FlowerFormer: Empowering Neural Architecture Encoding using a Flow-aware Graph Transformer

Dongyeong Hwang Hyunju Kim Sunwoo Kim Kijung Shin
Kim Jaechul Graduate School of AI, KAIST, Seoul, Republic of Korea
{dy.hwang, hyunju.kim, kswoo97, kijungs}@kaist.ac.kr

Abstract

The success of a specific neural network architecture is closely tied to the dataset and task it tackles; there is no one-size-fits-all solution. Thus, considerable efforts have been made to quickly and accurately estimate the performances of neural architectures, without full training or evaluation, for given tasks and datasets. Neural architecture encoding has played a crucial role in the estimation, and graph-based methods, which treat an architecture as a graph, have shown prominent performance. For enhanced representation learning of neural architectures, we introduce FlowerFormer, *a powerful graph transformer that incorporates the information flows within a neural architecture.* FlowerFormer *consists of two key components: (a) bidirectional asynchronous message passing, inspired by the flows; (b) global attention built on flow-based masking. Our extensive experiments demonstrate the superiority of* FlowerFormer *over existing neural encoding methods, and its effectiveness extends beyond computer vision models to include graph neural networks and auto speech recognition models. Our code is available at* http://github.com/y0ngjaenius/CVPR2024_FLOWERFormer.

1. Introduction

While deep learning models have demonstrated their efficacy across various applications, the performance of a specific neural architecture heavily depends on specific downstream tasks and datasets employed. As a result, numerous neural architectures have been developed [11, 12].

In response to this dependency, significant efforts have been made to rapidly and accurately predict the performances of neural architectures for given tasks and datasets. This endeavor is crucial because exhaustively training and/or evaluating many candidate neural architectures is an expensive process. To this end, researchers have primarily employed machine learning techniques [6, 19].

Especially, various neural architecture encoding methods have been proposed since obtaining an accurate representation of each architecture plays a crucial role in the estimation process. Their focus has mainly revolved around (a) transforming input neural architectures to appropriate data structures [20, 41] and (b) applying representation-learning models to the transformed structures [5, 42].

Some have treated neural architectures as graphs and applied graph representation learning. They, however, share some limitations. For instance, their basic message-passing mechanisms oversimplify neural-architecture characteristics [35, 40] and may suffer from over-smoothing [29], over-squashing [2], or limited expressiveness [32].

Graph Transformers (GTs), when incorporated with adequate information, are recognized for enhancing basic message passing, making them effective in various graph classification [13, 46] and regression [7, 22] tasks. One strength of GTs lies in their global attention mechanisms [38], where all nodes in an input graph contribute directly to forming the representation for each individual node.

However, without integrating relevant topological or external information of input graphs, the relevance of attention scores, and thus the effectiveness of GTs, might be impaired. For example, Niu et al. [28] showed the essentiality of using motif-based spatial embedding to incorporate the characteristics of molecule graphs into GTs.

In this work, we propose FlowerFormer (**Flow**-awar**e** g**r**aph trans**former**), a GT model specialized in capturing *information flows* within neural architectures, as illustrated in Fig. 1. The information flows of a neural architecture contain the characteristics of both forward and backward propagations of the architecture, and thus describe its fundamental properties. FlowerFormer includes two core modules: the *flow encode* module and the *flow-aware global attention* module. The former conducts bidirectional asynchronous message passing, imitating the forward and backward propagations within the input neural architecture. The latter applies global attention with masking schemes based on the flow-based dependencies between nodes.

Our extensive experiments on neural architecture performance prediction, conducted using five benchmark datasets,

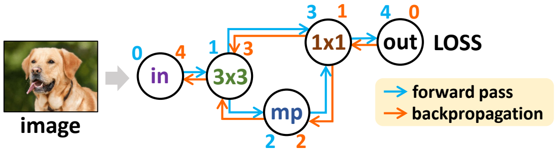

Figure 1. Information flows within an example neural architecture from the NAS-Bench-101 benchmark [45]. The architecture is represented as a directed graph where each node corresponds to an operation, and the topological structure of the graph encodes the sequence in which these operations are performed. For instance, the '1 × 1' (convolution) operation is executed only after the '3 × 3' (convolution) and 'mp' (max pooling) operations have been completed. The forward pass, depicted by blue arrows, is followed by the backpropagation of the loss, depicted by orange arrows. The number displayed above each node indicates the processing order within each flow.

validate the superiority of FLOWERFORMER over state-of-the-art neural encoding models [27, 44]. The results highlight the effectiveness of incorporating flows into GTs. Our contributions are summarized as follows:

- We propose FLOWERFORMER, a flow-aware GT-based neural architecture encoding model. To our best knowledge, FLOWERFORMER is the first GT model specifically designed to capture flows.
- FLOWERFORMER outperforms six baseline architectures, including the most recent ones [27, 44], by a substantial margin across three benchmark datasets in the computer vision domain. Specifically, in predicting the performance of neural architectures, it outperforms the top-performing baseline method by a margin of up to 4.38% in Kendall's Tau. Additionally, through ablation studies, we justify the design choices made in FLOWERFORMER.
- Beyond computer vision neural architectures, FLOWERFORMER also excels at performance prediction for graph neural networks and auto speech recognition architectures. In the benchmarks for these architectures, FLOWERFORMER achieves performance gains of up to 4.41% in Kendall's Tau over baseline models.

Our code is available at http://github.com/y0ngjaenius/CVPR2024_FLOWERFormer.

2. Related work

In this section, we briefly review related studies in neural architecture encoding and graph transformers (GTs).

2.1. Neural architecture encoding

Neural architecture encoding [17, 19, 20, 39, 41], which aims to learn representations of neural architectures, has gained considerable attention due to its significant downstream tasks, such as performance prediction (i.e., the prediction of task- and data-specific performance for given architectures without full training or evaluation).

One popular class of approaches is graph-based, modeling neural architectures as graphs and using graph neural networks [15] for representation learning. These approaches have also introduced topology-based graph similarity and operation-specific embeddings [4, 8].

Another significant approach aims to obtain representations that mimic the forward and/or backward passes within neural architectures. For instance, GATES [26] updates operation embeddings by mimicking the application of operations to information (which is also represented as a vector) and thus effectively replicating the forward-pass of convolution operations. Another method, TA-GATES [27], simulates an iterative process involving both forward and backward passes, with specialized handling for specific operations, e.g., skip-connections. However, these methods focus on flows only at a local level, by simulating a series of local operations, and may overlook a global-level perspective.

