Hardware Accelerators for Deep Neural Networks

In real-time applications, Machine Learning (ML) and Deep Neural Networks (DNNs) may run in embedded systems, that may have different requirements, limited energy power for example, without relying on internet connection and remote services.
In HPC applications ML and DNNs aim to offer Floating point operations per second (FLOPs) in the order of the TeraFLOPS or more (pre-exascale or exascale computing).
Thus we need both a high computing and/or more efficient representation of the information (that is typically composed by real numbers in ML, weights of DNN but also images).
Reducing power consumption and resource utilization is a strategic aspect.
The representation chosen for real numbers has a high impact on the synthesized hardware (cores, SoC accelerators, etc.).

master COPY

- Explore the state of art of information encoding in neural networks
- Deepen on the posit representation
- Explore the main trends in scalar and vector processors with application to neural networks
- Analyze a particular implementation of posits and deep neural networks and the trade-offs in using such format
- Study on a hardware implementation use-case of a posit processing unit (PPU) in a modern processor architecture pipeline

Real numbers and representations

The standard for real representation is **IEEE Float**.
The IEEE 754 floating point format is the standard for real number representation, fixed length format on 32-bit.
Three fields:

- 1-bit for sign
- 8-bit for exponent
- 23-bit for mantissa

The exponent is biased by a factor of 127
There are also some different sizes:

- FP16 (or half) with 5-bit for exponent and 10 for mantissa
- FP64 (or double) with 11-bits for exponent and 52-bits for mantissa
- FP128 (or quadruple) with 15-bits for exponent and 112-bits for mantissa

master COPY

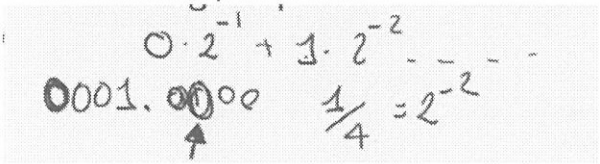
Another kind of standard is the fixed point standard.
The fixed-point format is quite common, although not a standard
Typically a fixed point on X bit, not defined a-priori, is composed as follow:

master COPY

- X/2 bits for the fractional part
- X/2 − 1 bits for the integer part
- 1 bit for the sign

For example we can represent the number 1.25 on a 8-bit fixed point as follow:

- The sign bit is 0 since the number is positive
- The integer part is 001 that is the integer representation of 1
- The fractional part is 0100



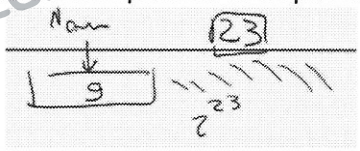
We though have some problems with ML and DNN, especially in embedded devices. IEEE Floats are starting to get older and some intrinsic problems are becoming evident:

• They are not portable across architectures, some manufacturers will implement some things, while others don't, and we may have problems with switching manufacturers, e.g. Intel, etc. This will cause different results.

• The hardware of the FPU, floating point unit, is very complex and resource costing.

We have the problem of handling normal and non-normal numbers, that may occupy more than one encoding.

E.g. NA encoding, we have an encoding of the first 9 bits which is fixed for NA, and the other 23 bits may vary, and these varying part will occupy 23 bits and so 2^23 encodings. We have a lot of encodings which are wasted that could be improved to improve the decimal accuracy.



• The IEEE 754 standard is never fully implemented in any architecture

They do quite well in an architecture but do not act too well in all of them.

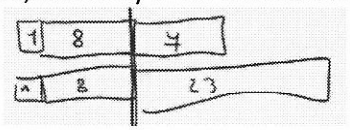
• Exception handling is troublesome (overflow? underflow? etc..)

• They are too general and not optimal for specific applications (e.g. machine learning)

One industry alternative is **BFLOAT16.**

Also known as Brain Float. Introduced by Google to be used in its TPU engine 1.

We have 1 bit for sign, 8 for exponent, but only 7 for mantissa.



The conversion between the two is simple, we just need to shift the number of 16 places, replicated, or in the opposite (below to above) we can truncate the least significant bit.

• Neural network focused

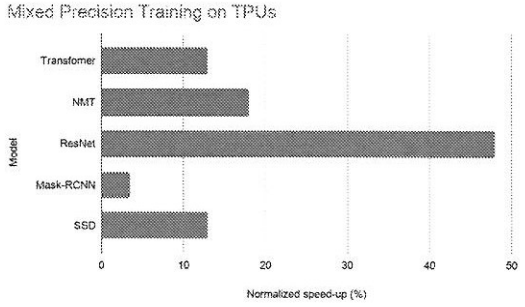
• Similar to IEEE FP16, main competitor, but has fewer number of bits in the exponent and its dynamic range is also smaller, cover a lower range of number on real axis, so FP16 seems to be practical for ML applications where the gradients tend to dash to the left or right.

