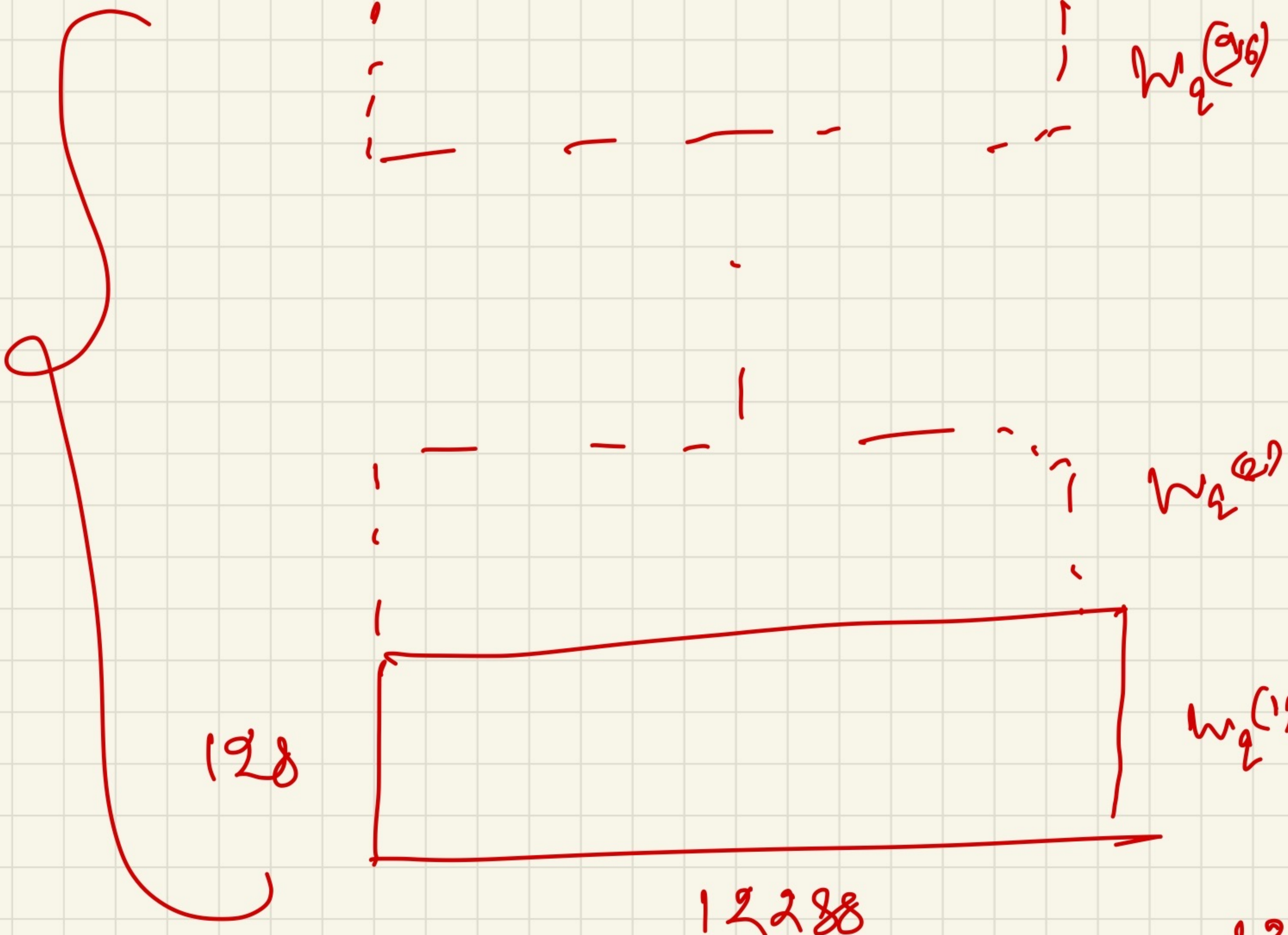


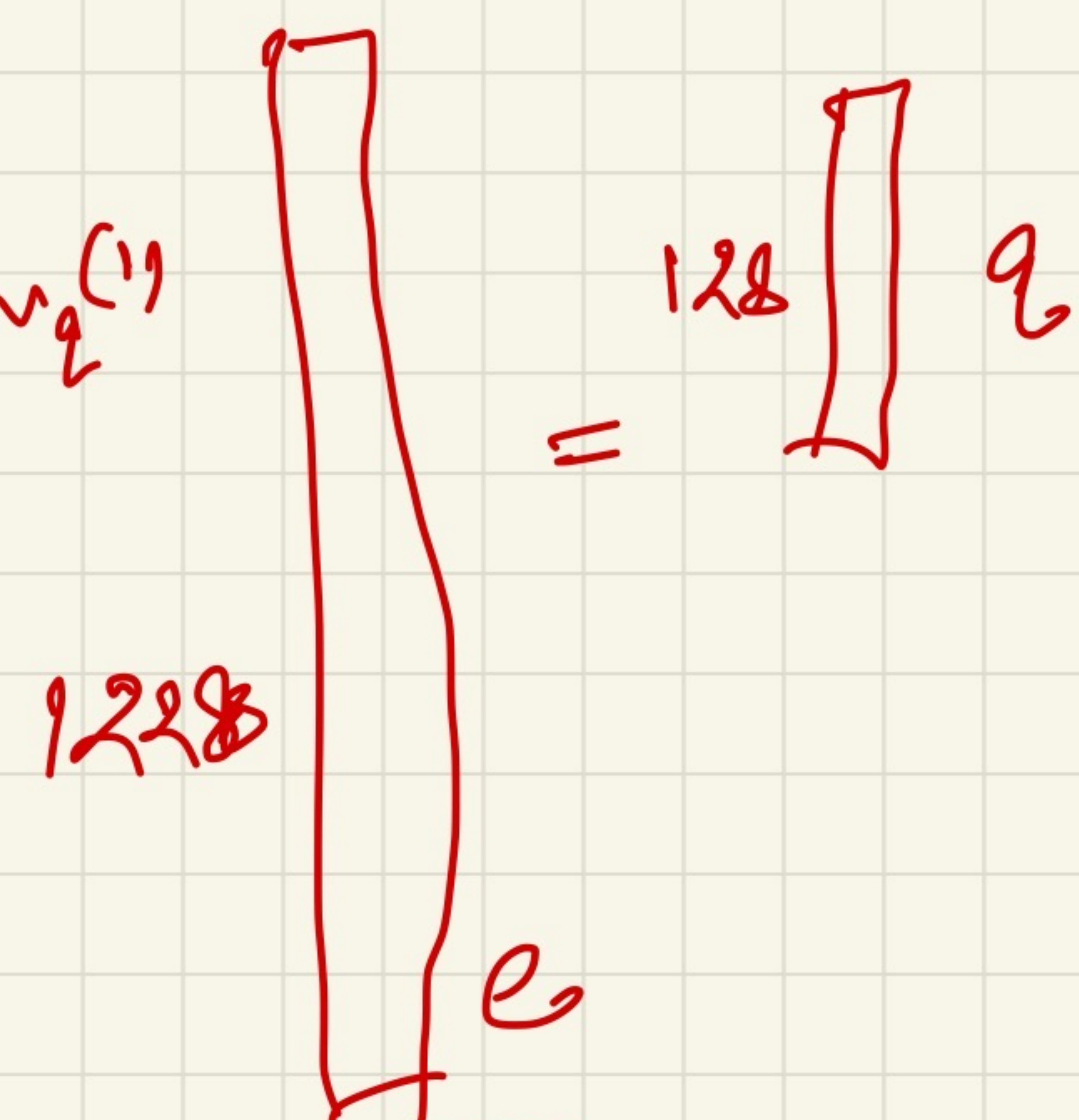
12288×12288

$\boxed{d_{model} \times 2 \times r}$

$\rightarrow \underline{B} \cdot \underline{A}$

$\underline{B} = d_{model} \times r$

$\underline{A} : d_{model} \times r$



$\boxed{r = 2}$

Which weight matrices to apply to?

Which weight matrices in Transformers should we apply LoRA to?

