Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

SEArch: A self-evolving framework for network architecture optimization

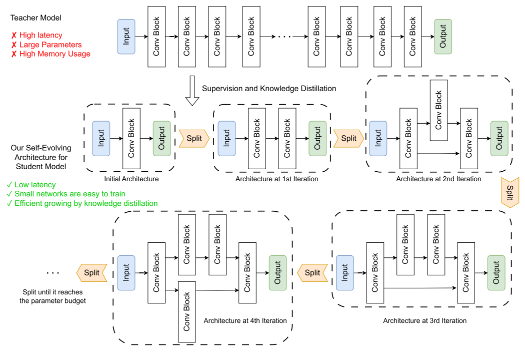
Yongqing Liang, Dawei Xiang, Xin Li[*]

Texas A&M University, College Station, TX 77845, United States

HIGHLIGHTS

- We propose a self-evolving network architecture optimization framework incorporating the advantages of network architecture pruning, knowledge distillation, and neural architecture search approaches. It can optimize an existing network into a more efficient network with better accuracy and smaller parameter size.
- We propose two new designs for network architecture optimization, the first one is the attention layer to identify the bottleneck of the network performance; the second one is edge-splitting scheme for efficient network modification and construction.
- We demonstrate our framework through comprehensive experiments achieving state-of-the-art accuracy and parameter sizes compared to existing network pruning and knowledge distillation algorithms.

GRAPHICAL ABSTRACT


Teacher Model
✗ High latency
✗ Large Parameters
✗ High Memory Usage
Input
Conv Block
Output
Supervision and Knowledge Distillation
Our Self-Evolving Architecture for Student Model
✓ Low latency
✓ Small networks are easy to train
✓ Efficient growing by knowledge distillation
Initial Architecture
Split
Architecture at 1st Iteration
Architecture at 2nd Iteration
Architecture at 3rd Iteration
Architecture at 4th Iteration
Split until it reaches the parameter budget

ARTICLE INFO

Communicated by K. Li

Keywords:
Deep neural network
Knowledge distillation
Neural network pruning
Automated machine learning
Neural architecture search

ABSTRACT

This paper studies a fundamental network optimization problem that finds a network architecture with optimal performance (low loss) under given resource budgets (small number of parameters and/or fast inference). Unlike existing network optimization approaches such as network pruning, knowledge distillation (KD), and network architecture search (NAS), in this work we introduce a self-evolving pipeline to perform network optimization. In this framework, a simple network iteratively and adaptively modifies its structure by using the guidance from a teacher network, until it reaches the resource budget. An attention module is introduced to transfer the knowledge from the teacher network to the student network. A splitting edge scheme is designed to help the student model find an optimal macro architecture. The proposed framework combines the advantages of pruning, KD, and NAS, and hence, can efficiently generate networks with flexible structure and desirable performance. Extensive experiments on CIFAR-10, CIFAR-100, and ImageNet demonstrate that our framework achieves great performance in this network architecture optimization task.

[*] Corresponding author.
Email address: xinli@tamu.edu (X. Li).

https://doi.org/10.1016/j.neucom.2025.130980
Received 15 August 2024; Received in revised form 31 January 2025; Accepted 8 July 2025
Available online 10 July 2025

1. Introduction

Deep neural networks (DNNs) have achieved state-of-the-art performance in numerous computer vision and natural language processing tasks. The architectures of these networks are often manually designed and fine-tuned using large datasets. While these designed networks achieve state-of-the-art accuracy on various tasks, they often result in complex architectures with a large number of parameters. In many applications, networks need to be simplified to be deployed on specific platforms with constrained resources. For example, mobile and portable devices have limited processing power and memory capacity, which require the deployed networks to have a small parameter size. Another case is real-time artificial intelligence tasks, which have latency requirements that necessitate networks with fewer parameters while maintaining good accuracy.

To simplify or optimize a given well-performing but oversized network, three main network optimization strategies have been widely studied in the literature: (1) network pruning [1], (2) knowledge distillation [2], and (3) neural architecture search [3]. Generally, an ideal network should achieve high accuracy on benchmarks while operating within a user-defined parameter budget.

- Network pruning methods iteratively and selectively remove layers or reduce convolutional channels according to the given budget. Although generally computationally efficient for smaller parameter sizes, these methods frequently lead to sub-optimal accuracy on the benchmark.
- Knowledge distillation (KD) methods transfer knowledge from a larger, trained network (the teacher) to a smaller network (the student). The presence of a teacher network enhances training convergence. However, the architecture of the student network is usually manually predefined, without optimizing the network architecture, which can compromise its accuracy.
- Neural architecture search (NAS) methods aim to discover the most optimal architecture by navigating a vast space of network components. While this approach can yield networks with adaptable structures and state-of-the-art scores, conducting a thorough search is frequently prohibitively expensive and demands substantial computational resources.

We denote the given well-trained network in need of simplification as the *teacher model* and assume that it has acquired effective feature maps. Our goal is to discover a new network, denoted as the *student model*, with a constrained number of parameters while maintaining optimal performance, such as classification accuracy. We introduce a **S**elf-**E**volving **Arch**itecture optimization framework (SEArch) that is both effective (in terms of performance) and efficient, leveraging the strengths of these three strategies. Our **main idea** is to iteratively evolve a student network, allowing it to dynamically adjust its architecture until the given resource constraint is reached. The student network initiates with a very simple structure. During the optimization process, we iteratively identify and modify the bottlenecks under the guidance of the teacher model's feature maps.

Unlike network pruning methods that simply trim down the original network and knowledge distillation methods that refine a network with a fixed structure, our evolving approach allows for flexible adjustments to the overall network architecture. Compared with NAS methods that discover a network from a large supernet, our SEArch framework constructs an optimal architecture in a reverse manner. We iteratively identify the bottleneck of the student network and refine the corresponding structure, hence, our search demonstrates faster convergence.

Our **main contributions** are summarized as follows:

- We propose a **S**elf-**E**volving network **Arch**itecture optimization framework, or SEArch. This framework iteratively adjusts the network architecture topology, incorporating the benefits of network pruning (network parameter reduction), knowledge distillation (high accuracy preservation), and architecture search methods (flexible architecture design).
- To facilitate faster transfer learning and network topology optimization, we design an effective attention mechanism to identify the bottleneck of the current neural network, and an edge-splitting scheme to optimize the structure of the bottleneck for better performance.
- Comprehensive experiments demonstrate our framework achieves state-of-the-art accuracy and parameter sizes when compared with existing network pruning and knowledge distillation algorithms.

2. Related work

Network pruning. Traditional pruning approaches can be grouped into two categories: structured pruning and unstructured pruning. Structured pruning He et al. [4], Lin et al. [5] removed entire layers or filters of a network to preserve its structural regularity. The resulting networks can be easily developed and deployed. Unstructured pruning Hou et al. [6] sparsified the convolutional weights or feature maps. The pruned networks require specific hardware to speed up the training and inference. These methods focus on removing the least significant network parameters or components. And the pruned networks often have a clear performance drop. Dong and Yang [7] built a new network with flexible channels layer-by-layer and cannot optimize the global topological architectures. Hence, their expressive capability and performance are limited by this fixed architecture. DNN quantization and decomposition depend on hardware and are out of our scope. Recently, Liu et al. [8] used the attention-based adaptive method for network pruning. Zheng et al. [9] evaluated the direct and indirect effects of filters to prune the network.

Knowledge distillation. Knowledge distillation (KD) transfers information from a trained teacher network to a (often smaller) student network to reduce network complexity. Hinton et al. [10] introduced a knowledge distillation strategy to compress the model. The student network learns how a large network studies given tasks under this teaching procedure. Recently, Shao et al. [2], Xu et al. [11], Cho et al. [12], Zhang et al. [13], Lu et al. [14] also proposed distillation methods to make student model more efficient. Besides, Lin et al. [15] introduced a "target-aware transformer" to align varying semantic information at the same spatial location. Zhang et al. [13] applied knowledge distillation to the salient object detection task. Lu et al. [14] proposed an adaptive lightweight network construction method. The network architecture distillability is estimated based on architecture similarity and clustering ability. Dong et al. [16] used feature semantic similarity between the teacher and student models as an indicator of distillation performance and built a training-free search framework. Inspired by existing KD methods, our approach transfers both response-based knowledge (output from the final layers) and feature-based knowledge (output from intermediate layers) to the student model. Finally, unlike existing KD methods where the student's architecture is manually designed and remains fixed during optimization, our scheme actively optimizes the student model's architecture throughout the knowledge transfer process, and this helps improve the network's performance.

