Parameter-Efficient Fine-Tuning for Large Language Models & Training LLM as Scale

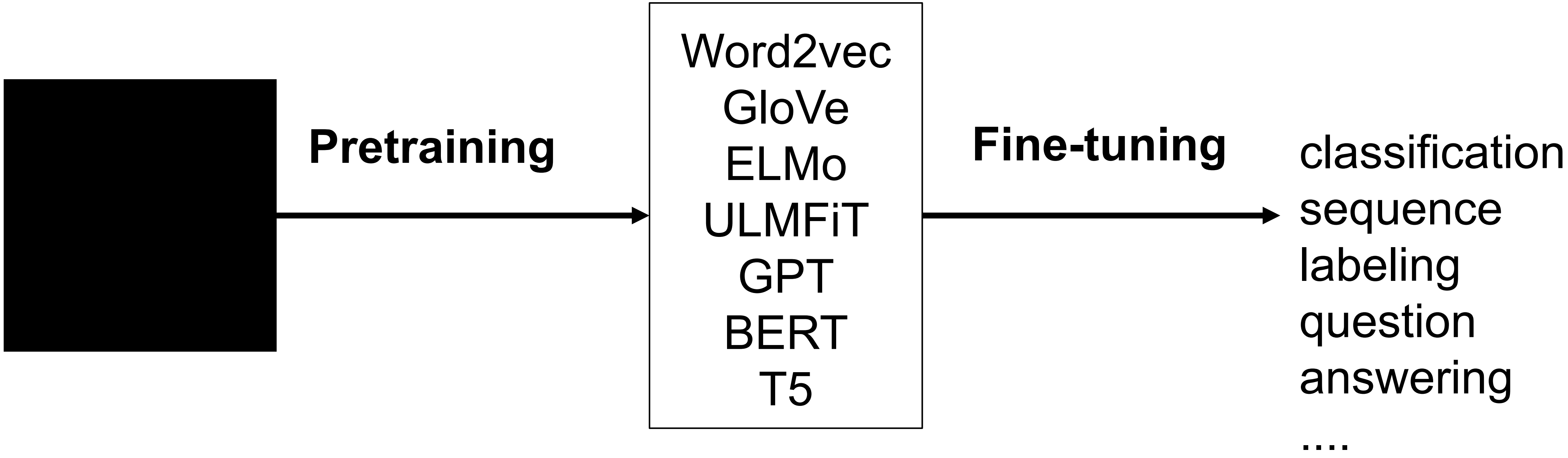
Dinesh Raghu

IBM Research

Gaurav Pandey

IBM Research

Transfer Learning before the Large Language Models Era

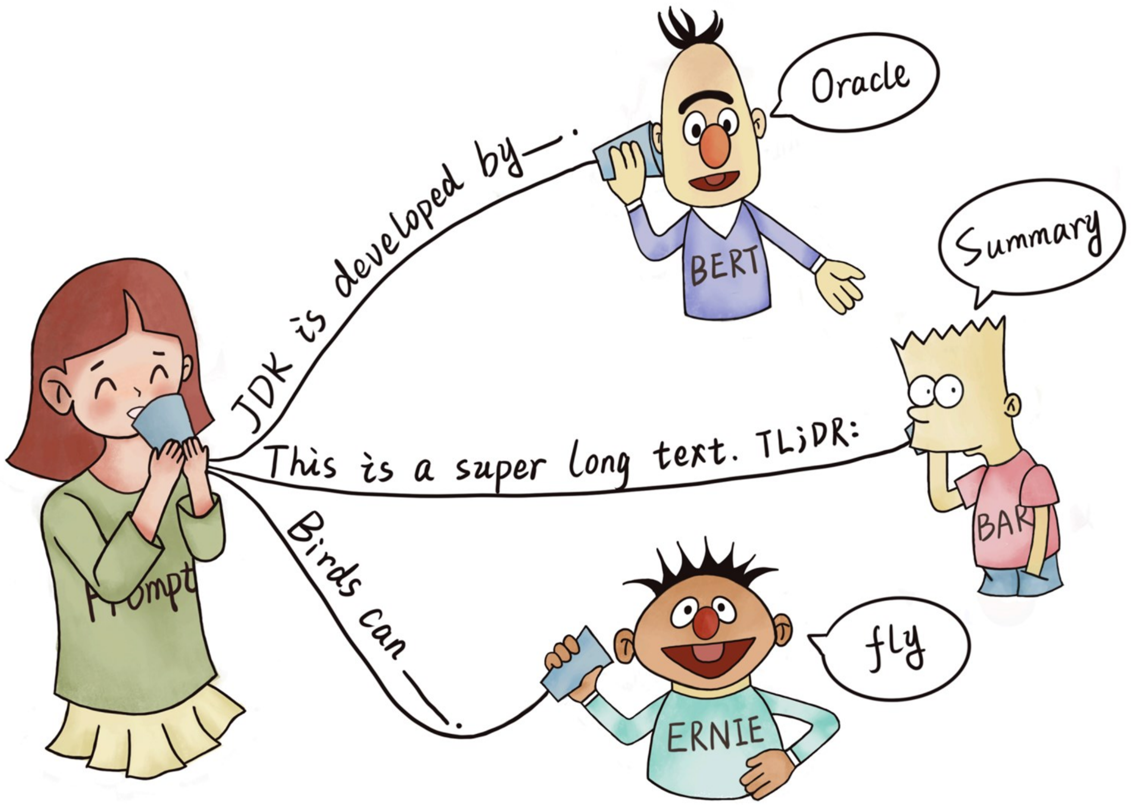


Credit: [NAACL 2019 Transfer learning tutorial]

Transfer Learning in the Large Language Models Era

Pretraining

BERT
BART
ERNIE
GPT-3
PaLM



In-context learning has mostly replaced fine-tuning for large models

Credit: Liu et al. (2021)

Downsides of In-context Learning

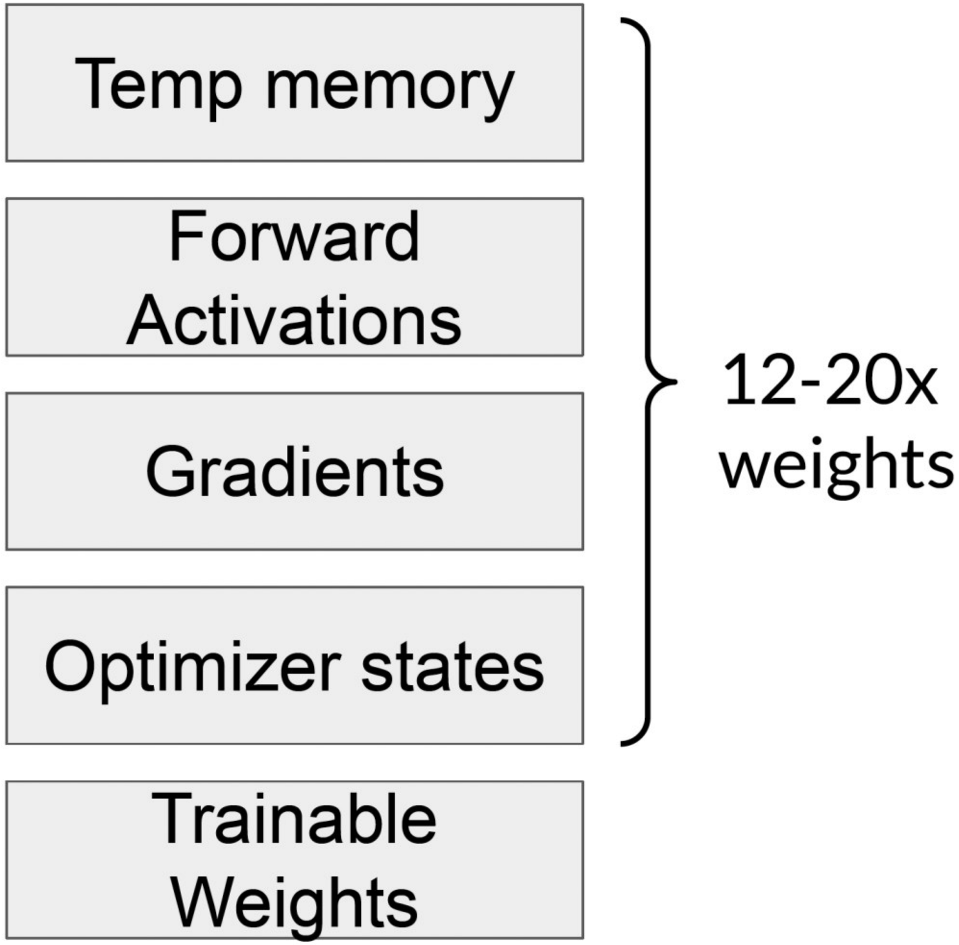
Downsides of In-context Learning

1. **Poor performance:** Prompting generally performs worse than fine-tuning [Brown et al., 2020].
2. **Sensitivity** to the wording of the prompt [Webson & Pavlick, 2022], order of examples [Zhao et al., 2021; Lu et al., 2022], etc.
3. **Lack of clarity** regarding what the model learns from the prompt. Even random labels work [Min et al., 2022]!
4. **Inefficiency:** The prompt needs to be processed *every time* the model makes a prediction.

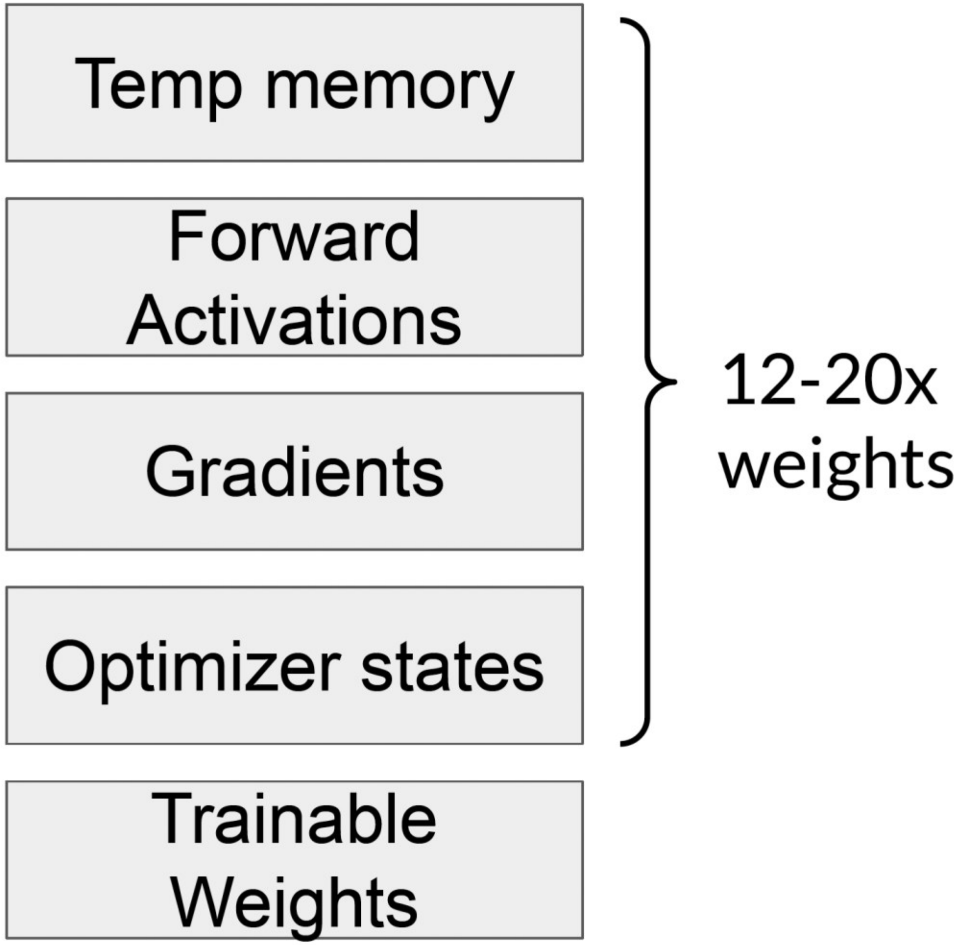
Credit: EMNLP 2022 PEFT Tutorial

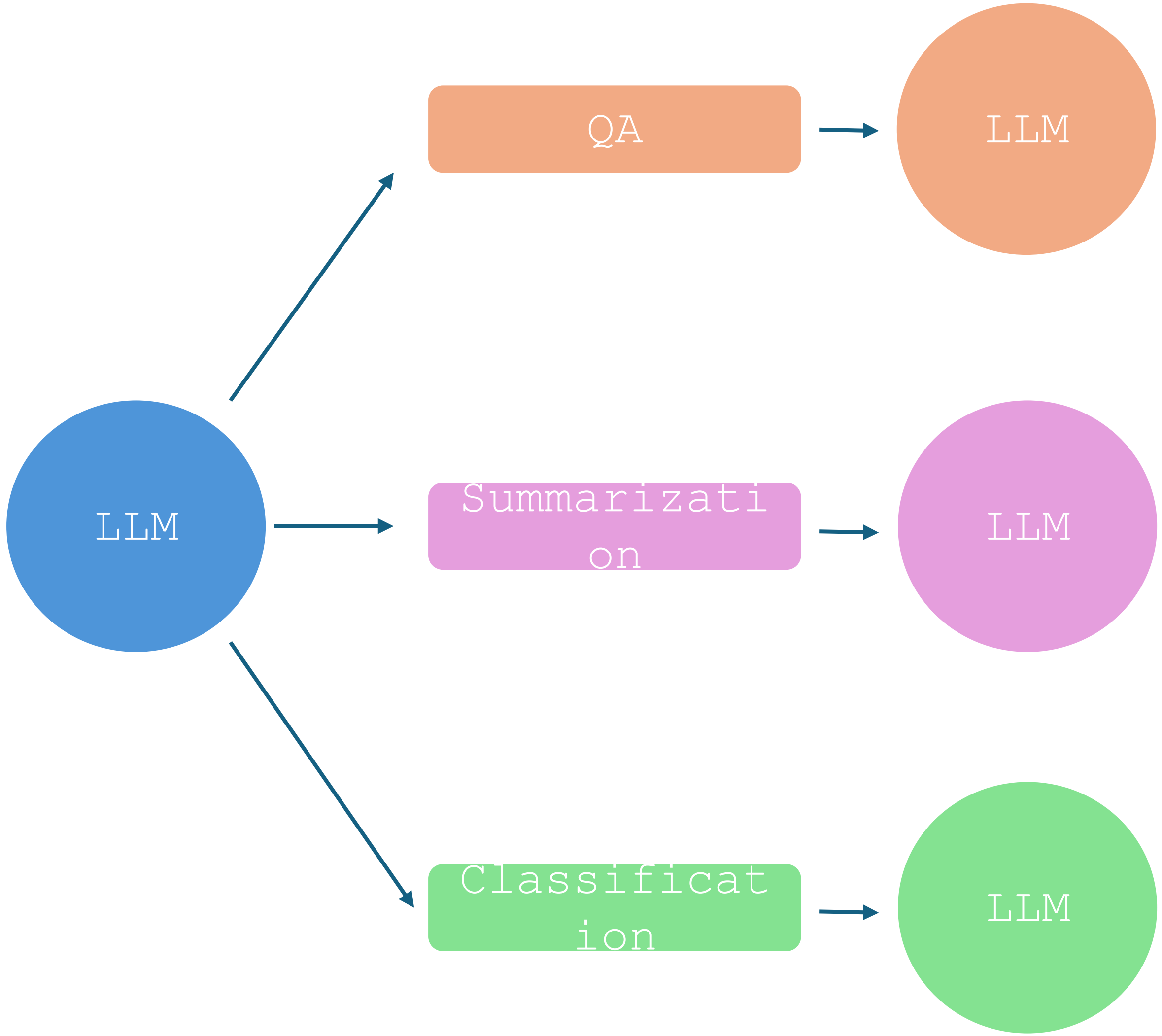
Why is Full Fine Tuning in LLMs challenging?

Why is Full Fine Tuning in LLMs challenging?



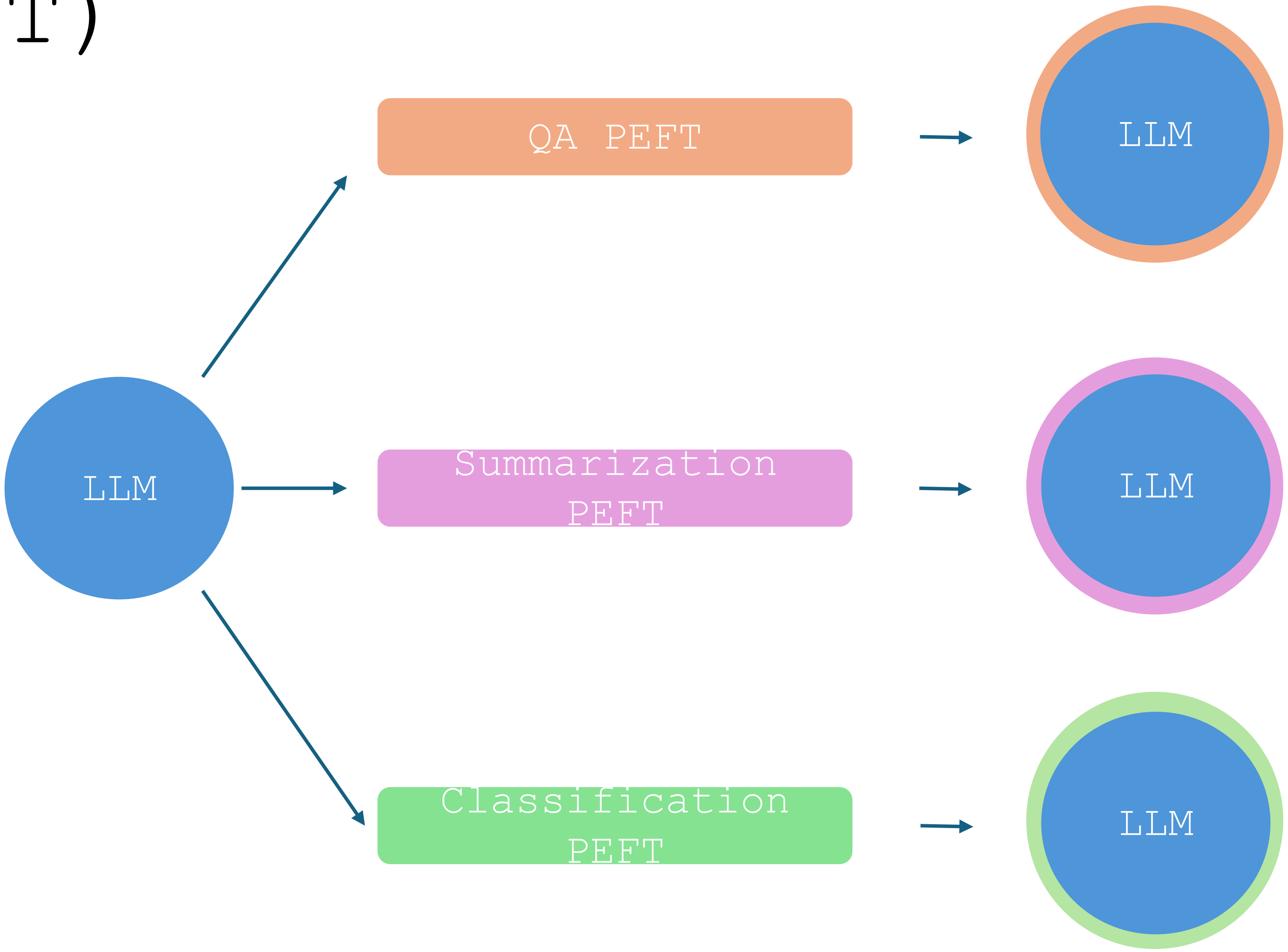
Why is Full Fine Tuning in LLMs challenging?





What can we do then?

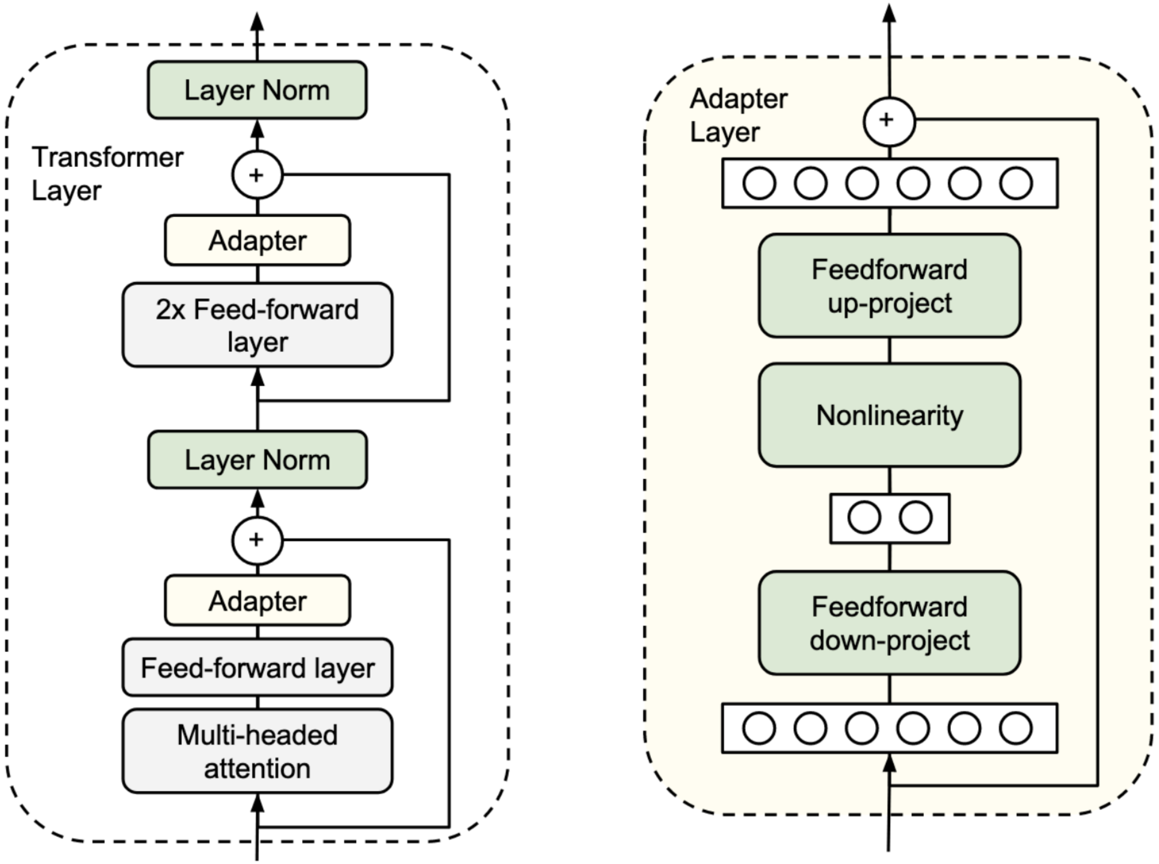
Parameter Efficient Fine Tuning (PEFT)



Outline

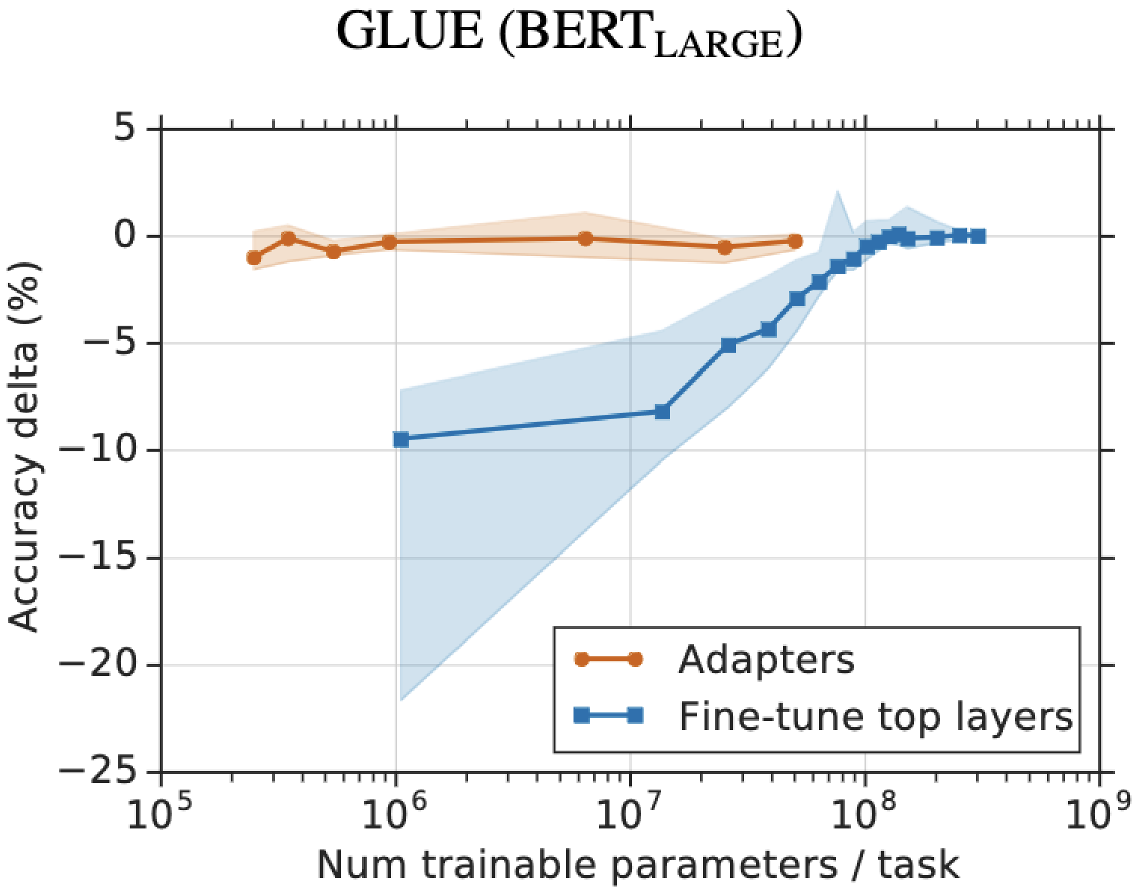
- Adapters
- Prompt Tuning
- Low Rank Adapters

Adapters



Architecture of adapter module and its integration with the transformer [Houlsby et al., 2019]

Adapters



Accuracy versus the number of trained parameters, aggregated across tasks. The lines and shaded areas indicate the 20th, 50th, and 80th percentiles across tasks.
[Houlsby et al., 2021]

