CS 490: NATURAL LANGUAGE PROCESSING

Dan Goldwasser, Abulhair Saparov

Lecture 24: Efficiency

WORKING WITH LARGE MODELS

- NLP models benefit from increasing scale.
- But they can be very computationally expensive,
  - Especially as we scale them up.
- How much does it cost to perform model inference? Training?
- The forward pass consists of multiple operations of different types:
  - Matrix multiplication
  - Softmax
  - Vector addition
  - Vector-scalar operations
  - Activation functions
  - etc...

COST OF VECTOR OPERATIONS

- Some operations, such as vector operations, are relatively cheap.
- Adding two vectors of dimension d takes d floating-point operations.
- Multiplying a d-dimensional vector with a scalar (or dividing) takes d FLOPs.
- What about memory?
  - If computing $f(x) = x + b$, where b is a learnable vector parameter,
  - There is a total of d parameters we need to store in memory.

MATRIX MULTIPLICATION

- Matrix multiplication is significantly more expensive.
- Consider multiplying two matrices $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$.

$$[AB]_{ij} = \sum_{u=1}^{k} A_{iu} B_{uj}.$$

- We must compute each element in the destination matrix.
  - The destination matrix AB has dimension $n \times m$.
  - Each element is computed using a dot product with length k.
  - The dot product requires k multiplications, followed by k additions.
- Thus the total cost (in terms of time) is: $2 \cdot n \cdot k \cdot m$.
- For memory: Suppose we have $f(X) = AX$ where A is learnable.
  - We need to store $n \cdot k$ parameters in memory.

SOFTMAX

- Transformers perform a softmax operation in every attention layer.
  - Each layer has H softmax operations, where H is the number of attention heads.

$$f(\alpha)_k = \frac{\exp(\alpha_k)}{\sum_{j=1}^{K} \exp(\alpha_j)}.$$

- Suppose α is has dimension d.
- Softmax requires computing d exponential operations,
  - Followed by a sum (d operations),
  - Followed by d division operations.
- Softmax requires no additional memory.

SOFTMAX

- Transformers perform a softmax operation in every attention layer.
  - Each layer has H softmax operations, where H is the number of attention heads.

$$f(\alpha)_k = \frac{\exp(\alpha_k)}{\sum_{j=1}^{K} \exp(\alpha_j)}.$$

- The exponential function is expensive to compute.
- The exponential function is transcendental,
  - Meaning we can not write it down as a finite sequence of addition, subtraction, multiplication, or division operations.
  - It is approximated by computing terms in an infinite series.

ACTIVATION FUNCTIONS

- The cost of an activation function depends on the choice of function.
- Some activation functions require the computation of transcendental functions,
  - E.g., sigmoid, tanh.
  - $GELU(x) = \Phi(x) \cdot x$ *where* Φ *is the CDF for the normal distribution.*
  - $Swish(\mathrm{x}) = \sigma(kx) \cdot x$ where k is a learnable parameter.
- Others are much cheaper:
  - $ReLU(\mathrm{x}) = \mathbb{1}\{x > 0\} \cdot x$
- These functions cost d operations,
- But each operation may be more expensive due to transcendental functions.
- Trivia: GPT-2 used GELU.

ACTIVATION FUNCTIONS

- The cost of an activation function depends on the choice of function.
- Others contain matrix multiplications:
  - $GLU(\mathrm{x}) = \sigma(W_1 x + b_1) \odot (W_2 x + b_2)$ where W_1, W_2, b_1, b_2 are learnable.
  - $SwiGLU(\mathrm{x}) = Swish(W_1 x + b_1) \odot (W_2 x + b_2)$
- Due to the additional matrix products, these functions cost $4d^2$.
- Trivia: Llama-2, Llama-3, Deepseek, OLMO, PaLM use SwiGLU.
- For memory, the simpler activation functions require no additional parameters.
  - But SwiGLU requires $2(d^2 + d)$ parameters.

COST OF FORWARD PASS

- Given the cost of each component, we can now compute the total cost of the forward pass for a given ML model.
- Let's take the decoder-only transformer for example.

Operation	Parameters	FLOPs per Token
Embed	$(n_{\text{vocab}} + n_{\text{ctx}})d_{\text{model}}$	$4d_{\text{model}}$
Attention: QKV	$n_{\text{layer}}d_{\text{model}}3d_{\text{attn}}$	$2n_{\text{layer}}d_{\text{model}}3d_{\text{attn}}$
Attention: Mask	—	$2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}}$
Attention: Project	$n_{\text{layer}}d_{\text{attn}}d_{\text{model}}$	$2n_{\text{layer}}d_{\text{attn}}d_{\text{model}}$
Feedforward	$n_{\text{layer}}2d_{\text{model}}d_{\text{ff}}$	$2n_{\text{layer}}2d_{\text{model}}d_{\text{ff}}$
De-embed	—	$2d_{\text{model}}n_{\text{vocab}}$
Total (Non-Embedding)	$N = 2d_{\text{model}}n_{\text{layer}}(2d_{\text{attn}} + d_{\text{ff}})$	$C_{\text{forward}} = 2N + 2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}}$

Note: These are matrix products

[Adam Casson, Transformer FLOPs, 2023]

COST OF FORWARD PASS

- Total number of non-embedding parameters for decoder-only transformer:

$$N = 2d_{model}n_{layer}(2d_{attn} + d_{ff}).$$

- Total number of FLOPs per token for the forward pass of a decoder-only transformer:

$$C_{forward} = 4d_{model}n_{layer}(2d_{attn} + d_{ff}) + 2n_{layer}n_{ctx}d_{attn},$$
$$= 2N + 2n_{layer}n_{ctx}d_{attn}.$$

- In most modern LLMs, $d_{model} = d_{attn}$, and d_{ff} is set to a multiple of d_{model} ($4d_{model}$ is a common choice).
- Assuming these choices, we can re-write the above:

$$N = 2d_{model}n_{layer}(2d_{model} + 4d_{model}) = 2d_{model}n_{layer}(6d_{model}) = 12d_{model}^{2}n_{layer},$$
$$C_{forward} = 2N + 2n_{layer}n_{ctx}d_{model}.$$

[Adam Casson, Transformer FLOPs, 2023]

COST OF FORWARD PASS

$$N = 12 d_{model}^2 n_{layer},$$

$$C_{forward} = 2N + 2 n_{layer} n_{ctx} d_{model}.$$

- Note that N *depends on the square of* d_{model}, *whereas the term* $2 n_{layer} n_{ctx} d_{model}$ is linear in d_{model}.
- Thus, if d_{model} is large, the $2N$ term will dominate in the forward pass cost.
- Take GPT-3 as an example: $n_{layer} = 96$, $n_{ctx} = 4096$, $d_{model} = 12288$.

