Learning to Flow from Generative Pretext Tasks for Neural Architecture Encoding

Sunwoo Kim[1], **Hyunjin Hwang**[1], **Kijung Shin**[1,2]
[1]Kim Jaechul Graduate School of AI, [2]School of Electrical Engineering
Korea Advanced Institute of Science and Technology (KAIST)
{kswoo97, julia510, kijungs} @ kaist.ac.kr

Abstract

The performance of a deep learning model on a specific task and dataset depends heavily on its neural architecture, motivating considerable efforts to rapidly and accurately identify architectures suited to the target task and dataset. To achieve this, researchers use machine learning models—typically neural architecture encoders—to predict the performance of a neural architecture. Many state-of-the-art encoders aim to capture information flow within a neural architecture, which reflects how information moves through the forward pass and backpropagation, via a specialized model structure. However, due to their complicated structures, these flow-based encoders are significantly slower to process neural architectures compared to simpler encoders, presenting a notable practical challenge. To address this, we propose FGP, a novel pre-training method for neural architecture encoding that trains an encoder to capture the information flow without requiring specialized model structures. FGP trains an encoder to reconstruct a flow surrogate, our proposed representation of the neural architecture's information flow. Our experiments show that FGP boosts encoder performance by up to 106% in Precision@1%, compared to the same encoder trained solely with supervised learning.

1 Introduction

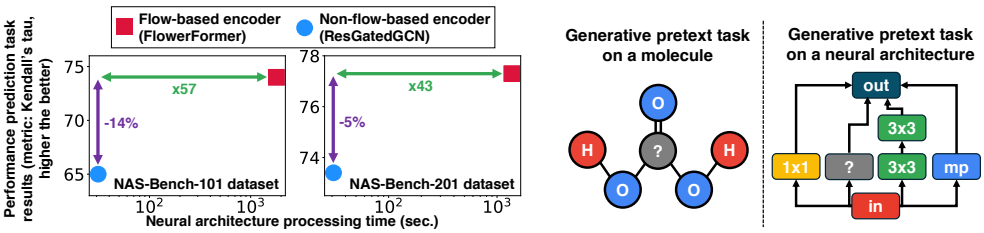
Deep learning has achieved outstanding performance across diverse machine learning tasks, including those in computer vision [9, 57]. However, a particular neural architecture (i.e., deep-learning model) that performs well on one task or dataset may underperform on others, highlighting the importance of choosing an architecture suited to the target task and dataset [29, 11, 35]. A naive approach—exhaustively training and evaluating each neural architecture—incurs substantial costs since each architecture requires expensive computational resources and time for training and evaluation on the target task and dataset.

A promising approach for alleviating expensive training and evaluation costs is to use machine learning techniques to *predict* the performance of neural architectures. Due to their effectiveness, performance predictors are widely leveraged for neural architecture search to find good neural architectures rapidly and accurately [10, 45, 16, 33]. Recently, various neural architecture encoders, which are typically deep learning models, have been developed to be used for performance predictors [34, 49, 15, 32].

The focus of many state-of-the-art neural architecture encoders [34, 15] lies in capturing the *information flow* within a neural architecture—a concept that describes how input data is propagated in the forward pass and how gradients flow during the backpropagation. To capture the information flow, these flow-based encoders use complex model structures, especially graph neural networks with asynchronous message passing [40], specially designed to capture the flow within an architecture.

Challenge 1. However, these flow-based encoders face a practical challenge related to efficiency. The complicated structures of flow-based encoders result in significantly longer processing times

39th Conference on Neural Information Processing Systems (NeurIPS 2025).



(a) The total processing time of FlowerFormer [15] and ResGatedGCN [3] for the 14K and 15K architectures in Nas-Bench-10 and NAS-Bench-201, respectively.

(b) Example generative pretext tasks: deep learning models are trained to predict the masked atoms or operations (colored in gray).

Figure 1: **Remaining challenges in neural architecture encoding. Challenge 1**: Regarding model structures, as shown in (a), flow-based encoders, though more effective, are considerably slower than non-flow-based encoders. **Challenge 2**: Regarding pretext objectives, in (b), while a deep learning model learns chemical rules by identifying masked atoms in a molecule, it gains no clear training guidance when identifying masked operations in a neural architecture, where most operations are possible options due to the absence of explicit rules.

compared to the simpler structures of non-flow-based encoders. In particular, flow-based encoders *sequentially* perform message passing according to the graph's topological order (see Section 3.2.1), instead of computing all messages simultaneously as in non-flow-based encoders; this sequential process results in significantly greater computation time. Specifically, as shown in Figure 1 (a), a flow-based encoder (spec., FlowerFormer [15]) is up to 57 times slower than non-flow-based encoders [3, 47] (refer to Appendix B.1 for further results). Since performance predictors aim to quickly identify effective neural architectures, this slower processing time poses a considerable practical bottleneck when using flow-based encoders.

Another line of research focuses on the effective pre-training of neural architecture encoders. Neural architecture encoders require sufficient training data—architectures paired with their ground-truth performance on the target task and dataset—since deep learning models are often vulnerable to the overfitting issue. However, obtaining enough neural architectures along with their performance entails heavy costs. To address this, various *generative pre-training methods* for neural architecture encoding have been explored [48, 17]. Note that in various domains, it is known that by using well-designed generative pretext tasks, deep learning models can capture data patterns [12, 6, 13, 19] that lead to performance improvements in label-scarce scenarios [55, 53, 18].

Challenge 2. However, existing generative pre-training methods, which are often adapted from other domains [48, 17], may be sub-optimal for neural architecture encoding. Consider a generative pretext task that involves predicting masked parts of the input, used for both neural architectures [17] and molecules [13]. In Figure 1 (b), only a limited set of atoms can occupy the masked area due to chemical rules, which deep learning models learn through this task. In contrast, such rules are largely absent in neural architectures; as shown in Figure 1 (b), most operations are possible options for the masked part. This characteristic makes it unclear what specific advantage the model gains from this generative pretext task in the context of neural architecture.

To address both key challenges, we propose a novel and specialized generative pre-training method for neural architecture encoding, **FGP** (**F**low-based **G**enerative **P**re-training). FGP enables an encoder to capture the information flow within a neural architecture, even if the encoder structure is not specially designed to capture the information flow (e.g., the non-flow-based encoders [47, 3]). We train an encoder to reconstruct *flow surrogate*, a proposed representation of a neural architecture's information flow, allowing the encoder to capture the information flow of the architecture. FGP addresses the two practical challenges outlined earlier: (1) it enables encoders to become flow-aware without requiring a specialized model structure that would increase computational time significantly, and (2) it provides clear training guidance to the encoder, allowing it to effectively learn the information flow within a neural architecture through our pretext task.

We demonstrate the effectiveness of our pre-training method compared to baseline pre-training methods across multiple downstream tasks, including performance prediction and neural architecture search. Specifically, compared to the strongest baseline method [56], FGP shows up to 46.5% gain in terms of Precision@1% in performance prediction. Our key contributions are as follows:

- We propose a novel generative pre-training method, FGP, for neural architecture encoding. FGP trains a neural architecture encoder to learn the information flow within a neural architecture even without a specialized encoder structure (Section 3).
- In the performance prediction experiment, FGP outperforms the baseline pre-training methods in 23 out of 27 settings, demonstrating its efficacy (Section 4.2).
- In the neural architecture search (NAS) experiment—a key application of neural architecture encoders—FGP outperforms the baseline pre-training methods (Section 4.3).

Our code and datasets are available at https://github.com/kswoo97/FGPAnom.

2 Related Work and Preliminaries

In this section, we first review related studies, followed by an outline of the preliminary concepts essential to our study.

2.1 Related work

2.1.1 Neural architecture encoding.

Neural architecture encoding aims to learn good representations of neural architectures [49, 15, 34, 32, 50]. Its notable application is neural architecture performance prediction, which can reduce the cost of finding a proper neural architecture for the target task and dataset.

Neural architectures, including Transformers [5] and convolutional neural networks [51], can be expressed as graphs where nodes represent operations (e.g., 3x3 conv and max pooling) and edges represent connections between them. Therefore, graph neural networks (GNNs) [3, 47, 23] are widely used as neural architecture encoders [42, 43, 48, 4, 10]. GNNs encode a neural architecture graph by aggregating and transforming features from the neighborhood of each node.

Recent works have increasingly focused on capturing the information flow within a given neural architecture by using specialized model structures [32, 34, 15]. A representative example is FlowerFormer [15], a state-of-the-art neural architecture encoder that employs sequential information processing to imitate the forward-pass and backpropagation in a neural architecture. Although more effective, these flow-based encoders require significantly more processing time, due to their sequential processing, compared to simpler GNN-based encoders (refer to Figure 1 (a)).

2.1.2 Pre-training for neural architecture encoding.

Due to the high cost of obtaining a neural architecture's ground-truth performance on the target task and dataset (i.e., the architecture's 'label'), it is essential to train accurate performance predictors—typically using neural architecture encoders—in label-scarce scenarios, as discussed in Section 1. To this end, generative pre-training methods [22, 12], known for enhancing the generalization capabilities of deep learning models [55, 53, 18], have been adopted. These methods leverage unlabeled neural architectures (i.e., those with unknown ground-truth performance) to pre-train a neural architecture encoder and fine-tune the encoder on the target downstream task, such as the performance prediction tasks [48, 17].

Arch2vec [48] is based on a variational graph autoencoder [22], which trains a neural architecture encoder to reconstruct edges in graphs representing neural architectures. GMAE [17] builds on masked autoencoder [12], where certain operations within a neural architecture are masked, and the encoder is trained to predict these masked operations.

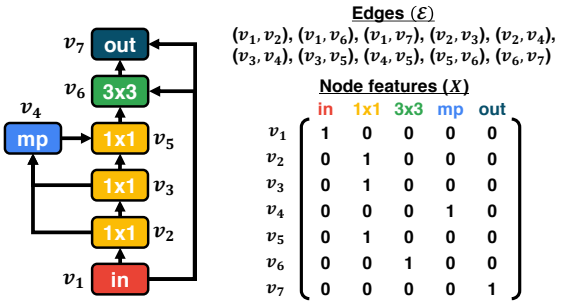
In contrast, Zhao et al. [56] leveraged zero-cost proxies [1, 25, 14], which are neural architecture's statistics correlated with its ground-truth performance. They trained a neural architecture encoder to predict the zero-cost proxies of a neural architecture, enabling it to learn which types of architectures are more likely to yield high performance.

2.2 Preliminary

Neural architecture graph. A neural architecture is often modeled as a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, defined by a set of nodes $\mathcal{V} = \{v_1, \cdots, v_{|\mathcal{V}|}\}$ that represent operations, and a set of directed

edges $\mathcal{E} \subseteq \{(v_i, v_j) : v_i, v_j \in \mathcal{V}\}$ that represent connections between consecutive operations (refer to Figure 2 for an example). Specifically, when operation v_j follows operation v_i, the two are connected by a directed edge (v_i, v_j), with v_i as the source and v_j as the destination. Each node is assigned a feature vector, typically a one-hot encoded vector indicating its operation type (refer to Figure 2). We denote the feature vector matrix as $\mathbf{X} \in \{0,1\}^{|\mathcal{V}| \times |O|}$, where $\mathcal{O}$ represents the set of all possible operations. The i-th row of $\mathbf{X}$, denoted by $\mathbf{X}_{i,:}$, corresponds to the feature vector of node v_i. Therefore, a neural architecture can also be expressed as $\mathcal{G} = (\mathbf{X}, \mathcal{E})$.

Neural architecture encoders. Neural architecture encoders transform a neural architecture graph $\mathcal{G} = (\mathbf{X}, \mathcal{E})$ into its vector representation (i.e., embedding). Formally, given $\mathcal{G}$ as an input, a neural architecture encoder f_θ produces a vector $\mathbf{z} \in \mathbb{R}^d$ (i.e., $f_\theta(\mathbf{X}, \mathcal{E}) = \mathbf{z}$). For encoders that give embeddings for each operation [47, 15] (i.e., $f_\theta(\mathbf{X}, \mathcal{E}) = \mathbf{Z}' \in \mathbb{R}^{|\mathcal{V}| \times d}$), we apply mean pooling to obtain $\mathbf{z}$, as in [15].



(a) An architecture from NAS-Bench-101. **(b)** Graph representation of the example neural architecture in **(a)**.

Figure 2: **Graph modeling of a neural architecture.** An example of how a neural architecture is modeled as a directed acyclic graph.

3 Proposed method

In this section, we introduce our proposed generative pre-training method for neural architecture encoding, **FGP** (**F**low-based **G**enerative **P**re-training). FGP trains a neural architecture encoder to learn the information flow within a given neural architecture, even if the encoder lacks a specialized structure for capturing this flow.

