

CARS: Continuous Evolution for Efficient Neural Architecture Search

Zhaohui Yang^{1,2}, Yunhe Wang², Xinghao Chen², Boxin Shi^{3,4},
Chao Xu¹, Chunjing Xu², Qi Tian^{2*}, Chang Xu⁵

¹ Key Lab of Machine Perception (MOE), Dept. of Machine Intelligence, Peking University.

² Noah's Ark Lab, Huawei Technologies. ³ NELVT, Dept. of CS, Peking University. ⁴ Peng Cheng Laboratory.

⁵ School of Computer Science, Faculty of Engineering, University of Sydney.

{zhaohuiyang, shiboxin}@pku.edu.cn; xuchao@cis.pku.edu.cn

{yunhe.wang, xinghao.chen, tian.qil, xuchunjing}@huawei.com; c.xu@sydney.edu.au

Abstract

Searching techniques in most of existing neural architecture search (NAS) algorithms are mainly dominated by differentiable methods for the efficiency reason. In contrast, we develop an *efficient continuous evolutionary* approach for searching neural networks. Architectures in the population that share parameters within one SuperNet in the latest generation will be tuned over the training dataset with a few epochs. The searching in the next evolution generation will directly inherit both the SuperNet and the population, which accelerates the optimal network generation. The non-dominated sorting strategy is further applied to preserve only results on the *Pareto* front for accurately updating the *SuperNet*. Several neural networks with different model sizes and performances will be produced after the continuous search with *only 0.4 GPU days*. As a result, our framework provides a series of networks with the number of parameters ranging from 3.7M to 5.1M under mobile settings. These networks surpass those produced by the state-of-the-art methods on the benchmark *ImageNet* dataset.

1. Introduction

Convolutional neural networks have made great progress in a large range of computer vision tasks, such as recognition [22, 19, 18], detection [12], and segmentation [21]. Over-parameterized deep neural network can produce impressive performance but will consume huge computational resources at the same time. The efficient block design [57, 56, 55], tensor decomposition [27, 52, 53, 45, 46], pruning [28, 29, 13], distillation [4, 51] and quantization [25] are popular techniques to make networks efficient. Designing novel network architectures heavily relies on human experts' knowledge and experience, and may take many tri-

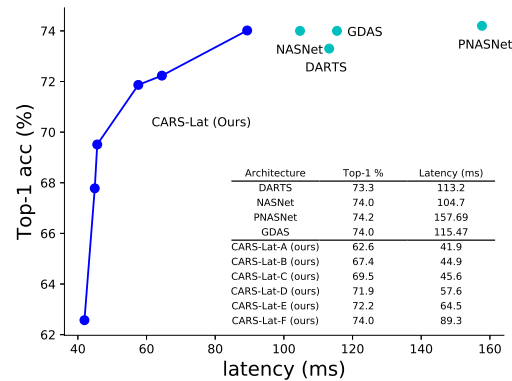


Figure 1. CARS-Lat models are searched on *CIFAR-10* dataset. The search phase considers validation performance and device-aware variable, *i.e.*, *mobile device latency on HUAWEI P30 Pro*. The Top-1 accuracies are the performances on the ILSVRC2012 dataset.

als before achieving meaningful results [22]. To accelerate this process and make it automated, network architecture search (NAS) [44, 14, 36, 20, 16] has been proposed, and the learned architectures have exceeded those human-designed architectures on a variety of tasks. However, these searching methods usually require lots of computational resources to search for architectures of acceptable performance.

Techniques for searching neural architectures are mainly clustered into three groups, *i.e.*, Evolution Algorithm (EA) based, Reinforcement Learning (RL) based, and gradient-based methods. EA-based works, [38, 37, 48, 42, 40, 41] initialize a set of models and evolve for better architectures, which is time-consuming, *e.g.*, Real *et al.* takes 3150 GPU days for searching [38]. RL-based works, [60, 58] use the controller to predict a sequence of operations, and train different architectures to gain the rewards. Given an architecture, these methods have to train it for a large num-

*Corresponding author.

ber of epochs, and then evaluate its performance to guide for evolution or optimize the controller, which makes the searching stage less efficient. Gradient-based methods (*e.g.*, DARTS [32]) first train a SuperNet and introduce the attention mechanism on the connections while searching, then remove weak connections after searching. This phase is conducted by gradient descent optimization and is quite efficient. However, the searched architectures suffer from lack of variety.

Although some experiments in [37] show that the evolutionary algorithm discovers better neural architectures than RL-based approaches, the search cost of EA is much expensive due to the evaluation procedure of each individual, *i.e.*, a neural network in the evolutionary algorithm is independently evaluated. Moreover, there could be some architectures with extremely worse performance in the search space. If we directly follow the weight sharing approach proposed by ENAS [35], the SuperNet has to be trained to compensate for those worse search space. It is necessary to reform existing evolutionary algorithms for efficient yet accurate neural architecture search.

In this paper, we propose an efficient EA-based neural architecture search framework. A continuous evolution strategy is developed to maximally utilize the knowledge we have learned in the last evolution generation. Specifically, a SuperNet is first initialized with considerable cells and blocks. Individuals in the evolutionary algorithm representing architectures derived in the SuperNet will be generated through several benchmark operations (*i.e.*, crossover and mutation). Non-dominated sort strategy is adopted to select several excellent architectures with different model sizes and accuracies, and corresponding cells in the SuperNet will be updated for subsequent optimization. The evolution procedure in the next generation is continuously executing based on the updated SuperNet and the multi-objective solution set obtained by the non-dominated sorting. In addition, we propose to exploit a protection mechanism to avoid the small model trap problem. The proposed continuous evolution architecture search (CARS) can provide a series of models on the Pareto front with high efficiency. The superiority of our method is verified on benchmark datasets over the state-of-the-art methods.

