NeST: A Neural Network Synthesis Tool Based on a Grow-and-Prune Paradigm

Xiaoliang Dai
Princeton University
xdai@princeton.edu

Hongxu Yin
Princeton University
hongxuy@princeton.edu

Niraj K. Jha
Princeton University
jha@princeton.edu

Abstract

Deep neural networks (DNNs) have begun to have a pervasive impact on various applications of machine learning. However, the problem of finding an optimal DNN architecture for large applications is challenging. Common approaches go for deeper and larger DNN architectures but may incur substantial redundancy. To address these problems, we introduce a network growth algorithm that complements network pruning to learn both weights and compact DNN architectures during training. We propose a DNN synthesis tool (NeST) that combines both methods to automate the generation of compact and accurate DNNs. NeST starts with a randomly initialized sparse network called the seed architecture. It iteratively tunes the architecture with gradient-based growth and magnitude-based pruning of neurons and connections. Our experimental results show that NeST yields accurate, yet very compact DNNs, with a wide range of seed architecture selection. For the LeNet-300-100 (LeNet-5) architecture, we reduce network parameters by $70.2\times$ ($74.3\times$) and floating-point operations (FLOPs) by $79.4\times$ ($43.7\times$). For the AlexNet and VGG-16 architectures, we reduce network parameters (FLOPs) by $15.7\times$ ($4.6\times$) and $30.2\times$ ($8.6\times$), respectively. NeST's grow-and-prune paradigm delivers significant additional parameter and FLOPs reduction relative to pruning-only methods.

1 Introduction

Over the last decade, deep neural networks (DNNs) have begun to revolutionize myriad research domains, such as computer vision, speech recognition, and machine translation [1–3]. Their ability to distill intelligence from a dataset through multi-level abstraction can even lead to super-human performance [4]. Thus, DNNs are emerging as a new cornerstone of modern artificial intelligence.

Though critically important, how to efficiently derive an appropriate DNN architecture from large datasets has remained an open problem. Researchers have traditionally derived the DNN architecture by sweeping through its architectural parameters and training the corresponding architecture until the point of diminishing returns in its performance. This suffers from three major problems. First, the widely used back-propagation (BP) algorithm assumes a fixed DNN architecture and only trains weights. Thus, training cannot improve the architecture. Second, a trial-and-error methodology can be inefficient when DNNs get deeper and contain millions of parameters. Third, simply going deeper and larger may lead to large, accurate, but over-parameterized DNNs. For example, Han et al. [5] showed that the number of parameters in VGG-16 can be reduced by $13\times$ with no loss of accuracy.

To address these problems, we propose a DNN synthesis tool (NeST) that trains both DNN weights and architectures. NeST is inspired by the learning mechanism of the human brain, where the number of synaptic connections increases upon the birth of a baby, peaks after a few months, and decreases steadily thereafter [6]. NeST starts DNN synthesis from a seed DNN architecture (*birth point*). It

---

allows the DNN to **grow** connections and neurons based on gradient information (*baby brain*) so that the DNN can adapt to the problem at hand. Then, it **prunes** away insignificant connections and neurons based on magnitude information (*adult brain*) to avoid redundancy. A combination of network growth and pruning algorithms enables NeST to generate accurate and compact DNNs. We used NeST to synthesize various compact DNNs for the MNIST [7] and ImageNet [8] datasets. NeST leads to drastic reductions in the number of parameters and floating-point operations (FLOPs) relative to the DNN baselines, with no accuracy loss.

2 Related Work

An evolutionary algorithm provides a promising solution to DNN architecture selection through evolution of network architectures. Its search mechanism involves iterations over mutation, recombination, and most importantly, evaluation and selection of network architectures [9, 10]. Additional performance enhancement techniques include better encoding methods [11] and algorithmic redesign for DNNs [12]. All these assist with more efficient search in the wide DNN architecture space.

Reinforcement learning (RL) has emerged as a new powerful tool to solve this problem [13–16]. Zoph et al. [13] use a recurrent neural network controller to iteratively generate groups of candidate networks, whose performance is then used as a reward for enhancing the controller. Baker et al. [14] propose a Q-learning based RL approach that enables convolutional architecture search. A recent work [15] proposes the NASNet architecture that uses RL to search for architectural building blocks and achieves better performance than human-invented architectures.

The structure adaptation (SA) approach exploits network *clues* (e.g., distribution of weights) to incorporate architecture selection into the training process. Existing SA methods can be further divided into two categories: constructive and destructive. A constructive approach starts with a small network and iteratively adds connections/neurons [17, 18]. A destructive approach, on the other hand, starts with a large network and iteratively removes connections/neurons. This can effectively reduce model redundancy. For example, recent pruning methods, such as network pruning [5, 19–21], layer-wise surgeon [22], sparsity learning [23–26], and dynamic network surgery [27], can offer extreme compactness for existing DNNs with no or little accuracy loss.

3 Synthesis Methodology

In this section, we propose NeST that leverages both constructive and destructive SA approaches through a grow-and-prune paradigm. Unless otherwise stated, we adopt the notations given in Table 1 to represent various variables.

