INTEGER QUANTIZATION FOR DEEP LEARNING INFERENCE: PRINCIPLES AND EMPIRICAL EVALUATION

Hao Wu[1] Patrick Judd[1] Xiaojie Zhang[1] Mikhail Isaev[2*] Paulius Micikevicius[1]

[1]NVIDIA
{skyw, pjudd, viczhang, pauliusm}@nvidia.com

[2]Georgia Institute of Technology
michael.v.isaev@gatech.edu

ABSTRACT

Quantization techniques can reduce the size of Deep Neural Networks and improve inference latency and throughput by taking advantage of high throughput integer instructions. In this paper we review the mathematical aspects of quantization parameters and evaluate their choices on a wide range of neural network models for different application domains, including vision, speech, and language. We focus on quantization techniques that are amenable to acceleration by processors with high-throughput integer math pipelines. We also present a workflow for 8-bit quantization that is able to maintain accuracy within 1% of the floating-point baseline on all networks studied, including models that are more difficult to quantize, such as MobileNets and BERT-large.

1 Introduction

While 32-bit single-precision floating-point was the dominant numerical format for Deep Learning (DL) applications, more recently a variety of alternative formats have been proposed to increase the computational performance of deep learning applications. It is becoming commonplace to train neural networks in 16-bit floating-point formats, either IEEE fp16 [35] or bfloat16 [57], supported by most DL accelerators. Once trained, neural networks can be deployed for inference using even lower-precision formats, including floating-point, fixed-point, and integer. Low-precision formats offer several performance benefits. First, many processors provide higher throughput math pipelines the low-bit formats, which can speed up math-intensive operations, such as convolutions and matrix multiplications. Second, smaller word sizes reduce memory bandwidth pressure, improving performance for bandwidth-limited computations. Third, smaller word sizes lead to lower memory size requirements, which can improve cache utilization as well as other aspects of memory-system operation.

In this paper we focus on integer quantization for neural network inference, where trained networks are modified to use integer weights and activations so that integer math pipelines can be used for many operations. Table 1 lists the relative tensor operation throughputs of various data types on the NVIDIA Turing Graphics Processing Unit (GPU) architecture [40]. Math-intensive tensor operations executed on 8-bit integer types can see up to a 16x speed-up compared to the same operations in fp32. Memory-limited operations could see up to a 4x speed-up compared to the fp32 version, due to the smaller word size. Other processors, such as TPUv1 [23], Intel CPUs with VNNI

Input Data type	Accumulation Data type	Math Throughput	Bandwidth Reduction
FP32	FP32	1x	1x
FP16	FP16	8x	2x
INT8	INT32	16x	4x
INT4	INT32	32x	8x
INT1	INT32	128x	32x

Table 1: Benefits of lower precision data types for tensor operations on the NVIDIA Turing GPU architecture

*Work done during an internship at NVIDIA

instructions [28], and a number of emerging accelerator designs also provide significant acceleration for int8 operations. The process of neural network quantization can be automated by software tools [36, 61] or controlled manually. In either case, care must be taken to minimize any impact quantization has on the model accuracy.

In this paper we review the mathematical fundamentals underlying various integer quantization choices (Section 3) as well as techniques for recovering accuracy lost due to quantization (Section 5). Section 6 combines this information into a recommended workflow. In Section 4 and the Appendices we present empirical evaluation of various quantization choices on a wide range of network models from different application domains - image processing, language modeling, language translation, and speech recognition. These models include the major network topologies - convolutional networks, recurrent networks, as well as attention-based networks. With the presented workflow for int8 quantization we are able to maintain model accuracy within 1% of each baseline floating-point network, even for the networks that are known to be challenging to quantize, such as MobileNets and BERT-large.

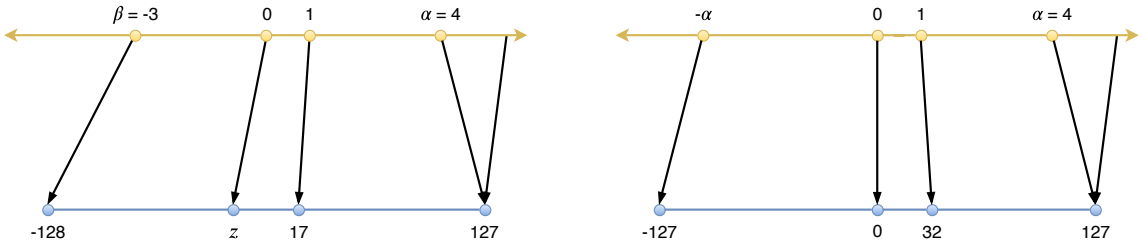
2 Related Work

Vanhoucke et al. [52] showed that earlier neural networks could be quantized after training to use int8 instructions on Intel CPUs while maintaining the accuracy of the floating-point model. More recently it has been shown that some modern networks require training to maintain accuracy when quantized for int8. Jacob et al. [20] described models optimized for inference where all inference operations were performed with integer data types. Here batch normalization layers were folded into the preceding convolution layer before quantization, reducing the number of layers that needed to be executed during inference. Krishnamoorthi [26] evaluated various quantization methods and bit-widths on a variety of Convolutional Neural Networks (CNNs). He showed that even with per-channel quantization, networks like MobileNet do not reach baseline accuracy with int8 Post Training Quantization (PTQ) and require Quantization Aware Training (QAT). McKinstry et al. [33] demonstrated that many ImageNet CNNs can be finetuned for just one epoch after quantizing to int8 and reach baseline accuracy. They emphasized the importance of using an annealing learning rate schedule and a very small final learning rate. They also set the quantization range based on a percentile of activations sampled from the training set. Instead of using fixed ranges, Choi et al. [6] proposed PACT which learns the activation ranges during training.

Much of the earlier research in this area focused on very low bit quantization [7, 13, 59], all the way down to ternary (2-bit) [60, 34] and binary weights [8] and activations [45, 18]. These works showed that for lower bit-widths, training with quantization was required to achieve high accuracy, though accuracy was still lower than the floating-point network on harder tasks such as ImageNet image classification [47]. They also demonstrated the importance of techniques such as using higher precision for weight updates and the Straight-through Estimator (STE) for gradient backpropagation [3]. Also, in many cases the first and last layer were not quantized, or quantized with a higher bit-width, as they are more sensitive to quantization [59, 45, 18]. Multi-bit quantization schemes use either uniform [7, 59], or non-uniform quantization [13, 60, 34, 2]. Uniform quantization enables the use of integer or fixed-point math pipelines, allowing computation to be performed in the quantized domain. Non-uniform quantization requires dequantization, e.g. a codebook lookup, before doing computation in higher precision, limiting its benefits to model compression and bandwidth reduction. This paper focuses on leveraging quantization to accelerate computation, so we will restrict our focus to uniform quantization schemes.

While much of the aforementioned work has focused on CNNs for image classification, there are also many examples of applying quantization to other types of network architectures. Wu et al. [55] described how Google's Neural Machine Translation (GNMT), which employs a Long Short Term Memory (LSTM) Recurrent Neural Network (RNN), was trained with hard range constraints on multiple tensors to be more amenable to PTQ. A similar strategy was taken on MobileNet v2 [48], which restricts activations to be in the range [0, 6] (ReLU6). Bhandare et al. [4] quantized the smaller base Transformer [53] model targeting the int8 VNNI instructions on Intel CPUs. They use KL-Divergence [36] to calibrate the quantization ranges and apply PTQ. Zafrir et al. [58] quantized BERT [10] to int8 using both PTQ and QAT. In this paper, we present an evaluation of int8 quantization on all of the major network architectures with both PTQ and QAT.

