Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks

Sambhav R. Jain [* 1] **Albert Gural** [* 2] **Michael Wu** [1] **Chris H. Dick** [1]

Abstract

We propose a method of training quantization thresholds (TQT) for uniform symmetric quantizers using standard backpropagation and gradient descent. Contrary to prior work, we show that a careful analysis of the straight-through estimator for threshold gradients allows for a natural range-precision trade-off leading to better optima. Our quantizers are constrained to use power-of-2 scale-factors and per-tensor scaling of weights and activations to make it amenable for hardware implementations. We present analytical support for the general robustness of our methods and empirically validate them on various CNNs for ImageNet classification. We are able to achieve near-floating-point accuracy on traditionally difficult networks such as MobileNets with less than 5 epochs of quantized (8-bit) retraining. Finally, we present Graffitist, a framework that enables automatic quantization of TensorFlow graphs for TQT (available at github.com/Xilinx/graffitist).

1 Introduction

Low-precision quantization (such as uniform quantization between two clipping thresholds) is an important technique enabling low-power and high-throughput DNN inference. However, this reduced precision leads to commensurate reductions in accuracy.

Retraining weights with quantization-in-the-loop is a useful technique to regain some lost accuracy. However the quantization thresholds are typically fixed after initial calibration, leading to (a) lack of ability to adapt to changing weight and activation distributions during training, and (b) calibration based on local quantization errors that is agnostic to the final network loss. We address these two issues by treating thresholds as learnable parameters, trained using standard backpropagation and gradient descent. Therefore during quantized training, (a) our thresholds can be trained along with weights simultaneously, and (b) the gradients are computed on the overall loss meaning the learned thresholds are more optimal for the network as a whole.

We propose a general method for training quantization thresholds (TQT) using accurate gradients in Section 3. With thresholds that automatically train to achieve a range-precision trade-off, this work enables hardware amenable per-tensor and power-of-2 scaling constraints with minimal loss in accuracy. We provide an easy-to-implement and fast convergence training scheme, which trains thresholds in log-domain with an adaptive optimizer. In Section 4 we present a framework for automatic quantization and retraining of TensorFlow graphs using our methods. We demonstrate that our implementation and hyperparameter recommendations are robust, through experiments in Section 5 and analytical discussion in Appendix B. Finally we present insights from TQT in Section 6.

2 Related Work

Network quantization became popular with BinaryNet (Courbariaux et al., 2016), which quantized weights and activations to +1 and -1 and trained weights using the straight-through estimator (STE) (Bengio et al., 2013). Other works looked at similar low bitwidth networks, such as XOR-Nets (Rastegari et al., 2016), ternary networks (Li et al., 2016; Zhu et al., 2016), and TTQ (Zhu et al., 2016). To achieve higher accuracies, researchers started examining higher bitwidth quantization such as in DoReFa-Net (Zhou et al., 2016), WRPN (Mishra et al., 2017), HWGQ (Cai et al., 2017), LQ-Nets (Zhang et al., 2018) and QIL (Jung et al., 2018).

More recent work in DNN quantization has focused on practical considerations for hardware implementations, with research advertising one or more of the following: uniform quantization to allow integer arithmetic, per-tensor quantization to increase homogeneity of compute requirements, power-of-2 scale factors to allow scaling with efficient bit-shifts, and symmetric quantization to avoid cross-terms

[*]Equal contribution [1]Xilinx Inc., San Jose, California, USA [2]Stanford University, Stanford, California, USA. Correspondence to: Sambhav R. Jain <sambhav@alumni.stanford.edu>, Albert Gural <agural@alumni.stanford.edu>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020. Copyright 2020 by the author(s).

with each computation arising from a zero-point (Krishnamoorthi, 2018). Work in this area includes NVIDIA's TensorRT (Migacz, 2017), Google's Quantization-Aware Training (QAT) (Jacob et al., 2017; TensorFlow, 2017a), IBM's FAQ (McKinstry et al., 2018), PACT (Choi et al., 2018), NICE (Baskin et al., 2018) and FAT (Goncharenko et al., 2018). TensorRT uses local Kullback-Leibler (KL) divergence minimization to calibrate quantization thresholds and shows good performance for traditional CNNs, but uses floating-point scale-factors and does not explore retraining. FAQ uses percentile initialization to determine clipping thresholds, but does not train them. PACT introduced the idea of training not only the weights but also the clipping parameter α for clipped ReLU using gradient descent and STE:

$$\frac{\partial y_q(x;\alpha)}{\partial \alpha} = \begin{cases} 0 & x \in (-\infty, \alpha) \\ 1 & x \in [\alpha, +\infty) \end{cases} \tag{1}$$

Both QAT and FAT support training quantization thresholds using a gradient similar to (1), likewise NICE trains a clamping parameter c_a, initialized α standard deviations from the mean of the input distribution, using a gradient similar to (1). However, we show in Section 3.5 that these formulations of clipped threshold gradients do not balance range and precision, resulting in poor 8-bit quantization performance for difficult networks such as MobileNets (Howard et al., 2017; Sandler et al., 2018) shown in Table 1.

In contrast, and independently of our work, IBM's LSQ (Esser et al., 2019) found a gradient definition that is similar to ours. However, direct comparisons of our results are not possible due to the large differences between our experiments and applications. For instance, LSQ learns the scale-factors directly, which leads to stability issues, requiring careful fine-tuning of hyperparameters and consequent retraining for 90 epochs. We address this issue in Section 3 with a gradient formulation to train log-thresholds instead, which we show in Appendix B to have better stability guarantees and faster convergence. Secondly, LSQ does not constrain scale-factors to power-of-2 and uses higher precision in the first and last layers to retain performance, incurring additional implementation complexity. Lastly, LSQ does not explore quantization on difficult networks such as MobileNets, which from our experiments are seen to benefit the most from training quantization thresholds.

3 Trained Quantization Thresholds

A simple design choice for a uniform quantizer is one that uses an affine mapping between the real domain r and the quantized domain q, such as

$$r = s \cdot (q - z) \tag{2}$$

where constants s (scale-factor) and z (zero-point) are the quantization parameters. Generally, s is a positive real number, and z is a quantized value that maps to the real zero[1].

Table 1. Comparison of MobileNet 8-bit quantization performance between Google's QAT (from Table 4 of (Krishnamoorthi, 2018)) and ours (TQT). Our quantization scheme is strictly more constrained, yet achieves better top-1 accuracy (%) on ImageNet.

Method	Precision	Quantization Scheme	Top-1
MobileNet v1 1.0 224			
QAT	FP32		70.9
	INT8	per-channel, symmetric, real scaling	70.7
	INT8	per-tensor, asymmetric, real scaling	70.0
TQT	FP32		71.1
	INT8	per-tensor, symmetric, p-of-2 scaling	**71.1**
MobileNet v2 1.0 224			
QAT	FP32		71.9
	INT8	per-channel, symmetric, real scaling	71.1
	INT8	per-tensor, asymmetric, real scaling	70.9
TQT	FP32		71.7
	INT8	per-tensor, symmetric, p-of-2 scaling	**71.8**

3.1 Quantizer Constraints

While the affine quantizer allows for a direct mapping from floating point values to integers (without the need for lookup tables), there is added cost due to special handling of zero-points and real-valued scale-factors, as illustrated in Appendix A. For efficient fixed-point implementations, we constrain our quantization scheme to use:

1. **Symmetric:** By setting $z = 0$, the affine quantizer in (2) reduces to a symmetric quantizer:

$$r = s \cdot q \tag{3}$$

   Thus we can drop the cross-terms from a matrix multiplication or convolution operation involving zero-points (see Appendix A.1).

2. **Per-tensor scaling:** All elements in a given weight or activation tensor are quantized using a single scale-factor s. While it is common practice to use per-channel scaling for networks with depthwise convolutions such as MobileNets, we find that per-tensor scaling combined with 8-bit TQT is sufficient.

3. **Power-of-2 scaling:** Scale-factors are constrained to the form $s = 2^{-f}$ (where f is an integer denoting the fractional length; f can be positive or negative). This enables scaling using simple bit-shifts without the overhead of a fixed-point multiply operation (see Appendix A.2).

[1]This formulation satisfies the domain-specific constraint that the real zero be exactly representable (Jacob et al., 2016b; 2017; Krishnamoorthi, 2018).

3.2 Linear Quantizer - Forward Pass

The quantization function $q(x; s)$ for a tensor x is parameterized only by its scale-factor s, which depends on threshold t and bit-width b of the tensor[2]. $q(x; s)$ performs quantization by applying four point-wise operations (in order): scale, round, saturate and de-quant.

Scale: Tensor elements are scaled such that the lowest power-of-2 larger than raw threshold t (i.e., $2^{\lceil log_2(t) \rceil}$, where $\lceil . \rceil$ denotes ceil[3]) is mapped to the largest value supported in the quantized domain (i.e., 2^{b-1} if signed, or 2^b if unsigned). Naturally, elements that fall out of the saturation threshold $2^{\lceil log_2(t) \rceil}$ in either direction would be clipped.

Round: The scaled tensor elements are round to nearest integers using bankers rounding (round-half-to-even) denoted by $\lfloor . \rceil$. This prevents an overall upward or downward bias which is known to impact end-to-end inference accuracy in neural networks (Jacob et al., 2017).

Saturate: Once scaled and rounded, elements in the tensor that exceed the largest supported value in the quantized domain are clipped: $\text{clip}(x; n, p) = \min(\max(x, n), p)$. Since we apply clipping to the scaled tensor, the clipping limits (n, p) are independent of the real bounds. A signed tensor is clipped to $\left(-2^{b-1}, 2^{b-1} - 1\right)$ and an unsigned tensor to $\left(0, 2^b - 1\right)$.

De-quant: The last step undoes the scaling step. Therefore, we emulate the effect of quantization while retaining the original scale of the input tensor.

Putting together the point-wise operations from above, the quantization function $q(x; s)$ can be formally written as:

$$q(x; s) := \text{clip}\left(\left\lfloor \frac{x}{s} \right\rceil; n, p\right) \cdot s, \tag{4}$$

where $n = -2^{b-1}, p = 2^{b-1} - 1$ and $s = \frac{2^{\lceil \log_2 t \rceil}}{2^{b-1}}$ for signed data; $n = 0, p = 2^b - 1$ and $s = \frac{2^{\lceil \log_2 t \rceil}}{2^b}$ for unsigned data.

