

Densely Connected Search Space for More Flexible Neural Architecture Search

Jiemin Fang^{1†}, Yuzhu Sun^{1†}, Qian Zhang², Yuan Li², Wenyu Liu¹, Xinggang Wang¹

¹School of EIC, Huazhong University of Science and Technology ²Horizon Robotics
 {jaminfong, yzsun, liuwu, xgwang}@hust.edu.cn
 {qian01.zhang, yuan.li}@horizon.ai

Abstract

Neural architecture search (NAS) has dramatically advanced the development of neural network design. We revisit the search space design in most previous NAS methods and find the number of blocks and the widths of blocks are set manually. However, block counts and block widths determine the network scale (depth and width) and make a great influence on both the accuracy and the model cost (FLOPs/latency). In this paper, we propose to search block counts and block widths by designing a densely connected search space, i.e., DenseNAS. The new search space is represented as a dense super network, which is built upon our designed routing blocks. In the super network, routing blocks are densely connected and we search for the best path between them to derive the final architecture. We further propose a chained cost estimation algorithm to approximate the model cost during the search. Both the accuracy and model cost are optimized in DenseNAS. For experiments on the MobileNetV2-based search space, DenseNAS achieves 75.3% top-1 accuracy on ImageNet with only 361MB FLOPs and 17.9ms latency on a single TITAN-XP. The larger model searched by DenseNAS achieves 76.1% accuracy with only 479M FLOPs. DenseNAS further promotes the ImageNet classification accuracies of ResNet-18, -34 and -50-B by 1.5%, 0.5% and 0.3% with 200M, 600M and 680M FLOPs reduction respectively. *

1. Introduction

In recent years, neural architecture search (NAS) [50, 51, 35, 37] has demonstrated great successes in designing neural architectures automatically and achieved remarkable performance gains in various tasks such as image classification [51, 35], semantic segmentation [6, 26] and object detection [15, 45]. NAS has been a critically important topic for architecture designing.

*The related code is available at <https://github.com/JaminFong/DenseNAS>

[†]The work is performed during an internship at Horizon Robotics.

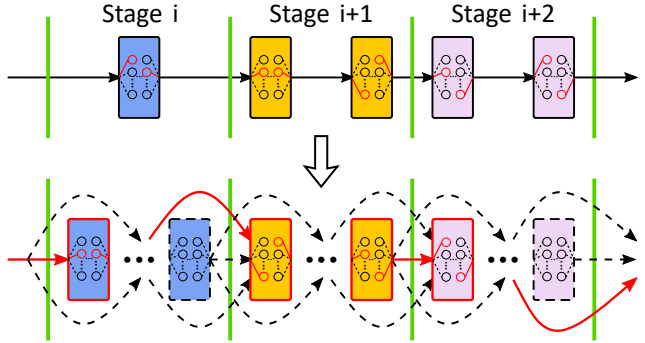


Figure 1: Search space comparison between conventional methods and DenseNAS. *Upper*: Conventional search spaces manually set a fixed number of blocks in each stage. The block widths are set manually as well. *Bottom*: The search space in DenseNAS allows more blocks with various widths in each stage. Each block is densely connected to its subsequent ones. We search for the best path (the red line) to derive the final architecture, in which the number of blocks in each stage and the widths of blocks are allocated automatically.

In NAS research, the search space plays a crucial role that constrains the architectures in a prior-based set. The performance of architectures produced by NAS methods is strongly associated with the search space definition. A more flexible search space has the potential to bring in architectures with more novel structures and promoted performance. We revisit and analyze the search space design in most previous works [51, 41, 4, 43]. For a clear illustration, we review the following definitions. *Block* denotes a set of layers/operations in the network which output feature maps with the same spatial resolution and the same width (number of channels). *Stage* denotes a set of sequential blocks whose outputs are under the same spatial resolution settings. Different blocks in the same stage are allowed to have various widths. Many recent works [4, 43, 8] stack the inverted residual convolution modules (MBConv) defined in MobileNetV2 [38] to construct the search space. They search for different kernel sizes and expansion ratios in each

MBConv. The depth is searched in terms of layer numbers in each block. The searched networks with MBConvs show high performance with low latency or few FLOPs.

In this paper, we aim to perform NAS in a more flexible search space. Our motivation and core idea are illustrated in Fig. 1. As the upper part of Fig. 1 shows, the number of blocks in each stage and the width of each block are set manually and fixed during the search process. It means that the depth search is constrained within the block and the width search cannot be performed. It is worth noting that the scale (depth and width) setting is closely related to the performance of a network, which has been demonstrated in many previous theoretical studies [36, 32] and empirical results [16, 42]. Inappropriate width or depth choices usually cause drastic accuracy degradation, significant computation cost, or unsatisfactory model latency. Moreover, we find that recent works [38, 4, 43, 8] manually tune width settings to obtain better performance, which indicates the design of network width demands much prior-based knowledge and trial-and-error.

We propose a densely connected search space to tackle the above obstacles and name our method as *DenseNAS*. We show our novel designed search space schematically in the bottom part of Fig. 1. Different from the search space design principles in the previous works [4, 43], we allow more blocks with various widths in one stage. Specifically, we design the *routing blocks* to construct the densely connected super network which is the representation of the search space. From the beginning to the end of the search space, the width of the routing block increases gradually to cover more width options. Every routing block is connected to several subsequent ones. This formulation brings in various paths in the search space and we search for the best path to derive the final architecture. As a consequence, the block widths and counts in each stage are allocated automatically. Our method extends the depth search into a more flexible space. Not only the number of layers within one block but also the number of blocks within one stage can be searched. The block width search is enabled as well. Moreover, the positions to conduct spatial down-sampling operations are determined along with the block counts search.