Transformer-based models [18, 43] are capable of capturing global-level perspectives through attention mechanisms. NAR-Former [44], a multi-stage fusion transformer, is one of the state-of-the-art methods for predicting neural architecture performance. They (1) represent a neural architecture as a sequence of operations to employ a transformer-based model and (2) leverage multiple valid sequences from the same architecture for augmentation.

In this work, we unify all three dimensions—graph learning, flow modeling, and global attention—by introducing a novel flow-aware GT, marking the first instance of such integration to the best of our knowledge.

2.2. Graph transformers (GTs)

Graph transformers (GTs) [10, 13, 16, 32, 36, 46] apply global (i.e., graph-level) attention between all node pairs. Recently, GTs show remarkable performance in various graph-level tasks, including molecular property prediction [14, 33], image classification [25, 49], and human interaction recognition [30].

To further improve their effectiveness, global attention is often supplemented with topological and/or external information. The information includes eigenvectors of adjacency and Laplacian matrices [16, 36] and pair-wise node similarity derived from shortest paths, diffusion kernels, random walks, etc [16, 24, 46].

Some GTs are tailored for specific types of graphs. For molecular graphs, where motifs play key roles, Niu et al. [28] employ motif-based spatial embeddings in a GT. DAGFormer [21] is designed for directed acyclic graphs (DAGs) and incorporates depth-based positional encodings and reachability-based attention. Note that DAGFormer is designed for general DAGs, and it is not optimized for encoding neural architectures, especially in capturing architecture-specific flows.

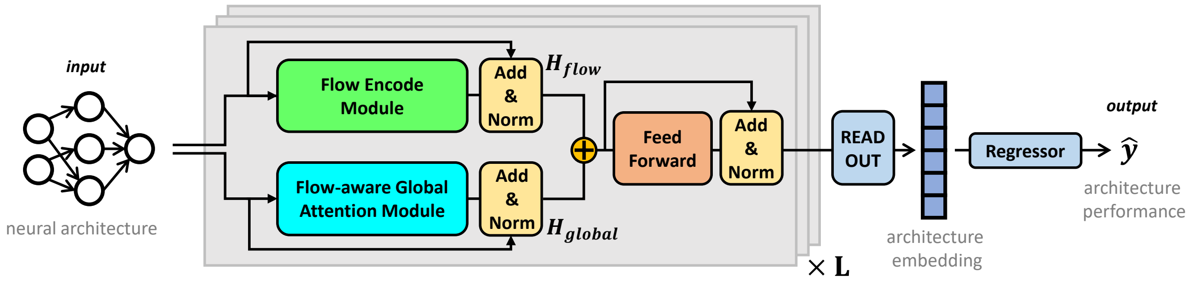

Figure 2. Overview of proposed FlowerFormer, which contains two key modules in each of its layers: the *flow encode* module and the *flow-aware global attention* module. The *flow encode* module performs bidirectional asynchronous message passing, inspired by forward and backward passes, to produce a node embedding matrix H_{flow}. The *flow-aware global attention* module computes attention with a flow-based masking scheme to yield another node embedding matrix H_{global}. These two embedding matrices, H_{flow} and H_{global}, are combined and then projected to produce updated node embeddings at each layer. This process is iterated over L layers, and the output node embeddings are aggregated to form the final architecture embedding, which is fed into a regressor for performance prediction.

3. Proposed method: FlowerFormer

In this section, we present **FlowerFormer** (**Flow**-awar**e** g**r**aph trans**former**), a graph transformer model designed to capture information flows within an input neural architecture. First, we provide the motivation behind FlowerFormer in Sec. 3.1. Then, we describe how an input neural architecture is represented as a graph in Sec. 3.2. After that, we elaborate on how FlowerFormer learns the representation of the neural architecture graph. Specifically, we describe two core modules of FlowerFormer, collectively referred to as Flower, in Sec. 3.3. Lastly, we present the overall framework (refer to Fig. 2) in Sec. 3.4.

3.1. Motivation of capturing information flows

Despite the remarkable success of Graph Transformers (GTs) in various graph-level tasks, including graph classification [13, 46] and regression [7, 22], their application for encoding neural architectures has received relatively limited attention. Existing applications of GTs suggest that additional design choices for accurately capturing the underlying characteristics of input graphs (on top of global attention mechanism between all pairs of nodes) are essential for the effectiveness of GTs. Refer to Sec. 2.2 for some examples.

In this work, we focus on a crucial aspect: capturing *information flows* within neural architectures (i.e., input graphs). Information flows include both the forward pass of data and the backpropagation of gradients. Hence, it is essential to capture information flows for incorporating how neural architectures are trained and conduct inference into their embeddings (i.e., the encoded neural architectures).

3.2. Input modeling

We represent a given neural architecture as a directed acyclic graph (DAG), with each node representing an operation (e.g., pooling or convolution). Each directional edge between two nodes indicates the information flow between the corresponding operations, aligning with the direction of data propagation during the forward pass. An illustrative example can be found on the left-hand side of Figure 3.

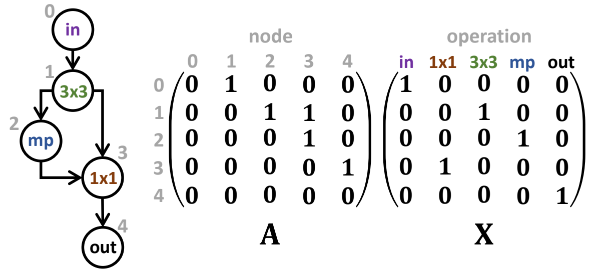

Figure 3. An example neural architecture from the NAS-Bench-101 dataset, represented as a directed acyclic graph (DAG), and its adjacency matrix A. Each column of the node feature matrix X corresponds to a specific operation, and each row in X is a one-hot vector indicating the type of operation associated with the corresponding node.

We denote the graph representation of a neural architecture by $G = (A, X)$, a tuple of an adjacency matrix $A \in \{0,1\}^{N\times N}$ and a node (i.e., operation) feature matrix $X \in \{0,1\}^{N\times D}$, where N is the number of nodes and D is the number of operations. The adjacency matrix encodes direct connections between node pairs in a graph. Its binary entries indicate whether a directional edge exists between each pair of nodes. Specifically, the (i, j)-th entry of A is set to 1 if there is a directed edge from the i-th node (denoted as v_i) to the j-th node (denoted as v_j), and 0 otherwise. Each node is associated with a one-hot feature vector representing its corresponding operation, and these vectors are stacked vertically to form the node feature matrix X. Refer to Fig. 3 for an example.