	# of Trainable Parameters = 18M						
Weight Type Rank r	W_q 8	W_k 8	W_v 8	W_o 8	W_q, W_k 4	W_q, W_v 4	W_q, W_k, W_v, W_o 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**

Adapting both Wq and Wv gives the best performance overall.

What is the optimal rank r for LoRA?

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

LoRA already performs competitively with a very small r

- For GPT-3, $d_{model} = 12288$, 96 decoders and 96 attention heads

LoRA	$r_v = 2$	4.7 M
	$r_q = r_v = 1$	4.7 M
	$r_q = r_v = 2$	9.4 M
	$r_q = r_k = r_v = r_o = 1$	9.4 M
	$r_q = r_v = 4$	18.8 M
	$r_q = r_k = r_v = r_o = 2$	18.8 M
	$r_q = r_v = 8$	37.7 M
	$r_q = r_k = r_v = r_o = 4$	37.7 M
	$r_q = r_v = 64$	301.9 M
	$r_q = r_k = r_v = r_o = 64$	603.8 M

Applying LoRA to Transformer

Model & Method	# Trainable Parameters	E2E NLG Challenge				
		BLEU	NIST	MET	ROUGE-L	CIDEr
GPT-2 M (FT)*	354.92M	68.2	8.62	46.2	71.0	2.47
GPT-2 M (Adapter^{L})*	0.37M	66.3	8.41	45.0	69.8	2.40
GPT-2 M (Adapter^{L})*	11.09M	68.9	8.71	46.1	71.3	2.47
GPT-2 M (Adapter^{H})	11.09M	$67.3_{\pm.6}$	$8.50_{\pm.07}$	$46.0_{\pm.2}$	$70.7_{\pm.2}$	$2.44_{\pm.01}$
GPT-2 M (FT^{Top2})*	25.19M	68.1	8.59	46.0	70.8	2.41
GPT-2 M (PreLayer)*	0.35M	69.7	8.81	46.1	71.4	2.49
GPT-2 M (LoRA)	0.35M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.85}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{71.8}_{\pm.1}$	$\mathbf{2.53}_{\pm.02}$
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (Adapter^{L})	0.88M	$69.1_{\pm.1}$	$8.68_{\pm.03}$	$46.3_{\pm.0}$	$71.4_{\pm.2}$	$\mathbf{2.49}_{\pm.0}$
GPT-2 L (Adapter^{L})	23.00M	$68.9_{\pm.3}$	$8.70_{\pm.04}$	$46.1_{\pm.1}$	$71.3_{\pm.2}$	$2.45_{\pm.02}$
GPT-2 L (PreLayer)*	0.77M	70.3	8.85	46.2	71.7	2.47
GPT-2 L (LoRA)	0.77M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.89}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{72.0}_{\pm.2}$	$2.47_{\pm.02}$

GPT-2 medium (M) and large (L) with different adaptation methods on the E2E NLG Challenge. For all metrics, higher is better. LoRA outperforms several baselines with comparable or fewer trainable parameters

From LoRA to QLoRA

- Uses a high-precision technique to quantize a pretrained model to $4-$bit.
- It then adds a small set of learnable Low-rank adapter weights, tuned by back-propagating gradients through the quantized weights.

How does the quantization work?

Quantizing a 32-bit floating point (FP32) tensor to Int8 integer with range $[-127, 127]$

$$\mathbf{X}^{\text{Int8}} = \text{round}\left(\frac{127}{\text{absmax}(\mathbf{X}^{\text{FP32}})}\mathbf{X}^{\text{FP32}}\right) = \text{round}(c^{\text{FP32}} \cdot \mathbf{X}^{\text{FP32}}),$$

$[-127, 127]$

where c is the *quantization constant* or *quantization scale*. Dequantization is the inverse:

$$\text{dequant}(c^{\text{FP32}}, \mathbf{X}^{\text{Int8}}) = \frac{\mathbf{X}^{\text{Int8}}}{c^{\text{FP32}}} = \mathbf{X}^{\text{FP32}}$$

