Searching by Generating: Flexible and Efficient One-Shot NAS with Architecture Generator

Sian-Yao Huang and Wei-Ta Chu
National Cheng Kung University, Tainan, Taiwan
{P76084245,wtchu}@gs.ncku.edu.tw

Abstract

In one-shot NAS, sub-networks need to be searched from the supernet to meet different hardware constraints. However, the search cost is high and N times of searches are needed for N different constraints. In this work, we propose a novel search strategy called architecture generator to search sub-networks by generating them, so that the search process can be much more efficient and flexible. With the trained architecture generator, given target hardware constraints as the input, N good architectures can be generated for N constraints by just one forward pass without re-searching and supernet retraining. Moreover, we propose a novel single-path supernet, called unified supernet, to further improve search efficiency and reduce GPU memory consumption of the architecture generator. With the architecture generator and the unified supernet, we propose a flexible and efficient one-shot NAS framework, called Searching by Generating NAS (SGNAS). With the pre-trained supernt, the search time of SGNAS for N different hardware constraints is only 5 GPU hours, which is $4N$ times faster than previous SOTA single-path methods. After training from scratch, the top1-accuracy of SGNAS on ImageNet is 77.1%, which is comparable with the SOTAs. The code is available at: https://github.com/eric8607242/SGNAS.

1. Introduction

It is time-consuming and difficult to manually design neural architectures under specific hardware constraints. Neural architecture search (NAS) [30][2][1] aiming at automatically searching the best neural architecture is thus highly demanded. However, how to efficiently and flexibly determine the architectures conforming to various constraints is still very challenging [4].

The earliest NAS methods were developed based on reinforcement learning (RL) [33][1] or the evolution algorithm [30]. However, extremely expensive computation is needed. For example, 2,000 GPU days are needed by an RL method [1], and 3,150 GPU days are needed by the evolution algorithm [30].

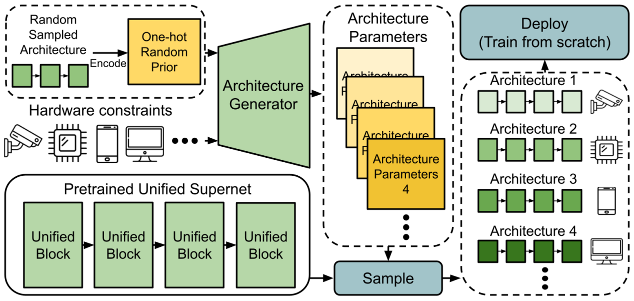


Figure 1. Overview of SGNAS. Given the target hardware constraint as the input, the architecture generator can generate architecture parameters instantly within the inference time of one forward pass. With the generated parameters, the specific architectures can be sampled from the unified supernet.

To improve searching efficiency, one-shot NAS methods [2][26][5][37] were proposed to encode the entire search space into an over-parameterized neural network, called a *supernet*. Once the supernet is trained, all sub-networks in the supernet can be evaluated by inheriting the weights of the supernet without additional training. One-shot NAS methods can be divided into two categories: differentiable NAS (DNAS) and single-path NAS.

In addition to optimizing the supernet, DNAS [16][26][37][38][28] utilizes additional differentiable parameters, called *architecture parameters*, to indicate the architecture distribution in the search space. Because DNAS couples architecture parameters optimization with supernet optimization, for N different hardware constraints, the supernet and the architecture parameters should be trained jointly for N times to find N different best architectures. This makes DNAS methods inflexible.

In contrast, single-path methods [18][10][9][40] decouple supernet training from architecture searching. For supernet training, only a single path consisting of one block in each layer is activated and is optimized in one iteration. The main idea is to simulate discrete neural architectures in the search space and save GPU memory consumption. Once the supernet is trained, different search strategies, like the evolution algorithm [40][18], can be used to search the architecture under different constraints without retraining the supernet. Single-path methods are thus more flexible than DNAS. However, re-executing the search strategy N times for N different constraints is costly and not flexible enough.

On top of one-shot NAS, we especially investigate *efficiency* and *flexibility*. For efficiency, we mean that, when the supernet is available, the time required to search the best architecture for a specific hardware constraint. For flexibility, we mean that, when N different hardware constraints are to be met, how much total time required to search for N best architectures. As a comparison instance, GreedyNAS [40] takes almost 24 GPU hours to search for the best neural architecture under a specific constraint. Totally $24N$ GPU hours are required for N different constraints.

In this work, we focus on improving efficiency and flexibility of the search strategy of the single-path method. The main idea is *searching the best architecture by generating it*. First, we decouple supernet training from architecture searching and train the supernet as a single-path method. After obtaining the supernet, we propose to build an *architecture generator* to generate the best architecture directly. Given a hardware constraint as input, the architecture generator can generate the architecture parameter within the inference time of one forward pass. This method is extremely efficient and flexible. The total search time for various hardware constraints of the architecture generator is only 5 GPU hours. Moreover, we do not need to re-execute search strategies or re-train the supernet once the architecture generator is trained. When N different constraints are to be met, the search strategy only needs to be conducted once, which is more flexible than N searches required in previous single-path methods [10][40][9].

The aforementioned idea is on top of a trained supernet. However, we notice that searching on a single-path supernet still requires a lot of GPU memory and time because of the huge number of supernet parameters and complex supernet structure. Previous single-path NAS methods [18][40][9] determine a block for each layer, and there may be different candidate blocks with various configurations. For example, GreedyNAS [40] has 13 types of candidate blocks for each layer, and thus size of the search space is 13^L, where L denotes the total number of layers in the supernet. Inspired by the fine-grained supernet in AtomNAS [28], we propose a novel single-path supernet called *unified supernet* to reduce GPU memory consumption. In the unified supernet, we only construct a block called *unified block* in each layer. There are multiple sub-blocks in the unified block, and each sub-block can be implemented by different operations. By combining sub-blocks, all configurations can be described in a block. In this way, the number of parameters of the unified supernet is much fewer than previous single-path methods.

The contributions of this paper are summarized as follows. With the architecture generator and the unified supernet, we propose Searching by Generating NAS (SGNAS), which is a flexible and efficient one-shot NAS framework. We illustrate the process of SGNAS in Fig. 1. Given various hardware constraints as the input, the architecture generator can generate the best architecture for different hardware constraints instantly in one forward pass. After training the best architecture from scratch, the evaluation results show that SGNAS achieves 77.1% top-1 accuracy on the ImageNet dataset [14] at around 370M FLOPs, which is comparable with the state-of-the-arts of the single-path methods. Meanwhile, SGNAS outperforms SOTA single-path NAS in terms of efficiency and flexibility.

2. Related Work

Recently, one-shot NAS [26][37][18] has received much attention because of reduced search cost brought by supernet. To futher reduce the search cost, a number of methods have been proposed, which can be roughly divided into two types: efficient NAS and flexible NAS.

2.1. Efficient NAS

For efficiency, many methods were proposed to improve the supernet training strategy or redesign the supernet architecture. Stamoulis et al. [32] proposed a single-path supernet to encode architectures with shared convolutional kernel parameters, which reduce search cost of differential NAS. To reduce the huge cost when training on large-scale datasets, training supernet and searching on proxy datasets like CIFAR10 or part of ImageNet was proposed in [37][36][21][26][39][26]. PC-DARTS [38] only sampled a small part of the supernet for training in each iteration to reduce computation cost. DA-NAS [12] designed a data-adaptive pruning strategy for efficient architecture search.

