

Numerics and AI

Paulius Micikevicius

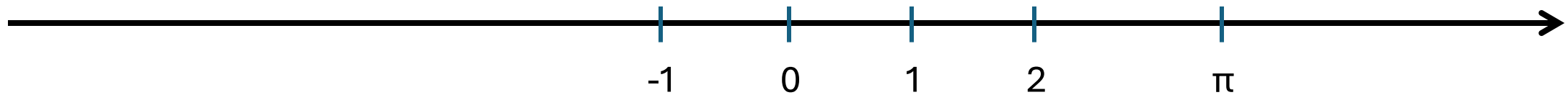
GPU Mode: November 15, 2025

Outline

- **Basics of number representation**
- **Brief history of AI numerics**
- **Some FP Quirks**
- **Note:**
 - a lot of material is borrowed from public NVIDIA presentations where most of my numerics work was done

Real Number Line

- **Infinite number of values**
 - Overall: arbitrarily large and small magnitudes -> **infinite range**
 - Also between any pair of values -> **infinite precision**



Computer Numerics

- **Cannot do infinite** 😊
 - Finite memory
 - Finite silicon area (or time to complete an operation)
- **Must adopt some number representation that samples the real number line**
 - By definition there will be values that cannot be represented accurately

Note on Terminology: Precision and Accuracy

- **Precision:**

- Refers to sample spacing (finer sampling -> *higher* precision)
- Note: something can be precise but inaccurate

- **Accuracy:**

- Refers to how close a sample is to the actual value

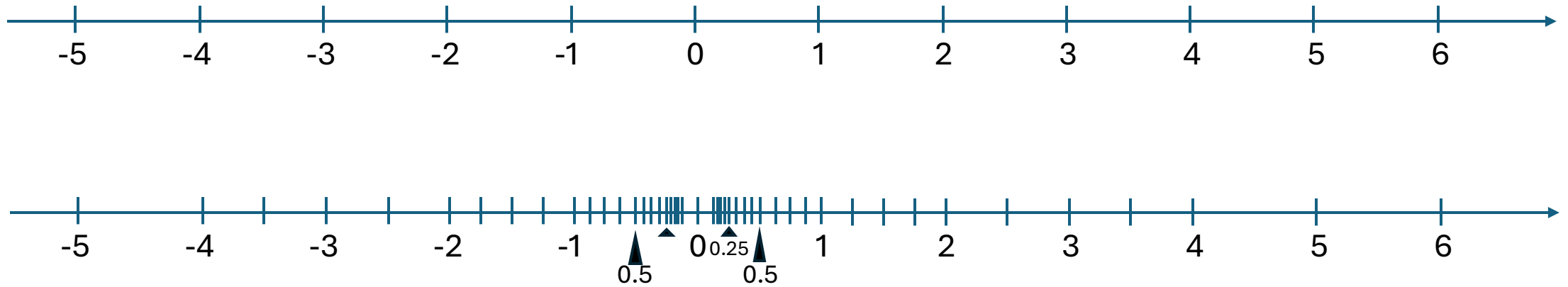
- **Example: pi**

- 3.141543 is more precise and more accurate than 3.14
- 3.143738 is more precise but less accurate than 3.14

Two Widespread Format Categories

- **We'll consider just binary representations**
 - There are things like binary-coded decimals, etc. -> out of scope for us
 - The question is how to decode N -bit patterns (so, distribute $\sim 2^N$ samples)
- **Integer variants**
 - Represent integers
 - Sample the real number line at equal intervals (1 unit increments)
- **Floating point**
 - Represent integers and fractions
 - Sample the real number line at varying intervals
 - Uniformly for each power of 2 interval

Integer vs Floating Point Sampling



Integer Representations

- **Unsigned:**

- N unique samples (given N bits): $0, 1, \dots, (2^N - 1)$
- In C “overflows” wrap around
 - 8-bit: $255 + 1 \rightarrow 0$
 - Because of this sometimes code can be slower than signed integers
 - Example: LayerNorm kernel with 128 elements per thread: 143 vs 168 registers

- **Sign-magnitude:**

- One bit for sign, remaining $(N - 1)$ bits for magnitude
- Symmetric sampling around 0 (same number of positives and negatives)
- Implies positive and negative zeros
 - Can be seen as a waste of a sample, but can be useful for “Lamport” style signaling

- **Twos-complement:**

- Only one zero representations
- *Assymmetric* around 0: one more negative sample ($-128, \dots, 0, \dots, 127$ for 8-bit)

Integer Adoption for AI

- **Mostly for inference**
- **Some training**
 - TPUs: [Accurate Quantized Training \(AQT\) for TPU v5e | Google Cloud Blog](#)
- **Pros:**
 - In the early days was the only way to go below 32 or 16 bits
 - Saves memory footprint and bw, higher math rate
- **Cons: maintaining accuracy isn't easy**
 - Networks are trained in floating point (i.e. have fractions)
 - Add *scaling* factors to get integers to represent fractions
 - Requires calibration, QAT, etc.

Linear Operations Can Cheaply Deal with Scales

- $x = (x_i | 1 \leq i \leq k)$ be a vector of reals.
- x^q be the integer-quantized version of x .
- $x' = (x_i^q * dscale_x | 1 \leq i \leq k)$ be the approximation of x after dequantization.
- Define y and y' similarly.

Then,

$$z' = \sum_i x'_i * y'_i = dscale_x * dscale_y \sum_i x_i^q * y_i^q$$

Illustration

- From: <https://arxiv.org/abs/2004.09602>

- Figure 2:

- values that will map to max int with different calibrations

- Table 5:

- Best calibration varies by network
- Compare BERT-L and EfficientNet b0
 - 99.99+ percentiles:
 - Best for BERT-L
 - Terrible for EfficientNet b0
 - Entropy and 99.9 percentile:
 - Catastrophic for BERT-L
 - Best for EfficientNet b0

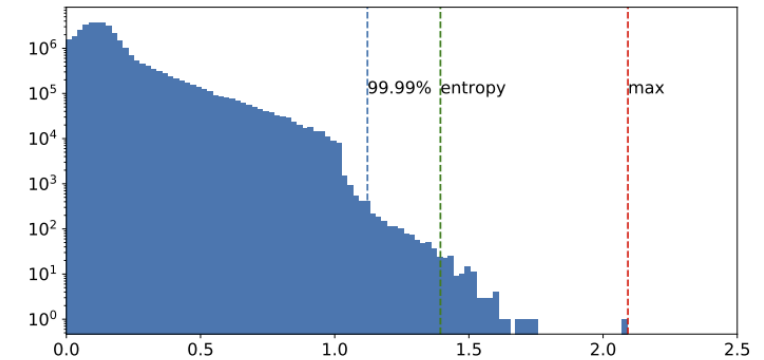


Figure 2: Histogram of input activations to layer 3 in ResNet50 and calibrated ranges

Models	fp32	Max	Entropy	99.9%	99.99%	99.999%	99.9999%
MobileNet v1	71.88	69.51	70.19	70.39	70.29	69.97	69.57
MobileNet v2	71.88	69.41	70.28	70.68	71.14	70.72	70.23
ResNet50 v1.5	76.16	75.82	76.05	75.68	75.98	75.97	76.00
ResNet152 v1.5	78.32	77.93	78.21	77.62	78.17	78.17	78.19
Inception v3	77.34	72.53	77.54	76.21	77.52	77.43	77.37
Inception v4	79.71	0.12	79.60	78.16	79.63	79.12	71.19
ResNeXt50	77.61	77.31	77.46	77.04	77.39	77.45	77.39
ResNeXt101	79.30	78.74	79.09	78.77	79.15	79.17	79.05
EfficientNet b0	76.85	22.3	72.06	70.87	68.33	51.88	42.49
EfficientNet b3	81.61	54.27	76.96	77.80	80.28	80.06	77.13
Faster R-CNN	36.95	36.38	36.82	35.22	36.69	36.76	36.78
Mask R-CNN	37.89	37.51	37.75	36.17	37.55	37.72	37.80
Retinanet	39.30	38.90	38.97	35.34	38.55	39.19	39.19
FCN	63.70	63.40	64.00	62.20	64.00	63.90	63.60
DeepLabV3	67.40	67.20	67.40	66.40	67.40	67.50	67.40
GNMT	24.27	24.31	24.53	24.34	24.36	24.38	24.33
Transformer	28.27	21.23	21.88	24.49	27.71	20.22	20.44
Jasper	96.09	95.99	96.11	95.77	96.09	96.09	96.03
BERT Large	91.01	85.92	37.40	26.18	89.59	90.20	90.10

Table 5: Post training quantization accuracy. Weights use per-channel or per-column max calibration. Activations use the calibration listed. Best quantized accuracy per network is in bold.

Floating Point

IEEE Floating Point (FP) Encoding

- **Bit fields (ExMy):**

- Sign: 1 bit (0: positive, 1: negative)
- Exponent: E bits
 - Exponent for a power of 2
 - Biased integer: bias = $2^{E-1} - 1$
- Mantissa: M bits
 - Fractional bits correspond to: $2^{-1}, 2^{-2}, \dots$
 - There is an implied **1** bit before the binary point, unless we're dealing with subnormals

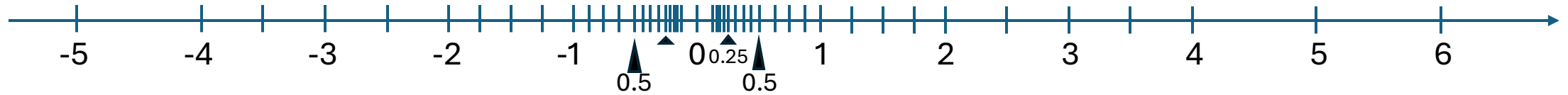
- **FP16 example:**

- E5M10: 1 sign, 5 exponent, 10 mantissa bits
 - Exponent bias: $2^{5-1} - 1 = 15$
- $0.10001.1010000000_2 = 6.5_{10}$
 - $2^{17-15} * 1.101_2 = 2^2 * (1 + 2^{-1} + 2^{-3})_{10} = 4 * 1.625_{10} = 6.5_{10}$

FP Sampling

- **Exponent bits define which powers of 2 are sampled**
 - FP16 (E5M10): exponent samples 2^{-15} to 2^{15}
 - Really 2^{-25} to 2^{15} because of subnormals, more on that later
 - Largely determines the dynamic range
- **Mantissa bits define samples between successive powers of 2**
 - E5M10: $2^{10} = 1024$ samples for each power of 2
 - Defines the *precision*
- **Key observation:**
 - FP samples the real number line with equal number of samples between each pair of adjacent powers of 2
 - Example: E5M10 has 1024 samples between 0.25 and 0.5, between 128 and 256

ExM2 Sampling of the Real Number Line



More on FP Encoding Details

- **There are a number of special values**
 - Reserved bit patterns, usually in the exponent field (all 0s or all 1s)
- **Zeros (positive and negative)**
 - Exponent and mantissa bits all set to 0
- **Subnormals:**
 - Exponent bits set to 0s, mantissa bits non-0
 - More on that in the next slide
- **Infinities (2 encodings)**
 - Positive and negative
 - Exponent bits set to 1s, mantissa bits set to 0s
- **NaNs (2^{M+1} encodings in IEEE)**
 - Exponent bits set to 1s, mantissa bits contain at least one 1
 - Generated by things like $\sqrt{-1}$, sum of positive and negative infinity, etc.