More complex methods have also been proposed for training quantized models. Distillation has been used to train a quantized "student" model with a high precision, and often larger, "teacher" model. It has been applied to training quantized CNNs [37, 43], LSTMs [43] and Transformers [24]. Leng et al. [31] used the Alternating Direction Method of Multipliers (ADMM) as an alternative to STE when training quantized model. These methods generally target lower bit-width quantization, as QAT has been shown to be sufficient for int8 quantization. We have also found QAT to be sufficient for int8 quantization on the models we evaluated, and as such we chose not to included these methods in our evaluation of int8 quantization.



(a) Affine quantization

(b) Scale quantization

Figure 1: Quantization mapping of real values to int8

3 Quantization Fundamentals

We focus on uniform integer quantization as it enables computing matrix multiplications and convolutions in the integer domain, allowing the use of high throughput integer math pipelines. Uniform quantization can be divided in to two steps. First, choose the range of the real numbers to be quantized, clamping the values outside this range. Second, map the real values to integers representable by the bit-width of the quantized representation (round each mapped real value to the closest integer value).

In this Section we will consider higher precision floating-point formats like fp16 and fp32 to be real numbers for the purpose of discussion. Enabling integer operations in a pre-trained floating-point neural network requires two fundamental operations:

`Quantize`: convert a real number to a quantized integer representation (e.g. from fp32 to int8).

`Dequantize`: convert a number from quantized integer representation to a real number (e.g. from int32 to fp16).

We will first define the quantize and dequantize operations in Section 3.1 and discuss their implications in neural network quantization in Sections 3.2 and 3.3. Then we will discuss how the real ranges are chosen in Section 3.4.

3.1 Range Mapping

Let $[\beta, \alpha]$ be the range of representable real values chosen for quantization and b be the bit-width of the signed integer representation. Uniform quantization transforms the input value $x \in [\beta, \alpha]$ to lie within $[-2^{b-1}, 2^{b-1} - 1]$, where inputs outside the range are clipped to the nearest bound. Since we are considering only uniform transformations, there are only two choices for the transformation function: $f(x) = s \cdot x + z$ and its special case $f(x) = s \cdot x$, where $x, s, z \in \mathbb{R}$. In this paper we refer to these two choices as *affine* and *scale*, respectively.

3.1.1 Affine Quantization

Affine quantization maps a real value $x \in \mathbb{R}$ to a b-bit signed integer $x_q \in \{-2^{b-1}, -2^{b-1} + 1, \ldots, 2^{b-1} - 1\}$. Equations 1 and 2 define affine transformation function, $f(x) = s \cdot x + z$:

$$s = \frac{2^b - 1}{\alpha - \beta} \tag{1}$$

$$z = -\operatorname{round}(\beta \cdot s) - 2^{b-1} \tag{2}$$

where s is the scale factor and z is the zero-point - the integer value to which the real value zero is mapped. In the 8-bit case, $s = \frac{255}{\alpha - \beta}$ and $z = -round(\beta \cdot s) - 128$. Note that z is rounded to an integer value so that the real value of zero is exactly representable. This will result in a slight adjustment to the real representable range $[\beta, \alpha]$ [20].

The quantize operation is defined by Equation 3 and 4:

$$\text{clip}(x, l, u) \begin{cases} l, & x < l \\ x, & l \leq x \leq u \\ u, & x > u \end{cases} \tag{3}$$

$$x_q = \text{quantize}(x, b, s, z) = \text{clip}(\text{round}(s \cdot x + z), -2^{b-1}, 2^{b-1} - 1) \tag{4}$$

where $\text{round}()$ rounds to the nearest integer. Figure 1a shows the mapping of real values to int8 representation with affine quantization. Note that s is the ratio of the integer-representable and chosen real ranges.

Equation 5 shows the corresponding dequantize function, which computes an approximation of the original real valued input, $\hat{x} \approx x$.

$$\hat{x} = \text{dequantize}(x_q, s, z) = \frac{1}{s}(x_q - z) \tag{5}$$

3.1.2 Scale Quantization

Scale quantization performs range mapping with only a scale transformation. For simplicity we describe the symmetric variant of scale quantization (often called symmetric quantization [26]), where the input range and integer range are symmetric around zero. This means that for int8 we use the integer range $[-127, 127]$, opting not to use the value -128 in favor of symmetry. For 8-bit quantization, losing one out of 256 representable values is insignificant, but for lower bit quantization the trade-off between representable values and symmetry should be re-evaluated.

Figure 1b illustrates the mapping of real values to int8 with scale quantization. Equation 6 and 7 define scale quantization of a real value x, with a chosen representable range $[-\alpha, \alpha]$, producing a b-bit integer value, x_q.

$$s = \frac{2^{b-1} - 1}{\alpha} \tag{6}$$

$$x_q = \text{quantize}(x, b, s) = \text{clip}(\text{round}(s \cdot x), -2^{b-1} + 1, 2^{b-1} - 1) \tag{7}$$

Equation 8 shows the corresponding dequantize operation for scale quantization.

$$\hat{x} = \text{dequantize}(x_q, s) = \frac{1}{s} x_q \tag{8}$$

3.2 Tensor Quantization Granularity

There are several choices for sharing quantization parameters among tensor elements. We refer to this choice as quantization granularity. At the coarsest, per-tensor granularity, the same quantization parameters are shared by all elements in the tensor. The finest granularity would have individual quantization parameters per element. Intermediate granularities reuse parameters over various dimensions of the tensor - per row or per column for 2D matrices, per channel for 3D (image-like) tensors, etc.

We will consider two factors when choosing granularity: impact on model accuracy and computational cost. To understand the computational cost, we will examine matrix multiplication (note that this results in no loss of generality for math-intensive operations since convolutions can be expressed as matrix multiplications [5, 54]).

Consider a linear (fully-connected) layer that performs a matrix multiplication $Y = XW$, where $X = (x_{ik}) \in \mathbb{R}^{m \times p}$ is the input activation tensor, $W = (w_{kj}) \in \mathbb{R}^{p \times n}$ is the weight tensor, and $Y = (y_{ij}) \in \mathbb{R}^{m \times n}$ is the output tensor. The result of the real-valued matrix multiplication $Y = XW$ can be approximated with quantized tensors $X_q = (x_{q,ik}) \in \mathbb{Z}^{m \times p}$ and $W_q = (w_{q,kj}) \in \mathbb{Z}^{p \times n}$, by first dequantizing them, and then performing the matrix multiplication. First, consider tensors quantized at the finest granularity, per-element, with scale quantization:

$$y_{ij} = \sum_{k=1}^{p} x_{ik} \cdot w_{kj} \approx \sum_{k=1}^{p} \text{dequantize}(x_{q,ik}, s_{q,ik}) \cdot \text{dequantize}(w_{q,kj}, s_{w,kj}) = \sum_{k=1}^{p} \frac{1}{s_{x,ik}} x_{q,ik} \cdot \frac{1}{s_{w,kj}} w_{q,kj} \quad (9)$$

In order to use integer matrix multiplication the scales must be factored out of the summation on the right-hand side of Equation 9, for which the scales must be independent of k:

$$\frac{1}{s_{x,i} \cdot s_{w,j}} \sum_{k=1}^{p} x_{q,ik} \cdot w_{q,kj} \quad (10)$$

Thus, integer matrix multiplication is possible as long as the quantization granularity is per-row or per-tensor for activations and per-column or per-tensor for weights. For activations, only per-tensor quantization is practical for performance reasons. In the above formulation different rows belong to either different batch instances or items in a sequence and thus row count can vary at inference time. This prevents the per-row scaling factor from being computation offline (which would not be meaningful for different instances in a mini-batch), whereas determining them online imposes a compute overhead and in some cases results in poor accuracy (Dynamic Quantization discussion in [58]).

