Path-Level Network Transformation for Efficient Architecture Search

Han Cai [1] **Jiacheng Yang** [1] **Weinan Zhang** [1] **Song Han** [2] **Yong Yu** [1]

Abstract

We introduce a new function-preserving transformation for efficient neural architecture search. This network transformation allows reusing previously trained networks and existing successful architectures that improves sample efficiency. We aim to address the limitation of current network transformation operations that can only perform layer-level architecture modifications, such as adding (pruning) filters or inserting (removing) a layer, which fails to change the topology of connection paths. Our proposed path-level transformation operations enable the meta-controller to modify the path topology of the given network while keeping the merits of reusing weights, and thus allow efficiently designing effective structures with complex path topologies like Inception models. We further propose a bidirectional tree-structured reinforcement learning meta-controller to explore a simple yet highly expressive tree-structured architecture space that can be viewed as a generalization of multi-branch architectures. We experimented on the image classification datasets with limited computational resources (about 200 GPU-hours), where we observed improved parameter efficiency and better test results (97.70% test accuracy on CIFAR-10 with 14.3M parameters and 74.6% top-1 accuracy on ImageNet in the mobile setting), demonstrating the effectiveness and transferability of our designed architectures.

1. Introduction

Designing effective neural network architectures is crucial for the performance of deep learning. While many impressive results have been achieved through significant manual architecture engineering (Simonyan & Zisserman, 2014; Szegedy et al., 2015; He et al., 2016; Huang et al., 2017b), this process typically requires years of extensive investigation by human experts, which is not only expensive but also likely to be suboptimal. Therefore, automatic architecture design has recently drawn much attention (Zoph & Le, 2017; Zoph et al., 2017; Liu et al., 2017; Cai et al., 2018; Real et al., 2018; Pham et al., 2018).

Most of the current techniques focus on finding the optimal architecture in a designated search space starting from scratch while training each designed architecture on the real data (from random initialization) to get a validation performance to guide exploration. Though such methods have shown the ability to discover network structures that outperform human-designed architectures when vast computational resources are used, such as Zoph et al. (2017) that employed 500 P100 GPUs across 4 days, they are also likely to fail to beat best human-designed architectures (Zoph & Le, 2017; Real et al., 2017; Liu et al., 2018), especially when the computational resources are restricted. Furthermore, insufficient training epochs during the architecture search process (much fewer epochs than normal to save time) may cause models to underperform (Baker et al., 2017), which would harm the efficiency of the architecture search process.

Alternatively, some efforts have been made to explore the architecture space by network transformation, starting from an existing network trained on the target task and reusing its weights. For example, Cai et al. (2018) utilized Net2Net (Chen et al., 2016) operations, a class of function-preserving transformation operations, to further find high-performance architectures based on a given network, while Ashok et al. (2018) used network compression operations to compress well-trained networks. These methods allow transferring knowledge from previously trained networks and taking advantage of existing successful architectures in the target task, thus have shown improved efficiency and require significantly fewer computational resources (e.g., 5 GPUs in Cai et al. (2018)) to achieve competitive results.

However, the network transformation operations in Cai et al. (2018) and Ashok et al. (2018) are still limited to only performing *layer-level* architecture modifications such as adding (pruning) filters or inserting (removing) a layer, which does not change the topology of connection paths in a neural network. Hence, they restrict the search space to having the same path topology as the start network, i.e. they would always lead to chain-structured networks when given

[1]Shanghai Jiao Tong University, Shanghai, China [2]Massachusetts Institute of Technology, Cambridge, USA. Correspondence to: Han Cai <hcai@apex.sjtu.edu.cn>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

a chain-structured start point. As the state-of-the-art convolutional neural network (CNN) architectures have gone beyond simple chain-structured layout and demonstrated the effectiveness of multi-path structures such as Inception models (Szegedy et al., 2015), ResNets (He et al., 2016) and DenseNets (Huang et al., 2017b), we would hope such methods to have the ability to explore a search space with different and complex path topologies while keeping the benefits of reusing weights.

In this paper, we present a new kind of transformation operations for neural networks, phrased as *path-level* network transformation operations, which allows modifying the path topologies in a given network while allowing weight reusing to preserve the functionality like Net2Net operations (Chen et al., 2016). Based on the proposed path-level operations, we introduce a simple yet highly expressive tree-structured architecture space that can be viewed as a generalized version of multi-branch structures. To efficiently explore the introduced tree-structured architecture space, we further propose a bidirectional tree-structured (Tai et al., 2015) reinforcement learning meta-controller that can naturally encode the input tree, instead of simply using the chain-structured recurrent neural network (Zoph et al., 2017).

Our experiments of learning CNN cells on CIFAR-10 show that our method using restricted computational resources (about 200 GPU-hours) can design highly effective cell structures. When combined with state-of-the-art human-designed architectures such as DenseNets (Huang et al., 2017b) and PyramidNets (Han et al., 2017), the best discovered cell shows significantly improved parameter efficiency and better results compared to the original ones. Specifically, without any additional regularization techniques, it achieves 3.14% test error with 5.7M parameters, while DensNets give a best test error rate of 3.46% with 25.6M parameters and PyramidNets give 3.31% with 26.0M parameters. And with additional regularization techniques (DropPath (Zoph et al., 2017) and Cutout (DeVries & Taylor, 2017)), it reaches 2.30% test error with 14.3M parameters, surpassing 2.40% given by NASNet-A (Zoph et al., 2017) with 27.6M parameters and a similar training scheme. More importantly, NASNet-A is achieved using 48,000 GPU-hours while we only use 200 GPU-hours. We further apply the best learned cells on CIFAR-10 to the ImageNet dataset by combining it with CondenseNet (Huang et al., 2017a) for the $Mobile$ setting and also observe improved results when compared to models in the mobile setting.

