The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Efficient Architecture Search by Network Transformation

Han Cai,[1] **Tianyao Chen,**[1] **Weinan Zhang,**[1*] **Yong Yu,**[1] **Jun Wang**[2]
[1]Shanghai Jiao Tong University, [2]University College London
{hcai,tychen,wnzhang,yyu}@apex.sjtu.edu.cn, j.wang@cs.ucl.ac.uk

Abstract

Techniques for automatically designing deep neural network architectures such as reinforcement learning based approaches have recently shown promising results. However, their success is based on vast computational resources (e.g. hundreds of GPUs), making them difficult to be widely used. A noticeable limitation is that they still design and train each network from scratch during the exploration of the architecture space, which is highly inefficient. In this paper, we propose a new framework toward efficient architecture search by exploring the architecture space based on the current network and reusing its weights. We employ a reinforcement learning agent as the meta-controller, whose action is to grow the network depth or layer width with function-preserving transformations. As such, the previously validated networks can be reused for further exploration, thus saves a large amount of computational cost. We apply our method to explore the architecture space of the plain convolutional neural networks (no skip-connections, branching etc.) on image benchmark datasets (CIFAR-10, SVHN) with restricted computational resources (5 GPUs). Our method can design highly competitive networks that outperform existing networks using the same design scheme. On CIFAR-10, our model without skip-connections achieves 4.23% test error rate, exceeding a vast majority of modern architectures and approaching DenseNet. Furthermore, by applying our method to explore the DenseNet architecture space, we are able to achieve more accurate networks with fewer parameters.

对于每一个网络的设计与训练都是从头开始的

加速：基于当前网络进行设计并重用权重

Introduction

The great success of deep neural networks in various challenging applications (Krizhevsky, Sutskever, and Hinton 2012; Bahdanau, Cho, and Bengio 2014; Silver et al. 2016) has led to a paradigm shift from feature designing to architecture designing, which still remains a laborious task and requires human expertise. In recent years, many techniques for automating the architecture design process have been proposed (Snoek, Larochelle, and Adams 2012; Bergstra and Bengio 2012; Baker et al. 2017; Zoph and Le 2017; Real et al. 2017; Negrinho and Gordon 2017), and promising results of designing competitive models against human-designed models are reported on some benchmark datasets (Zoph and Le 2017; Real et al. 2017). Despite the promising results as reported, their success is based on vast computational resources (e.g. hundreds of GPUs), making them difficult to be used in practice for individual researchers, small sized companies, or university research teams. Another key drawback is that they still design and train each network from scratch during exploring the architecture space without any leverage of previously explored networks, which results in high computational resources waste.

由特征设计转为框架设计

In fact, during the architecture design process, many slightly different networks are trained for the same task. Apart from their final validation performances that are used to guide exploration, we should also have access to their architectures, weights, training curves etc., which contain abundant knowledge and can be leveraged to accelerate the architecture design process just like human experts (Chen, Goodfellow, and Shlens 2015; Klein et al. 2017). Furthermore, there are typically many well-designed architectures, by human or automatic architecture designing methods, that have achieved good performances at the target task. Under restricted computational resources limits, instead of totally neglecting these existing networks and exploring the architecture space from scratch (which does not guarantee to result in better performance architectures), a more economical and efficient alternative could be exploring the architecture space based on these successful networks and reusing their weights.

idea，重用模型，重用权重

In this paper, we propose a new framework, called EAS, Efficient Architecture Search, where the meta-controller explores the architecture space by *network transformation* operations such as widening a certain layer (more units or filters), inserting a layer, adding skip-connections etc., given an existing network trained on the same task. To reuse weights, we consider the class of function-preserving transformations (Chen, Goodfellow, and Shlens 2015) that allow to initialize the new network to represent the same function as the given network but use different parameterization to be further trained to improve the performance, which can significantly accelerate the training of the new network especially for large networks. Furthermore, we combine our framework with recent advances of reinforcement learning (RL) based automatic architecture designing methods (Baker et al. 2017; Zoph and Le 2017), and employ a RL based agent as the meta-controller.

[*] Correspondence to Weinan Zhang.

Our experiments of exploring the architecture space of the plain convolutional neural networks (CNNs), which purely consists of convolutional, fully-connected and pooling layers without skip-connections, branching etc., on image benchmark datasets (CIFAR-10, SVHN), show that EAS with limited computational resources (5 GPUs) can design competitive architectures. The best plain model designed by EAS on CIFAR-10 with standard data augmentation achieves 4.23% test error rate, even better than many modern architectures that use skip-connections. We further apply our method to explore the DenseNet (Huang et al. 2017) architecture space, and achieve 4.66% test error rate on CIFAR-10 without data augmentation and 3.44% on CIFAR-10 with standard data augmentation, surpassing the best results given by the original DenseNet while still maintaining fewer parameters.

Related Work and Background

Automatic Architecture Designing There is a long standing study on automatic architecture designing. Neuro-evolution algorithms which mimic the evolution processes in the nature, are one of the earliest automatic architecture designing methods (Miller, Todd, and Hegde 1989; Stanley and Miikkulainen 2002). Authors in (Real et al. 2017) used neuro-evolution algorithms to explore a large CNN architecture space and achieved networks which can match performances of human-designed models. In parallel, automatic architecture designing has also been studied in the context of Bayesian optimization (Bergstra and Bengio 2012; Domhan, Springenberg, and Hutter 2015; Mendoza et al. 2016). Recently, reinforcement learning is introduced in automatic architecture designing and has shown strong empirical results. Authors in (Baker et al. 2017) presented a Q-learning agent to sequentially pick CNN layers; authors in (Zoph and Le 2017) used an auto-regressive recurrent network to generate a variable-length string that specifies the architecture of a neural network and trained the recurrent network with policy gradient.