2. Related Works

2.1. Network Architecture Search

Gradient-based Network Architecture Search (NAS) methods contain two steps: network parameter optimization and architecture optimization. The network parameter optimization step optimizes the parameters in the standard layers (*i.e.*, convolution, batch normalization, fully connected layer). The architecture optimization step learns the pattern of accurate network architectures.

The parameter optimization step can be divided into two categories, independent optimization and sharing optimization. Independent optimization learns each network separately, *i.e.*, AmoebaNet [37] takes thousands of GPU days to evaluate thousands of models. To accelerate training, [11, 10] initialize parameters by network morphism. One-shot methods [1, 17] step further by sharing all the parameters for different architectures within one SuperNet. Rather than training thousands of different architectures, only one SuperNet is required to be optimized.

The architecture optimization step includes RL-based, EA-based, and gradient-based approaches. RL-based methods [60, 61, 35] use the recurrent network as the network architecture controller, and the performances of the generated architectures are utilized as the rewards for training the controller. The controller converges during training and finally outputs architectures with superior performance. EA-based approaches [48, 37] search architectures with the help of evolutionary algorithms. The validation accuracy of each individual is utilized as the fitness to evolve the next generation. Gradient-based approaches [32, 49, 47] view the network architecture as a set of learnable parameters and optimize the parameters by the standard back-propagation algorithm.

2.2. Multi-objective Network Architecture Search

Considering multiple complementary objectives, *i.e.*, accuracy, the number of parameters, float operations (FLOPs), energy, and latency, there is no single architecture that surpasses all the others on all the objectives. Therefore, architectures within the Pareto front are desired. Many different works have been proposed to deal with multi-objective network architecture search. NEMO [26] and MNASNet [44] target at speed and accuracy. DPPNet and LEMONADE [8, 11] consider device-related and device-agnostic objectives. MONAS [23] targets at accuracy and energy. NSGANet [33] considers FLOPs and accuracy.

These methods are less efficient for models are optimized separately. In contrast, our architecture optimization and parameter optimization steps are conducted alternatively. Besides, the parameters for different architectures are shared, which makes the search stage much more efficient.

3. Approach

In this section, we develop a novel continuous evolutionary approach for searching neural architectures, namely CARS. The CARS search stage includes **two procedures**, *i.e.*, **parameter optimization** and **architecture optimization**.

We use the **Genetic Algorithm (GA)** for architecture evolution because GA maintains a set of well-performed architectures that cover a vast space. We maintain a set of architectures (a.k.a. connections) $C = \{C_1, \dots, C_P\}$, where P is the population size. The architectures in the population

are gradually updated according to the proposed pNSGA-III method during the architecture optimization step. To make the search stage efficient, we maintain a SuperNet \mathcal{N} , which shares parameters W for different architectures. The parameter sharing strategy dramatically reduces the computational complexity of separately training these different architectures.

3.1. SuperNet of CARS

Different networks are sampled from the SuperNet \mathcal{N} , and each network \mathcal{N}_i can be represented by a set of full precision parameters W_i and a set of binary connection parameters (i.e., $\{0, 1\}$) C_i . The 0-element in connection C_i means the network does not contain this connection to transform data flow, and the 1-element connection means the network uses this connection. From this point of view, each network \mathcal{N}_i could be represented as (W_i, C_i) pair.

Full precision parameters W are shared by a set of networks. If these network architectures are fixed, the parameters could be optimized through back-propagation. The optimal W fits for all the networks \mathcal{N}_i to achieve higher recognition performance. After the parameters are converged, we could alternately optimize the binary connections C by the GA algorithm. These two steps form the main optimization of our proposed method. We will introduce these two optimization steps in the following.

3.2. Parameter Optimization

The parameters W are the collection of all the parameters in the network. The parameters W_i of the i -th individual are $W_i = W \odot C_i$, $i \in \{1, \dots, P\}$, where the \odot is the mask operation that keeps the parameters of the complete graph only for the positions corresponding to 1-elements in the connection C_i . Denote X as the input data, the prediction of this network is $\mathcal{N}_i(X)$, where \mathcal{N}_i is the i -th architecture. The prediction loss can be expressed as $L_i = \mathcal{H}(\mathcal{N}_i(X), Y)$, where \mathcal{H} is the criterion and Y is the target. The gradient of W_i can be calculated as

$$dW_i = \frac{\partial L_i}{\partial W_i} = \frac{\partial L_i}{\partial W} \odot C_i. \quad (1)$$

Parameters W should fit all the individuals, and thus the gradients for all networks are accumulated to calculate the gradient of parameters W

$$dW = \frac{1}{P} \sum_{i=1}^P dW_i = \frac{1}{P} \sum_{i=1}^P \frac{\partial L_i}{\partial W} \odot C_i. \quad (2)$$

Any layer is only optimized by networks that use this layer during forwarding. By collecting the gradients of individuals in the population, the parameters W are updated through the SGD algorithm.

As we have maintained a large set of architectures with shared weights in the SuperNet, we borrow the idea of

stochastic gradient descent and use mini-batch architectures for updating parameters. Accumulating the gradients for all networks would take much time for one-step gradient descent, and thus we use mini-batch architectures for updating shared weights. We use B different architectures where $B < P$, and the indices of architectures are $\{n_1, \dots, n_B\}$ to update parameters. The efficient parameter updating of Eqn 2 is detailed as Eqn 3

$$dW \approx \frac{1}{B} \sum_{j=1}^B \frac{\partial L_{n_j}}{\partial W_{n_j}}. \quad (3)$$

Hence, the gradients over a mini-batch of architectures are taken as an unbiased approximation of the averaged gradients of all the P different individuals. The time cost for each update could be largely reduced, and the appropriate mini-batch size leads to a balance between efficiency and accuracy.