3.1 Motivation, challenge, and overview

Motivation. Information flow refers to how input data propagates through a neural network during the forward pass and how gradients are transmitted back during the backpropagation. This concept plays a fundamental role in understanding how a neural network learns and makes predictions, making it a critical feature of neural architectures [32, 34, 15]. Although state-of-the-art neural architecture encoders successfully capture the information flow through specialized model structures [34, 15], complexity leads to significantly longer processing times compared to simpler, non-flow-based encoders (refer to Figure 1 (a)). Therefore, we anticipate that training a simple neural architecture encoder—especially without a specialized structure for information flow—to learn this flow can address the computational efficiency issue while enhancing the encoder's overall effectiveness.

Challenge. The main challenge in learning the information flow is defining an appropriate *training objective*. To address this, we introduce *flow surrogate*, a vector representing the information flow within a neural architecture. This surrogate serves as a pre-training objective, guiding the encoder to understand the information flow in neural architectures. We obtain the surrogate by passing random vectors through a directed acyclic graph that represents an architecture, which is a highly simplified simulation of a neural network's forward pass and backpropagation.[1] Notably, each architecture has its own flow surrogate, which can be obtained with a one-time computational effort without any learning procedure. This does not require prior knowledge of the architecture's ground-truth performance on the target task. We detail the process of obtaining the flow surrogate (i.e., pre-training objective) in Section 3.2.

Overview. We now provide an overview of FGP. Given a set of unlabeled neural architectures (i.e., architectures with unknown ground-truth performance on the target task and dataset), we first express each architecture as a graph (i.e., neural architecture graph), as outlined in Section 2.2. Next, we obtain each architecture's flow surrogate (i.e., pre-training objective), as detailed in Section 3.2. Lastly, we train a neural architecture encoder to learn the information flow by reconstructing the obtained flow surrogate, as detailed in Section 3.3.

[1] We analyze the alternatives and reliability of using random vectors in Appendix B.9.

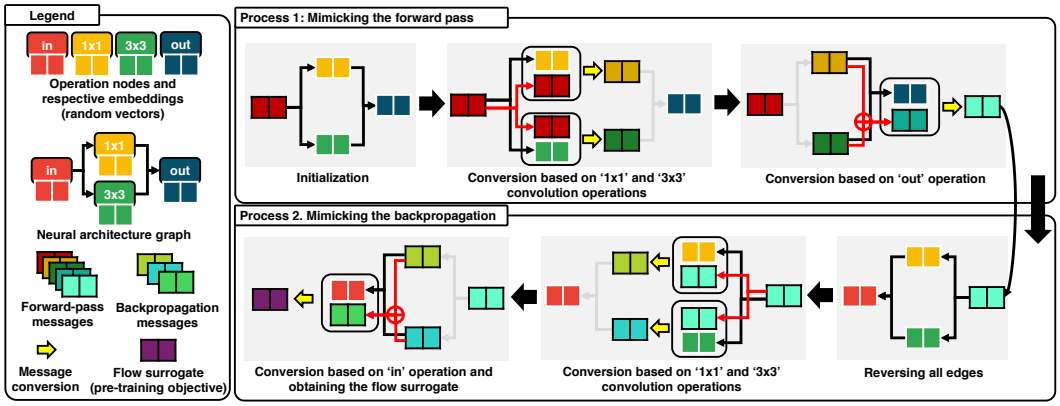


Figure 3: **Generation of the proposed pre-training objective.** Red edges mark active computation for the current step; gray edges indicate inactive computation. Starting from 'in' (input), a message (spec., randomly-drawn embedding of 'in') flows through the '1x1' and '3x3' operations, where it is converted depending on each operation type. The two converted messages are summed and flow to 'out' (output) for further conversion. This process is then reversed, and the message flows back through the operations until it returns to 'in'. The resulting message, called the architecture's flow surrogate, serves as the pre-training objective.

3.2 Proposed pre-training objective

We describe the process for obtaining the flow surrogate, our proposed representation of a neural architecture's information flow, which serves as our pre-training objective. We sequentially propagate random vectors through a directed acyclic graph that represents a neural architecture. Specifically, the random vectors, called messages, are passed between nodes (operations) along edges (connections between operations). When a node receives messages from other nodes, its message is converted based on its specific operation type, capturing unique operational characteristics. Notably, to simulate both the forward pass and backpropagation in a highly simplified manner, we: (1) propagate messages in the forward direction along the edges to model the forward pass, and (2) repeat the process in the reverse direction to model backpropagation. An overview is in Figure 3. After completing both propagation steps, the final converted message(s) serve as the flow surrogate for the corresponding neural architecture. We detail this process in three steps: (1) assigning a topological order, (2) mimicking the forward pass, and (3) mimicking the backpropagation. Note that this process resembles how flow-based encoders generate outputs. However, in our approach, the process is used only **once** per architecture (with random vectors) to obtain pre-training objectives and not repeated for training.

3.2.1 Assigning topological order

For a given neural architecture, we first obtain its graph expression $\mathcal{G} = (\mathbf{X}, \mathcal{E})$, as outlined in Section 2.2. Recall that this graph is a directed acyclic graph. In this graph, we assign a topological order to each node, indicating the sequence in which each node receives the information flow. An example of topological order assignment is provided in Figure 4. Order-1 nodes are defined as nodes without any incoming edges. After finding order-1 nodes, we remove them and the edges where they serve as source nodes from the graph. Next, order-2 nodes are those that, in the modified graph, have no incoming edges. We repeat this process until all nodes have been assigned an order. We denote a set of order-t nodes as $\mathcal{V}^{(t)}$, and denote the last order as T. Thus, the node set $\mathcal{V}$ is split into disjoint subsets $\mathcal{V}^{(1)}, \ldots, \mathcal{V}^{(T)}$.

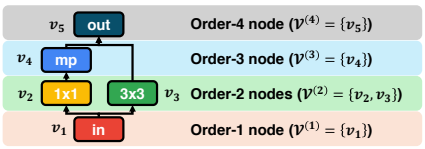


Figure 4: **Example of topological order assignment in a neural architecture graph.** Assignment results are ($\mathcal{V}^{(1)} = \{v_1\}$, $\mathcal{V}^{(2)} = \{v_2, v_3\}$, $\mathcal{V}^{(3)} = \{v_4\}$, and $\mathcal{V}^{(4)} = \{v_5\}$), and $T = 4$.

3.2.2 Mimicking the forward pass

Once the topological order assignment (Section 3.2.1) is complete, we model the forward pass of a neural architecture. To this end, we propagate forward-pass messages from node to node along

the edges of the neural architecture graph. Here, the forward-pass message of a node represents the hidden state at the corresponding node (i.e., operation) within the neural architecture, and each node has a distinct forward-pass message. We refer to these forward-pass messages as *fp-messages*. When a node v_i receives fp-messages from other nodes along edges, the fp-message of v_i is converted according to the incoming fp-messages and the operation type of v_i. The process of propagation and conversion proceeds in the order of $\mathcal{V}^{(1)}, \mathcal{V}^{(2)}, \ldots, \mathcal{V}^{(T)}$, until the fp-messages of order-T nodes (i.e., $v_j \in \mathcal{V}^{(T)}$) are converted. An example is provided in the 'Process 1' box of Figure 3.

We denote a fp-message of a node v_i as $\mathbf{f}_i \in \mathbb{R}^k$. Also, each node $v_i \in \mathcal{V}$ has an embedding, denoted by $\mathbf{h}_i \in \mathbb{R}^k$, based on its operation type. Specifically, by using a matrix $\mathbf{P} \in \mathbb{R}^{|\mathcal{O}| \times k}$ randomly sampled from $\mathcal{N}(0, \sigma^2)$, we compute the node embedding matrix $\mathbf{H} = \mathbf{PX}$, where its $i-$th row corresponds to v_i's embedding (i.e., $\mathbf{H}_{i,:} = \mathbf{h}_i$). Note that nodes performing the same operation share the same embedding across all architectures. Additionally, we set the fp-messages of the order-1 nodes, which do not have any incoming edges, with a *randomly initialized vector* $\mathbf{r} \in \mathbb{R}^k$ (i.e., $\mathbf{f}_j = \mathbf{r}, \forall v_j \in \mathcal{V}^{(1)}$). Note that fp-messages of all order-1 nodes are initialized as $\mathbf{r}$ in every neural architecture.

We formalize the propagation and conversion process at node $v_i \in \mathcal{V}^{(t)}$. When multiple fp-messages arrive at v_i (e.g., 'out' operation node in Figure 3), we first aggregate these arriving fp-messages using sum pooling as $\mathbf{m}_i = \sum_{v_j \in \mathcal{N}^{(i)}} \mathbf{f}_j$, where $\mathcal{N}^{(i)} = \{v_j : (v_j, v_i) \in \mathcal{E}\}$. In this process, using mean or max (rather than sum) pooling for neighbor aggregation yields lower performance than our approach (refer to Appendix B.11).

Next, we convert the pooled fp-messages $\mathbf{m}_i \in \mathbb{R}^k$ according to the operation of v_i, obtaining the converted fp-message of v_i, denoted by $\mathbf{f}_i \in \mathbb{R}^k$, as $\mathbf{f}_i = \alpha \mathbf{m}_i + (1-\alpha)\texttt{ReLU}\left([\mathbf{h}_i \| \mathbf{m}_i]\mathbf{W}\right)$, where $[\mathbf{a}\|\mathbf{b}]$ is a concatenation of vectors $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{W} \in \mathbb{R}^{2k \times k}$ is a fixed projection matrix, and $\alpha \in [0, 1]$ is a weighing hyperparameter. Here, $\mathbf{W}$ and α are shared across all architectures and nodes. When the fp-messages of order-T nodes (i.e., $v_j \in \mathcal{V}^{(T)}$) are converted, the process of mimicking the forward pass is complete.

3.2.3 Mimicking the backpropagation

Once the process of mimicking the forward pass (Section 3.2.2) is done, we do a reverse propagation process. This process aims to mimic the backpropagation step in a neural network. To this end, we propagate backpropagation messages, which we call *bp-messages*. The bp-message of a node represents the gradient at the corresponding node (i.e., operation) within the neural architecture, and each node has a distinct bp-message. The overall process (i.e., propagation and conversion) is similar to that for mimicking the forward pass (Section 3.2.2), but this process proceeds in the order of $\mathcal{V}^{(T)}, \mathcal{V}^{(T-1)}, \ldots, \mathcal{V}^{(1)}$, until the bp-messages of order-1 nodes (i.e., $v_j \in \mathcal{V}^{(1)}$) are converted. An example is in the 'Process 2' box of Figure 3. We denote the bp-message of node v_i as $\mathbf{b}_i \in \mathbb{R}^k$. Furthermore, each order-T node's bp-message is initialized to the corresponding node's fp-message (i.e., $\mathbf{b}_j = \mathbf{f}_j, \forall v_j \in \mathcal{V}^{(T)}$), to mimic the dependence of backpropagation on the forward pass.

We formalize the propagation and conversion process at node $v_i \in \mathcal{V}^{(t)}$. Here, bp-messages arrive from the nodes at the endpoints of v_i's outgoing edges, and when multiple bp-messages arrive at v_i (e.g., 'in' operation node in Figure 3), we aggregate these incoming bp-messages using sum pooling as follows: $\mathbf{m}'_i = \sum_{v_j \in \mathcal{K}^{(i)}} \mathbf{b}_j$, where $\mathcal{K}^{(i)} = \{v_j : (v_i, v_j) \in \mathcal{E}\}$.

We then convert the pooled bp-messages $\mathbf{m}'_i \in \mathbb{R}^k$ according to the operation of v_i, obtaining the converted bp-message of v_i, denoted by $\mathbf{b}_i \in \mathbb{R}^k$, as $\mathbf{b}_i = \alpha \mathbf{m}'_i + (1-\alpha)\texttt{ReLU}\left([\mathbf{h}_i \| \mathbf{m}'_i]\mathbf{W}\right)$. When the bp-messages of order-1 nodes (i.e., $v_j \in \mathcal{V}^{(1)}$) are converted, the process of mimicking the backpropagation is complete.

After mimicking the forward-pass and backpropagation, we sum the bp-messages of the order-1 nodes to obtain the flow surrogate $\mathbf{s} \in \mathbb{R}^k$ (i.e., $\mathbf{s} = \sum_{v_i \in \mathcal{V}^{(1)}} \mathbf{b}_i$). The resulting vector $\mathbf{s}$ serves as the pre-training objective for the corresponding architecture, the flow surrogate representing the information flow within $\mathcal{G}$. Although this process can be repeated over multiple rounds, such methods underperform compared to our approach, as detailed in Appendix B.10. Further discussion of the surrogate's expressiveness and theoretical properties is in Appendices B.12 and B.13, respectively.