Outline

- Adapters
- Prompt Tuning
- Low Rank Adapters

Prompt Tuning

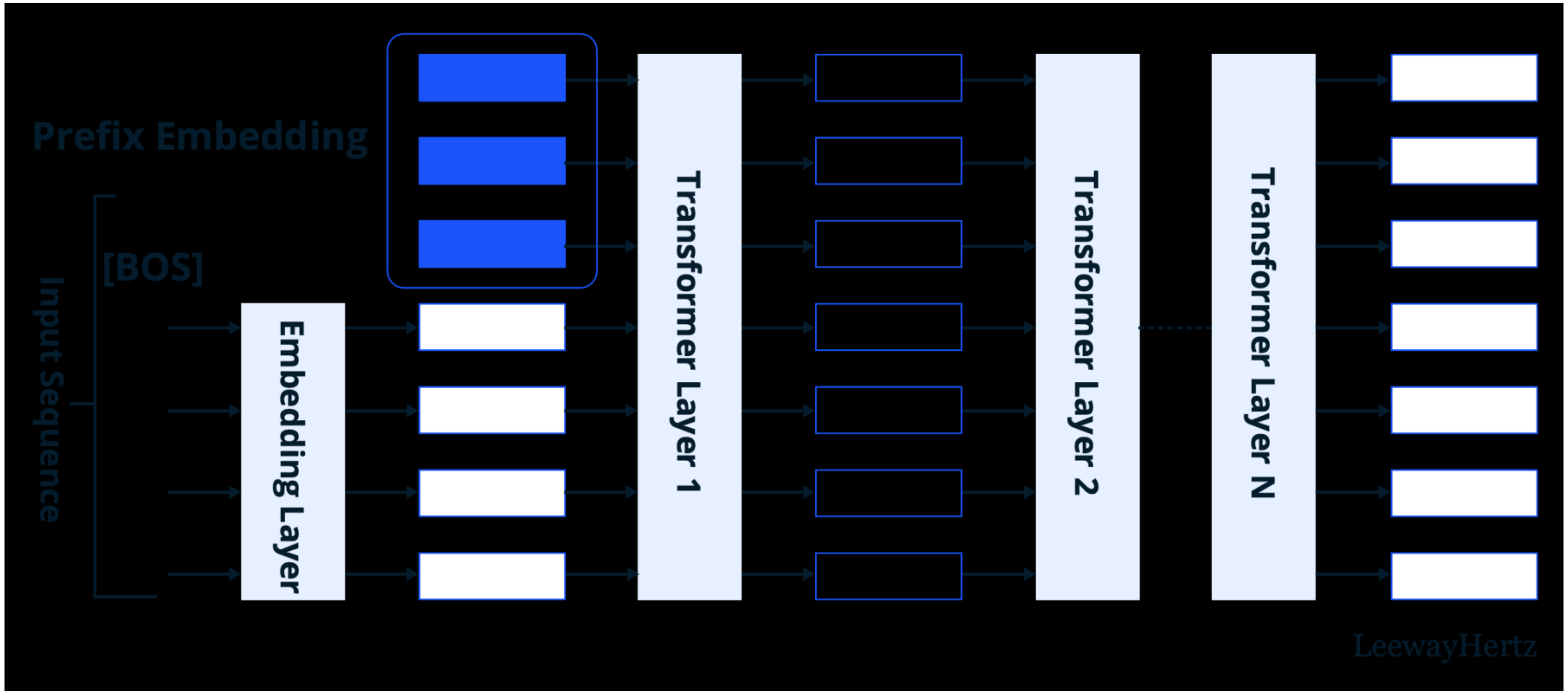
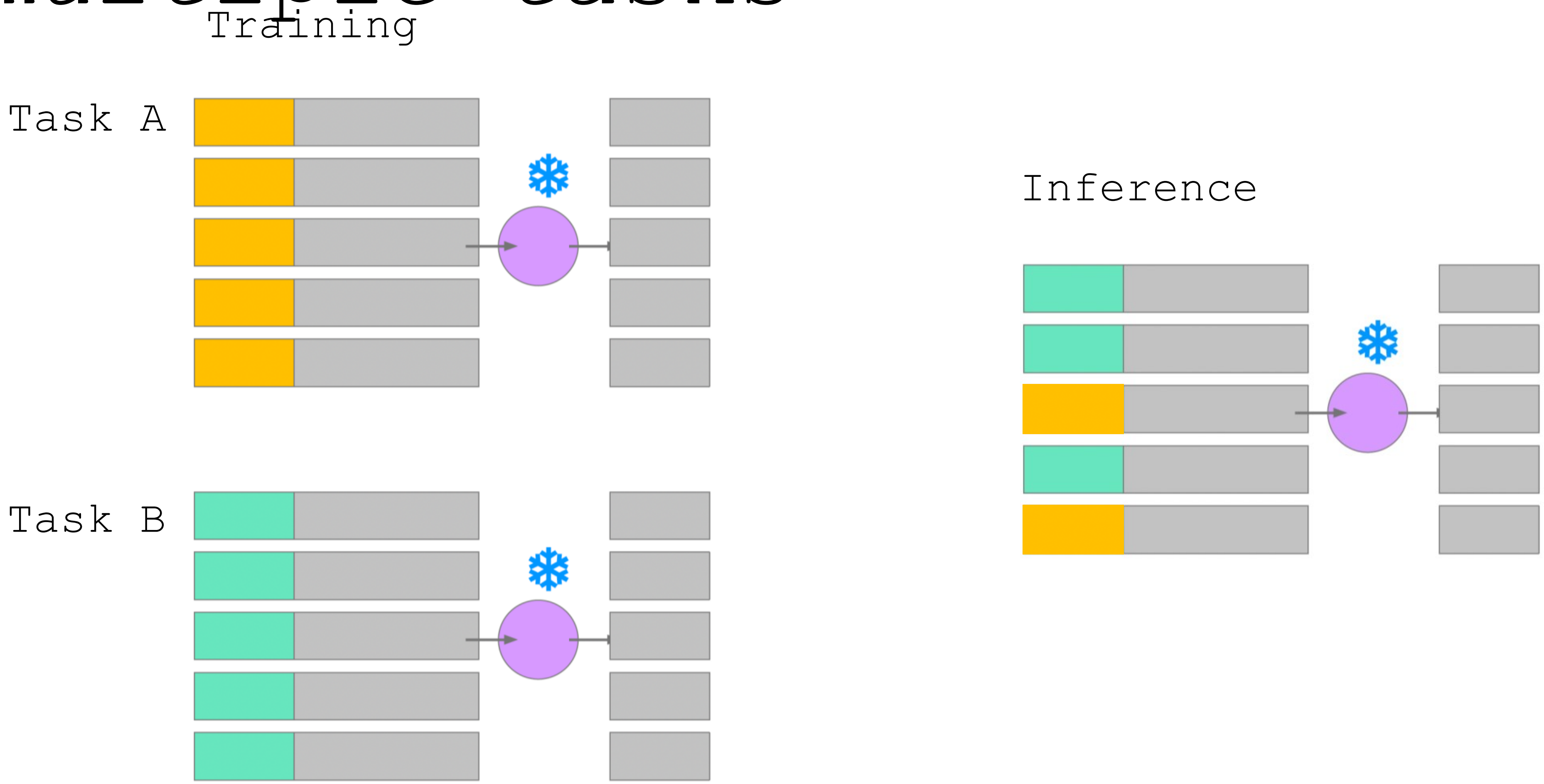


Image Credits: leewayhertz.com

Prompt Tuning: Easy to batch multiple tasks

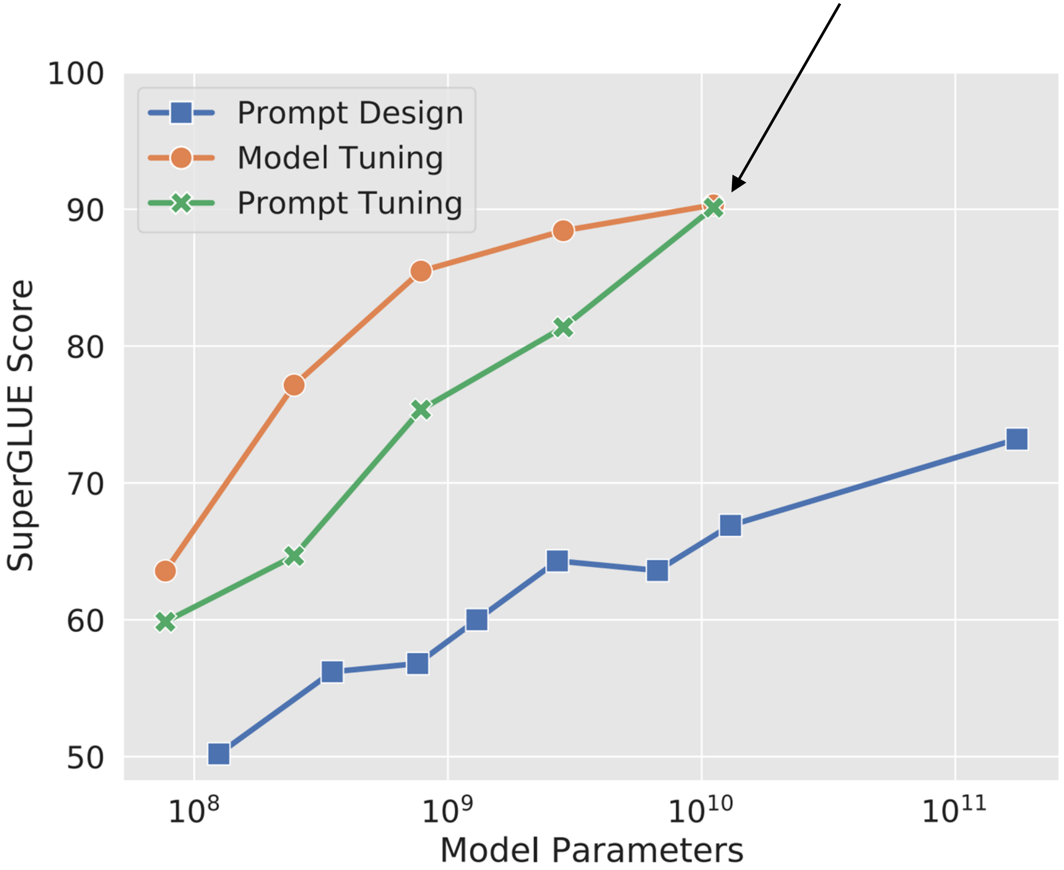


Adapted from DeepLearning.AI

Prompt Tuning

Prompt tuning performs poorly at smaller model sizes and on harder tasks [Mahabadi et al., 2021; Liu et al., 2022]

Prompt tuning only matches fine-tuning at the largest model size



Prompt tuning vs standard fine-tuning and prompt design across T5 models of different sizes [Lester et al., 2021]

Multi-Layer Prompt Tuning

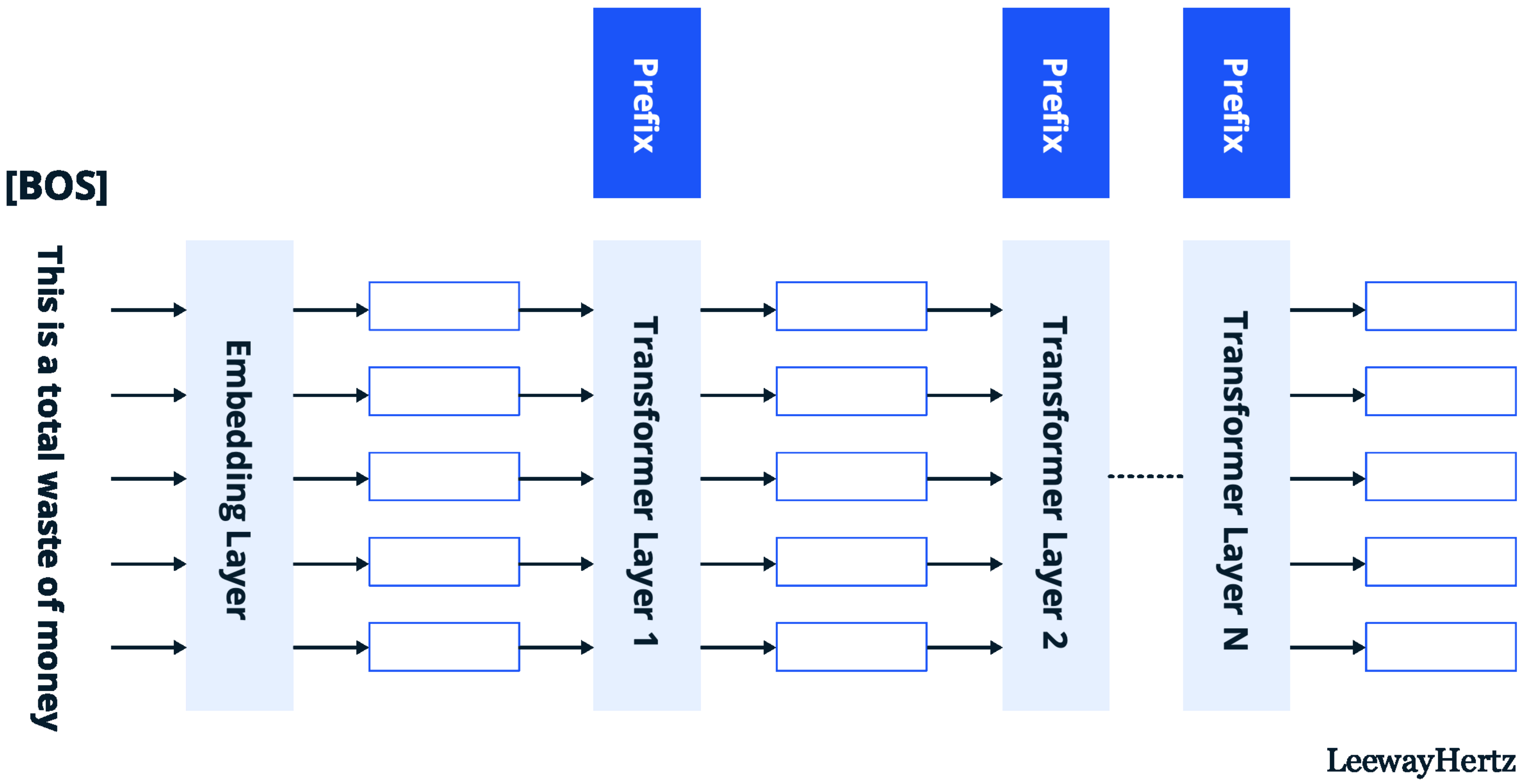


Image Credits: leewayhertz.com

Outline

- Adapters
- Prompt Tuning
- Low Rank Adapters

Low-Rank Composition

- Li et al. [2018] show that models can be optimized in a low-dimensional, randomly oriented subspace rather than the full parameter space

Standard fine-tuning:

$$\theta^{(D)} = \theta_0^{(D)} + \theta_\tau^{(D)}$$

Low-rank fine-tuning:

$$\theta^{(D)} = \theta_0^{(D)} + P\theta^{(d)}$$

A random $D \times d$ projection matrix

Credit: EMNLP 2022 PEFT Tutorial

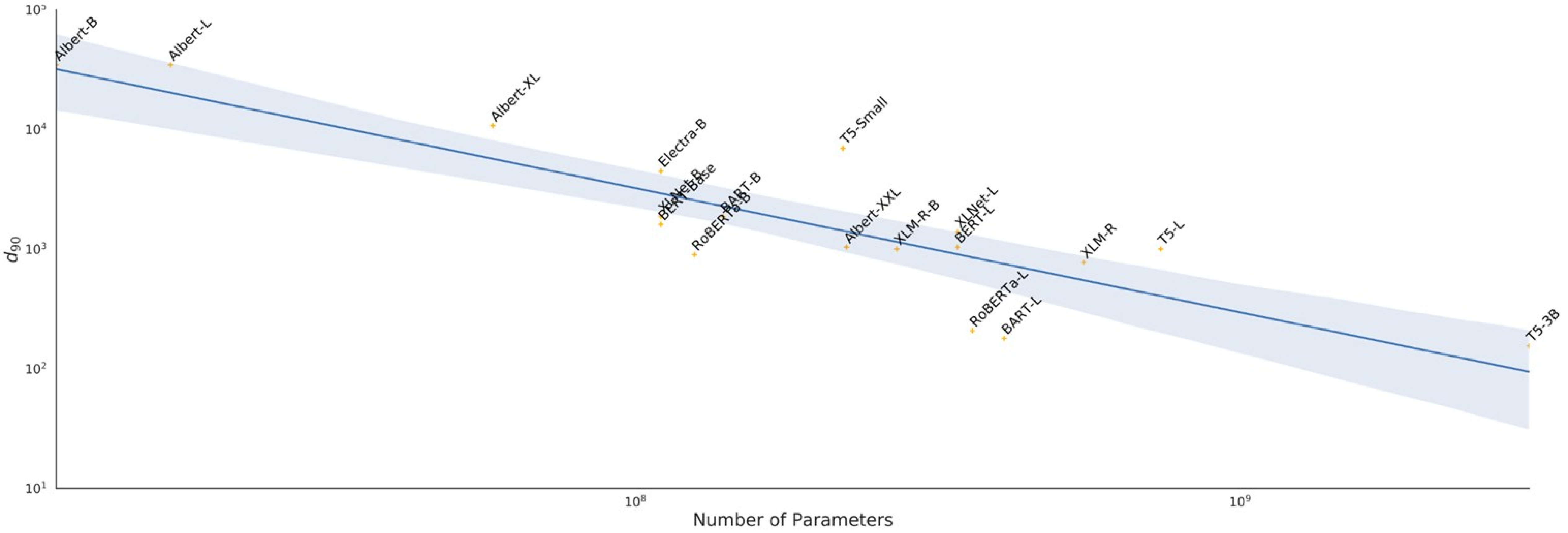
Intrinsic Dimensionality

- Li et al. [2018] refer to the minimum d where a model achieves within 90% of the full parameter model performance, d_{90} as the intrinsic dimensionality of a task
- Aghajanyan et al. [2021] investigate the intrinsic dimensionality of different NLP tasks and pre-trained models

- Observations:
  - Intrinsic dimensionality decreases during pre-training
  - Larger models have lower intrinsic dimensionality

Credit: EMNLP 2022 PEFT Tutorial

Intrinsic Dimensionality



Intrinsic dimension d_{90} on the MRPC dataset for models of different sizes [Aghajanyan et al., 2021]

Structure Aware Low Rank Tuning

- Aghajanyan et al. [2021] also propose a structure-aware version
- Allocate one scalar λ_i per layer to learn layer-wise scaling:

$$\theta_i^{(D)} = \theta_{0,i}^{(D)} + \lambda_i P \theta_i^{(d)}$$

- However, storing the random matrices still requires a lot of extra space and is slow to train [Mahabadi et al., 2021]

Credit: EMNLP 2022 PEFT Tutorial

Low-rank Adaptation (LoRA)

- Instead of learning a low-rank factorization via a random matrix P, we can learn the projection matrix directly (if it is small enough)
- LoRA [Hu et al., 2022] learns two low-rank matrices A and B that are applied to the self-attention weights:

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

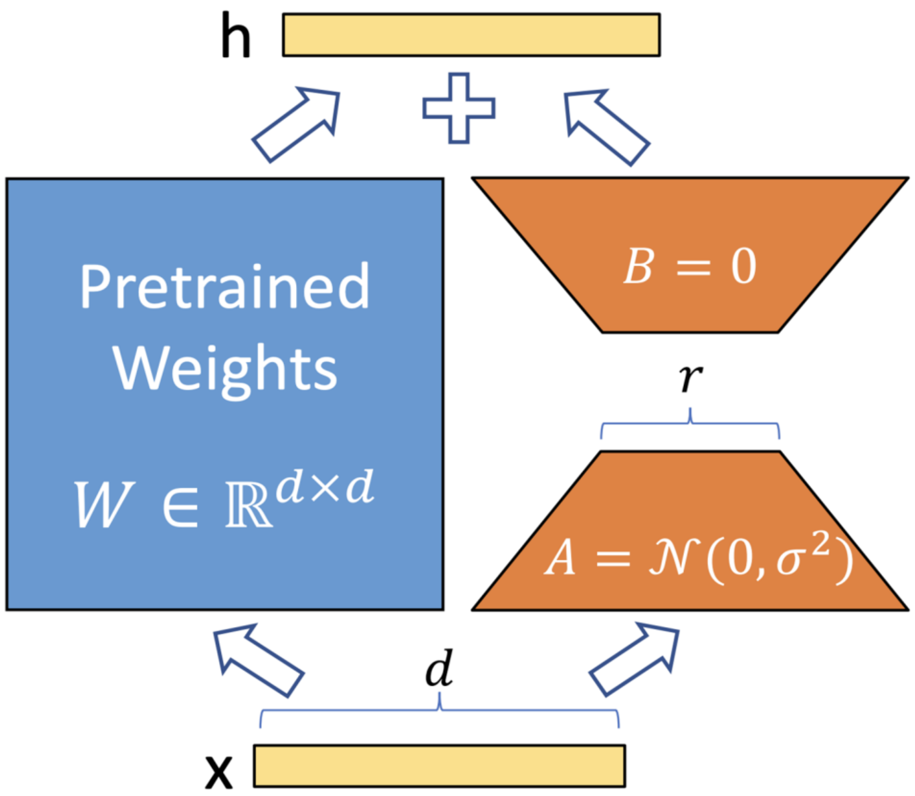


Image Credits: Hu et al., 2022

LoRA: Effect of rank on Performance

	Weight Type	$r=1$	$r=2$	$r=4$	$r=8$	$r=64$
WikiSQL($\pm 0.5\%$)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI ($\pm 0.1\%$)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

Validation accuracy on WikiSQL and MultiNLI with different rank [Hu et al., 2021]

Other Extensions of LoRA

- LongLoRA [Chen et al., 2024]
  - Sparse Local attention to support longer context length during finetuning
- LoRA+ [Hayou et al., 2024]
  - different learning rates for the LoRA adapter matrices A and B improves finetuning speed
- DyLoRA [Valipou et al., 2023]
  - selects rank without requiring multiple runs of training

Quantized LoRA

- Finetune a 65B model on a single 48GB GPU

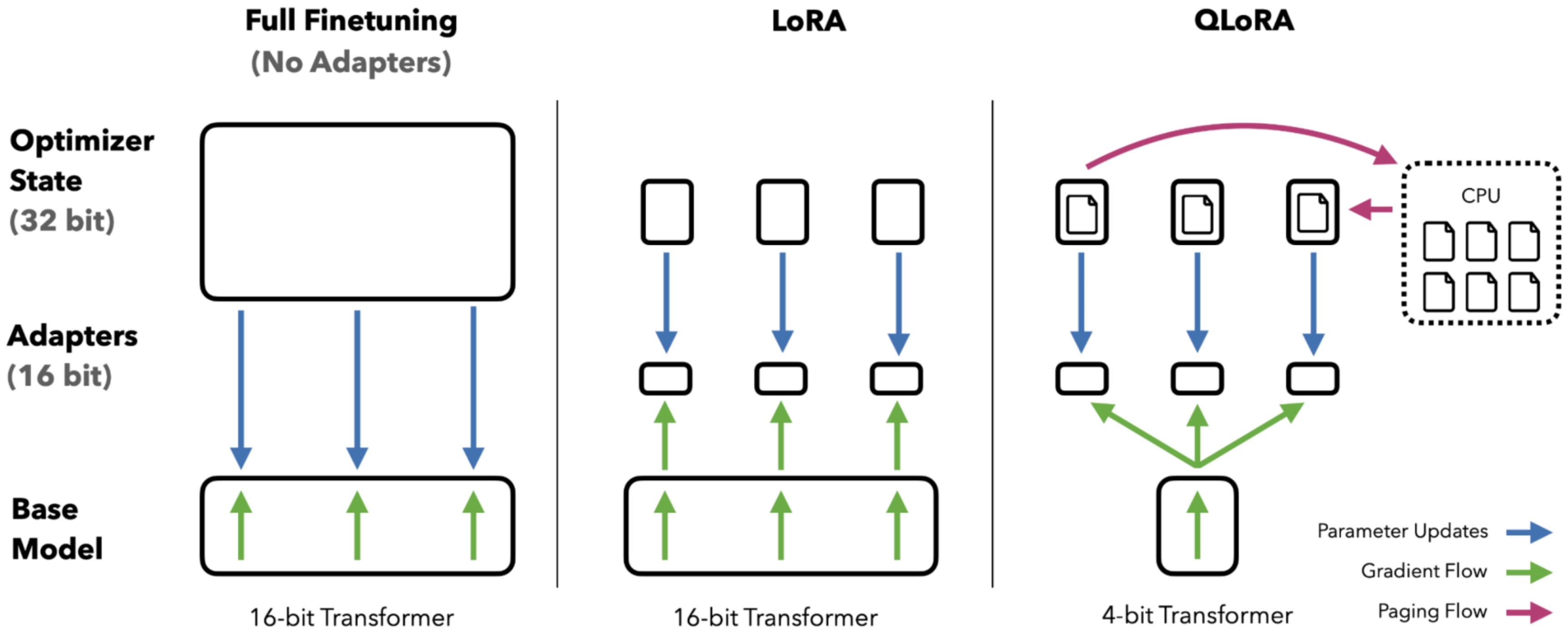


Image Credits: [Dettmers et al., 2023]

Summary

- Adapters
- Prompt Tuning
- Low Rank Adapters

Training LLMs at Scale

Table of Contents

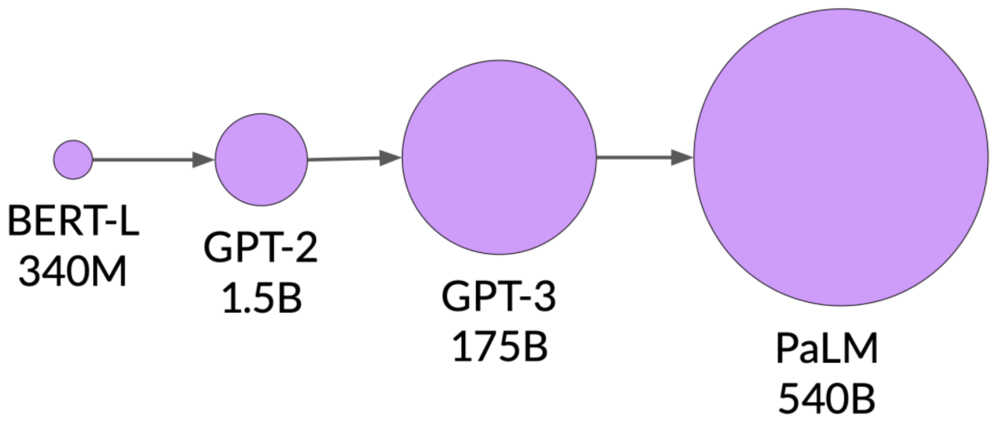
- Model sizes and GPU memory consumption
- Methods for training at scale
  - Quantization of parameters
  - Data Parallel
    - Distributed Data Parallel
    - FSDP/ Deepspeed ZeRO
  - Model Parallel – Not covered

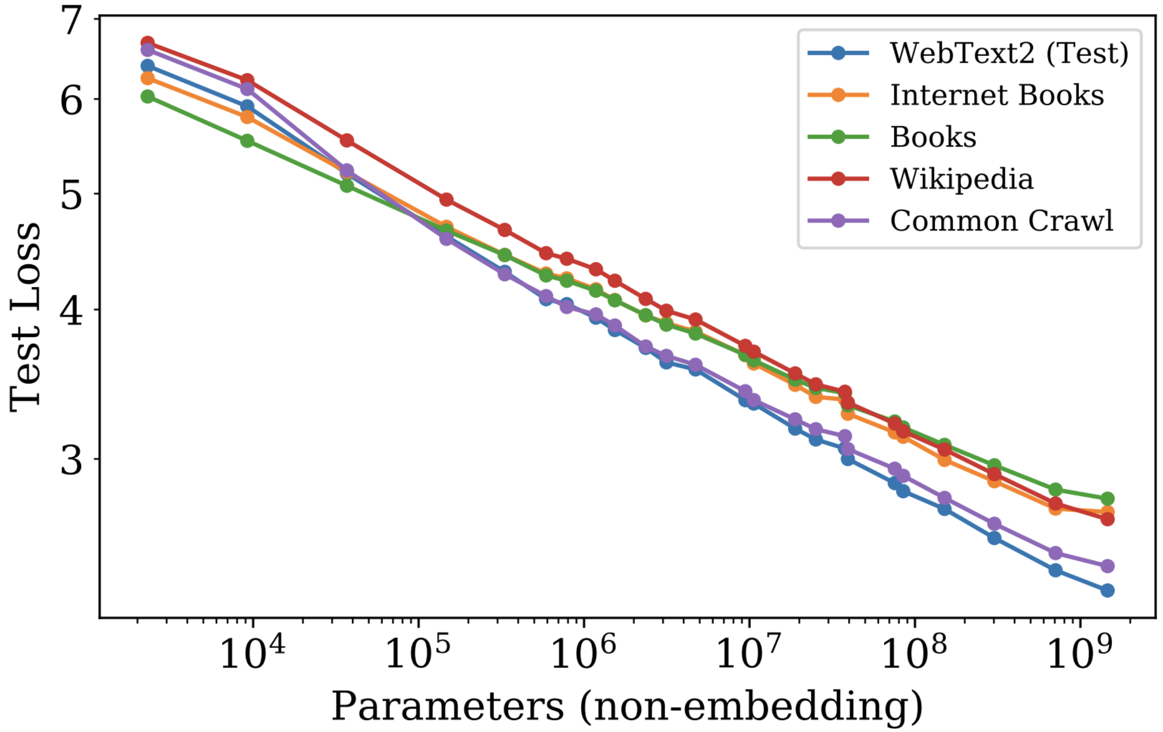
Adapted from a blog by MSR and slides from deeplearning.ai

Table of Contents

- **Model sizes and GPU memory consumption**
- Methods for training at scale
  - Quantization of parameters
  - Data Parallel
    - Distributed Data Parallel
    - FSDP/ Deepspeed ZeRO
  - Model Parallel – Not covered

Model sizes





How do we train such large models?

