ShrinkNets: Learning Network Size while Training

Anonymous Authors[1]

Abstract

As neural networks become widely deployed in different applications and on different hardware, it has become increasingly important to optimize inference time and model size along with model accuracy. Most current techniques optimize model size, model accuracy and inference time in different stages, resulting in suboptimal results and computational inefficiency. In this work, we propose a new technique called **ShrinkNets** that optimizes all three of these metrics simultaneously. Specifically we present a new method to simultaneously optimize network size and model performance by neuron-level pruning during training. Neuron-level pruning not only produces much smaller networks for a given accuracy threshold but also produces dense weight matrices that are amenable to efficient inference. By applying our technique to convolutional as well as fully connected models, we show that **ShrinkNets** can reduce network size by 35X with a 6X improvement in inference time. In addition, the most accurate networks found by **ShrinkNets** have accuracy that is similar to or better than those found by statically-sized networks.

1. Introduction

Neural networks are increasingly being deployed in a wide variety of applications and on diverse hardware architectures ranging from laptops to phones to embedded sensors. This wide variety of deployment settings means that inference time and model size are becoming as important as prediction quality when assessing model performance. However, these three dimensions of performance (model quality, inference time and model size) are largely optimized separately today, often with suboptimal results.

Of course, the problem of finding compressed or small networks is not new. Existing techniques to make a pre-trained neural network smaller can be categorized in two approaches: (1) quantization (**?**) and code compilation (Ma et al., 2016), techniques that can be applied blindly to any network, and (2) techniches which analyze the structure of the network and systematically prune connections or neurons (Han et al., 2015b; Cun et al., 1990) based on some loss function. While the first category is always useful, but only has limited impact on the network size, the second category can reduce the model size much more but has several drawbacks: First, those techniques often negatively impact model quality. Second, they can (surprisingly) negatively impact inference time as they transform dense matrix operations into sparse ones, which can be substantially slower to execute on modern hardware (Han et al., 2015b), especially GPUs which do not efficiently support sparse linear algebra. Third, these techniques generally start by optimizing a particular architecture for prediction performance, and then, as a post- processing step, applying compression to generate a smaller model that meets the resource constraints of the deployment setting. Because the network architecture is essentially fixed during this post-processing, model architectures that work better in small settings may be missed – this is especially true in large networks like many-layered CNNs, where it is infeasible to try explore even a small fraction of possible network configurations.

In contrast, in this paper we present a new method to simultaneously optimize network size and model performance. The key idea is to learn the right network size at the same time that we optimize for prediction performance for a specific task. Our approach, called *ShrinkNets*, starts with an *over-sized* network, and dynamically shrinks it by eliminating unimportant neurons—those that do not contribute to prediction performance—during training time. To do this, ShrinkNets must detect the unimportant neurons, and remove them from the network. The removal of entire neurons and not connections is crucial, because it leads to dense networks, which in turn leads to better inference performance. To support the detection and removal of unimportant neurons we introduce a new layer, called **Switch layer**. Introducing this new layer simply requires us to add a term to the loss function to optimize the weights of this layer during training. ShrinkNets has two main benefits. First, it explores the architecture of models that are both

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

small and perform well, rather than starting with a high-performance model and making it small. It does this using a single new hyperparameter that effectively models the target network size. This allows us to efficiently generate a family of smaller and accurate models without an exhaustive and expensive hyperparameter search over the number of neurons in each layer. Second, in contrast to existing neural network compression techniques (Aghasi et al., 2016; Han et al., 2015b), our approach results in models that are not only small, but where the layers are dense, which means that inference time is also improved, e.g., on GPUs.

In summary, our contributions are as follows:

1. We propose a novel technique based on dynamically switching on and off neurons, which allows us to optimize the network size while the network is trained.

2. We implement our technique so as to remove entire neurons, therefore leading not only to small and well-performing networks, but also dense, which means higher inference performance. Furthermore, our switching layers used during training can be savely removed before the model is used for inference, leading to zero additional overhead.

3. We show that our technique is a relaxation of group LASSO (Yuan & Lin, 2006) and prove that our problem admits many global minima.

4. We evaluate ShrinkNets with both fully-connected as well as convolutional neural networks. For `CIFAR10`, we achieve the same accuracy than a traditionally trained network while reducing the network size by a factor of $2.2x$. Further, when willing to sacrifice only 1% of performance, ShrinkNets finds networks which are 35% smaller. All in all, this leads to speedups in inference time of up to 6x.

2. Related Work

There are several lines of work related to optimizing network structure.

Hyperparameter optimization techniques: One way to optimize network architecture is to use hyperparameter optimization. Although many methods have been proposed, e.g., (Bergstra & Yoshua, 2012; Snoek et al., 2012), randomized search has been shown to work surprising well. The more complex methods for hyperparameter optimization include techniques, e.g., (Snoek et al., 2012) typically select hyperparameter combinations that come from uncertain areas of the hyperparameter space to search efficiency. As a generalization of this, methods based on bandit algorithms (e.g. (Li et al., 2016; Jamieson & Talwalkar, 2016)) have also become a popular way to tune hyperparameters by quickly discarding model configurations that perform badly. Although these methods could in theory be used to tune the number of neurons per layer of a network, in practice no related work proposes this, because treating each layer as a hyperparameter would lead an excessively large search space. In contrast, with ShrinkNets the size of the network can be tuned with a single parameter, as we explain in the next section..

