Distributed Optimization for Machine Learning

Lecture 20 - Memory footprint of GPT and mixed precision training

ECE 5290/7290 & ORIE 5290

Tianyi Chen

School of Electrical and Computer Engineering
Cornell Tech, Cornell University

November 10, 2025

Table of Contents

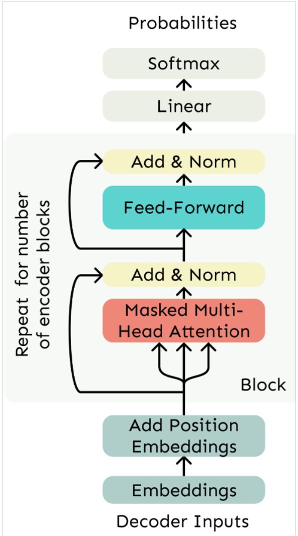
Parameter analysis of GPT

Memory analysis of GPT

Mixed precision training

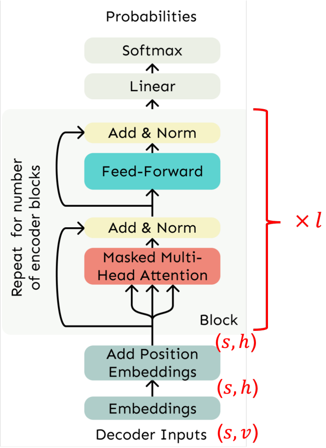
GPT: Generative pre-trained transformer

- A generative pre-trained transformer (GPT) is a type of LLMs.
- GPT is based on the decoder-only transformer.
- Each block consists of:
  - Self-attention
  - Add & Norm
  - Feed-forward
  - Add & Norm
- We will analyze the parameters, memories, and computation costs for decoder-only transformer.



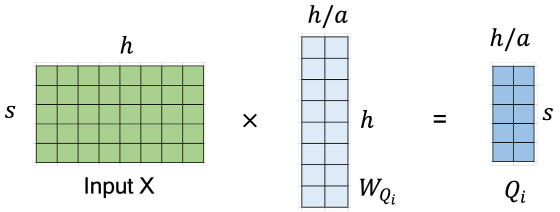
Notations for GPT

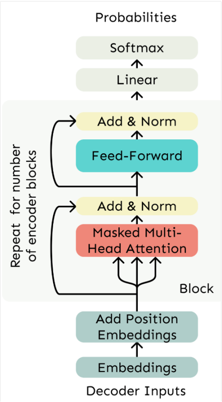
- Number of the transformer layers: ℓ
- Sequence length: s
- Vocabulary size: v
- Embedding representation dims: h



Multi-head self-attention computations

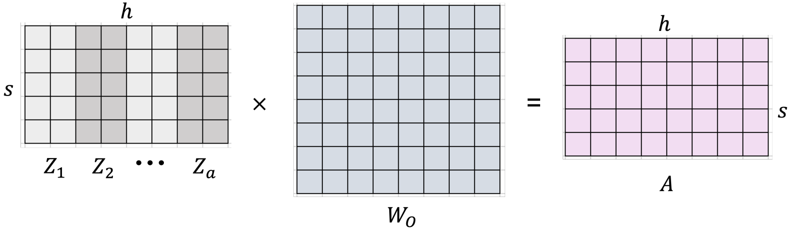
- Number of heads: a
- Dims of each W_{Q_i}, W_{K_i} and W_{V_i}: $h \times \frac{h}{a}$





Multi-head self-attention memory

- Number of heads: a
- Dims of each W_O: $h \times h$
- Dims of each W_{Q_i}, W_{K_i} and W_{V_i}: $h \times \frac{h}{a}$



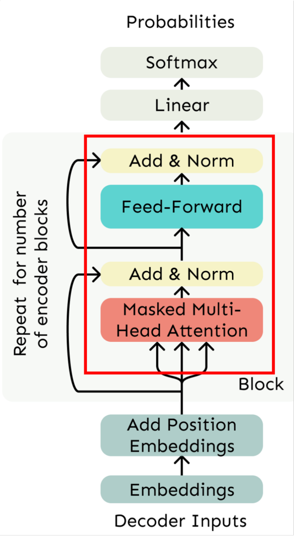
We need to store $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ and $\mathbf{W}_O$, which is in total $4h^2$

