Finetuning a Transformer

Mohamed Abbas KONATE

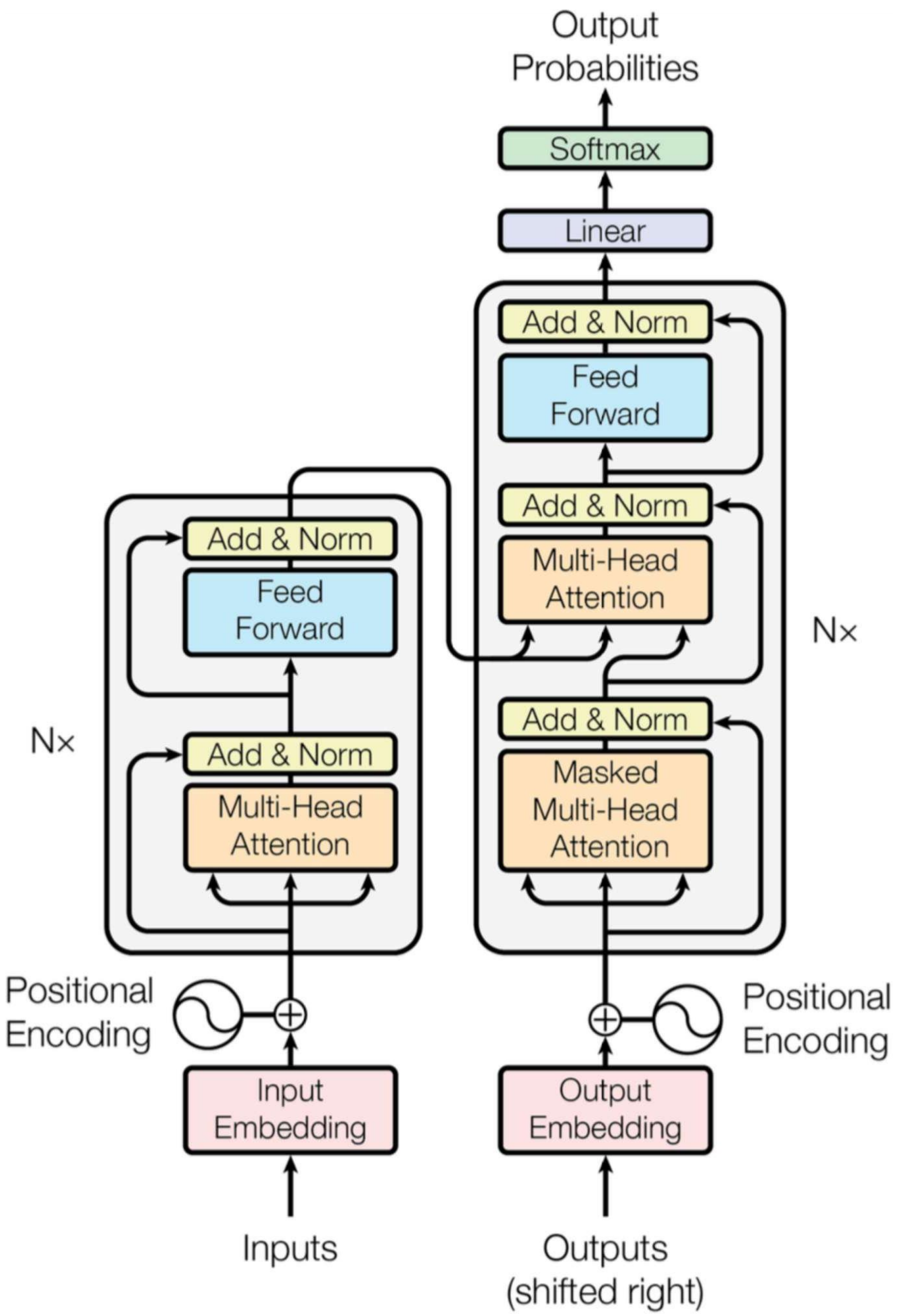
Plan

Finetuning : kesako ?
Le challenge du finetuning
Knowledge Distillation
Lora
Adapters

But du chapitre : Découvrir comment on réduit la taille des paramètres et comment finetuner un modèle

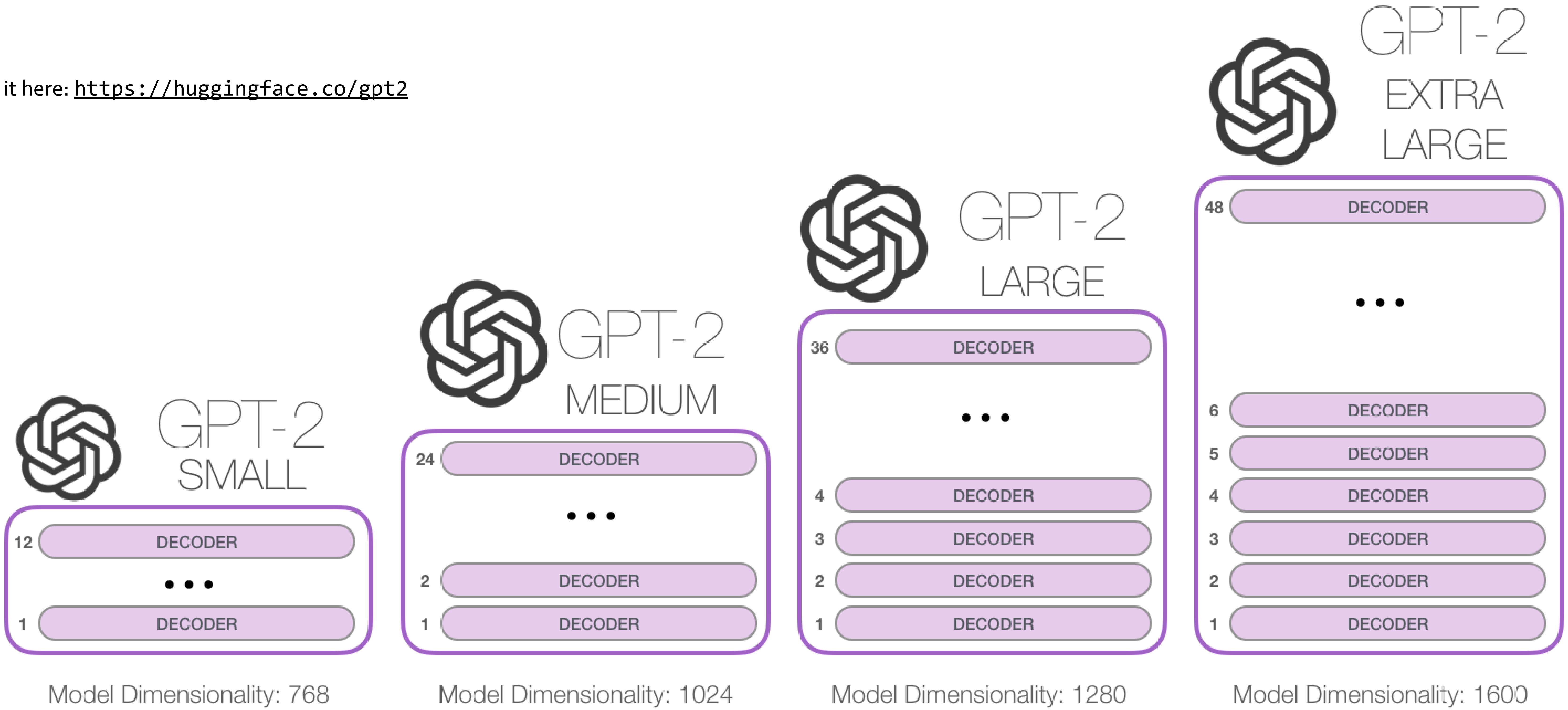
Rappel

Rappel


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Nx
Add & Norm
Multi-Head Attention
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

GPT2: Model Sizes

Play with it here: https://huggingface.co/gpt2



117M parameters 345M 762M 1542M

[Image by http://jalammar.github.io/illustrated-gpt2/]

GPT-3: A Very Large Language Model (2020)

More layers & parameters
Bigger dataset
Longer training
Larger embedding/hidden dimension
Larger context window

[Slide credit: Sbhya Chhabria & Michael Tang]

Transformer FLOPs: The Quick Estimate

- Let N be number of parameters (the sum of size of all matrices)
- Let D be the number of tokens in pre-training dataset.
- The total cost of pre-training on this dataset is:

$$C \sim 6ND$$

- You can already see how this relates to our constraints:
  - If you have a fixed compute budget C, increasing D means decreasing N (and vice versa).

Model compression

Compression: An Overview

Quantization

Stores or performs computation on 4/8 bit integers instead of 16/32 bit floating point numbers.

The most effective and practical way do training/inference of a large model.

Can be combined with pruning (GPTQ) and Distillation (ZeroQuant).

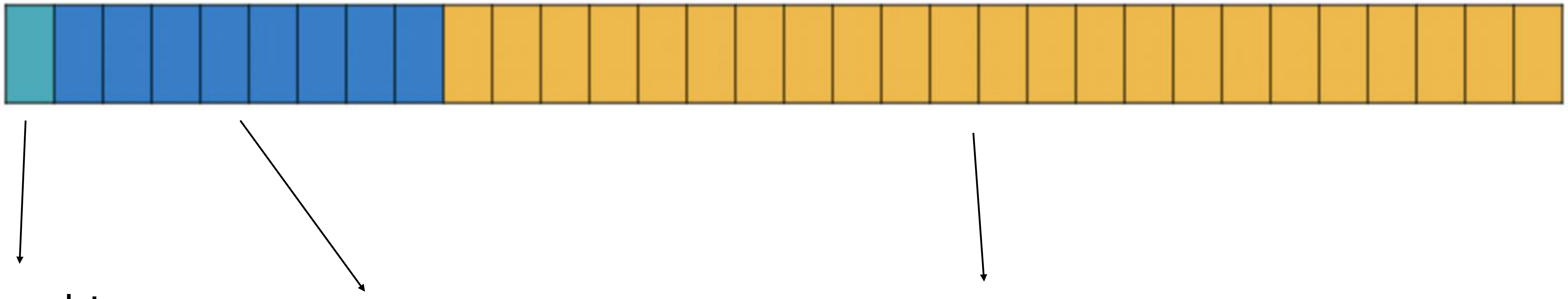
Today!

