Fantastic Language Models and How to Build Them

Guest Lecture — CS 4644/7643 Deep Learning

Georgia Tech || Zoom || Folks 2x-ing the Recording

October 20, 2025

Georgia Tech

Siddharth Karamcheti

On the Importance of "Building"

Today — A *practical* take on large-scale language models (LLMs).

Whirlwind tour of the full pipeline:

- **Model Architecture** — Evolution of the Transformer
- **Training at Scale** — From 124M to 100B+ Parameters
- **(Briefly) Fine-tuning & Inference** — Tips & Tricks

Punchline: From "folk knowledge" —> insight / intuition / (re-)discovery!

Please ask lots of questions! Why is this information useful to <YOU>?

Part I. Evolution of the Transformer

"Experiment is the mother of knowledge."
— Madeline L'Engle, *A Wrinkle in Time*

Recipe for a Good™ Language Model

Massive amounts of cheap, easy to acquire data…

X

… a simple, high-throughput way to consume it!

Natural to scale with data.
Composable and “general”.

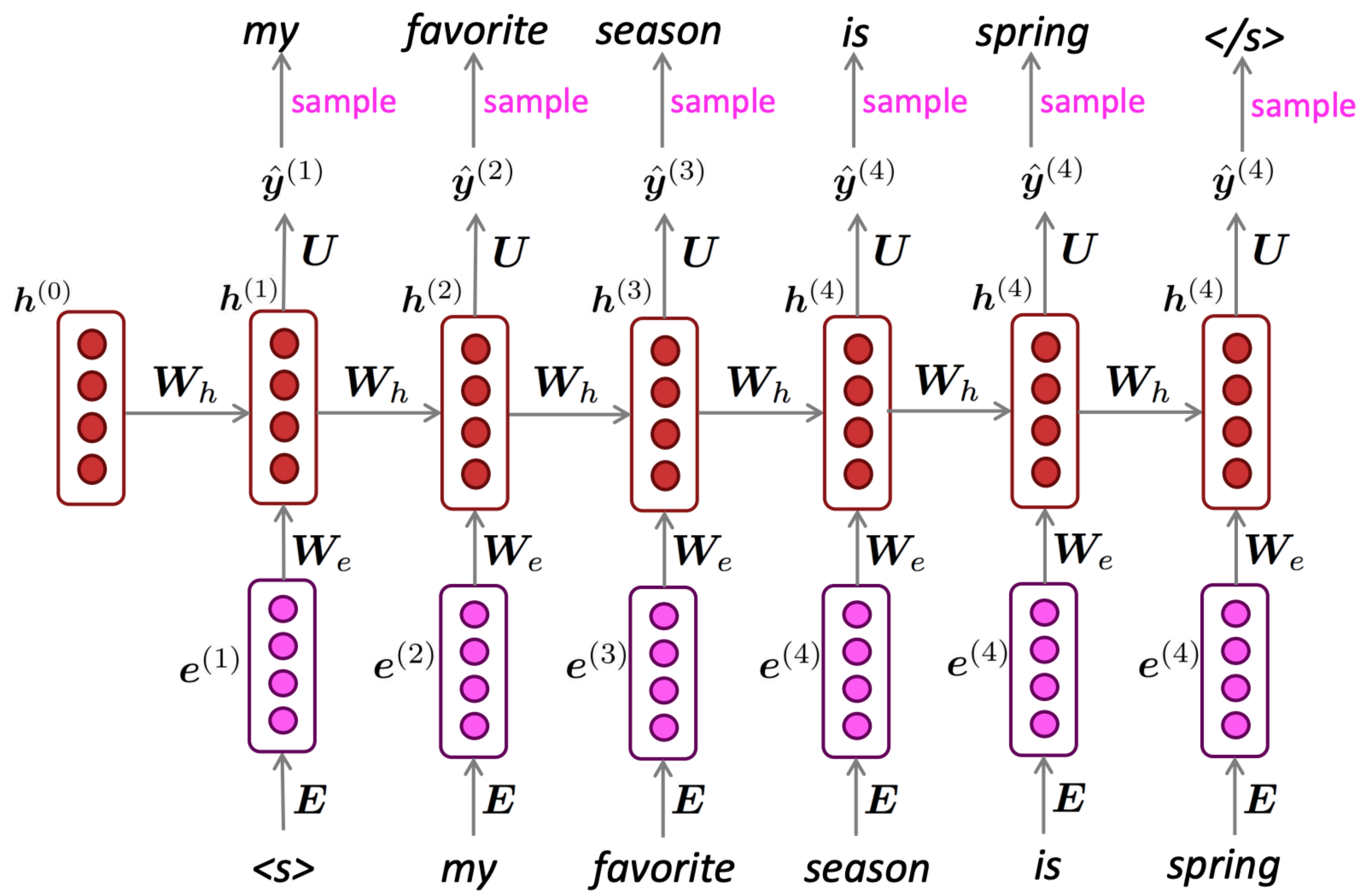
Fast & parallelizable training.
High hardware utilization.

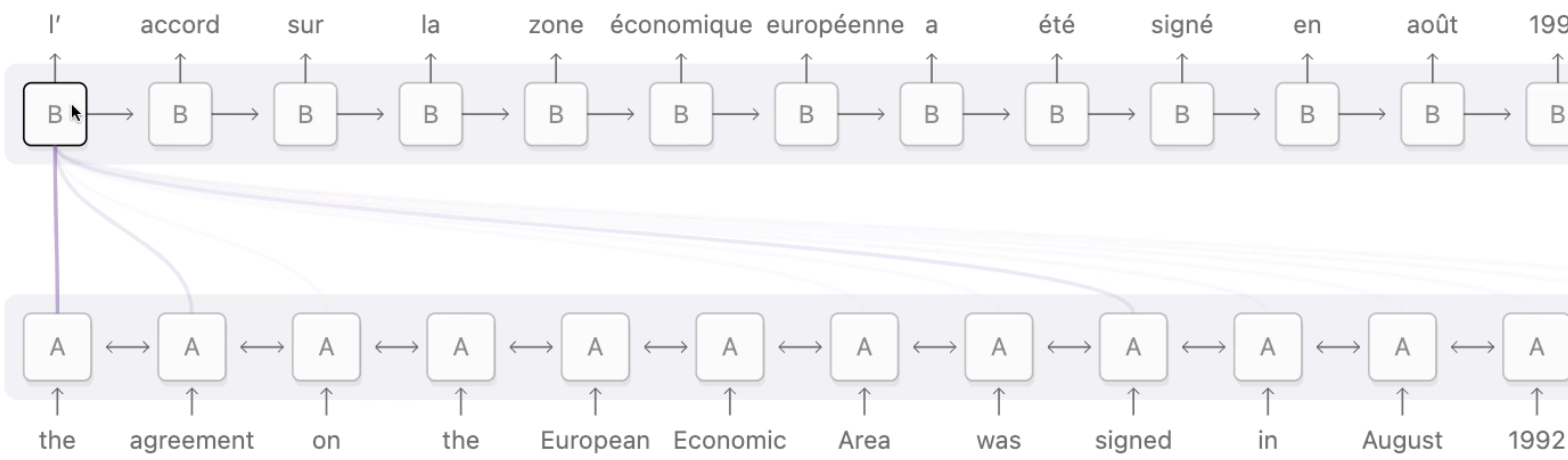
Minimal “assumptions” on
relationships between data?

< Let’s Rewind >

Pre-2017 — Historical Context

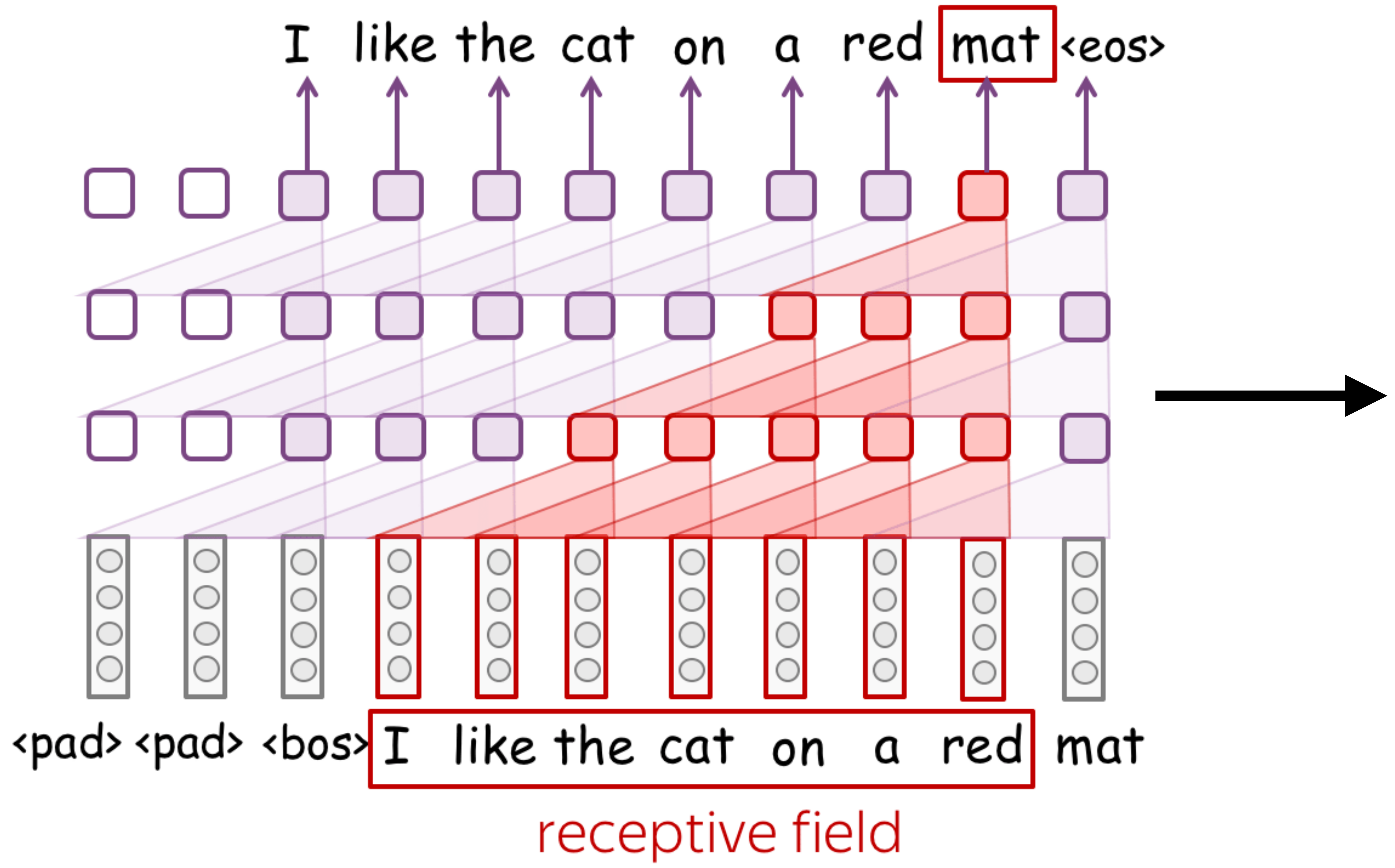
RNNs

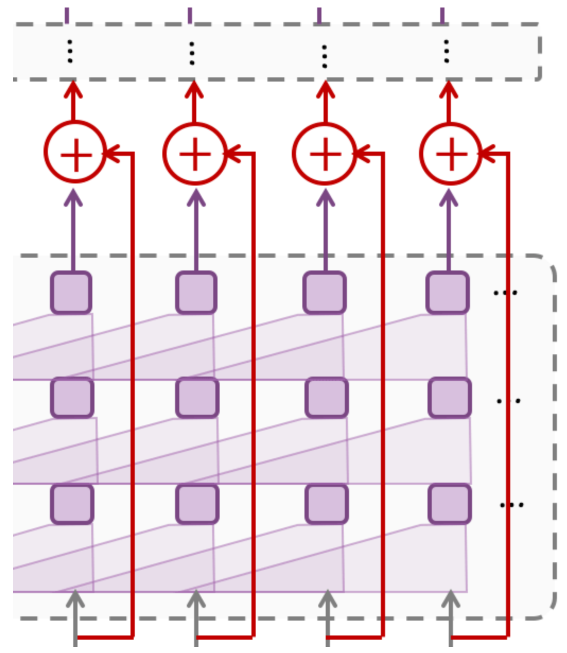




RNN Key Ideas: Long Context, **Attention**

CNNs





Residual Connection

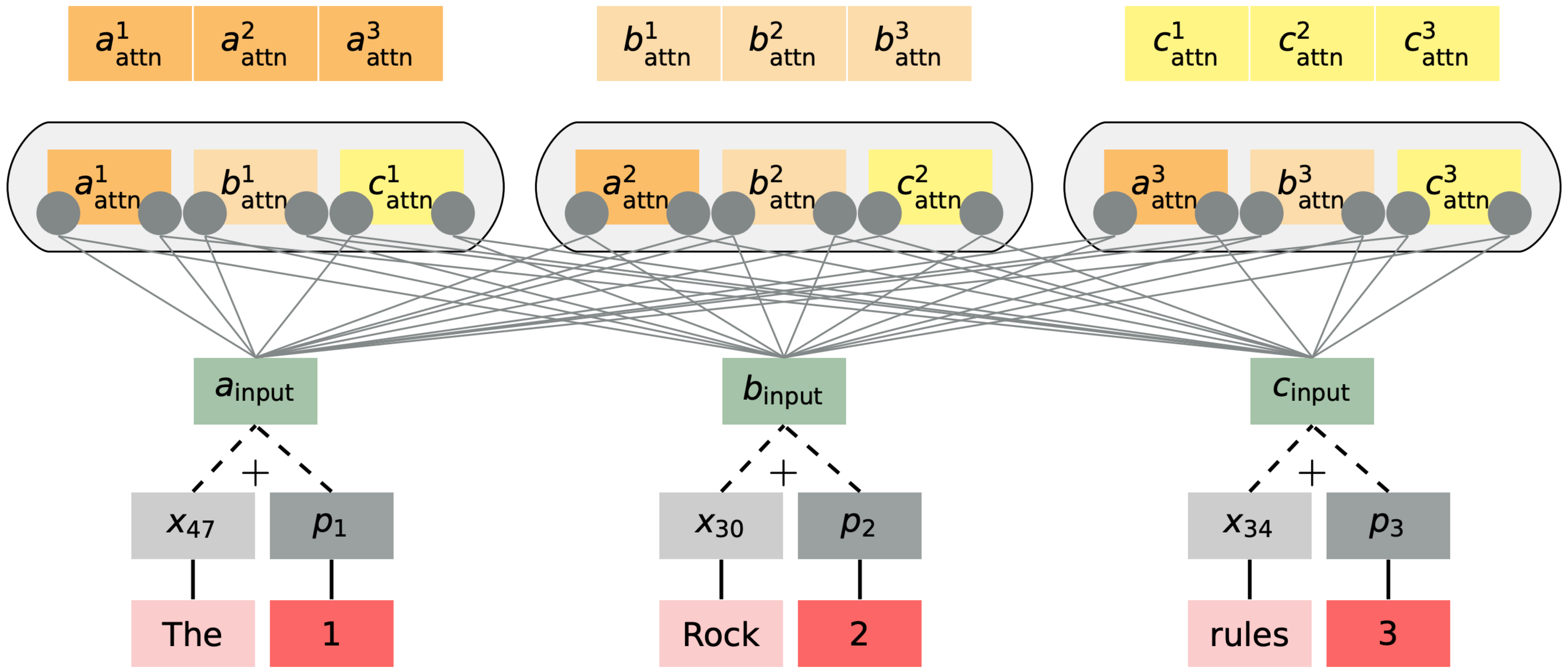
CNN Key Ideas:

- Layer: Multiple "Filters" (Views)
- **Scaling Depth** w/ Residuals
- **Parallelizable!**

< How do I do better? >

Reference: "Attention and Augmented Recurrent Neural Networks," Chris Olah and Shan Carter. *Distill, 2016.*
Reference: "Convolutional Neural Networks for Text," Lena Voita. ML for NLP @ YSDA

