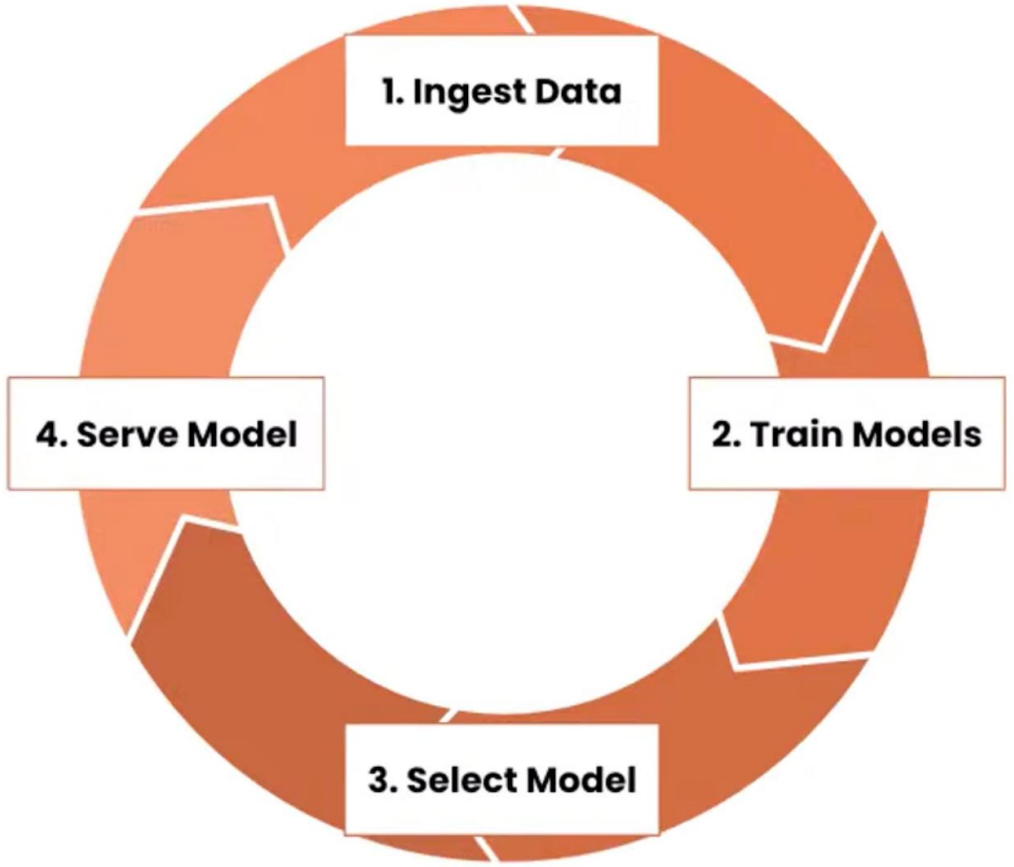
Some Notes on LLMs in the Wild

Igor Kotenkov

@stm

Here's our plan...

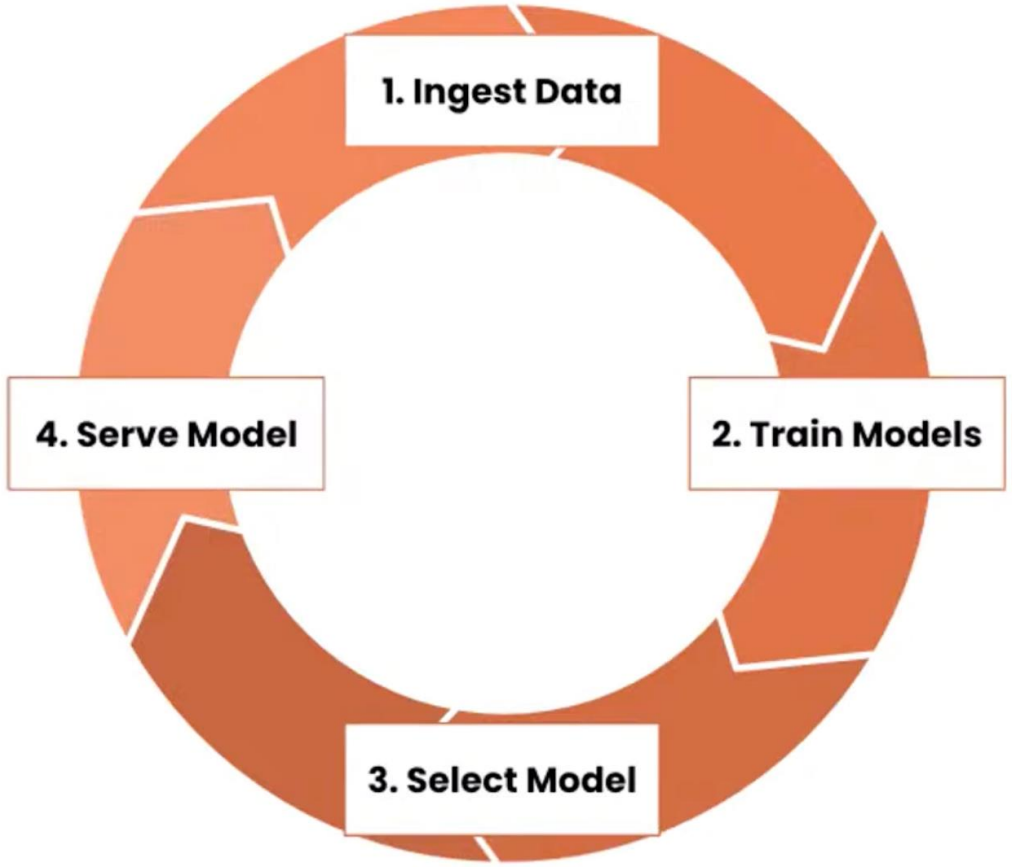
The (Old) ML Product Lifecycle



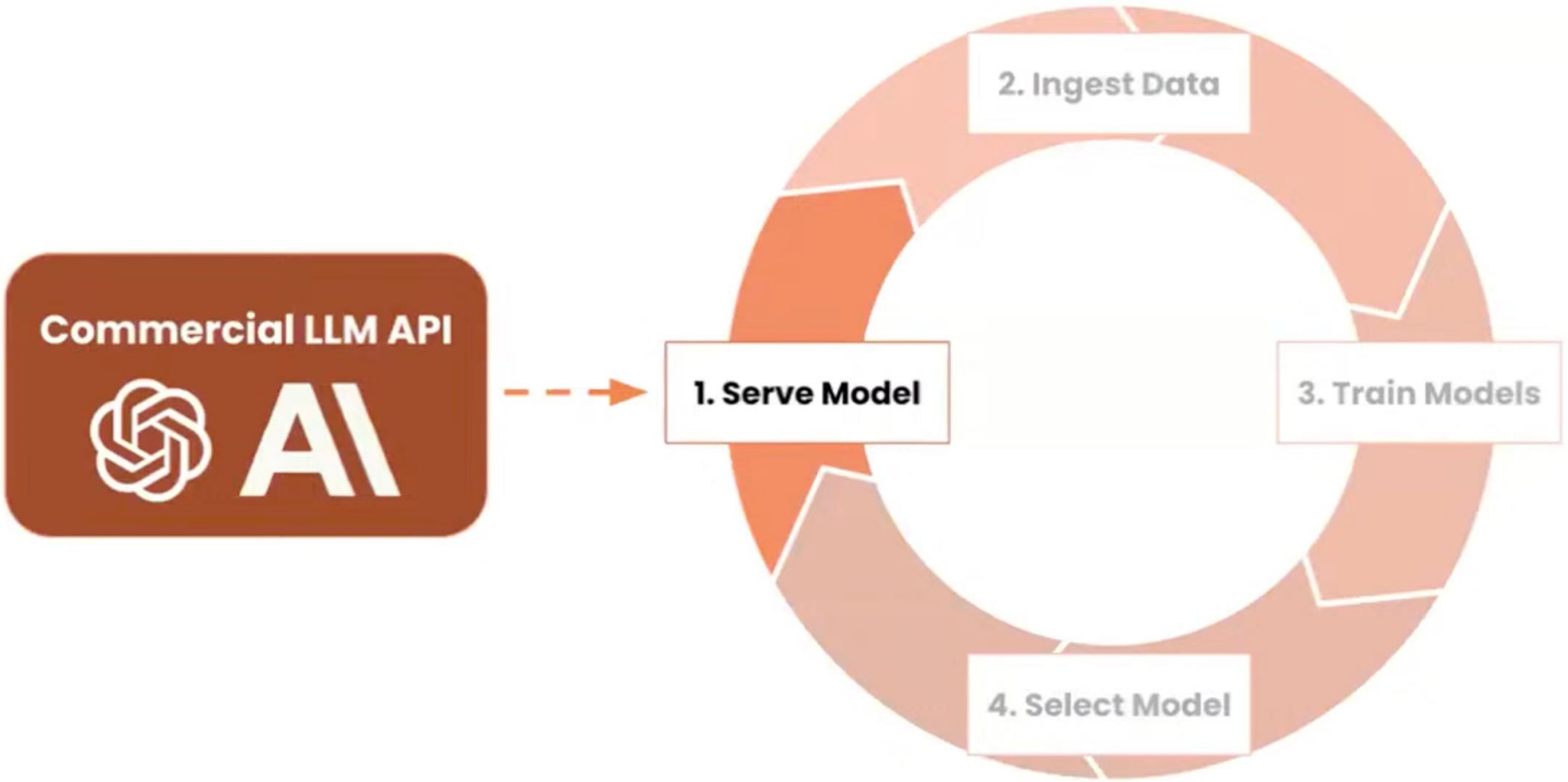
The (Old) ML Product Lifecycle

For the first iteration, lots of effort goes on:

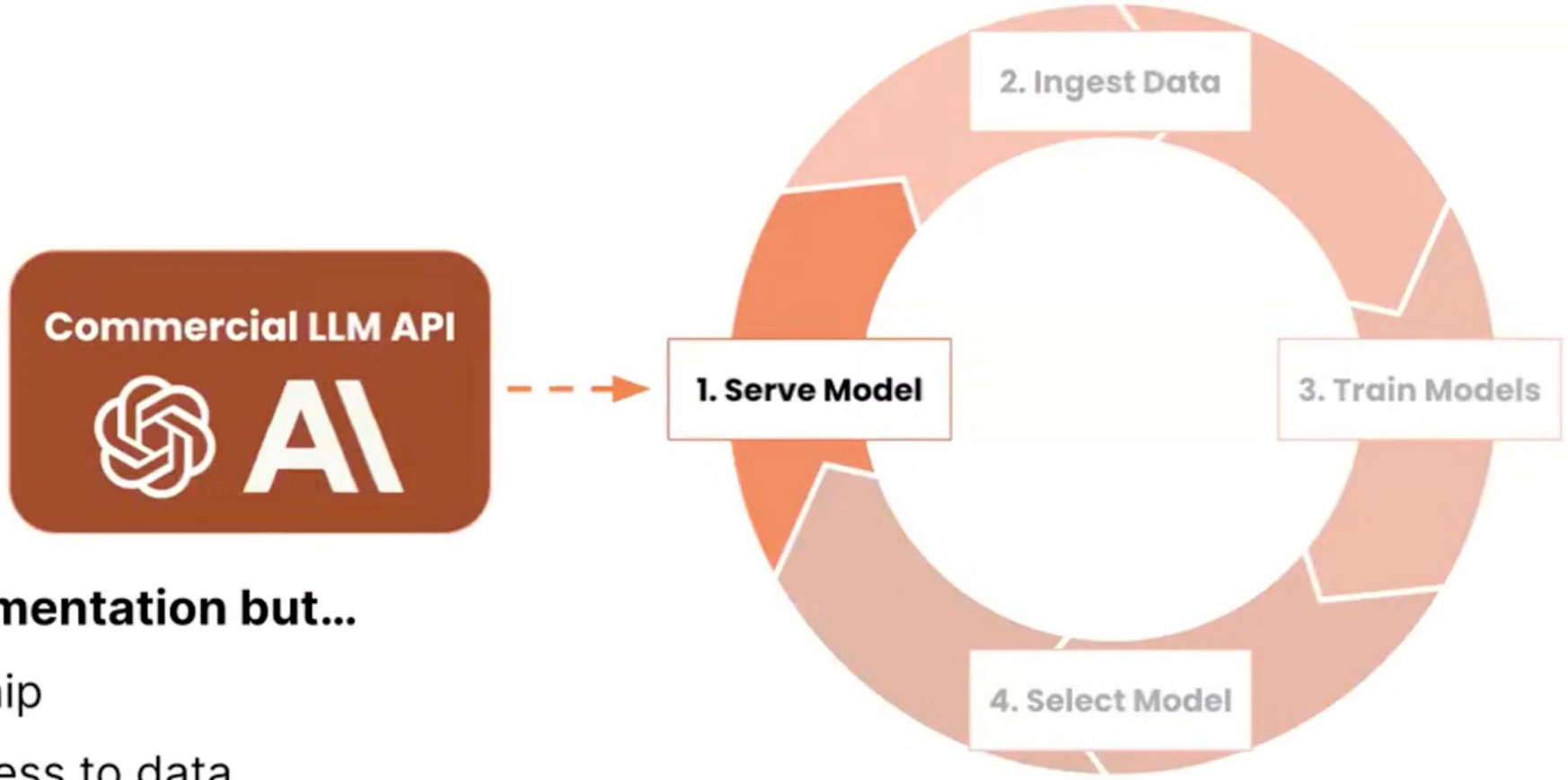
- Data preparation
- Hyperparameter selection
- Deploying and monitoring



The (New) ML Product Lifecycle



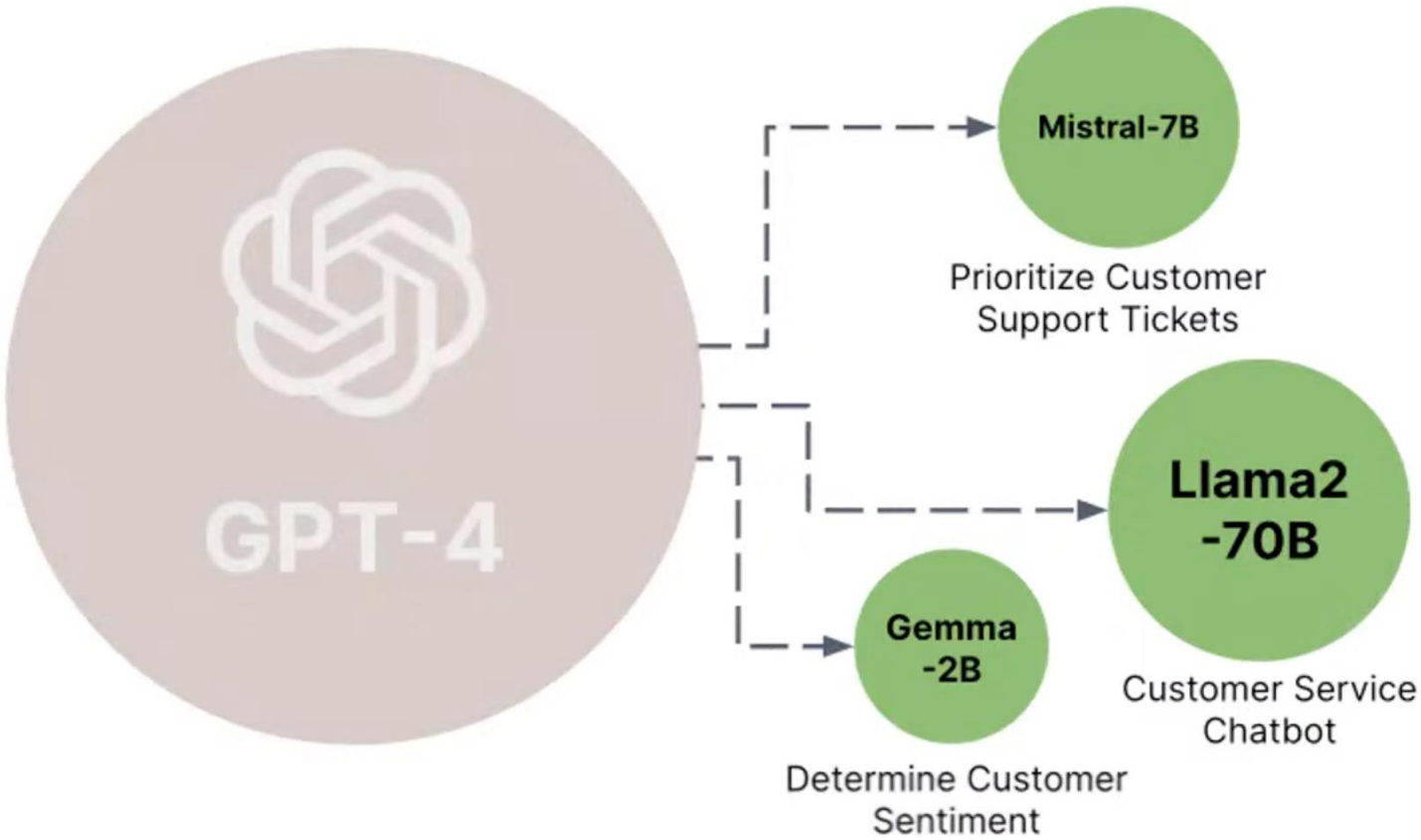
The (New) ML Product Lifecycle



Great for rapid experimentation but...

- ✘ Lack model ownership
- ✘ Need to give up access to data
- ✘ Too slow, expensive & overkill for many tasks

