Graph transformers (GTs) have emerged as a promising architecture that is theoretically more expressive than message-passing graph neural networks (GNNs). However, typical GT models have at least quadratic complexity and thus cannot scale to large graphs. While there are several linear GTs recently proposed, they still lag behind GNN counterparts on several popular graph datasets, which poses a critical concern on their practical expressivity. To balance the trade-off between expressivity and scalability of GTs, we propose Polynormer, a polynomial-expressive GT model with linear complexity. Polynormer is built upon a novel base model that learns a high-degree polynomial on input features. To enable the base model permutation equivariant, we integrate it with graph topology and node features separately, resulting in local and global equivariant attention models. Consequently, Polynormer adopts a linear local-to-global attention scheme to learn high-degree equivariant polynomials whose coefficients are controlled by attention scores. Polynormer has been evaluated on 13 homophilic and heterophilic datasets, including large graphs with millions of nodes. Our extensive experiment results show that Polynormer outperforms state-of-the-art GNN and GT baselines on most datasets, even without the use of nonlinear activation functions. Source code of Polynormer is freely available at: github.com/cornell-zhang/Polynormer.
fortunately, both prior work [Platonov et al., 2023] and our empirical results indicate that those linear GT models still underperform state-of-the-art GNN counterparts on several popular datasets, which poses a serious concern regarding the practical advantages of linear GTs over GNNs.

In this work, we provide an orthogonal way to ease the tension between expressivity and scalability of GTs. Specifically, we propose Polynormer, a linear GT model that is polynomial-expressive: an $L$-layer Polynormer can expressively represent a polynomial of degree $2^L$, which maps input node features to output node representations and is equivariant to node permutations. Note that the polynomial expressivity is well motivated by the Weierstrass theorem which guarantees any smooth function can be approximated by a polynomial [Stone, 1948]. To this end, we first introduce a base attention model that explicitly learns a polynomial function whose coefficients are determined by the attention scores among nodes. By imposing the permutation equivariance constraint on polynomial coefficients based on graph topology and node features separately, we derive local and global attention models respectively from the base model. Subsequently, Polynormer adopts a linear local-to-global attention paradigm for learning node representations, which is a common practice for efficient transformers in language and vision domains yet less explored on graphs. To demonstrate the efficacy of Polynormer, we conduct extensive experiments by comparing Polynormer against 22 competitive GNNs and GTs on 13 node classification datasets that include homophilic and heterophilic graphs with up to millions of nodes. We believe this is possibly one of the most extensive comparisons in literature. Our main technical contributions are summarized as follows:

- To the best of our knowledge, we are the first to propose a polynomial-expressive graph transformer, which is achieved by introducing a novel attention model that explicitly learns a high-degree polynomial function with its coefficients controlled by attention scores.

- By integrating graph topology and node features into polynomial coefficients separately, we derive local and global equivariant attention modules. As a result, Polynormer harnesses the local-to-global attention mechanism to learn polynomials that are equivariant to node permutations.

- Owing to the high polynomial expressivity, Polynormer without any activation function is able to surpass state-of-the-art GNN and GT baselines on multiple datasets. When further combined with ReLU activation, Polynormer improves accuracy over those baselines by a margin of up to 4.06% across 11 out of 13 node classification datasets, including both homophilic and heterophilic graphs.

- Through circumventing the computation of dense attention matrices and the expensive PE/SE methods used by prior arts, our local-to-global attention scheme has linear complexity in regard to the graph size. This renders Polynormer scalable to large graphs with millions of nodes.

2 BACKGROUND

There is an active body of research on GTs and polynomial networks, from which we draw inspiration to build a polynomial-expressive GT with linear complexity. In the following, we present preliminaries for both areas and provide an overview of their related work.

**Graph transformers** exploit the Transformer architecture [Vaswani et al. 2017] on graphs. Specifically, given an $n$-node graph $G$ and its node feature matrix $X \in \mathbb{R}^{n \times d}$, where $d$ represents the node feature dimension, a GT layer first projects $X$ into query, key, and value matrices, i.e., $Q = XW_Q$, $K = XW_K$, $V = XW_V$, where $W_Q$, $W_K$, $W_V \in \mathbb{R}^{d \times d}$ are three trainable weight matrices. Subsequently, the output $X'$ with self-attention is calculated as:

$$S = \frac{QK^T}{\sqrt{d}}, \ X' = \text{softmax}(S)V$$  \hspace{1cm} (1)$$

where $S \in \mathbb{R}^{n \times n}$ is the self-attention matrix. For simplicity of illustration, we assume query, key, and value have the same dimension and only consider the single-head self-attention without bias terms. The extension to the multi-head attention is standard and straightforward.

As Equation [1] completely ignores the graph topology, various PE/SE methods have been proposed to incorporate the critical graph structural information into GTs. Concretely, [Kreuzer et al., 2021]; [Dwivedi et al., 2022]; [Rampasek et al., 2022]; [Bo et al., 2023] use the top-$k$ Laplacian eigenpairs as node PEs, while requiring nontrivial computational costs to learn the sign ambiguity of Laplacian eigenvectors. Similarly, SE methods also suffer from high complexity for computing the distance of all node pairs or sampling graph substructures [Ying et al., 2021] [Chen et al., 2022a] [Zhao et al.].
Polynomial networks aim to learn a function approximator where each element of the output is expressed as a polynomial of the input features. Formally, we denote the node-m vector product of a tensor $T \in \mathbb{R}^{l_1 \times l_2 \times \cdots \times l_m}$ with a vector $u \in \mathbb{R}^{l_n}$ by $T \times_m u$. Given the input feature $x \in \mathbb{R}^n$ and the output $y \in \mathbb{R}^o$, polynomial networks learn a polynomial $P : \mathbb{R}^n \rightarrow \mathbb{R}^o$ with degree $R \in \mathbb{N}$:

$$y_j = P(x)_j = b_j + \{w_j^{[1]} T x + x T w_j^{[2]} x + w_j^{[3]} x \times_1 x \times_2 x \times_3 x + \cdots + w_j^{[R]} \times_r x\}$$

(2)

where $b_j \in \mathbb{R}$ and $\{w_j^{[1]} \in \mathbb{R}^{n \times n \times \cdots \times n}\}$ are learnable parameters for the $j$-th element of output $y$. Note that Equation 2 can be naturally extended to a more general case where both the input and output of the polynomial $P$ are matrices or higher-order tensors. Moreover, suppose $x$ and $y$ have the same dimension (e.g., $x, y \in \mathbb{R}^n$), we say a polynomial $P$ is permutation equivariant if for any permutation $\alpha$ of the indices $\{1, 2, \cdots, n\}$, the following property holds:

$$P(\alpha \cdot x) = \alpha \cdot P(x) = \alpha \cdot y$$

(3)

Note that when $x \in \mathbb{R}^n$ represents node features for an $n$-node graph (i.e., each node feature has a scalar value), the aforementioned equivariance property essentially indicates the polynomial $P$ is equivariant to node permutations. It is worth mentioning that the polynomial networks are fundamentally distinct from the polynomial graph filtering methods widely explored in spectral-based GNNs, which purely focus on the polynomials of graph shift operators rather than node features.

In the literature, the notion of learnable polynomial functions can be traced back to the group method of data handling (GMDH), which learns partial descriptors that capture quadratic correlations between specific pairs of input features (Ivakhnenko, 1971). Later, the pi-sigma network (Shin & Ghosh, 1991) and its extensions (Voutilainen et al., 2003; Li, 2003) have been proposed to capture higher-order feature interactions, while they struggle with scaling to high-dimensional input features. Recently, Chrysos et al. (2020) introduce P-nets that utilize a special kind of skip connections to efficiently implement the polynomial expansion with high-dimensional features. In addition, Chrysos et al. (2022) express as polynomials a collection of popular neural networks, such as AlexNet (Krizhevsky et al., 2012), ResNet (He et al., 2016), and SENet (Hu et al., 2018), and improve their polynomial expressivity accordingly. While these approaches have demonstrated promising results on image and audio classification tasks, they are not directly applicable to graphs.

To learn polynomials on graph-structured data, Maron et al. (2018) first introduce the basis of constant and linear functions on graphs that are equivariant to node permutations. Subsequently, a series of expressive graph models have been developed by leveraging polynomial functions (Maron et al., 2019; Chen et al., 2019; Azizian & Lelarge, 2020; Hua et al., 2022). More recently, Puny et al. (2023) demonstrate that the polynomial expressivity is a finer-grained measure than the traditional Weisfeiler-Lehman (WL) hierarchy for assessing the expressive power of graph learning models. Besides, they devise a graph polynomial model that achieves strictly better than 3-WL expressive power with quadratic complexity. However, these prior studies either only consider polynomials on local structures or cannot scale to large graphs due to their high complexity. In contrast, our work learns a high-degree polynomial on node features while integrating graph topology into polynomial coefficients. As a result, the learned polynomial function captures both local and global structural information with linear complexity, rendering it applicable to large-scale graphs.

3 Methodology

In this work, we follow the common setting that there are a graph $G = (V, E)$ and its node feature matrix $X \in \mathbb{R}^{n \times d}$, where $n$ and $d$ denote the number of nodes and node feature dimension, respectively. Our goal is to design a polynomial-expressive GT model $F$ that produces node representations $Y = F(G, X) = P_{G}(X)$, where $P_{G}$ is a high-degree polynomial on $X$ whose learnable
Given the input node features, we theoretically analyze its polynomial expressivity. Motivated by the above example, we proceed to establish a base model using the following definition:

**Definition 3.1.** Given an $n$-node graph with node features $X \in \mathbb{R}^{n \times d}$, a model $\mathcal{P} : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{n \times d}$ is $r$-polynomial-expressive if for any node $i$ and degree-$(r-1)$ monomial $M^{r-1}$ formed by rows in $X$, $\mathcal{P}$ can be parameterized such that $\mathcal{P}(X)_i = X_{i} \odot M^{r-1}$, where $\odot$ denotes the Hadamard product.

