RENAS: Reinforced Evolutionary Neural Architecture Search

Yukang Chen[1,2] **Gaofeng Meng**[1,2] **Qian Zhang**[3] **Shiming Xiang**[1,2]
Chang Huang[3] **Lisen Mu**[3] **Xinggang Wang**[4]
[1]National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Sciences
[2]School of Artificial Intelligence, University of Chinese Academy of Sciences
[3]Horizon Robotics [4] Huazhong University of Science and Technology
{*yukang.chen,gfmeng,smxiang*}*@nlpr.ia.ac.cn*
{*qian01.zhang,chang.huang,lisen.mu*}*@horizon.ai,* {*xgwang*}*@hust.edu.cn*

Abstract

Neural Architecture Search (NAS) is an important yet challenging task in network design due to its high computational consumption. To address this issue, we propose the Reinforced Evolutionary Neural Architecture Search (RENAS), which is an evolutionary method with reinforced mutation for NAS. Our method integrates reinforced mutation into an evolution algorithm for neural architecture exploration, in which a mutation controller is introduced to learn the effects of slight modifications and make mutation actions. The reinforced mutation controller guides the model population to evolve efficiently. Furthermore, as child models can inherit parameters from their parents during evolution, our method requires very limited computational resources. In experiments, we conduct the proposed search method on CIFAR-10 and obtain a powerful network architecture, RENASNet. This architecture achieves a competitive result on CIFAR-10. The explored network architecture is transferable to ImageNet and achieves a new state-of-the-art accuracy, i.e., 75.7% top-1 accuracy with 5.36M parameters on mobile ImageNet. We further test its performance on semantic segmentation with DeepLabv3 on the PASCAL VOC. RENASNet outperforms MobileNet-v1, MobileNet-v2 and NASNet. It achieves 75.83% mIOU without being pretrained on COCO.

1. Introduction

Recent several years have witnessed the great success of neural networks [34, 14, 35, 33, 17] in tackling various challenging tasks, e.g., image classification, object detection and semantic segmentation. However, designing handcrafted neural networks is still a laborious task due to the heavy reliance on expert experience and large amount of trials. For example, hundreds of experts in academia and industry have made great efforts to optimize the architectures of neural networks that increase the top-5 accuracy to 96.43% on the ImageNet challenge from AlexNet [20], VGG [31], Inception [34] to ResNet [14].

Techniques in automated network architecture design have attracted increasing research interests. Many neural architecture search methods have been proposed and proven to be capable of yielding high-performance models. A large portion of these methods are based on Reinforcement Learning (RL) [41, 3, 40]. Typical RL-based NAS methods construct networks sequentially, e.g., by using a RNN controller [41, 27, 40] to determine a sequence of operator and connection tokens. In addition to RL, Evolution Algorithm (EA) is also employed in many works [28, 32, 29, 25, 36]. In EA-based NAS methods, a population of architectures are initialized first and then evolved with their validation accuracies as fitnesses.

EA and RL have achieved the state-of-the-art performance in the task of NAS. However, both of them have limitations respectively: 1) For EA-based NAS, it tends to evolve a population of architectures that guarantees the diversity of potential results. However, as the evolution progress relies heavily on random uncontrollable mutation, the efficiency of EA has no guarantee. For instance, AmoebaNet [28] is searched by an EA-based method and has better final results than its RL counterpart, NASNet [41]. But in the same search space, AmoebaNet [28] uses more computational resources than NASNet [41] (3150 GPU days vs 2000 GPU days). 2) For RL-based NAS, it relies on hyperparameters to guarantee stability. But when determining an architecture layer by layer, RL controller needs to try tens of actions to get a positive reward as a supervisory signal. This makes the training process inefficient.

In this paper, we propose the Reinforced Evolutionary

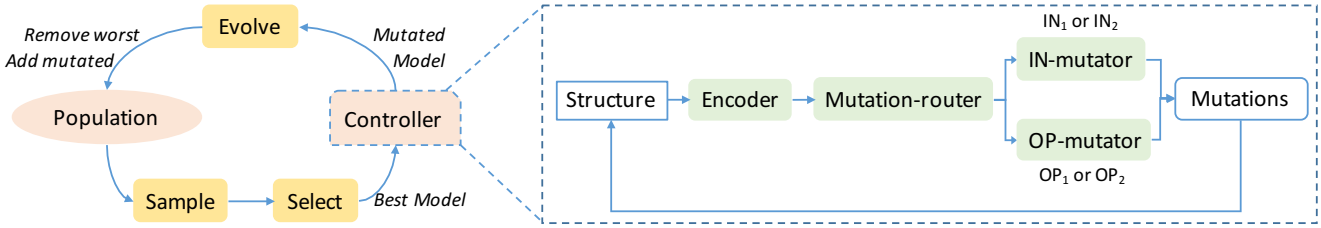

Figure 1. The evolutionary neural architecture search framework and the structure of the reinfored mutation controller.

Neural Architecture Search (RENAS), which integrates RL into the evolution framework to address the above issues. Our method introduces a reinforced mutation controller to help the efficient exploration of the search space. Thanks to the nature of EA, the child model could inherit most parameters from its parent, which in turn makes the search more efficient. Our main contributions are summarized as below:

- A novel neural architecture search framework is proposed with EA and RL integrated. This framework integrates the advantages of both of them and ensures the search efficiency.

