Parameter-Efficient Fine-Tuning

fzeng

04/30/2024

Fine-Tuning

- Copy weights from pre-trained network
- Perform training on downstream task of interest
- Learn new set of weights
- Naively: use the same architecture and all weights updated
  - For over-parameterized networks like LLM, requires a lot of data to converge
  - Large memory footprint to train all parameters

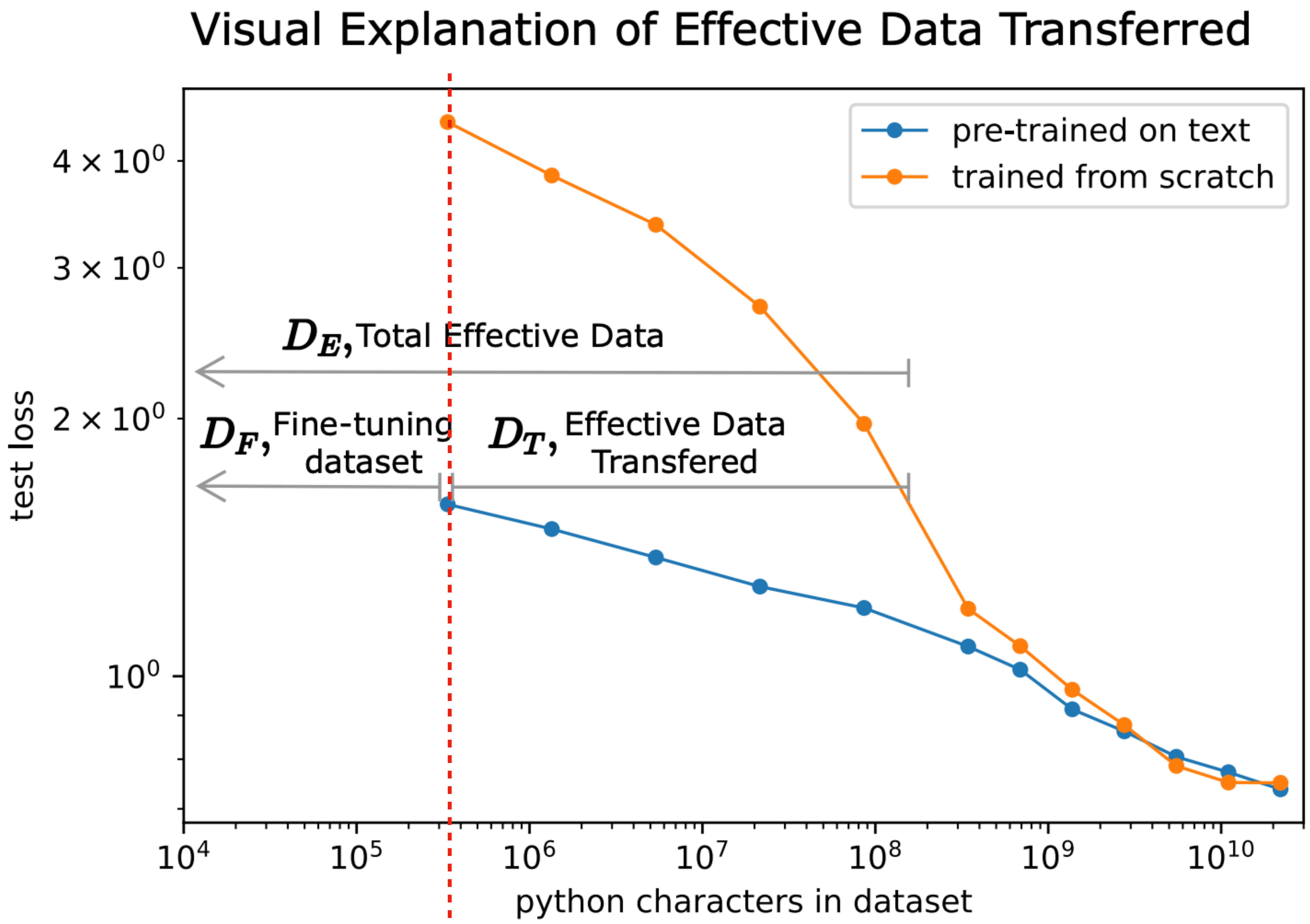
Outline

- Why is fine-tuning necessary?
- Adapter methods (LoRA)
- Quantization
- Prefix Tuning

Scaling Laws for Transfer

Why Fine-Tune At All?

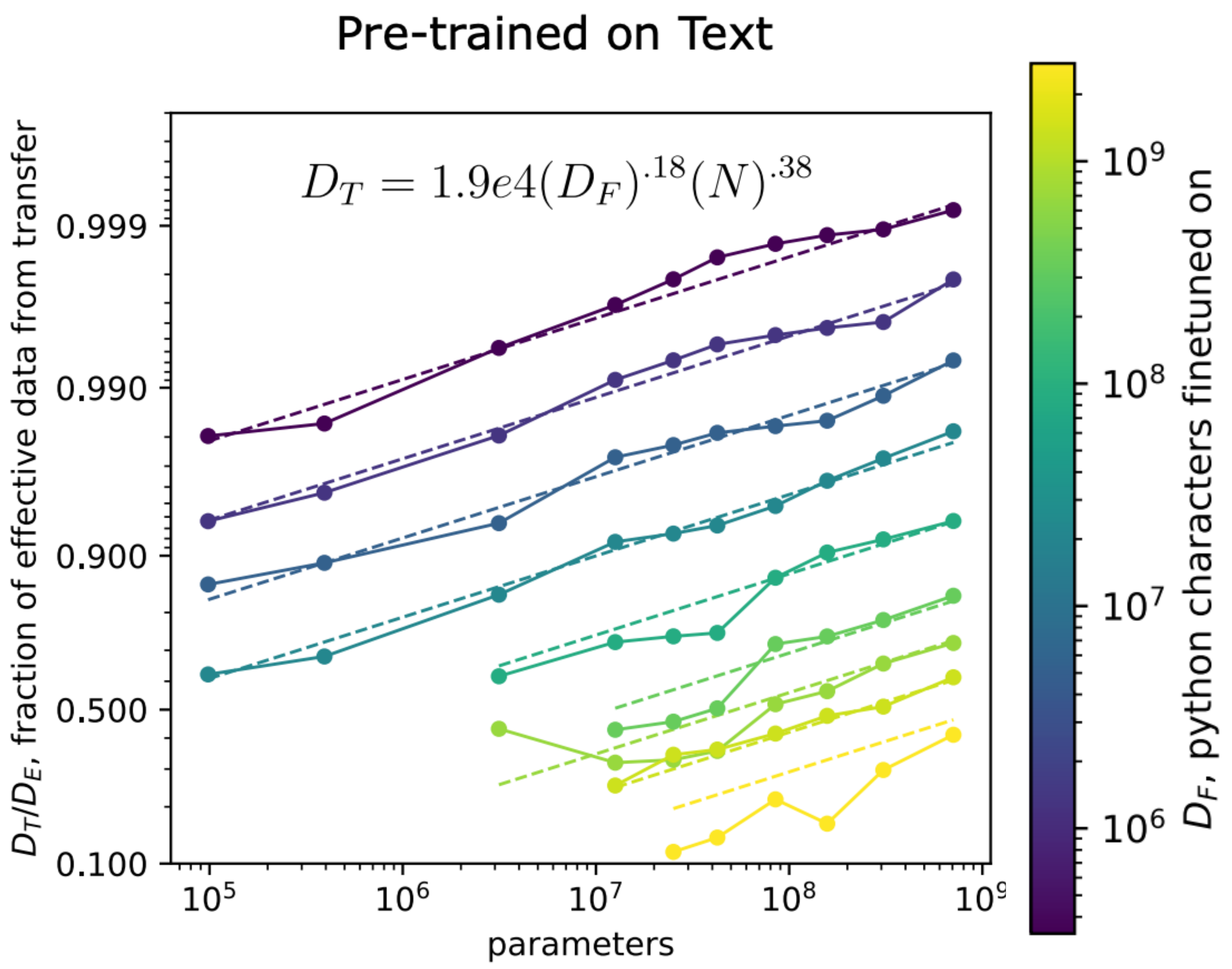
- Models pre-trained on large datasets have acquired good representations
- Important in low-data regime
- Right: 40M Transformer model, pre-training dataset 24b characters
- At 3e5 chars, fine-tuning performs as well as training from scratch with 1000x more data



Scaling Laws for Transfer (Hernandez et al, 2021)

Scaling Laws for Transfer in the Low-Data Regime

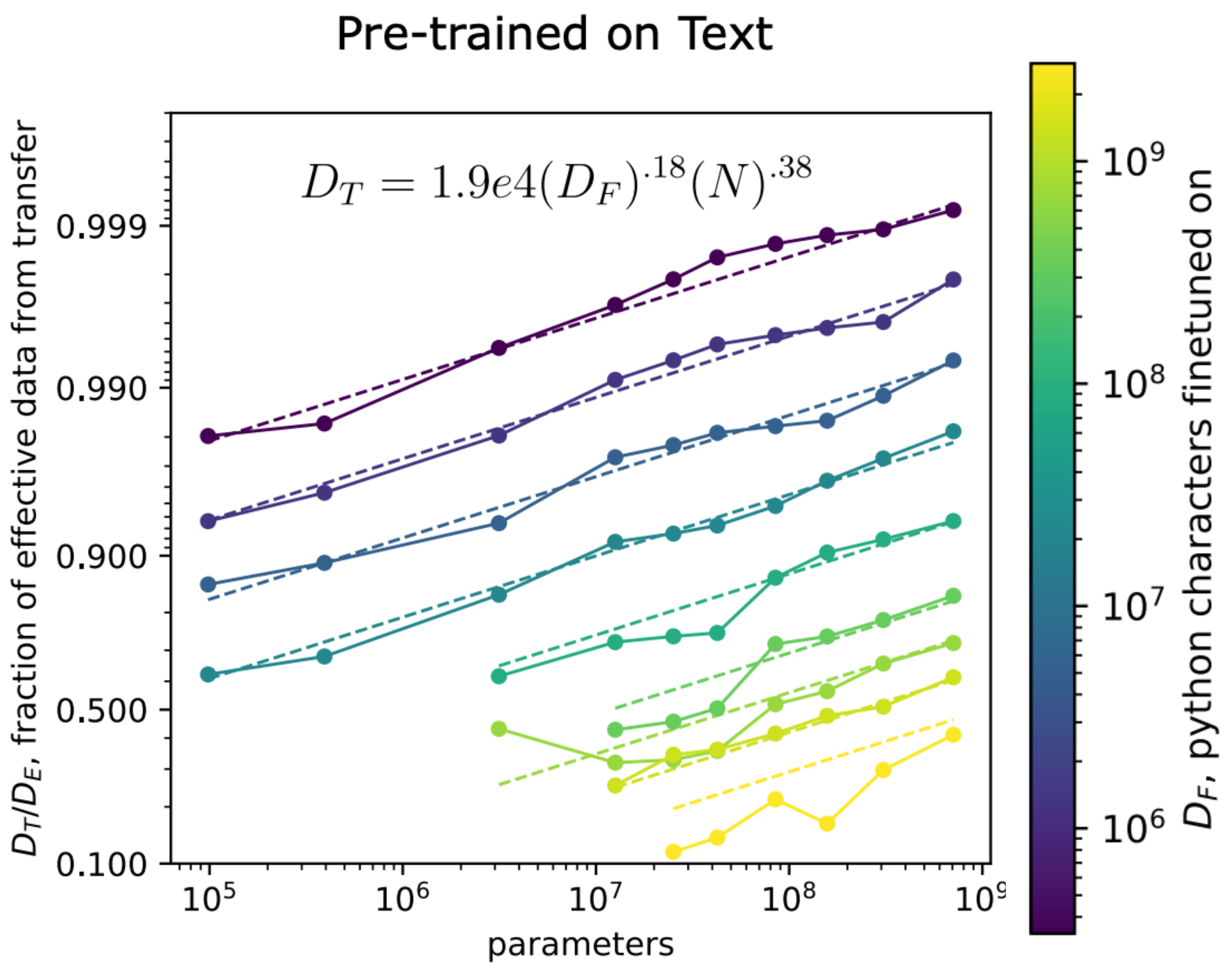
- Data transfer also follows a power-law (similar to neural scaling laws)
- D_T = effective data transferred $= k(D_F)^{\alpha}(N)^{\beta}$
- k: transfer multiplier
- D_F: size of fine-tuning distribution
- N: number of non-embedding parameters



Scaling Laws for Transfer (Hernandez et al, 2021)

Scaling Laws for Transfer in the Low-Data Regime

- D_T = effective data transferred $= k(D_F)^{\alpha}(N)^{\beta}$
- For fine-tuning on Python on a model pre-trained on text, $\beta \approx 2\alpha$
- Increasing fine-tuning dataset by 100x gives same improvement as increasing size of model by 10x



Scaling Laws for Transfer (Hernandez et al, 2021)

Scaling Laws for Transfer

	Transfer Coefficients		
Transfer from	k	α	β
Text $\Longrightarrow$ Python	1.9e4	0.18	0.38
50% Text and 50% non-python code $\Longrightarrow$ Python	2.1e5	0.096	0.38

- α: measures similarity between pre-training and fine-tuning distribution (smaller for closer similarity)
  - Smaller α means less transfer in the high-data regime
- Can conduct experiments to get α, β to understand trade-off between more data or larger model size

Scaling Laws for Transfer (Hernandez et al, 2021)

Scaling Laws for Transfer

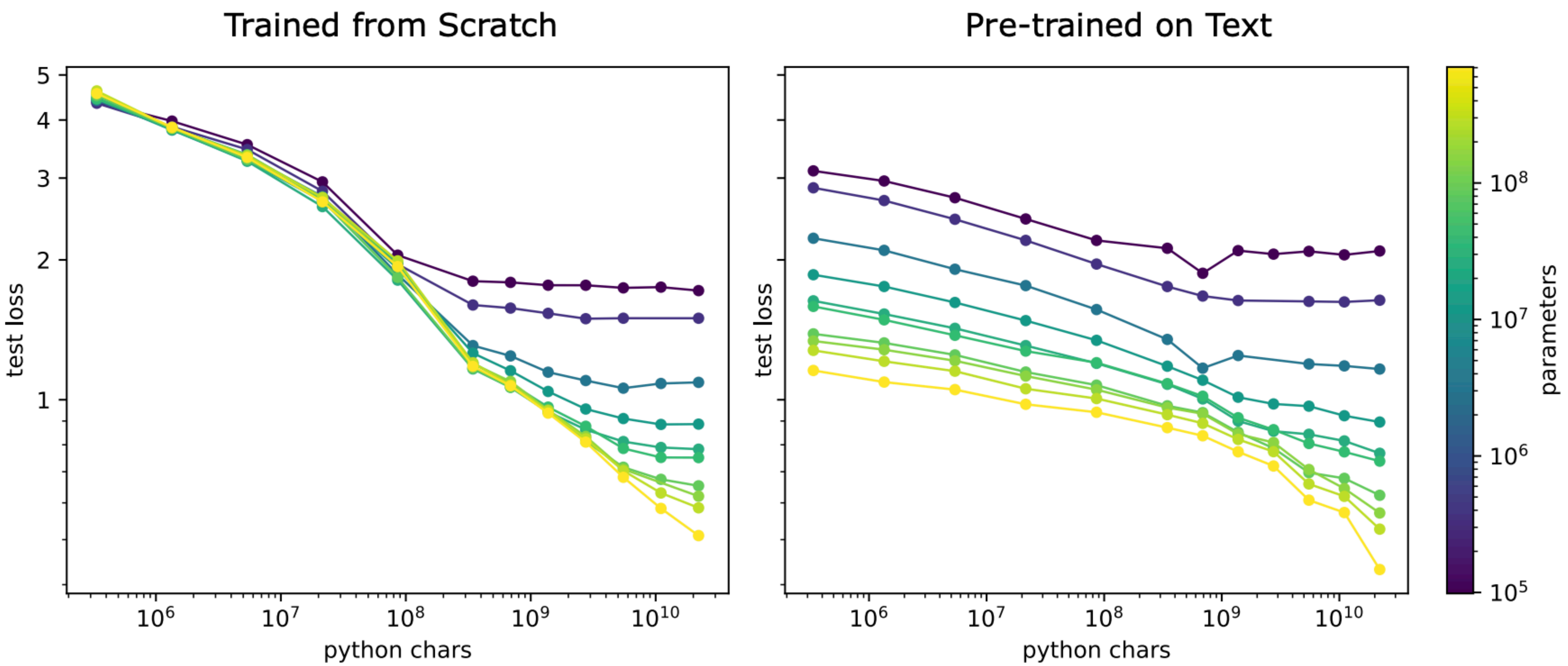
- On low-data regime $D_T \gg D_F$:
  Effective data multiplier $= \dfrac{D_F + D_T}{D_F} \approx \dfrac{D_T}{D_F} = \dfrac{k(N)^{\beta}}{\left(D_F\right)^{1-\alpha}}$

- As fine-tuning data D_F increases, multiplier decreases

Scaling Laws for Transfer (Hernandez et al, 2021)

Can Pre-Training be Harmful?

