

# Structured Multi-Hashing for Model Compression

Elad Eban*	Yair Movshovitz-Attias*	Hao Wu	Mark Sandler
Google Research.	Google Research.	Google Research.	Google Research.
elade@google.com	yairmov@google.com	haou@google.com	sandler@google.com
Andrew Poon	Yerlan Idelbayev†	Miguel Á. Carreira-Perpiñán	
Google Research.	University of California, Merced.	University of California, Merced.	
ayp@google.com	yidelpayev@ucmerced.edu	mcarreira-perpinan@ucmerced.edu	

## Abstract

Despite the success of deep neural networks (DNNs), state-of-the-art models are too large to deploy on low-resource devices or common server configurations in which multiple models are held in memory. Model compression methods address this limitation by reducing the memory footprint, latency, or energy consumption of a model with minimal impact on accuracy. We focus on the task of reducing the number of learnable variables in the model.

In this work we combine ideas from weight hashing and dimensionality reductions resulting in a simple and powerful structured multi-hashing method based on matrix products that allows direct control of model size of any deep network and is trained end-to-end.

We demonstrate the strength of our approach by compressing models from the ResNet, EfficientNet, and MobileNet architecture families. Our method allows us to drastically decrease the number of variables while maintaining high accuracy. For instance, by applying our approach to EfficientNet-B4 (16M parameters) we reduce it to the size of B0 (5M parameters), while gaining over 3% in accuracy over B0 baseline.

On the commonly used benchmark CIFAR10 we reduce the ResNet32 model by 75% with no loss in quality, and are able to do a 10x compression while still achieving above 90% accuracy.

\*The author contribute equally to this paper.

Elad and Yair contributed equally to the paper. They jointly proposed the idea of structured-multi-hashing. Yair was the main contributor to the manuscript. Elad wrote most of the code and ran EfficientNet experiments. Hao contributed to coding and experiments. Yerlan ran CIFAR and ResNet experiments and simplified some aspects of the structured hashing. Miguel advised Yerlan on issues about optimization and deep net compression. Mark and Andrew helped with MobileNet, and ResNet experiments.

†Work performed while at Google Research.

## 1. Introduction

The main factor driving the success of machine learning in recent years is the ability to build and train increasingly larger Deep Neural Networks (DNNs). This has been enabled by a combination of algorithmic advances such as ReLU activations [15, 34], Batch Normalization [23], and residual connections [17]; large training datasets [8]; and faster, specialized, hardware [25].

Overwhelmingly, when given enough data, larger models show improvements in accuracy. However, this march upwards comes with a cost in terms of latency, energy, and memory consumption. For example, the popular Resnet-101 model [17] has 44 Million parameters and requires 150MB of storage; AmoebaNet-A [39] requires 469M parameters and 1800MB. The size of DNNs limits their deployment in devices with low resources such as mobile phones and wearables. On server side, multi-tenancy – the practice of serving multiple models from the same hardware accelerator – is also affected by the model size. Furthermore, during inference, layers deeper in the network can be heavily affected by the cost of loading the weights.

From a scientific perspective, these models have many more parameters than the number of data points in the datasets they are trained on. This seems counter-intuitive as it seems to contradict learning-theory (e.g. VC dimension properties [45]), but has been widely recognized as critical property of DNNs [2, 10, 1]. One wonders: Do the parameters of a network live in a lower dimensional space? Can we restrict the model class in a way that models in it can be represented efficiently (e.g. have low Kolmogorov Complexity) without sacrificing accuracy? Can we find an intrinsic connection between the number of parameters of a model to its performance [29]?

Note that low dimensionality assumptions are core in many CNN components. For example, convolutions are low dimensional linear maps and separable convolutions (i.e. depthwise followed by  $1 \times 1$  convolutions) are based on de-

composition restrictions. However, making strong assumptions about individual elements in the model can be overly restrictive.

There is considerable interest in the machine learning community in making models cheaper: Reducing their size, either in number of parameters or as bytes on disk; lowering their latency; or reducing their memory and energy consumption during inference. Here we refer to these methods as *model compression*.

We can partition the model compression field into several types of techniques: architectural modifications, such as width multiplier, a move to separable convolutions, or filter number optimization [28, 14]; Neuron pruning, either during or after training; disk size compression [36]; weight quantization [11]; and hashing [47, 42]. These approaches are in many ways complementary and have been used together [16]. In practice, hashing methods induce identity constraints between model weight that are mapped to the same variables. In addition, they lack memory locality which makes them slow and increases their RAM footprint. This has limited their adoption.

We present a new hashing approach for reducing the number of trainable variables in a model. We consider all weights in the DNN as if they are tiled into a single, large, matrix and represent it as a sum of products of multiple hashes, computed as matrix product. This defines a multi-hash from model weights into sets of trainable variables in which full collisions are exponentially rare, and are replaced by higher order correlations between weights. Using this representation, we then train the reduced model end to end.

We call this Structured Multi-Hashing (*SMH*). *SMH* has a specific locality pattern which reduces cache misses and increases the efficiency of the compressed model. This representation is unique: it is not a linear subspace nor does it assume that any specific operation in the network is low rank. Furthermore, by re-parameterizing hashing as a matrix product, the implementation becomes both simple and fast. It has little overhead in training or inference and results in much faster models compared to hashed models. We demonstrate the efficacy of *SMH* by applying it to state-of-the-art image classification models and drastically reducing their number of variables.