Formulating the Self-Attention Block

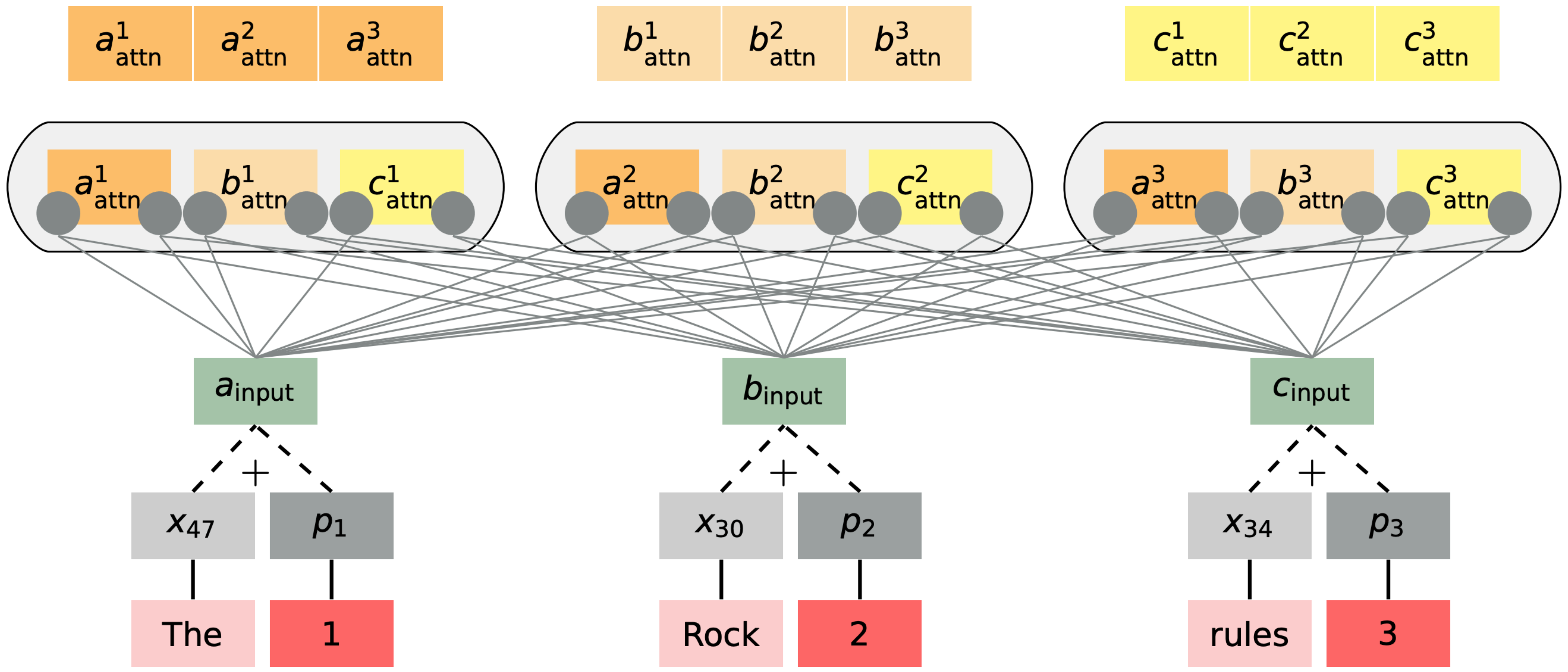



```python
class Attention(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int):
        super().__init__()
        self.n_heads, self.dk = n_heads, (embed_dim // n_heads)
        self.qkv = nn.Linear(embed_dim, 3 * embed_dim)
        self.proj = nn.Linear(embed_dim, embed_dim)

    def forward(self, x: Tensor[bsz, seq, embed_dim]):
        q, k, v = rearrange(
            self.qkv(x),
            "bsz seq (qkv nh dk) -> qkv bsz nh seq dk",
            qkv=3,
            nh=self.n_heads,  # Different "views" (like CNN filters)!
            dk=self.dk,
        ).unbind(0)
```

Self-Attention: “The” —> **query**, **key,** & **value**

Multi-Headed: Different “views” per layer

Formulating the Self-Attention Block




```python
class Attention(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int):
        super().__init__()
        self.n_heads, self.dk = n_heads, (embed_dim // n_heads)
        self.qkv = nn.Linear(embed_dim, 3 * embed_dim)
        self.proj = nn.Linear(embed_dim, embed_dim)

    def forward(self, x: Tensor[bsz, seq, embed_dim]):
        q, k, v = rearrange(
            self.qkv(x),
            "bsz seq (qkv nh dk) -> qkv bsz nh seq dk",
            qkv=3,
            nh=self.n_heads,  # Different "views" (like CNN filters)!
            dk=self.dk,
        ).unbind(0)

        # RNN Attention --> *for each view*
        scores = torch.softmax(
            q @ (k.transpose(-2, -1)),
            dim=-1
        )
        return self.proj(
            rearrange(scores @ v, "b nh seq dk -> b seq (nh dk)")
        )
```

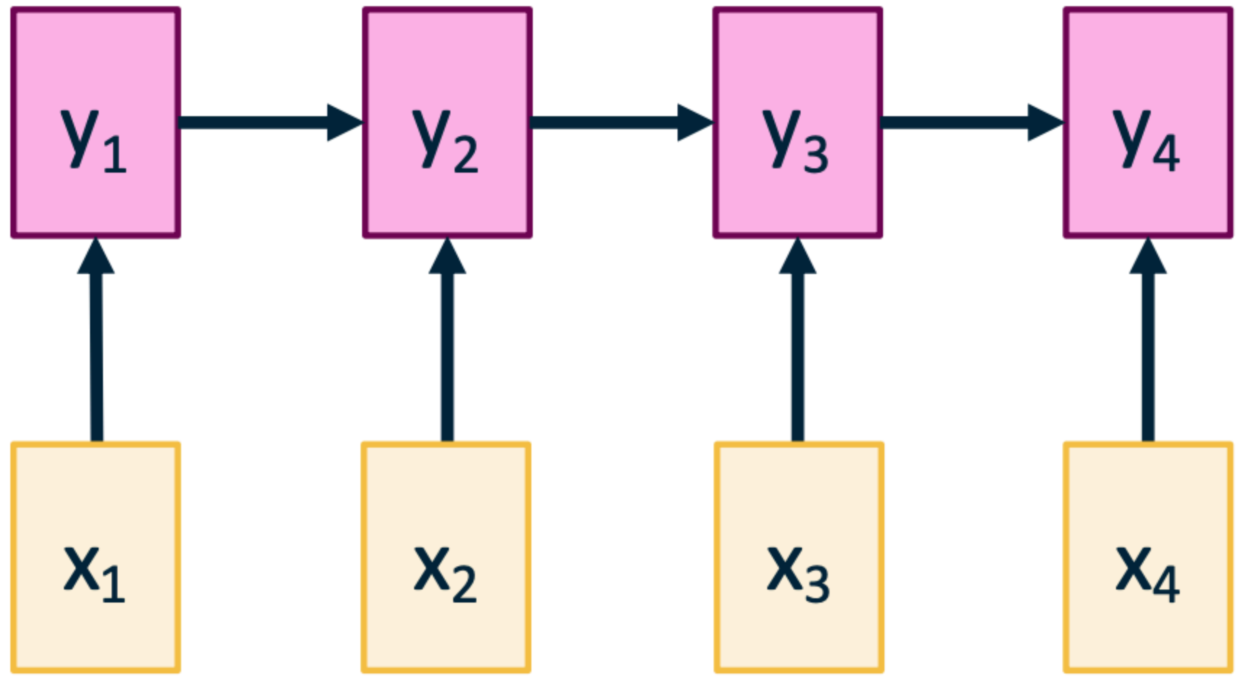
Self-Attention: "The" —> **query**, **key,** & **value**

Multi-Headed: Different "views" per layer

< Is this actually better? >

Aside — Self-Attention & Parallelization

Recurrent Neural Network

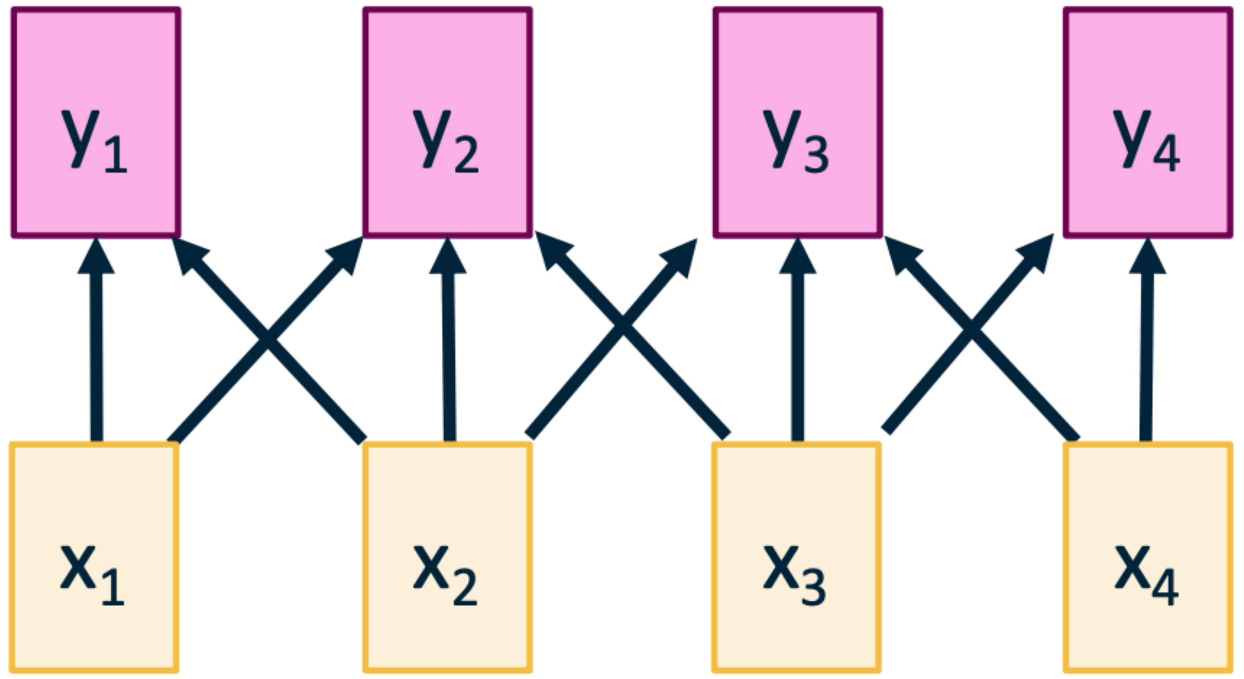


Works on **Ordered Sequences**

(+) Good at long sequences: After one RNN layer, h_T "sees" the whole sequence

(-) Not parallelizable: need to compute hidden states sequentially

1D Convolution

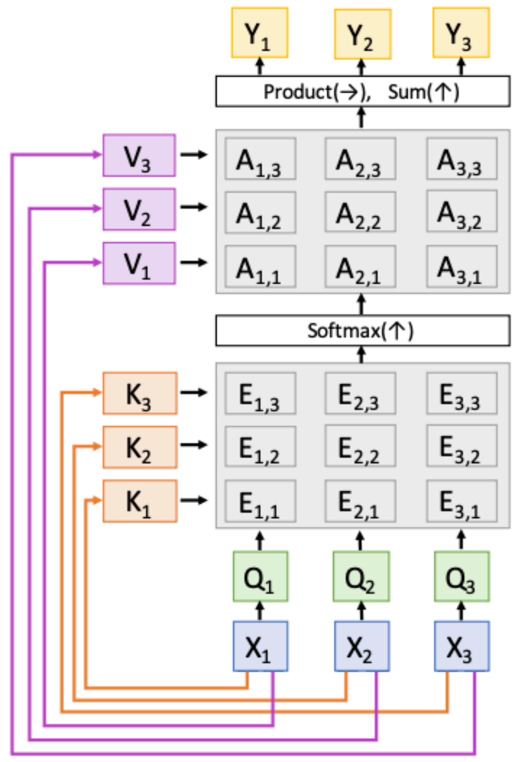


Works on **Multidimensional Grids**

(-) Bad at long sequences: Need to stack many conv layers for outputs to "see" the whole sequence

(+) Highly parallel: Each output can be computed in parallel

Self-Attention



Works on **Sets of Vectors**

(+) Good at long sequences: after one self-attention layer, each output "sees" all inputs!

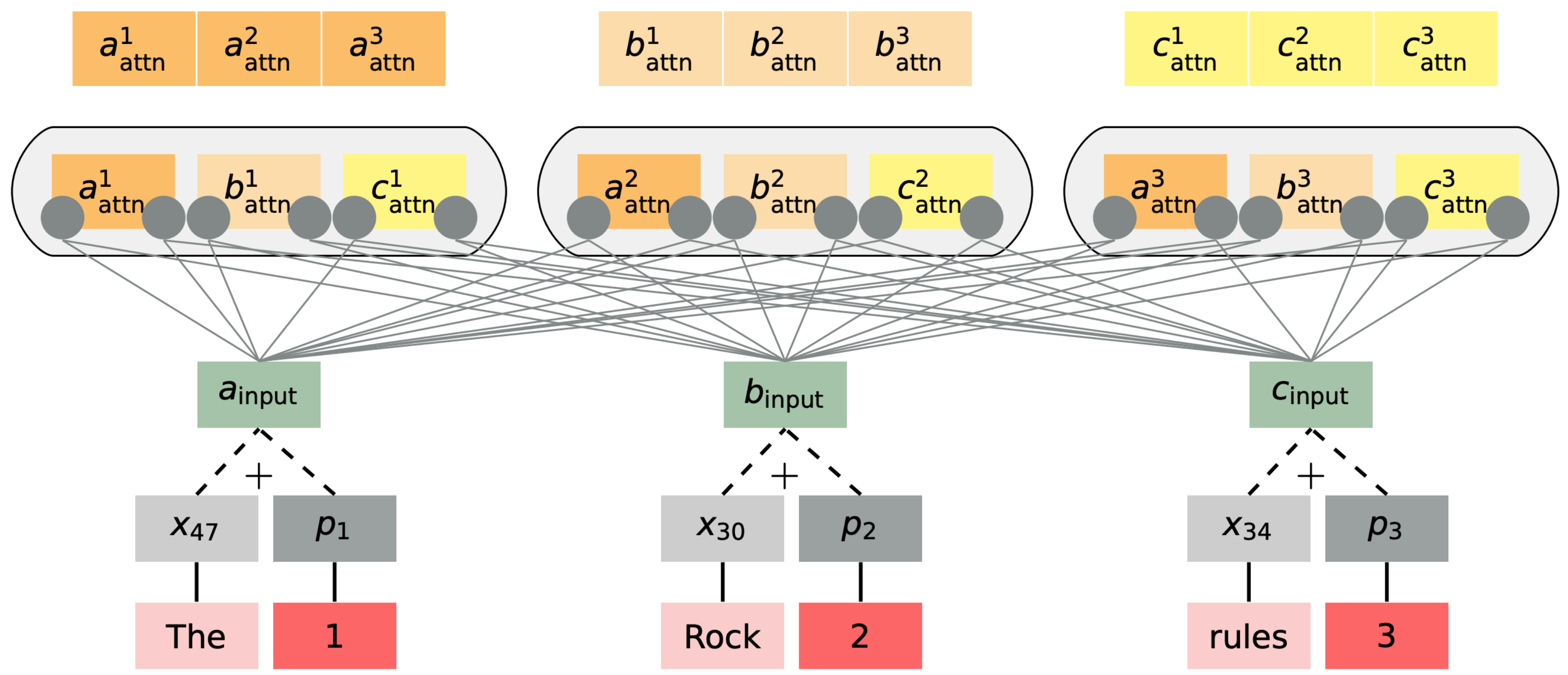
(+) Highly parallel: Each output can be computed in parallel

