DSConv: Efficient Convolution Operator

Marcelo Gennari do Nascimento
University of Oxford
Active Vision Lab
marcelo@robots.ox.ac.uk

Roger Fawcett
Intel Corporation
https://www.omnitek.tv/about
roger.fawcett@intel.com

Victor Adrian Prisacariu
University of Oxford
Active Vision Lab
victor@robots.ox.ac.uk

Abstract

Quantization is a popular way of increasing the speed and lowering the memory usage of Convolution Neural Networks (CNNs). When labelled training data is available, network weights and activations have successfully been quantized down to 1-bit. The same cannot be said about the scenario when labelled training data is not available, e.g. *when quantizing a pre-trained model, where current approaches show, at best, no loss of accuracy at 8-bit quantizations.*

We introduce DSConv, a flexible quantized convolution operator that replaces single-precision operations with their far less expensive integer counterparts, while maintaining the probability distributions over both the kernel weights and the outputs. We test our model as a plug-and-play replacement for standard convolution on most popular neural network architectures, ResNet, DenseNet, GoogLeNet, AlexNet and VGG-Net and demonstrate state-of-the-art results, with less than 1% loss of accuracy, without retraining, using only 4-bit quantization. We also show how a distillation-based adaptation stage with unlabelled data can improve results even further.

1. Introduction

A popular method to make neural networks faster and use less memory is quantization, which replaces 32-bit floating point weights and, potentially, activations with lower bit (*i.e*. lower precision) representations, while aiming to maintain accuracy.

Quantization is often used in neural network *compression*. This aims to reduce the memory occupied by the network weights as much as possible to, for example, lower the overall memory footprint required to store the network. It can also be used to increase neural network inference speed (*fast inference*), when applied to both weights and activations, by substituting expensive floating-point Multiply and Accumulate (MAC) operations with cheaper alternatives such as integer, bitwise operations or addition-only operations.

The best quantization results are achieved when labelled training data is available, as the quantized model can be fitted to the dataset, which feeds the training algorithm with prior knowledge of what the activation maps will look like and what the expected output will be. Maintaining a high accuracy becomes much more difficult when only a pre-trained model is available.

In this paper we focus on this latter scenario, and quantize both weights and activations to produce neural networks that are both smaller and have faster inference. Our key insight is that, in the absence of training data, this can be best achieved by forcing the probability distributions over the weights and activations of the low precision quantized model to mirror those of the original full-precision model. We introduce a novel convolution operator, which we call *DSConv*, that factorises the convolution weights into (i) a low-precision component with the same size as the original kernel and (ii) a high-precision distribution shift component, with a variable size (*e.g*. as small as one *float 32* value per kernel). A similar procedure, inspired by the block floating point approach [35], is used to quantize activations. We also show that accuracy can be improved when using a distillation [19] inspired weight adaptation approach, that uses the original pre-trained model and unlabelled input data.

The main contribution of this paper is the introduction of a convolution operator that (i) serves as a “drop-in and play” replacement for standard convolution and uses low-bit fixed point computation for the bulk of operations without the need of retraining using labelled data, and (ii) provides a hyperparameter that can be tuned to favor accuracy or memory usage/speed of computation for any given task. Our quantization strategy is able to achieve state-of-the-art results, as demonstrated by our experimental section.

The remainder of this paper is structured as follows. §2 presents the previous papers on quantization. §3 explains the method in detail. §4 shows the results of the experiments performed in a variety of architectures and settings. §5 concludes the paper with a discussion on its performance and possible applications and limitations.

2. Related Work

The use of low-bit width networks saves a significant amount of memory and computation, especially when targeted to custom hardware. For example, for 8-bit operations, [26] reports up to 10x increase in speed, and [12] reports up to 30x in energy saving and chip area. We categorize the previous research that tries to increase the neural network efficiency in two groups:

Quantization with labelled data. Most research in neural network quantization has focused on problems that involve retraining, by either starting from scratch or by adapting from an existing pre-trained network. BinaryConnect, BWN and TWN [11, 32, 28] use 1-bit and ternary weights to make the FP-MACS addition only. XNOR-Net and BNN [32, 11] applied 1-bit quantized weight and activations to ImageNet for fast inference, at the cost of a significant drop in accuracy. WRPN [30] improved this accuracy by using wider versions of the same architectures. Early demonstrations of low-bit network acceleration in custom hardware include Ristretto [16], which also uses data to quantize the network to 8-bit models. Many other papers followed, by also training the quantization scheme, using binary basis vectors, such as in LQ-Nets [38], Halfway Gaussian Quantization (HWGQ) [6], and ABC-Net [29]. DoReFa-Net [40] also quantized gradients, alongside weights and activations.

The compression problem has also mostly been dealt with by using retraining with access to labelled data. In DeepCompression [17], two of the three steps of the algorithm require retraining (pruning and quantization), with Huffman Encoding being performed without the need for data. HashedNet [7] use the "hashing trick" to save significant amounts of memory when storing the network, but still require labelled data for tuning. The more recent approach [31], uses distillation [3, 19] obtain compressed weights, but also requires the full labelled training set.

Several approaches introduce novel float data formats. Examples are Dynamic Fixed Point [10], which substitutes the normal floating point numbers with a mix of both fixed and floating point; and Flexpoint [25], which aims to leverage the range of floating point numbers and the computational complexity of fixed point and promises to perform forward and backwards operations with limited range. The idea of substituting the representation of single-precision (FP32) values in favour of other formats is also adopted in the *bfloat16* format in Tensorflow [2], which employs the *binary float 16* format that uses 7-bits for the mantissa instead of the usual 23-bits.