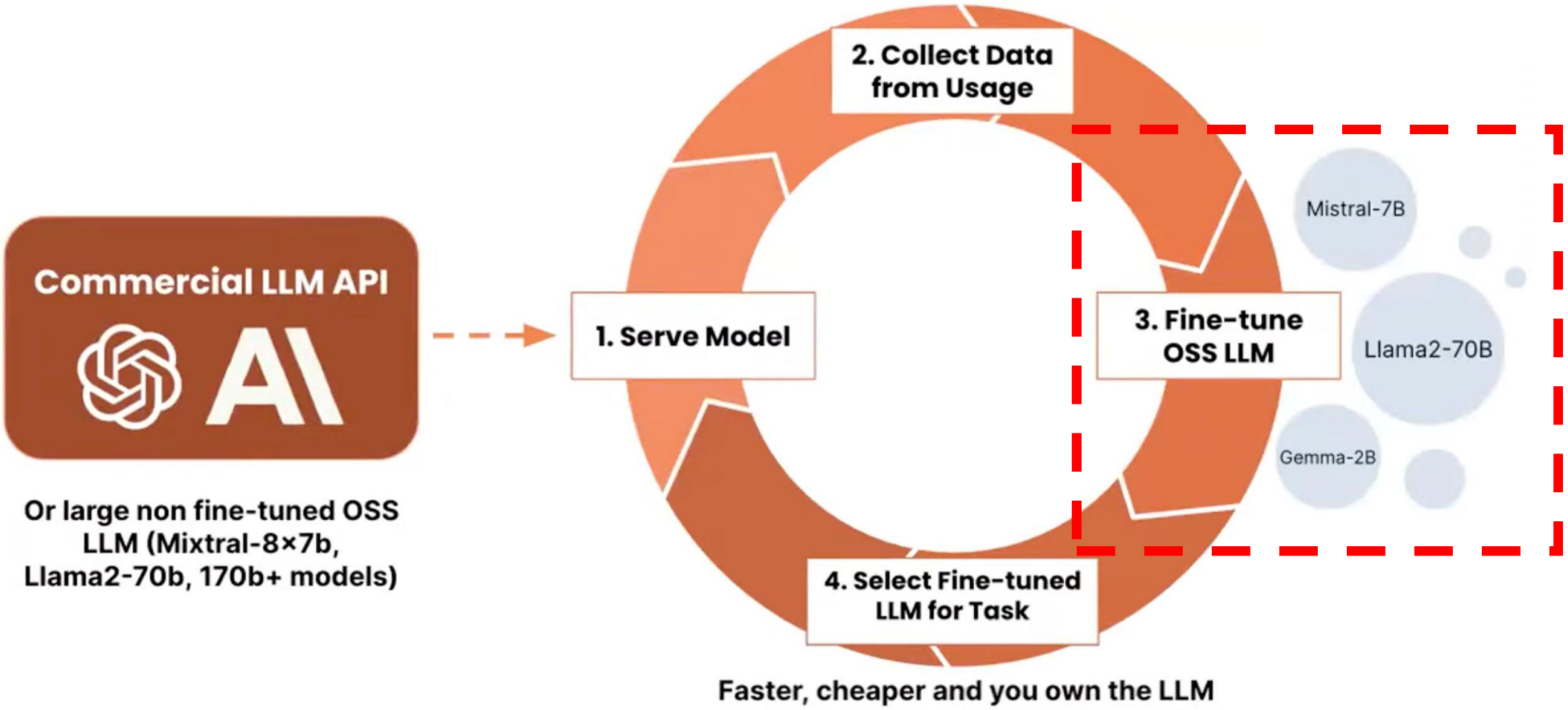
The (New) ML Product Lifecycle



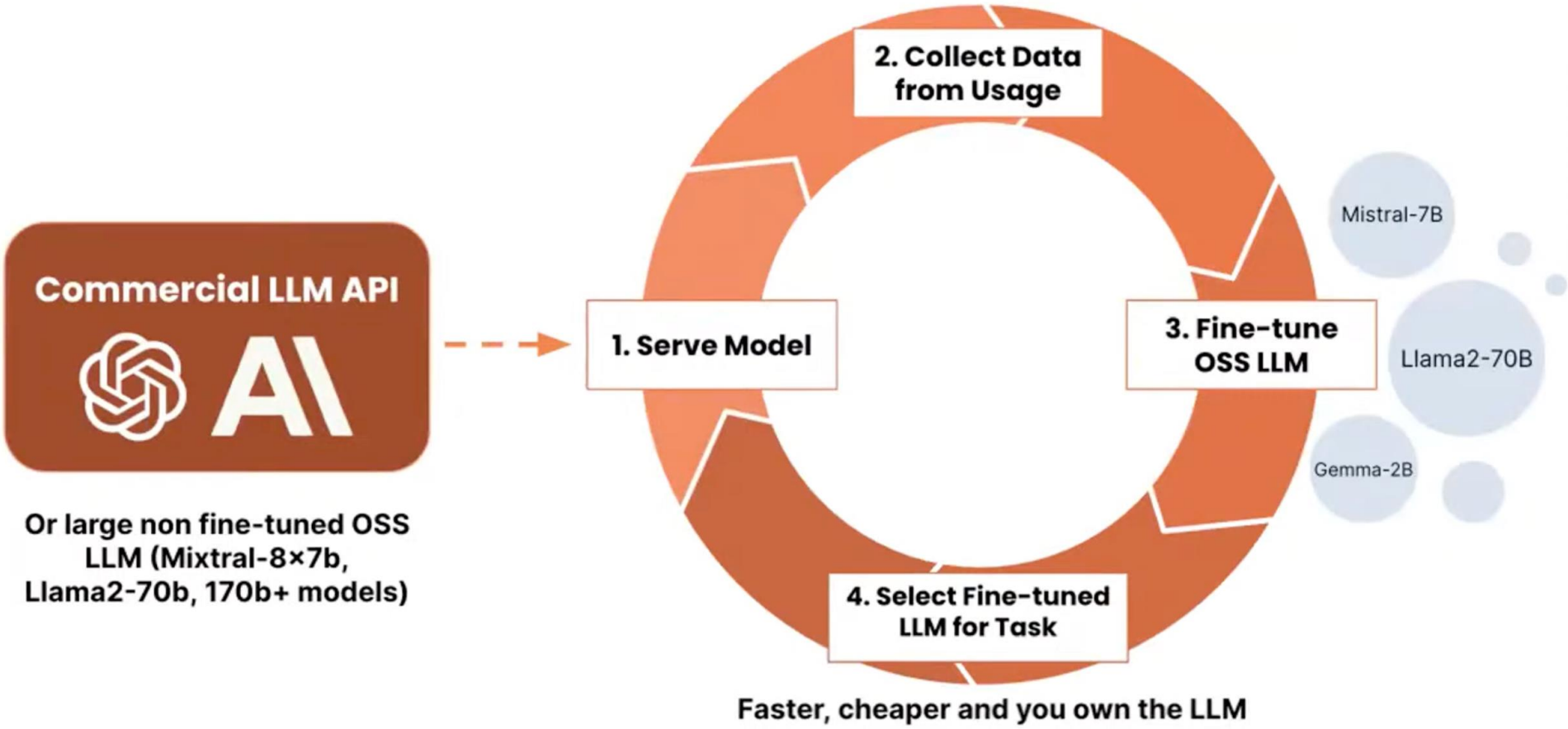
Benefits of smaller task-specific LLMs

- ✔ Own your models
- ✔ Don’t share data
- ✔ Smaller and faster
- ✔ Control the output

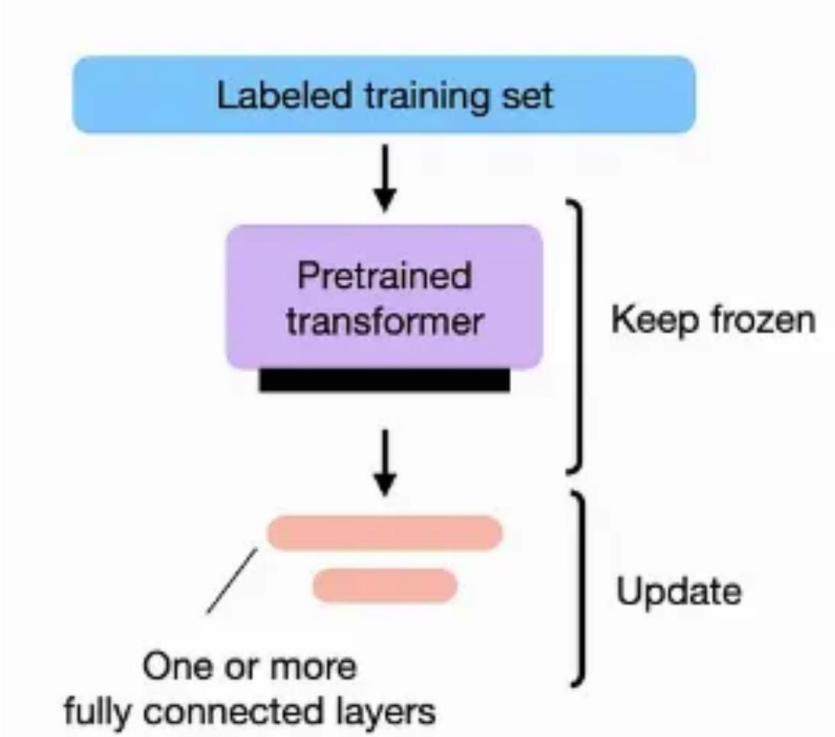
The (New) ML Product Lifecycle



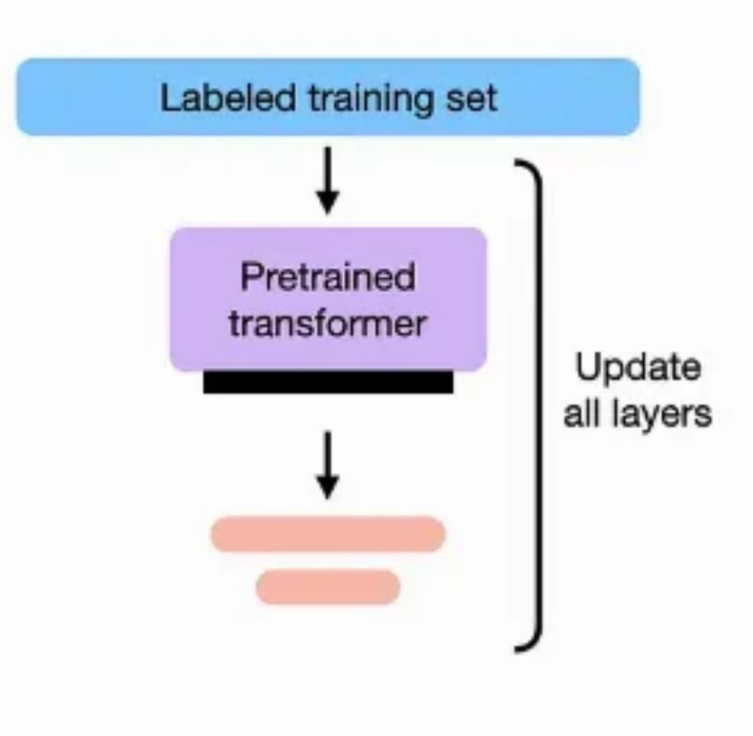
The (New) ML Product Lifecycle



Basic Finetuning Approach

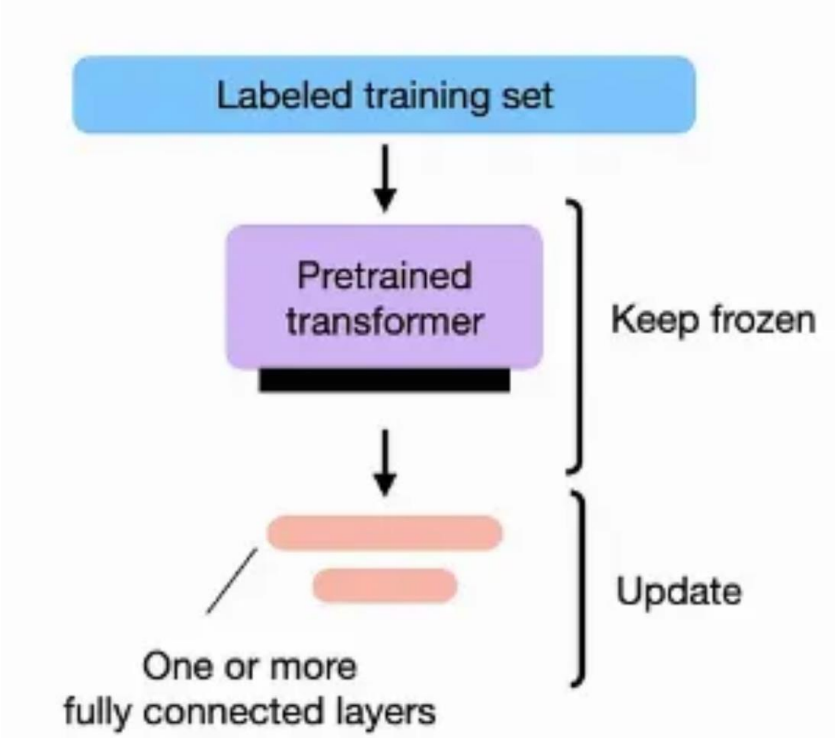


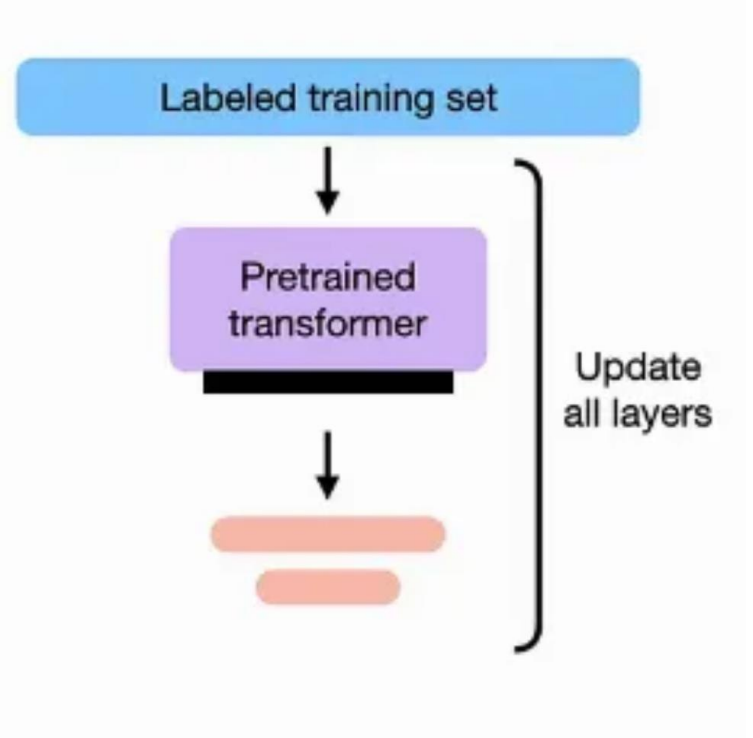
Tune «The Head»



Tune the whole model

Basic Finetuning Approach





Tune «The Head»

Changes almost nothing

Tune the whole model

Might be too expensive or cause overfitting

Why and how espensive, exaclty?

Finetuning memory cost, per parameter:

- The weight itself: 4 bytes

Why and how espensive, exaclty?

Finetuning memory cost, per parameter:

- The weight itself: 4 bytes
- The weight's gradient: 4 bytes

Why and how espensive, exaclty?

Finetuning memory cost, per parameter:

- The weight itself: 4 bytes
- The weight's gradient: 4 bytes
- Optimizer state: 4+4 bytes

Initialize a parameter state

- `state` is the optimizer state of the parameter (tensor)
- `group` stores optimizer attributes of the parameter group
- `param` is the parameter tensor θ_{t-1}

```
83    def init_state(self, state: Dict[str, any], group: Dict[str, any], param: nn.Parameter):
```

This is the number of optimizer steps taken on the parameter, t

```
93        state['step'] = 0
```

Exponential moving average of gradients, m_t

```
95        state['exp_avg'] = torch.zeros_like(param, memory_format=torch.preserve_format)
```

Exponential moving average of squared gradient values, v_t

```
97        state['exp_avg_sq'] = torch.zeros_like(param, memory_format=torch.preserve_format)
```

Great website: https://nn.labml.ai/optimizers/adam.html

Why and how espensive, exaclty?

Finetuning memory cost, per parameter:

- The weight itself: 4 bytes
- The weight's gradient: 4 bytes
- Optimizer state: 4+4 bytes

16 bytes per parameter!
70B model -> 1.12 TB of GPU memory

Initialize a parameter state

- `state` is the optimizer state of the parameter (tensor)
- `group` stores optimizer attributes of the parameter group
- `param` is the parameter tensor θ_{t-1}

This is the number of optimizer steps taken on the parameter, t

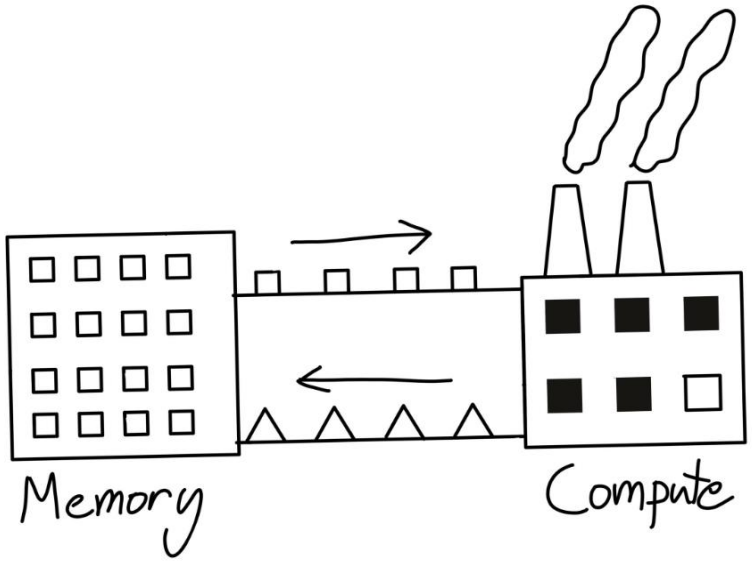
Exponential moving average of gradients, m_t

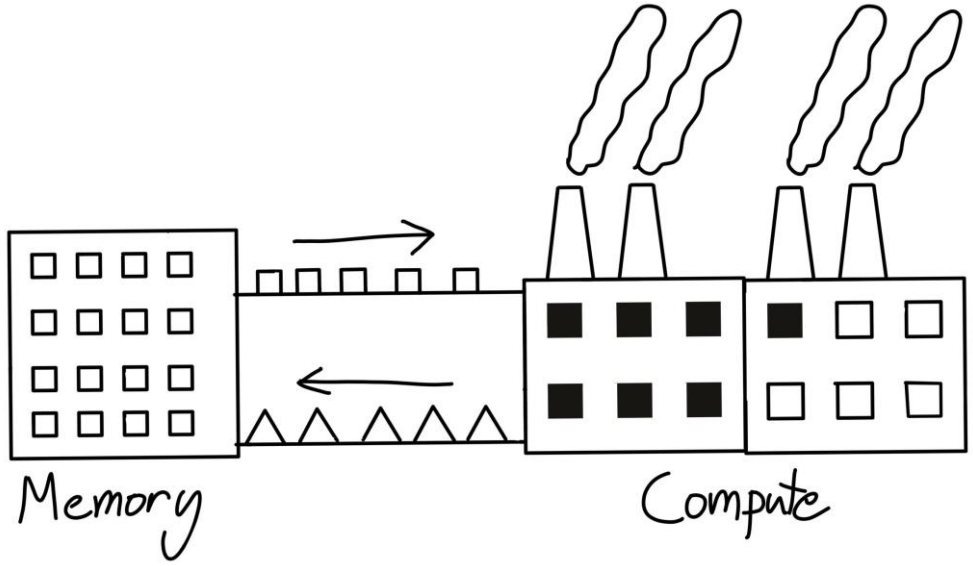
Exponential moving average of squared gradient values, v_t

```
83    def init_state(self, state: Dict[str, any], group: Dict[str, any], param: nn.Parameter):
93        state['step'] = 0
95        state['exp_avg'] = torch.zeros_like(param, memory_format=torch.preserve_format)
97        state['exp_avg_sq'] = torch.zeros_like(param, memory_format=torch.preserve_format)
```

Great website: https://nn.labml.ai/optimizers/adam.html

Why do we care about memory?



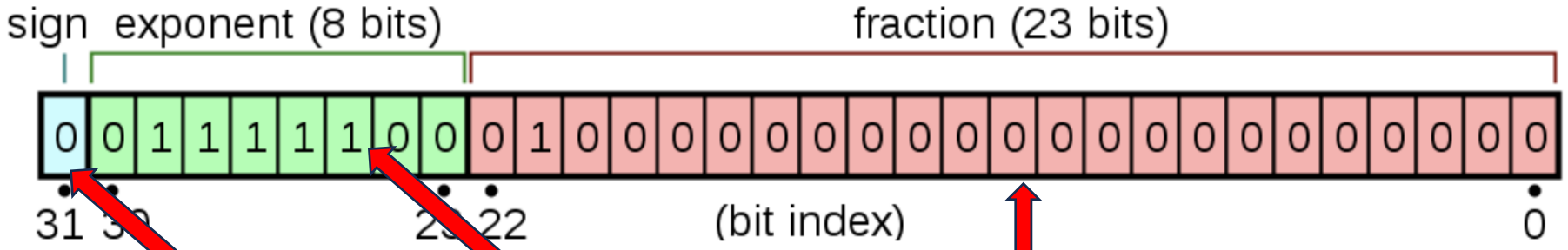


Why do we care about memory?

Attention	Standard	FLASHATTENTION
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

Read more: HERE and HERE

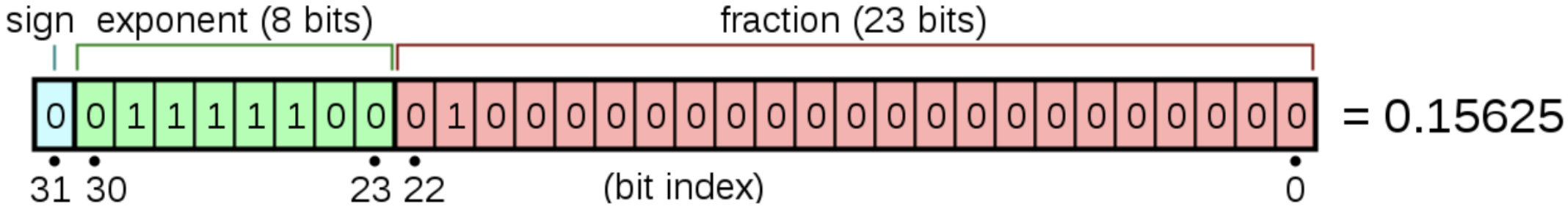
Optimization Step 1: Lower Precision



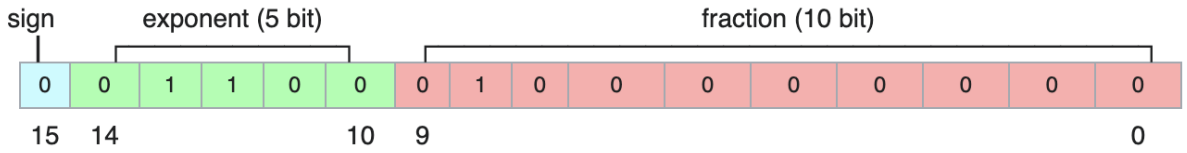
$$(-1)^{\text{sign}} \times 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right) = \text{value}$$

$$(+1) \times 2^{-3} \times 1.25 = +0.15625$$

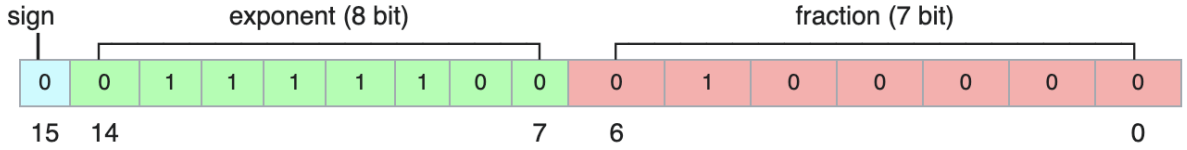
Optimization Step 1: Lower Precision



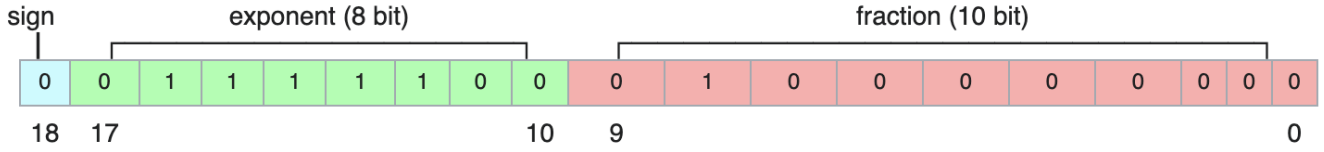
IEEE half-precision 16-bit float



bfloat16

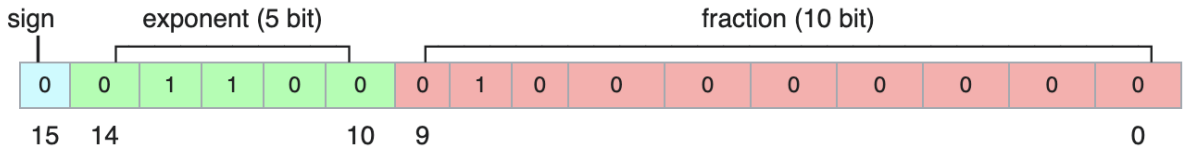


NVidia's TensorFloat (19 bits)