## 2. Related Work

Numerous efforts have been made on model compression, here we give a brief overview of different approaches.

**Hashing** The seminal work of Weinberger *et al.* [47, 42] showed how useful hashing is in the context of linear classifiers. The work builds upon the kernel-trick and is designed to allow more efficient training and inference when the number of features and labels is huge.

Chen *et al.* [5] extended this idea to the context of deep

networks introducing HashNets. Each layer in the network is independently hashed into a smaller set of variables.

Reagen *et al.* [38] use Bloomier filters [4] in order to index the weights. This work takes a post-training/pruning approach, the filters are not trained from scratch, and fine-tuning is needed to achieve good performance. Similarly, Locality Sensitive Hashing has been used in [43] to maintain smaller weight pool.

**Pruning** is the process of removing unnecessary weights [28, 32] or entire neurons/filters [31, 50, 14] of the trained neural networks with the goal of maintaining as close as possible performance to the unpruned version. This can be achieved by penalizing the model with sparsifying norms [14, 31, 3] or by ranking the weights/neurons [28], in one or multiple iterations.

**Weight Quantization** Model quantization works by adjusting parameter values to lower precision [11, 33, 48, 24] or even binary weights [7, 51]. This has the desired effect of drastically reducing the size, and can be efficiently combined with pruning [16, 6] to get even higher compression ratios.

**Decomposition** We can largely identify pure low-rank methods [9, 10, 49] that apply matrix decomposition to fully connected and suitably reshaped convolutional weight matrices, and its generalization — tensor decomposition of layers [35, 27, 12, 46]. The ranks of the decompositions can be learned too [30, 49, 22].

**Architecture Design** A separate line of research is building compact models and training them from scratch [21, 20, 44], rather than compressing overparametrized ones. This is intrinsically manual process, and reinforcement learning methods are used to automate this task [18].

## 3. Method

Our method is based on a hashing scheme applied to the original variables of the model. It is inspired by Chen *et al.* [5] but rather than having a many-to-one mapping between weights and trainable variables, we use a many-to-many mapping. This exponentially reduces the probability of a full collision in the hash. Further, our approach maintains memory locality and so can be implemented efficiently, without latency overhead during inference.

Note that normally there is a 1-to-1 correspondence between the set of weight tensors of a model, and its set of trainable variables. In fact the names *weights* and *variables* are often used interchangeably. However, when considering model compression, specifically a hashing based approach, one needs to be clear about this distinction. Here we refer to *weights* as the tensor values in convolution kernels, fully connected layers, biases and so on. We call *variables* the set of trainable elements into which weights get hashed.

We denote by  $w^l$  the weight tensor of layer  $l$ , and the elements of  $w^l$  as  $[i_1, \dots, i_k]$  for a rank  $k$  tensor (e.g.  $k = 4$  for 2D convolutions,  $k = 2$  for fully connected layers).

For simplicity of notations, we will use  $w^l[i]$  to cover both kernel-weights and biases. We define  $W = \{w^l\}_{l=1}^L$  the set of all weights of a network, and  $|W|$  is the total number of weights of the model. These weights are essentially the set of all tensors which we set out to compress: convolution kernels, fully connected weight matrices, and biases, with the exception of the scale and bias parameters of batch-normalization[23], as these parameters can be absorbed into the next operation during inference.

Here we should note that while it is common to hash or quantize weights *after* training, we consider the use case of hashing weights into variables while the model is training. This allows the values of the hashing variables to be learned using back propagation.

**Simple-hashing** A simple hashing scheme is based on an underlying set of  $\mathcal{S}$  variables, which we call a variable pool and denote by  $\mathcal{V}$ .

The hashing is defined by a function  $h$ :

$$h : [l, i_l] \rightarrow \mathcal{S}$$

where  $l$  takes values from 1 to  $L$  - the number of layers in the network, and  $i_l$  are indices into  $w^l$ .

The hash  $h$  induces a mapping between model weights and variable:  $w^l[i] = \mathcal{V}[h(l, i)]$ .

The simple-hashing is similar to the one proposed in [5], with the difference that it does not have random signs, and it operates on all the variables in the network (compared to their per layer approach). This simple-hashing scheme is shown in Figure 1a and can be thought of training a network with a variable sharing pattern induced by the collisions of  $h$ . The number of collisions is equal to the number of weights reduced by the hashing scheme and when compression is not trivial, there is a large number of constraints. This could pose a problem if, for instance, a certain layer needs its weights to be of large magnitude, while another layer requires small values. When optimizing the model, an unfortunate compromise would arise.

To overcome these hard collisions, we propose a multi-hashing scheme shown in Figure 1b that induces a different set of constraints on the network in which the collisions induce softer, smoother, non-linear constraints.

**Multi-hashing** An  $M$ -hashing<sup>1</sup> is defined with a set of  $M$  hash functions  $\{h_m\}_1^M$  and  $M$  variable pools  $\{\mathcal{V}_m\}_1^M$ , and a reducer function  $\phi : \mathbb{R}^M \rightarrow \mathbb{R}$ .

<sup>1</sup>To clarify, we use the term multi-hash differently than commonly used in computer science theory - We employ our multi-hashes in *parallel* to each weight index  $i_l$  to produce a set of variables, and then combine them using the reducer function and produce a value to be placed in  $w^l[i]$ .

