GROWING EFFICIENT DEEP NETWORKS
BY STRUCTURED CONTINUOUS SPARSIFICATION

Xin Yuan
University of Chicago
yuanx@uchicago.edu

Pedro Savarese
TTI-Chicago
savarese@ttic.edu

Michael Maire
University of Chicago
mmaire@uchicago.edu

ABSTRACT

We develop an approach to growing deep network architectures over the course of training, driven by a principled combination of accuracy and sparsity objectives. Unlike existing pruning or architecture search techniques that operate on full-sized models or supernet architectures, our method can start from a small, simple seed architecture and dynamically grow and prune both layers and filters. By combining a continuous relaxation of discrete network structure optimization with a scheme for sampling sparse subnetworks, we produce compact, pruned networks, while also drastically reducing the computational expense of training. For example, we achieve 49.7% inference FLOPs and 47.4% training FLOPs savings compared to a baseline ResNet-50 on ImageNet, while maintaining 75.2% top-1 accuracy — all without any dedicated fine-tuning stage. Experiments across CIFAR, ImageNet, PASCAL VOC, and Penn Treebank, with convolutional networks for image classification and semantic segmentation, and recurrent networks for language modeling, demonstrate that we both train faster and produce more efficient networks than competing architecture pruning or search methods.

1 INTRODUCTION

Deep neural networks are the dominant approach to a variety of machine learning tasks, including image classification (Krizhevsky et al., 2012; Simonyan & Zisserman, 2015), object detection (Girshick, 2015; Liu et al., 2016), semantic segmentation (Long et al., 2015; Chen et al., 2017) and language modeling (Zaremba et al., 2014; Vaswani et al., 2017; Devlin et al., 2019). Modern neural networks are overparameterized and training larger networks usually yields improved generalization accuracy. Recent work (He et al., 2016; Zagoruyko & Komodakis, 2016; Huang et al., 2017) illustrates this trend through increasing depth and width of convolutional neural networks (CNNs). Yet, training is compute-intensive, and real-world deployments are often limited by parameter and compute budgets.

Neural architecture search (NAS) (Zoph & Le, 2017; Liu et al., 2019; Luo et al., 2018; Pham et al., 2018; Savarese & Maire, 2019) and model pruning (Han et al., 2016; 2015; Guo et al., 2016) methods aim to reduce these burdens. NAS addresses an issue that further compounds training cost: the enormous space of possible network architectures. While hand-tuning architectural details, such as the connection structure of convolutional layers, can improve performance (Iandola et al., 2016; Sifre & Mallat, 2014; Chollet, 2017; Howard et al., 2017; Zhang et al., 2018; Huang et al., 2018), a principled way of deriving such designs remains elusive. NAS methods aim to automate exploration of possible architectures, producing an efficient design for a target task under practical resource constraints. However, during training, most NAS methods operate on a large supernet architecture, which encompasses candidate components beyond those that are eventually selected for inclusion in the resulting network (Zoph & Le, 2017; Liu et al., 2019; Luo et al., 2018; Pham et al., 2018; Savarese & Maire, 2019). Consequently, NAS-based training may typically be more thorough, but more computationally expensive, than training a single hand-designed architecture.

Model pruning techniques similarly focus on improving the resource efficiency of neural networks during inference, at the possible expense of increased training cost. Common strategies aim to generate a lighter version of a given network architecture by removing individual weights (Han et al., 2015; 2016; Molchanov et al., 2017) or structured parameter sets (Li et al., 2017; He et al., 2018; Luo et al., 2017). However, the majority of these methods train a full-sized model prior to pruning and,
Figure 1: Growing Networks during Training. We define an architecture configuration space and simultaneously adapt network structure and weights. (a) Applying our approach to CNNs, we maintain auxiliary variables that determine how to grow and prune both filters (i.e. channel-wise) and layers, subject to practical resource constraints. (b) By starting with a small network and growing its size, we utilize fewer resources in early training epochs, compared to pruning or NAS methods. (c) Consequently, our method significantly reduces the total computational cost of training, while delivering trained networks of comparable or better size and accuracy.

We take a unified view of pruning and architecture search, regarding both as acting on a configuration space, and propose a method to dynamically grow deep networks by continuously reconfiguring their architecture during training. Our approach not only produces models with efficient inference characteristics, but also reduces the computational cost of training; see Figure 1. Rather than starting with a full-sized network or a supernet, we start from simple seed networks and progressively adjust (grown and prune) them. Specifically, we parameterize an architectural configuration space with indicator variables governing addition or removal of structural components. Figure 2(a) shows an example, in the form of a two-level configuration space for CNN layers and filters. We enable learning of indicator values (and thereby, architectural structure) via combining a continuous relaxation with binary sampling, as illustrated in Figure 2(b). A per-component temperature parameter ensures that long-lived structures are eventually baked into the network’s discrete architectural configuration.

While the recently proposed AutoGrow (Wen et al., 2020) also seeks to grow networks over the course of training, our technical approach differs substantially and leads to significant practical advantages. At a technical level, AutoGrow implements an architecture search procedure over a predefined modular structure, subject to hand-crafted, accuracy-driven growing and stopping policies. In contrast, we parameterize architectural configurations and utilize stochastic gradient descent to learn the auxiliary variables that specify structural components, while simultaneously training the weights within those components. Our unique technical approach yields the following advantages:

- **Fast Training by Growing**: Training is a unified procedure, from which one can request a network structure and associated weights at any time. Unlike AutoGrow and the majority of pruning techniques, fine-tuning to optimize weights in a discovered architecture is optional. We achieve excellent results even without any fine-tuning stage.

- **Principled Approach via Learning by Continuation + Sampling**: We formulate our approach in the spirit of learning by continuation methods, which relax a discrete optimization problem to an increasingly stiff continuous approximation. Critically, we introduce an additional sampling step to this strategy. From this combination, we gain the flexibility of exploring a supernet architecture, but the computational efficiency of only actually training a much smaller active subnetwork.

- **Budget-Aware Optimization Objectives**: The parameters governing our architectural configuration are themselves updated via gradient decent. We have flexibility to formulate a variety of resource-sensitive losses, such as counting total FLOPs, in terms of these parameters.

- **Broad Applicability**: Though we use progressive growth of CNNs in width and depth as a motivating example, our technique applies to virtually any neural architecture. One has flexibility in how to parameterize the architecture configuration space. We also show results with LSTMs.

We demonstrate these advantages while comparing to recent NAS and pruning methods through extensive experiments on classification, semantic segmentation, and word-level language modeling.
Optimization with Structured Continuation

In structured methods, pruning is applied at the level of neurons, channels, or even layers. For example, L1-pruning (Li et al., 2017) removes channels based on the norm of their filters. He et al. (2018) use group sparsity to smooth the pruning process after training. MorphNet (Gordon et al., 2018) works on LSTMs (Hochreiter & Schmidhuber, 1997) by collectively removing the columns and rows of the weight matrices via group LASSO. Although structured pruning methods and our algorithm share the same spirit of generating efficient models, we gain training cost savings by growing networks from small initial architectures instead of pruning full-sized ones.