3.3 Proposed flow generative pre-training

We now present the details of FGP, our flow generative pre-training method for neural architecture encoding. As outlined in Section 3.1, FGP trains a neural architecture encoder to reconstruct the flow surrogate (i.e., pre-training objective) $\mathbf{s} \in \mathbb{R}^k$ (Section 3.2) of a given neural architecture (Section 3.2). Then, *any* encoder, without requiring a specialized model structure, is trained to capture the information flow through three steps: (1) encoding, (2) decoding, and (3) computing the pre-training loss.

Encoding and decoding. For a given neural architecture, we start by deriving its graph expression $\mathcal{G} = (\mathbf{X}, \mathcal{E})$, as described in Section 2.2. We then encode this graph into an embedding $\mathbf{z} \in \mathbb{R}^d$ using a neural architecture encoder f_θ (i.e., $f_\theta(\mathbf{X}, \mathcal{E}) = \mathbf{z}$), which we aim to train. After, we decode the embedding by using an MLP decoder g_ϕ to obtain the reconstructed surrogate $\hat{\mathbf{s}} \in \mathbb{R}^k$ (i.e., $g_\phi(\mathbf{z}) = \hat{\mathbf{s}}$), ensuring the encoder focuses on learning generalizable representations while the decoder handles task-specific transformations.

Pre-training loss. Then, we compute the reconstruction loss $\mathcal{L}_{rec}$ by measuring the squared ℓ_2-distance between the original surrogate $\mathbf{s}$ (Section 3.2) and the reconstructed surrogate $\hat{\mathbf{s}}$ (i.e., $\mathcal{L}_{rec} = \|\mathbf{s} - \hat{\mathbf{s}}\|_2^2$). Notably, FGP can be combined with auxiliary learning objectives, such as predicting zero-cost proxies of a neural architecture [1, 27]. The overall training loss is $\mathcal{L} = \lambda_1 \mathcal{L}_{rec} + \lambda_2 \mathcal{L}_{aux}$, where λ_1 and λ_2 are loss-weighing hyperparameters. Details on our usage of auxiliary learning objectives are provided in Appendix D.2. The learnable parameters θ and ϕ are optimized to minimize $\mathcal{L}$ via gradient descent. After pre-training, the trained encoder f_θ can be fine-tuned to perform certain downstream tasks, such as neural architecture's performance prediction.

4 Experiment

In this section, we demonstrate the effectiveness of FGP in several applications, including performance prediction and neural architecture search. Specifically, we answer the following questions:

- RQ1. How effective is FGP in predicting the performance of neural architectures?
- RQ2. How effective is FGP in neural architecture search?
- RQ3. Can the proposed flow surrogate well represent the ground-truth performance of architectures?
- RQ4. Are all the key components of FGP essential?
- RQ5. How long does it take to train an encoder by FGP?

4.1 Experimental setup

Datasets and splits. We leverage three computer vision neural architecture datasets, which are NAS-Bench-101 (NB-101) [51], NAS-Bench-201 (NB-201) [8], and NAS-Bench-301 (NB-301) [39] datasets. In addition, we provide results on other domain datasets (spec., natural language processing and graph representation learning) in Appendix B.6. For NB-101 and NB-201 datasets, we follow the training and test splits provided in [34, 15]. For the NB-301 dataset, since the baseline method ZC-Proxy [56] requires certain numerical properties of architectures, we use a subset of the original NB-301 dataset where these properties are available. We sample 40 architectures from the test set to create a validation set, following the approach in [15]. Further details are provided in Appendix A.1.

Backbone encoders. We use 3 backbone graph-based neural architecture encoders, ResGatedGCN [3], GIN [47], and FlowerFormer [15]. Specifically, ResGatedGCN and GIN are GNN-based encoders, which do not have a specialized structure for capturing the information flow (i.e., non-flow-based encoders). In contrast, FlowerFormer is the SOTA flow-based encoder with a specialized design to capture the flow. Moreover, we present analysis for non-graph-based models in Appendix B.14.

Baseline methods. We use five baselines: an encoder trained solely on supervised learning (N/A) and four pre-training methods for neural architecture encoding, which are the (1) graph-contrastive-learning method (GraphCL [52]) and (2) generative methods based on connections (Arch2vec [48]), operations (GMAE [17]), and zero-cost proxies (ZC-Proxy [56]).

Pre-training, fine-tuning, and evaluation protocol. For each dataset and neural architecture encoder, we first pre-train the encoder using a specific pre-training method across the whole dataset (i.e., both training and test sets). Note that ground-truth performances of any architectures are **not used** during

Table 1: **Performance prediction results.** Mean and standard deviation on each dataset and metric. The best performance is highlighted in blue. N/A denotes an encoder solely trained with supervised learning, without any pre-training method. Gain from N/A denotes the performance improvement of FGP compared to N/A. All values are multiplied by 100 to save space in the table. FGP outperforms the baseline pre-training methods in 23 out of 27 settings.

Encoder	Pre-training method	Kendall's Tau			Precision@1%			Precision@5%		
		NB-101	NB-201	NB-301	NB-101	NB-201	NB-301	NB-101	NB-201	NB-301
ResGatedGCN	N/A	65.0 (7.8)	73.4 (1.5)	54.4 (4.1)	18.2 (7.9)	29.7 (11.8)	18.4 (2.6)	46.2 (12.2)	51.7 (4.2)	40.6 (1.7)
	GraphCL	66.9 (5.0)	73.7 (1.9)	54.6 (3.5)	21.3 (8.8)	31.6 (12.2)	20.4 (1.7)	46.5 (8.3)	52.3 (5.6)	39.7 (2.3)
	Arch2vec	65.8 (5.9)	74.1 (1.2)	57.7 (3.0)	21.8 (13.4)	28.3 (11.3)	25.5 (3.0)	44.9 (9.8)	52.5 (5.1)	42.9 (3.0)
	GMAE	68.1 (4.7)	74.8 (1.2)	57.0 (2.7)	21.8 (11.7)	35.1 (11.2)	22.2 (5.1)	49.9 (6.3)	57.8 (5.4)	41.8 (3.5)
	ZC-Proxy	68.3 (6.7)	79.9 (0.8)	57.9 (2.5)	26.2 (6.2)	44.3 (11.0)	23.0 (3.1)	52.1 (6.3)	61.5 (3.0)	42.6 (3.2)
	FGP	74.8 (4.8)	82.2 (0.7)	58.4 (2.4)	37.5 (13.0)	48.9 (8.1)	23.2 (4.6)	61.7 (3.6)	62.3 (2.5)	43.2 (2.3)
	Gain from N/A	+15.1%	+12.0%	+7.4%	+106.0%	+64.6%	+26.1%	+33.6%	+20.5%	+6.4%
GIN	N/A	62.8 (5.9)	65.7 (1.4)	52.3 (2.7)	26.9 (14.9)	25.0 (11.8)	19.2 (3.5)	48.1 (12.0)	47.6 (5.5)	38.9 (1.3)
	GraphCL	64.9 (3.8)	66.9 (0.9)	52.4 (2.8)	32.3 (18.9)	18.9 (6.2)	20.2 (3.4)	50.2 (6.3)	45.0 (3.0)	39.6 (1.4)
	Arch2vec	63.7 (2.5)	68.0 (0.7)	55.0 (1.7)	30.0 (19.0)	18.6 (7.4)	21.2 (3.1)	49.4 (6.4)	45.6 (2.8)	40.7 (1.7)
	GMAE	66.4 (4.2)	70.6 (2.2)	55.3 (1.6)	31.0 (13.4)	27.8 (14.4)	20.8 (3.1)	52.9 (3.0)	49.7 (5.0)	41.5 (1.9)
	ZC-Proxy	65.2 (8.3)	75.3 (2.3)	53.0 (3.0)	22.7 (12.4)	29.0 (14.3)	18.2 (4.4)	49.9 (13.0)	50.4 (7.2)	38.2 (2.6)
	FGP	67.8 (3.7)	79.2 (1.7)	55.2 (2.2)	33.2 (10.0)	35.6 (12.7)	23.5 (2.2)	55.8 (3.8)	54.1 (3.6)	42.0 (2.2)
	Gain from N/A	+8.0%	+20.6%	+5.5%	+23.4%	+42.4%	+22.4%	+16.0%	+13.7%	+8.0%
FlowerFormer	N/A	74.0 (3.6)	77.3 (1.5)	55.6 (4.7)	35.3 (13.9)	35.6 (14.1)	19.4 (4.5)	56.4 (5.2)	56.2 (4.3)	40.7 (4.2)
	GraphCL	69.4 (4.7)	77.0 (3.2)	55.7 (3.9)	35.3 (7.7)	34.1 (14.9)	21.2 (2.6)	57.5 (2.5)	55.7 (6.3)	41.9 (1.9)
	Arch2vec	76.0 (2.8)	77.8 (1.8)	59.2 (2.9)	38.4 (9.1)	33.3 (15.0)	24.2 (3.0)	59.8 (3.9)	57.5 (6.0)	44.3 (2.6)
	GMAE	74.3 (3.2)	78.7 (1.1)	58.9 (3.2)	33.6 (9.6)	36.4 (16.2)	26.3 (5.2)	56.2 (5.0)	58.7 (5.9)	44.5 (1.8)
	ZC-Proxy	74.6 (3.9)	82.3 (1.2)	58.5 (2.2)	37.8 (5.7)	45.0 (6.6)	25.3 (2.9)	58.1 (3.3)	60.8 (3.8)	41.9 (1.8)
	FGP	76.3 (3.6)	83.6 (1.7)	60.1 (2.9)	40.6 (13.1)	48.3 (3.3)	24.2 (5.4)	58.9 (4.0)	65.0 (4.5)	45.0 (1.4)
	Gain from N/A	+3.1%	+8.2%	+8.1%	+15.0%	+35.7%	+24.7%	+4.4%	+15.7%	+10.6%

this pre-training stage, and despite this fact, it is also possible to use only the training set even for pre-training (see Appendix B.3). Following pre-training, we fine-tune the encoder using only the training set to optimize it for the performance prediction task, using the ground-truth performance of the training set in this phase. After fine-tuning, we evaluate the encoder on the test set using three evaluation metrics: Kendall's tau [38], Precision-@1%, and Precision-@5%, all widely adopted for this evaluation process [34, 15]. We employ three distinct dataset splits and three model initializations, resulting in a total of nine experimental configurations. Further details are in Appendix A.

4.2 RQ1: Performance prediction experiments

We assess FGP in the performance prediction task compared to the baseline pre-training methods.

Setup. In practice, only a small subset of the full neural architecture search space has known ground-truth performance values due to the high cost of training and evaluating architectures. To simulate this circumstance, we use 1% of the training set for fine-tuning neural architecture encoders—a common setting in neural architecture encoding research [34, 15]. In addition, analyses under varying (1) pre-training dataset sizes and (2) training dataset sizes are in Appendix B.4 and B.2, respectively.

Results. As shown in Table 1, FGP outperforms all baselines in 23 out of 27 settings, demonstrating the effectiveness of learning information flow through pre-training in performance prediction. Note that the performance improvement achieved by FGP in the performance prediction task is not restricted to a specific dataset or encoder. While the improvement is greater in the non-flow-based encoders (i.e., ResGatedGCN and GIN), it can also improve the performance of the flow-based encoder (i.e., FlowerFormer); we further explore the potential reasons behind the improvement of FlowerFormer in Appendix B.16.

4.3 RQ2: Neural architecture search experiments

Performance prediction is often integrated into neural architecture search (NAS) to automatically identify neural architectures suitable to the target task and dataset. Accordingly, we evaluate the efficacy of FGP in NAS, assessing its performance against those of existing pre-training methods.

Setup. As in [15], we use NPENAS [42] as our backbone NAS algorithm, which uses a performance predictor to evaluate neural architectures in the search process. Note that an accurate performance predictor enhances search efficacy, leading to the discovery of higher-performing architectures.

To assess each pre-training method, we (1) pre-train a performance predictor on neural architectures within the designated search space and (2) initialize NPE-NAS's predictor with this pre-trained model at the beginning of each search. We run 10 trials with different initializations. NAS-Bench-201 and ResGatedGCN serve as the search space and performance predictor, respectively. Further details are in Appendix A.5.

Results. As shown in Figure 5, the NAS method, equipped with a performance predictor pre-trained using FGP, identifies the more effective neural architecture than all competitors, each employing a distinct pre-training method. Notably, FGP consistently performs best at every NAS step.

