CS388: Natural Language Processing

Lecture 21: Efficiency and LLMs

Greg Durrett
Announcements

- Check-ins due today, will be graded as promptly as we can
- Final presentations start in 2.5 weeks, reports due May 3
This Lecture

- Decoding optimizations: exact decoding, but faster
  - Speculative decoding
  - Medusa heads
  - Flash attention

- Model pruning
  - Pruning LLMs
  - Distilling LLMs

- Model compression
Decoding Optimizations
Decoding Basics

I saw the dog running to the

Prompt (prefix of $p$ tokens)  Decoded tokens ($k$)

Operations for one decoder pass: $O(pL)$
Operations for $k$ decoder passes: $O(pk^2L)$

Transformer layers (non-parallelizable ops): $O(kL)$
Speculative Decoding

I saw the dog running to the house quickly

Prompt (prefix of $p$ tokens) Decoded tokens ($k$)

- Key idea: a forward pass for several tokens at a time is $O(L)$ serial steps, since the tokens can be computed in parallel
- Can we predict many tokens with a weak model and then “check” them with a single forward pass?
Speculative Decoding

I saw the dog running

Distribution over vocabulary

Prompt (prefix of $p$ tokens) Decoded tokens ($k$)

- When sampling, we need the whole distribution
- When doing greedy decoding, we only need to know what token was the max
We can use a small, cheap model to do inference, then check that “to”, “the”, “house”, “quickly” are really the best tokens from a bigger model

Leviathan et al. (2023)
Speculative Decoding: Flow

I saw the dog running to the house quickly

- Produce decoded tokens one at a time from a fast draft model...

I saw the dog running to the house quickly

- Confirm that the tokens are the max tokens from the slower main model. Any “wrong” token invalidates the rest of the sequence
Speculative Decoding

Can also adjust this to use sampling. Treat this as a proposal distribution $q(x)$ and may need to reject + resample (rejection sampling)
Speculative Decoding

- Find the first index that was rejected by the sampling procedure, then resample from there

\[ \text{Inputs: } M_p, M_q, \text{prefix.} \]
\[ \forall \gamma \text{ guesses } x_1, \ldots, x_\gamma \text{ from } M_q \text{ autoregressively.} \]
\[ \text{for } i = 1 \text{ to } \gamma \text{ do} \]
\[ q_i(x) \leftarrow M_q(\text{prefix} + [x_1, \ldots, x_{i-1}]) \]
\[ x_i \sim q_i(x) \]
\[ \text{end for} \]
\[ \forall \text{ Run } M_p \text{ in parallel.} \]
\[ p_1(x), \ldots, p_{\gamma+1}(x) \leftarrow \]
\[ M_p(\text{prefix}), \ldots, M_p(\text{prefix} + [x_1, \ldots, x_\gamma]) \]
\[ \forall \text{ Determine the number of accepted guesses } n. \]
\[ r_1 \sim U(0, 1), \ldots, r_\gamma \sim U(0, 1) \]
\[ n \leftarrow \min(\{i - 1 \mid 1 \leq i \leq \gamma, r_i > \frac{p_i(x)}{q_i(x)}\} \cup \{\gamma\}) \]
\[ \forall \text{ Adjust the distribution from } M_p \text{ if needed.} \]
\[ p'(x) \leftarrow p_{n+1}(x) \]
\[ \text{if } n < \gamma \text{ then} \]
\[ p'(x) \leftarrow \text{norm}(\max(0, p_{n+1}(x) - q_{n+1}(x))) \]
\[ \text{end if} \]
\[ \forall \text{ Return one token from } M_p, \text{ and } n \text{ tokens from } M_q. \]
\[ t \sim p'(x) \]
\[ \text{return } \text{prefix} + [x_1, \ldots, x_n, t] \]
Medusa Heads

- The “draft model” consists of multiple prediction heads trained to predict the next k tokens

https://www.together.ai/blog/medusa
Evaluate multiple candidates at once using a customized attention layer. In this image: 2 x 3 candidates

https://www.together.ai/blog/medusa
Medusa Heads

- Speedup with no loss in accuracy!

https://www.together.ai/blog/medusa
Other Decoding Improvements

- Most other approaches to speeding up require changing the model (making a faster Transformer) or making it smaller (distillation, pruning; discussed next)

- Batching parallelism: improve throughput by decoding many examples in parallel. (Does not help with latency, and it’s a little bit harder to do in production if requests are coming in asynchronously)

- Low-level hardware optimizations?
  
