arm

Packing Neural Networks into End-User Client Devices: How Number Representation Shrinks the Footprint

By Liangzhen Lai, Naveen Suda and Vikas Chandra

Using a mix of floating-point and fixed-point representation in a neural network reduces memory and power requirements, while retaining accuracy, and supports deployment on end-user client devices.

Convolutional Neural Networks (CNNs) are a type of neural network closely associated with computer vision, which is the segment of artificial intelligence that helps computers see, identify, and process images. CNNs use a structure originally inspired by the part of the brain that processes visual information, called the visual cortex, and have been shown to be almost as good as humans when it comes to recognizing faces. In situations that involve noticing fine-grained characteristics – to identify a particular breed of dog or species of bird, for example – CNNs have demonstrated accuracy that's even better than humans.

Outside of computer vision, neural networks (NNs) are used for natural language processing, because they're good at parsing sentences and predicting patterns in word usage. They're used for drug discovery and toxicology, because they're good at predicting the interactions between molecules and biological proteins, and for games like checkers and Go, because they're good at evaluating positions and suggesting next moves. NNs are also used for recommender systems, which are systems that predict the rating or preference users would give to an item – such as a movie, a restaurant, or even a potential romantic partner – based on their previous choices.

The Embedded Challenge

When compared to fully-connected neural networks, CNNs are a good option for embedded systems, since they deliver good performance in a smaller memory footprint. Unlike fully-connected neural networks, CNNs share weights across convolutional layers and, as a result, need less memory to make decisions.

Having said that, though, running a CNN still takes quite a bit of memory and a fairly high degree of computational complexity. AlexNet, for example, which is one of the earliest CNNs developed, requires 1.45 billion operations per input image, with 240 MB weights.

It can be difficult, if not impossible, for an embedded system to meet these kinds of processing and memory requirements. But, by making some changes to the way CNNs operate, we can make them more compact and bring them within reach of more end-user client devices.

CNNs are typically trained on a high-performance CPU/GPU with 32-bit floating-point data. To reduce storage requirements, memory bandwidth, and power consumption, researchers have looked at replacing floating-point representation with fixed-point representation. The results have been generally good, but some network architectures displayed reduced accuracy and needed retraining.

We propose a somewhat different approach. Instead of completely eliminating the floating-point representation, we recommend a mix, with floating-point representation for weights and fixed-point representation for activations. It's an approach that delivers an acceptable degree of accuracy, even with limited bit width, and works consistently across popular CNNs, including AlexNet, SqueezeNet, GoogLeNet, and VGG-16. It also makes the hardware implementation more efficient. Using our approach, we were able to reduce the storage requirements for weights by as much as 36%, and the power consumption of the multiplier by as much as 50%.

This paper gives a high-level introduction to the recommended setup, but before we begin our explanation, we start with a quick refresher on a key point of computer science: the difference between fixed-point and floating-point representations of scale factor.

Fixed-Point and Floating-Point Representations of Scale Factor

In computer science, a scale factor is used to represent a real-world set of numbers on a different scale, to fit in a specific number format. For example, a 16-bit unsigned integer can only represent

the base-10 values from 0 to 65,535. To extend that range and represent the base-10 values from 0 to 131,070, a scale factor of ½ is introduced.

The scale factor increases the number of values that can be represented, but also decreases the precision. For example, with a scale factor of ½, there's no way to represent the number 3. A stored number 1 represents a real-world 2, and a stored 2 represents a real-world 4. The tradeoff is that, by using scale factor, you can represent a wider range of numbers, but with less precision.

Scale factor can be indicated using a fixed-point representation or a floating-point representation.

This is what we do, too, but with a twist. For the accumulation/addition steps, we follow common convention and used fixed-point numbers, which can be wider (that is, have a larger bit width), than either of the inputs. But for the multiplier, we use a multiplication operation with one fixed-point number input, one floating-point number inputs, and a fixed-point number output. We implement this using a multiplier and a shifter, as shown in Figure 4.

