Lecture 5
BMSystem

Weilin Zhao, Zhengyan Zhang, Guoyang Zeng

THUNLP

BMTrain

THUNLP

https://github.com/OpenBMB/BMTrain

What we want?

Why we need BMTrain?

1. Pretrained Language Models becoming larger.
   Larger models performs better.
2. Training these models becoming more expensive.
   Training code becoming more complex.

What we want to do?

1. Easier.
2. Faster.
3. Cheaper.

How to achieve our goal?

1. Analyze where the GPU memory goes.
2. Understand the cooperation mode between multiple GPUs.

What we want?

Why we need BMTrain?

1. Pretrained Language Models becoming larger.
   Larger models performs better.
2. Training these models becoming more expensive.
   Training code becoming more complex.

What we want to do?

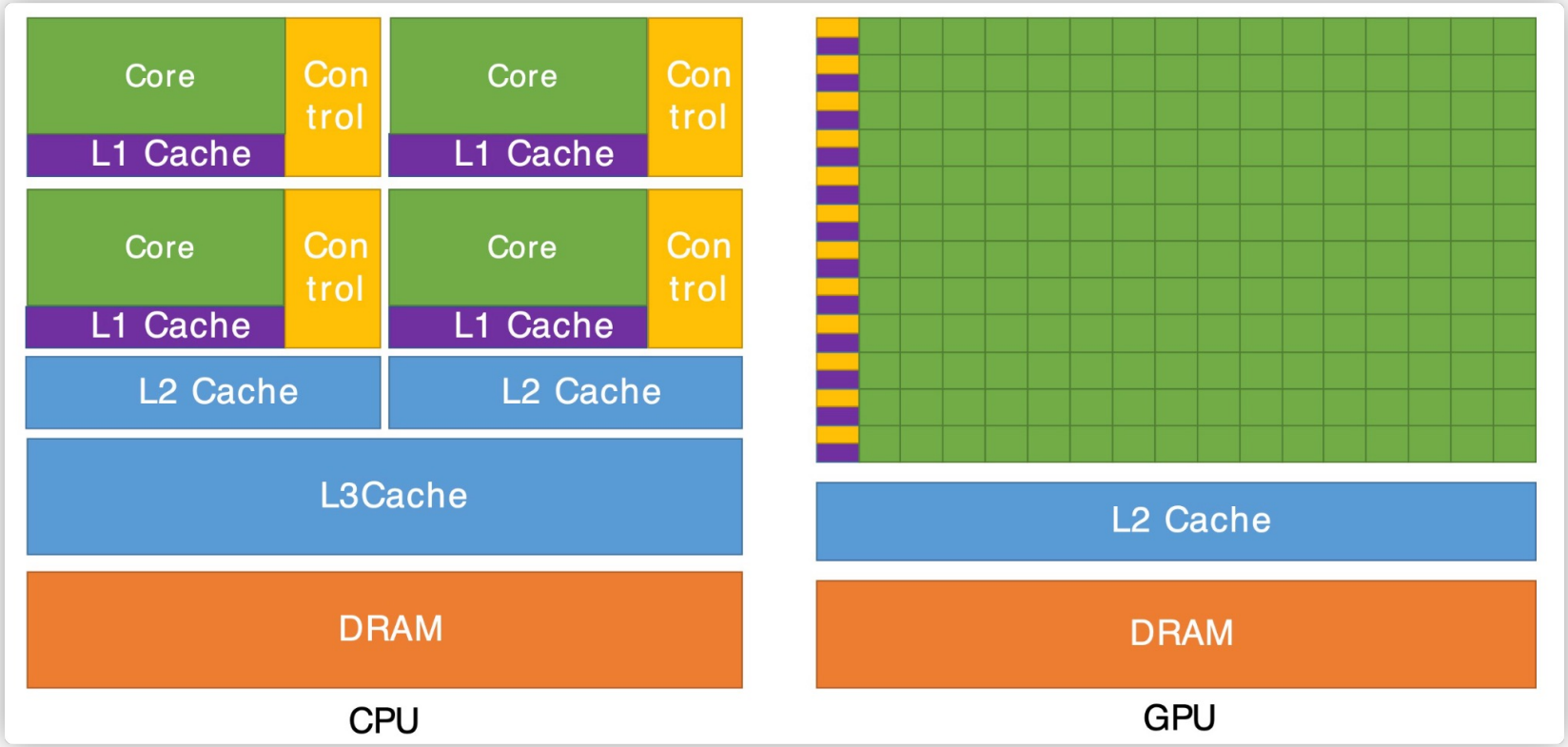
1. Easier.
2. Faster.
3. Cheaper.

How to achieve our goal?

1. Analyze where the GPU memory goes.
2. Understand the cooperation mode between multiple GPUs.

CPU vs. GPU

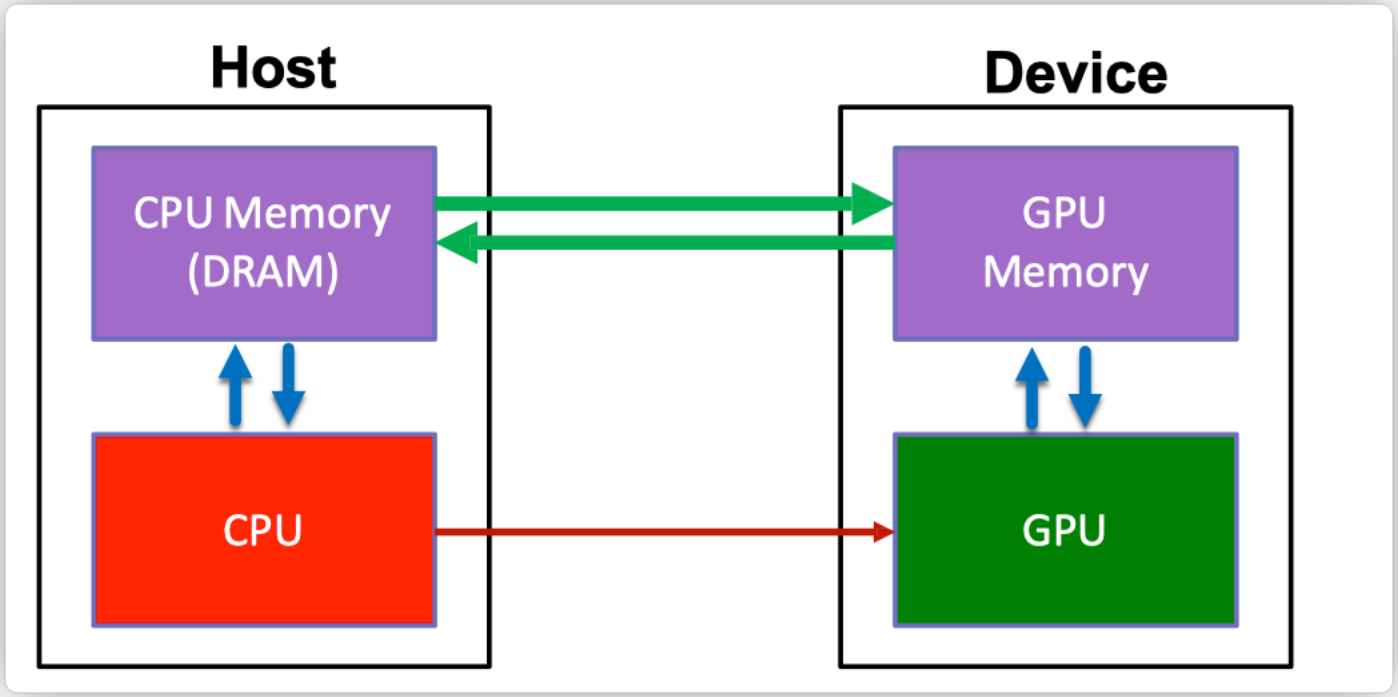
1. CPU: small number of large cores.
2. GPU: large number of small cores.



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

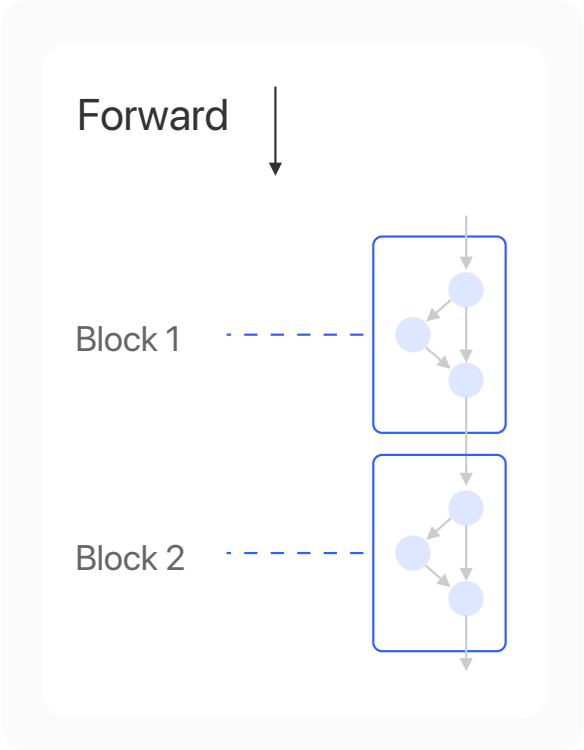
CPU vs. GPU

1. CPU controls GPU.



Memory Components

1. Parameter

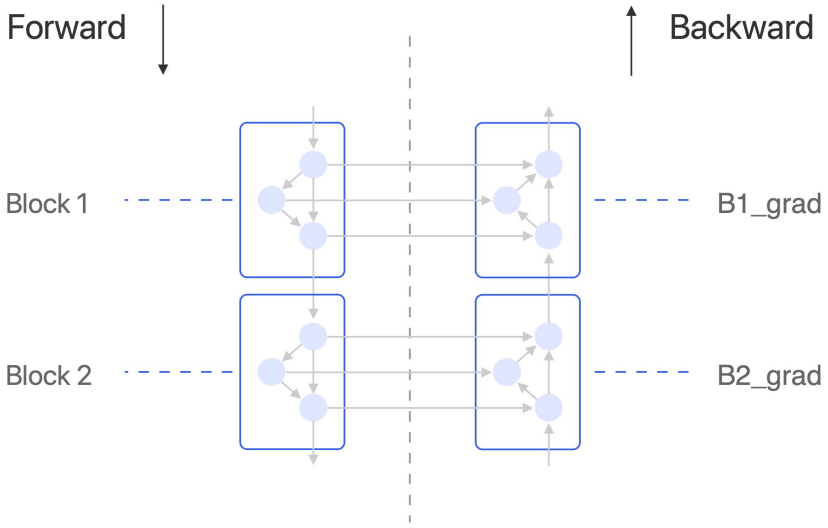


Parameter : 1

Vanilla Finetuning

Memory Components

2. Gradient



Parameter : 1

Gradient : 1

Vanilla Finetuning

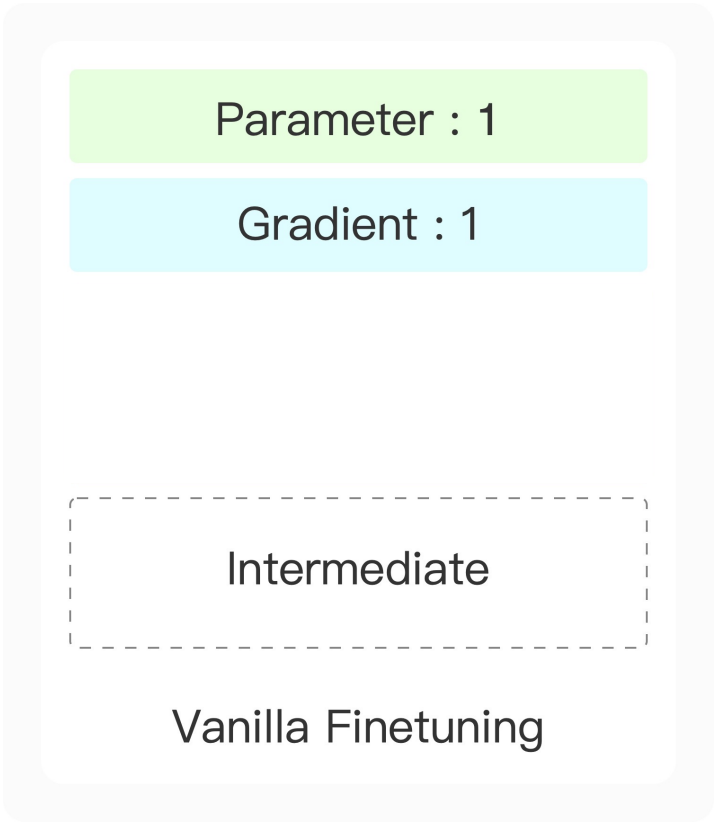
Memory Components

3. Intermediate

The input of each Linear Module needs to be saved for backward.

Each with Shape [Batch, SeqLen, Dim]

$$\begin{aligned} \textbf{Forward:} \quad & \mathbf{y} = W\mathbf{x} \\ \textbf{Backward:} \quad & \nabla\mathbf{x} = W^T\nabla y \\ & \nabla W = (\nabla\mathbf{y})\mathbf{x}^T \end{aligned}$$



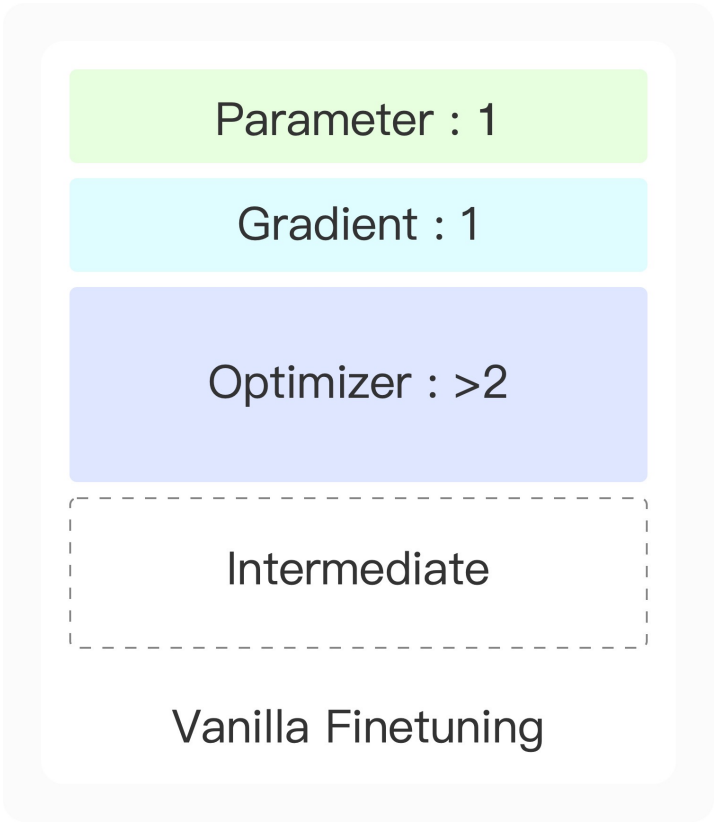
Memory Components

4. Optimizer

Commonly used Adam Optimizer needs to store extra states.

The number of states is greater than 2 times the number of parameters.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$



Adam: A Method for Stochastic Optimization

Memory Components

For example

11B PLM's parameters

$$\frac{11 * 10^9 * 4(FP32)}{1024^3} \approx 40GB$$

What we want?

Why we need BMTrain?

1. Pretrained Language Models becoming larger.
   Larger models performs better.
2. Training these models becoming more expensive.
   Training code becoming more complex.

What we want to do?

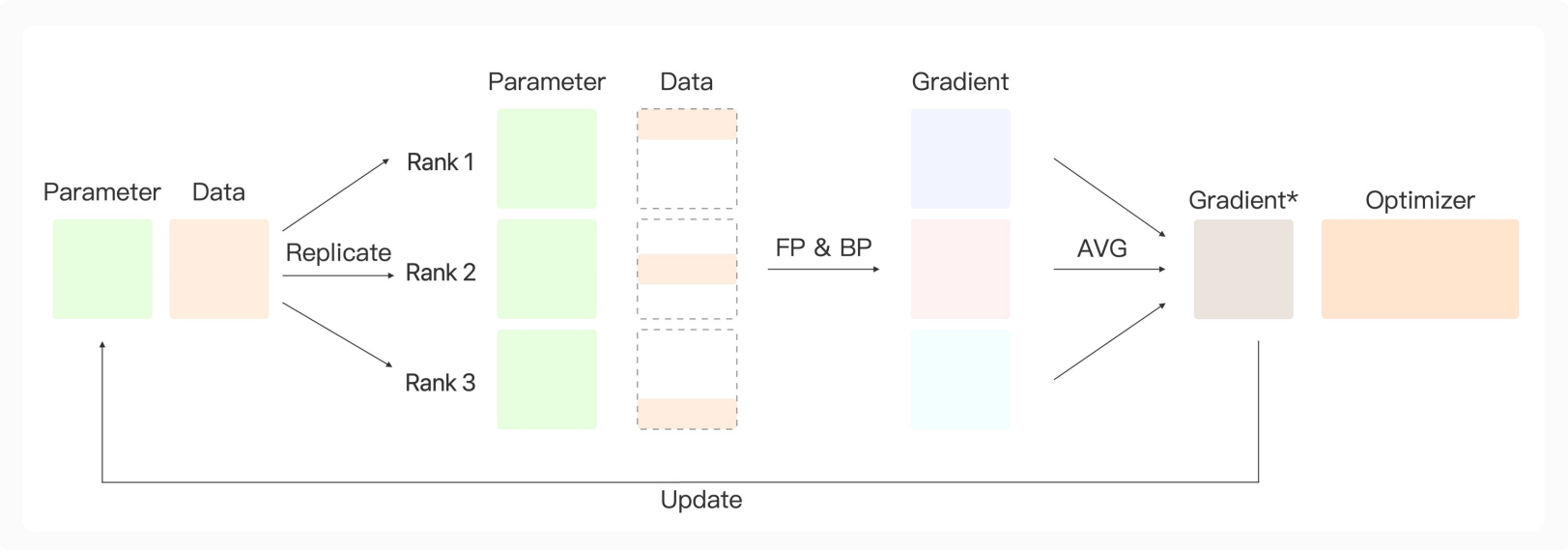
1. Easier.
2. Faster.
3. Cheaper.

How to achieve our goal?

1. Analyze where the GPU memory goes.
2. Understand the cooperation mode between multiple GPUs.

Data Parallel

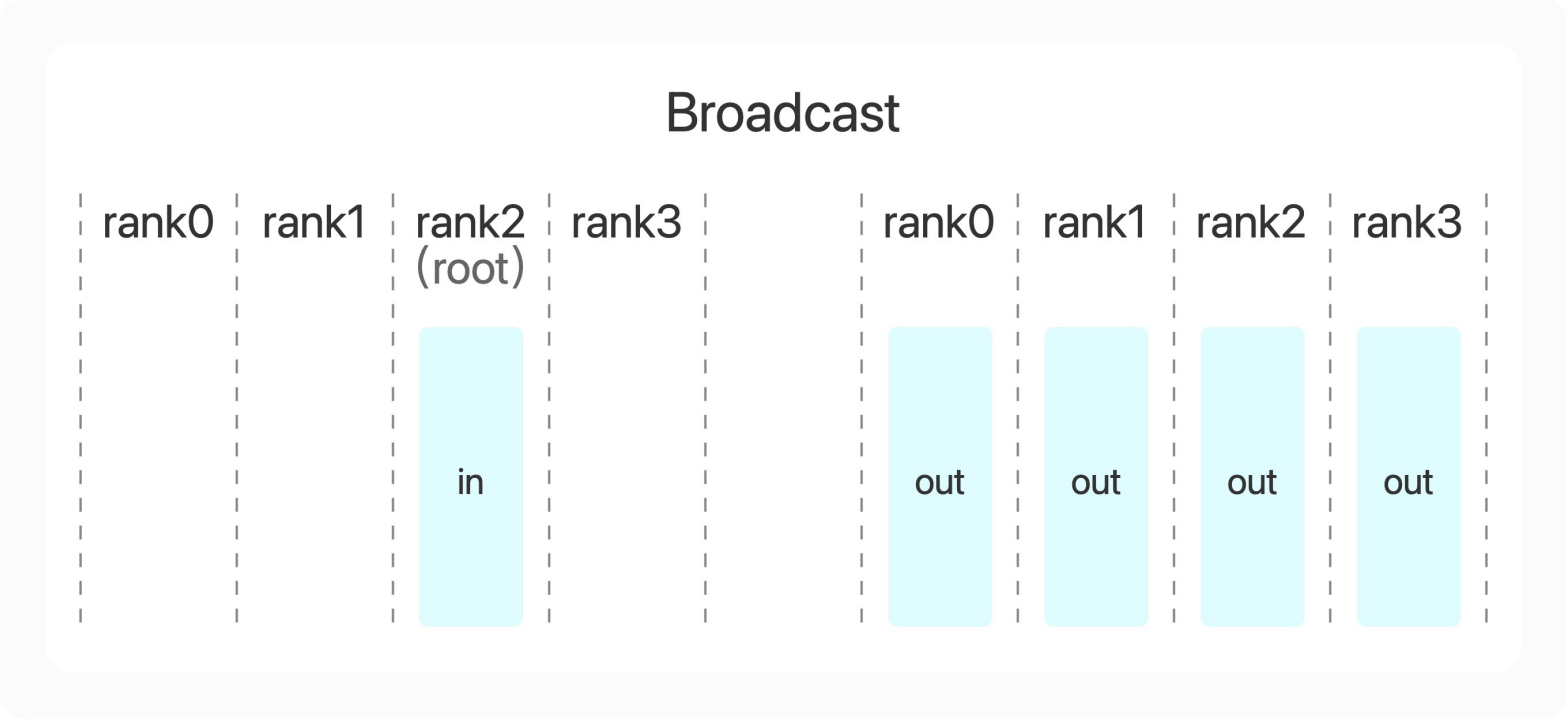
1. There is a parameter server.
2. Forward:
   - The parameter is replicated on each device
   - Each replica handles a portion of the input.
3. Backward
   - Gradients from each replica are averaged.
   - Averaged gradients are used to update the parameter server.



