NN-Former: Rethinking Graph Structure in Neural Architecture Representation

Supplementary Material

1. Methods Details

1.1. Implementation for ASMA

We present Python-style code for calculating the attention matrix in the ASMA module in Listing 1. ASMA is motivated by the importance of sibling nodes. In the accuracy prediction, sibling nodes provide complementary features, such as parallel 1x1 and 3x3 convolutions extracting pixel features and local aggregations, respectively. Although the two nodes are neither connected nor reachable through transitive closure, their information can influence each other. This conclusion has been studied in works such as Inception [40] and RepVGG [6]. In latency prediction, sibling nodes can run in parallel. For example, if two parallel 1x1 convolutions are merged into one, it takes only one CUDA kernel and fully utilizes parallel computing. Hence it is reasonable for ASMA to fuse the sibling nodes information directly.

Listing 1. Calculating the attention matrix in ASMA.

```
def attention_matrix(Q, K, A):
    # Q: query, K: key, A: adjacency matrix
    # Calculate the attention scores
    attn = torch.matmul(Q, K.mT) /
        math.sqrt(Q.size(-1))
    # Prepare attention masks
    pe = torch.stack([A, A.mT, A.mT @ A, A
        @ A.mT], dim=1)
    pe = pe + torch.eye(L, dtype=A.dtype,
        device=A.device)
    # Apply masking
    attn = attn.masked_fill(pe == 0,
        -torch.inf)
    # Softmax operation
    attn = F.softmax(attn, dim=-1)
    return attn
```

To implement the masking operation, the values at the non-zero positions remain unchanged, while the other values are set to minus infinity. Consequently, the softmax operation on these masked values results in zeroes.

1.2. Proof of sibling nodes identification

In the paper, we use $\boldsymbol{A}^T\boldsymbol{A}$ to represent sibling nodes that share the same successor. Here we provide trivial proof. $\boldsymbol{A}_{ij} = 1$ denotes there is a directed edge linked from node i to node k. Thus $\boldsymbol{A}^T_{ki} = 1$ denotes that there is a directed edge linked from node i to node k. Thus $\left(\boldsymbol{A}^T\boldsymbol{A}\right)_{kj} = \sum_v \boldsymbol{A}^T_{kv}\boldsymbol{A}_{vj} \geq \boldsymbol{A}^T_{ki}\boldsymbol{A}_{ij} = 1$, which denotes that node k and node j share a same successor i. Similar to $\boldsymbol{A}\boldsymbol{A}^T$, where $\left(\boldsymbol{A}\boldsymbol{A}^T\right)_{kj} \geq 1$ if node k and node j share a same predecessor.

1.3. Proof of Bi-directional Graph Isomorphism Feed-Forward Network

We begin by summarizing the BIGFFN as the common form of message-passing GNNs, and then prove the isomorphism property. Modern message-passing GNNs follow a neighborhood aggregation strategy, where we iteratively update the representation of a node by aggregating representations of its neighbors. To make comparison with modern GNNs, we follow the same notations, where the feature of node v is denoted as h_v. The l-th layer of a GNN is composed of aggregation and combination operation:

$$a_v^{(l)} = \text{AGGREGATE}\left(h_u^{(l)} : u \in \mathcal{N}(v)\right), \tag{14}$$

$$h_v^{(l)} = \text{COMBINE}\left(h_v^{(l-1)}, a_v^{(l)}\right), \tag{15}$$

where $h_v^{(l)}$ is the feature vector of node v at the l-th iteration/layer. In our cases, the graph is directional, thus the neighborhood $\mathcal{N}(v)$ is also divided into forward propagation nodes $\mathcal{N}^+(v)$ and backward propagation nodes $\mathcal{N}^-(v)$:

$$a_v^{(l)} = \text{AGGREGATE}\left(h_u^{(l-1)} : u \in \mathcal{N}^+(v) \cup \mathcal{N}^-(v)\right). \tag{16}$$

