8-bit Methods for Efficient Deep Learning

Tim Dettmers

Large models are not easily accessible

Model	Inference memory	Fine-tuning memory
T5-11B	22 GB	176 GB
OPT-66B	132 GB	1,056 GB
BLOOM 176B	350 GB	2,800 GB

Raffel et al., 2020, T5. Zhang et al., 2022, OPT., BigScience, 2022, BLOOM.

Large models are not easily accessible

Model	Inference memory	Fine-tuning memory
T5-11B	22 GB	176 GB
OPT-66B	132 GB	1,056 GB
BLOOM 176B	352 GB	2,800 GB

LLM.int8()　　8-bit optimizers

Model	Inference memory	Fine-tuning memory
T5-11B	11 GB	66 GB
OPT-66B	66 GB	396 GB
BLOOM 176B	176 GB	1,056 GB

Raffel et al., 2020, T5. Zhang et al., 2022, OPT., BigScience, 2022, BLOOM.

Overview of my work in this talk

8-Bit Approximations for Parallelism in Deep Learning. **Tim Dettmers**, *ICLR* 2015.

8-bit Optimizers via Block-wise Quantization. **Tim Dettmers**, Mike Lewis, Sam Shleifer, Luke Zettlemoyer, *ICLR* 2022 ***Spotlight***.

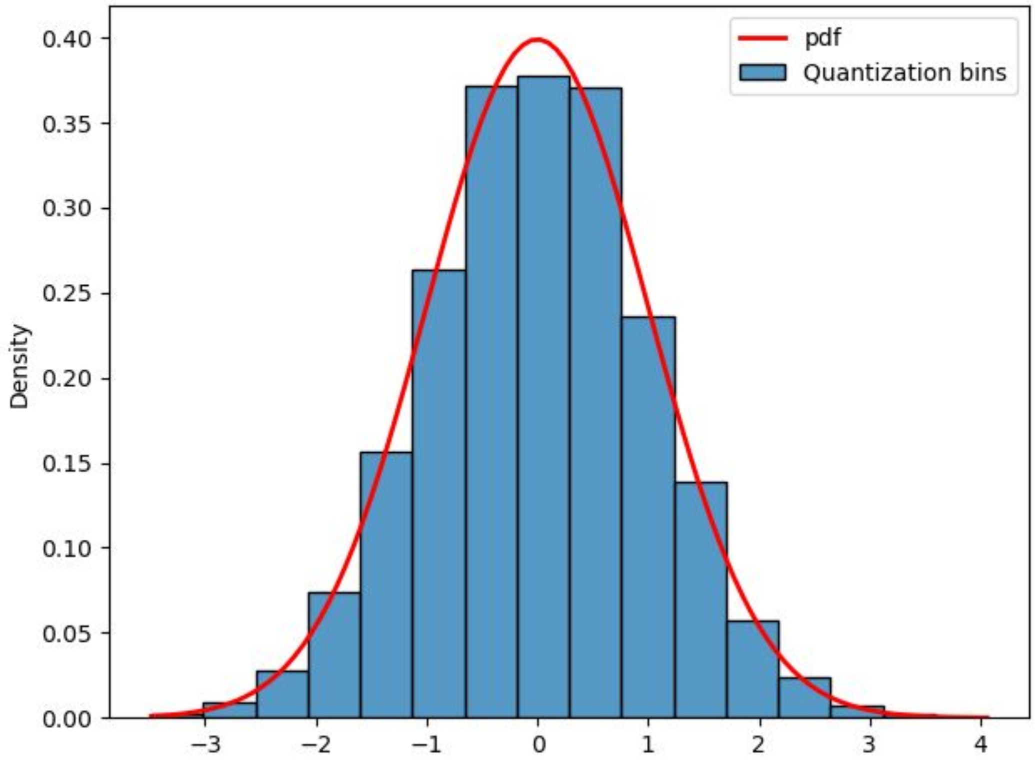
LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale . **Tim Dettmers**, Mike Lewis, Younes Belkada, Luke Zettlemoyer, *NeurIPS* 2022.

The case for 4-bit precision: k-bit Inference Scaling Laws. **Tim Dettmers**, Luke Zettlemoyer, i*n submission*.

Personalize your own ChatGPT on a single GPU. **Tim Dettmers**, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer, *in progress*.

Background

How does quantization work?



Quantization as a mapping

Most general form of describe quantization is through a mapping from integers to float values normalized to the range -1.0 and 1.0.

Int4	0 1 2 3 4 …	-7 -6 -5 -4 -3 …
FP4	0 1 2 3 4 5 …	-12 -8 -6 -4 -3 …

Quantization as a mapping

Most general form of describe quantization is through a mapping from integers to float values normalized to the range -1.0 and 1.0.

Int4	0	1	2	3	4 …
	-7	-6	-5	-4	-3 …
	-1	-0.86	-0.71	-0.57	-0.43 …

FP4	0	1	2	3	4 …
	-12	-8	-6	-4	-3 …
	-1	-0.67	-0.5	-0.33	-0.25 …

The mapping format { index : float value} generalizes to all data types.

Quantization as a mapping

Most general form of describe quantization is through a mapping from integers to float values normalized to the range -1.0 and 1.0.

Int4 maps -7, -6, … 6, 7 -> -1.0, -0.86 … 0.86, 1.0

Given a tensor X of any real data type. We can apply 8-bit quantization as follows:

1. Normalize X into the range [-1.0, 1.0]
2. Find the closest value in the data type

Step (1) is usually done by dividing by the absolute maximum (absmax) value.

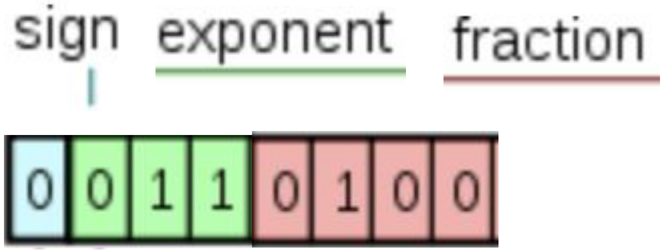
Quantization Example: A non-standard 2-bit data type

Map: {Index: 0, 1, 2, 3 -> Values: -1.0, 0.3, 0.5, 1.0}

Input tensor: [10, -3, 5, 4]

1. Normalize with absmax: [10, -3, 5, 4] -> [1, -0.3, 0.5, 0.4]
2. Find closest value: [1, -0.3, 0.5, 0.4] -> [1.0, 0.3, 0.5, 0.5]
3. Find the associated index: [1.0, 0.3, 0.5, 0.5] -> [3, 1, 2, 2] -> store
4. Dequantization: load -> [3, 1, 2, 2] -> lookup -> [1.0, 0.3, 0.5, 0.5] -> denormalize -> [10, 3, 5, 5]

Floating point data types (FP8)



3 bits for exponent, 4 for fraction:

- Good for large/small numbers
- Bad for precise numbers

1 bits for exponent, 6 for fraction:

- Good for precise numbers
- Bad for large/small numbers

Dynamic exponent quantization

Exponent: 1e-2

Fraction: 0.1 + 0.9*9/16 = 0.606

Sign "-"

Indicator bit

1	0	0	1	1	0	0	1

- 1e-2 * 0.606 = -6.06e-3

Dynamic exponent and fraction bits:

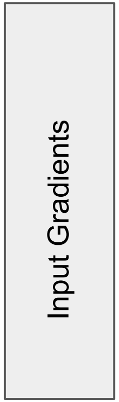
- Good for small and large numbers
- High precision for small and intermediate numbers
- Bad precision for very large numbers

8-bit Approximations for Parallelism in Deep Learning. Dettmers, 2015.