3.3. Architecture Optimization

As for the architecture optimization procedure, we use the evolution algorithm together with the non-dominated sorting strategy. The non-dominated sorting strategy has been introduced in the NSGA-III [7]. Denote $\{\mathcal{N}_1, \dots, \mathcal{N}_P\}$ as P different networks and $\{\mathcal{F}_1, \dots, \mathcal{F}_M\}$ as M different measurements we want to minimize. The measurements, for example, the number of parameters, float operations, latency, energy, and accuracy, could have some conflicts, which increase the difficulty in discovering an optimal solution that minimizes all these metrics.

In practice, \mathcal{N}_i dominates \mathcal{N}_j if two conditions are satisfied: (1) For any of the measurements, the performance of \mathcal{N}_i is not worse than that of \mathcal{N}_j . (2) The model \mathcal{N}_i behaves better than \mathcal{N}_j on at least one measurement. Formally, the definition of domination can be summarized as below.

Definition 1. Considering two networks \mathcal{N}_i and \mathcal{N}_j , and a series of measurements $\{\mathcal{F}_1, \dots, \mathcal{F}_M\}$ we want to minimize. If

$$\begin{aligned} \mathcal{F}_k(\mathcal{N}_i) &\leq \mathcal{F}_k(\mathcal{N}_j), \quad \forall k \in \{1, \dots, M\} \\ \mathcal{F}_k(\mathcal{N}_i) &< \mathcal{F}_k(\mathcal{N}_j), \quad \exists k \in \{1, \dots, M\}, \end{aligned} \quad (4)$$

\mathcal{N}_i is said to dominate \mathcal{N}_j , i.e., $\mathcal{N}_i \preceq \mathcal{N}_j$.

According to the above definition, if \mathcal{N}_i dominates \mathcal{N}_j , \mathcal{N}_j can be replaced by \mathcal{N}_i during the evolution procedure since \mathcal{N}_i performs better in terms of at least one metric and not worse on other metrics. By exploiting this approach, we can select a series of excellent neural architectures from the population in the current generation. Then, these networks can be utilized for updating the corresponding parameters in the SuperNet.

Although the above non-dominated sorting strategy uses the NSGA-III method [7] to select some better models for

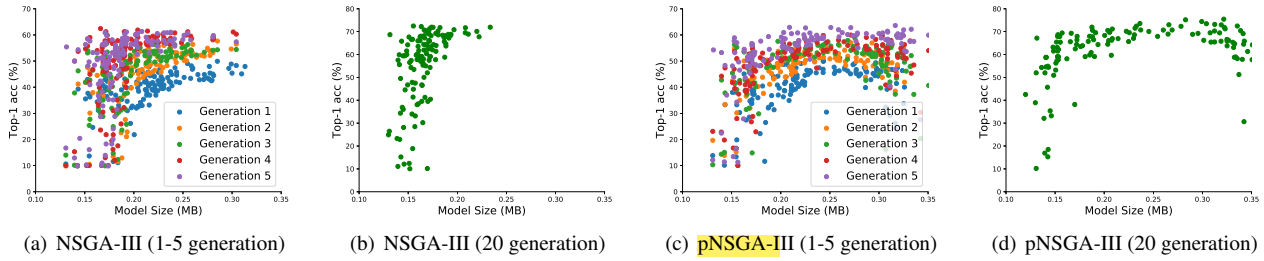


Figure 2. Comparison between different evolution strategies. SuperNet is trained on the train set and evaluated on the validation set. Figure 2(a) shows five evolution generations using NSGA-III. Evolution with NSGA-III suffers from small model trap, which lead the distribution bias to smaller models, and Figure 2(b) shows the distribution of evolution with NSGA-III after 20 generations, where the maintained architectures are all the small models. Figure 2(c) shows the evolution generations using proposed pNSGA-III. Evolution with pNSGA-III provides protection for larger models, and Figure 2(d) shows the distribution of evolution with pNSGA-III after 20 generations, where the maintained architectures covers a large range over the model size dimension.

updating parameters, there exists a *small model trap phenomenon* during the search procedure. Specifically, since the parameters in the SuperNet still need optimization, the accuracy of each individual architecture in the current generation may not always stand for its performance that can be eventually achieved, as discussed in NASBench-101 [54]. Thus, some smaller models with fewer parameters but higher test accuracy tend to dominate those larger models of lower accuracy but have the potential for achieving higher accuracies, as shown in Figure 3.

