Learning Network Size while Training with ShrinkNets

Anonymous Authors[1]

Abstract

Let's write the abstract at the end

1. Introduction

As neural networks become increasingly widely deployed in a variety of applications – ranging from improving camera quality on mobile phones (**?**) to language translation (**?**) to text auto-completion (**?**) – and on diverse hardware architectures – from laptops to phones to embedded sensors – inference performance and model size are becoming as important in learning as metrics measures of prediction quality. However, these three aspect of model performance – quality, performance and 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 typically aim make a pre-trained neural network smaller (**??**) by taking one of two approaches: either by applying quantization (**?**) or code compilation (**?**) techniques that can be applied blindly to any network, or by analyzing the structure of the network and systematically pruning connections (Han et al., 2015; Cun et al., 1990) or neurons (**?**) based on some loss function. Although these techniques can substantially reduce model size, they have several drawbacks. First, they 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 (**?**), 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 both 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. Our approach, called *ShrinkNets*, starts with an *oversized* network, and dynamically shrinks it by eliminating neurons during training time. Our approach has two main benefits. First, we explore the architecture of models that are both small and perform well, rather than starting with a high- performance model and making it small. 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., 2015), 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.

XXX people actually don't treat network size as a hyperparameter; we do!

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 as the network is trained.
2. **Sam:** xxx postprocessing techniques to strip out switch layers
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. **Sam:** Some claim about model size vs performance
5. We demonstrate the efficacy of our technique on both convolutional and fully-connected neural nets, showing that ShrinkNets finds networks within +/-X% of best hand-crafted accuracy in XX% of training time compared to existing hyperparameter optimization methods. **Sam:** Mention the specific model/dataset where this is true.
6. We demonstrate that ShrinkNets can achieve this accuracy with only YY% of neurons. **Sam:** This claim should be something about how much smaller a network with say 99% accuracy is one some real dataset
7. **Sam:** Something about dense networks and then bene-

[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.

fit over optimal brain damage

8. **Sam:** Some claim about inference time
9. **Sam:** Some claim about compatibility with existing compression techniques?

2. Related Work

- Other techniques for choosing network size (e.g., hyperparmeter optimization) – manual effort vs optimal techniques – people actually don't treat network size as a hyperparameter; we do! Also brute force techniques (**?**)
- distillation techniques – train a big net, then train a smaller net from it
- post-training compression techniques – brain damage ,
- group sparsity e.g., (Scardapane et al., 2017) and non-parametric neural networks –
- training dynamics paper: first overfitting and then randomization?, **Gl:** Here is the ref, if you can introduce it in the flow (Shwartz-Ziv & Tishby, 2017)

Given the importance of network structure, many techniques have been proposed to find the best network structure for a given learning task. These techniques broadly fall into four categories: hyperparameter optimization strategies, post-training model compression for inference as well as model simplification, techniques to resize models during training, and automated architecture search methods.

The most popular techniques for hyperparameter optimization include simple methods such as random search (Bergstra & Yoshua, 2012) which have been shown to work suprisingly well compared to more complex methods such as those based on Bayesian optimization (Snoek et al., 2012). Techniques such as (Snoek et al., 2012) model generalization performance via Gaussian Processes (Rasmussen & Williams, 2006) and select hyperparameter combinations that come from uncertain areas of the hyperparameter space. Recently, methods based on bandit algorithms (e.g. (Li et al., 2016; Jamieson & Talwalkar, 2016)) have also become a popular way to tune hyperparameters. As noted before, all of the above techniques require many tens to hundreds of models to be trained, making this process computationally inefficient and slow.

In contrast with hyperparameter tuning methods, some methods such as DeepCompression (Han et al., 2015) seek to compress the network to make inference more efficient. This is accomplished by pruning connections and quantizing weights. On similar lines, multiple techniques such as (Romero et al., 2014; Hinton et al., 2015) have been proposed for distilling a network into a simpler network or a different model. Unlike our technique which works during training, these techiques are used after training and it would be interesting to apply them to ShrinkNets as well. (Abadi et al., 2016) share the common goal of removing entire blocks of parameter to maintain dense matrices, however their method only applies to convolutional layers.

The techniques closed to our work are those based on group sparsity such as (Scardapane et al., 2017), and those like (Philipp & Carbonell, 2017) who grow and shrink dynamically the size of the network during training.

Finally, there has also been recent work in automatically learning model architecture through the use of genetic algorithms and reinforcement learning techniques (Zoph & Le, 2016; Zoph et al., 2017). These techniques are focused more on learning higher-level architectures (e.g. building blocks for neural network architectures) as opposed to learning network size.

