Supplementary Material: Densely Connected Search Space for More Flexible Neural Architecture Search

Jiemin Fang[1†], Yuzhu Sun[1†], Qian Zhang[2], Yuan Li[2], Wenyu Liu[1], Xinggang Wang[1‡]
[1]School of EIC, Huazhong University of Science and Technology [2]Horizon Robotics
{jaminfong, yzsun, liuwy, xgwang}@hust.edu.cn
{qian01.zhang, yuan.li}@horizon.ai

A. Appendix

A.1. Implementation Details

Before the search process, we build a lookup table for every operation latency of the super network as described in Sec. 3.3. We set the input shape as $(3, 224, 224)$ with the batch size of 32 and measure each operation latency on one TITAN-XP GPU. All models and experiments are implemented using PyTorch [7].

For the search process, we randomly choose 100 classes from the original 1K-class ImageNet training set. We sample 20% data of each class from the above subset as the validation set. The original validation set of ImageNet is only used for evaluating our final searched architecture. The search process takes 150 epochs in total. We first train the operation weights for 50 epochs on the divided training set. For the last 100 epochs, the updating of architecture parameters (α, β) and operation weights (w) alternates in each epoch. We use the standard GoogleNet [9] data augmentation for the training data preprocessing. We set the batch size to 352 on 4 Tesla V100 GPUs. The SGD optimizer is used with 0.9 momentum and 4×10^{-5} weight decay to update the operation weights. The learning rate decays from 0.2 to 1×10^{-4} with the cosine annealing schedule [6]. We use the Adam optimizer [2] with 10^{-3} weight decay, $\beta = (0.5, 0.999)$ and a fixed learning rate of 3×10^{-4} to update the architecture parameters.

For retraining the final derived architecture, we use the same data augmentation strategy as the search process on the whole ImageNet dataset. We train the model for 240 epochs with a batch size of 1024 on 8 TITAN-XP GPUs. The optimizer is SGD with 0.9 momentum and 4×10^{-5} weight decay. The learning rate decays from 0.5 to 1×10^{-4} with the cosine annealing schedule.

†The work is performed during the internship at Horizon Robotics.
‡Corresponding author.

A.2. Viterbi Algorithm for Block Deriving

The Viterbi Algorithm [4] is widely used in dynamic programming which targets at finding the most likely path between hidden states. In DenseNAS, only a part of routing blocks in the super network are retained to construct the final architecture. As described in Sec. 3.4, we implement the Viterbi algorithm to derive the final sequence of blocks. We treat the routing block in the super network as each hidden state in the Viterbi algorithm. The path probability p_{ij} serves as the transition probability from routing block B_i to B_j. The total algorithm is described in Algo. 1. The derived block sequence holds the maximum transition probability.

Algorithm 1: The Viterbi algorithm used for deriving the block sequence of the final architecture.

```
Input: input block $B_0$, routing blocks $\{B_1, \dots, B_N\}$, ending block $B_{N+1}$,
       connection numbers $\{M_1, \dots, M_{N+1}\}$, path probabilities
       $\{p_{ji} | i = 1, \dots, N+1, j = i-1, \dots, i-M_i\}$
Output: the derived block sequence $X$
1  $P[0] \leftarrow 1$ ; // record the probabilities
2  $S[0] \leftarrow 0$ ; // record the block indices
3  for $i \leftarrow 1, \dots, N+1$ do
4      $P[i] \leftarrow \max_{i-1 \le j \le i-M_i} (P[i-1] \cdot p_{ji})$;
5      $S[i] \leftarrow \arg\max_{i-1 \le j \le i-M_i} (P[i-1] \cdot p_{ji})$;
6  $X[0] \leftarrow B_{N+1}$;
7  $idx \leftarrow N+1$;
8  $count \leftarrow 1$;
9  do
10     $X[count] \leftarrow B_{S[idx]}$;
11     $idx \leftarrow S[idx]$;
12     $count \leftarrow count + 1$;
13 while $idx \neq 0$;
14 $reversX$;
```

Table 1: Architectures searched by DenseNAS in the ResNet-based search space.