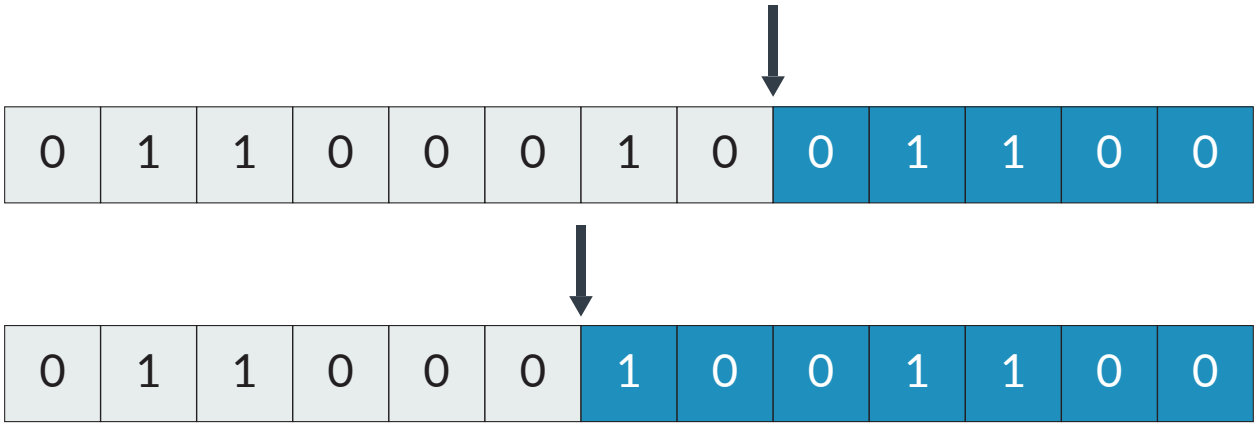


Figure 1. Fixed-Point Representation of Scale Factor

Floating-Point Representation of Scale Factor

With a floating-point representation, the format is similar to scientific notation, which is a form of shorthand that helps scientists handle very large and very small numbers. (In scientific notation, instead of writing 0.000123 we write 1.23 x 10-4).

The IEEE standard 754 defines the format for floating-point representation in computer applications. The representation is given in three parts: the sign (which determines if the number is positive or negative), the mantissa (which determines the most significant digits, or significand), and the exponent (which determines the scale of the value). Figure 2 gives an example.