Collective Communication.

1. Broadcast

Send data from one GPU to other GPUs.

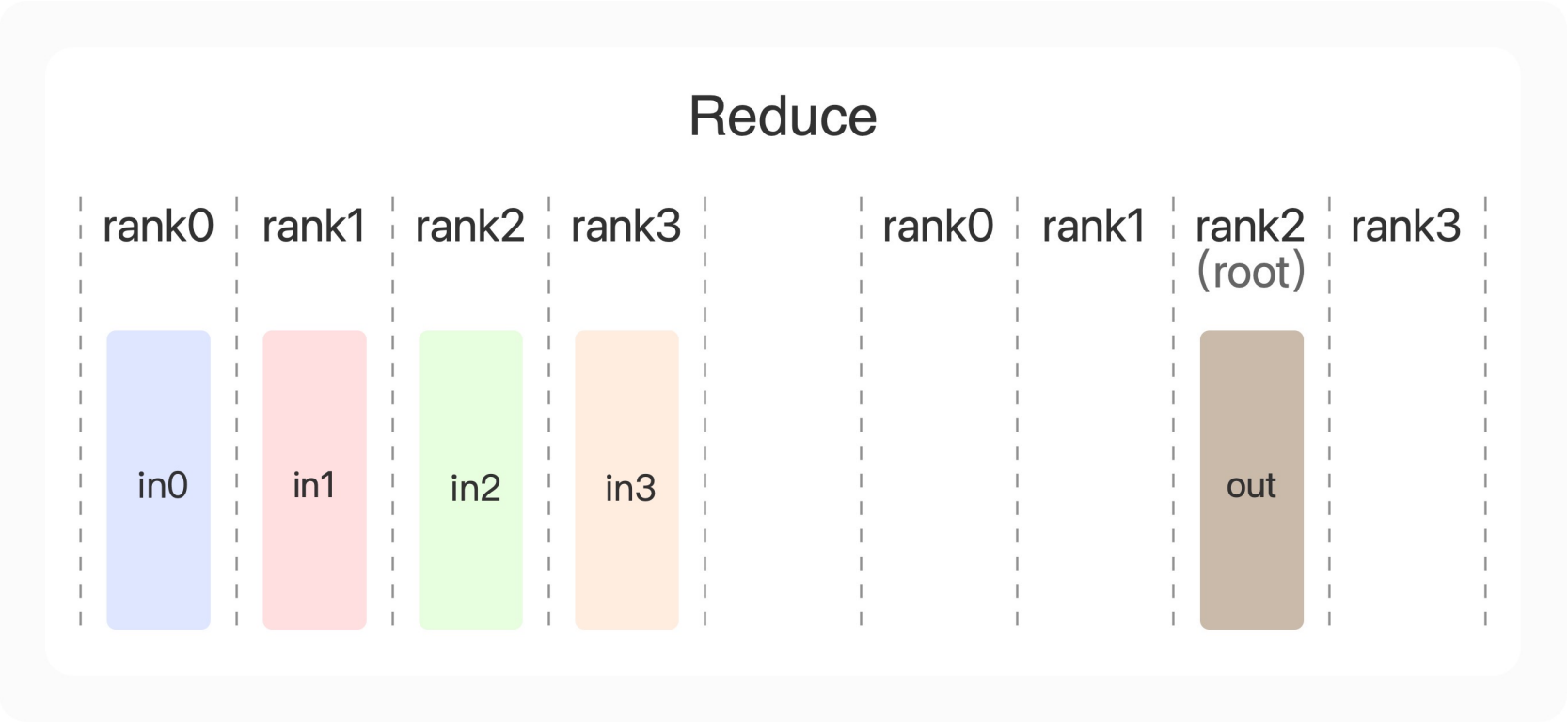


https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

Collective Communication.

2. Reduce

Reduce (Sum/Average) data of all GPUs, send to one GPU.

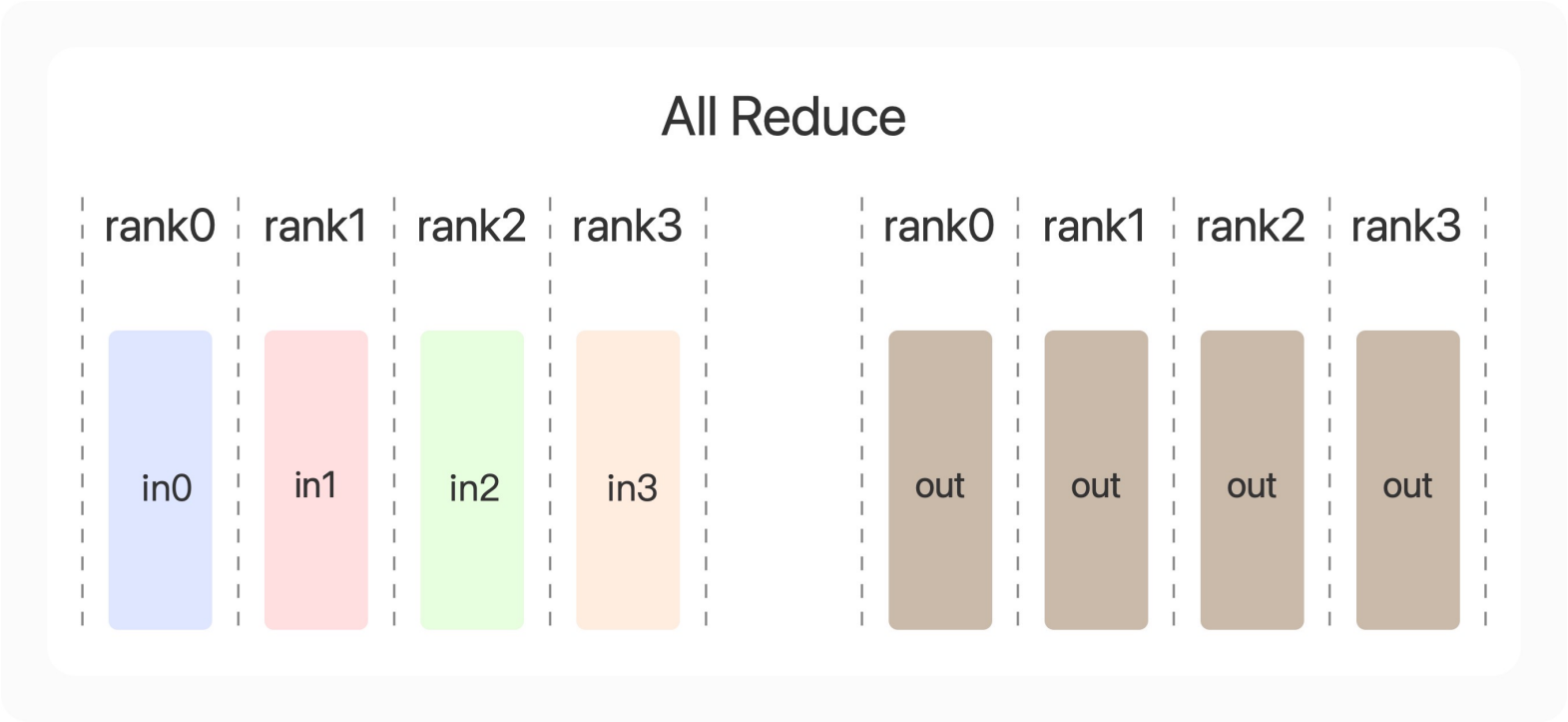


https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

Collective Communication.

3. All Reduce

Reduce (Sum/Average) data of all GPUs, send to all GPUs.

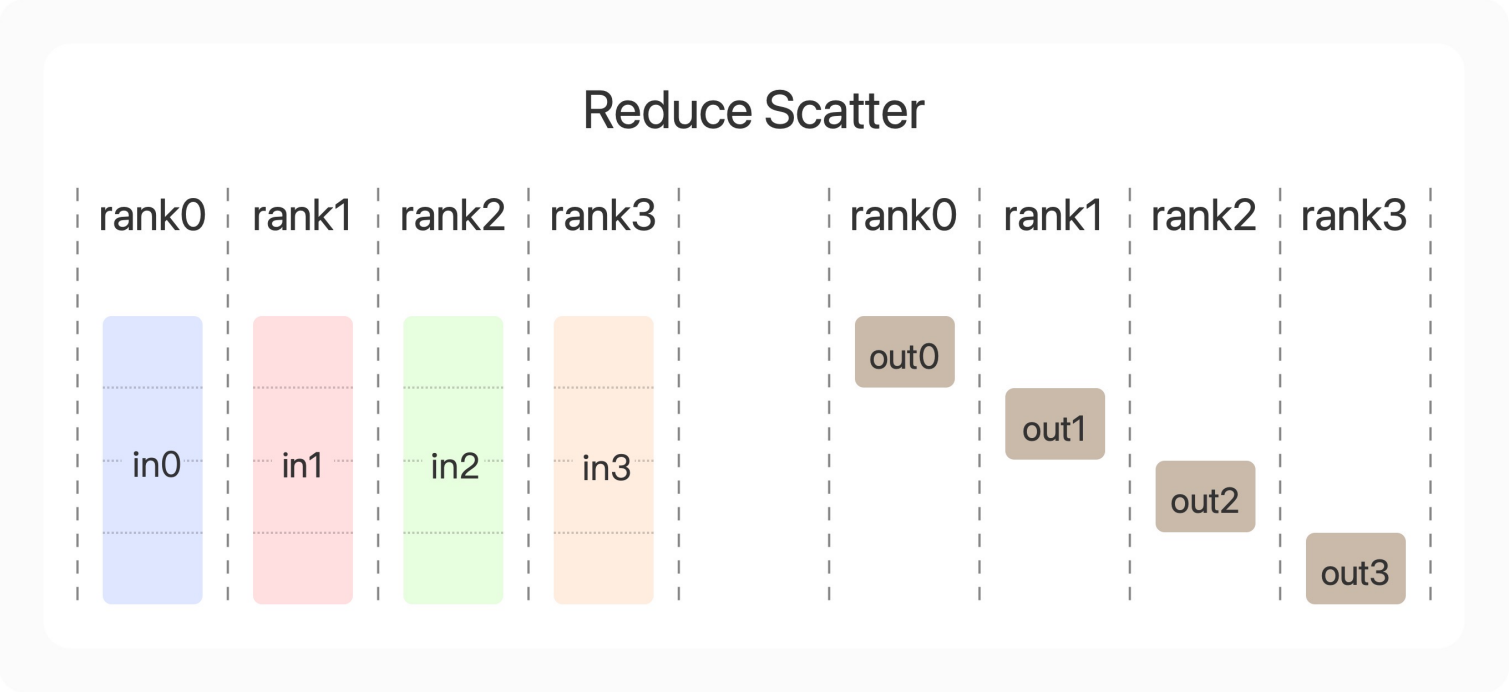


https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

Collective Communication.

4. Reduce Scatter

Reduce (Sum/Average) data of all GPUs, send portions to all GPUs.

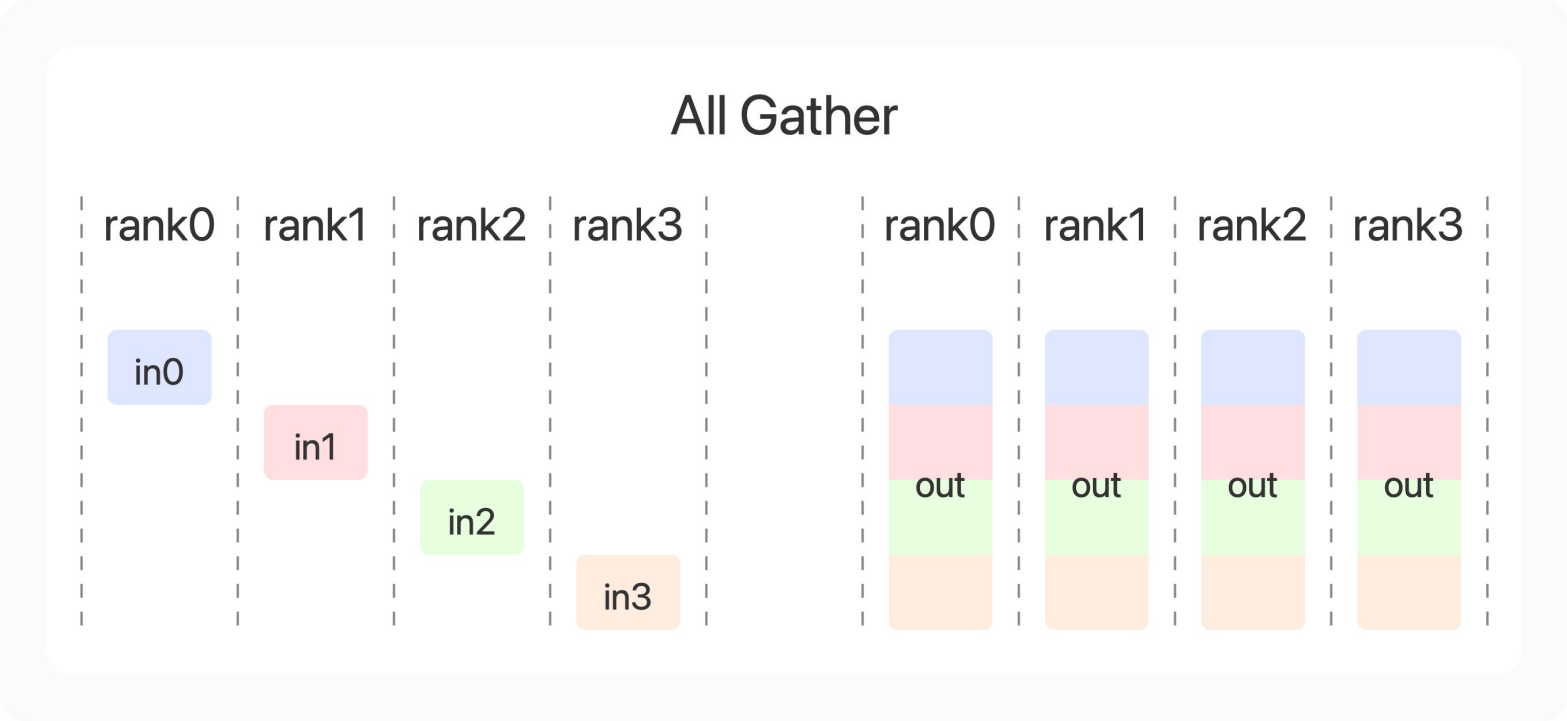


https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

Collective Communication.

5. All Gather

Gather data of all GPUs, send all GPUs.



https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

Methods

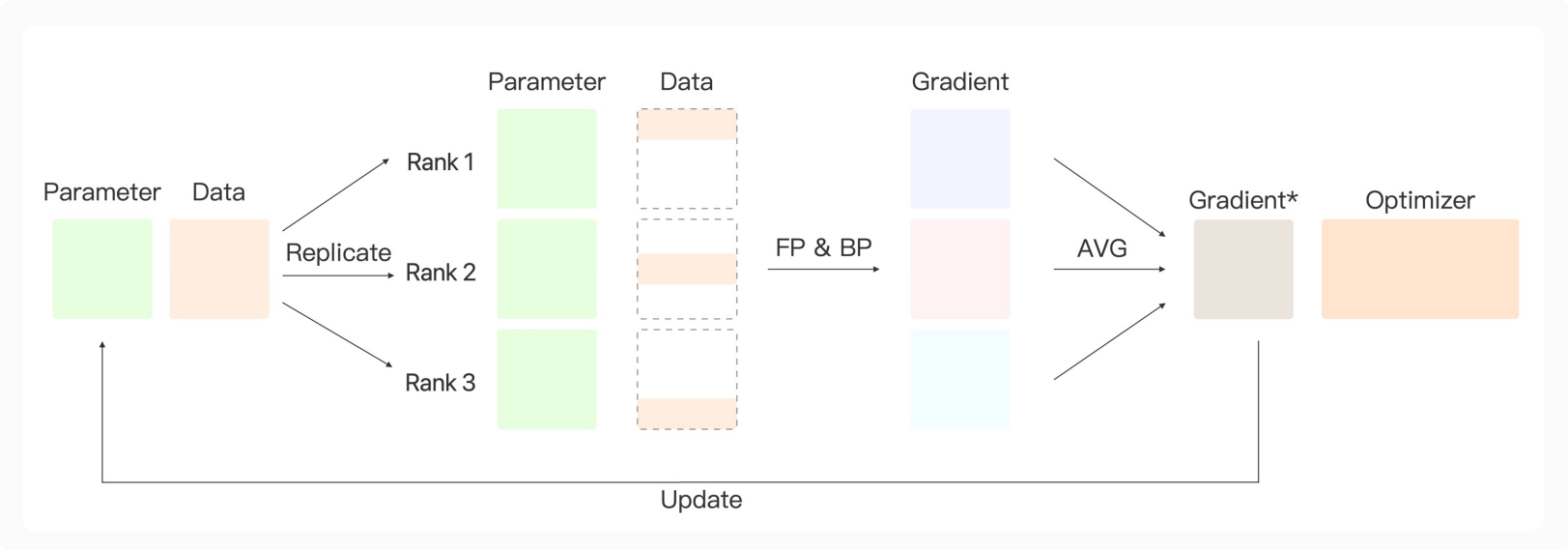
1. Data Parallel
2. Model Parallel
3. ZeRO
4. Pipeline Parallel

Methods

1. Data Parallel
2. Model Parallel
3. ZeRO
4. Pipeline Parallel

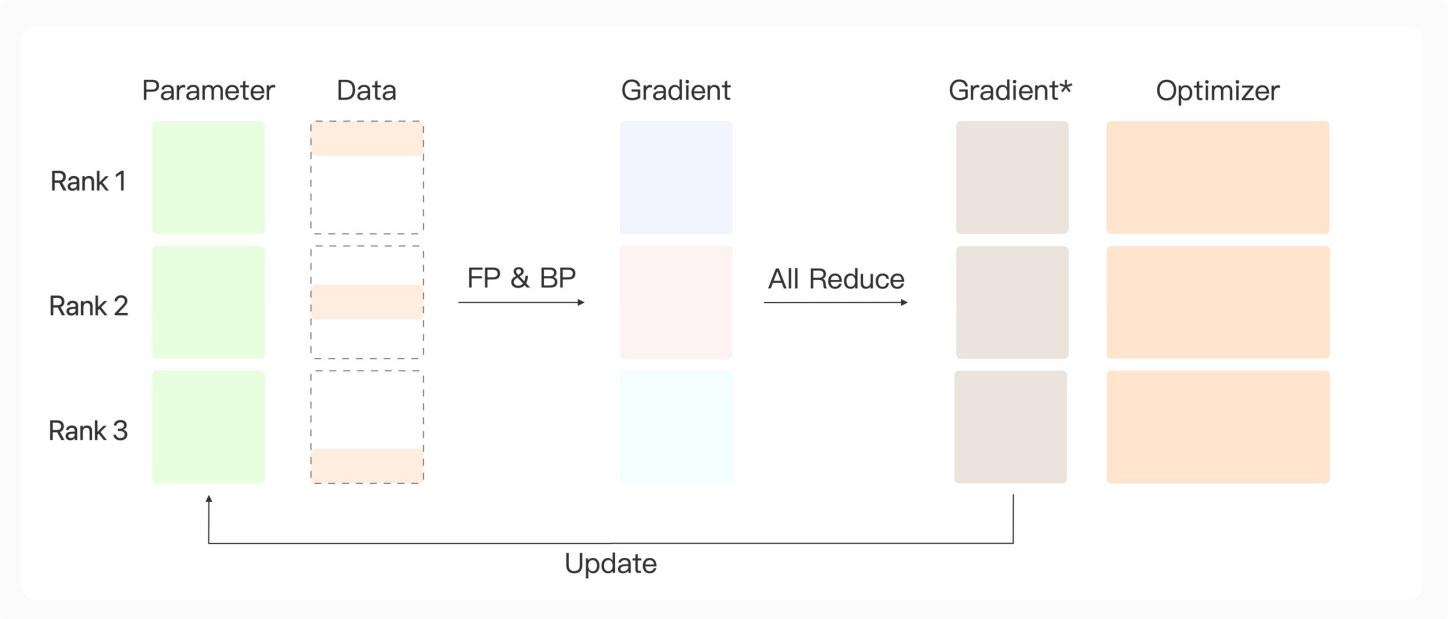
Data Parallel

1. There is a parameter server.
2. Forward:
   - The parameter is replicated on each device
   - Each replica handles a portion of the input.
3. Backward
   - Gradients from each replica are averaged.
   - Averaged gradients are used to update the parameter server.



Distributed Data Parallel

1. There is no parameter server.
2. Forward:
   - Each replica handles a portion of the input.
3. Backward:
   - Gradients from each replica are averaged using All Reduce.
   - Each replica owns optimizer and update parameters itself.
   - Since gradients are shared, parameters are synced.

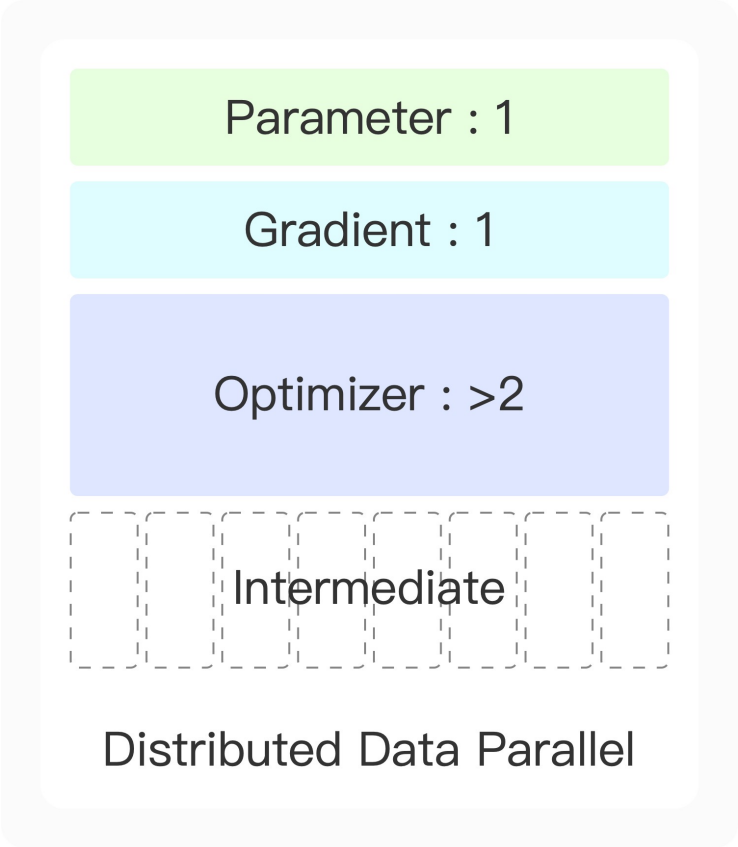


Data Parallel

The input of each Linear Module needs to be saved for backward.

