Wootz: A Compiler-Based Framework for Fast CNN Pruning via Composability*

Hui Guan
North Carolina State University
USA
hguan2@ncsu.edu

Xipeng Shen
North Carolina State University
USA
xshen5@ncsu.edu

Seung-Hwan Lim
Oak Ridge National Lab
USA
lims1@ornl.gov

Abstract

Convolutional Neural Networks (CNN) are widely used for Deep Learning tasks. CNN pruning is an important method to adapt a large CNN model trained on general datasets to fit a more specialized task or a smaller device. The key challenge is on deciding which filters to remove in order to maximize the quality of the pruned networks while satisfying the constraints. It is time-consuming due to the enormous configuration space and the slowness of CNN training.

The problem has drawn many efforts from the machine learning field, which try to reduce the set of network configurations to explore. This work tackles the problem distinctively from a programming systems perspective, trying to speed up the evaluations of the remaining configurations through *computation reuse* via a compiler-based framework. We empirically uncover the existence of *composability* in the training of a collection of pruned CNN models, and point out the opportunities for computation reuse. We then propose *composability-based CNN pruning*, and design a compression-based algorithm to efficiently identify the set of CNN layers to pre-train for maximizing their reuse benefits in CNN pruning. We further develop a compiler-based framework named Wootz, which, for an arbitrary CNN, automatically generates code that builds a Teacher-Student scheme to materialize composability-based pruning. Experiments show that network pruning enabled by Wootz shortens the state-of-art pruning process by up to 186X while producing significantly improved pruning results.

CCS Concepts • **Computing methodologies** **Neural networks**; • **Software and its engineering** **Compilers**;

Keywords CNN, network pruning, compiler, composability

ACM Reference Format:
Hui Guan, Xipeng Shen, and Seung-Hwan Lim. 2019. Wootz: A Compiler-Based Framework for Fast CNN Pruning via Composability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314652

1 Introduction

As a major class of Deep Neural Networks (DNN), Convolutional Neural Networks (CNN) are important for a broad range of deep learning tasks, from face recognition [36], to image classification [32], object detection [53], human pose estimation [61], sentence classification [29], and even speech recognition and time series data analysis [37]. The core of a CNN usually consists of many convolutional layers, and most computations at a layer are convolutions between its neuron values and a set of *filters* on that layer. A filter consists of a number of *weights* on synapses, as Figure 1 (a) illustrates.

CNN pruning is a method that reduces the size and complexity of a CNN model by removing some parts, such as weights or filters, of the CNN model and then retraining the reduced model, as Figure 1 (b) illustrates. It is an important approach to adapting large CNNs trained on general datasets to meet the needs of more specialized tasks [60, 66]. An example is to adapt a general image recognition network trained on a general image set (e.g., ImageNet [54]) such that the smaller CNN (after retraining) can accurately distinguish different bird species, dog breeds, or car models [41, 43, 47, 66]. Compared to designing a CNN from scratch for each specific task,

*We thank Louisnoel Pouchet and the anonymous PLDI reviewers for the helpful comments. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1525609, CNS-1717425, CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314652

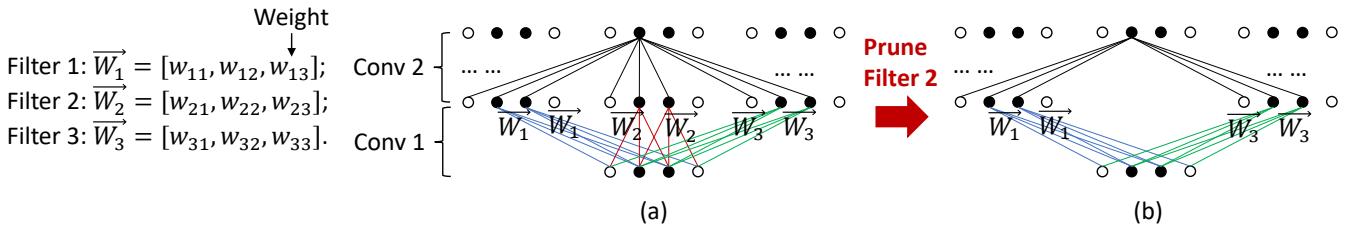


Figure 1. CNN and CNN Pruning. Conv1 and Conv2 are the first two consecutive convolutional layers in the CNN.

CNN pruning is an easier and more effective way to achieve a high-quality network [15, 41, 47, 51, 60]. Moreover, CNN pruning is an important method for fitting a CNN model on a device with limited storage or computing power [16, 65].

The most commonly used CNN pruning is filter-level pruning, which removes a set of unimportant filters from each convolutional layer. The key problem for filter-level pruning is how to determine the set of filters to remove from each layer to meet users' needs: The entire configuration space is as large as $2^{|W|}$ (W for the entire set of filters) and it often takes hours to evaluate just one configuration (i.e., training the pruned network and then testing it).

The problem is a major barrier for timely solution delivery in Artificial Intelligence (AI) product development. The prior efforts have been, however, mostly from the machine learning community [19, 25, 39, 43, 47]. They leverage DNN algorithm-level knowledge to reduce the enormous configuration space to a smaller space (called *promising subspace*) that is likely to contain a good solution, and then evaluate these remaining configurations to find the best.

Although these prior methods help mitigate the problem, network pruning remains a time-consuming process. One reason is that, despite their effectiveness, no prior techniques can guarantee the inclusion of the desirable configuration in a much reduced subspace. As a result, to decrease the risk of missing the desirable configuration, practitioners often end up with a still quite large subspace of network configurations that takes days for many machines to explore. It is also quite often true that modifications need to make to the CNN models, datasets, or hardware settings throughout the development process of an AI product; each of the changes could make the result of a CNN pruning obsolete and call for a rerun of the entire pruning process. Our conversations with AI product developers indicate that the long pruning process is one of the major hurdles for shortening the time to market AI products.

This study distinctively examines the problem from the programming systems perspective. Specifically, rather than improving the attainment of promising subspace as all prior work focuses on, we try to drastically speed up the evaluations of the remaining configurations in the promising subspace through cross-network *computation reuse* via a compiler-based framework, a direction complementary to prior solutions. We achieve the goal through three-fold innovations.

First, we empirically uncover the existence of *composability* in the training of a collection of pruned CNN models, and reveal the opportunity that the composability creates for saving computations in CNN pruning. The basic observation that leads to this finding is that two CNN networks in the promising subspace often differ in only some layers. In the current CNN pruning methods, the two networks are both trained from scratch and then tested for accuracy. A question we ask is whether the training results of the common layers can be reused across networks to save some training time. More generally, we view the networks in a promising subspace as compositions of a set of building blocks (a *block* is a sequence of CNN layers). The question is if we first pre-train (some of) these building blocks and then assemble them into the to-be-explored networks, can we shorten the evaluations of these networks and the overall pruning process? Through a set of experiments, we empirically validate the hypothesis, based on which, we propose *composability-based CNN pruning* to capture the idea of reusing pre-trained blocks for pruning (§ 3).

Second, we propose a novel *hierarchical compression-based algorithm*, which, for a given CNN and promising subspace, efficiently identifies the set of blocks to pre-train to maximize the benefits of computation reuse. We prove that identifying the optimal set of blocks to pre-train is NP-hard. Our proposed algorithm provides a linear-time heuristic solution by applying Sequitur [49], a hierarchical compression algorithm, to the CNN configurations in the promising subspace (§ 5).

Finally, based on all those findings, we developed Wootz[1], the first compiler-based framework that, for an arbitrary CNN (in Caffe Prototxt format) and other inputs, automatically generates TensorFlow code to build Teacher-Student learning structures to materialize composability-based CNN pruning (§ 4,§ 6).

We evaluate the technique on a set of CNNs and datasets with various target accuracies. For ResNet-50 and Inception-V3, it shortens the pruning process by up to 186.7X and 30.2X respectively. Meanwhile, the models it finds are significantly more compact (up to 70% smaller) than those by the default pruning scheme for the same target accuracy (§ 7).

[1]The name is after Wootz steel, the legendary pioneering steel alloy developed in the 6th century BC; Wootz blades give the sharpest cuts.