QLoRA: Quantization Example for 2-bit

[-1 0.3 0.5 1]
0 1 2 3

[-1 -0.33 0.33 1]

[1 0.3 0.5 0.8]

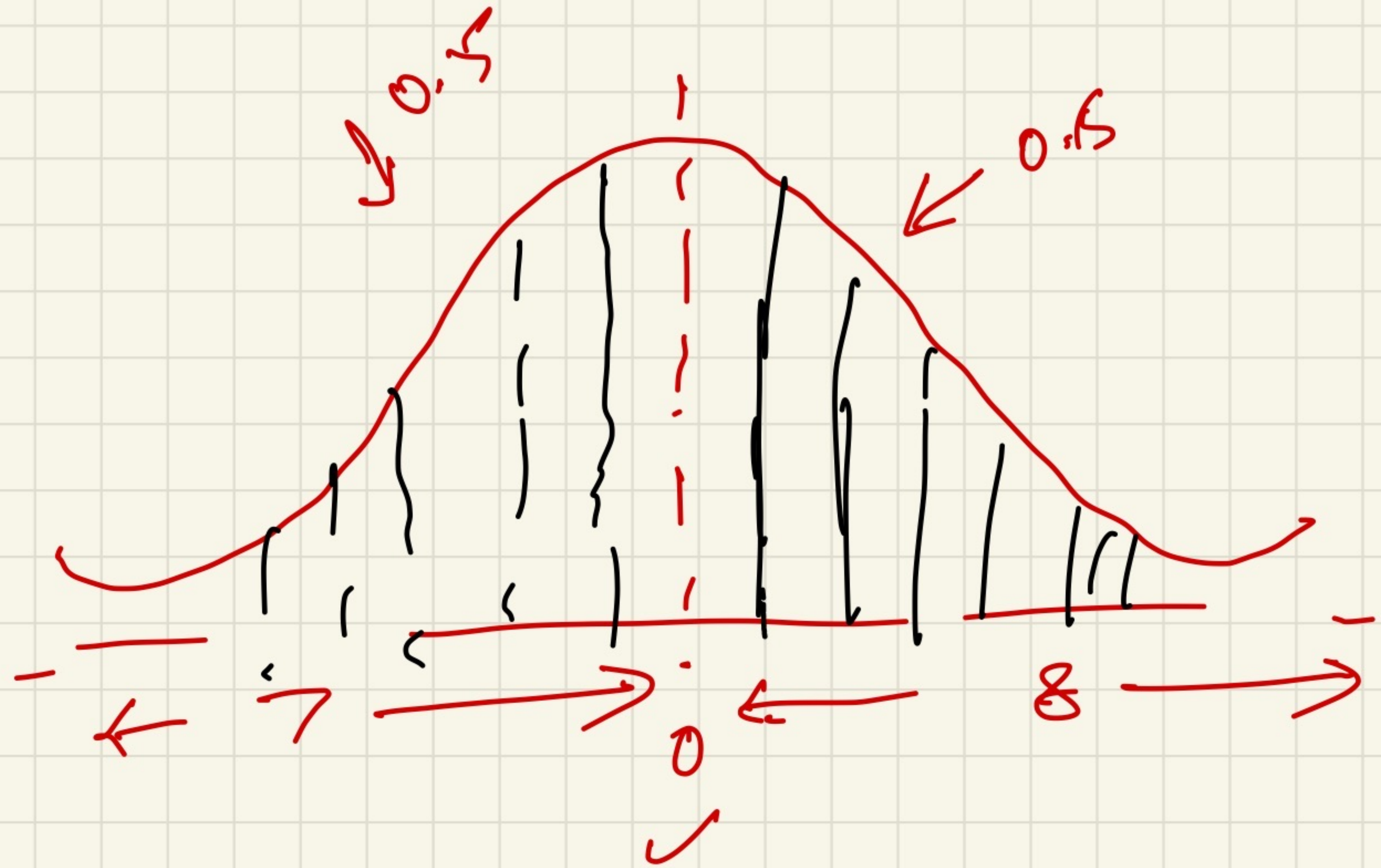
Map: {Index: 0, 1, 2, 3 -> Values: -1.0, 0.3, 0.5, 1.0}