2. Related Work and Background

2.1. Architecture Search

Architecture search that aims to automatically find effective model architectures in a given architecture space has been studied using various approaches which can be categorized as neuro-evolution (Real et al., 2017; Liu et al., 2018), Bayesian optimization (Domhan et al., 2015; Mendoza et al., 2016), Monte Carlo Tree Search (Negrinho & Gordon, 2017) and reinforcement learning (RL) (Zoph & Le, 2017; Baker et al., 2017; Zhong et al., 2017; Zoph et al., 2017).

Since getting an evaluation of each designed architecture requires training on the real data, which makes directly applying architecture search methods on large datasets (e.g., ImageNet (Deng et al., 2009)) computationally expensive, Zoph et al. (2017) proposed to search for CNN cells that can be stacked later, rather than search for the entire architectures. Specifically, learning of the cell structures is conducted on small datasets (e.g., CIFAR-10) while learned cell structures are then transferred to large datasets (e.g., ImageNet). This scheme has also been incorporated in Zhong et al. (2017) and Liu et al. (2018).

On the other hand, instead of constructing and evaluating architectures from scratch, there are some recent works that proposed to take network transformation operations to explore the architecture space given a trained network in the target task and reuse the weights. Cai et al. (2018) presented a recurrent neural network to iteratively generate transformation operations to be performed based on the current network architecture, and trained the recurrent network with REINFORCE algorithm (Williams, 1992). A similar framework has also been incorporated in Ashok et al. (2018) where the transformation operations change from Net2Net operations in Cai et al. (2018) to compression operations.

Compared to above work, in this paper, we extend current network transformation operations from layer-level to path-level. Similar to Zoph et al. (2017) and Zhong et al. (2017), we focus on learning CNN cells, while our approach can be easily combined with any existing well-designed architectures to take advantage of their success and allow reusing weights to preserve the functionality.

2.2. Multi-Branch Neural Networks

Multi-branch structure (or motif) is an essential component in many modern state-of-the-art CNN architectures. The family of Inception models (Szegedy et al., 2015; 2017; 2016) are successful multi-branch architectures with carefully customized branches. ResNets (He et al., 2016) and DenseNets (Huang et al., 2017b) can be viewed as two-branch architectures where one branch is the identity mapping. A common strategy within these multi-branch architectures is that the input feature map $\boldsymbol{x}$ is first distributed to each branch based on a specific allocation scheme (either $split$ in Inception models or $replication$ in ResNets and DenseNets), then transformed by primitive operations (e.g., convolution, pooling, etc.) on each branch, and fi-

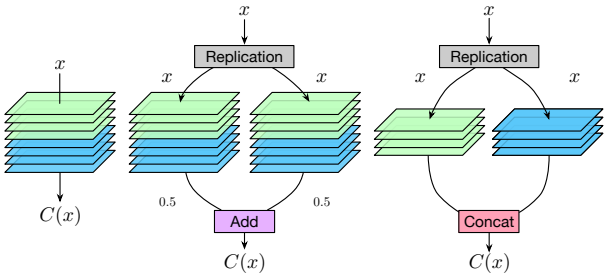


Figure 1. Convolution layer and its equivalent multi-branch motifs.

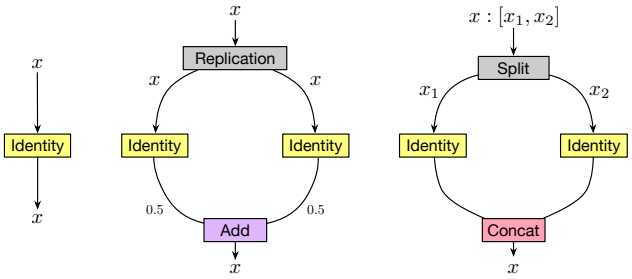


Figure 2. Identity layer and its equivalent multi-branch motifs.

nally aggregated to produce an output based on a specific merge scheme (either *add* in ResNets or *concatenation* in Inception models and DenseNets).

According to the research of Veit et al. (2016), ResNets can be considered to behave as ensembles of a collection of many paths of differing length. Similar interpretations can also be applied to Inception models and DenseNets. As the Inception models have demonstrated the merits of carefully customized branches where different primitive operations are used in each branch, it is thus of great interest to investigate whether we can benefit from more complex and well-designed path topologies within a CNN cell that make the collection of paths from the ensemble view more abundant and diverse.

In this work, we explore a tree-structured architecture space where at each node the input feature map is allocated to each branch, going through some primitive operations and the corresponding child node, and is later merged to produce an output for the node. It can be viewed as a generalization of current multi-branch architectures (tree with a depth of 1) and is able to embed plentiful paths within a CNN cell.

2.3. Function-Preserving Network Transformation

Function-preserving network transformation refers to the class of network transformation operations that initialize a student network to preserve the functionality of a given teacher network. Net2Net technique (Chen et al., 2016) introduces two specific function-preserving transformation operations, namely Net2WiderNet operation which replaces a layer with an equivalent layer that is wider (e.g., more filters for convolution layer) and Net2DeeperNet operation which replaces an identity mapping with a layer that can be initialized to be identity, including normal convolution layers with various filters (e.g., 3×3, 7×1, 1×7, etc.), depthwise-separable convolution layers (Chollet, 2016) and so on. Additionally, network compression operations (Han et al., 2015) that prune less important connections (e.g., low weight connections) to shrink the size of a given model without reducing the performance can also be viewed as one kind of function-preserving transformation operations.

Our approach builds on existing function-preserving transformation operations and further extends to path-level architecture modifications.

3. Method

3.1. Path-Level Network Transformation

We introduce operations that allow replacing a single layer with a multi-branch motif whose merge scheme is either *add* or *concatenation*. To illustrate the operations, we use two specific types of layers, i.e. identity layer and normal convolution layer, as examples, while they can also be applied to other similar types of layers, such as depthwise-separable convolution layers, analogously.

