Neurocomputing xxx (xxxx) xxx

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

A neural architecture generator for efficient search space

Kun Jing, Jungang Xu *, Zhen Zhang

School of Computer Science and Technology, University of Chinese Academy of Sciences, Huaibei Town, Huairou District, Beijing 101408, China

ARTICLE INFO

Article history:
Received 26 June 2021
Revised 2 September 2021
Accepted 29 October 2021
Available online xxxx
Communicated by Zidong Wang

Keywords:
Neural architecture search
Large-scale architecture space
Generative adversarial network
Neural architecture generator
Graph neural network

ABSTRACT

Neural architecture search (NAS) has made significant progress in recent years. However, the existing methods usually search architectures in a small-scale, well-designed architecture space, discover only one architecture in a single search, and hardly rework, which severely limits their potential. In this paper, we propose a novel neural architecture generator (NAG) that can efficiently sample architectures in a large-scale architecture space. Like a generative adversarial network (GAN), our model consists of two components: (1) a generator that can generate directed acyclic graphs (DAGs) as cells or blocks of neural architectures and (2) a discriminator that can estimate the probability that a DAG comes from cells of real architectures rather than the generator. Furthermore, we employ a random search with NAG (RS-NAG) to discover the optimal architecture according to the customized requirements. Experimental results show that the NAG can generate diverse architectures with our customized requirements multiple times after one adversary training. Furthermore, compared with the existing methods, our RS-NAG achieves the competitive results with 2.50% and 25.5% top-1 accuracies on two benchmark datasets – CIFAR-10 and ImageNet.

1. Introduction

The design of neural architecture is the driving force of deep learning and its applications, which requires much time and expert knowledge. Recently, neural architecture search (NAS) has been proposed and developed to design neural architecture automatically. Much research [1–8] has shown that finding a superior architecture using reinforcement learning (RL) [1–4] and evolutionary algorithm (EA) [5–8] is feasible in a given architecture space. Furthermore, some differentiable methods [9–13], such as differentiable architecture search (DARTS) [9] and neural architecture optimization (NAO) [13], have been proved to be more efficient.

However, because these methods do not sufficiently utilize architecture access, they have to search or optimize the architectures on the manually designed subset of the whole architecture space. Although the elaborate design of the architecture space based on expert knowledge can significantly reduce the size of architecture space, the reduction severely limits their potential, i.e., they may miss better architectures that are not in the designed subset. Besides, they are ineffective, as they can discover only one optimal architecture in one search and need to take the more expensive costs to search for another architecture that meets another requirement.

To address these problems, we propose a novel neural architecture generator (NAG) to efficiently sample architectures in a large-scale architecture space. In this paper, we employ the framework of the generative adversarial network (GAN) to train the NAG because the quality of the generated architectures cannot be directly measured. Since the training process involves graph generation and discrimination, we utilize the graph neural network (GNN) to extract the spatial features of graphs for finer-grained graph modeling, seeing Section 3.2.1 and 3.2.2 for details. With the help of GAN and GNN, we expect the NAG to capture or learn the distribution or network topology characteristics of these architectures that meet the given requirements. The overview of the NAG training is briefly shown in Fig. 1, which is consistent with the GAN training. Our model is implemented by a special GAN, which consists of two components: 1) a generator (i.e., NAG) mainly implemented by an improved graph recurrent neural network (GraphRNN) in our proposal, which can dynamically generate various directed acyclic graphs (DAGs, i.e., architectures) with random-length nodes and random edges as architectures according to noises, and 2) a discriminator implemented by a graph convolutional network (GCN), which can distinguish whether a DAG comes from the NAS dataset or the NAG. If labels are introduced in the training process, the NAG can generate DAGs with the customized requirements. It is noted that the NAG can theoretically

* Corresponding author.
E-mail addresses: jingkun18@mails.ucas.ac.cn (K. Jing), xujg@ucas.ac.cn (J. Xu), zzhang586@wisc.edu (Z. Zhang).

https://doi.org/10.1016/j.neucom.2021.10.118
0925-2312/

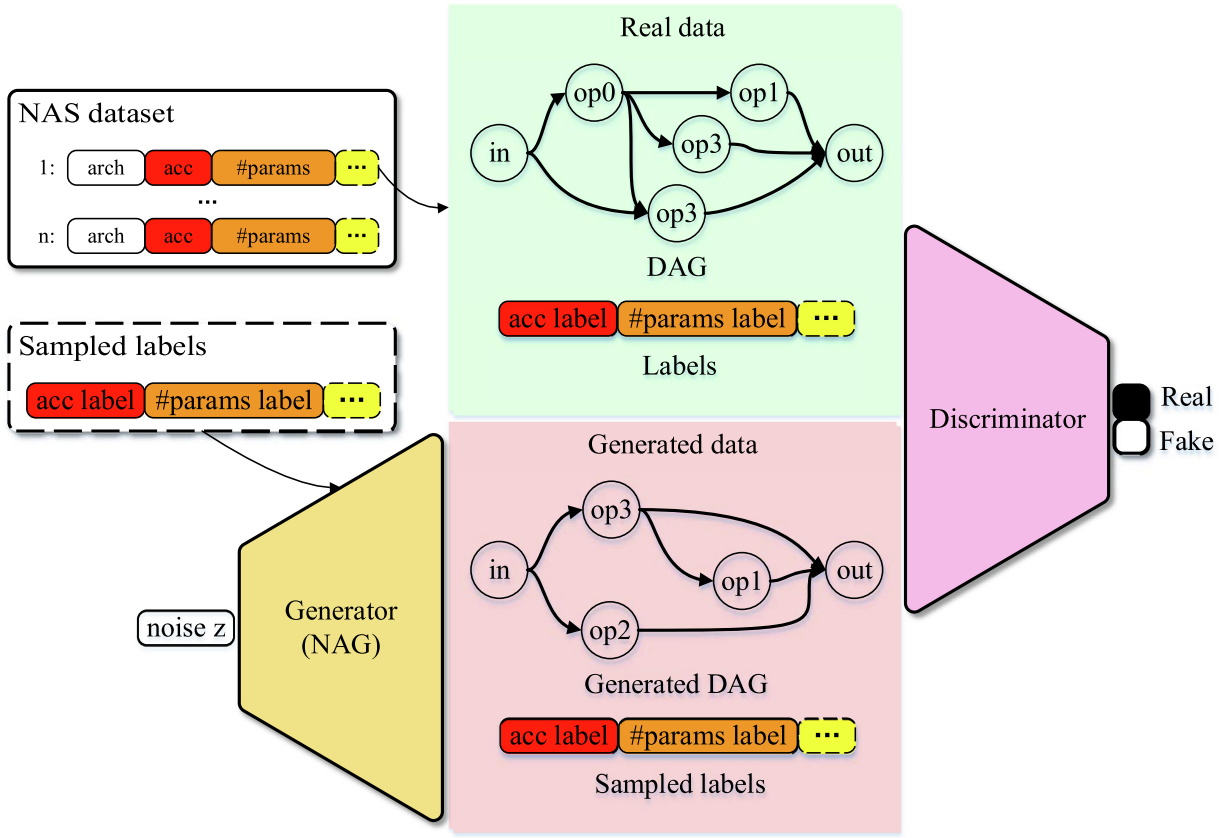


Fig. 1. The overview of the NAG training. The framework is inspired by the vanilla conditional generative adversarial network. The DAGs and labels are sampled from the NAS dataset. The sampled labels are random in the training stage and manually set in the inference stage. The NAG outputs the generated DAGs according to the noises $z \sim p_z(z)$.

generate any DAGs on the condition that the NAS dataset includes various DAGs with different numbers of nodes and connections. Once the NAG is trained, it can generate architectures multiple times. Like human experts, the NAG can automatically design architectures with the customized requirements according to prior architecture knowledge (i.e., the NAS dataset), which is a trial and error process. Due to adversarial training, NAG can explore a more extensive search space with fewer permutations and combinations. In other words, it can easily explore in the right and fruitful directions. To make full use of the advantages of NAG, we expand the original architecture space in DARTS space [9]. Furthermore, we employ a random search with NAG (RS-NAG) to discover the optimal architecture for image processing according to the customized requirements.

To compare our proposed method with other NAS methods, we build a NAS dataset on our architecture space using a one-shot model and improved parameter sharing for the GAN training. We report the competitive experimental results of RS-NAG on two benchmark datasets, namely CIFAR-10 [14] and ImageNet [15]. Moreover, ablation studies and more experiments further verify the effectiveness of the NAG.