8-bit Optimizers

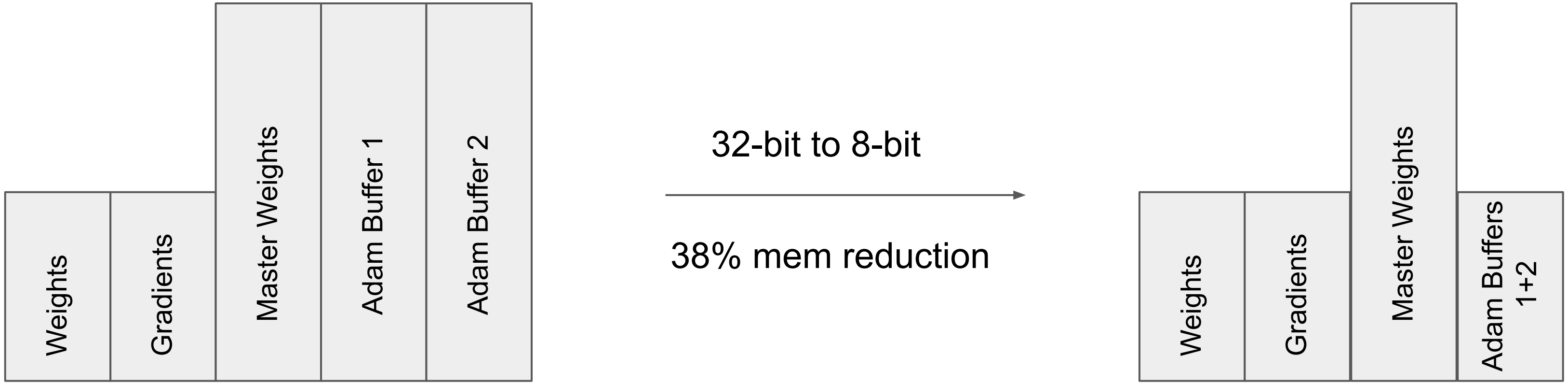
Motivation: Optimizers take up a lot of memory!

Memory depends on seq len, batch size, and model size

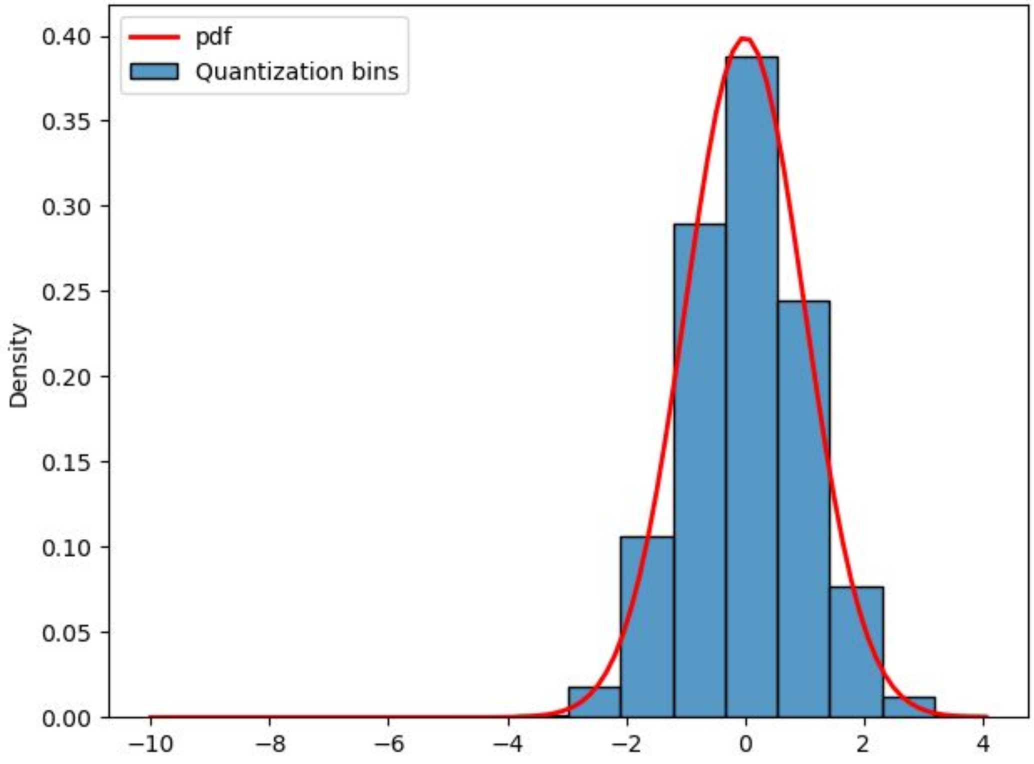


Memory that only depends on model size

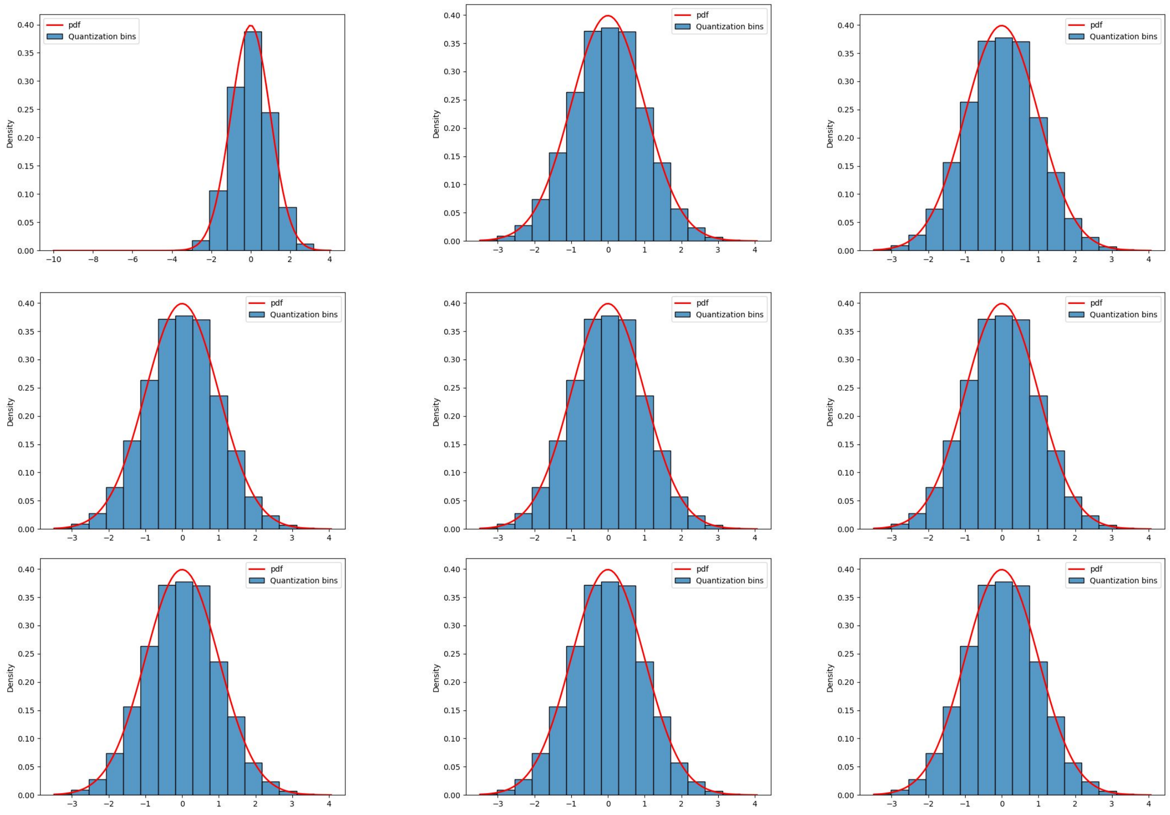
8-bit optimizers reduce memory consumption by 40%



What do outliers in quantization look like?



Block-wise quantization



Putting it together: 8-bit optimizers

Quantization

Updated optimizer states

Optimizer State	-3.1	0.1	-0.03	1.2
Chunk into blocks	$\boxed{-3.1 \quad 0.1}$		$\boxed{-0.03 \quad 1.2}$	
Find block-wise absmax	3.1		1.2	
Normalize with absmax	-1.0	0.032	-0.025	1.0
Find closest 8-bit value	-1.0	0.0329	-0.0242	1.0
Find corresponding index	0	170	80	255

Store index values

Dequantization

Load Index values

Index	0	170	80	255
Lookup values	-1.0	0.0329	-0.0242	1.0
Denormalize by absmax	-1.0*3.1	0.0329*3.1	-0.0242*1.2	1.0*1.2
Dequantized optimizer states	-3.1	0.102	-0.029	1.2

Update optimizer states

Results: Same accuracy/perplexity as 32-bit!