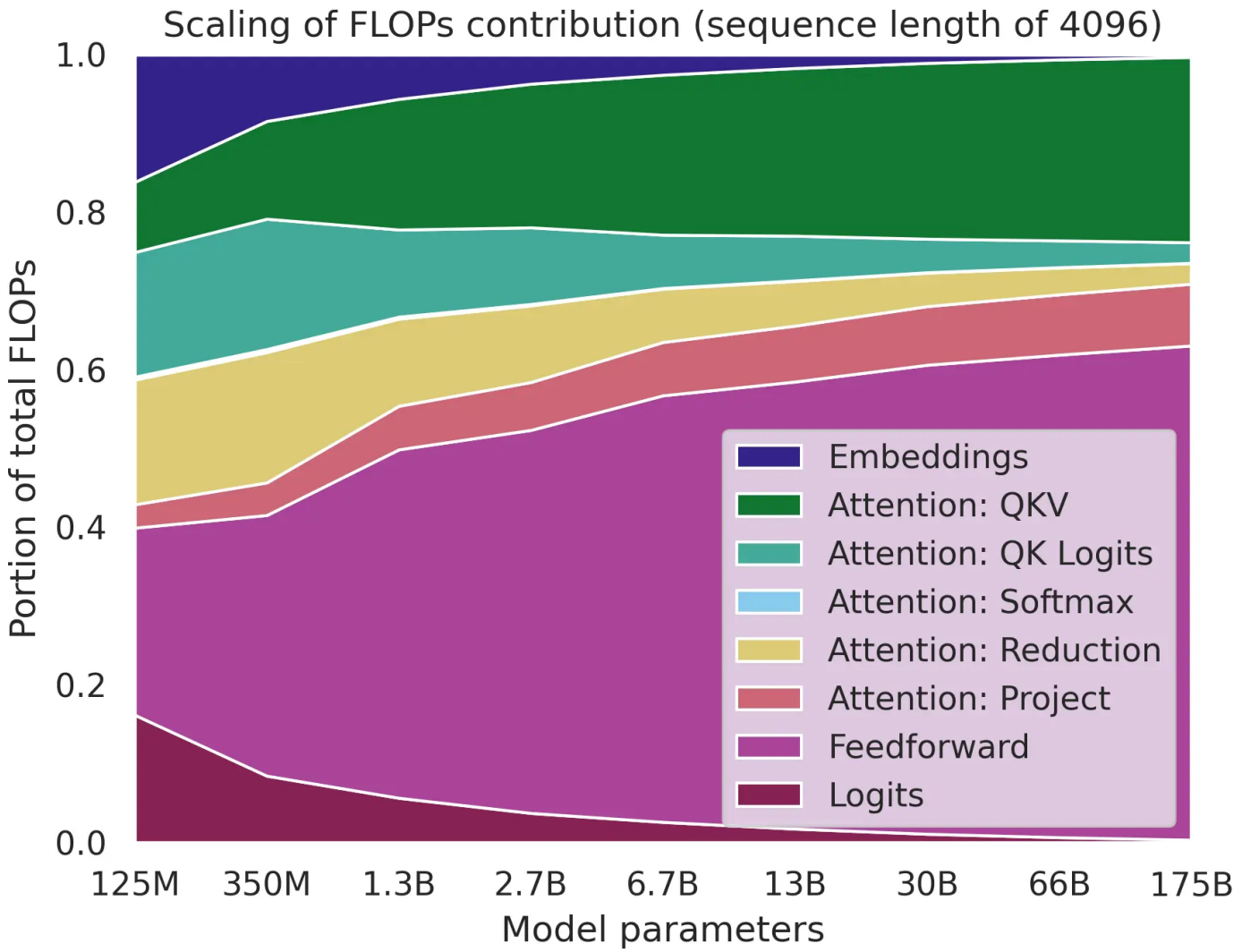
$$N = 12(12288^2)(96) = 174(10^9),$$

$$C_{forward} = 2(174)(10^9) + 2(96)(4096)(12288) = 348(10^9) + 9.7(10^9).$$

- The first term makes up > 97% of the total FLOPs.

[Adam Casson, Transformer FLOPs, 2023]

COST OF FORWARD PASS



[Adam Casson, Transformer FLOPs, 2023]

COST OF TRAINING

- We now know how much it costs to perform a forward pass for a decoder-only transformer.
- But how much does it cost to train?
- Recall that the cost of the transformer is dominated by matrix products.
- Let's examine a single matrix product that happens in the middle of a neural network:

$$Y = XA$$

  - where X is the input activations (from the previous layer),
  - A is the weight matrix,
  - and Y is the output activations (for the next layer).

COST OF TRAINING

$$Y = XA$$

- We want to compute $\frac{\partial L}{\partial A}$, where L is the loss function.
  - By the chain rule:

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial A} = X^{\mathrm{T}}\frac{\partial L}{\partial Y}$$

- But we also need to compute $\frac{\partial L}{\partial X}$, in order to compute the gradients for the previous layer (backpropagation).
  - Again we use the chain rule:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial L}{\partial Y}A^{\mathrm{T}}.$$

- So we need 2 matrix multiplications for each linear layer in the network.

COST OF TRAINING

$$Y = XA$$

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial A} = X^{\mathrm{T}}\frac{\partial L}{\partial Y}$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial L}{\partial Y}A^{\mathrm{T}}.$$

- Notice we also need to have computed X in the above formula for $\frac{\partial L}{\partial X}$.
- This is why we perform a forward pass before each backward pass.
  - The forward pass computes X for all linear layers.
  - The backward pass computes gradients with respect to A for all linear layers
- The forward pass requires 1 matrix multiplication for each linear layer.
- So computing the gradient requires 3 times as many matrix products.

COST OF TRAINING

- For a transformer, since the forward pass costs $2N$ FLOPs per token, where N is the number of parameters,
- That would imply that each step of training costs $6N$ FLOPs per token.
- How much memory do we need?
  - We need memory to store the model parameters: N
  - We need to store the gradient with respect to each parameter: N
  - The optimizer may also require additional memory:
    - Adam (Kingma and Ba, 2014) stores 2 values per parameter: $2N$
  - The activations require additional memory,
    - But this is not a simple function of N.
    - This grows linearly with respect to batch size!

COST OF TRAINING

- Example memory usage for training GPT-2 Small:

GPT2 Small	Predicted	Actual	Diff
N Parameters	124,373,760	124,373,760	0
Model Memory (bytes)	547,826,688	547,826,688	0
Gradients (bytes)	497,495,040	497,495,040	0
Adam buffers (bytes)[1]	994,990,080	994,990,380	300
CuBLAS workspace (bytes)[2]	17,039,360	17,039,360	0
Gaps (bytes)		6,855,368	-
Inputs / Targets (bytes)	196,608	Not visible	-
Others not visble on segment (bytes)		196,620	-
Total (bytes)[3]	2,057,547,776	2,064,403,456	6,855,680

Batch size = *12*
Total = ~*16.5N*

[Edward Rees, Transformer Memory Arithmetic: Understanding all the Bytes in nanoGPT, 2023]

COST OF TRAINING

- What is more expensive for LLMs? Training or inference?
  - What about for deployed models (e.g., ChatGPT)?
  - How long do they need to be deployed before inference cost exceeds training cost?
- GPT-3 was trained on $300(10^9)$ tokens.
  - So the number of training FLOPs was roughly $6(300)(10^9)N = 1.8(10^{12})N$.
  - How many inference forward passes is this equivalent to?
  - Divide the number of training FLOPs by $2N$:
    - $900(10^9)$ forward passes
  - Suppose each prompt generates 10000 tokens.
    - 90 million prompts would match the cost of training.

