Quantization

Aaron Mueller
CS599 B1: Advanced Natural Language Processing
Oct. 22, 2026

Administrivia

- Quiz 2 grades are out!
- I’m aiming to get your project proposals graded by the end of the week. Once those are out, you’ll have 1 week to revise and resubmit if you’re not happy with your grade.
- Reminder: I will be away next week (Oct. 26 - Oct. 30) for EMNLP
  - No office hours
  - Pre-recorded lecture for Tue.
  - No content/pre-recorded lecture for Thu. Enjoy the mini break!
  - Feel free to use Piazza for questions
  - I *may* be able to do ad hoc meetings if needed, but it will probably be at weird/early hours your time

Efficiency

- As researchers, engineers, and users, we would like to be able to use the best possible models despite not having OpenAI money.

- *How can we get the best model for our hardware?*

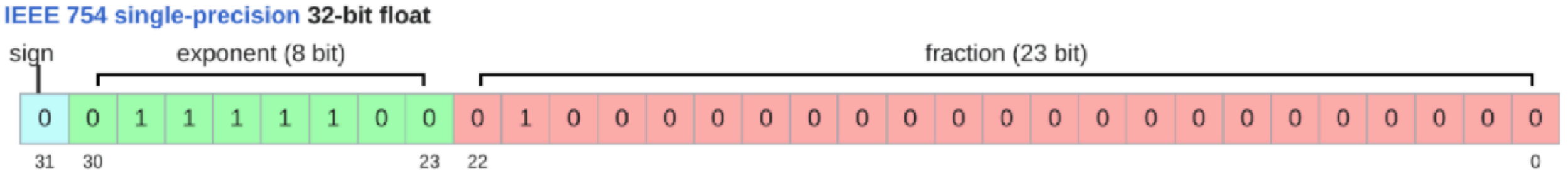
- We'll explore two strategies:
  - Tuesday: mixture-of-experts models
  - Today: quantization

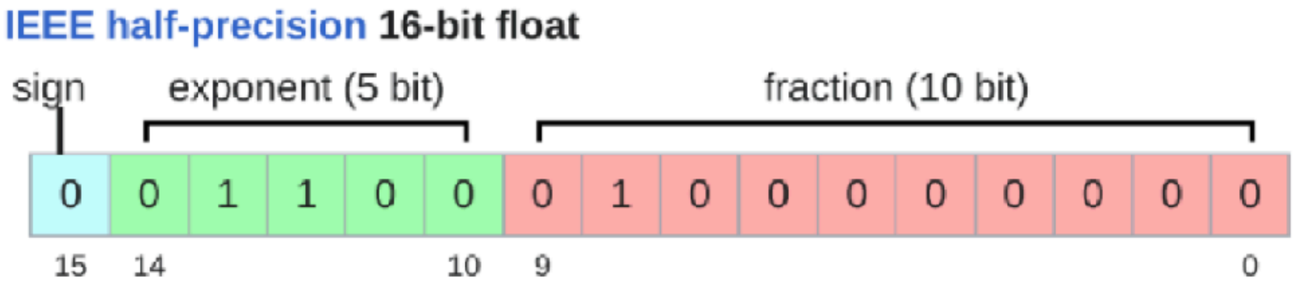
LLM Sizes

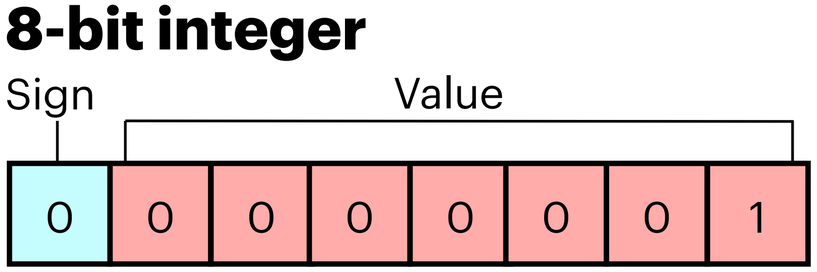
- Qwen3-8B:
  - 8 billion parameters * 2 bytes per parameter = 16 gigabytes
  - We need ≈1.5x the model weights to run inference (because of activations/ other buffer items), so it’s more like 24 GB

- Macbook Air: 16GB shared memory
  - Inference requires swapping to disk
- T4 GPU: 16GB memory
  - Inference leads to out-of-memory errors

