A. Appendix: Layer-specific details

A.1. Mathematical functions

Math functions such as hyperbolic tangent, the logistic function, and softmax often appear in neural networks. No lookup tables are needed since these functions are implemented in pure fixed-point arithmetic similarly to how they would be implemented in floating-point arithmetic[7].

A.2. Addition

Some neural networks use a plain Addition layer type, that simply adds two activation arrays together. Such Addition layers are more expensive in quantized inference compared to floating-point because *rescaling* is needed: one input needs to be rescaled onto the other's scale using a fixed-point multiplication by the multiplier $M = S_1/S_2$ similar to what we have seen earlier (end of section 2.2), before the actual addition can be performed as a simple integer addition; finally, the result must be rescaled again to fit the output array's scale[8].

A.3. Concatenation

Fully general support for concatenation layers poses the same rescaling problem as Addition layers. Because such rescaling of uint8 values would be a lossy operation, and as it seems that concatenation ought to be a lossless operation, we prefer to handle this problem differently: instead of implementing lossy rescaling, we introduce a requirement that all the input activations and the output activations in a Concatenation layer have the same quantization parameters. This removes the need for rescaling and concatenations are thus lossless and free of any arithmetic[9].

B. Appendix: ARM NEON details

This section assumes familiarity with assembly programming on the ARM NEON instruction set. The instruction mnemonics below refer to the 64-bit ARM instruction set, but the discussion applies equally to 32-bit ARM instructions.

The fixed-point multiplications referenced throughout this article map exactly to the SQRDMULH instruction. It is very important to use the correctly-rounding instruction SQRDMULH and not SQDMULH[10].

The rounding-to-nearest right-shifts referenced in section 2.2 do not map exactly to any ARM NEON instruction. The problem is that the "rounding right shift" instruction, RSHL with variable negative offset, breaks ties by rounding *upward*, instead of rounding them *away from zero*. For example, if we use RSHL to implement the division $-12/2^3$, the result will be -1 whereas it should be -2 with "round to nearest". This is problematic as it results in an overall upward bias, which has been observed to cause significant loss of end-to-end accuracy in neural network inference. A correct round-to-nearest right-shift can still be implemented using RSHL but with suitable fix-up arithmetic around it[11].

For efficient NEON implementation of the matrix multiplication's core accumulation, we use the following trick. In the multiply-add operation in (10), we first change the operands' type from uint8 to int8 (which can be done by subtracting 128 from the quantized values and zero-points). Thus the core multiply-add becomes

$$\texttt{int32 += int8 * int8.} \tag{B.1}$$

As mentioned in section 3, with a minor tweak of the quantized training process, we can ensure that the weights, once quantized as int8 values, never take the value -128. Hence, the product in (B.1) is never $-128 * -128$, and is therefore always less than 2^{14} in absolute value. Hence, (B.1) can accumulate *two* products on a local int16 accumulator before that needs to be accumulated into the true int32 accumulator. This allows the use of an 8-way SIMD multiplication (SMULL on int8 operands), followed by an 8-way SIMD multiply-add (SMLAL on int8 operands), followed by a pairwise-add-and-accumulate into the int32 accumulators (SADALP)[12].

C. Appendix: Graph diagrams

D. Experimental protocols

D.1. ResNet protocol

Preprocessing. All images from ImageNet [3] are resized preserving aspect ratio so that the smallest side of the image is 256. Then the center 224×224 patch is cropped and the means are subtracted for each of the RGB channels.

Optimization. We use the momentum optimizer from TensorFlow [1] with momentum 0.9 and a batch size of 32. The learning rate starts from 10^{-5} and decays in a staircase fashion by 0.1 for every 30 epochs. Activation quantization is delayed for 500,000 steps for reasons discussed in section 3. Training uses 50 workers asynchronously, and stops after validation accuracy plateaus, normally after 100 epochs.

[7]Pure-arithmetic, SIMD-ready, branch-free, fixed-point implementations of at least tanh and the logistic functions are given in gemmlowp [18]'s fixedpoint directory, with specializations for NEON and SSE instruction sets. One can see in TensorFlow Lite [5] how these are called.

[8]See the TensorFlow Lite [5] implementation.

