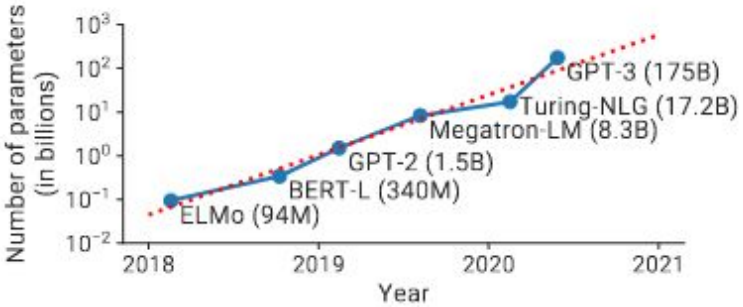
Neural Networks Memory Footprint Reduction during Training

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
- **DALLE-E model**: 1024 V100 GPU
- **VIT model**: 2500 TPU v3 core-days

Recent «world record»: Megatron-LM

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

https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/

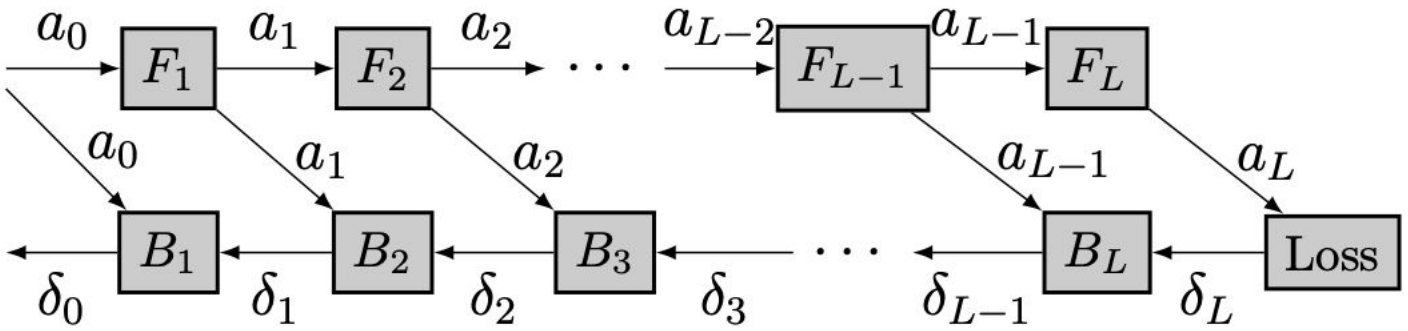
How to train a big model?

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

```
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)



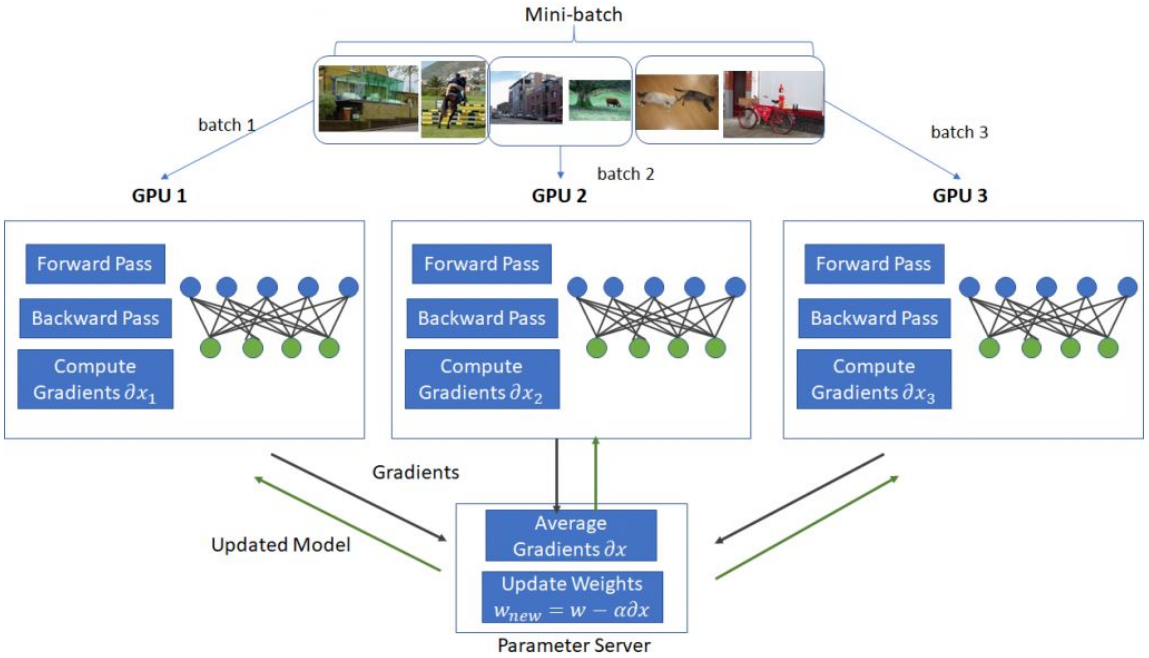
Methods to Train Large Neural Networks

Method	# of GPUs	Approx. computations	Communication costs per iteration activation values / weight values / activation grads / weight grads	Batch size per GPU increase?	# of FLOP per iteration
No data parallelism	1	baseline	baseline	baseline	baseline
Rematerialization	≥ 1	✗	= / = = / =	✓	↑
Offloading:					
activations	≥ 1	✗	↑ / = = / =	✓	=
weights	≥ 1	✗	= / ↑ = / ↑ or =	✓	=
tensors in GPU cache	≥ 1	✗	↑ or = / ↑ or = = / ↑ or =	✓	=
Approx. gradients:					
lower-bit activation grad.**	≥ 1	✓	↓ / = = / =	✓	↓ or =