Therefore, we propose to improve the conventional NSGA-III for protecting these larger models, namely pNSGA-III. More specifically, the pNSGA-III algorithm takes the increasing speed of the accuracy into consideration. We take the validation accuracy and the number of parameters as an example. For NSGA-III method, the non-dominated sorting algorithm considers two different objectives and selects individuals according to the sorted Pareto stages. For the proposed pNSGA-III, besides considering the number of parameters and accuracy, we also conduct a non-dominated sorting algorithm that considers the increasing speed of the accuracy and the number of parameters. Then the two different Pareto stages are merged.¹ Assuming P is the population size, after having two Pareto stages $R_{1...n_1}, Q_{1...n_2}$, we gradually merge two Pareto stages from

¹The concerns about better middle sized models: The pNSGA-III process two times. For the first time, non-dominated sorting considers model size and accuracy (larger models with higher accuracy and smaller models with lower accuracy, the left side sorting). For the second time, non-dominated sorting considers the model size and increasing speed of accuracy (larger models with slower accuracy increasing speed and smaller models with faster accuracy increasing speed, the right side sorting). For the better middle-sized models, because they are larger and have higher accuracy than the small models, so they are ranked in the first few Pareto fronts for the first run. Also, the better middle-sized model with higher precision has a higher accuracy increasing speed than large models. Thus they are also ranked in the first few Pareto fronts for the second run. The selected architectures to construct the next generation is a hat-like form. So our proposed pNSGA-III will keep the accurate middle sized models in the next generation without omissions.

the first Pareto front, and the union set U_i after merging the i -th front is $U_i = (R_1 \cup Q_1) \cup \dots \cup (R_i \cup Q_i)$. We keep the first P individuals from $U_{\max(n_1, n_2)}$. In this way, the large networks with slower performance increasing speed could be kept in the population.

In Figure 2, the populations of using NSGA-III and pNSGA-III are visualized. If we use NSGA-III to update architectures, the small model trap problem is encountered. It is obvious that the pNSGA-III can protect large models during evolution and provide a wide range of models. More detailed discussions are introduced in the following section.

3.4. Continuous Evolution for CARS

In summary, there are two steps for searching optimal architectures by using the proposed CARS pipeline, 1) Architecture Optimization 2) Parameter Optimization. In addition, the parameter warm-up is also introduced to update the parameters at first.

Parameter Warmup. Since the shared weights of our SuperNet are randomly initialized, if the architectures in the population are also randomly initialized, the most frequently used operations for all the architectures would be trained more times compared with other operations. Thus, by following *one-shot NAS* methods [1, 17, 6, 47], we use a uniform sampling strategy to initialize the parameters in the SuperNet. In this way, the SuperNet trains each possible operation with the same possibility. For example, in DARTS [32] pipeline, there are eight different operations for each node, including convolution, pooling, identity mapping and no connection. Each operation will be sampled with a probability of $\frac{1}{8}$.

Architecture Optimization. After initializing the parameters of the SuperNet, we first randomly sample P different architectures, where P is a hyper-parameter and denotes the number of maintained individuals in the population. During

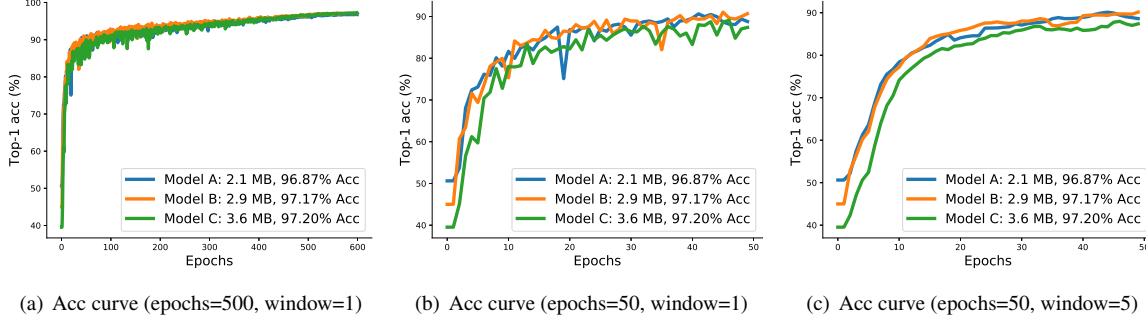


Figure 3. The accuracy curves of **three models with different model sizes**. The left figure shows the accuracy curves of training for 600 epochs, the middle figure shows the accuracy curves for the first 50 epochs, and the right figure shows the curves smoothed by a window size of 5.

the architecture evolution step, we first generate $t \times P$ offsprings, where t is the hyper-parameter controlling the expand ratio. We then use pNSGA-III to sort the architectures and select P individuals from $(t + 1) \times P$ individuals. The selected P architectures form the next generation.

Parameter Optimization. Given a set of architectures, we use the proposed mini-batch architectures update scheme for parameter optimization according to Eqn 3.

Algorithm 1 summarizes the detailed procedure of the proposed continuous evolutionary algorithm for searching neural architectures.

3.5. Search Time Analysis

During the search stage of CARS, the train set is used for updating network parameters, and the validation set is used for updating architectures. Assuming the average training time on the train set for one architecture is T_{tr} , and the inference time on the validation set is T_{val} . The first warmup stage takes E_{warm} epochs, and it needs $T_{warm} = E_{warm} \times T_{tr}$ in this stage to initialize parameters in the SuperNet \mathcal{N} .

Assuming the architectures evolve for E_{evo} generations in total. And each generation contains parameter optimization and architecture optimization steps. The parameter optimization step trains the SuperNet for E_{param} epochs on train set between generations, thus the time cost for parameter optimization in one evolution generation is $T_{param} = E_{param} \times T_{tr} \times B$, and the B is the mini-batch size. For the architecture optimization step, all the individuals can be inferred in parallel, so the time cost in this step could be calculated as $T_{arch} = T_{val}$. Thus the total time cost for E_{evo} evolution generations is $T_{evo} = E_{evo} \times (T_{param} + T_{arch})$. All the searching time cost in CARS is,

$$\begin{aligned} T_{total} &= T_{warm} + T_{evo} \\ &= E_{warm} \times T_{tr} + \\ &\quad E_{evo} \times (E_{param} \times T_{tr} \times B + T_{val}) \end{aligned} \quad (5)$$

Algorithm 1 Continuous Evolution for Efficient Neural Architecture Search

Input: SuperNet \mathcal{N} , connections $C = \{C_1^{(0)}, \dots, C_P^{(0)}\}$, offspring expand ratio t , evolution number E_{evo} , multi-objectives $\{\mathcal{F}_1, \dots, \mathcal{F}_M\}$, parameter optimization epochs E_{param} , and criterion \mathcal{H} .