Numeric Data Types

How numbers are represented in modern computing systems

Floating-Point Numbers

Example: 32-bit floating-point number in IEEE 754 (FP32)



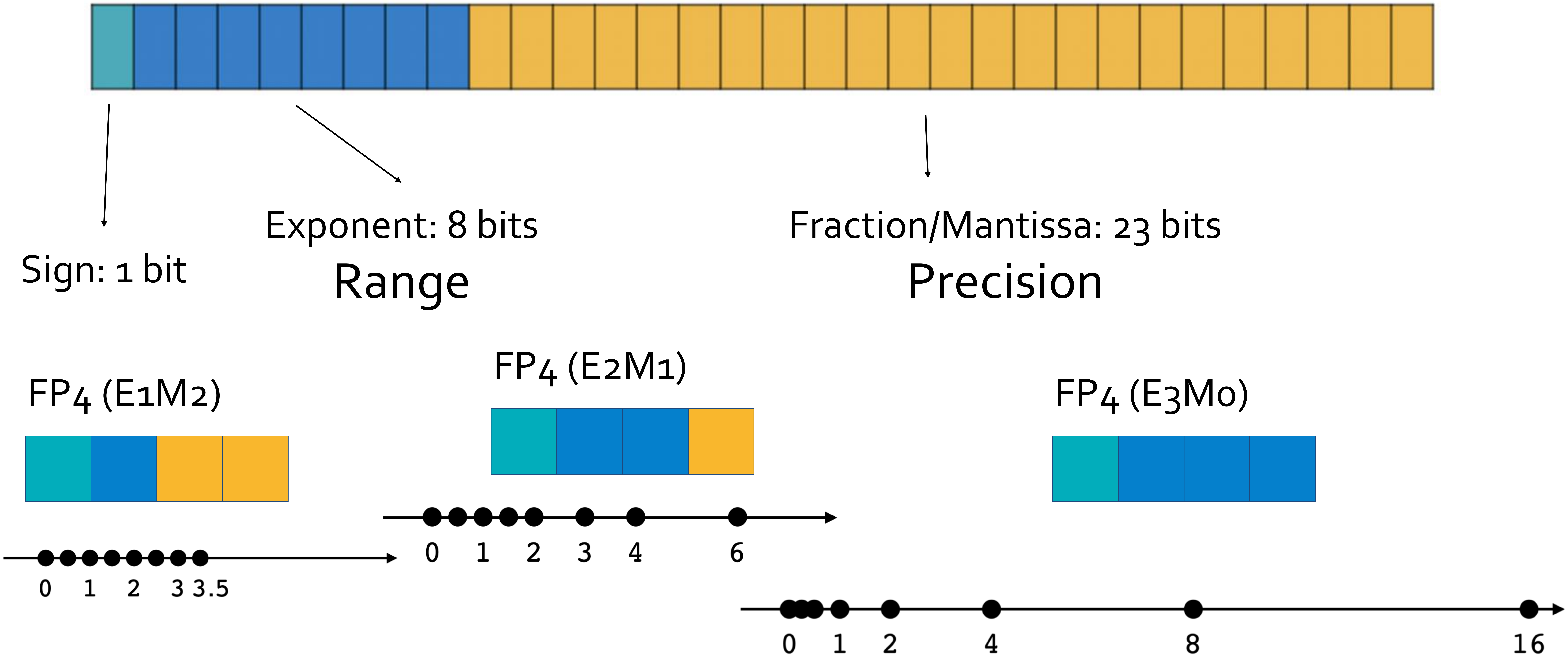
Sign: 1 bit

Exponent: 8 bits

Fraction/Mantissa: 23 bits

$$\text{Number} = (-1)^{\text{sign}} \times (1 + \text{Fraction}) \times 2^{\text{Exponent - 127}}$$

Floating-Point Numbers



Floating-Point Numbers

Exponent Fraction

IEEE 754 Single Precision 32-bit Float (FP32)

8 23

IEEE 754 Half Precision 16-bit Float (FP16)

5 10

Google Brain Float (BF 16)

8 7

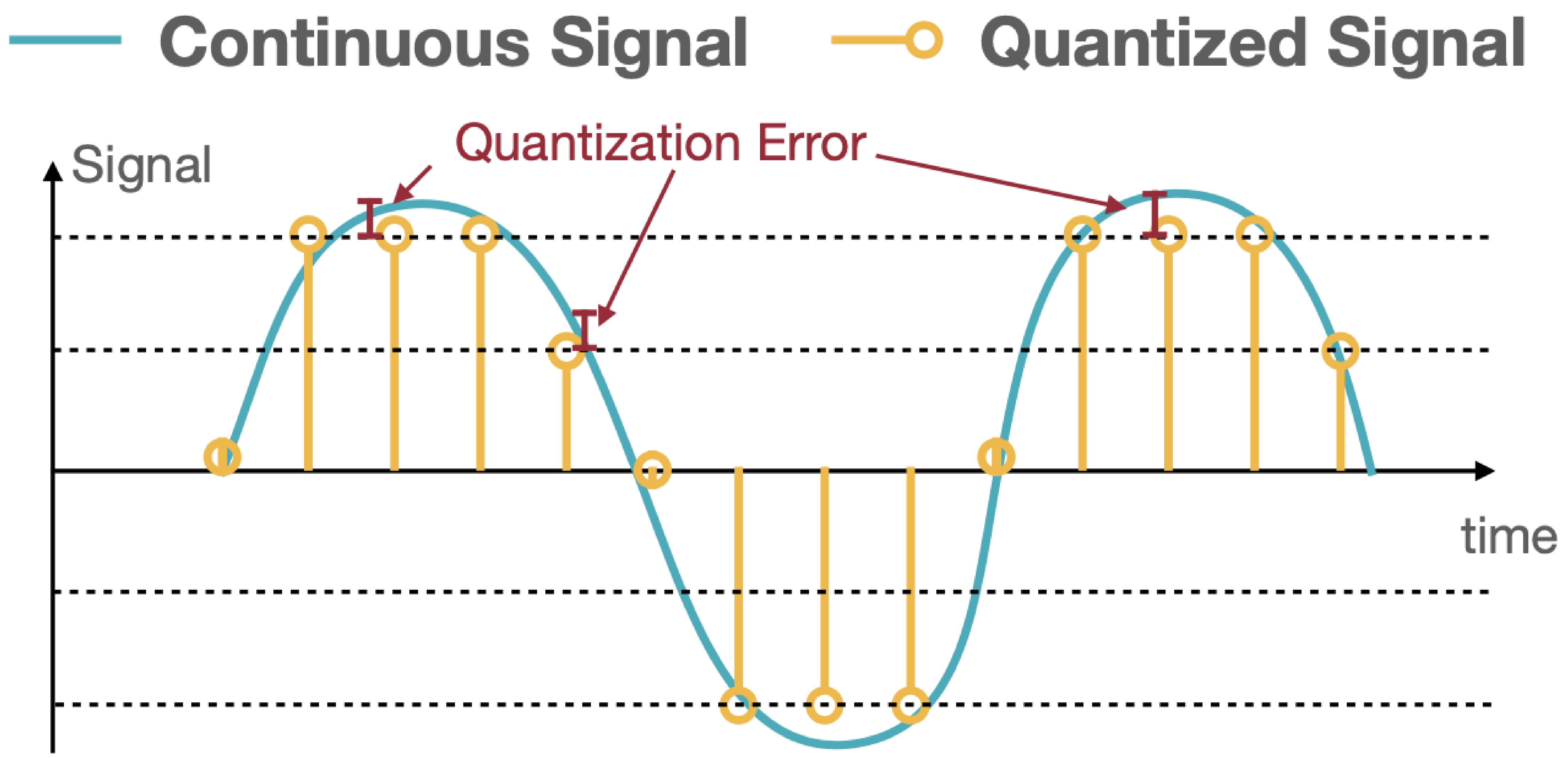
Nvidia FP8 (E4M3)

4 3

Quantization

Representing numbers using a discrete set

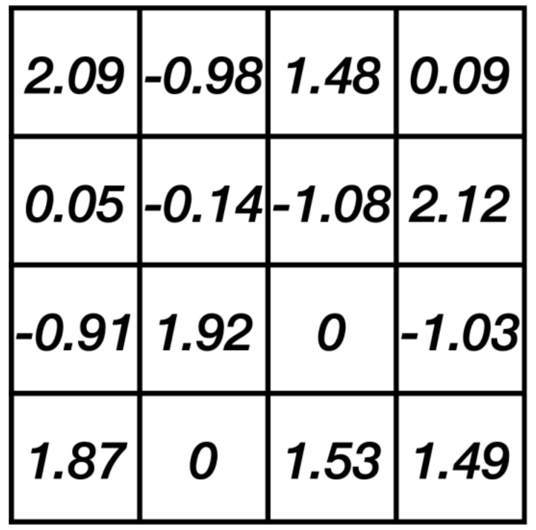
What is Quantization?