Quantization without labelled data. Whereas the problem of quantizing with labelled data has been researched extensively, the problem of quantizing without data has received far less attention. Recent papers that explore this possibility are [39, 8, 23, 4], which either report results only for 8-bit quantization or employ calibration data of some sort - *i.e.* an unlabelled small fraction of the validation dataset that is used for weight adaptation. Industry approaches have implemented quantization techniques that use only a small amount of unlabelled data, in systems such as TensorRT [1]. In this instance, they can successfully quantize a network to 8-bits with no loss of accuracy (sometimes even with improved accuracy) from 1000s of sampled images [1]. Other examples include the Google TPU, and Project Brainwave ([15, 9]), all of which quantize neural networks to 8-bits for fast inference. Another work that shows that 8-bit quantization does not affect efficiency significantly is [26], where they show that this is true even when quantizing both activation and weights.

In this paper, we show that quantization can be done effectively to 4-bits for both weights and activations, without the need of retraining labelled data, with further potential improvements when using adaptation with unlabelled data.

3. Method

For a given neural network inference $f(x)$, the prediction of $f(Mx)$ should be identical (considering that biases are scaled accordingly), for $M \in \mathbb{R}$. *DSConv* is built on the intuition that this property holds for some nonlinear transforms of x, as long as the relative distribution of the weights and activation values remains the same. We believe one such transform to be quantization, *i.e.* we can scale and bias quantized weights and activations in a way that is friendly for low precision representation and still maintain the same neural network accuracy, as long as distribution over the weights and activations remains unchanged.

Adopting this strategy to the entire 4D tensor would yield a very high cropping error, since a single scaling factor M would not be able to single-handedly capture the entire tensor distribution. In this paper we adopt the more general strategy of using a tensor of scaling factors, whose size is adjusted to capture the range of values with higher fidelity. Every tensor of floating point values is divided into two components: one tensor with the same size of the original, composed of low-bit integer values, and another one with a fraction of the size, composed of floating point scaling factors. Each scaling factor is responsible for the scaling of a subgroup of B integer values along its tensor depth dimension, where B is the block size hyperparameter.

The steps taken by *DSConv* are as follows: **(I)** From a pre-trained network, divide the weight tensor depth-wise into blocks of variable length B and quantize each block; **(II)** Use the block floating point (BFP) format to quantize the activations, where the block is the same size as the weight tensor; **(III)** Multiply the integer values of the activations and the weight tensor to maximize inference speed; **(IV)** Multiply the final values by their respective scales to shift the distribution of the individual blocks to the correct range.

3.1. Weight Quantization

We propose a method for quantizing weights that shares one floating-point value for each block of size B, along the depth dimension of each weight tensor filter. An example for the resulting sizes for each filter can be seen in Figure 1.

Given a weight tensor of size (C_o, C_i, K_h, K_w) and a block size hyperparameter B, we first divide the tensor into two components: the *Variable Quantized Kernel* (VQK), which is composed of low-bit values, and is of the same size as the original tensor; and the *Kernel Distribution Shift* (KDS), composed of single precision numbers ξ, and of size $(C_o, \lceil \frac{C_i}{B} \rceil, K_h, K_w)$, where $\lceil x \rceil$ is the ceiling operation.

The B hyperparameter can seamlessly be modified to accommodate for trade-off between floating point arithmetic and fixed point arithmetic, with $B = 1$ for pure floating point to $B \geq C_i$ for maximum fixed point.

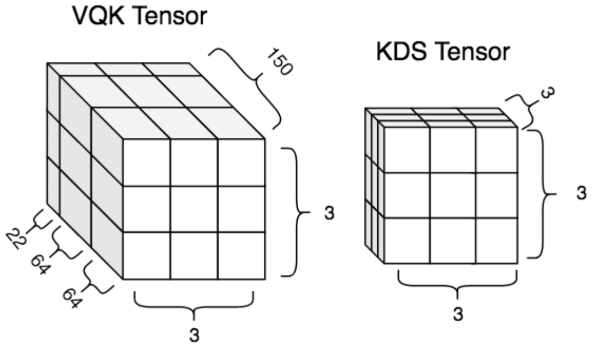

Figure 1. Size of VQK and KDS for each weight filter, for the case of $B = 64$. This reduces the number of FP MACs from 1350 to 27.

The VQK then holds integer values in 2s complement such that for a specific number of bits b chosen, the weights are in the interval:

$$w_q \in \mathbb{Z}, b \in \mathbb{N} \mid -2^{b-1} \leq w_q \leq 2^{b-1} - 1, \qquad (1)$$

This allows all the operations to be performed using 2s complement arithmetic, as explained in §3.3.

By simply changing the normal convolution to *DSConv*, the memory saved per tensor weight is:

$$p = \frac{b}{32} + \frac{\lceil \frac{C_i}{B} \rceil}{C_i} \qquad (2)$$

Equation 2 shows that, for large enough values of B and C_i, the memory saved is approximately the number of bits b divided by 32. For illustration purposes, Table 1 shows the numerical results for realistic values of C_i, B, b and p for some layers of the GoogLeNet [34] architecture. As it can be seen, significant memory saving can be achieved by only quantizing, with no additional method such as Huffman Coding [21].