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 *switch layers* added after each layer in the network; these layers allow us to disable certain 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, out 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, shrinking its size and yielding faster inference times. Broadly, these stages work as follows:

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 $\boldsymbol{\theta} \in \{0, 1\}$. 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 **switch layer**; 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 objective of the network and a factor of the L0 norm of the vector containing all the switches. Because finding an optimal binary assignment is an NP-Hard problem, we relax the problem by allowing $\boldsymbol{\theta}$ to be a real number instead of a binary value and constrain it using the L1 instead

of L0 norm.

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. Existing techniques focus on neuron removal after training, and require an extra fine-tuning process to compensate for the removal.**Sam:** cite something?

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 switch layer, S, containing c switches. The switch layer is parametrized by a vector $\theta \in \mathbb{R}^c$ 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:

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

Once passed through the switch layer, each output channel i produced by L is scaled by the corresponding θ_i. Note that when $\theta_i = 0$, 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 switch layers 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 switch layers on the loss function. The effect of switch layers can be expressed in terms of a sparsity constraint that pushes values in the θ vector to 0. In this way, given a neural network parameterized by weights $\boldsymbol{W}$ and switch layer parameters $\boldsymbol{\theta}$, we optimize ShrinkNet loss as follows

$$L_{SN}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{W}, \boldsymbol{\theta}) = L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{W}) + \lambda \|\theta\|_1 + \lambda_2 \|\boldsymbol{W}\|_p \quad (2)$$

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:

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

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 (**?**). In any partitioning of the set of parameters θ defining a model in p groups: $\boldsymbol{\theta} = \bigcup_{i=1}^{P} \boldsymbol{\theta}_i$, group sparsity is defined as: **Sam:** you used to say "we define 'it' as:" – I am assume 'it' is group sparsity

$$\Omega_\lambda^{gp} = \lambda \sum_{i=1}^{p} \sqrt{\#\boldsymbol{\theta_i}} \, \|\boldsymbol{\theta_i}\|_2 \quad (4)$$

Sam: define notation – what is λ; what is $\#\theta$, what is Ω? Where does the square root come from? 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:

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

Because $\forall i, \#\boldsymbol{\theta}_i = n$, **ra:** is # the general way of expressing cardinality? why not $|x|$?to make the notation simpler, we embedded $\sqrt{n}$ inside λ.

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. The original ShrinkNet problem is **Sam:** do you mean "can be defined formally as" – we never gave this definition of ShrinkNets. Also, you need to explain the formula / notation; what is y, what is β, what is this equation representing (the optimization function?):

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

And the Group Sparsity problem is:

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

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 ShrinkNet regularization term we can prove that:

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

has global minimums for all $p > 0$ (proposition A.4). This is the reason we defined the *regularized ShrinkNet penalty* above as **Sam:** where?:**ra:** the notation is slightly different, better to make it consistent

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

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. **Sam:** why?

4. ShrinkNets in Practice

In this section we discuss practical aspects of ShrinkNets, including the neuron removal process and several additional 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. Since after removing a neuron it will no longer be able to contribute to prediction accuracy, we must devise strategies that remove neurons while remaining robust to changing values. We observe three strategies:

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 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. This strategy works well in practice but it is also sensitive to gradient noise. **ra:** explain what's the consequence of that 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 kill a neuron.

Sign variance strategy: Instead of killing 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, but we have found it is the best performing in practice.

The last two strategies perform better than the threshold strategy. Both effectively allow us to shrink the network by varying λ, and result in networks that are both small and perform well.

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. **ra:** can't we just simply prevent the following problem at network design time?In the unlikely case that a switch layer is sandwiched between two non-linearly scalable layers, we leave the switch layer as is.

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:

- 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.
- Whether, because these smaller networks are dense, they result in improved inference times on both CPUs and GPUs.
- 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.
- **Sam:** something else?

Implementation: We implemented **switch Layers** 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 (). 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 (**Gl:** should we make them explicit ? **Sam:** yes). 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. **Gl:** Should I also give the details about the removal strategy we used and the γ and threshold I used ? because this is clearly getting boring **Sam:** probably not necessary 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 *Switch Layers* 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 *Switch Layers* 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. For this reason, we simply started with a network with double the recommended size for each layer as an upper bound (this is larger than what ImageNet, for example uses). Thus, for the classification layers we use 5000 neurons as a starting limit (ImageNet uses 4096 **Gl:** This is on the top of my head, need to be double checked).

