Scaling LLM Training: Part I

Vedant Nanda
Researcher @ Aleph Alpha Research

ALEPH ALPHA

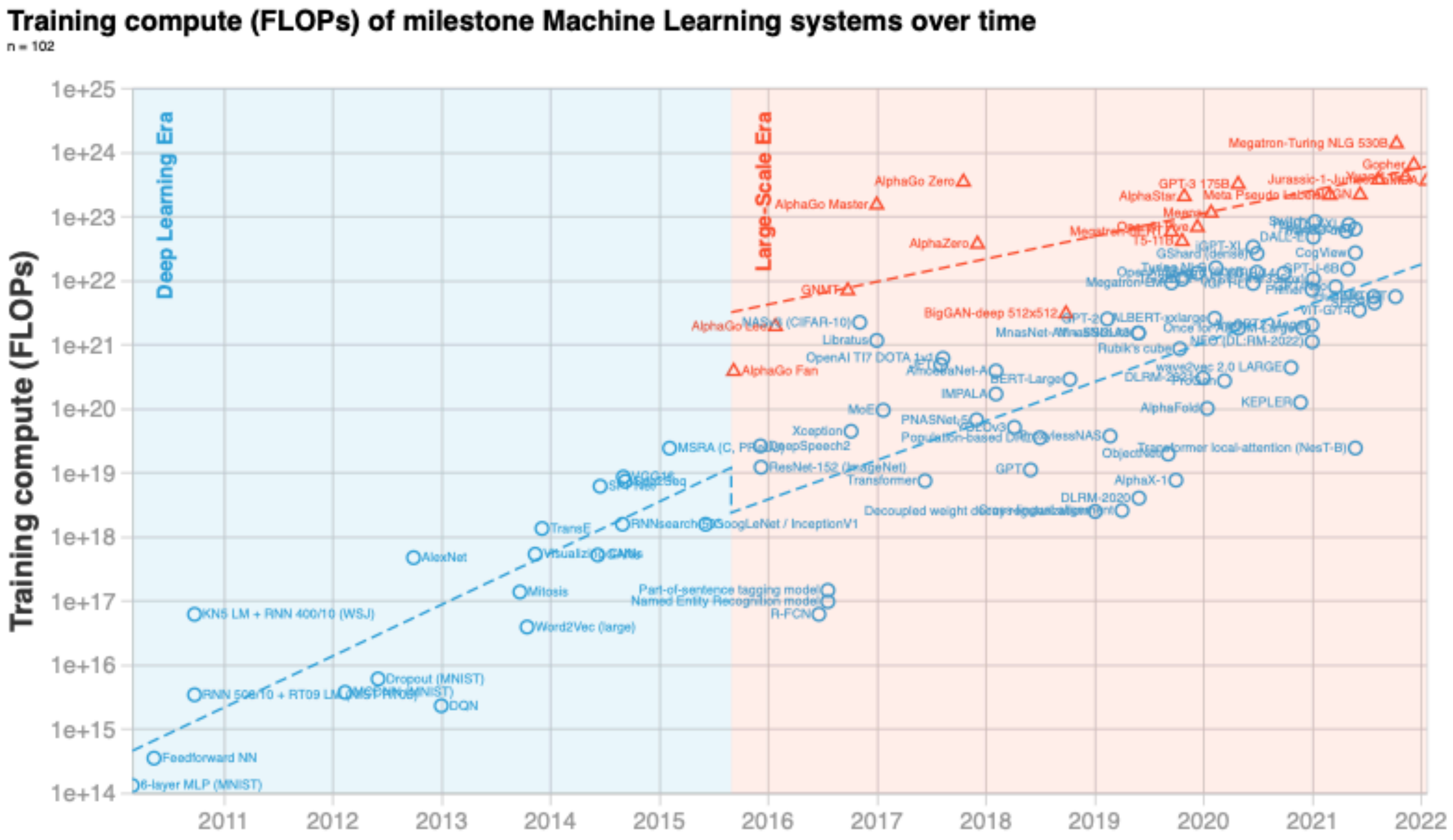
About Me

- PhD in CS from University of Maryland and MPI-SWS
- At Aleph Alpha for the last year: optimizing inference and pre-training
- We released base and instruct tokenizer-free models at ICLR last month
  - https://huggingface.co/collections/Aleph-Alpha
- Alt title for this lecture: stuff I wish I knew about pre-training LLMs

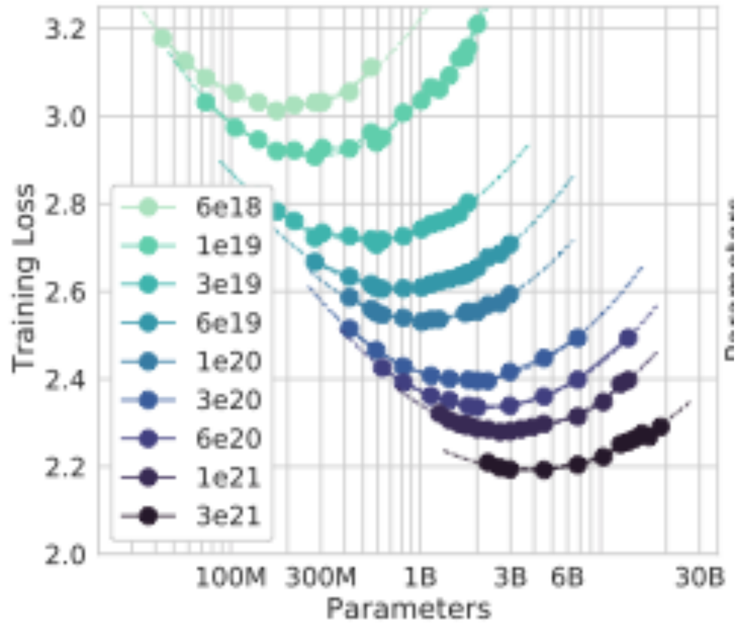
This Lecture

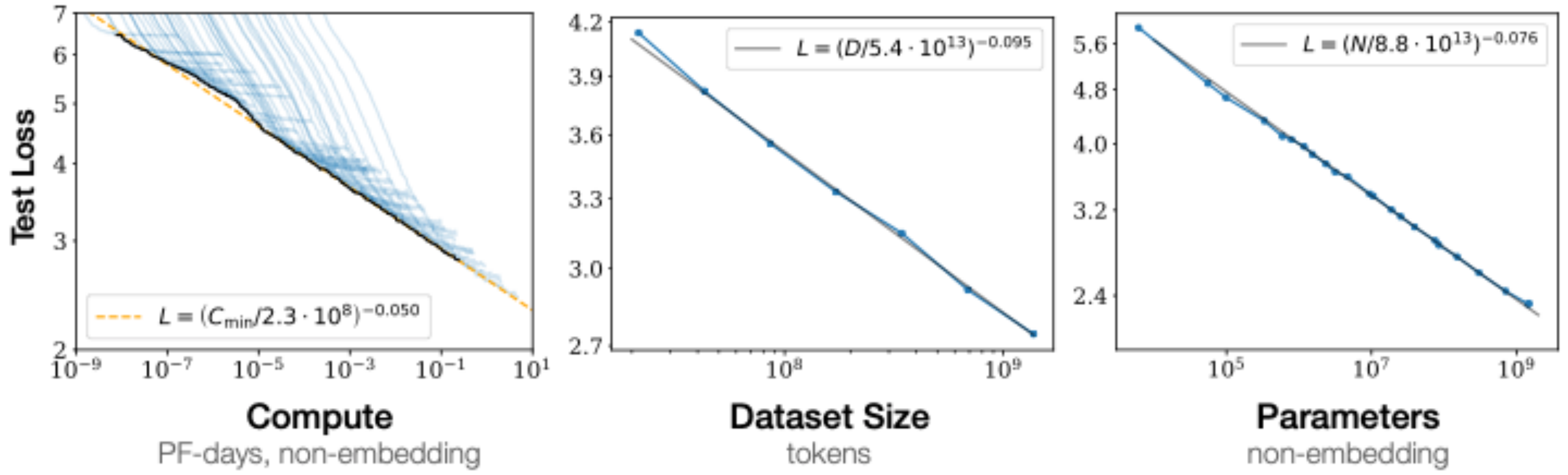
- Motivation (5 min)
- Deep dive into transformer compute (20 min)
- Understanding training workloads (20 min)
- Hands-on exercise (45 min)
- Scaling LLMs on one GPU: activation checkpointing (10 min)
- Scaling LLMs on one GPU: gradient accumulation (10 min)
- Scaling beyond one GPU: data parallelism (20 min)
- Hands-on exercise (45 min)

Scale: A Key Ingredient in SOTA LLMs



“Compute Trends Across Three Eras of Machine Learning” Sevilla et al. 2022



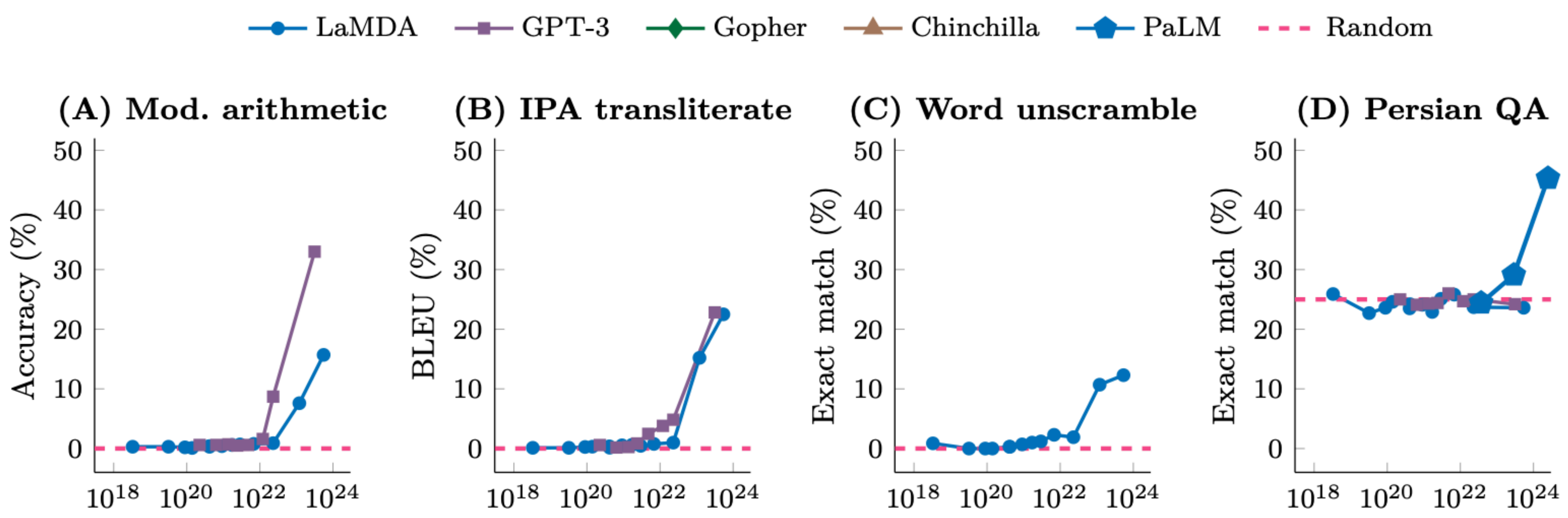


“Training Compute-Optimal Large Language Models” Hoffmann et al. 2022

“Scaling Laws for Neural Language Models” Kaplan et al. 2022

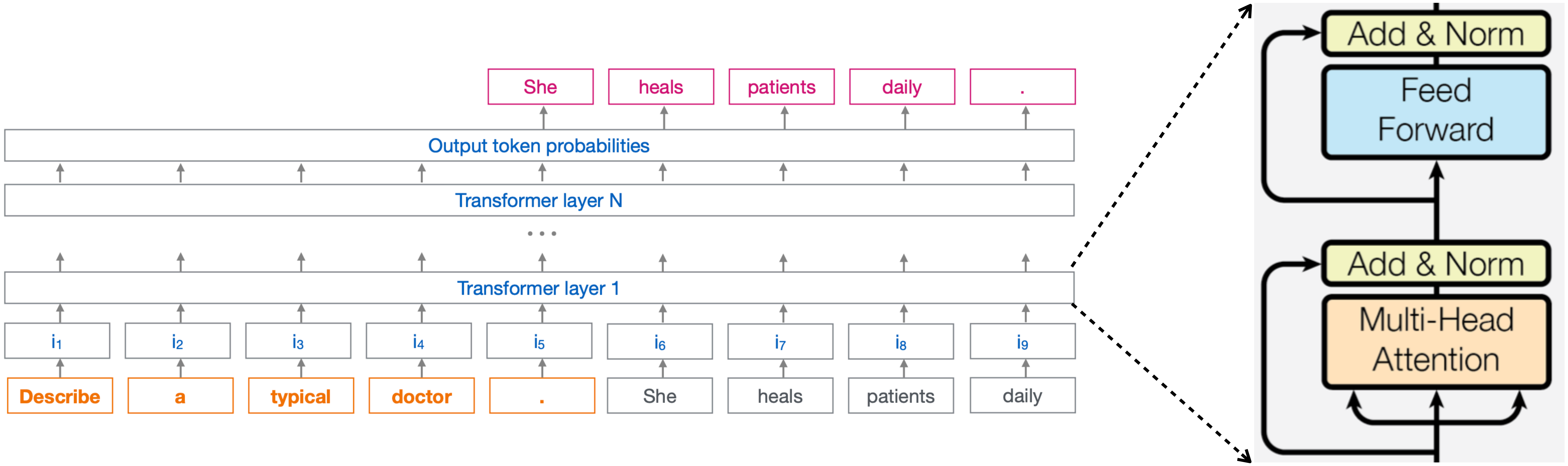
Why Should ML Researchers Care?