We integrate our search space into the differentiable NAS framework by relaxing the search space. We assign a probability parameter to each output path of the routing block. During the search process, the distribution of probabilities is optimized. The final block connection paths in the super network are derived based on the probability distribution. To optimize the cost (FLOPs/latency) of the network, we design a *chained estimation algorithm* targeted at approximating the cost of the model during the search.

Our contributions can be summarized as follows.

- We propose a densely connected search space that enables network/block widths search and block counts

search. It provides more room for searching better networks and further reduces expert designing efforts.

- We propose a chained cost estimation algorithm to precisely approximate the computation cost of the model during search, which makes the DenseNAS networks achieve high performance with low computation cost.
- In experiments, we demonstrate the effectiveness of our method on the MobileNetV2 [38]-based search space and achieve SOTA performance. Our searched network achieves 75.3% accuracy on ImageNet [10] with only 361MB FLOPs and 17.9ms latency on a single TITAN-XP.
- DenseNAS can further promote the ImageNet classification accuracies of ResNet-18, -34 and -50-B [17] by 1.5%, 0.5% and 0.3% with 200M, 600M, 680M FLOPs and 1.5ms, 2.4ms, 6.1ms latency reduction respectively.

2. Related Work

Search Space Design NASNet [51] is the first work to propose a cell-based search space, where the cell is represented as a directed acyclic graph with several nodes inside. NASNet searches for the operation types and the topological connections in the cell and repeat the searched cell to form the whole network architecture. The depth of the architecture (*i.e.*, the number of repetitions of the cell), the widths and the occurrences of down-sampling operations are all manually set. Afterwards, many works [27, 35, 37, 29] adopt a similar cell-based search space. However, architectures generated by cell-based search spaces are not friendly in terms of latency or FLOPs. Then MnasNet [41] stacks MBConvs defined in MobileNetV2 [38] to construct a search space for searching efficient architectures. ProxylessNAS [4], FBNet [43] and ChamNet [9] simplify the search space by searching for the expansion ratios and kernel sizes of MBConv layers.

Some works study more about the search space. Liu *et al.* [28] proposes a hierarchical search space that allows flexible network topologies (directed acyclic graphs) at each level of the hierarchies. Auto-DeepLab [26] creatively designs a two-level hierarchical search space for semantic segmentation networks. CAS [48] customizes the search space design for real-time segmentation networks. RandWire [44] explores randomly wired architectures by designing network generators that produce new families of models for searching. Our proposed method designs a densely connected search space beyond conventional search constraints to generate the architecture with a better trade-off between accuracy and model cost.

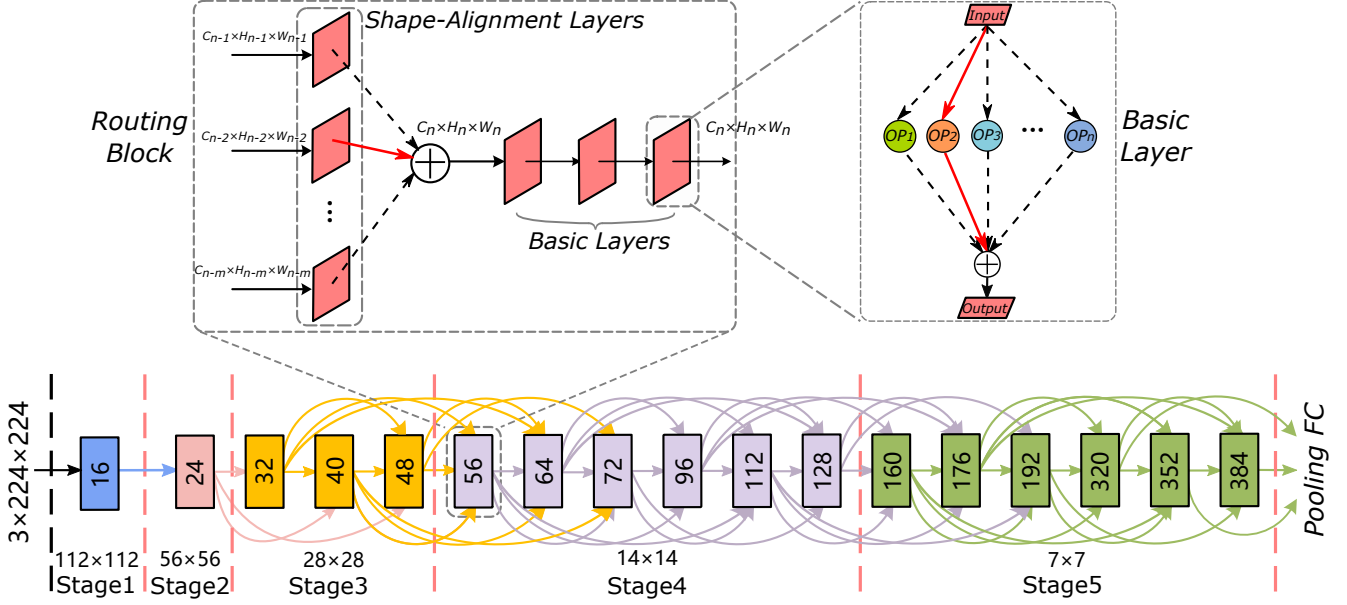


Figure 2: We define our search space on three levels. *Upper right:* A *basic layer* that contains a set of candidate operations. *Upper left:* The proposed *routing block* which contains *shape-alignment layers*, an element-wise sum operation and some basic layers. It takes multiple input tensors and outputs one tensor. *Bottom:* The proposed *dense super network* which is constructed with densely connected routing blocks. Routing blocks in the same stage hold the same color. Architectures are searched within various path options in the super network.

