Progressive Neural Architecture Generation

Supplementary Material

In the supplementary materials, we provide 1) more detailed theoretical analysis for SCC (Sec. 6), 2) more details on optimization of **PNAG** (Sec. 7), 3) more details on experimental setup and implementation of **PNAG** (Sec. 8), and 4) additional experiments (Sec. 9).

6. Theoretical Analysis of SCC

This section gives a more detailed inference process for the theoretical analysis of the SCC strategy. To prove the effectiveness of the SCC, we utilize the **Lyapunov Stability Theorem** to demonstrate that the intermediate sub-architectures generated at each step gradually approach the input feature, ensuring a stable generation path and reducing accumulated deviation.

Proposition. *Asymptotic stability of the sub-architecture generation trajectory is guaranteed if the learning rate satisfies the condition* $\alpha < -\frac{2\left(e-e_t'\right)^T \nabla_{e_t'} \mathcal{L}\left(e_t'\right)}{\left\|\nabla_{e_t'} \mathcal{L}(e_t')\right\|_2^2}$.

Lyapunov Stability Theorem. The Lyapunov Stability Theorem is a tool used to analyze the stability of dynamic systems. By constructing a Lyapunov function $V(x)$, we can determine whether a system tends toward stability over time. Suppose the state evolution of a system is described by the following dynamic equation:

$$\dot{x} = f(x), \tag{14}$$

where x is the state variable of the system. The core idea of the Lyapunov Stability Theorem is to introduce a Lyapunov function $V(x)$, which satisfies the following conditions:

- **Non-negativity:** $V(x) \geq 0$, and $V(x) = 0$ and only if $x = x^*$ (the equilibrium point).
- **Negative definiteness:** The evolution of the system near the equilibrium point causes $V(x)$ to decrease monotonically, that is, $\dot{V}(x) = \frac{dV}{dt} \leq 0$. If $\dot{V}(x) < 0$ for $x \neq x^*$, the system is asymptotically stable.

The time derivative of the Lyapunov function $V(x)$ is expressed as:

$$\dot{V}(x) = \nabla V(x) \cdot f(x) \tag{15}$$

if $\dot{V}(x) \leq 0$ for all x, then the system is stable at the equilibrium point.

Proof. In the SCC strategy, we aim to ensure that the intermediate sub-architecture g_t gradually approaches the input feature e at each generation step through SCC. Thus, we define the Lyapunov function as:

$$V\left(e_t'\right) = \left\|e - \text{interpolate}(e_t', (N, N))\right\|_2. \tag{16}$$

For simplicity, we represent this equation as:

$$V\left(e_t'\right) = \left\|e - e_t'\right\|_2^2, \tag{17}$$

where e_t' is the feature map of sun-architecture g_t.

This Lyapunov function describes the "deviation energy" between the intermediate sub-architecture and the input feature. We aim for $V(e_t')$ to decrease at each generation step, ensuring stability in the generation process.

In a discrete generation process, we approximate the time derivative of the Lyapunov function with the difference form:

$$\Delta V\left(e_t'\right) = V\left(e_{t+1}'\right) - V\left(e_t'\right). \tag{18}$$

If $\Delta V\left(e_t'\right) < 0$, then the deviation energy decreases at each step, indicating a stable generation path. Substituting the update equation $e_{t+1}' = e_t' - \alpha \nabla_{e_t'} \mathcal{L}\left(e_t'\right)$, where α is the learning rate and $\mathcal{L}\left(e_t'\right)$ is the loss function in the main paper, we then have:

$$\Delta V\left(e_t'\right) = \left\|e - \left(e_t' - \alpha \nabla_{e_t'} \mathcal{L}\left(e_t'\right)\right)\right\|_2^2 - \left\|e - e_t'\right\|_2^2. \tag{19}$$

Expanding $\Delta V\left(e_t'\right)$:

$$\begin{aligned}
\Delta V\left(e_t'\right) &= \left(e - e_t' + \alpha \nabla_{e_t'} \mathcal{L}\left(e_t'\right)\right)^T \left(e - e_t' + \alpha \nabla_{e_t'} \mathcal{L}\left(e_t'\right)\right) \\
&\quad - \left\|e - e_t'\right\|_2^2 \\
&= \left\|e - e_t'\right\|_2^2 + 2\alpha \left(e - e_t'\right)^T \nabla_{e_t'} \mathcal{L}\left(e_t'\right) \\
&\quad + \alpha^2 \left\|\nabla_{e_t'} \mathcal{L}\left(e_t'\right)\right\|_2^2 - \left\|e - e_t'\right\|_2^2 \\
&= 2\alpha \left(e - e_t'\right)^T \nabla_{e_t'} \mathcal{L}\left(e_t'\right) + \alpha^2 \left\|\nabla_{e_t'} \mathcal{L}\left(e_t'\right)\right\|_2^2.
\end{aligned} \tag{20}$$

To ensure $\Delta V\left(e_t'\right) < 0$, we need to satisfy the following inequality:

$$2\alpha \left(e - e_t'\right)^T \nabla_{e_t'} \mathcal{L}\left(e_t'\right) + \alpha^2 \left\|\nabla_{e_t'} \mathcal{L}\left(e_t'\right)\right\|_2^2 < 0. \tag{21}$$

We obtain the constraint for the learning rate α:

$$\alpha < -\frac{2\left(e - e_t'\right)^T \nabla_{e_t'} \mathcal{L}\left(e_t'\right)}{\left\|\nabla_{e_t'} \mathcal{L}\left(e_t'\right)\right\|_2^2}. \tag{22}$$

This constraint ensures $\Delta V\left(e_t'\right) < 0$, meaning the Lyapunov function decreases at each step, which validates the asymptotic stability of the generation path.