**Neural Architecture Search.** NAS methods have greatly improved the performance achieved by small network models. Pioneering NAS approaches use reinforcement learning (Zoph et al., 2018; Zoph & Le, 2017) and genetic algorithms (Real et al., 2019; Xie & Yuille, 2017) to search for transferable network blocks whose performance surpasses many manually designed ones. However, such approaches require massive computation during the search — typically thousands of GPU days. To reduce computational cost, recent efforts utilize more efficient search techniques, such as direct gradient-based optimization (Liu et al., 2019; Luo et al., 2018; Pham et al., 2018; Tan et al., 2019; Cai et al., 2019; Wortsman et al., 2019). Nevertheless, most NAS methods perform search in a supernet space which requires more computation than training typically-sized architectures.

**Network Growing.** Network Morphism (Wei et al., 2016) searches for efficient deep networks by extending layers while preserving the parameters. Recently proposed Autogrow (Wen et al., 2020) takes an AutoML approach to growing layers. These methods either require a specially-crafted policy to stop growth (e.g., after a fixed number of layers) or rely on evaluating accuracy during training, incurring significant additional computational cost.

**Learning by Continuation.** Continuation methods are commonly used to approximate intractable optimization problems by gradually increasing the difficulty of the underlying objective, for example by adopting gradual relaxations to binary problems. Wu et al. (2019); Xie et al. (2019b; 2020) use gumbel-softmax (Jang et al., 2017) to back-propagate errors during architecture search and spatial feature sparsification. Savarese et al. (2020) propose continuous sparsification to speed up pruning and ticket search (Frankle & Carbin, 2019). Despite the success of continuation methods in producing sparse networks upon the completion of training, they do not operate on sparse networks.
during training and instead work with a real-valued relaxation. Postponing actual elimination of near zeroed-out components prevents naive application of these methods from reducing training costs.

3 Method

3.1 Architectural Configuration Space

A network topology can be seen as a directed acyclic graph consisting of an ordered sequence of nodes. Each node $x_{in}^{(i)}$ is an input feature and each edge is a computation cell with structured hyperparameters (e.g., filter and layer numbers in convolutional networks). An architectural configuration space can be parameterized by associating a mask variable $m \in \{0, 1\}$ with each computation cell (edge), which enables training-time pruning ($m = 1 \to 0$) and growing ($m = 0 \to 1$) dynamics.

As a running example, we consider a two-level configuration space for CNN architectures, depicted in Figure 2(a), that enables dynamically growing networks in both width (channel-wise) and depth (layer-wise). Alternative configuration spaces are possible; we defer to the Appendix details on how we parameterize the design of LSTM architectures.

**CNN Channel Configuration Space:** For a convolutional layer with $l_{in}$ input channels, $l_{out}$ output channels (filters) and $k \times k$ sized kernels, the $i$-th output feature is computed based on the $i$-th filter, i.e. for $i \in \{1, \ldots, l_{out}\}$:

$$x_{out}^{(i)} = f(x_{in}; F^{(i)} \cdot m_{c}^{(i)}) ,$$

where $m_{c}^{(i)} \in \{0, 1\}$ is a binary parameter that removes the $i$-th output channel when set to zero and $f$ denotes the convolutional operation. $m_{c}^{(i)}$ is shared across a filter and broadcasts to the same shape as the filter tensor $F^{(i)}$, enabling growing/pruning of the entire filter. As Figure 2(a) (top) shows, we start from a slim channel configuration. We then query the indicator variables and perform state transitions: (1) When flipping an indicator variable from 0 to 1 for the first time, we grow a randomly initialized filter and concatenate it to the network. (2) If an indicator flips from 1 to 0, we temporarily detach the corresponding filter from the computational graph; it will be grown back to its original position if its indicator flips back to 1, or otherwise be permanently pruned at the end of training. (3) For other cases, the corresponding filters either survive and continue training or remain detached pending the next query to their indicators. Our method automates architecture evolution, provided we can train the indicators.

**CNN Layer Configuration Space:** To grow network depth, we design a layer configuration space in which an initial shallow network will progressively expand into a deep trained model, as shown in Figure 2(a) (bottom). Similar to channel configuration space, where filters serve as basic structural units, we require a unified formulation to support the growing of popular networks with shortcut connections (e.g., ResNets) and without (e.g., VGG-like plain nets). We first introduce an abstract layer class $f_{layer}$ as a basic structural unit, which operates on input features $x_{in}$ and generates output features $x_{out}$. $f_{layer}$ can be instantiated as convolutional layers for plain nets or residual blocks for ResNets, respectively. We define the layer configuration space as:

$$x_{out} = g(x_{in}; f_{layer} \cdot m_{i}^{(j)}) = \begin{cases} f_{layer}(x_{in}), & \text{if } m_{i}^{(j)} = 1 \\ x_{in}, & \text{if } m_{i}^{(j)} = 0 \end{cases}$$

where $m_{i}^{(j)} \in \{0, 1\}$ is the binary indicator for $j$-th layer $f_{layer}$, with which we perform state transitions analogous to the channel configuration space. Layer indicators have priority over channel indicators: if $m_{i}^{(j)}$ is set as 0, all filters contained in the corresponding layer will be detached, regardless of the state their indicators. We do not detach layers that perform changes in resolution (e.g., strided convolution).

3.2 Growing with Structured Continuous Sparseification

We can optimize a trade-off between accuracy and structured sparsity by considering the objective:

$$\min_{w, m_{c, i, f_{layer}}} L_{E}(g(f(x; w \odot m_{c}); f_{layer} \cdot m_{i})) + \lambda_{1} \parallel m_{c} \parallel_{0} + \lambda_{2} \parallel m_{i} \parallel_{0} ,$$

Published as a conference paper at ICLR 2021
where \( f \) is the operation in Eq. (1) or Eq. (9) (in Appendix A.6), while \( g \) is defined in Eq. (2). \( w \odot m_c \) and \( f_{layer} \cdot m_l \) are general expressions of structured sparsified filters and layers and \( L_E \) denotes a loss function (e.g., cross-entropy loss for classification). The \( \ell_0 \) terms encourage sparsity, while \( \lambda_{1,2} \) are trade-off parameters between \( L_E \) and the \( \ell_0 \) penalties.

**Budget-aware Growing.** In practice, utilizing Eq. (3) might require a grid search on \( \lambda_1 \) and \( \lambda_2 \) until a network with desired sparsity is produced. To avoid such a costly procedure, we propose a budget-aware growing process, guided by a target budget in terms of model parameters or FLOPs. Instead of treating \( \lambda_1 \) and \( \lambda_2 \) as constants, we periodically update them as:

\[
\lambda_1 \leftarrow \lambda_{1, base} \cdot \Delta u, \quad \lambda_2 \leftarrow \lambda_{2, base} \cdot \Delta u, \quad (4)
\]

where \( \Delta u \) is calculated as the target sparsity \( u \) minus current network sparsity \( u_G \), and \( \lambda_{1, base} \), \( \lambda_{2, base} \) are initial base constants. In early growing stages, since the network is too sparse and \( \Delta u \) is negative, the optimizer will drive the network towards a state with more capacity (wider/deeper). The regularization effect gradually weakens as the network’s sparsity approaches the budget (and \( \Delta u \) approaches zero). This allows us to adaptively grow the network and automatically adjust its sparsity level while simultaneously training model weights. Appendix A.1 provides more detailed analysis. Our experiments default to defining budget by parameter count, but also investigate alternative notions of budget.

