Elucidating Graph Neural Networks,
Transformers, and Graph Transformers

Contents

1 Introduction **2**

2 Graph Representation Learning **2**
2.1 Preliminaries on Graphs and Notation 3

3 Graph Neural Networks and Message Passing **4**
3.1 Permutation Invariance and Equivariance 5
3.2 Traditional Graph Neural Network Architectures 5
3.3 Expressivity and Limitations 8

4 Transformers **10**
4.1 The Transformer Block 10
4.2 Self-attention 11
4.3 Positional Encodings 13
4.4 Skip connections, Layer Normalization, and other details 13

5 Graph Transformers **15**
5.1 The Graph Transformer Block 16
5.2 Graph Positional and Structural Encodings 17
5.3 Global Attention and Scaling to Large Graphs 18

©2024 Haitz Sáez de Ocáriz Borde.

Elucidating Graph Neural Networks, Transformers, and Graph Transformers

Haitz Sáez de Ocáriz Borde
University of Oxford
Oxford, UK

1. Introduction

Transformers now serve as the predominant artificial deep neural network. Initially conceived within the realm of Natural Language Processing (NLP), they have seamlessly transitioned to various other domains of machine learning, such as Computer Vision (CV), through innovations like the Vision Transformer (ViT), and more recently to generative modeling with architectures such as Diffusion Transformers (DiTs). However, these adapted versions of the Transformer still preserve the fundamental building blocks of the original virtually unchanged. Hence, Transformers have established themselves as the more widely used *neural architecture*. Concurrently, Transformers are entering the domain of graph representation learning, an area traditionally dominated by Graph Neural Networks (GNNs) and the message-passing framework.

This paper aims to present an overview of graph representation learning, delve into traditional GNNs, revisit the Transformer architecture, and explore the adaptation of Transformers for graphs. Additionally, we seek to examine the relationship between Graph Transformers and traditional GNNs. We discuss all these topics from a unified perspective, as we believe there is a lack of resources in the literature that consolidate these concepts into a single manuscript. We presume the reader is familiar with Deep Learning.

2. Graph Representation Learning

Graph representation learning involves creating algorithms and models designed to process and glean insights from data pertaining to nodes, edges, and substructures within a graph. In essence, we want to model functions over graphs. Graphs not only offer an elegant theoretical framework but also a mathematical basis for analyzing real-world systems. This work particularly focuses on graph representation learning in the context of Deep Learning.

Intuitively, one can think of a graph as a collection of entities or nodes which are somehow related to each other. This is a very generic concept which can be used to model a variety of real-world systems. For example, a social network is composed of people (nodes) which are connected to each other (edges) because they have family, friends, colleagues, etc. Likewise, a molecule has atoms (nodes) which are connected to each other with bonds (edges). A country is composed of cities (nodes) which are connected with roads and highways (edges). The brain itself is made of

interconnected neurons. In the same way, artificial neural networks themselves are computational graphs too.

2.1 Preliminaries on Graphs and Notation

Next, let us define graphs mathematically. We define a graph as an ordered tuple:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}), \tag{1}$$

where $\mathcal{V}$ is a set of nodes (or vertices), and $\mathcal{E} \subseteq (\mathcal{V} \times \mathcal{V})$ is a 2-tuple set representing the edges (or links) in the graph. If v_i and v_j are nodes in $\mathcal{G}$, their relation is recorded with $(v_i, v_j) \in \mathcal{E}$ if the edge is directed from v_i to v_j. Edges can be directed or undirected. Directed edges are uni-directional relations from a source node v_i to a target node v_j; thus, $(v_i, v_j) \in \mathcal{E}$, and importantly, $(v_i, v_j) \neq (v_j, v_i)$. In contrast, undirected edges are bi-directional, so $(v_i, v_j) = (v_j, v_i)$. The (one-hop) neighborhood of a node v_i is the set of nodes that share an edge with v_i, denoted as $\mathcal{N}(v_i) = \{v_j | (v_i, v_j) \in \mathcal{E}\}$.

Graphs can be represented using matrices. For a graph with $N = |\mathcal{V}|$ number of nodes, its adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ represents the connectivity structure between nodes. $\mathbf{A}$ can be weighted or unweighted. If it is weighted, its entries $A_{ij} \in \mathbb{R}$ represent the weight or strength of the connection, and if $(v_i, v_j) \notin \mathcal{E}$, then $A_{ij} = 0$. In the case of an unweighted adjacency matrix, $A_{ij} = 1$ when there is an edge and $A_{ij} = 0$ when there is no edge. So that,

$$A_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in \mathcal{E} \\ 0 & \text{if } (v_i, v_j) \notin \mathcal{E}. \end{cases} \tag{2}$$

In general for the purposes of this text, we can assume our graphs' edges to be unweighted and undirectional, hence the adjacency matrices we are concerned about are binary and symmetric. A diagonal degree matrix $\mathbf{D} \in \mathbb{R}^{N \times N}$ is defined as the matrix where each entry on the diagonal is the row-sum of the adjacency matrix: $D_{ii} = \sum_j A_{ij}$. Based on this we can define the Laplacian of the graph

$$\mathbf{L} = \mathbf{D} - \mathbf{A}, \tag{3}$$

or equivalently,

$$L_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

The graph Laplacian provides valuable information about the topology of a graph and is utilized in various mathematical and computational techniques for analyzing and understanding graphs and networks. Geometrically, it can be interpreted as quantifying how similar a node is to its neighbors. We will come back to it later in Section 5.2.

In our context we are interested in graphs with node features. Each node $v_i \in \mathcal{V}$ will have an associated D-dimensional feature vector $\mathbf{x}_i \in \mathbb{R}^{1 \times D}$. We can stack the feature vectors for all nodes in the graph into a single matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$:

$$\mathbf{X} = \begin{bmatrix} -\mathbf{x}_1- \\ -\mathbf{x}_2- \\ \vdots \\ -\mathbf{x}_N- \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1D} \\ x_{21} & x_{22} & \dots & x_{2D} \\ & & \vdots & \\ x_{N1} & x_{N2} & \dots & x_{ND} \end{bmatrix}, \tag{5}$$

where, for instance, the scalar x_{ND} is the last feature of node N in the graph.

3. Graph Neural Networks and Message Passing

Having established the fundamental mathematical notation for describing graphs, let us delve into Graph Neural Networks (GNNs), with a particular focus on the prevalent message passing framework that underlies most GNNs. GNNs represent a category of deep learning techniques tailored for making inferences on data organized in the form of graphs. These models leverage the message passing framework, wherein pairs of nodes engage in the exchange of vector-based messages. This exchange facilitates the updating of node representations and in this way they harness inter-connections between instances as a geometric prior.