Based on the Lyapunov Stability Theorem, by constructing the Lyapunov function $V\left(e_t'\right) = \left\|e - e_t'\right\|_2^2$ and controlling the learning rate α, we can demonstrate that the SCC strategy gradually reduces the deviation energy $V\left(e_t'\right)$ at

each generation step. This ensures the stability and convergence of the generation path. The consistency regularization term in SCC guides intermediate steps towards the input feature, effectively suppressing accumulated deviation and providing stability for the generation process. This analysis provides a solid theoretical foundation for the effectiveness of the SCC method.

7. Optimization Details of PNAG

In this section, we provide more details not mentioned in the main paper, including the neural architecture encoding (Sec. 7.1) and the task encoding (Sec. 7.2). In addition, we will describe the application of **PNAG** to conditional NAG and transferable NAG. (Sec. 7.3).

7.1. Neural Architecture Encoding

The search space in our study is cell-based architectures. Following [17, 18], each cell can be represented as a labeled directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents nodes and $\mathcal{E}$ represents edges. Each node $v_i \in \mathcal{V}$ is associated with an operation from a predefined set, while the edges are represented by an upper triangular adjacency matrix A. The neural architecture encoder is built on a Transformer encoder, with an embedding layer and L Transformer blocks. Each node v_i is first fed into a semantic embedding layer:

$$\text{Emb}_i = \text{Embedding}\left(v_i\right). \tag{23}$$

The embedded vectors are then contextualized across L Transformer layers, where the hidden state after the $l-th$ layer is updated using attention:

$$\begin{aligned} Q_k = H^{l-1}W_q^k, \quad K_k = H^{l-1}W_k^k, \quad V_k = H^{l-1}W_v^k, \\ \hat{H}_i^l = \text{softmax}\left(\frac{Q_k K_k^T + M}{\sqrt{d_h}}\right)V_k, \\ H^l = \text{concat}\left(\hat{H}_1^l, \ldots, \hat{H}_{n_{\text{headed}}^l}^l\right), \\ H^l = \text{ReLU}\left(H^l W_1 + b_1\right)W_2 + b_2, \end{aligned} \tag{24}$$

where Q_k, K_k, and V_k are the query, key, and value vectors for the attention operation of the $k-th$ head respectively, W_q^k, W_k^k, W_v^k are learnable parameters, d_h is the dimension of the hidden state, n_{headed}^l is the number of the head in transformer block, W_1, b_1, W_2, b_2 are weights in the feed-forward layer, and M is an attention mask in the Transformer, which is determined by the value of the adjacency matrix A of a neural architecture Arc following the equation:

$$\boldsymbol{M}_{i,j} = \begin{cases} 0, & \text{if} \quad Arc_{i,j} = 1 \\ -\infty, & \text{if} \quad Arc_{i,j} = 0, \end{cases} \tag{25}$$

where $Arc_{i,j}$ denotes the edge between the node i and j. In our study, a pair of nodes (operations) in the architecture is considered dependent if a directed edge is connected. The output of the final layer is the representation of the cell $\mathcal{G}$, which also serves as the result of the neural architecture after encoding.

7.2. Task Encoding

In Sec. 3.5, we extend **PNAG** to transferable NAGs for unseen tasks. Prior to this, we need to encode the tasks, as the datasets of different tasks are not uniform in terms of format, size, *etc.*, which puts a higher demand on the generalization ability of the predictor.

To improve the generalization capability, it is essential to encode the tasks and strengthen the relationships between them effectively. Following TASK2VEC [1], we represent the task embedding using the Fisher information Matrix (FIM). Specifically, given an input image x and an unknown label y, we use a probe network to model the joint distribution $p_w(y \mid x)$ parameterized by weights w. The goal is to approximate the posterior $p(y \mid x)$ by minimizing the cross-entropy loss function:

$$H_{p_w,\hat{p}}(y \mid x) = \mathbb{E}_{x,y\sim\hat{p}}\left[-\log p_w(y \mid x)\right], \tag{26}$$

where $\hat{p}$ is the empirical distribution based on the training dataset $\boldsymbol{D} = \{(x_i, y_i)\}_{i=1}^N$. Further, consider that not all network weights contribute equally to the task. To quantify the importance of weight, a perturbation δw is introduced to the weights w, and the Kullback-Leibler (KL) divergence between the original and perturbed distributions is computed. This can be approximated by:

$$\mathbb{E}_{x\sim\hat{p}}\left[KL\left(p_{w+\delta w}(y \mid x) \| p_w(y \mid x)\right)\right] = \delta w^T \cdot F \cdot \delta w, \tag{27}$$

where F is FIM and is defined as:

$$F = \mathbb{E}_{x,y\sim\hat{p}(x)p_w(y|x)}\left[\nabla_w \log p_w(y \mid x) \nabla_w \log p_w(y \mid x)^T\right]. \tag{28}$$

This matrix captures the sensitivity of the joint distribution $p_w(y \mid x)$ to changes in the network parameters. The corresponding entries in the FIM will be small for parameters that do not significantly affect classification performance.