COST OF TRAINING

Disruption and innovation in search don't come for free. The costs to train an LLM, as we detailed here, are high. More importantly, inference costs far exceed training costs when deploying a model at any reasonable scale. In fact, the costs to inference ChatGPT exceed the training costs on a weekly basis. If ChatGPT-like LLMs are deployed into search, that represents a direct transfer of $30 billion of Google's profit into the hands of the picks and shovels of the computing industry.

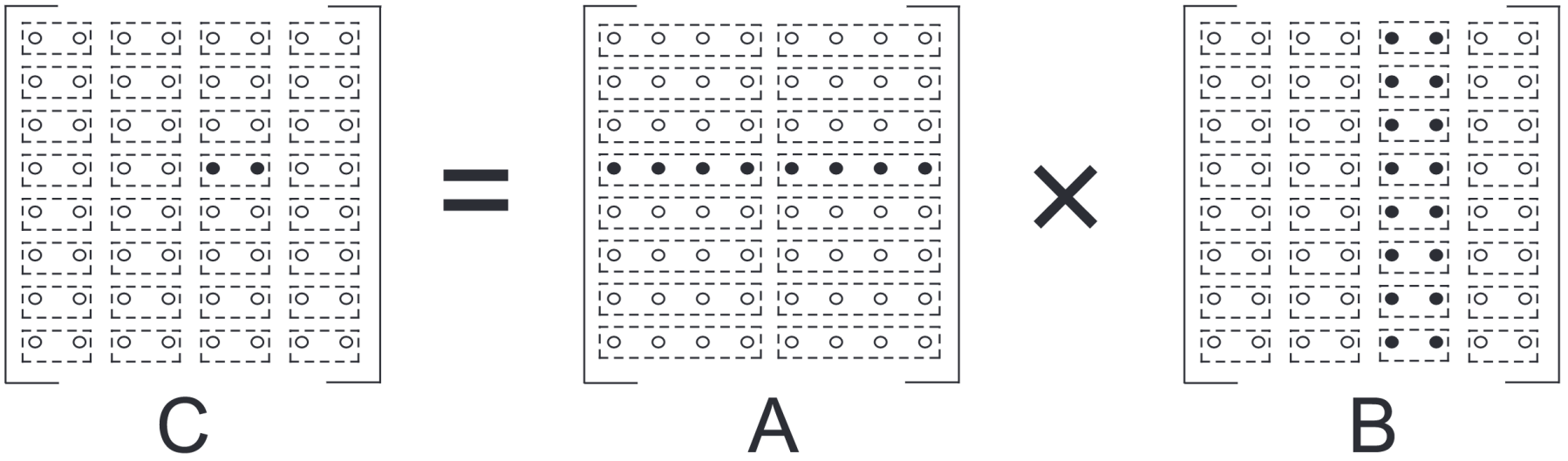
[Patel and Ahmad, The Inference Cost Of Search Disruption – Large Language Model Cost Analysis, 2023]

WORKING WITH LARGE MODELS

- NLP models benefit from increasing scale.
- But they can be very computationally expensive,
  - Especially as we scale them up.
- At very large scales, forward and backward passes require *many* floating-point operations (FLOPs).
- Can we speed up these operations?
  - We know that matrix multiplication is the most expensive operation.
  - Parallelize matrix multiplication!

PARALLELIZATION

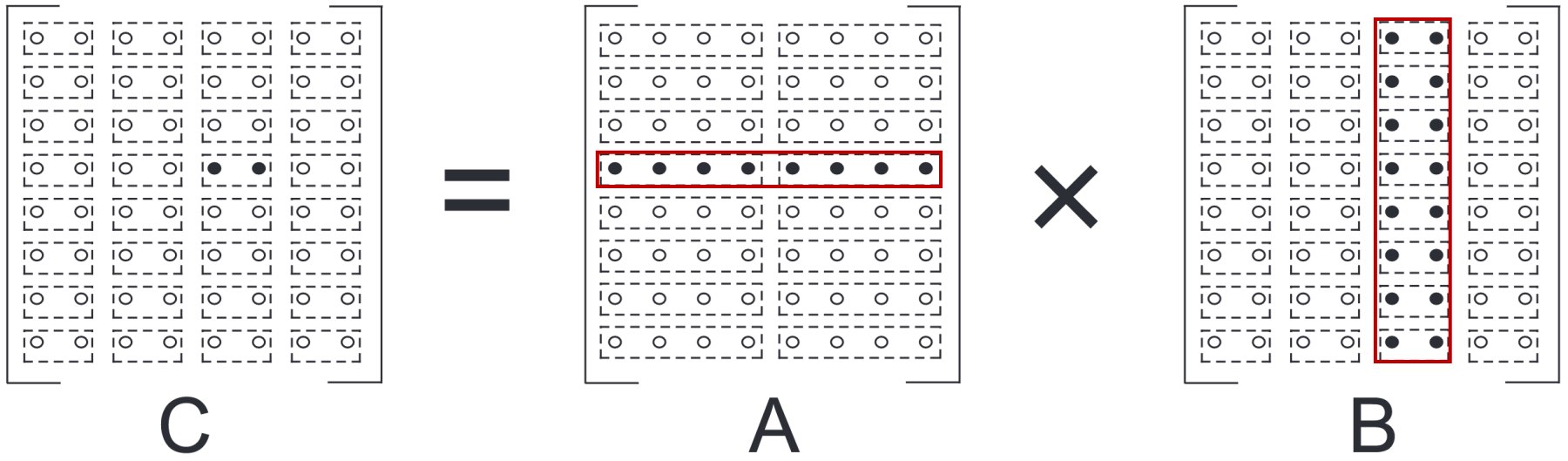
- But how do we parallelize matrix multiplication?
- Suppose we have many available threads.
- And we want to compute: $C = AB$.



[Li et al., GPU Matrix Multiplication, Multi- and Many-Core Technologies: Architectures, Programming, Algorithms and Applications, 2013]

