Do Transformers Really Perform Bad for Graph Representation?

Chengxuan Ying[1]*, **Tianle Cai**[2], **Shengjie Luo**[3]*,
Shuxin Zheng[4]†, **Guolin Ke**[4], **Di He**[4]†, **Yanming Shen**[1], **Tie-Yan Liu**[4]
[1]Dalian University of Technology [2]Princeton University
[3]Peking University [4]Microsoft Research Asia
yingchengsyuan@gmail.com, tianle.cai@princeton.edu, luosj@stu.pku.edu.cn
{shuz†, guoke, dihe†, tyliu}@microsoft.com, shen@dlut.edu.cn

Abstract

The Transformer architecture has become a dominant choice in many domains, such as natural language processing and computer vision. Yet, it has not achieved competitive performance on popular leaderboards of graph-level prediction compared to mainstream GNN variants. Therefore, it remains a mystery how Transformers could perform well for graph representation learning. In this paper, we solve this mystery by presenting Graphormer, which is built upon the standard Transformer architecture, and could attain excellent results on a broad range of graph representation learning tasks, especially on the recent OGB Large-Scale Challenge. Our key insight to utilizing Transformer in the graph is the necessity of effectively encoding the structural information of a graph into the model. To this end, we propose several simple yet effective structural encoding methods to help Graphormer better model graph-structured data. Besides, we mathematically characterize the expressive power of Graphormer and exhibit that with our ways of encoding the structural information of graphs, many popular GNN variants could be covered as the special cases of Graphormer. The code and models of Graphormer will be made publicly available at https://github.com/Microsoft/Graphormer

1 Introduction

The Transformer [49] is well acknowledged as the most powerful neural network in modelling sequential data, such as natural language [11, 35, 6] and speech [17]. Model variants built upon Transformer have also been shown great performance in computer vision [12, 36] and programming language [19, 63, 44]. However, to the best of our knowledge, Transformer has still not been the de-facto standard on public graph representation leaderboards [22, 14, 21]. There are many attempts of leveraging Transformer into the graph domain, but the only effective way is replacing some key modules (e.g., feature aggregation) in classic GNN variants by the softmax attention [50, 7, 23, 51, 61, 46, 13]. Therefore, it is still an open question whether Transformer architecture is suitable to model graphs and how to make it work in graph representation learning.

In this paper, we give an affirmative answer by developing Graphormer, which is directly built upon the standard Transformer, and achieves state-of-the-art performance on a wide range of graph-level prediction tasks, including the very recent Open Graph Benchmark Large-Scale Challenge (OGB-LSC) [21], and several popular leaderboards (e.g., OGB [22], Benchmarking-GNN [14]). The Transformer is originally designed for sequence modeling. To utilize its power in graphs, we believe

*Interns at MSRA.
†Corresponding authors.

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

the key is to properly incorporate structural information of graphs into the model. Note that for each node i, the self-attention only calculates the semantic similarity between i and other nodes, without considering the structural information of a graph reflected on the nodes and the relation between node pairs. Graphormer incorporates several effective structural encoding methods to leverage such information, which are described below.

First, we propose a *Centrality Encoding* in Graphormer to capture the node importance in the graph. In a graph, different nodes may have different importance, e.g., celebrities are considered to be more influential than the majority of web users in a social network. However, such information isn't reflected in the self-attention module as it calculates the similarities mainly using the node semantic features. To address the problem, we propose to encode the node centrality in Graphormer. In particular, we leverage the *degree centrality* for the centrality encoding, where a learnable vector is assigned to each node according to its degree and added to the node features in the input layer. Empirical studies show that simple centrality encoding is effective for Transformer in modeling the graph data.

Second, we propose a novel *Spatial Encoding* in Graphormer to capture the structural relation between nodes. One notable geometrical property that distinguishes graph-structured data from other structured data, e.g., language, images, is that there does not exist a canonical grid to embed the graph. In fact, nodes can only lie in a non-Euclidean space and are linked by edges. To model such structural information, for each node pair, we assign a learnable embedding based on their spatial relation. Multiple measurements in the literature could be leveraged for modeling spatial relations. For a general purpose, we use the distance of the shortest path between any two nodes as a demonstration, which will be encoded as a bias term in the softmax attention and help the model accurately capture the spatial dependency in a graph. In addition, sometimes there is additional spatial information contained in edge features, such as the type of bond between two atoms in a molecular graph. We design a new edge encoding method to further take such signal into the Transformer layers. To be concrete, for each node pair, we compute an average of dot-products of the edge features and learnable embeddings along the shortest path, then use it in the attention module. Equipped with these encodings, Graphormer could better model the relationship for node pairs and represent the graph.

By using the proposed encodings above, we further mathematically show that Graphormer has strong expressiveness as many popular GNN variants are just its special cases. The great capacity of the model leads to state-of-the-art performance on a wide range of tasks in practice. On the large-scale quantum chemistry regression dataset[3] in the very recent Open Graph Benchmark Large-Scale Challenge (OGB-LSC) [21], Graphormer outperforms most mainstream GNN variants by more than 10% points in terms of the relative error. On other popular leaderboards of graph representation learning (e.g., MolHIV, MolPCBA, ZINC) [22, 14], Graphormer also surpasses the previous best results, demonstrating the potential and adaptability of the Transformer architecture.

2 Preliminary

In this section, we recap the preliminaries in Graph Neural Networks and Transformer.

Graph Neural Network (GNN). Let $G = (V, E)$ denote a graph where $V = \{v_1, v_2, \cdots, v_n\}$, $n = |V|$ is the number of nodes. Let the feature vector of node v_i be x_i. GNNs aim to learn representation of nodes and graphs. Typically, modern GNNs follow a learning schema that iteratively updates the representation of a node by aggregating representations of its first or higher-order neighbors. We denote $h_i^{(l)}$ as the representation of v_i at the l-th layer and define $h_i^{(0)} = x_i$. The l-th iteration of aggregation could be characterized by AGGREGATE-COMBINE step as

$$a_i^{(l)} = \text{AGGREGATE}^{(l)}\left(\left\{h_j^{(l-1)} : j \in \mathcal{N}(v_i)\right\}\right), \quad h_i^{(l)} = \text{COMBINE}^{(l)}\left(h_i^{(l-1)}, a_i^{(l)}\right), \quad (1)$$

where $\mathcal{N}(v_i)$ is the set of first or higher-order neighbors of v_i. The AGGREGATE function is used to gather the information from neighbors. Common aggregation functions include MEAN, MAX, SUM, which are used in different architectures of GNNs [26, 18, 50, 54]. The goal of COMBINE function is to fuse the information from neighbors into the node representation.

[3] https://ogb.stanford.edu/kddcup2021/pcqm4m/

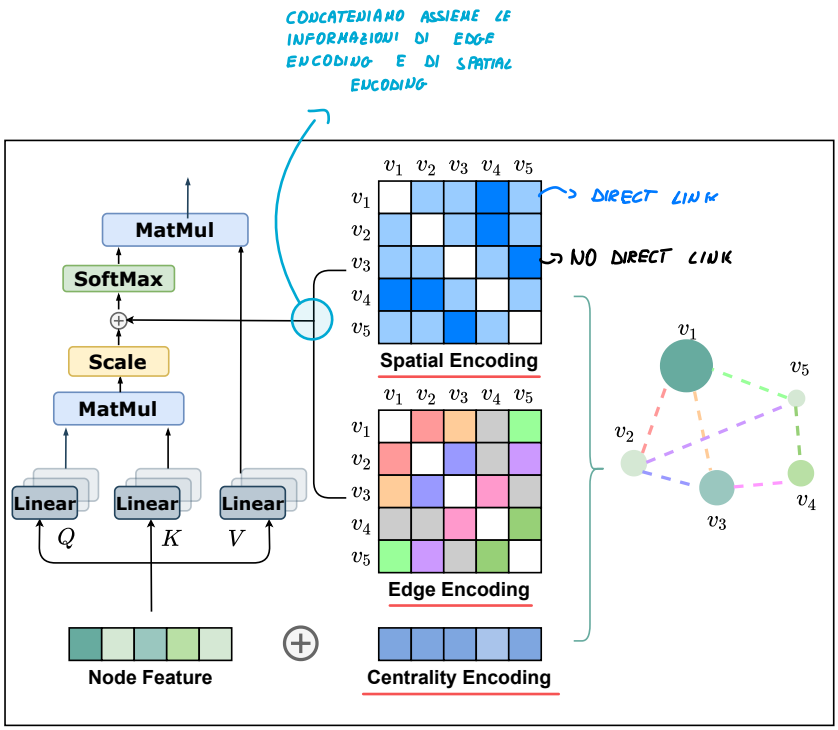


Figure 1: An illustration of our proposed centrality encoding, spatial encoding, and edge encoding in Graphormer.