Neural architecture search. NAS methods typically construct a huge and comprehensive search space including many building block architectures, then search for an optimal combination of them, using deep learning based, evolutionary algorithm based, or gradient based algorithms. Although various approximate algorithms (e.g., cell-based [3, 17], knowledge transfer based [18] and differentiable architecture [19,20]) have been explored to accelerate the architecture search and training, the prohibitively expensive computational cost of thorough training and searching significantly hinders the effectiveness of NAS. On the contrary, we search for the new architecture in a reverse manner where our SEArch starts from a simple network and iteratively adds new convolutional operators to improve its performance.

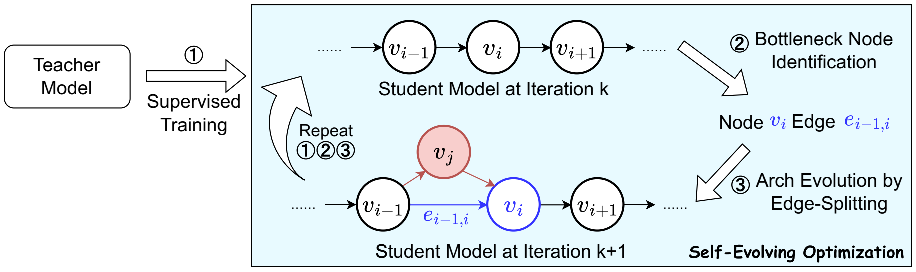

Teacher Model
①
Supervised Training
Repeat ①②③
Student Model at Iteration k
② Bottleneck Node Identification
Node v_i Edge $e_{i-1,i}$
③ Arch Evolution by Edge-Splitting
Student Model at Iteration k+1
Self-Evolving Optimization

Fig. 1. An overview of the proposed pipeline. Given a well-trained teacher model that requires optimization, we iteratively evolve and grow a student network until the given resource constraint is reached. In each iteration, the optimization process includes three stages: (1) transferring knowledge from the teacher model to the student model by supervised training; (2) selecting a candidate edge according to the modification value score; and (3) modifying the network structure of the candidate edge by edge splitting to enhance its expressiveness.

3. Approach

Compared with mainstream network optimization approaches that shrink existing networks or find an optimal network from a huge supernet, our self-evolving neural network optimization is in a reverse manner. It begins with a basic, primitive network (*student* model) and progressively evolves into superior ones by incorporating new convolutional operators and modifying its architectural topology. Knowledge is transferred from the teacher network to the student network through learning intermediate layers' outputs and final predictions. This growing scheme assists the network in better maintaining its performance, and in some cases, even surpassing that of the original network. Fig. 1 shows an overview of the proposed pipeline. In this work, we validate this design on the classic image classification task, where a color image is taken as input and its class label is predicted.

3.1. Definitions and pipeline overview

A neural network can be described using a directed acyclic graph $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ comprising $\mathbb{V}$ nodes and $\mathbb{E}$ edges. In this context, each node $v_i \in \mathbb{V}$ represents a latent feature map, and each directed edge $e_{i,j} \in \mathbb{E}$ is associated with a specific operation $o_{i,j}$ that transforms feature map from v_i to v_j. When a node v_i has more than one incident edge, each edge generates a feature map, and the resulting feature map of v_i is the sum of these individual feature maps.

Given a well-trained network, denoted as the *teacher* model, $\hat{\mathcal{G}} = (\hat{\mathbb{V}}, \hat{\mathbb{E}})$ requires optimization with respect to the number of parameters or FLOPs. We identify the longest path originating from the input and terminating at the final feature map. The feature maps along this path constitute a list of nodes in order $\hat{\mathbb{V}} = \{\hat{v}_1, \hat{v}_2, \dots, \hat{v}_m\}$ and the teacher model is partitioned into m layers. $\hat{v}_1$ and $\hat{v}_m$ are the source and sink of the graph $\hat{\mathcal{G}}$, respectively.

The student model is initialized to a basic two-layer network, denoted as $\mathcal{G}_1 = (\mathbb{V}_1, \mathbb{E}_1)$. $\mathcal{G}_1$ comprises two nodes $\mathbb{V}_1 = \{v_1, v_2\}$, where v_1 represents the input image and v_2 is the final feature map fed into the classifier. The only edge $e_{1,2} = (v_1, v_2)$ in $\mathbb{E}_1$ denotes a convolution operation that transforms data from v_1 to v_2. Subsequently, the proposed framework iteratively optimizes the student model $\mathcal{G}_z (z \geq 1)$ until its model size reaches the budgetary limit B. Our pipeline consists of four stages. In each iteration, the first stage involves supervised training, which transfers knowledge from the teacher model to the student model $\mathcal{G}_z$ (Section 3.2). Once the student model converges, the second stage identifies the bottleneck to refine (Section 3.3). The third stage modifies the topological structure of this bottleneck through edge-splitting (Section 3.4). We repeat the first to the third stages for each iteration until the student model reaches the user-defined parameter size. Fig. 1 shows an overview of the proposed pipeline. Fig. 2 illustrates an example of how the student's architecture evolves during the self-evolving scheme. The fourth stage involves end-to-end training of the student model on the ground truth data without the teacher model.

3.2. Supervised learning through the teacher model

Training student model $\mathcal{G}_z$ allows us to optimize its parameter weights and identify the bottleneck structure for architecture evolution. The student model is expected to imitate the behavior of the teacher model, including the network's output and the inner feature maps. Hence, we use the inner feature maps $\hat{\mathbb{V}}$ produced by the teacher model as checkpoints to supervise the student model's inner nodes.

In the first iteration, the student model is a two-node network with nodes v_1 and v_2. Here v_1 represents the input image while v_2 contains the final feature map. We construct a mapping table q to map each student node to a teacher node in $\hat{\mathbb{V}}$. Then q_i indicates that teacher node $\hat{v}_{q_i}$ transfers knowledge to the student node v_i during training. For example, if $q_2 = m$, it means that the student node v_2 is supervised by the final layer $\hat{v}_m$ of the teacher model. When introducing a new node v_k into the student model, we match it with an appropriate teacher node $\hat{v}_{k'}$, denoted by $q_k = k'$. Further details on the construction of q can be found in Section 3.4.

The student model's weights are optimized from two perspectives using the training set. First, we minimize the classification loss $\mathcal{L}_{cls}$ that computes the differences between the student model's output and the ground truth labels. When the task is image classification, we use Cross Entropy Loss for $\mathcal{L}_{cls}$. The second objective function aims to minimize the learning loss between the student model and the teacher model. Although student node (v_i) and its corresponding teacher node ($\hat{v}_{q_i}$) can have feature maps with identical height and width dimensions, they often differ in the number of channels. Inspired by Lin et al. [15], we employ an attention module f_a to facilitate the aggregation of feature channels from the teacher model for transfer learning. Unlike [15], which focuses on finding matching features in spatial locations, our model aims to identify matching features in channel space. For every channel in the student's feature map, an attention query is used to calculate the corresponding channel weights in the teacher's feature map. These calculated weights are then used to project the teacher's feature maps into a feature space that matches the channel dimensions of the student model's feature map. A projected feature map for the teacher node $\hat{v}_{q_i}$ can be obtained as

$$f_a(\hat{v}_{q_i}) = \mathrm{Atten}(v_i, \hat{v}_{q_i}, \hat{v}_{q_i}). \tag{1}$$

We use the L2 norm to measure the differences between feature maps of the student node v_i and teacher node $\hat{v}_{q_i}$, denoted as

$$R_{inner}(v_i) = ||v_i - f_a(\hat{v}_{q_i})||_2^2. \tag{2}$$

Then, we define the imitation loss $\mathcal{L}_{inner}$ of the student model as the average of the difference across all student nodes,

$$\mathcal{L}_{inner} = \frac{1}{|\mathbb{V}|} \sum_{v_i \in \mathbb{V}} R_{inner}(v_i), \tag{3}$$

where $|\mathbb{V}|$ represents the number of nodes in the student model. Fig. 3 visualizes the supervised training in one iteration. We train the parameter weight of the student model by minimizing

