Permute, Quantize, and Fine-tune: Efficient Compression of Neural Networks

Julieta Martinez[*,1] Jashan Shewakramani[*,1,2] Ting Wei Liu[*,1,2]
Ioan Andrei Bârsan[1,3] Wenyuan Zeng[1,3] Raquel Urtasun[1,3]

[1]Uber Advanced Technologies Group [2]University of Waterloo [3]University of Toronto
{julieta,jashan,tingwei.liu,andreib,wenyuan,urtasun}@uber.com

Abstract

Compressing large neural networks is an important step for their deployment in resource-constrained computational platforms. In this context, vector quantization is an appealing framework that expresses multiple parameters using a single code, and has recently achieved state-of-the-art network compression on a range of core vision and natural language processing tasks. Key to the success of vector quantization is deciding which parameter groups should be compressed together. Previous work has relied on heuristics that group the spatial dimension of individual convolutional filters, but a general solution remains unaddressed. This is desirable for pointwise convolutions (which dominate modern architectures), linear layers (which have no notion of spatial dimension), and convolutions (when more than one filter is compressed to the same codeword). In this paper we make the observation that the weights of two adjacent layers can be permuted while expressing the same function. We then establish a connection to rate-distortion theory and search for permutations that result in networks that are easier to compress. Finally, we rely on an annealed quantization algorithm to better compress the network and achieve higher final accuracy. We show results on image classification, object detection, and segmentation, reducing the gap with the uncompressed model by 40 to 70% w.r.t. the current state of the art. All our experiments can be reproduced using the code at https://github.com/uber-research/permute-quantize-finetune.

1. Introduction

State-of-the-art approaches to many computer vision tasks are currently based on deep neural networks. These networks often have large memory and computational requirements, limiting the range of hardware platforms on which they can operate. This poses a challenge for applications such as virtual reality and robotics, which naturally rely on mobile and low-power computational platforms for large-scale deployment. At the same time, these networks are often overparameterized [5], which implies that it is possible to compress them – thereby reducing their memory and computation demands – without much loss in accuracy.

Scalar quantization is a popular approach to network compression where each network parameter is compressed individually, thereby limiting the achievable compression rates. To address this limitation, a recent line of work has focused on vector quantization (VQ) [13, 47, 54], which compresses multiple parameters into a single code. Conspicuously, these approaches have recently achieved state-of-the-art compression-to-accuracy ratios on core computer vision and natural language processing tasks [10, 48].

A key advantage of VQ is that it can naturally exploit redundancies among groups of network parameters, for example, by grouping the spatial dimensions of convolutional filters in a single vector to achieve high compression rates. However, finding which network parameters should be compressed jointly can be challenging; for instance, there is no notion of spatial dimension in fully connected layers, and it is not clear how vectors should be formed when the vector size is larger than a single convolutional filter – which is always true for pointwise convolutions. Current approaches either employ clustering (*e.g.*, k-means) using the order of the weights as obtained by the network [13, 47, 54], which is suboptimal, or search for groups of parameters that, when compressed jointly, minimize the reconstruction error of the network activations [10, 48, 54], which is hard to optimize.

In this paper, we formalize the notion of redundancy among parameter groups using concepts from rate-distortion theory, and leverage this analysis to search for permutations of the network weights that yield functionally equivalent, yet easier-to-quantize networks. The result is Permute, Quantize, and Fine-tune (PQF), an efficient algorithm that first searches for permutations, codes and codebooks that minimize the reconstruction error of the network weights, and then uses gradient-based optimization to recover the accuracy of the uncompressed network. Our main contributions can be summarized as follows:

1. We study the invariance of neural networks under permutation of their weights, focusing on constraints in-

duced by the network topology. We then formulate a *permutation optimization* problem to find functionally equivalent networks that are easier to quantize. Our result focuses on improving a quantization lower bound of the weights; therefore

2. We use an efficient *annealed quantization algorithm* that reduces quantization error and leads to higher accuracy of the compressed networks. Finally,

3. We show that the reconstruction error of the network parameters is *inversely correlated* with the final network accuracy after gradient-based fine-tuning.

Put together, the above contributions define a novel method that produces state-of-the-art results in terms of model size vs. accuracy. We benchmark our method by compressing popular architectures for image classification, and object detection & segmentation, showcasing the wide applicability of our approach. Our results show a 40-60% relative error reduction on Imagenet object classification over the current state-of-the-art when compressing a ResNet-50 [21] down to about 3 MB (~31× compression). We also demonstrate a relative 60% (resp. 70%) error reduction in object detection (resp. mask segmentation) on COCO over previous work, by compressing a Mask-RCNN architecture down to about 6.6 MB (~26× compression).

2. Related Work

There is a vast literature on compressing neural networks. Efforts in this area can broadly be divided into pruning, low-rank approximations, and quantization.

Weight pruning: In its simplest form, weight pruning can be achieved by removing small weights [16, 18], or approximating the importance of each parameter using second-order terms [7, 19, 30]. More sophisticated approaches use meta-learning to obtain pruning policies that generalize to multiple models [22], or use regularization terms during training to reduce parameter count [36]. Most of these methods prune individual weights, and result in sparse networks that are difficult to accelerate on commonly available hardware. To address these issues, another line of work aims to remove unimportant channels, producing networks that are easier to accelerate in practice [23, 31, 38].

Low-rank approximations: These methods can achieve acceleration by design [6, 25, 29, 42], as they typically factorize the original weight matrix into several smaller matrices. As a result, the original computationally-heavy forward pass can be replaced by a multiplication of several smaller vectors and matrices.

Scalar quantization: These techniques constrain the number of bits that each parameter may take, in the extreme case using binary [4, 39, 44, 53] or ternary [57] values. 8-bit quantization methods have proven robust and efficient, which has motivated their native support by popular deep learning libraries such as PyTorch[1] and Tensorflow Lite[2], with acceleration often targeting CPUs. We refer the reader to the survey by [41] for a recent comprehensive overview of the subject. In this context, reducing each parameter to a single bit yields a theoretical compression ratio of 32× (although, in practice, fully-connected and batch norm layers are not quantized [39]). To obtain higher compression ratios, researchers have turned to vector quantization.

