Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

Benoit Jacob Skirmantas Kligys Bo Chen Menglong Zhu
Matthew Tang Andrew Howard Hartwig Adam Dmitry Kalenichenko
{benoitjacob,skligys,bochen,menglong,
mttang,howarda,hadam,dkalenichenko}@google.com
Google Inc.

Abstract

The rising popularity of intelligent mobile devices and the daunting computational cost of deep learning-based models call for efficient and accurate on-device inference schemes. We propose a quantization scheme that allows inference to be carried out using integer-only arithmetic, which can be implemented more efficiently than floating point inference on commonly available integer-only hardware. We also co-design a training procedure to preserve end-to-end model accuracy post quantization. As a result, the proposed quantization scheme improves the tradeoff between accuracy and on-device latency. The improvements are significant even on MobileNets, a model family known for run-time efficiency, and are demonstrated in ImageNet classification and COCO detection on popular CPUs.

1. Introduction

Current state-of-the-art Convolutional Neural Networks (CNNs) are not well suited for use on mobile devices. Since the advent of AlexNet [20], modern CNNs have primarily been appraised according to classification / detection accuracy. Thus network architectures have evolved without regard to model complexity and computational efficiency. On the other hand, successful deployment of CNNs on mobile platforms such as smartphones, AR/VR devices (HoloLens, Daydream), and drones require small model sizes to accommodate limited on-device memory, and low latency to maintain user engagement. This has led to a burgeoning field of research that focuses on reducing the model size and inference time of CNNs with minimal accuracy losses.

Approaches in this field roughly fall into two categories. The first category, exemplified by MobileNet [10], SqueezeNet [16], ShuffleNet [32], and DenseNet [11], designs novel network architectures that exploit computation / memory efficient operations. The second category quantizes the weights and / or activations of a CNN from 32 bit floating point into lower bit-depth representations. This methodology, embraced by approaches such as Ternary weight networks (TWN [22]), Binary Neural Networks (BNN [14]), XNOR-net [27], and more [8, 21, 26, 33, 34, 35], is the focus of our investigation. Despite their abundance, current quantization approaches are lacking in two respects when it comes to trading off latency with accuracy.

First, prior approaches have not been evaluated on a reasonable baseline architecture. The most common baseline architectures, AlexNet [20], VGG [28] and GoogleNet [29], are all over-parameterized by design in order to extract marginal accuracy improvements. Therefore, it is easy to obtain sizable compression of these architectures, reducing quantization experiments on these architectures to proof-of-concepts at best. Instead, a more meaningful challenge would be to quantize model architectures that are already efficient at trading off latency with accuracy, e.g. MobileNets.

Second, many quantization approaches do not deliver verifiable efficiency improvements on real hardware. Approaches that quantize only the weights ([2, 4, 8, 33]) are primarily concerned with on-device storage and less with computational efficiency. Notable exceptions are binary, ternary and bit-shift networks [14, 22, 27]. These latter approaches employ weights that are either 0 or powers of 2, which allow multiplication to be implemented by bit shifts. However, while bit-shifts can be efficient in custom hardware, they provide little benefit on existing hardware with multiply-add instructions that, when properly used (i.e. pipelined), are not more expensive than additions alone. Moreover, multiplications are only expensive if the operands are wide, and the need to avoid multiplications diminishes with bit depth once both weights and activations are quantized. Notably, these approaches rarely provide on-device measurements to verify the promised timing improvements. More runtime-friendly approaches quantize both the weights and the activations into 1 bit representa-
Figure 1.1: Integer-arithmetic-only quantization. (a) Integer-arithmetic-only inference of a convolution layer. The input and output are represented as 8-bit integers according to equation 1. The convolution involves 8-bit integer operands and a 32-bit integer accumulator. The bias addition involves only 32-bit integers (section 2.4). The ReLU6 nonlinearity only involves 8-bit integer arithmetic. (b) Training with simulated quantization of the convolution layer. All variables and computations are carried out using 32-bit floating-point arithmetic. Weight quantization (“wt quant”) and activation quantization (“act quant”) nodes are injected into the computation graph to simulate the effects of quantization of the variables (section 3). The resultant graph approximates the integer-arithmetic-only computation graph in panel a), while being trainable using conventional optimization algorithms for floating point models. (c) Our quantization scheme benefits from the fast integer-arithmetic circuits in common CPUs to deliver an improved latency-vs-accuracy tradeoff (section 4). The figure compares integer quantized MobileNets [10] against floating point baselines on ImageNet [3] using Qualcomm Snapdragon 835 LITTLE cores.