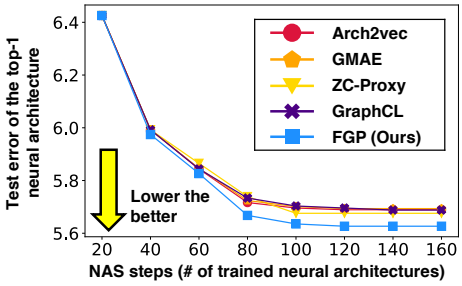


Figure 5: **NAS results.** The mean test error of the top-1 architecture across 10 trials is reported. The backbone NAS algorithm NPE-NAS [42] adopts pre-trained performance predictors, with each line representing a distinct pre-training approach.

4.4 RQ3: Flow surrogate analysis

We provide a qualitative analysis of the relation between the flow surrogate of neural architecture (Section 3.2) and the ground-truth performance of the corresponding architecture. Specifically, we aim to verify whether our flow surrogate well represents the architecture's ground-truth performance, separating architectures with differing performance..

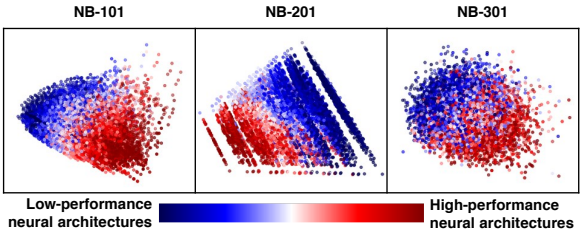


Figure 6: **Pre-training objective visualization.** PCA visualization of flow surrogates (i.e., pre-training objective), where colors represent the performance of the corresponding architecture.

Setup. We visualize the distribution of the flow surrogate by using PCA. We color each data point according to the corresponding neural architecture's ground-truth performance.

Results. As shown in Figure 6, the proposed flow surrogate effectively represents the performance of neural architectures, distinguishing high-performing architectures from low-performing ones in separate regions. This visualization indicates that our pre-training objective, flow surrogate, can well guide a neural architecture encoder in identifying architectures likely to achieve high performance.

4.5 RQ4: Ablation study

We demonstrate the necessity of the key components of FGP in achieving high performance.

Setup. We use four variants of FGP:

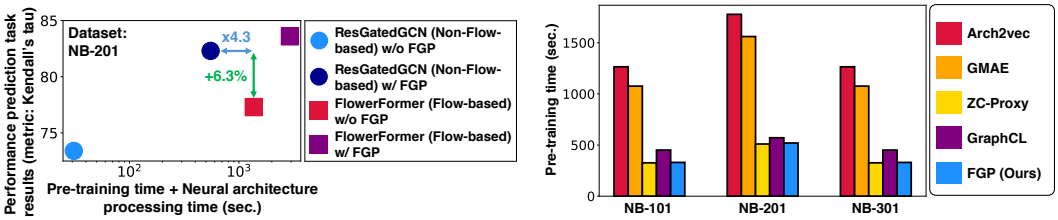
- **V1.** This does not perform the flow surrogate reconstruction (w/o $\mathcal{L}_{gen}$).
- **V2.** This does not use the auxiliary training objective (w/o $\mathcal{L}_{aux}$).
- **V3.** This skips mimicking the forward pass (Section 3.2.2) in obtaining the flow surrogate (w/o Forward).
- **V4.** This skips mimicking the backpropagation. (Section 3.2.3) in obtaining the flow surrogate (w/o Backward).

Figure 7: **Ablation study.** Performance of the variants of FGP. All values are multiplied by 100 to save space. The best performance is highlighted in blue.

Variants of FGP	Kendall's Tau			Precision@1%		
	NB-101	NB-201	NB-301	NB-101	NB-201	NB-301
w/o $\mathcal{L}_{gen}$	68.3 (6.7)	79.9 (0.8)	57.9 (2.5)	26.2 (6.2)	44.3 (11.0)	23.0 (3.1)
w/o $\mathcal{L}_{aux}$	71.5 (5.6)	74.5 (0.7)	58.0 (2.7)	35.6 (8.7)	36.5 (2.2)	23.6 (3.2)
w/o Forward	73.5 (4.3)	81.4 (0.7)	57.1 (3.6)	36.3 (9.2)	44.6 (9.0)	23.4 (4.1)
w/o Backward	72.4 (4.8)	81.4 (0.6)	57.5 (2.5)	31.0 (9.8)	43.3 (9.1)	24.2 (2.7)
FGP	74.8 (4.8)	82.2 (0.7)	58.4 (2.4)	37.5 (13.0)	48.9 (8.1)	23.2 (4.6)

We use ResGatedGCN as our backbone encoder and evaluate under the performance prediction task. Other settings are the same as in Section 4.2.

Results. As shown in Table 7, FGP achieves superior performance over its four variants in five out of six settings, underscoring the necessity of its core components for achieving high performance.



(a) Encoders' runtime (pre-training time included) and performance-prediction results ($\times 100$).

(b) Total pre-training time for each method using the ResGatedGCN backbone.

Figure 8: **Speed analysis.** (a) Pre-training the non-flow encoder (ResGatedGCN) with FGP improves both performance and speed over the flow-based encoder (FlowerFormer) without FGP. (b) FGP is the second-fastest among the pre-training methods for neural architecture encoding.

4.6 RQ5: Speed analysis

We provide a speed analysis of FGP.

Setup. We analyze (1) the total runtime and performance prediction results with (w/) and without (w/o) FGP pre-training for the flow-based encoder (FlowerFormer) and the non-flow-based encoder (ResGatedGCN) and (2) the pre-training runtime of each pre-training method, equipped with ResGatedGCN. We set the batch size and pre-training epochs to 256 and 200, respectively. Also, a comparison of all methods under equal runtime is in Appendix B.5.

Results. First, as shown in Figure 8 (a), pre-training the non-flow-based encoder (ResGatedGCN) with FGP shows both performance and speed gain over the flow-based encoder (FlowerFormer) without FGP in performance prediction. That is, despite this performance superiority, the total runtime for the pre-training and architecture processing of ResGatedGCN remains significantly shorter than the processing time alone of FlowerFormer. Second, as shown in Figure 8 (b), FGP is the second-fastest pre-training method, comparable to the fastest one, ZC-Proxy. Thus, despite the large performance gain compared to the baselines, FGP does not impose a significantly greater computational burden.

5 Conclusion

In this work, we propose a novel generative pre-training method for neural architecture encoding, called FGP, which enables a neural architecture encoder to learn the information flow within a neural architecture, even without a specialized model structure. To this end, FGP trains a neural architecture encoder to reconstruct the flow surrogate, our proposed pre-training objective, based on information flow. Our extensive experiments demonstrate the efficacy of FGP in performance prediction and neural architecture search. Code and datasets are in https://github.com/kswoo97/FGPAnom.

Acknowledgements

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2022-II220871, Development of AI Autonomy and Knowledge Enhancement for AI Agent Collaboration, 60%) (No. RS-2022-II220157, Robust, Fair, Extensible Data-Centric Continual Learning, 20%) (No. RS-2024-00457882, AI Research Hub Project, 10%) (No. RS-2019-II190075, Artificial Intelligence Graduate School Program (KAIST), 10%).

References

[1] M. S. Abdelfattah, A. Mehrotra, Ł. Dudziak, and N. D. Lane. Zero-cost proxies for lightweight nas. In *ICLR*, 2021.

[2] U. A. Bakshi and L. A. V. Bakshi. *Electrical circuit analysis*. Technical Publications, 2020.

[3] X. Bresson and T. Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[4] Y. Chen, Y. Guo, Q. Chen, M. Li, W. Zeng, Y. Wang, and M. Tan. Contrastive neural architecture search with neural architecture comparators. In *CVPR*, 2021.

[5] K. T. Chitty-Venkata, M. Emani, V. Vishwanath, and A. K. Somani. Neural architecture search for transformers: A survey. *IEEE Access*, 10:108374–108412, 2022.

[6] J. Delvin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[8] X. Dong and Y. Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *ICLR*, 2020.

[9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[10] L. Dudziak, T. Chau, M. Abdelfattah, R. Lee, H. Kim, and N. Lane. Brp-nas: Prediction-based nas using gcns. In *NeurIPS*, 2020.

[11] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[12] K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick. Masked autoencoders are scalable vision learners. In *CVPR*, 2022.

[13] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec. Strategies for pre-training graph neural networks. In *ICLR*, 2019.

[14] Y.-C. Huang, W.-H. Li, C.-H. Tsou, J.-C. Chen, and C.-S. Chen. Up-nas: Unified proxy for neural architecture search. In *CVPR*, 2024.

[15] D. Hwang, H. Kim, S. Kim, and K. Shin. Flowerformer: Empowering neural architecture encoding using a flow-aware graph transformer. In *CVPR*, 2024.

[16] G. Jawahar, M. Abdul-Mageed, L. V. Lakshmanan, and D. Ding. Llm performance predictors are good initializers for architecture search. In *ACL*, 2024.

[17] K. Jing, J. Xu, and P. Li. Graph masked autoencoder enhanced predictor for neural architecture search. In *IJCAI*, 2022.

[18] S. Kim, S. Kang, F. Bu, S. Y. Lee, J. Yoo, and K. Shin. Hypeboy: Generative self-supervised representation learning on hypergraphs. In *ICLR*, 2024.

[19] S. Kim, S. Y. Lee, F. Bu, S. Kang, K. Kim, J. Yoo, and K. Shin. Rethinking reconstruction-based graph-level anomaly detection: limitations and a simple remedy. In *NeurIPS*, 2024.

[20] S. Kim, S. Y. Lee, Y. Gao, A. Antelmi, M. Polato, and K. Shin. A survey on hypergraph neural networks: an in-depth and step-by-step guide. In *KDD*, 2024.

[21] S. Kim, S. Y. Lee, J. Yoo, and K. Shin. 'hello, world!': Making gnns talk with llms. In *EMNLP*, 2025.

[22] T. N. Kipf and M. Welling. Variational graph auto-encoders. In *NeurIPS Bayesian Deep Learning Workshop*, 2016.

[23] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[24] N. Klyuchnikov, I. Trofimov, E. Artemova, M. Salnikov, M. Fedorov, A. Filippov, and E. Burnaev. Nas-bench-nlp: neural architecture search benchmark for natural language processing. *IEEE Access*, 10:45736–45747, 2022.

[25] A. Krishnakumar, C. White, A. Zela, R. Tu, M. Safari, and F. Hutter. Nas-bench-suite-zero: Accelerating research on zero cost proxies. In *NeurIPS*, 2022.

[26] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[27] J. Lee and B. Ham. Az-nas: Assembling zero-cost proxies for network architecture search. In *CVPR*, 2024.

[28] W. Lin, X. Peng, Z. Yu, and T. Jin. Hypergraph neural architecture search. In *AAAI*, 2024.

[29] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. In *ICLR*, 2019.

[30] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.

[31] S. Lu, J. Li, J. Tan, S. Yang, and J. Liu. Tnasp: A transformer-based nas predictor with a self-evolution framework. In *NeurIPS*, 2021.

[32] X. Ning, Y. Zheng, T. Zhao, Y. Wang, and H. Yang. A generic graph-based neural architecture encoding scheme for predictor-based nas. In *ECCV*, 2020.

[33] X. Ning, C. Tang, W. Li, Z. Zhou, S. Liang, H. Yang, and Y. Wang. Evaluating efficient performance estimators of neural architectures. In *NeurIPS*, 2021.

[34] X. Ning, Z. Zhou, J. Zhao, T. Zhao, Y. Deng, C. Tang, S. Liang, H. Yang, and Y. Wang. Ta-gates: An encoding scheme for neural network architectures. In *NeurIPS*, 2022.

[35] Y. Ou, Y. Feng, and Y. Sun. Towards accurate and robust architectures via neural architecture search. In *CVPR*, 2024.

[36] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[37] Y. Qin, Z. Zhang, X. Wang, Z. Zhang, and W. Zhu. Nas-bench-graph: Benchmarking graph neural architecture search. In *NeurIPS*, 2022.

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

[39] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter. Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. In *ICLR*, 2021.

[40] V. Thost and J. Chen. Directed acyclic graph neural networks. In *ICLR*, 2021.

[41] Z. Tong, Y. Liang, C. Sun, X. Li, D. Rosenblum, and A. Lim. Digraph inception convolutional networks. In *NeurIPS*, 2020.

[42] C. Wei, C. Niu, Y. Tang, Y. Wang, H. Hu, and J. Liang. Npenas: Neural predictor guided evolution for neural architecture search. *IEEE Transactions on Neural Networks and Learning Systems*, 34(11):8441–8455, 2022.