(-) Very memory intensive

< Great! But… what am I missing? >

Formulating the Self-Attention Block

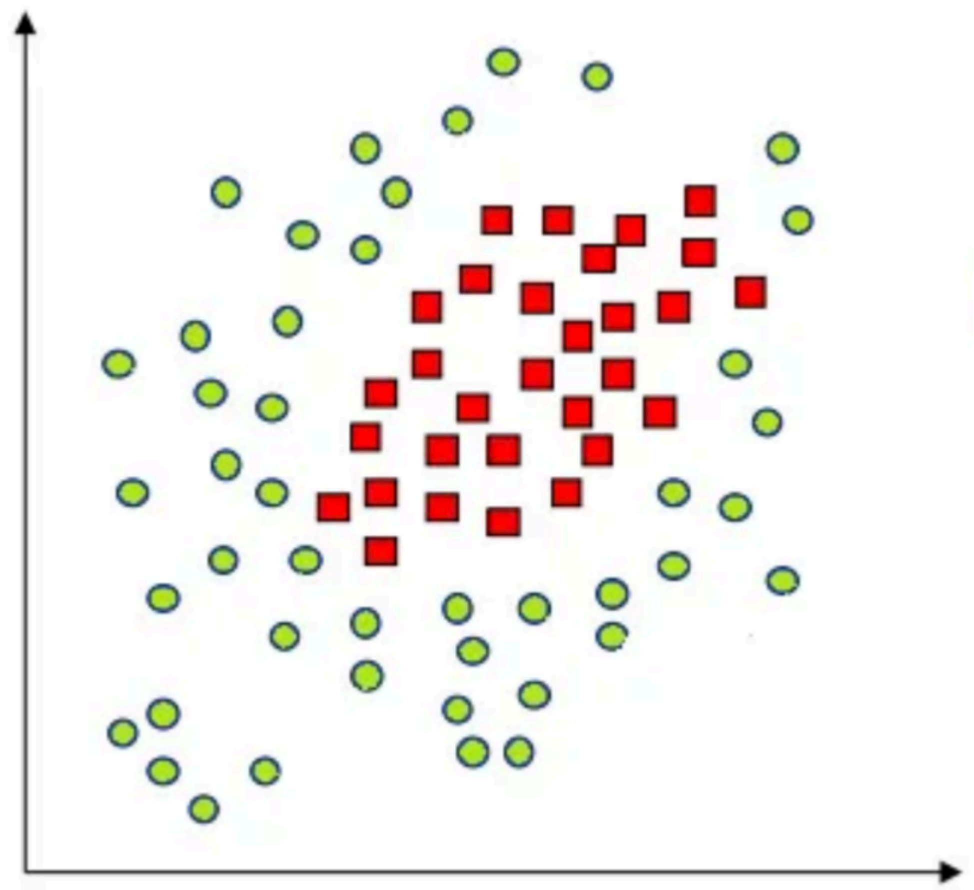
```python
class Attention(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int):
        super().__init__()
        self.n_heads, self.dk = n_heads, (embed_dim // n_heads)
        self.qkv = nn.Linear(embed_dim, 3 * embed_dim)
        self.proj = nn.Linear(embed_dim, embed_dim)

    def forward(self, x: Tensor[bsz, seq, embed_dim]):
        q, k, v = rearrange(
            self.qkv(x),
            "bsz seq (qkv nh dk) -> qkv bsz nh seq dk",
            qkv=3,
            nh=self.n_heads,  # Different "views" (like CNN filters)!
            dk=self.dk,
        ).unbind(0)

        # RNN Attention --> *for each view*
        scores = torch.softmax(
            q @ (k.transpose(-2, -1)),
            dim=-1
        )
        return self.proj(
            rearrange(scores @ v, "b nh seq dk -> b seq (nh dk)")
        )
```



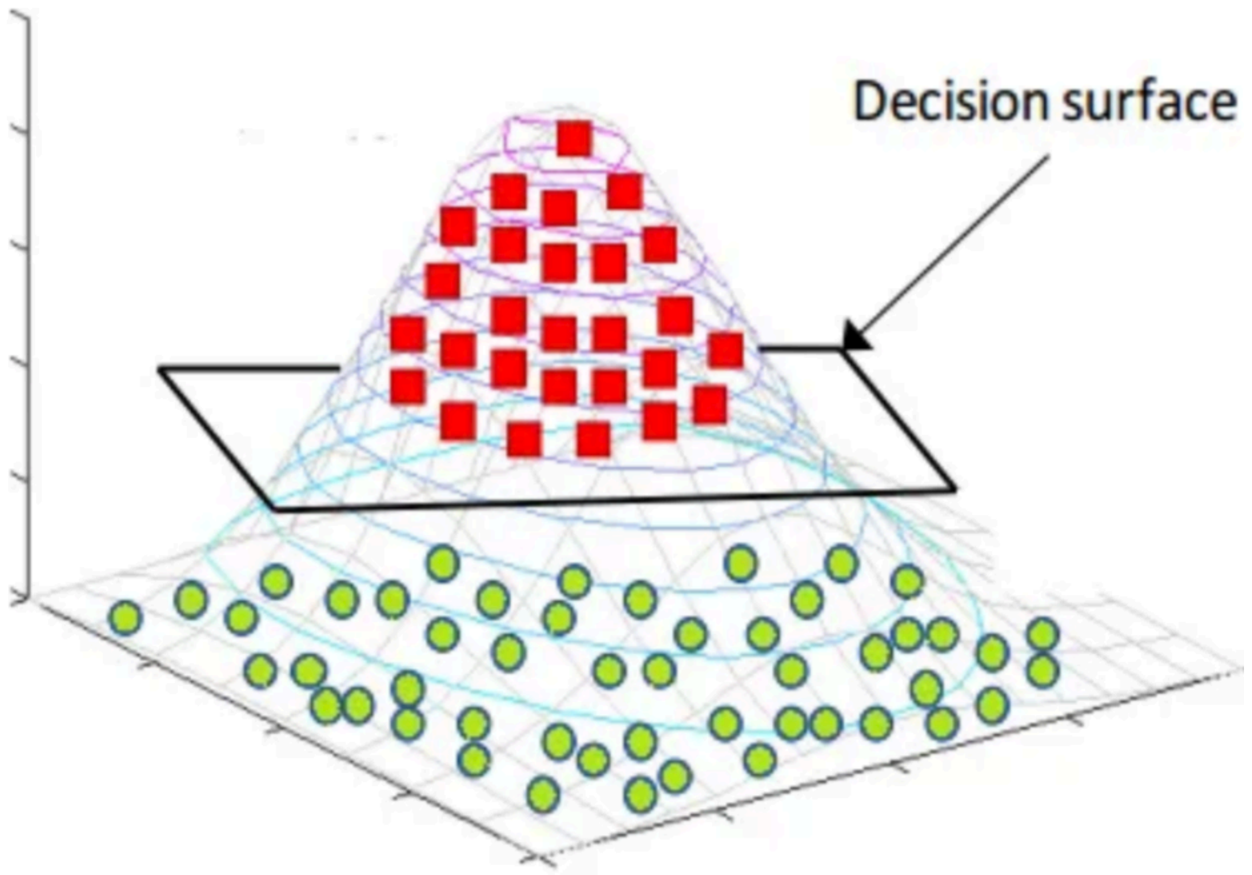
< Where's my nonlinearity? >

Expressivity through Nonlinearity

```python
class ExpressiveTransformerBlock(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int, up: int = 4):
        super().__init__()
        self.attn = Attention(embed_dim, n_heads)

        # Project *up* to high-dimension, nonlinear, compress!
        self.mlp = nn.Sequential(
          nn.Linear(embed_dim, up * embed_dim),
          nn.ReLU(),
          nn.Linear(up * embed_dim, embed_dim)
        )

    def forward(self, x: T[bsz, seq, embed_dim]):
        x = x + self.attn(x)
        x = x + self.mlp(x)
        return x
```



ML 101 —> SVMs & "Implicit Lifting"

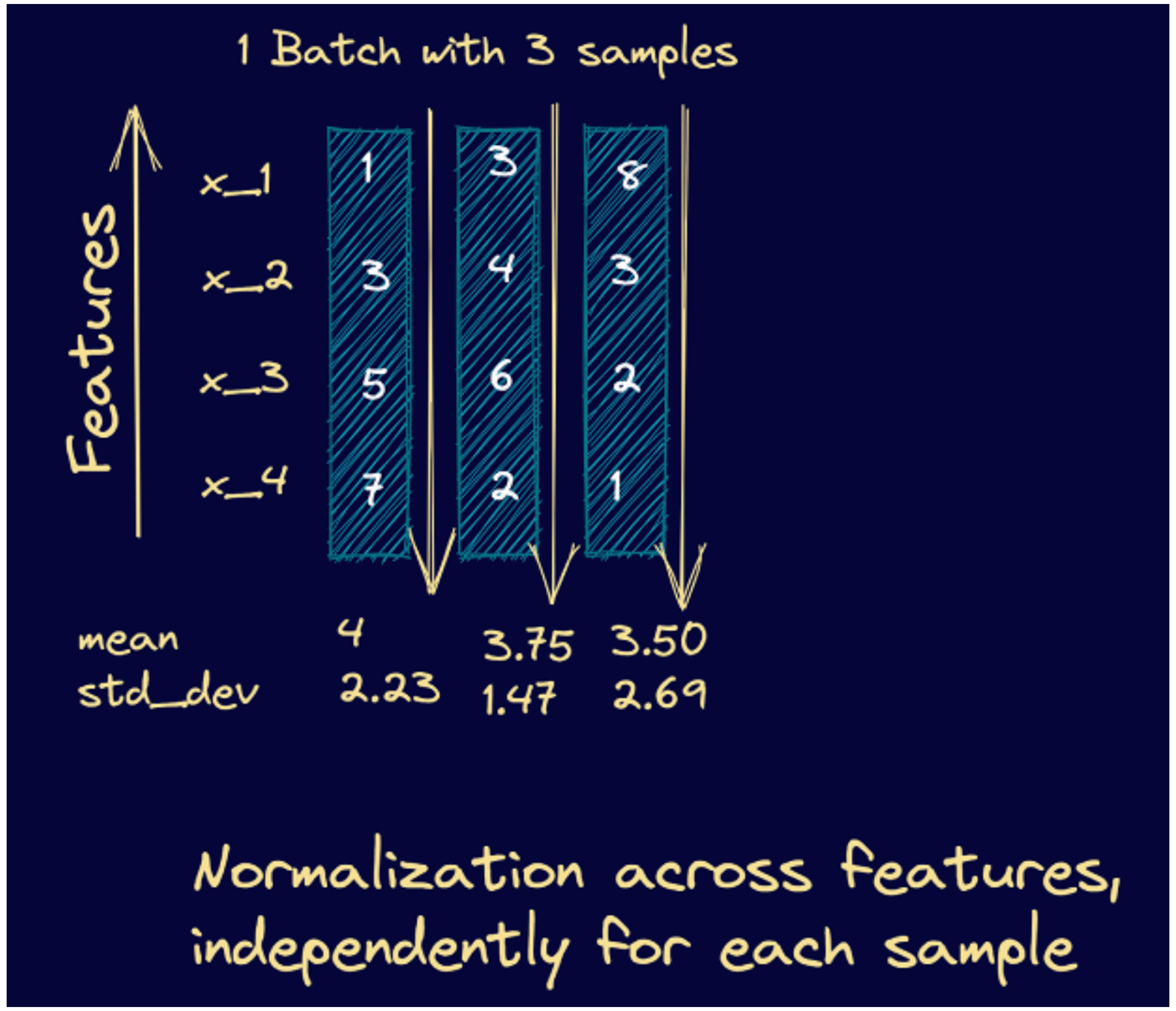


Residual + MLP —> "Sharpen" + "Forget"

< New Problem — Activations Blow Up! >

Going Deeper —> Activation Instability

```
class NormalizedTransformerBlock(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int, up: int = 4):
        super().__init__()
        self.attn = Attention(embed_dim, n_heads)
        self.mlp = nn.Sequential(
            nn.Linear(embed_dim, up * embed_dim),
            nn.ReLU(),
            nn.Linear(up * embed_dim, embed_dim)
        )

        # Add Normalization Layers
        self.attn_norm = nn.LayerNorm(embed_dim)
        self.mlp_norm = nn.LayerNorm(embed_dim)

    def forward(self, x: T[bsz, seq, embed_dim]):
        x = self.attn_norm(x + self.attn(x))
        x = self.mlp_norm(x + self.mlp(x))
        return x
```



Layer Normalization

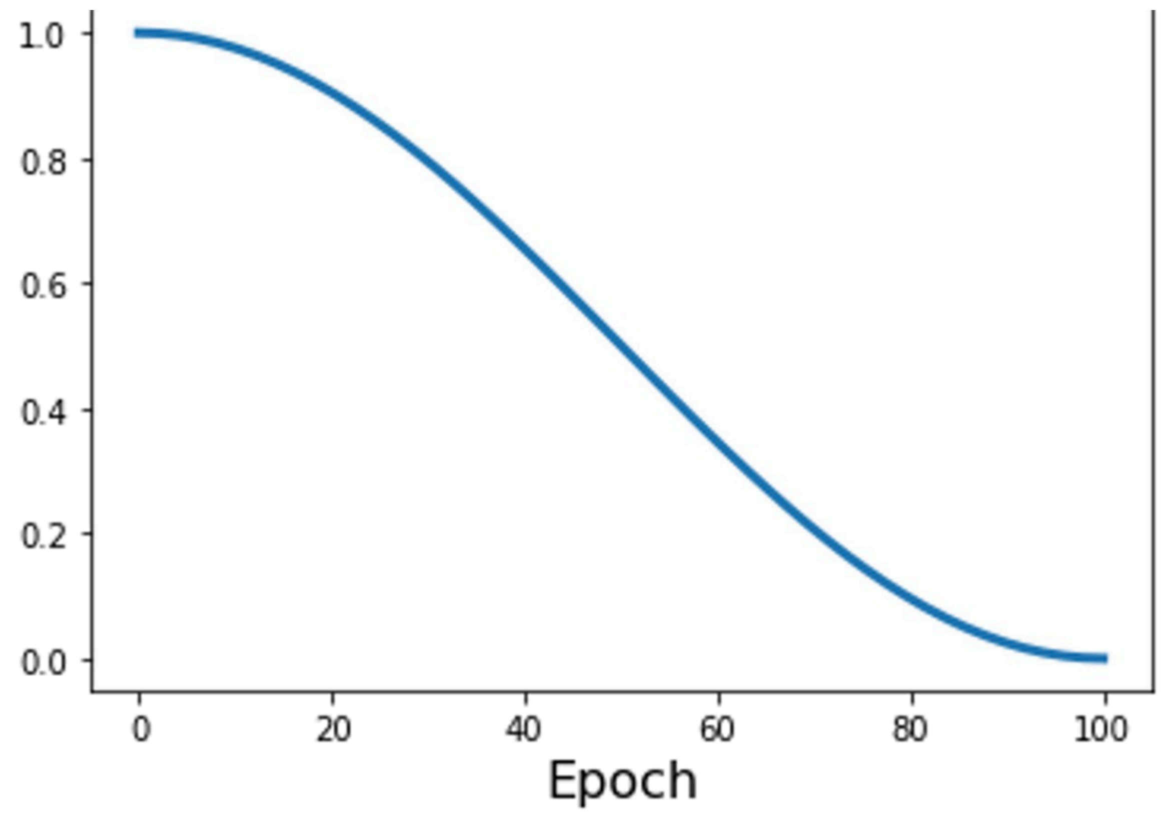
< And... we're done? >

Reference: "Build Better DL Models with Batch and Layer Normalization," Priya Bala — pinecone.io

Well, Shucks —> Emergent Optimization Problems

Typical LR Decay

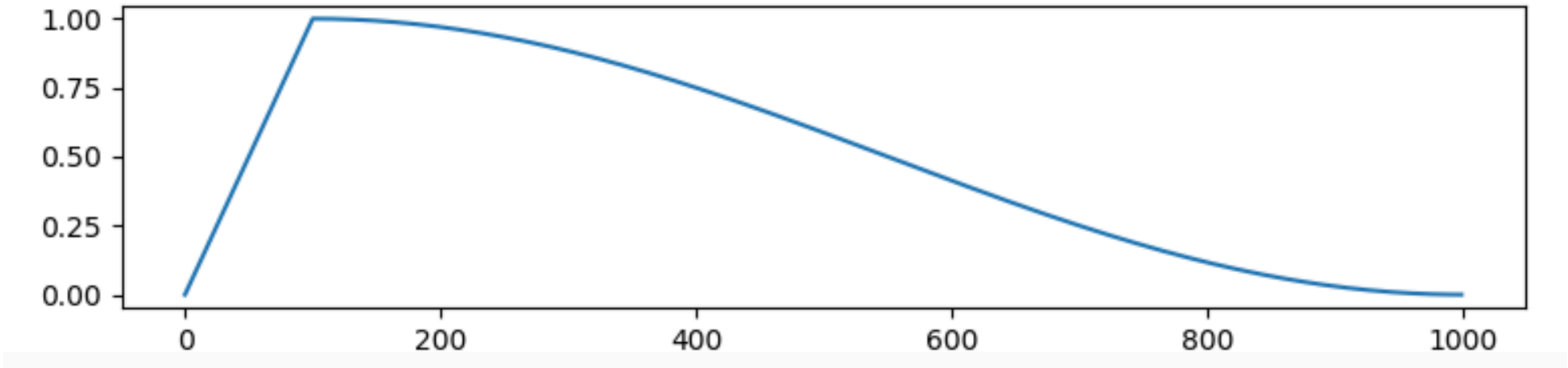
[ML 101, **Lecture 10**]



Normalized Transformer

Transformer Pretraining LR Schedule

Linear Warmup (5% of Training) then Decay...