2 Background on CNN Pruning

This section gives some background important for following the rest of the paper. For a CNN with L convolutional layers, let $W_i = \{W_i^j\}$ represent the set of filters on its i-th convolutional layer, and W denote the entire set of filters (i.e., $W = \cup_{i=1}^{L} W_i$.) For a given training dataset D, a typical objective of CNN pruning is to find the smallest subset of W, denoted as W', such that the accuracy reachable by the pruned network $f(W', D)$ (after being re-trained) has a tolerable loss (a predefined constant α) from the accuracy by the original network $f(W, D)$. Besides space, the pruning may seek for some other objectives, such as maximizing the inference speed [67], or minimizing the amount of computations [19] or energy consumption [65].

The optimization problem is challenging because the entire network configuration space is as large as $2^{|W|}$ and it is time-consuming to evaluate a configuration, which involves the re-training of the pruned CNN. Previous work simplifies the problem as identifying and removing the least important filters. Many efficient methods on finding out the importance of a filter have been proposed [20, 25, 39, 42, 43, 47].

The pruning problem then becomes *to determine how many least important filters* to remove from each convolutional layer. Let γ_i be the number of filters removed from the i-th layer in a pruned CNN and $\boldsymbol{\gamma} = (\gamma_1, \cdots, \gamma_L)$. Each $\boldsymbol{\gamma}$ specifies a **configuration**. The size of the configuration space is still combinatorial, as large as $\prod_{i=1}^{L} |\Gamma_i|$, where Γ_i is the number of choices γ_i can take.

Prior efforts have concentrated on how to reduce the configuration space to a promising subspace [4, 19, 23]. But CNN training is slow and the reduced space still often takes days to explore. This work focuses on a complementary direction, accelerating the examinations of the promising configurations.

3 Composability-Based CNN Pruning: Idea and Challenges

The fundamental reason for Wootz to produce large speedups for CNN pruning is its effective capitalization of computation reuse in CNN pruning, which is built on the *composability in CNN pruning* empirically unveiled in this study. Two pruned networks in a promising subspace often differ in only some of the layers. The basic idea of *composability-based CNN pruning* is to reuse the training results of the common layers across the pruned networks. Although the idea may look straightforward, to our best knowledge, no prior CNN pruning work has employed such reuse, probably due to a series of open questions and challenges:

- First, there are bi-directional data dependencies among the layers of a CNN. In CNN training, for an input image, there is a forward propagation that uses a lower layer's output, which is called **activation maps**, to compute the activation maps of a higher layer; it is followed by a backward propagation, which updates the weights of a lower layer based on the errors computed with the higher layer's activation maps. As a result of the bi-directional dependencies, even just one-layer differences between two networks could cause very different weights to be produced for a common (either higher or lower) layer in the two networks. Therefore, it remains unclear whether the training results of a common layer could help with the training of different networks.
- Second, if a pre-trained layer could help, it is an open question how to maximize the benefits. A pre-trained sequence of consecutive layers may have a larger impact than a single pre-trained layer does on the whole network, but it may also take more time to produce and has fewer chances to be reused. How to determine which sets of layers or sequences of layers to pre-train to maximize the gains has not been explored before.
- Third, how to pre-train just a piece of a CNN? The standard CNN back propagation training algorithm uses input labels as the ground truth to compute errors of the current network configurations and adjust the weights. If we just want to train a piece of a CNN, what ground truth should we use? What software architecture should be built to do the pre-training and do it efficiently? g
- Fourth, existing DNN frameworks support only the standard DNN training and inference. Users have to write code to do CNN pruning themselves, which is already complicated for general programmers. It would add even more challenges to ask them to additionally write the code to pre-train CNN pieces, and then reuse the results during the evaluations of the networks.

For the first question, we conduct a series of experiments on 16 large CNNs (four popular CNN models trained on four datasets). Section 7.2 reports the details; here we just state the key observations. Pre-trained layers bring a network to a much improved starting setting, making the initial accuracies of the network 50-90% higher than the network without pre-trained layers. That leads to 30-100% savings of the training time of the network. Moreover, it helps the network converge to a significantly higher level of accuracy (by 1%-4%). These findings empirically confirm the potential of *composability-based CNN pruning*.

To effectively materialize the potential, we have to address the other three challenges. Wootz offers the solution.

4 Overview of Wootz Framework

This section gives an overview of Wootz. Wootz is a software framework that automatically enables composability-based CNN pruning. As Figure 2 shows, its input has four parts:

- The to-be-pruned CNN model, written in Caffe Prototxt (with a minor extension), which is a user-friendly text format (from Caffe) for CNN model specifications [27].
- The promising subspace that contains the set of pruned networks configurations worth exploring, following the

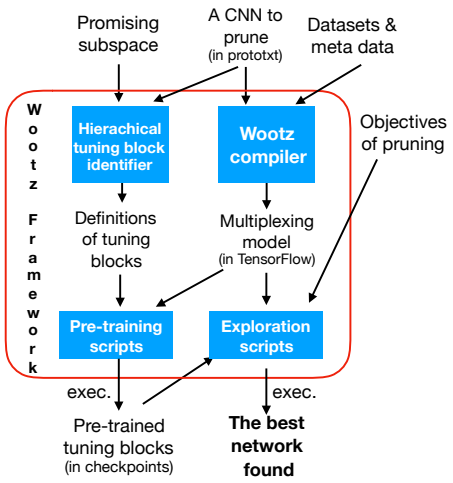


Figure 2. Overview of Wootz Framework.

```
''' An example of a promising subspace specification that contains two
configurations. Each number is a pruning rate for a convolutional
layer. For example, the first configuration means the first and third
layers are pruned with pruning rate 0.3, the second and fourth layers
are not pruned. '''
configs=[[0.3, 0, 0.3, 0], [0.5, 0, 0.3, 0]]

''' The configurations should be either a Numpy array or a python list
that can be serialized using Pickle as below. Users only need to
provide configs_path to the compiler. '''
pickle.dump(configs, open(configs_path, "wb"))
```

(a) Promising subspace specifications.

```
# Format:
[min, max] [ModelSize, Accuracy]
constraint [ModelSize, Accuracy] [<, >, <=, <=] [Value]

# Example:
min ModelSize
constraint Accuracy > 0.8
```

(b) Pruning objectives specifications.

Figure 3. Formats for the specifications of *promising subspaces* (a) and *pruning objectives (b)*.

format in Figure 3 (a). The subspace may come from the user or some third-party tools that reduce the configuration space for CNN pruning [4, 19, 23].

- The dataset for training and testing, along with some meta data on the training (e.g., learning rates, maximum training steps), following the format used in Caffe Solver Prototxt [1].
- The objectives of the CNN pruning, including the constraints on model size or accuracy, following the format shown in Figure 3 (b).

The Wootz framework consists of four main components as shown in Figure 2. (1) The *hierarchical tuning block identifier* tries to define the set of *tuning blocks*. A **tuning block** is a sequence of pruned consecutive CNN layers taken as a unit for pre-training. Suitable definitions of *tuning blocks* help maximize reuse while minimizing the pre-training overhead. (2) From the given CNN model specified in Prototxt, the *Wootz compiler* generates a *multiplexing model*, which is a function written in TensorFlow that, when invoked, specifies the structure of the full to-be-pruned CNN model, the network structure—which implements a Teacher-Student scheme—for pre-training tuning blocks, or pruned networks assembled with pre-trained tuning blocks, depending on the arguments the function receives. (3) The *pre-training scripts* are some generic Python functions that, when run, pre-train each tuning block based on the outputs from the first two components of Wootz. (4) The final component, *exploration scripts*, explores the promising pruned networks assembled with the pre-trained tuning blocks. The exploration of a network includes first fine-tuning the entire network and then testing it for accuracy. The exploration order is automatically picked by the *exploration scripts* based on the pruning objectives to produce the best network as early as possible. Both the *pre-training scripts* and the *exploration scripts* can run on one machine or multiple machines in a distributed environment through MPI.

Wootz is designed to help pruning methods that have their promising subspace known at front. There are methods that do not provide the subspace explicitly [68]. They, however, still need to tune the pruning rate for each layer and the exploration could also contain potentially avoidable computations. Extending Wootz to harvest those opportunities is a direction worth future exploration.

Next, we explain the *hierarchical tuning block identifier* in § 5, and the other components in § 6.

5 Hierarchical Tuning Block Identifier

Composability-based CNN pruning faces a trade-off between the pre-training cost and the time savings the pre-training results bring. The tradeoff depends on the definitions of the unit for pre-training, that is, the definition of *tuning blocks*. A *tuning block* is a unit for pre-training; it consists of a sequence of consecutive CNN layers pruned at certain rates. It can have various sizes, depending on the number of CNN layers it contains. The smaller it is, the less pre-training time it takes and the more reuses it tends to have across networks, but at the same time, its impact to the training time of a network tends to be smaller.

