

# Towards Deterministic Rounding Behaviour in Quantized Convolutions

(A design-note-in-progress) 9-Nov-2018

Greg Smith smithg@qti.qualcomm.com

## 1 Introduction

This document discusses the specification of our 8-bit and 16-bit quantization mechanism, and proposes a minor adjustment which will support, under a reasonable set of conditions, bit-reproducible rounding in convolution operations (i.e. exactly the same results can be reproduced by an independent implementation). This can be useful when network designers wish to include compensation for run-time rounding errors as part of network training, and thus need to model those errors exactly.

To illustrate, consider an  $8 \times 8 \rightarrow 8$  convolution with 32-bit biases, for the case where the output quantization range is predetermined (and weights and biases are pre-quantized).

This operation can be broken down as these steps:

- A convolution, processed as integer operations on the quantized inputs
- Addition of bias values; the scaling of these depends on quantization
- An offset, scaling, saturation operation which generates the final output. The scaling and offset are functions of the input and output scaling, and the operation itself can be expressed in simple integer calculation (add, multiply, right-shift, saturate).

A goal of this document is to develop a recommendation which allows the result of the process to be reproducible exactly in another implementation, provided:

- The quantization ranges of the input, output, and weights comply with the recommendations below (and, of course, match exactly across the two implementations);
- The 32-bit bias values are encoding using a quantization step which matches  $\text{in\_step} * \text{weight\_step} * k$  (calculated as a floating point result) where  $\text{in\_step}$  and  $\text{weight\_step}$  are the quantization steps of those inputs, and  $k$  is a power of 2. This allows the scaling of the bias values to the operation to be reproducible [This needs more work].
- The calculation and use of the scaling and offset values, for the convolution output, are done as described below [yet to be added].

## 2 Terms

- **Floating point** or **float** refers to 32-bit IEEE floating point.
- **Lossless** refers to a computation done without no rounding, so that the actual result is the same as the mathematically exact result. Integer multiply and adds are lossless. Floating point overflow/underflow, and integer overflow, are not considered, and assumed not to occur by design.
- **Reproducible** describes a computation which may have rounding errors, but from which we can expect two implementations to produce identical results, provided the sequence of such operations is adequately specified.

## 3 Quantization Overview

When storing “Quantized” data we use a finite number ( e.g.  $2^8$  or  $2^{16}$  ) of integer codes to represent continuous data; for a given set quantized values  $q_i$  , we need to know the quantization parameters in order to convert these to the 'dequantized' values  $x_i$  they represent.

This is a simple linear relationship, and we have two ways to represent it:

- Scale and zero  $s, z$ :

$$x_i = s(q_i - z) \quad (1)$$

- Nominal min and max  $\alpha, \beta$  :

$$x_i = \alpha + \frac{\beta - \alpha}{N} q_i \quad (2)$$

Here,  $\alpha$  is simply the dequantized value associated with quantized value  $q_i=0$  , and  $\beta$  is the value associated with  $q_i=N$  ; for 8-bit data we use  $N=255$ , so that  $\alpha, \beta$  represent the smallest and largest possible  $x$ .

Conversions between the two forms may be made as follows:

$$s = \frac{\beta - \alpha}{N}; \quad z = -\frac{\alpha}{s} \quad (3)$$

$$\alpha = -zs; \quad \beta = (N - z)s \quad (4)$$

For reasons of computational convenience, we always require:

$$s > 0; \quad z \in \mathbb{Z}; \quad 0 \leq z \leq N \quad (5)$$

... which together imply:  $\alpha \leq 0 \leq \beta; \quad \alpha < \beta$

i.e. the range may not be empty, and must include zero.

## 4 Lossless Conversion Between Representations

It is possible to restrict the allowed quantization parameters, in such a way as to guarantee that the computations in (3) and (4) are lossless when performed in floating point.