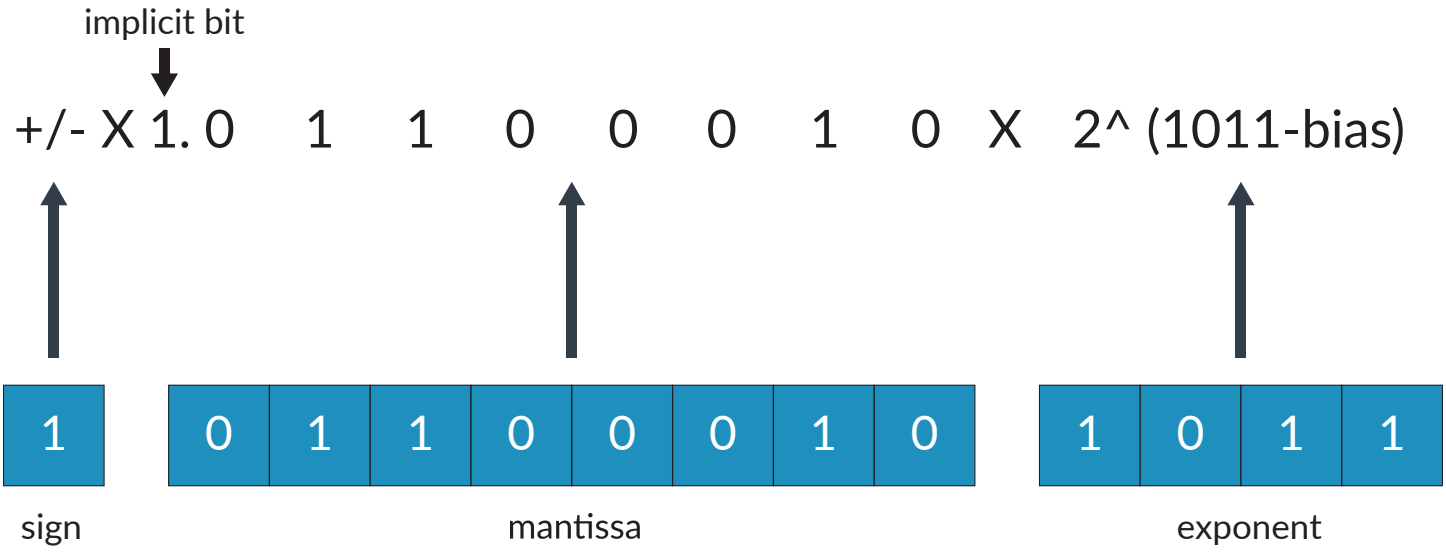


Figure 2. Floating-Point Representation of Scale Factor

According to IEEE 754, the mantissa can assume an implicit bit. This ensures that the value of the mantissa is always between 1 and 2, so the leading bit 1 can be omitted to save storage space. However, such an implicit bit limits the smallest representable number for the mantissa or significand. The exponent is typically represented in an unsigned integer number with a bias. For example, an 8-bit exponent with a bias of 127 can represent the numbers from -127 to 128 (or, more precisely, from 0-127 to 255-127).

In developing our proposed representation scheme for CNN operations, we kept in mind the differences between fixed-point and floating-point representation, and the effect that choosing one over the other has on computational complexity.

In general, it's more efficient, from a hardware standpoint, to do arithmetic with fixed-point representation. But this can limit precision, since the range of values supported by fixed-point representation is narrower. By contrast, performing math operations with floating-point representation may take more resources to implement, but can represent a much wider range of values, and thereby increase precision.

Let's see what that means for CNN operation.

CNN weights

In neural networks, a weight indicates the strength of a connection between units. Weights are used to help the network know what's important and know the degree of influence a particular input should have on the decision to be made.

Here's an example of how weights help a neural network make decisions. Let's say I've developed a neural network to help me decide when I should walk to work. I've programmed it to consider two criteria, or inputs: the day of the week and the weather conditions. Since I only work from Monday to Friday, weekdays will have a stronger weight. That is, Tuesday will be weighed more heavily than Saturday, and is more likely to generate a "yes" answer. Similarly, rainy weather will have a lighter weight than sunny weather and is more likely to generate a "no" answer. Each time I ask the network if I should walk to work, it looks at the specified inputs – the day and the weather – and assigns a weight to each before making a recommendation.