Input tensor: $\boxed{[10, -3, 5, 4]}$ → [1 -0.3 0.5 0.4]

[3 1 2 1]

1. Normalize with absmax: [10, -3, 5, 4] -> [1, -0.3, 0.5, 0.4]
2. Find closest value: [1, -0.3, 0.5, 0.4] -> [1.0, 0.3, 0.5, 0.5]
3. Find the associated index: [1.0, 0.3, 0.5, 0.5] -> [3, 1, 2, 2] -> store
4. Dequantization: load -> [3, 1, 2, 2] -> lookup -> [1.0, 0.3, 0.5, 0.5] -> denormalize -> [10, 3, 5, 5] → error

NF4



QLoRA: Block-wise Quantization and other innovations

What is the issue with quantization?

Problem: If an outlier appears in the input, then some of the bins would not be utilized well with few or no numbers in those bins

Solution: Use block-wise $k-$bit quantization

Chunk input into contiguous blocks that are independently quantized, each with their quantization constant c

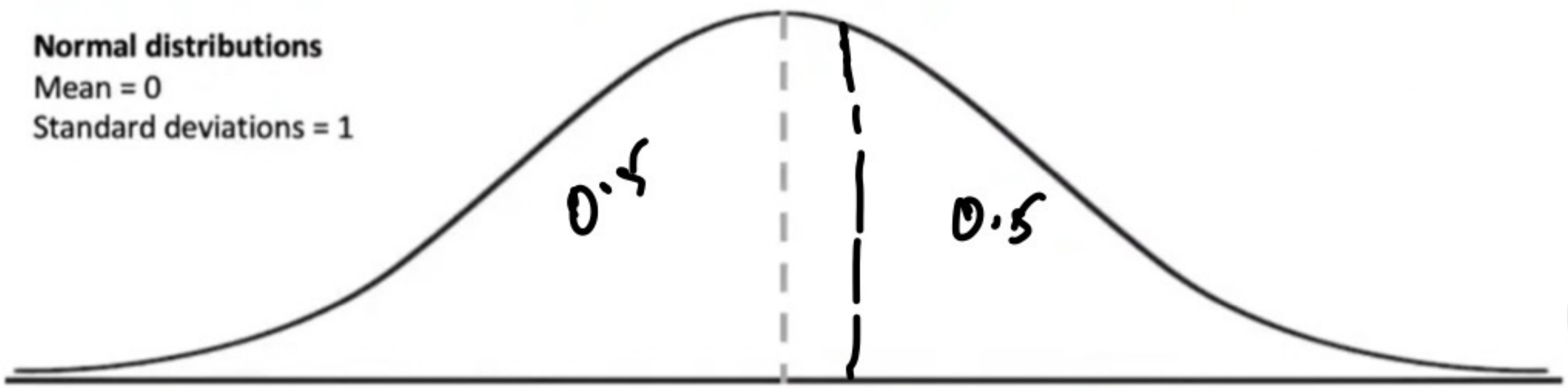
0.127 bit/param

Other innovations:

- For a block of size 64, a $32-$bit parameter is used as constant, so 0.5 bit/parameter. A **double quantization** can be used to save more.
- Use quantization where each quantization bin has an equal number of values assigned from the input

4-bit NormalFloat (NF4) Data Type

An information-theoretically optimal data type that ensures each quantization bin has an equal number of values assigned from the input tensor.



1.85

-1.85 [0.97, 0.9 , 0.83, 0.77, 0.7 , 0.63, 0.57] [0.56, 0.62, 0.68, 0.73, 0.79, 0.85, 0.91, 0.97] **(Probability)**

[-1.85, -1.29, -0.97, -0.73, -0.53, -0.34, -0.17] [0.15, 0.3, 0.45, 0.62, 0.81, 1.04, 1.34, 1.85] **(Z-score)**

[-1.85, -1.29, -0.97, -0.73, -0.53, -0.34, -0.17, 0 , 0.15, 0.3, 0.45, 0.62, 0.81, 1.04, 1.34, 1.85] **(Concatenation)**