Model Compression: Model compression techniques focus on reducing the model size *after* training, in contrast to ShrinkNets, which reduces it *while* training. Optimal brain damage (Cun et al., 1990) identifies connections in a network that are unimportant and then prunes these connections. DeepCompression (Han et al., 2015b) takes this one step further and in addition to pruning connections, it quantizes weights to make inference extremely efficient. A different vein of work such as (Romero et al., 2014; Hinton et al., 2015) proposes techniques for distilling a network into a simpler network or a different model. Because these techniques work after training, they are orthogonal and complementary to ShrinkNets. Further, some of these techniques, e.g., (Han et al., 2015a; Cun et al., 1990), produce sparse matrices that are not likely to improve inference times even though they reduce network size.

Dynamically Sizing Networks: The techniques closest to our proposed method are those based on group sparsity such as (Scardapane et al., 2017), and those like (Philipp & Carbonell, 2017) that dynamically grow and shrink the size of the network during training. (Scardapane et al., 2017) presents a method that also deactivates neurons using a loss function based on group-sparsity. However, the exact details of how their method works are not given, and their experimental results (on a small, fully connected network), are substantially worse than ours as shown in Section **??**. (Philipp & Carbonell, 2017) propose a method called Adaptive Radial-Angular Gradient Descent that adds neurons on the fly and removes neurons via an l_2 penalty. However, this approach requires a new optimizer and takes longer to converge compared to ShrinkNets.

3. The ShrinkNets Approach

In this section we describe the **ShrinkNets** approach, which allows us to learn the network size as part of the training process. The key idea is the addition of **SwitchLayers** after each layer in the network; these layers allow us to selectively deactivate neurons during training. We start by introducing the basic method, and then explain how to adapt the training procedure to support this new layer.

3.1. Overview

At a high-level, our approach consists of two interconnected stages. The first stage identifies neurons that do not improve the prediction accuracy of the network and deactivates them. The second stage then *removes* neurons from the network

(explicitly shrinking weight matrices and updating optimizer state) thus leading to smaller networks and also faster inference times.

Deactivating Neurons On-The-Fly: During the first stage, **ShrinkNets** applies an on/off switch to every neuron of an initially over-sized network. We model the on/off switches by multiplying each input (or output) of each layer by a parameter $\boldsymbol{\beta} \in \{0, 1\}$. A value of 0 will deactivate the neuron, while 1 will let the signal go through. These switches are part of a new layer, called the **SwitchLayer**; this layer applies to fully connected as well as convolutional layers.

Our objective is to minimize the number of "on" switches to reduce the model size as much as possible while preserving prediction accuracy. This can be achieved by jointly minimizing the training loss of the network and applying an L0 norm to the $\boldsymbol{\beta}$ parameters of the **SwitchLayer**. Since minimizing the L0 norm is an NP-Hard problem, we instead relax the constraint to an L1 norm by constraining $\boldsymbol{\beta}$ to be a real number instead of a binary value.

Neuron Removal: During this stage, the neurons that are deactivated by the switch layers are actually removed from the network, effectively shrinking the network size. This step improves inference times, as we demonstrate in our evaluation. We choose to remove neurons at training time because we have observed that this allows the remaining active neurons to adapt to the new network architecture and we can avoid a post-training step to prune deactivated neurons.

In the remainder of this section we describe in detail the switch layer as well as and the training process for **ShrinkNets**, and then describe the removal process in Section 4.1.

3.2. The Switch Layer

Let L be a layer in a neural network that takes an input tensor $\boldsymbol{x}$ and produces an output tensor y of shape $(c \times d_1 \times \cdots \times d_n)$ where c is the number of neurons in that layer. For instance, for fully connected layers, n=0 and the output is single dimensional vector of size c (ignoring batch size for now) while for a 2-D convolutional layer, n=2 and c is the number of output channels or feature maps.

We want to tune the size of L by applying a **SwitchLayer**, S, containing c switches. The **SwitchLayer** is parametrized by a vector $\boldsymbol{\beta} \in \mathbb{R}^c$ such that the result of applying S to $L(\boldsymbol{x})$ is a also a tensor size $(c \times d_1 \times \cdots \times d_n)$ such that:

$$S_\beta(L(\boldsymbol{x}))_{i,\ldots} = \boldsymbol{\beta}_i L(\boldsymbol{x})_{i,\ldots} \forall i \in [1 \ldots c] \tag{1}$$

Once passed through the switch layer, each output channel i produced by L is scaled by the corresponding $\boldsymbol{\beta}_i$. Note that when $\boldsymbol{\beta}_i = 0$, the i^{th} channel is multiplied by zero and will not contribute to any computation after the switch layer. If this happens, we say the switch layer has *deactivated* the neuron corresponding to channel i of layer L.

We place **SwitchLayer** after each layer whose size we wish to tune; these are typically fully connected and convolutional layers. We discuss next how to train **ShrinkNets**.

3.3. Training ShrinkNets

For training, we need to account for the effect of the **SwitchLayers** on the loss function. The effect of **SwitchLayers** can be expressed in terms of a sparsity constraint that pushes values in the $\boldsymbol{\beta}$ vector to 0. In this way, given a neural network parameterized by weights $\boldsymbol{\theta}$ and switch layer parameters $\boldsymbol{\theta}$, we optimize **ShrinkNets** loss as follows

$$L_{SN}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}, \boldsymbol{\beta}) = L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\beta}) + \lambda \left\|\beta\right\|_1 + \lambda_2 \left\|\boldsymbol{\theta}\right\|_p^p \tag{2}$$

This expression augments the regular training loss with a regularization term for the switch parameters and another on the network weights.