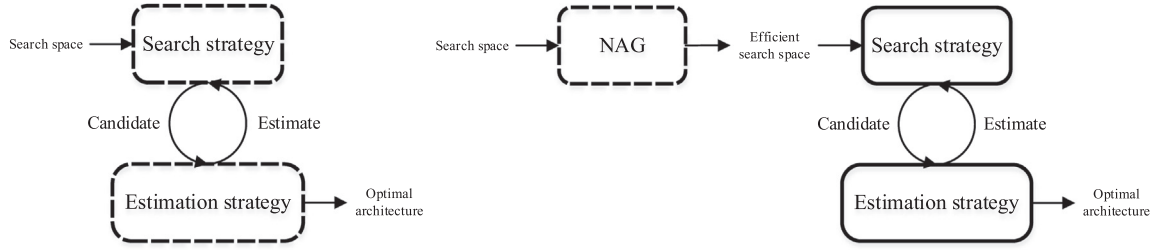
Our contributions are as follows:

- We propose an extended architecture space and a novel NAG. After adversarial training on the NAS dataset built on the target dataset, the NAG can generate various architectures according to the customized requirements. Moreover, the NAG can be trained once and generate architectures multiple times.
- We can enable the extensible NAG to generate architectures that meet the customized requirements by introducing other performance indicator labels, such as inference delay, etc.
- We also propose two improvement techniques, i.e., the probability graph and the improved parameters sharing. The probability graph guarantees gradient flow and more robust training. The improved parameter sharing realizes the parameter sharing between cells with different numbers of nodes and improves the efficiency of the NAS dataset building.
- Extensive experiments on the image classification task verify the effectiveness of the NAG. RS-NAG can efficiently discover the optimal architecture that meets the customized requirements. We show the architecture that achieves highly competitive results with 2.50% test error on CIFAR-10 and 25.5% top-1 error on ImageNet.

2. Related work

2.1. Neural architecture search

The crucial components of the existing NAS methods consist of search strategy and estimation strategy. The search strategies of NAS are roughly divided into three categories, RL-based methods [1–3,16], EA-based methods [5,7,8], and differentiable methods [9,13,12,11,10]. Unlike RL-based [1] and EA-based methods [5] that are conducted over a discrete and non-differentiable architecture space, DARTS [9] relaxes the space into a continuous space. It uses gradient information for a more efficient search. The estimation strategies of NAS are to improve the efficiency of network estimation, including brute-force, training proxy [4,8,17], performance predictor [16,18], parameter sharing [3,9], and lightweight performance proxy [19,20]. All existing NAS methods search for only one architecture in an elaborately designed architecture space (e.g., DARTS requires a fixed number of nodes and edges). As shown in



(a) Conventional NAS framework. (b) Our NAS framework.

Fig. 2. The comparison of conventional NAS frameworks with ours. The dotted box is the component where a NAS work focuses and contributes. The solid box is the component where a NAS work does not care about. Compared with the existing NAS methods, our contribution lies in how to get an efficient search space.

Fig. 2, our NAS framework also includes the NAG that can build an efficient search space. In the efficient search space, we can get competitive architectures using random search. Our NAS method has three advantages over the existing one: NAG can generate various architectures with different customized requirements instead of searching for the optimal architecture; RS-NAG can search for the optimal architecture from the efficient search space generated by NAG; for the architecture space, the number of nodes and edges in each DAG (cell) can be variable. To our knowledge, our work may be the first attempt to introduce a neural architecture generator to get the customized architectures, which can build an efficient search space.

2.2. Generative adversarial network

GAN is a significant model for image generation. In the GAN [21], the generator and discriminator are trained alternately to realize adversarial learning, where the process is a zero-sum game and can converge to Nash equilibrium. In order to remedy the two defects of the low quality of the generated data and the instability of the training process, least-squares GAN (LSGAN) [22] replaces the cross-entropy loss with the least square loss. Besides, there are many works [23–26] to improve the GAN training. Furthermore, conditional GAN (CGAN) [27] introduces some labels to guide the generation process and enables the generator to output data in the specified style. For the efficient search space, we obtain a NAG by adversarial training in our proposed special GAN, which can capture the distributions of the architectures that meet the different customized requirements and can efficiently explore the architecture space.

2.3. Graph neural network

GNN [28] is a deep neural network that propagates messages along the edges between the nodes on graphs to deal with graph problems. GCN [29] is one of the most significant layer-wise GNN models, which can extract spatial features of graphs. Graph generation is one of the most concerned graph problems. GraphRNN [30] can dynamically generate a graph in an autoregressive way. A neural architecture is viewed as a DAG in our work, and we utilize GNN to tackle graph information. GCN is used as the discriminator to obtain graph features. The decoder of VS-GAE [31] inspires our generator. The introduction of GNN makes the architecture properties modeled better on a fine-grained level.

3. Neural architecture generator

In theory, our method can be used for various tasks, e.g., image processing and natural language processing. In this paper, we only focus on the image classification task, which is the basic task of image processing.

3.1. Large-scale architecture space

3.1.1. Macro architecture

Like the previous works [3,9,13], our architecture space is also based on cells. We stack each normal cell N_{cell} times, followed by a reduction cell, in which the image height and width are halved via max-pooling, and the number of channels is doubled. We repeat this pattern N_{stack} times, followed by a global average pooling and a final dense softmax layer. The initial layer of the model is a stem consisting of one 3×3 convolution with 128 output channels. The macro architecture is shown in Fig. 3a.

3.1.2. Micro architecture

To facilitate the use of GNN, each cell, also named micro architecture, is considered as a DAG, in which nodes represent any candidate operations and edges represent tensor flows. In the DAG, except for input and output nodes, there are other candidate operation types of nodes, e.g., separable convolutions, dilated separable convolutions, and pooling operations. An example of a micro architecture is shown in Fig. 3b. The graph with five nodes is described by a 5×5 adjacency matrix and a 5×7 feature matrix. The element at i-th row, j-th column of the adjacency matrix indicates a directed edge from source node i to target node j. In addition to the first operation "in" and the last operation "out", other operations are one of five candidate operations according to their one-hot vectors in the feature matrix. To avoid human bias and explore more architectures, we expand the micro architecture space to the space that includes all possible DAGs with less than N nodes.

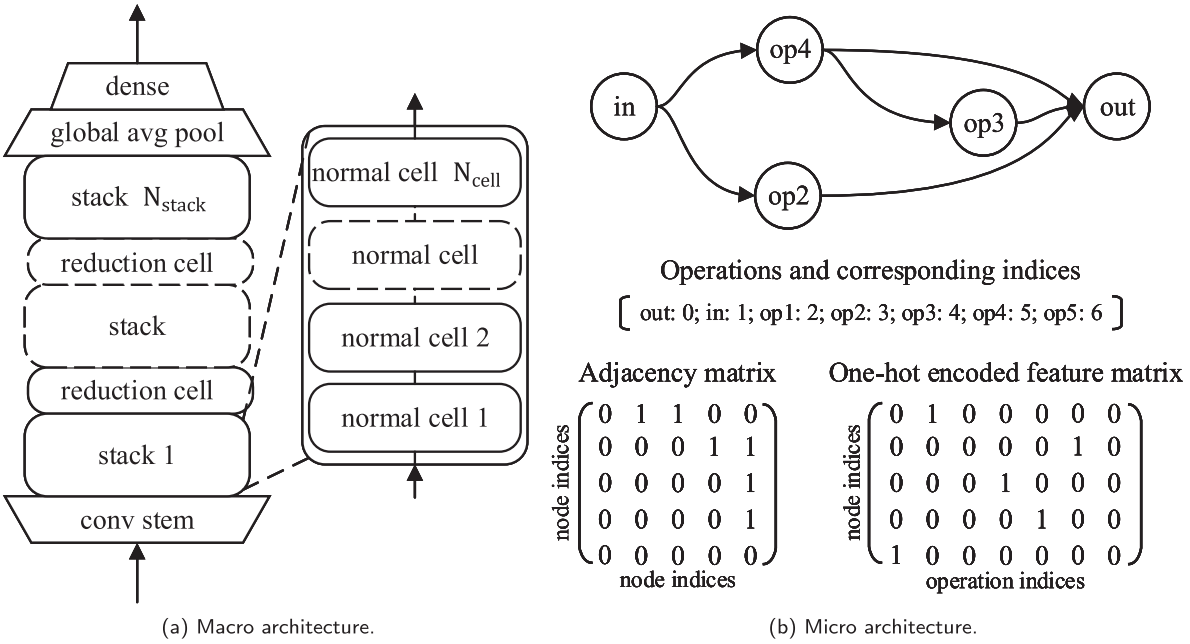
As a result, the size of the architecture space increases from $n_{op}^{2\times n_{node}} \cdot \prod_{i=2}^{n_{node}+1} \binom{i}{2}$ to $n\prime_{op}^{n\prime_{node}} \cdot 2^{(n\prime_{node}+1)(n\prime_{node}+2)/2}$, where $n_{op}, n\prime_{op}$ are the number of the candidate operations of the original and our space, and $n_{node}, n\prime_{node}$ are the number of intermediate nodes in the DAGs. Normally, $n\prime_{node} = 2 \times n_{node}$ and $n\prime_{op} = n_{op}$.