The AGGREGATE function in BGIFFN is defined as a matrix multiplication followed by concatenation:

$$\text{AGGREGATE} : \boldsymbol{H} \mapsto \text{Concat}\left(\boldsymbol{A}\boldsymbol{H}\boldsymbol{W}^+, \boldsymbol{A}^T\boldsymbol{H}\boldsymbol{W}^-\right), \tag{17}$$

where $\boldsymbol{W}^+$ and $\boldsymbol{W}^-$ are the linear transform for forward and backward propagation, respectively. This is equivalent to a bidirectional neighborhood aggregation followed by a concatenation operation:

$$a_v^{(l)} = \text{Concat}\left(\sum_{u\in\mathcal{N}^+(v)} h_u^{(l-1)}\boldsymbol{W}^+, \sum_{u\in\mathcal{N}^-(v)} h_u^{(l-1)}\boldsymbol{W}^-\right), \tag{18}$$

and the COMBINE function is defined as follows:

$$h_v^{(l)} = \text{ReLU}\left(h_v^{(l-1)}\boldsymbol{W}_1 + a_v^{(l)}\right)\boldsymbol{W}_2. \tag{19}$$

We quote Theorem 3 in [48]. For simple reference, we provide the theorem in the following:

Theorem 1 (Theorem 3 in [48]). *With a sufficient number of GNN layers, a GNN $\mathcal{M} : \mathcal{G} \mapsto \mathbb{R}^d$ maps any graphs G_1 and G_2 that the Weisfeiler-Lehman test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold:*

a) $\mathcal{T}$ aggregates and updates node features iteratively with

$$h_v^{(l)} = \phi\left(h_v^{(l-1)}, f\left(\left\{h_v^{(l-1)} : u \in \mathcal{N}(v)\right\}\right)\right), \tag{20}$$

where the function f, which operates on multisets, and φ are injective.

b) $\mathcal{T}$'s graph-level readout, which operates on the multiset of node features $\left\{h_v^{(l)}\right\}$, is injective.

Please refer to [48] for the proof. In our cases, the difference lies in condition a), where our tasks use directed acyclic graphs. Thus we modify condition a) as follows:

Theorem 2 (Modified condition for undirected graph). *a) $\mathcal{T}$ aggregates and updates node features iteratively with*

$$h_v^{(l)} = \phi\left(h_v^{(l-1)}, f\left(\left\{h_v^{(l-1)} : u \in \mathcal{N}^+(v) \cup \mathcal{N}^-(v)\right\}\right)\right), \tag{21}$$

where the function f, which operate on multisets, and φ are injective.

The proof is trivial, as it turns back to the original undirected graph. Following the Corollary 6 in [48], we can build our bidirectional graph isomorphism feed-forward network:

Corollary 1 (Corollary 6 in [48]). *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ so that for infinitely many choices of ϵ, including all irrational numbers, $h(c, X) = (1+\epsilon)\cdot f(c) + \sum_{x\in X} f(x)$ is unique for each pair (c, X), where $c \in \mathcal{X}$ and $X \subset \mathcal{X}$ is a multiset of bounded size. Moreover, any function g over such pairs can be decomposed as $g(c, X) = \varphi\left((1+\epsilon)\cdot f(c) + \sum_{x\in X} f(x)\right)$ for some function φ.*

In our cases, ϵ is substituted by a linear transform with weights $\boldsymbol{W}_1$. f is the aggregation function, and φ is the combine function. There exist choices of f and φ that are injective, thus the conditions are satisfied.