In addition, for graph representation tasks, a READOUT function is designed to aggregate node features $h_i^{(L)}$ of the final iteration into the representation h_G of the entire graph G:

$$h_G = \text{READOUT}\left(\left\{h_i^{(L)} \mid v_i \in G\right\}\right). \tag{2}$$

READOUT can be implemented by a simple permutation invariant function such as summation [54] or a more sophisticated graph-level pooling function [1].

Transformer. The Transformer architecture consists of a composition of Transformer layers [49]. Each Transformer layer has two parts: a self-attention module and a position-wise feed-forward network (FFN). Let $H = [h_1^\top, \cdots, h_n^\top]^\top \in \mathbb{R}^{n \times d}$ denote the input of self-attention module where d is the hidden dimension and $h_i \in \mathbb{R}^{1 \times d}$ is the hidden representation at position i. The input H is projected by three matrices $W_Q \in \mathbb{R}^{d \times d_K}$, $W_K \in \mathbb{R}^{d \times d_K}$ and $W_V \in \mathbb{R}^{d \times d_V}$ to the corresponding representations Q, K, V. The self-attention is then calculated as:

$$Q = HW_Q, \quad K = HW_K, \quad V = HW_V, \tag{3}$$

$$A = \frac{QK^\top}{\sqrt{d_K}}, \quad \text{Attn}(H) = \text{softmax}(A)V, \tag{4}$$

where A is a matrix capturing the similarity between queries and keys. For simplicity of illustration, we consider the single-head self-attention and assume $d_K = d_V = d$. The extension to the multi-head attention is standard and straightforward, and we omit bias terms for simplicity.

3 Graphormer

In this section, we present our Graphormer for graph tasks. First, we elaborate on several key designs in the Graphormer, which serve as an inductive bias in the neural network to learn the graph representation. We further provide the detailed implementations of Graphormer. Finally, we show that our proposed Graphormer is more powerful since popular GNN models [26, 54, 18] are its special cases.

3.1 Structural Encodings in Graphormer

As discussed in the introduction, it is important to develop ways to leverage the structural information of graphs into the Transformer model. To this end, we present three simple but effective designs of encoding in Graphormer. See Figure 1 for an illustration.

3.1.1 Centrality Encoding

In Eq.4, the attention distribution is calculated based on the semantic correlation between nodes. However, node centrality, which measures how important a node is in the graph, is usually a strong signal for graph understanding. For example, celebrities who have a huge number of followers are important factors in predicting the trend of a social network [40, 39]. Such information is neglected in the current attention calculation, and we believe it should be a valuable signal for Transformer models.

In Graphormer, we use the degree centrality, which is one of the standard centrality measures in literature, as an additional signal to the neural network. To be specific, we develop a *Centrality Encoding* which assigns each node two real-valued embedding vectors according to its indegree and outdegree. As the centrality encoding is applied to each node, we simply add it to the node features as the input.

$$h_i^{(0)} = x_i + z^-_{\deg^-(v_i)} + z^+_{\deg^+(v_i)}, \tag{5}$$

where $z^-, z^+ \in \mathbb{R}^d$ are learnable embedding vectors specified by the indegree $\deg^-(v_i)$ and outdegree $\deg^+(v_i)$ respectively. For undirected graphs, $\deg^-(v_i)$ and $\deg^+(v_i)$ could be unified to $\deg(v_i)$. By using the centrality encoding in the input, the softmax attention can catch the node importance signal in the queries and the keys. Therefore the model can capture both the semantic correlation and the node importance in the attention mechanism.

3.1.2 Spatial Encoding

An advantage of Transformer is its global receptive field. In each Transformer layer, each token can attend to the information at any position and then process its representation. But this operation has a byproduct problem that the model has to explicitly specify different positions or encode the positional dependency (such as locality) in the layers. For sequential data, one can either give each position an embedding (i.e., absolute positional encoding [49]) as the input or encode the relative distance of any two positions (i.e., relative positional encoding [45, 47]) in the Transformer layer.

However, for graphs, nodes are not arranged as a sequence. They can lie in a multi-dimensional spatial space and are linked by edges. To encode the structural information of a graph in the model, we propose a novel *Spatial Encoding*. Concretely, for any graph G, we consider a function $\phi(v_i, v_j) : V \times V \to \mathbb{R}$ which measures the spatial relation between v_i and v_j in graph G. The function ϕ can be defined by the connectivity between the nodes in the graph. In this paper, we choose $\phi(v_i, v_j)$ to be the distance of the shortest path (SPD) between v_i and v_j if the two nodes are connected. If not, we set the output of ϕ to be a special value, i.e., -1. We assign each (feasible) output value a learnable scalar which will serve as a bias term in the self-attention module. Denote A_{ij} as the (i, j)-element of the Query-Key product matrix A, we have:

$$A_{ij} = \frac{(h_i W_Q)(h_j W_K)^T}{\sqrt{d}} + b_{\phi(v_i, v_j)}, \tag{6}$$

where $b_{\phi(v_i,v_j)}$ is a learnable scalar indexed by $\phi(v_i, v_j)$, and shared across all layers.

Here we discuss several benefits of our proposed method. First, compared to conventional GNNs described in Section 2, where the receptive field is restricted to the neighbors, we can see that in Eq. (6), the Transformer layer provides a global information that each node can attend to all other nodes in the graph. Second, by using $b_{\phi(v_i,v_j)}$, each node in a single Transformer layer can adaptively attend to all other nodes according to the graph structural information. For example, if $b_{\phi(v_i,v_j)}$ is learned to be a decreasing function with respect to $\phi(v_i, v_j)$, for each node, the model will likely pay more attention to the nodes near it and pay less attention to the nodes far away from it.

3.1.3 Edge Encoding in the Attention

In many graph tasks, edges also have structural features, e.g., in a molecular graph, atom pairs may have features describing the type of bond between them. Such features are important to the graph representation, and encoding them together with node features into the network is essential. There are mainly two edge encoding methods used in previous works. In the first method, the edge features are added to the associated nodes' features [22, 30]. In the second method, for each node, its associated edges' features will be used together with the node features in the aggregation [15, 54, 26]. However, such ways of using edge feature only propagate the edge information to its associated nodes, which may not be an effective way to leverage edge information in representation of the whole graph.

L'IDEA QUI È QUELLA DI AGGIUNGERE UN ULTERIORE ADDENDO AL CALCOLO DELL'ATTENZIONE CHE TENGA CONTO DEGLI EDGE FEATURES

To better encode edge features into attention layers, we propose a new edge encoding method in Graphormer. The attention mechanism needs to estimate correlations for each node pair (v_i, v_j), and we believe the edges connecting them should be considered in the correlation as in [34, 51]. For each ordered node pair (v_i, v_j), we find (one of) the shortest path $\text{SP}_{ij} = (e_1, e_2, ..., e_N)$ from v_i to v_j, and compute an average of the dot-products of the edge feature and a learnable embedding along the path. The proposed edge encoding incorporates edge features via a bias term to the attention module. Concretely, we modify the (i, j)-element of A in Eq. (3) further with the edge encoding c_{ij} as:

$$A_{ij} = \frac{(h_i W_Q)(h_j W_K)^T}{\sqrt{d}} + b_{\phi(v_i, v_j)} + c_{ij}, \quad \text{where } c_{ij} = \frac{1}{N} \sum_{n=1}^{N} x_{e_n} (w_n^E)^T, \tag{7}$$

EDGE FEATURE

WEIGHT EMBEDDING

where x_{e_n} is the feature of the n-th edge e_n in SP_{ij}, $w_n^E \in \mathbb{R}^{d_E}$ is the n-th weight embedding, and d_E is the dimensionality of edge feature.

3.2 Implementation Details of Graphormer

Graphormer Layer. Graphormer is built upon the original implementation of classic Transformer encoder described in [49]. In addition, we apply the layer normalization (LN) before the multi-head self-attention (MHA) and the feed-forward blocks (FFN) instead of after [53]. This modification has been unanimously adopted by all current Transformer implementations because it leads to more effective optimization [43]. Especially, for FFN sub-layer, we set the dimensionality of input, output, and the inner-layer to the same dimension with d. We formally characterize the Graphormer layer as below:

$$h^{'(l)} = \text{MHA}(\text{LN}(h^{(l-1)})) + h^{(l-1)} \tag{8}$$

$$h^{(l)} = \text{FFN}(\text{LN}(h^{'(l)})) + h^{'(l)} \tag{9}$$

Special Node. As stated in the previous section, various graph pooling functions are proposed to represent the graph embedding. Inspired by [15], in Graphormer, we add a special node called `[VNode]` to the graph, and make connection between `[VNode]` and each node individually. In the AGGREGATE-COMBINE step, the representation of `[VNode]` has been updated as normal nodes in graph, and the representation of the entire graph h_G would be the node feature of `[VNode]` in the final layer. In the BERT model [11, 35], there is a similar token, i.e., `[CLS]`, which is a special token attached at the beginning of each sequence, to represent the sequence-level feature on downstream tasks. While the `[VNode]` is connected to all other nodes in graph, which means the distance of the shortest path is 1 for any $\phi(\texttt{[VNode]}, v_j)$ and $\phi(v_i, \texttt{[VNode]})$, the connection is not physical. To distinguish the connection of physical and virtual, inspired by [25], we reset all spatial encodings for $b_{\phi(\texttt{[VNode]}, v_j)}$ and $b_{\phi(v_i, \texttt{[VNode]})}$ to a distinct learnable scalar.