As the above solutions rely on designing or training networks from scratch, significant computational resources have been wasted during the construction. In this paper, we aim to address the efficiency problem. Technically, we allow to reuse the existing networks trained on the same task and take network transformation actions. Both function-preserving transformations and an alternative RL based meta-controller are used to explore the architecture space. Moreover, we notice that there are some complementary techniques, such as learning curve prediction (Klein et al. 2017), for improving the efficiency, which can be combined with our method.

网络转换技术

Network Transformation and Knowledge Transfer Generally, any modification to a given network can be viewed as a network transformation operation. In this paper, since our aim is to utilize knowledge stored in previously trained networks, we focus on identifying the kind of network transformation operations that would be able to reuse pre-existing models. The idea of reusing pre-existing models or knowledge transfer between neural networks has been studied before. Net2Net technique introduced in (Chen, Goodfellow, and Shlens 2015) describes two specific function-preserving transformations, namely Net2WiderNet and Net2DeeperNet, which respectively initialize a wider or deeper student network to represent the same functionality of the given teacher network and have proved to significantly accelerate the training of the student network especially for large networks. Similar function-preserving schemes have also been proposed in ResNet particularly for training very deep architectures (He et al. 2016a). Additionally, the network compression technique presented in (Han et al. 2015) prunes less important connections (low-weight connections) in order to shrink the size of neural networks without reducing their accuracy.

In this paper, instead, we focus on utilizing such network transformations to reuse pre-existing models to efficiently and economically explore the architecture space for automatic architecture designing.

Reinforcement Learning Background Our meta-controller in this work is based on RL (Sutton and Barto 1998), techniques for training the agent to maximize the cumulative reward when interacting with an environment (Cai et al. 2017). We use the REINFORCE algorithm (Williams 1992) similar to (Zoph and Le 2017) for updating the meta-controller, while other advanced policy gradient methods (Kakade 2002; Schulman et al. 2015) can be applied analogously. Our action space is, however, different with that of (Zoph and Le 2017) or any other RL based approach (Baker et al. 2017), as our actions are the network transformation operations like adding, deleting, widening, etc., while others are specific configurations of a newly created network layer on the top of preceding layers. Specifically, we model the automatic architecture design procedure as a sequential decision making process, where the state is the current network architecture and the action is the corresponding network transformation operation. After T steps of network transformations, the final network architecture, along with its weights transferred from the initial input network, is then trained in the real data to get the validation performance to calculate the reward signal, which is further used to update the meta-controller via policy gradient algorithms to maximize the expected validation performances of the designed networks by the meta-controller.

Architecture Search by Net Transformation

In this section, we first introduce the overall framework of our meta-controller, and then show how each specific network transformation decision is made under it. We later extend the function-preserving transformations to the DenseNet (Huang et al. 2017) architecture space where directly applying the original Net2Net operations can be problematic since the output of a layer will be fed to all subsequent layers.

We consider learning a meta-controller to generate network transformation actions given the current network architecture, which is specified with a variable-length string (Zoph and Le 2017). To be able to generate various types

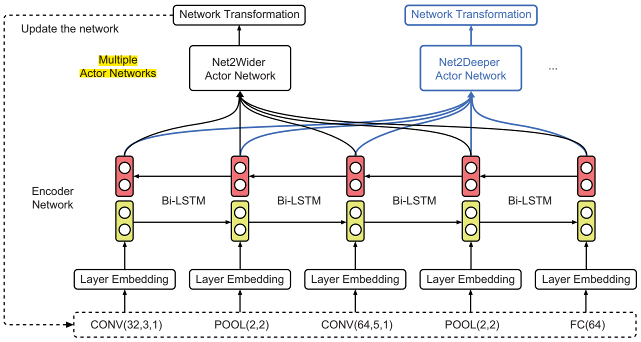

Figure 1: Overview of the RL based meta-controller in EAS, which consists of an encoder network for encoding the architecture and multiple separate actor networks for taking network transformation actions.

of network transformation actions while keeping the meta-controller simple, we use an encoder network to learn a low-dimensional representation of the given architecture, which is then fed into each separate actor network to generate a certain type of network transformation actions. Furthermore, to handle variable-length network architectures as input and take the whole input architecture into consideration when making decisions, the encoder network is implemented with a bidirectional recurrent network (Schuster and Paliwal 1997) with an input embedding layer. The overall framework is illustrated in Figure 1, which is an analogue of end-to-end sequence to sequence learning (Sutskever, Vinyals, and Le 2014; Bahdanau, Cho, and Bengio 2014).

使用编码器学习一个给定网络架构的低维表示

Actor Networks

Given the low dimensional representation of the input architecture, each actor network makes necessary decisions for taking a certain type of network transformation actions. In this work, we introduce two specific actor networks, namely Net2Wider actor and Net2Deeper actor which correspond to Net2WiderNet and Net2DeeperNet respectively.