Vector quantization (VQ): VQ of neural networks was pioneered by Gong *et al*. [13], who investigated scalar, vector, and product quantization [26] (PQ) of fully-connected (FC) layers, which were the most memory-demanding layers of convolutional neural networks (CNNs) at the time. Wu *et al*. [54] used PQ to compress both FC and convolutional layers of CNNs; they noticed that minimizing the quantization error of the network parameters produces much worse results than minimizing the error of the activations, so they sequentially quantized the layers to minimize error accumulation. However, neither Gong *et al*. [13] nor Wu *et al*. [54], explored end-to-end training, which is necessary to recover the network accuracy as the compression ratio increases.

Son *et al*. [47] clustered 3×3 convolutions using vector quantization, and fine-tuned the centroids via gradient descent using additional bits to encode filter rotation, resulting in very compact codebooks. However, they did not explore the compression of FC layers nor pointwise convolutions (which dominate modern architectures), and did not explore the relationship of quantization error to accuracy.

Stock *et al*. [48] use PQ to compress convolutional and FC layers using a clustering technique designed to minimize the reconstruction error of the layer outputs (which is computationally expensive), followed by end-to-end training of the cluster centroids via distillation. However, their approach does not optimize the grouping of the network parameters for quantization, which we find to be crucial to obtain good compression. Chen *et al*. [2] improve upon the results of [48] by minimizing the reconstruction error of the parameters and the task loss jointly; however, their method also uses more fine-tuning epochs, so a direct comparison is hard.

Different from previous approaches, our method exploits the invariance of neural networks under permutation of their weights for the purpose of vector compression. Based on this observation, we draw connections to rate distortion theory, and use an efficient permutation optimization algorithm that makes the network easier to quantize. We also use an annealed clustering algorithm to further reduce quantization error, and show that there is a direct correlation between the

[1] pytorch.org/docs/stable/quantization.html
[2] tensorflow.org/lite/performance/post_training_quantization

quantization error of a network weights and its final accuracy after fine-tuning. These contributions result in an efficient method that largely outperforms its competitors on a wide range of applications.

3. Learning to Compress a Neural Network

In this paper we compress a neural network by compressing the weights of its layers. Specifically, instead of storing the weight matrix $\mathbf{W}$ of a layer explicitly, we learn an encoding $\mathcal{B}(\mathbf{W})$ that takes considerably less memory. Intuitively, we can decode $\mathcal{B}$ to a matrix $\widehat{\mathbf{W}}$ that is "close" to $\mathbf{W}$, and use $\widehat{\mathbf{W}}$ as the weight matrix for the layer. The idea is that if $\widehat{\mathbf{W}}$ is similar to $\mathbf{W}$, the activations of the layer should also be similar. Note that the encoding will be different for each of the layers.

3.1. Designing the Encoding

For a desired compression rate, we design the encoding $\mathcal{B}$ to consist of a codebook $\mathcal{C}$, a set of codes $\mathbf{B}$, and a permutation matrix $\mathbf{P}$. The permutation matrix preprocesses the weights so that they are easier to compress without affecting the input-output mapping of the network, while the codes and codebook attempt to express the permuted weights as accurately as possible using limited memory.

Codes and codebook: Let $\mathbf{W} \in \mathbb{R}^{m\times n}$ denote the weight matrix of a fully-connected (FC) layer, with m the input size of the layer, and n the size of its output. We split each column of $\mathbf{W}$ into column subvectors $\mathbf{w}_{i,j} \in \mathbb{R}^{d\times 1}$, which are then compressed individually:

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_{1,1} & \mathbf{w}_{1,2} & \cdots & \mathbf{w}_{1,n} \\ \mathbf{w}_{2,1} & \mathbf{w}_{2,2} & \cdots & \mathbf{w}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w}_{\hat{m},1} & \mathbf{w}_{\hat{m},2} & \cdots & \mathbf{w}_{\hat{m},n} \end{bmatrix}, \tag{1}$$

where $\hat{m} = m/d$, and $\hat{m}\cdot n$ is the total number of subvectors. Intuitively, larger d results in fewer subvectors and thus higher compression rates. The set $\{\mathbf{w}_{i,j}\}$ is thus a collection of d-dimensional blocks that can be used to construct $\mathbf{W}$.

Instead of storing all these subvectors, we approximate them by a smaller set $\mathcal{C} = \{\mathbf{c}(1), \ldots, \mathbf{c}(k)\} \subseteq \mathbb{R}^{d\times 1}$, which we call the *codebook* for the layer. We refer to the elements of $\mathcal{C}$ as *centroids*. Let $b_{i,j} \in \{1, \ldots, k\}$ be the index of the element in $\mathcal{C}$ that is closest to $\mathbf{w}_{i,j}$ in Euclidean space:

$$b_{i,j} = \arg\min_t \|\mathbf{w}_{i,j} - \mathbf{c}(t)\|_2^2, \tag{2}$$

The *codes* $\mathbf{B} = \{b_{i,j}\}$ are the indices of the codes in the codebook that best reconstruct every subvector $\{\mathbf{w}_{i,j}\}$. The approximation $\widehat{\mathbf{W}}$ of $\mathbf{W}$ is thus the matrix obtained by replacing each subvector $\mathbf{w}_{i,j}$ with $\mathbf{c}(b_{i,j})$:

$$\widehat{\mathbf{W}} = \begin{bmatrix} \mathbf{c}(b_{1,1}) & \mathbf{c}(b_{1,2}) & \cdots & \mathbf{c}(b_{1,n}) \\ \mathbf{c}(b_{2,1}) & \mathbf{c}(b_{2,2}) & \cdots & \mathbf{c}(b_{2,n}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{c}(b_{\hat{m},1}) & \mathbf{c}(b_{\hat{m},2}) & \cdots & \mathbf{c}(b_{\hat{m},n}) \end{bmatrix}. \tag{3}$$

We refer to the process of expressing the weight matrix in terms of codes and a codebook as *quantization*.