Stage	Output Size	DenseNAS-R1	DenseNAS-R2	DenseNAS-R3
1	112×112	3×3, 32, stride 2		
2	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 1$	$\begin{bmatrix} 3 \times 3, 48 \\ 3 \times 3, 48 \end{bmatrix} \times 1$	$\begin{bmatrix} 1 \times 1, 48 \\ 3 \times 3, 48 \\ 1 \times 1, 192 \end{bmatrix} \times 1$
3	28×28	$\begin{bmatrix} 3 \times 3, 72 \\ 3 \times 3, 72 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 72 \\ 3 \times 3, 72 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 72 \\ 3 \times 3, 72 \\ 1 \times 1, 288 \end{bmatrix} \times 4$
4	14×14	$\begin{bmatrix} 3 \times 3, 176 \\ 3 \times 3, 176 \end{bmatrix} \times 6$ $\begin{bmatrix} 3 \times 3, 192 \\ 3 \times 3, 192 \end{bmatrix} \times 3$	$\begin{bmatrix} 3 \times 3, 176 \\ 3 \times 3, 176 \end{bmatrix} \times 16$ $\begin{bmatrix} 3 \times 3, 208 \\ 3 \times 3, 208 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 176 \\ 3 \times 3, 176 \\ 1 \times 1, 704 \end{bmatrix} \times 16$ $\begin{bmatrix} 1 \times 1, 208 \\ 3 \times 3, 208 \\ 1 \times 1, 832 \end{bmatrix} \times 4$
5	7×7	$\begin{bmatrix} 3 \times 3, 288 \\ 3 \times 3, 288 \end{bmatrix} \times 1$ $\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 1$	$\begin{bmatrix} 3 \times 3, 288 \\ 3 \times 3, 288 \end{bmatrix} \times 2$ $\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 1$	$\begin{bmatrix} 1 \times 1, 288 \\ 3 \times 3, 288 \\ 1 \times 1, 1152 \end{bmatrix} \times 2$ $\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 1$
6	1×1	average pooling, 1000-d fc, softmax		

A.3. Dropping-path Search Strategy

The super network includes all the possible architectures defined in the search space. To decrease the memory consumption and accelerate the search process, we adopt the dropping-path search strategy [1, 3] (which is mentioned in Sec. 3.4). When training the weights of operations, we sample one path of the candidate operations according to the architecture weight distribution $\{w_o^\ell | o \in \mathcal{O}\}$ in every basic layer. The dropping-path strategy not only accelerates the search but also weakens the coupling effect between operation weights shared by different sub-architectures in the search space. To update the architecture parameters, we sample two operations in each basic layer according to the architecture weight distribution. To keep the architecture weights of the unsampled operations unchanged, we compute a re-balancing bias to adjust the sampled and newly updated parameters.

$$\mathtt{bias}_s = \ln \frac{\sum_{o \in \mathcal{O}_s} \exp(\alpha_o^\ell)}{\sum_{o \in \mathcal{O}_s} \exp({\alpha'}_o^\ell)}, \tag{1}$$

where $\mathcal{O}_s$ refers to the set of sampled operations, α_o^ℓ denotes the original value of the sampled architecture parameter in layer ℓ and ${\alpha'}_o^\ell$ denotes the updated value of the architecture parameter. The computed bias is finally added to the updated architecture parameters.

A.4. Implementation Details of ResNet Search

We design the ResNet-based search space as follows. As enlarging the kernel size of the ResNet block causes a huge computation cost increase, the candidate operations in the basic layer only include the basic block [5] and the skip connection. That means we aim at width and depth search for ResNet networks. During the search, the batch size is set as 512 on 4 Tesla V100 GPUs. The search process takes 70 epochs in total and we start to update the architecture parameters from epoch 10. We set all the other search settings and hyper-parameters the same as that in the MobileNetV2 [8] search. For the architecture retraining, the same training settings and hyper-parameters are used as that for architectures searched in the MobileNetV2-based search space. The architectures searched by DenseNAS are shown in Tab. 1.

Table 2: Comparisons of different cost estimation methods.

Estimation Method	FLOPs	GPU Latency	Top-1 Acc(%)
local cost estimation	396M	27.5ms	74.8
chained cost estimation	361M	17.9ms	75.3

A.5. Experimental Comparison of Cost Estimation Method

We study the design of the model cost estimation algorithm in Sec. 4.5. $1,500$ models are derived based on the randomly generated architecture parameters. Cost values predicted by our proposed chained cost estimation algorithm demonstrate a stronger correlation with the real values and more accurate prediction results than the compared local cost estimation strategy. We further perform the same search process as DenseNAS on the MobileNetV2 [8]-based search space with the local estimation strategy and show the searched results in Tab. 2. DenseNAS with the chained cost estimation algorithm shows a higher accuracy with lower latency and fewer FLOPs. It proves the effectiveness of the chained cost estimation algorithm on achieving a good trade-off between accuracy and model cost.

References

[1] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Understanding and simplifying one-shot architecture search. In *ICML*, 2018. 2

[2] Yoshua Bengio and Yann LeCun, editors. *ICLR*, 2015. 1

[3] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019. 2

[4] G David Forney. The viterbi algorithm. *Proceedings of the IEEE*, 1973. 1

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 2

[6] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. In *ICLR*, 2017. 1

[7] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 1

[8] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018. 2, 3

[9] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015. 1