- Yes, for small models:



- Hypothesized due to pre-training being like a poor initialization point that fine-tuning has trouble recovering from ("ossification")

Scaling Laws for Transfer (Hernandez et al, 2021)

Challenges of Full-Parameter Fine-Tuning

1.5b parameters does not mean using 6gb of vRAM

- Consider a “small” 1.5b GPT-2 model
- Surely you can fine-tune this on your RTX 4080 16GB GPU?

```
RuntimeError: CUDA out of memory. Tried to allocate 200.00 MiB (GPU 0; 15.78 GiB total
capacity; 14.56 GiB already allocated; 38.44 MiB free; 14.80 GiB reserved in total by
PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb to avoid
fragmentation. See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al, 2020)

Where did all the memory go?

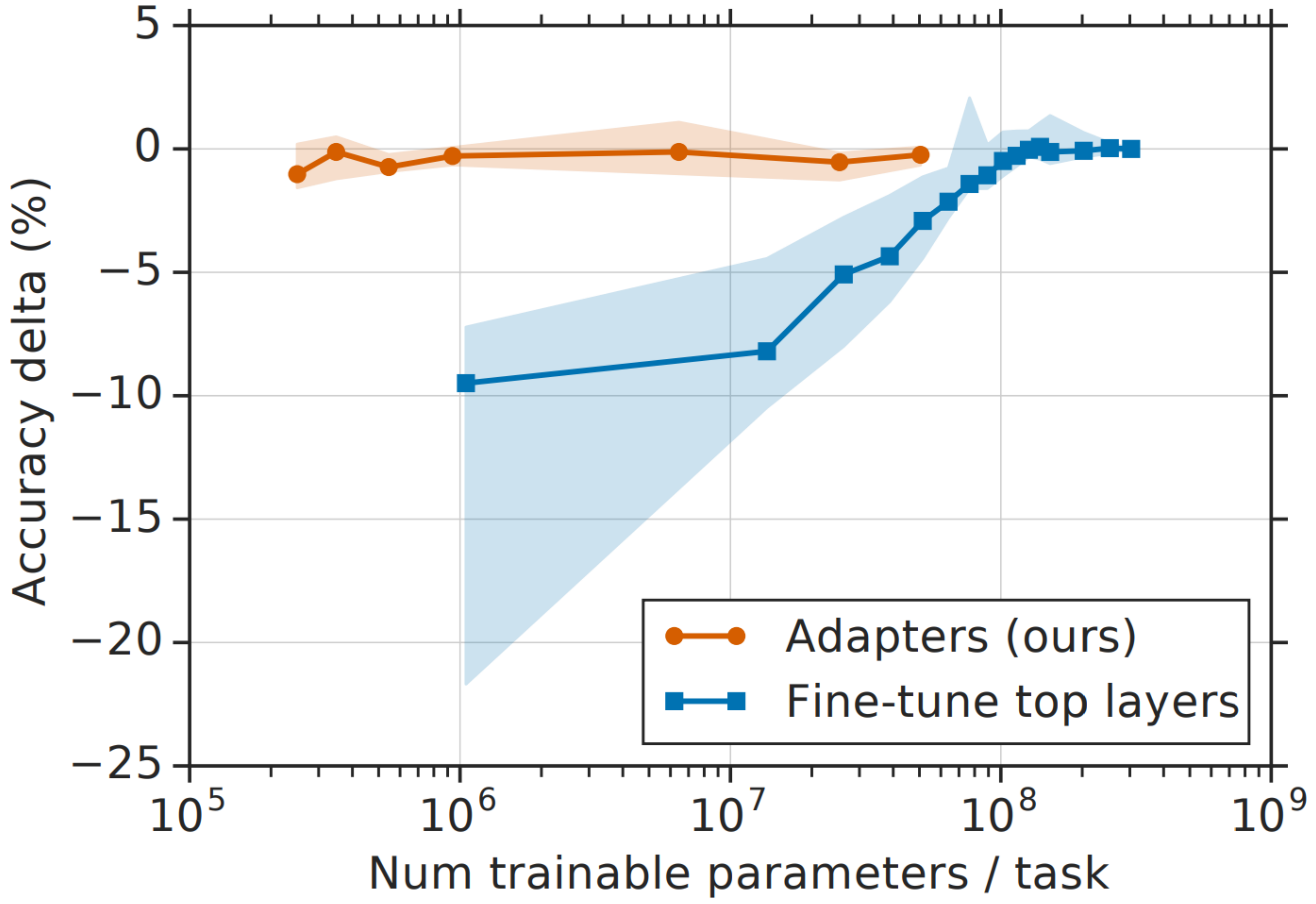
- For float32 data types:
- Parameters: 4 bytes
- Gradients: 4 bytes
- Optimizer state:
  - Suppose we use Adam (most popular for Transformers), which tracks weight and variance in updates
  - 2 * 4 bytes
- Activations: variable (depends on model architecture)
- Also: memory fragmentation, temporary buffers allocated (for gradient norm computation, etc)
- Total: at least 1.5b * (4 + 4 + 8) bytes = 24GB vRAM

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al, 2020)

Adapter Methods

Simplest method: fine-tune top layers

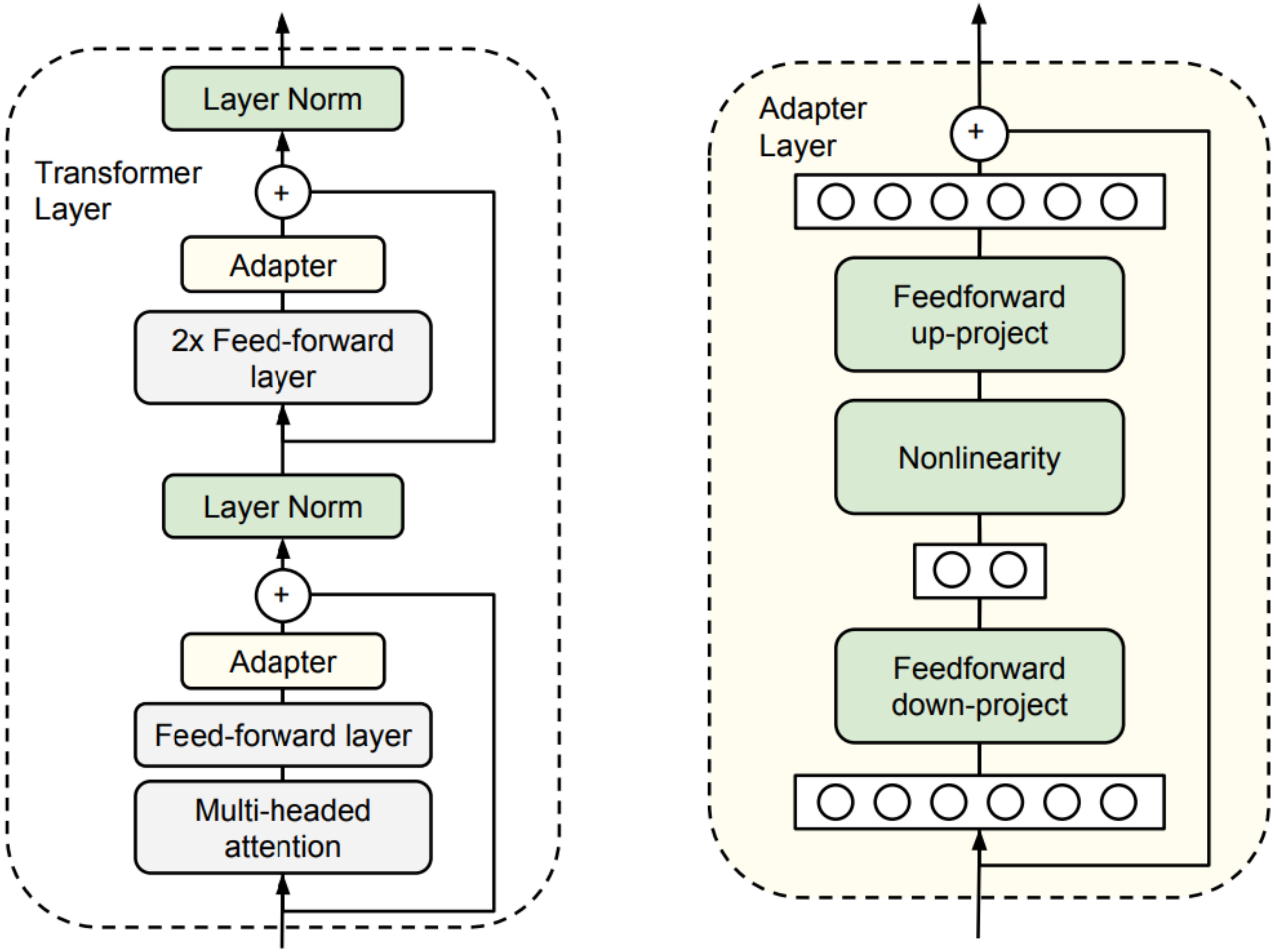
- Freeze all weights but those at top layers (or add additional layers to fine-tune)
- Idea: as you go up the Transformer layers, you build up to higher representations
- Top representations corresponds to high-level features most useful for a specific task
- Outperformed by adapters



Parameter-Efficient Transfer Learning for NLP (Houlsby et al. 2019)

Adapters

- Houlsby et al. introduced adapter modules in Transformer layers which are fine-tuned (all other parameters fixed)
- Adds 3.6% extra parameters



Parameter-Efficient Transfer Learning for NLP (Houlsby et al. 2019)

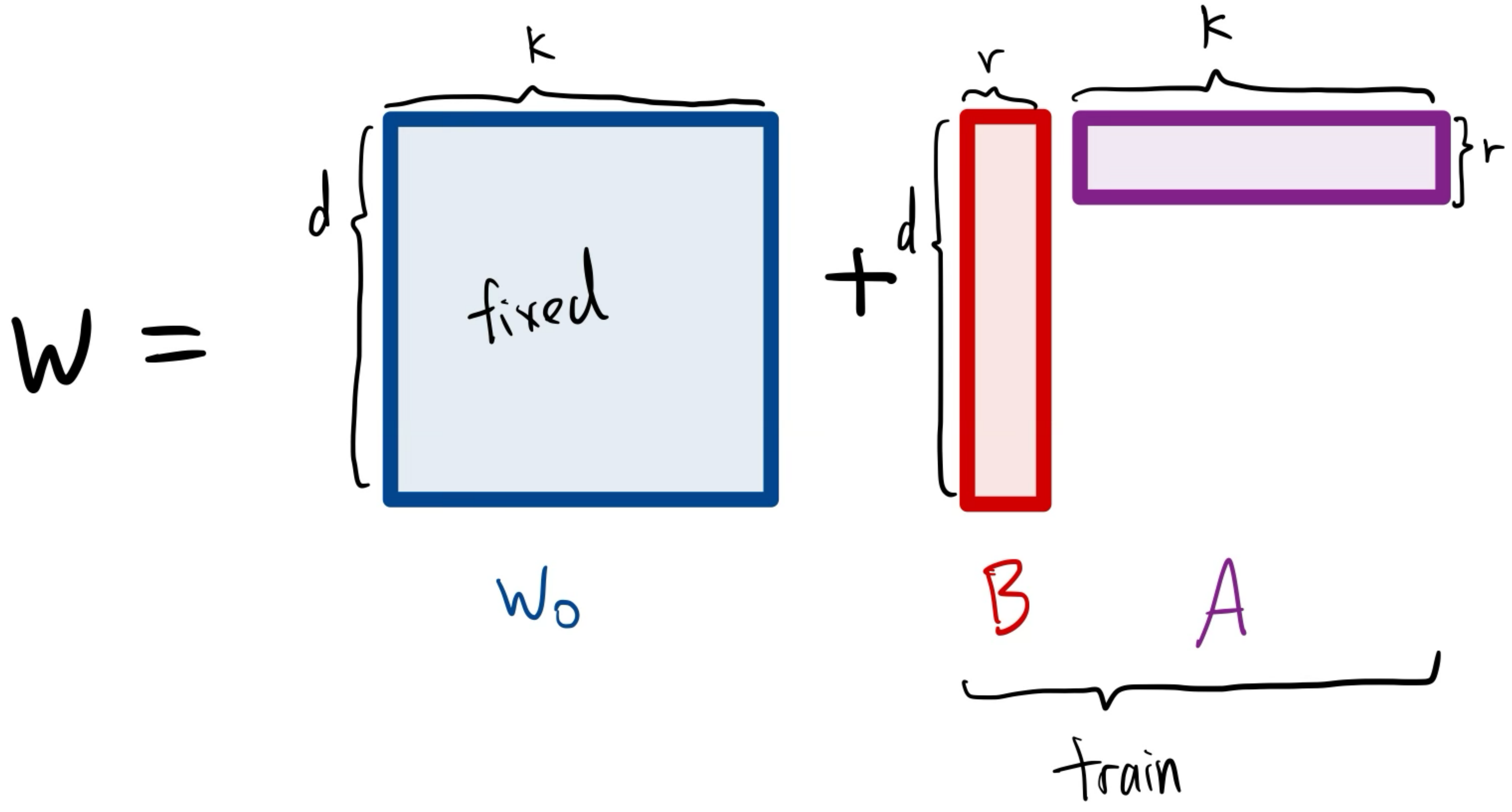
LoRA: Low-Rank Adaptation

- Downside of adapters:
  - Increased inference latency
  - Performs worse than full-parameter fine-tuning
- LoRA addresses both!

LoRA: Low-Rank Adaptation of Large Language Models (Hu et al, 2021)