[9]This is implemented in this part of the TensorFlow Lite [5] Converter

[10]The fixed-point math function implementations in gemmlowp [18] use such fixed-point multiplications, and ordinary (non-saturating) integer additions. We have no use for general saturated arithmetic.

[11]It is implemented here in gemmlowp [18].

[12]This technique is implemented in the optimized NEON kernel in gemmlowp [18], which is in particular what TensorFlow Lite uses (see the choice of `L8R8WithLhsNonzeroBitDepthParams` at this line).

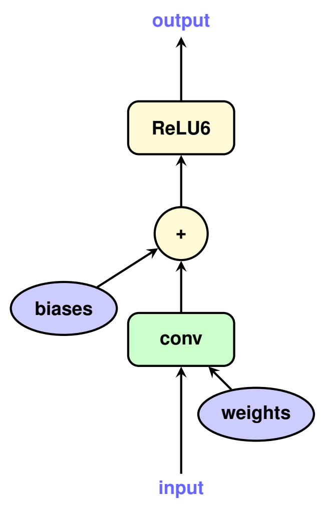

Figure C.1: Simple graph: original

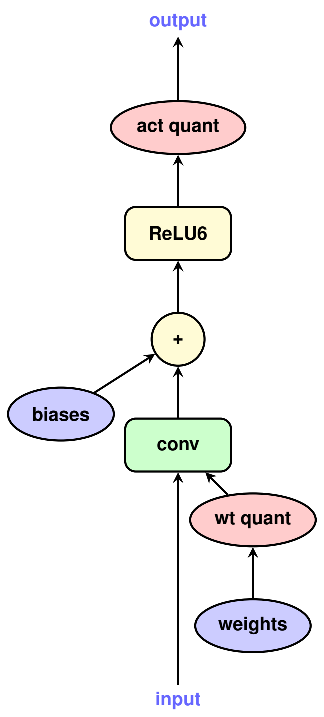

Figure C.2: Simple graph: quantized

D.2. Inception protocol

All results in table 4.3 were obtained after training for approximately 10 million steps, with batches of 32 samples, using 50 distributed workers, asynchronously. Training data were ImageNet 2012 299×299 images with labels. Image augmentation consisted of: random crops, random horizontal flips, and random color distortion. The optimizer used was RMSProp with learning rate starting at 0.045 and decaying exponentially and stepwise with factor 0.94 after every 2 epochs. Other RMSProp parameters were: 0.9 momentum, 0.9 decay, 1.0 epsilon term. Trained parameters were EMA averaged with decay 0.9999.

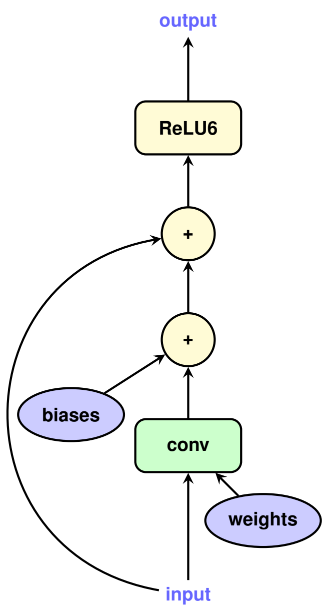

Figure C.3: Layer with a bypass connection: original

D.3. COCO detection protocol

Preprocessing. During training, all images are randomly cropped and resized to 320×320. During evaluation, all images are directly resized to 320×320. All input values are normalized to $[-1, 1]$.

Optimization. We used the RMSprop optimizer from TensorFlow [1] with a batch size of 32. The learning rate starts from 4×10^{-3} and decays in a staircase fashion by a factor of 0.1 for every 100 epochs. Activation quantization is delayed for $500,000$ steps for reasons discussed in section 3. Training uses 20 workers asynchronously, and stops after validation accuracy plateaus, normally after approximately 6 million steps.

Metrics. Evaluation results are reported with the COCO primary challenge metric: AP at IoU=.50:.05:.95. We follow the same train/eval split in [13].