2.2. Flexible NAS

For flexibility, in OFA [4] a single full network is carefully trained. Sub-networks inherit weights from the network and can be directly deployed without training from scratch. An accuracy predictor is trained after training supernet to guide the process for searching a specialized sub-network. FBNetV3 [13] trained a predictor on a proxy dataset. The accuracy predictor estimates performance of a candidate sub-network. However, it is still time-consuming to train an accuracy predictor.

In this work, we focus on improving the *search strategy* in terms of both efficiency and flexibility. Note that our search strategy can be incorporated with the methods mentioned above.

3. Searching by Generating NAS

3.1. Background

Given a supernet A represented by weights $\boldsymbol{w}$, to find an architecture that achieves the best performance while meeting a specific hardware constraint, we need to find the best sub-network a^* from A which achieves the minimum validation loss $\mathcal{L}_{val}(a, \boldsymbol{w})$. Sampling a from A is a non-differentiable process. To optimize a by the gradient descent algorithm, DNAS [37][5][26] relaxes the problem as finding a set of continuous architecture parameters $\boldsymbol{\alpha}$, and computes the weighted sum of outputs of candidate blocks by the Gumbel softmax function [22]:

$$x_{l+1} = \sum_i m_l^i \cdot b_l^i(x_l), \tag{1}$$

$$m_l^i = \frac{exp(\alpha_l^i + g_l^i/\tau)}{\sum_{k=1}^{K} exp(\alpha_l^k + g_l^k/\tau)}, \tag{2}$$

where x_l is the input tensor of the lth layer, b_l^i is the ith block of the lth layer, and thus $b_l^i(x_l)$ denotes the output of the ith block. The term α_l^i is the weight of the ith block in the lth layer. The term g_l^i is a random variable sampled from the Gumbel distribution $Gumbel(0, 1)$ and τ is the temperature parameter. The value m_l^i is the weight for the output $b_l^i(x_l)$.

After relaxation, DNAS can be formulated as a bi-level optimization:

$$\boldsymbol{\alpha}^* = \min_{\boldsymbol{\alpha}} \mathcal{L}_{val}(\boldsymbol{w}^*, \boldsymbol{\alpha}) \tag{3}$$

$$\text{s.t. } \boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w}} \mathcal{L}_{train}(\boldsymbol{w}, \boldsymbol{\alpha}) \tag{4}$$

where $\mathcal{L}_{train}(\boldsymbol{w}, \boldsymbol{\alpha})$ is the training loss.

Because of the bi-level optimization of $\boldsymbol{w}$ and $\boldsymbol{\alpha}$, the best architecture $\boldsymbol{\alpha}^*$ sampled from the supernet is only suitable to a specific hardware constraint. With this searching process, for N different hardware constraints, the supernet and architecture parameters should be retrained for N times. This makes DNAS less flexible.

In contrast, single-path methods [18][10][9][8] decouple supernet training from architecture searching. For supernet training, only a single path consisting of one block in each layer is activated and is optimized in one iteration to simulate discrete neural architecture in the search space. We can formulate the process as:

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w}} \mathbb{E}_{a\sim\Gamma(A)}(\mathcal{L}_{train}(\boldsymbol{w}(a))) \tag{5}$$

where $\boldsymbol{w}(a)$ denotes the subset of $\boldsymbol{w}$ corresponding to the sampled architecture a, and $\Gamma(A)$ is a prior distribution of $a \in A$. The best weights $\boldsymbol{w}^*$ to be determined are the ones yielding the minimum expected training loss. After training, the supernet is treated as a performance estimator to all architectures in the search space. With the pretrained supernet weights w^*, we can search the best architecture a^*:

$$a^* = \operatorname*{argmin}_{a\in A} \mathcal{L}_{val}(\boldsymbol{w}^*(a)). \tag{6}$$

Single-path methods are more flexible than DNAS, because supernet training and architecture search are decoupled. Once the supernet is trained, for N different constraints, only architecture search should be conducted for N times.

In this work, we propose to decouple supernet training from architecture searching and train supernet as in single-path NAS (Eq. (5)). After supernet training, we search the best architecture by the gradient descent algorithm as in DNAS (Eq. (3)). Instead of training architecture parameters for one specific hardware constraint, we propose a novel search strategy called *architecture generator* to largely increase flexibility and efficiency.

3.2. Architecture Generator

3.2.1 Essential Idea

Given the target hardware constraint C, the architecture generator can generate the best architecture parameters for the hardware constraint C. The process of the architecture generator can be described as $\boldsymbol{\alpha} = G(C)$ such that $Cost(\boldsymbol{\alpha}) < C$. With the architecture generator G, the objective function of the architecture searching in Eq. (3) can be reformulated as :

$$G^* = \min_{G} \mathcal{L}_{val}(\boldsymbol{w}^*, \boldsymbol{\alpha}). \tag{7}$$

To make G^* generate the best architecture parameters for different hardware constraints accurately, we propose the hardware constraint loss $\mathcal{L}_C$ as:

$$\mathcal{L}_C(\boldsymbol{\alpha}, C) = (Cost(\boldsymbol{\alpha}) - C)^2, \tag{8}$$

where the cost yielded by the generated architecture $Cost(\boldsymbol{\alpha})$ is estimated by:

$$Cost(\boldsymbol{\alpha}) = \sum_l \sum_i m_l^i \cdot Cost(b_l^i). \tag{9}$$

The term $Cost(b_l^i)$ is the constant cost of the ith block in the l layer and m_l^i is the weight of different blocks described in Eq. (2). The cost $Cost(\boldsymbol{\alpha})$ is differentiable with respect to m_l^i and $\boldsymbol{\alpha}$, similarly in [37][21]. Note that Eq. (9) is also highly correlated with latency, as mentioned in [37] and [21]. By combining the hardware constraint loss $\mathcal{L}_C$ and the cross entropy loss $\mathcal{L}_{val}$ defined in Eq. (7), the overall loss of the architecture generator $\mathcal{L}_G$ is:

$$\mathcal{L}_G = \mathcal{L}_{val}(\boldsymbol{w}^*, \boldsymbol{\alpha}) + \lambda\mathcal{L}_C(\boldsymbol{\alpha}, C). \tag{10}$$

where λ is a hyper-parameter to trade-off the validation loss and hardware constraint loss.

3.2.2 Accurate Generation with Random Prior

In practice, we found that the architecture generator easily overfits to a specific hardware constraint. The reason is that it is too difficult to generate complex and high-dimensional architecture parameters based on a given simple integer hardware constraint C.

To address this issue, a prior is given as input to stabilize the architecture generator. We randomly sample a neural architecture from the search space, and encode the neural architecture into a one-hot vector to be the prior knowledge of architecture parameters. We name it as a random prior $B = B_1, ..., B_L$. Formally, $B_l = one\_hot(a_l), l = 1, ..., L$, where a_l is the lth layer of the neural architecture a randomly sampled from A, and L is the total number of layers in the supernet. With the random prior, the architecture generator is to learn the residual from the random prior to the best architecture parameters, making training architecture generator more stable and accurately (blue line in Fig. 7(a)), and the process of the architecture generator can be reformulated as $\boldsymbol{\alpha} = G(C, B)$ such that $Cost(\boldsymbol{\alpha}) < C$.