Each with Shape:

1. Without Data Parallel [Batch, Len, Dim]
2. With Data Parallel -> [Batch/n, Len, Dim]



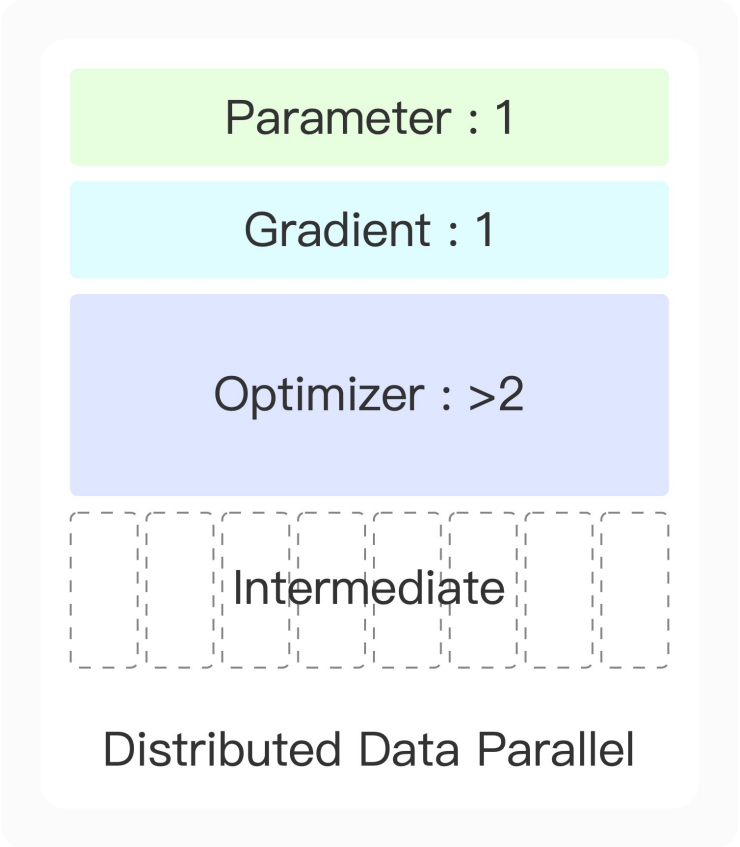
Data Parallel

The input of each Linear Module needs to be saved for backward.

Each with Shape:

1. Without Data Parallel [Batch, Len, Dim]
2. With Data Parallel -> [Batch/n, Len, Dim]

Batch/n >= 1



Methods

1. Data Parallel
2. Model Parallel
3. ZeRO
4. Pipeline Parallel

Model Parallel

- Partition the matrix parameter into sub-matrices.
- Sub-matrices are separated into different GPUs.
- Each GPU handle the sample input.

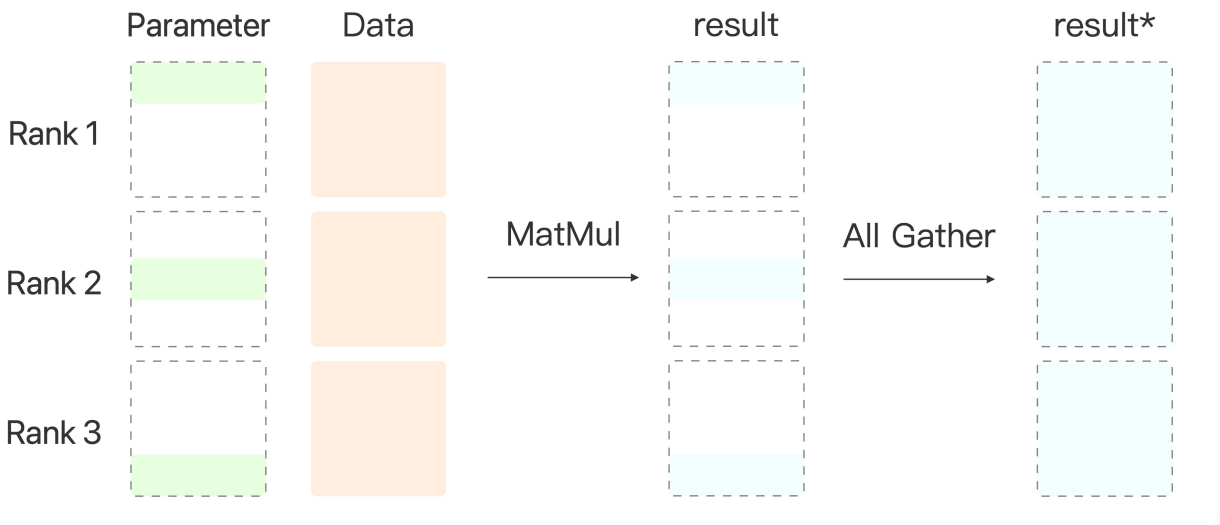
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1x + 2y \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 3x + 4y \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 5x + 6y \end{bmatrix}$$

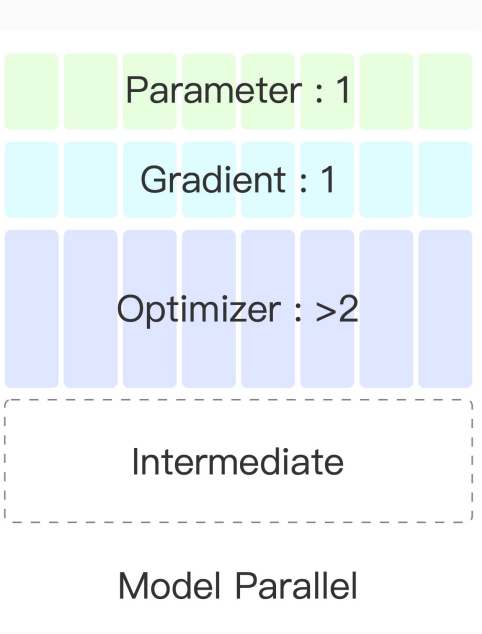
$$\begin{aligned} \mathbf{y}_A &= W_{A \times B}\ \mathbf{x}_B \\ &= [W^{(1)}_{\frac{A}{n} \times B}; W^{(2)}_{\frac{A}{n} \times B}; \cdots; W^{(n)}_{\frac{A}{n} \times B}]\ \mathbf{x}_B \\ &= [W^{(1)}_{\frac{A}{n} \times B}\ \mathbf{x}_B; W^{(2)}_{\frac{A}{n} \times B}\ \mathbf{x}_B; \cdots; W^{(n)}_{\frac{A}{n} \times B}\ \mathbf{x}_B] \end{aligned}$$

Megatron-lm: Training multi-billion parameter language models using model parallelism.

Model Parallel

- Partition the matrix parameter into sub-matrices.
- Sub-matrices are separated into different GPUs.
- Each GPU handle the sample input.



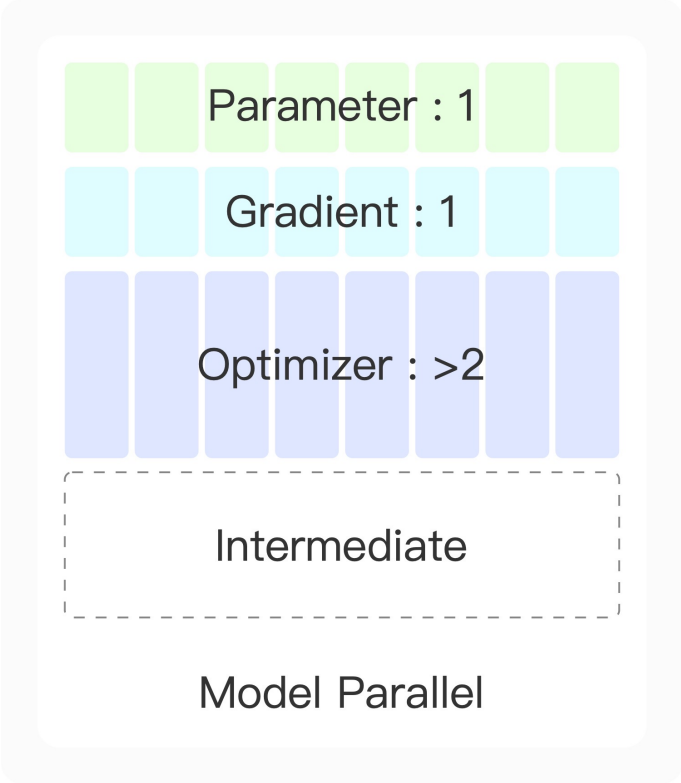


Megatron-lm: Training multi-billion parameter language models using model parallelism.

Model Parallel

- Partition the matrix parameter into sub-matrices.
- Sub-matrices are separated into different GPUs.
- Each GPU handle the sample input.

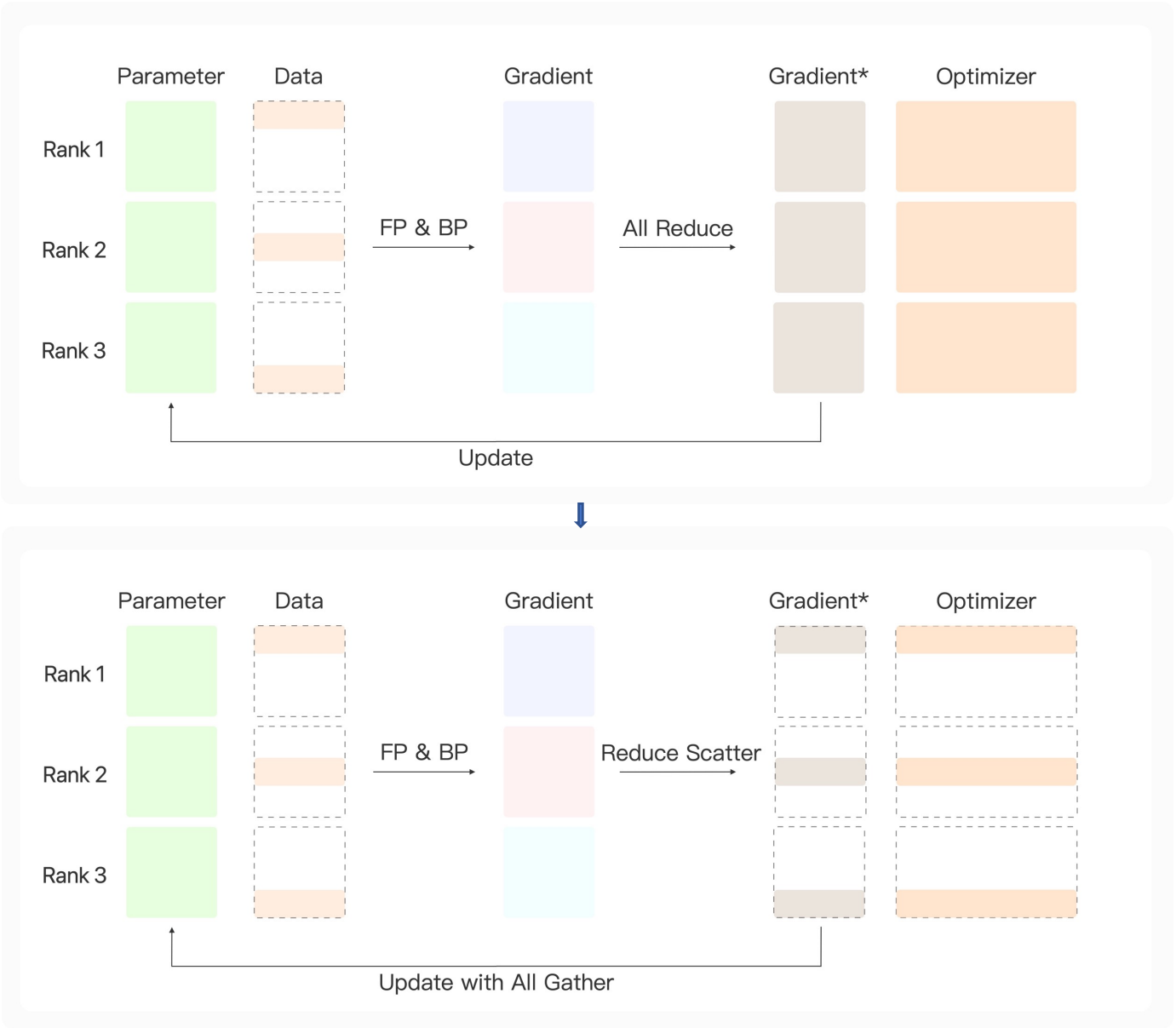
Intermediates are not partitioned.



Methods

1. Data Parallel
2. Model Parallel
3. ZeRO
4. Pipeline Parallel

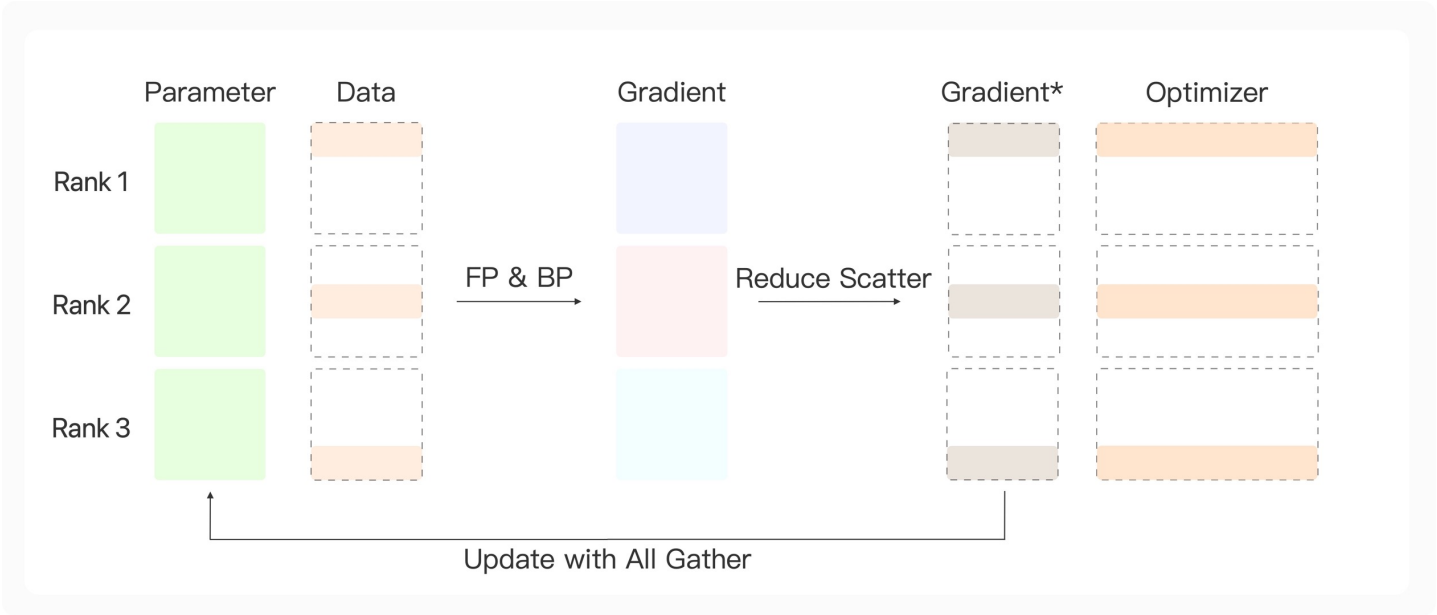
Zero Redundancy Optimizer



Zero Redundancy Optimizer

ZeRO-Stage 1

1. Each replica handles a portion of the input.
2. Forward.
3. Backward.
4. Average all gradients using Reduce Scatter.
5. Each replica owns part of optimizer & update part of params.
6. Updated parameter are synced using All Gather.

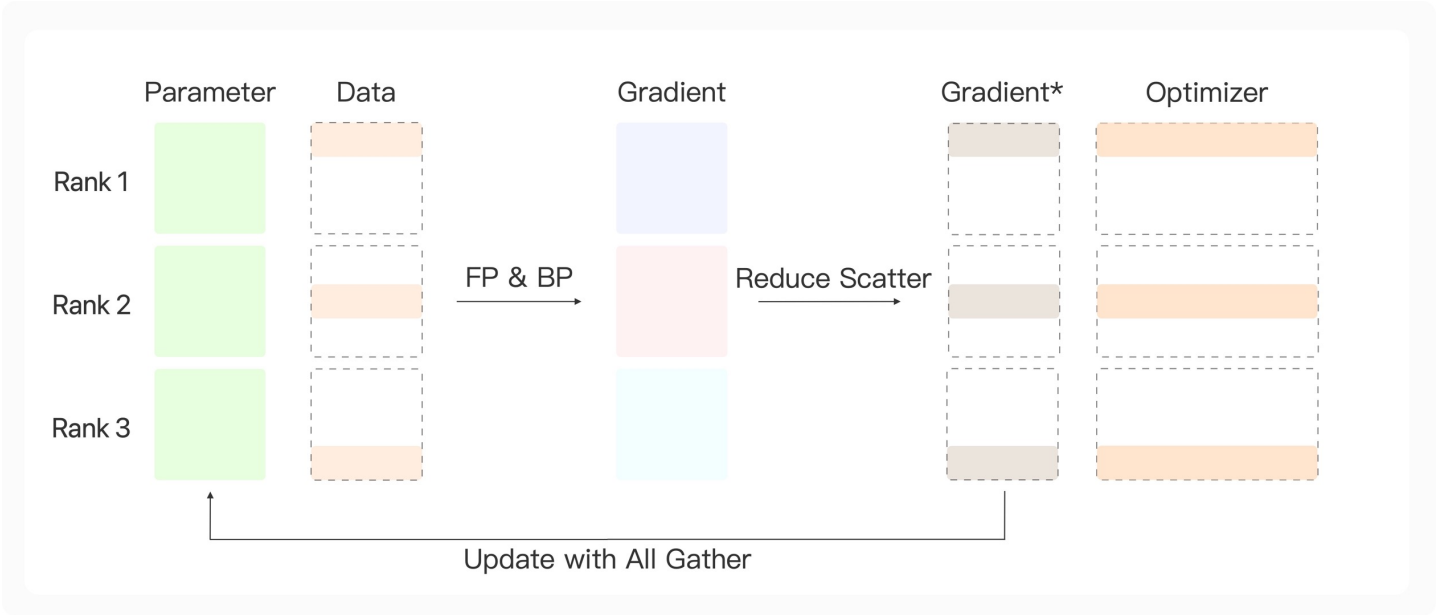


ZeRO: Memory optimizations Toward Training Trillion Parameter Models

Zero Redundancy Optimizer

ZeRO-Stage 2

1. Each replica handles a portion of the input.
2. Forward.
3. Backward (Average gradients using Reduce Scatter).
4.
5. Each replica owns part of optimizer & update part of params.
6. Updated parameter are synced using All Gather.

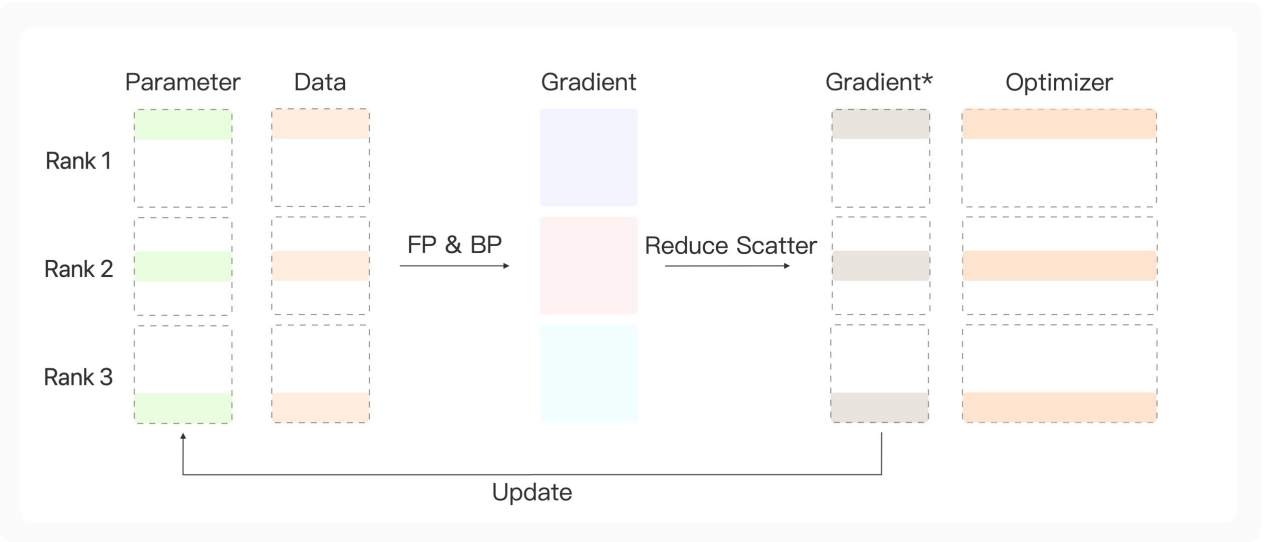


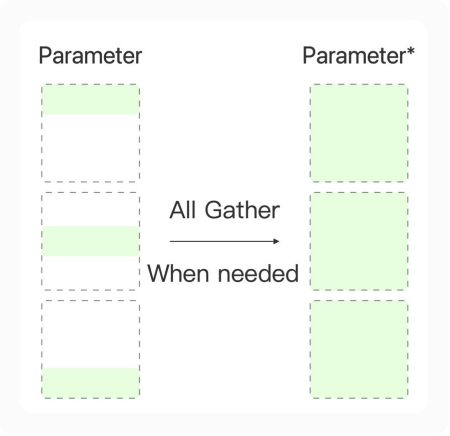
ZeRO: Memory optimizations Toward Training Trillion Parameter Models

Zero Redundancy Optimizer

ZeRO-Stage 3

1. Each replica handles a portion of the input.
2. Forward (Share parameters using All Gather).
3. Backward (Average gradients using Reduce Scatter).
4.
5. Each replica owns part of optimizer & update part of params.