So for a given promising subspace of networks, a question for composability-based CNN pruning is how to define the best sets of tuning blocks. The solution depends on the appearing frequencies of each sequence of layers in the subspace, their pre-training times, and the impact of the pre-training results on the training of the networks. For a clear understanding of the problem and its complexity, we define *optimal tuning block definition problem* as follows.

Optimal Tuning Block Definition Problem Let A be a CNN consisting of L layers, represented as $A_1 \cdot A_2 \cdot A_3 \cdot \ldots \cdot A_L$, where $\cdot$ stands for layer stacking and A_i stands for the i-th

layer (counting from input layer). $C = \{A^{(1)}, A^{(2)}, \ldots, A^{(N)}\}$ is a set of N networks that are derived from filter pruning of A, where $A^{(n)}$ represents the n-th derived network from A, and ${A_i}^{(n)}$ stands for the i-th layer of $A^{(n)}$, $i = 1, 2, \ldots, L$.

Optimal tuning block definition problem is to come up with a set of tuning blocks $B = \{B_1, B_2, \ldots, B_K\}$ such that the following two conditions are met:

1. Every B_k, $k = 1, 2, \cdots, K$, is part of a network in C—that is, $\forall\ B_k, \exists\ A^{(n)}, n \in \{1, 2, \cdots, N\}$, such that $B_k = {A_l}^{(n)} \cdot {A_{l+1}}^{(n)} \cdot \ldots \cdot {A_{l+b_k-1}}^{(n)}, 1 \le l \le L - b_k + 1$, where b_k is the number of layers contained in B_k.
2. B is an optimal choice—that is, $\arg\min_B(\sum_{k=1}^{K} T(B_k) + \sum_{n=1}^{N} T(A^{(n,B)}))$, where, $T(B_k)$ is the time taken to pre-train block B_k, and $T(A^{(n,B)})$ is the time taken to train $A^{(n,B)}$ to reach the accuracy objective[2]; $A^{(n,B)}$ is the blocked-trained version of $A^{(n)}$ with B as the tuning blocks.

A restricted version of the problem is that only a predefined set of pruning rates (e.g., {30%, 50%, 70%}) are used when pruning a layer in A to produce the set of pruned networks in C—which is a common practice in filter pruning.

Even this restricted version is NP-hard, provable through a reduction of the problem to the classic *knapsack problem* [22] (detailed proof omitted for sake of space.) A polynomial-time solution is hence in general hard to find, if ever possible. The NP-hardness motivates our design of a heuristic algorithm, which does not aim to find the optimal solution but to come up with a suitable solution efficiently. The algorithm does not use the training time as an explicit objective to optimize but focuses on layer reuse. It is a hierarchical compression-based algorithm, described next.

Hierarchical Compression-Based Algorithm Our algorithm leverages Sequitur [49] to efficiently identify the frequent sequences of pruned layers in the network collection C. As a linear-time hierarchical compression algorithm, Sequitur infers a hierarchical structure from a sequence of discrete symbols. For a given sequence of symbols, it derives a context-free grammar (CFG), with each rule in the CFG reducing a repeatedly appearing string into a single rule ID. Figure 4 gives an example. Its top part shows the concatenated sequence of layers of four networks pruned at various rates; the subscripts of the numbers indicate the pruning rate, that is, the fraction of the least important filters of a layer that are removed. The lower part in Figure 4 shows the CFG produced by Sequitur on the string. A full expansion of rule $r0$ would give the original string. The result can also be represented as a Directed Acyclic Graph (DAG) as the right graph in Figure 4 shows with each node corresponding to one rule.

[2]In our framework, $T(x)$ is not statically known or approximated, but instead explicitly computed (via training) for each x (i. e, B_k or $A^{(n,B)}$).

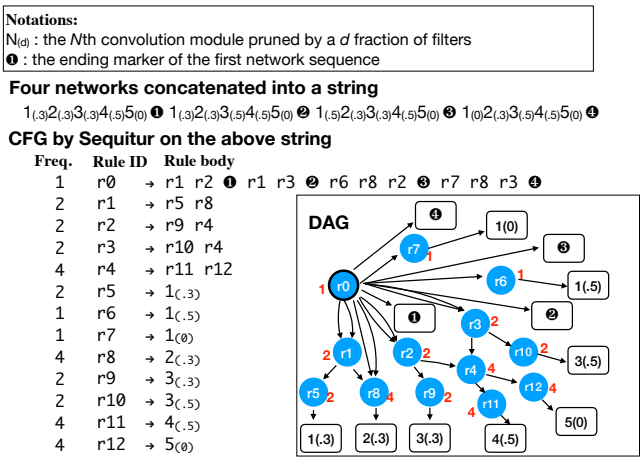


Figure 4. Sequitur applies to a concatenated sequence of layers of four networks pruned at rates: 0%, 30%, 50%.

Applying Sequitur to the concatenated sequence of all networks in the *promising subspace*, our *hierarchical compression-based algorithm* gets the corresponding CFG and the DAG. Let R be the collection of all the rules in the CFG, and S be the solution to the tuning block identification problem which is initially empty. Our algorithm then heuristically fills S with subsequences of CNN layers (represented as rules in the CFG) that are worth pre-training.

It does it based on the appearing frequencies of the rules in the *promising subspace* and their sizes (i.e., the number of layers a rule contains). It employs two heuristics: (1) A rule cannot be put into S if it appears in only one network (i.e., its appearing frequency is one); (2) a rule is preferred over its children rules only if that rule appears as often as its most frequently appearing descendant.

The first heuristic is to ensure that the pre-training result of the sequence can benefit more than one network. The second heuristic is based on the following observation: A pre-trained sequence typically has a larger impact than its subsequences all together have on the quality of a network; however, the extra benefits are usually modest. For instance, a ResNet CNN network assembled from 4-block long pre-trained sequences has an initial accuracy of 0.716, 3.1% higher than the same network but assembled from 1-block long pre-trained sequences. The higher initial accuracy helps save extra training steps (epochs) for the network, but the saving is limited (up to 20% of the overall training time). Moreover, a longer sequence usually has a lower chance to be reused. For these reasons, we employ the aforementioned heuristics to help keep S small and hence the pre-training overhead low while still achieving a good number of reuses.

Specifically, the algorithm takes a post-order (children before parent) traversal of the DAG that Sequitur produces. (Before that, all edges between two nodes on the DAG are

combined into one edge.) At a node, it checks its frequency. If it is greater than one, it checks whether its frequency equals the largest frequency of its children. If so, it marks itself as a potential tuning block, unmarks its children, and continues the traversal. Otherwise, it puts a "dead-end" mark on itself, indicating that it is not worth going further up in the DAG from this node. When the traversal reaches the root of the DAG or has no path to continue, the algorithm puts all the potential tuning blocks into S as the solution and terminates.

Note that a side product from the process is a *composite vector* for each network in the promising subspace. As a tuning block is put into S, the algorithm, by referencing the CFG produced by Sequitur, records the ID of the tuning block in the *composite vectors* of the networks that can use the block. *Composite vectors* will be used in the global fine-tuning phase as described in the next section.

The *hierarchical compression-based algorithm* is designed to be simple and efficient. More detailed modeling of the time savings and pre-training cost of each sequence for various CNNs could potentially help yield better definitions of tuning blocks, but it would add significant complexities and runtime overhead. Our exploration in § 7.3 shows that the *hierarchical compression-based algorithm* gives a reasonable trade-off.

6 Mechanisms for Composability-Based Pruning and Wootz Compiler

The core operations in Composability-based CNN pruning includes pre-training of tuning blocks, and global fine-tuning of networks assembled with the pre-trained blocks. This section first explains the mechanisms we have designed to support these operations efficiently, and then describes the implementation of Wootz compiler and scripts that automatically materializes the mechanisms for an arbitrary CNN.

6.1 Mechanisms

Pre-Training of Tuning Blocks The standard CNN back propagation training algorithm uses input labels as the ground truth to compute errors of the current network and adjusts the weights iteratively. To train a tuning block, the first question is what ground truth to use to compute errors. Inspired by Teacher-Student networks [5, 7, 21], we adopt a similar Teacher-Student mechanism to address the problem.