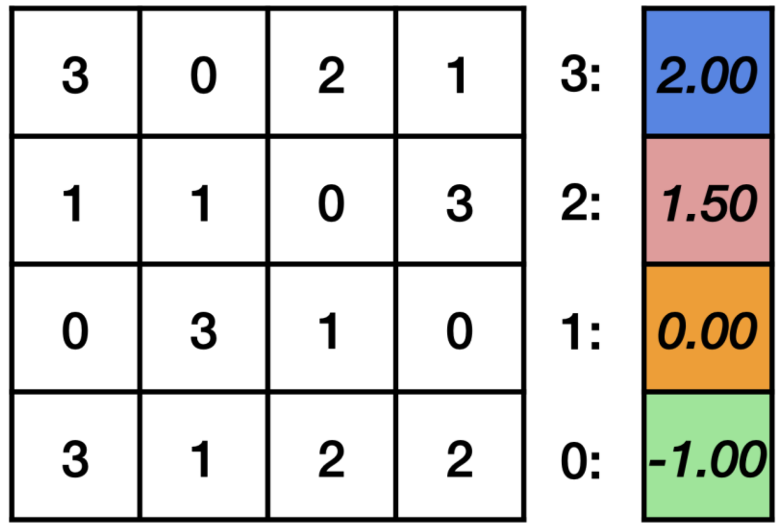


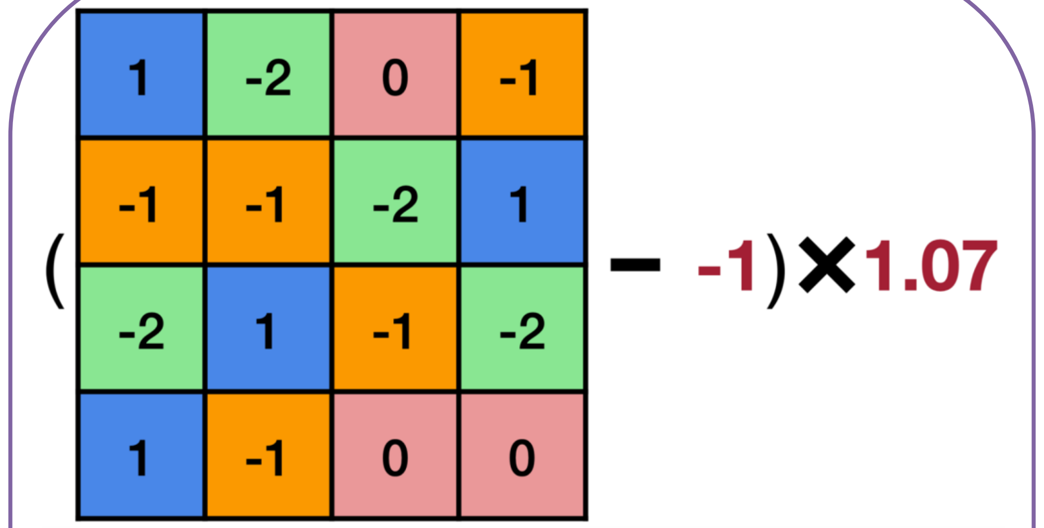
The process of mapping input values from a large set (often a continuous set) to output values in a (countable) smaller set, often with a finite number of elements.

Overview of Quantization Methods

Today's Focus







		K-Means	Linear
Storage	Floating Point	Integer Weights; Floating Point Codebook	Integer
Computation	Floating Point	Floating Point	Integer

Linear Quantization

Affine Mapping from floating point numbers to integers

Original 32-bit float, Quantized 2-bit signed int, Zero point, Scale, Reconstructed 32-bit float

$$\underbrace{\begin{bmatrix} 2.09 & -0.98 & 1.48 & 0.09 \\ 0.05 & -0.14 & -1.08 & 2.12 \\ -0.91 & 1.92 & 0 & -1.03 \\ 1.87 & 0 & 1.53 & 1.49 \end{bmatrix}}_{\text{Original 32-bit float}} \Rightarrow \Bigg(\underbrace{\begin{bmatrix} 1 & -2 & 0 & -1 \\ -1 & -1 & -2 & 1 \\ -2 & 1 & -1 & -2 \\ 1 & -1 & 0 & 0 \end{bmatrix}}_{\text{Quantized 2-bit signed int}} - \underbrace{-1}_{\text{Zero point}} \Bigg) \times \underbrace{1.07}_{\text{Scale}} = \underbrace{\begin{bmatrix} 2.14 & -1.07 & 1.07 & 0 \\ 0 & 0 & -1.07 & 2.14 \\ -1.07 & 2.14 & 0 & -1.07 \\ 2.14 & 0 & 1.07 & 1.07 \end{bmatrix}}_{\text{Reconstructed 32-bit float}}$$

How to find these numbers?

Linear Quantization

Affine Mapping from floating point numbers to integers

Original 32-bit float			
2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Quantized 2-bit signed int			
1	-2	0	-1
-1	-1	-2	1
-2	1	-1	-2
1	-1	0	0

Zero point: -1

Scale: 1.07

Reconstructed 32-bit float			
2.14	-1.07	1.07	0
0	0	-1.07	2.14
-1.07	2.14	0	-1.07
2.14	0	1.07	1.07

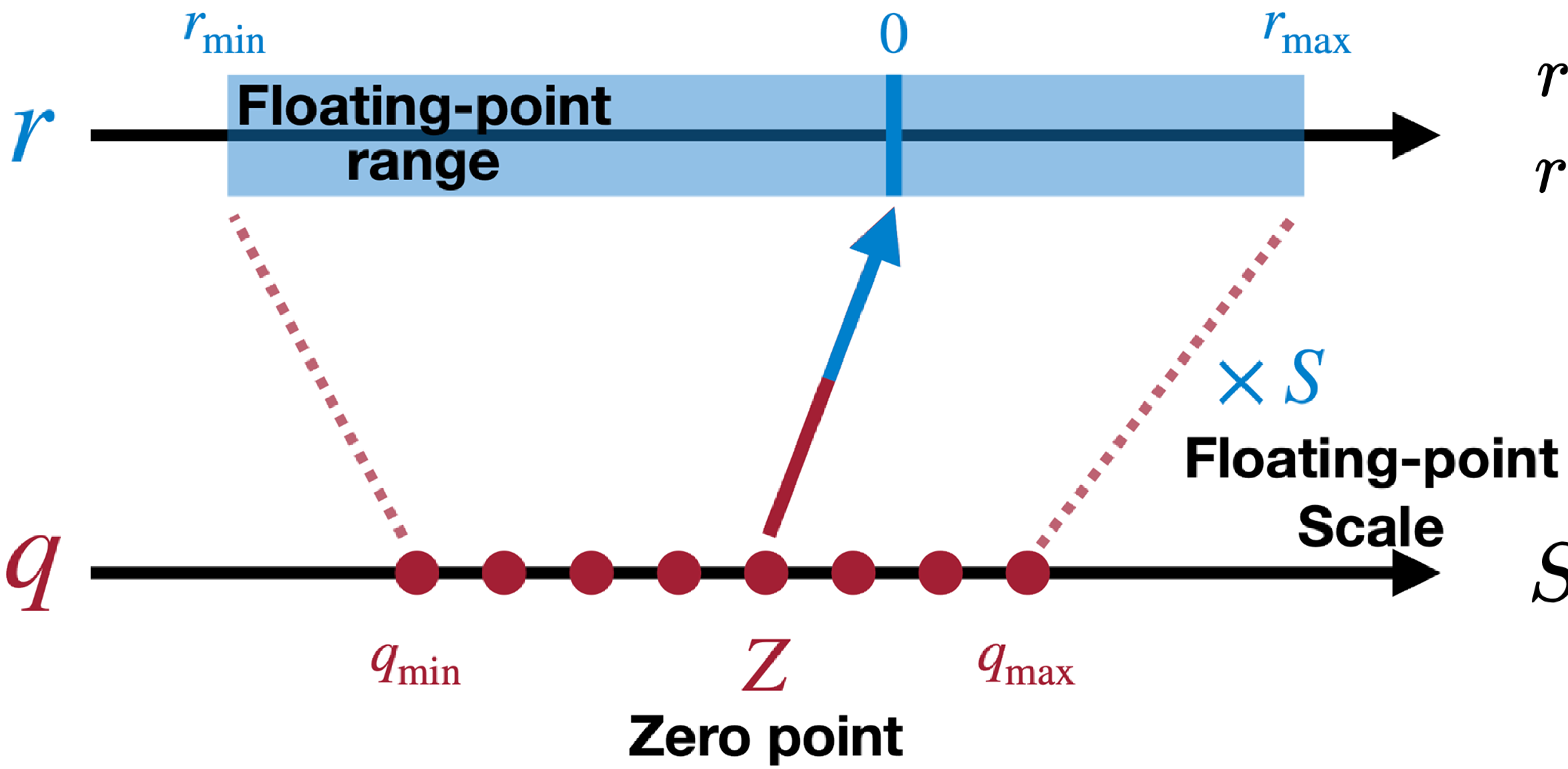
$$\left(\begin{bmatrix} 1 & -2 & 0 & -1 \\ -1 & -1 & -2 & 1 \\ -2 & 1 & -1 & -2 \\ 1 & -1 & 0 & 0 \end{bmatrix} - (-1) \right) \times 1.07 = \begin{bmatrix} 2.14 & -1.07 & 1.07 & 0 \\ 0 & 0 & -1.07 & 2.14 \\ -1.07 & 2.14 & 0 & -1.07 \\ 2.14 & 0 & 1.07 & 1.07 \end{bmatrix}$$

$$\underbrace{r}_{\text{floating-point}} \approx (\underbrace{q}_{\text{integer}} - \underbrace{Z}_{\text{integer}}) \times \underbrace{S}_{\text{floating-point}}$$

Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference (Jacob et al., CVPR 2018)

Linear Quantization

Scale Derivation | r = S(q-z)