NAS Method Some early works [50, 51, 49] propose to search architectures based on reinforcement learning (RL) methods. Then evolutionary algorithm (EA) based methods [12, 28, 37] achieve great performance. However, RL and EA based methods bear huge computation cost. As a result, ENAS [35] proposes to use weight sharing for reducing the search cost.

Recently, the emergence of differentiable NAS methods [29, 4, 43] and one-shot methods [3, 1] greatly reduces the search cost and achieves superior results. DARTS [29] is the first work to utilize the gradient-based method to search neural architectures. They relax the architecture representation as a super network by assigning continuous weights to the candidate operations. They first search on a small dataset, *e.g.*, CIFAR-10 [22], and then apply the architecture to a large dataset, *e.g.*, ImageNet [11], with some manual adjustments. ProxylessNAS [4] reduces the memory consumption by adopting a dropping path strategy and conducts search directly on the large scale dataset, *i.e.*, ImageNet. FBNet [43] searches on the subset of ImageNet and uses the Gumbel Softmax function [21, 33] to better optimize the distribution of architecture probabilities. TAS [13] utilizes a differentiable NAS scheme to search and prune the width and depth of the network and uses knowledge distillation (KD) [18] to promote the performance of the pruned network. It is challenging for differentiable/one-shot NAS methods to search for more flexible architectures as they

need to integrate all sub-architectures into the super network. The proposed DenseNAS tends to solve this problem by integrating a densely connected search space into the differentiable paradigm and explores more flexible search schemes in the network.

3. Method

In this section, we first introduce how to design the search space targeted at a more flexible search. A *routing block* is proposed to construct the densely connected super network. Secondly, we describe the method of relaxing the search space into a continuous representation. Then, we propose a *chained cost estimation* algorithm to approximate the model cost during the search. Finally, we describe the whole search procedure.

3.1. Densely Connected Search Space

As shown in Fig. 2, we define our search space using the following three terms, *i.e.*, (*basic layer*, *routing block* and *dense super Network*). Firstly, a *basic layer* is defined as a set of all the candidate operations. Then we propose a novel *routing block* which can aggregate tensors from different routing blocks and transmit tensors to multiple other routing blocks. Finally, the search space is constructed as a *dense super network* with many routing blocks where there are various paths to transmit tensors.

3.1.1 Basic Layer

We define the *basic layer* to be the elementary structure in our search space. One basic layer represents a set of candidate operations which include MBConvs and the skip connection. MBConvs are with kernel sizes of $\{3, 5, 7\}$ and expansion ratios of $\{3, 6\}$. The skip connection is for the depth search. If the skip connection is chosen, the corresponding layer is removed from the resulting architecture.

3.1.2 Routing Block

For the purpose of establishing various paths in the super network, we propose the *routing block* with the ability of aggregating tensors from preceding routing blocks and transmit tensors to subsequent ones. We divide the routing block into two parts, *shape-alignment layers* and *basic layers*.

Shape-alignment layers exist in the form of several parallel branches, while every branch is a set of candidate operations. They take input tensors with different shapes (including widths and spatial resolutions) which come from multiple preceding routing blocks and transform them into tensors with the same shape. As shape-alignment layers are required for all routing blocks, we exclude the skip connection in candidate operations of them. Then tensors processed by shape-alignment layers are aggregated and sent to several basic layers. The subsequent basic layers are used for feature extraction whose depth can also be searched.

3.1.3 Dense Super Network

Many previous works [41, 4, 43] manually set a fixed number of blocks, and retain all the blocks for the final architecture. Benefiting from the aforementioned structures of routing blocks, we introduce more routing blocks with various widths to construct the *dense super network* which is the representation of the search space. The final searched architecture is allowed to select a subset of the routing blocks and discard the others, giving the search algorithm more room.

We define the super network as \mathcal{N}_{sup} and assume it to consist of N routing blocks, $\mathcal{N}_{sup} = \{B_1, B_2, \dots, B_N\}$. The network structure is shown in Fig. 2. We partition the entire network into several stages. As Sec. 1 defines, each stage contains routing blocks with various widths and the same spatial resolution. From the beginning to the end of the super network, the widths of routing blocks grow gradually. In the early stage of the network, we set a small growing stride for the width because large width settings in the early network stage will cause huge computational cost. The growing stride becomes larger in the later stages. This design principle of the super network allows more possibilities of block counts and block widths.

We assume that each routing block in the super network connects to M subsequent ones. We define the connection

between the routing block B_i and its subsequent routing block B_j ($j > i$) as C_{ij} . The spatial resolutions of B_i and B_j are $H_i \times W_i$ and $H_j \times W_j$ respectively (normally $H_i = W_i$ and $H_j = W_j$). We set some constraints on the connections to avoid the stride of the spatial down-sampling exceeding 2. Specifically, C_{ij} only exists when $j - i \leq M$ and $H_i/H_j \leq 2$. Following the above paradigms, the search space is constructed as a dense super network based on the connected routing blocks.