We next analyze why such loss function works to train **ShrinkNets** and its connection to group sparsity:

Relation to Group Sparsity (LASSO): **ShrinkNets** removes neurons, i.e., inputs and outputs of layers. For a fully connected layer defined as:

$$f_{\boldsymbol{A},\boldsymbol{b}}(\boldsymbol{x}) = a(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}) \tag{3}$$

where $\boldsymbol{A}$ represents the connections and $\boldsymbol{b}$ the bias, removing an input neuron j is equivalent to having $\left(\boldsymbol{A}^T\right)_j = \boldsymbol{0}$. Removing an output neuron i is the same as setting $\boldsymbol{A}_i = \boldsymbol{0}$ and $\boldsymbol{b}_i = 0$. Solving optimization problems while trying to set entire group of parameters to zero is the goal of group sparsity regularization (Scardapane et al., 2017). In any partitioning of the set of parameters $\boldsymbol{\theta}$ defining a model in p groups: $\boldsymbol{\theta} = \bigcup_{i=1}^{P} \boldsymbol{\theta}_i$, group sparsity penalty is defined as (where λ is the regularization parameter):

$$\Omega_\lambda^{gp} = \lambda \sum_{i=1}^{p} \sqrt{\mathbf{card}(\boldsymbol{\theta_i})} \left\|\boldsymbol{\theta_i}\right\|_2 \tag{4}$$

In fully-connected layers, the groups are either: columns of $\boldsymbol{A}$ if we want to remove inputs, or rows of $\boldsymbol{A}$ and the corresponding entry in $\boldsymbol{b}$ if we want to remove outputs. For simplicity, we focus our analysis in the simple one-layer case. In this case, filtering outputs does not make sense, so we only consider removing inputs. The group sparsity regularization then becomes (when $\sqrt{n}$ is folded into the λ)

$$\Omega_\lambda^{gp} = \lambda \sum_{j=1}^{p} \left\| \left(\boldsymbol{A^T}\right)_{\boldsymbol{j}} \right\|_2 \tag{5}$$

Group sparsity and **ShrinkNets** try to achieve the same goal. We discuss next how they are related to each other. First let's

recall the two problems. In the context of approximating $\boldsymbol{y}$ with a linear regression from features $\boldsymbol{x}$), the original **ShrinkNets** problem is

$$\min_{\boldsymbol{A},\boldsymbol{\beta}} \|\boldsymbol{y} - \boldsymbol{A}\text{diag}\left(\boldsymbol{\beta}\right)\boldsymbol{x}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1 \tag{6}$$

And the group sparsity problem is:

$$\min_{\boldsymbol{A}} \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2 + \Omega_\lambda^{gp} \tag{7}$$

We can prove that under the condition: $\forall j \in [\![1, p]\!], \left\|\left(\boldsymbol{A}^T\right)_j\right\|_2 = 1$ the two problems are equivalent (proposition A.1, see Apendix). However if we relax this constraint then **ShrinkNets** becomes non-convex and has no global minimum (propositions A.2 and A.3, also in Appendix). Fortunately, by adding an extra term to the **ShrinkNets** regularization term we can prove that:

$$\min_{\boldsymbol{A},\boldsymbol{\beta}} \|\boldsymbol{y} - \boldsymbol{A}\text{diag}\left(\boldsymbol{\beta}\right)\boldsymbol{x}\|_2^2 + \Omega_\lambda^s + \lambda_2 \|A\|_p^p \tag{8}$$

has global minimums for all $p > 0$ (proposition A.4). This is the reason we defined the regularized **ShrinkNets** penalty above as:

$$\Omega_{\lambda,\lambda_2,p}^{rs} = \lambda \|\boldsymbol{\beta}\|_1 + \lambda_2 \|\boldsymbol{\theta}\|_p^p \tag{9}$$

In practice we observed that $p = 2$ or $p = 1$ are good a choice; note that the latter will also introduce additional sparsity into the parameters because the L_1 is, thest best convex approximation of the L_0 norm.

4. ShrinkNets in Practice

In this section we discuss practical aspects of **ShrinkNets**, including neuron removal and several optimizations.

4.1. On-The-Fly Neuron Removal

Switch layers are initialized with weights sampled from $\mathcal{N}(0, 1)$; their values change as part of the training process so as to switch *on* or *off* neurons. When neurons are deactivated by switch layers, we need to remove them to actually shrink the network size. In the following we discuss three potential strategies to remove neurons.

Threshold strategy: Under this strategy neurons are removed based on a fixed threshold expressed in terms of its absolute value. This is the method used by deep compression (). We found this method is very sensitive to the initializaiton and not robust to noise in the gradients—which occurs commonly when training networks with stochastic gradient descent—because the threshold depends on the scale of each weight.

Sign change strategy: Under this strategy neurons are removed when the weight value changes its sign. As before this strategy is also sensitive to gradient noise; if we sample the gradient in a way that is not fully representative of the dataset we might experience one-time zero crossing which could wrongly remove a neuron.

