GIST: Efficient Data Encoding for Deep Neural Network Training

Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko
Presented by: Peniel Argaw, Elizabeth Bondi, Hamish Nicholson, and Sam Lurye
Background
Dog

Convolution + Pooling layers

Input

Fully connected layers
Convolution layer

Input

Convolution layer
Pooling layer (max pooling)
Pooling layer (max pooling)
Motivation

RuntimeError: CUDA out of memory. Tried to allocate 12.50 MiB (GPU 0; 10.92 GiB total capacity; 8.57 MiB already allocated; 9.28 GiB free; 4.68 MiB cached) #16417

CUDA Out of Memory error but CUDA memory is almost empty

I am currently training a lightweight model on very large amount of textual data (about 70GiB of text). For that I am using a machine on a cluster (‘grele’ of the grid5000 cluster network).

I am getting after 3h of training this very strange CUDA Out of Memory error message:

```
RuntimeError: CUDA out of memory. Tried to allocate 12.50 MiB (GPU 0; 10.92 GiB total capacity; 8.57 MiB already allocated; 9.28 GiB free; 4.68 MiB cached).
```

According to the message, I have the required space but it does not allocate the memory.

Any idea what might cause this?

For information, my preprocessing relies on `torch.multiprocessing.Queue` and an iterator over the lines of my source data to preprocess the data on the fly.
Unable to allocate cuda memory, when there is enough of cached memory

Can someone please explain this:

```
RuntimeError: CUDA out of memory.
Tried to allocate 350.00 MiB
(GPU 0; 7.93 GiB total capacity; 5.73 GiB already allocated;
324.56 MiB free; 1.34 GiB cached)
```

If there is 1.34 GiB cached, how can it not allocate 350.00 MiB?

There is only one process running. torch-1.0.0/cuda10

And a related question:

Are there any tools to show which python objects consume GPU RAM (besides the pytorch preloaded structures which take some 0.5GB per process) ? i.e. is there some way to query pytorch for a reference to variables that are on CUDA and perhaps from there make some deductions?

Thank you.
The stacktrace is listed below. I've changed the batch_size and sequence_length to a smaller number, and tried both gpus.
Any other personal experience?
Solutions?
Any other personal experience? Solutions?
any other pre-trained word embeddings for this. Would like to understand if you guys have a work around for this.

Hello @tsu3010 this likely happens because a mini-batch is pushed through the BERT model that requires too much GPU memory, i.e. there are too long texts in the dataset and the mini-batch size too large (see issue 549)

You could try reducing the mini-batch size from 32 to 8. You could filter or truncate long texts from the dataset to make it so a mini-batch fits into memory.
GIST

RuntimeError: CUDA out of memory. Needed to allocate 20.00 MiB (GPU 0; 15.96 GiB total capacity; 14.66 GiB already allocated; .... #685

allpetiwala opened this issue on May 2, 2019 · 14 comments

any other pre-trained word embeddings for this. Would like to understand if you guys have a work around for this.

Hello @tsu3010, unlikely happens because a mini-batch is passed through the BERT model that requires too much GPU memory, i.e. there are too long texts in the dataset and the mini-batch size too large (see issue 54).

You could try reducing the mini-batch size from 32 to 8. You could filter or remove long texts from the dataset to make it so a mini-batch fits into memory.
“Dog”
"Dog"
Figure 1
Single → Half Precision

```python
>>> x = numpy.float32(68.123)
>>> x
68.123

>>> y = numpy.float16(68.123)
>>> y
68.1
```
vDNN

CPU

GPU
How does this compare to MISTIQUE, KeystoneML?
GIST Overview
GIST Schedule Builder

Apply encodings

Static liveness analysis

CNTK static memory allocator
GIST Schedule Builder

Apply encodings

Static liveness analysis

CNTK static memory allocator
Apply encodings

- Lossless Encoding
- Lossy Encoding
GIST Schedule Builder

Apply encodings

Static liveness analysis

CNTK static memory allocator
Temporal Gap

Forward propagation

Backward propagation

Error calculation

Static liveness analysis

Memory

Forward propagation (FP32 format)

Backward propagation (FP32 format)

Temporal Gap (Encoded format)
Static liveness analysis
Static liveness analysis

CNTK static memory allocator

Stashed feature maps

Immediately consumed
Static liveness analysis

CNTK static memory allocator

Stashed feature maps

Immediately consumed

10MB

10

8
Immediately consumed

Stashed feature maps

10  8
w/o static liveness analysis

2  10
w/ static liveness analysis
Can this be implemented to avoid the extra memory usage?
Details
Max-Pooling Forward Pass

X

\[
\begin{array}{cccc}
5 & 23 & 14 & 17 \\
8 & 11 & 14 & 2 \\
0 & 0 & 4 & 1 \\
3 & 9 & 9 & 5 \\
\end{array}
\]

2x2 Max-Pool

Y

\[
\begin{array}{cc}
23 & 17 \\
9 & 9 \\
\end{array}
\]
Naive Backpropagation

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.2</td>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0.4</td>
<td>0.5</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \frac{dY}{dX} \]