### 4.1 8-bit Quantization, $N=255$ :

By choosing a value of  $s$  which can be expressed in at most 16 mantissa bits (i.e. it can be expressed exactly as  $s = k2^e$  where  $k, e \in \mathbb{Z}$ ,  $32768 \leq k \leq 65535$ ) the operations in (4) are lossless when done in single-precision floats, and the resulting  $\alpha, \beta$  are such that  $\beta - \alpha$  is lossless and yields exactly 255  $s$ ; thus the formulae for  $s$  and  $z$  in (3) will give the original value exactly (see section 7 regarding division by 255).

Also, with these constraints, it is possible to losslessly convert quantized data to float: either  $x_i = s(q_i - z)$  or  $x_i = \alpha + sq_i$  may be used, and the conversion will be exact if done in IEEE floats.

Conversion from float to quant is lossy, of course; it may be useful to define a reproducible method based on a given quantization specification, but currently I don't have one and it may not be trivial to bring the existing code to comply to a given specification – values which fall nearly or exactly halfway between two quantization levels are at risk of being quantized differently in different implementations. Furthermore, the process of finding a quantization spec to cover a given range of floats is not as yet defined in a reproducible form.

### 4.2 16-bit Quantization

For the 16-bit case, it's possible to use  $N = 2^{16} - 1 = 65535$ , but in order to get the exact computations we would need to generate min/max values which are representable in floats as differing by exact multiples of 65535, which is a problematic constraint when only 24 mantissa bits are available.

Instead we use  $N = 2^{16} = 65536$ , and continue to use (3) and (4); this means that  $\alpha$  is still the minimum representable value (code 0) but  $\beta$  is the 'nominal' maximum: the value which *would* be represented by code 65536.

If we choose values of  $s$  which are representable in 22 mantissa bits, the multiplications in (4) to find  $\alpha, \beta$  are generally not lossless, but if floating point operations are done with convergent rounding (the default mode) the difference  $\beta - \alpha$  in (3) will always<sup>1</sup> give exactly  $65536s$ ; the multiplication by  $(1/N)$  is lossless

---

1 I don't have a proof for this, but testing over all  $2^{37}$  cases (all cases with a specific  $s$  exponent) found no failures.

and thus we can recover  $s$  exactly. The value of  $z$  found as  $z = -\alpha/s$  can be inexact, but the worst absolute error is about  $\pm 2^{-8}$  so  $z$  can be easily corrected by rounding to the nearest integer.

Dequantization of 16-bit floats should be done as  $x_i = s(q_i - z)$  where the subtraction is an exact integer operation, the difference is converted losslessly to float, and the multiplication by  $s$  is lossy but reproducible.

## 5 Determining Quantization For a Given Range of Data

### 5.1 8-bit Quantization, $N=255$

Whenever we need to quantize a given range of floating point values  $\{x_i\}$ , the following process is recommended to find the quantization parameters:

- Find the actual range over  $\{x_i\}$ , including zero:
 
$$x_{min} = \min\{0, x_0, x_1, \dots\}$$

$$x_{max} = \max\{0, x_0, x_1, \dots\}$$
- In the case where  $x_{min}=0$  :
  - set  $z = 0$ , and  $s = x_{max}/255$
  - If  $x_{max}=0$ , all of the  $x_i$  are zero and we can use an arbitrary value such as  $s = 2^{-8}$
- Otherwise if  $x_{max}=0$  :
  - set  $z = 255$ , and  $s = -x_{min}/255$
- Otherwise we need to find an integer zero point in 1...254, and scale which encompass the desired range. This method finds the smallest feasible  $s$ :
  - Find approximate value for the zero point,  $\hat{z} = -\frac{255 \cdot x_{min}}{x_{max} - x_{min}}$  in range  $[0, 255]$
  - Two candidate  $z$  values are the nearest integers  $z_0 = \lfloor \hat{z} \rfloor$ ,  $z_1 = z_0 + 1$
  - Corresponding minimal  $s$  values are  $s_0 = -\frac{x_{min}}{z_0}$ ,  $s_1 = \frac{x_{max}}{255 - z_1}$  ;
  - use  $(s, z) = (s_0, z_0)$  if  $z_0 > 0$  and ( $z_1 \geq 255$  or  $s_0 < s_1$ ), otherwise use  $(s, z) = (s_1, z_1)$
- In any case round the  $s$  value upward, if needed, to the next value which may be expressed in 16 mantissa bits.
- $\alpha, \beta$  can be then found via (4); and it is guaranteed that the conditions in (5) are met, and that  $\alpha \leq x_{min}$ ,  $x_{max} \leq \beta$ ; no solution with a smaller  $s$  exists.