	Channel (C_i)	Block (B)	Bit (b)	Saving (p)
Inception (4a)	128	64	4	14.1%
Inception (4a)	128	128	4	13.3%
Inception (4a)	128	32	3	12.5%
Inception (4c)	256	128	3	10.2%

Table 1. Memory savings by quantizing only.

Given a known pre-trained model, the weights of each block are stretched and rounded to fit in the interval in Equation 1, and they are stored in the VQK. Next, we explored two possible methods to calculate the KDS values: (i) minimizing the KL-Divergence, which seeks to find the minimum loss of information between the distribution of the original weights and the kernel distribution shifter and emphasizes the idea that the resulting VQK, after being shifted, should have a similar distribution to the original weights; or (ii) minimizing the L2 norm (Euclidean distance), which can have the interpretation that parameters should be the closest to the optimum value of the original network.

To minimise the KL-Divergence we first take the softmax values of both the shifted VQK and the original distributions:

$$T_j = \frac{e^{w_j}}{\sum_i e^{w_i}}, \; I_j = \frac{e^{\hat{\xi} \cdot w_{q_j}}}{\sum_i e^{\hat{\xi} \cdot w_{q_i}}} \qquad (3)$$

We then use gradient descent to minimize the following for each slice:

$$\xi = \min_{\hat{\xi}} \sum_j T_j \log\left(\frac{T_j}{I_j}\right), \quad \forall \, (1, B, 1, 1) \text{ slices} \qquad (4)$$

where ξ is the KDS value for that block.

The other method minimises the following L2 norm for each slice:

$$\xi = \min_{\hat{\xi}} \sum_{i=0}^{B-1} (w_{q_i} \, \hat{\xi} - w_i)^2 \qquad (5)$$

which has the closed form solution:

$$\therefore \;\; \xi = \frac{\sum_{i=0}^{B-1} w_i w_{q_i}}{\sum_{i=0}^{B-1} w_{q_i}^2}, \qquad \forall \, (1, B, 1, 1) \text{ slices} \qquad (6)$$

In practice, both strategies produced approximately equal values. We performed all the experiments using the L2 norm approach, since it has a closed form solution.

Algorithm 1 summarizes the process of initializing both the VQK and the KDS given a pre-trained network model.

3.2. Activation Quantization

Our approach aims to achieve good performance in the absence of any training data. This means that we have **no prior knowledge** of what values or distribution the activation maps will have. Therefore, this quantization cannot be data-driven. Instead, we used an approach inspired by the block floating point (BFP) method of [35, 33, 14, 9, 15].

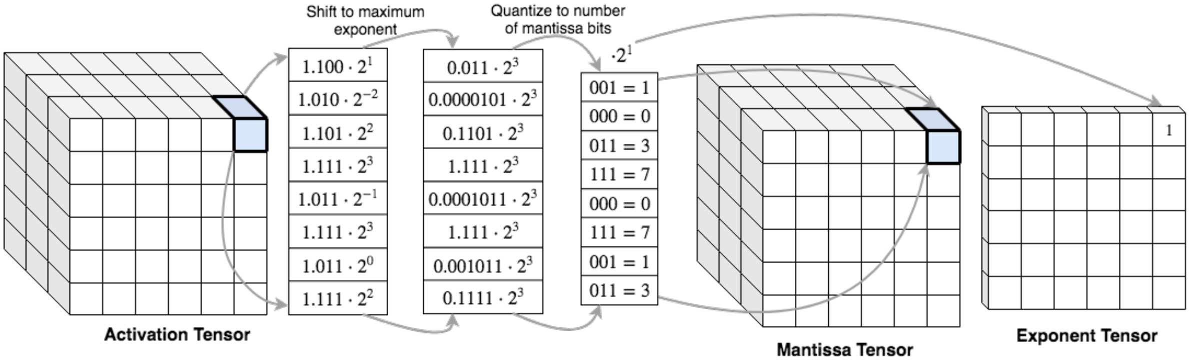

Figure 2. Example of quantizing activation. This is the specific case where the mantissa bit was set to 3 and the block hyperparameter was set to 8. Note that ½ LSB rounding performed when cropping. Note that this is performed after the ReLU layer, which means that all values are unsigned positive.

Algorithm 1 Weight Initialization

```
    Input bit-length $b$, pre-trained weights $\mathbf{w}$, block size $B$
procedure QUANTIZE
    $m \leftarrow 2^{b-1} - 1$
    for all Block $B$ do:
        $w_m \leftarrow \arg\max_{\mathbf{w}}(|w|)$
        $s \leftarrow m / w_m$
        for all $w_i$ in $B$ do:
            $w_q \leftarrow \text{round}(w_i \cdot s)$
        $\xi = \sum_{i=0}^{B-1} w_i w_{q_i} / \sum_{i=0}^{B-1} w_{q_i}^2$
    return $\xi$, $\mathbf{w_q}$
```

Figure 2 shows our activation quantization approach. For a given mantissa width, the *activation tensor* is divided into blocks, and, for each block, we find the maximum exponent. The mantissa values of all the other activations in the block is shifted such that they match the maximum exponent, which is then cropped (using ½ LSB rounding) to match the number of specified bits. This results in two tensors: a *mantissa tensor*, which has the same shape as the original tensor, but populated with b bits; and an *exponent tensor*, which has size $(C_o, \lceil \frac{C_i}{B} \rceil, H, W)$.