3.3 Linear Quantizer - Backward Pass

To train the weights and thresholds of the quantized network with gradient descent, we derive the local gradients of our quantizer $q(x; s)$ with respect to input x and scale-factor s. We carefully use the STE to approximate gradients of round/ceil to 1, without approximating round/ceil to be identity in the backward pass. Specifically, we define $\frac{\partial}{\partial x}\lfloor x \rceil = \frac{\partial}{\partial x}\lceil x \rceil = 1$, but $\lfloor x \rceil \neq x$ and $\lceil x \rceil \neq x$.

[2]We fix b for each tensor based on the footprint of the fixed-point hardware it maps to (albeit configurable), and allow t (hence s) to be trained with backpropagation.

[3]The ceil function ensures a power-of-2 scale-factor that is initially biased in the direction of having more elements within the clipping range.

Considering the three cases of how $\lfloor \frac{x}{s} \rceil$ compares to n and p, we re-write (4) as:

$$q(x; s) := \begin{cases} \left\lfloor \frac{x}{s} \right\rceil \cdot s & \text{if } n \leq \lfloor \frac{x}{s} \rceil \leq p, \\ n \cdot s & \text{if } \lfloor \frac{x}{s} \rceil < n, \\ p \cdot s & \text{if } \lfloor \frac{x}{s} \rceil > p. \end{cases} \tag{5}$$

The local gradient with respect to scale-factor s is:

$$\nabla_s q(x; s) := \begin{cases} \left\lfloor \frac{x}{s} \right\rceil - \frac{x}{s} & \text{if } n \leq \lfloor \frac{x}{s} \rceil \leq p, \\ n & \text{if } \lfloor \frac{x}{s} \rceil < n, \\ p & \text{if } \lfloor \frac{x}{s} \rceil > p. \end{cases} \tag{6}$$

Noting that $\nabla_{(\log_2 t)} s = s \ \ln(2)$,

$$\nabla_{(\log_2 t)} q(x; s) := s \ \ln(2) \cdot \begin{cases} \left\lfloor \frac{x}{s} \right\rceil - \frac{x}{s} & \text{if } n \leq \lfloor \frac{x}{s} \rceil \leq p, \\ n & \text{if } \lfloor \frac{x}{s} \rceil < n, \\ p & \text{if } \lfloor \frac{x}{s} \rceil > p \end{cases} \tag{7}$$

The choice to train thresholds in the log-domain is simple yet effective for various stability reasons discussed in detail in Appendix B.

Similarly, the local gradient with respect to input x is:

$$\nabla_x q(x; s) := \begin{cases} 1 & \text{if } n \leq \lfloor \frac{x}{s} \rceil \leq p, \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

3.4 Interpretation of Gradients

To qualitatively understand the role of threshold gradient $\nabla_{(\log_2 t)} q(x; s)$ and input gradient $\nabla_x q(x; s)$ during back-propagation, let us consider the following toy problem: A single quantizer optimized using least-square-error loss $L = \left(q(x; s) - x\right)^2 / 2$. The overall gradients of L are:

$$\nabla_{(\log_2 t)} L = (q(x; s) - x) \cdot \nabla_{(\log_2 t)} q(x; s) \tag{9}$$

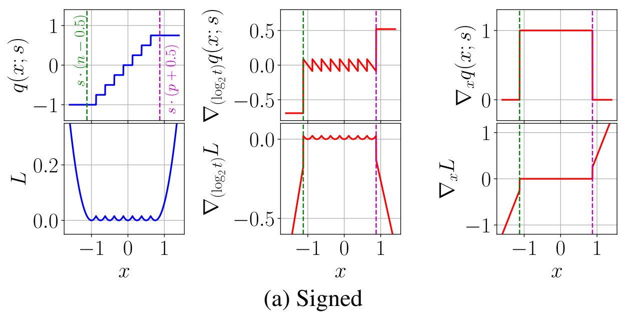
$$\nabla_x L = (q(x; s) - x) \cdot (\nabla_x q(x; s) - 1) \tag{10}$$

Figure 1 shows the forward and backward pass transfer curves for our quantizer. As noted, the exact clipping thresholds of x in the real domain are $x_n = s \cdot (n - 0.5)$ and $x_p = s \cdot (p + 0.5)$.

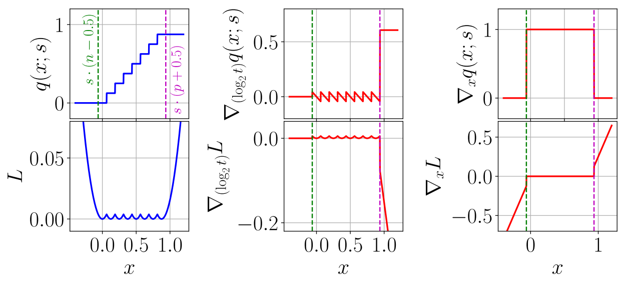
Role of threshold gradients: As seen from the plots of $\nabla_{(\log_2 t)} L$ vs. x in Figure 2, threshold gradients are positive for x within clipping thresholds (x_n, x_p) and negative otherwise. When most of the input distribution[4] falls within (x_n, x_p), the cumulative threshold gradient is positive causing $\log_2 t$ to decrease[5]. In other words, the limits (x_n, x_p) get pulled inward in favor of larger precision.

[4]Gaussian in this example, but the analysis holds in general.

[5]From the update rule $\log_2 t := \log_2 t - \alpha \nabla_{(\log_2 t)} L$ where α is the learning rate.



(a) Signed



(b) Unsigned

Figure 1. Forward pass (blue) and backward pass (red) transfer curves of our quantizer for signed and unsigned data. Local gradients shown in the top rows, and overall gradients of L_2-loss in the bottom rows. We pick bit-width $b = 3$ and raw threshold $t = 1.0$ in this example.

Similarly, when most of the input distribution falls outside (x_n, x_p), the cumulative threshold gradient is negative, $\log_2 t$ increases, and the limits (x_n, x_p) get pushed outward in favor of larger dynamic range. This technique is naturally robust to distributions with long tails or outliers, by achieving range-precision trade-off through gradient-based optimization.

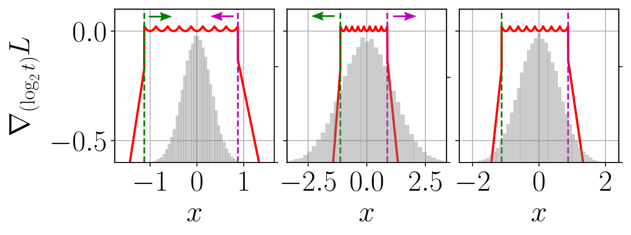


Figure 2. Trained quantization thresholds move inward (left) or outward (center) to achieve range-precision trade-off. When converged (right), the positive gradients from x within (x_n, x_p) cancel the negative gradients from x outside(x_n, x_p).

Role of Input Gradients: Using a similar analysis as for threshold gradients, we see that the input gradients $\nabla_x L$ are non-zero for values of x that fall outside (x_n, x_p), biased to keep them from getting clipped. This encourages the weight and activation distributions to be tighter.

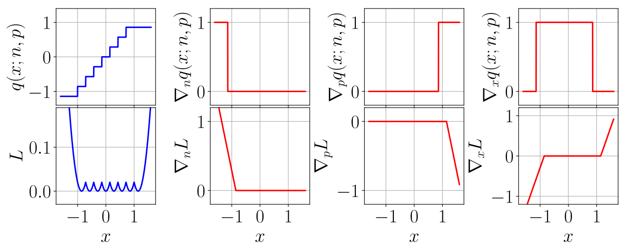


Figure 3. Forward pass (blue) and backward pass (red) transfer curves of TensorFlow's FakeQuant for signed data. Local gradients shown in the top rows, and overall gradients of L_2-loss in the bottom rows. We pick bit-width $b = 3$ and clipping thresholds $n = -1.125, p = 0.875$ to match with our example.

To summarize, threshold gradients help train optimal thresholds for clipping weights and activations, whereas input gradients nudge the weights and activations to tighter bounds. By simultaneously training clipping thresholds and weights of the quantized network through backpropagation, we adopt joint (mutual) optimization over a global loss.

3.5 Comparison to Clipped Threshold Gradients

In contrast, certain quantizer implementations define threshold gradients by simply clipping the upstream gradients at the saturation thresholds. For example TensorFlow's FakeQuant (used for QAT) defines gradients with respect to min/max thresholds as a clip function.

In the forward pass, FakeQuant operation (TensorFlow, 2016a) is mathematically equivalent to our formulation (except with zero-point), defined as:

$$q(x; n, p) := \left\lfloor \frac{\text{clip}(x; n, p) - n}{\frac{p - n}{2^b - 1}} \right\rceil \cdot \frac{p - n}{2^b - 1} + n, \quad (11)$$

However, in the backward pass they treat the round function in (11) to be identity, reducing (11) to a clip function with clipped gradients. That is, gradients with respect to thresholds (n, p) are trivially clipped to zero for x within (n, p), as seen in FakeQuant's transfer curves in Figure 3 and its kernel definition (TensorFlow, 2016b). As a result, the overall gradients only push the limits (n, p) outward, training to the min/max of the input distributions and strictly favoring range over precision. We believe this behavior can be corrected to allow effective range-precision trade-off, as seen in Figure 2 with the toy L_2 model, by carefully using the STE such that $\frac{\partial}{\partial x}\lfloor x \rceil = 1$, but $\lfloor x \rceil \neq x$ in the backward pass. While the actual loss landscape is non-trivial, we empirically observe similar qualitative behavior to our toy L_2 model, in Section 5.3.

Another popular clipping threshold method (applicable to ReLU activations) is PACT, which has similar behavior to TensorFlow's FakeQuant. As seen in (1), the gradient with respect to clipping threshold α takes a value of either 0 or 1 depending on whether the quantizer input x lies to the left or right of α. This results in a tendency of α to train to the max limits of the distribution of x. To combat this tendency, a regularizer on the magnitude of α is applied to the loss function. However, this requires an additional parameter λ_α to be tuned manually and has no awareness for the loss landscape or the quantization bitwidth.