Review: Precision and Data Types

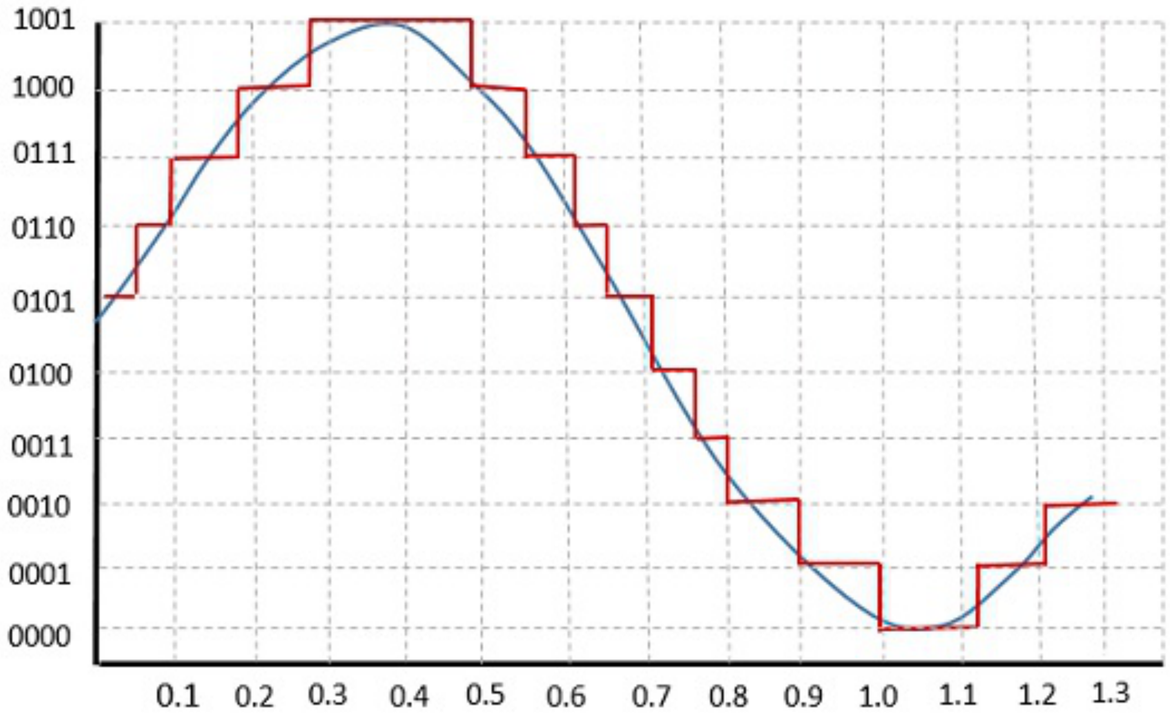






Quantization

- **Quantization** is the mapping of high-precision values to a smaller discrete set of lower-precision values



Quantization

Example

- Let's say we have a data type $l5$ with values [0, 1, 2, 3, 4, 5] and a data type $l3$ with values [0, 2, 4]. We want to quantize [3, 1, 2, 3] (in $l5$) into $l3$.

    1. Normalize range of $l5$ into $l3$

        a. Find the max value: max([3, 1,2 ,3]) = 3

        b. Divide the vector by that max: [3, 1, 2, 3] / 3 = [1, 0.33, 0.66, 1]

        c. Multiply by the range of the target data type:
        [1, 0.33, 0.66, 1] * 4 = [4, 1.33, 2.66, 4]

    2. Round to the nearest value in $l3$

        [4, 1.33, 2.66, 4] -> [4, 2, 2, 4]

*De*quantization

Example

- Let's say we have a data type $l5$ with values [0, 1, 2, 3, 4, 5] and a data type $l3$ with values [0, 2, 4]. We want to *de*quantize [4, 2, 2, 4] (a quantized [3, 1, 2, 3]) vector back into $l5$.

    1. Normalize range of $l3$ into $l5$

        a. Find the max value: max([4, 2, 2, 4]) = 4

        b. Divide the vector by that max: [4, 2, 2, 4] / 4 = [1, 0.5, 0.5, 1]

        c. Multiply by the max of the original vector:
        [1, 0.5, 0.5, 1] * 3 = [3, 1.5, 1.5, 3]

    2. Round to the nearest value in $l5$

        [3, 1.5, 1.5, 3] -> [3, 2, 2, 3] This produces one error.

Quantization

- To quantize a vector, we need low-bit **codes** plus metadata needed to interpret them, such as *scales* or a *codebook*
    - **Code:** an integer index that points to a specific entry in the codebook
    - **Codebook:** a lookup table (indexed by codes) containing a fixed set of representative values
    - **Scale:** a multiplier or scaling factor

8-bit Quantization

- For large transformer LLMs, the feed-forward and attention projection layers are 95% of the parameters and 65-85% of the computation
- **Goal:** quantize these parameters to use fewer bits
- Use low-bit precision matrix multiplication
- First, scale inputs into 8-bit codes [-127, 127] using the following:

$X_{f16} \in \mathbb{R}^{T \times h}$: fp16 input matrix

$X_{i8} \in \mathbb{Z}^{T \times h}$: int8 quantized matrix

$$X_{i8} = \text{round}\left(\frac{127 \cdot X_{f16}}{\max_{ij}(|X_{f16}|)}\right)$$

8-bit Quantization

- Matrix multiplication:

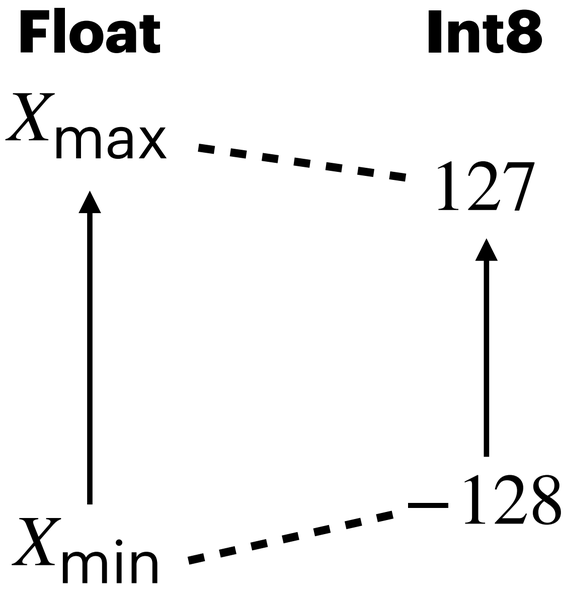
$$XW \approx \frac{1}{s_x s_w} Z_{i32} = \frac{1}{s_x s_w} Q(X) Q(W)$$

Scaling factors

- The $Q()$ notation refers to quantized (integer) matrices/vectors, and performing 8-bit matrix multiplication
- **Vector-wise quantization:** quantize each row of X and each column of W separately

8-bit Quantization with Zero Point

- int8 = round($s \cdot$ FP) + offset
  - $s = \frac{255}{\max - \min}$
  - offset = $-\text{round}\left(\frac{255}{\max - \min}\min\right) - 128$
  - FP = $\frac{\text{int8} - \text{offset}}{s}$



- Good for asymmetric data; in such cases, this approach would use all integers (whereas the previous approach would waste its negative integers)