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) **Gl:** try to find the paper that introduces this heuristic use, where **Sam:** describe what the heuristic is – doubling of neurons up to apoint? For *Static Networks* we sample the size between 0.1 and 2 times the size **Sam:** size of what? optimized for ImageNet. **Sam:** Why are we using ImageNet and not CIFAR10 as the comparison point? We report the same numbers as we did for *ShrinkNets* and we compare the two distributions. **Sam:** I worry that this method of selecting network sizes for Static will of course constrain it to larger sizes and worse performance. What if we just randomly sample networks? Would it look terrible?

The results are shown in the top figure of fig. 1, with blue dots indicating ShrinkNet models 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. **Sam:** Would be good to show Lambda values on the figure somehow. **Sam:** I thought we were going to summarize with distributional plots instead of Pareto frontiers?

Note that the best performing ShrinkNets models has **ra:** xx accuracy while the best static model has **ra:** xx accuracy, while the ShrinkNets model is **ra:** xx times smaller. In addition, if we give up just 1% error, ShrinkNets finds a model that is **ra:** xx times smaller.

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 **ra:** xx accuracy while the best static model has **ra:** xx accuracy, while the ShrinkNets model is **ra:** xx times smaller. In addition, if we give up just 1% error, ShrinkNets finds a model that is **ra:** xx times smaller.

5.1.3. 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`) and 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. **Sam:** Can we cite something showing that local sparsity does not improve performance even w/ small nets?, at least for larger batches when hardware can be fully utilized.

ra: checkpoint

5.3. Architectures obtained after convergence

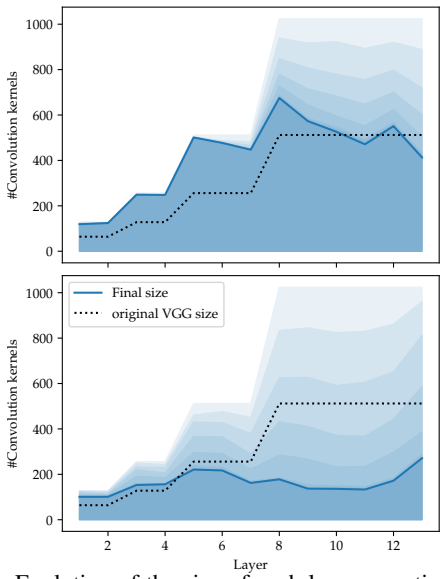

Figure 3. Evolution of the size of each layer over time (lighter: beginning, darker: end). On top a very large network performing 92.07%, at the bottom a simpler model with 90.5% accuracy.

ra: the following is interesting, but how is it related to the above? 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 **Sam:** name it 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 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 **Sam:** identical performance to the best static network?;

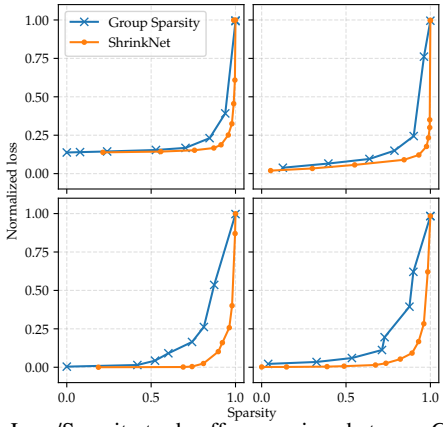

Figure 4. Loss/Sparsity trade off comparison between Group Sparsity and Shrinknet on linear and logistic regression. From top to bottom and left to right we show the results for `scm1d`, `oes97`, `gina_prior2` and `gsadd`.

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* **Sam:** how much smaller?. 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

A. Appendix

A.1. Proofs

Unless stated explicitely, all the propositions consider the Multi-Target linear regression problem.