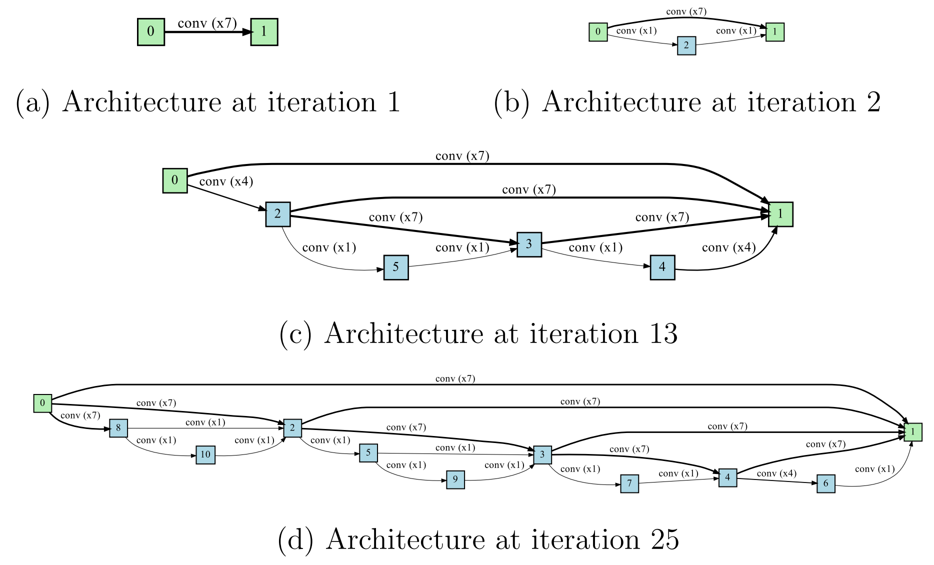

conv (x7)
(a) Architecture at iteration 1
conv (x1)
(b) Architecture at iteration 2
conv (x4)
(c) Architecture at iteration 13
(d) Architecture at iteration 25

Fig. 2. Visualizations of our SEArch algorithm for optimizing the architecture of student network at different iterations. An example shows how our algorithm builds a network from scratch. The experiment was conducted on CIFAR-10. In this graph, the edges are operations and the nodes are the data for processing. The green nodes v_0 and v_1 are the first stage (input image) and the final stage (final feature map), respectively. The blue nodes are the added structures through the architecture's self-evolving optimization. When there is more than one incoming edge to a node, we add the results together. Conv denotes the 3×3 residual separable convolution, and the notation $(\times N)$ indicates the number of stacked operations. From left to right, the input data are processed/convolved by different operations to produce the output node. (a) shows the initial status of the network. (d) shows the final status of the optimized network. (a–d) show the changes in the network architectures at different iterations.

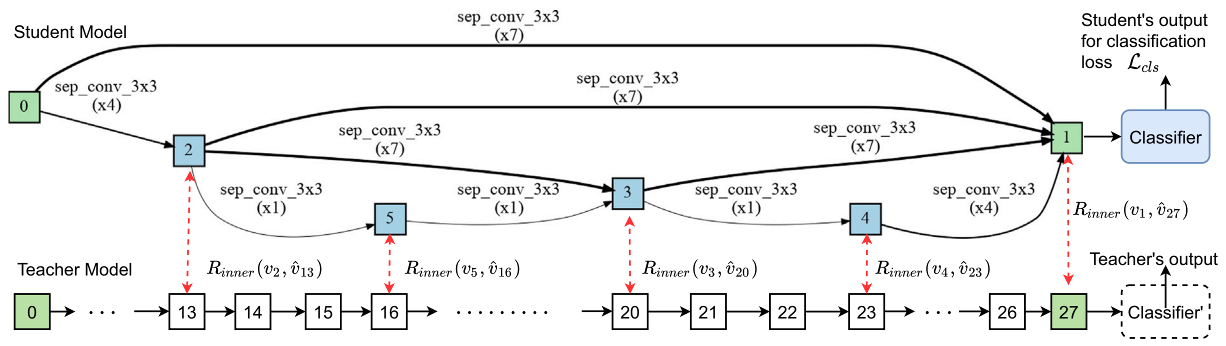

Student Model
sep_conv_3x3 (x7)
sep_conv_3x3 (x4)
sep_conv_3x3 (x1)
Student's output for classification loss $\mathcal{L}_{cls}$
Classifier
$R_{inner}(v_1, \hat{v}_{27})$
Teacher Model
$R_{inner}(v_2, \hat{v}_{13})$
$R_{inner}(v_5, \hat{v}_{16})$
$R_{inner}(v_3, \hat{v}_{20})$
$R_{inner}(v_4, \hat{v}_{23})$
Teacher's output
Classifier

Fig. 3. Illustrations of supervised learning for a student model. The student model is at the search iteration 13 on CIFAR-100. The teacher model is a well-trained ResNet-56 network that has 27 layers (at the bottom). Both models take the same input image (Node 0). The classification loss $\mathcal{L}_{cls}$ compares the outputs of the student model and the training labels. R_{inner} computes the differences in feature maps between the student and the teacher, indicated by dashed red arrows. At each iteration, the student model is trained on the training set under the supervision of the teacher model (inner nodes) and the ground truth data (last node). An attention module is used to distill knowledge from the teacher model to the student model, guided by the loss R_{inner}.

$$\mathcal{L} = \mathcal{L}_{cls} + \alpha\mathcal{L}_{inner}. \tag{4}$$

In our experiment, we simply set $\alpha = 1$ at the beginning and use a cosine function to gradually decrease the α to zero.

3.3. Bottleneck identification for architecture optimization

We model network learning as an information gain process. The main idea is similar to network pruning approaches which identify the local structure of the network for optimization in the next stage. Here we design a scheme to identify the bottleneck of the student model and then improve its topological architecture. A bottleneck implies that making modifications to this local structure could lead to large potential performance gains.

Although R_{inner} indicates the difference between the student node and the teacher node, it doesn't actually reflect the bottleneck node (see the ablation study). We further define a modification value score S to estimate the bottleneck node.

Considering a node v_j in the student model, we define another student node v_i that shares its incoming incident edge (u_i, v_j) as its *precursor node*. Similarly, we define a student node v_k that shares its outgoing incident edge (v_j, u_k) as its *successor node*.

(1) First, if the feature map at node v_j is inaccurate, it propagates this inaccuracy, affecting all its successor nodes. Hence, improving the accuracy of v_j will benefit all its successor nodes. We use the out-degree $\deg_j^+$ to denote the number of successor nodes of node v_j. Assuming that the error $R_{inner}(v_j)$ could be minimized to a theoretical minimum of 0 after refining node v_j, the positive impact would be proportional to $\deg_j^+ \times R_{inner}(v_j)$.

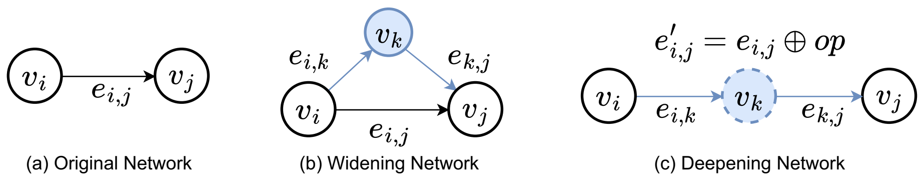


Fig. 4. Illustrations of the proposed edge-splitting optimizations. For example, (a) is the initial state of the most valuable structure for modification. Here, v_i and v_j represent the input and the output of the operation $e_{i,j}$, respectively. Since this structure acts as the bottleneck of the entire network, we modify it to enhance its learning capability. (b) illustrates the first option: widening the network by adding a supervised node v_k. This modification increases the learning capability of the local region from v_i to v_j by introducing an extra transform and two learnable convolutional operations. (c) demonstrates the second option: deepening the network by adding an implicit node v_k. This operation further enhances learning capability by incorporating an additional learnable convolutional operation.

(2) Second, to modify the feature map of node v_j, we often need to change the network architecture from its precursor nodes to v_j. Each time, we select one precursor node v_i and add additional convolutional operations to increase the depths/widths of the local network, named, splitting the edge $e_{i,j}$. We use the in-degree $\deg_j^-$ to denote the number of precursor nodes of node v_j. Assuming each incoming edge of v_j independently and evenly contributes to the error $R_{inner}(v_j)$, an improvement from modifying one of the $\deg_j^-$ incoming edges can be expected to be proportional to $\frac{1}{\deg_j^-} \times R_{inner}(v_j)$.