approx. matmul**	≥ 1	✓	↓ / = = / =	✓	↕
lower-bit weight grad**	≥ 1	✓	= / = = / ↓	✓	↓ or =
Data parallelism*	> 1	✗	baseline	✗	=
Partitioning:					
optim. state	> 1	✗	= / ↑ = / =	✓	=
+ gradients	> 1	✗	= / ↑ = / =	✓	=
+ parameters	> 1	✗	= / ↑ = / =	✓	=
Model parallelism*	> 1	✗	↑ / = ↑ / ↓	✓	=
Pipeline parallelism	> 1	✓/✗	↑ / = ↑ / ↕	✓	=

https://arxiv.org/abs/2202.10435

Types of Parallelism

Data Parallelism



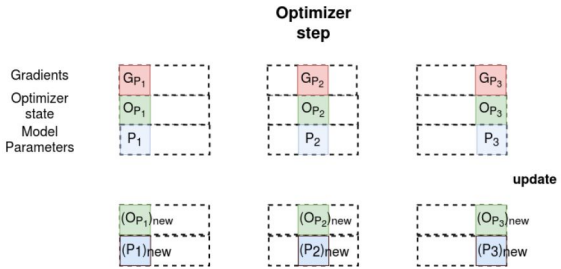
Data parallelism:

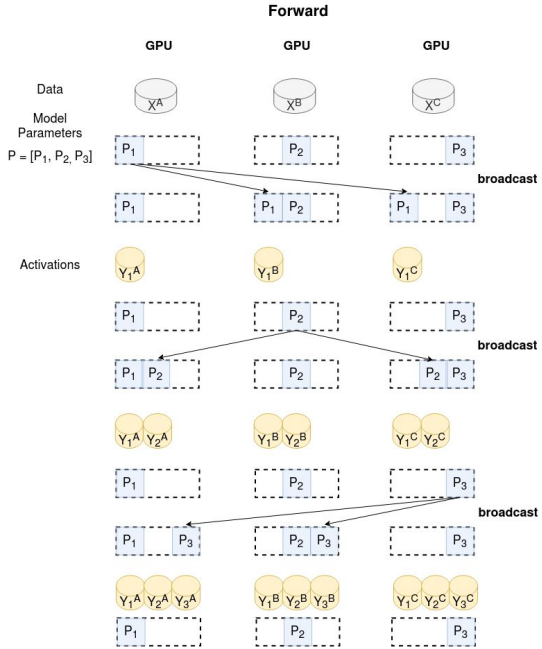
+ speeds up training
- weights and gradients must fit on the same device

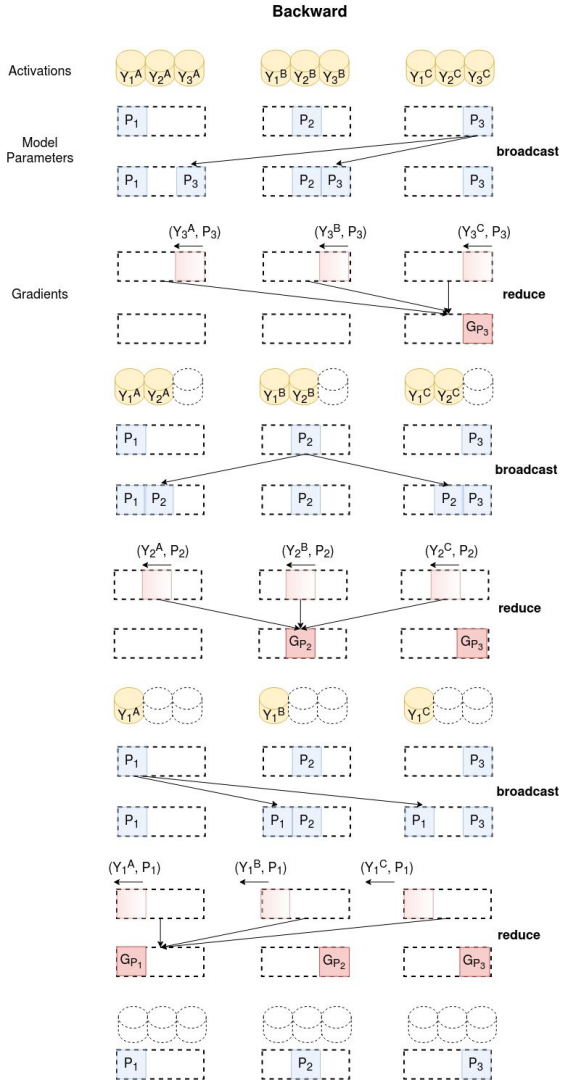
Data Parallelism using ZeRO

Data parallelism using ZeRO:

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



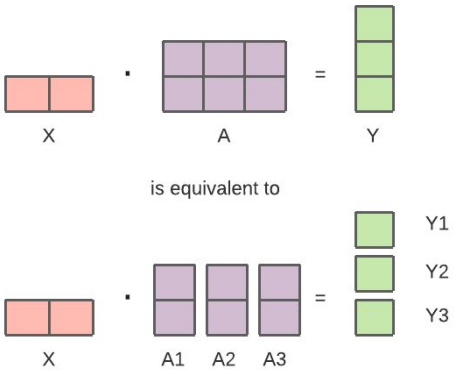


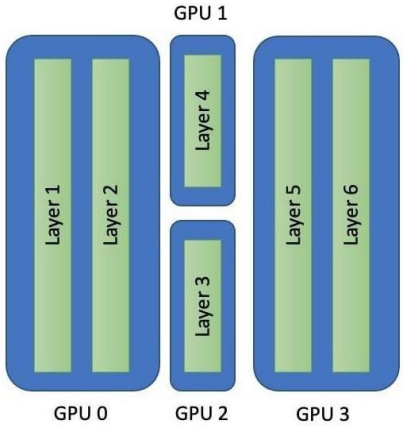


Model Parallelism

On tensor level
("MP", "Horizontal MP")

On layers' level
("Naive MP", "Vertical MP")

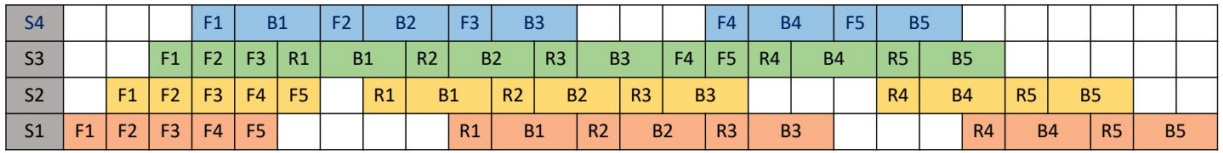




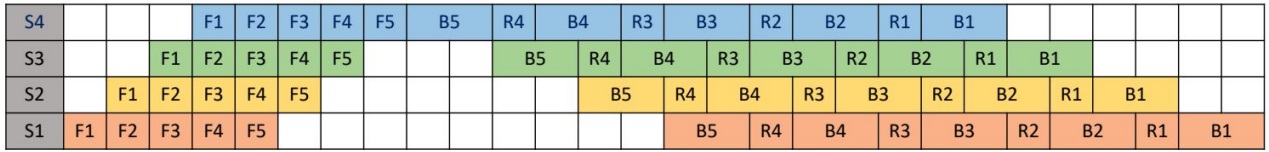
\+ you can train models that do not fit on one device

\- 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.