LoRA: Low-Rank Adaptation

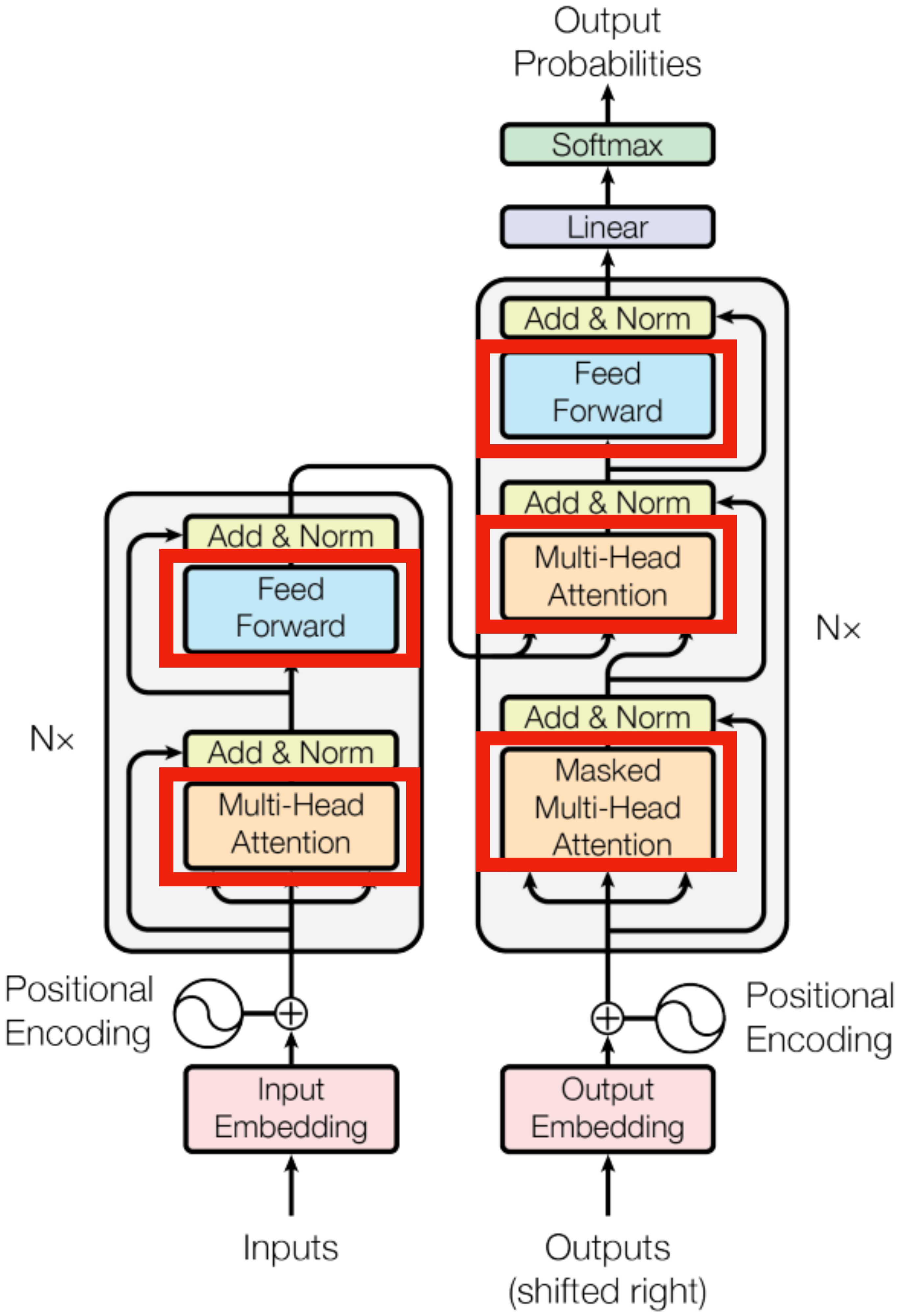
- Have a large pre-trained $d \times k$ weight matrix W_0
- Instead of fine-tuning all weights, consider fine-tuning a low-rank r adapter AB, where $A \in \mathbb{R}^{d \times r}, B \in \mathbb{R}^{r \times k}$
  - r as small as 1
- For inputs x, forward pass yields $h = W_0 x + BAx$
- During back propagation, W_0 is frozen and we only update BA



LoRA: Low-Rank Adaptation of Large Language Models (Hu et al, 2021)

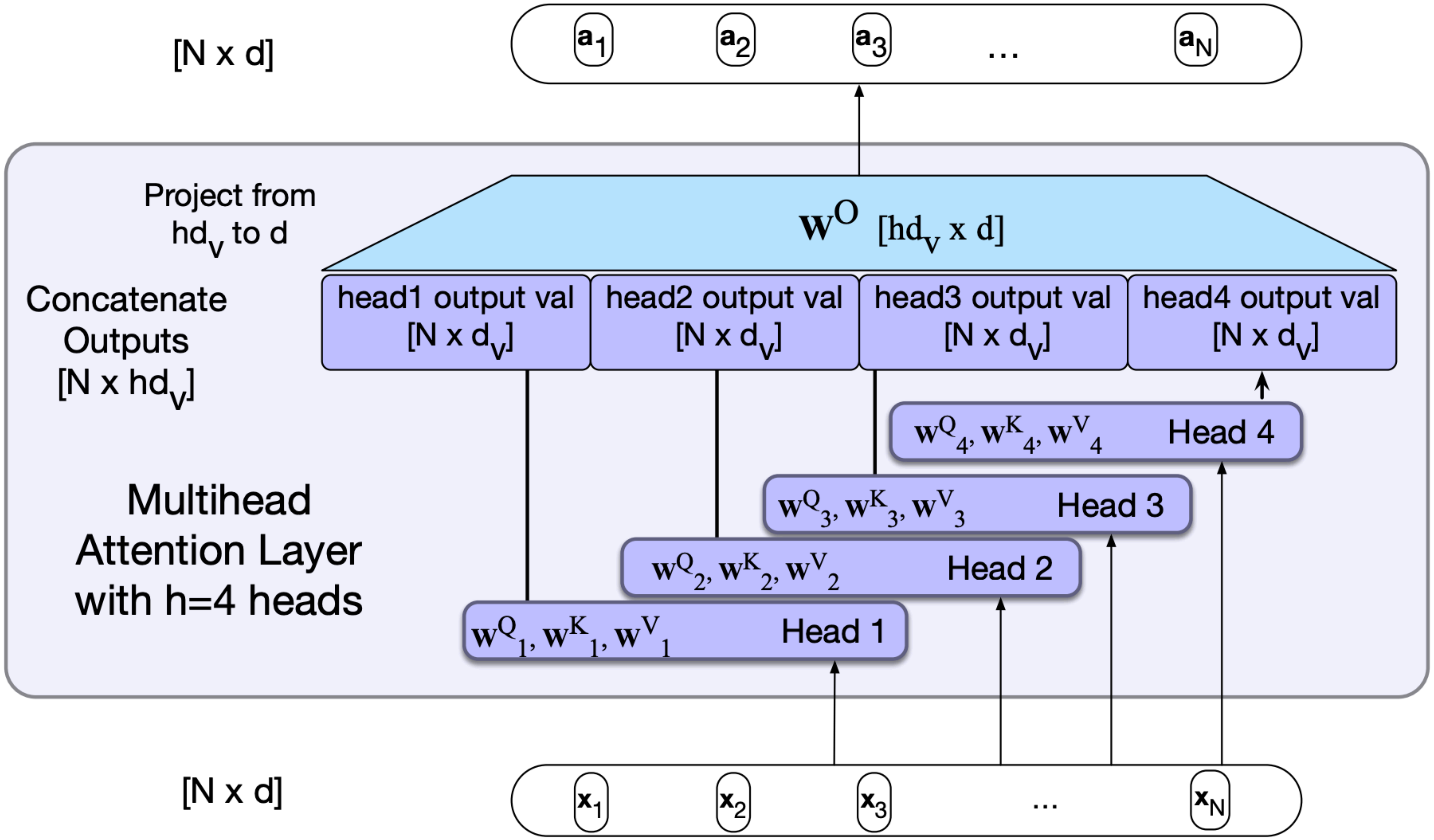
Which weight matrices to use LoRA for?

- Candidates for Transformer models:
  - Weight matrices for self-attention
    $\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \mathbf{K} = \mathbf{X}\mathbf{W}^K, \mathbf{V} = \mathbf{X}\mathbf{W}^V$
  - Weight matrices for output projection
    $\mathbf{W}^o$
  - Weights for feed-forward networks



LoRA: Low-Rank Adaptation of Large Language Models (Hu et al, 2021)

$\mathbf{W}^q, \mathbf{W}^k, \mathbf{W}^v, \mathbf{W}^o$ recap



LoRA: Low-Rank Adaptation of Large Language Models (Hu et al, 2021)

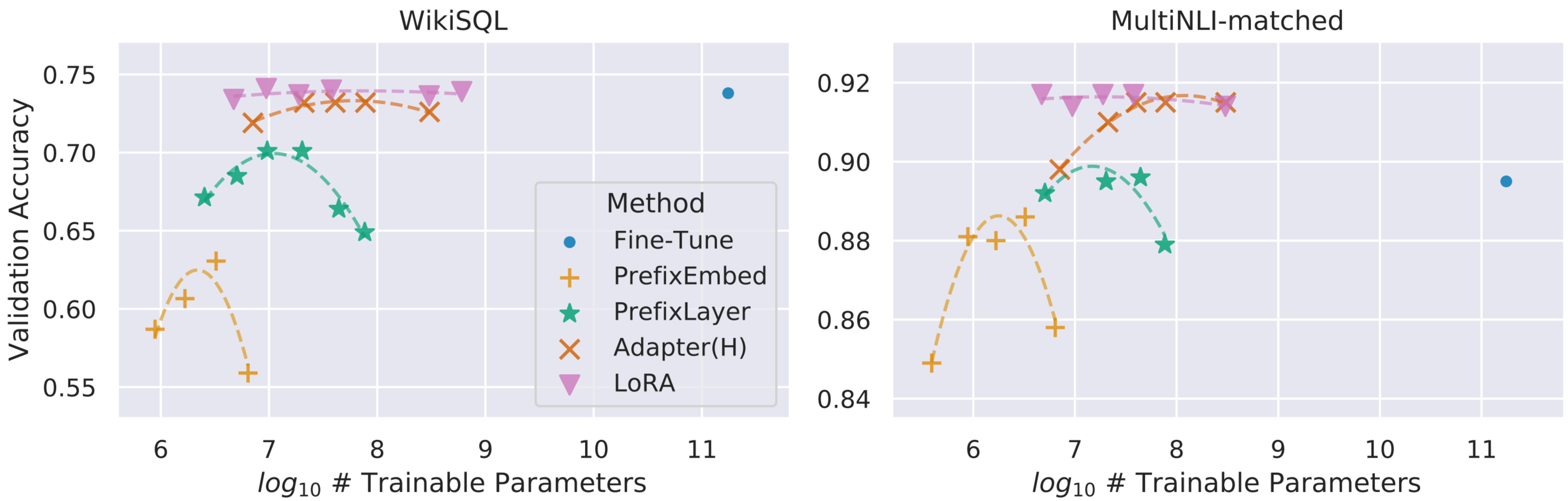
Which weight matrices to use LoRA for?

	# of Trainable Parameters = 18M						
Weight Type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	2
WikiSQL ($\pm 0.5\%$)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI ($\pm 0.1\%$)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

- Generally having both $\mathbf{W}_q, \mathbf{W}_v$ gives best result (as low as rank 4)
- Only fine-tune on an additional 0.01% extra parameters! (18M/175B for GPT-3)

LoRA: Low-Rank Adaptation of Large Language Models (Hu et al, 2021)

Results

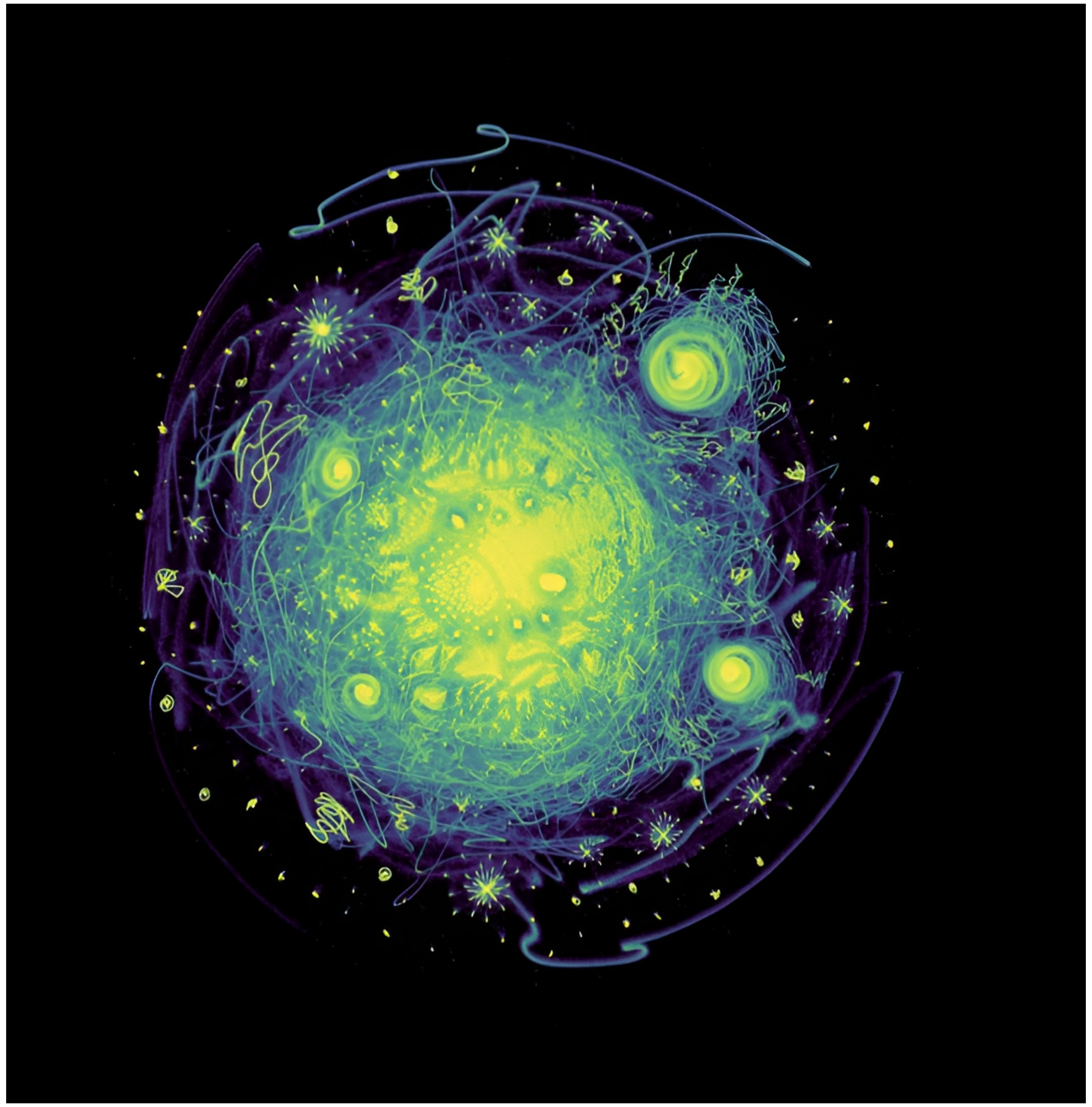


- Note: prefix-tuning approaches start to perform worse when there are too many parameters, hypothesized due to mismatch between input and pre-training data distribution

LoRA: Low-Rank Adaptation of Large Language Models (Hu et al, 2021)

Why does LoRA work?

- Over-parameterized models have intrinsic low-rank structure after training
- Manifold hypothesis: real-world data lives on a low-dimensional manifold inside a high-dimensional space



LoRA: Low-Rank Adaptation of Large Language Models (Hu et al, 2021)