PARALLELIZATION

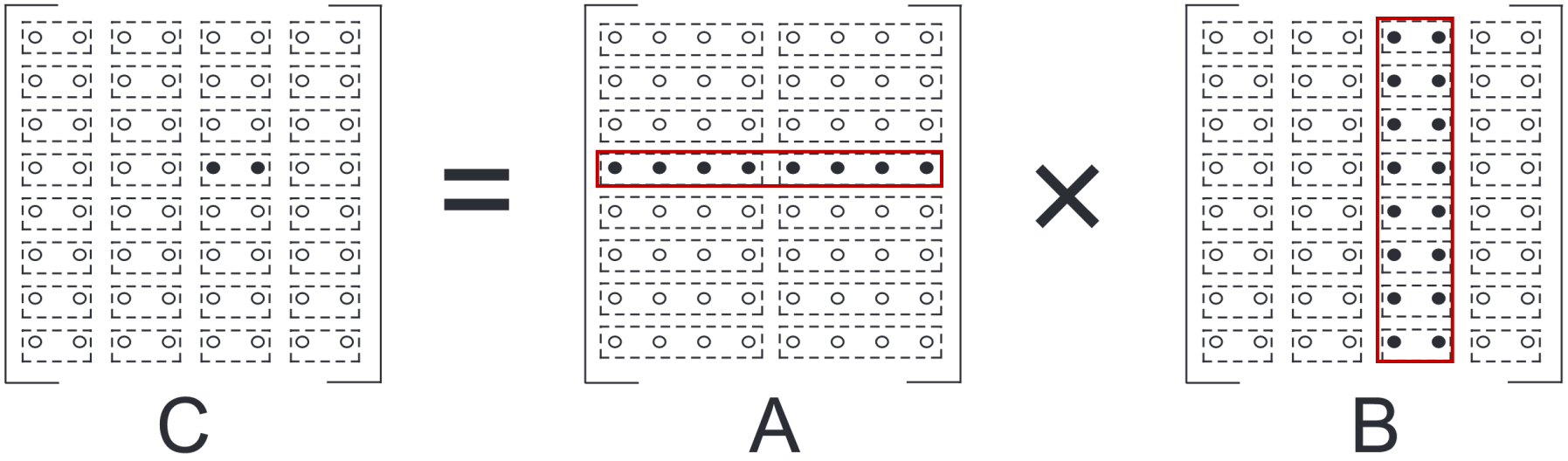
- We can tile the matrices into submatrices.
- For example, we can assign the task of computing the highlighted `1 x 2` submatrix in C to one thread.
  - It multiplies the `1 x 8` submatrix in A with the `8 x 2` submatrix in B.



[Li et al., GPU Matrix Multiplication, Multi- and Many-Core Technologies: Architectures, Programming, Algorithms and Applications, 2013]

PARALLELIZATION

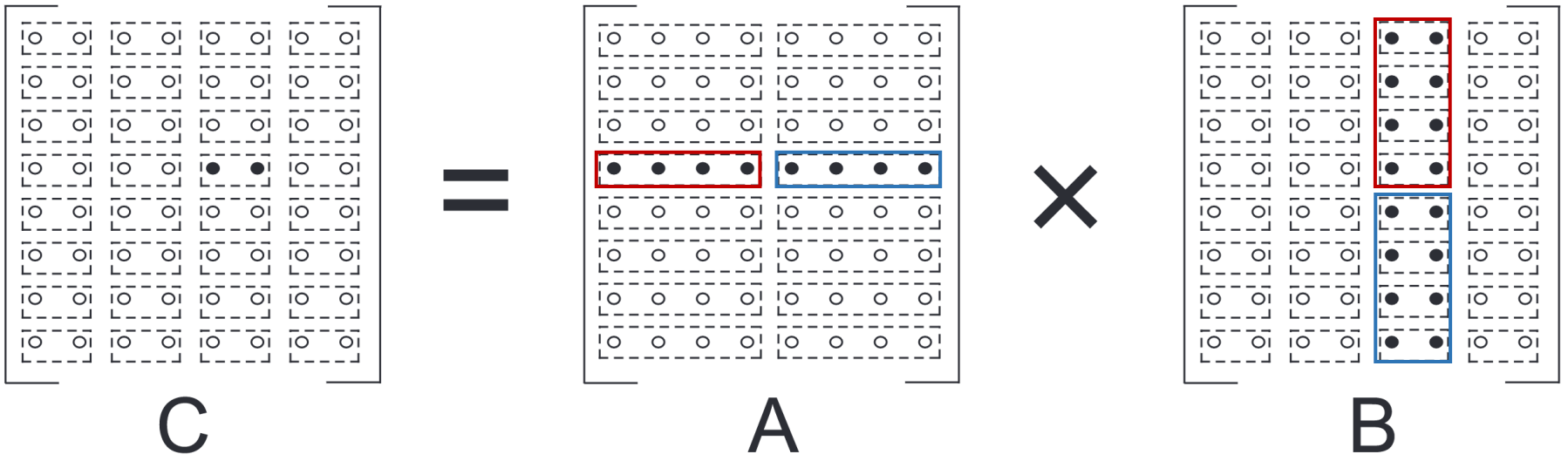
- We can tile the matrices into submatrices.
- With this approach, if $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$, and we have nm threads,
- The matrix multiplication can be completed in $2k$ operations.



[Li et al., GPU Matrix Multiplication, Multi- and Many-Core Technologies: Architectures, Programming, Algorithms and Applications, 2013]

PARALLELIZATION

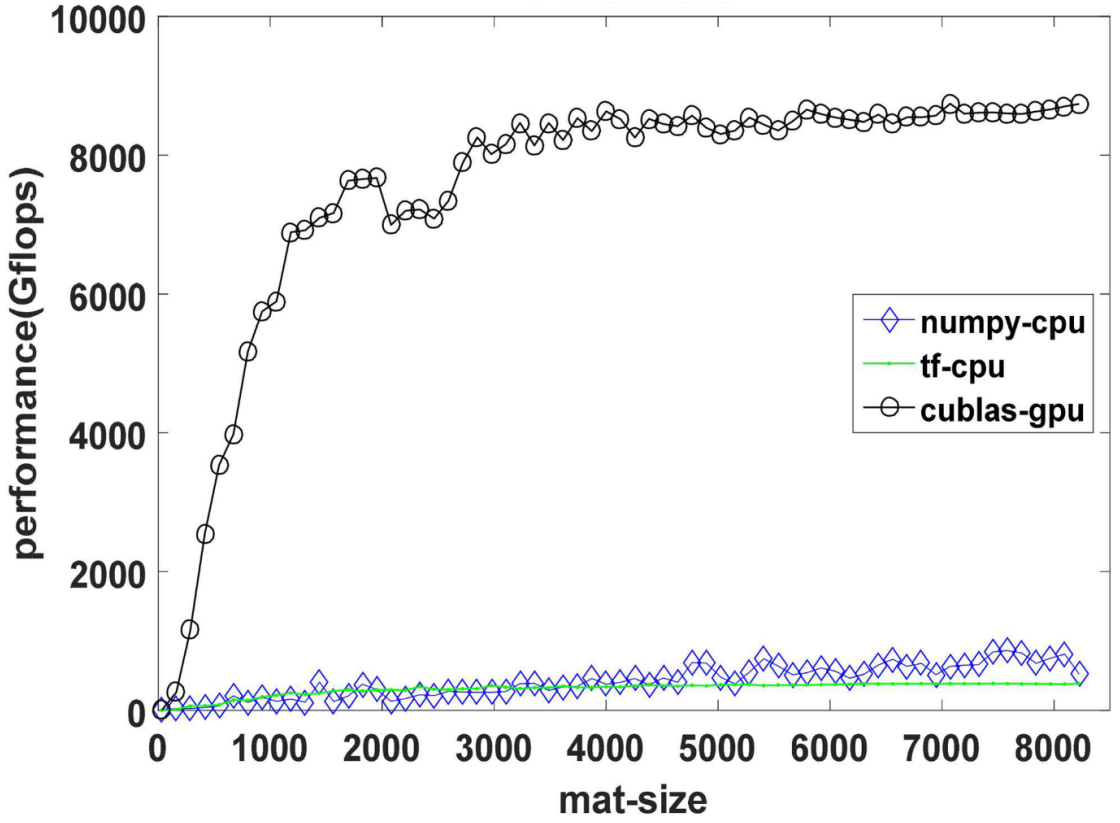
- We can break the matrices down even further:
  - Have one thread multiply the left portion of the highlighted submatrix in A (1 x 4) with the top portion of the highlighted submatrix in B (4 x 2).
  - Have a different thread multiply the right submatrix in A with the bottom portion in B.
  - Then the two resulting 1 x 2 matrices would be summed to produce the result in C.