- 1: Warm up the SuperNet \mathcal{N} for E_{warm} epochs.
- 2: **for** $e = 1, \dots, E_{evo}$ **do**
- 3: **for** $i = 1, \dots, E_{param}$ **do**
- 4: **for** Mini-batch data X , target Y in loader **do**
- 5: Random sample B indices n_1, \dots, n_B .
- 6: Select the corresponding B connections C_1, \dots, C_{n_B} according to the indices.
- 7: Mask the SuperNet \mathcal{N} to form the sampled networks $\mathcal{N}_1, \dots, \mathcal{N}_{n_B}$.
- 8: Forward B sampled networks.
- 9: Calculate loss $L = \frac{1}{B} \sum_{i=1}^B \mathcal{H}(\mathcal{N}_{n_i}(X), Y)$.
- 10: Compute the gradients according to Eqn 3.
- 11: Update the network parameters W .
- 12: **end for**
- 13: **end for**
- 14: Update $\{C_1^{(e)}, \dots, C_P^{(e)}\}$ using pNSGA-III.
- 15: **end for**

Output: Architectures $C = \{C_1^{(E_{evo})}, \dots, C_P^{(E_{evo})}\}$.

4. Experiments

In this section, we first introduce the SuperNet, and experimental details in our experiments. Then, we examine the small model trap phenomenon, and compare NSGA-III with our proposed pNSGA-III. We search on the CIFAR-10 dataset two times, which considers device-agnostic and device-aware objectives, respectively. All the searched architectures are evaluated on the CIFAR-10 and ILSVRC2012 dataset. These two datasets are the benchmarks for the recognition task.

Table 1. Comparison with state-of-the-art image classifiers on **CIFAR-10 dataset**. The multi-objectives used for architecture optimization are performance and model size. We follow DARTS and use the **cutout strategy for training**.

Architecture	Test Error (%)	Params (M)	Search Cost (GPU days)	Search Method
DenseNet-BC [24]	3.46	25.6	-	manual
PNAS [30]	3.41	3.2	225	SMBO
ENAS + cutout [35]	2.91	4.2	4	RL
NASNet-A + cutout [61]	2.65	3.3	2000	RL
AmoebaNet-A + cutout [37]	3.12	3.1	3150	evolution
Hierarchical evolution [31]	3.75	15.7	300	evolution
SNAS (mild) + cutout [49]	2.98	2.9	1.5	gradient
SNAS (moderate) + cutout [49]	2.85	2.8	1.5	gradient
SNAS (aggressive) + cutout [49]	3.10	2.3	1.5	gradient
DARTS (first) + cutout [32]	3.00	3.3	1.5	gradient
DARTS (second) + cutout [32]	2.76	3.3	4	gradient
Random Search [32]	3.29	3.2	4	random
RENA [59]	3.87	3.4	-	RL
NSGANet [33]	3.85	3.3	8	evolution
LEMONADE [11]	3.05	4.7	80	evolution
CARS-A	3.00	2.4	0.4	evolution
CARS-B	2.87	2.7	0.4	evolution
CARS-C	2.84	2.8	0.4	evolution
CARS-D	2.95	2.9	0.4	evolution
CARS-E	2.86	3.0	0.4	evolution
CARS-F	2.79	3.1	0.4	evolution
CARS-G	2.74	3.2	0.4	evolution
CARS-H	2.66	3.3	0.4	evolution
CARS-I	2.62	3.6	0.4	evolution

4.1. Experimental Settings

SuperNet Backbones. To illustrate the effectiveness of our method, we evaluate our CARS on a popular used search space same as DARTS [32]. DARTS is a differentiable NAS system and searches for **reduction and normal cells**. The normal cell is used for the layers that have the same spatial size of input feature and output feature. The **reduction cell is used for layers with downsampling on input feature maps**. After searching for these two kinds of cells, the network is constructed by stacking a set of searched cells. The search space contains eight different operations, including four types of convolution, two kinds of pooling, skip connect, and no connection.

Evolution Details. In the DARTS search space, each intermediate node in a cell is connected with two previous nodes. Crossover and mutation are conducted on the corresponding nodes. Both crossover ratio and mutation ratio are set to 0.25, and we randomly generate new architectures with a probability of 0.5. For the crossover operation, each node has a ratio of 0.5 to crossover its connections, and for mutation operation, each node has a ratio of 0.5 to be randomly reassigned.

4.2. Experiments on CIFAR-10

Our experiments on CIFAR-10 include the demonstration of the small model trap phenomenon, the comparison of NSGA-III and pNSGA-III, the device-agnostic and device-

aware search. The evaluation are conducted on the CIFAR-10 dataset and the large ILSVRC2012 dataset.