FP Subnormals

- **Special interpretation of exponent bits**
 - Just min debiased exponent value (for example, -14 for FP16)
- **No implied 1 bit before the binary point**
 - $0.00000.1010000000_2 = 2^{-14} * 0.101_2 = 2^{-15} * (1 + 2^{-2})_{10} = \dots$
- **Purpose:**
 - Makes use of the mantissa encodings when exponent is all 0s
 - Extends the dynamic range down (but with decreasing precision)
 - Extra M powers of 2 can be covered
 - More leading mantissa bits set to 0 -> lower powers of 2
 - This also implies fewer samples for those powers of 2 (fewer remaining mantisa bits)
 - $0.000000.1xxxx\dots$: 2^{-15} power of 2, 9 bits of precision
 - $0.000000.001xx\dots$: 2^{-17} power of 2, 7 bits of precision

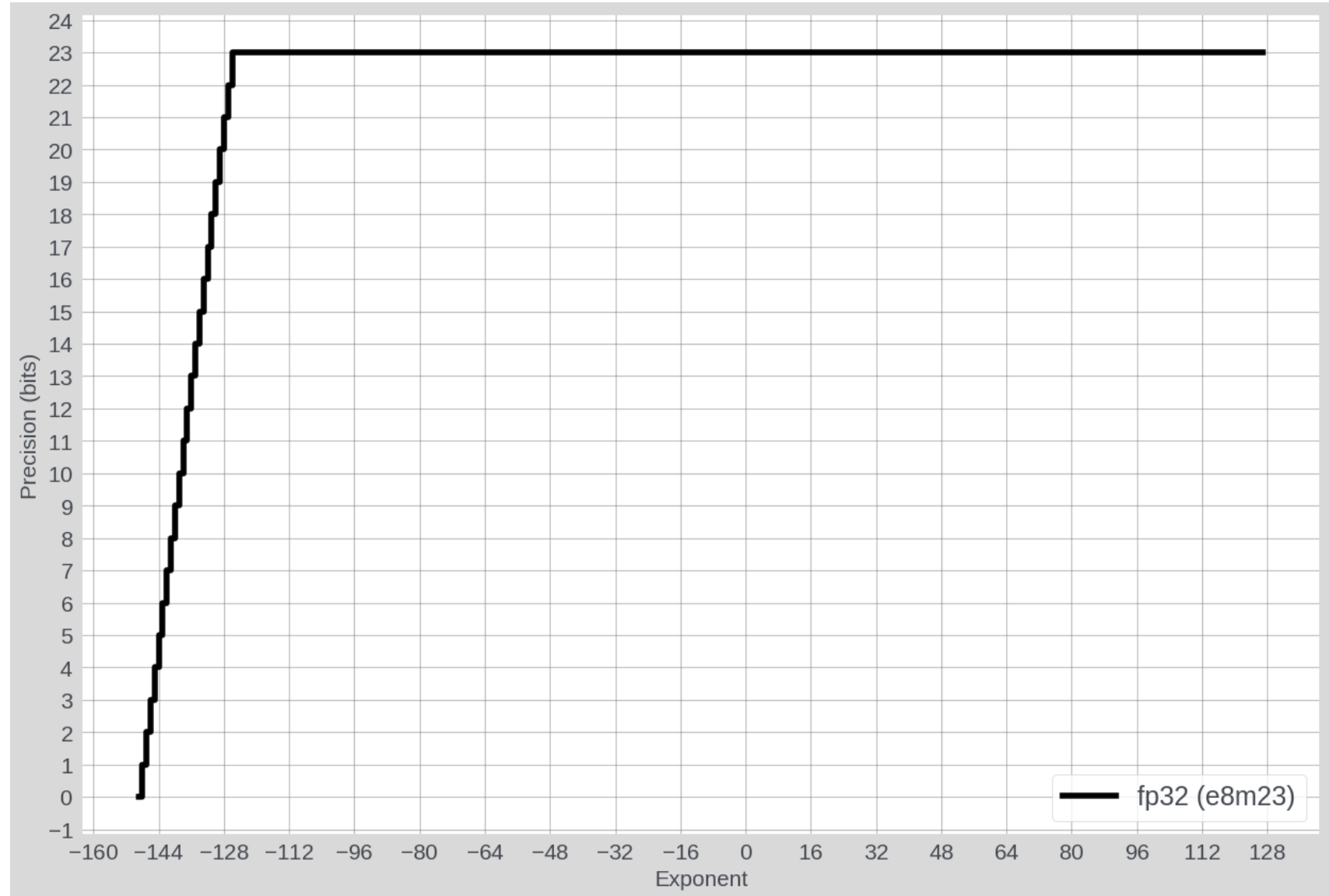
Range Precision Graph (FP32)

- **X-axis:**

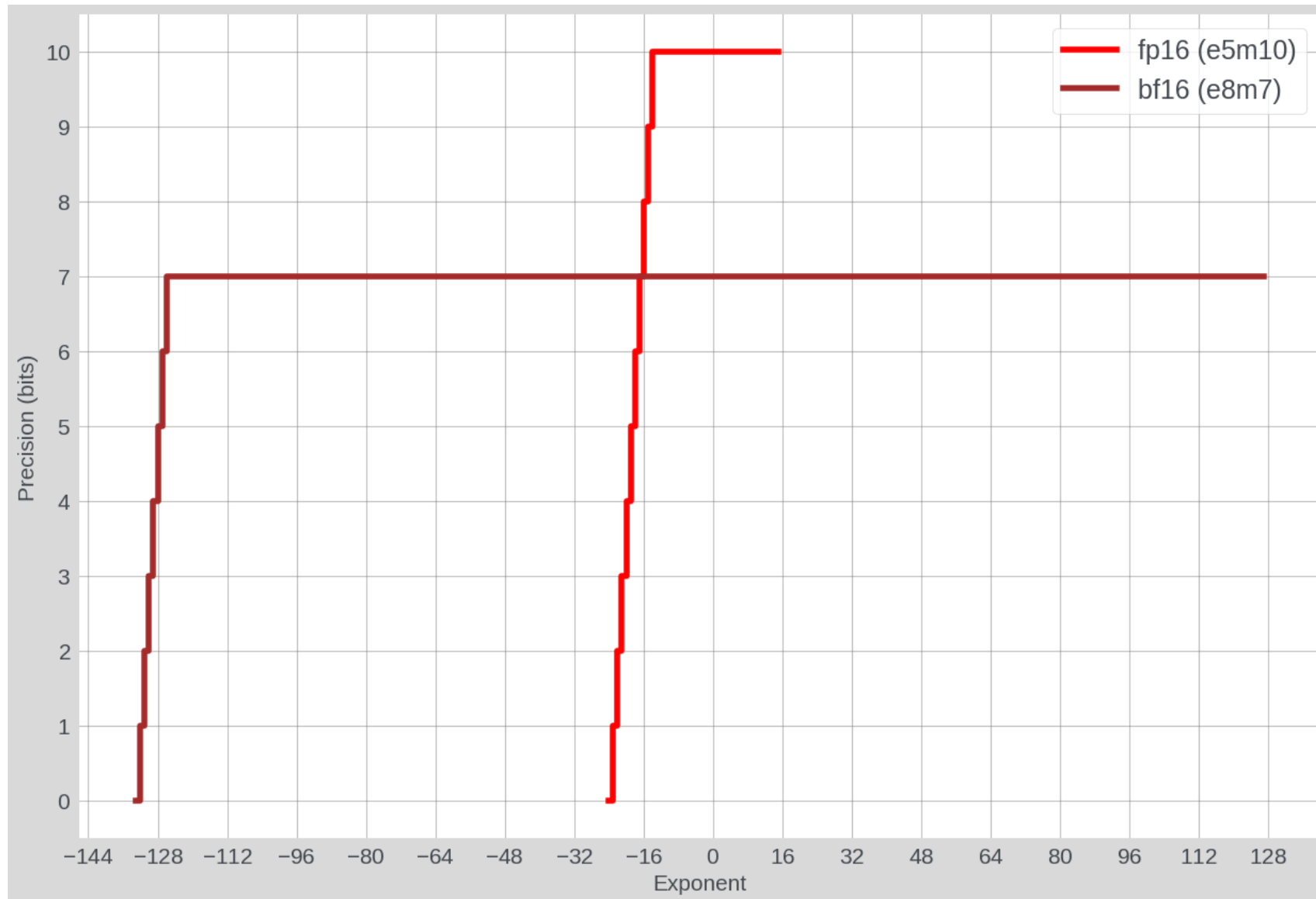
- Debiased exponent values
- Think: $\log(x)$ for any magnitude x