Furthermore, our BGIFFN distinguishes the forward and backward propagation, yielding stronger capability in modeling graph topology. Our method corresponds to a stronger "directed WL test", which applies a predetermined injective function z to update the WL node labels $k_v^{(l)}$:

$$k_v^{(l)} = z\left(k_v^{(l)}, \left\{k_v^{(l)} : u \in \mathcal{N}^+(v)\right\}, \left\{k_v^{(l)} : u \in \mathcal{N}^-(v)\right\}\right), \tag{22}$$

and the condition is modified as:

Theorem 3 (Modified condition for directed graph). *a) $\mathcal{T}$ aggregates and updates node features iteratively with*

$$h_v^{(l)} = \phi\left(h_v^{(l-1)}, f\left(\left\{h_v^{(l-1)} : u \in \mathcal{N}^+(v)\right\}\right), g\left(\left\{h_v^{(l-1)} : u \in \mathcal{N}^-(v)\right\}\right)\right), \tag{23}$$

where the function f and g, which operate on multisets, and φ are injective.

Proof. The proof is a trivial extension to Theorem 1. Let $\mathcal{T}$ be a GNN where the condition holds. Let G_1 and G_2 be any graphs that the directed WL-test (which means propagating on the directed graph) decides as non-isomorphic at iteration L. Because the graph-level readout function is injective, it suffices to show that $\mathcal{T}$'s neighborhood aggregation process embeds G_1 and G_2 into different multisets of node features with sufficient iterations. We will show that for any iteration l, there always exists an injective function φ such that $h_v^{(k)} = \varphi\left(k_v^{(l)}\right)$. This holds for $l = 0$ because the initial node features are the same for WL and GNN $k_v^{(0)} = h_v^{(0)}$. So φ could be the identity function for $k = 0$. Suppose this holds for iteration $k-1$, we show that it also holds for l. Substituting $h_v^{(l-1)}$ with $\varphi\left(h_v^{(l-1)}\right)$ gives us:

$$h_v^{(l)} = \phi\left(\varphi\left(h_v^{(l-1)}\right), f\left(\left\{\varphi\left(h_v^{(l-1)}\right) : u \in \mathcal{N}^+(v)\right\}\right), g\left(\left\{\varphi\left(h_v^{(l-1)}\right) : u \in \mathcal{N}^-(v)\right\}\right)\right), \tag{24}$$

Since the composition of injective functions is injective, there exists some injective function ψ so that

$$h_v^{(l)} = \psi\left(h_v^{(l-1)}, \left\{h_v^{(l-1)} : u \in \mathcal{N}^+(v)\right\}, \left\{h_v^{(l-1)} : u \in \mathcal{N}^-(v)\right\}\right). \tag{25}$$

Then we have

$$h_v^{(l)} = \psi \circ z^{-1} z\left(k_v^{(l)}, \left\{k_v^{(l)} : u \in \mathcal{N}^+(v)\right\}, \left\{k_v^{(l)} : u \in \mathcal{N}^-(v)\right\}\right), \tag{26}$$

and thus $\varphi = \psi \circ z^{-1}$ is injective because the composition of injective functions is injective. Hence for any iteration l, there always exists an injective function φ such that $h_v^{(l)} = \varphi\left(h_v^{(l-1)}\right)$. At the L-th iteration, the WL test decides that G_1 and G_2 are non-isomorphic, that is the multisets k_v^L are different for G_1 and G_2. The graph neural network $\mathcal{T}$'s node embeddings $\left\{h_v^{(L)}\right\} = \left\{\varphi\left(k_v^{(L)}\right)\right\}$ must also be different for G_1 and G_2 because of the injectivity of φ. □

1.4. Implementation for BGIFFN

We present Python-style code for the BGIFFN module in Listing 2. BGIFFN is intended to extend Graph Isomorpsim to the bidirectional modeling of DAGs. It extracts the topological features simply and effectively, assisting the Transformer backbone in learning the DAG structure. Various works use convolution to enhance FFN in vision [16] and language tasks [46]. It is reasonable for BGIFFN to assist Transformer in neural predictors.

Listing 2. Calculation for BGIFFN.

```
def bgiffn(x, A, W_1, W_forward,
    W_backward, W_2):
   # x: node features, A: adjacency matrix
   # W_1, W_forward, W_backward, W_2: the
       weight for linear transform
   aggregate = torch.cat((A @ x @
       W_forward, A.mT @ x @ W_backward),
       dim=-1)
   combine = F.relu(x @ W_1 + aggregate) @
       W_2
   return combine
```

2. Experiment Details

We present implementation details of our proposed NN-Former. For accuracy prediction, we show the experiment settings on NAS-Bench-101 in Section 2.1.1 and NAS-Bench-201 in Section 4.1. For latency prediction, we show the experiment settings on NNLQ [27] in Section 2.2.1.