Small Model Trap. In Figure 3, the accuracy curves of three models are shown. The number of parameters are 2.1M, 2.9M, and 3.6M, respectively. After training for 600 epochs, the accuracies on the CIFAR-10 dataset and the model sizes have a positive correlation, which are 96.87%, 97.17%, and 97.20%. We observe the accuracy curves of the first 50 epochs and conclude two main reasons that result in the small model trap phenomenon. The two reasons are (1) small models are naturally converge faster, and (2) accuracy fluctuates during training. For the largest Model-C, its accuracy is consistently lower than Model-A and Model-B in the first 50 epochs. Therefore, if the NSGA-III algorithm is used, the Model-C will be eliminated, which is the first motivation for our proposed pNSGA-III. This is because larger models are more complex, thus harder to be optimized. For Model-B and Model-A, the accuracy curves are similar (Figure 3(c)). However, due to the accuracy fluctuation during training (Figure 3(b)), if the accuracy of Model-A is higher than Model-B in one epoch, the Model-B would be eliminated for the non-dominated sorting strategy, which is the second reason we proposed. Both two reasons may lead to the elimination of large models in the process of architecture update. Thus, the proposed pNSGA-III is necessary to address the small model trap problem.

NSGA-III vs. pNSGA-III. We use CARS to search for architectures with different NSGA methods during the architecture optimization step. The multi-objectives are the number of parameters and model size. We visualize the distribution trend of the architectures maintained in the population. As Figure 2 shows, updating architectures by using the NSGA-III would encounter the small model trap problem, and large models are eliminated during architecture optimization step. In contrast, updating architectures by using the pNSGA-III protect larger models. The larger models have the potential to increase their accuracies during later epochs but converge slower than small models at the beginning. It is essential to maintain larger models in the population rather than dropping them during the architecture optimization stage if the search target is to find models with various computing resources.

Search on CIFAR-10. We split the CIFAR-10 train set into two parts, *i.e.*, 25,000 images for updating network parameters and 25,000 for updating architectures. The split strategy is the same as DARTS [32] and SNAS [49]. We search for 500 epochs in total, and the parameter warmup stage lasts for the first 10% epochs (50). After that, we initialize the population, which maintains 128 different architectures and gradually evolve them using proposed pNSGA-III. We use pNSGA-III to update architectures after the network parameters are updated for ten epochs.

Evaluate on CIFAR-10. After finishing the CARS search stage, there are $N = 128$ architectures maintained in the population. We evaluate some architectures that have the similar model sizes with previous works [32, 49] for comparison. We retrain the searched architectures on the CIFAR-10 dataset. All the training parameters are the same as DARTS [32].

We compare the searched architectures with state-of-the-arts in Table 1. All the searched architectures can be found in the supplementary material². Our searched architectures have the number of parameters that vary from 2.4M to 3.6M on CIFAR-10 dataset, and the performances of these architectures are on par with the state-of-the-arts. Meanwhile, if we evolve architectures by using NSGA-III method rather than pNSGA-III, we could only search for a set of architectures with approximately 2.4M parameters without larger models, and the models perform relatively poor.

Compared to previous methods like DARTS and SNAS, our method is capable of searching for architectures over a broad range of the searching space. The CARS-G achieves comparable accuracy with DARTS (second-order), resulting in an approximate 2.75% error rate with smaller model size. Using the same 3.3M parameters as DARTS (second-

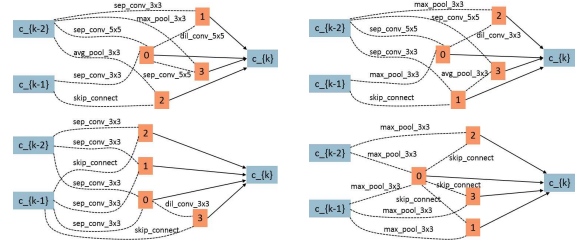


Figure 4. CARS-H and DARTS. On the top are the normal and reduction blocks of CARS-H, and the bottom are the normal and reduction blocks in DARTS (second order).

order), our CARS-H achieves lower test error, 2.66% vs. 2.76%. For the small models, our searched CARS-A/C/D also achieve comparable results with SNAS. Besides, our large model CARS-I achieves a lower error rate 2.62% with slightly more parameters. The overall trend from CARS-A to CARS-J is that the error rate gradually decreases while increasing the model size. These models are all Pareto solutions. Compared to other multi-objective methods like RENA [59], NSGANet [33] and LEMONADE [11], our searched architectures also show superior performance over these methods.

Comparison on Searched Cells. In order to have an explicit understanding of the proposed method, we further visualize the normal and reduction cells searched by CARS and DARTS in Figure 4. The CARS-H and DARTS (second-order) have a similar number of parameters (3.3M), but the CARS-H has higher accuracy compared to DARTS (second-order). It can be found in Figure 4, there are **more parameters in the CARS-H reduction block** for preserving more useful information, and the **size of the normal block of CARS-H is smaller than that of the DARTS** (second-order) to avoid unnecessary computations. The CARS maintains a population that covers a large range of search space.

4.3. Evaluate on ILSVRC2012

We evaluate the transferability of the searched architectures by training them on the ILSVRC2012 dataset. We use 8 Nvidia Tesla V100 to train the models, and the batch size is 640. We train 250 epochs in total. The learning rate is 0.5 with a linear decay scheduler, and we warm up the learning rate for the first five epochs. Momentum is 0.9, and weight decay is $3e-5$. Label smooth is also used with a smooth ratio of 0.1.

The results in Table 2 show the transferability of our searched architectures. Our models cover an extensive range of parameters. The model sizes range from 3.7M to 5.1M, and the FLOPs range from 430 to 590 MFLOPs. For different deploy environments, we can easily select an architecture that satisfies the computing resources. This experiment considers the device-agnostic variables, model size and performance, thus the latencies of CARS A-I are not strictly

²<https://github.com/huawei-noah/CARS>

Table 2. An overall comparison on ILSVRC2012 dataset. The CARS models are the architectures searched on the CIFAR-10 dataset.