Optimizer	Task	Data	Model	Metric†	Time	Mem saved
32-bit AdamW	GLUE	Multiple	RoBERTa-Large	88.9	–	Reference
32-bit AdamW	GLUE	Multiple	RoBERTa-Large	88.6	17h	0.0 GB
32-bit Adafactor	GLUE	Multiple	RoBERTa-Large	**88.7**	24h	1.3 GB
8-bit AdamW	GLUE	Multiple	RoBERTa-Large	**88.7**	**15h**	**2.0 GB**
32-bit Momentum	CLS	ImageNet-1k	ResNet-50	77.1	–	Reference
32-bit Momentum	CLS	ImageNet-1k	ResNet-50	77.1	118h	0.0 GB
8-bit Momentum	CLS	ImageNet-1k	ResNet-50	**77.2**	**116 h**	**0.1 GB**
32-bit Adam	MT	WMT'14+16	Transformer	29.3	–	Reference
32-bit Adam	MT	WMT'14+16	Transformer	29.0	126h	0.0 GB
32-bit Adafactor	MT	WMT'14+16	Transformer	29.0	127h	0.3 GB
8-bit Adam	MT	WMT'14+16	Transformer	**29.1**	**115h**	**1.1 GB**
32-bit Momentum	MoCo v2	ImageNet-1k	ResNet-50	67.5	–	Reference
32-bit Momentum	MoCo v2	ImageNet-1k	ResNet-50	67.3	30 days	0.0 GB
8-bit Momentum	MoCo v2	ImageNet-1k	ResNet-50	**67.4**	**28 days**	**0.1 GB**
32-bit Adam	LM	Multiple	Transformer-1.5B	9.0	308 days	0.0 GB
32-bit Adafactor	LM	Multiple	Transformer-1.5B	**8.9**	316 days	5.6 GB
8-bit Adam	LM	Multiple	Transformer-1.5B	9.0	**297 days**	**8.5 GB**
32-bit Adam	LM	Multiple	GPT3-Medium	10.62	795 days	0.0 GB
32-bit Adafactor	LM	Multiple	GPT3-Medium	10.68	816 days	1.5 GB
8-bit Adam	LM	Multiple	GPT3-Medium	**10.62**	**761 days**	**1.7 GB**
32-bit Adam	Masked-LM	Multiple	RoBERTa-Base	3.49	101 days	0.0 GB
32-bit Adafactor	Masked-LM	Multiple	RoBERTa-Base	3.59	112 days	0.7 GB
8-bit Adam	Masked-LM	Multiple	RoBERTa-Base	**3.48**	**94 days**	**1.1 GB**

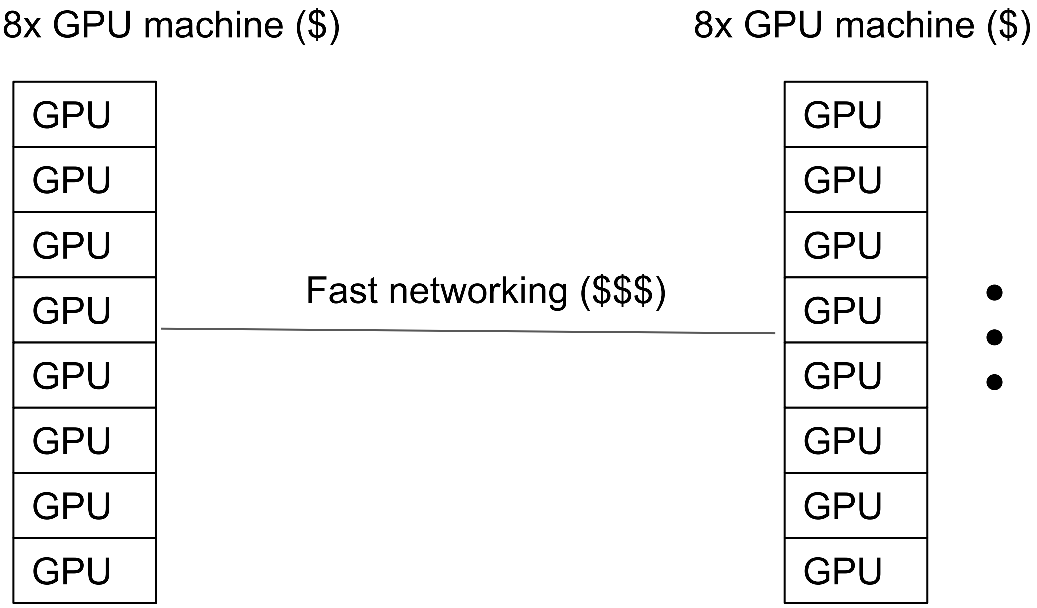
†**Metric**: GLUE=Mean Accuracy/Correlation. CLS/MoCo = Accuracy. MT=BLEU. LM=Perplexity.

Ablations on Language Modeling: All components needed!

Parameters	Optimizer	Dynamic	Block-wise	Stable Emb	Unstable (%)	Perplexity
209M	32-bit Adam				0	16.7
	32-bit Adam			✓	0	**16.3**
	8-bit Adam				90	253.0
	8-bit Adam			✓	50	**194.4**
	8-bit Adam	✓			10	18.6
	8-bit Adam	✓		✓	0	**17.7**
	8-bit Adam	✓	✓		0	16.8
	8-bit Adam	✓	✓	✓	0	**16.4**
1.3B	32-bit Adam				0	10.4
1.3B	8-bit Adam	✓			100	N/A
1.3B	8-bit Adam	✓		✓	80	10.9
1.5B	32-bit Adam				0	9.0
1.5B	8-bit Adam	✓	✓	✓	0	9.0

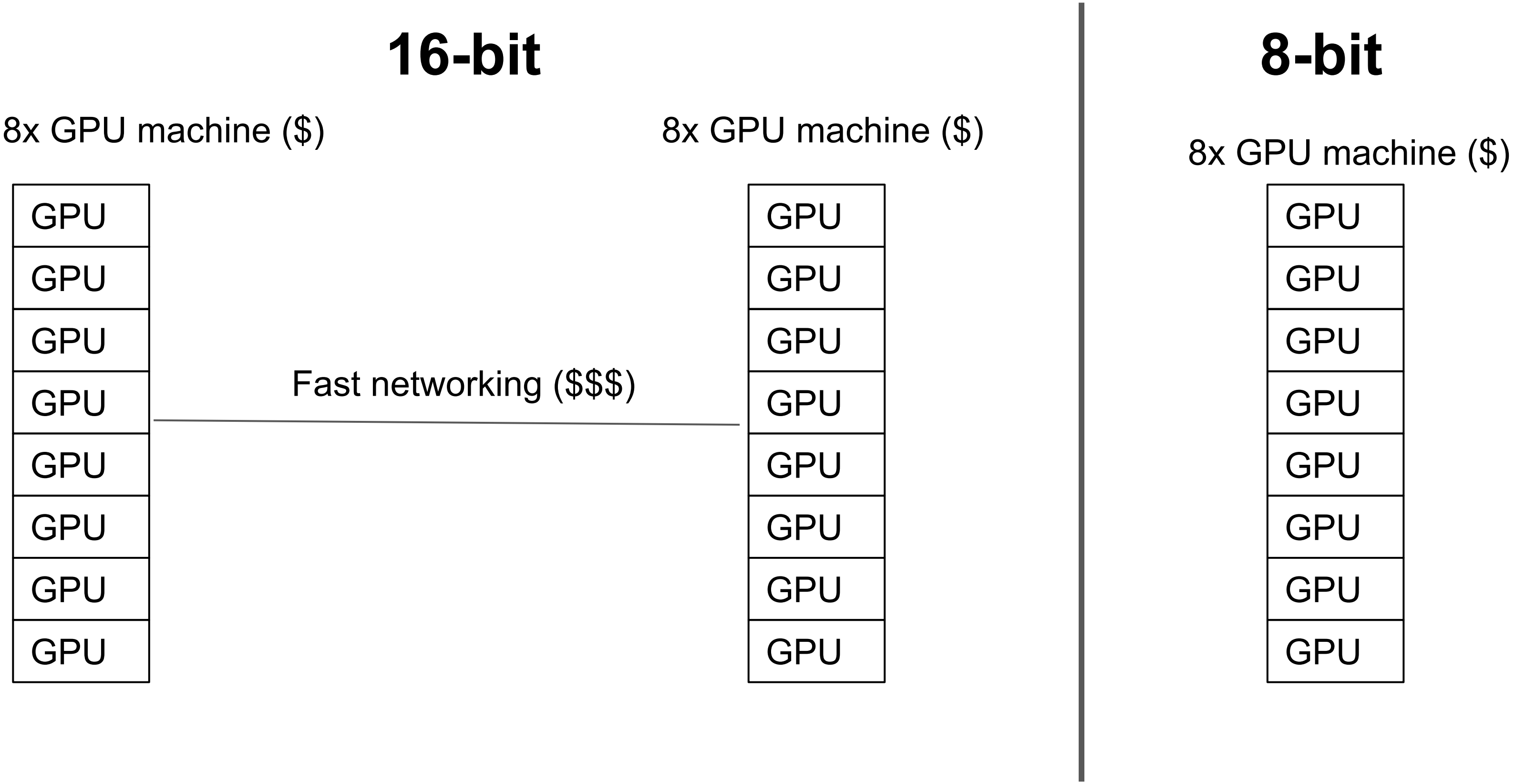
LLM.int8()