For a convolution layer, denoted as $C(\cdot)$, to construct an equivalent multi-branch motif with N branches, we need to set the branches so as to mimic the output of the original layer for any input feature map $\boldsymbol{x}$. When these branches are merged by *add*, the allocation scheme is set to be *replication* and we set each branch to be a replication of the original layer $C(\cdot)$, which makes each branch produce the same output (i.e. $C(\boldsymbol{x})$), and finally results in an output $N \times C(\boldsymbol{x})$ after being merged by *add*. To eliminate the factor, we further divide the output of each branch by N. As such the output of the multi-branch motif keeps the same as the output of the original convolution layer, as illustrated in Figure 1 (middle). When these branches are merged by *concatenation*, the allocation scheme is also set to be *replication*. Then we split the filters of the original convolution layer into N parts along the output channel dimension and assign each part to the corresponding branch, which is later merged to produce an output $C(\boldsymbol{x})$, as shown in Figure 1 (right).

For an identity layer, when the branches are merged by *add*, the transformation is the same except that the convolution layer in each branch changes to the identity mapping in this case (Figure 2 (middle)). When the branches are merged by *concatenation*, the allocation scheme is set to be *split* and each branch is set to be the identity mapping, as is illustrated in Figure 2 (right).

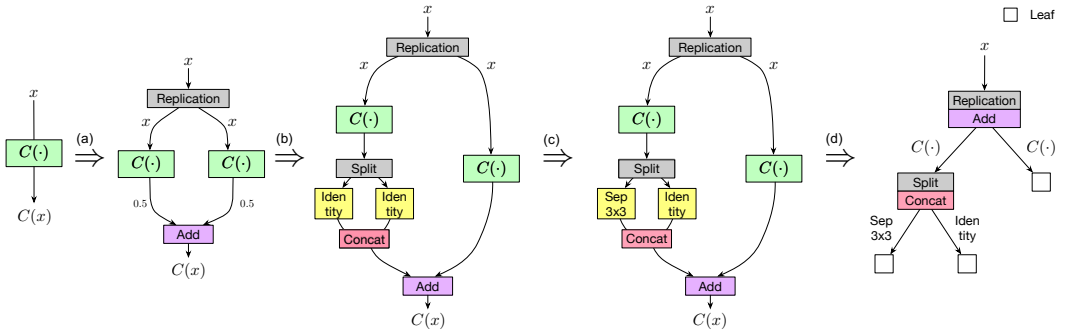


Figure 3. An illustration of transforming a single layer to a tree-structured motif via path-level transformation operations, where we apply Net2DeeperNet operation to replace an identity mapping with a 3×3 depthwise-separable convolution in (c).

Note that simply applying the above transformations does not lead to non-trivial path topology modifications. However, when combined with Net2Net operations, we are able to dramatically change the path topology, as shown in Figure 3. For example, we can insert different numbers and types of layers into each branch by applying Net2DeeperNet operation, which makes each branch become substantially different, like Inception Models. Furthermore, since such transformations can be repetitively applied on any applicable layers in the neural network, such as a layer in the branch, we can thus arbitrarily increase the complexity of the path topology.

3.2. Tree-Structured Architecture Space

In this section, we describe the tree-structured architecture space that can be explored with path-level network transformation operations as illustrated in Figure 3.

A tree-structured architecture consists of edges and nodes, where at each node (except leaf nodes) we have a specific combination of the allocation scheme and the merge scheme, and the node is connected to each of its child nodes via an edge that is defined as a primitive operation such as convolution, pooling, etc. Given the input feature map $\boldsymbol{x}$, the output of node $N(\cdot)$, with m child nodes denoted as $\{N_i^c(\cdot)\}$ and m corresponding edges denoted as $\{E_i(\cdot)\}$, is defined recursively based on the outputs of its child nodes:

$$\begin{aligned} \boldsymbol{z}_i &= allocation(\boldsymbol{x}, i), \\ \boldsymbol{y}_i &= N_i^c(E_i(\boldsymbol{z}_i)), \quad 1 \leq i \leq m, \\ N(x) &= merge(\boldsymbol{y}_1, \cdots, \boldsymbol{y}_m), \end{aligned} \tag{1}$$

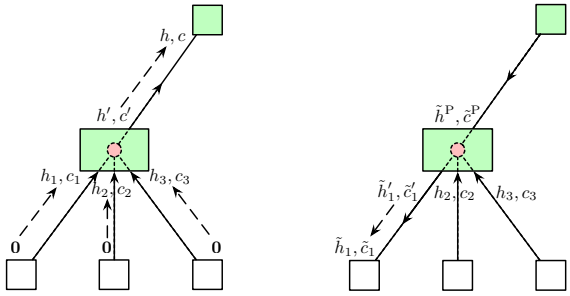
where $allocation(\boldsymbol{x}, i)$ denotes the allocated feature map for i^{th} child node based on the allocation scheme, and $merge(\cdot)$ denotes the merge scheme that takes the outputs of child nodes as input and outputs an aggregated result which is also the output of the node. For a leaf node that has no child node, it simply returns the input feature map as its output. As defined in Eq. (1), for a tree-structured architecture, the feature map is first fed to its root node, then spread to all subsequent nodes through allocation schemes at the nodes and edges in a top-down manner until reaching leaf nodes, and finally aggregated in mirror from the leaf nodes to the root node in a bottom-up manner to produce a final output feature map.

Notice that the tree-structured architecture space is not the full architecture space that can be achieved with the proposed path-level transformation operations. We choose to explore the tree-structure architecture space for the ease of implementation and further applying architecture search methods such as RL based approaches (Cai et al., 2018) that would need to encode the architecture. Another reason for choosing the tree-structured architecture space is that it has a strong connection to existing multi-branch architectures, which can be viewed as tree-structured architectures with a depth of 1, i.e. all of the root node's child nodes are leaf.

