Parameter-Efficient Fine-Tuning (PEFT)

AI with Deep Learning
EE4016

Prof. Lai-Man Po
Department of Electrical Engineering
City University of Hong Kong

https://medium.com/@lmpo/parameter-efficient-fine-tuning-of-large-language-models-4ed51860e1da

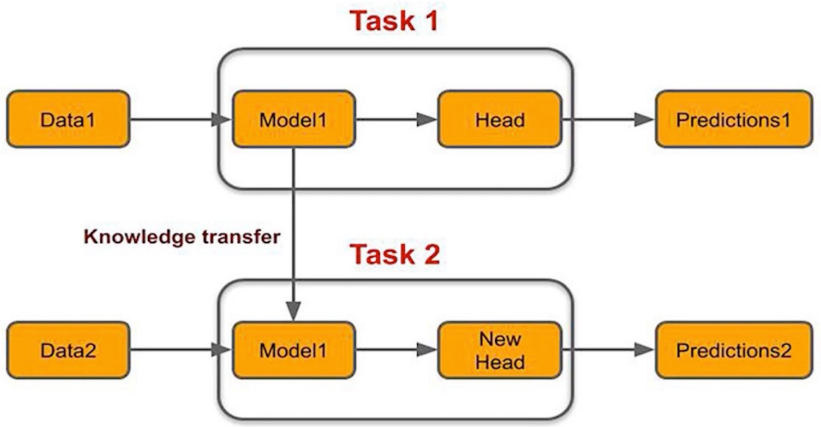
Content

- Review of Finetuning Approaches
- Parameter Efficient Finetuning (PEFT)
  - **Adapter Layer**
  - **LoRA (Low Rank Adaptation)**
  - **QLoRA (Quantized LoRA)**

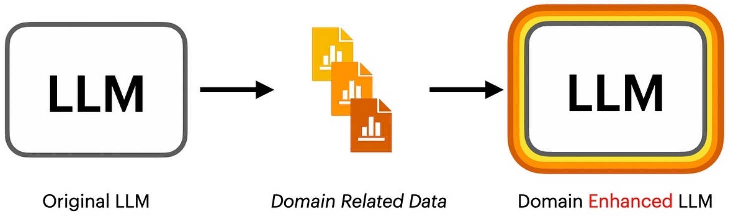
Transfer Learning and Finetuning

- **Transfer learning and fine-tuning are two key techniques in machine learning** that leverage **pre-trained models** to improve performance on new tasks.
- While **they are often used interchangeably**, they have distinct methodologies and applications.
- This approach is based on the idea that a model trained on one task can be adapted to perform well on another related task.

Transfer Learning



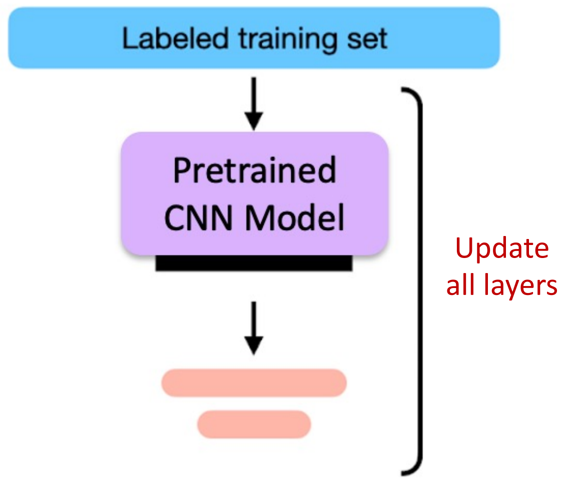
Fine-Tune LLM



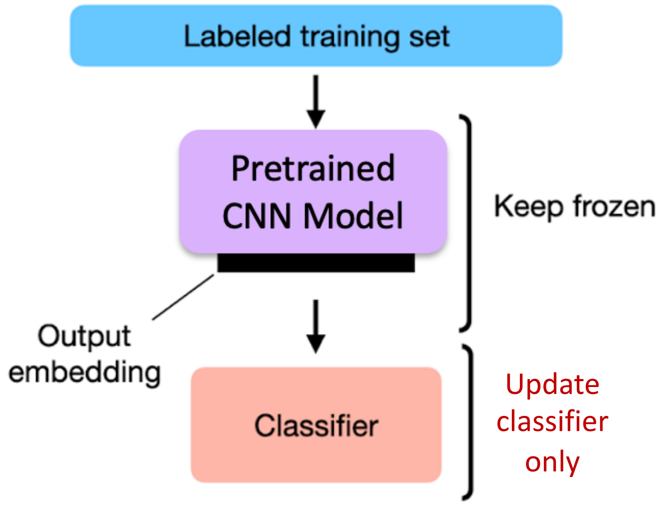
Finetuning of Convolutional Neural Networks (CNNs)

- In computer vision application with CNNs, finetuning is often applied to pre-trained models like ResNet and EfficientNet, which were initially trained using **supervised learning** on large, labeled datasets.
- There are 3 popular finetuning approaches:

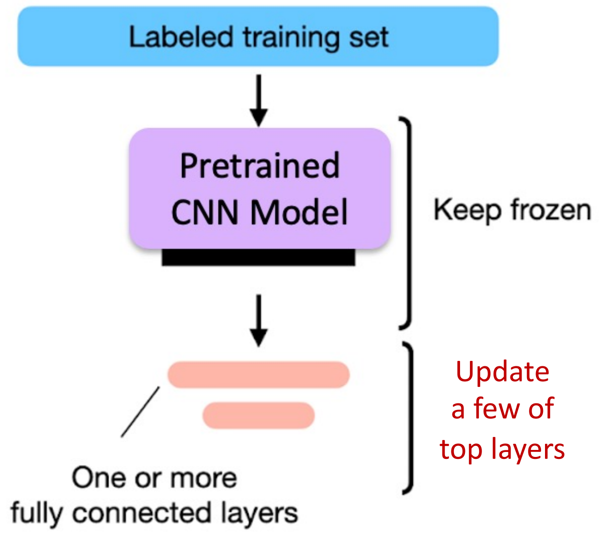
(1) Full Finetuning



(2) Feature-based Approach



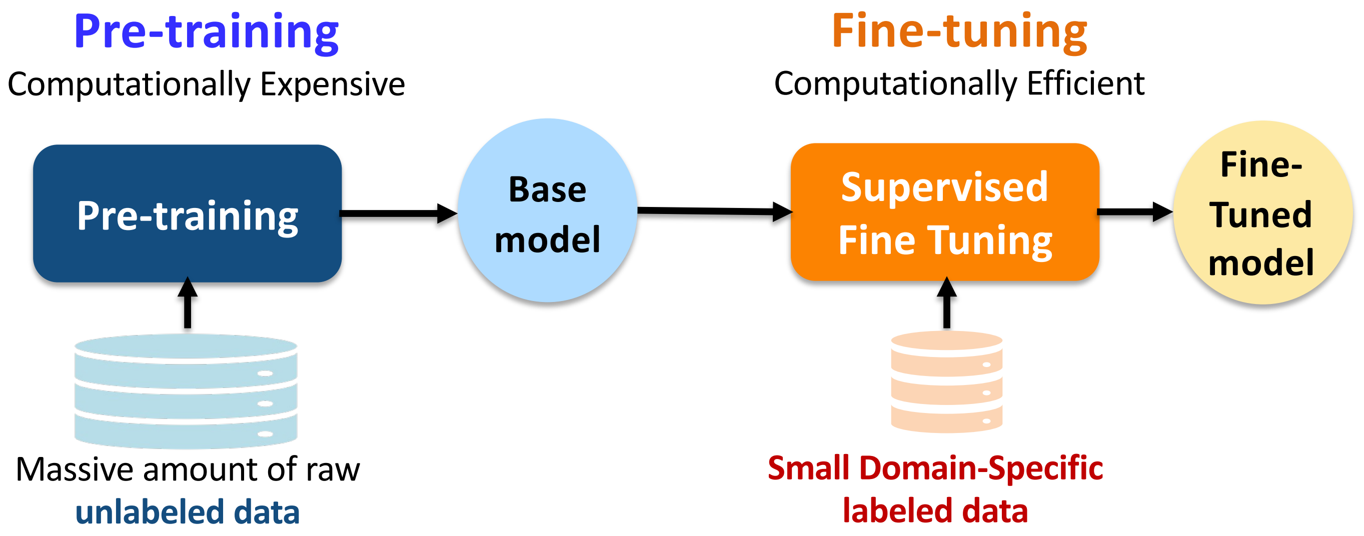
(3) Top-Layer Finetuning



Labeled training set
Pretrained CNN Model
Keep frozen
Update a few of top layers
One or more fully connected layers

Fine-tuning of Large Language Models

- In NLP, **pre-trained LLMs** like BERT and GPT-3 are initially trained on large unlabeled datasets via **Self-Supervised Learning (SSL)**.
- For specific tasks like chatbots or summarization, these models are **fine-tuned** on **smaller domain-specific labeled datasets**, improving performance while being **more computationally efficient** than training from scratch.

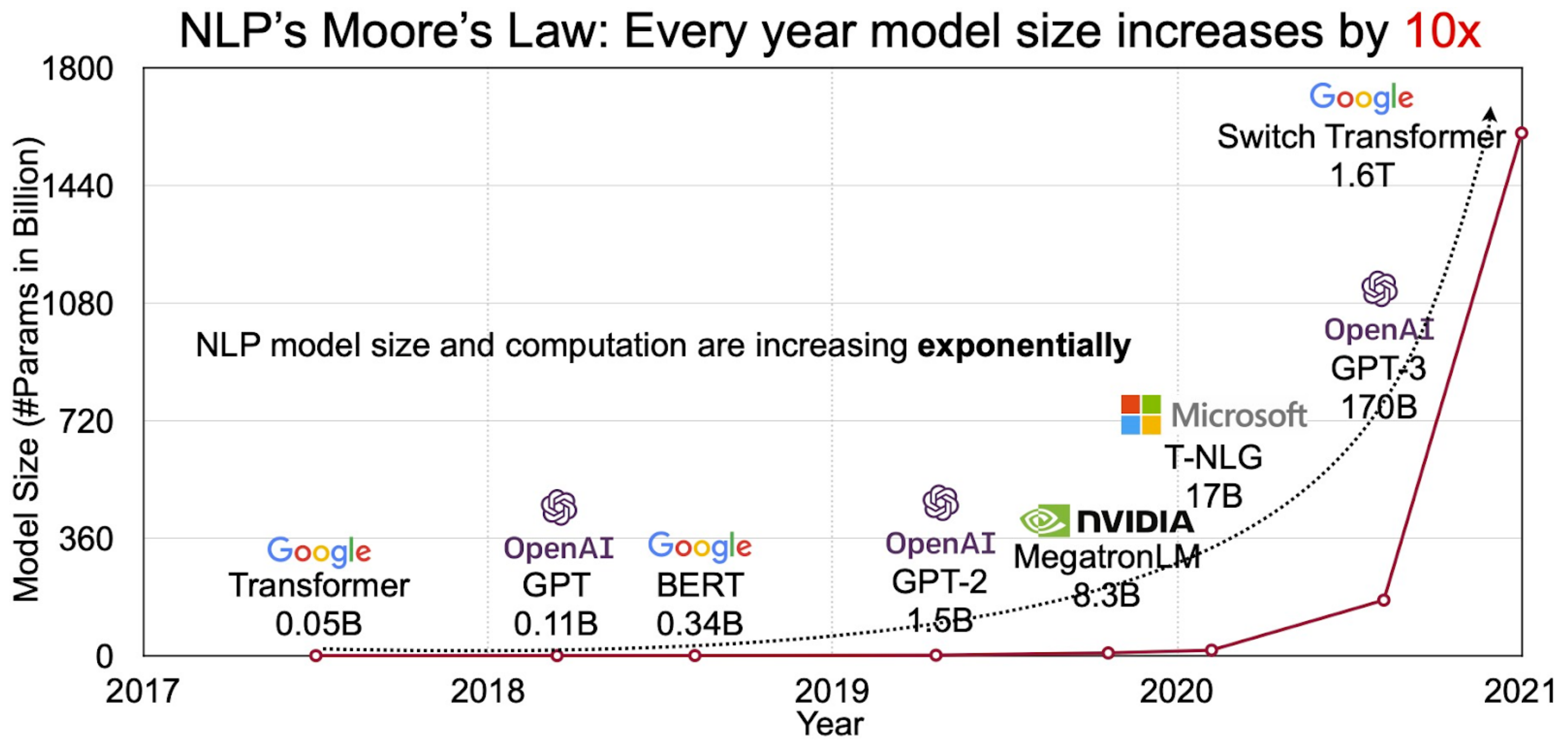

Pre-training
Computationally Expensive
Pre-training
Massive amount of raw unlabeled data
Base model
Fine-tuning
Computationally Efficient
Supervised Fine Tuning
Small Domain-Specific labeled data
Fine-Tuned model

LLM Finetuning: From General to Specific

- In the realm of NLP, finetuning of LLMs is a crucial step in transforming general-purpose pretrained models into specialized models tailored to meet the unique demands of specific applications.
- **This process effectively bridges the gap between the generic, pretrained models and the nuanced requirements of a particular task or domain.**
  - For example, finetuning a pretrained GPT-3 model on a dataset of medical reports and patient notes enables it to adapt to complex medical terminology and jargon, significantly enhancing its performance in generating accurate patient reports.
  - This targeted finetuning unlocks the full potential of LLMs in specialized applications.