Finally, we define the *modification value* score S for each student node v_j as

$$S(v_j) = \frac{\deg_j^+}{\deg_j^-} R_{inner}(v_j), \tag{5}$$

where $R_{inner}(v_j)$ is the deviation of this node's feature map from the teacher model, and $\frac{\deg_j^+}{\deg_j^-}$ is the adjustment term that estimates the potential performance gain from modifying the local structure. We compute S for each student node using the validation dataset. The node with the largest S is selected and its local structure will be refined by edge-splitting. In our experiments, we selected the edge associated with the closest precursor node for architecture evolution.

3.4. Architecture evolution by edge splitting

To optimize the bottleneck structure of the current network, we propose an edge-splitting approach that can update the network connections by adding new blocks to it. While its capability is comparable to neural architecture search (NAS), our method avoids the prohibitively expensive computational costs associated with NAS's exhaustive training and search processes. Unlike NAS, our approach iteratively enhances its performance by adding new convolutional operators in a targeted manner.

Suppose the current student network is $\mathcal{G}_z = (\mathbb{V}, \mathbb{E})$. Node $v_j \in \mathbb{V}$ and edge $e_{i,j} = (v_i, v_j) \in \mathbb{E}$ are selected for improvement by edge-splitting. Our design is based on the observation that: *Increasing the depth or width of a network can either maintain or improve output accuracy*. Hence, if the model size of a given network (e.g., number of parameters, FLOPs) has not reached the budget limit, we evolve its architecture by adding new convolutional operations to enhance both its size and performance.

To refine a local structure at node v_j, which has not approximated the teacher network well, we introduce a new node v_k between its precursor node v_i and v_j, and perform two types of edge-splitting: widening the network and deepening the network.

(1) *Widening the network.* One option is to create a new branch beside $e_{i,j}$ by inserting a new node v_k in the student model, as illustrated in Fig. 4(b). Two new convolutional operations are also added to the student model as $e_{i,k}$ and $e_{k,j}$. This operation increases the width of the student network to improve its capability.

At the beginning, the student model typically consists of only a few layers (nodes), while the teacher model—being more complex and well-trained—may contain hundreds of layers (nodes). When adding a new layer to the student model, the ideal approach is to select an inner layer from the teacher model for knowledge distillation and supervision. To achieve this, we designed the following equation to facilitate the selection of the inner layer. For example, when adding a layer between v_i and v_j in the student model, their corresponding nodes in the teacher model are q_i and q_j. We then select the inner node between q_i and q_j as the supervision target, as illustrated in Fig. 3. The red lines in the figure indicate the supervision relationship:

$$q_k = \lfloor c \cdot q_i + (1 - c) \cdot q_j \rfloor, \tag{6}$$

where $c \in [0, 1]$ is a hyper-parameter. Users can adjust the parameter c to obtain different knowledge distillation results. We conducted an ablation study in the experiment section, we demonstrated that using the middle node ($c = 0.5$) yields the highest accuracy (see Table 1). This can be explained by the intuition that the information gain of a shallow network is proportional to its depth. Thus the total information gain

Table 1
Ablation study on CIFAR-10. "Base Acc." and "Pruned Acc" are accuracy of the baseline and optimized networks. "Acc. Drop" is the accuracy drop (smaller is better), where a negative value means the optimized model outperforms the baseline. "PARAMs" is the network parameters.

Method	Base Acc. (%)	Pruned Acc. (%)	Acc. Drop (%)	PARAMs (M)	FLOPs (M)
Baseline: ResNet-56	93.95	–	–	0.85	126.6
(A) Ours w/o S (Eq. 5)	93.95	84.18 (±3.34)	9.77	0.41	90.9
(B) Ours $c = 0.25$ (Eq. 6)	93.95	94.62	−0.67	0.40	78.9
(B) Ours $c = 0.75$ (Eq. 6)	93.95	94.50	−0.55	0.40	142.1
(C) Random splitting (0.40)	93.95	91.79 (±2.14)	2.16	0.40	63.1
Our full model (0.40)	93.95	**94.82** (±0.14)	**−0.87**	0.40	63.1
Baseline: ResNet-110	94.04	–	–	1.73	255.0
(C) Random splitting (0.68)	94.04	92.81 (±0.52)	1.23	0.68	85.5
Our full model (0.68)	94.04	**95.00** (±0.02)	**−0.96**	0.68	85.5

Bold values indicate the best performance in each column.

Algorithm 1 Self-evolving architecture overview.

```
Require: Teacher model $\hat{\mathcal{G}}$, training $D_T$, valid $D_V$, budget $B$, $B_{op}$, Conv op
Ensure: Student model $\mathcal{G}$
 1: $\mathcal{G}_1 \leftarrow$ two-layer network, $z \leftarrow 1$
 2: while $|\mathcal{G}_z| < B$ do
 3:     Train $\mathcal{G}_z$ on $D_T$ using Eq. (4)
 4:     Select $v_j$ and $e_{i,j}$ on $D_V$ using Eq. (5)
 5:     if $|e_{i,j}| < B_{op}$ then
 6:         Deepen $e'_{ij} = e_{ij} \oplus op$
 7:     else
 8:         Widen. Add node $v_k$, $e_{i,k}$, $e_{k,j}$
 9:         Set $q_k$ from $\hat{\mathcal{G}}$ using Eq. (6)
10:     end if
11:     $z = z + 1$
12: end while
```

can be represented as $c(1-c)$, which is maximized at $c = 0.5$. We also discussed the detailed comparisons of choosing c in the ablation study.

(2) *Deepening the network.* The second option is to deepen the network by adding a new node v_k between v_i and v_j with new convolutional operations. To reduce the network latency, we didn't assign a teacher node to supervise the training of node v_j. In other words, this operation can be considered as stacking/concatenating a new convolutional operation op to the old ones on $e_{i,j}$, as illustrated in Fig. 4(c).

While either widening or deepening the local structure could increase the capability of the network, deepening operation is simpler and introduces less computational overhead. In our self-evolving framework, we first use the deepening modification until the number of stacked operations reaches a predefined number B_{op}. Algorithm 1 summarizes the proposed self-evolving framework.

3.5. Convolutional operation

We perform the network optimization through self-evolving by incorporating the network creation capability from neural architecture search. Most architecture search algorithms first define a set of candidate operations O, then search and pick the best operations during optimization. Having a big operation set provides more choices and bigger diversity in designing networks, but it significantly slows down the search runtime. Recent study Yang et al. [21] suggested that when searching a neural architecture, the quality of macro-structures (edge connections) is more important than micro-structures (operations). In this work, we only use 3×3 separable convolution as op to build the network. The expressivity analysis can be found in Appendix A.

4. Experiments

We evaluated our SEArch algorithm using image classification task and compared it with state-of-the-art network optimization approaches: network pruning and knowledge distillation. Due to the page limit, we discuss the main results here and move the implementation details and runtime analysis to Appendices B and C.

4.1. Ablation study

We conducted ablation studies on the CIFAR-10 dataset to evaluate the effectiveness of each component in the proposed SEArch model.

Effectiveness of modification value score in Eq. (5). The results are reported in Table 1 Exp (A). We removed the adjustment term $\frac{\deg^+}{\deg^-}$ in Eq. (5), and the rest parts are the same as our full model. With this setting, the bottleneck node was selected with the largest R_{inner}. The accuracy of the optimized network drops by 9.77 % (see the Line Ours w/o S). This value was computed as the difference between the baseline accuracy (93.95 %) and the pruned accuracy (84.18 %). Our full model has 0.87 % accuracy gain instead of the accuracy drop. It shows that our modification value score in Eq. (5) is critical to identifying the bottleneck to guide the architecture modification.

Different supervision layer of teacher model in Eq. (6). In Eq. (6), the parameter c determines the index of intermediate layers of the teacher model that is assigned to supervise the new node in the student model. We conducted experiments by setting $c = 0.25$ (close to the precursor node) and $c = 0.75$ (close to the bottleneck node). In Table 1 Exp (B), the results for $c = 0.25$ and $c = 0.75$ are slightly weaker than our full model ($c = 0.50$). We concluded that our SEArch is robust to different settings of c. Besides the information gain process intuition, the experiment supports choosing the middle layer of the teacher model which provides the best results.