3.2. Relaxation of Search Space

We integrate our search space by relaxing the architectures into continuous representations. The relaxation is implemented on both the basic layer and the routing block. We can search for architectures via back-propagation in the relaxed search space.

3.2.1 Relaxation in the Basic Layer

Let \mathcal{O} be the set of candidate operations described in Sec. 3.1.1. We assign an architecture parameter α_o^ℓ to the candidate operation $o \in \mathcal{O}$ in basic layer ℓ . We relax the basic layer by defining it as a weighted sum of outputs from all candidate operations. The architecture weight of the operation is computed as a *softmax* of architecture parameters over all operations in the basic layer:

$$w_o^\ell = \frac{\exp(\alpha_o^\ell)}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^\ell)}. \quad (1)$$

The output of basic layer ℓ can be expressed as

$$x_{\ell+1} = \sum_{o \in \mathcal{O}} w_o^\ell \cdot o(x_\ell), \quad (2)$$

where x_ℓ denotes the input tensor of basic layer ℓ .

3.2.2 Relaxation in the Routing Block

We assume that the routing block B_i outputs the tensor b_i and connects to m subsequent blocks. To relax the block connections as a continuous representation, we assign each output path of the block an architecture parameter. Namely the path from B_i to B_j has a parameter β_{ij} . Similar to how we compute the architecture weight of each operation above, we compute the probability of each path using a *softmax* function over all paths between the two routing blocks:

$$p_{ij} = \frac{\exp(\beta_{ij})}{\sum_{k=1}^m \exp(\beta_{ik})}. \quad (3)$$

For routing block B_i , we assume it takes input tensors from its m' preceding routing blocks ($B_{i-m'}, B_{i-m'+1}, B_{i-m'+2} \dots B_{i-1}$). As shown in Fig. 2, the input tensors from these routing blocks differ in terms of width and spatial resolution. Each input tensor is transformed to a same

size by the corresponding branch of shape-alignment layers in B_i . Let H_{ik} denotes the k th transformation branch in B_i which is applied to the input tensor from B_{i-k} , where $k = 1 \dots m'$. Then the input tensors processed by shape-alignment layers are aggregated by a weighted-sum using the path probabilities,

$$x_i = \sum_{k=1}^{m'} p_{i-k,i} \cdot H_{ik}(x_{i-k}). \quad (4)$$

It is worth noting that the path probabilities are normalized on the output dimension but applied on the input dimension (more specifically on the branches of shape-alignment layers). One of the shape-alignment layers is essentially a weighted-sum mixture of the candidate operations. The layer-level parameters α control which operation to be selected, while the outer block-level parameters β determine how blocks connect.

3.3. Chained Cost Estimation Algorithm

We propose to optimize both the accuracy and the cost (latency/FLOPs) of the model. To this end, the model cost needs to be estimated during the search. In conventional cascaded search spaces, the total cost of the whole network can be computed as a sum of all the blocks. Instead, the global effects of connections on the predicted cost need to be taken into consideration in our densely connected search space. We propose a *chained cost estimation* algorithm to better approximate the model cost.

We create a lookup table which records the cost of each operation in the search space. The cost of every operation is measured separately. During the search, the cost of one basic layer is estimated as follows,

$$\text{cost}^\ell = \sum_{o \in \mathcal{O}} w_o^\ell \cdot \text{cost}_o^\ell, \quad (5)$$

where cost_o^ℓ refers to the pre-measured cost of operation $o \in \mathcal{O}$ in layer ℓ . We assume there are N routing blocks in total (B_1, \dots, B_N). To estimate the total cost of the whole network in the densely connected search space, we define the chained cost estimation algorithm as follows.

$$\begin{aligned} \tilde{\text{cost}}^N &= \text{cost}_b^N \\ \tilde{\text{cost}}^i &= \text{cost}_b^i + \sum_{j=i+1}^{i+m} p_{ij} \cdot (\text{cost}_{align}^{ij} + \text{cost}_b^j), \end{aligned} \quad (6)$$

where cost_b^i denotes the total cost of all the basic layers of B_i which can be computed as a sum $\text{cost}_b^i = \sum_\ell \text{cost}_b^{\ell,i}$, m denotes the number of subsequent routing blocks to which B_i connects, p_{ij} denotes the path probability between B_i and B_j , and cost_{align}^{ij} denotes the cost of the

shape-alignment layer in block B_j which processes the data from block B_i .

The cost of the whole architecture can thus be obtained by computing $\tilde{\text{cost}}^1$ with a recursion mechanism,

$$\text{cost} = \tilde{\text{cost}}^1. \quad (7)$$

We design a loss function with the cost-based regularization to achieve the multi-objective optimization:

$$\mathcal{L}(w, \alpha, \beta) = \mathcal{L}_{CE} + \lambda \log_\tau \text{cost}, \quad (8)$$

where λ and τ are the hyper-parameters to control the magnitude of the model cost term.