Parameter-Efficient Finetuning (PEFT)

LLMs are Becoming Very Large Indeed



The size of LLMs has been rapidly increasing, with models like GPT-3 having 175 billion parameters, and recent models like Google's PaLM surpassing the trillion-parameter mark, enabling more capable but also **more resource-intensive models**.

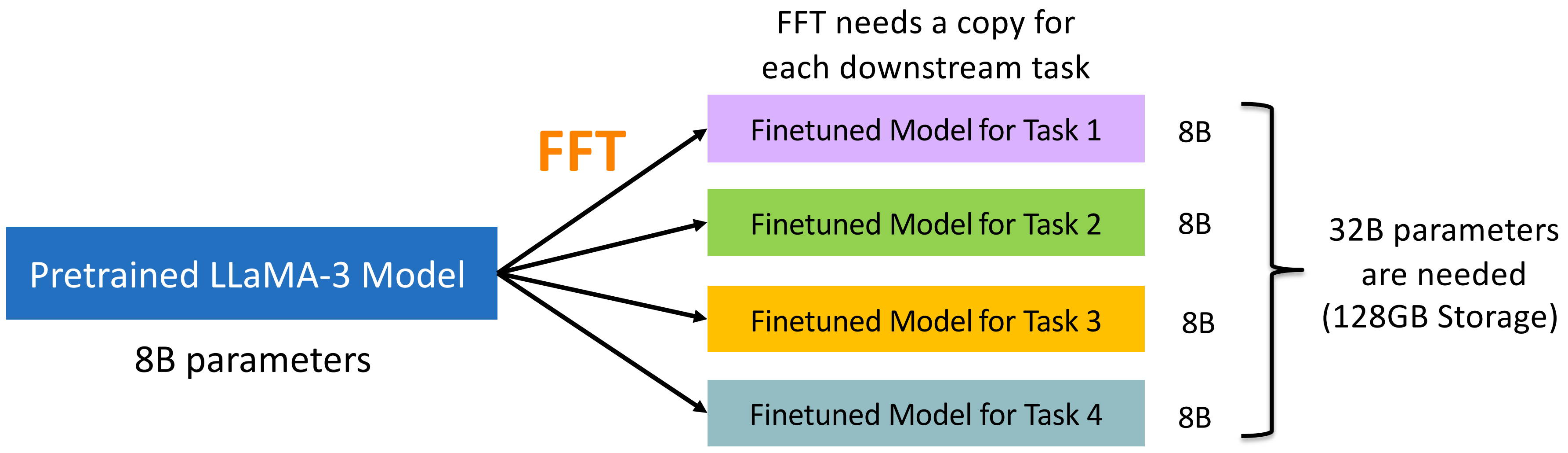
Naively Fine-Tuning LLaMA3-8B takes 128GB of RAM!

- **Fine-tuning small models like LLaMA3 8B on regular consumer GPUs can be challenging due to the significant memory requirements:**
  1. **Memory Requirements**: LLaMA3 8B has 8 billion parameters and if it's loaded in full-precision (float32 format-> 4 bytes/parameter), then the total memory requirements for loading the model would be numberOfParams*bytesPerParam = 8 billion*4 = **32GB of memory.**
     - Given that many consumer GPUs/ free versions of software like Google Colab have memory constraints (e.g., NVIDIA T4 16GB on Google Colab), the model cannot even be loaded!
  2. **Fine-Tuning memory requirements**: In the case of full fine-tuning with the regular 8bit Adam optimizer using a half-precision model (2 bytes/param), we need to allocate per parameter: 2 bytes for the weight, 2 bytes for the gradient, and 12 bytes for the Adam optimizer states. This results in a total of 16 bytes per trainable parameter, requiring over 120GB of GPU memory!!
     - This would require at least 3A40s with 48GB GPU VRAM, which would mean fine-tuning wouldn't be accessible by public.

https://medium.com/polo-club-of-data-science/memory-requirements-for-fine-tuning-llama-2-80f366cba7f5

Full Fine-Tuning (FFT)

- **Full Fine-Tuning** is required to updates all pre-trained model parameters to adapt to a new task, leading to improved performance. However, FFT has two main drawbacks:
  1. **Computational expense**: Even fine-tuning small models like LLaMA3 with 8B parameters can be resource-intensive.
  2. **Storage costs**: Saving entire models for each checkpoint (finetuned model) **can also be storage-prohibitive**.

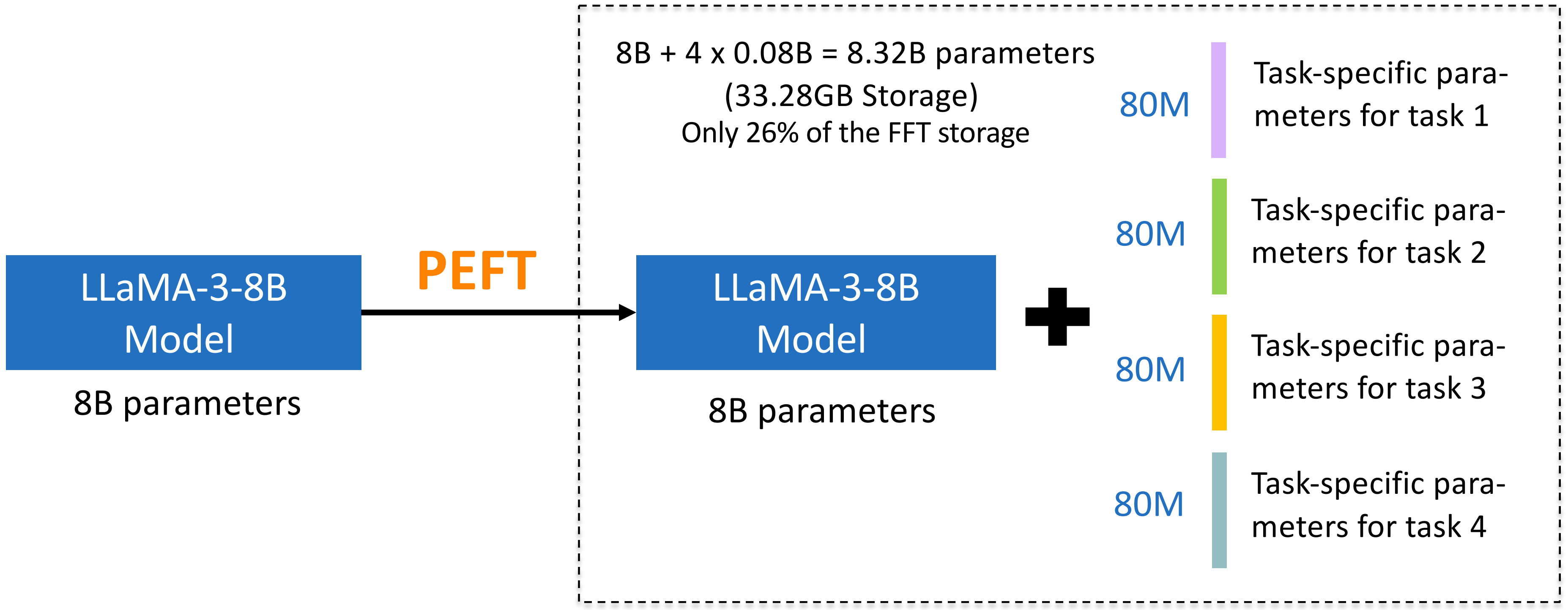

FFT needs a copy for each downstream task
FFT
Pretrained LLaMA-3 Model
8B parameters
Finetuned Model for Task 1
Finetuned Model for Task 2
Finetuned Model for Task 3
Finetuned Model for Task 4
8B
8B
8B
8B
32B parameters are needed (128GB Storage)

Parameter-Efficient Finetuning (PEFT)

- To address the issues of FFT limitations, **PEFT methods were developed, adapting only a small subset of a model's parameters to a new task.**
- The importance of PEFT in practical LLM applications lies in its ability to:
  1. Lower hardware requirements and reduce memory needs
  2. Speed up training times and reduce GPU usage
  3. Improve modeling performance by reducing overfitting
  4. **Minimize storage needs by sharing weights across tasks**

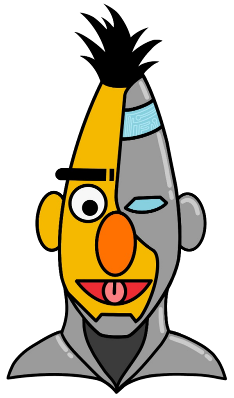
Reduce the Number of Parameters by PEFT

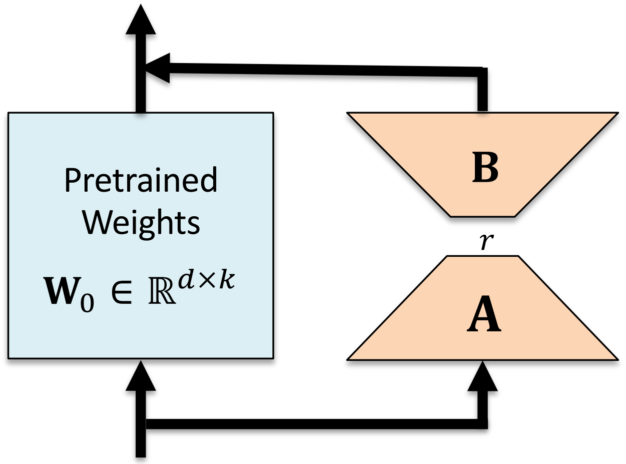
- **PEFT** enables the reuse of pretrained model weights, requiring only a small number of additional task-specific parameters.