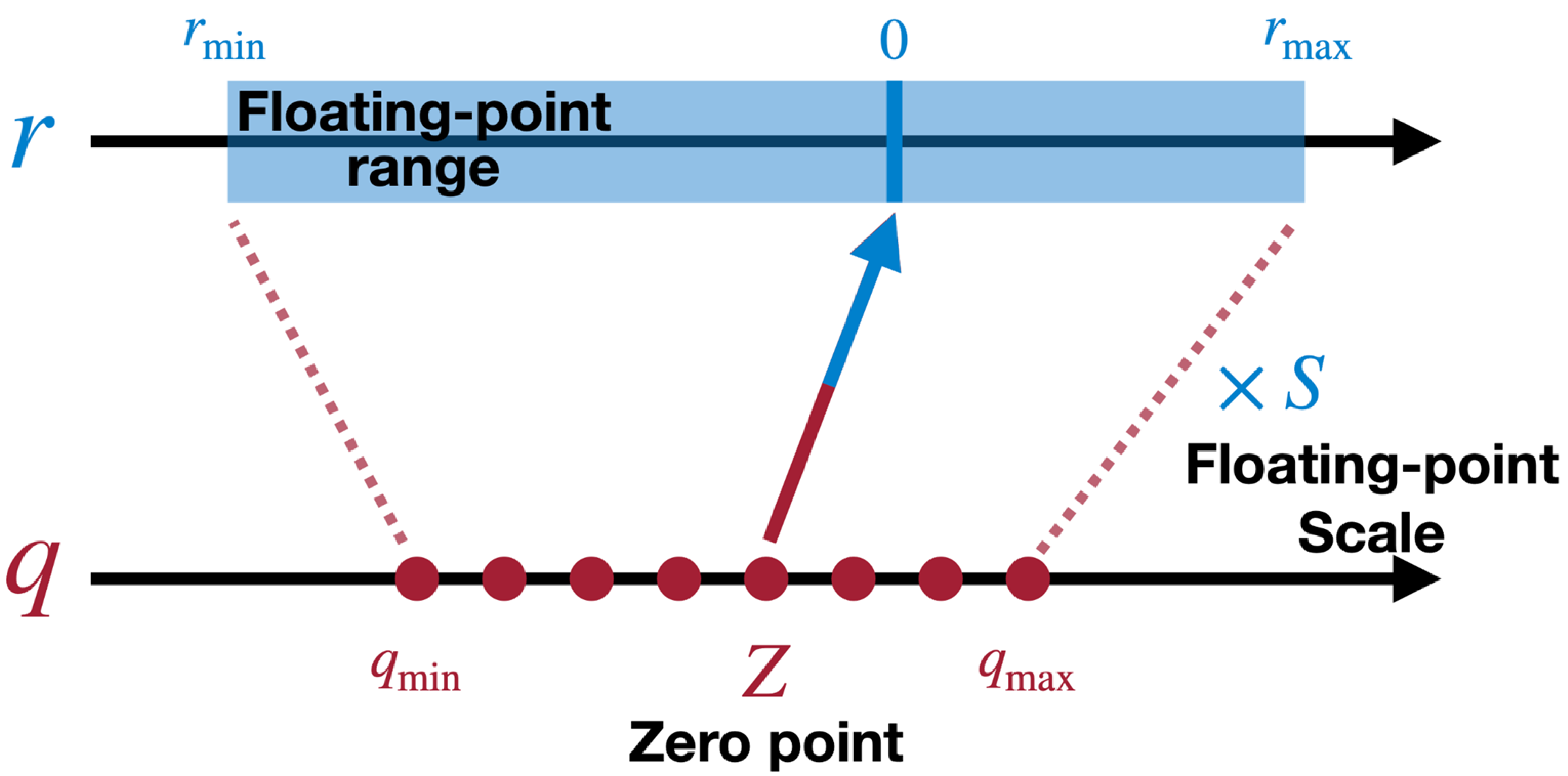
$$r_{max} = S(q_{max} - Z)$$
$$r_{min} = S(q_{min} - Z)$$

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference (Jacob et al., CVPR 2018)

Linear Quantization

Zero point Derivation | r = S(q-z)



$$r_{\min} = S\left(q_{\min} - Z\right)$$

$$Z = q_{\min} - \frac{r_{\min}}{S}$$

$$Z = \operatorname{round}\left(q_{\min} - \frac{r_{\min}}{S}\right)$$

Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference (Jacob et al., CVPR 2018)

GPTQ: Experiment Results

OPT	Bits	125M	350M	1.3B	2.7B	6.7B	13B	30B	66B	175B
full	16	27.65	22.00	14.63	12.47	10.86	10.13	9.56	9.34	8.34
RTN	4	37.28	25.94	48.17	16.92	12.10	11.32	10.98	110	10.54
GPTQ	4	**31.12**	**24.24**	**15.47**	**12.87**	**11.39**	**10.31**	**9.63**	**9.55**	**8.37**
RTN	3	1.3e3	64.57	1.3e4	1.6e4	5.8e3	3.4e3	1.6e3	6.1e3	7.3e3
GPTQ	3	**53.85**	**33.79**	**20.97**	**16.88**	**14.86**	**11.61**	**10.27**	**14.16**	**8.68**

Table 3: OPT perplexity results on WikiText2.

BLOOM	Bits	560M	1.1B	1.7B	3B	7.1B	176B
full	16	22.42	17.69	15.39	13.48	11.37	8.11
RTN	4	25.90	22.00	16.97	14.76	12.10	8.37
GPTQ	4	**24.03**	**19.05**	**16.48**	**14.20**	**11.73**	**8.21**
RTN	3	57.08	50.19	63.59	39.36	17.38	571
GPTQ	3	**32.31**	**25.08**	**21.11**	**17.40**	**13.47**	**8.64**

Table 4: BLOOM perplexity results for WikiText2.

GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers (Frantar et al., ICLR 2023)

Compression: An Overview

Quantization

Stores or performs computation on 4/8 bit integers instead of 16/32 bit floating point numbers.

The most effective and practical way do training/inference of a large model.

Can be combined with pruning (GPTQ) and Distillation (ZeroQuant).

Distillation

Train a small model (the student) on the outputs of a large model (the teacher).

In essence, distillation = model ensembling. Therefore we can distill between model with the same architecture (self-distillation)

Can be combined with pruning.

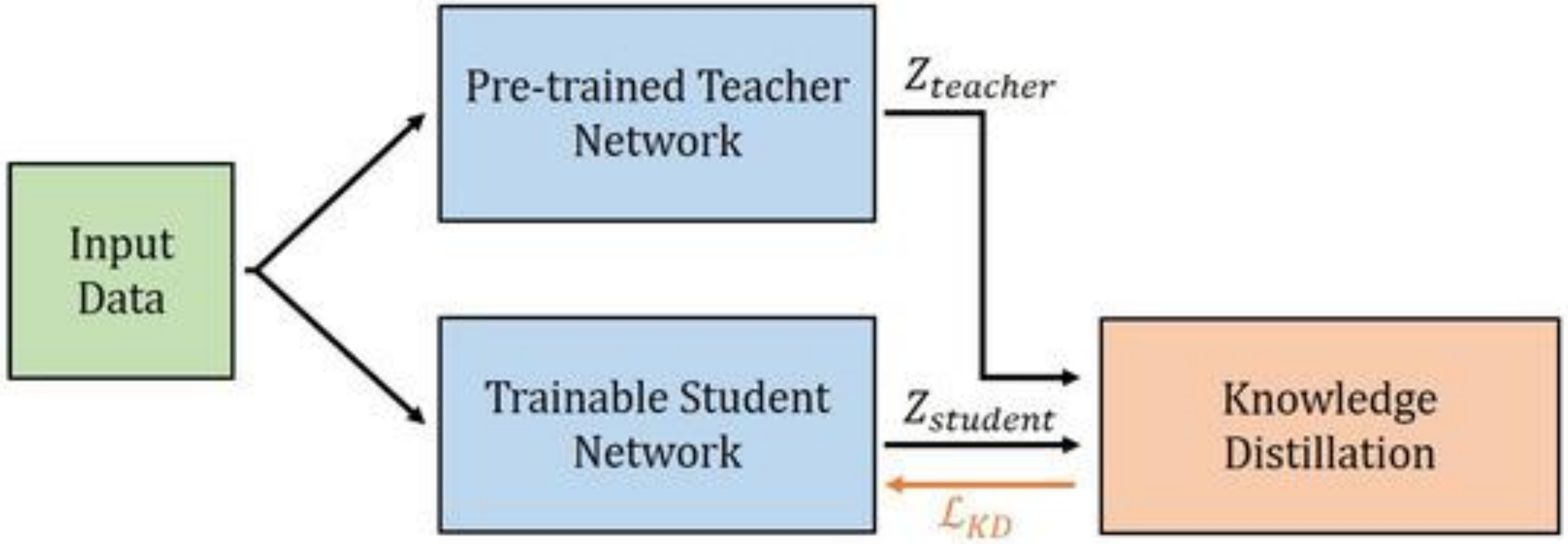
Today!

Distillation

Training a small model to match the distribution of a large one

Distillation

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right).$$



Training objective:

Minimizing KL Divergence between teacher output and student output

Essentially: We are using the soft labels from the teacher to train student

Distilling the Knowledge in Neural Networks (Hinton et al., 2014)

Compression: An Overview

Quantization

Stores or performs computation on 4/8 bit integers instead of 16/32 bit floating point numbers.

The most effective and practical way do training/inference of a large model.

Can be combined with pruning (GPTQ) and Distillation (ZeroQuant).

Distillation

Train a small model (the student) on the outputs of a large model (the teacher).

In essence, distillation = model ensembling. Therefore we can distill between model with the same architecture (self-distillation)

Can be combined with pruning.