3.2. Components

As shown in Fig. 1, we use the framework of **the vanilla CGAN** to control the NAG to generate the architecture under certain conditions, where there are two significant components: a generator and a discriminator. The generator (i.e. NAG) is mainly implemented by an improved **GraphRNN**, which can dynamically generate various DAGs (i.e., architectures) with random-length nodes and random edges. The discriminator is implemented by a **GCN**, which can distinguish whether a DAG comes from the NAS dataset or the NAG.

3.2.1. Generator

We expect that the NAG can take labels of performance indicators of interest and a noise z sampled from noise prior distribution



(a) Macro architecture. (b) Micro architecture.

Fig. 3. The architecture space of NAG. (left) The macro architecture of the discovered architecture. (top-right) An example cell. (bottom-right) The matrix representation of the top-right cell.

$p_z(z)$ as input and map them into a DAG G with n nodes, which is formulated as

$$G = f_{generator}(c), \quad c = (z, E_1[l_1], E_2[l_2], \cdots), \quad z \sim p_z(z), \tag{1}$$

where E_i denotes the embedding layer of the i-th label l_i.

Inspired by the variational-sequential graph autoencoder [31], we use an improved **GraphRNN** to construct the NAG, which sequentially generates new nodes with a determined candidate operation and new input edges of the new node. The NAG consists of four modules, i.e., **PropGraph**, **AddNode**, **InitNode**, and **AddEdge**, which are illustrated in Fig. 4. Starting from the graph with only one input node, we repeat these four modules until one output node is generated or the number of nodes is up to the maximum we set.

The **PropGraph** module propagates messages on the current graph $G^{(t)}$, updates the current embeddings $H^{(t)}$ of the existing nodes, and aggregates the updated node embeddings $H^{(t)}_{updated}$ for the graph embedding $h_{G^{(t)}}$. The graph embedding $h_{G^{(t)}}$ is a current graph state used to generate new nodes and edges. This is formulated as

$$h_{G^{(t)}}, H^{(t)}_{updated} = f_{prop}\left(H^{(t)}, G^{(t)}\right). \tag{2}$$

The function f_{prop} can be specifically written as

$$\begin{aligned} H^{(t)}_{updated,v} &= U^{(t)}\left(H^{(t)}_v, msg_v\right), \ v \in G^{(t)}, \quad msg_v \\ &= A\left(M^{(t)}\left(H^{(t)}_v, H^{(t)}_u\right)\right), \quad u \in N(v), \quad h_{G^{(t)}} \\ &= A_G\left(H^{(t)}_{updated}\right), \end{aligned} \tag{3}$$

where A and A_G are aggregation functions for messages and the graph embedding $h_{G^{(t)}}$ respectively; $N(v)$ denotes the neighbours of node v; $M^{(t)}$ is a function for messages between nodes; $U^{(t)}$ is an update function. It is noted that directed edges of DAGs are regarded as bi-directed edges during message propagation, which means that message propagation is bi-directed.

According to the state $h_{G^{(t)}}$ of the original graph, and the concatenation c of the noise and the label embeddings, the **AddNode** module determines the type of the candidate operation of a new node,

$$P_{node_{t+1}} = softmax(f_{addNode}(c, h_{G^{(t)}})), \tag{4}$$

where $P_{node_{t+1}}$ denotes the probability distribution for candidate operations of the $(t+1)$-th node that cannot be an input node.

The goal of the **InitNode** module is to initialize the embedding of the new node. Specifically,

$$h_{t+1} = f_{initNode}(c, h_{G^{(t)}}, E_{node}[argmax(P_{node_{t+1}})]), \tag{5}$$

where the inputs include the concatenation c, the state $h_{G^{(t)}}$ of the graph, and the embedding $E_{node}[argmax(P_{node_{t+1}})]$ of the $(t+1)$-th node. Then, the node embeddings $H^{(t+1)} = \left(H^{(t)}_{updated}, h_{t+1}\right)$ of the graph in next time step are obtained.

After a new node is created, the **AddEdge** module determines which edges between old and new nodes are selected, which depends on the embeddings $h_v \in H^{(t)}_{updated}$ of the old node, the embeddings h_{t+1} of the new node, the graph state $h_{G^{(t)}}$, and the concatenation c.

$$P_{edges_{t+1},v} = \sigma(f_{addEdge}(h_v, h_{t+1}, c, h_{G^{(t)}})), \tag{6}$$

where σ is the sigmoid function; the edge $v \to (t+1)$ is added to the graph if the probability $P_{edges_{t+1},v}$ of the edge $v \to (t+1)$ is greater than 0.5.

Finally, the new graph $G^{(t+1)}$ is created according to the old graph $G^{(t)}$, $P_{node_{t+1}}$, and $P_{edges_{t+1}}$.

3.2.2. Discriminator

The discriminator aims to estimate the probability of a graph coming from the real data, i.e., the NAS dataset. As shown in Fig. 4, the discriminator takes a graph and performance indicator labels of interest and then outputs a Bernoulli distribution.

To utilize graph information better, we use an L-layers **GCN** to extract graph embedding. Then, the last dense layers are used to

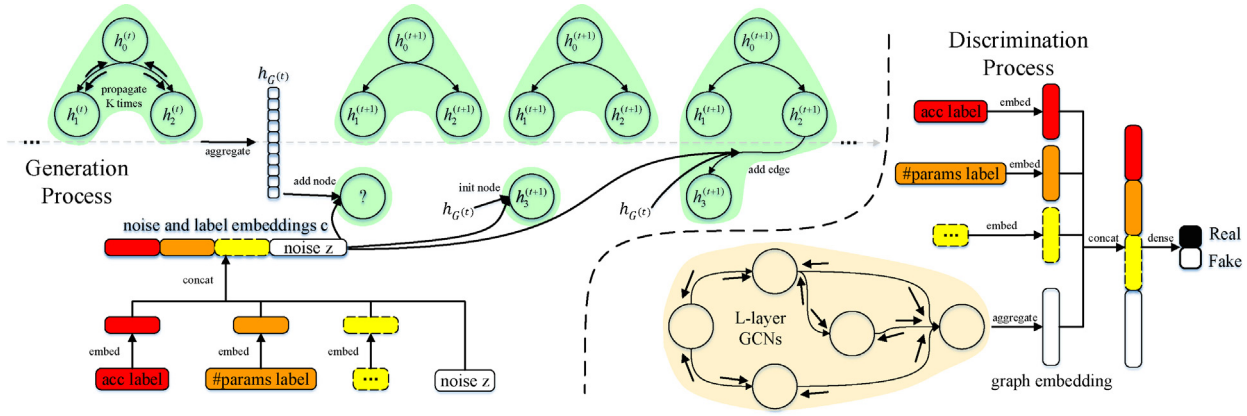


Fig. 4. The illustration of a single iteration of the graph generation process (left) and graph discrimination process (right). The generator generates the architecture (DAG) according to the random noise z and the input labels through **PropGraph**, **AddNode**, **InitNode**, and **AddEdge** modules. The discriminator takes the architecture (DAG) and the labels as input, encodes the architecture, and makes a binary classification to forecast whether they match or not.

map the embedding of a graph and its labels to the Bernoulli distribution that indicates whether it is real data.

$$\hat{y} = f_{discriminator}(G) = f_{MLP}(h_G, E_1[l_1], E_2[l_2], \cdots), \quad h_G = A_G\left(H^{(L)}\right), \tag{7}$$

$$H^{(l+1)} = ReLU\left(\widehat{A}H^{(t)}W^{(l)}\right), \quad \widehat{A} = \widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}, \tag{8}$$

$$H_v^{(0)} = E_{node}[argmax(P_{node_v})], \tag{9}$$

where $\widetilde{A} = A + A^T + I$ means bi-directed message propagation in GCN; A is an adjacency matrix of the input graph G; I is an identity matrix and $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$ is a degree matrix; $W^{(l)}$ is the trainable parameters of l-th layer of GCN; $argmax(P_{node_v})$ is a candidate operation of node v. It is noted that the embedding layers E_{node}, E_i and the graph aggregation function A_G are different from those in the NAG.