Architecture	Top-1 Acc (%)	Top-5 Acc (%)	Params (M)	+ × (M)	Search Cost (GPU days)	Search Method
ResNet50 [22]	75.3	92.2	25.6	4100	-	manual
MorphNet [15]	75.2	-	15.5	3880	-	manual
InceptionV1 [43]	69.8	90.1	6.6	1448	-	manual
MobileNetV2 (1 ×) [39]	72.0	90.4	3.4	300	-	manual
ShuffleNetV2 (2 ×) [34]	74.9	90.1	7.4	591	-	manual
PNAS [30]	74.2	91.9	5.1	588	224	SMBO
AutoSlim [55]	75.4	-	8.3	532	-	greedy
SNAS (mild) [49]	72.7	90.8	4.3	522	1.5	gradient
DARTS [32]	73.3	91.3	4.7	574	4	gradient
PDARTS [5]	75.6	92.6	4.9	557	0.3	gradient
PARSEC [3]	74.0	91.6	5.6	548	1	gradient
ProxylessNAS (GPU) [2]	75.1	92.5	7.1	465	8.3	gradient
FBNet-C [47]	74.9	-	5.5	375	20	gradient
RCNet [50]	72.2	91.0	3.4	294	8	gradient
GDAS [9]	74.0	91.5	5.3	581	0.8	gradient
NASNet-A [61]	74.0	91.6	5.3	564	2000	RL
MNASNet-A1 [44]	75.2	92.5	3.9	312	-	RL
AmoebaNet-A [37]	74.5	92.0	5.1	555	3150	evolution
CARS-A	72.8	90.8	3.7	430	0.4	evolution
CARS-B	73.1	91.3	4.0	463	0.4	evolution
CARS-C	73.3	91.4	4.2	480	0.4	evolution
CARS-D	73.3	91.5	4.3	496	0.4	evolution
CARS-E	73.7	91.6	4.4	510	0.4	evolution
CARS-F	74.1	91.8	4.5	530	0.4	evolution
CARS-G	74.2	91.9	4.7	537	0.4	evolution
CARS-H	74.7	92.2	4.8	559	0.4	evolution
CARS-I	75.2	92.5	5.1	591	0.4	evolution

positive related to the final performance. The latencies are 82.9, 83.3, 83.0, 90.0, 93.8, 92.2, 98.1, 97.2, 100.6 (ms) on HUAWEI P30 Pro.

The CARS-I surpasses PNAS by 1% Top-1 accuracy with the same number of parameters and approximate FLOPs. The CARS-G shows superior results over DARTS by 0.9% Top-1 accuracy with the same number of parameters. Also, CARS-D surpasses SNAS (mild) by 0.6% Top-1 accuracy with the same number of parameters. For different models of NASNet and AmoebaNet, our method also has various models that achieve higher accuracy using the same number of parameters. By using the proposed pNSGA-III, the larger architectures like CARS-I could be protected during architecture optimization stages. Because of the efficient parameter sharing strategy, we could search a set of superior transferable architectures during the one-time search.

For the experiment that considers device-aware variables, *i.e.*, runtime latency and performance, we evaluate the searched architectures on the ILSVRC2012 dataset. The results are shown in Figure 1. The searched architectures cover an actual runtime latency from 40ms to 90ms and surpass the counterparts.

5. Conclusion

The EA-based NAS methods are able to find models with high-performance, but the search time is extremely long because each candidate network is trained separately. In order to make this efficient, we propose a continuous evolution architecture search method, namely, CARS. During evolution, CARS maximally utilizes the learned knowledge in the latest evolution generation, such as architectures and parameters. A SuperNet is constructed with considerable cells and blocks. Individuals are generated through the benchmark operations in the evolutionary algorithm. The non-dominated sorting strategy (pNSGA-III) is utilized to select architectures for updating the SuperNet. Experiments on benchmark datasets show that the proposed CARS can efficiently provide several architectures on the Pareto front. The searched models are superior to the state-of-the-arts in terms of model size/latency and accuracy.

Acknowledgement This work is supported by National Natural Science Foundation of China under Grant No. 61876007, 61872012, Australian Research Council under Project DE180101438, and Beijing Academy of Artificial Intelligence (BAAI).