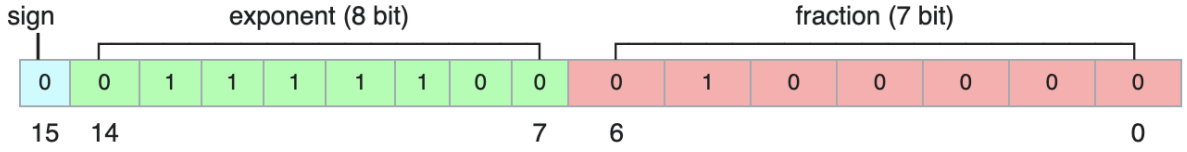


Optimization Step 1: Lower Precision

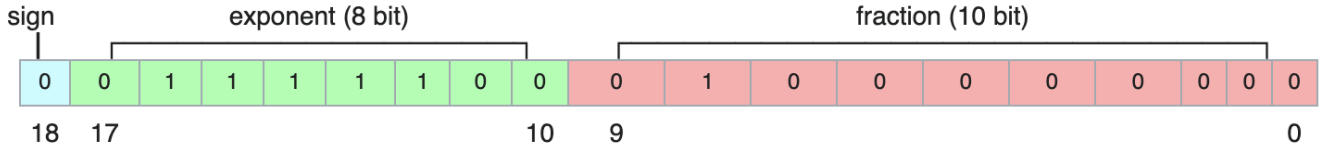
IEEE half-precision 16-bit float



bfloat16

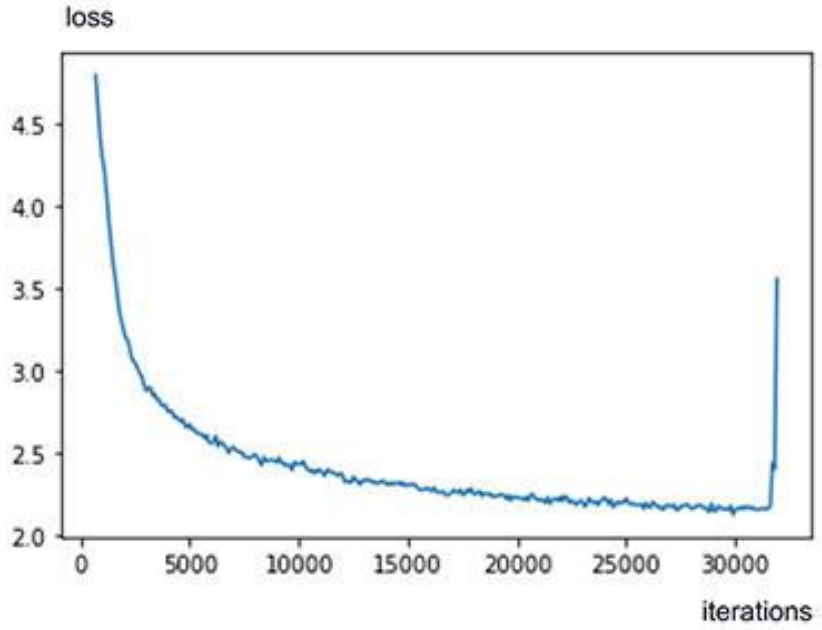


NVidia's TensorFloat (19 bits)



Mixed Precision:

- FP32 + FP/BF16
- FP/BF16 + FP8



Optimization Step 1: Lower Precision

https://habr.com/ru/companies/yandex/articles/672396/:

Чтобы компенсировать расхождения, мы стали вычислять следующие слои и операции в tf32 (или fp32 на старых карточках):

- Softmax в attention (вот и пригодились наши ядра), softmax по токенам перед лоссом.
- Все слои LayerNorm.
- Все операции с Residual — это позволило не накапливать ошибку и градиенты по глубине сети.
- all_reduce градиентов, о котором было написано раньше.

bf16 как основной тип для весов.

Вычисления, требующие точности, делаем в tf32.

Optimization Step 2: What to Train?

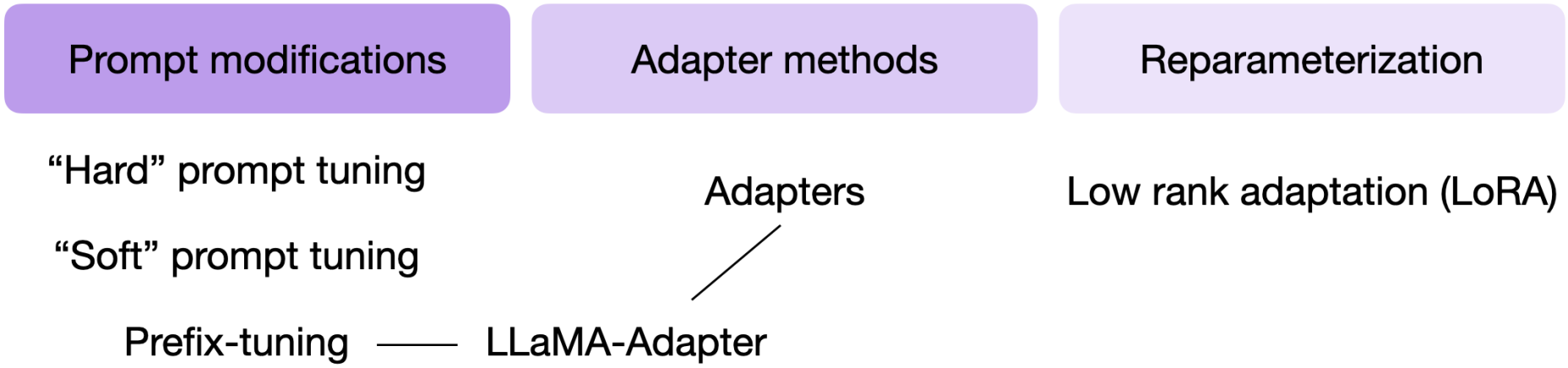
Finetuning memory cost, per parameter:

- The weight itself: 4 bytes
- The weight's gradient: 4 bytes
- Optimizer state: 4+4 bytes

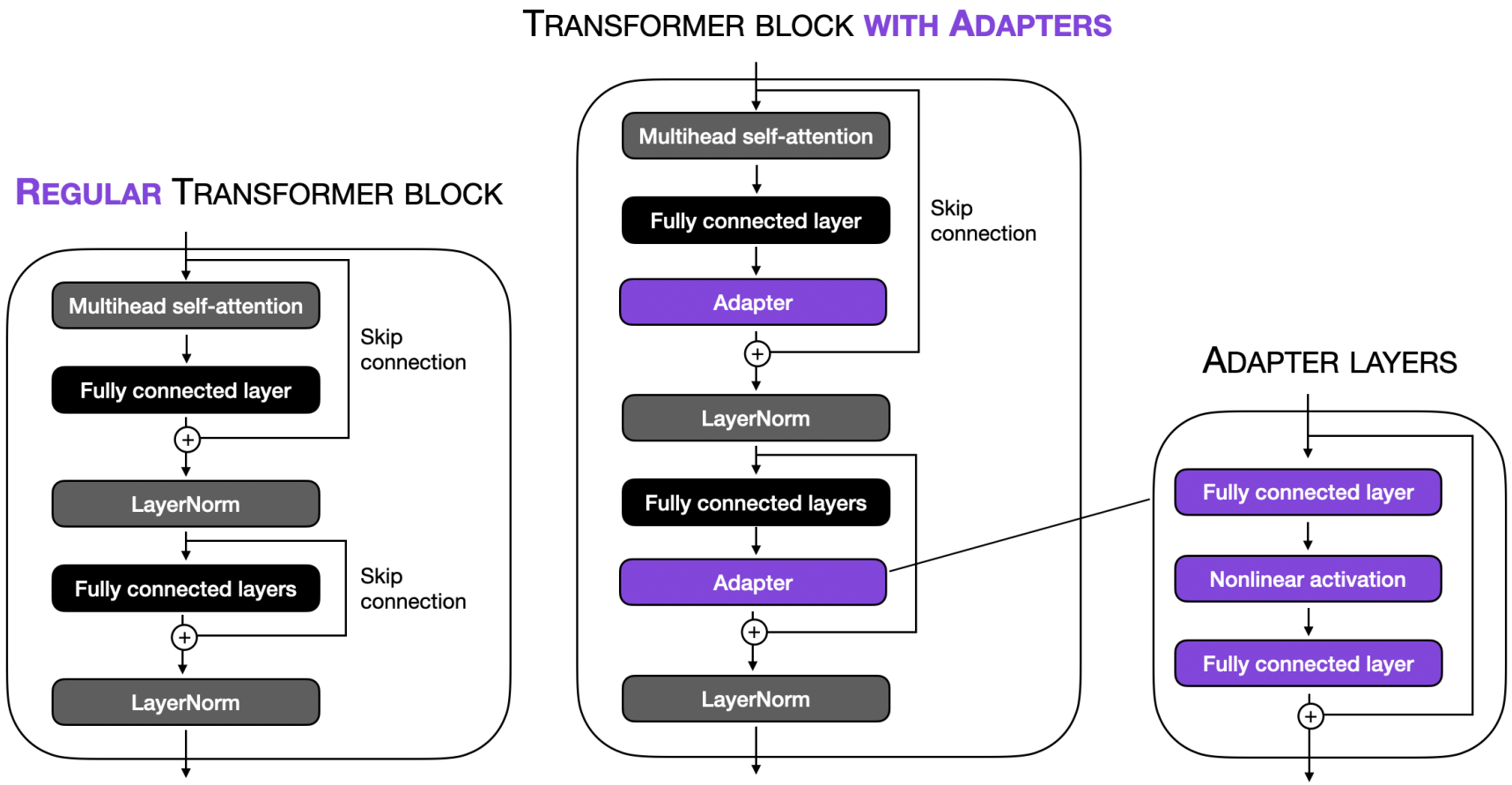
} Only for traiable!

Optimization Step 2: What to Train?

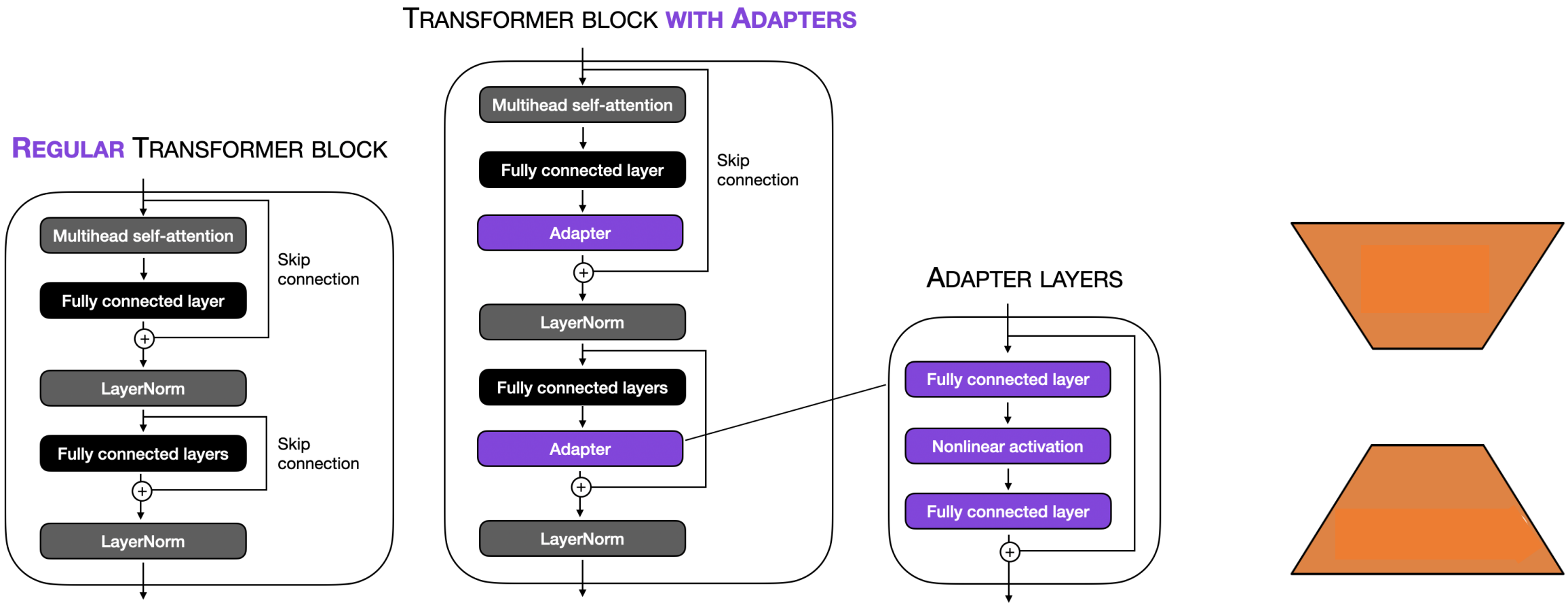
PEFT: **P**arameter-**E**fficient **F**ine-**T**uning



Optimization Step 2: Adapters



Optimization Step 2: Adapters

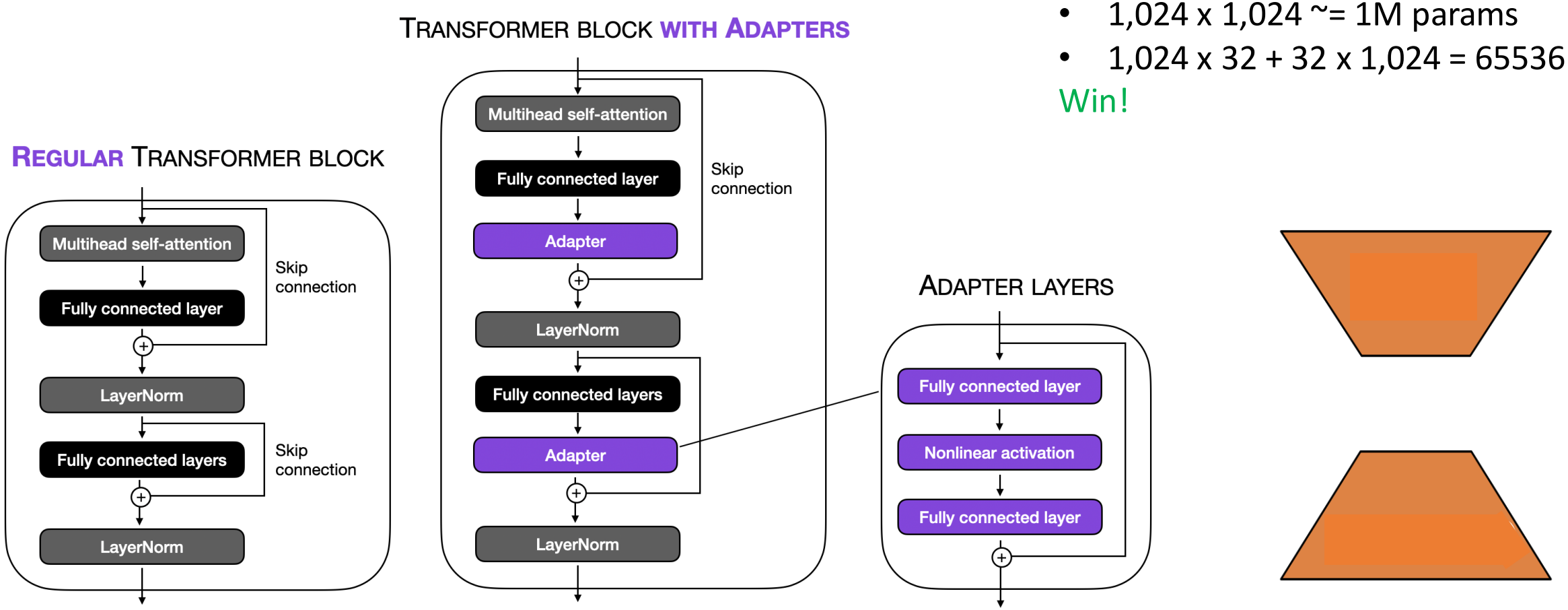


Optimization Step 2: Adapters

Let's assume d=1024, then:

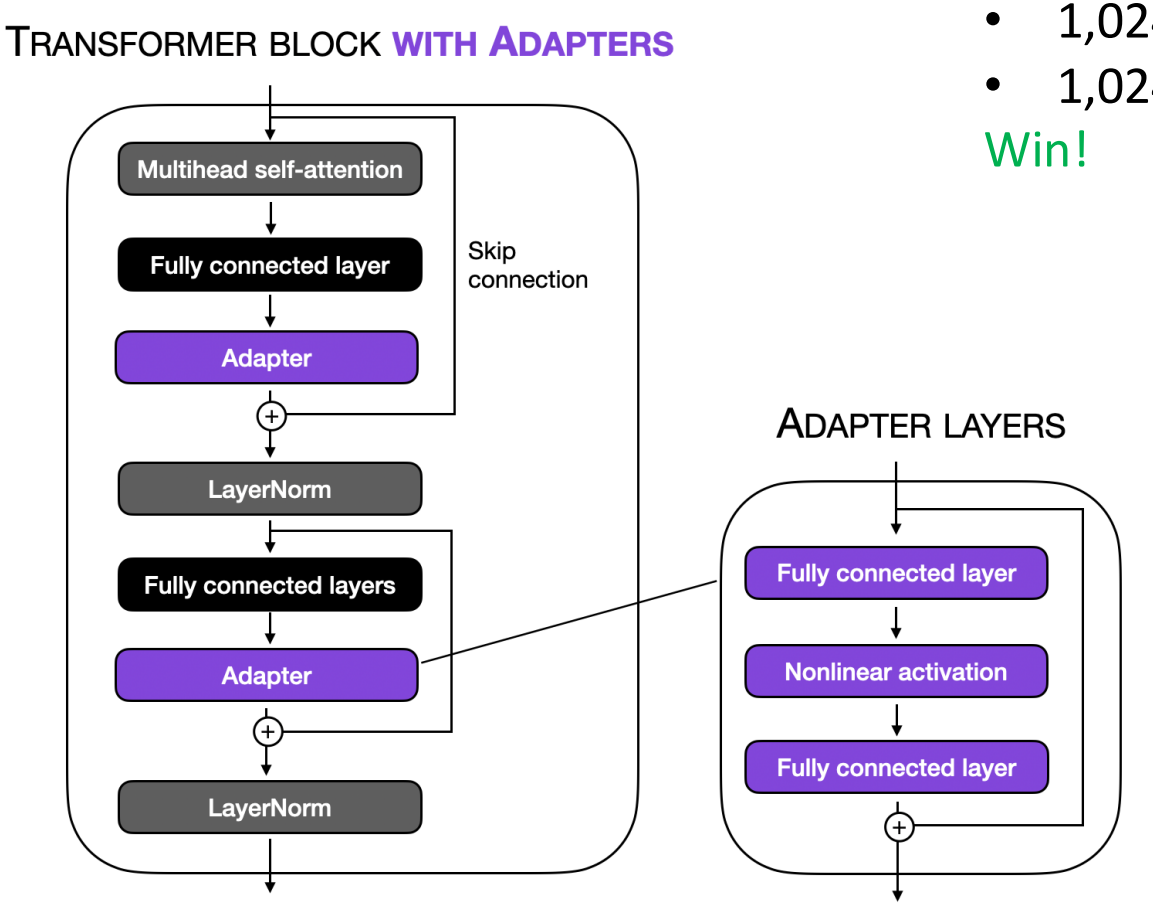
- 1,024 x 1,024 ~= 1M params
- 1,024 x 32 + 32 x 1,024 = 65536

Win!