Table 1: Notations and descriptions

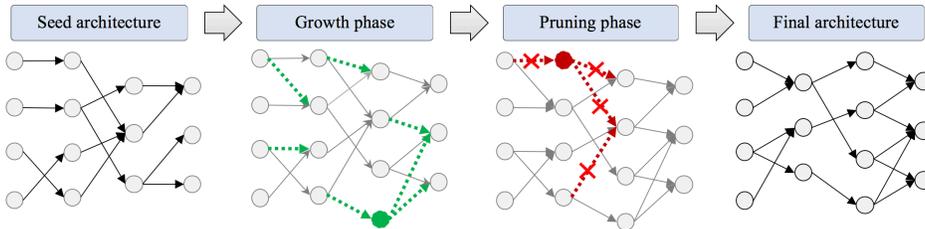| Label	Description	Label	Description
L	DNN loss function	$\mathbf{W}^l$	weights between $(l-1)^{th}$ and l^{th} layer
x_n^l	output value of nth neuron in l^{th} layer	$\mathbf{b}^l$	biases in l^{th} layer
u_m^l	input value of mth neuron in l^{th} layer	$\mathbf{R}^{d_0 \times d_1 \times d_2}$	d_0 by d_1 by d_2 matrix with real elements

Figure 1: An illustration of the architecture synthesis flow in NeST.

3.1 Neural Network Synthesis Tool

We illustrate the NeST approach with Fig. 1. Synthesis begins with an initial seed architecture, typically initialized as a sparse and partially connected DNN. We also ensure that all neurons are connected in the seed architecture. Then, NeST utilizes two sequential phases to synthesize the
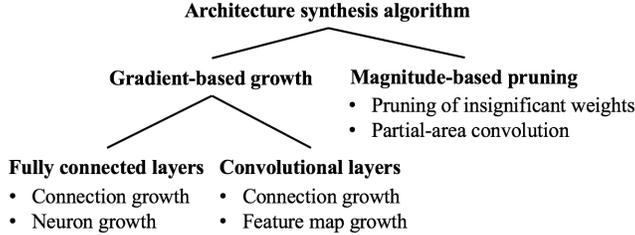
Figure 2: Major components of the DNN architecture synthesis algorithm in NeST.
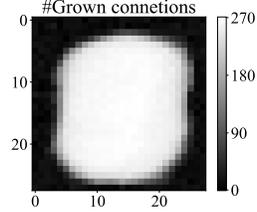

Figure 3: Grown connections from the input layer to the first layer of LeNet-300-100.

DNN: (i) gradient-based growth phase, and (ii) magnitude-based pruning phase. In the growth phase, the gradient information in the architecture space is used to gradually grow new connections, neurons, and feature maps to achieve the desired accuracy. In the pruning phase, the DNN inherits the synthesized architecture and weights from the growth phase and iteratively removes redundant connections and neurons, based on their magnitudes. Finally, NeST comes to rest at a lightweight DNN model that incurs no accuracy degradation relative to a fully connected model.

3.2 Gradient-based Growth

In this section, we explain our algorithms to grow connections, neurons, and feature maps.

3.2.1 Connection Growth

The connection growth algorithm greedily activates useful, but currently 'dormant,' connections. We incorporate it in the following learning policy:

Policy 1: Add a connection w iff it can quickly reduce the value of loss function L.

The DNN seed contains only a small fraction of active connections to propagate gradients. To locate the 'dormant' connections that can reduce L effectively, we evaluate $\partial L/\partial w$ for all the 'dormant' connections w (computed either using the whole training set or a large batch). Policy 1 activates 'dormant' connections iff they are the most efficient at reducing L. This can also assist with avoiding local minima and achieving higher accuracy [28]. To illustrate this policy, we plot the connections grown from the input to the first layer of LeNet-300-100 [7] (for the MNIST dataset) in Fig. 3. The image center has a much higher grown density than the margins, consistent with the fact that the MNIST digits are centered.

From a neuroscience perspective, our connection growth algorithm coincides with the Hebbian theory: "Neurons that fire together wire together [29]." We define the stimulation magnitude of the mth presynaptic neuron in the $(l+1)^{th}$ layer and the nth postsynaptic neuron in the lth layer as $\partial L/\partial u_m^{l+1}$ and x_n^l, respectively. The connections activated based on Hebbian theory would have a strong correlation between presynaptic and postsynaptic cells, thus a large value of $\left|(\partial L/\partial u_m^{l+1})x_n^l\right|$. This is also the magnitude of the gradient of L with respect to w (w is the weight that connects u_m^{l+1} and x_n^l):

$$|\partial L/\partial w| = \left|(\partial L/\partial u_m^{l+1})x_n^l\right| \tag{1}$$

Thus, this is mathematically equivalent to Policy 1.

3.2.2 Neuron Growth

Our neuron growth algorithm consists of two steps: (i) connection establishment and (ii) weight initialization. The neuron growth policy is as follows:

Policy 2: In the lth layer, add a new neuron as a shared intermediate node between existing neuron pairs that have high postsynaptic (x) and presynaptic ($\partial L/\partial u$) neuron correlations (each pair contains one neuron from the $(l-1)^{th}$ layer and the other from the $(l+1)^{th}$ layer). Initialize weights based on batch gradients to reduce the value of L.

