8-bit Methods for Efficient Deep Learning

Tim Dettmers
Large models are not easily accessible

<table>
<thead>
<tr>
<th>Model</th>
<th>Inference memory</th>
<th>Fine-tuning memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5-11B</td>
<td>22 GB</td>
<td>176 GB</td>
</tr>
<tr>
<td>OPT-66B</td>
<td>132 GB</td>
<td>1,056 GB</td>
</tr>
<tr>
<td>BLOOM 176B</td>
<td>350 GB</td>
<td>2,800 GB</td>
</tr>
</tbody>
</table>

Raffel et al., 2020, T5. Zhang et al., 2022, OPT. BigScience, 2022, BLOOM.
Large models are not easily accessible

<table>
<thead>
<tr>
<th>Model</th>
<th>Inference memory</th>
<th>Fine-tuning memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5-11B</td>
<td>22 GB</td>
<td>176 GB</td>
</tr>
<tr>
<td>OPT-66B</td>
<td>132 GB</td>
<td>1,056 GB</td>
</tr>
<tr>
<td>BLOOM 176B</td>
<td>352 GB</td>
<td>2,800 GB</td>
</tr>
</tbody>
</table>

LLM.int8()  8-bit optimizers

<table>
<thead>
<tr>
<th>Model</th>
<th>Inference memory</th>
<th>Fine-tuning memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5-11B</td>
<td>11 GB</td>
<td>66 GB</td>
</tr>
<tr>
<td>OPT-66B</td>
<td>66 GB</td>
<td>396 GB</td>
</tr>
<tr>
<td>BLOOM 176B</td>
<td>176 GB</td>
<td>1,056 GB</td>
</tr>
</tbody>
</table>

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.


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

-1   -0.86   -0.71   -0.57   -0.43 …                  -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: \{\text{Index: 0, 1, 2, 3} \rightarrow \text{Values: -1.0, 0.3, 0.5, 1.0}\}

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

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

- 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

- Weights
- Gradients
- Main Weights
- Adam Buffer 1
- Adam Buffer 2
8-bit optimizers reduce memory consumption by 40%

32-bit to 8-bit

38% mem reduction
What do outliers in quantization look like?
Block-wise quantization
Putting it together: 8-bit optimizers

**Quantization**

Updated optimizer states

<table>
<thead>
<tr>
<th>Optimizer State</th>
<th>-3.1</th>
<th>0.1</th>
<th>-0.03</th>
<th>1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunk into blocks</td>
<td>-3.1</td>
<td>0.1</td>
<td>-0.03</td>
<td>1.2</td>
</tr>
<tr>
<td>Find block-wise absmx</td>
<td>3.1</td>
<td>1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normalize with absmx</td>
<td>-1.0</td>
<td>0.032</td>
<td>-0.025</td>
<td>1.0</td>
</tr>
<tr>
<td>Find closest 8-bit value</td>
<td>-1.0</td>
<td>0.0329</td>
<td>-0.0242</td>
<td>1.0</td>
</tr>
<tr>
<td>Find corresponding index</td>
<td>0</td>
<td>170</td>
<td>80</td>
<td>255</td>
</tr>
</tbody>
</table>

**Dequantization**

Load Index values

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>170</th>
<th>80</th>
<th>255</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup values</td>
<td>-1.0</td>
<td>0.0329</td>
<td>-0.0242</td>
<td>1.0</td>
</tr>
<tr>
<td>Denormalize by absmx</td>
<td>-1.0*3.1</td>
<td>0.0329*3.1</td>
<td>-0.0242*1.2</td>
<td>1.0*1.2</td>
</tr>
<tr>
<td>Dequantized optimizer states</td>
<td>-3.1</td>
<td>0.102</td>
<td>-0.029</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Store index values

Update optimizer states
Results: Same accuracy/perplexity as 32-bit!

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

\(^{\dagger}\)Metric: GLUE=Mean Accuracy/Correlation, CLS/MoCo=Accuracy, MT=BLEU, LM=Perplexity.
Ablations on Language Modeling: All components needed!

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Optimizer</th>
<th>Dynamic</th>
<th>Block-wise</th>
<th>Stable Emb</th>
<th>Unstable (%)</th>
<th>Perplexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>209M</td>
<td>32-bit Adam</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>16.7</td>
</tr>
<tr>
<td></td>
<td>32-bit Adam</td>
<td>✓</td>
<td></td>
<td></td>
<td>0</td>
<td>16.3</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td></td>
<td>✓</td>
<td></td>
<td>90</td>
<td>253.0</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td>✓</td>
<td></td>
<td></td>
<td>50</td>
<td>194.4</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>10</td>
<td>18.6</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>17.7</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>16.8</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>16.4</td>
</tr>
<tr>
<td>1.3B</td>
<td>32-bit Adam</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>10.4</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td>✓</td>
<td></td>
<td></td>
<td>100</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>80</td>
<td>10.9</td>
</tr>
<tr>
<td>1.5B</td>
<td>32-bit Adam</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>9.0</td>
</tr>
<tr>
<td></td>
<td>8-bit Adam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>9.0</td>
</tr>
</tbody>
</table>
LLM.int8()
Large models such as OPT-175B need more than one computer to be run.

- 8x GPU machine ($)
- Fast networking ($$$)
- 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%). It’s 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]

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]
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>83% of the time</td>
<td></td>
<td>17% of the time</td>
</tr>
<tr>
<td>[0.3, -0.1, 0.4]</td>
<td></td>
<td>[0.3, -0.1, -15.0]</td>
</tr>
<tr>
<td>[-0.2, 0.5, 0.1]</td>
<td></td>
<td>[-0.2, 0.5, -17.0]</td>
</tr>
<tr>
<td>[0.3, 0.9, -0.7]</td>
<td></td>
<td>[0.3, 0.9, -22.0]</td>
</tr>
</tbody>
</table>
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

- Graph 1: Percentage of layers or tokens affected vs. Parameters in billions.
- Graph 2: Percentage of layers or tokens affected vs. C4 perplexity, showing emergence of outlier features.
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.

\[
C_{f16} \approx \sum_{h \in O} X_{f16}^h W_{f16}^h + S_{f16} \cdot \sum_{h \notin O} X_{i8}^h W_{i8}^h
\]
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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Hardware</th>
<th>GPU Memory</th>
<th>Largest Model that can be run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise</td>
<td>8x A100</td>
<td>80 GB</td>
<td>OPT-175B / BLOOM</td>
</tr>
<tr>
<td>Enterprise</td>
<td>8x A100</td>
<td>40 GB</td>
<td>OPT-175B / BLOOM</td>
</tr>
<tr>
<td>Academic server</td>
<td>8x RTX 3090</td>
<td>24s GB</td>
<td>OPT-175B / BLOOM</td>
</tr>
<tr>
<td>Academic desktop</td>
<td>4x RTX 3090</td>
<td>24 GB</td>
<td>OPT-66B</td>
</tr>
<tr>
<td>Paid Cloud</td>
<td>Colab Pro</td>
<td>15 GB</td>
<td>OPT-13B</td>
</tr>
<tr>
<td>Free Cloud</td>
<td>Colab</td>
<td>12 GB</td>
<td>T0/T5-11B</td>
</tr>
</tbody>
</table>

8-bit                                      16-bit

OPT-175B / BLOOM  OPT-175B / BLOOM
OPT-175B / BLOOM  OPT-66B
OPT-175B / BLOOM  OPT-66B
OPT-66B           OPT-30B
OPT-13B           GPT-J-6B
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.

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

![Comparison with GPTQ](image-url)
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.