# Benchmarking Graph Transformers Toward Scalability for Large Graphs

by

Katherine S. Lim

S.B. Computer Science and Molecular Biology, Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN COMPUTER SCIENCE AND MOLECULAR BIOLOGY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024



| | |
|---|---|
| Authored by: | Katherine S. Lim<br>Department of Electrical Engineering and Computer Science<br>May 10, 2024 |
| Certified by: | Arvind<br>Johnson Professor of Computer Science and Engineering, Thesis Supervisor |
| Accepted by: | Katrina LaCurts<br>Chair, Master of Engineering Thesis Committee |

# Benchmarking Graph Transformers Toward Scalability for Large Graphs

by

Katherine S. Lim

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN COMPUTER SCIENCE AND MOLECULAR BIOLOGY

## ABSTRACT

Graph transformers (GTs) have gained popularity as an alternative to graph neural networks (GNNs) for deep learning on graph-structured data. In particular, the self-attention mechanism of GTs mitigates the fundamental limitations of over-squashing, over-smoothing, and limited expressiveness that GNNs face. Furthermore, like transformers used for natural language processing and computer vision, GTs have the potential to become foundation models that can be used for various downstream tasks. However, current GTs do not scale well to large graphs, due to computational cost. Here, we formulated a GT architecture as part of a larger scheme to build a GT made scalable through hierarchical attention and graph coarsening. Specifically, our goal was to optimize the GT building block of the scalable GT. By adding GraphGPS-inspired message-passing neural network (MPNN) layers to a modified version of the Spectral Attention Network (SAN) and performing hyperparameter tuning, we built a GT architecture that performs comparably to GraphGPS on the node classification task on the Cora and CiteSeer datasets. Compared to the modified version of SAN that we started with, our architecture is faster to train and evaluate, and also obtains higher node classification accuracies on the Cora and CiteSeer datasets. Our results demonstrate how message passing can effectively complement self-attention in GTs such as SAN to improve node classification performance. With further architectural improvement, we expect our model to serve as an effective building block for scalable GTs. Such scalable GTs may be used for node classification on large graphs, a common task for industrial applications.

Thesis supervisor: Arvind
Title: Johnson Professor of Computer Science and Engineering

# Acknowledgments

I would like to thank my MEng thesis supervisor Arvind, my project supervisor Jie Chen, and my initial point of contact Xuhao Chen for their mentorship and for providing me the opportunity to work on this project. I'd also like to thank other members of the graph transformers project group, Tim Kaler, Ryan Deng, and Kelly Ho, for their helpful suggestions.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Many real-world systems, including user-item interaction networks [1–3], Bitcoin transaction networks [4], and molecules [3, 5], can be represented as graphs. Machine learning, in particular deep learning, is a promising way of modeling and analyzing such systems [3]. Most deep learning models for graph-structured data are graph neural networks (GNNs) [6], which update node states at each time step based on aggregated messages from neighboring nodes [7, 8] (Figure 1.1). However, due to their sparse message passing mechanism, GNNs suffer fundamental limitations [9] of over-squashing [8, 10], over-smoothing [11, 12], and expressiveness bounds [13, 14]. Over-squashing occurs when a GNN has many layers and thus encounters graph bottlenecks, where the information gathered from a node's receptive field—which grows exponentially in number of nodes as the number of layers in the GNN increases—is compressed into a fixed-length vector and is thereby distorted [8, 10] (Figure 1.2a). In GNNs, the use of many layers is a prerequisite to modeling long-range interactions in a computational graph—to capture a long-range interaction between a node and its $\ell$-hop neighbor, $\ell$ GNN layers are required [12, 15]. Thus, over-squashing causes GNN model performance to be poor when the prediction task depends on capturing long-range patterns in the data [8]. Next, over-smoothing refers to the phenomenon where, on heterophilic graphs, which produce a low information-to-noise ratio during message passing, node features become

increasingly similar as the number of layers in the GNN increases [11, 12, 16] (Figure 1.2b). Specifically, node features become smoothed to the point where representations of nodes from different classes can no longer be distinguished [11, 12]. Such over-smoothing substantially decreases GNN performance on node classification tasks [11]. Lastly, GNNs have limited expressiveness—as an upper bound, GNNs cannot be more powerful than the 1-dimensional Weisfeiler-Leman graph isomorphism heuristic (1-WL) in distinguishing graph structures [13, 14]. Not only does the 1-WL fail to distinguish some graphs [17], many popular GNN variants are less powerful than the 1-WL [14]. For instance, GCN [18] and GraphSAGE [19] are less powerful than the 1-WL and fail to distinguish a few simple graph structures (Figure 1.2c), due to the way the models perform neighbor aggregation [14]. Such limited expressiveness decreases GNN performance on graph classification tasks that require the model to learn from the network structure rather than rely on input node features [14].

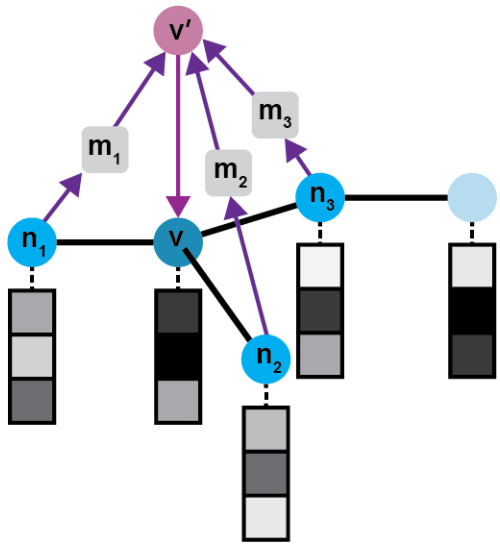


Figure 1.1: Illustration of GNN message passing mechanism. Blue circles are graph nodes; squares connected to the graph nodes via dotted lines are node features; gray rounded boxes (labeled with 'm') are messages; pink circle is the updated node. Adapted from Wu et al. [20].

Transformers [22], which have been successful in natural language processing (NLP) [23–25] and computer vision (CV) [26, 27], have gained popularity as an alternative to GNNs for deep learning on graph-structured data [9]. Excitingly, graph transformers (GTs) have the potential to become foundation models [28], i.e., pretrained models that can be used

for a variety of downstream tasks, similar to transformer foundation models used for NLP and CV [23, 24, 29]. By definition, GTs rely on a self-attention mechanism that allows all nodes to attend to all other nodes in the graph, regardless of the graph's connectivity [9, 17] (Figure 1.3). This self-attention mechanism mitigates the aforementioned issues (*cf.* Figure 1.2) that GNNs suffer from [9]. In particular, GTs, unlike GNNs, are capable of modeling long-range interactions in a graph since all nodes are effectively connected to each other [15].

However, a major concern with GTs is that they do not scale well to large graphs, due to computational cost [9]. Specifically, due to GTs' self-attention computation, the computational complexity of GTs is quadratic on the order of $O(N^2)$ on a graph with $N$ nodes and $E$ edges, in contrast to the linear $O(E)$ complexity of GNNs [9]. As such, GT performance has mainly been demonstrated on small graphs, or for graph classification problems only [31]. In particular, there lacks a systematic study of GTs for the practical task of node classification on large graphs [31].

Here, we formulated a GT architecture as part of a larger scheme to build a GT made scalable through a hierarchical attention and graph coarsening approach (Figure 1.4). In this approach, the graph is subdivided into clusters/local neighborhoods, and each cluster's nodes are passed into a multi-head attention module that runs multiple attention mechanisms in parallel. The attention outputs are then passed to another multi-head attention module. This approach allows for the reduction of computational complexity while maintaining the ability to capture long-range interactions in the graph. My project specifically focused on optimizing the architecture of each graph-transformer-layer (GTL) block (Figure 1.4).

Our model builds upon the lab's modified version of the Spectral Attention Network (SAN) [17] by adding GraphGPS [9]-inspired message-passing neural network (MPNN) layers and by optimizing hyperparameters. Our model results in faster runtimes and higher node classification accuracies than the lab's modified version of SAN on the Cora [32] and CiteSeer [33] datasets. Compared to GraphGPS, our model achieves higher node classifica-

tion accuracies on the Cora dataset and similar accuracies on the CiteSeer dataset.

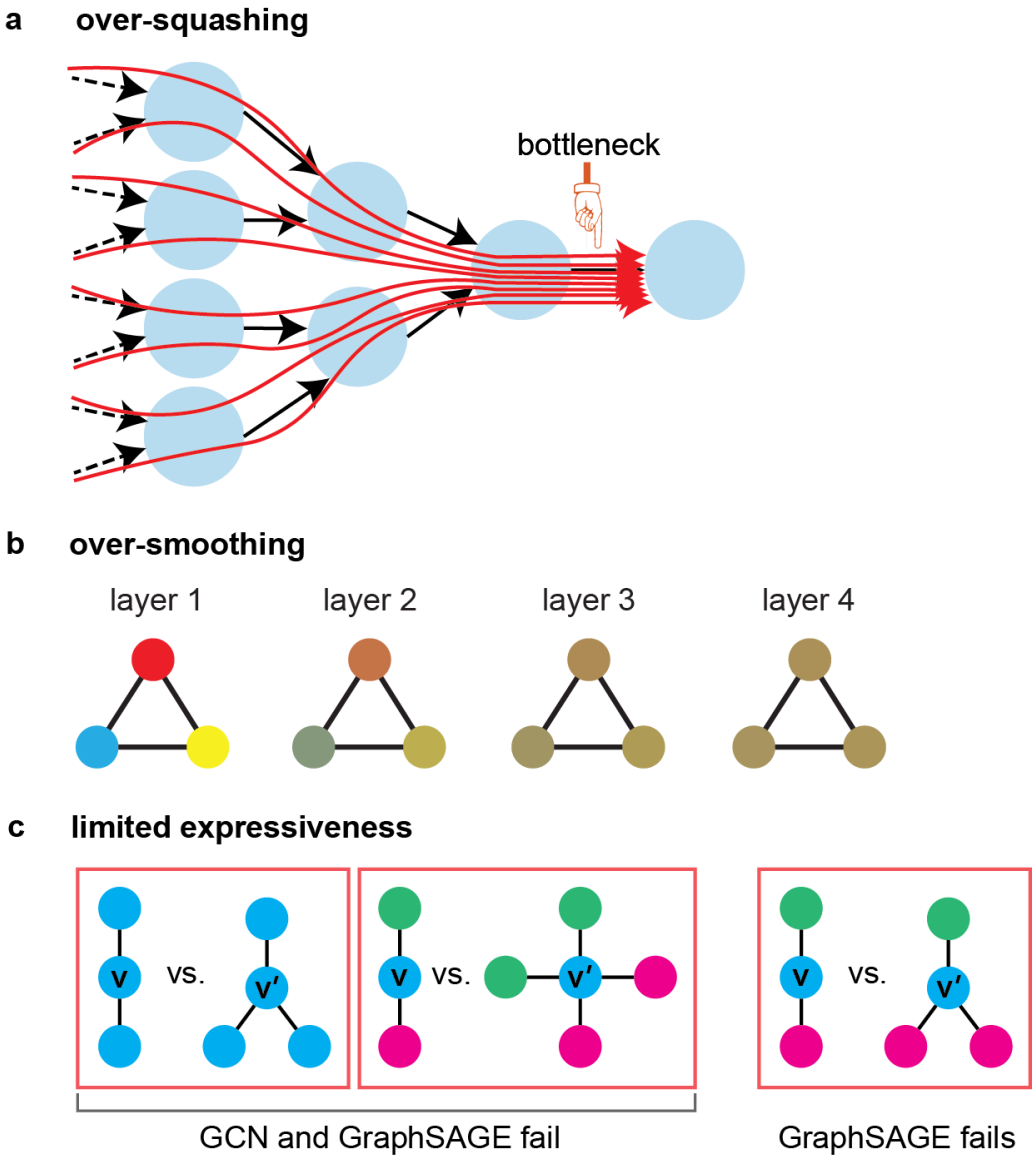


Figure 1.2: Fundamental limitations of GNNs. Colored circles are graph nodes. (a) Illustration of the graph bottleneck that causes over-squashing in GNNs. Black arrows are graph edges; red curved arrows represent information flow. Adapted from Alon and Yahav [8]. (b) Illustration of the over-smoothing problem in GNNs. Nodes are color coded by node feature. Adapted from Balla [21]. (c) Examples of graph structures that GCN or GraphSAGE fail to distinguish. Nodes are color coded by node feature. Adapted from Xu et al. [14].