3.3 How Powerful is Graphormer?

In the previous subsections, we introduce three structural encodings and the architecture of Graphormer. Then a natural question is: *Do these modifications make Graphormer more powerful than other GNN variants?* In this subsection, we first give an affirmative answer by showing that Graphormer can represent the AGGREGATE and COMBINE steps in popular GNN models:

NB

Fact 1. *By choosing proper weights and distance function ϕ, the Graphormer layer can represent AGGREGATE and COMBINE steps of popular GNN models such as GIN, GCN, GraphSAGE.*

The proof sketch to derive this result is: 1) Spatial encoding enables self-attention module to distinguish neighbor set $\mathcal{N}(v_i)$ of node v_i so that the softmax function can calculate mean statistics over $\mathcal{N}(v_i)$; 2) Knowing the degree of a node, mean over neighbors can be translated to sum over neighbors; 3) With multiple heads and FFN, representations of v_i and $\mathcal{N}(v_i)$ can be processed separately and combined together later. We defer the proof of this fact to Appendix A.

Moreover, we show further that by using our spatial encoding, Graphormer can go beyond classic message passing GNNs whose expressive power is no more than the 1-Weisfeiler-Lehman (WL) test. We give a concrete example in Appendix A to show how Graphormer helps distinguish graphs that the 1-WL test fails to.

Connection between Self-attention and Virtual Node. Besides the superior expressiveness than popular GNNs, we also find an interesting connection between using self-attention and the virtual node heuristic [15, 31, 24, 22]. As shown in the leaderboard of OGB [22], the virtual node trick, which augments graphs with additional supernodes that are connected to all nodes in the original graphs, can significantly improve the performance of existing GNNs. Conceptually, the benefit of the virtual node is that it can aggregate the information of the *whole graph* (like the READOUT function) and then propagate it to *each node*. However, a naive addition of a supernode to a graph can potentially lead to inadvertent over-smoothing of information propagation [24]. We instead find that such a graph-level aggregation and propagation operation can be naturally fulfilled by vanilla self-attention without additional encodings. Concretely, we can prove the following fact:

READOUT?

Fact 2. *By choosing proper weights, every node representation of the output of a Graphormer layer without additional encodings can represent MEAN READOUT functions.*

This fact takes the advantage of self-attention that each node can attend to all other nodes. Thus it can simulate graph-level READOUT operation to aggregate information from the whole graph. Besides the theoretical justification, we empirically find that Graphormer does not encounter the problem of over-smoothing, which makes the improvement scalable. The fact also inspires us to introduce a special node for graph readout (see the previous subsection).

4 Experiments

We first conduct experiments on the recent OGB-LSC [21] quantum chemistry regression (i.e., PCQM4M-LSC) challenge, which is currently the biggest graph-level prediction dataset and contains more than 3.8M graphs in total. Then, we report the results on the other three popular tasks: ogbg-molhiv, ogbg-molpcba and ZINC, which come from the OGB [22] and benchmarking-GNN [14] leaderboards. Finally, we ablate the important design elements of Graphormer. A detailed description of datasets and training strategies could be found in Appendix B.

4.1 OGB Large-Scale Challenge

Baselines. We benchmark the proposed Graphormer with GCN [26] and GIN [54], and their variants with virtual node (-VN) [15]. They achieve the state-of-the-art valid and test mean absolute error (MAE) on the official leaderboard[4] [21]. In addition, we compare to GIN's multi-hop variant [5], and 12-layer deep graph network DeeperGCN [30], which also show promising performance on other leaderboards. We further compare our Graphormer with the recent Transformer-based graph model GT [13].

Settings. We primarily report results on two model sizes: **Graphormer** ($L = 12, d = 768$), and a smaller one **Graphormer$_{\text{SMALL}}$** ($L = 6, d = 512$). Both the number of attention heads in the attention module and the dimensionality of edge features d_E are set to 32. We use AdamW as the optimizer, and set the hyper-parameter ϵ to 1e-8 and $(\beta 1, \beta 2)$ to (0.99,0.999). The peak learning rate is set to 2e-4 (3e-4 for **Graphormer$_{\text{SMALL}}$**) with a 60k-step warm-up stage followed by a linear decay learning rate scheduler. The total training steps are 1M. The batch size is set to 1024. All models are trained on 8 NVIDIA V100 GPUS for about 2 days.

[4] https://github.com/snap-stanford/ogb/tree/master/examples/lsc/pcqm4m#performance

Table 1: Results on PCQM4M-LSC. * indicates the results are cited from the official leaderboard [21].

method	#param.	train MAE	validate MAE
GCN [26]	2.0M	0.1318	0.1691 (0.1684*)
GIN [54]	3.8M	0.1203	0.1537 (0.1536*)
GCN-VN [26, 15]	4.9M	0.1225	0.1485 (0.1510*)
GIN-VN [54, 15]	6.7M	0.1150	0.1395 (0.1396*)
GINE-VN [5, 15]	13.2M	0.1248	0.1430
DeeperGCN-VN [30, 15]	25.5M	0.1059	0.1398
GT [13]	0.6M	0.0944	0.1400
GT-Wide [13]	83.2M	0.0955	0.1408
$\text{Graphormer}_{\text{SMALL}}$	12.5M	0.0778	0.1264
Graphormer	47.1M	**0.0582**	**0.1234**

Results. Table 1 summarizes performance comparisons on PCQM4M-LSC dataset. From the table, GIN-VN achieves the previous state-of-the-art validate MAE of 0.1395. The original implementation of GT [13] employs a hidden dimension of 64 to reduce the total number of parameters. For a fair comparison, we also report the result by enlarging the hidden dimension to 768, denoted by GT-Wide, which leads to a total number of parameters of 83.2M. While, both GT and GT-Wide do not outperform GIN-VN and DeeperGCN-VN. Especially, we do not observe a performance gain along with the growth of parameters of GT.

Compared to the previous state-of-the-art GNN architecture, Graphormer noticeably surpasses GIN-VN by a large margin, e.g., 11.5% relative validate MAE decline. By using the ensemble with ExpC [55], we got a 0.1200 MAE on complete test set and won the first place of the graph-level track in OGB Large-Scale Challenge[21, 58]. As stated in Section 3.3, we further find that the proposed Graphormer does not encounter the problem of over-smoothing, i.e., the train and validate error keep going down along with the growth of depth and width of models.

4.2 Graph Representation

In this section, we further investigate the performance of Graphormer on commonly used graph-level prediction tasks of popular leaderboards, i.e., OGB [22] (OGBG-MolPCBA, OGBG-MolHIV), and benchmarking-GNN [14] (ZINC). Since pre-training is encouraged by OGB, we mainly explore the transferable capability of a Graphormer model pre-trained on OGB-LSC (i.e., PCQM4M-LSC). Please note that the model configurations, hyper-parameters, and the pre-training performance of pre-trained Graphormers used for MolPCBA and MolHIV are different from the models used in the previous subsection. Please refer to Appendix B for detailed descriptions. For benchmarking-GNN, which does not encourage large pre-trained model, we train an additional $\text{Graphormer}_{\text{SLIM}}$ ($L = 12, d = 80$, total param.$= 489K$) from scratch on ZINC.

Baselines. We report performance of GNNs which achieve top-performance on the official leaderboards[5] *without additional domain-specific features*. Considering that the pre-trained Graphormer leverages external data, for a fair comparison on OGB datasets, we additionally report performance for fine-tuning GIN-VN pre-trained on PCQM4M-LSC dataset, which achieves the previous state-of-the-art valid and test MAE on that dataset.

Settings. We report detailed training strategies in Appendix B. In addition, Graphormer is more easily trapped in the over-fitting problem due to the large size of the model and the small size of the dataset. Therefore, we employ a widely used data augmentation for graph - FLAG [27], to mitigate the over-fitting problem on OGB datasets.