**Learning by Continuation.** Another issue in optimizing Eq. (3) is that \( \|m_c\|_0 \) and \( \|m_l\|_0 \) make the problem computationally intractable due to the combinatorial nature of binary states. To make the configuration space continuous and the optimization feasible, we borrow the concept of learning by continuation (Cao et al., 2017; Wu et al., 2019; Savarese et al., 2020; Xie et al., 2020). We reparameterize \( m \) as the binary sign of a continuous variable \( s \): \( \text{sign}(s) \) is 1 if \( s > 0 \) and 0 if \( s < 0 \). We rewrite the objective in Eq. (3) as:

\[
\min_{w, \sigma, \beta \neq 0, f_{layer}} \quad L_E \left( \beta \left( f(x; w \odot \text{sign}(s_c)); f_{layer} \cdot \text{sign}(s_l) \right) \right) + \lambda_1 \|\text{sign}(s_c)\|_1 + \lambda_2 \|\text{sign}(s_l)\|_1 \quad (5)
\]

We attack the hard and discontinuous optimization problem in Eq. (5) by starting with an easier objective which becomes harder as training proceeds. We use a sequence of functions whose limit is the sign operation: for any \( s \neq 0 \), \( \lim_{\beta \to \infty} \sigma(\beta s) = \text{sign}(s) \) if \( \sigma \) is sigmoid function or \( \lim_{\beta \to 0} \sigma(\beta s) = \text{sign}(s) \) if \( \sigma \) is gumbel-softmax \( \frac{\exp((\text{logit}(s_i) + g_{0,1}(s_i))/\beta)}{\sum_{j \in \{0,1\}} \exp((\text{logit}(s_i) + g_{0,1}(s_i))/\beta)} \) (Jang et al., 2017), where \( \beta > 0 \) is a temperature parameter and \( g_{0,1} \) is gumbel. By periodically changing \( \beta \), \( \sigma(\beta s) \) becomes harder to optimize, while the objectives converges to original discrete one.

**Maintaining Any-time Sparsification.** Although continuation methods can make the optimization feasible, they only conduct sparsification via a thresholding criterion in the inference phase. In this case, the train-time architecture is dense and not appropriate in the context of growing a network. To effectively reduce computational cost of training, we maintain a sparse architecture by introducing an 0-1 sampled auxiliary variable \( q \) based on the probability value \( \sigma(\beta s) \). Our final objective becomes:

\[
\min_{w, \sigma, \beta \neq 0, f_{layer}} \quad L_E \left( \beta \left( f(x; w \odot \sigma(\beta s_c) \odot q_c); f_{layer} \cdot \sigma(\beta s_l) \cdot q_l \right) \right) + \lambda_1 \|\sigma(\beta s_c)\|_1 + \lambda_2 \|\sigma(\beta s_l)\|_1 \quad (6)
\]

where \( q_c \) and \( q_l \) are random variables sampled from Bern(\( \sigma(\beta s_c) \)) and Bern(\( \sigma(\beta s_l) \)), which effectively maintains any-time sparsification and avoids sub-optimal thresholding, as shown in Figure 2(b).

**Improved Temperature Scheduler.** In existing continuation methods, the initial \( \beta \) value is usually set as \( \beta_0 = 1 \) and a scheduler is used at the end of each training epoch to update \( \beta \) in all activation functions \( \sigma \), typically following \( \beta = \beta_0 \cdot \gamma^t \), where \( t \) is current epoch and \( \gamma \) is a hyperparameter (> 1 when \( \sigma \) is the sigmoid function, < 1 when \( \sigma \) is gumbel softmax). Both \( \gamma \) and \( t \) control the speed
Table 1: Comparison with the channel pruning methods L1-Pruning (Li et al., 2017), SoftNet (He et al., 2018), ThiNet (Luo et al., 2017), Provable (Liebenwein et al., 2020) and BAR (Lemaire et al., 2019) on CIFAR-10.

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>Val Acc(%)</th>
<th>Params(M)</th>
<th>FLOPs(%)</th>
<th>Train-Cost Savings(×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16</td>
<td>Original</td>
<td>92.9 ± 0.16 (-0.0)</td>
<td>14.99 (100%)</td>
<td>100</td>
<td>1.0×</td>
</tr>
<tr>
<td></td>
<td>L1-Pruning</td>
<td>91.8 ± 0.12 (-1.1)</td>
<td>2.98 (19.9%)</td>
<td>19.9</td>
<td>2.5×</td>
</tr>
<tr>
<td></td>
<td>SoftNet</td>
<td>92.1 ± 0.09 (-0.8)</td>
<td>5.40 (36.0%)</td>
<td>36.1</td>
<td>1.6×</td>
</tr>
<tr>
<td></td>
<td>ThiNet</td>
<td>90.8 ± 0.11 (-2.1)</td>
<td>5.40 (36.0%)</td>
<td>36.1</td>
<td>1.6×</td>
</tr>
<tr>
<td></td>
<td>Provable</td>
<td>92.4 ± 0.12 (-0.5)</td>
<td>0.85 (5.7%)</td>
<td>15.0</td>
<td>3.5×</td>
</tr>
<tr>
<td>Ours</td>
<td>92.20 ± 0.10 (-0.4)</td>
<td>0.754 ± 0.005 (75.0%)</td>
<td>13.55 ± 0.03</td>
<td>4.95 ± 0.17 ×</td>
<td></td>
</tr>
<tr>
<td>ResNet-20</td>
<td>Original</td>
<td>91.3 ± 0.12 (-0.0)</td>
<td>0.27 (100%)</td>
<td>100</td>
<td>1.0×</td>
</tr>
<tr>
<td></td>
<td>L1-Pruning</td>
<td>90.9 ± 0.10 (-0.4)</td>
<td>0.15 (55.6%)</td>
<td>55.4</td>
<td>1.1×</td>
</tr>
<tr>
<td></td>
<td>SoftNet</td>
<td>90.8 ± 0.13 (-0.5)</td>
<td>0.14 (53.6%)</td>
<td>50.6</td>
<td>1.2×</td>
</tr>
<tr>
<td></td>
<td>ThiNet</td>
<td>89.2 ± 0.18 (-2.1)</td>
<td>0.18 (67.1%)</td>
<td>67.3</td>
<td>1.1×</td>
</tr>
<tr>
<td></td>
<td>Provable</td>
<td>90.8 ± 0.08 (-0.5)</td>
<td>0.10 (37.3%)</td>
<td>54.5</td>
<td>1.7×</td>
</tr>
<tr>
<td>Ours</td>
<td>90.91 ± 0.07 (-0.4)</td>
<td>0.096 ± 0.002 (35.8%)</td>
<td>50.20 ± 0.01</td>
<td>2.40 ± 0.09 ×</td>
<td></td>
</tr>
<tr>
<td>WRN-28</td>
<td>Original</td>
<td>96.2 ± 0.10 (-0.0)</td>
<td>36.5 (100%)</td>
<td>100</td>
<td>1.0×</td>
</tr>
<tr>
<td></td>
<td>L1-Pruning</td>
<td>95.2 ± 0.10 (-1.0)</td>
<td>7.6 (20.8%)</td>
<td>49.5</td>
<td>1.5×</td>
</tr>
<tr>
<td></td>
<td>BAR(16x V)</td>
<td>92.0 ± 0.08 (-2.0)</td>
<td>2.3 (6.3%)</td>
<td>1.5</td>
<td>2.6×</td>
</tr>
<tr>
<td>Ours</td>
<td>95.32 ± 0.11 (-0.9)</td>
<td>3.443 ± 0.010 (9.3%)</td>
<td>28.25 ± 0.04</td>
<td>3.12 ± 0.11 ×</td>
<td></td>
</tr>
</tbody>
</table>