3.2.3 The Generator Training Algorithm

We illustrate the algorithm of architecture generator training in Algorithm 1. In each iteration, given the target constraint and the random prior, the architecture generator can generate the architecture parameters $\boldsymbol{\alpha}$ (as illustrated in Fig. 1). With $\boldsymbol{\alpha}$, the corresponding cost $C_{\boldsymbol{\alpha}}$ can be calculated by Eqn. (9). We can predict $\hat{y}$ based on the pretrained supernet N with $\boldsymbol{\alpha}$. The total loss is given by Eqn. (10). No matter what constraint is given, the architecture generator generates architecture parameters to get the best prediction results. Therefore, training the generator is equivalent to searching the best architectures for various constraints in the proposed SGNAS.

3.2.4 The Architecture of the Generator

Fig. 2 illustrates the architecture of the generator. We set the channel size of all convolutional layers as 32 and set the stride as 1, making sure the output shape same as the shape of the random prior. Please refer to supplementary materials for detailed configurations of the architecture generator and random prior representations.

3.3. Unified Supernet

Previous single-path NAS [10][9][18][40] adopts the MobilenetV2 inverted bottleneck [31] as the basic building block. Given the input tensor X, the corresponding output Y_{out} is obtained by

$$Y_{out} = P^{c_3,c_2}(D_{K\times K}(P^{c_2,c_1}(X))), \tag{11}$$

Algorithm 1 Training Architecture Generator

```
Require: $B$: Random prior; $N$: Unified supernet; $G$: Generator; $[C_L, C_H]$: Pre-define hardware constraint interval; $D_{val}$: Validation dataset; $T$: Max iterations;
 1: for $t = 1, ..., T$ do;
 2:     Get a data batch $X$ and $y$ from $D_{val}$
 3:     Randomly sample $C_{target}$ from $[C_L, C_H]$
 4:     $\boldsymbol{\alpha} = G(B, C_{target})$
 5:     $C_{\boldsymbol{\alpha}} = Cost(\boldsymbol{\alpha})$
 6:     $\hat{y} = N(X, \boldsymbol{\alpha})$
 7:     $\mathcal{L}_{total} = \mathcal{L}_{val}(y, \hat{y}) + \lambda\mathcal{L}_C(C_{target}, C_{\boldsymbol{\alpha}})$
 8:     Calculate gradients from $\mathcal{L}_{total}$
 9:     Update $G$ from gradients
10: end for
```

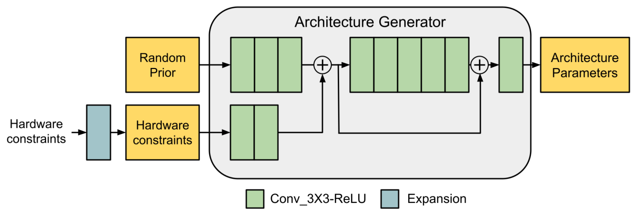


Figure 2. Structure of the architecture generator. Given the input target hardware constraint, the expansion layer expands the input to a tensor with shape same as the random prior.

where P^{c_j,c_i} denotes the pointwise convolution with the input channel size c_i and output channel size c_j, $D_{K\times K}$ denotes the depthwise convolution with $K \times K$ kernel size. Eq. (11) represents that X of c_1 channels is first expanded to a tensor of c_2 channels, which can be described as $c_2 = e \times c_1$, and then a depthwise convolution is conducted. The term e denotes the expansion rate of the inverted bottleneck. After that, the tensor of c_2 channels is embedded into the output tensor of c_3 channels. Because one basic building block can only represent one configuration with one kernel size and one expansion rate, previous single-path NAS [5][9][10] needs to construct blocks of various configurations in each layer, which leads an exponential increase in parameter numbers and complexity of the supernet.

In this work, we propose a novel single-path supernet called *unified supernet* to improve efficiency and flexibility of the architecture generator. The only type of block, i.e., *unified block*, is constructed in each layer. The unified block is built with only the maximum expansion rate e_{max}, i.e., $c_2 = e_{max} \times c_1$.

Fig. 3 illustrates the idea of a unified block. To make the unified block represent all possible configurations, we replace the depthwise convolution D by e_{max} sub-blocks, and each sub-block can be implemented by different operations or skip connection. The output tensor of the first pointwise convolution Y_1 is equally split into e_{max} parts,

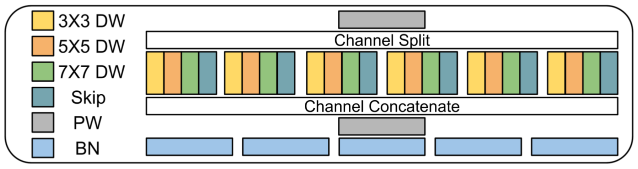


Figure 3. Illustration of a unified block. Given the input, the first pointwise convolution expands the input channel size to e_{max} times. The channel split layer splits the tensor into e_{max} parts and feeds to each sub-block, respectively.

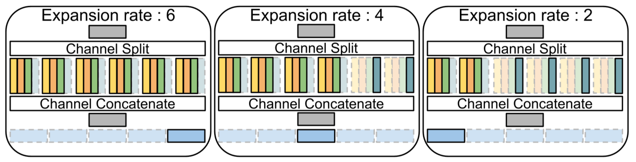


Figure 4. Different expansion rates can be simulated by sub-blocks with different operations. For example, the expansion rate 6 can be simulated if no skip connection is implemented, and the expansion rate 2 can be simulated if four skip connections are implemented.

$Y_{1,1}, Y_{1,2}, ..., Y_{1,e_{max}}$. With the sub-blocks d_i and split tensors $Y_{1,i}$, we can reformulate $D_{K\times K}$ in Eq. (11) as:

$$d_1(Y_{1,1}) \circ d_2(Y_{1,2}) \circ \cdots \circ d_{e_{max}}(Y_{1,e_{max}}), \tag{12}$$

where $\circ$ denotes the channel concatenation function.

With the sub-blocks implemented by different operations, we can simulate blocks with various expansion rates, as shown in Fig. 4. The unified supernet thus can significantly reduce the parameters and GPU memory consumption. It is interesting that the MixConv in MixNet [35] is a special case of our search space if different sub-blocks are implemented by different kernel sizes.

3.3.1 Large Variability of BNs Statistics

As in [28], [41], and [3], we suffer from the problem of unstable running statistics of batch normalization (BN). In the unified supernet, because one unified block would represent different expansion rates, the BN scales change more dramatically during training. To address the problem, BN recalibration [28][41][3] is used to recalculate the running statistics of BNs by forwarding thousands of training data after training. On the other hand, shadow batch normalization (SBN) [8] or switchable batch normalization [42] are used to stabilize BN. In this work, we utilize SBN to address the large variability issue, as illustrated in Fig. 4. In our setting, there are five different expansion rates, i.e., 2, 3, 4, 5, and 6. We thus take five BNs after the second pointwise convolution block to capture the BN statistics for different expansion rates. With SBN, we can capture different statistics and make supernet training more stable.