**Polynomial expressivity of prior models.** For typical message-passing GNN models, we can unify their convolution layer as: $\mathcal{P}(X)_i = \sum_j c_{i,j} X_j$, where $c_{i,j}$ is the edge weight between nodes $i$ and $j$. As each output is a linear combination of input node features, these models are at most $1$-polynomial-expressive. Hence, they mainly rely on the activation function to capture nonlinearity, instead of explicitly learning high-degree polynomials. In regard to GTs and high-order GNNs (e.g., gating-based GNNs), they only capture a subset of all possible monomials with a certain degree, resulting in their limited polynomial expressivity, as discussed in Appendix C.

**A motivating example.** Before constructing a polynomial-expressive model, let us first examine a simplified scenario where node features $x \in \mathbb{R}^n$, i.e., each node feature is a scalar. In this context, we consider a model $\mathcal{P}$ that outputs $y = \mathcal{P}(x) = (W x) \odot (x + b)$, where $W \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$ are weight matrices. Figure 1 shows that this model $\mathcal{P}$ is able to represent degree-2 polynomials, which consist of all possible monomials of degree up to 2 (except the constant term). Besides, as $W$ controls the coefficients of quadratic monomials (i.e., $x_1 x_2$, $x_1 x_3$, and $x_2 x_3$), we can interpret $W$ as a general attention matrix, where $W_{i,j}$ represents the importance of node $j$ to node $i$.

Motivated by the above example, we proceed to establish a base model using the following definition: we then theoretically analyze its polynomial expressivity.

**Definition 3.2.** Given the input node features $X^{(l)} \in \mathbb{R}^{n \times d}$ and trainable weight matrices $W \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times d}$, a model $\mathcal{P}$ is defined as the base model if its $l$-th layer is computed as:

$$X^{(l)} = (W^{(l)} X^{(l-1)}) \odot (X^{(l-1)} + B^{(l)})$$

**Theorem 3.3.** An $L$-layer base model $\mathcal{P}$ is $2^L$-polynomial-expressive.

The complete proof for Theorem 3.3 is available in Appendix A. Theorem 3.3 shows that the polynomial expressivity of the base model increases exponentially with the number of layers. Additionally, Appendices A and B reveal that $W$ and $B$ control the learned polynomial coefficients, which can be viewed as attention scores for measuring the importance of different node feature interactions.

**Limitations of the base model.** As the matrices $W$ and $B$ fail to exploit any graph inductive bias, the base model learns polynomials that are not equivariant to node permutations. Besides, the model has quadratic complexity due to the $n^2$ size of $W$ and thus cannot scale to large graphs. To address both limitations, we derive two equivariant models with linear attention in Section 3.2.
3.2 Equivariant Attention Models with Polynomial Expressivity

For clarity, we omit the layer index \( l \) in the ensuing discussion unless it is explicitly referenced. Instead of learning the non-equivariant matrix \( B \in \mathbb{R}^{n \times d} \) in Equation 4, we replace it with learnable weights \( \beta \in \mathbb{R}^{d} \) for each node, i.e., \( B = 1 \beta^T \), where \( 1 \in \mathbb{R}^{n} \) denotes the all-ones vector. In addition, we apply linear projections on \( \mathbf{X} \) to allow interactions among feature channels. This leads to Equation 5, where \( \mathbf{V} = \mathbf{XW}_v \) and \( \mathbf{H} = \mathbf{XW}_H \) with trainable matrices \( \mathbf{W}_v, \mathbf{W}_H \in \mathbb{R}^{d \times d} \).

\[
\mathbf{X} = (\mathbf{WV}) \circ (\mathbf{H} + 1\beta^T) \tag{5}
\]

Subsequently, it is straightforward to show that we only need to achieve permutation equivariance on the term \( \mathbf{WV} \) in Equation 5 to build an equivariant model. To this end, we introduce the following two equivariant attention models, by leveraging graph topology and node features respectively.

Equivariant local attention. We incorporate graph topology information by setting \( \mathbf{W} = \mathbf{A} \), where \( \mathbf{A} \) is a (general) sparse attention matrix such that the nonzero elements in \( \mathbf{A} \) represent attention scores of adjacent nodes. While \( \mathbf{A} \) can be implemented by adopting any sparse attention methods previously proposed, we choose the GAT attention scheme (Veličković et al., 2017) due to its efficient implementation. We leave the exploration of other sparse attention approaches to future work. Consequently, the term \( \mathbf{AV} \) holds the equivariance property, i.e., \((P\mathbf{AP}^T)(P\mathbf{V}) = P(\mathbf{AV})\) for any node permutation matrix \( P \). It is noteworthy that replacing \( \mathbf{W} \) with \( \mathbf{A} \) essentially imposes a sparsity constraint on the learned polynomial of the base model, such that polynomial coefficients are nonzero if and only if the corresponding nonmonomial terms are formed by features of nearby nodes.

Equivariant global attention. By setting \( \mathbf{W} = \text{softmax}(\mathbf{S}) \), where \( \mathbf{S} \) denotes the global self-attention matrix defined in Equation 1, the term \( \mathbf{WV} \) in Equation 5 becomes \( \text{softmax}(\mathbf{S})\mathbf{V} \) that is permutation equivariant (Yun et al., 2019). According to our discussion in Appendix A, an L-layer model based on the updated Equation 5 learns an equivariant polynomial of degree \( 2^L \) with the coefficients determined by attention scores in \( \mathbf{S} \). More importantly, the learned polynomial contains all nonmonomial basis elements that capture global and high-order feature interactions. Nonetheless, the dense attention matrix \( \mathbf{S} \) still has quadratic complexity in regard to the number of nodes, rendering the approach not scalable to large graphs. To tackle this issue, we linearize the global attention by introducing a simple kernel trick in Equation 6, where \( \mathbf{Q} = \mathbf{XW}_Q, \mathbf{K} = \mathbf{XW}_K \) with weight matrices \( \mathbf{W}_Q, \mathbf{W}_K \), and \( \sigma \) denotes the sigmoid function to guarantee the attention scores are positive. The denominator term in Equation 6 ensures that the sum of attention scores is normalized to 1 per node. In this way, we preserve two key properties of the \( \text{softmax} \) function: non-negativity and normalization. As we can first compute \( \sigma(\mathbf{K}^T)\mathbf{V} \) whose output is then multiplied by \( \sigma(\mathbf{Q}) \), Equation 6 avoids computing the dense attention matrix, resulting in the linear global attention. In Appendix D, we further demonstrate the advantages of our approach over prior kernel-based linear attention methods in terms of hyperparameter tuning and training stability on large graphs.

\[
\mathbf{WV} = \frac{\sigma(\mathbf{Q})\sigma(\mathbf{K}^T)}{\sigma(\mathbf{Q}) \sum_i \sigma(\mathbf{K}^T_{i,:})} \mathbf{V} = \frac{\sigma(\mathbf{Q})\sigma(\mathbf{K}^T)\mathbf{V}}{\sigma(\mathbf{Q}) \sum_i \sigma(\mathbf{K}^T_{i,:})} \tag{6}
\]

Complexity analysis. Given a graph with \( n \) nodes and \( m \) edges, suppose the hidden dimension is \( d \ll n \), then the local attention model has the complexity of \( O(md + nd^2) \). Since we exploit the kernel trick to linearize the computation of global attention in Equation 6, the complexity of global attention model is reduced from \( O(n^2d) \) to \( O(nd^2) \). Hence, both proposed equivariant attention models have linear complexity with respect to the number of nodes/edges.

Discussion. It is noteworthy that the introduced local and global attention models are intended to enable the base model to learn high-degree equivariant polynomials, which is fundamentally distinct from the purpose of prior attention models in literature. This allows our proposed Polynomial model to outperform those attention-based baselines, even without nonlinear activation functions.

3.3 The Polynomial Architecture

Guided by the aforementioned equivariant models with linear attention, Figure 2b shows the Polynomial architecture that adopts a local-to-global attention scheme based on the following modules:

Local attention module. We employ the local attention approach via replacing \( \mathbf{W} \) in Equation 5 with the sparse attention matrix \( \mathbf{A} \) as discussed in Section 5.2, which explicitly learns an equivariant
Figure 2: Distinctions between the attention schemes in previous work and Polynormer — (a) Prior GTs use a local-and-global attention scheme, involving the simultaneous use of local and global attentions in each layer; (b) We adopt a local-to-global attention scheme, where local and global attention modules are applied sequentially; Note that the softmax operator in (a) and the attention normalization in (b) are omitted for brevity.

Polynormer attention scores to showcase its ability of capturing critical global structures.

Experimental Setup. Appendix E shows the details of all datasets used in our experiments, which consist of 7 homophilic and 6 heterophilic graphs. Notably, the heterophilic datasets are from [Platonov et al., 2023; Lim et al., 2021], which have addressed serious issues (e.g., train-test data leakage) of conventional popular datasets such as chameleon and squirrel. We compare Polynormer against 10 competitive GNNs. Note that prior graph polynomial models except tGNN (Hua et al., 2022) discussed in Section 2 run out of GPU memory even on the smallest dataset in our experiments.
experiments. Besides, we also evaluate 6 GTs that have shown promising results on the node classification task. We report the performance results of baselines from their original papers or official leaderboards whenever possible, as those results are obtained by well-tuned models. For baselines whose results are not publicly available on given datasets, we tune their hyperparameters to achieve the highest possible accuracy. Detailed hyperparameter settings of baselines and Polynormer are available in Appendix H. Our hardware information is provided in Appendix F.

