Quantizing deep convolutional networks for efficient inference: A whitepaper

Raghuraman Krishnamoorthi
raghuramank@google.com

June 2018

Contents

Abstract

We present an overview of techniques for quantizing convolutional neural networks for inference with integer weights and activations.

1. Per-channel quantization of weights and per-layer quantization of activations to 8-bits of precision post-training produces classification accuracies within 2% of floating point networks for a wide variety of CNN architectures (section 3.1).
2. Model sizes can be reduced by a factor of 4 by quantizing weights to 8-bits, even when 8-bit arithmetic is not supported. This can be achieved with simple, post training quantization of weights (section 3.1).
3. We benchmark latencies of quantized networks on CPUs and DSPs and observe a speedup of 2x-3x for quantized implementations compared to floating point on CPUs. Speedups of up to 10x are observed on specialized processors with fixed point SIMD capabilities, like the Qualcomm QDSPs with HVX (section 6).
4. Quantization-aware training can provide further improvements, reducing the gap to floating point to 1% at 8-bit precision. Quantization-aware training also allows for reducing the precision of weights to four bits with accuracy losses ranging from 2% to 10%, with higher accuracy drop for smaller networks (section 3.2).
5. We introduce tools in TensorFlow and TensorFlowLite for quantizing convolutional networks (Section 3).
6. We review best practices for quantization-aware training to obtain high accuracy with quantized weights and activations (section 4).
7. We recommend that per-channel quantization of weights and per-layer quantization of activations be the preferred quantization scheme for hardware acceleration and kernel optimization. We also propose that future processors and hardware accelerators for optimized inference support precisions of 4, 8 and 16 bits (section 7).

1 Introduction

Deep networks are increasingly used for applications at the edge. Devices at the edge typically have lower compute capabilities and are constrained in memory and power consumption. It is also necessary to reduce the amount of communication to the cloud for transferring models to the device to save on power and reduce network connectivity requirements. Therefore, there is a pressing need for techniques to optimize models for reduced model size, faster inference and lower power consumption.

There is extensive research on this topic with several approaches being considered: One approach is to build efficient models from the ground up [1],[2] and [3]. Another technique is to reduce the model size by applying quantization, pruning and compression techniques [4], [5] and [6]. Faster inference has been achieved by having efficient kernels for computation in reduced precision like GEMMLOWP [7], Intel MKL-DNN [8] , ARM CMSIS [9], Qualcomm SNPE [10], Nvidia TensorRT [11] and custom hardware for fast inference [12], [13] and [14].

One of the simpler ways to reduce complexity of any model is to reduce the precision requirements for the weights and activations. This approach has many advantages:

- It is broadly applicable across a range of models and use cases. One does not need to develop a new model architecture for improved speed. In many cases, one can start with an existing floating point model and quickly quantize it to obtain a fixed point quantized model with almost no accuracy loss, without needing to re-train the model. Multiple hardware platforms and libraries support fast inference with quantized weights and activations, so there is no need to wait for new hardware development.
- Smaller Model footprint: With 8-bit quantization, one can reduce the model size a factor of 4, with negligible accuracy loss. This can be done without needing any data as only the weights are quantized. This also leads to faster download times for model updates.
- Less working memory and cache for activations: Intermediate computations are typically stored in cache for reuse by later layers of a deep network and reducing the precision at which this data is stored leads to less working memory needed. Having lower precision weights and activations allows for better cache reuse.
- Faster computation: Most processors allow for faster processing of 8-bit data.
- Lower Power: Moving 8-bit data is 4 times more efficient than moving 32-bit floating point data. In many deep architectures, memory access can dominate power consumption [2]. Therefore reduction in amount of data movement can have a significant impact on the power consumption.

All the factors above translate into faster inference, with a typical speedup of 2-3x due to the reduced precision for both memory accesses and computations. Further improvements in speed and power consumption are possible with processors and hardware accelerators optimized for low precision vector arithmetic.

2 Quantizer Design

In this section, we review different design choices for uniform quantization.

2.1 Uniform Affine Quantizer

Consider a floating point variable with range (x_{min}, x_{max}) that needs to be quantized to the range $(0, N_{levels} - 1)$ where $N_{levels} = 256$ for 8-bits of precision. We derive two parameters: Scale (Δ) and Zero-point(z) which map the floating point values to integers (See [15]). The scale specifies the step size of the quantizer and floating point zero maps to zero-point [4]. Zero-point is an integer, ensuring that zero is quantized with no error. This is important to ensure that common operations like zero padding do not cause quantization error.

For one sided distributions, therefore, the range (x_{min}, x_{max}) is relaxed to include zero. For example, a floating point variable with the range (2.1,3.5) will be relaxed to the range (0,3.5) and then quantized. Note that this can cause a loss of precision in the case of extreme one-sided distributions.

Once the scale and zero-point are defined, quantization proceeds as follows:

$$x_{int} = round(\frac{x}{\Delta}) + z \tag{1}$$

$$x_Q = clamp(0, N_{levels} - 1, x_{int}) \tag{2}$$

where

$$\begin{aligned} clamp(a, b, x) &= a & x \leq a \\ &= x & a \leq x \leq b \\ &= b & x \geq b \end{aligned}$$

The de-quantization operation is:

$$x_{float} = (x_Q - z)\Delta \tag{3}$$

While the uniform affine quantizer allows for storing weights and activations at 8-bits of precision, there is an additional cost due to the zero-point. Consider a 2D convolution between a weight and an activation:

$$y(k, l, n) = \Delta_w \Delta_x conv(w_Q(k, l, m; n) - z_w, x_Q(k, l, m) - z_x) \tag{4}$$

$$y(k, l, n) = conv(w_Q(k, l, m; n), x_Q(k, l, m)) - z_w \sum_{k=0}^{K-1} \sum_{l=0}^{K-1} \sum_{m=0}^{N-1} x_Q(k, l, m) \tag{5}$$

$$- z_x \sum_{k=0}^{K-1} \sum_{l=0}^{K-1} \sum_{m=0}^{N-1} w_Q(k, l, m; n) + z_x z_w \tag{6}$$

A naive implementation of convolution, by performing the addition of zero-point prior to the convolution, leads to a 2x to 4x reduction in the throughput due to wider (16/32-bit) operands. One can do better by using the equation above and noting that the last term is a constant and each of the other terms requires N multiplies, which is 3x more operations than the 8-bit dot product. This can be further improved by noting that the weights are constant at inference and by noting that the sum over activations is identical for all convolutional kernels of the same size. However, this requires optimizing convolution kernels. For an indepth discussion, please see [16].

2.2 Uniform symmetric quantizer

A simplified version of the affine quantizer is the symmetric quantizer, which restricts zero-point to 0. With the symmetric quantizer, the conversion operations simplify to:

$$x_{int} = round(\frac{x}{\Delta}) \tag{7}$$

$$x_Q = clamp(-N_{levels}/2, N_{levels}/2 - 1, x_{int}) \quad \text{if signed} \tag{8}$$

$$x_Q = clamp(0, N_{levels} - 1, x_{int}) \quad \text{if un-signed} \tag{9}$$

For faster SIMD implementation, we further restrict the ranges of the weights. In this case, the clamping is modified to:

$$x_Q = clamp(-(N_{levels}/2 - 1), N_{levels}/2 - 1, x_{int}) \qquad \text{if signed} \qquad (10)$$
$$x_Q = clamp(0, N_{levels} - 2, x_{int}) \qquad \text{if un-signed} \qquad (11)$$

Please see [4], Appendix B for more details.

The de-quantization operation is:

$$x_{out} = x_Q \Delta$$

2.3 Stochastic quantizer

Stochastic quantization models the quantizer as an additive noise, followed by rounding. The stochastic quantizer is given by:

$$x_{int} = round(\frac{x + \epsilon}{\Delta}) + z, \qquad \epsilon \sim Unif(-\frac{1}{2}, \frac{1}{2})$$
$$x_Q = clamp(0, N_{levels} - 1, x_{int})$$

The de-quantization operation is given by equation 3. Note that in expectation, the stochastic quantizer reduces to a pass-through of the floating point weights, with saturation for values outside the range. Therefore, this function is well behaved for purposes of calculating gradients. We do not consider stochastic quantization for inference as most inference hardware does not support it.

2.4 Modeling simulated quantization in the backward pass

For Quantization-aware training, we model the effect of quantization using simulated quantization operations, which consist of a quantizer followed by a de-quantizer, i.e,

$$x_{out} = SimQuant(x) \qquad (12)$$
$$= \Delta\, clamp\big(0, N_{levels} - 1, round(\frac{x}{\Delta}) - z\big) \qquad (13)$$

Since the derivative of a simulated uniform quantizer function is zero almost everywhere, approximations are required to model a quantizer in the backward pass. An approximation that has worked well in practice (see [5]) is to model the quantizer as specified in equation 14 for purposes of defining its derivative (See figure 1).

$$x_{out} = clamp(x_{min}, x_{max}, x) \qquad (14)$$

The backward pass is modeled as a "straight through estimator" (see [5]). Specifically,

$$\delta_{out} = \delta_{in} I_{x \in S} S : x : x_{min} \le x \le x_{max} \qquad (15)$$

where $\delta_{in} = \frac{\partial L}{\partial w_{out}}$ is the backpropagation error of the loss with respect to the simulated quantizer output.