- **Y-axis:**

- Number of samples for that power of 2
- Aka *precision*



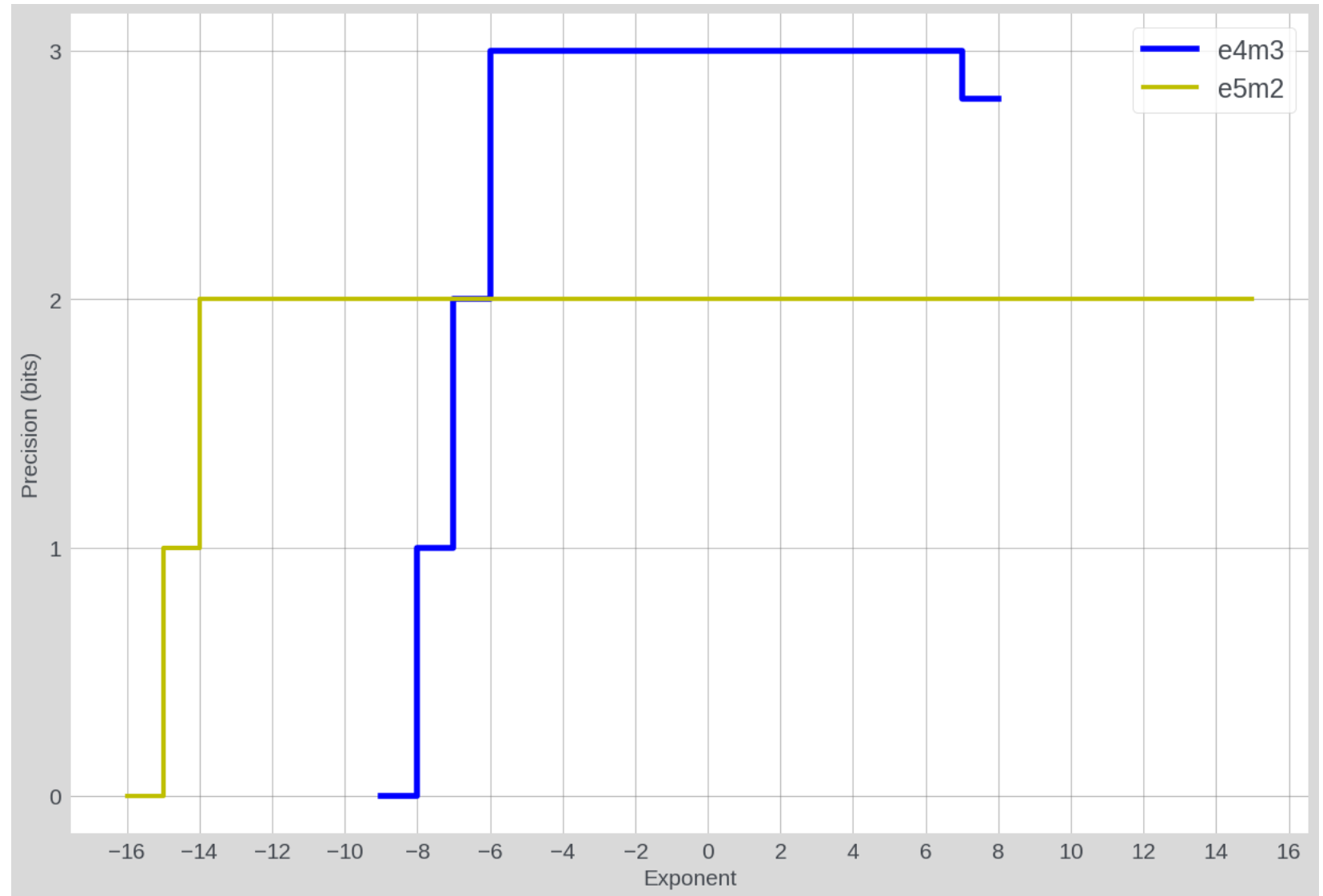
16-bit FP Formats



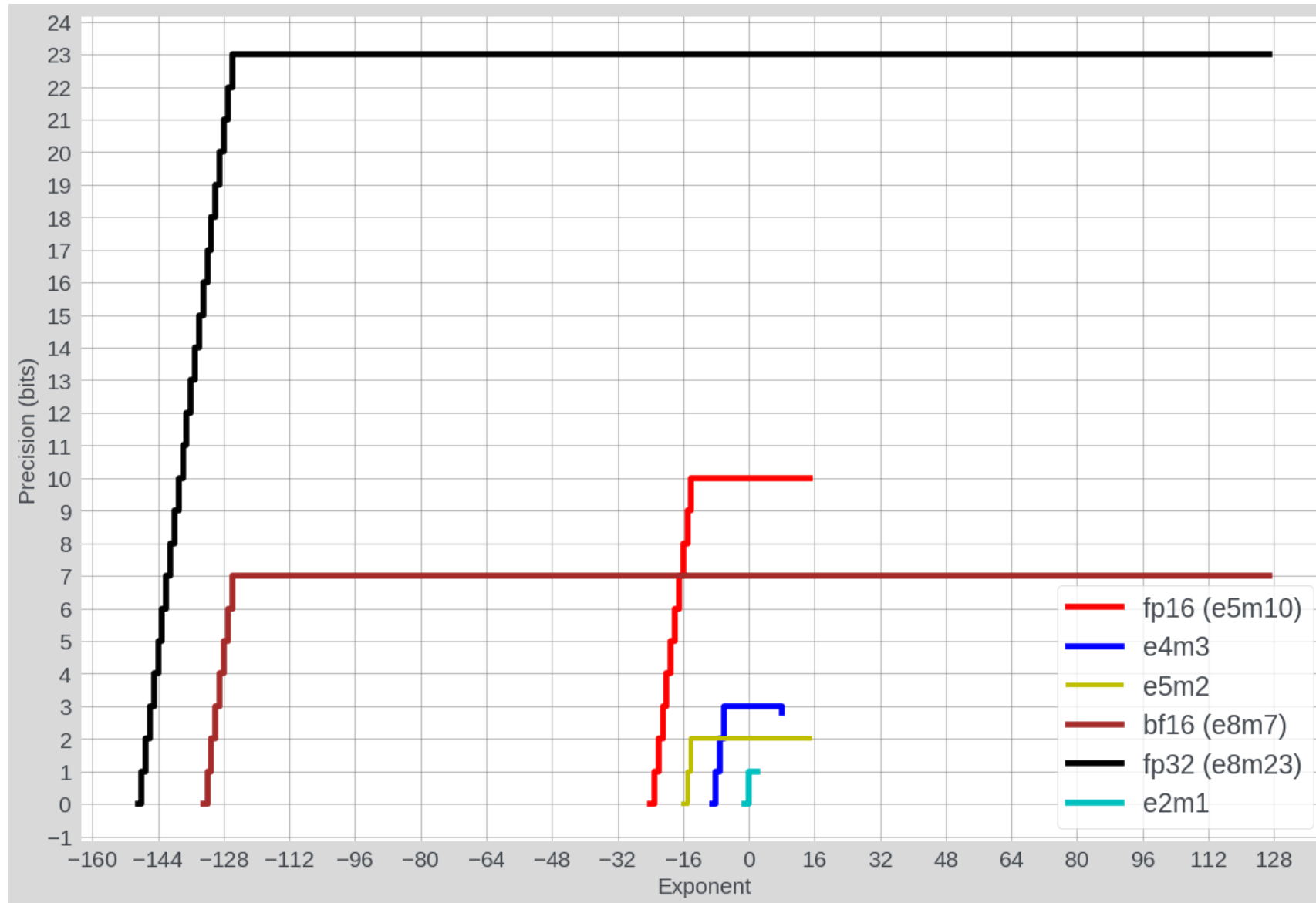
Going Smaller: FP8 Formats

- **E4M3 extends dynamic range beyond what IEEE-like encoding would do:**

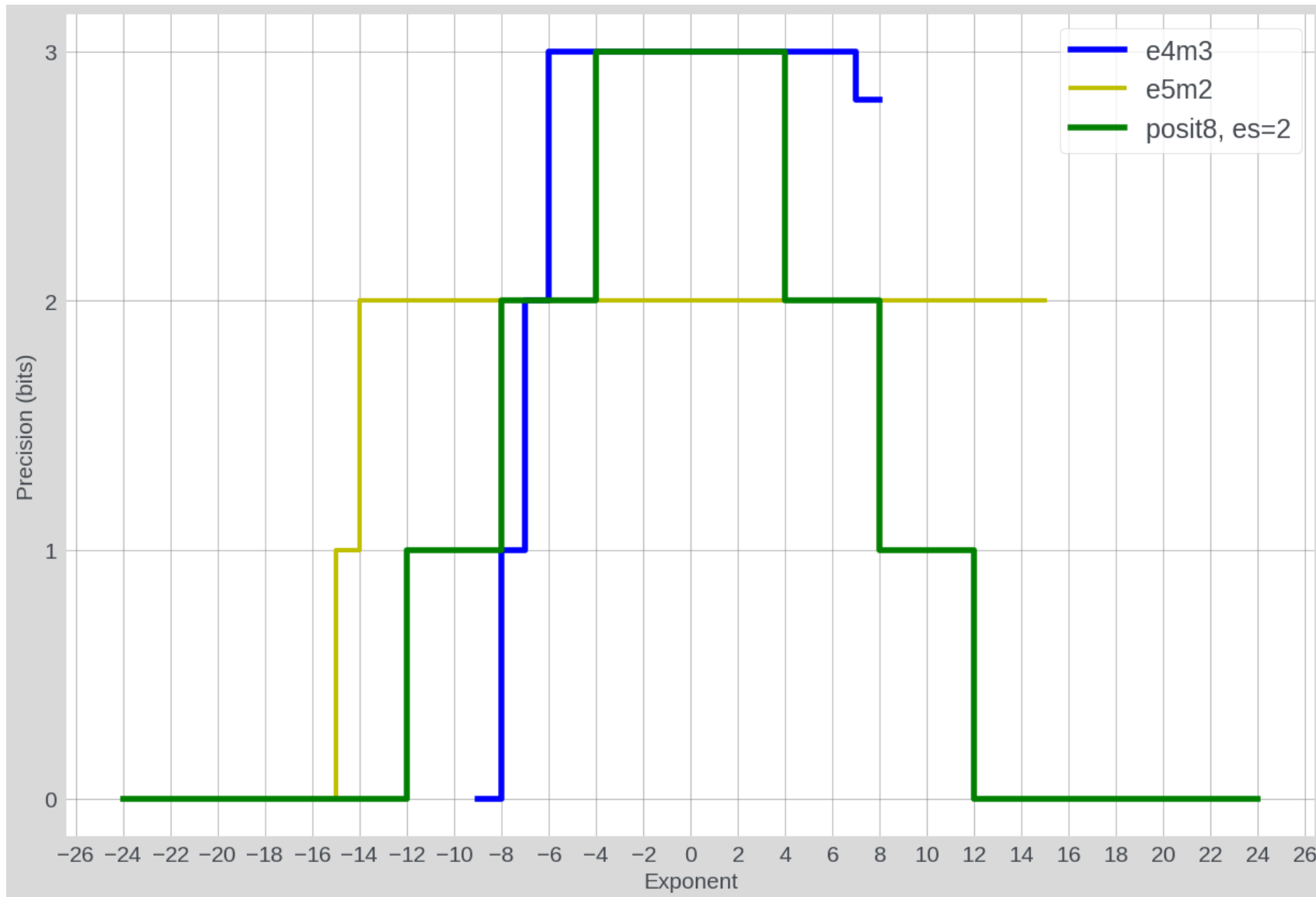
- Doesn't encode infinities
- Has only 1 NaN
 - 0.1111.110
- Because of this it adds an additional exponent 8 but with only 7 samples (the 8th one is reserved for NaN)



Altogether Now



8-bit FP formats and 8-bit Posit (es=2)

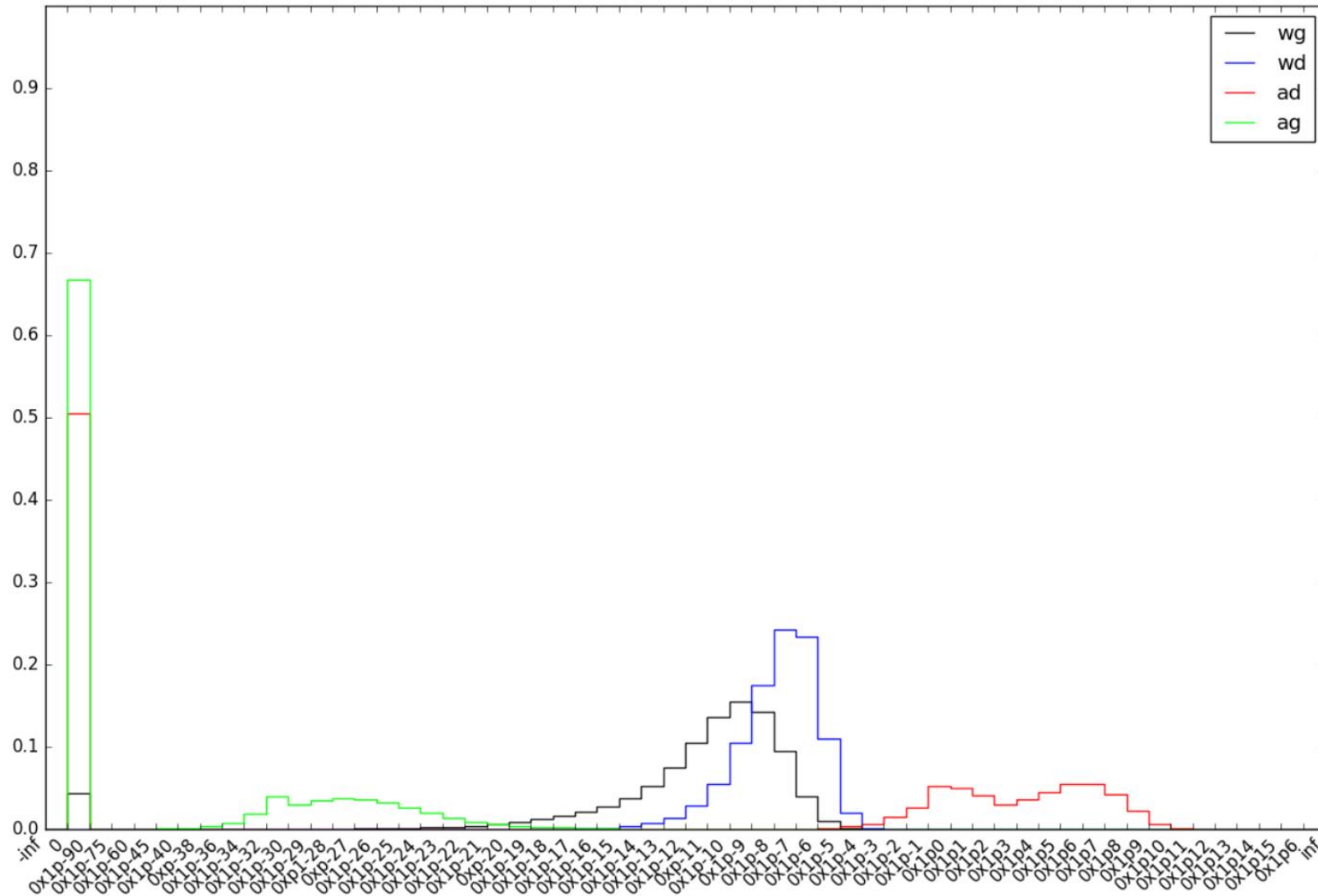


AI Numerics Journey

What Sampling Does AI Require?

- We don't know 😊
- **Lots of interest in reducing bits per value**
 - Reduces memory footprint
 - Reduces memory bandwidth pressure
 - Enables packing more math units into the same silicon area
- **Empirical studies inform/evolve mixed-precision recipes:**
 - History: FP32 -> FP16/BF16 -> FP8 -> FP4 (FP6)