Results. Table 2, 3 and 4 summarize performance of Graphormer comparing with other GNNs on MolHIV, MolPCBA and ZINC datasets. Especially, GT [13] and SAN [28] in Table 4 are recently proposed Transformer-based GNN models. Graphormer consistently and significantly outperforms previous state-of-the-art GNNs on all three datasets by a large margin. Specially, except Graphormer,

[5] https://ogb.stanford.edu/docs/leader_graphprop/ https://github.com/graphdeeplearning/benchmarking-gnns/blob/master/docs/07_leaderboards.md

Table 2: Results on MolPCBA.

method	#param.	AP (%)
DeeperGCN-VN+FLAG [30]	5.6M	28.42±0.43
DGN [2]	6.7M	28.85±0.30
GINE-VN [5]	6.1M	29.17±0.15
PHC-GNN [29]	1.7M	29.47±0.26
GINE-APPNP [5]	6.1M	29.79±0.30
GIN-VN[54] (fine-tune)	3.4M	29.02±0.17
Graphormer-FLAG	119.5M	**31.39**±0.32

MHHH

Table 3: Results on MolHIV.

method	#param.	AUC (%)
GCN-GraphNorm [5, 8]	526K	78.83±1.00
PNA [10]	326K	79.05±1.32
PHC-GNN [29]	111K	79.34±1.16
DeeperGCN-FLAG [30]	532K	79.42±1.20
DGN [2]	114K	79.70±0.97
GIN-VN[54] (fine-tune)	3.3M	77.80±1.82
Graphormer-FLAG	47.0M	**80.51**±0.53

Table 4: Results on ZINC.

method	#param.	test MAE
GIN [54]	509,549	0.526±0.051
GraphSage [18]	505,341	0.398±0.002
GAT [50]	531,345	0.384±0.007
GCN [26]	505,079	0.367±0.011
GatedGCN-PE [4]	505,011	0.214±0.006
MPNN (sum) [15]	480,805	0.145±0.007
PNA [10]	387,155	0.142±0.010
GT [13]	588,929	0.226±0.014
SAN [28]	508, 577	0.139±0.006
$\text{Graphormer}_{\text{SLIM}}$	489,321	**0.122**±0.006

the other pre-trained GNNs do not achieve competitive performance, which is in line with previous literature [20]. In addition, we conduct more comparisons to fine-tuning the pre-trained GNNs, please refer to Appendix C.

4.3 Ablation Studies

We perform a series of ablation studies on the importance of designs in our proposed Graphormer, on PCQM4M-LSC dataset. The ablation results are included in Table 5. To save the computation resources, the Transformer models in table 5 have 12 layers, and are trained for 100K iterations.

Node Relation Encoding. We compare previously used positional encoding (PE) to our proposed spatial encoding, which both aim to encode the information of distinct node relation to Transformers. There are various PEs employed by previous Transformer-based GNNs, e.g., Weisfeiler-Lehman-PE (WL-PE) [61] and Laplacian PE [3, 14]. We report the performance for Laplacian PE since it performs well comparing to a series of PEs for Graph Transformer in previous literature [13]. Transformer architecture with the spatial encoding outperforms the counterpart built on the positional encoding, which demonstrates the effectiveness of using spatial encoding to capture the node spatial information.

Centrality Encoding. Transformer architecture with degree-based centrality encoding yields a large margin performance boost in comparison to those without centrality information. This indicates that the centrality encoding is indispensable to Transformer architecture for modeling graph data.

Edge Encoding. We compare our proposed edge encoding (denoted as via attn bias) to two commonly used edge encodings described in Section 3.1.3 to incorporate edge features into GNN, denoted as via node and via Aggr in Table 5. From the table, the gap of performance is minor between the two conventional methods, but our proposed edge encoding performs significantly better, which indicates that edge encoding as attention bias is more effective for Transformer to capture spatial information on edges.

Table 5: Ablation study results on PCQM4M-LSC dataset with different designs.

Node Relation Encoding		Centrality	Edge Encoding			valid MAE
Laplacian PE[13]	Spatial		via node	via Aggr	via attn bias(Eq.7)	
-	-	-	-	-	-	0.2276
✓	-	-	-	-	-	0.1483
-	✓	-	-	-	-	0.1427
-	✓	✓	-	-	-	0.1396
-	✓	✓	✓	-	-	0.1328
-	✓	✓	-	✓	-	0.1327
-	✓	✓	-	-	✓	0.1304

5 Related Work

In this section, we highlight the most recent works which attempt to develop standard Transformer architecture-based GNN or graph structural encoding, but spend less effort on elaborating the works by adapting attention mechanism to GNNs [33, 60, 7, 23, 1, 50, 51, 61, 48].

5.1 Graph Transformer

There are several works that study the performance of pure Transformer architectures (stacked by transformer layers) with modifications on graph representation tasks, which are more related to our Graphormer. For example, several parts of the transformer layer are modified in [46], including an additional GNN employed in attention sub-layer to produce vectors of Q, K, and V, long-range residual connection, and two branches of FFN to produce node and edge representations separately. They pre-train their model on 10 million unlabelled molecules and achieve excellent results by fine-tuning on downstream tasks. Attention module is modified to a soft adjacency matrix in [41] by directly adding the adjacency matrix and RDKit[6]-computed inter-atomic distance matrix to the attention probabilites. Very recently, Dwivedi *et al.* [13] revisit a series of works for Transformer-based GNNs, and suggest that the attention mechanism in Transformers on graph data should only aggregate the information from neighborhood (i.e., using adjacent matrix as attention mask) to ensure graph sparsity, and propose to use Laplacian eigenvector as positional encoding. Their model GT surpasses baseline GNNs on graph representation task. A concurrent work [28] propose a novel full Laplacian spectrum to learn the position of each node in a graph, and empirically shows better results than GT.

5.2 Structural Encodings in GNNs

Path and Distance in GNNs. Information of path and distance is commonly used in GNNs. For example, an attention-based aggregation is proposed in [9] where the node features, edge features, one-hot feature of the distance and ring flag feature are concatenated to calculate the attention probabilites; similar to [9], path-based attention is leveraged in [56] to model the influence between the center node and its higher-order neighbors; a distance-weighted aggregation scheme on graph is proposed in [59]; it has been proved in [32] that adopting distance encoding (i.e., one-hot feature of the distance as extra node attribute) could lead to a strictly more expressive power than the 1-WL test.

Positional Encoding in Transformer on Graph. Several works introduce positional encoding (PE) to Transformer-based GNNs to help the model capture the node position information. For example, Graph-BERT [61] introduces three types of PE to embed the node position information to model, i.e., an absolute WL-PE which represents different nodes labeled by Weisfeiler-Lehman algorithm, an intimacy based PE and a hop based PE which are both variant to the sampled subgraphs. Absolute Laplacian PE is employed in [13] and empircal study shows that its performance surpasses the absolute WL-PE used in [61].

Edge Feature. Except the conventionally used methods to encode edge feature, which are described in previous section, there are several attempts that exploit how to better encode edge features: an attention-based GNN layer is developed in [16] to encode edge features, where the edge feature

[6] https://www.rdkit.org/

is weighted by the similarity of the features of its two nodes; edge feature has been encoded into the popular GIN [54] in [5]; in [13], the authors propose to project edge features to an embedding vector, then multiply it by attention coefficients, and send the result to an additional FFN sub-layer to produce edge representations;

6 Conclusion

We have explored the direct application of Transformers to graph representation. With three novel graph structural encodings, the proposed Graphormer works surprisingly well on a wide range of popular benchmark datasets. While these initial results are encouraging, many challenges remain. For example, the quadratic complexity of the self-attention module restricts Graphormer's application on large graphs. Therefore, future development of efficient Graphormer is necessary. Performance improvement could be expected by leveraging domain knowledge-powered encodings on particular graph datasets. Finally, an applicable graph sampling strategy is desired for node representation extraction with Graphormer. We leave them for future works.

7 Acknowledgement

We would like to thank Mingqi Yang and Shanda Li for insightful discussions.

References

[1] Jinheon Baek, Minki Kang, and Sung Ju Hwang. Accurate learning of graph representations with graph multiset pooling. *ICLR*, 2021.

[2] Dominique Beaini, Saro Passaro, Vincent Létourneau, William L Hamilton, Gabriele Corso, and Pietro Liò. Directional graph networks. In *International Conference on Machine Learning*, 2021.

[3] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[4] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[5] Rémy Brossard, Oriel Frigo, and David Dehaene. Graph convolutions that can finally model local structure. *arXiv preprint arXiv:2011.15069*, 2020.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[7] Deng Cai and Wai Lam. Graph transformer for graph-to-sequence learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7464–7471, 2020.

[8] Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-yan Liu, and Liwei Wang. Graphnorm: A principled approach to accelerating graph neural network training. In *International Conference on Machine Learning*, 2021.

[9] Benson Chen, Regina Barzilay, and Tommi Jaakkola. Path-augmented graph transformer network. *arXiv preprint arXiv:1905.12712*, 2019.

[10] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. Principal neighbourhood aggregation for graph nets. *Advances in Neural Information Processing Systems*, 33, 2020.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[13] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *AAAI Workshop on Deep Learning on Graphs: Methods and Applications*, 2021.

[14] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

[15] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272. PMLR, 2017.

[16] Liyu Gong and Qiang Cheng. Exploiting edge features for graph neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9211–9219, 2019.

[17] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.

[18] William L Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[19] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

[20] W Hu, B Liu, J Gomes, M Zitnik, P Liang, V Pande, and J Leskovec. Strategies for pre-training graph neural networks. In *International Conference on Learning Representations (ICLR)*, 2020.