D.4. Face detection and face attribute classification protocol

Preprocessing. Random 1:1 crops are taken from images in the Flickr-based dataset used in [10] and resized to 320×320 pixels for face detection and 128×128 pixels for face attribute classification. The resulting crops are flipped horizontally with a 50% probability. The values for each of the RGB channels are renormalized to be in the range $[-1, 1]$.

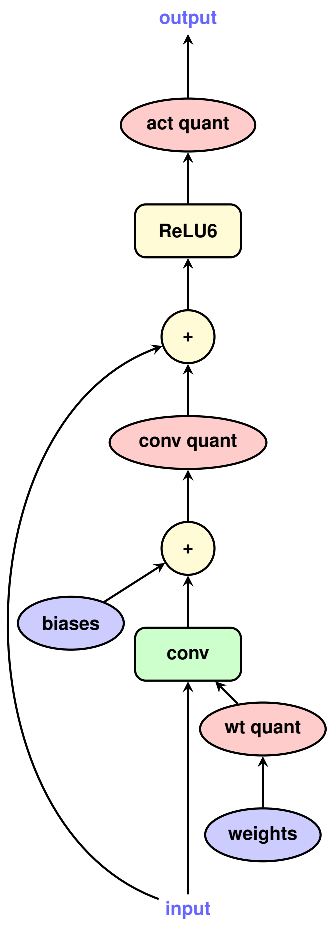

Figure C.4: Layer with a bypass connection: quantized

Face Detection Optimization. We used the RMSprop optimizer from TensorFlow [1] with a batch size of 32. The learning rate starts from 4×10^{-3} and decays in a staircase fashion by a factor of 0.1 for every 100 epochs. Activation quantization is delayed for $500,000$ steps for reasons discussed in section 3. Training uses 20 workers asynchronously, and stops after validation accuracy plateaus, normally after approximately 3 million steps.

Face Attribute Classification Optimization. We followed the optimization protocol in [10]. We used the Adagrad optimizer from Tensorflow[1] with a batch size of 32 and a constant learning rate of 0.1. Training uses 12 workers asynchronously, and stops at 20 million steps.

Latency Measurements. We created a binary that runs the face detection and face attributes classification models repeatedly on random inputs for 100 seconds. We pushed this binary to Pixel and Pixel 2 phones using the `adb push` command, and executed it on 1, 2, and 4 LITTLE cores, and 1, 2, and 4 big cores using the `adb shell` command with the appropriate `taskset` specified. We reported the average runtime of the face detector model on

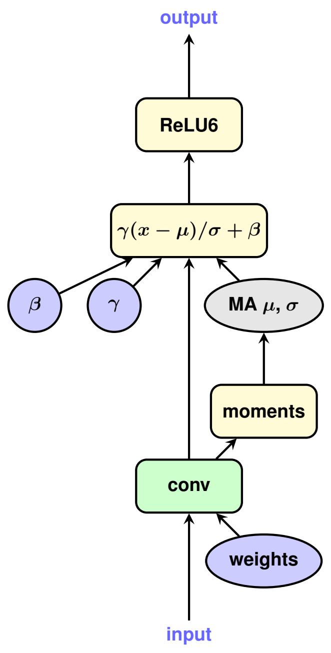

Figure C.5: Convolutional layer with batch normalization: training graph

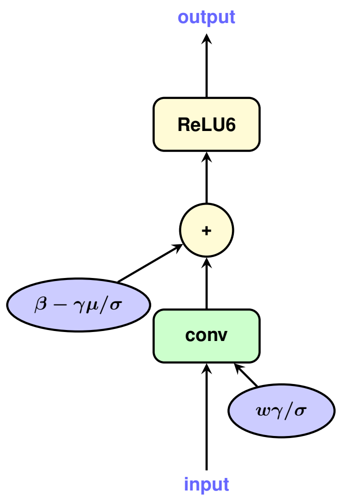

Figure C.6: Convolutional layer with batch normalization: inference graph

320×320 inputs, and of the face attributes classifier model on 128×128 inputs.

Face Attribute Classification Optimization. We followed the optimization protocol in [10]. We used the Adagrad optimizer from Tensorflow[1] with a batch size of 32 and a constant learning rate of 0.1. Training uses 12 workers asynchronously, and stops at 20 million steps.