Sign variance strategy: Instead of removing on the first zero crossing, we can instead measure the exponential moving variance of the sign $(-1, 1)$ of each parameter in the *switch layer*. When the value of the exponential moving variances goes over a predefined threshold, we consider the neuron deactivated, and therefore we remove it. This strategy introduces two extra parameters, one to control the behavior of the inertia of the variance, and the threshold. However, we found that this strategy performs the best in practice and is used throughout our evaluation section.

4.2. Additional Optimizations for ShrinkNets

Preparing for Inference: With ShrinkNets we obtain reduced-sized networks during training, which is the first steps towards faster inference. This networks are readily available for inference. However, because they include switch layers—and therefore more parameters—they introduce unnecessary overhead at inference time. To avoid this overhead, we reduce the network parameters by combining each switch layer with its respective network layer by multiplying the respective parameters before emitting the final trained network. As a result, the final network is a dense network without any switching layers.

Neural Garbage Collection: ShrinkNets decides on-the-fly which neurons to remove. Since ShrinkNets removes a large fraction of neurons, we must dynamically resize the network at runtime to not unnecessarily impact network training time. We implemented a neural garbage collection method as part of our library which takes care of updating the necessary network layers as well as updating optimizer state to reflect the neuron removal.

5. Evaluation

The goal of our evaluation is to explore (1) whether, by varying λ, **ShrinkNets** can efficiently explore (in terms of number of training runs) the spectrum of high-accuracy models from small to large, on both CNNs and fully connected networks. Our results show that, for each network size, we obtain models that perform as well or better than *Static Networks*, trained via traditional hyperparameter optimization; (2) whether, because these smaller networks are dense, they result in improved inference times on both CPUs and GPUs; and (3) whether the **ShrinkNets** approach results in network architectures that are substantially different than the best network architectures (in terms of relative number of neurons per layered) identified in the literature.

Implementation: We implemented **SwitchLayers** and the

associated training procedure as a library in pytorch (Paszke et al., 2017). The layer can be freely mixed with other popular layers such as convolutional layers, batchnorm layers, fully connected layers, and used with all the traditional optimizers. We use our implementation to evaluate **ShrinkNets** throughout the evaluation section.

5.1. Can ShrinkNets achieve good accuracy?

To answer this question we compare **ShrinkNets** with a traditional network. In both cases, we need to perform hyperparameter optimization to explore different network architectures. We perform random search, which is an effective technique for this purpose (Bergstra & Yoshua, 2012). We evaluate **ShrinkNets** on two well-known datasets. One for which it is not possible to explore the entire space of network architectures (`CIFAR10`) and one for which it is possible to do so (`COVERTYPE`).

Setup: We assume no prior knowledge on the optimal batch size, learning rate, λ or weight decay (λ_2). Instead, we trained a number of models, randomly and independently selecting the values of these parameters from a range of reasonable values. Training is done using gradient descent and the *Adam* optimizer (Kingma & Ba, 2014). Specifically, we start with the learning rate sampled randomly; for every 5 epochs of non-improvement in validation accuracy we divide the learning rate by 10. We stop training after 400 epochs or when the learning rate is under 10^{-7}, whichever comes first. For each of the models we trained, we pick the epoch with the best validation accuracy and report the corresponding testing accuracy. Because of the nature of our method, it can happen that for networks that are aggressively compressed, the best validation accuracy is obtained early in training, before the size has converged. To be sure that accuracy measured corresponds to the final shape and not the starting shape, we only consider the second half of the training when picking the best epoch. For each model, we also measure the total size, in terms of number of floating point parameters, excluding the **SwitchLayers** because as described in section 4.1, these are eliminated after training.

5.1.1. Large Network Setting: `CIFAR10`

`CIFAR10` is an image classification dataset containing 60000 color images ($3 \times 32 \times 32$), belonging to 10 different classes. We use it with the `VGG16` network (Srivastava et al., 2014), which consists of alternating convolutional layers and *MaxPool* layers interleaved by *BatchNorm* (Ioffe & Szegedy, 2015) and *ReLU* (Nair & Hinton, 2010) layers. The two last layers are fully connected layers separated by a *ReLU* activation function.

We applied **ShrinkNets** to the VGG16 network by adding **SwitchLayers** after each *BatchNorm* layer and each fully connected layer (except the last). Recall that **ShrinkNets** assume that the starting size of the network is an upper bound on the optimal size. Thus, we started with a network with 2x the recommended size for each layer as an upper bound (this is larger than what ImageNet uses).

We compare against classical (*Static*) networks. In such networks, the number of parameters that control the size is large: 13 parameters for the convolutional layers and 2 for the fully connected layers. **ShrinkNets** effectively fuse all these parameters in a single λ, but in conventional architectures where all of these parameters are free, it is infeasible to obtain a reasonable sample of a search space of this size. For this reason, we rely on the conventional heuristic that the original VGG architecture (and many CNNs) use, where the number of channels is doubled every after *MaxPool* laxer. For *Static Networks* we sample the size between 0.1 and 2 times the size original one, optimized for ImageNet. We report the same numbers as we did for **ShrinkNets** and we compare the two distributions.

The results are shown in the top figure of fig. 1, with blue dots indicating models produced by **ShrinkNets** and orange dots indicating static networks. For each model, we plot its accuracy and model size. The lines show the Pareto frontier of models in each of the two optimization settings. ShrinkNets explore the trade-off between model size and accuracy more effectively.

Note that the best performing ShrinkNets models has 92.07% accuracy which is identical to the accuracy of the static network, while the ShrinkNets model is 2.22 times smaller. In addition, if we give up just 1% error, ShrinkNets finds a model that is 35.5 times smaller than any static network that performs as good or better.