- Arguably the most pivotal Deep Learning paper (AlexNet circa 2012) was a systems paper that trained a CNN with model parallelism
- Many architectures died because they didn't scale (eg: LSTM)
  - *The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.* — Bitter Lesson by Rich Sutton
- Emergence at scale; many innovations today are at the edge of hardware

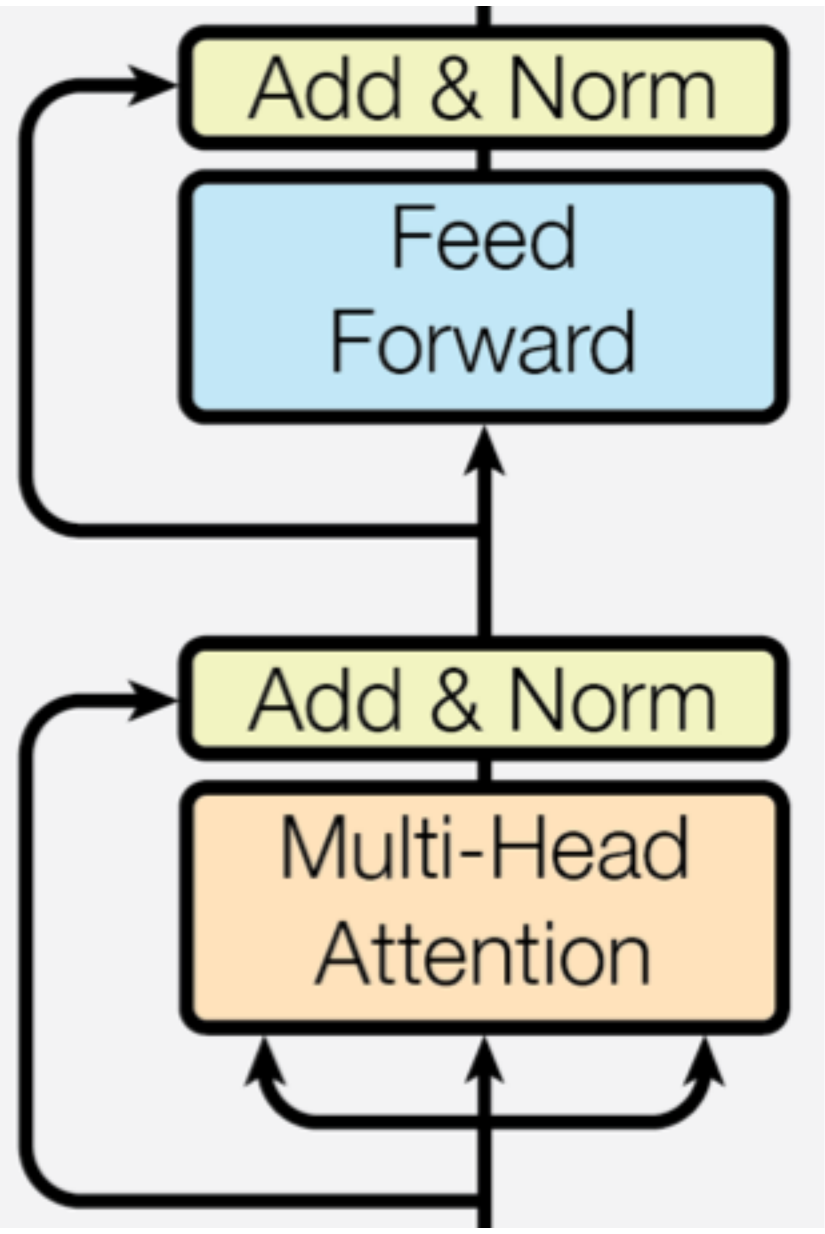


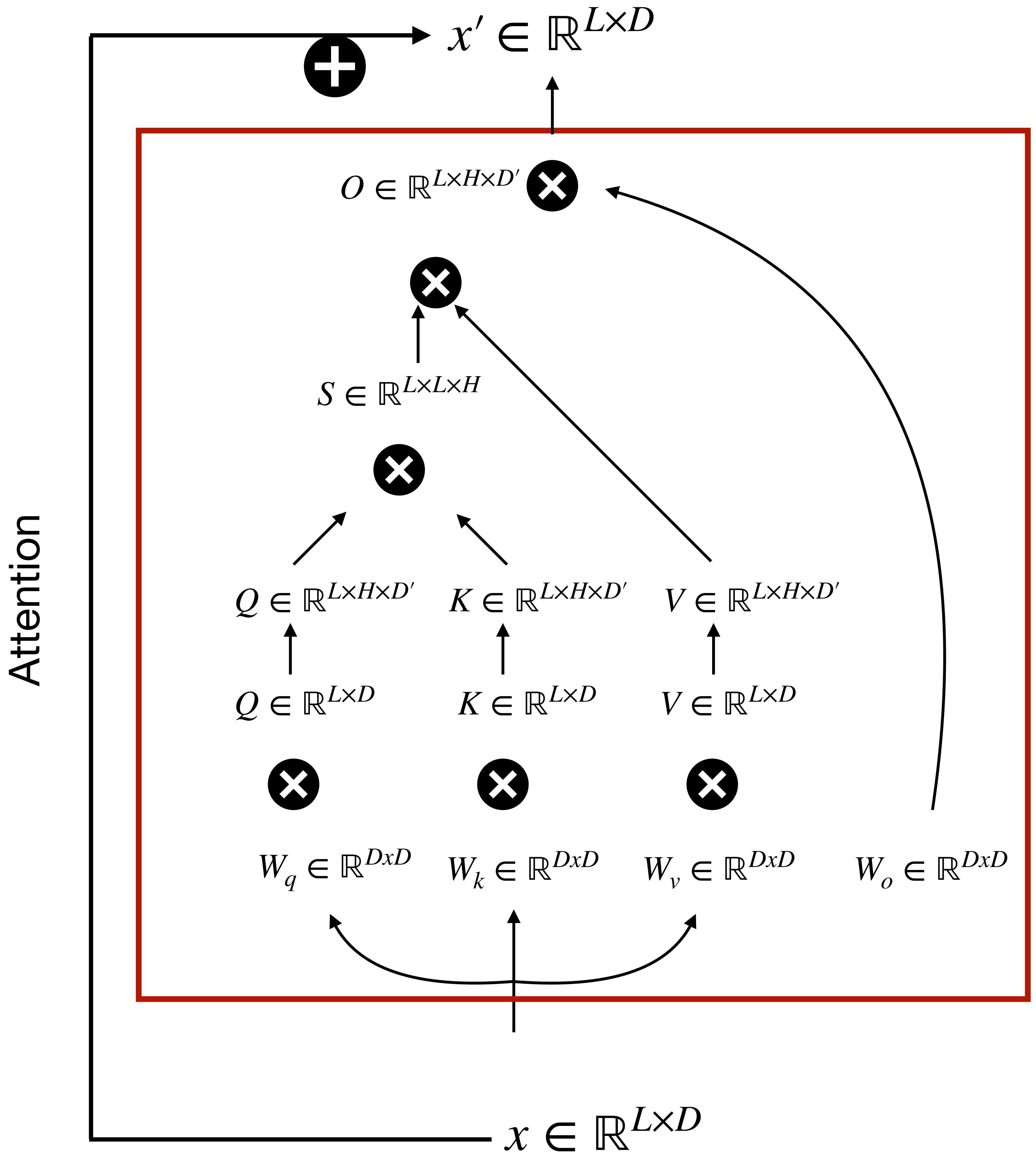
Emergent Abilities of Large Language Models, 2022

Transformers Recap

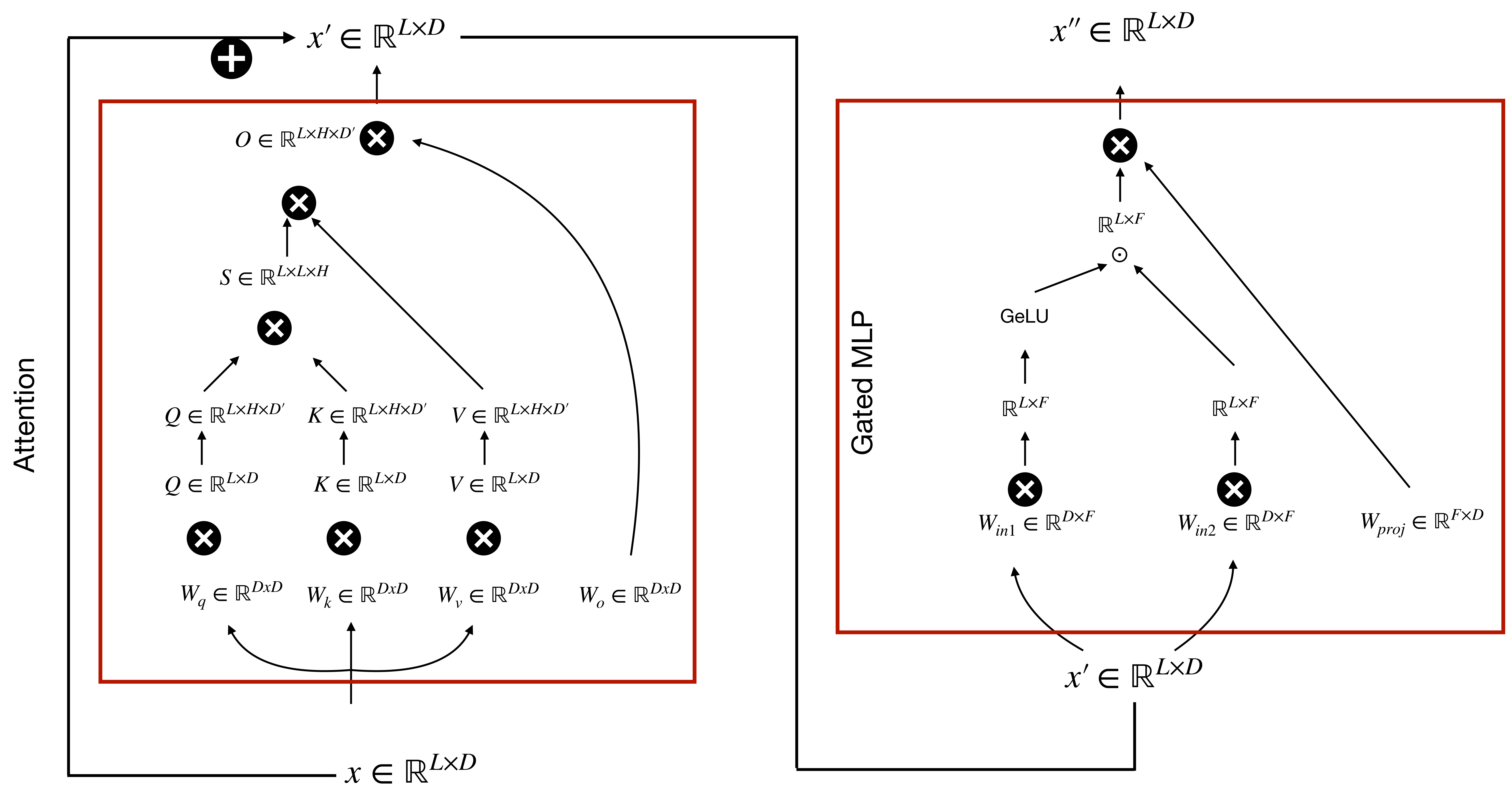


Transformers: What Do We Compute?





Transformers: What Do We Compute?



Transformers: What Do We Compute?

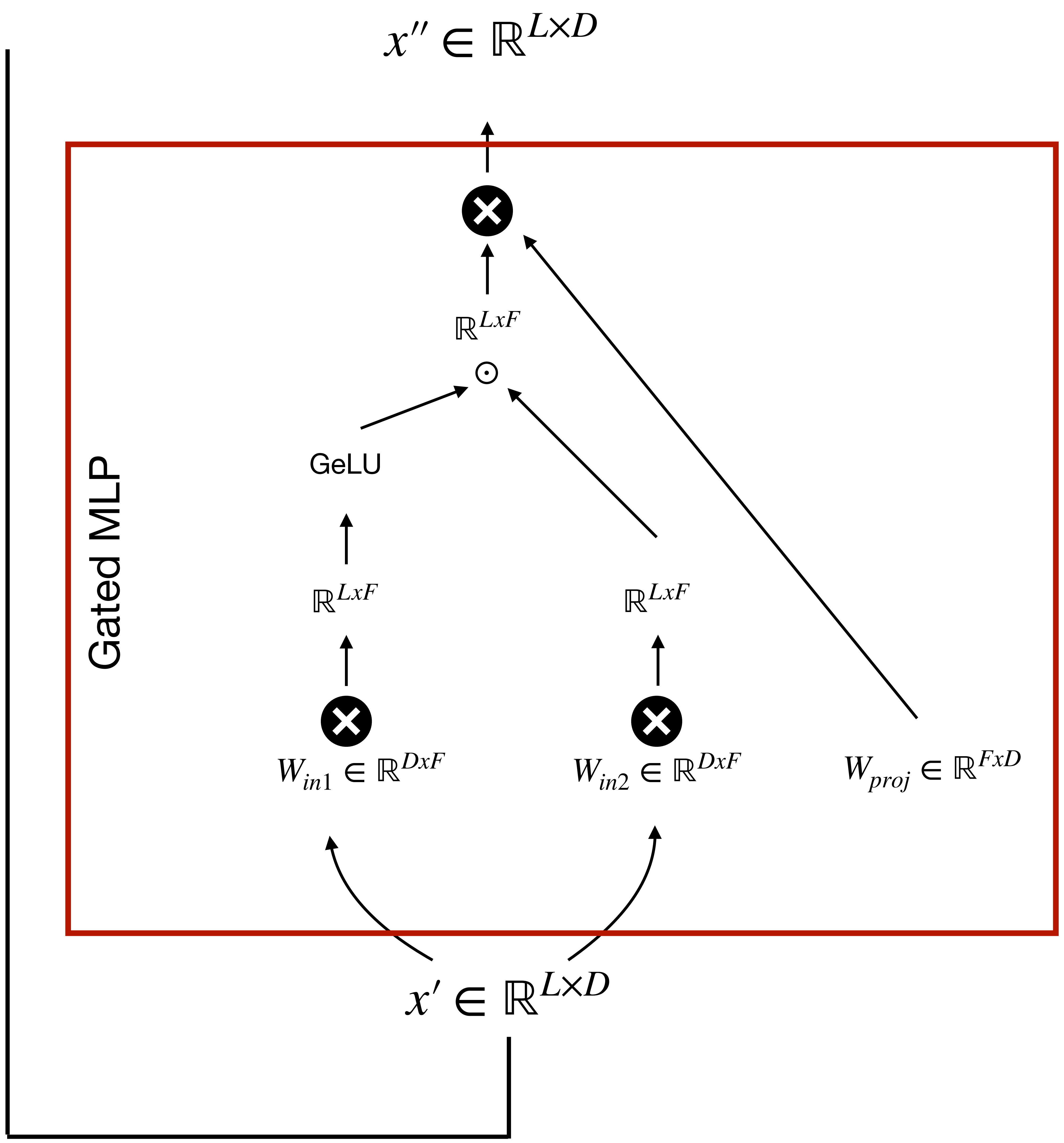
Why gating?