Face Detection Multi-core Timing. We investigated face detection latency as a function of the number of cores,

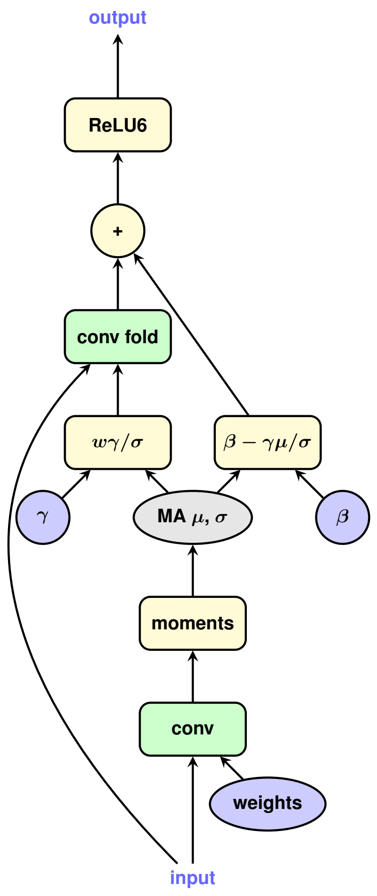

Figure C.7: Convolutional layer with batch normalization: training graph, folded

DM	type	LITTLE Cores			big Cores		
		1	2	4	1	2	4
100%	floats	711	–	–	337	–	–
	8 bits	372	238	167	154	100	69
50%	floats	233	–	–	106	–	–
	8 bits	134	96	74	56	40	30
25%	floats	100	–	–	44	–	–
	8 bits	67	52	43	28	22	18

Table D.1: Face detection: latency of floating point and quantized models on Qualcomm Snapdragon 835 cores.

and showed the results in D.1.

Age Classification Performance. Table D.2 shows the age precision of 5 years for varying weight and activation bit depths. Precision at 5 years refers to the fraction of age predictions that are within 5 years of the ground-truth. It is evident that when the sum of bit depth being equal, the models with the identical weight and activation bit depths achieve the most favorable precision.

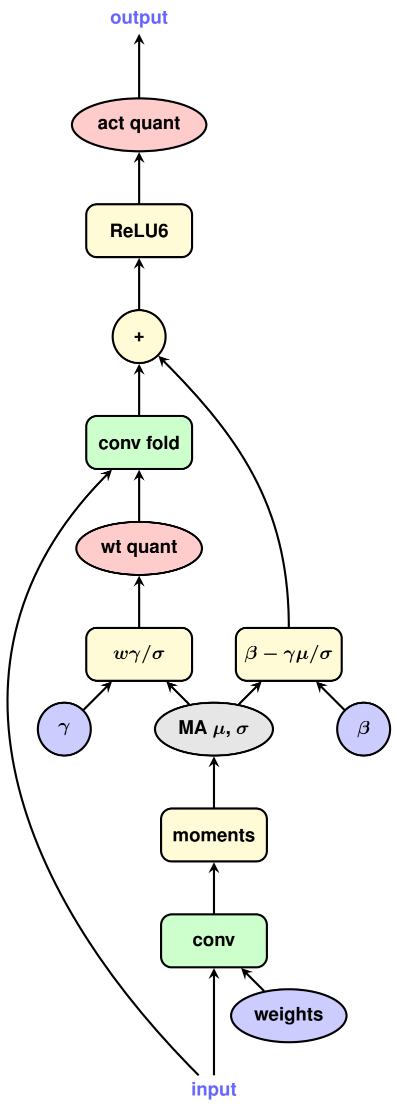

Figure C.8: Convolutional layer with batch normalization: training graph, folded and quantized

wt. \ act.	8	7	6	5	4
8	-1.3%	-1.6%	-3.2%	-6.0%	-9.8%
7	-1.8%	-1.2%	-4.6%	-7.0%	-9.9%
6	-2.1%	-4.9%	-2.6%	-7.3%	-9.6%
5	-3.1%	-6.1%	-7.8%	-4.4%	-10.0%
4	-10.6%	-20.8%	-17.9%	-19.0%	-19.5%

Table D.2: Face attributes: Age precision at difference of 5 years for quantized model (varying weight and activation bit depths) compared with floating point.