graph ViT / Hierarchical Transformer
- Conclusion

Higher-order GNNs with positional information



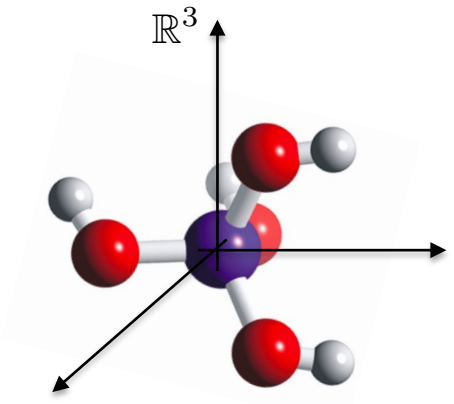
Hyper-edges $=\{i,j,r\}, \{i,k,r\}, \{i,j,k\}$

- How to improve the limited expressivity of standard MP-GNNs ?
  - Use higher-order node interactions (k-tuples with $k > 2$) with WL-GNNs[1,2,3,4].
    - Theorem[5] (universal approximator) : Any continuous permutation-invariant function can be approximated by WL-GNNs, but, in the worst case, a sufficient construction may require tensor order up to $k = \text{poly}(n) = n(n-1)/2$.
    - But these networks are costly, with at least $O(n^3)$ memory for higher expressivity.
  - Add symmetry-breaking positional or structural information, i.e. node IDs.
    - Theorem[6] : MP-GNNs are provable more expressive than the standard 1-WL test when node IDs.
    - Theorem[7] : MP-GNNs are Turing-complete if depth $d \geq \delta_G$ layers (graph diameter), width is unbounded, and nodes have unique IDs.

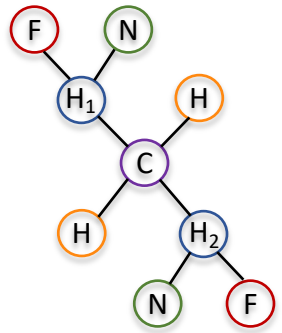
[1] Maron, Ben-Hamu, Shamir, Lipman, Invariant and equivariant graph networks, 2019
[2] Maron, Ben-Hamu, Serviansky, Lipman, Provably powerful graph networks, 2019
[3] Chen, Chen, Villar, Bruna, Can graph neural networks count substructures? 2020
[4] Azizian, Lelarge, Expressive power of invariant and equivariant graph neural networks, 2020
[5] Maron Fetaya, Segol, Lipman, On the universality of invariant networks, 2019
[6] Murphy, Srinivasan, Rao, Ribeiro, Relational pooling for graph representations, 2019
[7] Loukas, What graph neural networks cannot learn: depth vs width, 2019

Geometric positional vs. structural encodings

- Positional Encoding (PE) :
  - Encodes the position of a node relative to a global / geometric reference system.
  - Examples : 2D/3D coordinates / rotations.
- Structural Encoding (SE) :
  - Encodes the intrinsic structural role or neighborhood of a node in the graph, independent of any external coordinate system.
  - Nodes with structurally equivalent neighborhoods may receive identical SEs.
  - Examples : Node degree, motif / graphlet counts



Positional Encoding
with 3D coordinates



Structural Encoding
with a 3-node graphlet

$SE(H_1) = SE(H_2)$
for a 3-node graphlet, since
H1 and H2 have the same
local structure {F,N,C}.

Design principles

- There is no universally optimal PE and SE.
- The appropriate choice depends on the graph family and downstream task.
- What properties a meaningful PE/SE must satisfy ?
  - Permutation equivariance : Relabeling nodes relabel their positional features accordingly.
  - Discriminability : Distinct structural nodes have unique representation.
  - Distance sensitivity : Nearby nodes have similar positional features, while far away nodes should be dissimilar.
  - Structural sensitivity : Structurally similar nodes have similar structural features, while structurally different nodes should be dissimilar / distinguished.
  - Scalability : Construction and storage should remain practical for large graphs, i.e. O(n), and possibly exploiting the graph structure sparsity.

Laplacian Eigenvectors as PEs

- Laplacian eigenvectors[1] :
  - Spectral techniques that embed graphs into a Euclidean space.
  - Low-frequency eigenvectors capture smooth, global graph connectivity patterns.
  - They are obtained from the eigen-decomposition (EVD) of the graph Laplacian.
  - Computational complexity is costly $O(n^3)$, but sparse/approximate methods improve scalability[2].
- Laplacian eigenvectors as PEs[3] :
  - Hybrid positional and structural node encodings, equivariant to node relabeling.
  - Unique and distance-sensitive : Two nodes far away on graph have dissimilar PEs, and inversely.
  - LapPEs are the graph generalizations of standard Transformer's PE[4].
  - Capture meaningful global positions, but have sign and eigenspace-basis ambiguities.

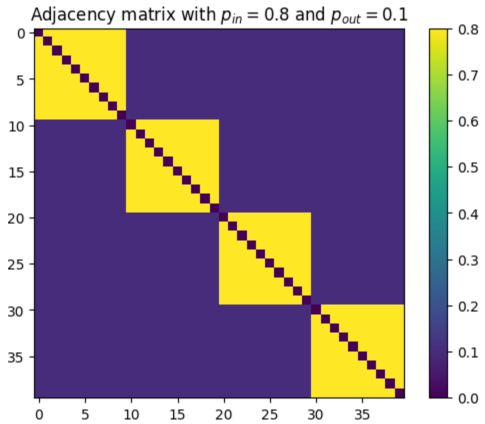
$$L = \mathrm{I} - D^{-1/2} A D^{-1/2} = \Phi \Lambda \Phi^T \in \mathbb{R}^{n \times n}$$

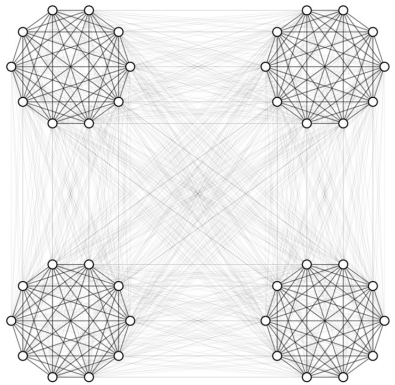
where $\Phi \in \mathbb{R}^{n \times n}$ are the eigenvectors of the Laplacian.

[1] Belkin-Niyogi, Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering, 2001
[2] Fowlkes-etal, Spectral grouping using the nystrom method, 2004
[3] Dwivedi-Joshi-Laurent-Bengio-Bresson, Benchmarking graph neural networks, 2020
[4] Vaswani-etal, Attention is all you need, 2017

Limitations of Laplacian PEs

- Eigenspace-basis ambiguity / repeated eigenvalues :
  - Laplacian eigenvectors have additional symmetries when the Laplacian has multiple eigenvectors with the same eigenvalue.
  - Example 1 : A graph with three connected components has three 3 zero eigenvalues, i.e. $\lambda_1 = \lambda_2 = \lambda_3 = 0$.
  - Example 2 : A stochastic block model with 4 balanced communities with the same in-/out- probabilities, has 3 eigenvectors with the same eigenvalues $\lambda_2 = \lambda_3 = \lambda_4 = 4$.





Limitations of Laplacian PEs

- Continued[1] :
  - Let $L = \Phi\Lambda\Phi^T$ be the eigendecomposition of the graph Laplacian.
  - If eigenvalue λ has multiplicity m, let $\Phi_\lambda \in \mathbb{R}^{N \times m}$ contain an orthonormal basis of its eigenspace $E_\lambda = \{u \in R^N : Lu = \lambda u\}$.
  - For any orthogonal matrix $Q \in O(m)$ (rotations and reflections), the matrix $U_\lambda = \Phi_\lambda Q$ is another basis of the same eigenspace.
  - Indeed, $LU_\lambda = L(\Phi_\lambda Q) = (L\Phi_\lambda)Q = \lambda\Phi_\lambda Q = \lambda U_\lambda$, i.e. U_λ are also eigenvectors with the eigenvalue λ.
  - Hence, LapPEs may be arbitrarily rotated or reflected inside a repeated-eigenvalue space.
  - Equivalently, LapPEs are unique up to an orthogonal change of basis inside each eigenspace.
  - This ambiguity motivates basis-invariant spectral encodings[2].
  - However, in practice, this ambiguity does not appear to be critical, based on personal experiments.

[1] Horn-Johnson, Matrix Analysis, 2012
[2] Lim-etal, Sign and Basis Invariant Networks for Spectral Graph Representation Learning, 2023

Limitations of Laplacian PEs

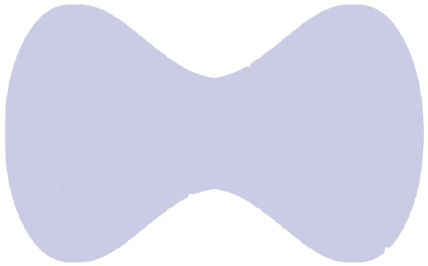
- Sign ambiguity[1] :
  - LapPEs have also sign symmetries, i.e. each normalized eigenvector is defined up to sign : $\pm u_k \in \mathbb{R}^n$, $1 \leq k \leq K$.
  - Hence, LapPEs are unique up to to their sign values.
  - For K eigenvectors, there are 2^K possible independent sign configurations.
- A common mitigation is sign augmentation during the training[2,3] :
  - Simply flip the signs of the eigenvectors during training. This makes the network learn to be independent to eigenvector signs.
- In practice, we select $K \ll n$, and therefore 2^K is much smaller than n! (the number of possible ordering of the nodes), which means smaller amount of ambiguities to be resolved by the network.
- Despite these limitations, LapPEs remain widely used in practice as they offer good performance.
- LapPEs are used in Graph Transformer[2], Directed-GNN[4], SAN[5].

[1] Horn-Johnson, Matrix Analysis, 2012
[2] Dwivedi-Bresson, A generalization of transformer networks to graphs, 2020
[3] Dwivedi-Joshi-Laurent-Bengio-Bresson, Benchmarking graph neural networks, 2020
[4] Beani-etal, Directional graph networks, 2020
[5] Kreuzer-etal, Rethinking Graph Transformers with Spectral Attention, 2021

Structural embeddings

- Diffusion-based SEs :
  - Diffusion processes capture multi-scale graph structure.
  - Examples of graph diffusion include heat diffusion kernels[1,2], random walks s.a pageRank[3].
  - Full pairwise diffusion requires $O(n^2)$ for computation.
  - However, it is done once, as a pre-processing step.
- The probability that a random walk starting at node i returns to j after k steps is

$$(P^k)_{ii} = \Pr(X_k = i \mid X_0 = i), \quad \text{with } P = D^{-1}A \text{ the random walk matrix}$$





[1] Kondor, Vert, Diffusion kernels, 2004
[2] Talmon, Cohen, Gannot, Coifman, Diffusion maps for signal processing, 2013
[3] Page, Brin, Motwani, Winograd, The PageRank citation ranking: Bringing order to the web, 1999

Random walk SEs

- Diagonal K-step random walk return probabilities define the random walk SEs (RWSE)[1] :
  - Compact node encoding with O(n.K) memory.
  - RWSE has no eigenvector sign/basis ambiguity, unlike LapPE[2], i.e. the network is not required to learn additional invariance.
  - However, RWSE does not guarantee unique node ID.
  - Nodes with identical return-probability signatures remain indistinguishable, meaning that the node IDs are only unique if each node has a unique K-hop neighborhood for a sufficient large K.

$$p_i^{\text{RWSE}} = \begin{bmatrix} (P)_{ii} \\ (P^2)_{ii} \\ \vdots \\ (P^K)_{ii} \end{bmatrix} \in \mathbb{R}^K, \qquad \text{with } P = AD^{-1} \in \mathbb{R}^{n \times n}$$

[1] Dwivedi-Luu-Laurent-Bengio-Bresson, Graph neural networks with learnable structural and positional representations, 2021
[2] Belkin-Niyogi, Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering, 2001

Decoupling structural and PEs / SEs

- **Standard MP-GNN inject the positional information at the input layer[1] :**

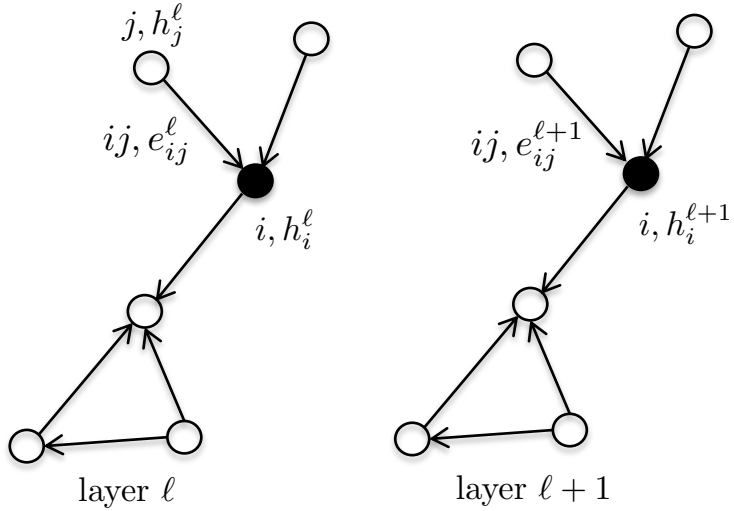
$$\textbf{Node-update} : h_i^{\ell+1} = f_h(h_i^\ell, \{h_j^\ell\}_{j\in\mathcal{N}_i}, e_{ij}^\ell)$$

$$\textbf{Edge-update} : e_{ij}^{\ell+1} = f_e(h_i^\ell, h_j^\ell, e_{ij}^\ell)$$

$$\text{with } h_i^{\ell=0} = \text{LE}(\text{Concat}(h_i^0, p_i))$$

graph PE

Input node feature



- **We can decouple and learn structural and positional encodings separately[2] :**

$$\textbf{Node-update} : h_i^{\ell+1} = f_h(h_i^\ell, p_i^\ell, \{h_j^\ell, p_j^\ell\}_{j\in\mathcal{N}_i}, e_{ij}^\ell)$$

$$\textbf{Edge-update} : e_{ij}^{\ell+1} = f_e(h_i^\ell, h_j^\ell, e_{ij}^\ell)$$

$$\textbf{PE/SE-update} : p_i^{\ell+1} = f_p(p_i^\ell, \{p_j^\ell\}_{j\in\mathcal{N}_i})$$

[1] Dwivedi, Joshi, Laurent, Bengio, Bresson, Benchmarking graph neural networks, 2020
[2] Dwivedi, Luu, Laurent, Bengio, Bresson, Graph neural networks with learnable structural and positional representations, 2021

Outline

- Review of graph network architectures
- Graph positional encoding
- **Message-passing and over-squashing**
- Graph Transformer
- Graph ViT / Hierarchical Transformer
- Conclusion

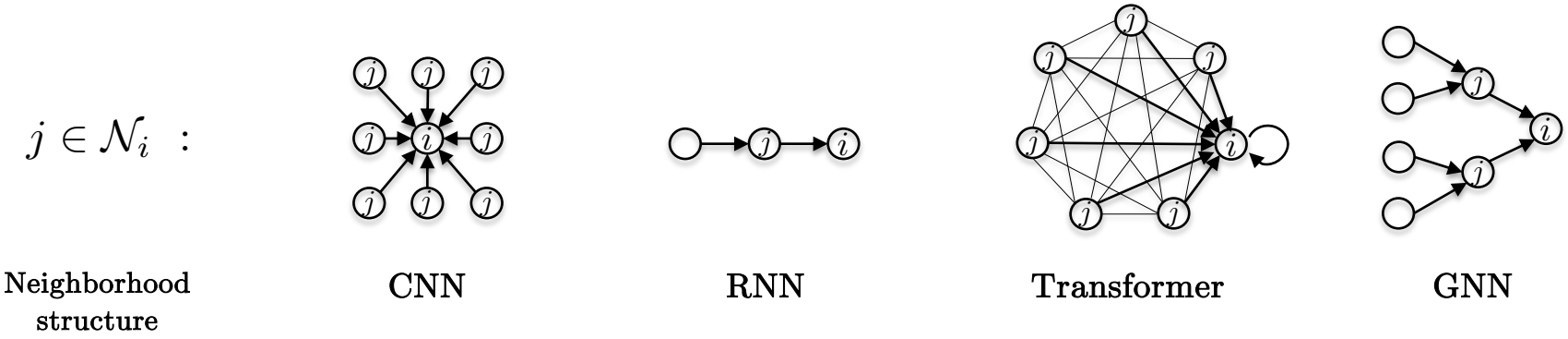
A unified message-passing view

- Most neural networks s.a. CNNs, RNNs, Transformers and GNNs can all be viewed as message-passing architectures over different domain structures.
- New feature representation is obtained by aggregating messages $j \in \mathcal{N}_i$ from a domain-dependent neighborhood / local receptive field that depends specifically on the model architecture.
- General update equation is defined as :

Next layer — Current layer

$$h_i^{\ell+1} = f_{\mathrm{NN}}\left(h_i^{\ell}, \{h_j^{\ell}\}_{j \in \mathcal{N}_i \text{ or } j \to i}\right) \in \mathbb{R}^d$$

Learnable equivariant function — Neighborhood structure / local receptive field



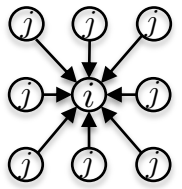
CNN for grids

- CNN such as ResNet[1] is designed for regular grids.
- Contributions
  - Exploit local spatial structure with local receptive fields.
  - Share convolution kernels across spatial locations, providing translation equivariance.
  - Build hierarchical representations by using strided convolutions for spatial down-sampling.
  - Highly GPU parallelizable and computationally efficient on regular grids.

CNN update equation :

$$h_i^{\ell+1} = \eta\Big(\sum_{j \in \mathcal{N}_i \cup i} W_j^{\ell}\, h_{i+j}^{\ell}\Big) \in \mathbb{R}^d,\ \ell = 0, ..., L-1$$

where W^{ℓ} is the convolutional kernel, and
h are the convolutional features.