CNN Training Value Histogram

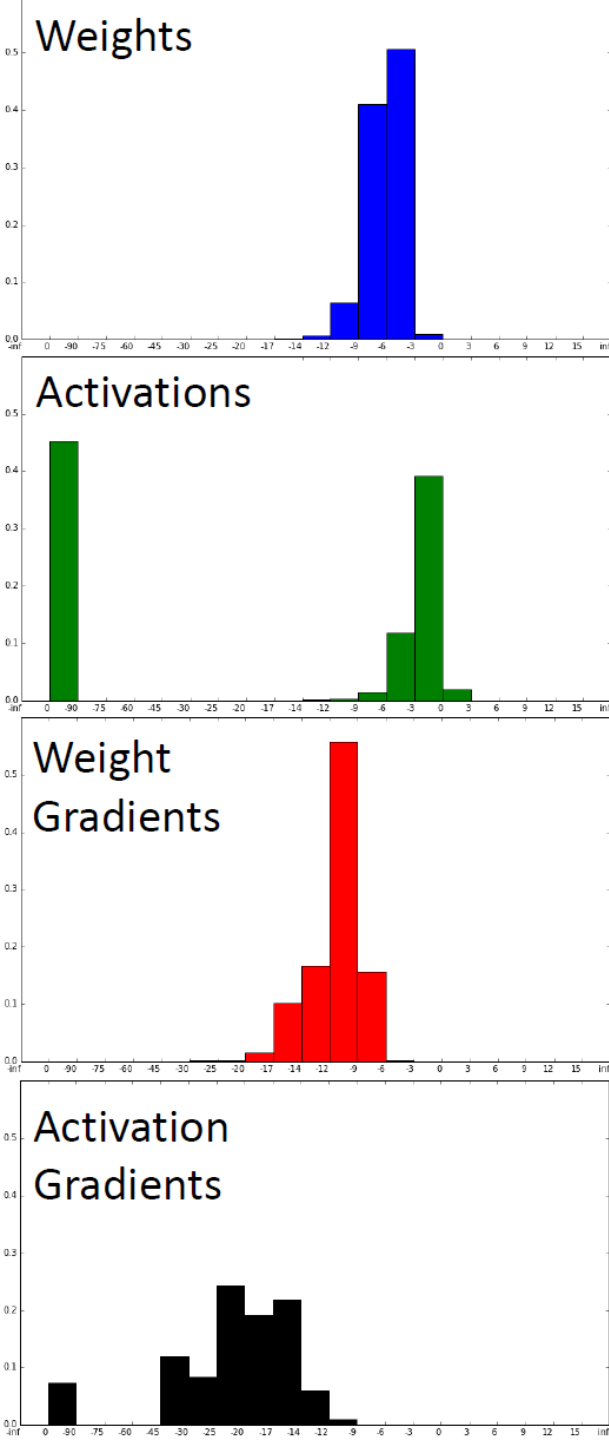


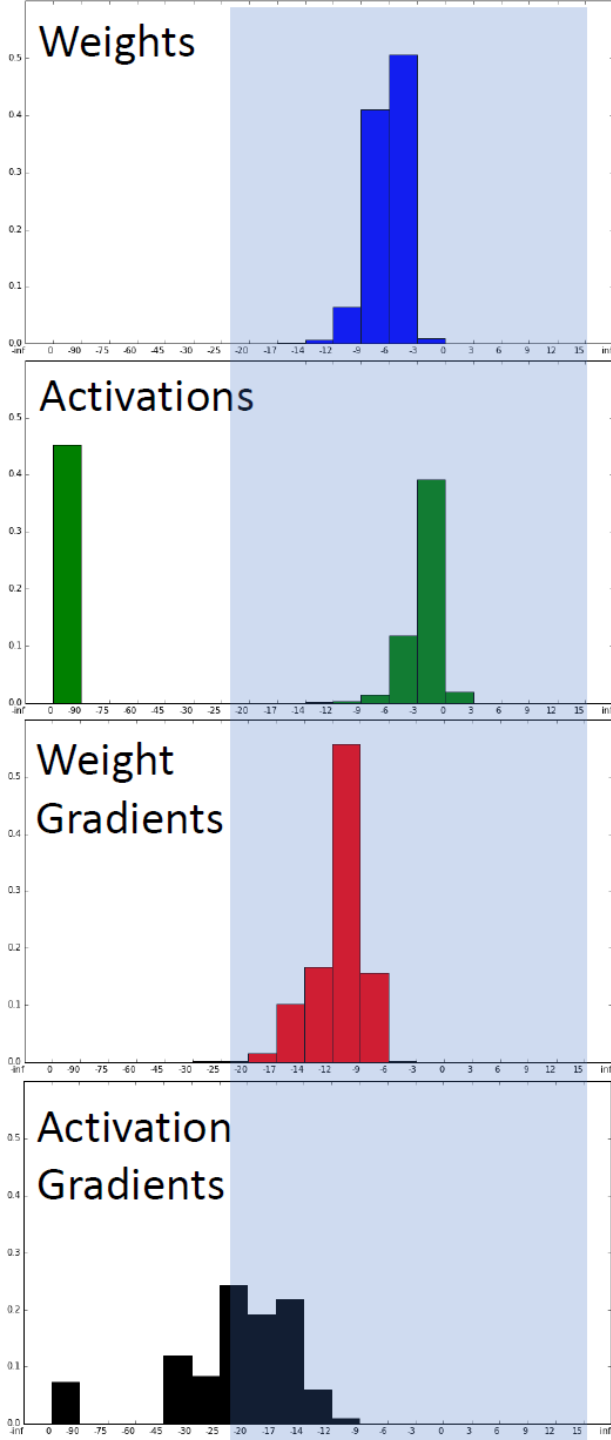
Order of increasing value magnitudes:

- Activation gradients
- Weight gradients
- Weights
- Activations

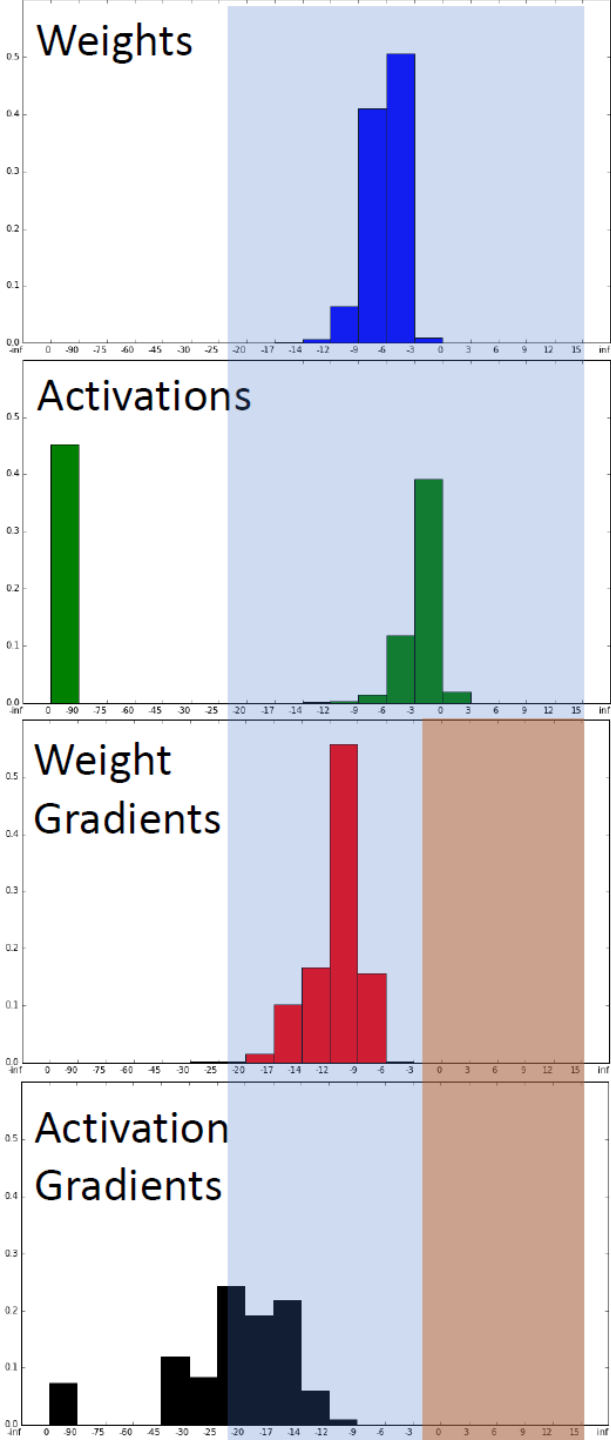
FP32 Histograms of CNN Training

- Collected over all tensors, throughout training

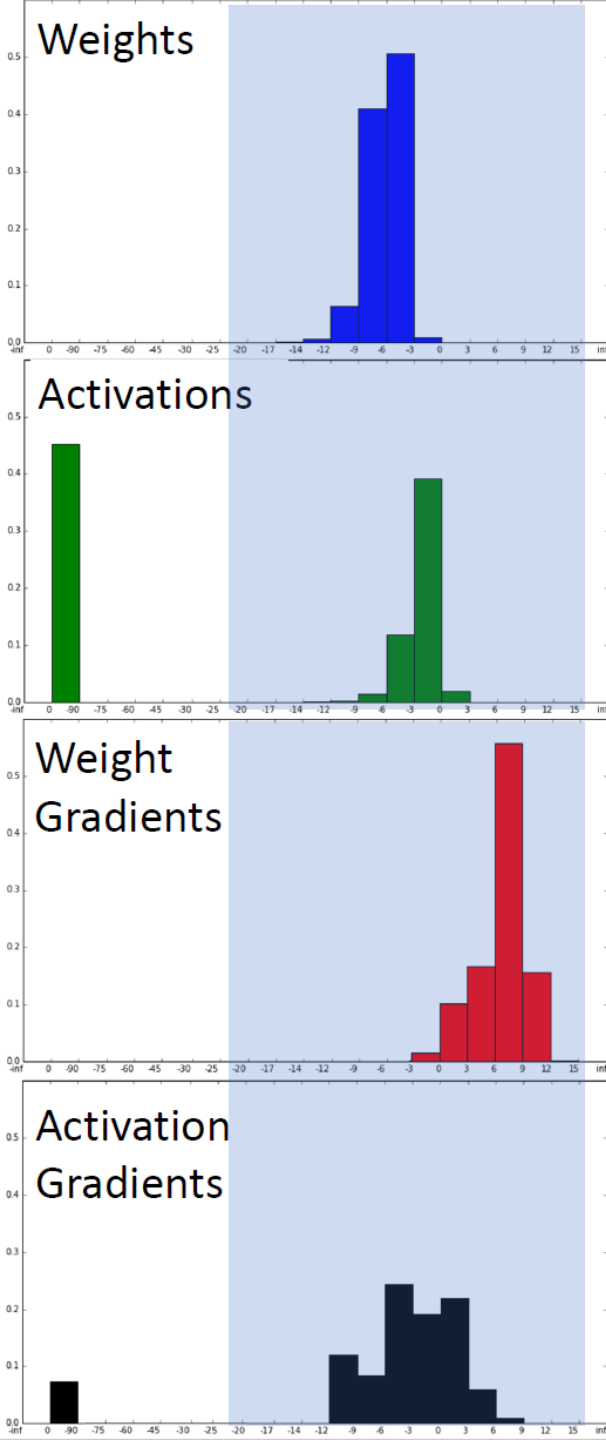




- **Blue indicates FP16 representable range**

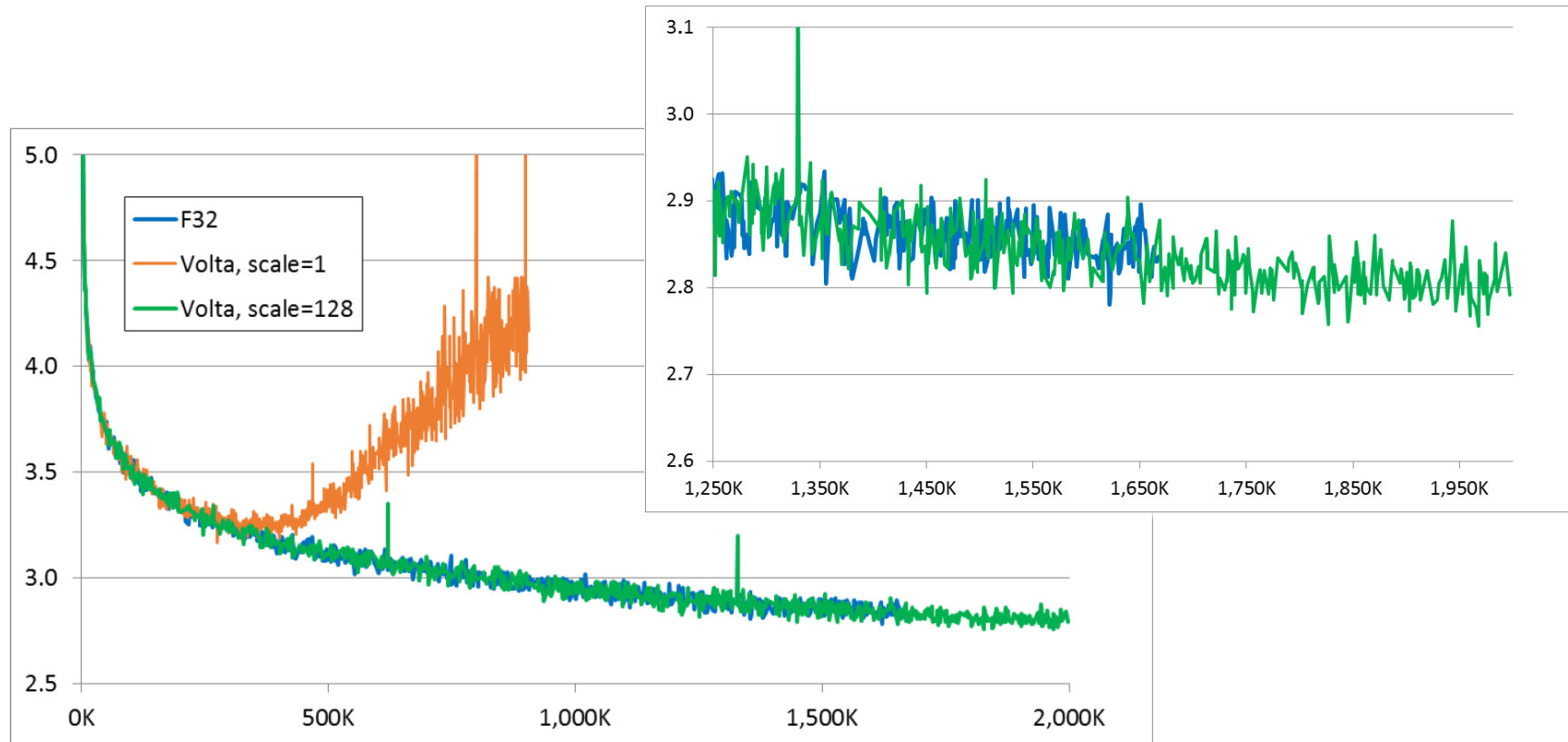


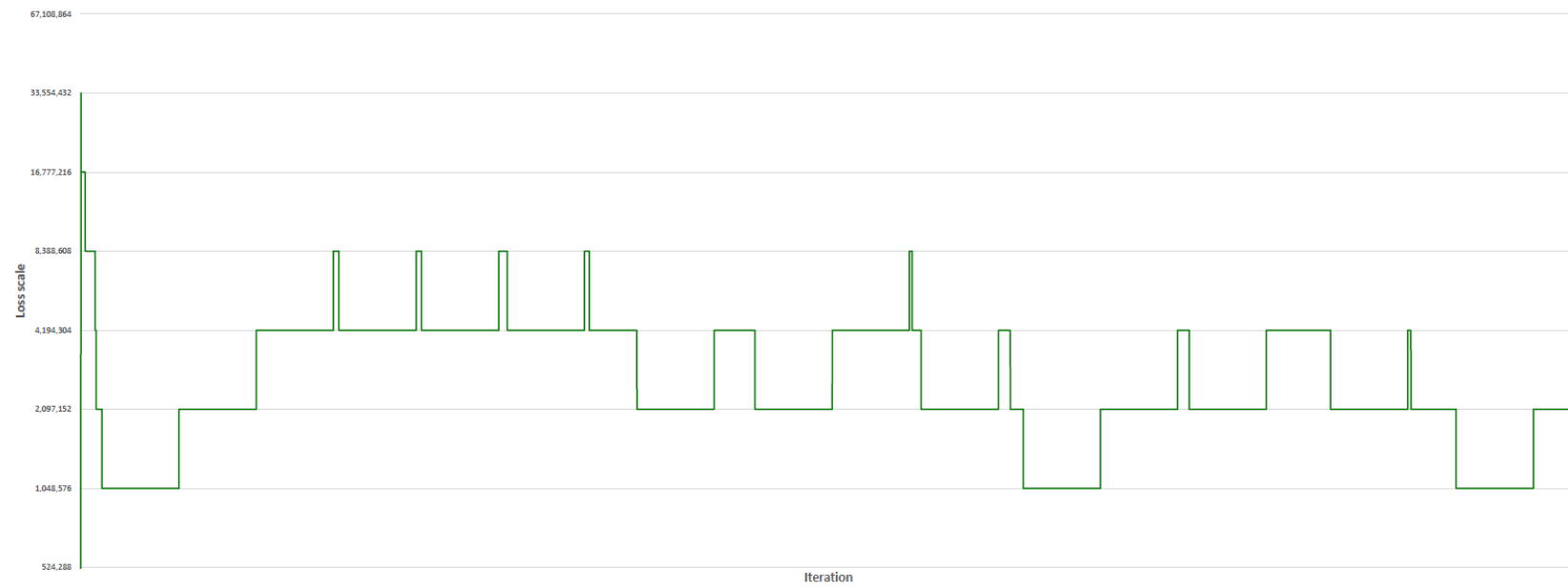
- Orange indicates unused range of FP16



- **Solution: apply a scale factor to loss**
 - by chain rule will scale all gradients
 - moves values into FP16 representable range
 - unscale prior to weight update by optimizer