• Maintain same number of exponent bits as FP32: conversion is easier (just a truncation of mantissa) and not troublesome

• Compared to FP16 it has higher dynamic range but less decimal accuracy

• Allows for mixed precision training: network training phase is optimized by the tensor compiler, using the low-precision type for the weights and activations when possible.

The speedup of the training phase in % is shown, considering the time to train the NN wrt 32 bit FLOAT:



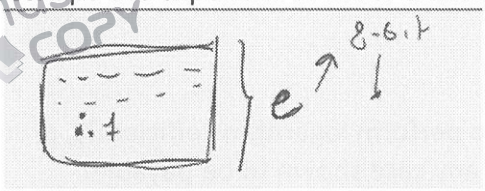
Flexpoint is an alternative.

- Proposed by Intel
- Tries to mix fixed point and floating point
- A shared exponent is broadcasted to entire layers or tensor in a DNN.

The numbers inside a tensor are represented like a fixed point with an integer part and a fraction part while they share a common exponent.

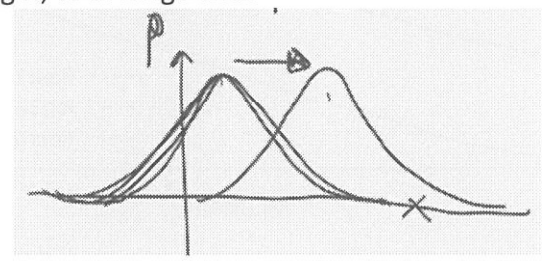
We will have a tensor with all numbers like int part.frac part and they will share a common exponent, of e.g. 8 bits or more.

Not representing an exponent for each we save memory, and it can be also adjusted during training if the model is converging to weights which is outside the range of the exponent, e.g. waste shifting to the right, we reshape the exponent to cover a different dynamic range.



This reduces the computational effort since fixed point arithmetic is used internally.

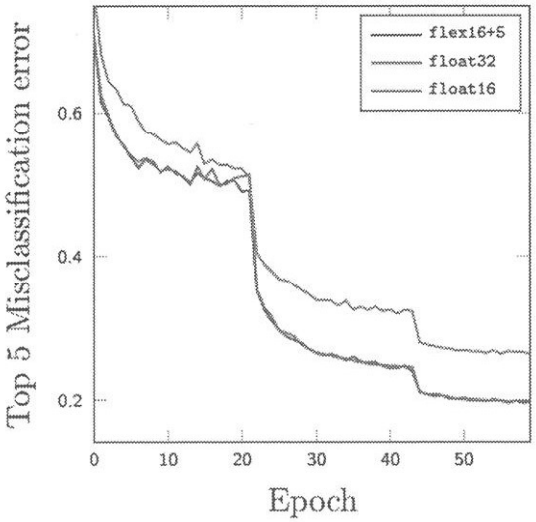
We may start the training with a distribution of weights in a certain manner, the distribution may be detected to be skewed to the right, for example, increasing them in magnitudes. We can re-adapt the value by increasing it, to arrange this.



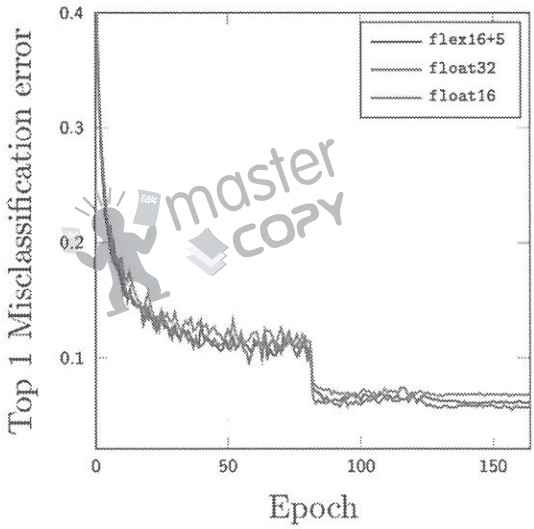
The report shows the comparison of two networks, flex16+5 is 16 fixed point with integer and fraction part and 5 as an exponent.

The performance is better than float16 while being comparable with float32.

In CIFAR-10 the difference is not so big.



(a) ImageNet1k AlexNet



(b) CIFAR-10 ResNet

We use though less bits for a comparable accuracy.