For maximum performance, activations should use per-tensor quantization granularity. Weights should be quantized at either per-tensor or per-column granularity for linear layers of the form $Y = XW$ (per-row for linear layers of the form $Y = XW^T$). The corresponding granularity to per-column in convolutions is per-kernel, or equivalently per-output-channel since each kernel produces a separate output channel [27, 29]. This is commonly referred to as "per-channel" weight quantization in literature and we follow that convention [21, 25, 26, 38, 46]. We examine the granularity impact on accuracy in Section 4.1.

3.3 Computational Cost of Affine Quantization

While both affine and scale quantization enable the use of integer arithmetic, affine quantization leads to more computationally expensive inference. As shown in equation 10, scale quantization results in an integer matrix multiply, followed by a point-wise floating-point multiplication. Given that a typical dot-product in a DNN comprises 100s to 1000s of multiply-add operations, a single floating-point operation at the end is a negligible cost. Furthermore, if per-tensor quantization is used for both arguments, a single floating-point multiplier is needed and is part of the GEMM API (often referred to as alpha) in BLAS libraries [11].

Affine quantization yields a more complex expression:

$$\begin{aligned} y_{ij} &\approx \sum_{k=1}^{p} \frac{1}{s_x}(x_{q,ik} - z_x)\frac{1}{s_{w,j}}(w_{q,kj} - z_{w,j}) \\ &= \frac{1}{s_x s_{w,j}} \Bigg(\underbrace{\sum_{k=1}^{p} x_{q,ik} w_{q,kj}}_{(1)} - \underbrace{\sum_{k=1}^{p} (w_{q,kj} z_x + z_x z_{w,j})}_{(2)} - \underbrace{\sum_{k=1}^{p} x_{q,ik} z_{w,j}}_{(3)} \Bigg) \end{aligned} \quad (11)$$

Computation can be broken down into three terms, as annotated in Equation 11. The first term is the integer dot product, just as in scale quantization (Equation 10). The second term consists of only integer weights and zero-points. As a result, this term can be computed offline, only adding an element-wise addition at inference time. If the layer has a bias then this term can be folded in without increasing inference cost. The third term, however, involves the quantized input matrix X_q, and thus cannot be computed offline. This extra computation, depending on implementation, can introduce considerable overhead, reducing or even eliminating the throughput advantage that integer math pipelines have over reduced precision floating-point. Note that this extra computation is incurred only if affine quantization is used for the weight matrix. Thus, to maximize inference performance we recommend using scale quantization for weights. While affine quantization could be used for activations without a performance penalty, we show in later sections that scale quantization is sufficient for int8 quantization of all the networks we studied.

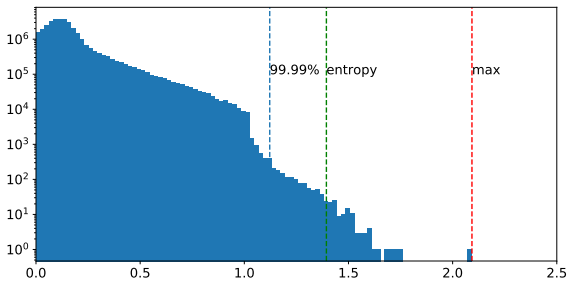


Figure 2: Histogram of input activations to layer 3 in ResNet50 and calibrated ranges

3.4 Calibration

Calibration is the process of choosing α and β for model weights and activations. For simplicity we describe calibration of a symmetric range, as needed for scale quantization. In this paper we consider three calibration methods:

`Max`: Use the maximum absolute value seen during calibration [52].

`Entropy`: Use KL divergence to minimize information loss between the original floating-point values and values that could be represented by the quantized format. This is the default method used by TensorRT [36].

`Percentile`: Set the range to a percentile of the distribution of absolute values seen during calibration [33]. For example, 99% calibration would clip 1% of the largest magnitude values.

Figure 2 shows a log scaled histogram of activations feeding into layer1.0.conv2 of ResNet50. Ranges derived from max, entropy, and 99.99% percentile calibration are shown with dashed lines. Note that the activations are strictly non-negative because this layer directly follows a ReLU activation function [39]. Max calibration represents the largest value in the distribution, maintaining the full range while having low precision. Clipping the distribution trades off a large clipping error on a few outlier values for smaller rounding errors on a majority of the values. Both entropy and percentile calibration clip some outlier values in order to increase the resolution of inlier values.

4 Post Training Quantization

In this section we evaluate various Post Training Quantization (PTQ) parameter choices, as described in Section 3. Quantization parameters are calibrated offline by processing the trained model weights and activations generated by running inference on a sample dataset, no further training is involved. These quantization parameters are evaluated on a variety of neural network tasks and models, summarized in Table 2. More details on these networks can be found in Appendix A. The selected models comprise multiple types of network architectures: convolutional feed forward networks, recurrent networks, and attention-based networks. We report accuracy metrics computed on the evaluation set of the corresponding dataset. Metrics for all tasks are reported as percentages, where higher is better and 100% is a perfect score. For metric consistency, we report word accuracy (WAcc) for speech recognition instead of the more commonly used Word Error Rate (WER), where $\mathrm{WAcc} = 100\% - \mathrm{WER}$. Note that accuracy metrics for different tasks are computed in very different ways, thus it is not meaningful to compare absolute changes in accuracy when quantizing different models. Therefore, when discussing accuracy impact we will refer to the relative accuracy change, computed by $(\mathrm{acc}_{int8} - \mathrm{acc}_{fp32})/\mathrm{acc}_{fp32}$.

Our experiments are conducted using PyTorch [42], with custom quantization operations. We focus on quantizing the computationally intensive operations, including convolutions, linear (fully-connected) layers, LSTM cells, projection layers and other matrix multiplications. Most of the other layers, such as softmax and batch normalization, are not quantized unless stated otherwise. An operation is quantized by quantizing all of its inputs (e.g. weights and activations). The output of a quantizated operation is not quantized to int8 because the operation that follows it may require higher precision, e.g. nonlinear operations. Furthermore, consecutive operations can be executed with a fused implementation, avoiding memory reads and writes for the intermediate values. Therefore we leave quantization of the output activations to the input of the next operation. Appendix C discusses how batch normalization can be eliminated by folding it into the preceding layer for inference.

Task	Model	Accuracy	Metric	Dataset (evaluation set)
Classification	MobileNet v1	71.88	Top1	ImageNet 2012 (val)
	MobileNet v2	71.88		
	ResNet50 v1.5	76.16		
	ResNet152 v1.5	78.32		
	Inception v3	77.34		
	Inception v4	79.71		
	ResNeXt50	77.61		
	ResNeXt101	79.30		
	EfficientNet b0	76.85		
	EfficientNet b3	81.61		
Detection	Faster R-CNN	36.95	mAP	COCO 2017 (val)
	Mask R-CNN	37.89		
	Retinanet	39.30		
Segmentation	FCN	63.70	mIoU	COCO 2017 (val)
	DeepLabV3	67.40		
Translation	GNMT	24.27	BLEU	WMT16 en-de (newtest2014)
	Transformer	28.27		
Speech Recognition	Jasper	96.09 (3.91)	WAcc (WER)	LibriSpeech (test-clean)
Language model	BERT Large	91.01	F1	Squad v1.1 (dev)