[Welleck, 2026]

```python
import numpy as np
x = np.array([-3.0, 1.0, 2.0, 4.0])
x_min = np.min(x)
x_max = np.max(x)

qmin, qmax = -128, 127

if x_max - x_min != 0:
    scale = (qmax - qmin) / (x_max - x_min)
else:
    scale = 1.0

zero_point = np.round(qmin - x_min * scale)
zero_point = np.clip(zero_point, qmin, qmax).astype(np.int8)

x_quantized = np.round(x * scale + zero_point)
x_quantized = np.clip(x_quantized, qmin, qmax).astype(np.int8)

x_dequantized = (x_quantized.astype(np.float32) - zero_point) / scale

print("Original array:    ", x)
print("Quantized array:   ", x_quantized)
print("Dequantized array: ", x_dequantized)
print("Scale", scale)
print("Zero-point", zero_point)
mse = np.mean((x - x_dequantized) ** 2)
print("Mean Squared Error:", mse)
```

✓ 0.0s

```
Original array:     [-3.  1.  2.  4.]
Quantized array:    [-128   17   54  127]
Dequantized array:  [-2.99215686  0.98823529  2.00392157  4.00784314]
Scale 36.42857142857143
Zero-point -19
Mean Squared Error: 6.92041522491351e-05
```

Problems in Quantization

- An error is like a loss of information
- If we have many errors and propagate them through the network, they can easily accumulate, and may significantly degrade the quality of the system
- *How can we make quantization more precise?* We'll see two approaches:
  - Control for outliers: **int8**
  - Space values based on distributional properties: **nf4**

Information Loss

- As we've observed, the error can be nonzero even in very simple cases.

- Because quantization depends on normalizing based on the min and max, information loss can be quite extreme in the presence of **outliers**

Outlier Features

- Let's say we wanted to quantize x into a datatype with codes -4, -3, ..., 3, 4

$$x = [0.1,\ 0.2,\ 0.3,\ 10]$$

- Divide by max value to get $[0.1,0.2,0.3,10]/10 = [0.01,0.02,0.03,1]$
- Our codes will correspond to values [-1.0, -0.75, -0.5, ..., 0.5, 0.75, 1.0]
- Round to nearest code: [0, 0, 0, 4]
- The outlier is reconstructed perfectly, but all other values are collapsed to the same code
- If we had excluded the outlier, all values would have mapped to different codes

LLM.int8()

- **Dettmers et al. [2022]** observed that Transformer LMs have *outlier* features
- In models with $\geq$6.7B parameters, outliers occur in *all* layers
- **Hypothesis:** quantization results in large performance degradation due to outlier features

LLM.int8()

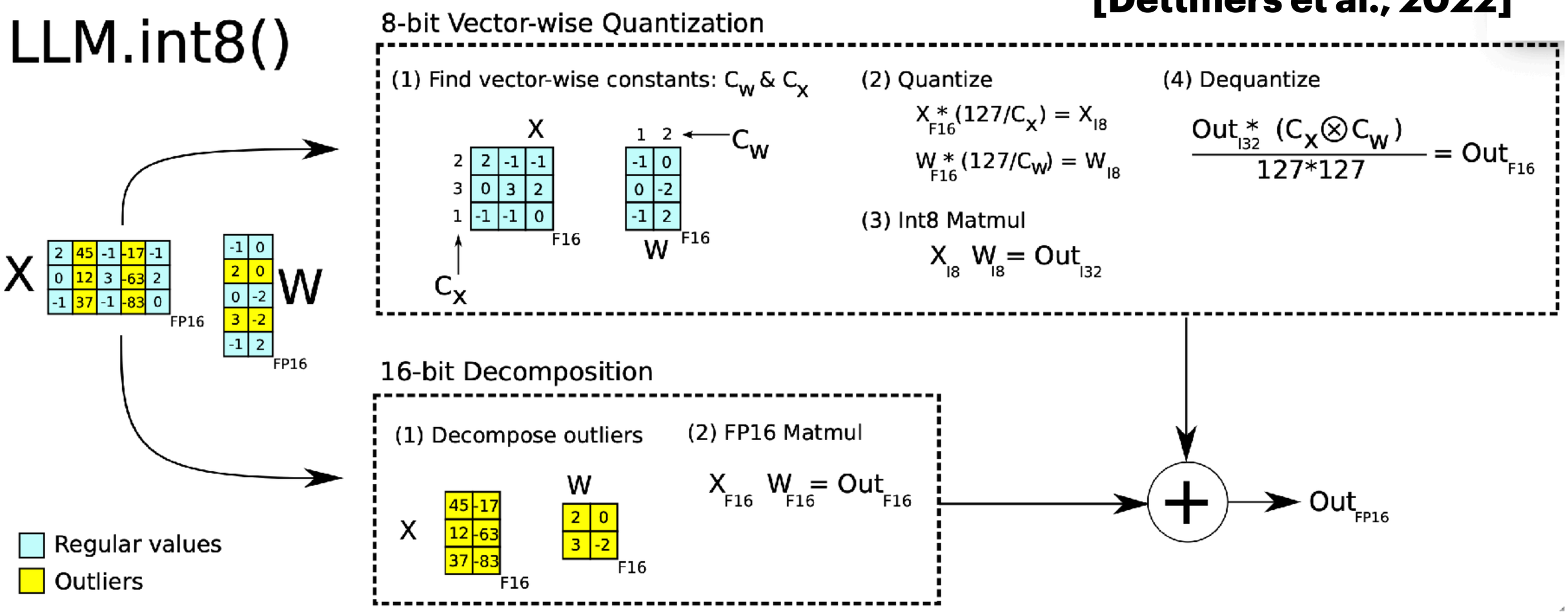
- **Idea:** perform computations that involve outliers in higher precision, perform other computations in lower precision, then merge the results

$$XW = X_{:,O}W_{O,:} + X_{:,R}W_{R,:}$$

- Where O indexes dimensions containing outliers, and R contains the rest of the dimensions
- $X_{:,O}W_{O,:}$happens in higher precision, and $X_{:,R}W_{R,:}$ happens in int8, followed by rescaling before being added with the first term
- Only ≈0.1% of activations are outliers, so empirically, this isn’t a bad trade-off

