Low-Precision Arithmetic
The standard approach

**Single-precision floating point (FP32)**

- 32-bit floating point numbers

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sign | 8-bit exponent | 23-bit mantissa |

- Usually, the represented value is

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

![Floating-point number representation](image)

- **sign** 8-bit exponent 23-bit mantissa

• When the exponent is all 0s, and the mantissa is all 0s
  • This represents the real number 0
  • Note the possibility of “negative 0”

• When the exponent is all 0s, and the mantissa is nonzero
  • Called a “denormal number” — degrades precision gracefully as 0 approached

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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<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>8-bit exponent</td>
<td>23-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>
<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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• When the exponent is all 1s (255), and the mantissa is all 0s
  • This represents infinity or negative infinity, depending on the sign
  • Indicates overflow or division by 0 occurred at some point
    • Note that these events usually do not cause an exception, but sometimes do!

• When the exponent is all 1s (255), but the mantissa is nonzero
  • Represents something that is not a number, called a NaN value.
  • This usually indicates some sort of compounded error.
  • The bits of the mantissa can be a message that indicates how the error occurred.
DEMO
Measuring Error in Floating Point Arithmetic

If $x \in \mathbb{R}$ is a real number within the range of a floating point representation, and $\tilde{x}$ is the closest representable floating-point number to it, then

$$|\tilde{x} - x| = |\text{round}(x) - x| \leq |x| \cdot \varepsilon_{\text{machine}}.$$ 

Here, $\varepsilon_{\text{machine}}$ is called the machine epsilon and bounds the relative error of the format.
Error of Floating-Point Computations

If \( x \) and \( y \) are real-numbers representable in a floating-point format, \( \circ \) denotes an (infinite-precision) binary operation (such as +, \( \cdot \), etc.) and \( \bullet \) denotes the floating-point version of that operation, then

\[
x \bullet y = \text{round}(x \circ y) \quad \text{and} \quad |(x \bullet y) - (x \circ y)| \leq |x \circ y| \cdot \varepsilon_{\text{machine}},
\]

as long as the result is in range.
Exceptions to this error model

• If the exact result is larger than the largest representable value
  • The floating-point result is **infinity** (or minus infinity)
  • This is called **overflow**

• If the exact result falls in the range of denormal numbers, there may be more error than the model predicts

• If there is an invalid computation
  • e.g. the square root of a negative number, or infinity + (-infinity)
  • the result is **NaN**
How can we use this info to make our ML systems more scalable?
Low-precision compute

• Idea: replace the 32-bit or 64-bit floating point numbers traditionally used for ML with **smaller numbers**
  • For example, 16-bit floats or 8-bit integers

• New specialized **chips for accelerating ML training**.
  • Many of these chips leverage **low-precision compute**.

Google’s TPU  NVIDIA’s GPUs  Intel’s NNP
A low-precision alternative
FP16/Half-precision floating point

- 16-bit floating point numbers

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

sign 5-bit exp 10-bit mantissa

- Usually, the represented value is

\[ x = (-1)^{\text{sign bit}} \cdot 2^{\text{exponent}-15} \cdot 1.\text{significand}_2 \]
Benefits of Low-Precision: Compute

- Use **SIMD/vector instructions** to run more computations at once

**SIMD Precision**

- 64-bit float vector
  - F64 F64 F64 F64
- 32-bit float vector
  - F32 F32 F32 F32 F32 F32 F32
- 16-bit int vector
  - [16-bit int vector]
- 8-bit int vector
  - [8-bit int vector]

**SIMD Parallelism**

- 4 multiplies/cycle
  - (vmulpd instruction)
- 8 multiplies/cycle
  - (vmulps instruction)
- 16 multiplies/cycle
  - (vpmaddwd instruction)
- 32 multiplies/cycle
  - (vpmaddubsw instruction)
Benefits of Low Precision: Memory

- Puts **less pressure** on memory and caches

**Precision in DRAM**

- 64-bit float vector
- 32-bit float vector
- 16-bit int vector
- 8-bit int vector

**Memory Throughput**

- 5 numbers/\(\text{ns}\)
- 10 numbers/\(\text{ns}\)
- 20 numbers/\(\text{ns}\)
- 40 numbers/\(\text{ns}\)

(assuming ~40 GB/sec memory bandwidth)
Benefits of Low Precision: Communication

- Uses **less network bandwidth** in distributed applications

---

**Precision in DRAM**

- 32-bit float vector
- 16-bit int vector
- 8-bit int vector
- Specialized lossy compression