We call this a BFP approach because we are essentially "sharing" the exponent for each block of size B. This allows for a control over how coarse the quantization is, and how much cropping error we are willing to accept to get the lowest bit-length for the mantissa tensor.

This approach has the added benefit of allowing low-bit integer operations between the weights and activations, as we show in §3.3. Therefore, the trade-off between efficiency and speed of computation is as follows: the higher the value of B, the bigger the cropping error will be, but the exponent tensor and the KDS will be shallower. This is a different trade-off to the number of bits b, which adds more computational complexity and memory to the mantissa tensor. The values of b and B are then inversely proportional to each other and counter-balance each other's positive and negative effects. The goal then becomes to get the most accuracy with the lowest number of mantissa bits b and largest value for the block size B.

3.3. Inference

During inference, the hardware can take advantage of the fact that the VQK and the mantissa tensors are low-bit integer values, which allows it to save time performing integer operations rather than floating point operations. The data path is illustrated in Figure 3.

First, each of the blocks of the VQK and the mantissa tensor are dot producted, resulting in one value each. All of these operations can be conducted in low-bit fixed point arithmetic, which saves significant processing time. At the end of the block multiplications, the result is a tensor of the same size as both the exponent tensor and the KDS.

The exponent tensor is merged with the KDS tensor by adding its value to the exponent of the KDS tensor values. This results in a tensor of the same size of floating point numbers. Finally, this tensor multiplies the result of the product of the VQK and the KDS, and yields a single floating point number as the output activation.

Notice that the inference is as highly parallelizable as a standard convolution, but instead of performing most of the multiplications using floating point arithmetic, the majority can be substituted by integer multiplications, saving energy, bandwidth and computation time.

This also means that, for each weight and activation multiplication, the number of blocks is proportional to the number of total floating point MAC operations, and the size of the tensor itself gives the number of INT MAC operations.

Batch Normalization Folding. Similar to [23], we perform "folding" of the Batch Normalization (BN) [22] parameters in models that have them. Since batch normalization has been shown to improve training (see [22]), we keep it during the training phase and only fold it for inference.

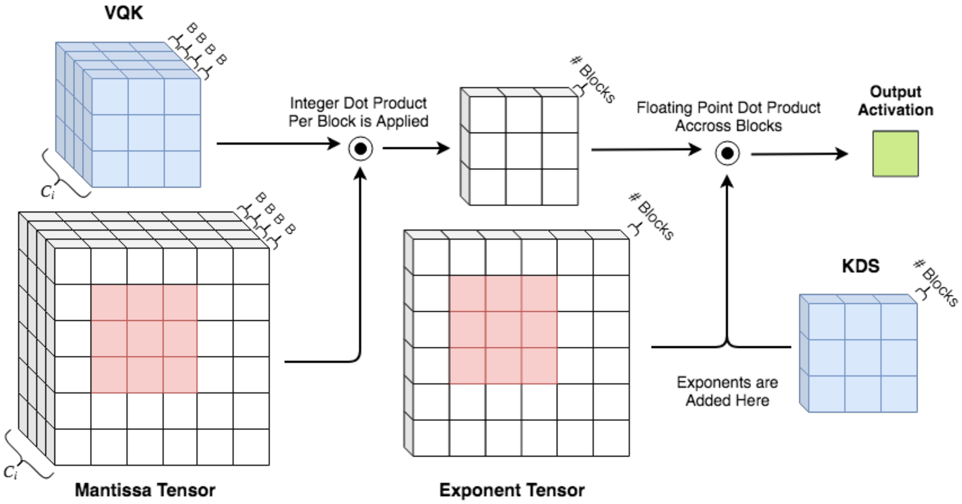

Figure 3. Example of convolution being performed, with VQK tensor (in blue) multiplying one section of the Mantissa Tensor (in red). Each block of the VQK performs a dot product with each block of the Mantissa tensor. The result is a tensor with depth equal to the number of blocks depth-wise. The Exponent Tensor performs addition of the exponent value of the KDS, and the result is multiplied by the result of the dot product of the VQK and the Mantissa tensor, and that becomes the final output activation. This is performed for every filter.

When folding the BN parameters, we do so with the KDS, since they are unique per channel and use FP32 values. We perform the folding using the equations:

$$\xi_{fold} = \frac{\xi\gamma}{\sqrt{\sigma_B^2 + \epsilon}} \tag{7}$$

$$b_{fold} = \beta - \frac{\gamma\mu_b}{\sqrt{\sigma_B^2 + \epsilon}} \tag{8}$$

where the parameters γ, σ, ϵ, β and μ are as defined in [22], ξ is the KDS tensor and b_{fold} is the resulting bias of the *DSConv*.

3.4. Distillation for Unlabelled Data Adaptation

It is often the case that unlabelled data may be available, as shown by the vast array of unsupervised learning methods available. For this specific scenario, we adopt a strategy similar to [19]. We use the distillation loss without labelled data to try to regress the FP32 model to the quantized one, by using the FP32 logits as the target, and minimizing the loss for regression.

We create a “shadow” model which holds single-precision numbers. Before each inference, this model is quantized to the VQK and KDS, inference is performed and the gradients are calculated. During the update phase, the gradients are accumulated as single-precision numbers, and the method is performed until convergence.