2.1. Accuracy Prediction

For the network input, each operation type is represented by a 32-dimensional vector using one-hot encoding. Subsequently, this encoding is converted into a 160-channel feature by a linear transform and a layer normalization. The model contains 12 transformer blocks commonly employed in vision transformers [10]. Each block comprises ASMA and BGIFFN modules. The BGIFFN has an expansion ratio of 4, mirroring that of a vision transformer. The output class token is transformed into the final prediction value through a linear layer. Initialization of the model follows a truncated normal distribution with a standard deviation of 0.02. During training, Mean Squared Error (MSE) loss is utilized, alongside other augmentation losses as outlined in NAR-Former [52] with $\lambda_1 = 0.2$ and $\lambda 2 = 1.0$. The model is trained for 3000 epochs in total. A warm-up [15] learning rate from 1e-6 to 1e-4 is applied for the initial 300 epochs, and cosine annealing [28] is adopted for the remaining duration. AdamW [29] with a coefficient (0.9, 0.999) is utilized as the optimizer. The weight decay is set to 0.01 for all the layers except that the layer normalizations and biases use no weight decay. The dropout rate is set to 0.1. We use the Exponential Moving Average (EMA) [37] with a decay rate of 0.99 to alleviate overfitting. Each experiment takes about 1 hour to train on an RTX 3090 GPU.

2.1.1. Experiments on NAS-Bench-101.

NAS-Bench-101 [54] provides the performance of each architecture on CIFAR-10 [24]. It is an operation-on-node (OON) search space, which means nodes represent operations, while edges illustrate the connections between these nodes. Following the approach of TNASP [30], we utilize the validation accuracy from a single run as the target during training, and the mean test accuracy over three runs is used as ground truth to assess the Kendall's Tau [38]. The metrics on the test set are computed using the final epoch model, the top-performing model, and the best Exponential Moving Average (EMA) model on the validation set. The highest-performing model is documented.

2.1.2. Experiments on NAS-Bench-201.

NAS-Bench-201 offers three sets of results for each architecture, corresponding to CIFAR-10, CIFAR-100, and ImageNet-16-120. This study focuses on the CIFAR-10 dataset, consistent with the setup in TNASP [30].

NAS-Bench-201 [8] is originally operation-on-edge (OOE) search space, while we transformed the dataset into the OON format. NAS-Bench-201 contains the performance of each architecture on three datasets: CIFAR-10 [24], CIFAR-100 [24], and ImageNet-16-120 (a downsampled subset of ImageNet [5]). We use the results on CIFAR-10 in our experiments following previous TNASP [30], NAR-Former [52] and PINAT [31]. In the preprocessing, we drop the useless operations taht only have zeroized input or output. The metrics on the test set are computed using the final epoch model, the top-performing model, and the best Exponential Moving Average (EMA) model on the validation set. The highest-performing model is documented.

As for the results in the 10% setting, we argue that these results are not a good measurement. Concretely, the predictors are trained on the validation accuracy of NAS-Bench-201 networks, and evaluated on the test accuracy. We calculate Kendall's Tau between ground truth validation accuracy and test accuracy on this dataset which is 0.889. It indicates an unneglectable gap between the predictors' training and testing. Thus the results around and higher than 0.889 are less valuable to reflect the performance of predictors. For further studies, we also provide a new setting for this dataset. Both training and evaluation are conducted on the test accuracy of NAS-Bench-201 networks, and the training samples are dropped during evaluation. This setting has no gap between the training and testing distribution. As shown in Table 13, our methods surpass both NAR-Former [52] and NAR-Former V2 [53], showcasing the strong capability of our NN-Former.