  - Easy things like caching (KV cache: keys + values for context tokens are cached across multiple tokens)
Flash Attention

- Does extra computation during attention, but avoids expensive reads/writes to GBU “high-bandwidth memory.” Recomputation is all in SRAM and is very fast
- Essentially: store a running sum for the softmax, compute values as needed
Flash Attention

Algorithm 0 Standard Attention Implementation

Require: Matrices $Q, K, V \in \mathbb{R}^{N \times d}$ in HBM.
1: Load $Q, K$ by blocks from HBM, compute $S = QK^T$, write $S$ to HBM.
2: Read $S$ from HBM, compute $P = \text{softmax}(S)$, write $P$ to HBM.
3: Load $P$ and $V$ by blocks from HBM, compute $O = PV$, write $O$ to HBM.
4: Return $O$.

Algorithm 1 FlashAttention

Require: Matrices $Q, K, V \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size $M$.

[dividing stuff into blocks]
5: for $1 \leq j \leq T_c$ do
6:  Load $K_j, V_j$ from HBM to on-chip SRAM.
7:  for $1 \leq i \leq T_r$ do
8:   Load $Q_i, O_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9:   On chip, compute $S_{ij} = Q_i K_j^T \in \mathbb{R}^{B_r \times B_c}$.

[more computation, writes to HBM]
Flash Attention

<table>
<thead>
<tr>
<th>Models</th>
<th>ListOps</th>
<th>Text</th>
<th>Retrieval</th>
<th>Image</th>
<th>Pathfinder</th>
<th>Avg</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>36.0</td>
<td>63.6</td>
<td>81.6</td>
<td>42.3</td>
<td>72.7</td>
<td>59.3</td>
<td>—</td>
</tr>
<tr>
<td>FLASH_ATTENTION</td>
<td>37.6</td>
<td>63.9</td>
<td>81.4</td>
<td>43.5</td>
<td>72.7</td>
<td>59.8</td>
<td>2.4×</td>
</tr>
<tr>
<td>Block-sparse FLASH_ATTENTION</td>
<td>37.0</td>
<td>63.0</td>
<td>81.3</td>
<td>43.6</td>
<td>73.3</td>
<td>59.6</td>
<td>2.8×</td>
</tr>
<tr>
<td>Linformer [84]</td>
<td>35.6</td>
<td>55.9</td>
<td>77.7</td>
<td>37.8</td>
<td>67.6</td>
<td>54.9</td>
<td>2.5×</td>
</tr>
<tr>
<td>Linear Attention [50]</td>
<td>38.8</td>
<td>63.2</td>
<td>80.7</td>
<td>42.6</td>
<td>72.5</td>
<td>59.6</td>
<td>2.3×</td>
</tr>
<tr>
<td>Performer [12]</td>
<td>36.8</td>
<td>63.6</td>
<td>82.2</td>
<td>42.1</td>
<td>69.9</td>
<td>58.9</td>
<td>1.8×</td>
</tr>
<tr>
<td>Local Attention [80]</td>
<td>36.1</td>
<td>60.2</td>
<td>76.7</td>
<td>40.6</td>
<td>66.6</td>
<td>56.0</td>
<td>1.7×</td>
</tr>
<tr>
<td>Reformer [51]</td>
<td>36.5</td>
<td>63.8</td>
<td>78.5</td>
<td>39.6</td>
<td>69.4</td>
<td>57.6</td>
<td>1.3×</td>
</tr>
<tr>
<td>Smyrf [19]</td>
<td>36.1</td>
<td>64.1</td>
<td>79.0</td>
<td>39.6</td>
<td>70.5</td>
<td>57.9</td>
<td>1.7×</td>
</tr>
</tbody>
</table>

- Gives a speedup for free — with no cost in accuracy (modulo numeric instability)
- Outperforms the speedup from many other approximate Transformer methods, which perform substantially worse
Model Compression
Approaches to Compression

‣ Pruning: can we reduce the number of neurons in the model?
  ‣ Basic idea: remove low-magnitude weights

‣ Issue: sparse matrices are not fast, matrix multiplication is very fast on GPUs so you don’t save any time!
Approaches to Compression

‣ Pruning: can we reduce the number of neurons in the model?
  ‣ Basic idea: remove low-magnitude weights
  ‣ Instead, we want some kind of structured pruning. What does this look like?

