What Is int8 Quantization and Why Is It Popular for Deep Neural Networks?

By Ram Cherukuri, MathWorks

Deep learning deployment on the edge for real-time inference is key to many application areas. It significantly reduces the cost of communicating with the cloud in terms of network bandwidth, network latency, and power consumption.

However, edge devices have limited memory, computing resources, and power. This means that a deep learning network must be optimized for embedded deployment.

int8 quantization has become a popular approach for such optimizations not only for machine learning frameworks like TensorFlow and PyTorch but also for hardware toolchains like NVIDIA® TensorRT and Xilinx® DNNDK—mainly because int8 uses 8-bit integers instead of floating-point numbers and integer math instead of floating-point math, reducing both memory and computing requirements.

These requirements can be considerable. For instance, a relatively simple network like AlexNet is over 200 MB, while a large network like VGG-16 is over 500 MB [1]. Networks of this size cannot fit on low-power microcontrollers and smaller FPGAs.

In this article we take a close look at what it means to represent numbers using 8 bits and see how int8 quantization, in which numbers are represented in integers, can shrink memory and bandwidth usage by as much as 75%.

int8 Representation

We start with a simple example using a VGG16 network consisting of several convolution and ReLU layers and a few fully connected and max pooling layers. First, let's look at how a real-world number (a weight in one of the convolution layers in this case) can be represented with an integer. The function `fi` in MATLAB® gives us the best precision scaling for the weight using 8-bit word length. This means that we get the best precision with a scaling factor of 2^-12 and store it as the bit pattern, 01101110, which represents the integer 110.

$$\text{Real\_number} = \text{stored\_integer} * \text{scaling\_factor}$$
Real_number=stored_integer∗scaling_factor

$$0.0269 = 110 * 2^{-12}$$
0.0269=110∗2−12

The script is as follows:

```
a=net.Layers(9).Weights(1,1,1,8)
```

```
a = single
    0.0269
```

```
b=fi(a,1,8)
```

```
b =
    0.0269

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12
```

```
b.bin
```

```
ans = '01101110'
```

```
b.Value
```

```
ans = '0.02685546875'
```

Now let's consider all the weights of the layer. Using `fi` again, we find that the scaling factor that would give the best precision for all weights in the convolution layer is 2^-8. We visualize the distribution of the dynamic range of the weights in a histogram. The histogram shows that most weights are distributed in the range of 2^-3 and 2^-9 (Figure 1). It also indicates the symmetrical nature of the weight distribution.

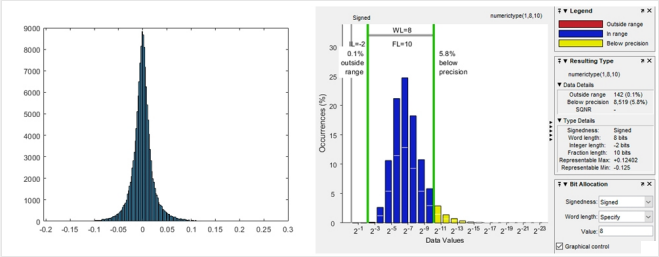


Figure 1. Distribution of weights from the convolution layer in VGG16.

This example showed one way to quantize and represent with 8-bit integers. There are couple of other alternatives:

Selecting a different scaling factor by considering the precision tradeoff. Because we chose a scaling factor of 2^-8, nearly 22% of the weights are below precision. If we chose a scaling factor of 2^-10, only 6% of the weights would be below precision, but 0.1% of the weights would be beyond range. This tradeoff is also illustrated by the error distribution and maximum absolute error (Figure 2). We could choose 16-bit integers, but then we would be using twice as many bits. On the other hand, using 4 bits will lead to significant precision loss or overflows.