LLaMA-3-8B Model
8B parameters
PEFT
8B + 4 x 0.08B = 8.32B parameters
(33.28GB Storage)
Only 26% of the FFT storage
LLaMA-3-8B Model
8B parameters
80M
Task-specific parameters for task 1
80M
Task-specific parameters for task 2
80M
Task-specific parameters for task 3
80M
Task-specific parameters for task 4

PEFT Techniques

1. **Adapter Tuning** (2019) – Add new intermediate modules
2. **LoRA** (2021) – Low-Rank decomposition
3. **QLoRA** (2023) – Quantized LoRA

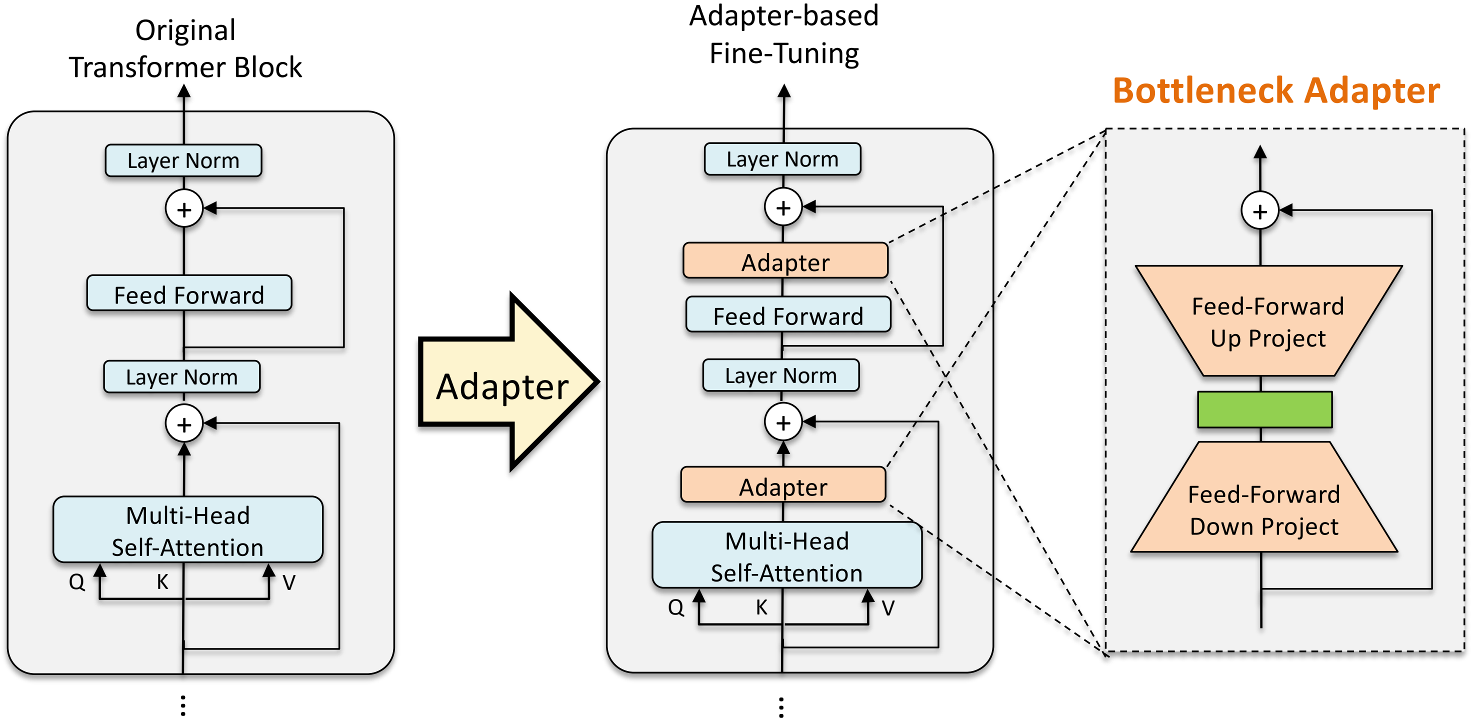




LoRA

Adapter-based Fine Tuning (Houlsby and Giurgiu et al., 2019-02)

- **Adapter Tuning** involves inserting **Adapter modules**, consisting of two bottleneck-style feedforward modules, into each transformer block of a pretrained LLM.



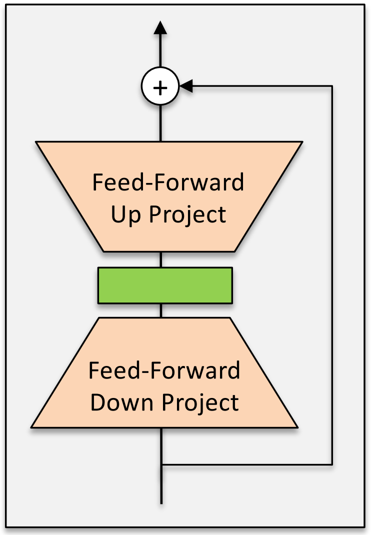
- The adapter Layers have **a small number of parameters and can be finetuned** while keeping the weights of the pre-trained model fixed.
- As such, each fine-tuned version of the model only has a small number of task-specific parameters associated with it.

https://arxiv.org/pdf/1902.00751.pdf

Why Adapter Layer with 2 Fully-Connected Layers?

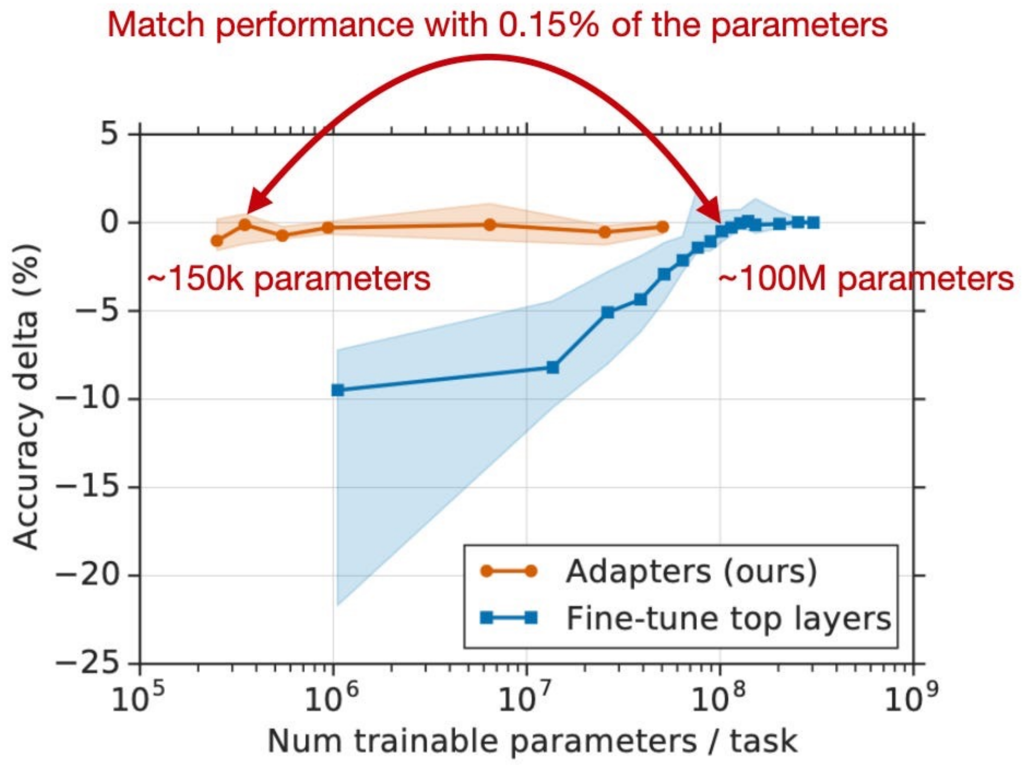
- Note that the fully-connected layers of the adapters are usually relatively small and **have a bottleneck structure**
  - Each adapter block's first fully-connected layer projects the input down onto **a low-dimensional representation**.
  - The second fully connected layer projects the input back into the input dimension.
    - For example, assume the first fully connected layer projects a 1024-dimensional input down to 24 dimensions, and the second fully connected layer projects it back into 1024 dimensions.
    - This means we introduced 1,024 x 24 + 24 x 1,024 = **49,152 weight parameters.**
  - In contrast, a single fully-connected layer that reprojects a 1024-dimensional input into a 1024-dimensional space would have 1024 x 1024 = **1,048,576 weight parameters.**

Bottleneck Adapter



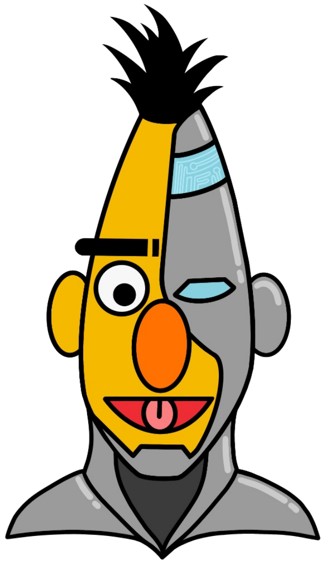
Adapter Performance on Finetuning BERT

1. **Adapter-trained BERT models achieve similar performance to fully fine-tuned ones** while training only **3.6%** of the parameters. This suggests significant parameter efficiency.
2. Adapters outperform top-layer finetuning using even fewer parameters. This implies higher efficiency than training just the output layers of BERT.