Standard choice in all latest architectures:
Llama, DeepSeek, Qwen, Mistral

4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

GLU Variants Improve Transformer. Noam Shazeer 2020



Quick Primer on Compute

It's matmuls all the way down

$$Y = A\,.\,B \qquad \underset{\text{Activations}}{A} \in \mathbb{R}^{M \times D}, \underset{\text{Weights}}{B} \in \mathbb{R}^{D \times N}$$

We count compute as FLOPs, ie, **FL**oating Point **OP**erations

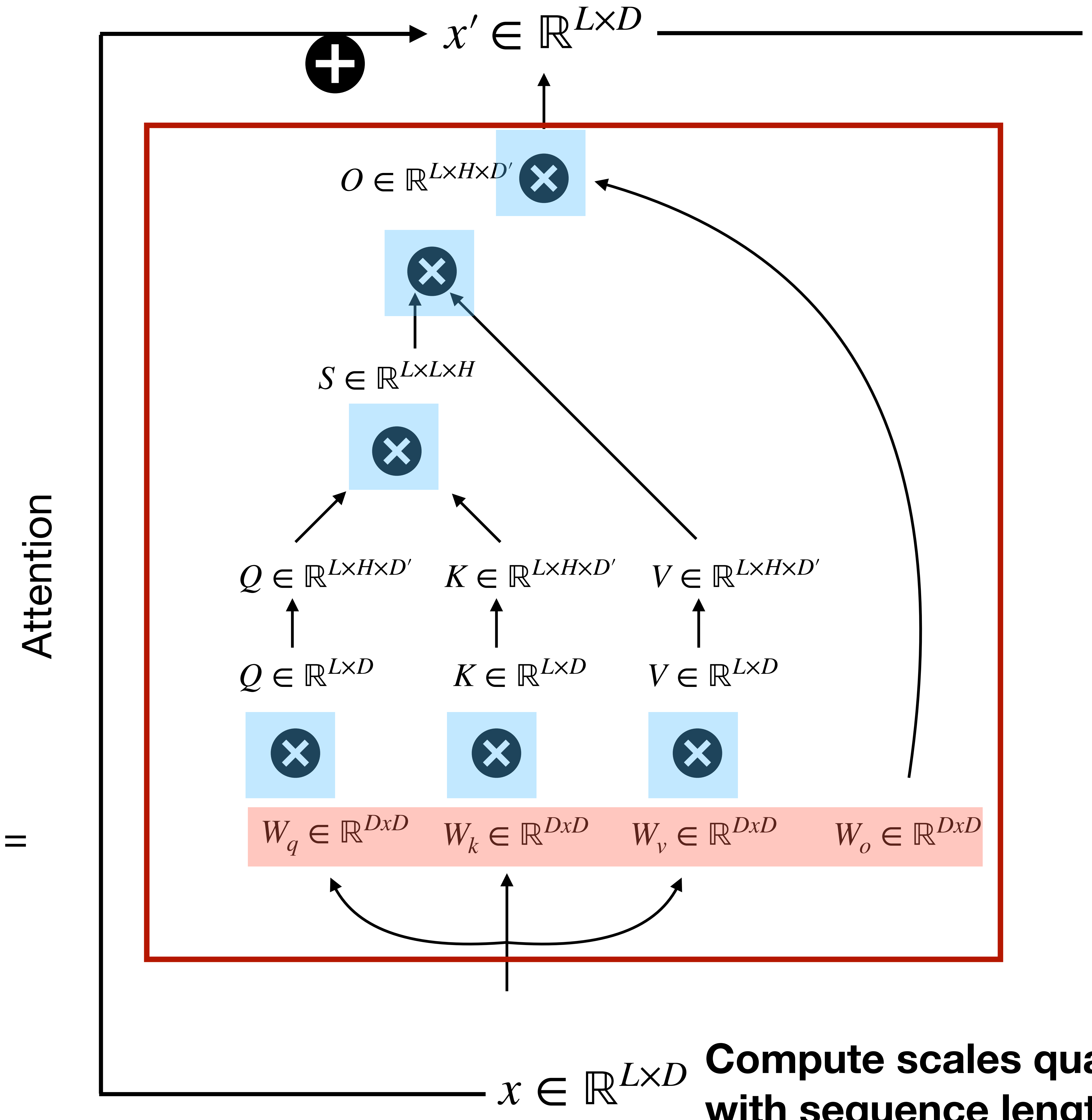
Forward FLOPs: D multiplications and D additions per entry of Y = 2.D.M.N

Backward FLOPs 4.D.M.N **2x forward pass FLOPs**

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial B} = A^T \cdot \frac{\partial L}{\partial Y} \qquad \frac{\partial L}{\partial A} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial A} = \frac{\partial L}{\partial Y} \cdot B^T$$

2.D.M.N 2.D.M.N

Transformers: What Do We Compute?



Total params = 4.D.D

	FLOPs (inference)	FLOPs (training)	
$x \,.\, W_q$	2.L.D.D	6.L.D.D	
$x \,.\, W_k$	2.L.D.D	6.L.D.D	
$x \,.\, W_v$	2.L.D.D	6.L.D.D	
$Q \,.\, K^T$	2.D'.L.L.H = 2.D.L.L	6.D.L.L	
$softmax$	L (sum every row). L (scale every entry in row). H (do this for every head) = L.L.H	O(L.L.H)	H << D
$S \,.\, V$	2.L.L.D'.H = 2.D.L.L	6.D.L.L	
$W_o \,.\, O$	2.L.D.D	6.L.D.D	**3x more compute in training for same L**
Compute scales quadratically with sequence length	8.L.D.D + 4.D.L.L	24.L.D.D + 12.D.L.L	

Transformers: What Do We Compute?

	FLOPs (inference)	**FLOPs (training)**
$x'.W_{in1}$	2.L.D.F	6.L.D.F
$x'.W_{in2}$	2.L.D.F	6.L.D.F
$x'.W_{proj}$	2.L.D.F	6.L.D.F
	6.L.D.F	18.L.D.F

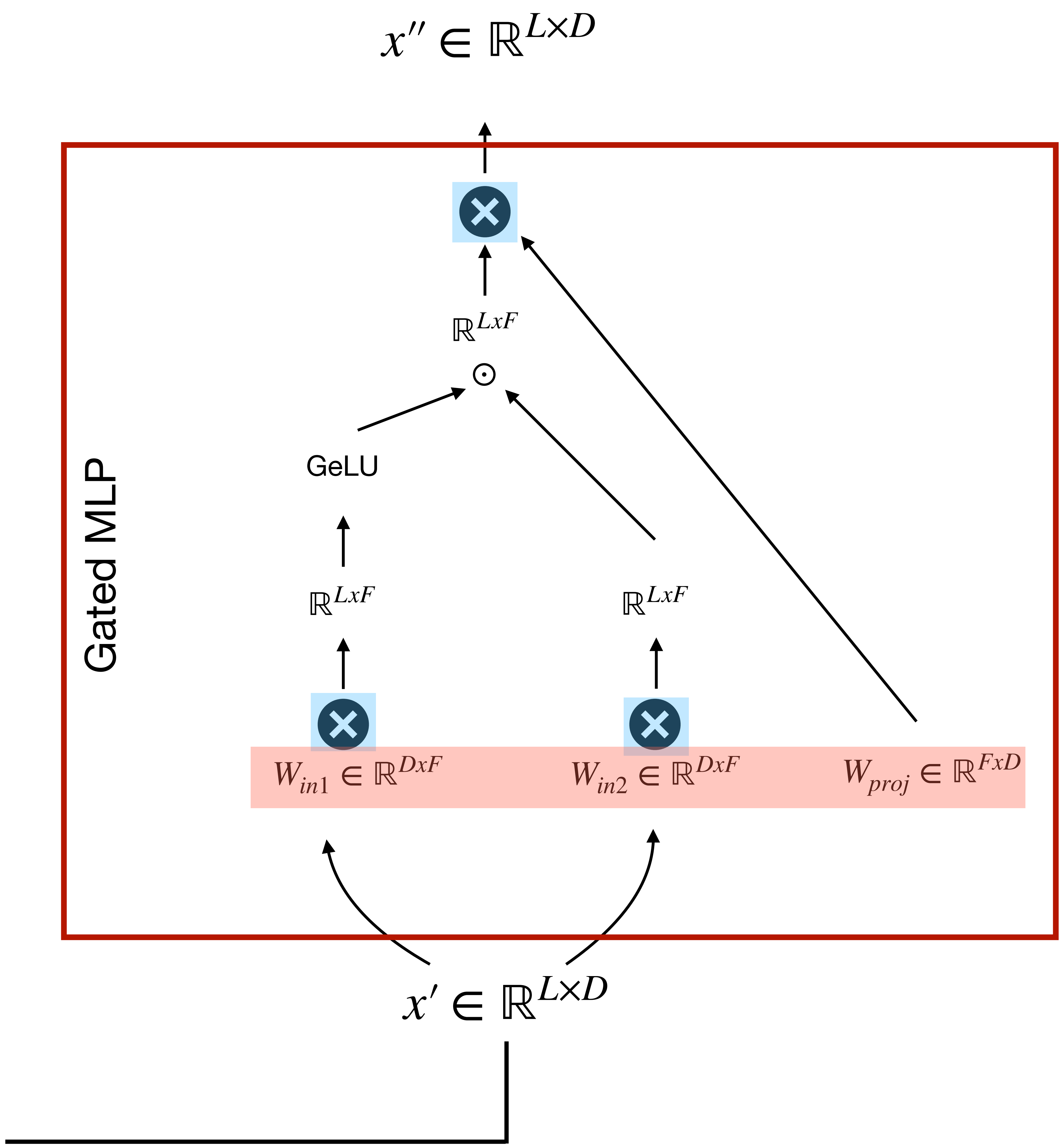
Total Params = 3.D.F

Typically, F ≈ 3.5D (Llama3)

Total Params ≈ 10.D.D

vs params in attention = 4.D.D

Most params in MLP layers!



Optional Exercise

Calculate FLOPs for attention variants such as
GQA (Llama) and MLA (DeepSeek)

Inference vs Training Compute

For each transformer layer

Inference

Attention

8.L.D.D + 4.D.L.L

MLP

6.L.D.F

8.L.D.D + 4.D.L.L + 21.L.D.D

In the worst case, L = O (1),
ie one user, one small prompt

Total compute = 29D.D + 4D

Plugging F = 3.5D

For most L (<4D)
MLPs have majority of
the FLOPs

Training

Attention

24.L.D.D + 12.D.L.L

MLP

18.L.D.F

24.L.D.D + 12.D.L.L + 63.L.D.D

Typical batch sizes = O(1e+6) (eg Llama3)

Total compute = 87DD.O(1e+6) + 12D.O(1e+12)

Extended Reading:
Roofline Model

On most GPUs, Inference is memory bound!

Way more compute in training!

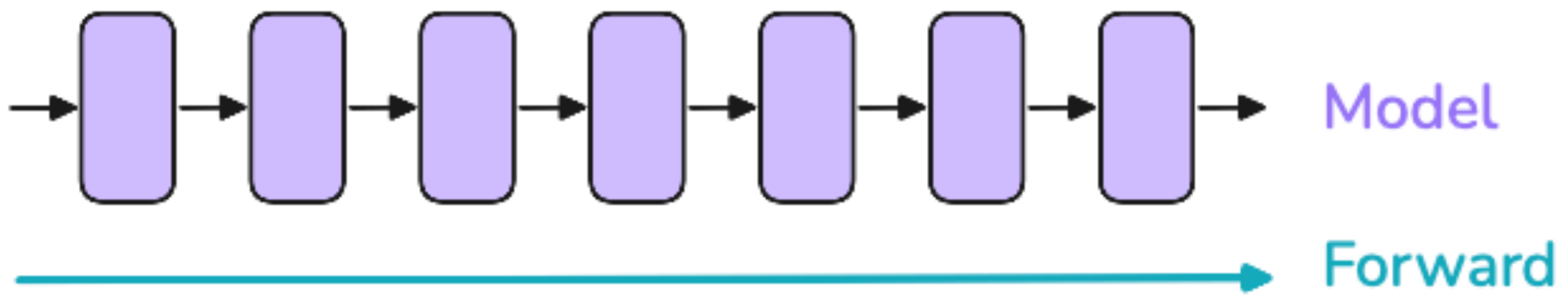
Profiling Inference and Training

Hands-on exercise: Use PyTorch profiler to analyze inference vs training workloads

LLM Training

Key difference from inference: We hold activations in memory for the backward pass