5.1.2. Small Network Setting: `COVERTYPE`

The `COVERTYPE` (Blackard, 1998) dataset contains 581012 descriptions of geographical area (elevation, inclination, etc...) and the goal is to predict the type of forest growing in each area. We picked this dataset for two reasons. First it is simple, such that we can reach good accuracy with only a few fully-connected layers. This is important because we want to show that *ShrinkNets* find sizes as good as *Static Networks*, even if we are sampling the entire space of possible network sizes. Second, Scardapane et al (Scardapane et al., 2017) perform their evaluation on this dataset, which allows us to compare the results obtained by our method with the method in (Scardapane et al., 2017).

We compare ShrinkNets against the same architecture used in (Scardapane et al., 2017), i.e., a three fully-connected layers network with no *Dropout* (Srivastava et al., 2014) and no *BatchNorm*. In this case, for the *Static Networks*, we independently sample the sizes of the three different layers to explore all possible architectures.

The results are shown in the top figure of fig. 2, with the two optimization methods plotted as before. Here, *Static* method finds models that perform well at a variety of sizes, because it is able to explore the entire parameter space. This is as expected; the fact that ShrinkNets performs as well as the Static indicates that ShrinkNets is doing an effective job of exploring the parameter space using just the single λ parameter.

Note that the best performing ShrinkNets models has 96.91% accuracy while the best static model is only 96.66% accurate, while the ShrinkNets model is 2.51 times smaller. In addition, if we give up just 0.5% error, ShrinkNets finds a model that is 38.6 times smaller than any static network with equivalent accuracy.

Summary. We demonstrated that it is possible to achieve networks with good accuracy when using ShrinkNets both when the network space cannot be explored entirely (`CIFAR10`) as well as when it can, e.g., `COVERTYPE`. The most important result is not that ShrinkNets finds networks of good accuracy, but that those networks are much smaller than those found by a static method. The impact of the network size on inference time is the subject of our next evaluation goal.

5.2. Can ShrinkNets speed up inference?

The previous experiment showed that ShrinkNets finds networks of similar or better accuracy than static networks that are much smaller. We now explore if the reduction in size translates into an improvement of the inference time.

As noted in the introduction, for some applications, compact models that offer fast inference times are as important as absolute accuracy. In this section, we study the relationship between accuracy, network size and inference time. To do this, we select the smallest model that achieves a given accuracy for the both ShrinkNets and Static approach. For each model, we measure the time to run inference with the model. We then compute the ratio of the network size and inference time between ShrinkNets and Static at each accuracy level, and plot them on the bottom of Figure 1 and 2. We limit our plots to the models with $80 - 100\%$ accuracy range because those are the ones that we consider to be practically useful.

The middle plot in each figure shows the ratio of model size between ShrinkNets and Static (values >1 mean ShrinkNets are smaller) at different accuracy levels. These figures show that is that size improvements are are particularly significant for `CIFAR10`. In the range of accuracies we are interested in, improvements in size go from 4x to 40x. On the `COVERTYPE` dataset, the compression ratio is always above 1 but it rarely exceeds 3x, except for very high accuracies where *ShrinkNets* finds excellent, small solutions. The fact that the `COVERTYPE` networks are not dramatically smaller is expected: as the distribution at the top of Figure 2 shows, the static method is able to explore most of the parameter search space, so finds a range of models that perform well at different sizes.

For speedup, we experimented with both CPUs and GPUs, and with different batch sizes, where batch size indicates the number of inputs simultaneously fed to the model for inference. For each data set/GPU/CPU combination, we show results with batch size 1, as well as with a batch size large enough to fully utilize the hardware on each dataset and hardware configuration. For example, for `CIFAR10` on CPU, a batch size of 64 fully utilizes the CPU, whereas a GPU can execute many more models in parallel, so we use a larger batch size of 1024. For `COVERTYPE`, because the model is so much smaller, larger batches are needed to fully utilize the hardware. Note that when using a batch size of 1 on GPU, we do not expect to (and do not) observe any improvement because inference times are very small (typically about 10 μs), such that setup time dominates overall runtime.

The bottom four graphs in each figure show the results. Again, the `CIFAR10` results show the benefit of the ShrinkNets approach most dramatically. On CPU, speedups range up to 6x depending on the batch size, with many models exceeding 3x speedup. In general, speedups are less than compression ratios, due to overheads in problem setup, invocation, and result generation in Python/PyTorch. On GPU, the speedups are less substantial because the CUDA benchmarking utility that we use for evaluation can choose better algorithms for larger matrices which masks some of our benefit, although they are still often 1.5x–2x faster for large batch sizes. The speedup results on `COVERTYPE` are similar to those for network size: because the networks are not much smaller, they are not much faster either.

A key takeaway of these speedup results is that, unlike local sparsity compression methods, our methods' improvement on size translates directly to higher throughput at inference time (Han et al., 2015a).