[21] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430*, 2021.

[22] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[23] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, pages 2704–2710, 2020.

[24] Katsuhiko Ishiguro, Shin-ichi Maeda, and Masanori Koyama. Graph warp module: an auxiliary module for boosting the power of graph neural networks in molecular graph analysis. *arXiv preprint arXiv:1902.01020*, 2019.

[25] Guolin Ke, Di He, and Tie-Yan Liu. Rethinking the positional encoding in language pre-training. *ICLR*, 2020.

[26] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[27] Kezhi Kong, Guohao Li, Mucong Ding, Zuxuan Wu, Chen Zhu, Bernard Ghanem, Gavin Taylor, and Tom Goldstein. Flag: Adversarial data augmentation for graph neural networks. *arXiv preprint arXiv:2010.09891*, 2020.

[28] Devin Kreuzer, Dominique Beaini, William Hamilton, Vincent Létourneau, and Prudencio Tossou. Rethinking graph transformers with spectral attention. *arXiv preprint arXiv:2106.03893*, 2021.

[29] Tuan Le, Marco Bertolini, Frank Noé, and Djork-Arné Clevert. Parameterized hypercomplex graph neural networks for graph classification. *arXiv preprint arXiv:2103.16584*, 2021.

[30] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. Deepergcn: All you need to train deeper gcns. *arXiv preprint arXiv:2006.07739*, 2020.

[31] Junying Li, Deng Cai, and Xiaofei He. Learning graph-level representation for drug discovery. *arXiv preprint arXiv:1709.03741*, 2017.

[32] Pan Li, Yanbang Wang, Hongwei Wang, and Jure Leskovec. Distance encoding: Design provably more powerful neural networks for graph representation learning. *Advances in Neural Information Processing Systems*, 33, 2020.

[33] Yuan Li, Xiaodan Liang, Zhiting Hu, Yinbo Chen, and Eric P. Xing. Graph transformer, 2019.

[34] Xi Victoria Lin, Richard Socher, and Caiming Xiong. Multi-hop knowledge graph reasoning with reward shaping. *arXiv preprint arXiv:1808.10568*, 2018.

[35] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[36] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *arXiv preprint arXiv:2103.14030*, 2021.

[37] Shengjie Luo, Shanda Li, Tianle Cai, Di He, Dinglan Peng, Shuxin Zheng, Guolin Ke, Liwei Wang, and Tie-Yan Liu. Stable, fast and accurate: Kernelized attention with relative positional encoding. *NeurIPS*, 2021.

[38] Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[39] P David Marshall. The promotion and presentation of the self: celebrity as marker of presentational media. *Celebrity studies*, 1(1):35–48, 2010.

[40] Alice Marwick and Danah Boyd. To see and be seen: Celebrity practice on twitter. *Convergence*, 17(2):139–158, 2011.

[41] Łukasz Maziarka, Tomasz Danel, Sławomir Mucha, Krzysztof Rataj, Jacek Tabor, and Stanisław Jastrzębski. Molecule attention transformer. *arXiv preprint arXiv:2002.08264*, 2020.

[42] Maho Nakata and Tomomi Shimazaki. Pubchemqc project: a large-scale first-principles electronic structure database for data-driven chemistry. *Journal of chemical information and modeling*, 57(6):1300–1308, 2017.

[43] Sharan Narang, Hyung Won Chung, Yi Tay, William Fedus, Thibault Fevry, Michael Matena, Karishma Malkan, Noah Fiedel, Noam Shazeer, Zhenzhong Lan, et al. Do transformer modifications transfer across implementations and applications? *arXiv preprint arXiv:2102.11972*, 2021.

[44] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? In *International Conference on Machine Learning*. PMLR, 2021.

[45] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[46] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying Wei, Wenbing Huang, and Junzhou Huang. Self-supervised graph transformer on large-scale molecular data. *Advances in Neural Information Processing Systems*, 33, 2020.

[47] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, 2018.

[48] Yunsheng Shi, Zhengjie Huang, Wenjin Wang, Hui Zhong, Shikun Feng, and Yu Sun. Masked label prediction: Unified message passing model for semi-supervised classification. *arXiv preprint arXiv:2009.03509*, 2020.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[50] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *ICLR*, 2018.

[51] Guangtao Wang, Rex Ying, Jing Huang, and Jure Leskovec. Direct multi-hop attention based graph neural network. *arXiv preprint arXiv:2009.14332*, 2020.

[52] Sinong Wang, Belinda Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[53] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pages 10524–10533. PMLR, 2020.

[54] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[55] Mingqi Yang, Yanming Shen, Heng Qi, and Baocai Yin. Breaking the expressive bottlenecks of graph neural networks. *arXiv preprint arXiv:2012.07219*, 2020.

[56] Yiding Yang, Xinchao Wang, Mingli Song, Junsong Yuan, and Dacheng Tao. Spagan: Shortest path graph attention network. *Advances in IJCAI*, 2019.

[57] Chengxuan Ying, Guolin Ke, Di He, and Tie-Yan Liu. Lazyformer: Self attention with lazy update. *arXiv preprint arXiv:2102.12702*, 2021.

[58] Chengxuan Ying, Mingqi Yang, Shuxin Zheng, Guolin Ke, Shengjie Luo, Tianle Cai, Chenglin Wu, Yuxin Wang, Yanming Shen, and Di He. First place solution of kdd cup 2021 & ogb large-scale challenge graph-level track. *arXiv preprint arXiv:2106.08279*, 2021.

[59] Jiaxuan You, Rex Ying, and Jure Leskovec. Position-aware graph neural networks. In *International Conference on Machine Learning*, pages 7134–7143. PMLR, 2019.

[60] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. Graph transformer networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[61] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140*, 2020.

[62] Chen Zhu, Yu Cheng, Zhe Gan, Siqi Sun, Tom Goldstein, and Jingjing Liu. Freelb: Enhanced adversarial training for natural language understanding. In *ICLR*, 2020.

[63] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations*, 2020.

A Proofs

A.1 SPD can Be Used to Improve WL-Test

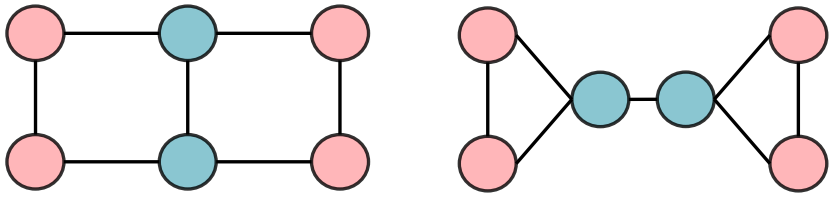


Figure 2: These two graphs cannot be distinguished by 1-WL-test. But the SPD sets, i.e., the SPD from each node to others, are different: The two types of nodes in the left graph have SPD sets $\{0,1,1,2,2,3\}, \{0,1,1,1,2,2\}$ while the nodes in the right graph have SPD sets $\{0,1,1,2,3,3\}, \{0,1,1,1,2,2\}$.

1-WL-test fails in many cases [38, 32], thus classic message passing GNNs also fail to distinguish many pairs of graphs. We show that SPD might help when 1-WL-test fails, for example, in Figure 2 where 1-WL-test fails, the sets of SPD from all nodes to others successfully distinguish the two graphs.

A.2 Proof of Fact 1

MEAN AGGREGATE. We begin by showing that self-attention module with Spatial Encoding can represent MEAN aggregation. This is achieved by in Eq. (6): 1) setting $b_\phi = 0$ if $\phi = 1$ and $b_\phi = -\infty$ otherwise where ϕ is the SPD; 2) setting $W_Q = W_K = 0$ and W_V to be the identity matrix. Then $\mathrm{softmax}(A)V$ gives the average of representations of the neighbors.

SUM AGGREGATE. The SUM aggregation can be realized by first perform MEAN aggregation and then multiply the node degrees. Specifically, the node degrees can be extracted from Centrality Encoding by an additional head and be concatenated to the representations after MEAN aggregation. Then the FFN module in Graphormer can represent the function of multiplying the degree to the dimensions of averaged representations by the universal approximation theorem of FFN.

MAX AGGREGATE. Representing the MAX aggregation is harder than MEAN and SUM. For each dimension t of the representation vector, we need one head to select the maximal value over t-th dimension in the neighbor by in Eq. (6): 1) setting $b_\phi = 0$ if $\phi = 1$ and $b_\phi = -\infty$ otherwise where ϕ is the SPD; 2) setting $W_K = e_t$ which is the t-th standard basis; $W_Q = 0$ and the bias term (which is ignored in the previous description for simplicity) of Q to be $T\mathbf{1}$; and $W_V = e_t$, where T is the temperature that can be chosen to be large enough so that the softmax function can approximate hard max and $\mathbf{1}$ is the vector whose elements are all 1.