Params typically in bfloat16



Memory Progression

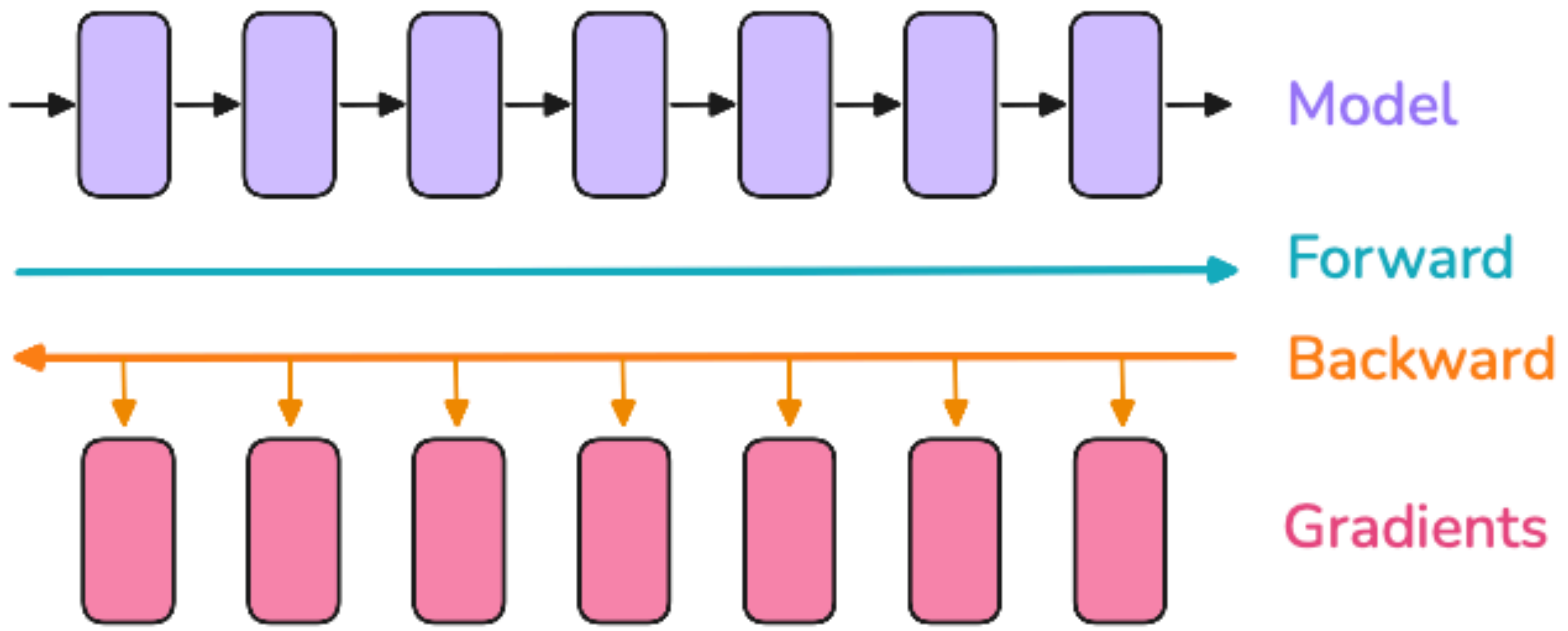
2.P bytes

Ultrascale Playbook. HuggingFace 2025

LLM Training

Key difference from inference: We hold activations in memory for the backward pass

Params typically in bfloat16



Grads same precision as params*

Memory Progression

2.P bytes

2.P bytes

Ultrascale Playbook. HuggingFace 2025

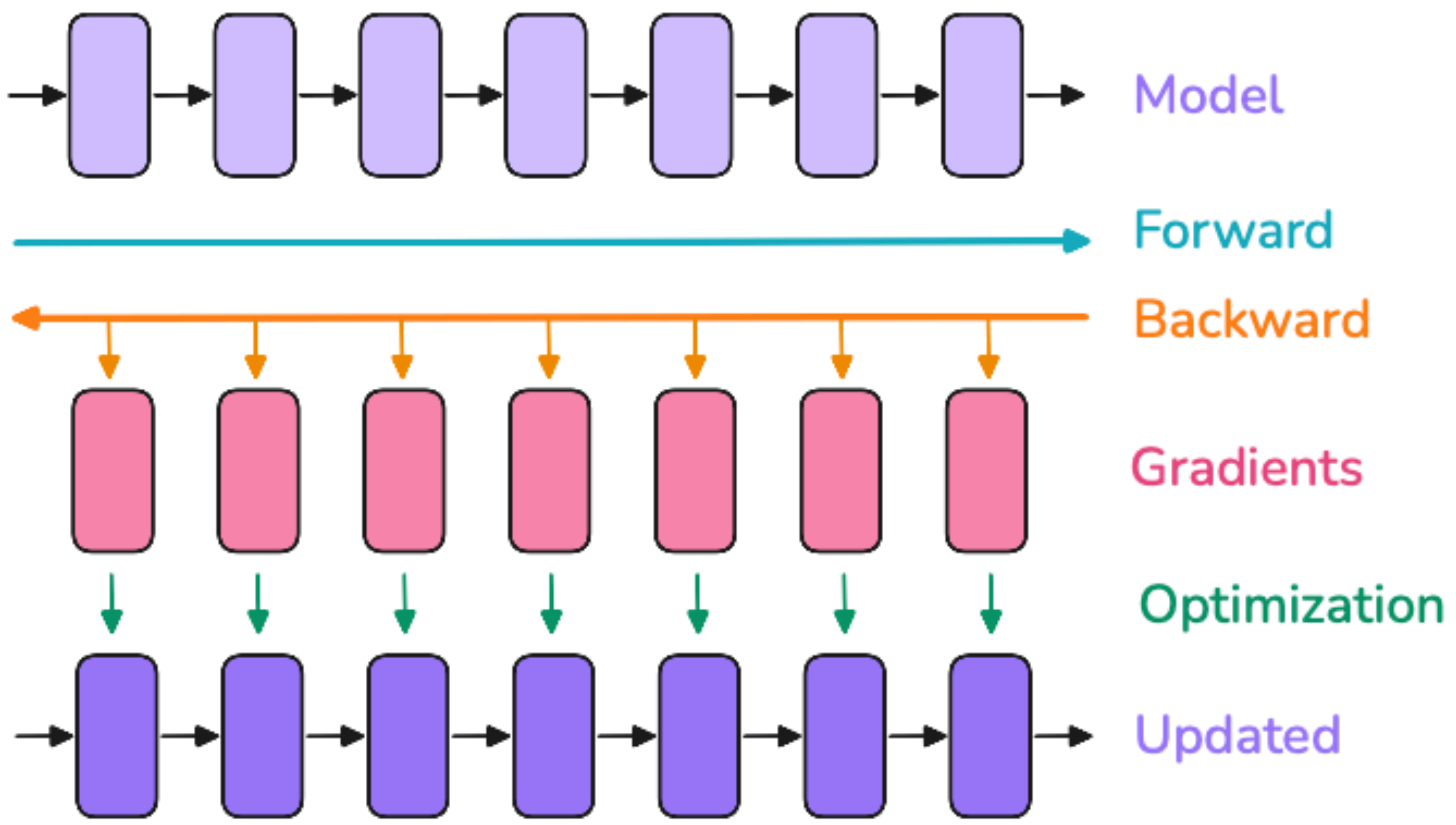
LLM Training

Key difference from inference: We hold activations in memory for the backward pass

Params typically in bfloat16

Grads same precision as params*

Optimizer (AdamW) in fp32, with fp32 copy of weights**



Ultrascale Playbook. HuggingFace 2025

Memory Progression

2.P bytes

2.P bytes

4.P bytes for fp32 weights

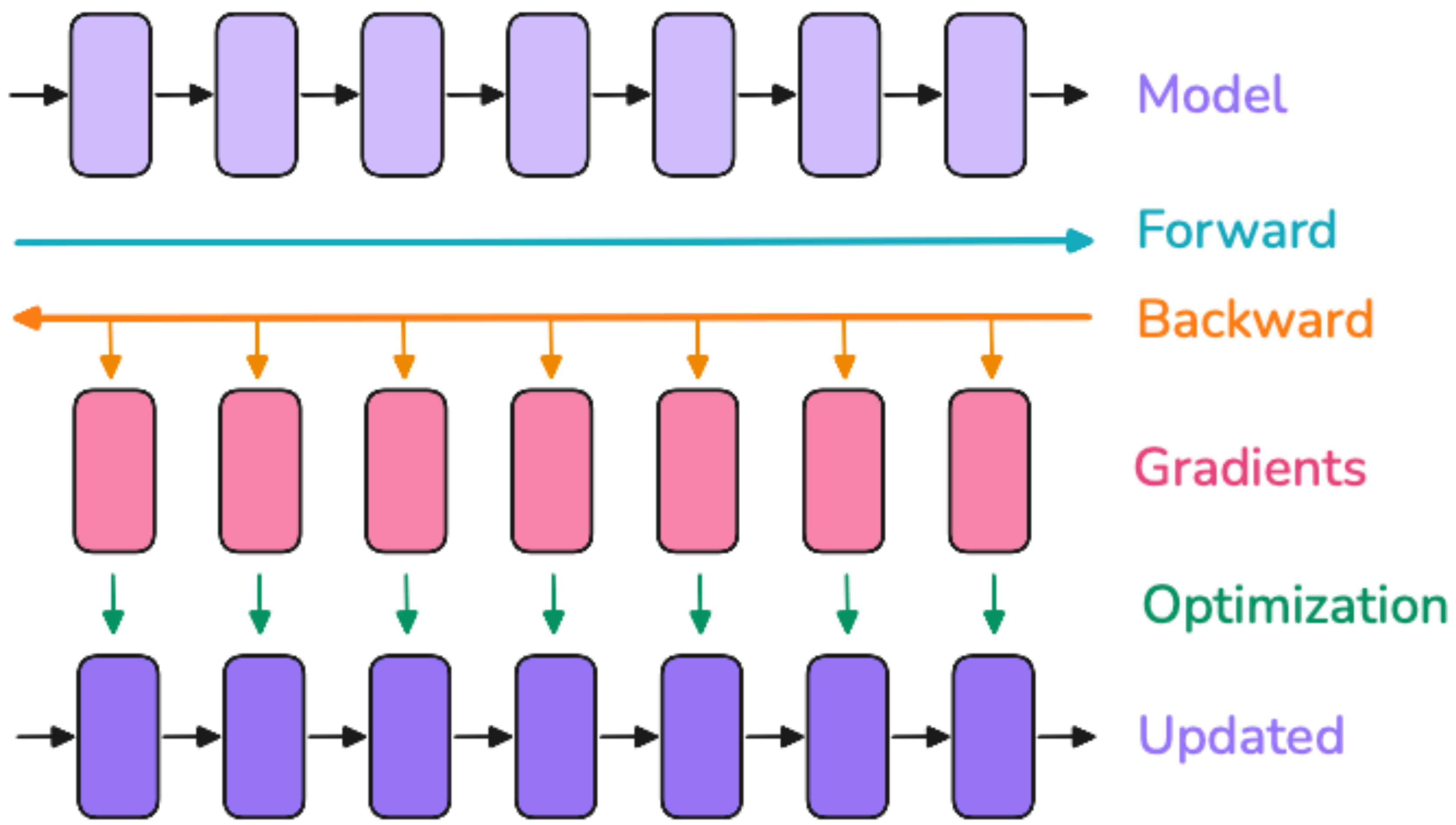
4.P bytes for momentum

4.P bytes for variance

* Later we will see that sometimes, for stability it makes sense to store gradients in fp32

** As a rule of thumb, operations that “accumulate” should be done in fp32 to avoid underflow in bf16

LLM Training: Memory



	Mixed precision	Mixed precision w fp32 grads	Full precision
Params	2.P	2.P	4.P
Grads	2.P	4.P	4.P
Optimizer	12.P	12.P	8.P
	16.P	18.P	16.P
Llama 8B	**128GB**	**144GB**	**128GB**

Q1: Why use mixed precision when it requires same memory as full?

Technical Specifications		
	H100 SXM	**H100 NVL**
FP64	34 teraFLOPS	30 teraFLOPS
FP64 Tensor Core	67 teraFLOPS	60 teraFLOPS
FP32	67 teraFLOPS	60 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS	835 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS	1,671 teraFLOPS

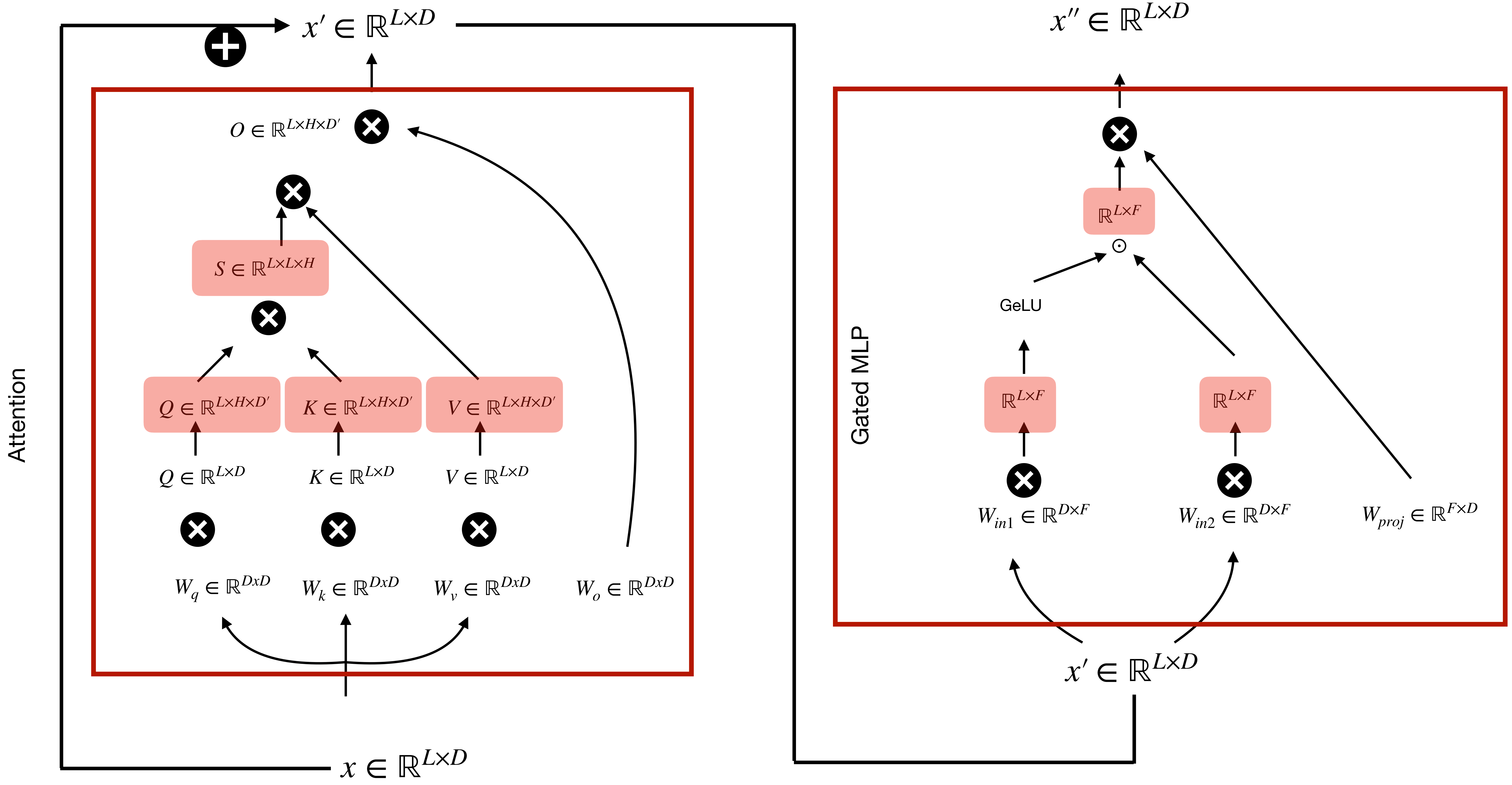
Much faster!