Optimization Step 2: Adapters

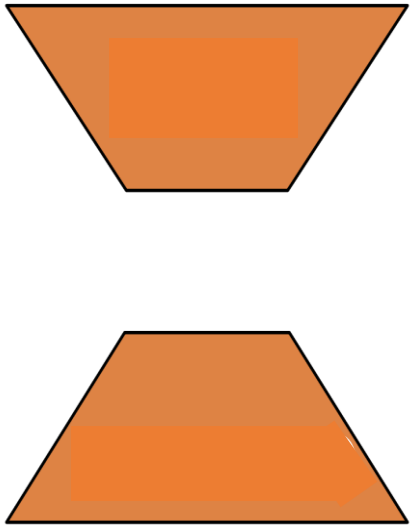
«a BERT model trained with the adapter method reaches a modeling performance comparable to a fully finetuned BERT model while only requiring the training of 3.6% of the parameters»



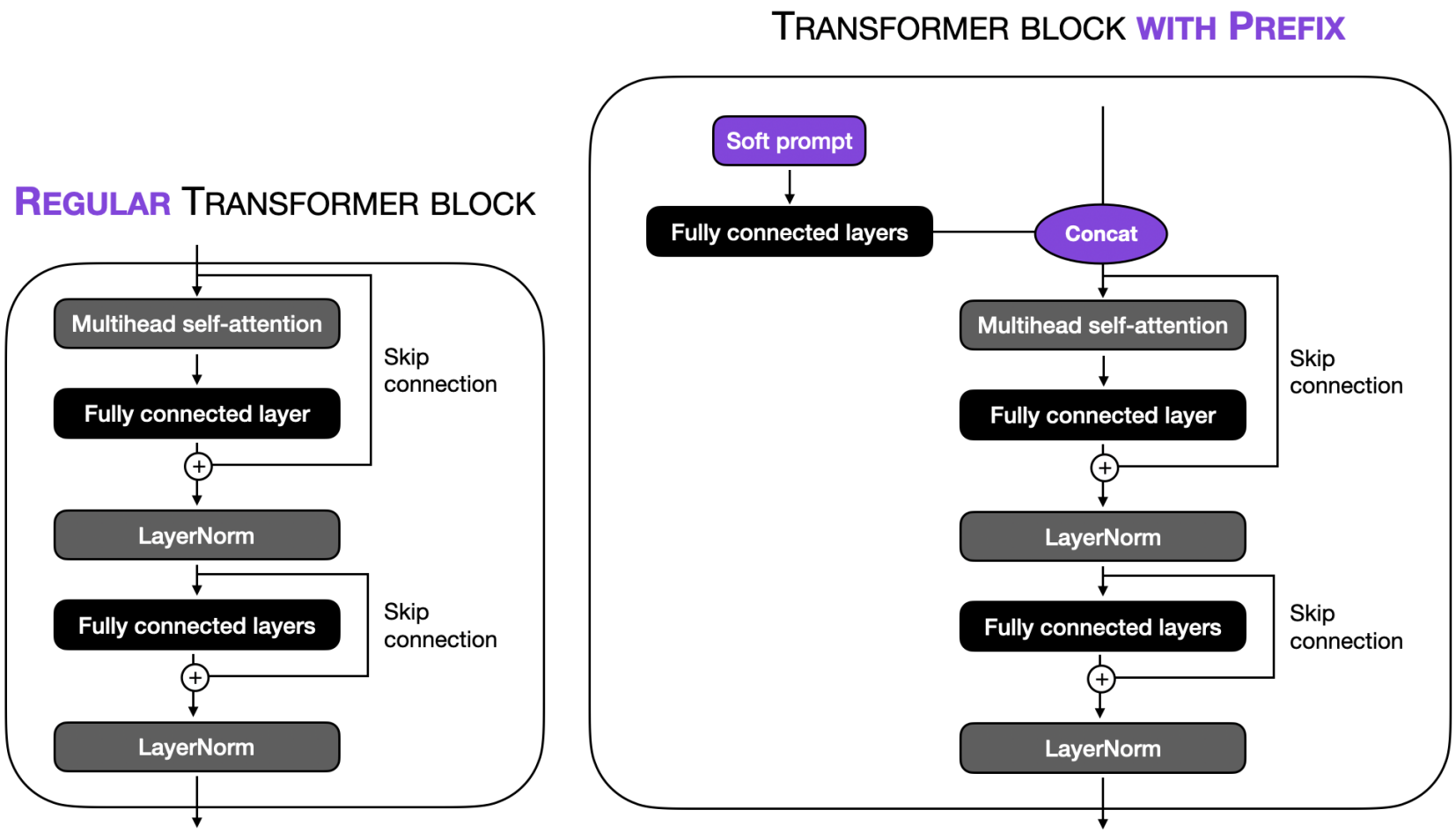
Let's assume d=1024, then:

- 1,024 x 1,024 ~= 1M params
- 1,024 x 32 + 32 x 1,024 = 65536

Win!

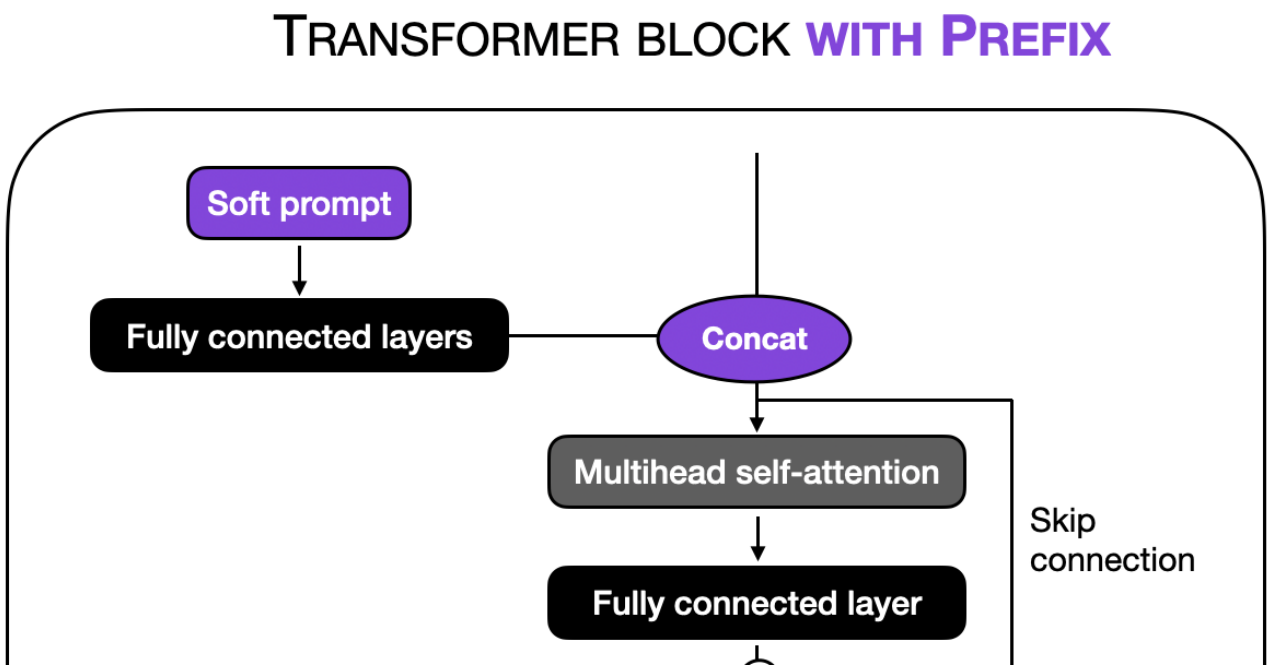


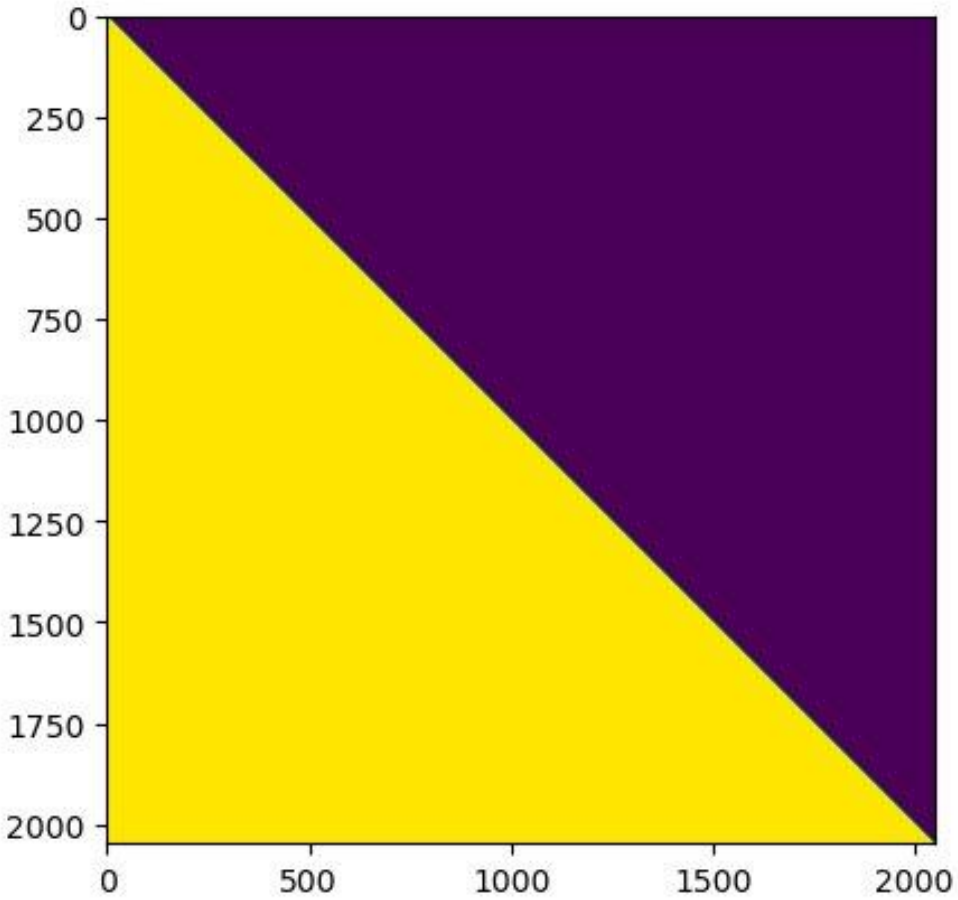
Optimization Step 2: Prefix Tuning



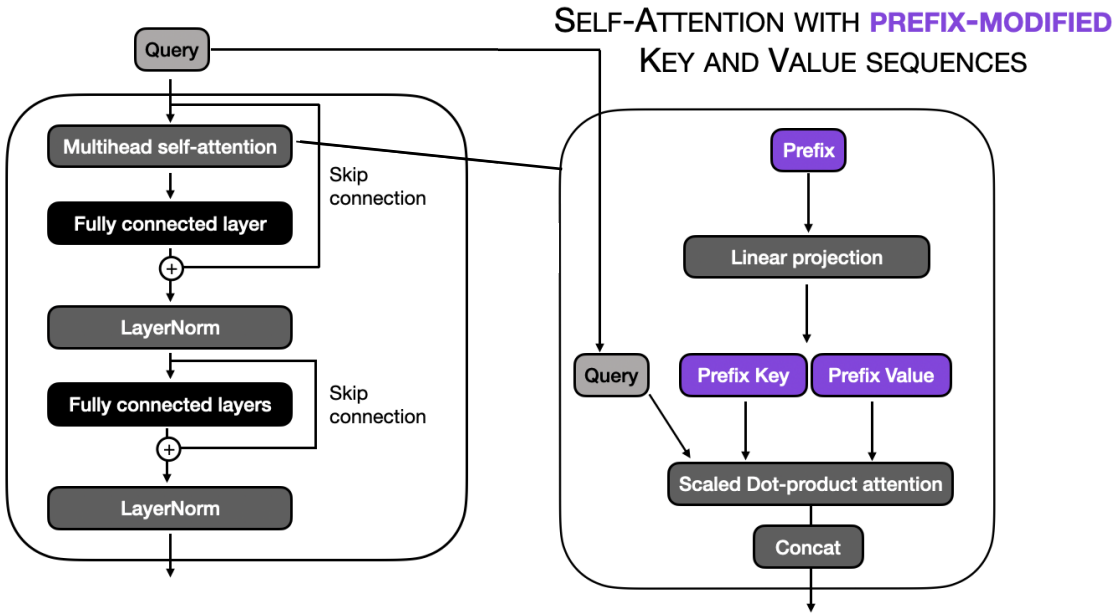
We can have multiple soft prompts, each corresponding to a different task, and provide the appropriate prefix during inference to achieve optimal results for a particular task

Optimization Step 2: Prefix Tuning





Optimization Step 2: Prefix Tuning

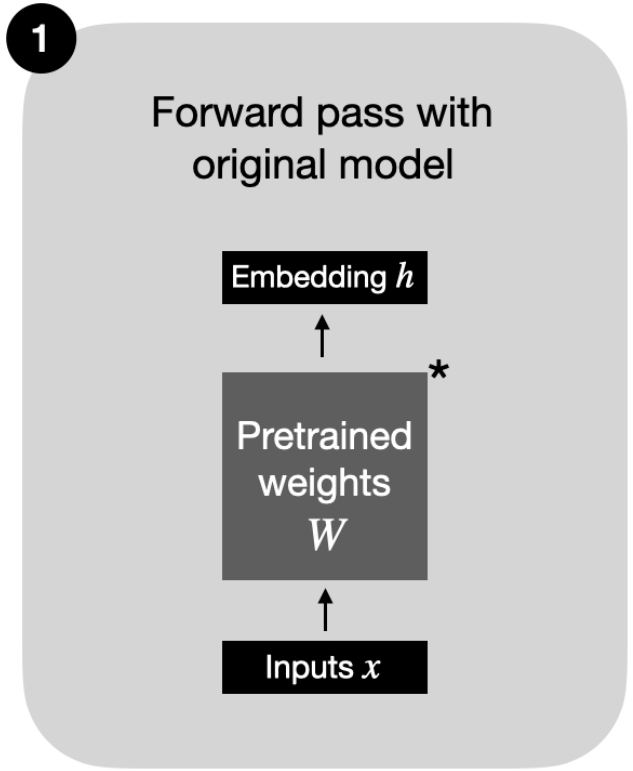


$$Q_l = \text{Linear}_\text{q}(\ t_l\ );$$
$$K_l = \text{Linear}_\text{k}(\ [P_l;\ T_l;\ t_l]\ );$$
$$V_l = \text{Linear}_\text{v}(\ [P_l;\ T_l;\ t_l]\ ).$$

$$[P_l;\ T_l]\ \in \mathbb{R}^{(K+M)\times C}.$$

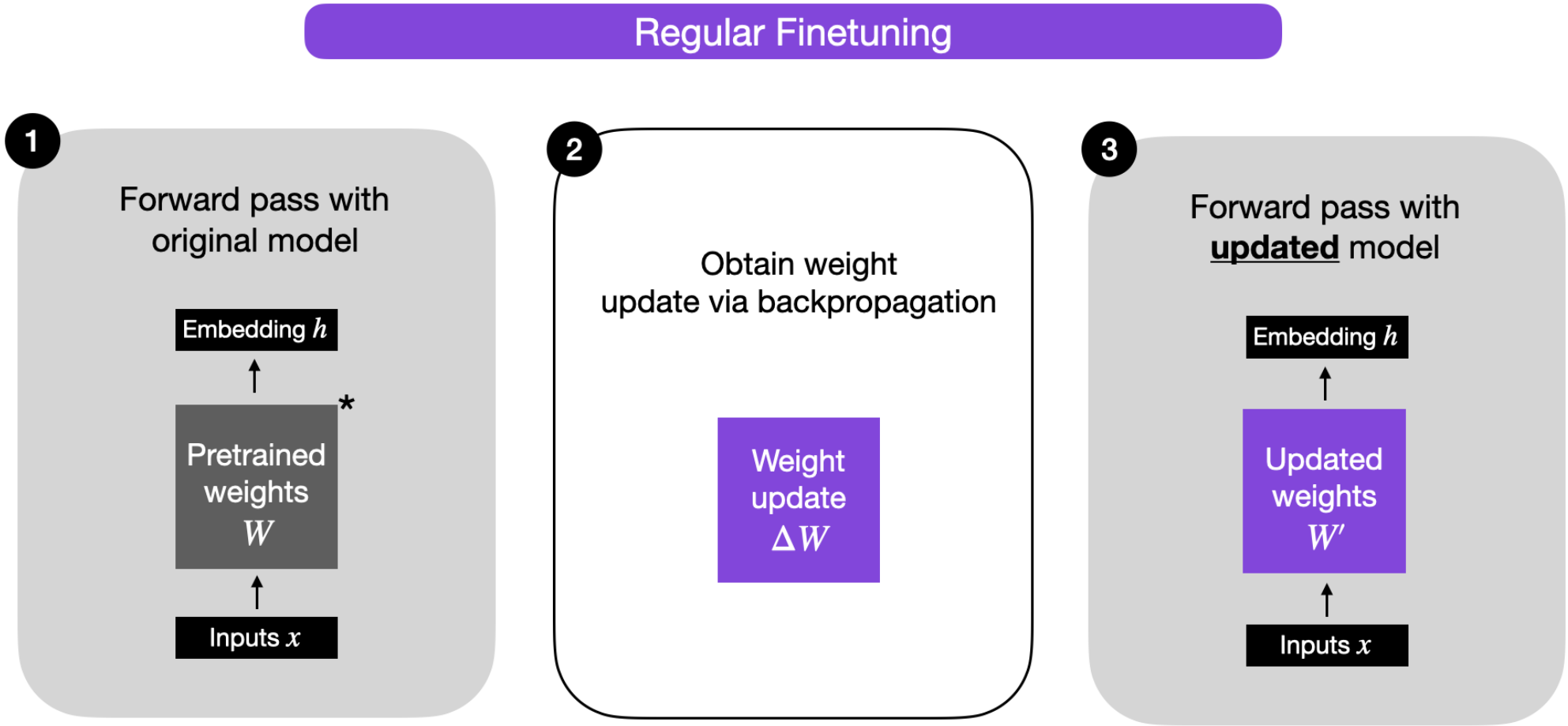
Optimization Step 2: LoRA

Regular Finetuning



* The pretrained model could be any LLM, e.g., an encoder-style LLM (like BERT) or a generative decoder-style LLM (like GPT)

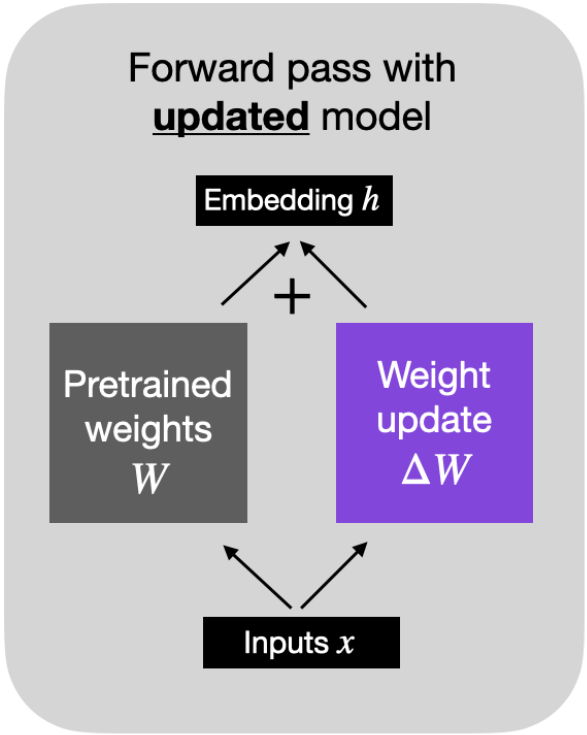
Optimization Step 2: LoRA



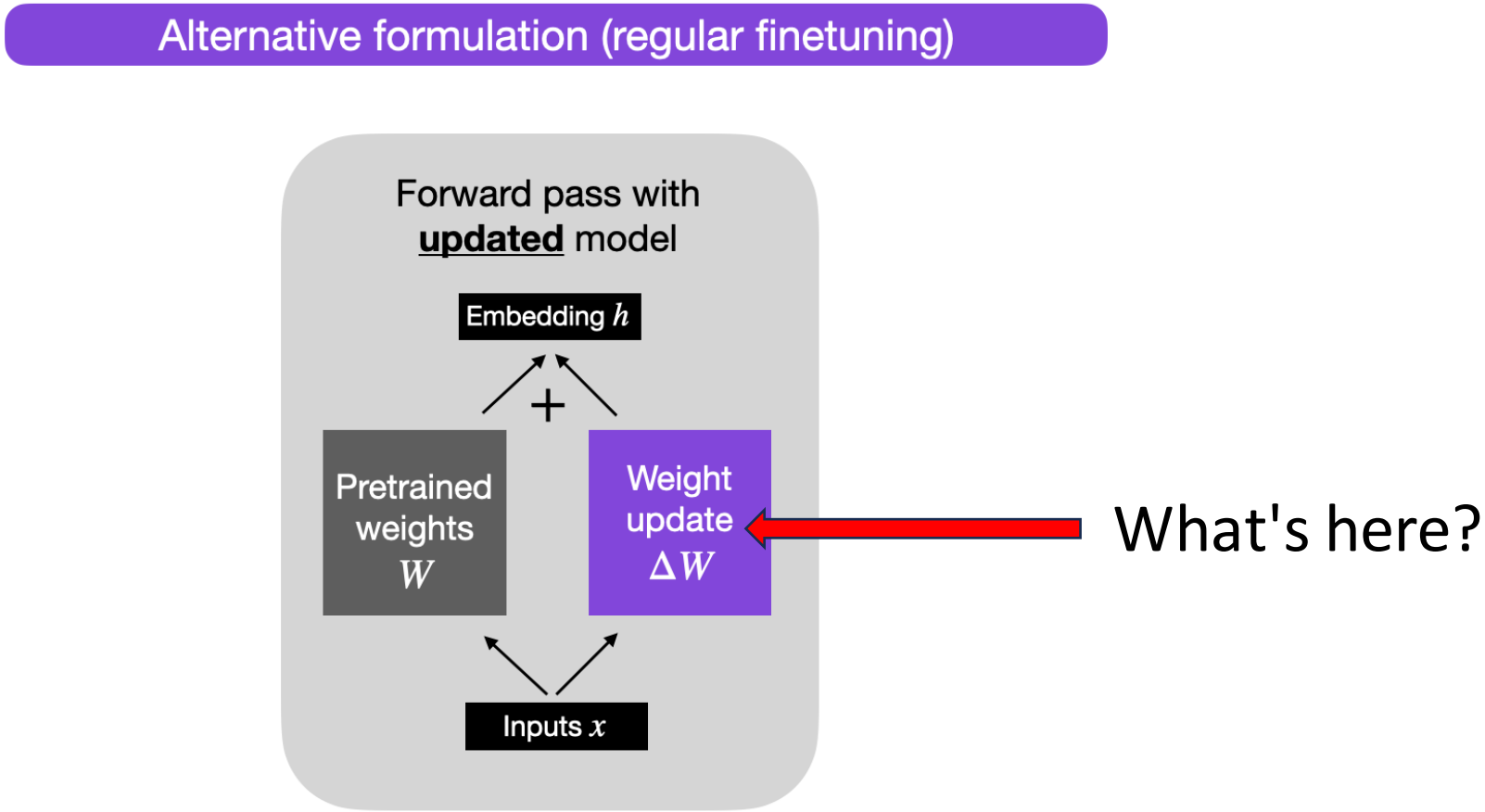
* The pretrained model could be any LLM, e.g., an encoder-style LLM (like BERT) or a generative decoder-style LLM (like GPT)

Optimization Step 2: LoRA

Alternative formulation (regular finetuning)



Optimization Step 2: LoRA



Optimization Step 2: LoRA

INTRINSIC DIMENSIONALITY EXPLAINS THE EFFECTIVENESS OF LANGUAGE MODEL FINE-TUNING

Armen Aghajanyan, Luke Zettlemoyer, Sonal Gupta
Facebook
{armenag,lsz,sonalgupta}@fb.com

empirical and theoretical intuitions to explain this remarkable phenomenon. We empirically show that common pre-trained models have a very low intrinsic dimension; in other words, there exists a low dimension reparameterization that is as effective for fine-tuning as the full parameter space. For example, by optimiz-

Optimization Step 2: LoRA

INTRINSIC DIMENSIONALITY EXPLAINS THE EFFECTIVENESS OF LANGUAGE MODEL FINE-TUNING

Armen Aghajanyan, Luke Zettlemoyer, Sonal Gupta
Facebook
{armenag,lsz,sonalgupta}@fb.com

empirical and theoretical intuitions to explain this remarkable phenomenon. We empirically show that common pre-trained models have a very low intrinsic dimension; in other words, there exists a low dimension reparameterization that is as effective for fine-tuning as the full parameter space. For example, by optimiz-

A low intrinsic dimension means the data can be effectively represented or approximated by a lower-dimensional space while retaining most of its essential information or structure. In other words, this means we can decompose the new weight matrix for the adapted task into lower-dimensional (smaller) matrices without losing too much important information.