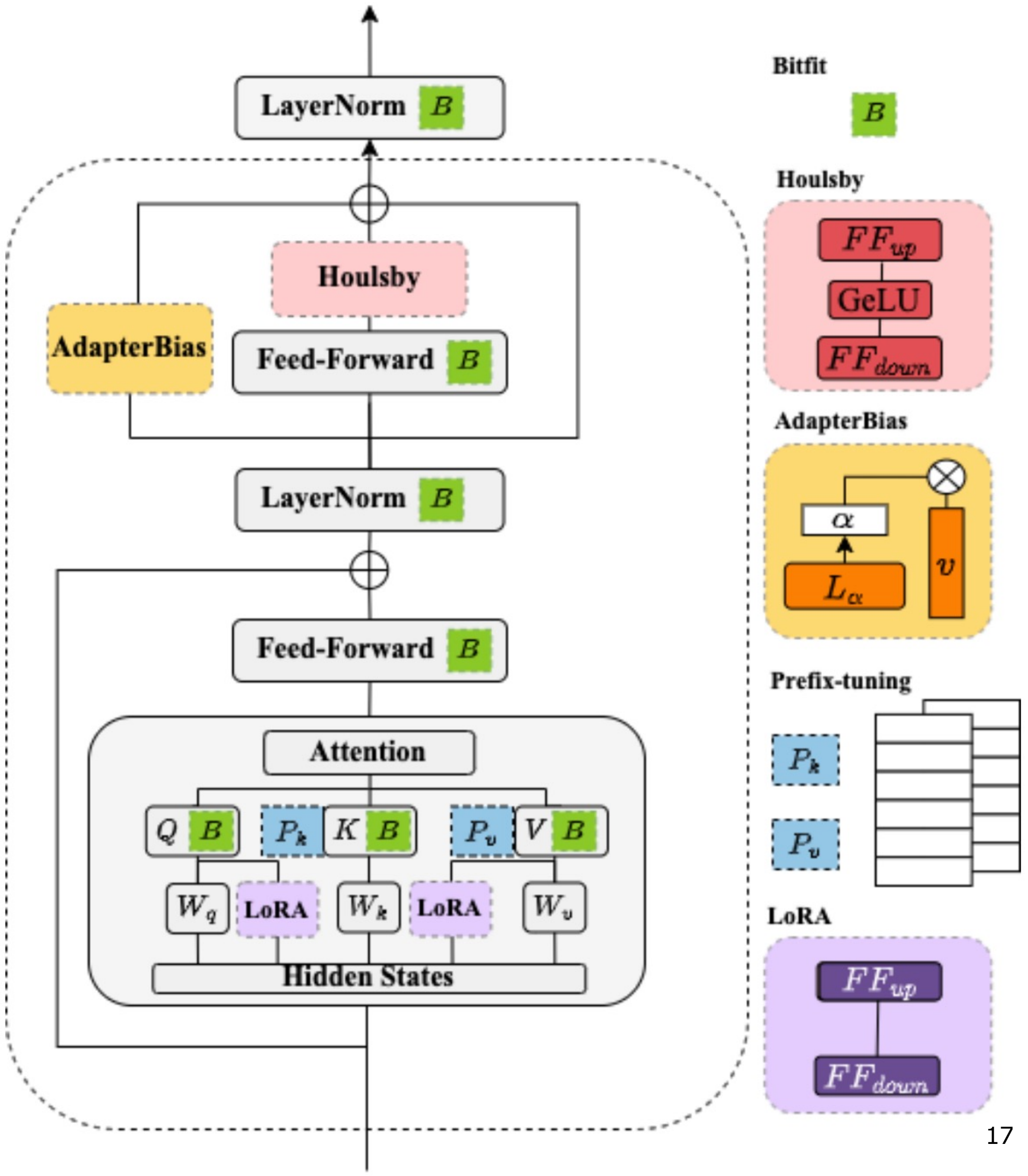


Other Types of Adapter

- https://adapterhub.ml/



https://arxiv.org/abs/2210.06175

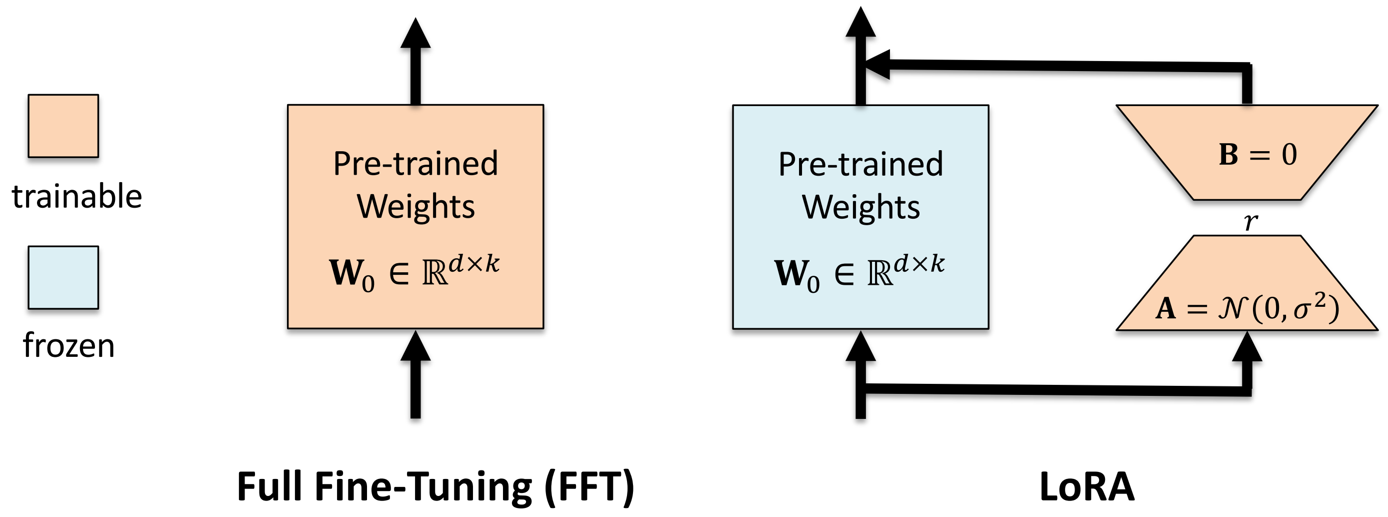


Low-Rank Adaptation (LoRA)

https://arxiv.org/pdf/2106.09685.pdf

LoRA (Edward Hu et al., 2021-06)

- **Low-Rank Adaptation (LoRA)** is a groundbreaking technique of PEFT for LLMs.
- It introduces a **parallel low-rank adapter** to the weights of linear layers **reducing memory overhead and computational costs during finetuning.**



trainable
frozen
Pre-trained Weights
$\mathbf{W}_0 \in \mathbb{R}^{d\times k}$
Pre-trained Weights
$\mathbf{W}_0 \in \mathbb{R}^{d\times k}$
$\mathbf{B} = 0$
r
$\mathbf{A} = \mathcal{N}(0, \sigma^2)$
Full Fine-Tuning (FFT)
LoRA

https://arxiv.org/abs/2106.09685

What is LoRA?

- **LoRA – Low-Rank Adaptation**
  - **Low-rank**: Rank r of the matrix is **smaller than** matrix's dimension d
  - **Rank**: Minimum number of independent rows/columns
  - **Adaptation**: **Fine-tuning of models**

Rank-1 Matrix

$$rank \begin{bmatrix} 2 & 10 & 1 \\ 4 & 20 & 2 \\ 6 & 30 & 3 \end{bmatrix} = 1$$

Rank-2 Matrix

$$rank \begin{bmatrix} 2 & 10 & 1 \\ 7 & 20 & 2 \\ 6 & 30 & 3 \end{bmatrix} = 2$$

Rank-3 Matrix

$$rank \begin{bmatrix} 2 & 3 & 1 \\ 7 & 5 & 2 \\ 6 & 1 & 3 \end{bmatrix} = 3$$

$$d = 3$$

Recap: Matrix Rank

- The rank of a matrix $\mathbf{A} \in \mathbb{R}^{d \times k}$ is equal to the minimum number of linearly independent columns or rows, and always satisfies:

$$\operatorname{rank}(\mathbf{A}) \leq \min(d, k)$$

- A matrix with $\operatorname{rank}(\mathbf{A}) = \min(d, k)$ is called a **Full-Rank Matrix**. For example, the follow 3x3 matrix has a rank of 3, making it a full-rank matrix:

$$\operatorname{rank}\left(\begin{bmatrix} 2 & 3 & 1 \\ 7 & 5 & 2 \\ 6 & 1 & 3 \end{bmatrix}\right) = 3$$

- A matrix with $\operatorname{rank}(\mathbf{A}) < \min(d, k)$ is called a **Low-Rank Matrix**. Here are two examples of low-rank matrices:

$$\operatorname{rank}\left(\begin{bmatrix} 2 & 10 & 1 \\ 4 & 20 & 2 \\ 6 & 30 & 3 \end{bmatrix}\right) = 1 \quad \operatorname{rank}\left(\begin{bmatrix} 2 & 10 & 1 \\ 7 & 20 & 2 \\ 6 & 30 & 3 \end{bmatrix}\right) = 2$$

Rep: Rank Matrix Decomposition

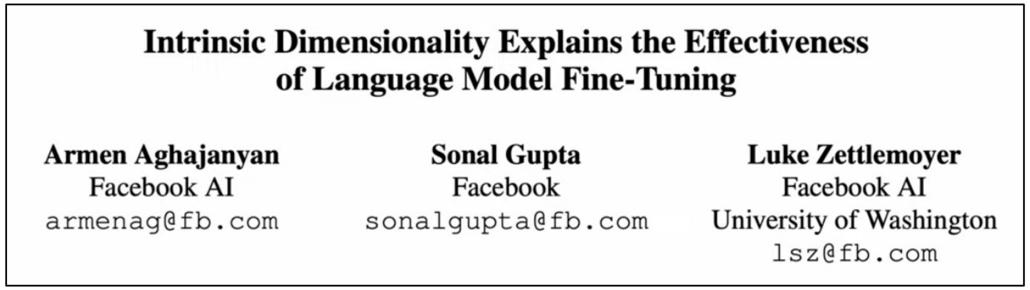
- **Low-Rank Matrices can be Decomposed into Low-Dimensional Matrices**
- A low-rank matrix can be decomposed into the product of two low-dimensional matrices. For instance, a rank-1 3x3 matrix can be decomposed as follows:

$$\underbrace{\begin{bmatrix} 2 & 10 & 1 \\ 4 & 20 & 2 \\ 6 & 30 & 3 \end{bmatrix}}_{3\times 3} = \underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}}_{3\times 1} \times \underbrace{\begin{bmatrix} 2 & 20 & 30 \end{bmatrix}}_{1\times 3}$$

  - This decomposition reduces the number of coefficients needed to represent the matrix from 9 to 6 = (3×1 + 1×3).
- In general, a rank-r $n\times n$ matrix can be decomposed into an $n\times r$ matrix and an $r\times n$ matrix.
- For r=1, the reduction in coefficients is: $n^2 \Rightarrow 2n$
- The general reduction in coefficients for a rank-r matrix is: $n^2 \Rightarrow 2\cdot r\cdot n$

Motivation of LoRA

- Core Finding: **Low Intrinsic Dimensionality in Language Models**
- Significance of Intrinsic Dimensionality:
  - Intrinsic dimensionality is a crucial metric that explains why large language models are efficiently fine-tunable with limited data.
- Brader Impact:
  - Understanding intrinsic dimensionality could lead to more resource-efficient and effective ways to train and deploy language Models



Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning

Armen Aghajanyan
Facebook AI
armenag@fb.com

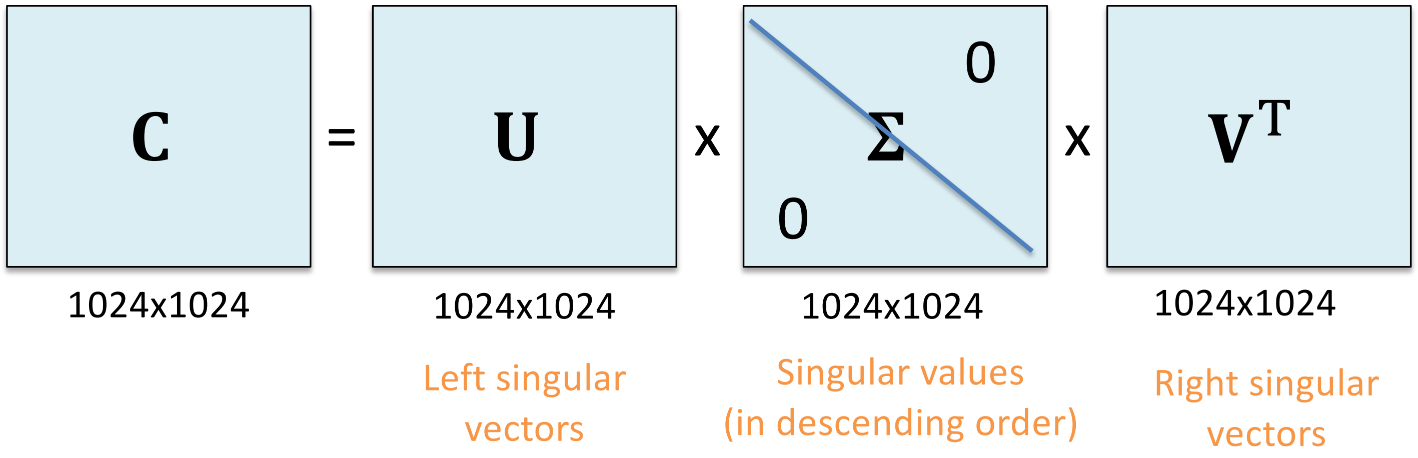
Sonal Gupta
Facebook
sonalgupta@fb.com

Luke Zettlemoyer
Facebook AI
University of Washington
lsz@fb.com

Recap: Singular Value Decomposition (SVD)

- **SVD** is a general matrix factorization technique.
  - $\mathbf{U}$: basis vectors for column space
  - $\mathbf{V}^{\mathrm{T}}$: basic vectors for row space
  - $\boldsymbol{\Sigma}$: diagonal matrix of singular values