Q2: Are we missing something?

Activations!

Recall: We hold activations in memory for the backward pass

How Much Do Activations Need?



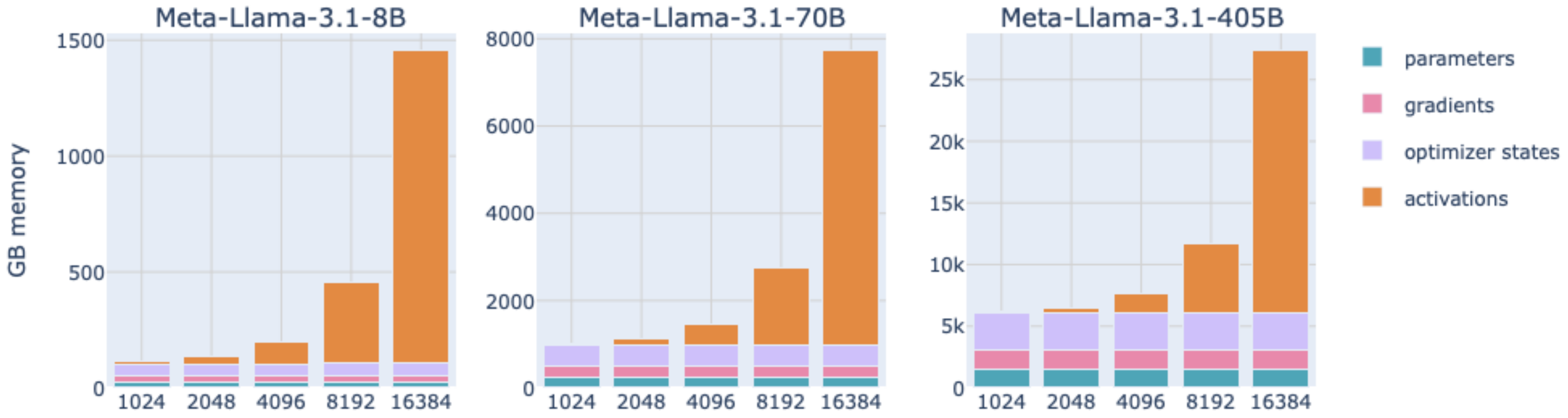
Major contributor: materializing the attention matrix

For 1 sequence of length L, act mem = O(L^2)

For `bs` sequences of length L, act mem = O(bs.L^2)

How Does Memory Scale?

Keeping `bs` = `1`, for different Llama model sizes



For smaller sequences, this is negligible, but it very quickly dwarfs the memory footprint of params, grads, optimizer combined!

Recap

- Calculating FLOPs in transformer layers for inference vs training
- Understanding compute and memory in LLM training workloads

Fitting Things on One GPU

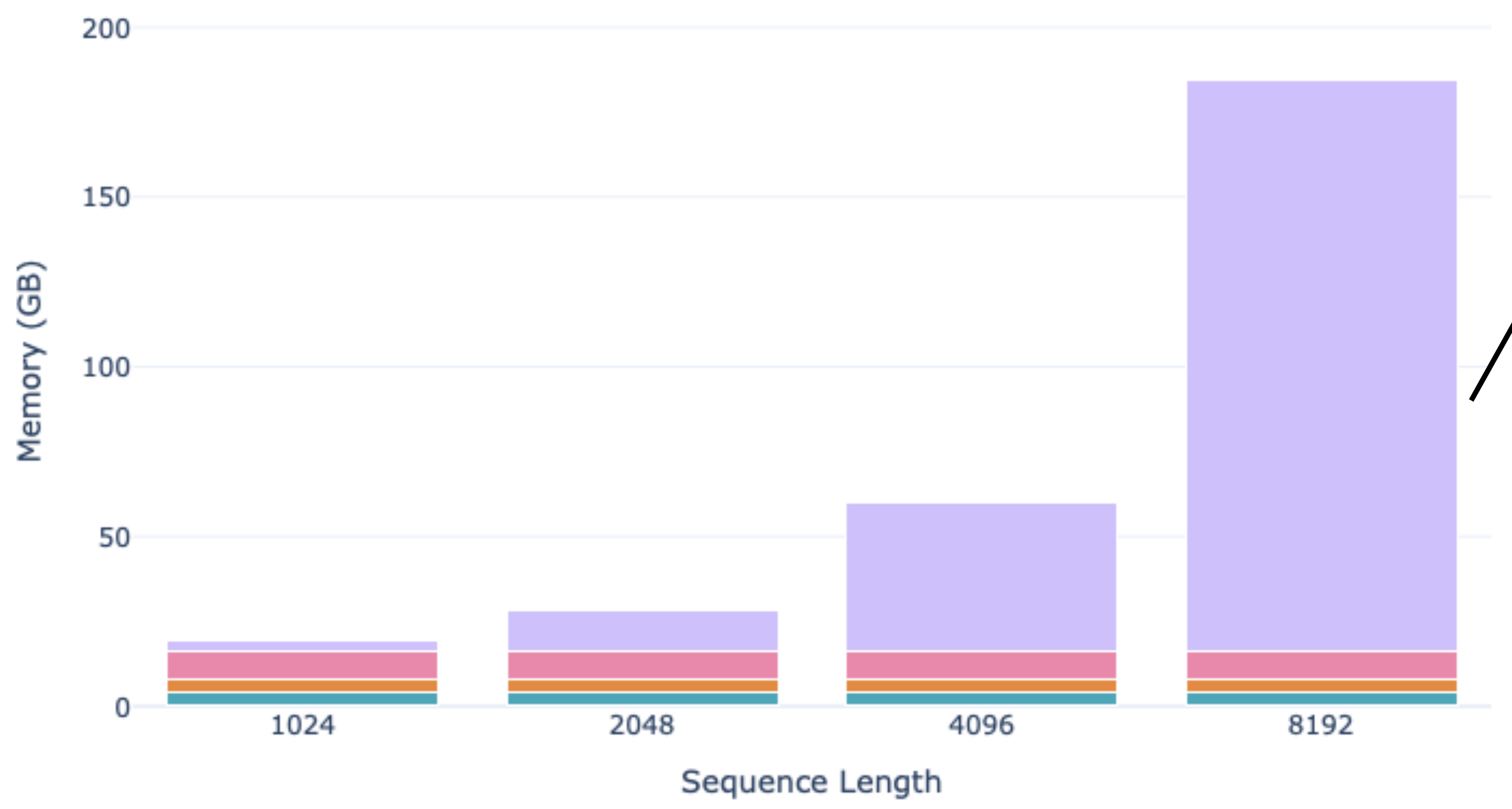
	Mixed precision
Params	2.P
Grads	2.P
Optimizer	12.P
	16.P
Llama 8B	**128GB** ✗

You cannot train a model where params + grads + optimizer don't fit in memory

=> Reduce parameter count

Llama 1B **16GB** ✓

However…



Activation memory explodes at standard training context lengths

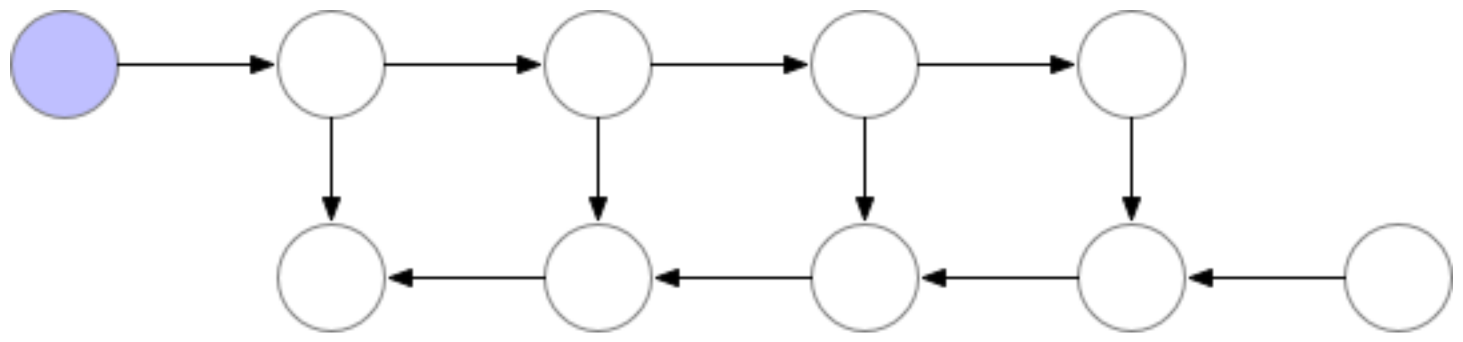
Activation Checkpointing

Key concept: Trade off memory for compute

Other names: gradient checkpointing, activation recomputation, rematerialization

Drop activations from memory and re-compute when needed

Default

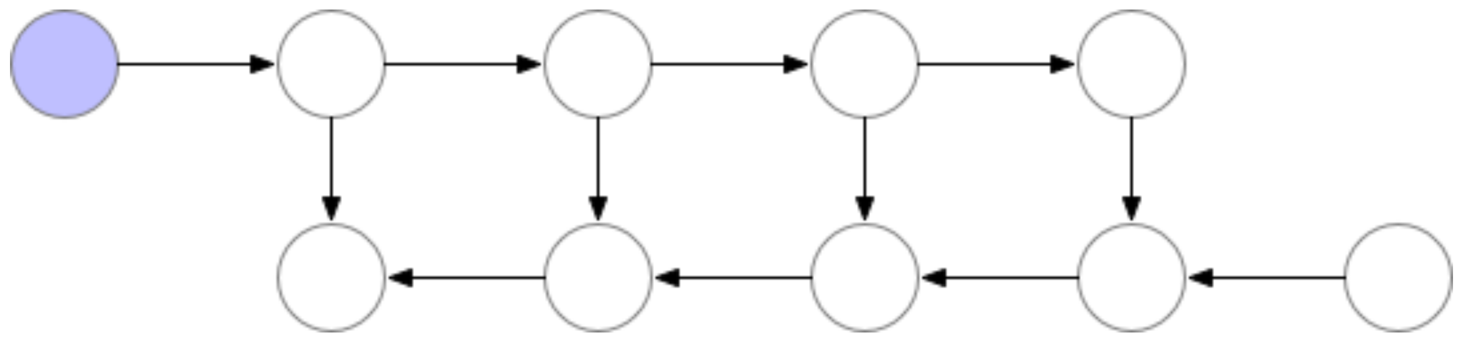


Memory: O(num_layers)

Compute: O(num_layers)

Forward per layer: 1

Keep every √num_layer in memory



Memory: O(√num_layers)

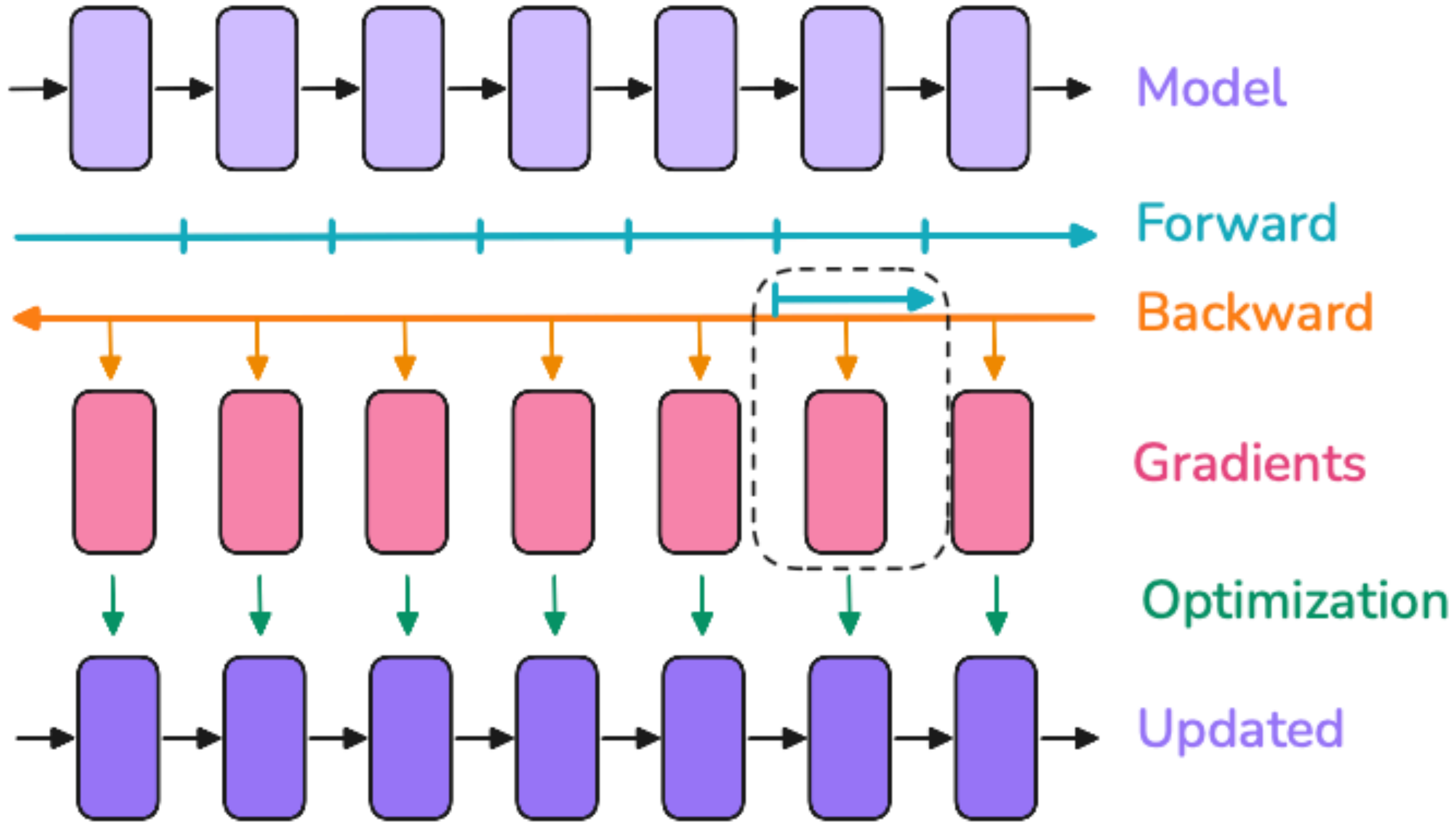
Compute: O(num_layers)