3.3. NAS Dataset Building with Improved Parameter Sharing

As shown in Fig. 1, the NAS dataset is the key to training the NAG. A simple way to obtain NAS datasets is to build it directly from the NAS benchmarks [32–34] or the NAS works that report architectures and their performance indicators. Another way is to build the NAS dataset on the target dataset. We randomly sample some architectures in our proposed architecture space and then train and evaluate them on the target dataset to acquire the values of a set of performance indicators (e.g., accuracy, number of parameters, FLOPs) that we care about.

Unfortunately, the training and evaluation of the latter are incredibly time-consuming, which causes a bottleneck. Recently, many studies [3,13,9] have demonstrated that parameter sharing can significantly reduce the computational complexity of evaluating child architectures. Following NAO [13], we train and evaluate each sampled child architecture in a one-shot agent model with parameter sharing. However, parameter sharing between child models with different numbers of nodes is difficult. To solve this problem, we propose an improved parameter-sharing method with a simple minimum priority principle, i.e., a child model with n nodes inherits the parameters of the first n nodes in the parent one-shot network. Our method is based on the prior knowledge that the same first n nodes of different architectures tend to have consistent outputs. In this way, we can get a look-up table, each of whose items is composed of architecture and the values of its several performance indicators.

Then, we put several performance indicator labels on each architecture in the NAS dataset according to relative performance indicators rankings instead of by exact performance values. For example, all architectures in the NAS dataset are ranked according to their accuracies from high to low for the accuracy label. Moreover, the first, second, and last 1/3 architectures are labeled as *high-accuracy*, *middle-accuracy*, and *low-accuracy* architectures. Given the architectures, the relative rankings of the performance indicators that only depend on the architecture (e.g., number of parameters, FLOPs) are definite. Many NAS studies [13,35] have proven that the relative ranking of the accuracy of the child architectures in the one-shot model and the real architectures tend to be positively correlated. Finally, we obtain the NAS dataset, each of whose items is composed of architecture and its multiple performance labels.

3.4. Training and inference

To make the training more stable, we empirically use the least square loss function like LSGAN [22], where $a = 0$ and $b = c = 1$ (i.e., the 0–1 binary coding scheme). Considering the introduction of labels that improve the training, we calculate the stochastic gradients of discriminator and generator, respectively

$$\nabla_{\theta_D}\left[\frac{1}{2m}\sum_{i=1}^{m}\left(D(\alpha^{(i)}|l_{\alpha^{(i)}}) - 1\right)^2 + \frac{1}{2m}\sum_{i=1}^{m}\left(D(G(z^{(i)}|l_{z^{(i)}})|l_{z^{(i)}})\right)^2\right], \tag{10}$$

$$\nabla_{\theta_G}\frac{1}{2m}\sum_{i=1}^{m}\left(D(G(z^{(i)}|l_{z^{(i)}})|l_{z^{(i)}}) - 1\right)^2, \tag{11}$$

where $\alpha^{(i)}$ is a architecture sampled from NAS dataset; $l_{\alpha^{(i)}}$ denotes the label of the architecture $\alpha^{(i)}$; $z^{(i)}$ is a noise sampled from noise prior distribution $p_z(z)$; $l_{z^{(i)}}$ is a random label of $z^{(i)}$. The first term attempts to make the discriminator distinguish the real and generated architectures as much as possible. The second term tries to make the generator cheat the discriminator. The optimizations of these two gradients are executed alternately.

After convergence of the training, the NAG learnes the mapping from latent space to architecture space. Given an expected label and some latent space samples, we can obtain different neural architectures with the customized requirements. For example, we set *high-accuracy and low-FLOPs* as the input label, and then the NAG can generate different architectures with high accuracy and low delay. To make the architectures meet the customized requirements more strictly, we evaluate a few randomly sampled architectures and pick up the top-N_{need} architectures.

The detailed algorithm is shown in Algorithm 1.

Algorithm 1: Random search with Neural Architecture Generation.

Input: Number of performance indicators of interest N_f, Number of sampled architectures N_s, Number of architecture needed N_{need}.

Build NAS Dataset $\mathscr{D} = \left(\alpha, l_\alpha^{(1)}, \cdots, l_\alpha^{(N_f)}\right)$ in the way described in Section 3.3, where α is an architecture in the randomly sampled architecture pool A and the indicator label $L_\alpha^{(i)}$ is i-th indicators label of architecture α.

while not converge **do**

Sample minibatch of m noises $\{z^{(i)}\}$ and labels $\{l_{z^{(i)}}\}$ by random sampling.

Sample minibatch of m archs $\{\alpha^{(i)}\}$ and labels $\{l_{\alpha^{(i)}}\}$ from NAS dataset $\mathscr{D}$.

Update discriminator using Formula 10.

Sample minibatch of m noises $\{z^{(i)}\}$ and labels $\{l_{z^{(i)}}\}$ by random sampling.

Update generator using Formula 11.

end while

Randomly sample N_s architectures with expected labels from generator G. Train them from scratch and evaluate them to obtain the values of indicators.

Output: Top-N_{need} architectures that meet the customized requirements.

3.5. Gradient flow from discriminator to generator using probability graph

During the training of our model, the generator is updated by Formula 11. However, the generated DAG $G\left(z^{(i)}|l^{(i)}\right)$ is discrete, which breaks the gradient flow from the discriminator to the generator. Meanwhile, the process of looking up the embedding table for node embedding is non-differentiable. According to the above observations, we propose a probability graph trick to ensure that the gradient flows from the discriminator to the generator. During the training, the generator generates continuous probability values of nodes and edges, which avoids the discrete representation of graphs. Meanwhile, we replace the embedding layer in Formula 9 with a linear transformation,

$$H_v^{(0)} = P_{node_v} \cdot W_{embedding}, \tag{12}$$

where P_{node_v} is obtained by Formula 4, and $W_{embedding}$ is the weight of the embedding layer. Furthermore, the element A_{ij} of the adjacency matrix in Formula 8 denotes the probability of the edge between node i and node j. Besides, this improvement is also conducive to the robust adversarial training of NAG in two aspects. Specifically, this improvement introduces noises into the probability values of generator outputs. The noises can make the model more robust, which is proved by the previous GAN work [36]. This improvement also prevents the sparse gradients that are produced when NAG is trained without the probability graph trick and disturbs the training. In Section 4.4, the ablation study demonstrates the effectiveness of the trick of probability graph.

4. Experiments

In this section, we conduct RS-NAG experiments on two image classification benchmark datasets – CIFAR-10 [14] and ImageNet [15], respectively (searching on CIFAR-10 and evaluating on these two datasets). Because of the different architecture representations from previous works [1,13], we adopt the architecture space similar to those works as possible for a fair comparison. Finally, we perform a series of ablation studies. In the supplementary, we conduct and discuss more experiments on NAS-Bench-101 for verifying NAG.

4.1. Implementation details

4.1.1. Architecture space

We use five types of candidate operations in our space, including identity, 3×3 separable convolution, 5×5 separable convolution, 3×3 average pooling, and 3×3 max pooling. For a fair comparison with the existing methods, the maximum number N of nodes in each cell is 10, which is similar to other existing methods that have two branches in each node and a total of five nodes. We stack normal cells $N_{cell} = 6$ times in each stage and still repeat the stage $N_{stack} = 3$ times.