Optimization Step 2: LoRA

INTRINSIC DIMENSIONALITY EXPLAINS THE EFFECTIVENESS OF LANGUAGE MODEL FINE-TUNING

Armen Aghajanyan, Luke Zettlemoyer, Sonal Gupta
Facebook
{armenag,lsz,sonalgupta}@fb.com

empirical and theoretical intuitions to explain this remarkable phenomenon. We empirically show that common pre-trained models have a very low intrinsic dimension; in other words, there exists a low dimension reparameterization that is as effective for fine-tuning as the full parameter space. For example, by optimiz-

$$\underset{3\times 3}{\begin{bmatrix} 2 & 20 & 1 \\ 4 & 40 & 2 \\ 6 & 60 & 3 \end{bmatrix}} = \underset{3\times 1}{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}} \times \underset{1\times 3}{\begin{bmatrix} 2 & 20 & 30 \end{bmatrix}}$$

Optimization Step 2: LoRA

INTRINSIC DIMENSIONALITY EXPLAINS THE EFFECTIVENESS OF LANGUAGE MODEL FINE-TUNING

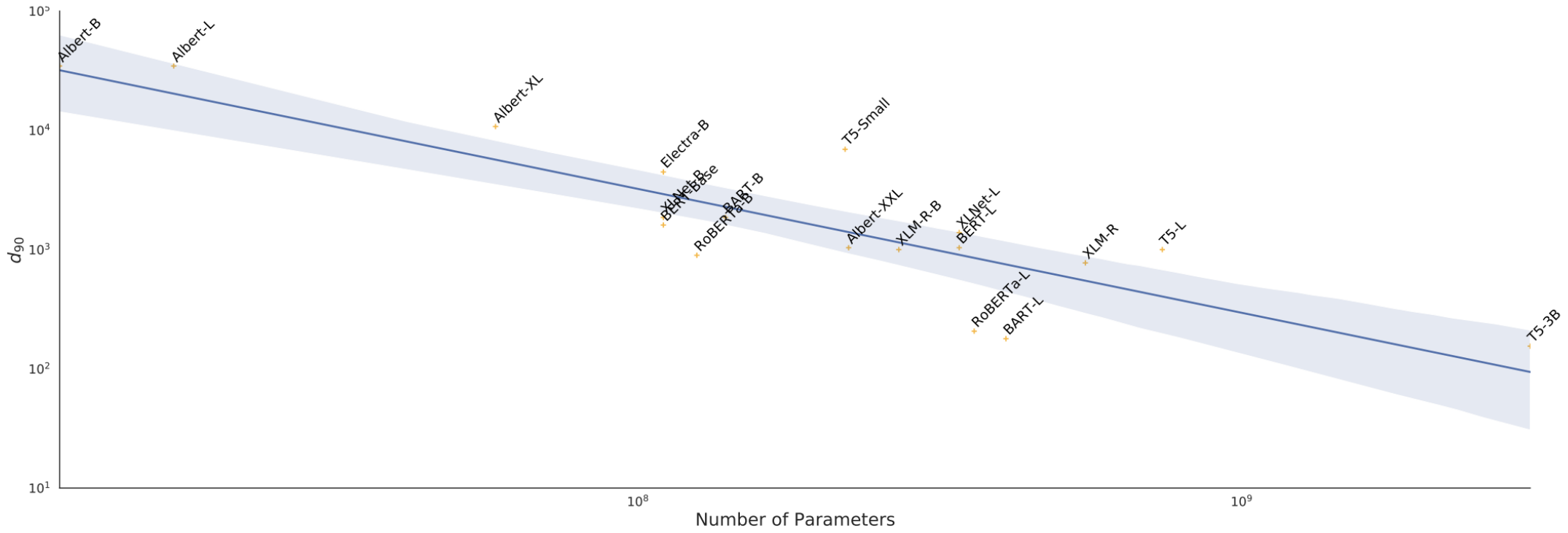
Armen Aghajanyan, Luke Zettlemoyer, Sonal Gupta
Facebook
{armenag,lsz,sonalgupta}@fb.com

empirical and theoretical intuitions to explain this remarkable phenomenon. We empirically show that common pre-trained models have a very low intrinsic dimension; in other words, there exists a low dimension reparameterization that is as effective for fine-tuning as the full parameter space. For example, by optimiz-

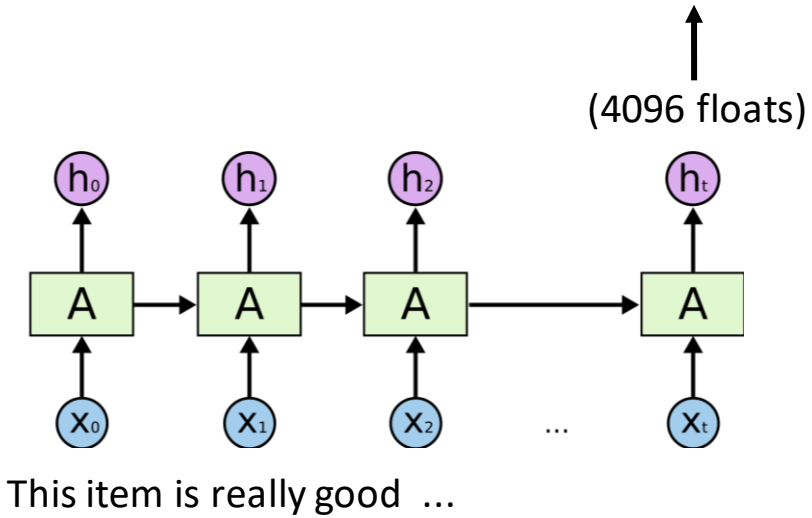
$$\begin{bmatrix} 2 & 20 & 1 \\ 4 & 40 & 2 \\ 6 & 60 & 3 \end{bmatrix}$$ Rank = 1

$$\begin{bmatrix} 2 & 20 & 1 \\ 4 & 70 & 2 \\ 6 & 60 & 3 \end{bmatrix}$$

Optimization Step 2: LoRA

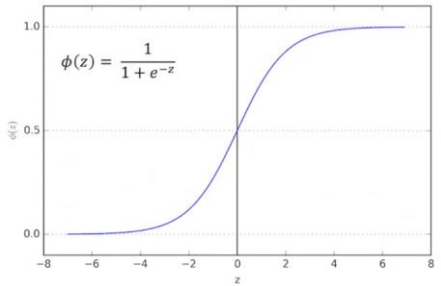


A glimpse into the Past: Sentiment Neuron

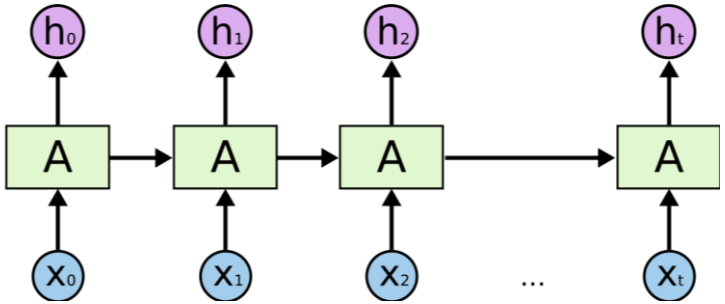


A glimpse into the Past: Sentiment Neuron

Logistic Regression
(**Linear Probing**)



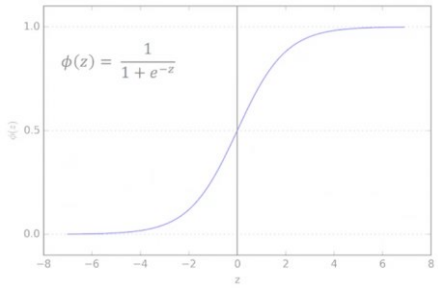
(4096 floats)



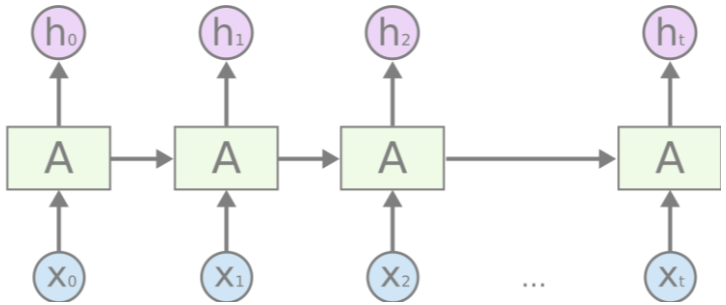
This item is really good ...

A glimpse into the Past: Sentiment Neuron

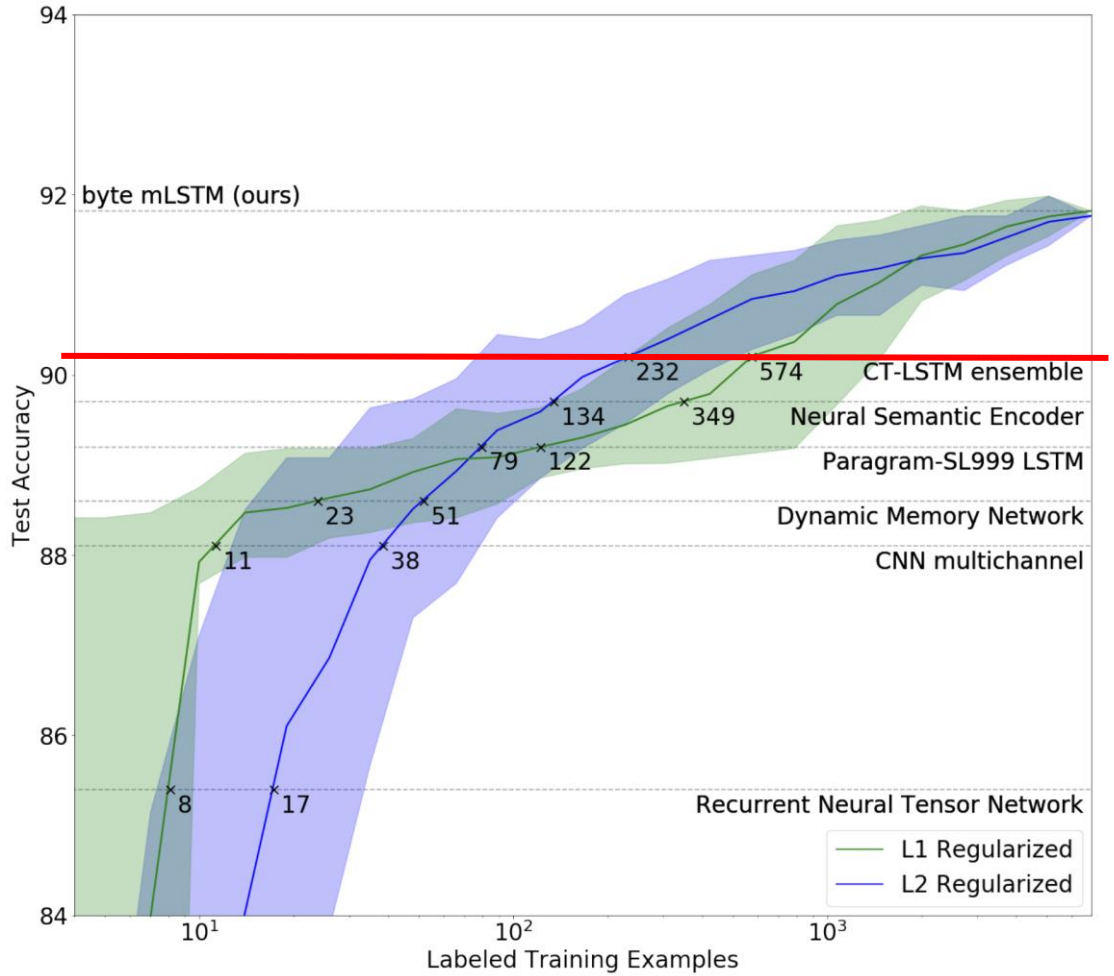
Logistic Regression
(**Linear Probing**)



(4096 floats)



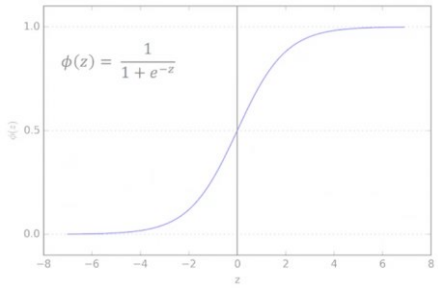
This item is really good ...



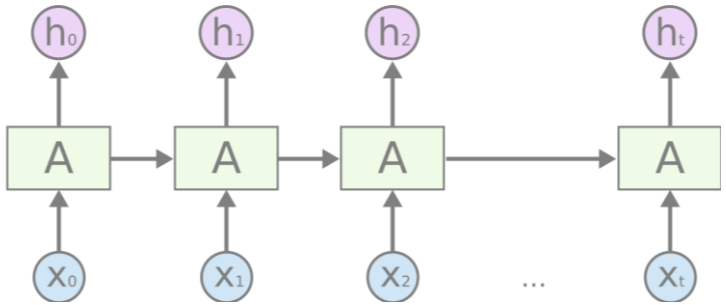
Metrics on Stanford Sentiment Treebank (SST)
Rotten Tomatoes

A glimpse into the Past: Sentiment Neuron

Logistic Regression
(**Linear Probing**)

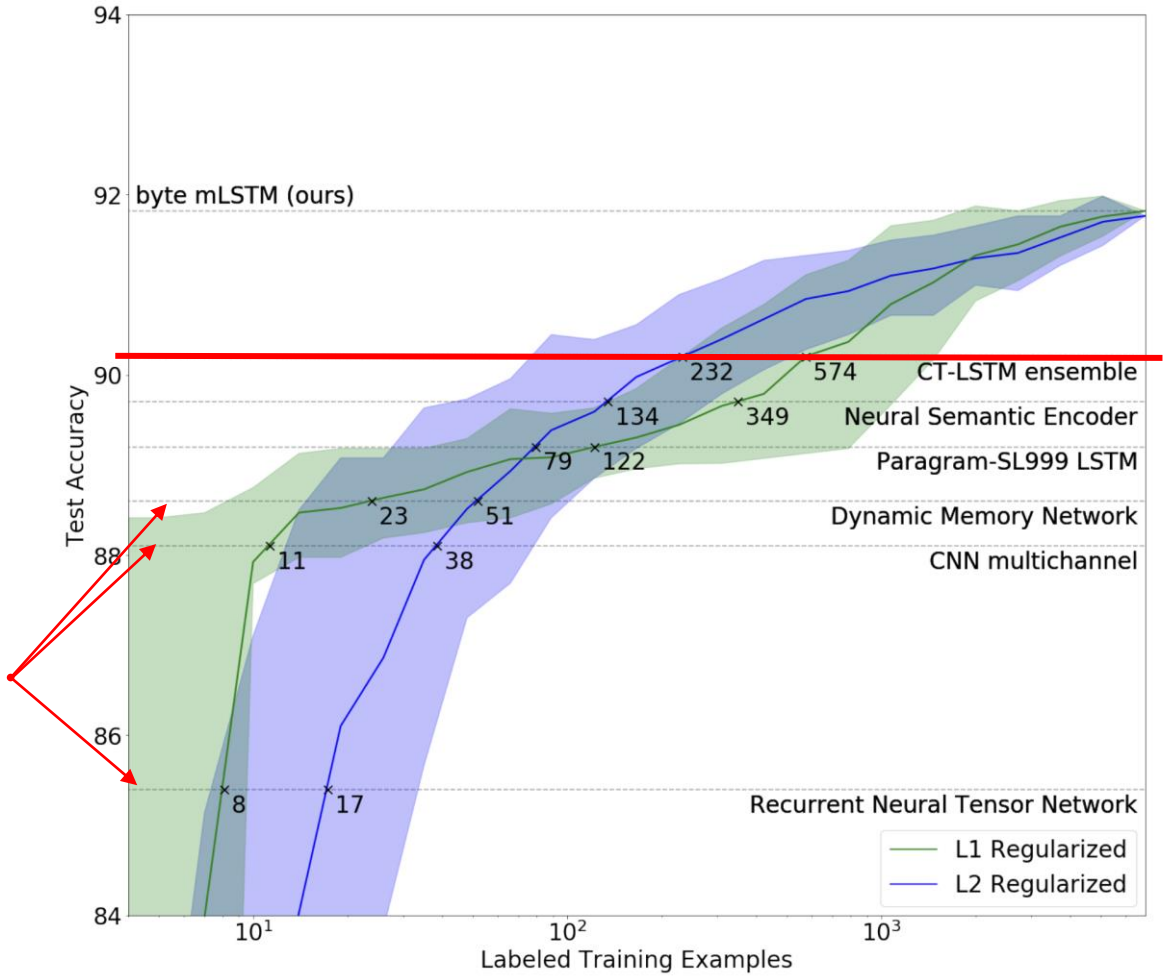


(4096 floats)



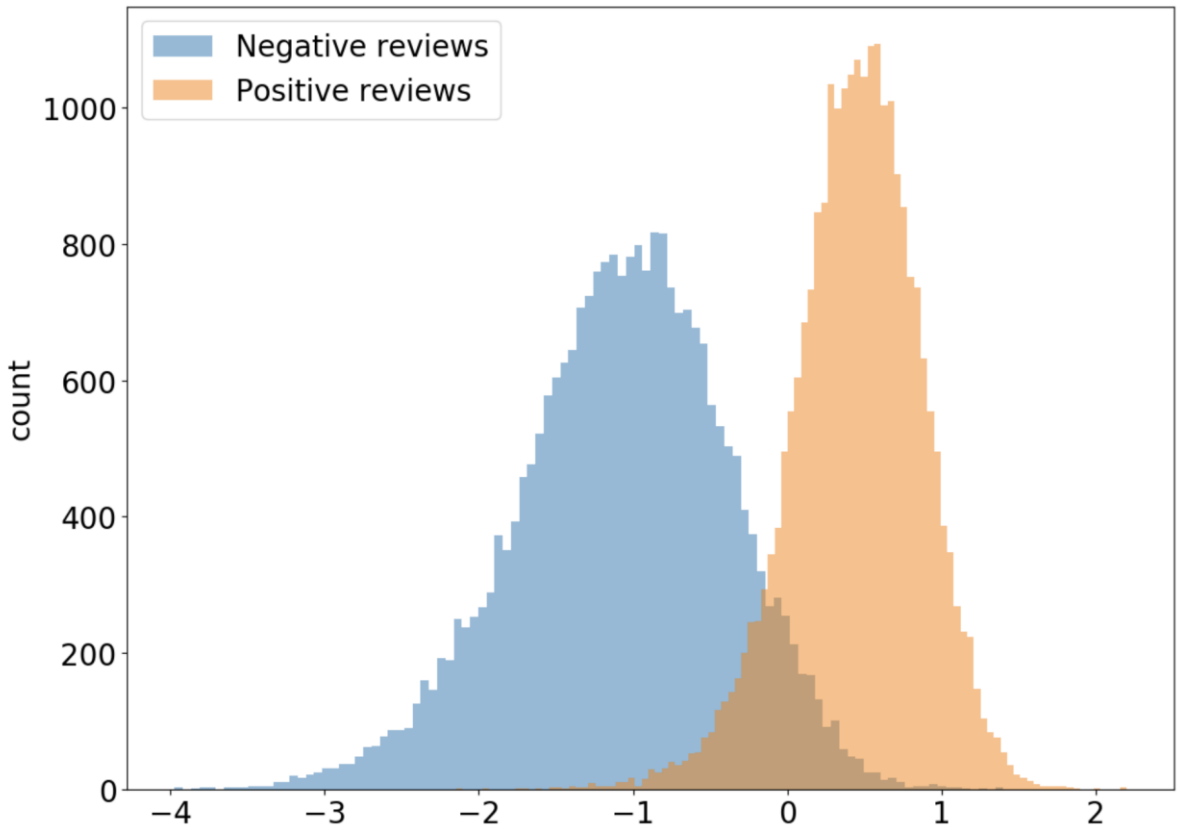
This item is really good ...