Our work draws inspiration from [7], which leverages low-precision fixed-point arithmetic to accelerate the training speed of CNNs, and from [31], which uses 8-bit fixed-point arithmetic to speed up inference on x86 CPUs. Our quantization scheme focuses instead on improving the inference speed vs accuracy tradeoff on mobile CPUs.

2. Quantized Inference

2.1. Quantization scheme

In this section, we describe our general quantization scheme, that is, the correspondence between the bit-representation of values (denoted q below, for “quantized value”) and their interpretation as mathematical real numbers (denoted r below, for “real value”). Our quantization scheme is implemented using integer-only arithmetic during inference and floating-point arithmetic during training, with both implementations maintaining a high degree of correspondence with each other. We achieve this by first providing a mathematically rigorous definition of our quantization scheme, and separately adopting this scheme for both integer-arithmetic inference and floating-point training.

1The quantization scheme described here is the one adopted in TensorFlow Lite [5] and we will refer to specific parts of its code to illustrate aspects discussed below.

2We had earlier described this quantization scheme in the documentation of gemmlowp [18]. That page may still be useful as an alternate treatment of some of the topics developed in this section, and for its self-contained example code.
A basic requirement of our quantization scheme is that it permits efficient implementation of all arithmetic using only integer arithmetic operations on the quantized values (we eschew implementations requiring lookup tables because these tend to perform poorly compared to pure arithmetic on SIMD hardware). This is equivalent to requiring that the quantization scheme be an affine mapping of integers \( q \) to real numbers \( r \), i.e. of the form

\[
    r = S(q - Z) \quad (1)
\]

for some constants \( S \) and \( Z \). Equation (1) is our quantization scheme and the constants \( S \) and \( Z \) are our quantization parameters. Our quantization scheme uses a single set of quantization parameters for all values within each activation array and within each weights array; separate arrays use separate quantization parameters.

For 8-bit quantization, \( q \) is quantized as an 8-bit integer (for \( B \)-bit quantization, \( q \) is quantized as an \( B \)-bit integer). Some arrays, typically bias vectors, are quantized as 32-bit integers, see section 2.4.

The constant \( S \) (for “scale”) is an arbitrary positive real number. It is typically represented in software as a floating-point quantity, like the real values \( r \). Section 2.2 describes methods for avoiding the representation of such floating-point quantities in the inference workload.

The constant \( Z \) (for “zero-point”) is of the same type as quantized values \( q \), and is in fact the quantized value \( q \) corresponding to the real value 0. This allows us to automatically meet the requirement that the real value \( r = 0 \) be exactly representable by a quantized value. The motivation for this requirement is that efficient implementation of neural network operators often requires zero-padding of arrays around boundaries.

Our discussion so far is summarized in the following quantized buffer data structure\(^3\), with one instance of such a buffer existing for each activation array and weights array in a neural network. We use C++ syntax because it allows the unambiguous conveyance of types.

```cpp
template<typename QType> // e.g. QType=uint8
struct QuantizedBuffer {
    vector<QType> q; // the quantized values
    float S; // the scale
    QType Z; // the zero-point
};
```

2.2. Integer-arithmetic-only matrix multiplication

We now turn to the question of how to perform inference using only integer arithmetic, i.e. how to use Equation (1) to translate real-numbers computation into quantized-values computation, and how the latter can be designed to involve only integer arithmetic even though the scale values \( S \) are not integers.

Consider the multiplication of two square \( N \times N \) matrices of real numbers, \( r_1 \) and \( r_2 \), with their product represented by \( r_3 = r_1 r_2 \). We denote the entries of each of these matrices \( r_\alpha \) (\( \alpha = 1, 2 \) or \( 3 \)) as \( r_\alpha^{(i,j)} \) for \( 1 \leq i, j \leq N \), and the quantization parameters with which they are quantized as \( (S_\alpha, Z_\alpha) \). We denote the quantized entries by \( q_\alpha^{(i,j)} \).