One Base Model, Many Adapters

- Can fine-tune many low-rank adapters for different tasks with the same base model
- During inference time, dynamically swap out for appropriate A, B matrices depending on task, while using shared base model weights
- Much cheaper than serving n different full-parameter fine-tuned models!

Startups have been built on this idea

Predibase

Efficient Fine-Tuning and Serving

Train and deploy task-specific open-source models in record time and under budget.

First-class fine-tuning experience

Predibase offers state-of-the-art fine-tuning techniques out of the box such as quantization, low-rank adaptation, and memory-efficient distributed training to ensure your fine-tuning jobs are fast and efficient—even on commodity GPUs.

The most cost-effective serving infra

With Serverless Fine-Tuned Endpoints and token-based pricing you can stop paying for GPU resources you don't need. Our unique serving infra–LoRAX–lets you cost-effectively serve many fine-tuned adapters on a single GPU in dedicated deployments.

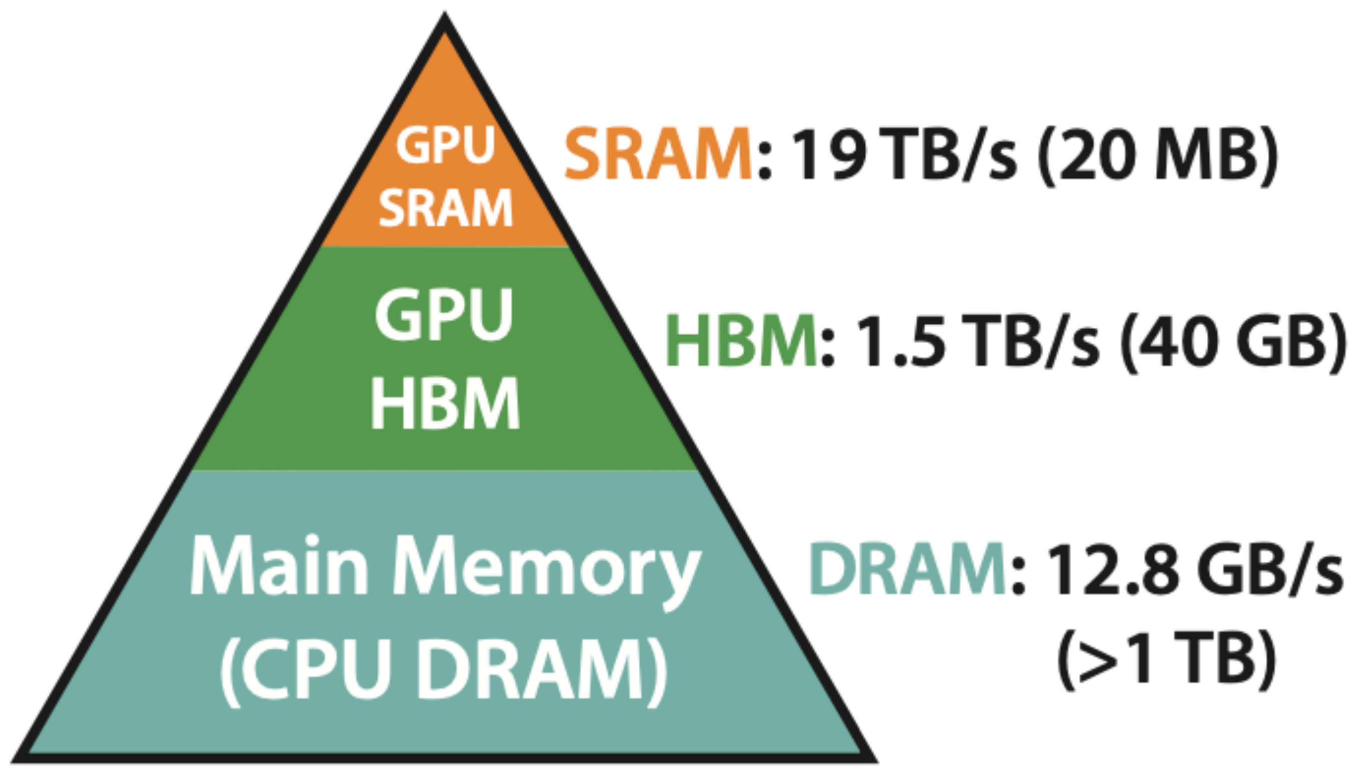
Your Models, Your Property

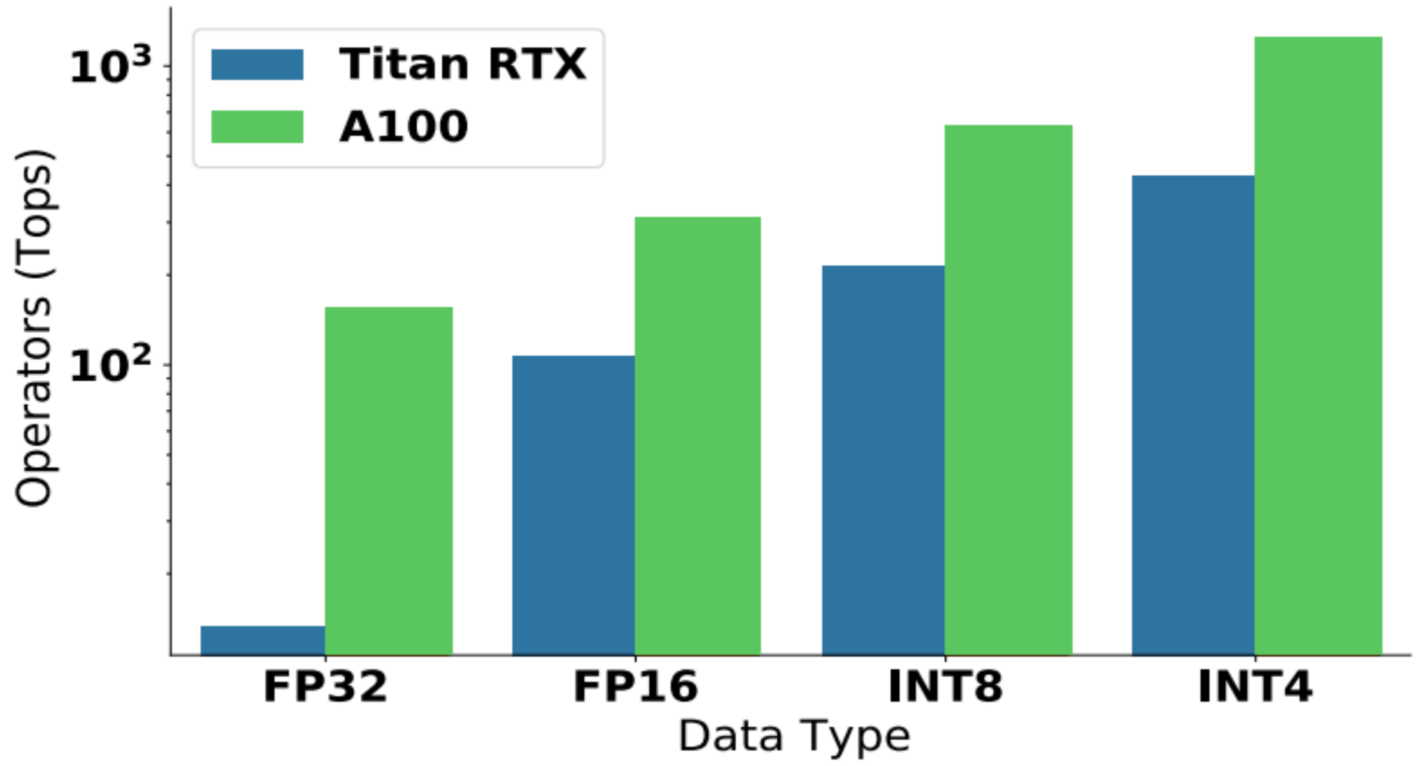
Start owning and stop renting your LLMs. The models you build and customize on Predibase are your property, regardless of whether you use the Predibase Cloud and Serverless Fine-Tuned Endpoints or deploy inside your VPC.

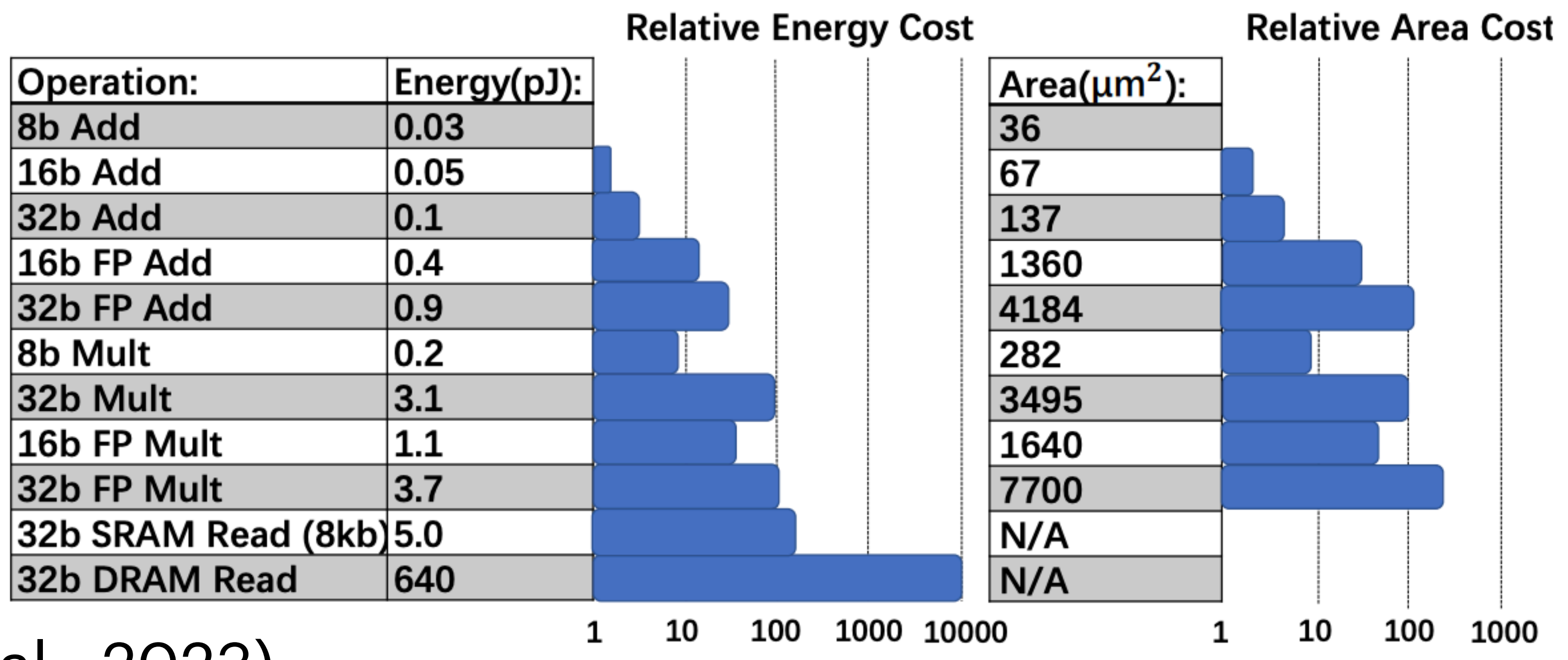
Quantization

Quantization

- Using a lower-precision quantized representation (i.e INT8 instead of FP32)
- Smaller memory footprint
  - Also less memory traffic, better able to take advantage of GPU memory hierarchy
- Less power consumption
  - Integer ALU faster, smaller, & consume less power than floating point ALUs





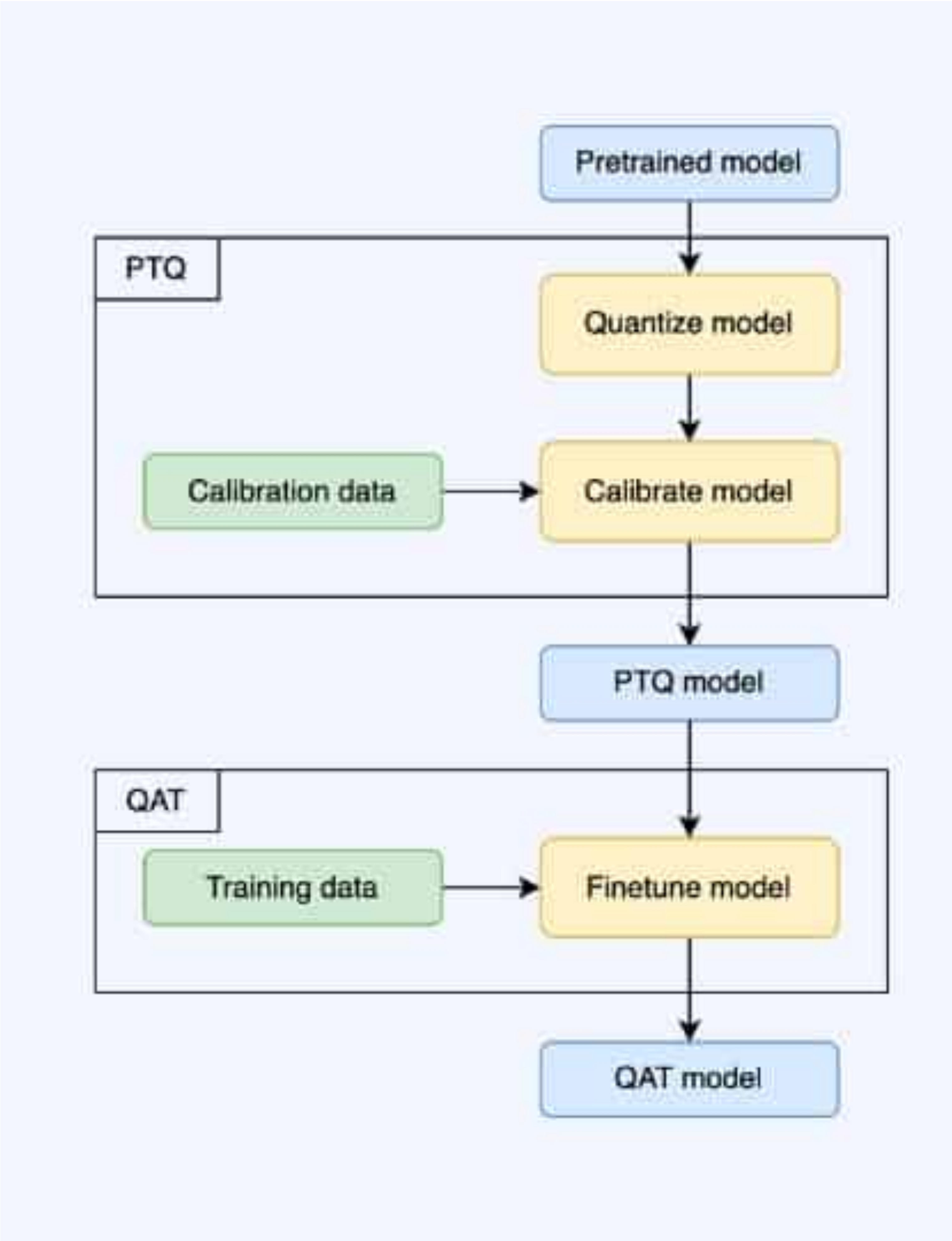


Operation:	Energy(pJ):	Area(μm^2):
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8kb)	5.0	N/A
32b DRAM Read	640	N/A

Full Stack Optimization of Transformer Inference: a Survey (Kim et al., 2023)

Quantization

- 2 main approaches
  - Post-training quantization (PTQ)
    - Quantize model after training
    - Requires calibration
  - Quantization-aware training (QAT)
    - Incorporates quantization while model is training