Bottleneck identification vs. random edge-splitting. A random edge-splitting setting is to randomly pick edges from the student model, and randomly adding new operations during iterative optimization until the model reaches the predefined model size budget. We ran the random edge-splitting three times and recorded the average results. With the proposed bottleneck identification strategy, our SEArch model is able to select key nodes for architecture optimization. Table 1 Exp (C) reports the performance of the random splitting strategy and our full SEArch model. When the baseline is ResNet-56, the random splitting strategy has a 2.16 % accuracy drop while our model achieved 0.87 % accuracy improvement from the baseline model. When the baseline is ResNet-110, the random splitting strategy has 1.22 % accuracy drop while our model yields 0.97 % accuracy gain.

4.2. Comparisons with network pruning methods

Results on CIFAR-10. We follow the common evaluation settings of network pruning papers. On CIFAR-10, we evaluated our SEArch algorithm on ResNet with depths 56 and 110. Note that the weights of the baseline models vary in these papers. We chose a baseline model with relatively high accuracy for fair comparisons because when starting with a more accurate model, it is harder for pruning to maintain the accuracy. We chose the widely adopted pretrained model Chenyaofo [34] as the baseline model. We set the number of parameters as our pruning target to 0.40 M. Following experiments done in other pruning papers, we ran the model three times and reported the "mean (± std)" results. The comparisons of optimizing ResNet-110 are discussed in Appendix D.

The quantitative comparisons of optimizing ResNet-56 are shown in Table 2. The proposed SEArch outperforms the existing pruning methods. Our method pruned the ResNet-56 by 50.2 % FLOPs, and the optimized networks actually surpassed the baseline by 0.87 % in accuracy. Conventional pruning methods remove redundant filters and trim the network, and the accuracy of the pruned network is often worse than the original network. The reason is that they use linear representation

Table 2
Comparisons with network pruning methods for optimizing ResNet-56 on CIFAR-10. The "FLOPs ↓" is the pruned ratio on FLOPs. The fields have the same meaning as Table 1.

Method	Base Acc. (%)	Pruned Acc. (%)	Acc. Drop (%)	FLOPs (M)	FLOPs ↓ (%)
Polar [22]	93.80	93.83	−0.03	–	47.0
HRank [5]	93.26	93.17	0.09	62.7	50.0
Greg-1 [1]	93.36	93.06 (±0.09)	0.30	62.0	50.2
LFPC [4]	93.59	93.24 (±0.17)	0.35	59.1	52.9
ResRep [23]	93.71	93.71 (±0.02)	0.00	59.1	52.9
DMC [24]	93.62	93.69	−0.07	62.7	50.0
SNAP [25]	93.49	93.81	−0.32	60.2	52.0
GNN-RL [26]	93.49	93.59	−0.10	57.7	54.0
DIEF [9]	92.86	92.94	−0.08	–	49.5
AASC [8]	93.05	93.02	0.03	–	53.2
Ours (0.40)	93.95	**94.82** (±0.14)	**−0.87**	63.1	50.2

Bold values indicate the best performance in each column.

Table 3
Comparisons with pruning methods for optimizing ResNet-56 on CIFAR-100. The fields have the same meaning as Table 2.

Method	Base Acc. (%)	Pruned Acc. (%)	Acc. Drop (%)	FLOPs (M)	FLOPs ↓ (%)
Polar [22]	72.49	72.46	0.06	–	25.0
OICSR-GL [27]	75.87	76.23	−0.66	–	38.5
LFPC [4]	71.41	70.83	0.58	60.8	51.6
FPGM [28]	71.41	69.66	1.75	59.4	52.6
Ours (0.80)	70.79	73.86 (±0.14)	**−3.08**	86.8	31.8
Ours (0.40)	70.79	73.00 (±0.20)	**−2.20**	56.9	55.3

Bold values indicate the best performance in each column.

Table 4
Comparisons with pruning methods for optimizing ResNet-50 on ImageNet. * represents that the PARAMs are estimated from FLOPs.

Method	Top-1 Acc. (%)	Top-5 Acc. (%)	PARAMs (M)
ResNet50	76.15	92.87	25.56
HRank [5]	71.98	91.01	13.77
CURL [29]	73.39	91.46	6.67
AutoPruner [30]	74.16	91.25	12.60
DNAL [19]	74.07	92.02	15.34
Eproxy [31]	74.3	91.9	4.9
TE-NAS [32]	75.5	92.5	5.4
GReg-1 [33]	75.45	–	4.42
DIEF [9]	74.05	91.87	9.35*
DIEF [9]	72.95	91.10	7.44*
AASC [8]	74.73	–	10.83
AASC [8]	73.68	–	6.31
SEArch (ours)	**75.51**	**92.61**	**4.98**

Bold values indicate the best performance in each column.

to approximate the original network, which could lose accuracy during feature reduction.

Results on CIFAR-100. The quantitative comparisons of pruning ResNet-56 on CIFAR-100 are shown in Table 3. Note that the weights of the baseline models vary in these papers, we chose our baseline from a public repository for fair comparisons. Our method pruned the ResNet-56 by 31.8 % and 55.3 % FLOPs, the optimized networks improved better accuracy of 3.08 % and 2.20 % over the baseline model. While existing methods such as SFP He et al. [28] and Polar Zhuang et al. [22] have 1.75 % and 0.06 % accuracy dropped from the baseline model. Fig. 2 visualizes how the proposed SEArch framework optimizes the architecture of the student model (0.40 M).

Results on ImageNet. We also conducted experiments for ResNet-50 optimization on ImageNet. The results are shown in Table 4. We follow the mobile setting and set our maximum parameter count to 5.0 M. Compared with the recent optimization methods, our method optimized the baseline network to an extremely small size (from 25.56 M to 5.0 M, 20 % size of the original network), while preserving comparable Top-1 and Top-5 accuracy.

4.3. Comparisons with knowledge distillation methods

Following the common settings in KD papers, we conducted experiments on CIFAR-10 dataset. We selected ResNet-56 as the teacher model and transferred the knowledge to a smaller network. Quantitative comparisons are reported in Table 5. State-of-the-art KD papers selected ResNet-20 as the student model and the architecture optimization is not considered during knowledge transfer. In contrast, our method integrates the architecture optimization technique to search for the optimal architecture for the student network. Because ResNet-20 has 0.27 M parameters, we set the target parameters of the searched network to 0.27 M for fair comparisons. Under the same parameter count, the optimized architecture by our SEArch has the best accuracy of 93.58 %, which outperforms the existing KD papers. Both KD methods and ours require training on the dataset to transfer the knowledge, our method requires longer time to optimize the architecture (see Appendix C for runtime analysis) but achieves better accuracy. We reported the results of SSKD and HSAKD from [41] Table 2

Table 5
Comparisons with KD methods on CIFAR-10 The teacher model is ResNet-56 and has 0.85 M parameters. Experiments in existing KD papers used ResNet-20 which has 0.27 M parameters and the architecture is fixed. Our method SEArch optimized the network architecture and its parameter count is also 0.27 M. Acc. is the accuracy of the smaller network in percentage (%).

Method	Acc. (%)
KD [10]	92.63
AT [35]	92.87
FT [36]	93.15
OD [37]	93.19
Tf-KD(S) [38]	92.59
CRD [39]	93.20
IE-AT [40]	93.30
IE-FT [40]	93.43
IE-OD [40]	93.47
SSKD [41]	92.73
HSAKD [41]	93.46
Ours (0.27 M PARAMs)	**93.58**

Bold values indicate the best performance in each column.

4.4. Comparisons with neural architecture search

We compared SEArch with the representative NAS methods in Table 6. Generally, our SEArch can obtain the network architecture much faster than NAS methods because of the well-designed and smaller search space. NAS methods find an optimal network from a big super-network, while our method constructs the optimal one from a primitive network with the guidance of teacher model, which is much more efficient than NAS methods. Networks generated using these NAS methods all have relatively big complexity and parameters, e.g., with about 3.3 million parameters. A most recent NAS method, BaLeNAS [45], took 14.4 GPU hours to search for a network with 3.8 million parameters and 97.5 % accuracy. In contrast, our method used only 4.0 GPU hours to converge, and yielded a network with 94.01 % accuracy with only 0.40 million parameters. The search cost of SEO is the lowest and the optimized network has comparable performance with smallest model size.