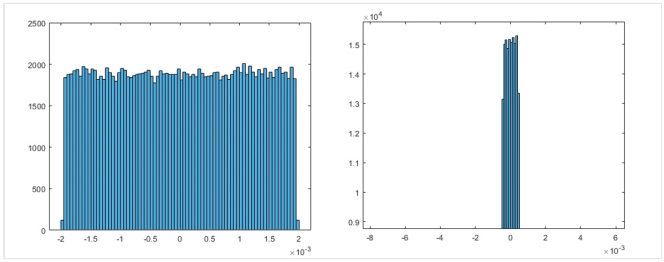


Figure 2. Histogram distribution of the error with scaling factor of 2^-8 (left) and 2^-10 (right) and the corresponding maximum absolute error.

Specifying a bias when calling `fi`, based on the distribution of weights.

$$\text{Real\_number} = \text{stored\_integer} * \text{scaling\_factor} + \text{bias}$$

Real_number=stored_integer∗scaling_factor+bias

You can do a similar analysis for any network—say, ResNet50 or Yolo—and identify an integer data type or scaling factor that can represent the weights and biases within a certain tolerance.

There are two key benefits to representing the data in integers using int8:

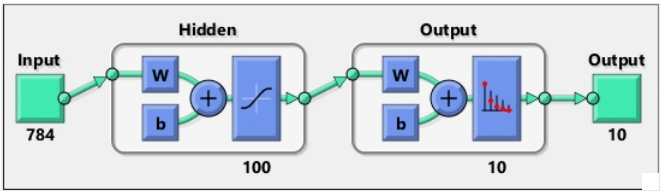
- You can reduce data storage requirements by a factor of 4, since single-precision floating point requires 32 bits to represent a number. The result is a reduction in the memory used to store all the weights and biases and in the power consumed in transferring all the data, since energy consumption is dominated by memory access.
- You can gain further speed-up by using integer computation instead of floating-point math, depending on the target hardware. For instance, you might be able to use half-precision floating point on NVIDIA GPUs. Native half computation is not supported on most CPUs. All targets, however, support integer math, and some also provide certain target-specific intrinsics, such as SIMD support, that can offer significant speed-up when using integers for the underlying computations.

Quantizing a Network to int8

The core idea behind quantization is the resiliency of neural networks to noise; deep neural networks, in particular, are trained to pick up key patterns and ignore noise. This means that the networks can cope with the small changes in the weights and biases of the network resulting from the quantization error—and there is growing body of work indicating the minimal impact of quantization on the accuracy of the overall network. This, coupled with significant reduction in memory footprint, power consumption, and gains in computational speed [1,2], makes quantization an efficient approach for deploying neural networks to embedded hardware.

We'll apply the ideas we discussed above to a network. For simplicity, we'll use a simple network for MNIST digit classification consisting of two layers. Deep networks used for image classification and object detection like VGG16 or ResNet include a wide variety of layers. The convolution layers and the fully connected layers are the most memory-intensive and computationally intensive layers.

Our network mimics the properties of these two layers. We modeled this network in Simulink® so that we can observe the signal flow and take a closer look at the guts of the computation (Figure 3).



In each layer, we are going to replace the weights and biases with scaled int8 integers and then multiply the output of the matrix multiplication with the fixed exponent to rescale. When we validate the predictions of the modified network on the validation data set, the confusion matrix shows that the int8 representation still maintains 95.9% accuracy (Figure 4).

Figure 3. The MNIST network.