The alternatives: **Logarithmic numbers**, **LNS**
Based on the logarithmic number system
Key idea: multiplication of two numbers is the sum of the logarithms of the two numbers.
Multiplications and division can be a burden for the hw, the area of multiplication and division is typically very large, we can image to not have this area then.
Using logarithmic properties we can only use addition and subtraction, since if we represent every number like this, we just need to sum the exponents, and analogous for division:

$$y_1 = 2^{x_1}$$
$$y_2 = 2^{x_2}$$
$$y_1 y_2 = 2^{x_1 + x_2}$$
$$y_1 / y_2 = 2^{x_1 - x_2}$$

master COPY
master COPY

Logarithm table are employed to perform logarithms efficiently, precomputed for example.
The trade-of of the storage is in any case shown to be payable.

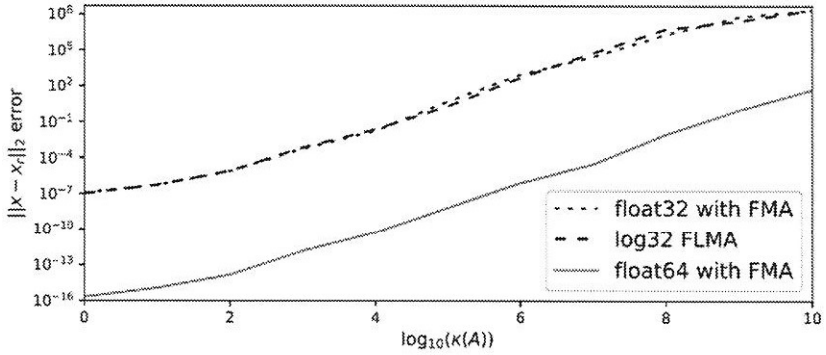


Fig. 5. Least squares $Ax = b$ solution average error via Householder QR, 64×64 matrices with condition number $\kappa(A)$ from 1 to 10^{10}.

An alternative is **posit numbers**, introduced and created by John L. Gustafson.
Try to solve most of the problems seen before with IEEE floats.
Challenges:
- Prove themselves worth to replace floats in the future, at least in ML and DNN
- Reach proper hardware support firstly in application specific hardware
- Then, possibly, in general purpose processors

master COPY

A posit number has two configuration parameters: nbits, esbits.
The posit format is a fixed-length format, with up to four fields
- Sign field: 1-bit
- Regime field: variable length, run-length encoding.
- Exponent field: at most es bits
- Fraction field: variable length mantissa

The variable length will depend on the length of the regime field, encoded with a run-length approach, the value of the regime depend on its length and viceversa.
ES, the upper limit to the exp field length, can also be 0 is a particular posit format, totally absent.
Even if we have es != 0 we may have that the regime field will occupy all the space and the exponent will collapse.
The fraction field occupies the remaining part.

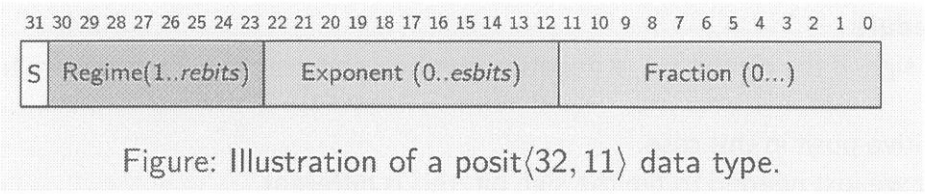


Figure: Illustration of a posit⟨32, 11⟩ data type.

E.g. 11 bits for the exponent, 32 overall bits

Given a Posit on nbits; esbits, represented by the integer X and let e and f be respectively the exponent and fraction values and k the value encoded by the regime, the real number r represented by that encoding is:

$$r = \begin{cases} 0, \text{ if } X = 0 \\ \text{NaN}, \text{ if } X = -2^{(nbits-1)} \\ sign(X) \times useed^k \cdot 2^e \cdot (1+f), \text{ otherwise} \end{cases}$$

This part is equivalent to IEEE Float:

$2^e \cdot (1+f)$,

But the difference is on useed^k, fixed value corresponding to 2^2^esbits, fixed and configured when we set esbits.

e regime, the real number

$useed^k = \left(2^{2^{esbits}}\right)^k$

K is the super-exponent.

If we want to decode the number in a real number we have to perform this operation.

The regime field is a variable length field, spanning from 1 to nbits − 1 bits.

It is the first to be computed to determine the others.

It is encoded and decoded following a run-length approach.

Its value l representing the regime length is the number of subsequent identical bits b ended by the opposite bit.



For example, given the bitstring 11110 the value l will be 4. If the strike is composed by 0s b = 0 will be negative, otherwise b = 1.

We discard the sign, count the bits which are identical, and as soon we see a different bit we stop.

Then we look at the stopping bit.

Given the run-length l the actual k value (see formula in previous slide) is

$$k = \begin{cases} -l, \text{ if } b = 0 \\ l - 1, \text{ otherwise} \end{cases}$$

It will depend on the stopping bit b.

Even if we have esbits=3 we may not have the exponent, as we said, if the regime is so. large.