$$3 \times \frac{h^2}{\alpha} \times \alpha = 3h^2 \quad \text{and} \quad h^2$$

Feed-forward layer memory

$$X' = \mathrm{ReLU}(A \cdot W_1 + b_1) \cdot W_2 + b_2$$

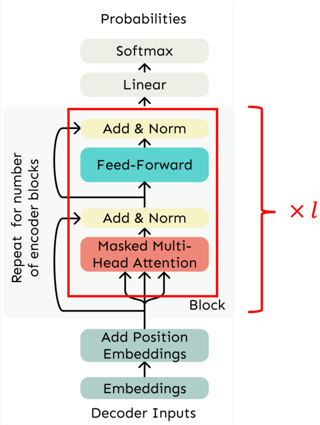
- Dims of W_1: $h \times 4h$
- Dims of each W_2: $4h \times h$
- We need to store W_1 and W_2 : $8h^2$
- The storage of b_1 and b_2 can be ignored



Transformer block memory

- Multi-head attentions: $4h^2$
- Feed-forward layers: $8h^2$
- ℓ layers of attentions:

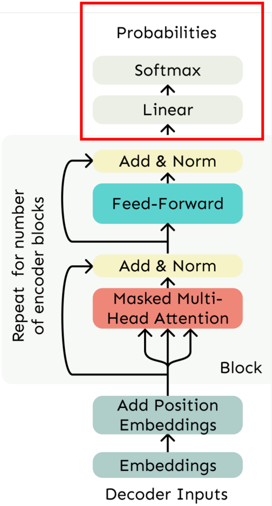
$$(4h^2 + 8h^2) \times \ell = 12\ell h^2$$



Probability predictions memory

$$p = \mathrm{Softmax}(X \cdot \mathbf{W}_V + b_v)$$

- Dimension of $\mathbf{W}_V$: $h \times v$
- We need to store $\mathbf{W}_V$: hv parameters
- b_V can be ignored

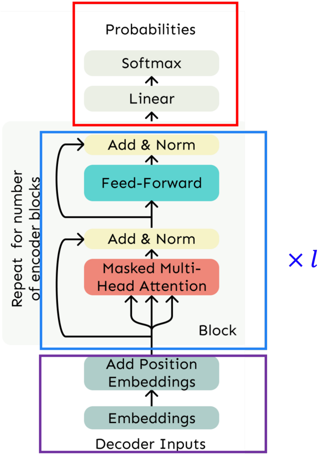


Total number of parameters in GPT

- Embeddings: vh
- Attention blocks: $12\ell h^2$
- Probability predictions: vh

Total parameters:

$$12\ell h^2 + 2vh$$



Example: LLaMA parameters

- Now we compare our theoretical evaluations with the LLaMA model.
- $12\ell h^2 + 2vh$ is a very accurate estimation.

Actual params	**Embedding h**	**Attention layers ℓ**	**Vocab size v**	**Estimated params**
6.7B	4096	32	32000	6,704,594,944
13.0B	5120	40	32000	12,910,592,000
32.5B	6656	60	32000	32,323,665,920
65.2B	8192	80	32000	64,948,797,440

Table: Comparison of actual and estimated parameters for LLaMA models

Table of Contents

Parameter analysis of GPT

Memory analysis of GPT

Mixed precision training

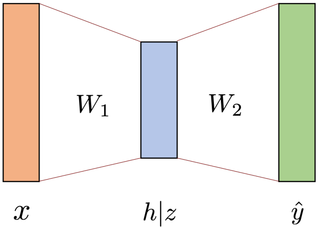
From parameters to memory usage

- We now know how to compute the **total number of parameters**:

$$N_{\text{params}} = 12\ell h^2 + 2vh$$

- But model **size alone doesn't tell the full story.**
- During training or inference, we must also consider:
  - Parameter storage (weights)
  - Optimizer states and gradients
  - Intermediate activation functions
  - KV cache for attention
- These determine the **true memory footprint** of a model.