    - QLoRA
- Calibration: determining range of values that the weights/activations take on for rescaling



Absmax quantization

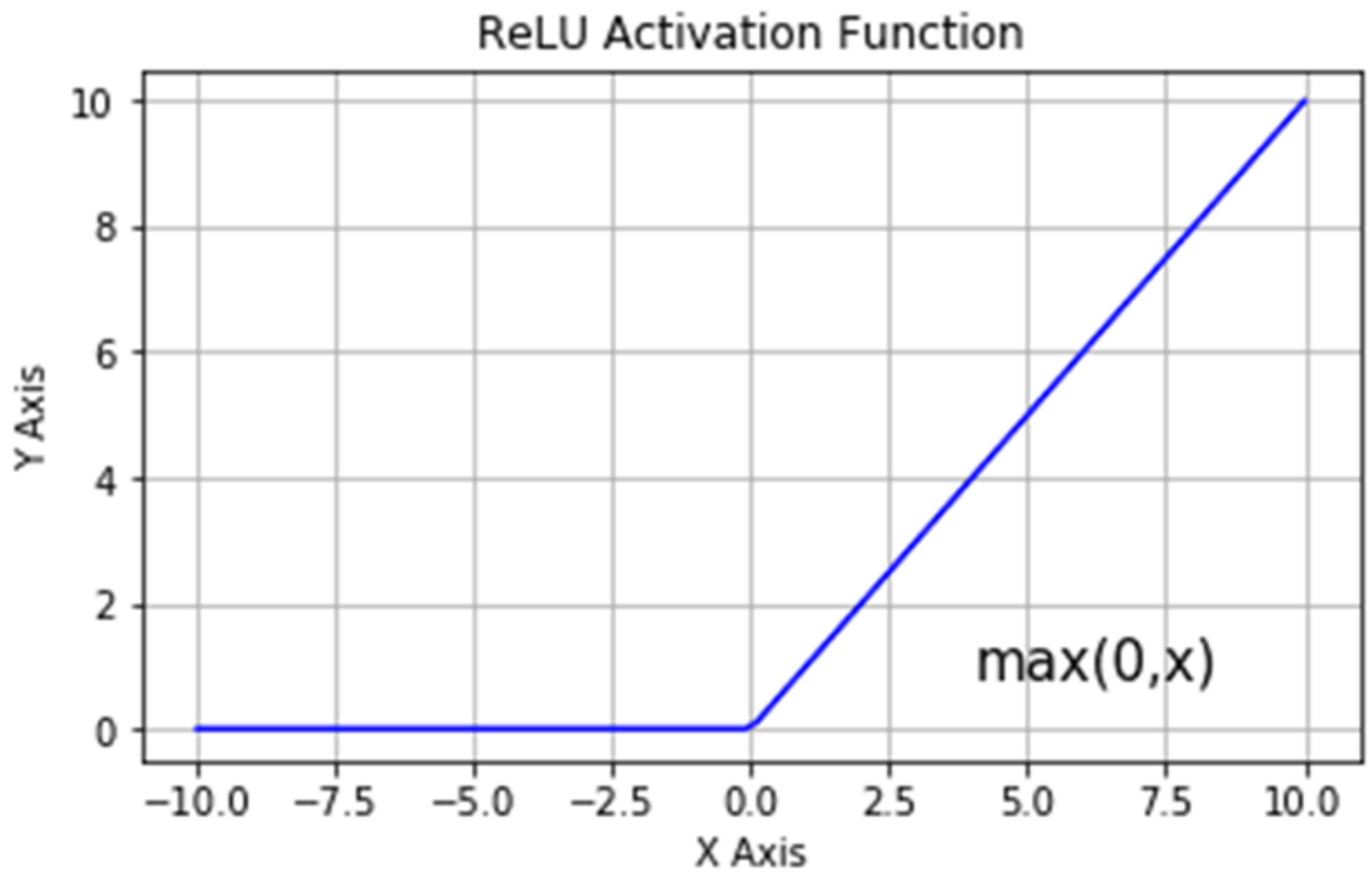
- FP16 input matrix $\mathbf{X}_{f16}$
- To scale inputs into 8-bit range $[-127, 127]$:
  - Divide by abs maximum of tensor
  - Scale by half of range
- Overall:

$$\mathbf{X}_{i8} = \left\lfloor \frac{127 \cdot \mathbf{X}_{f16}}{\max_{ij}\left(\left|\mathbf{X}_{f16_{ij}|}\right|\right)} \right\rceil = \left\lfloor \frac{127}{\|\mathbf{X}_{f16}\|_{\infty}} \mathbf{X}_{f16} \right\rceil = \left\lfloor s_{x_{f16}} \mathbf{X}_{f16} \right\rceil$$

where $\lfloor \rceil$ denotes rounding to nearest integer

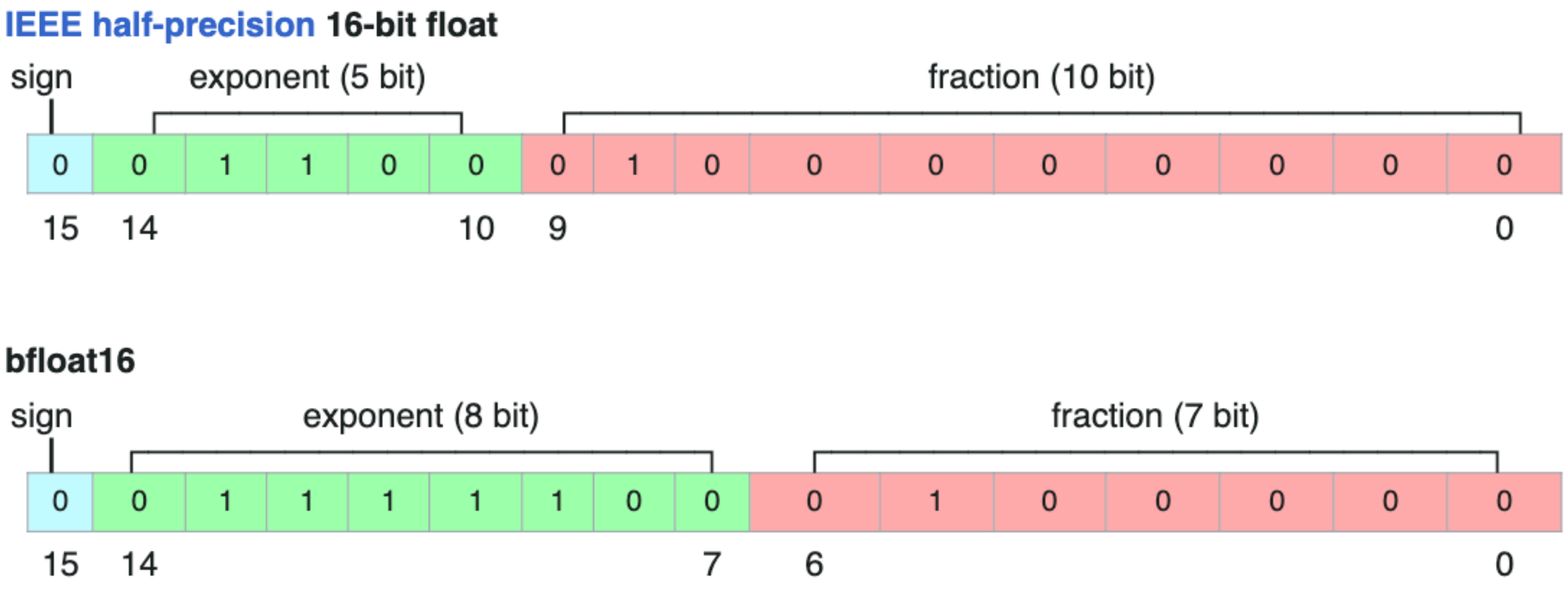
Zeropoint quantization

- Absmax quantization wasteful for asymmetric distributions
  - I.e ReLU activations never use negative values
- Instead: scale by range, then offset by smallest value to make sure all values in range are used
- There are SIMD instructions to do this efficiently (i.e PMADDUBSW)



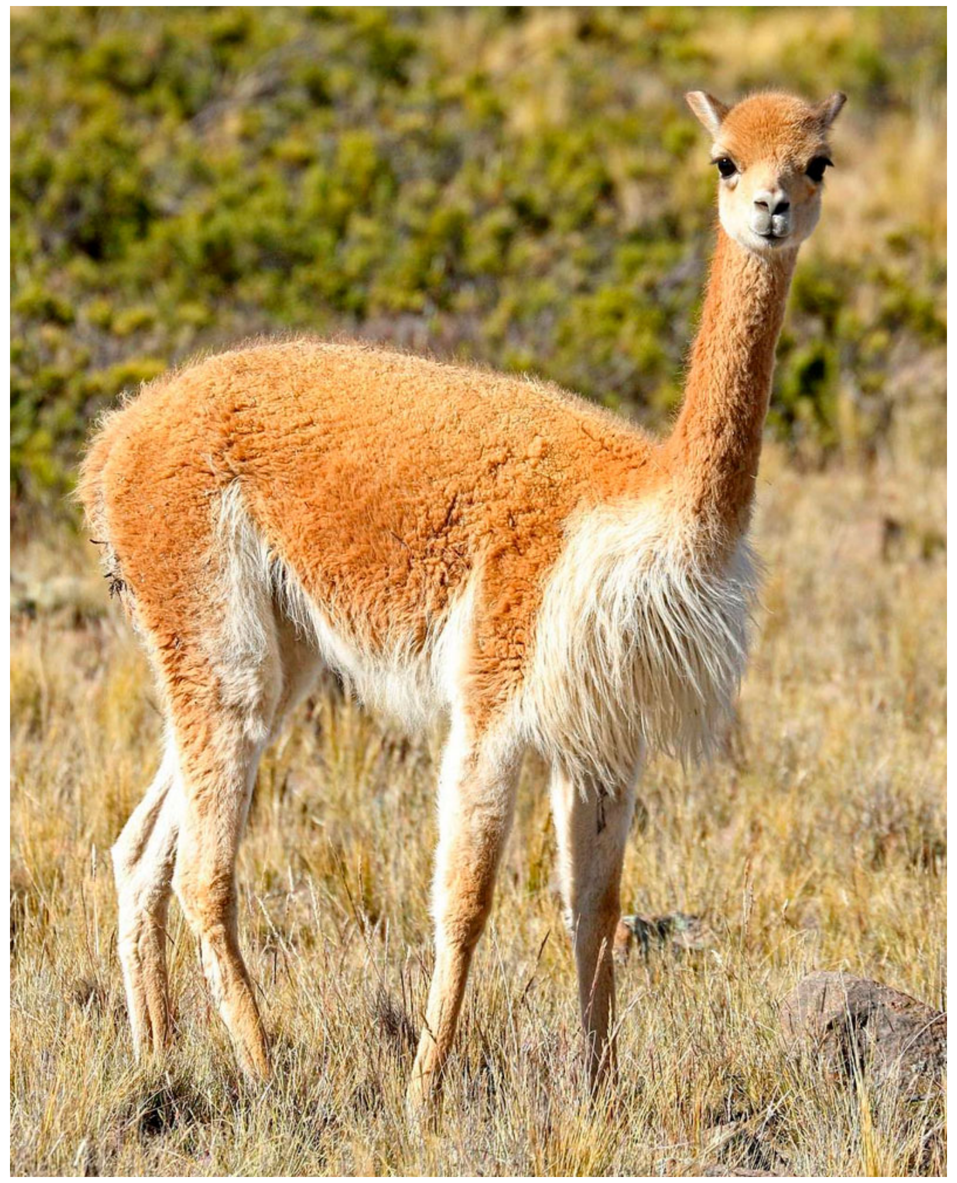
bfloat16

- Use more bits for exponents to sacrifice significand precision
- Supports wider range of values



Pop Quiz

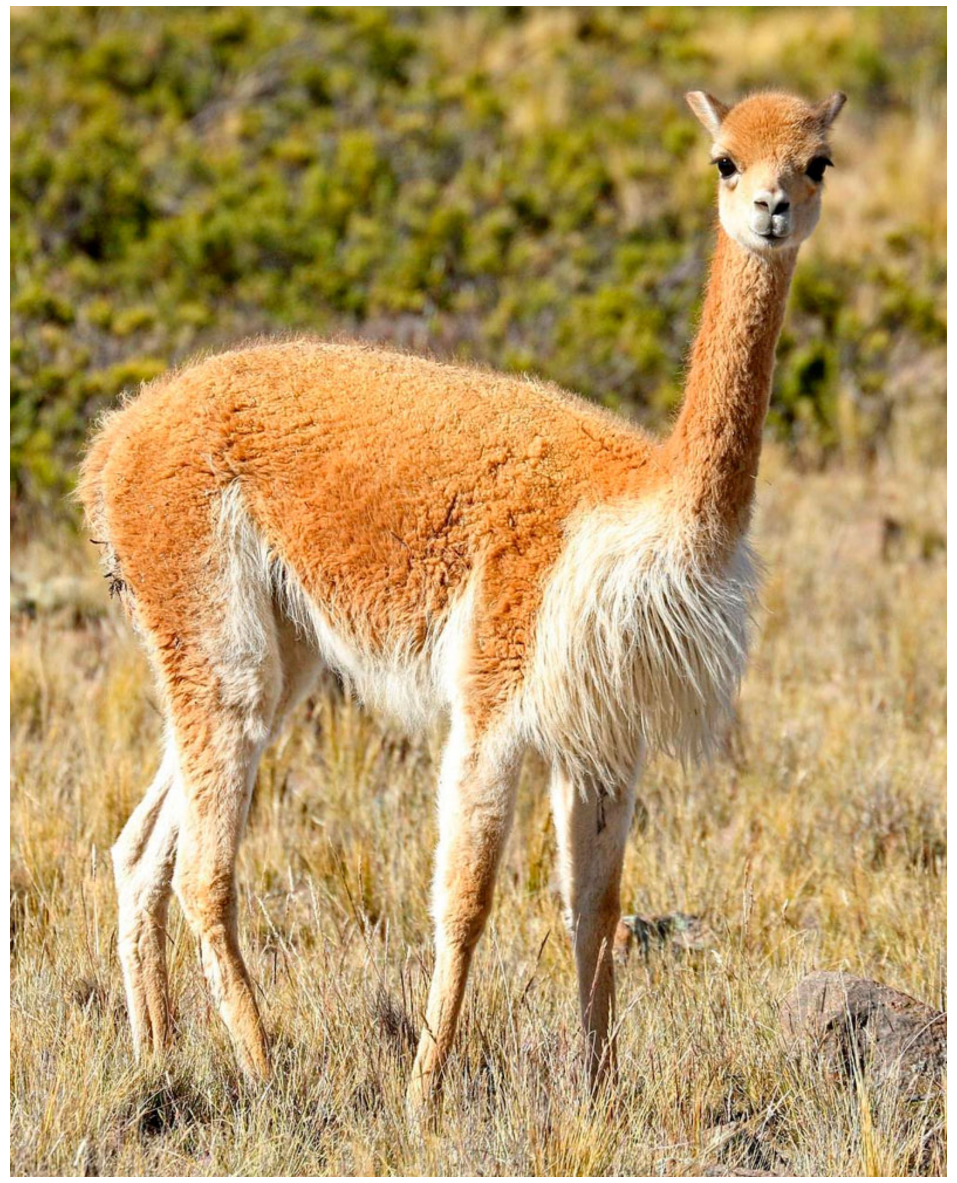
What are these animals?

Pop Quiz

They are all named after LLMs!

Llama (Meta)



Vicuna (LMSYS)

Guanaco (QLoRA)

Alpaca (Stanford)

Block-wise Quantization

- Large outlier features in tensors can cause common small magnitude values to lose a lot of accuracy (quantization error)
- Non-linear quantization methods can address this, but at significant computational cost
- Solution: block-wise quantization
  - Split tensor into blocks
  - Quantize each block independently
  - Improves quantization precision by isolating outliers

8-bit Optimizers via Block-wise Quantization (Dettmers et al., 2021)

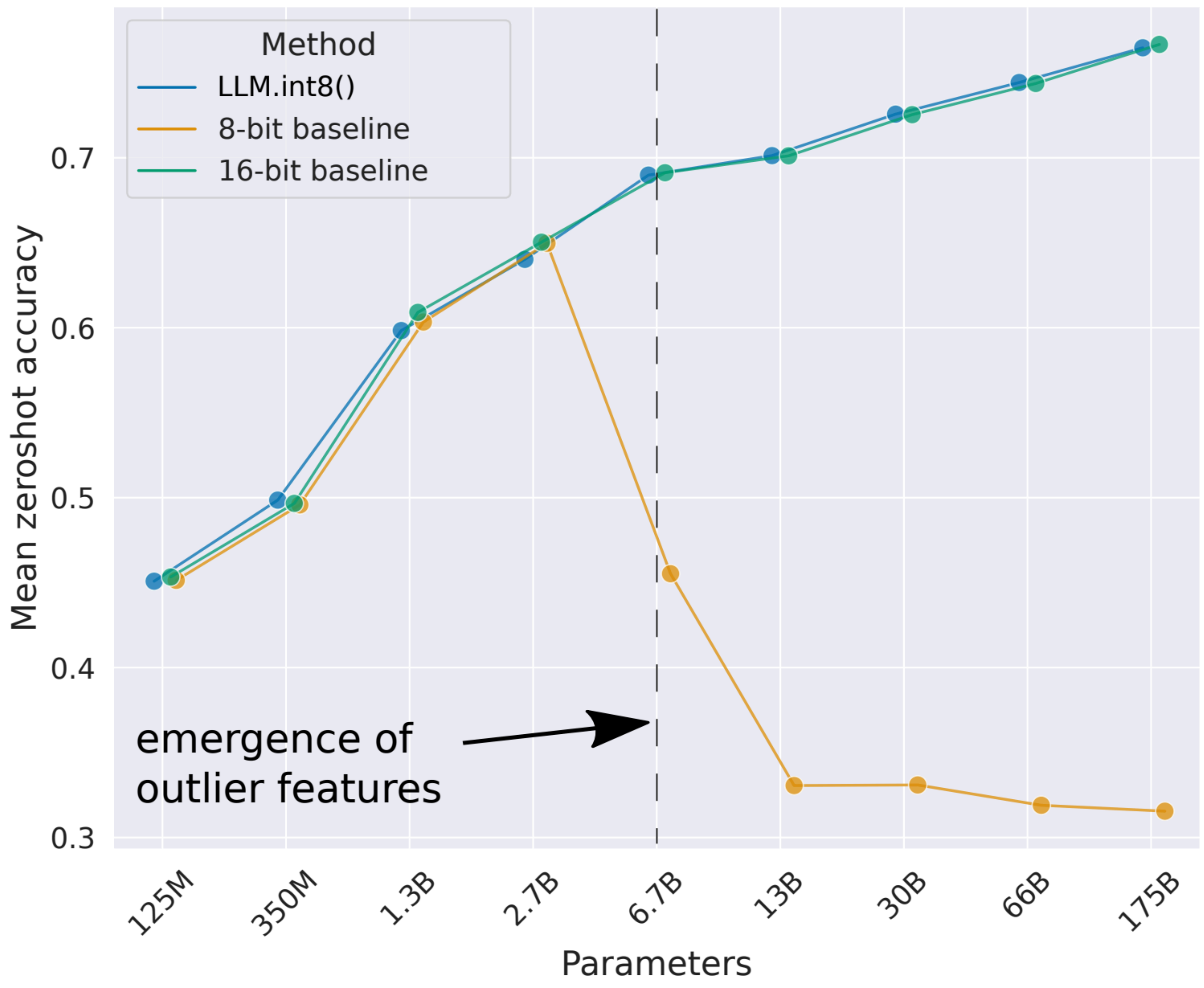
Vector-wise Quantization

- We can apply the same idea for tensors
- Suppose we have hidden states $\mathbf{X}_{f16} \in \mathbb{R}^{b \times h}$ and weights $\mathbf{W}_{f16} \in \mathbb{R}^{h \times o}$
- Computing $\mathbf{X}_{f16}\mathbf{W}_{f16}$ requires computing the dot product of each row of $\mathbf{X}_{f16}$ against each column of $\mathbf{W}_{f16}$