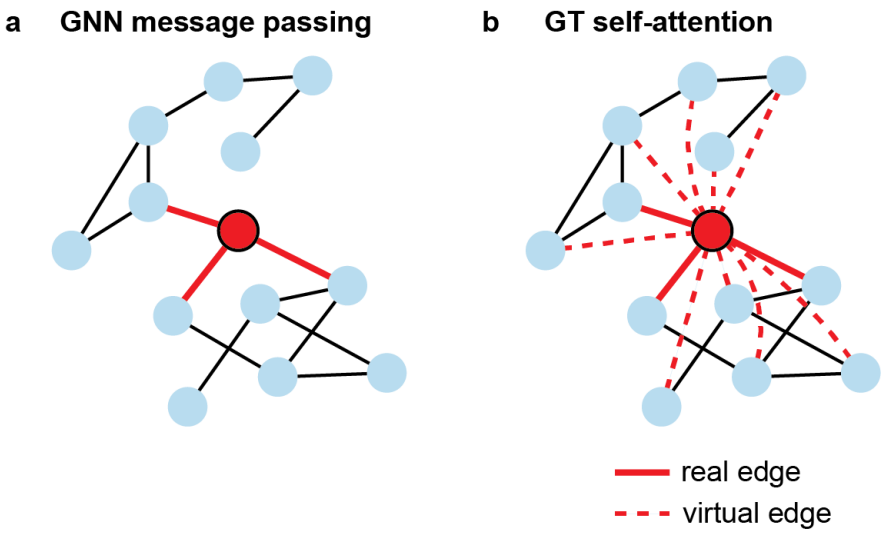


Figure 1.3: Illustration of GT self-attention, contrasted with GNN message passing. Colored circles are graph nodes. Real- and virtual-edge drawing style adapted from Galkin [30].

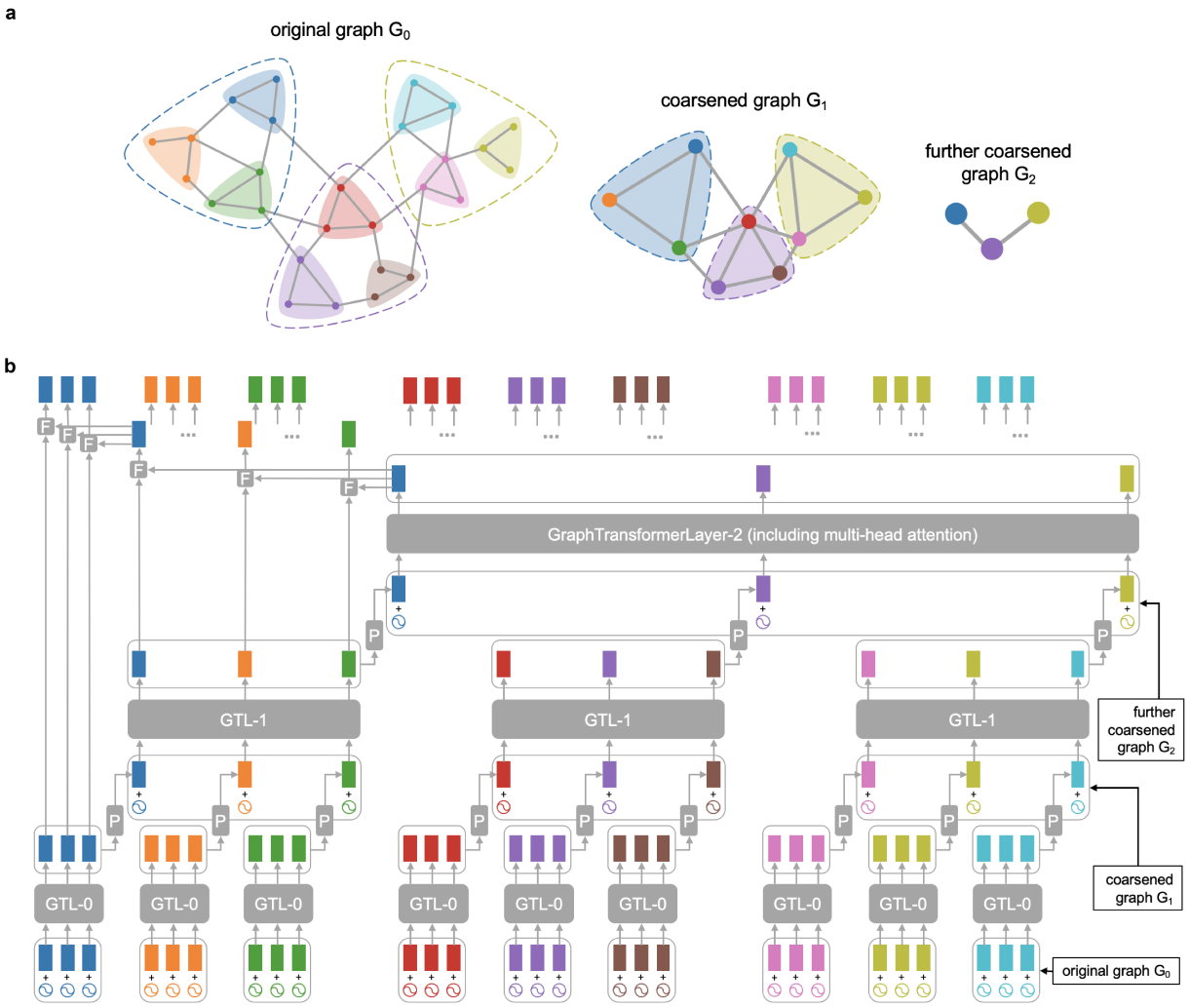


Figure 1.4: Hierarchical attention and graph coarsening approach. Node clusters are color coded. (a) Illustration of graph coarsening with an example graph. (b) Diagram of scalable GT architecture implementing hierarchical attention and graph coarsening on the example graph. Each horizontal gray box represents a GT layer (my project focused on formulating this layer).

# Chapter 2

# Related Work

## 2.1 Graph Neural Networks

### 2.1.1 Graph Convolutional Network

Graph Convolutional Network (GCN) is a semi-supervised GNN that generalizes the 1-WL with motivation from a first-order approximation of spectral graph convolutions [18] (Equation 2.1).

$$h_i^{(\ell+1)} = \sigma\left(\sum_{j \in N_i} \frac{1}{c_{ij}} h_j^{(\ell)} W^{(\ell)}\right) \tag{2.1}$$

where

$h_i^{(\ell)}$ = vector of activations of node $i$ in layer $\ell$

$N_i$ = node $i$'s set of neighboring node indices

$c_{ij}$ = $\sqrt{|N_i||N_j|}$; normalization constant for the edge $(i, j)$

$W^{(\ell)}$ = weight matrix for layer $\ell$

$\sigma(\cdot)$ = ReLU activation function

With this layer-wise propagation rule, GCN shows an ability to encode both local graph structure and node features for node classification tasks [18].

### 2.1.2 Graph Convolutional Network via Initial Residual and Identity Mapping

Graph Convolutional Network via Initial Residual and Identity Mapping (GCNII) builds upon GCN by adding (i) a skip connection between each layer and the input layer and (ii) an identity matrix to each layer's weight matrix [34] (Equation 2.2). These modifications to GCN are implemented specifically to mitigate over-smoothing [34]. In contrast to GCN, where a $k$-layer model corresponds to a $k^{\text{th}}$-order polynomial filter with *fixed* coefficients, a GCNII model with $k$ layers can express a $k^{\text{th}}$-order polynomial filter with *arbitrary* coefficients [34].

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\left((1-\alpha_\ell)\tilde{\mathbf{P}}\mathbf{H}^{(\ell)} + \alpha_\ell\mathbf{H}^{(0)}\right)\left((1-\beta_\ell)\mathbf{I}_n + \beta_\ell\mathbf{W}^{(\ell)}\right)\right) \tag{2.2}$$

where

| | |
|---|---|
| $\mathbf{H}^{(\ell+1)}$ | $= \ell^{\text{th}}$ layer of GCNII |
| $\tilde{\mathbf{P}} = \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$ | = graph convolution matrix with renormalization (as in Kipf and Welling [18]) |
| $G = (V, E)$ | = simple, connected, undirected graph |
| $\mathbf{A}$ | = adjacency matrix of $G$ |
| $\mathbf{D}$ | = diagonal degree matrix of $G$ |
| $\tilde{G} = (V, \tilde{E})$ | = graph $G$ with a self-loop added to each node |
| $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ | = adjacency matrix of $\tilde{G}$ |
| $\tilde{\mathbf{D}} = \mathbf{D} + \mathbf{I}$ | = diagonal degree matrix of $\tilde{G}$ |
| $\mathbf{I}_n$ | = identity matrix |
| $\mathbf{W}^{(\ell)}$ | $= \ell^{\text{th}}$ weight matrix |
| $\alpha_\ell, \beta_\ell$ | = hyperparameters |

### 2.1.3 Graph Attention Network

Graph Attention Network (GAT) is a GNN consisting of stacked layers in which nodes attend to their respective neighborhoods in a way that is parallelizable [35]. Multiple message-passing mechanisms are run, and the output features of those mechanisms are concatenated [35] (Figure 2.1; Equation 2.3). On the model's final prediction layer, the outputs of the message-passing mechanisms are averaged before being passed to a nonlinear function [35] (Figure 2.1; Equation 2.4).