- asdf

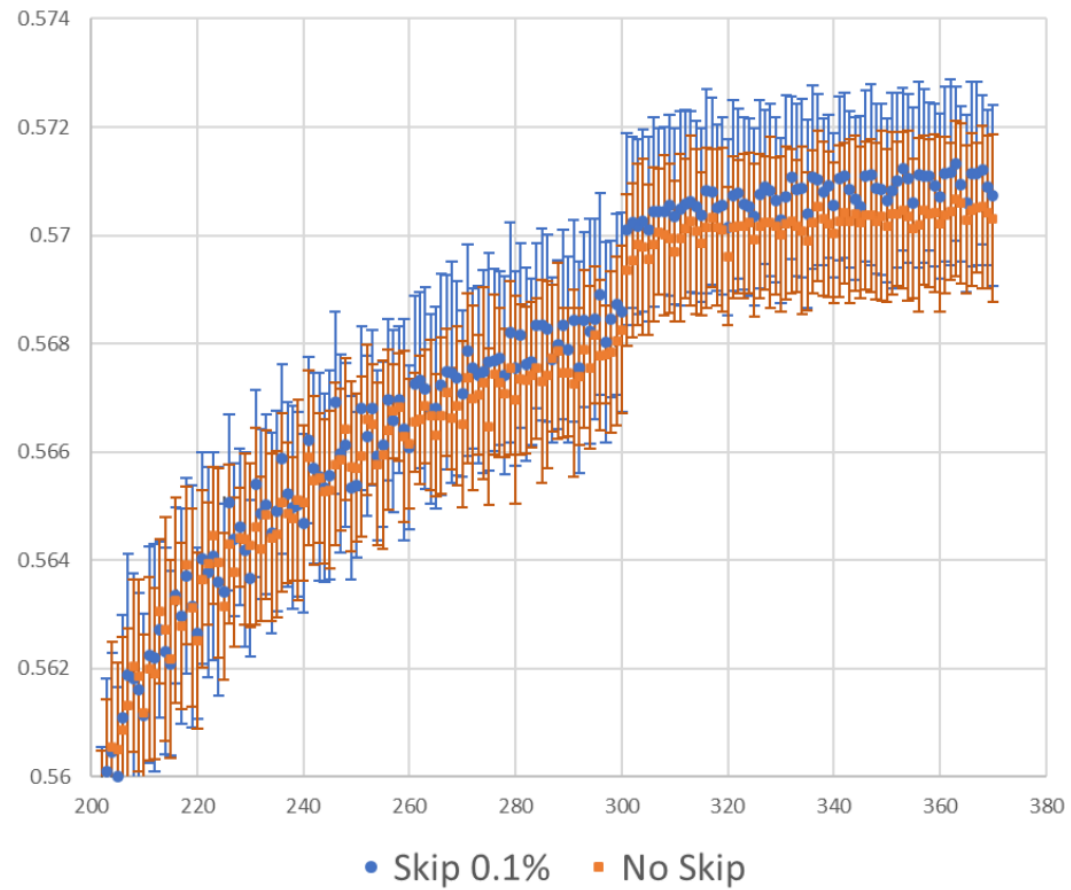




Smallest scaling factor = 2^{20} -> max dW magnitude didn't exceed 2^{-5}

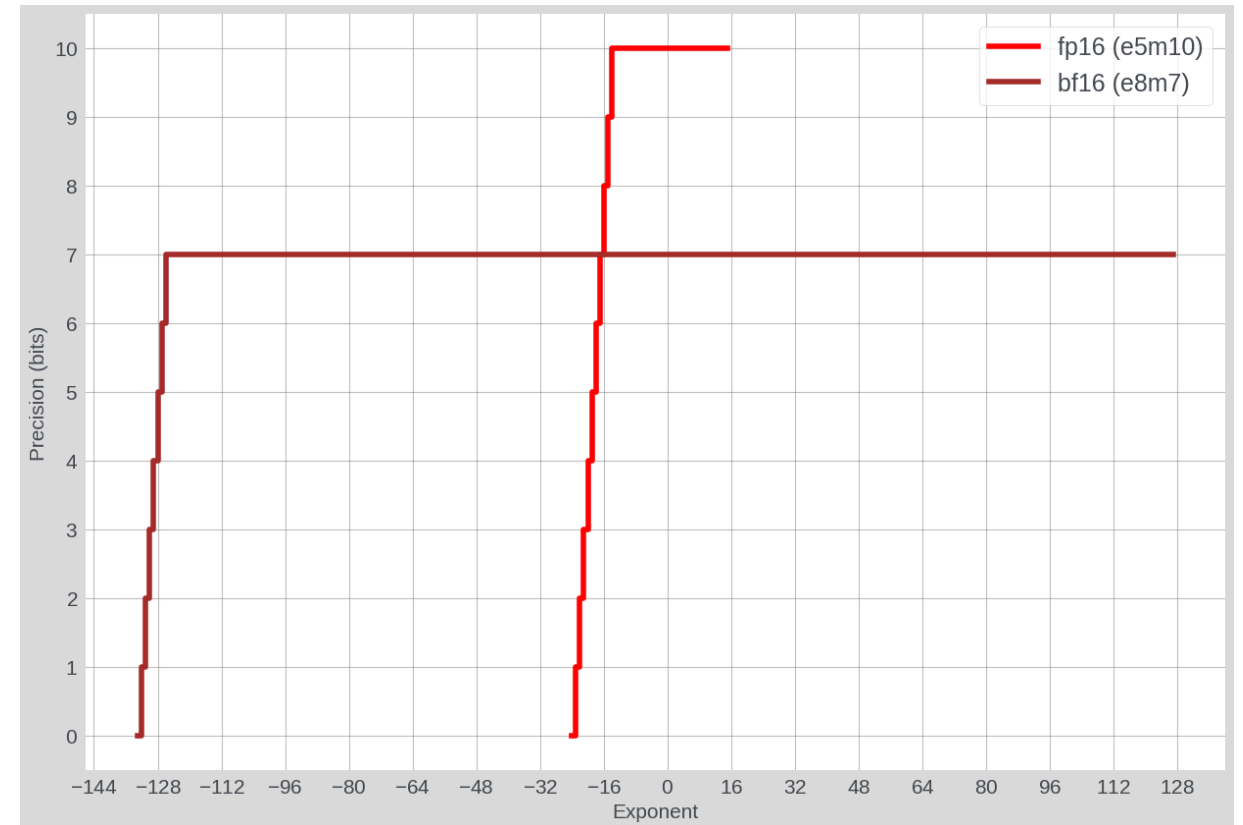
Update Skipping

- asdf



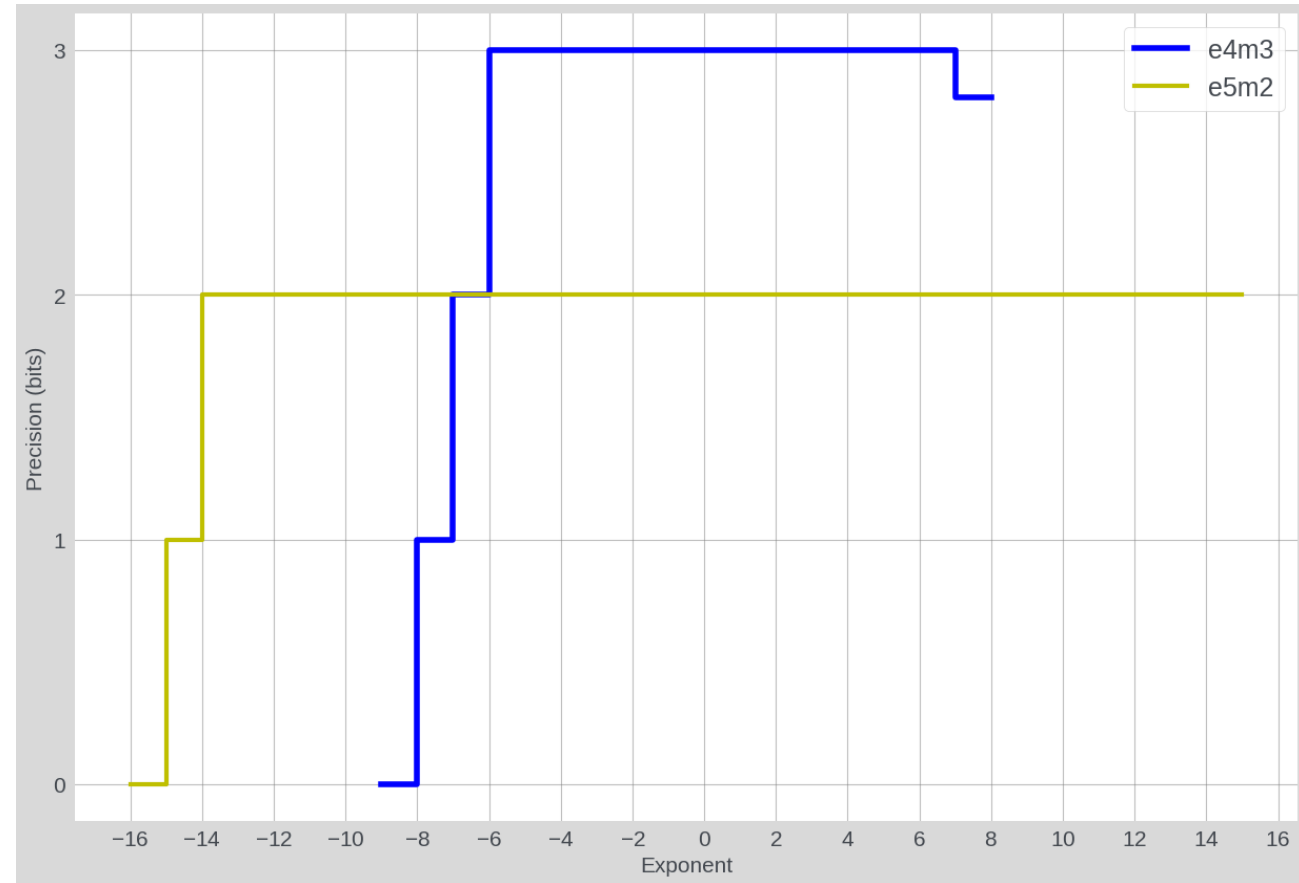
16-bit Training

- **Has long replaced FP32**
- **BF16 superseded FP16**
 - Higher dynamic range made implementations simpler
 - no need for scale factors
 - Occasionally cases come up where FP16's larger mantissa is beneficial
 - Some evidence of long sequences and attention softmax



FP8

- **Two formats:**
 - E4M3
 - E5M2
- **Several variants:**
 - OCP, Graphcore, Tesla, IEEE working group
 - Differences
 - +-0 or only one zero
 - Exponent bias value
 - More philosophical than impactful on AI workloads