COMBINE. The COMBINE step takes the result of AGGREGATE and the previous representation of current node as input. This can be achieved by the AGGREGATE operations described above together with an additional head which outputs the features of present nodes, i.e., in Eq. (6): 1) setting $b_\phi = 0$ if $\phi = 0$ and $b_\phi = -\infty$ otherwise where ϕ is the SPD; 2) setting $W_Q = W_K = 0$ and W_V to be the identity matrix. Then the FFN module can approximate any COMBINE function by the universal approximation theorem of FFN.

A.3 Proof of Fact 2

MEAN READOUT. This can be proved by setting $W_Q = W_K = 0$, the bias terms of Q, K to be $T\mathbf{1}$, and W_V to be the identity matrix where T should be much larger than the scale of b_ϕ so that $T^2\mathbf{1}\mathbf{1}^\top$ dominates the Spatial Encoding term.

B Experiment Details

B.1 Details of Datasets

We summarize the datasets used in this work in Table 6. PCQM4m-LSC is a quantum chemistry graph-level prediction task in recent OGB Large-Scale Challenge, originally curated under the PubChemQC project [42].

Table 6: Statistics of the datasets.

Dataset	Scale	# Graphs	# Nodes	# Edges	Task Type
PCQM4M-LSC	Large	3,803,453	53,814,542	55,399,880	Regression
OGBG-MolPCBA	Medium	437,929	11,386,154	12,305,805	Binary classification
OGBG-MolHIV	Small	41,127	1,048,738	1,130,993	Binary classification
ZINC (sub-set)	Small	12,000	277,920	597,960	Regression

The task of PCQM4M-LSC is to predict DFT(density functional theory)-calculated HOMO-LUMO energy gap of molecules given their 2D molecular graphs, which is one of the most practically-relevant quantum chemical properties of molecule science. PCQM4M-LSC is unprecedentedly large in scale comparing to other labeled graph-level prediction datasets, which contains more than 3.8M graphs. Besides, we conduct experiments on two molecular graph datasets in popular OGB leaderboards, i.e., OGBG-MolPCBA and OGBG-MolHIV. They are two molecular property prediction datasets with different sizes. The pre-trained knowledge of molecular graph on PCQM4M-LSC could be easily leveraged on these two datasets. We adopt official scaffold split on three datasets following [21, 22]. In addition, we employ another popular leaderboard, i.e., benchmarking-gnn [14]. We use the ZINC datasets, which is the most popular real-world molecular dataset to predict graph property regression for contrained solubility, an important chemical property for designing generative GNNs for molecules. Different from the scaffold spliting in OGB, uniform sampling is adopted in ZINC for data splitting.

B.2 Details of Training Strategies

B.2.1 PCQM4M-LSC

Table 7: Model Configurations and Hyper-parameters of Graphormer on PCQM4M-LSC.

	$\text{Graphormer}_{\text{SMALL}}$	Graphormer
#Layers	6	12
Hidden Dimension d	512	768
FFN Inner-layer Dimension	512	768
#Attention Heads	32	32
Hidden Dimension of Each Head	16	24
FFN Dropout	0.1	0.1
Attention Dropout	0.1	0.1
Embedding Dropout	0.0	0.0
Max Steps	$1M$	$1M$
Max Epochs	300	300
Peak Learning Rate	3e-4	2e-4
Batch Size	1024	1024
Warm-up Steps	$60K$	$60K$
Learning Rate Decay	Linear	Linear
Adam ϵ	1e-8	1e-8
Adam (β_1, β_2)	(0.9, 0.999)	(0.9, 0.999)
Gradient Clip Norm	5.0	5.0
Weight Decay	0.0	0.0

We report the detailed hyper-parameter settings used for training Graphormer in Table 7. We reduce the FFN inner-layer dimension of $4d$ in [49] to d, which does not appreciably hurt the performance but significantly save the parameters. The embedding dropout ratio is set to 0.1 by default in many previous Transformer works [11, 35]. However, we empirically find that a small embedding dropout ratio (e.g., 0.1) would lead to an observable performance drop on validation set of PCQM4M-LSC. One possible reason is that the molecular graph is relative small (i.e., the median of #atoms in each molecule is about 15), making graph property more sensitive to the embeddings of each node. Therefore, we set embedding dropout ratio to 0 on this dataset.

B.2.2 OGBG-MolPCBA

Pre-training. We first report the model configurations and hyper-parameters of the pre-trained Graphormer on PCQM4M-LSC. Empirically, we find that the performance on MolPCBA benefits from the large pre-training model size. Therefore, we train a deep Graphormer with 18 Transformer layers on PCQM4M-LSC. The hidden dimension and FFN inner-layer dimension are set to 1024. We set peak learning rate to 1e-4 for the deep

Table 8: Hyper-parameters for Graphormer on OGBG-MolPCBA, where the **text in bold** denotes the hyper-parameters we eventually use.

	Graphormer
Max Epochs	{2, 5, **10**}
Peak Learning Rate	{2e-4, **3e-4**}
Batch Size	256
Warm-up Ratio	0.06
Attention Dropout	0.3
m	{1, 2,3,**4**}
α	0.001
ϵ	0.001

Graphormer. Besides, we enlarge the attention dropout ratio from 0.1 to 0.3 in both pre-training and fine-tuning to prevent the model from over-fitting. The rest of hyper-parameters remain unchanged. The pre-trained Graphormer used for MolPCBA achieves a valid MAE of 0.1253 on PCQM4M-LSC, which is slightly worse than the reports in Table 1.

Fine-tuning. Table 8 summarizes the hyper-parameters used for fine-tuning Graphormer on OGBG-MolPCBA. We conduct a grid search for several hyper-parameters to find the optimal configuration. The experimental results are reported by the mean of 10 independent runs with random seeds. We use FLAG [27] with minor modifications for graph data augmentation. In particular, except the step size α and the number of steps m, we also employ a projection step in [62] with maximum perturbation ϵ. The performance of Graphormer on MolPCBA is quite robust to the hyper-parameters of FLAG. The rest of hyper-parameters are the same with the pre-training model.

B.2.3 OGBG-MolHIV

Table 9: Hyper-parameters for Graphormer on OGBG-MolHIV, where the **text in bold** denotes the hyper-parameters we eventually use.

	Graphormer
Max Epochs	8
Peak Learning Rate	2e-4
Batch Size	128
Warm-up Ratio	0.06
Dropout	0.1
Attention Dropout	0.1
m	{1,**2**,3,4}
α	{0.001, 0.01, 0.1, **0.2**}
ϵ	{**0**, 0.001, 0.01, 0.1}

Pre-training. We use the Graphormer reported in Table 1 as the pre-trained model for OGBG-MolHIV, where the pre-training hyper-parameters are summarized in Table 7.

Fine-tuning. The hyper-parameters for fine-tuning Graphormer on OGBG-MolHIV are presented in Table 9. Empirically, we find that the different choices of hyper-parameters of FLAG (i.e., step size α, number of steps m, and maximum perturbation ϵ) would greatly affect the performance of Graphormer on OGBG-MolHiv. Therefore, we spend more effort to conduct grid search for hyper-parameters of FLAG. We report the best hyper-parameters by the mean of 10 independent runs with random seeds.

B.2.4 ZINC

To keep the total parameters of Graphormer less than 500K per the request from benchmarking-GNN leaderboard [14], we train a slim 12-layer Graphormer with hidden dimension of 80, which is called Graphormer$_{\text{SLIM}}$ in Table 4, and has about 489K learnable parameters. The number of attention heads is set to 8. Table 10 summarizes the detailed hyper-parameters on ZINC. We train 400K steps on this dataset, and employ a weight decay of 0.01.

Table 10: Model Configurations and Hyper-parameters on ZINC(sub-set).

	$\text{Graphormer}_{\text{SLIM}}$
#Layers	12
Hidden Dimension	80
FFN Inner-Layer Hidden Dimension	80
#Attention Heads	8
Hidden Dimension of Each Head	10
FFN Dropout	0.1
Attention Dropout	0.1
Embedding Dropout	0.0
Max Steps	$400K$
Max Epochs	$10K$
Peak Learning Rate	2e-4
Batch Size	256
Warm-up Steps	$40K$
Learning Rate Decay	Linear
Adam ϵ	1e-8
Adam (β_1, β_2)	(0.9, 0.999)
Gradient Clip Norm	5.0
Weight Decay	0.01

Table 11: Hyper-parameters for fine-tuning GROVER on MolHIV and MolPCBA.

	GROVER	$\text{GROVER}_{\text{LARGE}}$
Dropout	{0.1, 0.5}	{0.1, 0.5}
Max Epochs	{10, 30, 50}	{10, 30}
Learning Rate	{5e-5, 1e-4, 5e-4, 1e-3}	{5e-5, 1e-4, 5e-4, 1e-3}
Batch Size	{64, 128}	{64, 128}
Initial Learning Rate	1e-7	1e-7
End Learning Rate	1e-9	1e-9

B.3 Details of Hyper-parameters for Baseline Methods

In this section, we present the details of our re-implementation of the baseline methods.