3.3.2 Architecture Redundancy

Denote two sub-blocks as b_1 and b_2. In the unified supernet, for example, the case of b_1 using 3×3 kernel size and b_2 using skip connection is distinct from the case of b_1 using skip connnection and b_2 using 3×3 kernel size. However these two cases actually correspond to the same sub-network, and thus the architecture redundancy problem arises. This redundancy makes the unified supernet more complex and hard to train. To address this issue, we force that skip connection can only be used in sub-blocks with higher index. For example, if we want to train a unified block with expansion rate 3, only the last three sub-blocks can be skip connection. We call this strategy forced sampling (FS). Please refer to supplementary materials for details of architecture redundancy and forced sampling.

4. Experiments

4.1. Experimental Settings

We adopt the macro structure of supernet (e.g., channel size of each layer and layer number) same as [10] and [5], but utilize the proposed unified blocks to reduce GPU memory consumption and the number of parameters. Each sub-block in the unified blocks can be realized based on convolutional kernel sizes 3, 5, or 7, or the skip connection. We set the minimum and maximum expansion rates as 2 and 6, respectively. The size of our search space is 80^{19}. Please refer to supplementary materials for more details.

For experiments on the ImageNet dataset [14], we train the unified supernet for 50 epochs using batch size 256 and adopt the stochastic gradient descent optimizer. The learning rate is decayed with the cosine annealing strategy [27] from the initial value 0.045. After supernet training, the architecture generator is trained for 50 epochs by the Adam optimizer with the learning rate 0.001. After searching/generating the best architecture under hardware constraints, we adopt the RMSProp optimizer with 0.9 momentum [33] to train the searched architecture from scratch. Learning rate is increased from 0 to 0.16 in the first 5 epochs with batch size 256, and then decays 0.03 every 3 epochs.

4.2. Experiments on ImageNet

4.2.1 Comparison with Baselines

Li and Talwalkar [23] presented that a random search approach usually achieves satisfactory performance. To make comparison, we randomly select 1,000 candidate architectures with FLOPs under 320 millions (320M) from the unified supernet and pick the architecture yielding the highest top-1 accuracy, as mentioned in [23]. Besides, we also search the network with FLOPs under 320M by the evolution algorithm [18] as another baseline.

Table 1. Performance comparison with baselines.

Search Strategy	Search Time (GPU hrs)	FLOPs (M)	Top-1 (%)
Random search	$34N$	322	74.63
Evolution algorithm	$34N$	318	74.67
SGNAS	**5**	324	**74.87**

Table 2. Comparison with the SOTAs for different hardware constraints. †: training with AutoAugment [11]. ‡: searching on a proxy dataset. The unit of search time and train time is GPU hours.

Method	FLOPs (M)	Top-1 (%)	Train time	Search time
MobileNetV2 [31]	300	72.0	–	–
EfficientNet B0 [34]	390	76.3	–	–
MixNet-M [35]	360	77.0	–	–
MixPath-A [8]	349	76.9	240	–
AtomNAS-C [28]	363	77.6	0	$816N$
PC-DARTS [38]	597	75.8	0	$91N$
ScarletNAS-A [9]	365	76.9	240	$48N$
GreedyNAS-A [40]	366	77.1	168	$\sim 24N$
SGNAS-A (Ours)	373	77.1	280	**5**
FBNetV2-L1 [36]	325	77.2	0	$600N^\ddagger$
Proxyless-R [5]	320	74.6	0	$200N$
FairNAS-C [10]	325	$76.7^\dagger$	240	$48N$
ScarletNAS-B [9]	329	76.3	240	$48N$
SPOS [18]	326	74.5	288	$\sim 24N$
GreedyNAS-B [40]	324	76.8	168	$\sim 24N$
SGNAS-B (Ours)	326	76.8	280	**5**
MobileNetV3-L [19]	219	75.2	–	–
ScarletNAS-C [9]	280	75.6	240	$48N$
GreedyNAS-C [40]	284	76.2	168	$\sim 24N$
SGNAS-C (Ours)	281	76.2	280	**5**

Table 1 shows the comparison results. As can be seen, with around 320M FLOPs, the proposed SGNAS achieves the highest top-1 accuracy. Both baselines take around 34 GPU hours to complete the search. For N different hardware constraints, the search strategy should be re-executed for N times, and the search time of each of two baselines is $34N$ GPU hours totally. In contrast, SGNAS only takes 5 GPU hours totally for N different hardware constraints, which is much more efficient and flexible than the baselines.

4.2.2 Comparison with SOTAs

This section is dedicated to compare with various SOTA one-shot NAS methods that utilize the augmented techniques (e.g., Swish activation function [29] and Squeeze-and-Excitation [20]). We directly modify the searched architecture by replacing all ReLU activation with H-Swish [19] activation and equip it with the squeeze-and-excitation module as in AtomNAS [28].

For comparison, similar to the settings in ScarletNAS [9] and GreedyNAS [40], we search architectures under 275M, 320M, and 365M FLOPs, and denote the searched architecture as SGNAS-C, SGNAS-B, and SGNAS-A, respectively. The comparison results are shown in Table 2. The column "Train time" denotes that the time needed to train the supernet, and the column "Search time" denotes that the time needed to search the best architecture based on the pre-trained supernet. Because DNAS couples architecture searching with supernet optimization, we list the time needed for the entire pipeline in the "Search time" column. As can be seen, our SGNAS is competitive with SOTAs in terms of top-1 accuracy under different FLOPs. For example, SGNAS-A achieves 77.1% top-1 accuracy, which outperforms ScarletNAS [9] by 0.2%, outperforms MixNet-M [35] by 0.1%, outperforms MixPath-A [8] by 0.2%, and is comparable with GreedyNAS-A [40].

More importantly, SGNAS achieves much higher search efficiency. With the architecture generator and the unified supernet, even for N different architectures under N different hardware constraints, totally only 5 GPU hours are needed for SGNAS on a Tesla V100 GPU. However, FairNAS [10], GreedyNAS [40], and ScarletNAS [9] need $48N$, $24N$, and $48N$ GPU hours, respectively, because of the cost of re-executing search. Supernet retraining is needed for FBNetV2 [36] and AtomNAS [28], which makes search very inefficient.

Note that after finding the best architecture, training from scratch is required in most methods in Table 2 (including SGNAS, except for AtomNAS [28]). However, training a supernet that can be directly deployed to many constraints (like AtomNAS) needs expensive computation. Even with the time for training from scratch, SGNAS is still more efficient and flexible than AtomNAS.

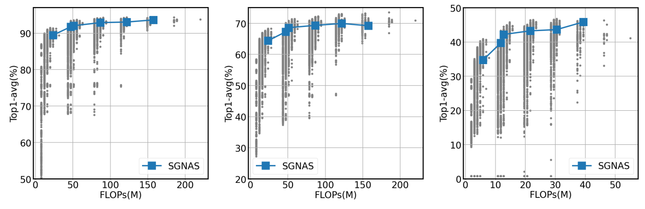
4.3. Experiments on NAS-Bench-201

To demonstrate efficiency and robustness of SGNAS more fairly, we evaluate it based on a NAS benchmark dataset called NAS-Bench-201 [17]. NAS-Bench-201 includes 15,625 architectures in total. It provides full information of the 15,625 architectures (e.g., top-1 accuracy and FLOPs) on CIFAR-10, CIFAR-100, and ImageNet-16-120 datasets [7], respectively.