- We can perform absmax quantization for each row and column for $\mathbf{X}_{f16}$ and $\mathbf{W}_{f16}$ respectively, with normalization constants $\mathbf{c}_{x_{f16}}, \mathbf{c}_{w_{f16}}$ respectively
- $\mathbf{C}_{f_{16}} \approx \frac{1}{\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}} \mathbf{C}_{i32} = \frac{1}{\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}} Q\left(\mathbf{A}_{f16}\right) Q\left(\mathbf{B}_{f16}\right)$

8-bit Optimizers via Block-wise Quantization (Dettmers et al., 2021)

Outlier Features

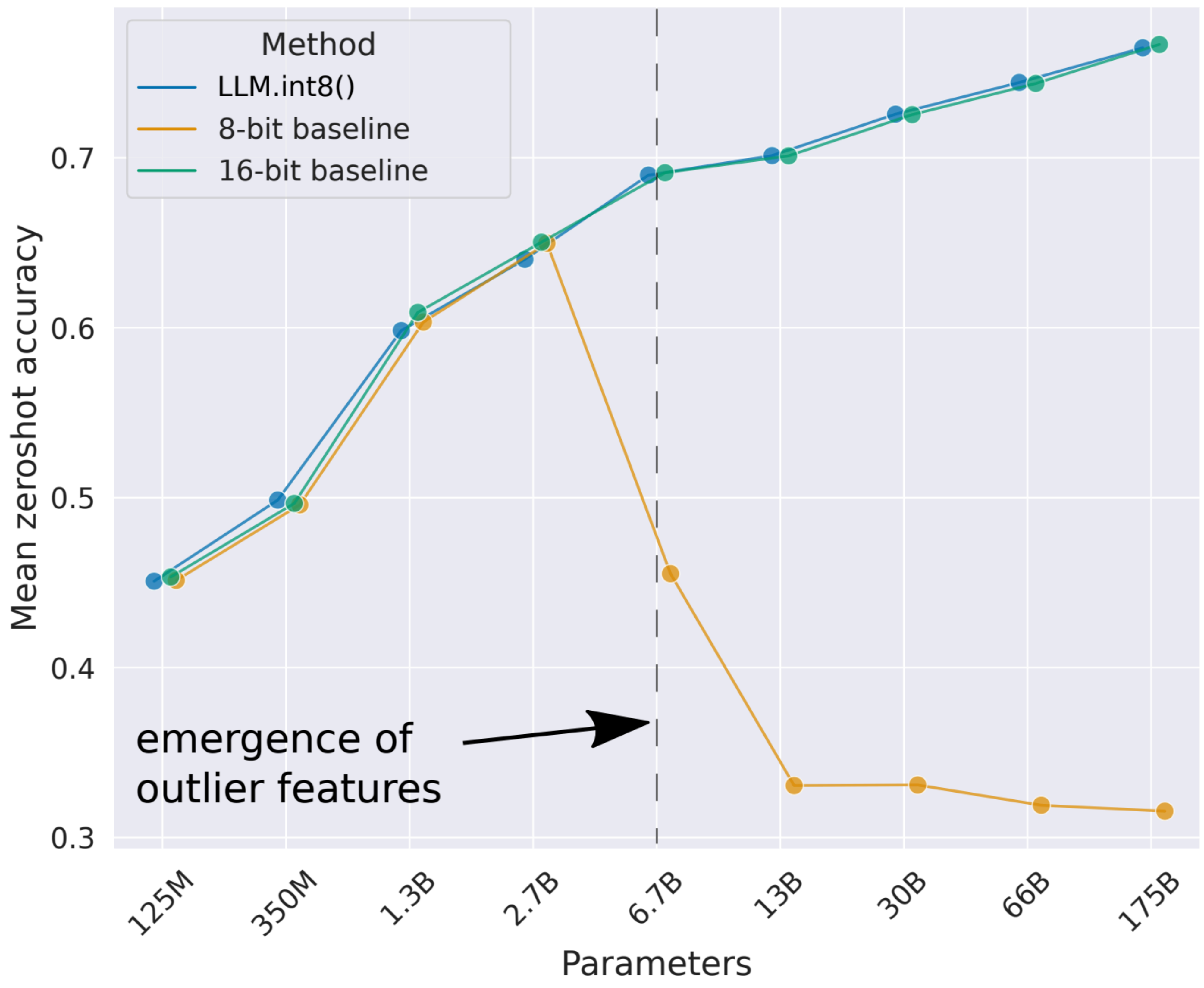
- As model size increases beyond 6.7B, emergence of sparse but large magnitude outlier features ruin quantization precision
- What if we just throw outlier features away?



LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale (Dettmers et al., 2022)

Outlier Features Are Essential

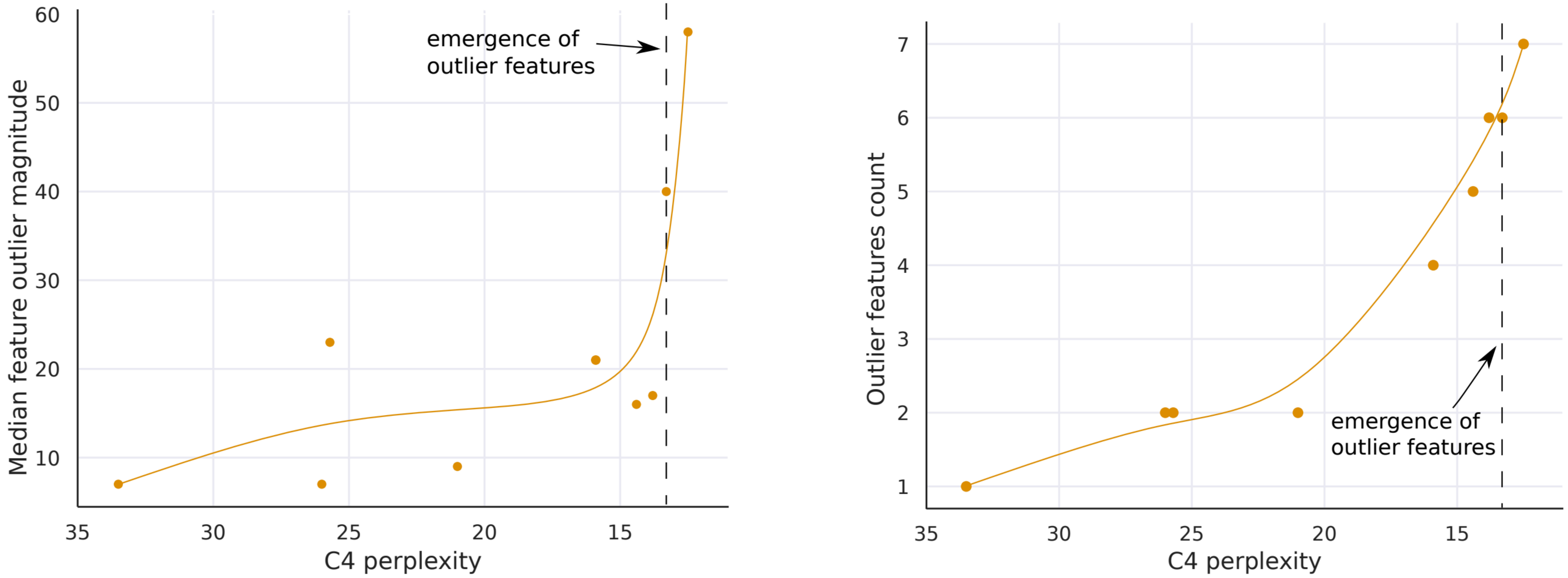
- At 6.7B, 150k outliers occur per sequence concentrated along 6 feature dimensions
- Setting these to 0 causes validation perplexity to increase by 600-1000% despite being only 0.1% of input features
- In contrast, setting same amount of random features to 0 only degrades perplexity by 0.1%



LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale (Dettmers et al., 2022)

Outlier Features Are Huge

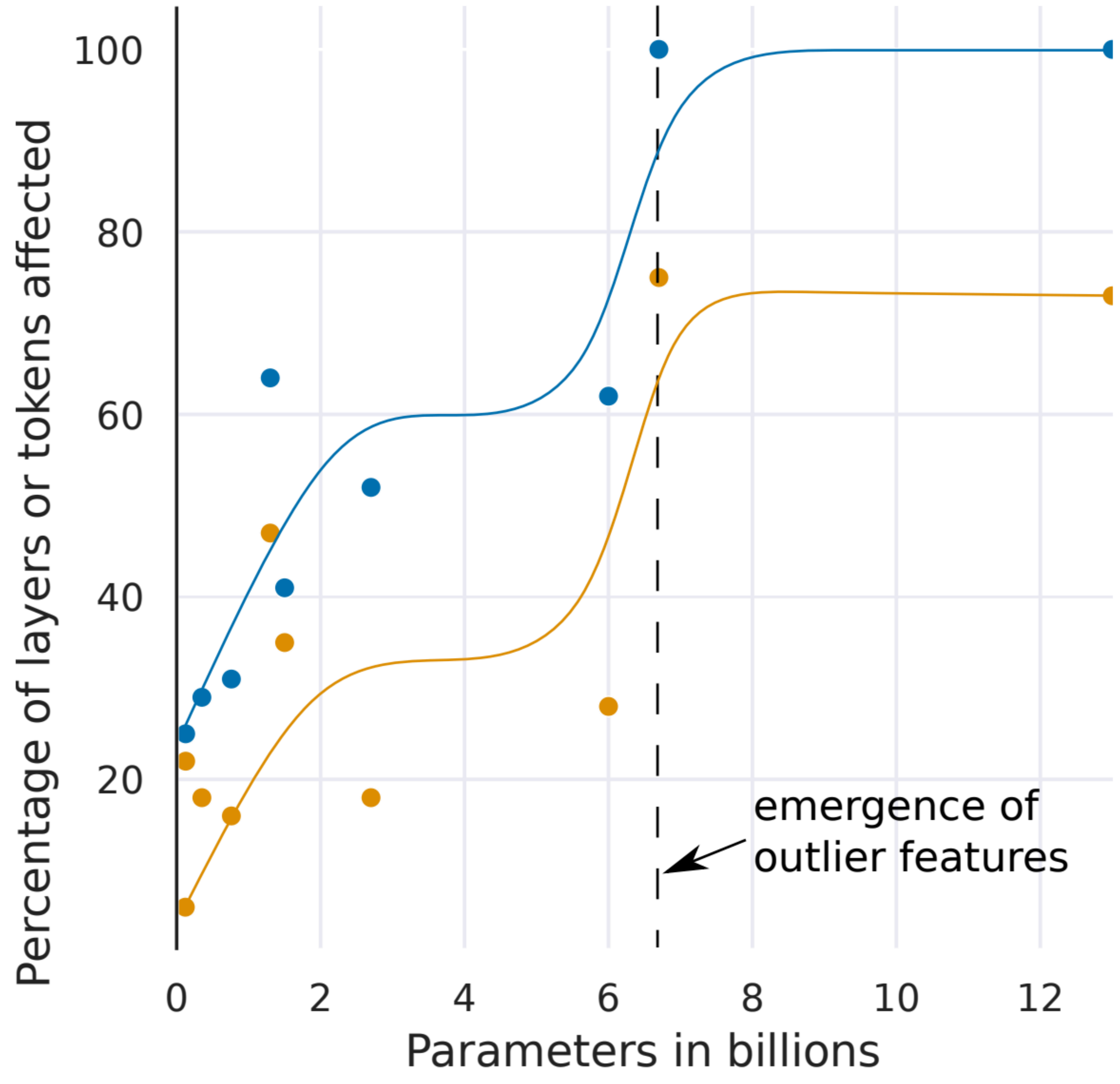
- As evaluation perplexity decreases, outlier features also blow up in magnitude & further ruins quantization precision
- Fortunately outlier features are rare

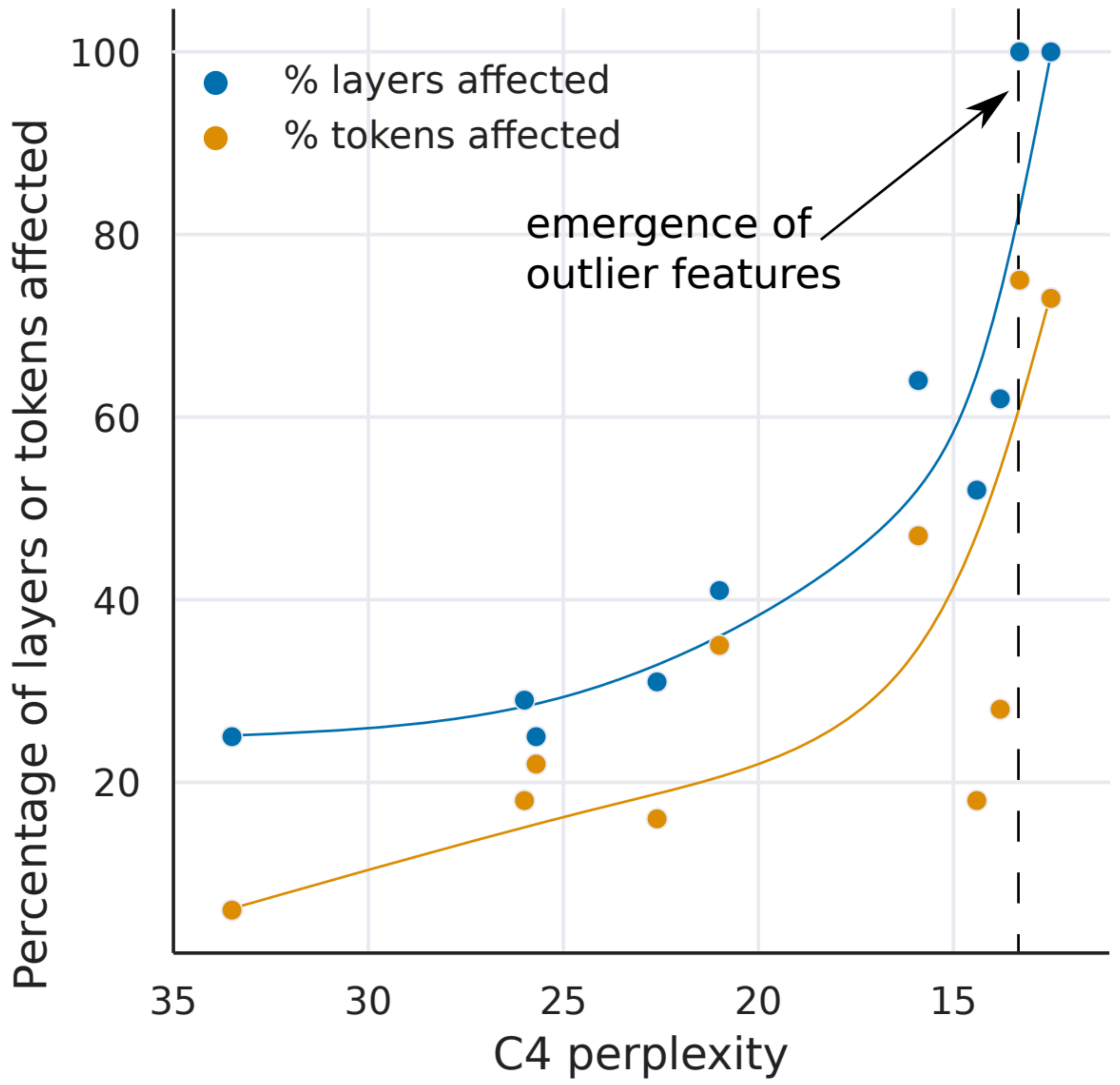


LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale (Dettmers et al., 2022)

Outlier Features Affects Almost All Layers and Tokens

- Almost all layers and 75% tokens affected by outlier features beyond 6.7B
- Not actually a phase shift but smooth transition when measured against perplexity





LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale (Dettmers et al., 2022)

Solution: mixed precision

- Simple solution: Don't quantize dimensions with outlier features!
- Let O be the set of all outlier feature dimensions
- Then