A message passing GNN layer l (ignoring edge and graph level features for simplicity here) over a graph $\mathcal{G}$ is computed as

$$\mathbf{x}_i^{(l+1)} = \phi\Big(\mathbf{x}_i^{(l)}, \bigoplus_{j \in \mathcal{N}(v_i)} \psi(\mathbf{x}_i^{(l)}, \mathbf{x}_j^{(l)})\Big), \tag{6}$$

where ψ is a message passing function, $\bigoplus$ is an aggregation function, which must be permutation-invariant (Section 3.1), e.g., the sum or max, and lastly, ϕ is a readout or update function. Both ψ and ϕ may be realised as MultiLayer Perceptrons (MLPs). However, many special cases exist which have given rise to a plethora of GNN layers. Note that, in principle, the update equation is local and it is only dependent on the one-hop neighborhood of the node. Message passing relies on pairwise communication as its fundamental building block. For clarity, Equation 6 can be further decoupled into three update rules as typically implemented in frameworks such as Pytorch Geometric:

$$\mathbf{m}_{ij}^{(l)} \leftarrow \psi(\mathbf{x}_i^{(l)}, \mathbf{x}_j^{(l)}), \tag{Message}$$

$$\mathbf{a}_i^{(l)} \leftarrow \bigoplus_{j \in \mathcal{N}(v_i)} \mathbf{m}_{ij}^{(l)}, \tag{Aggregate}$$

$$\mathbf{x}_i^{(l+1)} \leftarrow \phi\Big(\mathbf{x}_i^{(l)}, \mathbf{a}_i^{(l)}\Big). \tag{Update}$$

This localized approach enhances scalability and couples the neural network computational graph with the dataset input graph. This has proven to be a good inductive bias for many graph representation learning tasks but can also be a potential weakness since it is susceptible to over-squashing, over-smoothing, issues handling long-range dependencies, edge incompleteness, noise, and limits the expressive power of the architecture (see Section 3.3).

3.1 Permutation Invariance and Equivariance

Next, we would like to discuss two important notions in the contexts of GNNs: permutation invariance and permutation equivariance. A permutation invariant function satisfies:

$$f(\mathbf{PX}) = f(\mathbf{X}), \tag{7}$$

for all permutation matrices $\mathbf{P} \in \mathbb{R}^{N \times N}$. The intuitive idea behind permutation invariance is that, given an unordered set, we want our function to be agnostic to the order in which we feed the feature representations of each node to it. A permutation equivariant function on the other hand satisfies:

$$F(\mathbf{PX}) = \mathbf{P}F(\mathbf{X}). \tag{8}$$

That is, if we permute the input features, the output features get permuted in the same way. Alternatively, we can think of equivariance as being able to commute transformations. Let $p(\mathbf{X}) = \mathbf{PX}$, then we have permutation equivariance when

$$p \circ F = F \circ p, \tag{9}$$

where we use $\circ$ to denote composition. In the context of graphs we have a set of nodes $\mathcal{V}$, but also a set of edges $\mathcal{E}$ represented via the adjacency matrix $\mathbf{A}$, hence the permutation invariance equation for graphs becomes:

$$f(\mathbf{PX}, \mathbf{PAP}^T) = f(\mathbf{X}, \mathbf{A}), \tag{10}$$

and likewise, the permutation equivariance function formulation becomes:

$$F(\mathbf{X}, \mathbf{PAP}^T) = \mathbf{P}F(\mathbf{X}, \mathbf{A}). \tag{11}$$

In simple terms, the equations above aim to convey that when we permute the node features in a graph, we should also permute its adjacency matrix accordingly. This is achieved by multiplying the adjacency matrix with the permutation matrix and its transpose on both sides, resulting in $\mathbf{PAP}^T$. GNN layers define permutation equivariant functions on graphs by shared application of permutation-invariant message passing functions. Indeed, the same holds true when analyzing the Transformer's self-attention mechanism, but permutation invariance is disrupted by leveraging positional encodings. In NLP this is done because although the ideal connectivity between words is unknown, we assume the relative position between words is important.

3.2 Traditional Graph Neural Network Architectures

In practice, Equation 6 is implemented leveraging matrix multiplication. Note that we can convolve node features with that of its neighbours using the following operation:

$$\left(\mathbf{AX}^{(l)}\right)_i = \sum_{j \in \mathcal{N}(v_i)} \mathbf{x}_j^{(l)}, \tag{12}$$

where $(\cdot)_i$ extracts row i from the resulting matrix. To add self-loops, one can simply consider the matrix $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ instead:

$$\left(\hat{\mathbf{A}}\mathbf{X}^{(l)}\right)_i = \mathbf{x}_i^{(l)} + \sum_{j \in \mathcal{N}(v_i)} \mathbf{x}_j^{(l)}. \tag{13}$$

Based on this observation it is possible to formulate traditional GNN architectures such as Graph Convolutional Networks (GCNs) and Graph Attention Networks (GATs).

Graph Convolutional Networks. In particular, for GCNs we use the following update equation at each layer:

$$\underbrace{\mathbf{X}^{(l+1)}}_{N\times D_{out}} = \sigma\left(\underbrace{\tilde{\mathbf{A}}}_{N\times N}\underbrace{\mathbf{X}^{(l)}}_{N\times D_{in}}\underbrace{\mathbf{W}^{(l)}}_{D_{in}\times D_{out}}\right) = \sigma\left(\underbrace{\hat{\mathbf{D}}^{-1/2}}_{N\times N}\underbrace{\hat{\mathbf{A}}}_{N\times N}\underbrace{\hat{\mathbf{D}}^{-1/2}}_{N\times N}\underbrace{\mathbf{X}^{(l)}}_{N\times D_{in}}\underbrace{\mathbf{W}^{(l)}}_{D_{in}\times D_{out}}\right). \quad (14)$$

In the equation above, $\hat{\mathbf{D}} \in \mathbb{R}^{N\times N}$ represents the degree matrix of $\hat{\mathbf{A}} \in \mathbb{R}^{N\times N}$, which is used to normalize the features. This formulation recurs across various GNN architectures and in the Transformer architecture: a matrix on the left, $\tilde{\mathbf{A}}$, specifies how to aggregate representations from different nodes, while another matrix on the right, $\mathbf{W}^{(l)}$, linearly projects the features. In the case of the GCN architecture, $\tilde{\mathbf{A}}$ is not learnable. Let us look at how it aggregates features row-wise:

$$\left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}^{(l)}\right)_i = c_{ii}\mathbf{x}_i^{(l)} + \sum_{j\in\mathcal{N}(v_i)} c_{ij}\mathbf{x}_j^{(l)}\ ;\ c_{ij} = \frac{1}{\sqrt{deg(v_i)deg(v_j)}}. \quad (15)$$

Note that the degree matrix is fixed and it is a function of the input graph topology. The learnable weigths are introduced via the matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{D\times D}$ which is learned at each layer l:

$$\left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}^{(l)}\mathbf{W}^{(l)}\right)_i = \left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}^{(l)}\right)_i\mathbf{W}^{(l)} = \left(c_{ii}\mathbf{x}_i^{(l)} + \textstyle\sum_{j\in\mathcal{N}(v_i)} c_{ij}\mathbf{x}_j^{(l)}\right)\mathbf{W}^{(l)}. \quad (16)$$