We construct a network structure that contains both the pruned block to pre-train and the original full CNN model. They are put side by side as shown in Figure 5 (a) with the input to the counterpart of the tuning block in the full model also flowing into the pruned tuning block as its input, and the output activation map of the counterpart block flowing into the pruned tuning block as the "ground truth" of its output. When the standard back propagation algorithm is applied to the tuning block in this network structure, it effectively minimizes the reconstruction error between the output activation maps from the pruned tuning block and the ones from its unpruned counterpart in the full network. (In CNN pruning, the full model has typically already been trained beforehand to perform well on the datasets of interest.) This design essentially uses the full model as the "teacher" to train the pruned tuning blocks. Let O_k and O'_k be the vectorized output activation maps from the unpruned and pruned tuning block, and W'_k be the weights in the pruned tuning block. The optimization objective in this design is:

$$\min_{W'_k} \frac{1}{|O_k|} \|O_k - O'_k\|_2^2. \tag{1}$$

Only the parameters in the pruned tuning block are updated in this phase to ensure the pre-trained blocks are reusable.

This Teacher-Student design has three appealing properties. First, it addresses the missing "ground truth" problem for tuning block pre-training. Second, as the full CNN model runs along with the pre-training of the tuning blocks, it provides the inputs and "ground truth" for the tuning blocks on the fly; there is no need to save to storage the activation maps which can be space-consuming considering the large number of input images for training a CNN. Third, the structure is friendly for concurrently pre-training multiple tuning blocks. As Figure 5 (b) shows, connections can be added between the full model and multiple pruned blocks; the pre-training of these blocks can then happen in one run, and the activation maps produced by a block in the full model can be seamlessly reused across the pre-training of multiple pruned blocks.

Global Fine-Tuning The local training phase outputs a bag of pre-trained pruned tuning blocks, as shown in Figure 5 (c) (tuning blocks in the original network could also be included). At the beginning of the *global fine-tuning* phase is an assembly step, which, logically, assembles these training blocks into each of the networks in the promising subspace. Physically, this step just needs to initialize the pruned networks in the promising subspace with the weights in the corresponding tuning blocks. We call the resulting network *a block-trained network*. Recall that one of the side products of the tuning block identification step is a *composite vector* for each network which records the tuning blocks the network can use; these vectors are used in this assembly step. Figure 5 (d) gives a conceptual illustration; three networks are assembled with three different sets of pre-trained tuning blocks.

As a pruned block with only a subset of parameters has a smaller model capacity, a *global fine-tuning step* is required to further recover the accuracy performance of a block-trained network. This step runs the standard CNN training on the *block-trained networks*. All the parameters in the networks are updated during the training. Compared with training a default pruned network, fine-tuning a block-trained network usually takes much less training time as the network starts with a much better set of parameter values as shown in § 7.

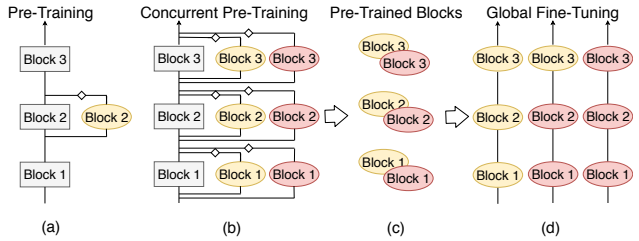


Figure 5. Illustration of composability-based network pruning. Eclipses are pruned tuning blocks; rectangles are original tuning blocks; diamonds refer to the activation map reconstruction error. Different colors of pruned tuning blocks correspond to different pruning options.

6.2 Wootz Compiler and Scripts

Wootz compiler and scripts offer an automatic way to materialize the mechanisms for an arbitrary CNN model. The proposed method is not restricted to a particular DNN framework, though we demonstrate its ability using TensorFlow.

We first provide brief background on TensorFlow [2] that is closely relevant to this part. TensorFlow offers a set of APIs for defining, training, and evaluating a CNN. To specify the structure of a CNN, one needs to call APIs in a Python script, which arranges a series of operations into a computational graph. In a TensorFlow computational graph, nodes are operations that consume and produce tensors, and edges are tensors that represent values flowing through the graph. CNN model parameters are held in TensorFlow variables, which represent tensors whose values can be changed by operations. Because a CNN model can have hundreds of variables, it is a common practice to name variables in a hierarchical way using *variable scopes* to avoid name clashes. A popular option to store and reuse the parameters of CNN model is TensorFlow *checkpoints*. Checkpoints are binary files that map variable names to tensor values. The tensor value of a variable can be restored from a checkpoint by matching the variable name.

TensorFlow APIs with other assistant libraries (e.g., Slim [57]) offer conveniences for standard CNN model training and testing, but *not* for CNN pruning, let alone *composability-based pruning*. Asking a general programmer to implement *composability-based pruning* in TensorFlow for each CNN model would add tremendous burdens on the programmer. She would need to write code to identify tuning blocks, create TensorFlow code to implement the customized CNN structures to pre-train each tuning block, generate checkpoints, and use them when creating the block-trained CNN networks for global fine-tuning.

Wootz compiler and scripts mitigate the difficulty by automating the process. The fundamental motivating observation is that the codes for two different CNN models follow the same pattern. Differences are mostly on the code specifying the structure of the CNN models (both the original and the extended for pre-training and global fine tuning). The idea is to build code templates and use the compiler to automatically adapt the templates based on the specifications of the models.

Multiplexing Model An important decision in our design of Wootz is to take Prototxt as the format of an input to-be-pruned CNN model. Because our tool has to derive code for pre-training and fine-tuning of the pruned models, our compiler would need to analyze the TensorFlow code from users, which could be written in various ways and complex to analyze. Prototxt has a clean fixed format. It is easy for programmers to write and simple for our compiler to analyze.

Given a to-be-pruned CNN model specified in Prototxt, the compiler first generates the *multiplexing model*, which is a piece of TensorFlow code defined as a Python function. It is multiplexing in the sense that an invocation of the code specifies the structure of the original CNN model, or the structure for pre-training, or the global fine tuning model; which of the three modes is used at an invocation of the multiplexing model is determined by one of its input arguments, *mode_to_use*. The multiplexing design allows easy code reuse as the three modes share much common code for model specifications. Another argument, *prune_info*, conveys to the multiplexing model the pruning information, including the set of tuning blocks to pre-train in this invocation and their pruning rates.

The compiler-based code generation needs to provide mainly two-fold support. It needs to map CNN model specifications in Prototxt to TensorFlow APIs. Our implementation, specifically, generates calls to TensorFlow-Slim API [55] to add various CNN layers based on the parsing results of the Prototxt specifications. The other support is to generate the code to also specify the derived network structure for pre-training each tuning block contained in *prune_info*. Note that the layers contained in a tuning block are the same as a section of the full model except for the number of filters in the layers and the connections flowing into the block. The compiler hence emits code for specifying each of the CNN layers again, but with connections flowing from the full network, and sets the "depth" argument of the layer-adding API call (a TensorFlow-Slim API [55]) with the info retrieved from *prune_info* such that the layer's filters can change with *prune_info* at different calls of the multiplexing model. In addition, the compiler encloses the code with condition checks to determine, based on *prune_info*, at an invocation of the multiplexing model whether the layer should be actually added into the network for pre-training. The code generation for the global fine-tuning is similar but simpler. In such a form, the generated *multiplexing model* is adaptive to the needs of different modes and the various pruning settings.

Once the multiplexing model is generated, it is registered at the nets factory in Slim Model Library [57] with its unique model name. The nets factory is part of the functional programming Slim Model Library is based on. It contains a dictionary mapping a model name to its corresponding model

function for easy retrieval and use of the models in other programs.

Pre-Training Scripts The pre-training scripts contain a generic pre-training Python code and a wrapper that is adapted from a Python template by the Wootz Compiler to the to-be-pruned CNN model and meta data. The pre-training Python code retrieves the multiplexing model from nets factory based on the registered name, and repeatedly invokes the model function with the appropriate arguments, with each call generating one of the pre-train networks. After defining the loss function, it launches a TensorFlow session to run the pre-training process.