Adapted from a blog by MSR and slides from deeplearning.ai

Training a 1B parameter model

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter

Adapted from a blog by MSR and slides from deeplearning.ai

Training a 1B parameter model in FP32

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter

1B parameters $= 4 * 1\text{B} \approx 4GB$

Training a 1B parameter model

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter

$$\text{1B parameters} = 4 * 1\text{B} \approx 4GB$$

4 GB

parameters

Adapted from a blog by MSR and slides from deeplearning.ai

Training a 1B parameter model

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter

$$\text{1B parameters} = 4 * 1\text{B} \approx 4GB$$

4 GB

parameters

≈ 8 GB

activations

Gradient of parameters at layer (t) is a function of

- the Gradient at layer (t+1)
- the activations at layer (t)

Can vary significantly with sequence length

Training a 1B parameter model

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter

$$\text{1B parameters} = 4 * 1\text{B} \approx 4GB$$

4 GB	≈ 8 GB	4 GB
parameters	activations	gradients

Adapted from a blog by MSR and slides from deeplearning.ai

Training a 1B parameter model

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter

$$\texttt{1B parameters} = 4 * 1\text{B} \approx 4GB$$

4 GB	≈ 8 GB	4 GB	4 GB	4 GB
parameters	activations	gradients	Gradient Running Average	Squared Gradient Running Average

Optimizer State

Adapted from a blog by MSR and slides from deeplearning.ai

Training a 1B parameter model

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter

$$\texttt{1B parameters} = 4 * 1\text{B} \approx 4GB$$

$\boxed{\text{4 GB (Gradient Running Average)} \quad \text{4 GB (Squared Gradient Running Average)}}$

4 GB	4 GB	4 GB	4 GB	≈ 8 GB
parameters	gradients	Gradient Running Average	Squared Gradient Running Average	Activations

24 GB

Adapted from a blog by MSR and slides from deeplearning.ai

Table of Contents

- Model sizes and GPU memory consumption
- **Methods for training at scale**
  - **Quantization of parameters**
  - Data Parallel
    - Distributed Data Parallel
    - FSDP/ Deepspeed ZeRO
  - Model Parallel – Not covered

Adapted from a blog by MSR and slides from deeplearning.ai

FP32 vs BF16

FP32 **4 bytes memory**

$$\underbrace{0}_{\substack{\textbf{Sign} \\ \text{1 bit}}} \quad \underbrace{10000000}_{\substack{\textbf{Exponent} \\ \text{8 bits}}} \quad \underbrace{10010010000111111011000}_{\substack{\textbf{Fraction} \\ \text{23 bits}}}$$

BFLOAT16 | BF16 **2 bytes memory**

$$\underbrace{0}_{\substack{\textbf{Sign} \\ \text{1 bit}}} \quad \underbrace{10000000}_{\substack{\textbf{Exponent} \\ \text{8 bits}}} \quad \underbrace{1001001}_{\substack{\textbf{Fraction} \\ \text{7 bits}}}$$

$$V = (-1)^{sign} \times 2^{exponent\,-127} \times 1.mantissa$$

- Also, known as truncated fp32
- Same range as FP32
    - Allows numbers as large as 2^{128} and as small as 2^{-127}
- Reduced precision
    - x and $x + x * 2^{-7}$ are treated as distinct

Quantization

	Bits	Exponent	Fraction	Memory needed to store one value
FP32	32	8	23	4 bytes
FP16	16	5	10	2 bytes
BFLOAT16	16	8	7	2 bytes
INT8	8	-/-	7	1 byte

FLAN T5

For loading a 1B parameter model in GPU memory

4 GB	>	2 GB	>	2 GB	>	1 GB
FP32		FP16		BF16		INT8

Mixed Precision Training in BF16

2 GB
BF16

parameters

2 GB
BF16

gradients

≈ 2 GB

Activations

4 GB

Gradient Running Average

4 GB

Squared Gradient Running Average

4 GB
FP32

Master Copy
Of parameters

$\approx 18GB$

Grows with
- The length of the sequence
- Number of parameters

Adapted from a blog by MSR and slides from deeplearning.ai

Table of Contents

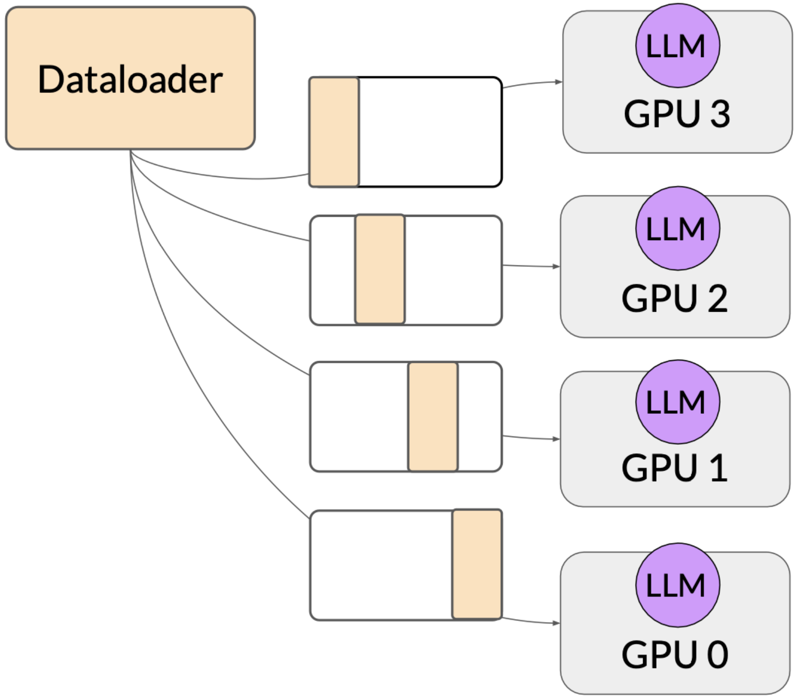
- Model sizes and GPU memory consumption
- **Methods for training at scale**
  - Quantization of parameters
  - **Data Parallel**
    - **Distributed Data Parallel**
    - FSDP/ Deepspeed ZeRO
  - Model Parallel – Not covered

Multi-GPU training

- Data Parallelism
  - Split the dataset across the GPUs/nodes
  - Distributed data parallel
    - Minimizes communication among GPUs
    - Aggregates gradients across GPUs at the end of each training step
    - Each GPU holds the entire model.
  - Deepspeed Zero/FSDP
    - Reduces memory footprint of data parallel
    - Each GPU holds only a portion of the model
    - More communication overhead

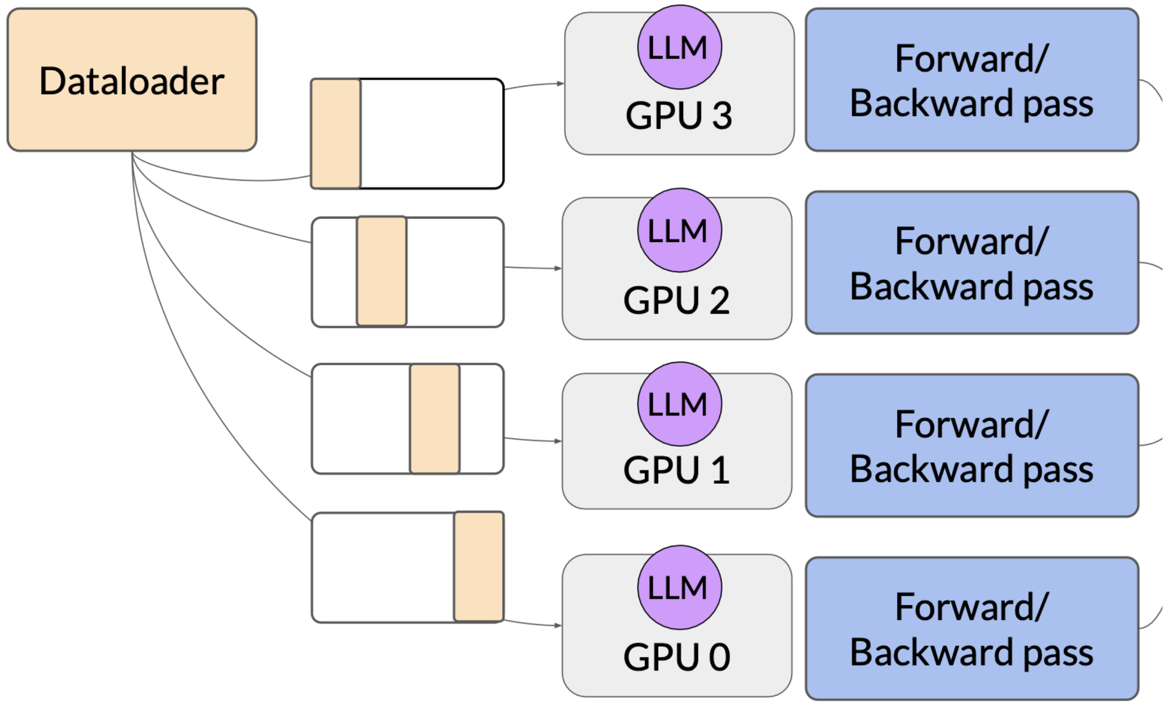
Adapted from a blog by MSR and slides from deeplearning.ai

Distributed Data Parallel



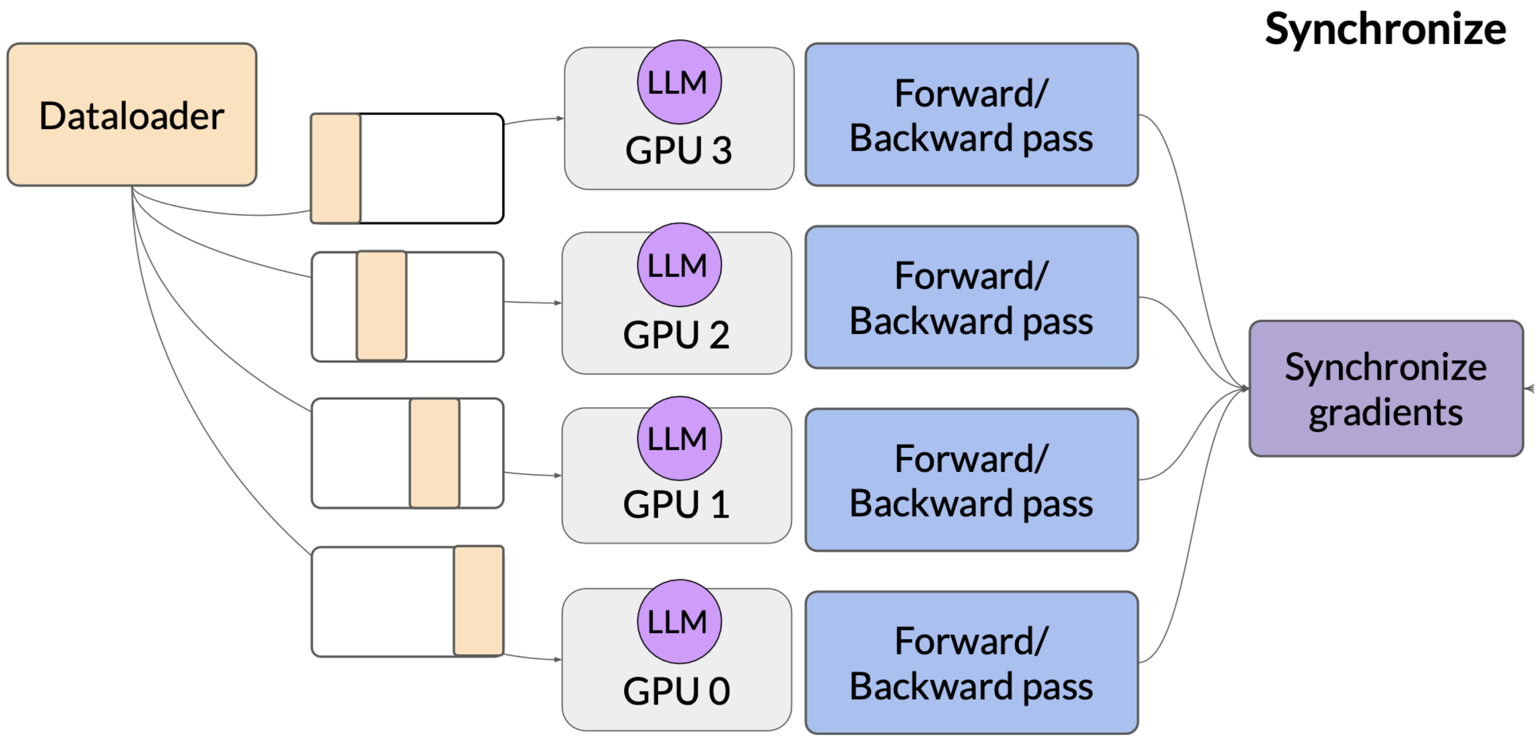
Adapted from a blog by MSR and slides from deeplearning.ai

Distributed Data Parallel



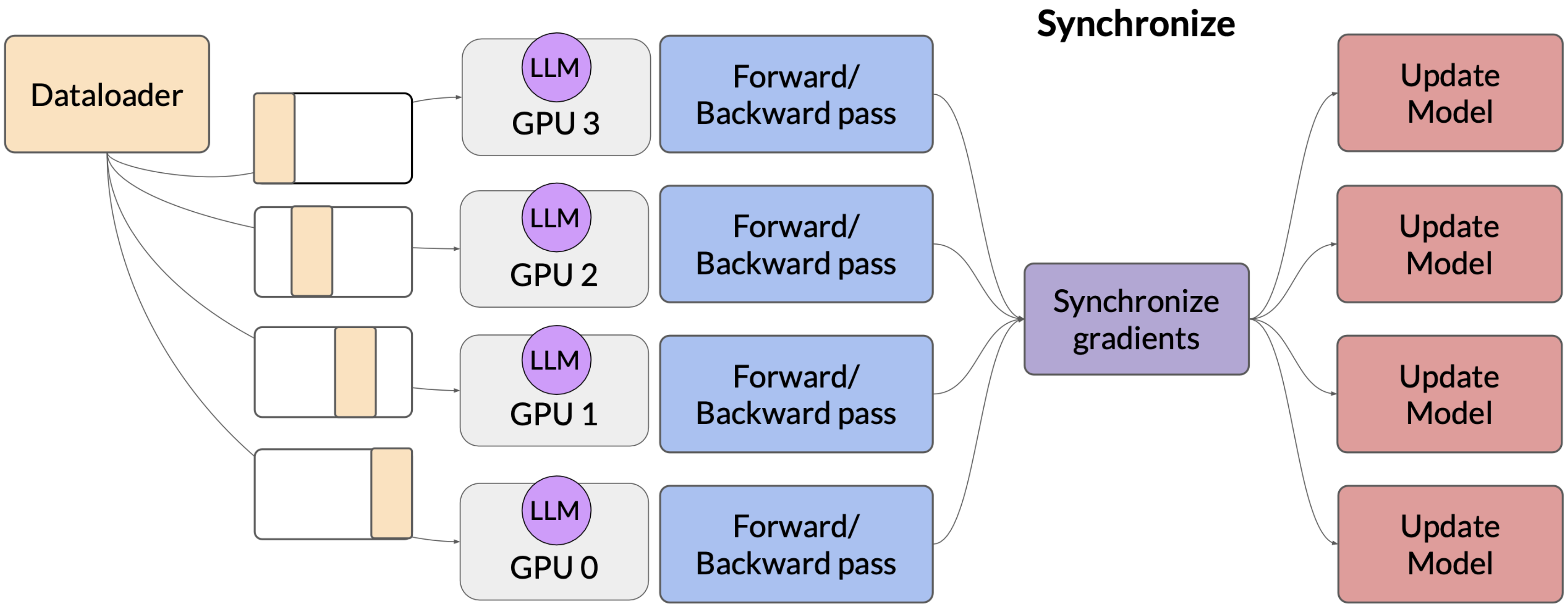
Adapted from a blog by MSR and slides from deeplearning.ai

Distributed Data Parallel



Adapted from a blog by MSR and slides from deeplearning.ai

Distributed Data Parallel



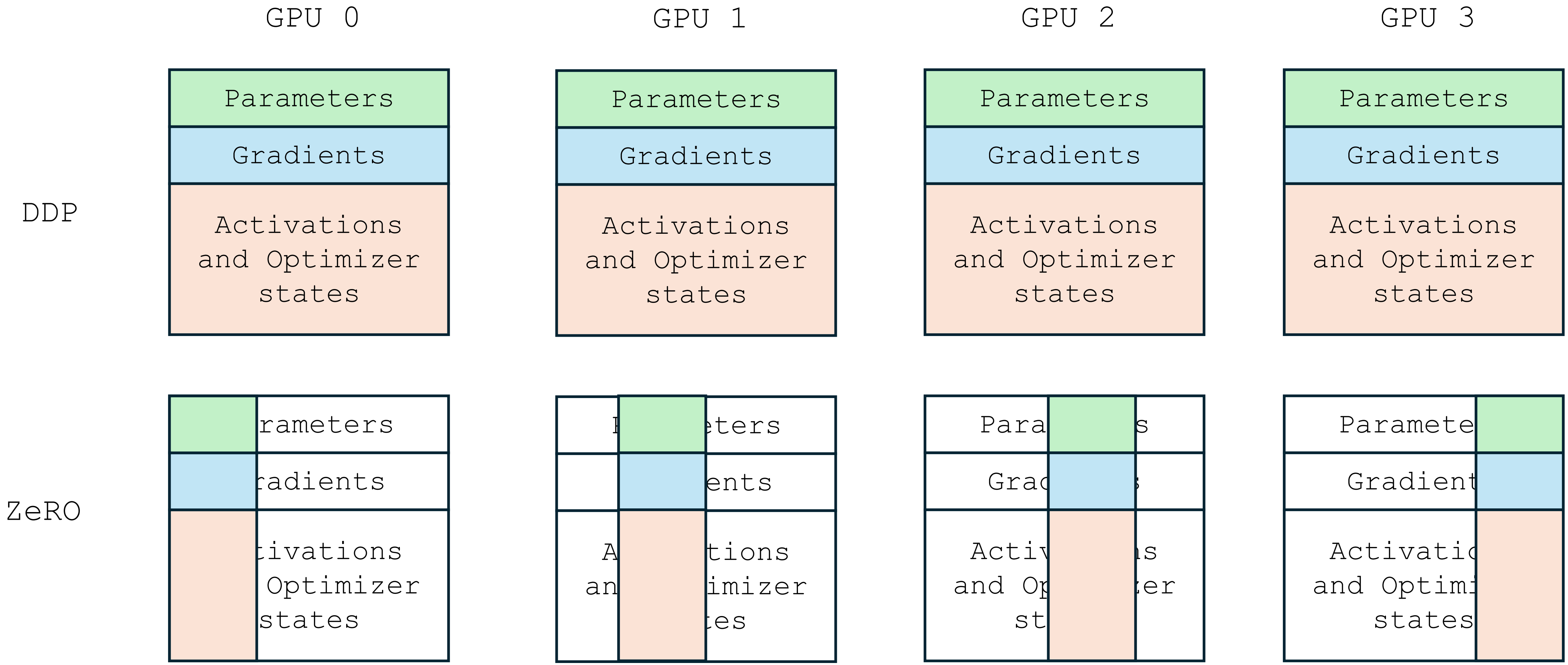
Each GPU contains the model parameters, gradients and activations

Adapted from a blog by MSR and slides from deeplearning.ai

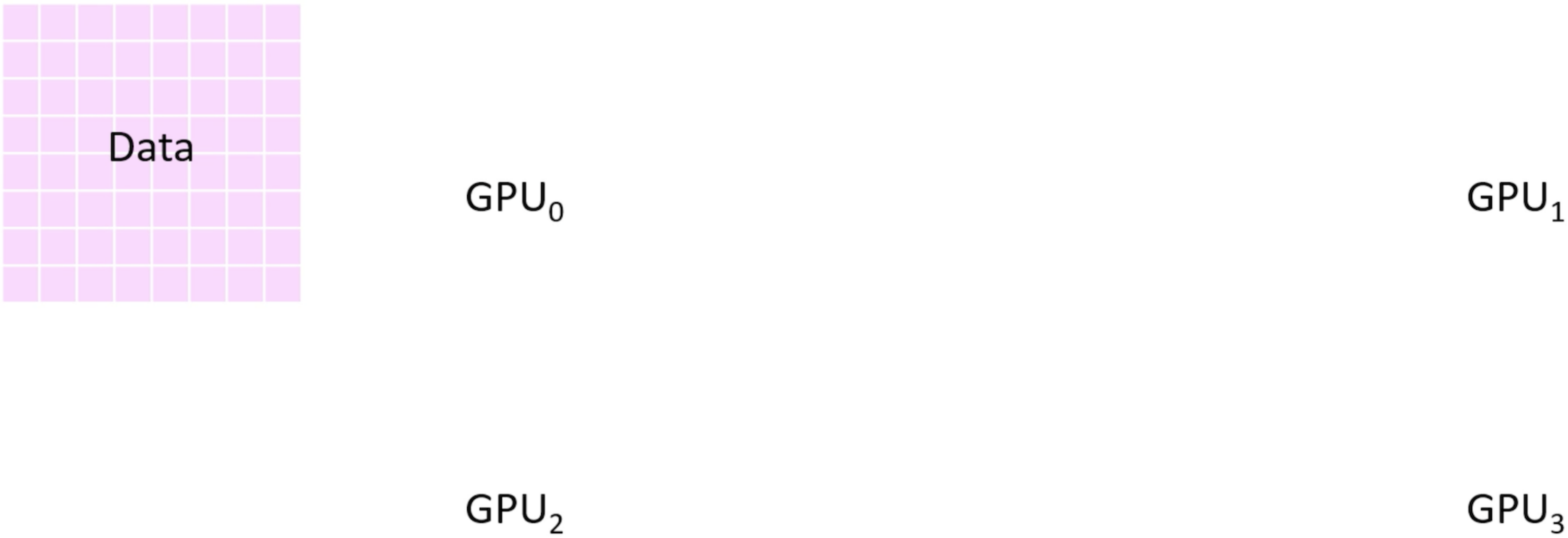
Table of Contents

Adapted from a blog by MSR and slides from deeplearning.ai

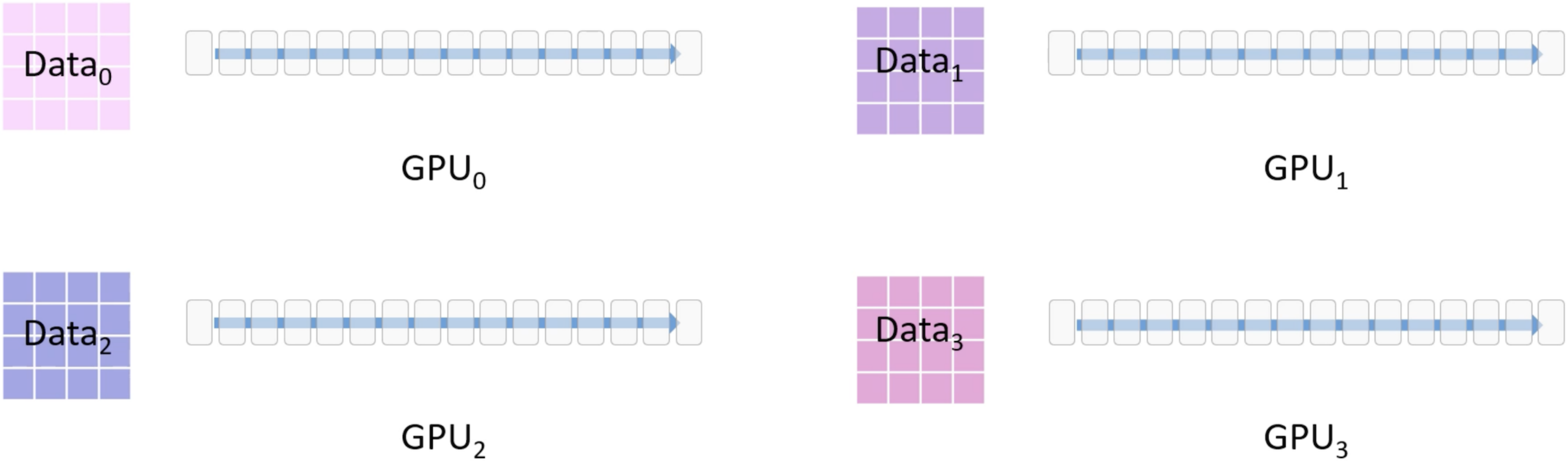
Zero Redundancy Optimizer (ZeRO)



Adapted from a blog by MSR and slides from deeplearning.ai



Adapted from a blog by MSR and slides from deeplearning.ai

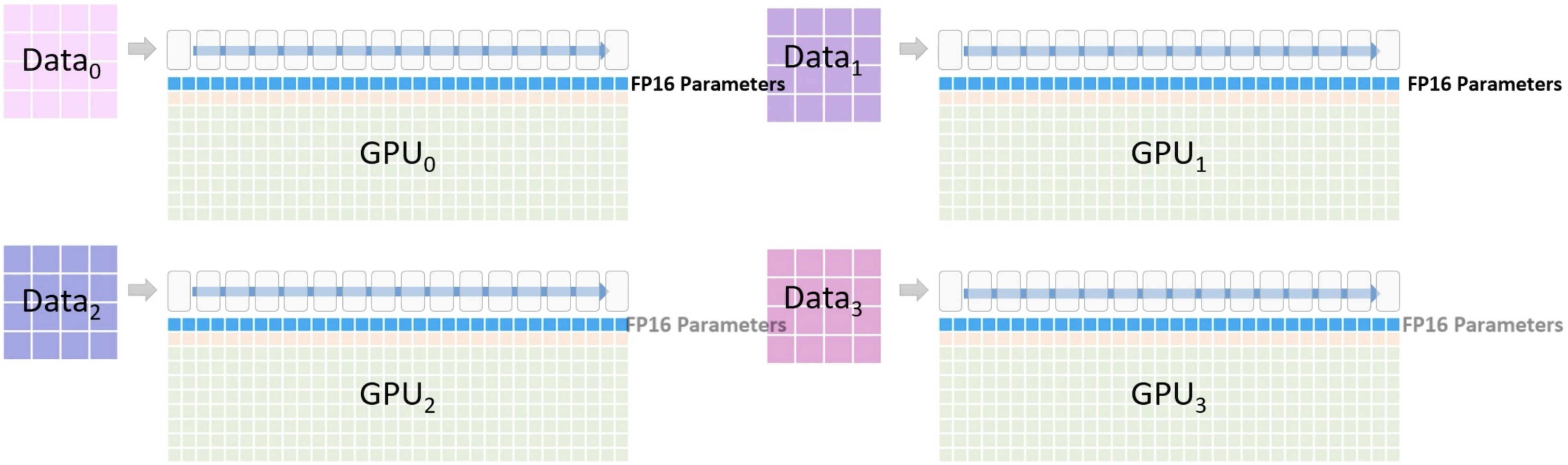


We will use 4-way data parallelism and ZeRO memory optimization
Each GPU will optimize the same model on different data