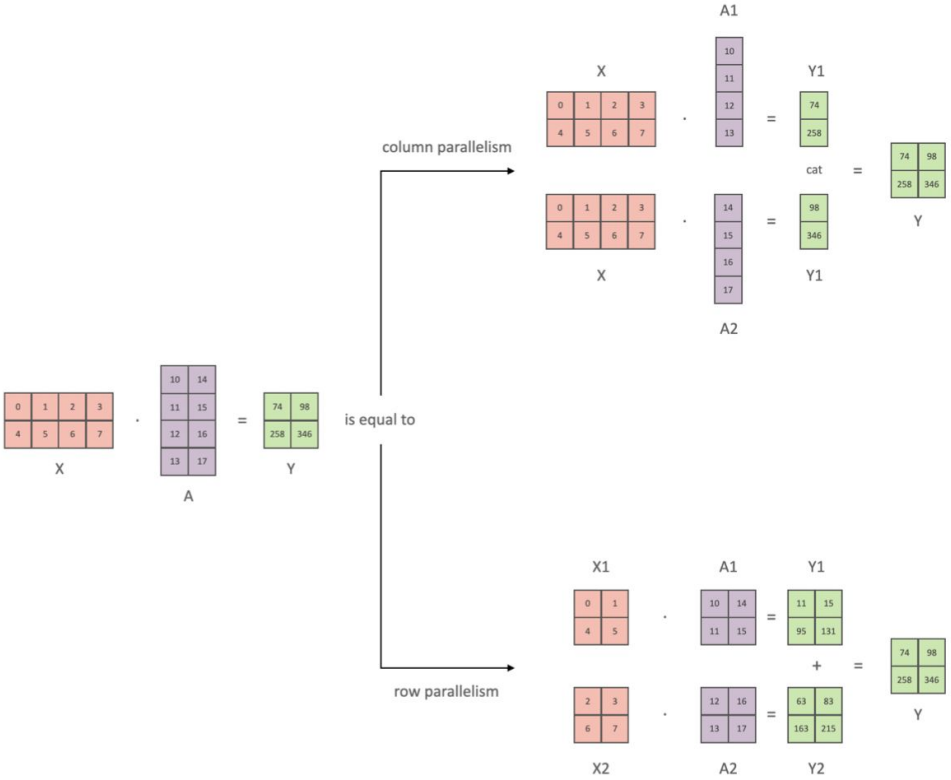


(a) Varuna Schedule

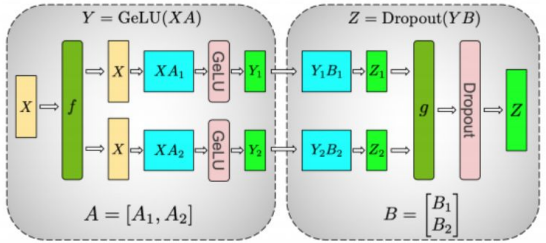


(b) Gpipe Schedule

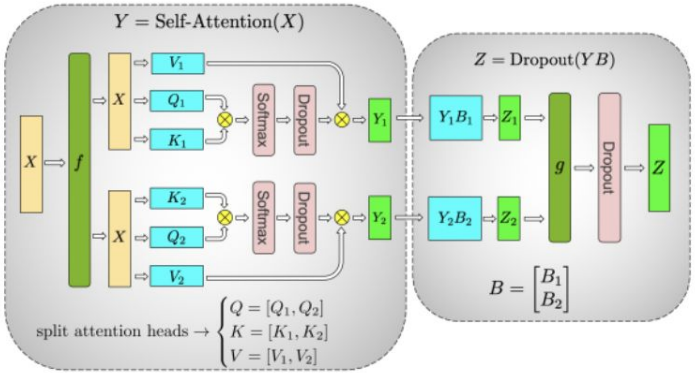
Model Parallelism: on Tensor Level



Tensor parallelism for Transformer

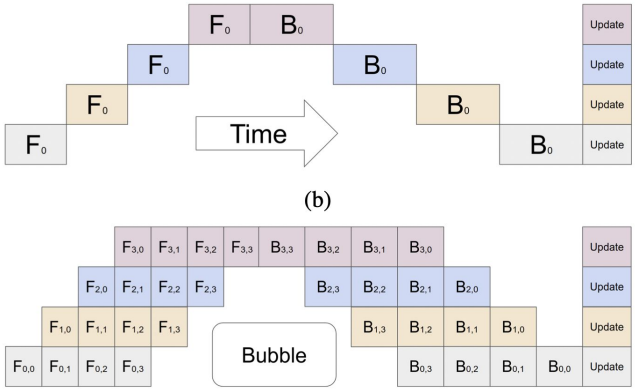


(a) MLP



(b) Self-Attention

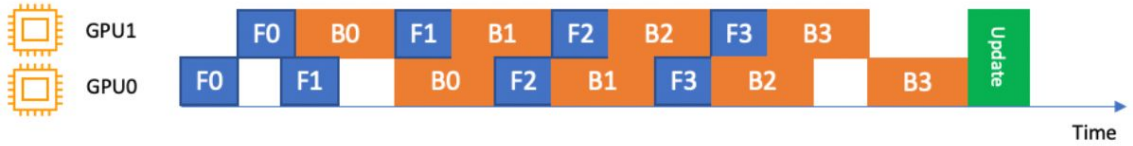
Pipeline Parallelism



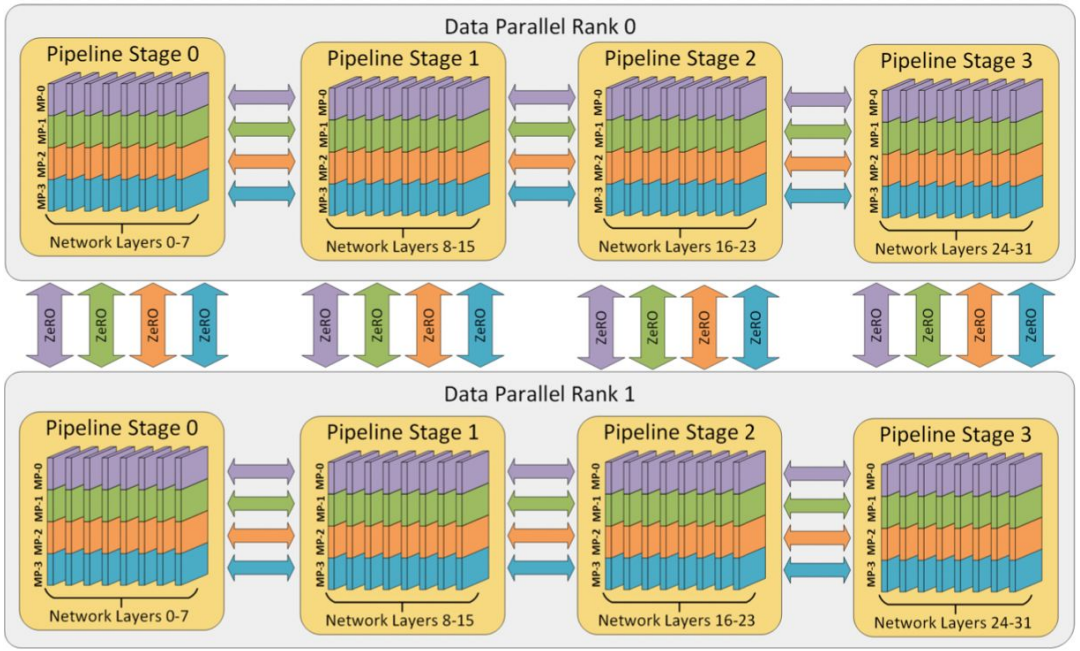
GPipe
batches are divided into micro-batches to reduce downtime

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 tesor-parallelism * 4 groups pipeline-parallelism * 2 groups data-parallelism

MP-n denotes tensor-parallelism

Optimal Strategies for Pipeline Parallelism

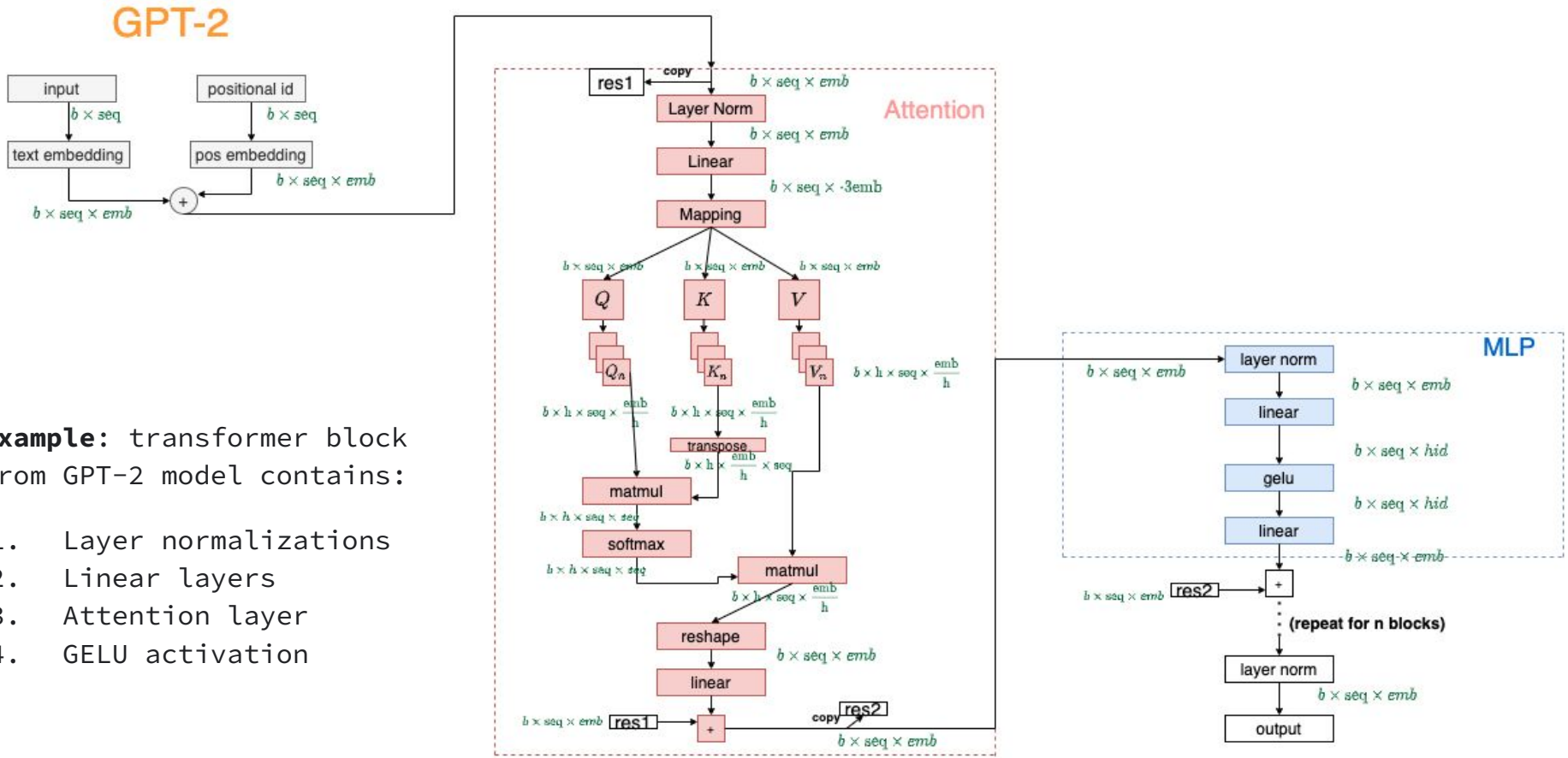
Paper	FlexFlow	PipeDream	PipeDream-2BM	Piper
Types of parallelism	tensor-, model-, data-, operator-	data-, pipeline-	data-, pipeline-	data-, tensor-, pipeline-