Table 2: Summary of networks and pre-trained model accuracy

Model	fp32	Per-channel	Per-channel fold BN	Per-tensor	Per-tensor fold BN
MobileNet v1	71.88	71.59	71.59	69.58	66.88
MobileNet v2	71.88	71.61	71.61	71.12	70.21
ResNet50 v1.5	76.16	76.14	76.14	75.83	75.84
ResNeXt50	77.61	77.62	77.62	77.48	77.45
EfficientNet b0	76.85	76.72	76.72	76.68	12.93

Table 3: Accuracy with int8 quantization of weights only: per-tensor vs per-channel granularity. Fold BN indicates batch norms were folded into the preceding convolution before quantization

4.1 Weight Quantization

We first evaluate weight quantization in isolation, since their values do not depend on network inputs, and demonstrate that max calibration is sufficient to maintain accuracy for int8 weights. Table 3 compares the accuracy impact of the per-tensor and per-channel quantization granularities, which in Section 3.2 were shown to require minimal compute overheads. While per-tensor quantization results in substantial accuracy losses for some networks, accuracy loss is more pronounced and even catastrophic for EfficientNet once batch-normalization (BN) parameters are folded into convolution layers. BN folding (Appendix C) is a common technique to speed up inference as it completely eliminates this memory-limited operation without changing the underlying mathematics. However, as BN parameters are learned per channel, their folding can result in significantly different weight value distributions across channels. Fortunately, as Table 3 shows, per-channel quantization granularity maintains model accuracy even with BN folding. Table 4 reports per-channel (per-column for linear layers) granularity and indicates that max calibration is sufficient to maintain accuracy when quantizing weights to int8. The rest of the experiments in this paper use per-channel max calibration for weights.

4.2 Activation Quantization

Table 5 shows activation quantization results for different calibration methods: max, entropy and percentiles from 99.9% to 99.9999%. Details on activation calibration can be found in Appendix A. In all cases, weights were quantized per-channel with max calibration as described in Section 4.1.

Model	fp32	Accuracy	Relative	Model	fp32	Accuracy	Relative
MobileNet v1	71.88	71.59	-0.40%	Faster R-CNN	36.95	36.86	-0.24%
MobileNet v2	71.88	71.61	-0.38%	Mask R-CNN	37.89	37.84	-0.13%
ResNet50 v1.5	76.16	76.14	-0.03%	Retinanet	39.30	39.20	-0.25%
ResNet152 v1.5	78.32	78.28	-0.05%	FCN	63.70	63.70	0.00%
Inception v3	77.34	77.44	0.13%	DeepLabV3	67.40	67.40	0.00%
Inception v4	79.71	79.64	-0.09%	GNMT	24.27	24.41	0.58%
ResNeXt50	77.61	77.62	0.01%	Transformer	28.27	28.58	1.10%
ResNeXt101	79.30	79.29	-0.01%	Jasper	96.09	96.10	0.01%
EfficientNet b0	76.85	76.72	-0.17%	Bert Large	91.01	90.94	-0.08%
EfficientNet b3	81.61	81.55	-0.07%				

Table 4: Accuracy with int8 quantization of weights only: per-channel granularity, max calibration

Models	fp32	Max	Entropy	99.9%	99.99%	99.999%	99.9999%
MobileNet v1	71.88	69.51	70.19	**70.39**	70.29	69.97	69.57
MobileNet v2	71.88	69.41	70.28	70.68	**71.14**	70.72	70.23
ResNet50 v1.5	76.16	75.82	**76.05**	75.68	75.98	75.97	76.00
ResNet152 v1.5	78.32	77.93	**78.21**	77.62	78.17	78.17	78.19
Inception v3	77.34	72.53	**77.54**	76.21	77.52	77.43	77.37
Inception v4	79.71	0.12	79.60	78.16	**79.63**	79.12	71.19
ResNeXt50	77.61	77.31	**77.46**	77.04	77.39	77.45	77.39
ResNeXt101	79.30	78.74	79.09	78.77	79.15	**79.17**	79.05
EfficientNet b0	76.85	22.3	**72.06**	70.87	68.33	51.88	42.49
EfficientNet b3	81.61	54.27	76.96	77.80	**80.28**	80.06	77.13
Faster R-CNN	36.95	36.38	**36.82**	35.22	36.69	36.76	36.78
Mask R-CNN	37.89	37.51	37.75	36.17	37.55	37.72	**37.80**
Retinanet	39.30	38.90	38.97	35.34	38.55	**39.19**	**39.19**
FCN	63.70	63.40	64.00	62.20	64.00	**63.90**	63.60
DeepLabV3	67.40	67.20	67.40	66.40	67.40	**67.50**	67.40
GNMT	24.27	24.31	**24.53**	24.34	24.36	24.38	24.33
Transformer	28.27	21.23	21.88	24.49	**27.71**	20.22	20.44
Jasper	96.09	95.99	**96.11**	95.77	96.09	96.09	96.03
BERT Large	91.01	85.92	37.40	26.18	89.59	**90.20**	90.10

Table 5: Post training quantization accuracy. Weights use per-channel or per-column max calibration. Activations use the calibration listed. Best quantized accuracy per network is in bold.

For most of the networks, there is at least one activation calibration method that achieves acceptable accuracy, except for MobileNets, EfficientNets, Transformer and BERT where the accuracy drop is larger than 1%. Max calibration leads to inconsistent quality across various networks, leading to particularly large accuracy drops for Inception v4, EfficientNets and Transformer, presumably due to their outlier values. 99.9% percentile calibration clips the large magnitude values too aggressive and leads to significant accuracy drops on most networks. The best post training quantization results are achieved with entropy, 99.99%, or 99.999% percentile calibrations, though no single calibration is best for all networks.

5 Techniques to Recover Accuracy

While many networks maintain accuracy after post training quantization, there are cases where accuracy loss is substantial. A number of techniques are available to recover accuracy. The simplest one is partial quantization, described in Section 5.1, which leaves the most sensitive layers unquantized. One also has an option to train networks with quantization, as described in Section 5.2. Finally, there are also approaches that jointly learn the model weights and quantization parameters.

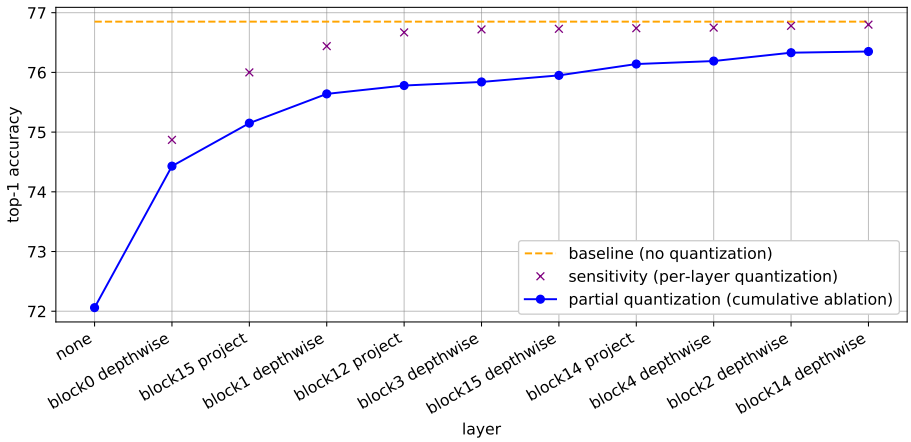


Figure 3: Partial quantization of EfficientNet b0, showing the 10 most sensitive layers in order of increasing accuracy. Sensitivity shows the accuracy from the sensitivity analysis when only the corresponding layer inputs are quantized. Partial quantization shows the accuracy when the corresponding layer, and all layers to the left, are not quantized.

	fp32		Full int8		Partial int8	
Model	Accuracy	Calibration	Total quantized layers	Accuracy	Skipped layers	Accuracy
MobileNet v1	71.88	max	28	69.51	2	71.50
EfficientNet b0	76.85	entropy	82	72.06	10	76.35
EfficientNet b3	81.61	99.99%	131	76.96	3	81.27
Transformer	28.27	max	121	21.23	5	28.20
BERT large	91.01	max	244	85.92	141	90.41

Table 6: Partial post training quantization

5.1 Partial Quantization

Often just a few quantized layers contribute to most of the accuracy loss of a quantized model. We can trade off some performance to increase accuracy by leaving these sensitive layers unquantized (i.e. leaving their inputs and computation in floating-point). Since quantization of one layer affects the inputs of others, finding the optimal set of layers to quantize can require evaluating an exponential number of configurations. Instead, we propose using a one-at-a-time sensitivity analysis as a more tractable approach to infer which layers contribute most to the accuracy drop.