In this way, the representation of the different tasks with FIM is not only unified in format (they are all represented with the weights of the probe network) but also the connections between the different tasks can be analysed by the matrix, which is crucial for the generalization ability of the predictor.

7.3. Condititional NAG and Transferable NAG

We aim to generate a suitable neural architecture for the **unseen task**. Therefore, based on **PNAG**, we consider conditional Neural Architecture Generation (conditional NAG), where the required property y (*e.g.*, accuracy) is input to generate architectures. To verify that the generated architecture

meets the requirements, we use a neural predictor $P(y \mid \hat{Arc})$ to evaluate the properties of the architecture. However, in the implementation, the property is task-specific, so the neural predictor tends to be dataset-aware, *i.e.*, $P(y \mid \boldsymbol{D}, \hat{Arc})$, which facilitates subsequent transfer to unseen tasks.

The PANG-based generation model (including the predictor) is trained across the task distribution using a meta-dataset $\mathcal{S}$ (detailed in Sec. 8.3) composed of triplets containing (dataset, architecture, accuracy). Specifically, we define the meta-dataset as $\mathcal{S} := \left\{\left(\hat{Arc}^{(i)}, y_i, \boldsymbol{D}_i\right)\right\}_{i=1}^{K}$, which includes K tasks. Each task is constructed by randomly sampling an architecture $Arc^{(i)}$, accompanied by its accuracy y_i evaluated on the dataset $\boldsymbol{D}_i$. The training objective is to minimize the following loss function:

$$\phi^* \in \arg\min_{\phi} \sum_{i=1}^{K} \mathcal{L}\left(y_i, P\left(\boldsymbol{D}_i, \hat{Arc}^{(i)}\right)\right). \tag{29}$$

In the inference phase, the model aims to generate the architecture that maximizes the accuracy of the dataset, which inherently represents the process of **transferable NAG**. **Notably**, during the training phase, *Property* refers to the accuracy of an architecture on the dataset, represented by a triplet (dataset, architecture, accuracy), where the architecture is encoded in a continuous form. In the inference phase, *Property* includes only dataset information, with the architecture and performance values initially obtained through random sampling or other sampling strategies (detailed in Sec. 8.5). These values are then iteratively refined and updated under the guidance of the predictor until the optimal architecture is generated.

Furthermore, when confronted with an unseen task $\tilde{\boldsymbol{D}}$, the transferable NAG can be reformulated as:

$$\begin{aligned} &p\left(g_1, g_2, \ldots, g_T \mid P(y \mid \tilde{\boldsymbol{D}}, \hat{Arc})\right) = \\ &\prod_{t=1}^{T} p\left(g_t \mid, g_1, g_2, \ldots, g_{t-1}, P(y \mid \tilde{\boldsymbol{D}}, \hat{Arc})\right). \end{aligned} \tag{30}$$

8. Details on Experiment Setup

In this section, we first introduce the search space (Sec. 8.1), then describe the baselines (Sec. 8.2) involved in the comparisons and the datasets (Sec. 8.3) used in the experiments. Finally, we give the training pipeline of the architecture (Sec. 8.4), the acquisition optimization strategy used in **PNAG** (Sec. 8.5), and the implementation details of the experiments (Sec. 8.6)

8.1. Search Space

In this study, as the main target of the generative modelling treatment, the architecture Arc, which comprises N nodes, is characterized by its operator type matrix $\mathbf{V} \in \mathbb{R}^N$ and an upper triangular adjacency matrix $\mathbf{E} \in \mathbb{R}^{N \times N}$. Therefore, Arc can be denoted as $(\mathbf{V}, \mathbf{E}) \in \mathbb{R}^N \times \mathbb{R}^{N \times N}$, where $\mathbf{V}$ is a discrete matrix with values in the range $[0, O]$, O denotes the number of predefined structure (operation) types in the search space.

NAS-Bench-201 search space is composed of cell-based neural architectures. Each cell is a directed acyclic graph and consists of a fixed form of 4 nodes and 6 edges, where the nodes all denote **Sum** operations and each edge denotes a different operation, including **zeroize**, **skip connection**, 1×1 **convolution**, 3×3 **convolution**, and 3×3 **average pooling**. Thus, there are 15,625 architectures in the entire search space. In the experiment, to facilitate the manipulation of edges and nodes, we transformed the original graph representation of the architectures into a new graph representation. In the new graph representation, each node represents an operation and each edge represents a connection. After the transformation, each cell contains 8 nodes (*i.e.*, N is 8), while each node can choose 7 operations (*i.e.*, O is 7) due to the inclusion of input and output operation. Thus, during the generation process, the number of operations in each sub-architecture is gradually scaled up from 3 to 8, for a total of 6 steps. Furthermore, the macro skeleton is constructed by stacking various components, including one stem cell, three stages consisting of 5 repeated cells each, residual blocks [8] positioned between the stages, and a final classification layer. The final classification layer consists of an average pooling layer and a fully connected layer with a softmax function. The stem cell, which serves as the initial building block, comprises a 3×3 convolution with 16 output channels, followed by a batch normalization layer. Each cell within the three stages has a varying number of output channels: 16, 32, and 64, respectively. The intermediate residual blocks feature convolution layers with a stride of 2, enabling down-sampling.

