Lecture 23: Low-Precision Machine Learning

In this lecture, we’ll talk about the relatively recent trend of using number formats with a small number of bits to reduce the computational cost of machine learning training. But first, we need to understand the baseline: what do traditional machine learning applications do? Usually, we reason about machine learning algorithms as if they were computing with infinite-precision real numbers, but of course this isn’t actually the case. Traditional machine learning systems use 32-bit “single-precision” floating point numbers (or occasionally 64-bit “double-precision” floating point numbers). How are these numbers represented?

![Floating point representation](http://example.com/floating-point-diagram)

represented number = (-1)\(\text{sign} \cdot 2^{\text{exponent} - 127} \cdot 1.b_{22}b_{21}b_{20} \ldots b_0\)

Note the implicit leading 1-bit before the mantissa. This is how floating point numbers are represented, except for three special cases. The first special case occurs for the smallest representable numbers, when the exponent = 0. In this case,

represented number = (-1)\(\text{sign} \cdot 2^{-126} \cdot 0.b_{22}b_{21}b_{20} \ldots b_0\).

When the mantissa is also zero, this represents 0 (note the possibility of negative zero). When the mantissa is non-zero, these are so-called denormal numbers. Denormal numbers can cause performance issues in some code, since they require separate computational pathways. As a result, denormal numbers are often flushed to zero—we just round them to zero when they occur. Typically this has minimal numerical impact on machine learning code.

The second special case occurs for the largest exponent value, when exponent = 255 = 2\(8\) − 1. Here, there are two possibilities. If the mantissa is zero, then the floating point value represents either positive infinity (+\(\infty\)) or negative infinity (−\(\infty\)), depending on the sign bit. If the mantissa is nonzero, then the floating point value represents something that is not a number, called a NaN value. This usually indicates some sort of error. Here, the bits of the mantissa can contain a message that indicates how the error occurred.

**Floating-point operations** are usually equivalent to doing the following.

- Interpret the ordinary floating-point number inputs as real numbers.
- Perform the desired operation on those real numbers, producing a real-number result.
- Find the floating-point number that is closest to that result, and return that floating-point number.

In practice this is done with a circuit that is equivalent to this procedure, since obviously we can’t actually compute with real numbers. This last rounding step has bounded relative error, such that for some constant \(\epsilon\) (called the machine epsilon)

\[|\text{FloatingPointOp}(x, y) - \text{RealNumberOp}(x, y)| \leq |\text{RealNumberOp}(x, y)| \cdot \epsilon\]
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.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>5-bit exp</td>
<td>10-bit mantissa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{cccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\text{sign} & \text{8-bit exp} & \text{7-bit mantissa}
\end{array}
\]

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:

\[
\begin{array}{cccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\text{sign} & \text{fixed-point number}
\end{array}
\]

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.