During sensitivity analysis a single layer is quantized at a time, and model accuracy is evaluated. We refer to layers that result in lower accuracy when quantized as being more "sensitive" to quantization. We sort the layers in descending order of sensitivity, and skip quantization of the most sensitive layers until the desired accuracy is achieved.

Figure 3 shows an example of sensitivity analysis and partial quantization of EfficientNet b0. Starting from entropy calibration, we quantize one layer at a time and evaluate accuracy. For clarity we are only showing the 10 most sensitive layers. In this example, skipping the 10 most sensitive layers reduces the relative top-1 accuracy drop to 0.65%. Since there are 82 convolution layers, keeping 10 in floating-point while quantizing the remaining 72 maintains most of the performance benefit.

As reported in Table 5, MobileNet v1, EfficientNets, Transformer, and BERT all incurred a substantial loss in accuracy when quantized with various calibrations. We list the results of partial quantization for these networks in Table 6. With the exception of BERT, these networks need to skip quantization of only a few of the most-sensitive layers to recover accuracy to within 1% of the fp32 accuracy. For BERT, sensitivity analysis does not reveal any particular layer that contributes more to the accuracy drop. As a result we cannot identify a small subset of layers to leave in floating-point. To address this we need to consider different approaches. Section 5.2, incorporates quantization with training to recover accuracy. Additionally, Appendix D examines the GELU activation function in BERT and presents a simple augmentation to significantly improve post training quantization accuracy.

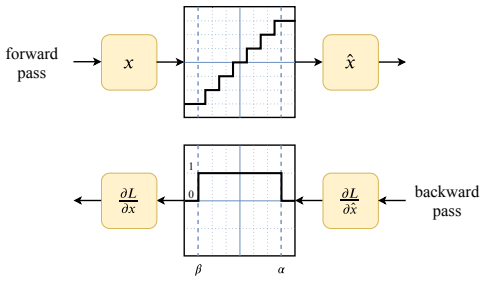


Figure 4: 3-bit fake quantization forward and backward pass with STE derivative approximation.

5.2 Quantization-Aware Training

Quantization Aware Training (QAT) describes the technique of inserting quantization operations in to the neural network before training or fine-tuning, to allow the network to adapt to the quantized weights and activations. Appendix B illustrates how this can lead to a better result. We apply QAT to fine-tuning as it has been shown that starting from a pre-trained network and fine-tuning leads to better accuracy [37, 26] and requires significantly fewer iterations [33]. This also allows us to leverage the calibrated pre-trained models from Section 4. Note that we keep the quantization ranges fixed throughout fine-tuning. Another approach is to learn the ranges, which we evaluate in Section 5.3.

A common approach to implementing QAT is to insert fake quantization, also called simulated quantization [26], operations into a floating-point network. Equation 12 defines fake quantization as a quantize and dequantize operation that produces an approximate version of the input, $\hat{x} \approx x$, where x and $\hat{x}$ are both floating-point values.

$$\hat{x} = \text{dequantize}(\text{quantize}(x, b, s), b, s) \tag{12}$$

We add fake quantization operations at the inputs of the operation we wish to quantize to simulate the effects of quantization. Recall the matrix multiplication example in Section 3.2. Equation 9 is effectively a fake quantized matrix multiplication. After training, we transform the network to enable a quantized integer matrix multiply as shown in Equation 10.

One challenge to training in floating-point with quantization is that the quantization operation's derivative is undefined at the step boundaries and zero everywhere else. The derivative is required to compute loss gradients on the backward pass of each training iteration. QAT addresses this by using the Straight-through Estimator (STE) [3] as shown in Figure 4. As defined in Equation 13, STE approximates the derivative of the fake quantization function to be 1 for inputs in the representable range $[\beta, \alpha]$ as defined in Section 3.1.

$$\frac{d\hat{x}}{dx} = \begin{cases} 0, & y < \beta \\ 1, & \beta \leq y \leq \alpha \\ 0, & y > \alpha \end{cases} \tag{13}$$

Table 7 summarizes the best results of both post training quantization and fine-tuned quantization. PTQ best reports the best result for each quantized network in Table 5 and the corresponding calibration. QAT reports the accuracy after fine-tuning using the best calibration as determined by PTQ. Details of the finetuning methodology and the complete set of QAT results can be found in Appendix A.2.

As Table 7 shows, quantization-aware fine-tuning improves accuracy in most cases, the only exceptions being ResNeXt-101, Mask R-CNN, and GNMT where post training quantization achieves a marginally better result. It is worth noting that for all 3 of these cases the differences in accuracy are essentially at the noise level (differences in accuracy one would observe when training from different random initializations). We do not interpret these cases as evidence that fine-tuning reduces accuracy, they are more likely to indicate that fine-tuning does not appreciably change accuracy beyond run-to-run variation. Likewise, we do not interpret cases where accuracy is higher than fp32 as quantization acting as a regularizer, it is more likely to be noise or the result of the additional fine-tuning. EfficientNet b3 is another case worth examining - as our code did not have auto augmentation [9], used to train the original model, fine-tuning even in fp32 causes a slight accuracy drop to 81.3. Nevertheless, with fine-tuning all networks were able to maintain their accuracy well within 1% of the original pre-trained fp32 model.

Model	fp32 Accuracy	PTQ best Calibration	PTQ best Accuracy	PTQ best Relative	QAT Accuracy	QAT Relative
MobileNet v1	71.88	99.9%	70.39	-2.07%	72.07	0.26%
MobileNet v2	71.88	99.99%	71.14	-1.03%	71.56	-0.45%
ResNet50 v1.5	76.16	Entropy	76.05	-0.14%	76.85	0.91%
ResNet152 v1.5	78.32	Entropy	78.21	-0.14%	78.61	0.37%
Inception v3	77.34	Entropy	77.54	0.26%	78.43	1.41%
Inception v4	79.71	99.99%	79.63	-0.10%	80.14	0.54%
ResNeXt50	77.61	Entropy	77.46	-0.19%	77.67	0.08%
ResNeXt101	79.30	99.999%	79.17	-0.16%	79.01	-0.37%
EfficientNet b0	76.85	Entropy	72.06	-6.23%	76.95	0.13%
EfficientNet b3	81.61	99.99%	80.28	-1.63%	81.07	-0.66%
Faster R-CNN	36.95	Entropy	36.82	-0.35%	36.76	-0.51%
Mask R-CNN	37.89	99.9999%	37.80	-0.24%	37.75	-0.37%
Retinanet	39.30	99.999%	39.19	-0.28%	39.25	-0.13%
FCN	63.70	Entropy	64.00	0.47%	64.10	0.63%
DeepLabV3	67.40	99.999%	67.50	0.15%	67.50	0.15%
GNMT	24.27	Entropy	24.53	1.07%	24.38	0.45%
Transformer	28.27	99.99%	27.71	-1.98%	28.21	-0.21%
Jasper	96.09	Entropy	96.11	0.02%	96.10	0.01%
BERT Large	91.01	99.999%	90.20	-0.89%	90.67	-0.37%

Table 7: Summary of Post Training Quantization and Quantization Aware Training. PTQ best reports the best accuracy and corresponding calibration for each model. QAT reports accuracy after fine-tuning starting from the best PTQ model.