DARTS search space is structured around a supernet composed of three core components: an initial stem layer, a sequence of eight searchable cells, and a final classification head with global average pooling. Normal and reduction cells are used, where reduction cells are positioned at 1/3 and 2/3 of the network depth. Each cell, regardless of type, is a directed acyclic graph with 7 nodes and 14 edges, and each edge can be assigned one of eight candidate operations, including: (1) none, (2) 3×3 average pooling, (3) 3×3 max pooling, (4) skip-connection, (5) 3×3 SepConv, (6) 5×5 SepConv, (7) 3×3 DilConv, (8) 5×5 DilConv. Assuming a consistent architecture for both normal and reduction cells, this design yields 10^{18} possible architectures. Similar to NAS-Bench-201, we redefine the roles of nodes and edges within each cell. After this transformation, the resulting cell contains 12 nodes (*i.e.*, N is 12), while each node can choose 10 operations (*i.e.*, O is 10). The operation of each sub-architecture is gradually scaled up from 3 to 10.

MobileNetV3 search space is a layer-wise space, containing optional building blocks **MBConvs**, **squeeze and excitation**, and **modified swish nonlinearity**. Each architecture in the search space consists of 5 stages, and the number of building blocks in each stage ranges from 2 to 4. Therefore, the maximum number of architectures is $5 \times 4 = 20$. For each building block, the kernel size can be chosen from the set $\{3, 5, 7\}$, and the expansion ratio can be chosen from $\{3, 4, 6\}$. This implies that there are a total of $3 \times 3 = 9$ possible operation types available for each layer. Therefore, the size of N is 20, and the size of O is 9. The search space contains approximately 10^{19} architectures, offering a larger search space for architecture generation. Furthermore, the number of nodes of the sub-architecture is gradually expanded from 3 to 22, for a total of 20 steps. To improve the computational efficiency, we limit the generation step to 11 ($\{3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 22\}$).

8.2. Baselines

In this section, we give a detailed description of all the baselines in Table 1. We classify the comparison algorithms according to different optimization strategies.

Random-based NAS. RSPS explores random search in NAS, focusing on its reproducibility. It uses random sampling for architectures and evaluates performance to select the best.

Evolution-based NAS. RFGIAug proposes enhancing performance predictors in NAS through graph isomorphism. It samples architectures by augmenting graph representations and optimizes by improving predictor accuracy. **HAAP** uses homogeneous architecture augmentation to improve NAS predictors. It samples similar architectures and optimizes by refining the predictor's generalization.

Gradient-based NAS. Ostr-DARTS optimizes NAS by leveraging operation strength. It samples architectures using differentiable methods and optimizes by dynamically pruning weak operations. **STEN** proposes a one-shot NAS method using self-evaluated template networks. It samples architectures within a single supernetwork and optimizes through self-evaluation of shared weights. **GDAS** introduces a fast NAS approach using weight-sharing and early stopping, leveraging a regularized evolution algorithm for architecture sampling and progressive resizing to improve efficiency. **PC-DARTS** proposes partial channel connections to improve memory efficiency in NAS. It samples architectures by operating on a subset of channels and optimizes using differentiable search with reduced computational overhead. **DrNAS** introduces a Dirichlet distribution to model architecture sampling probabilities. It optimizes architectures using differentiable search by dynamically updating the distribution to balance exploration and exploitation.

Bayesian optimization-based NAS. BOHB combines Bayesian optimization with Hyperband to efficiently sample hyperparameters. It optimizes by balancing exploration and exploitation through Bayesian modeling while leveraging Hyperband for early stopping. **GP-UCB** involves using a vanilla Gaussian Process (GP) surrogate, where the Upper Confidence Bound (UCB) acquisition function is employed. **BANANAS** combines Bayesian optimization with an ensemble of neural predictors to guide NAS. It samples architectures using acquisition functions and optimizes by training predictors on architecture-performance pairs. **NASBOWl** employs Bayesian optimization with Weisfeiler-Lehman graph kernels for interpretable NAS. It samples architectures as graphs and optimizes by leveraging kernel-based similarity measures to predict performance. **HEBO** introduces a high-dimensional Bayesian optimization framework for efficient hyperparameter search. It samples configurations using a variational model and optimizes with trust-region constraints to balance exploration and exploitation.

Transferable-based NAS or NAG. TransferNAS (NAG) methods differ from traditional NAS methods mainly in their ability to rapidly adapt to new datasets using a priori knowledge. Traditional NAS approaches typically build the search process from scratch each time they are applied to a different dataset, which can be time-consuming and computationally expensive. In contrast, TransferNAS (NAG) approaches build on insights and patterns learned from previous tasks, allowing them to omit the search process when working with new datasets. This knowledge transfer not only accelerates the search but also typically results in architectures that are better suited to the new task, leading to improved performance.