4 Framework for TQT

We released Graffitist[6], an end-to-end software stack built on top of TensorFlow, to quantize and retrain deep neural networks (DNNs) using TQT for accurate and efficient inference on fixed-point hardware. Fundamentally, Graffitist is a flexible and scalable framework to process low-level graph descriptions of DNNs, comprising of a (growing) library of transforms to implement various neural net optimizations. Each graph transform consists of unique pattern matching and manipulation algorithms that when run sequentially produce an optimized output graph. It is still in experimental stages as we continue to add support for more operation types, layer topologies, network styles, graph optimizations, and compression techniques. Graffitist stands on the shoulders of giants and the interface is inspired in part by earlier tools from TensorFlow (TensorFlow, 2016c; 2017a).

4.1 Graph Optimizations

Graffitist applies several optimizations to the input graph prior to quantization. For example, folding batch normalization layers into preceding convolutional or fully connected or depthwise convolutional layers' weights. We adopt the following best practices from (Jacob et al., 2017; Krishnamoorthi, 2018; TensorFlow, 2017a): (a) ensure folded batch norms in training and inference graphs are mathematically equivalent (i.e., distributions seen during training match those during inference); (b) apply batch norm corrections for switching between batch and moving average statistics to reduce jitter in training folded weights due to noisy batch updates; (c) freeze batch norm moving mean and variance updates post convergence for improved accuracy. Other optimizations include collapsing concat-of-concat layers into single concat, splicing identity nodes not involved in control edges, transforming average pool layers into depthwise conv layers with reciprocal[7] multiplier as weights, and explicitly merging input scales for scale preserving ops such as concat, bias-add, eltwise-add, and maximum (for leaky relu).

[6]Available at github.com/Xilinx/graffitist.

[7]Reciprocal being $1/F^2$ where F is the kernel size.

4.2 Quantization Modes

Graffitist allows for quantization in either static or retrain modes.

Static Mode. Quantization thresholds (hence scale factors) are determined based on statistics of weights and activations derived from a calibration dataset. Specifically, weight thresholds (per-tensor) are set to the maximum absolute value (Table 2), and activation thresholds (per-tensor) are chosen such as to minimize the symmetric Kullback-Leibler-J distance (D'Alberto & Dasdan, 2009) for each quantization layer locally. This is done in a strictly topological order to ensure inputs to a layer are quantized (and fixed) prior to quantizing the current layer. The entire optimization and calibration process is automated and only requires a single API call to Graffitist.

Retrain Mode. Quantization thresholds and weights are simultaneously trained on a global loss. Recovery is achieved within 5 epochs of TQT retraining. This requires two separate API calls to Graffitist - first to generate a quantized training graph that can be trained with native TensorFlow on GPU, and second to generate an equivalent quantized inference graph that accurately models the target fixed-point implementation. The benefit of a hardware-accurate inference graph is twofold: (i) much before deployment, one can quickly validate the inference accuracy of the quantized network using CPU/GPU, and (ii) scale factors and quantized weights from TQT can be ported directly onto the target of choice. On tests across several networks, we found that our inference graphs run on the CPU were bit-accurate to our fixed-point implementation on the FPGA.

4.3 Layer Precisions

While Graffitist supports configurable bit-widths for weights and activations, for the scope of this paper we use two modes: INT8 with 8/8 (W/A) and INT4 with 4/8 (W/A). The choice of 4/8 as opposed to 4/4 is primarily guided by the availability of 4x8 multipliers; even in the absence of this, the INT4 mode still allows for 50% weight compression (double packing weights per byte) and reduced memory footprint for fetching weights. The internal precisions for different layer topologies are defined below. Quantization layers marked as q' indicate that their scale-factors are explicitly merged / shared. To avoid double quantization, input tensors are assumed to be already quantized by the previous layer, with the exception of the primary input (placeholder) which is explicitly quantized.

- Compute layers (e.g., conv, matmul, depthwise conv) are quantized as:

$$q_8\left(q'_{16}\left(\sum\left(q_{8/4}(w)\cdot q_8(x)\right)\right)+q'_{16}(b)\right),$$

where x is the input tensor, w is the weight tensor, and

b is the bias tensor. If followed by a ReLU or ReLU6 activation function, the last $q_8()$ stage is delayed to until after ReLU/ReLU6, and uses unsigned datatype to utilize the extra sign bit.

- Eltwise-add layer is quantized as:

$$q_8\left(q_8'(x) + q_8'(y)\right),$$

where x and y are the input tensors. Similar to the compute layer case, the last $q_8()$ stage is delayed and uses unsigned datatype if followed by ReLU/ReLU6.

- Leaky ReLU is quantized as:

$$q_8\left(\max\left(q_{16}'(x), q_{16}'\left(q_{16}(\alpha)\cdot q_{16}'(x)\right)\right)\right),$$

where x is the input tensor, and α is the slope of activation function for negative inputs. The last $q_8()$ stage on the previous compute layer is skipped when it is followed by Leaky ReLU. Instead a $q_{16}()$ stage is used to retain high internal precision for the α-multiply op.

- Average pool is quantized as:

$$q_8\left(\sum\left(q_8(r)\cdot q_8(x)\right)\right),$$

where x is the input tensor, and r is the reciprocal.

- Concat is not quantized because the input scales are merged explicitly, and hence it is lossless:

$$\text{concat}(q_8'(x), q_8'(y), q_8'(z)),$$

where x, y, and z are input tensors.

4.4 Fused Kernel Implementation

The quantization layer defined in (4) and (6) may be trivially implemented using native TensorFlow ops and tf.stop_gradient as depicted in Figure 4. However this low-level implementation has a large memory footprint during training due to the need for storing intermediate tensors for gradient computation in the backward pass. This impacts the maximum batch size that can fit on a single GPU. To overcome this, Graffitist is packaged with fused quantization kernels that are pre-compiled for CPU/GPU. The fused implementation is efficient, helps avoid memory overhead and allows training using larger batch sizes compared to the native implementation.

5 Experiments

We evaluate TQT on variants of five classes of CNNs trained and validated on ImageNet (ILSVRC14) classification dataset (Russakovsky et al., 2015). The networks include VGG {16, 19} (Simonyan & Zisserman, 2014), Inception v{1, 2, 3, 4} (Szegedy et al., 2014; Ioffe & Szegedy, 2015; Szegedy et al., 2015; 2016), ResNet v1 {50, 101, 152} (He et al., 2015), MobileNet v{1, 2} 1.0 224 (Howard et al., 2017; Sandler et al., 2018), and DarkNet 19 (Redmon & Farhadi, 2016). We obtained the models, pre-trained weights (FP32) and pre-processing for each of these networks from the TF-Slim model zoo (TensorFlow, 2017b) except for DarkNet 19 which was converted to TensorFlow using DW2TF (Hao & Jain, 2018).

We are interested in a scalable and production-ready approach to INT8/INT4 quantization that maps well on generic fixed-point hardware. While our simplifying constraints (from Section 3.1) may not be ideal for lower bit-widths, the fundamentals of TQT are more generally applicable even without these constraints. To limit the scope of this paper to the least-common-denominator fixed-point quantization, we do not make comparisons with other state-of-the-art low-bitwidth quantization schemes. Instead we draw comparisons of TQT (wt+th) retraining to static quantization and wt-only retraining. We can derive many interesting insights from this analysis.

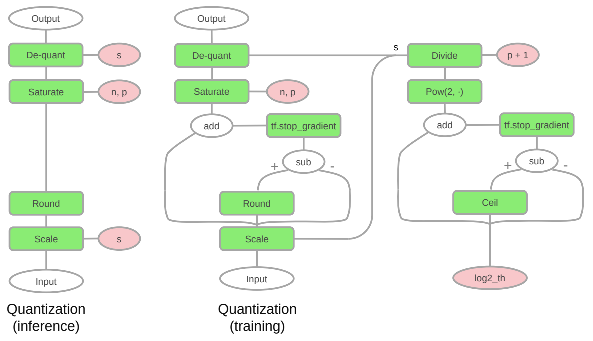


Figure 4. Illustration of the unfused quantization layer using the STE on threshold and input gradient paths. During backpropagation, the round and ceil functions are hidden by tf.stop_gradient.

Table 2. Threshold initialization scheme using MAX or 3SD initialization for weights and KL-J distance calibrated for activations.

Mode		Threshold Initialization	
		weights	activations
Static		MAX	KL-J
Retrain	wt	MAX	KL-J
	wt,th	3SD	KL-J

5.1 Threshold Initializations

Calibration sets are prepared for each network using a batch of 50 unlabeled images, randomly sampled from the validation set, with applied pre-processing. This is used for initializing the thresholds in both static and retrain modes. When thresholds are not trained, they are initialized to MAX for weights, and KL-J distance calibrated for activations.

However when training thresholds, we find it useful to initialize the weight thresholds based on n standard deviations or percentile of the weight distribution rather than MAX. Table 2 summarizes the threshold initialization scheme we used for all our experiments.

5.2 Implementation Details

Before exporting the models to TensorFlow protocol buffers (.pb) for Graffitist to absorb, we make the following synthetic modifications: (i) replace tf.reduce_mean with tf.nn.avg_pool (if any), (ii) remove auxiliary logit layers (if any), and (iii) remove dropouts (if any). Additionally, we disable data-augmentation (e.g., random flip / crop) during retraining. These modifications are done keeping in mind that TQT focuses primarily on learning thresholds through backpropagation, while allowing previously trained weights to be fine-tuned using a relatively small learning rate. As expected, most of the recovery is achieved within a fraction of an epoch due to thresholds converging, and the rest of it (up to 5 epochs) is just weights adjusting to the new thresholds. Because the overall training steps required with TQT are so few compared to from-scratch training, and that pre-trained weight distributions are not allowed to wildly change (overfit), we find it best to disable data-augmentation and dropout regularization.

Based on the stability analysis and hyperparameter recommendations in Appendix B.2 and B.3, we use the Adam optimizer with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$ for training thresholds and weights in all our experiments. The initial learning rate is set to $1e-2$ for thresholds and $1e-6$ for weights. Learning rates are decayed exponentially (with staircase enabled) by a factor of 0.94 every $3000 \cdot (24/N)$ steps for weights and by a factor of 0.5 every $1000 \cdot (24/N)$ steps for thresholds, where N is the batch size. We use a batch size of 24 for all networks except for ResNet v1 152 and Inception v4 for which a batch of 16 is used. Softmax cross-entropy loss is used to compute quantization threshold gradients and this loss, together with weight regularization (if any), are used to compute weight gradients. Batch norm moving means and variances are frozen after 1 epoch.