Equation (1) then becomes:

\[
    r_\alpha^{(i,j)} = S_\alpha (q_\alpha^{(i,j)} - Z_\alpha). \quad (2)
\]

From the definition of matrix multiplication, we have

\[
    S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1)S_2(q_2^{(j,k)} - Z_2), \quad (3)
\]

which can be rewritten as

\[
    q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2), \quad (4)
\]

where the multiplier \( M \) is defined as

\[
    M := \frac{S_1S_2}{S_3}. \quad (5)
\]

In Equation (4), the only non-integer is the multiplier \( M \). As a constant depending only on the quantization scales \( S_1, S_2, S_3 \), it can be computed offline. We empirically find it to always be in the interval \((0, 1)\), and can therefore express it in the normalized form

\[
    M = 2^{-n}M_0 \quad (6)
\]

where \( M_0 \) is in the interval \([0.5, 1)\) and \( n \) is a non-negative integer. The normalized multiplier \( M_0 \) now lends itself well to being expressed as a fixed-point multiplier (e.g. int16 or int32 depending on hardware capability). For example, if int32 is used, the integer representing \( M_0 \) is the int32 value nearest to \( 2^{31}M_0 \). Since \( M_0 \geq 0.5 \), this value is always at least \( 2^{30} \) and will therefore always have at least 30 bits of relative accuracy. Multiplication by \( M_0 \) can thus be implemented as a fixed-point multiplication\(^4\). Meanwhile, multiplication by \( 2^{-n} \) can be implemented with an efficient bit-shifting, albeit one that needs to have correct round-to-nearest behavior, an issue that we return to in Appendix B.

2.3. Efficient handling of zero-points

In order to efficiently implement the evaluation of Equation (4) without having to perform \( 2N^3 \) subtractions and

\(^3\)The actual data structures in the TensorFlow Lite \([5]\) Converter are QuantizationParams and Array in this header file. As we discuss in the next subsection, this data structure, which still contains a floating-point quantity, does not appear in the actual quantized on-device inference code.

\(^4\)The computation discussed in this section is implemented in TensorFlow Lite \([5]\) reference code for a fully-connected layer.
without having to expand the operands of the multiplication into 16-bit integers, we first notice that by distributing the multiplication in Equation (4), we can rewrite it as

\[
q_3^{(i,k)} = Z_3 + M \left( N Z_1 Z_2 - Z_1 a_2^{(k)} \right) - Z_2 a_1^{(i)} + \sum_{j=1}^{N} q_1^{(i,j)} q_2^{(j,k)} \tag{7}
\]

where

\[
a_2^{(k)} := \sum_{j=1}^{N} q_2^{(j,k)}, \quad a_1^{(i)} := \sum_{j=1}^{N} q_1^{(i,j)}. \tag{8}
\]

Each \(a_2^{(k)}\) or \(a_1^{(i)}\) takes only \(N\) additions to compute, so they collectively take only \(2N^2\) additions. The rest of the cost of the evaluation of (7) is almost entirely concentrated in the core integer matrix multiplication accumulation

\[
\sum_{j=1}^{N} q_1^{(i,j)} q_2^{(j,k)} \tag{9}
\]

which takes \(2N^3\) arithmetic operations; indeed, everything else involved in (7) is \(O(N^2)\) with a small constant in the \(O\). Thus, the expansion into the form (7) and the factored-out computation of \(a_2^{(k)}\) and \(a_1^{(i)}\) enable low-overhead handling of arbitrary zero-points for anything but the smallest values of \(N\), reducing the problem to the same core integer matrix multiplication accumulation (9) as we would have to compute in any other zero-points-free quantization scheme.