From the equation above we can see that all feature vectors are multiplied by the same learnable weight matrix and that local information is introduced into the equation via the c_{ij} coefficients. Lastly, in Equation 14, σ is a non-linearity such as ReLU. Hence, we can see that GNNs behave similarly to MLPs: they use a linear projection and a non-linear activation function to update feature representations. However, instead of updating each node feature independently, they aggregate information from the one-hop neighbourhood. If we set $\mathcal{E} = \emptyset = \mathcal{N}(v_i) \rightarrow \hat{\mathbf{A}} = \mathbf{I}$, we can recover the standard MLP formulation:

$$\mathbf{x}_i^{(l+1)} = \sigma\left(\left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}^{(l)}\right)_i\mathbf{W}^{(l)}\right) = \sigma\left(c_{ii}\mathbf{x}_i^{(l)}\mathbf{W}^{(l)}\right) = \mathrm{ReLU}\left(\mathbf{x}_i^{(l)}\hat{\mathbf{W}}^{(l)}\right), \quad (17)$$

where we have merged the normalization coefficient into the $\hat{\mathbf{W}}^{(l)} \in \mathbb{R}^{D\times D}$ matrix. In this case, we neglect the bias. Additionally, for MLPs it is usual to see the weight matrix multiplied from the left in most texts, whereas in our case, it is multiplying from the right. This is simply because of the particular ordering we have chosen for rows and columns.

Graph Attention Networks. GATs can be seen as a generalization of GCNs in which we learn the coefficients multiplying each feature vector (unlike GCN in which these are a function of the graph topology):

$$\mathbf{x}_i^{(l+1)} = \sigma\left(\alpha_{ii}^{(l)}\mathbf{x}_i^{(l)}\mathbf{W}^{(l)} + \sum_{j\in\mathcal{N}(v_i)} \alpha_{ij}^{(l)}\mathbf{x}_j^{(l)}\mathbf{W}^{(l)}\right) ;\ \alpha_{ij}^{(l)} = \alpha_{ij}^{(l)}(\mathbf{X}^{(l)}). \tag{18}$$

Note that in this case, the coefficients α_{ij} are a learnable function of the input matrix $\mathbf{X}^{(l)}$. To be more precise, Equation 18 represents the update rule for attentional GNNs, of which GAT is a specific instantiation. Before delving into more mathematical details, we would like to highlight that the original GAT formulation involves linear additive self-attention. It has been stated in the literature that Transformers are Graph Attention Networks with a fully connected adjacency matrix plus positional encodings. While it is true that Transformers perform attentional message passing, as will be discussed later, it may mislead the reader to believe that the same attention mechanism and transformation are applied in Transformers as in GATs. However, this is not true: both GAT and Transformers are attentional GNNs, but they have different update rules and utilize different attention mechanisms.

Although the original GAT paper formulated its attention mechanism nodewise, in this work, we aim to present the architecture rewritten in its matrix form so that we can effectively compare all architectures from a unified perspective. The full GAT update rule[1] can be written as follows:

$$\mathbf{X}^{(l+1)} = \sigma\left(\tfrac{1}{K}\textstyle\sum_{k=1}^{K} \text{softmax}\left(\text{LeakyReLU}\left(||_N\mathbf{X}^{(l)}\mathbf{a}_1^k + \left(||_N\mathbf{X}^{(l)}\mathbf{a}_2^k\right)^T + \mathbf{B}\right)\right)\mathbf{X}^{(l)}\mathbf{W}^{(l,k)}\right). \tag{19}$$

The equation above is quite cumbersome, so we will try to break it down. The formulation about considers multiple attention heads, a total of K. For each head k we have a set of weights $\mathbf{W}^{(l,k)} \in \mathbb{R}^{D_{in}\times D_{out}}$, and vectors $\mathbf{a}_1^k, \mathbf{a}_2^k \in \mathbb{R}^{D_{in}\times 1}$. Note that $\mathbf{X}^{(l)}\mathbf{a}_1^k, \mathbf{X}^{(l)}\mathbf{a}_2^k \in \mathbb{R}^{N\times 1}$. These are concatenated N times using the operation $||_N(\cdot)$, or in PyTorch/Tensorflow language, they are broadcasted, so that $||_N\mathbf{X}^{(l)}\mathbf{a}_1^k + \left(||_N\mathbf{X}^{(l)}\mathbf{a}_2^k\right)^T \in \mathbb{R}^{N\times N}$. Lastly, $\mathbf{B} \in \mathbb{R}^{N\times N}$ is a matrix with entries:

$$B_{ij} = \begin{cases} 0 & \text{if } (v_i, v_j) \in \mathcal{E} \\ -\infty & \text{if } (v_i, v_j) \notin \mathcal{E}, \end{cases} \tag{20}$$

which forces the softmax operation to assign coefficients of zero to non-existing edges. Note that Transformers, especially the decoder part, do exactly the same trick to enforce causal interactions between the generated tokens (causal attention). Also, including the $\mathbf{B}$ matrix before the softmax operator ensures that the total probability along the existing edges adds up to one. On the contrary, if we were to compute the softmax for a fully-connected graph first and later zero out entries that are not present in the input graph, the total probability would not add up to one. For simplicity we can consider a single attention head instead and rewrite the equation for $K = 1$:

$$\underbrace{\mathbf{X}^{(l+1)}}_{N\times D_{out}} = \sigma\left(\text{softmax}\left(\text{LeakyReLU}\left(\underbrace{||_N\mathbf{X}^{(l)}\mathbf{a}_1}_{N\times N} + \underbrace{\left(||_N\mathbf{X}^{(l)}\mathbf{a}_2\right)^T}_{N\times N} + \underbrace{\mathbf{B}}_{N\times N}\right)\right)\underbrace{\mathbf{X}^{(l)}}_{N\times D_{in}}\ \underbrace{\mathbf{W}^{(l)}}_{D_{in}\times D_{out}}\right). \tag{21}$$

1. Note that we avoid some superfluous matrix multiplications present in the original paper.

We can further simplify this equation and write it in the same form as equation 14:

$$\underbrace{\mathbf{X}^{(l+1)}}_{N\times D_{out}} = \sigma\left(\underbrace{\tilde{\mathbf{A}}^{(l)}}_{N\times N}\underbrace{\mathbf{X}^{(l)}}_{N\times D_{in}}\underbrace{\mathbf{W}^{(l)}}_{D_{in}\times D_{out}}\right);\tilde{\mathbf{A}}^{(l)} = \tilde{\mathbf{A}}^{(l)}(\mathbf{X}^{(l)}). \tag{22}$$