In Appendix B.3, we discussed the post-convergence oscillations of thresholds around the critical integer threshold $\log_2 t^*$ due to our power-of-2 scaling constraint. When thresholds cross this integer level, it can change the distributions of downstream activations, requiring weights and thresholds of the following layers to adapt to it. To minimize this effect, we incrementally freeze thresholds starting at $1000 \cdot (24/N)$ steps, once every 50 steps in the order of increasing absolute gradient magnitude, if they are on the correct side of $\log_2 t^*$ (determined using an EMA). This is automatically handled by the training scripts packaged with Graffitist.

5.3 Results

Table 3 reports the single-crop ImageNet validation accuracy for 12 networks. Default image sizes are used: 299×299 for Inception v$\{3, 4\}$, 256×256 for Darknet 19 and 224×224 for all other networks. Standard pre-processing for each network is applied to center crop, resize, and normalize the input data. The different trials include pre-trained FP32 baseline, static INT8 run, and 4 retrain runs - FP32 wt-only, INT8 wt-only, INT8 wt+th and INT4 wt+th. Here, INT8 is 8/8 (W/A) and INT4 is 4/8 (W/A). FP32 baseline numbers are reported as validated on our end. For an unbiased comparison, we train the FP32 weights using the same procedure (optimizers, learning rates, decay, BN freeze etc.) as with our quantized weight retraining. This FP32 wt-only retraining serves as a fair baseline to our INT8 and INT4 retrain results. That said, we do not use the retrained FP32 weights to initialize any of our INT8/INT4 retraining runs, and they always start from pre-trained FP32 weights. This is done to keep the overhead of retraining to a minimum.

6 Discussion

The validation accuracy and epoch count corresponding to the best checkpoint are noted in Table 3. As we see, all the networks converge within 5 epochs. Variance on the reported accuracy stems from a few sources (in decreasing order): (a) best rather than mean validation (our findings in Appendix D suggest this variance is within 0.2%), (b) non-determinism due to inexact floating point math (empirically within 0.1%), (c) round to one decimal (bound to 0.05%). Keeping these variance bounds on accuracy in mind, we can draw interesting insights into the benefits of TQT.

6.1 Insights from TQT

Our experiments demonstrate floating-point accuracy for 8-bit quantization and near-floating-point accuracy for 4-bit quantization for most networks. We see that static quantization incurs a higher loss than retrained methods. This is expected because (a) weights are not trained to adapt to the quantized network, and (b) quantization thresholds are picked using local statistics instead of being optimized on a global loss. For networks that are easier to quantize to INT8 (e.g., VGGs, Inceptions, ResNets), we find that retraining weights alone while fixing thresholds to their pre-calibrated values (based on Table 2) is sufficient. In such cases, TQT (wt+th) retraining shows no added benefit. However, for networks known to be difficult to quantize (e.g., MobileNets, DarkNets), TQT (wt+th) retraining yields up to 4% higher top-1 accuracy compared to wt-only training for INT8, and can match FP32 accuracy even with per-tensor, uniform symmetric, power-of-2 scaling constraints. This demonstrates the range-precision trade-off through trained thresholds in action. For lower precisions such as INT4,

Table 3. Quantization accuracy achieved on different ImageNet CNNs for static quantization, weight-only quantized retraining, and weight+threshold quantized retraining (TQT). Training is run until validation accuracy plateaus (max 5 epochs). We also compare to floating-point retraining to isolate the impact of our quantization methods from our training setup.

Mode		Precision	Bit-width	Accuracy (%)		Epochs
			(W/A)	top-1	top-5	
VGG 16						
		FP32	32/32	70.9	89.8	
Static		INT8	8/8	70.4	89.7	
Retrain	wt	FP32	32/32	71.9	90.5	1.0
	wt	INT8	8/8	71.8	90.5	1.0
	wt,th	INT8	8/8	71.7	90.4	0.9
	wt,th	INT4	4/8	71.5	90.3	4.0
VGG 19						
		FP32	32/32	71.0	89.8	
Static		INT8	8/8	70.4	89.7	
Retrain	wt	FP32	32/32	71.8	90.4	1.0
	wt	INT8	8/8	71.7	90.4	1.0
	wt,th	INT8	8/8	71.7	90.4	1.0
	wt,th	INT4	4/8	71.2	90.1	2.0
Inception v1						
		FP32	32/32	69.8	89.6	
Static		INT8	8/8	68.6	88.9	
Retrain	wt	FP32	32/32	70.3	90.0	2.8
	wt	INT8	8/8	70.6	90.3	3.5
	wt,th	INT8	8/8	70.7	90.2	2.4
	wt,th	INT4	4/8	67.2	88.2	4.0
Inception v2						
		FP32	32/32	74.0	91.8	
Static		INT8	8/8	73.1	91.3	
Retrain	wt	FP32	32/32	74.3	92.2	3.3
	wt	INT8	8/8	74.4	92.3	4.7
	wt,th	INT8	8/8	74.4	92.4	2.5
	wt,th	INT4	4/8	71.9	90.8	4.8
Inception v3						
		FP32	32/32	78.0	93.9	
Static		INT8	8/8	76.8	93.3	
Retrain	wt	FP32	32/32	78.3	94.2	2.1
	wt	INT8	8/8	78.2	94.1	2.0
	wt,th	INT8	8/8	78.3	94.3	1.2
	wt,th	INT4	4/8	76.4	93.1	4.4
Inception v4						
		FP32	32/32	80.2	95.2	
Static		INT8	8/8	79.4	94.6	
Retrain	wt	FP32	32/32	80.2	95.2	0.0
	wt	INT8	8/8	80.1	95.3	1.7
	wt,th	INT8	8/8	80.1	95.2	1.5
	wt,th	INT4	4/8	78.9	94.7	4.2

Mode		Precision	Bit-width	Accuracy (%)		Epochs
			(W/A)	top-1	top-5	
MobileNet v1 1.0 224						
		FP32	32/32	71.0	90.0	
Static		INT8	8/8	0.6	3.6	
Retrain	wt	FP32	32/32	71.1	90.0	3.4
	wt	INT8	8/8	67.0	87.9	4.6
	wt,th	INT8	8/8	71.1	90.0	2.1
	wt,th	INT4	4/8	–	–	
MobileNet v2 1.0 224						
		FP32	32/32	70.1	89.5	
Static		INT8	8/8	0.3	1.2	
Retrain	wt	FP32	32/32	71.7	90.7	3.2
	wt	INT8	8/8	68.2	89.0	2.7
	wt,th	INT8	8/8	71.8	90.6	2.2
	wt,th	INT4	4/8	–	–	
DarkNet 19						
		FP32	32/32	73.0	91.4	
Static		INT8	8/8	68.7	89.7	
Retrain	wt	FP32	32/32	74.4	92.3	3.1
	wt	INT8	8/8	72.9	91.6	3.8
	wt,th	INT8	8/8	74.5	92.3	1.8
	wt,th	INT4	4/8	73.2	91.6	2.8
ResNet v1 50						
		FP32	32/32	75.2	92.2	
Static		INT8	8/8	74.3	91.7	
Retrain	wt	FP32	32/32	75.4	92.5	3.7
	wt	INT8	8/8	75.3	92.3	1.0
	wt,th	INT8	8/8	75.4	92.3	1.9
	wt,th	INT4	4/8	74.4	91.7	2.0
ResNet v1 101						
		FP32	32/32	76.4	92.9	
Static		INT8	8/8	74.8	92.0	
Retrain	wt	FP32	32/32	76.6	93.2	1.2
	wt	INT8	8/8	76.3	93.0	1.0
	wt,th	INT8	8/8	76.4	93.1	0.9
	wt,th	INT4	4/8	75.7	92.5	2.0
ResNet v1 152						
		FP32	32/32	76.8	93.2	
Static		INT8	8/8	76.2	93.0	
Retrain	wt	FP32	32/32	76.8	93.3	1.0
	wt	INT8	8/8	76.7	93.3	1.5
	wt,th	INT8	8/8	76.7	93.3	1.4
	wt,th	INT4	4/8	76.0	93.0	1.9

we find that wt-only training does not recover, and so TQT (wt+th) retraining is necessary. The INT4 accuracy falls short of FP32, and we believe this maybe due to (a) our quantization constraints in Section 3.1, and (b) the first/last layers not retaining full precision[8].

[8]We quantize first/last layers to a minimum of INT8, so that they can be mapped on the same fixed-point hardware used for other layers.

6.2 MobileNet Comparisons

For more difficult networks such as MobileNets, it is well known that symmetric, per-tensor quantization done post-training or through calibrate-only methods is detrimental (Krishnamoorthi, 2018; Goncharenko et al., 2018). We believe this is true, in particular due to the use of depthwise convolutions with irregular weight distributions and widely varying ranges between channels. With wt-only retraining we are only able to recover to within 4% of floating-point accuracy. However, with TQT (wt+th) retraining, our re-