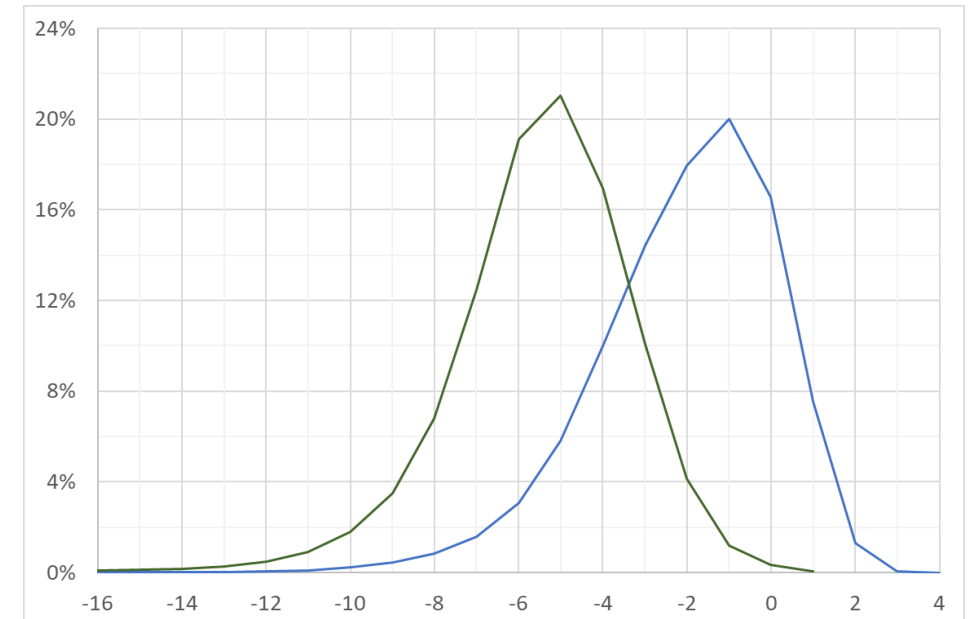
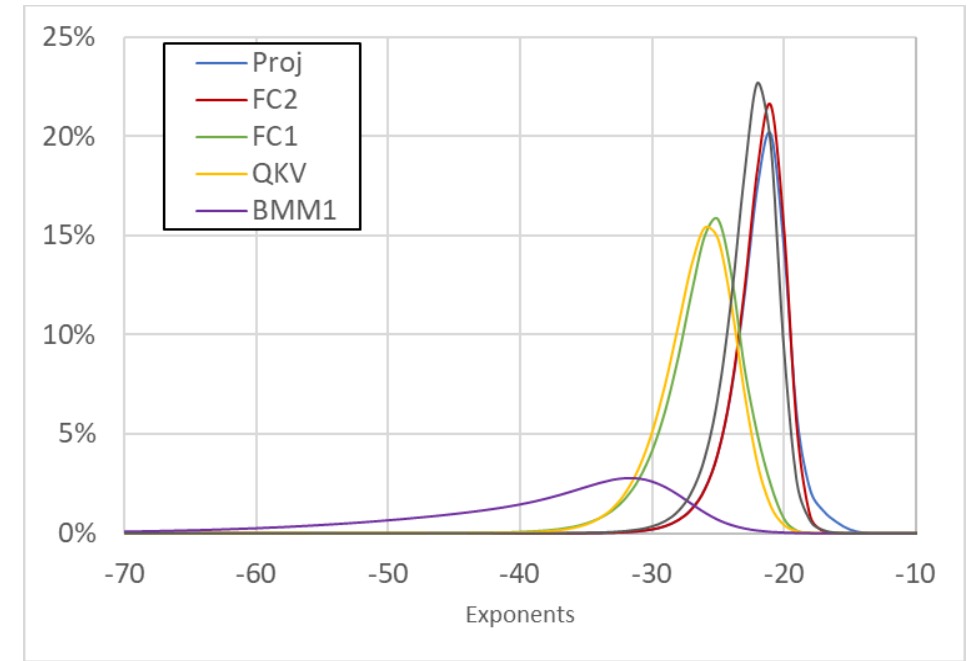
FP8 Dynamic Range

- **FP8 Dynamic range:**

- E5M2: ~32 binades
- E4M3: ~18 binades
- Insufficient to have a single scale factor for all tensors

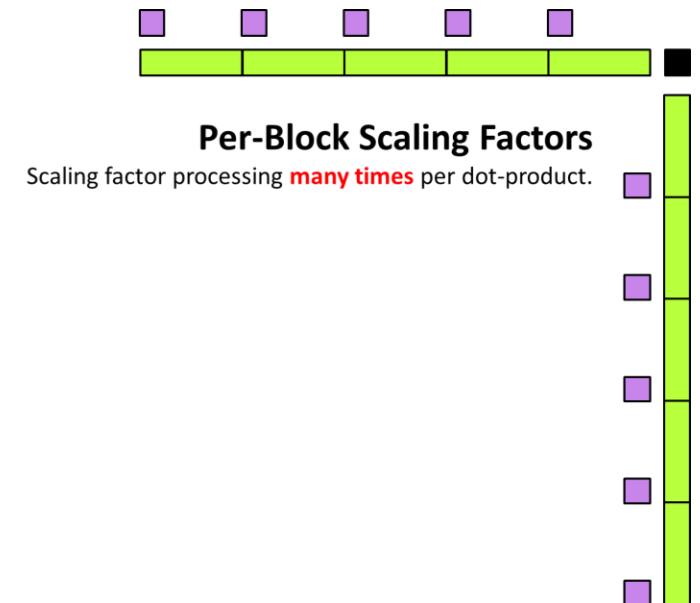
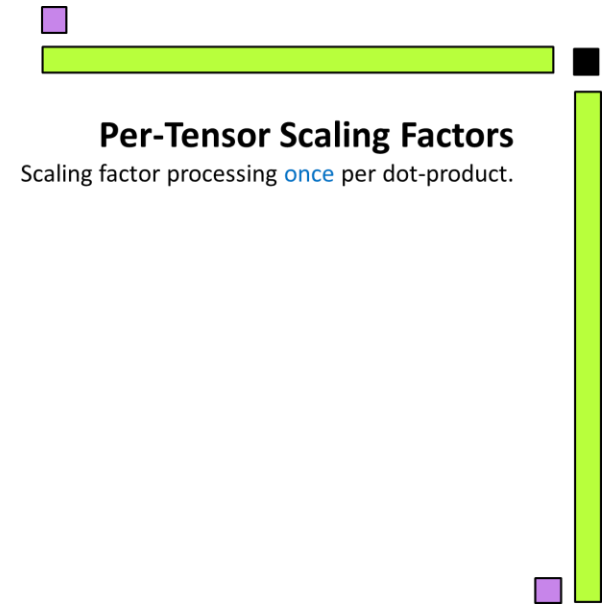
- **Scale factor granularity:**

- Per-tensor: works for some networks
- Per-subtensor: needed by some networks
 - Per-row: cheaper computationally
 - Per-block: pre-Blackwell/MI300 (MXFP support) gets expensive as blocks get smaller



Scale Factor Granularity

- **Result quality: benefits from smaller blocks**
 - Required dynamic range is narrower
 - Large “outlier” magnitudes have a smaller “blast radius”
 - Flush to 0 fewer other values
- **Compute performance (speed):**
 - Matrix multiplies benefit from larger blocks
 - Encoding can benefit from smaller blocks



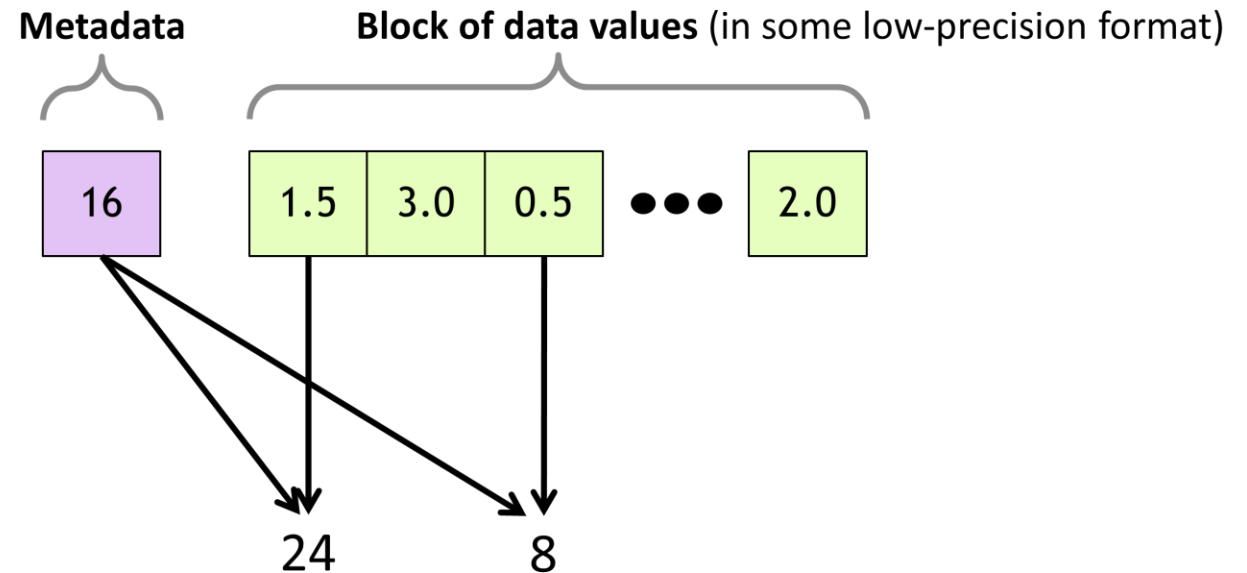
OCP MX Block Formats

- **Fine-grained scaling:**

- Data: 32-element blocks
 - OCP FP8, FP6, FP4 encoding of elements
- Metadata: E8M0 scale factor

- **HW support:**

- No slowdowns for GEMMs
- NVIDIA Blackwell
- AMD MI300



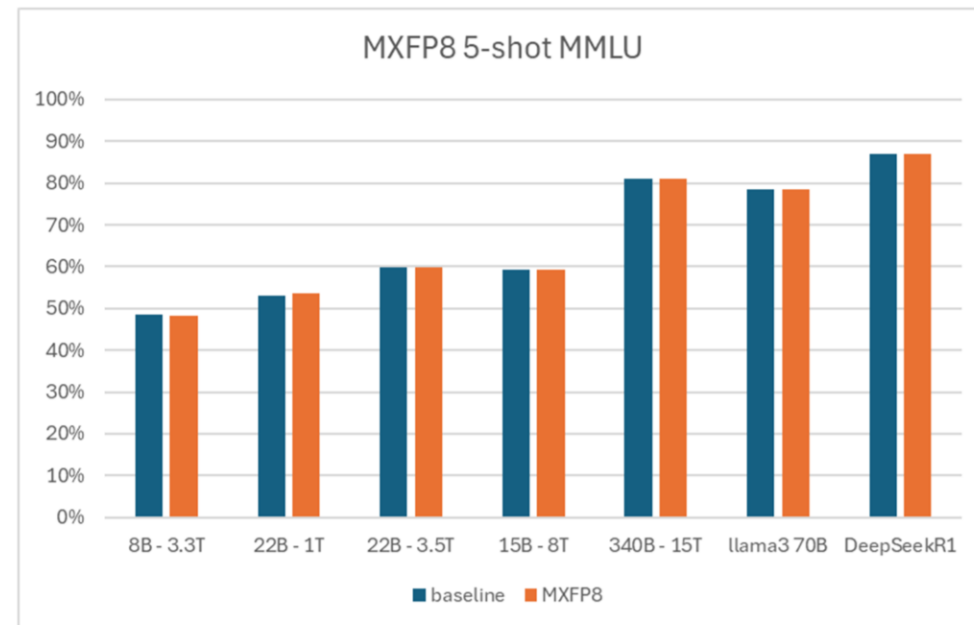
MXFP8 Inference

- **Models trained in higher precision (BF16, FP16):**

- Don't need QAT, don't need PTQ
- Inference engine:
 - Converts model params to MXFP8 offline
 - Converts activations to MXFP8 at run-time (metadata is computed per block, based on its values)

- **Empirical data:**

- DeepSeekR1:
 - FP8: 87.05
 - MXFP8: 87.03
- Llama 3 70B:
 - BF16: 78.6
 - FP8: 78.4
- Nemotron 340B:
 - BF16: 81.1
 - MXFP8: 81.0



Baseline:

- DeepSeek R1: FP8
- All others: BF16

MXFP8 Training

- **Only E4M3**
 - Including activation gradients
 - E5M2 was needed for “vanilla” FP8 with coarser scale factor granularity
 - Using E4M3 enables quantizing more layers, which needed more precision