4.1 PERFORMANCE ON HOMOPHILIC AND HETEROPHILIC GRAPHS

Table 1: Averaged node classification accuracy (%) ± std over 10 runs on homophilic datasets. — Polynormer-r denotes Polynormer with ReLU activation. We highlight the top first, second, and third results per dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Computer</th>
<th>Photo</th>
<th>CS</th>
<th>Physics</th>
<th>WikiCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCN</td>
<td>89.65 ± 0.52</td>
<td>92.70 ± 0.20</td>
<td>92.92 ± 0.12</td>
<td>96.18 ± 0.07</td>
<td>77.47 ± 0.85</td>
</tr>
<tr>
<td>GraphSAGE</td>
<td>91.20 ± 0.29</td>
<td>94.59 ± 0.14</td>
<td>93.91 ± 0.13</td>
<td>96.49 ± 0.06</td>
<td>74.77 ± 0.95</td>
</tr>
<tr>
<td>GAT</td>
<td>90.78 ± 0.13</td>
<td>93.87 ± 0.11</td>
<td>93.61 ± 0.14</td>
<td>96.17 ± 0.08</td>
<td>76.91 ± 0.82</td>
</tr>
<tr>
<td>GCNII</td>
<td>91.04 ± 0.41</td>
<td>94.30 ± 0.20</td>
<td>92.22 ± 0.14</td>
<td>95.97 ± 0.11</td>
<td>78.68 ± 0.55</td>
</tr>
<tr>
<td>GPRGNN</td>
<td>89.32 ± 0.29</td>
<td>94.49 ± 0.14</td>
<td>95.13 ± 0.09</td>
<td>96.85 ± 0.08</td>
<td>78.12 ± 0.23</td>
</tr>
<tr>
<td>APPNP</td>
<td>90.18 ± 0.17</td>
<td>94.32 ± 0.14</td>
<td>94.49 ± 0.07</td>
<td>96.54 ± 0.07</td>
<td>78.87 ± 0.11</td>
</tr>
<tr>
<td>PPRGo</td>
<td>88.69 ± 0.21</td>
<td>93.61 ± 0.12</td>
<td>92.52 ± 0.15</td>
<td>95.51 ± 0.08</td>
<td>77.89 ± 0.42</td>
</tr>
<tr>
<td>GPRGo</td>
<td>91.81 ± 0.20</td>
<td>94.50 ± 0.11</td>
<td>95.25 ± 0.05</td>
<td>97.07 ± 0.05</td>
<td>78.44 ± 0.53</td>
</tr>
<tr>
<td>OrderedGNN</td>
<td>92.03 ± 0.13</td>
<td>95.10 ± 0.20</td>
<td>95.00 ± 0.10</td>
<td>97.00 ± 0.08</td>
<td>79.01 ± 0.68</td>
</tr>
<tr>
<td>tGNN</td>
<td>83.40 ± 1.33</td>
<td>89.92 ± 0.72</td>
<td>92.85 ± 0.48</td>
<td>96.24 ± 0.24</td>
<td>71.49 ± 1.05</td>
</tr>
<tr>
<td>GraphGPS</td>
<td>91.19 ± 0.54</td>
<td>95.06 ± 0.13</td>
<td>93.93 ± 0.12</td>
<td>97.12 ± 0.19</td>
<td>78.66 ± 0.49</td>
</tr>
<tr>
<td>NAGphormer</td>
<td>91.22 ± 0.14</td>
<td>95.49 ± 0.11</td>
<td>95.75 ± 0.09</td>
<td>97.34 ± 0.03</td>
<td>77.16 ± 0.72</td>
</tr>
<tr>
<td>Explorer</td>
<td>91.47 ± 0.17</td>
<td>95.35 ± 0.22</td>
<td>94.93 ± 0.01</td>
<td>96.80 ± 0.09</td>
<td>78.54 ± 0.49</td>
</tr>
<tr>
<td>NodeFormer</td>
<td>86.98 ± 0.62</td>
<td>93.46 ± 0.35</td>
<td>95.64 ± 0.22</td>
<td>96.45 ± 0.28</td>
<td>74.73 ± 0.94</td>
</tr>
<tr>
<td>DIFFormer</td>
<td>91.99 ± 0.76</td>
<td>95.10 ± 0.47</td>
<td>94.78 ± 0.20</td>
<td>96.60 ± 0.18</td>
<td>73.46 ± 0.56</td>
</tr>
<tr>
<td>GOAT</td>
<td>90.96 ± 0.90</td>
<td>92.96 ± 1.48</td>
<td>94.21 ± 0.38</td>
<td>96.24 ± 0.24</td>
<td>77.00 ± 0.77</td>
</tr>
<tr>
<td>Polynormer</td>
<td>93.18 ± 0.18</td>
<td>96.11 ± 0.23</td>
<td>95.51 ± 0.29</td>
<td>97.22 ± 0.06</td>
<td>79.53 ± 0.83</td>
</tr>
<tr>
<td>Polynormer-r</td>
<td>93.68 ± 0.21</td>
<td>96.46 ± 0.26</td>
<td>95.53 ± 0.16</td>
<td>97.27 ± 0.08</td>
<td>80.10 ± 0.67</td>
</tr>
</tbody>
</table>

Performance on homophilic graphs. We first compare Polynormer with 16 popular baselines that include SOTA GNNs or GTs on 5 common homophilic datasets. As shown in Table 1, Polynormer is able to outperform all baselines on 3 out of 5 datasets, which clearly demonstrate the efficacy of the proposed polynomial-expressive model. Moreover, incorporating the ReLU function can lead to further improvements in the accuracy of Polynormer. This is because the nonlinearity imposed by ReLU introduces additional higher-order monomials, which in turn enhance the quality of node representations. It is noteworthy that the accuracy gain of Polynormer over SOTA baselines is 1.65% on Computer, which is a nontrivial improvement given that those baselines have been finely tuned on these well-established homophilic datasets.

Performance on heterophilic graphs. Table 2 reports the average results over 10 runs on heterophilic graphs. Notably, the GT baselines underperform SOTA GNNs on all 5 datasets, which raises concerns about whether those prior GT models properly utilize the expressivity brought by the self-attention module. Moreover, there is no baseline model that consistently ranks among the top 3 models across 5 datasets. In contrast, by integrating the attention mechanism into the polynomial-expressive model, Polynormer surpasses all baselines on 4 out of 5 datasets. Furthermore, Polynormer-r consistently outperforms baselines across all datasets with the accuracy improvement by a margin up to 3.55% (i.e., 97.46% vs. 93.91% on minesweeper).

4.2 PERFORMANCE ON LARGE GRAPHS

We conduct experiments on large graphs by comparing Polynormer against GTs as well as some representative GNN models. Since the graphs in ogbn-products and poiec are too large to be loaded into the GPU memory for full-batch training, we leverage the random partitioning method adopted by prior GT models [2022] [2023] to perform mini-batch training. As shown in Table 3, Polynormer-r outperforms all baselines on these large graphs with the accuracy gain of up to 4.06%. Besides, we can observe that the accuracy of Polynormer drops 1.64% when removing ReLU activation on ogbn-arxiv. Thus, we believe the nonlinearity associated with ReLU is more critical on
Table 2: Averaged node classification results over 10 runs on heterophilic datasets — Accuracy is reported for roman-empire and amazon-ratings, and ROC AUC is reported for minesweeper, tolokers, and questions. Polynormer-r denotes Polynormer with ReLU activation. We highlight the top first, second, and third results per dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>roman-empire</th>
<th>amazon-ratings</th>
<th>minesweeper</th>
<th>tolokers</th>
<th>questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCN</td>
<td>73.69 ± 0.74</td>
<td>48.70 ± 0.63</td>
<td>89.75 ± 0.52</td>
<td>83.64 ± 0.67</td>
<td>76.09 ± 1.27</td>
</tr>
<tr>
<td>GraphSAGE</td>
<td>85.74 ± 0.67</td>
<td>53.63 ± 0.39</td>
<td>93.51 ± 0.57</td>
<td>82.43 ± 0.44</td>
<td>76.44 ± 0.62</td>
</tr>
<tr>
<td>GAT-sep</td>
<td>88.75 ± 0.41</td>
<td>52.70 ± 0.62</td>
<td>93.91 ± 0.35</td>
<td>83.78 ± 0.43</td>
<td>76.79 ± 0.71</td>
</tr>
<tr>
<td>H2GCN</td>
<td>60.11 ± 0.52</td>
<td>36.47 ± 0.23</td>
<td>89.71 ± 0.31</td>
<td>73.35 ± 1.01</td>
<td>63.59 ± 1.46</td>
</tr>
<tr>
<td>GPRGNN</td>
<td>64.85 ± 0.27</td>
<td>44.88 ± 0.34</td>
<td>86.24 ± 0.61</td>
<td>72.94 ± 0.97</td>
<td>55.48 ± 0.91</td>
</tr>
<tr>
<td>FSGNN</td>
<td>79.92 ± 0.56</td>
<td>52.74 ± 0.83</td>
<td>90.08 ± 0.70</td>
<td>82.76 ± 0.61</td>
<td>78.86 ± 0.92</td>
</tr>
<tr>
<td>GloGNN</td>
<td>59.63 ± 0.69</td>
<td>36.89 ± 0.14</td>
<td>51.08 ± 1.23</td>
<td>73.39 ± 1.17</td>
<td>65.74 ± 1.19</td>
</tr>
<tr>
<td>GGNCN</td>
<td>74.46 ± 0.54</td>
<td>43.00 ± 0.32</td>
<td>87.54 ± 1.22</td>
<td>77.31 ± 1.14</td>
<td>71.10 ± 1.57</td>
</tr>
<tr>
<td>OrderedGNN</td>
<td>77.68 ± 0.39</td>
<td>47.29 ± 0.65</td>
<td>80.58 ± 1.08</td>
<td>75.60 ± 1.36</td>
<td>75.09 ± 1.00</td>
</tr>
<tr>
<td>G²-GNN</td>
<td>82.16 ± 0.78</td>
<td>47.93 ± 0.58</td>
<td>91.83 ± 0.56</td>
<td>82.51 ± 0.80</td>
<td>74.82 ± 0.92</td>
</tr>
<tr>
<td>DIR-GNN</td>
<td>91.23 ± 0.32</td>
<td>47.89 ± 0.39</td>
<td>87.05 ± 0.69</td>
<td>81.19 ± 1.05</td>
<td>76.13 ± 1.24</td>
</tr>
<tr>
<td>tGNN</td>
<td>79.95 ± 0.75</td>
<td>48.21 ± 0.53</td>
<td>91.93 ± 0.77</td>
<td>78.04 ± 1.75</td>
<td>76.38 ± 1.79</td>
</tr>
<tr>
<td>GraphGPS</td>
<td>82.00 ± 0.61</td>
<td>53.10 ± 0.42</td>
<td>90.63 ± 0.67</td>
<td>83.71 ± 0.48</td>
<td>71.73 ± 1.47</td>
</tr>
<tr>
<td>NAGphormer</td>
<td>74.34 ± 0.77</td>
<td>51.26 ± 0.72</td>
<td>84.19 ± 0.66</td>
<td>78.32 ± 0.95</td>
<td>68.17 ± 1.53</td>
</tr>
<tr>
<td>Exphormer</td>
<td>89.03 ± 0.37</td>
<td>53.51 ± 0.46</td>
<td>90.74 ± 0.53</td>
<td>83.77 ± 0.78</td>
<td>73.94 ± 1.06</td>
</tr>
<tr>
<td>NodeFormer</td>
<td>64.49 ± 0.73</td>
<td>43.86 ± 0.35</td>
<td>86.71 ± 0.88</td>
<td>78.10 ± 1.03</td>
<td>74.27 ± 1.46</td>
</tr>
<tr>
<td>DiffFormer</td>
<td>79.10 ± 0.32</td>
<td>47.84 ± 0.65</td>
<td>90.89 ± 0.58</td>
<td>83.57 ± 0.68</td>
<td>72.15 ± 1.31</td>
</tr>
<tr>
<td>GOAT</td>
<td>71.59 ± 1.25</td>
<td>44.61 ± 0.50</td>
<td>81.09 ± 1.02</td>
<td>83.11 ± 1.04</td>
<td>75.76 ± 1.66</td>
</tr>
<tr>
<td>Polynormer</td>
<td>92.13 ± 0.50</td>
<td>54.46 ± 0.40</td>
<td>96.96 ± 0.52</td>
<td>84.83 ± 0.72</td>
<td>77.95 ± 1.06</td>
</tr>
<tr>
<td>Polynormer-r</td>
<td>92.55 ± 0.37</td>
<td>54.81 ± 0.49</td>
<td>97.46 ± 0.36</td>
<td>85.91 ± 0.74</td>
<td>78.92 ± 0.89</td>
</tr>
</tbody>
</table>