These define the following mapping between model weights and variables:

$$w^l[i] = \phi(\mathcal{V}_1[h_1(l, i)], \dots, \mathcal{V}_M[h_M(l, i)])$$

The choice of the reduction function is an important component in the multi-hashing scheme. The sum function is an example of a simple reducing function:

$$\phi(x_1, \dots, x_M) = \sum_{m=1}^M x_m \quad (1)$$

other functions such as the product can also be considered.

### 3.1. Structured Multi-Hashing

Instead, we use multi-hashing to partition the variables into groups which share some dependency structure. Common hash functions would create random partitioning, but this loses a property which could be important for our use case: memory locality. Neighboring weights in the network can be mapped to arbitrary variables in the pool and so a layer  $l$  potentially needs to access all the variable pools to compute its output.

Specifically, when we consider an implementation where we do not unpack the hashing offline, but compute the values of the weight tensors on the fly, then the cost of fetching all the variable pools could be significant. We propose the notion of structured hashing which will increase the memory locality.

We define  $2M$  sum-product reducer as:

$$\phi_{\sum \prod}(x_1, \dots, x_{2M}) = \sum_{m=1}^M x_{2m-1} x_{2m} \quad (2)$$

Combined with a carefully chosen hashing scheme, this reducer will maintain memory locality and is efficient during inference.

First we re-parameterize the way we refer to the weights. Let  $N = |W|$  be the number of total weights in the network. Rather than  $w^l[i]$  we think of all the weights of a model as if coming from a single square matrix of dimension  $n$ , where  $n = \lceil \sqrt{N} \rceil$ . The mapping between the weights and the elements of the matrix is trivially achieved by tiling the weights in the order of their creation:

$$w^l[i] = \hat{M}[r_i, c_i] \quad (3)$$

where  $r_i$  and  $c_i$  are row and column indices determined by  $i$ . This mapping is illustrated in Figure 2.

The core idea of our structured hashing approach is to encode the locality of weights using matrix operations. A multi-hashing scheme determines how we *represent*  $M_W$ . We define a  $2M$  Matrix-Product Multi-Hashing:

$$\hat{M} = [\mathcal{V}_1; \mathcal{V}_2; \dots; \mathcal{V}_M]^t [\mathcal{V}_{M+1}; \mathcal{V}_{M+2}; \dots; \mathcal{V}_{2M}] \quad (4)$$

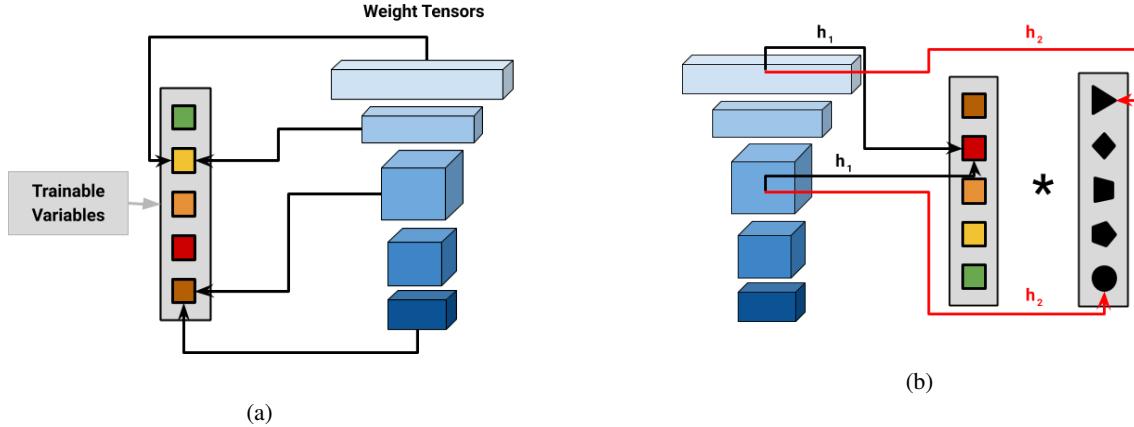


Figure 1: (a) Model compression using hashing. Each value in the set of weight tensors in the model (blue cubes) is mapped into a single variable in the set of trainable variables. When strong compression is desired, many weight values are mapped into the same variable, creating equality constraints in the network. (b) Model compression using multi-hashing. Each value in the set of weight tensors in the model (blue cubes) is mapped into a number of trainable variables, each mapping using a different hash function. The probability of two weights being mapped to the same *set* diminishes exponentially.

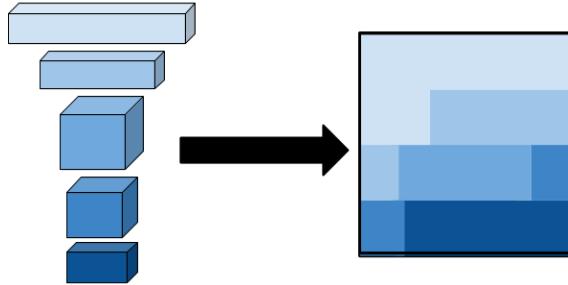


Figure 2: We conceptually represent the full set of weights of a deep model as a single, square, matrix. In this matrix  $w^l[i]$  is mapped to some coordinates  $M_W[r, k]$ . We then apply structured multi-hash to represent this matrix using a sum-product of smaller matrices.