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 will split this proof in two different sections. the first one will demonstrate that there is at least one global minimum. and the second will show how to construct 2^k distinct solutions from a single global minimum. In order

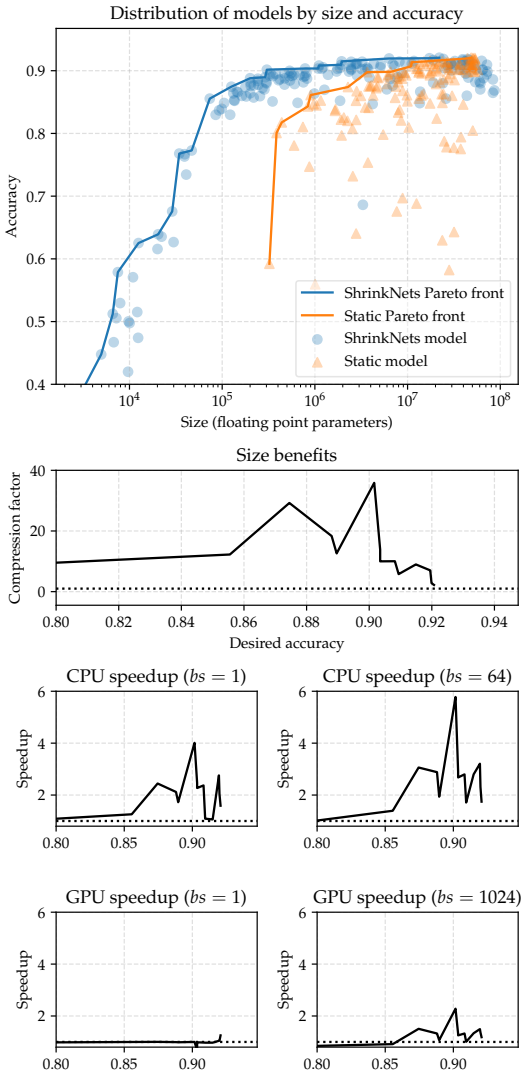

Figure 1. Summary of the result of random search over the hyper-parameters the `CIFAR10` dataset

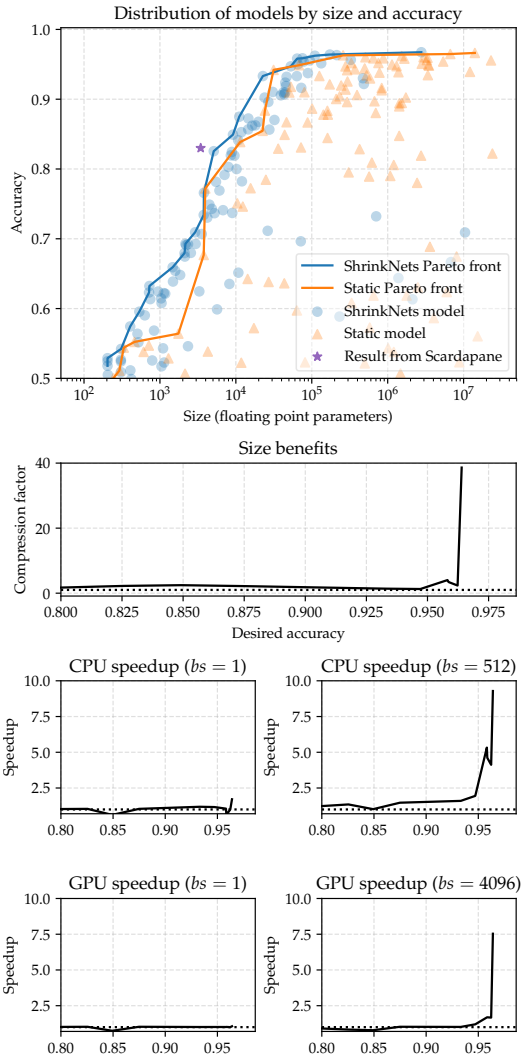

Figure 2. Summary of the result of random search over the hyper-parameters the `COVERTYPE` dataset **Sam:** label the x axes on the bottom plots; change "Desired accuracy" "Accuracy"

to prove this statement we will show that for any solution $\boldsymbol{A}$ in the first problem, there exists a solution in the second with the exact same value, and vice-versa.

Part 1: We assume we have a potential solution $\boldsymbol{A}$ for the first problem and 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:

$$
\begin{aligned}
&\|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2 + \lambda \sum_{j=1}^{p} \left\| \left(A^T\right)_j \right\|_2 \\
&= \|\mathbf{y} - \mathbf{A}'\text{diag}\left(\boldsymbol{\beta}\right)\mathbf{x}\|_2^2 + \lambda \sum_{j=1}^{p} \left\| \left(A'^T\right)_j \beta_j \right\|_2 \\
&= \|\mathbf{y} - \mathbf{A}'\text{diag}\left(\boldsymbol{\beta}\right)\mathbf{x}\|_2^2 + \lambda \sum_{j=1}^{p} |\beta_j| \cdot 1 \\
&= \|\mathbf{y} - \mathbf{A}'\text{diag}\left(\boldsymbol{\beta}\right)\mathbf{x}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1
\end{aligned}
$$

Which concludes the first part.