$$\mathbf{C}_{f16} \approx \sum_{h \in O} \mathbf{X}_{f16}^{h} \mathbf{W}_{f16}^{h} + \frac{1}{\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}} \cdot \sum_{h \notin O} \mathbf{X}_{i8}^{h} \mathbf{W}_{i8}^{h}$$

- Only ~7 outlier feature dimensions for Transformers up to 13B, so only adds 0.1% additional memory

LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale (Dettmers et al., 2022)

QLoRA

- Most of LoRA memory usage not from adapter parameters, but from activation gradients
  - I.e LLaMA 7B batch size 1: input gradients take up 567 MB but LoRA parameters only use 26 MB
- Marginal savings from trying to use fewer adapter parameters
- They really wanted to fine-tune LLaMA 70B on just 2 consumer GPUs but LoRA requires 154 GB memory -> 8x consumer GPUs

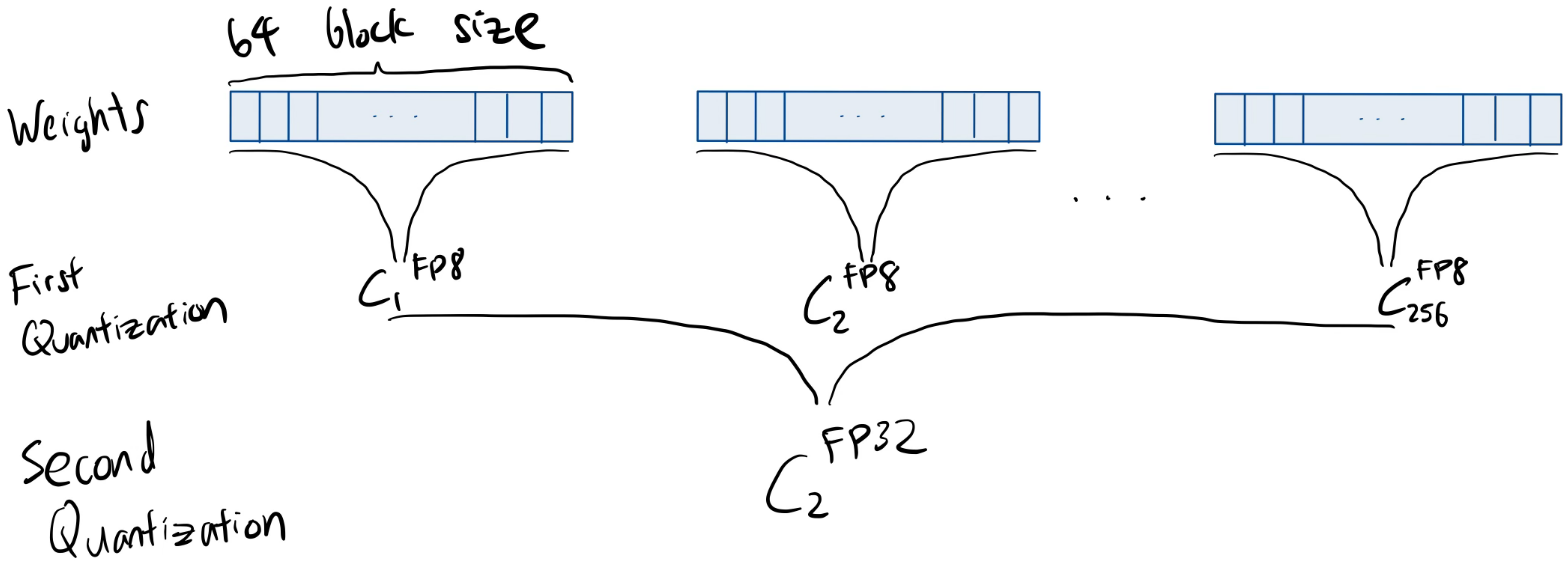
QLoRA: Efficient Finetuning of Quantized LLMs

Double Quantization

- Block-wise quantization require scaling constants to be saved for each block
- Smaller blocks help reduce effect of outliers, but requires higher memory usage
  - I.e block size of 64 with 32-bit scaling constants gives 0.5 bits/parameter memory overhead
- Solution: quantize the quantization constants!

QLoRA: Efficient Finetuning of Quantized LLMs

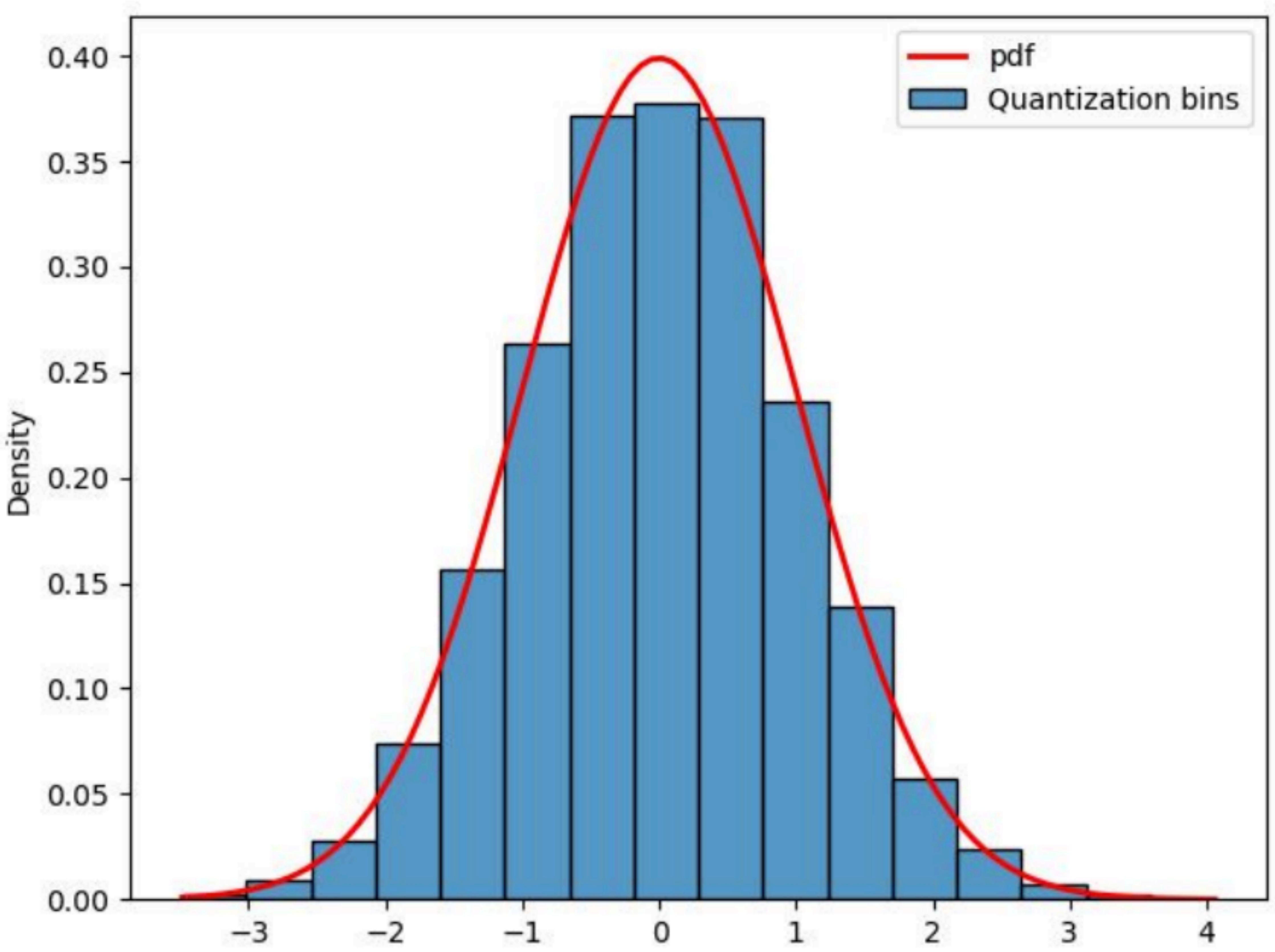
Double Quantization

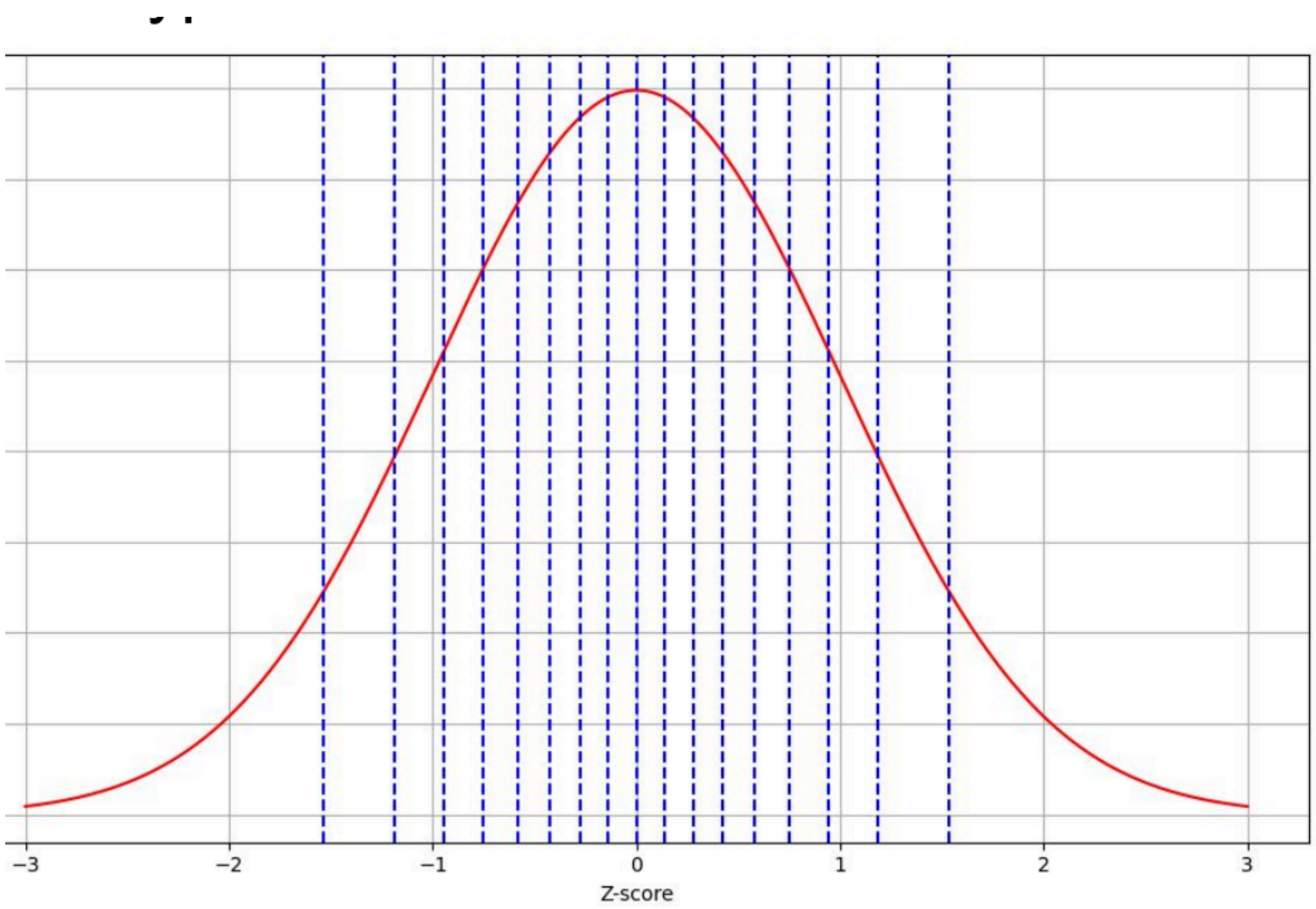


- Memory overhead: 8/64 + 32/(64 · 256) = 0.127 bits per parameter, reduction of 75%

QLoRA: Efficient Finetuning of Quantized LLMs

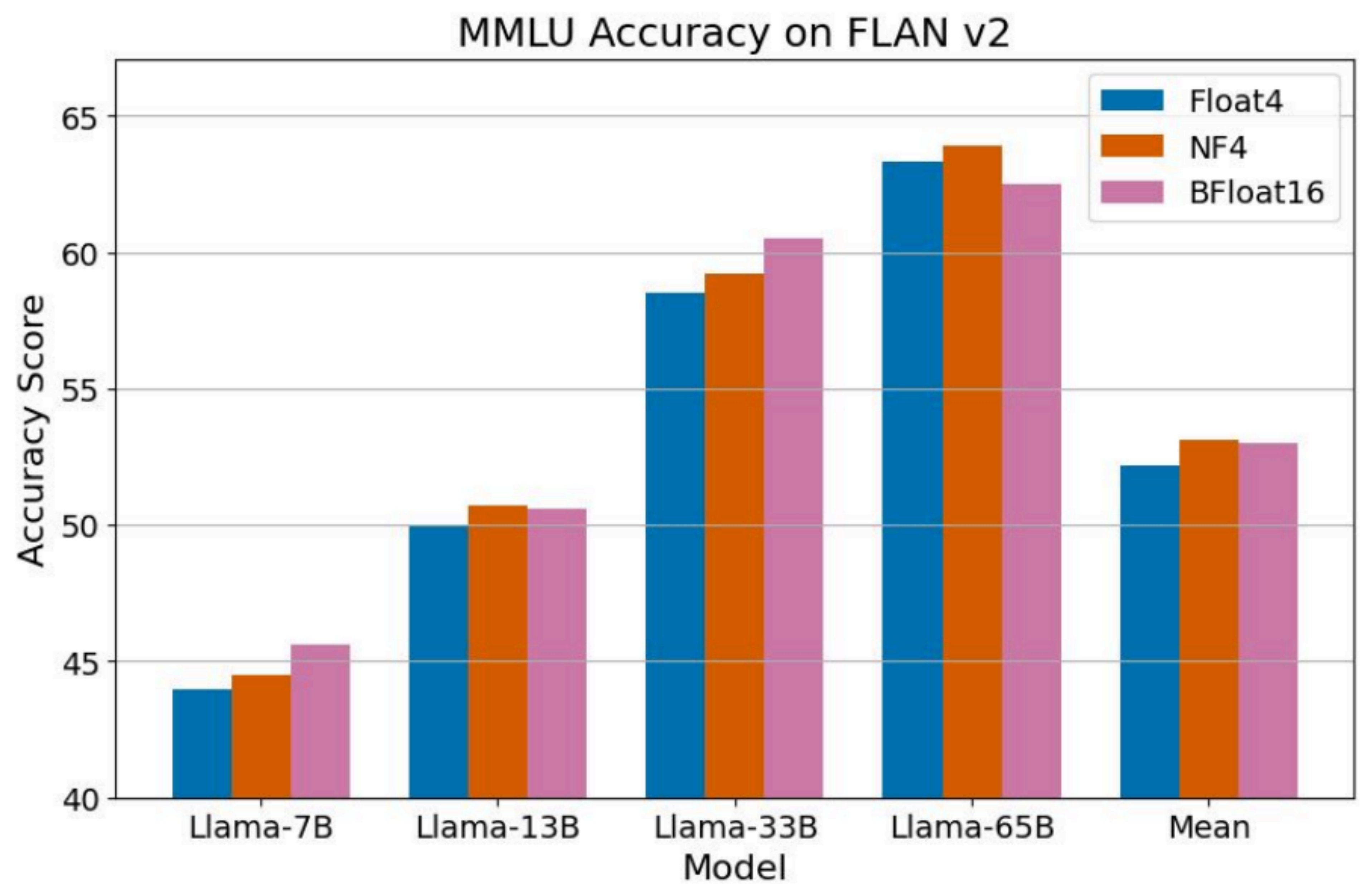
4-bit NormalFloat (NF4)





QLoRA

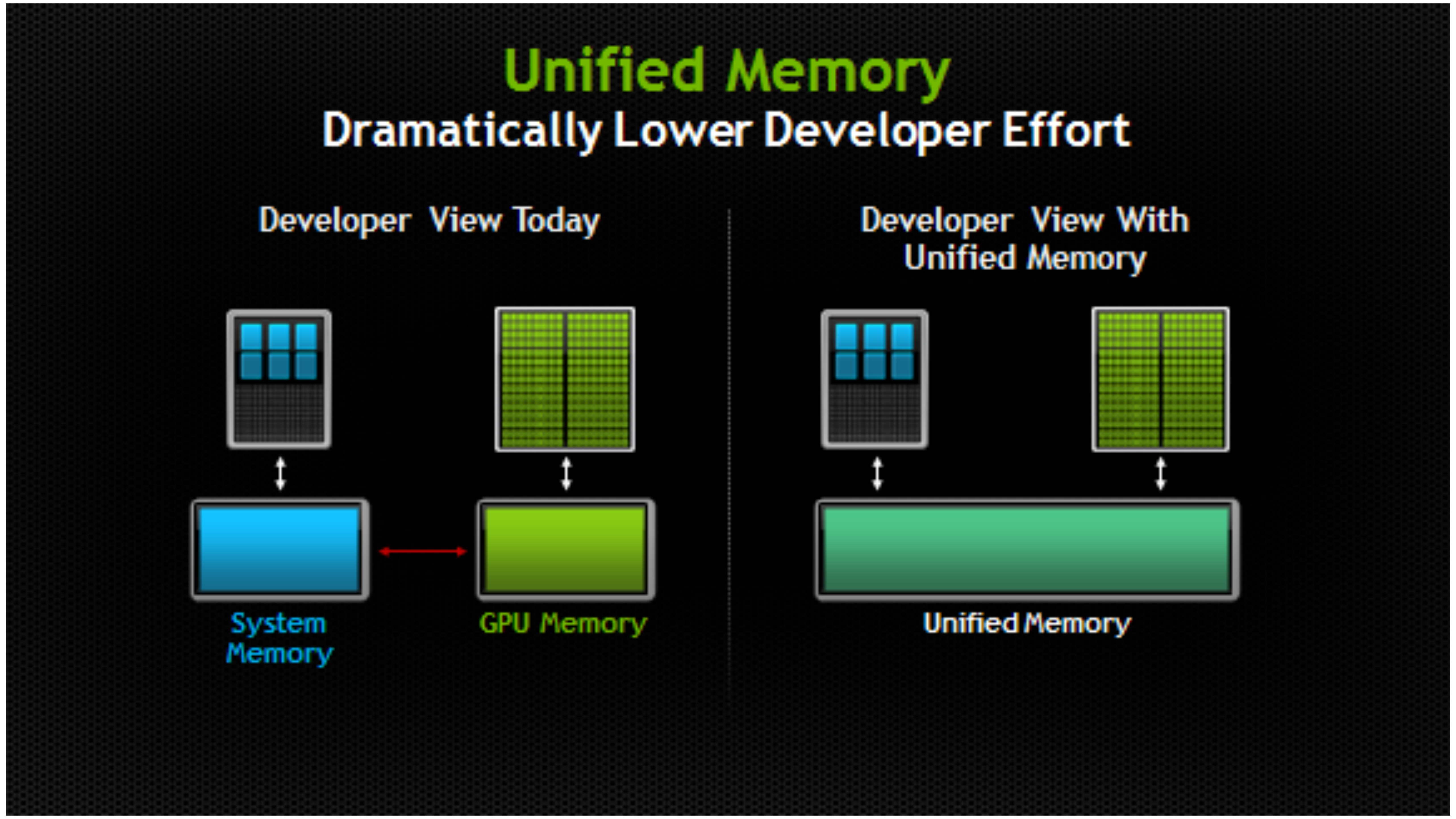
- Recall LoRA: $h = W_0 x + BAx$
- QLoRA:
$$\mathbf{Y}^{\mathrm{BF16}} = \mathbf{X}^{\mathrm{BF16}} \,\text{doubleDequant}\left(c_1^{\mathrm{FP32}}, c_2^{\mathrm{k-\,bit}}, \mathbf{W}^{\mathrm{NF4}}\right) + \mathbf{X}^{\mathrm{BF16}}\mathbf{L}_1^{\mathrm{BF16}}\mathbf{L}_2^{\mathrm{BF16}}$$
- Inputs and adapter parameters remain in BF16
- Replicates performance of BF16 fine-tuning!



QLoRA: Efficient Finetuning of Quantized LLMs

Paged Optimizers

- Memory spikes during training (i.e with long sequence length inputs) can cause GPU OOM
- They introduced paged optimizers to page optimizer state information between GPU and CPU memory



QLoRA: Efficient Finetuning of Quantized LLMs (Dettmers et al., 2023)