MetaD2A focuses on transferable NAS by learning to generate neural architecture graphs conditioned on datasets. It samples architectures using a graph generative model and optimizes by aligning generated architectures with dataset-specific performance metrics. **TNAS** introduces a meta-learned Bayesian surrogate model to accelerate NAS across datasets. The method leverages prior architecture-performance data from source tasks to guide sampling for target tasks. Architectures are sampled using Bayesian optimization with an acquisition function informed by the surrogate, which predicts performance on the target dataset. Optimization involves iterative updates to the surrogate with task-specific evaluations, enabling efficient transferability and reduced search time. **DiffusionNAG** employs a diffusion model to generate transferable neural architectures guided by a performance predictor. Architectures are sampled as iterative refinements of initial random graphs using the diffusion model, conditioned on dataset-specific features. Optimization integrates a predictor to evaluate and guide the refinement process, ensuring that generated architectures align with the desired performance metrics on the target task. Different from DiffusionNAG, **`PNAG` (our method)** uses an autoregressive approach for architecture generation, focus-

ing on improving the generation efficiency and generation quality. We use a predictor-guided approach to iteratively generate suitable architectures through acquisition optimization strategies.

8.3. Datasets

In this section, we provide a brief description of the datasets and meta-dataset used in the experiments.

Image Datasets. We evaluate our method on four datasets, including CIFAR-10, CIFAR-100, Aircraft, and Oxford IIIT Pets. The details of the datasets are as follows: **CIFAR-10**. CIFAR-10 is a dataset consisting of 60,000 color images across 10 different classes, with 6,000 images per class. Each image is of size 32×32 pixels and represents everyday objects such as animals and vehicles. **CIFAR-100**. CIFAR-100 is similar to CIFAR-10 but contains 100 different classes, each with 600 images, making a total of 60,000 images. Each class is more granular, and the dataset provides a more challenging classification task with the same 32×32 pixel format. **Aircraft**. The Aircraft dataset contains around 10,000 images of aircraft, categorized into 30 different aircraft model variants. It is used for fine-grained visual classification tasks, focusing on distinguishing between different types of airplanes. **Oxford-IIIT Pets**. The Oxford-IIIT Pets dataset consists of 7,349 images of 37 different breeds of cats and dogs. Each breed has roughly 200 images, and the dataset is used for both fine-grained classification and segmentation tasks. Following [2], the dataset is divided into 85% for training and 15% for testing. All images are resized to 32×32. For CIFAR-10 and CIFAR-100, we use the predefined splits from the NAS-Bench-201 benchmark. For Aircraft and Oxford-IIIT Pets, we create random validation and test splits by dividing the test set into two equal-sized subsets.

Meta-dataset. In the training of dataset-aware predictor, we utilized triplet data in the form of (task, architecture, accuracy). In [11], the authors provided such a meta-dataset, which includes 4,230 task samples for the NB201 and 15,000 task samples for the MBV3. All task datasets are derived from ImageNet [4]. Each task consists of a dataset with 20 categories.

8.4. Training Pipeline for Neural Architecture

In Sec. 4.2 of the main paper, we obtain the optimal architectures under different datasets by using the predictor and performing the actual training on Top-K architectures. Therefore, in this section, we will provide a training pipeline for neural architectures under different search spaces.

NAS-Bench-201 and DARTS. In accordance with the training pipeline described by [5], each architecture in NAS-Bench-201 is trained using Stochastic Gradient Descent (SGD) with Nesterov momentum, optimizing with cross-entropy loss over 200 epochs. To regularize the training, a weight decay of 0.0005 is applied, and the learning rate is decreased from 0.1 to 0 following a cosine annealing schedule. Consistent hyperparameters are maintained across different datasets to ensure uniformity. Additionally, data augmentation techniques, such as random horizontal flipping (with a probability of 0.5), random cropping of 32×32 patches with 4 pixels of padding on each side, and normalization over RGB channels, are employed for all datasets.

MobileNetV3. To ensure a fair comparison, we use the same training pipeline as DiffusionNAG [2]. We fine-tuned a subset of a pre-trained supernet from the MobileNetV3 search space, initially trained on the ImageNet 1K dataset. This fine-tuning was conducted for CIFAR-10, CIFAR-100, Aircraft, and Oxford-IIIT Pets datasets, using the experimental setup from [3]. The procedure began by activating the subnet along with its parameters that were pre-trained on ImageNet 1K for each neural architecture. The fully connected layers for the classifier were then randomly initialized, ensuring that the output size matched the number of classes in the target dataset. Subsequently, fine-tuning on the target dataset was carried out for 20 epochs. The training settings were based on MetaD2A. We employed SGD with cross-entropy loss and set a weight decay of 0.0005. The learning rate was decreased from 0.1 to 0 using a cosine annealing schedule. Images were resized to 224×224 pixels, and data augmentation methods were used, including random horizontal flipping, Cutout with a length of 16, AutoAugment, and DropPath with a drop rate of 0.2.