Net2Wider Actor Net2WiderNet operation allows to replace a layer with a wider layer, meaning more units for fully-connected layers, or more filters for convolutional layers, while preserving the functionality. For example, consider a convolutional layer with kernel $\boldsymbol{K}_l$ whose shape is $(k_w^l, k_h^l, f_i^l, f_o^l)$ where k_w^l and k_h^l denote the filter width and height, while f_i^l and f_o^l denote the number of input and output channels. To replace this layer with a wider layer that has $\hat{f}_o^l$ $(> f_o^l)$ output channels, we should first introduce a random remapping function G_l, which is defined as

$$G_l(j) = \begin{cases} j & 1 \leq j \leq f_o^l \\ \text{random sample from } \{1, \cdots, f_o^l\} & f_o^l < j \leq \hat{f}_o^l \end{cases}. \tag{1}$$

With the remapping function G_l, we have the new kernel $\hat{\boldsymbol{K}}_l$ for the wider layer with shape $(k_w^l, k_h^l, f_i^l, \hat{f}_o^l)$

$$\hat{\boldsymbol{K}}_l[x, y, i, j] = \boldsymbol{K}_l[x, y, i, G_l(j)]. \tag{2}$$

As such, the first f_o^l entries in the output channel dimension of $\hat{\boldsymbol{K}}_l$ are directly copied from $\boldsymbol{K}_l$ while the remaining $\hat{f}_o^l - f_o^l$ entries are created by choosing randomly as defined in G_l. Accordingly, the new output of the wider layer is $\hat{\boldsymbol{O}}_l$ with $\hat{\boldsymbol{O}}_l(j) = \boldsymbol{O}_l(G_l(j))$, where $\boldsymbol{O}_l$ is the output of the original layer and we only show the channel dimension to make the notation simpler.

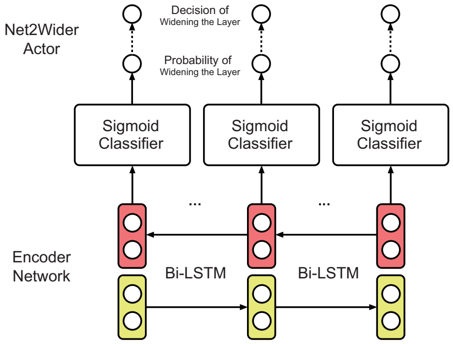

Figure 2: Net2Wider actor, which uses a shared sigmoid classifier to simultaneously determine whether to widen each layer based on its hidden state given by the encoder network.

To preserve the functionality, the kernel $\boldsymbol{K}_{l+1}$ of the next layer should also be modified due to the replication in its input. The new kernel $\hat{\boldsymbol{K}}_{l+1}$ with shape $(k_w^{l+1}, k_h^{l+1}, \hat{f}_i^{l+1} = \hat{f}_o^l, f_o^{l+1})$ is given as

$$\hat{\boldsymbol{K}}_{l+1}[x, y, j, k] = \frac{\boldsymbol{K}_{l+1}[x, y, G_l(j), k]}{\left|\{z | G_l(z) = G_l(j)\}\right|}. \tag{3}$$

For further details, we refer to the original Net2Net work (Chen, Goodfellow, and Shlens 2015).

In our work, to be flexible and efficient, the Net2Wider actor simultaneously determines whether each layer should be extended. Specifically, for each layer, this decision is carried out by a shared sigmoid classifier given the hidden state of the layer learned by the bidirectional encoder network. Moreover, we follow previous work and search the number of filters for convolutional layers and units for fully-connected layers in a discrete space. Therefore, if the Net2Wider actor decides to widen a layer, the number of filters or units of the layer increases to the next discrete level, e.g. from 32 to 64. The structure of Net2Wider actor is shown in Figure 2.

共享sigmoid分类器来同时决定是否对每一层进行拓展

Net2Deeper Actor Net2DeeperNet operation allows to insert a new layer that is initialized as adding an identity mapping between two layers so as to preserve the functionality. For a new convolutional layer, the kernel is set to be identity filters while for a new fully-connected layer, the weight matrix is set to be identity matrix. Thus the new layer is set with the same number of filters or units as the layer below at first, and could further get wider when Net2WiderNet operation is performed on it. To fully preserve the functionality, Net2DeeperNet operation has a constraint on the activation function ϕ, i.e. ϕ must satisfy $\phi(\boldsymbol{I}\phi(\boldsymbol{v})) = \phi(\boldsymbol{v})$ for all vectors $\boldsymbol{v}$. This property holds for rectified linear activation (ReLU) but fails for sigmoid and tanh activation. However, we can still reuse weights of existing networks with sigmoid

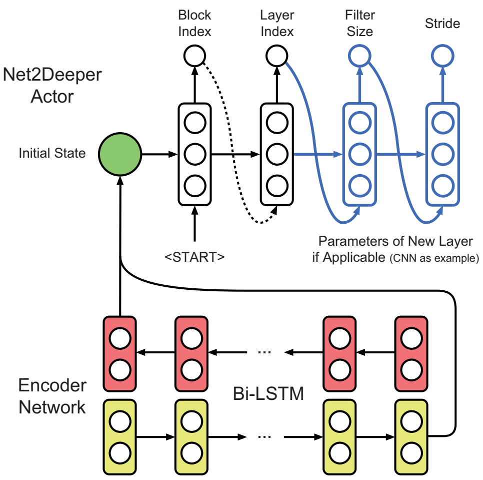

Figure 3: Net2Deeper actor, which uses a recurrent network to sequentially determine where to insert the new layer and corresponding parameters for the new layer based on the final hidden state of the encoder network given the input architecture.

or tanh activation, which could be useful compared to random initialization. Additionally, when using batch normalization (Ioffe and Szegedy 2015), we need to set output scale and output bias of the batch normalization layer to undo the normalization, rather than initialize them as ones and zeros. Further details about the Net2DeeperNet operation is provided in the original paper (Chen, Goodfellow, and Shlens 2015).