Forward per layer: 1 to 2

Credits: https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9

Activation Checkpointing

Easy to implement in PyTorch



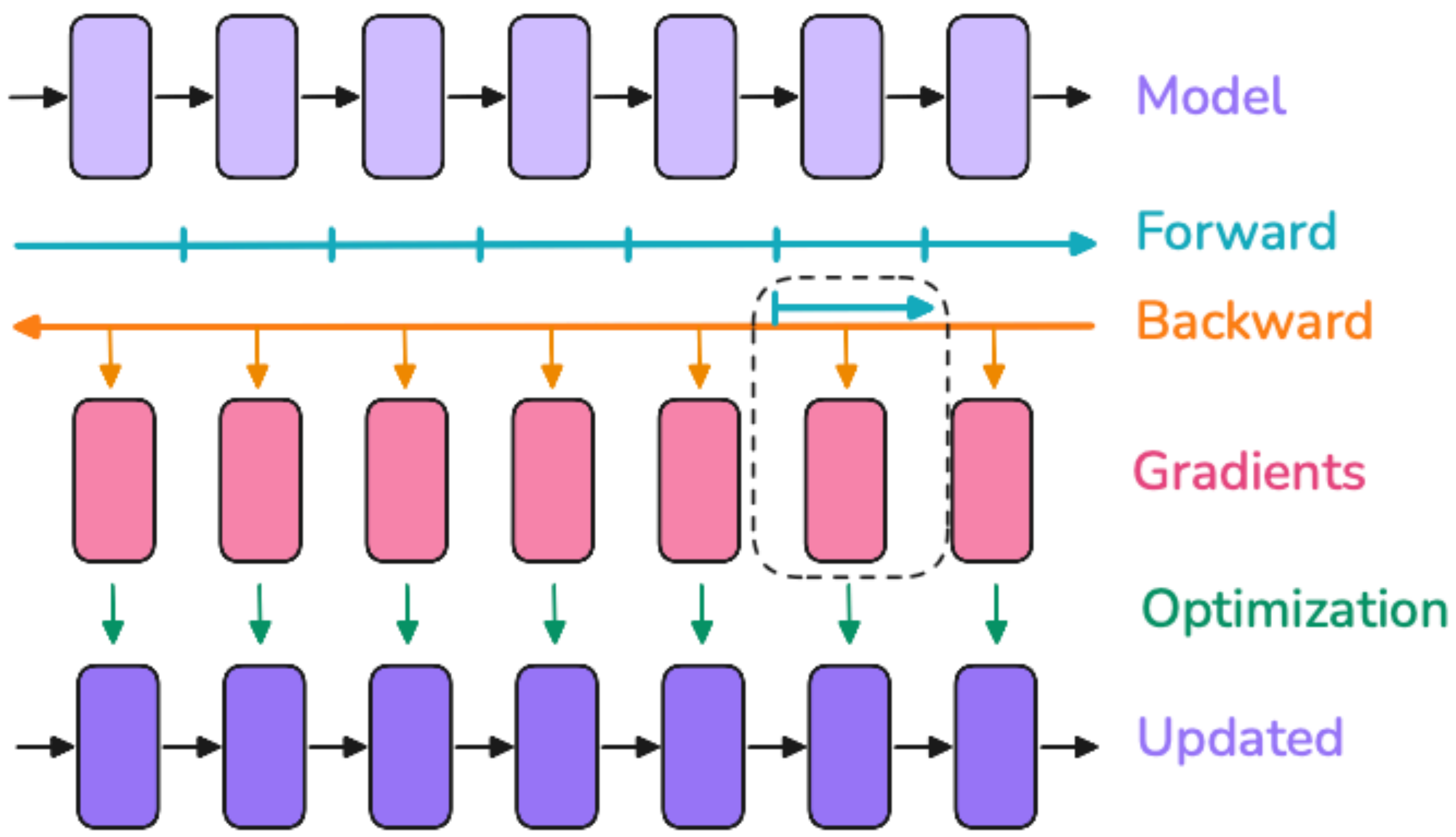

```python
class Model(nn.Module):
    def __init__(self, input_size=512, hidden_size=1024, num_layers=8):…

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
```

```python
from torch.utils.checkpoint import checkpoint
# Model with Activation Checkpointing on Every Layer
class CheckpointedModel(nn.Module):
    def __init__(self, input_size=512, hidden_size=1024, num_layers=8, use_checkpoint=True):…

    def forward(self, x):
        for layer in self.layers:
            # Use checkpointing during training to save memory
            # The checkpoint function will:
            # 1. Run the forward pass normally
            # 2. Discard intermediate activations
            # 3. Recompute them during backward pass when needed
            x = checkpoint(layer, x, use_reentrant=False)

        return x
```

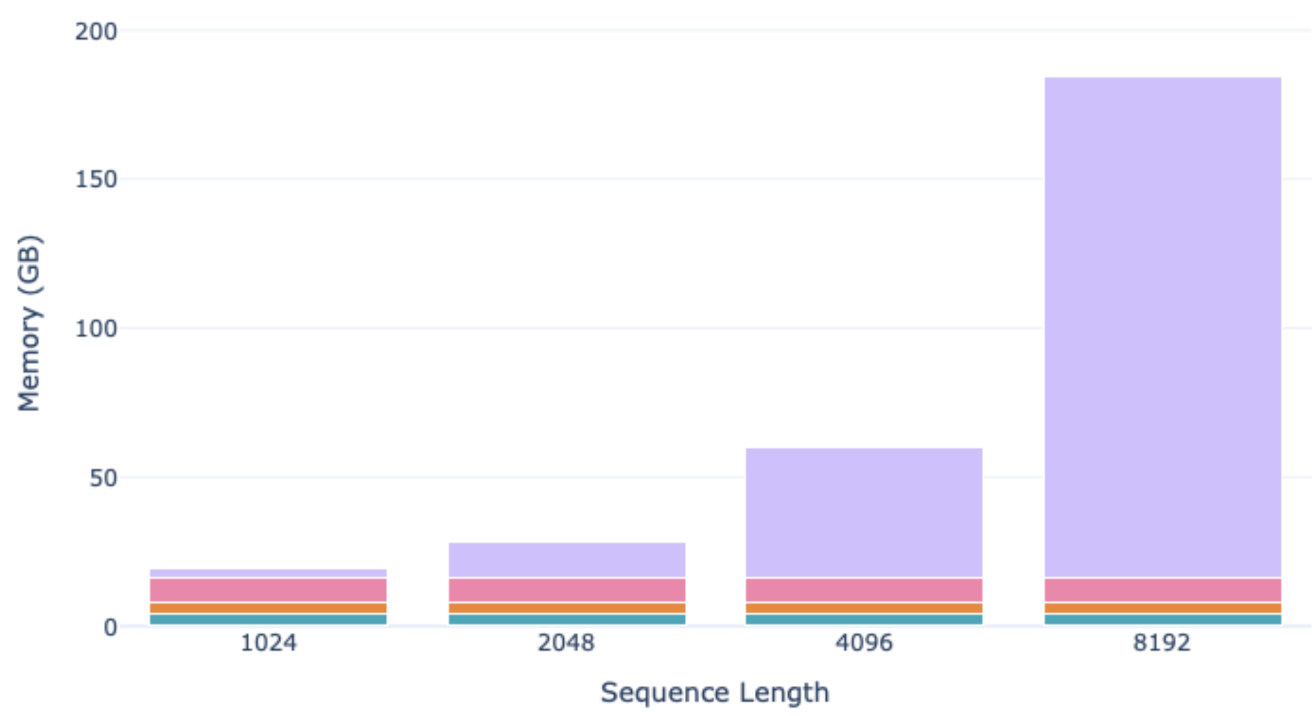
https://docs.pytorch.org/docs/stable/checkpoint.html

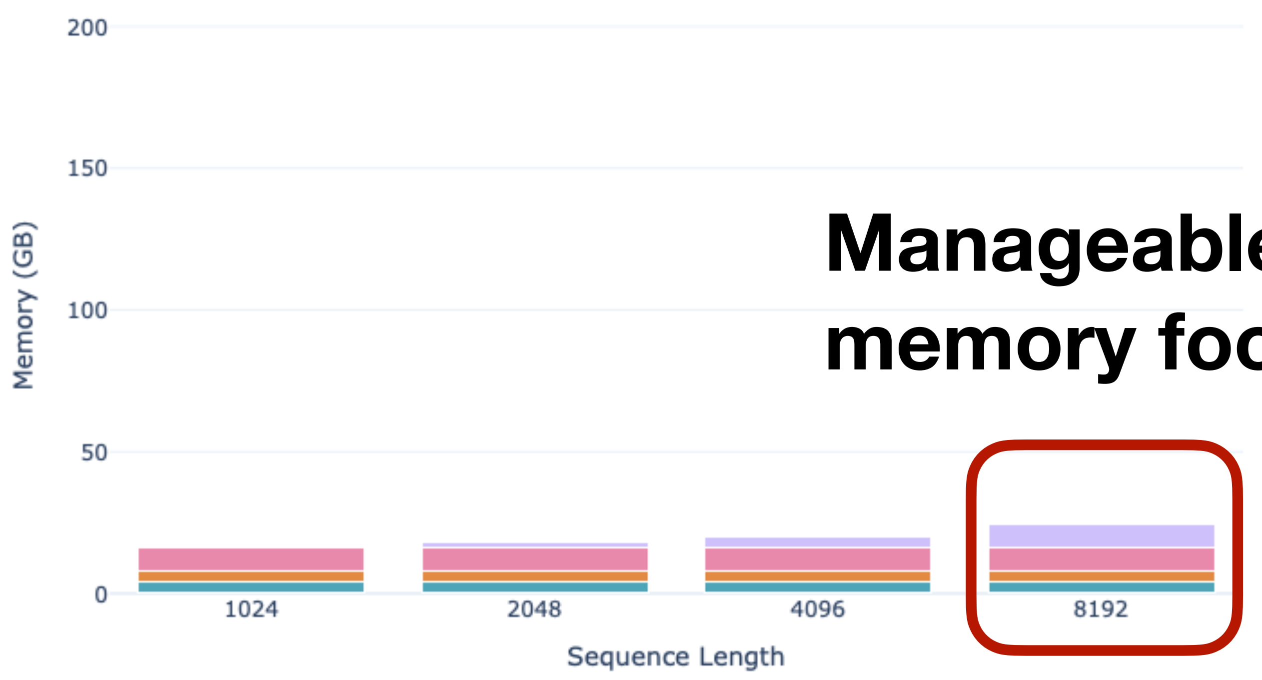
Activation Checkpointing

Drop activations from memory and re-compute when needed



1. Flash Attention comes with in-built recomputation —takes care of exponential explosion from materializing attention matrix
2. Selective re-computation (i.e. checkpointing after residual of a transformer block) is generally a good rule of thumb
3. FLOPs ↑ , Memory ↓





Manageable memory footprint

Are we done?