To apply architecture search methods on the tree-structured architecture space, we need to further specify it by defining the set of possible allocation schemes, merge schemes and primitive operations. As discussed in Sections 2.2 and 3.1, the allocation scheme is either $replication$ or $split$ and the merge scheme is either add or $concatenation$. For the primitive operations, similar to previous work (Zoph et al., 2017; Liu et al., 2018), we consider the following 7 types of layers:

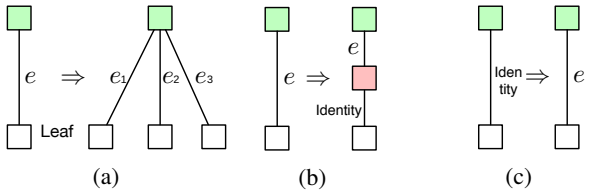
- 1×1 convolution
- Identity
- 3×3 depthwise-separable convolution
- 5×5 depthwise-separable convolution
- 7×7 depthwise-separable convolution
- 3×3 average pooling
- 3×3 max pooling

Here, we include pooling layers that cannot be initialized as identity mapping. To preserve the functionality when pooling layers are chosen, we further reconstruct the weights in the student network (i.e. the network after transformations) to mimic the output logits of the given teacher network, using the idea of knowledge distillation (Hinton et al., 2015). As pooling layers do not dramatically destroy the functionality for multi-path neural networks, we find that the reconstruction process can be done with negligible cost.



(a) Bottom-up (b) Top-down

Figure 4. Calculation procedure of bottom-up and top-down hidden states.



(a) (b) (c)

Figure 5. Illustration of transformation decisions on nodes and edges. (a) The meta-controller transforms a node with only one leaf child node to have multiple child nodes. Both merge scheme and branch number are predicted. (b) The meta-controller inserts a new leaf node to be the child node of a previous leaf node and they are connected with an identity mapping. (c) The meta-controller replaces an identity mapping with a layer (can be identity) chosen from the set of possible primitive operations.

3.3. Architecture Search with Path-Level Operations

In this section, we present an RL agent as the meta-controller to explore the tree-structured architecture space. The overall framework is similar to the one proposed in Cai et al. (2018) where the meta-controller iteratively samples network transformation actions to generate new architectures that are later trained to get the validation performances as reward signals to update the meta-controller via policy gradient algorithms. To map the input architecture to transformation actions, the meta-controller has an encoder network that learns a low-dimensional representation of the given architecture, and distinct softmax classifiers that generate corresponding network transformation actions.

In this work, as the input architecture now has a tree-structured topology that cannot be easily specified with a sequence of tokens, instead of using the chain-structure Long Short-Term Memory (LSTM) network (Hochreiter & Schmidhuber, 1997) to encode the architecture (Zoph et al., 2017), we propose to use a tree-structured LSTM. Tai et al. (2015) introduced two kinds of tree-structured LSTM units, i.e. Child-Sum Tree-LSTM unit for tree structures whose child nodes are unordered and N-ary Tree-LSTM unit for tree structures whose child nodes are ordered. For further details, we refer to the original paper (Tai et al., 2015).

In our case, for the node whose merge scheme is add, its child nodes are unordered and thereby the Child-Sum Tree-LSTM unit is applied, while for the node whose merge scheme is $concatenation$, the N-ary Tree-LSTM unit is used since its child nodes are ordered. Additionally, as we have edges between nodes, we incorporate another normal LSTM unit for performing hidden state transitions on edges. We denote these three LSTM units as $ChildSumLSTM^{\uparrow}$, $NaryLSTM^{\uparrow}$ and $LSTM^{\uparrow}$, respectively. As such, the hidden state of the node that has m child nodes is given as

$$h', c' = \begin{cases} ChildSumLSTM^{\uparrow}(s, [h_1^c, c_1^c], \cdots, [h_m^c, c_m^c]) & \text{if add} \\ NaryLSTM^{\uparrow}(s, [h_1^c, c_1^c], \cdots, [h_m^c, c_m^c]) & \text{if concat} \end{cases},$$
$$h, c = LSTM^{\uparrow}(e, [h', c']), \tag{2}$$

where $[h_i^c, c_i^c]$ denotes the hidden state of i^{th} child node, s represents the allocation and merge scheme of the node, e is the edge that connects the node to its parent node, and $[h, c]$ is the hidden state of the node. Such calculation is done in a bottom-up manner as is shown in Figure 4a.

Note that the hidden state calculated via Eq. (2) only contains information below the node. Analogous to bidirectional LSTM, we further consider a top-down procedure, using two new LSTM units ($NaryLSTM^{\downarrow}$ and $LSTM^{\downarrow}$), to calculate another hidden state for each node. We refer to these two hidden states of a node as bottom-up hidden state and top-down hidden state respectively. For a node, with m child nodes, whose top-down hidden state is $[\tilde{h}^p, \tilde{c}^p]$, the top-down hidden state of its i^{th} child node is given as

$$\tilde{h}'_i, \tilde{c}'_i = NaryLSTM^{\downarrow}(s, [\tilde{h}^p, \tilde{c}^p], [h_1, c_1], \cdots, \overbrace{[\mathbf{0}, \mathbf{0}]}^{i^{th}\text{ child node}}, \cdots),$$
$$\tilde{h}_i, \tilde{c}_i = LSTM^{\downarrow}(e_i, [\tilde{h}'_i, \tilde{c}'_i]), \tag{3}$$

where $[h_j, c_j]$ is the bottom-up hidden state of j^{th} child node, s is the allocation and merge scheme of the node, e_i is the edge that connect the node to its i^{th} child node, and $[\tilde{h}_i, \tilde{c}_i]$ is the top-down hidden state of i^{th} child node. As shown in Figure 4b and Eq. (3), a combination of the bottom-up hidden state and top-down hidden state now forms a comprehensive hidden state for each node, containing all information of the architecture.