- We design a reinforced mutation controller to learn the effects of slight modifications and make actions to guide the evolution. This technique helps the population evolve to a better status in fewer iterations.

- A powerful neural architecture, RENASNet, is discovered. It achieves a competitive accuracy on CIFAR-10, i.e., 2.88% $\pm$ 0.02 and a new state-of-the-art on mobile ImageNet with 75.7% top-1 accuracy and 5.36M parameters. We further test its performance on semantic segmentation with DeepLabv3 [5] on the PASCAL VOC 2012 [11]. RENASNet outperforms the state-of-the-art networks and achieves 75.83% mIOU without being pretrained on COCO [21].

2. Related Work

2.1. RL-based NAS

Reinforcement learning gains much research attention in recent works [41, 3, 40, 1]. In NAS [40], neural networks are specified by variable-length strings which are generated by a RNN controller. The network specified by a string is then trained to return a validation accuracy. In turn, the controller is updated with policy gradient using the accuracy as reward. In this framework, networks specified by strings are generated layer by layer. The success reported by NAS [40] inspires many other valuable works, but the expensive computational cost (28 days with 800 GPUs) limits its wide application, since training and evaluating a single model is time-consuming.

2.2. EA-based NAS

Evolution process in the nature is intuitively similar to NAS. Thus, many early automatic architecture search methods [26, 37, 32, 29, 25, 36] adopt EA to evolve a population of models. For instance, the large scale evolutionary method [29] explores a CNN search space with neuroevolution algorithm, which returns networks matching the human-designed models. The framework of our paper is based on AmoebaNets [28], in which a common evolutionary algorithm, tournament selection strategy, matches or even outperforms its RL baseline [41] in speed and accuracy. However, evolution process is slow due to the random mutation. To address this issue, we introduce a controller for mutation to guide the evolution process.

2.3. Efficient NAS

Difficulties of NAS mainly come from the extremely large search space and the time-consuming model evaluation. In NASNet [41], computational cost is saved with cell-wise search space, which is adopted by the following works [28, 27, 22]. Instead of exploring the whole network architecture, NASNet [41] centers on learning cell structures which are then stacked multiple times into a complete network, making the output networks scalable for various datasets and tasks. In addition, a variety of techniques on accelerating evaluation have proven effective: BlockQNN [39] improves the search speed with an early-stop strategy. ENAS [27] utilizes parameter sharing among child models instead of training from scratch. EAS [3] utilizes the Net2Net transformation [6] to reuse parameters. Accuracy prediction, used in this work [2], is also a novel technique to save computational resources, although the accuracy predictor might not always be accurate enough.

2.4. Integration of EA and RL

RL has shown its capacity of accelerating evolutionary progress via Baldwinian or Lamarckian mechanisms [9]. The idea of Integration RL and EA has been previously investigated, but our method is distinctly different from previous works. In [10], RL is used to enhance standard tree-based genetic programming in maze problems. In [18], integration of EA and RL is used to improve the adaption actions of a real robot. In [15], EA is integrated into a multi-

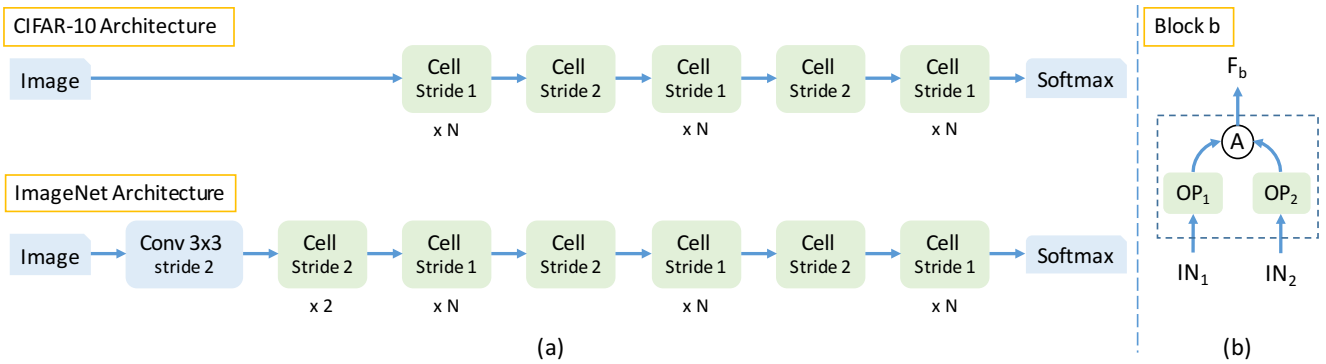

Figure 2. (a) Architectures employed for CIFAR-10 and ImageNet datasets respectively. During searching, the network structure is specified by the cell structure. The image size in ImageNet (224x224) is much larger than that in CIFAR-10 (32x32). So there are additional reduction cells and convolution 3x3 with stride 2 in ImageNet architectures to downsample feature maps. (b) Each cell consists of `#B` blocks. Each block takes in two inputs $\{i_1, i_2\}$, apply specific operations $\{o_1, o_2\}$ to them respectively and then combine them with element-wise addition to generate a feature map O_b. We search for $\{i_1, i_2, o_1, o_2\}$ for `#B` blocks to construct a reasonable cell structure, which in turn constitutes a network.

agent Q-learning to shrink the search space.