at which the temperature increases during training. Continuation methods with global temperature schedulers have been successfully applied in pruning and NAS. However, in our case, a global schedule leads to unbalanced dynamics between variables with low and high sampling probabilities: increasing the temperature of those less sampled at early stages may hinder their training altogether, as towards the end of training the optimization difficulty is higher. To overcome this issue, we propose a separate, structure-wise temperature scheduler by making a simple modification: for each mask variable, instead of using the current epoch number $t$, we set a separate, structure-wise temperature scheduler by making a simple modification: for each mask $s$ which is increased only when its associated indicator variable is sampled as 1 in Eq. (6).

$$T = T_0 \cdot \beta^t,$$

where $t$ are vectors associated with the $\sigma$ functions. Experiments use this separate scheduler by default, but also compare the two alternatives. Algorithm 1 summarizes our optimization procedure.

4 Experiments

We evaluate our method against existing channel pruning, network growing, and neural architecture search (NAS) methods on: CIFAR-10 (Krizhevsky et al., 2014) and ImageNet (Deng et al., 2009) for image classification, PASCAL (Everingham et al., 2015) for semantic segmentation and the Penn Treebank (PTB) (Marcus et al., 1993) for language modeling. See dataset details in Appendix A.2. In tables, best results are highlighted in bold and second best are underlined.

4.1 Comparing with Channel Pruning Methods

Implementation Details. For fair comparison, we only grow filters while keeping other structured parameters of the network (number of layers/blocks) the same as unpruned baseline models. Our method involves two kinds of trainable variables: model weights and mask weights. For model weights, we adopt the same hyperparameters used to train the corresponding unpruned baseline models, except for setting the dropout keep probability for language modeling to 0.65. We initialize mask weights such that a single filter is activated in each layer. We train with SGD, an initial learning rate of 0.1, weight decay of $10^{-6}$ and momentum 0.9. Trade-off parameter $\lambda_1^{bas}$ is set to 0.5 on all tasks; $\lambda_2$ is not used since we do not perform layer growing here. We set $\sigma$ as the sigmoid function and $\gamma$ as the total number of epochs.

VGG-16, ResNet-20, and WideResNet-28-10 on CIFAR-10. Table 1 summarizes the models produced by our method and competing channel pruning approaches. Note that training cost is calculated based on overall FLOPS during pruning and growing stages. Our method produces sparser networks with less accuracy degradation, and, consistently saves more computation during training — a consequence of growing from a simple network. For an aggressively pruned WideResNet-28-10, we observe that BAR (Lemaire et al., 2019) might not have enough capacity to achieve negligible accuracy drop, even with knowledge distillation (Hinton et al., 2015) during training. Note that we
Table 2: Comparison with channel pruning methods: L1-Pruning (Li et al., 2017), SoftNet (He et al., 2018) and Provable (Liebenwein et al., 2020) on ImageNet.

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>Top-1 Acc(%)</th>
<th>Params(M)</th>
<th>FLOPs(%)</th>
<th>Train-Cost Savings(×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-50</td>
<td>Original</td>
<td>76.1 (-0.0)</td>
<td>23.0 (100%)</td>
<td>100</td>
<td>1.0(×)</td>
</tr>
<tr>
<td></td>
<td>L1-Pruning</td>
<td>74.7 (-1.4)</td>
<td>19.6 (85.2%)</td>
<td>77.5</td>
<td>1.1(×)</td>
</tr>
<tr>
<td></td>
<td>SoftNet</td>
<td>74.6 (-1.5)</td>
<td>N/A</td>
<td>58.2</td>
<td>1.2(×)</td>
</tr>
<tr>
<td></td>
<td>Provabe</td>
<td>75.2 (-0.9)</td>
<td>15.2 (65.9%)</td>
<td>70.0</td>
<td>1.2(×)</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>75.2 (-0.9)</td>
<td>14.1 (61.2%)</td>
<td>50.3</td>
<td>1.9(×)</td>
</tr>
</tbody>
</table>

Table 3: Results comparing with AutoGrow (Wen et al., 2020) on CIFAR-10 and ImageNet.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Methods</th>
<th>Variants</th>
<th>Found Net</th>
<th>Val Acc(%)</th>
<th>Depth</th>
<th>Sparse Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>Ours</td>
<td>Basic3ResNet</td>
<td>23-29-31</td>
<td>94.50</td>
<td>83</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Plain3Net</td>
<td>11-14-19</td>
<td>90.99</td>
<td>44</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>AutoGrow</td>
<td>Basic3ResNet</td>
<td>42-42-42</td>
<td>94.27</td>
<td>126</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Plain3Net</td>
<td>23-22-22</td>
<td>90.82</td>
<td>67</td>
<td>✔</td>
</tr>
<tr>
<td>ImageNet</td>
<td>Ours</td>
<td>Bottleneck4ResNet</td>
<td>5-6-5-7</td>
<td>77.41</td>
<td>23</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Plain4Net</td>
<td>3-4-4-5</td>
<td>70.79</td>
<td>16</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>AutoGrow</td>
<td>Bottleneck4ResNet</td>
<td>6-7-3-9</td>
<td>77.33</td>
<td>25</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Plain4Net</td>
<td>5-5-5-4</td>
<td>70.54</td>
<td>19</td>
<td>✔</td>
</tr>
</tbody>
</table>

report our method’s performance as mean ± standard deviation, computed over 5 runs with different random seeds. The small observed variance shows that our method performs consistently across runs.

ResNet-50 and MobileNetV1 on ImageNet. To validate effectiveness on large-scale datasets, we grow, from scratch, filters of the widely used ResNet-50 on ImageNet; we do not fine-tune. Table 2 shows our results best those directly reported in papers of respective competing methods. Our approach achieves 49.7% inference and 47.4% training cost savings in terms of FLOPs while maintaining 75.2% top-1 accuracy, without any fine-tuning stage. Appendix A.4 shows our improvements on the challenging task of growing channels of an already compact MobileNetV1. In addition, Figure 3 shows the top-1 accuracy/FLOPs trade-offs for MobileNetV1 on ImageNet, demonstrating that our method dominates competing approaches.

Deeplab-v3-ResNet-101 on PASCAL VOC. Appendix A.5 provides semantic segmentation results.