Quantizing the activation maps after each inference would cause the gradients to be zero everywhere. To avoid this problem, we use the Straight-Through Estimator (STE) [5, 37], to calculate the backwards gradient. Particularly, we use the ReLU STE since it was shown in [37] that it gives a better accuracy than using the Identity STE for deeper networks. The gradient is then also accumulated in a “shadow” FP32 model, which is quantized after each batch iteration.

We use the ADAM Optimizer [24], with initial learning rate 10^{-5} and after the loss plateaus, this rate is changed to 10^{-6}. All other hyperparameters and data augmentation details follow their respective original papers.

We use 960 images (30 batches of 32) from the validation dataset for the distillation, and we test the accuracy using the rest of the images (49,040 images in total).

4. Experiments and Results

We tested our method on various neural network architectures: ResNet [18], AlexNet [27], GoogleNet [34], and DenseNet [20]. We benchmarked our results on the ImageNet dataset [13] (more specifically ILSVRC2012), which has 1.2M images in the training set and 50k images in the validation set. The results reported use images drawn from the validation set. We tested our algorithm for all the tasks indicated in the introduction. This section continues as follows: §4.1 finds the theoretical computational saving for DSConv; §4.2 shows the results without training or adaptation; §4.3 shows the accuracies when the model is adapted with unlabelled data; and §4.4, for comparison with previous methods, shows the results for the retraining performed in DSConv using labelled data.

4.1. Theoretical Computational Load on Block Size

Computational load is traditionally reported as a function of number of MAC operations needed in order to complete the algorithm. We note two caveats: integer MACs are far less complex than FP MACs and, when supported by a hardware implementation, can be run orders of magnitudes faster than FP operations [26]; our method also relies on the ability to create the mantissa tensor and the exponent tensor dynamically (the VQK and the KDS are created statically, so they are not considered here). This requires `MAX`, `SHIFT` and `MASK` operations. These can be implemented efficiently in custom hardware with few clock cycles. Therefore we will focus on the comparison between number of INT vs FP operations to assess the advantage of using this method.

In order for our method to be faster than normal convolution, the time spent to perform the INT operations must be less than the time spent on the FP32 operations. This difference is a function of the block size and on the channel parameter. Equation 9 shows the relation between the time for an INT operation and the time for an FP operation:

$$T_{int} \leq T_{FP} \frac{C_i - \lceil \frac{C_i}{B} \rceil}{C_i(1+\eta)} \tag{9}$$

The values T_{int} and T_{FP} capture the amount of time needed to perform an INT and an FP operation, respectively. The parameter η is an "ideality" parameter that represents the overall overhead in the `MAX`, `SHIFT` and `MASK` operations to perform *DSConv* in comparison to the normal convolution operator.

Also notice that, if C_i is divisible by B (which is often the case), then Equation 9 becomes independent of the channel size and simplifies to:

$$T_{int} \leq T_{FP} \frac{1-\frac{1}{B}}{1+\eta}, \text{ if } B \mid C_i \tag{10}$$

Table 2 shows the ratio $1-\frac{1}{B}$ for the most common block sizes experimented, when $\eta = 0$. As can be seen, if the time to compute an INT value is less than 0.75 of the time to compute a floating point operation, then all block sizes bigger than 4 will be faster than the normal convolution. This is often likely to be the case. For example, in modern CPUs and in some GPUs, 8-bit operations can be up to 10x faster than FP32 operations [26], and lower bit operations can potentially be even faster in custom hardware such as FPGAs. In custom software, operations in less than 8-bit are also often faster.

Block	4	8	16	32	64	128
Ratio	0.750	0.875	0.938	0.969	0.984	0.992

Table 2. Relationship of Block Size and speed ratio needed.

The block size B imposes a limit on how much faster *DSConv* can be over traditional convolution operators. Naturally, *DSConv* can be up to $min(C_i, B)$ times faster than the traditional convolution, since it has $min(C_i, B)$ times less floating point operations than a normal convolution. For example, for block sizes of 128 to 256 and channel sizes of more than 256, *DSConv* can be up to two orders of magnitude faster than a normal convolution.

4.2. Accuracy Before Retraining or Adaptation

Our method is designed to produce accurate results even when training data is not available, by quantizing from a pre-trained network.

		Accuracy (% @ Top1 and Top5)					
Block	**Bit (W/A)**	**ResNet50**		**ResNet34**		**ResNet18**	
-	32 / 32	76.1	92.9	73.3	91.4	69.8	89.0
256	8 / 32	76.1	92.9	73.3	91.4	69.7	89.0
128	6 / 32	75.9	92.8	73.2	91.4	69.5	89.0
16	4 / 32	75.1	92.3	72.6	91.0	67.7	87.8
4	2 / 32	65.1	86.2	66.8	87.6	59.1	81.7
128	8 / 8	76.1	92.9	73.3	91.4	69.7	89.1
64	6 / 6	75.9	92.8	73.2	91.4	69.6	89.0
64	5 / 5	75.4	92.6	72.7	91.0	68.9	88.5
16	4 / 4	74.8	92.1	72.3	90.8	67.3	87.7

		Accuracy (% @ Top1 and Top5)					
Block	**Bit (W/A)**	**GoogLeNet**		**VGG19**		**Dense121**	
-	32 / 32	67.6	88.3	72.4	90.9	74.4	92.0
256	8 / 32	67.6	88.3	72.3	90.9	74.4	92.0
128	6 / 32	67.1	88.0	72.4	90.9	74.2	91.8
16	4 / 32	63.3	85.4	72.1	90.7	72.9	91.2
4	2 / 32	27.6	51.1	69.4	89.0	62.7	84.4
128	8 / 8	67.6	88.3	72.3	90.9	74.4	92.0
128	6 / 6	67.1	88.0	72.4	90.8	74.2	91.8
64	5 / 5	65.5	86.8	72.3	90.8	73.8	91.6
16	4 / 8	63.3	85.4	72.1	90.7	72.9	91.2

Table 3. Accuracy of Fast Inference and Compression without data as a function of Bit width (in Weights and Activation) and Block Size.