[Li et al., GPU Matrix Multiplication, Multi- and Many-Core Technologies: Architectures, Programming, Algorithms and Applications, 2013]

GPUS VS CPUS

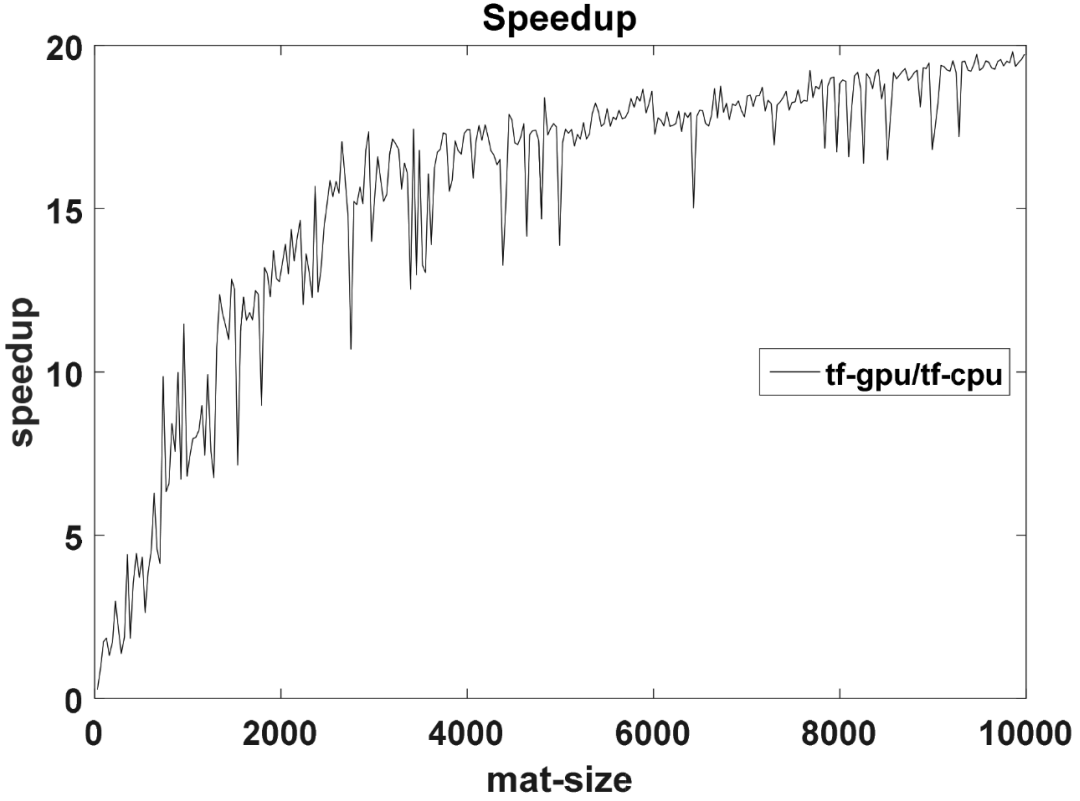
- GPUs are much better than CPUs at parallel processing.



[Huang et al., GPU computing performance analysis on matrix multiplication, 2019]

GPUS VS CPUS

- GPUs are much better than CPUs at parallel processing.



[Huang et al., GPU computing performance analysis on matrix multiplication, 2019]

GPU BEST PRACTICES

- In order to maximize the benefit from using GPUs, we need to keep all our matrices in GPU memory.
- So for ML applications, that requires that all operations required by the model forward/backward pass are implemented on the GPU.
- It is (relatively) expensive to transfer data to/from GPU memory and system memory.

WORKING WITH LARGE MODELS

- NLP models benefit from increasing scale.
- GPT-3, for example, has ~175 billion parameters.
- Each parameter and activation is stored as a 16-bit floating point number.
- Thus, you need 350 GB of VRAM just to load the model parameters.
  - You need more memory to store (batched) activations and cache for inference.
  - You need a lot more to do any training or fine-tuning.
- The H200 GPU has 192 GB of VRAM.
  - Consumer GPUs have 16 or 24 GB.
- To work with models that don't fit in one GPU's VRAM, we need to find ways to distribute inference/training across multiple GPUs.

PARALLELISM

- If a model is too large to fit in the memory of 1 GPU,
- We can split it into smaller pieces across many GPUs.
  - We can divide large matrices into submatrices.
  - Give each submatrix to a node.
  - Called tensor parallelism.
  - We won't go into detail here. (consider taking other courses to learn more!)
- One simple approach: data parallelism.
  - If we have a large batch containing multiple examples,
  - Split it into "minibatches".
  - Each minibatch goes to a node.

WHAT IF WE DON'T HAVE 1000 GPUS?

- We don't all have access to massive clusters with thousands of GPUs.
- How do we train or run inference on models that are too large for 1 or 4 GPUs?
- We can use approximation to reduce the memory footprint of the model.
- Approximation may lead to a cost in accuracy.
- Suppose we have a model that is small enough to fit in our memory for inference, but too large for training.

  (recall the memory cost of training is significantly larger than that for inference)

APPROXIMATING MATRIX PRODUCTS

- The weight matrices of the linear layers are the largest contributors to a model's memory footprint.
- During training, in a linear layer, we compute the forward pass:

$$f(X) = XW^T + b.$$

- In the backward pass, we compute the gradient of the loss with respect to the parameters W and b.
- Then we update the values of W and b according to the step size γ.

$$W_{new} = W - \gamma \frac{\partial L}{\partial W}.$$

APPROXIMATING MATRIX PRODUCTS

- We can gather all the gradient updates into a single ΔW term:

$$W_{new} = W + \Delta W.$$

- This way, during training, we keep W unchanged and only keep track of ΔW.
- So the forward pass becomes:

$$f(X) = X(W + \Delta W)^T + b.$$

- Let's try to approximate ΔW using a product of smaller matrices:

$$\Delta W = AB,$$

- Where ΔW has dimension $d \times d$, A has dimension $d \times r$, and B has dimension $r \times d$, where r is much smaller than d.