The weights in a neural network can, in theory, be given to the network ahead of time, but in most cases, weights are either unknown or assigned random numbers at the beginning of operation. The object is for the neural network to define weights as it learns, and continue to adjust the weights over time, to improve accuracy. This is something that neural networks are particularly good at.

The range/precision tradeoff we saw with scale factor also applies to weights. A wider range of weight values will yield greater precision, but there's often more computing overhead involved. Consider our walk-to-work network. When assigning a weight to the weather, we might want to have a wide range of options, so the network can distinguish between very sunny, fairly sunny, and partly cloudy, and between mist, light drizzle, and downpour. This is why many neural networks use floating-point representation

for weights – because it supports a broader range. But the added complexity of the mathematics can be a problem for embedded systems.

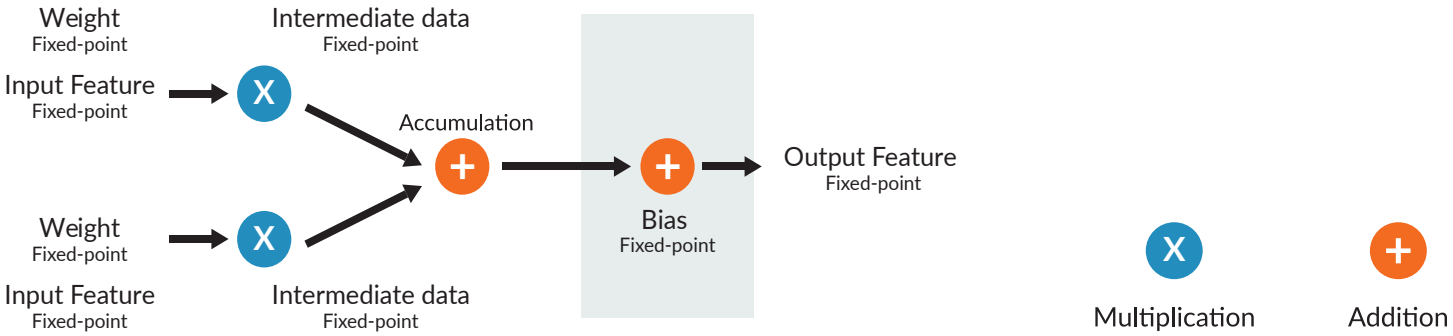
One approach to streamlining a neural network is to use fixed-point representation, instead of floating-point representation, for the weights. When it comes to CNN weights, though, our research shows that using fixed-point representation for weights can be problematic, because different CNN architectures need different bit widths to produce an acceptable level of accuracy. With AlexNet and SqueezeNet, for example, you can get by with a 7-bit fixed-point number, but with GoogLeNet and VGG-16, you need a fixed-point number of at least 10 or 11 bits to get acceptable accuracy. This variation in accuracy makes it hard to create a hardware implementation that works consistently with all the leading CNN architectures.