2-Stacked-LSTMs on PTB: We detail the extensions to recurrent cells and compare our proposed method with ISS based on vanilla two-layer stacked LSTM in Appendix A.6. As shown in Table 8, our method finds more compact model structure with lower training cost, while achieving similar perplexity on both validation and test sets.

4.2 Comparing with AutoGrow

Implementation Details. We grow both filters and layers. We follow AutoGrow’s settings in exploring architectural variations that define our initial seed network, layer-wise configuration space and basic structural units for layer: Basic3ResNet, Bottleneck4ResNet, Plain3Net, Plain4Net. Different from the initialization of AutoGrow using full-sized filters in each layer, our channel-wise configuration space starts from single filter and expands simultaneously with layers. Appendix A.7 contains a detailed comparison of seed architectures. For training model weights, we adopt the hyperparameters of the ResNet or VGG models corresponding to initial seed variants. For layer-wise and channel-wise mask variables, we initialize the weights such that only a single filter in each layer and one basic unit in each stage (e.g., BasicBlock in Basic3ResNet) is active. We use SGD training with initial learning rate of 0.1, weight decay of 10^{-6} and momentum of 0.9 on all datasets. The learning rate scheduler is the same as for the corresponding model weights. Trade-off parameters \( \lambda^{\text{base}}_1 \) and \( \lambda^{\text{base}}_2 \) are set as 1.0 and 0.1 on all datasets. For fair comparison, we fine-tune our final models with 40 epochs and 20 epochs on CIFAR-10 and ImageNet, respectively.

Results on CIFAR-10 and ImageNet. Table 3 compares our results to those of AutoGrow. For all layer-wise growing variants across both datasets, our method consistently yields a better depth and width configuration than AutoGrow, in terms of accuracy and training/inference costs trade-off. Regarding the training time of Bottleneck4ResNet on ImageNet, AutoGrow requires 61.6 hours for the growing phase and 78.6 hours for fine-tuning on 4 TITAN V GPUs, while our method takes 48.2 and 31.3 hours, respectively. Our method offers 43% more train-time savings than AutoGrow. We not only require fewer training epochs, but also grow from a single filter to a relatively sparse network, while AutoGrow always keeps full-sized filter sets without any reallocation.
4.3 Comparing with NAS Methods

As a fair comparison with NAS methods involving search and re-training phases, we also divide our method into growing and training phases. Specifically, we grow layers and channels from the Bottleneck4ResNet seed architecture directly on ImageNet by setting $\lambda_{1}^{\text{base}} = 2.0, \lambda_{2}^{\text{base}} = 0.1$ and the parameter budget under 7M. Then we resume training the grown architecture and compare with existing NAS methods in terms of parameters, top-1 validation accuracy and V100 GPU hours required by the search or growing stages, as shown in Table 4. Note that DARTS (Liu et al., 2019) conducts search on CIFAR-10, then transfers to ImageNet instead of direct search. This is because DARTS operates on a supernet by including all the candidate paths and suffers from GPU memory explosion. In terms of epoch-wise FLOPs, results shown in Figure 1(c) are for training an equivalent of ResNet-20 on CIFAR-10 in comparison with DARTS and Provable channel pruning approach (Liebenwein et al., 2020). Also note that the EfficientNet-B0 architecture, included in Table 4, is generated by grid search in the MnasNet search space, thus having the same heavy search cost. To achieve the reported performance, EfficientNet-B0 utilizes additional squeeze-and-excitation (SE) (Hu et al., 2018) modules, AutoAugment (Cubuk et al., 2019), as well as much longer re-training epochs on ImageNet.

<table>
<thead>
<tr>
<th>Method</th>
<th>Params</th>
<th>Top-1</th>
<th>Search/Grow Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmoebaNet-A</td>
<td>5.1M</td>
<td>74.5%</td>
<td>76K GPU hours</td>
</tr>
<tr>
<td>MnasNet</td>
<td>4.4M</td>
<td>74.0%</td>
<td>40K GPU hours</td>
</tr>
<tr>
<td>EfficientNet-B0</td>
<td>5.3M</td>
<td>77.1% (+SE)</td>
<td>40K GPU hours</td>
</tr>
<tr>
<td>DARTS</td>
<td>4.7M</td>
<td>73.1%</td>
<td>N/A</td>
</tr>
<tr>
<td>ProxylessNet(GPU)</td>
<td>7.1M</td>
<td>75.1%</td>
<td>200 GPU hours</td>
</tr>
<tr>
<td>Ours</td>
<td>6.8M</td>
<td>74.3%</td>
<td>80 GPU hours</td>
</tr>
<tr>
<td>Ours</td>
<td>6.7M</td>
<td>74.8%</td>
<td>110 GPU hours</td>
</tr>
<tr>
<td>Ours</td>
<td>6.9M</td>
<td>75.1%</td>
<td>140 GPU hours</td>
</tr>
</tbody>
</table>

ProxylessNet still starts with an over-parameterized supernet, but applies a pruning-like search method by binarizing the architecture parameters and forcing only one path to be activated at search-time. This enables directly searching on ImageNet, achieving 200× more search cost savings than MnasNet. Contrasting with ProxylessNet, our method progressively adds filters and layers to simple seed architectures while maintaining sparsification, which leads to savings of not only epoch-wise computation but also memory consumption, enabling faster, larger-batch training. As such, we further save 45% of the GPU search hours, while achieving comparable accuracy-parameter trade-offs.

4.4 Analysis

Training Cost Savings. Figure 4 illustrates our sparsification dynamics, showing epoch-wise FLOPs while growing a ResNet-20. Appendix A.8 presents additional visualizations. Even with fully parallel GPU hardware, starting with few filters and layers in the network will ultimately save wall-clock time, as larger batch training (Goyal et al., 2017) can always be employed to fill the hardware.

Figure 5 shows validation accuracy, model complexity, and layer count while growing Basic3ResNet. Complexity is measured as the model parameters ratio of AutoGrow’s target model. At the end of 160 epochs, our method’s validation accuracy is 92.36%, which is higher than AutoGrow’s 84.65% at 360 epochs. We thus require fewer fine-tuning epochs to achieve a final 94.50% accuracy on CIFAR.

Budget-Aware Growing. In Figure 6, for ResNet-20 on CIFAR-10, we compare architectures obtained by (1) uniform pruning: a naive predefined pruning method that prunes the same percentage of channels in each layer, (2) ours: variants of our method by setting different model parameter sparsities as target budgets during growing, and (3) direct design: our grown architectures re-initialized with random weights and re-trained. In most budget settings, our growing method outperforms direct design and uniform pruning, demon-
stratifying higher parameter efficiency. Our method also appears to have positive effect in terms of regularization or optimization dynamics, which are lost if one attempts to directly train the final compact structure. Appendix A.9 investigates FLOPs-based budget targets.

**Comparing with Random Baseline.** In addition to the uniform pruning baseline in Figure 6, we also compare with a random sampling baseline to further separate the contribution of our configuration space and growing method, following the criterion in (Xie et al., 2019a; Li & Talwalkar, 2019; Yu et al., 2020; Radosavovic et al., 2019). Specifically, this random baseline replaces the procedure for sampling entries of \( q \) in Eq. 6. Instead of using sampling probabilities derived from the learned mask parameters \( s \), it samples with fixed probability. As shown in Table 5, our method consistently performs much better than this random baseline. These results, as well as the more sophisticated baselines in Figure 6, demonstrate the effectiveness of our growing and pruning approach.