Adapted from a blog by MSR and slides from deeplearning.ai

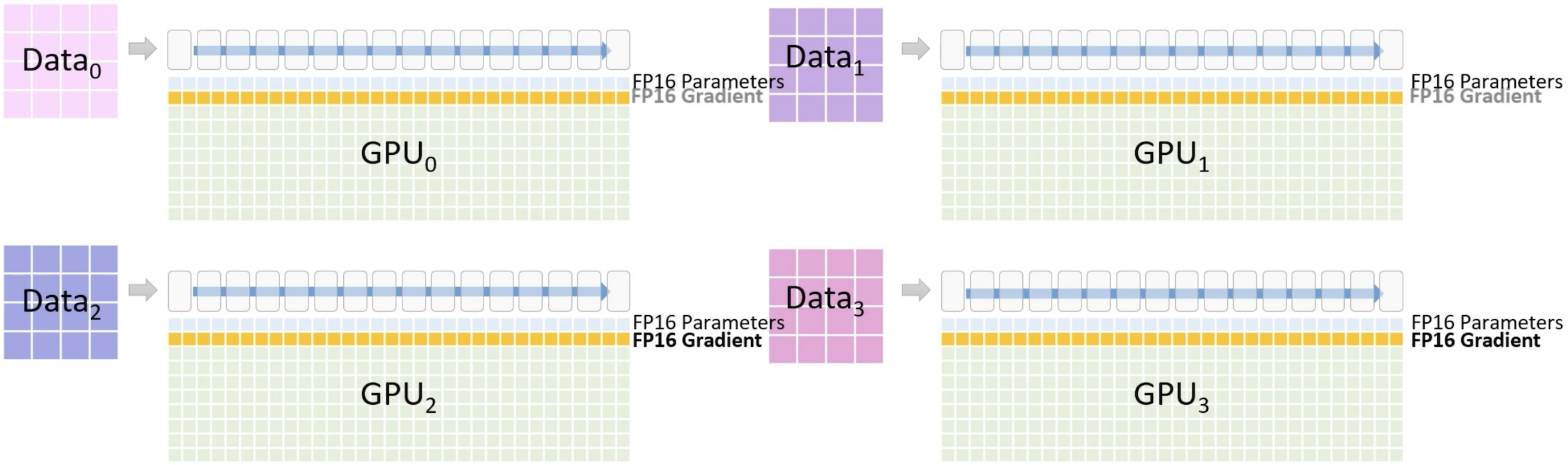
Each cell represents GPU memory used by its corresponding transformer layer

Adapted from a blog by MSR and slides from deeplearning.ai

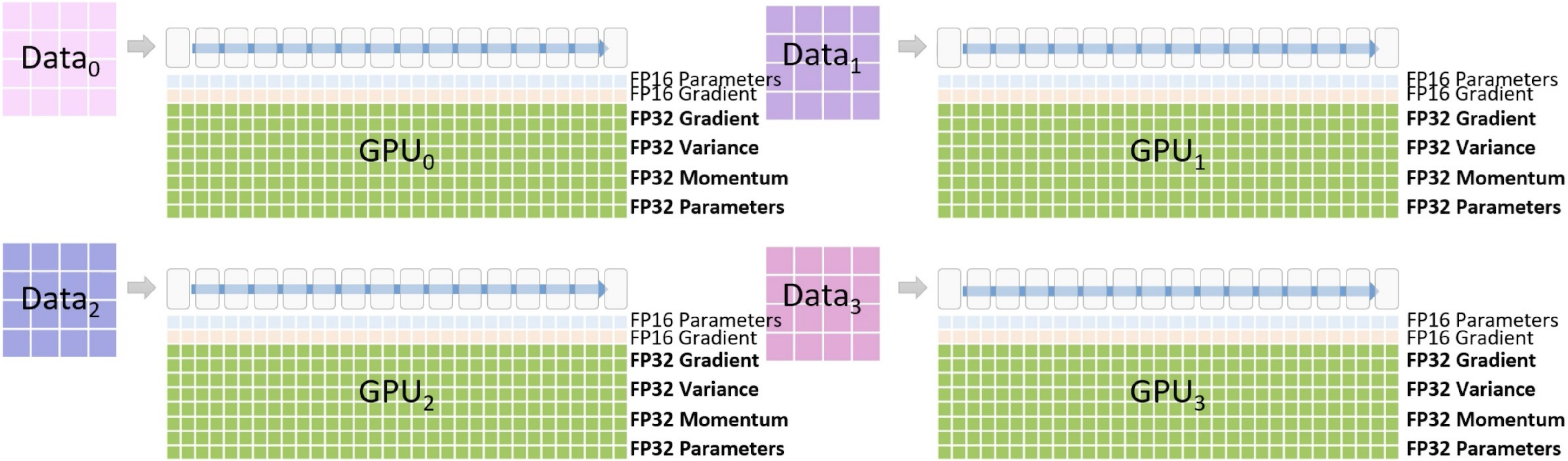


The first row is the fp16 version of the model parameters

Adapted from a blog by MSR and slides from deeplearning.ai

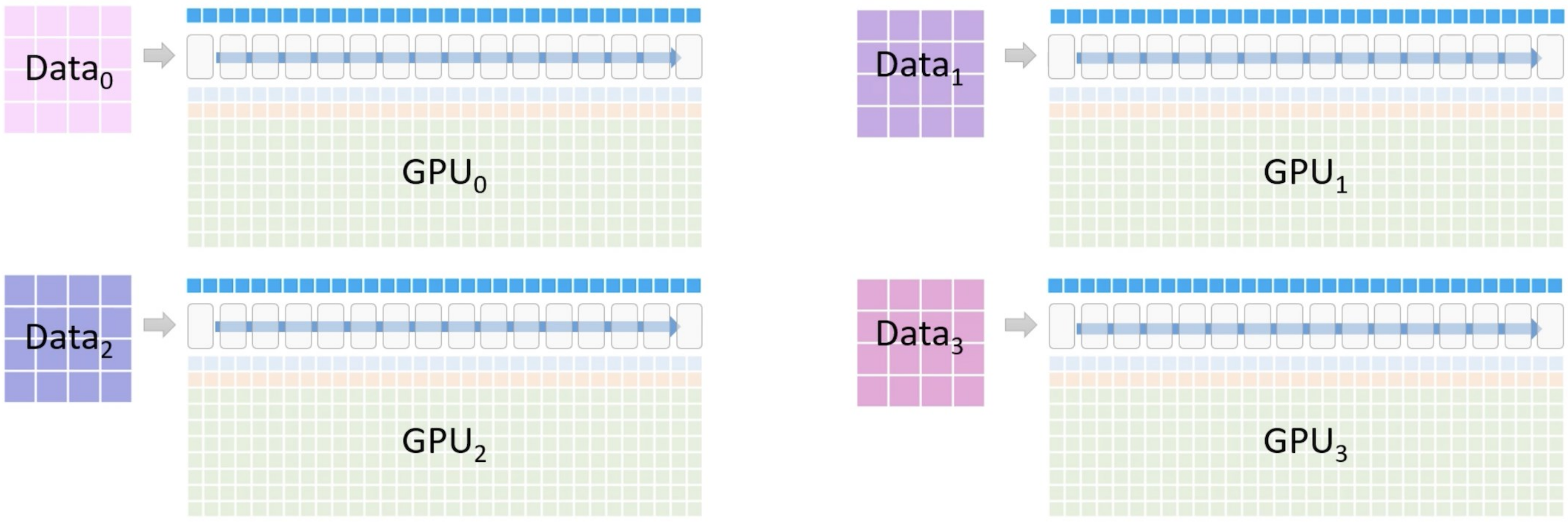


The 2nd row is the fp16 version of the gradient
This will be used in the backwards pass to update the weights



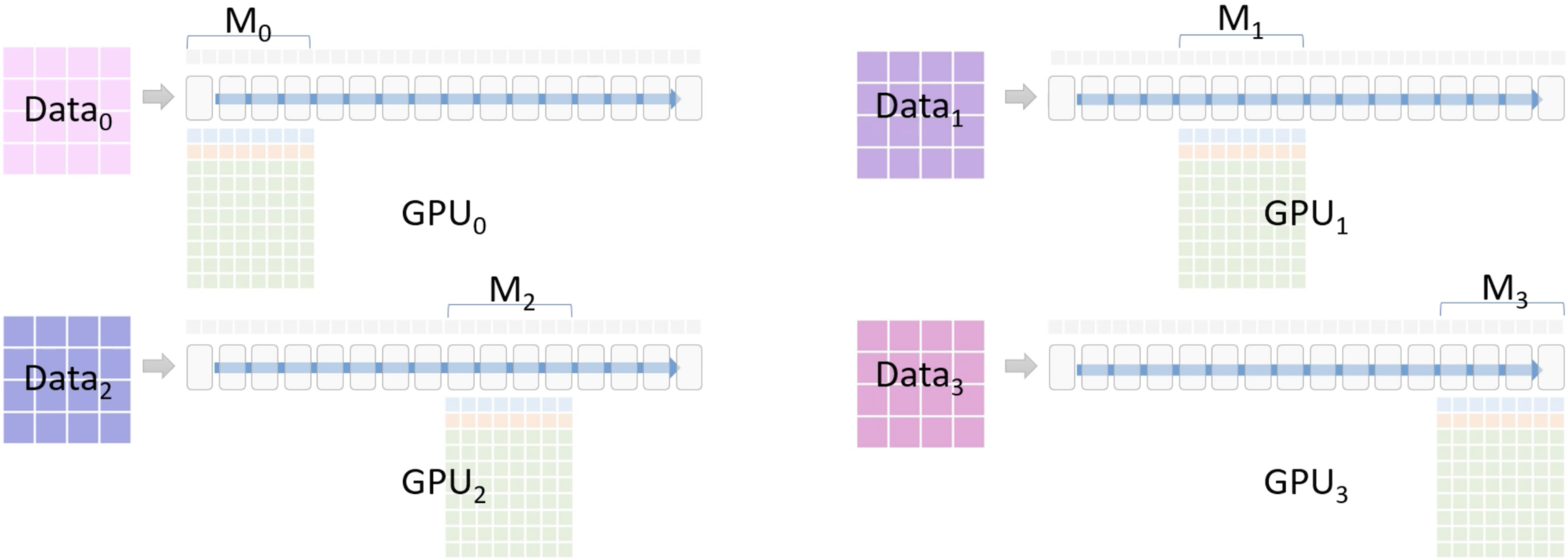
The last (massive) block of memory is used by the Optimizer.
This is not used until after the fp16 gradients are computed

Adapted from a blog by MSR and slides from deeplearning.ai



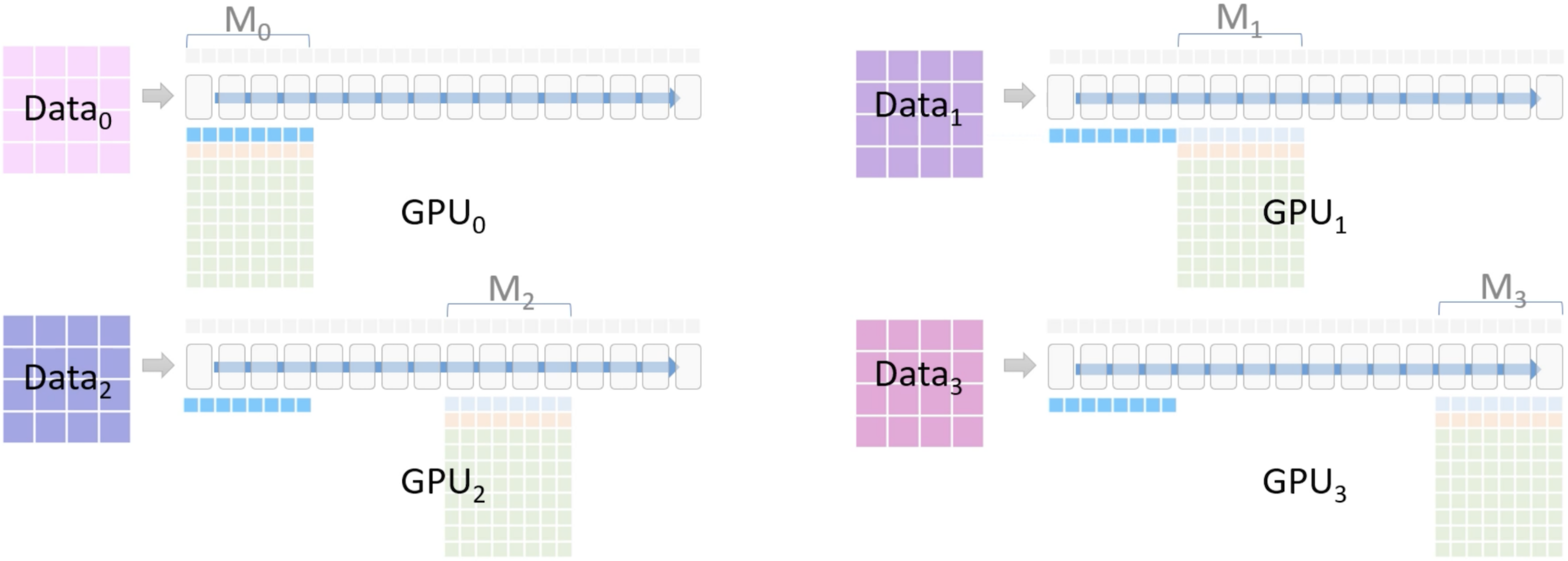
We also need a buffer to keep all the activations for each transformer layer. (e.g. Attention heads, MLPs, etc)

Adapted from a blog by MSR and slides from deeplearning.ai



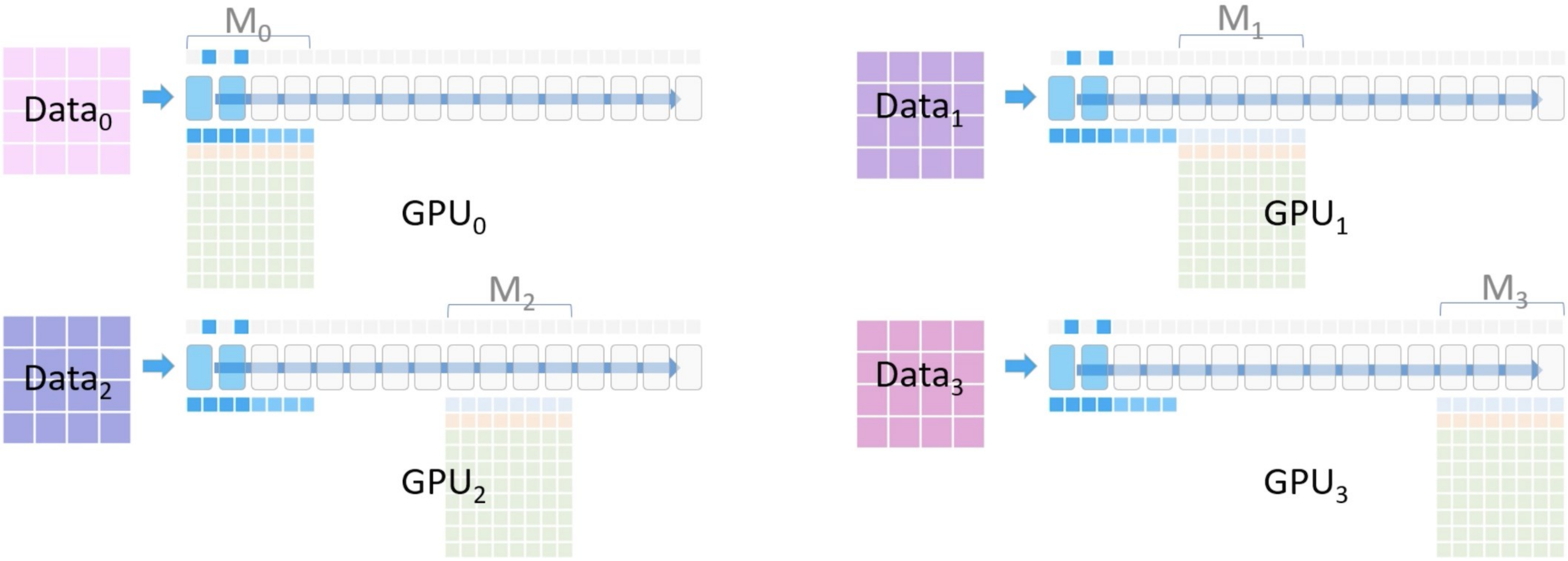
Each GPU is responsible for 1 piece of the end model

Adapted from a blog by MSR and slides from deeplearning.ai



Only GPU_0 initially has the model parameters for M_0.
It broadcasts them to $GPU_{1,2,3}$

Adapted from a blog by MSR and slides from deeplearning.ai

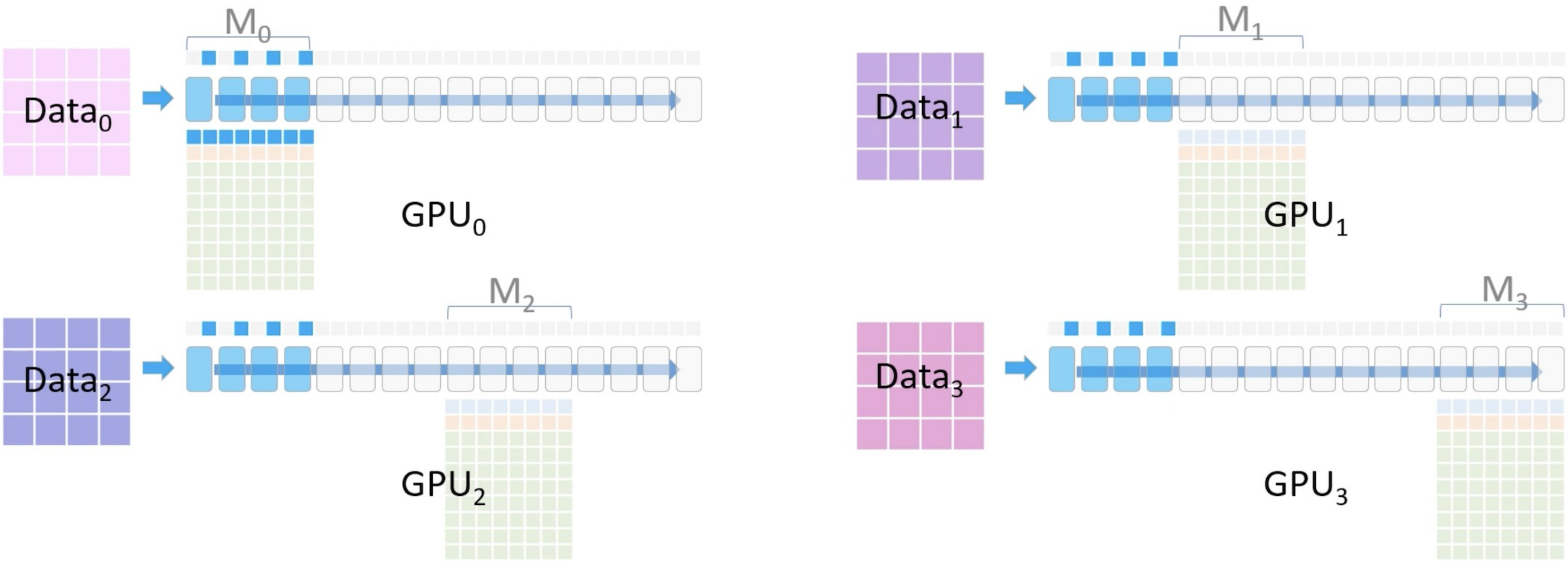


Run the forward pass

Each GPU runs on M_0's parameters using its own data

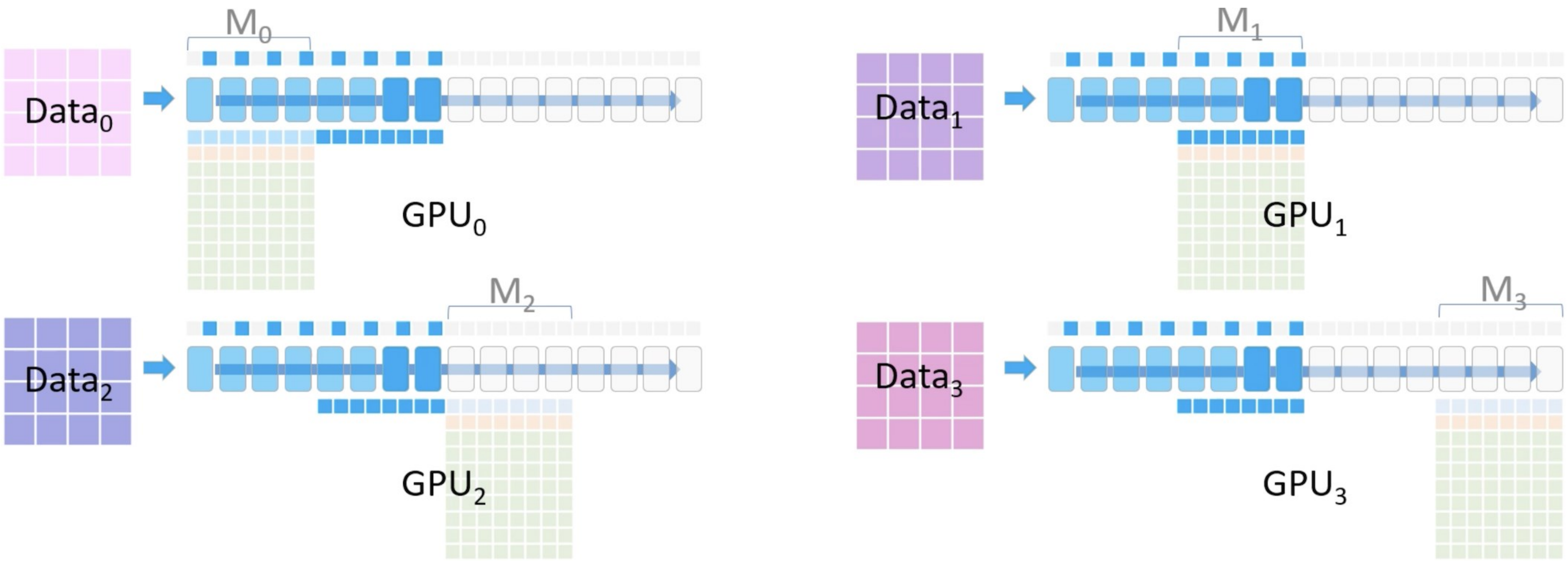
Only part of each layer's activations are retained

Adapted from a blog by MSR and slides from deeplearning.ai



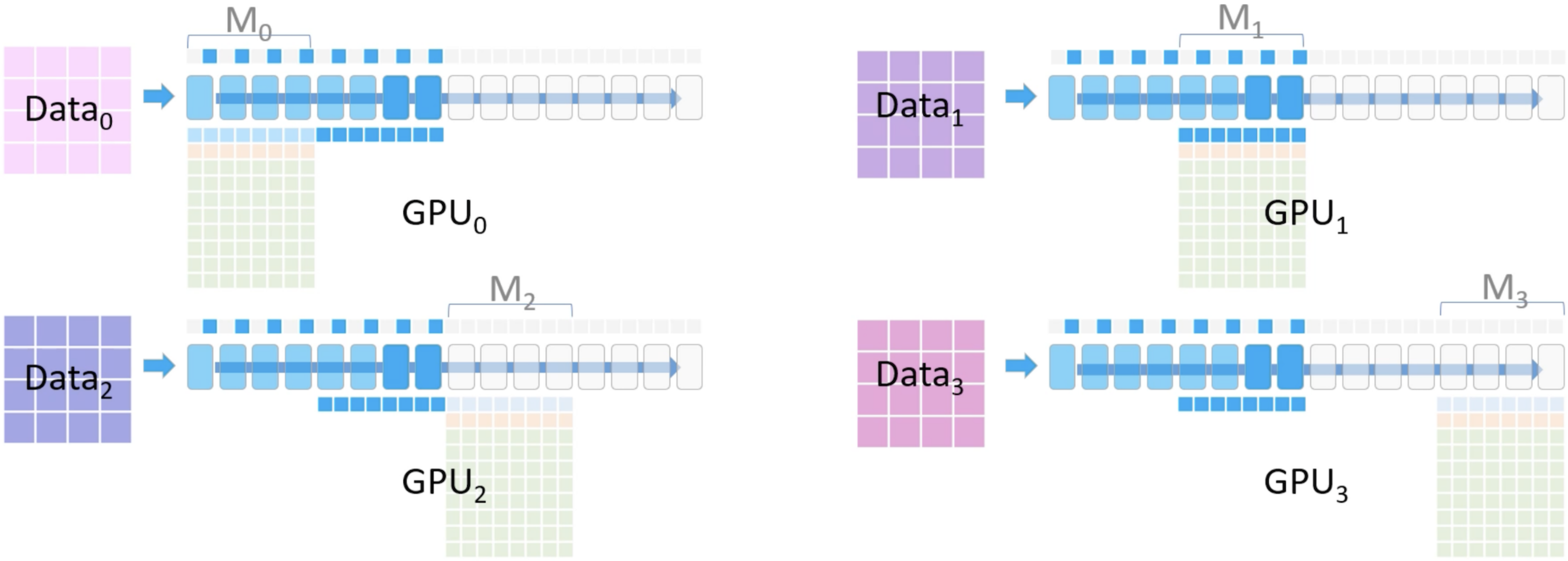
Once M_0 is complete, $GPU_{1,2,3}$ can delete the parameters for M_0

Adapted from a blog by MSR and slides from deeplearning.ai



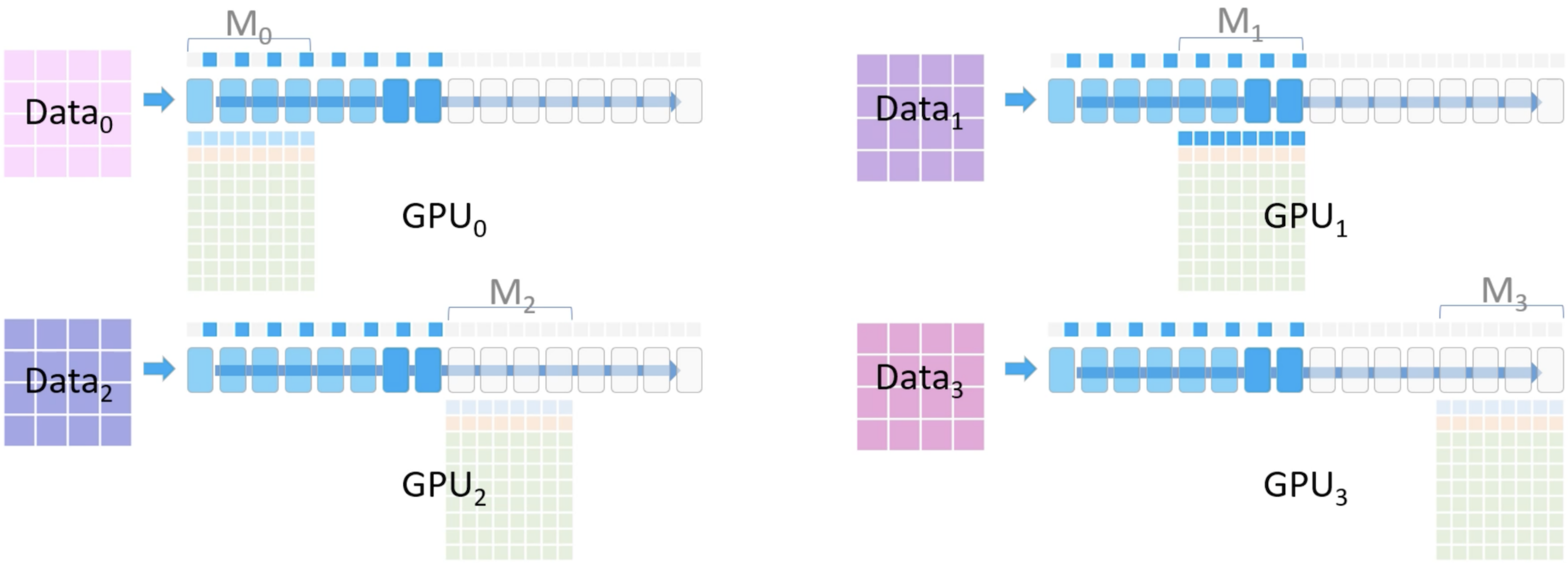
The forward pass continues across all GPUs on M_1