We analyzed CNN weight distribution and properties and found that representation range is the main factor that determines the accuracy of CNN decision-making, or inference. We also found that floating-point representation is a more efficient, and more consistent representation for CNN weights. For these reasons, we recommend staying with floating-point representation for weights.

.CNN activations

Activations are the mathematical functions that let neural networks make decisions. They are, in essence, the computational "magic" that happens in the hidden layers between the inputs and the outputs.

The two most compute- and data-intensive layers of a CNN are the convolutional and fully-connected layers. The computations in these layers are typically implemented as multiply-accumulate (MAC) operations, which use a mix of multiplication and addition. Figure 3 shows how weights and activations interact in these two



The input features are multiplied with the weights to get the intermediate data, or partial sums. The partial sums are then accumulated to generate the output features. Since fixed-point arithmetic is typically more efficient for hardware implementation, most hardware accelerators implement the MAC operations using fixed-point representation.

This is what we do, too, but with a twist. For the accumulation/addition steps, we follow common convention and used fixed-point numbers, which can be wider (that is, have a larger bit width), than either of the inputs. But for the multiplier, we use a multiplication operation with one fixed-point number input, one floating-point number inputs, and a fixed-point number output. We implement this using a multiplier and a shifter, as shown in Figure 4.

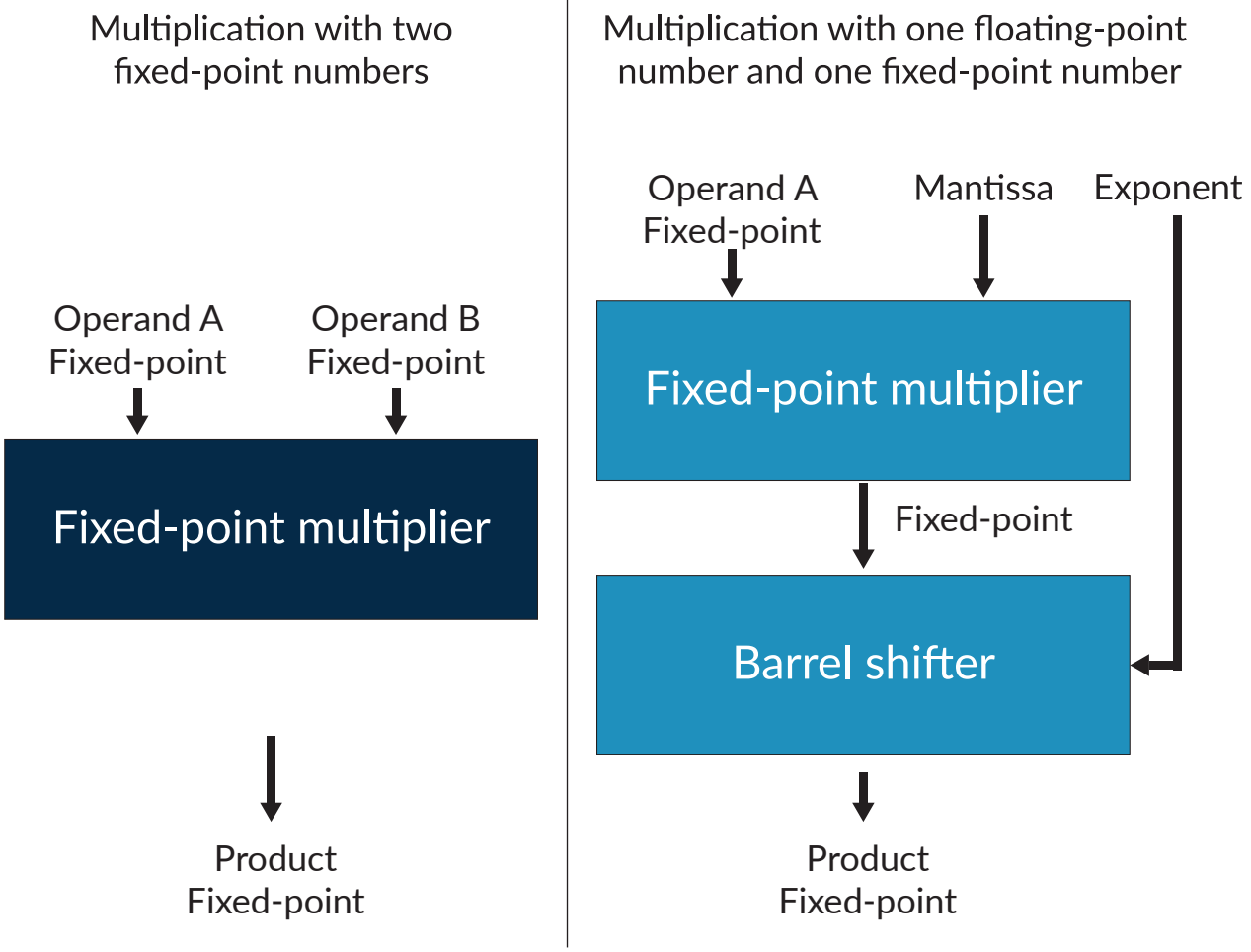


Figure 4. Two hardware implementations of multiplication

The multiplier multiplies the fixed-point number with the mantissa part of the floating-point number, and the shifter shifts the results according to the exponent value. The small size of the multiplier and the simplicity of the barrel shifter combine to make the multiplication function more efficient.

To show how this works, we implemented the multiplier with different operand configurations using a commercial 16-nm process technology and libraries. The results are highlighted in Figure 5.