The wrapper calls the pre-training Python code with required arguments such as model name and the set of tuning blocks to train. As the tuning blocks coexisting in a pruned network cannot have overlapping layers, one pruned network can only enable the training of a limited set of tuning blocks. We design a simple algorithm to partition the entire set of tuning blocks returned by the Hierarchical Tuning Block Identifier into groups. The pre-training Python script is called to train only one group at a time. The partition algorithm is as follows:

```
Inputs: B {the entire set of tuning blocks}
Outputs: G {the set of groups of tuning blocks}
B.sort() {sort by the contained lowest conv layers}
G = {{B[0]}}
for b ∈ B[1 :] do
  for g ∈ G do
    any([overlap(b, e) for e in g])? G.add({b}):g.add(b)
```

The meta data contains the training configurations such as dataset name, dataset directory, learning rate, maximum training steps and batch size for pre-training of tuning blocks. The set of options to configure are predefined, similar to the Caffe Solver Prototxt [1]. The compiler parses the meta data and specifies those configurations in the wrapper.

Executing the wrapper produces pre-trained tuning blocks that are stored as TensorFlow checkpoints. The mapping between the checkpoint files and trained tuning blocks are also recorded for the model variable initialization in the global fine-tuning phase. The pre-training script can run on a single node or multiple nodes in parallel to concurrently train multiple groups through MPI.

Exploration Scripts Exploration scripts contain a generic global fine-tuning Python code and a Python-based wrapper. The global fine-tuning code invokes the multiplexing model to generate the pruned network according to the configuration to evaluate. It then initializes the network through the checkpoints produced in the pre-train process and launches a TensorFlow session to train the network.

In addition to feeding the global fine-tuning Python code with required arguments (e.g. the configuration to evaluate), the Python-based wrapper provides code to efficiently explore the promising subspace. The order of the exploration is dynamically determined by the objective function.

The compiler first parses the file that specifies the objective of pruning to get the metric that needs to be minimized or maximized. The order of explorations is determined by the corresponding *MetricName*. In case the *MetricName* is *ModelSize*, the best exploration order is to start from the smallest model and proceed to larger ones. If the *MetricName* is *Accuracy*, the best exploration order is the opposite order as a larger model tends to give a higher accuracy.

To facilitate concurrent explorations on multiple machines, the compiler generates a task assignment file based on the order of explorations and the number of machines to use specified by the user in the meta data. Let c be the number of configurations to evaluate and p be the number of machines available, the i-th node will evaluate the $i + p * j$-th smallest (or largest) model, where $0 \leq j \leq \lfloor c/p \rfloor$.

7 Evaluations

We conduct a set of experiments to examine the efficacy of the Wootz framework. Our experiments are designed to answer the following three major questions: 1) Whether pre-training the tuning blocks of a CNN helps the training of that CNN reach a given accuracy sooner? We refer to it as the *composability hypothesis* as its validity is the prerequisite for the *composability-based CNN pruning* to work. 2) How much benefits we could get from *composability-based CNN pruning* on both the speed and the quality of network pruning while counting the pre-training overhead? 3) How much extra benefits we could get from *hierarchical tuning block identifier*?

We first describe the experiment settings (datasets, learning rates, machines, etc.) in § 7.1, then report our experiment results in § 7.2 and § 7.3 to answer each of the three questions.

7.1 Experiment Settings

Models and Datasets Our experiments use four popular CNN models: ResNet-50 and ResNet-101, as representatives of the Residual Network family [18], and Inception-V2 and Inception-V3, as representatives of the Inception family [59]. They have 50, 101, 34, 48 layers respectively. These models represent a structural trend in CNN designs, in which, several layers are encapsulated into a generic module of a fixed structure—which we call *convolution module*—and a network is built by stacking many such modules together. Such CNN models are holding the state-of-the-art accuracy in many challenging deep learning tasks. The structures of these models are described in input Caffe Prototxt[3] files and then converted to the multiplexing models by the Wootz compiler.

For preparation, we adapt the four CNN models already trained on a general image dataset ImageNet [54] (ILSVRC

[3]We add to Prototxt a new construct "module" for specifying the boundaries of *convolution modules*.

2012) to each of four specific image classification tasks with the domain-specific datasets, Flowers102 [50], CUB200 [63], Cars [31], and Dogs [28]. It gives us 16 trained full CNN models. The accuracy of the trained ResNets and Inceptions on the test datasets are listed in columns *Accuracy* in Table 1.

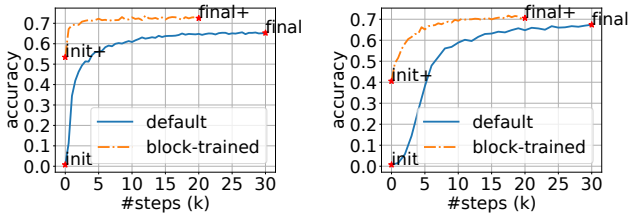
The four datasets for CNN pruning are commonly used in fine-grained recognition [14, 24, 30, 47, 69], which is a typical usage scenario of CNN pruning. Table 1 reports the statistics of the four datasets, including the data size for training (*Train*), the data size for testing (*Test*), and the number of classes (*Classes*). For all experiments, network training is performed on the training sets while accuracy results are reported on the testing sets.

Baseline for Comparison In CNN pruning, the full CNN model to prune has typically been already trained on the datasets of interest. When filters in the CNN are pruned, a new model with fewer filters is created, which inherits the remaining parameters of the affected layers and the unaffected layers in the full model. The promising subspace consists of such models. The *baseline approach* trains these models as they are. Although there are prior studies on accelerating CNN pruning, what they propose are all various ways to reduce the configuration space to a promising subspace. To the best of our knowledge, when exploring the configurations in the promising subspace, they all use the *baseline approach*. As our method is the first for speeding up the exploration of the promising space, we compare our results with those from the *baseline approach*.

We refer to a pruned network in the baseline approach a *default network* while the one initialized with pre-trained tuning blocks in our method a *block-trained network*.

Promising Subspace The 16 trained CNNs contain up to hundreds of convolutional layers. A typical practice is to use the same pruning rate for the convolutional layers in one *convolution module*. We adopt the same strategy. The importance of a filter is determined by its ℓ_1 norm as previous work [39] proposes. Following prior CNN pruning practice [39, 43], the top layer of a convolution module is kept unpruned; it helps ensure the dimension compatibility of the module.

There are many ways to select the promising subspace, i.e., the set of promising configurations worth evaluating. Previous works select configurations either manually [39, 43] or based on reinforcement learning with various rewards or algorithm design [4, 19]. As that is orthogonal to the focus of this work, to avoid bias from that factor, our experiment forms the promising spaces through random sampling [6] of the entire pruning space. A promising space contains 500 pruned networks, whose sizes follow a close-to-uniform distribution. In the experiments, the pruning rate for a layer can be one of $\Gamma = \{30\%, 50\%, 70\%\}$.



(a) ResNet-50 **(b)** Inception-V3

Figure 6. Accuracy curves of the *default* and *block-trained* networks on dataset CUB200. Each network has 70% least important filters pruned at all convolution modules.

Objective of Pruning There are different pruning objectives including minimizing model size, computational cost, memory footprint or energy consumption. Even though an objective of pruning affects the choice of the best configuration, all objectives require the evaluation of the set of promising configurations. Our composability-based CNN pruning aims at accelerating the training of a set of pruned networks and thus can work with any objective of pruning.

For the demonstration purpose, we set the objective of pruning as finding the smallest network (min ModelSize) that meets a given accuracy threshold (Accuracy <= *thr_acc*). We get a spectrum of *thr_acc* values by varying the accuracy drop rate α from that of the full model from -0.02 to 0.08. We include negative drop rates because it is possible that pruning makes the model more accurate.

Meta Data on Training The meta data on the training in both the baseline approach and our composability-based approach are as follows. Pre-training of tuning blocks takes 10,000 steps for all ResNets, with a batch size 32, a fixed learning rate 0.2, and a weight decay 0.0001; it takes 20,000 steps for all Inceptions, with batch size 32, a fixed learning rate 0.08, and a weight decay 0.0001. The global fine-tuning in the composability-based approach and the network training in the baseline approach uses the same training configurations: max number of steps 30,000, batch size 32, weight decay 0.00001, fixed learning rate 0.001[4].

All the experiments are performed with TensorFlow 1.3.0 on machines each equipped with a 16-core 2.2GHz AMD Opteron 6274 (Interlagos) processor, 32 GB of RAM and an NVIDIA K20X GPU with 6 GB of DDR5 memory. One network is trained on one GPU.