Based on the search space defined by NAS-Bench-201, we follow SETN [15] to train the supernet by uniform sampling. After that, the architecture generator is applied to search architectures on the supernet. We search based on the CIFAR-10 dataset and look up the ground-truth performance of the searched architectures on CIFAR-10, CIFAR-100, and ImageNet-16-120 datasets, respectively. This process is run for three times, and the average performance is calculated as in Table 3. We see that the architectures searched by SGNAS outperform previous methods on both CIFAR-10 and ImageNet16-120. It is worth noting that,

Table 3. Performance comparison on different datasets in the NAS-Bench-201 benchmark.

Method	Search Time (GPU hrs)	CIFAR-10 Val	CIFAR-10 Test	CIFAR-100 Val	CIFAR-100 Test	ImageNet-16-120 Val	ImageNet-16-120 Test
optimal	N/A	91.61	94.37	73.49	73.51	46.77	47.31
RSPS [23]	2.6	84.16±1.69	87.66±1.69	59.00±4.60	58.33±4.34	31.56±3.28	31.14±3.88
DARTS [26]	2.2N	39.77±0.00	54.30±0.00	15.03±0.00	15.61±0.00	16.43±0.00	16.32±0.00
SETN [15]	7.9N	82.25±5.17	86.19±4.63	56.86±7.59	56.87±7.77	32.54±3.63	31.90±4.07
GDAS [16]	6.6N	90.00±0.21	93.51±0.13	**71.14±0.27**	**70.61±0.26**	41.70±1.26	41.84±0.90
SGNAS (Ours)	**2.5**	**90.18±0.31**	**93.53±0.12**	70.28±1.2	70.31±1.09	**44.65±2.32**	**44.98±2.10**



(a) (b) (c)

Figure 5. Search results of SGNAS on the CIFAR10, CIFAR100, and ImageNet16-120 datasets. (a) Result on CIFAR-10; (b) Result on CIFAR100; (c) Result on ImageNet16-120.

Table 4. Performance comparison on the COCO object detection. †: Our implementation result. * reported in [9][10].

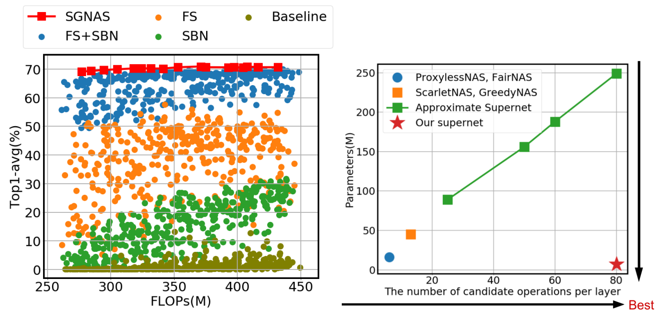
Model	FLOPs(M)	Top-1 (%)	mAP (%)
MobileNetV2* [31]	300	72.0	28.3
MixNet-M* [35]	360	77.0	31.3
FairNAS-A* [10]	392	77.5	32.4
Scarlet-A* [9]	365	76.9	31.4
MobileNetV2†	300	72.0	29.4
SGNAS-A (Ours)	373	77.1	33.9

with the supernet training strategy same as SETN [15], our result greatly surpasses SETN [15] on all three datasets. Moreover, the required search time of SGNAS is only 2.5 GPU hours even for N different hardware constraints.

We show the 15,625 architectures in NAS-Bench-201 on each dataset as gray dots in Fig. 5, and draw the architectures searched by the architecture generator under different FLOPs as blue rectangles. After searching once, the architecture generator can generate all blue rectangles directly without re-searching. Moreover, various generated architectures approach the best among all choices.

4.4. Performance on Object Detection

To verify the transferability of SGNAS on object detection, we adopt the RetinaNet [24] implemented in MMDetection [6] to do object detection, but replace its backbone by the network searched by SGNAS. The models are trained and evaluated on the MS COCO dataset [25] (train2017 and val2017, respectively) for 12 epochs with batch size 16 [10][9]. We use the SGD optimizer with 0.9 momentum and 0.0001 weight decay. The initial learning rate is 0.01, and is multiplied by 0.1 at epochs 8 and 11. Table 4 shows that SGNAS has better transferability than the baselines, especially in terms of mAP.



(a) (b)

Figure 6. (a) Top-1 validation accuracy of randomly sampled architectures (blue dots) and the architectures searched by SGNAS (red rectangles). Performance of other variants is also shown. (b) The relationship between the number of supernet parameters and the number of candidate operations in each layer.

Table 5. Comparison in terms of of GPU memory consumption and search time of the architecture generator.

Unified Supernet	Search Space	Batch Size	Search Time (GPU hrs)	Memory Cost (GPU)
×	6^{19}	32 128	11	10.5GB 40 GB
✓	80^{19}	32 128	**5**	**9GB** **28GB**

5. Ablation Studies

5.1. Analysis of SGNAS

In Fig. 6(a), we randomly sample 360 architectures from the search space and illustrate the corresponding top-1 validation accuracies as blue dots. Moreover, we draw the architectures searched by SGNAS under different hardware constraint as red rectangles. As can be seen, the architectures searched by SGNAS are almost always the best.

5.2. Analysis of Unified Supernet

5.2.1 Efficiency of Unified Supernet

To show efficiency of the proposed unified supernet, we report the relationship between the total number of parameters in the (unified) supernet and the number of candidate operations per layer in Fig. 6(b). For fair comparison, we calculate the number of parameters of different supernets all based on 19 layers. As can be seen, the number of possible operations of the unified supernet in each layer is 7 times larger than GreedyNAS [40] and ScarletNAS [9], but the number of parameters needed to represent this unified supernet is only 1/6 times of them. The number of possible operations is 13 times larger than FairNAS [10] and ProxylessNAS [5], but the number of supernet parameters for the unified supernet is only half of them. To compare under the same size of search space, we estimate the number of required supernet parameters in previous single-path methods [10][40][9][5] when the number of possible operations in each layer increases to 25, 50, 60, and 80, and show them by green squares. Fig. 6(b) shows that the required parameters are significantly boosted when the number of possible operations increases, while the unified supernet avoids this intractability. Under the same size of search space, the number of needed parameters to represent the unified supernet is only 1/35 times of estimated supernets.

Table 5 shows the comparison in terms of GPU memory consumption and search time of the architecture generator when it works based on the unified supernet or based on the previous single-path supernet [10][5]. Based on the unified supernet, the GPU memory consumption reduces to 12 GB and the search time is only 0.45 times of that based on the previous supernet.

5.2.2 Training Stabilization

Although the unified supernet largely reduces supernet parameters, the large search space makes the supernet hard to train. To study the effect of force sampling (FS) and shadow batch normalization (SBN) [8] on supernet training, we train the supernet based on different settings, including baseline, with FS, with SBN, and with both FS and SBN. After training the supernet, we randomly sample 360 architectures from the search space and show the corresponding top-1 accuracies in Fig. 6(a). Without FS and SBN, because of large variability and complex architecture, the baseline supernet is hard to train. After utilizing SBN, variability can be well characterized, and the performance becomes more stable. After applying FS, complexity of the supernet is greatly reduced by reducing architecture redundancy. Performance is largely boosted when redundancy is reduced. With both FS and SBN, the unified supernet can more consistently represent architectures with better performance.