[43] W. Wen, H. Liu, Y. Chen, H. Li, G. Bender, and P.-J. Kindermans. Neural predictor for neural architecture search. In *ECCV*, 2020.

[44] C. White, W. Neiswanger, S. Nolen, and Y. Savani. A study on encodings for neural architecture search. In *NeurIPS*, 2020.

[45] C. White, A. Zela, R. Ru, Y. Liu, and F. Hutter. How powerful are performance predictors in neural architecture search? In *NeurIPS*, 2021.

[46] F. Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*, pages 196–202. Springer, 1992.

[47] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

[48] S. Yan, Y. Zheng, W. Ao, X. Zeng, and M. Zhang. Does unsupervised architecture representation learning help neural architecture search? In *NeurIPS*, 2020.

[49] Y. Yi, H. Zhang, W. Hu, N. Wang, and X. Wang. Nar-former: Neural architecture representation learning towards holistic attributes prediction. In *CVPR*, 2023.

[50] Y. Yi, H. Zhang, R. Xiao, N. Wang, and X. Wang. Nar-former v2: rethinking transformer for universal neural network representation learning. In *NeurIPS*, 2023.

[51] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *ICML*, 2019.

[52] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, and Y. Shen. Graph contrastive learning with augmentations. In *NeurIPS*, 2020.

[53] Y. Yuan, C. Shao, J. Ding, D. Jin, and Y. Li. Spatio-temporal few-shot learning via diffusive neural network generation. In *ICLR*, 2024.

[54] X. Zhai, J. Puigcerver, A. Kolesnikov, P. Ruyssen, C. Riquelme, M. Lucic, J. Djolonga, A. S. Pinto, M. Neumann, A. Dosovitskiy, et al. A large-scale study of representation learning with the visual task adaptation benchmark. *arXiv preprint arXiv:1910.04867*, 2019.

[55] R. Zhang, X. Hu, B. Li, S. Huang, H. Deng, Y. Qiao, P. Gao, and H. Li. Prompt, generate, then cache: Cascade of foundation models makes strong few-shot learners. In *CVPR*, 2023.

[56] J. Zhao, X. Ning, E. Liu, B. Ru, Z. Zhou, T. Zhao, C. Chen, J. Zhang, Q. Liao, and Y. Wang. Dynamic ensemble of low-fidelity experts: Mitigating nas "cold-start". In *AAAI*, 2023.

[57] L. Zhu, B. Liao, Q. Zhang, X. Wang, W. Liu, and X. Wang. Vision mamba: Efficient visual representation learning with bidirectional state space model. In *ICML*, 2024.

A Experimental details

Table 2: **Dataset statistics.** Total counts of training and test data, along with the average time (in seconds) required to obtain the proposed flow surrogate (defined in Section 3.2).

Dataset	# of train	# of test	Time for surrogate (secs.)
Nas-Bench-101 [51]	7,290	7,290	0.002
Nas-Bench-201 [8]	7,813	7,812	0.002
Nas-Bench-301 [39]	5,611	5,610	0.005

A.1 Dataset details

In this section, we elaborate on the leveraged three convolution neural architecture datasets. Dataset statistics are provided in Table 2. Specifically, we represent each architecture based on an operation-on-node type graph [34], where each operation corresponds to a node, and each edge represents a connection between two operations.

- **NAS-Bench-101** [51]. This dataset includes 423K convolutional neural architectures, where each data point represents a specific neural architecture along with its test accuracy on the CIFAR-10 image classification task [26]. We leverage a subset of the entire search space provided by Hwang et al. [15], Ning et al. [34].
- **NAS-Bench-201** [8]. This dataset includes 15K convolutional neural architectures, where each data point represents a specific neural architecture along with its test accuracy on the CIFAR-10 image classification task [26].
- **NAS-Bench-301** [39]. This dataset includes 10^{16} convolutional neural architectures, where each data point represents a specific neural architecture along with its surrogate performance (i.e., an estimated performance produced by a performance predictor) on the CIFAR-10 image classification task [26]. Each architecture is represented by two graphs: a normal cell and a reduction cell. Following Siems et al. [39], we connect the output node of the normal cell graph to the input nodes of the reduction cell, creating a single unified graph that represents the architecture. We leverage a subset of the entire search space provided by Abdelfattah et al. [1], where zero-cost proxies are publicly available.

For non-flow-based neural architecture encoders (i.e., ResGatedGCN [3] and GIN [47]), which lack specialized message passing for directed acyclic graphs, we transform each neural architecture graph into an undirected graph by adding a reverse edge for every existing graph.

A.2 Machines and implementation

We conducted our experiments on a machine with NVIDIA RTX 8000 D6 GPUs (48GB memory) and two Intel Xeon Silver 4214R processors. FGP is primarily implemented using the Pytorch (v1.12.1) and Pytorch Geometric (v2.2.0) libraries. We used AdamW [30] as the learning optimizer.

A.3 Fine-tuning protocol

In this section, we detail our fine-tuning protocol. For a given neural architecture encoder f_θ, which is randomly initialized or pre-trained with a particular pre-training method, we fine-tune the neural architecture encoder to perform the performance prediction task.

Regression. Recall that a neural architecture encoder outputs a $d-$dimensional vector $\mathbf{z}^{(i)} \in \mathbb{R}^d$, representing a given neural architecture $\mathcal{G}^{(i)}$. To estimate the performance of the corresponding neural architecture, denoted as $y^{(i)} \in \mathbb{R}$, we employ a feed-forward-network-based regressor $r_\xi : \mathbb{R}^d \mapsto \mathbb{R}$. The performance estimate $\hat{y}^{(i)}$ is obtained by projecting $\mathbf{z}^{(i)}$ using this regressor (i.e., $\hat{y}^{(i)} = r_\xi(\mathbf{z}^{(i)})$).

Learning objective. Similar to our use of a zero-cost proxy (refer to Appendix D.2), to train a model to predict which architectures are 'more likely' to perform well, we leverage a margin-ranking

loss. For two neural architectures $\mathcal{G}^{(i)}$ and $\mathcal{G}^{(j)}$ within the same batch, we compute the following prediction loss:

$$\mathcal{L}_{fine} = \sum_{(i,j):y^{(i)}>y^{(j)}} \max\left(0, m' - (\hat{y}^{(i)} - \hat{y}^{(j)})\right), \tag{1}$$

where m' is a margin hyperparameter. The parameters θ and ξ are trained to minimize $\mathcal{L}_{fine}$ (Eq. 1) via gradient descent. This updating process continues until it satisfies the validation stopping criterion or reaches the maximum number of fine-tuning epochs.

A.4 Hyperparameters

Neural architecture encoders. For ResGatedGCN [3] and GIN [47], whose best hyperparameter configurations are not publicly provided, we tuned their hidden dimensions and maximum fine-tuning epochs within $\{64, 128, 256\}$ and $\{300, 400, 500\}$, according to the validation set performance on the performance prediction task. We fixed their number of layers, fine-tuning learning rate, and fine-tuning weight decay, as $3, 10^{-3}$, and 10^{-6}, respectively. After finding the hyperparameter configuration that gives the best validation performance, the chosen configuration is used across all the methods (i.e., the same encoder structure and pre-training scheme are leveraged across different pre-training methods). For FlowerFormer [15], we follow the hyperparameter configurations provided in their official implementation https://github.com/y0ngjaenius/CVPR2024_FLOWERFormer.

Pre-training methods. For every pre-training method, including the baseline methods and our proposed method FGP, we tuned the following hyperparameters:

- **Learning rate** within $\{10^{-3}, 5 \times 10^{-4}, 10^{-4}\}$,
- **Projection head dimension** within $\{32, 64, 128, 256\}$,
- **Projection head number of layers** within $\{1, 2, 3\}$,
- **Weight decay** within $\{10^{-6}, 0.0\}$.

Additionally, for FGP, we tuned the learning objective coefficients (λ_1, λ_2) within $\{(\frac{1}{2}, \frac{1}{2}), (\frac{1}{3}, \frac{2}{3}), (\frac{2}{3}, \frac{1}{3})\}$.

A.5 Details regarding neural architecture search

In this section, we provide details regarding our neural architecture search experiment (Section 4.3). Note that the process below is based on NPENAS [42].

We first pre-train a neural architecture encoder using a designated pre-training strategy, such as FGP. At the start of each round, the encoder is initialized with the pre-trained weights. The search begins with 20 neural architectures, each with known ground-truth performance. Using these, we fine-tune the encoder to predict performance. For each of the 20 architectures, we generate 5 valid candidates via mutation. Among the mutated architectures, we select the top 20 based on predicted performance and evaluate them. The selected architectures and their performances are added to the budget. With the updated pool, now containing 40 evaluated architectures, we repeat the process until the budget reaches a predefined size.

B Additional experimental results

B.1 Analysis of encoding time of diverse architecture encoders

In this section, we provide further details regarding the encoding time analysis, which is provided in Figure 1. Specifically, we provide detailed mean encoding time per architecture for the flow-aware neural architecture and the simple GNN-based neural architecture.

Setup. We compare the encoding time of a flow-aware encoder (i.e., FlowerFormer) with non-flow-aware encoders (i.e., ResGatedGCN and GIN). Specifically, we measure the time required to generate representations for all neural architectures—both training and test—and report the average per architecture.

Table 3: **Performance prediction results on various training set ratios.** Mean and standard deviation on each dataset and metric. The best performance is highlighted in blue. All values are multiplied by 100. N/A denotes an encoder solely trained with supervised learning, without any pre-training. Gain from N/A denotes the performance improvement of FGP compared to N/A. FGP outperforms the baselines in 14/18 cases.

Training set ratio	Pre-training method	Kendall's Tau			Precision@1%			Precision@5%		
		NB-101	NB-201	NB-301	NB-101	NB-201	NB-301	NB-101	NB-201	NB-301
5%	N/A	75.8 (3.4)	84.6 (0.7)	67.2 (1.1)	42.9 (11.3)	54.3 (3.7)	31.9 (2.9)	64.5 (3.6)	69.4 (2.8)	51.9 (3.4)
	GraphCL	76.0 (5.0)	85.4 (0.7)	66.4 (1.1)	42.8 (8.6)	57.0 (3.0)	29.9 (3.8)	63.4 (6.3)	69.8 (2.8)	50.6 (1.9)
	Arch2vec	76.9 (3.7)	85.3 (0.7)	68.4 (1.3)	42.0 (12.8)	56.7 (5.0)	34.5 (5.4)	65.1 (3.1)	70.2 (3.1)	53.2 (4.3)
	GMAE	76.2 (4.7)	87.0 (0.5)	67.8 (1.8)	41.7 (8.1)	59.0 (5.4)	35.4 (4.9)	62.5 (5.6)	73.6 (1.2)	53.1 (3.5)
	ZC-Proxy	78.0 (4.1)	87.7 (0.4)	66.6 (1.2)	42.7 (7.5)	56.4 (4.6)	33.9 (4.4)	67.3 (3.0)	72.2 (3.2)	51.7 (4.7)
	FGP	79.4 (3.3)	88.0 (0.4)	68.6 (2.4)	45.8 (11.0)	59.7 (5.4)	34.8 (4.6)	66.8 (3.2)	73.6 (2.2)	53.3 (5.0)
	Gain from N/A	+4.7%	+4.0%	+2.1%	+6.8%	+9.9%	+9.1%	+3.6%	+6.1%	+2.7%
10%	N/A	77.3 (5.1)	88.3 (0.3)	69.9 (1.2)	43.2 (8.9)	57.9 (2.2)	35.4 (5.2)	65.8 (4.0)	74.0 (1.0)	56.6 (2.0)
	GraphCL	77.3 (5.9)	88.9 (0.3)	69.1 (2.3)	42.7 (9.0)	58.6 (4.1)	35.6 (3.7)	66.3 (3.8)	73.3 (1.3)	55.3 (2.8)
	Arch2vec	78.3 (5.1)	88.8 (0.4)	71.2 (1.8)	43.2 (11.6)	57.0 (3.3)	37.2 (5.6)	66.5 (4.3)	73.9 (1.1)	56.7 (4.0)
	GMAE	78.1 (5.9)	89.9 (0.4)	70.8 (1.0)	45.5 (10.6)	61.3 (5.1)	38.8 (4.8)	67.6 (3.6)	75.4 (0.7)	58.6 (1.5)
	ZC-Proxy	79.0 (4.8)	89.7 (0.2)	70.2 (0.9)	44.0 (9.0)	58.2 (1.3)	37.4 (4.4)	67.2 (3.0)	74.5 (1.4)	56.8 (1.4)
	FGP	79.9 (4.3)	89.9 (0.3)	71.5 (0.6)	45.8 (10.2)	58.4 (1.0)	38.9 (5..0)	67.8 (3.4)	76.0 (1.0)	56.9 (2.4)
	Gain from N/A	+3.4%	+1.8%	+2.3%	+6.0%	+1.0%	+9.8%	+3.0%	+2.7%	+0.1%

Results. As shown in Table 4, FlowerFormer, a flow-aware encoder, takes at most 94x more encoding time compared to GIN, which is a simple GNN encoder that does not have a specialized architecture to capture the information flow. This result supports our claim that flow-aware encoders are significantly slower than simple GNN-based encoders.