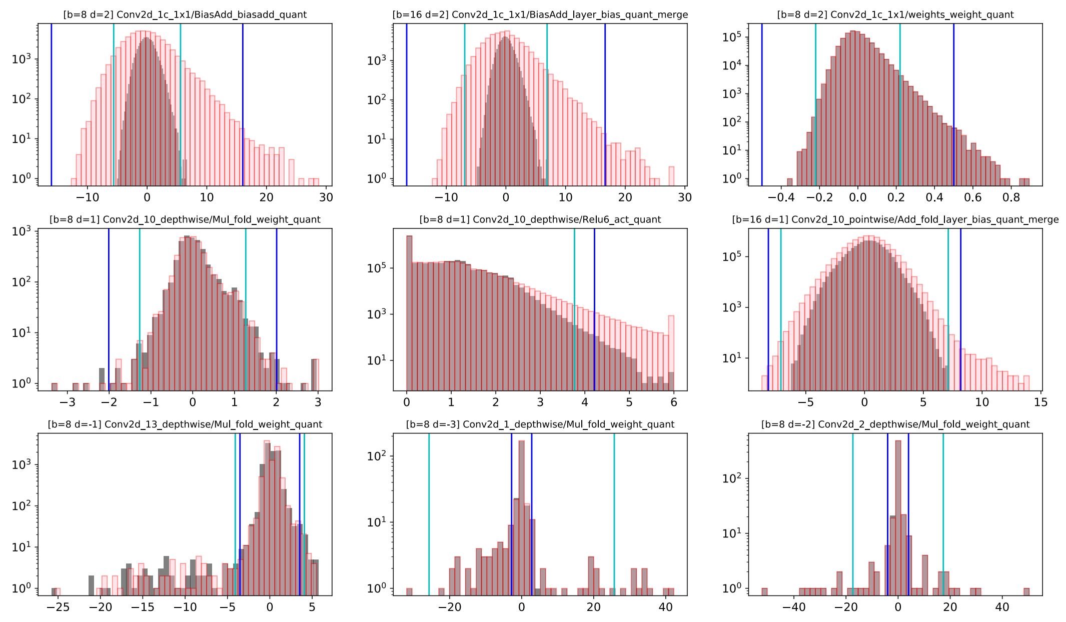
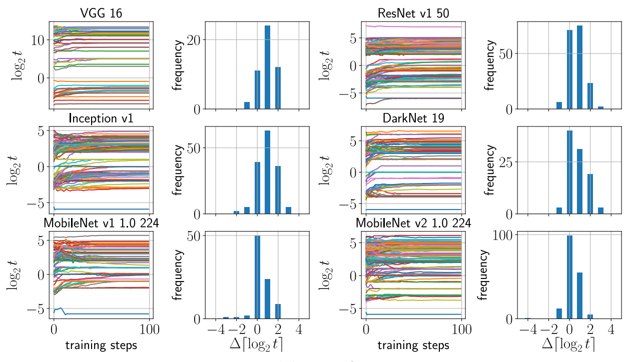
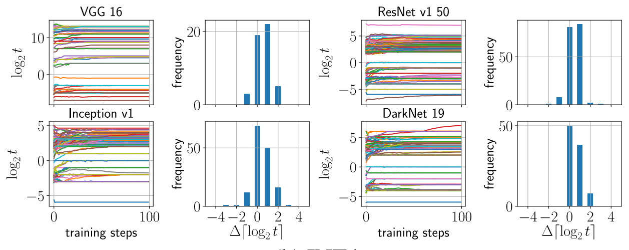


Figure 5. Selected weight and activation distributions of MobileNet v1 before (black) and after (red) quantized TQT (wt+th) retraining for thresholds that changed by non-zero integer amount in log-domain. Initial thresholds (cyan) and trained thresholds (blue) are also plotted. These are the raw thresholds t. Also indicated above each plot are bit-width b and threshold deviation $d := \Delta\lceil\log_2 t\rceil$ for the quantized layer. A positive deviation indicates preference for range over precision, and a negative deviation indicates otherwise. We note that depthwise convolutions' weights have unique threshold training behavior with a strong preference for precision compared to range.



(a) INT8



(b) INT4

Figure 6. Threshold deviations during TQT training. For each network, the left plot shows the value of each of the thresholds over the first 100 training steps, and the right plot shows a histogram of deviations from the start (initialized thresholds) to the end (trained thresholds) of training.

sults for 8-bit are the highest we have seen using symmetric, power-of-2 scaled, per-tensor quantization, even matching floating-point accuracy with no loss. We draw a few comparisons with Google's QAT results for MobileNets in Table 1 and observe that we incur no loss with INT8 quantization even with stricter constraints. We believe this is due to the fact that our threshold gradient formulation is in fact able to balance range-precision effectively.

In Figure 5 we analyze the retrained distributions for a few quantized layers in MobileNet v1, highlighting the importance of range-precision trade-off. As seen with the depthwise convolutional layers' weights, the trained thresholds move-in from their initialized values by up to 3 integer bins in the log-domain, favoring precision over dynamic range. For some other layers, the thresholds move-out from their initialized values, favoring range over precision. For more such layers with non-zero threshold deviations, see Figure 10 in Appendix.

Figure 6 shows a histogram of deviations of trained thresholds for different networks under 8-bit and 4-bit quantized retraining. We find that larger positive deviations are seen in the 8-bit case compared to the 4-bit case. This intuitively makes sense as the method decides to favor range with more bits of precision, but cuts back on range when only few bits of precision are available.

7 Conclusion

In Section 3, we proposed a general method for training quantization thresholds (TQT), amenable to most generic fixed-point hardware by constraining our method to uniform, symmetric, power-of-2 scaled, per-tensor quantization. We showed that our quantizer's gradient formulation allowed a unique range-precision trade-off, essential for high-accuracy quantized networks. We demonstrated a robust, fast convergence training scheme for TQT utilizing log-domain threshold training with an adaptive optimizer. In Section 4, we presented Graffitist, a framework for automatic quantization and retraining of TensorFlow graphs with our methods. In Section 5, we empirically validated our methods on a suite of standard CNNs trained on ImageNet. Finally, in Sections 6, we provided insightful discussions on TQT and state-of-the-art results for 8-bit MobileNet quantization.

Our work and results demonstrate the effectiveness of our techniques for high accuracy quantization of neural networks for fixed-point inference. While our work covers a major use case for quantization, there are many other quantization flavors we could explore in future work. For example, it would be useful to see how well the techniques we designed for strict power-of-2 scaling generalize to non power-of-2 scale-factors. Some additional relaxations of our constraints we could explore include per-channel rather than per-tensor quantization, which could potentially allow for more aggressive bitwidths on difficult networks like MobileNets, and non-symmetric or even non-uniform quantization schemes, where threshold training via backpropagation and gradient descent has been tried with mild success. We would not be surprised to see our methods and analysis techniques have broader applicability for more general classes of quantizers and problems beyond ImageNet.

References

Baskin, C., Liss, N., Chai, Y., Zheltonozhskii, E., Schwartz, E., Giryes, R., Mendelson, A., and Bronstein, A. M. Nice: Noise injection and clamping estimation for neural network quantization. *arXiv preprint arXiv:1810.00162*, 2018.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Cai, Z., He, X., Sun, J., and Vasconcelos, N. Deep learning with low precision by half-wave gaussian quantization. *arXiv preprint arXiv:1702.00953*, 2017.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V., and Gopalakrishnan, K. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

D'Alberto, P. and Dasdan, A. Non-parametric information-theoretic measures of one-dimensional distribution functions from continuous time series. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, pp. 685–696. SIAM, 2009.

Esser, S. K., McKinstry, J. L., Bablani, D., Appuswamy, R., and Modha, D. S. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.

Goncharenko, A., Denisov, A., Alyamkin, S., and Terentev, E. Fast adjustable threshold for uniform neural network quantization. *arXiv preprint arXiv:1812.07872*, 2018.

Hao, Y. and Jain, S. R. Darknet to tensorflow (DW2TF). `https://github.com/jinyu121/DW2TF/releases/tag/v1.2`, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Hinton, G., Srivastava, N., and Swersky, K. Lecture 6a overview of mini-batch gradient descent (2012). *Coursera Lecture slides https://class. coursera. org/neuralnets-2012-001/lecture*, 2012.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *arXiv preprint arXiv:1712.05877*, 2017.

Jacob, B. et al. Gemmlowp: Efficient handling of offsets. `https://github.com/google/gemmlowp/blob/master/doc/low-precision.md#efficient-handling-of-offsets`, 2016a.

Jacob, B. et al. Gemmlowp: Building a quantization paradigm from first principles. `https://github.com/google/gemmlowp/blob/master/doc/quantization.md`, 2016b.

Jung, S., Son, C., Lee, S., Son, J., Kwak, Y., Han, J.-J., Hwang, S. J., and Choi, C. Learning to quantize deep networks by optimizing quantization intervals with task loss. *arXiv preprint arXiv:1808.05779*, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

Li, F., Zhang, B., and Liu, B. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

McKinstry, J. L., Esser, S. K., Appuswamy, R., Bablani, D., Arthur, J. V., Yildiz, I. B., and Modha, D. S. Discovering low-precision networks close to full-precision networks for efficient embedded inference. *arXiv preprint arXiv:1809.04191*, 2018.

Migacz, S. 8-bit inference with tensorrt. In *GPU Technology Conference*, 2017.

Mishra, A., Nurvitadhi, E., Cook, J. J., and Marr, D. Wrpn: wide reduced-precision networks. *arXiv preprint arXiv:1709.01134*, 2017.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279*, 2016.

Redmon, J. and Farhadi, A. Yolo9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242*, 2016.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. *arXiv preprint arXiv:1801.04381*, 2018.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*, 2016.

TensorFlow. FakeQuant API. `https://www.tensorflow.org/versions/r1.13/api_docs/python/tf/quantization/fake_quant_with_min_max_vars`, 2016a.

TensorFlow. FakeQuant threshold gradients. `https://github.com/tensorflow/tensorflow/blob/v1.13.1/tensorflow/core/kernels/fake_quant_ops_functor.h#L179-L187`, 2016b.

TensorFlow. Graph transform tool. `https://github.com/tensorflow/tensorflow/blob/v1.13.1/tensorflow/tools/graph_transforms/README.md`, 2016c.

TensorFlow. Quantization-aware training. `https://github.com/tensorflow/tensorflow/blob/v1.13.1/tensorflow/contrib/quantize/README.md`, 2017a.

TensorFlow. Tf-slim pre-trained models. `https://github.com/tensorflow/models/blob/v1.13.0/research/slim/README.md#pre-trained-models`, 2017b.

Zhang, D., Yang, J., Ye, D., and Hua, G. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. *arXiv preprint arXiv:1807.10029*, 2018.

Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

A Cost of Affine Quantizer

A.1 Cross-terms due to zero-points

Consider two real numbers r_1 and r_2 and their product $r_3 = r_1 \cdot r_2$. Using the affine mapping from (2) to represent this, we get:

$$s_3(q_3 - z_3) = s_1(q_1 - z_1) \cdot s_2(q_2 - z_2), \tag{12}$$

which can be expressed as

$$q_3 = z_3 + \frac{s_1 s_2}{s_3}\big[q_1 q_2 - q_1 z_2 - q_2 z_1 + z_1 z_2\big]. \tag{13}$$

The cross-terms in (13) add complexity and often require special handling to remain efficient. While the added cost can be amortized over several accumulations of a matrix multiplication or convolution operation, it would still require optimizations[9], both algorithmic and kernel-level.