Large models such as OPT-175B need more than one computer to be run



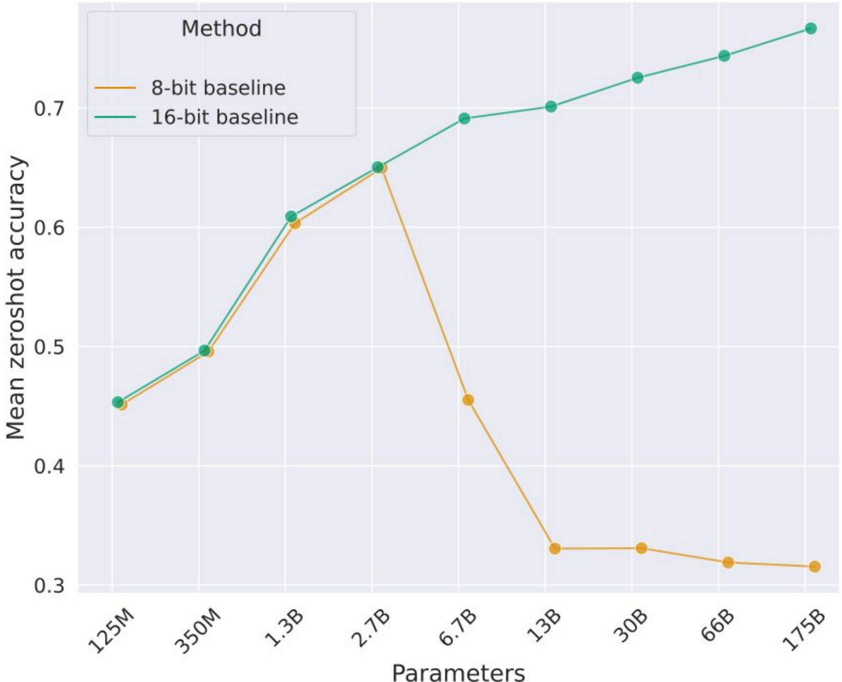
- OPT-175B requires 350 GB of GPU memory
- 15 consumer GPUs required for OPT-175B
- 7 high-end GPUs required ($15k per GPU)

With 8-bit weights we only need a single machine with consumer GPUs



Using OPT-175B on a single machine via 8-bit weights

With 8-bit weights we can reduce memory usage from 350 GiB to 175 GiB. This fits into a single machine with 8 consumer GPUs! Does it work?



The problem with quantizing outliers with large values

Absmax linear quantization with very large outliers:

3.50, 0.10, 0.02, 1.00, 0.30, 0.01, 0.05, 0.10 -> 127, 5, 1, 50, 15, 0, 2, 5

7.50, 0.10, 0.02, 1.00, 0.30, 0.01, 0.05, 0.10 -> 127, 2, 0, 25, 6, 0, 1, 2

15.0, 0.10, 0.02, 1.00, 0.30, 0.01, 0.05, 0.10 -> 127, 0, 0, 10, 3, 0, 0, 1

Vector-wise quantization

Matrix multiplication is a series of inner products

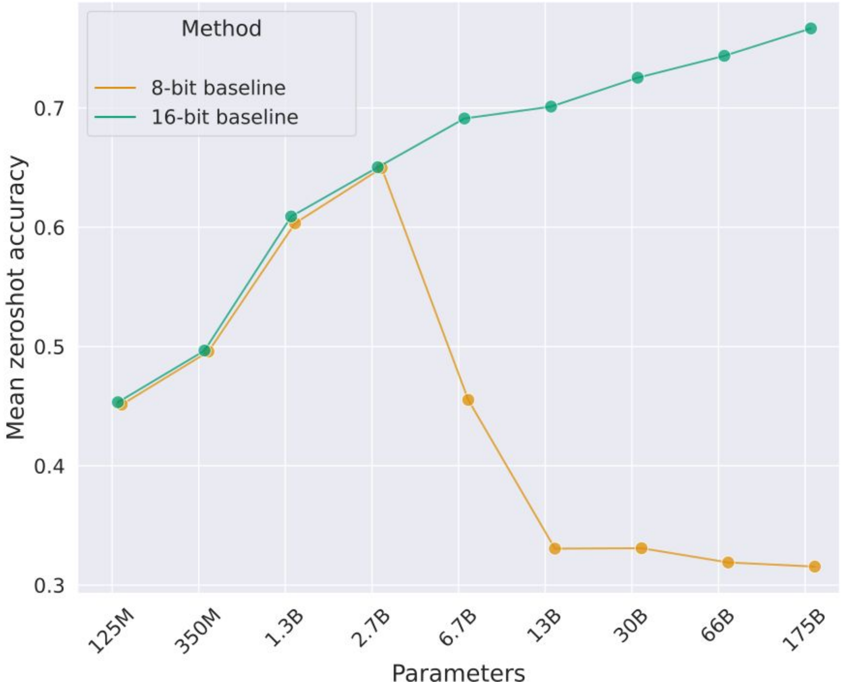
Use two unique normalization constants for each inner product.

A @ B = C: [b x h] @ [h x o] -> [b x o]

1. Normalize A and B:
    a. absmaxA_vec = A16.absmax(1): [b x h] -> [b]
    b. absmaxB_vec = B16.absmax(0): [h x o] -> [o]
    c. A8 = 127*A16/absmaxA_vec; B8 = 127*B16/absmaxB_vec: [b x h] * [b] -> [b x h]
2. C32 = A8 @ B8
3. C16f = C32/(127*127) * (absmaxA_vec @ absmaxB_vec): [b x o] + ([b] @ [o] -> [b x o])

Using OPT-175B on a single machine via 8-bit weights

With 8-bit weights we can reduce memory usage from 350 GiB to 175 GiB. This fits into a single machine with 8 consumer GPUs! Does it work?



Emergent Features

Finding outlier features in transformer hidden states

Hidden states with dimension [sequence, hidden_dim], outliers are in some hidden dimension.

You need to look in the right spots. At 2.7B there are 262,144 different hidden state values and only 960 are outliers (0.3%). Its easy to miss!

Fortunately, it becomes more common and highly systematic with scale.