2.4. Implementation of a typical fused layer

We take the \(q_1\) matrix to be the weights, and the \(q_2\) matrix to be the activations. Both the weights and activations are of type \texttt{uint8} (we could have equivalently chosen \texttt{int8}, with suitably modified zero-points). Accumulating products of \texttt{uint8} values requires a 32-bit accumulator, and we choose a signed type for the accumulator for a reason that will soon become clear. The sum in (9) is thus of the form:

\[
\texttt{int32} += \texttt{uint8} \times \texttt{uint8}. \tag{10}
\]

In order to have the quantized bias-addition be the addition of an int32 bias into this int32 accumulator, the bias-vector is quantized such that: it uses int32 as its quantized data type; it uses 0 as its quantization zero-point \(Z_{bias}\); and its quantization scale \(S_{bias}\) is the same as that of the accumulators, which is the product of the scales of the weights and of the input activations. In the notation of section 2.3,

\[
S_{bias} = S_1 S_2, \quad Z_{bias} = 0. \tag{11}
\]

Although the bias-vectors are quantized as 32-bit values, they account for only a tiny fraction of the parameters in a neural network. Furthermore, the use of higher precision for bias vectors meets a real need: as each bias-vector entry is added to many output activations, any quantization error in the bias-vector tends to act as an overall bias (i.e. an error term with nonzero mean), which must be avoided in order to preserve good end-to-end neural network accuracy\(^6\).

With the final value of the int32 accumulator, there remain three things left to do: scale down to the final scale used by the 8-bit output activations, \textit{cast down} to uint8 and \textit{apply the activation function} to yield the final 8-bit output activation.

The down-scaling corresponds to multiplication by the multiplier \(M\) in equation (7). As explained in section 2.2, it is implemented as a fixed-point multiplication by a normalized multiplier \(M_0\) and a rounding bit-shift. Afterwards, we perform a saturating cast to uint8 and to the range \([0, 255]\).

We focus on activation functions that are mere clamps, e.g. ReLU, ReLU6. Mathematical functions are discussed in Appendix A.1 and we do not currently fuse them into such layers. Thus, the only thing that our fused activation functions need to do is to further clamp the uint8 value to some sub-interval of \([0, 255]\) before storing the final uint8 output activation. In practice, the quantized training process (section 3) tends to learn to make use of the whole output uint8 \([0, 255]\) interval so that the activation function no longer does anything, its effect being subsumed in the clamping to \([0, 255]\) implied in the saturating cast to uint8.

3. Training with simulated quantization

A common approach to training quantized networks is to train in floating point and then quantize the resulting

\(^3\)The discussion in this section is implemented in TensorFlow Lite [5] for e.g. a Convolutional operator (reference code is self-contained, optimized code calls into gemmlowp [18]).

\(^6\)The quantization of bias-vectors discussed here is implemented here in the TensorFlow Lite [5] Converter.
weights (sometimes with additional post-quantization training for fine-tuning). We found that this approach works sufficiently well for large models with considerable representational capacity, but leads to significant accuracy drops for small models. Common failure modes for simple post-training quantization include: 1) large differences (more for small models. Common failure modes for simple post-
sentational capacity, but leads to significant accuracy drops
sufficiently well for large models with considerable repre-
ing for fine-tuning). We found that this approach works
weights (sometimes with additional post-quantization train-
amounts. The forward propagation pass however
simu-

## 3.1. Learning quantization ranges

Quantization ranges are treated differently for weight quantization vs. activation quantization:

- For weights, the basic idea is simply to set $a := \min w$, $b := \max w$. We apply a minor tweak to this so that the weights, once quantized as int8 values, only range in $[-127, 127]$ and never take the value $-128$, as this enables a substantial optimization opportunity (for more details, see Appendix B).

- For activations, ranges depend on the inputs to the network. To estimate the ranges, we collect $[a; b]$ ranges seen on activations during training and then aggregate them via exponential moving averages (EMA) with the smoothing parameter being close to 1 so that observed ranges are smoothed across thousands of training steps. Given significant delay in the EMA updating activation ranges when the ranges shift rapidly, we found it useful to completely disable activation quantization at the start of training (say, for 50 thousand to 2 million steps). This allows the network to enter a more stable state where activation quantization ranges do not exclude a significant fraction of values.