Trained on 6k+ examples!



Metrics on Stanford Sentiment Treebank (SST) Rotten Tomatoes

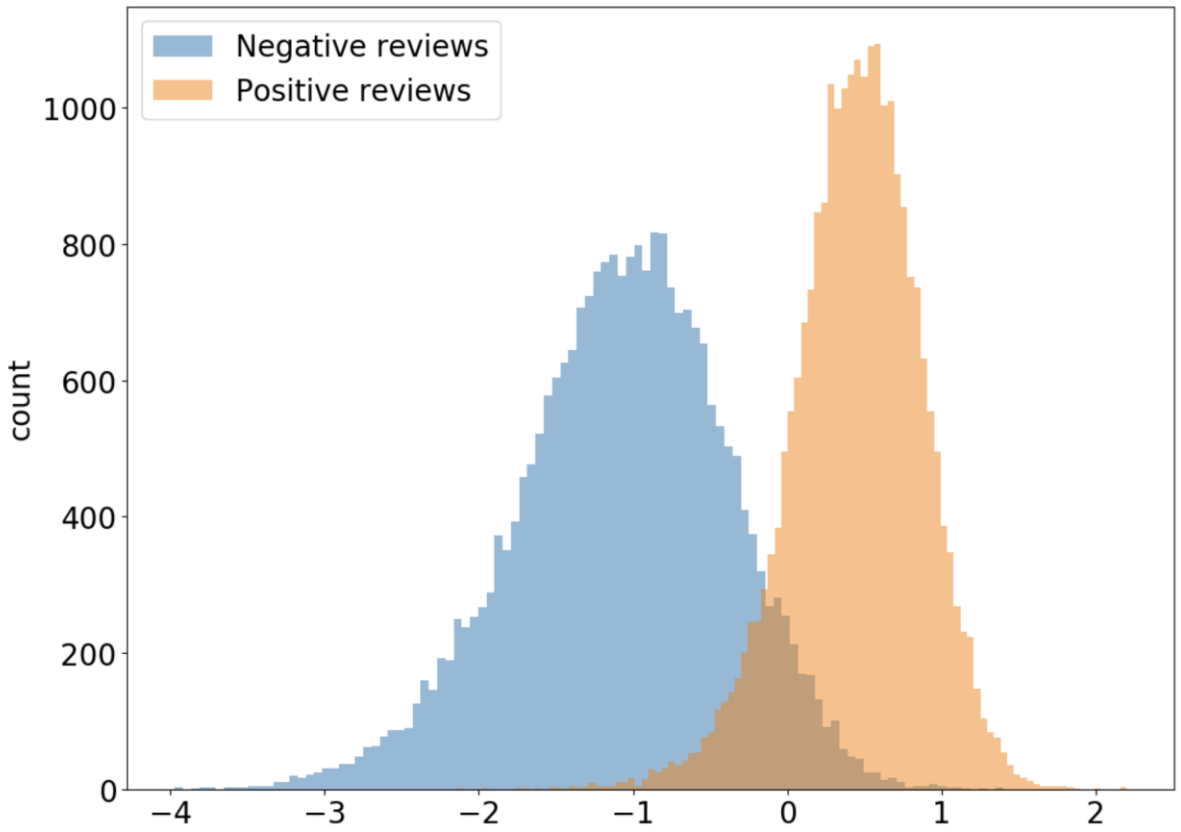
A glimpse into the Past: Sentiment Neuron



One neuron activations, IMDB dataset

A glimpse into the Past: Sentiment Neuron

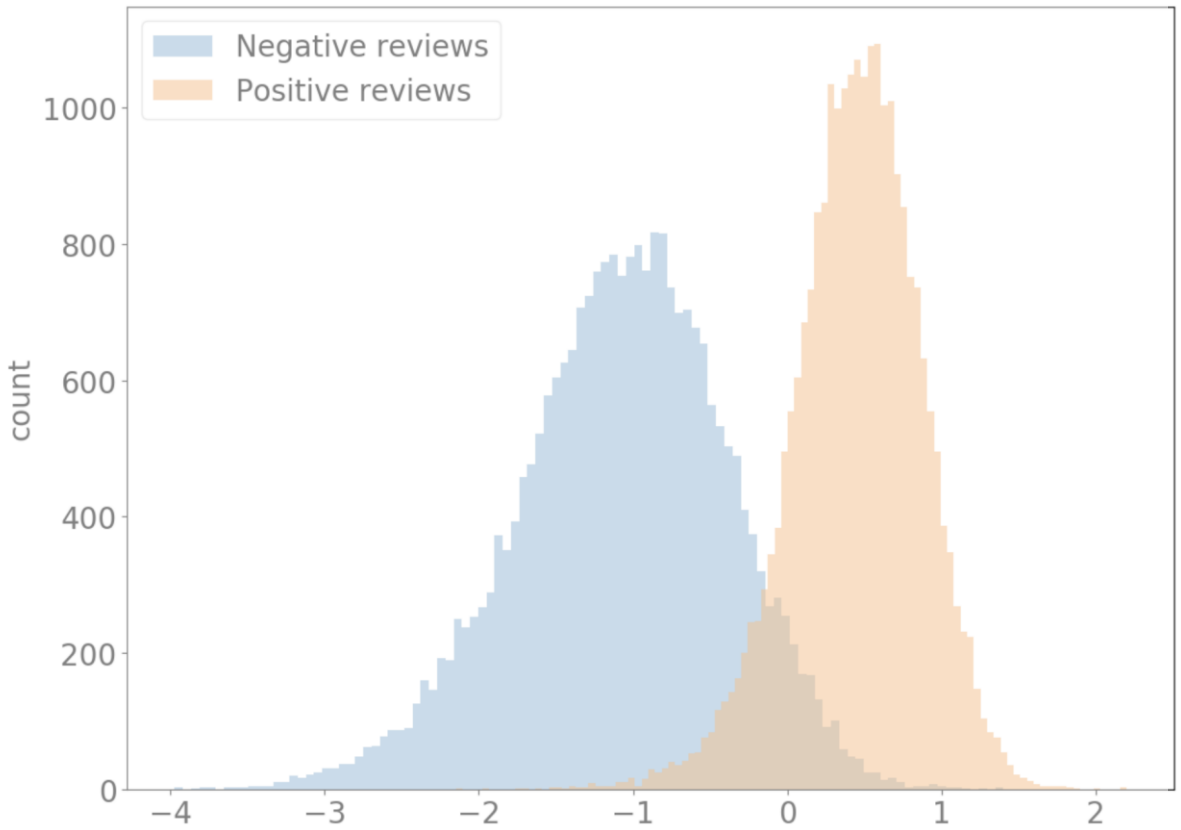
This bahavior is only observed for relatively big models (4k hidden units)



One neuron activations, IMDB dataset

A glimpse into the Past: Sentiment Neuron

This bahavior is only observed for relatively big models (4k hidden units)

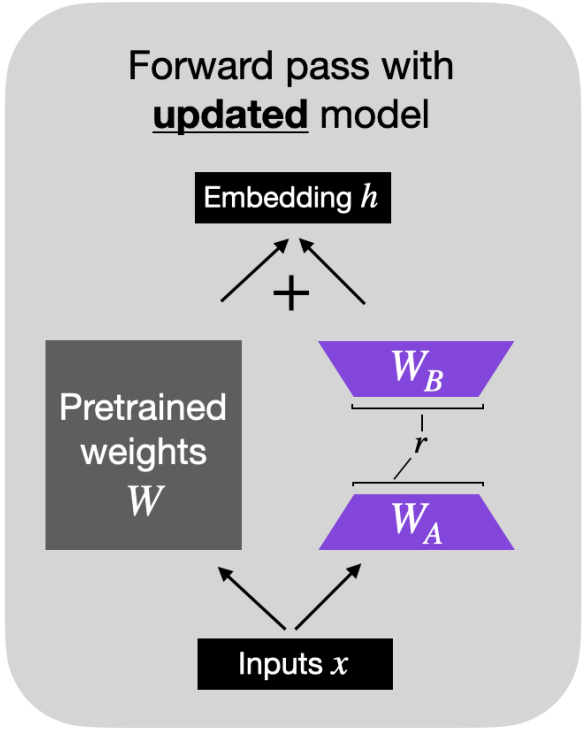


One neuron activations, IMDB dataset

This is one of Crichton's best books. The characters of Karen Ross, Peter Elliot, Munro, and Amy are beautifully developed and their interactions are exciting, complex, and fast-paced throughout this impressive novel. And about 99.8 percent of that got lost in the film. Seriously, the screenplay AND the directing were horrendous and clearly done by people who could not fathom what was good about the novel. I can't fault the actors because frankly, they never had a chance to make this turkey live up to Crichton's original work. I know good novels, especially those with a science fiction edge, are hard to bring to the screen in a way that lives up to the original. But this may be the absolute worst disparity in quality between novel and screen adaptation ever. The book is really, really good. The movie is just dreadful.

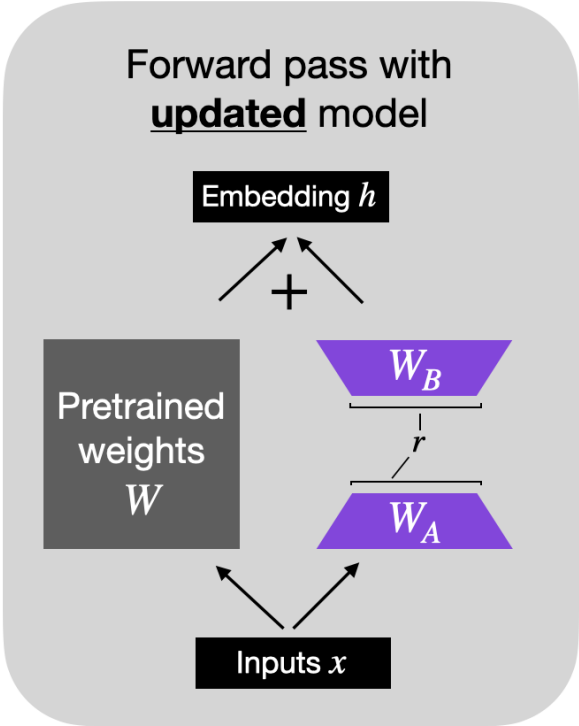
Optimization Step 2: Finally, LoRA

LoRA weights, W_A and W_B, represent ΔW



Optimization Step 2: Finally, LoRA

LoRA weights, W_A and W_B, represent ΔW

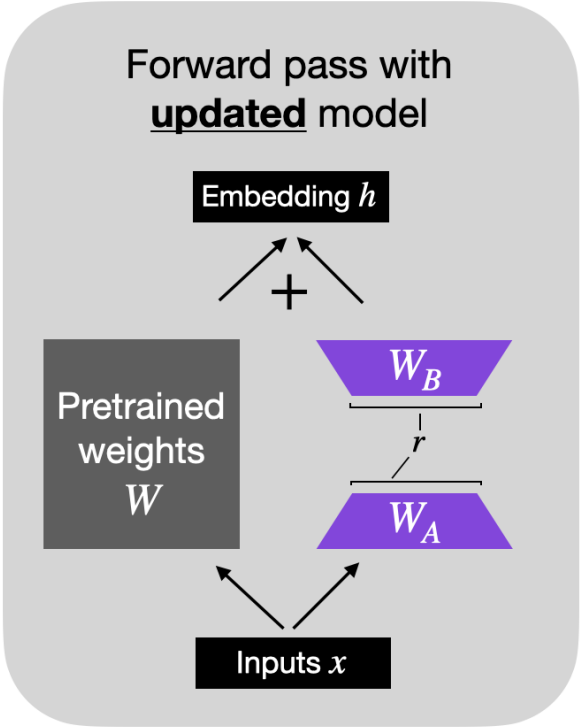



```python
class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        std_dev = 1 / torch.sqrt(torch.tensor(rank).float())
        self.A = torch.nn.Parameter(torch.randn(in_dim, rank) * std_dev)
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

Optimization Step 2: Finally, LoRA

LoRA weights, W_A and W_B, represent ΔW



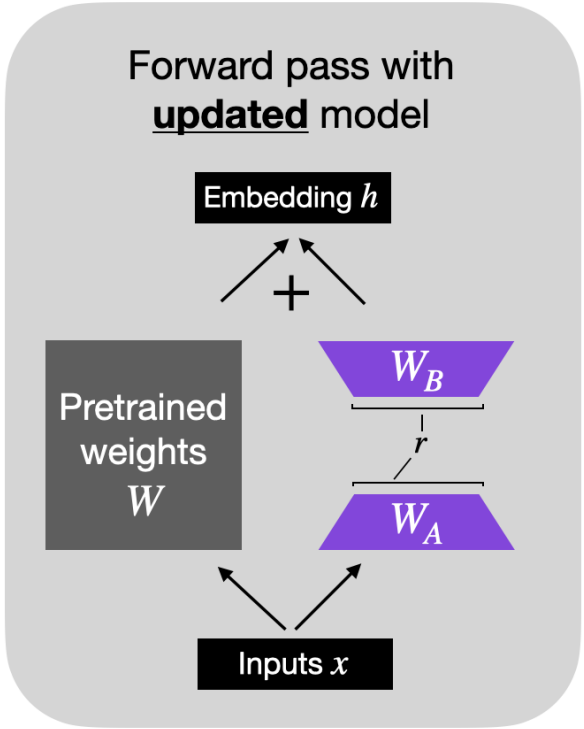
A higher "alpha" would place more emphasis on the low-rank structure or regularization, while a lower "alpha" would reduce its influence, making the model rely more on the original parameters. Adjusting "alpha" helps in striking a balance between fitting the data and preventing overfitting by regularizing the model.

```
class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        std_dev = 1 / torch.sqrt(torch.tensor(rank).float())
        self.A = torch.nn.Parameter(torch.randn(in_dim, rank) * std_dev)
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

Optimization Step 2: Finally, LoRA

LoRA weights, W_A and W_B, represent ΔW

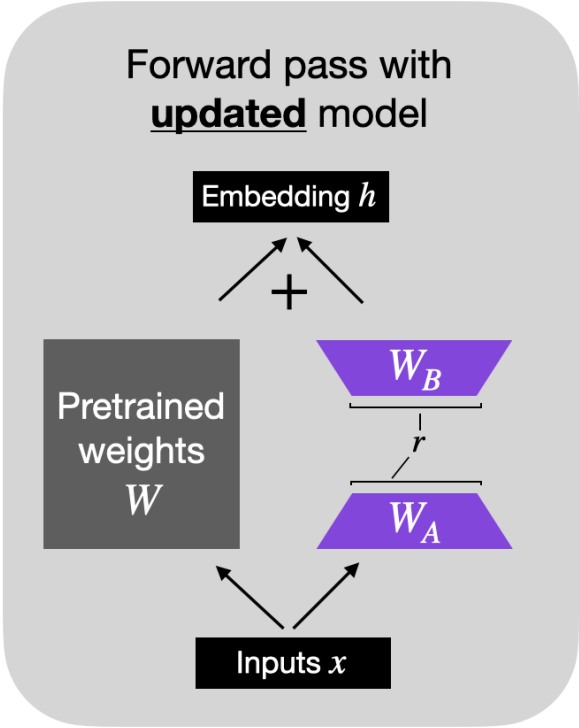


Some notes:

- The original study proposed to apply LoRA only on Attention Weights
- You can play with the number of layers affected
- Works well even with small ranks (~8)

Optimization Step 2: Finally, LoRA

LoRA weights, W_A and W_B, represent ΔW



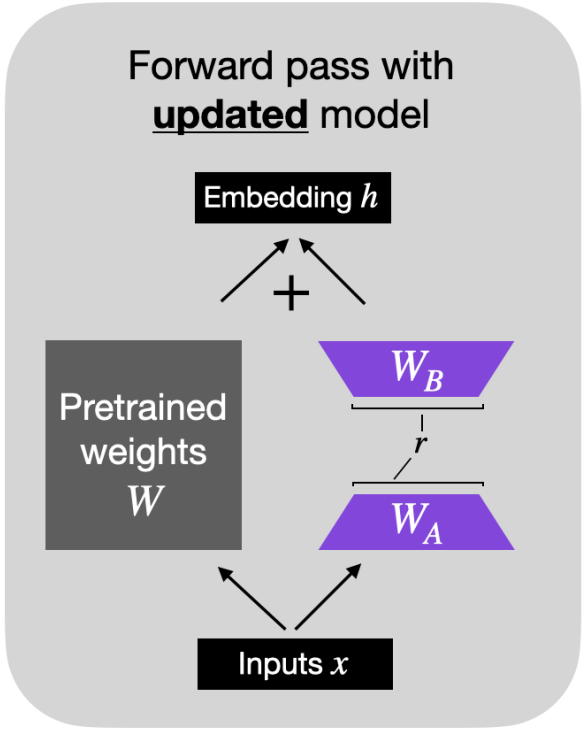
Some notes:

- The original study proposed to apply LoRA only on Attention Weights
- You can play with the number of layers affected
- Works well even with small ranks (~8)

Rank	7B	13B	70B	180B
1	0.00%	0.00%	0.00%	0.00%
2	0.01%	0.00%	0.00%	0.00%
8	0.02%	0.01%	0.01%	0.00%
16	0.04%	0.03%	0.01%	0.01%
512	1.22%	0.90%	0.39%	0.24%
1,024	2.45%	1.80%	0.77%	0.48%
8,192	19.58%	14.37%	6.19%	3.86%

Optimization Step 2: Finally, LoRA

LoRA weights, W_A and W_B, represent ΔW



Some notes:

- The original study proposed to apply LoRA only on Attention Weights
- You can play with the number of layers affected
- Works well even with small ranks (~8)
- The more «complex» your task, the bigger the rank should be
  - e.g., teaching behaviour that contradicts or falls outside the scope of pretraining
- Later studies (QLoRA) found that it's better to tune all linear layers, even with smaller ranks
  - For some tasks there's no difference between ranks of 8 and 256!

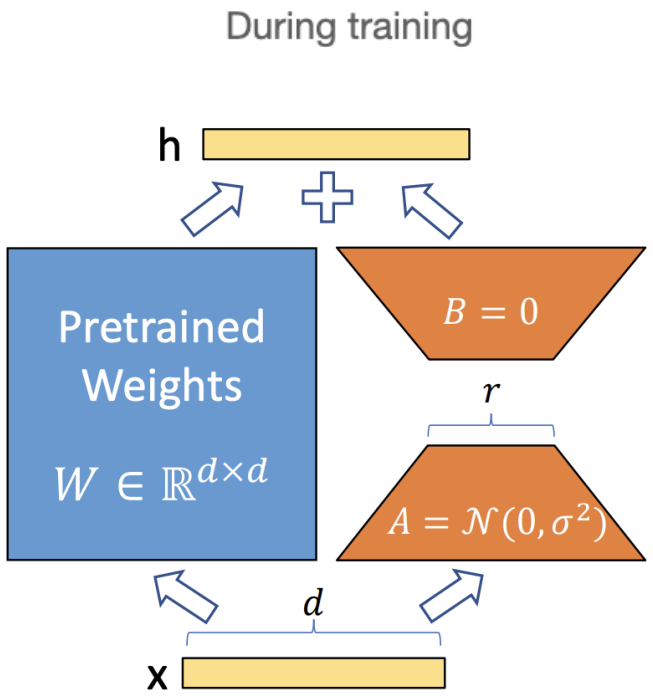
Optimization Step 2: Finally, LoRA

Model&Method	# Trainable Parameters	WikiSQL Acc. (%)	MNLI-m Acc. (%)	SAMSum R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 ($\text{Adapter}^{\text{H}}$)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 ($\text{Adapter}^{\text{H}}$)	40.1M	73.2	**91.5**	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	**91.7**	**53.8/29.8/45.9**
GPT-3 (LoRA)	37.7M	**74.0**	**91.6**	53.4/29.2/45.1

Some notes:

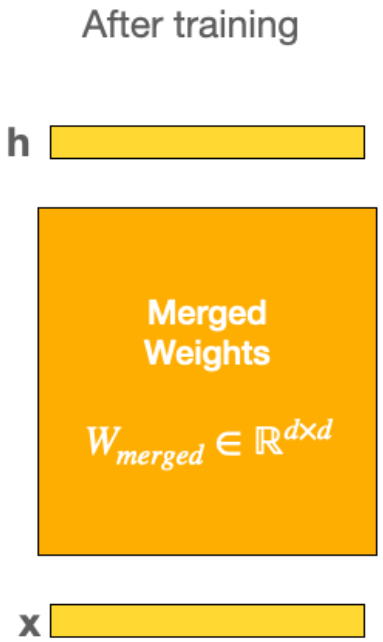
- The original study proposed to apply LoRA only on Attention Weights
- You can play with the number of layers affected
- Works well even with small ranks (~8)
- The more «complex» your task, the bigger the rank should be
  - e.g., teaching behaviour that contradicts or falls outside the scope of pretraining
- Later studies (QLoRA) found that it's better to tune all linear layers, even with smaller ranks
  - For some tasks there's no difference between ranks of 8 and 256!
- Often, LoRA works as a regularization, and thus improves performance compared to FT!