Algorithm 1 incorporates Policy 2 and illustrates the neuron growth iterations in detail. Before adding a neuron to the lth layer, we evaluate the bridging gradient between the neurons at the previous $(l-1)^{th}$ and subsequent $(l+1)^{th}$ layers. We connect the top $\beta \times 100\%$ (β is the growth ratio)

3

Algorithm 1 Neuron growth in the lth layer

Input: α - birth strength, β - growth ratio
Denote: M - number of neurons in the $(l+1)^{th}$ layer, N - number of neurons in the $(l-1)^{th}$ layer,
$\mathbf{G} \in R^{M \times N}$ - bridging gradient matrix, avg - extracts mean value of non-zero elements
Add a neuron in the lth layer, initialize $\mathbf{w}^{out} = \vec{0} \in R^M$, $\mathbf{w}^{in} = \vec{0} \in R^N$
for $1 \leq m \leq M, 1 \leq n \leq N$ **do**
    $G_{m,n} = \frac{\partial L}{\partial u_m^{l+1}} \times x_n^{l-1}$
end for
$thres = (\beta MN)^{th}$ largest element in $abs(\mathbf{G})$
for $1 \leq m \leq M, 1 \leq n \leq N$ **do**
    if $|G_{m,n}| > thres$ **then**
        $\delta w = \sqrt{|G_{m,n}|} \times rand\{1, -1\}$
        $w_m^{out} \leftarrow w_m^{out} + \delta w, w_n^{in} \leftarrow w_n^{in} + \delta w \times sgn(G_{m,n})$
    end if
    $\mathbf{w}^{out} \leftarrow \mathbf{w}^{out} \times \alpha \frac{avg(abs(\mathbf{W}^{l+1}))}{avg(abs(\mathbf{w}^{out}))}$, $\mathbf{w}^{in} \leftarrow \mathbf{w}^{in} \times \alpha \frac{avg(abs(\mathbf{W}^l))}{avg(abs(\mathbf{w}^{in}))}$
end for
Concatenate network weights $\mathbf{W}$ with $\mathbf{w}^{in}$, $\mathbf{w}^{out}$

---

correlated neuron pairs through a new neuron in the lth layer. We initialize the weights based on the bridging gradient to enable gradient descent, thus decreasing the value of L.

We implement a square root rule for weight initialization to imitate a BP update on the bridging connection w_b, which connects x_n^{l-1} and u_m^{l+1}. The BP update leads to a change in u_m^{l+1}:

$$|\Delta u_m^{l+1}|_{b.p.} = |x_n^{l-1} \times \delta w_b| = \eta |x_n^{l-1} \times G_{m,n}| \tag{2}$$

where η is the learning rate. In Algorithm 1, when we connect the newly added neuron (in the lth layer) with x_n^{l-1} and u_m^{l+1}, we initialize their weights to the square root of the magnitude of the bridging gradient:

$$|\delta w_n^{in}| = |\delta w_m^{out}| = \sqrt{|G_{m,n}|} \tag{3}$$

where δw_n^{in} (δw_m^{out}) is the initialized value of the weight that connects the newly added neuron with x_n^{l-1} (u_m^{l+1}). The weight initialization rule leads to a change in u_m^{l+1}:

$$|\Delta u_m^{l+1}| = |f(x_n^{l-1} \times \delta w_n^{in}) \times \delta w_m^{out}| \tag{4}$$

where f is the neuron activation function. Suppose $tanh$ is the activation function. Then,

$$f(x) = tanh(x) \approx x, \text{if } x \ll 1 \tag{5}$$

Since δw_n^{in} and δw_m^{out} are typically very small, the approximation in Eq. (5) leads to Eq. (6).

$$|\Delta u_m^{l+1}| \approx |x_n^{l-1} \times \delta w_n^{in} \times \delta w_m^{out}| = \frac{1}{\eta} \times |\Delta u_m^{l+1}|_{b.p.} \tag{6}$$

This is linearly proportional to the effect of a BP update. Thus, our weight initialization mathematically imitates a BP update. Though we illustrated the algorithm with the $tanh$ activation function, the weight initialization rule works equally well with other activation functions, such as rectified linear unit (ReLU) and leaky rectified linear unit (Leaky ReLU).

We use a birth strength factor α to strengthen the connections of a newly grown neuron. This prevents these connections from becoming too weak to survive the pruning phase. Specifically, after square root rule based weight initialization, we scale up the newly added weights by

$$\mathbf{w}^{out} \leftarrow \alpha \mathbf{w}^{out} \times \frac{avg(abs(\mathbf{W}^{l+1}))}{avg(abs(\mathbf{w}^{out}))}, \quad \mathbf{w}^{in} \leftarrow \alpha \mathbf{w}^{in} \times \frac{avg(abs(\mathbf{W}^l))}{avg(abs(\mathbf{w}^{in}))} \tag{7}$$

where avg is an operation that extracts the mean value of all non-zero elements. This strengthens new weights. In practice, we find $\alpha > 0.3$ to be an appropriate range.

3.2.3 Growth in the Convolutional Layers

Convolutional layers share the connection growth methodology of Policy 1. However, instead of neuron growth, we use a unique feature map growth algorithm for convolutional layers. In a convolutional layer, we convolve input images with kernels to generate feature maps. Thus, to add a feature map, we need to initialize the corresponding set of kernels. We summarize the feature map growth policy as follows:

Policy 3: To add a new feature map to the convolutional layers, randomly generate sets of kernels, and pick the set of kernels that reduces L the most.