Warmup: Linear neural network with batch gradient



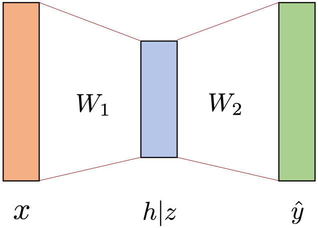
dims:	d	input dimension
	(p, d)	W_1
	p	hidden dimension
	(q, p)	W_2
	q	output dimension

Forward:

$$
\begin{aligned}
h_b &= W_1 x_b, \\
z_b &= \sigma(h_b), \\
y_b &= W_2 z_b, \\
f &= \frac{1}{B} \sum_{b=1}^{B} \mathcal{L}(y_b)
\end{aligned}
$$

Store $\{h_b, z_b, y_b\}_{b=1}^{B}$ during forward pass

Warmup: Linear neural network with batch gradient



dims:		
	d	input dimension
	(p, d)	W_1
	p	hidden dimension
	(q, p)	W_2
	q	output dimension

Backward:

$$\frac{\partial f}{\partial W_2} = \frac{1}{B}\sum_{b=1}^{B} \frac{\partial \mathcal{L}}{\partial y_b} z_b^\top,$$

$$\frac{\partial f}{\partial z_b} = W_2^\top \frac{\partial \mathcal{L}}{\partial y_b},$$

$$\frac{\partial f}{\partial h_b} = \frac{\partial f}{\partial z_b} \odot \sigma'(h_b),$$

$$\frac{\partial f}{\partial W_1} = \frac{1}{B}\sum_{b=1}^{B} \frac{\partial f}{\partial h_b} x_b^\top$$

Store $\nabla_{W_1} f(W_1)$ and $\nabla_{W_2} f(W_2)$ during backward pass

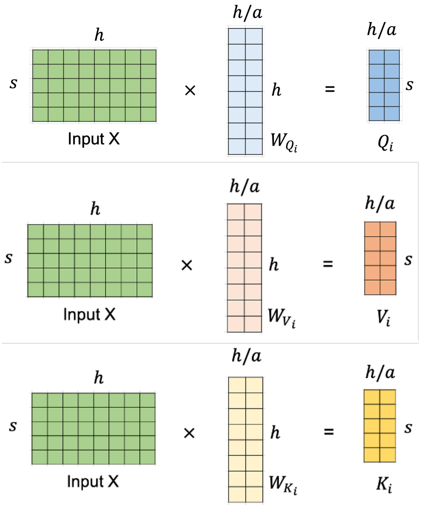
Memory decomposition of training LLMs

$$\textbf{Memory} = \underbrace{\text{Model}}_{\text{weights}} + \underbrace{\text{Gradients}}_{\text{backprop}} + \underbrace{\text{Optimizer States}}_{\text{e.g., Adam m,v}} + \underbrace{\text{Activations}}_{\text{forward/backward pass}}$$

Model Parameters P	Gradients P	Optimizer States $2P$	Activations (?)

Typical memory usage breakdown in training large models.

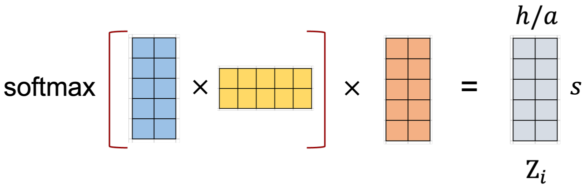
Activations in multi-head self-attention



Store α copies of $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$

$$3 \times \frac{sh}{\alpha} \times \alpha = 3sh$$

Activations in multi-head self-attention



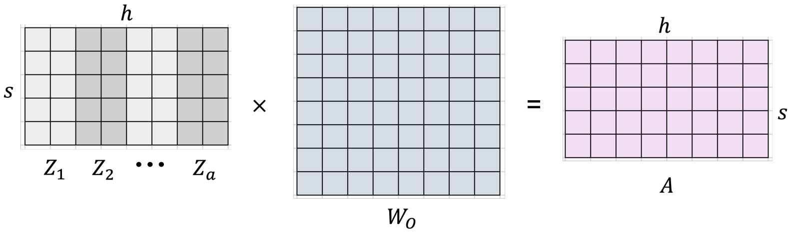
- Given $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i \in \mathbb{R}^{s \times h/\alpha}$, one head self-attention is

$$\mathrm{Attn}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) \;=\; \mathrm{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{h/a}}\right) \mathbf{V}_i$$

- Store $\mathbf{Q}_i \mathbf{K}_i^\top$ with s^2 parameters;
- Store $\mathrm{softmax}(\mathbf{Q}_i \mathbf{K}_i^\top)$ with s^2 parameters;
- Store $\mathrm{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{h/a}}\right) \mathbf{V}_i$ with sh/a parameters;

Activations in multi-head self-attention

- Number of heads: a
- Dims of each W_O: $h \times h$
- Dims of each W_{Q_i}, W_{K_i} and W_{V_i}: $h \times \frac{h}{a}$