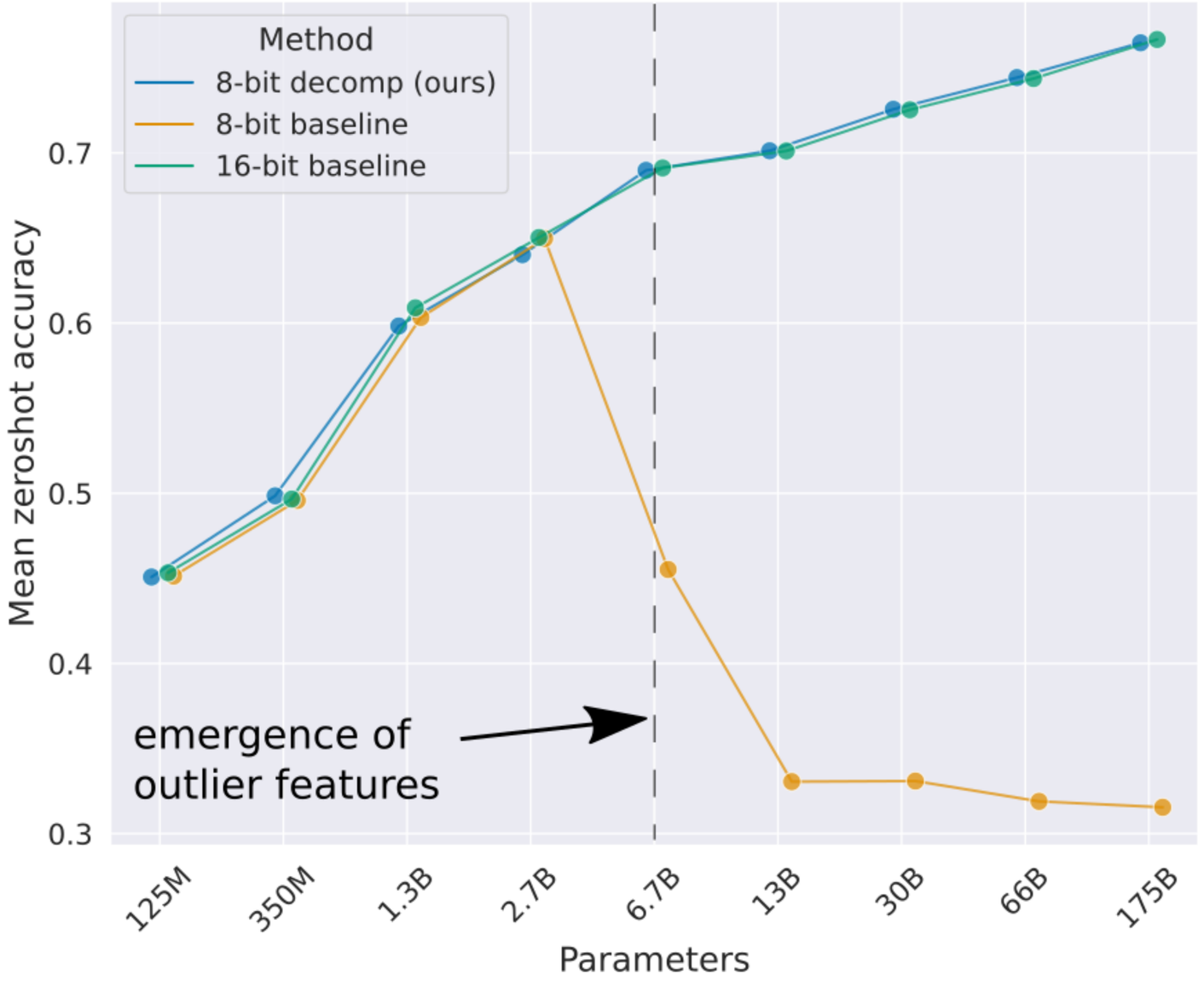
LLM.int8()

[Dettmers et al., 2022]



LLM.int8()

[Dettmers et al., 2022]



Tradeoffs

- Quantization reduces memory requirements for running a model
- Quantization usually results in some performance degradation

Parameters	125M	1.3B	2.7B	6.7B	13B
32-bit Float	25.65	15.91	14.43	13.30	12.45
Int8 absmax	87.76	16.55	15.11	14.59	19.08
Int8 zeropoint	56.66	16.24	14.76	13.49	13.94
Int8 absmax row-wise	30.93	17.08	15.24	14.13	16.49
Int8 absmax vector-wise	35.84	16.82	14.98	14.13	16.48
Int8 zeropoint vector-wise	25.72	15.94	14.36	13.38	13.47
Int8 absmax row-wise + decomposition	30.76	16.19	14.65	13.25	12.46
Absmax LLM.int8() (vector-wise + decomp)	25.83	15.93	14.44	**13.24**	**12.45**
Zeropoint LLM.int8() (vector-wise + decomp)	**25.69**	**15.92**	**14.43**	**13.24**	**12.45**

Validation perplexities

The NF4 Data Type

- **NormalFloat4 (NF4)**: a data type invented by **Dettmers et al. [2023]**
- With only 4 bits, we only have 16 possible values. How best to allocate these?
- Int8 uses *uniformly spaced* quantization levels.
- Nf4 uses *nonuniform spacing:* more precision near zero, less near the tails

Uniform vs. Nonuniform Spacing

- Under uniform spacing, a 4-bit data type lying in [-1, 1] would use 16 evenly spaced values between -1 and 1.
- Under nonuniform spacing, we allocate precision according to a normal distribution, such that we have more precision closer to 0 (the mean):