Permutation: The effectiveness of a given set of codes and codebooks depends on their ability to represent the original weight matrix $\mathbf{W}$ accurately. Intuitively, this is easier to achieve if the subvectors $\mathbf{w}_{i,j}$ are similar to one another. Therefore, it is natural to consider transformations of $\mathbf{W}$ that make the resulting subvectors easier to compress.

A feedforward network can be thought of as a directed acyclic graph (DAG), where nodes represent layers and edges represent the flow of information in the network. We refer to the starting node of an edge as a *parent* layer, and to the end node as a *child* layer. We note that the network is invariant under permutation of its weights, as long as the same permutation is applied to the output dimension for parent layers and the input dimension for children layers. Here, *our key insight is that we can search for permutations that make the network easier to quantize*.

Formally, consider a network comprised of two layers:

$$f(\mathbf{x}) = \phi(\mathbf{x}\mathbf{W}_2),\ \mathbf{W}_2 \in \mathbb{R}^{m\times n} \tag{4}$$

$$g(\mathbf{x}) = \phi(\mathbf{x}\mathbf{W}_1),\ \mathbf{W}_1 \in \mathbb{R}^{p\times m} \tag{5}$$

where ϕ represents a non-linear activation function. The network can be described as the function

$$f \circ g(\mathbf{x}) = f(g(\mathbf{x})) = \phi(g(\mathbf{x})\mathbf{W}_2), \tag{6}$$

where $\mathbf{x} \in \mathbb{R}^{1\times p}$ is the input to the network. Furthermore, from a topological point of view, g is the parent of f.

Given a permutation π of m elements $\pi : \{1, \ldots, m\} \to \{1, \ldots, m\}$, we denote $\mathbf{P}$ as the *permutation matrix* that results from reordering the rows of the $m \times m$ identity matrix according to π. Left-multiplying $\mathbf{P}$ with $\mathbf{X}$ has the effect of reordering the rows of $\mathbf{X}$ according to π.

Let $f_{\mathbf{P}_2}$ be the layer that results from applying the permutation matrix $\mathbf{P}_2$ to the input dimension of the weights of f:

$$f_{\mathbf{P}_2}(\mathbf{x}) = \phi(\mathbf{x}\mathbf{P}_2\mathbf{W}_2). \tag{7}$$

Analogously, let $g^{\mathbf{P}_2}$ be the layer that results from applying the permutation $\mathbf{P}_2$ to the output dimension of the weights of g:

$$g^{\mathbf{P}_2}(\mathbf{x}) = \phi(\mathbf{x}(\mathbf{P}_2\mathbf{W}_1^\top)^\top) = \phi(\mathbf{x}\mathbf{W}_1\mathbf{P}_2^\top). \tag{8}$$

Importantly, so long as ϕ is an element-wise operator, $g^{\mathbf{P}_2}$ produces the same output as g, only permuted:

$$g^{\mathbf{P}_2}(\mathbf{x}) = g(\mathbf{x})\mathbf{P}_2^\top, \tag{9}$$

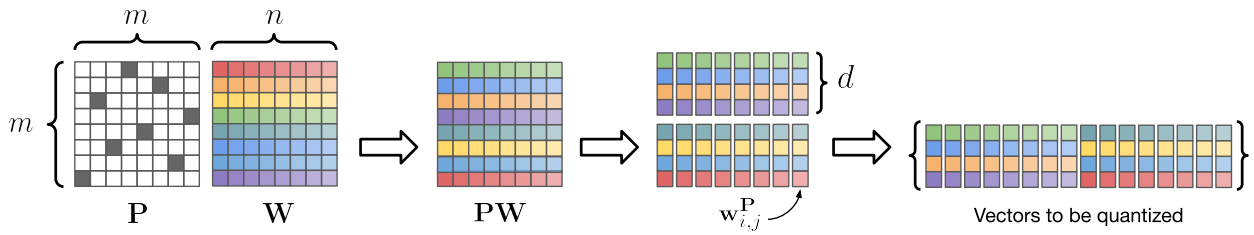

Figure 1: Permutation optimization of a fully-connected layer. Our goal is to find a permutation $\mathbf{P}$ of the weights $\mathbf{W}$ such that the resulting subvectors are easier to compress.

then we have

$$
\begin{aligned}
f_{\mathbf{P}_2} \circ g^{\mathbf{P}_2}(\mathbf{x}) &= f_{\mathbf{P}_2}(g^{\mathbf{P}_2}(\mathbf{x})) && (10)\\
&= f_{\mathbf{P}_2}(g(\mathbf{x})\mathbf{P}_2^\top) && (11)\\
&= \phi(g(\mathbf{x})\mathbf{P}_2^\top \mathbf{P}_2 \mathbf{W}_2) && (12)\\
&= \phi(g(\mathbf{x})\mathbf{W}_2) && (13)\\
&= f(g(\mathbf{x})) && (14)\\
&= f \circ g(\mathbf{x}), \quad \forall\, \mathbf{P}_2, \mathbf{x}. && (15)
\end{aligned}
$$

This *functional equivalence* has previously been used to characterize the optimization landscape of neural networks [1,43]. In contrast, here we focus on quantizing the permuted weight $\mathbf{P}_2\mathbf{W}_2$, and denote its subvectors as $\{\mathbf{w}_{i,j}^{\mathbf{P}_2}\}$. We depict the process of applying a permutation and obtaining new subvectors in Figure 1.

Extension to convolutional layers: The encoding of convolutional layers is closely related to that of fully-connected layers. Let $\mathbf{W} \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}} \times K \times K}$ denote the weights of a convolutional layer with C_{in} input channels, C_{out} output channels, and a kernel size of $K \times K$. The idea is to reshape $\mathbf{W}$ into a 2d matrix $\mathbf{W_r}$ of size $C_{\text{in}}K^2 \times C_{\text{out}}$, and then apply the same encoding method that we use with fully-connected layers. The result is an approximation $\widehat{\mathbf{W}}_\mathbf{r}$ to $\mathbf{W_r}$. We then apply the inverse of the reshaping operation on $\widehat{\mathbf{W}}_\mathbf{r}$ to get our approximation to $\mathbf{W}$.