In our method, a mutation controller is integrated into the evolution framework to learn the effects of modifications and make reasonable mutation actions. Compared to only RL methods and only EA methods, this integration brings us the following benefits:

1) RL training becomes more efficient. Because making modifications to a network needs much fewer actions to make than constructing a model layer by layer. As the child model is modified from the parent model, the mutation controller is easy to learn the effects of slight differences.

2) The evolution process becomes more efficient and stable with the help of the reinforced mutation controller. Model architectures and their fitnesses (validation accuracies) are previously neglected but valuable hints generated during evolution. We reuse these useful supervisory signals to train the mutation controller. It in turn eliminates the accumulation of harmful mutation.

3. Search Space

Rather than designing the entire convolutional network, we adopt the idea that learning cell structures [41]. In this section, we introduce the search space by factorizing each network into cells and blocks. The architecture frames and the inner block are illustrated in Fig. 2.

3.1. Block

Each block maps two inputs into one output feature map as shown in Fig. 2 (b). It takes two input feature maps $\{i_1, i_2\}$, applies two operators $\{o_1, o_2\}$ to them respectively and then combines them into an output O via element-wise addition A. For this reason, each block could be specified by a string of length 4, $\{i_1, i_2, o_1, o_2\}$. $\{i_1, i_2\}$ are selected from $\{O_1^c, O_2^c, ..., O_{b-1}^c, O_B^{c-1}, O_B^{c-2}\}$, where $O_1^c, ..., O_{b-1}^c$ are outputs of previous blocks in the current cell. O_B^{c-1} and O_B^{c-2} are outputs of the first and second previous cells. Operation choices for $\{o_1, o_2\}$ are selected from a set of 6 functions: 3x3 depth-wise separable convolution, 5x5 depthwise-separable convolution, 7x7 depth-wise separable convolution, 3x3 avg pooling, 3x3 max pooling, identity.

3.2. Cell

Each cell can be represented as a directed acyclic graph which consists of `#B` blocks. Assume there is an input feature map with shape $h \times w \times f$, where h and w denote the height and width of the feature and f means the channel number. Cells with stride 2 output features with shape $\frac{h}{2} \times \frac{w}{2} \times 2f$ while cells with stride 1 keep the shape of feature maps. Each cell consist of `#B` blocks. Therefore, we search for the structure of each block and how they connect together to build a cell.

3.3. Network

Each network could be specified with three factors: the cell structure, `#N` the number of cells to be stacked and `#F` the number of filters in the first layer. As we fix `#N` and `#F` during search, our search space is constrained to all possible cell structures. Once search finished, models are constructed with different sizes to fit various tasks or datasets. We adjust the number of cells repeated `#N` and the number of filters in the first layer to control the depth and width of networks. As illustrated in Fig. 2 (a), the architecture for ImageNet has two more cells with stride 2 and a deeper steam. Because the image size in ImageNet (224x224) is much larger than that in CIFAR-10 (32x32), it needs more downsample operations.

Each network therefore is specified with `5#B` tokens, `4#B` of which are variable during search. Because each cell consists of `#B` blocks and each block is specified by

Algorithm 1: The framework of RENAS

input : num blocks each cell `#B`, max num epochs `#E`, num filters in first layer `#F`, num cells in model `#N`, population size `#P`, sample size `#S`, training set D_{train}, validation set D_{val}

output: a population of models P

$P^{(0)} \leftarrow$ `initialize(#F, #N, #P)`

for *i=1:#E* **do**

  $S^{(i)} \leftarrow$ `sample(`$P^{(i-1)}$`, #S)`

  $B, W \leftarrow$ `select(`$S^{(i)}$`)`

  $C, \omega^B \leftarrow$ `reinforced-mutate(`B`)`

  $\omega^C \leftarrow$ `finetune(`C`,` ω^B`,` D_{train}`)`

  $f_C \leftarrow$ `evaluate(`C`,` ω^C`,` D_{val}`)`

  $P^{(i)} \leftarrow$ `push-pop(`$P^{(i-1)}$`,` C`,` f_C`,` W`)`

end

5 tokens: two inputs $\{i_1, i_2\}$, two operations $\{o_1, o_2\}$ and a combination operation A that is fixed as addition. Therefore, searching network architecture is converted to search for `4#B` variables. This search space is smaller than NASNet search space [41]. We use only one cell type and reduce the feature map size using cells with stride 2. Besides we use 6 candidate functions and fix combiners as element-wise addition. The complexity of the search space can be estimated with ease. Each block consists of 2 nodes. For each node we need to select its input from $b+1$ possible indexes and its operator from these 6 functions. As we set `#B=5`, there are $(6^5 \times (5+1)!)^2 = 3.1 \times 10^{13}$ possible networks, which is still an extremely large space.

4. Search Strategy

4.1. Evolution Framework

To search for architectures with high performance automatically, a population of models $\mathbf{P}$ is initialized randomly. Each *individual* of $\mathbf{P}$ is trained on the training set D_{train} and evaluated on the validation set D_{val}. Its fitness f is defined as the validation accuracy. At each evolutionary step, a subset $\mathbf{S}$ is randomly sampled from $\mathbf{P}$. According to their fitnesses, the best individual *B* and the worst individual *W* are selected among $\mathbf{S}$. *W* is excluded from $\mathbf{P}$ and *B* becomes a *parent* to produce a child *C* with *mutation*. *C* is then trained and evaluated to measure its fitness f_C. Afterwards *C* is put into $\mathbf{P}$. This scheme actually belongs to *tournament selection* [12], repeating competitions in random samples. The procedure is formulated in Algorithm 1.