Where  $\mathcal{V}_i$  are column vectors of size  $n$ . Note that the re-parameterization defined in Eq. (3) combined with the decomposition into hash functions in Eq. (4) implements a  $2M$  multi-hash with a sum-product reducer in a way that is memory efficient.

Note that our approach computes a low rank approximation of the weight matrix of the entire model. However, this does not assume that any specific layer is low rank nor does it enforce it.

### 3.2. Selecting the Number of Hashes

The size of the Matrix-Product Multi-Hash, i.e. the number of trainable variables it creates is  $2Mn$ . This is determined by the size of the hash vectors  $\mathcal{V}$  and the number of hashes. These define the size of the matrices in Eq. (4) —  $n \times M$  and  $M \times n$ . We can hash the model into any target size  $T$  (up to rounding errors in the order of  $\sqrt{N}$ ) by

setting  $M = \lceil T/2n \rceil$ .

### 3.3. Scaling and Initialization

Correctly initializing the weights of a deep network is often important for it to train well. As such this is an active area of research and there is a plethora of initialization methods available to practitioners, and each model architecture is paired with an initialization scheme that fits it. For our multi-hash compression method, to match the performance of the uncompressed model, we would want the weights to be initialized using a matching distribution. Note however that a single weight value in our method is the sum-of-products of  $2M$  variables. For a target distribution  $\mathbb{D}$  one can define  $2M$  distributions  $\mathbb{d}_m$  such that the distribution of their sum-product is equals  $\mathbb{D}$ . Specifically, for the commonly used Gaussian distribution the sum part is trivial as the Gaussian family is closed to additions. However, although well defined, a distribution where its product is a Gaussian is hard to sample from [37]. Instead, we focus on matching two properties of  $\mathbb{D}$ : its range, and scale. Note that the common practice in deep models is to use the same family of distributions in all layers, but with an appropriately selected per-layer-scale. We follow that practice here.

**Range** Initialization schemes can be categorized into two types: unbounded distributions (e.g. Normal), and bounded ones (e.g. uniform, truncated normal). When initializing variables in the pool we match the range property.

**Scale** The challenge with the scale is that different layers can be initialized to different scales. This happens for instance with Glorot [13] initialization where the standard deviation is a function of the fan-in and fan-out of the layer. In this setting, it is impossible to initialize the hash variables

such that all layers simultaneously have the desired scale. We solve this by first initializing variables so  $std(w^l) = 1$ , and then we re-scale each layer to match the target scale  $s^l$ . For the sum-product reducer we standardize the resulting weights by setting the standard deviation  $\sigma$  or the underlying variables according to:

$$\begin{aligned} std(w^l) &= std \left( \sum_{m=1}^M x_{2m-1} x_{2m} \right) \\ &= \sqrt{\sum_{m=1}^M var(x_{2m-1}) var(x_{2m})} \\ &= \sqrt{M\sigma^4} \end{aligned}$$

Setting  $\sigma = M^{-\frac{1}{4}}$  creates weights with unit standard deviation. Then, multiplying by  $s^l$  allows us to effectively control the scale of each layer.

### 3.4. Per-Layer Learnable Scale

While our multi-hash technique removes equality constraints, there are still dependencies between weights as they share some of the variables used in their sum-products. Consider two layers  $l$  and  $l'$ , it could be hard for the network to learn different scales for  $w^l$  and  $w^{l'}$  due to sharing of the underlying variables. The per-layer scaling mentioned above addresses this problem at initialization, but layers initialized with the same scale are bound to keep similar scales while training. We allow the per-layer scale to be a *learnable* variable, which provides the network with another degree of freedom to address this issue. Our experiments in Sec 4.6 show that this small set of extra variables (one per layer) are always helpful, and result in 0.5% – 1% improvement in accuracy.

## 4. Results

In this section, we evaluate our method on three model families: ResNets, EfficientNets, and MobileNets. We show that *SMH* compression can drastically reduce model size with minimal loss in quality. In fact, when compressing large models, we often outperform comparably sized models from the same family.

### 4.1. ResNet Models

ResNet architectures [17] are versatile and so are used in many applications. They are also popular as benchmark models. There are two main procedures used to make ResNet models cheaper: Changing the number of *layers* (e.g. ResNet101, ResNet50, ResNet18) and changing the number of *filters*, usually done with uniform scaling and is commonly known as width multiplier.

Figure 3 shows our structured multi-hashing compression on ResNet50 and ResNet101. Each point on the curve

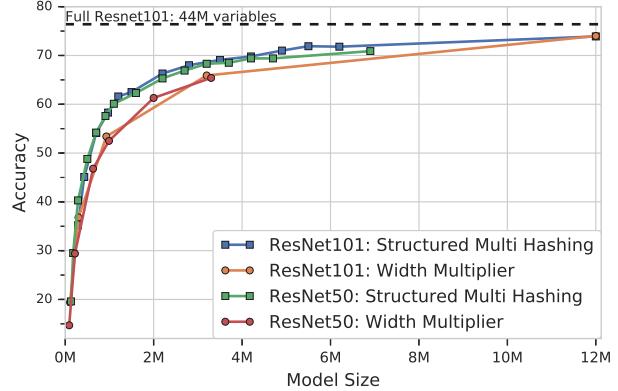


Figure 3: Accuracy vs Model Size with ResNet based models on Imagenet. We compare *SMH* to shrinking using Width Multiplier on ResNet50 and ResNet101. For both model types, *SMH* finds better tradeoffs between size and accuracy.

is one model trained to convergence to a specific target size. We compare with shrinking each one of the models using width multiplier. Note that *SMH* compresses the model more efficiently. For example, for an accuracy of 70% *SMH* models are half the size of the width multiplier models.