With our general input modeling scheme, FlowerFormer is readily applicable to different domains and neural architectures without such additional modelings or steps. By contrast, state-of-the-art neural encoding methods often rely on complex modelings and/or preprocessing steps, such as the specialized treatment of specific operations [27] and isomorphic augmentations [44] (refer to Sec. 2.1). The empirical superiority of FlowerFormer (refer to Sec. 4) despite its straightforward (yet elegant) input modeling is at-

```
Algorithm 1: Flow encode module
   Input: (1) $G = (A, X)$: an input neural architecture
          (2) $H$: an input node embedding matrix
   Output: $H$: updated node embedding matrix
1  /* step 1. topological sorting */
2  $\mathcal{T}^G \leftarrow$ topological generations of $G$
3  /* step 2. asynchronous forward message passing */
4  for $k = 1, \ldots, |\mathcal{T}^G|$ do
5      for $v_j \in T_k^G$ do
6          $h_j \leftarrow \mathrm{Comb}(h_j, \mathrm{Agg}\{m_e(h_j, h_i) : A_{ij} = 1\})$
7  /* step 3. asynchronous backward message passing */
8  for $k = |\mathcal{T}^G|, \ldots, 1$ do
9      for $v_j \in T_k^G$ do
10         $h_j \leftarrow \mathrm{Comb}(h_j, \mathrm{Agg}\{m_e(h_j, h_i) : A_{ji} = 1\})$
11 return $H$
```

tributed to our novel flow-aware GT architecture, which is described in the following subsection.

3.3. FLOWER layers

In this section, we introduce FLOWER layers, the basic units of FLOWERFORMER. A FLOWER layer consists of two core components: the *flow encode* module and the *flow-aware global attention* module. The flow encode module is a message-passing neural network (MPNN) that asynchronously passes messages in the forward and then the backward orders. The flow-aware global attention module is a self-attention module based on a flow-aware masking scheme. The outputs of the flow encode module and the flow-aware global attention module are node embedding matrices, denoted as $H_{flow}^{(l)} \in \mathbb{R}^{N \times d}$ and $H_{global}^{(l)} \in \mathbb{R}^{N \times d}$, respectively, for the l-th FLOWER layer. Below, we provide a detailed explanation of each module.

3.3.1 Flow encode module

As discussed in Sec. 3.1, we aim to enable a GT to capture the crucial aspect of neural architectures—*information flows*. To this end, the flow encode module conducts both asynchronous forward and backward message passing, resembling the forward pass (i.e., inference) and backpropagation (i.e., training) of neural architectures, respectively. These message-passing procedures are carried out in the (reversed) topological order in the input neural architecture graph, leading to updated node embeddings.

Pseudocode of the flow encode module is presented in Algorithm 1. It includes topological sorting, forward message passing, and backward message passing, in order, and each of these components is described below.

Topological sorting (Line 2): The first step is to divide nodes (i.e., operations) into topological generations. Recall that neural-architecture graphs are directed acyclic graphs (DAGs). Given a DAG G, its first topological generation, denoted as T_1^G, comprises the nodes without incoming edges in G. Then, for each $k > 1$, the k-th topological generation T_k^G comprises the nodes without incoming edges when all preceding generations are removed from G. The set of non-empty topological generations is denoted as $\mathcal{T}^G := \{T_1^G, \ldots, T_{|\mathcal{T}^G|}^G\}$. Refer to Fig. 4 for an example.

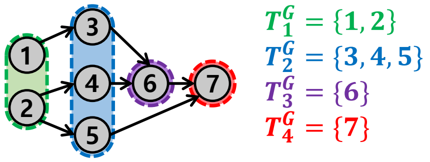

Figure 4. Example topological generations. Nodes 1 and 2 are devoid of incoming edges, and thus they constitute the first topological generation T_1^G. Upon removal of nodes 1 and 2, nodes 3, 4, and 5 no longer have incoming edges, and thus they compose the second generation T_2^G. Subsequently, nodes 6 and 7 form the third and fourth generations, respectively.

These topological generations are closely related to the data flow within a neural architecture. For the operations (i.e., nodes) in each generation to be executed, all operations in the preceding generations need to be complete. Conversely, during the process of backpropagation, gradients flow from subsequent generations to preceding generations.

Forward message passing (Line 4-Line 6): During the forward message passing step, node embeddings are updated asynchronously, following the order of the topological generations, akin to the forward pass within neural architectures. For each node v_j, its embedding h_j (i.e., the j-th row vector of H) is updated by the following three steps: (1) computing the message $m_e(h_j, h_i)$ for each incoming neighbor v_i, (2) aggregating these messages, and (3) combining the result with the current embedding h_j (Line 6). Note that the embeddings of all incoming neighbors, which belong to preceding generations, have already been updated by the time message calculation occurs. Also note that this differs from conventional synchronous graph message passing, where all node embeddings are updated simultaneously based on their input embeddings.

In our implementation, we use the sum aggregation as the Agg function. As m_e and Comb, we adopt the message function and the combine operator used in [37], as follows:

$$m_e(h_j, h_i) = \mathrm{softmax}(w_1^\top h_j + w_2^\top h_i) h_i, \quad (1)$$

$$\mathrm{msg}_j = \Sigma_{i:A_{ij}=1} m_e(h_j, h_i), \quad (2)$$

$$\mathrm{Comb}(h_j, \mathrm{msg}_j) = \mathrm{GRU}(h_j, \mathrm{msg}_j), \quad (3)$$

where $w_1 \in \mathbb{R}^d$ and $w_2 \in \mathbb{R}^d$ are learnable parameters.

Backward message passing (Line 8-Line 10): After the forward message passing step, we further update node embeddings through backward message passing, which resembles the process of backpropagation. This aligns with the standard practice in neural architecture training, where

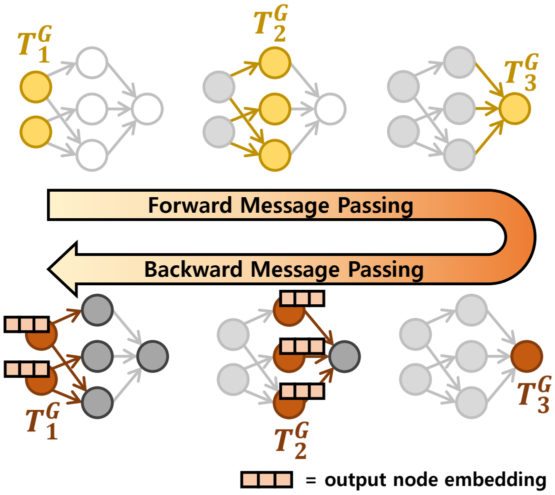

Figure 5. Flow encode module. During forward message passing, node embeddings are updated following the order of topological generations. Conversely, during backward message passing, node embeddings are updated in the reverse order of the generations.

backpropagation typically occurs after the forward pass for loss computation.