4.5. Experiments on NeRF

To demonstrate the generalizability of our search method, we also apply the method to the Multilayer Perceptron(MLP) in NeRF [46]. We used the pre-trained NeRF MLP as the teacher model and defined each layer in the MLP as a node. Similar to approaches in ResNet, we defined the distillation loss as the difference in output between corresponding nodes. By using distillation loss as guidance, we can identify the layers

Table 6
Quantitative comparisons with neural architecture search methods on CIFAR-10. Arch Size represents the number of parameters in millions. Search Cost is measured by a single NVIDIA 1080Ti GPU runtime in hours. Acc. is the test accuracy in percentage.

Method	PARAMs (M)	Search cost	Accuracy (%)
NASNet-A [42]	3.3	48,000	97.35
PC-DARTS [43]	3.6	7.2	97.43
DrNAS [44]	4.0	9.6	97.46
BaLeNAS [45]	3.8	14.4	**97.50**
DARTS [3]	3.3	36	97.00
DARTS* [3]	0.4	5.0	93.00
SEArch (ours)	**0.4**	**4.0**	94.01

Bold values indicate the best performance in each column.

Table 7
Results of our model on NeRF. Model size represents the parameter size compared to the original baseline NeRF model. naïve method corresponds to randomly retaining 50 % of neurons in MLP.

Method	Model size (%)	PSNR
Original NeRF	100 %	28.03
Naïve method	50 %	25.48
SEArch (ours)	50 %	27.08

where the student model performed the worst and subsequently increase the channel width in those layers. We started with 8 layers and 32 width on each layer, and increased the width by 32 each time we evolved. The optimized architecture for the Lego Scene emerged as an 8-layer MLP with channel widths of $[32, 32, 32, 64, 96, 256, 256, 256]$.

We evaluated our results on the Lego scene of the Blender synthetic dataset. Our SEArch-NeRF is built on NeRF-Pytorch [47]. Due to computational resource limitations, we opted for a smaller model and reduced the *number of samples per ray*. During the training period, the student model was trained for 200,000 iterations using the Adam optimizer with a learning rate of 5×10^{-4}. We compared the performance using PSNR.

As seen in Table 7, our pruning method achieved a 6 % higher PSNR than the naïve random selection method. Additionally, our method's performance is only 3.5 % lower than the full-sized baseline, while utilizing only 50 % of the parameters.

5. Conclusions

We present a self-evolving neural network optimization framework that combines the strengths of network optimization methods (i.e., network pruning, KD, and NAS). Starting from a basic structure, our student model iteratively identifies bottlenecks in the network and refines its architecture under the guidance of the teacher network. Our adoption of a single operation and the design of edge splitting enable more efficient local structure creation and topology modification. Experiments on image classification demonstrate the effectiveness of our approach.

Limitations and future work. The proposed framework optimizes an existing network in a manner similar to network pruning and KD. When compared to NAS, our approach achieves faster optimization, albeit with a trade-off in accuracy. Since the pruning guidance mainly comes from the teacher model, the student model might be constrained by the teacher model. And we plan to explore new metrics based on information gain theory for better guidance in autonomous network evolution.

Our framework also provides inspiration for enhancing recent transformer-based vision models like Vision Transformers (ViT) [48]. In these models, our approach could be adapted to assess and evolve each transformer layer. By measuring the distance of each layer from the teacher model, we can identify the most vulnerable layers and reinforce them. Future experiments will focus on the selection of metrics and strategies, which will further explore the potential of our evolving strategy in transformer-based models.

Acknowledgement

This work was partly supported by U.S. National Science Foundation CBET 2115405.

CRediT authorship contribution statement

Yongqing Liang: Writing – original draft, Software, Investigation, Conceptualization, Visualization, Methodology, Writing – review & editing, Data curation. **Dawei Xiang:** Visualization, Software, Data curation, Validation, Methodology, Writing – original draft. **Xin Li:** Formal analysis, Writing – review & editing, Project administration, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Single convolution expressivity analysis

We observed that the operations of DARTS Liu et al. [3] can be replaced using a single convolutional operation such as the 3×3 separable convolution. First, 3×3 average pooling, identity (skip connection), and *zero* are in the same group, which are special cases of the 3×3 convolution with specific weights. Second, following ResNet, 3×3 max pooling can be replaced by a 3×3 convolution with $stride = 2$. Third, 5×5 or 7×7 convolutions can be approximated by stacked 3×3 convolutions. Last, dilated separable convolutions are special cases of separable convolutions with larger kernel sizes and zero parameter weights. As results, 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable

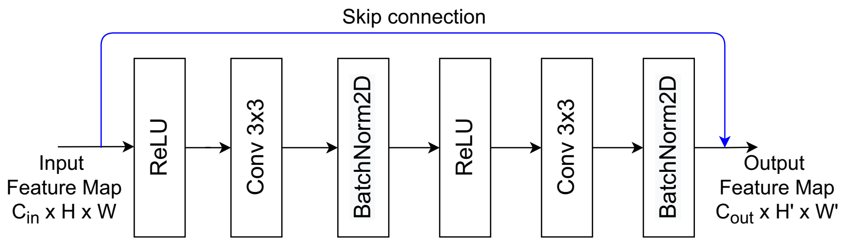

Skip connection
Input Feature Map C_in x H x W
ReLU
Conv 3x3
BatchNorm2D
ReLU
Conv 3x3
BatchNorm2D
Output Feature Map C_out x H' x W'

Fig. A.5. Conventional convolution unit stacking two full convolutions. ResNet improves its performance by adding a skip connection (blue line).

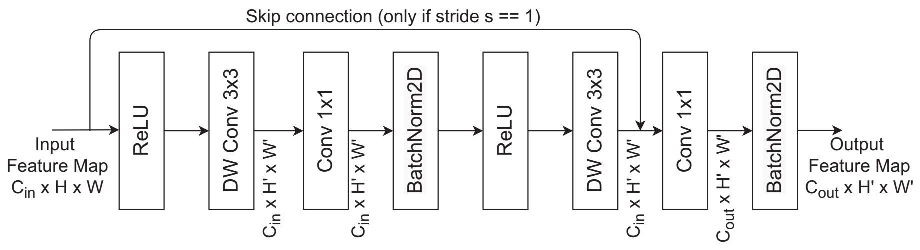


Fig. A.6. Convolution unit of the residual separable convolution 3×3, named sep_conv_3x3. DW Conv is the standard depth-wise separable convolution. C_{in} and C_{out} represent the feature channels of the input and output feature maps. The stride s equals either 1 or 2, which controls spatial size of the output feature map, $H' = H/s$ and $W' = W/s$.

convolutions can be induced to stacked 3×3 separable convolutions. Hence, using just a single operation greatly reduces the computational cost of operation search, yet still offers similar model expressivity.

Appendix B. Implementation details

We implemented our framework on PyTorch. The convolutional operation we chose is the 3×3 residual separable convolution, or sep_conv_3x3 for short. Fig. A.5 shows the conventional convolutional unit stacking two full convolutions (named Conv 3x3). ResNet improves its performance by adding a skip connection (indicated by the blue line). MobileNet Sandler et al. [49] found that depth-wise separable convolution is more efficient than full convolutions. We follow their design and replace the full convolutions with two layers: (1) The first layer is a depth-wise convolution that performs lightweight filtering (named DW Conv 3x3). (2) The second layer is a point-wise convolution that computes a linear combination of the input channels (named Conv 1x1). Fig. A.6 illustrates the detailed structure of the 3×3 residual separable convolution. We tried other sizes of convolution or operations but found out that 3×3 Conv has the best performance because its parameter size is small and gives more space for the macro structure search.

The maximum stacked operations per edge B_{op} is set to 7. To speed up the search, on each edge, we initialize the number of operations as 1, then we append 3 operations once when an unsupervised node is added.

In the searching phase, we train 10 epochs for each training stage, which is usually good enough to reveal the underperforming nodes in the current student model. We stop the self-evolving procedure when the size of the student model reaches the limit. Following the training settings of existing papers on CIFAR-10, the final student model is retrained for 400 epochs. We choose SGD with a momentum of 0.9 and a weight decay of 0.0003 in all the training. The initial learning rate is set to 0.025 and the step scheduler is used to tune the learning rate during training. On ImageNet, we choose learning rate 0.01, weight decay 0.001 and train the model for 500 epochs. We adopt a cyclical learning rate strategy after 300 epochs. We set the step size to be 30, and decrease the learning rate by 50 % until 0.0001 and then increase the learning rate by 100 % until 0.001 every 30 epochs. We didn't use data augmentation on all the training sets.