Models	fp32	Fixed max	Learned from max	Fixed best	Learned from best
Inception v3	77.34	76.43	78.33	78.43	78.50
Inception v4	79.71	68.38	73.88	80.14	80.00
Faster R-CNN	36.95	36.62	36.68	36.76	36.81
FCN	63.70	63.40	63.50	64.10	64.00
Transformer	28.27	28.42	28.08	28.21	28.39
Jasper	96.09	96.11	96.05	96.10	96.06
BERT Large	91.01	90.29	90.55	90.67	90.61

Table 8: Learned and fixed range fine-tuning accuracy. Activation ranges initialized to max and best PTQ accuracy

5.3 Learning Quantization Parameters

While the techniques described in the previous sections relied on quantization parameters calibrated on the pre-trained network, it is also possible to jointly learn the quantization parameters along with the model weights. PACT [6] proposed learning the ranges for activation quantization during training. In this section we adopt PACT as an enhancement to our quantization aware fine-tuning procedure. We follow the same fine-tuning schedule as before, described in Appendix A, but allow the ranges of each quantized activation tensor to be learned along with the weights, as opposed to keeping them fixed throughout fine-tuning.

Table 8 shows a selection of networks fine-tuned with fixed and learned activation ranges for different initial calibrations. The "best" calibration refers to the calibration that produced the best accuracy with PTQ, as shown in Table 5. When the activation quantization is initialized with max calibration, learning the range results in higher accuracy than keeping it fixed for most networks. In particular it results in substantial accuracy improvements where fixed max ranges resulted in a significant accuracy drop. However, when activation ranges are initialized the to the best calibration for each network, learning the ranges yield very similar results to fixed ranges. This suggests that learning the ranges does not offer additional benefit for int8 over QAT if activation ranges are already carefully calibrated. However, this may not be the optimal application of PACT. Comparing the learned range results on Inception v4 suggest that when starting from max, the network was not able to learn a good activation ranges in the given fine-tuning schedule. We expect that PACT would be able to learn a better range with longer fine-tuning, or a separate optimization schedule and hyperparameters for the range parameters, such and learning rate and weight decay.

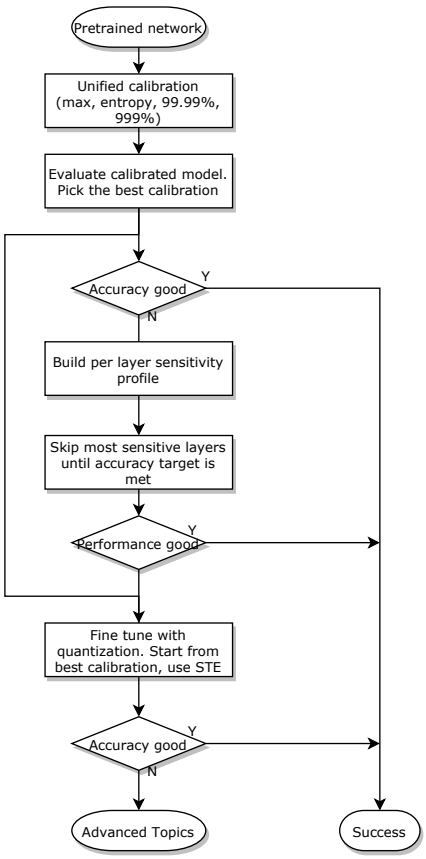


Figure 5: Flow chart of our recommended quantization workflow

6 Recommended Workflow

Based on the results in Sections 4 and 5, we recommend the following for int8 quantization:

- Weights:
  - Use scale quantization with per-column/per-channel granularity
  - Use a symmetric integer range for quantization [-127, 127]) and max calibration
- Activations:
  - Use scale quantization with with per-tensor granularity

We recommend the following procedure to quantize a pre-trained neural network.

- `PTQ`: Quantize all the computationally intensive layers (convolution, linear, matrix multiplication, etc.) and run activation calibration including max, entropy and 99.99%, 99.999% percentile. If none of the calibrations yield the desired accuracy continue to partial quantization or QAT.
- `Partial Quantization`: Perform sensitivity analysis to identify the most sensitive layers and leave them in floating-point. If the impact on computational performance is not acceptable or an acceptable accuracy cannot be reached, continue to QAT.
- `QAT`: Start from the best calibrated quantized model. Use QAT to fine-tune for around 10% of the original training schedule with an annealing learning rate schedule starting at 1% of the initial training learning rate. Refer to Appendix A.2 for specific hyperparameter choices.

Figure 5 summarizes the above workflow in a flowchart.

7 Conclusions

This paper reviewed the mathematical background for integer quantization of neural networks, as well as some performance-related reasons for choosing quantization parameters. We empirically evaluated various choices for int8 quantization of a variety of models, leading to a quantization workflow proposal. Following this workflow we demonstrated that all models we studied can be quantized to int8 with accuracy that either matches or is within 1% of the floating-point model accuracy. This included networks that are challenging for quantization, such as MobileNets and BERT. The workflow involves only post-training quantization, partial quantization, and quantization-aware fine-tuning techniques. Some more complex techniques, such as ADMM and distillation, were not required for int8 quantization of these models. However, these techniques should be evaluated when quantizing to even lower-bit integer representations, which we leave to future work.

References

[1] 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, Dan Mane, 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 Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Chaim Baskin, Eli Schwartz, Evgenii Zheltonozhskii, Natan Liss, Raja Giryes, Alex M Bronstein, and Avi Mendelson. Uniq: Uniform noise injection for non-uniform quantization of neural networks. *arXiv preprint arXiv:1804.10969*, 2018.

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[4] Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. Efficient 8-bit quantization of transformer neural machine language translation model. *arXiv preprint arXiv:1906.00532*, 2019.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[6] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

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

[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *NIPS*, 28:3123–3131, 2015.

[9] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 113–123, 2019.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[12] Boris Ginsburg, Patrice Castonguay, Oleksii Hrinchuk, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, Huyen Nguyen, and Jonathan M Cohen. Stochastic gradient methods with layer-wise adaptive moments for training of deep networks. *arXiv preprint arXiv:1905.11286*, 2019.

[13] 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.

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

[15] Dan Hendrycks and Kevin Gimpel. Gaussian Error Linear Units (GELUs). *arXiv preprint arXiv:1606.08415*, 2016.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Simplifying neural nets by discovering flat minima. In *Advances in neural information processing systems*, pages 529–536, 1995.

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[18] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

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

[20] 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.

[21] Sambhav R Jain, Albert Gural, Michael Wu, and Chris H Dick. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. *arXiv preprint arXiv:1903.08066*, 2(3):7, 2019.

[22] Stanisław Jastrzębski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[23] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[24] Marcin Junczys-Dowmunt, Kenneth Heafield, Hieu Hoang, Roman Grundkiewicz, and Anthony Aue. Marian: Cost-effective high-quality neural machine translation in c++. *arXiv preprint arXiv:1805.12096*, 2018.

[25] Alexander Kozlov, Ivan Lazarevich, Vasily Shamporov, Nikolay Lyalyushkin, and Yury Gorbachev. Neural network compression framework for fast model inference. *arXiv preprint arXiv:2002.08679*, 2020.

