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		

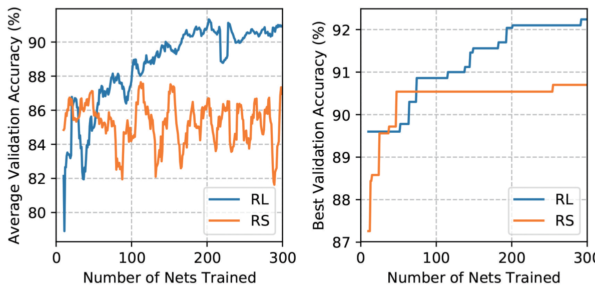

Figure 8. Progress of the architecture search process starting from scratch (a chain of identity maps) on CIFAR-10.

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

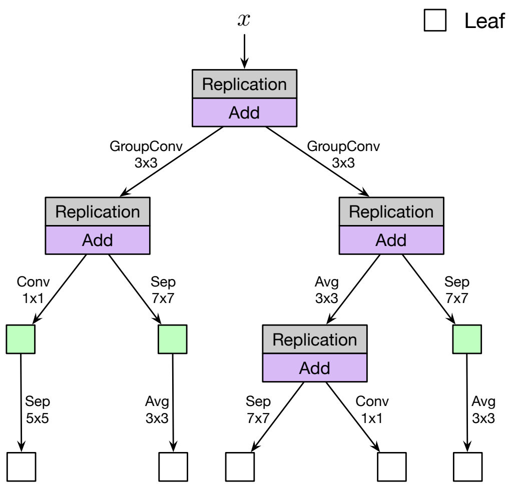

Figure 9. Detailed structure of TreeCell-B.

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 $M$
 1: $trained = 0$ // Number of trained networks
 2: $P_{nets} = []$ // Store results of trained networks
 3: randomly initialize the meta-controller $C$
 4: $G_c = []$ // Store gradients to be applied to $C$
 5: while $trained < M$ do
 6:   meta-controller $C$ samples a tree-structured $cell$
 7:   if $cell$ in $P_{nets}$ then
 8:     get the validation accuracy $acc_v$ of $cell$ from $P_{nets}$
 9:   else
10:     model = train(trans($baseNet$, $cell$), $trainSet$)
11:     $acc_v$ = evel(model, $valSet$)
12:     add ($cell$, $acc_v$) to $P_{nets}$
13:     $trained = trained + 1$
14:   end if
15:   compute gradients according to ($cell$, $acc_v$) and add to $G_c$
16:   if $len(G_c) == B$ then
17:     update $C$ according to $G_c$
18:     $G_c = []$
19:   end if
20: end while
```