Method for hyperparameters optimization	MCMC	Dynamic Programming	Dynamic Programming	Dynamic Programming
Activation recomputation during gradient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

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



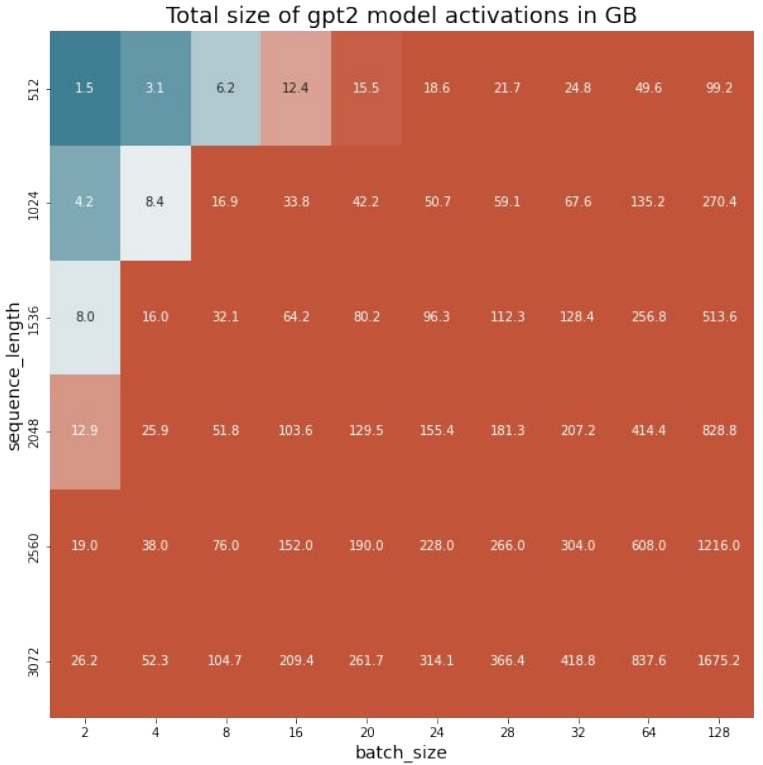
Example: transformer block from GPT-2 model contains:

1. Layer normalizations
2. Linear layers
3. Attention layer
4. GELU activation

GPT-2 Profiling: Memory

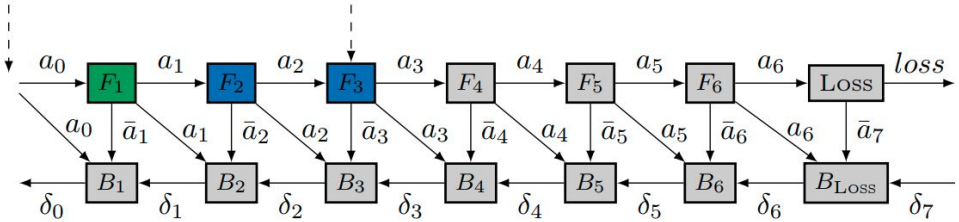
Memory in GB required to store all activations, depending on the batch size and the length of the token sequence.

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;

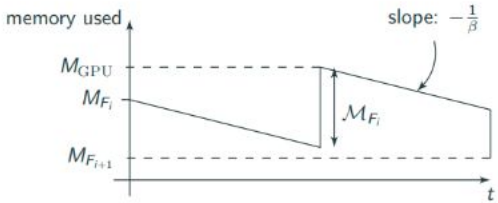


+ saves memory
- slows down training: when computing gradients, you have to recompute activations

Sequence