Given the hidden state at each node, we have various softmax classifiers for making different transformation decisions on applicable nodes as follows:

1. For a node that has only one leaf child node, the meta-controller chooses a merge scheme from $\{add, concatenation, none\}$. When add or $concatenation$ is chosen, the meta-controller further chooses the number of branches and then the network is transformed accordingly, which makes the node have multiple child nodes now (Figure 5a). When $none$ is chosen, nothing is done and the meta-controller will not make such decision on that node again.

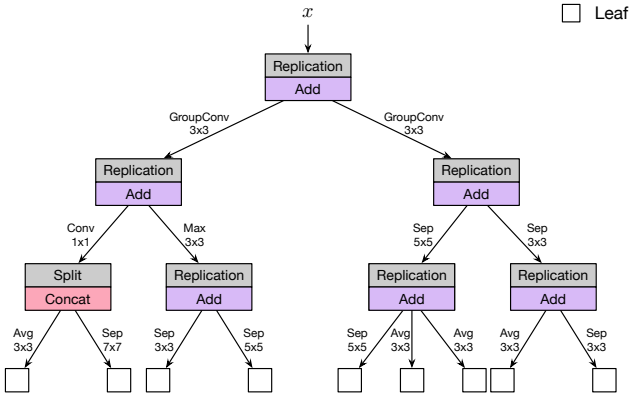


Figure 6. Detailed structure of the best discovered cell on CIFAR-10 (TreeCell-A). "GroupConv" denotes the group convolution; "Conv" denotes the normal convolution; "Sep" denotes the depthwise-separable convolution; "Max" denotes the max pooling; "Avg" denotes the average pooling.

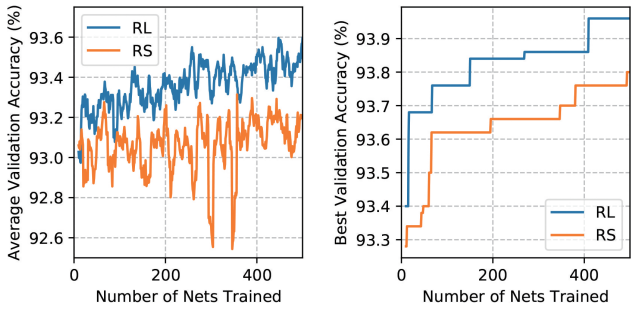


Figure 7. Progress of the architecture search process and comparison between RL and random search (RS) on CIFAR-10.

2. For a node that is a leaf node, the meta-controller determines whether to expand the node, i.e. insert a new leaf node to be the child node of this node and connect them with identity mapping, which increases the depth of the architecture (Figure 5b).

3. For an identity edge, the meta-controller chooses a new edge (can be identity) from the set of possible primitive operations (Section 3.2) to replace the identity edge (Figure 5c). Also this decision will only be made once for each edge.

4. Experiments and Results

Our experimental setting[1] resembles Zoph et al. (2017), Zhong et al. (2017) and Liu et al. (2018). Specifically, we apply the proposed method described above to learn CNN cells on CIFAR-10 (Krizhevsky & Hinton, 2009) for the image classification task and transfer the learned cell structures to ImageNet dataset (Deng et al., 2009).

4.1. Experimental Details

CIFAR-10 contains 50,000 training images and 10,000 test images, where we randomly sample 5,000 images from the training set to form a validation set for the architecture search process, similar to previous work (Zoph et al., 2017; Cai et al., 2018). We use a standard data augmentation scheme (mirroring/shifting) that is widely used for this dataset (Huang et al., 2017b; Han et al., 2017; Cai et al., 2018) and normalize the images using channel means and standard deviations for preprocessing.

For the meta-controller, described in Section 3.3, the hidden state size of all LSTM units is 100 and we train it with the ADAM optimizer (Kingma & Ba, 2014) using the REINFORCE algorithm (Williams, 1992). To reduce variance, we adopt a baseline function which is an exponential moving average of previous rewards with a decay of 0.95, as done in Cai et al. (2018). We also use an entropy penalty with a weight of 0.01 to ensure exploration.

At each step in the architecture search process, the meta-controller samples a tree-structured cell by taking transformation actions starting with a single layer in the base network. For example, when using a DenseNet as the base network, after the transformations, all 3×3 convolution layers in the dense blocks are replaced with the sampled tree-structured cell while all the others remain unchanged. The obtained network, along with weights transferred from the base network, is then trained for 20 epochs on CIFAR-10 with an initial learning rate of 0.035 that is further annealed with a cosine learning rate decay (Loshchilov & Hutter, 2016), a batch size of 64, a weight decay of 0.0001, using the SGD optimizer with a Nesterov momentum of 0.9. The validation accuracy acc_v of the obtained network is used to compute a reward signal. We follow Cai et al. (2018) and use the transformed value, i.e. $tan(acc_v \times \pi/2)$, as the reward since improving the accuracy from 90% to 91% should gain much more than from 60% to 61%. Additionally, we update the meta-controller with mini-batches of 10 architectures.

After the architecture search process is done, the learned cell structures can be embedded into various kinds of base networks (e.g., ResNets, DenseNets, etc.) with different depth and width. In this stage, we train networks for 300 epochs with an initial learning rate of 0.1, while all other settings keep the same.

4.2. Architecture Search on CIFAR-10

In our experiments, we use a small DenseNet-BC ($N = 2, L = 16, k = 48, G = 4$)[2], which achieves an accuracy of

[1] Experiment code: https://github.com/han-cai/PathLevel-EAS

[2] N, L and k respectively indicate the number of 3×3 convolution layers within each dense block, the depth of the network, and

Table 1. Test error rate (%) results of our best discovered architectures as well as state-of-the-art human-designed and automatically designed architectures on CIFAR-10. If "Reg" is checked, additional regularization techniques (e.g., Shake-Shake (Gastaldi, 2017), DropPath (Zoph et al., 2017) and Cutout (DeVries & Taylor, 2017)), along with a longer training schedule (600 epochs or 1800 epochs) are utilized when training the networks.