In this case the adjacency matrix is learned and depends on the input itself. If we want to reincorporate the multihead attention under the same formulation we can simply rewrite the equation as follows:

$$\underbrace{\mathbf{X}^{(l+1)}}_{N\times D_{out}} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\underbrace{\tilde{\mathbf{A}}^{(l,k)}}_{N\times N}\underbrace{\mathbf{X}^{(l)}}_{N\times D_{in}}\underbrace{\mathbf{W}^{(l,k)}}_{D_{in}\times D_{out}}\right);\tilde{\mathbf{A}}^{(l,k)} = \tilde{\mathbf{A}}^{(l,k)}(\mathbf{X}^{(l)}). \tag{23}$$

This is somewhat similar to the Transformer self-attention mechanism but we can notice a few discrepancies: 1) The attention mechanism used to compute the adjacency matrix is different (less expressive additive attention in GATs), 2) The Transformer's self-attention mechanism does not apply a non-linearity after matrix multiplication (pointwise non-linearities are only used as part of the MLP, Section 4.1), 3) The multi-head self-attention mechanism in Transformers applies matrix multiplication to aggregate the contribution from different heads instead of taking the mean. We encourage the reader to compare the math presented here against that for the Transformer's self-attention mechanism in Section 4.2. More importantly, GATs do not use positional encodings (unlike Transformers Section 4.3), which limits their expressivity and their ability to exploit global structural information.

3.3 Expressivity and Limitations

Next, we discuss and provide an overview of the limitations of traditional message-passing GNNs, which have motivated the current growing interest in Graph Transformers. Hence, our aim is to contextualize the current transition from traditional frameworks to Transformers and global attention-based methods, rather than delving into each issue in great mathematical detail.

Graph Isomorphism. In group theory, an isomorphism is a special type of mapping between two groups that preserves the group structure. More formally, let $(G, *)$ and $(H, \cdot)$ be two groups (with their respective sets and group laws). A function $\varphi : G \to H$ is called an isomorphism if it satisfies the following properties: (I) *Bijectivity*, the function φ is bijective, meaning it is both injective (one-to-one) and surjective (onto). This ensures that every element in H is the image of exactly one element in G under φ, and vice versa. (II) *Preservation of the group operation*, for any $g_1, g_2 \in G$, the isomorphism preserves the group operation, meaning $\varphi(g_1 * g_2) = \varphi(g_1) \cdot \varphi(g_2)$. In other words, applying the group operation in G and then mapping the result using φ is the same as first mapping the elements using φ and then applying the group operation in H. If there exists an isomorphism between two groups, we say that the groups are isomorphic. In this case, the groups have essentially the same algebraic structure, even though their elements and operations may look different. In the context of graphs, an isomorphism is a concept that similarly ensures that two graphs have essentially the same structure, despite potentially having different vertex and edge labels. Just as with groups, a bijective mapping between the vertices of two graphs is considered an isomorphism if it preserves adjacency relationships. This means that if two graphs are isomorphic, they share the same connectivity patterns, even if the individual vertices are labeled differently. Formally, if $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$ and $\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ are two graphs, an

isomorphism between them is a bijective function $\varphi : \mathcal{V}_\mathcal{G} \to \mathcal{V}_\mathcal{H}$ such that for any vertices $v_i, v_j \in \mathcal{V}_\mathcal{G}$, v_i is adjacent to v_j in $\mathcal{G}$ iff $\varphi(v_i)$ is adjacent to $\varphi(v_j)$ in $\mathcal{H}$.

Local message-passing and Expressivity. The Weisfeiler-Leman (WL) algorithm is a graph isomorphism test based on vertex labeling. It iteratively refines vertex labels by considering the labels of neighboring vertices. The basic idea is to assign each vertex a multiset (a set with repeated elements) of labels representing its neighborhood structure, and then update these labels iteratively based on the multiset of labels of neighboring vertices. By repeating this process, the algorithm generates a sequence of labelings that capture increasingly refined information about the graph's structure. The WL algorithm can be applied for graph isomorphism testing by comparing the label sequences generated for each graph. If two graphs have the same label sequence after a certain number of iterations, they are considered isomorphic. However, this approach is known to fail for graphs with large automorphism groups, specific symmetries, dense graphs, or more generally when limited global structure about the graph can be inferred only from the local vertices and edges. Both the WL algorithm and GNNs operate on graph-structured data and leverage local neighborhood information. In the literature, it has been shown that the expressivity of GNNs is bounded by the WL test (although it can actually be improved using positional and structural encodings, Section 5.2) . However, it is important to note that GNNs not only discriminate between different structures but also learn to map similar graph structures to similar embeddings and capture dependencies within the data. In simple terms, GNNs aim to provide a good embedding based on the graph structure on which they operate. However, by only updating their internal representations based on local neighborhoods, the representations they can learn are limited, as there are graph structures that cannot be distinguished solely by attending to the neighborhood of each node at a time, which may result in mapping distinct graphs onto the same embedding.

Over-smoothing. The receptive field of GNNs increases as more message-passing layers are stacked together. A single (standard) message-passing layer, such as GCN and GAT layers, updates the node representation based on the one-hop neighborhood. When multiple layers of these are combined, information propagates along the graph into higher-order neighborhoods proportional to the number of layers in the network. This promotes interacting nodes having similar feature representations after this diffusion process (or propagation of information along the graph). When a substantial number of layers is combined together, it can lead to an *over-smoothing* of the feature representations of the nodes of the graph; that is, they may become indistinguishable from each other. Remember that in essence, the GNN is a graph encoder, which we often combine with, for instance, an MLP for classification or regression (similar to CNNs in CV). The MLP, in this case, would take the feature representations learned by the message-passing layers and use them to make a prediction. If all representations are too similar, they lose discriminative information and it becomes difficult for the MLP to perform its task. On the contrary, it has been shown that a majority of the attention and weights of successful pre-trained Transformer models avoid over-smoothing.

Over-squashing, and capturing long range dependencies. In a graph representation learning problem, long-range dependencies emerge when the network's output relies on interactions among representations of distant nodes within the graph. Over-squashing occurs when messages exchanged between distant nodes become distorted, a phenomenon particularly notable in the context of message passing. In this paradigm, messages must traverse the input graph and negotiate graph bottlenecks. In many real-world scenarios, as we stack multiple message-

passing GNN layers, the receptive field of nodes exponentially expands. Consequently, representations of numerous neighboring nodes must be compressed into fixed-size vectors to facilitate message propagation between distant nodes. This compression leads to excessive distortion. On the other hand, in Transformers, all nodes can directly communicate with each other, thereby eliminating this issue.

4. Transformers

Transformers were originally designed for sequence modeling, functioning over sequences composed of N tokens, with each token represented by a feature vector of dimension D. Initially, tokens denoted words, sub-words, or other text subdivisions. However, in contemporary applications, a token is broadly defined as the numeric representation of an entity. For instance, one can subdivide an image into patches or cut an audio into segments and tokenize them. Similarly, in graph representation learning, each node in the graph can be viewed as a token. This expanded concept of tokens reflects the versatility and adaptability of Transformers across various domains beyond traditional text-based sequences. Tokens are obtained using a *tokenizer*, which is a pre-processing network that maps chunks of the original input into a feature vector that can be utilized by the rest of the architecture.