Hidden states in transformers: 125m

98.5% of the time

[0.3, -0.1, 0.4]

[-0.2, 0.5, 0.1]

[0.3, 0.9, -0.7]

Sequence dimension

Hidden/feature dimension

1.5% of the time

[0.3, -0.1, -3.0]

[-0.2, 0.5, -6.0]

[0.3, 0.9, -7.0]

Hidden states in transformers: 350m

95% of the time

[0.3, -0.1, 0.4]

[-0.2, 0.5, 0.1]

[0.3, 0.9, -0.7]

5% of the time

[0.3, -0.1, 5.0]

[-0.2, 0.5, 6.0]

[0.3, 0.9, 8.0]

Hidden states in transformers: 2.7B

91% of the time

[0.3, -0.1, 0.4]

[-0.2, 0.5, 0.1]

[0.3, 0.9, -0.7]

9% of the time

[0.3, -0.1, -16.0]

[-0.2, 0.5, -10.0]

[0.3, 0.9, -27.0]

Hidden states in transformers: 6.0B

83% of the time

[0.3, -0.1, 0.4]

[-0.2, 0.5, 0.1]

[0.3, 0.9, -0.7]

17% of the time

[0.3, -0.1, -15.0]

[-0.2, 0.5, -17.0]

[0.3, 0.9, -22.0]

Hidden states in transformers: 6.7B. Phase shift!

25% of the time

[0.3, -0.1, 0.4]

[-0.2, 0.5, 0.1]

[0.3, 0.9, -0.7]

75% of the time

[0.3, -0.1, -40.0]

[-0.2, 0.5, -45.0]

[0.3, 0.9, -61.0]

Hidden states in transformers: 13B

~25% of the time

[0.3, -0.1, 0.4]

[-0.2, 0.5, 0.1]

[0.3, 0.9, -0.7]

~75% of the time

[0.3, -0.1, -75.0]

[-0.2, 0.5, -65.0]

[0.3, 0.9, -50.0]

Hidden states in transformers: 66B

~25% of the time

[0.3, -0.1, 0.4]

[-0.2, 0.5, 0.1]

[0.3, 0.9, -0.7]

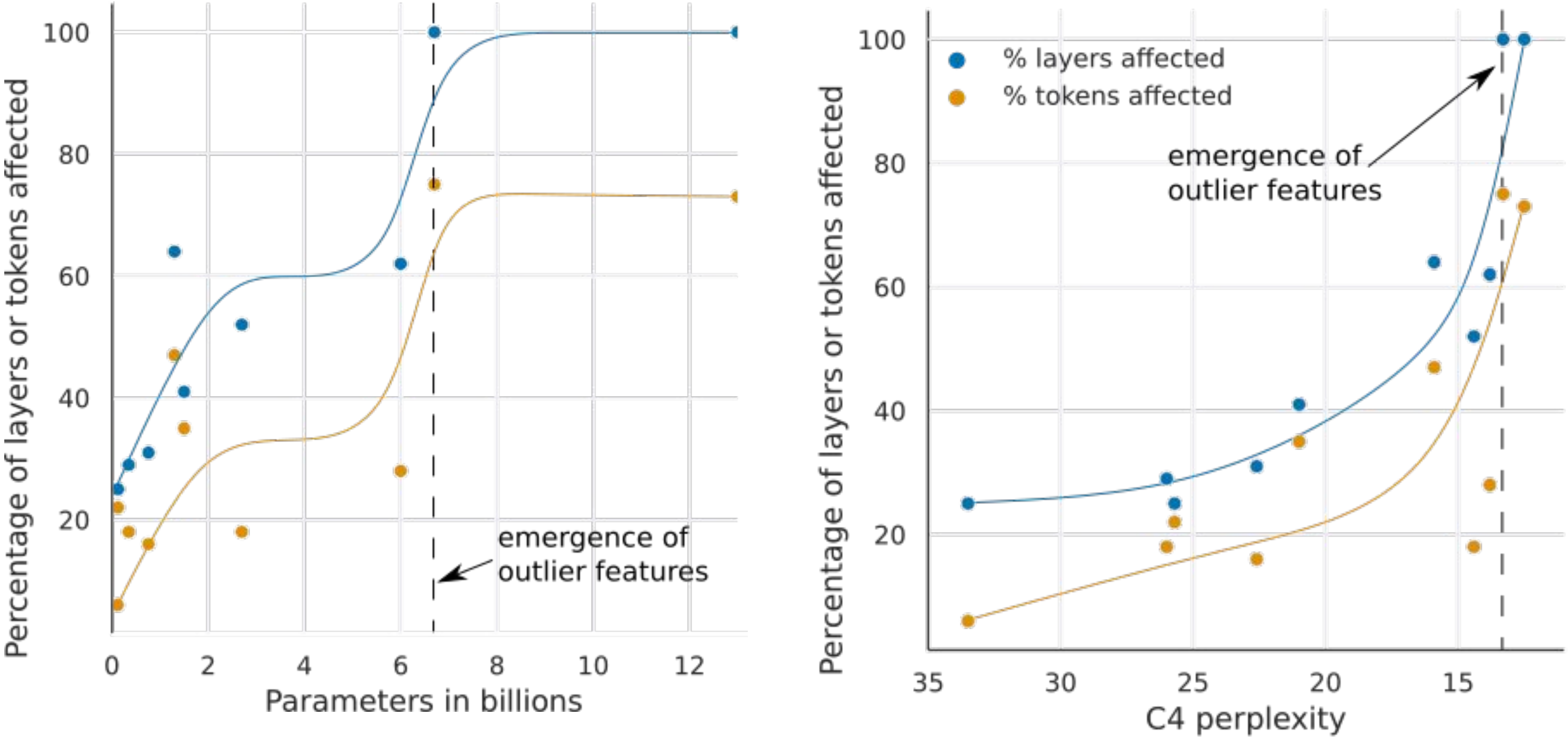
~75% of the time

[0.3, -0.1, -95.0]

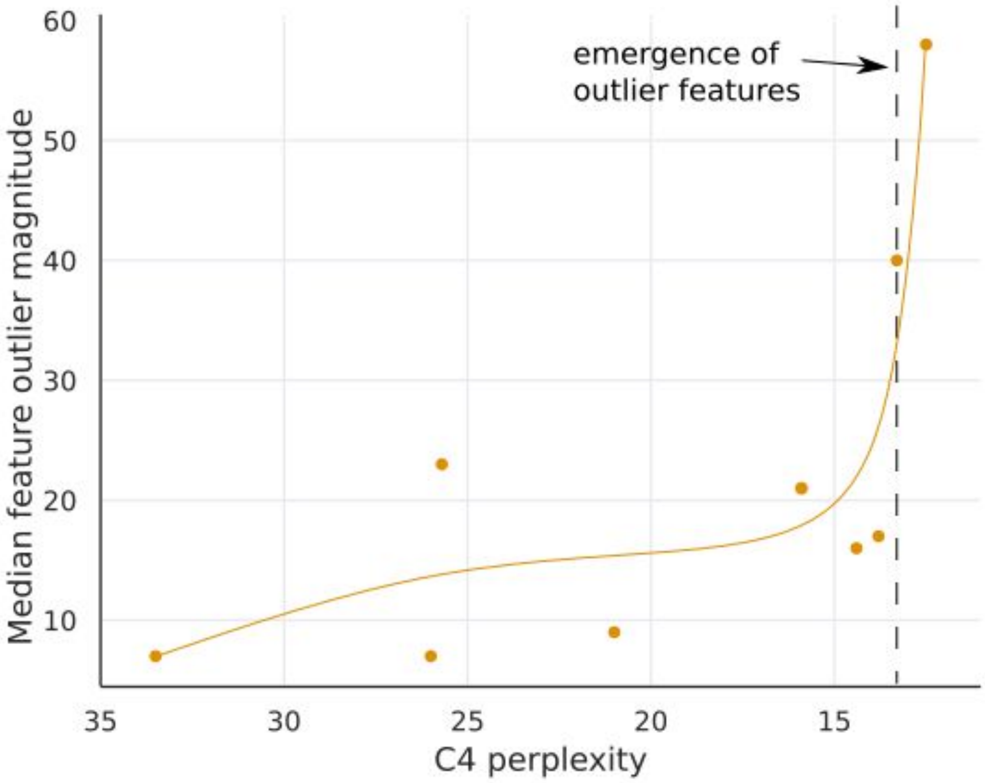
[-0.2, 0.5, -113.0]

[0.3, 0.9, -87.0]

Emergent features: sudden vs. smooth emergence



Emergent features: very large outliers after emergence



Further Analysis: Outliers are important for performance

Take 6.7B transformer language model.