Table 4: Detailed mean encoding time per architecture of each neural architecture encoder

	NB-101	NB-201
ResGatedGCN	0.0021	0.0022
GIN	0.0013	0.0014
FlowerFormer	0.1242	0.0849

B.2 Analysis under varying training set size

In this section, we present additional experimental results across different training set ratios. In the main paper, we focus on a scenario where only 1% of the training set is used for fine-tuning, simulating a label-scarce scenario (i.e., a limited number of architecture-performance pairs available). Recall that this is because obtaining architecture-performance pairs involves significant computational costs, making it crucial to develop a pre-training method that enhances neural architecture encoders in label-scarce scenarios. Furthermore, we evaluate each method in scenarios where additional architecture-performance pairs are available, corresponding to a larger training set ratio.

Setup. We explore two additional settings for fine-tuning: using (1) 5% and (2) 10% of the training set, respectively. We use ResGatedGCN [3] as the backbone neural architecture encoder. Other experimental settings are the same as those used in our neural architecture performance prediction experiments (Section 4.2).

Results. As shown in Table 3, FGP outperforms all baseline methods in 14 out of 18 settings. This result demonstrates that the superiority of FGP over existing pre-training methods is not restricted to a specific training set ratio setting.

B.3 Analysis under using only the training set, including for pre-training

In this section, we present additional experiment results when we only leverage the training dataset to perform pre-training. In the main paper, we leverage the test dataset with the training during the pre-training, since architectures within the search space can be obtained with negligible cost. In the new setting, we aim to determine whether an encoder pre-trained with FGP can generalize to unseen neural architectures.

Setup. We do not use the test dataset for pre-training, and therefore, each test architecture becomes an unseen architecture. We use ResGatedGCN [3] as the backbone neural architecture encoder. Other experimental settings are the same as those used in our neural architecture performance prediction experiments (Section 4.2).

Results. As shown in Table 5, FGP outperforms all baseline methods in all the settings. This result demonstrates that the encoder trained with FGP generalizes well to unseen neural architectures.

Table 5: **Performance prediction results when using only the training dataset even for pre-training.** Mean and standard deviation on each dataset and metric. The best performance is highlighted in blue. All values are multiplied by 100. N/A denotes an encoder solely trained with supervised learning, without any pre-training. Gain from N/A denotes the performance improvement of FGP compared to N/A. FGP outperforms the baseline methods in all the settings.

Pre-training method	Kendall's Tau		Precision@1%		Precision@5%	
	NB-101	NB-201	NB-101	NB-201	NB-101	NB-201
N/A	65.0 (7.8)	73.4 (0.7)	18.2 (7.9)	29.7 (3.7)	46.2 (12.2)	51.7 (4.2)
GraphCL	65.2 (6.4)	74.3 (1.6)	20.1 (12.6)	36.4 (9.7)	44.4 (11.3)	54.5 (1.7)
Arch2vec	63.2 (7.8)	73.5 (1.9)	18.5 (9.3)	31.7 (11.9)	42.4 (12.4)	53.3 (5.7)
GMAE	67.3 (5.2)	75.2 (1.4)	25.5 (10.1)	34.8 (9.9)	51.0 (7.0)	56.8 (4.9)
ZC-Proxy	69.8 (5.1)	79.8 (1.3)	29.8 (11.1)	39.2 (12.8)	56.4 (3.5)	58.9 (8.2)
FGP	73.5 (5.5)	81.4 (7.7)	37.0 (10.9)	45.1 (7.9)	60.1 (4.0)	61.5 (2.8)
Gain from N/A	+13.1%	+10.8%	+103.3%	+51.9%	+30.1%	+19.0%

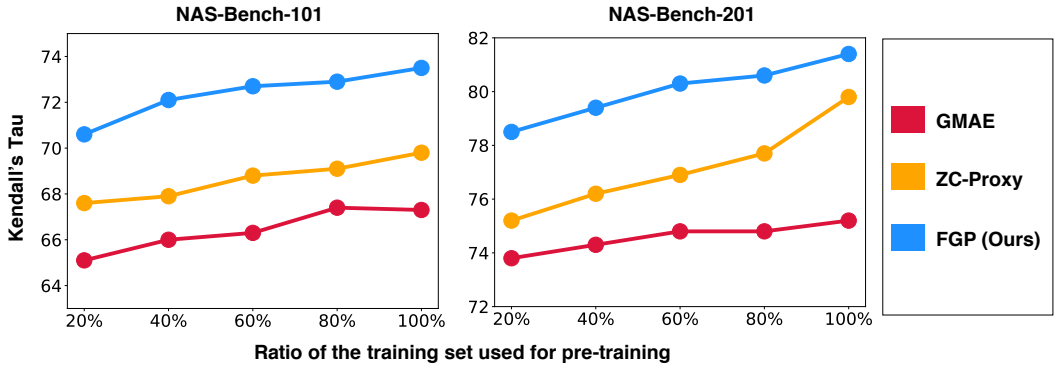


Figure 9: **Performance under varying size of pre-training dataset.** Performance prediction results when 20/40/60/80/100% of the training set is used for pre-training of each method. FGP consistently outperforms the two strongest baseline methods (GMAE and ZC-Proxy) across all the settings.

B.4 Analysis under varying pre-training set size

In this section, we present additional experimental results with varying pre-training dataset sizes, demonstrating that the effectiveness of FGP is not restricted to a specific data size.

Setup. We randomly sample 20%/40%/60%/80%/100% of the training set and use the sampled neural architectures for pre-training of a neural architecture, based on each pre-training strategy. We use ResGatedGCN as the backbone neural architecture encoder, and the two strongest pre-training method (i.e., ZC-Proxy and GMAE) as baseline methods. Other setups are the same as those used in our neural architecture performance prediction experiments (Section 4.2).

Results. As shown in Figure 9, the superiority of FGP holds under varying size of pre-training dataset size, demonstrating that the effectiveness of FGP is not limited to a particular dataset size.

B.5 Analysis under a fixed pre-training time

Recall that in Section 4.6, we provide analysis regarding the runtime of FGP and other baseline methods. In this section, we provide a performance comparison of pre-training methods under the fixed pre-trainig time.

Setup. We train a neural architecture for 100/200/300/400/500 seconds, by using each pre-training method. When the corresponding time limit is reached, we execute the pre-training process and fine-tune and evaluate the pre-trained encoder for the performance prediction task. We use ResGatedGCN as the backbone neural architecture encoder, and the two strongest pre-training method (i.e., ZC-Proxy and GMAE) as baseline methods. Other setups are the same as those used in our neural architecture performance prediction experiments (Section 4.2).

Results. As shown in Figure 10, the superiority of FGP holds under varying pre-training time split, demonstrating that the effectiveness of FGP is not limited to a particular dataset size.

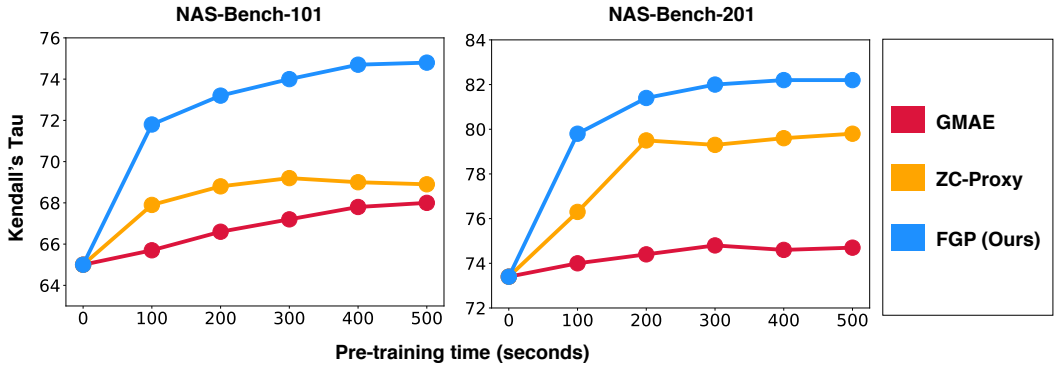


Figure 10: **Performance under the fixed pre-training time.** Performance prediction results under the fixed pre-training time of each pre-training method. FGP consistently outperforms the two strongest baseline methods (GMAE and ZC-Proxy) across all the settings.

Table 6: **Performance prediction results under various domains.** Mean and standard deviation on each dataset and metric. The best performance is highlighted in blue. All values are multiplied by 100. Since zero-cost proxy values for the NB-Graph dataset are not publicly available, we could not run the ZC-Proxy baseline on the NB-Graph dataset, which is marked by '-'. FGP outperforms the baselines on natural language processing datasets and graph representation learning datasets.

Pre-training method	Kendall's Tau		Precision@1%		Precision@5%	
	NB-NLP	NB-Graph	NB-NLP	NB-Graph	NB-NLP	NB-Graph
N/A	28.5 (6.1)	37.1 (6.3)	2.9 (1.4)	4.3 (2.3)	14.8 (6.9)	17.1 (4.1)
GraphCL	28.7 (5.3)	38.3 (13.3)	3.1 (1.3)	3.8 (1.5)	14.9 (6.1)	16.1 (4.3)
Arch2vec	28.4 (5.7)	38.8 (9.6)	3.0 (1.2)	3.2 (2.1)	15.1 (5.2)	15.9 (5.1)
GMAE	28.9 (6.6)	37.7 (8.6)	2.2 (1.0)	2.8 (2.1)	15.7 (5.3)	14.5 (4.7)
ZC-Proxy	28.7 (5.6)	-	2.9 (2.3)	-	15.5 (3.8)	-
FGP	29.7 (4.8)	41.1 (8.6)	4.3 (2.5)	5.3 (1.1)	16.2 (7.1)	18.3 (3.5)

B.6 Analysis under various domains

In this section, we explore the effectiveness of FGP in domains beyond computer vision, which are natural language processing and graph representation learning.

Setup. We use two additional benchmark datasets: (1) NAS-Bench-NLP [24], a dataset that consists of natural language processing models, and (2) NAS-Bench-Graph [37], a dataset that consists of graph representation learning models.

Results. As shown in Table 6, FGP outperforms the baselines in 5 out of 6 settings, demonstrate that the effectiveness of FGP does not limit to a computer vision domain.

B.7 Analysis under another task

Recall that NAS-Bench-101, NAS-Bench-201, and NAS-Bench-301 are benchmark datasets containing image classification performances of various architectures. In this section, we assess each method's effectiveness in predicting architecture performance on a different task—dataset transferability—using a new dataset.

Setup. Since no benchmark directly covers the transferability of architectures, we built a small neural architecture benchmark dataset. To this end, we build two datasets: (1) TinyImageNet (TIN) [7] $\rightarrow$ DTD [54] and (2) TinyImageNet (TIN) [7] $\rightarrow$ OxfordPet [54]. To this end, we randomly sampled 100 architectures from NAS-Bench-101, pre-trained each on TinyImageNet, and fine-tuned each on DTD or OxfordPet. We used each architecture's test accuracy on the target dataset as the architecture's transferability score. Architectures were split 50/50 into train/test sets. ResGatedGCN is used as a backbone neural architecture encoder. An architecture encoder was first pre-trained with a pre-training method (without transferability scores), then fine-tuned on the training set with transferability scores, and finally evaluated on the test set.

Table 7: **Performance prediction results under a new downstream task.** Mean and standard deviation on each dataset and metric. N/A denotes a performance of an encoder without pre-training. All values are multiplied by 100. The best performance is highlighted in blue. The superiority of FGP over baseline methods holds valid under a new downstream task, dataset transfer.