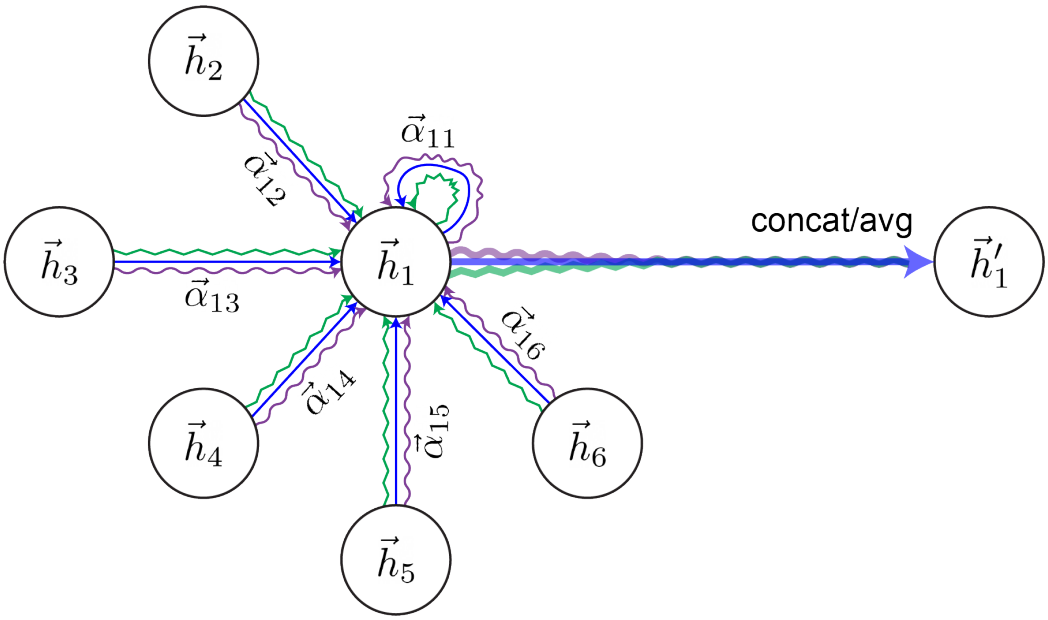


Figure 2.1: Illustration of GAT's message-passing mechanism for an example node 1 on its first-order neighborhood (nodes 1-6). Different arrow colors/styles represent independent message-passing computations (K=3 here, as an example). Adapted from Veličković et al. [35].

$$\vec{h}'_i = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j \in N_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right) \tag{2.3}$$

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in N_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right) \tag{2.4}$$

where

$K$ = number of independent message-passing mechanisms

$\|$ = concatenation

$$\alpha_{ij}^{k} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{a}^{T}\left[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j\right]\right)\right)}{\sum_{\ell \in N_i} \exp\left(\text{LeakyReLU}\left(\vec{a}^{T}\left[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_\ell\right]\right)\right)}$$

= normalized importance of node $j$'s features to node $i$

$F$ = number of input node features

$F'$ = number of output node features

$\vec{a} \in \mathbb{R}^{2F'}$ = weight vector

$.^{T}$ = transposition

$\mathbf{W} \in \mathbb{R}^{F' \times F}$ = weight matrix

$\vec{h}_i \in \mathbb{R}^{F}$ = input node feature vector for node $i$

$\vec{h}'_i \in \mathbb{R}^{F'}$ = output node feature vector for node $i$

$N_i$ = the first-order neighbors of node $i$, including $i$

$\sigma(\cdot)$ = nonlinear function

GAT is competitive with GCN in terms of computational efficiency, and furthermore improves upon GCN by enabling the assignment of different importances to different nodes in a neighborhood [35]. GAT works on both directed and undirected graphs, and is applicable to both transductive and inductive learning [35].

### 2.1.4 Spline-Based Convolutional Neural Network

Spline-Based Convolutional Neural Network (SplineCNN) uses a B-spline-based spatial, continuous, trainable convolution kernel to weight the aggregation of local node features within each model layer [36] (Equation 2.5). The kernel is used in a convolution operator that serves as a module in the SplineCNN architecture [36] (Equation 2.6). The convolution operator generalizes the traditional convolutional-neural-network (CNN) convolutional layer with odd filter sizes, excepting a normalization factor [36]. To get a convolutional layer that generates $M_{\text{out}}$ output feature maps, the convolution operator is applied $M_{\text{out}}$ times to the input data,

each time using different trainable parameters [36]. For each (input feature map, output feature map) pair, a different set of weights is trained [36].

$$\text{convolution kernel: functions } g_\ell : [a_1, b_1] \times \cdots \times [a_d, b_d] \to \mathbb{R} \tag{2.5}$$

$$\text{with } g_\ell(\mathbf{u}) = \sum_{\mathbf{p} \in \mathcal{P}} w_{\mathbf{p},\ell} \cdot B_{\mathbf{p}}(\mathbf{u})$$

where

| | |
|---|---|
| $N$ | = number of nodes |
| $\mathcal{V}$ | = set of nodes |
| $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ | = set of edges |
| $d$ | = dimensionality of the pseudo-coordinates for each directed edge |
| $\mathbf{U} \in [0,1]^{N \times N \times d}$ | = matrix of pseudo-coordinates $\mathbf{u}$ for each directed edge |
| $\mathbf{u}(i,j) \in [0,1]^d$ | = $d$-dimensional pseudo-coordinates for the directed edge $(i,j) \in \mathcal{E}$ |
| $m$ | = differing B-spline basis degree |
| $\mathbf{k} = (k_1, ..., k_d)$ | = $d$-dimensional kernel size |
| $(N_{1,i}^m)_{1 \le i \le k_1}, \cdots, (N_{d,i}^m)_{1 \le i \le k_d}$ | = $d$ open B-spline bases of degree $m$ based on equidistant knot vectors |
| $\mathcal{P} = (N_{1,i}^m)_i \times \cdots \times (N_{d,i}^m)_i$ | = Cartesian product of the B-spline bases |
| $\mathbf{p}$ | = element from $\mathcal{P}$ |
| $w_{\mathbf{p},\ell}$ | = trainable parameter for $\mathbf{p}$ |
| $B_{\mathbf{p}}(\mathbf{u})$ | $= \prod_{i=1}^{d} N_{i,p_i}^m(u_i)$ = product of the basis functions in $\mathbf{p}$ |
| $\ell$ | = index for each of the $M_{\text{in}}$ input feature maps |

$$\text{convolution operator for a node } i: \tag{2.6}$$

$$(\mathbf{f} \star \mathbf{g})(i) = \frac{1}{|\mathcal{N}(i)|} \sum_{\ell=1}^{M_{\text{in}}} \sum_{j \in \mathcal{N}(i)} f_\ell(j) \cdot g_\ell(\mathbf{u}(i,j))$$

where

| | |
|---|---|
| $\mathcal{V}$ | = set of nodes |
| $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ | = set of edges |
| $M_{\text{in}}$ | = number of input features per node |
| $\mathbf{f}(i) \in \mathbb{R}^{M_{\text{in}}}$ | = vector of $M_{\text{in}}$ input features for node $i$ |
| $\ell$ | = index for each of the $M_{\text{in}}$ input feature maps |
| $\{f_\ell(i) \mid i \in \mathcal{V}\}$ | = input feature map |
| $\mathbf{g} = (g_1, ..., g_{M_{\text{in}}})$ | = kernel functions |
| $\mathcal{N}(i)$ | = node $i$'s neighborhood set |
| $\mathbf{u}(i,j) \in [0,1]^d$ | = $d$-dimensional pseudo-coordinates for the directed edge $(i,j) \in \mathcal{E}$ |

### 2.1.5 GraphSAGE

GraphSAGE learns embedding functions that sample and aggregate local node features to produce node embeddings for inductive learning [19] (Figure 2.2). Such incorporation of node features in the model training process allows the model to learn both the topological structure and the node feature distribution within each node's neighborhood [19]. Additionally, the use of an unsupervised loss function in GraphSAGE allows for model training without task-specific supervision [19].

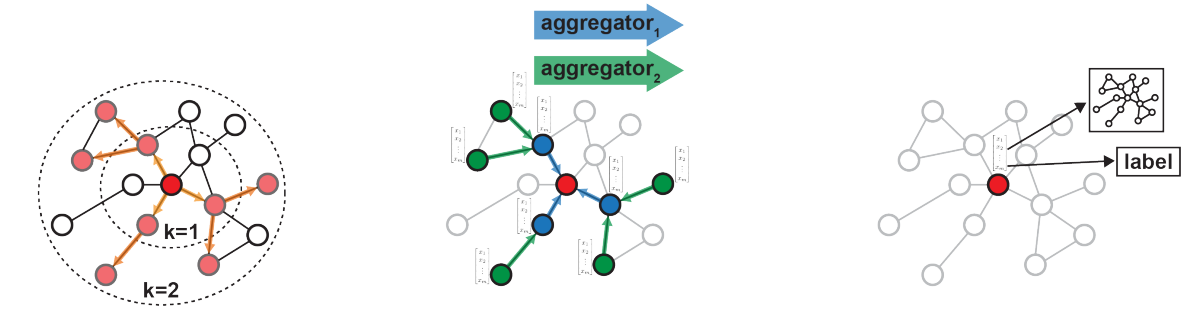


Figure 2.2: Illustration of GraphSAGE's local node feature sampling and aggregation approach. Adapted from Hamilton, Ying, and Leskovec [19].

## 2.2 Graph Transformers

Many recent studies have explored various GT architectures, e.g., using novel structural/ positional encodings, node sampling methods, or attention mechanisms.

### 2.2.1 Structural/Positional Encodings

#### Spectral Attention Network

The Spectral Attention Network (SAN) incorporates a learned positional encoding (LPE) that can make use of the full Laplacian spectrum of the graph to determine node positions [17] (Figure 2.3). This LPE is added to the node features, which are then passed to a fully-connected transformer, where multi-head attention (Equations 2.7, 2.8, 2.9) is performed [17]. In theory, the LPE increases the model's ability to distinguish graphs and detect similar substructures from their resonance [17].

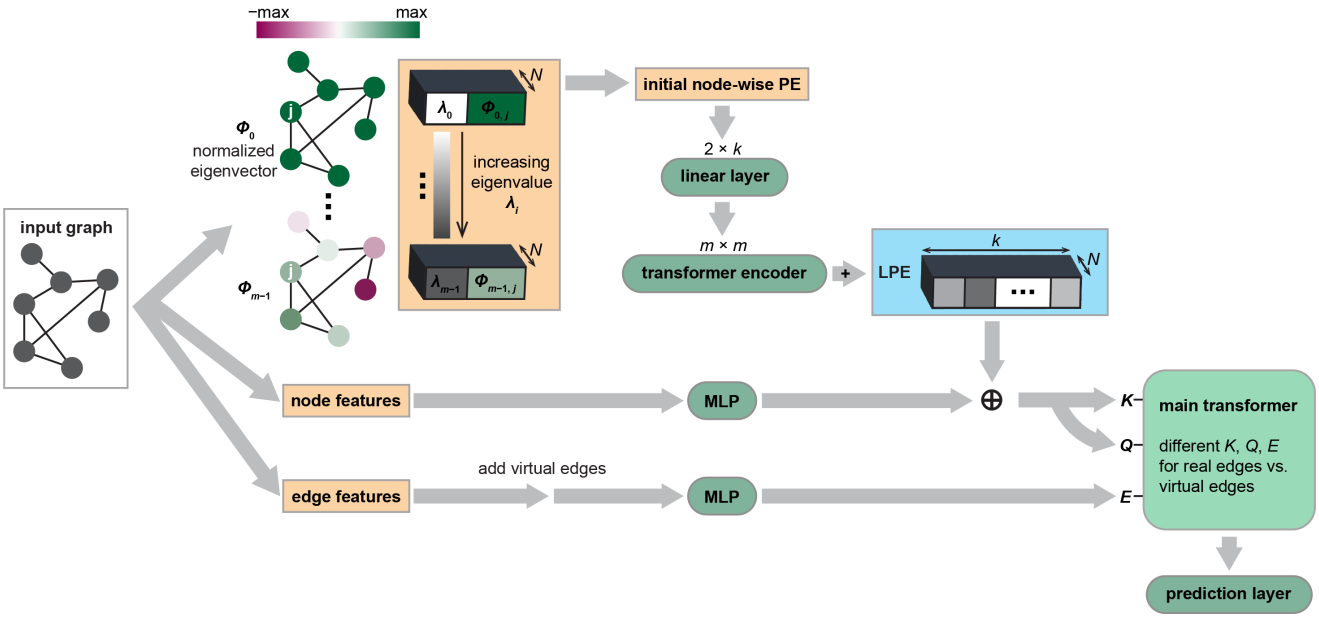


Figure 2.3: Diagram of SAN's architecture. PE: positional encoding; LPE: learned positional encoding; MLP: multilayer perceptron. Adapted from Kreuzer et al. [17].