Table 3: Averaged node classification accuracy (%) ± std on large-scale datasets — Polynormer-r denotes Polynormer with ReLU activation. We highlight the top first, second, and third results per dataset. OOM means out of memory.

<table>
<thead>
<tr>
<th>Model</th>
<th>ogbn-arxiv</th>
<th>ogbn-products</th>
<th>pokec</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCN</td>
<td>71.74 ± 0.29</td>
<td>75.64 ± 0.21</td>
<td>75.45 ± 0.17</td>
</tr>
<tr>
<td>GAT</td>
<td>72.01 ± 0.20</td>
<td>79.45 ± 0.59</td>
<td>72.23 ± 0.18</td>
</tr>
<tr>
<td>GPRGNN</td>
<td>71.10 ± 0.12</td>
<td>79.76 ± 0.59</td>
<td>78.83 ± 0.05</td>
</tr>
<tr>
<td>LINX</td>
<td>66.18 ± 0.33</td>
<td>71.59 ± 0.71</td>
<td>82.04 ± 0.07</td>
</tr>
<tr>
<td>GraphGPS</td>
<td>70.97 ± 0.41</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>NAGphormer</td>
<td>70.13 ± 0.55</td>
<td>73.55 ± 0.21</td>
<td>76.59 ± 0.25</td>
</tr>
<tr>
<td>Exphormer</td>
<td>72.44 ± 0.28</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>NodeFormer</td>
<td>67.19 ± 0.83</td>
<td>72.93 ± 0.13</td>
<td>71.00 ± 1.30</td>
</tr>
<tr>
<td>DiffFormer</td>
<td>69.86 ± 0.25</td>
<td>74.16 ± 0.31</td>
<td>73.89 ± 0.35</td>
</tr>
<tr>
<td>GOAT</td>
<td>72.41 ± 0.40</td>
<td>82.00 ± 0.43</td>
<td>66.37 ± 0.94</td>
</tr>
<tr>
<td>Polynormer</td>
<td>71.82 ± 0.23</td>
<td>82.97 ± 0.28</td>
<td>85.95 ± 0.07</td>
</tr>
<tr>
<td>Polynormer-r</td>
<td>73.46 ± 0.16</td>
<td>83.82 ± 0.11</td>
<td>86.10 ± 0.05</td>
</tr>
</tbody>
</table>

ogbn-arxiv, which is known to be a challenging dataset (Shirzad et al. 2023). In addition, Appendix I provides the training time and memory usage of Polynormer and G1 baselines. Moreover, we further evaluate Polynormer on an industrial-level graph benchmark named TpuGraphs (Piothilimithana et al. 2024), whose results are provided in Appendix J.

4.3 Ablation Analysis on Polynormer Attention Schemes

Figure [I] shows the comparison of the SOTA baseline (red bar), Polynormer without global attention (orange bar), a variant of Polynormer where the local and global attention modules are applied simultaneously to update node features per layer (blue bar), and Polynormer (green bar). We provide the detailed architecture of the Polynormer variant in Appendix C. Besides, we omit the results of Polynormer without local attention, as it completely ignores the graph structural information and thus performs poorly on the graph datasets.

By comparing the orange and green bars in Figure [3] we can observe that the local attention model achieves comparable results to Polynormer on homophilic graphs (Computer and Photo), while it lags behind Polynormer on heterophilic graphs (roman-empire and minesweeper). This observation...
Figure 3: Ablation studies on attention modules of Polynormer — “Local Attention” means only the local attention module is used. “Local-and-Global Attention” denotes the local and global attention modules are employed in parallel, while “Local-to-Global Attention” represents our proposed Polynormer model where the local attention module is followed by the global attention module.

highlights the efficacy of the global attention module in Polynormer, which captures global information that proves more beneficial for heterophilic graphs (Li et al., 2022), as further illustrated in Section 4.4. Moreover, Figure 3 also demonstrates that the local-to-global attention adopted by Polynormer performs better than its counterpart, i.e., local-and-global attention, which has been widely used by prior GT models (Rampasek et al., 2022; Wu et al., 2022; 2023; Kong et al., 2023). We attribute it to that the local-and-global attention may introduce the risk of mixing local and non-local interactions (Di Giovanni et al., 2023), while the local-to-global attention inherently avoids this issue. This is further confirmed by the results on Computer, where the local-and-global attention model performs even worse than the local attention alone model.

4.4 VISUALIZATION

Figure 4: Visualization on the importance of nodes (columns) to each target node (row) — Higher heatmap values indicate greater importance; Both subfigures (a) and (b) consider nodes are important if they share the same label as the target node, while (a) has an additional constraint that these nodes are at most 5-hop away from the target node; Subfigure (c) measures node importance based on the corresponding global attention scores in Polynormer.

For the sake of clear visualization, we randomly sample 100 nodes from minesweeper, based on which three heatmaps are drawn in Figure 4. Notably, the coordinates in each heatmap correspond to node indices in the graph. We first draw Figure 4(a) by assigning every heatmap value to 1 if the corresponding two nodes have the same label and their shortest path distance is up to 5, and 0 otherwise. This essentially highlights locally important neighbors per node. Similarly, we further visualize globally important nodes by removing the local constraint, resulting in Figure 4(b). Figure 4(c) visualizes the attention matrix in the last global attention layer of Polynormer. Note that we linearly scale the attention scores such that the maximum value is 1 for the visualization purpose.

By comparing Figures 4(a) and 4(b), we observe numerous globally important nodes that can be exploited for accurate predictions on target nodes. Figure 4(c) clearly showcases that the attention scores of Polynormer effectively differentiate those globally important nodes from unimportant ones. As a result, these attention scores enable Polynormer to focus on crucial monomial terms that consist of a sequence of globally important nodes, allowing it to learn critical global structural information.

5 CONCLUSIONS

This work introduces Polynormer that adopts a linear local-to-global attention scheme with high polynomial expressivity. Our experimental results indicate that Polynormer achieves SOTA results on a wide range of graph datasets, even without the use of nonlinear activation functions.
ACKNOWLEDGMENTS

This work is supported in part by NSF Awards #2118709 and #2212371, a Qualcomm Innovation Fellowship, and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES


Liheng Ma, Chen Lin, Derek Lim, Adriana Romero-Soriano, Puneet Kumar Dokania, Mark Coates, Philip H. S. Torr, and Ser Nam Lim. Graph inductive biases in transformers without message passing. ArXiv, abs/2305.17589, 2023.


A Proof for Theorem 3.3

Proof. For the convenience of our proof, we denote the initial feature matrix by \( X \in \mathbb{R}^{n \times d} \), the number of layers by \( L \), the number of nodes by \( n \), and \([n] = \{1, 2, \ldots, n\}\). Besides, we set \( B^{(l)} = 0, \forall l \in [L] \). The role of \( B^{(l)} \) is analyzed in Appendix B. By defining \( I \) to be an ordered set that consists of node indices (e.g., \( I = (i_1, i_2, i_3, i_4) \)), we define the following function \( c_L \) on coefficients:

\[
c_L(I; W) = \begin{cases} \prod_{l=1}^{L} W^{(l)}_{i_1, i_2} W^{(L-1)}_{i_3, i_4} \cdots W^{(2)}_{i_2, i_1} \cdots W^{(1)}_{i_2, i_1} & \text{if } f_l = 1 \\ 1 & \text{otherwise} \end{cases}
\]

(9)

where \( W^{(j)}_{i_a, i_b} \) represents the \((i_a, i_b)\)-th block in \( W \) at the \( j \)-th layer. Based on Equation 9, we further define the following monomial \( f_L \) of degree \( 2^L \), where each \( X_i \) denotes a \( d \)-dimensional feature vector of node \( i \):

\[
f_L(I; W, X) = c_L(I; W) X_{i_1} \odot X_{i_2} \cdots \odot X_{i_2L}
\]

(10)

Next, we prove by induction that the \( L \)-layer base model produces the following node representation \( X_i^{(L)} \) for any node \( i \in [n] \):

\[
X_i^{(L)} = \sum_{I \in S^{i}_{2L}} f_L(I; W, X)
\]

(11)

where \( S^{i}_{2L} \) is a set that represents all the combinations of choosing \( 2^L \) elements from \([n]\) (with replacement), with the first element fixed to be \( i \), i.e., \( S^{i}_{2L} := \{y \in [n]^{2L} \mid y_1 = i\} \).