2x2 Max-Pool backprop

\[
\begin{array}{cccc}
23 & 17 & & \\
9 & 9 & & \\
5 & 23 & 14 & 17 \\
8 & 11 & 14 & 2 \\
0 & 0 & 4 & 1 \\
3 & 9 & 9 & 5 \\
\end{array}
\]

\[
\begin{array}{cc}
0.2 & 0.8 \\
0.4 & 0.5 \\
\end{array}
\]
Gist Max-Pool Backpropagation

\[ \begin{align*}
\text{dX} &= \begin{bmatrix}
0 & 0.2 & 0 & 0.8 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0.4 & 0.5 & 0 \\
\end{bmatrix} \\
\text{dY} &= \begin{bmatrix}
0.2 & 0.8 \\
0.4 & 0.5 \\
1 & 1 \\
3 & 2 \\
\end{bmatrix}
\end{align*} \]

2x2 Max-Pool backprop

YToX_{map}
ReLU Standard Implementation

ReLU

<table>
<thead>
<tr>
<th>-4</th>
<th>3</th>
<th>2</th>
<th>-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>-1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>-4</td>
<td>6</td>
<td>-8</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-2</td>
<td>4</td>
</tr>
</tbody>
</table>

ReLU

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
ReLU

$y = \max(0, x)$
Binarize

ReLU

<table>
<thead>
<tr>
<th></th>
<th>-4</th>
<th>3</th>
<th>2</th>
<th>-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>-1</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-4</td>
<td>6</td>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-2</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

FP32

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Binarize

ReLU

FP32

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

1 bit

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Sparse Storage Dense Compute

Sparse conv-ReLU output

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

FP32

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Sparse Storage Dense Compute

Sparse conv-ReLU output

FP32

Non-zero values
3
3
1
2
4
7

Column indices
2
0
3
1
2
0

Non-zero values preceding the ith row
0
1
3
5

FP32
1 byte
Delayed Precision Reduction

ReLU

<table>
<thead>
<tr>
<th>-4</th>
<th>3</th>
<th>2</th>
<th>-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>-1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>-4</td>
<td>6</td>
<td>-8</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-2</td>
<td>4</td>
</tr>
</tbody>
</table>

FP32

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Delayed Precision Reduction

<table>
<thead>
<tr>
<th></th>
<th>FP32</th>
<th></th>
<th>FP16/FP10/FP8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-3</th>
<th>2</th>
<th>-8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>-4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>-2</td>
<td>4</td>
</tr>
</tbody>
</table>

ReLU
What other layers could we encode more efficiently?
Results
Experimental Setup

AlexNet

Network-in-Network

VGG16

Memory Footprint Ratio (MFR):

(memory used by baseline) / (memory used with GIST)
CNTK Baseline: does GIST use less memory overall?
Yes!

Figure 8: Evaluation of memory footprint reduction - Gist cuts down total memory footprint significantly
Figure 17: Impact of optimized hardware (dynamic allocation), Gist and optimized software (cuDNN)
Investigation Baseline: how much impact does each GIST encoding have on its target layer type?
Figure 11: *Gist* lossless encodings MFR on target categories

6% as much memory used in ReLU → Pool

14-50% as much memory used in ReLU/Pool → Conv
**Figure 10:** Impact of *Gist* lossless techniques (S - SSDC, B - Binarize, I - Inplace) on memory footprint of different data structures. Total MFR for each configuration is present at the top of each bar.
Lossy Encoding: Delayed Precision Reduction (DPR)
Figure 13: Impact of DPR encodings. Total MFR is present at the top and MFR achieved on the stashed feature maps is present at the bottom of each bar. Lowest precision for VGG16 with no loss in accuracy is FP16.

This includes the gray and green sections from the previous figure.

Immediately consumed memory increases.
**Figure 12:** Impact of DPR Encoding on accuracy. The smallest format, having same accuracy compared to FP32 format, for AlexNet and Overfeat is FP8, for Inception is FP10 and for VGG16 is FP16; DPR achieves aggressive bit savings.
Does any of this matter?
Figure 16: *Gist* enables training deeper networks with larger minibatch, achieving better performance. The largest minibatch that fits in the GPU memory is present at the bottom of the bar.
Key Takeaways

Figure 11: *Gist* lossless encodings MFR on target categories

Figure 12: Impact of DPR Encoding on accuracy. The smallest format, having same accuracy compared to FP32 format, for AlexNet and Overfeat is FP8, for Inception is FP10 and for VGG16 is FP16; DPR achieves aggressive bit savings.

Figure 16: *Gist* enables training deeper networks with larger minibatch, achieving better performance. The largest minibatch that fits in the GPU memory is present at the bottom of the bar.
Extra Discussion
Follow-up/How To?

https://pytorch.org/docs/stable/notes/cuda.html#memory-management

.half(), .float(), NVIDIA Apex

https://www.tensorflow.org/guide/gpu

https://www.tensorflow.org/guide/keras/mixed_precision