Attention Top-1 probability (single layer):

- Baseline: 40%
- Remove random dimensions: 39.9%
- Remove outliers: 15%

C4 validation perplexity (all layers)

- Baseline: 14.4 ppl
- Remove random dimensions: 14.4 ppl
- Remove outliers: 44.0 ppl

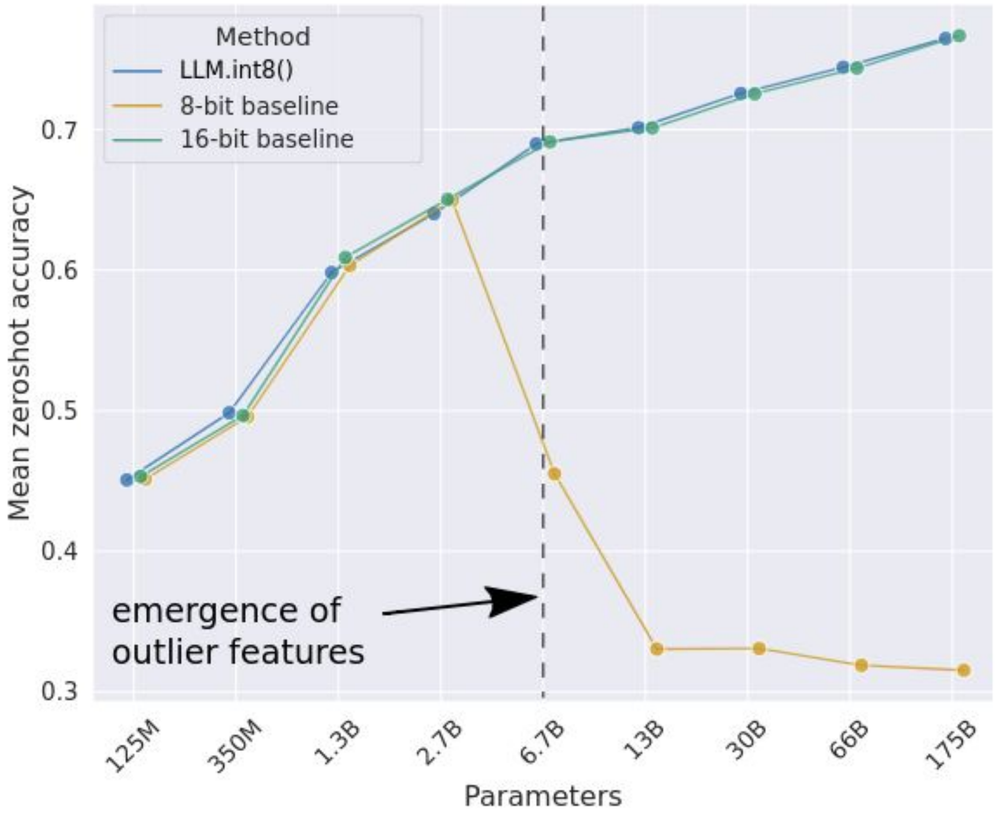
Mixed precision decomposition

Multiply outlier hidden/features dimensions (0.1%) in 16-bit.

Multiply other hidden/features dimensions (99.9%) in 8-bit.

$$\mathbf{C}_{f16} \approx \sum_{h \in O} \mathbf{X}^h_{f16} \mathbf{W}^h_{f16} + \mathbf{S}_{f16} \cdot \sum_{h \notin O} \mathbf{X}^h_{i8} \mathbf{W}^h_{i8}$$

No performance degradation with LLM.int8()



Quantization as a practical tool for memory reduction

Table 2: Different hardware setups and which methods can be run in 16-bit vs. 8-bit precision. We can see that our 8-bit method makes many models accessible that were not accessible before, in particular, OPT-175B/BLOOM.

			Largest Model that can be run	
Class	Hardware	GPU Memory	8-bit	16-bit
Enterprise	8x A100	80 GB	**OPT-175B / BLOOM**	**OPT-175B / BLOOM**
Enterprise	8x A100	40 GB	**OPT-175B / BLOOM**	OPT-66B
Academic server	8x RTX 3090	24s GB	**OPT-175B / BLOOM**	OPT-66B
Academic desktop	4x RTX 3090	24 GB	**OPT-66B**	OPT-30B
Paid Cloud	Colab Pro	15 GB	**OPT-13B**	GPT-J-6B
Free Cloud	Colab	12 GB	**T0/T5-11B**	GPT-2 1.3B

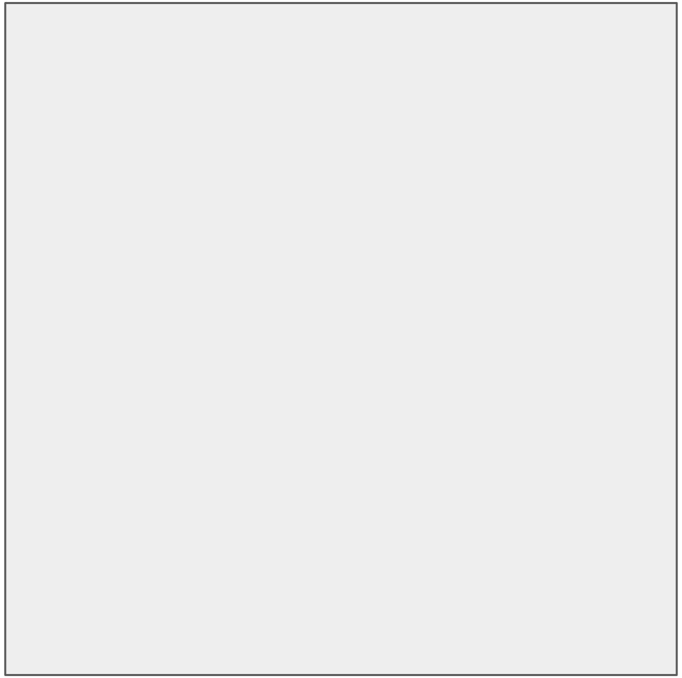
Bit-level Inference Scaling Laws

Inference cost are mostly loading the bits in the weight matrix!

Moderns GPUs can multiply 200 elements in the same time it takes to load 1 element from memory.