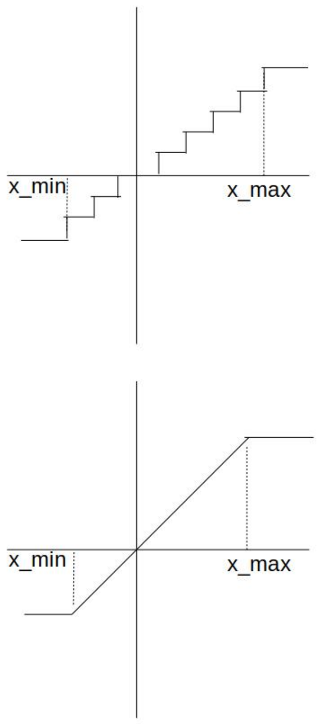


Figure 1: Simulated Quantizer (top), showing the quantization of output values. Approximation for purposes of derivative calculation (bottom).

2.5 Determining Quantizer parameters

The quantizer parameters can be determined using several criteria. For example, TensorRT [11] minimizes the KL divergence between the original and quantized distributions to determine the step size. In this work, we adopt simpler methods. For weights, we use the actual minimum and maximum values to determine the quantizer parameters. For activations, we use the moving average of the minimum and maximum values across batches to determine the quantizer parameters. For post training quantization approaches, one can improve the accuracy of quantized models by careful selection of quantizer parameters.

2.6 Granularity of quantization

We can specify a single quantizer (defined by the scale and zero-point) for an entire tensor, referred to as per-layer quantization. Improved accuracy can be obtained by adapting the quantizer parameters to each kernel within the tensor [17]. For example, the weight tensor is 4 dimensional and is a collection of 3 dimensional convolutional kernels, each responsible for producing one output feature map. per-channel quantization has a different scale and offset for each convolutional kernel. We do not consider per-channel quantization for activations as this would complicate the inner product computations at the core of conv and matmul operations. Both per-layer and per-channel quantization allow for efficient dot product and convolution implementation as the quantizer parameters are fixed per kernel in both cases.

3 Quantized Inference: Performance and Accuracy

Quantizing a model can provide multiple benefits as discussed in section 1. We discuss multiple approaches for model quantization and show the performance impact for each of these approaches.

3.1 Post Training Quantization

In many cases, it is desirable to reduce the model size by compressing weights and/or quantize both weights and activations for faster inference, without requiring to re-train the model. Post Training quantization techniques are simpler to use and allow for quantization with limited data. In this section, we study different quantization schemes for weight only quantization and for quantization of both weights and activations. We show that per-channel quantization with asymmetric ranges produces accuracies close to floating point across a wide range of networks.

3.1.1 Weight only quantization

A simple approach is to only reduce the precision of the weights of the network to 8-bits from float. Since only the weights are quantized, this can be done without requiring any validation data (See figure 2). A simple command line tool can convert the weights from float to 8-bit precision. This setup is useful if one only wants to reduce the model

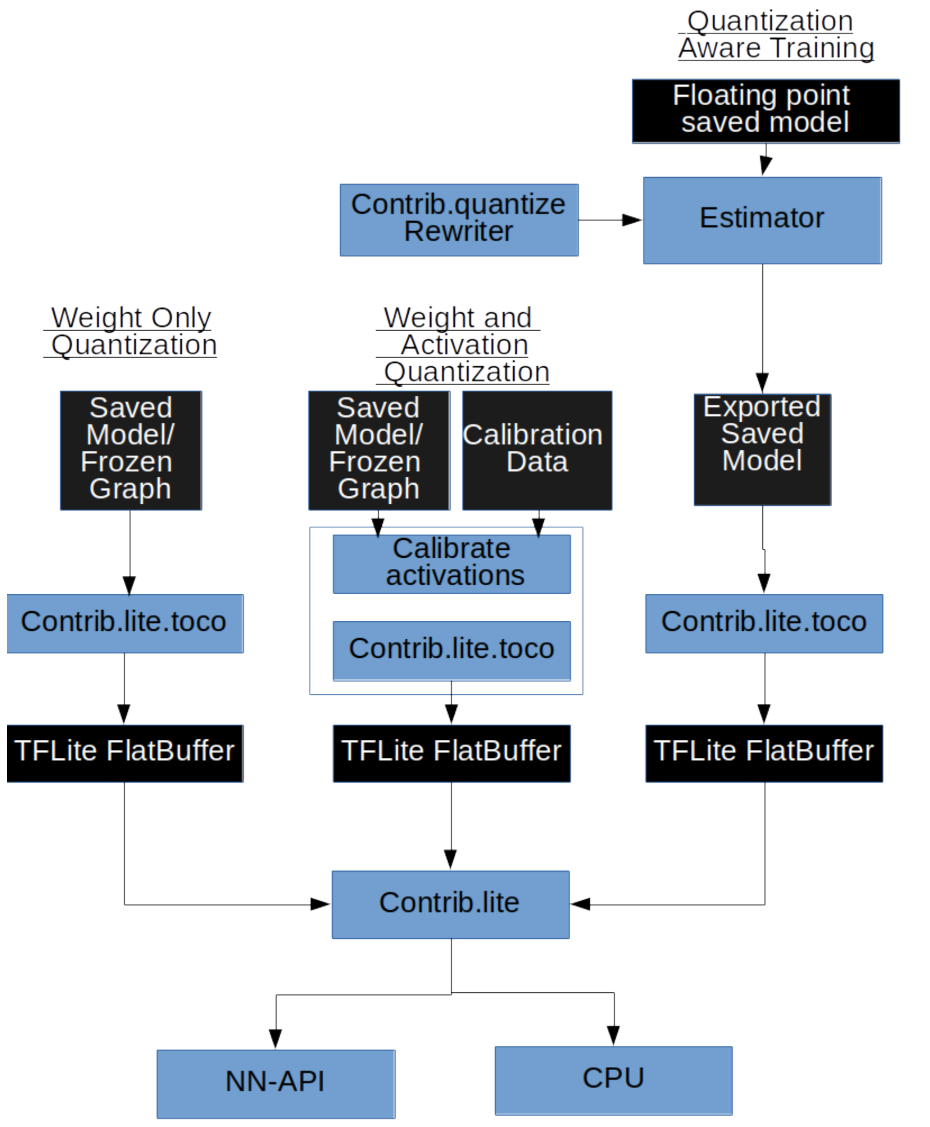


Figure 2: Overview of schemes for model quantization: One can quantize weights post training (left) or quantize weights and activations post training (middle). It is also possible to perform quantization aware training for improved accuracy

size for transmission and storage and does not mind the cost of performing inference in floating point.

3.1.2 Quantizing weights and activations

One can quantize a floating point model to 8-bit precision by calculating the quantizer parameters for all the quantities to be quantized. Since activations need to be quantized, one needs calibration data and needs to calculate the dynamic ranges of activations. (See figure 2) Typically, about 100 mini-batches are sufficient for the estimates of the ranges of the activation to converge.

3.1.3 Experiments

For evaluating the tradeoffs with different quantization schemes, we study the following popular networks and evaluate the top-1 classification accuracy. Table 1 shows the wide variation in model size and accuracy across these networks. We note that Mobilenet-v1 [2] and Mobilenet-v2[1] architectures use separable depthwise and pointwise convolutions with Mobilenet-v2 also using skip connections. Inception-v3 [18] and NasNet [19] use network in network building blocks with NasNet determining the architecture via reinforcement learning techniques. Resnets [20] pioneered the idea of skip connections and consist of multiple blocks each making residual corrections to the main path with no transformations. Resnet-v2 [21] is an enhancement to the resnet architecture using pre-activation layers for improved accuracy. Note that all results are obtained using simulated quantization of weights and activations.

Network	Model Parameters	Top-1 Accuracy on ImageNet (fp32)
Mobilenet_V1_0.25_128	0.47M	0.415
Mobilenet_V2_1_224	3.54M	0.719
Mobilenet_V1_1_224	4.25M	0.709
Nasnet_Mobile	5.3M	0.74
Mobilenet_V2_1.4_224	6.06M	0.749
Inception_V3	23.9M	0.78
Resnet_v1_50	25.6M	0.752
Resnet_v2_50	25.6M	0.756
Resnet_v1_152	60.4M	0.768
Resnet_v2_152	60.4M	0.778

Table 1: Deep Convolutional networks: Model size and accuracy

Weight only quantization: We first quantize only the weights post training and leave the activations un-quantized. From figure 2, we note that per-channel quantization is required to ensure that the accuracy drop due to quantization is small, with asymmetric, per-layer quantization providing the best accuracy.