Adapted from a blog by MSR and slides from deeplearning.ai



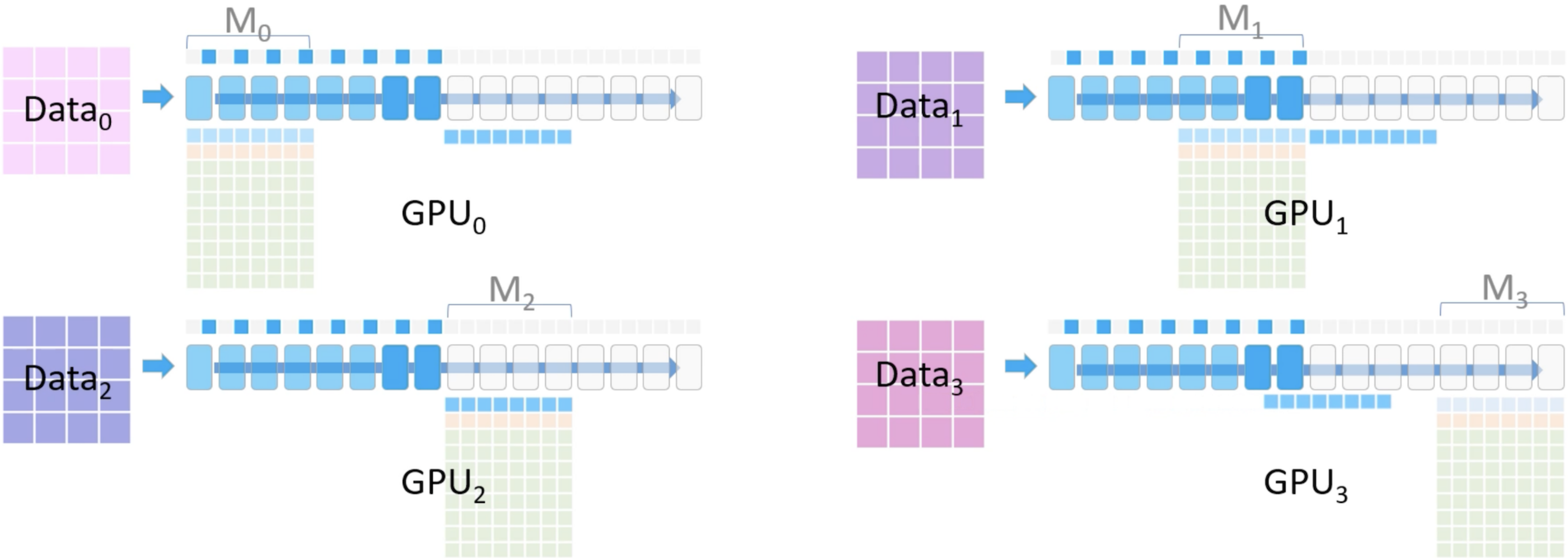
Once all GPUs have run M_1, $GPU_{0,2,3}$ can delete the parameters for M_1

Adapted from a blog by MSR and slides from deeplearning.ai



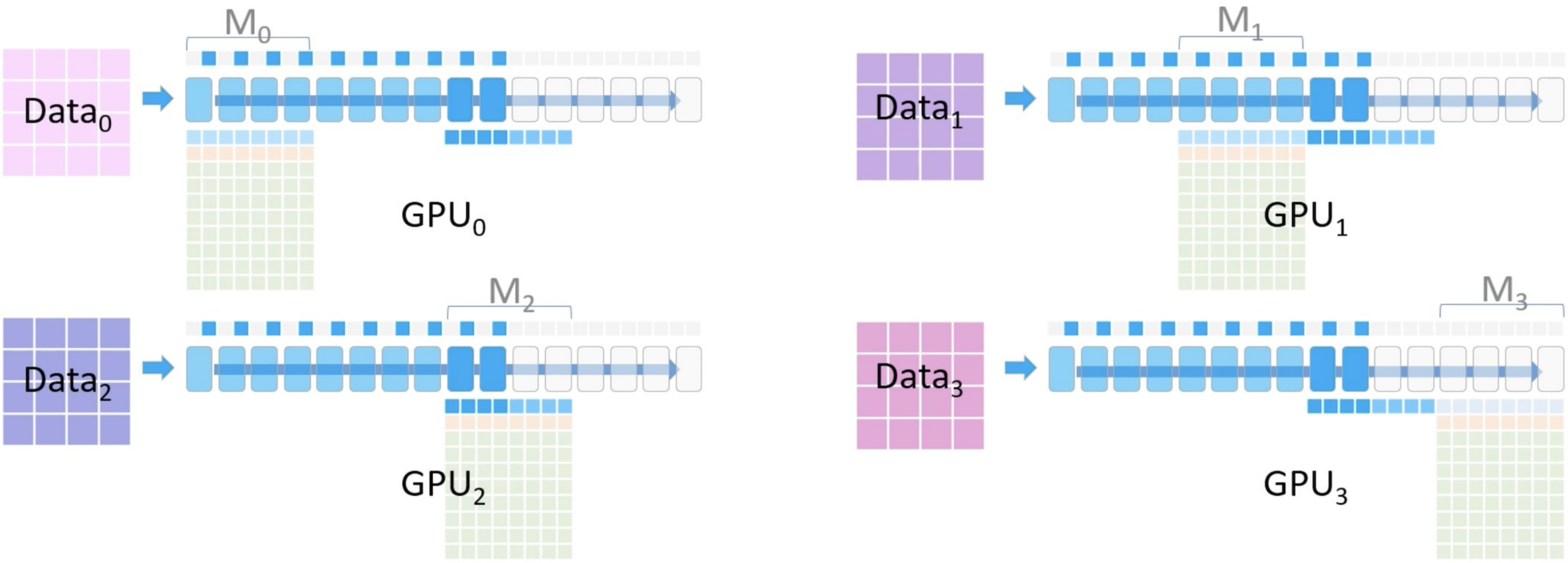
Once all GPUs have run M_1, $\text{GPU}_{0,2,3}$ can delete the parameters for M_1

Adapted from a blog by MSR and slides from deeplearning.ai



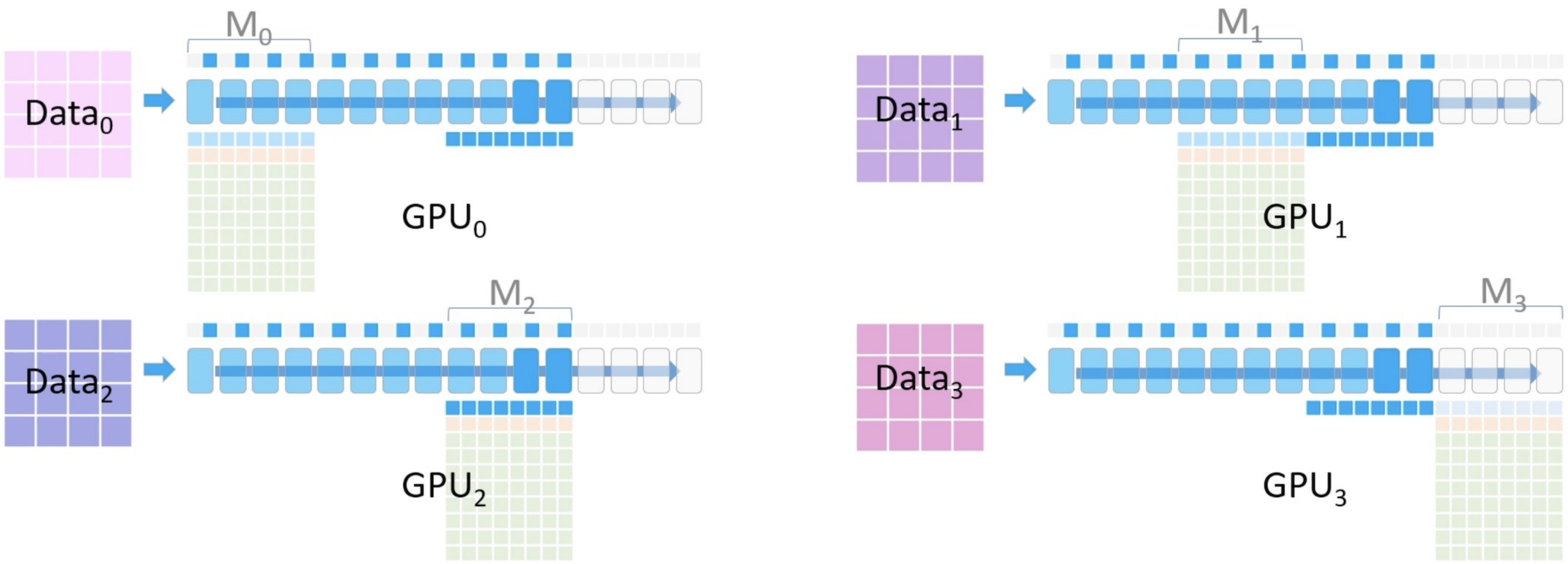
GPU_2 broadcasts the parameters for M_2

Adapted from a blog by MSR and slides from deeplearning.ai



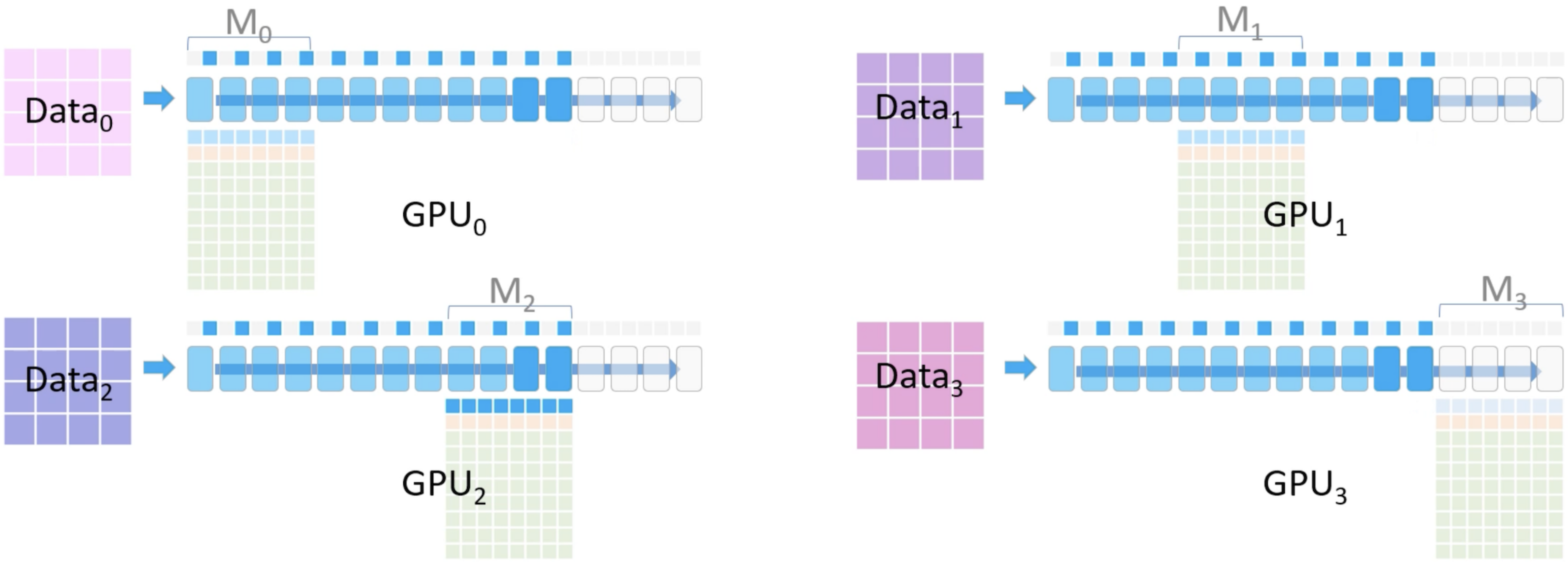
The forward pass continues across all GPUs on M_2's parameters

Adapted from a blog by MSR and slides from deeplearning.ai



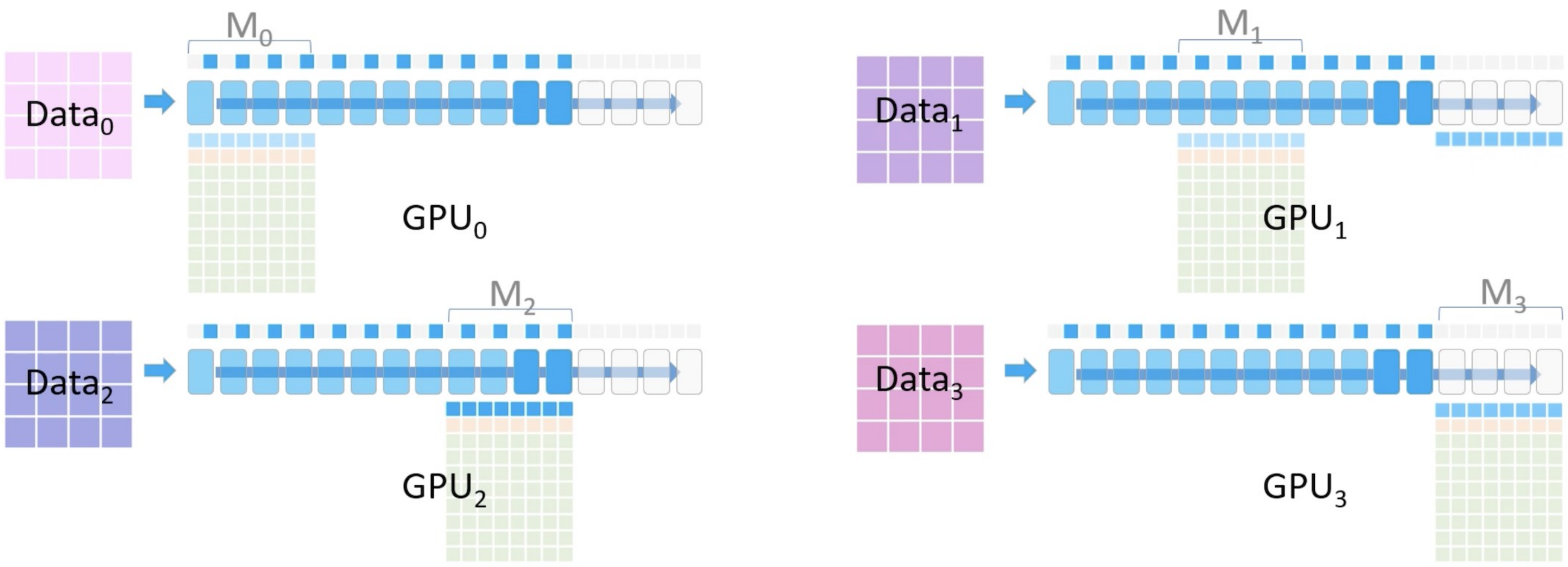
The forward pass continues across all GPUs on M_2's parameters

Adapted from a blog by MSR and slides from deeplearning.ai



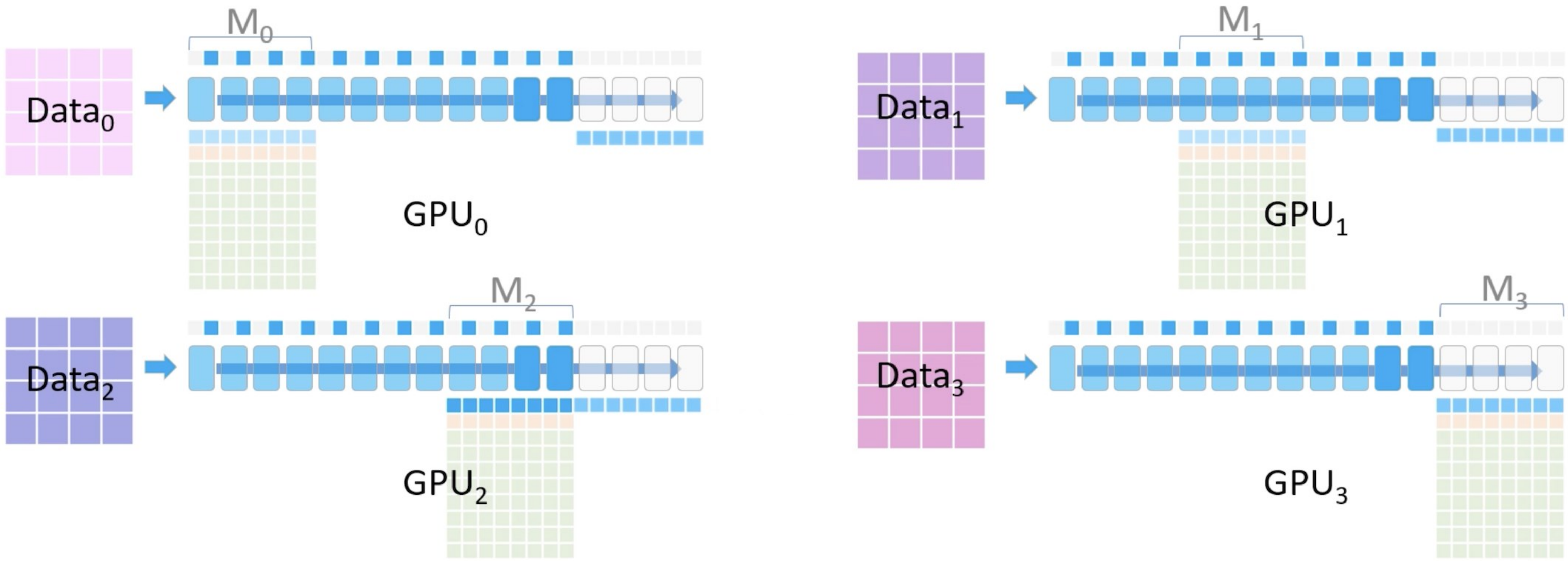
Once all GPUs have run M_2, $GPU_{0,1,3}$ can delete the parameters for M_2

Adapted from a blog by MSR and slides from deeplearning.ai



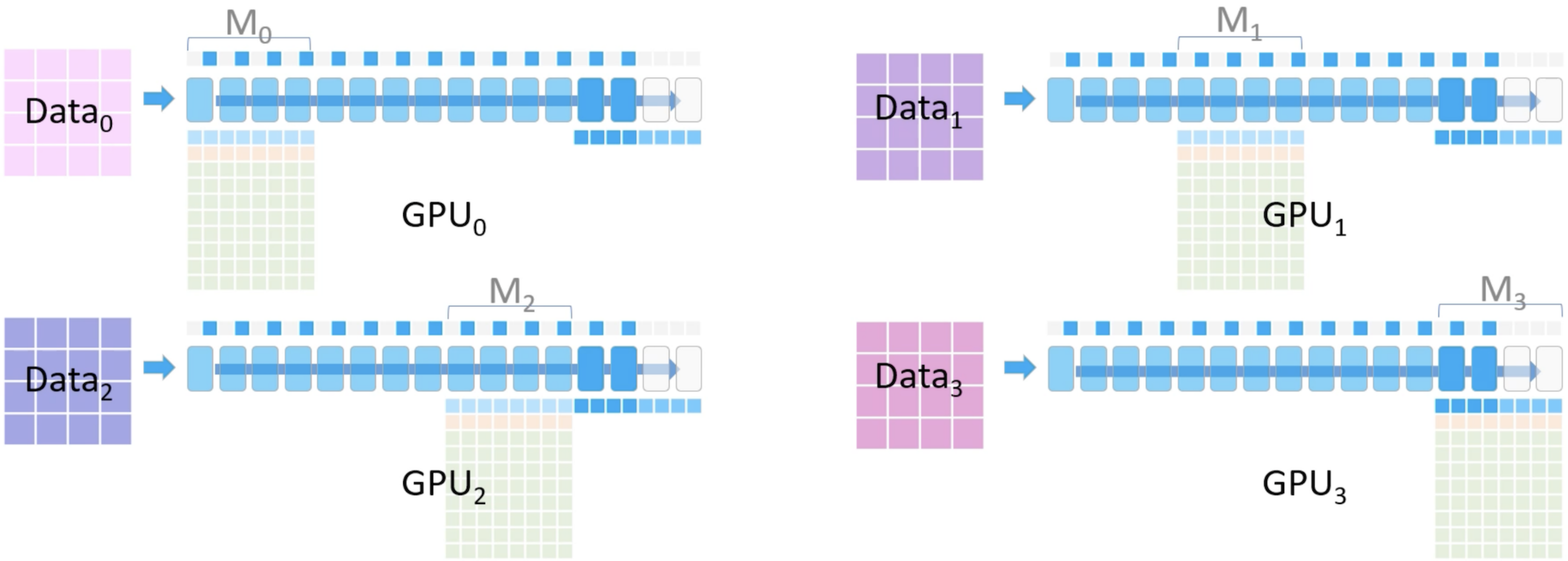
GPU$_3$ broadcasts the parameters for M$_3$

Adapted from a blog by MSR and slides from deeplearning.ai



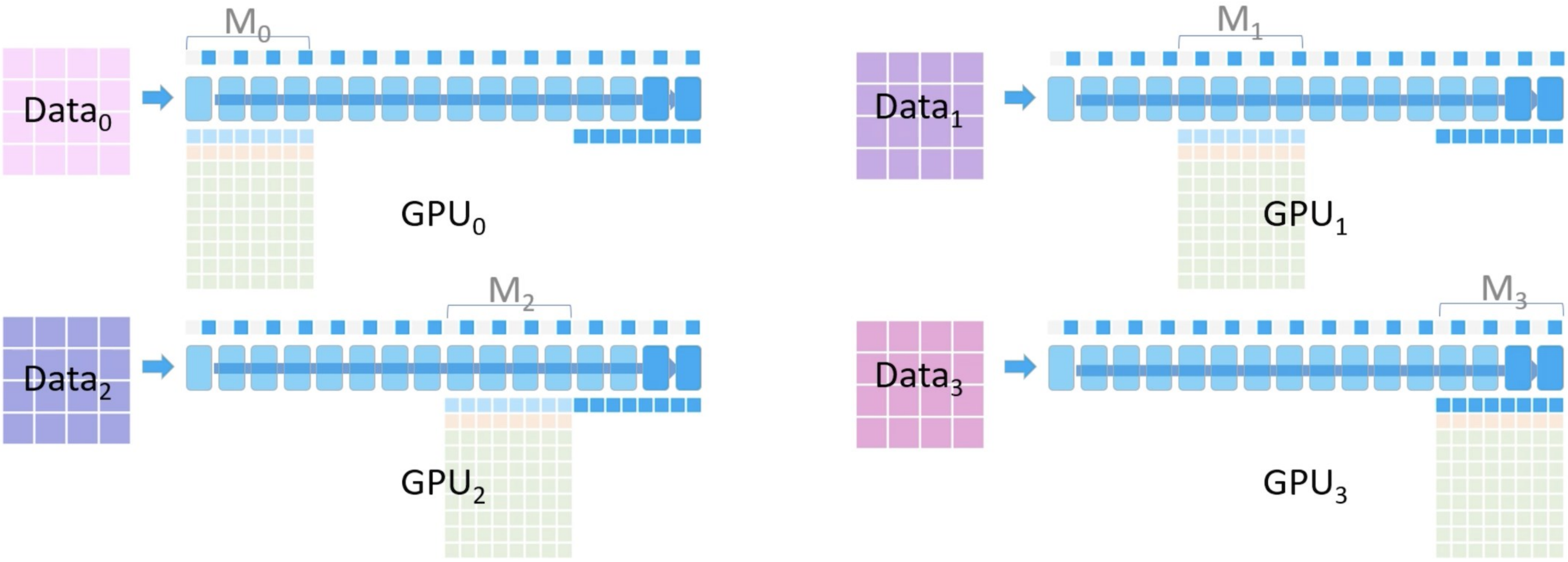
GPU_3 broadcasts the parameters for M_3

Adapted from a blog by MSR and slides from deeplearning.ai



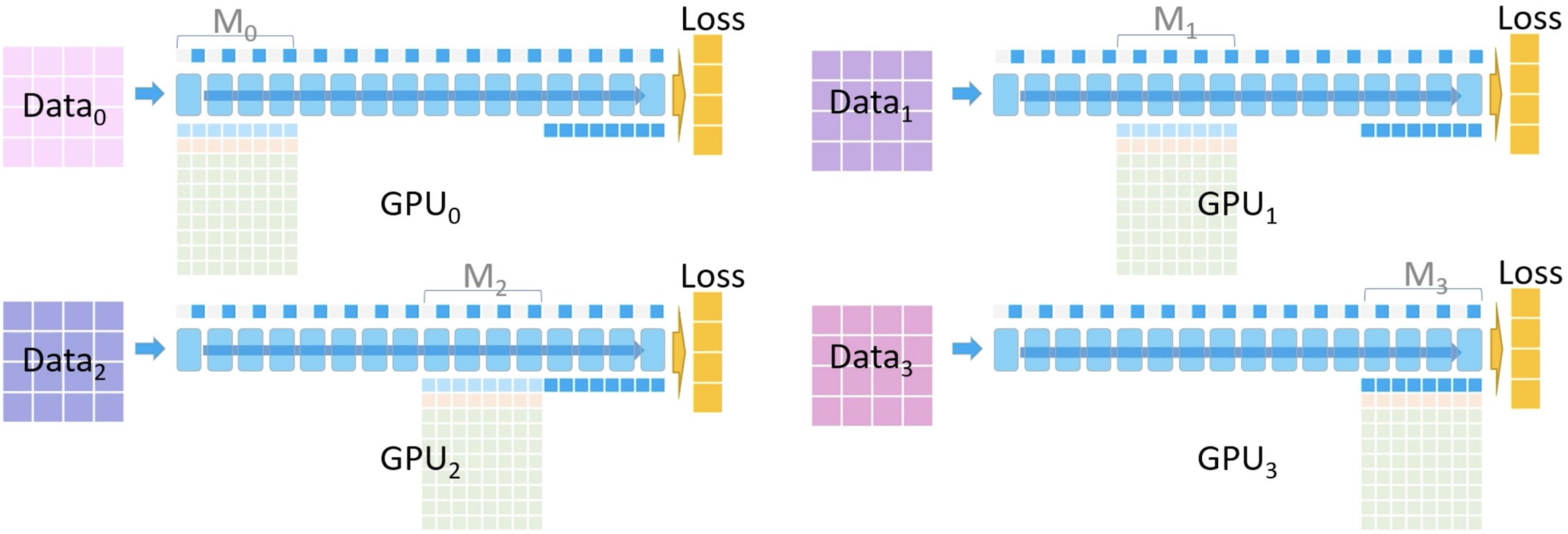
The forward pass continues on all GPUs for M_3