When $K > 1$, we set the codeword size d to a multiple of K^2 and limit the permutation matrix $\mathbf{P}$ to have a block structure such that the spatial dimensions of filters are quantized together. For pointwise convolutions (*i.e.*, $K = 1$), we set d to 4 or 8, depending on the desired compression rate.

We have so far considered networks where each layer has a single parent and a single child (*i.e.*, the topology is a chain). We now consider architectures where some layers may have more than one child or more than one parent.

Extension beyond chain architectures: AlexNet [28] and VGG [46] are examples of popular architectures with a chain topology. As a consequence, each layer can have a different permutation. However, architectures with more complicated topologies have more constraints on the permutations that they admit.

For example, consider Figure 2, which depicts six res-blocks as used in the popular ResNet-50 architecture. We start by finding a permutation for layer `4a`, and realize that its parent is layer `3c`. We also notice that layers `3c` and `2c` must share the same permutation for the residual addition to have matching channels. By induction, this is also true of layers `1c` and `1d`, which are now all parents of our initial layer `4a`. These parents have children of their own (layers `2a`, `3a` and `4d`), so these must be counted as siblings of `4a`, and must share the same permutation as `4a`. However, note that all `b` and `c` layers are only children, so they can have their own independent permutation.

Operations such as reshaping and concatenation in parent layers may also affect the permutations that a layer can tolerate while preserving functional equivalence. For example, in the detection head of Mask-RCNN [20], the output (of shape $256 \times 7 \times 7$) of a convolutional layer is reshaped (to $12\,544$) before before entering a FC layer. Moreover, the same tensor is used in another convolutional layer (without reshaping) for mask prediction. Therefore, the FC layer and the child convolutional layer must share the same permutation. In this case, the FC layer must keep blocks of $7 \times 7 = 49$ contiguous dimensions together to respect the channel ordering of its parent (and to match the permutation of its sibling). Determining the maximum set of independent permutations that an arbitrary network may admit (and finding efficient algorithms to do so) is a problem we leave for future work.

3.2. Learning the Encoding

Our overarching goal is to learn an encoding of each layer such that the *final* output of the network is preserved. Towards this goal, we search for a set of codes, codebook, and permutation that minimizes the quantization error E_t of every layer t of the network:

$$
E_t = \min_{\mathbf{P}_t, \mathbf{B}_t, \mathcal{C}_t} \frac{1}{\hat{m}n} \left\| \widehat{\mathbf{W}}_t - \mathbf{P}_t \mathbf{W}_t \right\|_F^2. \qquad (16)
$$

Our optimization procedure consists of three steps:

1. **Permute:** We search for a permutation of each layer that results in subvectors that are easier to quantize. We do this by minimizing the determinant of the covariance of the resulting subvectors.

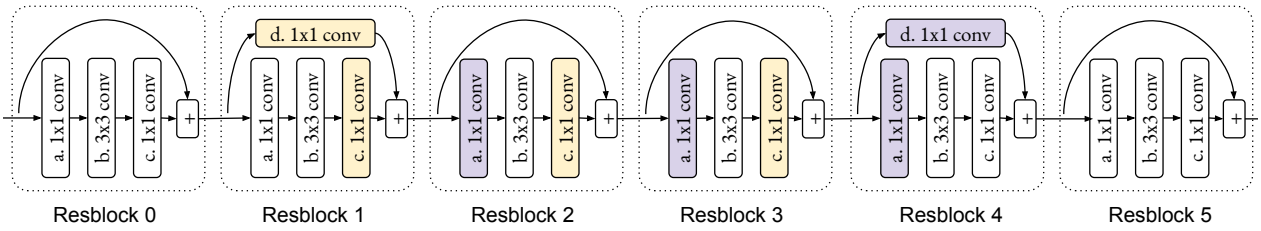

Figure 2: Parent-child dependencies in the Resblocks of a ResNet-50 architecture. Purple nodes are children and yellow nodes are parents, and must share the same permutation (in C_{in} for children and in C_{out} for parents) for the network to produce the same output.

2. **Quantize:** We obtain codes and codebooks for each layer by minimizing the difference between the approximated weight and the permuted weight.
3. **Fine-tune:** Finally, we jointly fine-tune all the codebooks with gradient-based optimization by minimizing the loss function of the original network over the training dataset.

We have found that minimizing the quantization error of the network weights (Eq. (16)) results in small inaccuracies that accumulate over multiple layers reducing performance; therefore, it is important to jointly fine-tune the network so that it can recover its original accuracy with gradient descent. We have also observed that *the quality of the intial reconstruction has a direct impact on the final network accuracy*. We now describe the three steps in detail.

3.2.1 Permute

In this step, our goal is to estimate a permutation $\mathbf{P}_t$ such that the permuted weight matrix $\mathbf{P}_t\mathbf{W}_t$ has subvectors $\{\mathbf{w}_{i,j}^{\mathbf{P}_t}\}$ that are easily quantizable. Intuitively, we want to minimize the spread of the vectors, as more compact vectors can be expressed more accurately given a fixed number of centroids. We now formalize this intuition and propose a simple algorithm to find good permutations.

A quantization lower bound: We assume that the weight subvectors that form the input to the quantization step come from a Gaussian distribution, $\mathbf{w}_{i,j}^{\mathbf{P}_t} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_t)$, with zero-mean and covariance $\mathbf{\Sigma}_t \in \mathbb{R}^{d\times d}$, which is a positive semi-definite matrix. Thanks to rate distortion theory [12], we know that the expected reconstruction error E_t must follow

$$E_t \geq k^{-\frac{2}{d}} d \left|\mathbf{\Sigma}_t\right|^{\frac{1}{d}}; \tag{17}$$

in other words, the error is lower-bounded by the determinant of the covariance of the subvectors of $\mathbf{P}_t\mathbf{W}_t$. We assume that we have access to a good minimizer such that, roughly, this bound is equal to the reconstruction error achieved by our quantization algorithm. Thus, for a fixed target compression bit-rate, we can focus on finding a permutation $\mathbf{P}_t$ that minimizes $\left|\mathbf{\Sigma}_t\right|$.