	Model	Reg	Params	Test error
Human designed	ResNeXt-29 (16×64d) (Xie et al., 2017)		68.1M	3.58
	DenseNet-BC ($N = 31, k = 40$) (Huang et al., 2017b)		25.6M	3.46
	PyramidNet-Bottleneck ($N = 18, \alpha = 270$) (Han et al., 2017)		27.0M	3.48
	PyramidNet-Bottleneck ($N = 30, \alpha = 200$) (Han et al., 2017)		26.0M	3.31
	ResNeXt + Shake-Shake (1800 epochs) (Gastaldi, 2017)	✓	26.2M	2.86
	ResNeXt + Shake-Shake + Cutout (1800 epochs) (DeVries & Taylor, 2017)	✓	26.2M	2.56
Auto designed	EAS (plain CNN) (Cai et al., 2018)		23.4M	4.23
	Hierarchical ($c_0 = 128$) (Liu et al., 2018)		-	3.63
	Block-QNN-A ($N = 4$) (Zhong et al., 2017)		-	3.60
	NAS v3 (Zoph & Le, 2017)		37.4M	3.65
	NASNet-A (6, 32) + DropPath (600 epochs) (Zoph et al., 2017)	✓	3.3M	3.41
	NASNet-A (6, 32) + DropPath + Cutout (600 epochs) (Zoph et al., 2017)	✓	3.3M	2.65
	NASNet-A (7, 96) + DropPath + Cutout (600 epochs) (Zoph et al., 2017)	✓	27.6M	2.40
Ours	TreeCell-B with DenseNet ($N = 6, k = 48, G = 2$)		3.2M	3.71
	TreeCell-A with DenseNet ($N = 6, k = 48, G = 2$)		3.2M	3.64
	TreeCell-A with DenseNet ($N = 16, k = 48, G = 2$)		13.1M	3.35
	TreeCell-B with PyramidNet ($N = 18, \alpha = 84, G = 2$)		5.6M	3.40
	TreeCell-A with PyramidNet ($N = 18, \alpha = 84, G = 2$)		5.7M	3.14
	TreeCell-A with PyramidNet ($N = 18, \alpha = 84, G = 2$) + DropPath (600 epochs)	✓	5.7M	2.99
	TreeCell-A with PyramidNet ($N = 18, \alpha = 84, G = 2$) + DropPath + Cutout (600 epochs)	✓	5.7M	2.49
	TreeCell-A with PyramidNet ($N = 18, \alpha = 150, G = 2$) + DropPath + Cutout (600 epochs)	✓	14.3M	**2.30**

93.12% on the held-out validation set, as the base network to learn cell structures. We set the maximum depth of the cell structures to be 3, i.e. the length of the path from the root node to each leaf node is no larger than 3 (Figure 6). For nodes whose merge scheme is add, the number of branches is chosen from $\{2, 3\}$ while for nodes whose merge scheme is $concatenation$, the number of branches is set to be 2. Additionally, we use very restricted computational resources for this experiment (about 200 GPU-hours $\ll$ 48,000 GPU-hours in Zoph et al. (2017)) with in total 500 networks trained.

The progress of the architecture search process is reported in Figure 7, where the results of random search (a very strong baseline for black-box optimization (Bergstra & Bengio, 2012)) under the same condition is also provided. We can find that the average validation accuracy of the designed architectures by the RL meta-controller gradually increases as the number of sampled architectures increases, as expected, while the curve of random search keeps fluctuating, which indicates that the RL meta-controller effectively focuses on the right search direction while random search fails. Therefore, with only 500 networks trained, the best model identified by RL, after 20 epochs training, achieves 0.16% better validation accuracy than the best model identified by random search.

We take top 10 candidate cells discovered in this experiment, and embed them into a relatively larger base network, i.e. DenseNet-BC ($N = 6, k = 48, G$) where G is chosen from $\{1, 2, 4\}$ to make different cells have a similar number of parameters as the normal 3×3 convolution layer (more details in the supplementary material). After 300 epochs training on CIFAR-10, the top 2 cells achieve 3.64% test error (TreeCell-A) and 3.71% test error (TreeCell-B), respectively. The detailed structure of TreeCell-A is given in Figure 6, while TreeCell-B's detailed structure is provided in the supplementary material. Under the same condition, the best cell given by random search reaches a test error rate of 3.98%, which is 0.34% worse than TreeCell-A.

4.3. Results on CIFAR-10

We further embed the top discovered cells, i.e. TreeCell-A and TreeCell-B, into larger base networks. Beside DenseNets, to justify whether the discovered cells starting with DenseNet can be transferred to other types of architectures such as ResNets, we also embed the cells into PyramidNets (Han et al., 2017), a variant of ResNets.

The summarized results are reported in Table 1. When combined with DenseNets, the best discovered tree cell (i.e. TreeCell-A) achieves a test error rate of 3.64% with only 3.2M parameters, which is comparable to the best result, i.e. 3.46% in the original DenseNet paper, given by a much

the growth rate, i.e. the number of filters of each 3×3 convolution layer. And we use the group convolution with $G = 4$ groups here. For DenseNet-BC, $L = 6 \times N + 4$, so we omit L in the following discussions for simplicity.

larger DenseNet-BC with 25.6M parameters. Furthermore, the test error rate drops to 3.35% as the number of parameters increases to 13.1M. We attribute the improved parameter efficiency and better test error rate results to the improved representation power from the increased path topology complexity introduced by the learned tree cells. When combined with PyramidNets, TreeCell-A reaches 3.14% test error with only 5.7M parameters while the best PyramidNet achieves 3.31% test error with 26.0M parameters, which also indicates significantly improved parameter efficiency by incorporating the learned tree cells for PyramidNets. Since the cells are learned using a DenseNet as the start point rather than a PyramidNet, it thereby justifies the transferability of the learned cells to other types of architectures.