4.2. Reinforced Mutation

The reinforced mutation is implemented with a mutation controller to learn the effects of slight modifications and make mutation actions. Fig. 1 shows the framework of our controller, which implements a mechanism of attention. The controller takes a string of `5#B` length which represents the given cell architecture. Specifically, our controller consists of 4 parts: (1) an Encoder (`Enc`) following an embedding layer to learn the effect of each part of the cell, (2) a Mutation-router (`Mut-rt`) to choose one from i_1, i_2, o_1, o_2 of the block, (3) an Input-mutator (`IN-mut`) to change node's input with a new input i_{new} (4) an OP-mutator (`OP-mut`) to change node's operator with a new operator o_{new} .

Encoder `Enc` is a bidirectional recurrent network with an input embedding layer. Hidden states learned by `Enc` indicate the effect of a local part on the whole network. For block b in `Enc`, its hidden states are $\{H^b_{i_1}, H^b_{i_2}, H^b_{o_1}, H^b_{o_2}\}$ where $H^b_{o_1}$ represents the effect of block b's o_1 on the whole network. As each model is specified by `5#B` numbers, `Enc` generates `5#B` hidden states each step. Besides, we initialize two begin states, H^{c-1}, H^{c-2}, which represent the information of the first and second previous cells.

For block b, the controller makes two decisions sequentially. At first, depending on $H^b_{i_1}, H^b_{i_2}, H^b_{o_1}, H^b_{o_2}$, `Mut-rt` decides which one of i_1, i_2, o_1, o_2 in block b needs to be modified. It is sampled with a mechanism of attention via softmax classifiers. If one of input indexes, i_1 or i_2, is chosen, the `IN-mut` would be activated to pick one from $\{O^c_1, ..., O^c_{b-1}, O^{c-1}_B, O^{c-2}_B\}$. Otherwise `OP-mut` would choose a new operator from that 6 operation choices. As there are B blocks in each cell, this process would be repeated for B times to modify a given architecture. Thus it makes `2#B` modification actions for each model. We describe the implementation details in the following.

Mutation-router `Mut-rt` is designed to find which ingredient of each block needs modification. For each block, `Mut-rt`'s inputs are a subset of `Enc`'s outputs $H^b_{i_1}, H^b_{i_2}, H^b_{o_1}, H^b_{o_2}$ and its output is one of i_1, i_2, o_1, o_2, an ID to mutate. We apply a fully connected layer to each hidden state use $softmax$ to compute the modification probability of each ingredient $P^b_{i_1}, P^b_{i_2}, P^b_{o_1}, P^b_{o_2}$ and sample one from i_1, i_2, o_1, o_2 with these probabilities.

IN-mutator `IN-mut` chooses a new input for the node, if $ID \in (i_1, i_2)$. Its inputs include the chosen ID's hidden state H^b_{ID}, the hidden states of all previous block's outputs $[H^1_A, ..., H^{b-1}_A]$, and the hidden states of previous and previous-previous cells H^{c-1}, H^{c-2}. We concat $[H^1_A, ..., H^{b-1}_A, H^{c-1}, H^{c-2}]$ with H^b_{ID} and apply a fully connected layer to them. Similar to `Mut-rt`, we use $softmax$ to compute the probability of replacing the original input with each substitute and then we determine i_{new} by choosing from $1, ..., b$-1, c-1, c-2 with these probabilities.

OP-mutator `OP-mut` outputs an new operator o_{new} depending on the input H^b_{ID}. This process is simple and similar to `Mut-rt`.

Algorithm 2: Mutation generated by Controller

input : num blocks each cell `#B`, a sequence a of `4#B` number specifying a cell

output: a sequence of mutation actions m

$H^{c-1}, H^{c-2} \leftarrow$ `Enc.begin()`

$H^1, ..., H^B \leftarrow \texttt{Enc}(H^{c-1}, a)$

for *b=1:#B* **do**

  $H^b_{i_1}, H^b_{i_2}, H^b_{o_1}, H^b_{o_2}, H^b_A \leftarrow H^b$

  $ID \leftarrow \texttt{Mut-rt}([H^b_{i_1}, H^b_{i_2}, H^b_{o_1}, H^b_{o_2}])$

  if $ID \in (i_1, i_2)$ **then**

    $i_{new} \leftarrow \texttt{IN-mut}\ (H^b_{ID}, [H^1_A, ..., H^{b-1}_A, H^{c-1}, H^{c-2}])$

    $m^{(b)} \leftarrow (ID, i_{new})$

  else

    $o_{new} \leftarrow \texttt{OP-mut}\ (H^b_{ID})$

    $m^{(b)} \leftarrow (ID, o_{new})$

  end

end

4.3. Search Details

Controller At each evolution step, the controller makes a sequence of mutation actions. Then a child model C is produced with the parent model modified. Then the validation accuracy f_C is computed with parameter inheriting which is introduced in the following paragraph. The reward γ is a nonlinear function [3] of f_C, i.e., $\gamma = tan(f_C \cdot \frac{\pi}{2})$, since the gain of improving accuracy should be larger while the validation accuracy of its parent is higher. The controller parameters θ is updated via policy gradient.