APPROXIMATING MATRIX PRODUCTS

- Thus our forward pass is now:

$$f(X) = X(W + AB)^T + b$$

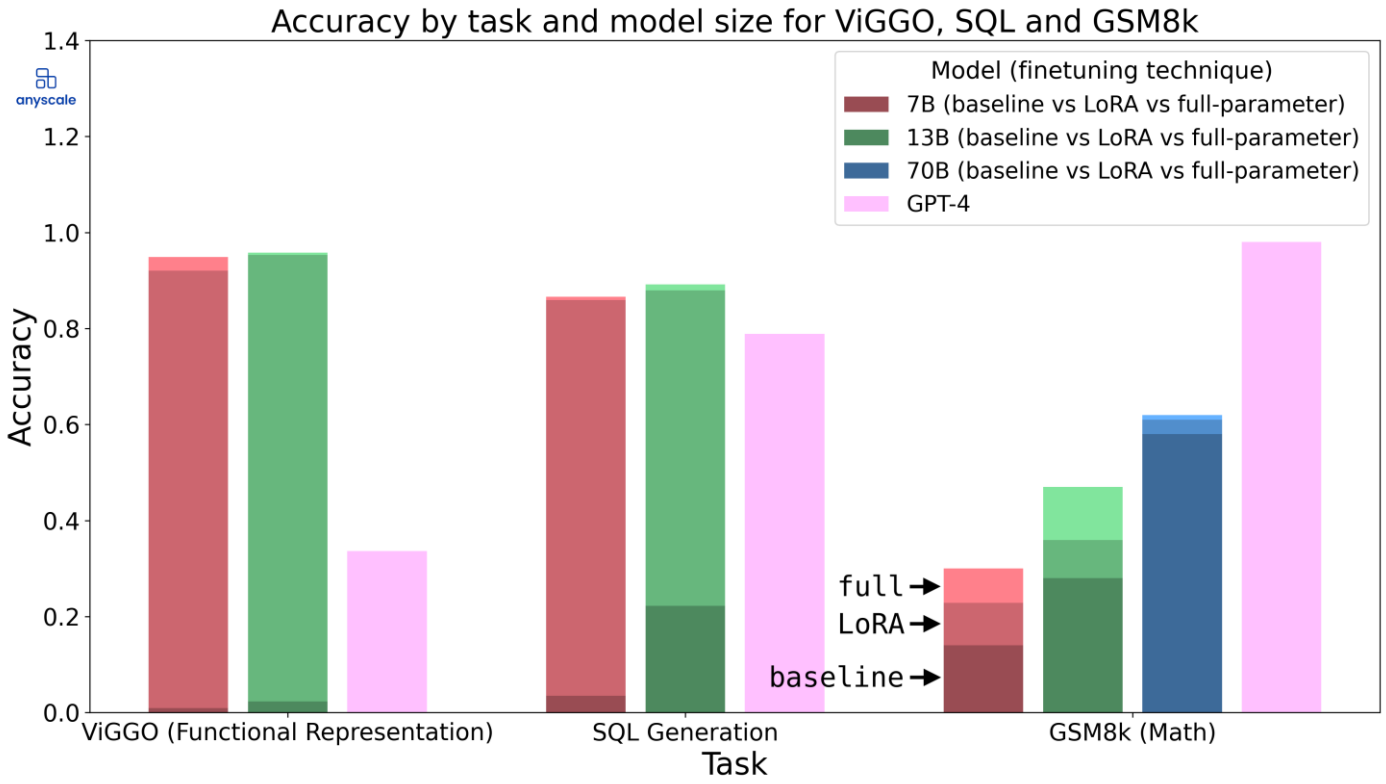
- Where W is a constant and only A and B are learnable parameters.
- So how many training parameters do we have?
  - Previously, we had $d^2 + d$ per linear layer.
  - Now, we have $2dr + d$, which can be much smaller when r is small.

LORA

- This approach is called Low-Rank Adaptation or LoRA (Hu and Shen et al., 2021).
- It works well if the changes to the weight matrix have low rank, which is often true in fine-tuning.
  - But this is not true in pretraining, where the learned weight matrices have high rank.
  - Thus LoRA is only used for fine-tuning.

LORA

- Fine-tuning vs LoRA experiments on Llama-2: ($r = 8$)



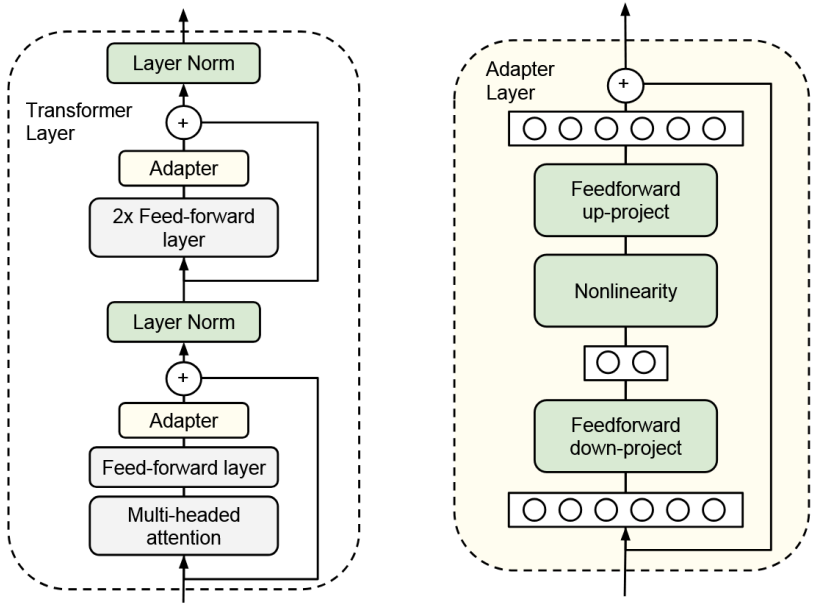
[Niederfahrenhorst et al., Fine-Tuning LLMs: LoRA or Full-Parameter? An in-depth Analysis with Llama 2, 2023]

PEFT

- LoRA is an example of parameter-efficient fine-tuning (PEFT).
- There are many other PEFT methods, and research into new methods is ongoing.
- One simple PEFT approach is to freeze all layers of the model except for the last layer.
  - This was a typical approach for fine-tuning BERT.
  - A simple extension is to fine-tune the last k layers.

PEFT

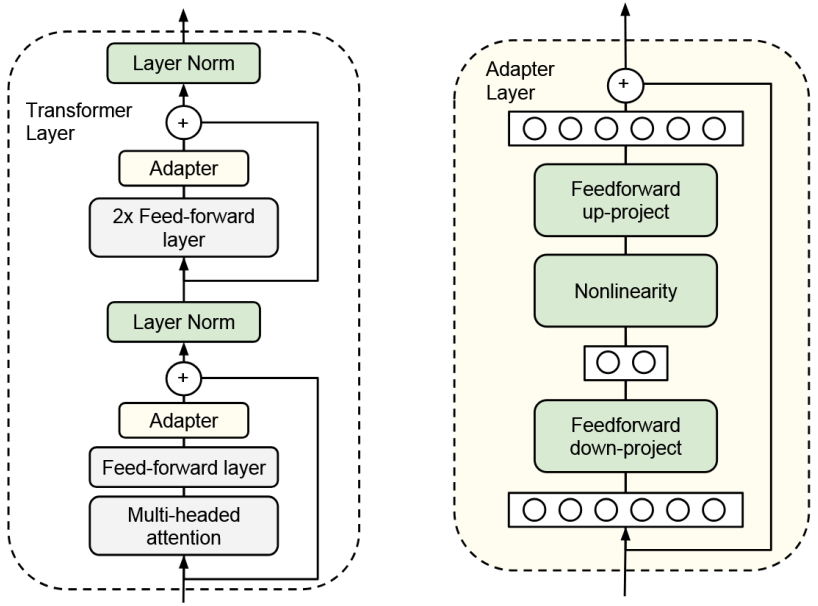
- LoRA is an example of parameter-efficient fine-tuning (PEFT).
- There are many other PEFT methods, and research into new methods is ongoing.
- Another class of PEFT methods are called adapter methods.