During the backward message passing step, node embeddings are updated asynchronously, following the reverse order of the topological generations. For each node v_j, the messages from its outgoing neighbors (rather than incoming neighbors) are computed and then aggregated (Line 10). The other details remain consistent with those of the forward message passing in to Eq. (1), Eq. (2), and Eq. (3).

Outputs: We denote the output node-embedding matrix of the flow encode module in the ℓ-th FLOWER layer, as $H_{flow}^{(\ell)} \in \mathbb{R}^{N \times d}$. That is,

$$H_{flow}^{(\ell)} = \mathrm{FlowEncoder}(G, H^{(\ell-1)}). \tag{4}$$

Here, $H^{(\ell-1)} \in \mathbb{R}^{N \times d}$ is the input node-embedding matrix obtained in layer-$(\ell-1)$, the previous layer (Eq. (6)).

3.3.2 Flow-aware global attention module

The flow-aware global attention module is designed to capture graph-level (i.e., architecture-level) characteristics, complementing the flow encode module which primarily focuses on local-level flows between directly connected operations. To this end, we employ a global attention mechanism of GTs; moreover, to accurately reflect the flows within architectures, we restrict attention scores to be computed only between nodes connected by at least one path of the flows. Specifically, we employ a masking strategy [21, 43] with a mask matrix $M \in \mathbb{R}^{N \times N}$ defined as follows (refer to Fig. 6 for an example of M):

$$M_{ij} = \begin{cases} 1 & \text{if } v_i \text{ lies on any directed path from } v_j \\ & \text{or } v_j \text{ lies on any directed path from } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

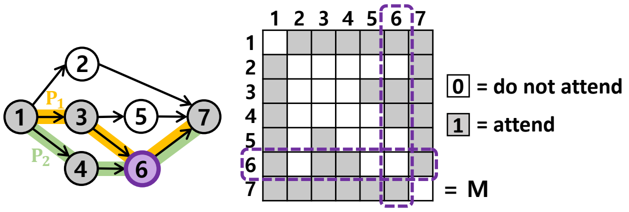

Figure 6. An example mask matrix M. Node 6 attends exclusively to nodes that appear in any path involving the node (P_1 and P_2). Nodes 1, 3, and 7 appear in P_1, and nodes 1, 4, and 7 appear in P_2; thus node 6 attends only to 1, 3, 4, and 7, as indicated by M.

Specifically, given the input node-embedding matrix $H^{(\ell-1)} \in \mathbb{R}^{N \times d}$ and the mask matrix M, the flow-aware global attention module computes its output node-embedding matrix $H_{global}^{(\ell)} \in \mathbb{R}^{N \times d}$ as follows:

$$H_{global}^{(\ell)} = \mathrm{MMHA}(H^{(\ell-1)}, H^{(\ell-1)}, H^{(\ell-1)}, M). \tag{5}$$

Here, MMHA is the Masked Multi-Head Attention module:

$$\mathrm{MMHA}(Q, K, V, M) = \mathrm{Concat}(\mathrm{head}_1, \dots, \mathrm{head}_s)W^0,$$

where $W^0 \in \mathbb{R}^{sd_v \times d}$ is the learnable projection matrix, s is the number of heads, and

$$\mathrm{head}_i = \mathrm{Attn}(QW_i^Q, KW_i^K, VW_i^V, M),$$
$$\mathrm{Attn}(Q, K, V, M) = \left(M \odot \mathrm{Softmax}\left(\frac{QK^T}{\sqrt{d_k}} \right) \right) V.$$

Here, $\odot$ is element-wise multiplication; and $W_i^Q \in \mathbb{R}^{d \times d_k}$, $W_i^K \in \mathbb{R}^{d \times d_k}$, and $W_i^V \in \mathbb{R}^{d \times d_v}$ denote i-th head's learnable query, key, and value projection matrices, respectively. We adhere to the condition $d_k = d_v = d/s$ for every head.

3.4. Overall framework: FLOWERFORMER

The overall framework of FLOWERFORMER is illustrated in Fig. 2. For each ℓ, it derives the output node embedding matrix $H^{(\ell)}$ from $H_{flow}^{(\ell)}$ (Eq. (4)) and $H_{global}^{(\ell)}$ (Eq. (5)) for the ℓ-th layer as follows:

$$H^{(\ell)} = \mathrm{FeedForward}(H_{flow}^{(\ell)} + H_{global}^{(\ell)}) \tag{6}$$

In our implementation, we employ a 2-layer MLP with ReLU activation [1] as the feedforward network. As shown in Fig. 2, note that we incorporate skip-connection and batch normalization in every module.

The output $H^{(\ell)}$ is used as the input of the next FLOWER layer, and for the first layer, we utilize a projected input feature matrix as the input by multiplying X with a learnable projection matrix $P \in \mathbb{R}^{D \times d}$, i.e., $H^{(0)} = XP$. Each FLOWER layer has a separate set of learnable parameters.

The node embeddings in the output $H^{(L)}$, where L represents the total number of FLOWER layers, are aggregated to drive the final embedding z_G of the input neural-architecture graph G as follows:

$$z_G = \mathrm{READOUT}(H^{(L)}),$$

For aggregation, we use mean pooling as the readout function in our implementation.

Application to performance prediction: The architecture embedding z_G is used for downstream tasks. For example, for performance prediction, it may serve as input to a regressor that outputs the estimated performance $\hat{y}_G$ as follows:

$$\hat{y}_G = \text{Regressor}(z_G).$$

In Sec. 4, we employ a fully connected layer as the regressor and utilize the following margin ranking loss for training both FLOWERFORMER and the regressor:

$$\mathcal{L} = \sum_{(i,j):y_i>y_j} \max(0, \text{margin} - (\hat{y}_i - \hat{y}_j)), \tag{7}$$

where y_i and y_j are the ground-truth performances of architectures G_i and G_j, respectively. For each pair of architectures G_i and G_j in the training set such that G_i outperforms G_j (i.e., $y_i > y_j$), the loss encourages $\hat{y}_i$ to be greater than $\hat{y}_j$ by at least a specified margin. Such designs for loss functions are commonly employed when it is important to make relative comparisons among instances (in our case, we compare neural architectures to recommend better ones).

4. Experiments

In this section, we review our experiments. For evaluation, we focus on the downstream task of predicting the performance of neural architectures. In Sec. 4.2, we compare the accuracies of FLOWERFORMER and six baseline methods, including two state-of-the-art methods (spec., TA-GATES [27] and NAR-Former [44]) using three performance prediction benchmark datasets composed of computer vision model architectures. In Sec. 4.3, we conduct an ablation study to validate each component of FLOWERFORMER. In Sec. 4.4, we extend our evaluation to datasets consisting of graph neural networks and auto speech recognition models. In Sec. 4.5, we examine the training and inference speed of FLOWERFORMER.