[-1. , -0.7 , -0.53, -0.39, -0.28, -0.18, -0.09, 0. , 0.08, 0.16, 0.25, 0.34, 0.44, 0.56, 0.72, 1.] **(Normalisation)**

0000 1111

A comparison of the memory requirements

2 bytes parameters

2 bytes gradients

8 bytes → Adam

You are fine-tuning a 65B model

- Normal fine-tuning: 12 bytes per parameter → 780 GB
- LoRA fine-tuning: 17.6 bits per parameter → 143 GB
- QLoRA fine-tuning: 5.6 bits per parameter → 45 GB

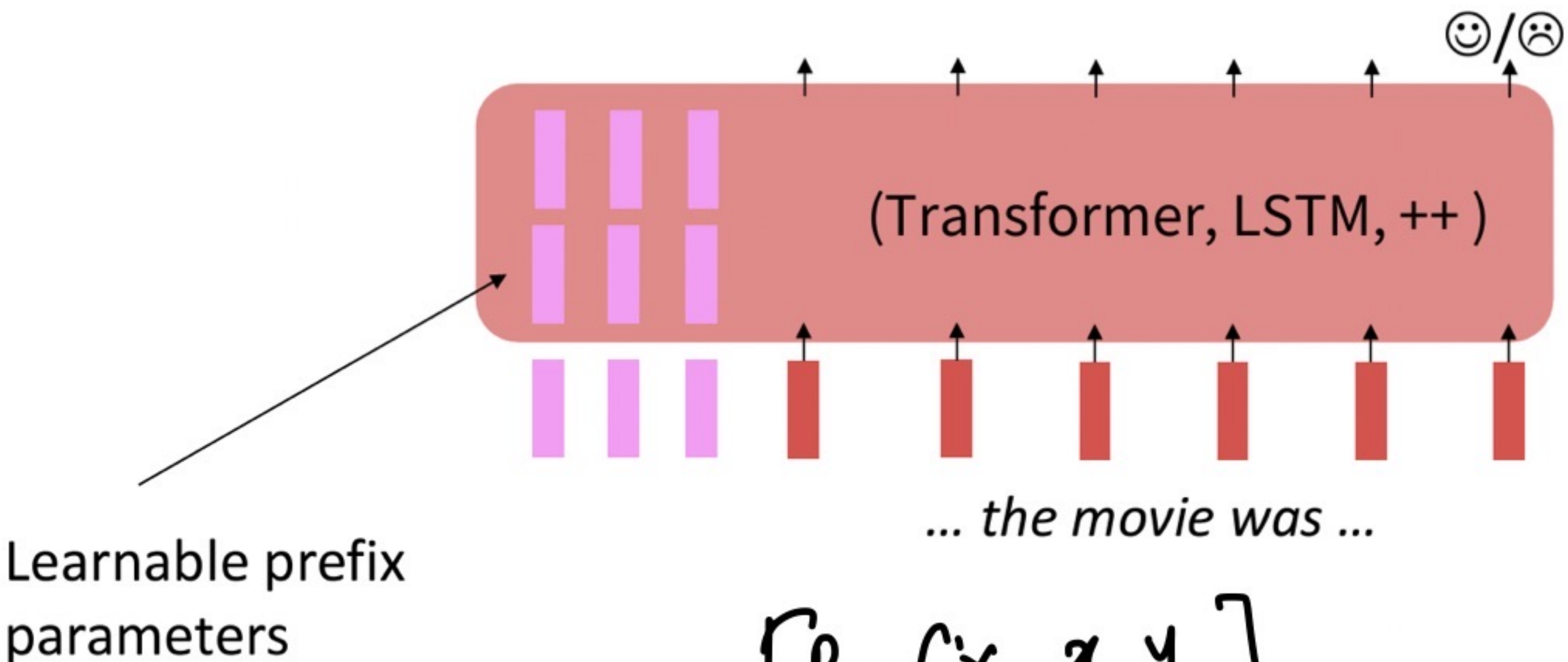
An input perspective: Prefix-tuning

Prefix-tuning adds a prefix of parameters, and freezes all pretrained parameters.