Prefix Tuning

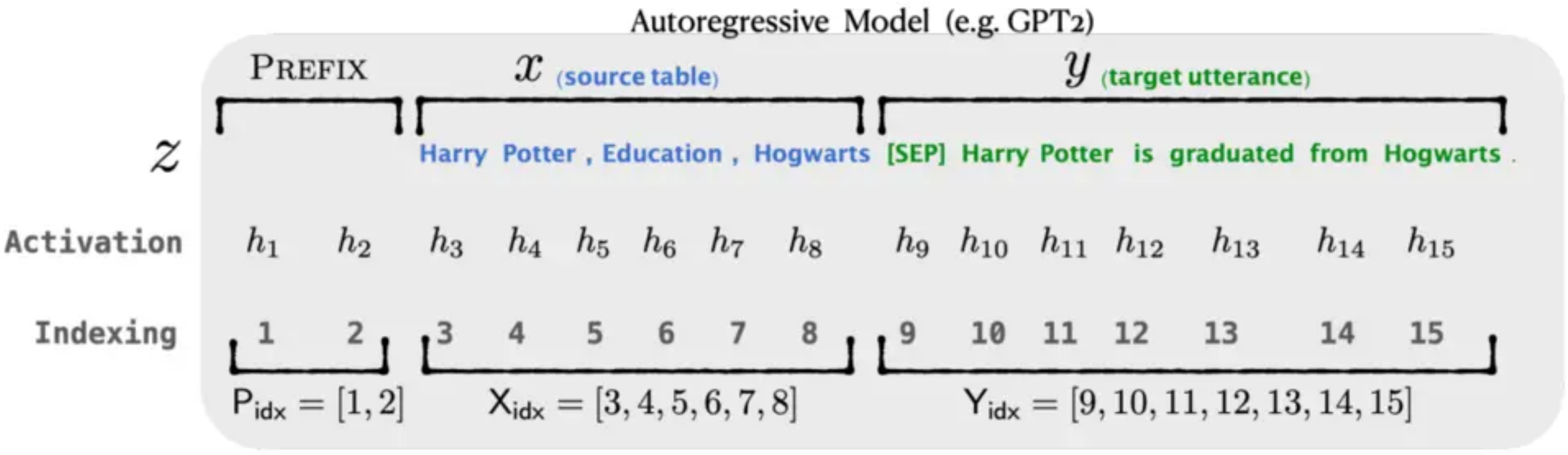
Prefix Tuning

- Alternative approach to adapters for parameter-efficient fine-tuning
- Suppose you have a task with inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$ (i.e text summarization)
- Could use in-context learning to do this, but:
  - Require coming up with prompt that the model can reliably follow
  - Hard to optimize prompts
    - Optimization over discrete tokens computationally challenging in general

Prefix-Tuning: Optimizing Continuous Prompts for Generation (Li and Liang, 2021)

Prefix Tuning

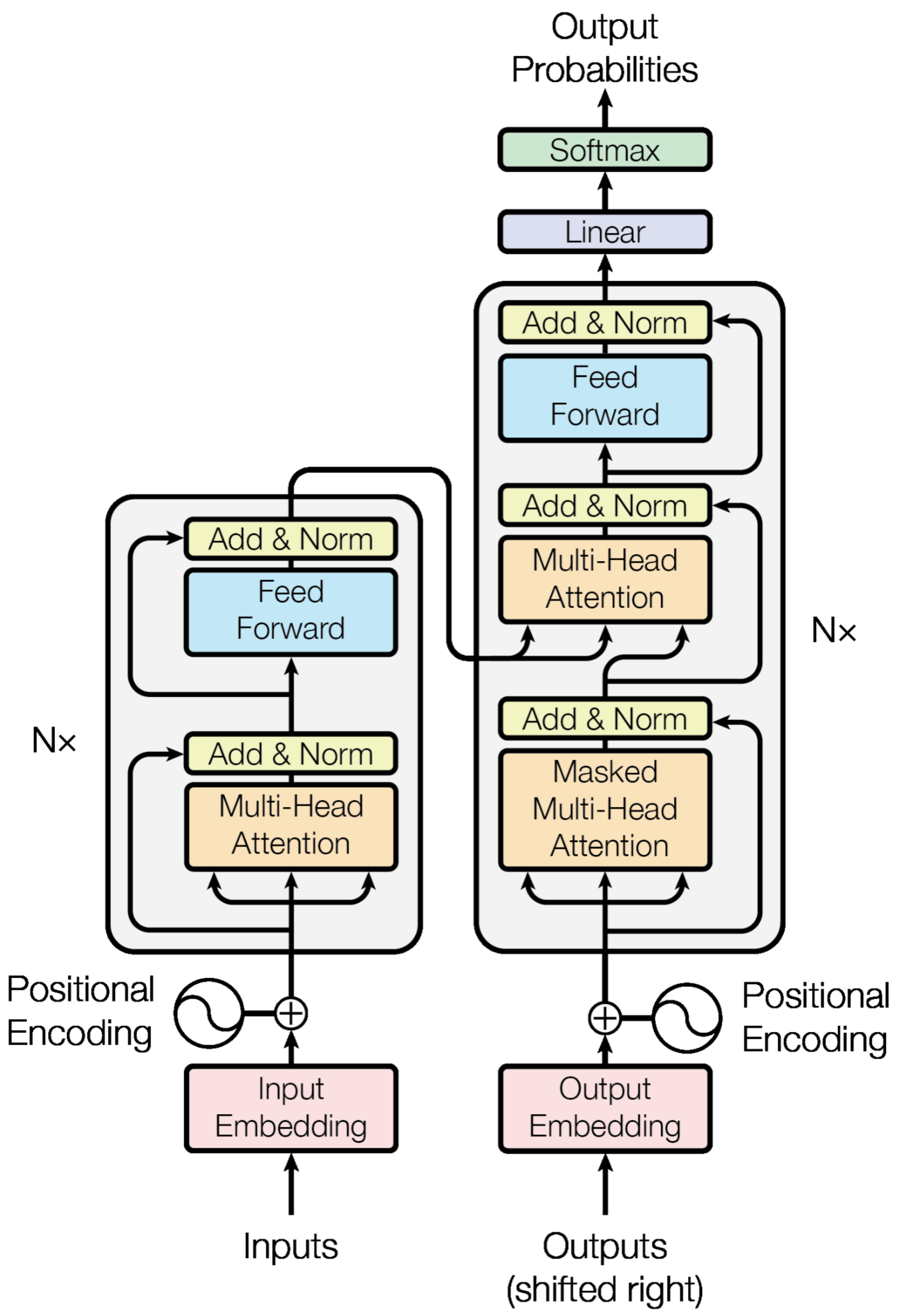
- In prefix-tuning, you add a prefix of activations to all Transformer layers

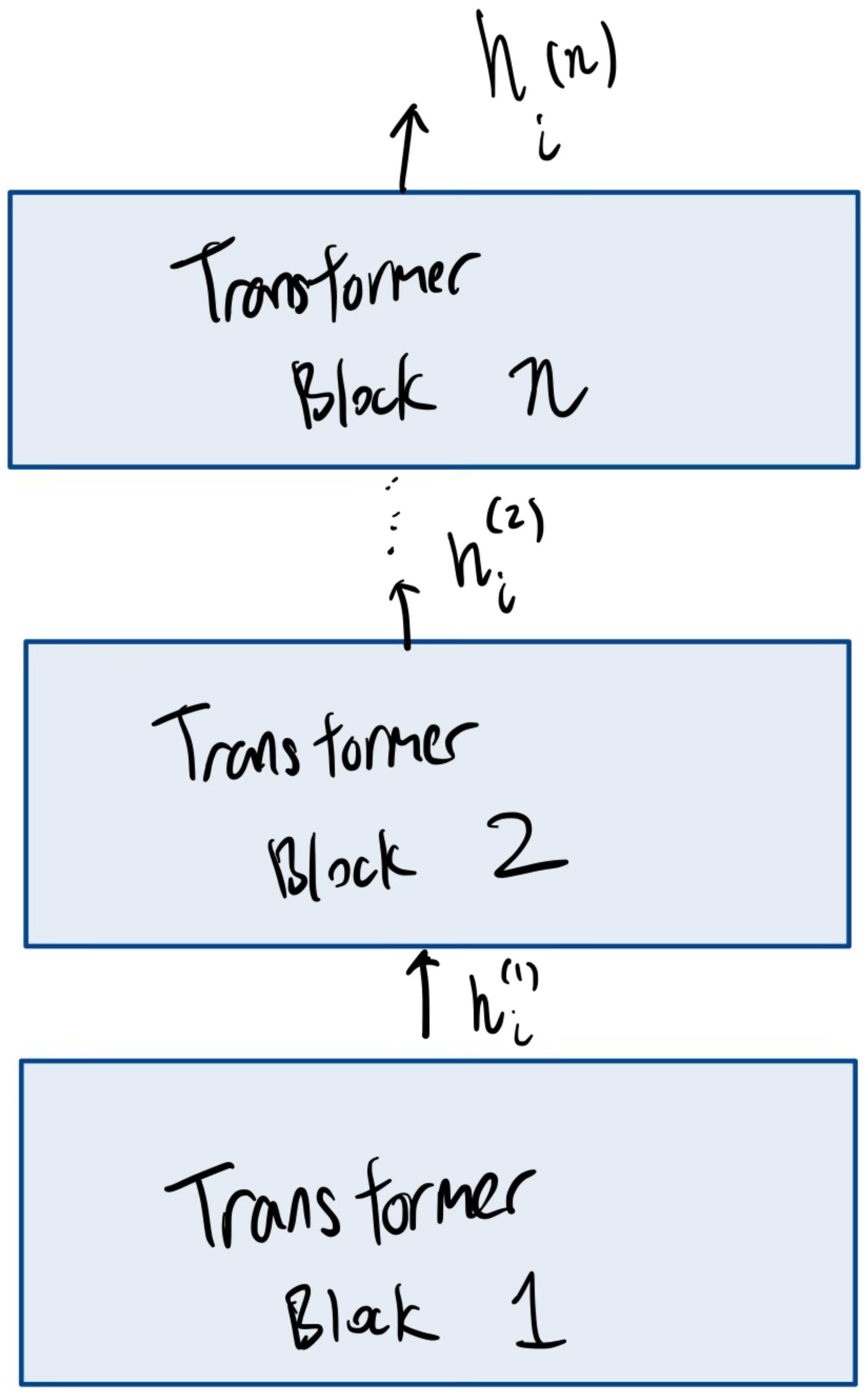


Prefix-Tuning: Optimizing Continuous Prompts for Generation (Li and Liang, 2021)

Learning Hidden Activations

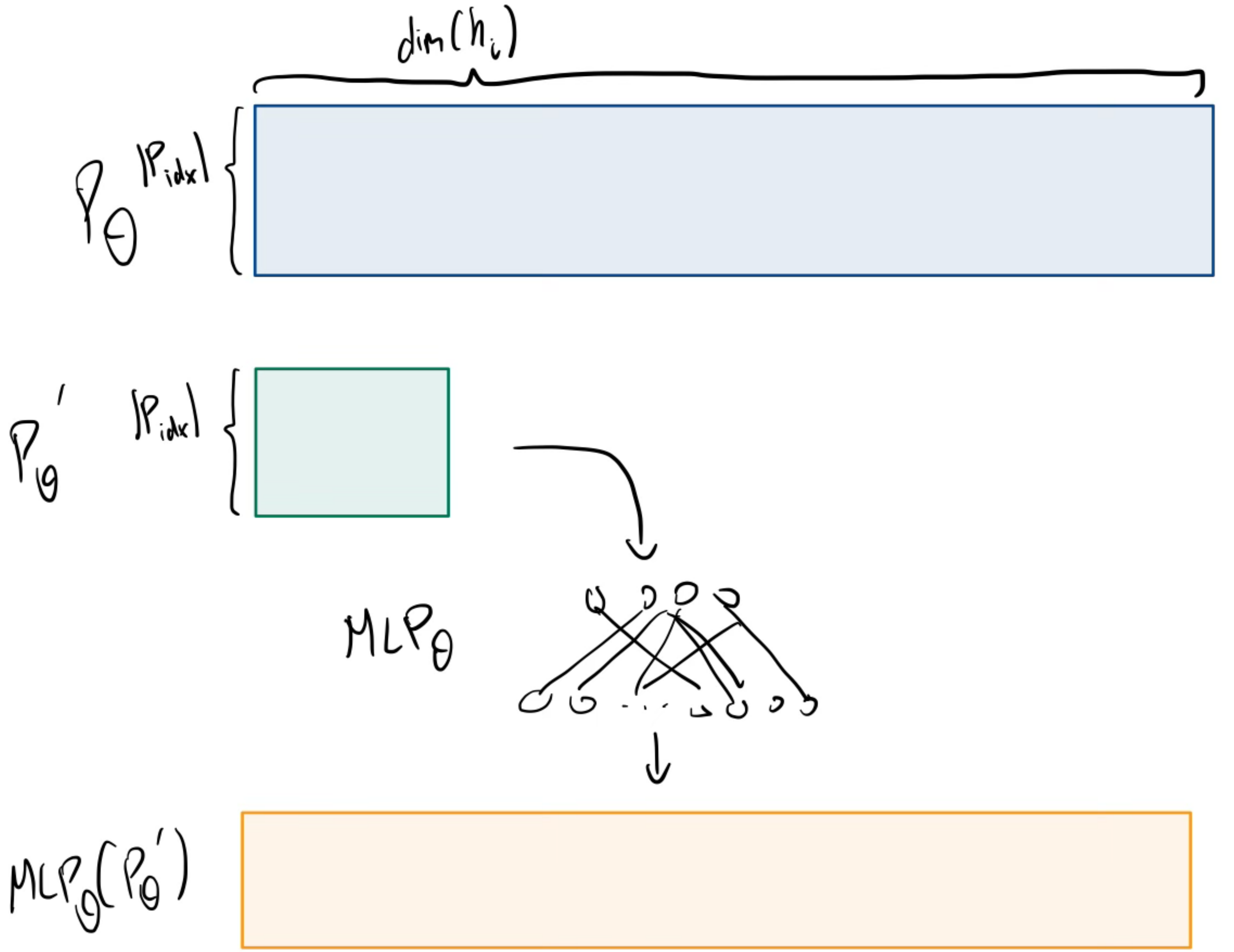
- For each of the prefix indices $i \in \mathrm{P}_{\mathrm{idx}}$, want to learn activations $h_i^{(1)}, h_i^{(2)}, \cdots, h_i^{(n)}$





Prefix-Tuning: Optimizing Continuous Prompts for Generation (Li and Liang, 2021)

Training Stability Mitigation



Prefix-Tuning: Optimizing Continuous Prompts for Generation (Li and Liang, 2021)

Prompt Tuning

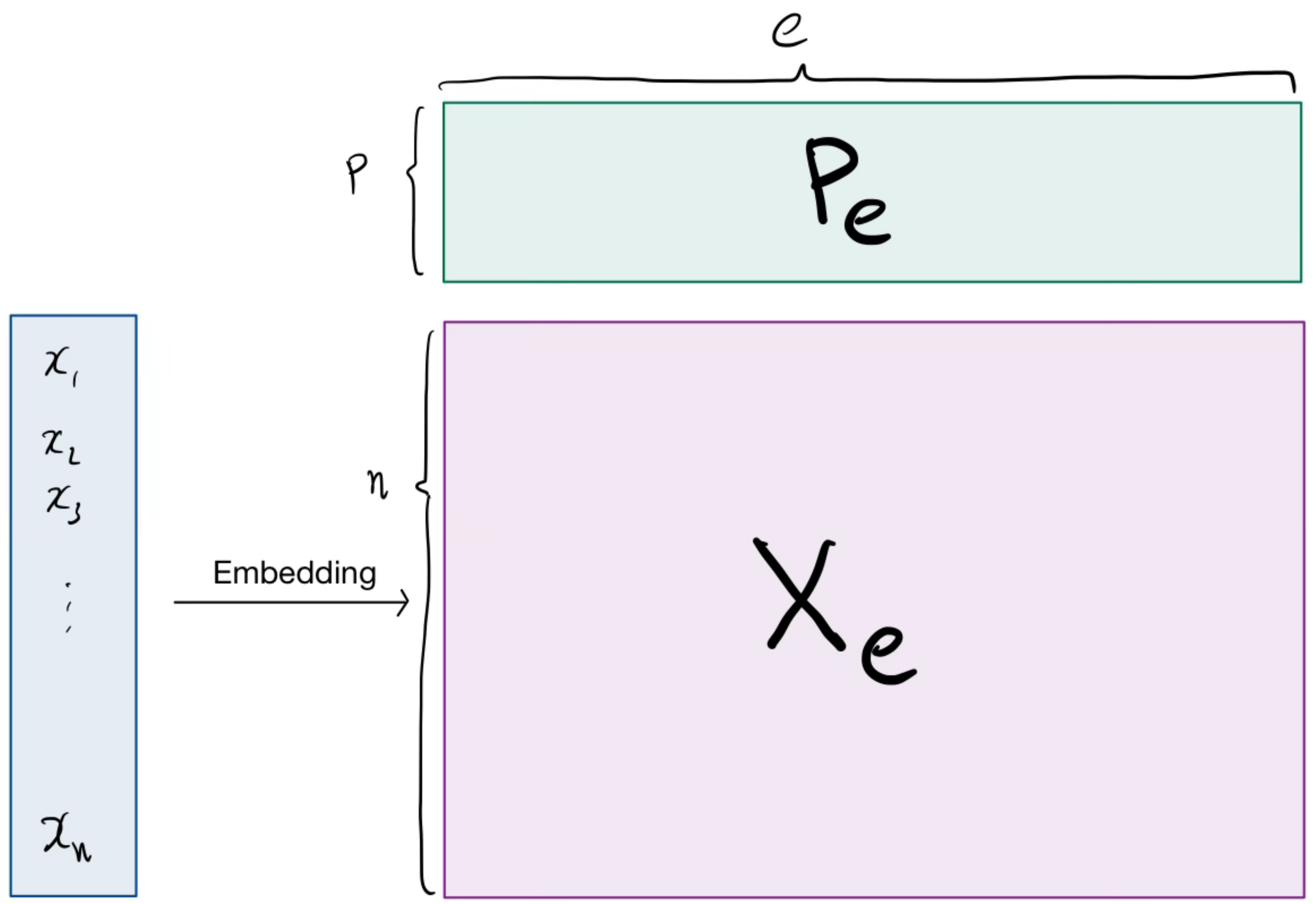
Prompt Tuning

- Concurrent work with prefix-tuning
- Instead of learning a prefix of activations, you learn a prefix of “soft prompts”
- In normal prompting: provide series of tokens which are embedded into vectors
- Soft prompting: learn a length-p prefix of embeddings

The Power of Scale for Parameter-Efficient Prompt Tuning (Lester, Al-Rfou, and Constant, 2021)

Prompt Tuning

- Input tokens $x_1, \cdots, x_n$ embedded into matrix $X_e \in \mathbb{R}^{n \times e}$
- Learn length p soft prompt $P_e \in \mathbb{R}^{p \times e}$
- Transformer input: concatenated input $[P_e; X_e]$
- Learn P_e by backpropagation



The Power of Scale for Parameter-Efficient Prompt Tuning (Lester, Al-Rfou, and Constant, 2021)

Conclusion

Summary

- Techniques to reduce parameters required for fine-tuning:
  - Adapter-based
  - Prefix/prompt tuning
- Techniques to further reduce memory requirement:
  - Quantization