8.5. Optimization Strategies in PNAG

In our approach, a single generation of the architectures by **PNAG** does not guarantee to satisfy the condition of maximizing the accuracy on the corresponding task, so it is necessary to use an optimization strategy to generate better architectures. Specifically, for each generated architecture, the optimization strategy is used to correct it. This process is repeated until the optimal architecture is generated. We provide two optimization strategies as shown in Sec. 4.2, one is to acquire samples by **Random** method and use the surrogate model to directly guide the generation of the architecture, and the other is to select the samples by an evolutionary algorithm (**EA**) combined with the surrogate model for the architecture generation. For the EA method, we only perform a mutation operation (add, delete, and alter structure) on the architecture.

8.6. Implementation Details of PNAG

In this section, we provide the model structures and the hyperparameters in the experiments, including the unconditional neural architecture generation model based on **PNAG**, the conditional neural architecture generation model, the dataset-aware neural predictor, and the model for task encoding. The specific hyperparameters are listed in Table 7.

Unconditional Neural Architecture Generation Model. The unconditional NAG model is based on the VAE and replaces the architecture generation part with an autoregressive model (*i.e.*, **PNAG**), as illustrated in Fig. 2(Stage 1). The process is self-supervised and aims to allow the model to learn the probability distribution of the architecture. Specifically, in the encoding part, we use the CATE model [17] (detailed in Sec. 7.1), and in the generation process, there is no network inference, and only the optimization of the codebook is involved. Then, the discrete structure is transformed into a continuous structure by **Embedding**. Finally, in the decoding part, we use two linear layers.

Conditional Neural Architecture Generation Model. The conditional NAG model is used to process the neural architecture after fusing the conditional information, intending to learn a more accurate representation of the architecture. Following [14], we adopt a standard decoder-only transformer similar to GPT-2 and VQGAN [7] to process the conditional input, replacing traditional layer normalization with adaptive normalization (AdaLN) [10]. So the transformer contains multiple AdaLN Self-Attention (AdaLNSelfAttn) layers.

Dataset-Aware Neural Predictor. The neural predictor includes the processing of the architecture, the processing of the task, and the prediction network. For architecture processing, we use the DiGCN model [15]. For the task, we get the encoded task vectors (detailed in the next part), which are then processed using a combination of "linear layer + Transfomer layer + linear layer". Finally, the information of the architecture and the task is fused, and the prediction results are obtained by a multilayer perceptron.

Model for Task Encoding. We used ResNet34 [8] as a probe network to encode the tasks.

9. Additional Experimental Results

In this section, we conduct two additional experiments to verify the influence of both the SCC strategy and the task encoding on generation.

9.1. Influence of SCC on Sub-Architecture

To further validate the advantages brought by the SCC strategy, we need to the validity of the sub-architecture. We selected 256 samples (one batch) for testing. The samples are chosen from the NAS-Bench-201 search space to verify whether the average validity of all the sub-architectures improves as the iteration progresses. As shown in Fig. 5, the average validity of the sub-architectures is effectively improved after the introduction of SCC, and at the end of the iteration, the average validity of all sub-architectures can be close to 100%, which proves that *the introduction of SCC strategy provides a stable generation path.*

Table 7. Hyperparameter settings under different models

Unconditional NAG model	
Hyperparameter	Value
The number of transformer blocks in CATE	12
The number of heads	8
Hidden Dimension of feed-forward layers	128
Hidden Dimension of transformer blocks	64
The Dimension of Codebook	7 in NB201, 9 in MBV3, 12 in DARTS
Dropout rate	0.1
Epoch	300
Learning Rate	0.0001
Batch size	256
Optimizer	Adam
Scheduler	CosineAnnealingLR
$\beta 1$ in Adam optimizer	0.9
$\beta 2$ in Adam optimizer	0.999
Conditional NAG model	
Hyperparameter	Value
The number of AdaLNSelfAttn	4
Hidden Dimension in AdaLNSelfAttn	256
Epoch	300
Learning Rate	0.001
Batch size	256
Optimizer	AdamW
Scheduler	CosineAnnealingWarmRestarts
$\beta 1$ in AdamW optimizer	0.9
$\beta 2$ in AdamW optimizer	0.999
Warmup Step	10
Neural Predicotr	
Hyperparameter	Value
The number of DiGCN layers	4
Hidden Dimension of DiGCN layers	144
Hidden Dimension of transformer blocks	1024
The head of transformer	8
Hidden Dimension of MLP layer	32
Dropout rate	0.1
Epoch	300
Learning Rate	0.0001
Batch size	256
Optimizer	AdamW
Scheduler	CosineAnnealingWarmRestarts
$\beta 1$ in AdamW optimizer	0.9
$\beta 2$ in AdamW optimizer	0.999
Warmup Step	10
Task Encoding	
Hyperparameter	Value
Epoch	60
Optimizer	Adam
Learning Rate	0.0004
Batch size	256
Weight Decay	0.0001
$\beta 1$ in Adam optimizer	0.9
$\beta 2$ in Adam optimizer	0.999

Table 8. Results of **PNAG** on CIFAR10 with and without task encoding.