Network	Asymmetric, per-layer	Symmetric , per-channel	Asymmetric, per-channel	Floating Point
Mobilenetv1_1_224	0.001	0.591	0.704	0.709
Mobilenetv2_1_224	0.001	0.698	0.698	0.719
NasnetMobile	0.722	0.721	0.74	0.74
Mobilenetv2_1.4_224	0.004	0.74	0.74	0.749
Inceptionv3	0.78	0.78	0.78	0.78
Resnet_v1_50	0.75	0.751	0.752	0.752
Resnet_v2_50	0.75	0.75	0.75	0.756
Resnet_v1_152	0.766	0.763	0.762	0.768
Resnet_v2_152	0.761	0.76	0.77	0.778

Table 2: Weight only quantization: per-channel quantization provides good accuracy, with asymmetric quantization providing close to floating point accuracy.

Weight and Activation Quantization: Next, we quantize weights and activations to 8-bits, with per-layer quantization for activations. For weights we consider both symmetric and asymmetric quantizers at granularities of both a layer and a channel. We first show results for Mobilenetv1 networks and then tabulate results across a broader range of networks.

We also compare the post training quantization accuracies of popular convolutional networks: Inception-V3, Mobilenet-V2, Resnet-v1-50, Resnet-v1-152, Resnet-v2-50, Resnet-v2-152 and Nasnet-mobile on ImageNet in figure 4.

Network	Asymmetric, per-layer	Symmetric , per-channel	Asymmetric, per-channel	Activation Only	Floating Point
Mobilenet-v1_1_224	0.001	0.591	0.703	0.708	0.709
Mobilenet-v2_1_224	0.001	0.698	0.697	0.7	0.719
Nasnet-Mobile	0.722	0.721	0.74	0.74	0.74
Mobilenet-v2_1.4_224	0.004	0.74	0.74	0.742	0.749
Inception-v3	0.78	0.78	0.78	0.78	0.78
Resnet-v1_50	0.75	0.751	0.751	0.751	0.752
Resnet-v2_50	0.75	0.75	0.75	0.75	0.756
Resnet-v1_152	0.766	0.762	0.767	0.761	0.768
Resnet-v2_152	0.761	0.76	0.76	0.76	0.778

Table 3: Post training quantization of weights and activations: per-channel quantization of weights and per-layer quantization of activations works well for all the networks considered, with asymmetric quantization providing slightly better accuracies.

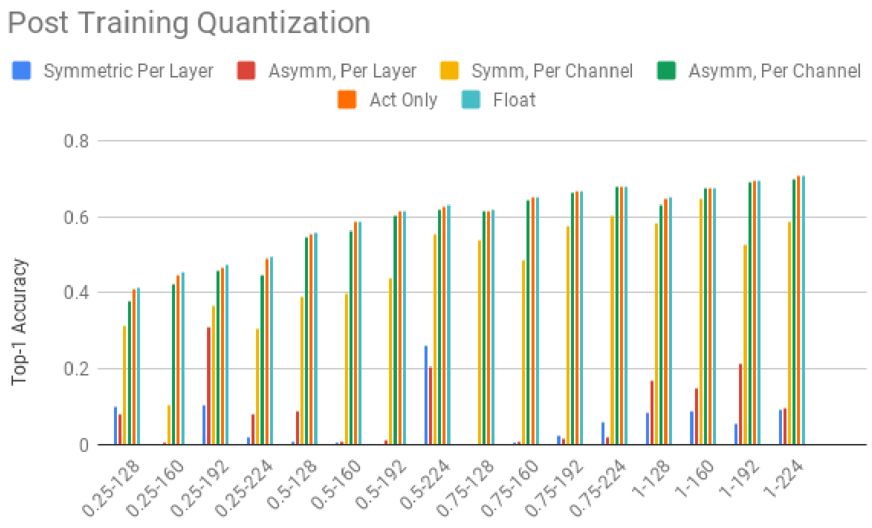


Figure 3: Comparison of post training weight and activation quantization schemes:Mobilenet-v1

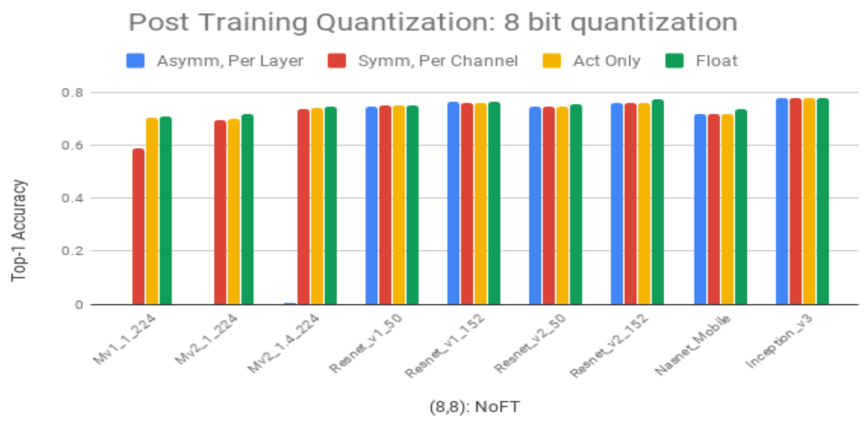


Figure 4: Comparison of post training quantization schemes

We make the following observations:

1. Per-channel quantization can provide good accuracy and can be a good baseline for post training quantization of weights and activations, with asymmetric quantization providing close to floating point accuracy for all networks.

2. Activations can be quantized to 8-bits with almost no loss in accuracy. The dynamic ranges of the activations are low due to a combination of:

   (a) Batch normalization with no scaling: Used in Inception-V3, which ensures that the activations of all feature maps have zero mean and unit variance.

   (b) ReLU6: Used in Mobilenet-V1, which restricts the activations to be in a fixed range (0,6) for all feature maps, thereby removing large dynamic range variations.

3. Networks with more parameters like Resnets and Inception-v3 are more robust to quantization compared to Mobilenets which have fewer parameters.

4. There is a large drop when weights are quantized at the granularity of a layer, particularly for Mobilenet architectures.

5. Almost all the accuracy loss due to quantization is due to weight quantization.

Weight quantization at the granularity of a layer causes large accuracy drops primarily due to batch normalization, which causes extreme variation in dynamic range across convolution kernels in a single layer. Appendix A has more details on the impact of batch normalization. Per-channel quantization side-steps this problem by quantizing at the granularity of a kernel, which makes the accuracy of per-channel quantization independent of the batch-norm scaling. However, the activations are still quantized with per-layer symmetric quantization.

Note that other approaches like weight regularization can also improve the accuracy of quantization post training, please see [22].

3.2 Quantization Aware Training

Quantization aware training models quantization during training and can provide higher accuracies than post quantization training schemes. In this section, we describe how quantization is modeled during training and describe how this can be easily done using automatic quantization tools in TensorFlow. We also evaluate the accuracies obtained for different quantization schemes with quantization aware training and show that even per-layer quantization schemes show high accuracies post training at 8-bits of precision. We also show that at 4 bit precision, quantization aware training provides significant improvements over post training quantization schemes.

We model the effect of quantization using simulated quantization operations on both weights and activations. For the backward pass, we use the straight through estimator (see section 2.4) to model quantization. Note that we use simulated quantized weights and activations for both forward and backward pass calculations. However, we maintain weights in floating point and update them with the gradient updates. This ensures that minor gradient updates gradually update the weights instead of underflowing. The updated weights are quantized and used for subsequent forward and backward pass computation. For SGD, the updates are given by:

$$w_{float} = w_{float} - \eta \frac{\partial L}{\partial w_{out}} . I_{w_{out} \in (w_{min}, w_{max})} \tag{16}$$

$$w_{out} = SimQuant(w_{float}) \tag{17}$$

Quantization aware training is achieved by automatically inserting simulated quantization operations in the graph at both training and inference times using the quantization library at [23] for Tensorflow [24]. We follow the approach outlined in [4] closely, with additional enhancements on handling batch normalization and in modeling quantization in the backward pass. A simple one-line change to the training or evaluation code automatically inserts simulated quantization operations into the training or eval graph.

For training, the code snippet is:

```
# Build forward pass of model.
...
loss = tf.losses.get_total_loss()

# Call the training rewrite which rewrites the graph in-place
# with FakeQuantization nodes and folds batchnorm for training.
# One can either fine tune an existing floating point model
# or train from scratch. quant_delay controls the onset
# of quantized training.

tf.contrib.quantize.create_training_graph(quant_delay=2000000)

# Call backward pass optimizer as usual.
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
optimizer.minimize(loss)
```

For evaluation, the code snippet is given below:

```
# Build eval model
...
logits, end_points = network_model(inputs,...)

# Call the eval rewrite which rewrites the graph in-place
# with FakeQuantization nodes and fold batchnorm for eval.
tf.contrib.quantize.create_eval_graph()
```

The high level conversion process is shown in figure 2.
The steps involved in training a quantized model are:

1. (Recommended): Fine tune from a floating point saved model: Start with a floating point pre-trained model or alternately train from scratch

2. Modify Estimator to add quantization operations: Add fake quantization operations to the model using the quantization rewriter at tf.contrib.quantize

3. Train model: At the end of this process, we have a savedmodel with quantization information (scale, zero-point) for all the quantities of interest. (weights and activations)

4. Convert model: The savedmodel with range information is transformed into a flatbuffer file using the tensorflow converter (TOCO) at: tf.contrib.lite.toco_convert. This step creates a flatbuffer file that converts the weights into integers and also contains information for quantized arithmetic with activations

5. Execute model: The converted model with integer weights can now be executed using the TFLite interpreter which can optionally execute the model in custom accelerators using the NN-API. One can also run the model on the CPU.