Recall: we kept bs=1, and scaled sequence length

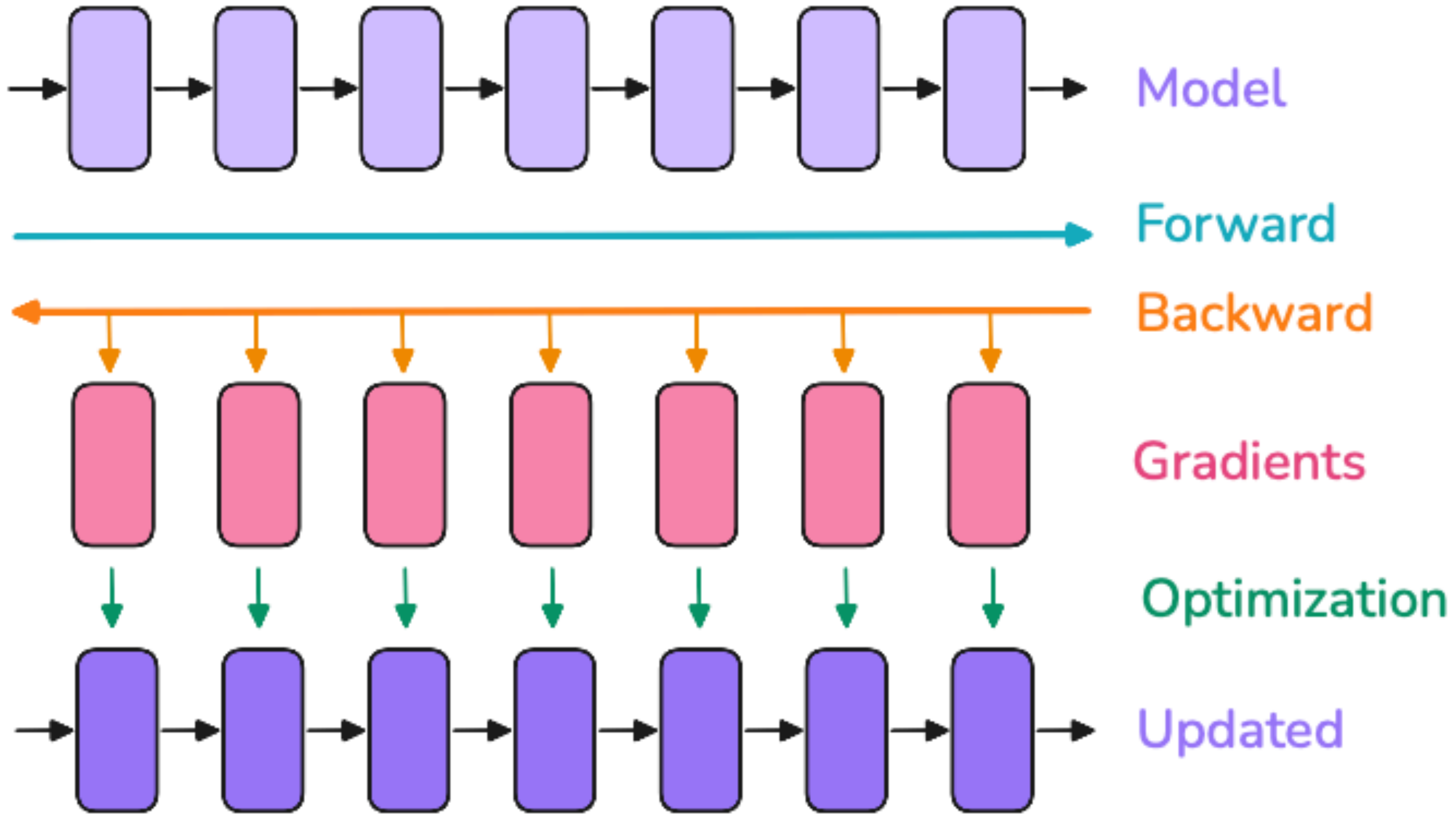
Also recall: For `bs` sequences of length L, act mem = O(bs.L^2)

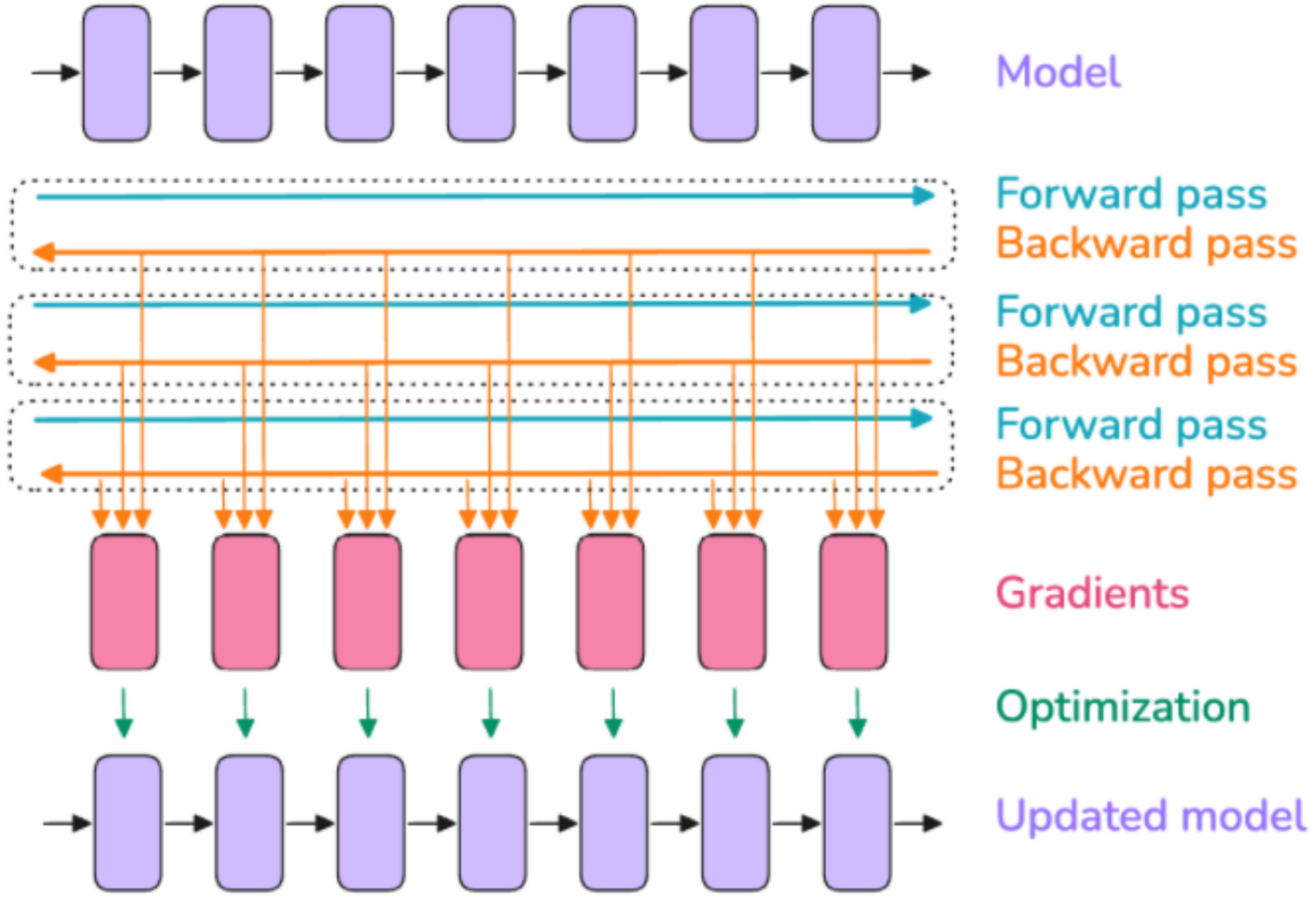
How do we scale bs without exploding activation memory?

Gradient Accumulation

Problem: we wish to emulate a larger bs than memory permits

Key observation: we don't have to immediately do an optimizer step after computing gradients

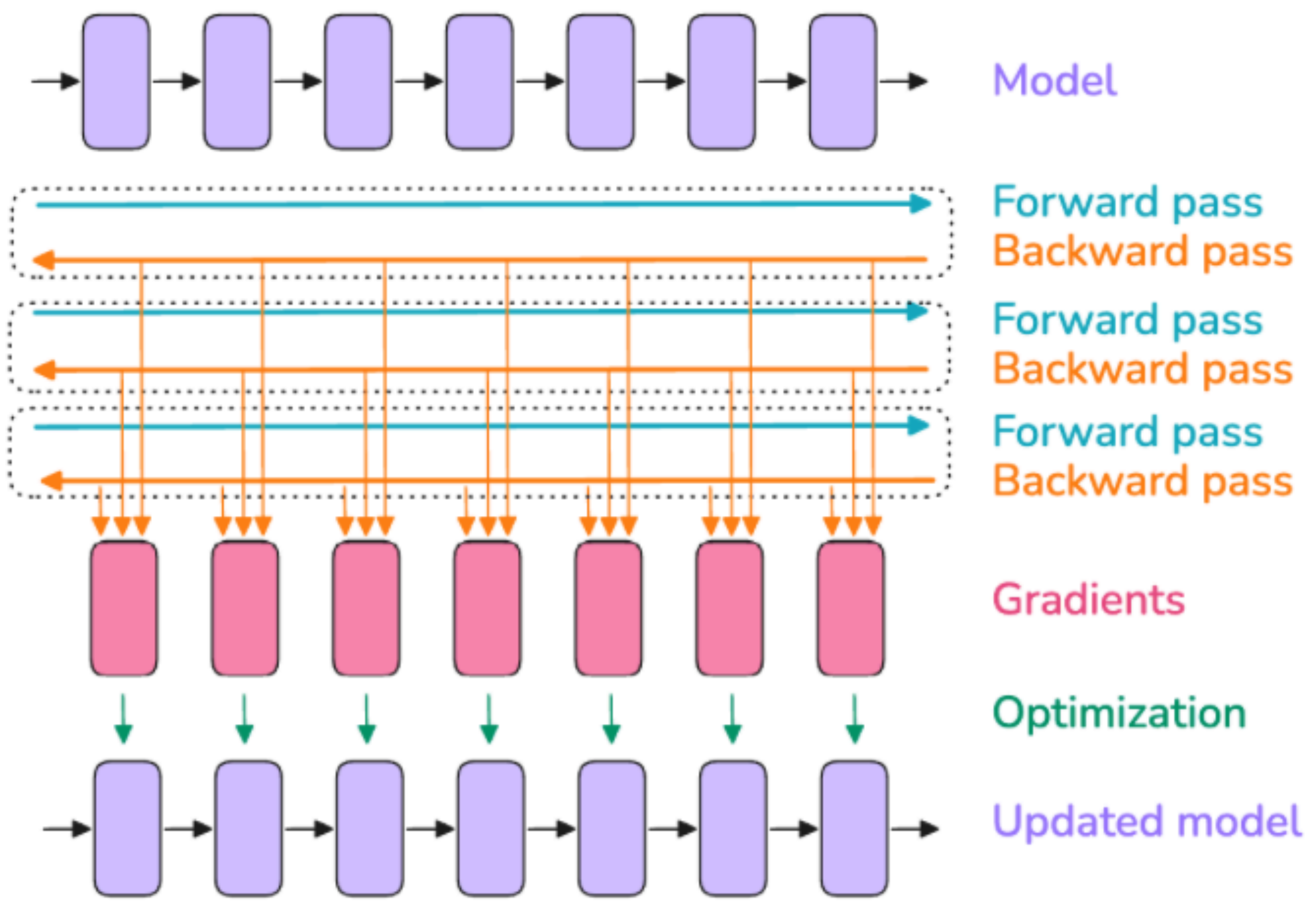




1. *Run multiple forward-backward passes*
2. *Keeping a running mean of the gradients*
3. *When you hit the target bs then do optimizer step*

Gradient Accumulation

Problem: we wish to run a batch size of 1000 but can only fit one sample in memory



- *Run 1000 times:*
  - *Run forward-backward pass on 1 sample*
  - *Accumulate gradient in buffer, ie, grad += current_grad / accumuation_steps*

We spent more compute (ran 1000 fwd-bwd passes) before doing an optimizer step to save memory

Key concept: Trade off memory for compute

Recap

- Calculating FLOPs in transformer layers for inference vs training
- Understanding compute and memory in LLM training workloads
- Fitting things on one GPU: trading off memory for compute
    - Activation checkpointing
    - Gradient accumulation

Why Is Scaling Hard?

`python train.py —small-model` —> `python train.py —big-model`

DUH!

Achieving “strong scaling”, ie, increase the number of chips used for training while achieving a proportional, linear increase in throughput, is hard due to communication overheads

Fitting everything (model, optimizer, gradients, activations) in memory

Choosing the right strategy for sharding / parallelizing when things don’t fit in memory

Alleviating communication bottlenecks that arise from parallelisms and sharding

Going Beyond A Single GPU

Let's assume we can model, gradients, optimizer, activation in one chip's memory

However, we have many chips lying around. How can we leverage these to get higher throughput?

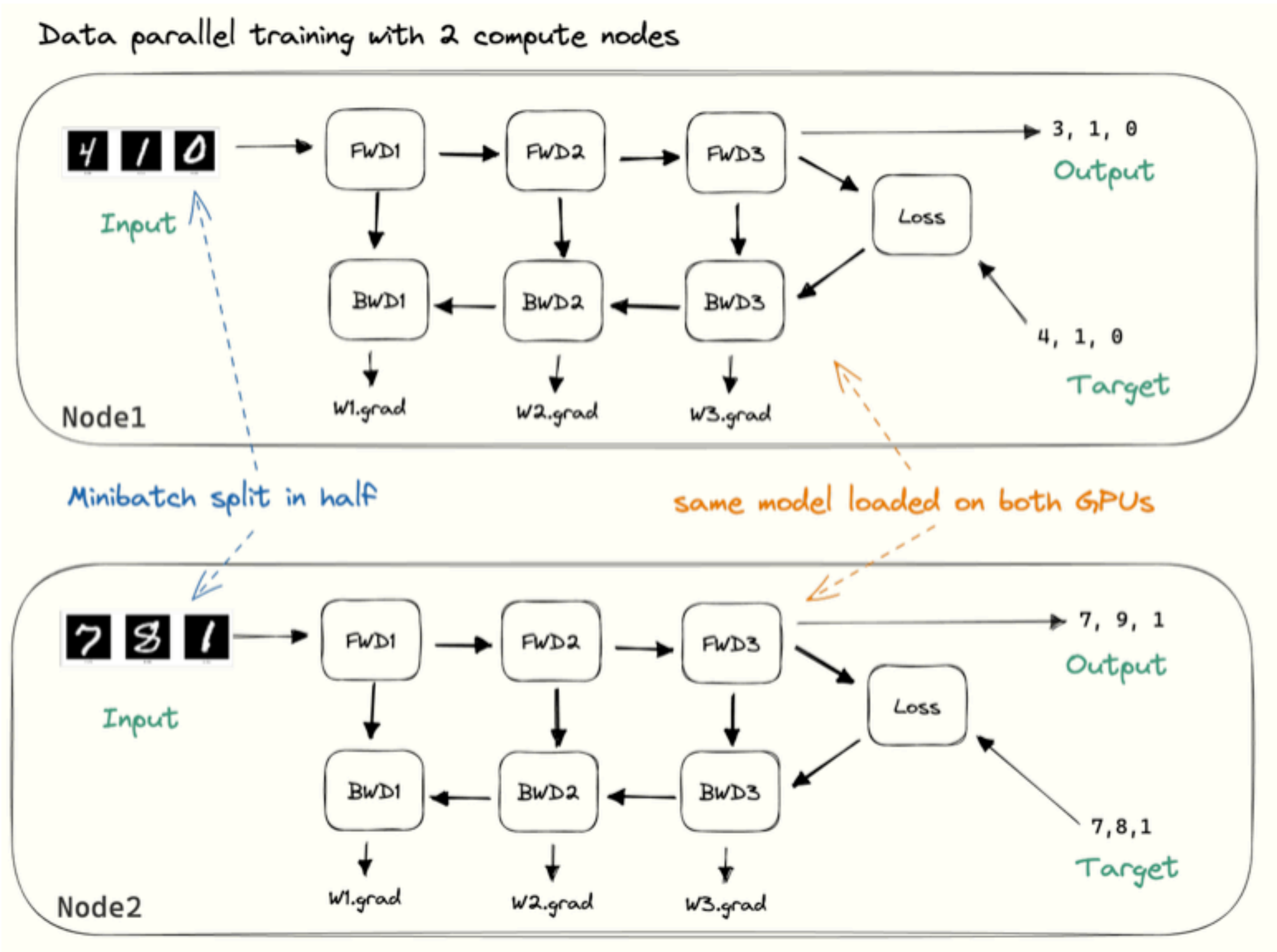
Data parallelism: run different batches of data in parallel on different chips