Note that the condition  $s_0 < s_1$  can be evaluated as  $(z_0 - 254)x_{min} < z_0 x_{max}$

This procedure can be used to “correct” a possibly unreliable  $(\alpha, \beta)$  range to a similar one which meets the criteria: if the  $(x_{min}, x_{max})$  are set to the given  $(\alpha, \beta)$ , the new  $(\alpha, \beta)$  resulting will

include the original range; and should be identical to the original range when that already meets the criteria (i.e. the process is idempotent).

However, the procedure is not considered to be reproducible for arbitrary  $(x_{min}, x_{max})$ ; in some cases, slight differences in the way the computation is done could produce different choice of  $z$ , for instance, when the  $s_0, s_1$  are very similar or identical.

Examples:

- $(x_{min}, x_{max}) = (0, 1) \Rightarrow (s, z) = (0.003921628, 0) \Rightarrow (\alpha, \beta) = (0, 1.0000151)$   
Here,  $s = 0x8081 \times 2^{-23}$  and  $\beta = 0x80007F \times 2^{-23}$  exactly.
- $(x_{min}, x_{max}) = (-1, 1) \Rightarrow (s, z) = (0.0078742504, 128) \Rightarrow (\alpha, \beta) = (-1.0079041, 1.0000298)$   
Here,  $s = 0x8103 \times 2^{-22}$  and  $\alpha = -0x8103 \times 2^{-15}$ ,  $\beta = 0x40007D \times 2^{-22}$  exactly. It's also possible to use the same value of  $s$ , and  $z=127$ .

## 5.2 16-bit Quantization, N=65536

The method for finding scaling over a given range is below. This assumes that we don't want  $x_{max}$  to map to at most code 65535; we don't want it to map to 65536 and then be truncated to 65535. So we want a quantization specification for which  $\alpha \leq x_{min}, x_{max} \leq (\beta - s)$ .

- Find the actual range over  $\{x_i\}$ , including zero:  

$$x_{min} = \min\{0, x_0, x_1, \dots\}$$

$$x_{max} = \max\{0, x_0, x_1, \dots\}$$
- In the case where  $x_{min}=0$ :
  - set  $z = 0$ , and  $s = x_{max}/65535$
  - If  $x_{max}=0$ , all of the  $x_i$  are zero and we can use an arbitrary value such as  $s = 2^{-16}$
- Otherwise if  $x_{max}=0$ :
  - set  $z = 65535$ , and  $s = -x_{min}/65535$
- Otherwise we need to find an integer zero point in 1..65534, and scale which encompass the desired range. This method finds the smallest  $s$ :
  - Find approximate value for the zero point,  $\hat{z} = -\frac{65535 \cdot x_{min}}{x_{max} - x_{min}}$  (in range  $[0, 65535]$ )
  - Two candidate  $z$  values are the nearest integers  $z_0 = \lfloor \hat{z} \rfloor$ ,  $z_1 = z_0 + 1$

- Corresponding minimal  $s$  values are  $s_0 = -\frac{x_{min}}{z_0}$ ,  $s_1 = \frac{x_{max}}{65535 - z_1}$  ;
- use  $(s, z) = (s_0, z_0)$  if  $z_0 > 0$  and ( $z_1 \geq 65535$  or  $s_0 < s_1$ ), otherwise use  $(s, z) = (s_1, z_1)$
- In any case round the  $s$  value upward, if needed, to the next value which may be expressed in 22 mantissa bits.
- $\alpha, \beta$  can be then found via (4); and it is guaranteed that the conditions in (5) are met, and that  $\alpha \leq x_{min}$ ,  $x_{max} \leq (\beta - s)$  ; no solution with a smaller  $s$  exists.