B.3.1 PCQM4M-LSC

The official Github repository of OGB-LSC[7] provides hyper-parameters and codes to reproduce the results on leaderboard. These hyper-parameters work well on almost all popular GNN variants, except the DeeperGCN-VN, which results in a training divergence. Therefore, for DeeperGCN-VN, we follow the official hyper-parameter setting[8] provided by the authors [30]. For a fair comparison to Graphormer, we train a 12-layer DeeperGCN. The hidden dimension is set to 600. The batch size is set to 256. The learning rate is set to 1e-3, and a step learning rate scheduler is employed with the decaying step size and the decaying factor γ as 30 epochs and 0.25. The model is trained for 100 epochs.

The default dimension of laplacian PE of GT [13] is set to 8. However, it will cause 2.91% small molecules (less than 8 atoms) to be filtered out. Therefore, for GT and GT-Wide, we set the dimension of laplacian PE to 4, which results in only 0.08% filtering out. We adopt the default hyper-parameter settings described in [13], except that we decrease the learning rate to 1e-4, which leads to a better convergence on PCQM4M-LSC.

B.3.2 OGBG-MolPCBA

To fine-tune the pre-trained GIN-VN on MolPCBA, we follow the hyper-parameter settings provided in the original OGB paper [22]. To be more concrete, we load the pre-trained checkpoint reported in Table 1 and fine-tune it on OGBG-MolPCBA dataset. We use the grid search on the hyper-parameters for better fine-tuning

[7] https://github.com/snap-stanford/ogb/tree/master/examples/lsc/pcqm4m

[8] https://github.com/lightaime/deep_gcns_torch/tree/master/examples/ogb/ogbg_mol#train

Table 12: Comparison to pre-trained Transformer-based GNN on MolHIV. * indicates that additional features for molecule are used.

method	#param.	AUC (%)
Morgan Finger Prints + Random Forest*	230K	**80.60**±0.10
GROVER* [46]	48.8M	79.33±0.09
$\text{GROVER}_{\text{LARGE}}$* [46]	107.7M	80.32±0.14
Graphormer-FLAG	47.0M	80.51±0.53

Table 13: Comparison to pre-trained Transformer-based GNN on MolPCBA. * indicates that additional features for molecule are used.

method	#param.	AP (%)
GROVER* [46]	48.8M	16.77±0.36
$\text{GROVER}_{\text{LARGE}}$* [46]	107.7M	13.05±0.18
Graphormer-FLAG	47.0M	**31.39**±0.32

performance. In particular, the learning rate is selected from $\{1e-5, 1e-4, 1e-3\}$; the dropout ratio is selected from $\{0.0, 0.1, 0.5\}$; the batch size is selected from $\{32, 64\}$.

B.3.3 OGBG-MolHIV

Similarly, we fine-tune the pre-trained GIN-VN on MolHIV by following the hyper-parameter settings provided in the original OGB paper [22]. We also conduct the grid search to look for optimal hyper-parameters. The ranges for each hyper-parameter of grid search are the same as the previous subsection.

C More Experiments

As described in the related work, GROVER is a Transformer-based GNN, which has 100 million parameters and pre-trained on 10 million unlabelled molecules using 250 Nvidia V100 GPUs. In this section, we report the fine-tuning scores of GROVER on MolHIV and MolPCBA, and compare with proposed Graphormer.

We download the pre-trained GROVER models from its official Github webpage[9], follow the official instructions[10] and fine-tune the provided pre-trained checkpoints with careful search of hyper-parameters (in Table 11). We find that GROVER could achieve competitive performance on MolHIV only if employing additional molecular features, i.e., morgan molecular finger prints and 2D features[11]. Therefore, we report the scores of GROVER by taking these two additional molecular features. Please note that, from the leaderboard[12], we can know such additional molecular features are very effective on MolHIV dataset.

Table 12 and 13 summarize the performance of GROVER and $\text{GROVER}_{\text{LARGE}}$ comparing with Graphormer on MolHIV and MolPCBA. From the tables, we observe that Graphormer could consistently outperform GROVER even without any additional molecular features.

D Discussion & Future Work

Complexity. Similar to regular Transformer, the attention mechanism in Graphormer scales quadratically with the number of nodes n in the input graph, which may be prohibitively expensive for large n and precludes its usage in settings with limited computational resources. Recently, many solutions have been proposed to address this problem in Transformer [25, 52, 57, 37]. This issue would be greatly benefit from the future development of efficient Graphormer.

Choice of centrality and ϕ. In Graphormer, there are multiple choices for the network centrality and the spatial encoding function $\phi(v_i, v_j)$. For example, one can leverage the L_2 distance in 3D structure between two atoms in a molecule. In this paper, we mainly evaluate general centrality and distance metric in graph theory, i.e., the degree centrality and the shortest path. Performance improvement could be expected by leveraging domain knowledge powered encodings on particular graph dataset.

[9] https://github.com/tencent-ailab/grover

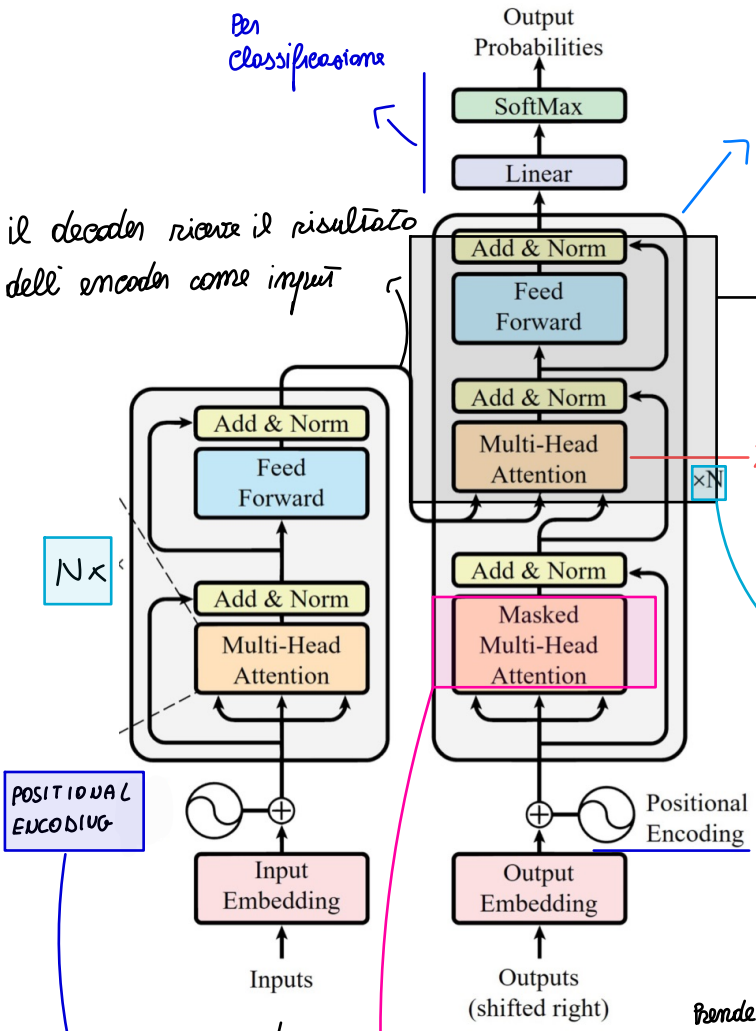
[10] https://github.com/tencent-ailab/grover/blob/main/README.md#finetuning-with-existing-data

[11] https://github.com/tencent-ailab/grover#optional-molecular-feature-extraction-1

[12] https://ogb.stanford.edu/docs/leader_graphprop/

Node Representation. There is a wide range of node representation tasks on graph structured data, such as finance, social network, and temporal prediction. Graphormer could be naturally used for node representation extraction with an applicable graph sampling strategy. We leave it for future work.

Attention layer
e transformers



Per classificazione

DECODER BLOCK

il decoder riceve il risultato dell'encoder come input

> questo è esattamente come l'encoder

Questo è lo stesso di prima ma due degli input è il risultato dell'encoder e l'altro arriva dal decoder

↗ V, K arriva dall'encoder

↘ Q dal decoder

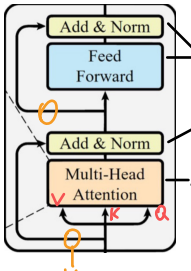
Questo Nx che si trova sia nell'encoder che nel decoder significa che questi blocchi sono ripetuti "N" volte: invece di avere un encoder e mandare il risultato al decoder, abbiamo tanti encoder (N appunto) e poi il decoder.

POSITIONAL ENCODING

Senza questo cambiando l'ordine delle parole avrei lo stesso risultato (positional invariant) ma per sequenze di parole questo non è il caso!

Prende un input (per esempio un testo di Shakespeare), creiamo degli embedding per l'input e passiamo questo embedding al rettangolino in riquadro (l'encoder) che viene detto transformer block:

ENCODER !