Pruning

Removing excessive model weights to lower parameter count.

A lot of the work are done solely for research purposes.

Cultivated different routes of estimating importances of parameters.

Today!

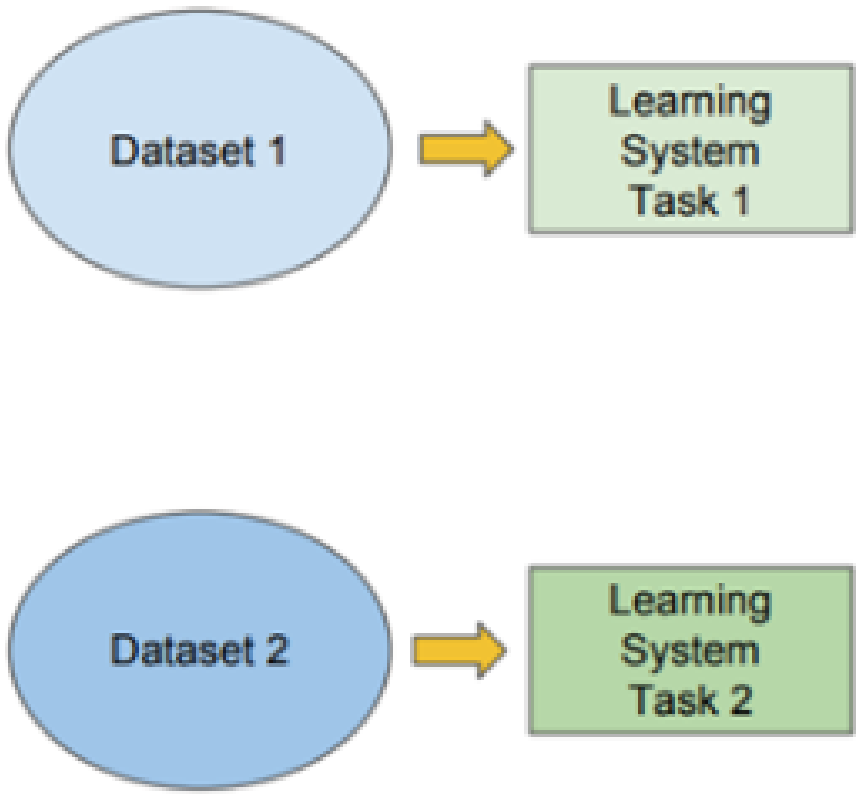
Adaptation via Fine-Tuning

Transfer Learning

Traditional ML vs Transfer Learning

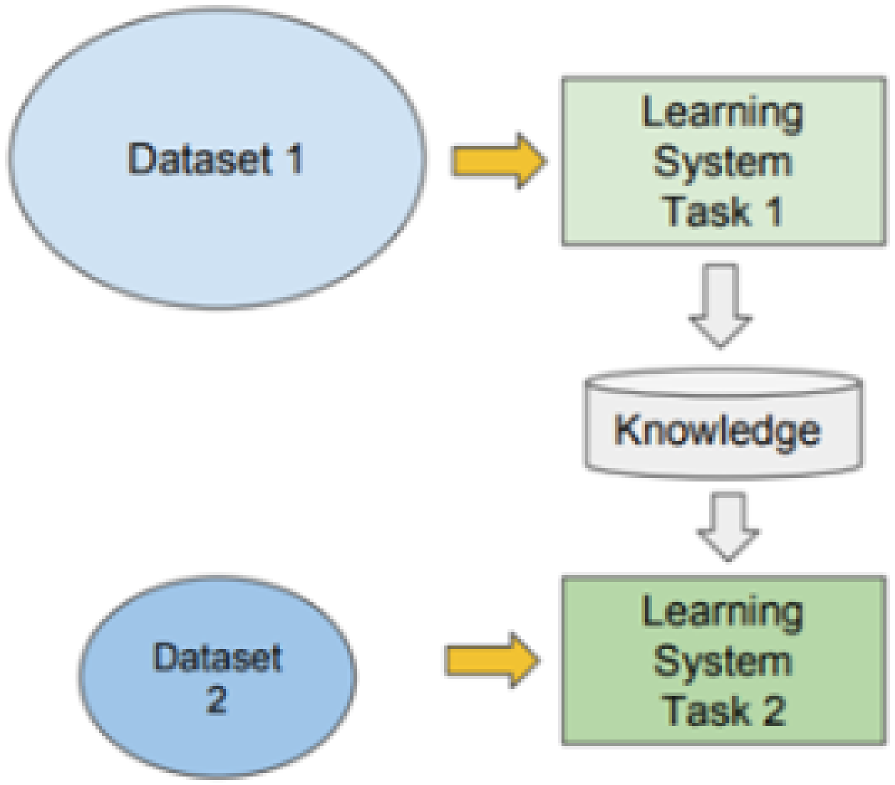
Traditional ML

- Isolated, single task learning:
  - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks

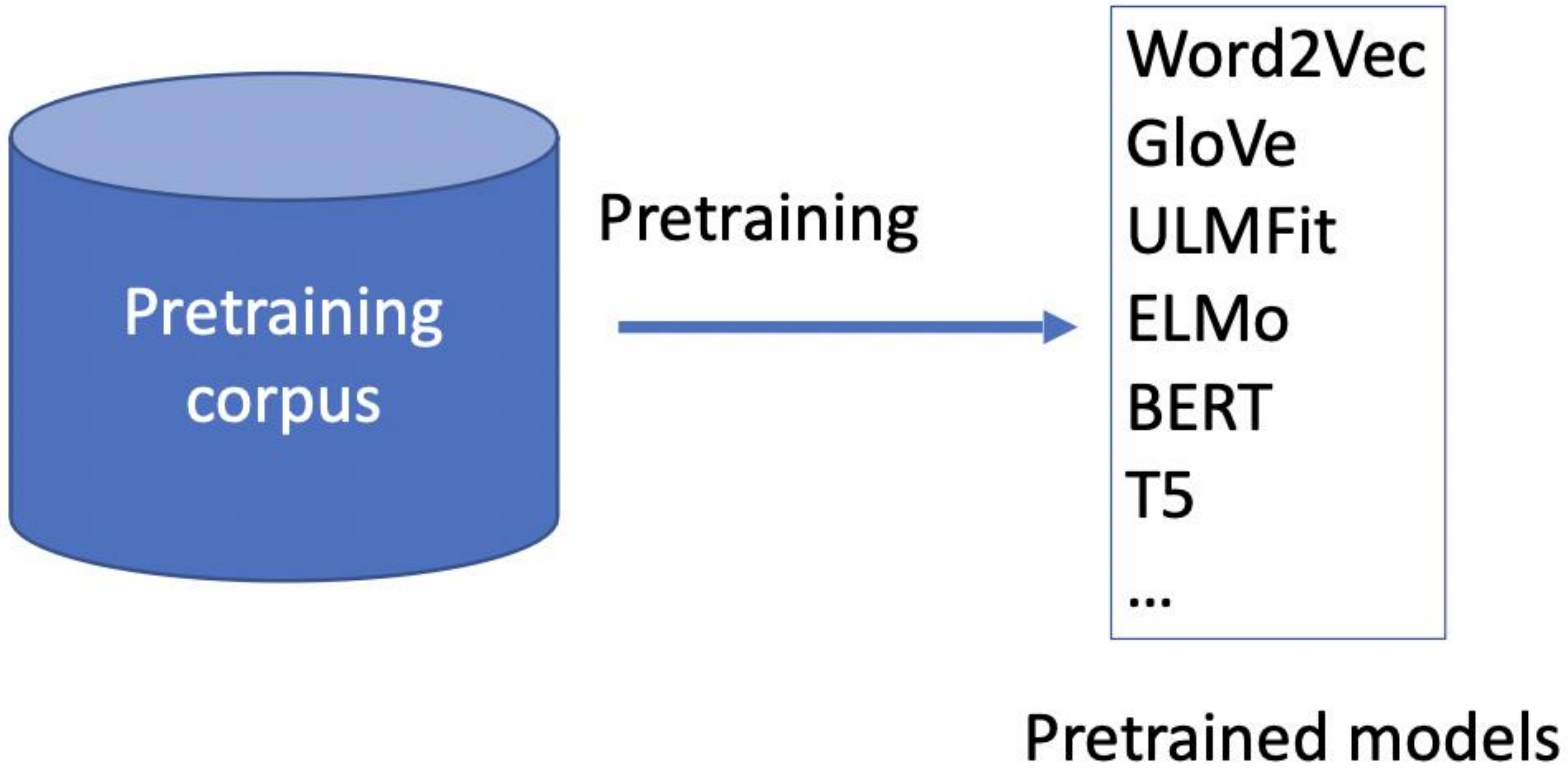


Transfer Learning

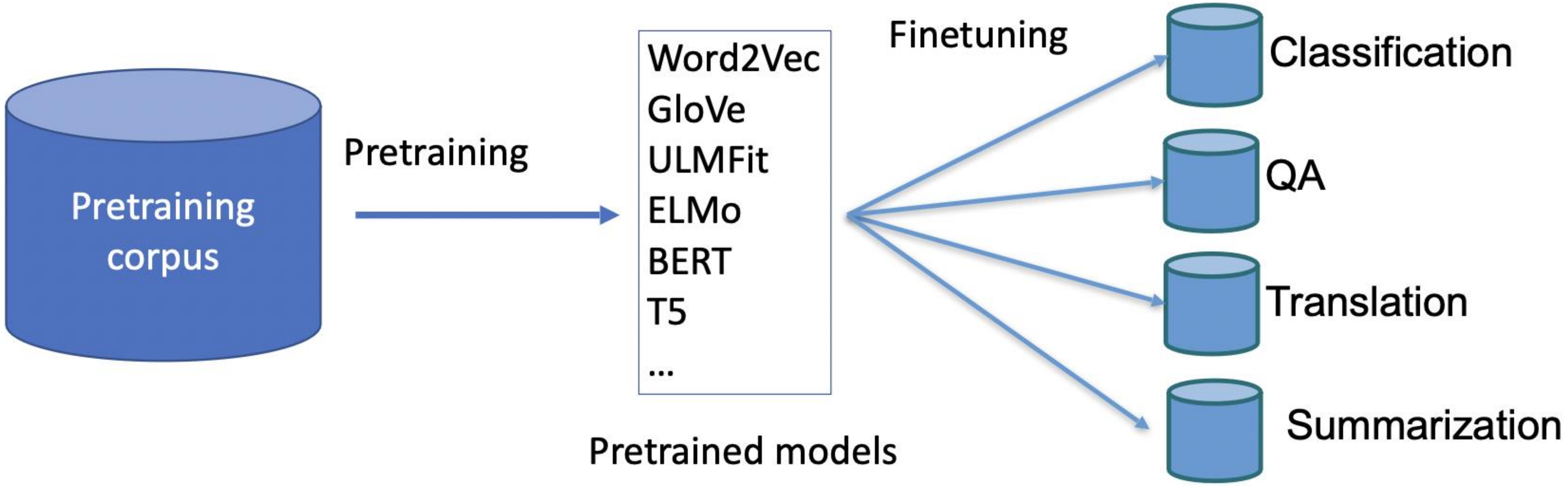
- Learning of a new tasks relies on the previous learned tasks:
  - Learning process can be faster, more accurate and/or need less training data



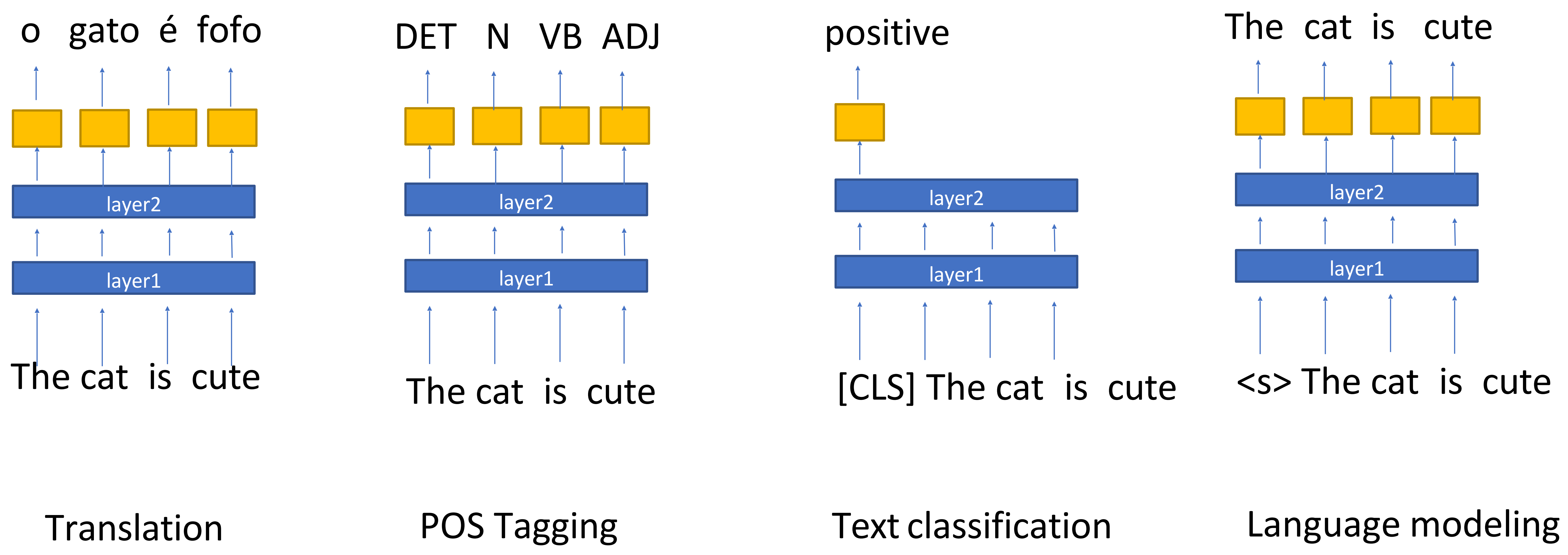
Fine-Tuning for Tasks



Fine-Tuning for Tasks

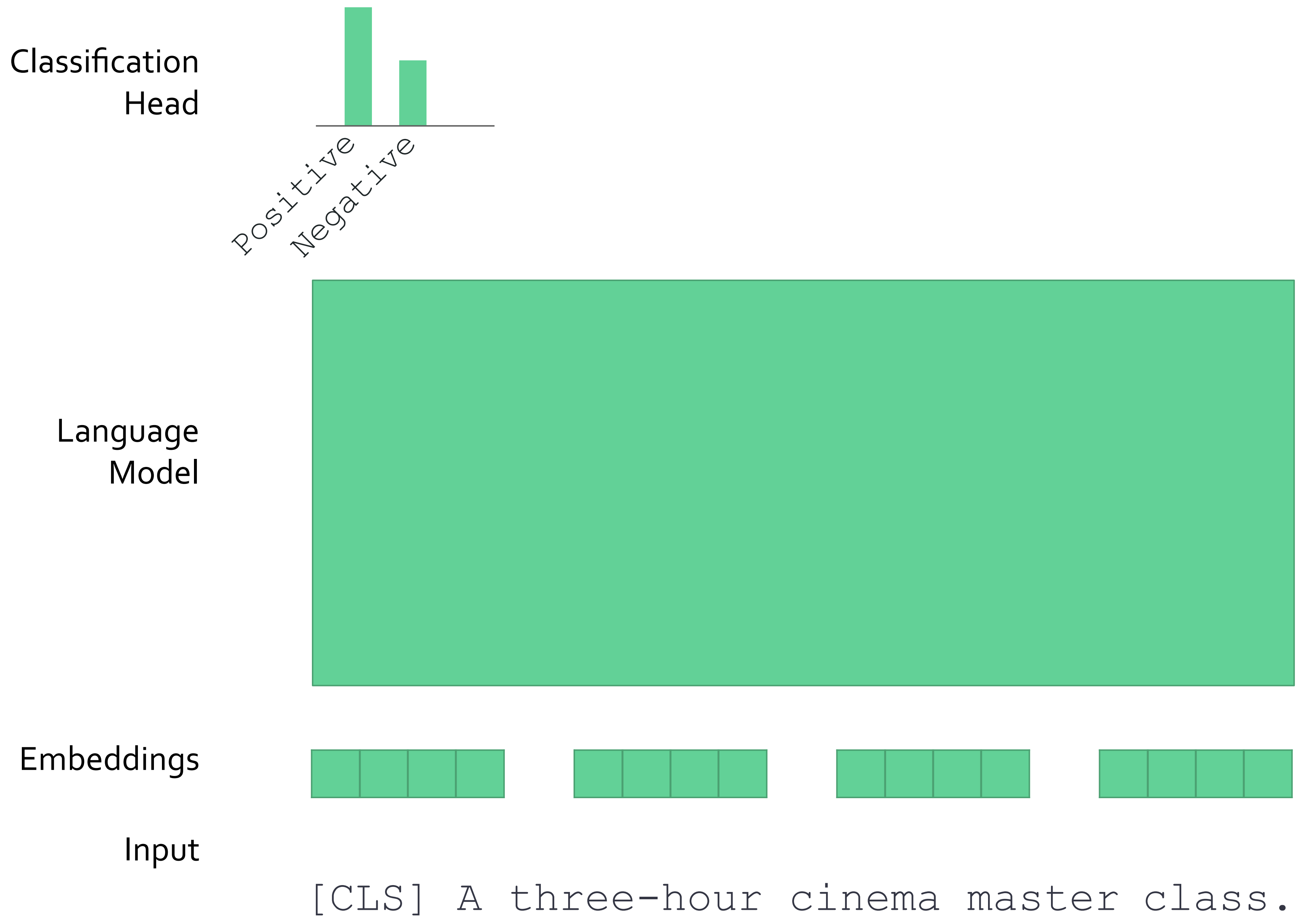


Fine-Tuning for Tasks



Johns Hopkins Whiting School of Engineering

Fine-tuning Pre-trained Models



- Whole model tuning:
  - Run an optimization defined on your task data that updates **all** model parameters
- Head-tuning:
  - Run an optimization defined on your task data that updates the parameters of the model “head”

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

[ACL 2022 Tutorial Beltagy, Cohan, Logan IV, Min and Singh]

Parameter-efficient Fine-tuning

- In fine-tuning we need to updating and storing all the parameters of the LM
  - We would need to store a copy of the LM for each task