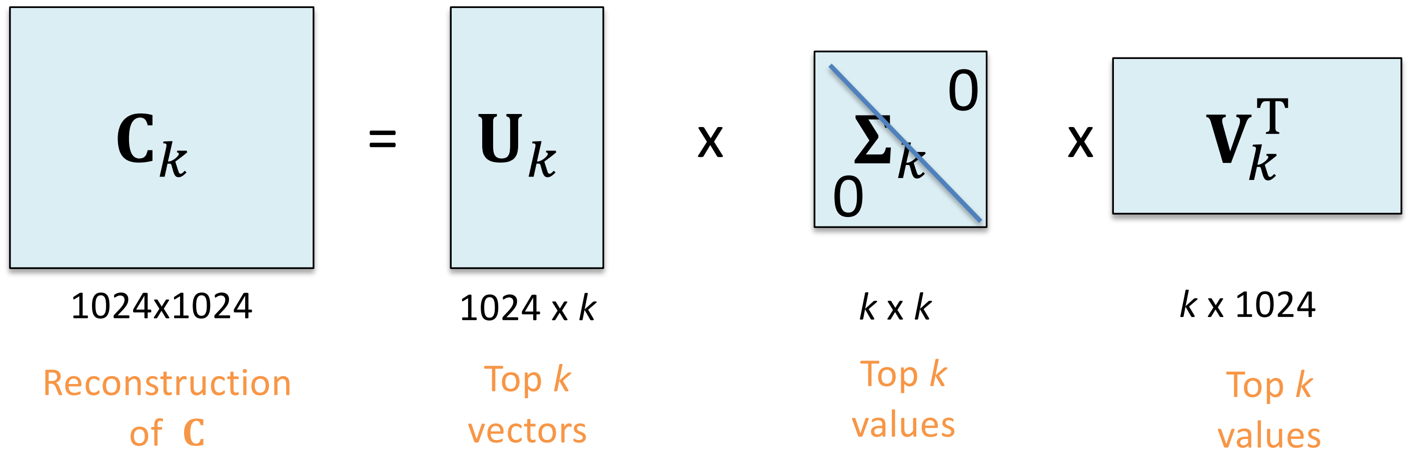
$$\mathbf{C} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}}$$



Low-Rank Approximation with SVD

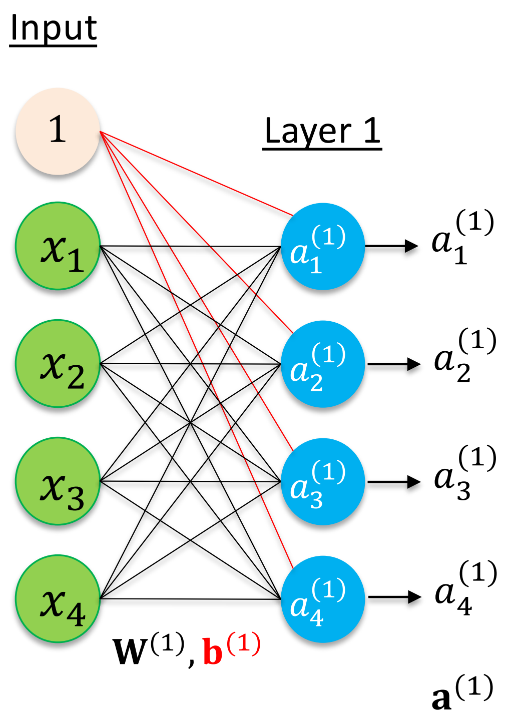
- We can approximate $n \times m$ matrix $\mathbf{C}$ by keeping only the k largest singular values
- k is called the rank
- Fewer parameters are required
  - $\mathbf{C}$: 2^{20} parameters
  - $\mathbf{U}_k, \mathbf{\Sigma}_k, \mathbf{V}_k^{\mathrm{T}}$: $2 \times (2^{10} \times k) + k^2$
  - If $k = 9$: 16448 parameters (1.56%)
  - Frobenius norm: the difference between $\mathbf{C}$ and $\mathbf{C}_k$

$$\mathbf{C} \approx \mathbf{C}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^{\mathrm{T}}$$



Recap: Matrix Representation of FFN

- Formulation of **Hidden Layer 1**: $\mathbf{a}^{(1)} = g(\boldsymbol{z}^{(1)}) = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$



https://arxiv.org/pdf/2012.13255.pdf

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad \boxed{\mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} & w_{1,4}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} & w_{2,4}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} & w_{3,4}^{(1)} \\ w_{4,1}^{(1)} & w_{4,2}^{(1)} & w_{4,3}^{(1)} & w_{4,4}^{(1)} \end{bmatrix}} \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ b_4^{(1)} \end{bmatrix}$$

- **Common pre-trained models** have been empirically shown that having a very low intrinsic dimension (low-rank)
- In other words, there exists a low dimension reparameterization that is as effective for finetuning as the full-parameter space.

LoRA (Low-Rank Adaptation)

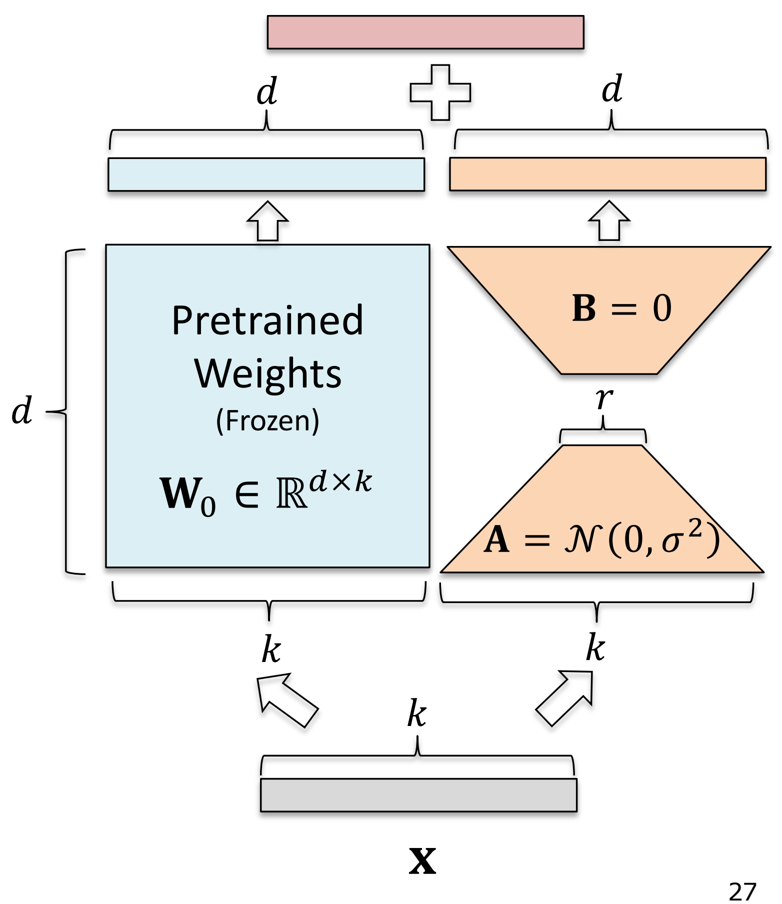
- **Use Low-rank submodules** to modify hidden representations
  - Pretrained Weights: $\mathbf{W}_0 \in \mathbb{R}^{d\times k}$
  - Introduce two new smaller matrices $\mathbf{A} \in \mathbb{R}^{r\times k}$ and $\mathbf{B} \in \mathbb{R}^{d\times r}$
  - $r \ll \min(d, k)$ and r is usually between 1 to 32
  - The input $\mathbf{x} \in \mathbb{R}^{k\times 1}$ and output $\mathbf{h} \in \mathbb{R}^{d\times 1}$ are **column vectors**

$$\mathbf{h} = (\mathbf{W}_0 + \Delta\mathbf{W})\mathbf{x} = (\mathbf{W}_0 + \mathbf{BA})\mathbf{x}$$

where $\Delta\mathbf{W} = \mathbf{BA}$ is a low-rank matrix

https://arxiv.org/abs/2106.09685

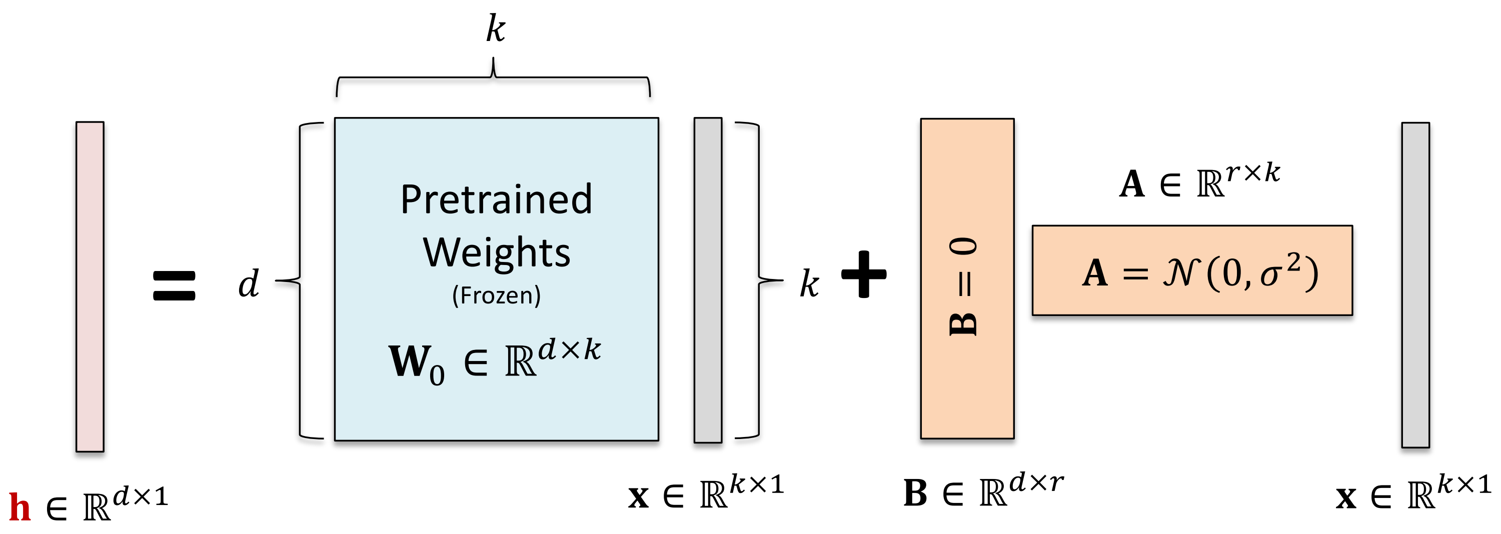
$$\mathbf{h} = \mathbf{W}_0\mathbf{x} + \mathbf{BA}\mathbf{x}$$



d
d
d
Pretrained Weights
(Frozen)
$\mathbf{W}_0 \in \mathbb{R}^{d\times k}$
$\mathbf{B} = 0$
r
$\mathbf{A} = \mathcal{N}(0, \sigma^2)$
k
k
k
$\mathbf{x}$