### 4.2. EfficientNet Models

The family of EfficientNet models [44] provides a natural and strong baseline for comparison. Using a large scale study of model hyper-parameters that affect size and latency, they propose a model scaling formula. By applying this formula, the authors propose 8 different models spanning from very large (B7) to very small (B0).

To evaluate the merit of our compression technique, we apply it to a subset of the EfficientNet family (namely B0 to B6). For each model, we use the size of smaller variants as target sizes. For example, we compress the B3 architecture to 5.3M, 7.8M, and 9.2M parameters corresponding to the sizes of B0, B1 and B2 variants. Figure 4 shows the results of applying this procedure. The bars are grouped by the target size to which they are compressed. Each bar indicates a starting model architecture.

Note that we can significantly improve accuracy for any desirable model size compared with the original model. For example, the original B0 model has an accuracy of 76.3%, but a B6 architecture compressed to the size of B0 has 80.3%. Even more drastic is comparing between groups — a B4 model compressed to the size of B1 outperforms the original B3 model even though it is 35% smaller.

### 4.3. MobileNets

MobileNets [20, 41, 19] are a family of models specifically targeted to mobile devices. These models have been

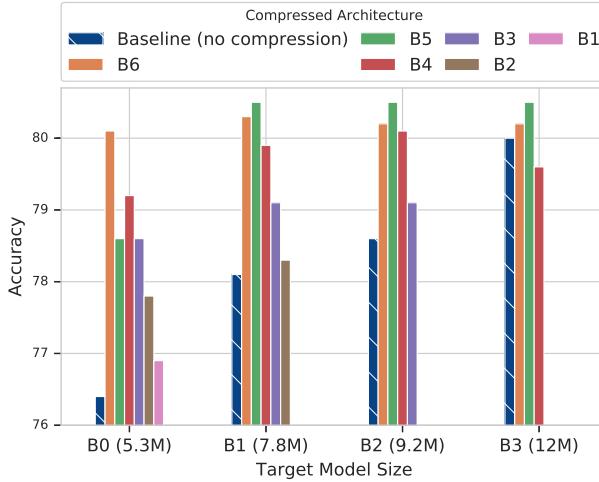


Figure 4: Accuracy vs Target Model Size on EfficientNet. Bars represent original architecture used. They are grouped by the size of the model after compression. For example, compressing a B6 architecture to the size of B0 has an accuracy of 80.1% compared with the original B0 at 76.4%. Note that the B5 and B6 architectures are slow to train and are harder to find stable hyper-parameters.

primarily optimized for FLOPs. However they are also significantly smaller than other models considered above.

In this section, we measure the efficacy of applying structured multi-hashing to MobileNetV2 and MobileNetV3. The comparison is presented on Figure 5. In addition to the width-multiplier as we did for ResNet, we also impose an additional, stronger baseline based on a combination of width-multiplier and resolution multiplier.

Width-multiplication reduces both FLOPs and model size, while structured-hashing only reduces the model size. To make a stronger baseline that produces comparable FLOPs, when we apply width multiplier  $\alpha$  to reduce the model size, we increase resolution by  $\approx \alpha$  that brings FLOP count back to the original cost.

Note, in contrast with [44], and following [40], we don't actually use higher resolution image. Instead, we simply up-sample the input. This guarantees that all models are trained on exactly the same data. It is interesting that for MobileNetV2, the multi-hash approach beats both baselines. On the other hand, for MobileNetV3, the stronger baseline produces slightly better trade-off curve around the full model. However, we note that the strong baseline is both slower and requires more memory to train (due to high spatial resolution of early tensors). In fact, we were unable to train the strong baseline example with multiplier less than 0.4, which required using input upsampled to 450x450. Another potential issue that limits the usefulness of the strong baseline is that it requires fractional upscaling which introduces image artifacts.

Compression Method	Target Model Size	Accuracy	Samples Per Second
<i>SMH</i>	5.3M	<b>0.774</b>	6060
1X Hash	5.3M	0.762	4000
2X Hash	5.3M	0.765	2800
10X Hash	5.3M	0.770	790
<i>SMH</i>	7.9M	<b>0.782</b>	6040
1X Hash	7.9M	0.773	3900
2X Hash	7.9M	0.775	2500
10X Hash	7.9M	0.779	760

Table 1: Hashing methods comparison on EfficientNet B2 model. We compare *SMH* to non-structured hashing with 1,2 and 10 hash functions. Note that not only is *SMH* more accurate but also much faster. We measure training samples per second on TPU V3 with a 4x4 topology.