Child Models Child models are trained and evaluated with its parameters inherited from their parents. For each alive model B in the population, we store its architecture string, its fitness f_B and its learnable parameters ω^B. As each child model C is generated from its parent model with slight modifications, differences between them only exist in the mutated layers. Therefore the child could inherit most parameters from the parent B. ω^C are classified into inheritable parameters ω^C_{inh} and new initialized parameters ω^C_{new}. And its fitness (validation accuracy) f_C could be evaluated with fine-tuning instead of training from scratch. During fine-tuning, we train ω^C on a whole pass through D_{train} with the learning rate of ω^C_{new} 10 times large as that of ω^C_{inh}. In the experiments, the learning rate of ω^C_{new} equals to 0.01.

Deriving Architectures During search, we set each cell contains `#B=5` blocks, and `#F=24` filters in the first convolution cell and we unroll the cells for `#N=2`. After the maximum number of epochs `#E` is reached, we only retrain the models in the population from scratch and then take the model with highest accuracy. It is possible to improve our results by retraining more sampled models from scratch as done by other works [41, 40], but it is unfair to prove the performance of our controller. In the experiments, the population size `#P` is set as 20. For better comparison, we set `#F` and `#N` same to NASNets [41].

5. Experimental Results

In this section, we first show our implementation details. Then, we compare our searched architecture RENASNet (as illustrated in Fig. 3) with both state-of-the-art hand-design networks and other searched models on CIFAR-10 and ImageNet datasets. Ablation studies are made to show the search efficiency of RENAS. Further experiments show that RENASNet can be successfully transferred to achieve the semantic segmentation task.

5.1. Implementation Details

5.1.1 Datasets Details

CIFAR-10 CIFAR-10 [19] consists of 50,000 training images and 10,000 test images. 5,000 images are partitioned from the training set as a validation set. All images are whitened with the channel mean subtracted and the channel standard deviation divided. Then, we crop 32 x 32 patches from images and pad them to 40 x 40. These patches are also randomly fliped horizontally for data augmentation. When retraining the result architecture, we also use the cutout augmentation [8].

ImageNet For data augmentation on ImageNet [7], we resize the original input images with its shorter side randomly sampled in [256, 480] for scale augmentation [31]. 224×224 patches are randomly cropped from images. Other standard operations, i.e., horizontal flip, mean pixel subtraction and the standard color augmentation in Alexnet [20], are also conducted [20]. For the last 20 epochs, we withdraw most augmentations and only keep crop and flip augumentations for fine-tuning.

5.1.2 Training details

CIFAR-10 When training models on CIFAR-10, we use standard SGD optimizer with momentum rate set to 0.9, auxiliary classifier located at $\frac{2}{3}$ of the maximum depth weighted by 0.4, weight decay 3×10^{-4}, and dropout of 0.5 in the final softmax layer. In addition, we drop each path with probability 0.5 for regularization. Our batch size is 64 on each GPU and 2 GPUs are used. The learning rate initially is set to 0.05 and later decays with a cosine restart schedule for 630 epochs.

ImageNet When training models on ImageNet, we train each model for 200 epochs, using standard SGD optimizer with momentum rate set to 0.9, auxiliary classifier located at $\frac{2}{3}$ of the maximum depth weighted by 0.4, weight decay 4×10^{-5}. Our batch size is 64 on each GPU and 4 GPUs

Table 1. CIFAR-10 results. The top section presents the top hand-design networks, the middle section presents other architecture search results and the bottom section shows our results. **#Params** means the number of free parameters.

Model	Cutout	GPUs	Days	#Params	Error(%)	Method
DenseNet-BC [17]	-	-	-	25.6M	3.46	-
PNASNet-5 [22]	-	100	1.5	3.2M	3.41 ± 0.09	SMBO
NASNet-A + cutout [41]	✓	500	4	3.3M	2.65	RL
AmoebaNet-B + cutout [28]	✓	450	7	2.8M	2.55 ± 0.05	EA
ENAS + cutout [27]	✓	1	0.5	4.6M	2.89	RL
DARTS (first order) + cutout [23]	✓	1	1.5	2.9M	2.94	Gradient
DARTS (second order) + cutout [23]	✓	1	4	3.4M	2.83 ± 0.06	Gradient
RENASNet (6, 32) + cutout	✓	4	1.5	3.5M	2.88 ± 0.02	EA&RL

Table 2. ImageNet classification results in the mobile setting. The results of hand-design models are in the top section, other NAS results are presented in the middle section and the result of our model is in the bottom section.