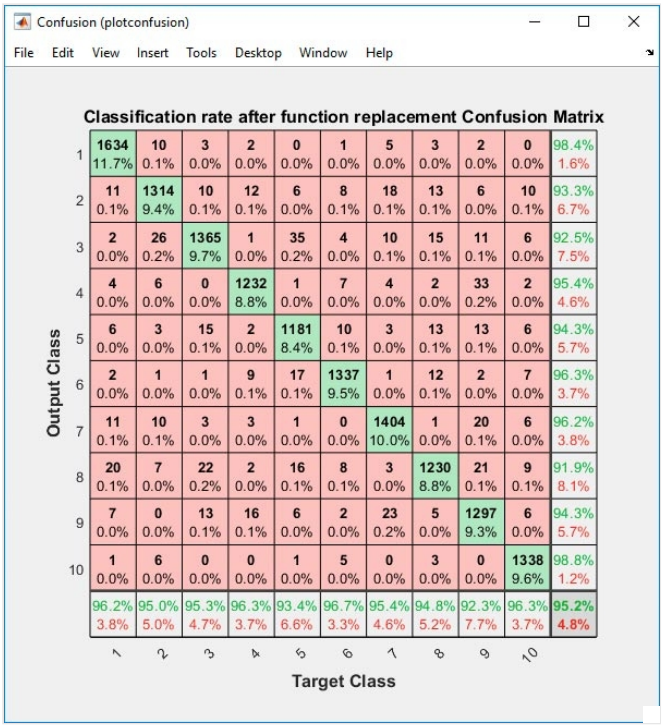


Figure 4. Confusion matrix of the scaled MNIST.

To understand the efficiency gains from quantizing the weights and biases to int8, let's deploy this network to an embedded hardware target—in this example, an ST discovery board (STM32F746G). We are going to analyze two key metrics:

- Memory usage
- Run-time execution performance

When we try to deploy the original model (in double-precision floating point), it does not even fit on the board, and the RAM overflows. The easiest fix is to cast the weights and biases into a single data type. The model now fits on the target hardware, but there is still room for improvement.

We use the scaled model that uses int8 for weight and bias matrices, but the computation is still in single precision (Figure 5).

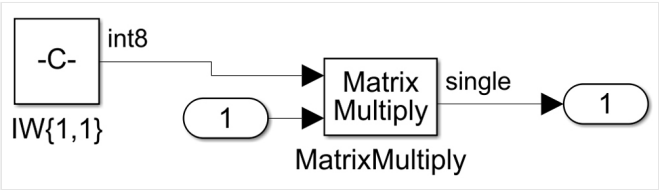


Figure 5. Matrix multiplication in layer1. Weights are int8, but input data is in single precision and the underlying computation is in single precision.

The generated code consumes 4x less memory, as expected (Figure 6).

```
"### Invoking postbuild tool "Executable Size" ..."

"C:/PROGRA~3/MATLAB/SUPPOR~1/R2019a/3P778C~1.INS/GNUARM~1.INS/win/bin/arm-none-eabi-size" ./Pattern.elf

text    data     bss     dec     hex filename
39644  291384   13144  344172    5406c ./Pattern.elf
"### Done invoking postbuild tool."
"### Successfully generated all binary outputs."
```

```
"### Invoking postbuild tool "Executable Size" ..."

"C:/PROGRA~3/MATLAB/SUPPOR~1/R2019a/3P778C~1.INS/GNUARM~1.INS/win/bin/arm-none-eabi-size" ./pattern_scaled.elf

text    data     bss     dec     hex filename
40764   77456   12848  131068   1fffc ./pattern_scaled.elf
"### Done invoking postbuild tool."
"### Successfully generated all binary outputs."
```

Figure 6. Left: Single-precision code. Right: int8 code.

However, the execution time on the discovery board shows that the single-precision variant takes an average of 14.5 milliseconds (around 69 fps) to run while the scaled version is a little slower and takes an average of 19.8 milliseconds (around 50 fps). This might be because of the overhead of the casts to single precision, as we are still doing the computations in single precision (Figure 7).

This example covered just one aspect of quantization—storing weights and biases in int8. By applying the same principle to standard off-the-shelf networks like AlexNet and VGG, you could reduce their memory footprint by 3x [1].