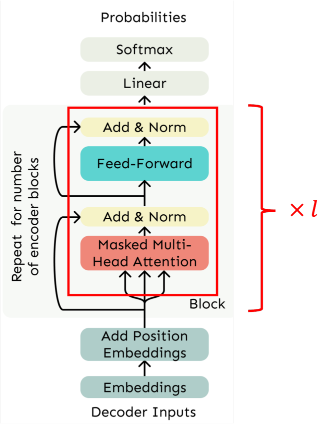


We need to store **A**, which is in total *sh*

Activations in Transformer block

- Multi-head attentions: $5sh + 2s^2a$
- Feed-forward layers: $9sh$
- ℓ layers of attentions:

$$(2s^2a + 14sh) \times \ell$$

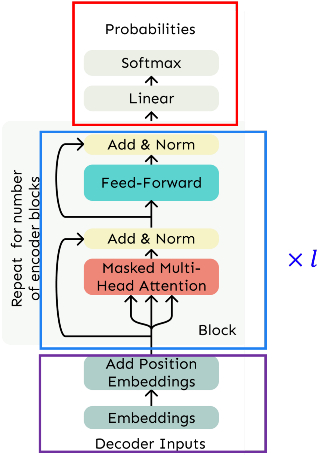


Total number of activations in GPT

- Number of the transformer layers: ℓ
- Sequence length: s
- Embedding representation dims: h
- Embedding activations: sh
- Self-attention activations: $(2s^2\alpha + 14sh) \times \ell$
- Probability activations: $2sv$

Total parameters with batch size b:

$$(2s^2a + 14sh) \times \ell \times b$$



Total memory of training LLMs

$$\textbf{Memory} = \underbrace{\text{Model}}_{\text{weights}} + \underbrace{\text{Gradients}}_{\text{backprop}} + \underbrace{\text{Optimizer States}}_{\text{e.g., Adam m,v}} + \underbrace{\text{Activations}}_{\text{forward/backward pass}}$$

Model Parameters P	Gradients P	Optimizer States $2P$	Activations (?)

$$\left(48\ell h^2 \quad + \quad b\ell(2s^2a + 14sh)\right) \quad \times \quad 4 \text{ Bytes (32 bits)}$$

- When hidden state h is large, the model dominates the memory
- If batch-size b or sequence length s is large, the activation dominates
- The activation-incurred memory cannot be ignored

Memory estimation for GPT-3

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

- GPT-3 has 175B parameters; its model consumes

$$4 \times 175 \times 10^9 \text{ Bytes} = 700\,\text{GB}$$

- Its gradients take 700 GB; optimizer states take 1.4 TB.
- GPT has sequence length $s = 2048$. When $b = 1$, its activation takes 444 GB (63% of the model).
- When $b = 128$, its activation is 81 times the model size.

Table of Contents

Parameter analysis of GPT

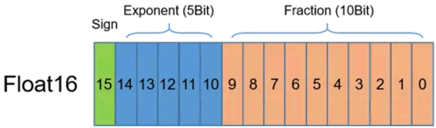
Memory analysis of GPT

Mixed precision training

Full precision training and low precision training

- Full precision training (e.g., FP32)
  - each parameter takes 4 Bytes
  - used in training most DNNs; very precise
  - takes a lot of computations and memories
- Low precision training (e.g., FP16)
  - each parameter takes 2 Bytes
  - train larger models due to computational and memory efficiency
  - not precise enough; overflow and underflow occur occasionally

Float 16 (FP16) precision



- **Sign**: 1 bit; 0 for positive and 1 for negative
- **Exponent**: 5 bits; range: 00001(1)-11110(30); value: $2^{-14} \sim 2^{15}$

$$\text{Example:} \quad 00111(7) \quad \longrightarrow \quad 2^{7-15} = 2^{-8}$$

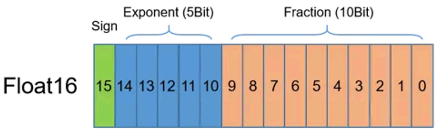
where -15 is the offset

- **Fraction**: 10 bits;

$$\begin{aligned} \text{Example:} \quad 1001000000 \quad &\longrightarrow \quad 1.1001000000 \\ &\longrightarrow \quad 1 + 576/1024 = 1.5625 \end{aligned}$$

where binary 1001000000 translates into decimal 576

Float 16 (FP16) precision



- **Translation law**

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-15} \times (1 + \frac{\text{fraction}}{1024})$$

- **Largest positive number**:

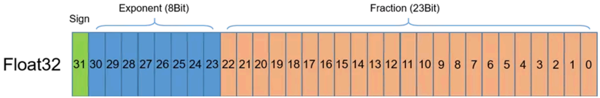
$$(-1)^0 \times 2^{15} \times (1 + \frac{1023}{1024}) = 65504$$

The range of FP16 is $[-65504, +65504]$.