3.4. Search Procedure

Benefiting from the continuously relaxed representation of the search space, we can search for the architecture by updating the architecture parameters (introduced in Sec. 3.2) using stochastic gradient descent. We find that at the beginning of the search process, all the weights of the operations are under-trained. The operations or architectures which converge faster are more likely to be strengthened, which leads to shallow architectures. To tackle this, we split our search procedure into two stages. In the first stage, we only optimize the weights for enough epochs to get operations sufficiently trained until the accuracy of the model is not too low. In the second stage, we activate the architecture optimization. We alternatively optimize the operation weights by descending $\nabla_w \mathcal{L}_{train}(w, \alpha, \beta)$ on the training set, and optimize the architecture parameters by descending $\nabla_{\alpha, \beta} \mathcal{L}_{val}(w, \alpha, \beta)$ on the validation set. Moreover, a dropping-path training strategy [1, 4] is adopted to decrease memory consumption and decouple different architectures in the super network.

When the search procedure terminates, we derive the final architecture based on the architecture parameters α, β . At the layer level, we select the candidate operation with the maximum architecture weight, i.e., $\arg \max_{o \in \mathcal{O}} \alpha_o^\ell$. At the network level, we use the Viterbi algorithm [14] to derive the paths connecting the blocks with the highest total transition probability based on the output path probabilities. Every block in the final architecture only connects to the next one.

4. Experiments

In this section, we first describe the implementation details. Secondly, we show the performance with MobileNetV2-based search space on the ImageNet [10] classification task. Then we apply the architectures searched on ImageNet to the COCO [24] object detection task. We further extend our DenseNAS to the ResNet [17]-based search space. Finally, we conduct some ablation studies and analysis.

Table 1: Our results on the ImageNet classification with the MobileNetV2-based search space compared with other methods. Our models achieve higher accuracies with lower latencies. For the GPU latency, we measure all the models with the same setup (on one TITAN-XP with batch size of 32).

Model	FLOPs	GPU Latency	Top-1 Acc(%)	Search Time (GPU hours)
1.4-MobileNetV2 [38]	585M	28.0ms	74.7	-
NASNet-A [51]	564M	-	74.0	48K
AmoebaNet-A [37]	555M	-	74.5	76K
DARTS [29]	574M	36.0ms	73.3	96
DenseNAS-Large	479M	28.9ms	76.1	64
1.0-MobileNetV1 [20]	575M	16.8ms	70.6	-
1.0-MobileNetV2 [38]	300M	19.5ms	72.0	-
DenseNAS-A	251M	13.6ms	73.1	64
MnasNet [41]	317M	19.7ms	74.0	91K
FBNet-B [43]	295M	18.9ms	74.1	216
Proxyless(mobile) [4]	320M	21.3ms	74.6	200
DenseNAS-B	314M	15.4ms	74.6	64
MnasNet-92 [41]	388M	-	74.8	91K
FBNet-C [43]	375M	22.1ms	74.9	216
Proxyless(GPU) [4]	465M	22.1ms	75.1	200
DenseNAS-C	361M	17.9ms	75.3	64
Random Search	360M	26.9ms	74.3	64

4.1. 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 [34].

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 [40] 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 [31]. 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.

Table 2: Object detection results on COCO. The FLOPs are calculated with 1088×800 input.

Method		Params	FLOPs	mAP(%)
MobileNetV2 [38]	RetinaNet	11.49M	133.05B	32.8
DenseNAS-B		12.69M	133.09B	34.3
DetNAS [7]		13.41M	133.26B	33.3
FBNet-C [43]		12.65M	134.17B	34.9
Proxyless(GPU) [4]		14.62M	135.81B	35
DenseNAS-C	SSDLite	13.24M	133.91B	35.1
MobileNetV2 [38]		4.3M	0.8B	22.1
Mnasnet-92 [41]		5.3M	1.0B	22.9
FBNet-C [43]		6.27M	1.06B	22.9
Proxyless(GPU) [4]		7.91M	1.24B	22.8
DenseNAS-C		6.87M	1.05B	23.1

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.

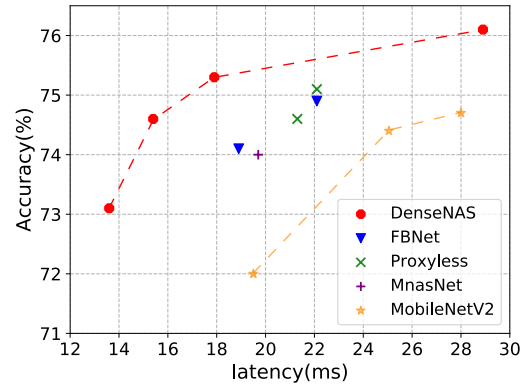


Figure 3: The comparison of model performance on ImageNet under the MobileNetV2-based search spaces.

4.2. Performance on MobileNetV2-based Search Space

We implement DenseNAS on the MobileNetV2 [38]-based search space, set the GPU latency as our secondary optimization objective, and search models with different sizes under multiple latency optimization magnitudes (defined in Eq. 8). The ImageNet results are shown in Tab. 1. We divide Tab. 1 into several parts and compare DenseNAS models with both manually designed models [20, 38] and models searched by other NAS methods. DenseNAS achieves higher accuracies with both fewer FLOPs and lower latencies. In the compared NAS methods, the block counts and block widths in the search space are set and adjusted manually. DenseNAS allocates block counts and block widths automatically. We further visualize the results in Fig. 3, which clearly demonstrates that DenseNAS achieves a better trade-off between model accuracy and latency. The searched architectures are shown in Fig. 4.