Searching for permutations: We make use of Expression (17) and focus on obtaining a permutation $\mathbf{P}_t$ that minimizes the determinant of the covariance of the set $\{\mathbf{w}_{i,j}^{\mathbf{P}_t}\}$. We follow an argument similar to that of Ge *et al*. [11], and note that the determinant of any positive semi-definite matrix $\mathbf{\Sigma}_t \in \mathbb{R}^{d\times d}$, with elements $\sigma_{i,j}^t$, satisfies Hadamard's inequality:

$$\left|\mathbf{\Sigma}_t\right| \leq \prod_{i=1}^{d} \sigma_{i,i}^t; \tag{18}$$

that is, the determinant of $\mathbf{\Sigma}_t$ is upper-bounded by the product of its diagonal elements.

Motivated by this inequality, we greedily obtain an initial $\mathbf{P}_t$ that minimizes the product of the diagonal elements of $\mathbf{\Sigma}_t$ by creating d buckets of row indices, each with capacity to hold $\hat{m} = m/d$ elements. We then compute the variance of each row of $\mathbf{W}_t$, and greedily assign each row index to the non-full bucket that results in lowest bucket variance. Finally, we obtain $\mathbf{P}_t$ by interlacing rows from the buckets so that rows from the same bucket are placed d rows apart. $K \times K$ convolutions can be handled similarly, assuming that $\mathbf{P}_t$ has a block structure, and making use of the more general Fischer's inequality. Please refer to the supplementary material for more details.

There are $\mathcal{O}(m!)$ possible permutations of $\mathbf{W}_t$, so greedy algorithms are bound to have limitations on the quality of the solution that they can find. Thus, we refine our solution via stochastic local search [24]. Specifically, we iteratively improve the candidate permutation by flipping two dimensions chosen at random, and keeping the new permutation if it results in a set of subvectors whose covariance has lower determinant $\left|\mathbf{\Sigma}_t\right|$. We repeat this procedure for a fixed number of iterations, and return the best permutation obtained.

3.2.2 Quantize

In this step, we estimate the codes $\mathbf{B}_t$ and codebook $\mathcal{C}_t$ that approximate the permuted weight $\mathbf{P}_t\mathbf{W}_t$. Given a fixed permutation, this is equivalent to the well-known k-means problem. We use an *annealed* quantization algorithm called SR-C originally due to Zeger *et al*. [56], and recently adapted by Martinez *et al*. [40] to multi-codebook quantization. Empirically, SR-C achieves lower quantization error than the

vanilla k-means algorithm, and is thus a better minimizer of Expr. (17).

A stochastic relaxation of clustering: The quantization lower bound from Expression (17) suggests that the k-means algorithm can be annealed by scheduling a perturbation such that the determinant of the covariance of the set $\{\mathbf{w}_{i,j}^{\mathbf{P}_t}\}$ decreases over time. Due to Hadamard's inequality (*i.e.*, Expresion (18)), this can be achieved by adding decreasing amounts of noise to $\mathbf{w}_{i,j}^{\mathbf{P}_t}$ sampled from a zero-mean Gaussian with diagonal covariance.

Therefore, after randomly initializing the codes, we iteratively update the codebook and codes with a noisy codebook update (which operates on subvectors with additive diagonalized Gaussian noise), and a standard k-means code update. We decay the noise according to the schedule $(1-(\tau/I))^{\gamma}$, where τ is the current iteration, I is the total number of update iterations, and γ is a constant. We use $\gamma = 0.5$ in all our experiments. For a detailed description, please refer to Algorithm 1.

Algorithm 1 SR-C: Stochastic relaxation of k-means.

procedure SR-C($\{\mathbf{w}_{i,j}^{\mathbf{P}_t}\}, \mathbf{\Sigma}_t, k, T, \gamma$)
  $\mathbf{B}_t \leftarrow$ InitializeCodes(k)
  for $\tau \leftarrow 1, \ldots, T$ **do**
    # Add scheduled noise to subvectors
    for $\mathbf{w}_{i,j}^{\mathbf{P}_t} \in \{\mathbf{w}_{i,j}^{\mathbf{P}_t}\}$ **do**
      $\mathbf{x}_{i,j} \sim \mathcal{N}(\mathbf{0}, \mathrm{diag}(\mathbf{\Sigma}_t))$
      $\hat{\mathbf{w}}_{i,j}^{\mathbf{P}_t} \leftarrow \mathbf{w}_{i,j}^{\mathbf{P}_t} + (\mathbf{x}_{i,j} \times (1 - (\tau/I))^{\gamma})$
    end for
    # Noisy codebook update
    $\mathcal{C}_t \leftarrow \arg\min_{\mathcal{C}} \sum_{i,j} \|\hat{\mathbf{w}}_{i,j}^{\mathbf{P}_t} - \mathbf{c}(b_{i,j})\|_2^2$
    # Regular codes update
    $\mathbf{B}_t \leftarrow \arg\min_{\mathbf{B}} \sum_{i,j} \|\mathbf{w}_{i,j}^{\mathbf{P}_t} - \mathbf{c}(b_{i,j})\|_2^2$
  end for
  return $\mathbf{B}_t, \mathcal{C}_t$
end procedure

3.2.3 Fine-tune

Encoding each layer independently causes errors in the activations to accumulate, resulting in degradation of performance. It is thus important to fine-tune the encoding in order to recover the original accuracy of the network. In particular, we fix the codes and permutations for the remainder of the procedure.