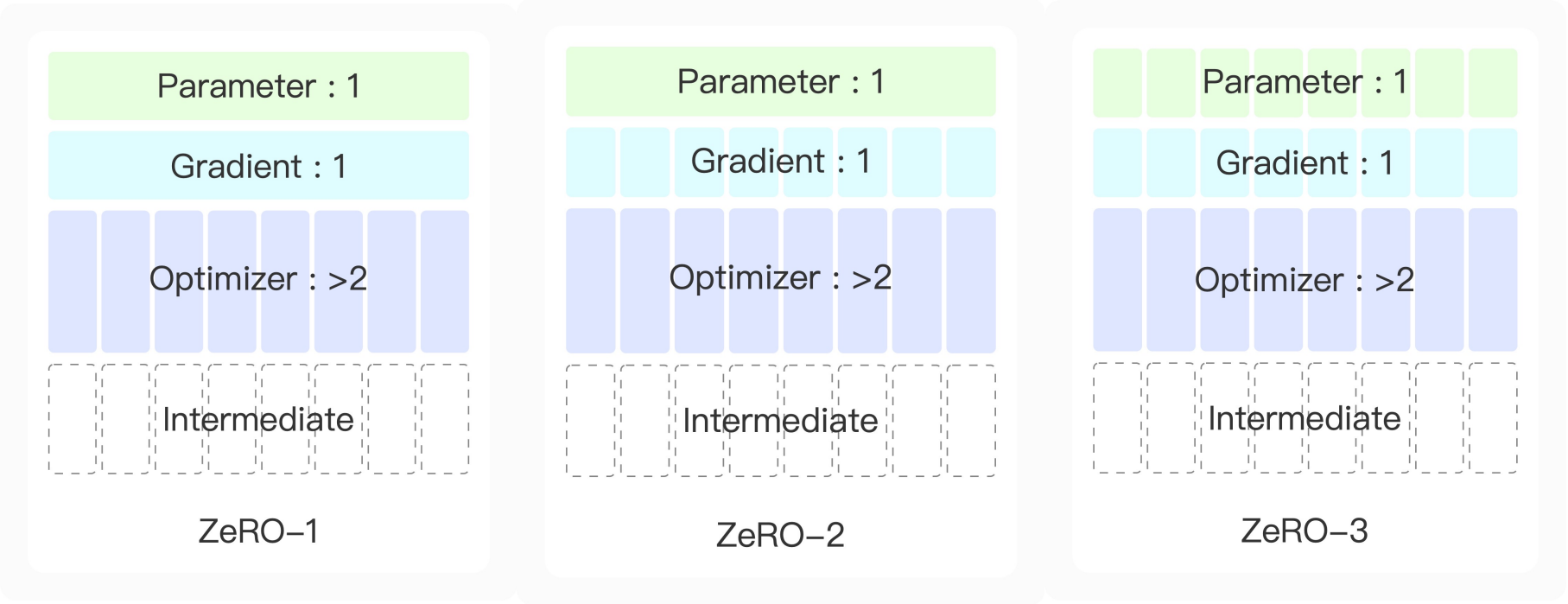




ZeRO: Memory optimizations Toward Training Trillion Parameter Models

Zero Redundancy Optimizer

Memory Analysis



ZeRO: Memory optimizations Toward Training Trillion Parameter Models

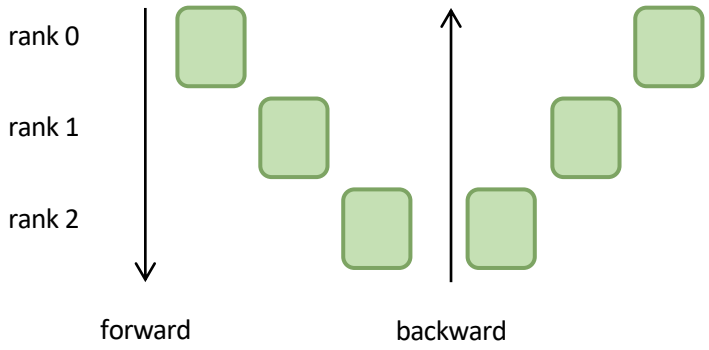
Methods

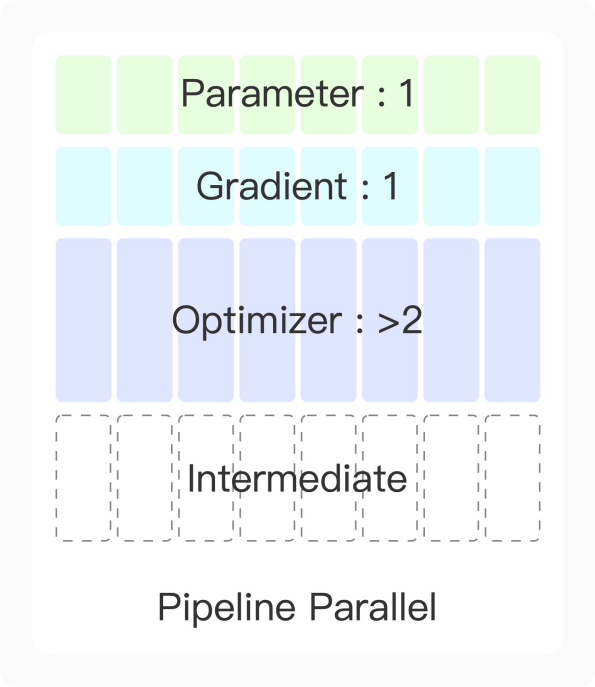
1. Data Parallel
2. Model Parallel
3. ZeRO
4. Pipeline Parallel

Pipeline Parallel

1. Transformer are partitioned layer by layer.
2. Different layers are put on different GPUs.

Forward : Layer i -> Layer i+1
Backward: Layer i -> Layer i-1





Gpipe: Efficient training of giant neural networks using pipeline parallelism.

Techniques

1. Mixed precision
2. Offloading
3. Overlapping
4. Checkpointing

Techniques

1. Mixed precision
2. Offloading
3. Overlapping
4. Checkpointing

Mixed Precision

FP32: 1.18e-38~3.40e38 with 6–9 significant decimal digits precision.
FP16: 6.10e−5 ~65504 with 4 significant decimal digits precision.

Advantages:

1. Math operations run much faster.
2. Math operations run even more faster with Tensor Core support.
3. Data transfer operations require less memory bandwidth.
4. Smaller range but not overflow.

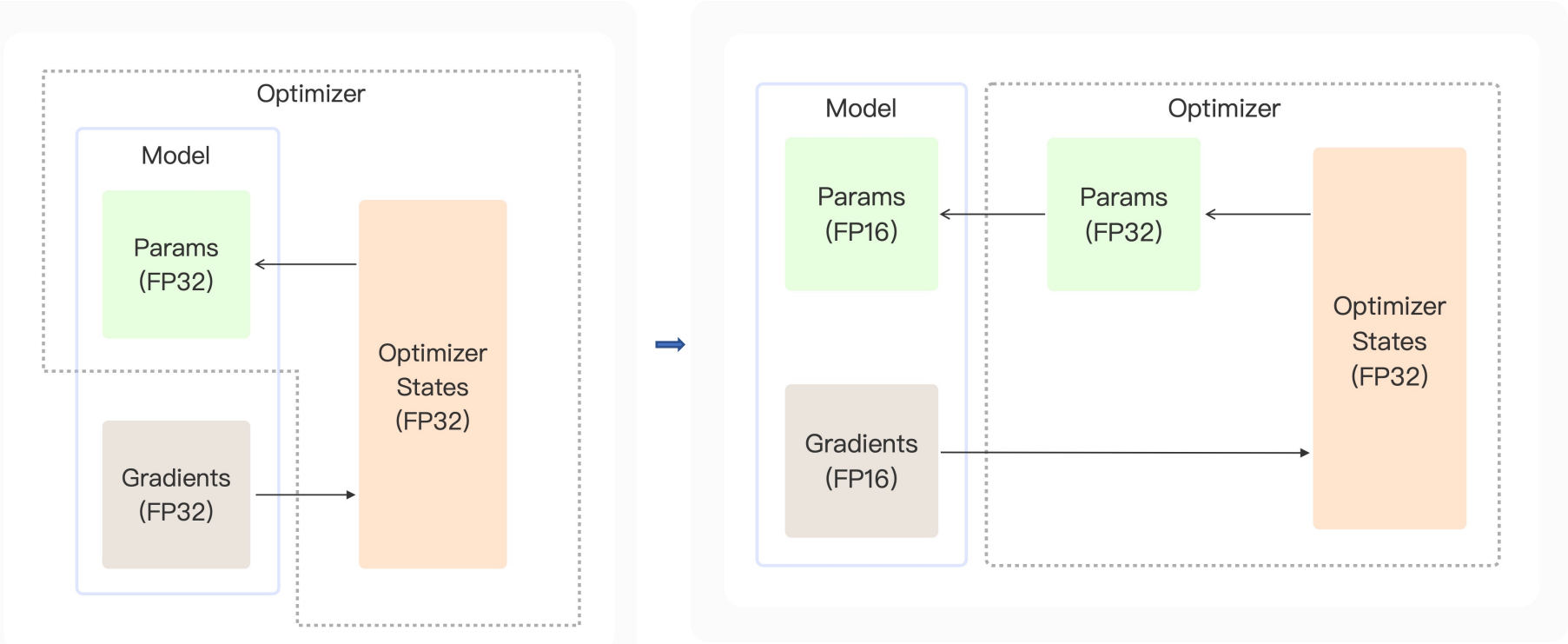
Disadvantages:

1. Weight update≈gradient∗lr
   Smaller range, especially underflow.

Mixed Precision Training.

Mixed Precision

Keep a master FP32 parameters in optimizer.



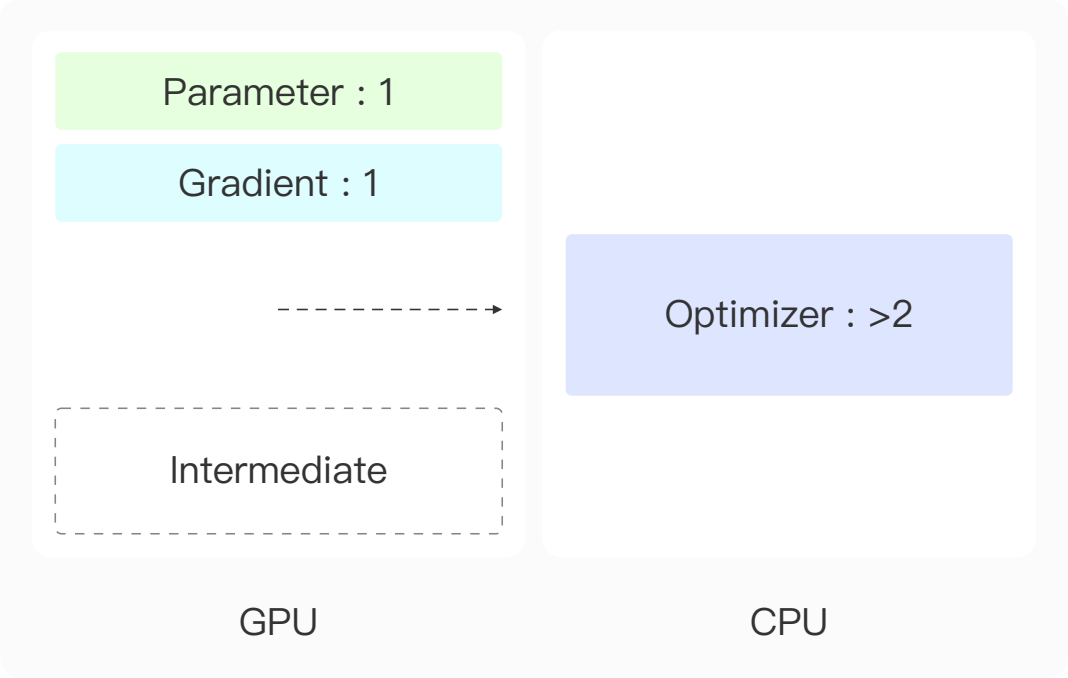
Mixed Precision Training.

Techniques

1. Mixed precision
2. Offloading
3. Overlapping
4. Checkpointing

Offloading

- Bind each GPU with multiple CPUs.
- Offload the partitioned optimizer states to CPU.
  1. Send Gradients from GPU to CPU.
  2. Update optimizer states on CPU (using OpenMP + SIMD).
  3. Send back updated parameters from CPU to GPU.



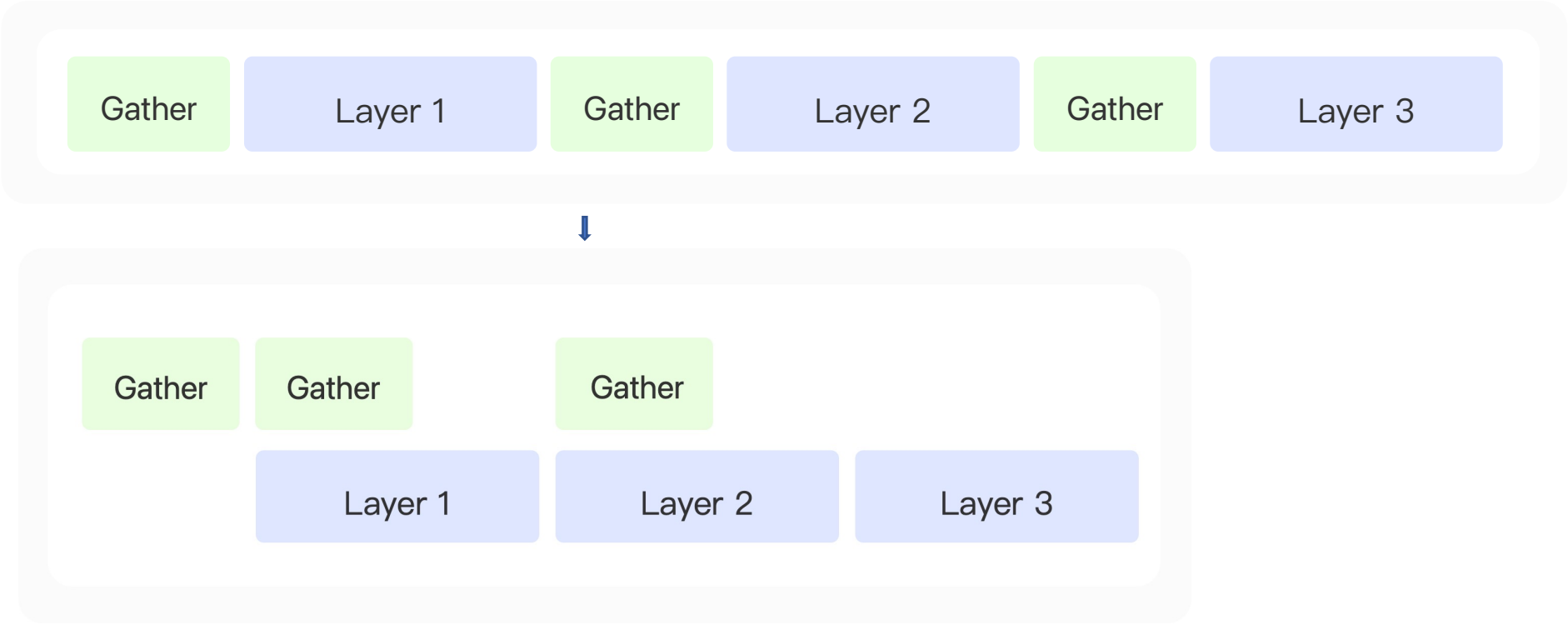
ZeRO-Offload: Democratizing Billion-Scale Model Training.

Techniques

1. Mixed precision
2. Offloading
3. Overlapping
4. Checkpointing

Overlapping

1. Memory operations are asynchronous.
2. Thus, we can overlap Memory operations with Calculations.



Techniques

1. Mixed precision
2. Offloading
3. Overlapping
4. Checkpointing

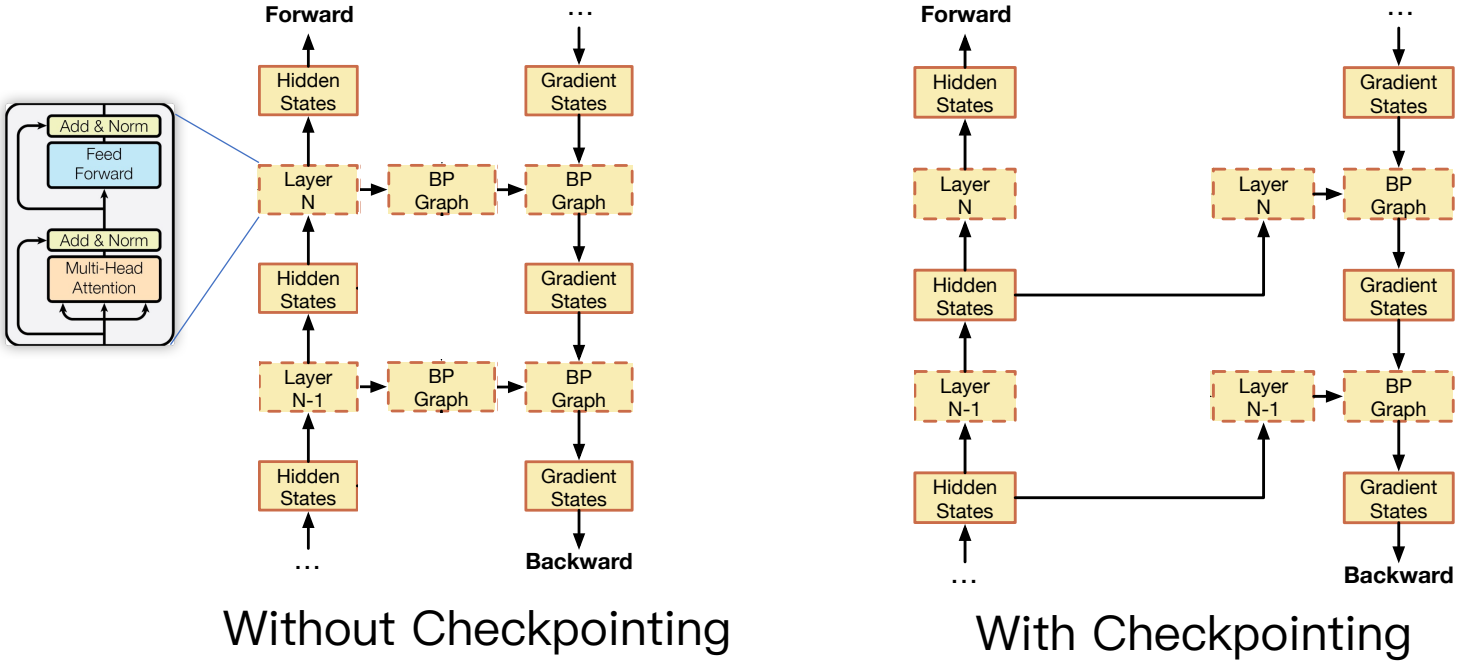
Checkpointing

Forward:

- Some hidden states (checkpoint) are reserved.
- All other intermediate results are immediately freed.

Backward:

- Freed intermediates are recomputed.
- And released again after obtaining gradient states.



Training Deep Nets with Sublinear Memory Cost.

Performance

Speedup.

Scene	Batch Size	Toolkit	Server Number (8 V100 Each)	Step Time	Speedup Estimation
10B Model Training	1024	DeepSpeed+Megatron	16	30s	600%
		BMTrain	4	20s	

Usage

Simple replacement.

```python
import torch
class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.param = torch.nn.Parameter(torch.empty(1024))
        self.module_list = torch.nn.ModuleList([
            SomeTransformerBlock(),
            SomeTransformerBlock(),
            SomeTransformerBlock()
        ])

    def forward(self):
        x = self.param
        for module in self.module_list:
            x = module(x, 1, 2, 3)
        return x
```

```python
import torch
import bmtrain as bmt
class MyModule(bmt.DistributedModule):
    def __init__(self):
        super().__init__()
        self.param = bmt.DistributedParameter(torch.empty(1024))
        self.module_list = bmt.TransformerBlockList([
            bmt.CheckpointBlock(SomeTransformerBlock()),
            bmt.CheckpointBlock(SomeTransformerBlock()),
            bmt.CheckpointBlock(SomeTransformerBlock())
        ])

    def forward(self):
        x = self.param
        x = self.module_list(x, 1, 2, 3)
        return x
```

Demo

link

BMCook

THUNLP

Growing Size of PLMs

- The model size of PLMs has been growing at a rate of about 10x per year

Date	Organization	Name	Model Size	Data Size	Time
2018.6	OpenAI	GPT	110M	4GB	3 Days
2018.10	Google	BERT	330M	16GB	50 Days
2019.2	OpenAI	GPT-2	1.5B	40GB	200 Days
2019.7	Facebook	RoBERTa	330M	160GB	3 Years
2019.10	Google	T5	11B	800GB	66 Years
2020.6	OpenAI	GPT-3	175B	2TB	355 Years

Time is evaluated on a single NVIDIA V100.