Model	#Params	#Mult-Adds	Top-1/Top-5 Acc(%)	Method
MobileNet-v1 [16]	4.2M	569M	70.6 / 89.5	-
MobileNet-v2 (1.4)[30]	6.9M	585M	74.7 / -	-
ShuffleNet-v1 2x [38]	$\approx$ 5M	524M	73.7 / -	-
ShuffleNet-v2 2x (with SE) [24]	$\approx$ 5M	597M	75.4 / -	-
NASNet-A [41]	5.3M	564M	74.0 / 91.6	RL
NASNet-B [41]	5.3M	488M	72.8 / 91.3	RL
NASNet-C [41]	4.9M	558M	72.5 / 91.0	RL
AmoebaNet-A [28]	5.1M	555M	74.5 / 92.0	EA
AmoebaNet-B [28]	5.3M	555M	74.0 / 91.5	EA
AmoebaNet-C [28]	5.1M	535M	75.1 / 92.1	EA
AmoebaNet-C (more filters) [28]	6.35M	570M	75.7 / 92.4	EA
PNASNet-5 [22]	5.1M	588M	74.2 / 91.9	SMBO
ENAS [27]*	5.1M	523M	74.3 / 91.9	RL
DARTS [23]	4.9M	595M	73.1 / 91.0	Gradient
RENASNet (4, 44)	5.36M	580M	**75.7 / 92.6**	EA&RL

* The result of ENAS was obtained by training with our setup, as it is not reported [27].

are used. The learning rate is initially set to 0.1 and later decays in a polynomial schedule.

5.1.3 Details of the Controller

For our controller, we use an LSTM with an embedding layer. The embedding size and the hidden state size of LSTM are both 100. The parameters of our controller are initialized with random values sampled from a normal distribution with a mean of zero and standard deviation of 0.01 and trained with Adam at a learning rate of 0.001. We apply a tanh constant of 2.5 and a temperature of 5.0 to the logits of the controller, and add the entropy of the controller to the reward with 0.1 weighted.

5.1.4 Details of architecture search space

For fair comparison, some details of our search space follow the NASNet search space:

(1) All convolutions follow an ordering of ReLU, convolution and batch normalization.

(2) Each separable convolution is applied twice sequentially to the input feature map.

(3) To match shapes in convolutional cells, 1x1 convolutions are applied as necessary.

(4) Separable convolutions do not employ batch normalization or ReLU between depthwise and pointwise convolutions.

5.2. Image Classification

5.2.1 Results on CIFAR-10

Here we report the performance of our searched model, RENASNet, and make comparisons to other state-of-the-art models in Table 1 on CIFAR-10. After the cell structures are fixed, we construct the entire networks same to the structure setting of NASNet[41] and train them with the details mentioned before. The simple notation (6, 32) denotes cell unroll for $N = 6$ times and $F = 32$ filters in the first cell. The CIFAR-10 results are presented in Table 1. RENASNet achieves a competitive result to other state-of-the-art mod-

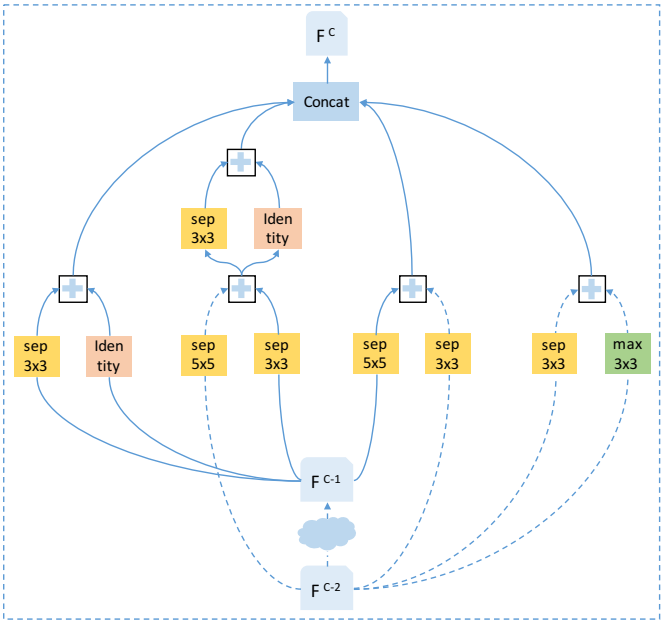

Figure 3. The searched cell structure by RENAS. The full outer architectures for CIFAR and ImageNet are shown in Fig. 2 (a).

els. Only NASNet-A and AmoebaNet have stable better performances than RENASNet while they use much more computational resources (2000 GPU days and 3150 GPU days) than ours. ENAS is more efficient than our method, but our model has less parameters with better accuracy.

5.2.2 Results on ImageNet

State-of-the-art image classifiers on ImageNet is shown in Table 2. We conduct the comparison in the mobile setting where the image size is 224x224 and the multi-add operation numbers of models are under 600M. Note that as the accuracy of ENAS[27] on ImageNet is not reported in the original paper, we trained it with all hyper-parameters and settings exactly same to RENASNet.

The results on ImageNet are more convincing because CIFAR-10 is small and easy to be over-fitting. The results on ImageNet are shown in Table 2. RENASNet surpasses both the manually designed models, including MobileNets [16, 30] and ShuffleNets [38, 24], and the other state-of-the-art NAS models. Especially for NASNet [41] and AmoebaNet [28], they are representative RL-based and EA-based methods respectively and spend much more GPUs and days than ours. In Table 1 and Table 2, we also compare with DARTS [23], which is a novel and gradient-based method. RENASNet is similar to DARTS [23] on CIFAR-10, but outperforms it on ImageNet.