	Kendall's Tau		Precision@10%	
	TIN → DTD	TIN → OxfordPet	TIN → DTD	TIN → OxfordPet
N/A	62.7 (4.6)	60.3 (4.4)	50.0 (31.1)	63.9 (12.4)
GMAE	63.4 (3.8)	61.0 (3.7)	55.5 (25.8)	63.9 (12.4)
ZC-Proxy	64.1 (4.9)	60.8 (8.1)	47.0 (29.7)	61.1 (11.7)
FGP	65.3 (4.3)	64.1 (3.1)	63.9 (12.4)	66.7 (11.7)

Table 8: **Performance prediction results under a new neural architecture encoder.** Mean and standard deviation on each dataset and metric. N/A denotes a performance of an encoder without pre-training. All values are multiplied by 100. The best performance is highlighted in blue. The superiority of FGP over baseline methods holds valid under a new encoder, DiGCN.

	Kendall's Tau		Precision@1%		Precision@5%	
	NB-101	NB-201	NB-101	NB-201	NB-101	NB-201
N/A	65.2 (4.4)	65.7 (1.5)	25.6 (5.5)	14.1 (4.8)	49.0 (1.1)	43.9 (2.6)
GMAE	65.1 (3.9)	66.0 (1.4)	26.3 (5.4)	16.2 (5.2)	49.5 (2.7)	44.2 (2.1)
ZC-Proxy	66.9 (4.3)	72.1 (1.3)	26.0 (3.9)	18.8 (4.7)	50.5 (5.7)	47.1 (1.9)
FGP	73.3 (3.7)	77.9 (2.0)	37.3 (8.6)	23.8 (8.1)	57.5 (5.6)	50.9 (5.8)

Results. As shown in Table 7, FGP outperforms all the baseline methods in all settings, demonstrating that its superiority extends beyond predicting image classification performance to also include architecture transferability prediction.

B.8 Analysis under a new neural architecture encoder

In this section, we explore the effectiveness of FGP under a new directed-graph-based neural architecture encoder.

Setup. We use DiGCN [41], a directed-graph-specialized neural architecture, as the backbone neural architecture encoder, and the two strongest pre-training methods (i.e., ZC-Proxy and GMAE) as baseline methods. Other setups are the same as those used in our neural architecture performance prediction experiments (Section 4.2).

Results. As shown in Table 8, FGP outperforms the baselines in all settings, demonstrate that the effectiveness of FGP is not limited to a particular neural architecture encoder.

Setup. We use two additional benchmark datasets: (1) NAS-Bench-NLP [24], a dataset that consists of natural language processing models, and (2) NAS-Bench-Graph [37], a dataset that consists of graph representation learning models.

Results. As shown in Table 6, FGP outperforms the baselines in 5 out of 6 settings, demonstrate that the effectiveness of FGP does not limit to a computer vision domain.

B.9 Analysis regarding the usage of random vectors

In this section, we present additional experiment results regarding the usage of random vectors and matrices for our flow surrogate (Section 3). We analyze two questions: (Q1) Do the values of random vectors and matrices significantly impact the effectiveness of FGP in downstream tasks? (Q2) Does using random vectors and matrices for flow surrogates perform better than using representations of architectures obtained from flow-aware encoders?

B.9.1 Q1. Various random initializations

Setup. We use five different random initializations of vectors and matrices to compute flow surrogates. Then, we compare the performance of each initialization in the performance prediction task. To this end, we use ResGatedGCN [3] as the backbone neural architecture encoder. Other setups are the same as those used in our neural architecture performance prediction experiments (Section 4.2).

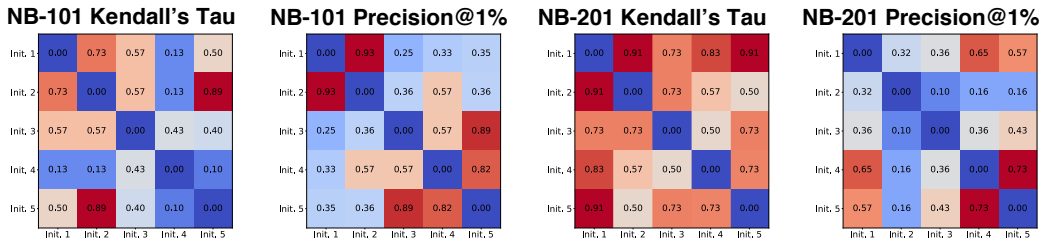


Figure 11: **Statistical test across diverse random initialization for flow surrogates.** p-values from the Wilcoxon Signed-Rank tests [46] between pairs of performance sets, each obtained from different initializations, are reported. At the significance level of $\alpha = 0.05$, none of the pairs provide sufficient statistical evidence to reject the null hypothesis. Thus, different random initializations for the flow surrogate yield the same performance distribution.

Table 9: **Performance prediction results under different derivations of flow surrogates.** Mean and standard deviation on each dataset and metric. All values are multiplied by 100. The best performance is highlighted in blue. Using random vectors to obtain flow surrogates outperforms using trained TA-GATEs as flow surrogates.

Flow surrogates	Kendall's Tau		Precision@1%		Precision@5%	
	NB-101	NB-201	NB-101	NB-201	NB-101	NB-201
TA-GATEs [34]	72.2 (5.6)	74.7 (11.4)	29.2 (7.2)	40.3 (8.5)	54.4 (7.3)	60.1 (5.1)
Random vectors propagated as in Section 3	74.8 (4.8)	82.2 (0.7)	37.5 (13.0)	48.9 (8.1)	61.7 (3.6)	62.3 (2.5)

We leverage the Wilcoxon-Signed-Rank test [46] to statistically compare the effects of different initialization. For each initialization setting, we conduct 9 performance prediction trials using flow surrogates generated with the corresponding initialization. For each pair of initializations, we apply the Wilcoxon Signed-Rank test to compare the performance sets derived from the respective initialization. The null hypothesis posits that the two performance distributions are identical, while the alternative hypothesis suggests they are distinct. We set the significance level at $\alpha = 0.05$. If the p-value of the test exceeds α, we lack sufficient statistical evidence to reject the null hypothesis, indicating that the performance sets from different initializations follow the same distribution.

Results. As shown in Figure 11, every p-value is smaller than 0.05, resulting in we do not have sufficient statistical evidence to reject the null hypothesis. These test results indicate that the performance sets derived from different random initializations of vectors and matrices for flow surrogates share the same distribution.

B.9.2 Q2. Other choices for flow surrogates

Setup. For flow surrogates, we use architecture embeddings obtained by the flow-aware neural architecture encoder, TA-GATEs [34], trained through supervised learning. In each experimental trial, we first train TA-GATEs using the training dataset. Next, the trained TA-GATEs are employed to obtain neural architecture representations for every neural architecture in the search space. Finally, these representations serve as flow surrogates for their corresponding neural architectures, which are then leveraged to pre-train another neural architecture encoder, ResGatedGCN with FGP.

Results. As shown in Table 9, flow surrogates derived using random vectors and matrices (as described in Section 3) outperform those derived using TA-GATEs, highlighting the effectiveness of flowing random vectors. We hypothesize that this result stems from the potential overfitting of flow-aware encoders to the training dataset, causing them to focus on specific architectural patterns present in the training data. In contrast, using random vectors likely avoids this bias, enabling the capture of broader patterns that are not limited to a specific subset of the dataset.

B.10 Analysis regarding alternative design choices of FGP

Recall that the proposed flow surrogate is achieved after a single round of sequential message passing (refer to Section 3.2). However, it is possible to perform message passing for multiple rounds. In this section, we analyze the effectiveness of flow-surrogate variants, that is achieved via multiple rounds of sequential message passing.

Table 10: **Performance prediction results under variants of the flow surrogate.** Mean and standard deviation on each dataset and metric. All values are multiplied by 100. The best performance is highlighted in blue. Using only a single round flow surrogate yields the best performance.

Propagation rounds	Kendall's Tau		Precision@1%		Precision@5%	
	NB-101	NB-201	NB-101	NB-201	NB-101	NB-201
Using only round 1 (Ours)	74.8 (4.8)	82.2 (0.7)	37.5 (13.0)	48.9 (8.1)	61.7 (3.6)	62.3 (2.5)
Using rounds 1 and 2	71.7 (5.4)	80.6 (0.5)	36.7 (5.9)	46.0 (6.0)	58.8 (3.4)	62.0 (1.9)
Using rounds 1, 2, and 3	70.7 (4.7)	81.6 (0.8)	34.0 (8.9)	47.3 (0.6)	58.2 (4.1)	62.5 (1.0)

Table 11: **Performance prediction results under various pooling functions.** Mean and standard deviation on each dataset and metric. All values are multiplied by 100. The best performance is highlighted in blue. Kendall's Tau is used as an evaluation metric. Sum pooling outperforms other pooling functions.

	Mean (variant 1)	Max (variant 2)	Sum (Ours)
NB-101	71.6 (5.8)	71.1 (6.4)	74.8 (4.8)
NB-201	81.7 (1.0)	81.5 (0.9)	82.2 (0.7)

Setup. We use two variants:

- **(Variant 1)** This variant uses **two** surrogates: (1) a flow surrogate obtained via a single message passing round and (2) a flow surrogate obtained via two consecutive message passing rounds. It uses different projection heads for respective flow surrogate reconstruction. Reconstruction is performed in a joint manner.
- **(Variant 2)** This variant uses three surrogates: a flow surrogate obtained via (1) a single message passing round, (2) a flow surrogate obtained via two consecutive message passing rounds, and (3) a flow surrogate obtained via three consecutive message passing rounds. It uses different projection heads for respective flow surrogate reconstruction.

All methods use ResGatedGCN as a backbone neural architecture encoder. Other setups are the same as those used in our neural architecture performance prediction experiments (Section 4.2).

Results. As shown in Table 10, a flow surrogate achieved via a single round of message passing, which is our proposed method, outperforms the alternatives in five out of six settings, justifying our design choice of flow surrogate.

B.11 Alternative pooling strategies

Note that we use the sum pooling to aggregate the messages from the neighbors (Section 3.2). In this section, we analyze the alternative pooling functions that are widely used in GNN literature.

Setup. We use the two strategies: (1) mean pooling and (2) max pooling. Specifically, we replace the sum pooling function with one of the two functions and use the resulting flow surrogate as the final flow surrogate. We use ResGatedGCN as the backbone neural architecture encoder, and other settings are the same as in Section 4.2.

Results. As shown in Table 11, sum pooling, which FGP uses to obtain flow surrogate, outperforms alternative strategies, demonstrating the effectiveness of our design choice for pooling function.

B.12 Expressiveness of our flow surrogate

In this section, we provide preliminary analysis regarding the expressiveness of our flow surrogate.

Setup. We analyze whether our flow surrogate can discriminate architectures containing different operations. We conduct three binary classification tasks to determine whether a given neural architecture includes: (1) a 1×1 convolution operation, (2) a 3×3 convolution operation, and (3) a pooling operation. We use flow surrogate representations as input features and train an MLP as the classifier, performing 10 trials. The architectures are split into 80% for training and 20% for testing.

Table 12: **Operation classification results.** Mean and standard deviation on each dataset and metric. All values are multiplied by 100. In every case, flow surrogates achieve accuracy higher than 92.

	1×1 Conv.	3×3 Conv.	Pooling
NB-101	93.2 (0.3)	99.9 (0.1)	92.1 (0.7)
NB-201	98.9 (0.3)	99.9 (0.1)	1.0 (0.1)

Table 13: **Performance under diverse backbone architecture encoders.** Mean and standard deviation on each dataset and metric. All values are multiplied by 100. Kendall's Tau is used as an evaluation metric. The best performance is highlighted in blue. Since the MLP-based model does not return node embeddings, GMAE is not applicable. Across diverse backbone encoders, FGP consistently outperforms the baseline methods.

Backbone	Datasets	w/o pre-training	GMAE	ZC-Proxy	FGP
MLP-based	NB-101	42.5 (5.4)	-	49.8 (7.3)	65.2 (4.1)
	NB-201	59.6 (2.2)	-	70.2 (1.8)	72.3 (1.3)
Transformer-based	NB-101	62.7 (3.6)	63.2 (4.5)	64.1 (8.5)	69.7 (6.3)
	NB-201	63.2 (2.0)	65.7 (2.4)	69.2 (3.2)	74.3 (1.5)

Results. As shown in Table 12, our flow surrogate achieves higher than 92% accuracy in all the cases. This result suggests that our flow surrogate can effectively distinguish architectures containing different operations, achieving

B.13 Theoretical properties of our flow surrogate

In this section, we investigate the theoretical properties of our flow surrogate. In a nutshell, we provide a preliminary theoretical analysis demonstrating that our flow-surrogate computation is invariant to node permutations (i.e., node indexing), a desirable property when representing the information flow of a neural architecture to ensure effective downstream task performance.

Motivation. [15, 34] demonstrates the importance of accurately capturing information flow for the downstream tasks (e.g., performance prediction and neural architecture search). In the NAS-Bench datasets used in our experiments, the information flow within a neural architecture is independent of the indexing of individual operations. Accordingly, it is desirable for the computation of flow surrogates to be *permutation-invariant* with respect to these operations.