By eliminating zero-points, the cross-terms vanish and the operation simplifies to:

$$q_3 = \frac{s_1 s_2}{s_3}\big[q_1 q_2\big]. \tag{14}$$

A.2 Real-valued scale-factors

With positive real scale-factors, the constant multiplier $s_1 s_2 / s_3$ in (14), empirically found to be in the interval (0, 1) (Jacob et al., 2017), can be expressed in the normalized form $2^{-n} s_0$ where n is a non-negative integer and s_0 is in the interval [0.5, 1). In other words, the accumulator (storing $q_1 q_2$) needs to be scaled by a fixed-point multiplier that approximates s_0 and right-shifted by n bits (with round-to-nearest):

$$q_3 = 2^{-n} s_0 \big[q_1 q_2\big]. \tag{15}$$

However, by constraining scale-factors s_1, s_2, s_3 to strict power-of-2, the scaling operation reduces to a rather simple bit-shift (with round-to-nearest):

$$q_3 = 2^{-f}\big[q_1 q_2\big]. \tag{16}$$

B Log Threshold Training

Initially, it may seem that with the definition of a gradient with respect to the raw threshold, backpropagation and gradient descent could be immediately used to train it. However, just as training weights in a vanilla neural network requires care in the choice of optimizer and learning rate, here too care must be taken to ensure training stability and convergence. There are three main properties we would like our training procedure to satisfy: numerical stability, scale invariance, and convergence. We discuss each of these issues and the engineering tweaks used to solve them here.

[9]Some of which are covered in (Jacob et al., 2016a; 2017; Krishnamoorthi, 2018).

B.1 Numerical Stability

One obvious problem with training raw thresholds $t \in \mathbb{R}^+$ is that gradient updates could potentially bump a threshold to a negative value, causing $\log_2 t$ and therefore scale-factor s to diverge. If this happens even once, the network as a whole will break. An easy solution is to train $\log_2 t$ as opposed to t itself, since its domain is $\log_2 t \in \mathbb{R}$. Using log thresholds is convenient because it already appears in the expression for $s(t)$. However, the most important benefit is described in Section B.2, where the log representation makes ensuring scale invariance very easy.

B.2 Scale Invariance

For a given input distribution we prefer that the threshold gradients have similar magnitudes regardless of the position of the threshold itself. This *threshold scale invariance* is useful for making sure training is not too slow when the thresholds are far from their optimal values. Similarly, the properties of our threshold gradients should not depend on the scale of the input distribution. This *input scale invariance* is important because it ensures that quantized training behaves the same way for the different weights and activations in the network, even if the variance of their distributions vary over many orders of magnitude.

Unfortunately, neither of these scale invariances hold. Far from improving, Figure 7 shows that in moving from raw threshold training (left) to log threshold training (middle), both scale invariance properties of the threshold gradients actually degrade.

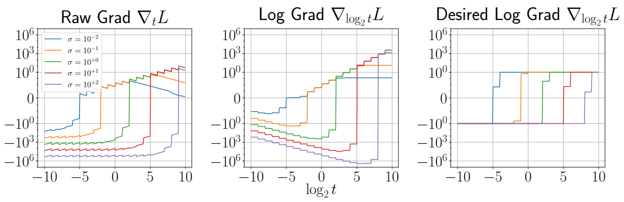


Figure 7. Gradients of L_2-loss with respect to raw threshold (left) or log threshold (middle, right) versus log threshold, for Gaussian(σ) inputs of varying σ. Desired (normed) gradients for the log threshold case are shown on the right.

Threshold scale invariance: Updates to the log threshold would be threshold scale invariant if the gradients on both sides of the negative-to-positive jump were flat, as seen in the right plot of Figure 7. However, this is not the case for log threshold gradients (center plot of Figure 7). On the left-of-jump side, as $\log_2 t$ decreases, gradients of (hence

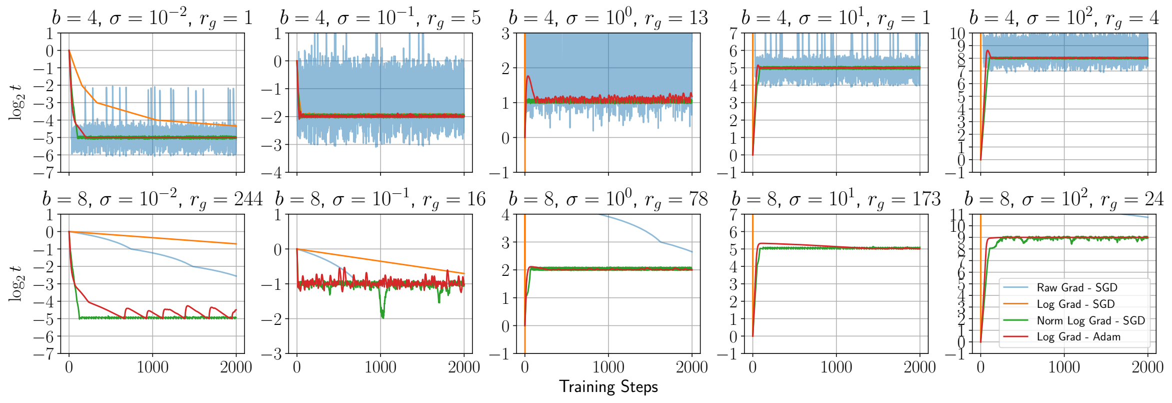


Figure 8. Raw, log and normed log threshold training on L_2-loss for 2000 steps with learning rate $\alpha = 0.1$. We compare different bit-widths - 4 (top) and 8 (bottom), and Gaussian(σ) inputs of varying σ - smallest (left) to largest (right). The empirical value of r_g is estimated from the last few hundred steps of Adam.

updates to) $\log_2 t$ get exponentially smaller, meaning it will converge very slowly to lower optimal values (see the log grad SGD case in the left plots of Figure 8). Similarly, on the right-of-jump side, as $\log_2 t$ increases, updates to $\log_2 t$ increase exponentially, meaning it will converge very quickly and possibly unstably to higher optimal values (see the log grad SGD case in the right plots of Figure 8). In the raw threshold domain, we would like gradients of (hence updates to) t to scale proportional to t. This is also not the case for the left-of-jump side of raw threshold gradients (left plot of Figure 7). In other words, the raw and log threshold gradients are swapped from what we would prefer on the left-of-jump sides.

Input scale invariance: Updates to the log threshold are input scale invariant if the gradients are threshold scale invariant and x-axis shifted copies for varying input scales, as seen in the right plot of Figure 7. However, this is not the case for log threshold gradients (center plot of Figure 7) as the gradient magnitudes depend on the scale of the input. In fact when accounting for the threshold scale dependence, the gradient magnitudes depend quadratically on the scale of the input.

Normed gradients: While neither raw or log threshold gradients have the desired properties of scale invariance, only minimal modifications to our log threshold gradient is needed to get these properties to hold (see desired log threshold gradient on the right of Figure 7). In particular, if we normalize the gradient g_i by its bias-corrected moving average variance, we achieve a close approximation of the desired gradients $\tilde{g_i}$, shown in (17). To improve stability, we can encapsulate (17) in a clipping function to guarantee no large gradients, shown in (18).

Yet another desired property highlighted in Figure 7 is that near the jump, the ratio of the gradient magnitudes to either side of the jump is to be preserved between the original and normed gradient cases. This is important for the convergence dynamics of the system discussed in Section B.3. In dynamic situations, the gradient normalization solution (17) approximates this feature as well.

$$\begin{aligned} v_i &\leftarrow \beta v_{i-1} + (1-\beta) g_i^2 \\ \hat{v_i} &\leftarrow \frac{v_i}{1-\beta^i} \\ \tilde{g_i} &\leftarrow \frac{g_i}{\sqrt{\hat{v_i}} + \epsilon} \qquad (17) \\ \tilde{g_i} &\leftarrow \tanh\left(\frac{g_i}{\sqrt{\hat{v_i}} + \epsilon}\right) \qquad (18) \end{aligned}$$

Figure 8 shows training curves on the toy L_2 quantization error problem across various bit-widths, input scales, and optimization algorithms. Raw gradient with SGD fails for large σ and converges too slowly for small σ, as we would expect from Sections B.1 and B.2. Additionally, they have b, σ-dependent stability once converged. Switching from raw to log threshold gradients, we see that log gradient with Adam performs well, yet log gradient with SGD performs poorly, with weak convergence rates for small σ and divergence for large σ. However, after performing gradient normalization (18), normed log gradient with SGD performs well, demonstrating that lack of proper gradient norming is the main issue preventing convergence using standard gradient descent. Besides the differing convergence rates, another characteristic becomes immediately obvious - stability after convergence. For example, raw gradient method tends to oscillate wildly between multiple integer-level log thresholds, whereas normed log gradient method is better behaved and tends to stay within a single integer log threshold band.

Adam optimizer: While gradient norming (18) led to good results with SGD, we note that Adam without this gradient norming also works quite well. It is easy to see why this is -

Adam has built-in gradient norming (Kingma & Ba, 2014). Thus we can avoid redefining the gradients by simply using an optimizer that includes adaptive gradients, such as Adam or RMSprop (Hinton et al., 2012). While RMSprop appears to superficially resemble (18) more closely than Adam, we suspect Adam has better behavior in the absence of gradient clipping due to its use of moments to smooth the gradients. To use Adam safely, we derive rough bounds on the learning rate and momentum parameters to ensure the oscillations seen in Figure 8 for log gradient with Adam do not exceed a single integer bin. This is important because if they move across bins often, the network may have more trouble adapting to the changing distributions from a given quantized layer, in an effect that may be similar to the motivation for batch normalization (Ioffe & Szegedy, 2015).

B.3 Convergence

One primary cause of the sharp gradient jumps seen in Figure 7 is our insistence on power-of-2 scaling. In the forward pass, features downstream from the quantized layer are completely unaware of intermediate non-power-of-2 scale-factors so there are sharp jumps at integral $\log_2 t$, similar to what might be observed when using the STE for traditional quantization. The net effect is a bang-bang like operation.

In more detail, for a given input distribution there is some critical integer threshold $\log_2 t^*$ before which the gradients are negative (causing positive threshold updates) and after which the gradients are positive. This negative feedback will force the threshold to oscillate around $\log_2 t^*$. The gradients g_l and g_h on either side of $\log_2 t^*$ tend to be fairly constant within a distance 1 of $\log_2 t^*$ due to power-of-2 scaling. For simplicity, assume $|g_l| > |g_h|$ so that the ratio $r_g = -g_l/g_h > 1$. As r_g grows, we would expect the following behavior: the threshold stays in the higher bin for a while, slowly decaying until reaching the lower bin, at which point a large $|g_l|$ causes it to jump back to the higher bin, where it begins a slow decay again. This behavior can be observed in the left plots of Figure 8 and are shown in more detail in Figure 9.