#### 4.4. Compared to Non Structured Hashing

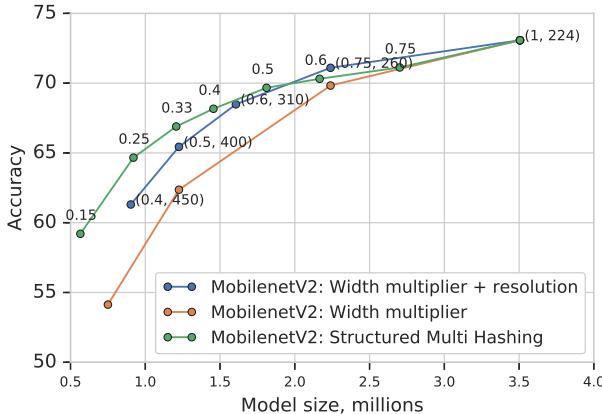
Here we compare the results of *SMH* to a single and multi-hashing baselines. We implement model hashing over the full set of network weights. For standard hashing we hash each weight into  $K \in \{1, 2, 10\}$  sets of trainable variables and use the sum reducer defined in Eq. (1). We compare the methods on an EfficientNet B2 model compressed to 5M and 7.9M trainable variables (the size of B0, and B1 respectively). Table 1 shows the results. *SMH* is both more accurate and much faster than standard hashing. The memory locality of *SMH* and its implementation as a matrix product result in this low overhead. Note that *SMH* in these experiments is using  $\sim 800$  hash functions.

#### 4.5. Extreme Compression

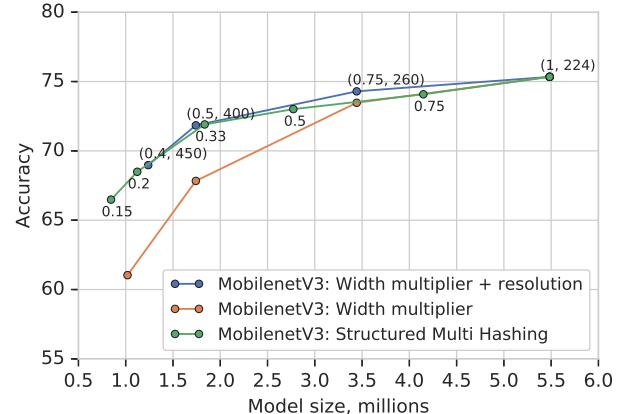
As noted above, deep networks are known to be over-parameterized. Here, we examine this notion further. We ask the question: *Can deep models be accurate when using an extremely small number of trainable variables?* Can this be done for an architecture that was not specifically designed for this purpose? To answer this question we perform two sets of experiments, on CIFAR10 [26] using a ResNet32 model, and on ImageNet using EfficientNet models.

In Figure 6 different compression method applied to a ResNet32 model trained on the CIFAR10 dataset are shown. First note that using our multi-hash approach we can effectively discard 75% of the variables in the model, without loss in performance. Furthermore, we can create a model with only 10% of the original size (only 50K variables) and still maintain an accuracy above 90%.

Secondly, we take EfficientNet B4 and B5 architectures and compress them using *SMH* by 10x to 2M and 3M parameters respectively. In Table 2 we compare them to vanilla EfficientNet models with similar accuracy and see



(a)



(b)

Figure 5: Applying *SMH* to MobilenetV2 (a) and MobilenetV3 (b) on ImageNet. We compare to two baselines. Width-multiplier: all layers are shrunk by a constant factor. Width-multiplier + resolution is a stronger baseline where we additionally up-sample the input image to maintain the same number of FLOPS. Label for strong baseline such as  $(0.4, 450)$  represent multiplier and up-sampled resolution respectively. Label for *SMH* is the target compression rate.

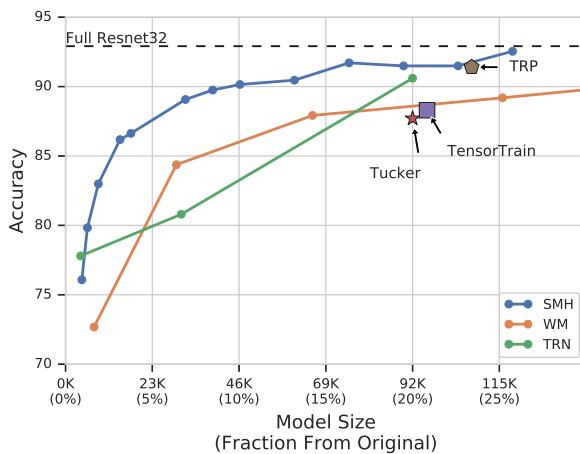


Figure 6: Accuracy vs Model Size on CIFAR10. We compare a number of compression methods on the ResNet32 model. We focus on the extreme compression regime. The rightmost points represent models that are 25% of the original ResNet32. Here TRP is low-rank ResNet32 from [49], Tucker, Tensor-Train and Tensor-Ring-Net (TRN) results obtained from [46], WM is width multiplier.

accuracy gains in both cases.

#### 4.6. Per-Layer Learnable Scale

To examine the benefits of adding a per-layer scale variable, as described in Sec 3.4, we train 16 EfficientNet based models on ImageNet. We train five different base models B0 to B4, and for each base model set a number of target sizes to compress to. We then train the models until conver-

Model	Accuracy	Model Size
B0	76.3%	5M
<i>SMH</i> <sub>2M</sub> B4	76.6%	2M
<i>SMH</i> <sub>3M</sub> B5	78.3%	3M
B1	78.8%	7.9M

Table 2: *SMH* B4 model is 60% smaller than B0 with the same accuracy. *SMH* B5 model is 40% smaller and 2% better than B0 and 63% smaller than B1 with slightly lower accuracy.

gence with and without the per-layer scale variable using the same hyper-parameters.