‣ Still a challenge: if different layers have different sizes, your GPU utilization may go down
Idea 1: targeted structured pruning

Parameterization and regularization encourage sparsity, even though the z’s are continuous

Sheared Llama

Mengzhou Xia et al. (2023)
Train for a while with the z’s, then prune the network. Then enter stage 2: continued pre-training on new data

Idea 2: dynamic batch loading. Update the weights controlling the mix of data you use during pre-training (sample more from domains of data with high loss)
### Sheared Llama

<table>
<thead>
<tr>
<th>Model</th>
<th>#tokens for training</th>
<th>Continued LogiQA</th>
<th>BoolQ (32)</th>
<th>LM LAMBADA</th>
<th>World Knowledge NQ (32)</th>
<th>World Knowledge MMLU (5)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLaMA2-7B</td>
<td>(2T)†</td>
<td>30.7</td>
<td>82.1</td>
<td>28.8</td>
<td>73.9</td>
<td>46.6</td>
<td>64.6</td>
</tr>
<tr>
<td>OPT-1.3B</td>
<td>(300B)†</td>
<td><strong>26.9</strong></td>
<td>57.5</td>
<td>58.0</td>
<td>6.9</td>
<td>24.7</td>
<td>48.2</td>
</tr>
<tr>
<td>Pythia-1.4B</td>
<td>(300B)†</td>
<td>27.3</td>
<td>57.4</td>
<td><strong>61.6</strong></td>
<td>6.2</td>
<td><strong>25.7</strong></td>
<td>48.9</td>
</tr>
<tr>
<td>Sheared-LLaMA-1.3B</td>
<td>(50B)</td>
<td><strong>26.9</strong></td>
<td><strong>64.0</strong></td>
<td>61.0</td>
<td><strong>9.6</strong></td>
<td><strong>25.7</strong></td>
<td><strong>51.0</strong></td>
</tr>
<tr>
<td>OPT-2.7B</td>
<td>(300B)†</td>
<td>26.0</td>
<td>63.4</td>
<td>63.6</td>
<td>10.1</td>
<td>25.9</td>
<td>51.4</td>
</tr>
<tr>
<td>Pythia-2.8B</td>
<td>(300B)†</td>
<td>28.0</td>
<td>66.0</td>
<td>64.7</td>
<td>9.0</td>
<td>26.9</td>
<td>52.5</td>
</tr>
<tr>
<td>INCITE-Base-3B</td>
<td>(800B)</td>
<td>27.7</td>
<td>65.9</td>
<td>65.3</td>
<td>14.9</td>
<td><strong>27.0</strong></td>
<td>54.7</td>
</tr>
<tr>
<td>Open-LLaMA-3B-v1</td>
<td>(1T)</td>
<td>28.4</td>
<td>70.0</td>
<td>65.4</td>
<td><strong>18.6</strong></td>
<td><strong>27.0</strong></td>
<td>55.1</td>
</tr>
<tr>
<td>Open-LLaMA-3B-v2</td>
<td>(1T)†</td>
<td>28.1</td>
<td>69.6</td>
<td>66.5</td>
<td>17.1</td>
<td>26.9</td>
<td>55.7</td>
</tr>
<tr>
<td>Sheared-LLaMA-2.7B</td>
<td>(50B)</td>
<td><strong>28.9</strong></td>
<td><strong>73.7</strong></td>
<td><strong>68.4</strong></td>
<td>16.5</td>
<td>26.4</td>
<td><strong>56.7</strong></td>
</tr>
</tbody>
</table>

- (Slightly) better than models that were “organically” trained at these larger scales

Mengzhou Xia et al. (2023)
Approaches to Compression

- Pruning: can we reduce the number of neurons in the model?
  - Basic idea: remove low-magnitude weights
  - Instead, we want some kind of structured pruning. What does this look like?
- Knowledge distillation
  - Classic approach from Hinton et al.: train a student model to match distribution from teacher
Suppose we have a classification model with output $P_{teacher}(y \mid x)$

Minimize $KL(P_{teacher} \mid \mid P_{student})$ to bring student dist close to teacher

Note that this is not using labels — it uses the teacher to “pseudo-label” data, and we label an entire distribution, not just a top-one label
DistilBERT