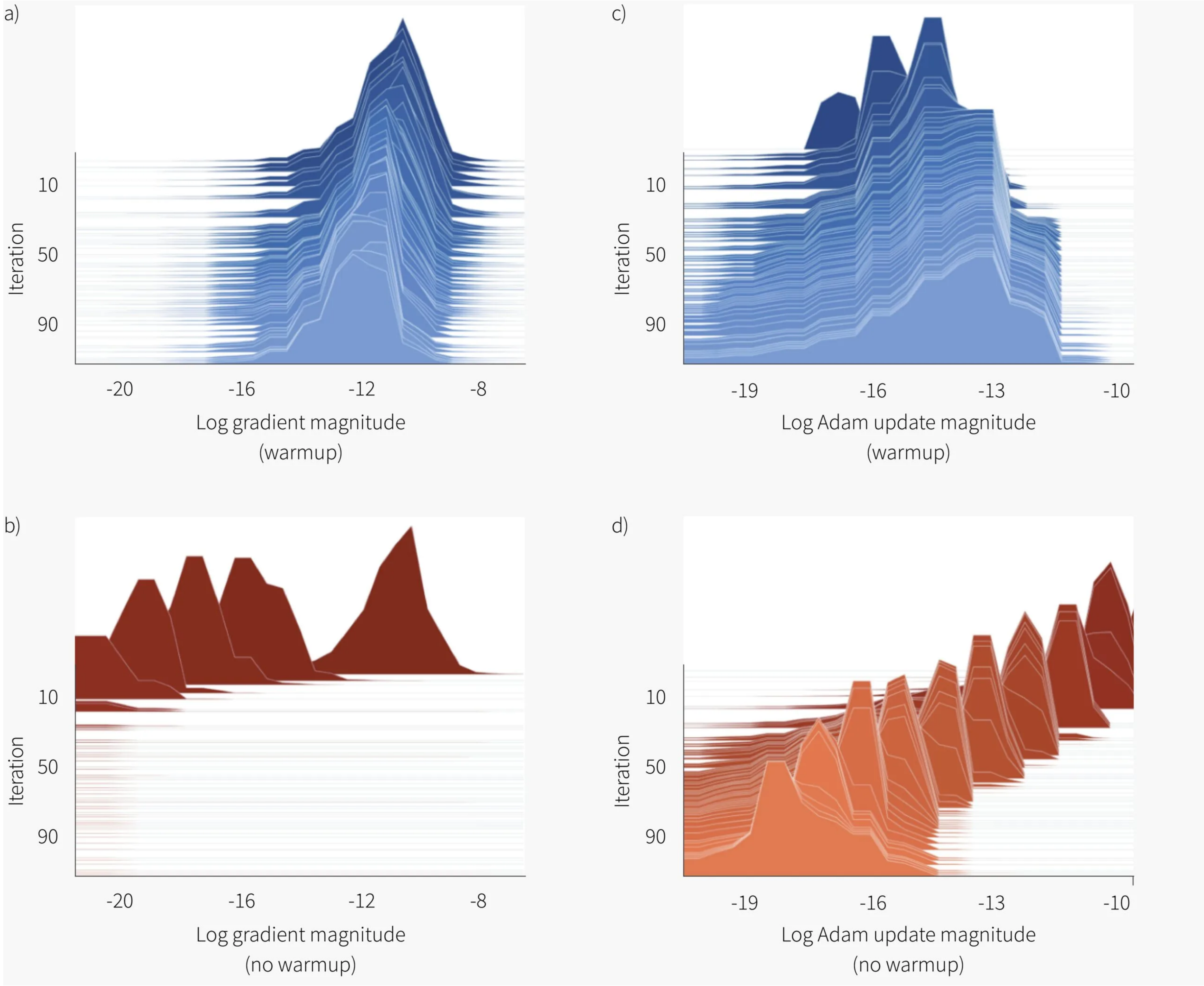


Learning Rate Warmup —> Breaks conventional machine learning wisdom?

< Ok but... why? >

Reference: "The Annotated Transformer," Sasha Rush. *Harvard NLP (2018)*

3 Years Later...



3.1. Problem in Transformer Optimization

In this section we demonstrate that the requirement for warmup comes from a combined effect of high variance in the Adam optimizer and backpropagation through layer normalization. Liu et al. (2020) showed that at the begin-

of the input. Specifically, the gradient has the following property:

$$\left\|\frac{\partial \mathrm{LN}(\boldsymbol{x})}{\partial \boldsymbol{x}}\right\| = O\left(\frac{\sqrt{d}}{||\boldsymbol{x}||}\right) \quad (1)$$

where $\boldsymbol{x}$ is the input to layer normalization and d is the embedding dimension. If input norm $||\boldsymbol{x}||$ is larger than $\sqrt{d}$ then backpropagation through layer normalization has a down scaling effect that reduces gradient magnitude for lower layers. Compounding across multiple layers this can quickly lead to gradient vanishing.

< Ok, now we're done...? >

Reference: "Transformers III Training; Tricks for Training Transformers," Borealis AI — 8/6/2021.
Reference: "Improving Transformer Optimization through Better Initialization," Huang et. al., *ICML 2020.*

The "Modern" Transformer (Mid-2023)

```python
class ModernTransformerBlock(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int, up: int = 4):
        super().__init__()
        self.attn = Attention(embed_dim, n_heads, qk_bias=False)
        self.mlp = nn.Sequential(
            SwishGLU(embed_dim, up * embed_dim),
            nn.Linear(up * embed_dim, embed_dim)
        )

        # Post-Norm --> *Pre-Norm*
        self.pre_attn_norm = RMSNorm(embed_dim)
        self.pre_mlp_norm = RMSNorm(embed_dim)

    def forward(self, x: T[bsz, seq, embed_dim]):
        x = x + self.attn(self.pre_attn_norm(x))
        x = x + self.mlp(self.pre_mlp_norm(x))
        return x
```

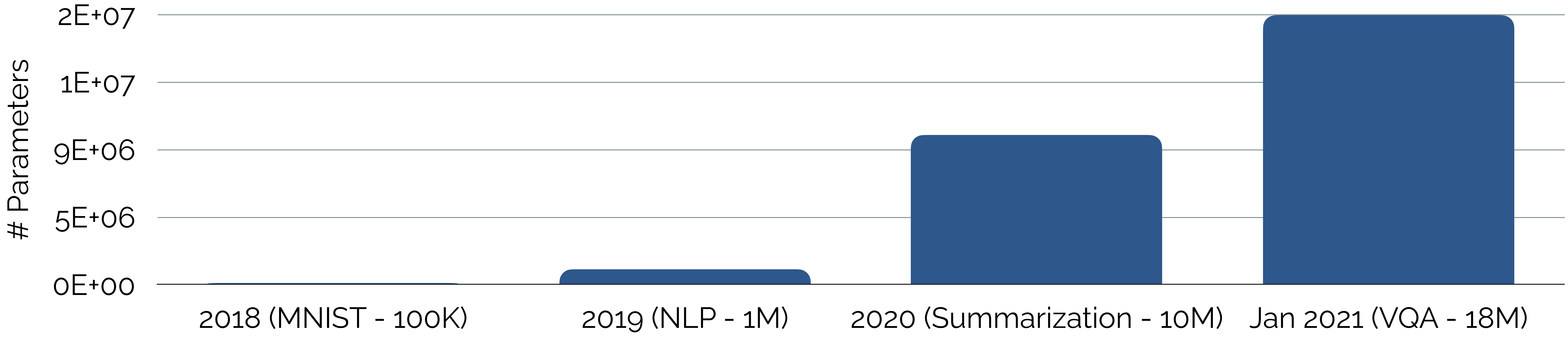
```python
# SwishGLU -- A Gated Linear Unit (GLU) with Swish Activation
class SwishGLU(nn.Module):
    def __init__(self, in_dim: int, out_dim: int):
        super().__init__()
        self.swish = nn.SiLU()
        self.project = nn.Linear(in_dim, 2 * out_dim)

    def forward(self, x: T[bsz, seq, embed_dim]):
        projected, gate = self.project(x).tensor_split(2, dim=-1)
        return projected * self.swish(gate)


# RMSNorm -- Simple Alternative to LayerNorm
class RMSNorm(nn.Module):
    def __init__(self, dim: int, eps: float = 1e-8):
        super().__init__()
        self.scale, self.eps = dim**-0.5, eps
        self.g = nn.Parameter(torch.ones(dim))

    def forward(self, x: T[bsz, seq, embed_dim]):
        norm = torch.norm(x, dim=-1, keepdim=True) * self.scale
        return x / norm.clamp(min=self.eps) * self.g
```

< The End? >

Part II. Training at Scale

“Nothing in life is to be feared. It is only to be understood.”
— Marie Curie

Short Story — My Deep Learning Trajectory



- "Standard Pipeline": Train on 1 GPU (e.g., on Colab) —> ~**max of a few hours.**

- Let's train a GPT-2 Small (124M)!
    - **Problem:** Batch > 4 goes OOM on a decent GPU = > ***12 GB*** of GPU RAM
    - Simple Trick —> *Gradient Accumulation!*
        - But… **99.63 Days** to train on Single GPU (400K Steps)

GPT-2 Training Clock

99.63 D

Shortening the Clock —> The Scaling Toolbox

GPT-2 Training Clock

99.63 D

Goal: 100 Days on 1 GPU —> ~**4 Days on 16 GPUs**

- **Data Parallelism** — Scaling *across* GPUs & Nodes
- **Mixed Precision** — Bits, Bytes, and TensorCores
- **ZeRO Redundancy** — Minimizing Memory Footprint

Later… Model Parallelism — Hardware Limitations — Software Optimization

*Even if you're not training big models… **understanding breeds innovation!***

Data Parallelism — A Toy Example

GPT-2 Training Clock

99.63 D

```
BATCH_SIZE = 128

class MLP(nn.Module):
    def __init__(
      self, n_classes: int = 10, mnist_dim: int = 784, hidden: int = 128
    ):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(mnist_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, n_classes)
        )

    def forward(self, x: T[bsz, mnist_dim]):
        return self.mlp(x)

# Main Code
dataloader = DataLoader(dataset=torchvision.datasets(...), batch_size=BATCH_SIZE)
model = MLP()

# Train Loop
criterion, opt = nn.CrossEntropyLoss(), optim.AdamW(model.parameters())
for (inputs, labels) in dataloader:
    loss = criterion(model(inputs), labels)
    loss.backward(); opt.step(); opt.zero_grad()
```

Idea —> Parallelize?

SIMD

Single Instruction, Multiple Data

SPMD

Single **Program**, Multiple Data

< Seems hard? >

(Distributed) Data Parallelism — Implementation

GPT-2 Training Clock

99.63 D

7.2 D — 16 GPUs w/ Data Parallelism (DDP)

```
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data.distributed import DistributedSampler

BATCH_SIZE, WORLD_SIZE = 128, 8  # World Size == # of GPUs
class MLP(nn.Module):
    def __init__(
      self, n_classes: int = 10, mnist_dim: int = 784, hidden: int = 128
    ):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(mnist_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, n_classes)
        )

    def forward(self, x: T[bsz, mnist_dim]):
        return self.mlp(x)

# Main Code
train_set = torchvision.dataset(...)
dist_sampler = DistributedSampler(dataset=train_set)
dataloader = DataLoader(
  train_set, sampler=dist_sampler, batch_size=BATCH_SIZE // WORLD_SIZE
)

model = DDP(
  MLP(),
  device_ids=[os.environ["LOCAL_RANK"]],
  output_device=os.environ["LOCAL_RANK"]
)
```

Auto-Partitions Data across Processes

Simple Wrapper around nn.Module()

```
# Train Loop
criterion, opt = nn.CrossEntropyLoss(), optim.AdamW(model.parameters())
for (inputs, labels) in dataloader:
    loss = criterion(model(inputs), labels)
    loss.backward(); opt.step(); opt.zero_grad()

# Run: `torchrun --nnodes 1 --nproc_per_node=8 main.py`
```

Nifty Utility —> Spawns Processes

Important — Memory Footprint of Training?

Standard (Float 32) Memory Footprint

[Excludes Activations + Temporary Buffers]

32b Parameters

32b Gradients

Model

32b Parameter Copies

~Adam

32b Momentum

32b Variance

Optimizer

Lower Bound on "Static" Memory (w/ Adam):

= **# Parameters * 20 Bytes**

Activation Memory >> Static Memory

Training Implications

- 1B Parameters —> 18 GB (~**31 GB w/ BSZ = 1**)
- 175B Parameters —> ***3 TB (w/o activations!)***

Facts about Floating Points

- Float32 — Standard defined in IEEE-754
  - Sign (1) — Exponent (8) — Significand (23)
  - Wide Range —> up to 1e38

< Do we need *all* 32 bits? >

Reference: "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," Rajbhandari, Rasley, Ruwase, and He. *SC 2020*.