If normed log gradients and SGD are used together, the dynamics are fairly simple. Let $\log_2 t_i \leftarrow \log_2 t_{i-1} - \alpha\tilde{g}_i$ be the SGD update on normed log gradient $\tilde{g}_i$ (18). Then because $|\tilde{g}_i| \leq 1$ by design, a given jump in the sawtooth-like pattern will have magnitude bounded by learning rate α. Thus by selecting $\alpha \ll 1$, we can ensure convergence within a threshold bin.

However in our experiments, we used the implementationally simpler approach of unnormed log gradients with the Adam optimizer. While simpler to implement, the analysis is more complicated due to the second-order nature of the optimizer. Adam has three key hyperparameters: α, β_1, β_2 and operates by keeping track of a moving mean of gradients $m_i \leftarrow \beta_1 m_{i-1} + (1-\beta_1)g_i$ and a moving variance $v_i \leftarrow \beta_1 v_{i-1} + (1-\beta_1)g_i^2$ before applying update rule $\theta_i \leftarrow \theta_{i-1} - \alpha \cdot m_i/\sqrt{v_i}$. In practice, bias correction is used to get $\hat{m}_i, \hat{v}_i$, but when considering settling dynamics for $i \to \infty$, this bias correction is insignificant. Typical values are $\alpha \approx 10^{-3}, \beta_1 \approx 0.9, \beta_2 \approx 0.999$.

In Appendix C, a detailed analysis of convergence for Adam is carried out. From this analysis a simple set of guidelines emerge. First, the learning rate is set to guarantee $\alpha < 0.1/\sqrt{p}$. Next, we ensure $1/e < \beta_1 < 1$ to satisfy the limits of our analysis. Finally, we make sure $r_g \approx p \ll 1/(1-\beta_2) \Rightarrow 1-\beta_2 \ll 1/p$. These results are summarized in Table 4. For simplicity, we use $\alpha = 0.01, \beta_1 = 0.9, \beta_2 = 0.999$ for all of our training.

Table 4. Guidelines for log threshold training with Adam, assuming $b = 2^{b-1} - 1$ for signed data.

Bit-width	b	4	8
α	$\leq \frac{0.1}{\sqrt{2^{b-1}-1}}$	≤ 0.035	≤ 0.009
β_1	$\geq 1/e$	$\geq 1/e$	$\geq 1/e$
β_2	$\geq 1 - \frac{0.1}{2^{b-1}-1}$	≥ 0.99	≥ 0.999
Steps	$\approx \alpha^{-1} + (1-\beta_2)^{-1}$	≈ 100	≈ 1000

C Analysis of Adam Convergence

Let T be the period of oscillations at convergence. If we assume $T \ll 1/(1-\beta_2)$, then we can treat the moving variance estimate as if it is a constant $v_i = ((T-1)g_h^2 + g_l^2)/T \approx g_l^2(1/r_g^2 + 1/T)$. However, we cannot make the same assumption for the relationship between T and β_1. Instead, based on our earlier discussion in Section B.3 of the bang-bang behavior, we assume that a gradient g_l is seen for a single step, then g_h is seen for $T-1$ steps. Then for a given cycle of this behavior, $m_i = \beta_1^i(\beta_1 m_0 + (1-\beta_1)g_l) + (1-\beta_1^i)g_h$, where m_0 is the steady-state minimum mean during the cycle. Because this is steady-state, we can solve for m_0 and m_i:

$$m_i = \beta_1^i(\beta_1 m_0 + (1-\beta_1)g_l) + (1-\beta_1^i)g_h$$

$$m_T = m_0 = \beta_1^T(\beta_1 m_0 + (1-\beta_1)g_l) + (1-\beta_1^T)g_h$$

$$m_0 = \frac{\beta_1^T(1-\beta_1) - (1-\beta_1^T)/r_g}{1-\beta_1^{T+1}} g_l \tag{19}$$

$$\frac{m_i}{g_l} = \beta_1^{i+1}\frac{\beta_1^T(1-\beta_1) - (1-\beta_1^T)/r_g}{1-\beta_1^{T+1}} + \beta_1^i(1-\beta_1+\frac{1}{r_g}) - \frac{1}{r_g} \tag{20}$$

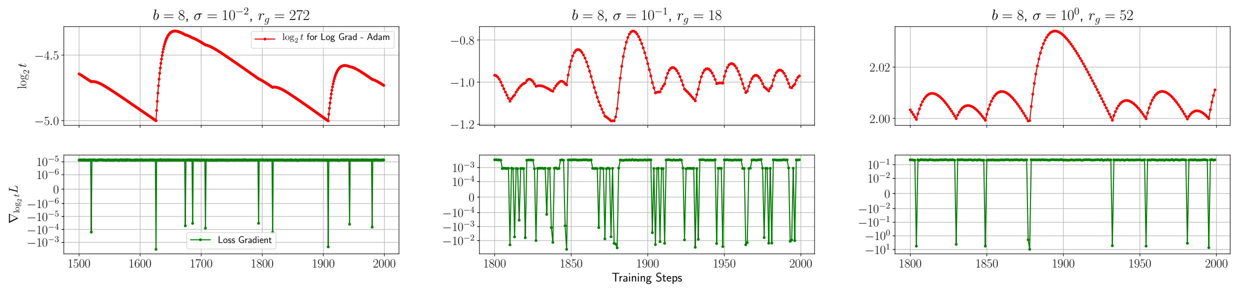


Figure 9. Close up of Figure 8 for the Adam-trained log threshold gradient on a few select settings.

Adam updates look like $\theta_i \leftarrow \theta_{i-1} - \alpha \cdot m_i/\sqrt{v_i}$ or $\theta_i \leftarrow \theta_0 - \alpha \sum_{j=0}^{i} m_j/\sqrt{v_j}$. We can solve for T by finding when $\theta_T = \theta_0$ or $\sum_{i=0}^{T} m_i/\sqrt{v_i} = 0$. As an intermediate step, we find:

$$\begin{aligned}\Delta_t\theta &= \sum_{i=0}^{t} \frac{m_i}{\sqrt{v_i}}\\ &= \sum_{i=0}^{t} \frac{\beta_1^i\left(\beta_1 \frac{\beta_1^T(1-\beta_1)-(1-\beta_1^T)/r_g}{1-\beta_1^{T+1}} + 1 - \beta_1 + \frac{1}{r_g}\right) - \frac{1}{r_g}}{\sqrt{1/r_g^2 + 1/T}}\\ &= \frac{1}{\sqrt{\frac{1}{r_g^2}+\frac{1}{T}}}\left[\frac{1-\beta_1^{t+1}}{1-\beta_1}\left(\beta_1 \frac{\beta_1^T(1-\beta_1)-(1-\beta_1^T)/r_g}{1-\beta_1^{T+1}}\right.\right.\\ &\left.\left.+1-\beta_1+\frac{1}{r_g}\right) - \frac{t+1}{r_g}\right] \qquad (21)\end{aligned}$$

Now, we set $\Delta_T\theta = 0$:

$$\begin{aligned}0 &= \frac{1}{\sqrt{\frac{1}{r_g^2}+\frac{1}{T}}}\left[\frac{1-\beta_1^{T+1}}{1-\beta_1}\left(\beta_1 \frac{\beta_1^T(1-\beta_1)-(1-\beta_1^T)/r_g}{1-\beta_1^{T+1}}\right.\right.\\ &\left.\left.+1-\beta_1+\frac{1}{r_g}\right) - \frac{T+1}{r_g}\right]\\ &= \beta_1^{T+1} - \frac{\beta_1(1-\beta_1^T)}{r_g(1-\beta_1)} + 1 - \beta_1^{T+1} + \frac{1-\beta_1^{T+1}}{r_g(1-\beta_1)} - \frac{T+1}{r_g}\\ T &= r_g \qquad (22)\end{aligned}$$

The worst case happens when r_g is large, so if we substitute $T \leftarrow r_g$ and assume $r_g \gg 1$, we get:

$$\begin{aligned}\Delta_t\theta &\approx \sqrt{r_g}\left[\frac{1-\beta_1^{t+1}}{1-\beta_1}\left(\beta_1 \frac{\beta_1^{r_g}(1-\beta_1)-(1-\beta_1^{r_g})/r_g}{1-\beta_1^{r_g+1}}\right.\right.\\ &\left.\left.+1-\beta_1+\frac{1}{r_g}\right) - \frac{t+1}{r_g}\right] \qquad (23)\\ &= \sqrt{r_g}\left[\frac{1-\beta_1^{t+1}}{1-\beta_1}c_1 - \frac{t+1}{r_g}\right] \qquad (24)\end{aligned}$$

where we replace the large expression in (23) with c_1 in (24). We now solve for the critical point of $\Delta_t\theta$ to determine $t_{max} = \text{argmax}_t \Delta_t\theta$.

$$\begin{aligned}0 &= \frac{d}{dt}\Delta_t\theta\\ &= \sqrt{r_g}\left[\frac{\ln(\beta_1^{-1})\beta_1^{t_{max}+1}}{1-\beta_1}c_1 - \frac{1}{r_g}\right]\\ \beta_1^{t_{max}+1} &= \frac{1}{\ln(\beta_1^{-1})}\frac{1-\beta_1}{r_g \cdot c_1} \qquad (25)\\ &= \frac{1}{\ln(\beta_1^{-1})}\frac{1-\beta_1^{r_g+1}}{1+r_g}\\ t_{max} &= \log_{\beta_1}\left(\frac{1}{\ln(\beta_1^{-1})}\frac{1-\beta_1^{r_g+1}}{1+r_g}\right) - 1 \qquad (26)\end{aligned}$$

Plugging (25) and (26) into (24),

$$\begin{aligned}\Delta_{t_{max}}\theta \approx \sqrt{r_g}&\left[\frac{c_1}{1-\beta_1} - \frac{1}{r_g\ln(\beta_1^{-1})}\right.\\ &\left.-\frac{1}{r_g}\log_{\beta_1}\left(\frac{1}{\ln(\beta_1^{-1})}\frac{1-\beta_1^{r_g+1}}{1+r_g}\right)\right] \qquad (27)\end{aligned}$$

To simplify this expression, note that $\beta_1 < 1$ and $r_g \gg 1$ so $1-\beta_1^{r_g} \approx 1$. Then $c_1/(1-\beta_1) \approx 1 + 1/r_g \approx 1$ and:

$$\Delta_{t_{max}}\theta \approx \sqrt{r_g}\left[1 + \frac{1+\ln(r_g \ln\beta_1^{-1})}{r_g\ln\beta_1}\right] \qquad (28)$$

Further, if $1/e < \beta_1 < 1$, then the right term is negative and the expression has a simple upper bound:

$$\Delta_{t_{max}}\theta < \sqrt{r_g} \tag{29}$$

In practice, we notice that sometimes noise can can cause θ to stay on the high-gradient side of the threshold boundary for multiple steps, causing the momentum to build up. Thus, to be safe, we recommend designing for $\Delta_{t_{max}}\theta < 10\sqrt{r_g}$.