Adapted from a blog by MSR and slides from deeplearning.ai



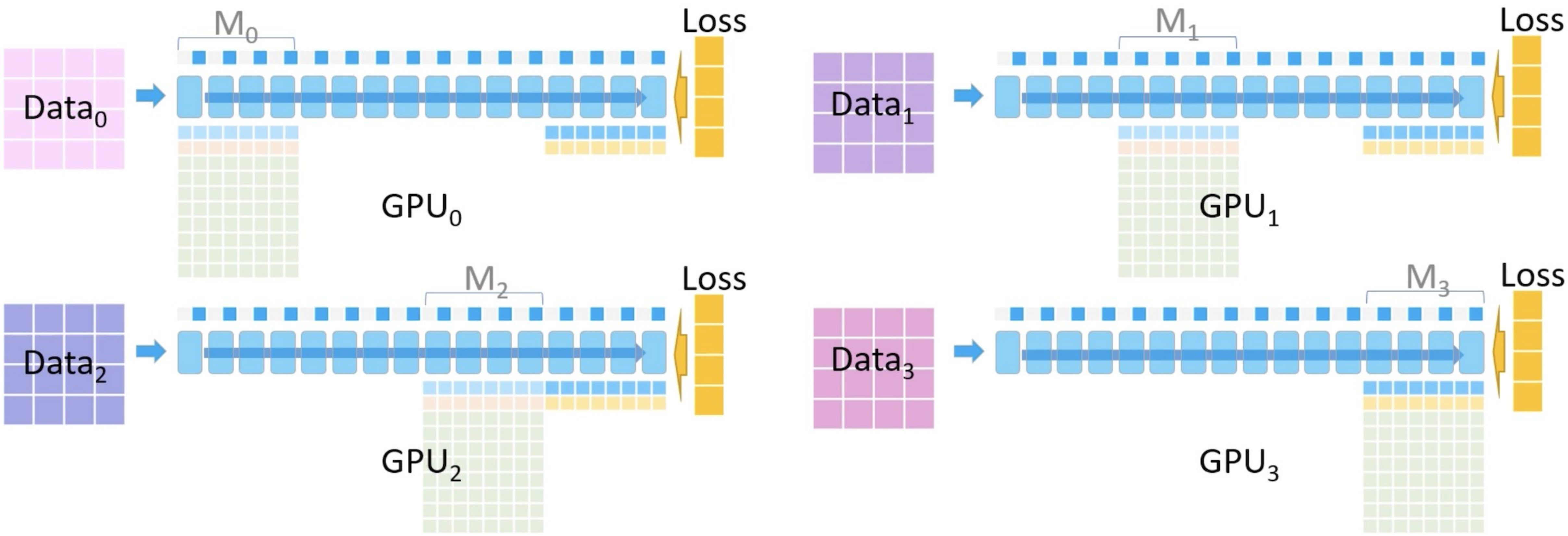
The forward pass continues on all GPUs for M_3's parameters

Adapted from a blog by MSR and slides from deeplearning.ai



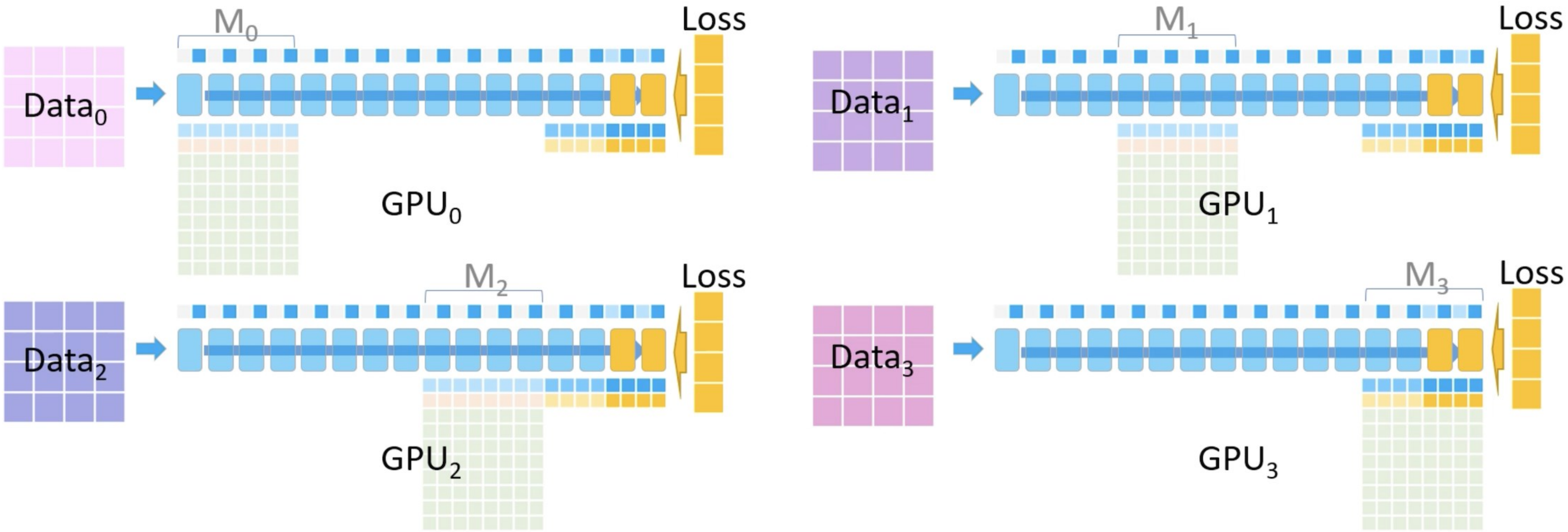
The forward pass is complete.
The loss is computed on each GPU for its respective dataset

Adapted from a blog by MSR and slides from deeplearning.ai



The backwards pass starts.
$GPU_{0,1,2}$ will hold a temporary buffer M_3 gradients on $Data_{0,1,2}$

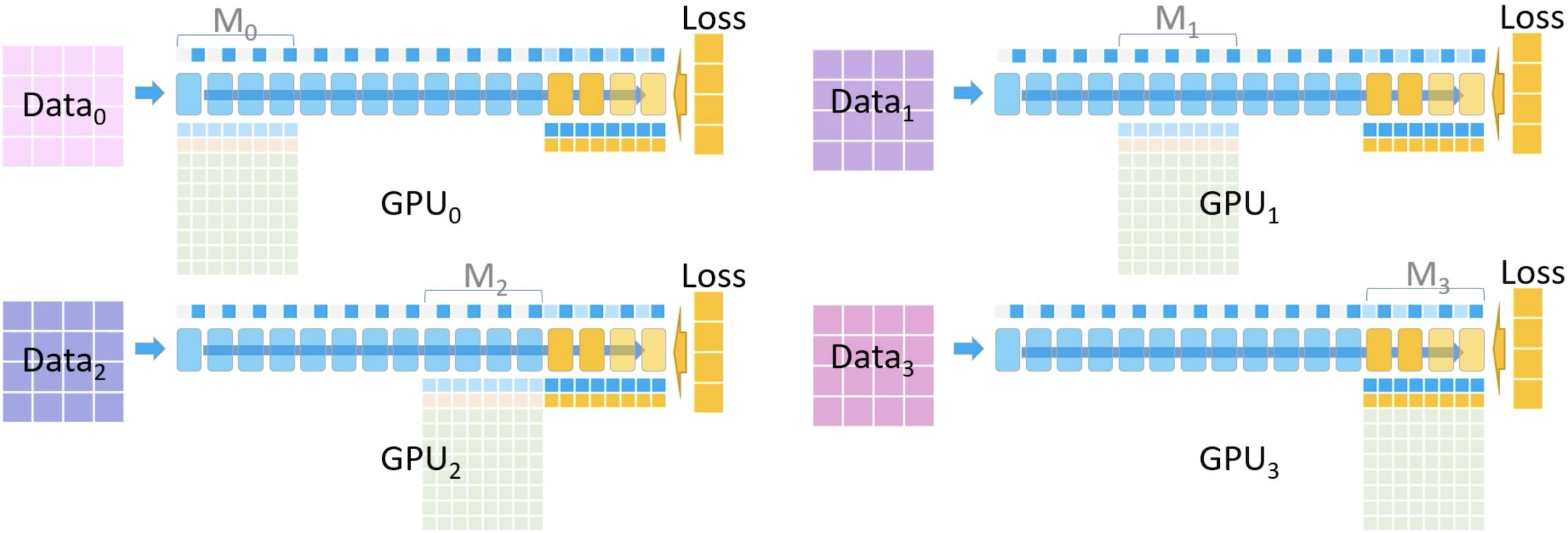
Adapted from a blog by MSR and slides from deeplearning.ai



The backwards pass proceeds on M_3

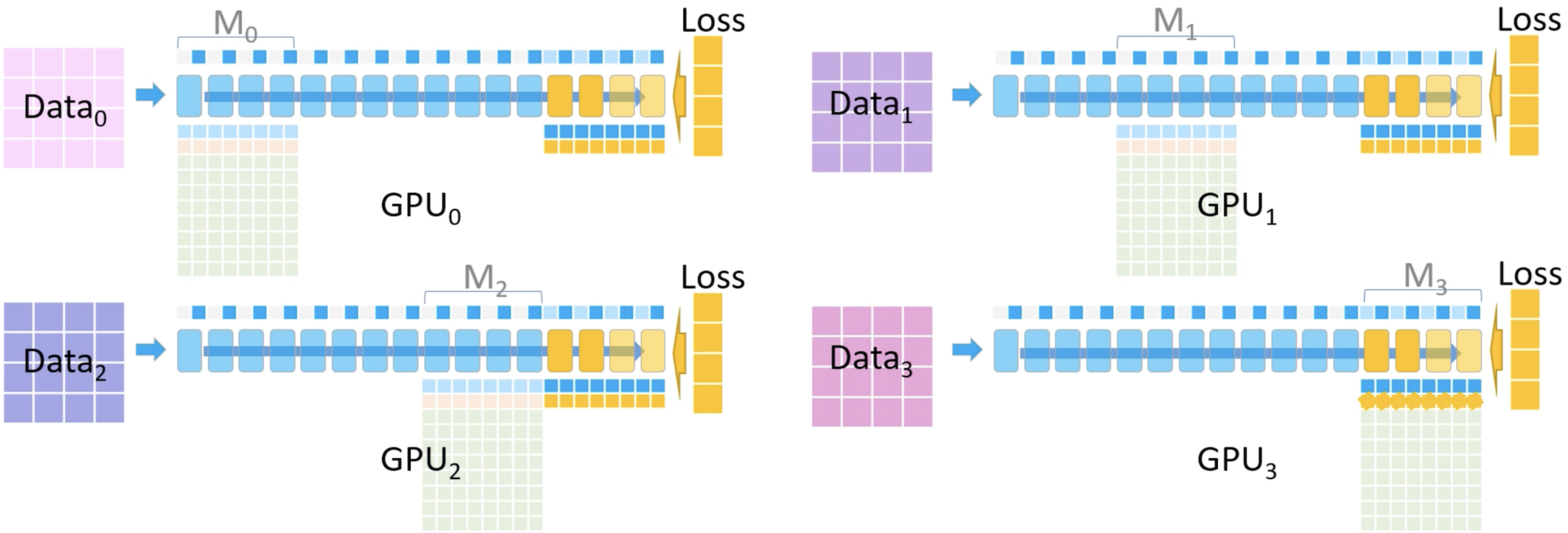
The activations for M_3 are recomputed from the saved partial activations

Adapted from a blog by MSR and slides from deeplearning.ai



The backwards pass proceeds on M_3

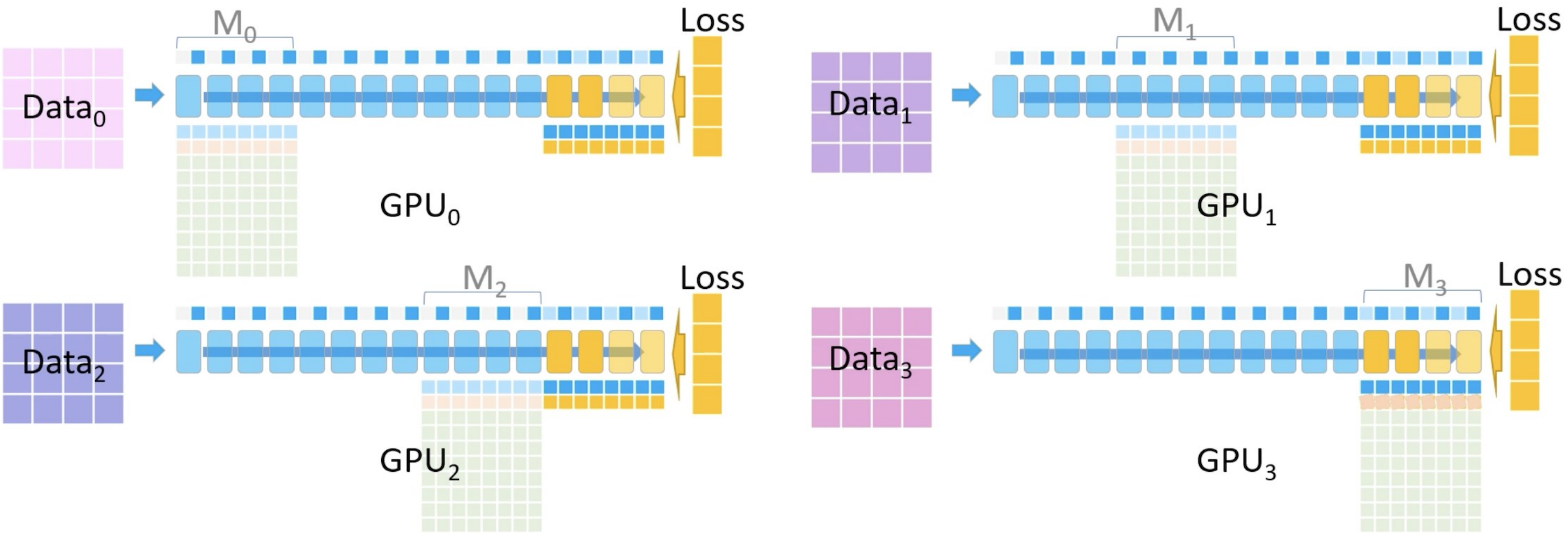
Adapted from a blog by MSR and slides from deeplearning.ai



$GPU_{0,1,2}$ pass their M_3 gradients to GPU_3

GPU_3 performs gradient accumulation and holds final M_3 for all Data

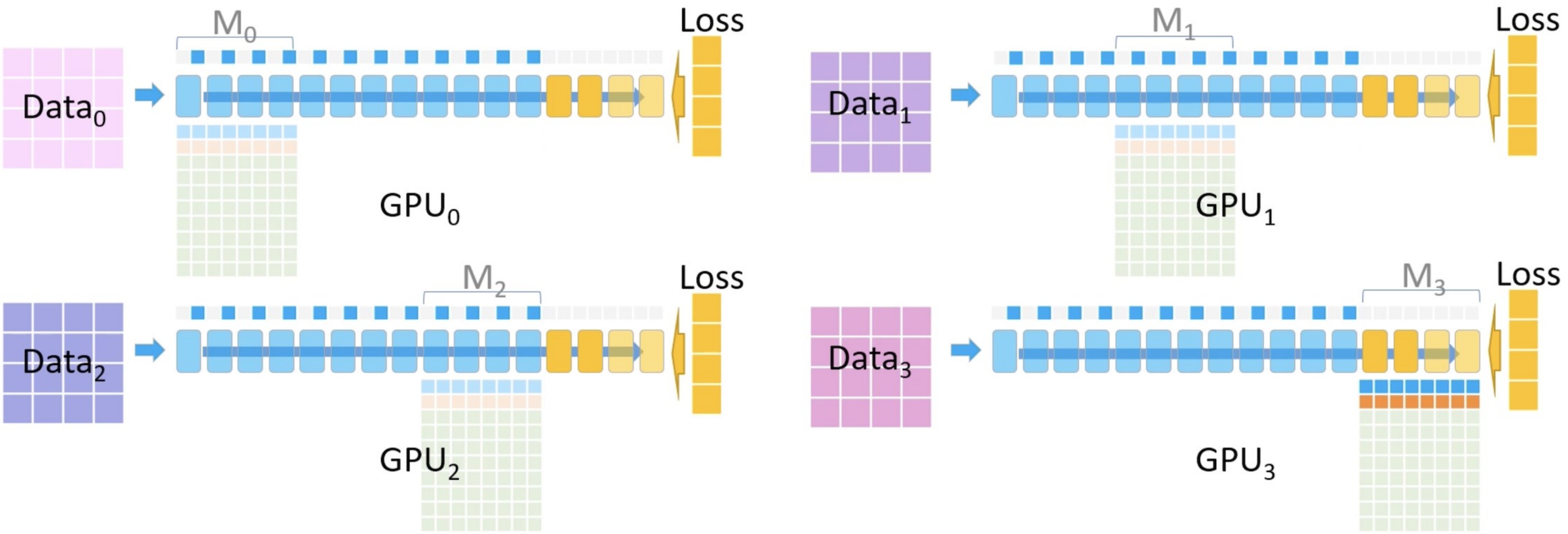
Adapted from a blog by MSR and slides from deeplearning.ai



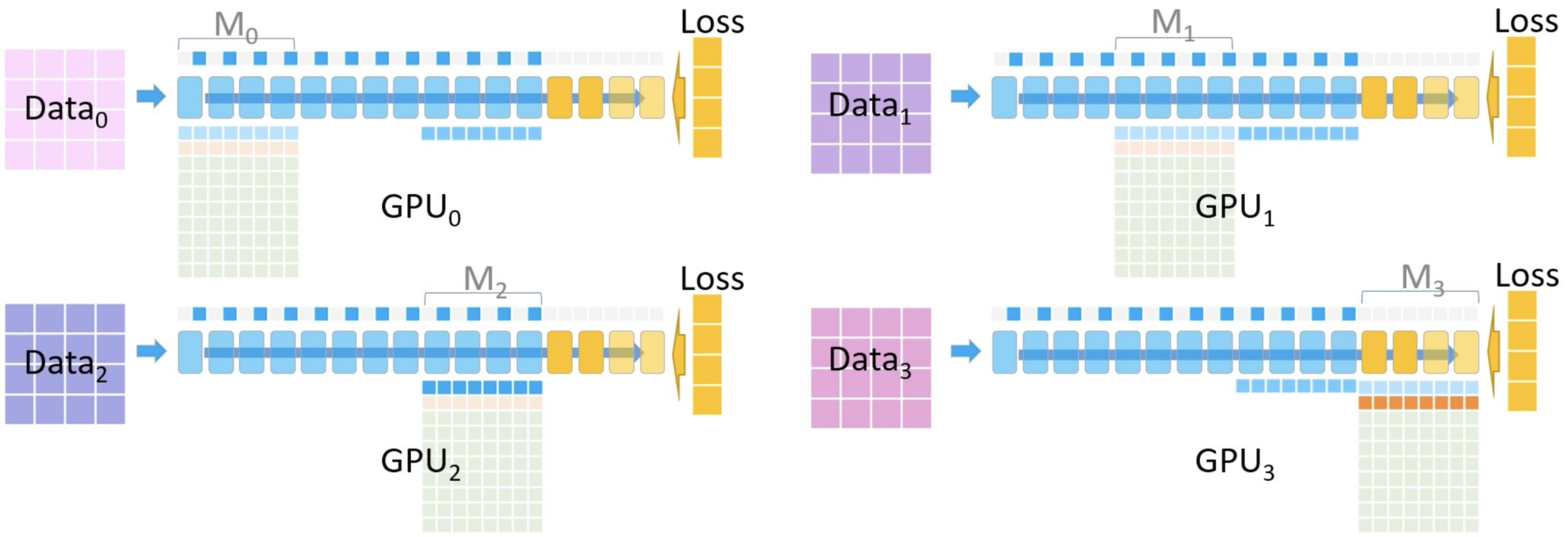
$GPU_{0,1,2}$ pass their M_3 gradients to GPU_3

GPU_3 performs gradient accumulation and holds final M_3 for all Data

Adapted from a blog by MSR and slides from deeplearning.ai

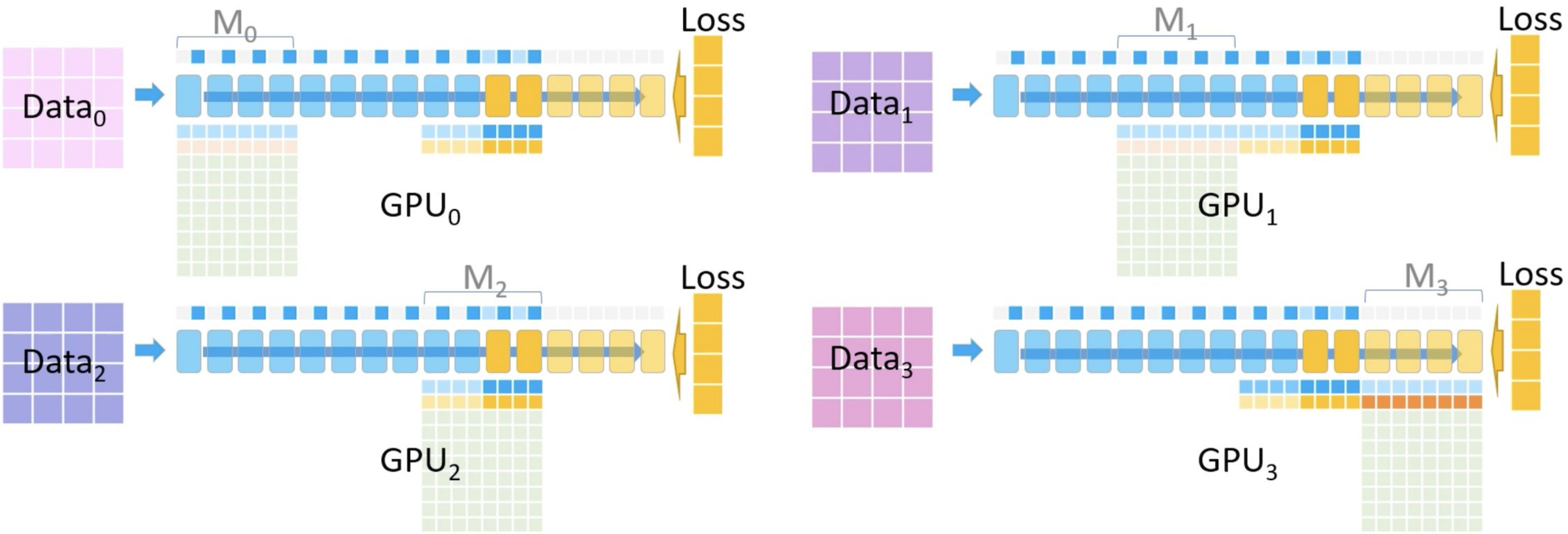


$GPU_{0,1,2}$ delete their temporary M_3 gradients and parameters.
All M_3 activations are deleted



GPU_2 passes M_2's parameters to $GPU_{0,1,3}$ so they can run the backwards pass and compute gradients for M_2

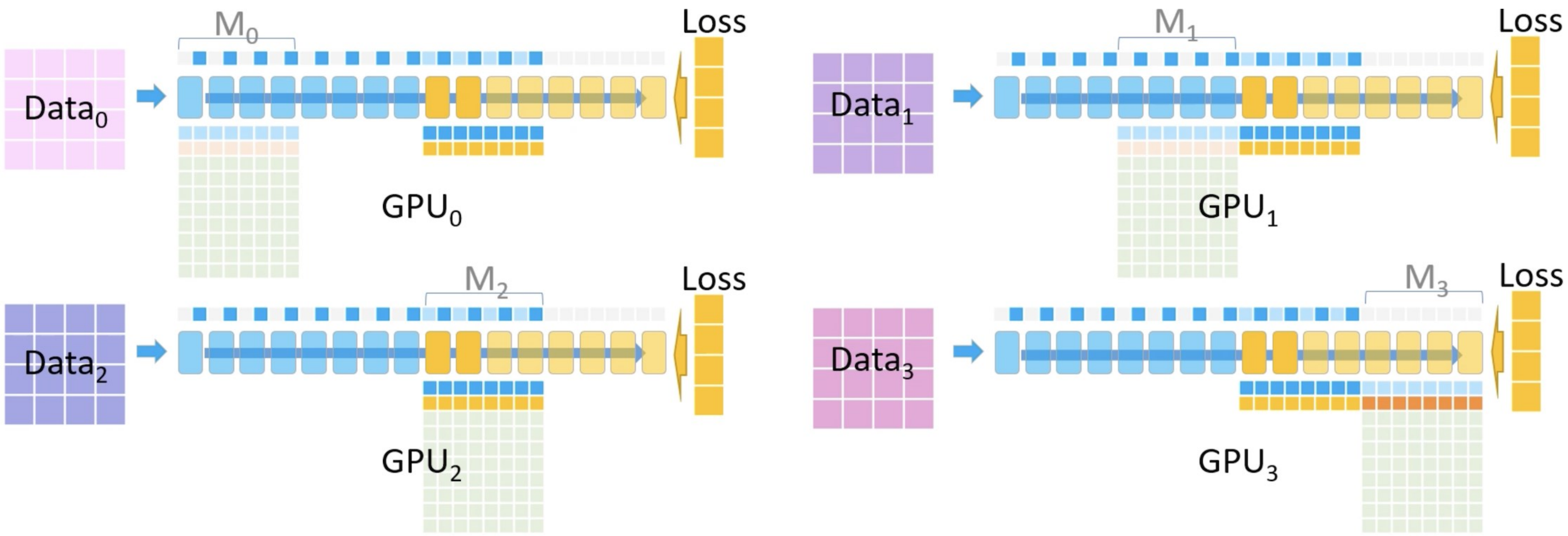
Adapted from a blog by MSR and slides from deeplearning.ai



The backward pass continues on M_2

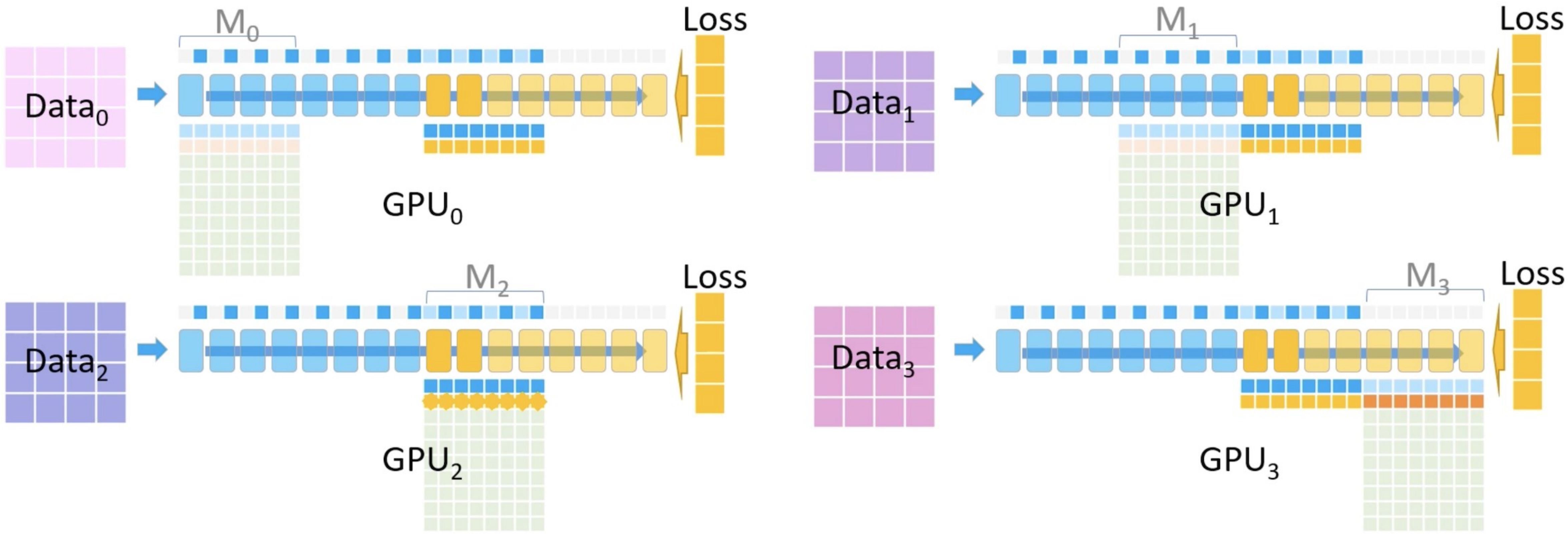
The activations for M_2 are recomputed from the saved partial activations