**Memory Throughput**

- 10 numbers /ns
- 20 numbers /ns
- 40 numbers /ns
- >40 numbers /ns

(assuming ~40 GB/sec network bandwidth)
Benefits of Low Precision: Power

• Low-precision computation can even have a super-linear effect on energy

• Memory energy can also have quadratic dependence on precision
Effects of Low-Precision Computation

• **Pros**
  • Fit more numbers (and therefore more training examples) in memory
  • Store more numbers (and therefore larger models) in the cache
  • Transmit more numbers per second
  • Compute faster by extracting more parallelism
  • Use less energy

• **Cons**
  • Limits the numbers we can represent
  • Introduces **quantization error** when we store a full-precision number in a low-precision representation
Numeric properties of 16-bit floats

- A larger machine epsilon (larger rounding errors) of $\epsilon_{\text{machine}} \approx 9.8 \times 10^{-4}$
  - Compare 32-bit floats which had $\epsilon_{\text{machine}} \approx 1.2 \times 10^{-7}$.

- A smaller overflow threshold (easier to overflow) at about $6.5 \times 10^4$
  - Compare 32-bit floats where it’s $3.4 \times 10^{38}$

- A larger underflow threshold (easier to underflow) at about $6.0 \times 10^{-8}$.
  - Compare 32-bit floats where it’s $1.4 \times 10^{-45}$

With all these drawbacks, does anyone use this?
Half-precision floating point support

• Supported on most modern machine-learning-targeted GPUs
  • Including efficient implementation on NVIDIA Pascal GPUs

<table>
<thead>
<tr>
<th>GPU</th>
<th>DFMA (FP64 TFLOP/s)</th>
<th>FFMA (FP32 TFLOP/s)</th>
<th>HFMA2 (FP16 TFLOP/s)</th>
<th>DP4A (INT8 TIOP/s)</th>
<th>DP2A (INT16/8 TIOP/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP100 (Tesla P100 NVLink)</td>
<td>5.3</td>
<td>10.6</td>
<td>21.2</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>GP102 (Tesla P40)</td>
<td>0.37</td>
<td>11.8</td>
<td>0.19</td>
<td>43.9</td>
<td>23.5</td>
</tr>
<tr>
<td>GP104 (Tesla P4)</td>
<td>0.17</td>
<td>8.9</td>
<td>0.09</td>
<td>21.8</td>
<td>10.9</td>
</tr>
</tbody>
</table>

Table 1: Pascal-based Tesla GPU peak arithmetic throughput for half-, single-, and double-precision fused multiply-add instructions, and for 8- and 16-bit vector dot product instructions. (Boost clock rates are used in calculating peak throughputs. TFLOP/s: Tera Floating-point Operations per Second. TIOP/s: Tera Integer Operations per Second. https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/)

• Good empirical results for deep learning

Framework Support

• In TensorFlow, can convert precision of types explicitly
  • `tensorflow.cast(x, 'float16')`
  • Can also set the data type for neural networks

• PyTorch also has similar half-precision support

```python
half(memory_format=torch.preserve_format) → Tensor
```

`self.half()` is equivalent to `self.to(torch.float16)`. See `to()`.
One way to address limited range: more exponent bits
Bfloat16 — “brain floating point”

• Another 16-bit floating point number

Q: 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?
Bfloat16 (continued)

• Main benefit: numeric range is now the same as single-precision float
  • Since it looks like a truncated 32-bit float
  • This is useful because ML applications are more tolerant to quantization error than they are to overflow

• Also supported on specialized hardware
An alternative to low-precision floating point

**Fixed point numbers**

- \( p + q + 1 \) –bit fixed point number

The represented number is

\[
\begin{align*}
x &= (-1)^{\text{sign bit}} \left( \text{integer part} + 2^{-q} \cdot \text{fractional part} \right) \\
&= 2^{-q} \cdot \text{whole thing as signed integer}
\end{align*}
\]
Arithmetic on fixed point numbers

- **Simple and efficient**
  - Can just use preexisting integer processing units
  - **Lower power** than floating point operations with the same number of bits

- **Mostly exact**
  - Underflow impossible
  - Overflow can happen, but is easy to understand
  - Can always convert to a higher-precision representation to avoid overflow

- Can represent a **much narrower range of numbers than float**
Support for fixed-point arithmetic

- **Anywhere integer arithmetic is supported**
  - CPUs, GPUs
  - Although not all GPUs support 8-bit integer arithmetic
  - And AVX2 does not have all the 8-bit arithmetic instructions we’d like

- Particularly effective on **FPGAs and ASICs**
  - Where floating point units are costly

- Sadly, very **little support for other precisions**
  - 4-bit operations would be particularly useful
Breakout Questions

• **Q:** What are the upsides/downsides of using fixed-point numbers for ML?
  • Compared to floating-point?