5.3. Study of Random Priors

To enable the generator to generate architectures under various hardware constraints accurately, random prior is given as the input of the generator. In Fig. 7(a), we show the correlation between the target FLOPs and the FLOPs of the generated architectures. With the random prior, the generator can generate architectures much more accurately. With the random prior, the Kendall tau correlation between the target FLOPs and the generated is 1, while the Pearson correlation is 0.99, which are significantly positive.

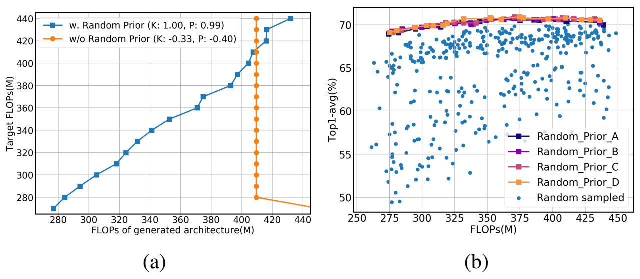


Figure 7. (a) The relationship between target FLOPs and the FLOPs of generated architecture. (b) Performance of the architectures randomly sampled from the unified supernet (blue dots) and those generated by the architecture generator trained based on four random priors.

We randomly sample four sub-networks A, B, C, and D from the unified supernet as four priors to train the architecture generator. Inherited from the weights of the unified supernet, the top-1 validation accuracy of these four sub-networks are 58.02%, 63.36%, 66.48%, and 68.48%, respectively. Fig. 7(b) shows that, no matter starting from good priors or bad priors, the corresponding trained architecture generators are able to generate the architecture yielding the best performance. This shows that random priors are not to improve the top-1 accuracy, but to give reasonable priors to make the architecture generator generate good architectures under the target constraints.

6. Conclusion

To improve efficiency and flexibility of finding best subnetworks from the supernet under various hardware constraints, we propose the idea of architecture generator that searches the best architecture by generating it. This approach is very efficient and flexible for that only one forward pass is needed to generate good architectures for various constraints, comparing to previous one-shot NAS methods. To ease GPU memory consumption and boost searching, we propose the idea of unified supernet which consists of a stack of unified blocks. We show that the proposed one-shot framework, called SGNAS (searching by generating NAS), is extremely efficient and flexible by comparing with state-of-the-art methods. We further comprehensively investigate the impact of architecture generator and unified supernet from multiple perspectives. Please refer to supplementary materials for the limitation of SGNAS.

Acknowledgement This work was funded in part by Qualcomm through a Taiwan University Research Collaboration Project and in part by the Ministry of Science and Technology, Taiwan, under grants 108-2221-E-006-227-MY3, 107-2923-E-006-009-MY3, and 109-2218-E-002-015.

References

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *Proceedings of International Conference on Learning Representations*, 2017. 1

[2] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Understanding and simplifying one-shot architecture search. In *Proceedings of International Conference on Machine Learning*, pages 550–559, 2018. 1

[3] Maxim Berman, Leonid Pishchulin, Ning Xu, Matthew B. Blaschko, and Gerard Medioni. AOWS: Adaptive and optimal network width search with latency constraints. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 5

[4] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *Proceedings of International Conference on Learning Representations*, 2020. 1, 2

[5] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *Proceedings of International Conference on Learning Representations*, 2019. 1, 3, 4, 5, 6, 8, 11

[6] Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, Wansen Feng, Ziwei Liu, Jiarui Xu, Zheng Zhang, Dazhi Cheng, Chenchen Zhu, Tianheng Cheng, Qijie Zhao, Buyu Li, Xin Lu, Rui Zhu, Yue Wu, Jifeng Dai, Jingdong Wang, Jianping Shi, Wanli Ouyang, Chen Change Loy, and Dahua Lin. MMDetection: Open mmlab detection toolbox and benchmark. *arXiv preprint arXiv:1906.07155*, 2019. 7

[7] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the CIFAR datasets. *CoRR*, abs/1707.08819, 2017. 6

[8] Xiangxiang Chu, Xudong Li, Yi Lu, Bo Zhang, and Jixiang Li. Mixpath: A unified approach for one-shot neural architecture search. *arXiv preprint arXiv:2001.05887*, 2020. 3, 5, 6, 8

[9] Xiangxiang Chu, Bo Zhang, Jixiang Li, Qingyuan Li, and Ruijun Xu. Scarlet-nas: Bridging the gap between scalability and fairness in neural architecture search. *arXiv preprint arXiv:1908.06022*, 2019. 1, 2, 3, 4, 6, 7, 8, 11

[10] Xiangxiang Chu, Bo Zhang, and Ruijun Xu. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *arXiv preprint arXiv:1907.01845*, 2019. 1, 2, 3, 4, 5, 6, 7, 8, 11

[11] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2019. 6

[12] Xiyang Dai, Dongdong Chen, Mengchen Liu, Yinpeng Chen, and Lu Yuan. Da-nas: Data adapted pruning for efficient neural architecture search. In *Proceedings of European Conference on Computer Vision*, 2020. 2

[13] Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, and Joseph E. Gonzalez. Fbnetv3: Joint architecture-recipe search using neural acquisition function, 2020. 2

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2009. 2, 5, 11

[15] Xuanyi Dong and Yi Yang. One-shot neural architecture search via self-evaluated template network. In *Proceedings of IEEE International Conference on Computer Vision*, pages 3681–3690, 2019. 6, 7

[16] Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four gpu hours. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 1761–1770, 2019. 1, 7

[17] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *Proceedings of International Conference on Learning Representations*, 2020. 6

[18] 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. In *Proceedings of European Conference on Computer Vision*, 2020. 1, 2, 3, 4, 5, 6

[19] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. In *Proceedings of IEEE/CVF International Conference on Computer Vision*, 2019. 6

[20] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2018. 6

[21] Yibo Hu, Xiang Wu, and Ran He. Tf-nas: Rethinking three search freedoms of latency-constrained differentiable neural architecture search. In *Proceedings of European Conference on Computer Vision*, 2020. 2, 3

[22] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax, 2016. 3, 11

[23] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2019. 5, 7

[24] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. In *Proceedings of IEEE International Conference on Computer Vision*, pages 2999–3007, 2017. 7

[25] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollar. Microsoft coco: Common objects in context, 2014. 7

[26] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *Proceedings of International Conference on Learning Representations*, 2019. 1, 2, 3, 7

[27] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic gradient descent with warm restarts, 2016. 5

[28] Jieru Mei, Yingwei Li, Xiaochen Lian, Xiaojie Jin, Linjie Yang, Alan Yuille, and Jianchao Yang. AtomNAS: Fine-grained end-to-end neural architecture search. In *Proceedings of International Conference on Learning Representations*, 2020. 1, 2, 5, 6

[29] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2018. 6

[30] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of AAAI Conference on Artificial Intelligence*, page 4780–4789, 2019. 1

[31] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2018. 4, 6, 7, 11

[32] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. In *arXiv preprint arXiv:1904.02877*, 2019. 2