A simple example showing the graph transformation for a convolutional layer is shown in figure 5.

3.2.1 Operation Transformations for Quantization

It is important to ensure that all quantization related artifacts are faithfully modeled at training time. This can make trivial operations like addition, figure 6 and concatenation , figure 7 non-trivial due to the need to rescale the fixed point values so that addition/concatenation can occur correctly.

In addition, it is important to ensure that fusion of operations at inference time is modeled correctly during training. For example, consider an add followed by a ReLU operation. In this case, one can fuse the addition and the ReLU operation at inference time in most platforms. To match this, fake quantization operations should not be placed between the addition and the ReLU operations.

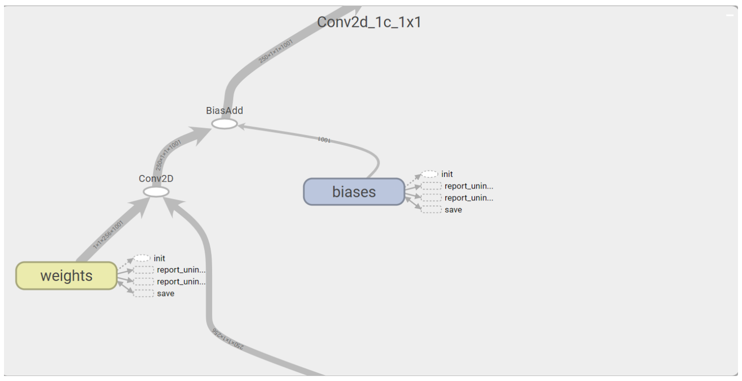
3.2.2 Batch Normalization

In this section, we describe several strategies for quantizing batch normalization layers. In section 4 and show that batch normalization with correction and freezing provides the best accuracy.

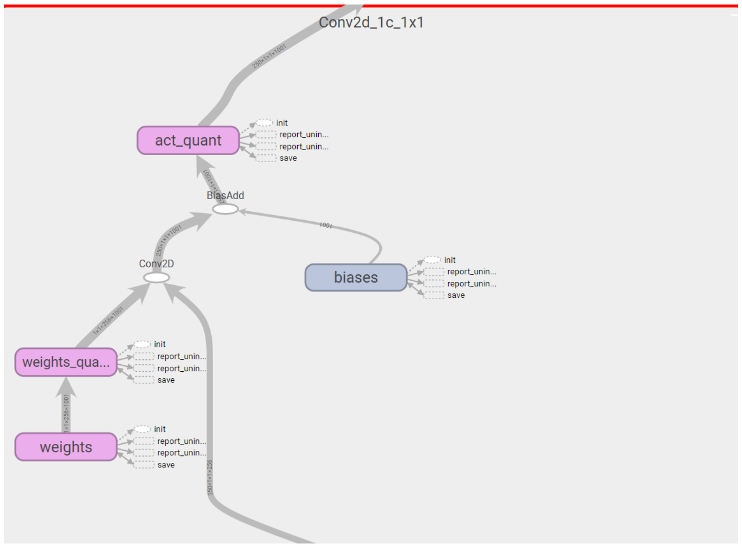
Batch normalization [25], is a popular technique that normalizes the activation statistics at the output of every layer, reducing dependencies across layers while significantly improving model accuracy.

Batch normalization is defined by the following equations:

$$x_{bn} = \gamma \left(\frac{x - \mu_B}{\sigma_B} \right) + \beta \tag{18}$$



(a) Floating Point



(b) Fixed Point

Figure 5: Convolutional layer: Before and After Graph Transformation

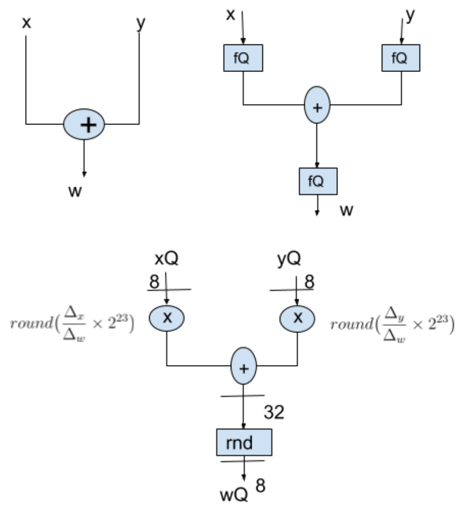

x
y
w
x
y
fQ
fQ
+
fQ
w
xQ
yQ
8
8
$round(\frac{\Delta_x}{\Delta_w} \times 2^{23})$
$round(\frac{\Delta_y}{\Delta_w} \times 2^{23})$
x
x
+
32
rnd
wQ
8

Figure 6: Fixed point transformation of element-wise add

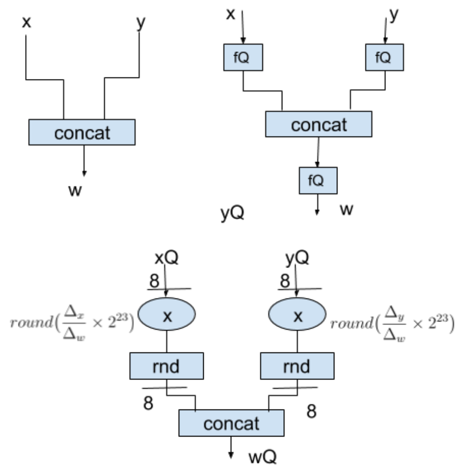

x
y
concat
w
x
y
fQ
fQ
concat
fQ
w
yQ
xQ
yQ
8
8
$round(\frac{\Delta_x}{\Delta_w} \times 2^{23})$
$round(\frac{\Delta_y}{\Delta_w} \times 2^{23})$
x
x
rnd
rnd
8
8
concat
wQ

Figure 7: Fixed point transformation of concat

for training and

$$x_{bn} = \gamma \left(\frac{x - \mu}{\sigma} \right) + \beta \tag{19}$$

for inference.

Where μ_B and σ_B are the batch mean and standard deviations. μ and σ are the long term mean and standard deviations and are computed as moving averages of batch statistic during training.

For inference, we fold the batch normalization into the weights as defined by equations 20 and 21. Therefore, at inference there is no explicit batch normalization. The weights and biases are modified to account for batch normalization instead:

$$W_{inf} = \frac{\gamma W}{\sigma} \tag{20}$$

$$Bias_{inf} = \beta - \frac{\gamma \mu}{\sigma} \tag{21}$$

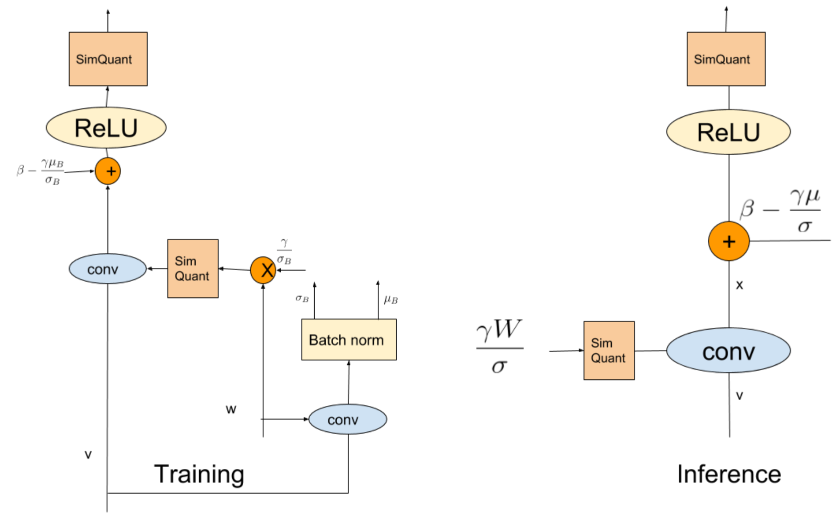

SimQuant
ReLU
$\beta - \frac{\gamma \mu_B}{\sigma_B}$
+
conv
Sim Quant
X
$\frac{\gamma}{\sigma_B}$
σ_B
μ_B
Batch norm
w
conv
v
Training
SimQuant
ReLU
$\beta - \frac{\gamma \mu}{\sigma}$
+
x
$\frac{\gamma W}{\sigma}$
Sim Quant
conv
v
Inference

Figure 8: Baseline approach for folding batch norms for quantized inference

For quantized inference, we first consider folding batch norms and training them as shown in figure 8. We note that batch normalization uses batch statistics during training, but uses long term statistics during inference. Since the batch statistics vary every batch, this introduces undesired jitter in the quantized weights and degrades the accuracy of quantized models. (green curve in 14 and 15) A simple solution would be to switch to using long term moving averages during training, however, this eliminates batch normalization (i.e the mean and variance used do not correspond to the batch statistics) and causes instability in training. The graph rewriter implements a solution that eliminates the mismatch between training and inference with batch normalization (see figure 9):