Part 2: Assuming we take an $\boldsymbol{A}'$ that satisfy 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 let'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: $N(\boldsymbol{x}) = (\bigcirc_{i=1}^{n} f_{\boldsymbol{A_i},\boldsymbol{\beta_i},\boldsymbol{b_i}})(\boldsymbol{x})$. For $\lambda > 0$, $\lambda_2 > 0$ and $p > 0$ we have:*

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

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

Proof. We will split this proof in two parts. First we will 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. $\Omega^{rs}_{\lambda,\lambda_2,p}$ is trivially coercive. Since we have a sum of terms, all bounded by below by 0 and one of them is coercive, then the entire function admit 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 $\boldsymbol{A_i}$ do not change the the first part of the objetive because the two factors cancel each other. The two norms do not change either because by definition the norm is independant of the sign. As a result these two sets of parameter have the same value and by extension also global minimuma. 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.

□

A.2. Multi-Target Linear and Multi-Class Logistic regressions experiments

As we showed, Group sparsity share similarities with our method, and we claim that ShrinkNets are a relaxation of group sparsity. In this experiment we want to compare the two aproaches. We decided to focus on multi-target linear regression because in the single target case, groups in the Group Sparsity problem would have a size of one ($\boldsymbol{A}$ would be a vector in this case).

The evaluation will be done on two datasets `scm1d` and `oes97` (Spyromitros-Xioufis et al., 2016) for linear regressions and we will use `gina_prior2` (Guyon et al., 2007) and the *Gas Sensor Array Drift Dataset* (Vergara et al., 2012) (that we shorten in `gsadd`) for logistic regressions.

For each dataset we fit with different regularization parameters and measure the error and sparsity obtained after convergence. In this context we define sparsity as the ratio of columns that have all their weight under 10^{-3} in absolute value. Regularization parameters were choosed in order to obtain the widest sparsity spectrum. Loss is normalized depending on the problem to be in the $[0, 1]$ range. We summarized the results in fig. 4. From our experiments it is clear that ShrinkNets can fit the data closer than Group Sparsity for the same amount of sparsity. The fact that we are able to reach very low loss demonstrate that even if our objective function is non convex, in practice it works as good or better as convex alternatives.

References

Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, Kudlur, Manjunath, Levenberg, Josh, Monga, Rajat, Moore, Sherry, Murray, Derek G, Steiner, Benoit, Tucker, Paul, Vasudevan, Vijay, Warden, Pete, Wicke, Martin, Yu, Yuan, Zheng, Xiaoqiang, Brain, Google, Osdi, Implementation, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, Kudlur, Manjunath, Levenberg, Josh, Monga, Rajat, Moore, Sherry, Murray, Derek G, Steiner, Benoit, Tucker, Paul, Vasudevan, Vijay, Warden, Pete, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. TensorFlow : A System for Large-Scale Machine Learning. In *OSDI*, 2016. ISBN 9781931971331.

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.

Guyon, I., Saffari, A., Dror, G., and Cawley, G. Agnostic learning vs. prior knowledge challenge. In *2007 International Joint Conference on Neural Networks*, pp. 829–834, Aug 2007. doi: 10.1109/IJCNN.2007.4371065.

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*, 2015.

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.

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.

Rasmussen, CE and Williams, CKI. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, USA, 1 2006. ISBN 0-262-18253-X.

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.

Shwartz-Ziv, Ravid and Tishby, Naftali. Opening the Black Box of Deep Neural Networks via Information. *arXiv*, pp. 1–19, mar 2017.

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.

Spyromitros-Xioufis, Eleftherios, Tsoumakas, Grigorios, Groves, William, and Vlahavas, Ioannis. Multi-target regression via input space expansion: treating targets as inputs. *Machine Learning*, 104(1):55–98, 2016. ISSN 1573-0565. doi: 10.1007/s10994-016-5546-z.

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.

Vergara, Alexander, Vembu, Shankar, Ayhan, Tuba, Ryan, Margaret A., Homer, Margie L., and Huerta, Ramn. Chemical gas sensor drift compensation using classifier ensembles. *Sensors and Actuators B: Chemical*, 166-167:320 – 329, 2012. ISSN 0925-4005. doi: https://doi.org/10.1016/j.snb.2012.01.074.

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.

Zoph, Barret and Le, Quoc V. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

Zoph, Barret, Vasudevan, Vijay, Shlens, Jonathon, and Le, Quoc V. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.