Quantization of LSTM Layers by a Heuristic Approach

Alexander Samuelsson, Johan Malm, Albert Seward

Imagimob, Embassy House, Medborgarplatsen 3, 118 26, Stockholm

Introduction

Long short-term memory cells (LSTMs) introduced by Hochreiter and Schmidhuber in 1997 [1] have been used successfully in state-of-the-art models that treat time-series data and can be credited with the success for solving the difficult problems in language modelling, translation, robotics, and gaming. Their idea was to equip traditional Recurrent Neural Networks (RNNs) with "forget gates" so that information could optionally flow through the network unaltered, which solved the so-called vanishing gradient problem. This allowed the network to be trained on arbitrarily long sequences. Implementation of neural networks on digital computers can often be done using floating-point numbers, since most modern PCs, smartphones, tablets and more powerful microcontrollers (MCUs) for embedded systems have a floating-point unit (FPU). But for the tiniest MCUs on the market (e.g., ARM's M0-M3 cores) the best performance is achieved by transforming the data to integer numbers and performing all the calculations with integer arithmetic.

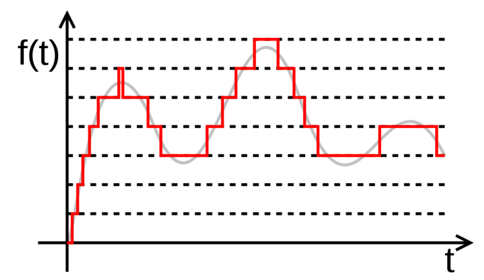

Fig. 1. Quantizing a continuous signal.

By specifying the integer bit width to be 16 or 8 the memory consumption can be reduced as well. The conversion from floating-point numbers and operations to integers requires great care, to not introduce large errors in the calculation (Fig. 1). LSTMs are known to be hard to quantize due to their capability to remember features over of long sequences of inputs, which could easily amplify small errors if the quantization procedure is not done correctly.

Post-training Quantization

Post-training quantization is to start out as usual with a neural network defined in floating-point numbers, train it, and then quantize the end-result. It is a flexible solution, as the trained model can be used both on platforms with and without an FPU. In *Full integer* quantization, the idea is to replace all the floating-point values in the already trained model with integers, typically 8-bit integers with range [-128, 127] or 16-bit integers with range [-32 768, 32 767]. These ranges are less compared to what a 32-bit floating-point value can represent and therefore, care is needed for this transformation. Simple rounding of the floating-point range [-2, 5] generates a large quantization error, as shown below in Fig. 2.

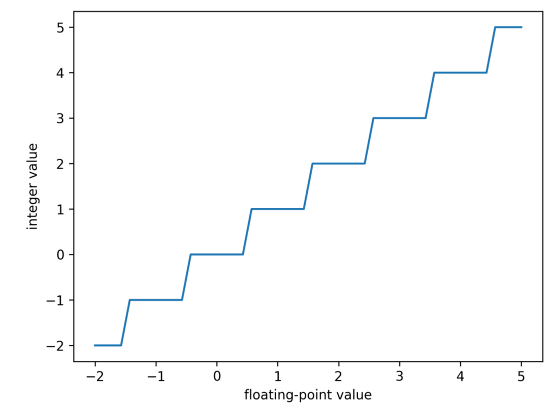

Fig. 2. Simple rounding generates a large quantization error.

Instead, by monitoring the largest and smallest values in the calculations (each tensor of the neural network) for some typical data that will be passed through the network, we use a simple linear relationship $y = kx + m$ to map the floating-point values to integers (the *quantize operation*, $q(\cdot)$):

$$integer\_value = round\left(\frac{float\_value}{scale}\right) + zero\_point \qquad (1)$$

By finding proper values for the scale and zero-point parameters in the above relationship we can make sure that the integer ranges are respected ([-128, 127] in the case of int8), and no overflow will occur. Note that we are free to choose these values, e.g., it is legitimate to keep the minimum value fixed (-128), while changing the scale to something else, and vice versa (keep max as 127 and change the scale) or keep the zero-point and set a new scale. This observation is the foundation of the approach outlined here. Using relation (1) above with proper scale and zero-point values yields a much smaller quantization error as shown in Fig. 3 below.