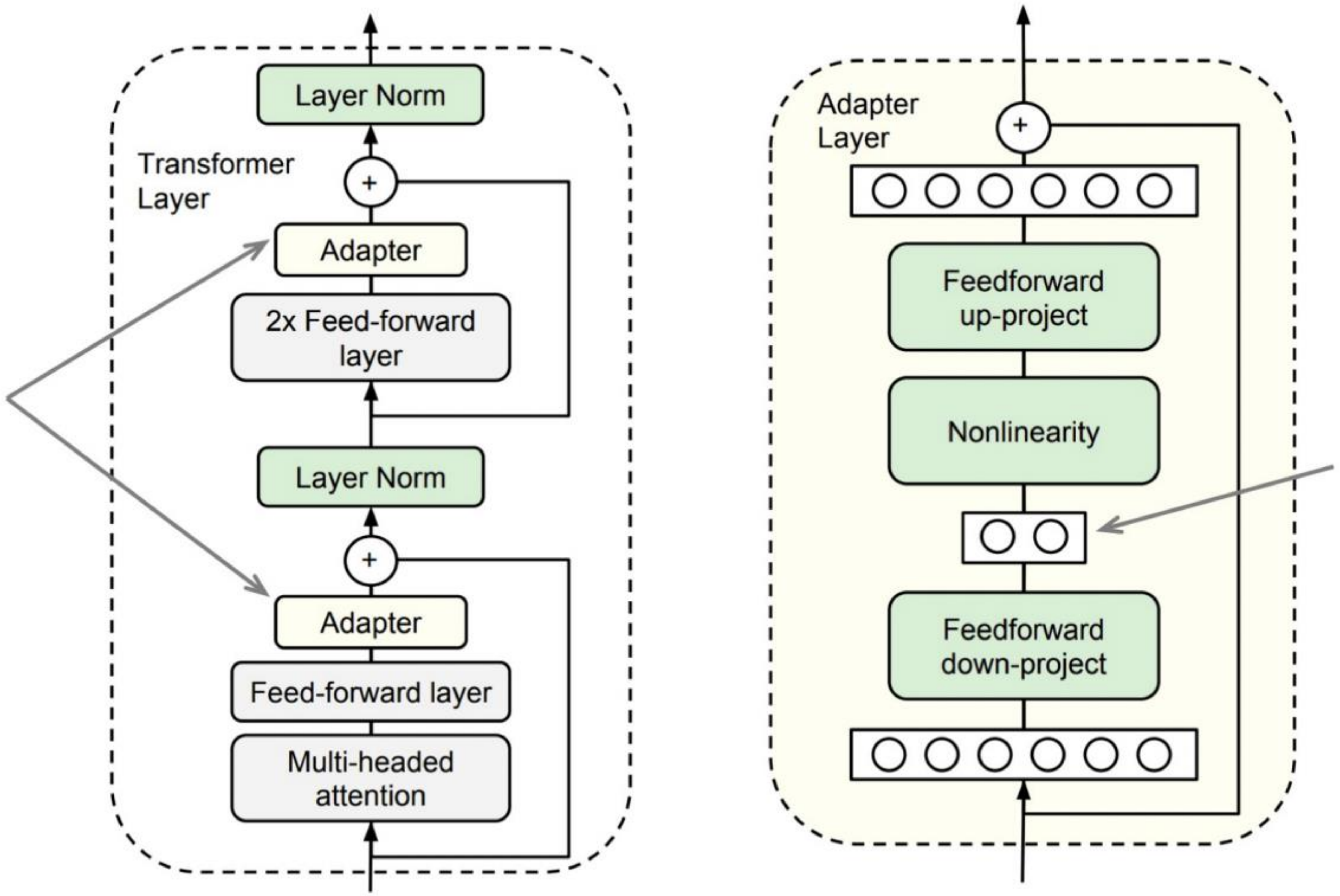
- With large models, storage management becomes difficult
  - E.g., A model of size 170B parameters requires ~340Gb of storage
  - If you fine-tune a separate model for 100 tasks:

    340 * 100 = 34 TB of storage!

Adapters

- **Idea:** train small sub-networks and only tune those.
  - FF projects to a low dimensional space to reduce parameters.
- No need to store a full model for each task, **only the adapter params**.

Only these are trained, everything else is fixed and is the same for all tasks



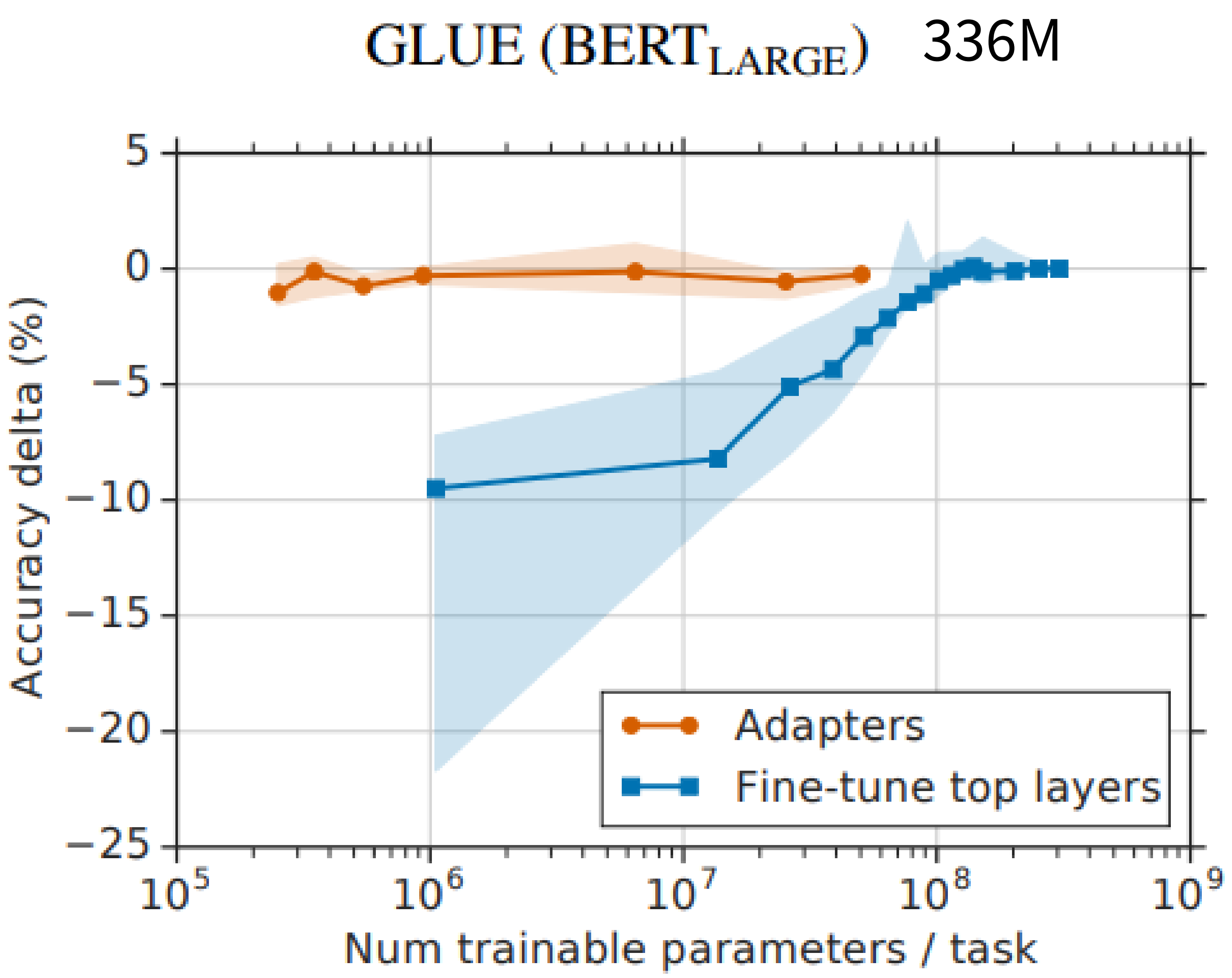
Small hidden size, i.e. an adaptor has only a few parameters (which is good!)

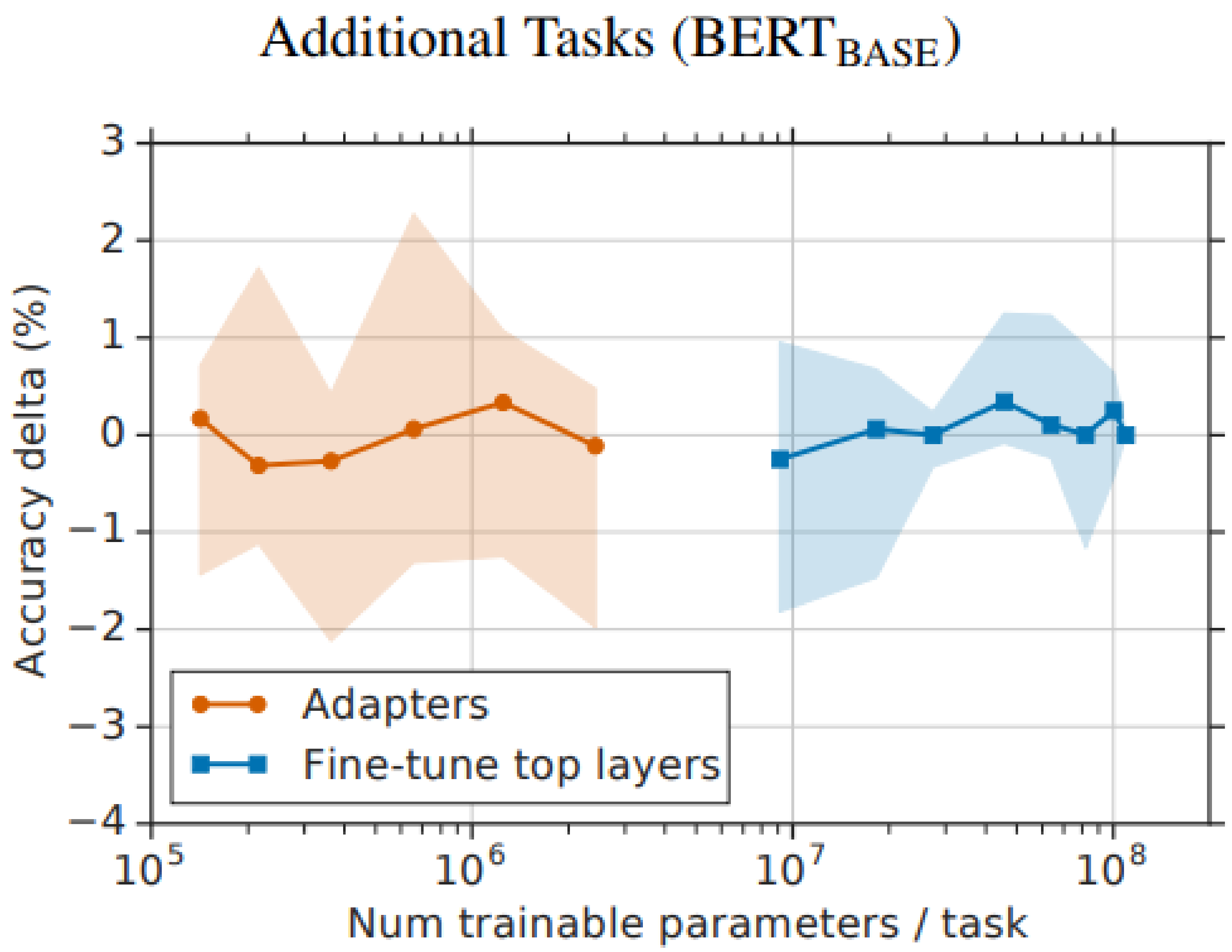
["Parameter-Efficient Transfer Learning for NLP", Houlsby et al., 2019.]