$F_1^c, F_2^n, F_3^n, F_4^e, F_5^e, F_6^e, \text{Loss}, B_{\text{Loss}}, B_6, B_5, B_4, F_1^c, F_2^n, F_3^e, B_3, F_1^e, F_2^e, B_2, B_1$

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



+ saves memory
- slows down training at low bandwidth β

Optimization Methods

Optimization of Large Scale Models

Problem:

$$\frac{1}{N}\sum_{i=1}^{N} L(\Phi(x_i, \boldsymbol{w}), y_i) \to \min_{\boldsymbol{w}},$$

batch size — loss function — training data — model weights

Algorithm:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \boldsymbol{h}_t,$$

step size

(SGD) $$\boldsymbol{h}_t = \boldsymbol{g}_t = \frac{1}{N_b} \sum_{(x_i, y_i) \in X_b} \frac{\partial L(\Phi(x_i, \boldsymbol{w}), y_i)}{\partial \boldsymbol{w}}\bigg|_{\boldsymbol{w}=\boldsymbol{w}_t}$$

batch size — batch

Questions:

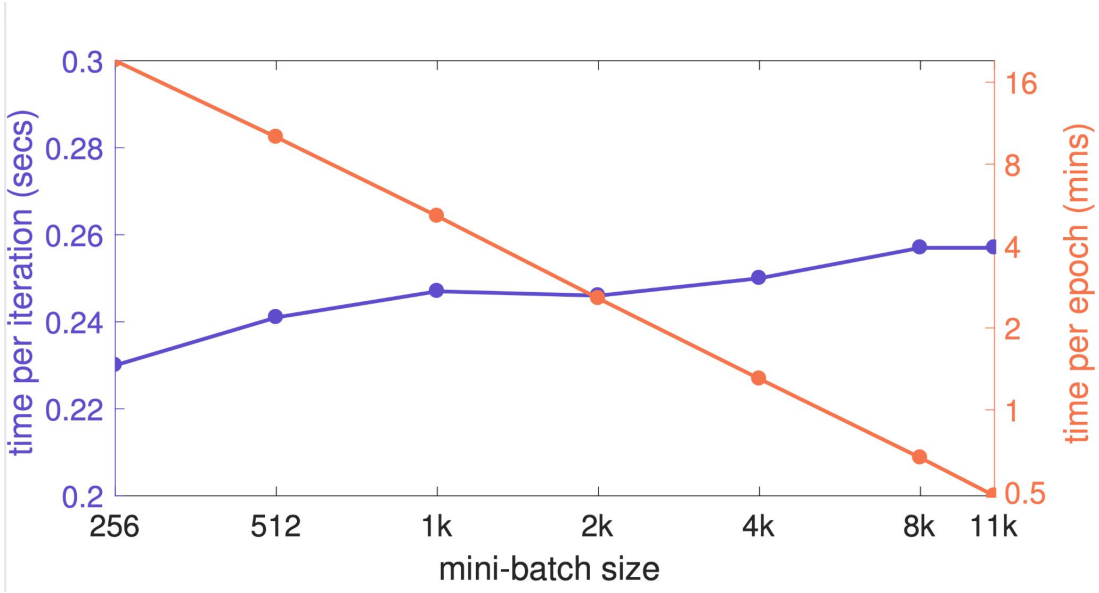
- How to choose step size?