Base case. When \( L = 1 \), we have:

\[
X_i^{(1)} = \left( \sum_{j=1}^{n} W^{(1)}_{i, j} X_j \right) \odot X_i
\]

\[
= \sum_{j=1}^{n} W^{(1)}_{i, j} X_i \odot X_j
\]

\[
= \sum_{I \in S^{i}_{2}} f_I(I; W, X)
\]

(12)

Induction step. When \( L = l \), suppose we have \( X_i^{(l)} = \sum_{I \in S^{i}_{2L}} f_L(I; W, X) \). Then, for \( L = l + 1 \), we have:

\[
X_i^{(l+1)} = \left( \sum_{j=1}^{n} W^{(l+1)}_{i, j} X_j^{(l)} \right) \odot X_i^{(l)}
\]

\[
= \left( \sum_{j=1}^{n} W^{(l+1)}_{i, j} \sum_{J \in S^{i}_{2L}} f_J(J; W, X) \right) \odot \left( \sum_{I \in S^{i}_{2L}} f_I(I; W, X) \right)
\]

\[
= \left( \sum_{j=1}^{n} W^{(l+1)}_{i, j} \sum_{J \in S^{i}_{2L}} \left( \sum_{I \in S^{i}_{2L}} f_I(I; W, X) \right) \right) \odot \left( \sum_{I \in S^{i}_{2L}} f_I(I; W, X) \right)
\]

\[
= \sum_{I \in S^{i}_{2L}} \sum_{j=1}^{n} W^{(l+1)}_{i, j} \sum_{J \in S^{i}_{2L}} c_I(I; W, X) X_{i_1} \odot X_{i_2} \cdots \odot X_{i_{2L}}
\]

\[
= \sum_{I \in S^{i}_{2L}} \sum_{j=1}^{n} c_I(I; W, X) W^{(l+1)}_{i, j} X_{i_1} \odot X_{i_2} \cdots \odot X_{i_{2L}}
\]

(13)
The first 3 equations and the last equation above are straightforward. For the fourth equation, we essentially expand the product in LHS to the sum in RHS. For the second last equation, we merge the weight matrix \( W^{(l+1)} \) into the coefficient function \( c_l(J; W) \), and then replace the index set \( J = \{j_1, j_2, \ldots, j_{2^L}\} \) with \( I' = \{i_{2^L+1}, i_{2^L+2}, \ldots, i_{2^L+1}\} \), which is combined with the set \( \{i_1, i_2, \ldots, i_{2^L}\} \) to obtain the new index set \( I = \{i_1, i_2, \ldots, i_{2^L+1}, i_{2^L+1}\} \in S_{2^{L-1}}^{2}. \) This complements our proof for Equation [11].

As for any node \( i \), the first index of \( I \in S_{2^{L}}^{2} \) is always \( i \), we can rewrite Equation [11] as:

\[
X_i^{(L)} = \sum_{I' \in S_{2^{L-1}}^{2}} f_{L}(I; W, X) = \sum_{I' \in S_{2^{L-1}}^{2}} c_{L}(I; W) X_{i_{1}} \odot X_{i_{2}} \odot \cdots \odot X_{i_{2^L}}
\]

\[
X_i = \sum_{I' \in S_{2^{L-1}}^{2}} \sum_{I'' \in S_{2^{L-1}}^{2}} W_{i_{1}',i_{2}',i_{3}'}^{(L-1)} W_{i_{3}',i_{4}'}^{(L-2)} \cdots W_{i_{2^{L-2}'},i_{2^{L-1}'}}^{(1)} X_{i_{1}'} \odot \cdots \odot X_{i_{2^L-1}'}
\]

where \( S_{2^{L-1}}^{2} \) is a set that represents all the combinations of choosing \( 2^L - 1 \) elements from \([n]\)(with replacement), i.e., \( S_{2^{L-1}}^{2} = \{y \in [n]^{2^L-1}\} \).

As a result, for any degree-\((2^L-1)\) monomial \( M^{2^L-1} \) formed by rows in \( X \), we can denote it by \( X_{i_{1}} \odot X_{i_{2}} \odot \cdots \odot X_{i_{2^L-1}} \), which corresponds to a specific ordered index set \( I' \in S_{2^{L-1}}^{2} \) in Equation [14]. Therefore, we can always parameterize weight matrices \( W^{(l)}, l \in [L] \) such that only the elements with indices \( I' \) that determine the given monomial are 1 in Equation [14] and all other elements in \( W^{(l)} \) are set to 0. Consequently, Equation [14] with parameterized \( W^{(l)} \) becomes \( X_i \odot M^{2^L-1} \), which complements our proof for Theorem 3.3.

Discussion. Notably, Equation [9] reveals that the weight matrix \( W \) controls the coefficients of the monomial terms of degree \( 2^L \) in the learned polynomial. Thus, if we replace \( W \) with any attention matrix \( S \) (e.g., the global self-attention matrix described in Section 3.2), then the attention scores in \( S \) naturally control all the monomial coefficients, which essentially capture the importance of different (global) node feature interactions with order \( 2^L \). In Appendix B, we are going to provide the analysis to demonstrate that the matrix \( B \) in Equation [4] controls the coefficients of lower-degree monomials in practice.

B Analysis on \( B \) in Equation [4]

By ignoring the weight matrix \( B \) in Equation [4], we have demonstrated in Appendix A that the base model learns a polynomial that encompasses all possible monomials of degree \( 2^L \). However, our proof in Appendix A also reveals that the learned polynomial is unable to represent any monomials with degrees smaller than \( 2^L \). This raises concerns since these lower-degree monomial terms are often crucial for the model predictions (Hua et al., 2022). Fortunately, we are going to show that incorporating \( B \) into Equation [11] enables the \( L \)-layer base model to capture all monomial terms with degrees up to \( 2^L \) (except the constant term).

Specifically, we can consider each monomial term of the learned polynomial as generated in the following way: for each layer \( l \) of the base model, we select two monomials, denoted as \( M^{(l)}_{1} \) and
\(M_l^{(l)}\), from the terms \((W^{(l)}X^{(l-1)})\) and \((X^{(l-1)}+B^{(l)})\) in Equation 4, respectively. Consequently, the generated monomial after layer \(l\) is obtained by the Hadamard product, i.e., \(M_l^{(l)} = M_1^{(l)} \circ M_2^{(l)}\). We continue this process recursively until we obtain \(M^{(L)}\) at the \(L\)-th layer.

Regarding the selection of a monomial \(M_2^{(l)}\) from \((X^{(l-1)}+B^{(l)})\), two scenarios arise: (i) If \(M_2^{(l)}\) is a monomial term from the polynomial \(X^{(l-1)}\), it increases the degree of the generated monomial \(M^{(l)}\) by up to the degree of \(X^{(l-1)}\) (i.e., \(2^{l-1}\)), as analyzed in Appendix A. (ii) If \(M_2^{(l)} = B^{(l)}\), it does not increase the degree.

If we never choose \(M_2^{(l)} = B^{(l)}\) for all \(l \in [L]\), then the generated monomial at layer \(L\) has a degree of \(2^L\), as proven in Appendix A. Hence, each time we select \(M_2^{(l)} = B^{(l)}\) at layer \(l\), we reduce the degree of a monomial at the \(L\)-th layer (whose original degree is \(2^L\)) by \(2^{l-1}\). In other words, the more we opt for \(M_2^{(l)} = B^{(l)}\), the smaller the degree of the monomial at the \(L\)-th layer, and vice versa. In the extreme scenario where we choose \(M_2^{(l)} = B^{(l)}\) for all \(l \in [L]\), we obtain a monomial of degree \(2^L - 2^{L-1} - \ldots - 2^1 - 2^0 = 1\). Consequently, the weight matrices \(B^{(l)}, l \in [L]\), enable the base model to capture all possible monomials with degrees ranging from 1 up to \(2^L\).

**Discussion.** The above analysis demonstrates the importance of \(B\) in Equation 3, i.e., controlling the coefficients of lower-degree monomials, which is another key distinction of our approach to previous gating-based GNNs that ignore \(B\) in their gating units (Yan et al., 2022; Rusch et al., 2022; Song et al., 2023).

### C POLYNOMIAL EXPRESSIVITY OF PRIOR GRAPH MODELS

**Polynomial expressivity of GTs.** For the sake of simplicity, we omit the \texttt{softmax} operator in GTs and assume each node feature contains a scalar value, i.e., \(x \in \mathbb{R}^n\) for an \(n\)-node graph. Besides, we denote the weights for query, key, and value by \(w_q, w_k,\) and \(w_v\) respectively. Consequently, for any node \(i\), a GT layer produces \(x'_i = w_q w_k w_v \sum x_j x_j x_j = w_q w_k w_v \sum x_j x_j^2\), which consists of degree-3 monomials. However, there are still many degree-3 monomials that GTs fail to capture. For instance, any monomial in the set \(\{x_i x_j | i \neq j\}\) cannot be captured by \(x'_i\), resulting in the limited polynomial expressivity of prior GT models.

**Polynomial expressivity of high-order GNNs.** We initially focus on a closely related model known as tGNN, which is recently proposed by Hua et al. (2022). In tGNN, the polynomial for each target node \(i\) is computed as \([x_{i,1}] W \odot \ldots \odot [x_{i,k}] W\), where \([x_{i,j}]\) represents the concatenation of the feature vector of node \(i\) with a constant 1. As asserted by Hua et al. (2022), tGNN learns a multilinear polynomial in which no variables appear with a power of 2 or higher. Consequently, tGNN cannot represent any monomials where a specific variable has a power of at least 2, limiting its polynomial expressivity.