Optimization Step 2: LoRA for Production

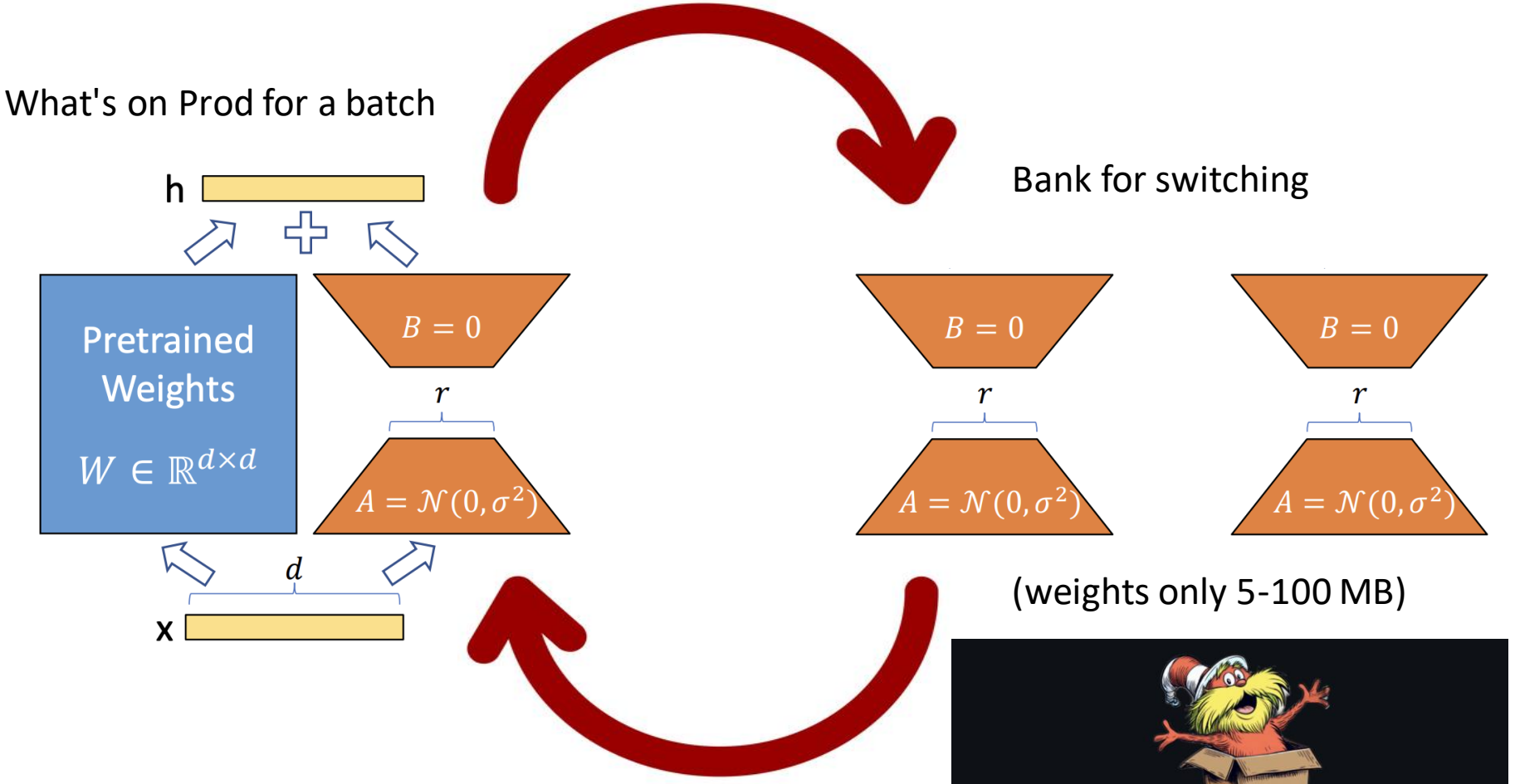


$$h = Wx + BAx$$

$$h = \underbrace{(W + BA)}_{W_{merged}}x$$



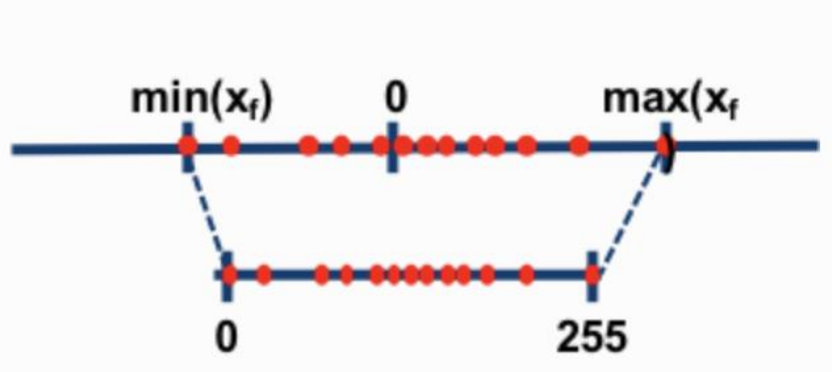
Optimization Step 2: LoRA for Production



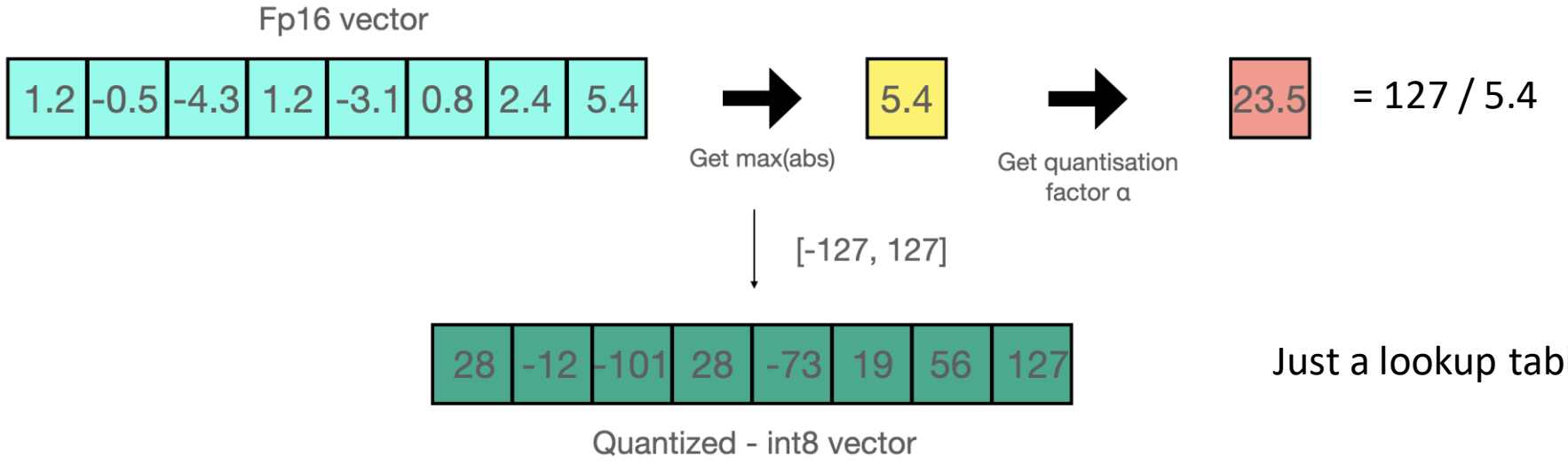
Optimization Step 3: Quantization

Fp16 vector

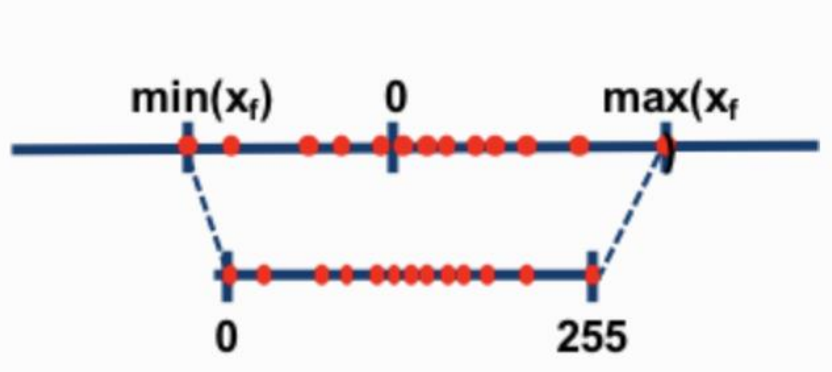
1.2	-0.5	-4.3	1.2	-3.1	0.8	2.4	5.4



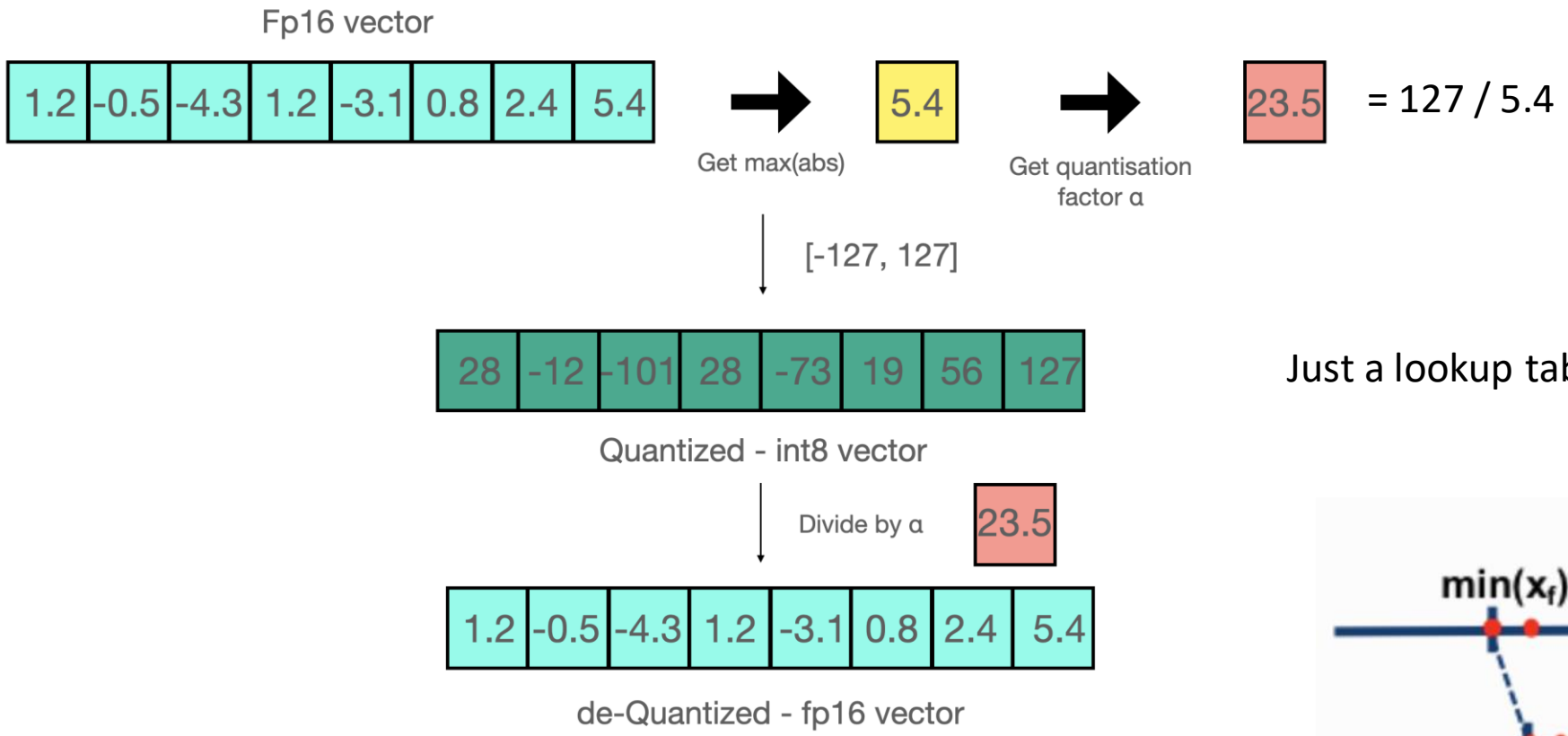
Optimization Step 3: Quantization



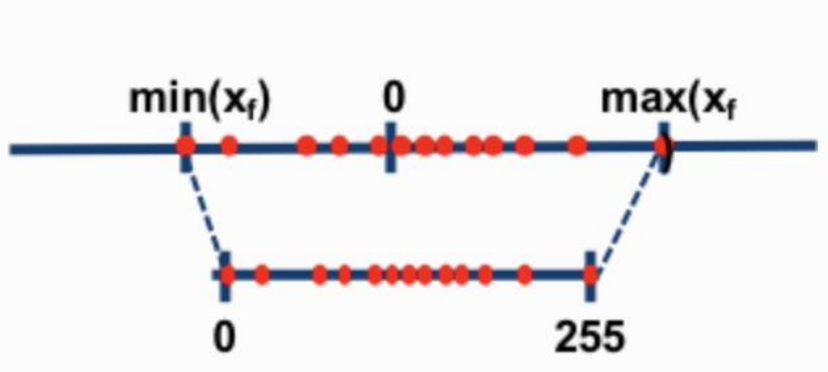
Just a lookup table



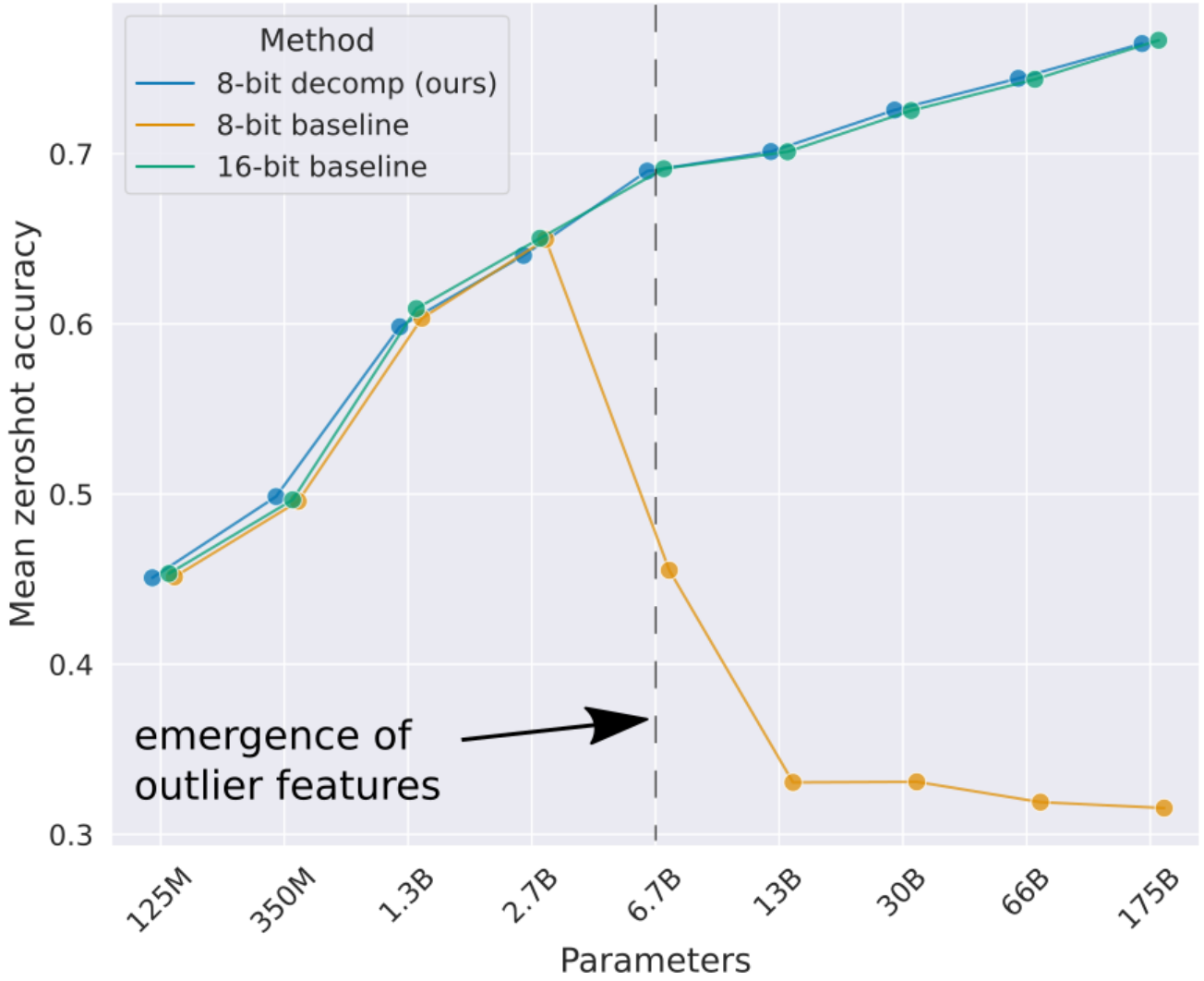
Optimization Step 3: Quantization



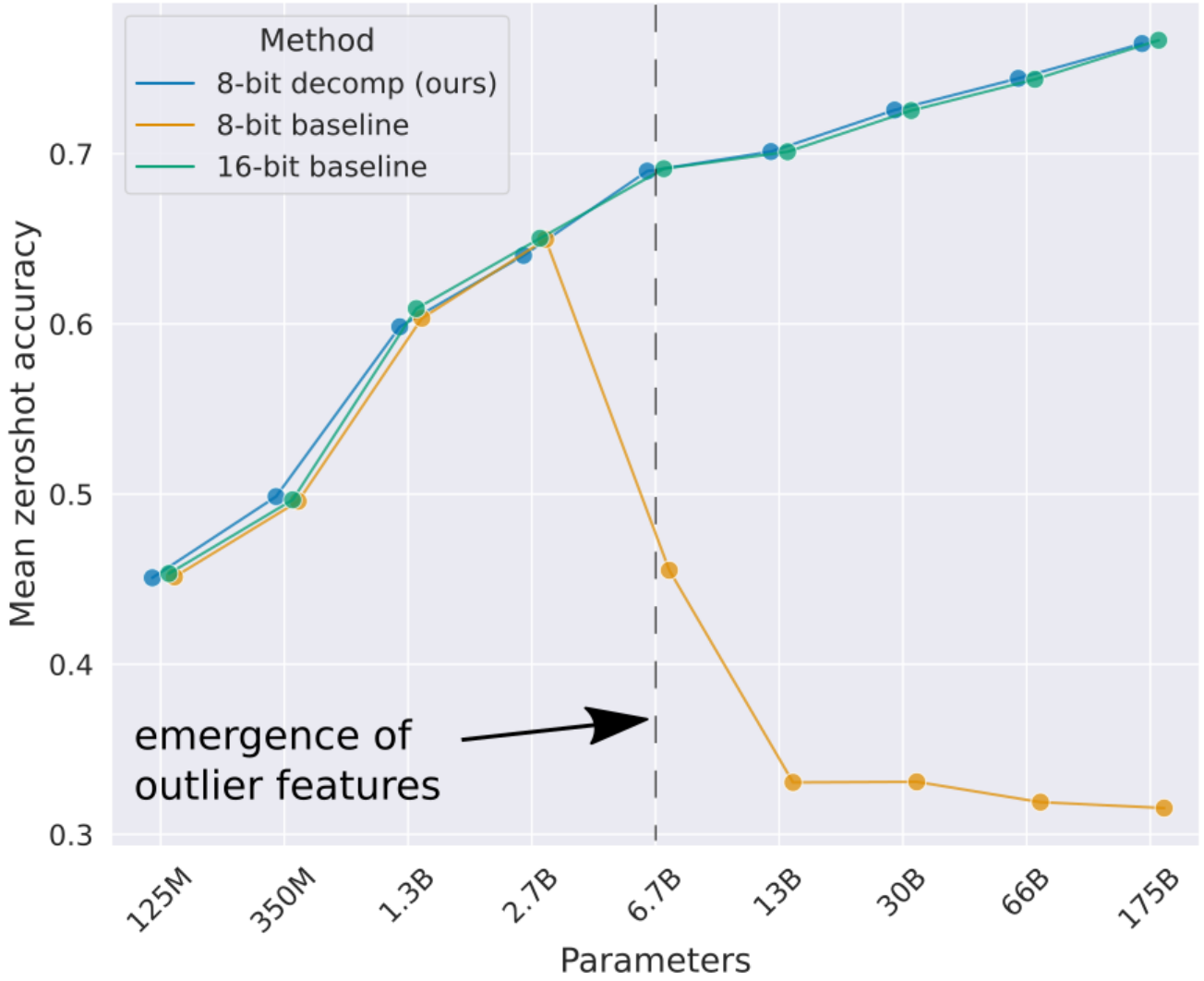
Just a lookup table



Optimization Step 3: Quantization



Optimization Step 3: Quantization



Original activations:

[-0.10, -0.23, 0.08, -0.38, -0.28, -0.29, -2.11, 0.34, -0.53, -67.0]