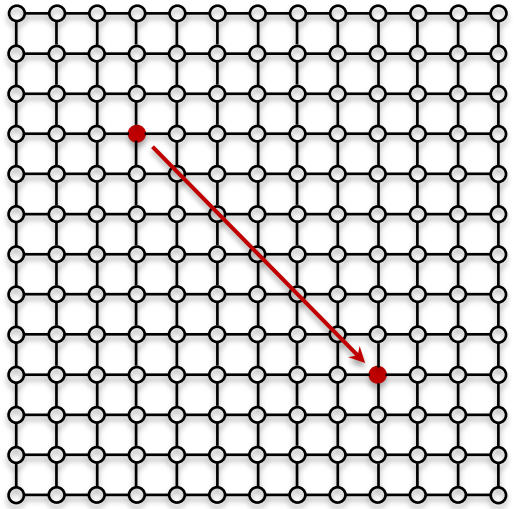
CNN neighbors $\mathcal{N}_i$

[1] He-etal, Deep Residual Learning for Image Recognition, 2016

CNN limitations

- Limitations
  - Assume a regular grid with a fixed notion of local neighborhood.
  - Translation equivariance does not imply equivariance for rotation, scale or deformation s.a. diffeomorphism.
  - Long-range interactions require many layers as the receptive field area grows as $O(L^2)$ for fixed-size local kernels, and L is the number of convolutional layers.
  - For an 1,000 x 1,000 image, then we would need 1,000 layers to propagate the information to the whole image support.

$$h^L_{i_x,i_y} = f_{\text{CNN}}(\{x_{p,q} : |p - i_x|,\ |q - i_y| \leq L\})$$
with receptive field size $(1 + 2rL)^2 = O(L^2)$,
for $k \times k$ kernel using $k = 2r + 1$.



Six 3 × 3 layers (r=1) are required for one red pixel to interact with the other red one.

RNN for sequences

- RNN s.a. LSTM[1] is designed for ordered sequences.
- Contributions
  - This model learns vectorial representation of sequences using a recurrent function with parameters shared across time.
  - It exploits the ordered, one-dimensional structure of sequences by design.
  - Major advantage is to represent sequences of any arbitrary length, as the recurrent formula is local.

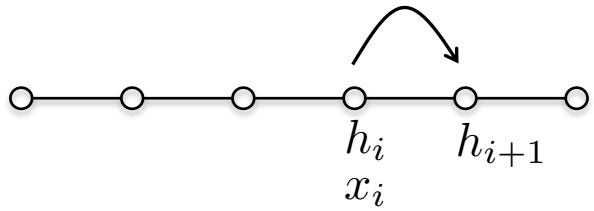
RNN update equation :

$$h_{i+1} = f_\theta(h_i, x_i) \in \mathbb{R}^d, \ i = 0, ..., L-1$$

where x_i is the current input word, and h_i is the summary of the sequence up to the i-th word of the sequence.



RNN neighbors $\mathcal{N}_i$



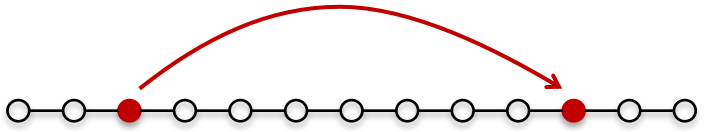
[1] Hochreiter, Schmidhuber, Long short-term memory, 1997

RNN limitations

- Limitations
  - Hard to train because they are non-linear dynamical systems, i.e. any small perturbation can amplify or vanish, a.k.a. vanishing or exploding gradients over long sequences.
  - LSTM gating mitigates, but does not eliminate, long-range optimization issues.
  - Sequential recurrence breaks parallelism, making training slower than CNNs / Transformers. This is a major bottleneck for long sequences and large-scale datasets.
  - Long-range interactions require stacking many layers and the receptive field area grows as O(L), as L being the number of recurrent steps.
  - RNN cannot learn long-range dependencies (no more than 50 steps for LTSM) between words because of over-squashing (explained soon).

$$h_L = f_{\text{RNN}}(x_{L-1}, \ldots, x_0) \in \mathbb{R}^d$$

with the receptive field size is L.



Eight layers are required for the red word to interact with the other red word.

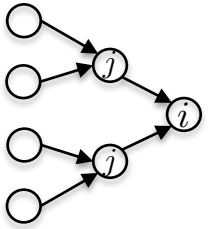
GNN for graphs

- GNN[1] is designed for graph-structured data.
- Contributions
  - Exploit graph connectivity with local message passing over neighboring nodes.
  - Share learnable functions across nodes / edges, independently of graph size.
  - Node / edge representations are permutation-equivariant, graph readouts are permutation-invariant.
  - Sparse local message passing can scale as $O(n+n_E) = O(n)$ (sparse real graphs) per layer.

GNN update equation :

$$h_i^{\ell+1} = f_{\theta_n^\ell}\left(h_i^\ell, \{h_j^\ell, e_{ij}^\ell : j \in \mathcal{N}_i\}\right) \in \mathbb{R}^d$$

$$e_{ij}^{\ell+1} = f_{\theta_e^\ell}\left(e_{ij}^\ell, h_i^\ell, h_j^\ell\right) \in \mathbb{R}^d$$

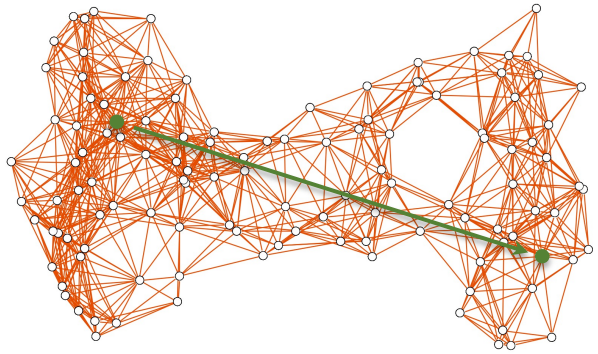


GNN neighbors $\mathcal{N}_i$

[1] Scarselli-etal, The Graph Neural Network Model, 2009

GNN limitations

- Limitations
  - Standard MP-GNNs have bounded expressivity, typically at most that of 1-WL.
  - Graph positional encoding are necessary, but they are not universal.
  - Deep message passing can lead to over-smoothing of node representations with simple convolution operations.
  - Sparse linear matrix computations are much less hardware-efficient than dense ones.
  - Long-range communication requires many layers for two far away nodes to “talk”.



Ten layers are required for one green node to interact with the other green one.

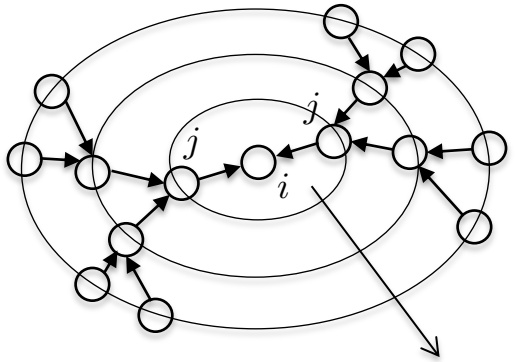
GNN limitations

- Continued
  - For a binary tree, the L-hop neighborhood grows as $O(2^L)$, i.e. exponentially with the number of layers.
  - As long-range information requires many layers, then GNN suffers from aggregation overload and over-squashing (explained soon).

$$h_i^{\ell+1} = f_{\text{GNN}}(\{h_j : j \in \mathcal{N}_\ell(i) \cup i\}) \in \mathbb{R}^d, \ \ell = 0, ..., L-1$$

with the receptive field size is $|\mathcal{N}_\ell(i)| = O(2^\ell)$

for a binary tree graph,



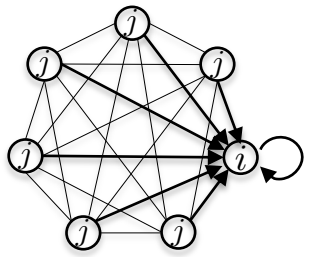
Transformer for sets

- Transformer[1] is designed for sets.
- Contributions
  - Self-attention directly connects every pair of tokens, enabling global / long-range interactions between word tokens.
  - This network is permutation-equivariant, i.e. treat words independently of their order.
  - Absolute positional encodings was first used to inform the model about sequence order, but it breaks model shift equivariance.
  - Relative positional encodings s.a. RotPE[2] makes the model shift equivariant.
  - Transformer layers are highly parallelizable on GPUs, unlike sequential RNN, with dense matrix multiplication and causal mask to hide future words.

Vanilla Transformer update equation :

$$h^{\ell+1} = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V \in \mathbb{R}^{n\times d}, \quad \text{and } h^{\ell=0} = x + p_{\text{AbsPE}} \in \mathbb{R}^{n\times d}$$

$$\text{with } Q = h^\ell W_Q^\ell,\ K = h^\ell W_K^\ell,\ V = h^\ell W_V^\ell \in \mathbb{R}^{n\times d}$$



Transformer neighbors $\mathcal{N}_i$

[1] Vaswani-etal, Attention Is All You Need, 2017
[2] Su-etal, RoFormer: Enhanced Transformer with Rotary Position Embedding, 2021

Transformer limitation : quadratic attention

- Limitations
  - Full self-attention layer is dense as is considers all n^2 token pairs.
  - It requires $O(n^2 d)$ for attention computation, and $O(n^2)$ memory for the attention matrix (conventional implementation).
  - Quadratic attention becomes increasingly expensive for long sequences, i.e. $n > 1M$ tokens.
  - On academic GPUs, the quadratic cost limits Transformers to relatively short sequences, i.e. $n \leq 100$ to 1,000 for $d = 512$.
  - Positional structure is not built into self-attention and must be introduced explicitly (not a practical issue, only theoretical one).

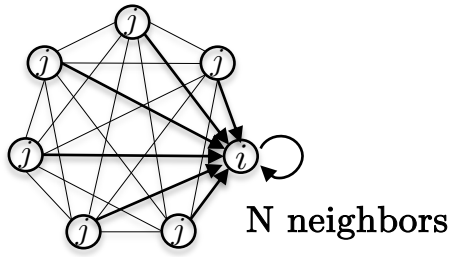
Global receptive field and depth

- Continued
  - A full self-attention layer has a global receptive field. Every token can directly receive information from all n tokens, unlike RNN O(1) and CNN O(k²).

$$h_i^L = f_{\text{Transf}}(h_i^{L-1}, \{h_j^{L-1}\}_{j=1}^N)$$

with the receptive field size is $O(N)$ where

N is the sequence length and L the number of layers.



  - Hence, one attention layer provides a one-hop communication path between any pair of tokens, close and far away tokens in the sequence. A single layer is thus enough for long-range word communication.
  - However, direct connectivity does not imply that one layer can perform any arbitrary complex computation, such as multi-step reasoning.
  - Stacking layers actually enables repeated transformations and interactions, which has been a strong property of this model.
  - But repeated layers may also lead to global mixing, representation collapse or over-squashing issue.

Over-loading and over-squashing

- **Aggregation over-loading : When many representations (here vectors) are aggregated, the model must identify and combine the few relevant signals among many irrelevant ones.**

$$f^{\ell}_{\text{MP-NN}}(\,[\,]^{\ell},[\,]^{\ell},...,[\,]^{\ell}\,) \quad \rightarrow \quad [\,]^{\ell+1}$$

select useful vectors from a large set of vectors to update the representation.

- **Compression over-squashing : Information from a rapidly growing receptive field must be compressed into fixed-dimensional hidden representations, making relevant long-range information difficult to preserve.**

$$f_{\text{compress}}(\,[\,],[\,],...,[\,]\,) = [\,]$$

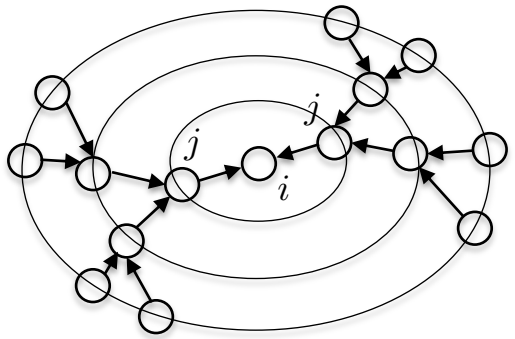
lossless compression of a large set of vectors into a fixed d-dimensional vector.

What causes over-loading and over-squashing ?

- Cause 1 : Stacking layers increases information compression issue
  - As most message-passing models, except Transformer, require multiple propagation layers to communicate over long distances, this causes the receptive field to increase leading to a large aggregation of vectors to update the representation.
  - This creates an information bottleneck where the model must decide what to compress, i.e. throw away 95% of aggregated vectors, and only keep 5%, useful for the correct update.

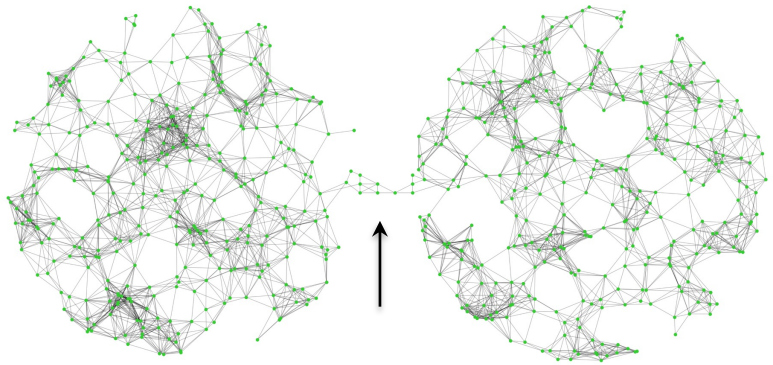
$$
\begin{aligned}
h_i^{\ell+1} &= f_{\text{NN}}(\{h_j^{\ell}\}_{j\in\mathcal{N}_i}),\ \in \mathbb{R}^d,\ \mathcal{N}_i \leftarrow \mathcal{N}_i \cup i \\
&= f_{\text{NN}}(\{f_{\text{NN}}(\{h_j^{\ell-1}\}_{j\in\mathcal{N}_i})\}_{j\in\mathcal{N}_i}) \\
&= \dots \\
&= f_{\text{NN}}(\{f_{\text{NN}}(\{f_{\text{NN}}(\dots f_{\text{NN}}(\{h_j^{0}\}_{j\in\mathcal{N}_i})\}_{j\in\mathcal{N}_i})\}_{j\in\mathcal{N}_i})\}_{j\in\mathcal{N}_i})
\end{aligned}
$$

Growing receptive field



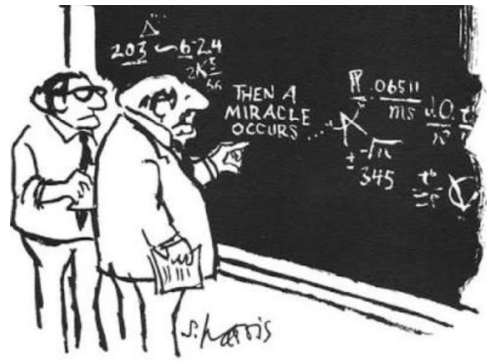
Global attention and graph bottlenecks

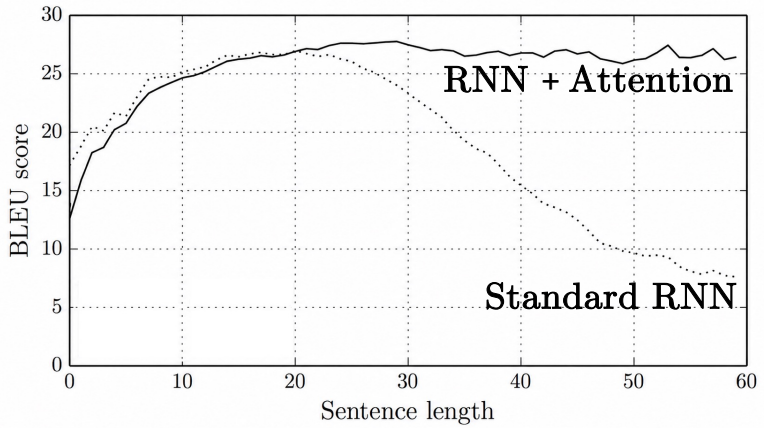
- Cause 2 : Multi-step reasoning also creates information-selection issue
  - Global attention solves long-range node interaction with a direct communication path, hence mitigating over-squashing when the solution only requires one-hop interaction.
  - However, one layer is generally insufficient for complex reasoning tasks.
  - Models still need stacking layers to compose interactions, and enable multi-step computation.
  - Each token must repeatedly select and compress information from many globally accessible representations, which cause aggregation over-loading and over-squashing.
- Cause 3 : Graph bottlenecks produce over-squashing.
  - In GNNs, a small cut or narrow bridge between large graph regions forces many long-range signals through only a few nodes / edges and fixed-dimensional hidden states.



Attention mitigates over-squashing

- Surprisingly, Transformer[1] effectively mitigates over-squashing by accessing direct long-range interactions through attention, even in deep architectures e.g. GPT-3 175B uses 96 Transformer layers[2].
- The first success of attention was introduced in a hybrid architecture combining RNN encoder-decoder with an attention mechanism[3], allowing the decoder to directly select relevant encoder states.
- Transformers remove recurrence entirely, and deep Transformers now dominate deep learning architectures.
- We next extend this idea to graphs, using attention to create adaptive communication between graph nodes.





Figure[1]. Accuracy of translated sequences (higher is better).