**Temperature Scheduler.** We compare our structure-wise temperature control to a global one in channel growing experiments on CIFAR-10 using VGG-16, ResNet-20, and WideResNet-28-10. Table 1 results use our structure-wise scheduler. To achieve similar sparsity with the global scheduler, the corresponding models suffer accuracy drops of 1.4%, 0.6%, and 2.5%. With the global scheduler, optimization of mask variables stops early in training and the following epochs are equivalent to directly training a fixed compact network. This may force the network to be stuck with a suboptimal architecture. Appendix A.10 investigates learning rate and temperature schedule interactions.

## 5 Conclusions

We propose a simple yet effective method to grow efficient deep networks via structured continuous sparsification, which decreases the computational cost not only of inference but also of training. The method is simple to implement and quick to execute; it automates the network structure reallocation process under practical resource budgets. Application to widely used deep networks on a variety of tasks shows that our method consistently generates models with better accuracy-efficiency trade-offs than competing methods, while achieving considerable training cost savings.

**Acknowledgments.** This work was supported by the University of Chicago CERES Center for Unstopable Computing and the National Science Foundation under grant CNS-1956180.
REFERENCES


Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In ICML, 2018.


A Appendix

A.1 More Detailed Analysis for Budget-aware Growing

Conducting grid search on trade-off parameters $\lambda_1$ and $\lambda_2$ is prohibitively laborious and time-consuming. For example, to grow an efficient network on CIFAR-10, one needs to repeat many times a run of 160-epochs training, and then pick the best model from all grown candidates. To avoid this tedious iterative process, instead of using constants $\lambda_1$ and $\lambda_2$, we dynamically update $\lambda_1$ and $\lambda_2$ in our one-shot budget-aware growing optimization.

Here we discuss about how budget-aware dynamic growing works in our method. Without loss of generality, we derive the $m_c$’s SGD update rule for the $\ell_0$ regularization term in Eq. 3 as:

$$m_c \leftarrow m_c - \eta \lambda_1^{\text{base}} \Delta u \frac{\delta \ell}{\delta m_c} - \eta \mu \lambda_1^{\text{base}} \Delta u m_c$$  \hspace{1cm} (8)

where $\eta$ is the learning rate and $\mu$ is the weight decay factor. At the beginning of growing epochs, when the architecture is very over-sparsified, $\Delta u$ and $\lambda_1^{\text{base}} \Delta u$ are negative values. Then $m_c$’s update is along the opposite direction of the $\ell_0$ regularization term’s gradients, encouraging $m_c$’s sparsification. As a result, some zero-valued $m_c$ will be activated and the model complexity is strongly increased to acquire enough capacity for successful training. Then, growing becomes gradually weaker as the network’s sparsity approaches the budget ($\Delta u$ to zero). Note that if the architecture is over-parameterized, $\Delta u$ and $\lambda_1^{\text{base}} \Delta u$ become positive and SGD’s update rule is the same as that of $\ell_0$ regularization. As such, our budget-aware growing can automatically and dynamically adapt the architecture complexity not only based on the task loss $L_E$ but also on the practical budget requirements in the one-shot training process.

We also note that NAS methods usually use the validation accuracy as a target during their architecture optimization phase, which may require some prior knowledge of validation accuracy on a given dataset. Our growing procedure chooses sparsity budget instead of accuracy as target because: (1) During growing, validation accuracy is influenced not only by architectures but also model weights. Directly using $\Delta \text{acc}$ may lead to sub-optimal architecture optimization. (2) A sparsity budget target is more practical and easier to set according to target devices for deployment.

A.2 Details of Evaluation Datasets

Evaluation is conducted on various tasks to demonstrate the effectiveness of our proposed method. For image classification, we use CIFAR-10 (Krizhevsky et al., 2014) and ImageNet (Deng et al., 2009): CIFAR-10 consists of 60,000 images of 10 classes, with 6,000 images per class. The train and test sets contain 50,000 and 10,000 images respectively. ImageNet is a large dataset for visual recognition which contains over 1.2M images in the training set and 50K images in the validation set covering 1,000 categories. For semantic segmentation, we use the PASCAL VOC 2012 (Everingham et al., 2015) benchmark which contains 20 foreground object classes and one background class. The original dataset contains 1,464 (train), 1,449 (val), and 1,456 (test) pixel-level labeled images for training, validation, and testing, respectively. The dataset is augmented by the extra annotations provided by (Hariharan et al., 2011), resulting in 10,582 training images. For language modeling, we use the word level Penn Treebank (PTB) dataset (Marcus et al., 1993) which consists of 929k training words, 73k validation words, and 82k test words, with 10,000 unique words in its vocabulary.

A.3 Unpruned Baseline Models

For CIFAR-10, we use VGG-16 (Simonyan & Zisserman, 2015) with BatchNorm (Ioffe & Szegedy, 2015), ResNet-20 (He et al., 2016) and WideResNet-28-10 (Zagoruyko & Komodakis, 2016) as baselines. We adopt a standard data augmentation scheme (shifting/mirroring) following (Lin et al., 2013; Huang et al., 2016), and normalize the input data with channel means and standard deviations. Note that we use the CIFAR version of ResNet-20\(^1\), VGG-16\(^2\), and WideResNet-28-10\(^3\). VGG-16, ResNet-20, and WideResNet-28-10 are trained for 160, 160, and 200 epochs, respectively, with a

\(^1\)https://github.com/akamaster/pytorch_resnet_cifar10/blob/master/resnet.py
\(^2\)https://github.com/kuangliu/pytorch-cifar/blob/master/models/vgg.py
\(^3\)https://github.com/meliketoy/wide-resnet.pytorch/blob/master/networks/wide_resnet.py
batch size of 128 and initial learning rate of 0.1. For VGG-16 and ResNet-20, we divide learning rate by 10 at epochs 80 and 120, and set the weight decay and momentum as \(10^{-4}\) and 0.9. For WideResNet-28-10, the learning rate is divided by 5 at epochs 60, 120, and 160; the weight decay and momentum are set to \(5 \times 10^{-4}\) and 0.9. For ImageNet, we train the baseline ResNet-50 and MobileNetV1 models following the respective papers. We adopt the same data augmentation scheme as in (Gross & Wilber, 2016) and report top-1 validation accuracy. For semantic segmentation, the performance is measured in terms of pixel intersection-over-union (IOU) averaged across the 21 classes (mIOU). We use Deeplab-v3-ResNet-101\(^4\) (Chen et al., 2017) as the baseline model following the training details in (Chen et al., 2017). For language modeling, we use vanilla two-layer stacked LSTM (Zaremba et al., 2014) as a baseline. The dropout keep ratio is 0.35 for the baseline model. The vocabulary size, embedding size, and hidden size of the stacked LSTMs are set as 10,000, 1,500, and 1,500, respectively, which is consistent with the settings in (Zaremba et al., 2014).