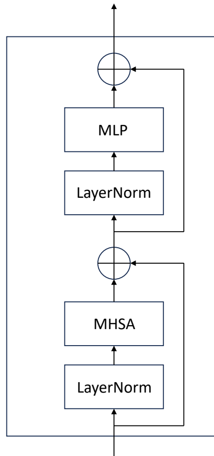

Figure 1: Transformer block schematic.

4.1 The Transformer Block

The foundation of the Transformer architecture is depicted in Figure 1. In particular, we show a single encoder Transformer block. We can summarize the operations performed by the block as follows:

$$\texttt{h = MHSA(LayerNorm(x)) + x}, \tag{24}$$

$$\texttt{y = MLP(LayerNorm(h)) + h}, \tag{25}$$

where in this particular equations we use `x, y, h` to refer to the input, output, and intermediate representations respectively. The two main components of the Transformer block are multi-head self-attention (MHSA) and a multi-layer perceptron

(MLP). These are complemented with layer normalizations (LayerNorms) and skip connections, which aid in training. Note that in this paper, we utilize the Pre-LN Transformer formulation, as it is widely accepted to be better than the original Post-LN Transformer formulation. The only difference between the two architectures is the location of the LayerNorms. Interestingly, this has been one of the only minor lasting changes to the original Transformer formulation which has proven to be incredibly robust.

Broadly speaking, the MHSA is the part of the Transformer responsible for modeling interactions between tokens, while the MLP processes each token's feature vectors independently and applies a feature-wise transformation. In other words, MHSA processes the input feature matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$ column-wise, whereas the MLP processes it row-wise. Indeed the MHSA is the part of the Transformer architecture that can be expressed as an attentional GNN. We will primarily focus on this part of the architecture since we are interested in its relation to graph representation learning and GNNs. Also, note that depending on the exact architecture, MHSA may be replaced with cross-attention or causal attention. However, for simplicity, we will not delve into such details in this text.

4.2 Self-attention

Self-attention uses the following update rule:

$$\underbrace{\mathbf{X}^{(l+1)}}_{N \times D_{out}} = \underbrace{\tilde{\mathbf{A}}^{(l)}}_{N \times N} \underbrace{\mathbf{X}^{(l)}}_{N \times D_{in}} \underbrace{\mathbf{W}_V^{(l)}}_{D_{in} \times D_{out}} ; \tilde{\mathbf{A}}^{(l)} = \tilde{\mathbf{A}}^{(l)}(\mathbf{X}^{(l)}). \tag{26}$$

Note that $\mathbf{W}_V^{(l)}$ simply performs a linear projection of the input features. In particular, the key of self-attention is $\tilde{\mathbf{A}}^{(l)}$ which defines how to convolve the input features, or in other words, how to combine the feature representations across tokens. We call this operation *self*-attention because $\tilde{\mathbf{A}}^{(l)} = \tilde{\mathbf{A}}^{(l)}(\mathbf{X}^{(l)})$ is computed as a function of the features themselves. In the context of Transformers we can think of $\tilde{\mathbf{A}}^{(l)}$ as the combination of two matrices:

$$\tilde{\mathbf{A}}^{(l)}(\mathbf{X}^{(l)}) = \mathbf{W}_{\mathcal{E}}^{(l)} \odot \mathbf{A} = \mathbf{W}_{\mathcal{E}}^{(l)} \odot \mathbf{J}_N = \mathbf{W}_{\mathcal{E}}^{(l)}(\mathbf{X}^{(l)}), \tag{27}$$

where $\mathbf{J}_N \in \mathbb{R}^{N \times N}$ is a square matrix full of 1s, that is, it represents a fully connected graph with self-loops and $\odot$ is the Hadamard (elementwise) product. While Equation 27 may appear somewhat redundant, our intention is to emphasize that the Transformer architecture, akin to conventional GNNs, utilizes the same adjacency matrix across all its layers. However, in this instance, the adjacency matrix lacks a sense of locality; instead, it represents a fully-connected graph. Locality, in this context, is acquired through the $\mathbf{W}_{\mathcal{E}}^{(l)} \in \mathbb{R}^{N \times N}$ edge weight matrix. Consequently, the Transformer autonomously discovers the graph's connectivity structure, effectively engaging in latent graph inference (or graph structure learning).

Technically, the Transformer operates within the message-passing framework, albeit conducting computations over a fully-connected graph. The classification of this process as message-passing becomes a matter of interpretation and delves into a semantic discussion. In this scenario, there is a debate about whether we should still characterize it as message-passing, as we are not actively leveraging any pre-existing local structure. Instead, the Transformer discovers this structure by examining the entire graph simultaneously. This helps prevent information bottlenecks, but can be sensitive to initialization.

Next, in the interest of clarity, let us try to rewrite Equation 26 in terms of the standard key-query-value formulation of Transformers. Firstly, the key component of the self-attention mechanism is the cross-token interaction due to the inner product computation between keys and queries:

$$\tilde{\mathbf{A}}^{(l)} = \mathbf{W}_{\mathcal{E}}^{(l)} = \text{softmax}\left(\frac{\mathbf{Q}^{(l)}(\mathbf{K}^{(l)})^T}{\sqrt{D_K}}\right) = \text{softmax}\left(\frac{\mathbf{X}^{(l)}\mathbf{W}_Q^{(l)}(\mathbf{X}^{(l)}\mathbf{W}_K^{(l)})^T}{\sqrt{D_K}}\right). \quad (28)$$

In particular the equation above describes scaled dot-product self-attention, where key $\mathbf{K}$ and $\mathbf{Q}$ are linear projections of the input features, obtained by multiplying the input matrix by $\mathbf{W}_K^{(l)} \in \mathbb{R}^{D_{in}\times D_K}$ and $\mathbf{W}_Q^{(l)} \in \mathbb{R}^{D_{in}\times D_K}$, respectively. Regarding the normalization constant $\sqrt{D_K}$, if we assume the components of the keys and queries are independent random variables with a mean of 0 and a variance of 1, the inner product will have a variance of D_K. Hence, the normalization, which preserves a variance of 1, is desirable since we do not want the softmax to saturate as D_K increases. Also, note that technically when comparing this attention mechanism with GATs, in the case of the Transformer, $\tilde{A}_{ij} \geq 0 \, \forall i, j$, whereas for GATs, $\tilde{A}_{ij} = 0 \, \forall i, j \notin \mathcal{E}$. Lastly, the value matrix would correspond to $\mathbf{V}^{(l)} = \mathbf{X}^{(l)}\mathbf{W}_V^{(l)}$. Hence, we can rewrite Equation 26 in terms of the key, query, and value matrices:

$$\mathbf{X}^{(l+1)} = \tilde{\mathbf{A}}^{(l)}\mathbf{V}^{(l)}; \tilde{\mathbf{A}}^{(l)} = \tilde{\mathbf{A}}^{(l)}(\mathbf{Q}^{(l)}, \mathbf{K}^{(l)}). \quad (29)$$