Mixed Precision Training

GPT-2 Training Clock

99.63 D

7.2 D — 16 GPUs w/ Data Parallelism (DDP)

6.01 D — 16 GPUs w/ DDP, FP16

Mixed Precision (FP16) Memory Footprint

[Excludes Activations + Temporary Buffers]

Model
- **16b** Parameters
- **16b** Gradients

Optimizer
- 32b Parameter Copies
- **~Adam**
  - 32b Momentum
  - 32b Variance

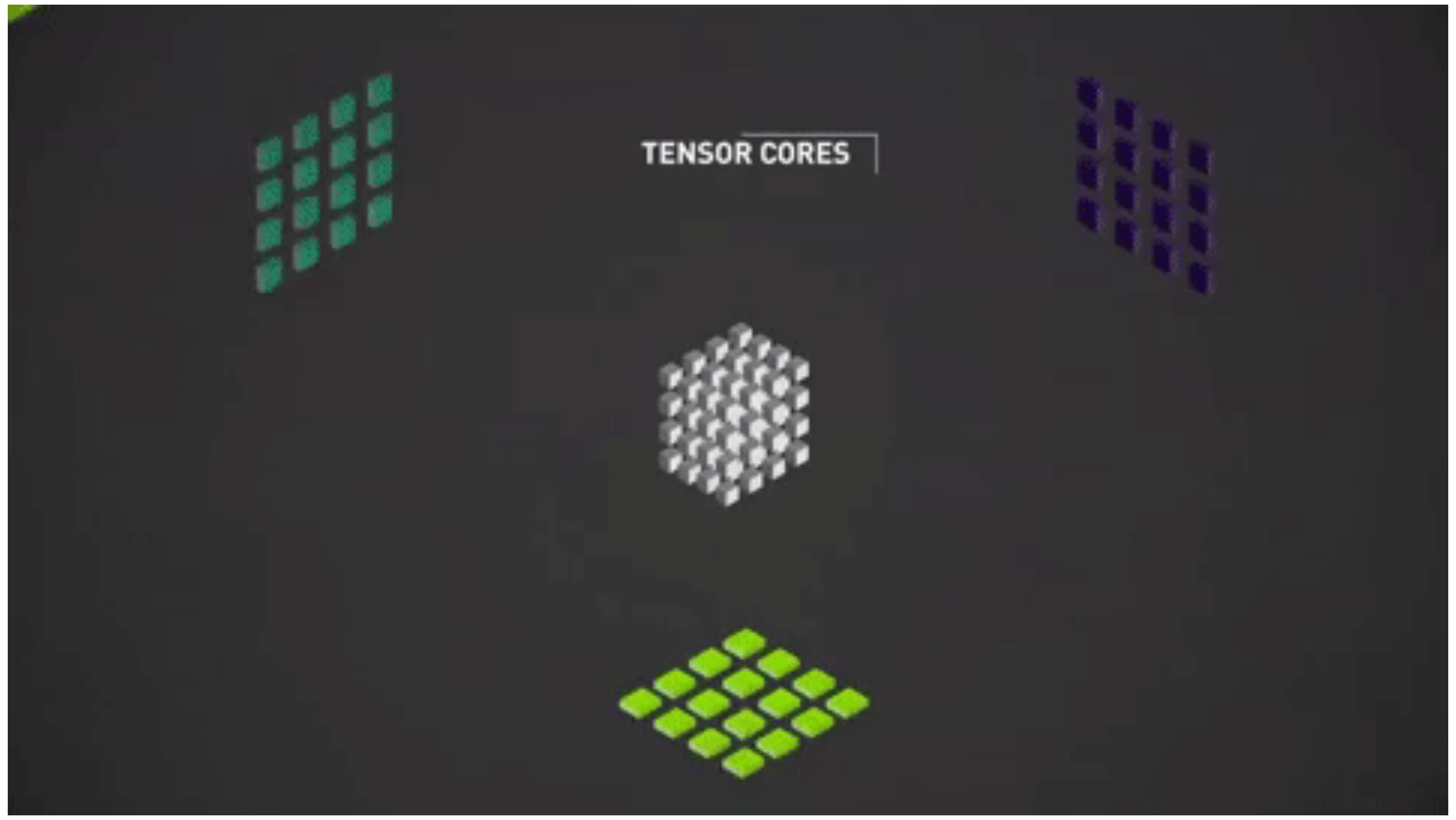
Lower Bound on "Static" Memory (w/ Adam):
= **# Parameters * 16 Bytes**

Activation Memory —> ***halved!***

Hmm… Optimizer Memory?

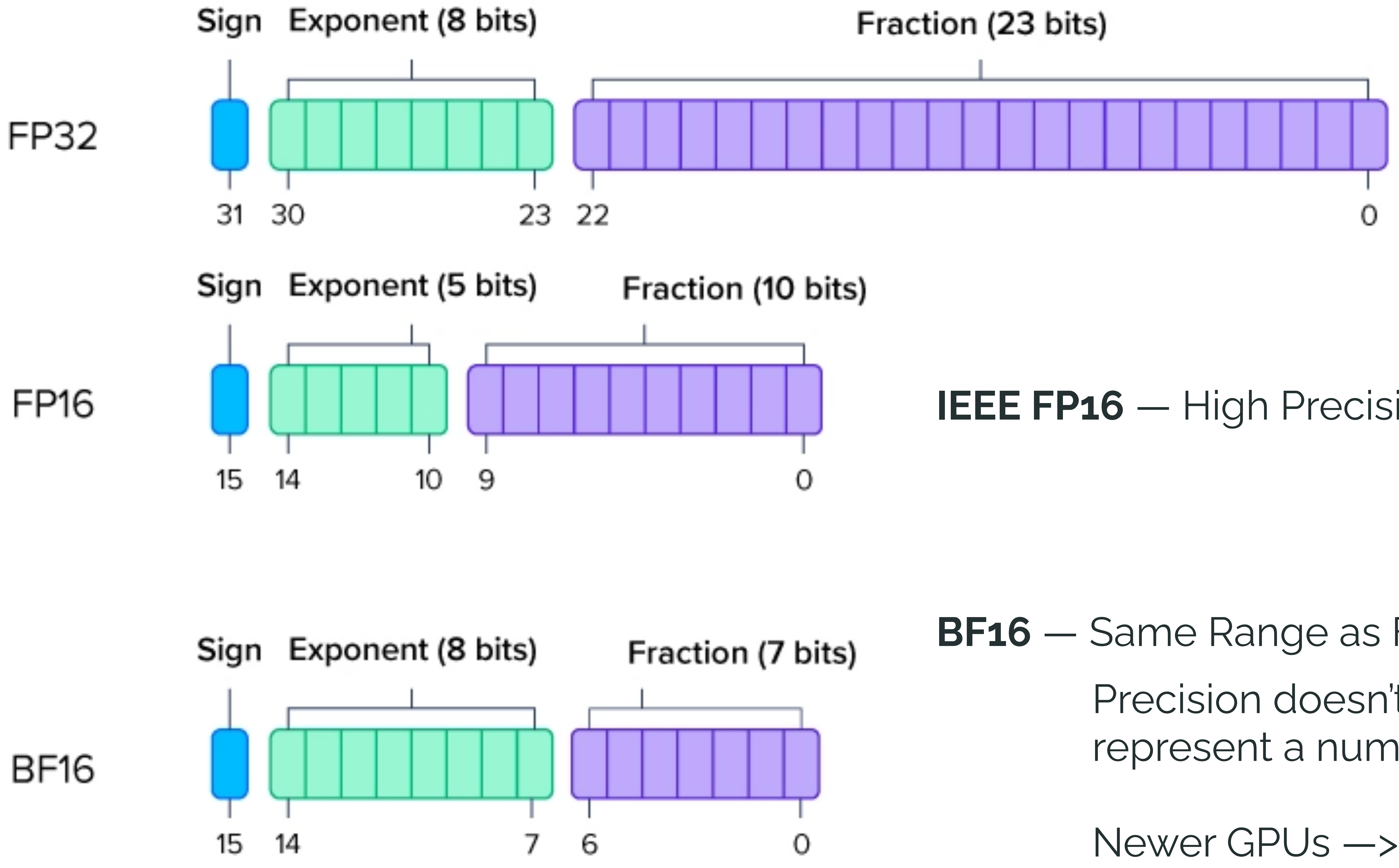
FP16 **does not mean *everything*** is FP16.

Real Gain: NVIDIA Tensor Core Speedup!



Reference: "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," Rajbhandari, Rasley, Ruwase, and He. *SC 2020*.

(Aside) Brain Float 16 (BF16) — A New Standard



IEEE FP16 — High Precision, **Limited Range**

BF16 — Same Range as FP32!

Precision doesn't matter if you can't represent a number!

Newer GPUs —> native support!

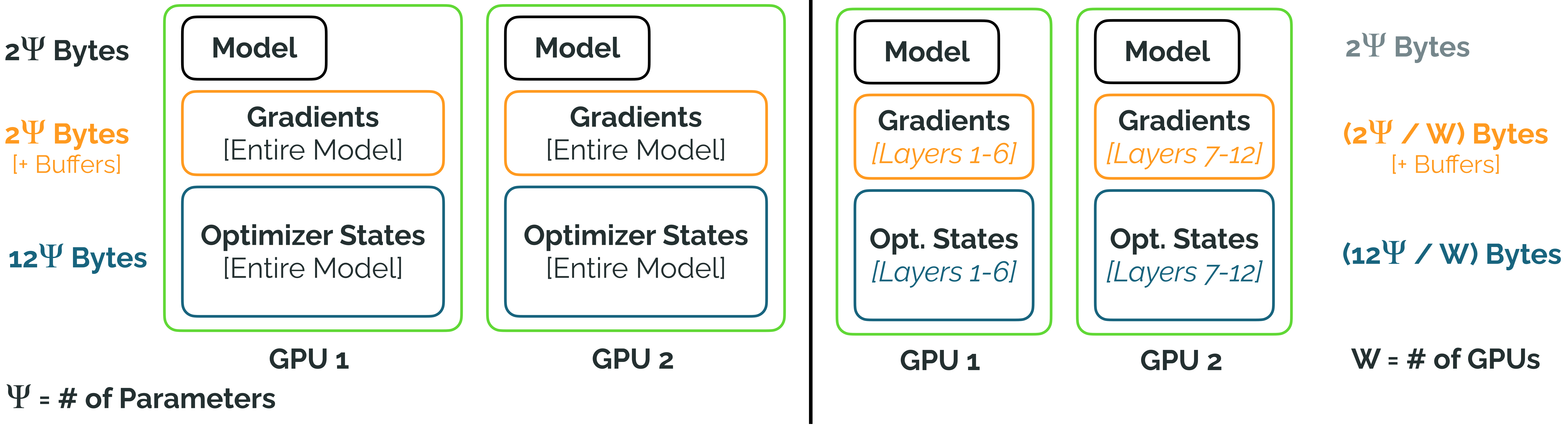
Eliminate Redundancies —> ZeRO

GPT-2 Training Clock

Punchline: "Shards" Memory by # of GPUs!

Standard Data Parallelism
"Replicate everything but the data!"

ZeRO Data Parallelism (ZeRO-2)
"Replicate only what you need"



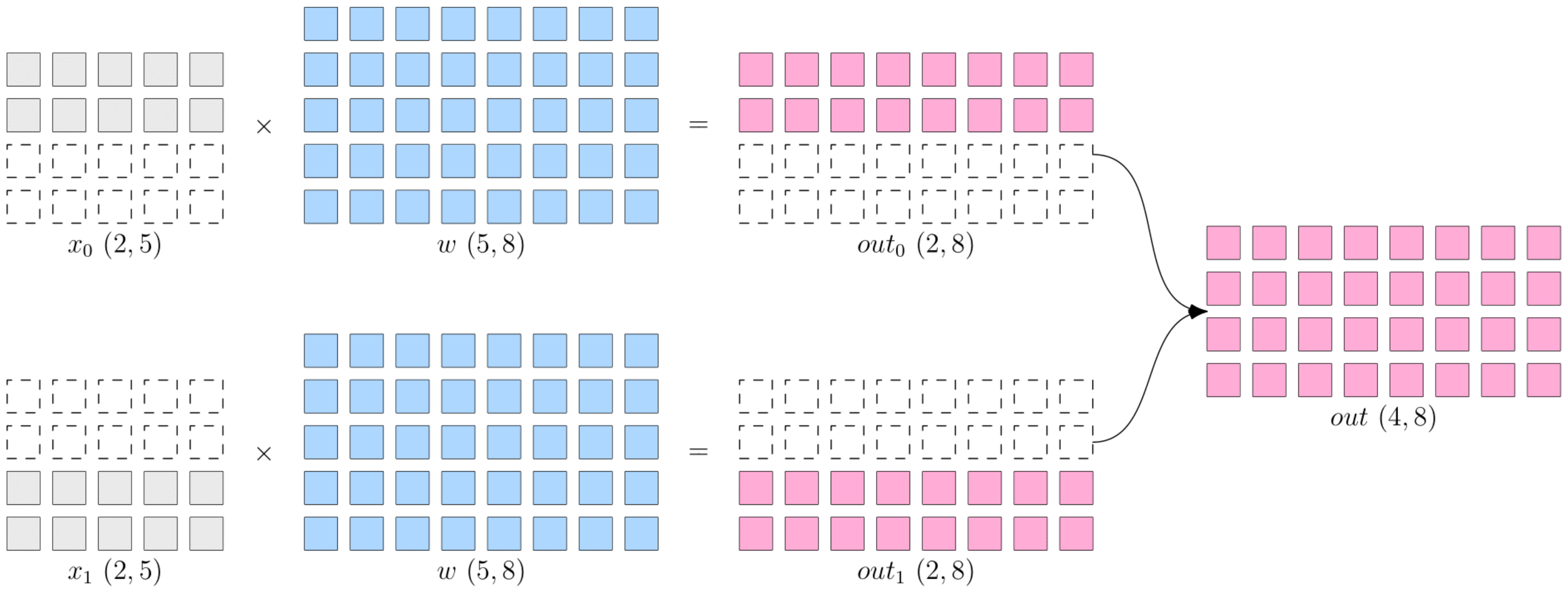
Reference: "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," Rajbhandari, Rasley, Ruwase, and He. *SC 2020*.

Alas — Hitting a (Communication) Wall

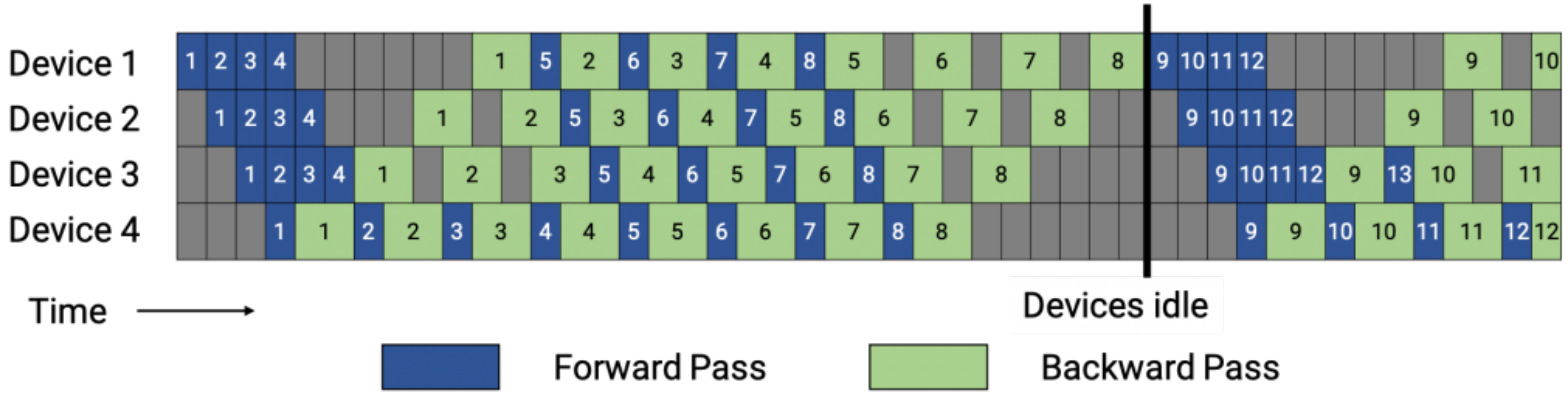
Problem — At some point, communication cost between nodes is too much!