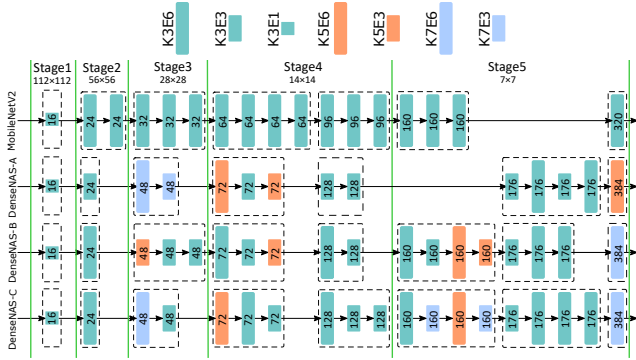


Figure 4: Visualization of the searched architectures. We use rectangles with different colors and widths to denote the layer operations. $KxEy$ denotes the MBConv with kernel size $x \times x$ and expansion ratio y . We label the width in each layer. Layers in the same block are contained in the dashed box. We separate the stages with the green lines.

4.3. Generalization Ability on COCO Object Detection

We apply the searched DenseNAS networks on the COCO [24] object detection task to evaluate the generalization ability of DenseNAS networks and show the results in Tab. 2. We choose two commonly used object detection frameworks RetinaNet [25] and SSDLite [30, 38] to conduct our experiments. All the architectures shown in Tab. 2 are utilized as the backbone networks in the detection frameworks. The experiments are performed based on the MMDetection [5] framework.

We compare our results with manually designed models [38] and ones generated by NAS methods. Results of MobileNetV2 [38], FBNet [43] and ProxylessNAS [4] are obtained by our re-implementation and all models are trained under the same settings and hyper-parameters for fair comparisons. DetNAS [7] is a recent work that aims at searching the backbone architectures directly on object detection tasks. Though DenseNAS searches on the ImageNet classification task and apply the searched architectures on detection tasks, our DenseNAS models still obtain superior detection performance in terms of both accuracy and FLOPs. The superiority over the compared methods demonstrates the great generalization ability of DenseNAS networks.

4.4. Performance on ResNet-based Search Space

We apply our DenseNAS framework on the ResNet [17]-based search space to further evaluate the generalization ability of our method. It is convenient to implement DenseNAS on ResNet [17] as we set the candidate operations in the basic layer as the basic block defined in ResNet [17] and the skip connection. Note that the ResNet-based search

Table 3: ImageNet classification results of ResNets and DenseNAS networks searched on the ResNet-based search spaces.

Model	Params	FLOPs	GPU Latency	Top-1 Acc(%)
ResNet-18 [17]	11.7M	1.81B	13.5ms	72.0
DenseNAS-R1	11.1M	1.61B	12.0ms	73.5
ResNet-34 [17]	21.8M	3.66B	24.6ms	75.3
DenseNAS-R2	19.5M	3.06B	22.2ms	75.8
ResNet-50-B [17]	25.6M	4.09B	47.8ms	77.7
DenseNAS-R3	24.7M	3.41B	41.7ms	78.0

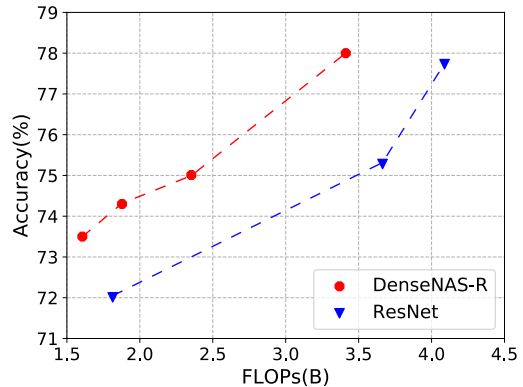


Figure 5: Graphical comparisons between ResNets and DenseNAS networks searched on the ResNet-based search space.

space is also constructed as a densely connected super network. We search for several architectures with different FLOPs and compare them with the original ResNet models on the ImageNet [10] classification task in Tab. 3. We further replace all the basic blocks in DenseNAS-R2 with the bottleneck blocks and obtain DenseNAS-R3 to compare with ResNet-50-B. All the models in Tab. 3 are trained under the same settings and hyper-parameters for a fair comparison. Our proposed DenseNAS promotes the accuracy of ResNet-18, -34 and -50-B by 1.5%, 0.5% and 0.3% with 200M, 600M, 680M fewer FLOPs and 1.5ms, 2.4ms, 6.1ms lower latency respectively. We visualize the comparison results in Fig. 5 and the performance on the ResNet-based search space further demonstrates the great generalization ability and effectiveness of DenseNAS.

4.5. Ablation Study and Analysis

Comparison with Other Search Spaces To further demonstrate the effectiveness of our proposed densely connected search space, we conduct the same search algorithm used in DenseNAS on the search spaces of FBNet and ProxylessNAS as well as a new search space which is constructed following the settings of block counts and block

widths in MobileNetV2. The three search spaces are denoted as FBNet-SS, Proxyless-SS and MBV2-SS respectively. All the search/training settings and hyper-parameters are the same as that we use for DenseNAS. The results are shown in Tab. 4 and DenseNAS achieves the highest accuracy with the lowest latency.

Table 4: Comparisons with other search spaces on ImageNet. SS: Search Space. MBV2: MobileNetV2.