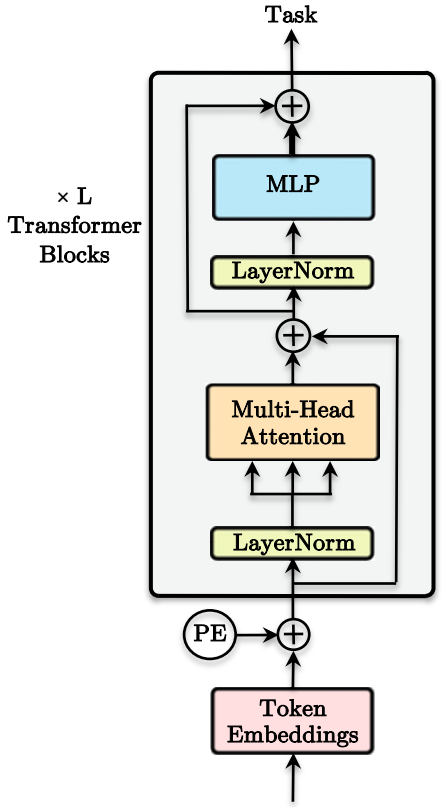
[1] Vaswani-etal, Attention Is All You Need, 2017
[2] Brown-etal, Language Models are Few-Shot Learners, 2020
[3] Bahdanau-Cho-Bengio, Neural machine translation by jointly learning to align and translate, 2014

Outline

- Review of graph network architectures
- Graph positional encoding
- Message-passing and over-squashing
- **Graph Transformer**
- Graph ViT / Hierarchical Transformer
- Conclusion

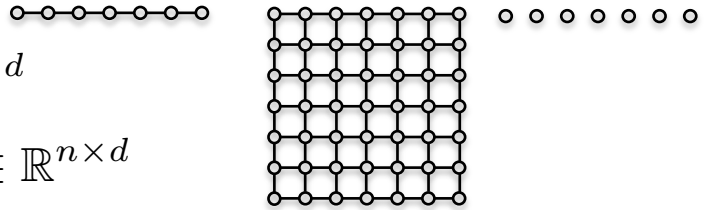
Transformer block

- Key ingredients of a Transformer block[1] :
  - Multi-head attention (MHA) : Applies self-attention across the token dimension.
  - Multi-layer perceptron (MLP) : Applies a feed-forward transformation along the feature dimension, independently to each token.
  - Residual connections (RC) and LayerNorm (LN) : Stabilize optimization, and enable effective training of deep networks.
  - Positional encodings (PE) : Inject information about token order before or within the Transformer blocks.
- Transformer blocks have been applied to sequences[1], images[2], sets[3].



$$h^{\ell+\frac{1}{2}} \leftarrow h^{\ell} + \mathrm{MHA}(\mathrm{LN}(h^{\ell})) \in \mathbb{R}^{n \times d}$$

$$h^{\ell+1} \leftarrow h^{\ell+\frac{1}{2}} + \mathrm{MLP}(\mathrm{LN}(h^{\ell+\frac{1}{2}})) \in \mathbb{R}^{n \times d}$$



[1] Vaswani-etal, Attention is all you need, 2017
[2] Dosovitskiy-etal, An image is worth 16x16 words: Transformers for image recognition at scale, 2020
[3] Lee-etal, Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks, 2019

Self-attention as learnable message-passing

- **Self-attention lets each token receive information from every other token, including itself.**
- **The new token representation aggregates value vectors using softmax attention weights A_{ij}.**
- **Attention matrix A_{ij} can be interpreted as a directed edge weight j → i in a fully-connected token graph.**

Vector representation :

$$\text{Head}(h)_i = \sum_{j=1}^{n} A_{ij} v_j \in \mathbb{R}^{d_h} \quad \text{single attention head}$$

$$\text{with } A_{ij} = \frac{\exp\left(q_i^\top k_j / \sqrt{d_h}\right)}{\sum_{r=1}^{n} \exp\left(q_i^\top k_r / \sqrt{d_h}\right)} \geq 0, \ \sum_{j=1}^{n} A_{ij} = 1 \quad \text{attention score}$$

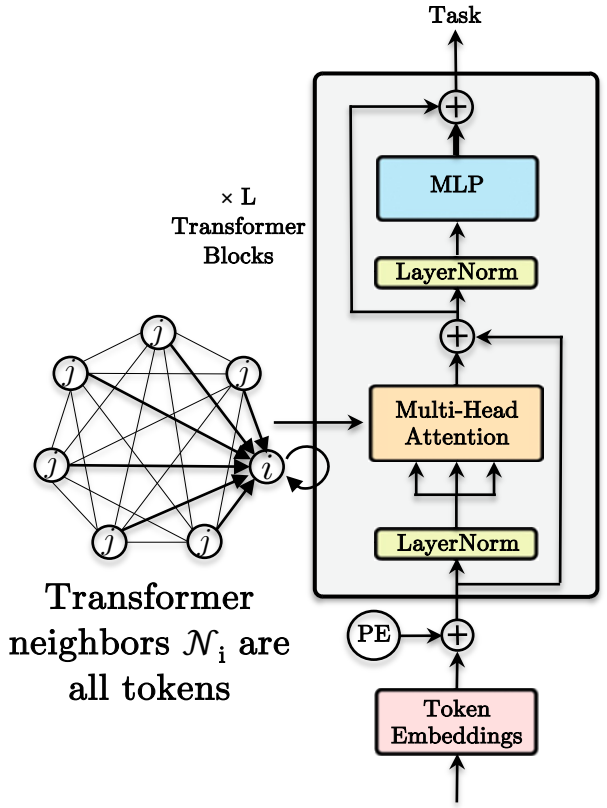
$$\text{and } q_i = h_i W_Q, k_i = h_i W_K, v_i = h_i W_V \in \mathbb{R}^{d_h} \quad \text{learnable dictionary}$$

Matrix representation :

$$\text{Head}(h) = Av \in \mathbb{R}^{n \times d_h}$$

$$\text{with } A = \text{softmax}_{\text{row}}\left(\frac{qk^\top}{\sqrt{d_h}}\right) \in \mathbb{R}^{n \times n}$$

$$\text{and } q = hW_Q, k = hW_K, v = hW_V \in \mathbb{R}^{n \times d_h}$$



Multi-head self-attention

- **Multi-head attention (MHA) computes several self-attention heads in parallel.**
- **Each head uses its own query, key and value projections and can learn different token-to-token interactions.**
- **Combining multiple heads allows the model to represent several interaction patterns simultaneously, i.e. increasing the model's expressivity.**

For head $m = 1, \ldots, H$:

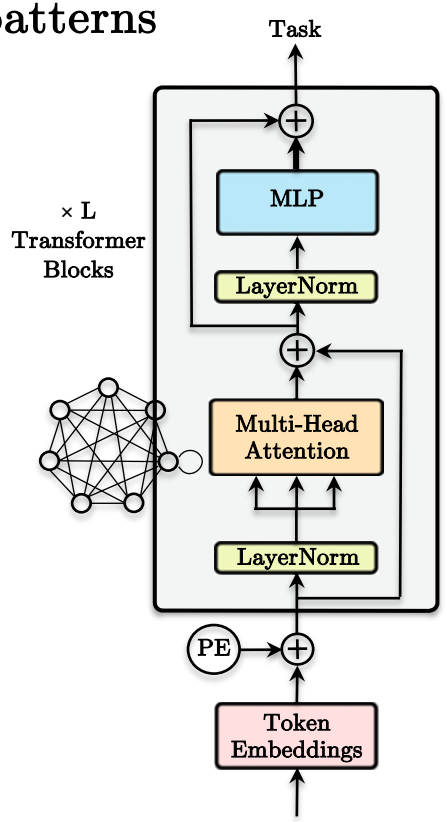
$$\text{Head}_m(h) = A_m v_m \in \mathbb{R}^{n \times d_h},$$

$$\text{with } A_m = \text{softmax}\left(\frac{q_m k_m^\top}{\sqrt{d_h}}\right) \in \mathbb{R}^{n \times n},$$

$$\text{with } q_m = hW_Q^m,\ k_m = hW_K^m,\ v_m = hW_V^m \in \mathbb{R}^{N \times d_h},\ d_h = d/H,$$

Lastly, concatenate the heads and apply a linear transformation :

$$\text{MHA}(h) = \text{Concat}\left(\text{Head}_1, \ldots, \text{Head}_H\right) W_O \in \mathbb{R}^{n \times d}.$$

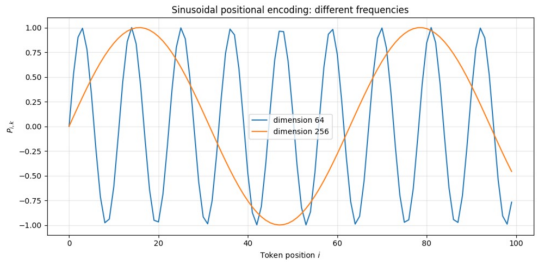


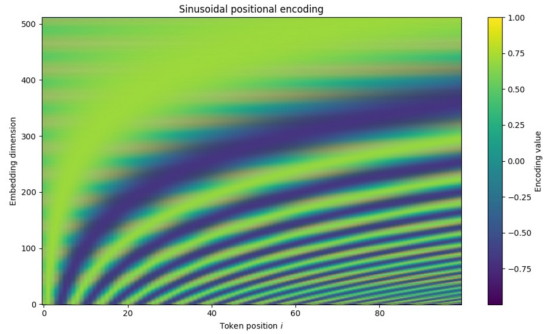
Token positional encodings

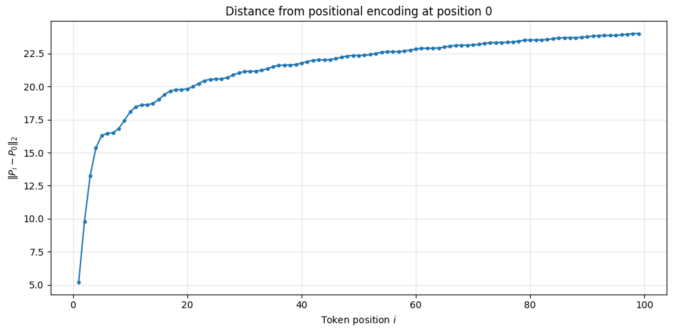
- Self-attention without positional information is permutation-equivariant.
- This implies that content-only self-attention does not distinguish different token orderings.
- Sequence position can be introduced with absolute PEs at the input, or with relative-position mechanisms in attention, such as RoPE.
- The original Transformer uses sinusoidal absolute PEs with geometrically spaced frequencies, encoding position at multiple scales.

$$\text{with } p_{ik} = \begin{cases} \sin(2\pi f_k i) \text{ if } k \text{ even} \\ \cos(2\pi f_k i) \text{ if } k \text{ odd} \end{cases}, \quad \text{frequency } f_k = \frac{10,000^{\frac{d}{\lfloor 2k \rfloor}}}{2\pi}$$

$$\text{and } i = 0, ..., n-1, \; k = 0, ..., \frac{d}{2} - 1$$







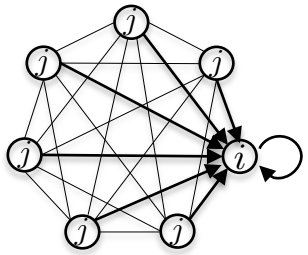
Graph Transformer

- GT[1] contributions :
  - Generalize Transformer[2] to arbitrary graphs using sparse neighborhood attention with O(N+E) complexity per layer, or dense global attention with $O(N^2)$ complexity.
  - Exploit graph connectivity as a structural inductive bias, constraining which nodes exchange information.
  - Use Laplacian eigenvectors[3] as node positional encodings, analogous to sinusoidal positional encodings for sequences.
  - Incorporate edge features directly into attention scores, e.g. through bilinear interactions, while maintaining explicit edge representations. It is critical for modeling molecular bonds or relations between entities in knowledge graphs.
  - Graph Transformer[1] and hybrid GNN+Transformer architectures[4] have become effective and widely used models in AI-for-Science, s.a. in molecular biology[5]. For example, GPS++[4] won the NeurIPS 2022 OGB Large-Scale Challenge[6] .

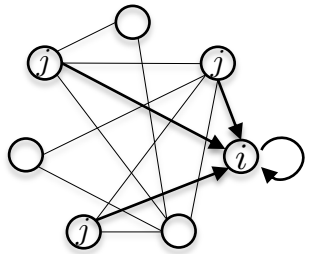
[1] Dwivedi-Bresson, A generalization of transformer networks to graphs, AAAI 2021
[2] Vaswani-etal, Attention is all you need, 2017
[3] Belkin, Niyogi, Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering, 2001
[4] Rampasek et-al, Recipe for a General, Powerful, Scalable Graph Transformer, 2022
[5] Ying-etal, Do Transformers Really Perform Bad for Graph Representation, 2021
[6] https://ogb.stanford.edu/neurips2022/workshop

Sparse graph attention

- Graph self-attention
  - Graph self-attention restricts each node i to attend only to its neighbors $j \in \mathcal{N}_i$, including itself.
  - For the rest of the lecture, we will use $\mathcal{N}_i \leftarrow \mathcal{N}_i \cup \{i\}$.
  - Attention scores are normalized locally over the neighborhood of each node.
  - Neighbor features are aggregated using learned attention weights to obtain an attention-based message-passing update.
  - Sparse graph attention requires $O(E)$ pairwise interactions per layer, compared with $O(N^2)$ for standard dense global attention.



Standard Transformer
Neighbors $\mathcal{N}_i$ are
all tokens



Graph Transformer
Neighbors $\mathcal{N}_i$ are
node neighbors

Sparse graph attention

- **Graph self-attention equations**

Vector representation :

$$\text{gHead}(h)_i = \sum_{j \in \mathcal{N}_i} A^g_{ij} v_j \in \mathbb{R}^{d_h} \quad \text{single graph attention head}$$

$$\text{with } A^g_{ij} = \frac{\exp\left(q_i^\top k_j / \sqrt{d_h}\right)}{\sum_{r \in \mathcal{N}_i} \exp\left(q_i^\top k_r / \sqrt{d_h}\right)} \geq 0, \sum_{j \in \mathcal{N}_i} A^g_{ij} = 1 \quad \text{graph attention score}$$

$$\text{and } q_i = h_i W_Q, k_i = h_i W_K, v_i = h_i W_V \in \mathbb{R}^{d_h} \quad \text{learnable dictionary}$$

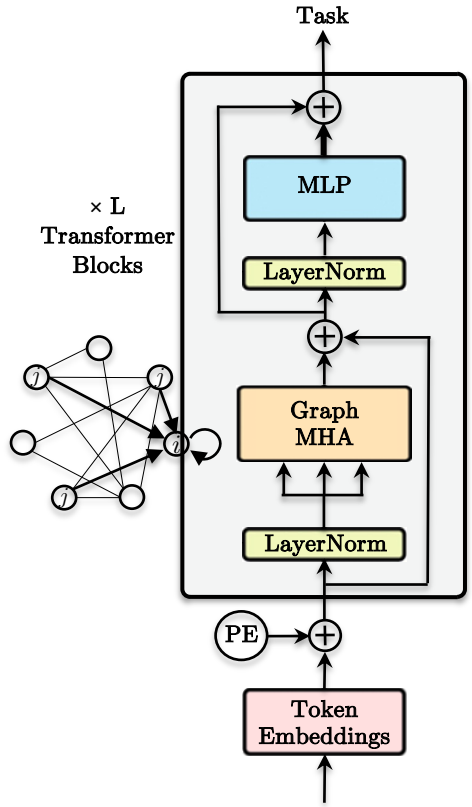
Matrix representation :

$$\text{gHead}(h) = A^g v \in \mathbb{R}^{n \times d_h}$$

$$\text{with } A^g = \text{softmax}_{\text{row}}\left(\frac{qk^\top}{\sqrt{d_h}} + M^g\right) \in \mathbb{R}^{n \times n}$$

$$(M^g)_{ij} = \begin{cases} 0 & \text{if } ij \in E \text{ (connected)} \\ -\infty & \text{if } ij \text{ not connected} \end{cases} \quad \text{graph connectivity mask}$$

$$\text{and } q = hW_Q, k = hW_K, v = hW_V \in \mathbb{R}^{n \times d_h}$$



Graph MHA

- **Graph multi-head self-attention (gMHA) computes several neighborhood-attention patterns in parallel.**
- **Each head uses independent dictionaries (a.k.a. projections) and can learn different node-to-node interactions, constrained by the graph topology.**

For head $m = 1, \ldots, H$:

$$\text{gHead}_m(h) = A^g_m v_m \in \mathbb{R}^{n \times d_h},$$

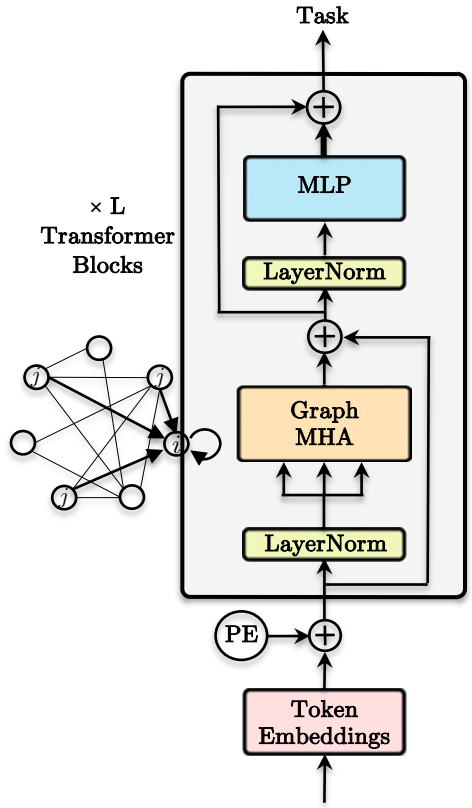
$$\text{with } A^g_m = \text{softmax}\left(\frac{q_m k_m^\top}{\sqrt{d_h}} + M^g\right) \in \mathbb{R}^{n \times n},$$

$$\text{and } (M^g)_{ij} = \begin{cases} 0 & \text{if } ij \in E \text{ (connected)} \\ -\infty & \text{if } ij \text{ not connected} \end{cases}$$

$$q_m = hW^m_Q,\ k_m = hW^m_K,\ v_m = hW^m_V \in \mathbb{R}^{n \times d_h},\ d_h = d/H,$$

Lastly, concatenate the heads and apply a linear transformation :

$$\text{gMHA}(h) = \text{Concat}\left(\text{gHead}_1, \ldots, \text{gHead}_H\right) W_O \in \mathbb{R}^{n \times d}.$$



Graph Fourier modes as positional encodings

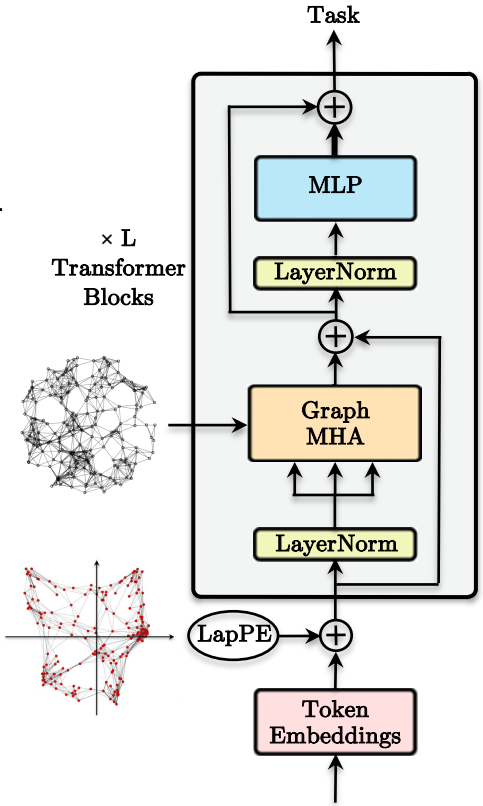
- **Graph Laplacian eigenvectors provide node positional encodings (LapPEs).**
- **They define graph Fourier modes, analogous to sine/cosine Fourier modes on regular domains.**

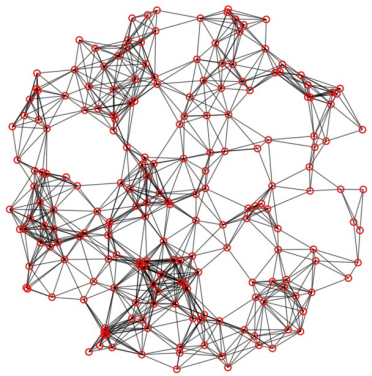
$$L = I - D^{-1/2} A D^{-1/2} = \Phi \Lambda \Phi^\top \in \mathbb{R}^{n \times n}, \; 0 = \lambda_1 \leq \cdots \leq \lambda_N$$

where $\Phi = [\phi_1, ..., \phi_n] \in \mathbb{R}^{n \times n}$ contains the Laplacian eigenvectors.

