Neural Networks
Memory Footprint Reduction
during Training

Julia Gusak, Skoltech

March 2022, INRIA Bordeaux
Large scale deep learning models

- Standard way to train: stochastic gradient descent
- **For many domains**, it has been found that larger models and larger datasets give better performance
- It includes: natural language processing (NLP), self-supervised learning, vision transformers, contrastive language image pretraining, etc.

The time when you can train a large model on 1 GPU or on several GPUs is quickly going away!
Challenges in training

Two challenges:

● Computational time

● Memory for the model and batch
Some computational costs

- **CLIP model:**
  - 18 days on 592 V100 GPUs (ResNet backbone)
  - 12 days on 256 V100 GPUs

- **DALL-E model:** 1024 V100 GPU

- **VIT model:** 2500 TPU v3 core-days
Recent «world record»: Megatron-LM

<table>
<thead>
<tr>
<th>Model size</th>
<th>Hidden size</th>
<th>Number of layers</th>
<th>Number of parameters (billion)</th>
<th>Model-parallel size</th>
<th>Number of GPUs</th>
<th>Batch size</th>
<th>Achieved teraFLOPs per GPU</th>
<th>Percentage of theoretical peak FLOPs</th>
<th>Achieved aggregate petaFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7B</td>
<td>2304</td>
<td>24</td>
<td>1.7</td>
<td>1</td>
<td>32</td>
<td>512</td>
<td>137</td>
<td>44%</td>
<td>4.4</td>
</tr>
<tr>
<td>3.6B</td>
<td>3072</td>
<td>30</td>
<td>3.6</td>
<td>2</td>
<td>64</td>
<td>512</td>
<td>138</td>
<td>44%</td>
<td>8.8</td>
</tr>
<tr>
<td>7.5B</td>
<td>4096</td>
<td>36</td>
<td>7.5</td>
<td>4</td>
<td>128</td>
<td>512</td>
<td>142</td>
<td>46%</td>
<td>18.2</td>
</tr>
<tr>
<td>18B</td>
<td>6144</td>
<td>40</td>
<td>18.4</td>
<td>8</td>
<td>256</td>
<td>1024</td>
<td>135</td>
<td>43%</td>
<td>34.6</td>
</tr>
<tr>
<td>39B</td>
<td>8192</td>
<td>48</td>
<td>39.1</td>
<td>16</td>
<td>512</td>
<td>1536</td>
<td>138</td>
<td>44%</td>
<td>70.8</td>
</tr>
<tr>
<td>76B</td>
<td>10240</td>
<td>60</td>
<td>76.1</td>
<td>32</td>
<td>1024</td>
<td>1792</td>
<td>140</td>
<td>45%</td>
<td>143.8</td>
</tr>
<tr>
<td>145B</td>
<td>12288</td>
<td>80</td>
<td>145.6</td>
<td>64</td>
<td>1536</td>
<td>2304</td>
<td>148</td>
<td>47%</td>
<td>227.1</td>
</tr>
<tr>
<td>310B</td>
<td>16384</td>
<td>96</td>
<td>310.1</td>
<td>128</td>
<td>1920</td>
<td>2160</td>
<td>155</td>
<td>50%</td>
<td>297.4</td>
</tr>
<tr>
<td>530B</td>
<td>20480</td>
<td>105</td>
<td>529.6</td>
<td>280</td>
<td>2520</td>
<td>2520</td>
<td>163</td>
<td>52%</td>
<td>410.2</td>
</tr>
<tr>
<td>1T</td>
<td>25600</td>
<td>128</td>
<td>1008.0</td>
<td>512</td>
<td>3072</td>
<td>3072</td>
<td>163</td>
<td>52%</td>
<td>502.0</td>
</tr>
</tbody>
</table>

How to train a big model?

- We have: the model and the data.
- We train using stochastic gradient descent (SGD)
- Given a batch of size
- We compute: forward, backward

```python
model = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = Variable(data), Variable(target)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
```
What is computed

For a backward pass, we need to store activations! They consume 0.1 - 10x of the memory of the model (depending on the batch size)
## Methods to Train Large Neural Networks