5.3. Architectures obtained after convergence

ShrinkNets effectively explores the frontier of model size and accuracy. For a given target accuracy, the size needed is significantly smaller than when we use the "channel doubling" heuristic commonly used to size convolutional neural networks. This suggests that this conventional heuristic may not in fact be optimal, especially when looking for smaller models. Empirically we observed this to often be the case. For example, during our experimentations on the `MNIST` (LeCun et al., 2001) and `FashionMNIST` (Xiao et al., 2017) datasets (not reported here due to space constraints), we observed that even though these datasets have the same

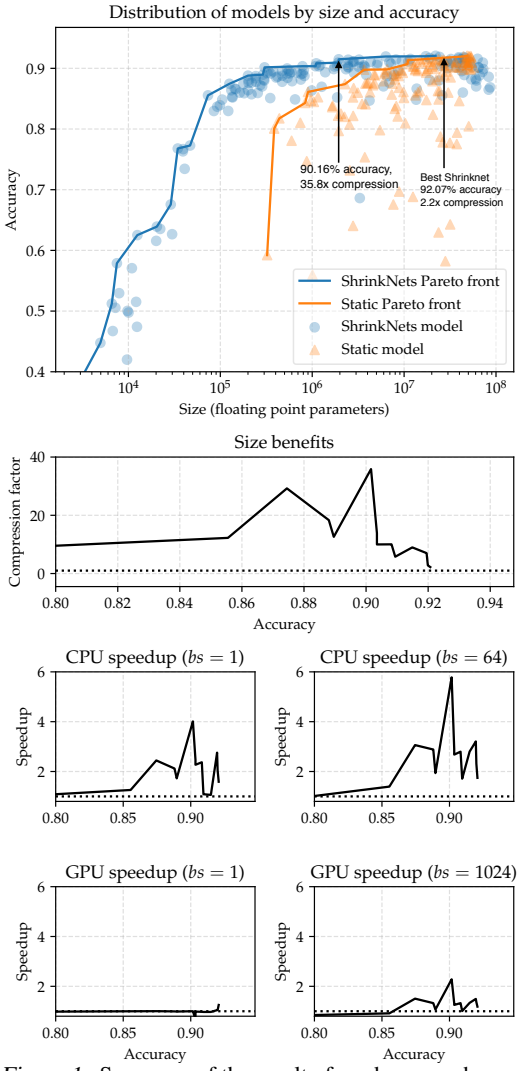

Figure 1. Summary of the result of random search over the hyper-parameters the `CIFAR10` dataset

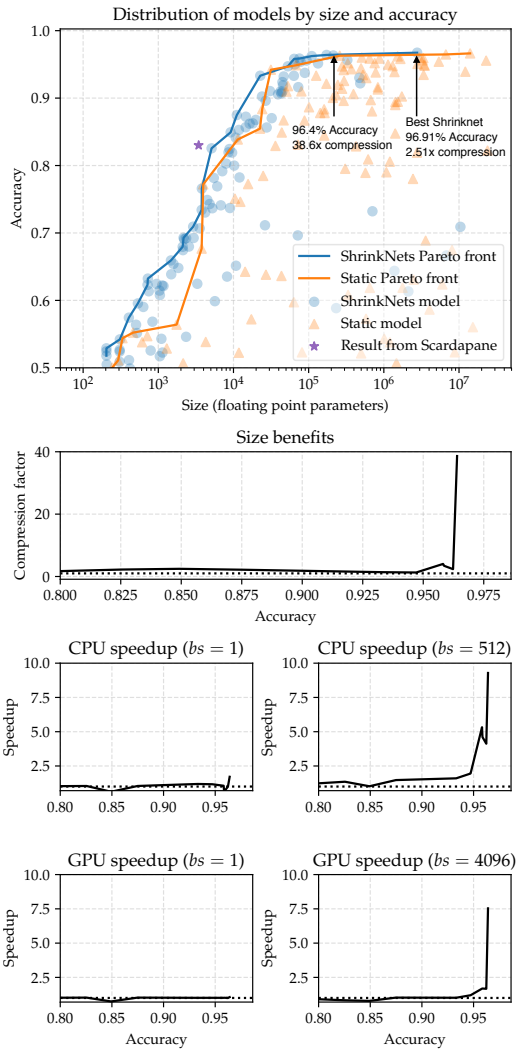

Figure 2. Summary of the result of random search over the hyper-parameters the `COVERTYPE` dataset

number of classes, input features, and output distributions, for a fixed λ *ShrinkNets* converged to considerably bigger networks in the case of `FashionMNIST`. This evidence shows that optimal architecture not only depends on the output distribution or shape of the data but actually reflects the dataset. This makes sense, as `MNIST` is a much easier problem than `FashionMNIST`.

To illustrate this point on a larger dataset, we show two examples of architectures learned by *ShrinkNets* in Figure 3. The top plot shows the model with the best test accuracy, with identical performance to the best static network; the bottom shows a network that slightly under-performs the best in terms of accuracy but is significantly smaller that the best equivalent *Static Network*. In the plot, the dashed line shows the number of neurons in each layer of the original VGG net, and the shaded regions show the size of the ShrinkNet as it converges (with the darkest region representing the fully converged network). Observe that the final network that is trained looks quite different in the two cases, with the optimal performing network appearing similar to the original VGG net, whereas the shrunken network allocates many fewer neurons to the middle layers, and then additional neurons to the final fewer layers.

6. Conclusion

We presented ShrinkNets, an approach to learn deep network sizes while training. ShrinkNets consists of the Switch Layer, which deactivates neurons, as well as of a method to remove them, which shrinks network sizes leading to faster inference times. We demonstrated these claims on on two well-known datasets, for which we achieved networks of the same accuracy than traditional neural networks, but up to 35X smaller; all in all leading to inference speedups of up to 6x.