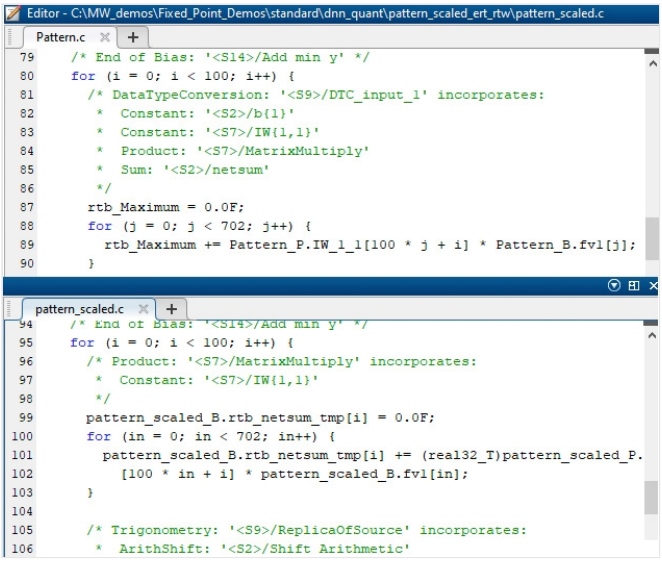


Figure 7. Top: Generated code for single precision. Bottom: Scaled version.

TensorFlow, for instance, enables post-training quantization to 8 bits in two forms—just the weights with floating-point kernels and full integer quantization of both weights and activations [3]. While TensorFlow uses a scaling factor with bias to map to the int8 range [-128, 127], NVIDIA TensorRT avoids the need for bias by encoding the weights to the [-128, 127] range by determining a threshold that minimizes loss of information and saturating the values beyond the threshold range [4].

To leverage the benefits of full integer quantization, we need to scale or convert the input to each layer to an integer type, as well. This requires us to determine the right scaling for the layer's input and then rescale back after the integer multiplication. But will int8 be the right data type, will there be overflows, and will the accuracy of the network be acceptable?

These questions are the essence of fixed-point analysis—and, in fact, the digit recognition documentation example illustrates how to convert an MNIST network using fixed-point data types [5]. Following the steps illustrated in that example, we came up with an 8-bit representation for the weights with drop-in accuracy under 1% (Figure 8).

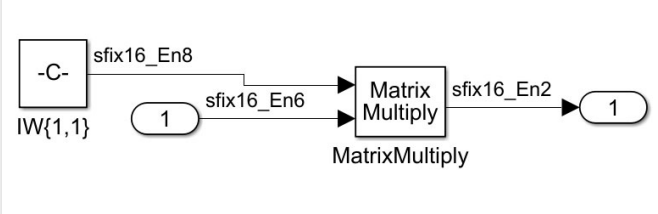


Figure 8. Model converted to use 16-bit word length.

The generated code is not only a quarter the size; it is also faster, 11 milliseconds ~ 90 fps (Figure 9).

Figure 9. Left: Code generated from the fixed-point model. Right: Scaled weights from the first layer of the MNIST network.

Other Quantization Techniques

We have looked at only a few of the many strategies being researched and explored to optimize deep neural networks for embedded deployment. For instance, the weights in the first layer, which is 100x702 in size, consists of only 192 unique values. Other quantization techniques that could be applied include the following:

Use weight sharing by clustering the weights and use Huffman coding to reduce the number of weights [1].

Quantize the weights to the nearest powers of two. This speeds up the computation significantly, as it replaces multiplication operations with much faster arithmetic shift operations.

Replace activation functions with lookup tables to speed up the computation of activation functions such as `tanh` and `exp`. For instance, in the generated code shown in Figure 9, we can further optimize the performance of the network by replacing the `tanh` function with a lookup table.

A deep learning application is more than just the network. You also need to take the pre- and postprocessing logic of the application into consideration. Some of the tools and techniques we discussed have been used for quantizing such algorithms for a couple of decades now. They can be used to quantize not just the network but the entire application.

You can explore all these optimization ideas in MATLAB. You can explore the feasibility and impact of quantizing to further limited precision integer data types like int4 or explore floating-point data types like half precision. The results can be impressive: Song, Huizi, and Willian [1] used a combination of these techniques to reduce the size of networks like AlexNet and VGG by 35x and 49x, respectively.

Published 2019