[Houlsby et al., 2019]

ADAPTER METHODS

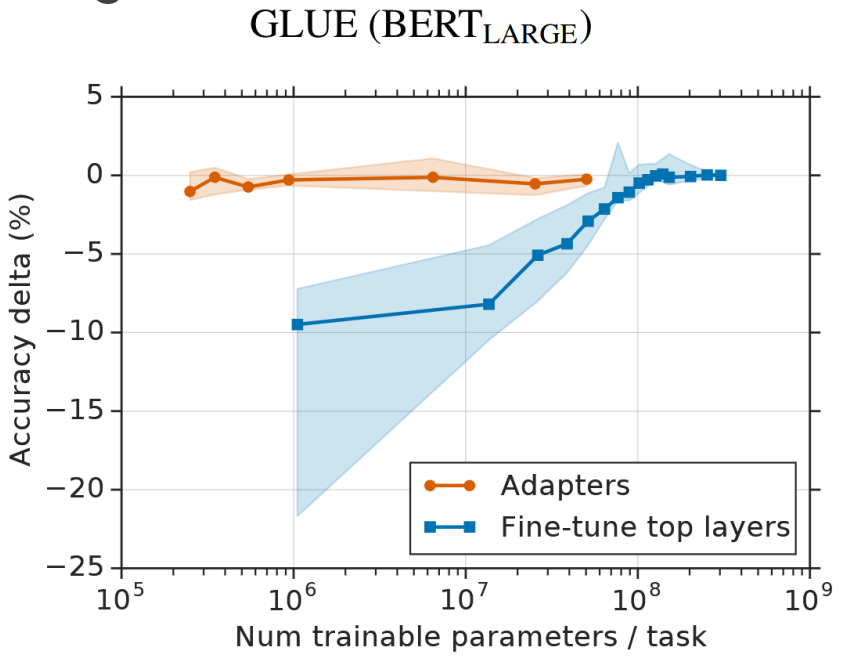
- The idea is to add a small network inside the attention and FF layers.
  - Called the “adapter.”
- Keep all model parameters fixed except for those in the adapter, which is much smaller than the original model.



[Houlsby et al., 2019]

ADAPTER METHODS

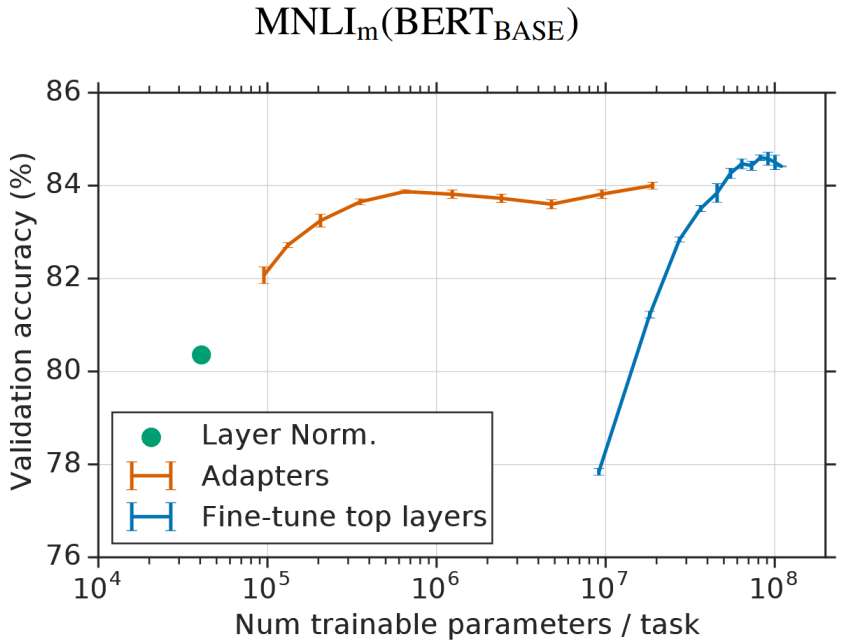
- The idea is to add a small network inside the attention and FF layers.
  - Called the “adapter.”
- Keep all model parameters fixed except for those in the adapter, which is much smaller than the original model.



[Houlsby et al., 2019]

ADAPTER METHODS

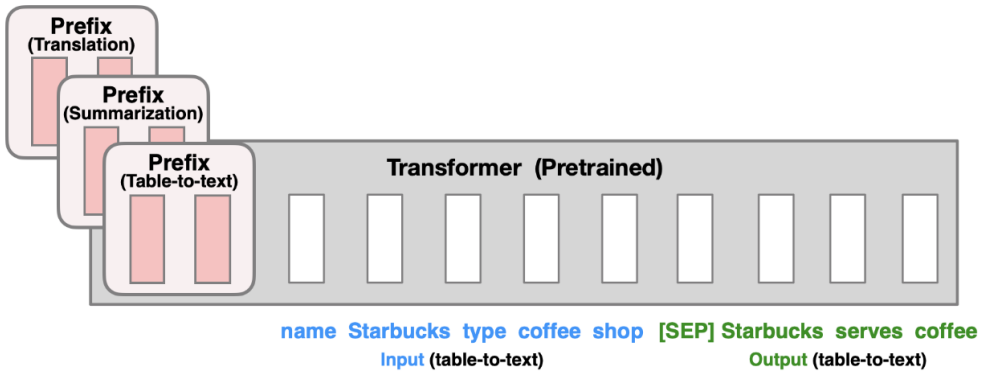
- The idea is to add a small network inside the attention and FF layers.
  - Called the “adapter.”
- Keep all model parameters fixed except for those in the adapter, which is much smaller than the original model.



[Houlsby et al., 2019]