Let $\mathcal{L}$ be the *original* loss function of the network (*e.g.*, cross-entropy for classification). We note that $\mathcal{L}$ is differentiable with respect to each of the learned centroids – since these are continuous – so we use the original training set to fine-tune the centroids with gradient-based learning:

$$\mathbf{c}(i) \leftarrow \mathbf{c}(i) - u\left(\frac{\partial \mathcal{L}}{\partial \mathbf{c}(i)}, \theta\right), \tag{19}$$

Model	Regime	d_K	d_{pw}	d_{fc}
ResNet-18	Small blocks	K^2	4	4
	Large blocks	$2K^2$	4	4
ResNet-50	Small blocks	K^2	4	4
	Large blocks	$2K^2$	8	4

Table 1: Subvector sizes and compression regimes.

where $u(\cdot,\cdot)$ is an update rule (such as SGD, RMSProp [50] or Adam [27]) with hyperparameters θ (such as learning rate, momentum, and decay rates).

4. Experiments

We test our method on ResNet [21] architectures for image classification and Mask R-CNN [20] for object detection and instance segmentation. We compress standard ResNet-18 and ResNet-50 models that have been pretrained on ImageNet, taking the weights directly from the PyTorch model zoo. We train different networks with $k \in \{256, 512, 1024, 2048\}$. We also clamp the size of the codebook for each layer to $\min(k, n \times C_{\mathrm{out}}/4)$.

Small vs. large block sizes: To further assess the trade-off between compression and accuracy, we use two compression regimes. In the *large blocks* regime, we use a larger subvector size d for each layer, which allows the weight matrix to be encoded with fewer codes, and thus leads to higher compression rates. To describe the subvector sizes we use for each layer, we let d_K denote the subvector size for a convolutional layer with filters of size $K \times K$. In the special case when $K = 1$, corresponding to a pointwise convolution, we denote the subvector size by d_{pw}. Finally, fully-connected layers have a subvector size of d_{fc}. We summarize our subvector sizes for each model and compression regime in Table 1.

Bit allocation: We compress all the fully-connected and convolutional layers of a network. However, following [48], we do not compress the first convolutional layer (since it occupies less than 0.05% of the network size), the bias of the fully-connected layers, or the batchnorm layers. While we train with 32-bit floats, we store our final model using 16-bit floats, which has a negligible impact on validation accuracy (less than 0.02%). Finally, we fuse batchnorm layers into two vectors, which can be done with algebraic manipulation and is a trick normally used to speed up inference. Please refer to the supplementary material for a detailed breakdown of the bit allocation in our models.

Hyperparameters: We use a batch size of 128 for ResNet-18 and a batch size of 64 for ResNet-50. For annealed k-means, we implement SR-C in the GPU, and run it for 1 000 iterations. We fine-tune the codebooks for 9 epochs using Adam [27] with an initial learning rate of 10^{-3}, which is

	Ratio	Size	Acc.	Gap
Semi-sup R50 [55]	–	97.50 MB	79.30	–
BGD [48]	19×	5.20 MB	76.12	3.18
Semi-sup R50 [55]	–	97.50 MB	*78.72	–
Our PQF	19×	5.09 MB	**77.15**	1.57

Table 2: ImageNet classification starting from a semi-supervised ResNet-50. We set a new state of the art in terms of accuracy vs model size. *Reproduced from downloaded model.

gradually reduced to 10^{-6} using cosine annealing [37]. Fine-tuning is the most expensive part of this process, and takes around 8 hours both for ResNet-18 (with 1 GPU) and for ResNet-50 (with 4 GPUs). In the latter case, we scale the learning rate by a factor of 4, following Goyal *et al.* [14]. For permutation optimization, we perform 1 000 local search iterations; this is done in the CPU in parallel for each independent permutation. This process takes less than 5 minutes for ResNet-18, and about 10 minutes for ResNet-50 on a 12-core CPU.

Baselines: We compare the results of our method against a variety of network compression methods: Binary Weight Network (BWN) [44], Trained Ternary Quantization (TTQ) [57], ABC-Net [35], LR-Net [45], Deep Compression (DC) [17], Hardware-Aware Automated Quantization (HAQ) [52], CLIP-Q [51], Hessian AWare Quantization of Neural Networks with Mixed Precision (HAWQ) [9], and HAWQ-V2 [8]. We compare extensively against the recently-proposed Bit Goes Down (BGD) method of [48] because it is the current state of the art by a large margin. BGD uses as initialization the method due to Wu *et al.* [54], and thus subsumes it. All results presented are taken either from the original papers, or from two additional surveys [3, 15].

4.1. Image Classification

A summary of our results can be found in Figure 3. From the Figure, it is clear that our method outperforms all its competitors. On ResNet-18 for example, we can surpass the performance of ABC-Net (M=5) with our *small blocks* models at roughly 3× the compression rate. Our biggest improvement generally comes from higher compression rates, and is especially apparent for the larger ResNet-50. When using large blocks and $k = 256$ centroids, we obtain a top-1 accuracy of **72.18**% using only ∼3 MB of memory. This represents an absolute ∼**4**% improvement over the state of the art. On ResNet-50, our method consistently reduces the remaining error by 40-60% w.r.t. the state of the art.

Semi-supervised ResNet-50: We also benchmark our method using a stronger backbone as a starting point. We start from the recently released ResNet-50 model due to Yalniz *et al.* [55], which has been pre-trained on unlabelled images from the YFCC100M dataset [49], and fine-tuned on ImageNet. While the accuracy of this model is reported to be 79.30%, we obtain a slightly lower 78.72% after downloading the publicly-available model[3]; (contacting the authors we learned that the previous, slightly more accurate model, is no longer available for download). We use the *small blocks* compression regime with $k = 256$, mirroring the procedure described previously.

We show our results in Table 2, where our model attains a top-1 accuracy of **77.15**%. This means that we are able to outperform previous work by over 1% absolute accuracy, with a much smaller gap w.r.t. the uncompressed model. We find this result particularly interesting, as we originally expected distillation to be necessary to transfer the knowledge of the larger network pretrained on a large corpus of unlabeled images. However, our results show that at least part of this knowledge is retained through the initialization and structure that low-error clustering imposes on the compressed network.