In our experiment, we observe that the percentage reduction in L for Policy 3 is approximately twice as in the case of the naive approach that initializes the new kernels with random values.

3.3 Magnitude-based Pruning

We prune away insignificant connections and neurons based on the magnitude of weights and outputs:

Policy 4: Remove a connection (neuron) iff the magnitude of the weight (neuron output) is smaller than a pre-defined threshold.

We next explain two variants of Policy 4: pruning of insignificant weights and partial-area convolution.

3.3.1 Pruning of Insignificant Weights

Han et al. [5] show that magnitude-based pruning can successfully cut down the memory and computational costs. We extend this approach to incorporate the batch normalization technique. Such a technique can reduce the internal covariate shift by normalizing layer inputs and improve the training speed and behavior. Thus, it has been widely applied to large DNNs [30]. Consider the lth batch normalization layer:

$$\mathbf{u}^l = [(\mathbf{W}^l \mathbf{x}^{l-1} + \mathbf{b}^l) - \mathbf{E}] \oslash \mathbf{V} = \mathbf{W}^l_* \mathbf{x} + \mathbf{b}^l_* \tag{8}$$

where $\mathbf{E}$ and $\mathbf{V}$ are batch normalization terms, and $\oslash$ depicts the Hadamard (element-wise) division operator. We define effective weights $\mathbf{W}^l_*$ and effective biases $\mathbf{b}^l_*$ as:

$$\mathbf{W}^l_* = \mathbf{W}^l \oslash \mathbf{V}, \mathbf{b}^l_* = (\mathbf{b}^l - \mathbf{E}) \oslash \mathbf{V} \tag{9}$$

We treat connections with small effective weights as insignificant. Pruning of insignificant weights is an iterative process. In each iteration, we only prune the most insignificant weights (e.g., top 1%) for each layer, and then retrain the whole DNN to recover its performance.

3.3.2 Partial-area Convolution

In common convolutional neural networks (CNNs), the convolutional layers typically consume $\sim 5\%$ of the parameters, but contribute to $\sim 90\text{-}95\%$ of the total FLOPs [31]. In a convolutional layer, kernels shift and convolve with the entire input image. This process incurs redundancy, since not the whole input image is of interest to a particular kernel. Anwar et al. [32] presented a method to prune all connections from a not-of-interest input image to a particular kernel. This method reduces FLOPs but incurs performance degradation [32].

Instead of discarding an entire image, our proposed partial-area convolution algorithm allows kernels to convolve with the image areas that are of interest. We refer to such an area as ***area-of-interest***. We prune connections to other image areas. We illustrate this process in Fig. 4. The green area depicts ***area-of-interest***, whereas the red area depicts parts that are not of interest. Thus, green connections (solid lines) are kept, whereas red ones (dashed lines) are pruned away.

Partial-area convolution pruning is an iterative process. We present one iteration in Algorithm 2. We first convolve M input images with $M \times N$ convolution kernels and generate $M \times N$ feature maps, which are stored in a four-dimensional feature map matrix $\mathbf{C}$. We set the pruning threshold $thres$ to the $(100\gamma)^{th}$ percentile of all elements in $abs(\mathbf{C})$, where γ is the pruning ratio, typically 1% in our experiment. We mark the elements whose values are below this threshold as insignificant, and prune away their input connections. We retrain the whole DNN after each pruning iteration. In our current implementation, we utilize a mask $\mathbf{Msk}$ to disregard the pruned convolution area.

Partial-area convolution enables additional FLOPs reduction without any performance degradation. For example, we can reduce FLOPs in LeNet-5 [7] by $2.09\times$ when applied to MNIST. Compared to

Algorithm 2 Partial-area convolution

Input: I - M input images, **K** - kernel matrix, **Msk** - feature map mask, γ - pruning ratio
Output: Msk, F - N feature maps
Denote: C $\in R^{M \times N \times P \times Q}$ - Depthwise feature map, $\otimes$ - Hadamard (element-wise) multiplication
for $1 \le m \le M, 1 \le n \le N$ **do**
   $\mathbf{C}_{m,n} = convolve(\mathbf{I}_m, \mathbf{K}_{m,n})$
end for
$thres = (\gamma MNPQ)^{th}$ largest element in $abs(\mathbf{C})$
for $1 \le m \le M, 1 \le n \le N, 1 \le p \le P, 1 \le q \le Q$ **do**
   if $|C_{m,n,p,q}| < thres$ **then**
      $Msk_{m,n,p,q} = 0$
   end if
end for
$\mathbf{C} \leftarrow \mathbf{C} \otimes \mathbf{Msk}, \ \mathbf{F} \leftarrow \Sigma_{m=1}^{M} \mathbf{C}_m$
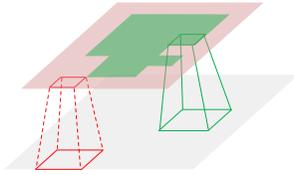

Figure 4: Pruned connections (dashed red lines) and remaining connections (solid green lines) in partial-area convolution.
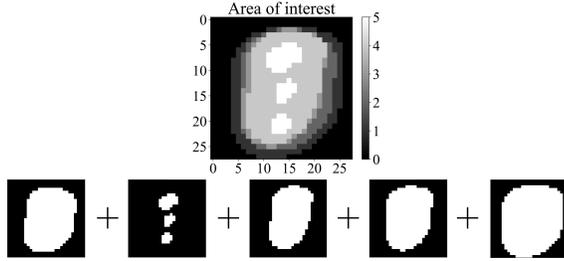

Figure 5: Area-of-interest for five different kernels in the first layer of LeNet-5.

the conventional CNNs that force a fixed square-shaped ***area-of-interest*** on all kernels, we allow each kernel to self-explore the preferred shape of its ***area-of-interest***. Fig. 5 shows the ***area-of-interest*** found by the layer-1 kernels in LeNet-5 when applied to MNIST. We observe significant overlaps in the image central area, which most kernels are interested in.