4.1.2. NAS dataset

Considering the cost of training and the convergence of the child architectures, we empirically set the same number (i.e., 4000) of the sampled architectures as NAO [13]. To build a NAS dataset on CIFAR-10, we trained 4000 sampled architectures in a one-shot model with improved parameter sharing. To accelerate the NAS dataset building, we train each child model of the one-shot model with three layers of cells in each stack and 20 channels. We randomly choose 5000 images from the training set as the validation set. Standard data pre-processing and augmentation, such as whitening, randomly cropping 32×32 patches from unsampled images of size 40×40, and randomly and horizontally flipping images, are applied to the original training set. The label smoothing of 0.1 is also used. The one-shot model is trained using SGD with a momentum of 0.9, where the arrangement of learning rate follows a single period cosine schedule with $l_{max} = 0.025$. We apply a stochastic drop-connection with a keep rate of 0.9 on each path and an l2 weight decay of 3e-4 for regularization. All the models are trained with the batch size of 96 and 150 epochs. The settings are the same as NAO [13]. The NAS dataset building takes less than 1 GPU day on a single NVIDIA V100 GPU card.

4.1.3. Accuracy-optimal search on CIFAR-10

Firstly, we conduct an accuracy-optimal search on CIFAR-10. After the adversarial training, we sample 10 architectures from the NAG and evaluate them following the setting of the NAS dataset building to get the top-1 architecture for high accuracy on the validation set. The settings of training NAG are as follows. The sizes of node embedding and graph embedding are set to 250 and 56 for generator and discriminator, respectively. The sizes of label embedding and noise are set to 10 and 100, respectively. To train NAG, we optimize the discriminator D and generator G alternately (update D one step, and then update G one step) using Adam optimizer for 200 epochs with the batch size of 64, the learning rate of 0.0002, and the dropout rate of 0.5. We flip the real and fake labels [21] to avoid early gradient vanishing of the generator.

4.1.4. Pareto-optimal front search on CIFAR-10

The accuracy-optimal search is a particular case of the Pareto-optimal front search, in which we only care about the validation accuracy. NAG can also generate some architectures with different numbers of parameters. Furthermore, we conduct a Pareto-optimal front search on CIFAR-10. Except for a newly added label of the number of parameters, other settings are the same as the accuracy-optimal search on CIFAR-10. We randomly sample three

architectures from the NAG and report the best one. The settings of architecture evaluation are exactly the same as those of the accuracy-optimal search on CIFAR-10. Another approach to gaining a model with high accuracy is to combine three architectures with the label of few parameters and high accuracy into an ensemble model by hard voting.

4.2. Results on CIFAR-10

The normal and reduction cells of the best architecture we discovered on CIFAR-10 are shown in Fig. 5. We train the discovered model (6 layers of cells each stack and 36 channels) with 600 epochs from scratch on CIFAR-10. We apply stochastic drop-connection with a keep rate of 0.8 and dropout layer with a keep rate of 0.6. Other settings are the same as the NAS dataset building.

The results on CIFAR-10 are shown in Table 1. Compared with architectures discovered by other methods, RS-NAG can discover a competitive architecture with 2.50% test error in 10 sampled architectures by the accuracy-optimal search. As can be seen, according to the customized requirements, NAG can also design architectures with different numbers of parameters and acceptable accuracy. Among them, the architectures with a medium number of parameters perform high accuracy, consistent with SNAS [11]. The reason may be that the same hyper-parameters of training networks are used for models with different sizes, which results in their under-fitting or over-fitting. It is worth noting that the results of the accuracy-optimal and Pareto-optimal front search are not comparable because of their different numbers N_s of the sampled architectures. The former samples ten architectures, while the latter samples three architectures. Besides, the ensemble model does not perform as well as expected. The experimental results prove that our RS-NAG method can discover competitive architectures in the efficient search space built by our NAG.

4.3. Transferring the discovered architecture to ImageNet

Furthermore, we transfer the architecture discovered on CIFAR-10 to ImageNet. We follow the same training settings used in the related work [13]. All results are listed in Table 2. The experimental results show that the discovered architecture on CIFAR-10 is indeed transferable to more complicated tasks, which is superior to or close to manually designed networks and is third only to PC-DARTS and P-DARTS.

4.4. Ablation study

We use the same settings used in the experiments on CIFAR-10 for ablation studies on NAS-Bench-101. The experimental results of all ablation studies are shown in Fig. 6.

4.4.1. The improvement of probability graph

To empirically prove the improvement of the probability graph, we compare it with a simple engineering trick[1]. Differing from the normal convergence of loss of NAG (marked with violet solid line), the discriminator loss of NAG without the probability graph (marked with blue solid line) converges to 0, and its generator loss does not converge (i.e., its generator learns nothings from real distribution). This result demonstrates that the improvement of the probability graph is beneficial to the training convergence.

[1] The trick $y\_hard - y\_soft.detach() + y\_soft$ is utilized in the Pytorch library. It achieves two effects: it makes the output value exactly one-hot (since we add and then subtract *y_soft* value); it makes the gradient equal to *y_soft* gradient (since we strip all other gradients)

4.4.2. The improvement of label flipping

Observing that the gradient of the generator tends to disappear in the early stage, we explore the improvement of label flipping by ablation study. For NAG without the label flipping (marked with orange solid line), its loss begins to converge after an early divergence and a hop, which confirms our observation. Thus, label flipping can improve the early learning ability and benefit from its loss convergence.

4.4.3. The choice of loss function

The choice of the loss function is significant for the training of NAG. As can be seen, in NAG, mean square error (MSE) loss (marked with violet solid line) performs better than binary cross-entropy (BCE) loss (marked with green solid line). Furthermore, Wasserstein distance loss is also verified to perform worse because it is challenging to use regularization methods such as gradient penalty.

4.4.4. The choice of aggregation method

We consider two aggregation methods, including mean (marked with red solid line) and weighted mean (marked with violet solid line). The fractions in the weighted mean are adjusted by a linear layer combined with a gating layer. As can be seen, the weighted-mean aggregation method can effectively reduce variance.

4.5. More experiments on NAS-Bench-101

To verify the performance of our proposed NAG, we conduct two more experiments (i.e., accuracy-optimal search and Pareto-optimal front search) on NAS-Bench-101 [32].

4.5.1. Accuracy-optimal search

Architecture space. The architecture space is the same as described in Section 3.1, which is compatible with the space of NAS-Bench-101. The difference is that we just need to take a normal cell into account. The maximum node number N in DAG is 7. We stack each normal cell $N_{cell} = 3$ times and repeat the stage $N_{stack} = 3$ times. There are three candidate operations: 3×3 convolution, 1×1 convolution, and 3×3 max-pooling.

NAS dataset. To verify our method with minimum cost, we build the NAS dataset by looking up NAS-Bench-101 dataset instead of training and evaluating the sampled architectures. In this experiment, only 1% of the NAS-Bench-101 architectures are sampled for training the NAG. According to the validation accuracy, these architectures are divided into three categories: low, medium, and high accuracy. The settings are exactly the same as the accuracy-optimal search on CIFAR-10.

Results. Fig. 7a shows the accuracies of 300 architectures (100 for each category) that are sampled by the trained NAG and not in NAS-Bench-101. We train and evaluate these generated architectures following the NAS-Bench-101 [32]. As can be seen, the accuracies of these architectures vary with different labels. Compared with Fig. 7a and Fig. 7b, the distributions of architectures from the NAG and NAS-Bench-101 are consistent. The above results demonstrate that the NAG can learn the real data distribution of architecture space from samples and generate the architectures that we have never seen before. Furthermore, we take 100 sampled architectures from the NAG using the high accuracy label and compare them with three standard NAS baselines. As shown in Table 3, experimental results demonstrate that RS-NAG outperforms three standard NAS baselines and has about 40 times sampling efficiency than random search. Compared with Table 1 and 2, NAG performs better in NAS-Bench-101 search space than in open-domain search