Search Space	FLOPs	GPU Latency	Top-1 Acc(%)
FBNet-SS [43]	369M	25.6ms	74.9
Proxyless-SS [4]	398M	18.9ms	74.8
MBV2-SS [38]	383M	32.1ms	74.7
DenseNAS	361M	17.9ms	75.3

Comparison with Random Search As the random search [23, 39] is treated as an important baseline to validate various NAS methods. We conduct random search experiments and show the results in Tab. 1. We randomly sample 15 models in our search space whose FLOPs are similar to DenseNAS-C. Then we train every model for 5 epochs on the ImageNet dataset. Finally, we select the one with the highest validation accuracy and train the best model under the same settings with DenseNAS. The total search cost of the random search is the same as DenseNAS. We observe that DenseNAS-C is 1% accuracy higher compared with the randomly searched model, which proves the effectiveness of DenseNAS.

Table 5: Comparison with different connection number settings on ImageNet.

Connect Num	GPU Latency	Top-1 Acc(%)	Search Epochs
3	18.9ms	74.8	150
4	17.9ms	75.3	150
5	19.2ms	74.6	150
5	16.7ms	74.9	200

The Number of Block Connections We explore the effect of the maximum number of connections between routing blocks in the search space. We set the maximum connection number as 4 in DenseNAS. Then we try more options and show the results in Tab. 5. When we set the connection number to 3, the searched model gets worse performance. We attribute this to the search space shrinkage which causes the loss of many possible architectures with good performance. As we set the number to 5 and the search process takes the same number of epochs as DenseNAS, *i.e.* 150 epochs. The performance of the searched model is not good, even worse than that of the connection number 3. Then we increase the search epochs to 200 and the search process achieves a comparable result with DenseNAS. This phenomenon indicates that larger search spaces need more

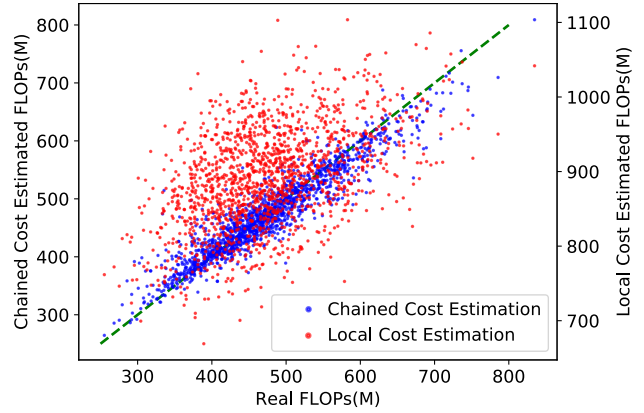


Figure 6: Predicted values of FLOPs computed by chained cost estimation and local cost estimation algorithm.

search cost to achieve comparable/better results with/than smaller search spaces with some added constraints.

Cost Estimation Method As the super network is densely connected and the final architecture is derived based on the total transition probability, the model cost estimation needs to take the effects of all the path probabilities on the whole network into consideration. We try a *local cost estimation* strategy that does not involve the global connection effects on the whole super network. Specifically, we compute the cost of the whole network by purely summing the cost of every routing block during the search as follows,

$$\text{cost} = \sum_i^B \left(\sum_{j=i-m}^{j=i-1} p_{ji} \cdot \text{cost}_{align}^{ji} + \text{cost}_b^i \right), \quad (9)$$

where all definitions in the equation are the same as that in Eq. 6. We randomly generate the architecture parameters (α and β) to derive the architectures. Then we draw the approximated cost values computed by the *local cost estimation* and our proposed *chained cost estimation* algorithm respectively, and compare the predicted values with the real cost values in Fig. 6. In this experiment, we take FLOPs as the model cost because the FLOPs is easier to measure than the latency. 1,500 models are sampled in total. The results show that the predicted cost computed by our *chained cost estimation* algorithm has a much stronger correlation with the real latency value. Moreover, the predicted values computed by the *chained cost estimation* algorithm approximate more to the real values. As the predicted values are computed based on the randomly generated architecture parameters which are not binary parameters, there are still differences between the predicted values and real values.

Architecture Analysis We visualize the searched architectures in Fig. 4. It shows that DenseNAS-B and -C have

one more block in the last stage than architectures in other methods, which indicates enlarging the depth in the last stage of the network tends to obtain a better accuracy. Moreover, the smallest architecture DenseNAS-A whose FLOPs is only 251M has one fewer block than DenseNAS-B and -C to decrease the model cost. The structures of the final searched architectures show the great flexibility of DenseNAS.

5. Conclusion

We propose a densely connected search space for more flexible architecture search, DenseNAS. We tackle the limitations in previous search space design in terms of the block counts and widths. The novel designed routing blocks are utilized to construct the search space. The proposed chained cost estimation algorithm aims at optimizing both accuracy and model cost. The effectiveness of DenseNAS is demonstrated on both MobileNetV2- and ResNet- based search spaces. We leave more applications, *e.g.* semantic segmentation, face detection, pose estimation, and more network-based search space implementations, *e.g.* MobileNetV3 [19], ShuffleNet [47] and VarGNet [46], for future work.

Acknowledgement

We thank Liangchen Song and Kangjian Peng for the discussion and assistance.