- **Smallest positive number**:

$$(-1)^0 \times 2^{-14} \times (0 + \frac{1}{1024}) \approx 2^{-24}$$

Float 32 (FP32) precision



- **Sign**: 1 bit; 0 for positive and 1 for negative
- **Exponent**: 8 bits; range: 00000001(1) - 11111110(254);
  Bias: 127, hence the exponent value is $2^{-126} \sim 2^{127}$.

$$\text{Example:} \quad 01111111(127) \quad \longrightarrow \quad 2^{127-127} = 2^0 = 1$$

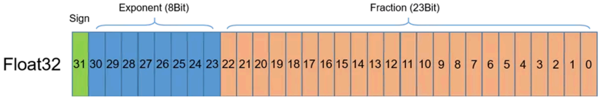
- **Fraction**: 23 bits;

$$\text{Example:} \quad 10010000000000000000000 \quad \rightarrow \quad 1.10010000000000000000000$$

$$\rightarrow \quad 1 + \frac{10010000000000000000000}{2^{23}} = 1 + \frac{4718592}{8388608} = 1.5625$$

where binary 10010000000000000000000 translates into 4718592.

Float 32 (FP32) precision



- **Translation law**

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times (1 + \frac{\text{fraction}}{2^{23}})$$

- **Range**: $[-3.40282 \times 10^{38}, +3.40282 \times 10^{38}]$

- **Smallest positive number**: 1.17549×10^{-38}

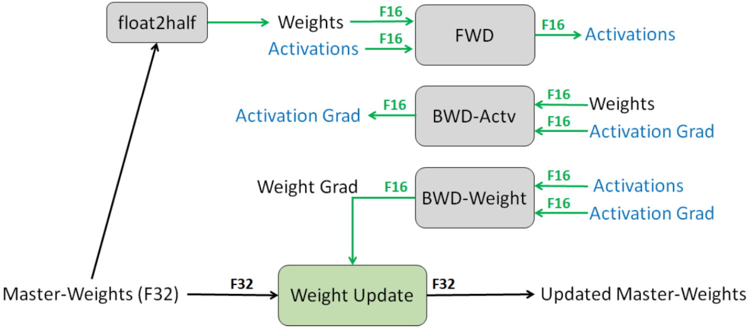
- FP 32 is much more powerful than FP 16; but takes more memory

Mixed precision training

- When both FP32 and FP16 are used in training, we get **Mixed precision training**; see (`micikevicius2017mixed`)
- Save memory and computations without hurting performance
- Three key techniques:
  - FP32 weight copies
  - Loss scaling
  - Arithmetic precision

Technique I - FP32 weight copies

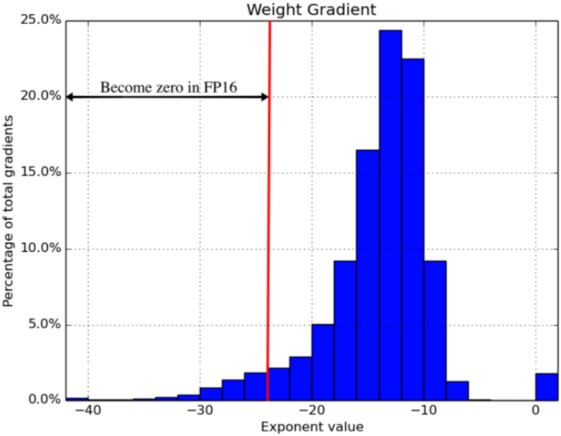
- An FP32 weight copy is maintained and updated with gradient



Technique I - FP32 weight copies

- Reason I: **maintain small values in the weight update**
- Weight update = learning rate $\times$ gradient; typically very small in late phase
- Values less than $2^{-24} \approx 5.96 \times 10^{-8}$ become 0 when using FP16
- About 5% values are less than 2^{-24}