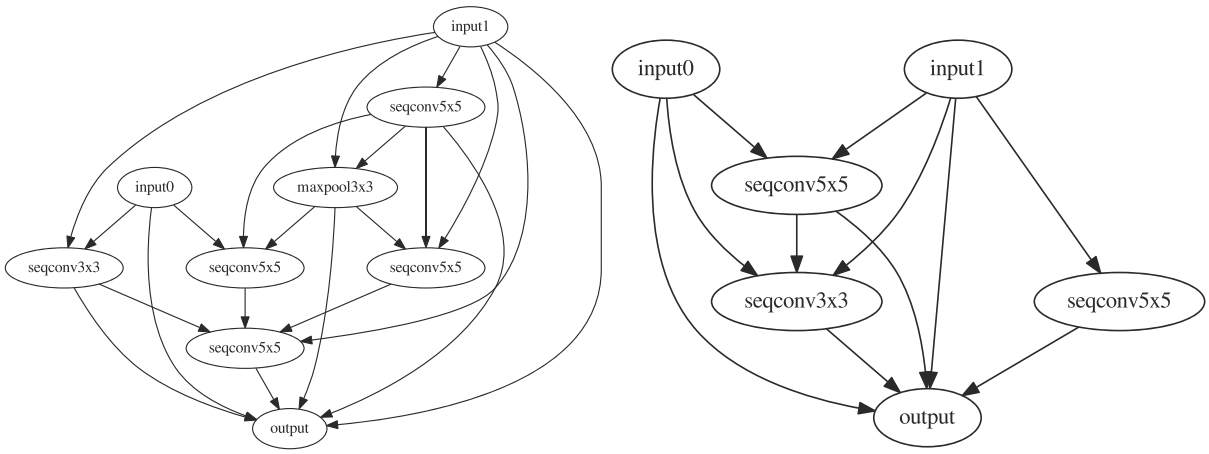


Fig. 5. The cell of the best architecture we discover. Clearly, the normal cell and reduction cell are different from previous NAS architectures: 1) the number of nodes in the two types of cells is different; 2) our discovered normal cell is more complex, deeper, and wider.

Table 1
The comparison of different CNN models on CIFAR-10. † The cost of RS-NAG is the total cost of training the NAG and random search, not including the cost of building NAS datasets. We run the accuracy-optimal experiment three times and report the average error and standard deviation. It is worth noting that the accuracy-optimal search and Pareto-optimal front search are not comparable because of their different numbers N_s of the sampled architectures. The former samples ten architectures, while the latter samples three architectures.

Architecture	Test Error (%)	Params (M)	N_s	Search Cost (GPU days)
NASNet-A + cutout [4]	2.65	3.3	20000	1800
AmoebaNet-A + cutout [8]	3.34	3.2	20000	3150
AmoebaNet-B + cutout [8]	2.55	2.8	27000	3150
PNASNet-5 [16]	3.41	3.2	1160	1160
NAONet-WS + cutout [13]	2.93	2.5	4000	0.3
ENAS + cutout [3]	2.89	4.6	–	0.45
DARTS(second order)+cutout [9]	2.76	3.3	–	4
SNAS(mild constraint)+cutout [11]	2.98	2.9	–	1.5
GDAS + cutout [12]	2.93	3.4	–	0.21
BayesNAS(lambda = 0.005)+cutout [37]	2.81	3.4	–	0.2
P-DARTS + cutout [38]	2.50	3.4	–	0.3
EPNAS + cutout [39]	2.79	4.3	600	8
PC-DARTS + cutout [40]	2.57	3.6	–	0.1
RS-NAG(high acc)+cutout	2.50 ± 0.02	4.2 ± 0.2	10	1†
RS-NAG(few params, high acc)+cutout	3.38	1.1	3	1†
RS-NAG(medium params, high acc)+cutout	2.66	3.0	3	1†
RS-NAG(large params, high acc)+cutout	3.05	4.1	3	1†
RS-NAG(ensemble model)+cutout	3.17	4.5	3	1†

Table 2
The comparison of different CNN models on ImageNet. All architectures are transferred from CIFAR-10 architectures. We only transfer their architectures rather than their weights and then train them on the Imagenet.

	Test Error (%)		
Architecture	Top-1	Top-5	Params (M)
MobileNet [41]	29.4	10.5	4.2
ShuffleNet [42]	26.3	10.2	∼5
MobileNet-V2 [43]	25.3	–	6.9
ShuffleNet-V2 [44]	25.1	–	7.4
NASNet-A [4]	26.0	8.4	5.3
NASNet-B [4]	27.2	8.7	5.3
NASNet-C [4]	27.5	9.0	4.9
AmoebaNet-A [8]	25.5	8	5.1
AmoebaNet-B [8]	26.0	8.5	5.3
PNASNet-5 [16]	25.8	8.1	5.1
NAONet [13]	25.7	8.2	11.35
DARTS(second order) [9]	26.7	8.7	4.7
SNAS(mild constraint) [11]	27.3	9.2	4.3
PC-DARTS [40]	25.1	7.8	5.3
P-DARTS [38]	24.4	7.4	4.9
RS-NAG(high acc)	25.5	7.8	5.0

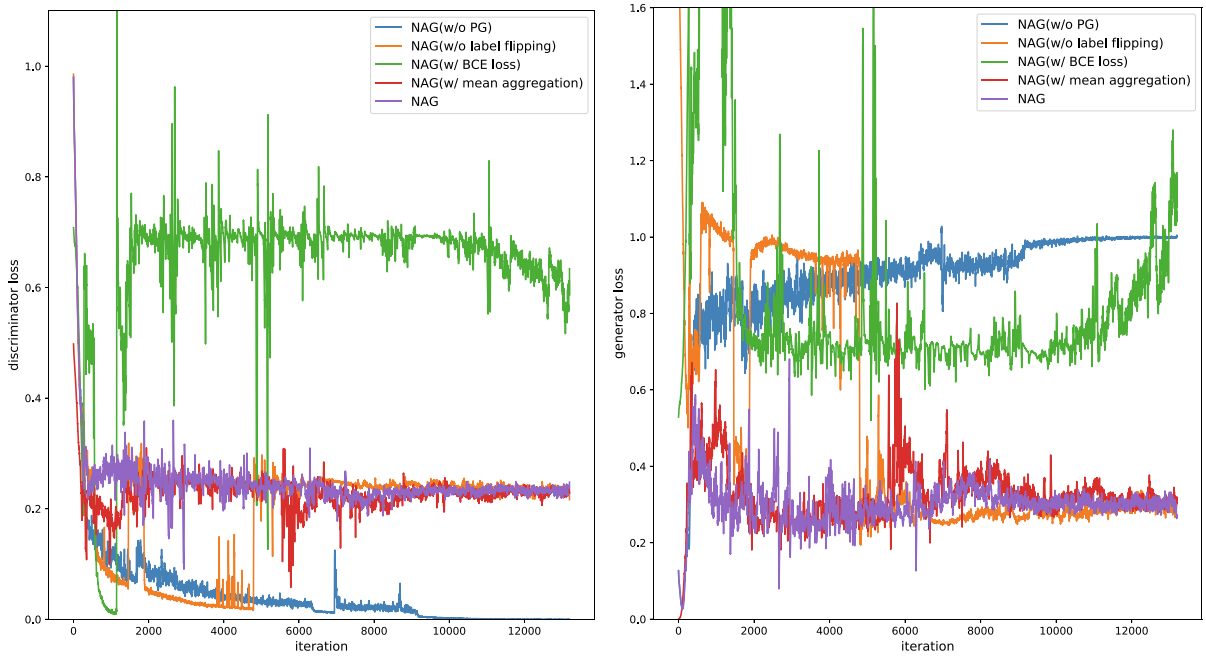
space, which is because the open-domain search space is much larger than NAS-Bench-101 search space.

4.5.2. Pareto-optimal front search

Architecture space. The architecture space is the exact implementation used in Section 4.5.1.