4 Experimental Results

We implement NeST using Tensorflow [33] and PyTorch [34] on Nvidia GTX 1060 and Tesla P100 GPUs. We use NeST to synthesize compact DNNs for the MNIST and ImageNet datasets. We select DNN seed architectures based on clues (e.g., depth, kernel size, etc.) from the existing LeNets, AlexNet, and VGG-16 architectures, respectively. NeST exhibits two major advantages:

- **Wide seed range**: NeST yields high-performance DNNs with a wide range of seed architectures. Its ability to start from a wide range of seed architectures alleviates reliance on human-defined architectures, and offers more freedom to DNN designers.

- **Drastic redundancy removal**: NeST-generated DNNs are very compact. Compared to the DNN architectures generated with pruning-only methods, DNNs generated through our grow-and-prune paradigm have much fewer parameters and require much fewer FLOPs.

4.1 LeNets on MNIST

We derive the seed architectures from the original LeNet-300-100 and LeNet-5 networks [7]. LeNet-300-100 is a multi-layer perceptron with two hidden layers. LeNet-5 is a CNN with two convolutional layers and three fully connected layers. We use the affine-distorted MNIST dataset [7], on which LeNet-300-100 (LeNet-5) can achieve an error rate of 1.3% (0.8%). We discuss our results next.

4.1.1 Growth Phase

First, we derive nine (four) seed architectures for LeNet-300-100 (LeNet-5). These seeds contain fewer neurons and connections per layer than the original LeNets. The number of neurons in each layer is the product of a ratio r and the corresponding number in the original LeNets (e.g., the seed architecture for LeNet-300-100 becomes LeNet-120-40 if $r = 0.4$). We randomly initialize only 10%
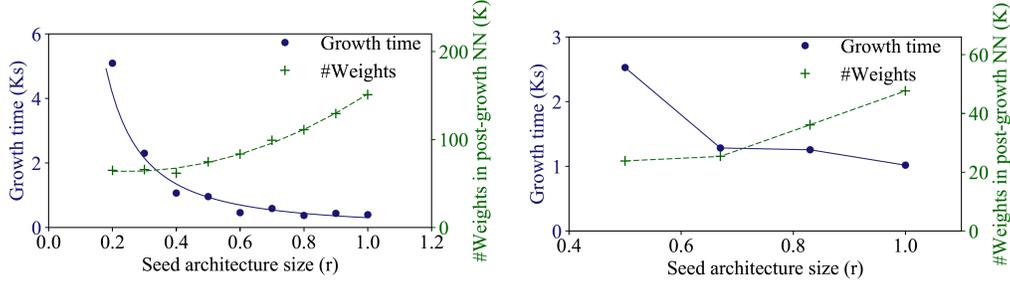
Figure 6: Growth time vs. post-growth DNN size trade-off for various seed architectures for LeNet-300-100 (left) and LeNet-5 (right) to achieve a 1.3% and 0.8% error rate, respectively.
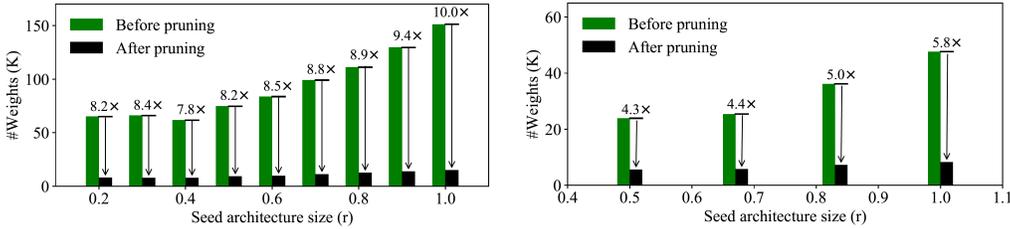

Figure 7: Compression ratio and final DNN size for different LeNet-300-100 (left) and LeNet-5 (right) seed architectures.

of all possible connections in the seed architecture. Also, we ensure that all neurons in the network are connected.

We first sweep r for LeNet-300-100 (LeNet-5) from 0.2 (0.5) to 1.0 (1.0) with a step-size of 0.1 (0.17), and then grow the DNN architectures from these seeds. We study the impact of these seeds on the GPU time for growth and post-growth DNN sizes under the same target accuracy (this accuracy is typically a reference value for the architecture). We summarize the results for LeNets in Fig. 6. We have two interesting findings for the growth phase:

1. Smaller seed architectures often lead to smaller post-growth DNN sizes, but at the expense of a higher growth time. We will later show that smaller seeds and thus smaller post-growth DNN sizes are better, since they also lead to smaller final DNN sizes.