Uniform Spacing

```
[−1.0, −0.87, −0.73, −0.60, −0.47, −0.33, −0.20, −0.07, 0.07, …, 0.87, 1.0]
```

Codes	1	2	3	4	5	6	7	8	9	…	15	16
	[−1.0,	−0.70,	−0.53,	−0.40,	−0.28,	−0.19,	−0.09,	0.00,	0.08,	…,	0.72,	1.0]

Nonuniform Spacing

NF4

- The theoretically optimal data type for a normal distribution N(0, 1) in range [-1, 1] is computed as follows:
  - Estimate the $2^k + 1$ quantiles of the distribution
  - Normalize its values into [-1, 1]
  - Quantize an input weight tensor by normalizing it into [-1, 1]: divide element w_i by the (absolute) maximum value a in the tensor, then round to the integer index of the closest value

$$u_i = \frac{w_i}{a}$$

Weight

$$q_i = \arg\min_{j \in \{0,\ldots,15\}} |u_i - c_j|$$

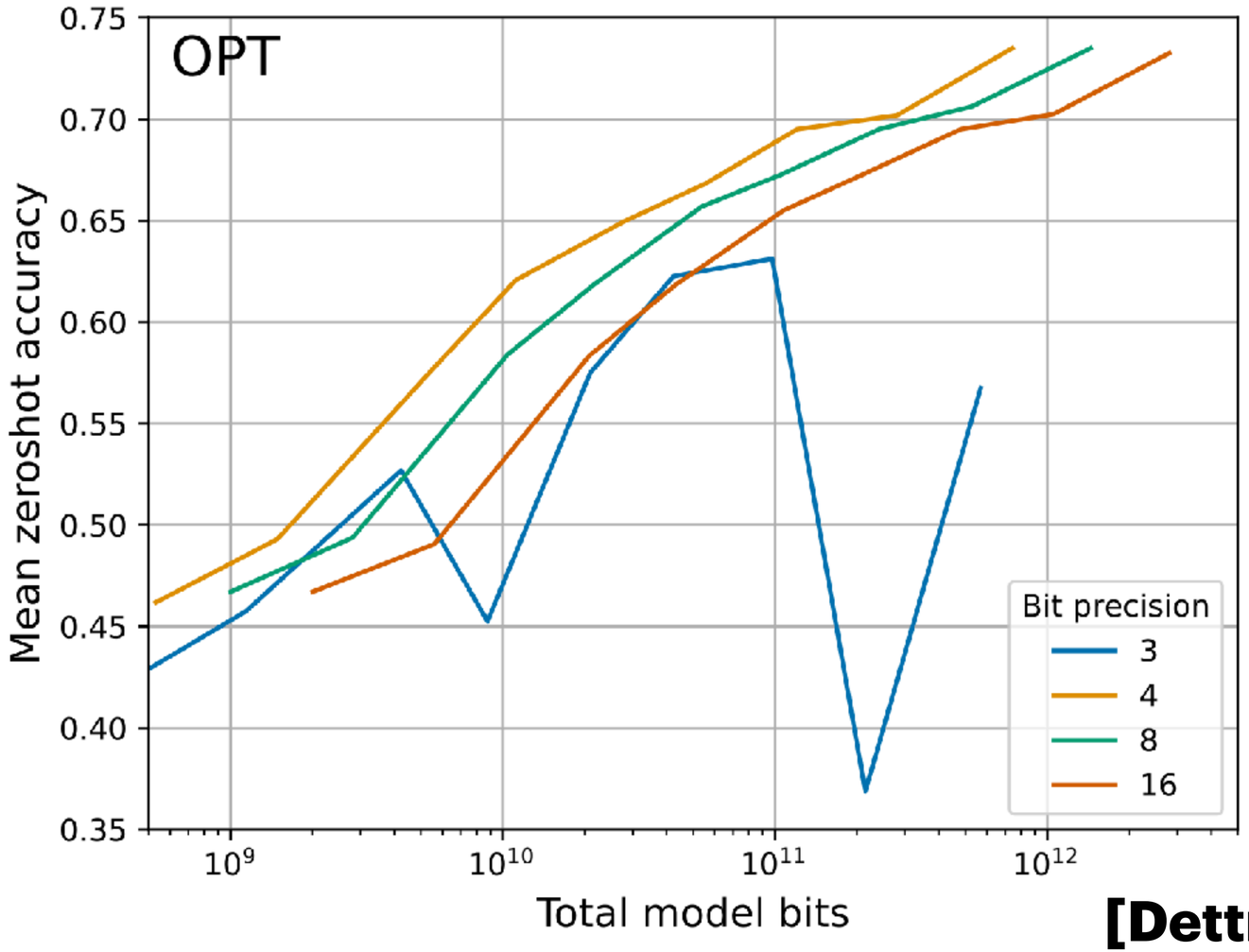
Quantized weight

Code

$$\hat{w}_i = a c_{q_i}$$

Dequantized weight

k-bit Scaling Laws



- Total bits ≈ num parameters * bits per parameter
- On this axis, a lower precision allows for more parameters
- We get a great trade-off at 4-bit precision
  - Any lower and things start to not work well