- Use a teacher model as a large neural network, such as BERT
- Make a small student model that is half the layers of BERT. Initialize with every other layer from the teacher

Sanh et al. (2019)
DistilBERT

<table>
<thead>
<tr>
<th>Model</th>
<th>Score</th>
<th>CoLA</th>
<th>MNLI</th>
<th>MRPC</th>
<th>QNLI</th>
<th>QQP</th>
<th>RTE</th>
<th>SST-2</th>
<th>STS-B</th>
<th>WNLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELMo</td>
<td>68.7</td>
<td>44.1</td>
<td>68.6</td>
<td>76.6</td>
<td>71.1</td>
<td>86.2</td>
<td>53.4</td>
<td>91.5</td>
<td>70.4</td>
<td>56.3</td>
</tr>
<tr>
<td>BERT-base</td>
<td>79.5</td>
<td>56.3</td>
<td>86.7</td>
<td>88.6</td>
<td>91.8</td>
<td>89.6</td>
<td>69.3</td>
<td>92.7</td>
<td>89.0</td>
<td>53.5</td>
</tr>
<tr>
<td>DistilBERT</td>
<td>77.0</td>
<td>51.3</td>
<td>82.2</td>
<td>87.5</td>
<td>89.2</td>
<td>88.5</td>
<td>59.9</td>
<td>91.3</td>
<td>86.9</td>
<td>56.3</td>
</tr>
</tbody>
</table>

Table 2: DistilBERT yields to comparable performance on downstream tasks. Comparison on downstream tasks: IMDb (test accuracy) and SQuAD 1.1 (EM/F1 on dev set). D: with a second step of distillation during fine-tuning.

<table>
<thead>
<tr>
<th>Model</th>
<th>IMDb (acc.)</th>
<th>SQuAD (EM/F1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-base</td>
<td>93.46</td>
<td>81.2/88.5</td>
</tr>
<tr>
<td>DistilBERT</td>
<td>92.82</td>
<td>77.7/85.8</td>
</tr>
<tr>
<td>DistilBERT (D)</td>
<td>-</td>
<td>79.1/86.9</td>
</tr>
</tbody>
</table>

Table 3: DistilBERT is significantly smaller while being constantly faster. Inference time of a full pass of GLUE task STS-B (sentiment analysis) on CPU with a batch size of 1.

<table>
<thead>
<tr>
<th>Model</th>
<th># param. (Millions)</th>
<th>Inf. time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELMo</td>
<td>180</td>
<td>895</td>
</tr>
<tr>
<td>BERT-base</td>
<td>110</td>
<td>668</td>
</tr>
<tr>
<td>DistilBERT</td>
<td>66</td>
<td>410</td>
</tr>
</tbody>
</table>

Sanh et al. (2019)
- Other Distillation

### Data

<table>
<thead>
<tr>
<th>Premise: A person on a horse jumps over a broken down airplane. Hypothesis: A person is training his horse for a competition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question: A gentleman is carrying equipment for golf, what is he likely to have?</td>
</tr>
<tr>
<td>Answers: (a) club (b) assembly hall (c) meditation center (d) meeting, (e) church</td>
</tr>
<tr>
<td>Luke scored 84 points after playing 2 rounds of a trivia game. If he gained the same number of points each round. How many points did he score per round?</td>
</tr>
</tbody>
</table>

### Rationale

| The person could be training his horse for a competition, but it is not necessarily the case. |
| The answer must be something that is used for golf. Of the above choices, only clubs are used for golf. So the answer is (a) club |
| Luke scored 84 points after 2 rounds. So he scored 84 points in 2 rounds. 84 / 2 = 42. The answer is (84 / 2) |

### Label

| neutral | club | (84 / 2) |

- How to distill models for complex reasoning settings? Still an open problem!  
  Cheng-Yu Hsieh et al. (2023)
Parameter-Efficient Tuning
Parameter-Efficient Tuning

- Rather than train all model parameters at once, can we get away with just training a small number of them?

- What are the advantages of this?