$[x, y]$

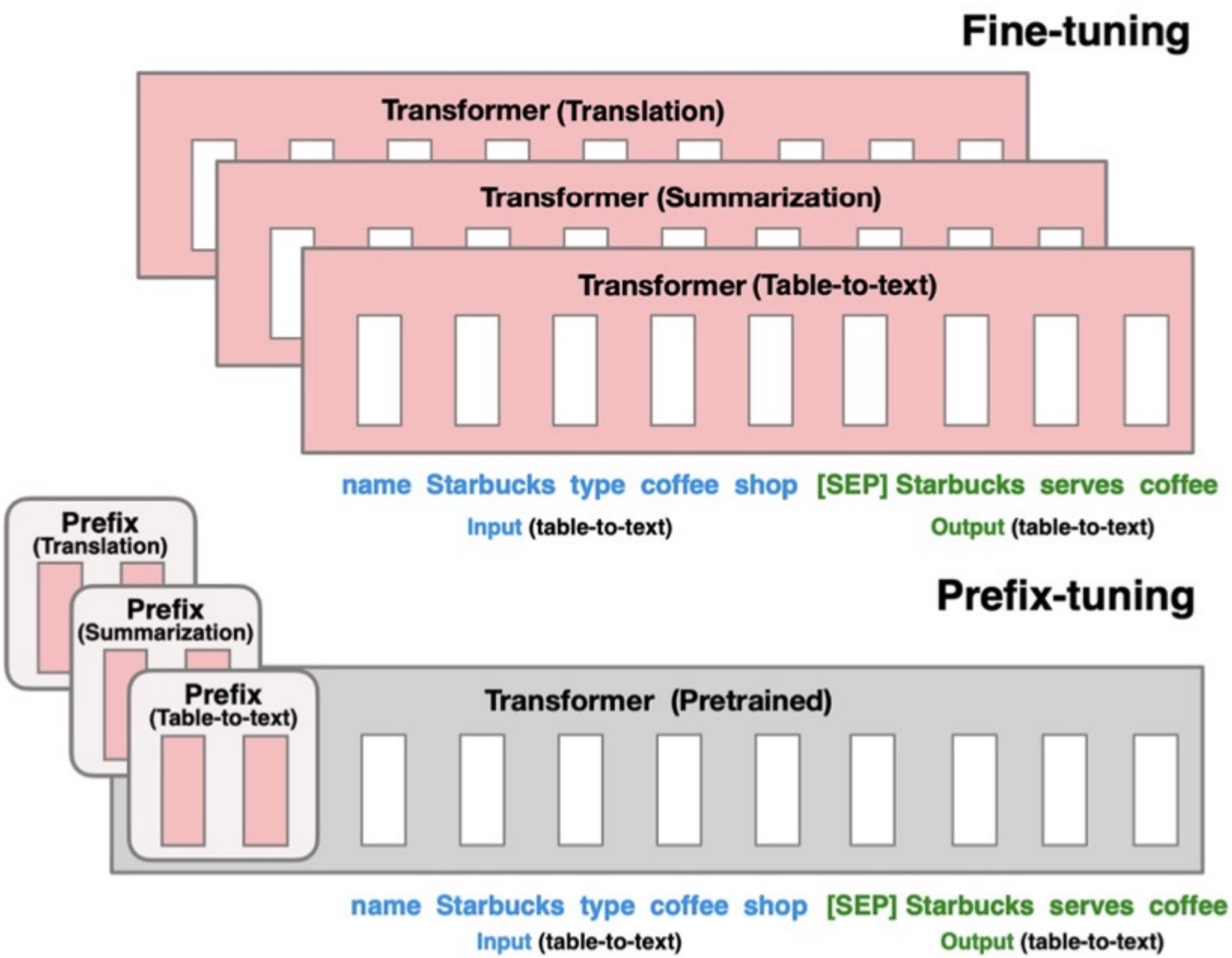
P_1 P_2 P_3

d_{model} d_{model} d_{model}



[prefix, x, y]

Prefix-tuning: We can learn task-specific prefixes



So, how does it work?