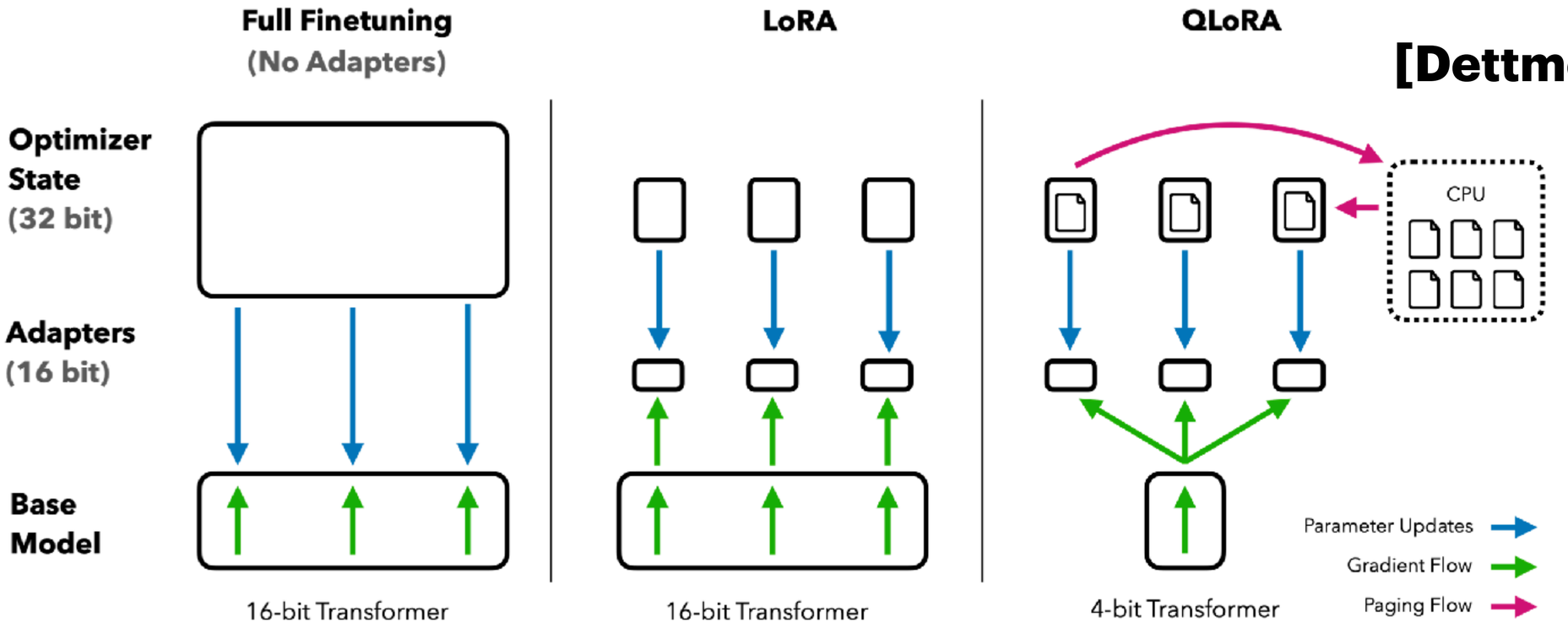
[Dettmers & Zettlemoyer, 2022]

Many other methods!

Quantization Method	On the fly quantization	CPU	CUDA GPU	ROCm GPU	Metal (Apple Silicon)	Intel GPU	Torch compile()	Bits	PEFT Fine Tuning	Serializable with 🤗 Transformers	🤗 Transform Suppor
AQLM	🔴	🟢	🟢	🔴	🔴	🟢	🟢	1/2	🟢	🟢	🟢
AutoRound	🔴	🟢	🟢	🔴	🔴	🟢	🔴	2/3/4/8	🔴	🟢	🟢
AWQ	🔴	🟢	🟢	🟢	🔴	🟢	?	4	🟢	🟢	🟢
bitsandbytes	🟢	🟢	🟢	🟡	🟡	🟢	🟢	4/8	🟢	🟢	🟢
compressed-tensors	🔴	🟢	🟢	🟢	🔴	🔴	🔴	1/8	🟢	🟢	🟢
EETQ	🟢	🔴	🟢	🔴	🔴	🔴	?	8	🟢	🟢	🟢
FP-Quant	🟢	🔴	🟢	🔴	🔴	🔴	🟢	4	🔴	🟢	🟢
GGUF / GGML (llama.cpp)	🟢	🟢	🟢	🔴	🟢	🟢	🔴	1/8	🔴	See Notes	See Notes
GPTQModel	🔴	🟢	🟢	🟢	🟢	🟢	🔴	2/3/4/8	🟢	🟢	🟢
AutoGPTQ	🔴	🔴	🟢	🟢	🔴	🔴	🔴	2/3/4/8	🟢	🟢	🟢
HIGGS	🟢	🔴	🟢	🔴	🔴	🔴	🟢	2/4	🔴	🟢	🟢
HQQ	🟢	🟢	🟢	🔴	🔴	🟢	🟢	1/8	🟢	🔴	🟢
optimum-quanto	🟢	🟢	🟢	🔴	🟢	🟢	🟢	2/4/8	🔴	🔴	🟢

QLoRA

- Recall fine-tuning with **LoRA**
- We can directly fine-tune a quantized model with **QLoRA**!