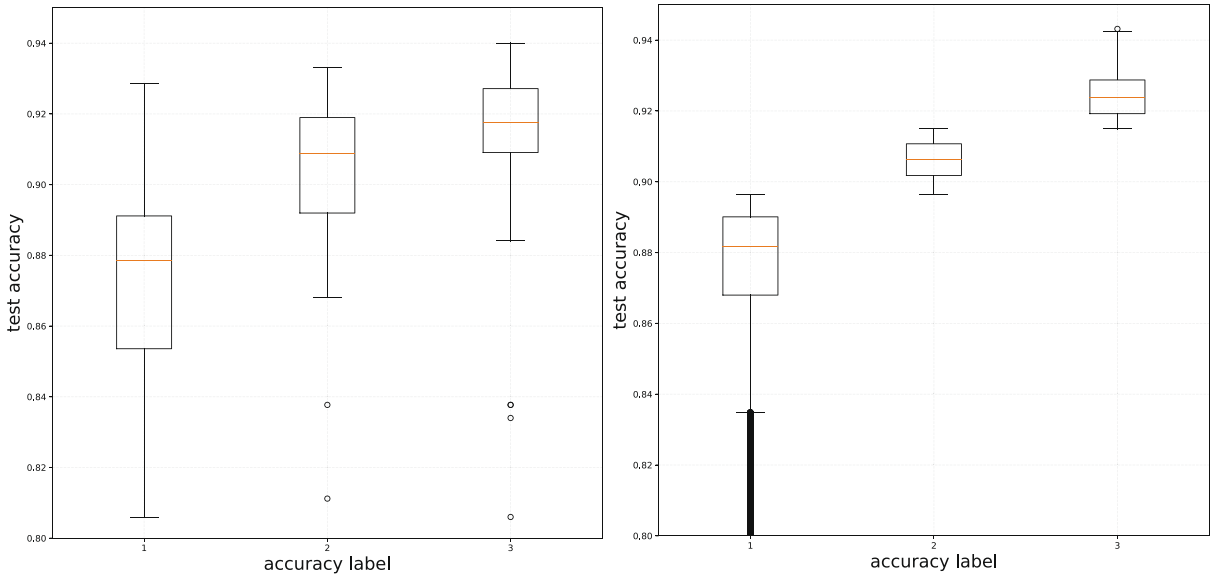
NAS dataset. We use the same NAS dataset used in Section 4.5.1. In this section, we focus on the Pareto-optimal front search. Most settings are the same as described in Section 4.5.1. The difference is that the number of parameters is also considered as a performance indicator of interest. In detail, these architectures also fall into nine categories according to a few, medium, large number of parameters, and low, medium, high accuracy.

Results. As shown in Fig. 8, 900 architectures (100 for each category, a total of 9 categories) are sampled from the NAG. In Fig. 8a, there are three types of architectures: architectures with a few number of parameters marked with asterisks, architectures with a medium number of parameters marked with crosses, and



(a) Discriminator loss. (b) Generator loss.

Fig. 6. The ablation studies of our improvements and choices. According to the training loss, the training of NAG (violet line) converges and is the most robust. (Color version for best viewing).



(a) NAG. (b) NAS-Bench-101.

Fig. 7. The comparison of the architecture distribution of NAG and NAS-Bench-101. (a) The architecture distribution captured by our generator; (b) The real architecture distribution of NAS-Bench-101.

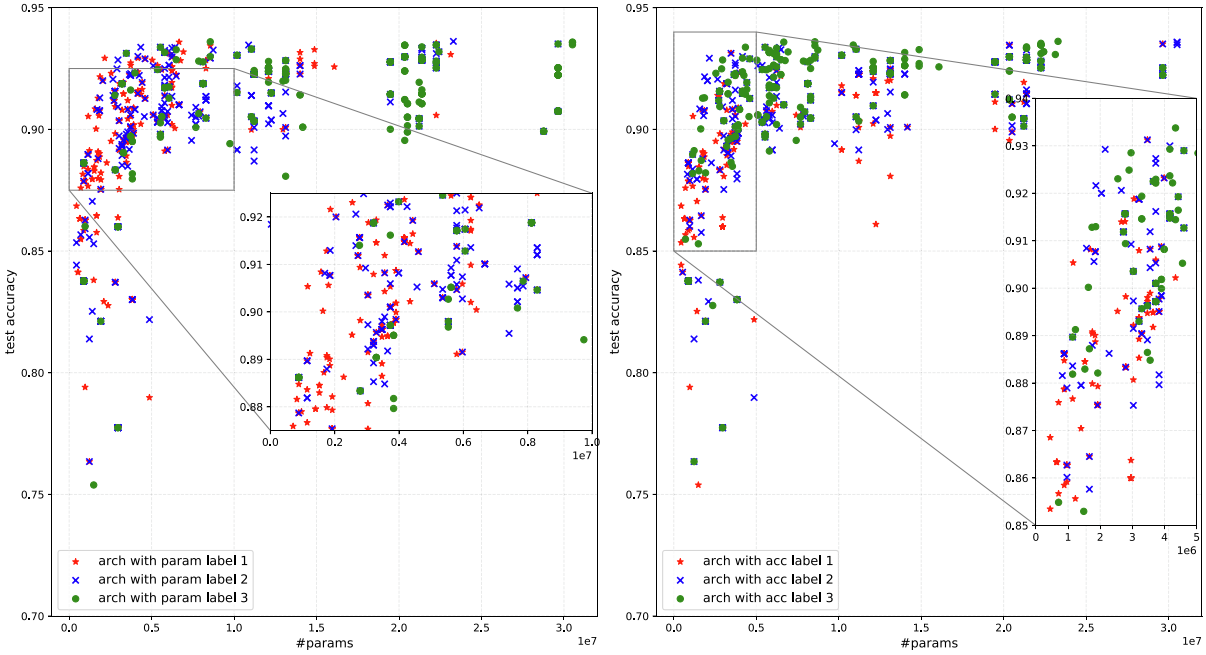
Table 3
The comparison of RS-NAG and three standard NAS baselines on NAS-Bench-101. We run each algorithm independently three times.

Method	Test Error(%)	N_s
NAS-Bench-101	5.68	423624
Sampled 1% architectures	6.02 ± 0.02	4236
Random Search	6.44 ± 0.25	100
Regularized Evolution	6.28 ± 0.20	100
Reinforcement Learning	6.17 ± 0.15	100
RS-NAG	**5.98** ± **0.06**	100

architectures with a large number of parameters marked with dots. Similarly, in Fig. 8b, asterisks, crosses, and dots represent three different architectures with low, medium, and high test accuracy, respectively. As can be seen, the NAG can capture the distribution of architectures with different indicator labels.

5. Conclusion

We propose a novel neural architecture generator, which can generate various architectures that meet the customized requirements in our proposed large-scale architecture space. After adver-



(a) Label for number of parameters. (b) Label for accuracy.

Fig. 8. The architecture distribution of Pareto-optimal front search on NAS-Bench-101. (a) The distribution of architectures with different labels of the number of parameters. The distribution of architectures with different labels of accuracy.

sarial training, our NAG can take advantage of GNN to extract graph information for modeling the architecture properties on a fine-grained level. And our NAG can capture the distribution of architectures with different requirements and efficiently explore those architectures in the architecture space. Experimental results on NAS-Bench-101, CIFAR-10, and ImageNet demonstrate that our NAG can generate diverse architectures according to the customized requirements. As a result, our RS-NAG can efficiently discover the competitive architectures in the larger architecture space, which can be transferable to more complicated tasks.

As shown in Table 1 and Table 2, although RS-NAG does not outperform the state-of-the-art NAS method on CIFAR-10 and ImageNet dataset, it reaches the competitive performance. In Table 3, we argue that NAG performs better in small search space, but there are still deficiencies in large search space, which is the direction of improvement in the future. Besides, like the existing NAS methods, NAG can also work for different tasks, not just image recognition. We expect our work to be followed and lead to breakthrough innovations in the field of neural architecture generation. In future, first, we would like to try other generating methods to improve the performance of the NAG in large search space, such as using a variational graph auto-encoder. Second, we would like to apply other varieties of GAN or GNN to generate better architectures. Finally, the efficiency of the NAS dataset building needs to be improved.

CRediT authorship contribution statement

Kun Jing: Conceptualization, Methodology, Software, Writing - original draft. **Jungang Xu:** Investigation, Writing - review & editing, Supervision. **Zhen Zhang:** Writing - review & editing, Validation.