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. 3, 5, 10
- [2] Yoshua Bengio and Yann LeCun, editors. *ICLR*, 2015. 6
- [3] Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. *arXiv:1708.05344*, 2017. 3
- [4] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019. 1, 2, 3, 4, 5, 6, 7, 8, 10
- [5] Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, Wansen Feng, Ziwei Liu, Jiarui Xu, Zheng Zhang, Dazhi Cheng, Chenchen Zhu, Tianheng Cheng, Qijie Zhao, Buyu Li, Xin Lu, Rui Zhu, Yue Wu, and Dahua Lin. Mmdetection: Open mmlab detection toolbox and benchmark. *arXiv:1906.07155*, 2019. 7
- [6] Liang-Chieh Chen, Maxwell D. Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jonathon Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *NeurIPS*, 2018. 1
- [7] Yukang Chen, Tong Yang, Xiangyu Zhang, Gaofeng Meng, Chunhong Pan, and Jian Sun. Detnas: Neural architecture search on object detection. In *NeurIPS*, 2019. 6, 7
- [8] Xiangxiang Chu, Bo Zhang, Ruijun Xu, and Jixiang Li. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *arXiv:1907.01845*, 2019. 1, 2
- [9] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matt Uyttendaele, and Niraj K. Jha. Chamnet: Towards efficient network design through platform-aware model adaptation. *arXiv:1812.08934*, 2018. 2
- [10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009. 2, 5, 7
- [11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009. 3
- [12] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *ECCV*, 2018. 3
- [13] Xuanyi Dong and Yi Yang. Network pruning via transformable architecture search. *arXiv:1905.09717*, 2019. 3
- [14] G David Forney. The viterbi algorithm. *Proceedings of the IEEE*, 1973. 5, 10
- [15] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *CVPR*, 2019. 1
- [16] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *CVPR*, 2018. 2
- [17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 2, 5, 7, 11
- [18] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv:1503.02531*, 2015. 3
- [19] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. *arXiv:1905.02244*, 2019. 9
- [20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017. 6
- [21] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *ICLR*, 2017. 3
- [22] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *University of Toronto*. 3
- [23] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv:1902.07638*, 2019. 8
- [24] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. In *ECCV*, 2014. 5, 7
- [25] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *ICCV*, 2017. 7

- [26] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan L. Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *CVPR*, 2019. 1, 2
- [27] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV*, 2018. 2
- [28] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *ICLR*, 2018. 2, 3
- [29] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *ICLR*, 2019. 2, 3, 6
- [30] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In *ECCV*, 2016. 7
- [31] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. In *ICLR*, 2017. 6
- [32] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *NeurIPS*, 2017. 2
- [33] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR*, 2017. 3
- [34] 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. 6
- [35] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018. 1, 2, 3
- [36] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In *ICML*, 2017. 2
- [37] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *arXiv:abs/1802.01548*, 2018. 1, 2, 3, 6
- [38] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018. 1, 2, 6, 7, 8, 11, 12
- [39] Christian Sciuto, Kaicheng Yu, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. *arXiv:1902.08142*, 2019. 8
- [40] 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. 6
- [41] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv:1807.11626*, 2018. 1, 2, 4, 6
- [42] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv:1905.11946*, 2019. 2
- [43] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *arXiv:1812.03443*, 2018. 1, 2, 3, 4, 6, 7, 8
- [44] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. In *ICCV*, 2019. 2
- [45] Hang Xu, Lewei Yao, Wei Zhang, Xiaodan Liang, and Zhen-guo Li. Auto-fpn: Automatic network architecture adaptation for object detection beyond classification. In *ICCV*, 2019. 1
- [46] Qian Zhang, Jianjun Li, Meng Yao, Liangchen Song, He-long Zhou, Zhichao Li, Wenming Meng, Xuezhi Zhang, and Guoli Wang. Vargnet: Variable group convolutional neural network for efficient embedded computing. *arXiv:1907.05653*, 2019. 9
- [47] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *arXiv:1707.01083*, 2017. 9
- [48] Yiheng Zhang, Zhaoan Qiu, Jingen Liu, Ting Yao, Dong Liu, and Tao Mei. Customizable architecture search for semantic segmentation. In *CVPR*, 2019. 2
- [49] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *CVPR*, 2018. 3
- [50] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016. 1, 3
- [51] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *arXiv:1707.07012*, 2017. 1, 2, 3, 6

A. Appendix

A.1. Viterbi Algorithm for Block Deriving

The Viterbi Algorithm [14] 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} is served 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.

A.2. 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, 4] (which is mentioned in Sec. 3.4). When training the weights of operations, we sample one path of the candidate operations according to the

Table 6: Architectures searched by DenseNAS in the ResNet-based search space.

Stage	Output Size	DenseNAS-R1	DenseNAS-R2	DenseNAS-R3
1	112×112	$3 \times 3, 32, \text{stride } 2$		
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		

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    $P[i] \leftarrow \max_{i-1 \leq j \leq i-M_i} (P[i-1] \cdot p_{ji})$ ;
5    $S[i] \leftarrow \arg \max_{i-1 \leq j \leq i-M_i} (P[i-1] \cdot p_{ji})$ ;
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 reverse  $X$ ;
```

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.

$$\text{bias}_s = \ln \frac{\sum_{o \in \mathcal{O}_s} \exp(\alpha_o^\ell)}{\sum_{o \in \mathcal{O}_s} \exp(\alpha'_o^\ell)}, \quad (10)$$

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 α'_o^ℓ denotes the updated value of the architecture parameter. The computed bias is finally added to the updated architecture parameters.

A.3. 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 [17] 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 [38] 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. 6.

Table 7: 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.4. 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 [38]-based search space with the local estimation strategy and show the searched results in Tab. 7. 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.