Huge Computational Cost

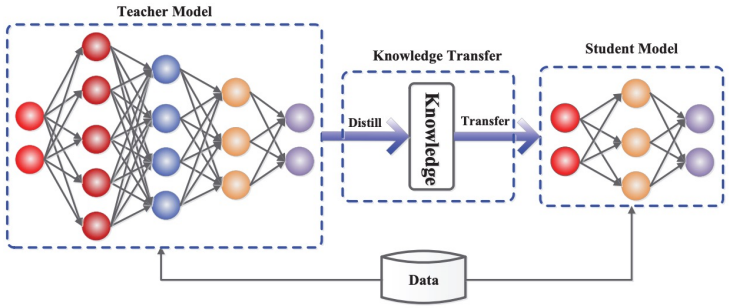
- The growing size comes with huge computational overhead
  - Limits the application of large PLMs in real-world scenarios
  - Leads to large carbon emissions

Consumption	**CO_2e (lbs)**
Air travel, 1 passenger, NY↔SF	1984
Human life, avg, 1 year	11,023
American life, avg, 1 year	36,156
Car, avg incl. fuel, 1 lifetime	126,000
Training one model (GPU)	
NLP pipeline (parsing, SRL)	39
w/ tuning & experimentation	78,468
Transformer (big)	192
w/ neural architecture search	626,155

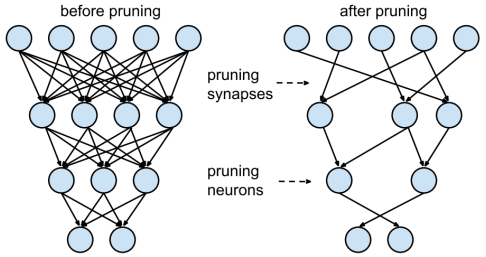
Strubell et al. Energy and Policy Considerations for Deep Learning in NLP. ACL 2019.

Towards Efficient PLMs

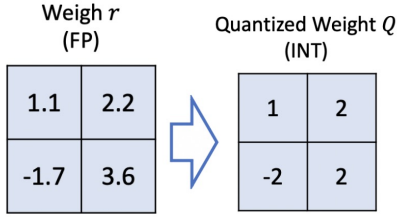
- Model Compression
  - Compress big models to small ones to meet the demand of real-world scenarios
- Existing Methods



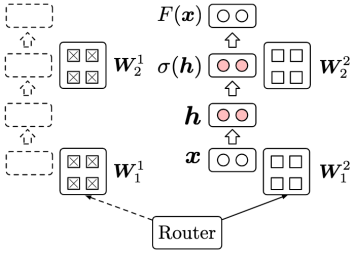
Knowledge Distillation



Model Pruning



Model Quantization



Model MoEfication

Guo et al. Knowledge Distillation: A Survey. 2021.
Han et al. Learning both Weights and Connections for Efficient Neural Networks. 2015.
Gholami et al. A Survey of Quantization Methods for Efficient Neural Network Inference. 2021.

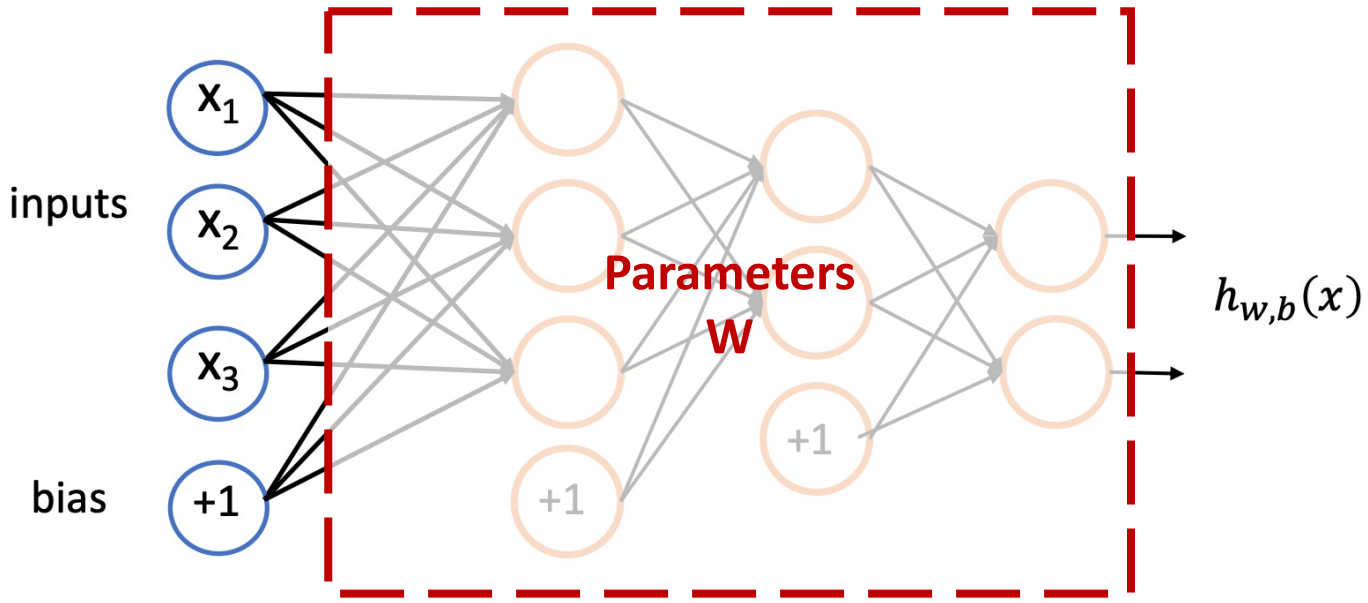
Knowledge Distillation

- Proposed by Hinton on NIPS 2014 Deep Learning Workshop
- Problem of Ensemble Model
  - Cumbersome and may be too computationally expensive
  - Similar to current PLMs
- Solution
  - The knowledge acquired by a large ensemble of models can be transferred to a single small model
  - We call “distillation” to transfer the knowledge from the **cumbersome model** to a **small model** that is more suitable for deployment.

Hinton et al. Distilling the Knowledge in a Neural Network. 2014.

Knowledge Distillation

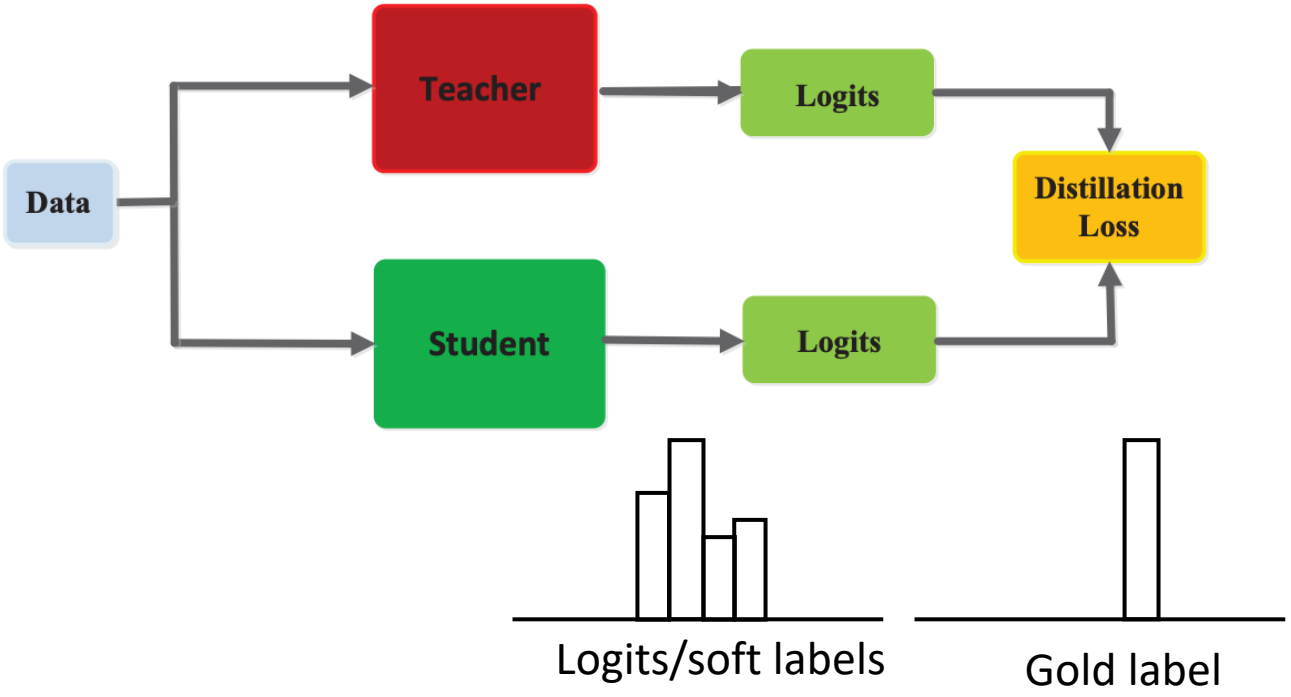
- What is knowledge?



- In a more abstract view, knowledge is a learned mapping from input vectors to output vectors

Knowledge Distillation

- Soft targets provide more information than gold labels



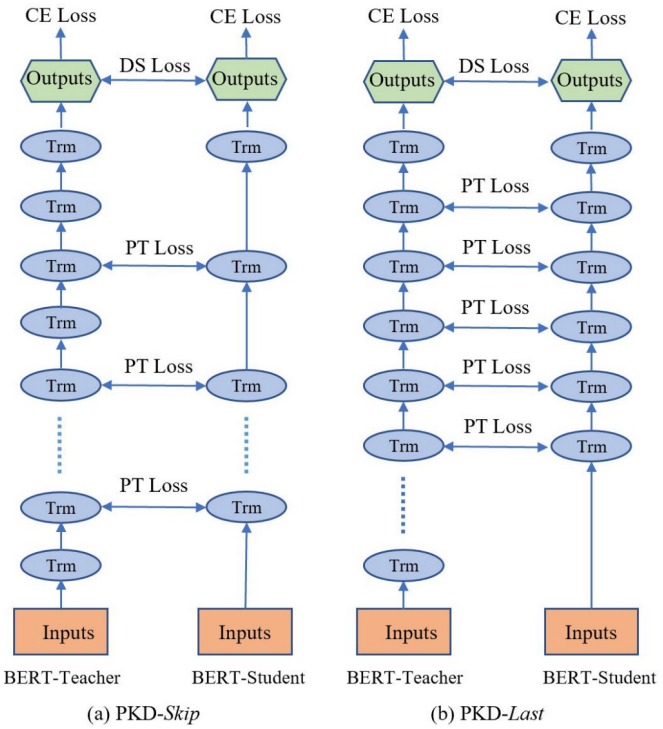
- **Key research question**: how to build more soft targets
  - Previous methods only use the output from the last layer

Guo et al. Knowledge Distillation: A Survey. 2021.

Knowledge Distillation

- Learn from **multiple intermediate layers** of the teacher model
- Mean-square loss between the normalized hidden states

$$L_{PT} = \sum_{i=1}^{N}\sum_{j=1}^{M}\left|\left|\frac{\mathbf{h}_{i,j}^{s}}{||\mathbf{h}_{i,j}^{s}||_2} - \frac{\mathbf{h}_{i,I_{pt}(j)}^{t}}{||\mathbf{h}_{i,I_{pt}(j)}^{t}||_2}\right|\right|_2^2$$

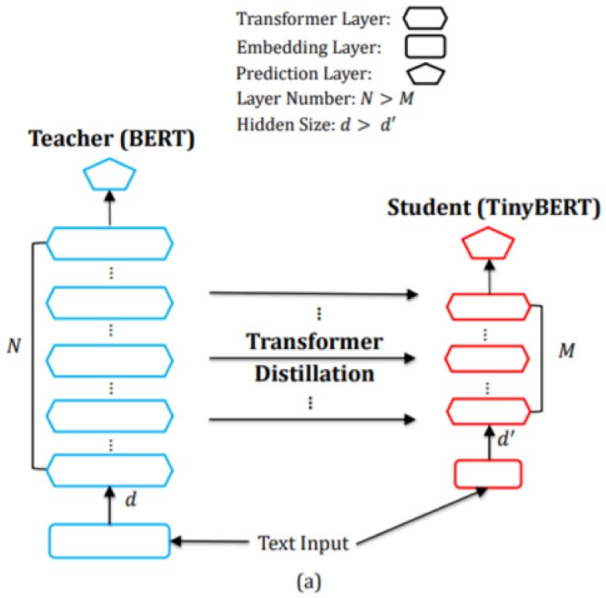


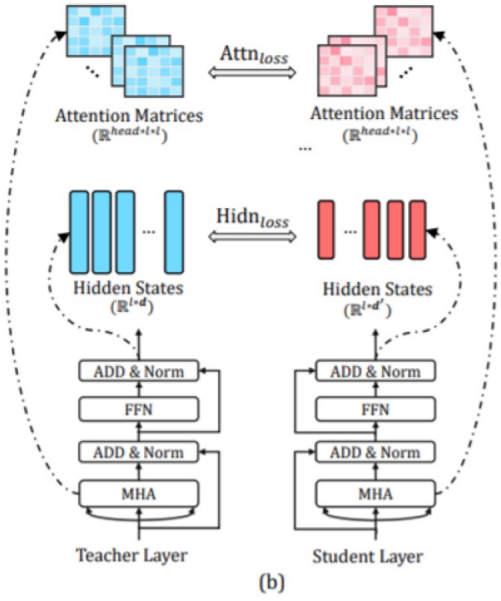
Sun et al. Patient Knowledge Distillation for BERT Model Compression. EMNLP 2019.

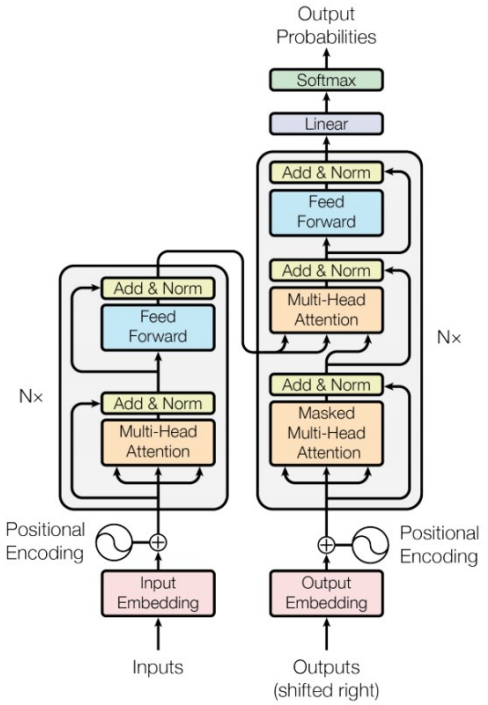
Knowledge Distillation

- Learn from multiple intermediate layers
- Learn from the embedding layer and output layer
- Learn from attention matrices

$$\mathcal{L}_{layer}(S_m, T_{g(m)}) = \begin{cases} \mathcal{L}_{embd}(S_0, T_0), & m = 0 \\ \mathcal{L}_{hidn}(S_m, T_{g(m)}) + \mathcal{L}_{attn}(S_m, T_{g(m)}), & M \geq m > 0 \\ \mathcal{L}_{pred}(S_{M+1}, T_{N+1}), & m = M + 1 \end{cases}$$





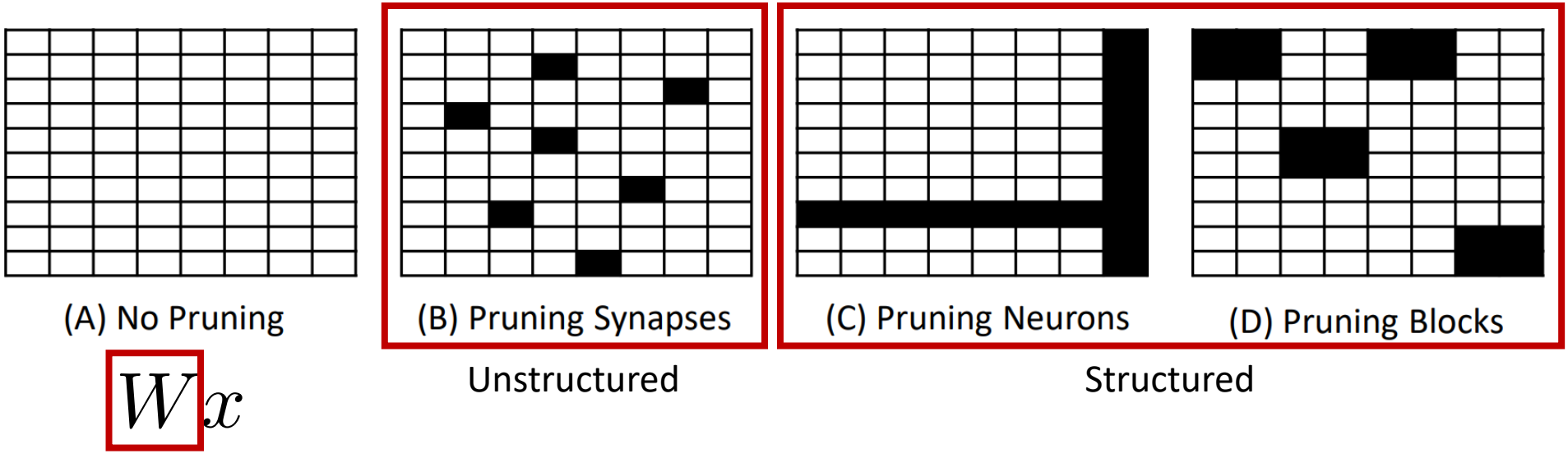


Transformer

Jiao et al. TinyBERT: Distilling BERT for Natural Language Understanding. Findings of EMNLP 2020.

Model Pruning

- Remove **the redundant parts** of the parameter matrix according to their important scores
- Unstructured pruning and structured pruning

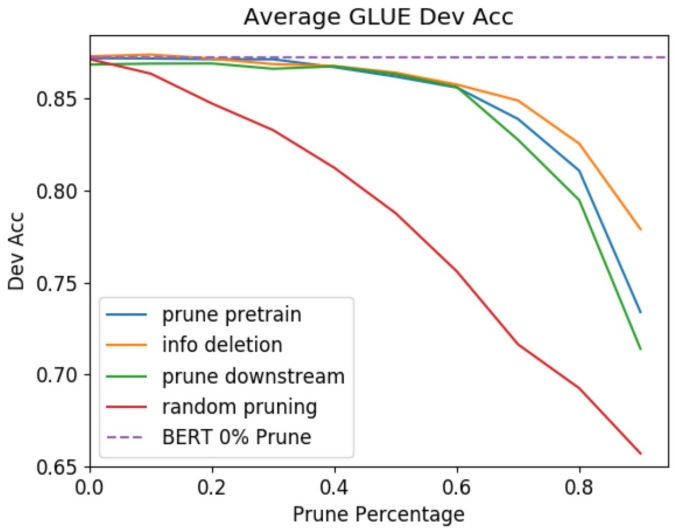


$$Wx$$

Gupta et al. Compression of Deep Learning Models for Text: A Survey. TKDD 2020.

Model Pruning

- Weight pruning (unstructured)

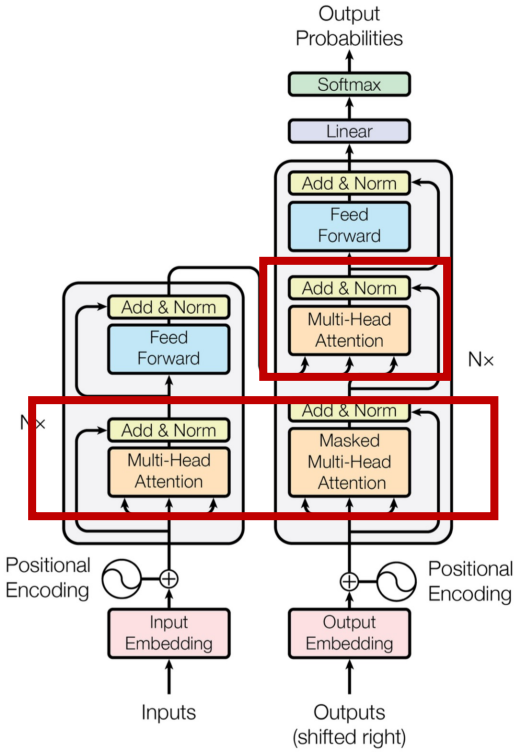


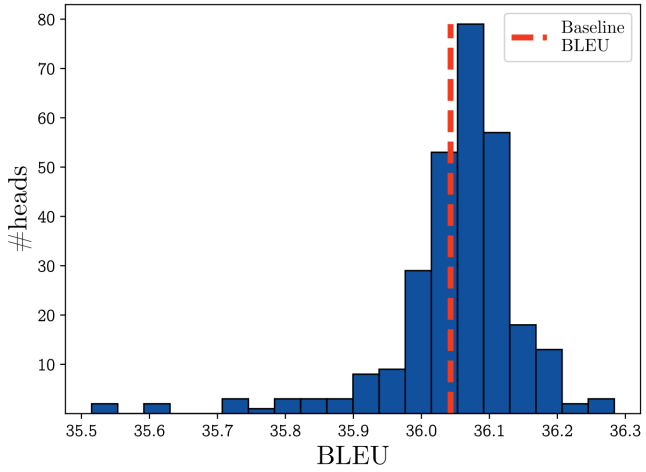
- 30-40% of the weights can be discarded without affecting BERT's universality (prune pre-train)
- Fine-tuning on downstream tasks does not change the nature (prune downstream)

Gordon et al. Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning. RL4NLP 2020.