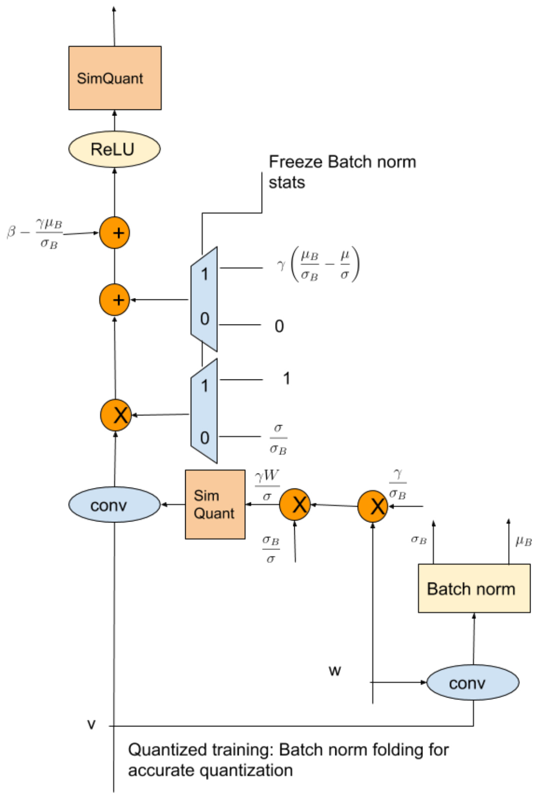

SimQuant
ReLU
Freeze Batch norm stats
$\beta - \frac{\gamma\mu_B}{\sigma_B}$
+
+
1
0
$\gamma\left(\frac{\mu_B}{\sigma_B} - \frac{\mu}{\sigma}\right)$
0
1
0
1
$\frac{\sigma}{\sigma_B}$
X
conv
Sim Quant
$\frac{\gamma W}{\sigma}$
X
X
$\frac{\gamma}{\sigma_B}$
$\frac{\sigma_B}{\sigma}$
σ_B
μ_B
Batch norm
w
conv
v
Quantized training: Batch norm folding for accurate quantization

Figure 9: Folding Batch normalization layers for improved performance

1. We always scale the weights with a correction factor to the long term statistics prior to quantization. This ensures that there is no jitter in the quantized weights

due to batch to batch variation.

$$c = \frac{\sigma_B}{\sigma} \tag{22}$$

$$w_{corrected} = c \times \frac{\gamma W}{\sigma_B} \tag{23}$$

2. During the initial phase of training, we undo the scaling of the weights so that outputs are identical to regular batch normalization. We also modify the bias terms correspondingly.

$$y = conv(Q(w_{corrected}), x) \tag{24}$$

$$y_{corrected} = y/c \tag{25}$$

$$bias = \beta - \gamma\mu_B/\sigma_B \tag{26}$$

$$bias_{corrected} = 0 \tag{27}$$

3. After sufficient training, switch from using batch statistics to long term moving averages for batch normalization, using the optional parameter freeze_bn_delay in `create_experimental_training_graph()` (about 300000 steps in figure 15 and 200000 in figure 14). Note that the long term averages are frozen to avoid instability in training. This corresponds to the normalization parameters used at inference and provides stable performance.

$$y = conv(Q(w_{corrected}), x) \tag{28}$$

$$y_{corrected} = y \tag{29}$$

$$bias = \beta - \gamma\mu_B/\sigma_B \tag{30}$$

$$bias_{correction} = \gamma(\mu_B/\sigma_B - \mu/\sigma) \tag{31}$$

3.2.3 Experiments

Quantization aware training closes the gap to floating point accuracy, even for per-layer quantization of weights. We repeat the same experiments for quantized weights and activations with training, starting from a floating point check-point and with batch normalization freezing and obtain the results shown in figures 11 and 10 and Table 4.

All the experiments have the following settings:

- Fine tune from a floating point checkpoint, we used the models in [26].
- Use Stochastic Gradient Descent for fine tuning, with a step size of 1e-5.

We note that:

1. Training closes the gap between symmetric and asymmetric quantization.
2. Training allows for simpler quantization schemes to provide close to floating point accuracy. Even per-layer quantization shows close to floating point accuracy (see column 4 in Table 4)

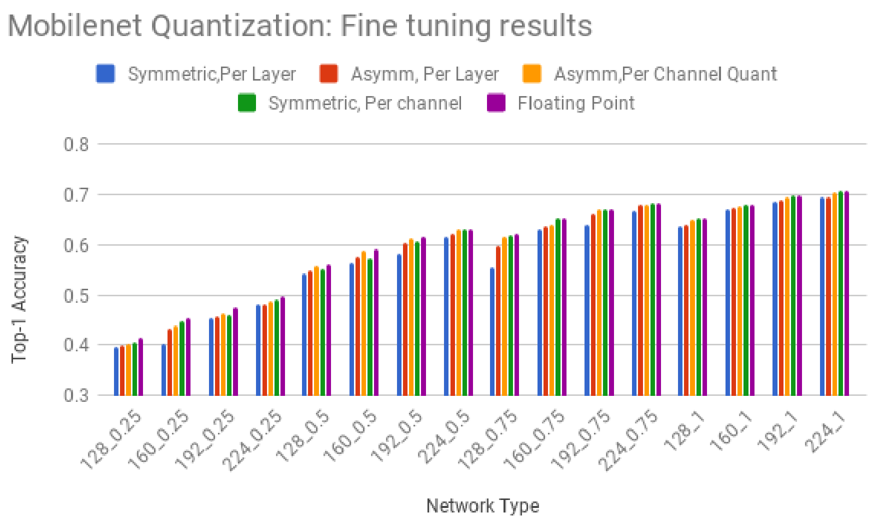


Figure 10: Comparison of quantization-aware training schemes:Mobilenet-v1

Network	Asymmetric, per-layer (Post Training Quantization)	Symmetric , per-channel (Post Training Quantization)	Asymmetric, per-layer (Quantization Aware Training)	Symmetric, per-channel (Quantization Aware Training)	Floating Point
Mobilenet-v1_1_224	0.001	0.591	0.70	0.707	0.709
Mobilenet-v2_1_224	0.001	0.698	0.709	0.711	0.719
Nasnet-Mobile	0.722	0.721	0.73	0.73	0.74
Mobilenet-v2_1.4_224	0.004	0.74	0.735	0.745	0.749
Inception-v3	0.78	0.78	0.78	0.78	0.78
Resnet-v1_50	0.75	0.751	0.75	0.75	0.752
Resnet-v2_50	0.75	0.75	0.75	0.75	0.756
Resnet-v1_152	0.766	0.762	0.765	0.762	0.768
Resnet-v2_152	0.761	0.76	0.76	0.76	0.778

Table 4: Quantization aware training provides the best accuracy and allows for simpler quantization schemes

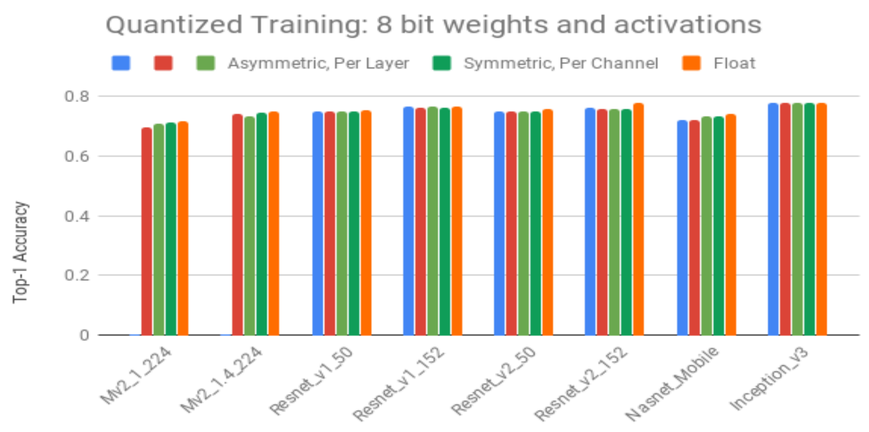


Figure 11: Comparison of quantization-aware training schemes

3.2.4 Lower Precision Networks

We note that at 8-bits of precision, post training quantization schemes provide close to floating point accuracy. In order to better understand the benefits of quantization aware training, we perform experiments to assess performance at 4 bit quantization for weights and activations.

We perform the following experiments:

- **Experiment 1: Per-channel quantization is significantly better than per-layer quantization at 4 bits.** We show that per-channel quantization provides big gains over per-layer quantization for all networks. At 8-bits, the gains were not significant as there were sufficient levels to represent the weights with high fidelity. At four bits, the benefits of per-channel quantization are apparent, even for post training quantization (columns 2 and 3 of Table 5).

- **Experiment 2: Fine tuning can provide substantial accuracy improvements at lower bitwidths.** It is interesting to see that for most networks, one can obtain accuracies within 5% of 8-bit quantization with fine tuning 4 bit weights (column 4 of Table 5). The improvements due to fine tuning are also more apparent at 4 bits. Note that activations are quantized to 8-bits in these experiments.