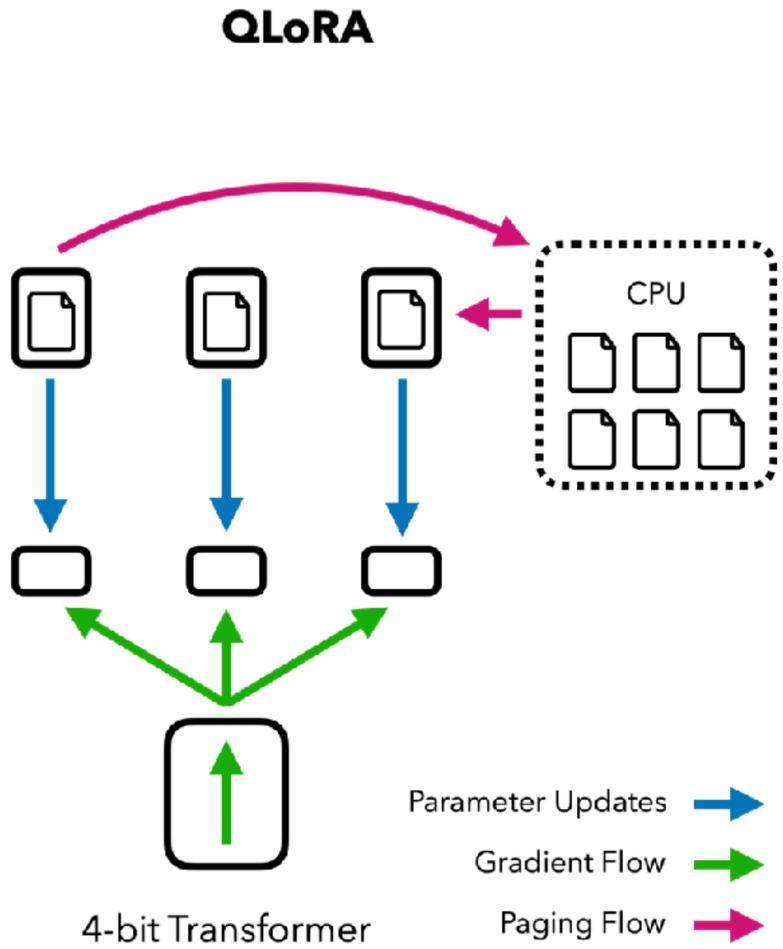
[Dettmers et al., 2023]

Uses nf4 and memory offloading techniques to reduce memory requirements of fine-tuning LLMs

QLoRA

$$y = \text{dequant}(W_q)x + \frac{\alpha}{r}BAx$$

- W_q: frozen quantized base weights
- A, B: trainable low-rank adapter matrices
- As in LoRA, the optimizer does not update W_q

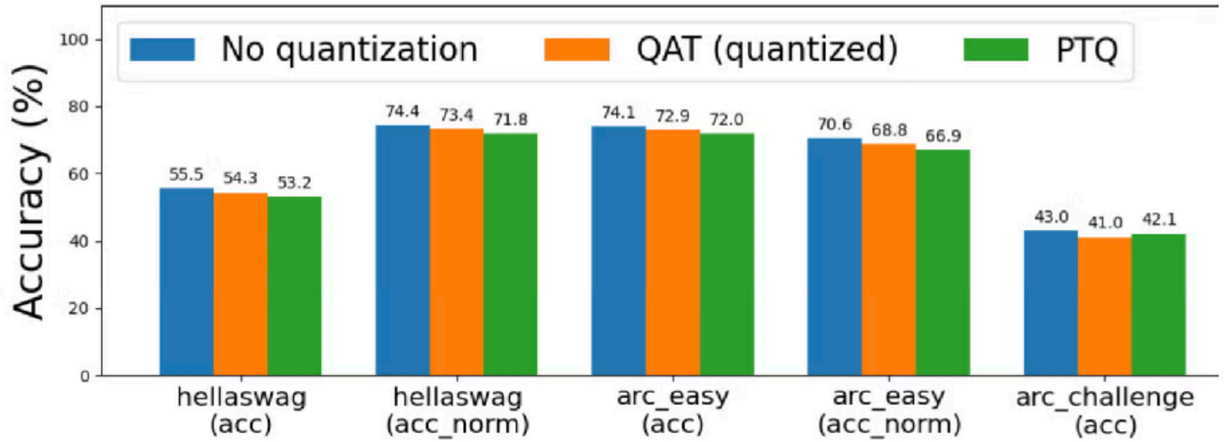


[Dettmers et al., 2023]

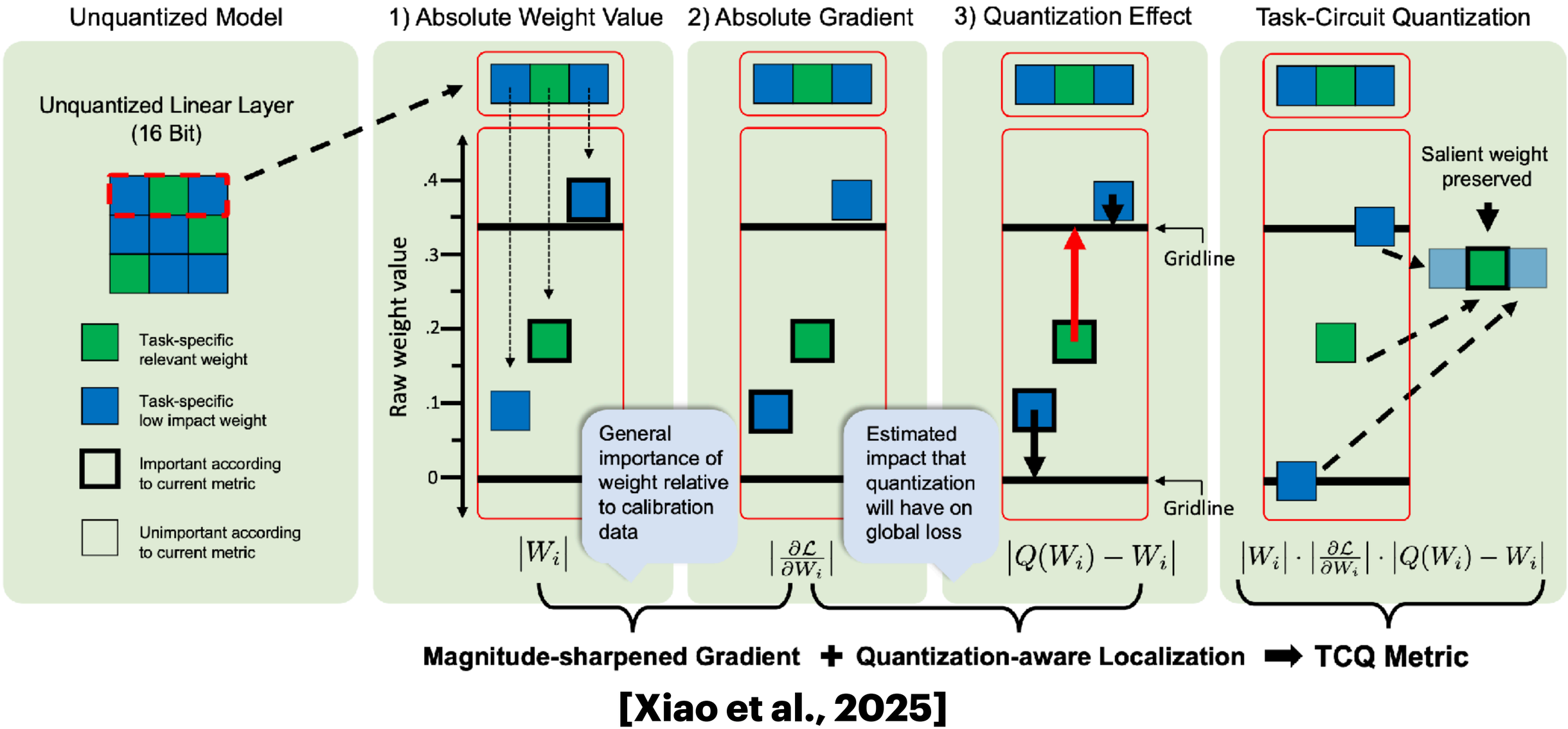
Quantization-aware Training (QAT)