Perm.	SR-C	Adam	Acc.	Δ
			62.29	−1.02
✓			62.55	−0.76
✓	✓		62.92	−0.39
✓	✓	✓	**63.31**	0.00

Table 3: Ablation study. ResNet18 on ImageNet w/large blocks.

Ablation study: In Table 3, we show results for ResNet-18 using large blocks, for which we obtain a final accuracy of **63.31%**. We add permutation optimization (Sec. 3.2.1), annealed k-means, as opposed to plain k-means (called SR-C in Sec. 3.2.2), and the use of the Adam optimizer with cosine annealing instead of plain SGD, as in previous work. From the Table, we can see that all our components are important and complementary to achieve top accuracy. It is also interesting to note that a baseline that simply does k-means and SGD fine-tuning is already ∼1% better than the current state-of-the-art. Since both annealed k-means and permutation optimization directly reduce quantization error before fine-tuning, these experiments demonstrate that minimizing the quantization error of the weights leads to higher final network accuracy.

4.2. Object Detection and Segmentation

We also benchmark our method on the task of object detection by compressing the popular ResNet-50 Mask-RCNN FPN architecture [20] using the MS COCO 2017 dataset [34]. We start from the pretrained model available on the PyTorch model zoo, and apply the same procedure described above for all the convolutional and linear layers (plus one deconvolutional layer, which we treat as a convolutional layer for the

[3] https://github.com/facebookresearch/semi-supervised-ImageNet1K-models

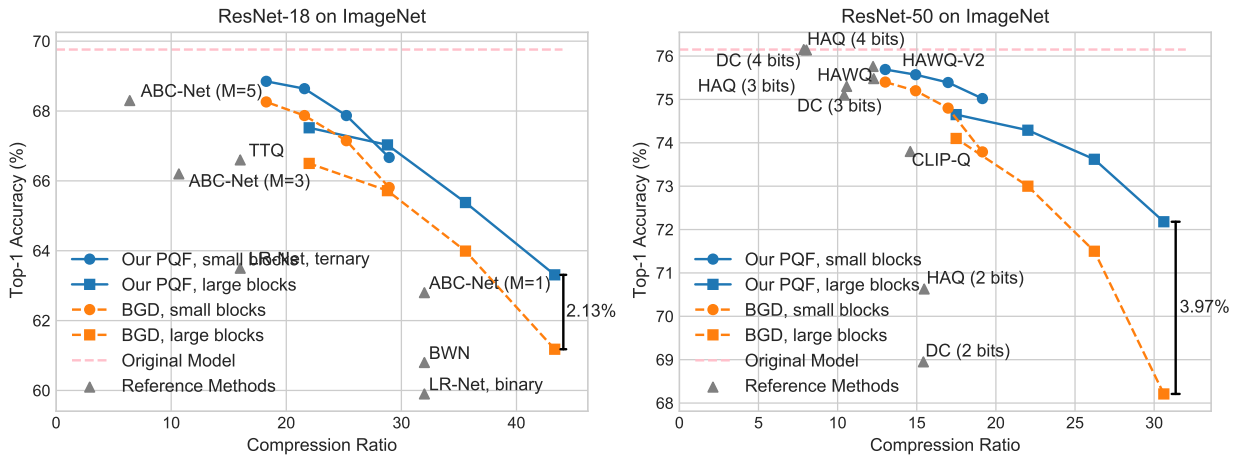

Figure 3: Compression results on ResNet-18 and ResNet-50. We compare accuracy vs. model size, using models from the PyTorch zoo as a starting point. In general, our method achieves higher accuracy compared to previous work.

	Size	Ratio	AP^{bb}	AP^{bb}_{50}	AP^{bb}_{75}	AP^{mk}	AP^{mk}_{50}	AP^{mk}_{75}
RetinaNet [33] (uncompressed)	145.00 MB	–	35.6	–	–	–	–	–
Direct	18.13 MB	8.0×	31.5	–	–	–	–	–
FQN [32]	18.13 MB	8.0×	32.5	51.5	34.7	–	–	–
HAWQ-V2 [8]	17.90 MB	8.1×	34.8	–	–	–	–	–
Mask-RCNN R-50 FPN [20] (uncompressed)	169.40 MB	–	37.9	59.2	41.1	34.6	56.0	36.8
BGD [48]	6.65 MB	26.0×	33.9	55.5	36.2	30.8	52.0	32.2
Our PQF	6.65 MB	26.0×	**36.3**	**57.9**	**39.4**	**33.5**	**54.7**	**35.6**

Table 4: Object detection results on MS COCO 2017. We compress a Mask R-CNN network with a ResNet-50 backbone, and include different object detection architectures used by other baselines. We report both bounding box (bb) and mask (mk) metrics for Mask R-CNN. We also report the accuracy at different IoU when available. The memory taken by [48] corresponds to the (correct) latest version on arXiv.

	Ratio	AP^{bb}	AP^{mk}
Mask-RCNN R-50 FPN [20]	–	37.9	34.6
BGD [48]	26.0×	33.9	30.8
Our PQF (no perm., no SR-C)		35.6	33.0
Our PQF (no perm.)	26.0×	35.8	33.1
Our PQF (full)		**36.3**	**33.5**

Table 5: Ablation results results on MS COCO 2017. Permutation optimization is particularly important for Mask-RCNN

purpose of compression). We use the small blocks regime with $k = 256$ centroids, for a model of 6.65 MB.

We compress and fine-tune the network on a single Nvidia GTX 1080Ti GPU with a batch size of 2 for 4 epochs. As before, we use Adam [27] and cosine annealing [37], but with an initial learning rate of 5×10^{-5}. Our results are presented in Table 4. We also compare against recent baselines such as the Fully Quantized Network (FQN) [32], and the second version of Hessian Aware Quantization (HAWQ-V2) [8], which showcase results compressing RetinaNet [33].