2. When the post-growth DNN size saturates due to the full exploitation of the synthesis freedom for a target accuracy, a smaller seed is no longer beneficial, as evident from the flat left ends of the dashed curves in Fig. 6.

4.1.2 Pruning Phase

Next, we prune the post-growth LeNet DNNs to remove their redundant neurons/connections. We show the post-pruning DNN sizes and compression ratios for LeNet-300-100 and LeNet-5 for the different seeds in Fig. 7. We have two major observations for the pruning phase:

1. Larger the pre-pruning DNN, larger is its compression ratio. This is because larger pre-pruning DNNs have a larger number of weights and thus also higher redundancy.

2. Larger the pre-pruning DNN, larger is its post-pruning DNN. Thus, to synthesize a more compact DNN, one should choose a smaller seed architecture (growth phase **finding 1**) within an appropriate range (growth phase **finding 2**).

4.1.3 Inference model comparison

We compare our results against related results from the literature in Table 2. Our results outperform other reference models from various design perspectives. Without any loss of accuracy, we are able to reduce the number of connections and FLOPs of LeNet-300-100 (LeNet-5) by 70.2× (74.3×) and 79.4× (43.7×), respectively, relative to the baseline Caffe model [36]. We include the model details in the Appendix.

Table 2: Different inference models for MNIST

Model	Method	Error	#Param	FLOPs
RBF network [7]	-	3.60%	794K	1588K
Polynomial classifier [7]	-	3.30%	40K	78K
K-nearest neighbors [7]	-	3.09%	47M	94M
SVMs (reduced set) [35]	-	1.10%	650K	1300K
Caffe model (LeNet-300-100) [36]	-	1.60%	266K	532K
LWS (LeNet-300-100) [22]	Prune	1.96%	4K	8K
Net pruning (LeNet-300-100) [5]	Prune	1.59%	22K	43K
Our LeNet-300-100: compact	**Grow+Prune**	**1.58%**	**3.8K**	**6.7K**
Our LeNet-300-100: accurate	**Grow+Prune**	**1.29%**	**7.8K**	**14.9K**
Caffe model (LeNet-5) [36]	-	0.80%	431K	4586K
LWS (LeNet-5) [22]	Prune	1.66%	4K	199K
Net pruning (LeNet-5) [5]	Prune	0.77%	35K	734K
Our LeNet-5	**Grow+Prune**	**0.77%**	**5.8K**	**105K**

Table 3: Different AlexNet and VGG-16 based inference models for ImageNet

Model	Method	ΔTop-1 err.	ΔTop-5 err.	#Param (M)	FLOPs (B)
Baseline AlexNet [37]	-	0.0%	0.0%	61 (1.0×)	1.5 (1.0×)
Data-free pruning [38]	Prune	+1.62%	-	39.6 (1.5×)	1.0 (1.5×)
Fastfood-16-AD [39]	-	+0.12%	-	16.4 (3.7×)	1.4 (1.1×)
Memory-bounded [40]	-	+1.62%	-	15.2 (4.0×)	-
SVD [41]	-	+1.24%	+0.83%	11.9 (5.1×)	-
LWS (AlexNet) [22]	Prune	+0.33%	+0.28%	6.7 (9.1×)	0.5 (3.0×)
Net pruning (AlexNet) [5]	Prune	-0.01%	-0.06%	6.7 (9.1×)	0.5 (3.0×)
Our AlexNet	**Grow+Prune**	**-0.02%**	**-0.06%**	**3.9 (15.7×)**	**0.33 (4.6×)**
Baseline VGG-16 [42]	-	0.0%	0.0%	138 (1.0×)	30.9 (1.0×)
LWS (VGG-16) [22]	Prune	+3.61%	+1.35%	10.3 (13.3×)	6.5 (4.8×)
Net pruning (VGG-16) [5]	Prune	+2.93%	+1.26%	10.3 (13.3×)	6.5 (4.8×)
Our VGG-16: accurate	**Grow+Prune**	**-0.35%**	**-0.31%**	**9.9 (13.9×)**	**6.3 (4.9×)***
Our VGG-16: compact	**Grow+Prune**	**+2.31%**	**+0.98%**	**4.6 (30.2×)**	**3.6 (8.6×)***

∗ Currently without partial-area convolution due to GPU memory limits.

4.2 AlexNet and VGG-16 on ImageNet

Next, we use NeST to synthesize DNNs for the ILSVRC 2012 image classification dataset [8]. We initialize a slim and sparse seed architecture base on the AlexNet [31] and VGG-16 [43]. Our seed architecture for AlexNet contains only 60, 140, 240, 210, and 160 feature maps in the five convolutional layers, and 3200, 1600, and 1000 neurons in the fully connected layers. The seed architecture for VGG-16 uses $r = 0.75$ for the first 13 convolutional layers, and has 3200, 1600, and 1000 neurons in the fully connected layers. We randomly activate 30% of all the possible connections for both seed architectures.

Table 3 compares the model synthesized by NeST with various AlexNet and VGG-16 based inference models. We include the model details in the Appendix. Our baselines are the AlexNet Caffe model (42.78% top-1 and 19.73% top-5 error rate) [5] and VGG-16 PyTorch model (28.41% top-1 and 9.62% top-5 error rate) [42]. Our grow-and-prune synthesis paradigm outperforms the pruning-only methods listed in Table 3. This may be explained by the observation that pruning methods potentially inherit a certain amount of redundancy associated with the original large DNNs. Network growth can alleviate this phenomenon.

