



The machine epsilon  $\epsilon_{\text{machine}}$  measures the numerical error caused by the floating point format. For single-precision floats,  $\epsilon_{\text{machine}} \approx 1.2 \times 10^{-7}$ .

There are a few special cases here as well.

- If the real-number result of the operation is larger in magnitude than the largest non-infinite representable floating point number (about  $\pm 3.4 \times 10^{38}$  for single-precision floats), then the operation returns positive or negative infinity, as appropriate. This is called *overflow*.
- If the real-number result of the operation is smaller in magnitude than  $1/2$  the smallest non-zero representable floating point number (about  $1.4 \times 10^{-45}$  for single-precision floats, assuming subnormal numbers are not flushed) then the result of the operation is zero. This is called *underflow*.
- If the operation does not make sense, such as division by zero or something like  $(+\infty) - (+\infty)$ , then the result is NaN.

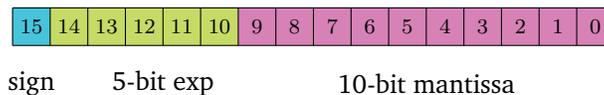
### What is the cost of 32-bit floating-point computation?

- Need to store 32 bits in memory for each number used in the algorithm.
- Need to have specialized hardware to support the 32-bit FP computation (this is almost always the case, but may not hold for some embedded devices).

How can we reduce this cost? **Reduce the number of bits!**

Motivation: *machine learning computations are already noisy* due both to random sampling of examples in SGD and measurement imprecision when gathering data from the real world. As long as the numerical error from a reduced-bit-count representation is small compared with the noise already observable in SGD, we can expect that the performance won't be impacted much.

**Half-precision floating point numbers.** 16-bit “half-precision” floating-point numbers have recently become popular for machine learning tasks, particularly for deep learning. These numbers are represented much like single-precision floats, except with fewer bits.



In comparison to single-precision floating point numbers, half-precision floats have:

- a larger machine epsilon (meaning more numerical errors due to rounding),  $\epsilon_{\text{machine}} \approx 9.8 \times 10^{-4}$
- a smaller overflow threshold (meaning more overflow could happen) of about  $6.5 \times 10^4$
- a larger underflow threshold of about  $6.0 \times 10^{-8}$ .

As long as the numbers we compute in the course of a learning algorithm stay far away from these thresholds, half-precision floating point numbers can do a pretty good job of approximating real numbers.

### What benefits can we expect to get from computing in lower precision?

**Pros of low-precision computing.**

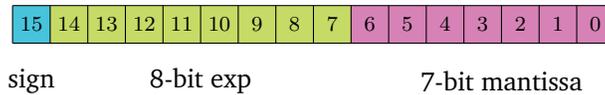
- Can fit more numbers (and therefore more training examples, activations, etc.) in memory
- Can store more numbers (and therefore larger models) in the cache
- Can transmit more numbers per second
- Can compute faster by extracting more parallelism in a fixed-width SIMD register
- Uses less energy

**Cons of low-precision computing.**

- Limits the range of numbers we can represent (the range between the overflow and underflow thresholds)
- Need specialized support from the hardware
- Introduces **quantization error** when we store a full-precision number in a low-precision representation

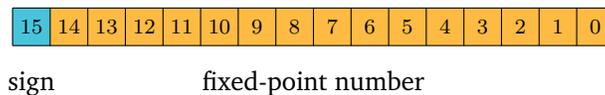
Let's try to address these cons.

**One way to address limited range: use more exponent bits.** Nothing (apart from the IEEE standard) forces us to size the exponent and mantissa of our 16-bit floating-point numbers as a 5-bit exponent and 10-bit mantissa. One popular alternate format (used in many ML processors including the TPU) is the **bfloat16** format. This splits the exponent and mantissa into 8/7 as follows



What can we say about the range of bfloat16 numbers as compared with IEEE half-precision floats and single-precision floats? How does their machine epsilon compare?

**One way to address the need for hardware support: use fixed-point arithmetic instead.** Fixed-point arithmetic represents a number as an integer times a *fixed* exponent. That is, we have something like this:



and the number that it represents is

$$\text{represented number} = 2^{\text{fixed exponent}-127} \cdot (\text{number as signed integer}) .$$

Most operations with fixed-point numbers can be done with integer arithmetic, which is supported on most processors already. It's also much cheaper to compute in terms of hardware costs. People have even used 8-bit fixed-point numbers for machine learning tasks!

**What are the downsides of using fixed-point numbers for ML? Can you think of a place where you've already used something like fixed-point numbers in a programming assignment?**

**One way to address quantization error: use stochastic rounding.** The standard mode of floating-point rounding is to round to the nearest representable number. This is the ordinary form of rounding that you probably learned in school.

However, for a lot of ML applications, especially where we are computing *averages*, this is not necessarily the best approach. The problem with round-to-nearest is that it is *biased* in a statistical sense. After rounding, the number no longer in expectation the same as the number before rounding.

To address this, people often use *stochastic rounding*, also known as *randomized rounding*. Randomized rounding rounds a number either up or down at random in such a way that the expected value of the output after rounding is equal to the input.

- This can be great for computing sums and averages where the law of large numbers will eventually kick in.
- Example: Hoeffding's inequality.