Transformers incorporate multiple attention heads in their architecture to enhance the model's effectiveness in processing input sequences. This design choice enables the model to capture diverse patterns and relationships within the data, makes it robust to initialization and is easily parallelizable in GPU. In particular, each attention head has its own set of parameters for query, key, and value transformations. The outputs are then concatenated and linearly transformed to produce the final output. Let us denote the number of attention heads as K. Given $\mathbf{Q}^{(l)}$, $\mathbf{K}^{(l)}$, and $\mathbf{V}^{(l)}$ are the query, key, and value matrices at layer l, respectively, for multi-head self-attention, we introduce the following transformations:

$$\mathbf{Q}^{(l,k)} = \mathbf{X}^{(l)}\mathbf{W}_Q^{(l,k)}; \quad \mathbf{K}^{(l,k)} = \mathbf{X}^{(l)}\mathbf{W}_K^{(l,k)}; \quad \mathbf{V}^{(l,k)} = \mathbf{X}^{(l)}\mathbf{W}_V^{(l,k)},$$

where k ranges from 1 to K, and $\mathbf{W}_Q^{(l,k)}$, $\mathbf{W}_K^{(l,k)}$, and $\mathbf{W}_V^{(l,k)}$ are learnable parameter matrices specific to each attention head. Then, for each head k, the attention scores, $\tilde{\mathbf{A}}^{(l,k)}$, are calculated using $\mathbf{Q}^{(l,k)}$ and $\mathbf{K}^{(l,k)}$:

$$\tilde{\mathbf{A}}^{(l,k)} = \text{softmax}\left(\frac{\mathbf{Q}^{(l,k)}(\mathbf{K}^{(l,k)})^T}{\sqrt{D_k}}\right). \quad (30)$$

The output for each attention head is:

$$\mathbf{H}^{(l,k)} = \tilde{\mathbf{A}}^{(l,k)}\mathbf{V}^{(l,k)}. \quad (31)$$

Next, the outputs of all attention heads are concatenated[2] and projected:

$$\underbrace{\mathbf{X}^{(l+1)}}_{N\times D_{out}} = \underbrace{(||_{k=1}^{K}\mathbf{H}^{(l,k)})}_{N\times KD_{out}} \underbrace{\mathbf{W}_O^{(l)}}_{KD_{out}\times D_{out}}, \quad (32)$$

where $\mathbf{W}_O^{(l)} \in \mathbb{R}^{KD_{out}\times D_{out}}$ is also learnable.

2. Note that technically this concatenation is slightly different to the one we used in Equation 21. In Equation 21 we use concatenation to denote broadcasting, whereas here we stack together the feature representations from all heads $\mathbf{H}^{(l,k)} \in \mathbb{R}^{N\times D_{out}}$ into feature vectors of size KD_{out}.

4.3 Positional Encodings

Positional encodings (or embeddings) play a crucial role in overcoming permutation invariance within the Transformer architecture, as discussed in Section 3.1. Their primary function is to provide the model with essential information about the relative positions of tokens. In contrast to recurrent neural networks (RNNs), Transformers lack inherent mechanisms for capturing the sequential order of input (time) data. Instead, they employ the self-attention mechanism to discern the significance of various segments within the input sequence. To integrate positional information into the sequence without resorting to the recursive nature of RNNs, positional encodings are added on top of the tokenized inputs.

By sidestepping information over-squashing (Section 3.3, one can view RNNs as performing message passing over a path graph), Transformers equipped with positional encodings offer a more efficient and effective approach to handling sequential data. This departure from the recurrence method not only leads to a substantial acceleration in training time but also holds the promise of theoretically capturing longer dependencies within a sentence since all nodes in the graph can directly communicaty with each other. The widely adopted positional encoding relies on sine and cosine functions:

$$\mathrm{PE}(p, 2d) = \sin\left(\frac{p}{10000^{(2d/D)}}\right), \tag{33}$$

$$\mathrm{PE}(p, 2d+1) = \cos\left(\frac{p}{10000^{(2d/D)}}\right). \tag{34}$$

In this context, p represents the position of a token in the sequence, d denotes the dimension of the positional encoding (feature vector entry), and D signifies the total number of dimensions of the model input. The positional encodings are subsequently combined element-wise with the input embeddings. The resulting embeddings encompass both token and positional information and are then input into the Transformer model. Notably, the sine and cosine functions are employed for encoding due to their advantageous characteristics, such as periodicity, smoothness, bounded amplitude, ease of generalization to arbitrary sequence lengths, and empirical success, among other reasons.

4.4 Skip connections, Layer Normalization, and other details

As previously mentioned, the focus of this document is to compare message passing GNNs to the Transformer architecture. The main resemblance between the two lies in terms of the MHSA mechanism, which leverages a fully-connected graph for direct communication between nodes. However, the Transformer block has other important components that primarily aid in training and stability, and these are indeed of paramount importance to the success of the architecture. These are also reused by Graph Transformers in Section 5.

Skip Connections. These connections allow the information from earlier layers to be directly passed to later layers, helping to mitigate the vanishing gradient problem and enabling easier flow of information through the network. Let us denote the input to a layer as x, the output of the layer as $l(x)$, and the input to the next layer as z. A skip connection allows us to add the input x directly to the output of the layer. Mathematically, a skip connection can be represented as: $z = l(x) + x$. During the backpropagation algorithm, gradients are calculated with respect to the parameters of the network using the chain rule. Let us denote the gradient of the loss function with respect to the parameters $W^{(l)}$ of a layer l as $\frac{\partial \mathcal{L}}{\partial W^{(l)}}$. Now,

consider a deep neural network with L layers. When we calculate gradients through backpropagation, they are successively multiplied. The gradient at layer l can be expressed as:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(L)}} \cdot \ldots \cdot \frac{\partial z^{(l+1)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}. \tag{35}$$

If the absolute value of $|\frac{\partial z^{(l+1)}}{\partial z^{(l)}}| < 1$ for all layers, the gradients will tend to diminish exponentially as they are propagated backward, leading to the vanishing gradient problem and the network will not be able to learn, that is, its weights will not be updated. In the extreme case, if for instance, $\frac{\partial z^{(l+1)}}{\partial z^{(l)}} = 0 \rightarrow \frac{\partial \mathcal{L}}{\partial W^{(l)}} = 0$. On the contrary, even if we have dead neurons and gradient does not flow through a particular layer, with skip connections we can still guarantee the gradient flowing to previous layers:

$$\frac{\partial z^{(l+1)}}{\partial z^{(l)}} = \frac{\partial (l(z^{(l)}) + z^{(l)})}{\partial z^{(l)}} = \frac{\partial l(z^{(l)})}{\partial z^{(l)}} + \frac{\partial z^{(l)}}{\partial z^{(l)}} = 0 + 1 \rightarrow \frac{\partial \mathcal{L}}{\partial W^{(l)}} \neq 0. \tag{36}$$

Skip connections can also offer other more intricate benefits in terms of expressivity, which require more cumbersome mathematical analysis and are beyond the scope of this text. For instance, they can enable GCNs to perform well on heterophilic graphs.