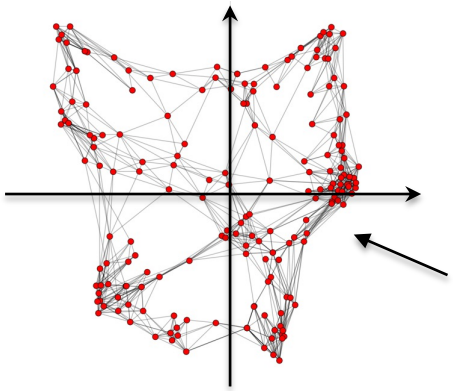
The first K non-trivial eigenvectors define the node PEs :

$p = [\phi_2, ..., \phi_{K+1}] \in \mathbb{R}^{n \times K}$ for connected graphs



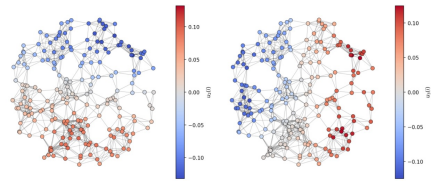


Graph



LapPEs

$\Phi = [\phi_2, \phi_3] \in \mathbb{R}^{n \times 2}$



First two non-trivial eigenvectors

Lab 1 : Vanilla Graph Transformer

- Graph Transformer without node PEs

Lecture : Graph Transformer

Lab 01 : Vanilla Graph Transformers (GT without Positional Encoding)

Xavier Bresson

Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
https://arxiv.org/pdf/2012.09699.pdf

```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/CS5284_2024_codes/codes/10_Graph_Transformers'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
    !pip install dgl==1.0.0 # Install DGL
```

```
# Libraries
import dgl
from dgl.data import MiniGCDataset
import dgl.function as fn
import matplotlib.pyplot as plt
import networkx as nx
import torch
from torch.utils.data import DataLoader
import torch.nn as nn
import time
```

Design the class of vanilla GraphTransformer networks with DGL

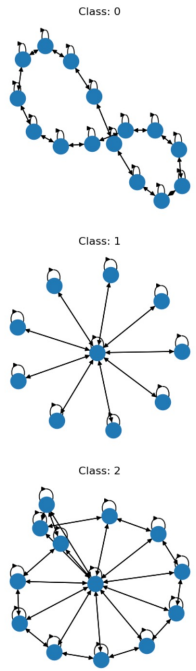
Node update equation:

$$\bar{h}^{\ell} = h^{\ell} + \text{gMHA}(\text{LN}(h^{\ell})) \in \mathbb{R}^{N\times d}$$
$$h^{\ell+1} = \bar{h}^{\ell} + \text{MLP}(\text{LN}(\bar{h}^{\ell})) \in \mathbb{R}^{N\times d}$$
$$\text{with gMHA}(h) = \text{Concat}_{k=1}^{H}(\text{gHA}(h_k))W_O \in \mathbb{R}^{N\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, W_O \in \mathbb{R}^{d\times d}$$
$$\text{gHA}(h) = \text{Softmax}\left(A_G \odot \frac{QK^T}{\sqrt{d'}}\right)V \in \mathbb{R}^{N\times d'=d/H}, A_G \in \mathbb{R}^{N\times N} \text{ (graph adjacency matrix)}$$
$$\text{gHA}(h)_i = \sum_{j\in\mathcal{N}_i} \underbrace{\frac{\exp(q_i^T k_j/\sqrt{d'})}{\sum_{j'\in\mathcal{N}_i}\exp(q_i^T k_{j'}/\sqrt{d'})}}_{\text{graph attention score}_{ij}} v_j \text{ (point-wise equation)}$$
$$Q = h_k W_Q, K = h_k W_K, V = h_k W_V \in \mathbb{R}^{N\times d'=d/H}, W_Q, W_K, W_V \in \mathbb{R}^{d'\times d'}$$
$$h^{\ell=0} = \text{LL}(h_0) \in \mathbb{R}^{N\times d} \text{ (input node feature)}$$

```
# class MLP layer for classification
class MLP_layer(nn.Module):

    def __init__(self, input_dim, output_dim, L=2): # L = nb of hidden layers
        super(MLP_layer, self).__init__()
        list_FC_layers = [ nn.Linear( input_dim, input_dim, bias=True ) for l in range(L) ]
        list_FC_layers.append(nn.Linear( input_dim, output_dim , bias=True ))
        self.FC_layers = nn.ModuleList(list_FC_layers)
        self.L = L

    def forward(self, x):
        y = x
        for l in range(self.L):
            y = self.FC_layers[l](y)
            y = torch.relu(y)
        y = self.FC_layers[self.L](y)
        return y
```



Lab 2 : Graph Transformer

- Graph Transformer with graph Laplacian PEs

Lecture : Graph Transformer

Lab 02 : Graph Transformers (GT with Positional Encoding)

Xavier Bresson

Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
https://arxiv.org/pdf/2012.09699.pdf

```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/CS5284_2024_codes/codes/10_Graph_Transformers'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
    !pip install dgl==1.0.0 # Install DGL
```

```
# Libraries
import dgl
from dgl.data import MiniGCDataset
import dgl.function as fn
import matplotlib.pyplot as plt
import networkx as nx
import torch
from torch.utils.data import DataLoader
import torch.nn as nn
import time
import sys; sys.path.insert(0, 'lib/')
from lib.utils import compute_ncut
```

Design the class of GraphTransformer networks (with positional encoding)

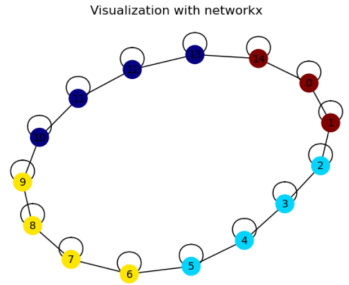
Node update equation:

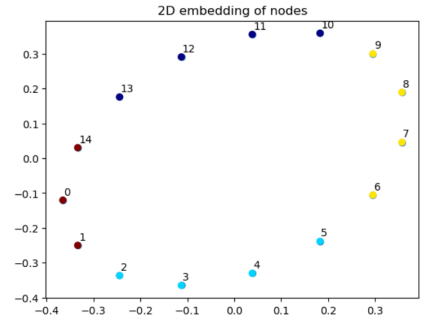
$$\bar{h}^{\ell} = h^{\ell} + \text{gMHA}(\text{LN}(h^{\ell})) \in \mathbb{R}^{N\times d}$$
$$h^{\ell+1} = \bar{h}^{\ell} + \text{MLP}(\text{LN}(\bar{h}^{\ell})) \in \mathbb{R}^{N\times d}$$
$$\text{with gMHA}(h) = \text{Concat}_{k=1}^{H}(\text{gHA}(h_k))W_O \in \mathbb{R}^{N\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, W_O \in \mathbb{R}^{d\times d}$$
$$\text{gHA}(h) = \text{Softmax}\left(A_G \odot \frac{QK^T}{\sqrt{d'}}\right)V \in \mathbb{R}^{N\times d'=d/H}, A_G \in \mathbb{R}^{N\times N} \text{ (graph adjacency matrix)}$$
$$\text{gHA}(h)_i = \sum_{j\in\mathcal{N}_i} \underbrace{\frac{\exp(q_i^T k_j/\sqrt{d'})}{\sum_{j'\in\mathcal{N}_i}\exp(q_i^T k_{j'}/\sqrt{d'})}}_{\text{graph attention score}_{ij}} v_j \text{ (point-wise equation)}$$
$$Q = h_k W_Q, K = h_k W_K, V = h_k W_V \in \mathbb{R}^{N\times d'=d/H}, W_Q, W_K, W_V \in \mathbb{R}^{d'\times d'}$$
$$h^{\ell=0} = \text{LL}_1(h_0) + \text{LL}_2(p_0) \in \mathbb{R}^{N\times d} \text{ (input node feature and positional encoding)}$$
$$\text{with } p_0 = \Phi_{\{2,..,K+1\}} \in \mathbb{R}^{N\times K},\ \Delta = \Phi\Lambda\Phi^T \in \mathbb{R}^{N\times N}$$

```
# class MLP layer for classification
class MLP_layer(nn.Module):

    def __init__(self, input_dim, output_dim, L=2): # L = nb of hidden layers
        super(MLP_layer, self).__init__()
        list_FC_layers = [ nn.Linear( input_dim, input_dim, bias=True ) for l in range(L) ]
        list_FC_layers.append(nn.Linear( input_dim, output_dim , bias=True ))
        self.FC_layers = nn.ModuleList(list_FC_layers)
        self.L = L

    def forward(self, x):
        y = x
        for l in range(self.L):
            y = self.FC_layers[l](y)
            y = torch.relu(y)
        y = self.FC_layers[self.L](y)
        return y
```





Graph Transformers with edge features

- **Graph Transformer with edge-aware self-attention mechanism.**
- **Learn edge-conditioned attention scores to modulate interactions between neighboring nodes[1] :**

Vector representation :

$$\text{gHead}(h)_i = \sum_{j \in \mathcal{N}_i} A^e_{ij} v_j \in \mathbb{R}^{d_h} \quad \text{single graph attention head}$$

$$\text{with } A^e_{ij} = \frac{\exp\left(q_i^\top E_{ij} k_j / \sqrt{d_h}\right)}{\sum_{r \in \mathcal{N}_i} \exp\left(q_i^\top E_{ir} k_r / \sqrt{d_h}\right)} \geq 0, \sum_{j \in \mathcal{N}_i} A^e_{ij} = 1 \quad \text{edge-aware attention score}$$

$$q_i = h_i W_Q, k_i = h_i W_K, v_i = h_i W_V \in \mathbb{R}^{d_h} \quad \text{learnable dictionary}$$

$$E_{ij} = \text{diag}(w_{ij}) \in \mathbb{R}^{d \times d}, \; w_{ij} = e_{ij} W_E \in \mathbb{R}^{d_h} \quad \text{edge features}$$

The edge-based attention score has a tri-linear form :

$$s_{ij} = q_i^\top E_{ij} k_j = \sum_{r=1}^{d_h} q_{ir} w_{ij,r} k_{jr} \in \mathbb{R},$$

which can be implemented with DGL/PyG message-passing operations :

$s_{ij} = \text{sum}_{\text{dim}=0}(q_i \odot w_{ij} \odot k_j)$ where $\odot$ is the element-wise product.

Dense edge-based graph attention

- **Graph attention can be implemented either with sparse edge-wise operations (DGL/PyG) or with dense tensors (PyTorch).**
- **The dense formulation maps efficiently to standard GPU tensor operations and does not require specialized sparse CUDA kernels.**
- **However, densification increases computation from O(E) to O(n^2) for speed and memory.**

Matrix / tensor representation :

$$\text{gHead}(h) = A^e v \in \mathbb{R}^{n \times d_h}$$

with $A^e = \text{softmax}_{\text{row}}(S) \in \mathbb{R}^{n \times n}$

$$S = \text{sum}_{\text{dim}=2}(\bar{q} \odot \bar{e} \odot \bar{k}) + M^g \in \mathbb{R}^{n \times n}$$

$$\bar{q} = \text{unsqueeze}_{\text{dim}=1}(q) = q[:, \text{None}, :] \in \mathbb{R}^{n \times 1 \times d_h}, \; q = hW_Q \in \mathbb{R}^{n \times d_h}$$

$$\bar{k} = \text{unsqueeze}_{\text{dim}=0}(k) = k[\text{None}, :, :] \in \mathbb{R}^{1 \times n \times d_h}, \; k = hW_K \in \mathbb{R}^{n \times d_h}$$

$$\bar{e} = \text{densify}(w) \in \mathbb{R}^{n \times n \times d_h}, \; w = eW_K \in \mathbb{R}^{n_E \times d_h}$$

$$\bar{q} \odot \bar{e} \odot \bar{k} \in \mathbb{R}^{n \times n \times d_h} \quad \text{broadcasting and multiplication}$$

Edge-based graph multi-head attention

- **Edge-based graph MHA (gMHA^e) computes several neighborhood-attention patterns in parallel.**
- **Different heads can learn different edge-conditioned node-to-node interactions, constrained by the graph topology.**

Sparse version :

For head $m = 1, \dots, H$:

$$\text{gHead}_m^e(h) = A_m^e v_m \in \mathbb{R}^{n \times d_h}, \; A_m^e \in \mathbb{R}^{n \times n}$$

$$(A_m^e)_{ij} = \frac{\exp\left(q_i^{m\top} E_{ij}^m k_j^m / \sqrt{d_h}\right)}{\sum_{r \in \mathcal{N}_i} \exp\left(q_i^{m\top} E_{ir}^m k_r^m / \sqrt{d_h}\right)} \geq 0, \; \sum_{j \in \mathcal{N}_i} (A_m^e)_{ij} = 1$$

$$q_i^m = h_i W_Q^m, k_i^m = h_i W_K^m, v_i^m = h_i W_V^m \in \mathbb{R}^{d_h}$$

$$E_{ij}^m = \text{diag}(w_{ij}^m) \in \mathbb{R}^{d \times d}, \; w_{ij}^m = e_{ij} W_E^m \in \mathbb{R}^{d_h}$$

Lastly, concatenate the heads and apply a linear transformation :

$$\text{gMHA}^e(h) = \text{Concat}\left(\text{gHead}_1^e, \dots, \text{gHead}_H^e\right) W_O \in \mathbb{R}^{n \times d}.$$

Dense edge-based graph MHA

- **Dense version :**

For head $m = 1, \ldots, H$:

$$\text{gHead}_m(h) = A^e_m v_m \in \mathbb{R}^{n \times d_h},$$

$$\text{with } A^e_m = \text{softmax}\left(S_m\right) \in \mathbb{R}^{n \times n},$$

$$S_m = \text{sum}_{\text{dim}=2}(\bar{q}_m \odot \bar{e}_m \odot \bar{k}_m) + M^g \in \mathbb{R}^{n \times n},\ d_h = d/H$$

$$\bar{q}_m = \text{unsqueeze}_{\text{dim}=1}(q_m) = q_m[:, \text{None}, :] \in \mathbb{R}^{n \times 1 \times d_h},\ q_m = hW^m_Q \in \mathbb{R}^{n \times d_h}$$

$$\bar{k}_m = \text{unsqueeze}_{\text{dim}=0}(k_m) = k_m[\text{None}, :, :] \in \mathbb{R}^{1 \times n \times d_h},\ k_m = hW^m_K \in \mathbb{R}^{n \times d_h}$$

$$\bar{e}_m = \text{densify}(w_m) \in \mathbb{R}^{n \times n \times d_h},\ w_m = eW^m_K \in \mathbb{R}^{n_E \times d_h}$$

Lastly, concatenate the heads and apply a linear transformation :

$$\text{gMHA}^e(h) = \text{Concat}\left(\text{gHead}^e_1, \ldots, \text{gHead}^e_H\right) W_O \in \mathbb{R}^{n \times d}.$$

Coupled node and edge updates

- **Graph Transformer with jointly updated node and edge representations[1] :**

For layer $\ell = 0, \dots, L-1$:

$$h^{\ell+\frac{1}{2}} \leftarrow h^{\ell} + \text{gMHA}^{n}(\text{LN}(h^{\ell}), \text{LN}(e^{\ell})) \in \mathbb{R}^{n \times d}$$

$$h^{\ell+1} \leftarrow h^{\ell+\frac{1}{2}} + \text{MLP}(\text{LN}(h^{\ell+\frac{1}{2}})) \in \mathbb{R}^{n \times d}$$

$$e^{\ell+\frac{1}{2}} \leftarrow e^{\ell} + \text{MHA}^{e}(\text{LN}(h^{\ell}), \text{LN}(e^{\ell})) \in \mathbb{R}^{n_e \times d}$$

$$e^{\ell+1} \leftarrow e^{\ell+\frac{1}{2}} + \text{MLP}(\text{LN}(e^{\ell+\frac{1}{2}})) \in \mathbb{R}^{n_e \times d}$$

with

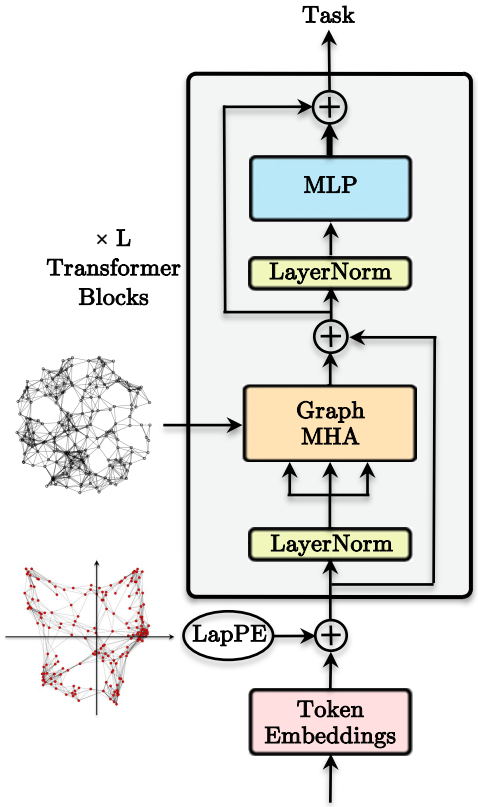
$$\text{gMHA}^{e}(h, e) = \text{Concat}\left(\text{gHead}^{e}_{1}, \dots, \text{gHead}^{e}_{H}\right) W_O \in \mathbb{R}^{n \times d}$$

$$\text{and } \text{gHead}^{e}_{m}(h, e)_{ij} = \frac{q^{m}_{i} \odot w^{m}_{ij} \odot k^{m}_{j}}{\sqrt{d_h}} \in \mathbb{R}^{d_h}, \; d_h = d/H$$

$$q^{m}_{i} = h_i W^{m}_{Q}, \; k^{m}_{i} = h_i W^{m}_{K}, \; v^{m}_{i} = h_i W^{m}_{V}, \; w^{m}_{ij} = e_{ij} W^{m}_{E} \in \mathbb{R}^{d_h}$$

Initialize with : $h^{\ell=0} = \text{Linear}_{x}(x) + \text{Linear}_{p}(p) \in \mathbb{R}^{n \times d}$

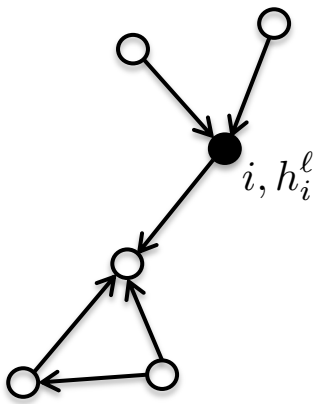
$$e^{\ell=0} = \text{Linear}_{e}(e) \in \mathbb{R}^{n_e \times d}$$



[1] Dwivedi-Bresson, A generalization of transformer networks to graphs, 2020

Standard Transformer as a special case of GT

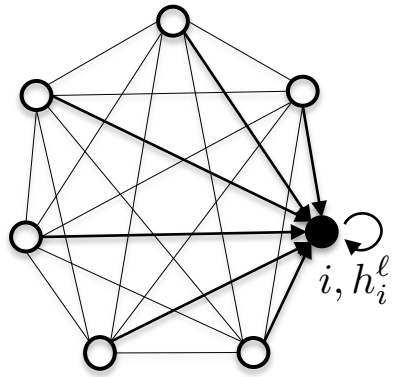
- Graph Transformer[1] reduces to standard Transformer[2] when
  - The graph is fully connected, i.e. $\mathcal{N}_i = V$, every node attends to every node.
  - Edge vectors become identity, i.e. $E_{ij} = \text{diag}(w_{ij}) = I_d$ (d-dim identity matrix) or $w_{ij} = 1_d$ (d-dim vector of ones).
  - Node LapPEs reduce to sinusoidal functions for a sequence of tokens.