The structure of the Net2Deeper actor is shown in Figure 3, which is a recurrent network whose hidden state is initialized with the final hidden state of the encoder network. Similar to previous work (Baker et al. 2017), we allow the Net2Deeper actor to insert one new layer at each step. Specifically, we divide a CNN architecture into several blocks according to the pooling layers and Net2Deeper actor sequentially determines which block to insert the new layer, a specific index within the block and parameters of the new layer. For a new convolutional layer, the agent needs to determine the filter size and the stride while for a new fully-connected layer, no parameter prediction is needed. In CNN architectures, any fully-connected layer should be on the top of all convolutional and pooling layers. To avoid resulting in unreasonable architectures, if the Net2Deeper actor decides to insert a new layer after a fully-connected layer or the final global average pooling layer, the new layer is restricted to be a fully-connected layer, otherwise it must be a convolutional layer.

Function-preserving Transformation for DenseNet

The original Net2Net operations proposed in (Chen, Goodfellow, and Shlens 2015) are discussed under the scenarios where the network is arranged layer-by-layer, i.e. the output of a layer is only fed to its next layer. As such, in some modern CNN architectures where the output of a layer would be fed to multiple subsequent layers, such as DenseNet (Huang et al. 2017), directly applying the original Net2Net operations can be problematic. In this section, we introduce several extensions to the original Net2Net operations to enable function-preserving transformations for DenseNet.

Different from the plain CNN, in DenseNet, the l^{th} layer would receive the outputs of all preceding layers as input, which are concatenated on the channel dimension, denoted as $[\boldsymbol{O}_0, \boldsymbol{O}_1, \cdots, \boldsymbol{O}_{l-1}]$, while its output $\boldsymbol{O}_l$ would be fed to all subsequent layers.

Denote the kernel of the l^{th} layer as $\boldsymbol{K}_l$ with shape $(k_w^l, k_h^l, f_i^l, f_o^l)$. To replace the l^{th} layer with a wider layer that has $\hat{f}_o^l$ output channels while preserving the functionality, the creation of the new kernel $\hat{\boldsymbol{K}}_l$ in the l^{th} layer is the same as the original Net2WiderNet operation (see Eq. (1) and Eq. (2)). As such, the new output of the wider layer is $\hat{\boldsymbol{O}}_l$ with $\hat{\boldsymbol{O}}_l(j) = \boldsymbol{O}_l(G_l(j))$, where G_l is the random remapping function as defined in Eq. (1). Since the output of the l^{th} layer will be fed to all subsequent layers in DenseNet, the replication in $\hat{\boldsymbol{O}}_l$ will result in replication in the inputs of all layers after the l^{th} layer. As such, instead of only modifying the kernel of the next layer as done in the original Net2WiderNet operation, we need to modify the kernels of all subsequent layers in DenseNet. For the m^{th} layer where $m > l$, its input becomes $[\boldsymbol{O}_0, \cdots, \boldsymbol{O}_{l-1}, \hat{\boldsymbol{O}}_l, \boldsymbol{O}_{l+1}, \cdots, \boldsymbol{O}_{m-1}]$ after widening the l^{th} layer, thus from the perspective of m^{th} layer, the equivalent random remapping function $\hat{G}_m$ can be written as

$$\hat{G}_m(j) = \begin{cases} j & 1 \le j \le f_o^{0:l} \\ f_o^{0:l} + G_l(j) & f_o^{0:l} < j \le f_o^{0:l} + \hat{f}_o^l \\ j - \hat{f}_o^l + f_o^l & f_o^{0:l} + \hat{f}_o^l < j \le f_o^{0:m} + \hat{f}_o^l - f_o^l \end{cases}, \quad (4)$$

where $f_o^{0:l} = \sum_{v=0}^{l-1} f_o^v$ is the number of input channels for the l^{th} layer, the first part corresponds to $[\boldsymbol{O}_0, \cdots, \boldsymbol{O}_{l-1}]$, the second part corresponds to $[\hat{\boldsymbol{O}}_l]$, and the last part corresponds to $[\boldsymbol{O}_{l+1}, \cdots, \boldsymbol{O}_{m-1}]$. A simple example of $\hat{G}_m$ is given as

$$\hat{G}_m : \{1, \cdots, 5, \overbrace{6, 7, 8, 9}^{\hat{\boldsymbol{O}}_l}, 10, 11\} \rightarrow \{1, \cdots, 5, \overbrace{6, 7, 6, 6}^{\hat{\boldsymbol{O}}_l}, 8, 9\}$$
$$\text{where } G_l : \{1, 2, 3, 4\} \rightarrow \{1, 2, 1, 1\}.$$

Accordingly the new kernel of m^{th} layer can be given by Eq. (3) with G_l replaced with $\hat{G}_m$.