We notice that there are some strong regularization techniques that have shown to effectively improve the performances on CIFAR-10, such as Shake-Shake (Gastaldi, 2017), DropPath (Zoph et al., 2017) and Cutout (DeVries & Taylor, 2017). In our experiments, when using DropPath that stochastically drops out each path (i.e. edge in the tree cell) and training the network for 600 epochs, as done in Zoph et al. (2017) and Liu et al. (2017), TreeCell-A reaches 2.99% test error with 5.7M parameters. Moreover, with Cutout, TreeCell-A further achieves 2.49% test error with 5.7M parameters and 2.30% test error with 14.3M parameters, outperforming all compared human-designed and automatically designed architectures on CIFAR-10 while having much fewer parameters (Table 1).

We would like to emphasize that these results are achieved with only 500 networks trained using about 200 GPU-hours while the compared architecture search methods utilize much more computational resources to achieve their best results, such as Zoph et al. (2017) that used 48,000 GPU-hours.

4.4. Results on ImageNet

Following Zoph et al. (2017) and Zhong et al. (2017), we further test the best cell structures learned on CIFAR-10, i.e. TreeCell-A and TreeCell-B, on ImageNet dataset. Due to resource and time constraints, we focus on the $Mobile$ setting in our experiments, where the input image size is 224×224 and we train relatively small models that require less than 600M multiply-add operations to perform inference on a single image. To do so, we combine the learned cell structures with CondenseNet (Huang et al., 2017a), a recently proposed efficient network architecture that is designed for the $Mobile$ setting.

The result is reported in Table 2. By embedding TreeCell-A into CondenseNet ($G_1 = 4, G_3 = 8$) where each block comprises a learned 1×1 group convolution layer with $G_1 = 4$ groups and a standard 3×3 group convolution layer with $G_3 = 8$ groups, we achieve 25.5% top-1 error and 8.0% top-5 error with 588M multiply-add operations, which significantly outperforms MobileNet and ShuffleNet, and is also better than CondenseNet ($G_1 = G_3 = 4$) with a similar number of multiply-add operations. Meanwhile, we find that TreeCell-B with CondenseNet ($G_1 = 4, G_3 = 8$) reaches a slightly better top-1 error result, i.e. 25.4%, than TreeCell-A.

Table 2. Top-1 (%) and Top-5 (%) classification error rate results on ImageNet in the $Mobile$ Setting ($\leq$ 600M multiply-add operations). "$\times+$" denotes the number of multiply-add operations.

Model	$\times+$	Top-1	Top-5
1.0 MobileNet-224 (Howard et al., 2017)	569M	29.4	10.5
ShuffleNet 2x (Zhang et al., 2017)	524M	29.1	10.2
CondenseNet ($G_1 = G_3 = 8$) (Huang et al., 2017a)	274M	29.0	10.0
CondenseNet ($G_1 = G_3 = 4$) (Huang et al., 2017a)	529M	26.2	8.3
NASNet-A ($N = 4$) (Zoph et al., 2017)	564M	26.0	8.4
NASNet-B ($N = 4$) (Zoph et al., 2017)	448M	27.2	8.7
NASNet-C ($N = 3$) (Zoph et al., 2017)	558M	27.5	9.0
TreeCell-A with CondenseNet ($G_1 = 4, G_3 = 8$)	588M	25.5	8.0
TreeCell-B with CondenseNet ($G_1 = 4, G_3 = 8$)	594M	25.4	8.1

When compared to NASNet-A, we also achieve slightly better results with similar multiply-add operations despite the fact that they used 48,000 GPU-hours to achieve these results while we only use 200 GPU-hours. By taking advantage of existing successful human-designed architectures, we can easily achieve similar (or even better) results with much fewer computational resources, compared to exploring the architecture space from scratch.

5. Conclusion

In this work, we presented path-level network transformation operations as an extension to current function-preserving network transformation operations to enable the architecture search methods to perform not only layer-level architecture modifications but also path-level topology modifications in a neural network. Based on the proposed path-level transformation operations, we further explored a tree-structured architecture space, a generalized version of current multi-branch architectures, that can embed plentiful paths within each CNN cell, with a bidirectional tree-structured RL meta-controller. The best designed cell structure by our method using only 200 GPU-hours has shown both improved parameter efficiency and better test accuracy on CIFAR-10, when combined with state-of-the-art human designed architectures including DenseNets and PyramidNets. And it has also demonstrated its transferability on ImageNet dataset in the $Mobile$ setting. For future work, we would like to combine the proposed method with network compression operations to explore the architecture space with the model size and the number of multiply-add operations taken into consideration and conduct experiments on other tasks such as object detection.

References

Ashok, A., Rhinehart, N., Beainy, F., and Kitani, K. M. N2n learning: Network to network compression via policy gradient reinforcement learning. *ICLR*, 2018.

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

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 2012.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. Efficient architecture search by network transformation. In *AAAI*, 2018.

Chen, T., Goodfellow, I., and Shlens, J. Net2net: Accelerating learning via knowledge transfer. *ICLR*, 2016.

Chollet, F. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, 2016.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

DeVries, T. and Taylor, G. W. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

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

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

Han, D., Kim, J., and Kim, J. Deep pyramidal residual networks. *CVPR*, 2017.

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

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

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 1997.

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

Huang, G., Liu, S., van der Maaten, L., and Weinberger, K. Q. Condensenet: An efficient densenet using learned group convolutions. *arXiv preprint arXiv:1711.09224*, 2017a.

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

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*, 2017.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. Hierarchical representations for efficient architecture search. *ICLR*, 2018.

Loshchilov, I. and Hutter, F. Sgdr: stochastic gradient descent with restarts. *arXiv preprint arXiv:1608.03983*, 2016.

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

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

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

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

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *CVPR*, 2015.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, 2017.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *ACL*, 2015.

Veit, A., Wilber, M. J., and Belongie, S. Residual networks behave like ensembles of relatively shallow networks. In *NIPS*, 2016.

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

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. Aggregated residual transformations for deep neural networks. In *CVPR*, 2017.

Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *arXiv preprint arXiv:1707.01083*, 2017.

Zhong, Z., Yan, J., and Liu, C.-L. Practical network blocks design with q-learning. *arXiv preprint arXiv:1708.05552*, 2017.

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

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.

A. Architecture Search Starting from Scratch

Beside utilizing state-of-the-art human-designed architectures, we also perform architecture search starting from scratch (i.e. a chain of identity mappings) to learn how much we can benefit from reusing existing well-designed architectures. The structure of the start point is provided in Table 3, where the identity mappings are later replaced by sampled cells to get new architectures and all other configurations keep the same as the ones used in Section 4.

The progress of the architecture search process is reported in Figure 8, where we can observe similar trends as the ones in Figure 7. Moreover, we find that the advantage of RL over RS is larger in this case (RL achieves 1.54% better validation accuracy than RS). After 300 epochs training on CIFAR-10, the best RL identified cell reaches 3.93% test error with 11.5M parameters, which is better than 4.44% given by the best random cell with 10.0M parameters, but is far worse than 3.14% given by TreeCell-A with 5.7M parameters.

Table 3. Start point network with identity mappings on CIFAR-10.

Model architecture	Feature map size	Output channels
3×3 Conv	32×32	48
[identity mapping] $\times 4$	32×32	48
1×1 Conv	32×32	96
3×3 average pooling, stride 2	16×16	96
[identity mapping] $\times 4$	16×16	96
1×1 Conv	16×16	192
3×3 average pooling, stride 2	8×8	192
[identity mapping] $\times 4$	8×8	192
8×8 global average pooling	1×1	192
10-dim fully-connected, softmax		

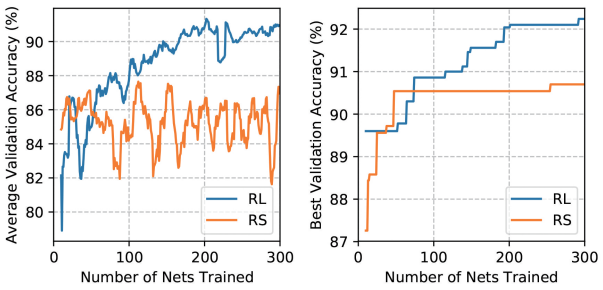


Figure 8. Progress of the architecture search process starting from scratch (a chain of identity maps) on CIFAR-10.

B. Details of Architecture Space

We find the following 2 tricks effective for reaching good performances with the tree-structured architecture space in our experiments.

B.1. Group Convolution

The base networks (i.e. DenseNets and PyramidNets) in our experiments use standard 3×3 group convolution instead of normal 3×3 convolution and the number of groups G is chosen from $\{1, 2, 4\}$ according to the sampled tree-structured cell. Specifically, if the merge scheme of the root node is *concatenation*, G is set to be 1; if the merge scheme is *add* and the number of branches is 2, G is set to be 2; if the merge scheme is *add* and the number of branches is 3, G is set to be 4. As such, we can make different sampled cells have a similar number of parameters as the normal 3×3 convolution layer.

B.2. Skip Node Connection and BN layer

Inspired by PyramidNets (Han et al., 2017) that add an additional batch normalization (BN) (Ioffe & Szegedy, 2015) layer at the end of each residual unit, which can enable the network to determine whether the corresponding residual unit is useful and has shown to improve the capacity of the network architecture. Analogously, in a tree-structured cell, we insert a skip connection for each child (denoted as $N_i^c(\cdot)$) of the root node, and merge the outputs of the child node and its corresponding skip connection via *add*. Additional, the output of the child node goes through a BN layer before it is merged. As such the output of the child node $N_i^c(\cdot)$ with input feature map $\boldsymbol{x}$ is given as:

$$O_i = add(\boldsymbol{x}, BN(N_i^c(\boldsymbol{x}))). \tag{4}$$

In this way, intuitively, each unit with tree-structured cell can at least go back to the original unit if the cell is not helpful here.

C. Detailed Structure of TreeCell-B

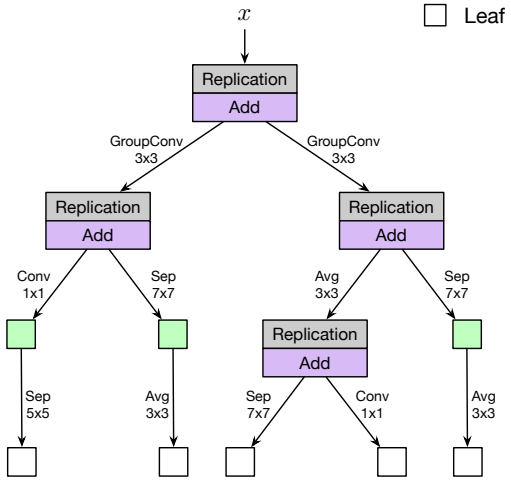


Figure 9. Detailed structure of TreeCell-B.

D. Meta-Controller Training Procedure

Algorithm 1 Path-Level Efficient Architecture Search

Input: base network $baseNet$, training set $trainSet$, validation set $valSet$, batch size B, maximum number of networks M

$trained = 0$ // Number of trained networks
$P_{nets} = []$ // Store results of trained networks
randomly initialize the meta-controller C
$G_c = []$ // Store gradients to be applied to C
while $trained < M$ **do**
  meta-controller C samples a tree-structured $cell$
  if $cell$ in P_{nets} **then**
    get the validation accuracy acc_v of $cell$ from P_{nets}
  else
    model = train(trans($baseNet$, $cell$), $trainSet$)
    acc_v = evel(model, $valSet$)
    add ($cell$, acc_v) to P_{nets}
    $trained = trained + 1$
  end if
  compute gradients according to ($cell$, acc_v) and add to G_c
  if $len(G_c) == B$ **then**
    update C according to G_c
    $G_c = []$
  end if
end while