$$\hat{h}_i^{\ell+1} = O_h^{\ell} \Big\Vert_{k=1}^{H} \left( \sum_{j \in V} w_{ij}^{k,\ell} V^{k,\ell} h_j^{\ell} \right) \tag{2.7}$$

where $O_h^{\ell} \in \mathbb{R}^{d \times d}$, $V^{k,\ell} \in \mathbb{R}^{d_k \times d}$,

$h_i^{\ell}$ = node $i$'s features at layer $\ell$

$\Vert$ = concatenation

$H$ = number of heads

$w_{ij}^{k,\ell}$ = attention weights (Equations 2.8 and 2.9)

$d$ = hidden dimension

$d_k$ = $\frac{d}{H}$ = dimension of a head

$$\hat{w}_{ij}^{k,\ell} = \begin{cases} \frac{Q^{1,k,\ell} h_i^{\ell} \circ K^{1,k,\ell} h_j^{\ell} \circ E^{1,k,\ell} e_{ij}}{\sqrt{d_k}} & \text{if } i \text{ and } j \text{ are connected in sparse graph} \\ \frac{Q^{2,k,\ell} h_i^{\ell} \circ K^{2,k,\ell} h_j^{\ell} \circ E^{2,k,\ell} e_{ij}}{\sqrt{d_k}} & \text{otherwise} \end{cases} \tag{2.8}$$

$$w_{ij}^{k,\ell} = \begin{cases} \frac{1}{1+\gamma} \cdot \text{softmax}\left( \sum_{d_k} \hat{w}_{ij}^{k,\ell} \right) & \text{if } i \text{ and } j \text{ are connected in sparse graph} \\ \frac{\gamma}{1+\gamma} \cdot \text{softmax}\left( \sum_{d_k} \hat{w}_{ij}^{k,\ell} \right) & \text{otherwise} \end{cases} \tag{2.9}$$

where $Q^{1,k,\ell}, Q^{2,k,\ell}, K^{1,k,\ell}, K^{2,k,\ell}, E^{1,k,\ell}, E^{2,k,\ell} \in \mathbb{R}^{d_k \times d}$, $\gamma \in \mathbb{R}^{+}$,

$Q^1, K^1, E^1$ = query, key, and edge projections for pairs of connected nodes

$Q^2, K^2, E^2$ = query, key, and edge projections for pairs of disconnected nodes

$h_i^{\ell}$ = node $i$'s features at layer $\ell$

$e_{ij}$ = edge feature embedding between nodes $i$ and $j$

$d$ = hidden dimension

$d_k$ = $\frac{d}{\text{number of heads}} = \frac{d}{H}$ = dimension of a head

$\circ$ = element-wise multiplication

$\gamma$ = hyperparameter that tunes the bias toward full-graph attention

For numerical stability, softmax outputs are clipped to the range $[-5, 5]$ [17].

### GraphGPS

GraphGPS incorporates positional encodings (PE), structural encodings (SE), and message-passing neural network (MPNN)/transformer hybrid layers [9] (Figure 2.4). PE and SE, which help GTs learn the structure of the input graph, have not only been shown to enhance GT performance, they also increase the expressive power of MPNNs [9]. At each "GPS" layer, node and edge features are updated using the sum of the outputs of a MPNN layer and a transformer layer [9] (Equation 2.10). By computing PE and SE only for real nodes and edges and excluding edge features from the transformer layers, GraphGPS achieves a linear computational complexity of $O(N + E)$ on a graph with $N$ nodes and $E$ edges [9]. As such, GraphGPS successfully scales to graphs of several thousand nodes [9].

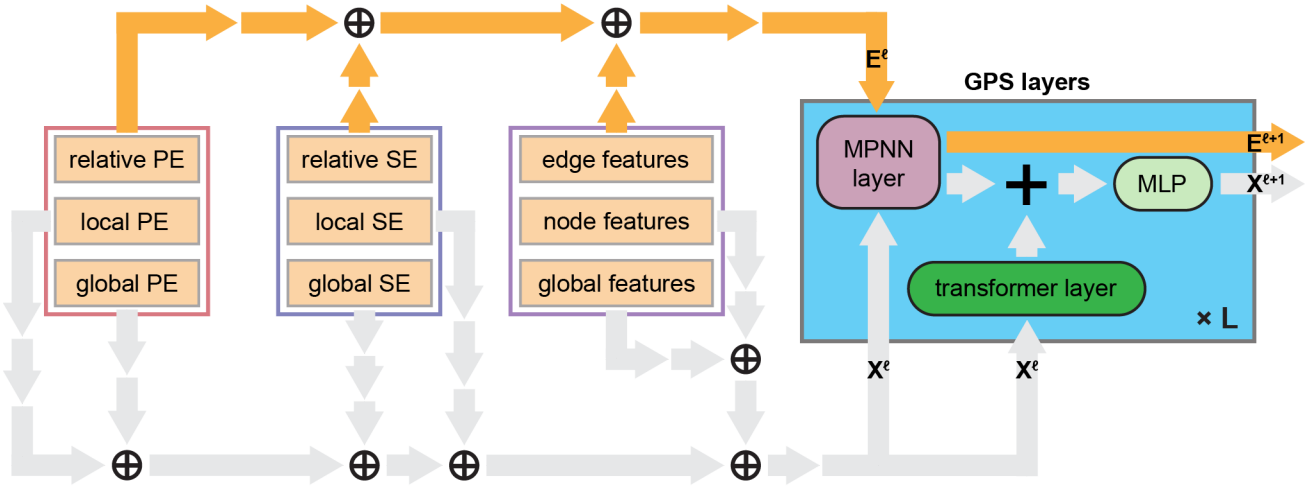


Figure 2.4: Diagram of GraphGPS's architecture. PE: positional encoding; SE: structural encoding; $X^\ell$: node features at layer $\ell$; $E^\ell$: edge features at layer $\ell$; MPNN: message-passing neural network; MLP: multilayer perceptron. Adapted from Rampášek et al. [9].

$$
\begin{aligned}
X^{\ell+1}, E^{\ell+1} &= \texttt{GPS}^{\ell}\left(X^{\ell}, E^{\ell}, A\right) \\
\text{computed as} \qquad X_M^{\ell+1}, E^{\ell+1} &= \texttt{MPNN}_e^{\ell}\left(X^{\ell}, E^{\ell}, A\right), \\
X_T^{\ell+1} &= \texttt{GlobalAttn}^{\ell}\left(X^{\ell}\right), \\
X^{\ell+1} &= \texttt{MLP}^{\ell}\left(X_M^{\ell+1} + X_T^{\ell+1}\right)
\end{aligned}
\tag{2.10}
$$

where $X^{\ell} \in \mathbb{R}^{N \times d_\ell}$, $E^{\ell} \in \mathbb{R}^{E \times d_\ell}$, $A \in \mathbb{R}^{N \times N}$,

| | |
|---|---|
| $N$ | = number of graph nodes |
| $E$ | = number of graph edges |
| $X^{\ell}$ | = $d_\ell$-dimensional node features |
| $E^{\ell}$ | = $d_\ell$-dimensional edge features |
| $A$ | = graph adjacency matrix |
| $\texttt{MPNN}_e^{\ell}$ | = MPNN with edge features at layer $\ell$ |
| $\texttt{GlobalAttn}^{\ell}$ | = global attention mechanism at layer $\ell$ |
| $\texttt{MLP}^{\ell}$ | = 2-layer multilayer perceptron (MLP) block |

### Graph Transformer

Inspired by recent architectures for computer vision, Graph Transformer (GraphTrans) learns short-range and long-range interactions in the graph via a GNN module followed by a transformer module [37] (Figure 2.5). A "<CLS>" readout token is used to aggregate the pairwise interactions learned by the transformer module into an embedding that is then converted to an output classification vector [37]. The transformer module does not use any positional encodings, and is thus permutation-invariant [37].

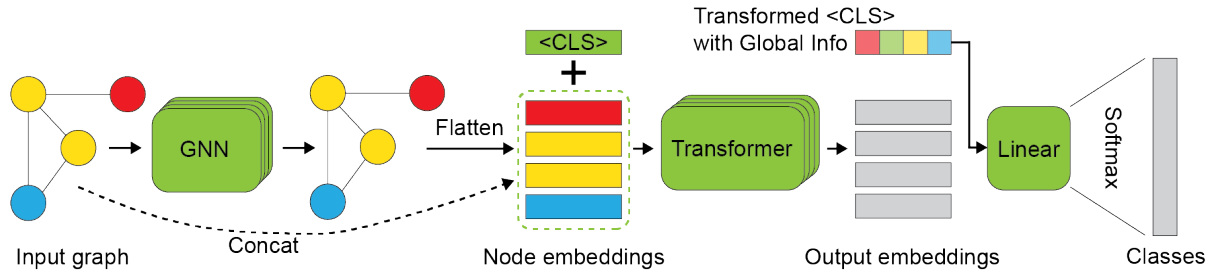


Figure 2.5: Diagram of GraphTrans's architecture. GNN: graph neural network; <CLS>: classification readout token. Adapted from Wu et al. [37].

### Structure-Aware Transformer

In order to produce more expressive node representations, the Structure-Aware Transformer (SAT) extracts node-rooted subtree or subgraph representations and uses them to help calculate the self-attention [38] (Figure 2.6; Equation 2.11). To get a representation of the $k$-subtree rooted at a node $u$, the $k$-subtree structure extractor takes the final node representation of $u$ after applying a GNN to the input graph [38] (Equation 2.12). This subtree computation is fast and can utilize any existing GNN [38]. The $k$-subgraph extractor, on the other hand, extracts the entire $k$-hop subgraph rooted at a node $u$ by aggregating the final node representations of all the nodes in $u$'s $k$-hop neighborhood after applying a GNN to the input graph [38] (Equation 2.13). This $k$-subgraph extractor allows for more expressive node representations than the $k$-subtree extractor, but does not scale as well to large graphs [38].

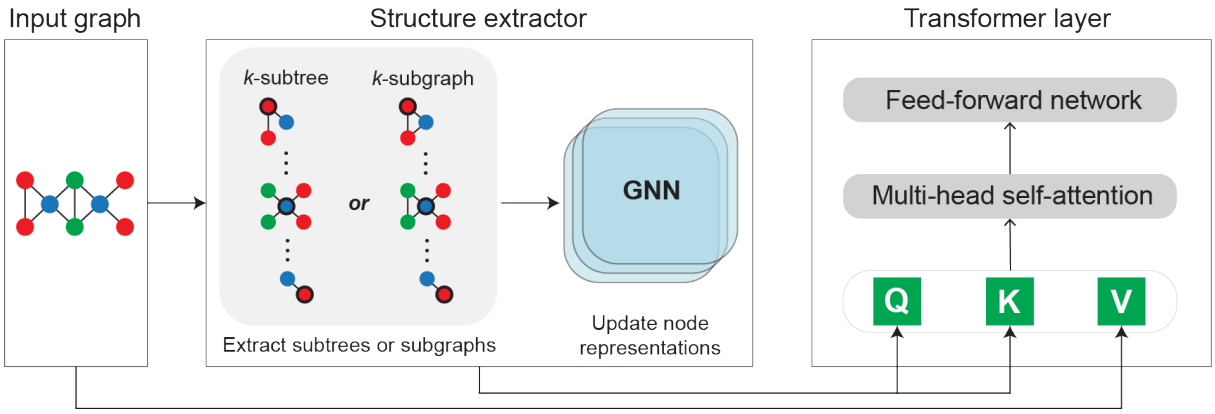


Figure 2.6: Diagram of SAT's structure extractor and its contribution to the multi-head attention calculation. GNN: graph neural network; Q: query matrix; K: key matrix; V: value matrix. Adapted from Chen, O'Bray, and Borgwardt [38] and BorgwardtLab [39].