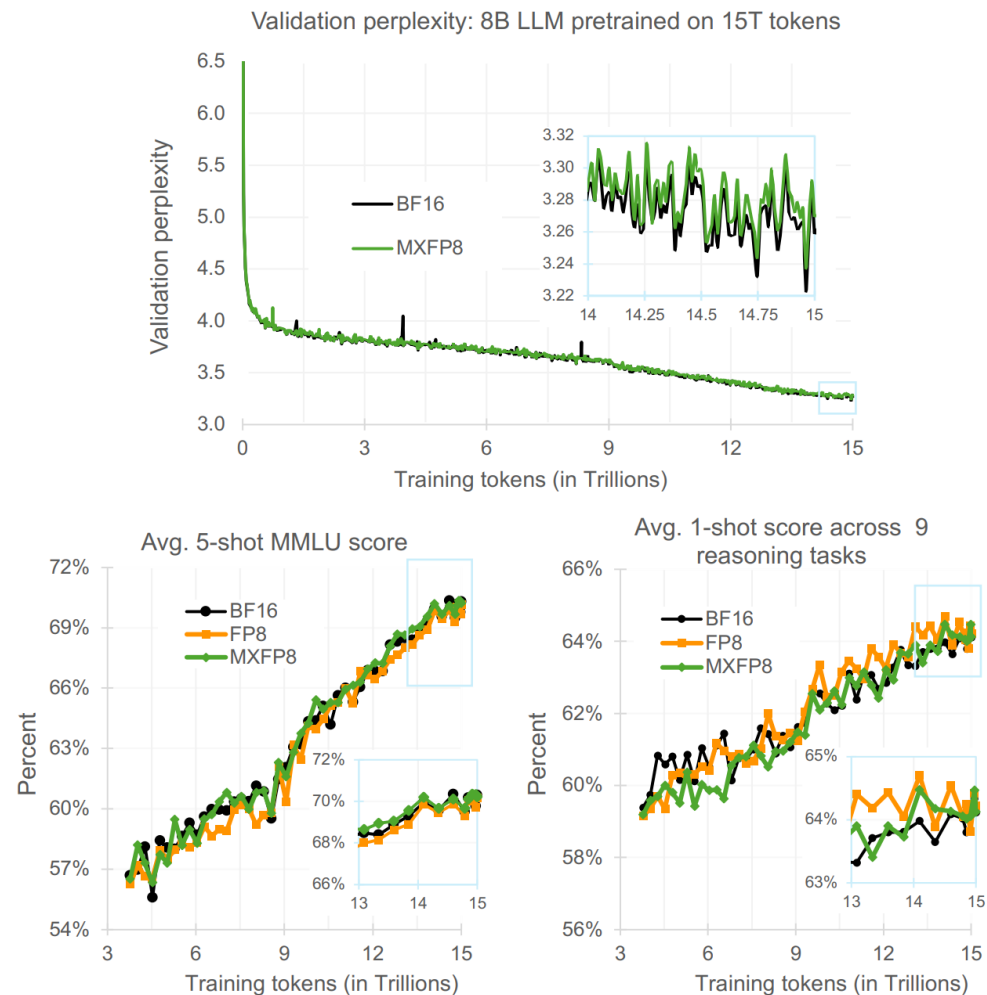


Figure 2: Pre-training a 8B LLM on 15T tokens. **Top:** Training behavior of BF16 vs MXFP8. **Bottom:** Comparing BF16, FP8 and MXFP8’s downstream task scores on MMLU and a set of 9 reasoning tasks. MXFP8 numerics use our proposed rounding method and E4M3 for all quantized tensors.

FP4

- **OCP MXFP spec defines the encoding**
 - E2M1: representable magnitudes are 0, 0.5, 1, 1.5, 2, 3, 4, 6
 - No infinities, no NaNs (rely on block exponent to signal that)
 - Enables to get an extra sample of the real number line

NVFP4 PTQ Inference

- **Observations:**

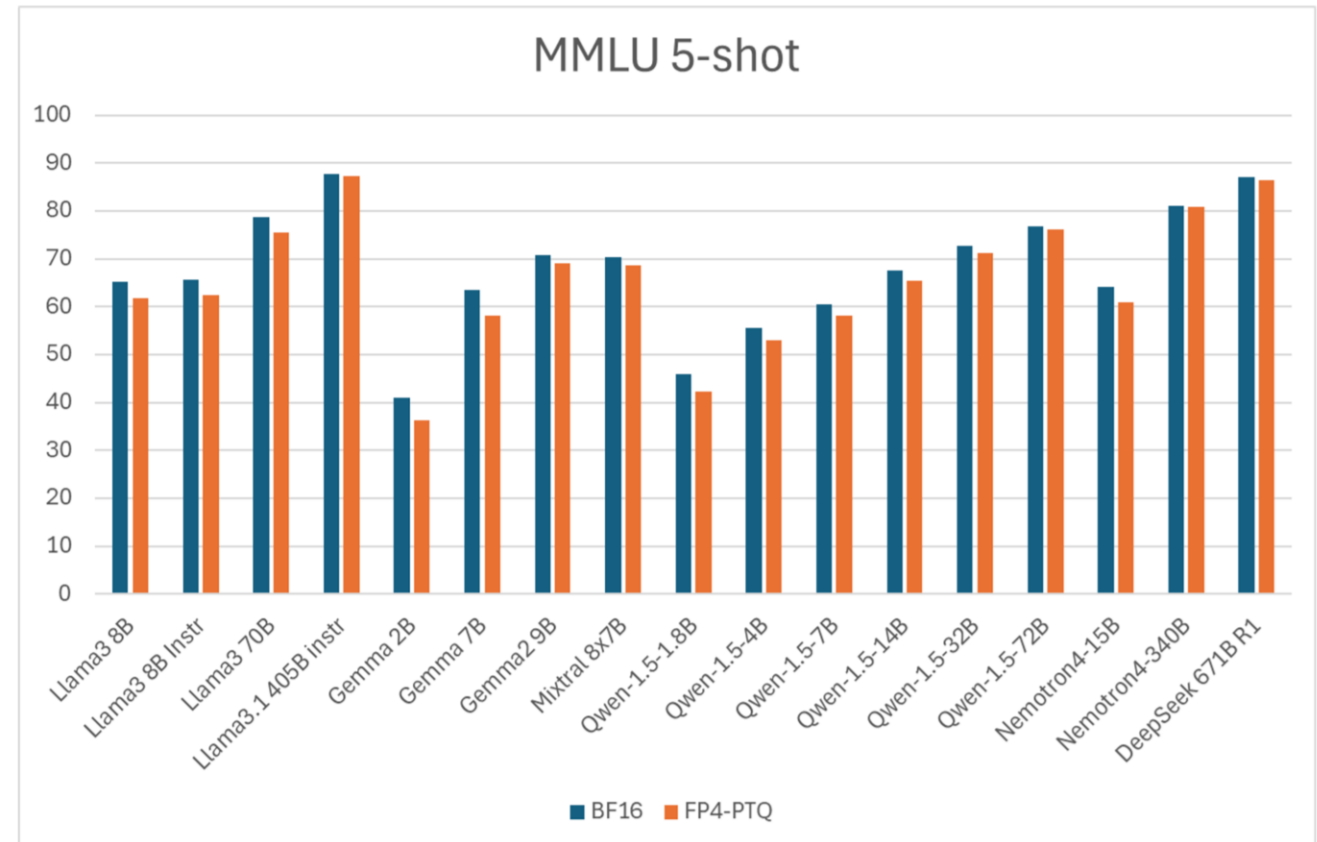
- PTQ reduces accuracy by a couple percentage points
- Larger models retain more accuracy
 - Stay within 0.5 percentage points
 - DeepSeek 671B R1
 - Llama3.1 405B instruct
 - Nemotron 340B

- **Quantization:**

- Weights and activations of linear layers of all transformer blocks
 - QKV, Proj, MLP
- MoE router: left in FP8
 - No impact on speed

- **Calibration:**

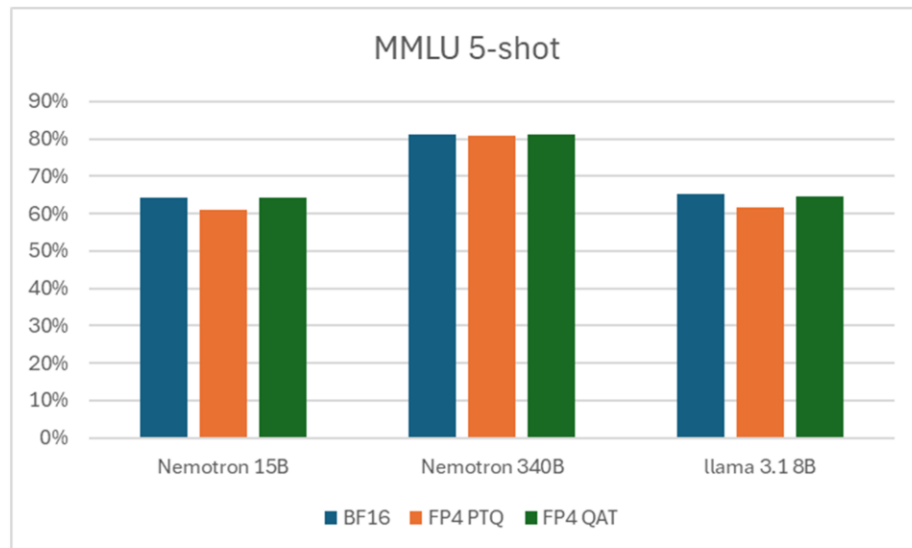
- Max-calibration
- Datasets:
 - Wikitext-103
 - CNN_DailyNews



NVFP4: Going Beyond PTQ

- **A few options to recover accuracy for cases where PTQ is not sufficient**

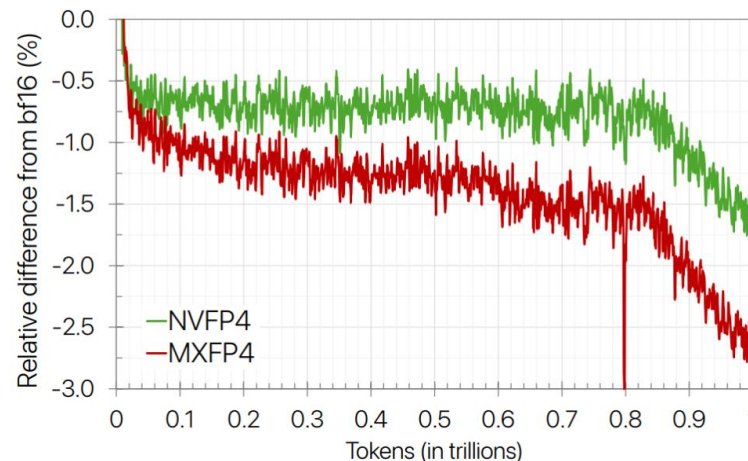
- Quantize a subset of operations
- Quantize only weights for all operations
 - Does not accelerate math
 - Helps with memory footprint and bw pressure – may be of benefit for low-batch gen-phase of inference
- Apply AWQ and similar techniques
- QAT: Quantization Aware Training



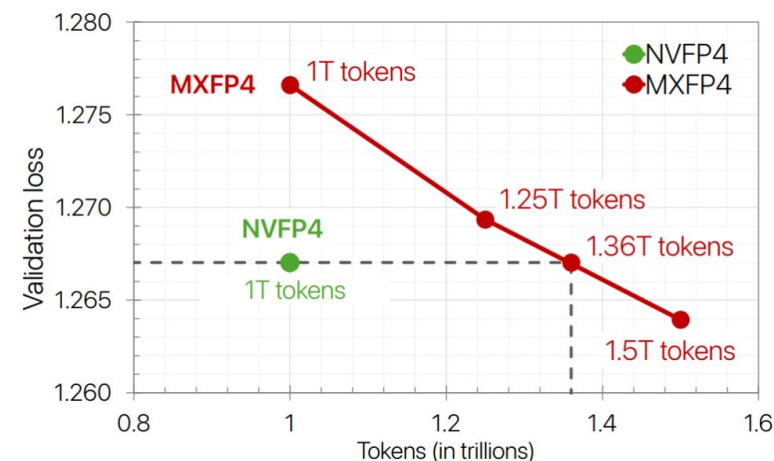
- **Nemotron QAT:**
 - QAT with original task loss
 - Using less than 0.05% of original data
- **Llama QAT:**
 - Distillation for each linear layer, using a subset of NVIDIA dataset

FP4 Training

- **MXFP4:**
 - 32-element block
 - E8M0 scale factor
- **NVFP4:**
 - 16-element block
 - E4M3 scale factor



(a) Relative difference between training loss of BF16 (baseline) and NVFP4 and MXFP4 pretraining.



(b) Final validation loss for NVFP4 and MXFP4 pretraining with different number of tokens.

Figure 6 | NVFP4 vs MXFP4 comparisons: (a) training-loss difference; (b) validation perplexity across token budgets.

More on Scale Factors

- **Scale factor selection recipes can vary**
- **Classical recipe:**
 - Find `amax` (maximum absolute value) in a block of higher precision values
 - Compute `encode_scale` so that `amax * encode_scale` is as close to the destination format maximum without overflowing
 - Some FP8 training sessions were falling behind when overflowing and saturating
- **Scale factor format may make a difference**
 - More scale factor precision can expose more of the format's range
 - More scale factor precision can more accurately represent block values

Scale Factor Precision Example

- **FP4 samples: 0, 0.5, 1, 1.5, 2, 3, 4, 6**
- **Say, block amax is 3.6**
- **E8M0 scale factor**
 - scale = 1 (2 too large since $2 * 3.6 = 7.2$ and would overflow 6)
 - 3.6 gets rounded up to **4** - we “waste” sample 6 of FP4 (~15% of samples)
- **E4M3 scale factor**
 - Scale = 1.625
 - $1.625 * 3.6$ rounds to **6** – we no longer waste 15% of samples
 - Also, 3.6 gets more accurately represented when decoded
 - **6** / 1.625 = 3.6923
 - **4** / 1.000 = 4

Other Work

- **Binary and Ternary Training**

- Train with $\{0, 1\}$ values, $\{-1, 0, 1\}$ values
 - With scale factors, of course
- One paper (there are more): [\[2402.17764\] The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits](#)

- **Codebooks**

- Use look up tables to compress memory footprint
- Example: 16-entry table \rightarrow 4-bit indices, values can be higher precision
- DNN use dates back to at least \sim 2015 work by Song Han et al
 - [\[1510.00149\] Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding](#)

Techniques to Adjust Networks for Low Precision

- **Active area of research**
- **General idea:**
 - Apply some transformations to tensors to reduce the required dynamic range
- **Examples:**
 - Additional normalizations
 - AWQ, SmoothQuant, SVDQuant, RHT, etc.