Layer Normalization. Layer normalization (LayerNorm) is a crucial technique in neural networks, especially within Transformer architectures, enhancing training stability and convergence speed. It operates by normalizing neuron activations within each layer independently. Mathematically, LayerNorm computes mean μ and standard deviation σ across activations:

$$\mu_i = \frac{1}{D} \sum_{d=1}^{D} x_{id}, \tag{37}$$

$$\sigma_i = \sqrt{\frac{1}{D} \sum_{d=1}^{D} (x_{id} - \mu_i)^2}, \tag{38}$$

which for consistency with previous notation would be applied to $\mathbf{X} \in \mathbb{R}^{N \times D}$ row-wise $(\mathbf{X})_i = x_i = \{x_{i1}, x_{i2}, ..., x_{iD}\}$. Then, it normalizes activations element-wise and LayerNorm introduces learnable parameters γ_i and β_i for scaling and shifting normalized activations.

$$\hat{x}_{id} = \frac{x_{id} - \mu_i}{\sigma_i},$$

$$y_{id} = \gamma_i \hat{x}_{id} + \beta_i.$$

These parameters, learned via backpropagation, offer flexibility in normalization and representation. In summary, the LayerNorm operation is formulated as:

$$\text{LayerNorm}(x_i) = \gamma_i \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta_i, \tag{39}$$

where $\epsilon > 0$ ensures numerical stability.

Both batch normalization (BatchNorm) and LayerNorm are commonly used techniques for normalization. BatchNorm calculates statistics across instances in a batch, while LayerNorm calculates statistics across the feature dimension. The choice between BatchNorm and LayerNorm often depends on the task, with BatchNorm being more common in CV tasks and the latter in NLP. One reason for the preference of LayerNorm in NLP tasks is the variability in sentence lengths, which makes it difficult to determine the appropriate normalization constant for BatchNorm. This variability can lead to instability during training due to fluctuations in the normalization constants across batches. Additionally, from a legacy perspective, in RNN architectures, which were commonly used in NLP before the rise of Transformers, it was also challenging to compute normalization statistics, making LayerNorm a more straightforward choice. It is worth noting, however, that LayerNorm remains a standard choice even in CV applications when using Vision Transformers (ViTs).

5. Graph Transformers

Graph Transformers represent an adaptation of conventional Transformers to graph representation learning. In particular, Graph Transformers leverage the input graph structure in the form of a soft inductive bias while retaining the global attention mechanism of traditional Transformers. Additionally, similar to traditional Transformers, they employ encodings, providing the network with global structural information and aiding expressivity.

In the realm of NLP, Transformers embrace full attention when constructing feature representations for words, treating sentences as fully connected graphs. This approach is rationalized by two primary considerations. Firstly, discerning meaningful sparse interactions or connections among words within a sentence proves challenging. The interdependence of words can fluctuate with context, user perspective, and the specific application. Given the myriad plausible connections among words, text datasets often lack explicit word interactions, justifying the adoption of full attention. In this way, the model autonomously determines the dependencies between words. Secondly, the fully connected graph paradigm in NLP typically involves graphs (context windows) with a modest number of nodes compared to graph representation learning tasks. This scale allows for computationally feasible attention in terms of both memory and time requirements, a characteristic that does not necessarily hold true for Graph Transformers.

On the other hand, real-world graph datasets exhibit arbitrary connectivity structures shaped by the application domain, with node sizes reaching up to millions or even billions. This diversity presents a wealth of information for neural network learning but renders the utilization of a fully connected graph impractical. Despite the computational complexities inherent in Graph Transformers, they offer a silver lining by enabling the modeling of long-range dependencies and interactions between nodes. This is achieved through the inherent construction that allows every node in the graph to communicate with all others (referred to as *global attention*, as opposed to *local attention* in GATs). Furthermore, Graph Transformers address prevalent shortcomings of traditional message passing GNNs, including over-smoothing, over-squashing, and limited expressivity. These issues are indeed reminiscent of prevalent issues in the pre-Transformer era in NLP, such as vanishing gradients and difficulties faced by more traditional models like RNNs in capturing long-range temporal dependencies.

5.1 The Graph Transformer Block

Unlike the conventional Transformer, Graph Transformer architectures have not yet converged to a widely accepted design. However, the trend appears to favor combining some form of less expensive global attention, instead of the fully connected scaled dot-product self-attention, with traditional message-passing GNNs.

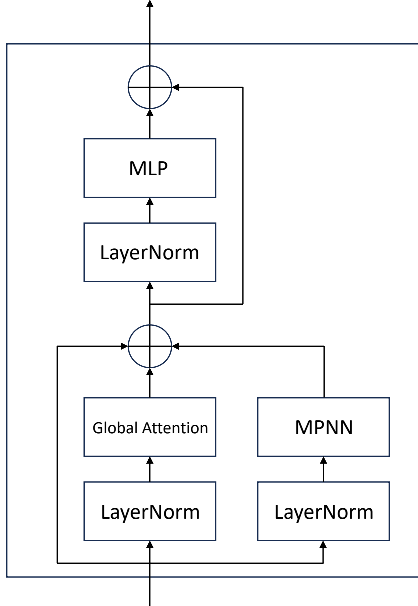

Figure 2: Graph Transformer block schematic. Other implementations use BatchNorm, which may also be placed after the Global Attention and the MPNN layer. Here, we attempt to modify the original Transformer block as minimally as possible for clarity.

In Figure 2 we depict a simplified schematic of the Graph Transformer block. We compute global attention over all tokens (or nodes), and local message passing leveraging the input graph using a message passing network (MPNN) layer[3]. Note that both operations occur in parallel, and the network can use the normalizations to gate the signal for each computation. We can summarize the block as follows:

$$\texttt{h = GlobalAttention(LayerNorm(x)) + MPNN(LayerNorm(x,A)) + x}, \tag{40}$$

$$\texttt{y = MLP(LayerNorm(h)) + h}. \tag{41}$$

3. Here we use the term MPNN instead of GNN to emphasize that we operate over the local graph structure in line with traditional architectures. The exact naming convention seems to slightly differ between different works in the literature. Here we use GNNs to refer to any neural network that operates on graphs, MPNN for GNNs that perform message passing over local neighbourhoods based on an input adjacency matrix, and Graph Tranformers for GNNs that are based on the Transformer architecture.

We use `x, y, h, A` to refer to the input, output, intermediate representation, and the dataset adjacency matrix. Typically, if the data also contains information such as edge attributes these are only passed to the MPNN.

In summary, broadly speaking, in Graph Transformers we absorb traditional message passing GNN architectures into the Transformer block, substitute the global attention mechanism with computationally less expensive alternatives to scale to larger graphs, and lastly, we use graph positional and structural encodings, which we will discuss next.