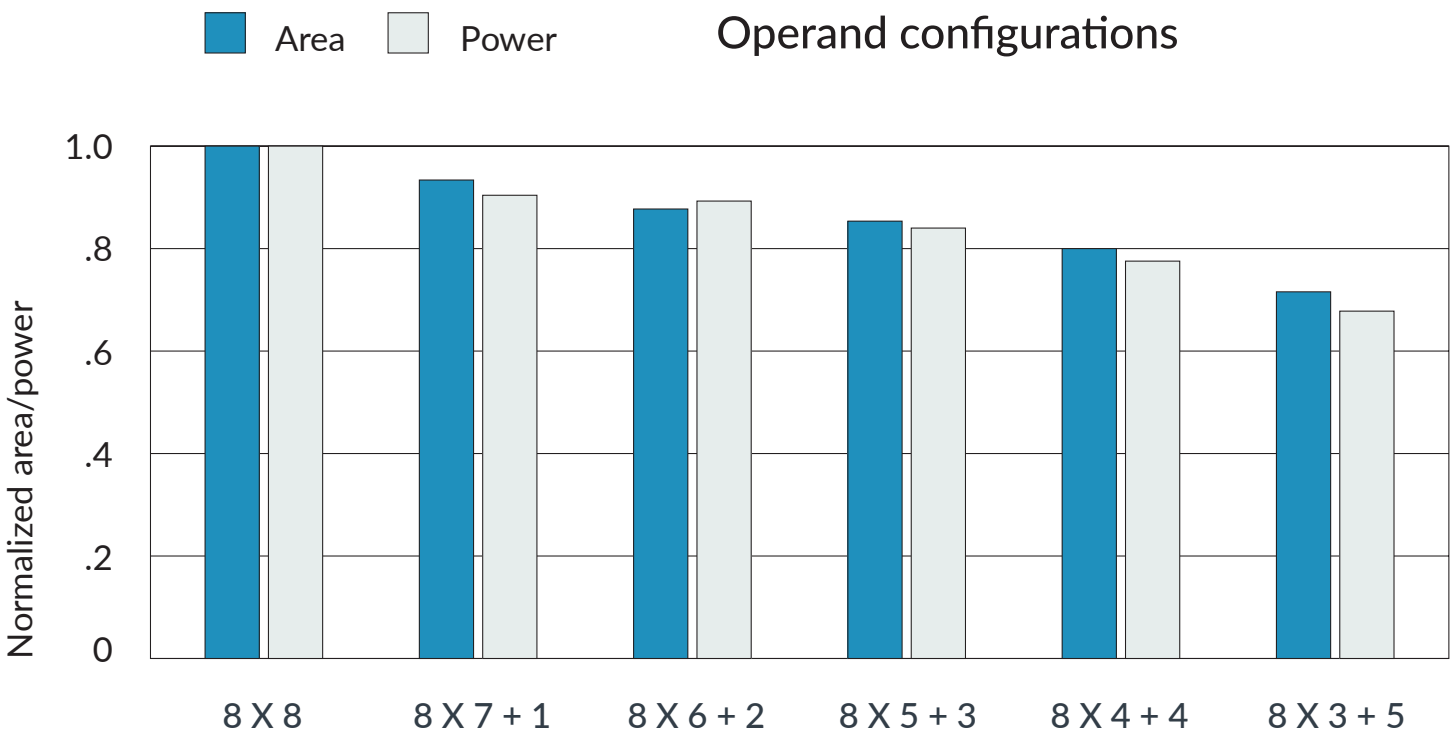


Figure 5. Hardware Tradeoff for Multiplier Configurations

The area and power are normalized with respect to the case with two 8-bit fixed-point operands. The floating-point operands are represented as mantissa bit width plus exponent bit width. The baseline is a multiplier with two 8-bit fixed-point operands (8x8). The area and power numbers are normalized with respect to the baseline. With the same bit width, the proposed scheme of combining fixed-point and floating-point operands can reduce both area and power. The reduction increases with fewer mantissa and more exponent bits, as a shifter is more efficient than a multiplier. For example, the floating-point operand with 4-bit mantissa and 4-bit exponent (8x4+4), can reduce the power and area by more than 20% compared to an 8x8 fixed-point multiplication.

Putting It All Together

We tested our recommended scheme, with floating-point representation for weights and fixed-point representation for activations, on the AlexNet CNN architecture, and got good results. We saw a 12.5% reduction in weight storage, and an 8% power reduction in multiplication.

But how would that translate to other CNN architectures? We wanted our number-representation scheme to work consistently in different architectures, so the hardware implementation could be both future-proof and viable for different CNN models. So we evaluated network accuracy with different bit-width settings, on

(m,e)	AlexNet	SqueezeNet	GoogLeNet	VCG-16
(7,0)	1.00	1.00	0.85	0.02
(10,0)	1.00	0.99	0.99	1.00
(3,4)	0.99	0.99	0.99	1.00

Table 1. Normalized Accuracy for CNNs

We denote the representation as (m, e), where m is the mantissa bit width and e is the exponent bit width (e=0 means fixed-point representation). The 7-bit fixed-point configuration (7, 0) used for AlexNet also works for SqueezeNet, but isn't good enough for GoogLeNet and VCC-16. The 10-bit fixed-point representation (10, 0) is required for acceptable accuracy across all the networks involved, and offers the consistency we were looking for.

Using our mix of floating-point and fixed-point representation, we only need a 7-bit floating-point (3,4) configuration. That means that, instead of using 11-bit weights (10-bit fixed-point numbers with and extra bit for the sign), we can use 8-bit weights (3-bit mantissa, 4-bit exponent, and 1 sign bit). This results in a 36% reduction in storage for weights and a 50% power reduction in multiplication.

Conclusion

Computational complexity and model size are two of the biggest challenges when it comes to moving deep-learning algorithms like CNNs from cloud-based, high-performance servers to end-user client devices. Using hardware accelerators has shown good performance at low power consumption when opting for fixed-point representation. This can significantly reduce storage requirements, memory bandwidth, and power consumption. In some cases, though, the elimination of floating-point representation has come at the cost of sacrificing accuracy.

Our approach, which uses floating-point representation for weights and fixed-point representation for activations, helps reduce memory and increase hardware efficiency, while maintaining an acceptable level of accuracy. What's more, while our work is based on mathematical quantization only, without the need for retraining the network, retraining can also be applied to reclaim part of whatever accuracy may be loss due to quantization.