- **Experiment 3: Lower precision activations:** We investigate the accuracies obtained with 4-bit activations for all layers with and without fine tuning. Note that activations are quantized on a per-layer basis. The weights are quantized at 8-bits of precision with per-channel granularity. We note that fine tuning improves accuracy in this case also. The losses due to activation quantization are more severe than that of weight quantization (see Table 6). Note that the quantization granularity is different for activations and weights, so this is not a fair comparison of the impact of quantization. We hypothesize that quantizing activations

Network \ QuantizationType	Asymmetric, per-layer (Post Training Quantization)	Symmetric,per-channel (Post Training Quantization)	Symmetric, per-channel (Quantization Aware Training)	Floating Point
Mobilenet_v1_1_224	0.02	0.001	0.65	0.709
Mobilenet_v2_1_224	0.001	0.001	0.62	0.719
Nasnet_Mobile	0.001	0.36	0.7	0.74
Mobilenet_v2_1.4_224	0.001	0.001	0.704	0.749
Inception-v3	0.5	0.71	0.76	0.78
Resnet_v1_50	0.002	0.54	0.732	0.752
Resnet_v1_152	0.001	0.64	0.725	0.768
Resnet_v2_50	0.18	0.72	0.73	0.756
Resnet_v2_152	0.18	0.74	0.74	0.778

Table 5: 4 bit Weight Quantization: per-channel quantization outperforms per-layer quantization, with fine tuning providing big improvements.

introduces random errors as the activation patterns vary from image to image, while weight quantization is deterministic. This allows for the network to learn weight values to better compensate for the deterministic distortion introduced by weight quantization.

Network \ QuantizationType	Post Training Quantization (8,4)	Quantization Aware Training (8,4)	Quantization Aware Training (4,8)	Floating Point
Mobilenet_v1_1_224	0.48	0.64	0.65	0.709
Mobilenet_v2_1_224	0.07	0.58	0.62	0.719
Resnet_v1_50	0.36	0.58	0.732	0.752
Nasnet_Mobile	0.04	0.4	0.7	0.74
Inception_v3	0.59	0.74	0.76	0.78

Table 6: 4 bit Activation Quantization with and without fine tuning. Note that weights are quantized with symmetric per-channel quantization at 8-bits. We also show results for 4-bit per-channel quantization of weights with 8-bit activations to compare with 8-bit weights and 4-bit activations

4 Training best practices

We experiment with several configurations for training quantized models: Our first experiment compares stochastic quantization with deterministic quantization. Subsequently, we study if training a quantized model from scratch provides higher accuracies than fine tuning from a floating point model. We also evaluate different methods for quantizing batch normalization layers and show that batch normalization with corrections provides the best accuracy. We also compare schemes that average weights during training with no averaging.

1. **Stochastic Quantization does not improve accuracy:** Stochastic quantization determines floating point weights that provide robust performance under stochastic quantization, which causes the quantized weights to vary from mini-batch to mini-batch. At inference, quantization is deterministic, causing a mismatch with training. We observe that due to this mis-match, stochastic quantization underperforms determinstic quantization (figure 12), which can be compensated better during training.

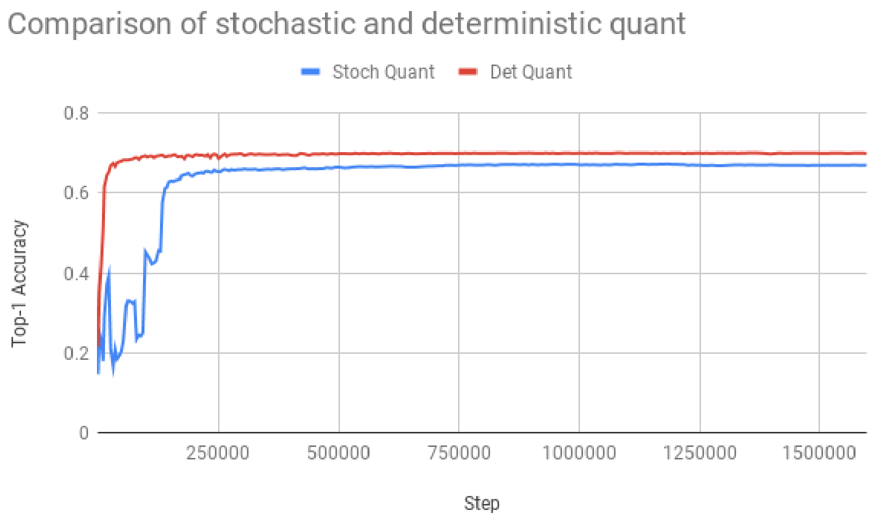


Figure 12: Comparison of stochastic quantization vs deterministic quantization during training

2. **Quantizing a model from a floating point checkpoint provides better accuracy:** The question arises as to whether it is better to train a quantized model from scratch or from a floating point model. In agreement with other work [27], we notice better accuracy when we fine tune a floating point model as shown in figure 13. This is consistent with the general observation that it is better to train a model with more degrees of freedom and then use that as a teacher to produce a smaller model ([28]).

3. **Matching Batch normalization with inference reduces jitter and improves accuracy.** We show results for two networks. In the first experiment (see figure

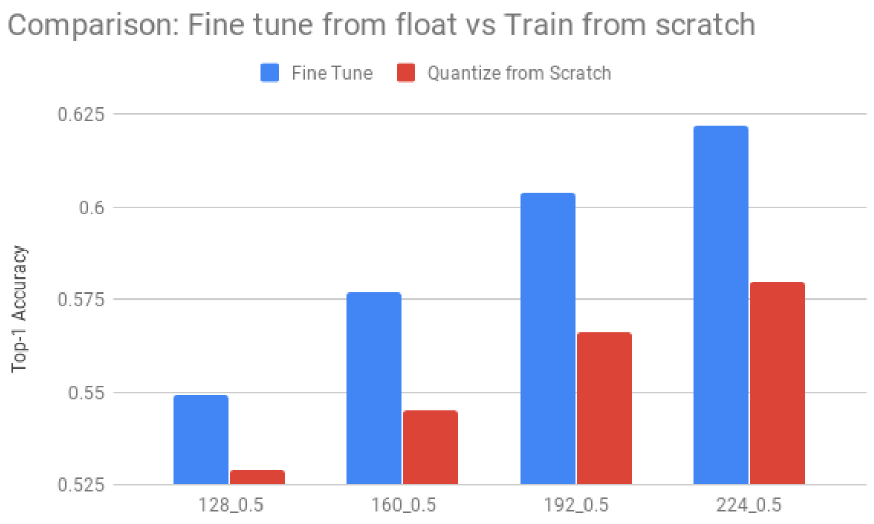


Figure 13: Fine tuning a floating point checkpoint provides better fixed point accuracy

14), we compare training with naive batch norm folding, batch renormalization and batch normalization with correction and freezing for Mobilenet-v1_1_224. We note stable eval accuracy and higher accuracy with our proposed approach. In the second experiment, we compare naive batch norm folding and batch normalization with correction and freezing for Mobilenet_v2_1_224. We note that corrections stabilize training and freezing batch norms provides additional accuracy gain, seen after step 400000 in figure 15.

4. **Use Exponential moving averaging for quantization with caution.** Moving averages of weights [29] are commonly used in floating point training to provide improved accuracy [30]. Since we use quantized weights and activations during the back-propagation, the floating point weights converge to the quantization decision boundaries. Even minor variations in the floating point weights, between the instantaneous and moving averages can cause the quantized weights to be significantly different, hurting performance, see drop in accuracy for the EMA curve in figure 15.

5 Model Architecture Recommendations

In this section, we explore choices of activation functions and tradeoffs between precision and width of a network.

- **Do not constrain activation ranges:** One can get slightly better accuracy by replacing ReLU6 non-linearity with a ReLU and let the training determine the

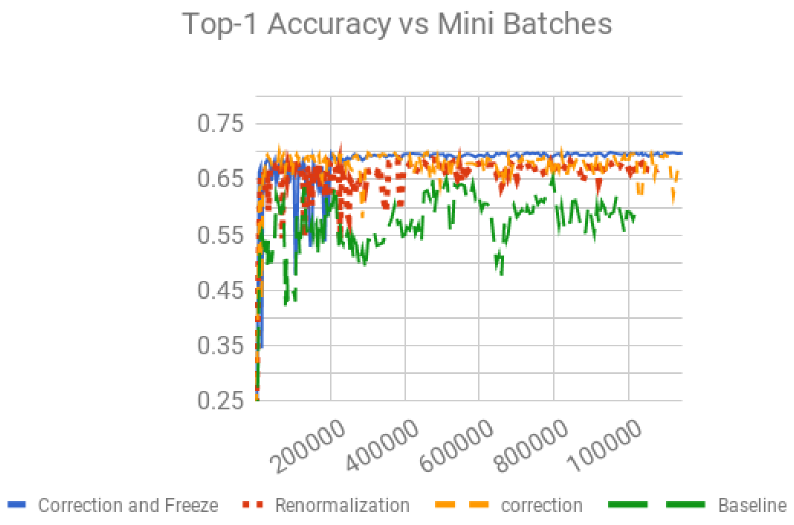


Figure 14: Mobilenet_v1_1_224: Comparison of Batch normalization quantization schemes: Batch normalization without corrections (green) shows a lot of jitter due to the changing scaling of weights from batch to batch. Batch renormalization (red) improves the jitter, but does not eliminate it. Quantizing the weights using moving average statistics reduces jitter, but does not eliminate it (orange). Freezing the moving mean and variance updates after step 200000 allows for quantized weights to adapt to the batch norm induced scaling and provides the best accuracy with minimal jitter (blue curve).