- Typical advantages: lower memory, easier to serve many models for use cases like personalization or multitasking

- Not an advantage: faster (it’s not)
\[ Q^{m,\ell}(x) = W_q^{m,\ell}x + b_q^{m,\ell} \]
\[ K^{m,\ell}(x) = W_k^{m,\ell}x + b_k^{m,\ell} \]
\[ V^{m,\ell}(x) = W_v^{m,\ell}x + b_v^{m,\ell} \]

\[ h_1^\ell = \text{att}(Q^{1,\ell}, K^{1,\ell}, V^{1,\ell}, ..., Q^{m,\ell}, K^{m,\ell}, V^{m,\ell}) \]

and then fed to an MLP with layer-norm (LN):

\[ h_2^\ell = \text{Dropout}(W_{m_1}^\ell \cdot h_1^\ell + b_{m_1}^\ell) \quad (1) \]

\[ h_3^\ell = g_{LN_1}^\ell \odot \frac{(h_2^\ell + x) - \mu}{\sigma} + b_{LN_1}^\ell \quad (2) \]

\[ h_4^\ell = \text{GELU}(W_{m_2}^\ell \cdot h_3^\ell + b_{m_2}^\ell) \quad (3) \]

\[ h_5^\ell = \text{Dropout}(W_{m_3}^\ell \cdot h_4^\ell + b_{m_3}^\ell) \quad (4) \]

\[ \text{out}^\ell = g_{LN_2}^\ell \odot \frac{(h_5^\ell + h_3^\ell) - \mu}{\sigma} + b_{LN_2}^\ell \quad (5) \]

\[
\begin{align*}
\text{Tune only the bias terms of the Transformer architecture, don’t fine-tune the weights} \\
\text{How many parameters do you think this is?}
\end{align*}
\]
Degraded performance, but only train <0.1% of the parameters of the full model!
LoRA

- Alternative: learn weight matrices as $(W + BA)$, where $BA$ is a product of two low-rank matrices.
  - If we have a $d \times d$ matrix and we use a rank reduction of size $r$, what is the parameter reduction from LoRA?
- Allows adding low-rank matrix on top of existing high-rank model
- Unlike some other methods, LoRA can be “compiled down” into the model (just add $BA$ into $W$)

Figure 1: Our reparametrization. We only train $A$ and $B$.

Hu et al. (2021)
LoRA

<table>
<thead>
<tr>
<th>Model &amp; Method</th>
<th># Trainable Parameters</th>
<th>MNLI</th>
<th>SST-2</th>
<th>MRPC</th>
<th>CoLA</th>
<th>QNLI</th>
<th>QQP</th>
<th>RTE</th>
<th>STS-B</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoB_{base} (FT)*</td>
<td>125.0M</td>
<td>87.6</td>
<td>94.8</td>
<td>90.2</td>
<td>63.6</td>
<td>92.8</td>
<td>91.9</td>
<td>78.7</td>
<td>91.2</td>
<td>86.4</td>
</tr>
<tr>
<td>RoB_{base} (BitFit)*</td>
<td>0.1M</td>
<td>84.7</td>
<td>93.7</td>
<td>92.7</td>
<td>62.0</td>
<td>91.8</td>
<td>84.0</td>
<td>81.5</td>
<td>90.8</td>
<td>85.2</td>
</tr>
<tr>
<td>RoB_{base} (Adpt^D)*</td>
<td>0.3M</td>
<td>87.1±0.1</td>
<td>94.2±1.1</td>
<td>88.5±1.1</td>
<td>60.8±4.1</td>
<td>93.1±1.1</td>
<td>90.2±0.2</td>
<td>71.5±2.7</td>
<td>89.7±3.1</td>
<td>84.4</td>
</tr>
<tr>
<td>RoB_{base} (Adpt^D)*</td>
<td>0.9M</td>
<td>87.3±1.1</td>
<td>94.7±3.1</td>
<td>88.4±1.1</td>
<td>62.6±9.1</td>
<td>93.0±2.2</td>
<td>90.6±0.2</td>
<td>75.9±2.2</td>
<td>90.3±1.1</td>
<td>85.4</td>
</tr>
<tr>
<td>RoB_{base} (LoRA)</td>
<td>0.3M</td>
<td>87.5±3</td>
<td>95.1±2</td>
<td>89.7±7</td>
<td>63.4±1.2</td>
<td>93.3±3</td>
<td>90.8±1.3</td>
<td>86.6±7</td>
<td>91.5±2.2</td>
<td>87.2</td>
</tr>
<tr>
<td>RoB_{large} (FT)*</td>
<td>355.0M</td>
<td>90.2</td>
<td>96.4</td>
<td>90.9</td>
<td>68.0</td>
<td>94.7</td>
<td>92.2</td>
<td>86.6</td>
<td>92.4</td>
<td>88.9</td>
</tr>
<tr>
<td>RoB_{large} (LoRA)</td>
<td>0.8M</td>
<td>90.6±2</td>
<td>96.2±5</td>
<td>90.9±1.2</td>
<td>68.2±1.9</td>
<td>94.9±3</td>
<td>91.6±1.3</td>
<td>87.4±2.5</td>
<td>92.6±2.2</td>
<td>89.0</td>
</tr>
</tbody>
</table>

- LoRA is much better than BitFit, even better than vanilla fine-tuning on GLUE!