4.1. Experimental settings

Below, we provide an overview of our experimental setup.

4.1.1 Datasets

We evaluate the effectiveness of neural architecture encoding methods using five benchmark datasets designed for performance prediction, spanning three domains:

- **Computer vision:** We use three datasets: NAS-Bench-101 [45, 47], NAS-Bench-201 [9], and NAS-Bench-301 [48]. These datasets contain computer vision models.
- **Speech recognition:** We employ NAS-Bench-ASR [23], which consists of auto speech recognition architectures.
- **Graph learning:** We include NAS-Bench-Graph [31], which consists of graph neural networks.

Table 1. Basic information about the benchmark datasets we used. The sizes of the training and test splits used in [27] are reported. Refer to Sec. 4.1.3 for details about training and test splits.

Dataset	Domain	# trains	# tests
NAS-Bench-101	Computer vision	7,290	7,290
NAS-Bench-201		7,813	7,812
NAS-Bench-301		5,896	51,072
NAS-Bench-ASR	Speech recognition	4,121	4,121
NAS-Bench-Graph	Graph learning	13,103	13,103

Refer to Tab. 1, for basic statistics, and the supplementary material, for details including our preprocessing methods.

4.1.2 Baseline methods

We utilize six baseline approaches, categorized as follows: **(a) Graph neural networks:** GatedGCN [3] and directed acyclic graph neural network (DAGNN) [37], **(b) Graph transformers:** GraphGPS [32] and DAGFormer [21], and **(c) Neural architecture encoders:** TA-GATES [27] and NAR-Former [44], which are state-of-the-art methods for neural architecture performance prediction. We use the official implementations of these methods, and the links can be found in the supplementary material.

4.1.3 Training and Evaluation protocol

For model training and evaluation, we follow the setting in [27], including their training and test splits. We use a subset of the training split as the actual training set, varying the size of this subset: 1%, 5%, 10%, and 50% of the training split. We use the first 40 architectures in the test split as a validation set for hyperparameter tuning and early stopping, the remaining ones in the split as a test set. In each setting, we perform 9 trials using the three different splits and three different random seeds, and we report the mean and standard deviation across these trials. As accuracy metrics, we use Kendall's Tau [34] to assess overall performance and Precision@K (which measures the proportion of correctly predicted top-K architectures among the true top-K outperforming architectures) for the performance of identifying the best architectures. Note that these metrics are commonly employed in the field of neural architecture encoding [26, 27, 44].

4.2. Performance on computer vision benchmarks

In this subsection, we focus on the computer vision benchmarks for which we have extensive baseline methods. In Tabs. 2 and 3, we report the performance prediction accuracies of the considered methods using two metrics across different training instance ratios. Notably, FLOWERFORMER consistently outperforms all baseline methods across all settings in terms of Kendall's Tau. In terms of Precision@K,

Table 2. Kendall's Tau (scaled up by a factor of 100, mean and standard deviation over 9 trials) on three datasets: NAS-Bench-101, NAS-Bench-201, NAS-Bench 301. In each setting, the best performances are highlighted in green. **NA**: there is no trivial extension of NAR-Former to NAS-Bench-301, which consists of two-cell architectures. Note that, in every setting, FLOWERFORMER performs best.

Datasets	NAS-Bench-101				NAS-Bench-201				NAS-Bench-301				Avg.
Training portions	1%	5%	10%	50%	1%	5%	10%	50%	1%	5%	10%	50%	Rank
GatedGCN [3]	67.4 (6.0)	79.6 (4.1)	82.0 (5.1)	84.8 (5.9)	70.9 (1.8)	84.1 (0.6)	88.6 (0.3)	92.3 (0.1)	61.8 (2.4)	70.0 (0.9)	71.4 (1.0)	72.7 (1.5)	4.91
DAGNN [37]	72.4 (4.5)	82.9 (3.1)	84.4 (4.4)	85.9 (5.3)	75.8 (1.0)	87.5 (0.8)	90.6 (0.2)	92.6 (0.0)	61.5 (1.9)	70.9 (0.5)	73.4 (1.2)	76.1 (1.3)	2.50
GraphGPS [32]	70.6 (4.4)	81.7 (3.8)	83.9 (4.2)	85.9 (5.1)	71.3 (1.3)	82.5 (0.6)	87.8 (0.5)	92.7 (0.1)	59.7 (1.8)	69.3 (0.9)	70.7 (1.2)	73.8 (0.7)	4.75
DAGFormer [21]	73.0 (4.3)	75.6 (5.2)	77.2 (7.0)	80.9 (5.9)	73.0 (73.0)	84.9 (0.8)	88.8 (0.5)	92.7 (0.1)	61.3 (2.0)	70.7 (0.8)	72.1 (0.8)	74.8 (1.0)	3.91
NAR-Former [44]	59.4 (8.8)	72.0 (8.2)	75.5 (10.2)	79.8 (5.9)	62.3 (4.0)	80.7 (1.8)	87.3 (0.7)	88.9 (0.3)	**NA**	**NA**	**NA**	**NA**	-
TA-GATES [27]	70.8 (6.0)	82.3 (2.7)	83.9 (3.5)	86.3 (3.9)	77.7 (1.7)	86.3 (0.8)	88.7 (0.3)	91.4 (0.5)	61.3 (1.2)	68.9 (1.6)	71.8 (1.6)	75.4 (0.7)	3.83
FLOWERFORMER	75.0 (2.9)	86.1 (0.8)	88.1 (0.2)	89.6 (0.1)	80.0 (0.8)	89.8 (0.3)	91.3 (0.2)	92.9 (0.1)	64.2 (1.6)	72.2 (1.0)	73.6 (1.3)	77.5 (0.7)	1.00

Table 3. Precision@K (scaled up by a factor of 100, mean and standard deviation of 9 trials). The proportion of training samples is fixed to 5%. In each setting, the best performances are highlighted in green. **NA**: there is no trivial extension of NAR-Former to NAS-Bench-301, which consists of two-cell architectures. Note that, in most cases, FLOWERFORMER identifies top-k architectures most accurately.