Inputs

Weight matrix

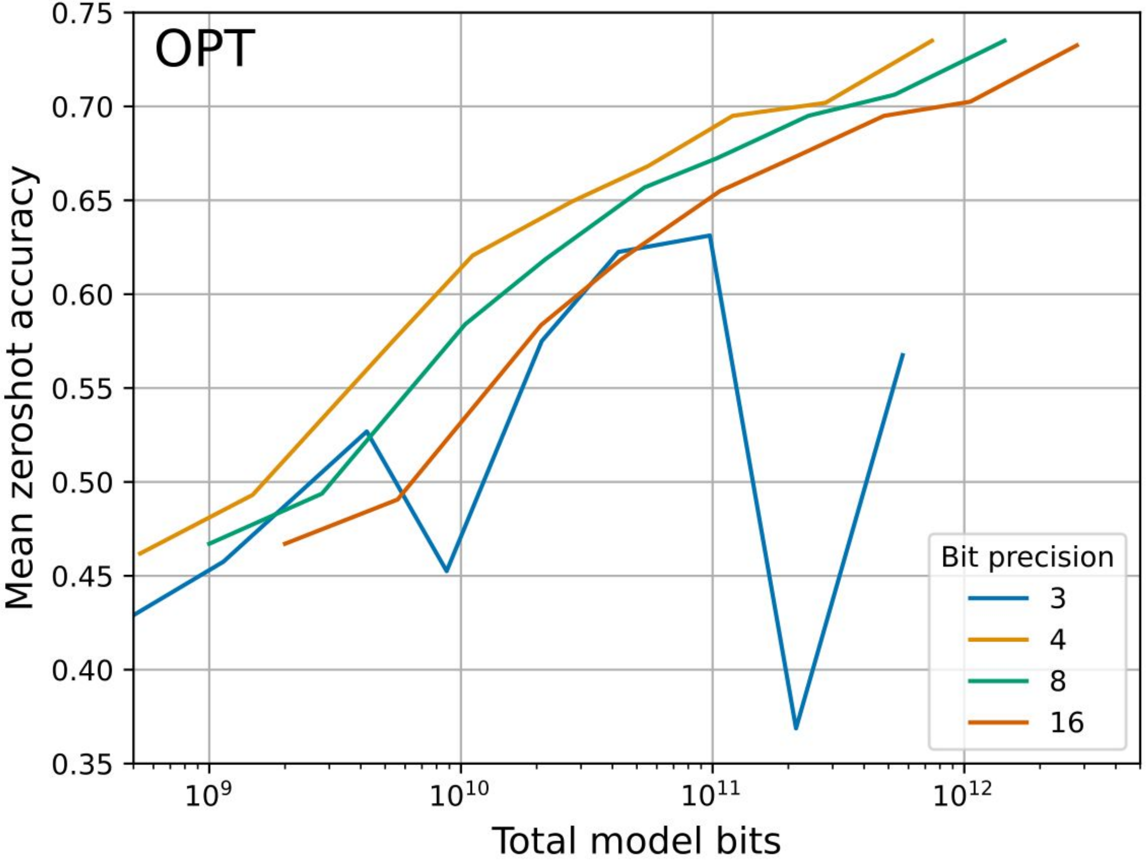


Inference costs of
4-bit 60B and 8-bit 30B LLMs similar

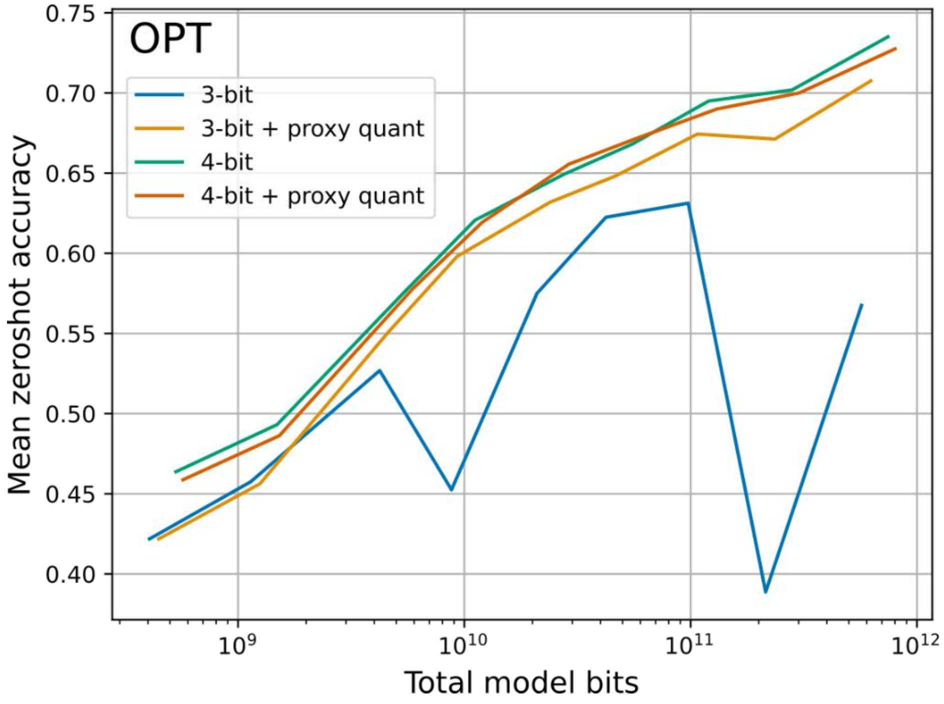
Bit-level scaling laws experimental setup overview

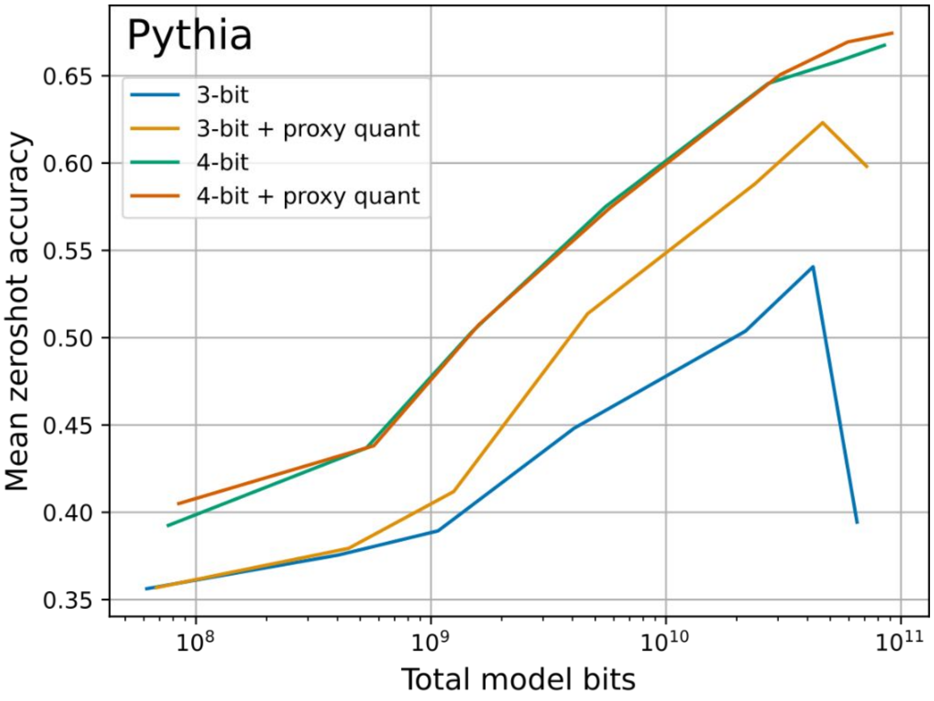
- 35,000 zero-shot experiments (Lambada, Winogrande, PiQA, HellaSwag)
- 19m to 176B parameters
- OPT, BLOOM, BLOOMZ, Pythia/NeoX, GPT-2
- 3 to 8 bit precision (2-bit -> random performance)
- Two quantization concepts: centralization, blocking/grouping
- 4 data types: Integer, Float, dynamic exponent, quantile quantization

Given a zero-shot accuracy, what is the best k-bit quantization?

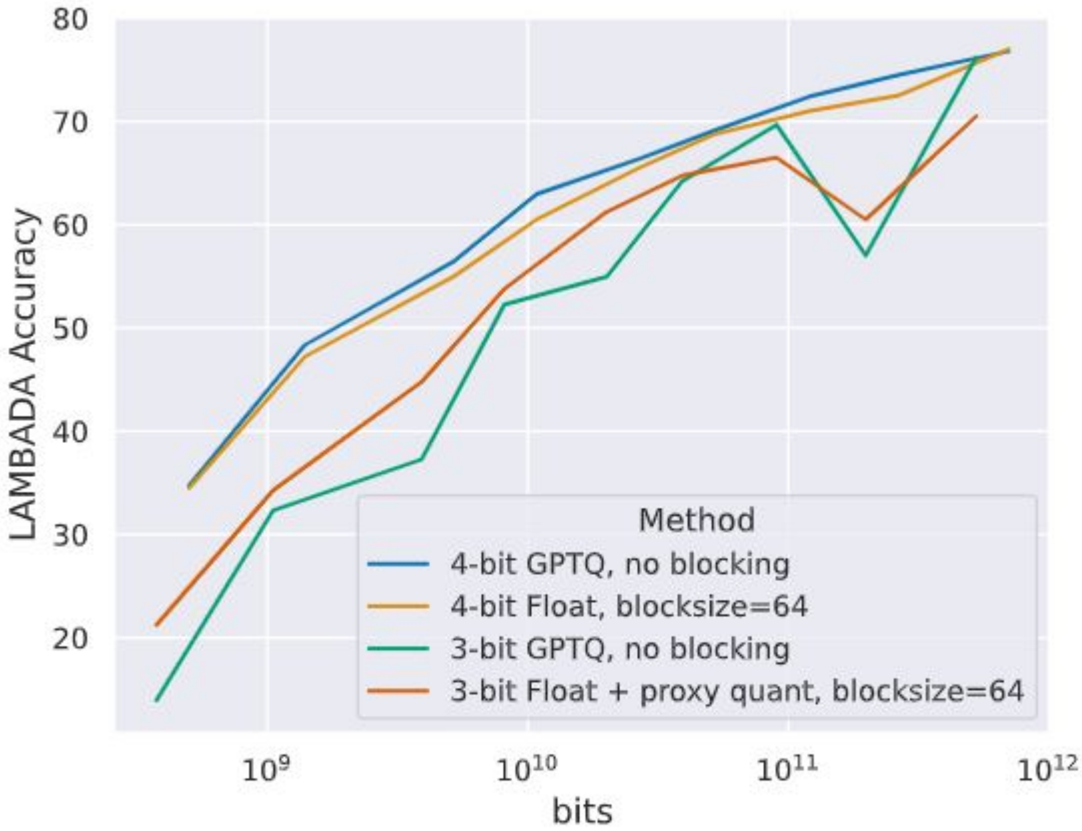


Does it help to treat outliers separately?