Answers:

Exploit Matrix Multiplication…



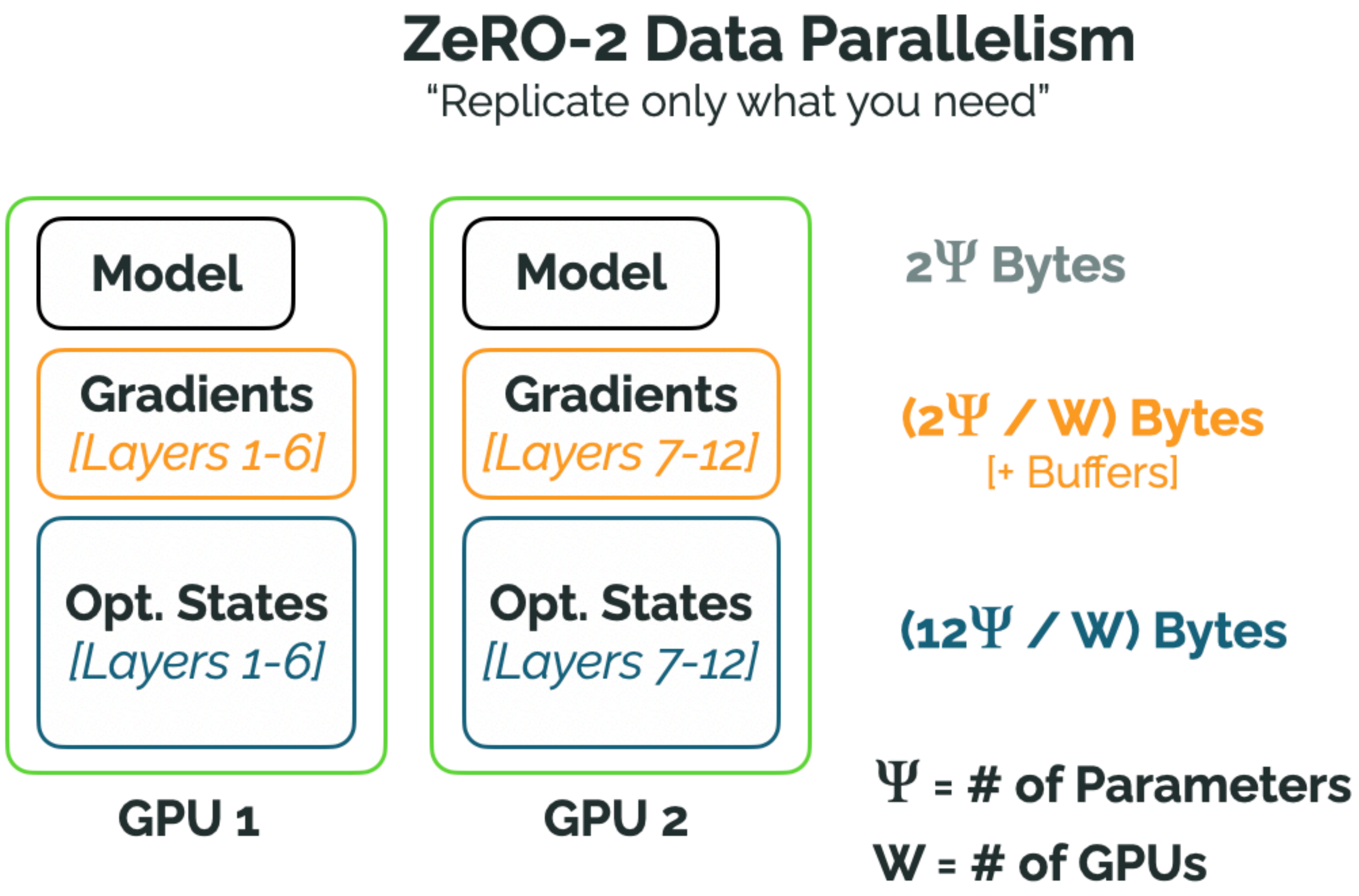
Schedule Backwards Pass Wisely…



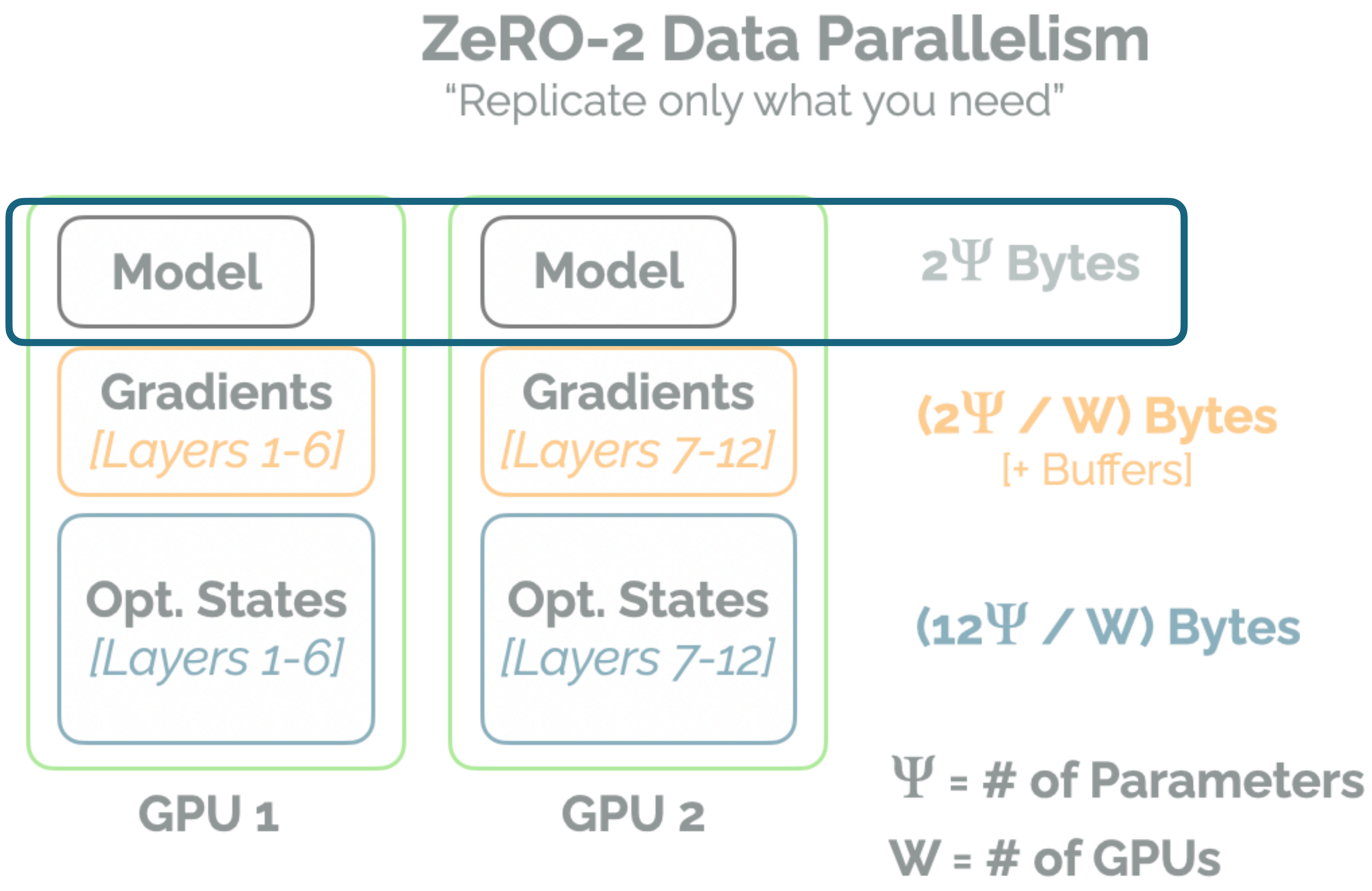
< Harder to implement, model-specific… miles to go? >

Reference: "PipeDream: Generalized Pipeline Parallelism for DNN Training," Narayanan et. al. *SOSP 2019*.
Reference: "Efficient Large-Scale LM Training on GPU Clusters using Megatron-LM," Narayanan et. al. *SC 2021*.

2024/2025 Update — Fully-Sharded Data Parallel (FSDP)



2024/2025 Update — Fully-Sharded Data Parallel (FSDP)



ZeRO-3 / FSDP — Shard model weights too!

Adds Latency — need to "gather" per GPU!

But — can be smart about "grouping" parameters!

As of PyTorch 2.2 (Stable)

```
from torch.distributed.fsdp import fully_shard

# "LLM" = [Block1 -> Block2 -> ... BlockN]
model = LLM()

# Shard the Model
for m in model.modules():
  # Group each Block Together!
  if isinstance(m, TransformerBlock):
    fully_shard(m)

# Group remaining parameters...
fully_shard(model)

# Same DataLoader Setup as DDP...

# Run: `torchrun ... main.py`
```

Part III. Fine-Tuning and Inference

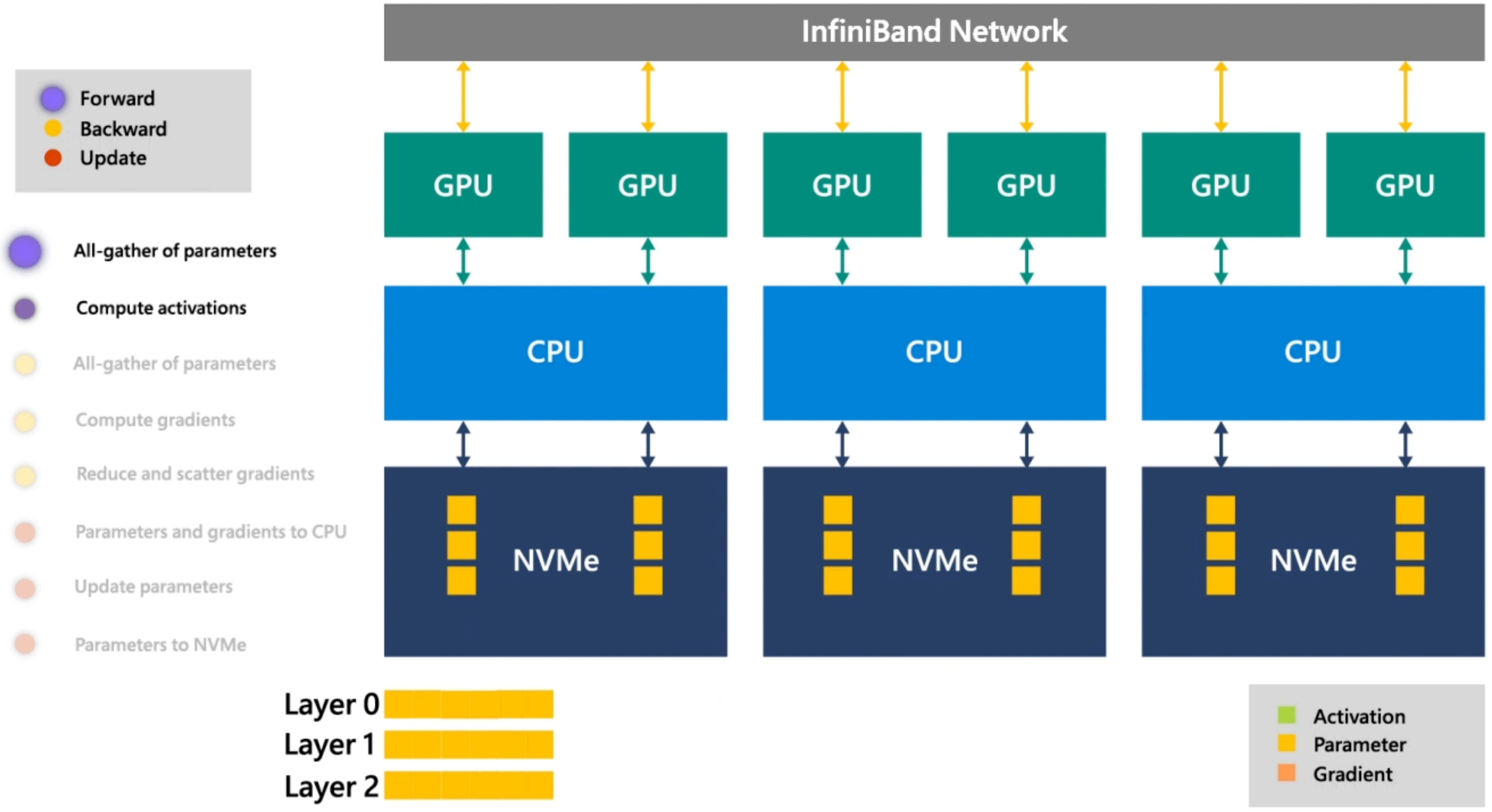
“It’s such a happiness, when good people get together.”
— Jane Austen, *Emma*

Tools for Training —> Tools for Fine-Tuning

Silver Lining — Learning to scale training —> informs *fine-tuning & inference!*

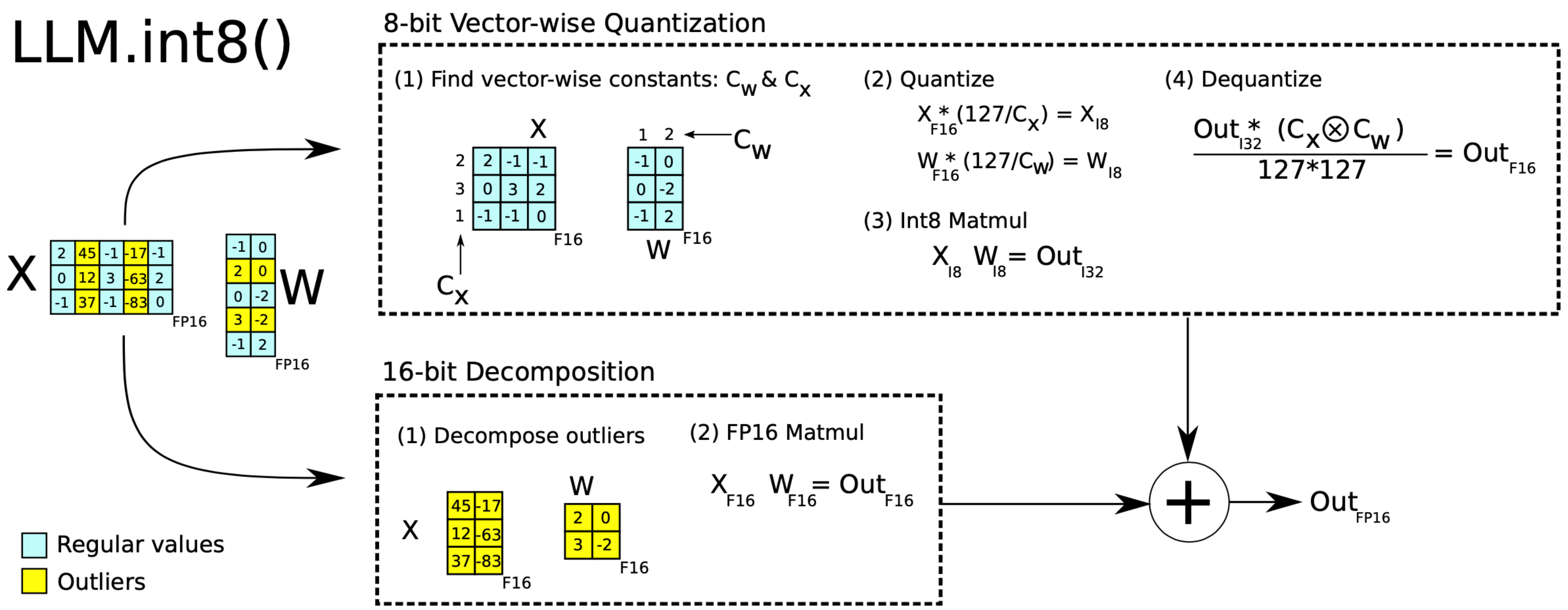
ZeRO Parallelism

ZeRO Infinity —> CPU/NVMe Offloading



Mixed Precision (BF16)

Quantization (8-Bit, 4-Bit)