Proof sketch Thost and Chen [40] theoretically demonstrates that asynchronous message passing on a directed acyclic graph—also used in our flow-surrogate computation—is invariant to node permutations when the permutation-invariant pooling function is employed for neighbor aggregation. Their theoretical result extends to our setting, indicating that our method likewise ensures permutation invariance regarding node indexing.

B.14 Alternative backbone neural architecture encoders

Recall that we use graph-based neural architecture encoders for our experiments, given that neural architectures are often expressed as graphs. In this section, we investigate the effectiveness of FGP under non-graph-based neural architectures.

Setup. We use two non-graph-based neural architecture encoders: MLP-based model [44] and Transformer-based model [31]. Specifically, we first pre-train each architecture encoder via a certain pre-training method, and then fine-tune the pre-trained encoder with the performance prediction task. Other settings are the same as in Section 4.2.

Results. As shown in Table 13, FGP consistently outperforms the baseline methods across diverse non-graph-based backbone neural architecture encoders, demonstrating that the effectiveness of FGP is not limited to a particular graph-based backbone encoder.

B.15 Hyperparameter sensitivity

In this section, we investigate the hyperparameter sensitivity of FGP.

Table 14: **Hyperparameter sensitivity analysis.** Mean and standard deviation on each dataset and metric. All values are multiplied by 100. Kendall's Tau is used as an evaluation metric. Across diverse settings of λ_1 and λ_2, FGP consistently outperforms the baseline methods.

	$(\lambda_1, \lambda_2) = (\frac{1}{2}, \frac{1}{2})$	$(\lambda_1, \lambda_2) = (\frac{1}{3}, \frac{2}{3})$	$(\lambda_1, \lambda_2) = (\frac{2}{3}, \frac{1}{3})$	ZC-Proxy	GMAE
NB-101	74.8 (4.8)	74.8 (4.4)	74.3 (4.5)	68.3 (6.7)	68.1 (4.7)
NB-201	82.2 (0.7)	82.3 (0.8)	81.7 (0.8)	79.9 (0.8)	74.8 (1.2)

Table 15: **FlowerFormer analysis under diverse pre-training data size.** Mean and standard deviation on each dataset and metric. All values are multiplied by 100. Kendall's Tau is used as an evaluation metric. Performance of the FlowerFormer pre-trained with FGP tends to increase as the size of the pre-training data increases.

Pre-training data ratio	0%	20%	40%	60%	80%	100%
NB-101	74.0 (3.6)	74.6 (3.8)	75.9 (1.9)	75.8 (4.0)	76.2 (5.1)	76.3 (3.6)
NB-201	77.3 (1.5)	81.1 (0.8)	82.3 (1.2)	82.4 (1.3)	82.7 (1.4)	83.5 (1.7)

Setup. We analyze the coefficients of each loss term, which are λ_1 and λ_2 described in Section 3.3. As detailed in Appendix A.4, search space of the coefficients are as follows: $(\lambda_1, \lambda_2) \in \{(\frac{1}{3}, \frac{2}{3}), (\frac{1}{2}, \frac{1}{2}), (\frac{2}{3}, \frac{1}{3})\}$. Therefore, we measure the performance of FGP under each configuration. We use ResGatedGCN as the backbone neural architecture encoder, and other settings are the same as in Section 4.2.

Results. As shown in Table 14, across diverse settings of λ_1 and λ_2, FGP consistently outperforms the baseline methods, demonstrating the robustness of FGP under the choice of λ_1 and λ_2.

B.16 Analysis regarding performance gain in FlowerFormer

Recall that FlowerFormer [15] is a flow-based encoder, which captures information flow within the neural architecture by using its architectural design. However, as shown in Table 1, the performance of FlowerFormer even improves with FGP. In this section, we further investigate this phenomenon. To this end, we first present the high-level hypothesis regarding the reason behind this phenomenon, and provide experiments that further support our hypothesis.

Hypothesis. We hypothesize that FGP enables FlowerFormer to learn a broader range of information flow beyond what is present in labeled architectures by leveraging a large number of unlabeled architectures, thereby enhancing its ability to capture more diverse flow patterns. When trained solely on labeled samples, FlowerFormer tends to learn information flow patterns limited to that specific set. In contrast, FGP leverages a much larger pool of neural architectures whose ground-truth performance is unknown, exposing FlowerFormer to a wider range of information flow patterns. This exposure enables FlowerFormer to learn more diverse and generalizable information flow representations.

Setup. By varying the number of neural architectures used during FGP pre-training, we aim—though not precisely—to control the diversity of information flow patterns to which FlowerFormer is exposed. A positive correlation between dataset size and FlowerFormer performance suggests that FGP helps FlowerFormer encounter a broader range of information flow patterns, leading to improved accuracy in performance prediction. We vary the proportion of the pre-training dataset used for FGP training—0% (no pre-training), 20%, 40%, 60%, 80%, and 100%—and evaluate FlowerFormer's performance on the performance prediction task. Kendall's Tau is used as the evaluation metric, and all other settings follow those described in Section 4.2.

Results. As shown in Table 15, the performance of FlowerFormer pre-trained with FGP tends to increase as the size of the pre-training dataset increases. This result supports our hypothesis on the performance improvement of the flow-based encoder (FlowerFormer) with FGP.

C Flow surrogate details

C.1 Details of node embeddings and messages

In this section, we elaborate on the node embeddings (i.e., operation representations) $\mathbf{h}_i \in \mathbb{R}^k, \forall v_i \in \mathcal{V}$, messages of the order-1 nodes $\mathbf{r}$, [2] and projection matrix $\mathbf{W} \in \mathbb{R}^{2k \times k}$.

- **Node embeddings.** We first randomly generate a transformation matrix $\mathbf{P} \in \mathbb{R}^{|\mathcal{O}| \times k}$, where $\mathcal{O}$ is a set of all the possible operations. Each entry of $\mathbf{P}$ is sampled from $\mathcal{N}(0, \sigma^2)$ independently to each other. Then, we multiply $\mathbf{P}$ with node features $\mathbf{X}$, obtaining the node embedding matrix $\mathbf{H}$ (i.e., $\mathbf{H} = \mathbf{XP}$), where $\mathbf{H}_{i,:} = \mathbf{h}_i$ holds.
- **Messages of the order-1 nodes.** We randomly sample each element of $\mathbf{r}$ from the uniform distribution $U(0, 1)$.
- **Projection matrix.** We randomly sample each element of $\mathbf{W}$ from $\mathcal{N}(0, \sigma^2)$.

C.2 Hyperparameters for obtaining flow surrogate

We provide a search space or a fixed value for each hyperparameter related to the flow surrogate:

- **Standard deviation of Gaussian distribution σ.** This is tuned within $\{10^{-1}, 10^0\}$.
- **Dimension k.** This is tuned within $\{4, 5, \cdots, 12\}$.
- **Identity coefficient α.** This is fixed as $\alpha = 0.5$.

D Zero-cost proxy details

In this section, we provide further details regarding zero-cost proxy, which we use its prediction as an auxiliary training objective of FGP.

D.1 Descriptions of zero-cost proxies

Overview. Zero-cost proxies are pruning-at-initialization metrics that represent certain characteristics of a neural architecture [1]. For a given neural architecture, these metrics can be obtained by simply building a deep learning model having the corresponding neural architecture (e.g., ℓ_2-norm of parameters) or performing a single forward pass and gradient computation (e.g., ℓ_2-norm of parameters' gradients). Therefore, compared to actual model training—which requires a substantial number of training iterations (i.e., forward passes and backpropagation)—obtaining zero-cost proxies is significantly more economical [1].

Usage of zero-cost proxies. Numerous studies on Neural Architecture Search (NAS) have demonstrated that these metrics exhibit a strong correlation with the ground-truth performance of neural architectures [1, 56, 14]. Consequently, various NAS approaches leverage zero-cost proxies to quickly assess the potential effectiveness of candidate architectures for the target task and dataset [14, 27]. Similarly, Zhao et al. [56] proposed pre-training a neural architecture encoder to predict the zero-cost proxies of a neural architecture, enabling the encoder to identify architectures more likely to achieve high performance. Note that this zero-cost-prediction-based pre-training [56] has been leveraged as our baseline method, which is called ZC-Proxy. Specifically, ZC-Proxy pre-trains the encoder following the scheme described in Appendix D.2.

D.2 Our usage of zero-cost proxy

Overview. Motivated by the recent success of zero-cost proxies in NAS, we use predicting a zero-cost proxy of a neural architecture as an auxiliary learning objective $\mathcal{L}_{aux}$. We use Jacobian Covariance zero-cost proxy [1], measuring how well a neural architecture distinguishes distinct inputs. We train a neural architecture encoder to predict the Jacobian Covariance of a given neural architecture. For NAS-Bench-101 [51] and NAS-Bench-301 [39], we use proxies provided by Krishnakumar et al. [25]. For NAS-Bench-201 [8], we use proxies provided by Abdelfattah et al. [1].

[2] Note that all the order-1 nodes share the same embedding $\mathbf{r}$.

Usage. Consider $\mathbf{z}^{(i)} \in \mathbb{R}^d$, an embedding of a neural architecture $\mathcal{G}^{(i)}$, obtained by a neural architecture encoder f_θ. We use a zero-cost-proxy regressor u_ρ to predict the architecture's zero-cost proxy, denoted as $c^{(i)} \in \mathbb{R}$. Specifically, the predicted zero-cost proxy $\hat{c}^{(i)} \in \mathbb{R}$ is obtained as follows: $\hat{c}^{(i)} = u_\rho(\mathbf{z}^{(i)})$. Since the objective of learning zero-cost proxies is to train a model to identify which architectures are *more likely* to perform better, we adopt a ranking-based loss. This loss encourages the model to focus on *relative performance*, specifically identifying which architecture outperforms another. To achieve this, we use margin ranking loss, which is widely used in training a neural architecture encoder [34, 15]. The loss function is formalized as:

$$\mathcal{L}_{aux} = \sum_{(i,j):c^{(i)}>c^{(j)}} \max\left(0, m - (\hat{c}^{(i)} - \hat{c}^{(j)})\right), \tag{2}$$

where $\mathcal{G}^{(i)}$ and $\mathcal{G}^{(j)}$ are two neural architectures within the same batch, and m is a margin hyperparameter. The parameters θ and ρ are optimized using gradient descent to minimize the auxiliary loss $\mathcal{L}_{aux}$ (Eq. 2).

E Discussions

In this section, we discuss limitations and broader impacts of our research.

E.1 Limitations and potential future works

While our work demonstrates empirical effectiveness in neural architecture encoding across various benchmark datasets and applications (Section 4), the theoretical properties of FGP remain under-explored—specifically, what types of flows it can or cannot capture. While we provide preliminary analysis regarding these concepts in Appendices B.12 and B.13, a further rigorous and in-depth analysis regarding them would enrich our understanding of our method. Thus, further theoretical investigation into our flow surrogate offers promising directions for improvement. In addition, given findings that LLMs understand graphs and produce interpretable graph representations [21], incorporating them into FGP is a promising path to improve interpretability. Moreover, several recent methods represent neural architectures in diverse formats, such as hypergraph [28], which model interactions among multiple nodes [20]. Therefore, extending FGP to such approaches would be an interesting direction.

E.2 Broader impacts

While our work focuses on neural architecture encoding, our approach has the potential to generalize to other data structures that exhibit a notion of *flow*, such as electrical circuits [2] and Petri Nets [36]. Accordingly, our flow surrogate presents practical opportunities for representing such structures in these domains.

F NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We propose a generative-pretraining method for neural architecture encoding.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss them in Appendix E.

   Guidelines:

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: We discuss them in Appendix B.13.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We present detailed experimental settings and hyperparameters, together with the source code.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We make all our code and datasets publicly available through https://github.com/kswoo97/FGPAnom.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We provide detailed settings and hyperparameter search space of our experiments.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We provide mean and standard deviation of our experimental results obtained via multiple trials.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide detailed information regarding the machines used in our experiments.

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

Answer: [Yes]

Justification: We carefully checked the NeurIPS Code of Ethics, and believe that we did not violate any of the ethical terms.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We discuss them in Appendix E.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We believe that our work does not pose such risks.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We adequately cited existing works.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We provide detailed README instructions for our code in https://github.com/kswoo97/FGPAnom.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: We do not contain any crowdsourcing and human-subjects-related experiments.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: We do not contain any crowdsourcing and human-subjects-related experiments.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: We only used LLMs to improve the writing.