To insert a new layer in DenseNet, suppose the new layer is inserted after the l^{th} layer. Denote the output of the new layer as $\boldsymbol{O}_{\text{new}}$, and its input is $[\boldsymbol{O}_0, \boldsymbol{O}_1, \cdots, \boldsymbol{O}_l]$. Therefore, for the m^{th} $(m > l)$ layer, its new input after the insertion is $[\boldsymbol{O}_0, \boldsymbol{O}_1, \cdots, \boldsymbol{O}_l, \boldsymbol{O}_{\text{new}}, \boldsymbol{O}_{l+1}, \cdots, \boldsymbol{O}_{m-1}]$. To preserve the functionality, similar to the Net2WiderNet case, $\boldsymbol{O}_{\text{new}}$ should be the replication of some entries in $[\boldsymbol{O}_0, \boldsymbol{O}_1, \cdots, \boldsymbol{O}_l]$. It is possible, since the input of the new layer is $[\boldsymbol{O}_0, \boldsymbol{O}_1, \cdots, \boldsymbol{O}_l]$. Each filter in the new layer can be represented with a tensor, denoted as $\hat{\boldsymbol{F}}$ with shape $(k_w^{\text{new}}, k_h^{\text{new}}, f_i^{\text{new}} = f_o^{0:l+1})$, where k_w^{new} and k_h^{new} denote the width and height of the filter, and f_i^{new} is the number of input channels. To make the output of $\hat{\boldsymbol{F}}$ to be a replication of the n^{th} entry in $[\boldsymbol{O}_0, \boldsymbol{O}_1, \cdots, \boldsymbol{O}_l]$, we can set $\hat{\boldsymbol{F}}$ (using the special case that k_w^{new} = k_h^{new} = 3 for illustration) as the following

$$\hat{\boldsymbol{F}}[x, y, n] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{5}$$

while all other values in $\hat{\boldsymbol{F}}$ are set to be 0. Note that n can be chosen randomly from $\{1, \cdots, f_o^{0:l+1}\}$ for each filter. After all filters in the new layer are set, we can form an equivalent random remapping function for all subsequent layers as is done in Eq. (4) and modify their kernels accordingly.

Experiments and Results

In line with the previous work (Baker et al. 2017; Zoph and Le 2017; Real et al. 2017), we apply the proposed EAS on image benchmark datasets (CIFAR-10 and SVHN) to explore high performance CNN architectures for the image classification task[1]. Notice that the performances of the final designed models largely depend on the architecture space and the computational resources. In our experiments, we evaluate EAS in two different settings. In all cases, we use restricted computational resources (5 GPUs) compared to the previous work such as (Zoph and Le 2017) that used 800 GPUs. In the first setting, we apply EAS to explore the plain CNN architecture space, which purely consists of convolutional, pooling and fully-connected layers. While in the second setting, we apply EAS to explore the DenseNet architecture space.

Image Datasets

CIFAR-10 The CIFAR-10 dataset (Krizhevsky and Hinton 2009) consists of 50,000 training images and 10,000 test images. We use a standard data augmentation scheme that is widely used for CIFAR-10 (Huang et al. 2017), and denote the augmented dataset as C10+ while the original dataset is denoted as C10. For preprocessing, we normalized the images using the channel means and standard deviations. Following the previous work (Baker et al. 2017; Zoph and Le 2017), we randomly sample 5,000 images from the training set to form a validation set while using the remaining 45,000 images for training during exploring the architecture space.

SVHN The Street View House Numbers (SVHN) dataset (Netzer et al. 2011) contains 73,257 images in the original training set, 26,032 images in the test set, and 531,131 additional images in the extra training set. For preprocessing, we divide the pixel values by 255 and do not perform any data augmentation, as is done in (Huang et al. 2017). We follow (Baker et al. 2017) and use the original training set during the architecture search phase with 5,000 randomly sampled images as the validation set, while training the final discovered architectures using all the training data, including the original training set and extra training set.

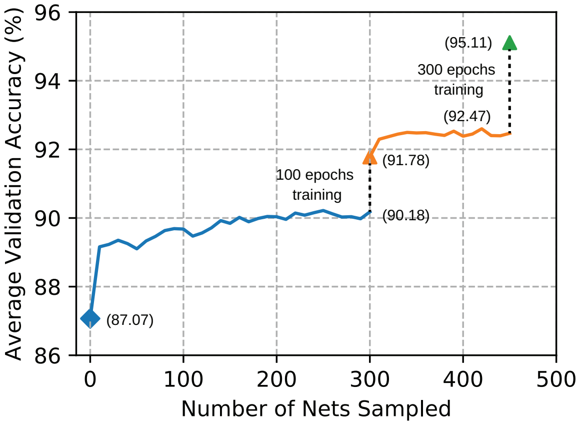

Figure 4: Progress of two stages architecture search on C10+ in the plain CNN architecture space.

Training Details

For the meta-controller, we use a one-layer bidirectional LSTM with 50 hidden units as the encoder network (Figure 1) with an embedding size of 16, and train it with the ADAM optimizer (Kingma and Ba 2015).