The second and fifth rows of Table 3 show that for both the compression and fast inference problems, no loss of accuracy can be achieved with 8-bit networks even with very high block sizes, as already demonstrated by previous papers and real-life applications [26, 16]. The results also shows that compression down to 4-bits (which in convolutions with channel size input of 256 would yield a **5x** compression rate) results in an accuracy drop of only 1% to 2% depending on the architecture. It can also be seen that very low-bit quantizations become noticeably unstable, varying greatly with architecture. At the extreme, using 2-bits, losses vary by as much as -40% for GoogLeNet and only -11% for ResNet50.

The last four rows show the results for the fast inference problem. Also as known in previous research papers [26, 16], models of 8/8 bits lose only around 0.1% accuracy. For models of 5/5 and 4/4, we get a drop of 1% to 3% in accuracy. To our knowledge, this is the smallest bitwidth for fast inference that has been reported when models are neither retrained nor adapted.

The variance with respect to architecture suggests that quantization for 5 or less bits is unstable. However, even for fast-inference with 8-bit accuracy, it can achieve stable and satisfactory results within 1% of the full precision model.

Accuracy with respect to Block Size Table 4 shows the accuracy with respect to block size. The table shows the results of quantizing the weights only, where the number in parenthesis represents the bit-width of the weights. Naturally, this represents a trade-off between memory and computational load against precision of the network. The largest discrepancy in accuracy can be seen in models that use 3 or 2 bit weights. For example, the GoogLeNet model with 3-bits improves its Top1 accuracy from 5.7% to 56.8% when changing from a block-size of 256 to 8.

Block	**256**	**128**	**64**	**32**	**16**	**8**	**4**
ResNet50 (4)	73.0	73.5	73.8	74.7	75.1	75.4	75.6
ResNet50 (3)	44.6	51.9	59.6	67.4	69.6	73.6	74.7
ResNet34 (4)	70.8	70.8	71.5	71.9	72.6	72.8	72.9
ResNet34 (3)	59.5	60.4	63.6	66.8	69.2	70.6	71.6
GoogLeNet (4)	52.5	57.0	59.1	61.7	63.3	65.6	66.5
GoogLeNet (3)	5.7	22.4	37.6	40.3	49.2	56.8	62.5
VGG19 (3)	67.6	68.6	69.5	70.4	71.1	71.6	71.8
VGG19 (2)	11.3	21.8	38.1	55.5	63.1	67.5	69.4

Table 4. Accuracy with respect to block size for the compression case with no data available.

When using 4-bit quantization schemes, a decrease in the block size achieves accuracy levels that are within 1% to 2% of the full precision network. This is the case for example for most networks with block sizes of 16 to 32.

4.3. Accuracy Adapted using Unlabelled Data

The results when adapting our network with extra unlabelled data are reported in Table 5. For each Block-Bit configuration, two results are reported: on the top we show the result before adaptation and the bottom (in bold) the result after adaptation using unlabelled data. This strategy increases inference accuracy using 4-bits only for the weights and for the activations to within 2% of the FP32 precision of the network, even for the extreme cases of using 128 as the block size.

For 3-bits, even though we recover up to 30% accuracy, there is still a considerable gap between the low-bit accuracies and the full precision ones. For ResNet50 this gap is of 6% whereas for GoogLeNet it can reach 10%.

Table 6 shows results of recent papers (introduced in the literature review) that use calibration. This is similar in spirit to our adaptation stage, in that both approaches use only unlabelled data. The notable exception is that we use distillation to convert a full-precision model to a low-precision model, whereas the other approaches generally calibrate just the optimal clipping strategy. It can be seen that, even using a big block size of 64, we achieve better performance. To our knowledge, this is the best result achieved for fast inference using only adaptation data.

		Accuracy (% @ Top1 and Top5)							
Block	**Bit (W/A)**	**ResNet50**		**ResNet34**		**ResNet18**		**GoogLeNet**	
32	4 / 4	74.1	91.8	71.3	90.2	66.4	87.1	61.2	81.9
		74.8	**92.1**	**71.8**	**90.6**	**68.3**	**88.1**	**66.1**	**87.2**
64	4 / 4	73.0	89.8	70.9	88.4	66.1	85.1	58.4	79.3
		74.8	**92.1**	**71.8**	**90.6**	**68.4**	**88.1**	**65.2**	**86.8**
128	4 / 4	72.6	89.6	70.2	87.9	65.8	84.8	55.8	77.3
		74.2	**92.0**	**71.3**	**90.5**	**67.5**	**87.8**	**64.7**	**86.3**
32	3 / 3	63.3	85.0	63.2	85.0	55.3	78.4	34.5	60.5
		72.6	**91.1**	**69.6**	**89.4**	**66.8**	**87.5**	**60.0**	**83.3**
64	3 / 3	54.4	77.9	58.1	81.3	30.1	51.6	29.3	53.9
		71.5	**90.4**	**69.1**	**89.3**	**65.8**	**87.0**	**56.7**	**81.0**

Table 5. Results of adaptation for a variety of architectures for the case where an adaptation dataset is provided.