Note that our current mask-based implementation of growth and pruning incurs a temporal memory overhead during training. If the model becomes deeper, as in the case of ResNet [44] or DenseNet [45], using masks to grow and prune connections/neurons/feature maps may not be economical due to this temporal training memory overhead. We plan to address this aspect in our future work.

5 Discussions

Our synthesis methodology incorporates three inspirations from the human brain.

First, the number of synaptic connections in a human brain varies at different human ages. It rapidly increases upon the baby's birth, peaks after a few months, and decreases steadily thereafter. A DNN experiences a very similar learning process in NeST, as shown in Fig. 8. This curve shares a very similar pattern to the evolution of the number of synapses in the human brain [46].
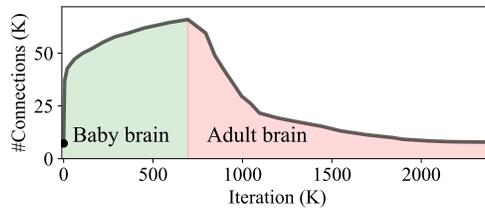

Figure 8: #Connections vs. synthesis iteration for LeNet-300-100.

Second, most learning processes in our brain result from rewiring of synapses between neurons. Our brain grows and prunes away a large amount (up to 40%) of synaptic connections every day [6]. NeST wakes up new connections, thus effectively rewiring more neurons pairs in the learning process. Thus, it mimics the 'learning through rewiring' mechanism of human brains.

Third, only a small fraction of neurons are active at any given time in human brains. This mechanism enables the human brain to operate at an ultra-low power (20 Watts). However, fully connected DNNs contain a substantial amount of insignificant neuron responses per inference. To address this problem, we include a magnitude-based pruning algorithm in NeST to remove the redundancy, thus achieving sparsity and compactness. This leads to huge storage and computation reductions.

6 Conclusions

In this paper, we proposed a synthesis tool, NeST, to synthesize compact yet accurate DNNs. NeST starts from a sparse seed architecture, adaptively adjusts the architecture through gradient-based growth and magnitude-based pruning, and finally arrives at a compact DNN with high accuracy. For LeNet-300-100 (LeNet-5) on MNIST, we reduced the number of network parameters by $70.2\times$ $(74.3\times)$ and FLOPs by $79.4\times$ $(43.7\times)$. For AlexNet and VGG-16 on ImageNet, we reduced the network parameters (FLOPs) by $15.7\times$ $(4.6\times)$ and $30.2\times$ $(8.6\times)$, respectively.

References

[1] A. Graves, A.-R. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, 2013, pp. 6645–6649.

[2] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Advances in Neural Information Processing Systems*, 2014, pp. 3104–3112.

[3] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *Proc. Advances in Neural Information Processing Systems*, vol. 2011, no. 2, 2011, p. 5.

[4] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.

[6] J. Hawkins, "Machines won't become intelligent unless they incorporate certain features of the human brain," *IEEE Spectrum*, vol. 54, no. 6, pp. 34–71, Jun. 2017.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[9] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *Proc. Int. Conf. Machine Learning*, 2017, pp. 2902–2911.

[10] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[11] R. Negrinho and G. Gordon, "Deeparchitect: Automatically designing and training deep architectures," *arXiv preprint arXiv:1704.08792*, 2017.

[12] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving deep neural networks," *arXiv preprint arXiv:1703.00548*, 2017.

[13] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[14] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[15] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *arXiv preprint arXiv:1707.07012*, 2017.

[16] Y. Zhou and G. Diamos, "Neural architect: A multi-objective neural architecture search with performance prediction," in *Proc. Conf. SysML*, 2018.

[17] T. Ash, "Dynamic node creation in backpropagation networks," *Connection Science*, vol. 1, no. 4, pp. 365–375, 1989.

[18] M. Mezard and J.-P. Nadal, "Learning in feedforward layered networks: The tiling algorithm," *J. of Physics A: Mathematical and General*, vol. 22, no. 12, p. 2191, 1989.

[19] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[20] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[21] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[22] X. Dong, S. Chen, and S. J. Pan, "Learning to prune deep neural networks via layer-wise optimal brain surgeon," *arXiv preprint arXiv:1705.07565*, 2017.

[23] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Advances in Neural Information Processing Systems*, 2016, pp. 2074–2082.

[24] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2015, pp. 806–814.

[25] Y. Sun, X. Wang, and X. Tang, "Sparsifying neural network connections for face recognition," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2016, pp. 4856–4864.

[26] D. Yu, F. Seide, G. Li, and L. Deng, "Exploiting sparseness in deep neural networks for large vocabulary speech recognition," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, 2012, pp. 4409–4412.

[27] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Proc. Advances in Neural Information Processing Systems*, 2016, pp. 1379–1387.

[28] S. Han, J. Pool, S. Narang, H. Mao, S. Tang, E. Elsen, B. Catanzaro, J. Tran, and W. J. Dally, "DSD: Regularizing deep neural networks with dense-sparse-dense training flow," *arXiv preprint arXiv:1607.04381*, 2016.