$$\text{SA-attn}(v) := \sum_{u \in V} \frac{\kappa_{\text{graph}}(S_G(v), S_G(u))}{\sum_{w \in V} \kappa_{\text{graph}}(S_G(v), S_G(w))} f(x_u) \tag{2.11}$$

where

| | |
|---|---|
| SA-attn | = structure-aware self-attention |
| $G$ | = input graph |
| $V$ | = set of nodes |
| $S_G(v)$ | = a subgraph in $G$, centered at node $v$ |
| $\mathbf{X}$ | = input node features |
| $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ | = query, key, and value matrices |
| $d_{out}$ | = dimension of $\mathbf{Q}$ |
| $\mathbf{W_Q}, \mathbf{W_K}, \mathbf{W_V}$ | = trainable parameters used to linearly project $\mathbf{X}$ to $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ |
| $f(x) = \mathbf{W_V} x$ | = the linear value function |
| $\varphi(u, G)$ | = structure extractor for node $u$ |
| $\kappa_{\text{graph}}(S_G(v), S_G(u))$ | $= \exp\left(\langle \mathbf{W_Q}\varphi(v, G), \mathbf{W_K}\varphi(u, G)\rangle / \sqrt{d_{out}}\right)$ |

$$k\text{-subtree GNN extractor: } \varphi(u, G) = \text{GNN}_G^{(k)}(u) \tag{2.12}$$

where

| | |
|---|---|
| $G$ | = input graph |
| $u$ | = a given node in $G$ |
| $\text{GNN}_G^{(k)}$ | = any GNN with $k$ layers, applied to $G$ |

$$k\text{-subgraph GNN extractor: } \varphi(u, G) = \sum_{v \in \mathcal{N}_k(u)} \text{GNN}_G^{(k)}(v) \tag{2.13}$$

where

| | |
|---|---|
| $G$ | = input graph |
| $u$ | = a given node in $G$ |
| $\mathcal{N}_k(u)$ | = the $k$-hop neighborhood of $u$, including $u$ |
| $\text{GNN}_G^{(k)}$ | = any GNN with $k$ layers, applied to $G$ |

#### Other Architectures

To effectively encode the graph structure into the model, Graphormer uses a node-degree-based centrality encoding, a spatial encoding, and an edge encoding that incorporates edge features and learnable embeddings [40]. The Graph Inductive Bias Transformer (GRIT) incorporates graph inductive biases through random walk positional encodings and degree scalers with batch normalization [41].

### 2.2.2 Node Sampling or Adapted Attention for Scalability

#### Neighborhood Aggregation Graph Transformer

Neighborhood Aggregation Graph Transformer (NAGphormer) uses a "Hop2Token" neighborhood aggregation module that, for each node, aggregates the features from $K$ hops to generate a sequence of $K$ tokens [42] (Figure 2.7; Equations 2.14 and 2.15). In this way, the model initially gives each node a token-sequence representation as opposed to an individual token representation [42]. These token-sequence representations are passed to a learnable linear projection, whose output is fed to a transformer encoder [42]. To further enhance model performance, an attention-based readout function (Equation 2.16) is used to learn the importance of each hop neighborhood and use the learned importances to weight the aggregation of neighborhood information across hops [42].

NAGphormer's use of token-sequence initial node representations (*cf.* Equation 2.15) allows for scale-up to large graphs [42]. Specifically, token-sequence node representations allow for mini-batch training where the GPU memory usage can be controlled by appropriately setting the batch size [42]. The space complexity of NAGphormer ends up being $O(b(K+1)^2 + b(K+1)d + d^2L)$, where $b$ is the batch size, $K$ is the number of hops, $d$ is the feature vector dimension, and $L$ is the number of layers [42]. The time complexity is $O(n(K+1)^2d)$, where $n$ is the number of nodes in the graph [42].

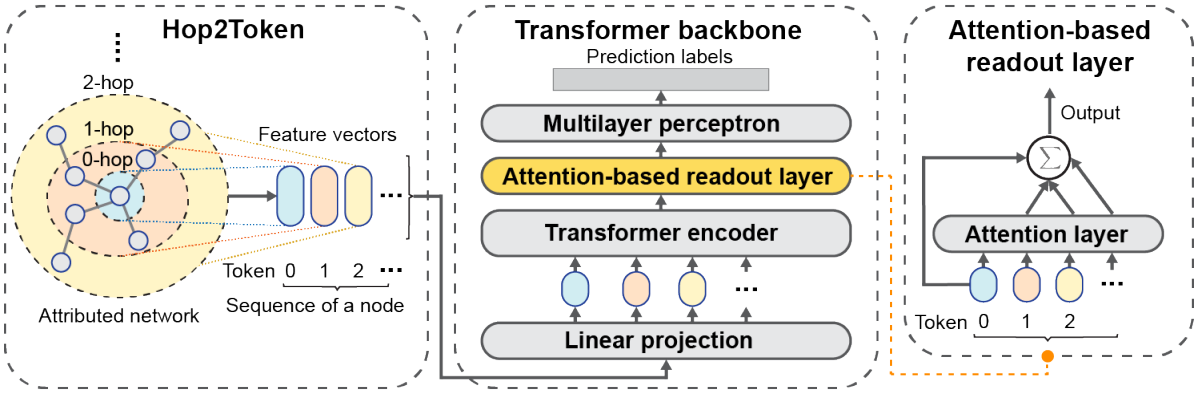


Figure 2.7: Diagram of NAGphormer's architecture. Adapted from Chen et al. [42].

$$k\text{-hop representation/token of a node } v\text{: } \mathbf{x}_v^k = \phi\left(\mathcal{N}^k\left(v\right)\right) \tag{2.14}$$

where

$V$ = set of nodes

$\mathcal{N}^k(v) = \{u \in V \mid d(v,u) \leq k\}$ = $v$'s $k$-hop neighborhood

$d(v,u)$ = length of shortest path between $v$ and $u$

$\mathcal{N}^0(v) = \{v\}$

$\phi$ = aggregation operator

$$\text{sequence of tokens for a node } v\text{: } \mathcal{S}_v = (\mathbf{x}_v^0, \mathbf{x}_v^1, ..., \mathbf{x}_v^K) \tag{2.15}$$

where

$\mathbf{x}_v^k$ = $v$'s $k$-hop representation/token (*cf.* Equation 2.14)

$K$ = hyperparameter

$$\text{attention-based readout function for a node } v\text{: } \mathbf{Z}_{out} = \mathbf{Z}_0 + \sum_{k=1}^{K} \alpha_k \mathbf{Z}_k \tag{2.16}$$

where

$\mathbf{Z}_0$ = representation/token of $v$ itself

$\mathbf{Z}_k$ = $k$-hop representation/token of $v$

$d_m$ = hidden dimension of the transformer

$\mathbf{W}_a \in \mathbb{R}^{1\times 2d_m}$ = learnable projection

$\|$ = concatenation operator

$.^T$ = transposition

$$\alpha_k = \frac{\exp\left(\left(\mathbf{Z}_0 \| \mathbf{Z}_k\right)\mathbf{W}_a^T\right)}{\sum_{i=1}^{K}\exp\left(\left(\mathbf{Z}_0 \| \mathbf{Z}_i\right)\mathbf{W}_a^T\right)}$$

= normalized attention coefficient for $v$'s $k$-hop neighborhood

### Gophormer

Gophormer reduces computational complexity, incorporates local structural information, and in theory helps prevent overfitting by inputting sampled ego-graphs, which each contain far fewer nodes on average than the full graph, to the transformer [43] (Figure 2.8; Equation 2.17). Global structural information is also incorporated by adding the same set of global nodes with learnable features to each ego-graph [43] (Equation 2.17). Within the transformer, a proximity-enhanced multi-head attention mechanism is used to capture fine-grained structural information [43] (Equation 2.18). To handle the uncertainty caused by ego-graph sampling, consistency regularization (Equation 2.19) and multi-sample inference (Equation 2.20) are used during training and testing, respectively [43]. Specifically, consistency regularization is included in the loss function to ensure similarity in model predictions across ego-graphs of the same center node [43]. In multi-sample inference, for each testing node, ego-graphs are sampled and the model's predictions on the ego-graphs are averaged to obtain the final prediction for the node [43].

$$\text{epoch-wise transformer input for a center node } v_c\text{: } \mathbf{H}_{c,s}^0 = \text{Concat}\left(\mathbf{X}_c^{(s)}, \mathbf{C}\right) \tag{2.17}$$

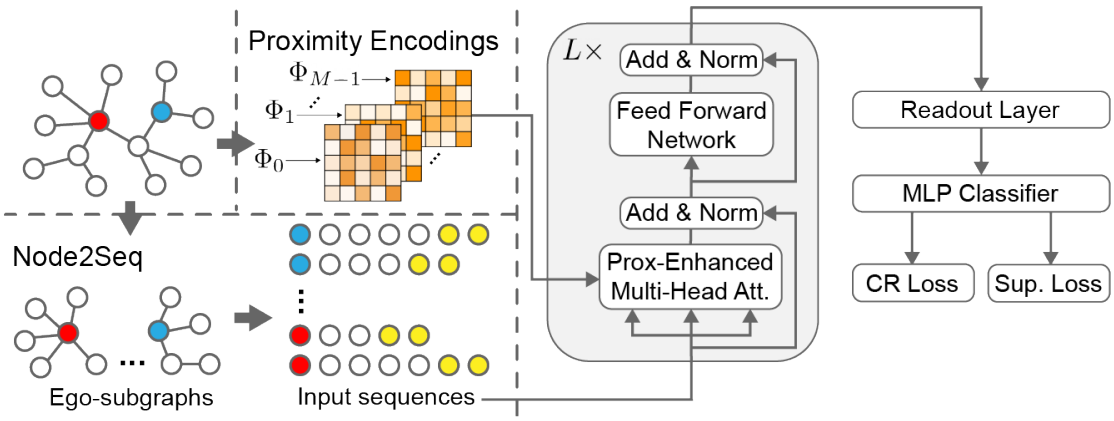


Figure 2.8: Diagram of initial graph processing and model architecture for Gophormer. Node color-coding scheme: red/blue: example node, white: context node, yellow: global node. Prox: proximity; Att.: attention; MLP: multilayer perceptron; CR: consistency regularization; Sup.: supervised. Adapted from Zhao et al. [43].

where

$S$ = number of ego-graphs sampled per epoch for each training node

$\mathcal{G}_c$ = $\{G_c^{(s)} \mid s \in \{1, 2, ..., S\}\}$ = $v_c$'s training ego-graphs