At each step, the meta-controller samples 10 networks by taking network transformation actions. Since the sampled networks are not trained from scratch but we reuse weights of the given network in our scenario, they are then trained for 20 epochs, a relative small number compared to 50 epochs in (Zoph and Le 2017). Besides, we use a smaller initial learning rate for this reason. Other settings for training networks on CIFAR-10 and SVHN, are similar to (Huang et al. 2017; Zoph and Le 2017). Specifically, we use the SGD with a Nesterov momentum (Sutskever et al. 2013) of 0.9, a weight decay of 0.0001, a batch size of 64. The initial learning rate is 0.02 and is further annealed with a cosine learning rate decay (Gastaldi 2017). The accuracy in the held-out validation set is used to compute the reward signal for each sampled network. Since the gain of improving the accuracy from 90% to 91% should be much larger than from 60% to 61%, instead of directly using the validation accuracy acc_v as the reward, as done in (Zoph and Le 2017), we perform a non-linear transformation on acc_v, i.e. $\tan(acc_v \times \pi/2)$, and use the transformed value as the reward. Additionally, we use an exponential moving average of previous rewards, with a decay of 0.95 as the baseline function to reduce the variance.

Explore Plain CNN Architecture Space

We start applying EAS to explore the plain CNN architecture space. Following the previous automatic architecture designing methods (Baker et al. 2017; Zoph and Le 2017), EAS searches layer parameters in a discrete and limited space. For every convolutional layer, the filter size is chosen from $\{1, 3, 5\}$ and the number of filters is chosen from $\{16, 32, 64, 96, 128, 192, 256, 320, 384, 448, 512\}$, while the stride is fixed to be 1 (Baker et al. 2017). For every fully-connected layer, the number of units is chosen from $\{64, 128, 256, 384, 512, 640, 768, 896, 1024\}$. Additionally,

[1] Experiment code and discovered top architectures along with weights: https://github.com/han-cai/EAS

Table 1: Simple start point network. $\mathrm{C}(n, f, l)$ denotes a convolutional layer with n filters, filter size f and stride l; $\mathrm{P}(f, l, \mathrm{MAX})$ and $\mathrm{P}(f, l, \mathrm{AVG})$ denote a max and an average pooling layer with filter size f and stride l respectively; $\mathrm{FC}(n)$ denotes a fully-connected layer with n units; $\mathrm{SM}(n)$ denotes a softmax layer with n output units.

Model Architecture	Validation Accuracy (%)
C(16, 3, 1), P(2, 2, MAX), C(32, 3, 1), P(2, 2, MAX), C(64, 3, 1), P(2, 2, MAX), C(128, 3, 1), P(4, 4, AVG), FC(256), SM(10)	87.07

we use ReLU and batch normalization for each convolutional or fully-connected layer. For SVHN, we add a dropout layer after each convolutional layer (except the first layer) and use a dropout rate of 0.2 (Huang et al. 2017).

Start with Small Network We begin the exploration on C10+, using a small network (see Table 1), which achieves 87.07% accuracy in the held-out validation set, as the start point. Different from (Zoph and Le 2017; Baker et al. 2017), EAS is not restricted to start from empty and can flexibly use any discovered architecture as the new start point. As such, to take the advantage of such flexibility and also reduce the search space for saving the computational resources and time, we divide the whole architecture search process into two stages where we allow the meta-controller to take 5 steps of Net2Deeper action and 4 steps of Net2Wider action in the first stage. After 300 networks are sampled, we take the network which performs best currently and train it with a longer period of time (100 epochs) to be used as the start point for the second stage. Similarly, in the second stage, we also allow the meta-controller to take 5 steps of Net2Deeper action and 4 steps of Net2Wider action and stop exploration after 150 networks are sampled.

The progress of the two stages architecture search is shown in Figure 4, where we can find that EAS gradually learns to pick high performance architectures at each stage. As EAS takes function-preserving transformations to explore the architecture space, we can also find that the sampled architectures consistently perform better than the start point network at each stage. Thus it is usually "safe" to explore the architecture space with EAS. We take the top networks discovered during the second stage and further train the networks with 300 epochs using the full training set. Finally, the best model achieves 95.11% test accuracy (i.e. 4.89% test error rate). Furthermore, to justify the transferability of the discovered networks, we train the top architecture (95.11% test accuracy) on SVHN from random initialization with 40 epochs using the full training set and achieves 98.17% test accuracy (i.e. 1.83% test error rate), better than both human-designed and automatically designed architectures that are in the plain CNN architecture space (see Table 2).

We would like to emphasize that the required computational resources to achieve this result is much smaller than those required in (Zoph and Le 2017; Real et al. 2017). Specifically, it takes less than 2 days on 5 GeForce GTX 1080 GPUs with totally 450 networks trained to achieve 4.89% test error rate on C10+ starting from a small network.

Further Explore Larger Architecture Space To further search better architectures in the plain CNN architecture space, in the second experiment, we use the top architectures discovered in the first experiment, as the start points to explore a larger architecture space on C10+ and SVHN. This experiment on each dataset takes around 2 days on 5 GPUs.

Table 2: Test error rate (%) comparison with CNNs that use convolutional, fully-connected and pooling layers alone.