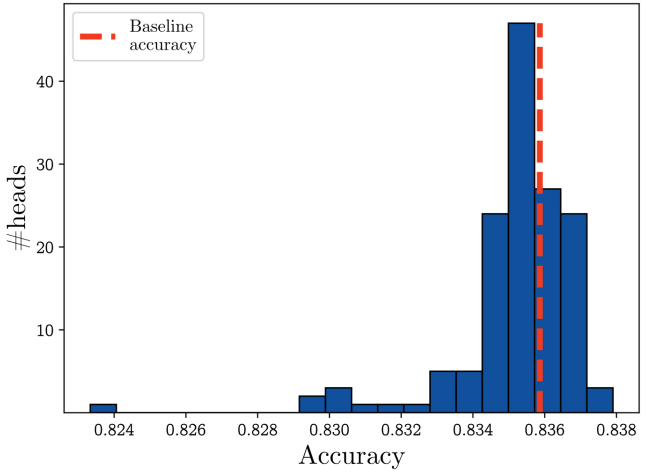
Model Pruning

- Attention head pruning (structured)
- Ablating one head





(a) WMT



(b) BERT

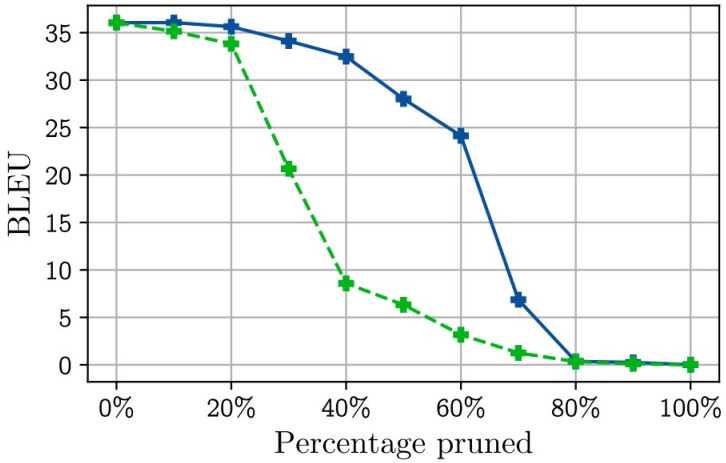
Michel et al. Are Sixteen Heads Really Better than One?. NeurIPS 2019.

Model Pruning

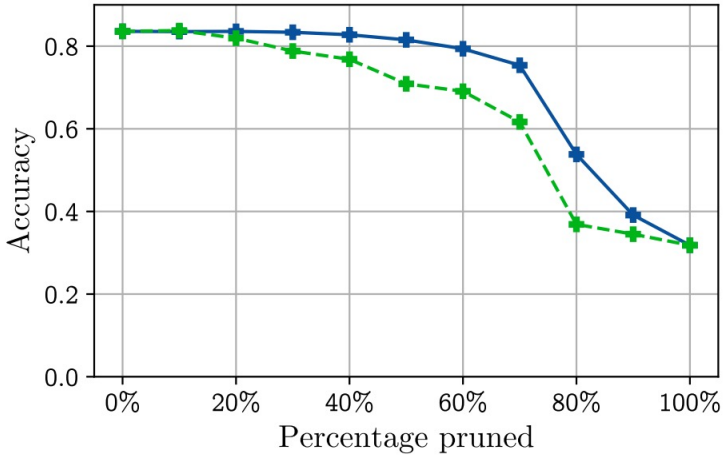
- Attention head pruning (structured)
- Define the importance scores of attention heads

$$I_h = \mathbb{E}_{x \sim X} \left| \mathrm{Att}_h(x)^T \frac{\partial \mathcal{L}(x)}{\partial \mathrm{Att}_h(x)} \right|$$

- Iteratively prune heads on different models (blue line)



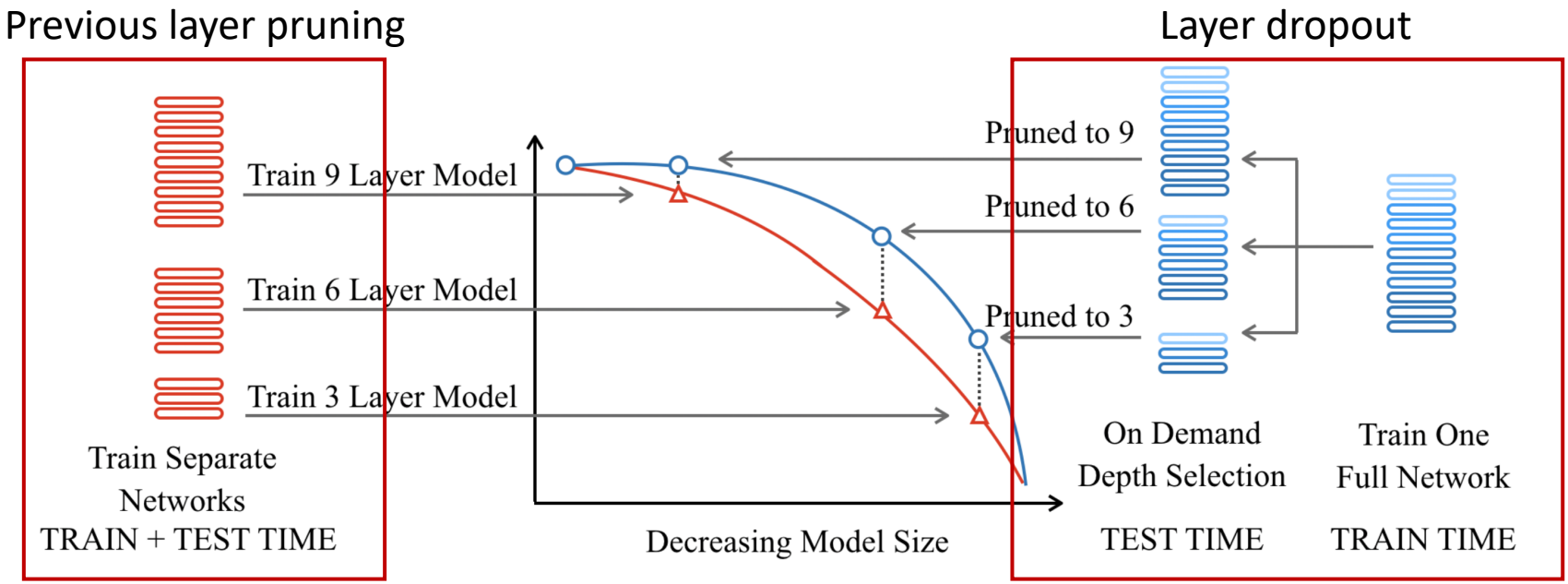
(a) Evolution of BLEU score on `newstest2013` when heads are pruned from WMT.



(b) Evolution of accuracy on the MultiNLI-matched validation set when heads are pruned from BERT.

Model Pruning

- Layer pruning (structured)
- Extend dropout from weights to layers
- Training: randomly drop layers
- Test: Select sub-networks with any desired depth



Fan et al. Reducing Transformer Depth on Demand with Structured Dropout. ICLR 2020.

Model Quantization

- Reduce **the number of bits** used to represent a value
  - Floating point representation -> Fixed point representation
- Three steps
  - Linear scaling

$$\mathrm{sc}(x) = \frac{x - \beta}{\alpha} \qquad \alpha = w_{\max} - w_{\min} \text{ and } \beta = w_{\min}$$

  - Quantize

$$\hat{x} = \frac{1}{2^k - 1} \mathrm{round}\left(\left(2^k - 1\right) \cdot \mathrm{sc}(x) \right)$$

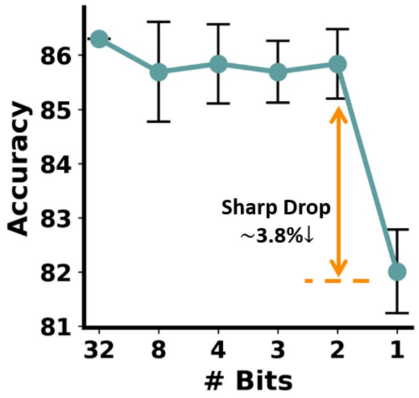
  - Scaling back

$$Q(x) = \alpha \cdot \hat{x} + \beta$$

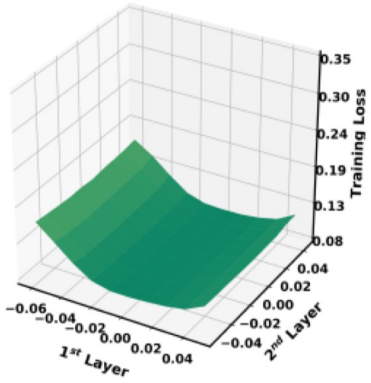
Example	
Float	8 bit Quantized
-10.0(min)	0
30.0(max)	255
10.0	128

Model Quantization

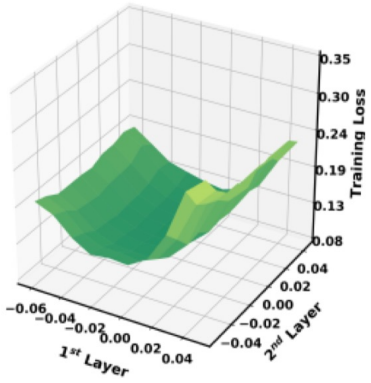
- Models with different precisions
  - Extreme quantization (1 bit) is difficult



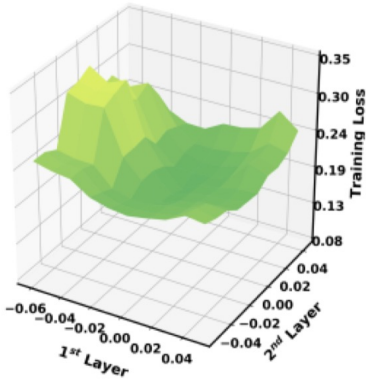
- Loss landscapes are sharper



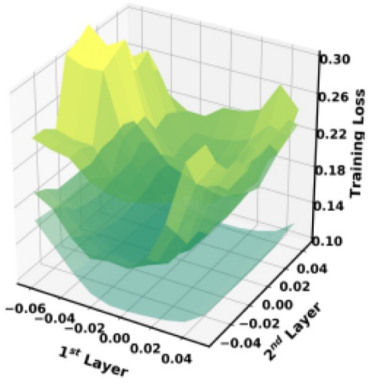
(a) Full-precision Model.



(b) Ternary Model.



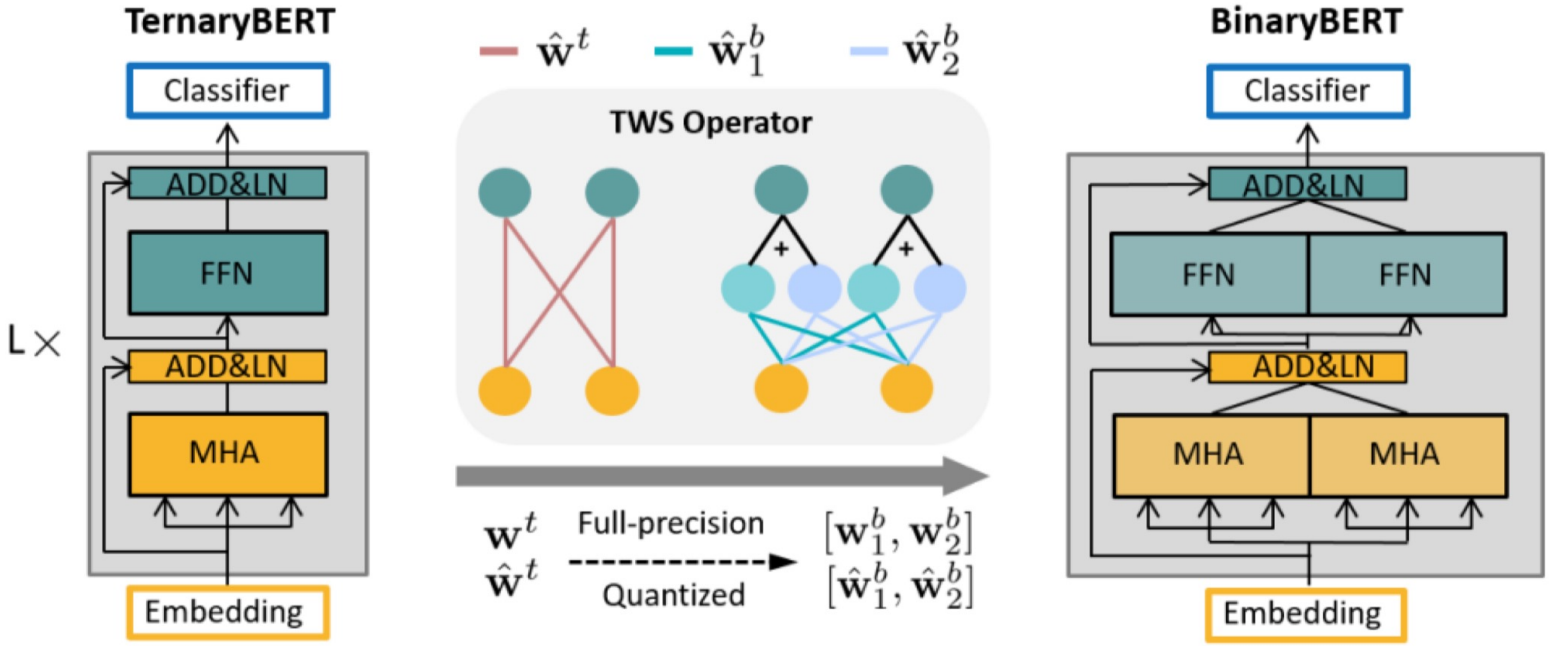
(c) Binary Model.



(d) All Together.

Model Quantization

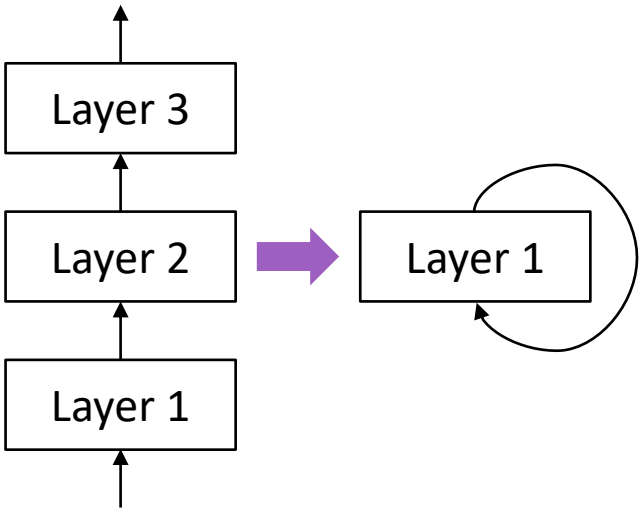
- Train a half-sized ternary model
- Initialize a binary model with the ternary model by weight splitting
- Fine-tune the binary model



Bai et al. BinaryBERT: Pushing the Limit of BERT Quantization. ACL 2021.

Other Methods: Weight Sharing

- ALBERT: Two parameter reduction techniques
  - Decompose the large vocabulary embedding matrix into two small matrices
  - Cross-layer parameter sharing



Model		Parameters	SQuAD1.1	SQuAD2.0	MNLI	SST-2	RACE	Avg	Speedup
BERT	base	108M	90.4/83.2	80.4/77.6	84.5	92.8	68.2	82.3	4.7x
	large	334M	92.2/85.5	85.0/82.2	86.6	93.0	73.9	85.2	1.0
ALBERT	base	12M	89.3/82.3	80.0/77.1	81.6	90.3	64.0	80.1	5.6x
	large	18M	90.6/83.9	82.3/79.4	83.5	91.7	68.5	82.4	1.7x
	xlarge	60M	92.5/86.1	86.1/83.1	86.4	92.4	74.8	85.5	0.6x
	xxlarge	235M	**94.1/88.3**	**88.1/85.1**	**88.0**	**95.2**	**82.3**	**88.7**	0.3x

Lan et al. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. ICLR 2020.

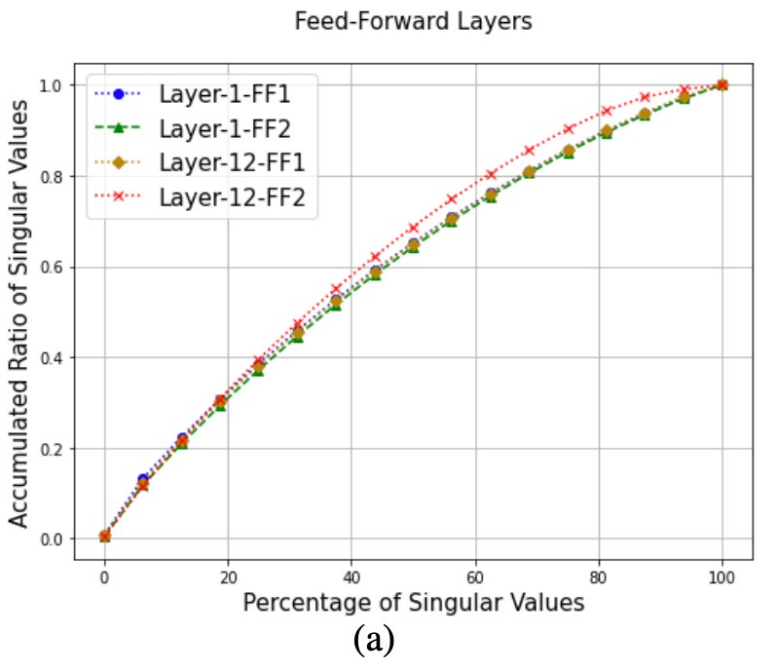
Other Methods: Low-rank Approximation

- Low-rank Approximation

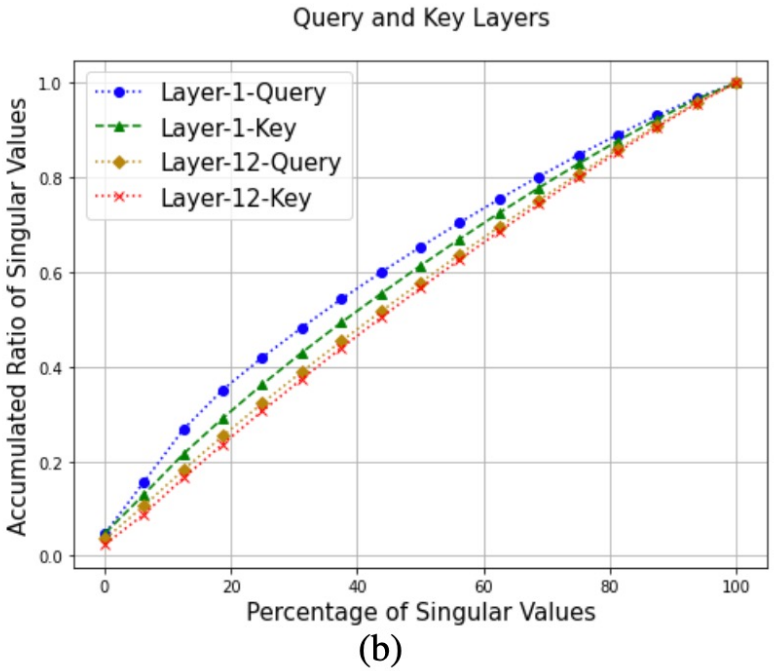
$$D = U\Sigma V^\top \in \mathbb{R}^{m\times n}, \quad m \geq n \qquad \Sigma =: \mathrm{diag}(\sigma_1, \ldots, \sigma_m)$$

$$\widehat{D}^* = U_1 \Sigma_1 V_1^\top$$

- Difficult to directly conduct low-rank approximation



(a)



(b)

Other Methods: Low-rank Approximation

- Decompose the input matrix

$$h = WX + b \approx W\boxed{U_{x,k}{V_{x,k}}^T X} + b$$
$$= (WU_{x,k}){V_{x,k}}^T X + b = W_{X,k}{V_{x,k}}^T X + b,$$

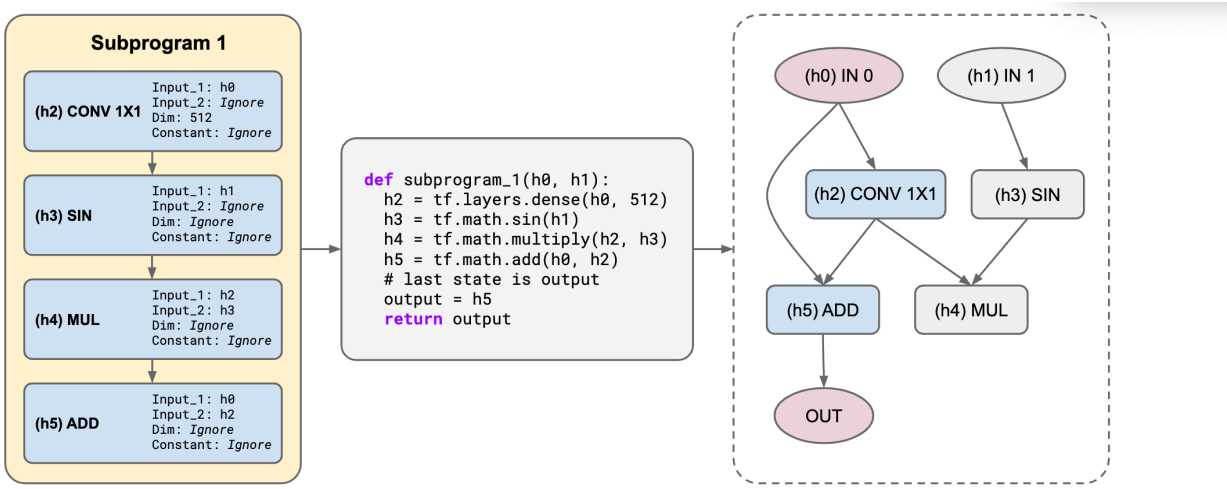
- Solve this equation: $\min_M \|WX - WMX\|_F^2, \quad \text{s.t.} \quad \text{rank}(M) = k,$
- Experimental results

Methods	MNLI	QQP	SST-2	QNLI	MRPC	RTE	CoLA	STS-B
Original	84.3	90.9	92.3	91.4	89.5	72.6	53.4	87.8
SVD	74.4	50.8	73.1	52.2	63.8	47.3	12.4	33.6
DRONE	82.0	89.4	90.0	88.5	86.7	70.0	52.5	85.8
DRONE-Retrain	82.6	90.1	90.8	89.3	88.0	71.5	53.2	87.8
CPU Speedup Ratio	1.60x	1.25x	1.64x	1.20x	1.92x	1.31x	1.33x	1.52x
GPU Speedup Ratio	1.28x	1.38x	1.45x	1.28x	1.56x	1.33x	1.29x	1.57x

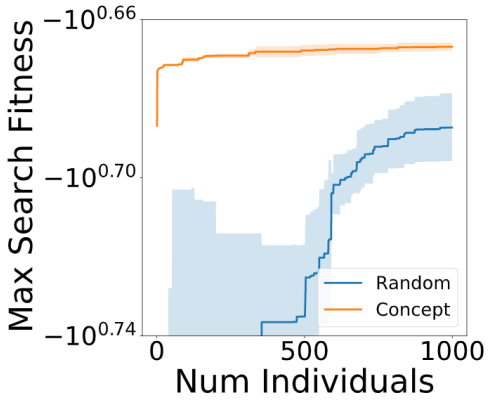
Chen et al. DRONE: Data-aware Low-rank Compression for Large NLP Models. NeurIPS 2021.