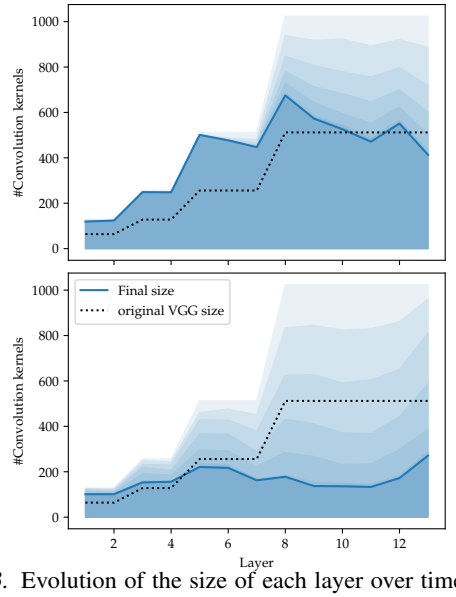

Figure 3. Evolution of the size of each layer over time (lighter: beginning, darker: end). On top a very large network performing 92.07%, at the bottom a simpler model with 90.5% accuracy.

A. Appendix

Here we prove the propositions given in Section **??**.

Proposition A.1. $\forall (n,p) \in \mathbb{N}_+^2, \boldsymbol{y} \in \mathbb{R}^n, \boldsymbol{x} \in \mathbb{R}^p \lambda \in \mathbb{R}$*:*

$$\min_{\mathbf{A}} \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2 + \lambda \sum_{j=1}^{p} \left\| \left(A^T\right)_j \right\|_2$$

$$= \min_{\mathbf{A}',\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{A}' diag\left(\boldsymbol{\beta}\right) \mathbf{x}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1$$

$$s.t. \forall j, 1 \leq j \leq p, \left\| \left(A'^T\right)_j \right\|_2^2 = 1$$

Proof. We split the proof into two parts. First, we demonstrate that there is at least one global minimum. Then, we how to construct 2^k distinct solutions from a single global minimum. In order to prove this second statement, we first show that for any solution $\boldsymbol{A}$ to the first problem, there exists a solution in the second with the exact same value, and vice-versa.

Part 1: Assume we have a potential solution $\boldsymbol{A}$ for the first problem. We define $\boldsymbol{\beta}$ such that $\boldsymbol{\beta}_j = \left\| \left(\boldsymbol{A}^T\right)_j \right\|_2^2$, and $\boldsymbol{A}' = \boldsymbol{A}\left(\text{diag}\left(\boldsymbol{\beta}\right)\right)^{-1}$. It is easy to see that the constraint on $\boldsymbol{A}'$ is satisfied by construction. Now:

$$\|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2 + \lambda \sum_{j=1}^{p} \left\| \left(A^T\right)_j \right\|_2$$

$$= \|\mathbf{y} - \mathbf{A}' \text{diag}\left(\boldsymbol{\beta}\right) \mathbf{x}\|_2^2 + \lambda \sum_{j=1}^{p} \left\| \left(A'^T\right)_j \beta_j \right\|_2$$

$$= \|\mathbf{y} - \mathbf{A}' \text{diag}\left(\boldsymbol{\beta}\right) \mathbf{x}\|_2^2 + \lambda \sum_{j=1}^{p} |\beta_j| \cdot 1$$

$$= \|\mathbf{y} - \mathbf{A}' \text{diag}\left(\boldsymbol{\beta}\right) \mathbf{x}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1$$

Part 2: Assuming we take an $\boldsymbol{A}'$ that satisfies the constraint and a $\boldsymbol{\beta}$, we can define $\boldsymbol{A} = \boldsymbol{A'}\text{diag}\left(\boldsymbol{\beta}\right)$. We can apply the same operations in reverse order and obtain an instance of the first problem with the same value.

Conclusion There is no way these two problems have different minima, because we are able to construct a solution to a problem from the solution of the other while preserving the value of the objective. □

Proposition A.2. $\|\mathbf{y} - \mathbf{A} diag\left(\boldsymbol{\beta}\right) \mathbf{x}\|_2^2$ *is not convex in* $\boldsymbol{A}$ *and* $\boldsymbol{\beta}$*.*

Proof. To prove this we will take the simplest instance of the problem: where everything is a scalar. We have $f(a,\beta) = (y - a\beta x)^2$. For simplicty we's take $y = 0$ and $x > 0$. If we consider two candidates $s_1 = (0,2)$ and $s_2 = (2,0)$, we have $f(s_1) = f(s_2) = 0$. However $f(\frac{2}{2}, \frac{2}{2}) = x > \frac{1}{2}f(0,2) + \frac{1}{2}f(2,0)$, which break the convexity property. Since we showed that a particular case of the problem is non-convex then necessarily the general case cannot be convex. □

Proposition A.3. $\min_{\mathbf{A},\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{A} diag\left(\boldsymbol{\beta}\right) \mathbf{x}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1$ *has no solution if* $\lambda > 0$*.*

Proof. Let's assume this problem has a minimum $\boldsymbol{A}^*, \boldsymbol{\beta}^*$. Let's consider $2\boldsymbol{A}^*, \frac{1}{2}\boldsymbol{\beta}^*$. Trivially the first component of the sum is identical for the two solutions, however $\lambda \left\|\frac{1}{2}\boldsymbol{\beta}\right\| < \lambda \|\boldsymbol{\beta}\|$. Therefore $\boldsymbol{A}^*, \boldsymbol{\beta}^*$ cannot be the minimum. We conclude that this problem has no solution. □