Furthermore, there exist multiple GNN models based on gating mechanisms that implicitly capture high-order interactions among node features (e.g., GGNN (Yan et al., 2022), \(G^2\)-GNN (Rusch et al., 2022), and OrderedGNN (Song et al., 2023)). However, these models primarily consider local structures and, therefore, are unable to learn high-order monomials formed by features of distant nodes. Additionally, few gating-based GNNs consider leveraging the weight matrix \(B\) in Equation 4 to learn lower-degree monomial terms, as discussed in Appendix B.

**Conclusion.** Due to the limitations of polynomial expressivity analyzed above, previous GTs and high-order GNNs still necessitate nonlinear activation functions to attain reasonable accuracy. In contrast, our method achieves higher accuracy than these models even without activation functions, thanks to its high polynomial expressivity.

### D COMPARISON TO KERNEL-BASED LINEAR ATTENTION MODELS

#### D.1 HYPERPARAMETER TUNING

In literature, several linear GTs employ kernel-based methods to approximate the dense self-attention matrix within the vanilla Transformer architecture. Concretely, both GraphGPS (with...
Performer-based attention) (Rampasek et al., 2022) and NodeFormer (Wu et al., 2022) utilize a kernel function based on positive random features (PRFs). However, PRFs introduce a critical hyperparameter, denoted as $m$, which determines the dimension of the transformed features. In addition to this, NodeFormer introduces another crucial hyperparameter, $\tau$, controlling the temperature in Gumbel-Softmax, which is integrated into the kernel function. Furthermore, the DiFFormer model (Wu et al., 2023) offers a choice between two types of kernel functions in practice.

Consequently, these kernel-based GTs require extensive tuning of these critical hyperparameters, which can be time-consuming, particularly when dealing with large graphs. In contrast, our proposed linear global attention lies in their choice of kernel function. CosFormer utilizes the ReLU function instead of the Sigmoid function $\sigma$ in Equation 6. While ReLU performs well on text data containing up to 16K tokens per sequence, we observe that it leads to training instability when applied to large graphs with millions of nodes.

Specifically, if we substitute ReLU into Equation 6, the denominator term $\sum_i \sigma(K_{i,:}^T)$ becomes $\sum_i ReLU(K_{i,:}^T)$. This modification results in a training loss of $NaN$ during our experiments on both the ogbn-products and pokec datasets. The issue arises because ReLU only sets negative values in $K_{i,:}$ to zero while preserving positive values. As a consequence, the term $\sum_i ReLU(K_{i,:}^T)$ accumulates potentially large positive values across millions of nodes. This leads to the denominator in Equation 6 exceeding the representational capacity of a 32-bit floating-point format and results in a $NaN$ loss during model training.

In contrast, our approach employs the Sigmoid function, which maps all elements in $K_{i,:}$ to the range $(0, 1)$. As a result, $\sum_i \sigma(K_{i,:}^T)$ does not produce excessively large values, avoiding the issue of $NaN$ loss.

### E Dataset Details

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Type</th>
<th>Homophily Score</th>
<th>Nodes</th>
<th>Edges</th>
<th>Classes</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>Homophily</td>
<td>0.700</td>
<td>13,752</td>
<td>245,861</td>
<td>10</td>
<td>767</td>
</tr>
<tr>
<td>Photo</td>
<td>Homophily</td>
<td>0.772</td>
<td>7,650</td>
<td>119,081</td>
<td>8</td>
<td>745</td>
</tr>
<tr>
<td>CS</td>
<td>Homophily</td>
<td>0.755</td>
<td>18,333</td>
<td>81,894</td>
<td>15</td>
<td>6,805</td>
</tr>
<tr>
<td>Physics</td>
<td>Homophily</td>
<td>0.847</td>
<td>34,493</td>
<td>247,962</td>
<td>5</td>
<td>8,415</td>
</tr>
<tr>
<td>WikiCS</td>
<td>Homophily</td>
<td>0.568</td>
<td>11,701</td>
<td>216,123</td>
<td>10</td>
<td>300</td>
</tr>
<tr>
<td>roman-empire</td>
<td>Heterophy</td>
<td>0.023</td>
<td>22,662</td>
<td>32,927</td>
<td>18</td>
<td>300</td>
</tr>
<tr>
<td>amazon-ratings</td>
<td>Heterophy</td>
<td>0.127</td>
<td>24,492</td>
<td>93,050</td>
<td>5</td>
<td>300</td>
</tr>
<tr>
<td>minesweeper</td>
<td>Heterophy</td>
<td>0.009</td>
<td>10,000</td>
<td>30,402</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>tolokers</td>
<td>Heterophy</td>
<td>0.187</td>
<td>11,758</td>
<td>519,000</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>questions</td>
<td>Heterophy</td>
<td>0.072</td>
<td>48,921</td>
<td>153,540</td>
<td>2</td>
<td>301</td>
</tr>
<tr>
<td>ogbn-arxiv</td>
<td>Homophily</td>
<td>0.416</td>
<td>169,343</td>
<td>1,166,243</td>
<td>40</td>
<td>128</td>
</tr>
<tr>
<td>ogbn-products</td>
<td>Homophily</td>
<td>0.459</td>
<td>2,449,029</td>
<td>61,859,140</td>
<td>47</td>
<td>100</td>
</tr>
<tr>
<td>pokec</td>
<td>Heterophy</td>
<td>0.000</td>
<td>1,632,803</td>
<td>30,622,564</td>
<td>2</td>
<td>65</td>
</tr>
</tbody>
</table>

Table 4 shows the statistics of all 13 datasets used in our experiments. The homophily score per dataset is computed based on the metric proposed by Lim et al. (2021) (higher score means more homophilic).

**Train/Valid/Test splits.** For Computer, Photo, CS, and Physics datasets, we adhere to the widely accepted practice of randomly dividing nodes into training (60%), validation (20%), and test (20%) sets (Chen et al., 2022b, Shirzad et al., 2023). As for the remaining datasets in our experiments, we
use the official splits provided in their respective papers (Mernyei & Cangea 2020; Hu et al. 2020; Lim et al. 2021; Platonov et al. 2023).
G²-GNN (Rusch et al., 2022). We set the hidden dimension to 512, the learning rate to 0.001, and the number of epochs to 2000. We perform hyperparameter tuning on the number of layers from \{5, 10\}, the dropout rate from \{0.3, 0.5, 0.7\}, and the exponent \(p\) from \{2, 3, 4\}.

DIR-GNN (Rossi et al., 2023). We set the hidden dimension to 512, the learning rate to 0.001, the number of epochs to 2000, and \(\alpha\) to 0.5. Besides, we choose GATConv with the type of jumping knowledge as “max”. We perform hyperparameter tuning on the number of layers from \{3, 5\}, the dropout rate from \{0.3, 0.5, 0.7\}.

GraphGPS (Rampasek et al., 2022). We choose GAT as the global attention layer type. We set the number of layers to 2, the number of heads to 8, the hidden dimension to 64, and the number of epochs to 2000. We perform hyperparameter tuning on the learning rate from \{1e−4, 5e−4, 1e−3\}, and the dropout rate from \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}.

NAfformer (Chen et al., 2022b). We set the hidden dimension to 512, the learning rate to 0.001, the batch size to 2000, and the number of epochs to 500. We perform hyperparameter tuning on the number of layers from \{1, 2, 3\}, the number of heads from \{1, 8\}, the number of hops from \{3, 7, 10\}, and the dropout rate from \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}.

Exphormer (Shirzad et al., 2023). We choose GAT as the local model and Exphormer as the global model. We set the number of epochs to 2000 and the number of heads to 8. We perform hyperparameter tuning on the learning rate from \{1e−4, 1e−3\}, the number of layers from \{2, 4\}, the hidden dimension form \{64, 80, 96\}, and the dropout rate from \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}.

NodeFormer (Wu et al., 2022). We set the number of epochs to 2000. Additionally, we perform hyperparameter tuning on the learning rate from \{1e−4, 1e−3, 1e−2\}, the number of layers from \{1, 2, 3\}, the hidden dimension from \{32, 64, 128\}, the number of heads from \{1, 4\}, \(K\) from \{30, 50\}, \(r_{\text{order}}\) from \{1, 2\}, the dropout rate from \{0.0, 0.3\}, and the temperature \(\tau\) from \{0.10, 0.15, 0.20, 0.25, 0.30, 0.40, 0.50\}.

DIFFormer (Wu et al., 2023). We use the “simple” kernel. Moreover, we perform hyperparameter tuning on the learning rate from \{1e−4, 1e−3, 1e−2\}, the number of layers from \{2, 3\}, the hidden dimension form \{64, 128\}, the number of heads from \{1, 8\}, \(\alpha\) from \{0.1, 0.2, 0.3\}, and the dropout rate from \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}.

GOAT (Kong et al., 2023). We set the “conv_type” to “full”, the number of layers to 1 (fixed by GOAT), the number of epochs to 200, the number of centroids to 4096, the hidden dimension to 256, the dropout of feed forward layers to 0.5, and the batch size to 1024. We perform hyperparameter tuning on the learning rate from \{1e−4, 1e−3, 1e−2\}, the global dimension from \{128, 256\}, and the attention dropout rate from \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}.