[33] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2019. 1, 5

[34] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proceedings of International Conference on Machine Learning*, volume 97, pages 6105–6114, 2019. 6

[35] Mingxing Tan and Quoc V. Le. Mixconv: Mixed depthwise convolutional kernels. In *Proceedings of British Machine Vision Conference*, 2019. 5, 6, 7

[36] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, Peter Vajda, and Joseph E. Gonzalez. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 2, 6

[37] 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. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2019. 1, 2, 3

[38] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. Pc-darts: Partial channel connections for memory-efficient architecture search. In *Proceedings of International Conference on Learning Representations*, 2020. 1, 2, 6

[39] Zhaohui Yang, Yunhe Wang, Xinghao Chen, Boxin Shi, Chao Xu, Chunjing Xu, Qi Tian, and Chang Xu. Cars: Continuous evolution for efficient neural architecture search. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 2

[40] Shan You, Tao Huang, Mingmin Yang, Fei Wang, Chen Qian, and Changshui Zhang. Greedynas: Towards fast one-shot nas with greedy supernet. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 1, 2, 4, 6, 8, 11

[41] Jiahui Yu and Thomas S. Huang. Universally slimmable networks and improved training techniques. In *Proceedings of IEEE International Conference on Computer Vision*, 2019. 5

[42] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. In *Proceedings of International Conference on Learning Representations*, 2019. 5

A. More Details of Experimental Settings

A.1. Dataset

We perform all experiments based on the ImageNet dataset [14]. Same as the settings in [5][40][9], we randomly sample 50,000 images (50 images for each class) from the training set as our validation set, and the rest is kept as the training set. The original validation set is taken as the test set to measure the final performance of each model. The resolution of input images is 224×224.

A.2. Supernet Training

We train the unified supernet for 50 epochs using batch size 256 and adopt the stochastic gradient descent optimizer with a momentum of 0.9 and weight decay of 4×10^{-5}. The learning rate is decayed based on the cosine annealing strategy from initial value 0.045. We train the unified supernet with strict fairness [10] so that each operation in all sub-blocks and each expansion rate are trained fairly.

A.3. Generator Training

After supernet training, the architecture generator is trained for 50 epochs using batch size 128 by the Adam optimizer with the learning rate 0.001, momentum (0.5, 0.999), and weight decay 0. The temperature τ of Gumbel Softmax [22] in Eq. (2) is initially set to 5.0 and annealed by a factor of 0.95 for each epoch. The trade-off parameter λ in Eq. (11) is set to 0.0003 in our experiment.

B. Details of Search Space

The marco-architecture of our unified supernet is shown in Table 6.

Table 6. Macro-architecture of the search space. MBConv 3 × 3 denotes MobileNetV2 [31] block with kernel size 3. Column-C denotes the number of output channel of a block. Column-N denotes the number of the blocks. Column-S denotes the stride of the first block when stacked for multiple blocks. Column-E denotes the expansion rate of the blocks, and the tuples of three values represent the lowest value, highest value, and steps between options (low, high, steps).

Input shape	Block	C	N	S	E
$224^2 \times 3$	Conv 3×3	32	1	2	-
$112^2 \times 32$	MBConv 3×3	16	1	1	1
$112^2 \times 16$	Unified Block	32	2	2	(2, 6, 1)
$56^2 \times 32$	Unified Block	40	4	2	(2, 6, 1)
$28^2 \times 40$	Unified Block	80	4	2	(2, 6, 1)
$14^2 \times 80$	Unified Block	96	4	1	(2, 6, 1)
$14^2 \times 96$	Unified Block	192	4	2	(2, 6, 1)
$7^2 \times 192$	Unified Block	320	1	1	(2, 6, 1)
$7^2 \times 320$	Unified Block	1280	1	1	(2, 6, 1)
$7^2 \times 1280$	Avg pool	-	1	1	-
1280	FC	1000	1	-	-

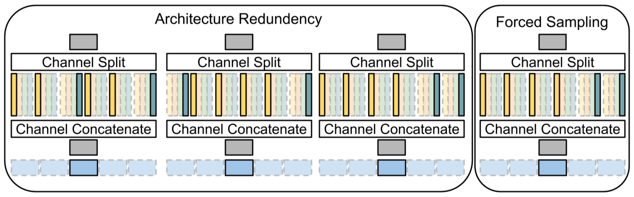


Figure 8. Illustration of architecture redundancy and forced sampling (FS).

C. More Details of Architecture Generator

A random prior is encoded into a one-hot format and then is reshaped into the shape of architecture parameters to be generated. The output of the architecture generator is a parameter map with size $LayerSize \times OperationNumber$. Motivated by generative adversarial networks, we reshape a random prior so that its shape is the same as the output map. We then feed it to the architecture generator, where we can apply 2D convolution with stride 1 for processing. Without carefully tuning convolution parameters, we can ensure that shape of the output map fits different search spaces. This design is to make the generator easily adapt to different settings. We have experimented with various structures for the architecture generator (e.g., fully connected) and found that convolutional layers yield reliable results.

D. More Details of Architecture Redundancy

We illustrate architecture redundancy in the left of Fig. 8 and forced sampling (FS) in the right of Fig. 8. In the four unified blocks in Fig. 8, four depthwise convolution with kernel size 3×3 and two skip connections are used in different sub-blocks. However, the three situations on the left of Fig. 8 are treated as different because of different arrangements. With FS, we enforce arrangement of the operations to be unique, and thus only the unified block on the right of Fig. 8 can be sampled.

E. Visualization of Searched Architectures

We visualize SGNAS-A, SGNAS-B, and SGNAS-C in Fig. 9. Besides, we also visualize the architectures searched by SGNAS under different hardware constraints in Fig. 10. It is interesting that even if the target hardware constraint is low (e.g., 280M), the expansion rate simulated by sub-blocks is still high in some layers (e.g., layer 1, layer 7, and layer 19).