5.2 Graph Positional and Structural Encodings

In traditional Transformer models, positional encodings are added to the input embeddings to provide the model with information about the order of tokens in the sequence. However, in graph Transformers, the notion of positional encoding is adapted to suit the graph structure of the input data. In graph-structured data, nodes do not have an inherent order like tokens in a sequence. Therefore, traditional positional encodings based on absolute or relative positions are not directly applicable.

In recent literature, various encoding methods have been explored, yet a consensus remains elusive. Nevertheless, encodings can be broadly classified into two categories: positional and structural. Positional encodings aim to convey the spatial positioning of a node within a graph. Consequently, nodes that are proximal within a graph or subgraph should have similar positional encodings. Conversely, structural encodings are designed to encapsulate the structural attributes of graphs or subgraphs, enhancing the network's expressiveness and generalizability. Therefore, nodes sharing analogous subgraphs or graphs should exhibit close structural encodings. Moreover, encodings can be categorized as local, global, or relative. Local encodings capture cluster and neighborhood information, typically applied atop node features. Global encodings provide overarching structural insights to the network as graph-level features. Lastly, relative encodings, such as distances between nodes computed as edge features, are commonly utilized in MPNN layers.

Laplace Eigenfunction Encodings. Next, we will discuss Laplace eigenfunctions, as they represent the most direct translation of the positional encodings discussed in Section 4.3 to graphs. While sinusoids serve well for linear sequences, their application becomes ambiguous for arbitrary graphs lacking a clear positional axis. Instead, in graph theory, the counterparts to sine and cosine functions are represented by the eigenvectors, denoted as Φ, of the graph Laplacian matrix, $\mathbf{L}$. In Euclidean space, the Laplacian operator corresponds to the divergence of the gradient, and its eigenfunctions manifest as sine and cosine functions. Thus, within the graph domain, the eigenvectors of the graph Laplacian naturally assume the role of sine functions, offering a meaningful analogy for positional encoding. Another property of eigenvalues is their ability to discern between various graph structures and sub-structures, akin to interpreting them as the frequencies of resonance within the graph. As discussed in Section 2.1, the graph Laplacian $\mathbf{L}$ is defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}$. To find its eigenvalues and eigenvectors one must solve the eigenvalue problem:

$$\mathbf{L}\Phi = \lambda\Phi, \tag{42}$$

where $\lambda \in \mathbb{R}$ represents an eigenvalue and $\Phi \in \mathbb{R}^{N\times 1}$ its corresponding eigenvector, N being the total number of nodes and assuming the Laplacian to be symmetric (in line with Section 2.1). Based on this computation we can obtain a number of eigenvectors, M, $||_{m=1}^{M}\Phi_{im} \in \mathbb{R}^{N\times M}$. With each entry $\Phi_i \in \mathbb{R}^{1\times M}$ corresponding to

node v_i in the graph. These features can be used to compute for instance, relative encodings between nodes v_i and v_j in the graph $d(v_i, v_j) = d(\Phi_i, \Phi_j)$. The number of eigenvalues and eigenvectors of a graph is closely related to its size, connectivity, and overall topology.

5.3 Global Attention and Scaling to Large Graphs

The global all-pair attention mechanism in Transformers imposes limitations on scalability due to its quadratic time and space complexity relative to the graph's node count. With each added layer, the computational graph grows exponentially, making training deep Transformers on extensive graphs with millions of nodes highly resource-intensive. To address this challenge, strategies such as partitioning interconnected nodes into smaller mini-batches are employed to alleviate the computational demands.

In the literature, researchers have worked on importing ideas from linear Transformers into Graph Transformer and graph representation learning, such as the attention mechanisms presented in Linformer, Performer, or Big Bird, to name a few. These often use row-rank approximations, avoid softmax attention, or employ sparsified or randomized attention, among other approaches. Most of these variations of the Transformer approximate the all-pair attention with a linear complexity operation, but they are strictly less expressive than traditional attention. In practice, in NLP these approximations of attention have been challenged by the (exact) FlashAttention operation, which re-implements the original attention mechanism but is particularly optimized at the GPU level. Nevertheless such approximations are still relevant for graphs, given the large number of nodes at hand. Several other papers in the graph representation domain have explored computing global attention over alternative graphs. For instance, one can introduce a global virtual node connected to all other nodes in the graph and compute attention with respect to it, so that the operation is linear in the number of nodes and all nodes are at a 2-hop distance from each other. Other options include leveraging expander graphs, which try to maximize connectivity while remaining as sparse as possible. Concurrently, some scalable architectures have questioned the need for deep attention itself and argued that a single global attention layer working in parallel with a deep MPNN is enough to scale to larger graphs.

Additionally, concerning generalization, in tasks involving small graphs, such as inductive molecular property prediction at the graph level, where labeled instances are abundant, highly-parameterized Transformer-based architectures may have adequate supervision for generalization. However, in tasks with large graphs, like transductive large social network classification or protein prediction at the node level, where there is typically only one large graph available and labeled nodes are limited, large Graph Transformers are prone to overfitting due to insufficient supervision.

Acknowledgements

I would like to thank in no particular order Petar Veličković, Richard E. Turner, Anastasis Kratsios, Marc T. Law, and Xiaowen Dong for insightful discussions.

References

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

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Ua6zuk0WRH.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=H4DqfPSibmx.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

Gbètondji J-S Dovonon, Michael M. Bronstein, and Matt J. Kusner. Setting the record straight on transformer oversmoothing, 2024.

Vijay Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs, 12 2020.

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020. URL https://arxiv.org/abs/2006.16236.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Devin Kreuzer, Dominique Beaini, William L. Hamilton, Vincent Létourneau, and Prudencio Tossou. Rethinking graph transformers with spectral attention. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=huAdB-Tj4yG.

William Peebles and Saining Xie. Scalable diffusion models with transformers. *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023.

Jason Phang, Yao Zhao, and Peter J Liu. Investigating efficiently extending transformers for long input summarization. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://openreview.net/forum?id=kQSlGF9lH6.

Ladislav Rampasek, Mikhail Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=lMMaNf6oxKM.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20 (1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J. Sutherland, and Ali Kemal Sinop. Exphormer: Scaling graph transformers with expander graphs, 2023. URL https://openreview.net/forum?id=8Tr3v4ueNd7.

Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M. Bronstein. Understanding over-squashing and bottlenecks on graphs via curvature. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=7UmjRGzp-A.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJXMpikCZ.

Qitian Wu, Wentao Zhao, Zenan Li, David Wipf, and Junchi Yan. Nodeformer: A scalable graph structure learning transformer for node classification. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=sMezXGG5So.

Qitian Wu, Wentao Zhao, Chenxiao Yang, Hengrui Zhang, Fan Nie, Haitian Jiang, Yatao Bian, and Junchi Yan. Simplifying and empowering transformers for large-graph representations. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=R4xpvDTWkV.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture, 2020.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=OeWooOxFwDa.