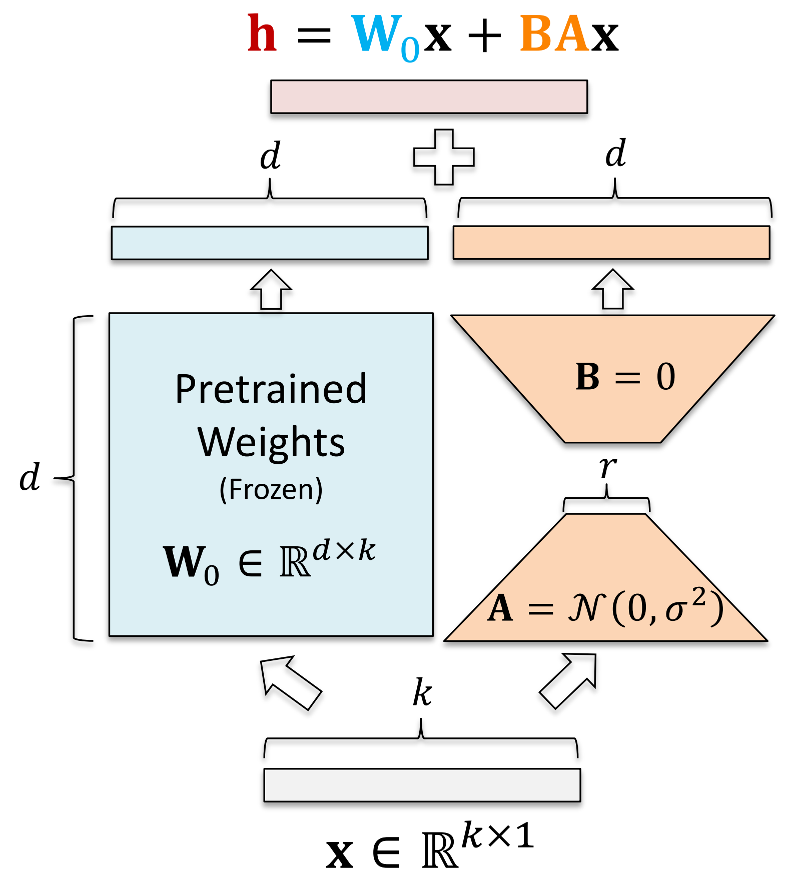
LoRA (Low-Rank Adaptation)

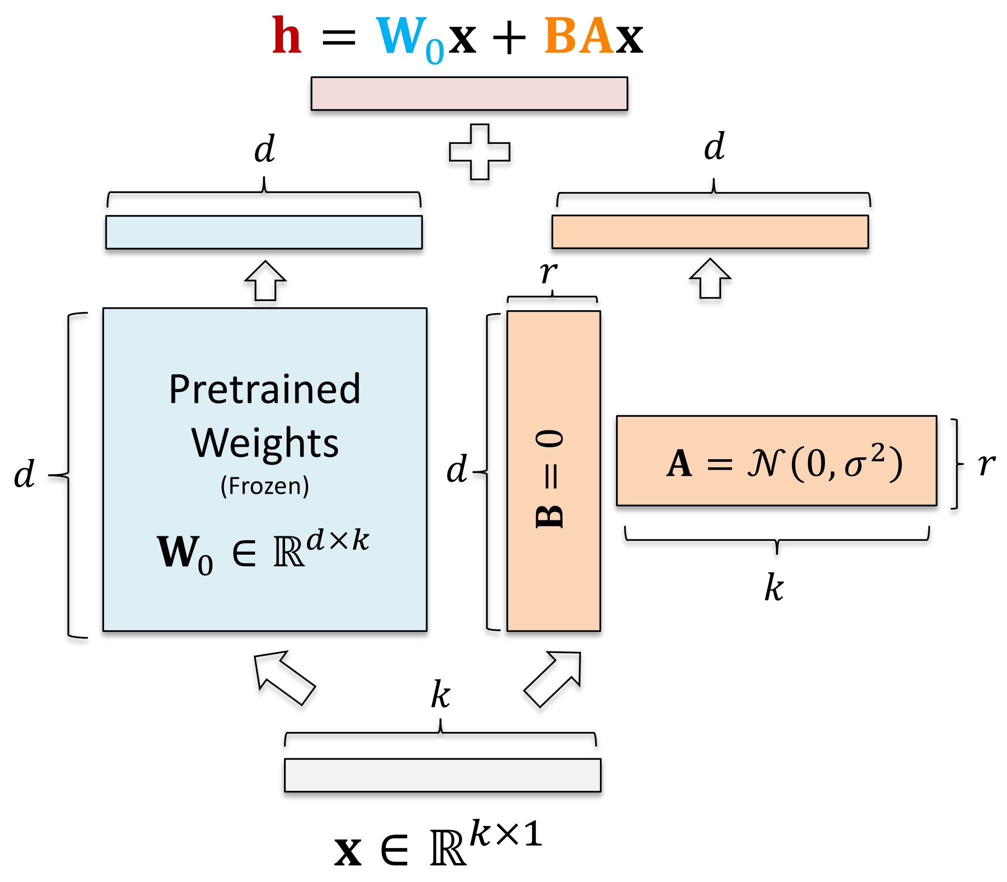
$$\mathbf{h} = \mathbf{W}_0\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$$



k
Pretrained Weights
(Frozen)
$\mathbf{W}_0 \in \mathbb{R}^{d\times k}$
d
k
$\mathbf{B} = 0$
$\mathbf{A} \in \mathbb{R}^{r\times k}$
$\mathbf{A} = \mathcal{N}(0,\sigma^2)$
$\mathbf{h} \in \mathbb{R}^{d\times 1}$
$\mathbf{x} \in \mathbb{R}^{k\times 1}$
$\mathbf{B} \in \mathbb{R}^{d\times r}$
$\mathbf{x} \in \mathbb{R}^{k\times 1}$

LoRA (Low-Rank Adaptation)


$\mathbf{h} = \mathbf{W}_0\mathbf{x} + \mathbf{BAx}$
d
d
d
Pretrained Weights
(Frozen)
$\mathbf{W}_0 \in \mathbb{R}^{d \times k}$
$\mathbf{B} = 0$
r
$\mathbf{A} = \mathcal{N}(0, \sigma^2)$
k
$\mathbf{x} \in \mathbb{R}^{k \times 1}$


$\mathbf{h} = \mathbf{W}_0\mathbf{x} + \mathbf{BAx}$
d
d
r
d
Pretrained Weights
(Frozen)
$\mathbf{W}_0 \in \mathbb{R}^{d \times k}$
d
$\mathbf{B} = 0$
$\mathbf{A} = \mathcal{N}(0, \sigma^2)$
r
k
k
$\mathbf{x} \in \mathbb{R}^{k \times 1}$

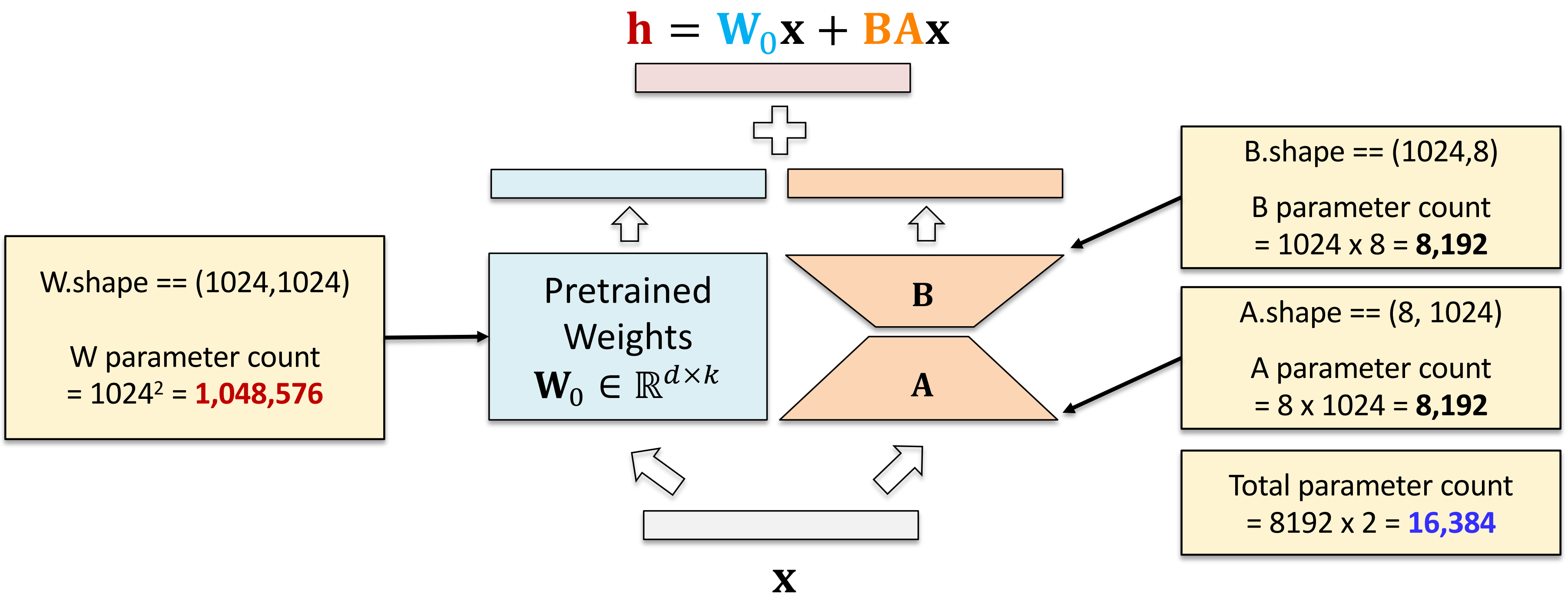
LoRA: $\Delta\mathbf{W} = \mathbf{BA}$

$$\Delta\mathbf{W} = \mathbf{BA} = \underbrace{\begin{bmatrix} 0.3 & -0.14 \\ -0.42 & 0.201 \\ 0.46 & 0.38 \\ 0.5 & 0.14 \end{bmatrix}}_{\mathbf{B}} \underbrace{\begin{bmatrix} 0.1 & -0.44 & 0.04 & 1.42 \\ -0.92 & 0.1 & 1.62 & -1.33 \end{bmatrix}}_{\mathbf{A}} = \underbrace{\begin{bmatrix} 0.15 & -0.14 & -0.21 & 0.612 \\ -0.22 & 0.204 & 0.308 & -0.86 \\ -0.30 & -0.16 & 0.634 & 0.147 \\ -0.07 & -0.2 & 0.246 & 0.523 \end{bmatrix}}_{\Delta\mathbf{W}}$$

$$\mathbf{h} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \mathbf{W}_0\mathbf{x} + \Delta\mathbf{W}\mathbf{x} = \underbrace{\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix}}_{\mathbf{W}_0} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \underbrace{\begin{bmatrix} 0.15 & -0.14 & -0.21 & 0.612 \\ -0.22 & 0.204 & 0.308 & -0.86 \\ -0.30 & -0.16 & 0.634 & 0.147 \\ -0.07 & -0.2 & 0.246 & 0.523 \end{bmatrix}}_{\Delta\mathbf{W}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

LoRA High Parameter Efficiency Example

- X% of weights trained => (a) lower memory footprint and (b) faster finetuning jobs



$\mathbf{h} = \mathbf{W}_0\mathbf{x} + \mathbf{BAx}$
Pretrained Weights
$\mathbf{W}_0 \in \mathbb{R}^{d\times k}$
B
A
x

W.shape == (1024,1024)

W parameter count
= 1024^2 = **1,048,576**

B.shape == (1024,8)

B parameter count
= 1024 x 8 = **8,192**

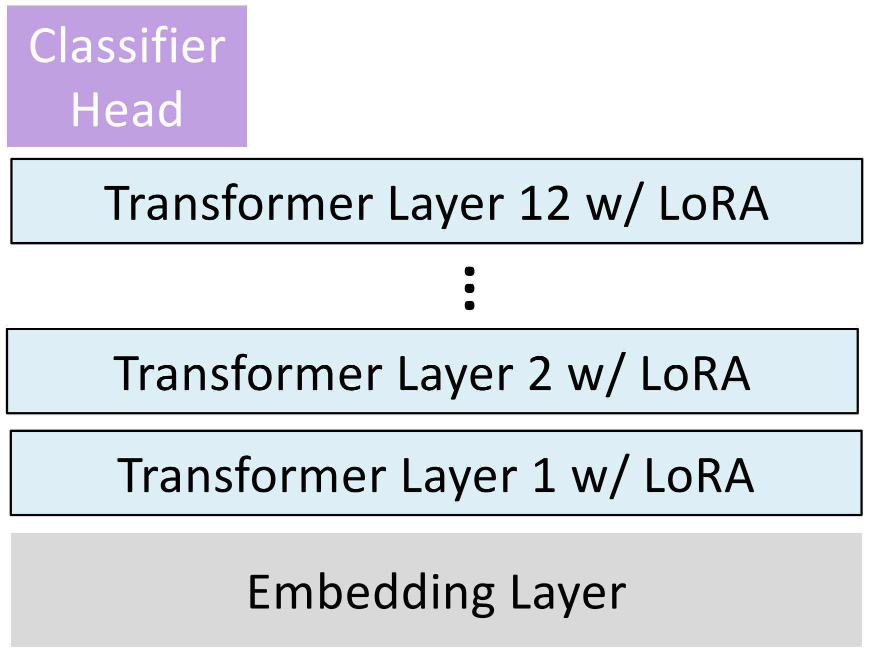
A.shape == (8, 1024)