[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] Akhilesh Kumar, Sailesh Kottapalli, Ian Steiner, Bob Valentine, Israel Hirsh, Geetha Vearaman, Lily Looi, Mohamed Arafa, Andy Rudoff, Sreenivas Mandava, Bahaa Fahim, and Sujal Vora. Future Intel Xeon Scalable Processor (Codename: Cascade Lake-SP). In *Hotchips 2018*, 2018.

[29] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.

[30] Dongsoo Lee and Byeongwook Kim. Retraining-based iterative weight quantization for deep neural networks. *arXiv preprint arXiv:1805.11233*, 2018.

[31] Cong Leng, Zesheng Dou, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with admm. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[32] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[33] Jeffrey L McKinstry, Steven K Esser, Rathinakumar Appuswamy, Deepika Bablani, John V Arthur, Izzet B Yildiz, and Dharmendra S Modha. Discovering low-precision networks close to full-precision networks for efficient embedded inference. *arXiv preprint arXiv:1809.04191*, 2018.

[34] Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. Ternary neural networks with fine-grained quantization. *arXiv preprint arXiv:1705.01462*, 2017.

[35] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[36] Szymon Migacz. Nvidia 8-bit inference width tensorrt. In *GPU Technology Conference*, 2017.

[37] Asit Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. *arXiv preprint arXiv:1711.05852*, 2017.

[38] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1325–1334, 2019.

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

[40] NVIDIA. NVIDIA Turing GPU architecture: Graphics reinvented. `https://www.nvidia.com/content/dam/en-zz/Solutions/designvisualization/technologies/turing-architecture/NVIDIATuring-Architecture-Whitepaper.pdf`, 2018.

[41] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

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

[44] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[45] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[46] Manuele Rusci, Alessandro Capotondi, and Luca Benini. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. *arXiv preprint arXiv:1905.13082*, 2019.

[47] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[48] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[49] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[50] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

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

[52] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS*, 2011.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[54] Pete Warden. Why gemm is at the heart of deep learning. https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow, 2015.

[55] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[56] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[57] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992*, 2018.

[58] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019.

[59] 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.

[60] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[61] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. Neural network distiller: A python package for dnn compression research. *arXiv preprint arXiv:1910.12232*, 2019.

Appendices

A Evaluation Details

A.1 Model Definitions

Model	Configuration	Calibration samples	Source
MobileNet v1	width_mult=1.0	1024	github.com/marvis/pytorch-mobilenet
MobileNet v2	width_mult=1.0	1024	github.com/pytorch/vision
ResNet50 v1.5		1024	
ResNet152 v1.5		1024	
Inception v3		1024	
Inception v4		1024	
ResNeXt50	32x4d	1024	
ResNeXt101	32x8d	1024	
EfficientNet b0		1024	github.com/lukemelas/efficientnet-pytorch
EfficientNet b3		1024	
Faster R-CNN	resnet50-fpn	512	github.com/pytorch/vision
Mask R-CNN	resnet50-fpn	512	
FCN	resnet101	512	
DeepLabV3	resnet101	512	
Retinanet	resnext101-32x4d-fpn	512	github.com/open-mmlab/mmdetection
Transformer	vaswani_en_de_big	14336 tokens	github.com/pytorch/fairseq
GNMT	4 layers	128	github.com/nvidia/deeplearningexamples
Jasper		full dev-clean	
Bert Large	fine-tuned for QA	128	github.com/huggingface/pytorch-transformers

Table 9: Network details

Table 9 lists additional details of the models listed in Table 2. We evaluated a large variety of CNNs for image classification. MobileNets are small networks that target inference on mobile devices [17, 48]. They are parameterized to scale to various channel widths and image resolutions. In this paper we evaluate the base configurations with width multiplier 1 and resolution 224x224. We also evaluated a number of larger CNNs [14, 56, 50, 49], including EfficientNets [51], which achieve state-of-the-art accuracy on ImageNet. All CNNs use 224x224 inputs except for Inception v3 and v4, which use 299x299. We evaluated two detection and two segmentation networks from Torchvision, and an additional segmentation network, RetinaNet [32]. We evaluated two translation models, the 4 layers GNMT model [55] and the large configuration of Transformer [53]. For speech recognition we evaluated Jasper which achieves state-of-the-art WER on public speech datasets [41]. For language modeling we use BERT large uncased and fine-tuned for question answering.

Models were calibrated with the number of samples listed from the training set of the respective dataset listed in Table 2, except for Jasper, which was calibrated on the dev set and evaluated on the test set. PyTorch implementations of all the models along were provided by the listed source repositories. We used the pre-trained weights provided by each repository, except for MobileNet v1 and EfficientNets where pre-trained weights were not available. MobileNet v1 was trained using the reference training script and hyperparameters for MobileNet v2 from Torchvision. Pre-trained weights for EfficientNets were converted to PyTorch from weights provided by TensorFlow [1][1].

[1] https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet

Task/Model	Dataset	Optimizer	Epochs	Initial learning rate	Batch size
Classification	ImageNet 2012	SGD	15	1.0e-3	256
Detection	COCO 2017	SGD	3	2.5e-5	16
Segmentation	COCO 2017	SGD	3	2.5e-5	32
Transformer	WMT16 en-de	ADAM	3	5.0e-4	max tokens = 3584
GNMT	WMT16 en-de	SGD	1	2.0e-3	1024, max seq. length = 50
Jasper	LibriSpeech	NovoGrad[12]	80	1.5e-2	256
BERT	SQuAD v1.1	ADAM	2	3.0e-5	12, max seq. length = 384

Table 10: Fine-tuning schedule and configuration

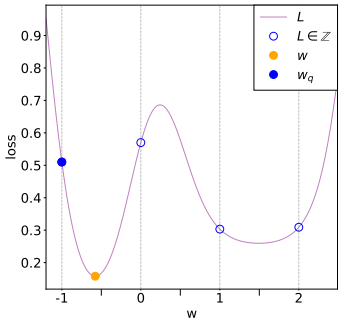
Models	fp32	Max	Entropy	99.9%	99.99%	99.999%	99.9999%
MobileNet v1	71.88	71.80	72.11	72.07	**72.14**	71.89	71.91
MobileNet v2	71.88	71.11	71.50	71.48	**71.56**	71.28	71.34
ResNet50 v1.5	76.16	76.68	**76.85**	76.59	76.67	76.77	76.81
ResNet152 v1.5	78.32	78.64	78.61	78.61	**78.69**	78.65	78.65
Inception v3	77.34	76.43	78.43	78.33	**78.49**	78.36	78.38
Inception v4	79.71	68.38	80.07	80.01	**80.14**	79.94	78.82
ResNeXt50	77.61	77.38	**77.67**	77.48	77.56	77.51	77.51
ResNeXt101	79.30	78.98	78.99	79.00	78.99	79.01	**79.04**
EfficientNet b0	76.85	76.16	76.95	76.85	**77.09**	76.98	76.63
EfficientNet b3	81.61	80.51	80.63	80.62	81.07	**81.09**	80.92
Faster R-CNN	36.95	36.62	36.76	36.31	36.76	**36.83**	36.76
Mask R-CNN	37.89	37.63	37.74	37.26	37.67	**37.76**	37.75
Retinanet	39.30	39.03	39.11	37.76	38.97	**39.25**	39.20
FCN	63.70	63.40	64.10	63.90	**64.20**	63.90	63.40
DeepLabV3	67.40	67.10	67.30	66.90	67.20	**67.50**	67.20
GNMT	24.27	**24.49**	24.38	24.35	24.41	24.48	24.35
Transformer	28.27	28.42	**28.46**	28.23	28.21	28.04	28.10
Jasper	96.09	**96.11**	96.10	95.23	95.94	96.01	96.08
BERT Large	91.01	90.29	90.14	89.97	90.50	**90.67**	90.60

Table 11: Fine-tuned quantization. Best accuracy in bold. Accuracy from best PTQ calibration per network underlined.