Appendix C. Runtime analysis

Experiments from other papers were performed on different hardware configurations, we found it very difficult to directly compare the runtime performance. Since the proposed SEArch framework optimizes architecture starting from a primitive one, when the target model size is much smaller than the baseline model size, our algorithm needs less time to prune the network. For example, recent network pruning literature TAS Dong and Yang [7] reported their runtime statistics. For the CIFAR-10 dataset, TAS searched a network with 92.65 % accuracy, and it cost 10.6 GPU hours on an NVIDIA V100 GPU. In contrast, our method SEArch searched a network with **94.01 % accuracy**, and it cost only **4.0 GPU hours** on an NVIDIA 1080Ti GPU, which reduced 62.3 % search time. Additionally, the training speed of V100 is 100 % faster than 1080Ti in general. Hence, **our proposed scheme can more efficiently optimize neural architectures**.

Comprehensive runtime analysis of our SEArch is reported in Table C.8. We ran our pipeline on a single NVIDIA 1080Ti GPU (11GB MEM). The running time depends on the baseline model size and target model size. As the network's depth and width increase, the computation time grows linearly. The most time-consuming step is the knowledge distillation from the teacher model to the student model, while the computation time for edge-splitting is negligible. Larger models require longer search/retraining time. Our model has small runtime and memory footprints. Note that **our SEArch framework is memory-friendly**: it can fit into a 4GB-MEM GPU when the batch size is set to 32.

Appendix D. Comparisons with network pruning on ResNet-110 on CIFAR-10

Recent network pruning methods also conducted experiments on optimizing ResNet-110 on CIFAR-10. The qualitative comparison is shown in Table D.9. Note that in this dataset, reducing the PARAMs to a

Table C.8
Runtime results under different pruning settings. The codes are implemented in PyTorch. The experiments are conducted on Ubuntu 18.04 with a single NVIDIA 1080Ti GPU. Our SEO model can optimize a network at low computational costs. (GPU hours/iters).

Dataset	CIFAR-100		CIFAR-10			
Baseline model	ResNet-56		ResNet-56		ResNet-110	
PARAMs (M)	0.40	0.80	0.40	0.80	0.68	0.86
Search time	3.7/24	12.7/38	4.0/25	18.3/47	11.8/38	13.5/44
Retraining time	8.7/400	19.0/400	10.1/400	22.4/400	15.6/400	17.2/400

Table D.9
Quantitative comparisons with network pruning methods for optimizing ResNet-110 on CIFAR-10. "Base Acc." and "Pruned Acc" are accuracy of the baseline and optimized networks. "Acc. ↓" is the accuracy drop (smaller is better), where a negative value means the resulting model outperforms the baseline. The "PARAMs ↓" is the pruned ratio of network parameters. The "FLOPs ↓" is the pruned ratio of FLOPs. The baseline of NetSlim is ResNet-164.

Method	Base Acc. (%)	Pruned Acc. (%)	Acc. ↓ (%)	PARAMs (M)	PARAMs ↓ (%)	FLOPs (M)	FLOPs ↓ (%)
NetSlim	94.58	94.73	−0.15	1.10	35.2	275	44.9
MIL	93.63	93.44	0.19	–	–	166	34.2
PFEC	93.53	93.30	−0.02	1.16	32.4	155	38.6
SFP	93.68	93.86 (±0.21)	−0.18	–	–	150	40.8
GAL	93.5	92.74	0.76	0.95	44.8	130.2	48.5
FPGM	93.68	93.74 (±0.10)	−0.16	–	–	121	52.3
TAS	94.97	94.33	0.64	–	–	119	53.0
HRank	93.50	93.36	0.14	0.70	59.2	105.7	58.2
LFPC	93.68	93.07 (±0.15)	0.61	–	–	101	60.3
ResRep	94.64	94.62 (±0.02)	0.02	–	–	–	58.2
LCAF	93.77	93.92	−0.15	–	60.1	–	59.8
Ours (0.86)	94.04	95.19 (±0.05)	**−1.15**	0.86	50.2	98.8	61.2
Ours (0.68)	94.04	95.00 (±0.02)	**−0.97**	0.68	60.4	85.5	66.4

Bold values indicate the best performance in each column.

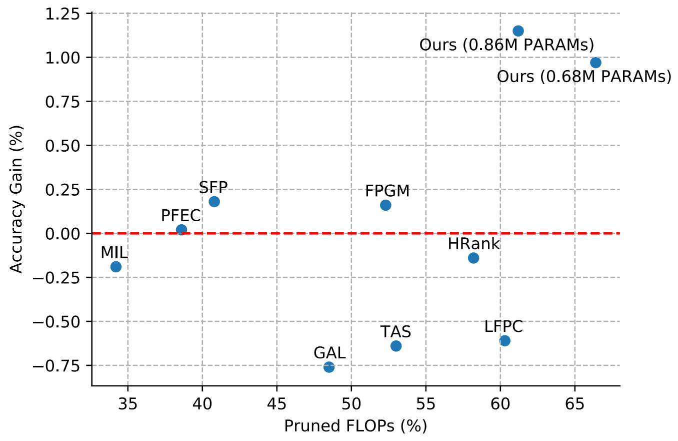


Fig. D.7. The accuracy gain of the optimized network over the optimized ratio on FLOPs. We compare the optimized networks from ResNet-110 on CIFAR-10. The accuracy gain means the accuracy improvement after pruning. Detailed comparisons are reported in Table D.9. The red horizontal line indicates the zero accuracy gain. Our optimized networks are on the right top positions that outperform the state-of-the-art network pruning methods.

half size is a common setting in pruning literature, and this corresponds to 0.86 M PARAMs. In addition, we also cut 60 % off FLOPs, which resolves to 0.68 M PARAMs, for a fair comparison with HRank [5]. Our method optimized the ResNet-110 by 61.2 % and 66.4 % FLOPs, the optimized networks improved the accuracy by 1.15 % and 0.97 % over the baseline model. Fig. D.7 plots the accuracy gain of the pruned network over the pruned ratio on FLOPs. Our method outperforms existing network pruning methods by achieving bigger accuracy gains and a higher pruned ratio on FLOPs.

Data availability

Data will be made available upon request.