Datasets	NAS-Bench-101 (5%)				NAS-Bench-201 (5%)				NAS-Bench-301 (5%)				Avg.
K (for P@Top K%)	1	5	10	50	1	5	10	50	1	5	10	50	Rank
GatedGCN [3]	44.4 (7.6)	65.6 (3.5)	76.2 (2.7)	90.5 (2.1)	42.3 (3.7)	68.5 (3.1)	80.9 (1.9)	94.1 (0.6)	19.1 (4.1)	55.2 (4.5)	71.8 (2.9)	85.4 (0.4)	4.83
DAGNN [37]	41.7 (5.9)	65.4 (4.2)	79.3 (2.9)	92.0 (1.3)	49.6 (6.2)	69.7 (3.0)	83.1 (0.7)	95.3 (0.9)	23.1 (2.1)	58.3 (3.4)	73.1 (1.5)	85.8 (0.4)	2.75
GraphGPS [32]	44.3 (12.2)	67.1 (2.7)	78.7 (1.9)	91.2 (2.0)	49.4 (4.6)	67.9 (4.9)	78.9 (3.4)	93.4 (0.3)	20.6 (2.1)	57.2 (3.8)	73.4 (2.5)	84.8 (0.5)	4.17
DAGFormer [21]	39.4 (7.9)	61.8 (5.6)	71.6 (5.0)	88.2 (2.4)	50.7 (5.8)	70.4 (2.9)	82.5 (2.3)	94.2 (0.5)	20.7 (3.4)	57.6 (3.7)	73.4 (2.5)	85.6 (0.4)	3.83
NAR-Former [44]	47.2 (9.9)	62.6 (7.9)	67.8 (8.4)	85.9 (5.2)	49.5 (6.5)	64.7 (2.0)	69.9 (2.0)	92.3 (1.0)	**NA**	**NA**	**NA**	**NA**	-
TA-GATES [27]	44.6 (9.7)	66.6 (4.0)	78.1 (4.6)	91.8 (1.2)	49.4 (3.1)	66.7 (3.3)	78.1 (2.8)	94.8 (0.7)	20.1 (5.0)	56.2 (6.1)	72.4 (3.6)	84.7 (0.7)	4.33
FLOWERFORMER	46.5 (11.2)	70.0 (1.5)	80.9 (1.8)	92.7 (1.7)	57.0 (5.4)	74.7 (1.8)	85.2 (1.3)	96.9 (0.7)	20.8 (3.7)	58.5 (2.5)	74.7 (2.4)	86.6 (0.6)	1.08

it performs best in 10 out of 12 settings, ranking second in the other settings. Two key observations are as follows.

The suboptimal performance of GraphGPS indicates that a graph transformer alone is insufficient in effectively representing neural architectures. Specifically, in terms of Kendall's Tau, the performance gap can be as large as 8.7 percentage points between FLOWERFORMER and GPS. We, thus, argue that our incorporation of information flows into a graph transformer, through the introduction of the flow encode module and the flow-aware global attention module, plays a pivotal role in FLOWERFORMER's success.

The superiority of FLOWERFORMER over TA-GATES highlights the importance of the global attention mechanism. While TA-GATES may capture the information flow at a local-level through its information propagation scheme, it does not adequately leverage the global context of neural architectures. FLOWERFORMER, on the other hand, uses the global attention mechanism to capture the graph-level (i.e., architecture-level) characteristics, empowering FLOWERFORMER to yield better representations of architectures.

In summary, our empirical findings substantiate that FLOWERFORMER serves as an effective predictor of neural architecture performance.

4.3. Ablation studies

In this subsection, we conduct ablation studies to validate the design choices made in FLOWERFORMER. Specifically, we aim to analyze the necessity of (a) asynchronous message-passing, (b) forward-backward message-passing, and (c) flow-aware global attention. To this end, we use four variants of FLOWERFORMER **(1)** without the flow encode module (i.e., eliminating both asynchronous and forward-backward message passing), **(2)** without asynchronous message passing, **(3)** without forward-backward message passing, and **(4)** without flow-aware global attention.

Table 4. Comparison with four variants of FLOWERFORMER in terms of Kendall's Tau, using the same setups as in Tab. 2. In each setting, the best performances are highlighted in green. **AS**: Asynchronous message passing. **FB**: Forward-backward message passing. **GA**: Global attention. In most cases, FLOWERFORMER, which is equipped with all components, outperforms all of its variants, thereby validating the effectiveness of each component.

Dataset	#	Components			Training Portions			
		AS	FB	GA	1%	5%	10%	50%
NB 101	**(1)**	✗	✗	✔	41.5(1.7)	42.5 (1.6)	41.1 (2.8)	43.1 (1.4)
	(2)	✗	✔	✔	65.5 (8.8)	56.8 (5.0)	53.8 (7.4)	69.6 (11.6)
	(3)	✔	✗	✔	76.7 (4.1)	83.9 (2.6)	84.6 (4.0	85.6 (5.4)
	(4)	✔	✔	✗	76.5 (3.0)	83.2 (3.9)	83.9 (5.1)	85.3 (6.3)
	-	✔	✔	✔	75.0 (2.9)	86.1 (0.8)	88.1 (0.2)	89.6 (0.1)
NB 201	**(1)**	✗	✗	✔	75.9 (1.2)	86.5 (0.2	88.2 (0.3)	89.7 (0.1)
	(2)	✗	✔	✔	73.7 (1.1)	85.6 (0.7)	89.2 (0.5)	92.9 (0.1)
	(3)	✔	✗	✔	76.2 (2.1)	88.6 (0.8)	90.9 (0.1)	92.9 (0.1)
	(4)	✔	✔	✗	79.3 (1.2)	89.5 (0.5)	91.1 (0.3)	92.9 (0.3)
	-	✔	✔	✔	79.0 (0.8)	89.8 (0.3)	91.3 (0.2)	92.9 (0.1)
NB 301	**(1)**	✗	✗	✔	63.3(2.7)	69.0 (2.8)	68.1 (2.9)	59.8 (2.3)
	(3)	✔	✗	✔	59.5(2.6)	69.0 (1.8)	50.2 (15.0)	45.3 (19.4)
	(4)	✔	✔	✗	60.9 (3.1)	69.8 (1.4)	70.9 (1.2)	67.7 (3.1)
	-	✔	✔	✔	64.2 (1.6)	72.2 (1.0)	73.6 (1.3)	77.5 (0.7)

As shown in Tab. 4, FLOWERFORMER, which is equipped with all the components, consistently outperforms all variants in most settings, confirming the efficacy of our design choices. Further observations deserve attention. First, the necessity of asynchronous message passing for capturing flows is confirmed by the superior performance of **(3)** over **(1)**, and that of FLOWERFORMER over **(2)**. Sec-

Table 5. Kendall's Tau (scaled up by a factor of 100, mean and standard deviation of 9 experiments) on two datasets beyond the computer vision domain: NAS-Bench-Graph (NB-G) [31] and NAS-Bench-ASR (NB-ASR) [23]. In each setting, the best performances are highlighted in green. In most cases, FLOWERFORMER performs best.