A.4 MobileNetV1 Channel Growing on ImageNet

To further validate the effectiveness of the proposed method on compact networks, we grow the filters of MobileNetV1 on ImageNet and compare the performance of our method to the results reported directly in the respective papers, as shown in Table 6. In MobileNetV1 experiments, following the same setting with Netadapt (Yang et al., 2018), we apply our method on both (1) small setting: growing MobileNetV1(128) with 0.5 multiplier while setting the original model’s multiplier as 0.25 for comparison and (2) large setting: growing standard MobileNetV1(224) while setting the original model’s multiplier as 0.75 for comparison. Note that MobileNetV1 is one of the most compact networks, and thus is more challenging to simplify than other larger networks. Our lower-cost growing method can still generate a sparser MobileNetV1 model compared with competing methods.

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>Top-1 Val Acc(%)</th>
<th>FLOPs(%)</th>
<th>Train-Cost Savings(×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNet V1(128)</td>
<td>Original(25%)</td>
<td>45.1 (+0.0)</td>
<td>100</td>
<td>1.0(×)</td>
</tr>
<tr>
<td></td>
<td>MorphNet</td>
<td>46.0 (+0.9)</td>
<td>110</td>
<td>0.9(×)</td>
</tr>
<tr>
<td></td>
<td>Netadapt</td>
<td><strong>46.3 (+1.2)</strong></td>
<td>81</td>
<td>1.1(×)</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>46.0 (+0.9)</td>
<td><strong>73</strong></td>
<td><strong>1.7</strong>(×)</td>
</tr>
<tr>
<td>MobileNet V1(224)</td>
<td>Original(75%)</td>
<td>68.8 (+0.0)</td>
<td>100</td>
<td>1.0(×)</td>
</tr>
<tr>
<td></td>
<td>Netadapt</td>
<td>69.1 (+0.3)</td>
<td>87</td>
<td>1.2(×)</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td><strong>69.3 (+0.5)</strong></td>
<td>83</td>
<td><strong>1.5</strong>(×)</td>
</tr>
</tbody>
</table>

A.5 Deeplab-v3-ResNet-101 on PASCAL VOC 2012

We also test the effectiveness of our proposed method on a semantic segmentation task by growing a Deeplab-v3-ResNet-101 model’s filter numbers from scratch directly on the PASCAL VOC 2012 dataset. We apply our method to both the ResNet-101 backbone and ASPP module. Compared to the baseline, the final generated network reduces the FLOPs by 58.5% and the parameter count by 49.8%, while approximately maintaining mIOU (76.5% to 76.4%). See Table 7.

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>mIOU</th>
<th>Params(M)</th>
<th>FLOPs(%)</th>
<th>Train-Cost Savings(×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deeplab</td>
<td>Original</td>
<td>76.5 (-0.0)</td>
<td>58.0 (100%)</td>
<td>100</td>
<td>1.0(×)</td>
</tr>
<tr>
<td></td>
<td>-v3- L1-Pruning</td>
<td>75.1 (-1.4)</td>
<td>45.7 (78.8%)</td>
<td>62.5</td>
<td>1.3(×)</td>
</tr>
<tr>
<td>ResNet101</td>
<td>Ours</td>
<td><strong>76.4 (-0.1)</strong></td>
<td><strong>29.1 (50.2%)</strong></td>
<td><strong>41.5</strong></td>
<td><strong>2.3</strong>(×)</td>
</tr>
</tbody>
</table>

\(^4\)https://github.com/chenxi116/DeepLabv3.pytorch
We compare our proposed method with ISS based on vanilla two-layer stacked LSTM. As shown
We focus on LSTMs (Hochreiter & Schmidhuber, 1997) with
VGG-16 on CIFAR-10, respectively. To better analyze the growing patterns, we visualize the channel
AutoGrow (Wen et al., 2020). For both ours and AutoGrow, “Basic” and “Bottleneck” refer to
variants of the seed architecture, we adopt three stages for CIFAR-10 and four stages for ImageNet.
due to the residual connections.
layers within each stage of ResNet-20. Also, the final channel sparsity of ResNet-20 is more uniform
channel ratios by 0.5 for better visualization. We see that our method grows more channels of earlier
Note that, for VGG-16, we divide it into 5 stages based on the pooling layer positions and normalize
dynamics grouped by stages in Figure 9 for ResNet-20 and Figure 10 for VGG-16, respectively.
Figure 7 and Figure 8 show the dynamics of train-time growing channel ratios of ResNet-20 and
PlainNets can be obtained by simply removing shortcuts from these ResNet seed variants with equal
In Table 9, we make a detailed comparison among initial seed architecture variants of ours and

\[\begin{align*}
    f_t &= \sigma_g((W_f \odot (e m^T)) x_t + (U_f \odot (m e^T)) h_{t-1} + b_f) \\
    i_t &= \sigma_g((W_i \odot (e m^T)) x_t + (U_i \odot (m e^T)) h_{t-1} + b_i) \\
    o_t &= \sigma_g((W_o \odot (e m^T)) x_t + (U_o \odot (m e^T)) h_{t-1} + b_o) \\
    c_t &= \sigma_i((W_c \odot (e m^T)) x_t + (U_c \odot (m e^T)) h_{t-1} + b_c) \\
    c_t &= f_t \odot c_{t-1} + i_t \odot c_t, \quad h_t = o_t \odot \sigma_h(c_t) \quad s.t. \quad m_c \in \{0, 1\}^{1h}, \quad e = 1^{1h},
\end{align*}\]

where \(\sigma_g\) is the sigmoid function, \(\odot\) denotes element-wise multiplication and \(\sigma_h\) is the hyperbolic
tangent function. \(x_t\) denotes the input vector at the time-step \(t\), \(h_t\) denotes the current hidden state,
and \(c_t\) denotes the long-term memory cell state. \(W_f, W_i, W_o, W_c\) denote the input-to-hidden weight
matrices and \(U_f, U_i, U_o, U_c\) denote the hidden-to-hidden weight matrices. \(m_c\) is binary indicator and
shared across all the gates to control the sparsity of hidden neurons.

We compare our proposed method with ISS based on vanilla two-layer stacked LSTM. As shown
in Table 8, our method finds more compact model structure at lower training cost, while achieving
similar perplexity on both validation and test sets. These improvements may be due to the fact that
our method dynamically grows and prunes the hidden neurons from very simple status towards a
better trade-off between model complexity and performance than that of ISS, which simply uses the

group lasso to penalize the norms of all groups collectively for compactness.