In both cases, the boundaries $[a; b]$ are nudged so that value 0.0 is exactly representable as an integer $z(a, b, n)$ after quantization. As a result, the learned quantization parameters map to the scale $S$ and zero-point $Z$ in equation 1:

$$
S = s(a, b, n), \quad Z = z(a, b, n)
$$

Below we depict simulated quantization assuming that the computations of a neural network are captured as a TensorFlow graph [1]. A typical workflow is described in Algorithm 1. Optimization of the inference graph by fusing

### Algorithm 1 Quantized graph training and inference

1: Create a training graph of the floating-point model.
2: Insert fake quantization TensorFlow operations in locations where tensors will be downcasted to fewer bits during inference according to equation 12.
3: Train in simulated quantized mode until convergence.
4: Create and optimize the inference graph for running in a low bit inference engine.
5: Run inference using the quantized inference graph.

and removing operations is outside the scope of this paper. Source code for graph modifications (inserting fake quantization operations, creating and optimizing the inference graph) and a low bit inference engine has been opensourced with TensorFlow contributions in [19].

Figure 1.1a and b illustrate TensorFlow graphs before and after quantization for a simple convolutional layer. Illustrations of the more complex convolution with a bypass connection in figure C.3 can be found in figure C.4.

Note that the biases are not quantized because they are represented as 32-bit integers in the inference process, with
a much higher range and precision compared to the 8 bit weights and activations. Furthermore, quantization parameters used for biases are inferred from the quantization parameters of the weights and activations. See section 2.4.

Typical TensorFlow code illustrating use of [19] follows:

```python
from tf.contrib.quantize import quantize_graph as qg
g = tf.Graph()
with g.as_default():
    output, total_loss, optimizer, train_tensor = ...
if is_training:
    quantized_graph = qg.create_training_graph(g)
else:
    quantized_graph = qg.create_eval_graph(g)
# Train or evaluate quantized_graph.
```

3.2. Batch normalization folding

For models that use batch normalization (see [17]), there is additional complexity: the training graph contains batch normalization as a separate block of operations, whereas the inference graph has batch normalization parameters “folded” into the convolutional or fully connected layer’s weights and biases, for efficiency. To accurately simulate quantization effects, we need to simulate this folding, and quantify weights after they have been scaled by the batch normalization parameters.

We do so with the following:

$$w_{\text{fold}} := \frac{\gamma w}{\sqrt{\sigma^2_B + \varepsilon}}.$$  \hfill (14)

Here $\gamma$ is the batch normalization’s scale parameter, $\sigma^2_B$ is the estimate of the variance of convolution results across the batch, and $\varepsilon$ is just a small constant for numerical stability.

After folding, the batch-normalized convolutional layer reduces to the simple convolutional layer depicted in figure 1.1a with the folded weights $w_{\text{fold}}$ and the corresponding folded biases. Therefore the same recipe in figure 1.1b applies. See Appendix for the training graph (figure C.5) for a batch-normalized convolutional layer, the corresponding inference graph (figure C.6), the training graph after batch-norm folding (figure C.7) and the training graph after both folding and quantization (figure C.8).

4. Experiments

We conducted two set of experiments, one showcasing the effectiveness of quantized training (Section 4.1), and the other illustrating the improved latency-vs-accuracy tradeoff of quantized models on common hardware (Section 4.2). The most performance-critical part of the inference workload on the neural networks being benchmarked is matrix multiplication (GEMM). The 8-bit and 32-bit floating-point GEMM inference code uses the gemmlowp library [18] for 8-bit quantized inference, and the Eigen library [6] for 32-bit floating-point inference.

<table>
<thead>
<tr>
<th>ResNet depth</th>
<th>50</th>
<th>100</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point accuracy</td>
<td>76.4%</td>
<td>78.0%</td>
<td>78.8%</td>
</tr>
<tr>
<td>Integer-quantized accuracy</td>
<td>74.9%</td>
<td>76.6%</td>
<td>76.7%</td>
</tr>
</tbody>
</table>

Table 4.1: ResNet on ImageNet: Floating-point vs quantized network accuracy for various network depths.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>BWN</th>
<th>TWN</th>
<th>INQ</th>
<th>FGQ</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight bits</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Activation bits</td>
<td>float32</td>
<td>float32</td>
<td>float32</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Accuracy</td>
<td>68.7%</td>
<td>72.5%</td>
<td>74.8%</td>
<td>70.8%</td>
<td>74.9%</td>
</tr>
</tbody>
</table>

Table 4.2: ResNet on ImageNet: Accuracy under various quantization schemes, including binary weight networks (BWN [21, 15]), ternary weight networks (TWN [21, 22]), incremental network quantization (INQ [33]) and fine-grained quantization (FGQ [26]).