Table 3 shows the accuracy difference in accuracy when adding per-layer scale variables. We usually see improvements of about 0.5% to 1%. Also note that this procedure never hurts performance.

#### 4.7. Targeted Weight Compression

When compressing a neural network, one can choose to target all weights, or a smaller subset of them. For example, in all our experiments we do not hash any of the Batch Normalization variables, as those can be absorbed in the following convolution during inference.

One natural distinction between model weights is to separate those coming from convolutional layers which are usually in the early stages of a model, and those coming from fully connected layers which are commonly at the later part of the network. Figure 7 shows five base architectures from the EfficientNet family (B1, ..., B5) all compressed to the size of B0 (5M variables). For each base model we once

Base Model	Target Size	Fixed Scale	Learnable Scale	Accuracy Difference
B0	2.0M	71.9%	<b>73.1%</b>	1.2%
	3.0M	73.7%	<b>74.0%</b>	0.3%
B1	2.0M	73.4%	<b>74.2%</b>	0.8%
	3.0M	75.2%	<b>76.5%</b>	1.3%
	5.0M	76.4%	<b>76.8%</b>	0.4%
B2	2.0M	73.9%	<b>74.9%</b>	1.0%
	3.0M	75.7%	<b>77.1%</b>	1.4%
	5.0M	77.0%	<b>77.8%</b>	0.8%
B3	2.0M	75.1%	<b>75.5%</b>	0.4%
	5.0M	78.0%	<b>78.6%</b>	0.6%
	7.9M	78.4%	<b>79.1%</b>	0.7%
	9.3M	78.2%	<b>79.1%</b>	0.9%
B4	5.0M	78.8%	<b>79.2%</b>	0.4%
	7.9M	79.2%	<b>79.7%</b>	0.5%
	9.3M	79.3%	<b>79.5%</b>	0.2%

Table 3: Accuracy improvement when adding a per-layer learnable scale. Average accuracy gain is 0.7%, note that the per-layer scale is *always* beneficial.

compress it by hashing all the weights, and once by only hashing weights coming from the convolutions.

The differences are not big, indicating that the multi hashing constraint has enough flexibility to make useful trade-offs. Note also that when starting with smaller architectures (B1, B2) it is better to limit the hashing to the convolutions. The fully connected layers in those models are smaller and have less representation power to spare.

When starting with bigger models however (B4, B5) the trends reverses and higher accuracy is achieved when letting the multi-hash compress all layers. For these models, the fully connected layers have many parameters and without access to those layers the hash must compress the rest of the network more drastically. For example, the B5 architecture has 2M weights in its dense layer, out of 30M. When compressing it to the size of B0 without hashing the dense layer we now need to hash 28M weights into 3M variables, a 89% compression of those layers.

## 5. Discussion

In this paper we have presented an efficient model compression method that builds on the idea of weight hashing, while addressing its key limitations: We eliminate hash collisions by introducing a multi-hash and reduce framework which maps each weight in a model into a set of trainable variables, and computes its value using a reduce operation on the set. Memory locality is preserved by eschewing random hashing, and defining a structured mapping instead. The *SMH* approach can be represented as a matrix product and does not add material overhead to model latency.

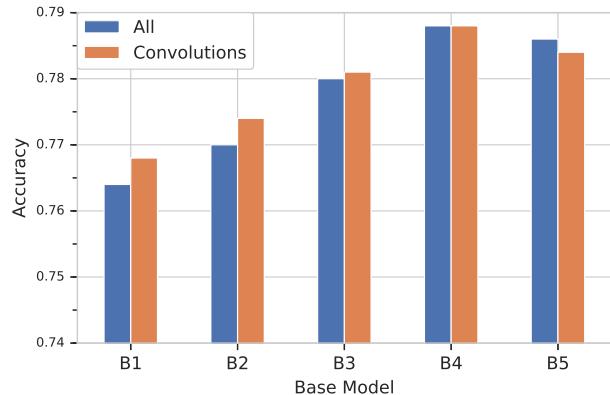


Figure 7: Compressing EfficientNet architectures to the size of B0 (5M). Bars are grouped by the architecture being compressed. We compare compressing all layers (blue), with hashing only convolutional layers (orange). There is a small difference - the method isn't sensitive to this design choice. Note that smaller models benefit from maintaining the fully connected layers untouched, while the larger base models gain from the flexibility to compress all layers.

We show that a well optimized hashed model can be strongly compressed with minimal loss in accuracy. We demonstrated our results on the widely used ResNet family of models, and on the newer and more powerful EfficientNet and MobileNet model family.

From a scientific perspective, model hashing is distinctly different from quantization or pruning. Model quantization changes the precision in which the underlying function is approximated, but does not change dimensionality of the approximator. Pruning induces some weight values to zero but this on its own has no effect on the overall dimension. If the pruning is strong enough to set complete rows of weight matrices to zero, or if it has a structured form, e.g. [14] it changes both the dimensionality of the approximator (number of variables) and its expressivity (number of layers, amount of non-linearity, etc). In contrast model hashing does not change precision, but affects only the dimension (i.e. number of variables).