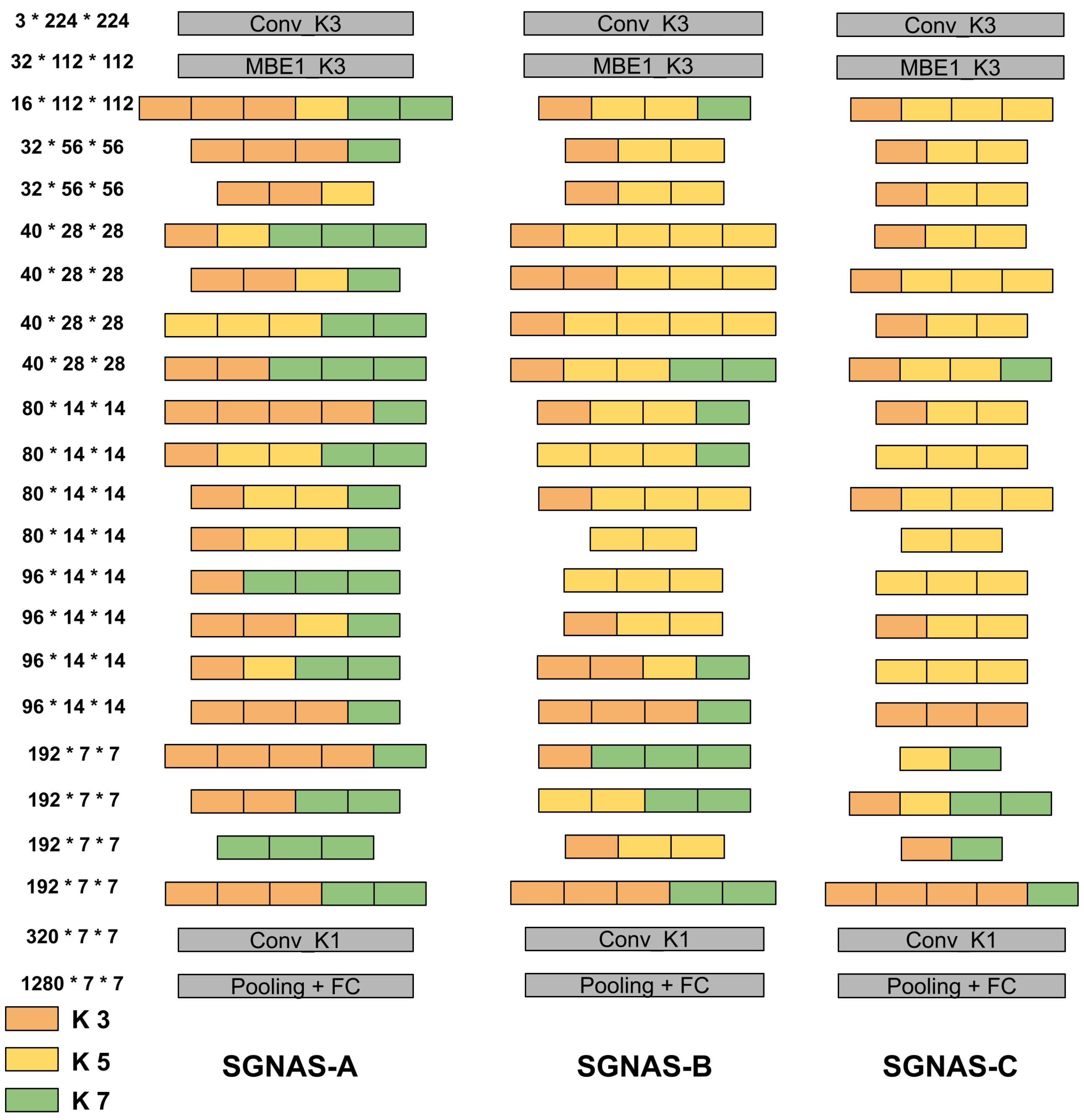


Figure 9. Visualization of the architectures searched by SGNAS (SGNAS-A, SGNAS-B, and SGNAS-C). "MBE1" denotes the mobile inverted bottleneck convolution layers with expansion rate 1. "KX" denotes depthwise convolution with the kernel size X. The gray blocks are predefined blocks before searching.

F. Limitation

Careful hyperparameter tuning: In SGNAS, the overall loss function of the architecture generator is defined in Eq. (10). However, in our experiment, carefully tuning the hyperparameter λ for different datasets is required to get trade off between hardware constraints and performance.

Architecture of the architecture generator: In SGNAS, we manually design architecture of the architecture generator. But we definitely believe that there is a better architecture for the generator. It is worth further study in the future.

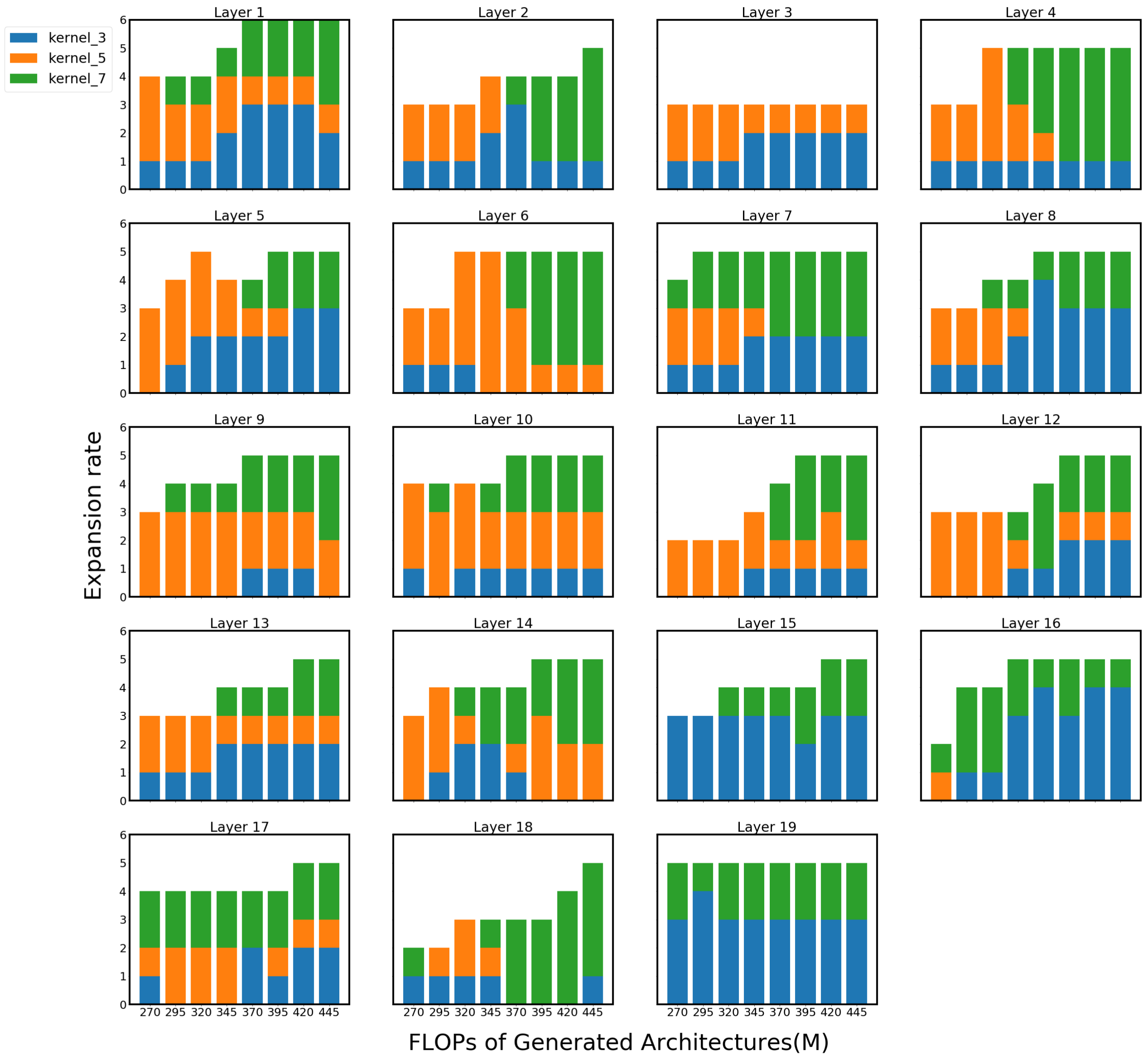


Figure 10. Visualization of the architectures search by SGNAS under different hardware constraints.