Dataset	Encoder	Training portions			
		1%	5%	10%	50%
NB-G	DAGNN [37]	48.1 (3.2)	64.4 (1.2)	67.4 (1.1)	🟩 73.1 (0.8)
	DAGFormer [21]	47.9 (0.6)	60.8 (1.6)	64.9 (1.0)	72.4 (0.3)
	TA-GATES [27]	33.1 (1.4)	34.1 (2.0)	35.4 (0.8)	35.7 (0.5)
	FLOWERFORMER	🟩 49.5 (1.1)	🟩 65.9 (1.3)	🟩 68.9 (0.6)	72.7 (0.2)
NB-ASR	DAGNN [37]	29.5 (3.9)	40.9 (2.4)	45.2 (1.3)	44.0 (0.4)
	DAGFormer [21]	29.9 (5.4)	42.5 (1.1)	45.3 (1.0)	34.6 (5.8)
	TA-GATES [27]	🟩 34.0 (2.3)	41.4 (2.0)	44.9 (2.2)	50.9 (0.8)
	FLOWERFORMER	31.1 (8.0)	🟩 44.0 (0.9)	🟩 47.3 (1.3)	🟩 52.2 (1.4)

ond, the advantage of forward-backward message passing is demonstrated by FLOWERFORMER's superiority over **(3)**. Lastly, incorporating flow-awareness into global attention is advantageous, as evidenced by FLOWERFORMER's advantage over variant **(4)**.

4.4. Performance in various domains

Since our input modeling does not require complex preprocessing, it can be readily applied to architectures across various domains. We apply FLOWERFORMER to graph neural networks on NAS-Bench-Graph and automatic speech recognition architectures on NAS-Bench-ASR. Among the baseline methods used in Sec. 4.2, we use the best method of each type: DAGNN, DAGFormer, and TA-GATES.

As shown in Tab. 5, FLOWERFORMER consistently performs best in most cases. These results indicate that FLOWERFORMER effectively captures important architectural characteristics across various domains. TA-GATES, which is tailored for encoding architectures in the computer vision domain, also shows strong performance in the domain of automatic speech recognition. TA-GATES effectively updates operation embeddings by multiplying operation embeddings and input information vectors, which is akin to convolutional mechanisms prevalent in auto speech recognition architectures. However, its effectiveness diminishes in scenarios where message passing between nodes, a key characteristic of graph neural networks, is required.

4.5. Training and inference speed

In this subsection, we compare the training and inference speeds of FLOWERFORMER and two state-of-the-art neural architecture encoding methods: NAR-Former and TA-GATES. To this end, we train all the models for 200 epochs with a batch size of 128, using an NVIDIA RTX 2080 GPU. We use NAS-Bench-101 with a training ratio of 1%. For a fair comparison, we exclude all additional time-consuming training strategies of NAR-Former and TA-GATES (e.g., input augmentation) in this experiment.

Table 6. Training and inference times on NAS-Bench-101 with a training ratio of 1%, 200 epochs, and a batch size of 128.

Encoder	Training time (sec)	Inference time (sec)	# Params
NAR-Former [44]	278.13 (13.01)	2.55 (0.07)	4,882,081
TA-GATES [27]	62.66 (0.54)	3.01 (0.21)	348,065
FLOWERFORMER	58.08 (1.39)	2.94 (0.07)	901,459

As shown in Tab. 6, FLOWERFORMER takes the shortest training time among the three methods. In particular, training FLOWERFORMER is 4.44× faster than training NAR-Former. This substantial speed advantage stems from the notable difference in model sizes, with NAR-Former having 5.35× the number of parameters compared to FLOWERFORMER. Compared to TA-GATES, FLOWERFORMER exhibits a slight speed advantage. Despite the small model size of TA-GATES, our specialized batch operations boost the training of FLOWERFORMER. Refer to the supplementary material for details of the batch operations.

In terms of inference time, there is not much difference among the three methods, and FLOWERFORMER ranks second. In practical scenarios, neural architecture performance prediction involves collecting labels (e.g., ground-truth performance) for the architectures in the training set, which requires time-consuming training of the architectures. For example, in the case of NAS-Bench-101, training just 1% of the architectures can take up to 24 GPU hours. Thus, inference speed is not a bottleneck, due to the extensive computational cost of training.

5. Conclusions

In this work, we propose FLOWERFORMER, a novel graph transformer model designed for neural architecture encoding. FLOWERFORMER excels at capturing information flows within neural architectures, considering both local and global aspects. Through comprehensive evaluations across five benchmarks for architecture performance prediction, FLOWERFORMER exhibits significant and consistent superiority over several state-of-the-art baseline methods, ultimately achieving state-of-the-art performance. Notably, FLOWERFORMER's superiority extends beyond computer-vision architectures, demonstrating its effectiveness for graph-learning and speech-recognition architectures.

Acknowledgements

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-00871, Development of AI Autonomy and Knowledge Enhancement for AI Agent Collaboration) (No. 2019-0-00075, Artificial Intelligence Graduate School Program (KAIST)).

References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018. 5

[2] Uri Alon and Eran Yahav. On the bottleneck of graph neu-

ral networks and its practical implications. *arXiv preprint arXiv:2006.05205*, 2020. 1

[3] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017. 6, 7

[4] Michail Chatzianastasis, George Dasoulas, Georgios Siolas, and Michalis Vazirgiannis. Graph-based neural architecture search with operation embeddings. In *ICCV*, 2021. 2

[5] Yaofo Chen, Yong Guo, Qi Chen, Minli Li, Wei Zeng, Yaowei Wang, and Mingkui Tan. Contrastive neural architecture search with neural architecture comparators. In *CVPR*, 2021. 1

[6] Ziye Chen, Yibing Zhan, Baosheng Yu, Mingming Gong, and Bo Du. Not all operations contribute equally: Hierarchical operation-adaptive predictor for neural architecture search. In *ICCV*, 2021. 1

[7] Zhe Chen, Hao Tan, Tao Wang, Tianrun Shen, Tong Lu, Qiuying Peng, Cheng Cheng, and Yue Qi. Graph propagation transformer for graph representation learning. In *IJCAI*, 2023. 1, 3

[8] Hsin-Pai Cheng, Tunhou Zhang, Yixing Zhang, Shiyu Li, Feng Liang, Feng Yan, Meng Li, Vikas Chandra, Hai Li, and Yiran Chen. Nasgem: Neural architecture search via graph embedding method. In *AAAI*, 2021. 2

[9] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *ICLR*, 2020. 6

[10] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *arXiv preprint arXiv:2012.09699*, 2020. 2