Model hashing then provides a useful tool for exploring the role of the number of variables within an architecture family. Our results on the ResNet family of models shows that number of variables tracks closely with accuracy. ResNet101 and ResNet50 based models, compressed to the same number of parameters perform almost indistinguishably from each other. This is true both for our hashing technique, and for the width multiplier baseline.

In contrast this does not hold for the EfficientNet or MobileNet family, in which different architectures (e.g. B6 vs B4, or V2 vs V3) compressed to the same size, differ significantly in their accuracy. Clearly more work is needed before we can fully understand the role of parameter counts.

## References

[1] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014. 1

[2] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine learning and the bias-variance trade-off. *ArXiv*, abs/1812.11118, 2018. 1

[3] Miguel Á. Carreira-Perpiñán and Yerlan Idelbayev. “Learning-compression” algorithms for neural net pruning. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR’18)*, pages 8532–8541, 2018. 2

[4] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayelet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Society for Industrial and Applied Mathematics, 2004. 2

[5] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015. 2, 3

[6] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the limit of network quantization. In *Proc. of the 5th Int. Conf. Learning Representations (ICLR 2017)*, 2017. 2

[7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015. 2

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 1

[9] Misha Denil, Babak Shakibi, Laurent Dinh, Marc’Aurelio Ranzato, and Nando De Freitas. Predicting parameters in deep learning. In *Advances in neural information processing systems*, pages 2148–2156, 2013. 2

[10] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann Le-Cun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014. 1, 2

[11] Julian Faraone, Nicholas Fraser, Michaela Blott, and Philip H.W. Leong. SYQ: Learning symmetric quantization for efficient deep neural networks. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR’18)*, pages 4300–4309, 2018. 2

[12] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry Vetrov. Ultimate tensorization: compressing convolutional and FC layers alike. *arXiv:1611.03214 [cs.LG]*, Nov. 10 2016. 2

[13] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010. 4

[14] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018. 2, 8

[15] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Movshovitz, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947, 2000. 1

[16] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 2

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 1, 5

[18] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for model compression and acceleration on mobile devices. In *Proc. 15th European Conf. Computer Vision (ECCV’18)*, 2018. 2

[19] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *CoRR*, abs/1905.02244, 2019. 5

[20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. 2, 5

[21] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5MB model size. *arXiv:1602.07360 [cs.CV]*, Nov. 4 2016. 2

[22] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. Low-rank compression of neural nets: Learning the rank of each layer. In *Proc. of the 2020 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR’20)*, 2020. 2

[23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 1, 3

[24] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018. 2

[25] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017. 1

[26] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. 6

[27] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, 2016. 2

[28] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990. 2

[29] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. *arXiv preprint arXiv:1804.08838*, 2018. 1

[30] Chong Li and C. J. Richard Shi. Constrained optimization based low-rank approximation of deep neural networks. In *Proc. 15th European Conf. Computer Vision (ECCV'18)*, pages 746–761. 2

[31] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *Proceedings of International Conference on Learning Representations 2017*, 2017. 2

[32] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018. 2

[33] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and M. Welling. Data-free quantization through weight equalization and bias correction. In *ICCV*, volume abs/1906.04721, 2019. 2

[34] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010. 1

[35] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P. Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 442–450, 2015. 2

[36] Deniz Oktay, Johannes Ballé, Saurabh Singh, and Abhinav Shrivastava. Model compression by entropy penalized reparameterization. *ArXiv*, abs/1906.06624, 2019. 2

[37] Iosif Pinelis. The exp-normal distribution is infinitely divisible. *arXiv preprint arXiv:1803.09838*, 2018. 4

[38] Brandon Reagen, Udit Gupta, Robert Adolf, Michael M Mitzenmacher, Alexander M Rush, Gu-Yeon Wei, and David Brooks. Weightless: Lossy weight encoding for deep neural network compression. *arXiv preprint arXiv:1711.04686*, 2017. 2

[39] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4780–4789, 2019. 1

[40] Mark Sandler, Jonathan Baccash, Andrey Zhmoginov, and Andrew Howard. Non-discriminative data or weak model? on the relative importance of data and model resolution. *ArXiv*, abs/1909.03205, 2019. 6

[41] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018. 5

[42] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and SVN Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10(Nov):2615–2637, 2009. 2

[43] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '17*, pages 445–454, New York, NY, USA, 2017. ACM. 2

[44] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019. 2, 5, 6

[45] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013. 1

[46] Wenqi Wang, Yifan Sun, Brian Eriksson, Wenlin Wang, and Vaneet Aggarwal. Wide Compression: Tensor Ring nets. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pages 9329–9338, 2018. 2, 7

[47] Kilian Weinberger, Anirban Dasgupta, Josh Attenberg, John Langford, and Alex Smola. Feature hashing for large scale multitask learning. *arXiv preprint arXiv:0902.2206*, 2009. 2

[48] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016. 2

[49] Yuhui Xu, Yuxi Li, Shuai Zhang, Wei Wen, Bota Wang, Yingyong Qi, Yiran Chen, Weiyao Lin, and Hongkai Xiong. Trained rank pruning for efficient deep neural networks. *arXiv:1812.02402*, Dec. 8 2018. 2, 7

[50] Tien-Ju Yang, Andrew G. Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *ECCV*, 2018. 2

[51] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv:1606.06160*, July 17 2016. 2