The condition  $s_0 < s_1$  can be evaluated as  $(z_0 - 65534) x_{min} < z_0 x_{max}$

For the 8-bit case it is possible to have  $\alpha < 0, \beta = 0$  ; but for the 16-bit case  $z \leq 65535$ , so  $\beta \geq s$

Unlike the  $N=255$  process, this is *not* idempotent – if resulting  $(\alpha, \beta)$  are used as  $(x_{min}, x_{max})$  and the process is run again, the new range will be larger.

Thus it should only be used to find a “new” nominal range to cover a given range of values, not to ‘correct’ existing nominal ranges.

Correcting an existing nominal range (i.e. modifying it as little as possible to meet conditions in (5), and the constraint on  $s$ ) can be done by the following idempotent method:

- find  $s, z$  using (3);
- round  $z$  to the nearest integer, limiting it to range 0..65535
- round  $s$  to the *nearest* value which can be expressed in 22 mantissa bits.
- Convert back to  $(\alpha, \beta)$  using (4)

Applying this correction to an arbitrary  $(\alpha, \beta)$  range may produce only a small change in the interpretation of the quantized codes; mostly due to the rounding of  $z$ , which produces an offset of at most  $\pm 2^{-17}(\beta - \alpha)$  .

In the special case where  $\alpha = 0$  , the correction can be done by simply rounding  $\beta$  to the nearest value which can be expressed in 22 mantissa bits, since in this case  $z=0$  and  $\beta = 2^{16}s$

Examples:

- $(x_{min}, x_{max}) = (0, 1) \Rightarrow (s, z) = (1.5259029 \times 10^{-5}, 0) \Rightarrow (\alpha, \beta) = (0, 1.0000157)$   
Here,  $s = 0x200021 \times 2^{-37}$  and  $\beta = 0x200021 \times 2^{-21}$  exactly.

- $(x_{min}, x_{max}) = (-1, 1) \Rightarrow (s, z) = (3.0518524 \times 10^{-5}, 32768)$   
 $\Rightarrow (\alpha, \beta) = (-1.0000305, 1.0000305)$   
 Here,  $s = 0x200041 \times 2^{-36}$  and  $\alpha, \beta = \mp 0x200041 \times 2^{-21}$  exactly.

## 6 Final Rounding in Convolutions

For a convolution, or a matrix multiply, we want to find

$$\mathbf{y} = \mathbf{x} * \mathbf{w} + \mathbf{b}$$

where the input values are actually represented by quantized values  $\hat{\mathbf{x}}, \hat{\mathbf{w}}, \hat{\mathbf{b}}$ , defined by quantization parameters

$$\mathbf{x} = s_x(\hat{\mathbf{x}} - z_x) \quad \mathbf{w} = s_w(\hat{\mathbf{w}} - z_w) \quad \mathbf{b} = s_b(\hat{\mathbf{b}} - z_b)$$

...and we want the output as  $\hat{\mathbf{y}}$  such that  $\mathbf{y} = s_y(\hat{\mathbf{y}} - z_y)$ . All of the  $s_*$  and  $z_*$  are given.

(the details of the convolution subscripts and summation are omitted, since the discussion applies to any sum-of-products operation).

Also, for now this discussion applies only to operations with 8-bit inputs and outputs.