H.2 Polynormer

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Warm-up Epochs</th>
<th>Local-to-Global Epochs</th>
<th>Local Layers</th>
<th>Global Layers</th>
<th>Dropout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>200</td>
<td>1000</td>
<td>5</td>
<td>1</td>
<td>0.7</td>
</tr>
<tr>
<td>Photo</td>
<td>200</td>
<td>1000</td>
<td>7</td>
<td>2</td>
<td>0.7</td>
</tr>
<tr>
<td>CS</td>
<td>100</td>
<td>1500</td>
<td>5</td>
<td>2</td>
<td>0.3</td>
</tr>
<tr>
<td>Physics</td>
<td>100</td>
<td>1500</td>
<td>5</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>WikiCS</td>
<td>100</td>
<td>1000</td>
<td>7</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>roman-empire</td>
<td>100</td>
<td>2500</td>
<td>10</td>
<td>2</td>
<td>0.3</td>
</tr>
<tr>
<td>amazon-ratings</td>
<td>200</td>
<td>2500</td>
<td>10</td>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>minesweeper</td>
<td>100</td>
<td>2000</td>
<td>10</td>
<td>3</td>
<td>0.3</td>
</tr>
<tr>
<td>tolokers</td>
<td>100</td>
<td>800</td>
<td>7</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>questions</td>
<td>200</td>
<td>1500</td>
<td>5</td>
<td>3</td>
<td>0.2</td>
</tr>
<tr>
<td>ogbn-arxiv</td>
<td>2000</td>
<td>500</td>
<td>7</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>ogbn-products</td>
<td>1000</td>
<td>500</td>
<td>10</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>pokec</td>
<td>2000</td>
<td>500</td>
<td>7</td>
<td>2</td>
<td>0.2</td>
</tr>
</tbody>
</table>
Like the baseline models, we set the hidden dimension to 512 and the learning rate to 0.001. Additionally, we introduce a warm-up stage dedicated to training the local module. This step ensures that the node representations generated by the local module capture meaningful graph structural information before being passed to the global module. Moreover, we leverage 8 heads for our attention modules. For the ogbn-products and pokec datasets, we use mini-batch training with a batch size of 100,000 and 550,000 respectively, while for the other datasets, we employ full batch training. Besides, we disable the local attention module on ogbn-arxiv and pokec datasets by replacing GAT (Veličković et al., 2017) with GCN (Kipf & Welling, 2016), which we empirically observe perform better with lower memory usage. In Table 5 we provide the critical hyperparameters of Polynormer used with each dataset.

Additional implementation details. We can rewrite Equation 7 as Equation 16:

$$X = H \odot AV + \sigma(1\beta^T) \odot AV \tag{16}$$

To improve training stability, we adopt a LayerNorm on $H \odot AV$. Besides, we empirically find that scaling LayerNorm($H \odot AV$) by $1 - \sigma(1\beta^T)$ typically improves model accuracy. Thus, we implement the local attention module as shown in Equation 17 and the global attention module follows a similar way:

$$X = (1 - \sigma(1\beta^T)) \odot \text{LayerNorm}(H \odot AV) + \sigma(1\beta^T) \odot AV \tag{17}$$

Appendix I provides the basic Polynormer implementation. We refer readers to our source code for the detailed model implementation and hyperparameter settings.

I Model Implementation

```python
# N: the number of nodes
# M: the number of edges
# D: the node feature dimension
# L1: the number of local attention layers
# L2: the number of global attention layers
# x: input node feature matrix with shape [N, D]
# edge_index: input graph structure with shape [2, M]
# local_convs: local attention layers (e.g., GATConv from PyG)
# local_betas: trainable weights with shape [L1, D]
# global_convs: global attention layers (implemented in Code 2)
# global_betas: trainable weights with shape [L2, D]

# equivariant local attention module
x_local = 0
for i, local_conv in enumerate(local_convs):
    h = h_lins[i](x)
    beta = F.sigmoid(local_betas[i]).unsqueeze(0)
    x = local_conv(x, edge_index) * (h + beta)
    x_local += x

# equivariant global attention module
x = x_local
for i, global_conv in enumerate(global_convs):
    g = g_lins[i](x)
    beta = F.sigmoid(global_betas[i]).unsqueeze(0)
    x = global_conv(x) * (g + beta)

# output linear layer
out = pred_lin(x)

# negative log-likelihood loss calculation
y_pred = F.log_softmax(out, dim=1)
loss = criterion(y_pred[train_idx], y_true[train_idx])
```

Code 1: PyTorch-style Pseudocode for Polynormer
# N: the number of nodes  
# M: the number of edges  
# D: the node feature dimension  
# H: the number of attention heads  
# x: node feature matrix reshaped to [N, D]  

# linear projections to get query, key, value matrices  
\[ q = q_{\text{lins}[i]}(x) \]  
\[ k = k_{\text{lins}[i]}(x) \]  
\[ v = v_{\text{lins}[i]}(x) \]  

# map q and k to (0, 1) for kernel approximation  
\[ q = F.\text{sigmoid}(q) \]  
\[ k = F.\text{sigmoid}(k) \]  

# numerator  
\[ kv = \text{torch.einsum}(\text{\textquotesingle}nmh, ndh \rightarrow mdh\text{\textquotesingle}, k, v) \]  
\[ \text{num} = \text{torch.einsum}(\text{\textquotesingle}nmh, ndh \rightarrow ndh\text{\textquotesingle}, q, kv) \]  

# denominator  
\[ k_{\text{sum}} = \text{torch.einsum}(\text{\textquotesingle}ndh \rightarrow dh\text{\textquotesingle}, k) \]  
\[ \text{den} = \text{torch.einsum}(\text{\textquotesingle}ndh, dh \rightarrow nh\text{\textquotesingle}, q, k_{\text{sum}}).\text{unsqueeze}(1) \]  

# linear global attention based on kernel trick  
\[ x = \frac{\text{num}}{\text{den}} \]  
\[ x = x.\text{reshape}(N, D) \]  
\[ x = \text{LayerNorm}(x) \]

---

**Code 2: PyTorch-style Pseudocode for Linear Global Attention Layer**

### Scalability Analysis of Polynormer

#### J.1 Profiling Results on Synthetic Graphs

![Polynormer Memory Usage vs. Graph Size](image1)

![Polynormer Training Time vs. Graph Size](image2)

**Figure 5:** GPU memory usage and training time of Polynormer on synthetic graphs.

To validate the linear complexity of Polynormer with respect to the number of nodes/edges in a graph, we generate synthetic Erdos-Renyi (ER) graphs with node count ranging from 1,000 to 200,000. We control the edge probability in the ER model to achieve an average node degree of around 5 and set the node feature dimension to 100. Figure 5(left) clearly shows that the memory usage of Polynormer linearly increases with graph size, confirming its linear space complexity. As for the time complexity of Polynormer, we observe in Figure 5(right) that the GPU training time remains nearly constant when increasing the graph size. We attribute it to that the GPU device (RTX A6000) used in our experiments can efficiently handle parallel computations on graphs with different sizes for full batch training. This observation is further supported by the CPU training...
time results, which linearly increase with graph size, as the CPU is less powerful than the GPU for parallel computation.

### J.2 Profiling Results on Large Realistic Graphs

Table 6: Training time and GPU memory usage on large graphs — We underline the results obtained via full batch training, and highlight the top first, second, and third results per dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>ogbn-arxiv</th>
<th></th>
<th>ogbn-products</th>
<th></th>
<th>pokec</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train/Epoch (s)</td>
<td>Mem. (GB)</td>
<td>Train/Epoch (s)</td>
<td>Mem. (GB)</td>
<td>Train/Epoch (s)</td>
<td>Mem. (GB)</td>
</tr>
<tr>
<td>GAT</td>
<td>0.16</td>
<td>10.64</td>
<td>0.97</td>
<td>9.04</td>
<td>1.36</td>
<td>9.57</td>
</tr>
<tr>
<td>GraphGPS</td>
<td>1.32</td>
<td>38.91</td>
<td></td>
<td></td>
<td>OOM</td>
<td></td>
</tr>
<tr>
<td>NAGphormer</td>
<td>4.26</td>
<td>5.15</td>
<td>9.64</td>
<td>7.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exphormer</td>
<td>0.74</td>
<td>34.04</td>
<td></td>
<td></td>
<td>OOM</td>
<td></td>
</tr>
<tr>
<td>NodeFormer</td>
<td>1.20</td>
<td>16.30</td>
<td>3.37</td>
<td>31.55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIFFormer</td>
<td>0.77</td>
<td>24.51</td>
<td>1.50</td>
<td>16.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOAT</td>
<td>12.32</td>
<td>6.98</td>
<td>20.11</td>
<td>9.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polynomial</td>
<td>0.31</td>
<td>16.09</td>
<td>3.13</td>
<td>12.93</td>
<td>2.64</td>
<td>21.54</td>
</tr>
</tbody>
</table>

Table 6 shows the profiling results of GAT, linear GTs, and Polynomial in terms of training time per epoch and memory usage on large realistic graphs. Note that we perform full batch training for all models whenever possible, since it avoids the nontrivial overhead associated with graph sampling in mini-batch training.

The results demonstrate that Polynomial consistently ranks among the top 3 fastest models, with relatively low memory usage. We attribute this efficiency advantage to the implementation of Polynomial that only involves common and highly-optimized built-in APIs from modern graph learning frameworks (e.g., PyG). In contrast, the GT baselines incorporate less common or expensive compute kernels (e.g., complicated kernel functions, Gumbel-Softmax, and nearest neighbor search), making them more challenging for graph learning frameworks to accelerate. Thus, we believe Polynomial is reasonably efficient and scalable in practice.

### K WL Expressivity of Polynomial

While we have shown that Polynomial is polynomial-expressive, it is also interesting to see its expressivity under the Weisfeiler-Lehman (WL) hierarchy. To this end, we are going to firstly show that Polynomial is at least as expressive as 1-WL GNNs, and then introduce a simple way that renders Polynomial strictly more expressive than 1-WL GNNs.

For the sake of illustration, let us revisit Equation 4 in the following:

\[
X = (WX) \odot (X + B)
\]  

(18)

where \( W \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times d} \) are trainable weight matrices, and \( X \in \mathbb{R}^{n \times d} \) represents node features. Suppose all nodes have identical features (followed by the original formulation of WL algorithm), if we replace \( B = \mathbf{1} \beta^T \), the term \( X + \mathbf{1} \beta^T \) is essentially a constant feature vector shared across nodes. As a result, the term \( (WX) \odot (X + \mathbf{1} \beta^T) \) is reduced to \( WX \). By properly designing the weight matrix \( W \) (e.g. the adjacency matrix with self-loop), the term \( WX \) can be reduced to 1-WL GNNs. Hence, Polynomial based on Equation 18 is at least as expressive as 1-WL GNNs.