FP Quirks

Quirks

- **Non-associativity**
- **FMA**
- **Rounding is pretty much the default reason for these**

FP Non Associativity

- **In FP the order of additions matters:**
 - $(A + B) + C$ is not always the same as $A + (B + C)$
 - Rounding to a finite number of samples is the cause (more on this later)
- **Things that can change the order of operations:**
 - Different HW: different dot-product instruction sizes
 - Same HW: different compiler optimizations, use of atomics, work-queue based implementations

FP “Swamping”

- Swamping occurs when addition of two different magnitudes results in a sum that’s just the larger magnitude
- Example with E5M2:

- $32 + 3.5 = 32$

- $1.00_2 \times 2^5 + 1.11_2 \times 2^1 = 1.00_2 \times 2^5 + 0.000111_2 \times 2^5 = 1.00\textcolor{red}{0111}_2 \times 2^5 = 1.00_2 \times 2^5$

Align the binary point prior to addition
(make exponents equal to the larger of the 2)

Round the result mantissa to 2 bits
(in this cases loses precision)

Effect of FP Addition Ordering

- **Example with E5M2:**

- Decimal: $32 + 3.5 + 4$
- Binary E5M2: $1.00_2 \times 2^5 + 1.11_2 \times 2^1 + 1.00_2 \times 2^2$

- **$(32 + 3.5) + 4 = 32$**

- $32 + 3.5 = 1.00_2 \times 2^5 + 0.000111_2 \times 2^5 = 1.000111_2 \times 2^5 = 1.00_2 \times 2^5 = 32$
- $32 + 4 = 1.00_2 \times 2^5 + 1.00_2 \times 2^2 = 1.00_2 \times 2^5 + 0.001_2 \times 2^5 = 1.001_2 \times 2^5 = 1.00_2 \times 2^5 = 32$

- **$32 + (3.5 + 4) = 40$**

- $3.5 + 4 = 1.11_2 \times 2^1 + 1.00_2 \times 2^2 = 0.111_2 \times 2^2 + 1.00_2 \times 2^2 = 1.111_2 \times 2^2 = 1.00_2 \times 2^3 = 8$
- $32 + 8 = 1.00_2 \times 2^5 + 1.00_2 \times 2^3 = 1.00_2 \times 2^5 + 0.01_2 \times 2^5 = 1.01_2 \times 2^5 = 40$

Note: we relied on swamping for this illustration, but there cases without the occurs without swamping

Fused Multiply Add

- **Performs $a * b + c$ in a single instruction**
 - Defined by IEEE 754, implemented by most modern chips (GPUs, CPUs)
 - Product $a * b$ required to be kept in full precision (i.e. no rounding prior to addition)
- **In most cases FMA is more accurate than FMUL, FADD pair**
- **But can lead to results that are surprising at first**
 - Classical example by Kahan: $x * x - y * y$ may not be 0 even if $x = y$
 - Occurs when $x * x$ product requires more bits than available in format (we round)
 - FMA implementation:
 - `temp = y * y` // Rounding will happen
 - `FMA(x, x, -temp)` // No rounding of $x * x$
 - FMUL, FADD implementation:
 - rounding for both $x * x$ and $y * y$
- **Compiler optimizations affect how much FMA is used**
 - Compilers usually provide flags to control this

Backup

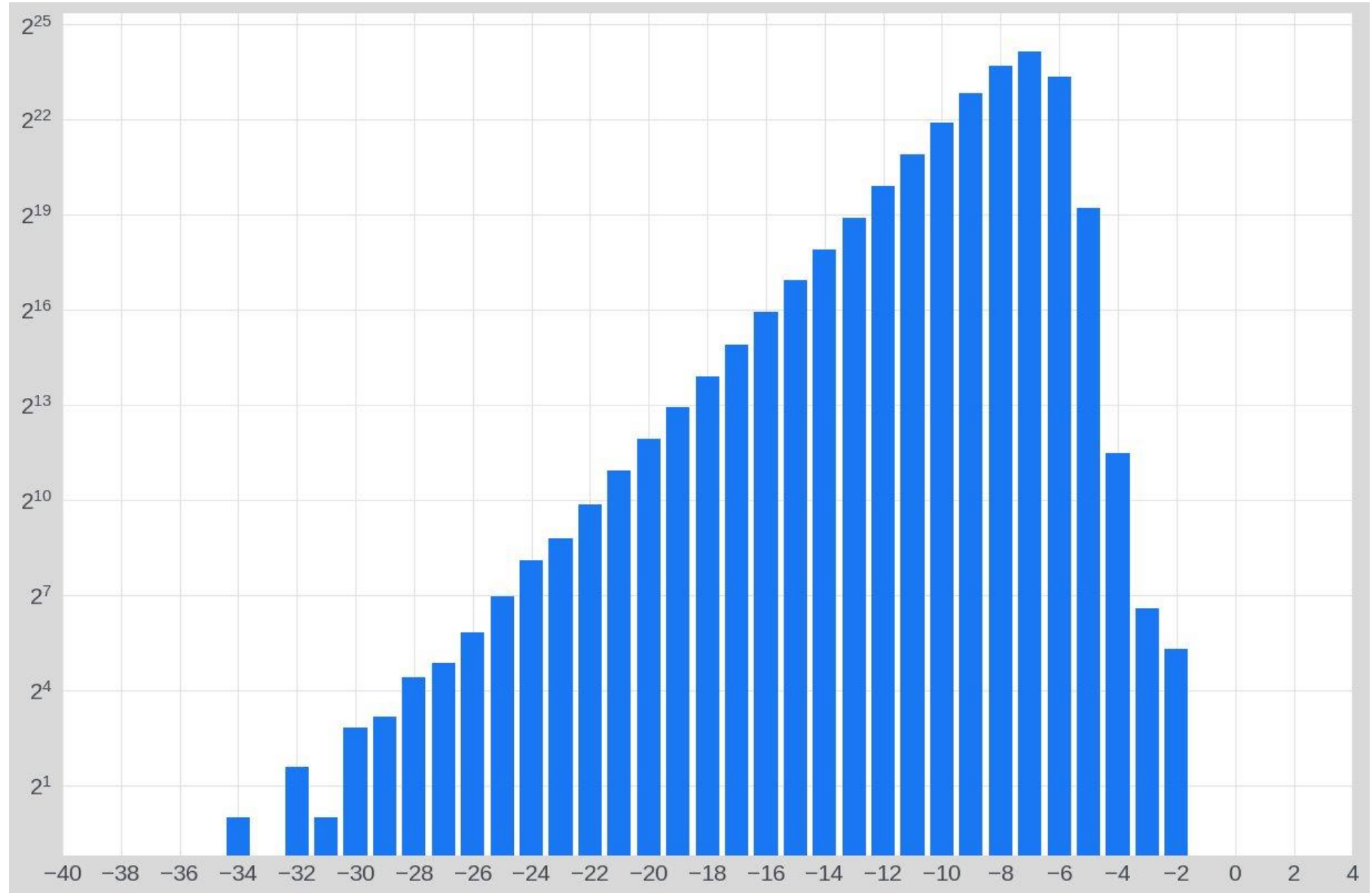
A Note on Fixed-Point Encoding

- The idea is that *binary* point is in a fixed position
 - Think of integers as having the binary point after the LSB
- For example, consider 4 bit **SI.FF** encoding
 - 1 bit sign (sign-magnitude encoding)
 - 1 integer bit
 - 2 fractional bits (i.e. 2 bits after the binary point)
 - Without loss of generality, consider just the positive values:
 - $0000_2 \rightarrow 0.00_{10}$
 - $0001_2 \rightarrow 0.25_{10}$
 - $0010_2 \rightarrow 0.50_{10}$
 - $0011_2 \rightarrow 0.75_{10}$
 - $0100_2 \rightarrow 1.00_{10}$
 - $0101_2 \rightarrow 1.25_{10}$
 - $0110_2 \rightarrow 1.50_{10}$
 - $0111_2 \rightarrow 1.75_{10}$
- Same as sign-magnitude integer for our purposes:
 - Note the same interval between each pair of successive samples (uniform sampling)
 - Sampling can be matched with integer and a scale factor (0.25 in the above example)

Exploring Weight Magnitudes

Weight Magnitude Histogram: Layer1.up_proj

- Dynamic range: ~32
- Note: graph is loglog



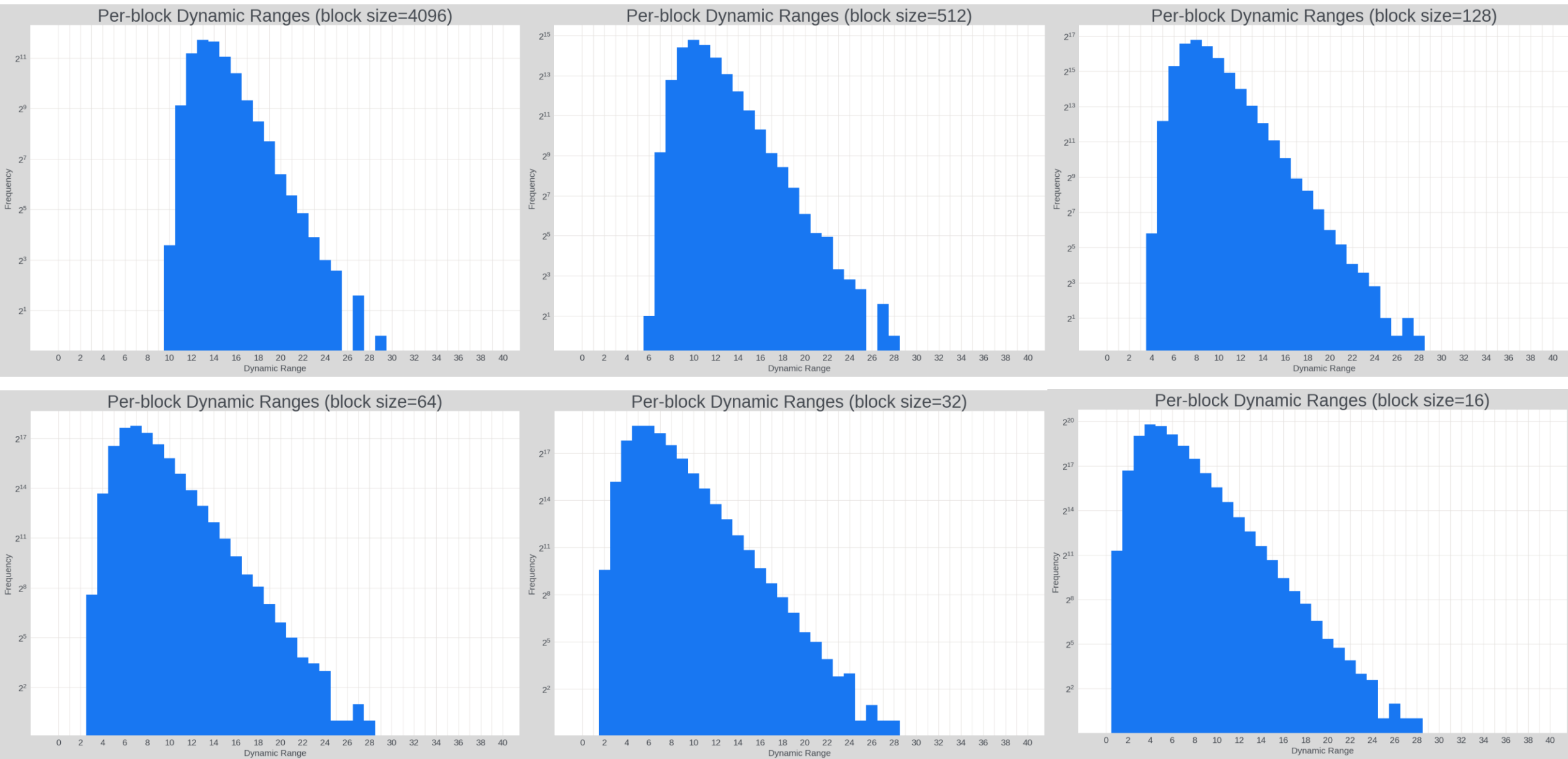
Per-Block Dynamic Range Histograms

- **What was done:**

- Partition the tensor into blocks of size B (1D blocks)
- Compute the dynamic range for each block
- Bin the dynamic ranges into a histogram

- **Observations:**

- Smaller blocks "shift" histogram "mass" to the left
 - There are more blocks as we reduce block size
 - As we subdivide a larger block, the resulting child blocks have the same or lower dynamic range than the parent block
- Number of blocks for the max dynamic ranges stays fairly constant
 - Suggests a few outlier large magnitudes



Note the axes are loglog. Note that y axis range increases as block size gets smaller (more and more total blocks)