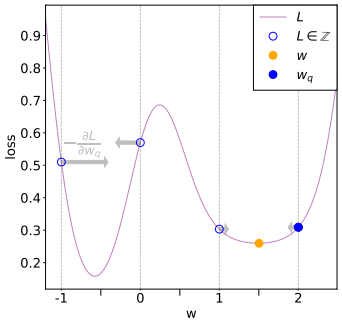
A.2 Quantization Aware Training

Table 10 shows the fine-tuning hyperparameters used in the quantization aware fine-tuning experiments. For networks that are trained on multiple datasets (detection/segmentation networks and BERT) we only fine-tuned on the final dataset (COCO and SQuAD). In general, only the initial learning rate value and learning rate schedule are changed from the original training session. We fine-tune for around 1/10th of the original training steps. The fine-tuning learning rate starts at 1/100th of the initial value used in the original training session and is decayed down to 1/100th of the initial fine-tuning learning rate. BERT is an exception. Since it pre-trains a language model and only fine-tunes on SQuAD for 2 epochs, we instead repeat the full fine-tuning schedule for QAT. We used a cosine annealing learning rate schedule which follows the monotonically decreasing half period of the cosine function.

Table 11 shows fine-tuned quantization accuracy for all networks and activation range calibration settings. Note that we always use the full per-column/per-channel range for weights (max calibration). It shows that with fine-tuning, accuracy improves for almost all the cases, especially those that suffer large accuracy drops after PTQ, for example max calibration. For many of the models, the best PTQ calibration is also the best calibration for QAT, indicated by results that are both bold and underlined. Even when QAT achieves higher accuracy with a different calibration, the difference in results is marginal. This result suggests that evaluating multiple activation calibrations during PTQ is a good heuristic to choose a calibration for QAT.



(a) Post training quantization



(b) After quantization aware fine-tuning

Figure 6: Example 1D loss function. The model, w, is scale quantized with scale factor 1. a) PTQ: model converges to a narrow minimum. b) QAT: model finds a wide minimum with lower loss quantization points.

B Intuition for QAT

To gain some intuition for why quantization-aware training may improve accuracy of the quantized model, consider the simple example in Figure 6. Neural networks are trained by minimizing a loss function with stochastic gradient descent. Loss gradients with respect to the network weights, $\frac{\delta L}{\delta w}$, are computed and weights are iteratively updated in the direction of the negative gradient until the model converges to some minimum. Figure 6a shows a one-dimensional loss function for a model with a single parameter, w, that has converged to a local minimum, $w \approx -0.5$. When post training quantization is applied, with a scale factor of 1 for sake of example, the weight is quantized to the nearest integer, $w_q = -1$, causing a significant increase in the loss. In such a case we say that the model has converged to a "narrow" minimum, since a small change in the weight leads to a large change in loss.

By training with quantization, we may potentially avoid these narrow minima by computing gradients with respect to the quantized weights, as shown in Figure 6b. In doing so, narrow minima will result in larger gradients, potentially allowing the model to explore the loss landscape for "wide" [22] or "flat" [16, 30] minima, where quantization points have lower loss, and thus higher accuracy.

C Batch normalization folding

Batch normalization folding is a common inference optimization applied to neural networks [20]. At inference, batch normalization layers performs the affine operation shown in Equation 14:

$$
\begin{aligned}
c &= \frac{\gamma}{\sqrt{Var[y] + \epsilon}} \\
d &= \beta - \frac{\gamma E[y]}{\sqrt{Var[y] + \epsilon}} \\
z &= \mathrm{BN}(y) = c \cdot y + d
\end{aligned}
\tag{14}
$$

where β, γ, $E[y]$, and $Var[y]$ are determined during training and fixed during inference, and ϵ is a constant [19]. Typically, following a fully connected layer the batch normalization is computed per activation. Consider a fully connected layer that performs the matrix multiplication and bias add shown in Equation 15:

$$
y_j = \sum_{k=1}^{p} x_k \cdot w_{kj} + b_j \tag{15}
$$

When the fully connected layer is followed by a batch normalization layer, $\mathbf{z} = \mathrm{BN}(\mathbf{x}W + \mathbf{b})$, the batch normalization can be folded into the weights and biases of the fully connected layer, as show in Equation 16:

$$z_j = \sum_{k=1}^{p} x_k \cdot \underbrace{c_j \cdot w_{kj}}_{w'_{kj}} + \underbrace{c_j \cdot b_j + d_j}_{b'_j} \tag{16}$$

resulting in a fully connected layer performing the operation $\mathbf{z} = \mathbf{x}W' + \mathbf{b}'$. Since convolutions can be mapped to fully connected layers, and batch normalization in CNNs is per channel, we can apply the same optimization.

D Novel activation functions

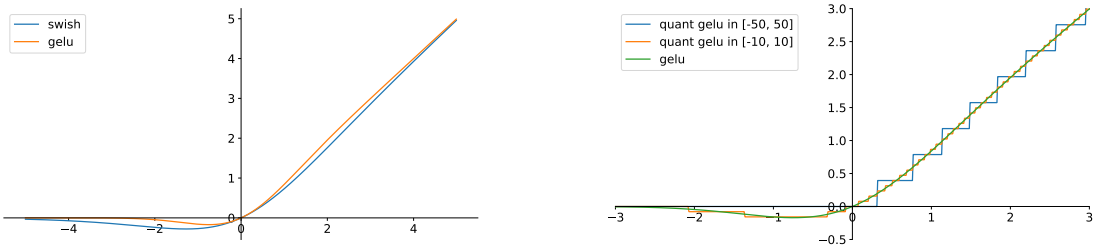
Two more recently developed activation functions are Swish (Equation 17) [44] and GELU (Equation 18) [15], used in EfficientNets and BERT, respectively.

$$\mathrm{swish}(x) = x \cdot \mathrm{sigmoid}(x) \tag{17}$$

$$\mathrm{GELU}(x) = \frac{x}{2}(1 + \mathrm{erf}(\frac{x}{\sqrt{2}})) \tag{18}$$

These activation functions, shown in Figure 7a, are both smooth and ReLU-like but with small, bounded negative output ranges. Specifically, Swish has an output range of $[-0.2785, \infty]$ and GELU has an output range of $[-0.1700, \infty]$. This poses a challenge for uniform quantization as it should represent both small negative values and large positive values.

Figure 7b shows the composition of GELU and fake quantization with different symmetric ranges. If the output of GELU is quantized to [-50, 50], then all negative values will round to zero. However, if we restrict the range to [-10, 10] then two negative values can be represented. Table 12 shows the accuracy of post training quantization with GELU outputs clipped to 10 (GELU10), and then calibrated with max calibration. Just by clipping the output of GELU we can achieve the best post training quantization accuracy with a simple max calibration, exceeding the previous best activation calibration by 0.46 F1. Furthermore, this result almost matches the best QAT result of 90.67 F1.



(a) GELU and Swish

(b) Quantizing GELU by scale quantization with different ranges

Figure 7: Quantization mapping of real values to int8

Model	fp32	Max	Entropy	99.9%	99.99%	99.999%	99.9999%	Max with GELU10
BERT Large	91.01	85.92	37.40	26.18	89.59	90.20	90.10	90.66

Table 12: BERT int8 post training quantization. Comparing previous calibration results to max with GELU10.