4.1. Quantized training of Large Networks

We apply quantized training to ResNets [9] and InceptionV3 [30] on the ImageNet dataset. These popular networks are too computationally intensive to be deployed on mobile devices, but are included for comparison purposes. Training protocols are discussed in Appendix D.1 and D.2.

4.1.1 ResNets

We compare floating-point vs integer-quantized ResNets in table 4.1. Accuracies of integer-only quantized networks are within 2% of their floating-point counterparts.

We also list ResNet50 accuracies under different quantization schemes in table 4.2. As expected, integer-only quantization outperforms FGQ [26], which uses 2 bits for weight quantization. INQ [33] (5-bit weight floating-point activation) achieves a similar accuracy as ours, but we provide additional run-time improvements (see section 4.2).

4.1.2 Inception v3 on ImageNet

We compare the Inception v3 model quantized into 8 and 7 bits, respectively. 7-bit quantization is obtained by setting the number of quantization levels in equation 12 to $n = 2^7$.

We additionally probe the sensitivity of activation quantization by comparing networks with ReLU6 and ReLU. The training protocol is in Appendix D.2.

Table 4.3 shows that 7-bit quantized training produces model accuracies close to that of 8-bit quantized training, and quantized models with ReLU6 have less accuracy degradation. The latter can be explained by noticing that ReLU6 introduces the interval $[0, 6]$ as a natural range for activations, while ReLU allows activations to take values from a possibly larger interval, with different ranges in dif-
different channels. Values in a fixed range are easier to quantize with high precision.

### 4.2. Quantization of MobileNets

MobileNets are a family of architectures that achieve a state-of-the-art tradeoff between on-device latency and ImageNet classification accuracy. In this section we demonstrate how integer-only quantization can further improve the tradeoff on common hardware.

#### 4.2.1 ImageNet

We benchmarked the MobileNet architecture with varying depth-multipliers (DM) and resolutions on ImageNet on three types of Qualcomm cores, which represent three different micro-architectures: 1) Snapdragon 835 LITTLE core, (figure. 1.1c), a power-efficient processor found in Google Pixel 2; 2) Snapdragon 835 big core (figure. 1.1), a high-performance core employed by Google Pixel 2; and 3) Snapdragon 821 big core (figure. 1.2), a high-performance core used in Google Pixel 1.

<table>
<thead>
<tr>
<th>Act. type</th>
<th>accuracy</th>
<th>recall 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReLU6 float</td>
<td>78.4%</td>
<td>94.1%</td>
</tr>
<tr>
<td>8 bits</td>
<td>75.4%</td>
<td>92.5%</td>
</tr>
<tr>
<td>7 bits</td>
<td>75.0%</td>
<td>92.4%</td>
</tr>
<tr>
<td>ReLU float</td>
<td>78.3%</td>
<td>94.2%</td>
</tr>
<tr>
<td>8 bits</td>
<td>74.2%</td>
<td>92.2%</td>
</tr>
<tr>
<td>7 bits</td>
<td>73.7%</td>
<td>92.0%</td>
</tr>
</tbody>
</table>

Table 4.3: Inception v3 on ImageNet: Accuracy and recall 5 comparison of floating point and quantized models.

Figure 4.2: Latency-vs-accuracy tradeoff of float vs. integer-only MobileNets on ImageNet using Snapdragon 821 big core.

Integer-only quantized MobileNets achieve higher accuracies than floating-point MobileNets given the same run-time budget. The accuracy gap is quite substantial (~10%) for Snapdragon 835 LITTLE cores at the 33ms latency needed for real-time (30 fps) operation. While most of the quantization literature focuses on minimizing accuracy loss for a given architecture, we advocate for a more comprehensive latency-vs-accuracy tradeoff as a better measure. Note that this tradeoff depends critically on the relative speed of floating-point vs integer-only arithmetic in hardware. Floating-point computation is better optimized in the Snapdragon 821, for example, resulting in a less noticeable reduction in latency for quantized models.