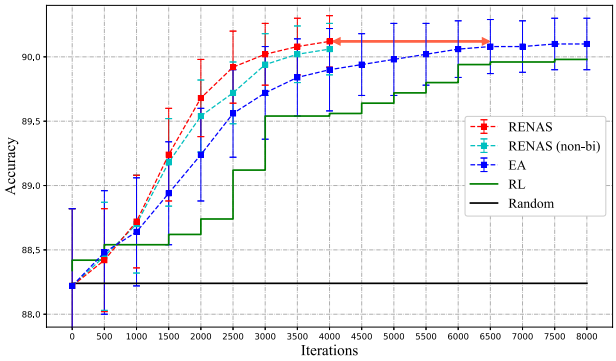

Figure 4. Efficiency Comparison under the same search space (unified Cell and 6 operation choices).

5.3. Search Efficiency

Honestly speaking, although the computation cost of RENAS is much less than NASNet and AmoebaNet, it can not reflect a fair comparison of search efficiency to RL-based and EA-based NAS methods. As stated in Section 3, the search space used in our experiments is smaller than the original NASNet search space. We have not distinguished Normal Cell and Reduction Cell and there are 6 operation choices. Thus, the efficiency of RENAS also comes from the search space.

In this section, we make a fair ablation study to compare the efficiency of RENAS to EA and RL under the same search space (unified Cell and 6 operation choices) as in Fig. 4. For the compared methods, we keep track of the searched models for every 500 iterations. All searched models are evaluated with 20 epochs training from scratch. EA and RENAS are evaluated by the accuracy mean and variance of models in the population. EA is conduct with the same settings to RENAS, except that mutation actions are made randomly. RL is evaluated on the best model over time. Random is a model randomly picked from the search space. As shown in the Fig. 4, RENAS achieves better efficiency than EA and RL. The speedup of RENAS over RL/EA is around 1.5 - 2.0 times.

We also make an additional comparison. As mentioned in Section 4.2, the controller is equipped with a bidirectional recurrent network to have a better capacity in architecture encoding. We compare RENAS to a counterpart with common recurrent network, RENAS (non-bi). It has exact same settings to RENAS, except the recurrent network. Fig. 4 shows the inferiority of RENAS (non-bi).

5.4. Semantic Segmentation

In this section, we make further experiments on semantic segmentation with DeepLabv3 [5]. All our experiments and comparison methods use Atrous Spatial Pyramid Pooling module (ASPP) [4] that contains three 3x3 convolutions

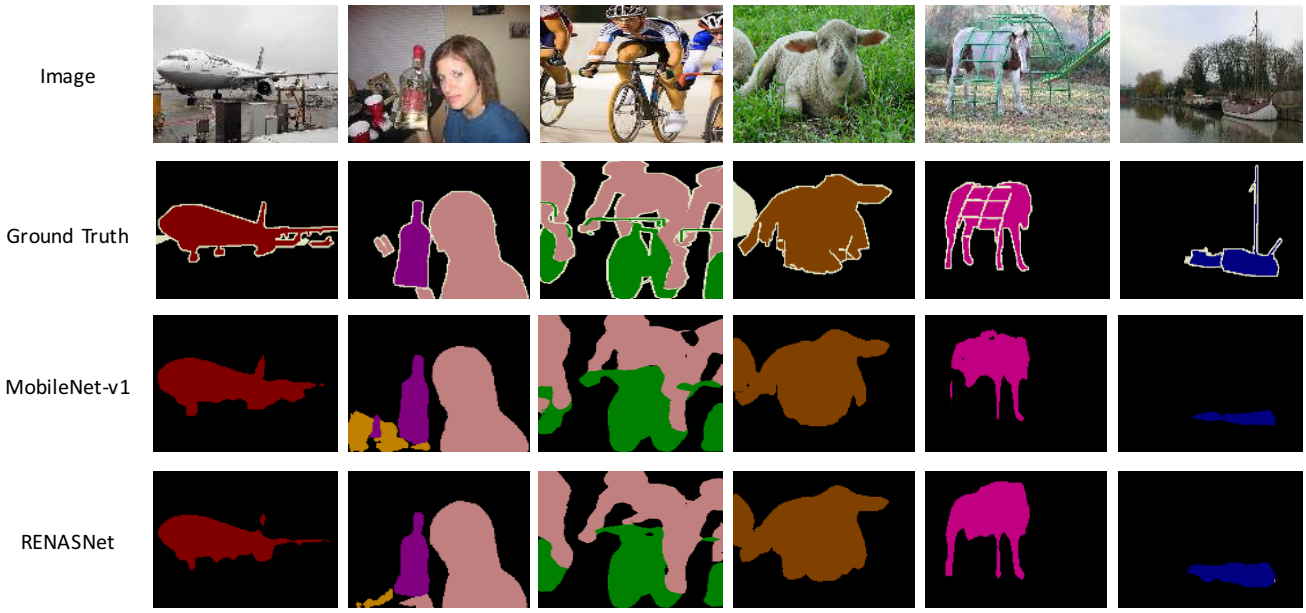

Figure 5. Segmentation Results Visualization

Table 3. Semantic Segmentation: DeepLabv3 on the PASCAL VOC 2012 validation set.

Model	Pretrain	#Params	mIOU(%)
MobileNet-v1 [16]	COCO	11.15M	75.29
MobileNet-v2 [30]	COCO	4.52M	75.70
MobileNet-v1 [16]	ImageNet	11.14M	68.79
MobileNet-v2 [30]	ImageNet	4.51M	70.02
NASNet-A [41]	ImageNet	12.39M	73.68
RENASNet	ImageNet	11.63M	**75.83**

with different atrous rates. The output stride is 16 that is the ratio of input image spatial resolution to final output resolution. We do not use Multi-scale and left-right Flipped inputs (MF), which is employed by some other works [30] for boosting the performance. Following our comparison methods, we conduct the experiments on the PASCAL VOC 2012 dataset [11] and standard extra annotated images from [13] with evaluation metric as mIOU.