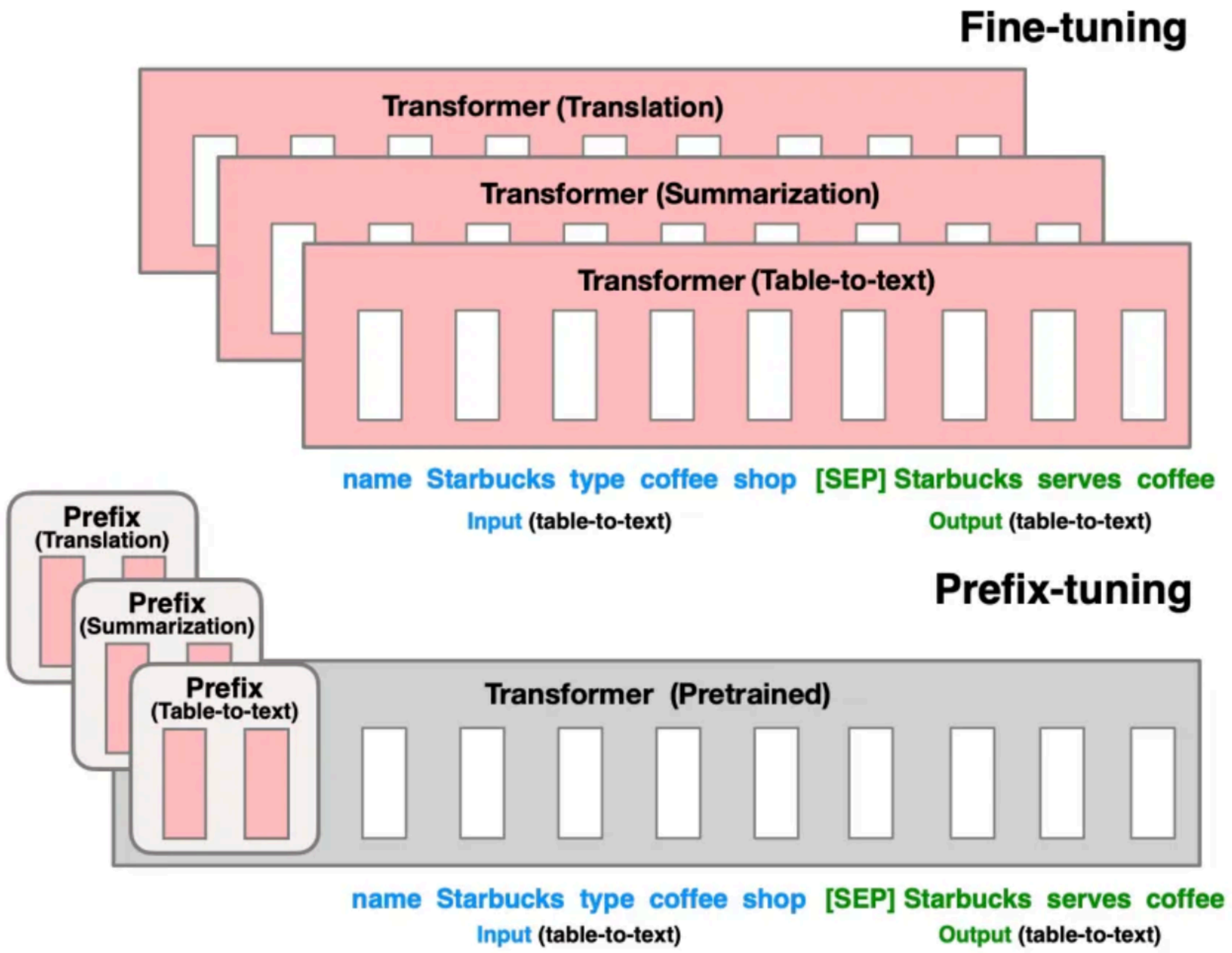


Powers `llama.cpp` and more!

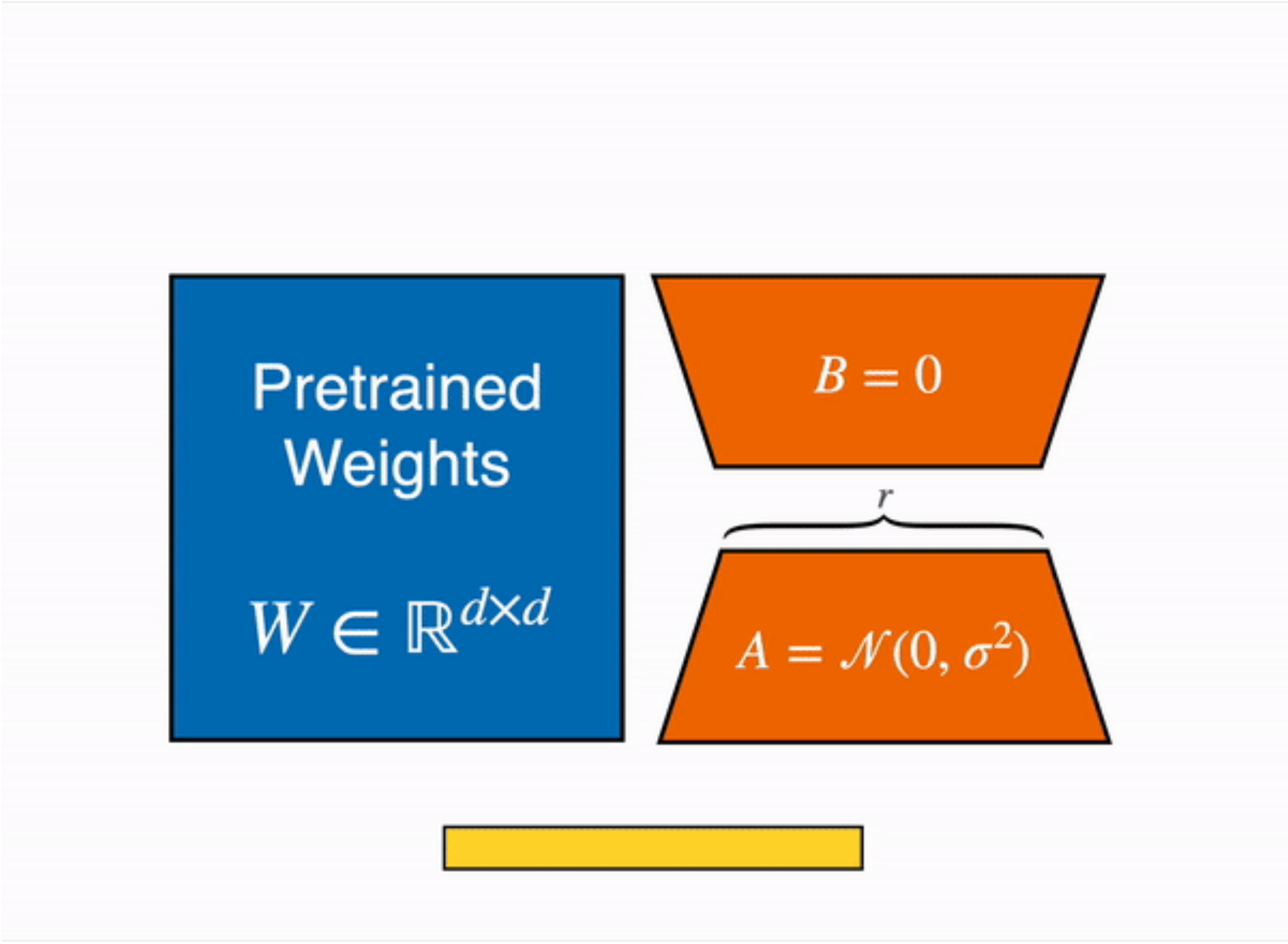
Reference: "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning" Rajbhandari et. al. *SC 2021.*
Reference: "LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale," Dettmers, Lewis, Belkada, and Zettlemoyer. *NeurIPS 2022.*

(Briefly) Parameter-Efficient Fine-Tuning

Prefix Tuning



(Quantized) LoRA (Low-Rank Adaptation)



...and many more!

Reference: https://github.com/huggingface/peft
Reference: Prefix-Tuning: Optimizing Continuous Prompts for Generation, Li and Liang, *ACL 2021.*

Optimizing LLM Inference (Beyond the KV Cache)

Kernel Fusion (Simplified)

`H = ReLU(XW) + bias`

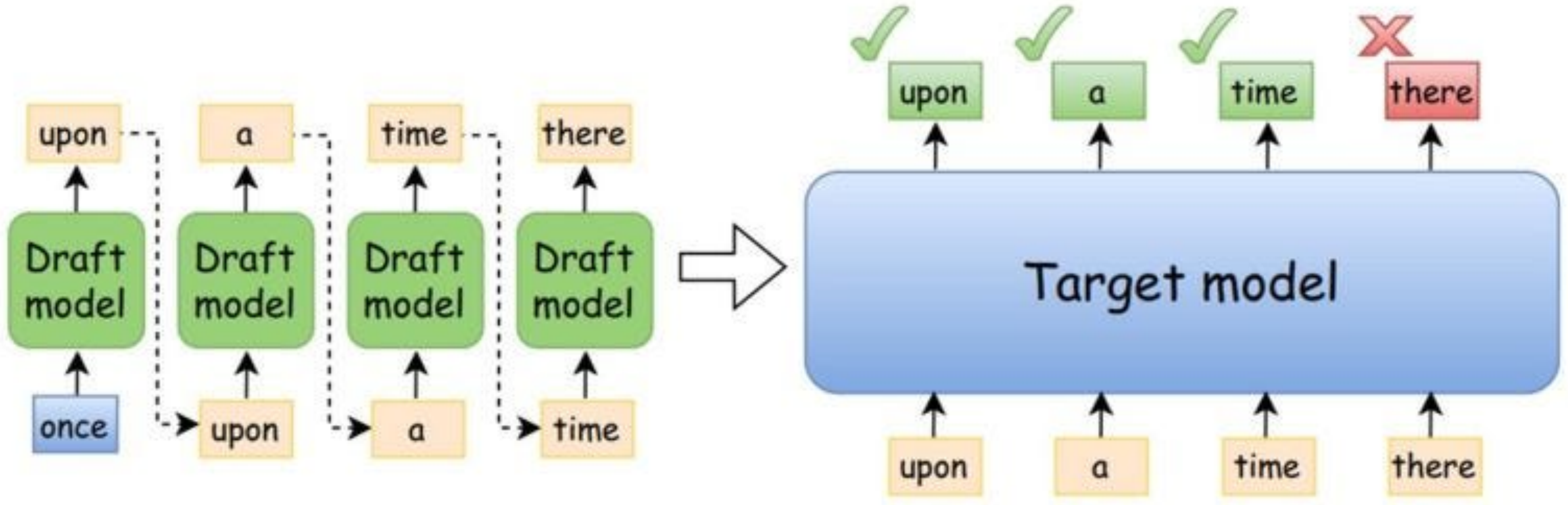
```
# Pass 1: Matrix Multiplication
for i in range(B):
  for j in range(M):
    accumulate = 0
    for k in range(K):
      accumulate += X[i][k] * W[k][j]
    H[i][j] = accumulate

# Pass 2: Add Bias
for i in range(B):
  for j in range(M):
    H[i][j] = H[i][j] + bias[j]

# Pass 3: ReLU
for i in range(B):
  for j in range(M):
    H[i][j] = max(0, H[i][j])
```

VS.

```
# Fuse Operations
for i in range(B):
  for j in range(M):
    accumulate = 0
    for k in range(K):
      accumulate += X[i][k] * W[k][j]
    accumulate = accumulate + bias[j]
    H[i][j] = max(0, accumulate)
```

(Latency) Speculative Decoding

Idea: “Draft” LLM to generate candidates (“guess”).
“Actual” LLM verifies in parallel (“check”).



Other Resources

Google’s Scaling Book:

https://jax-ml.github.io/scaling-book/

Sasha Rush’s GPU Puzzles (+ LLM Training Puzzles)

https://github.com/srush/gpu-puzzles

Modded NanoGPT:

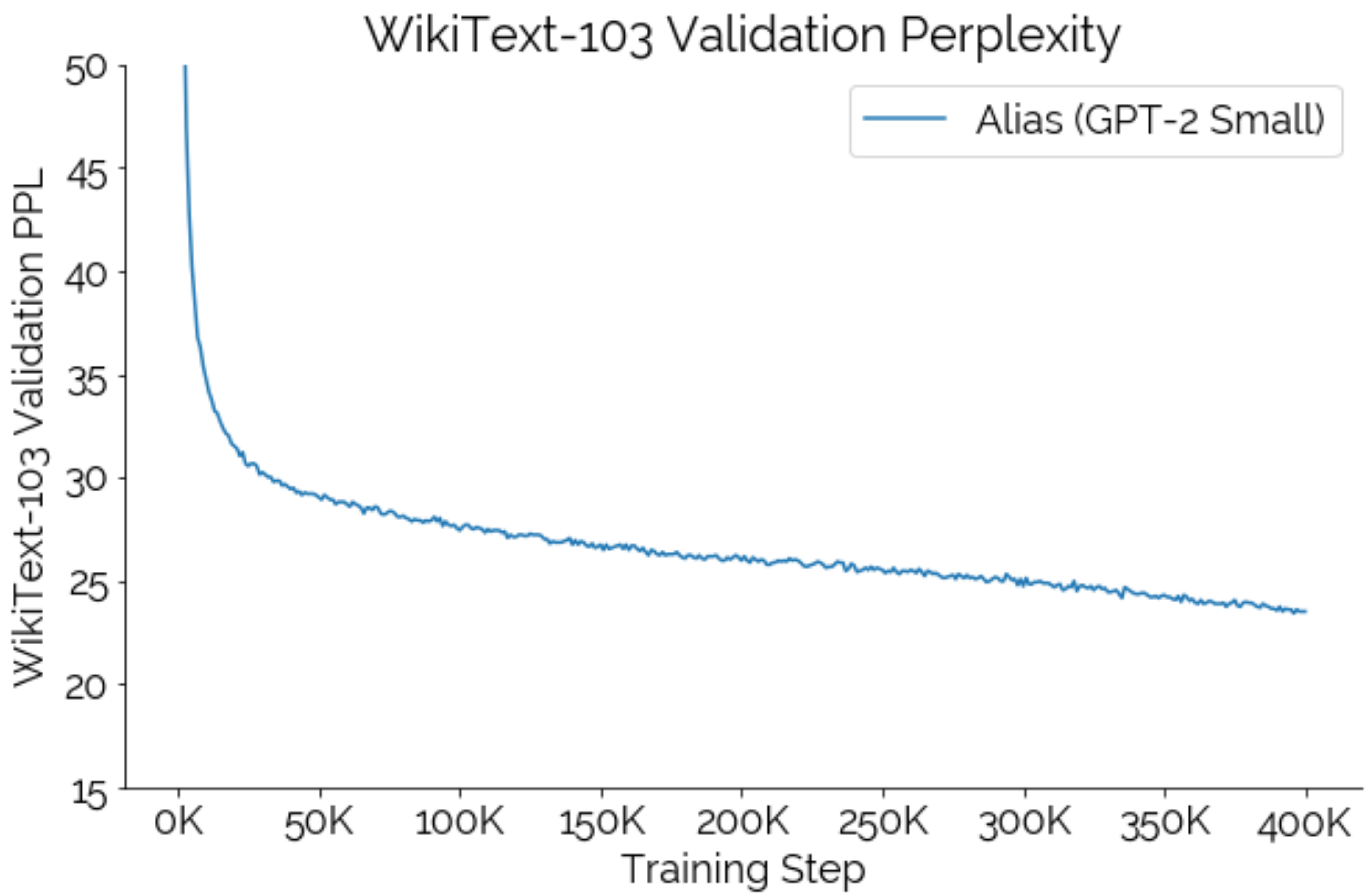
https://github.com/KellerJordan/modded-nanogpt