> Feed Forward

> Normalization

Parte più importante! Passiamo l'input a 3 Heads (Values, Keys, Queries): passo lo stesso input a tutti e 3.

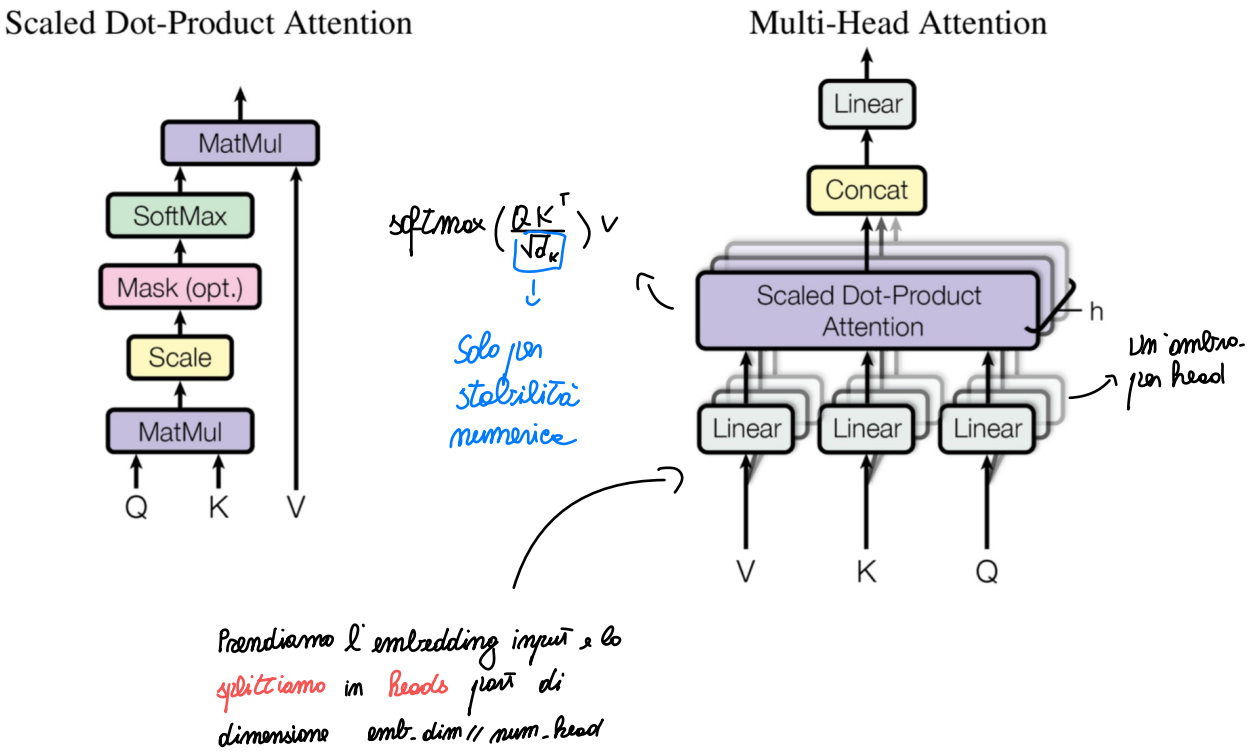
Skip Connection (ResNet)

Uno dei più grandi vantaggi dei transformers è la sua capacità di parallelizzare il calcolo andando a ricavare il risultato dell'encoder in diversi punti del testo tutti assieme creando data leakage (perché la 1° parola potrebbe essere una parola da tradurre e la 2° la prima parola tradotta e semplicemente non capisce un nulla della traduzione).

IN PRATICA IN QUESTO MODO IL 1° OUTPUT DEL DECODER HA ACCESSO SOLO AL 1° ELEMENTO, IL 2° OUTPUT SOLO AL 1° E AL 2°.

Positional encoding

Necessitiamo di un modo (per sequenze di parole, non per grafi in questo caso) per capire la posizione delle parole e tenerne conto: un modo per dire che se due parole sono molto vicine probabilmente sono accomunate da un significato comune.



Implementazione

```
Import torch
Import torch.nn as nn

class SelfaAttention(nn.Mudule):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size=embed_size
        self.heads=heads
        self._head_dim = embed_size//heads

        assert(self.head_dim*heads=embed_size)

        self.values=nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.keys=nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.queries=nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.fc_out = nn.Linear(heads*self.head_dim, embed_size)


    def forward(self, values, keys, query, mask):
        N=query.shape[0]
        value_len, key_len, query_len = value.shape[1], key.shape[1], query.shape[1]

        #splittiamo l'embedding in self.head pezzi:
        values=values.reshape(N, values_len, self.heads, self.head_dim)
        keys=keys.reshape(N, key_len, self.heads, self.head_dim)
        quesries=questies.reshape(N, query_len, self.heads, self.head_dim)

        #adesso effettuiamo il calcolo tra Q e K, per farlo usiamo torch.einsum()
        energy = torch.einsum()

        energy = torch.einsum("nqhd, nkhd->nhqk", [queries, keys])

        #se abbiamo una mask la applico:
        If mask is not None:
            energy=energy.masked_fill(mask==0, float("-1e20"))

        attention = torch.softmax(energy/ (self.e,pmbed_size**(1/2)), dim=3)
        out = torch.einsum("nhqk, nlhd->nqhd", [attention, values])
```

Le ombre del precedente grafico

Nota che stiamo costruendo questa matrici senza sapere dove le useremo (encoder o in quale parte del decoder), e siccome la dimensione varia questa sarà variabile

Nello specifico questa dimensione sarà pari alla dimensione delle sequenze input (encoder) o output (decoder).

Nuova dimensione

Nota che abbiamo queste dimensioni:

queries shape: (N, query_len, heads, heads_dim)
Keys shape: (N, key_len, heads, heads_dim)
Vorrei:
energy shape: (N, heads, query_len, key_len)

qui stai dicendo che prendo le 2 matrici a dx e ogni size la nomino 'n' ecc per dire quale coincide e cosa voglio come risultato.

Pensa a questa operazione come se la query_len fosse la frase target e la key_len è la sorgente (il testo iniziale): per ogni parola nella target mi chiedo quanta importanza do ad ogni parola sorgente (del testo)

- ∞

(N, query_len, heads, head_dim)

(N, heads, query_len, key_len)

(N, value_len, heads, heads_dim)

out = self.fc_out(out)
return out

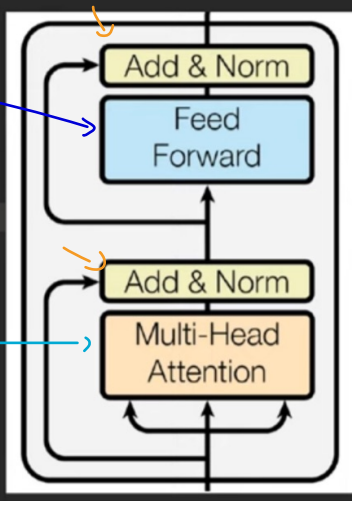
```
class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion*embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion*embed_size, embed_size)
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):
        attention = self.attention(value, key, query, mask)

        x = self.dropout(self.norm1(attention + query))
        forward = self.feed_forward(x)
        out = self.dropout(self.norm2(forward + x))
        return out
```

> SKIP CONNECTION



Questo è quello che fanno nel paper che lo presenta (Attention is all you need)

Adesso definiamo il modello di encoder e di decoder:

```
class Encoder(nn.Module):
    def __init__(
            self,
            src_vocab_size,
            embed_size,
            num_layers,
            heads,
            device,
            forward_expansion,
            dropout,
            max_length,
    ):
        super(Encoder, self).__init__()
        self.embed_size = embed_size
        self.device = device
        self.word_embedding = nn.Embedding(src_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [
                TransformerBlock(
                    embed_size,
                    heads,
                    dropout=dropout,
                    forward_expansion=forward_expansion,
                )
            ]
        )
```

```python
def forward(self, x, mask):
    N, seq_length = x.shape
    positions = torch.arange(0, seq_length).expand(N, seq_length).to(self.device)

    out = self.dropout(self.word_embedding(x) + self.position_embedding(positions))

    for layer in self.layers:
        out = layer(out, out, out, mask)

    return out
```

Cosa che rende il modello position aware.

```python
class Decoder(nn.Module):
    def __init__(
            self,
            trg_vocab_size,
            embed_size,
            num_layers,
            heads,
            forward_expansion,
            dropout,
            device,
            max_length,
    ):
        super(Decoder, self).__init__()
        self.device = device
        self.word_embedding = nn.Embedding(trg_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [DecoderBlock(embed_size, heads, forward_expansion, dropout, device)
             for _ in range(num_layers)]
        )

        self.fc_out = nn.Linear(embed_size, trg_vocab_size)
        self.dropout = nn.Dropout(dropout)
```

Adesso possiamo mettere tutto assieme dentro una classe Transformer