Sparsely connected graph $\mathcal{N}_i$

$$\frac{q_i^\top E_{ij} k_j}{\sqrt{d_h}} \xrightarrow[E_{ij}=I]{\mathcal{N}_i=V} \frac{q_i^\top k_j}{\sqrt{d_h}}$$

$$\Rightarrow$$

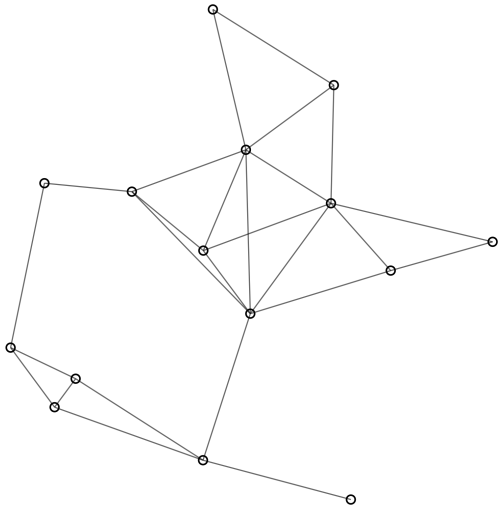


Fully connected graph $\mathcal{N}_i = V$

[1] Dwivedi-Bresson, A generalization of transformer networks to graphs, 2020
[2] Vaswani-etal, Attention is all you need, 2017

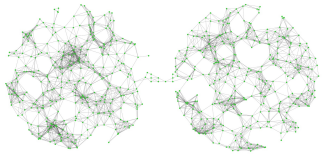
Sparse GT vs. dense GT

- Graph Transformer can use either local sparse or global dense attention.
- Sparse Graph Transformer
  - Local sparse attention : Attend only along graph edges.
  - Advantages :
    - $O(n+n_E)$ attention pairs, scales to large sparse graphs, e.g. n = 10^6 to 10^{10} (no OOM).
    - Efficiently implemented with PyG / DGL.
  - Limitations :
    - Irregular sparse operations are less GPU-efficient than dense ones.
    - We need to stack many layers for long-rage interaction.
    - However, multi-hop communication suffer from over-squashing.

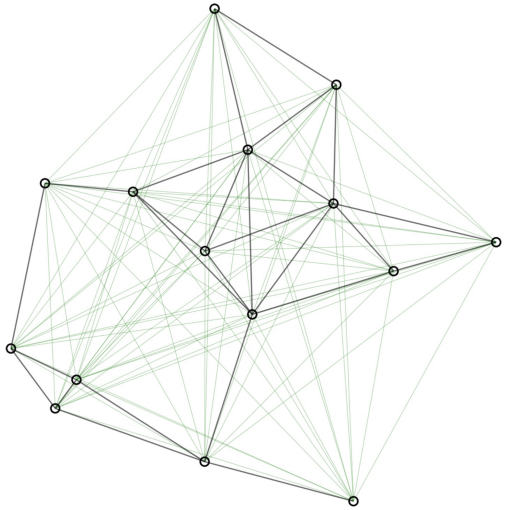


Sparse GT vs. dense GT

- Dense Graph Transformer
  - Global dense attention : Attend over all node pairs.
  - The missing edges are explicitly represented in a completed graph with a special type, e.g. 'None' or 'No-edge' type.
  - Advantages :
    - Direct long-range interactions.
    - Highly optimized dense GPU kernels with PyTorch.
    - Mitigates graph-path bottlenecks.



  - Limitations:
    - $O(n^2)$ attention pairs and memory pressure.
    - Practical only when n is sufficiently small, e.g. $n \leq 100$.



Lab 3 : Sparse GT with edge features

- Sparse Graph Transformer with edge features and DGL

Lecture : Graph Transformer

Lab 03 : Graph Transformers with edge features and DGL (sparse linear algebra)

Xavier Bresson

Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
https://arxiv.org/pdf/2012.09699.pdf

```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/CS5284_2024_codes/codes/10_Graph_Transformers'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
    !pip install dgl # Install DGL
```

```
# Libraries
import pickle
import dgl
import dgl.function as fn
import matplotlib.pyplot as plt
import torch
from torch.utils.data import DataLoader
import torch.nn as nn
import time
import networkx as nx
import sys; sys.path.insert(0, 'lib/')
from lib.utils import compute_ncut
from lib.molecules import Dictionary, MoleculeDataset, MoleculeDGL, Molecule
import os, datetime
```

Design the class of GraphTransformer networks with edge features

Node update equation:

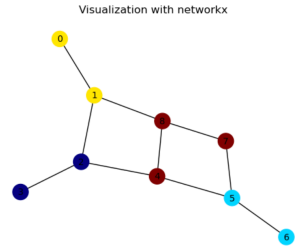
$$\bar{h}^{\ell} = h^{\ell} + \text{gMHA}(\text{LN}(h^{\ell}), \text{LN}(e^{\ell})) \in \mathbb{R}^{N\times d}$$
$$h^{\ell+1} = \bar{h}^{\ell} + \text{MLP}(\text{LN}(\bar{h}^{\ell})) \in \mathbb{R}^{N\times d}$$
$$\text{with gMHA}(h,e) = \text{Concat}_{k=1}^{H}(\text{gHA}(h_k,e_k))W_O \in \mathbb{R}^{N\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, e_k \in \mathbb{R}^{E\times d'}, W_O \in \mathbb{R}^{d\times d}$$
$$\text{gHA}(h,e)_i = \sum_{j\in N_i} \underbrace{\frac{\exp(q_i^T \text{diag}(e_{ij})k_j/\sqrt{d'})}{\sum_{j'\in N_i}\exp(q_i^T \text{diag}(e_{ij'})k_{j'}/\sqrt{d'})}}_{\text{graph attention score w/ edge feature}_{ij}} v_j \ \text{(point-wise equation)}$$
$$Q = h_k W_Q, K = h_k W_K, V = h_k W_V \in \mathbb{R}^{N\times d'=d/H}, E = e_k W_E \in \mathbb{R}^{E\times d'=d/H}, W_Q, W_K, W_V, W_E \in \mathbb{R}^{d'\times d'}$$
$$h^{\ell=0} = \text{LL}_1(h_0) + \text{LL}_2(p_0) \in \mathbb{R}^{N\times d} \text{ (input node feature and positional encoding)}$$
$$\text{with } p_0 = \Phi_{\{2,..,K+1\}} \in \mathbb{R}^{N\times K},\ \Delta = \Phi\Lambda\Phi^T \in \mathbb{R}^{N\times N}$$

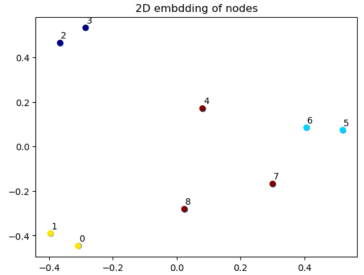
Edge update equation:

$$\bar{e}^{\ell} = e^{\ell} + \text{gMHE}(\text{LN}(e^{\ell}), \text{LN}(h^{\ell})) \in \mathbb{R}^{E\times d}$$
$$e^{\ell+1} = \bar{e}^{\ell} + \text{MLP}(\text{LN}(\bar{e}^{\ell})) \in \mathbb{R}^{E\times d}$$
$$\text{with gMHE}(e,h) = \text{Concat}_{k=1}^{H}(\text{gHE}(e_k,h_k))W_O^e \in \mathbb{R}^{E\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, e_k \in \mathbb{R}^{E\times d'}, W_O^e \in \mathbb{R}^{d\times d}$$
$$\text{with gHE}(e,h)_{ij} = q_i \odot e_{ij} \odot k_j/\sqrt{d'} \in \mathbb{R}^{d'} \text{ (point-wise equation)}$$
$$e^{\ell=0} = \text{LL}(e_0) \in \mathbb{R}^{E\times d} \text{ (input edge feature)}$$

```
# class graph multi head attention layer
class graph_MHA_layer(nn.Module): # MHA = Multi Head Attention

    def __init__(self, hidden_dim, head_hidden_dim, num_heads): # hidden_dim = d
        super().__init__()
        self.head_hidden_dim = head_hidden_dim # head_hidden_dim = d' = d/K
        self.num_heads = num_heads # number of heads = K
        self.WQ = nn.Linear(hidden_dim, head_hidden_dim * num_heads, bias=True) # define K x W matrix of size
        self.WK = nn.Linear(hidden_dim, head_hidden_dim * num_heads, bias=True)
        self.WV = nn.Linear(hidden_dim, head_hidden_dim * num_heads, bias=True)
        self.WE = nn.Linear(hidden_dim, head_hidden_dim * num_heads, bias=True)
```





Lab 4 : Dense GT with edge features

- **Dense Graph Transformer with edge features and PyTorch**

Lecture : Graph Transformer

Lab 04 : Graph Transformers with edge features and PyTorch (dense linear algebra)

Xavier Bresson

Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
https://arxiv.org/pdf/2012.09699.pdf

```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/CS5284_2024_codes/codes/10_Graph_Transformers'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
    !pip install dgl # Install DGL
```

```
# Libraries
import pickle
import matplotlib.pyplot as plt
import torch
from torch.utils.data import DataLoader
import torch.nn as nn
import time
import networkx as nx
import sys; sys.path.insert(0, 'lib/')
from lib.utils import compute_ncut
from lib.molecules import Dictionary, MoleculeDataset, MoleculeDGL, Molecule
import os, datetime
from rdkit import Chem
from rdkit.Chem import Descriptors
from rdkit.Chem import rdmolops
from rdkit import RDLogger
```

Design the class of dense GraphTransformer networks with edge features

Node update equation:

$$\bar{h}^{\ell} = h^{\ell} + \text{gMHA}(\text{LN}(h^{\ell}), \text{LN}(e^{\ell})) \in \mathbb{R}^{N\times d}$$
$$h^{\ell+1} = \bar{h}^{\ell} + \text{MLP}(\text{LN}(\bar{h}^{\ell})) \in \mathbb{R}^{N\times d}$$
$$\text{with gMHA}(h,e) = \text{Concat}_{k=1}^{H}(\text{gHA}(h_k,e_k))W_O \in \mathbb{R}^{N\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, e_k \in \mathbb{R}^{N\times N\times d'}, W_O \in \mathbb{R}^{d\times d}$$
$$\text{gHA}(h,e)_i = \sum_{j\in V} \underbrace{\frac{\exp(q_i^T \text{diag}(e_{ij})k_j/\sqrt{d'})}{\sum_{j'\in V}\exp(q_i^T \text{diag}(e_{ij'})k_{j'}/\sqrt{d'})}}_{\text{graph attention score w/ edge feature}_{ij}} v_j \ \text{(point-wise equation)}$$
$$Q = h_k W_Q, K = h_k W_K, V = h_k W_V \in \mathbb{R}^{N\times d'=d/H}, E = e_k W_E \in \mathbb{R}^{E\times d'=d/H}, W_Q, W_K, W_V, W_E \in \mathbb{R}^{d'\times d'}$$
$$h^{\ell=0} = \text{LL}_1(h_0) + \text{LL}_2(p_0) \in \mathbb{R}^{N\times d} \text{ (input node feature and positional encoding)}$$
$$\text{with } p_0 = \Phi_{\{2,..,K+1\}} \in \mathbb{R}^{N\times K},\ \Delta = \Phi\Lambda\Phi^T \in \mathbb{R}^{N\times N}$$

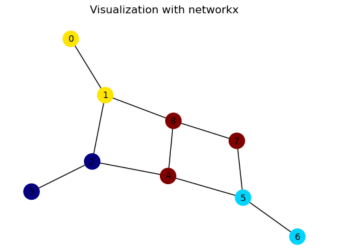
Edge update equation:

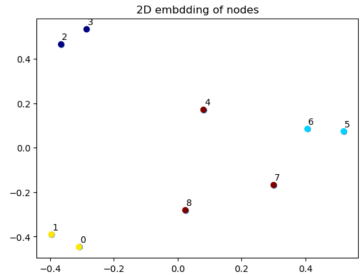
$$\bar{e}^{\ell} = e^{\ell} + \text{gMHE}(\text{LN}(e^{\ell}), \text{LN}(h^{\ell})) \in \mathbb{R}^{N\times N\times d}$$
$$e^{\ell+1} = \bar{e}^{\ell} + \text{MLP}(\text{LN}(\bar{e}^{\ell})) \in \mathbb{R}^{N\times N\times d}$$
$$\text{with gMHE}(e,h) = \text{Concat}_{k=1}^{H}(\text{gHE}(e_k,h_k))W_O^e \in \mathbb{R}^{N\times N\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, e_k \in \mathbb{R}^{N\times N\times d'}, W_O^e \in \mathbb{R}^{d\times d}$$
$$\text{with gHE}(e,h)_{ij} = q_i \odot e_{ij} \odot k_j/\sqrt{d'} \in \mathbb{R}^{d'} \text{ (point-wise equation)}$$
$$e^{\ell=0} = \text{LL}(e_0) \in \mathbb{R}^{N\times N\times d} \text{ (input edge feature)}$$

Differences between sparse GT and dense GT:

- The edge features are now dense : Sparse $e \in \mathbb{R}^{E\times d}$ to $e \in \mathbb{R}^{N\times N\times d}$.
- The absence of edges in the graph is now considered as a bond type e.g. `None` type with integer value `0` .
- The attention function is now connected to all nodes in the graph : Sparse attention with $\sum_{j\in\mathcal{N}_i}$ to dense attention $\sum_{j\in V}$.

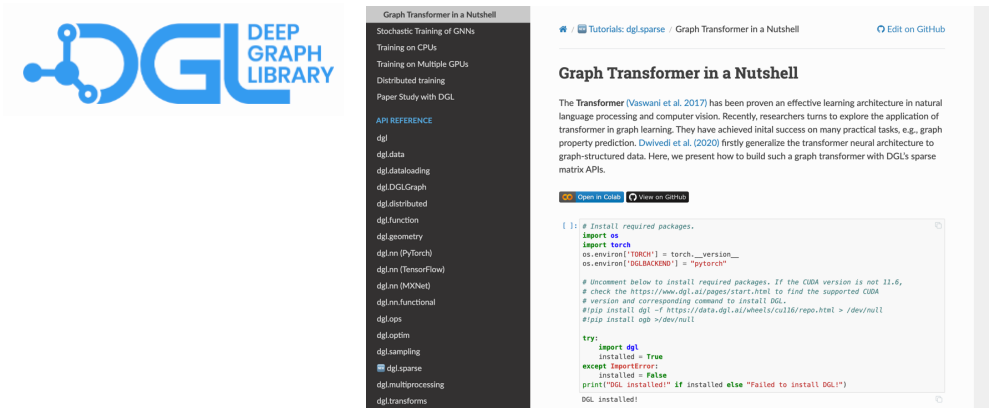
```
# Global constants
num_heads = 8; d = 16*num_heads; num_layers = 4; drop = 0.0; bs = 50
print('d, num_heads, num_layers, drop : ', d, num_heads, num_layers, drop)
```





Graph Transformer

- Further reading and code :
  - Blogpost (Vijay Dwivedi) : https://towardsdatascience.com/graph-transformer-generalization-of-transformers-to-graphs-ead2448cff8b
  - GitHub repo (1K∗) : https://github.com/graphdeeplearning/graphtransformer
  - Andrew Ng's article : https://www.deeplearning.ai/the-batch/issue-151
  - DGL tutorial : https://docs.dgl.ai/en/latest/notebooks/sparse/graph_transformer.html

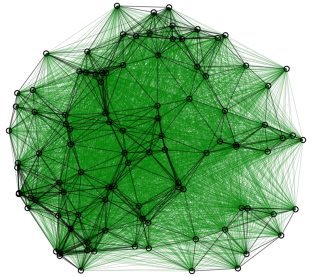




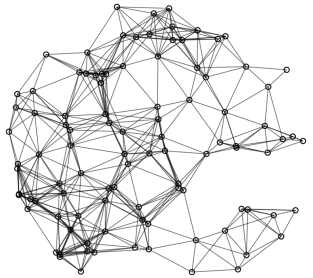


Scaling GT to large graphs

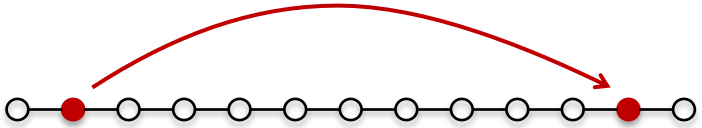
- Dense GT is expensive on large graphs since global attention requires $O(n^2)$ pairwise interactions.
- Sparse GTs may perform poorly on large graphs because long-range communication requires stacking many layers, which increase over-squashing and aggregation overload.
- How can we model long-range interactions when graphs contain millions or more nodes?
- The same scalability problem arises for standard Transformer on very long sequences : Global attention requires $O(n^2)$ compute and memory.
- A similar issue in vision : Global pixel-level attention on a 1,000 × 1,000 image would involve $N = 10^6$ pixel-positions.
- Proposed solution: Hierarchical architectures provide shorter long-range communication paths at lower cost and mitigate over-squashing.