Adapted from a blog by MSR and slides from deeplearning.ai



The backward pass continues on M_2

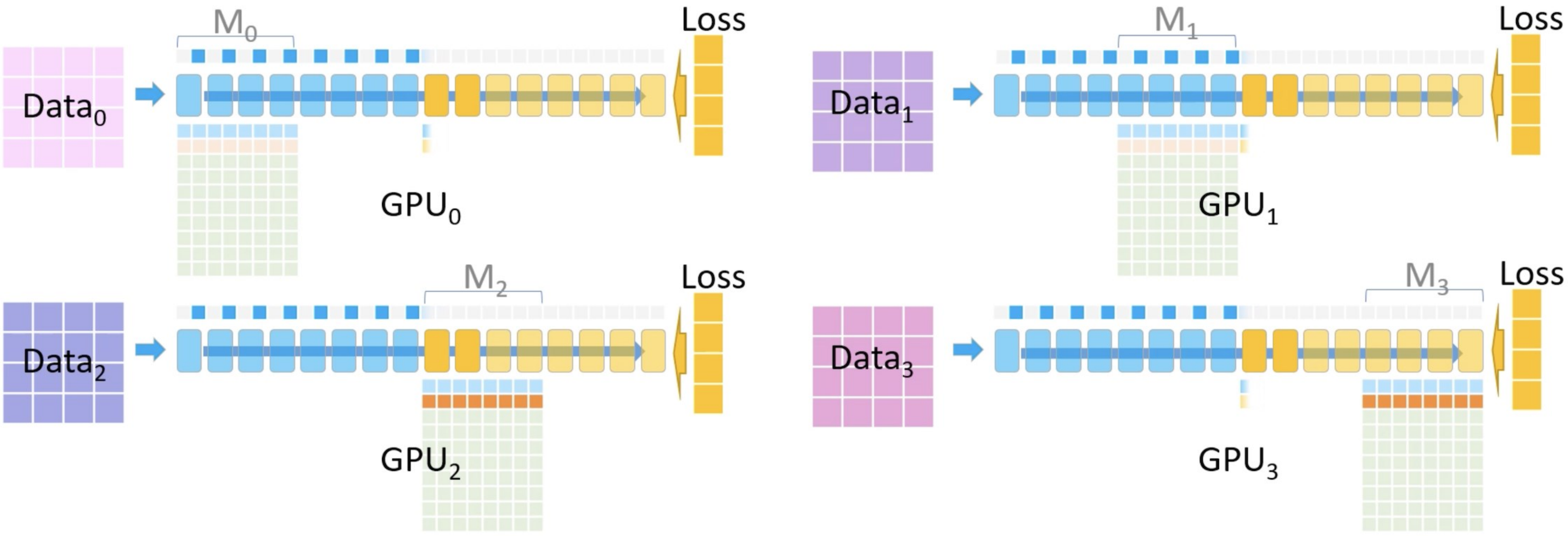
Adapted from a blog by MSR and slides from deeplearning.ai



$GPU_{0,1,3}$ pass their M_2 gradients to GPU_2

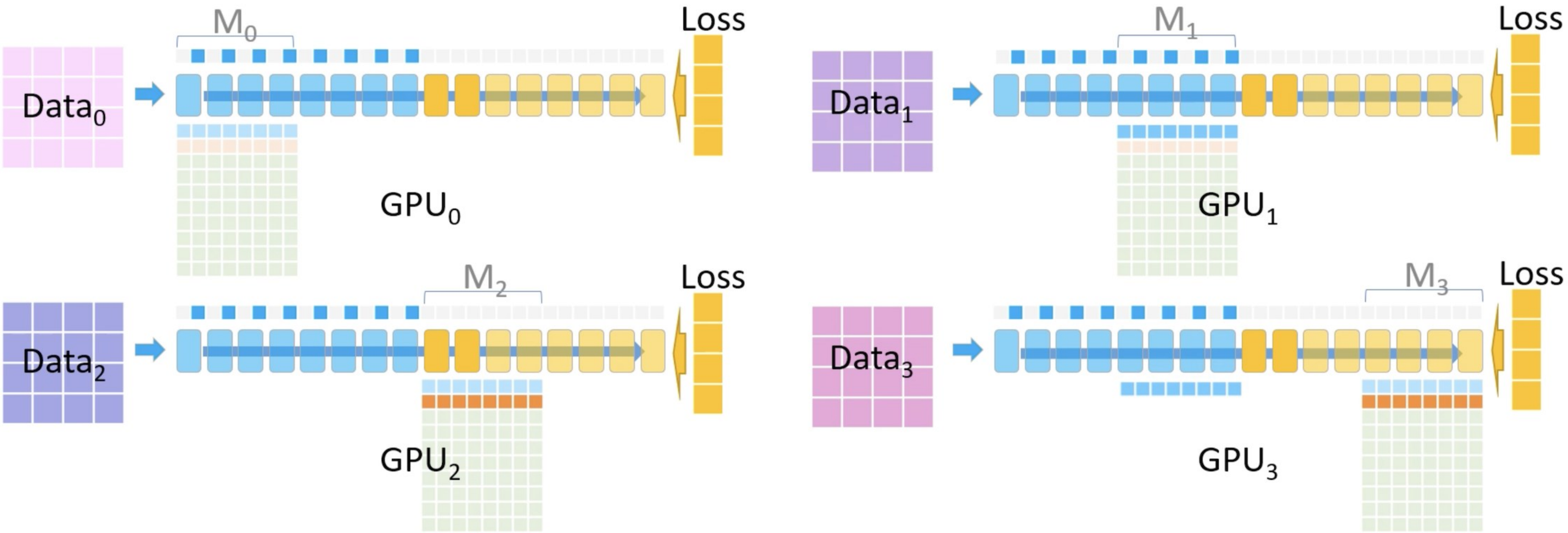
GPU_2 performs gradient accumulation and holds final M_2 gradients for all Data

Adapted from a blog by MSR and slides from deeplearning.ai

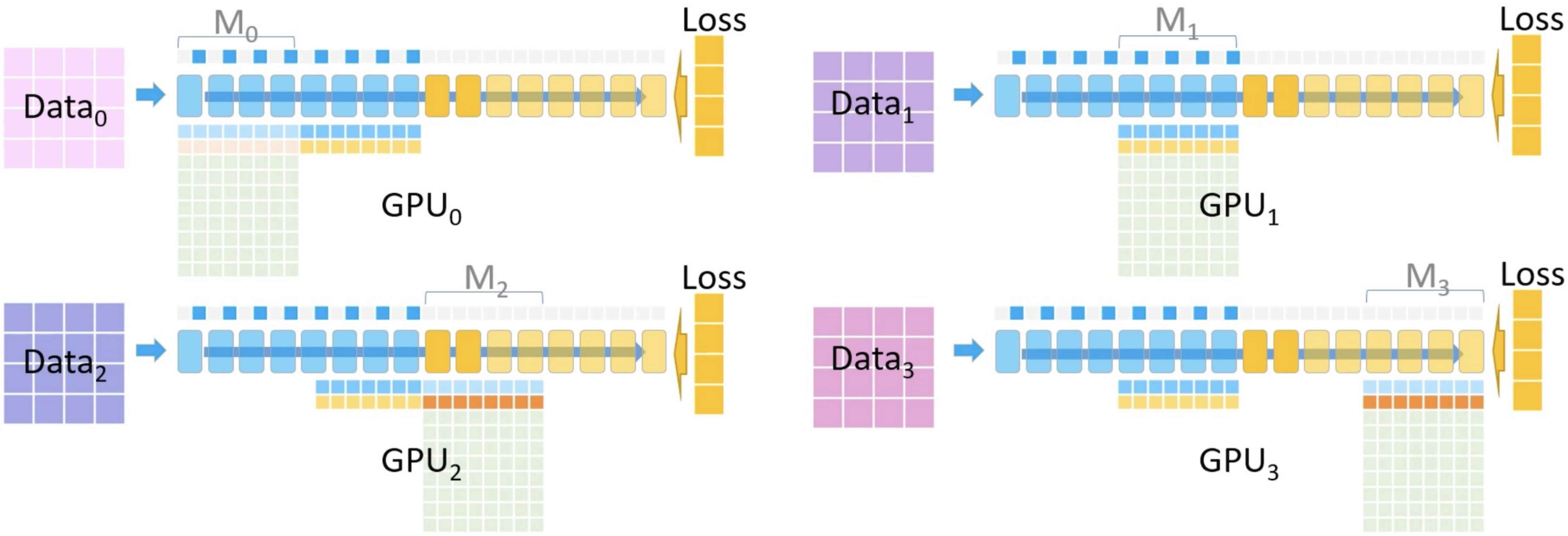


$GPU_{0,1,3}$ can delete their temporary M_2 gradients and parameters.
All M_2 activations are deleted

Adapted from a blog by MSR and slides from deeplearning.ai

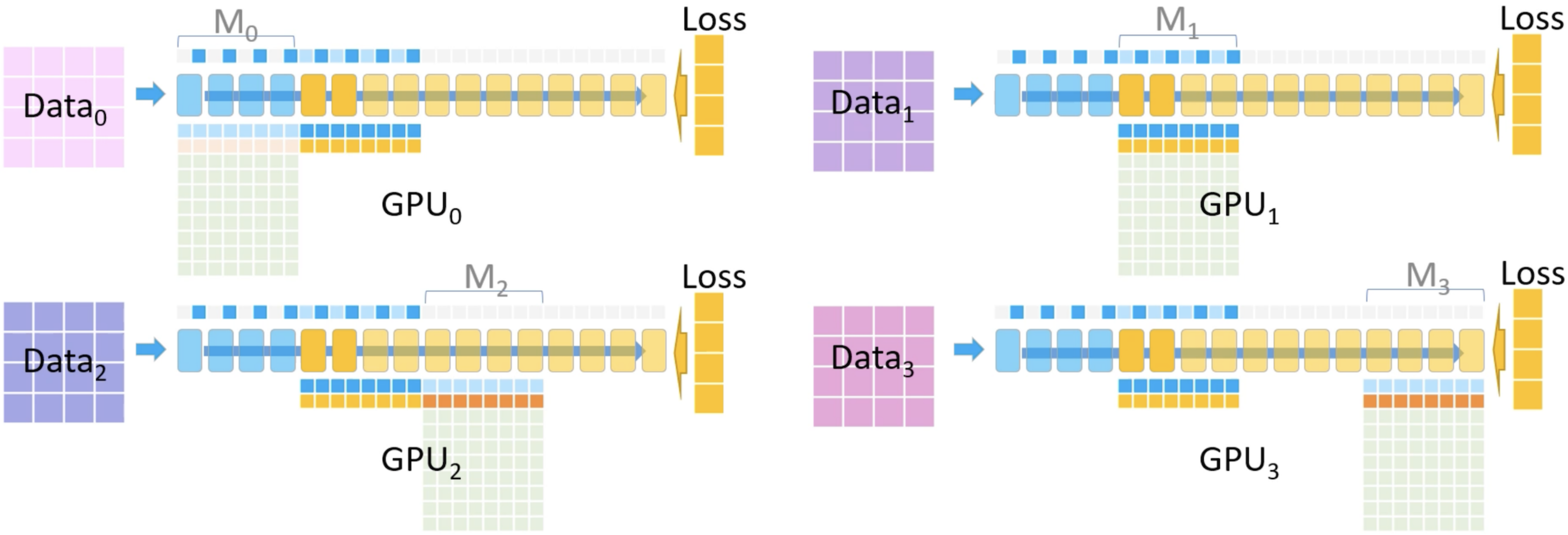


GPU_1 passes M_1's parameters to $GPU_{0,2,3}$ so they can run the backwards pass and compute gradients for M_1



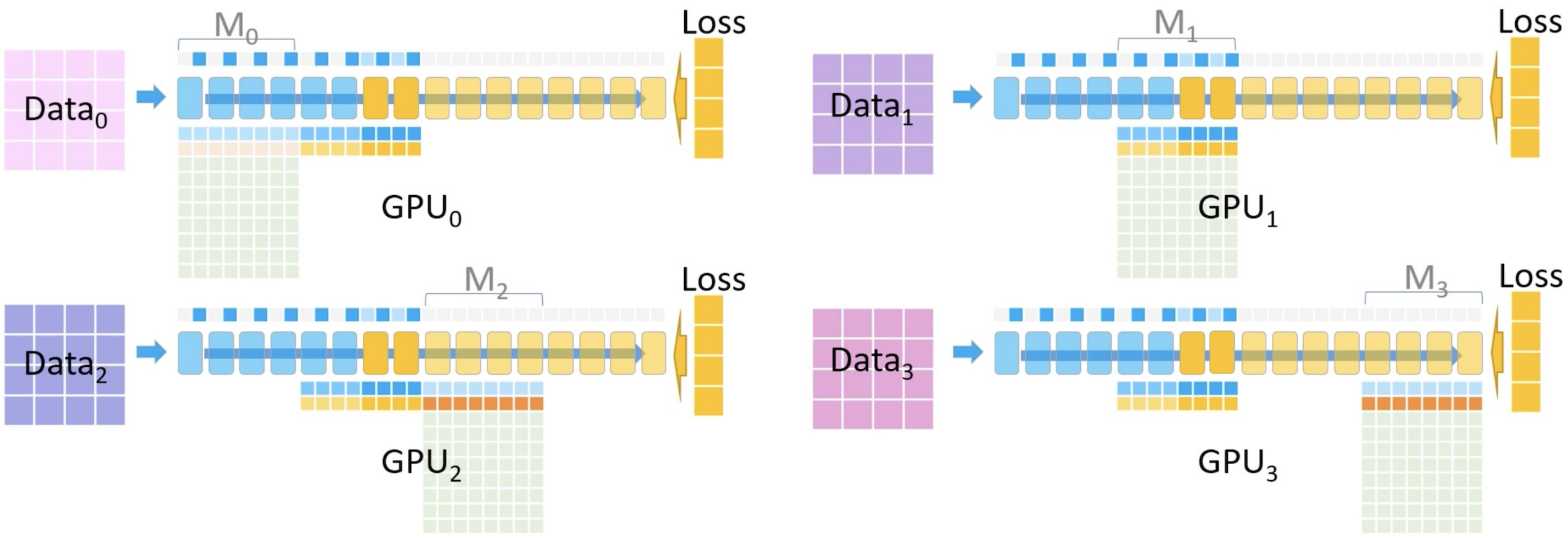
$GPU_{0,2,3}$ use temporary buffers to hold M_1's gradients

Adapted from a blog by MSR and slides from deeplearning.ai



The backward pass continues on M_1

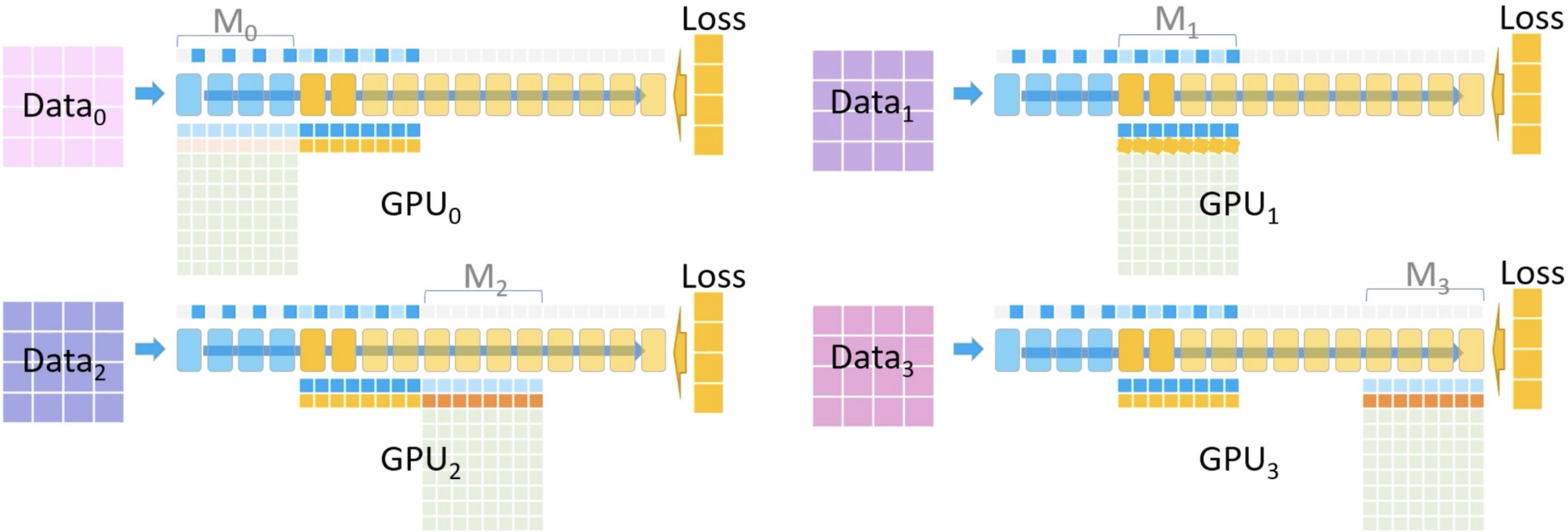
Adapted from a blog by MSR and slides from deeplearning.ai



The backward pass continues on M_1

The activations for M_1 are recomputed from the saved partial activations

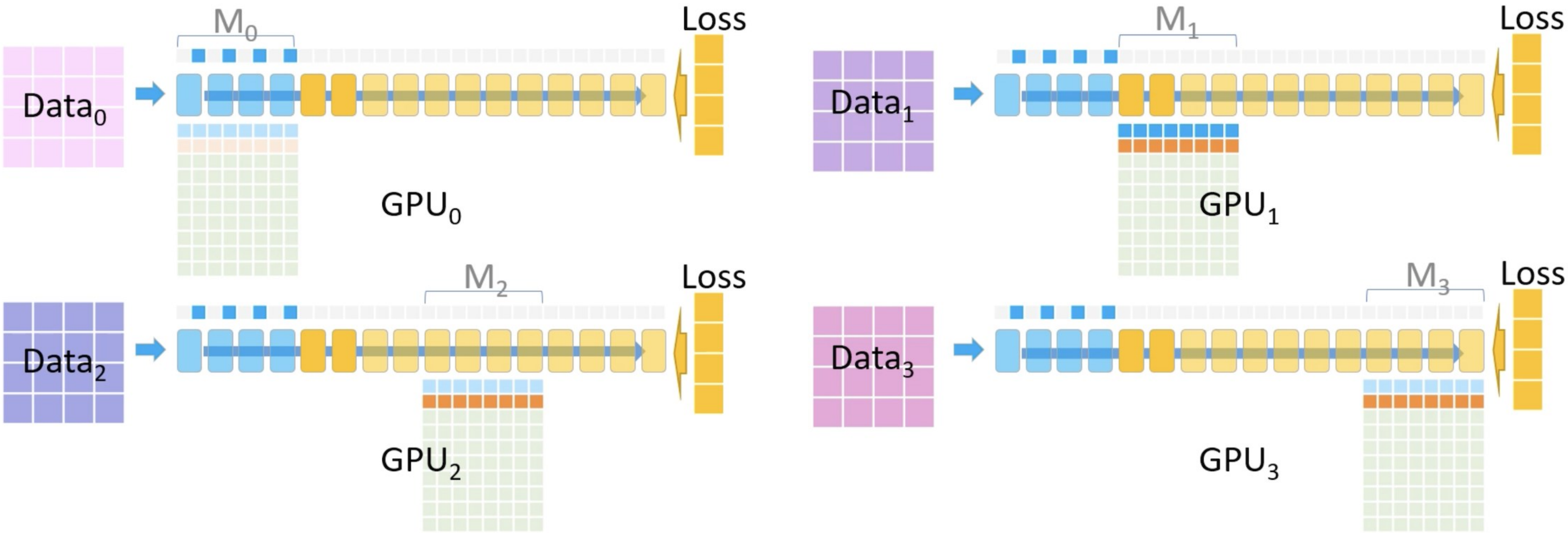
Adapted from a blog by MSR and slides from deeplearning.ai



GPU$_{0,2,3}$ pass their M$_1$ gradients to GPU$_1$

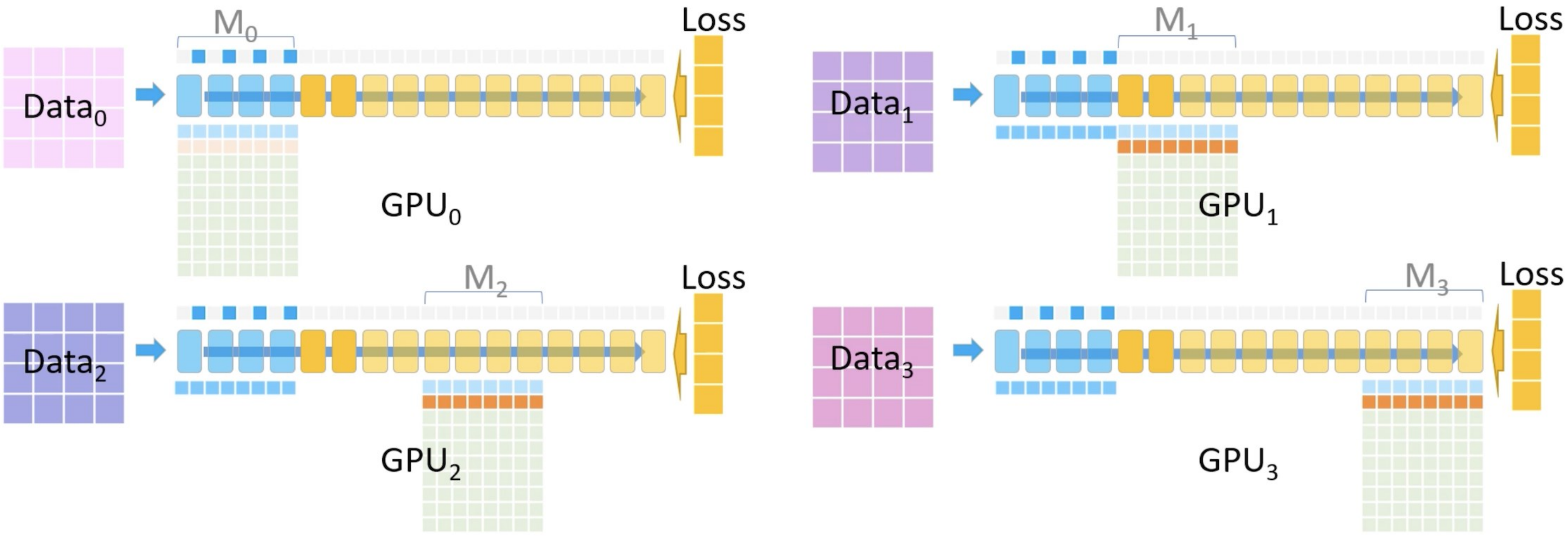
GPU$_1$ performs gradient accumulation and holds final M$_1$ gradients for all Data

Adapted from a blog by MSR and slides from deeplearning.ai



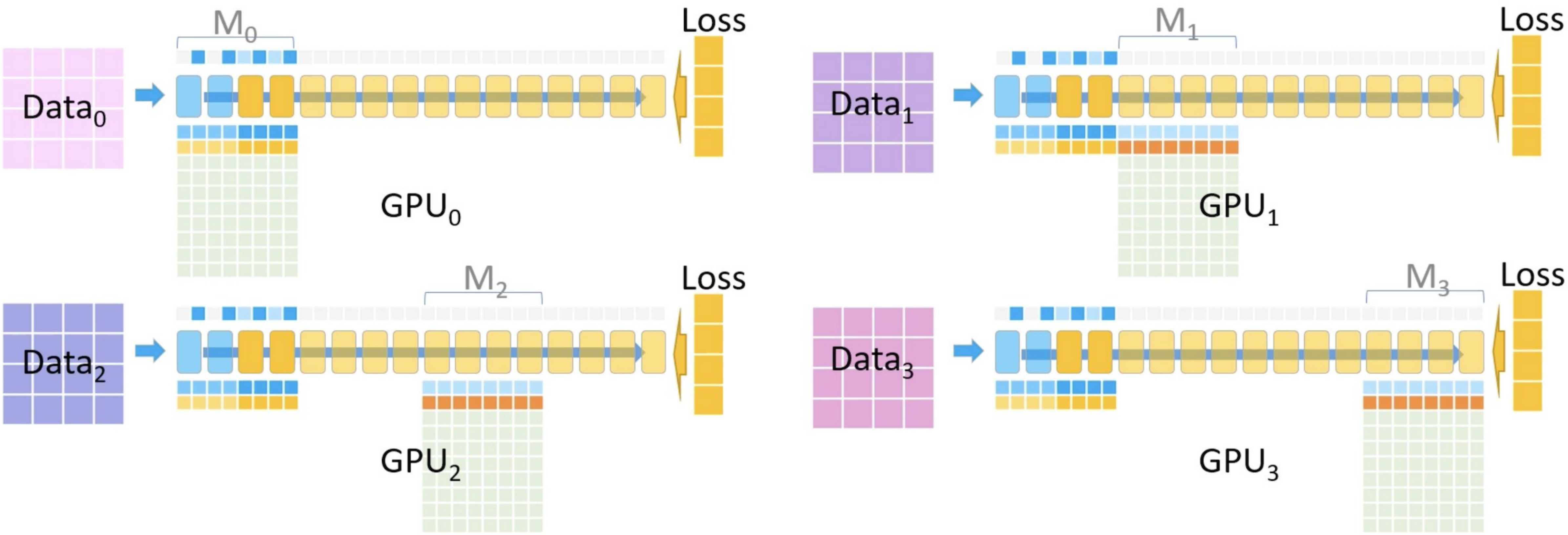
$GPU_{0,2,3}$ can delete their temporary M_1 gradients and parameters.
All M_1 activations are deleted

Adapted from a blog by MSR and slides from deeplearning.ai



GPU$_0$ passes M$_0$'s parameters to GPU$_{1,2,3}$ so they can run the backwards pass and compute gradients for M$_0$