			VGG16	AlexNet	ResNet18	ResNet50
	W	A	Top1	Top1	Top1	Top1
Naive [4]	4	8	29.0%	1.8%	0.8%	0.4%
CW [4, 26]	4	8	70.2%	52.9%	59.3%	72.4%
K+B [4]	4	8	70.0%	54.7%	67.0%	74.2%
OCS+MSE [39]	5	8	-	-	-	73.4%
Ours NA (16)	4	8	**71.3%**	**55.9%**	**67.6%**	**75.1%**
Ours NA (32)	4	8	**71.2%**	**55.4%**	**66.7%**	**74.7%**
Naive [4]	8	4	53.9%	41.6%	53.2%	52.7%
KLD [4, 1]	8	4	67.0%	49.6%	65.1%	70.8%
ACIQ [4]	8	4	70.5%	55.2%	68.9%	74.8%
Ours NA (16)	8	4	**71.5%**	**56.4%**	**69.6%**	**75.7%**
Ours NA (32)	8	4	**71.5%**	**56.4%**	**69.6%**	**75.6%**
Naive [4]	4	4	23.7%	1.8%	0.6%	0.4%
ACIQ [4]	4	4	68.9%	53.0%	65.3%	72.6%
OMSE+O [8]	4	4	-	54.5%	67.4%	72.6%
Our (64)	4	4	**71.1%**	**55.8%**	**68.4%**	**74.8%**

Table 6. Adaptation of our method vs previous papers. The Naive method refers to simple clipping. CW is Channel-Wise quantization adopted in [36, 26]. K+B is the K-Means + Bias method of [4]. KLD is the KL-Divergence method first proposed in [1]. OMSE+O is the OMSE + offset method of [8]. "Ours NA" refer to our method with No Adaptation

4.4. Accuracy After Labelled Data Retraining

We also compared *DSConv* with previous methods that retrain/finetune with labelled data (the vast majority in the literature). Training happens similarly to adaptation, but now we use labelled data and use cross-entropy loss in the classification error instead of using logits.

Table 7 shows the results for ImageNet on a variety of architectures. As many previous papers report different initial FP accuracy for the same architecture, we have also in-

ResNet 18

	W	A	Top1	Top5
FP	32	32	69.6	89.2
FP_{LQ} [38]	32	32	70.3	89.5
BWN [32]	1	32	60.8	83.0
TWN [28]	2	32	61.8	84.2
TWN [28]	2	32	65.3	86.2
TTQ [41]	2	32	66.6	87.2
LQ [38]	2	32	68.0	88.0
Ours (32)	2	32	**68.7**	**86.7**
LQ [38]	3	32	69.3	88.8
Ours (32)	3	32	**69.7**	**87.5**
LQ [38]	4	32	70.0	89.1
Ours (32)	4	32	**70.0**	**87.6**
XNOR [32]	1	1	51.2	73.2
DoReFa [40]	1	2	53.4	-
DoReFa [40]	1	4	59.2	-
Ours (32)	1	4	**65.2**	**86.2**
HWGQ [6]	1	2	59.6	82.2
ABC [29]	3	3	61.0	83.2
ABC [29]	5	5	65.0	85.9
Ours (128)	5	5	**70.0**	**89.3**
LQ [38]	1	2	62.6	84.3
LQ[38]	2	2	64.9	68.2
LQ [38]	3	3	68.2	87.9
Ours (16)	3	3	**69.2**	**88.9**
LQ [38]	4	4	69.3	88.8
Ours (64)	4	4	**69.8**	**89.2**

DenseNet121

	W	A	Top1	Top5
FP	32	32	75.0	92.3
DoReFa [40]	2	2	67.7	88.4
FP_{LQ} [38]	32	32	75.3	92.5
LQ [38]	2	2	69.6	89.1
FP_{Ours}	32	32	74.4	92.2
Ours (32)	2	32	**74.0**	**91.8**
Ours (16)	2	2	**72.1**	**90.6**

AlexNet

	W	A	Top1	Top5
FP	32	32	57.1	80.2
TWN [28]	2	32	54.5	76.8
FP_{LQ} [38]	32	32	61.8	83.5
LQ[38]	2	32	60.5	82.7
FP_{Ours}	32	32	**56.6**	**79.1**
Ours (32)	2	32	**55.0**	**78.1**

ResNet34

	W	A	Top1	Top5
FP	32	32	73.3	91.3
FP_{LQ} [38]	32	32	73.8	91.4
Ours (32)	3	32	**73.4**	**90.1**
Ours (32)	4	32	**73.6**	**90.1**
HWGQ [6]	1	2	64.3	85.7
Ours (64)	1	4	**68.2**	**86.8**
ABC [29]	3	3	66.7	87.4
ABC [29]	5	5	68.4	88.2
Ours(16)	4	4	**73.0**	**89.7**
LQ[38]	1	2	66.6	86.9
LQ [38]	2	2	67.8	89.1
LQ [38]	3	3	71.9	90.2
Ours (16)	3	3	**72.7**	**89.6**