7.2 Validation of the Composability Hypothesis

We first present empirical validations of the *composability hypothesis* (i.e., pre-training tuning blocks helps CNN reach

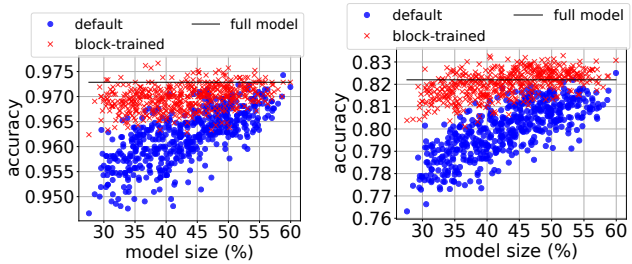
[4]We experimented with other learning rates and dynamic decay schemes. No single choice works best for all networks. We decided on 0.001 as it gives the overall best results for the baseline approach.

Table 1. Dataset statistics.

	Dataset	Size			Classes	Accuracy			
		Total	Train	Test		ResNet-50	ResNet-101	Inception-V2	Inception-V3
General	ImageNet [54]	1,250,000	1,200,000	50,000	1000	0.752	0.764	0.739	0.780
Special	Flowers102 [50]	8,189	6,149	2,040	102	0.973	0.975	0.972	0.968
	CUB200 [63]	11,788	5,994	5,794	200	0.770	0.789	0.746	0.760
	Cars [31]	16,185	8,144	8,041	196	0.822	0.845	0.789	0.801
	Dogs [28]	20,580	12,000	8,580	120	0.850	0.864	0.841	0.835

Table 2. Median accuracies of *default networks* (init, final) and *block-trained networks* (init+, final+).

Models	Accuracy Type	Flowers102	CUB200	Cars	Dogs
ResNet-50	init	0.035	0.012	0.012	0.010
	init+	0.926	0.662	0.690	0.735
	final	0.962	0.707	0.800	0.754
	final+	0.970	0.746	0.821	0.791
ResNet-101	init	0.048	0.021	0.009	0.028
	init+	0.932	0.698	0.663	0.733
	final	0.968	0.741	0.832	0.785
	final+	0.977	0.767	0.844	0.814
Inception-V2	init	0.030	0.011	0.011	0.010
	init+	0.881	0.567	0.552	0.630
	final	0.960	0.705	0.785	0.732
	final+	0.966	0.725	0.806	0.771
Inception-V3	init	0.029	0.011	0.009	0.012
	init+	0.866	0.571	0.542	0.563
	final	0.959	0.711	0.796	0.728
	final+	0.965	0.735	0.811	0.755



(a) Flowers102 **(b)** Cars

Figure 7. Accuracies of pruned networks of ResNet-50 after training. The model size of full ResNet-50 is 25.6 million.

an accuracy sooner) as its validity is the prerequisite for the *composability-based CNN pruning* to work.

Table 2 reports the median of the initial and final accuracies of all 500 *block-trained networks* and their *default* counterparts for each of the models on every dataset. The mean is very close (less than 1%) to the median in all the settings. In this experiment, the tuning blocks are simply the CNN modules in each network. Overall, *block-trained networks* yield better final accuracies than *default networks* do with one-third less training time.

To show details, the two graphs in Figure 6 give accuracy curves attained during the trainings of one of the pruned networks in ResNet-50 and Inception-V3 respectively. Dataset CUB200 is used. The initial accuracies (*init*) are close to zero for the *default* version, while 53.4% and 40.5% for the *block-trained* version (*init+*). Moreover, the *default version* gets only 65.3% and 67.3% final accuracies (*final*) respectively, while the *block-trained version* achieves 72.5% and 70.5% after only two-thirds of the training time. Results on other pruned networks show a similar trend.

The results offer strong evidence for the *composability hypothesis*, showing that pre-training the tuning blocks of a CNN can indeed help the training of that CNN reach a given accuracy sooner. The benefits do not come for free; overhead is incurred by the pre-training of the tuning blocks. We next report the performance of Wootz as a whole.

7.3 Results of Wootz

We first evaluate the performance of composability-based network pruning and then report the extra benefits from the hierarchical tuning blocks identifier.

Basic Benefits To measure the basic benefits from the composability-based method, these experiments use every convolution module in these networks as a tuning block. The extra benefits from hierarchical tuning block identification are reported later.

Figure 7 shows the final accuracies of all the 500 ResNet-50 variants trained with or without leveraging composability on the Flower102 and CUB200 datasets. For reference, we also plot the accuracies of the well-trained full ResNet-50 on the two datasets. The block-trained network gives a clearly better final accuracy overall, which echoes the results reported in the previous subsection.

Table 3 reports the comparisons between the block-trained version and the default version, in both speeds and network sizes, at various levels of tolerable accuracy drop rates α (negative means higher accuracy than the large network gives). The results are collected when 1, 4, or 16 machines are used for concurrent training for both the baseline and our method (indicated by the "#nodes" column). The time of the block-trained version already takes the pre-training time of tuning blocks into account ("overhead" in Table 3 shows the percentage in overall time). For the objective of pruning, the exploration order Wootz adopts is to start from the smallest models and proceed to larger ones.

The results show that the composability-based method avoids up to 99.6% of trial configurations and reduces the evaluation time by up to 186X for pruning ResNet-50; up to

96.7% reduction and 30X speedups for Inception-V3. The reduction of trial configurations is because the method improves the accuracy of the pruned networks as Figure 7 shows. As a result, the exploration meets a desirable configuration sooner. For instance, in Flower102 ($\alpha = 0$), the third smallest network can already reach the target accuracy in the block-trained version, while the 297th network meets the target in the default version. This not only shortens the exploration time, but also yields more compact (up to 70% smaller) networks as the "model size" columns in Table 3 show. Another reason for the speedup is that the training of a block-trained network takes fewer iterations to reach its final accuracy level than the default version, as Figure 6 has illustrated. So even when configurations are not reduced (e.g., Flower102, $\alpha = -1$), the block-trained exploration finishes sooner.

Table 4 shows the speedups by composability-based pruning with different subspace sizes. The speedups are higher as the number of configurations to explore increases. It is because the time for pre-training tuning blocks weights less as the total time increases and the reduction of configurations becomes more significant for a larger set. Another observation is that, when the number of configurations is only four, there is still a significant speedup in most of cases. The block training time is the time spent on pre-training all the tuning block variants (48 for ResNet-50 and 27 for Inception-V3). The speedup could be higher if tuning block identifier is applied, as shown next.

Extra Benefits from Tuning Blocks Identification Hierarchical tuning block identifier balances the overhead of training tuning blocks and the time savings they bring to the fine-tuning of pruned networks. Table 5 reports the extra speedups brought when it is used.

For datasets Flowers102 and CUB200, we experiment with two types of collections of configurations with $N = 8$. The first type, "collection-1", is a randomly sampled collection as mentioned earlier, and the second type, "collection-2", is attained by setting one pruning rate for a sequence of convolution modules, similar to the prior work [39] to reduce module-wise meta-parameters. For each type, we repeat the experiments five times with a new collection created each time. Each tuning block identified from the first collection tends to contain only one convolution module due to the independence in choosing the pruning rate for each module. But the average number of tuning blocks is less than the total number of possible pruned convolution modules (41 versus 48 for ResNet-50 and 27 versus 33 for Inception-V3) because of the small collection size. The latter one has tuning blocks that contain a sequence of convolution modules as they are set to use one pruning rate.

The extra speedups from the algorithm are substantial for both, but more on the latter one for the opportunities that some larger popular tuning blocks have for benefiting the networks in that collection. Because some tuning blocks selected by the algorithm are a sequence of convolution modules that frequently appear in the collections, the total number of tuning blocks becomes smaller (e.g., 27 versus 23 on Inception-V3.)

8 Related Work

Recent years have seen many studies on speeding up the training and inference of CNN, both in software and hardware. For the large volume, it is hard to list all; some examples are work on software optimizations [16, 26, 44, 70] and work on special hardware designs [8, 11, 13, 45, 48, 52, 56]. These studies are orthogonal to this work. Although they can potentially apply to the training of pruned CNNs, they are not specifically designed for CNN pruning. They focus on speeding up the computations within one CNN network. In contrast, our work exploits *cross-network* computation reuse, exploiting the special properties of CNN pruning—many configurations to explore, common layers shared among them, and most importantly, the *composability* unveiled in this work. We next concentrate on prior work closely related to CNN pruning.