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 1

[12] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017. 1

[13] Md Shamim Hussain, Mohammed J Zaki, and Dharmashankar Subramanian. Global self-attention as a replacement for graph convolution. In *KDD*, 2022. 1, 2, 3

[14] Yinghui Jiang, Shuting Jin, Xurui Jin, Xianglu Xiao, Wenfan Wu, Xiangrong Liu, Qiang Zhang, Xiangxiang Zeng, Guang Yang, and Zhangming Niu. Pharmacophoric-constrained heterogeneous graph transformer model for molecular property prediction. *Communications Chemistry*, 6(1):60, 2023. 2

[15] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 2

[16] Devin Kreuzer, Dominique Beaini, Will Hamilton, Vincent Létourneau, and Prudencio Tossou. Rethinking graph transformers with spectral attention. In *NeurIPS*, 2021. 2

[17] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV*, 2018. 2

[18] Shun Lu, Jixiang Li, Jianchao Tan, Sen Yang, and Ji Liu. Tnasp: A transformer-based nas predictor with a self-evolution framework. In *NeurIPS*, 2021. 2

[19] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. *NeurIPS*, 2018. 1, 2

[20] Renqian Luo, Xu Tan, Rui Wang, Tao Qin, Enhong Chen, and Tie-Yan Liu. Semi-supervised neural architecture search. In *NeurIPS*, 2020. 1, 2

[21] Yuankai Luo, Veronika Thost, and Lei Shi. Transformers over directed acyclic graphs. In *NeurIPS*, 2023. 2, 5, 6, 7, 8

[22] Liheng Ma, Chen Lin, Derek Lim, Adriana Romero-Soriano, Puneet K Dokania, Mark Coates, Philip Torr, and Ser-Nam Lim. Graph inductive biases in transformers without message passing. In *ICML*, 2023. 1, 3

[23] Abhinav Mehrotra, Alberto Gil CP Ramos, Sourav Bhattacharya, Łukasz Dudziak, Ravichander Vipperla, Thomas Chau, Mohamed S Abdelfattah, Samin Ishtiaq, and Nicholas Donald Lane. Nas-bench-asr: Reproducible neural architecture search for speech recognition. In *ICLR*, 2020. 6, 8

[24] Grégoire Mialon, Dexiong Chen, Margot Selosse, and Julien Mairal. Graphit: Encoding graph structure in transformers. *arXiv preprint arXiv:2106.05667*, 2021. 2

[25] Hoang D Nguyen, Xuan-Son Vu, and Duc-Trong Le. Modular graph transformer networks for multi-label image classification. In *AAAI*, 2021. 2

[26] Xuefei Ning, Yin Zheng, Tianchen Zhao, Yu Wang, and Huazhong Yang. A generic graph-based neural architecture encoding scheme for predictor-based nas. In *ECCV*, 2020. 2, 6

[27] Xuefei Ning, Zixuan Zhou, Junbo Zhao, Tianchen Zhao, Yiping Deng, Changcheng Tang, Shuang Liang, Huazhong Yang, and Yu Wang. Ta-gates: An encoding scheme for neural network architectures. In *NeurIPS*, 2022. 2, 3, 6, 7, 8

[28] Peisong Niu, Tian Zhou, Qingsong Wen, Liang Sun, and Tao Yao. Chemistry guided molecular graph transformer. In *NeurIPS 2022 Workshop: AI for Science: Progress and Promises*, 2022. 1, 2

[29] Kenta Oono and Taiji Suzuki. Graph neural networks exponentially lose expressive power for node classification. *arXiv preprint arXiv:1905.10947*, 2019. 1

[30] Yunsheng Pang, Qiuhong Ke, Hossein Rahmani, James Bailey, and Jun Liu. Igformer: Interaction graph transformer for skeleton-based human interaction recognition. In *ECCV*, 2022. 2

[31] Yijian Qin, Ziwei Zhang, Xin Wang, Zeyang Zhang, and Wenwu Zhu. Nas-bench-graph: Benchmarking graph neural architecture search. In *NeurIPS*, 2022. 6, 8

[32] Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. In *NeurIPS*, 2022. 1, 2, 6, 7

[33] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying Wei, Wenbing Huang, and Junzhou Huang. Self-supervised graph transformer on large-scale molecular data. In *NeurIPS*, 2020. 2

[34] Pranab Kumar Sen. Estimates of the regression coefficient based on kendall's tau. *Journal of the American statistical association*, 63(324):1379–1389, 1968. 6

[35] Han Shi, Renjie Pi, Hang Xu, Zhenguo Li, James Kwok, and Tong Zhang. Bridging the gap between sample-based and one-shot neural architecture search with bonas. *NeurIPS*, 2020. 1

[36] Yu Shi, Shuxin Zheng, Guolin Ke, Yifei Shen, Jiacheng You, Jiyan He, Shengjie Luo, Chang Liu, Di He, and Tie-Yan Liu. Benchmarking graphormer on large-scale molecular modeling datasets. *arXiv preprint arXiv:2203.04810*, 2022. 2

[37] Veronika Thost and Jie Chen. Directed acyclic graph neural networks. In *ICLR*, 2021. 4, 6, 7, 8

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017. 1

[39] Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1903.11059*, 2019. 2

[40] Wei Wen, Hanxiao Liu, Yiran Chen, Hai Li, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. In *ECCV*, 2020. 1

[41] Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. In *AAAI*, 2021. 1, 2

[42] Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? In *NeurIPS*, 2021. 1

[43] Shen Yan, Kaiqiang Song, Fei Liu, and Mi Zhang. Cate: Computation-aware neural architecture encoding with transformers. In *ICML*, 2021. 2, 5

[44] Yun Yi, Haokui Zhang, Wenze Hu, Nannan Wang, and Xiaoyu Wang. Nar-former: Neural architecture representation learning towards holistic attributes prediction. In *CVPR*, 2023. 2, 3, 6, 7, 8

[45] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *ICML*, 2019. 2, 6

[46] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? In *NeurIPS*, 2021. 1, 2, 3

[47] Arber Zela, Julien Siems, and Frank Hutter. Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search. In *ICLR*, 2019. 6

[48] Arber Zela, Julien Siems, Lucas Zimmer, Jovita Lukasik, Margret Keuper, and Frank Hutter. Surrogate nas benchmarks: Going beyond the limited search spaces of tabular nas benchmarks. In *ICLR*, 2022. 6

[49] Yi Zheng, Rushin H Gindra, Emily J Green, Eric J Burks, Margrit Betke, Jennifer E Beane, and Vijaya B Kolachalama. A graph-transformer for whole slide image classification. *IEEE transactions on medical imaging*, 41(11):3003–3015, 2022. 2