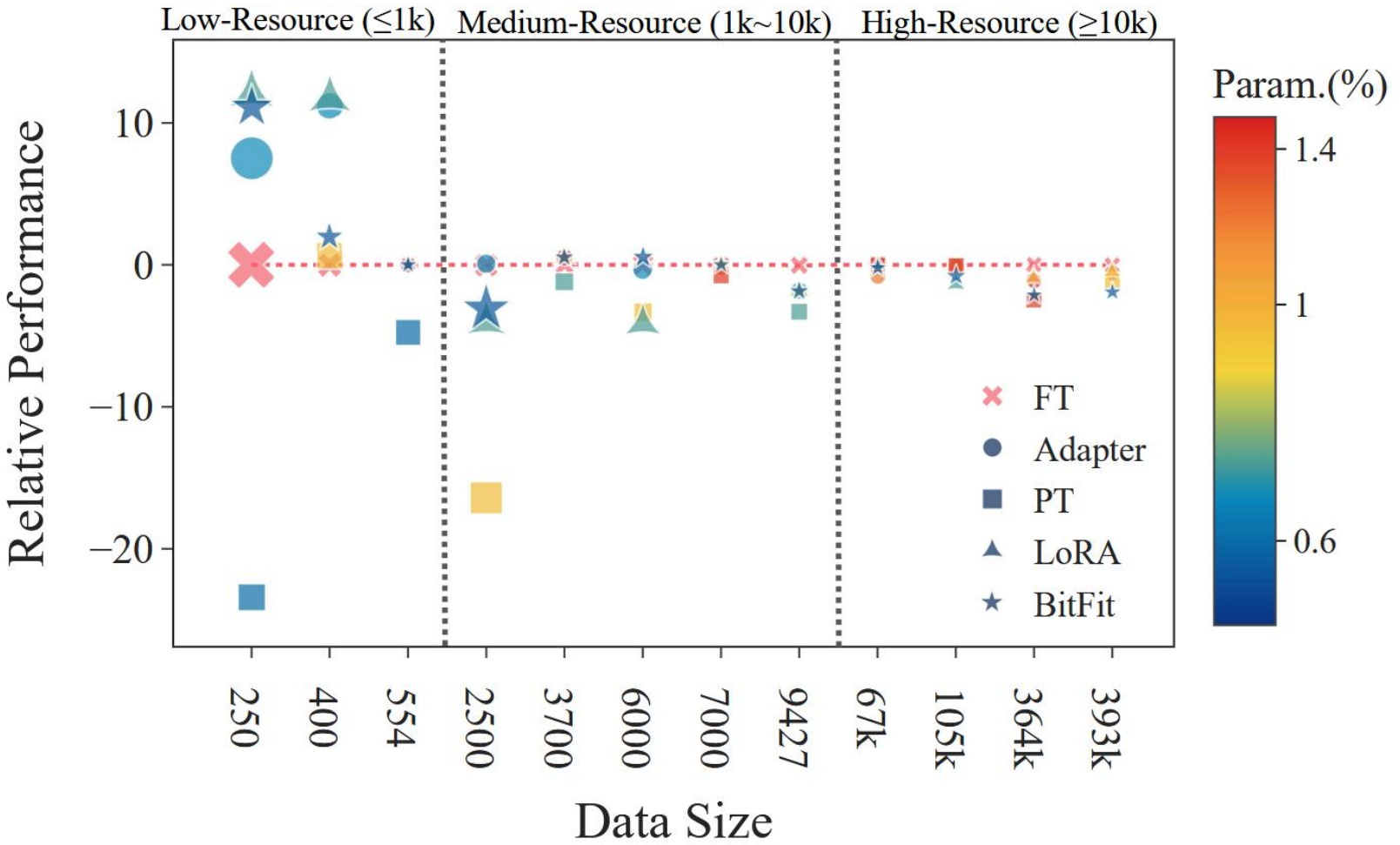
PREFIX TUNING

- Adapter methods and LoRA are similar in that they freeze the original model's parameters
  - And instead added a small number of trainable parameters.
- Prefix tuning is another PEFT method where new tokens are added to the beginning of the input prompt.
  - Unlike regular text tokens, these added tokens are continuous.
  - Their embeddings are the only trainable parameters in the model.



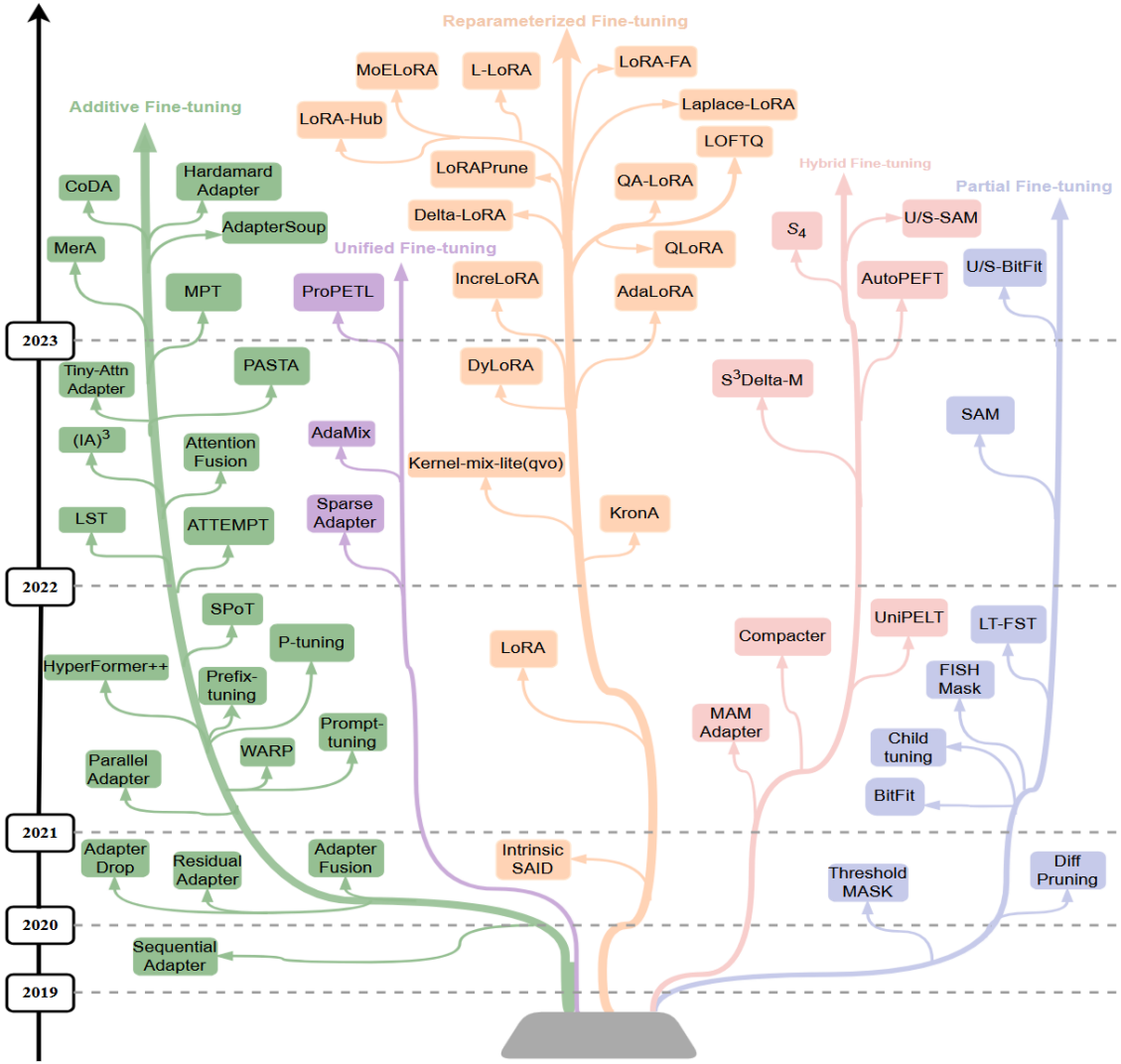
[Li and Liang, 2021]

PEFT



[Chen et al., 2022]

PEFT



[Xu et al., 2023]

NEXT TIME

- What if the original model is too large to fit in memory even for inference alone?
- Can we make the model smaller?
  - Quantization: Reduce the precision of the floating-point numbers in the model.
    - How can we reduce the precision without adversely affecting the model's accuracy?
    - It's not so simple, especially with very low precision.
  - Distillation: Use a larger model to train a small model.

QUESTIONS?