Deep neural networks are known to have many redundant parameters and thus could be pruned to more compact architectures. Network pruning can work at different granularity levels such as weights/connections [3, 17, 38], kernels [64] and filters/channels [39, 43, 47]. Filter-level pruning is a naturally structured way of pruning without introducing sparsity, avoiding creating the need for sparse libraries or specialized hardware. Given a well-trained network, different metrics to evaluate filters importance are proposed such as Taylor expansion [47], ℓ_1 norm of neuron weights [39], Average Percentage of Zeros [25], feature maps' reconstruction errors [20, 43], and scaling factors of batch normalization layers [42]. These techniques, along with general algorithm configuration techniques [6, 23, 58] and recent reinforcement learning-based methods [4, 19], show promise in reducing the configuration space worth exploring. This work distinctively aims at reducing the evaluation time of the remaining configurations by eliminating redundant training.

Another line of work in network pruning conducts pruning dynamically at runtime [12, 40, 46]. Their goals are however different from ours. Instead of finding the best small network, they try to generate networks that can adaptively activate only part of the network for inference on a given input. Because each part of the generated network may be needed for some inputs, the overall size of the generated network could be still large. They are not designed to minimize the network to meet the limited resource constraints on a system.

Sequitur [49] has been applied to various tasks, including program and data pattern analysis [9, 10, 33–35, 62]. We have not seen its use in CNN pruning.

Several studies attempt to train a student network to mimic the output of a teacher network [5, 7, 21]. Our method of pre-training tuning blocks is inspired by these work, but works at a different level: rather than for training an entire network,

Table 3. Speedups and configuration savings by composability-based pruning (when 1, 4, or 16 machines are used for both baseline and composability-based methods as "#nodes" column indicates). Notations are at the table bottom.

Dataset	α	#nodes	ResNet-50									Inception-V3								
			thr_acc	#configs		time (h)		model size		speedup (X)	overhead	thr_acc	#configs		time (h)		model size		speedup (X)	overhead
				base	comp	base	comp	base	comp				base	comp	base	comp	base	comp		
Flowers102	-1%	1	0.983	500	500	2858.7	1912.7	100%	100%	1.5	0.4%	0.978	500	500	3018.8	2023.5	100%	100%	1.5	0.5%
		4		500	500	718.1	481.0			1.5	0.5%		500	500	756.7	508.1			1.5	0.7%
		16		500	500	184.9	125.5			1.5	1.8%		500	500	194.8	133.6			1.5	2.7%
	0%	1	0.973	297	3	1639.4	16.9	45.4%	29.3%	97.0	40.4%	0.968	244	10	1428.6	47.3	43.2%	32.4%	30.2	23.3%
		4		300	4	412.6	5.2			79.3	43.5%		244	12	358.2	13.9			25.8	26.4%
		16		304	16	103.3	4.7			22.0	48.3%		256	16	94.8	6.5			14.6	56.4%
	1%	1	0.963	6	1	31.0	8.3	29.6%	27.6%	3.7	82.8%	0.958	27	1	152.6	13.9	33.9%	31.0%	11.0	79.0%
		4		8	4	10.4	3.2			3.3	70.6%		28	4	39.6	5.8			6.8	63.3%
		16		16	16	5.2	2.9			1.8	78.3%		32	16	11.2	5.6			2.2	71.0%
CUB200	4%	1	0.739	323	2	1807.3	12.7	46.6%	28.5%	142.3	53.7%	0.720	74	3	420.2	21.9	41.4%	33.7%	19.2	49.8%
		4		324	4	454.0	3.1			146.5	74.4%		76	4	106.4	6.7			15.9	54.5%
		16		336	16	118.7	3.1			38.3	74.4%		80	16	27.6	6.0			4.6	60.6%
	5%	1	0.731	297	1	1654.7	8.9	45.4%	27.6%	185.9	77.1%	0.710	44	1	247.8	14.1	38.5%	31.5%	17.6	77.5%
		4		300	4	418.8	2.8			149.6	81.4%		44	4	61.7	5.4			11.4	67.6%
		16		304	16	105.5	2.7			39.1	83.7%		48	16	16.4	5.2			3.2	70.6%
	6%	1	0.724	154	1	840.1	8.3	38.0%	27.6%	101.2	82.6%	0.700	29	1	162.5	12.8	35.9%	31.0%	12.7	85.1%
		4		156	4	214.2	2.6			82.4	86.7%		32	4	44.5	5.3			8.4	68.7%
		16		160	16	53.8	2.5			21.5	89.7%		32	16	10.8	5.1			2.1	71.9%
Cars	-1%	1	0.830	500	100	2864.9	362.4	100%	35.7%	7.9	1.9%	0.811	271	20	1586.8	85.6	40.1%	33.5%	18.5	12.8%
		4		500	100	720.4	90.9			7.9	2.5%		272	20	398.1	22.4			17.8	16.3%
		16		500	112	185.3	27.1			6.8	8.4%		272	32	99.4	11.1			9.0	32.8%
	0%	1	0.822	332	11	1848.6	44.4	46.9%	30.4%	41.6	15.4%	0.801	84	3	480.3	21.8	36.9%	31.3%	22.0	50.2%
		4		332	12	461.4	12.1			38.1	18.8%		84	4	120.5	7.2			16.7	50.6%
		16		336	16	115.9	5.2			22.3	44.0%		96	16	33.8	6.7			5.0	54.7%
	1%	1	0.814	189	2	1026.4	12.8	40.4%	28.5%	80.2	53.4%	0.791	33	1	186.4	14.2	34.4%	31.0%	13.1	77.0%
		4		192	4	259.7	4.9			53.0	46.7%		36	4	50.7	6.8			7.5	54.0%
		16		192	6	65.5	4.1			16.0	55.7%		48	16	16.4	6.2			2.6	59.1%
Dogs	6%	1	0.799	500	123	2848.1	441.1	60.0%	36.9%	6.5	1.6%	0.776	416	201	2470.7	786.0	100%	47.9%	3.1	1.4%
		4		500	124	709.8	111.2			6.4	2.0%		416	204	618.2	199.3			3.1	1.8%
		16		500	128	178.0	28.3			6.3	8.1%		416	208	153.2	52.7			2.9	6.9%
	7%	1	0.791	434	70	2445.4	251.8	51.9%	34.2%	9.7	2.7%	0.766	311	129	1822.2	503.2	56.0%	41.4%	3.6	2.2%
		4		436	72	606.2	63.9			9.5	3.6%		312	132	456.1	128.0			3.6	2.8%
		16		448	80	149.3	18.0			8.3	12.7%		320	144	116.2	36.4			3.2	10.0%
	8%	1	0.782	297	11	1632.8	42.3	45.4%	30.4%	38.6	16.2%	0.756	201	82	1164.1	322.9	47.9%	39.0%	3.6	3.4%
		4		300	12	411.7	10.1			40.8	22.7%		204	84	294.8	83.1			3.5	4.4%
		16		304	16	102.4	3.2			32.0	71.6%		208	96	75.0	26.1			2.9	13.9%

* *thr_acc*: accuracy corresponding to an accuracy drop rate α. *base*: baseline approach. *comp*: composability-based approach.
speedup: $Time_{base}/Time_{comp}$; overhead counted in $Time_{comp}$. *overhead*: block training time over the total time of *comp*.

Table 4. Speedups by composability-based pruning with different subspace sizes.

Dataset	alpha	subspace size	ResNet-50			Inception-V3		
			base time (h)	comp time (h)	speedup (X)	base time (h)	comp time (h)	speedup (X)
Flowers102	0%	4	22.7	13.4	1.7	20.3	16.8	1.2
		16	90.9	12.8	7.1	76.7	20.6	3.7
		64	364.8	21	17.4	224.7	25.4	8.8
		256	1460.7	13.5	108.2	809.4	40.7	19.9
CUB200	3%	4	22.8	11	2.1	23.6	26	0.9
		16	93.8	11.4	8.2	83.5	30	2.8
		64	369.6	15.5	23.8	292.5	29.2	10
		256	1472.9	20.7	71.2	1128.9	18.1	62.4

Table 5. Extra speedups brought by improved tuning block definitions.

Dataset	α	ResNet-50			Inception-V3		
		thr_acc	extra speedup (X)		thr_acc	extra speedup (X)	
			collection-1	collection-2		collection-1	collection-2
Flowers102	0%	0.973	1.05	0.98	0.968	1.12	1.14
	1%	0.963	1.19	1.21	0.958	1.08	1.15
	2%	0.953	1.06	1.14	0.949	1.15	1.23
CUB200	3%	0.747	1.04	1.08	0.737	1.00	1.03
	4%	0.739	1.04	1.20	0.729	1.08	1.09
	5%	0.731	1.11	1.15	0.722	1.03	1.04
geometric mean			1.08	1.12		1.08	1.11

we need to train pieces of a network. We are not aware of the prior use of such a scheme at this level.