References

[1] H. Wang, C. Qin, Y. Zhang, Y. Fu, Neural pruning via growing regularization, in: International Conference on Learning Representations, 2021, pp. 1–10, https://openreview.net/forum?id=o966_Is_nPA.
[2] R. Shao, W. Zhang, J. Wang, Conditional pseudo-supervised contrast for data-free knowledge distillation, Pattern Recognit. 143 (2023) 109781.
[3] H. Liu, K. Simonyan, Y. Yang, DARTS: differentiable architecture search, arXiv preprint arXiv:1806.09055, 2018.
[4] Y. He, Y. Ding, P. Liu, L. Zhu, H. Zhang, Y. Yang, Learning filter pruning criteria for deep convolutional neural networks acceleration, in: Proceedings of the IEEE/CVF Conference on CVPR, 2020, pp. 2009–2018.
[5] M. Lin, R. Ji, Y. Wang, Y. Zhang, B. Zhang, Y. Tian, L. Shao, HRank: filter pruning using high-rank feature map, in: Proceedings of the IEEE/CVF CVPR, 2020.
[6] Y. Hou, Z. Ma, C. Liu, Z. Wang, C.C. Loy, Network pruning via resource reallocation, Pattern Recognit. 145 (2024) 109886.
[7] X. Dong, Y. Yang, Network pruning via transformable architecture search, Adv. Neural Inf. Process. Syst. 32 (2019) 760–771.
[8] J. Liu, W. Liu, Y. Li, J. Hu, S. Cheng, W. Yang, Attention-based adaptive structured continuous sparse network pruning, Neurocomputing 590 (2024) 127698.
[9] Y. Zheng, P. Sun, Q. Ren, W. Xu, D. Zhu, A novel and efficient model pruning method for deep convolutional neural networks by evaluating the direct and indirect effects of filters, Neurocomputing 569 (2024) 127124.
[10] G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network, in: NIPS Deep Learning and Representation Learning Workshop, 2015, http://arxiv.org/abs/1503.02531.
[11] T.-B. Xu, P. Yang, X.-Y. Zhang, C.-L. Liu, Lightweightnet: toward fast and lightweight convolutional neural networks via architecture distillation, Pattern Recognit. 88 (2019) 272–284.
[12] Y. Cho, G. Ham, J.-H. Lee, D. Kim, Ambiguity-aware robust teacher (ART): enhanced self-knowledge distillation framework with pruned teacher network, Pattern Recognit. 140 (2023) 109541.
[13] J. Zhang, Y. Shi, J. Yang, Q. Guo, KD-SCFNet: towards more accurate and lightweight salient object detection via knowledge distillation, Neurocomputing 572 (2024) 127206.
[14] S. Lu, W. Zeng, X. Li, J. Ou, Adaptive lightweight network construction method for self-knowledge distillation, Neurocomputing 624 (2025) 129477.
[15] S. Lin, H. Xie, B. Wang, K. Yu, X. Chang, X. Liang, G. Wang, Knowledge distillation via the target-aware transformer, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 10915–10924.
[16] P. Dong, L. Li, Z. Wei, DisWOT: Student architecture search for distillation without training, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023, pp. 11898–11908.
[17] B. Ma, J. Zhang, Y. Xia, D. Tao, Inter-layer transition in neural architecture search, Pattern Recognit. 143 (2023) 109697.
[18] X. Qian, F. Liu, L. Jiao, X. Zhang, X. Huang, S. Li, P. Chen, X. Liu, Knowledge transfer evolutionary search for lightweight neural architecture with dynamic inference, Pattern Recognit. 143 (2023) 109790.
[19] Q. Guo, X.-J. Wu, J. Kittler, Z. Feng, Differentiable neural architecture learning for efficient neural networks, Pattern Recognit. 126 (2022) 108448.
[20] W. Wang, X. Zhang, H. Cui, H. Yin, Y. Zhang, FP-DARTS: fast parallel differentiable neural architecture search for image classification, Pattern Recognit. 136 (2023) 109193.
[21] A. Yang, P.M. Esperança, F.M. Carlucci, NAS evaluation is frustratingly hard, in: International Conference on Learning Representations, 2019.
[22] T. Zhuang, Z. Zhang, Y. Huang, X. Zeng, K. Shuang, X. Li, Neuron-level structured pruning using polarization regularizer, Adv. Neural Inf. Process. Syst. 33 (2020) 9865–9877.
[23] X. Ding, T. Hao, J. Tan, J. Liu, J. Han, Y. Guo, G. Ding, ResRep: lossless CNN pruning via decoupling remembering and forgetting, in: Proceedings of the IEEE/CVF ICCV, 2021, pp. 4510–4520.
[24] S. Gao, F. Huang, J. Pei, H. Huang, Discrete model compression with resource constraint for deep neural networks, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020.
[25] S. Gao, Z. Zhang, Y. Zhang, F. Huang, H. Huang, Structural alignment for network pruning through partial regularization, in: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2023, pp. 17402–17412.
[26] S. Yu, A. Mazaheri, A. Jannesari, Topology-aware network pruning using multi-stage graph embedding and reinforcement learning, in: K. Chaudhuri; S. Jegelka; L. Song; C. Szepesvari; G. Niu, S. Sabato, (Eds.), in: Proceedings of the 39th International Conference on Machine Learning, volume 162 of Proceedings of Machine Learning Research, PMLR, 2022, pp. 25656–25667, https://proceedings.mlr.press/v162/yu22e.html.
[27] J. Li, Q. Qi, J. Wang, C. Ge, Y. Li, Z. Yue, H. Sun, OICSR: out-in-channel sparsity regularization for compact deep neural networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 7046–7055.
[28] Y. He, P. Liu, Z. Wang, Z. Hu, Y. Yang, Filter pruning via geometric median for deep convolutional neural networks acceleration, in: Proceedings of the IEEE/CVF CVPR, 2019, pp. 4340–4349.
[29] J.-H. Luo, J. Wu, Neural network pruning with residual-connections and limited-data, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 1458–1467.
[30] J.-H. Luo, J. Wu, AutoPruner: an end-to-end trainable filter pruning method for efficient deep model inference, Pattern Recognit. 107 (2020) 107461.
[31] Y. Li, J. Li, C. Hao, P. Li, J. Xiong, D. Chen, Extensible and efficient proxy for neural architecture search, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2023, pp. 6199–6210.
[32] W. Chen, X. Gong, Z. Wang, Neural architecture search on ImageNet in four GPU hours: a theoretically inspired perspective, arXiv preprint arXiv:2102.11535, 2021.
[33] H. Wang, C. Qin, Y. Zhang, Y. Fu, Neural pruning via growing regularization, arXiv preprint arXiv:2012.09243, 2020.
[34] Chenyaofo, PyTorch CIFAR models, 2021, https://github.com/chenyaofo/pytorch-cifar-models.
[35] S. Zagoruyko, N. Komodakis, Paying more attention to attention: improving the performance of convolutional neural networks via attention transfer, arXiv preprint arXiv:1612.03928, 2016.
[36] J. Kim, S. Park, N. Kwak, Paraphrasing complex network: network compression via factor transfer, Adv. Neural Inf. Process. Syst. 31 (2018).
[37] B. Heo, J. Kim, S. Yun, H. Park, N. Kwak, J.Y. Choi, A comprehensive overhaul of feature distillation, in: Proceedings of the IEEE/CVF ICCV, 2019, pp. 1921–1930.
[38] L. Yuan, F.E. Tay, G. Li, T. Wang, J. Feng, Revisiting knowledge distillation via label smoothing regularization, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 3903–3911.
[39] Y. Tian, D. Krishnan, P. Isola, Contrastive representation distillation, arXiv preprint arXiv:1910.10699, 2019.
[40] Z. Huang, X. Shen, J. Xing, T. Liu, X. Tian, H. Li, B. Deng, J. Huang, X.-S. Hua, Revisiting knowledge distillation: an inheritance and exploration framework, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 3579–3588.
[41] W. Li, S. Shao, Z. Qiu, A. Song, Multi-perspective analysis on data augmentation in knowledge distillation, Neurocomputing 583 (2024) 127516.
[42] B. Zoph, V. Vasudevan, J. Shlens, Q.V. Le, Learning transferable architectures for scalable image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 8697–8710.
[43] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, H. Xiong, PC-DARTS: partial channel connections for memory-efficient architecture search, in: International Conference on Learning Representations, 2020, https://openreview.net/forum?id=BJlS634tPr.

[44] X. Chen, R. Wang, M. Cheng, X. Tang, C.-J. Hsieh, DrNAS: dirichlet neural architecture search, in: International Conference on Learning Representations, 2021, https://openreview.net/forum?id=9FWas6YbmB3.

[45] M. Zhang, S. Pan, X. Chang, S. Su, J. Hu, G.R. Haffari, B. Yang, BaLeNAS: differentiable architecture search via the bayesian learning rule, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 11871–11880.

[46] B. Mildenhall, P.P. Srinivasan, M. Tancik, J.T. Barron, R. Ramamoorthi, R. Ng, NeRF: representing scenes as neural radiance fields for view synthesis, Commun. ACM 65 (1) (2021) 99–106.

[47] L. Yen-Chen, NeRF-PyTorch, 2020, https://github.com/yenchenlin/nerf-pytorch/.

[48] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, N. Houlsby, An image is worth 16x16 words: transformers for image recognition at scale, in: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021, OpenReview.net, 2021, https://openreview.net/forum?id=YicbFdNTTy.

[49] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L.-C. Chen, MobileNetV2: inverted residuals and linear bottlenecks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.

Author biography

Yongqing Liang is a Ph.D. candidate at Texas A&M University. He received his B.Sc. degree from the School of Computer Science in Fudan University, and his M.Sc. degree from School of Electrical Engineering in Louisiana State University. His research interests include image and video content analysis, and 3D reconstruction and modeling. For more detail, see https://lyq.me/scholar.

Dawei Xiang received his Master of Science in Computer Science at Texas A&M University in 2024 with Grade 4.0. He received his Bachelor of Science in Statistics and Computer Science in Sun Yat-sen University in 2020 with Grade 3.9/4.0. His research interests is data analysis and machine learning.

Xin Li is a Professor at the Section of Visual Computing and Computational Media, College of Performance, Visualization, and Fine Arts, and a joint faculty member of Department of Computer Science and Engineering, at Texas A&M University. He got his B.S. degree in Computer Science at University of Science and Technology of China (USTC) in 2003, and Ph.D. in Computer Science from State University of New York at Stony Brook in 2008. His research areas are in visual computing, geometric modeling and processing, computer vision, and computer-aided design. For more detail, see https://people.tamu.edu/~xinli/.