Other Methods: Architecture Search

- Is the architecture of Transformer perfect?
- Neural architecture search based on Transformer
  - Pre-define several simple modules
  - Training several hours with each architecture



Combine simple modules



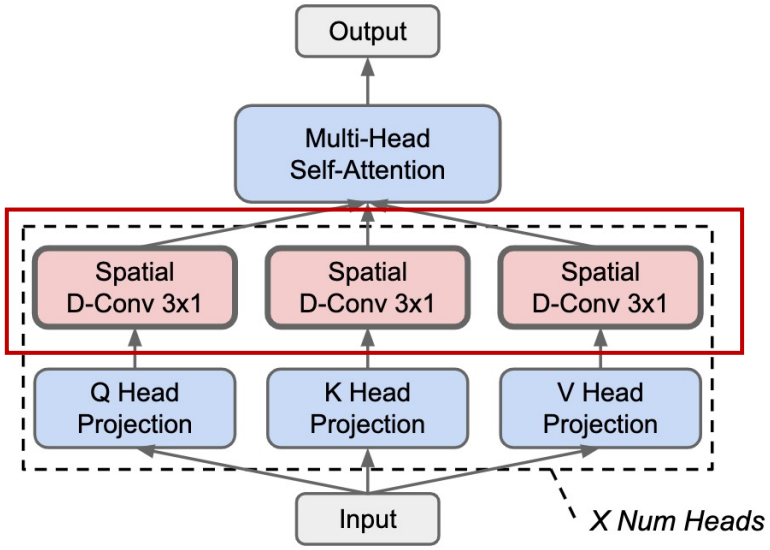
Search with/without prior knowledge

So et al. Primer: Searching for Efficient Transformersfor Language Modeling. NeurIPS 2021.

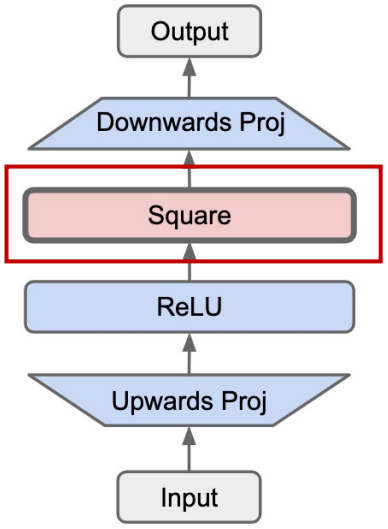
Other Methods: Architecture Search

- Two effective modifications

Multi-DConv-Head Attention (MDHA)



Squared ReLU in Feed Forward Block



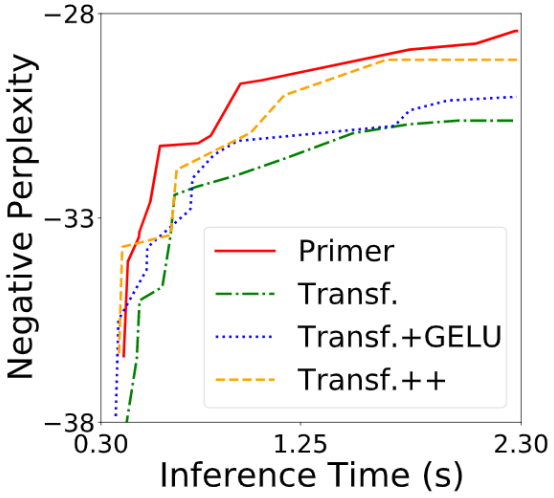
MDHA Projection Pseudo-code

```
# Use to create each K, Q,
     and V head of size 'hs'.
def mdha_projection(x, hs):
  # Create head.
  x = proj(x,
           head_size=hs,
           axis="channel")

  # Apply D—Conv to head.
  x = d_conv(x,
             width=3,
             head_size=hs,
             axis="spatial",
             mask="causal")
  return x
```

Other Methods: Architecture Search

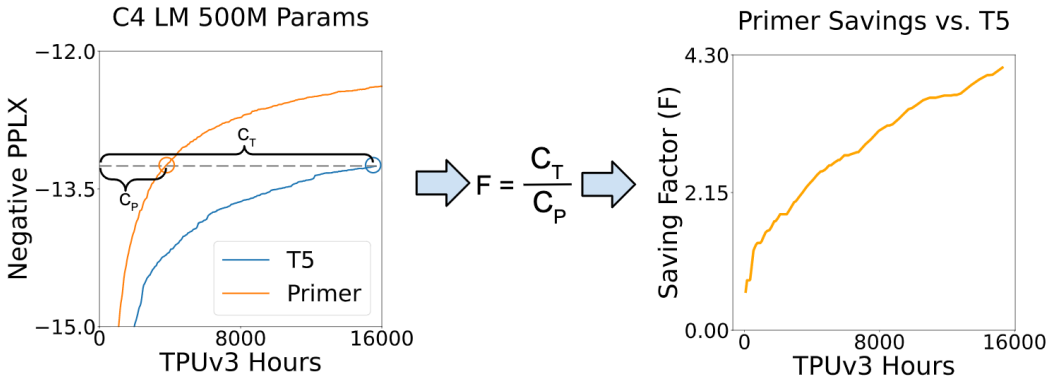
- Primer learns faster and better



Decoder

Model	Steps	TPUv3 Hours	PPLX
Original T5	1M	15.7K	13.25
T5++	251K	4.6K	13.25
Primer	207K	**3.8K**	13.25
T5++	1M	16.5K	12.69
Primer	480K	**8.3K**	12.69
Primer	1M	17.3K	12.35

Faster to reach the same PPLX



Encoder-Decoder

Summary

- Large-scale PLMs are extremely **over-parameterized**
- Several methods to improve model efficiency
  - Knowledge Distillation
  - Model Pruning
  - Model Quantization
  - …
- Our model compression toolkit: BMCook
  - Includes these methods for extreme acceleration of big models

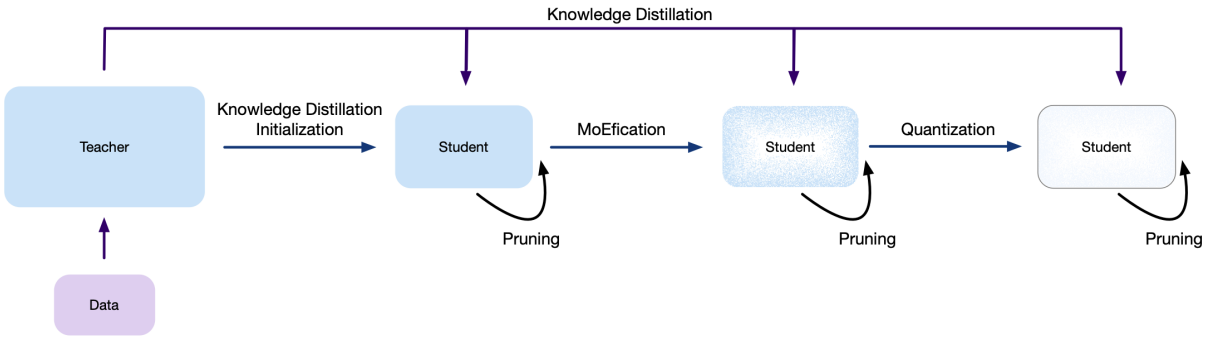
https://github.com/OpenBMB/BMCook

BMCook

- Compared to existing compression toolkits, BMCook supports all mainstream acceleration methods for PLMs

	Model Quantization	Model Pruning	Knowledge Distillation	Model MoEfication
TextPruner	-	✅	-	-
TensorFlow Lite	✅	✅	-	-
PyTorch	✅	✅	-	-
TextBrewer	-	✅	✅	-
BMCook	✅	✅	✅	✅

- Implement different compression methods with just a few lines of codes
- Compression methods can be combined in any way towards extreme acceleration



BMCook

- Core of BMCook: Compression Configuration File
- Implement various methods with few lines

```
# configuration file
config = ConfigParser('configs/bmcook.json')

# Distillation
Trainer.forward = BMDistill.set_forward(model, teacher, Trainer.forward, config)

# Pruning
BMPrune.compute_mask(model, config)
BMPrune.set_optim_for_pruning(optimizer)

# Quantization
BMQuant.quantize(model, config)

# MoEfication
Trainer.forward = BMMoE.get_hidden(model, config, Trainer.forward)
```

BMCook

- Configuration for distillation
- Support MSE loss and CE loss

```
"distillation": {
    "ce_scale": 0,

    "mse_hidn_scale": 1,
    "mse_hidn_module": [
      "[post]encoder.output_layernorm:[post]encoder.output_layernorm",
      "[pre]encoder.layers.9.ffn.layernorm_before_ffn:
[pre]encoder.layers.9.ffn.layernorm_before_ffn"],
    "mse_hidn_proj": false
},
```

BMCook

- Configuration for pruning
- Support unstructured pruning

```
"pruning": {
    "is_pruning": true,
    "pruning_mask_path": "prune_example.bin",
    "pruned_module": ["ffn.ffn.w_in.w.weight", "ffn.ffn.w_out.weight", "input_embedding"],
    "mask_method": "m4n2_1d"
}
```

- Configuration for quantization
- Replace all linear modules

```
"quantization": {
    "is_quant": true
},
```

BMInf

THUNLP

Introduction to BMInf

BMInf is the first toolkit released by OpenBMB.

Github repo: https://github.com/OpenBMB/BMInf

BMInf has received 270 stars (hope more after this course XD).

BMInf → **BM**Train → **BM**Cook

Sep 2021 Feb 2022 Mar 2022

Background

In June 2021, we released CPM-2 with 10 billion parameters.

It is powerful in many downstream tasks.

	CCPM	C^3	Sogou-Log	WMT20	Math23K	LCSTS	LCQMC	AdGen	CUGE
	Acc	Acc	MRR/NDCG	BLEU	Acc	Rouge-L	Acc	BLEU/Distinct	Score
mT5-small	$87.7_{(100)}$	$41.5_{(100)}$	$29.2/29.2_{(100)}$	$9.1_{(100)}$	$18.4_{(100)}$	$33.1_{(100)}$	$82.1_{(100)}$	$10.2/32.3_{(100)}$	100
mT5-large	$89.9_{(102)}$	$56.3_{(136)}$	$32.2/31.1_{(108)}$	$11.1_{(122)}$	$34.3_{(186)}$	$34.4_{(104)}$	$85.0_{(104)}$	$10.0/35.5_{(104)}$	126
mT5-XXL	$90.6_{(103)}$	$\mathbf{86.4}_{(\mathbf{208})}$	$\mathbf{36.9}/34.9_{(\mathbf{123})}$	$24.0_{(264)}$	$61.6_{(335)}$	$34.8_{(105)}$	$88.3_{(108)}$	$9.8/68.7_{(154)}$	190
CPM-2	$\mathbf{91.6}_{(\mathbf{104})}$	$86.1_{(207)}$	$36.3/\mathbf{35.5}_{(\mathbf{123})}$	$\mathbf{26.2}_{(\mathbf{288})}$	$\mathbf{69.4}_{(\mathbf{377})}$	$\mathbf{35.9}_{(\mathbf{108})}$	$\mathbf{89.2}_{(\mathbf{109})}$	$\mathbf{10.6/70.2}_{(\mathbf{161})}$	**198**

Background

In June 2021, we released CPM-2 with 10 billion parameters.

It is powerful in many downstream tasks.

Background

- high hardware requirements
  - For each demo we used 4xA100s for inference.
- inefficient
  - Each request takes about 10 seconds to handle.
- costly
  - The cost of 4xA100s is ¥1200 per day.

Background

Another thought:

serve demo on our server

make it possible for everyone
to run big models on their own computers

Background

What GPU does everyone use?

GTX 1060 seems to be the most popular one.

ALL VIDEO CARDS	FEB	MAR	APR	MAY	JUN	
NVIDIA GeForce GTX 1060	7.99%	8.18%	7.15%	7.17%	**7.02%**	-0.15%
NVIDIA GeForce GTX 1650	6.30%	6.08%	6.48%	6.45%	**6.51%**	+0.06%
NVIDIA GeForce GTX 1050 Ti	5.79%	5.53%	5.63%	5.62%	**5.36%**	-0.26%
NVIDIA GeForce RTX 2060	5.38%	5.62%	5.09%	5.08%	**5.03%**	-0.05%
NVIDIA GeForce RTX 3060 Laptop GPU	2.35%	2.22%	2.66%	2.68%	**2.99%**	+0.31%
NVIDIA GeForce GTX 1050	3.08%	2.87%	2.99%	2.85%	**2.81%**	-0.04%
NVIDIA GeForce GTX 1660 Ti	2.85%	2.71%	2.80%	2.81%	**2.66%**	-0.15%
NVIDIA GeForce RTX 3060	1.92%	2.48%	2.18%	2.36%	**2.62%**	+0.26%
NVIDIA GeForce GTX 1660 SUPER	2.37%	2.45%	2.37%	2.48%	**2.42%**	-0.06%

Background

What kind of GPU is the GTX 1060?

156 TFlops

40 GB

$ 10, 000

A100

4 TFlops

6 GB

$ 299

GTX 1060

Background

What if we could run big models on a single GTX 1060?

BMInf does this !

Difficulties

How difficult is it?

1. High Memory Footprint

   The checkpoint size of CPM-2 model is 22GB.

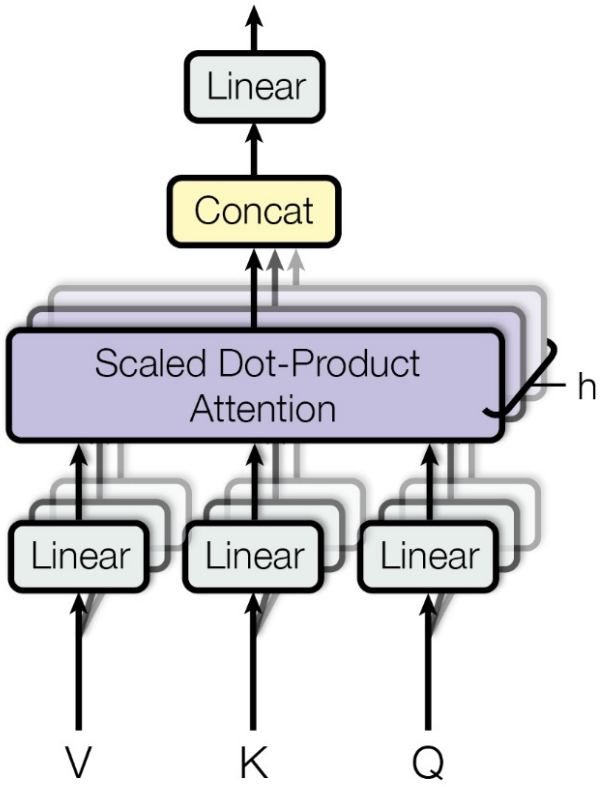
   It takes about 2 minutes to load the model from disk.

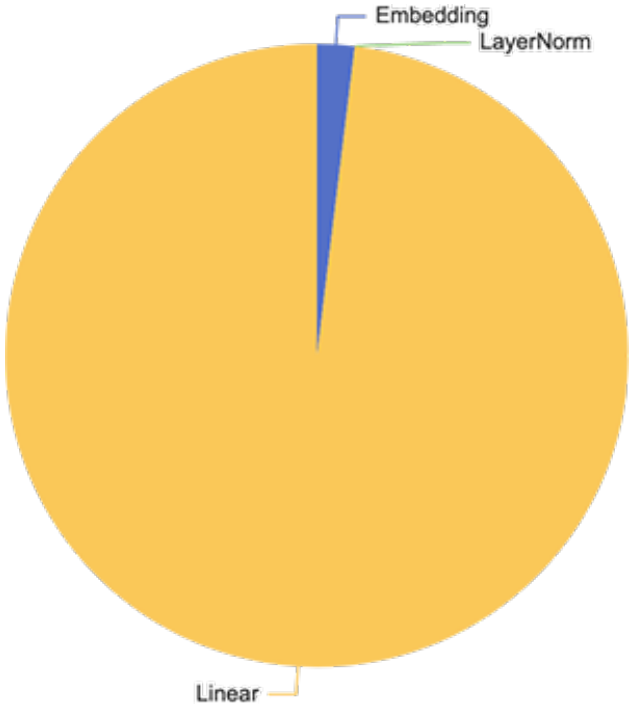
2. High Computing Power

   Generating 1 token with A100 takes 0.5 seconds.

Transformer

Dive into the Transformer structure:





Parameters in CPM-2

Linear Layer

The linear layer is actually matrix multiplication.

Input

0.1	-1.0	1.2
-0.5	0.7	-3.1
2.3	0.3	-0.5

Weight

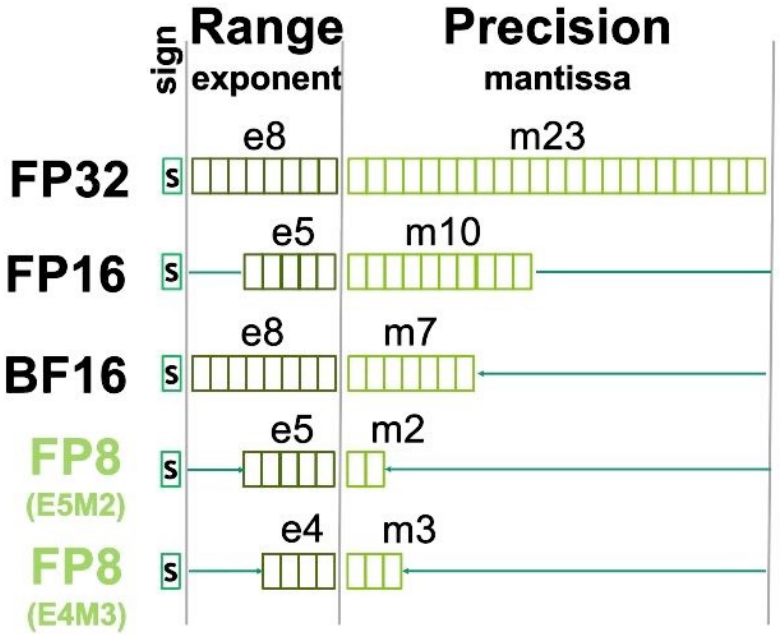
-2.0	-2.9	0.0
-1.4	-0.6	1.4
0.6	0.2	-0.4

Output

1.9	0.6	-1.9
-1.8	0.4	2.2
-5.3	-6.9	0.6

Linear Layer

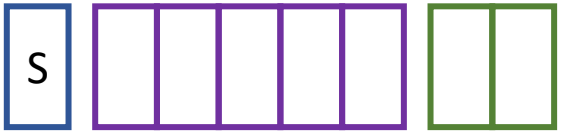
- using lower precision for speedup
- FP64 -> FP32 -> FP16 -> FP8? INT8



source: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

Int 8 vs FP 8

- INT 8
  - samller range
  - precise value

FP 8 

$6.10 \times 10^{-5} \sim 65504$

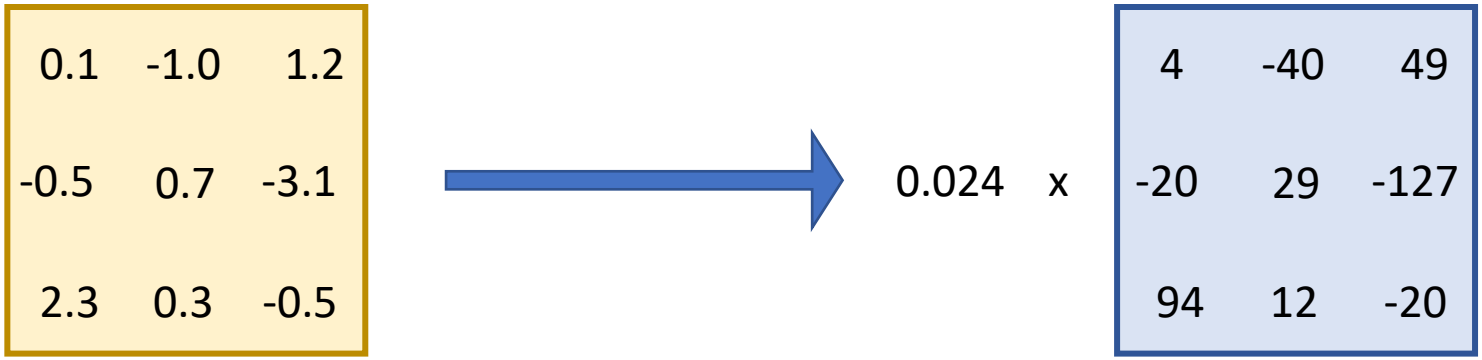
INT 8 

-128 ~ 127

Quantization

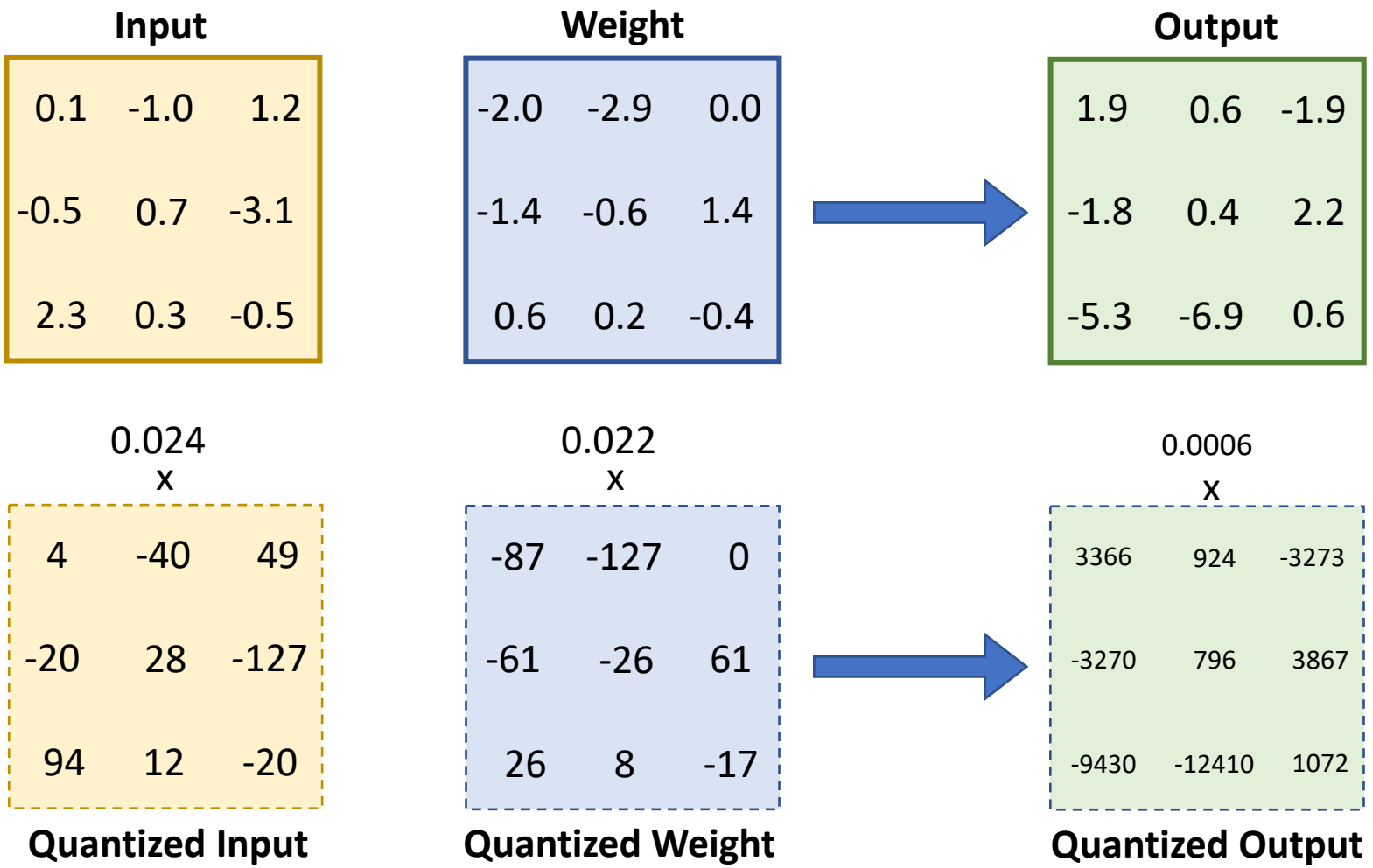
Using integers to simulate floating-point matrix multiplication