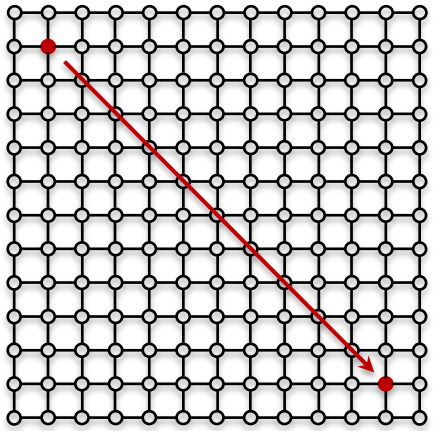
Dense GT



Sparge GT



Suppose the sequence has n = 1M word tokens



Suppose the image has n = 1M pixels

Outline

- Review of graph network architectures
- Graph positional encoding
- Message-passing and over-squashing
- Graph Transformer
- **Graph ViT / Hierarchical Transformer**
- Conclusion

Transformer for images

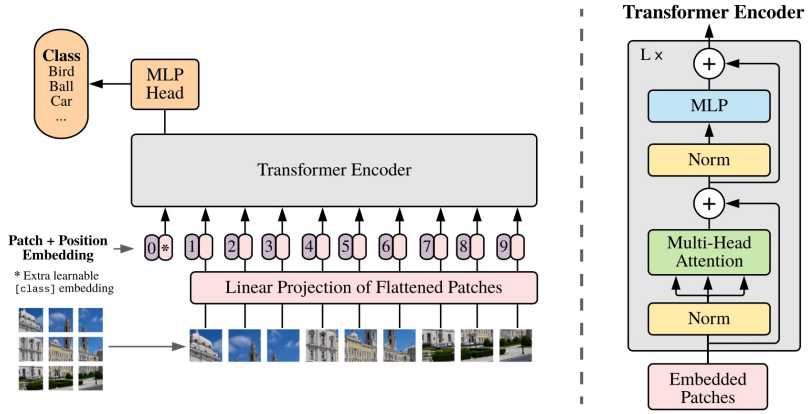
- Vision Transformer (ViT) architecture[1] :
  - Partition the image into a regular grid of small non-overlapping patches.
  - Flatten each patch and project it into a d-dimensional token embedding.
  - Add positional embeddings to the patch tokens and process the resulting sequence with standard Transformer encoder / self-attention blocks.
  - Use a classification token or pooled representation followed by an output head for the downstream task.



n = 256 x 256
≈ 66K pixels



p = 256 patches
of size 16 x 16 pixels
p ≪ n patches



Architecture

[1] Dosovitskiy-etal, An image is worth 16x16 words: Transformers for image recognition at scale, 2020

ViT advantages and limitations

- Advantages
  - Patchifying / domain sub-sampling reduces the number of attention tokens from n pixels to $p \ll n$ patches, reducing global-attention cost from $O(n^2)$ to $O(p^2)$.
  - For example, a (1,000×1,000)-pixel image has 10^6/1M pixels, and using (100×100)-pixel patches gives 10×10 = 100 patch tokens.
  - After embedding patches into fixed d-dim vectors, global self-attention provides direct long-range interactions between patches, mitigating the long communication paths in images.
- Limitations
  - ViT generally requires longer training and more compute per epoch compared to CNN.
  - The exact ratio depends on model size and hardware, but for ResNet-34/21.8M param vs. ViT-S/22.1M param, the training throughput is 1378 img/s vs. 625 img/s, and the time per epoch is 15.5min vs. 34.2 min[1].
  - When training budget and modern training recipes are used, DeiT-S and ResNet-50 achieve similar ImageNet-1k accuracy (76.8% vs. 76.9% at 100 epochs)[2].

[1] Touvron-etal, Training Data-Efficient Image Transformers & Distillation through Attention, 2021
[2] Bai-etal, Are Transformers More Robust Than CNNs?, 2021

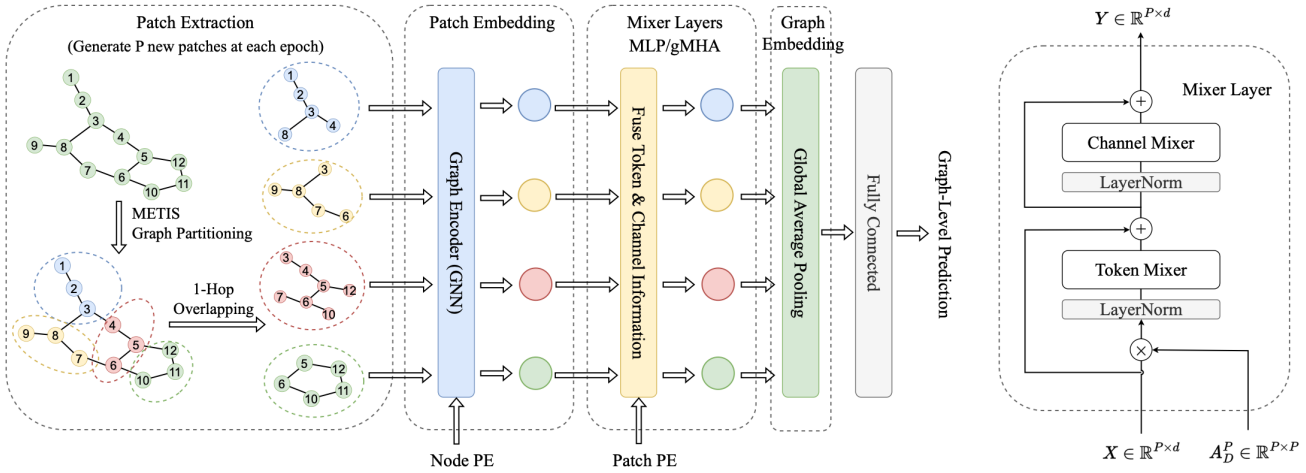
Generalizing ViT from images to graphs

- We aim at extending the ViT patch-based model from regular image grids to irregular graphs.
- Main challenge : Graphs are variable-size and unordered, requiring meaningful permutation-invariant graph patches/tokens.
- Advantages
  - GraphViT[1] captures long-range dependencies and empirically mitigates over-squashing.
  - Graph patch construction can be performed in near-linear time, typically $O(n_E)$, using graph partitioning / coarsening methods such as METIS or Heavy-Edge Matching (HEM)[2].
  - The overall complexity of GraphViT is $O(n+n_E)$, similar to MP-GNNs.
  - Global attention is applied to only $p \ll n$ graph patches: $O(p^2d)$ compute and $O(p^2+pd)$ memory, instead of $O(n^2d)$ and $O(n^2+nd)$ for node-level global attention.
  - GraphViT empirically demonstrates high graph-isomorphism expressivity, distinguishing challenging graphs beyond standard message-passing GNNs.

[1] He-Bresson-LeCun,, A Generalization of ViT/MLP-Mixer to Graphs, 2022
[2] Karypis-Kumar, A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, 1998

Four-stage architecture

- GraphViT generalizes patch-based ViT paradigm from regular grids to graphs in four key steps :
  - Step 1 / Graph patch extraction : Partition the graph into overlapping patches.
  - Step 2 / GNN patch embedding : Encode each graph patch into a token representation.
  - Step 3 / Positional encoding : Inject node- and patch-level structural information.
  - Step 4 / Token mixing + readout : Process patch tokens with Graph Transformer layers, followed by global pooling and a prediction head.



Generic architecture of Graph ViT

Step 1 : Domain partitioning

- For images:
  - Pixels lie on a regular 2D lattice.
  - The lattice can be efficiently partitioned into fixed-size, non-overlapping, rectangular patches, via fast pixel reordering.

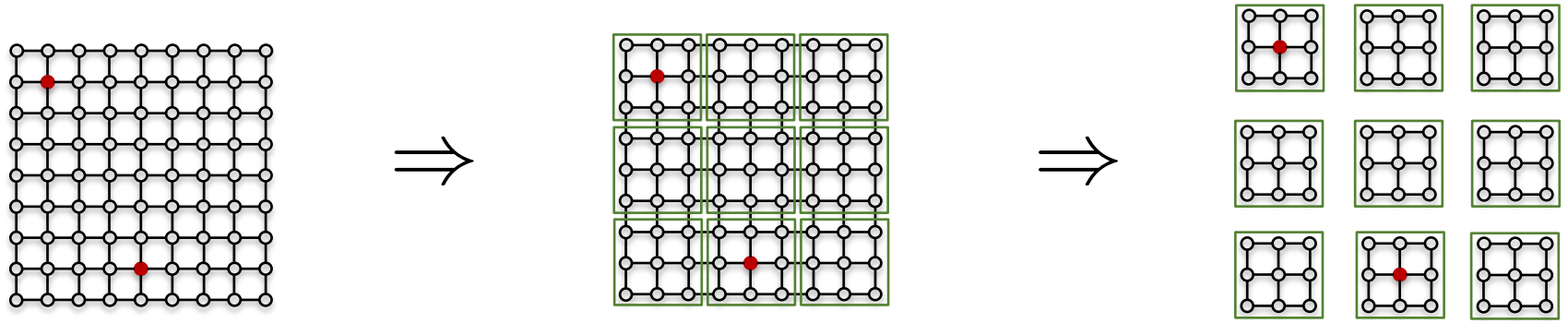


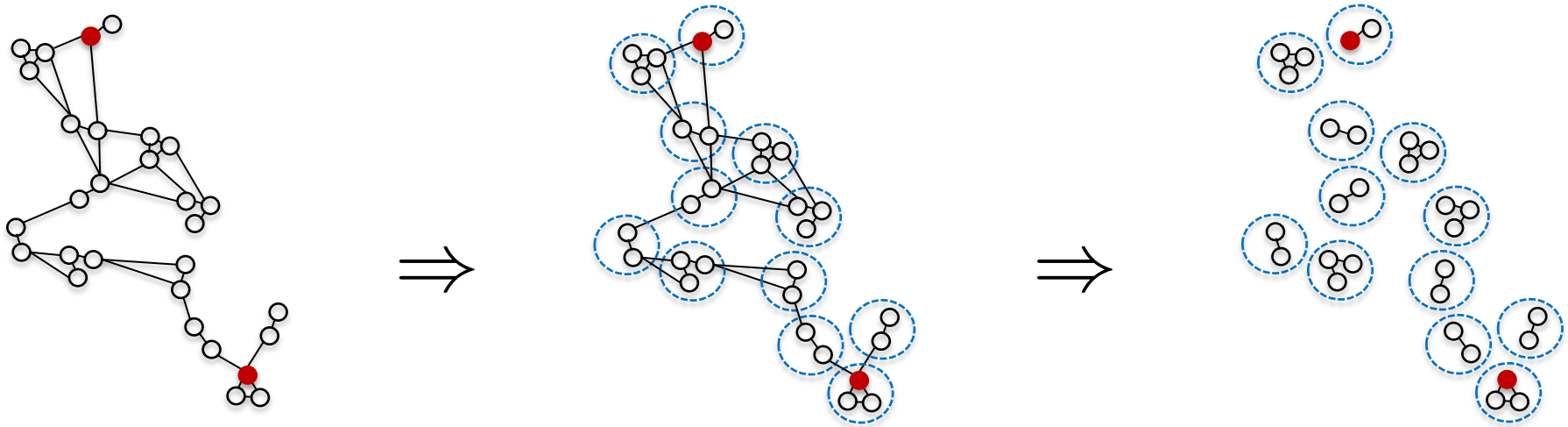
Image domain

Image partitioning

Image patches

Graph partitioning

- For graphs:
  - Graphs are irregular and lack a canonical grid-based partition into fixed-size patches.
  - Patch extraction can be formulated as a graph partitioning problem, which is a well-known task in graph theory[1].



Graph domain

Graph partitioning

Graph patches

[1] Chung, Spectral Graph Theory, 1996

Graph partitioning for patch extraction

- Graph partitioning has a long history with multiple algorithmic approaches[1].
- Partitioning objectives are usually NP-hard, motivating efficient approximation methods.
- For graph patch extraction, we seek a partitioning method with the following properties :
  - Applicable to arbitrary irregular graphs.
  - Produces patches with high intra-connectivity edges (clusters / communities) and few inter-patch edges (patches are well separated).
  - Scales efficiently to large sparse graphs, ideally with $O(n_E)$ or near-linear complexity.
- For a single-level partition, we use METIS[2] to obtain high-quality graph clusters[3].
- For multi-level hierarchies, we use Heavy-Edge Matching (HEM)[2] to recursively coarsen the graph, producing a sequence of increasingly coarse graphs.

[1] Von Luxburg, A tutorial on spectral clustering, 2007
[2] Karypis-Kumar, A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, 1998
[3] He-Bresson-LeCun, A Generalization of ViT/MLP-Mixer to Graphs, 2022

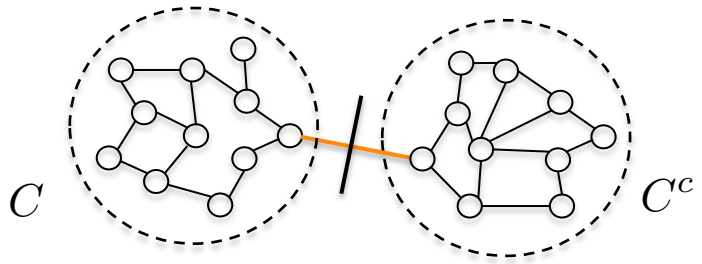
Graph partitioning with METIS

- METIS[1] is a fast graph partitioning method with three main stages :
  - Coarsen the graph, e.g. using Heavy-Edge Matching or related matching heuristics.
  - Partition the coarsest graph while minimizing edge cut subject to balance constraints.
  - Un-coarsen and refine the partition at progressively finer graph resolutions.
- The METIS objective favors few inter-partition edges while maintaining approximately balanced partitions :

$$\min_{C,C^c} \ \mathrm{Cut}(C, C^c) \ \ \text{s.t.} \ \ \mathrm{Vol}(C) \approx \mathrm{Vol}(C^c)$$

$$\text{where } \mathrm{Cut}(R, R^c) = \sum_{i \in R, i \in R^c} A_{ij} \ \ \text{cut function}$$

$$\text{and } \mathrm{Vol}(S) = \sum_{i \in S} d_i \ \ \text{volume function}$$



Constrained Cut loss
Cut(C,C^c) = 1
Vol(C) = 29
Vol(C^c) = 27

- METIS provides high-quality partitions with linear $O(n_E)$ cost on large sparse graphs.

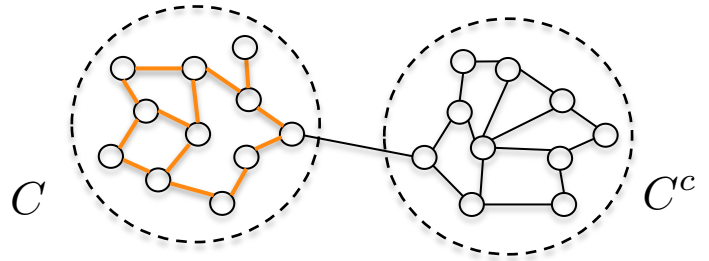
[1] Karypis-Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, 1998

Graph coarsening with HEM

- HEM[1] is a local graph coarsening heuristic that preferentially matches nodes connected by heavy edges.
- By constraining strongly connected node pairs, HEM locally favors high within-cluster association :

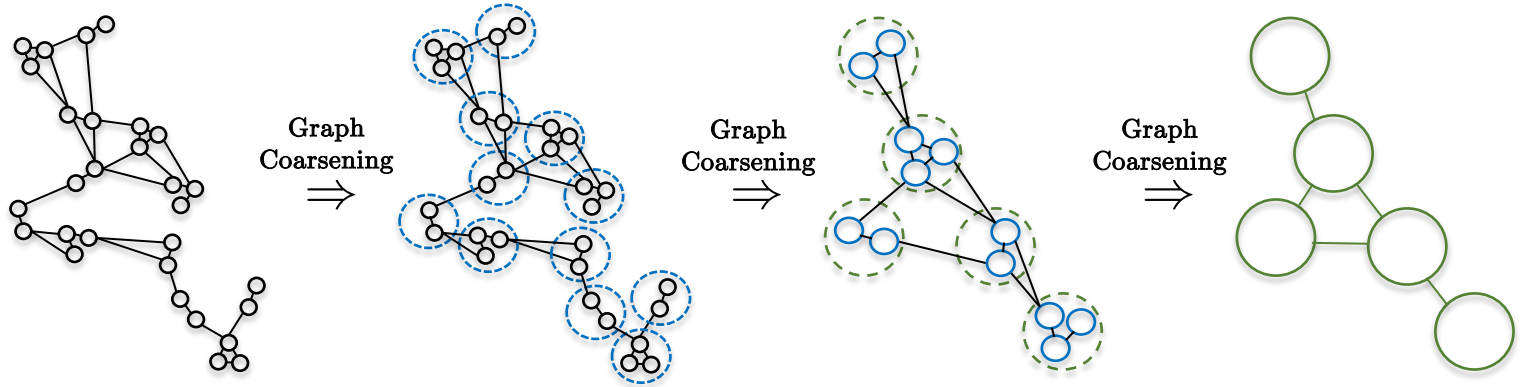
$$\max_{C,C^c} \ \text{Assoc}(C,C) + \text{Assoc}(C^c,C^c),$$

$$\text{where Assoc}(R,R) = \sum_{i\in R, j\in R} A_{ij} \quad \text{association function}$$



Associative loss
Assoc(C,C) = 15
Assoc(C^c,C^c) = 13
Total = 27

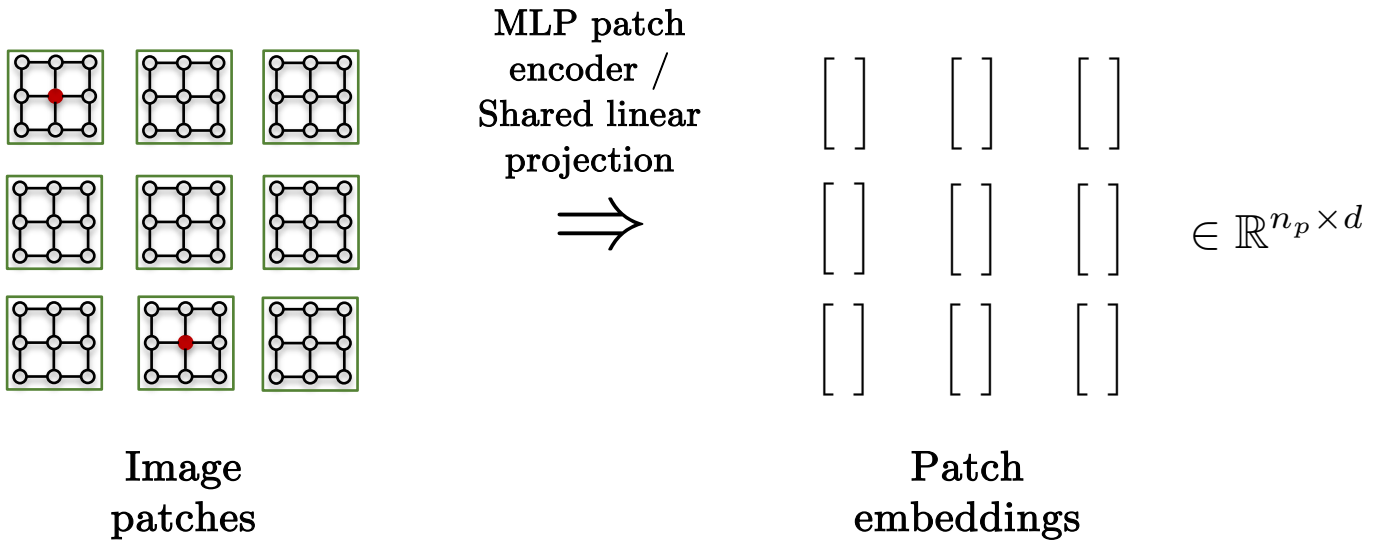
- A HEM coarsening pass runs in $O(n_E)$ time on a sparse graph.
- Repeated HEM coarsening produces a hierarchy of progressively smaller / coarser graphs.