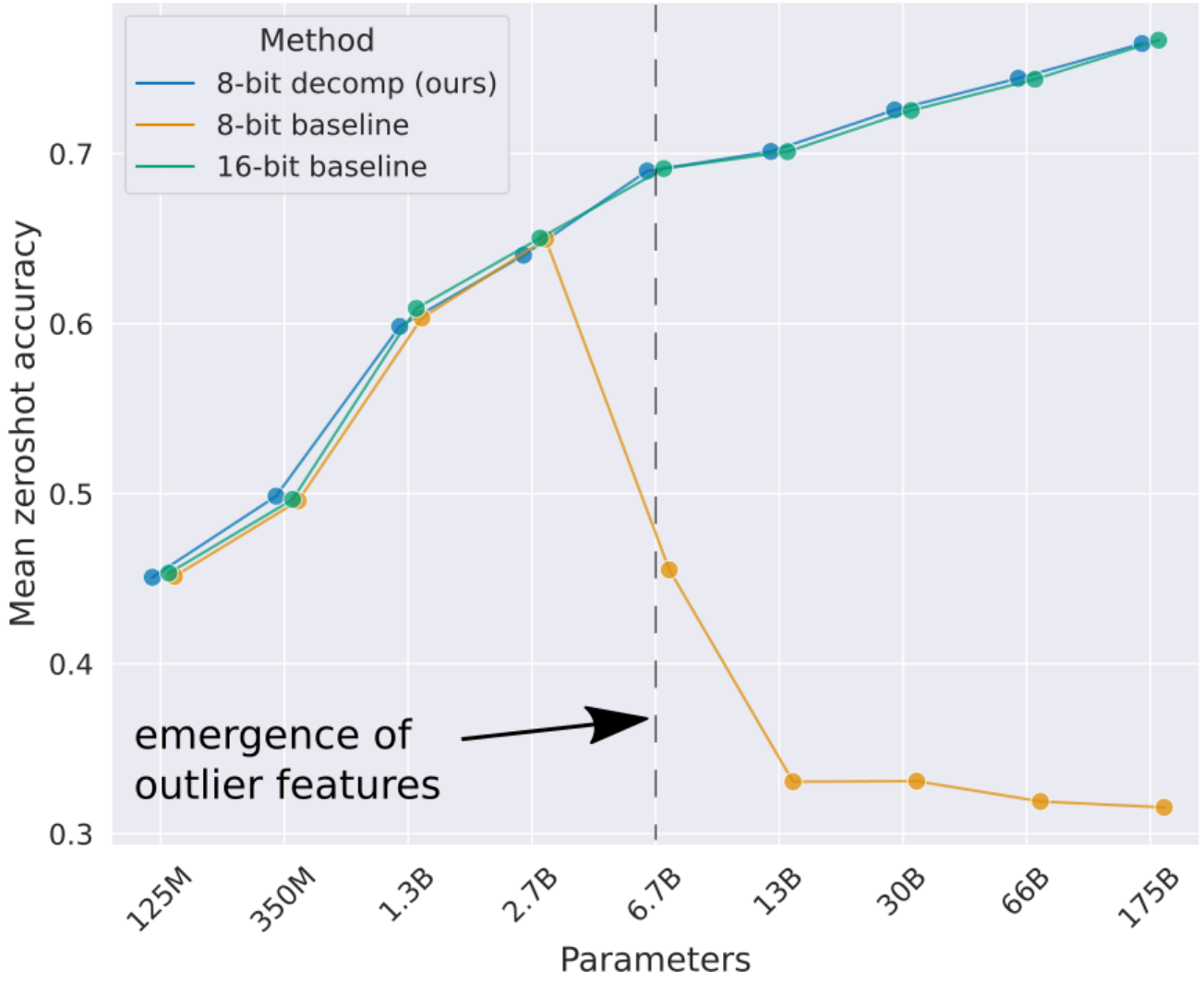
Quant + Dequant w/o an outlier

[-0.10, -0.23, 0.08, -0.38, -0.28, -0.28, -2.11, 0.33, -0.53]

Quant + Dequant w/ an outlier

[-0.00, -0.00, 0.00, -0.53, -0.53, -0.53, -2.11, 0.53, -0.53, -67.00]

Optimization Step 3: Quantization

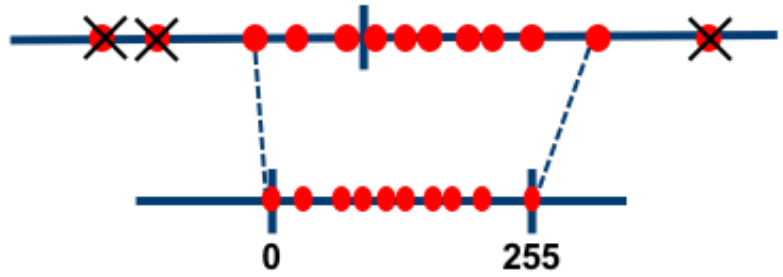


[0, 1, -60, 4]

[3, 0, -50, -2]

[-1, 0, -55, 1]

[3, 2, -60, 1]



Optimization Step 3: Quantization

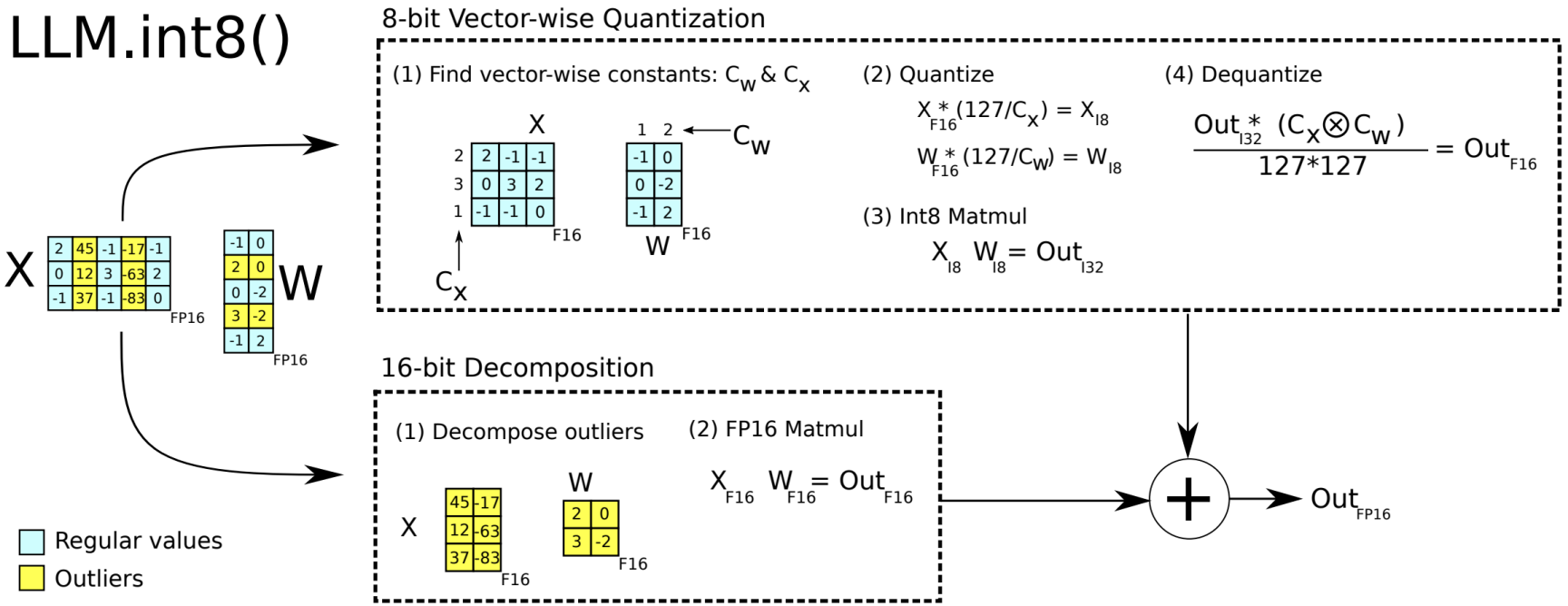
While up to 150k outliers exist per 2048 token sequence for a 13B model, these outlier features are highly systematic and only representing at most 7 unique feature dimensions h

Optimization Step 3: Quantization

We can separate these emergent features into a separate, high precision matrix multiplication, quantize the other 99.9% of values to Int8, can combine the output of both matrix multiplications. This avoids the information squishing to zero effect, and we can recover full transformer performance.

While up to 150k outliers exist per 2048 token sequence for a 13B model, these outlier features are highly systematic and only representing at most 7 unique feature dimensions h

Optimization Step 3: Quantization



We can separate these emergent features into a separate, high precision matrix multiplication, quantize the other 99.9% of values to Int8, can combine the output of both matrix multiplications. This avoids the information squishing to zero effect, and we can recover full transformer performance.

While up to 150k outliers exist per 2048 token sequence for a 13B model, these outlier features are highly systematic and only representing at most 7 unique feature dimensions h

Optimization Step 3: Quantization

Table 2: Different hardware setups and which methods can be run in 16-bit vs. 8-bit precision. We can see that our 8-bit method makes many models accessible that were not accessible before, in particular, OPT-175B/BLOOM.

			Largest Model that can be run	
Class	Hardware	GPU Memory	8-bit	16-bit
Enterprise	8x A100	80 GB	**OPT-175B / BLOOM**	**OPT-175B / BLOOM**
Enterprise	8x A100	40 GB	**OPT-175B / BLOOM**	OPT-66B
Academic server	8x RTX 3090	24 GB	**OPT-175B / BLOOM**	OPT-66B
Academic desktop	4x RTX 3090	24 GB	**OPT-66B**	OPT-30B
Paid Cloud	Colab Pro	15 GB	**OPT-13B**	GPT-J-6B
Free Cloud	Colab	12 GB	**T0/T5-11B**	GPT-2 1.3B

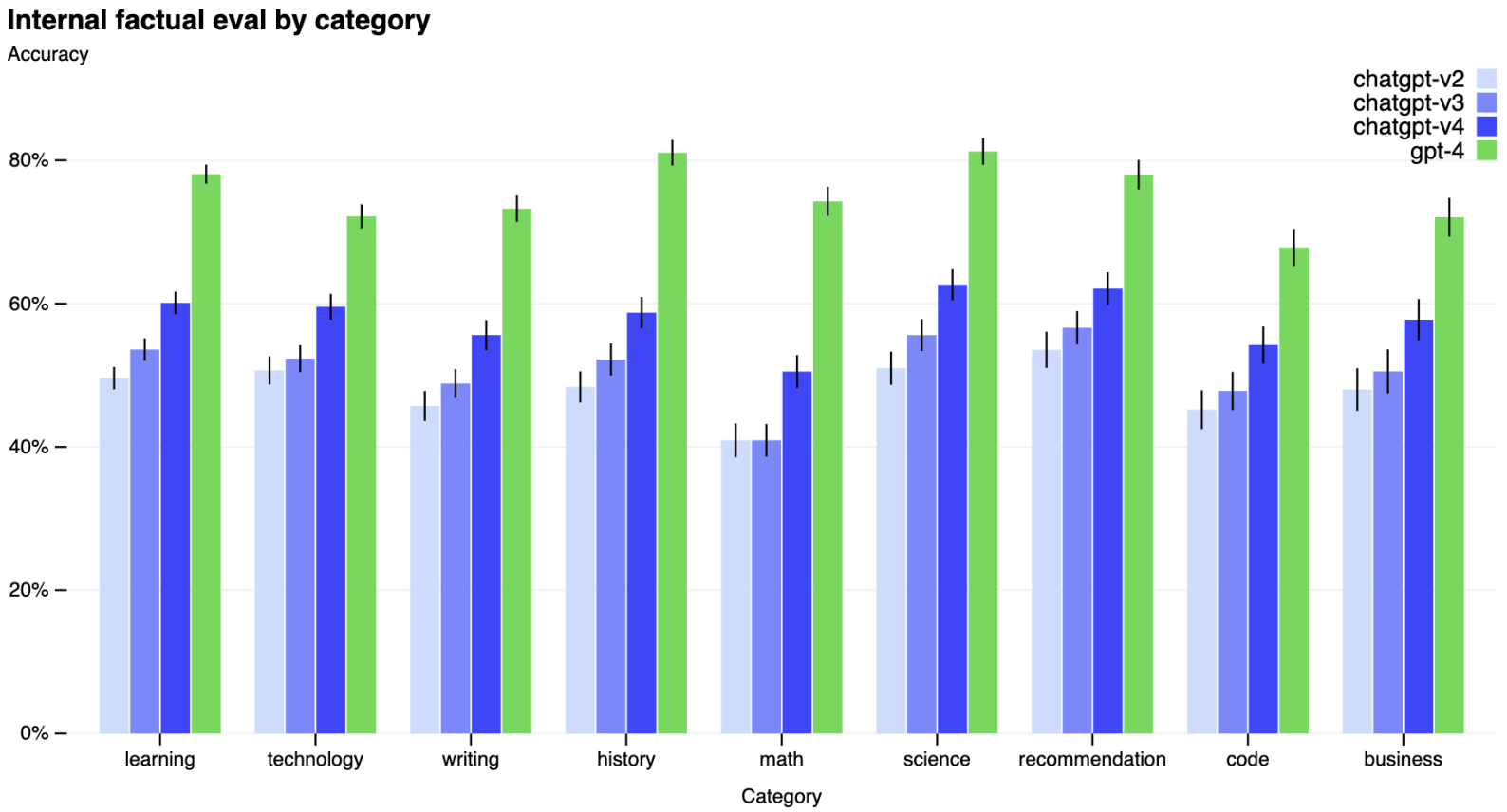
Optimization Step 3: Quantization

Table 5: Inference speedups compared to 16-bit matrix multiplication for the first hidden layer in the feed-forward of differently sized GPT-3 transformers. The hidden dimension is 4x the model dimension. The 8-bit without overhead speedups assumes that no quantization or dequantization is performed. Numbers small than 1.0x represent slowdowns. Int8 matrix multiplication speeds up inference only for models with large model and hidden dimensions.

GPT-3 Size	Small	Medium	Large	XL	2.7B	6.7B	13B	175B
Model dimension	768	1024	1536	2048	2560	4096	5140	12288
FP16-bit baseline	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
Int8 without overhead	0.99x	1.08x	1.43x	1.61x	1.63x	1.67x	2.13x	2.29x
Absmax PyTorch+NVIDIA	0.25x	0.24x	0.36x	0.45x	0.53x	0.70x	0.96x	1.50x
Vector-wise PyTorch+NVIDIA	0.21x	0.22x	0.33x	0.41x	0.50x	0.65x	0.91x	1.50x
Vector-wise	**0.43x**	**0.49x**	**0.74x**	**0.91x**	**0.94x**	**1.18x**	**1.59x**	**2.00x**
LLM.int8() (vector-wise+decomp)	0.14x	0.20x	0.36x	0.51x	0.64x	0.86x	1.22x	1.81x

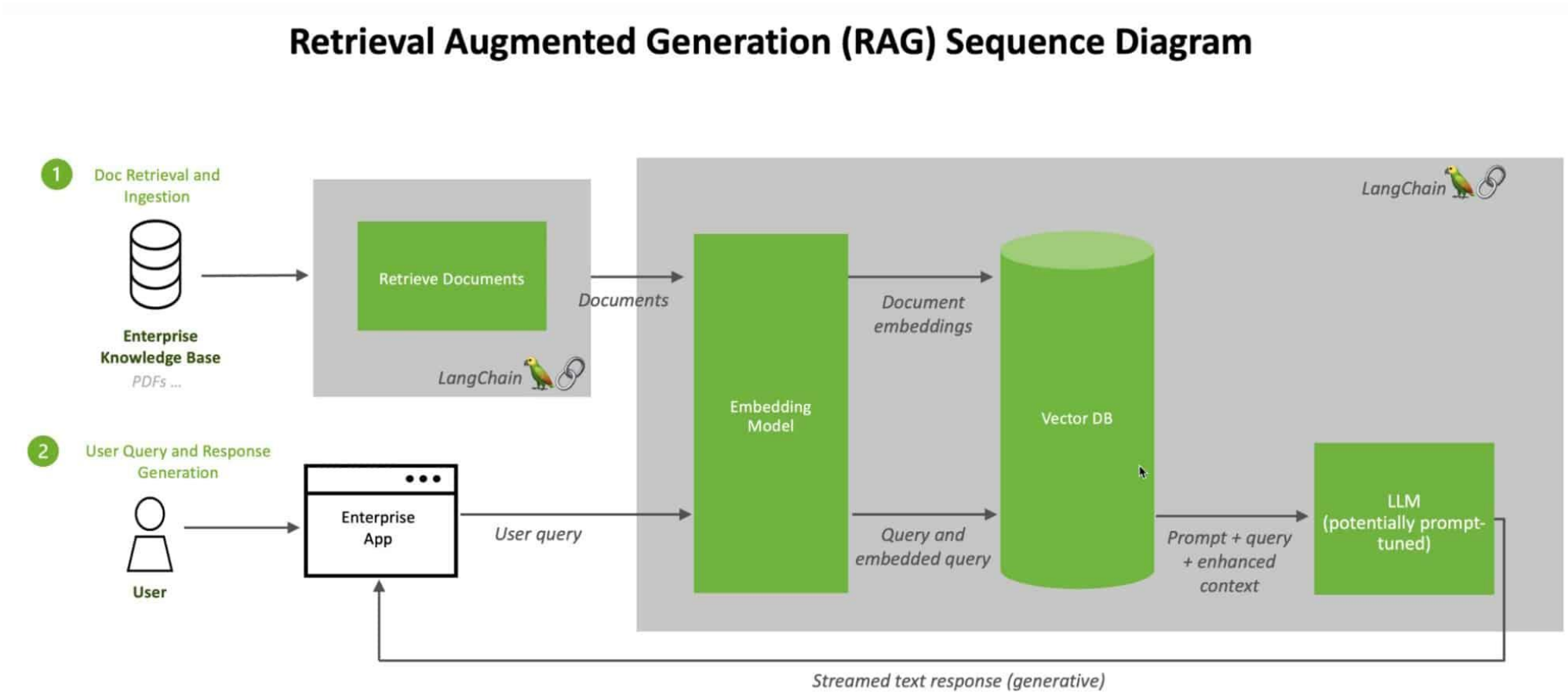
Bonus Part:
Retrieval-Augmented Generation (RAG)

Knowledge-intensive Questions

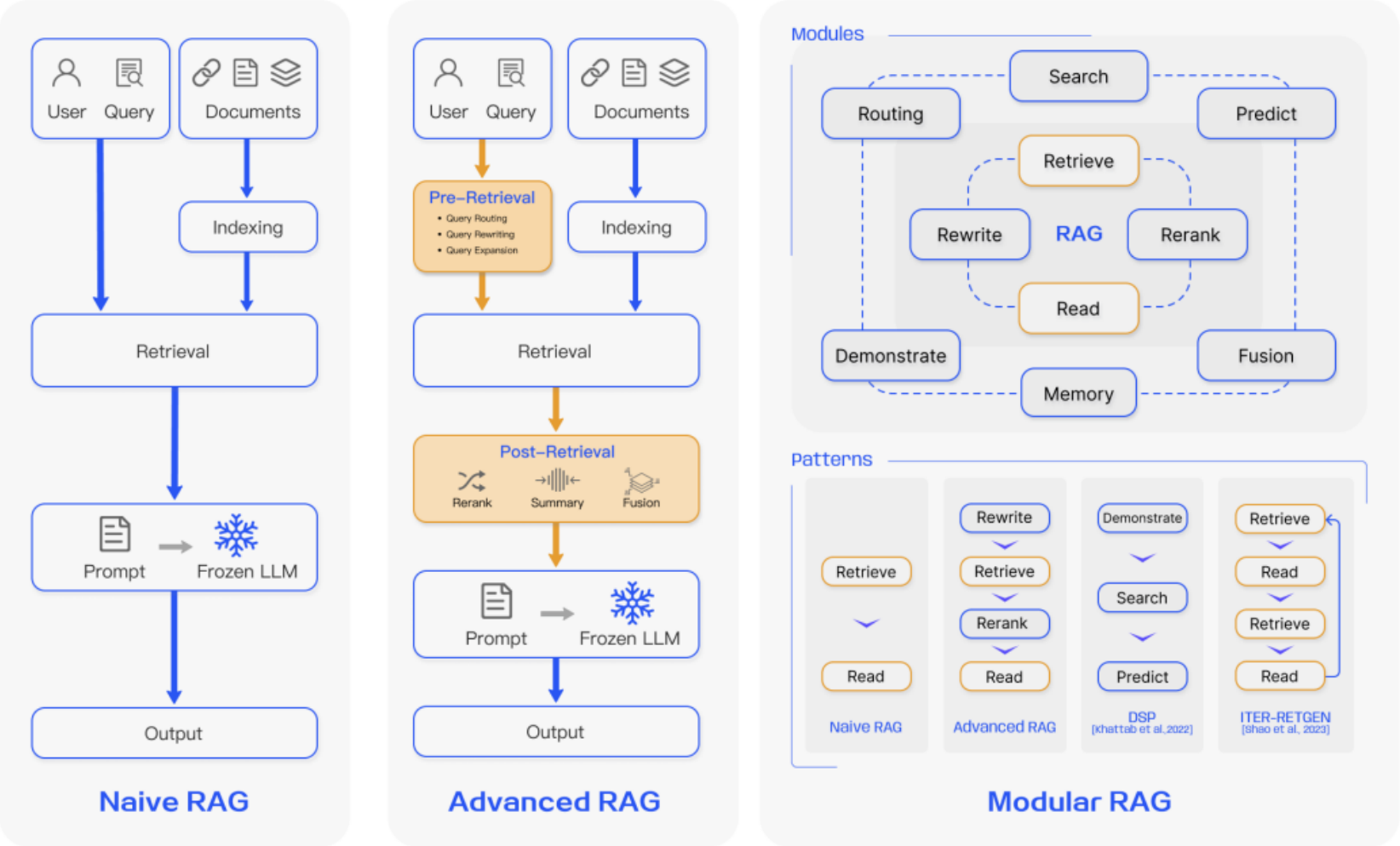


On nine categories of internal adversarially-designed factual evals, we compare GPT-4 (green) to the first three ChatGPT versions. There are significant gains across all topics. An accuracy of 1.0 means the model's answers are judged to be in agreement with human ideal responses for all questions in the eval.

Simple RAG Pipeline



RAG Paradigms



https://www.promptingguide.ai/research/rag

Question time!