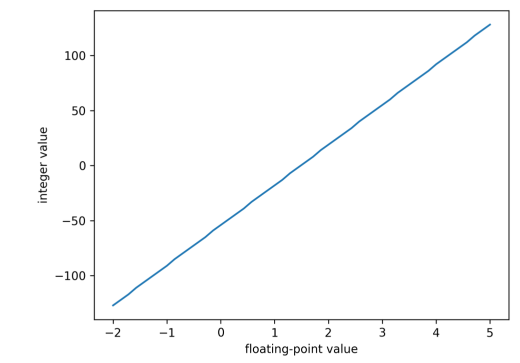

Fig. 3. Working with the full range gives a smaller quantization error.

The transformed values are used in the internal calculations, but in the end, we usually map the values back to physical values that make sense to us, which we do by using the inverse mapping (or the *dequantize operation*, $d(\cdot)$)

$$float\_value = (integer\_value - zero\_point) \cdot scale \qquad (2)$$

Except for transforming the data to integers, we also need to implement integer versions of all the operations in the neural network. In essence, this means that we rewrite the floating-point operations by using the mappings (1) and (2) above, following the idea in [2]. For a simple multiplication between the floating-point values a and b, we get,

$$mulop = a \cdot b = (a_i - z_a) \cdot scale_a \cdot (b_i - z_b) \cdot scale_b = (a_i - z_a) \cdot (b_i - z_b) \cdot scale_{ab} \qquad (3)$$

where,

$$scale_{ab} = scale_a \cdot scale_b.$$

Now, applying (2) to get the integer result of this operation we have,

$$mulop_{int} = round\left(\frac{mulop}{scale_{mulop}}\right) + z_{mulop} = round\left[\frac{(a_i - z_a)\cdot(b_i - z_b)\cdot scale_{ab}}{scale_{mulop}}\right] + z_{mulop} \qquad (4)$$

For Full integer quantization all values in the above expression shall be integers, so that they can efficiently be computed and stored on non-FPU MCUs. Using fixed-point multiplications or shifting for the division these operations can be efficiently handled.

The Nonlinear Integer Optimization Problem

The activation functions in the Imagimob solution are implemented as Lookup tables ("LUTs") to avoid the floating-point operations. Using all this, we can run the calculations as integer-only – all we need to do is to quantize the input data and dequantize the output data, shown in Fig. 4 below.

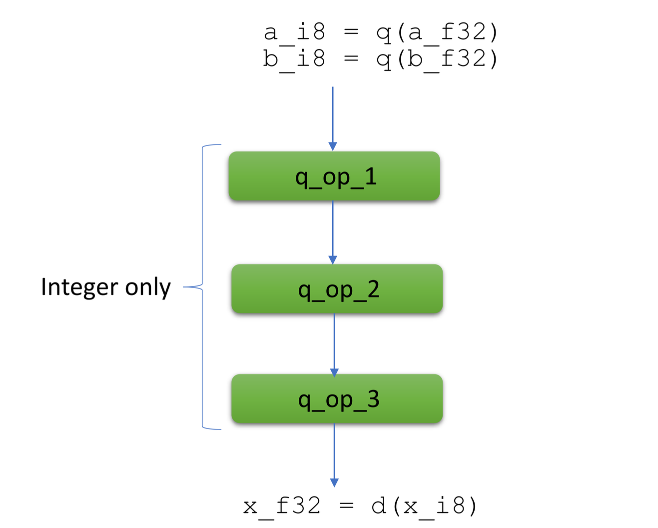

Fig. 4. All internal operations are done in integer arithmetics.

As noted earlier, we are free to choose scale and zero-point values, but we need to select them carefully to keep the quantization error low. As an example, we should choose scale values so that the error after the rounding operation in expression (4) becomes small, namely so that the scale in the numerator and denominator are equal. We can use this as a requirement or "contract" in our implementation:

$$scale_{ab} \overset{!}{=} scale_{mulop}$$

where we mean that the scales shall be made equal. We can achieve this by using the observation made earlier that we can set one of the scales, e.g., setting $scale_{mulop}$ to $scale_{ab}$, and then adjusting the minimum and maximum values for the integer accordingly. Often, however, the different scale and zero-point values from one operation appear in the next operation and become coupled. Noting down all relationships in the compute graph results in a large system of equations with integer solution. Often the system is overdetermined which leads to an optimization problem. The round-operator is often present in the equations, which makes the problem nonlinear.