$\mathbf{X}_c^{(s)}$ = node features of $G_c^{(s)}$

$n_g$ = number of global nodes added to each ego-graph

$d$ = hidden dimension of the transformer

$\mathbf{C}$ = $\{\mathbf{c}_i \in \mathbb{R}^{d\times 1}, i \in 1, ..., n_g\}$ = learnable features of the global nodes

$$\text{attention score for a node pair } \langle v_i, v_j \rangle\text{: } \alpha_{ij} = \frac{\left(\boldsymbol{h}_i \mathbf{W}_Q\right)\left(\boldsymbol{h}_j \mathbf{W}_K\right)^T}{\sqrt{d}} + \phi_{ij}^T \boldsymbol{b} \tag{2.18}$$

where

$\boldsymbol{h}_i$ = hidden representation at position $i$

$\mathbf{W}_Q$ = matrix used to project the input sequence to the query matrix

$\mathbf{W}_K$ = matrix used to project the input sequence to the key matrix

$d$ = hidden dimension of the transformer

$M$ = number of views of structural information

$\boldsymbol{b} \in \mathbb{R}^{M\times 1}$ = learnable parameters

$$
\begin{aligned}
\phi_{ij} &= \text{Concat}\left(\Phi_m(v_i, v_j) \mid m \in 0, 1, ..., M-1\right) = \text{proximity encoding vector} \\
\Phi_m(v_i, v_j) &= \begin{cases} \tilde{\mathbf{A}}^m[i,j], & \text{if } m < M-1 \\ \mathbb{I}(i,j), & \text{if } m = M-1 \end{cases} = \text{structural encoding function} \\
\mathbb{I}(i,j) &= \begin{cases} 1, & \text{if global nodes exist in } \langle v_i, v_j \rangle \\ 0, & \text{otherwise} \end{cases} \\
\tilde{\mathbf{A}} &= \text{Norm}(\mathbf{A} + \mathbf{I}) = \text{normalized adjacency matrix with self-loops} \\
.^T &= \text{transposition}
\end{aligned}
$$

$$
\text{loss: } \mathcal{L} = \mathcal{L}_{\text{sup}} + \lambda \left( \frac{1}{S} \sum_{v_c \in V_L} \sum_{s=1}^{S} \left\| \bar{\boldsymbol{y}}'_c - \tilde{\boldsymbol{y}}_c^{(s)} \right\|_2^2 \right) \tag{2.19}
$$

where

$\mathcal{L}_{\text{sup}}$ = supervised loss

$\lambda$ = coefficient

$S$ = number of ego-graphs sampled per epoch for each training node

$v_c$ = center node

$V_L$ = labeled training nodes

$\bar{\boldsymbol{y}}'_c$ = sharpened average distribution (as in Feng et al. [44]) of ego-graphs

$C$ = number of classes

$\tilde{\boldsymbol{y}}_c^{(s)} \in \mathbb{R}^{C \times 1}$ = classification result for the training ego-graph $s$

$$
\text{final prediction for a center node } v_c\text{: } \tilde{\boldsymbol{y}}_c = \frac{1}{S'} \sum_{s=1}^{S'} \tilde{\boldsymbol{y}}_c^{(s)} \tag{2.20}
$$

where

$S'$ = number of ego-graphs sampled for each testing node

$C$ = number of classes

$\tilde{\boldsymbol{y}}_c^{(s)} \in \mathbb{R}^{C \times 1}$ = classification result for the testing ego-graph $s$

### Exphormer

Exphormer uses a sparse attention mechanism based on local neighborhoods, expander graphs, and virtual global nodes [45] (Figure 2.9). Local neighborhood attention helps capture graph connectivity information and contributes $O(E)$ interaction edges, where $E$ is the number of edges in the input graph [45]. Expander graphs have useful spectral and random-walk-mixing properties that allow for the propagation of information between distant nodes via alternative short paths [45]. Compared to virtual nodes, expander graphs have the advantage of avoiding information bottlenecks [45]. A constant-degree expander graph, as used in Exphormer, contributes $O(N)$ edges, where $N$ is the number of nodes in the graph [45]. Lastly, global attention via a small, constant number of virtual nodes creates a "storage sink" and contributes $O(N)$ edges [45]. Ultimately, Exphormer achieves a linear computational complexity of $O(N + E)$ on a graph with $N$ nodes and $E$ edges [45].

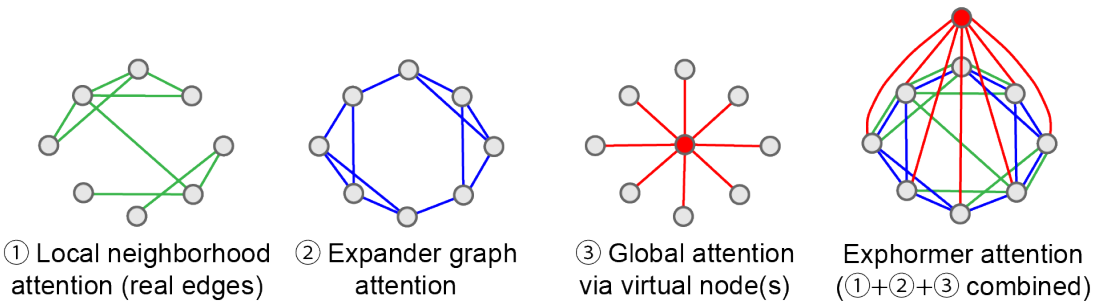


Figure 2.9: Components of Exphormer's sparse interaction graph/attention mechanism. The shown expander graph is of degree 3, as an example. Adapted from Shirzad et al. [45].

# Chapter 3

# Methods

To construct a competitive GT architecture that shows promise for scaling up to large graphs, we added GraphGPS-inspired MPNN layers to the lab's modified version of SAN's architecture (Figure 3.1). SAN was chosen as a starting point due to its strong performance on graph-level tasks and the convenience of its modular code [17]. In the lab's modified version of SAN, edge data have been removed, the graph-Laplacian-based positional encoding has been replaced with random walk landing probabilities [46], and the multilayer perceptron (MLP) readout layer has been modified for scalability and performance (Figure 3.2). Our addition of MPNN layers to the lab's modified version of SAN is based on the idea that message passing can complement self-attention to improve GT performance. For instance, by gathering information only from neighboring nodes, message passing can help increase the signal-to-noise ratio on homophilic graphs. The random walk landing probabilities are based on the random walk diffusion process, and are defined as the landing probabilities of the nodes to themselves [46] (Equation 3.1). When the $k$-hop topological neighborhood of each node is unique for sufficiently large $k$, the random walk probabilities provide unique node representations [46]. Our model's multi-head attention computation is the same as that of SAN (*cf.* Equations 2.7 and 2.9), except that edge data are not included in the calculation of attention weights (Equation 3.2). PyTorch Geometric [47] and Deep Graph Library [48]

were used to build the model.

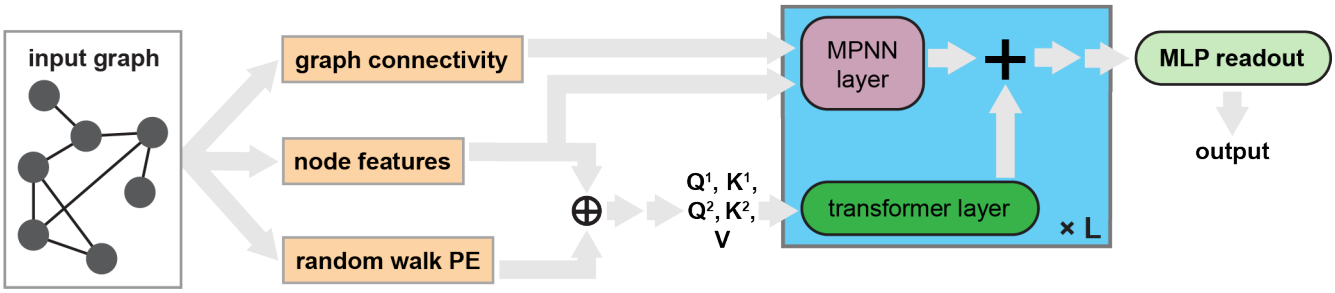


Figure 3.1: Diagram of our architecture. PE: positional encoding; MPNN: message-passing neural network; MLP: multilayer perceptron.

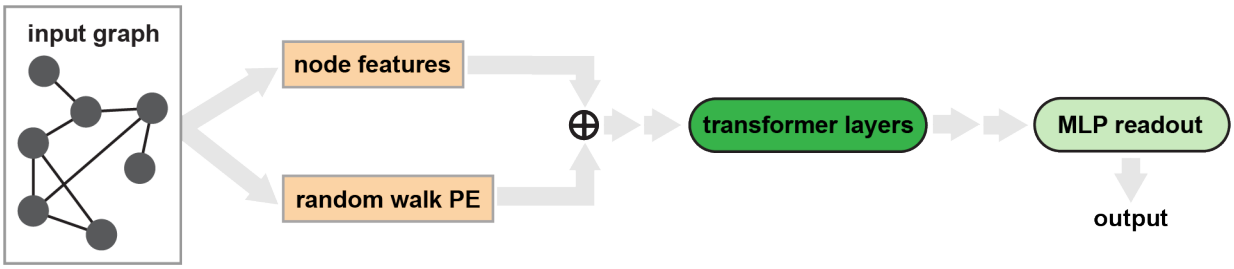


Figure 3.2: Diagram of the lab's modified version of SAN. PE: positional encoding; MLP: multilayer perceptron.

$$p_i^{\text{RWPE}} = \left[\text{RW}_{ii}, \text{RW}_{ii}^2, ..., \text{RW}_{ii}^k\right] \in \mathbb{R}^k \tag{3.1}$$

where

$$\begin{aligned}
\text{RW} &= AD^{-1} = \text{random walk operator} \\
\text{RW}_{ii} &= \text{random walk landing probability of node } i \text{ to itself} \\
k &= \text{number of steps of random walk} \\
\text{RWPE} &= \text{random walk positional encoding}
\end{aligned}$$

$$\hat{w}_{ij}^{k,\ell} = \begin{cases} \frac{Q^{1,k,\ell} h_i^\ell \circ K^{1,k,\ell} h_j^\ell}{\sqrt{d_k}} & \text{if } i \text{ and } j \text{ are connected in sparse graph} \\ \frac{Q^{2,k,\ell} h_i^\ell \circ K^{2,k,\ell} h_j^\ell}{\sqrt{d_k}} & \text{otherwise} \end{cases} \tag{3.2}$$

where $Q^{1,k,\ell}, Q^{2,k,\ell}, K^{1,k,\ell}, K^{2,k,\ell} \in \mathbb{R}^{d_k \times d}$,

$Q^1, K^1$ = query and key projections for pairs of connected nodes

$Q^2, K^2$ = query and key projections for pairs of disconnected nodes

$h_i^\ell$ = node $i$'s features at layer $\ell$

$d$ = hidden dimension

$d_k$ = $\frac{d}{\text{number of heads}} = \frac{d}{H}$ = dimension of a head

$\circ$ = element-wise multiplication

## 3.1 Datasets

The Cora [32] and CiteSeer [33] datasets were used for model evaluation in a setting where capturing long-range interactions is not necessary for strong model performance (Table 3.1). In both datasets, nodes are scientific publications and edges are citation relationships [49–52]. Each node/publication is classified into one of several classes (7 classes in Cora; 6 classes in CiteSeer) and has a feature vector (with length 1433 in Cora; length 3703 in CiteSeer) of zeros and ones, where each digit indicates whether or not a certain word appears in the publication [49–52].

Table 3.1: Medium-scale benchmark datasets for model evaluation.

| | **# Nodes** | **# Edges** | **# Classes** |
|---|---|---|---|
| Cora | 2708 | 10,556 | 7 |
| CiteSeer | 3327 | 9228 | 6 |

## 3.2 Model Formulation

To formulate our model, we experimented with various ways of improving the node classification accuracy of the lab's modified version of SAN on the Cora dataset, as detailed below.

### 3.2.1 Hyperparameter Optimization

The following hyperparameters were tuned via grid search: initial learning rate, gamma, number of learned positional encoding heads, number of GT heads, layer normalization, and batch normalization (Table 3.2). Such tuning significantly improves the model's node classification performance (Table 3.3).