References

- [1] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Understanding and simplifying one-shot architecture search. *ICML*, 2018.
- [2] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *ICLR*, 2019.
- [3] Francesco Paolo Casale, Jonathan Gordon, and Nicolo Fusi. Probabilistic neural architecture search. *arXiv*, 2019.
- [4] Hanting Chen, Yunhe Wang, Chang Xu, Zhaohui Yang, Chuanjian Liu, Boxin Shi, Chunjing Xu, Chao Xu, and Qi Tian. Data-free learning of student networks. *ICCV*, 2019.
- [5] Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive darts: Bridging the optimization gap for nas in the wild. *arXiv*, 2019.
- [6] Xiangxiang Chu, Bo Zhang, Ruijun Xu, and Jixiang Li. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *arXiv*, 2019.
- [7] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *TEC*, 2014.
- [8] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. *ECCV*, 2018.
- [9] Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four gpu hours. *CVPR*, 2019.
- [10] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *ICLR*, 2018.
- [11] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *ICLR*, 2019.
- [12] Ross Girshick. Fast r-cnn. *ICCV*, 2015.
- [13] Aidan N. Gomez, Ivan Zhang, Kevin Swersky, Yarin Gal, and Geoffrey E. Hinton. Learning sparse networks using targeted dropout. *arXiv*, 2019.
- [14] Xinyu Gong, Shiyu Chang, Yifan Jiang, and Zhangyang Wang. Autogan: Neural architecture search for generative adversarial networks. *ICCV*, 2019.
- [15] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. *arXiv*, 2017.
- [16] Jianyuan Guo, Kai Han, Yunhe Wang, Chao Zhang, Zhaohui Yang, Han Wu, Xinghao Chen, and Chang Xu. Hit-detector: Hierarchical trinity architecture search for object detection. *CVPR*, 2020.
- [17] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv*, 2019.
- [18] Kai Han, Yunhe Wang, Han Shu, Chuanjian Liu, Chunjing Xu, and Chang Xu. Attribute aware pooling for pedestrian attribute recognition. *IJCAI*, 2019.
- [19] Kai Han, Yunhe Wang, Qi Tian, Jianyuan Guo, Chunjing Xu, and Chang Xu. Ghostnet: More features from cheap operations. *arXiv*, 2019.
- [20] Chaoyang He, Haishan Ye, Li Shen, and Tong Zhang. Mile-nas: Efficient neural architecture search via mixed-level reformulation. *CVPR*, 2020.
- [21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. *ICCV*, 2017.
- [22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CVPR*, 2016.
- [23] Chi-Hung Hsu, Shu-Huan Chang, Da-Cheng Juan, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Shih-Chieh Chang. Monas: Multi-objective neural architecture search using reinforcement learning. *arXiv*, 2018.
- [24] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. *CVPR*, 2017.
- [25] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *NIPS*, 2016.
- [26] Ye-Hoon Kim, Bhargava Reddy, Sojung Yun, and Chanwon Seo. Nemo : Neuro-evolution with multiobjective optimization of deep neural network for speed and accuracy. *ICMLW*, 2017.
- [27] Shaohui Lin, Rongrong Ji, Chao Chen, Dacheng Tao, and Jiebo Luo. Holistic cnn compression via low-rank decomposition with knowledge transfer. *T-PAMI*, 2019.
- [28] Shaohui Lin, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, and Baochang Zhang. Accelerating convolutional networks via global & dynamic filter pruning. *IJCAI*, 2018.
- [29] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. Towards optimal structured cnn pruning via generative adversarial learning. *CVPR*, 2019.
- [30] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li-Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *ECCV*, 2018.
- [31] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *ICLR*, 2018.
- [32] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *ICLR*, 2019.
- [33] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. Nsga-net: A multi-objective genetic algorithm for neural architecture search. *GECCO*, 2019.
- [34] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. *ECCV*, 2018.
- [35] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *ICML*, 2018.
- [36] Ruijie Quan, Xuanyi Dong, Yu Wu, Linchao Zhu, and Yi Yang. Auto-reid: Searching for a part-aware convnet for person re-identification. *ICCV*, 2019.
- [37] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *AAAI*, 2019.

- [38] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. *ICML*, 2017.
- [39] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *CVPR*, 2018.
- [40] Mingzhu Shen, Kai Han, Chunjing Xu, and Yunhe Wang. Searching for accurate binary neural architectures. *ICCVW*, 2019.
- [41] Han Shu, Yunhe Wang, Xu Jia, Kai Han, Hanting Chen, Chunjing Xu, Qi Tian, and Chang Xu. Co-evolutionary compression for unpaired image translation. *ICCV*, 2019.
- [42] Dehua Song, Chang Xu, Xu Jia, Yiyi Chen, Chunjing Xu, and Yunhe Wang. Efficient residual dense block search for image super-resolution. *AAAI*, 2020.
- [43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CVPR*, 2015.
- [44] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. *CVPR*, 2018.
- [45] Yunhe Wang, Chang Xu, Chunjing Xu, Chao Xu, and Dacheng Tao. Learning versatile filters for efficient convolutional neural networks. *NIPS*, 2018.
- [46] Yunhe Wang, Chang Xu, Shan You, Dacheng Tao, and Chao Xu. Cnnpack: packing convolutional neural networks in the frequency domain. *NIPS*, 2016.
- [47] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *CVPR*, 2019.
- [48] Lingxi Xie and Alan Yuille. Genetic cnn. *ICCV*, 2017.
- [49] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *ICLR*, 2019.
- [50] Yunyang Xiong, Ronak Mehta, and Vikas Singh. Resource constrained neural network architecture search. *ICCV*, 2019.
- [51] Yixing Xu, Yunhe Wang, Hanting Chen, Kai Han, Chunjing Xu, Dacheng Tao, and Chang Xu. Positive-unlabeled compression on the cloud. *NeurIPS*, 2019.
- [52] Yingzhen Yang, Jiahui Yu, Nebojsa Jojic, Jun Huan, and Thomas S. Huang. Fsnet: Compression of deep convolutional neural networks by filter summary. *ICLR*, 2020.
- [53] Zhaohui Yang, Yunhe Wang, Chuanjian Liu, Hanting Chen, Chunjing Xu, Boxin Shi, Chao Xu, and Chang Xu. Legonet: Efficient convolutional neural networks with lego filters. *ICML*, 2019.
- [54] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *ICML*, 2019.
- [55] Jiahui Yu and Thomas Huang. Autoslim: Towards one-shot architecture search for channel numbers. *arXiv*, 2019.
- [56] Jiahui Yu and Thomas S. Huang. Universally slimmable networks and improved training techniques. *ICCV*, 2019.
- [57] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. *ICLR*, 2019.
- [58] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. *CVPR*, 2018.
- [59] Yanqi Zhou, Siavash Ebrahimi, Sercan Ö Arık, Haonan Yu, Hairong Liu, and Greg Diamos. Resource-efficient neural architect. *arXiv*, 2018.
- [60] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *ICLR*, 2017.
- [61] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *CVPR*, 2018.