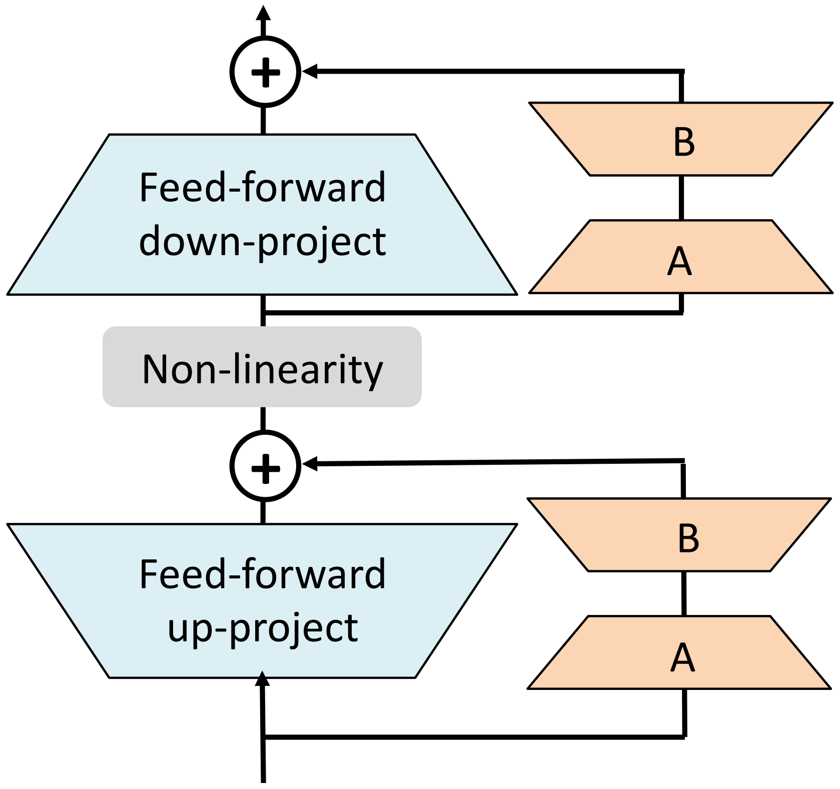
A parameter count
= 8 x 1024 = **8,192**

Total parameter count
= 8192 x 2 = **16,384**

Applying LoRA to Feed-Forward Networks

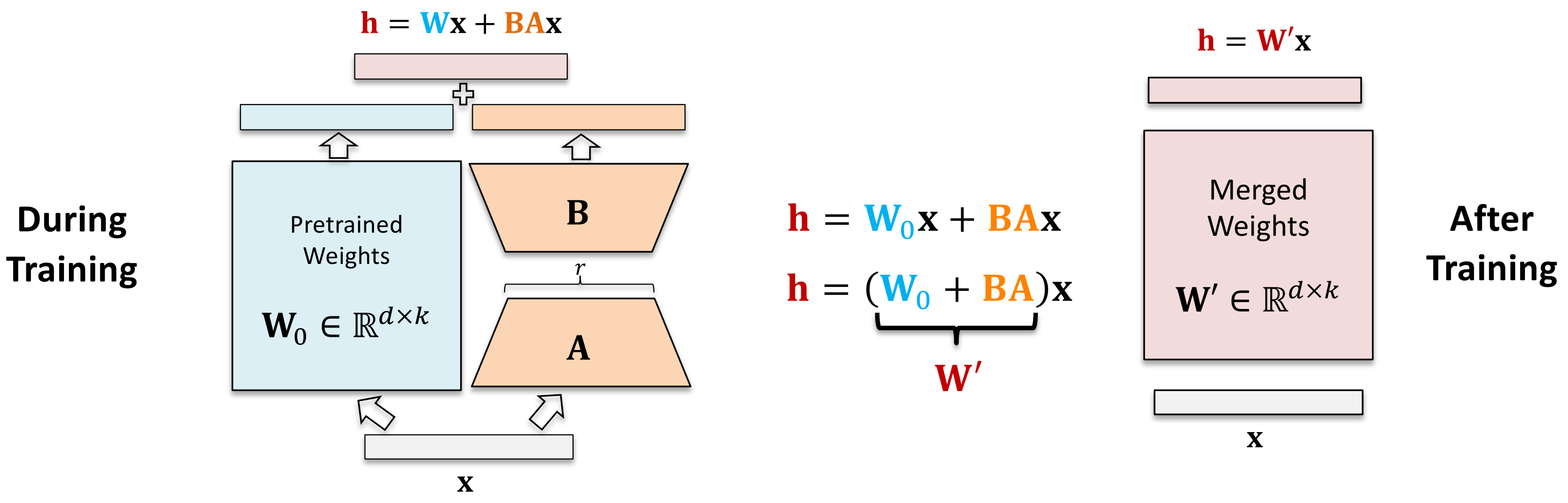
- **LoRA** can be applied to each **fully-connected layer and the classifier heads** are the task-specific modules


Classifier Head
Transformer Layer 12 w/ LoRA
⋮
Transformer Layer 2 w/ LoRA
Transformer Layer 1 w/ LoRA
Embedding Layer



Benefits of LoRA

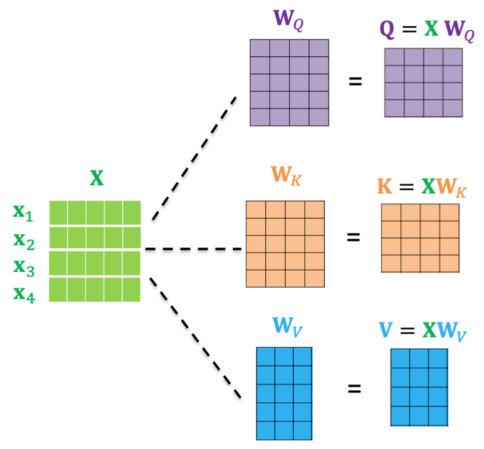
- **Less Parameters**: LoRA reduces computational requirements during training, leading to faster training and lower memory usage.
- **Flexibility**: Switch between different LoRA weights
- **Seamless Integration**: The $\Delta\mathbf{W}$ ($\mathbf{BA}$) weights from the rank decomposition can be **merged** with the original model weights by simply adding them together, without introducing any overhead during inference.



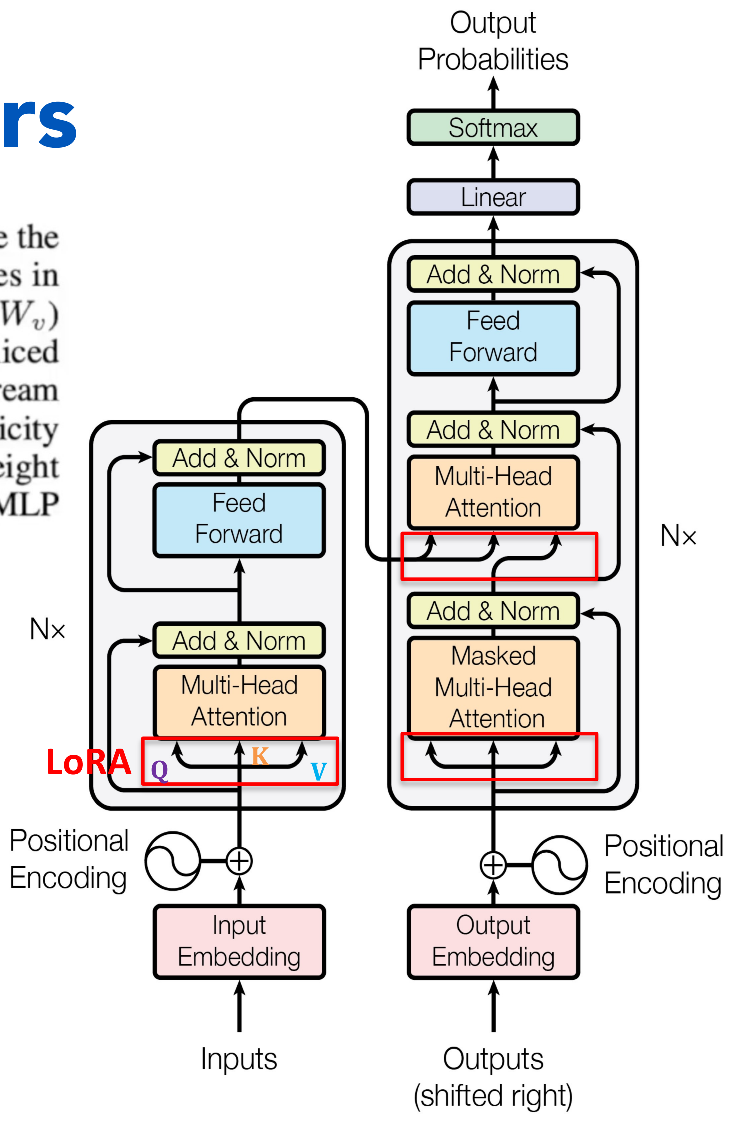
$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$
During Training
Pretrained Weights
$\mathbf{W}_0 \in \mathbb{R}^{d\times k}$
B
r
A
x
$\mathbf{h} = \mathbf{W}_0\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$
$\mathbf{h} = \underbrace{(\mathbf{W}_0 + \mathbf{B}\mathbf{A})}_{\mathbf{W}'}\mathbf{x}$
$\mathbf{h} = \mathbf{W}'\mathbf{x}$
Merged Weights
$\mathbf{W}' \in \mathbb{R}^{d\times k}$
x
After Training

Applying LoRA To Transformers

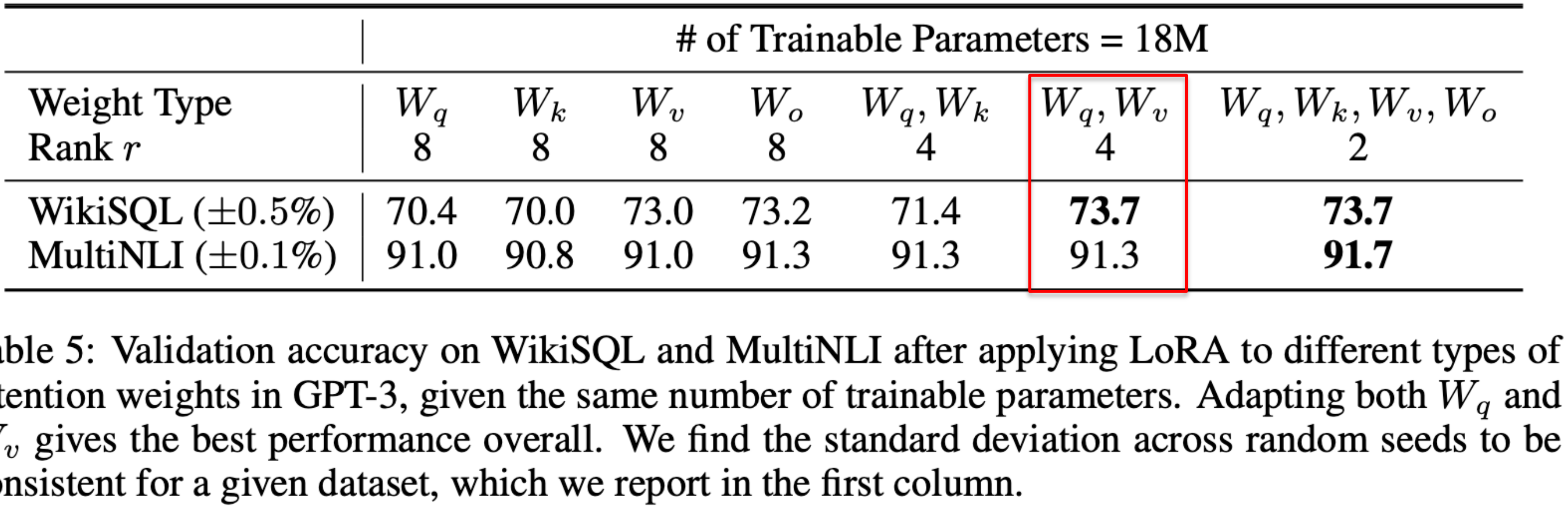
In principle, we can apply LoRA to any subset of weight matrices in a neural network to reduce the number of trainable parameters. In the Transformer architecture, there are four weight matrices in the self-attention module (W_q, W_k, W_v, W_o) and two in the MLP module. We treat W_q (or W_k, W_v) as a single matrix of dimension $d_{model} \times d_{model}$, even though the output dimension is usually sliced into attention heads. We limit our study to **only adapting the attention weights** for downstream tasks and freeze the MLP modules (so they are not trained in downstream tasks) both for simplicity and parameter-efficiency. We further study the effect on adapting different types of attention weight matrices in a Transformer in Section 7.1. We leave the empirical investigation of adapting the MLP layers, LayerNorm layers, and biases to a future work.