[1] Karypis-Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, 1998

Step 2 : From patches to tokens

- Step 2 : Encode patches into a vectorial representation
- For images :
  - Equal-size p × p patches are flattened and mapped by a shared linear projection from dimension p^2 (= 9 pixels) × c (= 3 colors) to dimension d.
  - This shared projection is equivalent to a convolution with kernel size p × p, stride p, and d output channels, enabling efficient implementation.



Graph patch embedding

- For graphs :
  - Graph patch encoders must handle irregular structure and variable numbers of nodes/edges, while being invariant to node ordering and discriminative across different graph patches.
  - As a result, we encode each patch with a message-passing GNN in three stages : First, node/ edge feature embedding, second graph convolution or attention layers, and third permutation-invariant pooling / readout.
  - Because message passing is restricted to small graph patches, communication paths remain short, substantially reducing long-range over-squashing and aggregation over-loading.

For patch $p = 1, ..., n_p$, node i and edge ij :

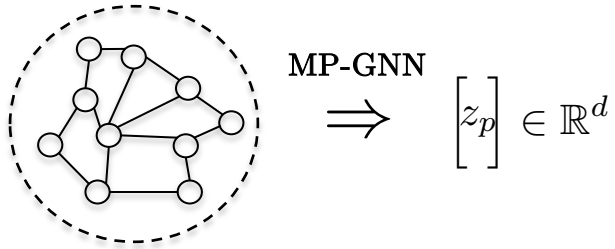
$$h^0_{p,i} = W^0_x x_{p,i} + W^0_p \mathrm{p}_{p,i} \in \mathbb{R}^d, \quad e^0_{p,ij} = W^0_e e_{p,ij} \in \mathbb{R}^d$$

For $\ell = 0, 1, ..., L-1$:

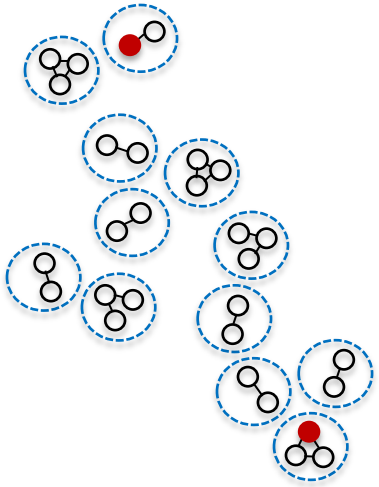
$$h^{\ell+1}_{p,i} = f_{\text{node}}\big(h^\ell_{p,i}, \{h^\ell_{p,j}, e^\ell_{p,ij} : j \in \mathcal{N}_i\}\big) \in \mathbb{R}^d$$

$$e^{\ell+1}_{p,ij} = f_{\text{edge}}\big(e^\ell_{p,ij}, h^\ell_{p,i}, h^\ell_{p,j}\big) \in \mathbb{R}^d$$

$$z_p = \mathrm{MLP}\Big(\frac{1}{|\mathcal{V}_p|} \sum_{i \in \mathcal{V}_p} h^L_{p,i}\Big) \in \mathbb{R}^d \quad \text{patch embedding}$$



From graph patches to graph tokens



Graph patches

$\{G_p\}_{p=1}^{n_p}$

Shared MP-GNN

$\Rightarrow$

$[\,][\,][\,][\,][\,][\,][\,]$

$z = [z_1, ..., z_{n_p}]^T \in \mathbb{R}^{n_p \times d}$

Graph patch embeddings

$\{z_p\}_{p=1}^{n_p} \in \mathbb{R}^d$

Step 3 : Multi-scale positional encodings

- Inject positional / structural information to increase the expressive power of the model.
- For images:
  - The regular 2D grid provides canonical coordinates for pixels and image patches.
- For graphs:
  - Arbitrary graphs have no canonical ordering or coordinates for nodes or graph patches.
  - GraphViT uses PEs at two scales : Node-level PE on the original graph and patch-level PE on the coarsened graph.
  - For a multi-level hierarchy, coarse graphs are constructed recursively using the assignment matrices. And, at each level, a structural PE is computed from the corresponding coarse graph.

Hierarchical graphs

First coarsened graph :

$A^1 = P_0^T A_0 P_0,\ A^0 = A$ (original graph), $P_0 \in \{0,1\}^{n_0 \times n_1},\ n_0 = n,\ n_1 = n_p,$

where P_0 is the node-to-patch assignment matrix :

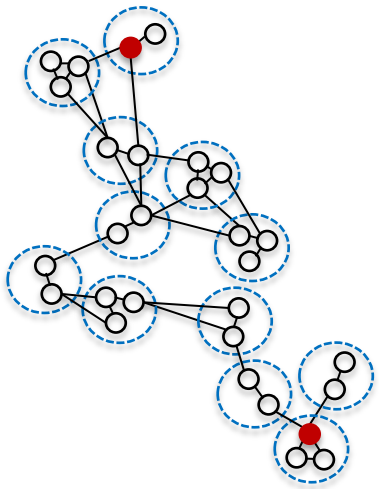
$$(P_0)_{ik} = \begin{cases} 1, & \text{if node } i \text{ belongs to patch } G_k^0 = G_k, \\ 0, & \text{otherwise.} \end{cases}$$

Equivalently, $A^1_{kl} = \sum_{i \in G_k} \sum_{j \in G_l} A_{ij} = \text{Assoc}(G_k, G_l).$

For hierarchical levels $h = 0, ..., n_h - 1$:

$A^{h+1} = P_h^T A^h P_h,$ where $P_h \in \{0,1\}^{n_h \times n_{h+1}}$ maps level-h super-nodes to level-(h+1) super-nodes.

$$(P_h)_{rs} = \begin{cases} 1, & \text{if super-node } r \text{ belongs to super-cluster } G_s^h, \\ 0, & \text{otherwise.} \end{cases}$$



Original graph and its first coarse communities

Hierarchical graphs with disconnected communities

For hierarchical GT, we will need to cut the graphs into disconnected communities.

The disconnected coarsened graphs are computed by removing removing all edges between different communities:

$$A^h_{\text{cut}} = A^h \odot (P_h P_h^T) \in \mathbb{R}^{n_h \times n_h}$$

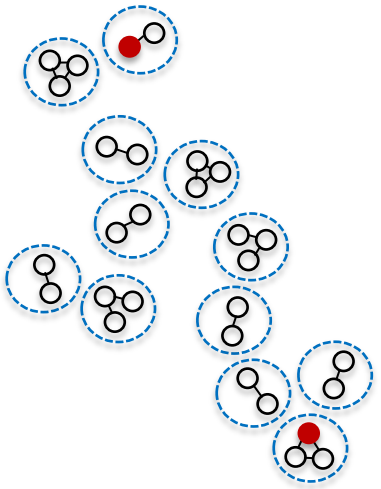
$$\text{with } \left(A^h_{\text{cut}}\right)_{ij} = \begin{cases} A^h_{ij}, & i,j \in G^h_k \text{ for some } k, \\ 0, & \text{otherwise.} \end{cases}$$

After reordering nodes by community, A^h_{cut} is block diagonal :

$$A^h_{\text{cut}} = \begin{bmatrix} A_{G^h_1} & 0 & \cdots & 0 \\ 0 & A_{G^h_2} & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & A_{G^h_{n_{h+1}}} \end{bmatrix}$$

and the graph becomes a union of disconnected subgraphs :

$$G^h_{\text{cut}} = G^h_1 \cup G^h_2 \cup \cdots \cup G^h_{n_{h+1}}$$



Graph of disconnected communities

Hierarchical feature pooling

The same assignment matrices $P_h, h = 0, ..., n_h - 1$ pool node features across hierarchical levels :

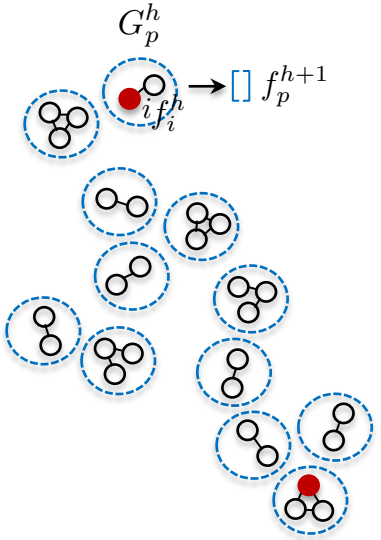
$$f^{h+1} = D_h^{-1} P_h^T f^h, \text{ where } f^{h+1} \in \mathbb{R}^{n_{h+1} \times d}, \ f^h \in \mathbb{R}^{n_h \times d},$$

$$P_h \in \{0,1\}^{n_h \times n_{h+1}}, \ D_h = \text{diag}\left(P_h^T \mathbf{1}\right) \in \mathbb{R}^{n_{h+1} \times n_{h+1}} \text{ (cluster-size matrix)}$$

Thus, for each community G_p^h, the corresponding super-node feature is

$$f_p^{h+1} = \frac{1}{|G_p^h|} \sum_{i \in G_p^h} f_i^h \in \mathbb{R}^d$$

For disconnected coarsened graphs, the pooling equations are unchanged, since pooling depends on community assignments P_h, not inter-community edges.



Feature pooling with communities

Step 4 : Transformer blocks and task prediction

- Mix patch representations and compute the final prediction.
- For images :
  - Patch embeddings and their positional encodings are processed by multiple Transformer blocks, followed by global pooling / readout and a prediction head.

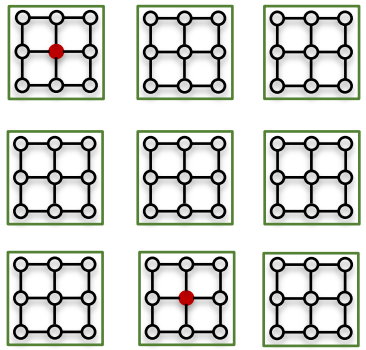


Image patches

MLP patch encoder / Shared linear projection

$\Rightarrow$



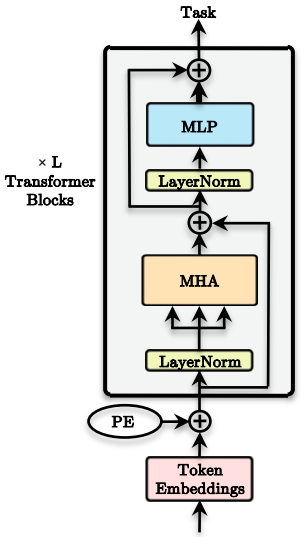
$$z = [z_1, ..., z_{n_p}]^T \in \mathbb{R}^{n_p \times d}$$

Image patch embeddings

$$\{z_p\}_{p=1}^{n_p} \in \mathbb{R}^d$$

Transformer

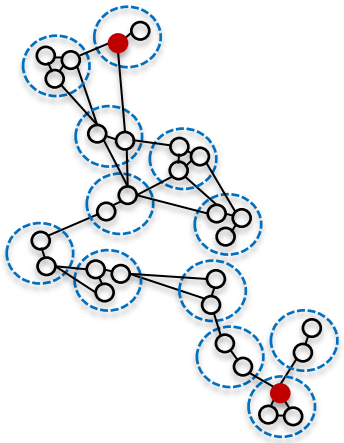
$\Rightarrow$



Patch-level Transformer + Readout function

1-level hierarchical Graph Transformer

- For graphs : Graph patch embeddings and their PEs are processed by multiple patch-level Graph Transformer blocks, followed by global pooling / readout and a prediction head.



Graph patches

$\{G_p\}_{p=1}^{n_p}$

Shared MP-GNN

$\Rightarrow$

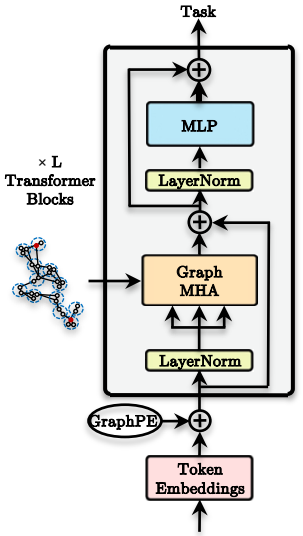
$z = [z_1, ..., z_{n_p}]^T \in \mathbb{R}^{n_p \times d}$

Graph patch embeddings

$\{z_p\}_{p=1}^{n_p} \in \mathbb{R}^d$

Graph Transformer

$\Rightarrow$



Patch-level Transformer + Readout function

$h^0 = z + f_{\text{linear}}(\text{PE}_{\text{patch}}) \in \mathbb{R}^{n_p \times d}$

For $\ell = 0, 1, ..., L-1$:

$h^{\ell+1} = \text{GT\_layers}(h^\ell, A^1) \in \mathbb{R}^{n_p \times d}$

Prediction : $\hat{y} = \text{MLP}(\text{mean}(\{h^L\})) \in \mathbb{R}^K$

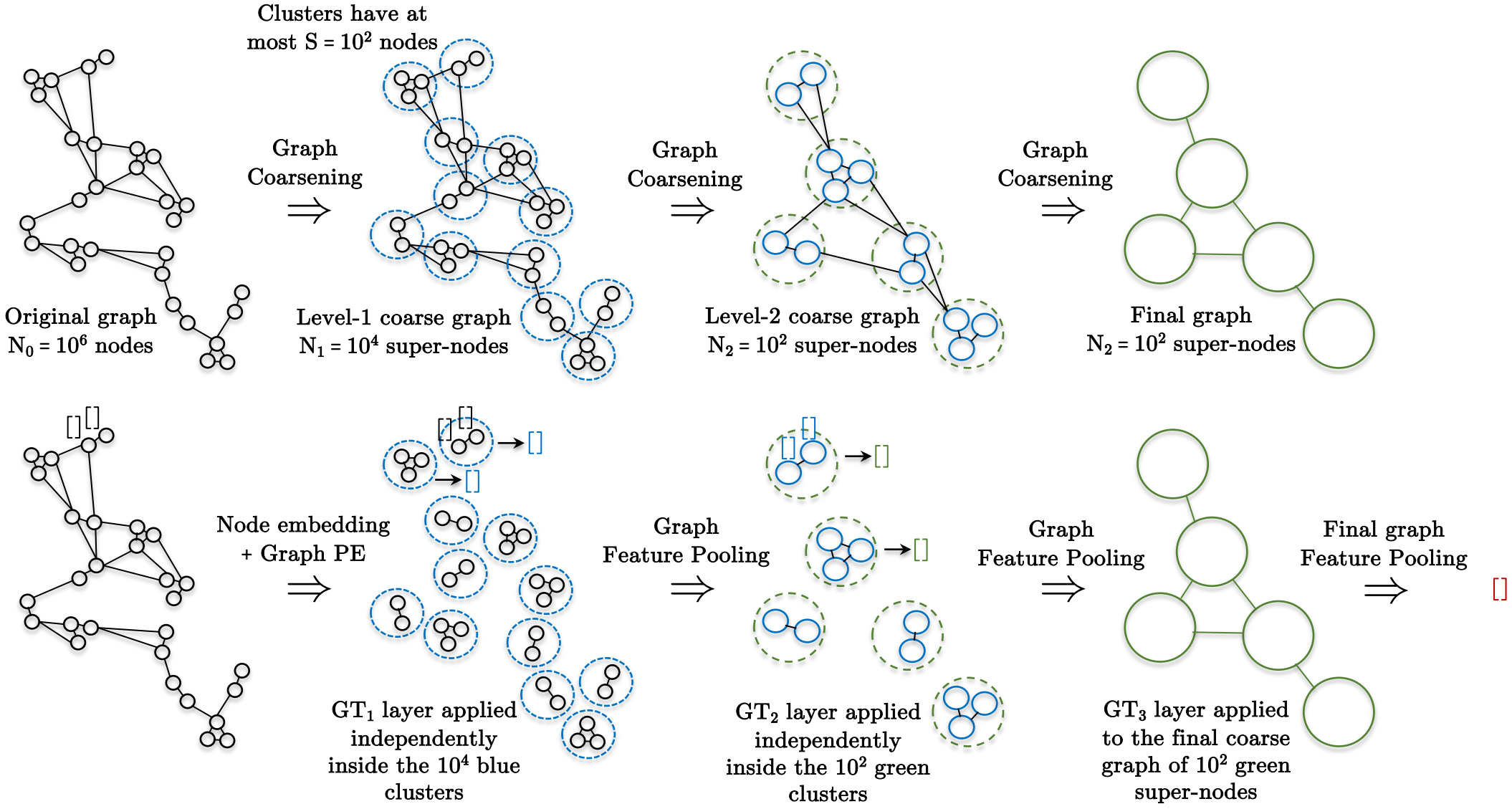
From 1-level to H-level hierarchical GT

- The hierarchy limits the number of nodes processed by each local GNN/GT block, controlling aggregation size and helping mitigate over-squashing.
- For example, consider a graph with 1,000,000 nodes and choose a maximum local-attention size of S = 100 nodes, i.e. the GT layer is applied to a maximum of S nodes.
- Repeated size-constrained graph coarsening (e.g. HEM with a cluster-size bound) builds a hierarchy of local communities containing at most S nodes / super-nodes.
  - Level 0 : Original graph with $N_0 = 1{,}000{,}000$ nodes.
  - Level 1 : Contains approx $N_1 = N_0/S = 10{,}000$ clusters, with each cluster containing at most S = 100 nodes. After pooling, the clusters become 10,000 super-nodes.
  - Level 2 (final) : Approx $N_2 = N_1/S = 100$ clusters, with each cluster containing at most S = 100 super-nodes. After pooling, the clusters become 100 super-nodes.

Hierarchical GT design

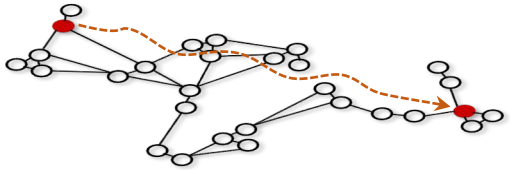
- HGT architecture :
  - Input (layer 0) : Embed the $N_0 = 10^6$ original nodes,
  - Local GT (layer 1) : Process the $N_1 = 10^4$ disconnected clusters independently and in parallel, each cluster containing at most $S = 10^2$ nodes,
  - Global GT (layer 2) : Apply full attention to the final $N_2 = 10^2$ super-nodes,
  - Final layer : Use a task-specific prediction head.
- HGT bounds local attention cost while progressively shortening long-range communication paths across the hierarchy.