Our method obtains a box AP of 36.3, and a mask AP of 33.5, which represent improvements of **2.4**% and **2.7**% over the best previously reported result, closing the gap to the uncompressed model by 60-70%. Compared to BGD [48], we also use fewer computational resources, as they used 8 V100 GPUs and distributed training for compression, while we use a single 1080Ti GPU. In Table 5, we show again that using both SR-C and permutation optimization is crucial to obtain the best results. These results demonstrate the ability of our method to generalize to more complex tasks beyond image classification.

5. Conclusion

We have demonstrated that the quantization error of the weights of a neural network is inversely correlated with its accuracy after codebook fine tuning. We have further proposed a method that exploits the functional equivalence of the network under permutation of its weights to find configurations of the weights that are easier to quantize. We have also shown that using an annealed k-means algorithm further reduces quantization error and improves final network accuracy. On ResNet-50, our method closes the relative gap to the uncompressed model by 40-70% compared to the previous state-of-the-art in a variety of visual tasks.

References

[1] An Mei Chen, Haw-minn Lu, and Robert Hecht-Nielsen. On the geometry of feedforward neural network error surfaces. *Neural computation*, 5(6):910–927, 1993. 4

[2] Weihan Chen, Peisong Wang, and Jian Cheng. Towards convolutional neural networks compression via global & progressive product quantization. In *BMVC*, 2020. 2

[3] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey on model compression and acceleration for deep neural networks. *CoRR*, 2017. 7

[4] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to + 1 or - 1. *arXiv preprint arXiv:1602.02830*, 2016. 2

[5] 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*, 2013. 1

[6] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, 2014. 2

[7] Xin Dong, Shangyu Chen, and Sinno Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Advances in Neural Information Processing Systems*, 2017. 2

[8] Zhen Dong, Zhewei Yao, Yaohui Cai, Daiyaan Arfreen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ-V2: Hessian aware trace-weighted quantization of neural networks. In *Advances in Neural Information Processing Systems*, 2020. 7, 8

[9] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Hessian-aware quantization of neural networks with mixed precision. In *ICCV*, 2019. 7

[10] Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Rémi Gribonval, Hervé Jégou, and Armand Joulin. Training with quantization noise for extreme model compression. In *ICLR*, 2021. 1

[11] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *CVPR*, 2013. 5

[12] Allen Gersho and Robert M Gray. *Vector quantization and signal compression*, chapter 8, pages 228–243. Springer Science & Business Media, 1991. 5

[13] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 1, 2

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017. 7

[15] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018. 7

[16] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient DNNs. In *Advances In Neural Information Processing Systems*, 2016. 2

[17] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *ICLR*, 2016. 7

[18] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, 2015. 2

[19] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems*, 1993. 2

[20] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In *ICCV*, 2017. 4, 6, 7, 8

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 2, 6

[22] 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 *ECCV*, 2018. 2

[23] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, 2017. 2

[24] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004. 5

[25] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *BMVC*, 2014. 2

[26] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Patten Analysis and Machine Intelligence*, 33(1), 2010. 2

[27] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015. 6, 8

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012. 4

[29] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. In *ICLR*, 2015. 2

[30] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, 1990. 2

[31] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *ICLR*, 2016. 2

[32] Rundong Li, Yan Wang, Feng Liang, Hongwei Qin, Junjie Yan, and Rui Fan. Fully quantized network for object detection. In *CVPR*, 2019. 8

[33] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Dollár. Focal loss for dense object detection. In *ICCV*, pages 2980–2988, 2017. 8

[34] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnich. Microsoft COCO: Common objects in context. In *ECCV*, 2014. 7

[35] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, 2017. 7

[36] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017. 2

[37] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic gradient descent with warm restarts. In *ICLR*, 2017. 7, 8

[38] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *ICCV*, 2017. 2

[39] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. In *ICLR*, 2020. 2

[40] Julieta Martinez, Shobhit Zakhmi, Holger H Hoos, and James J Little. LSQ++: Lower running time and higher recall in multi-codebook quantization. In *ECCV*, 2018. 5

[41] James O' Neill. An overview of neural network compression. *arXiv preprint arXiv:2006.03669*, 2020. 2

[42] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, 2015. 2

[43] A Emin Orhan and Xaq Pitkow. Skip connections eliminate singularities. In *ICLR*, 2018. 4

[44] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016. 2, 7

[45] Oran Shayer, Dan Levi, and Ethan Fetaya. Learning discrete weights using the local reparameterization trick. In *ICLR*, 2018. 7

[46] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015. 4

[47] Sanghyun Son, Seungjun Nah, and Kyoung Mu Lee. Clustering convolutional kernels to compress deep neural networks. In *ECCV*, pages 216–232, 2018. 1, 2

[48] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the bit goes down: Revisiting the quantization of neural networks. In *ICLR*, 2020. 1, 2, 6, 7, 8

[49] Bart Thomee, David A Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. YFCC100M: The new data in multimedia research. *Communications of the ACM*, 59(2):64–73, 2016. 7

[50] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012. 6

[51] Frederick Tung and Greg Mori. Deep neural network compression by in-parallel pruning-quantization. *IEEE Transactions on Patten Analysis and Machine Intelligence*, 2019. 7

[52] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-aware automated quantization with mixed precision. In *CVPR*, 2019. 7

[53] Ziwei Wang, Jiwen Lu, Chenxin Tao, Jie Zhou, and Qi Tian. Learning channel-wise interactions for binary convolutional neural networks. In *CVPR*, 2019. 2

[54] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *CVPR*, 2016. 1, 2, 7

[55] I Zeki Yalniz, Hervé Jégou, Kan Chen, Manohar Paluri, and Dhruv Mahajan. Billion-scale semi-supervised learning for image classification. *arXiv preprint arXiv:1905.00546*, 2019. 7

[56] Kenneth Zeger, Jacques Vaisey, Allen Gersho, et al. Globally optimal vector quantizer design by stochastic relaxation. *IEEE Transactions on Signal Processing*, 40(2):310–322, 1992. 5

[57] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. In *ICLR*, 2017. 2, 7