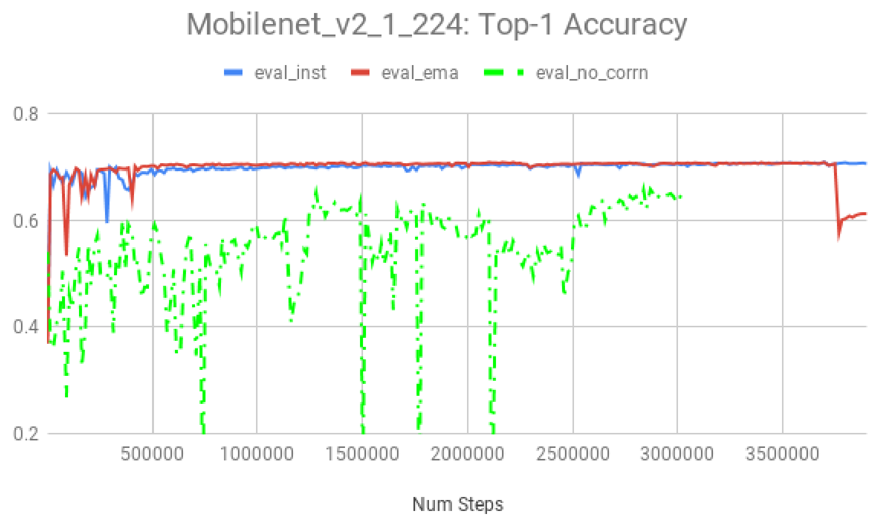


Figure 15: Mobilenet_v2_1_224: Impact of batch normalization corrections and freezing on accuracy. Quantizing without corrections shows high jitter (green curve). Correction with freezing show good accuracy (blue and red curves). The jitter in the eval accuracy drops significantly after moving averages are frozen (400000 steps). Note under performance of EMA weights (red curve) after sufficient training.

activation ranges (see figure 16)

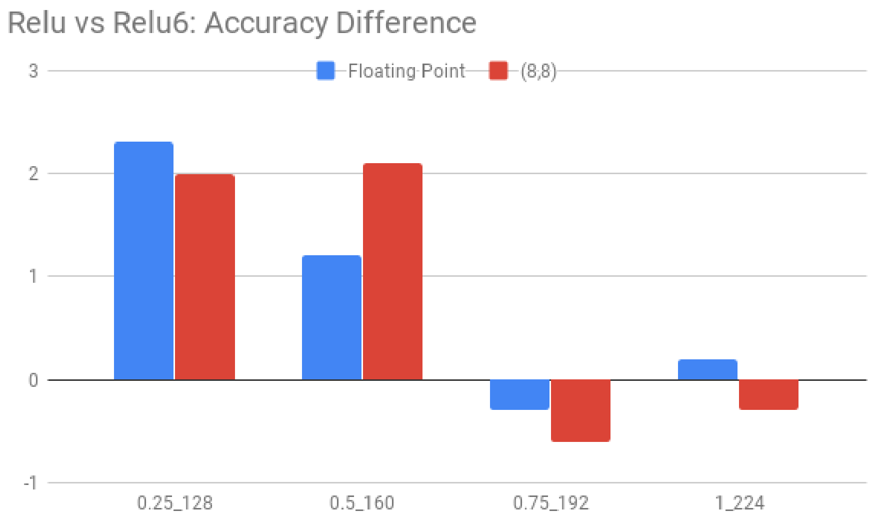


Figure 16: Accuracy improvement of training with ReLU over ReLU6 for floating point and quantized mobilenet-v1 networks.

- **Explore tradeoff of width vs quantization:** An over-parameterized model is more amenable to quantization. Even for leaner architectures like mobilenet, one can tradeoff the depth multiplier with the precisions of the weights. We compare the accuracies obtained with 4 bit per-channel quantization of weights with 8-bit quantization across different depth multipliers in figure 17. Note that this comparison allows us to evaluate a depth vs quantization tradeoff (see [31]). It is interesting to see that one can obtain a further 25% reduction in the model size for almost the same accuracy by moving to 4 bit precision for the weights.

6 Run-time measurements

We measure the run-times (Table 7) on a single large core of the Google Pixel 2 device for both floating point and quantized models. We also measure run-times using the Android NN-API on Qualcomm's DSPs. We see a speedup of 2x to 3x for quantized inference compared to float, with almost 10x speedup with Qualcomm DSPs.

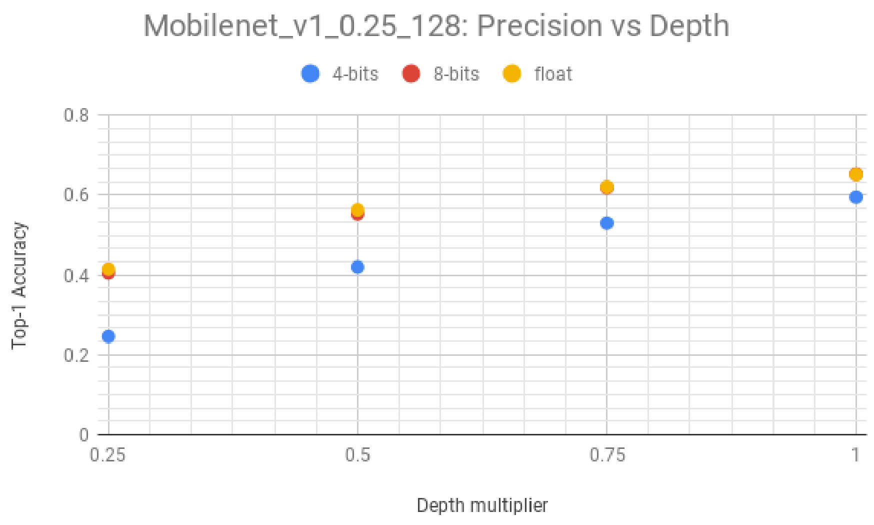


Figure 17: Width vs Precision tradeoff, illustrated for Mobilenet-v1_0.25_128, per-channel quantization of weights

Network \ Inference Platform	Floating point(CPU)	Fixed point (CPU)	Fixed point (HVX, NN-API)
Mobilenet_v1_1_224	155	68	16
Mobilenet_v2_1_224	105	63	15.5
Mobilenet_v1_1_224_SSD	312	152	
Inception_v3	1391	536	
Resnet_v1_50	874	440	
Resnet_v2_50	1667	1145	
Resnet_v1_152	2581	1274	
Resnet_v2_152	4885	3240	

Table 7: Inference time measurements on Pixel2 phone in milliseconds on a single large core.

7 Neural network accelerator recommendations

In order to fully realize the gains of quantized networks, we recommend that neural network accelerators consider the following enhancements:

1. **Aggressive operator fusion:** Performing as many operations as possible in a single pass can lower the cost of memory accesses and provide significant improvements in run-time and power consumption.

2. **Compressed memory access:** One can optimize memory bandwidth by supporting on the fly de-compression of weights (and activations). A simple way to do that is to support lower precision storage of weights and possibly activations.

3. **Lower precision arithmetic:** One can obtain further acceleration by supporting a range of precisions for arithmetic. Our recommendation is to support 4,8 and 16-bit weights and activations. While 4 and 8-bit precisions are sufficient for classification, higher precision support is likely needed for regression applications, like super-resolution and HDR image processing.

4. **Per-layer selection of bitwidths:** We expect that many layers of a network can be processed at lower precision. Having this flexibility can further reduce model size and processing time.

5. **Per-channel quantization:** Support for per-channel quantization of weights is critical to allow for:

   (a) Easier deployment of models in hardware, requiring no hardware specific fine tuning.

   (b) Lower precision computation.

8 Conclusions and further work

Based on our experiments, we make the following conclusions:

- **Quantizing models**

  1. Use symmetric-per-channel quantization of weights with post training quantization as a starting point. Optionally fine tune if there is an accuracy drop.
  2. Quantization aware training can narrow the gap to floating point accuracy and in our experiments, reduce the gap to within 5% of 8-bit quantized weights, even when all layers are quantized to 4 bits of precision.

- **Performance**

  1. Quantized inference at 8-bits can provide 2x-3x speed-up on a CPU and close to 10x speedup compared to floating point inference on specialized processors optimized for low precision wide vector arithmetic, like the Qualcomm DSP with HVX.

2. One can obtain a model size reduction of 4x with no accuracy loss with uniform quantization. Higher compression can be obtained with non-uniform quantization techniques like K-means ([6]).