Hu et al. (2021)
LLM Quantization

- A significant fraction of LLM training is just storing the weights
  - Normal floating-point precision: 4 bytes per weight, gets large for 10B+ parameter models!
- How much is needed for fine-tuning?
  - The Adam optimizer has to store at least 2 additional values for each parameter (first- and second-moment estimates)
  - Memory gets very large! Can we reduce this?
LLM Quantization

<table>
<thead>
<tr>
<th>Format</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE 754 Single Precision 32-bit Float (FP32)</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>IEEE 754 Half Precision 16-bit Float (FP16)</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Google Brain Float (BF 16)</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Nvidia FP8 (E4M3)</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

slide credit: Tianjian Li
### LLM Quantization

#### Original 32-bit float

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.09</td>
<td>-0.98</td>
<td>1.48</td>
<td>0.09</td>
</tr>
<tr>
<td>0.05</td>
<td>-0.14</td>
<td>-1.08</td>
<td>2.12</td>
</tr>
<tr>
<td>-0.91</td>
<td>1.92</td>
<td>0</td>
<td>-1.03</td>
</tr>
<tr>
<td>1.87</td>
<td>0</td>
<td>1.53</td>
<td>1.49</td>
</tr>
</tbody>
</table>

#### Quantized 2-bit signed int

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-2</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>-2</td>
<td>1</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Reconstructed 32-bit float

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.14</td>
<td>-1.07</td>
<td>1.07</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1.07</td>
<td>2.14</td>
</tr>
<tr>
<td>-1.07</td>
<td>2.14</td>
<td>0</td>
<td>-1.07</td>
</tr>
<tr>
<td>2.14</td>
<td>0</td>
<td>1.07</td>
<td>1.07</td>
</tr>
</tbody>
</table>

- Outlier weights can make it hard to find a good zero point/scale

slide credit: Tianjian Li
LLM Quantization

8-bit Vector-wise Quantization

(1) Find vector-wise constants: $C_W$ & $C_X$
(2) Quantize

$$X_{16}^{*} \frac{(127/C_X)}{} = X_{18}$$
$$W_{16}^{*} \frac{(127/C_W)}{} = W_{18}$$

(3) Int8 Matmul

$$X_{18} \cdot W_{18} = Out_{132}$$

(4) Dequantize

$$Out_{132} \frac{(C_X \otimes C_W)}{} = Out_{F16} \frac{(127*127)}{}$$

16-bit Decomposition

(1) Decompose outliers
(2) FP16 Matmul

$$X_{F16} \cdot W_{F16} = Out_{F16}$$

Solution: combine 8-bit and 16-bit quantization, where most stuff is 8-bit quantized

Dettmers et al. (2022)
## LLM Quantization

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

- Validation perplexity on language modeling. Prior Int8 techniques degrade, the decomposition maintains performance

Dettmers et al. (2022)
LLM Quantization

- Interestingly, the outlier features that require 16-bit quantization emerge at large scale.
QLoRA: Memory-efficient training

- 4-bit “normal float”, takes advantage of the fact that NN weights typically have a zero-centered normal distribution
- Paged optimizer state to avoid memory spikes (due to training examples with long sequence length)
Where is this going?

- **Better GPU programming**: as GPU performance starts to saturate, we’ll probably see more algorithms tailored very specifically to the affordances of the hardware.

- **Small models**, either distilled or trained from scratch: as LLMs get better, we can do with ~7B scale what used to be only doable with ChatGPT (GPT-3.5).

- **Continued focus on faster inference**: faster inference can be highly impactful across all LLM applications.
Decoding optimizations: speculative decoding gives a fast way to exactly sample from a smaller model. Also techniques like Flash Attention

Model optimizations to make models smaller: pruning, distillation

Model compression and quantization: standard compression techniques, but adapted to work really well for GPUs