FP32 weight copies



Technique I - FP32 weight copies

- Reason II: **big value-to-update ratio**
- The resolution in each period is shown as follows[1]

Min	Max	interval
0	2^{-13}	2^{-24}
2^{-13}	2^{-12}	2^{-23}
2^{-12}	2^{-11}	2^{-22}
2^{-11}	2^{-10}	2^{-21}
2^{-10}	2^{-9}	2^{-20}
2^{-9}	2^{-8}	2^{-19}
2^{-8}	2^{-7}	2^{-18}
2^{-7}	2^{-6}	2^{-17}
2^{-6}	2^{-5}	2^{-16}
2^{-5}	2^{-4}	2^{-15}
2^{-4}	$\frac{1}{8}$	2^{-14}
$\frac{1}{8}$	$\frac{1}{4}$	2^{-13}
$\frac{1}{4}$	$\frac{1}{2}$	2^{-12}
$\frac{1}{2}$	1	2^{-11}
1	2	2^{-10}
2	4	2^{-9}
4	8	2^{-8}
8	16	2^{-7}
16	32	2^{-6}
32	64	2^{-5}
64	128	2^{-4}
128	256	$\frac{1}{8}$
256	512	$\frac{1}{4}$
512	1024	$\frac{1}{2}$
1024	2048	1
2048	4096	2
4096	8192	4
8192	16384	8
16384	32768	16
32768	65519	32
65519	∞	∞

[1]This figure is from wikipedia

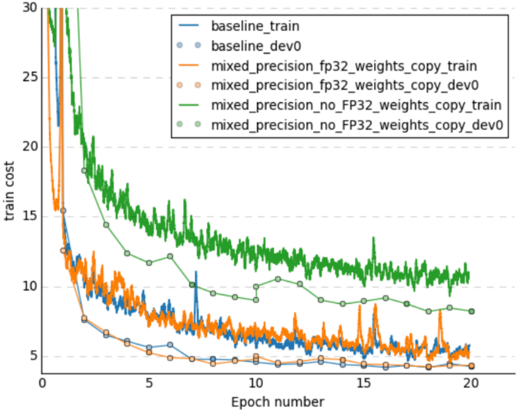
Technique I - FP32 weight copies

- If the value-to-update ratio is bigger than $2^{11} = 2048$, it holds that

$$\text{value} + \text{update} = \text{value}$$

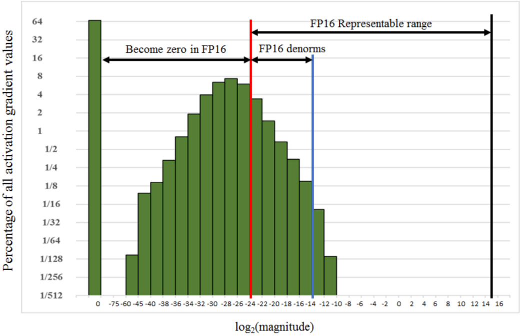
- The update has no influence on the value
- For reasons I and II, we maintain FP32 copies for both the weight and weight decay

Technique I - FP32 weight copies



Technique II - Loss scaling

- FP16 representation range $[2^{-24}, 2^{15}]$
- The activation gradient is typically very small; most values are smaller than 2^{-24}



Technique II - Loss scaling

- Scale-up the loss value before the back-propagation
- Unscale the gradient after back-propagation but before the update

$$g(x) = \frac{\partial L}{\partial x} = \frac{1}{c}\frac{\partial (c \cdot L)}{\partial x}$$

- Effectively shift the gradient value to the FP representation range
- Tricky to choose the scale-up coefficient
- $c = 8$ typically works

Technique II - Loss scaling

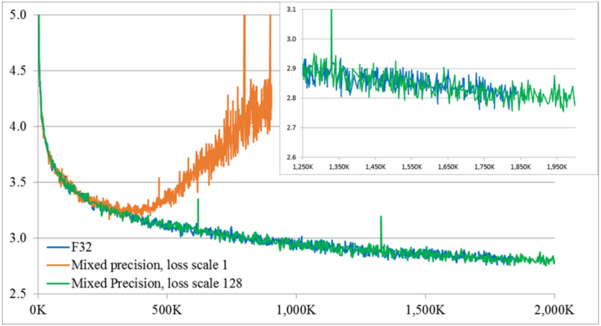


Figure: bigLSTM training perplexity.