2-level hierarchical GT

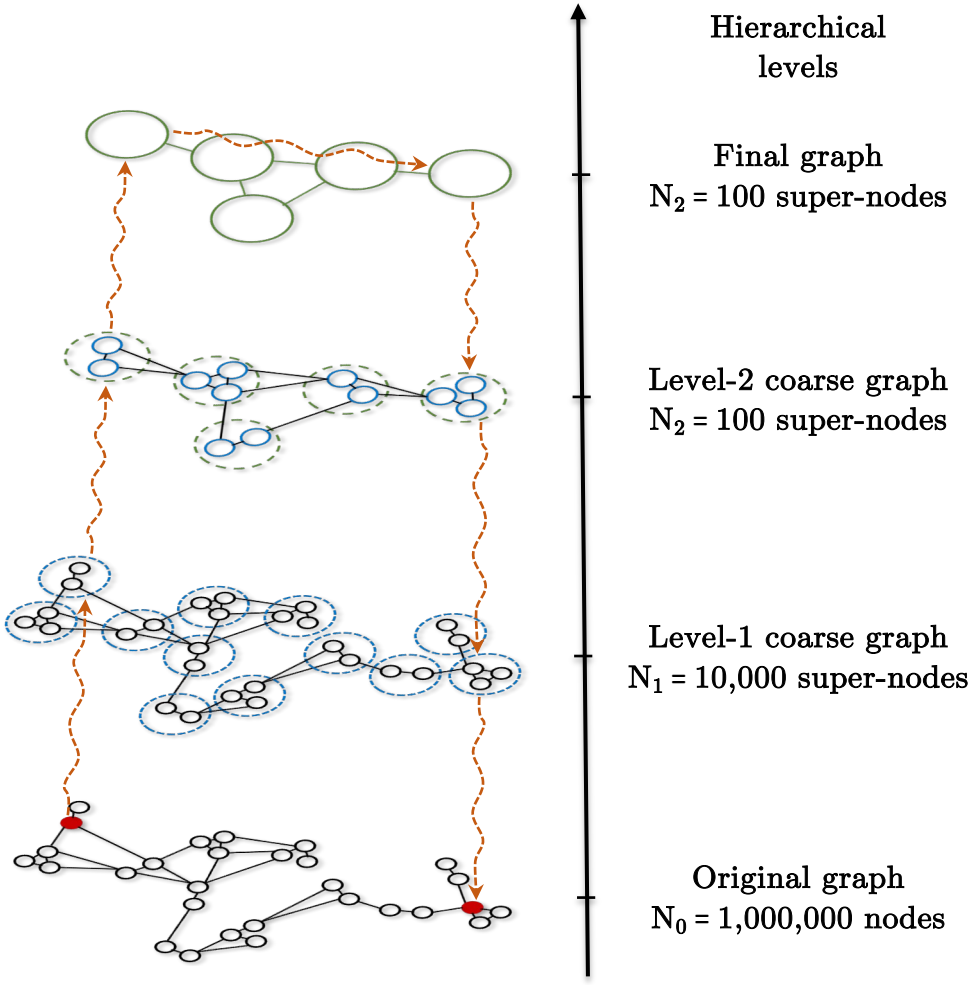


Hierarchical GT shortens communication paths

- Consider a graph with $n = 10^6$ nodes and a maximum local community size $S = 100$.
- How many hops are needed for two distant red nodes, separated by a shortest path of length $O(n)$, to exchange information?



Flat Graph Transformer
requires $O(n = 10^6)$ number of
hops (worst case)



Hierarchical Graph Transformer
requires $O(\log_S(n) = 100)$
number of hops

Complexity advantage of HGT

- Dense GT : Global attention over n nodes costs $O(n^2)$ attention interactions.
- Hierarchical GT with maximum cluster / super-node size S
  - Node embedding (layer 0) : $O(n)$
  - Local GT (layer 1) : $O(n/S.S^2)$ as we have $N_1 = n/S$ independent GT layers applied on S nodes.
  - Global GT (layer 2) : $O(S^2)$ as we have one final GT layer applied on $N_2 = S$ nodes.
  - Total is $O(n + n/S.S^2 + S^2) = O(n + n.S + S^2) = O(n)$ for fixed $S \ll n$.
- Including feature dimension d and L dense GT layers, we have :
  - GT is $O(n^2.L.d)$ vs. HGT $O(n.S.d)$ (independent of the number of hierarchical levels)
- HGT replaces $O(n^2)$ global attention with $O(n)$ linear hierarchical attention !

HGT complexity (details)

Select H such that $\lceil \frac{n}{S^L} \rceil = 1$, and $N_h = \lceil \frac{n}{S^h} \rceil$ be the number of clusters per hierarchical level, so that the final level has $N_H = \lceil \frac{n}{S^L} \rceil = 1$ cluster.

We have $(H-1)\times$ local GT layers, and $1\times$ global GT layer.

The complexity for the $(H-1)\times$ local GT layers is :

$$O\Big(\sum_{h=1}^{H-1} N_h S^2 d\Big), \text{ where } O(S^2 d) \text{ is the attention cost,}$$

$$= O\big(S^2 d \sum_{h=1}^{H-1} \frac{n}{S^h}\big), \text{ ignoring the ceiling operation } \lceil\cdot\rceil,$$

$$= O\big(nS^2 d \sum_{h=1}^{H-1} \frac{1}{S^h}\big) = O\big(nS^2 d \frac{1}{S}\big), \quad \text{given } \sum_{h=1}^{H-1} \frac{1}{S^h} = \frac{1 - S^{-(H-1)}}{S-1} \simeq \frac{1}{S},$$

$$= O\big(nSd\big)$$

And the complexity for the final global GT layer is :

$$O\big(N_H S^2 d\big) = O\big(S^2 d\big), \quad \text{as } N_H = 1$$

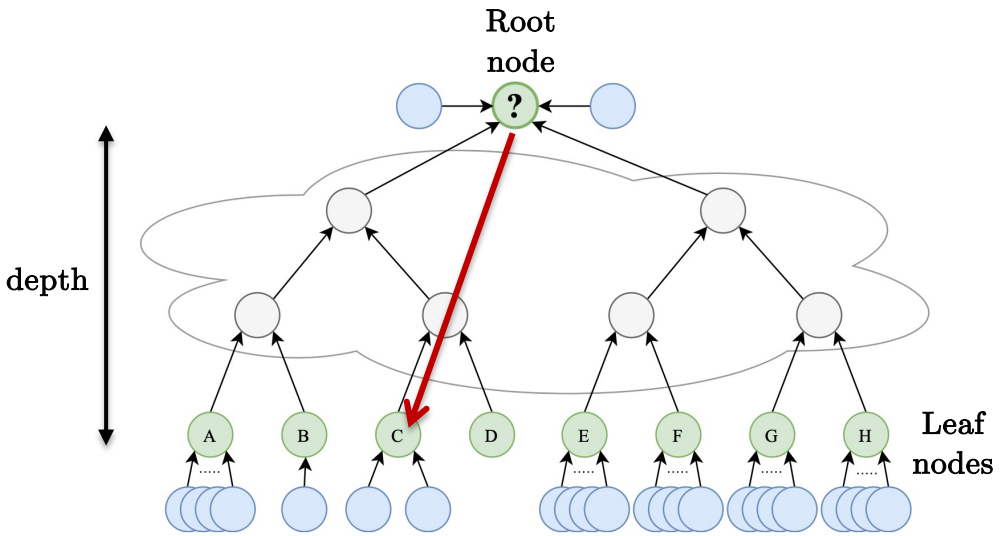
Including node embedding,

$$T_{\text{HGT}} = O\big(n + nSd + S^2 d\big) = O\big(nSd\big), \text{ independent of } H.$$

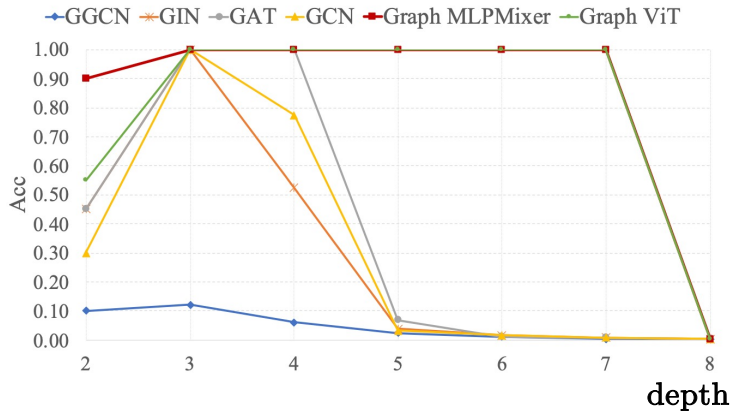
Evaluating model's over-squashing

- TreeNeighbourMatch[1] is a synthetic benchmark designed to expose over-squashing as the receptive field grows exponentially with tree depth.
- The task is to predict the label of the leaf node whose degree (number of blue neighbors) matches that of the root node.
- To succeed, the network must communicate across long distance (from the root to the leaf) and process information from $O(2^{L+1})$ aggregated nodes.
- Standard MP-GNNs fail as depth increases, whereas Graph ViT / Hierarchical GT mitigates over-squashing and generalizes well up to depth = 7.

Random assignment of $\{1,2,...,2^d\}$ blue neighbors to the root node. The goal is to predict the class of the root node.



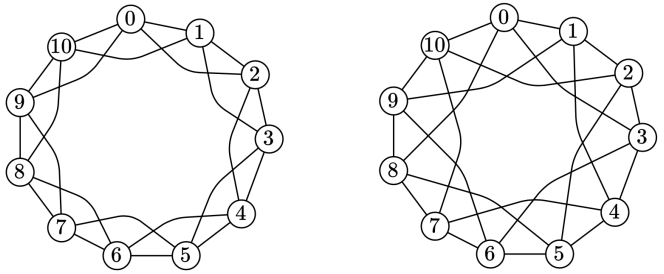
Test Accuracy w.r.t. the tree depth



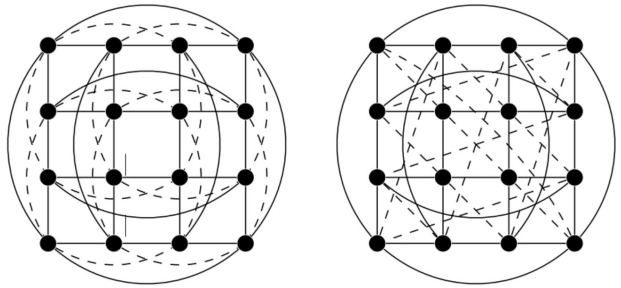
[1] Alon-Yahav, On the bottleneck of graph neural networks and its practical implications, 2020

Empirical high graph isomorphism expressivity

- Empirically, Graph ViT distinguishes non-isomorphic graphs from benchmarks designed to defeat 1-WL[1], 2-WL[2] and 3-WL[3], while standard MP-GNNs fail on these tasks.
- Graph-isomorphism benchmarks : Circular Skip Link (CSL) dataset[1] contains 150 4-regular graphs (1-WL failed) divided into 10 isomorphism classes. EXP dataset[2] has 1200 1&2-WL failed graphs that are split into two classes. SR25 dataset[3] contains 15 strongly regular 3-WL failed graphs with 25 nodes each.
- These results are empirical and do not establish a general analytical equivalence over k-WL.



Non-isomorphic graphs where 1-WL test failed to distinguish (w/ skip-link 2 and 3).



Non-isomorphic graphs where 3-WL test failed to distinguish (strongly regular graphs)

Model	CSL (ACC)	EXP (ACC)	SR25 (ACC)
GCN	10.00 ± 0.00	51.90 ± 1.96	6.67 ± 0.00
GatedGCN	10.00 ± 0.00	51.73 ± 1.65	6.67 ± 0.00
GINE	10.00 ± 0.00	50.69 ± 1.39	6.67 ± 0.00
GraphTrans	10.00 ± 0.00	52.35 ± 2.32	6.67 ± 0.00
GCN-MLP-Mixer	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
GatedGCN-MLP-Mixer	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
GINE-MLP-Mixer	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
GraphTrans-MLP-Mixer	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00

Table: Results are averaged over 4 runs with 4 different seeds.

[1] Murphy-etal, Relational pooling for graph representations, 2019
[2] Abboud-etal, The surprising power of graph neural networks with random node initialization, 2020
[3] Balcilar-etal, Breaking the limits of message passing graph neural networks, 2021

Lab 5 : Hierarchical GT

- GT with hierarchical mechanism

Lecture : Graph Transformer

Lab 05 : Hierarchical Graph Transformer vs. Graph Transformer

Xavier Bresson

[1]:
```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/CS5284_2026_codes/07_Graph_Transformers'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
```

This notebook compares three graph models:

1. **GCN (topology only)**
2. **Graph Transformer + LapPE**
3. **Hierarchical GT + LapPE**

using **HEM-style heavy-edge grouping** for fixed-size graph coarsening.

The hierarchical SBM size is controlled by only two parameters:

```
N_HIERARCHIES = 3
N_NODES_PER_COMMUNITY = 10
```

The graph sizes are derived automatically as

$$N_\ell = q^{H-\ell},$$

where $H =$ `N_HIERARCHIES` and $q =$ `N_NODES_PER_COMMUNITY`. With the default values, this gives

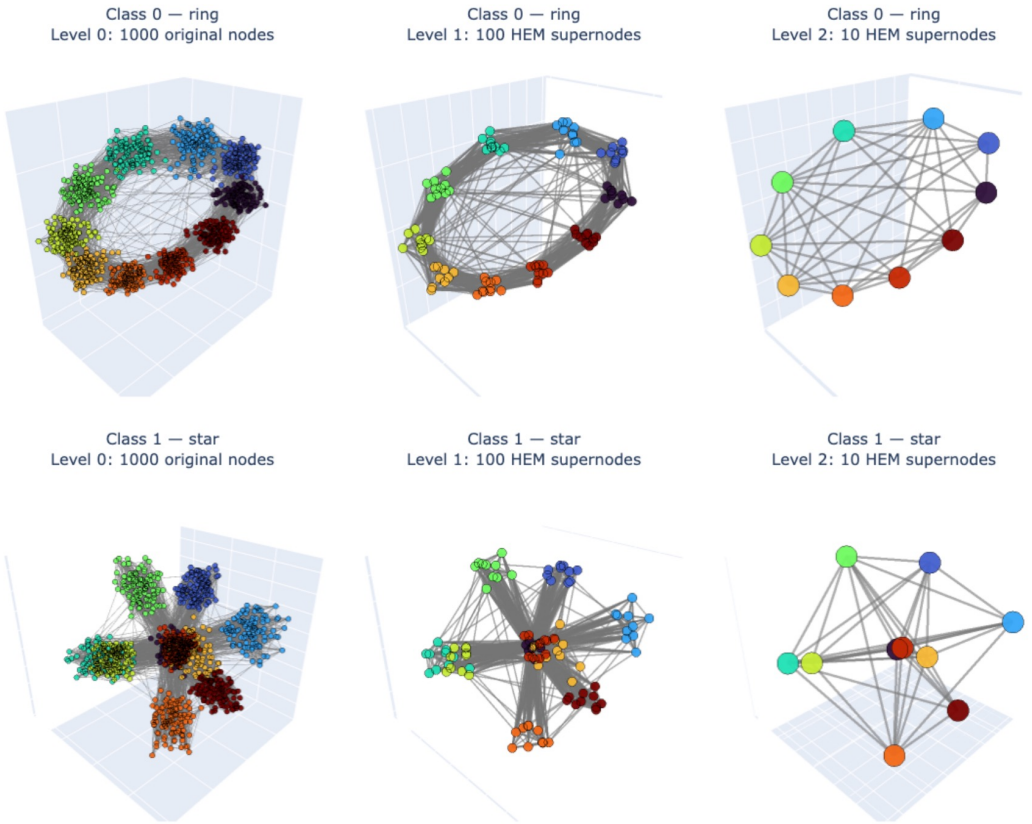
$$10^3 \to 10^2 \to 10^1 = 1000 \to 100 \to 10.$$

For a 10,000-node original graph and

```
N_HIERARCHIES = 4
N_NODES_PER_COMMUNITY = 10
```

the notebook would provide $10000 \to 1000 \to 100 \to 10$.

The number of classes stays fixed at 4. Their labels are determined by four different connectivity templates among the final coarse communities : ring, star, two dense modules with a bridge and complete bipartite.



[21]:

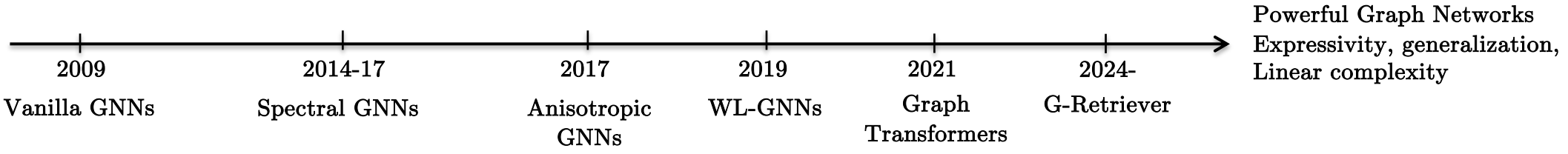
	model	test_accuracy	trainable_parameters	training_time_seconds	training_time_minutes	speed_vs_flat_GT
0	GCN (topology only)	0.5	748	40.853216	0.680887	25.408960
1	Graph Transformer + LapPE	1.0	57700	1038.037738	17.300629	1.000000
2	Hierarchical GT + LapPE	1.0	57700	204.489802	3.408163	5.076232

Outline

- Review of graph network architectures
- Graph positional encoding
- Message-passing and over-squashing
- Graph Transformer
- Graph ViT / Hierarchical Transformer
- **Conclusion**

Conclusion

- Evolution of graph neural architectures
  - Vanilla GNN ⇒ Spectral GNN ⇒ Anisotropic GNN ⇒ Expressive/WL-GNN ⇒ Graph Transformer ⇒ Graph ViT / Hierarchical GT ⇒ ?
- What comes next ?
  - Hardware-efficient graph computation : Combine scalable sparse / near-linear complexity with efficient execution on modern GPU accelerators.
  - Richer semantic representations : LLMs can augment node and edge features with information extracted from text and external knowledge.
  - Task-relevant context retrieval : Graph retrieval / GraphRAG[1] can select relevant subgraphs instead of processing the entire graph.
- Next, we will introduce G-Retriever[1] that combines GNN + GraphRAG + LLM.



[1] He-LeCun-Bresson-etal, G-retriever: Retrieval-augmented generation for textual graph understanding and question answering, 2024



Questions?