A rough estimate for the number of steps needed for convergence is $\mathcal{O}(\Delta\lceil\log_2 t\rceil/(\alpha|\tilde{g}|))$. Because of adaptive gradients, $|\tilde{g}|$ should be close to 1, provided we allow enough time for historical variance to decay - $\mathcal{O}(1/(1-\beta_2))$ steps[10]. Thus, the overall number of steps would be $\mathcal{O}(\Delta\lceil\log_2 t\rceil/\alpha + \Delta\lceil\log_2 t\rceil/(1-\beta_2))$. Assuming calibration is used, $\Delta\lceil\log_2 t\rceil$ should be close to 1, giving the simplified expression $\mathcal{O}(1/\alpha + 1/(1-\beta_2))$ steps.

Finally, we address how to approximate r_g. The operation of crossing a threshold boundary moves some fraction f of inputs $\{x_i\}$ from the $n \le \lfloor x/s \rceil \le p$ case to the $\lfloor x/s \rceil < n$ or $\lfloor x/s \rceil > p$ cases (assume only $\lfloor x/s \rceil > p$ for simplicity from here on). Using the toy L_2-loss model (9),

$$\nabla_{(\log_2 t)} L = s^2 \ln 2 \cdot \begin{cases} \left(\left\lfloor \frac{x}{s} \right\rceil - \frac{x}{s}\right)^2 & \text{if } n \le \left\lfloor \frac{x}{s} \right\rceil \le p, \\ n(n - x/s) & \text{if } \left\lfloor \frac{x}{s} \right\rceil < n, \\ p(p - x/s) & \text{if } \left\lfloor \frac{x}{s} \right\rceil > p \end{cases} \tag{30}$$

we see that for any given x_i, the ratio r_{gi} between the gradients in the outer and inner cases is $p(p - x_i/s)/(\lfloor x_i/s \rceil - x_i/s)^2$. But since x_i recently switched cases, $(p - x_i/s) < 1$. As a rough estimate, we might expect $r_{gi} \approx (1/2p)/(1/12) \approx 6p$. Averaged over the entire input, $r_g \approx 6fp \lessapprox p$. The $10\times$ over-design helps address some uncertainty in this measure as well.

Figure 9 shows a re-run of Figure 8 for the case of Adam optimization on log threshold gradients. These plots allow us to validate our Adam convergence analysis above. First we note that $p = 2^{8-1} - 1 = 127$, which is an approximate upper bound on r_g and well within the $10\times$ over-design principle. Next, notice that $T \approx r_g$. For example, in the $\sigma = 10^{-2}$ case, $T \approx 280$ while $r_g \approx 272$.

Most importantly, we expect the max log-threshold deviation to be upper-bounded by $\alpha\sqrt{r_g} = (1.6, 0.4, 0.7)$ from left to right if our original assumptions hold - that we visit the lower threshold bin for one step and stay in the upper bin for $T - 1$ steps. While the bound holds for all σ, it is close to not holding for $\sigma = 10^{-1}$. A brief inspection reveals why this is the case - the log threshold spends far more than one step in the lower threshold bin per period, violating our one-step assumption. This violation can be explained by looking at the gradients, which show that the lower threshold bin sometimes has positive gradients, depending on the randomness of the input Gaussian vector. These phenomena motivate our suggestion to over-design by $10\times$. The cost in additional steps needed to reach convergence seems like a worthwhile trade-off.

[10]This is a problem when historical gradient magnitudes were higher, as is usually the case when $\Delta\lceil\log_2 t\rceil < 0$, as seen in the small σ plots of Figure 8.

D Best or Mean Validation

We run validation every 1000 training steps and save the best top-1 score checkpoint. This approach was initially driven by a desire to better understand convergence and stability properties with our method, but we continued using it since intermediate validation was not too expensive for 5 epochs of retraining. However a valid concern is that this intermediate validation introduces a positive bias to our results through cherry-picking. To quantify this, we compare the positive-biased validation method to simply taking the average of validation scores at fixed intervals: 20%, 40%, 60%, 80% and 100% of the fifth epoch. As noted in Table 5, the differences between these methods on the top-1 accuracy are 0.1% and 0.2% for MobileNet v1 and VGG 16 respectively, suggesting that cherry-picking only results in a minor positive bias on our reported accuracy.

Table 5. Best validation (cherry-picked) is compared to the average of five validations (at pre-determined steps) in the last epoch, for two networks.

	Accuracy (%)		Epochs
	top-1	top-5	
MobileNet v1 1.0 224			
	70.982	89.886	4.2
	70.986	89.860	4.4
	71.076	89.930	4.6
	71.000	89.870	4.8
	71.022	89.944	5.0
Mean	**71.0**	**89.9**	
Best	**71.1**	**90.0**	2.1
VGG 16			
	71.448	90.438	4.2
	71.462	90.456	4.4
	71.434	90.436	4.6
	71.500	90.426	4.8
	71.458	90.456	5.0
Mean	**71.5**	**90.4**	
Best	**71.7**	**90.4**	0.9

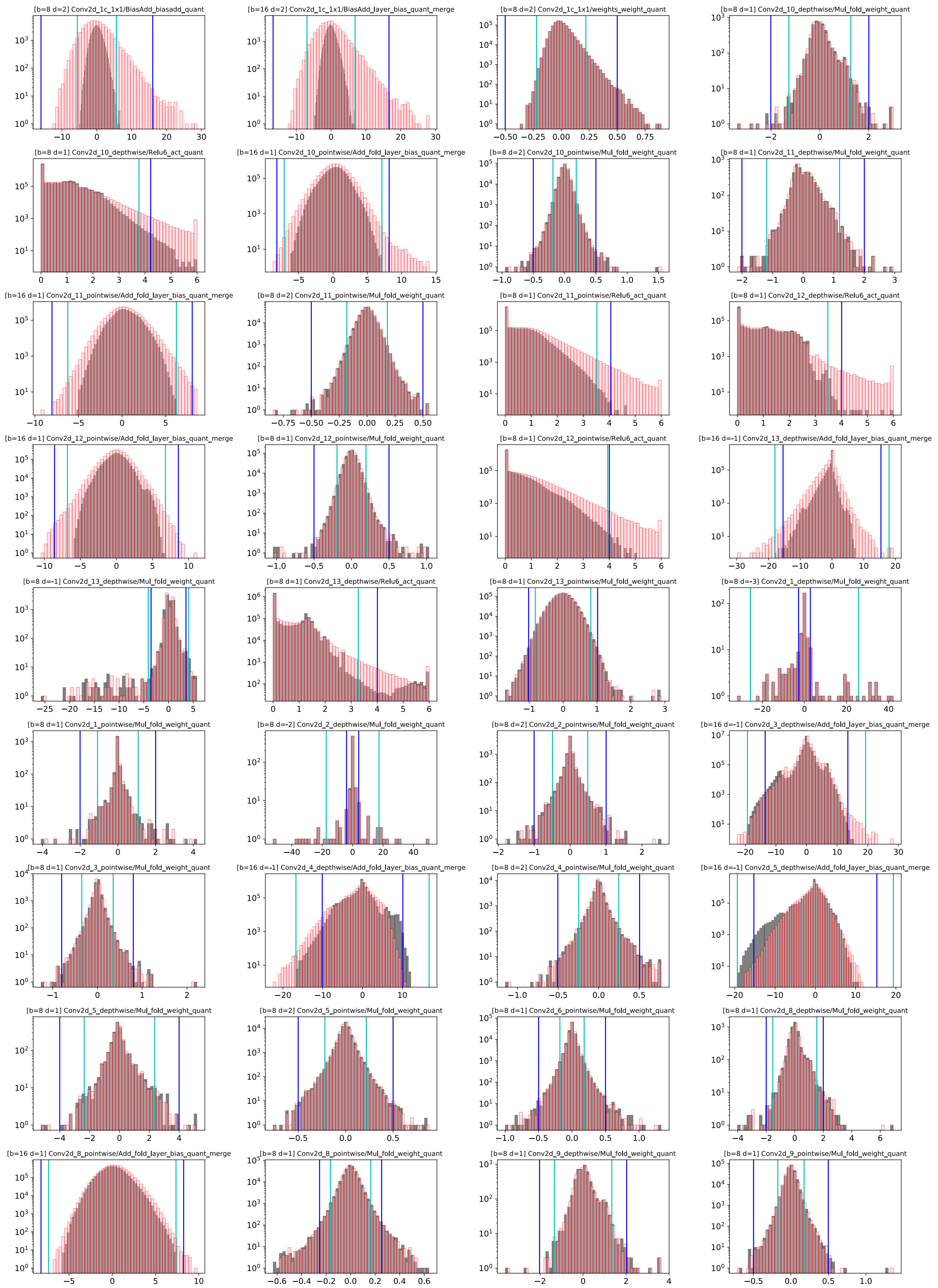


Figure 10. Weight and activation distributions of MobileNet v1 before (black) and after (red) quantized TQT (wt+th) retraining for thresholds that changed by non-zero integer amount in log-domain. Initial thresholds (cyan) and trained thresholds (blue) are also plotted. These are the raw thresholds t. Also indicated above each plot are bit-width b and threshold deviation $d := \Delta\lceil\log_2 t\rceil$ for the quantized layer. A positive deviation indicates preference for range over precision, and a negative deviation indicates otherwise.