- How to store vectors so they occupy less space?
- How to choose batch size?
- How to use many devices to speed up optimization?
- How to initialize weights?

Optimization Methods

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.

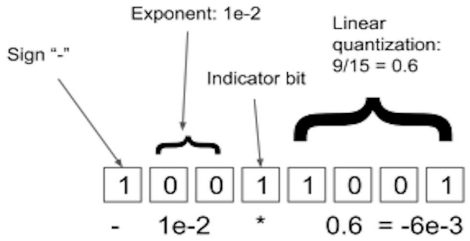
- Increasing the step should be carried out gradually (warmup - phase of the first few epochs).
- Layer-by-layer step size change allows you to increase the batch even more.

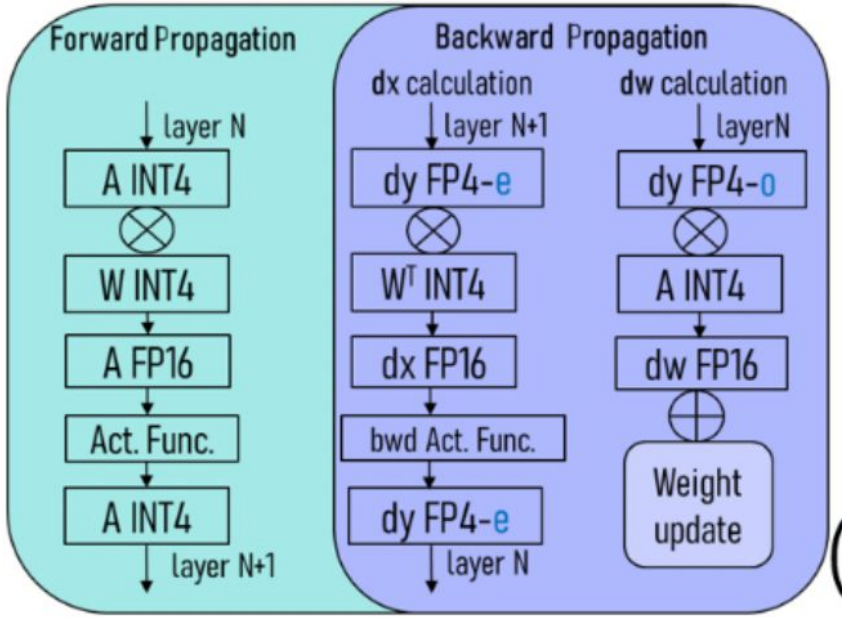


Optimization Methods

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

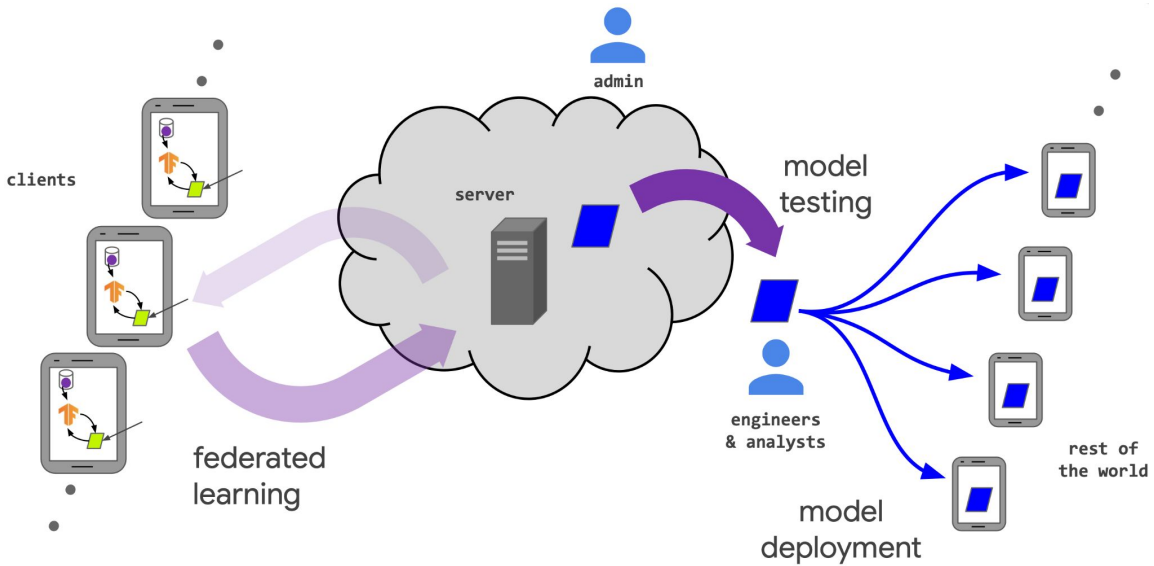




Optimization Methods

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

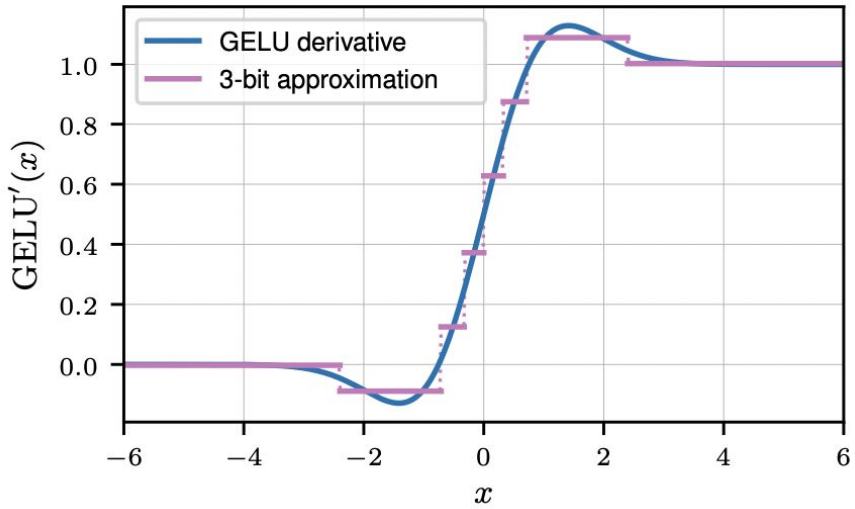
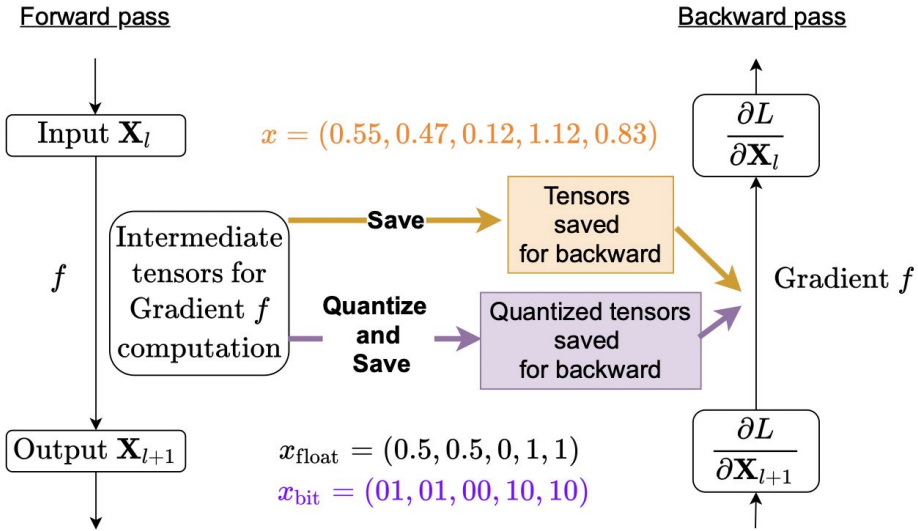