	Model	C10+	SVHN
human designed	Maxout (Goodfellow et al. 2013)	9.38	2.47
	NIN (Lin, Chen, and Yan 2013)	8.81	2.35
	All-CNN (Springenberg et al. 2014)	7.25	-
	VGGnet (Simonyan and Zisserman 2015)	7.25	-
auto designed	MetaQNN (Baker et al. 2017) (depth=7)	6.92	-
	MetaQNN (Baker et al. 2017) (ensemble)	-	2.06
	EAS (plain CNN, depth=16)	4.89	1.83
	EAS (plain CNN, depth=20)	**4.23**	**1.73**

The summarized results of comparing with human-designed and automatically designed architectures that use a similar design scheme (plain CNN), are reported in Table 2, where we can find that the top model designed by EAS on the plain CNN architecture space outperforms all similar models by a large margin. Specifically, comparing to human-designed models, the test error rate drops from 7.25% to 4.23% on C10+ and from 2.35% to 1.73% on SVHN. While comparing to MetaQNN, the Q-learning based automatic architecture designing method, EAS achieves a relative test error rate reduction of 38.9% on C10+ and 16.0% on SVHN. We also notice that the best model designed by MetaQNN on C10+ only has a depth of 7, though the maximum is set to be 18 in the original paper (Baker et al. 2017). We suppose maybe they trained each designed network from scratch and used an aggressive training strategy to accelerate training, which resulted in many networks under performed, especially for deep networks. Since we reuse the weights of pre-existing networks, the deep networks are validated more accurately in EAS, and we can thus design deeper and more accurate networks than MetaQNN.

We also report the comparison with state-of-the-art architectures that use advanced techniques such as skip-connections, branching etc., on C10+ in Table 3. Though it is not a fair comparison since we do not incorporate such advanced techniques into the search space in this experiment, we still find that the top model designed by EAS is highly competitive even comparing to these state-of-the-art modern architectures. Specifically, the 20-layers plain CNN with 23.4M parameters outperforms ResNet, its stochastic depth variant and its pre-activation variant. It also approaches the best result given by DenseNet. When comparing to automatic architecture designing methods that in-

Table 3: Test error rate (%) comparison with state-of-the-art architectures.

	Model	Depth	Params	C10+
human designed	ResNet (He et al. 2016a)	110	1.7M	6.61
	ResNet (stochastic depth) (Huang et al. 2017)	1202	10.2M	4.91
	Wide ResNet (Zagoruyko and Komodakis 2016)	16	11.0M	4.81
	Wide ResNet (Zagoruyko and Komodakis 2016)	28	36.5M	4.17
	ResNet (pre-activation) (He et al. 2016b)	1001	10.2M	4.62
	DenseNet ($L = 40, k = 12$) (Huang et al. 2017)	40	1.0M	5.24
	DenseNet-BC ($L = 100, k = 12$) (Huang et al. 2017)	100	0.8M	4.51
	DenseNet-BC ($L = 190, k = 40$) (Huang et al. 2017)	190	25.6M	**3.46**
auto designed	Large-Scale Evolution (250 GPUs)(Real et al. 2017)	-	5.4M	5.40
	NAS (predicting strides, 800 GPUs) (Zoph and Le 2017)	20	2.5M	6.01
	NAS (max pooling, 800 GPUs) (Zoph and Le 2017)	39	7.1M	4.47
	NAS (post-processing, 800 GPUs) (Zoph and Le 2017)	39	37.4M	**3.65**
	EAS (plain CNN, 5 GPUs)	20	23.4M	4.23

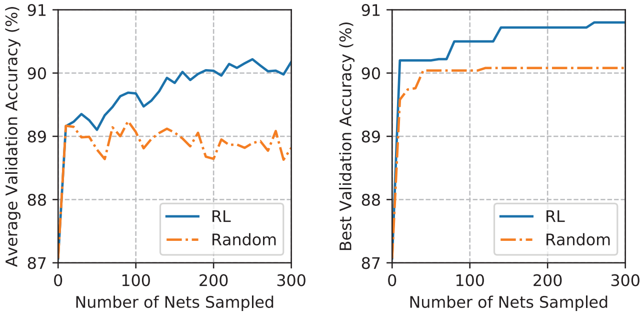

Figure 5: Comparison between RL based meta-controller and random search on C10+.

Table 4: Test error rate (%) results of exploring DenseNet architecture space with EAS.

Model	Depth	Params	C10	C10+
DenseNet ($L = 100, k = 24$)	100	27.2M	5.83	3.74
DenseNet-BC ($L = 250, k = 24$)	250	15.3M	5.19	3.62
DenseNet-BC ($L = 190, k = 40$)	190	25.6M	-	3.46
NAS (post-processing)	39	37.4M	-	3.65
EAS (DenseNet on C10)	70	8.6M	**4.66**	-
EAS (DenseNet on C10+)	76	10.7M	-	**3.44**

corporate skip-connections into their search space, our 20-layers plain model beats most of them except NAS with post-processing, that is much deeper and has more parameters than our model. Moreover, we only use 5 GPUs and train hundreds of networks while they use 800 GPUs and train tens of thousands of networks.

Comparison Between RL and Random Search Our framework is not restricted to use the RL based meta-controller. Beside RL, one can also take network transformation actions to explore the architecture space by random search, which can be effective in some cases (Bergstra and Bengio 2012). In this experiment, we compare the performances of the RL based meta-controller and the random search meta-controller in the architecture space that is used in the above experiments. Specifically, we use the network in Table 1 as the start point and let the meta-controller to take 5 steps of Net2Deeper action and 4 steps of Net2Wider action. The result is reported in Figure 5, which shows that the RL based meta-controller can effectively focus on the right search direction, while the random search cannot (left plot), and thus find high performance architectures more efficiently than random search.