Table 3.2: Tested hyperparameter values.

| **Hyperparameter** | **Tested values** |
|---|---|
| initial learning rate | 1e−4, 1e−3, 1e−2 |
| gamma | 1e−6, 1e−5, 1e−4 |
| number of learned positional encoding heads | 2, 4, 8, 16 |
| number of GT heads | 2, 4, 8, 16 |
| layer normalization | `True`, `False` |
| batch normalization | `True`, `False` |

Table 3.3: Model performance before and after hyperparameter optimization.

| **Hyperparameter optimization** | **Node classification accuracy on Cora** |
|---|---|
| − | $74.88 \pm 1.24\%$ |
| + | $77.98 \pm 1.24\%$ |

### 3.2.2 Positional Encoding

After hyperparameter tuning, we experimented with positional encodings. Hussain et al. [53] proposed a positional encoding based on the precalculated singular value decomposition (SVD) of the graph's adjacency matrix (Equations 3.3 and 3.4). Unlike Laplacian eigenvectors, singular values and vectors of the SVD can be used for both directed and undirected graphs [53]. This SVD-based positional encoding, alone or coupled with random walk landing probabilities, results in node classification performance similar to that of the original model that only uses random walk positional encodings (Table 3.4). Excluding all positional

encodings also results in similar node classification accuracies (Table 3.4). Hence, for subsequent experiments, the original random walk positional encoding was kept in the model architecture.

$$\mathbf{A} \overset{\text{SVD}}{\approx} \mathbf{U}\Sigma\mathbf{V}^T = (\mathbf{U}\sqrt{\Sigma}) \cdot (\mathbf{V}\sqrt{\Sigma})^T = \hat{\mathbf{U}}\hat{\mathbf{V}}^T \tag{3.3}$$

where

| | |
|---|---|
| $\mathbf{A}$ | = adjacency matrix, with self-loops |
| $\Sigma \in \mathbb{R}^{r \times r}$ | = diagonal matrix containing the top $r$ singular values |
| $N$ | = number of nodes in the graph |
| $\mathbf{U} \in \mathbb{R}^{N \times r}$ | = matrix containing the $r$ left singular vectors as columns |
| $\mathbf{V} \in \mathbb{R}^{N \times r}$ | = matrix containing the $r$ right singular vectors as columns |
| $.^T$ | = transposition |

$$\hat{\Gamma} = \hat{\mathbf{U}} \| \hat{\mathbf{V}} \tag{3.4}$$

where

| | |
|---|---|
| $\|$ | = concatenation along the columns |
| $N$ | = number of nodes in the graph |
| $\mathbf{U} \in \mathbb{R}^{N \times r}$ | = matrix containing the $r$ left singular vectors as columns |
| $\mathbf{V} \in \mathbb{R}^{N \times r}$ | = matrix containing the $r$ right singular vectors as columns |

### 3.2.3 Adding MPNNs

Next, we experimented with adding MPNN layers to the transformer architecture.

#### MPNN Placement

Inspired by GraphGPS's architecture [9], we summed the output of each transformer layer with that of a SplineConv [36] MPNN layer. Specifically, we tried adding the SplineConv

Table 3.4: Model performance with various positional encoding(s). PE: positional encoding.

| Positional encoding(s) | Node classification accuracy on Cora |
|---|---|
| SVD-based PE | $74.75 \pm 1.31\%$ |
| Random walk landing probabilities | $75.02 \pm 1.44\%$ |
| Both SVD-based PE and random walk landing probabilities | $75.50 \pm 0.73\%$ |
| No PE | $75.62 \pm 1.64\%$ |

output at four different locations within each layer (Figure 3.3). Addition of the SplineConv output at position 3 results in the best node classification performance and allows the model to significantly outperform the original, hyperparameter-optimized model (Table 3.5; *cf.* Table 3.3). Drawing inspiration from GraphTrans [37], we also tried adding SplineConv MPNN layers before the self-attention mechanism, rather than combining MPNN and transformer outputs at each layer. On the Cora dataset, this approach results in an average node classification accuracy of $79.45 \pm 0.97\%$, which is similar to the accuracy of the model with SplineConv output added at position 3 (Table 3.5). Hence, pre-self-attention MPNN layers were not used for later experiments.

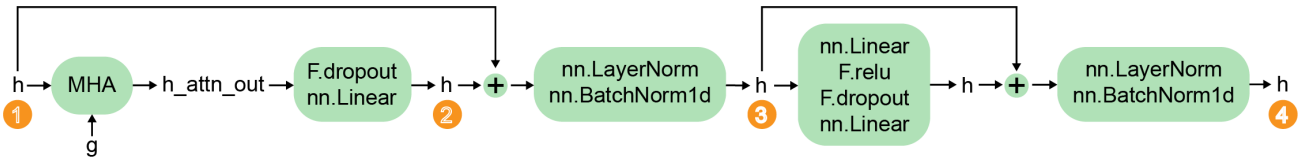


Figure 3.3: Diagram of an individual layer in our model. h: feature vector; g: graph object; MHA: multi-head attention. Each numbered orange circle represents a possible location for the addition of the output of a MPNN.

Table 3.5: Best location and model performance for each tested MPNN type.

| MPNN type | Best location | Node classification accuracy on Cora |
|---|---|---|
| SplineConv | 3 | $80.58 \pm 0.62\%$ |
| GATConv | 3 | $79.36 \pm 1.71\%$ |
| GCN2Conv | 2 | $79.19 \pm 0.61\%$ |
| SAGEConv | 1 | $79.05 \pm 1.01\%$ |

#### MPNN Type

Taking the GraphGPS-inspired model with SplineConv output added at position 3, we tried substituting SplineConv [36] with alternative MPNNs, i.e., GATConv [35], GCN2Conv [34], and SAGEConv [19], and also experimented again with the location at which the MPNN output is added (*cf.* Figure 3.3). The best GATConv substitution (at position 3) results in node classification performance similar to that of the model using SplineConv, whereas the best GCN2Conv and SAGEConv substitutions (at positions 2 and 1, respectively) result in significantly worse performance (Table 3.5). The GATConv substitution was kept in the model architecture for subsequent experiments.

### 3.2.4 Structure Extractor

Adding a $k$-subtree structure extractor as in SAT to the GraphGPS-inspired model results in node classification performance similar to that of the model without the structure extractor (Table 3.6). Thus, the structure extractor was not kept in the model architecture.

Table 3.6: Model performance with and without a structure extractor.

| Structure extractor | Node classification accuracy on Cora |
|---|---|
| + | $77.30 \pm 1.51\%$ |
| − | $79.36 \pm 1.71\%$ |

# Chapter 4

# Benchmarking Experiments

## 4.1 Procedure

We evaluated our model by comparing it to four baselines:

1. The lab's modified version of SAN
2. GraphGPS (as implemented in https://github.com/rampasek/GraphGPS)
3. GCN (as implemented in https://github.com/pyg-team/pytorch_geometric/blob/master/examples/gcn.py)
4. GAT (as implemented in https://github.com/pyg-team/pytorch_geometric/blob/master/examples/gat.py)

For each model, the maximum number of epochs was set to 1000. Each result below is the average ($\pm$ one standard deviation) of 10 differently-seeded runs.

## 4.2 Results

### 4.2.1 Node Classification Accuracy

We first evaluated our model and the baseline models based on their node classification accuracies on the Cora and CiteSeer datasets (Table 4.1). Our model obtains higher accuracies than the lab's modified version of SAN on both the Cora and CiteSeer datasets, indicating that the addition of MPNN layers and hyperparameter optimization significantly improves model performance. Compared to GraphGPS, our model obtains higher accuracies on the Cora dataset and similar accuracies on the CiteSeer dataset, showing that our model is competitive with this common graph transformer baseline. Finally, the GNN baselines, i.e., GCN and GAT, both obtain higher accuracies than our model on the Cora and CiteSeer datasets, demonstrating that further work is needed to achieve competitiveness with GNNs.

Table 4.1: Node classification accuracies.

| | **Cora** | **CiteSeer** |
|---|---|---|
| **Our model** | $80.43 \pm 1.03\%$ | $64.44 \pm 1.45\%$ |
| **Modified SAN** | $74.52 \pm 1.62\%$ | $55.47 \pm 2.39\%$ |
| **GraphGPS** | $75.25 \pm 2.04\%$ | $64.71 \pm 1.00\%$ |
| **GCN** | $81.97 \pm 0.42\%$ | $71.42 \pm 0.78\%$ |
| **GAT** | $82.51 \pm 0.58\%$ | $71.45 \pm 0.89\%$ |

"Modified SAN" refers to the lab's modified version of SAN. This model was run on a different machine.

### 4.2.2 Runtime

We also measured the amount of time it takes to run each model (Table 4.2). Our model runs faster than the lab's modified version of SAN on both the Cora and CiteSeer datasets. This improvement in speed may be attributable to changes we made to the hyperparameters of the lab's modified version of SAN via hyperparameter optimization. Our model is slower to run than the other baselines, i.e., GraphGPS, GCN, and GAT, reflecting the high time

complexity of the SAN architecture in our model [9, 17].

Table 4.2: Time elapsed (in seconds) during model training and evaluation.

| | Cora | CiteSeer |
|---|---|---|
| **Our model** | $148.93 \pm 17.26$ | $214.45 \pm 19.66$ |
| **Modified SAN** | $249.49 \pm 21.98$ | $321.74 \pm 3.02$ |
| **GraphGPS** | $36.23 \pm 25.74$ | $36.91 \pm 12.48$ |
| **GCN** | $8.85 \pm 9.02$ | $8.29 \pm 9.02$ |
| **GAT** | $2.76 \pm 1.63$ | $2.78 \pm 1.34$ |

"Modified SAN" refers to the lab's modified version of SAN. This model was run on a different machine.

### 4.2.3 GPU Memory Usage

Lastly, we measured the GPU memory usage of each model during training (Table 4.3). For each model, the GPU memory usage is the same across all runs, as expected. Compared to the lab's modified version of SAN, our model uses 1.424 GB and 1.570 GB more GPU memory on the Cora and CiteSeer datasets, respectively. These differences in GPU memory usage may be attributable to the MPNN layers that we added to the lab's modified version of SAN. Both our model and the lab's modified version of SAN use several more gigabytes of GPU memory than the other baselines, i.e., GraphGPS, GCN, and GAT, reflecting the high space complexity of the SAN architecture [9, 17].

Table 4.3: GPU memory usage (in GB).

| | Cora | CiteSeer |
|---|---|---|
| **Our model** | $6.180 \pm 0.000$ | $8.533 \pm 0.000$ |
| **Modified SAN** | $4.756 \pm 0.000$ | $6.963 \pm 0.000$ |
| **GraphGPS** | $1.082 \pm 0.000$ | $1.128 \pm 0.000$ |
| **GCN** | $1.02 \pm 0.00$ | $1.099 \pm 0.000$ |
| **GAT** | $1.049 \pm 0.000$ | $1.105 \pm 0.000$ |

"Modified SAN" refers to the lab's modified version of SAN. This model was run on a different machine.

# Chapter 5

# Conclusions

## 5.1 Summary

With the goal of optimizing the architecture of each GT-layer block of a scalable GT undergoing design by the lab, we formulated a GT that builds upon the lab's modified version of SAN. Specifically, we added GraphGPS-inspired MPNN layers to the model and performed hyperparameter tuning. To formulate this SAN/GraphGPS hybrid architecture, we experimented with various positional encodings, types of MPNN layers, and locations at which to add the MPNN layer output, as well as the use of a $k$-subtree structure extractor. We evaluated our model by comparing it to four baselines: (i) the lab's modified version of SAN, (ii) GraphGPS, (iii) GCN, and (iv) GAT. Specifically, we measured each model's node classification accuracy, runtime, and GPU memory usage on the Cora and CiteSeer datasets. Compared to the lab's modified version of SAN, our model is faster to run and achieves higher node classification accuracies on the Cora and CiteSeer datasets. In terms of GPU memory usage, our model uses 1.424 GB and 1.570 GB more memory than the lab's modified version of SAN on the Cora and CiteSeer datasets, respectively; these differences may be attributable to our addition of MPNN layers to the lab's modified version of SAN. Encouragingly, our model is competitive with GraphGPS in terms of node classification ac-

curacy on the Cora and CiteSeer datasets. GraphGPS, GCN, and GAT are all faster to run and use less GPU memory than our model; these differences in runtime and memory usage may be attributed to the high time and space complexity of the SAN architecture within our model. GCN and GAT also achieve higher node classification accuracies than our model, indicating that further work is needed to make our model competitive with GNNs.

