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. The search process takes 150 epochs in total. We first train the operation weights for 50 epochs on the divided training set. The learning rate decays from 0.5 to 10^{-4} with the cosine annealing schedule [6].

The optimizer is SGD with weight decay. The learning rate decays from 0.5 to 10^{-5} weight decay 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 \times 10^{-5} weight decay. The learning rate decays from 0.5 to 10^{-4} with the cosine annealing schedule.

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.

\[
\begin{align*}
\text{Input:} & \quad \text{input block } B_0, \text{ routing blocks } \{B_1, \ldots, B_N\}, \text{ ending block } B_{N+1}, \\
& \quad \text{connection numbers } \{M_1, \ldots, M_{N+1}\}, \text{ path probabilities } \\
& \quad \{p_{ij} | i = 1, \ldots, N+1, j = i-1, \ldots, i-M_i\} \\
\text{Output:} & \quad \text{the derived block sequence } X
\end{align*}
\]

\[
\begin{align*}
1 & \text{P}[0] \leftarrow 1; // \text{ record the probabilities} \\
2 & \text{S}[0] \leftarrow 0; // \text{ record the block indices} \\
3 & \text{for } i \leftarrow 1, \ldots, N + 1 \text{ do} \\
4 & \quad \text{P}[i] \leftarrow \max_{1 \leq j \leq i-M_i} (P[i-1] \cdot p_{ij}); \\
5 & \quad \text{S}[i] \leftarrow \arg\max_{1 \leq j \leq i-M_i} (P[i-1] \cdot p_{ij}); \\
6 & \quad X[0] \leftarrow B_{N+1}; \\
7 & \quad i_{\text{dx}} \leftarrow N + 1; \\
8 & \quad \text{count} \leftarrow 1; \\
9 & \quad \text{while } i_{\text{dx}} \neq 0; \\
10 & \quad X[\text{count}] \leftarrow B_{S[i_{\text{dx}}]}; \\
11 & \quad i_{\text{dx}} \leftarrow S[i_{\text{dx}}]; \\
12 & \quad \text{count} \leftarrow \text{count} + 1; \\
13 & \quad \text{end while} \\
14 & \text{reversX}; \\
\end{align*}
\]
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 \( \{ w_o^\ell | o \in O \} \) 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.

\[
\text{bias}_s = \ln \frac{\sum_{\ell \in \mathcal{O}_s} \exp(\alpha'_{o_\ell})}{\sum_{\ell \in \mathcal{O}_s} \exp(\alpha''_{o_\ell})},
\]

(1)

where \( \mathcal{O}_s \) refers to the set of sampled operations, \( \alpha'_{o_\ell} \) denotes the original value of the sampled architecture parameter in layer \( \ell \) 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 with the same form (instead of directly enlarging the kernel size of the ResNet block). 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 1: Architectures searched by DenseNAS in the ResNet-based search space.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Output Size</th>
<th>DenseNAS-R1</th>
<th>DenseNAS-R2</th>
<th>DenseNAS-R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>112 × 112</td>
<td>3 × 3, 32, stride 2</td>
<td>1 × 1, 48</td>
<td>1 × 1, 192</td>
</tr>
<tr>
<td>2</td>
<td>56 × 56</td>
<td>3 × 3, 64 × 1</td>
<td>3 × 3, 48 × 1</td>
<td>1 × 1, 192</td>
</tr>
<tr>
<td>3</td>
<td>28 × 28</td>
<td>3 × 3, 72 × 2</td>
<td>3 × 3, 72 × 4</td>
<td>1 × 1, 288</td>
</tr>
<tr>
<td>4</td>
<td>14 × 14</td>
<td>3 × 3, 176 × 6</td>
<td>3 × 3, 176 × 16</td>
<td>1 × 1, 208</td>
</tr>
<tr>
<td>5</td>
<td>7 × 7</td>
<td>3 × 3, 288 × 1</td>
<td>3 × 3, 288 × 2</td>
<td>1 × 1, 2048</td>
</tr>
<tr>
<td>6</td>
<td>1 × 1</td>
<td>average pooling, 1000-d fc, softmax</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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