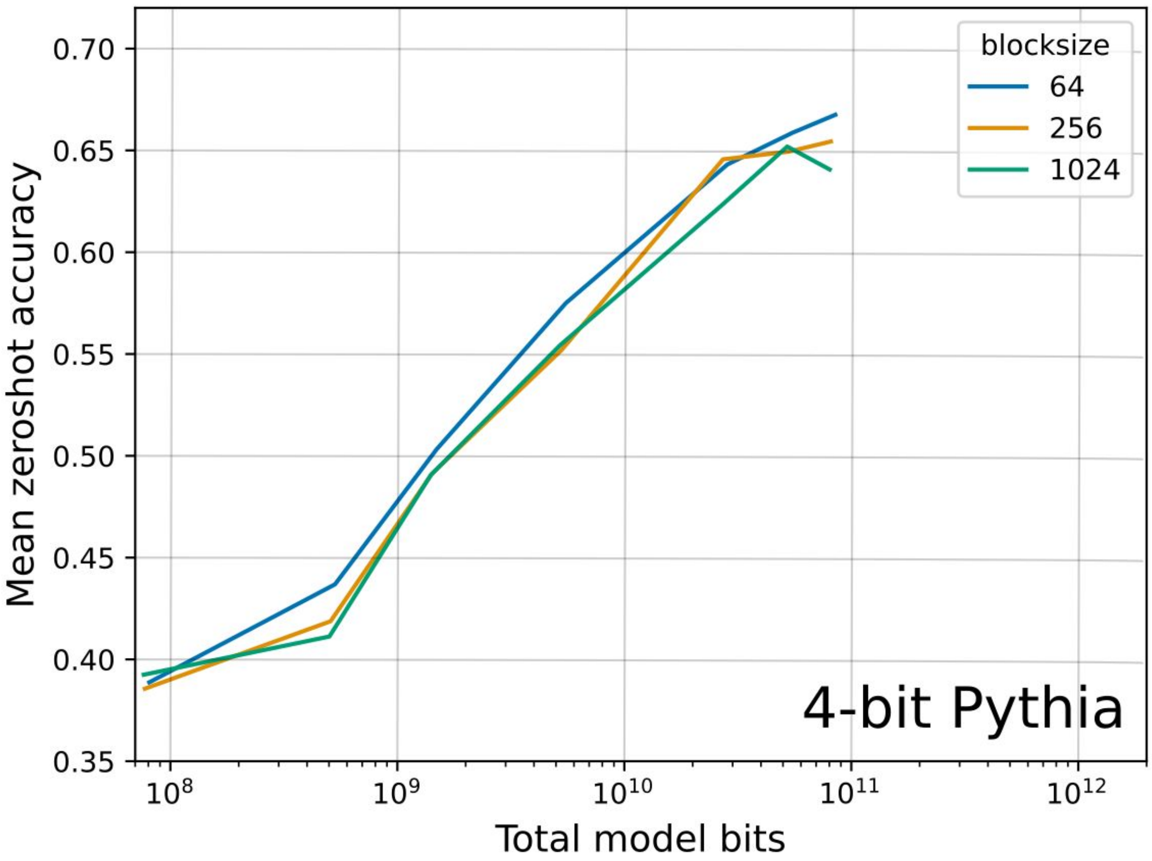




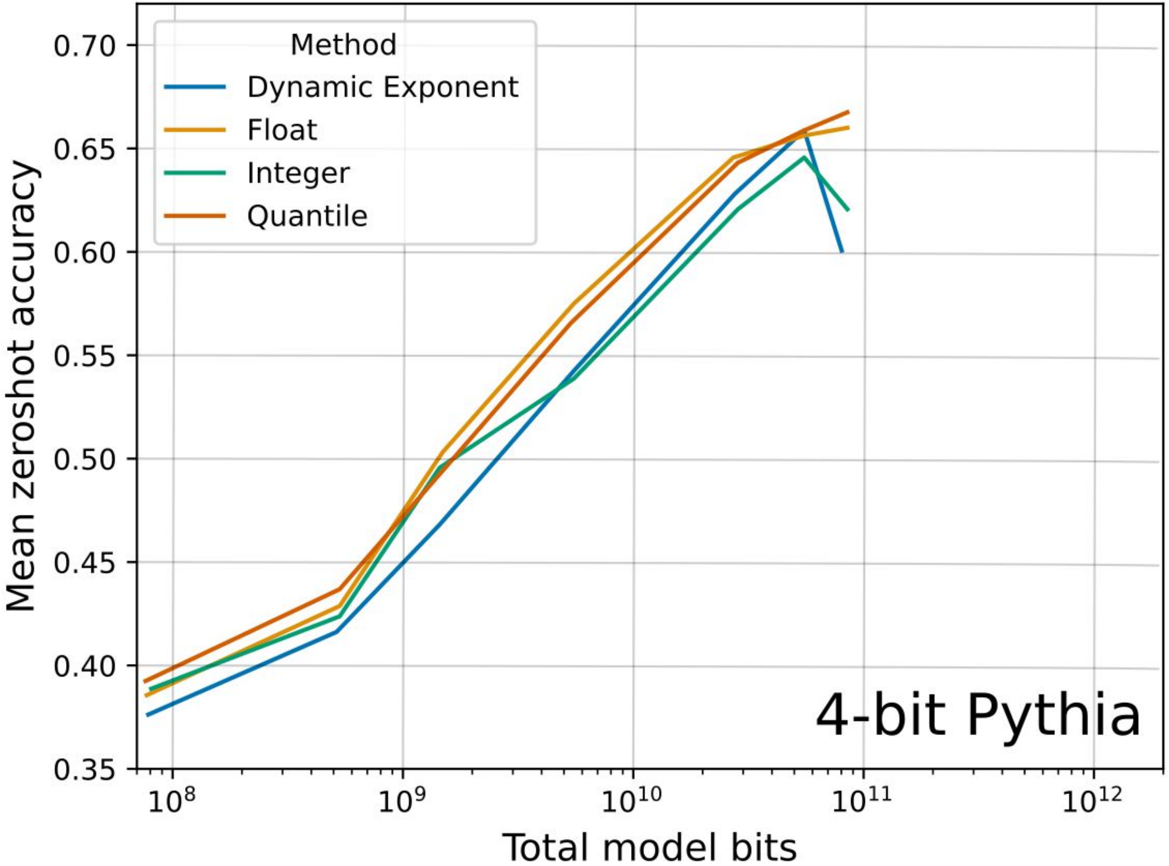
Comparison with GPTQ



What does help to improve scaling? Block size



What does help to improve scaling? Data types



Conclusion

8-bit optimizers make the training and fine-tuning more accessible.

LLM.int8() makes large language models more accessible, for example, zeroshot prompting for OPT-175 on a single node or 65B LLaMA on a single GPU.

Currently, 4-bit precision seems to be best for bit-level scaling of LLM inference. Improving bit-level scaling laws as a measure to improve inference latency.

k-bit methods work well in a variety settings and scales and can make deep learning more efficient and more accessible.