## 5.2 Future Work

Future work may focus on further modification of our model's architecture to achieve node classification accuracies competitive with those of GNNs, as well as integration of our model into the scalable GT architecture that the lab is currently developing. Ideally, after such integration, the scalable GT will be benchmarked and applied to datasets with industrial applications, such as financial forensics datasets [4]. Also, since one of the advantages of GTs over GNNs is the potential of GTs to effectively capture long-range interactions in graphs, it would also be interesting to benchmark our model and the scalable GT on datasets that require long-range reasoning, e.g., the Long Range Graph Benchmark datasets [15].

# Bibliography


[1] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. "LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation". In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval* (2020). DOI: 10.1145/3397271.3401063.

[2] Chen Gao, Xiang Wang, Xiangnan He, and Yong Li. "Graph Neural Networks for Recommender System". In: *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*. 2022, pp. 1623–1625. DOI: 10.1145/3488560.3501396.

[3] Yixin Liu, Ming Jin, Shirui Pan, Chuan Zhou, Yu Zheng, Feng Xia, and Philip S. Yu. "Graph Self-Supervised Learning: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* (2022), p. 1. DOI: 10.1109/TKDE.2022.3172903.

[4] Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I. Weidele, Claudio Bellei, Tom Robinson, and Charles E. Leiserson. "Anti-Money Laundering in Bitcoin: Experimenting with Graph Convolutional Networks for Financial Forensics". In: *KDD '19 Workshop on Anomaly Detection in Finance, August 2019*. DOI: 10.48550/arXiv.1908.02591.

[5] Laurianne David, Amol Thakkar, Rocío Mercado, and Ola Engkvist. "Molecular Representations in AI-Driven Drug Discovery: A Review and Practical Guide". In: *Journal of Cheminformatics* 12.1 (2020), p. 56. DOI: 10.1186/s13321-020-00460-5.

[6] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2021), pp. 4–24. DOI: 10.1109/TNNLS.2020.2978386.

[7] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the 34th International Conference on Machine Learning*. 2017, pp. 1263–1272. DOI: 10.48550/arXiv.1704.01212.

[8] Uri Alon and Eran Yahav. "On the Bottleneck of Graph Neural Networks and its Practical Implications". In: *ICLR 2021*. DOI: 10.48550/arXiv.2006.05205.

[9] Ladislav Rampášek, Mikhail Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. "Recipe for a General, Powerful, Scalable Graph Transformer". In: *36th Conference on Neural Information Processing Systems (NeurIPS 2022)*. DOI: 10.48550/arXiv.2205.12454.

[10] Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M. Bronstein. "Understanding Over-Squashing and Bottlenecks on Graphs via Curvature". In: *ICLR 2022*. DOI: 10.48550/arXiv.2111.14522.

[11] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. "Measuring and Relieving the Over-Smoothing Problem for Graph Neural Networks from the Topological View". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.4 (2020), pp. 3438–3445. DOI: 10.1609/aaai.v34i04.5747.

[12] T. Konstantin Rusch, Michael M. Bronstein, and Siddhartha Mishra. *A Survey on Oversmoothing in Graph Neural Networks*. 2023. DOI: 10.48550/arXiv.2303.10993.

[13] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. "Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.1 (2019), pp. 4602–4609. DOI: 10.1609/aaai.v33i01.33014602.

[14] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. "How Powerful are Graph Neural Networks?" In: *ICLR 2019*. DOI: 10.48550/arXiv.1810.00826.

[15] Vijay Prakash Dwivedi, Ladislav Rampášek, Mikhail Galkin, Ali Parviz, Guy Wolf, Anh Tuan Luu, and Dominique Beaini. "Long Range Graph Benchmark". In: *36th Conference on Neural Information Processing Systems (NeurIPS 2022)*. DOI: 10.48550/arXiv.2206.08164.

[16] Kai Guo, Xiaofeng Cao, Zhining Liu, and Yi Chang. "Taming Over-Smoothing Representation on Heterophilic Graphs". In: *Information Sciences* 647 (2023). DOI: 10.1016/j.ins.2023.119463.

[17] Devin Kreuzer, Dominique Beaini, William L. Hamilton, Vincent Létourneau, and Prudencio Tossou. "Rethinking Graph Transformers with Spectral Attention". In: *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*. DOI: 10.48550/arXiv.2106.03893.

[18] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *ICLR 2017*. DOI: 10.48550/arXiv.1609.02907.

[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *31st Conference on Neural Information Processing Systems (NeurIPS 2017)*. DOI: 10.48550/arXiv.1706.02216.

[20] Zhenqin Wu, Bharath Ramsundar, Evan N. Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S. Pappu, Karl Leswing, and Vijay Pande. "MoleculeNet: A Benchmark for Molecular Machine Learning". In: *Chemical Science* 9.2 (2018), pp. 513–530. DOI: 10.1039/C7SC02664A.

[21] Julia Balla. *Over-Squashing in Graph Neural Networks*. URL: https://minyoungg.github.io/MIT-deeplearning-blogs/2021/12/09/oversquashing-in-gnns/.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *31st Conference on Neural Information Processing Systems (NeurIPS 2017)*. DOI: 10.48550/arXiv.1706.03762.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of NAACL-HLT 2019*, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[24] Tom B Brown et al. "Language Models are Few-Shot Learners". In: *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*. DOI: 10.48550/arXiv.1706.02216.

[25] OpenAI. *GPT-4 Technical Report*. 2023. DOI: 10.48550/arXiv.2303.08774.

[26] Stéphane d'Ascoli, Hugo Touvron, Matthew L. Leavitt, Ari S. Morcos, Giulio Biroli, and Levent Sagun. "ConViT: Improving Vision Transformers with Soft Convolutional Inductive Biases". In: *Proceedings of the 38th International Conference on Machine Learning* (2021). DOI: 10.1088/1742-5468/ac9830.

[27] Kai Han et al. "A Survey on Vision Transformer". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.1 (2023), pp. 87–110. DOI: 10.1109/TPAMI.2022.3152247.

[28] Rishi Bommasani et al. "On the Opportunities and Risks of Foundation Models". In: *arXiv* (2021). URL: https://crfm.stanford.edu/assets/report.pdf.

[29] Alec Radford et al. "Learning Transferable Visual Models From Natural Language Supervision". In: *Proceedings of the 38th International Conference on Machine Learning* (2021). DOI: 10.48550/arXiv.2103.00020.

[30] Michael Galkin. *GraphGPS: Navigating Graph Transformers*. URL: https://towardsdatascience.com/graphgps-navigating-graph-transformers-c2cc223a051c.

[31] Kezhi Kong, Jiuhai Chen, John Kirchenbauer, Renkun Ni, C. Bayan Bruss, and Tom Goldstein. "GOAT: A Global Transformer on Large-Scale Graphs". In: *Proceedings of the 40th International Conference on Machine Learning*. 2023, pp. 17375–17390.

[32] Andrew Kachites Mccallum. "Automating the Construction of Internet Portals with Machine Learning". In: *Information Retrieval* 3 (2000), pp. 127–163. DOI: 10.1023/A:1009953814988.

[33] C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. "CiteSeer: An Automatic Citation Indexing System". In: *Proceedings of the Third ACM Conference on Digital Libraries*. 1998, pp. 89–98. DOI: 10.1145/276675.276685.

[34] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. "Simple and Deep Graph Convolutional Networks". In: *Proceedings of the 37th International Conference on Machine Learning* (2020). DOI: 10.48550/arXiv.2007.02133.

[35] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. "Graph Attention Networks". In: *ICLR 2018*. DOI: 10.48550/arXiv.1710.10903.

[36] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. "SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. DOI: 10.1109/CVPR.2018.00097.

[37] Zhanghao Wu, Paras Jain, Matthew A. Wright, Azalia Mirhoseini, Joseph E. Gonzalez, and Ion Stoica. "Representing Long-Range Context for Graph Neural Networks With Global Attention". In: *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*. DOI: 10.48550/arXiv.2201.08821.

[38] Dexiong Chen, Leslie O'Bray, and Karsten Borgwardt. "Structure-Aware Transformer for Graph Representation Learning". In: *Proceedings of the 39th International Conference on Machine Learning* (2022). DOI: 10.48550/arXiv.2202.03036.

[39] BorgwardtLab. *Structure-Aware Transformer*. URL: https://github.com/BorgwardtLab/SAT.

[40] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. "Do Transformers Really Perform Bad for Graph Representation?" In: *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*. DOI: 10.48550/arXiv.2106.05234.

[41] Liheng Ma, Chen Lin, Derek Lim, Adriana Romero-Soriano, Puneet K. Dokania, Mark Coates, Philip Torr, and Ser-Nam Lim. "Graph Inductive Biases in Transformers Without Message Passing". In: *Proceedings of the 40th International Conference on Machine Learning* (2023). DOI: 10.48550/arXiv.2305.17589.

[42] Jinsong Chen, Kaiyuan Gao, Gaichao Li, and Kun He. "NAGphormer: A Tokenized Graph Transformer for Node Classification in Large Graphs". In: *ICLR 2023*. DOI: 10.48550/arXiv.2206.04910.

[43] Jianan Zhao, Chaozhuo Li, Qianlong Wen, Yiqi Wang, Yuming Liu, Hao Sun, Xing Xie, and Yanfang Ye. "Gophormer: Ego-Graph Transformer for Node Classification". In: *Woodstock '18: ACM Symposium on Neural Gaze Detection*. DOI: 10.48550/arXiv.2110.13094.

[44] Wenzheng Feng, Jie Zhang, Yuxiao Dong, Yu Han, Huanbo Luan, Qian Xu, Qiang Yang, Evgeny Kharlamov, and Jie Tang. "Graph Random Neural Networks for Semi-Supervised Learning on Graphs". In: *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*. DOI: 10.48550/arXiv.2005.11079.

[45] Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J. Sutherland, and Ali Kemal Sinop. "Exphormer: Sparse Transformers for Graphs". In: *Proceedings of the 40th International Conference on Machine Learning* (2023). DOI: 10.48550/arXiv.2303.06147.

[46] Vijay Prakash Dwivedi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. "Graph Neural Networks With Learnable Structural and Positional Representations". In: *ICLR 2022*. DOI: 10.48550/arXiv.2110.07875.

[47] Matthias Fey and Jan Eric Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR 2019*. DOI: 10.48550/arXiv.1903.02428.

[48] Minjie Wang et al. "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks". In: *arXiv preprint arXiv:1909.01315* (2019).

[49] *Papers With Code: Cora*. URL: https://paperswithcode.com/dataset/cora.

[50] *Papers With Code: Citeseer*. URL: https://paperswithcode.com/dataset/citeseer.

[51] *DGL Data: CoraGraphDataset*. URL: https://docs.dgl.ai/en/1.1.x/generated/dgl.data.CoraGraphDataset.html.

[52] *DGL Data: CiteseerGraphDataset*. URL: https://docs.dgl.ai/en/1.1.x/generated/dgl.data.CiteseerGraphDataset.html.

[53] Md Shamim Hussain, Mohammed J. Zaki, and Dharmashankar Subramanian. "Global Self-Attention as a Replacement for Graph Convolution". In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2022, pp. 655–665. DOI: 10.1145/3534678.3539296.