Search Space	**Task Encoding**	**Accuracy** (%) on CIFAR10	**Improvement** (%)
NB201	without	94.08±0.29	-
NB201	with	94.37±0.00	0.29
MBV3	without	95.98±0.61	-
MBV3	with	97.67±0.05	1.69

9.2. Influence of Task Encoding

In this section, we assess the influence of task encoding on generalization to unseen tasks. We train **PNAG** with and without task encoding and test its performance on unseen tasks. The experiments are conducted on the CIFAR10 in two search spaces. As shown in Table 8, **PNAG** generated

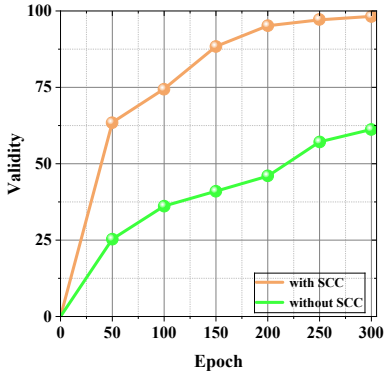

Figure 5. Influence of SCC strategies on the average validity of intermediate sub-architectures.

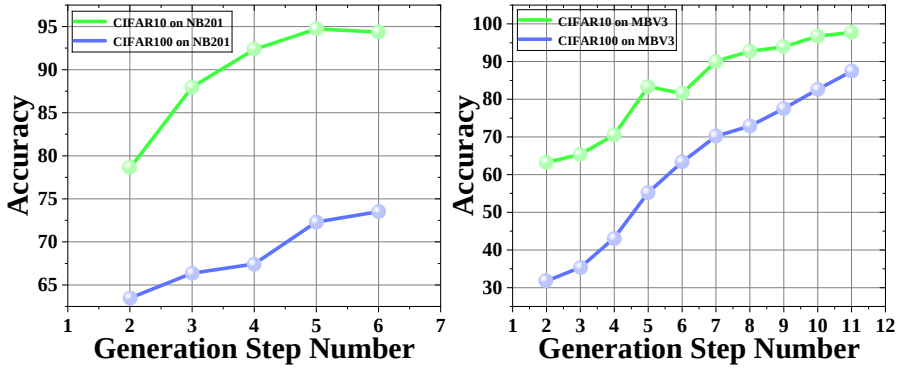

Figure 6. Results of the different number of generation steps on architectural performance.

Table 9. Results on TransNAS-Banch-101.

Method	Scene (ACC. %)	Object (ACC. %)	Jigsaw (ACC. %)	Layout (Loss $×10^{-2}$)	Segment. (mIoU %)
HNAS	54.29	44.08	94.56	-64.83	94.57
RoBoT	54.87	**45.59**	94.82	-61.16	94.58
EDNAG	54.89	45.52	94.71	-62.19	93.97
Arch-LLM	54.79	44.99	94.73	-61.39	94.34
PNAG	**54.91**	**45.59**	**94.86**	**-61.07**	**94.61**
optimal	54.94	45.59	95.37	-60.10	94.61

architectures of significantly higher quality when task encoding is applied, demonstrating that the *encoding strategy enhances the generalization capability of* ***PNAG*** *in conditional generation tasks.*

9.3. Influence of Generation Step Number

While the number of generation steps (equivalent to the number of sub-architectures) is predefined in our process (6 for NB201 and 11 for MBV3), its impact on the quality of generated architectures warrants investigation. Therefore, we conducted experiments with varying step numbers. As shown in Fig. 6, across different tasks and search spaces, the quality of architectures generated by **PNAG** consistently improves with an increasing number of steps. This demonstrates *a clear correlation between the number of generation steps and the quality of the resulting architectures.*

9.4. Expand to Other Tasks

To further validate the performance of the **PNAG** on other tasks, we conducted a comprehensive evaluation in the TransNAS-Banch-101 [6] search space (with a Micro size). This dataset includes not only traditional classification tasks but also jigsaw and segmentation tasks, better simulating the diverse applications encountered in the real world. For comprehensive comparison, we introduced several SOTA methods, including HNAS [13], RoBoT [9], EDNAG [19], and Arch-LLM [12].

The experimental results in Table 9 clearly demonstrate that the **PNAG** algorithm outperforms all other baseline methods across all tasks. Specifically, in the classification tasks, **PNAG** achieved accuracy rates of 54.91% for Scene and 45.59% for Object, outperforming most existing methods. In the Jigsaw task, **PNAG** achieved an accuracy of 94.86%, which is very close to the optimal result (95.37%), showcasing its strong capability in handling complex image structure tasks. In the Layout task, **PNAG**'s loss value was -61.07, showing a slight increase, but still exhibiting stronger robustness compared to other baseline methods. In the Segmentation task, **PNAG** achieved an mIoU of 94.61%, which is identical to the optimal result, demonstrating its outstanding performance in image segmentation tasks. Overall, **PNAG** demonstrated highly competitive performance in this multi-task evaluation, confirming its effectiveness and broad adaptability in complex application scenarios.

9.5. Results on Large Search Space

In this section, we apply **PNAG** to a larger search space, GraphNet [16], whose size is approximately 10^{390}. The implementation details are as follows: (1) Due to the vastness of the original search space, we sampled 400,000 architec-

Table 10. Results on GraphNet.