- **Idea:** fine-tune a model so that it is “prepared” for quantization at inference time
- To do this, we can insert artificial quantization operations into the computation graph. Add a quantize-dequantize operation into the model layers

```
# QAT: x_fq is still in float
# Fake quantize simulates the numerics of quantize + dequantize
x_fq = (x_float / scale + zp).round().clamp(qmin, qmax)
x_fq = (x_fq - zp) * scale
```

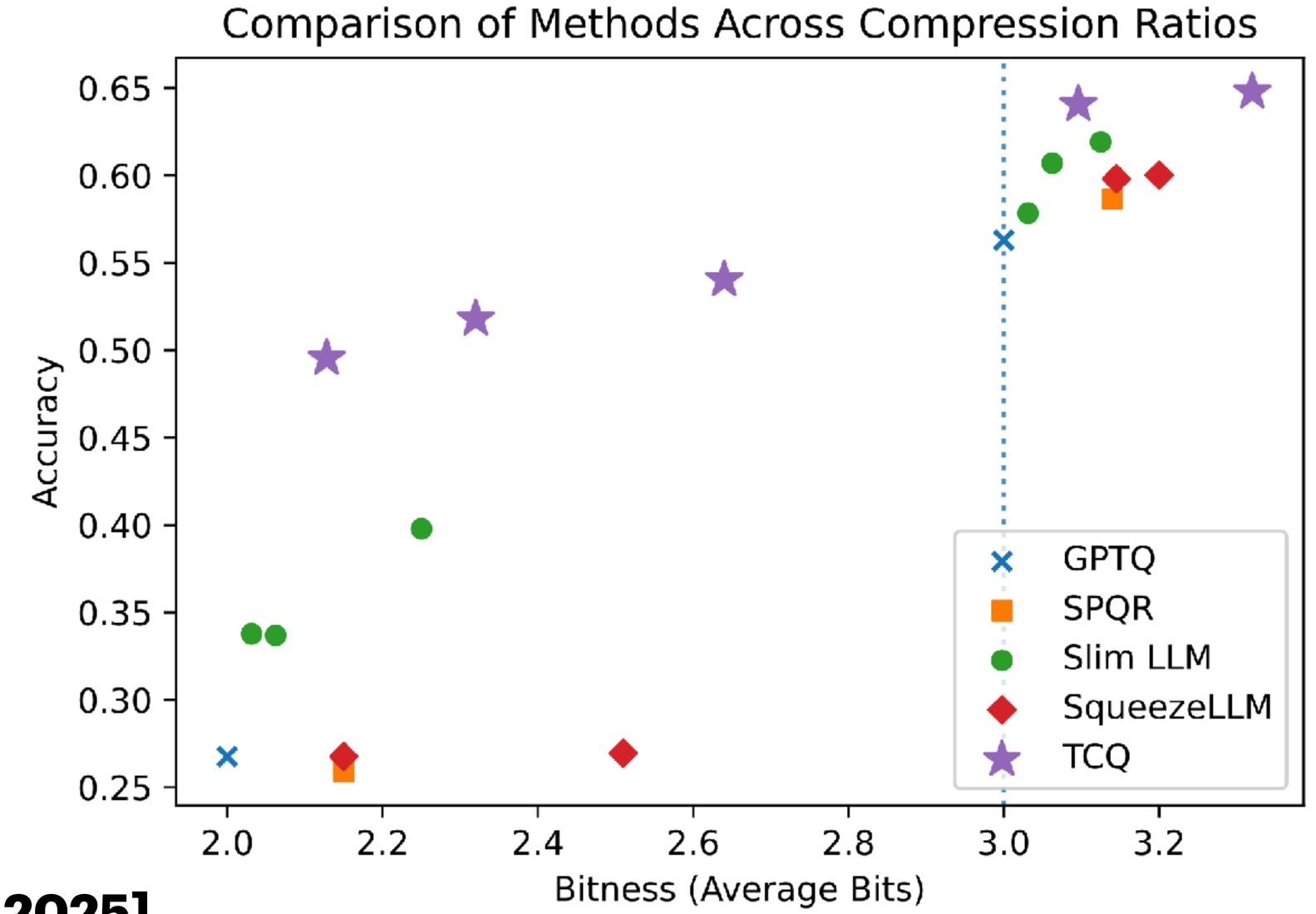


Task-circuit Quantization (TaCQ)



[Xiao et al., 2025]

Task-circuit Quantization (TaCQ)



[Xiao et al., 2025]