Figure 1. Optimized 3-bit piecewise-constant approximation of the derivative of the GELU activation function.

$$\mathbf{X}_{l+1} = f(\mathbf{X}_l)$$

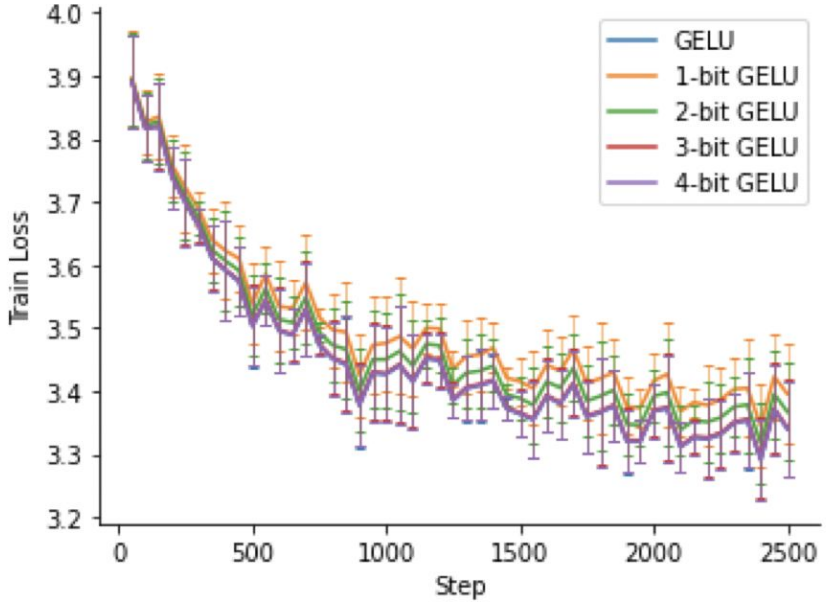
$$\frac{\partial L}{\partial \mathbf{X}_l} = \frac{\partial L}{\partial \mathbf{X}_{l+1}} f'(\mathbf{X}_l),$$

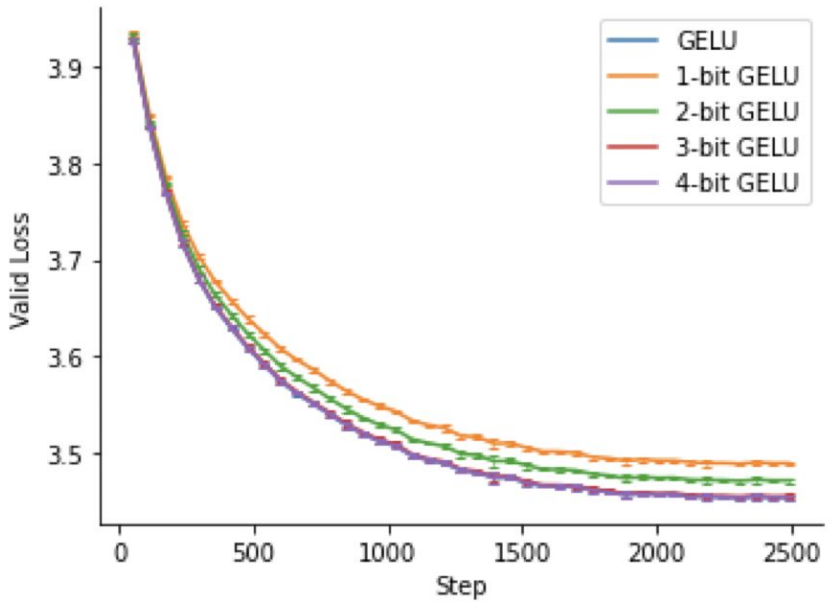
Approximate Activation Gradients: Few-bit Backward



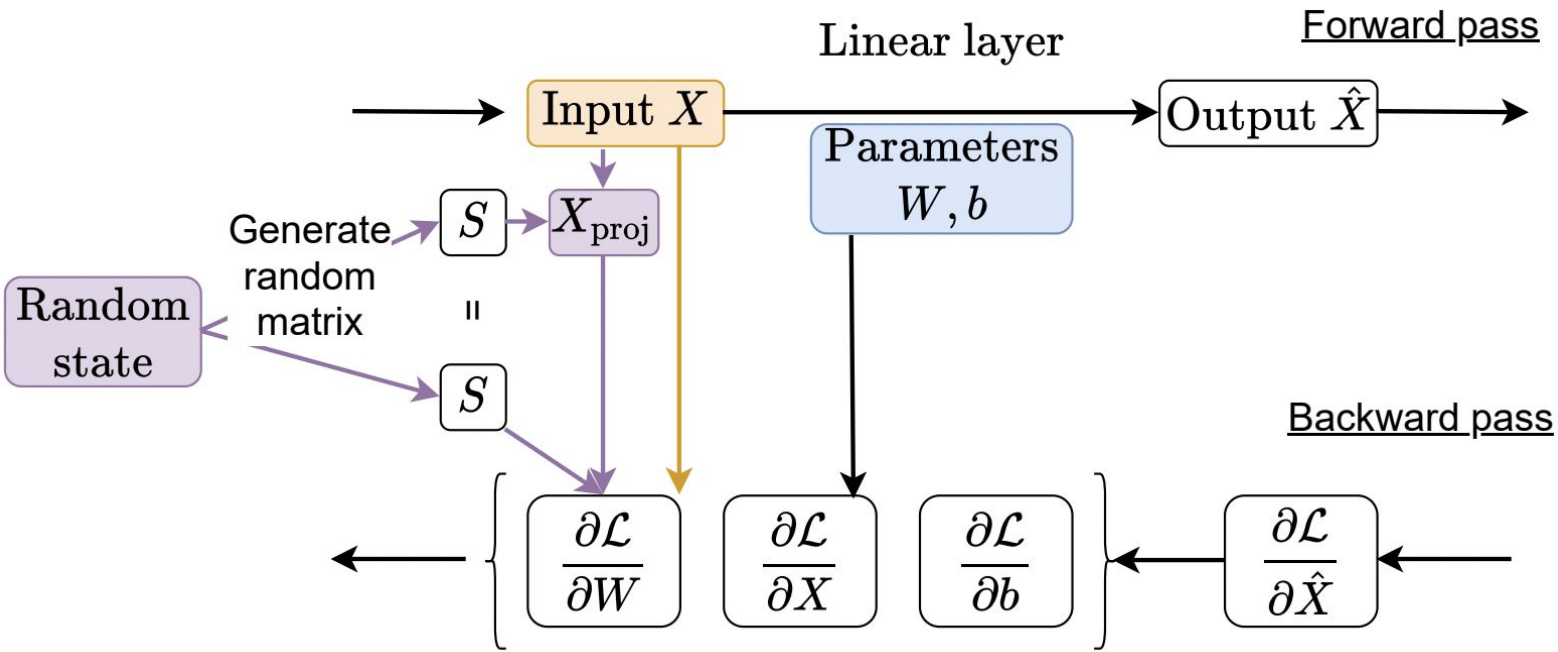

```python
# Globally stored
# piecewise-constant
# approximation parameters
s = [...]
y = [...]

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)
  $\hat{X} \leftarrow XW^\top + \mathbf{1}_B b^\top$
  Generate pseudo random number generator (PRNG) state and corresponding random matrix S
  $X_{\text{proj}} \leftarrow S^\top X$
  Save X_{proj} and PRNG state for the backward pass.
  return Y
end function

function BACKWARD($\partial_{\hat{X}}\mathcal{L}$, W, b, X_{proj})
  $\partial_X \mathcal{L} \leftarrow \partial_{\hat{X}}\mathcal{L} \cdot W^\top$
  Rematerialize matrix S from the PRNG state saved in the forward pass.
  $\partial_W \mathcal{L} \leftarrow \left(\partial_{\hat{X}}\mathcal{L}^\top \cdot S\right) \cdot X_{\text{proj}}$
  $\partial_b \mathcal{L} \leftarrow \partial_{\hat{X}}\mathcal{L}^\top \mathbf{1}_B$
  return $\partial_X \mathcal{L}$, $\partial_W \mathcal{L}$, $\partial_b \mathcal{L}$
end function

Approximate gradients

	Task	Batch Size	GELU	Linear Layer	Peak Memory, GiB	Saving, %
1	MRPC	128	Vanilla	Vanilla	11.30	0.0
2	MRPC	128	3-bit	Vanilla	9.75	13.8
3	MRPC	128	Vanilla	Randomized	9.20	18.6
4	MRPC	128	3-bit	Randomized	7.60	32.7