Imagimob's Solution

Difficult optimization problems can in computer science sometimes be solved using a heuristic approach, which means that with the help of the computer we try to find an approximate solution to a difficult problem. These solutions do not claim that they find the optimal solution to the problem, but the solution can often be good enough to work in practice.

Imagimob's approach consists of:

1) Finding the min and max values on all tensors using representative data.
2) Using the relationships between the scales and zero-points in the compute graph to update the min and max values to fulfil the contracts.
3) Generating quantized C code.
4) Comparing quantized withnon-quantized code to find the quantization error (final test).

The approach is verified by quantizing different types of neural network models consisting of convolutional, dense and LSTM layers with small maximum absolute errors and close to zero classification errors. The CPU gain between non-quantized and a quantized model running on an MCU without FPU is around 6 times, and memory requirements are reduced by 50 % for the quantized model when using 16-bit integer representation.

In Imagimob AI, quantization of a trained model is done with the click of a button in a user-friendly user interface as shown in Fig. 5 below. C code without any floating-point operations is generated, ready for integration in firmware, ready for deployment on a microcontroller.

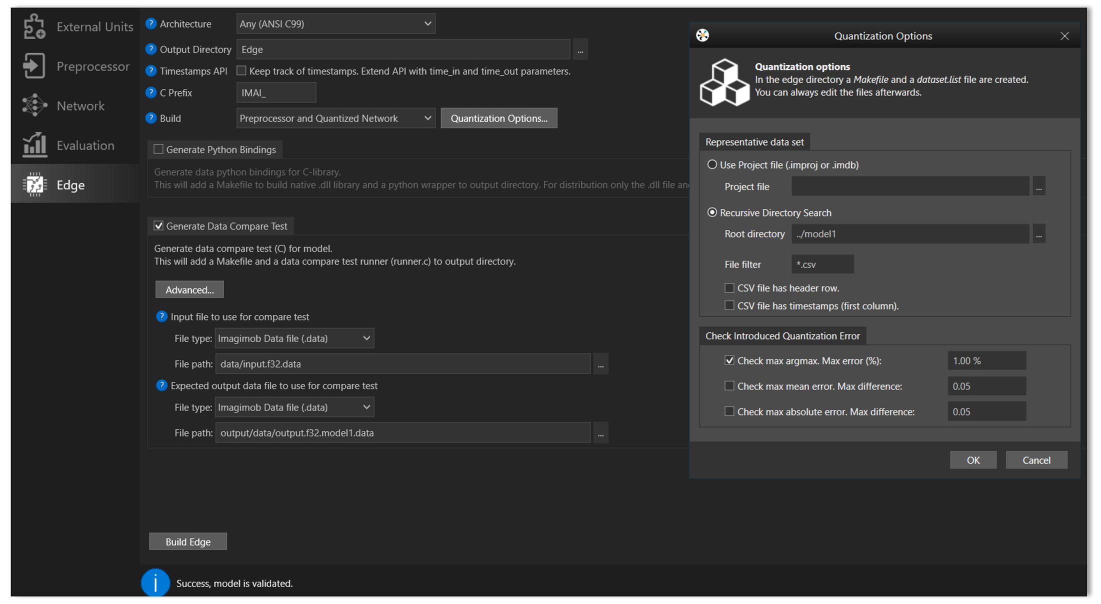

Fig. 5. A trained model is converted to quantized C code with no floating-point calculations.

Interested in learning more? If you find this useful, please visit us at www.imagimob.com or drop us an email info@imagimob.com.

References

[1] Hochreiter, Sepp; Schmidhuber, Jürgen (1997-11-01). Long Short-Term Memory. Neural Computation. 9 (8): 1735–1780

[2] B. Jacob et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. (2017-12-15)