P_1 P_2 P_3

- Prefix-tuning prepends a prefix for an auto-regressive LM to obtain $z = \{PREFIX; x; y\}$
- Let P_{idx} denote the sequence of prefix indices, and $|P_{idx}|$ denote the length of the prefix.

trainable params

$|P_{idx}| \times d_{model} \times L$

So, how does it work?

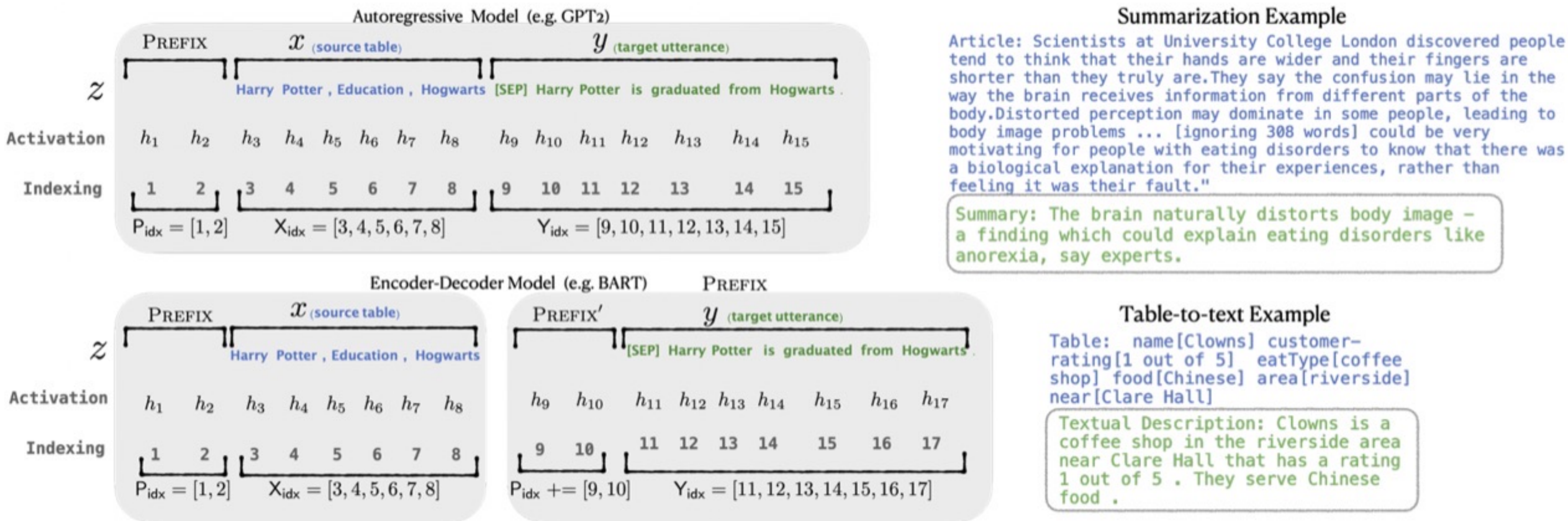


Figure 2: An annotated example of prefix-tuning using an autoregressive LM (top) and an encoder-decoder model (bottom). The prefix activations $\forall i \in \mathsf{P}_{\mathsf{idx}}, h_i$ are drawn from a trainable matrix P_θ. The remaining activations are computed by the Transformer.

Which layers do we tune?

Prefix-tuning only the embeddings

- When only at the embedding layer, trainable parameters are $d_{model} \times |P_{idx}|$.
- In subsequent layers, operations are like usual transformer.

Prefix-tuning all the layers

- When applied at all layers, trainable parameters are $d_{model} \times |P_{idx}| \times L$.
- In subsequent layers, operations are like usual transformer for all other tokens. For prefix tokens, activations are taken directly from a trainable parameter set.

Can we also do infixing?

l_p l_i

- $z = \{PREFIX; x; y\}$ vs. $z = \{x; INFIX; y\}$
- For decoder-only models, while prefixing can affect the activations of both x and y, infixing only impacts the activations of y
- In general, infixing slightly underperforms prefix, but both can also be used together.