Proposition A.4. *For this proposition we will not restrict ourselves to single layer but the composition of an an arbitrary large* (n) *layers as defined individually as* $f_{\boldsymbol{A}_i,\boldsymbol{\beta}_i,\boldsymbol{b}_i}(x) = a(\boldsymbol{A_i} diag\left(\boldsymbol{\beta_i}\right) \boldsymbol{x} + \boldsymbol{b_i})$*. The entire network follows as:* $N(\boldsymbol{x}) = \left(\bigcirc_{i=1}^{n} f_{\boldsymbol{A_i},\boldsymbol{\beta_i},\boldsymbol{b_i}}\right)(\boldsymbol{x})$*. For* $\lambda > 0$*,* $\lambda_2 > 0$ *and* $p > 0$ *we have:*

$$\min \|\boldsymbol{y} - N(\boldsymbol{x})\|_2^2 + \Omega_{\lambda,\lambda_2,p}^{rs}$$

has at least 2^k *global minimum where* $k = \sum_{i=1}^{n} \#\boldsymbol{\beta_i}$

Proof. We split this proof into two parts. First we show that there is at least one global minimum, then we will show how to construct $2^n - 1$ other distinct solutions with the same objective.

Part 1: The two components of the expression are always positive so we know that this problem is bounded by below by 0. $\Omega_{\lambda,\lambda_2,p}^{rs}$ is trivially coercive. Since we have a sum of terms, all bounded by below by 0 and one of them is coercive, so the entire function admits at least one global minimum.

Part 2: Let's consider one global minimum. For each component k of $\boldsymbol{\beta_i}$ for some i. Negating it and negating the

k^{th} column of $\boldsymbol{A_i}$ does not change the the first part of the objective because the two factors cancel each other. The two norms do not change either because by definition the norm is independent of the sign. As a result these two sets of parameters have the same value and by extension also a global minimum. It is easy to see that going from this global minimum, we can decide to negate or not each element in each $\boldsymbol{\beta_i}$. We have a binary choice for each parameter, there are $k = \sum_{i=1}^{n} \#\boldsymbol{\beta_i}$ parameters, so we have at least 2^k global minima.

□

References

Aghasi, Alireza, Abdi, Afshin, Nguyen, Nam, and Romberg, Justin. Net-Trim: Convex Pruning of Deep Neural Networks with Performance Guarantee. 2016.

Bergstra, James and Yoshua, Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305, 2012. ISSN 1532-4435. doi: 10.1162/153244303322533223.

Blackard, Jock A. *Comparison of Neural Networks and Discriminant Analysis in Predicting Forest Cover Types*. PhD thesis, Fort Collins, CO, USA, 1998. AAI9921979.

Cun, Yann Le, Denker, John S, and Solla, Sara a. Optimal Brain Damage. *Advances in Neural Information Processing Systems*, 2(1):598–605, 1990. ISSN 1098-6596. doi: 10.1.1.32.7223.

Han, Song, Mao, Huizi, and Dally, William J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. 2015a. doi: abs/1510.00149/1510.00149.

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

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

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

Jamieson, Kevin and Talwalkar, Ameet. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pp. 240–248, 2016.

Kingma, Diederik P. and Ba, Jimmy. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

LeCun, Y, Bottou, L, Bengio, Yoshua, and Haffner, P. Gradient-Based Learning Applied to Document Recognition. In *Intelligent Signal Processing*, pp. 306–351, 2001. ISBN 0018-9219. doi: 10.1109/5.726791.

Li, Lisha, Jamieson, Kevin, DeSalvo, Giulia, Rostamizadeh, Afshin, and Talwalkar, Ameet. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

Ma, Yufei, Suda, N., Cao, Yu, s. Seo, J., and Vrudhula, S. Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Aug 2016. doi: 10.1109/FPL.2016.7577356.

Nair, Vinod and Hinton, Geoffrey E. Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning*, (3):807–814, 2010. ISSN 1935-8237. doi: 10.1.1.165.6419.

Paszke, Adam, Gross, Sam, Chintala, Soumith, Chanan, Gregory, Yang, Edward, DeVito, Zachary, Lin, Zeming, Desmaison, Alban, Antiga, Luca, and Lerer, Adam. Automatic differentiation in pytorch. 2017.

Philipp, George and Carbonell, Jaime G. Nonparametric Neural Network. In *Proc. International Conference on Learning Representations*, number 2016, pp. 1–27, 2017.

Romero, Adriana, Ballas, Nicolas, Kahou, Samira Ebrahimi, Chassang, Antoine, Gatta, Carlo, and Bengio, Yoshua. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

Scardapane, Simone, Comminiello, Danilo, Hussain, Amir, and Uncini, Aurelio. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, 2017. ISSN 18728286. doi: 10.1016/j.neucom.2017.02.029.

Snoek, Jasper, Larochelle, Hugo, and Adams, Ryan P. Practical bayesian optimization of machine learning algorithms. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25*, pp. 2951–2959. Curran Associates, Inc., 2012.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. ISSN 15337928. doi: 10.1214/12-AOS1000.

Xiao, Han, Rasul, Kashif, and Vollgraf, Roland. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. 2017.

Yuan, Ming and Lin, Yi. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, 68(1):49–67, 2006. ISSN 13697412. doi: 10.1111/j.1467-9868.2005.00532.x.