[29] S. Lowel and W. Singer, "Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity," *Science*, vol. 255, no. 5041, p. 209, 1992.

[30] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Int. Conf. Machine Learning*, 2015, pp. 448–456.

[31] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[32] S. Anwar and W. Sung, "Compact deep convolutional neural networks with coarse pruning," *arXiv preprint arXiv:1610.09639*, 2016.

[33] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[34] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," *NIPS Workshop Autodiff*, 2017.

[35] C. J. Burges and B. Schölkopf, "Improving the accuracy and speed of support vector machines," in *Proc. Advances in Neural Information Processing Systems*, 1997, pp. 375–381.

[36] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[38] S. Srinivas and R. V. Babu, "Data-free parameter pruning for deep neural networks," *arXiv preprint arXiv:1507.06149*, 2015.

[39] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. Smola, L. Song, and Z. Wang, "Deep fried ConvNets," in *Proc. IEEE Int. Conf. Computer Vision*, 2015, pp. 1476–1483.

[40] M. D. Collins and P. Kohli, "Memory bounded deep convolutional networks," *arXiv preprint arXiv:1412.1442*, 2014.

[41] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Proc. Advances in Neural Information Processing Systems*, 2014, pp. 1269–1277.

[42] "PyTorch pre-trained VGG-16," https://download.pytorch.org/models/vgg16-397923af.pth, 2017.

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[45] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, "Densely connected convolutional networks," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, vol. 1, no. 2, 2017, pp. 3–12.

[46] G. Leisman, R. Mualem, and S. K. Mughrabi, "The neurological development of the child with the educational enrichment in mind," *Psicología Educativa*, vol. 21, no. 2, pp. 79–96, 2015.

Appendix

A Experimental details of LeNets

Table 4(a) and Table 4(b) show the smallest DNN models we could synthesize for LeNet-300-100 and LeNet-5, respectively. In these tables, Conv% refers to the percentage of *area-of-interest* over a full image for partial-area convolution, and Act% refers to the percentage of non-zero activations (the average percentage of neurons with non-zero output values per inference).

Table 4: Smallest synthesized LeNets

(a) LeNet-300-100 (error rate 1.29%)

Layer	#Weights	Act%	FLOPs
fc1	7032	46%	14.1K
fc2	718	71%	0.7K
fc3	94	100%	0.1K
Total	7844	N/A	14.9K

(b) LeNet-5 (error rate 0.77%)

Layer	#Weights	Conv%	Act%	FLOPs
conv1	74	39%	89%	45.2K
conv2	749	41%	57%	54.4K
fc1	4151	N/A	79%	4.7K
fc2	632	N/A	58%	1.0K
fc3	166	N/A	100%	0.2K
Total	5772	N/A	N/A	105K

B Experimental details of AlexNet

Table 5 illustrates the evolution of an AlexNet seed in the grow-and-prune paradigm as well as the final inference model. The AlexNet seed only contains 8.4M parameters. This number increases to 28.3M after the growth phase, and then decreases to 3.9M after the pruning phase. This final AlexNet-based DNN model only requires 325M FLOPs at a top-1 error rate of 42.76%.

Table 5: Synthesized AlexNet (error rate 42.76%)

Layers	#Parameters	#Parameters	#Parameters	Conv%	Act%	FLOPs
	Seed	Post-Growth	Post-Pruning			
conv1	7K	21K	17K	92%	87%	97M
conv2	65K	209K	107K	91%	82%	124M
conv3	95K	302K	164K	88%	49%	40M
conv4	141K	495K	253K	86%	48%	36M
conv5	105K	355K	180K	87%	56%	25M
fc1	5.7M	19.9M	1.8M	N/A	49%	2.0M
fc2	1.7M	5.3M	0.8M	N/A	47%	0.8M
fc3	0.6M	1.7M	0.5M	N/A	100%	0.5M
Total	8.4M	28.3M	**3.9M**	N/A	N/A	**325M**

Our models will be released soon.

C Experimental details of VGG-16

Table 6 illustrates the details of our final compact inference model based on the VGG-16 architecture. The final model only contains 4.6M parameters, which is 30.2× smaller than the original VGG-16.

Table 6: Synthesized VGG-16 (error rate 30.72%)

Layer	#Param	FLOPs	#Param	Act%	FLOPs
	Original VGG-16		Synthesized VGG-16		
conv1_1	2K	0.2B	1K	64%	0.1B
conv1_2	37K	3.7B	10K	76%	0.7B
conv2_1	74K	1.8B	21K	73%	0.4B
conv2_2	148K	3.7B	39K	76%	0.7B
conv3_1	295K	1.8B	79K	53%	0.4B
conv3_2	590K	3.7B	103K	57%	0.3B
conv3_3	590K	3.7B	110K	56%	0.4B
conv4_1	1M	1.8B	205K	37%	0.2B
conv4_2	2M	3.7B	335K	37%	0.2B
conv4_3	2M	3.7B	343K	35%	0.2B
conv5_1	2M	925M	350K	33%	48M
conv5_2	2M	925M	332K	32%	43M
conv5_3	2M	925M	331K	24%	41M
fc1	103M	206M	1.6M	38%	0.8M
fc2	17M	34M	255K	41%	0.2M
fc3	4M	8M	444K	100%	0.4M
Total	138M	30.9B	4.6M	N/A	3.6B