Table 13. **Accuracy prediction results on NAS-Bench-201 [8] when the training and testing data follow the same distribution.** We use different proportions of data as the training set and report Kendall's Tau on the whole dataset.

Method	Publication	Training Samples 10% (1563)
NAR-Former [52]	CVPR 2023	0.910
NAR-Former V2 [53]	NeurIPS 2023	0.921
NN-Former (Ours)	-	**0.935**

2.2. Latency prediction

2.2.1. Experiments on NNLQ.

There are two scenarios on latecny prediction on NNLQ [27]. In the first scenario, the training set is composed of the first 1800 samples from each of the ten network types, and the remaining 200 samples for each type are used as the testing set. The second scenario comprises ten sets of experiments, where each set uses one type of network as the test set and the remaining nine types serve as the training set. The network input is encoded in a similar way as NAR-Former V2 [53]. Each operation is represented by a 192-dimensional vector, with 32 dimensions of one-hot operation type encoding, 80 dimensions of sinusoidal operation attributes encoding, and 80 dimensions of sinusoidal feature shape encoding. Subsequently, this encoding is converted into a 512-channel feature by a linear transform and a layer normalization. The model contains 2 transformer blocks, the same as NAR-Former V2 [53]. Each block comprises ASMA and BGIFFN modules. The BGIFFN has an expansion ratio of 4, mirroring that of a common transformer [10]. The output features are summed up and transformed into the final prediction value through a 2-layer feed-forward network. Initialization of the model follows a truncated normal distribution with a standard deviation of 0.02. During training, Mean Squared Error (MSE) loss is utilized. The model is trained for 50 epochs in total. A warm-up [15] learning rate from 1e-6 to 1e-4 is applied for the initial 5 epochs, and a linear decay scheduler is adopted for the remaining duration. AdamW [29] with a coefficient (0.9, 0.999) is utilized as the optimizer. The weight decay is set to 0.01 for all the layers except that the layer normalizations and biases use no weight decay. The dropout rate is set to 0.05. We also use static features as NAR-Former V2 [53]. Each experiment takes about 4 hours to train on an RTX 3090 GPU.

3. Extensive experiments

3.1. Ablation on hyperparameters

This work adopts a Transformer as the backbone, and the hyperparameters of Transformers have been well-settled in previous research. This article follows the common training settings (from NAR-Former) and has achieved good results. Apart from these hyperparameters, we provide an ablation on the number of channels and layers in the predictor as shown in Table 14:

Table 14. **Ablation studies on hyperparameters.** All experiments are conducted on the NAS-Bench-101 [54] with the 0.04% training set. (a) Ablation study on the number of channels. (b) Ablation study on the number of transformer layers.

(a)

Num of Channels	KT
64	0.748
128	0.758
160 (Ours)	0.765

(b)

Num of Layers	KT
6	0.744
9	0.760
12 (Ours)	0.765

3.2. Comparison with Zero-Cost predictors

Zero-cost proxies are lightweight NAS methods, but they performs not as well as the model-based neural predictors.

Table 15. Comparison with zero-cost predictors.

NAS search space	NAS-Bench-101↑	NAS-Bench-201↑
grad_norm [1]	0.20	0.58
snip [1]	0.16	0.58
NN-Former	0.71	0.80

4. Model Complexity

4.1. Theoretical Analysis

Our ASMA method has less or equal computational complexity than the vanilla attention. On the dense graph, the vanilla self-attention has a complexity of $O(N^2)$ where N denotes the number of nodes. With the sibling connection preprocessed, our ASMA also has a complexity of $O(N^2)$. On sparse graphs, the vanilla self-attention is still a global operation thus the complexity is also $O(N^2)$. Our ASMA has a complexity of $O(NK)$, where K is the average degree and $K << N$ on sparse graphs. In practical applications, sparse graphs are common thus our method is efficient. The latency prediction experiments in the paper show that our predictor can cover the DAGs from 20 200 nodes, which is applicable for practical use.