- find the largest value in the matrix
- scale to 127 for quantification
- multiply scaling factor for dequantization



Quantization

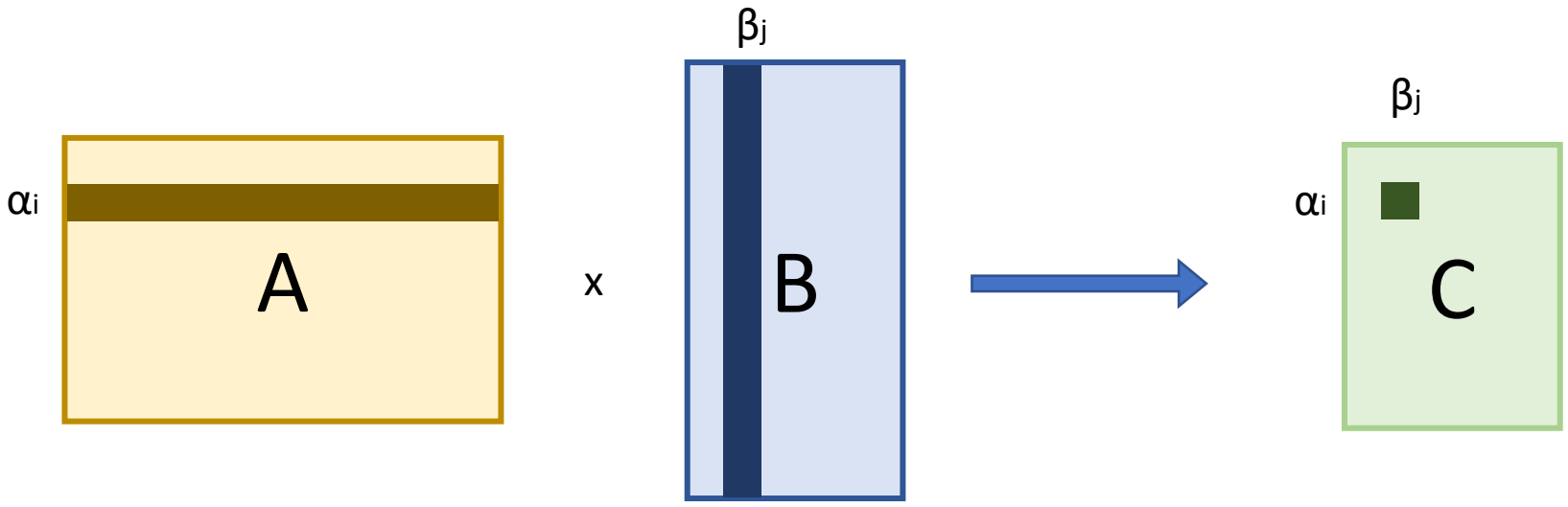
Matrix multiplication after quantization



Quantization

Row-wise matrix quantization:

- calculate the scaling factor for each row/column
- scale each row/column to -127~127



Quantization

We quantized the linear layer parameters of CPM-2.

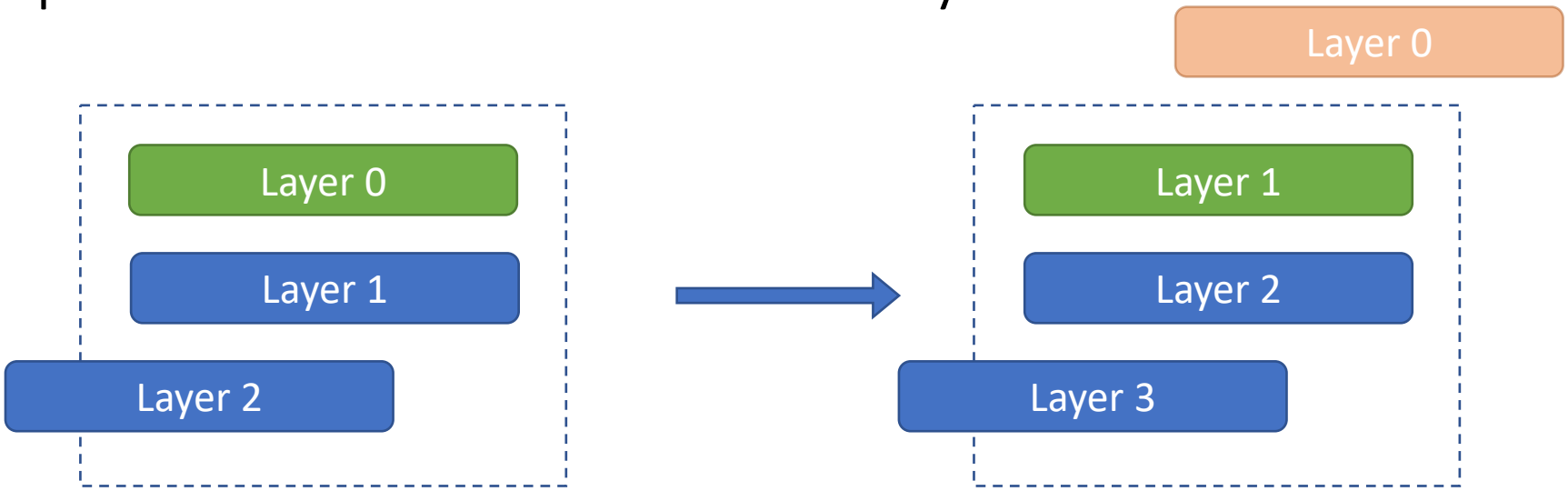
- model size is reduced by half
- 22GB -> 11 GB
- still too large for GTX 1060 (6GB memory)

Memory Scheduling

Not all parameters need to be placed on GPU.

- Move parameters that won't be used in a short time to CPU.
- Load parameters from CPU before use.
- Calculation and loading are performed in parallel.

Implemented in CUDA 6: Unified Memory

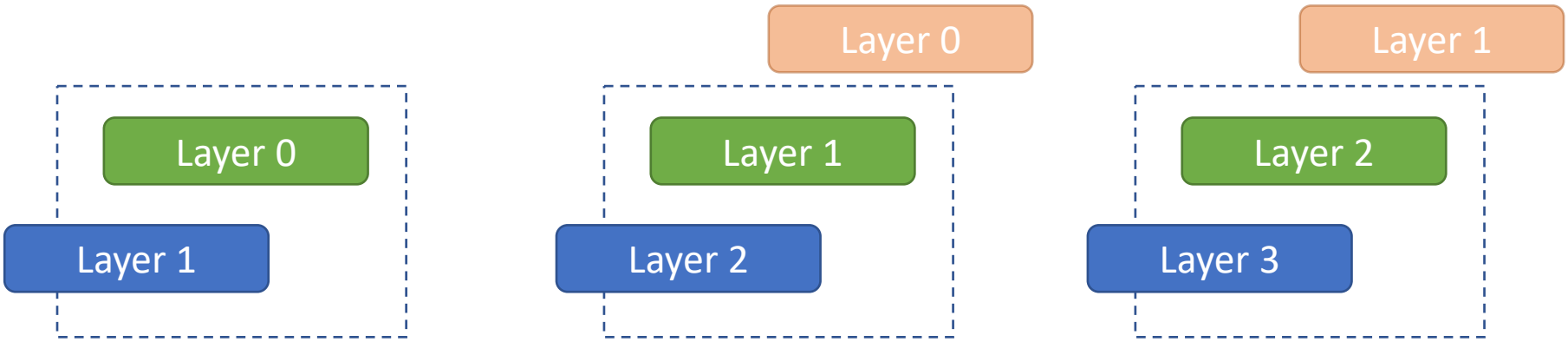


Memory Scheduling

We only need to store two layers of parameters in the GPU.

- one for calculating
- the other for loading

It's about 500MB for CPM-2 .



Memory Scheduling

In fact, it is much slower to load than to calculate.

- It takes a long time if we only place two layers on GPU.
- Put as many layers as possible on the GPU.

Assuming that up to **n** layers can be placed on the GPU.

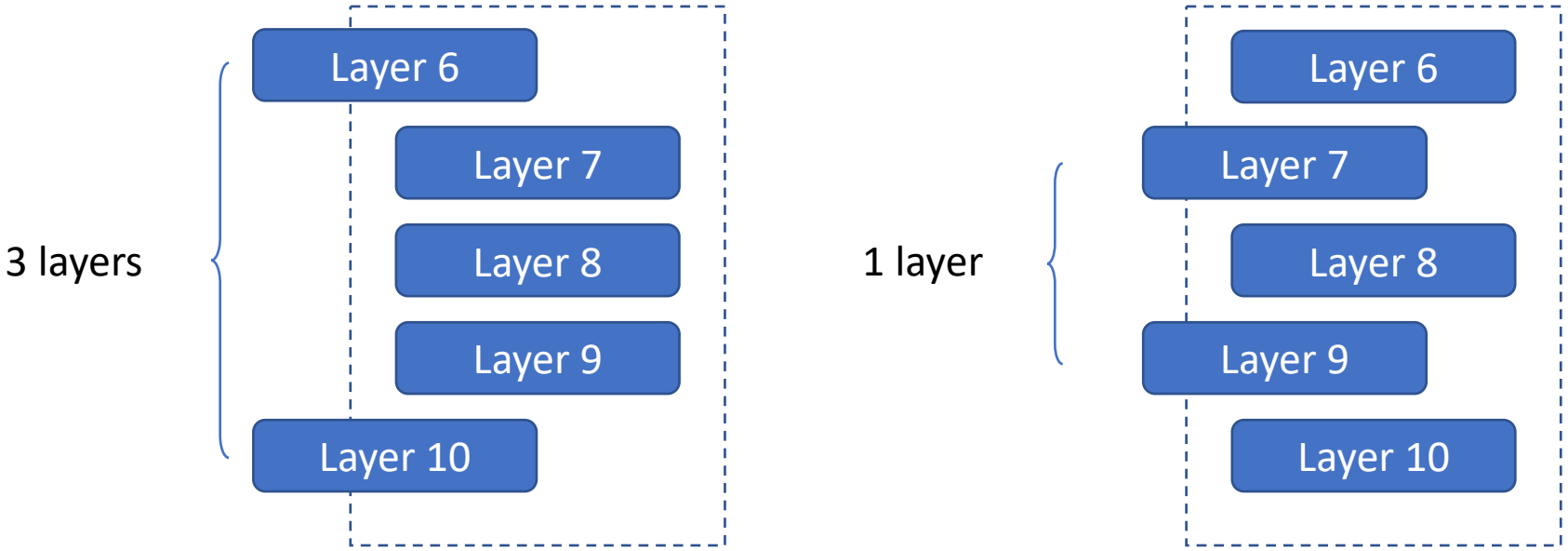
- **n - 2** layers are fixed on GPU that will not be moved to the CPU.
- **2** layers are used for scheduling.

Which layers are fixed on GPU?

Memory Scheduling

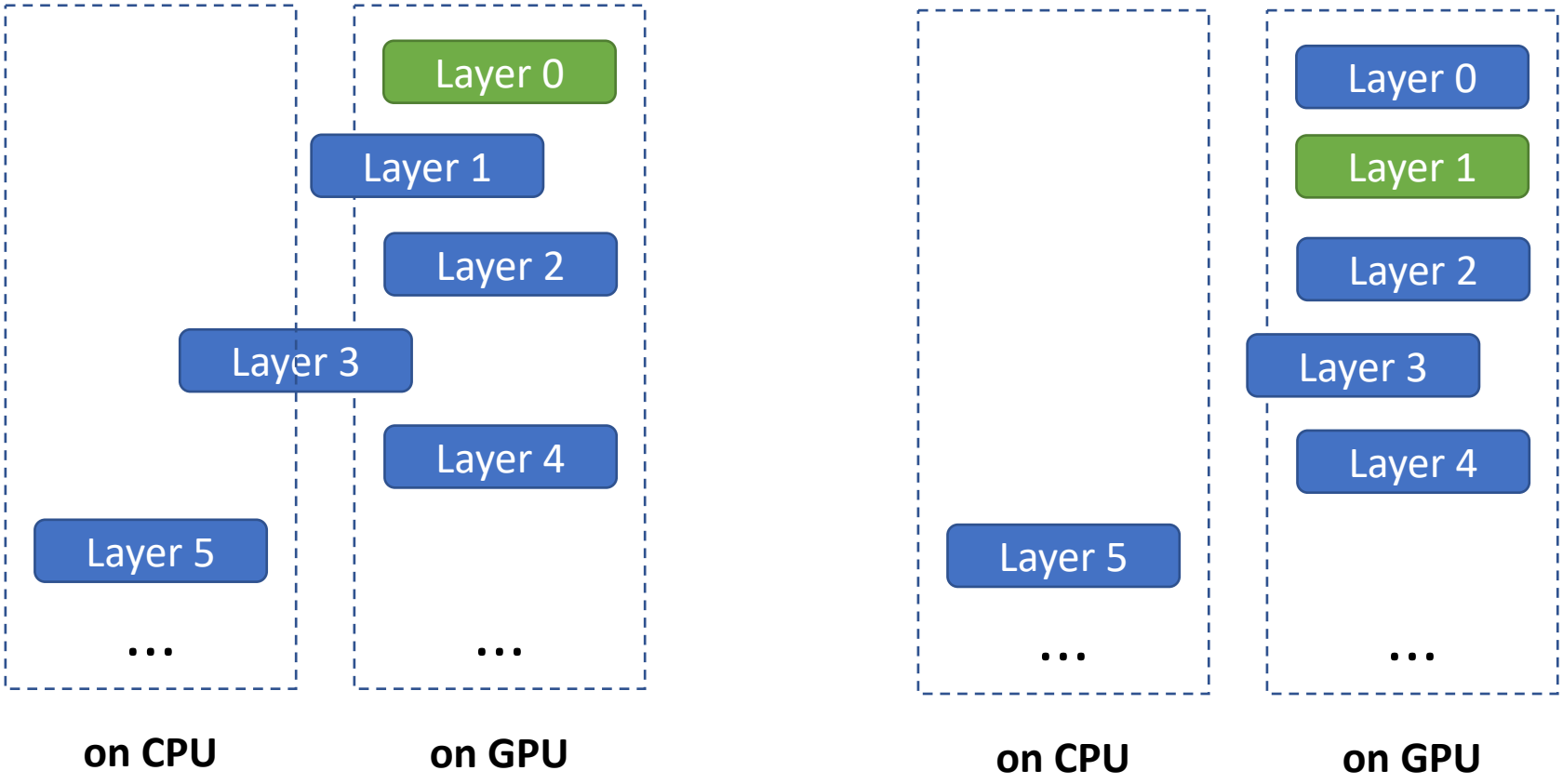
Consider two layers need to be placed on the CPU.

- A larger interval is always better than smaller one.
- Maximize the interval between two layers.



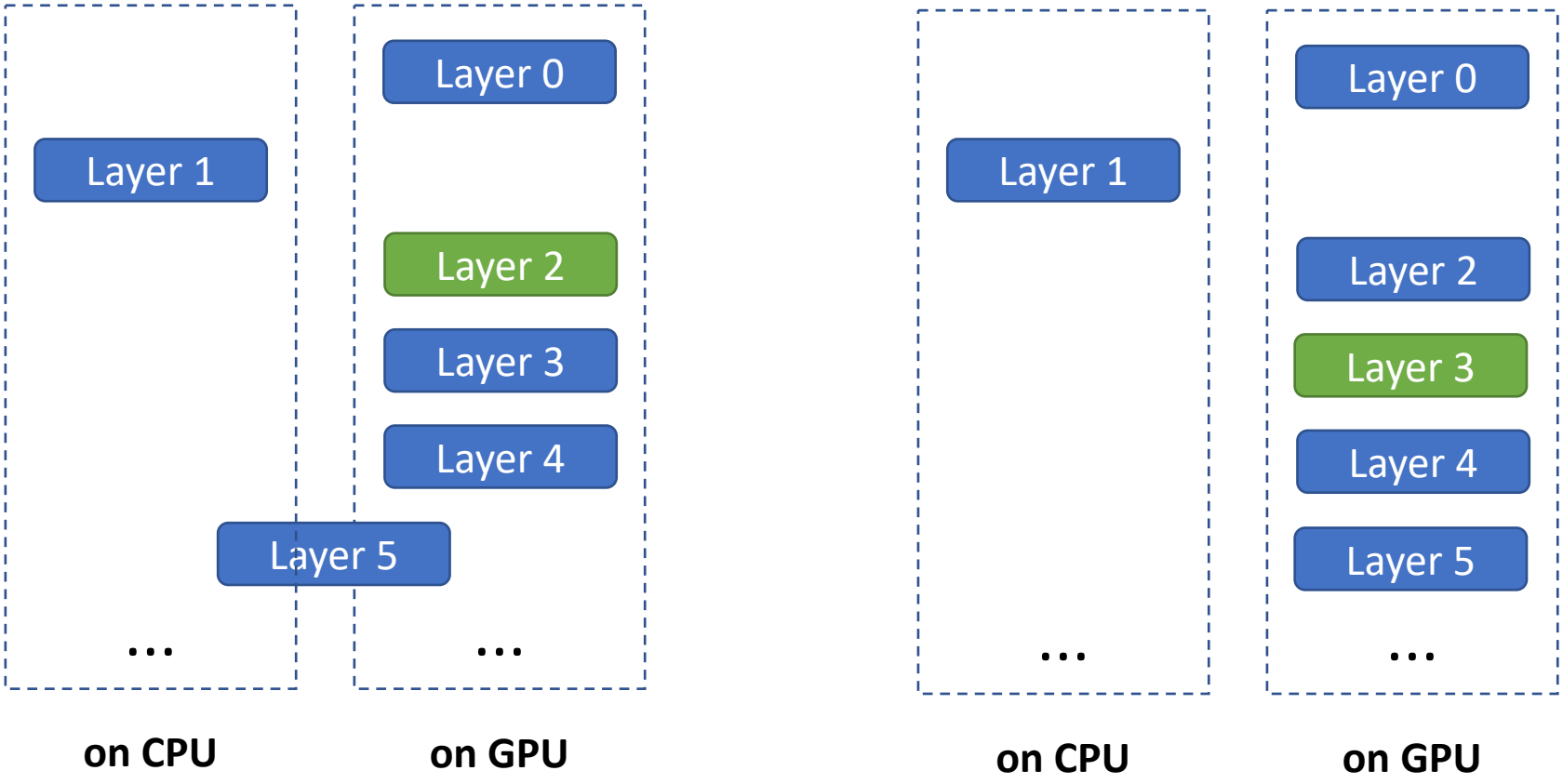
Memory Scheduling

Load layers 1 and 3 at the same time as layer 0 is calculated.



Memory Scheduling

When layer 1 is finished calculating, release it and load layer 5.



Performance

BMInf runs up CPM-2 on GTX 1060.

It also achieves good performance on better GPUs.

Implementation	GPU	Encoder Speed (tokens/s)	Decoder Speed (tokens/s)
BMInf	NVIDIA GeForce GTX 1060	533	1.6
BMInf	NVIDIA GeForce GTX 1080Ti	1200	12
BMInf	NVIDIA GeForce GTX 2080Ti	2275	19
BMInf	NVIDIA Tesla V100	2966	20
BMInf	NVIDIA Tesla A100	4365	26
PyTorch	NVIDIA Tesla V100	-	3
PyTorch	NVIDIA Tesla A100	-	7

Usage

Installation: pip install bminf

Hardware Requirements: GTX 1060 or later

OS: both Windows and Linux

Example:

```
import bminf
model = bminf.models.CPM1()
input = "清华大学是"
result, stoped = model.generate(
    input,
    max_tokens=8,
)
print(input + result)
```

Demo

Thanks for listening

THUNLP