Method	Pretraining time	CIFAR10 Acc (%)	CIFAR10 Cost	CIFAR100 Acc (%)	CIFAR100 Cost	ImageNet Acc (%)	ImageNet Cost
Evo(T-CET)	-	96.4	22	79.6	22	77.6	20
HL-Evo(T-CET)+CCNF	5	97.4	3.6	80.9	3.6	**78.4**	6
SG+CCNF	30	97.6	0.08	81.0	0.08	78.5	0.08
PNAG	**4.28**	**98.02±0.21**	**0.01**	**82.17±0.34**	**0.01**	78.13±0.28	**0.01**

tures for the first-stage pre-training (as shown in Fig. 2 of the main text), which cost 4.28 GPU hours. (2) We recognize that the GraphNet search space differs from NAS-Bench-201 or MobileNet, where defining a cell does not determine the final architecture. Therefore, we construct the complete architecture by generating several cells (3 to 5 in our experiments) and then repeatedly stacking them; (3) As there is no available neural predictor to directly evaluate the performance of generated architectures on GraphNet for specific tasks, we adopted a zero-cost approach for architecture evaluation. (4) Following the same experimental protocol as in GraphNet, we selected the top 10 scoring architectures for retraining on CIFAR10, and the top 1 for ImageNet-1K. (5) Furthermore, to ensure a fair comparison, we constrained our generation to architectures with fewer than 1 million parameters on the CIFAR datasets and no more than 450M FLOPs on the ImageNet dataset. Our final results, averaged over three independent runs for each dataset, are presented below. As can be seen, our method achieves SOTA on the CIFAR datasets. On ImageNet, our best accuracy (78.41%) is slightly lower than that of SG+CCNF, but it is achieved at a lower computational cost. These results demonstrate the effectiveness of our approach in much larger search spaces.

References

[1] Alessandro Achille, Michael Lam, Rahul Tewari, Avinash Ravichandran, Subhransu Maji, Charless C Fowlkes, Stefano Soatto, and Pietro Perona. Task2vec: Task embedding for meta-learning. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6430–6439, 2019. 2

[2] Sohyun An, Hayeon Lee, Jaehyeong Jo, Seanie Lee, and Sung Ju Hwang. Diffusionnag: Predictor-guided neural architecture generation with diffusion models. In *The Twelfth International Conference on Learning Representations*. 5

[3] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*. 5

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 5

[5] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020. 5

[6] Yawen Duan, Xin Chen, Hang Xu, Zewei Chen, Xiaodan Liang, Tong Zhang, and Zhenguo Li. Transnas-bench-101: Improving transferability and generalizability of cross-task neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5251–5260, 2021. 7

[7] Patrick Esser, Robin Rombach, and Bjorn Ommer. Taming transformers for high-resolution image synthesis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12873–12883, 2021. 6

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 3, 6

[9] Zhenfeng He, Yao Shu, Zhongxiang Dai, and Bryan Kian Hsiang Low. Robustifying and boosting training-free neural architecture search. *arXiv preprint arXiv:2403.07591*, 2024. 6, 7

[10] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4401–4410, 2019. 6

[11] Hayeon Lee, Eunyoung Hyung, and Sung Ju Hwang. Rapid neural architecture search by learning to generate graphs from datasets. *arXiv preprint arXiv:2107.00860*, 2021. 5

[12] Deshani Geethika Poddenige, Sachith Seneviratne, Damith Senanayake, Mahesan Niranjan, PN Suganthan, and Saman Halgamuge. Arch-llm: Taming llms for neural architecture generation via unsupervised discrete representation learning. *arXiv preprint arXiv:2503.22063*, 2025. 7

[13] Yao Shu, Zhongxiang Dai, Zhaoxuan Wu, and Bryan Kian Hsiang Low. Unifying and boosting gradient-based training-free neural architecture search. *Advances in neural information processing systems*, 35:33001–33015, 2022. 7

[14] Keyu Tian, Yi Jiang, Zehuan Yuan, Bingyue Peng, and Liwei Wang. Visual autoregressive modeling: Scalable image generation via next-scale prediction. *arXiv preprint arXiv:2404.02905*, 2024. 6

[15] Wei Wen, Hanxiao Liu, Yiran Chen, Hai Li, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. In *European Conference on Computer Vision*, pages 660–676. Springer, 2020. 6

[16] Lichuan Xiang, Łukasz Dudziak, Mohamed S Abdelfattah, Abhinav Mehrotra, Nicholas Donald Lane, and Hongkai Wen. Towards neural architecture search through hierarchical generative modeling. In *Forty-first International Conference on Machine Learning*, 2024. 7

[17] Shen Yan, Kaiqiang Song, Fei Liu, and Mi Zhang. Cate: Computation-aware neural architecture encoding with transformers. In *International Conference on Machine Learning*, pages 11670–11681. PMLR, 2021. 2, 6

[18] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International conference on machine learning*, pages 7105–7114. PMLR, 2019. 2

[19] Bingye Zhou and Caiyang Yu. Evolution meets diffusion: Efficient neural architecture generation. *arXiv preprint arXiv:2504.17827*, 2025. 7