First, we will find a integer convolution result as follows:

$$\hat{\mathbf{p}} = (\hat{\mathbf{x}} - z_x) * (\hat{\mathbf{w}} - z_w)$$

This is a lossless integer operation, and it is clear that

$$\begin{aligned} \mathbf{y} &= s_x s_w \hat{\mathbf{p}} + \mathbf{b} \\ \Rightarrow \hat{\mathbf{y}} - z_y &= \frac{s_x s_w}{s_y} \hat{\mathbf{p}} + \frac{1}{s_y} \mathbf{b} \\ \Rightarrow \hat{\mathbf{y}} - z_y &= \frac{s_x s_w}{s_y} \left( \hat{\mathbf{p}} + \frac{1}{s_x s_w} \mathbf{b} \right) \\ \Rightarrow \hat{\mathbf{y}} &= \frac{s_x s_w}{\lambda s_y} \left( \lambda \hat{\mathbf{p}} + \frac{\lambda s_b}{s_x s_w} (\hat{\mathbf{b}} - z_b) + \frac{\lambda s_y}{s_x s_w} z_y \right) \end{aligned}$$

(where  $\lambda$  is a scaling constant to be discussed below)

So the rest of the computation consists of:

- Scaling the bias: i.e. finding  $\hat{\mathbf{b}}_p = \frac{\lambda s_b}{s_x s_w} (\hat{\mathbf{b}} - z_b)$ , rounding to nearest integer;
- Finding the output offset,  $Z_p = \frac{\lambda s_y}{s_x s_w} z_y$ , and rounding it to the nearest integer

- Finding the overall scale  $S_p = \frac{S_x S_w}{\lambda S_y}$  in some fixed-point format.
- The result is then  $\hat{\mathbf{y}} = S_p(\lambda \hat{\mathbf{p}} + \hat{\mathbf{b}}_p + Z_p)$

$\lambda$  is chosen as a power of 2, and is usually 1; so the multiplication by  $\lambda$  is actually a shift.

In most cases,  $S_p$  is considerably less than one even when  $\lambda=1$ , so the process of rounding

$Z_p = \frac{1}{S_p} z_y$  and  $\hat{\mathbf{b}}_p$  to integer produces rounding errors which are small relative to the final rounding. In

situations where this is not true, it may be necessary to use a larger  $\lambda$  value to obtain a smaller  $S_p$ , thus diminishing the visibility of those rounding errors.

In order for this to be reproducible, we need to define the following in a way which makes them reproducible:

- How  $\lambda$  is chosen;
- How  $S_p$  is calculated and rounded;
- How  $\hat{\mathbf{b}}_p, Z_p$  are calculated and rounded; and
- How the multiplication by  $S_p$  is to be done.

[note – currently we support a  $\lambda$  in some of the convolutions, but it can't be used to control rounding errors as above since it's applied to the full sum, not just the  $\mathbf{p}$  term. It is used only to ensure  $S_p < 1$ ]

... In progress....

The multiplication by  $S_p$  is done by finding the full 64-bit product of the fixed-bit  $S_p$  value and the final sum; adding a rounding bias of  $2^{30}$ , and then shifting the result right 31 bits. The result is saturated so as not to exceed the range 0...255.

## 7 Note: Division by 255 in floats

If  $a, b$  are floating-point values such that the float product  $p = 255 \cdot b$  is lossless, and  $a = p$ , then the floating-division  $a/(\text{float})255.0$  will yield exactly  $b$ , we rely on this to have exact conversion from  $(\alpha, \beta)$  to  $s$  in the 8-bit case.

If we choose instead to find  $a * (\text{float})(1./255.0)$ , to reduce computation time, the exact result is *not* guaranteed, since  $(1/255.0)$  falls about midway between two representable values. The nearest float representation of  $(1/255.0)$  is too large by  $2.319 \times 10^{-10}$ ; so if it happens that the multiplication rounds upwards, the result can be inexact even though an exact quotient is representable. This can be avoided by doing the calculation as

$b = r_0 a + r_1 a$ , where  $r_0 = 3.921568859 \times 10^{-3}$ ,  $r_1 = -2.3191758 \times 10^{-10}$ ; the operation should



be done using a ‘fused multiply-add’ so the larger product is not fully rounded before the smaller is added.