We compare RENASNet with three other mobile networks, MobileNet-v1 [16], MobileNet-v2 [30] and NASNet-A [41] and summarize the results in Table 3. The results of models pretrained on COCO [21] are reported in [30]. Models pretrained on ImageNet are implemented by ourselves using exactly same hyper-parameters and settings. From the results, we have observed that: 1) The performance of this task relies heavily on pretrained models. Without being pretrained on COCO, MobileNet-v1 and MobileNet-v2 suffer from severe performance decay. 2) In terms of mIOU, RENASNet outperforms MobileNet-v1, MobileNet-v2 and NASNet-A [16] at the same settings. Moreover, RENASNet (75.83%) pretrained on ImageNet even performs better than MobileNet-v1 (75.29%) and MobileNet-v2 (75.70%) that pretrained on COCO. The segmentation results are visualized in Fig. 5.

6. Conclusion

In this paper, we have proposed a method for neural architecture search by integrating evolution algorithm and reinforcement learning into a unified framework. Inspired by the procedure of designing networks manually, we use a controller to learn the effects of modifications and make better mutation actions. The searched architecture, i.e., RENASNet, achieves competitive performance on CIFAR-10 and outperforms other state-of-the-art models on ImageNet (75.7% top-1 accuracy with 5.36M parameters). In addition, RENASNet also demonstrate its high performance on the semantic segmentation task. RENASNet outperforms other mobile size networks and achieves 75.83% mIOU without being pretrained on COCO. It shows that RENASNet can be transferred to other computer vision tasks in addition to image classification. In future, we will try to conduct NAS on other tasks, e.g., object detection.

Acknowledgement

This work was supported by the National Natural Science Foundation of China under Grants 91646207, 61773377, 61573352 and 61876212, and the Beijing Natural Science Foundation under Grant L172053.

References

[1] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *CoRR*, abs/1611.02167, 2016.

[2] B. Baker, O. Gupta, R. Raskar, and N. Naik. Accelerating neural architecture search using performance prediction. *CoRR*, abs/1705.10823, 2017.

[3] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang. Efficient architecture search by network transformation. In *AAAI*, pages 2787–2794, 2018.

[4] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 40(4):834–848, 2018.

[5] L. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking atrous convolution for semantic image segmentation. *CoRR*, abs/1706.05587, 2017.

[6] T. Chen, I. J. Goodfellow, and J. Shlens. Net2net: Accelerating learning via knowledge transfer. *CoRR*, abs/1511.05641, 2015.

[7] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and F. Li. Imagenet: A large-scale hierarchical image database. In *CVPR*, pages 248–255, 2009.

[8] T. Devries and G. W. Taylor. Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552, 2017.

[9] K. L. Downing. Adaptive genetic programs via reinforcement learning. In *GEC*, 2001.

[10] K. L. Downing. Reinforced genetic programming. *Genetic Programming and Evolvable Machines*, 2(3):259–288, 2001.

[11] M. Everingham, S. M. A. Eslami, L. J. V. Gool, C. K. I. Williams, J. M. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, 2015.

[12] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *FGA*, pages 69–93. 1990.

[13] B. Hariharan, P. Arbelaez, L. D. Bourdev, S. Maji, and J. Malik. Semantic contours from inverse detectors. pages 991–998, 2011.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[15] I. Hitoshi. Multi-agent reinforcement learning with genetic programming. In *GP*, 1998.

[16] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[17] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, pages 2261–2269, 2017.

[18] S. Kamio and H. Iba. Adaptation technique for integrating genetic programming and reinforcement learning for real robots. *IEEE Trans. Evolutionary Computation*, 9(3):318–333, 2005.

[19] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Technical report.*, 1(4):1–7, 2009.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.

[21] T. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. In *ECCV*, pages 740–755, 2014.

[22] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L. Li, L. Fei-Fei, A. L. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *ECCV*, pages 19–35, 2018.

[23] H. Liu, K. Simonyan, and Y. Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[24] N. Ma, X. Zhang, H. Zheng, and J. Sun. Shufflenet V2: practical guidelines for efficient CNN architecture design. In *ECCV*, pages 122–138, 2018.

[25] R. Miikkulainen, J. Z. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017.

[26] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms. In *ICGA*, pages 379–384, 1989.

[27] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. In *ICML*, pages 4092–4101, 2018.

[28] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018.

[29] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *ICML*, pages 2902–2911, 2017.

[30] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[31] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[32] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[33] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, pages 4278–4284, 2017.

[34] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, pages 1–9, 2015.

[35] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, pages 2818–2826, 2016.

[36] L. Xie and A. L. Yuille. Genetic CNN. In *ICCV*, pages 1388–1397, 2017.

[37] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Trans. Neural Networks*, 8(3):694–713, 1997.

[38] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CoRR*, abs/1707.01083, 2017.

[39] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu. Practical block-wise neural network architecture generation. In *CVPR*, 2018.

[40] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[41] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.