<table>
<thead>
<tr>
<th>Method</th>
<th>Perplexity (val,test)</th>
<th>Final Structure</th>
<th>Weight(M)</th>
<th>FLOPs(%)</th>
<th>Train-Cost Savings(×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>(82.57, 78.57)</td>
<td>(1500, 1500)</td>
<td>66.0M (100%)</td>
<td>100</td>
<td>1.0(×)</td>
</tr>
<tr>
<td>ISS</td>
<td>(82.59, 78.65)</td>
<td>(373, 315)</td>
<td>21.8M (33.1%)</td>
<td>13.4</td>
<td>3.8(×)</td>
</tr>
<tr>
<td>Ours</td>
<td>(82.46, 78.68)</td>
<td>(310, 275)</td>
<td>20.6M (31.2%)</td>
<td>11.9</td>
<td>5.1(×)</td>
</tr>
</tbody>
</table>

A.6 EXTENSION TO RECURRENT CELLS ON PTB DATASET

We focus on LSTMs (Hochreiter & Schmidhuber, 1997) with \(l_h\) hidden neurons, a common variant\(^4\)
of RNNs that learns long-term dependencies:

\[f_t = \sigma_g((W_f \odot (e m^T)) x_t + (U_f \odot (m e^T)) h_{t-1} + b_f)\]
\[i_t = \sigma_g((W_i \odot (e m^T)) x_t + (U_i \odot (m e^T)) h_{t-1} + b_i)\]
\[o_t = \sigma_g((W_o \odot (e m^T)) x_t + (U_o \odot (m e^T)) h_{t-1} + b_o)\]
\[c_t = \sigma_i((W_c \odot (e m^T)) x_t + (U_c \odot (m e^T)) h_{t-1} + b_c)\]
\[c_t = f_t \odot c_{t-1} + i_t \odot c_t, \quad h_t = o_t \odot \sigma_h(c_t) \quad s.t. \quad m_c \in \{0, 1\}^{1h}, \quad e = 1^{1h},\]

where \(\sigma_g\) is the sigmoid function, \(\odot\) denotes element-wise multiplication and \(\sigma_h\) is the hyperbolic
tangent function. \(x_t\) denotes the input vector at the time-step \(t\), \(h_t\) denotes the current hidden state,
and \(c_t\) denotes the long-term memory cell state. \(W_f, W_i, W_o, W_c\) denote the input-to-hidden weight
matrices and \(U_f, U_i, U_o, U_c\) denote the hidden-to-hidden weight matrices. \(m_c\) is binary indicator and
shared across all the gates to control the sparsity of hidden neurons.

We compare our proposed method with ISS based on vanilla two-layer stacked LSTM. As shown
in Table 8, our method finds more compact model structure at lower training cost, while achieving
similar perplexity on both validation and test sets. These improvements may be due to the fact that
our method dynamically grows and prunes the hidden neurons from very simple status towards a
better trade-off between model complexity and performance than that of ISS, which simply uses the

group lasso to penalize the norms of all groups collectively for compactness.

A.7 VARIANTS OF INITIAL SEED ARCHITECTURE

In Table 9, we make a detailed comparison among initial seed architecture variants of ours and
AutoGrow (Wen et al., 2020). For both ours and AutoGrow, “Basic” and “Bottleneck” refer to
ResNets with standard basic and bottleneck residual blocks, while “PlainLayers” refers to stacked
convolutional, batch normalization, and ReLU layer combinations. Similar with standard ResNets, for
variants of the seed architecture, we adopt three stages for CIFAR-10 and four stages for ImageNet.
PlainNets can be obtained by simply removing shortcuts from these ResNet seed variants with equal
stage numbers. For each stage, we start from only one growing unit, within which initial filter
numbers are also initialized at one for channel growing.

A.8 TRACK OF ANY-TIME SPARSIFICATION DURING CHANNEL GROWING

Figure 7 and Figure 8 show the dynamics of train-time growing channel ratios of ResNet-20 and
VGG-16 on CIFAR-10, respectively. To better analyze the growing patterns, we visualize the channel
dynamics grouped by stages in Figure 9 for ResNet-20 and Figure 10 for VGG-16, respectively.
Note that, for VGG-16, we divide it into 5 stages based on the pooling layer positions and normalize
dchannel ratios by 0.5 for better visualization. We see that our method grows more channels of earlier
layers within each stage of ResNet-20. Also, the final channel sparsity of ResNet-20 is more uniform
due to the residual connections.
Table 9: A detailed comparison among seed architecture variants of our method and AutoGrow (Wen et al., 2020). In growing units term, “Basic” and “Bottleneck” refer to ResNets with standard basic and bottleneck residual blocks while “PlainLayers” refers to standard convolutional layer, BN, and ReLu layer combinations in VGG-like networks without shortcuts.

<table>
<thead>
<tr>
<th>Families</th>
<th>Variants</th>
<th>Methods</th>
<th>Channel Growing</th>
<th>Growing Units</th>
<th>Stages</th>
<th>Shortcuts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet</td>
<td>Basic3ResNet</td>
<td>Ours</td>
<td>✓</td>
<td>Basic</td>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AutoGrow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bottleneck4ResNet</td>
<td>Ours</td>
<td>✓</td>
<td>Bottleneck</td>
<td>4</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AutoGrow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGG-like</td>
<td>Plain3Net</td>
<td>Ours</td>
<td>✓</td>
<td>PlainLayers</td>
<td>3</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AutoGrow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Plain4Net</td>
<td>Ours</td>
<td>✓</td>
<td>PlainLayers</td>
<td>4</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AutoGrow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: Epoch-wise retained channel ratio dynamics for each layer in ResNet-20.

Figure 8: Epoch-wise retained channel ratio dynamics for each layer in VGG-16.

A.9 FLOPS-BASED BUDGET-AWARE GROWING

We also investigate the effectiveness of setting a FLOPs target for budget-aware growing in Figure 11. We observe similar trends among uniform pruning, ours growing, and ours direct design: in most FLOPs budget settings, our growing method outperforms direct design and uniform pruning. We also observe that when setting extreme sparse FLOPs target (e.g., 85%), our method achieves lower accuracy than the other two variants. The reason is that our channel growing is forced to only grow architectures from ~99% sparsity up to ~85% FLOPs and ~90% parameters sparsity, during which models cannot acquire enough capacity to be well trained.

A.10 INTERACTIONS BETWEEN LEARNING RATE AND TEMPERATURE SCHEDULERS

Two factors influence the growing optimization speed in our method: temperature and learning rate, which are hyperparameters controlled by their respective schedulers. We first visualize the structure-wise separate temperature dynamics in Figure 12 by averaging temperatures per layer during ResNet-20 channel growing on CIFAR-10. We see that temperatures are growing with different rates for channels. Usually, low learning rate and high temperature in late training epochs make the network growing optimization become very stable. In Figure 13, we deliberately decay γ in the temperature scheduler, mirroring the learning rate decay schedule, in order to force growing until the end. As shown in Figure 14, our method is still adapting some layers even at the last epoch. We find that such instability degrades performance, since some newly grown filters may not have enough time to become well trained.

5The proposed configuration space can be readily applied to the compression of GRUs (Cho et al., 2014) and vanilla RNNs.
Published as a conference paper at ICLR 2021

Figure 9: Visualization of retained channel ratio dynamics for each stage in ResNet-20.

Figure 10: Visualization of retained channel ratio dynamics for each stage in VGG-16.

Figure 11: Pruned architectures obtained by ablated methods with different FLOPs sparsity.

Figure 12: Structure-wise separate temperature dynamics in channel growing.

Figure 13: Structure-wise separate decayed temperature dynamics in channel growing.

Figure 14: Track of epoch-wise train-time FLOPs for channel growing in ResNet-20.