<table>
<thead>
<tr>
<th>Method</th>
<th># of GPUs</th>
<th>Approx. computations</th>
<th>Communication costs per iteration activation values / weight values / activation grads / weight grads</th>
<th>Batch size per GPU increase?</th>
<th># of FLOP per iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>No data parallelism</td>
<td>1</td>
<td>X</td>
<td>baseline</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Rematerialization</td>
<td>≥ 1</td>
<td></td>
<td>= / = = / =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offloading:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>activations</td>
<td>≥ 1</td>
<td>X</td>
<td>↑ / = = / =</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>weights</td>
<td>≥ 1</td>
<td>X</td>
<td>= / ↑ = / ↑ or =</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>tensors in GPU cache</td>
<td>≥ 1</td>
<td>X</td>
<td>↑ or = / ↑ or =</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Approx. gradients:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lower-bit activation grad.**</td>
<td>≥ 1</td>
<td>✓</td>
<td>↓ / = = / =</td>
<td>✓</td>
<td>↓ or =</td>
</tr>
<tr>
<td>approx. matmul**</td>
<td>≥ 1</td>
<td>✓</td>
<td>↓ / = = / =</td>
<td>✓</td>
<td>↑</td>
</tr>
<tr>
<td>lower-bit weight grad**</td>
<td>≥ 1</td>
<td>✓</td>
<td>= / = = / ↓</td>
<td>✓</td>
<td>↓ or =</td>
</tr>
<tr>
<td>Data parallelism*</td>
<td>&gt; 1</td>
<td>X</td>
<td>baseline</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Partitioning:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>optim. state</td>
<td>&gt; 1</td>
<td>X</td>
<td>= / ↑ = / =</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>+ gradients</td>
<td>&gt; 1</td>
<td>X</td>
<td>= / ↑ = / =</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>+ parameters</td>
<td>&gt; 1</td>
<td>X</td>
<td>= / ↑ = / =</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Model parallelism*</td>
<td>&gt; 1</td>
<td>✓</td>
<td>↑ / = ↑ / ↓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Pipeline parallelism</td>
<td>&gt; 1</td>
<td>✓</td>
<td>↑ / = ↑ / ‡</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

https://arxiv.org/abs/2202.10435
Types of Parallelism
Data Parallelism

Data parallelism:
+ speeds up training
- weights and gradients must fit on the same device
Data Parallelism using ZeRO:

+ you can train models that do not fit on one device
- increases the number of transfers between devices
Model Parallelism

- you can train models that do not fit on one device
- bad GPU utilization: the device waits for the output of the previous layer of the model

To reduce GPU idle time, several approaches have been developed to organize a data pipeline between devices: GPipe, Megatron-LM, Varuna.

(a) Varuna Schedule

(b) Gpipe Schedule
Model Parallelism: on Tensor Level

Tensor parallelism for Transformer:

(a) MLP

(b) Self-Attention
Pipeline Parallelism

**GPipe**
batches are divided into micro-batches to reduce downtime

Interleaved Pipeline: Varuna, SageMaker, DeepSpeed

Backward for the first micro-batch is computed earlier than forward for the second micro-batch
3D Parallelism: PP+TP+DP(ZeRO)

32 GPUs are used: 4 groups tensor-parallelism * 4 groups pipeline-parallelism * 2 groups data-parallelism

MP-\(n\) denotes tensor-parallelism
Optimal Strategies for Pipeline Parallelism

<table>
<thead>
<tr>
<th>Paper</th>
<th>FlexFlow</th>
<th>PipeDream</th>
<th>PipeDream-2BM</th>
<th>Piper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types of parallelism</td>
<td>tensor-, model-, data-, operator-</td>
<td>data-, pipeline-</td>
<td>data-, pipeline-</td>
<td>data-, tensor-, pipeline-</td>
</tr>
<tr>
<td>Method for hyperparameters optimization</td>
<td>MCMC</td>
<td>Dynamic Programming</td>
<td>Dynamic Programming</td>
<td>Dynamic Programming</td>
</tr>
<tr>
<td>Activation recomputation during gradient computation (Activation checkpointing)</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Activation offloading to CPU</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Application</td>
<td>CNNs</td>
<td>CNNs</td>
<td>Transformers</td>
<td>Transformers</td>
</tr>
</tbody>
</table>

None of the known approaches use:
- activation offloading to the CPU, or
- a combination of two methods, recomputation of activations when computing gradients and offloading activations to CPU.
Activation Checkpointing & Offloading to CPU
Computational Graph for GPT-2

Example: transformer block from GPT-2 model contains:
1. Layer normalizations
2. Linear layers
3. Attention layer
4. GELU activation
GPT-2 Profiling: Memory

Memory in GB required to store all activations, depending on the batch size and the length of the token sequence.

The memory limit of one GPU V100-16GB is highlighted in red.
Methods to Reduce Activations Memory: Rotor

- Saving only part of activations in the forward pass and recomputing the rest during gradients computation;

- Sends activations to CPU and loading from CPU as needed to calculate gradients;

+ saves memory
- slows down training: when computing gradients, you have to recompute activations

Sequence

\[ F_1^c, F_2^n, F_3^n, F_4^e, F_5^c, F_6^e, \text{LOSS}, B_{\text{LOSS}}, B_6, B_5, B_4, F_1^c, F_2^n, F_3^e, B_3, F_1^e, F_2^e, B_2, B_1 \]
Optimization Methods
Optimization of Large Scale Models

Problem:

\[
\frac{1}{N} \sum_{i=1}^{N} L(\Phi(x_i, w), y_i) \rightarrow \min_w
\]

batch size  \rightarrow \text{loss function}  \rightarrow \text{training data}  \rightarrow \text{model weights}