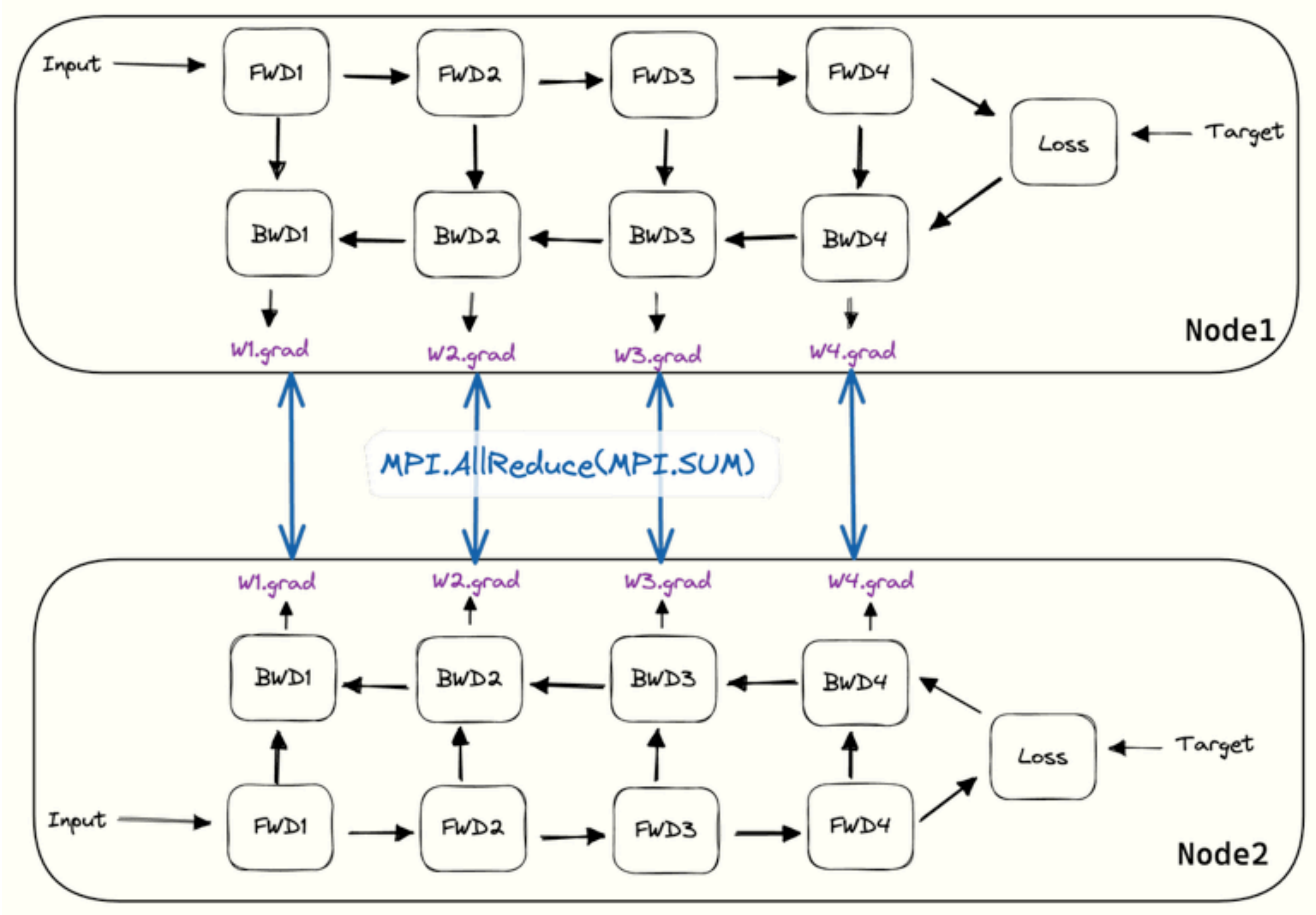
Corollary: For this to be in sync

1. **Model weights must be duplicated on different chips**
2. **After gradients are computed on different slices of data, they must be communicated across different GPUs**

Data Parallelism

Data parallelism: run different batches of data in parallel on different chips



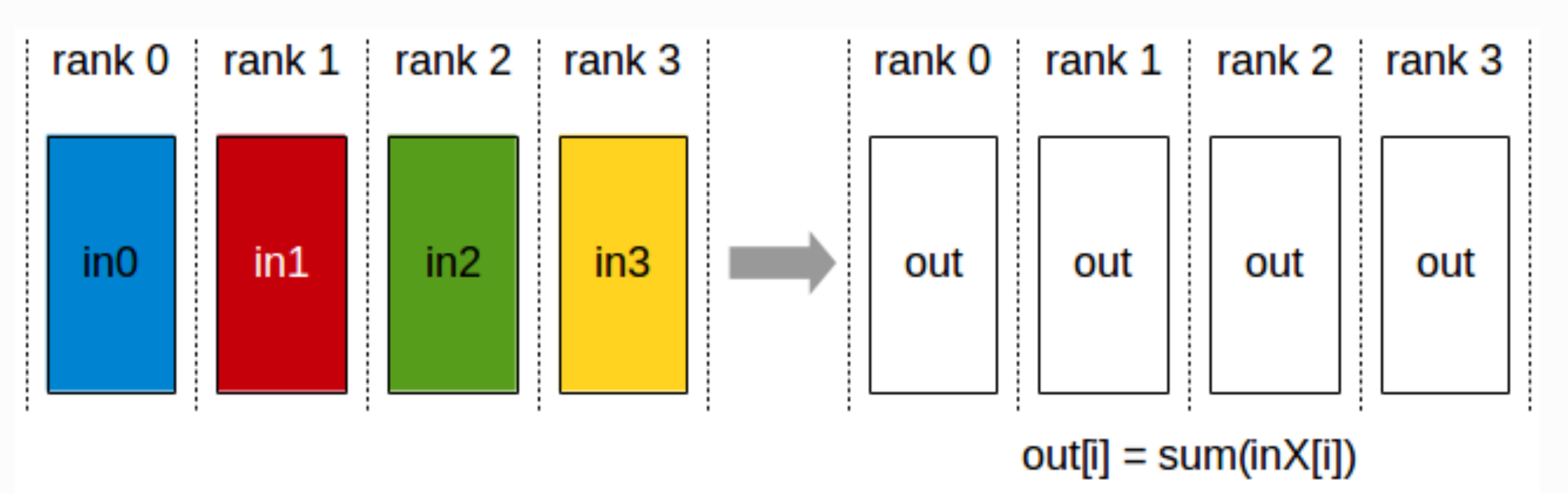


Credits: https://siboehm.com/articles/22/data-parallel-training

Data Parallelism: Communication Overhead

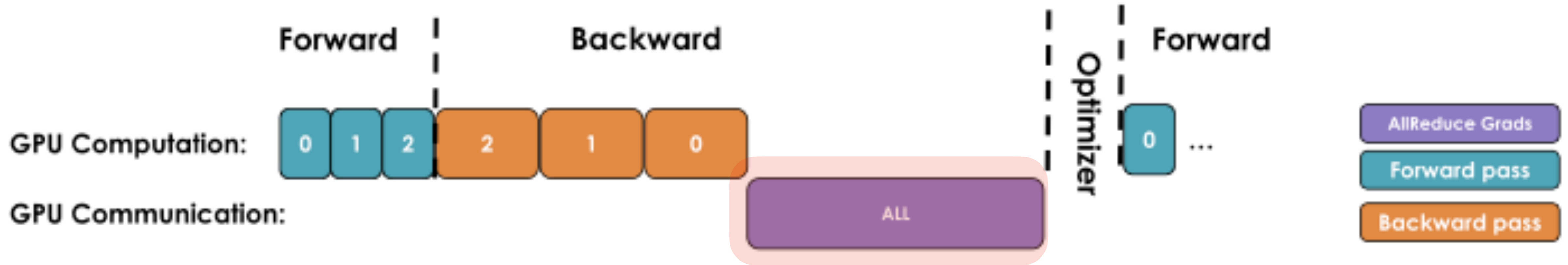
Data parallelism: run different batches of data in parallel on different chips

Requires an all-reduce* after gradients have been computed on each chip



All-Reduce operation: each rank receives the reduction of input values across ranks.

View on each chip:

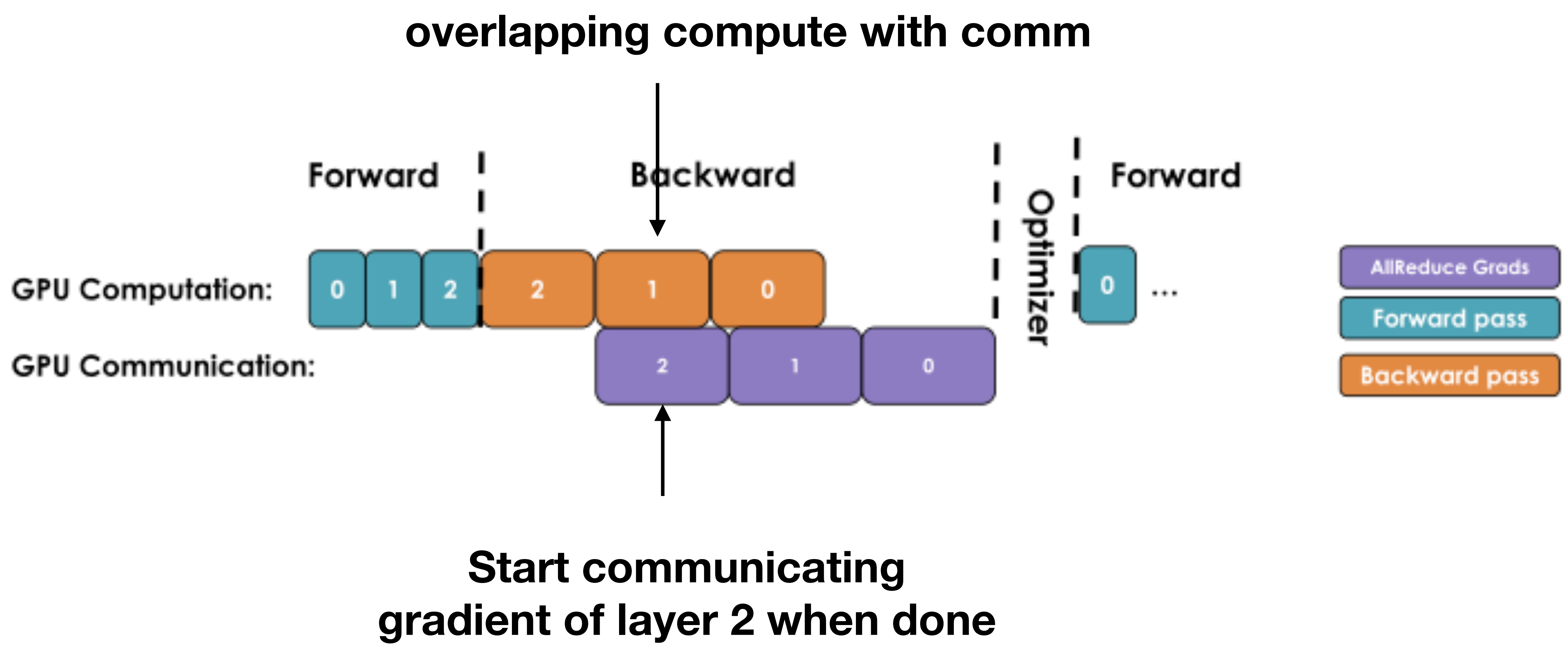


Every device is waiting for communication

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

Data Parallelism: Can We Do Better?

Key Idea: Overlap comms with compute, hide the bottleneck



More in Next Lecture!

Recap

- Calculating FLOPs in transformer layers for inference vs training
- Understanding compute and memory in LLM training workloads
- Fitting things on one GPU: trading off memory for compute
    - Activation checkpointing
    - Gradient accumulation
- Data parallelism: communication bottlenecks
    - Overlapping comms with compute to remove bottlenecks

References

1. Ultrascale Playbook, HuggingFace, https://huggingface.co/spaces/nanotron/ultrascale-playbook

2. How to Scale Your Model, https://jax-ml.github.io/scaling-book

3. https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9

4. Training Deep Nets with Sublinear Memory Cost, https://arxiv.org/abs/1604.06174

5. Reducing Activation Recomputation in Large Transformer Models, https://arxiv.org/abs/2205.05198

6. https://github.com/karpathy/nano-llama31