Declaration of Competing Interest

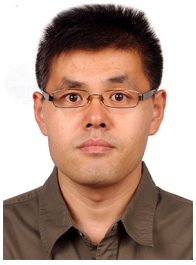
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] B. Zoph, Q.V. Le, Neural architecture search with reinforcement learning, in: Proc. ICLR'17, Toulon, France, 2017.
[2] B. Baker, O. Gupta, N. Naik, R. Raskar, Designing neural network architectures using reinforcement learning, in: Proc. ICLR'17, Toulon, France, 2017.
[3] H. Pham, M.Y. Guan, B. Zoph, Q.V. Le, J. Dean, Efficient neural architecture search via parameter sharing, in: Proc. ICML'18, Stockholm, Sweden, 2018, pp. 4092–4101.
[4] B. Zoph, V. Vasudevan, J. Shlens, Q.V. Le, Learning transferable architectures for scalable image recognition, in: Proc. IEEE CVPR'18, Salt Lake City, UT, USA, 2018, pp. 8697–8710.
[5] L. Xie, A.L. Yuille, Genetic CNN, in: Proc. IEEE ICCV'17, Venice, Italy, 2017, pp. 1388–1397.
[6] E. Real, S. Moore, A. Selle, S. Saxena, Y.L. Suematsu, J. Tan, Q.V. Le, A. Kurakin, Large-scale evolution of image classifiers, in: Proc. ICML'17, Sydney, NSW, Australia, 2017, pp. 2902–2911.
[7] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, K. Kavukcuoglu, Hierarchical representations for efficient architecture search, in: Proc. ICLR'18, Vancouver, BC, Canada, 2018.
[8] E. Real, A. Aggarwal, Y. Huang, Q.V. Le, Regularized evolution for image classifier architecture search, in: Proc. AAAI'19, Honolulu, Hawaii, USA, 2019, pp. 4780–4789.
[9] H. Liu, K. Simonyan, Y. Yang, DARTS: differentiable architecture search, in: Proc. ICLR'19, New Orleans, LA, USA, 2019.
[10] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, K. Keutzer, Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search, in: Proc. IEEE CVPR'19, Long Beach, CA, USA, 2019.
[11] S. Xie, H. Zheng, C. Liu, L. Lin, SNAS: stochastic neural architecture search, in: Proc. ICLR'19, New Orleans, LA, USA, 2019.
[12] X. Dong, Y. Yang, Searching for a robust neural architecture in four GPU hours, in: Proc. IEEE CVPR'19, Long Beach, CA, USA, 2019.
[13] R. Luo, F. Tian, T. Qin, E. Chen, T. Liu, Neural architecture optimization, in: Proc. NeurIPS'18, Montréal, Canada, 2018.
[14] A. Krizhevsky, V. Nair, G. Hinton, Learning multiple layers of features from tiny images, Tech. rep. (2009).
[15] J. Deng, W. Dong, R. Socher, L. Li, K. Li, F. Li, Imagenet: A large-scale hierarchical image database, in: Proc. IEEE CVPR'09, Miami, Florida, USA, 2009, pp. 248–255.
[16] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L. Li, L. Fei-Fei, A.L. Yuille, J. Huang, K. Murphy, Progressive neural architecture search, in: Proc. ECCV'18, Vol. 1, Munich, Germany, 2018, pp. 19–35.
[17] F.P. Such, A. Rawal, J. Lehman, K.O. Stanley, J. Clune, Generative teaching networks: Accelerating neural architecture search by learning to generate synthetic training data, in: Proc. ICML'20, 2020, pp. 9206–9216.
[18] X. Ning, Y. Zheng, T. Zhao, Y. Wang, H. Yang, A generic graph-based neural architecture encoding scheme for predictor-based NAS, in: Proc. ECCV'20, 2020, pp. 189–204.

[19] M.S. Abdelfattah, A. Mehrotra, Ł. Dudziak, N.D. Lane, Zero-cost proxies for lightweight nas, in: Proc. ICLR'21, 2021.
[20] M. Lin, P. Wang, Z. Sun, H. Chen, X. Sun, Q. Qian, H. Li, R. Jin, Zen-nas: A zero-shot NAS for high-performance deep image recognition, CoRR abs/2102.01063.
[21] I.J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A.C. Courville, Y. Bengio, Generative adversarial nets, in: Proc. NeurIPS'14, Montreal, Quebec, Canada, 2014, pp. 2672–2680.
[22] X. Mao, Q. Li, H. Xie, R.Y.K. Lau, Z. Wang, S.P. Smolley, Least squares generative adversarial networks, in: Proc. IEEE ICCV'17, Venice, Italy, 2017, pp. 2813–2821.
[23] T. Salimans, I.J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, Improved techniques for training gans, in: Proc. NeurIPS'16, Barcelona, Spain, 2016, pp. 2226–2234.
[24] M. Arjovsky, S. Chintala, L. Bottou, Wasserstein GAN, CoRR abs/1701.07875..
[25] L. Metz, B. Poole, D. Pfau, J. Sohl-Dickstein, Unrolled generative adversarial networks, in: Proc. ICLR'17, Toulon, France, 2017.
[26] T. Miyato, T. Kataoka, M. Koyama, Y. Yoshida, Spectral normalization for generative adversarial networks, in: Proc. ICLR'18, Vancouver, BC, Canada, 2018.
[27] M. Mirza, S. Osindero, Conditional generative adversarial nets, CoRR (abs/1411.1784.).
[28] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, M. Sun, Graph neural networks: A review of methods and applications, CoRR abs/1812.08434.
[29] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in: Proc. ICLR'17, Toulon, France, 2017.
[30] J. You, R. Ying, X. Ren, W.L. Hamilton, J. Leskovec, Graphrnn: Generating realistic graphs with deep auto-regressive models, in: Proc. ICML'18, Stockholm, Sweden, 2018, pp. 5694–5703.
[31] D. Friede, J. Lukasik, H. Stuckenschmidt, M. Keuper, A variational-sequential graph autoencoder for neural architecture performance prediction, CoRR abs/1912.05317.
[32] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, F. Hutter, Nas-bench-101: Towards reproducible neural architecture search, in: Proc. ICML'19, Long Beach, California, USA, 2019.
[33] A. Zela, J. Siems, F. Hutter, Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search, in: Proc. ICLR'20, Addis Ababa, Ethiopia, 2020.
[34] X. Dong, Y. Yang, Nas-bench-201: Extending the scope of reproducible neural architecture search, in: Proc. ICLR'20, Addis Ababa, Ethiopia, 2020.
[35] X. Zheng, R. Ji, L. Tang, B. Zhang, J. Liu, Q. Tian, Multinomial distribution learning for effective neural architecture search, in: Proc. IEEE/CVF ICCV'19, Seoul, Korea (South), 2019, pp. 1304–1313.
[36] M. Arjovsky, L. Bottou, Towards principled methods for training generative adversarial networks, in: Proc. ICLR'17, 2017.
[37] H. Zhou, M. Yang, J. Wang, W. Pan, Bayesnas: A bayesian approach for neural architecture search, in: Proc. ICML'19, Long Beach, California, USA, 2019, pp. 7603–7613.
[38] X. Chen, L. Xie, J. Wu, Q. Tian, Progressive differentiable architecture search: Bridging the depth gap between search and evaluation, in: Proc. IEEE/CVF ICCV'19, Seoul, Korea (South), 2019, pp. 1294–1303.
[39] Y. Zhou, P. Wang, EPNAS: efficient progressive neural architecture search, in: Proc. BMVC'19, Cardiff, UK, 2019, p. 71.
[40] Y. Xu, L. Xie, X. Zhang, X. Chen, G. Qi, Q. Tian, H. Xiong, PC-DARTS: partial channel connections for memory-efficient architecture search, in: Proc. ICLR'20, Addis Ababa, Ethiopia, 2020.
[41] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, Mobilenets: Efficient convolutional neural networks for mobile vision applications, CoRR abs/1704.04861.
[42] X. Zhang, X. Zhou, M. Lin, J. Sun, Shufflenet: An extremely efficient convolutional neural network for mobile devices, in: Proc. CVPR 2018, 2018.
[43] M. Sandler, A.G. Howard, M. Zhu, A. Zhmoginov, L. Chen, Mobilenetv 2: Inverted residuals and linear bottlenecks, in: Proc. CVPR 2018, Salt Lake City, UT, USA, June 18–22, 2018.
[44] N. Ma, X. Zhang, H. Zheng, J. Sun, Shufflenet V2: practical guidelines for efficient CNN architecture design, in: Proc. ECCV 2018, Munich, Germany, September 8–14, 2018.

Kun Jing is a Ph.D. student in School of Computer Science and Technology, University of Chinese Academy of Sciences. He received his B.S. degree in Internet of things engineering from Chongqing University of Posts and Telecommunications in 2018. His research interests are automated machine learning and natural language processing.



Jungang Xu is a full professor in School of Computer Science and Technology, University of Chinese Academy of Sciences. He received his Ph.D. degree in Computer Applied Technology from Graduate University of Chinese Academy of Sciences in 2003. His current research interests are automated machine learning, computer vision and natural language processing.

Zhen Zhang is a research assistant in School of Computer Science and Technology, University of Chinese Academy of Sciences. She received her M.S. degree in Data Science from the George Washington University in 2019. Her research interest is automated machine learning.