Algorithm:

\[ w_{t+1} = w_t + \alpha h_t, \]

step size

(\text{SGD}) \quad \[ h_t = g_t = \frac{1}{N_b} \sum_{(x_i, y_i) \in X_b} \frac{\partial L(\Phi(x_i, w), y_i)}{\partial w} \bigg|_{w=w_t} \]

batch size  \rightarrow \text{batch size}  \rightarrow \text{batch size}

Questions:

- How to choose step size?
- How to store vectors so they occupy less space?
- How to choose batch size?
- How to use many devices to speed up optimization?
- How to initialize weights?
Using batches of bigger sizes

- Training a model with a large batch takes less time due to parallelism.
- However, with a simple increase in the batch, the generalizing ability of the model is worse.
- When the batch size increases by $k$ times, the step size must be increased by $k$ times.
- Increasing the step should be carried out gradually (warmup - phase of the first few epochs).
- Layer-by-layer step size change allows you to increase the batch even more.
Optimization Methods

Using low-bit formats for data storage

- The use of floating point numbers and block quantization are essential.
- The bitsandbytes library from Facebook contains 8-bit optimizers
Optimization Methods

Distributed training and federated learning

- By using a large number of parallel computers, you can increase the batch and speed up training.
- Communications can be optimized by transmitting low-rank representations of gradients (PowerSGD and GradZIP methods); sparsification of gradients (Sketched SGD) or quantization of gradients.
- It is possible to do multiple local gradient descent steps on the GPU before forwarding to avoid local minima (post-local SGD).
Approximate Activation Gradients: Few-bit Backward

![Approximate Activation Gradients](image.png)

*Figure 1. Optimized 3-bit piecewise-constant approximation of the derivative of the GELU activation function.*
Approximate Activation Gradients: Few-bit Backward

Forward pass

Input $X_t$

$\text{Intermediate tensors for Gradient } f$ computation

$x = (0.55, 0.47, 0.12, 1.12, 0.83)$

$x_{\text{float}} = (0.5, 0.5, 0, 1, 1)$

$x_{\text{bit}} = (01, 01, 00, 10, 10)$

Quantize and Save

Tensors saved for backward

Backward pass

$\frac{\partial L}{\partial X_t}$

$\frac{\partial L}{\partial X_{t+1}}$

Gradient $f$

# Globally stored
# piecewise-constant
# approximation parameters
s = [...]  
y = [...]

def forward(X):
    X_pos = sortedsearch(s, X)
    save_for_backward(X_pos)
    return f(X)

def backward(dLdY):
    X_pos = get_saved_for_backward()
    return dLdY * y[X_pos]
Approximate Activation Gradients: Few-bit Backward
Approximate matrix multiplication: Randomized Backward
Approximate matrix multiplication: Randomized Backward

Algorithm 1 Forward and backward pass through a linear layer with a randomized matrix multiplication.

function FORWARD($X$, $W$, $b$)
\[
\hat{X} \leftarrow XW^T + 1_B b^T
\]
Generate pseudo random number generator (PRNG) state and corresponding random matrix $S$

$X_{\text{proj}} \leftarrow S^T X$

Save $X_{\text{proj}}$ and PRNG state for the backward pass.

return $Y$
end function

function BACKWARD($\partial_{\hat{X}} \mathcal{L}$, $W$, $b$, $X_{\text{proj}}$)
\[
\partial_X \mathcal{L} \leftarrow \partial_{\hat{X}} \mathcal{L} \cdot W^T
\]
Rematerialize matrix $S$ from the PRNG state saved in the forward pass.

$\partial_W \mathcal{L} \leftarrow (\partial_{\hat{X}} \mathcal{L}^T \cdot S) \cdot X_{\text{proj}}$

$\partial_b \mathcal{L} \leftarrow \partial_{\hat{X}} \mathcal{L}^T 1_B$

return $\partial_X \mathcal{L}$, $\partial_W \mathcal{L}$, $\partial_b \mathcal{L}$
end function
**Approximate gradients**

<table>
<thead>
<tr>
<th>Task</th>
<th>Batch Size</th>
<th>GELU</th>
<th>Linear Layer</th>
<th>Peak Memory, GiB</th>
<th>Saving, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MRPC</td>
<td>128</td>
<td>Vanilla</td>
<td>11.30</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>MRPC</td>
<td>128</td>
<td>3-bit</td>
<td>9.75</td>
<td>13.8</td>
</tr>
<tr>
<td>3</td>
<td>MRPC</td>
<td>128</td>
<td>Vanilla</td>
<td>9.20</td>
<td>18.6</td>
</tr>
<tr>
<td>4</td>
<td>MRPC</td>
<td>128</td>
<td>3-bit</td>
<td>7.60</td>
<td>32.7</td>
</tr>
</tbody>
</table>