To make Polynomial strictly more expressive than 1-WL GNNs, let us focus on how to properly design the weight matrix \( B \). Previously, we set \( B = \mathbf{1} \beta^T \). Note that the constant vector \( \mathbf{1} \in \mathbb{R}^n \) here is essentially the eigenvector \( v_1 \) that corresponds to the smallest (normalized) graph Laplacian eigenvalue. This motivates us to design \( B = v_2 \beta^T \), where \( v_2 \) denotes the eigenvector corresponding to the second smallest (i.e., first non-trivial) eigenvalue of the normalized Laplacian. Consequently, Polynomial can be built upon the following Equation:

\[
X = (WX) \odot (X + v_2 \beta^T)
\]  

(19)
Notably, the vector $v_2$ essentially encodes the node positional information from graph spectrum into polynomial coefficients $B$, which makes each output node feature unique in Equation 19 and thus allows Polynormer to distinguish non-isomorphic graphs that 1-WL GNNs fail to distinguish, as empirically confirmed in Figure 6. It is noteworthy that this approach is fundamentally different from prior PE methods based on Laplacian eigenvectors. Specifically, prior PE methods primarily focus on concatenating node positional encodings with node features. In contrast, we incorporate the positional information into the polynomial coefficients learned by Polynormer.

To distinguish the aforementioned two options of designing the weight matrix $B$, we denote the Polynormer of using the constant vector $v_1$ and the second Laplacian vector $v_2$ by Polynormer-v1 and Polynormer-v2, respectively. In the following, we empirically show that Polynormer-v2 is more powerful than 1-WL GNNs on distinguishing non-isomorphic graphs.

**Experimental setup.** Without loss of generality, we focus on the local attention layer of Polynormer, and set $\beta$ as well as initial node features to be 1. As the attention scores between identical features are not meaningful, we replace the sparse attention matrix introduced in Section 3.2 with the random walk matrix $\hat{A} = AD^{-1}$, where $A$ and $D$ denote the adjacency and degree matrices, respectively.

**Experimental results.** As shown in Figure 6, Polynormer-v2 is able to distinguish non-isomorphic graphs such as circular skip link (CSL) graphs, which are known to be indistinguishable by the 1-WL test as well as 1-WL GNNs Sato (2020); Rampasek et al. (2022); Dwivedi et al. (2022).

![Figure 6: Two example pairs of non-isomorphic graphs from Sato (2020) that cannot be distinguished by 1-WL test. Our Polynormer-v2 can distinguish them.](image)

While Polynormer-v2 is strictly more powerful than the 1-WL algorithm, our empirically results in Table 7 indicate that it does not clearly outperform Polynormer-v1 on realistic node classification datasets. One possible reason is that each node possesses unique features in realistic datasets, which diminishes the importance of incorporating additional node positional information into Polynormer. Hence, we implement Polynormer based on Polynormer-v1 in practice to avoid computing the Laplacian eigenvector, and leave the improvement of Polynormer-v2 to our future work.
Table 7: Comparison of Polynormer-v1 and Polynormer-v2.

<table>
<thead>
<tr>
<th></th>
<th>Computer</th>
<th>Photo</th>
<th>WikiCS</th>
<th>roman-empire</th>
<th>amazon-ratings</th>
<th>minesweeper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynormer-v1</td>
<td>93.18 ± 0.18</td>
<td>96.11 ± 0.23</td>
<td>79.53 ± 0.83</td>
<td>92.13 ± 0.50</td>
<td>54.46 ± 0.40</td>
<td>96.96 ± 0.52</td>
</tr>
<tr>
<td>Polynormer-v2</td>
<td>93.25 ± 0.22</td>
<td>96.09 ± 0.30</td>
<td>80.01 ± 0.55</td>
<td>91.33 ± 0.39</td>
<td>54.41 ± 0.35</td>
<td>96.48 ± 0.37</td>
</tr>
</tbody>
</table>

L. CASE STUDY: AI MODEL RUNTIME PREDICTION ON TPU

Apart from comparing Polynormer against conventional GRL models on mainstream graph datasets, we further evaluate it on TpuGraphs (Phothilimthana et al., 2024), a large-scale runtime prediction dataset on tensor computation graphs: [github.com/google-research-datasets/tpu_graphs](https://github.com/google-research-datasets/tpu_graphs)

**Dataset details.** Unlike prior datasets for program runtime prediction that are relatively small (up to 100 nodes), TpuGraphs is a runtime prediction dataset on full tensor programs, represented as computation graphs. Each graph within the dataset represents the primary computation of an ML program, typically encompassing one or more training steps or a single inference step. These graphs are obtained from open-source ML programs and include well-known models such as ResNet, EfficientNet, Mask R-CNN, and various Transformer models designed for diverse tasks like vision, natural language processing, speech, audio, recommender systems, and generative AI. Each data sample in the dataset comprises a computational graph, a compilation configuration, and the corresponding execution time when the graph is compiled with the specified configuration on a TPU v3, an accelerator tailored for ML workloads. The compilation configuration governs how the XLA (Accelerated Linear Algebra) compiler transforms the graph through specific optimization passes. The TpuGraphs dataset is composed of two main categories based on the compiler optimization level: (1) TpuGraphs-Layout (layout optimization) and (2) TpuGraphs-Tile (tiling optimization). Layout configurations dictate how tensors are organized in physical memory, determining the dimension order for each input and output of an operation node. Notably, TpuGraphs-Layout has 4 collections based on ML model type and compiler configuration as follows:

- **ML model type:**
  - NLP: computation graphs of BERT models,
  - XLA: computation graphs of ML models from various domains such as vision, NLP, speech, audio, and recommendation.

- **Compiler configuration:**
  - Default: configurations picked by XLA compiler’s heuristic.
  - Random: randomly picked configurations.

On the other hand, tile configurations control the tile size of each fused subgraph. The TpuGraphs-Layout comprises 31 million pairs of graphs and configurations, with an average of over 7,700 nodes per graph. In contrast, the TpuGraphs-Tile includes 13 million pairs of kernels and configurations, averaging 40 nodes per kernel subgraph.

L.1 RESULTS ON TpuGRAPHS-Layout Collections.

As shown in Table 8, we compare Polynormer against the GraphSAGE baseline provided by Google [Hamilton et al., 2017]. To ensure a fair comparison, we only change the model architecture while keeping all other configurations the same (i.e., no additional feature engineering). Experimental results show that Polynormer/Polynormer-local outperforms GraphSAGE on all collections. In particular, Polynormer achieves 10.3% accuracy improvement over GraphSAGE and 6.3% accuracy improvement over Polynormer-local on the XLA-Random collection, which is the most challenging collection since XLA consists of diverse graph structures from different domains and Random contains various compiler configurations. This showcases that the global attention of Polynormer effectively captures critical global structures in different graphs, owing to its high polynomial expressivity.

L.2 RESULTS ON TpuGRAPHS (Layout + Tile).

We further evaluate Polynormer on the whole TpuGraphs dataset that contains Layout and Tile optimizations. Table 9 shows that the local attention module alone in Polynormer is able to outperform
Table 8: Ordered pair accuracy (%) on TpuGraphs-Layout collections — “Polynormer-Local” denotes the local attention module in Polynormer.

<table>
<thead>
<tr>
<th>GraphSAGE</th>
<th>Polynormer-Local</th>
<th>Polynormer</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLP-Default</td>
<td>81.2</td>
<td>82.0</td>
</tr>
<tr>
<td>NLP-Random</td>
<td>93.0</td>
<td>94.1</td>
</tr>
<tr>
<td>XLA-Default</td>
<td>77.3</td>
<td>73.7</td>
</tr>
<tr>
<td>XLA-Random</td>
<td>76.3</td>
<td>81.3</td>
</tr>
</tbody>
</table>

Table 9: Averaged runtime prediction accuracy (%) on TpuGraphs — “Polynormer-Local” denotes the local attention module in Polynormer.

<table>
<thead>
<tr>
<th>TpuGraphs (Layout+Tile)</th>
<th>GraphSAGE</th>
<th>Polynormer-Local</th>
<th>Polynormer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30.1</td>
<td>49.8</td>
<td>61.9</td>
</tr>
</tbody>
</table>

the baseline model by 19.6%, indicating the efficacy of our polynomial-expressive architecture for capturing critical local structures. Moreover, the accuracy of Polynormer gets further improved when adding the global attention module, leading to a 31.8% accuracy improvement over the baseline model. This confirms the effectiveness of the proposed local-to-global attention scheme.

M Discussion on the Definition of Polynomial Expressivity

As graph learning models primarily learn node representations by aggregating input features from a set of nodes, they can be viewed as a multiset pooling problem with auxiliary information about the graph topology (Baek et al., 2021; Hua et al., 2022). In the context of polynomial networks on graphs, we follow a similar notion by focusing on polynomial functions that map input node features into output node representations, with the graph topology encoded into polynomial coefficients. Notably, such polynomials have been explored by prior studies Hua et al. (2022); Chrysos et al. (2020, 2022); Wang et al. (2021), which inspire us to formally provide Definition 3.1. Moreover, assessing the ability to learn high-degree polynomials is well motivated based on the Weierstrass theorem, which guarantees any smooth function can be approximated by a polynomial Stone (1948). Thus, models with higher polynomial expressivity are more capable of learning complex functions.

While Definition 3.1 does not explicitly mention graph topology, the utilization of graph topology in different graph learning models affects the inclusion of distinct monomial terms in the learned polynomial function, thereby impacting the polynomial expressivity. It is also worth noting that Definition 3.1 can be viewed as a simplified version of the definition on polynomial expressivity in Puny et al. (2023), since we consider graph structural information as polynomial coefficients rather than indeterminates. This simplification allows us to decouple the analysis on node features from graph topology, which renders us to develop a scalable model with high polynomial expressivity.