Explore DenseNet Architecture Space

We also apply EAS to explore the DenseNet architecture space. We use the DenseNet-BC ($L = 40, k = 40$) as the start point. The growth rate, i.e. the width of the non-bottleneck layer is chosen from $\{40, 44, 48, 52, 56, 60, 64\}$, and the result is reported in Table 4. We find that by applying EAS to explore the DenseNet architecture space, we achieve a test error rate of 4.66% on C10, better than the best result, i.e. 5.19% given by the original DenseNet while having 43.79% less parameters. On C10+, we achieve a test error rate of 3.44%, also outperforming the best result, i.e. 3.46% given by the original DenseNet while having 58.20% less parameters.

Conclusion

In this paper, we presented EAS, a new framework toward economical and efficient architecture search, where the meta-controller is implemented as a RL agent. It learns to take actions for network transformation to explore the architecture space. By starting from an existing network and reusing its weights via the class of function-preserving transformation operations, EAS is able to utilize knowledge stored in previously trained networks and take advantage of the existing successful architectures in the target task to explore the architecture space efficiently. Our experiments have demonstrated EAS's outstanding performance and efficiency compared with several strong baselines. For future work, we would like to explore more network transformation operations and apply EAS for different purposes such as searching networks that not only have high accuracy but also keep a balance between the size and the performance.

Acknowledgments

This research was sponsored by Huawei Innovation Research Program, NSFC (61702327) and Shanghai Sailing Program (17YF1428200).

References

Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. *ICLR*.

Baker, B.; Gupta, O.; Naik, N.; and Raskar, R. 2017. Designing neural network architectures using reinforcement learning. *ICLR*.

Bergstra, J., and Bengio, Y. 2012. Random search for hyper-parameter optimization. *JMLR*.

Cai, H.; Ren, K.; Zhang, W.; Malialis, K.; Wang, J.; Yu, Y.; and Guo, D. 2017. Real-time bidding by reinforcement learning in display advertising. In *WSDM*.

Chen, T.; Goodfellow, I.; and Shlens, J. 2015. Net2net: Accelerating learning via knowledge transfer. *ICLR*.

Domhan, T.; Springenberg, J. T.; and Hutter, F. 2015. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*.

Gastaldi, X. 2017. Shake-shake regularization. *arXiv preprint arXiv:1705.07485*.

Goodfellow, I. J.; Warde-Farley, D.; Mirza, M.; Courville, A.; and Bengio, Y. 2013. Maxout networks. *ICML*.

Han, S.; Pool, J.; Tran, J.; and Dally, W. 2015. Learning both weights and connections for efficient neural network. In *NIPS*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016a. Deep residual learning for image recognition. In *CVPR*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016b. Identity mappings in deep residual networks. In *ECCV*.

Huang, G.; Liu, Z.; Weinberger, K. Q.; and van der Maaten, L. 2017. Densely connected convolutional networks. *CVPR*.

Ioffe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ICML*.

Kakade, S. 2002. A natural policy gradient. *NIPS*.

Kingma, D., and Ba, J. 2015. Adam: A method for stochastic optimization. *ICLR*.

Klein, A.; Falkner, S.; Springenberg, J. T.; and Hutter, F. 2017. Learning curve prediction with bayesian neural networks. *ICLR*.

Krizhevsky, A., and Hinton, G. 2009. Learning multiple layers of features from tiny images.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*.

Lin, M.; Chen, Q.; and Yan, S. 2013. Network in network. *arXiv preprint arXiv:1312.4400*.

Mendoza, H.; Klein, A.; Feurer, M.; Springenberg, J. T.; and Hutter, F. 2016. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*.

Miller, G. F.; Todd, P. M.; and Hegde, S. U. 1989. Designing neural networks using genetic algorithms. In *ICGA*. Morgan Kaufmann Publishers Inc.

Negrinho, R., and Gordon, G. 2017. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*.

Netzer, Y.; Wang, T.; Coates, A.; Bissacco, A.; Wu, B.; and Ng, A. Y. 2011. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*.

Real, E.; Moore, S.; Selle, A.; Saxena, S.; Suematsu, Y. L.; Le, Q.; and Kurakin, A. 2017. Large-scale evolution of image classifiers. *ICML*.

Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M. I.; and Moritz, P. 2015. Trust region policy optimization. In *ICML*.

Schuster, M., and Paliwal, K. K. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature*.

Simonyan, K., and Zisserman, A. 2015. Very deep convolutional networks for large-scale image recognition. *ICLR*.

Snoek, J.; Larochelle, H.; and Adams, R. P. 2012. Practical bayesian optimization of machine learning algorithms. In *NIPS*.

Springenberg, J. T.; Dosovitskiy, A.; Brox, T.; and Riedmiller, M. 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Stanley, K. O., and Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation*.

Sutskever, I.; Martens, J.; Dahl, G.; and Hinton, G. 2013. On the importance of initialization and momentum in deep learning. In *ICML*.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *NIPS*.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning: An introduction*. MIT press Cambridge.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*.

Zagoruyko, S., and Komodakis, N. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146*.

Zoph, B., and Le, Q. V. 2017. Neural architecture search with reinforcement learning. *ICLR*.