#### 4.2.2 COCO

We evaluated quantization in the context of mobile real time object detection, comparing the performance of quantized 8-bit and float models of MobileNet SSD [10, 25] on the COCO dataset [24]. We replaced all the regular convolutions in the SSD prediction layers with separable convolutions (depthwise followed by \(1 \times 1\) projection). This modification is consistent with the overall design of MobileNets and makes them more computationally efficient. We utilized the Open Source TensorFlow Object Detection API [12] to train and evaluate our models. The training protocol is described in Appendix D.3. We also delayed quantization for 500 thousand steps (see section 3.1), finding that it significantly decreases the time to convergence.

Table 4.4 shows the latency-vs-accuracy tradeoff between floating-point and integer-quantized models. Latency was measured on a single thread using Snapdragon 835 cores (big and LITTLE). Quantized training and inference results in up to a 50% reduction in running time, with a minimal loss in accuracy (~1.8% relative).

#### 4.2.3 Face detection and attribute classification

To better examine quantized MobileNet SSD on a smaller scale, we benchmarked face detection on the face attribute
Table 4.4: Object detection speed and accuracy on COCO dataset of floating point and integer-only quantized models. Latency (ms) is measured on Qualcomm Snapdragon 835.

<table>
<thead>
<tr>
<th>DM</th>
<th>Type</th>
<th>mAP</th>
<th>LITTLE (ms)</th>
<th>big (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>floats</td>
<td>22.1</td>
<td>778</td>
<td>370</td>
</tr>
<tr>
<td></td>
<td>8 bits</td>
<td>21.7</td>
<td>687</td>
<td>272</td>
</tr>
<tr>
<td>50%</td>
<td>floats</td>
<td>16.7</td>
<td>270</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>8 bits</td>
<td>16.6</td>
<td>146</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 4.5: Face detection accuracy of floating point and integer-only quantized models. The reported precision / recall is averaged over different precision / recall values where an IOU of $x$ between the groundtruth and predicted windows is considered a correct detection, for $x \in \{0.5, 0.55, \ldots, 0.95\}$. Latency (ms) of floating point and quantized models are reported on Qualcomm Snapdragon 835 using a single LITTLE and big core, respectively.

<table>
<thead>
<tr>
<th>DM</th>
<th>Type</th>
<th>Precision</th>
<th>Recall</th>
<th>LITTLE (ms)</th>
<th>big (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>floats</td>
<td>68%</td>
<td>76%</td>
<td>711</td>
<td>337</td>
</tr>
<tr>
<td></td>
<td>8 bits</td>
<td>66%</td>
<td>75%</td>
<td>372</td>
<td>154</td>
</tr>
<tr>
<td>50%</td>
<td>floats</td>
<td>65%</td>
<td>70%</td>
<td>233</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>8 bits</td>
<td>62%</td>
<td>70%</td>
<td>134</td>
<td>56</td>
</tr>
<tr>
<td>25%</td>
<td>floats</td>
<td>56%</td>
<td>64%</td>
<td>100</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>8 bits</td>
<td>54%</td>
<td>63%</td>
<td>67</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 4.6: Face attributes: relative average category precision of integer-quantized MobileNets (varying weight and activation bit depths) compared with floating point.

Ablation study To understand performance sensitivity to the quantization scheme, we further evaluate quantized training with varying weight and activation quantization bit depths. The degradation in average precision for binary attributes and age precision relative to the floating-point baseline are shown in Tables 4.6 and Appendix D.2, respectively. The tables suggest that 1) weights are more sensitive to reduced quantization bit depth than activations, 2) 8 and 7-bit quantized models perform similarly to floating point models, and 3) when the total bit-depths are equal, it is better to keep weight and activation bit depths the same.

5. Discussion

We propose a quantization scheme that relies only on integer arithmetic to approximate the floating-point computations in a neural network. Training that simulates the effect of quantization helps to restore model accuracy to near-identical levels as the original. In addition to the $4 \times$ reduction of model size, inference efficiency is improved via ARM NEON-based implementations. The improvement advances the state-of-the-art tradeoff between latency on common ARM CPUs and the accuracy of popular computer vision models. The synergy between our quantization scheme and efficient architecture design suggests that integer-arithmetic-only inference could be a key enabler that propels visual recognition technologies into the real-time and low-end phone market.
References