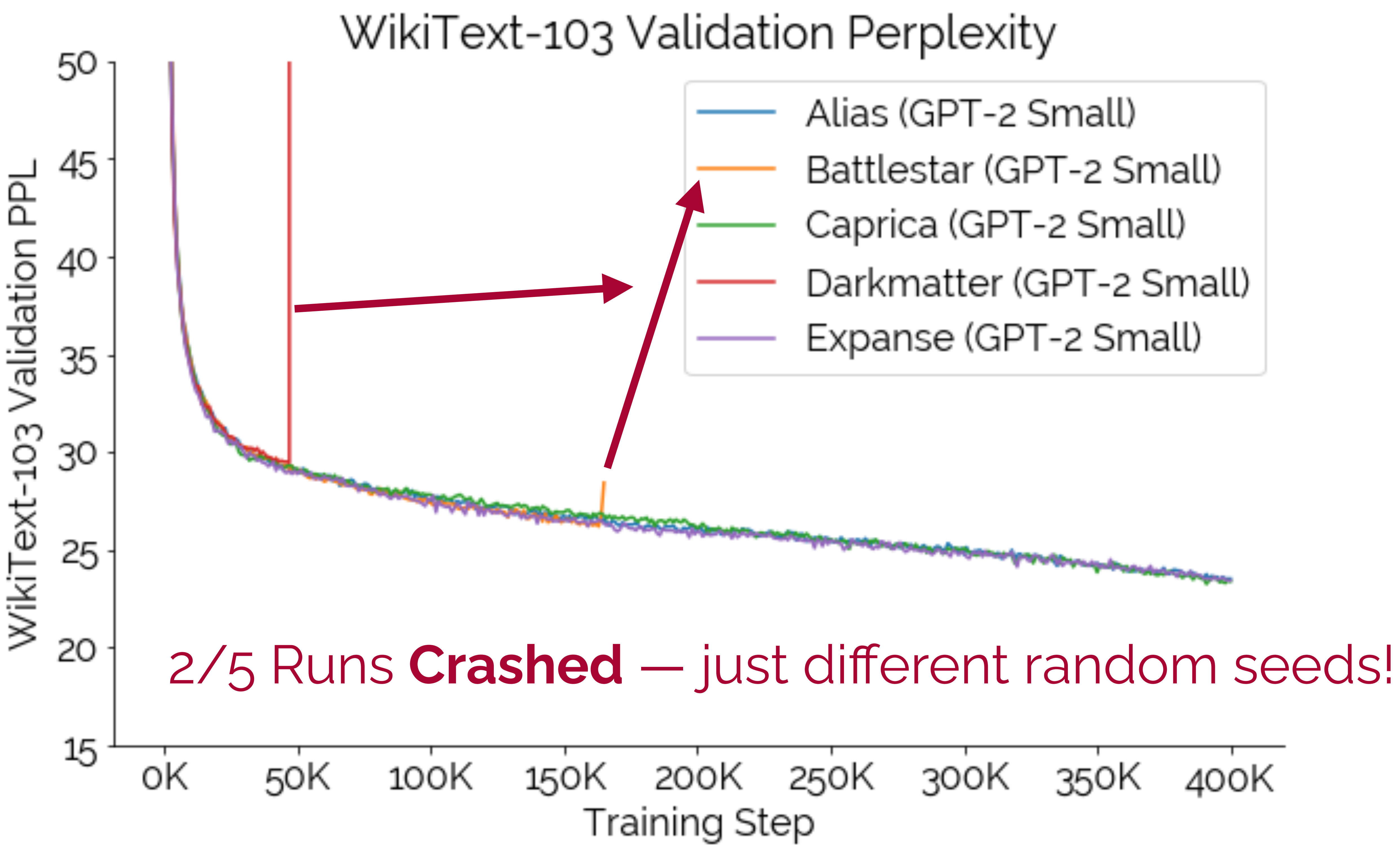
Coda. Curiosities & Lamentations

“In a world of diminishing mystery, the unknown persists.”
— Jhumpa Lahiri, *The Lowland*

Lamenting the Stability of Transformer Training

Reproducibility — what does reliability mean for LLM training (FP16 + ZeRO)?





What's going on?

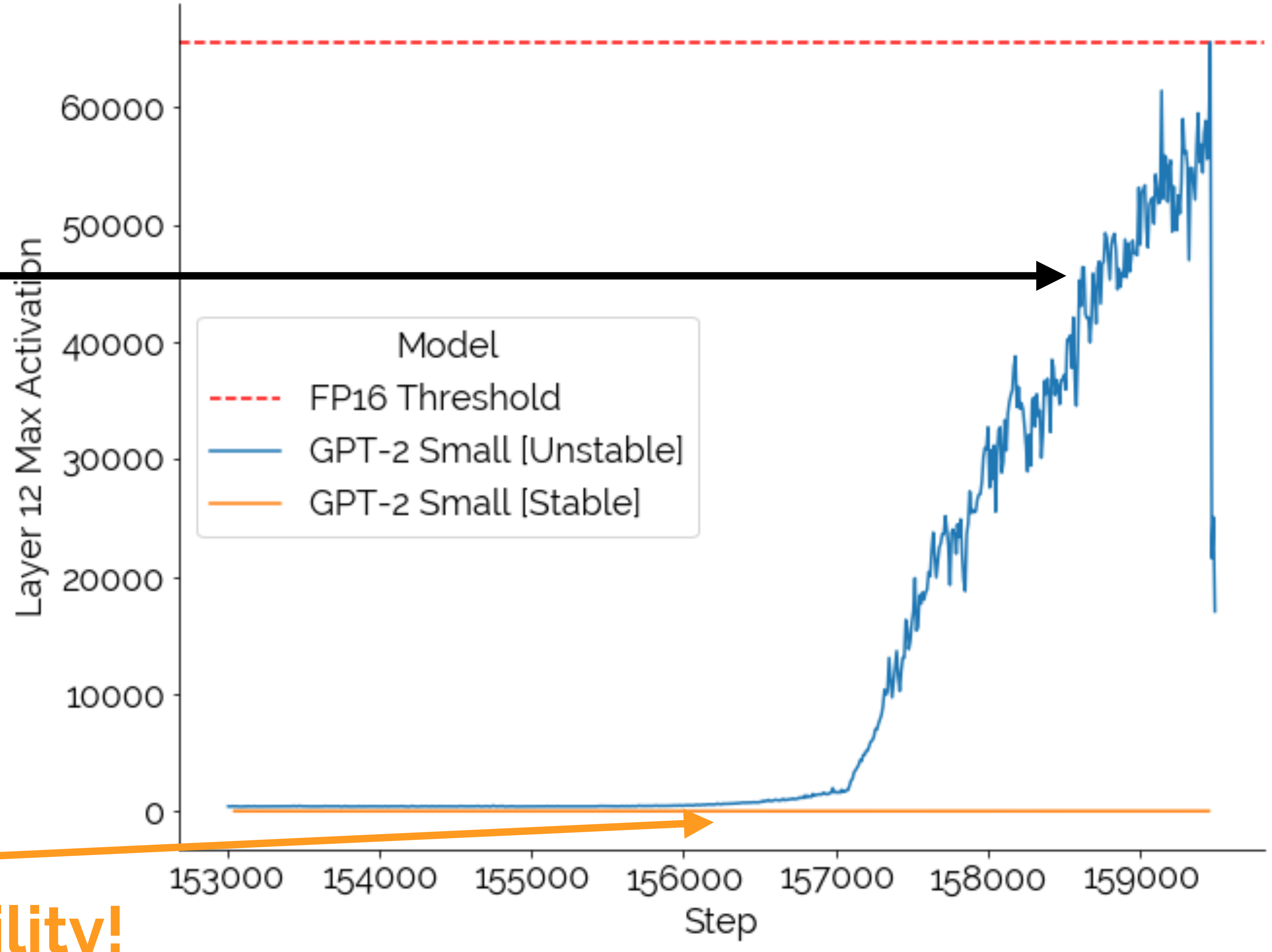
Zooming In — Order of Operations in Self-Attention

Recall: Scaled Dot-Product Attention → $\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$

Implementations…

```
def attn(query, key):
  d_k = query.size(-1)
  numerator = torch.matmul(query, k.T)
  denominator = math.sqrt(d_k)
  return numerator / denominator
```

```
def attn_stable(query, key):
  d_k = query.size(-1)
  key_stable = k.T / math.sqrt(d_k)
  return torch.matmul(query, key_stable)
```



Stability!

On “Folk Knowledge” & Implementation Details

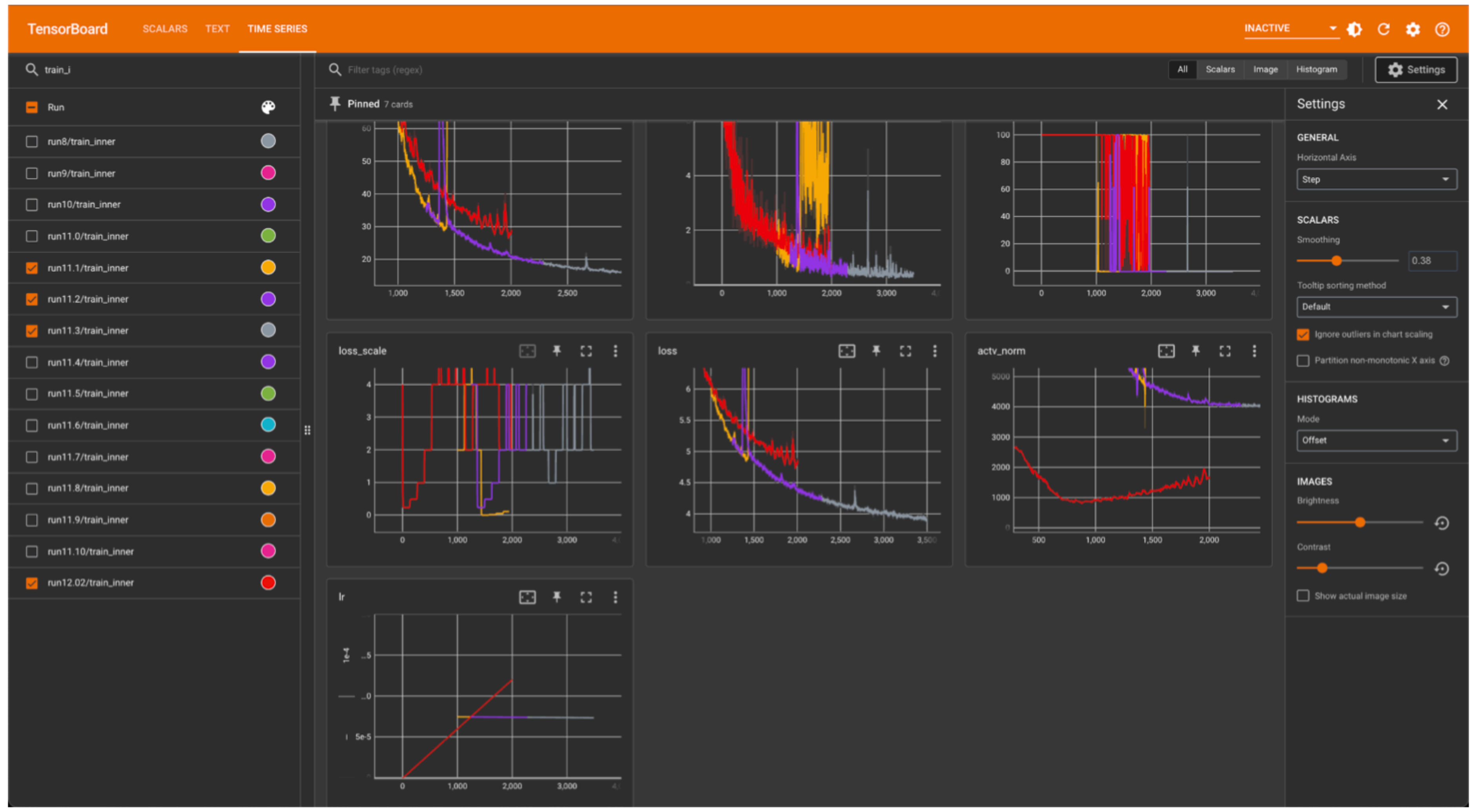
EleutherAI / **gpt-neo**

An implementation of model & data parallel GPT3-like models using the mesh-tensorflow library.

mesh / mesh_tensorflow / transformer / attention.py /

```
def attention(q,
              k,
    if float32_logits:
      k = mtf.cast(k, tf.float32)
      q = mtf.cast(q, tf.float32)
              dropout_rate=0.0,
              dropout_broadcast_dims=None,
              extra_logit=None,
              context=None,
              float32_logits=True,
              z_loss_coeff=None):
```

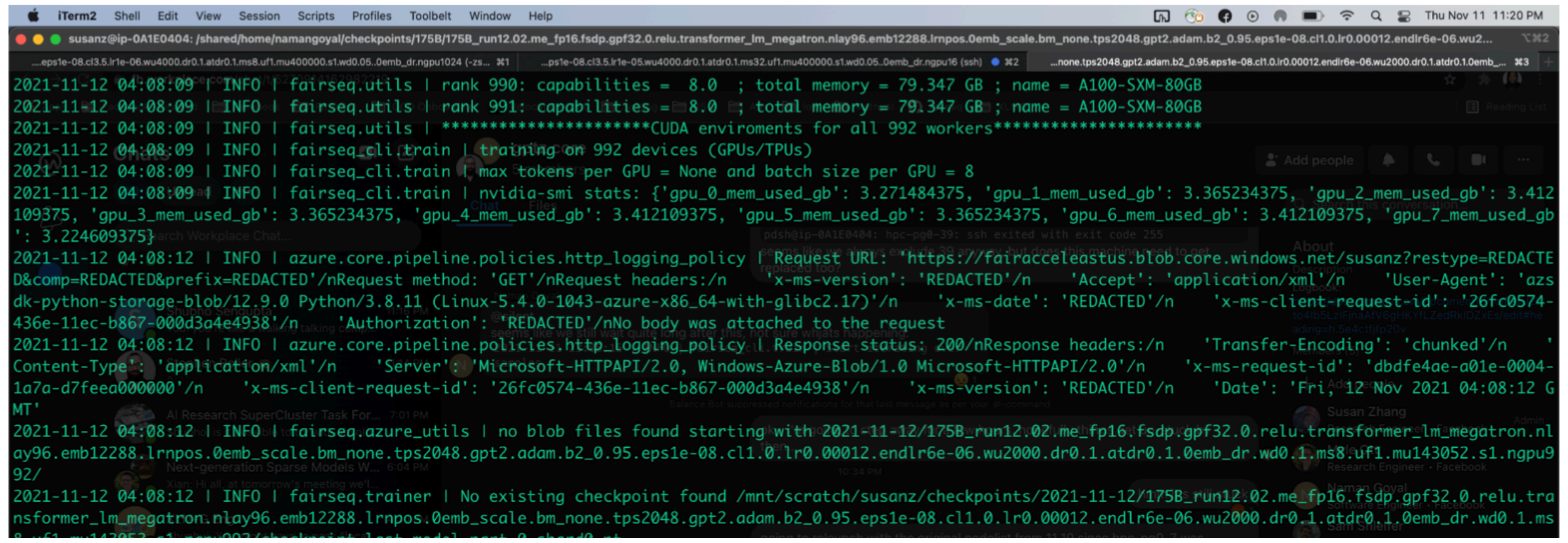
Analysis of Run 12.02 [Susan]



- Loss of ppl starting to oscillate more heavily, potentially indicating that LR is too high.
- CUDA error crashed the run after 2008 updates.

2021-11-11 11pm [Susan]: Run 12.02

- Relaunched with node 7 put back, same nodelist as 11.10.
- This worked! Expect roughly 2 minutes between “Start iterating over samples” and the first log line:



[1] “GPT-Neo Codebase” EleutherAI, 2021 (https://github.com/EleutherAI/gpt-neo).
[2] “OPT Logbook for OPT-175B,” Meta AI 2022 (https://github.com/facebookresearch/metaseq/tree/main/projects/OPT/chronicles)

2025 —> Details STILL Matter!

transformers / src / transformers / models / llama / modeling_llama.py

Code | Blame 532 lines (444 loc) · 21.3 KB

```python
def eager_attention_forward(
        module: nn.Module,
        query: torch.Tensor,
        key: torch.Tensor,
        value: torch.Tensor,
        attention_mask: Optional[torch.Tensor],
        scaling: float,
        dropout: float = 0.0,
        **kwargs: Unpack[TransformersKwargs],
    ):
        key_states = repeat_kv(key, module.num_key_value_groups)
        value_states = repeat_kv(value, module.num_key_value_groups)

        attn_weights = torch.matmul(query, key_states.transpose(2, 3)) * scaling
        if attention_mask is not None:
            causal_mask = attention_mask[:, :, :, : key_states.shape[-2]]
            attn_weights = attn_weights + causal_mask

        attn_weights = nn.functional.softmax(attn_weights, dim=-1, dtype=torch.float32).to(query.dtype)
        attn_weights = nn.functional.dropout(attn_weights, p=dropout, training=module.training)
        attn_output = torch.matmul(attn_weights, value_states)
        attn_output = attn_output.transpose(1, 2).contiguous()

        return attn_output, attn_weights
```

That’s all Folks!

“This wind, it is not an ending.”
— Robert Jordan, *A Memory of Light*