Adapters

IMPORTANT

- **Idea:** train small sub-networks and only tune those.
  - FF projects to a low dimensional space to reduce parameters.
- No need to store a full model for each task, **only the adapter params**.





[“Parameter-Efficient Transfer Learning for NLP”, Houlsby et al., 2019.]

Question

- Is parameter-efficient tuning more (1) computationally efficient; (2) memory-efficient than whole-model tuning?

- It is not faster! You still need to do the entire forward and backward pass.
- It is more memory efficient.
  - You only need to keep the optimizer state for parameters that you are fine-tuning and not all the parameters.

Reparametrization based methods

- Reparametrize the weights of the network using a low-rank transformation. This decreases the trainable parameter count while still allowing the method to work with high-dimensional matrices

LoRA : Low-Rank Adaptation

IMPORTANT

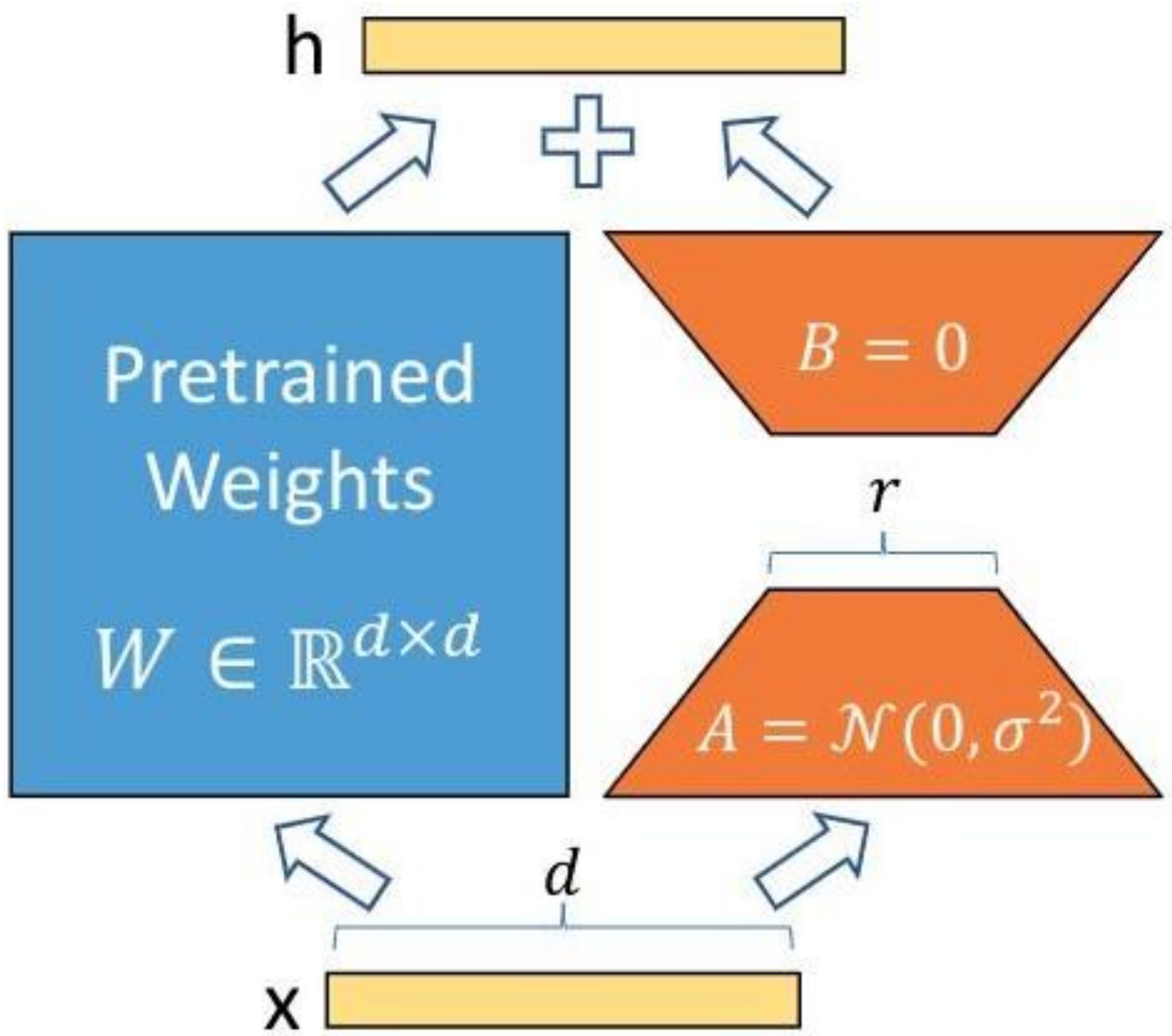
- Hypothesis: the intrinsic rank of the weight matrices in a large language model is low
- Parameter update for a weight matrix is decomposed into a product of two low-rank matrices

$$W \leftarrow W + \Delta W$$

$$\Delta W = BA$$

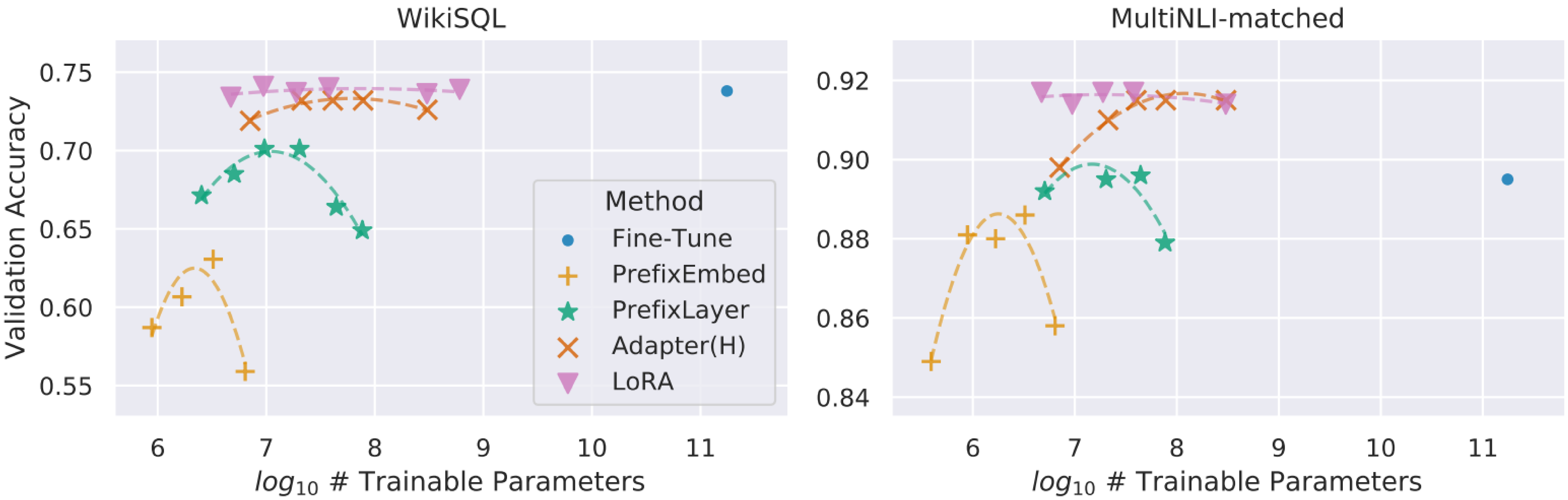
$$B \in \mathbb{R}^{d,r}, A \in \mathbb{R}^{r,k}, \quad r \ll min(k, d)$$

- A is initialized with random Gaussian Initialization, B is initialized to zero



["LoRA: Low-Rank Adaptation of Large Language Models" Hu et al., 2021.]

LoRA



WikiSQL
Validation Accuracy
0.75
0.70
0.65
0.60
0.55
6
7
8
9
10
11
log_{10} # Trainable Parameters
Method
Fine-Tune
PrefixEmbed
PrefixLayer
Adapter(H)
LoRA
MultiNLI-matched
0.92
0.90
0.88
0.86
0.84
6
7
8
9
10
11
log_{10} # Trainable Parameters

Question

- Is parameter-efficient tuning more (1) computationally efficient; (2) memory-efficient than whole-model tuning?

- It is faster! You only need to do the entire forward and backward pass of much less parameters + Caching.
- It is more memory efficient.
  - You only need to keep the optimizer state for parameters that you are fine-tuning and not all the parameters.

The end

Parameters Efficient Fine Tuning

Merci !

Mohamed Abbas KONATE

mohamed-abbas.konate@michelin.com