- Mixed-precision training with loss scale 128 (green) closely matches full-precision (F32, blue) performance.
- Without appropriate loss scaling (loss scale 1, orange), training diverges due to numerical underflow in gradients.

Technique III - Arithmetic precision

- Not as important as the above two techniques
- Three key computation steps: vector dot-products; reductions; point-wise operations
- It is suggested in (`micikevicius2017mixed`) that vector dot-products and reductions are read and written in FP16 but carried out in FP32
- Point-wise operations can be carried in FP16

Numerical studies

Table 1: ILSVRC12 classification top-1 accuracy.

Model	Baseline	Mixed Precision	Reference
AlexNet	56.77%	56.93%	(Krizhevsky et al., 2012)
VGG-D	65.40%	65.43%	(Simonyan and Zisserman, 2014)
GoogLeNet (Inception v1)	68.33%	68.43%	(Szegedy et al., 2015)
Inception v2	70.03%	70.02%	(Ioffe and Szegedy, 2015)
Inception v3	73.85%	74.13%	(Szegedy et al., 2016)
Resnet50	75.92%	76.04%	(He et al., 2016b)

Table 2: Detection network average mean precision.

Model	Baseline	MP without loss-scale	MP with loss-scale
Faster R-CNN	69.1%	68.6%	69.7%
Multibox SSD	76.9%	diverges	77.1%

- Across various architectures, **mixed precision training** achieves accuracy comparable to or slightly higher than full precision.
- For detection networks, **loss scaling is essential** — without it, training may diverge (e.g., Multibox SSD).

Nvidia AMP[2]

AMP FOR PYTORCH

As simple as two lines of code

Wrap the model and optimizer

```
model, optimizer = amp.initialize(model, optimizer)
```

Apply automatic loss scaling and backpropagate with scaled loss

```
with amp.scaled_loss(loss, optimizer) as scaled_loss:
    scaled_loss.backward()
```

[2] https://nvlabs.github.io/iccv2019-mixed-precision-tutorial/

Nvidia AMP[3]: An example

```
import torch
import amp
model = ...
optimizer = ...
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")
for data, label in data_iter:
    out = model(data)
    loss = criterion(out, label)
    optimizer.zero_grad()
    with amp.scaled_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
optimizer.step()
```

allows AMP to perform automatic casting

replaces `loss.backward()`

[3] https://nvlabs.github.io/iccv2019-mixed-precision-tutorial/

Case study: Mixed-precision Adam

- Adam is an optimization method widely-used in training LLMs
- Adam states use $33\% \sim 75\%$ memories
- For example, Adam states use 11GB for GPT2 and 41GB for T5
- It is urgent to reduce the memory footprint caused by Adam states

Case study: 8-bit Adam optimizer

- Recall the Adam optimizer

$$
\begin{aligned}
g_k &= \nabla F(x_k; \xi_k) \\
m_k &= \beta_1 m_{k-1} + (1-\beta_1) g_k \\
s_k &= \beta_2 s_{k-1} + (1-\beta_2) g_k \odot g_k \\
x_{k+1} &= x_k - \frac{\gamma}{\sqrt{s_k} + \epsilon} \odot m_k
\end{aligned}
$$

- 8-bit only supports $2^8 = 256$ values; much less than FP16 and FP32

- (`dettmers2021`) develops 8-bit Adam optimizer with block-wise quantization and dynamic quantization

8-bit optimizer procedure

- Quantize and store Adam states with 8 bit
- When updating and using state, dequantize it to FP32, update the weight, and quantize back to 8 bit
- 8-bit to 32-bit conversion element-by-element in registers; no additional temporary memory

Performance for common benchmarks

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.

Enable training larger models

	Largest finetunable Model (parameters)	
GPU size in GB	32-bit Adam	8-bit Adam
6	RoBERTa-base (110M)	RoBERTa-large (355M)
11	MT5-small (300M)	MT5-base (580M)
24	MT5-base (580M)	MT5-large (1.2B)
24	GPT-2-medium (762M)	GPT-2-large (1.5B)

- **Observation:** Using 8-bit Adam enables fine-tuning of models up to **three times larger** on the same GPU memory.
- This highlights the **scalability benefits of memory-efficient optimizers**, allowing larger models to be trained on same hardware.

Recap and fine-tuning

- What we have talked about today?
  - ⇒ What is the number of parameters in GPTs?
  - ⇒ What is the memory complexity in GPTs
  - ⇒ How to perform memory-efficient training of GPTs?

Welcome anonymous survey!