9 Conclusions

This work proposes a novel composability-based approach to accelerating CNN pruning via computation reuse. It designs a hierarchical compression-based algorithm to efficiently identify tuning blocks for pre-training and effective reuse. It further develops Wootz, the first compiler-based software framework that automates the application of the composability-based approach to an arbitrary CNN model. Experiments show that network pruning enabled by Wootz shortens the state-of-the-art pruning process by up to 186X while producing significantly better pruned networks. As CNN pruning is an important method to adapt a large CNN model to a more specialized task or to fit a device with power or space constraints, its required long exploration time has been a major barrier for timely delivery of many AI products. The promising results of Wootz indicate its potential for significantly lowering the barrier, and hence reducing the time to market AI products.

References

[1] [n. d.]. Caffe Solver Prototxt. https://github.com/BVLC/caffe/wiki/Solver-Prototxt.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[3] Alireza Aghasi, Afshin Abdi, Nam Nguyen, and Justin Romberg. 2017. Net-Trim: Convex Pruning of Deep Neural Networks with Performance Guarantee. In *Advances in Neural Information Processing Systems*. 3180–3189.

[4] Anubhav Ashok, Nicholas Rhinehart, Fares Beainy, and Kris M Kitani. 2017. N2N Learning: Network to Network Compression via Policy Gradient Reinforcement Learning. *arXiv preprint arXiv:1709.06030* (2017).

[5] Jimmy Ba and Rich Caruana. 2014. Do deep nets really need to be deep?. In *Advances in neural information processing systems*. 2654–2662.

[6] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[7] Cristian Bucilua, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 535–541.

[8] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA^2: Exploiting Temporal Redundancy in Live Computer Vision. *arXiv preprint arXiv:1803.06312* (2018).

[9] Trishul M Chilimbi. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 191–202.

[10] Trishul M Chilimbi and Martin Hirzel. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 199–209.

[11] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. *arXiv preprint arXiv:1805.03718* (2018).

[12] Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. 2017. Spatially adaptive computation time for residual networks. *arXiv preprint* (2017).

[13] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weize, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI.

[14] Jianlong Fu, Heliang Zheng, and Tao Mei. 2017. Look closer to see better: Recurrent attention convolutional neural network for fine-grained image recognition. In *Conf. on Computer Vision and Pattern Recognition*.

[15] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. 2018. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[16] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[17] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[19] Yihui He and Song Han. 2018. ADC: Automated Deep Compression and Acceleration with Reinforcement Learning. *arXiv preprint arXiv:1802.03494* (2018).

[20] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, Vol. 2. 6.

[21] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[22] D. S. Hochbaum. 1995. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company.

[23] Holger H Hoos. 2011. Automated algorithm configuration and parameter tuning. In *Autonomous search*. Springer, 37–71.

[24] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[25] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250* (2016).

[26] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. 448–456.

[27] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[28] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Fei-Fei Li. 2011. Novel dataset for fine-grained image categorization: Stanford dogs. In *Proc. CVPR Workshop on Fine-Grained Visual Categorization (FGVC)*, Vol. 2. 1.

[29] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[30] Jonathan Krause, Benjamin Sapp, Andrew Howard, Howard Zhou, Alexander Toshev, Tom Duerig, James Philbin, and Li Fei-Fei. 2016. The unreasonable effectiveness of noisy data for fine-grained recognition. In *European Conference on Computer Vision*. Springer, 301–320.

[31] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 2013. 3d object representations for fine-grained categorization. In *Computer Vision Workshops (ICCVW), 2013 IEEE International Conference on*. IEEE, 554–561.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[33] James R Larus. 1999. Whole program paths. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 259–269.

[34] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. 2005. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, 135–146.

[35] James Law and Gregg Rothermel. 2003. Whole program path-based dynamic impact analysis. In *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 308–318.

[36] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. 1997. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks* 8, 1 (1997), 98–113.

[37] Yann LeCun and Yoshua Bengio. 1995. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks* 3361, 10 (1995), 1995.

[38] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.

[39] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).

[40] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime Neural Pruning. In *Advances in Neural Information Processing Systems*. 2178–2188.

[41] Jiaming Liu, Yali Wang, and Yu Qiao. 2017. Sparse Deep Transfer Learning for Convolutional Neural Network.. In *AAAI*. 2245–2251.

[42] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning efficient convolutional networks through network slimming. In *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2755–2763.

[43] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. *arXiv preprint arXiv:1707.06342* (2017).

[44] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. Parameter Hub: High Performance Parameter Servers for Efficient Distributed Deep Neural Network Training. *arXiv preprint arXiv:1801.09805* (2018).

[45] Tao Luo, Shaoli Liu, Ling Li, Yuqing Wang, Shijin Zhang, Tianshi Chen, Zhiwei Xu, Olivier Temam, and Yunji Chen. 2017. Dadiannao: A neural network supercomputer. *IEEE Trans. Comput.* 66, 1 (2017), 73–88.

[46] Mason McGill and Pietro Perona. 2017. Deciding how to decide: Dynamic routing in artificial neural networks. *arXiv preprint arXiv:1703.06217* (2017).

[47] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint arXiv:1611.06440* (2016).

[48] Andreas Moshovos, Jorge Albericio, Patrick Judd, Alberto Delmás Lascorz, Sayeh Sharify, Tayler Hetherington, Tor Aamodt, and Natalie Enright Jerger. 2018. Value-Based Deep-Learning Acceleration. *IEEE Micro* 38, 1 (2018), 41–55.

[49] Craig G. Nevill-Manning and Ian H. Witten. 1997. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res.(JAIR)* 7 (1997), 67–82.

[50] Maria-Elena Nilsback and Andrew Zisserman. 2008. Automated flower classification over a large number of classes. In *Computer Vision, Graphics & Image Processing, 2008. ICVGIP'08. Sixth Indian Conference on*. IEEE, 722–729.

[51] Simon O'Keeffe and Rudi Villing. 2018. Evaluating pruned object detection networks for real-time robot vision. In *Autonomous Robot Systems and Competitions (ICARSC), 2018 IEEE International Conference on*. IEEE, 91–96.

[52] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper* 2, 11 (2015).

[53] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*. 91–99.

[54] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[55] N. Silberman S. Guadarrama. 2016. TensorFlow-Slim: a lightweight library for defining, training and evaluating complex models in TensorFlow. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim.

[56] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[57] N. Silberman and S. Guadarrama. 2016. TensorFlow-Slim image classification model library. https://github.com/tensorflow/models/tree/master/research/slim.

[58] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. 2951–2959.

[59] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[60] Qing Tian, Tal Arbel, and James J Clark. 2017. Deep lda-pruned nets for efficient facial gender classification. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2017 IEEE Conference on*. IEEE, 512–521.

[61] Jonathan J Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. 2014. Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in neural information processing systems*. 1799–1807.

[62] Neil Walkinshaw, Sheeva Afshan, and Phil McMinn. 2010. Using compression algorithms to support the comprehension of program traces. In *Proceedings of the Eighth International Workshop on Dynamic Analysis*. ACM, 8–13.

[63] Peter Welinder, Steve Branson, Takeshi Mita, Catherine Wah, Florian Schroff, Serge Belongie, and Pietro Perona. 2010. Caltech-UCSD birds 200. (2010).

[64] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*. 2074–2082.

[65] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[66] Jianbo Ye, Xin Lu, Zhe Lin, and James Z Wang. 2018. Rethinking the Smaller-Norm-Less-Informative Assumption in Channel Pruning of Convolution Layers. *arXiv preprint arXiv:1802.00124* (2018).

[67] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 548–560.

[68] Tianyun Zhang, Kaiqi Zhang, Shaokai Ye, Jiayu Li, Jian Tang, Wujie Wen, Xue Lin, Makan Fardad, and Yanzhi Wang. 2018. Adam-admm: A unified, systematic framework of structured weight pruning for dnns. *arXiv preprint arXiv:1807.11091* (2018).

[69] Bo Zhao, Xiao Wu, Jiashi Feng, Qiang Peng, and Shuicheng Yan. 2017. Diversified visual attention networks for fine-grained object classification. *IEEE Transactions on Multimedia* 19, 6 (2017), 1245–1256.

[70] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. TBD: Benchmarking and Analyzing Deep Neural Network Training. *arXiv preprint arXiv:1803.06905* (2018).