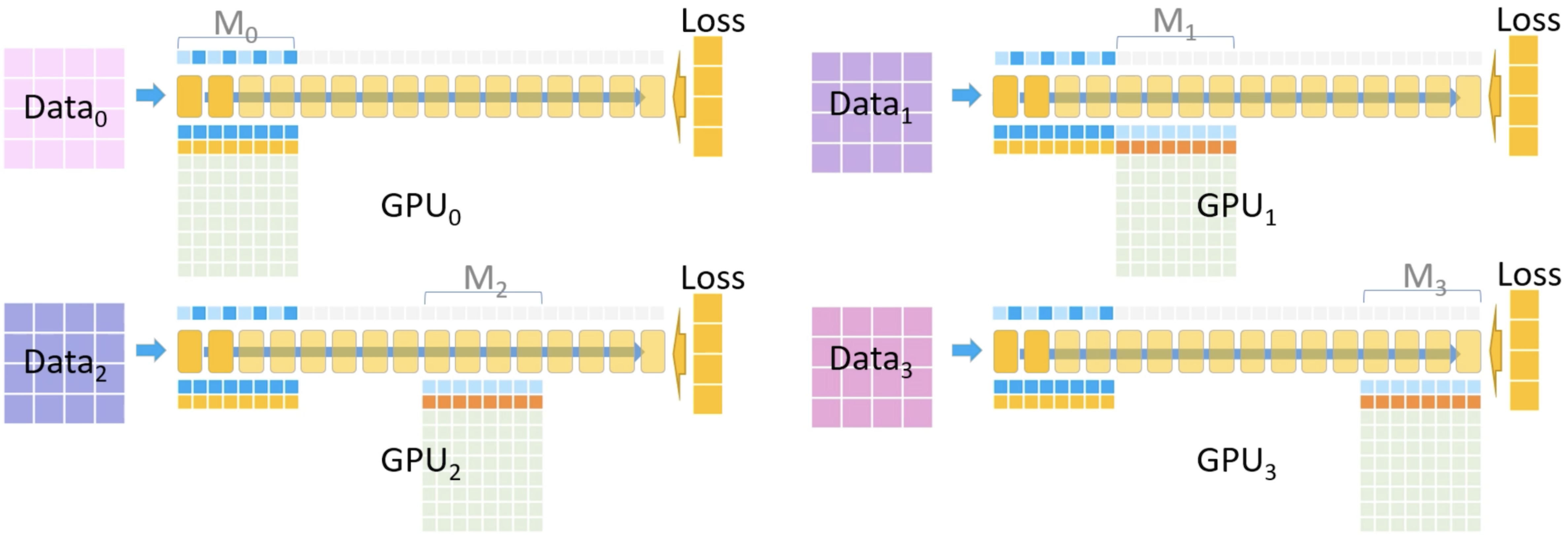
Adapted from a blog by MSR and slides from deeplearning.ai



The backward pass continues on M_0

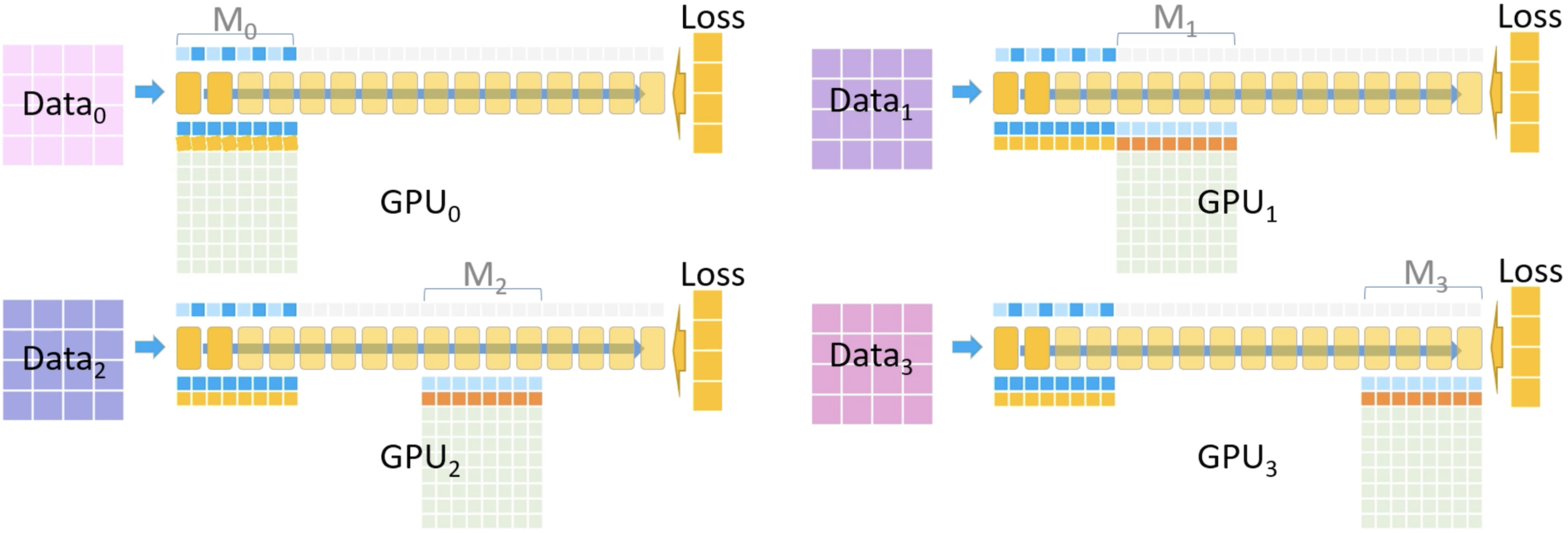
The activations for M_0 are recomputed from the saved partial activations

Adapted from a blog by MSR and slides from deeplearning.ai



The backward pass continues on M_0

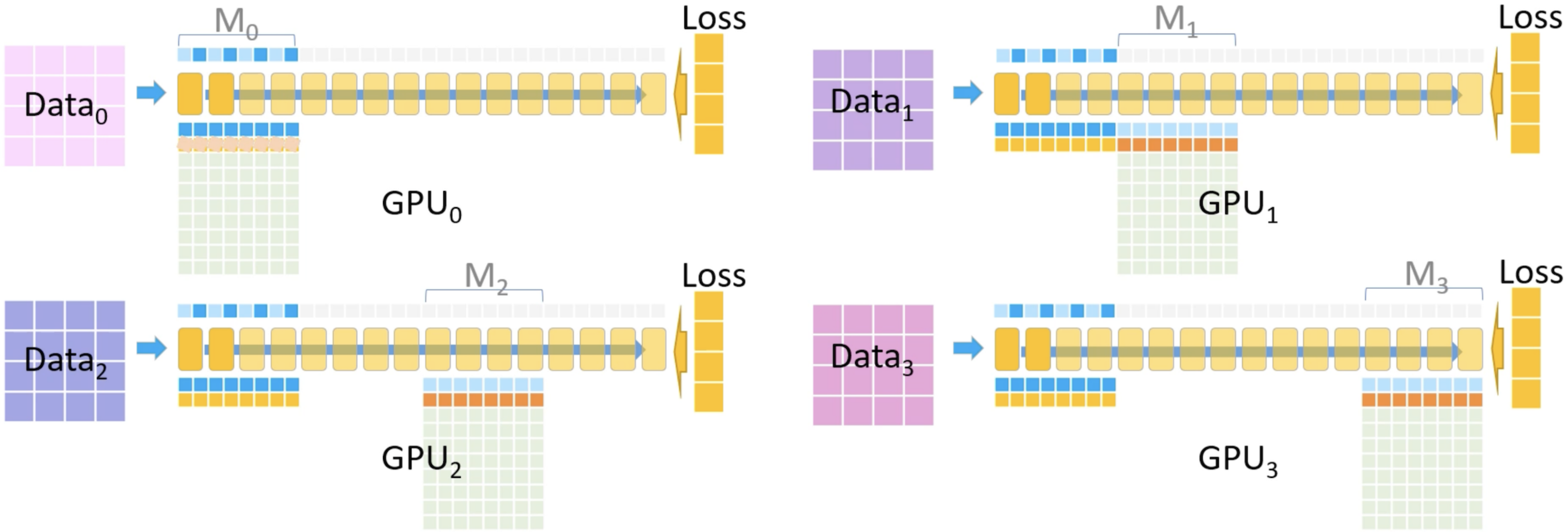
Adapted from a blog by MSR and slides from deeplearning.ai



$GPU_{1,2,3}$ pass their M_0 gradients to GPU_0

GPU_0 performs gradient accumulation and holds final M_0 gradients for all Data

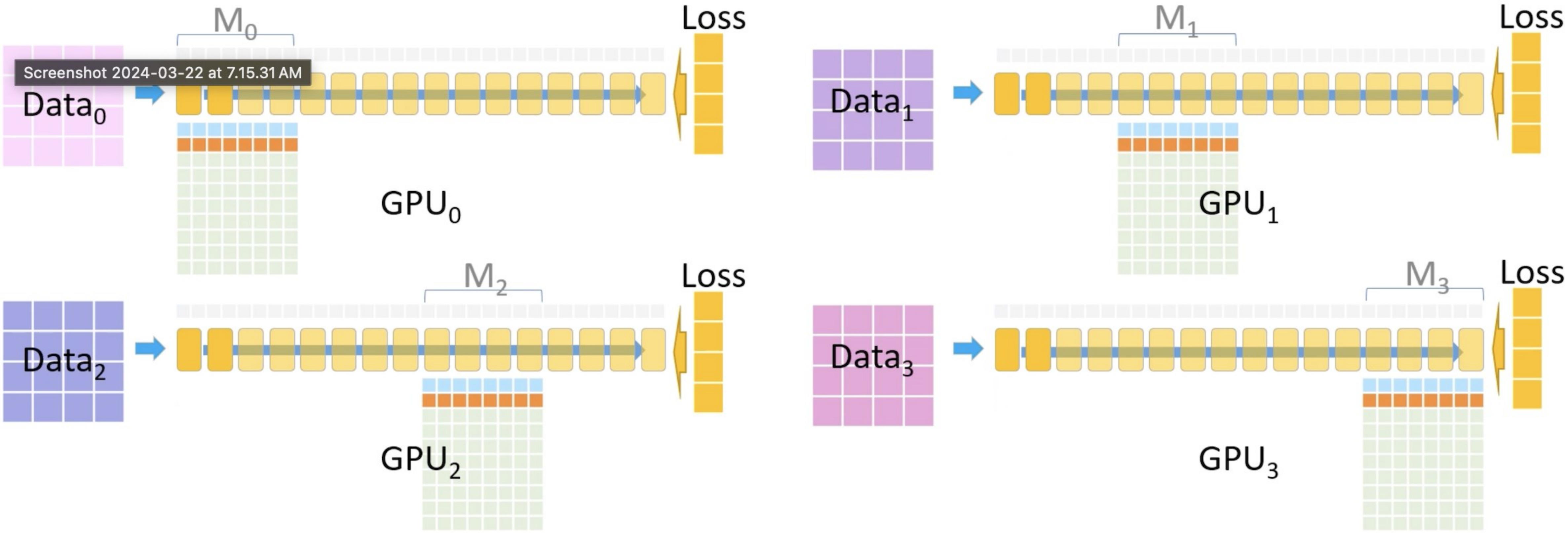
Adapted from a blog by MSR and slides from deeplearning.ai



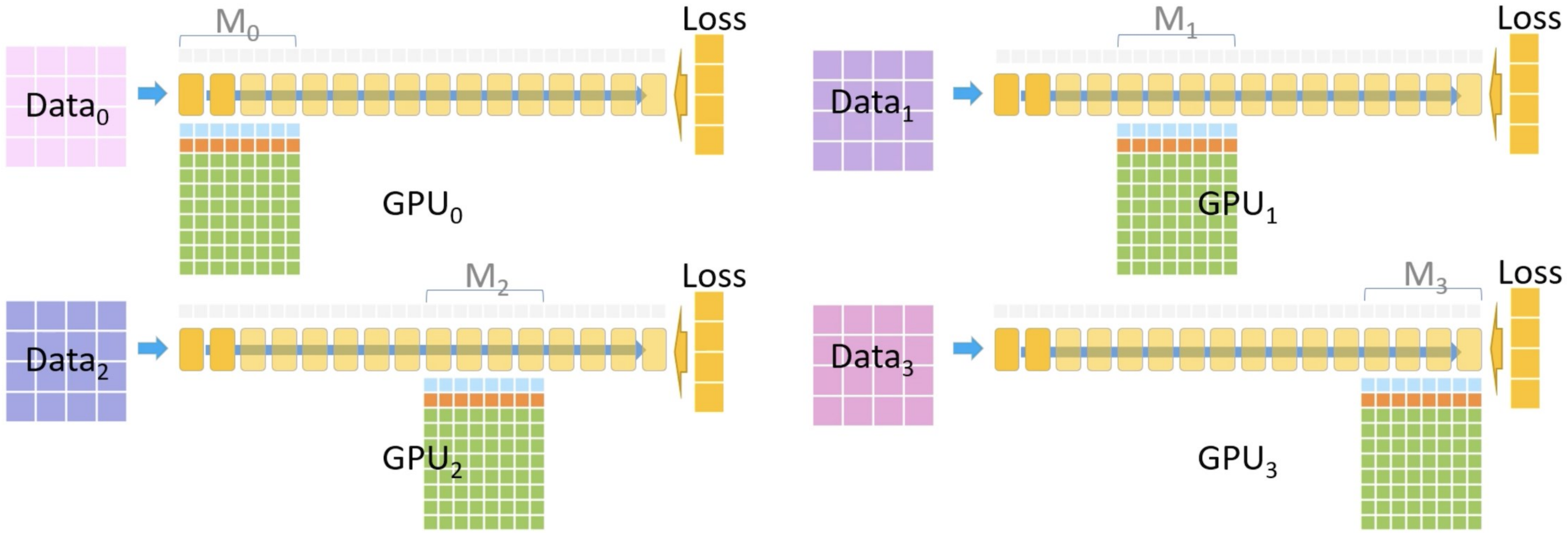
$GPU_{1,2,3}$ pass their M_0 gradients to GPU_0

GPU_0 performs gradient accumulation and holds final M_0 gradients for all Data

Adapted from a blog by MSR and slides from deeplearning.ai

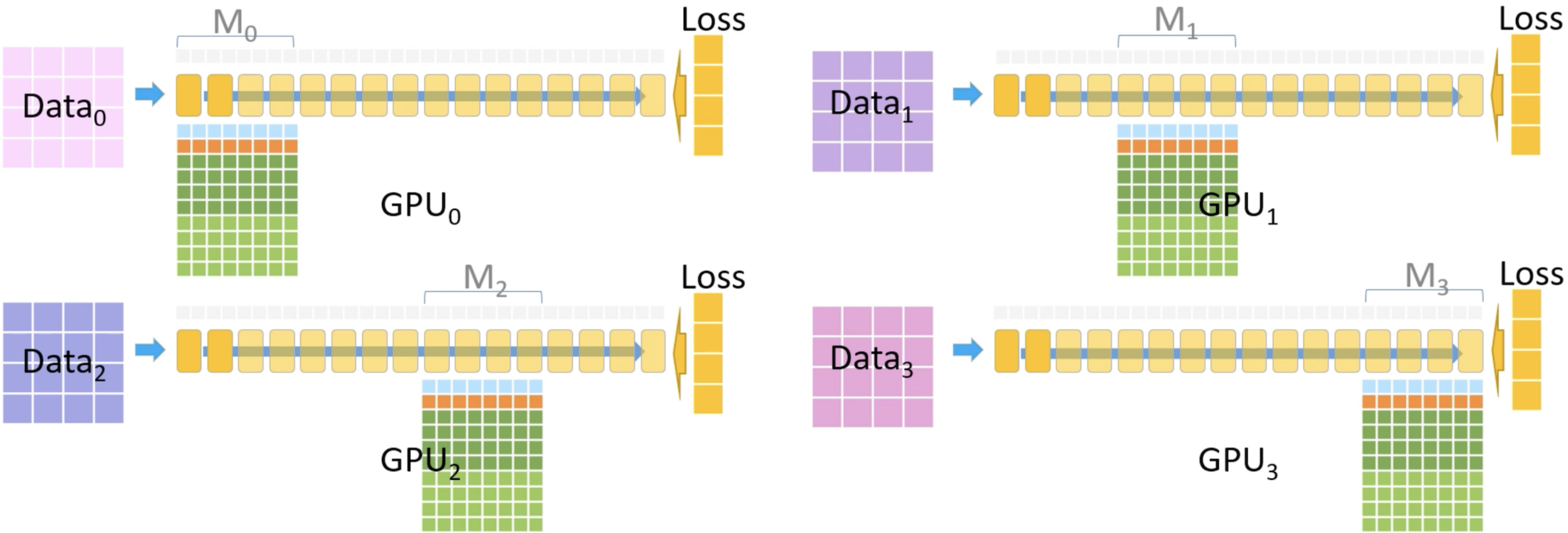


$GPU_{1,2,3}$ delete their temporary M_0 gradients and parameters
All M_0 activations are deleted

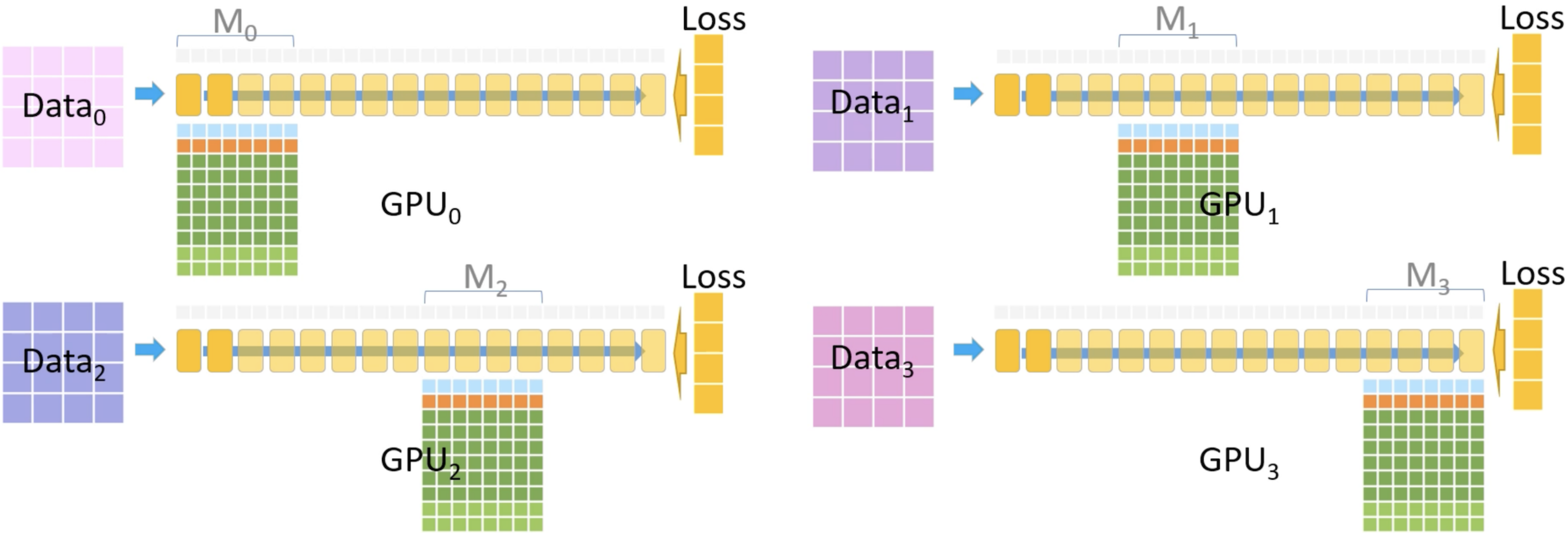


The Optimization step begins in parallel on each GPU

Adapted from a blog by MSR and slides from deeplearning.ai

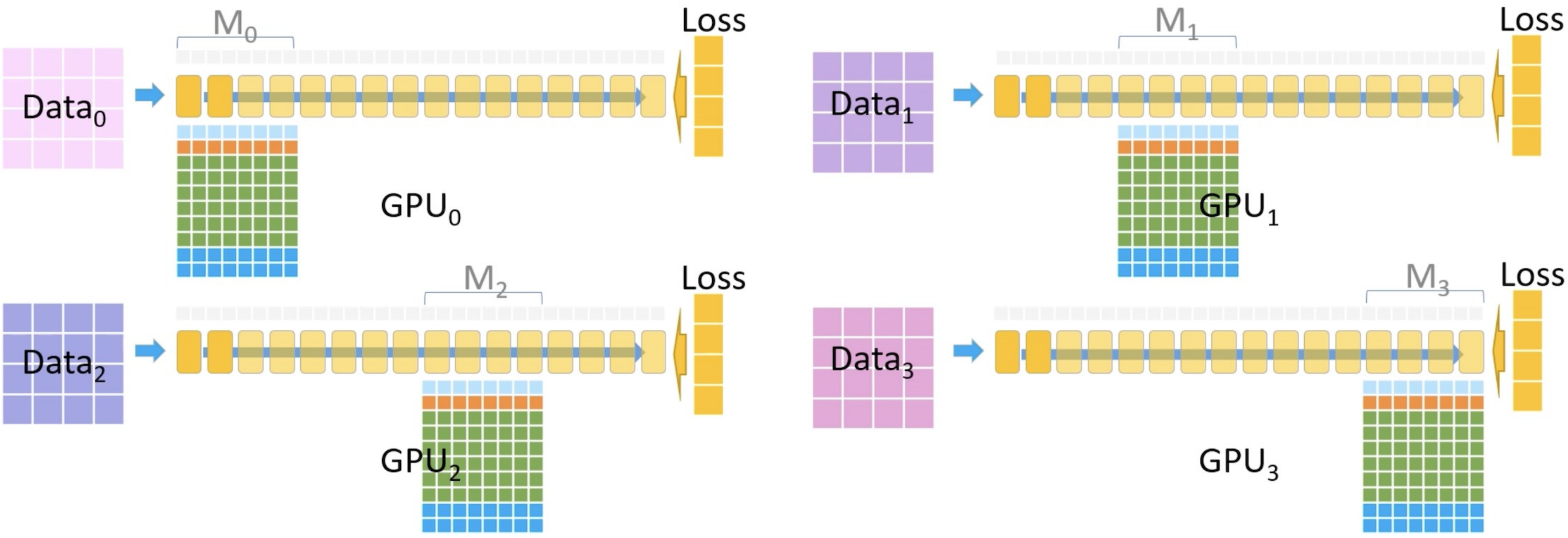


The optimizer runs



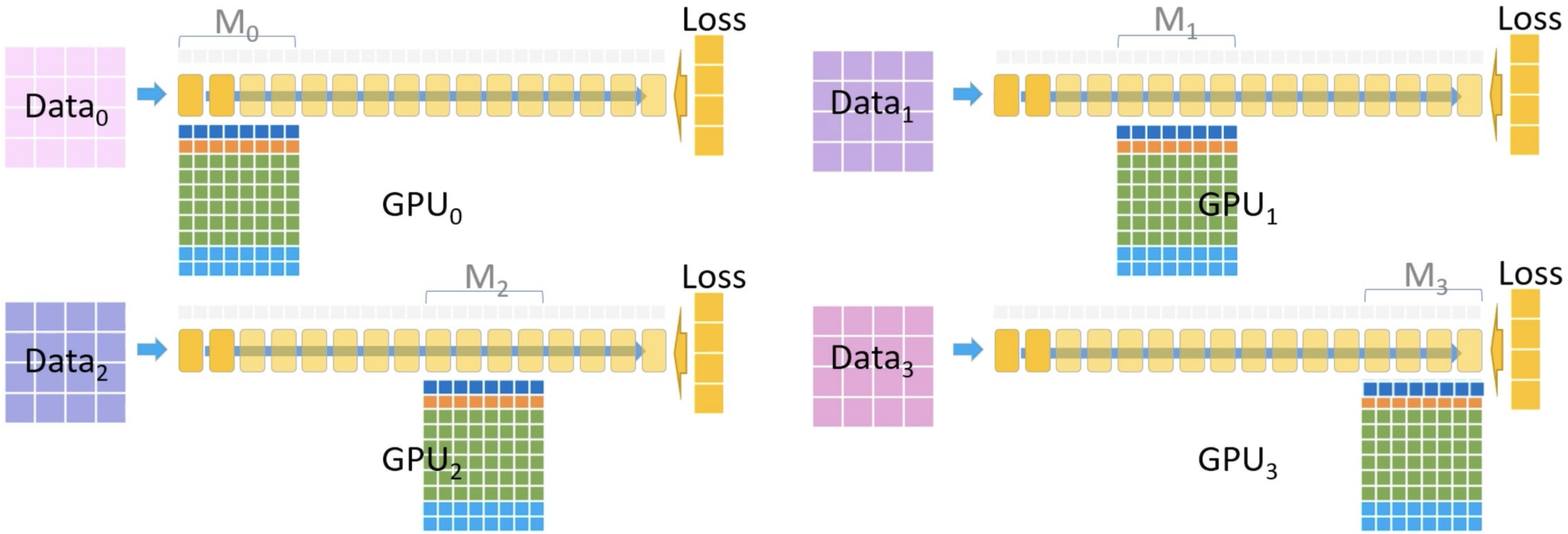
The optimizer runs

Adapted from a blog by MSR and slides from deeplearning.ai



The optimizer creates fp32 updated model weights
These are converted to fp16

Adapted from a blog by MSR and slides from deeplearning.ai

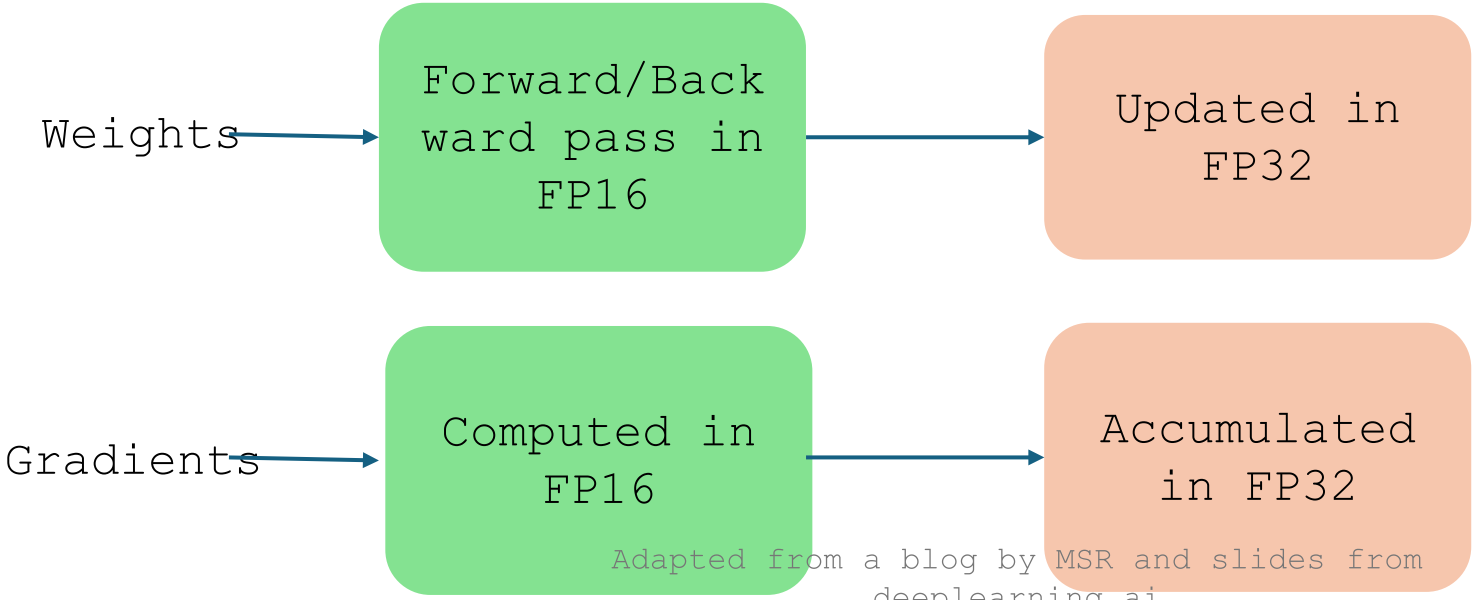


The fp16 weights become the model parameters for the next iteration
Training iteration complete!

Adapted from a blog by MSR and slides from deeplearning.ai

The need for two copies

- FP16 offers faster computation and reduced memory usage.
- Allows for faster forward and backward pass.
- FP32 weights/optimizer states are needed to ensure high precision.



Adapted from a blog by MSR and slides from deeplearning.ai

Summary

- Training at scale can be achieved by:
  - Quantization of parameters
  - Data Parallel
    - Distributed Data Parallel
    - FSDP/ Deepspeed ZeRO
  - Model Parallel – Not covered