- **Training Techniques**
  1. Quantization aware training can substantially improve the accuracy of models by modeling quantized weights and activations during the training process.
  2. It is critical to match quantized inference with the forward pass of training.
  3. Special handling of batch normalization is required to obtain improved accuracy with quantized models.
  4. Stochastic quantization during training underperforms deterministic quantization.
  5. Exponential Moving Averages of weights may under-perform instantaneous estimates during quantization aware training and must be used with caution.
- **Model architectures for quantization**
  1. There is a clear tradeoff between model size and compressibility. Larger models are more tolerant of quantization error.
  2. Within a single architecture, one can tradeoff feature-maps and quantization, with more feature maps allowing for lower bitwidth kernels.
  3. One can obtain improved accuracy by not constraining the ranges of the activations during training and then quantizing them, instead of restricting the range to a fixed value. In our experiments, it was better to use a ReLU than a ReLU6 for the activations.

Going forward, we plan to enhance our automated quantization tool to enable better quantization of networks by investigating the following areas:

1. Regularization techniques to better control the dynamic ranges of weights and activations can provide further improvements.
2. Distilled training to further improve the accuracy of quantized models [32].
3. Per-layer quantization of weights and activations to provide further compression and performance gains on hardware accelerators. Reinforcement learning has been applied successfully towards this problem in [33].

9 Acknowledgements

This work builds on the quantization scheme first introduced in [4]. Suharsh Sivakumar developed the [23] tool used for the quantization experiments, which extend the capabilities first described in [4]. Rocky Rhodes provided the performance measurement numbers for the models. We would also like to thank Cliff Young, Song Han, Rocky

Rhodes and Skirmantas Kligys for their useful comments. Pete Warden provided useful input on the scope of the paper and suggested several experiments included in this document.

References

[1] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," 2018.

[2] A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," Apr. 2017.

[3] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[4] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," Dec. 2017.

[5] M. Courbariaux, Y. Bengio, and J. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," 2015.

[6] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[7] GEMMLOWP, "Gemmlowp: a small self-contained low-precision GEMM library." https://github.com/google/gemmlowp.

[8] Intel(R) MKL-DNN, "Intel(R) Math Kernel Library for Deep Neural Networks." https://intel.github.io/mkl-dnn/index.html.

[9] ARM, "Arm cmsis nn software library." http://arm-software.github.io/CMSIS_5/NN/html/index.html.

[10] Qualcomm onQ blog, "How can Snapdragon 845's new AI boost your smartphone's IQ?." https://www.qualcomm.com/news/onq/2018/02/01/how-can-snapdragon-845s-new-ai-boost-your-smartphones-iq.

[11] Nvidia, "8 bit inference with TensorRT." http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf.

[12] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *CoRR*, vol. abs/1703.09039, 2017.

[13] Nvidia, "The nvidia deep learning accelerator." http://nvdla.org/.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," 2016.

[15] "Android Neural Network API." https://developer.android.com/ndk/guides/neuralnetworks/#quantized_tensors.

[16] "Gemmlowp:building a quantization paradigm from first principles." https://github.com/google/gemmlowp/blob/master/doc/quantization.md#implementation-of-quantized-matrix-multiplication.

[17] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," 2018.

[18] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," Dec. 2015.

[19] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," 2017.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," 2016.

[22] T. Sheng, C. Feng, S. Zhuo, X. Zhang, L. Shen, and M. Aleksic, "A quantization-friendly separable convolution for mobilenets," 2018.

[23] "Tensorflow quantization library." https://www.tensorflow.org/api_docs/python/tf/contrib/quantize.

[24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[25] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[26] "Slim model repository." https://github.com/tensorflow/models/tree/master/research/slim.

[27] A. K. Mishra and D. Marr, "Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy," 2017.

[28] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” Mar. 2015.

[29] B. Polyak, “New stochastic approximation type procedures,” Jan 1990.

[30] “Exponential moving average.”
https://www.tensorflow.org/api_docs/python/tf/train/ExponentialMovingAverage.

[31] A. K. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, “WRPN: wide reduced-precision networks,” *CoRR*, vol. abs/1709.01134, 2017.

[32] A. K. Mishra and D. Marr, “Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy,” 2017.

[33] Y. He and S. Han, “ADC: automated deep compression and acceleration with reinforcement learning,” *CoRR*, vol. abs/1802.03494, 2018.

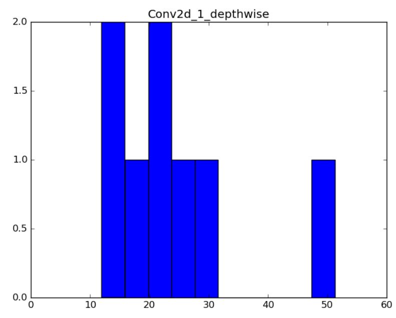
A Impact of Batch Normalization on Quantization

To understand the impact of batch normalization on the dynamic range of the folded weights (W), we consider the following metrics:

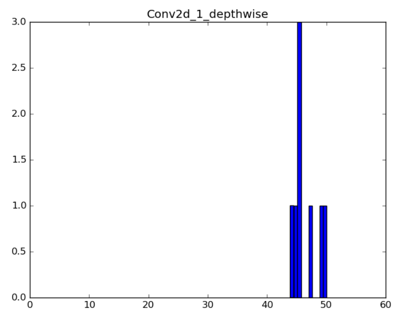
1. **SQNR:** We calculate the Signal to quantization noise ratio defined as:

$$SQNR = 10log_{10}\frac{\sum W^2}{\sum(W - SimQuant(W))^2}$$

for different quantization schemes. We note that per-channel quantization provides significant improvement in SQNR over per-layer quantization, even if only symmetric quantization is used in the per-channel case.



(a) Asymmetric, per-layer Quant



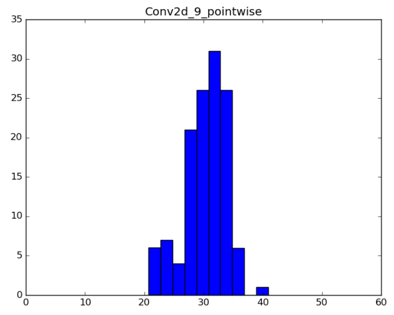
(b) Symmetric, per-channel Quant

Figure 18: Histogram of the SQNR per output feature map (in dB on the x-axis), showing the number of kernels for each SQNR bin for different weight quantization schemes for layer:Conv2d_1_depthwise, Mobilenet_v1_0.25_128. The total number of kernels is 8.

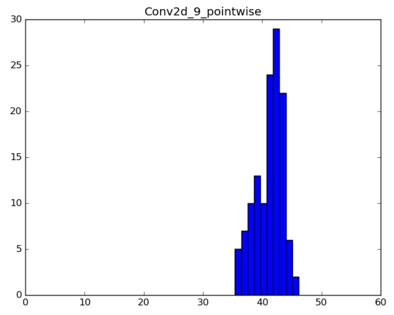
2. **Weight Power Distribution:** We also plot the distribution of the sample weights, normalized by the average power, i.e we plot

$$histogram(\frac{W^2}{E[W^2]})$$

for the weights before and after folding. We note that after folding, there are much larger outliers which severely degrade performance.



(a) Asymmetric, per-layer Quant



(b) Symmetric, per-channel Quant

Figure 19: Histogram of the SQNR per output feature map (in dB on the x-axis), showing the number of kernels for each SQNR bin for different weight quantization schemes for layer:Conv2d_9_pointwise, Mobilenet_v1_0.25_128. The total number of kernels is 128.

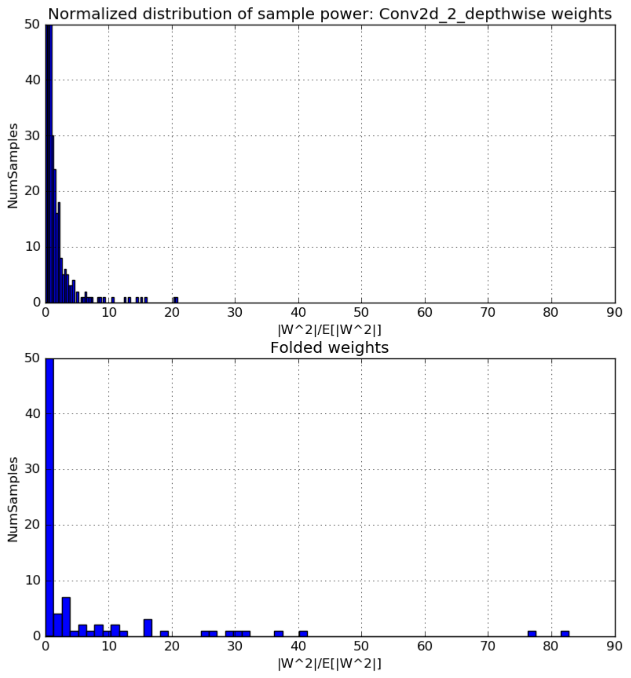


Figure 20: Weight histograms with and without folding: Mobilenet_V1_1_224, conv2d_2_depthwise, note the long tails of the distribution for folded weights.