ResNet50

	W	A	Top1	Top5
FP	32	32	76.0	93.0
FP_{LQ} [38]	32	32	76.4	93.2
LQ[38]	2	32	75.1	92.3
Ours (32)	2	32	**75.2**	**92.6**
LQ[38]	4	32	76.4	93.1
Ours(128)	4	32	**76.4**	**93.0**
HWGQ [6]	1	2	64.6	85.9
ABC [29]	5	5	70.1	89.7
LQ[38]	1	2	68.7	88.4
LQ[38]	2	2	71.5	90.3
Ours(32)	2	2	**72.5**	**91.2**
LQ[38]	3	3	74.2	91.6
Ours (32)	3	3	**75.2**	**92.4**
LQ[38]	4	4	75.1	92.4
Ours (64)	4	4	**76.2**	**92.9**
Ours (128)	4	4	**76.1**	**92.8**

GoogLeNet

	W	A	Top1	Top5
FP_{HWGQ} [6]	32	32	71.4	90.5
HWGQ [6]	1	2	63.0	84.9
FP_{LQ} [38]	32	32	72.9	91.3
LQ [38]	1	2	65.6	86.4
LQ [38]	2	2	68.2	88.1
FP_{Ours}	32	32	67.6	86.3
Ours (32)	4	4	**66.3**	**85.5**
Ours (64)	4	4	**65.7**	**85.1**

Table 7. Results of retraining for a variety of architectures. Table derived from [38]

cluded the initial FP of the single precision results to make an evaluation that takes into account the “upper limit” of the architecture itself.

From the results, it can be seen that our method can beat the state-of-the-art for a variety of cases, as long as the Block Size is adjusted properly to give more emphasis on accuracy rather than speed.

DSConv can beat the state-of-the-art when using bit sizes that are either 4 or 5. In these cases (such as ResNet18 using 5/5, ResNet50 using 4/4 and GoogLeNet using 4/4), we also use a large Block Size, with slightly better than the FP efficiency in ResNet18 when using $B = 128$ and bit sizes of 5/5, and $B = 64$ for bit sizes of 4/4.

In order to get state-of-the-art results for 3-bits or less, a lower block size is needed. This is shown for DenseNet121 results, which uses bit-width of 2 and Block Size of 16 to get 72.1% accuracy. Extremely low-bit weights and activations do not work very well because the assumption that lower information loss in quantization corresponds to higher accuracy starts to break down. This is supported by the fact that the state-of-the-art approaches for 1 and 2 bit weights are trained from scratch, which suggests that for these cases, quantizing from a pre-trained network is not ideal.

We also show good results for the compression case. ResNet50 with 4 bit and $B = 128$ illustrates that no loss of accuracy is observed, and even using only 2 bits, with accuracy staying within 1% using $B = 32$.

5. Conclusion

We presented *DSConv*, which proposes an alternative convolution operator that can achieve state-of-the-art results whilst quantizing models to up to 4-bits in weight and activation without retraining or adaptation.

We showed that our method can achieve state-of-the-art results without retraining in less than 8 bit settings, which makes it possible for fast inference and less power consumption for rapid deployment in custom hardware. By having the advantage of being tunable by the block size hyperparameter and not needing any training data in order to run, we propose that this method is very suitable for acceleration of convolutional neural networks of any kind.

When using unlabelled data and distillation from the FP32 model, we can achieve less than 1% loss using 4-bit for both weights and activations. Also, as in previous methods, we demonstrate that the assumption that lower information loss in weight quantization correspond to higher inference accuracy breaks down when quantizing to extremely low bits (1, 2 or 3 bits). In these cases, retraining seems inevitable since they are quintessentially different than higher accuracy models.

6. Acknowledgements

This research was partially supported by Intel (previously Omnitek), and we thank our colleagues from the Programmable Solutions Group who greatly assisted in this work. Victor Adrian Prisacariu would like to thank the European Commission Project Multiple-actOrs Virtual Empathic CARegiver for the Elder (MoveCare).

References

[1] NVidia TensorRT. http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf. Accessed: 2019-09-06.

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

[3] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2654–2662. Curran Associates, Inc., 2014.

[4] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. Post-training 4-bit quantization of convolution networks for rapid-deployment, 2018.

[5] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[6] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017.

[7] 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.

[8] Yoni Choukroun, Eli Kravchik, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference, 2019.

[9] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, Mar 2018.

[10] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[11] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*, 2015.

[12] William Dally. High-performance hardware for machine learning, 2015.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. 2009.

[14] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point, 2018.

[15] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[16] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE transactions on neural networks and learning systems*, (99):1–6, 2018.

[17] 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.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

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

[20] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017.

[21] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

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

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

[24] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[25] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in neural information processing systems*, pages 1742–1752, 2017.

[26] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

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

[28] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[29] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 345–353, 2017.

[30] Asit Mishra, Eriko Nurvitadhi, Jeffrey J Cook, and Debbie Marr. Wrpn: wide reduced-precision networks. *arXiv preprint arXiv:1709.01134*, 2017.

[31] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[32] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *Lecture Notes in Computer Science*, page 525–542, 2016.

[33] Zhourui Song, Zhenyu Liu, and Dongsheng Wang. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

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

[35] James H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, Inc., New York, NY, USA, 1994.

[36] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks, 2018.

[37] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley J. Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets. In *International Conference on Learning Representations*, 2019.

[38] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. *Lecture Notes in Computer Science*, page 373–390, 2018.

[39] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting, 2019.

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

[41] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization, 2016.