• **Q:** Can you think of a place where you’ve already used something like fixed-point numbers in a programming assignment?
A powerful hybrid approach

Block Floating Point

- Motivation: when storing a vector of numbers, often these numbers all lie in the same range.
  - So they will have the same or similar exponent, if stored as floating point.

- **Block floating point** shares a single exponent among multiple numbers.
A more specialized approach

Custom Quantization Points

• Even more generally, we can just have a list of $2^b$ numbers and say that these are the numbers a particular low-precision string represents
  • We can think of the bit string as indexing a number in a dictionary

• Gives us total freedom as to range and scaling
  • But computation can be tricky

• Some recent research into using this with hardware support
  • “The ZipML Framework for Training Models with End-to-End Low Precision: The Cans, the Cannots, and a Little Bit of Deep Learning” (Zhang et al 2017)
How is precision used for DNN training

• Signals flow through network in backpropagation
  • Generally, we assign a precision to each of the types of signals, and different types of signals can have different precisions

Types of signals in backpropagation:
• Training dataset
• Vectors that store weights/parameters
• Gradients
• Communication among parallel workers
• Activations
• Backward pass signals
• Weight accumulators
• Momentum/ADAM vectors
Mixed-Precision training

• Use **low precision for some numbers** and higher precision for others

1. Accumulate weights in single-precision.
2. Scale the loss to prevent underflow.
3. Use fused mixed-precision multiply-adds

\[ \text{FP16} \times \text{FP16} \rightarrow \text{ACC} \rightarrow \text{32-to-16} \]
Low-precision formats in general

• These are some of the most common formats used in ML
  • …but we’re not limited to using only these formats!

• There are many other things we could try
  • For example, floating point numbers with different exponent/mantissa sizes
  • Block floating point numbers with different block sizes/layouts
  • Fixed point numbers with nonstandard widths

• Problem: there’s no hardware support for these other things yet, so it’s hard to get a sense of how they would perform.
Theoretical Guarantees for Low Precision

• Reducing precision adds noise in the form of round-off error.

• Two approaches to rounding:
  • biased rounding – round to nearest number
  • unbiased rounding – round randomly: \( E[Q(x)] = x \)

Using this, we can prove guarantees that SGD works with a low precision model…since a low-precision gradient is an unbiased estimator.
Why unbiased rounding?

- Imagine running SGD with a low-precision model with update rule

\[ w_{t+1} = \tilde{Q} \left( w_t - \alpha_t \nabla f (w_t; x_t, y_t) \right) \]

- Here, \( Q \) is an unbiased quantization function

- In expectation, this is just gradient descent

\[
E[w_{t+1}|w_t] = E \left[ \tilde{Q} \left( w_t - \alpha_t \nabla f (w_t; x_t, y_t) \right)|w_t \right] \\
= E \left[ w_t - \alpha_t \nabla f (w_t; x_t, y_t)|w_t \right] \\
= w_t - \alpha_t \nabla f (w_t)
\]
Implementing unbiased rounding

- To implement an unbiased to-integer quantizer:
  sample $u \sim \text{Unif}[0, 1]$, then set $Q(x) = \lfloor x + u \rfloor$

- Why is this unbiased?

$$
E[Q(x)] = \lfloor x \rfloor \cdot P(Q(x) = \lfloor x \rfloor) + (\lfloor x \rfloor + 1) \cdot P(Q(x) = \lfloor x \rfloor + 1) \\
= \lfloor x \rfloor + P(Q(x) = \lfloor x \rfloor + 1) = \lfloor x \rfloor + P(\lfloor x + u \rfloor = \lfloor x \rfloor + 1) \\
= \lfloor x \rfloor + P(x + u \geq \lfloor x \rfloor + 1) = \lfloor x \rfloor + P(u \geq \lfloor x \rfloor + 1 - x) \\
= \lfloor x \rfloor + 1 + (\lfloor x \rfloor + 1 - x) = x.
$$
DEMO
Doing unbiased rounding efficiently

• We still need an efficient way to do unbiased rounding

• **Pseudorandom number generation can be expensive**
  • E.G. doing C++ rand or using Mersenne twister takes many clock cycles

• Empirically, we can use **very cheap** pseudorandom number generators
  • And still get good statistical results
  • For example, we can use XORSHIFT which is just a cyclic permutation
Benefits of Low-Precision Computation

Conclusion and Drawbacks of low-precision

• The drawback of low-precision arithmetic is the low precision!

• Low-precision computation means we accumulate more rounding error in our computations

• These rounding errors can add up throughout the learning process, resulting in less accurate learned systems

• The trade-off of low-precision: throughput/memory vs. accuracy