$\mathbf{W}_Q$
$\mathbf{Q} = \mathbf{X}\,\mathbf{W}_Q$
X
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
$\mathbf{W}_K$
$\mathbf{K} = \mathbf{X}\mathbf{W}_K$
$\mathbf{W}_V$
$\mathbf{V} = \mathbf{X}\mathbf{W}_V$
=



Which Weight Matrices?



	# 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**

Table 5: Validation accuracy on WikiSQL and MultiNLI after applying LoRA to different types of attention weights in GPT-3, given the same number of trainable parameters. Adapting both W_q and W_v gives the best performance overall. We find the standard deviation across random seeds to be consistent for a given dataset, which we report in the first column.

https://arxiv.org/pdf/2106.09685.pdf

Scaling Factor α

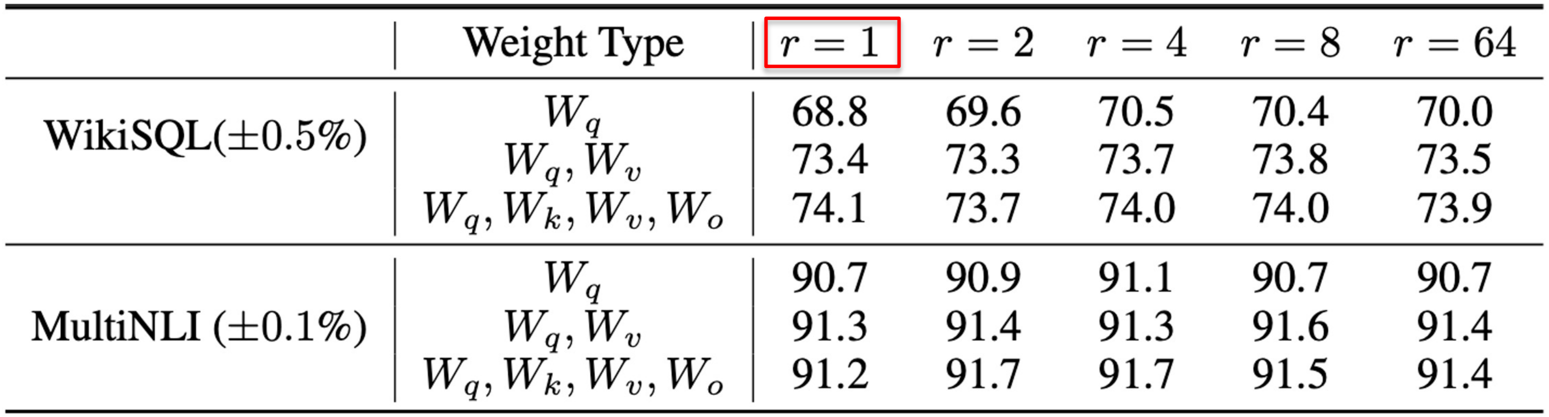
- The **scaling factor** α is used **to adjust the output** of matrices $\mathbf{B}$ and $\mathbf{A}$.

$$\mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \frac{\overset{\text{Scaling factor}}{\alpha}}{\underset{\text{Rank}}{r}}\mathbf{BA}$$

- It is divided by the rank r, which **represents the intrinsic dimension and determines the level of decomposition or compression applied to the weights.**
- Typically, the rank ranges from 1 to 64, while the scaling factor α **controls the amount of change applied to the original model weights**, striking a balance between the knowledge of the pre-trained model and its adaptation to a new task.
- Both the α and r are hyperparameters, which need to be tuned.
  - Basically, the scaling factor α helps in stabilizing other hyperparameters, such as learning rates, when the rank is varied. By adjusting the rank and incorporating the scaling factor, one can explore different levels of decomposition without needing to extensively tweak other parameters. This approach simplifies the process of finding the optimal level of decomposition for a given task.

How Low-Rank can LoRA go?

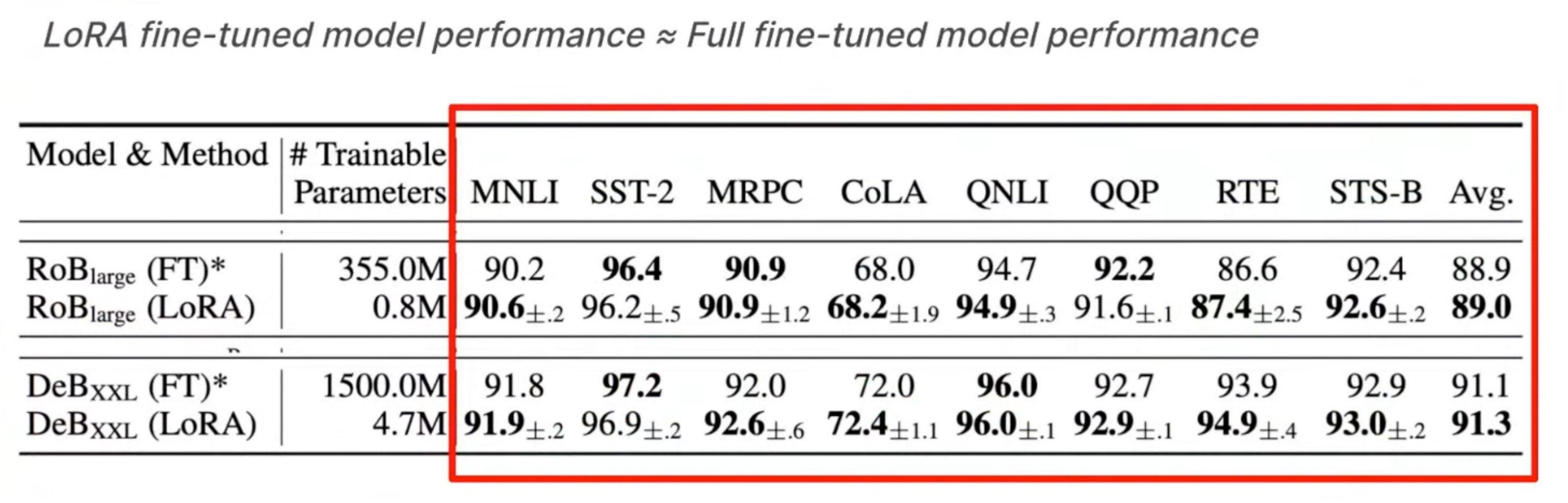
- LoRA works even with extremely small values of *r* such as 4, 2, or even 1.
- On the WikiSQL and MultNLI problem datasets, the authors found no statistically significant difference in performance when reducing the rank *r*=64 to *r*=1.



	Weight Type	$\boxed{r=1}$	$r=2$	$r=4$	$r=8$	$r=64$
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

https://arxiv.org/pdf/2106.09685.pdf

LoRA Performance Parity with Fully Finetuned LLMs

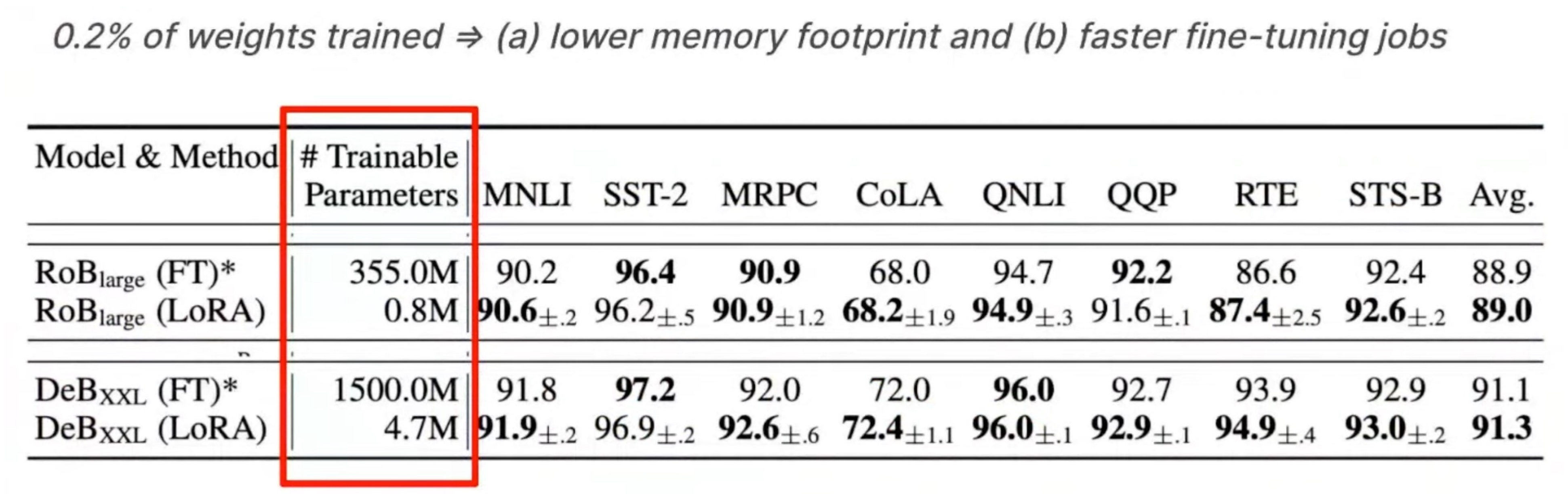


LoRA fine-tuned model performance ≈ Full fine-tuned model performance

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	**90.6**$_{\pm.2}$	96.2$_{\pm.5}$	**90.9**$_{\pm1.2}$	**68.2**$_{\pm1.9}$	**94.9**$_{\pm.3}$	91.6$_{\pm.1}$	**87.4**$_{\pm2.5}$	**92.6**$_{\pm.2}$	**89.0**
DeB_{XXL} (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB_{XXL} (LoRA)	4.7M	**91.9**$_{\pm.2}$	96.9$_{\pm.2}$	**92.6**$_{\pm.6}$	**72.4**$_{\pm1.1}$	**96.0**$_{\pm.1}$	**92.9**$_{\pm.1}$	**94.9**$_{\pm.4}$	**93.0**$_{\pm.2}$	**91.3**

LoRA is extremely competitive with full finetuning

Extremely Parameter Efficient Finetuning



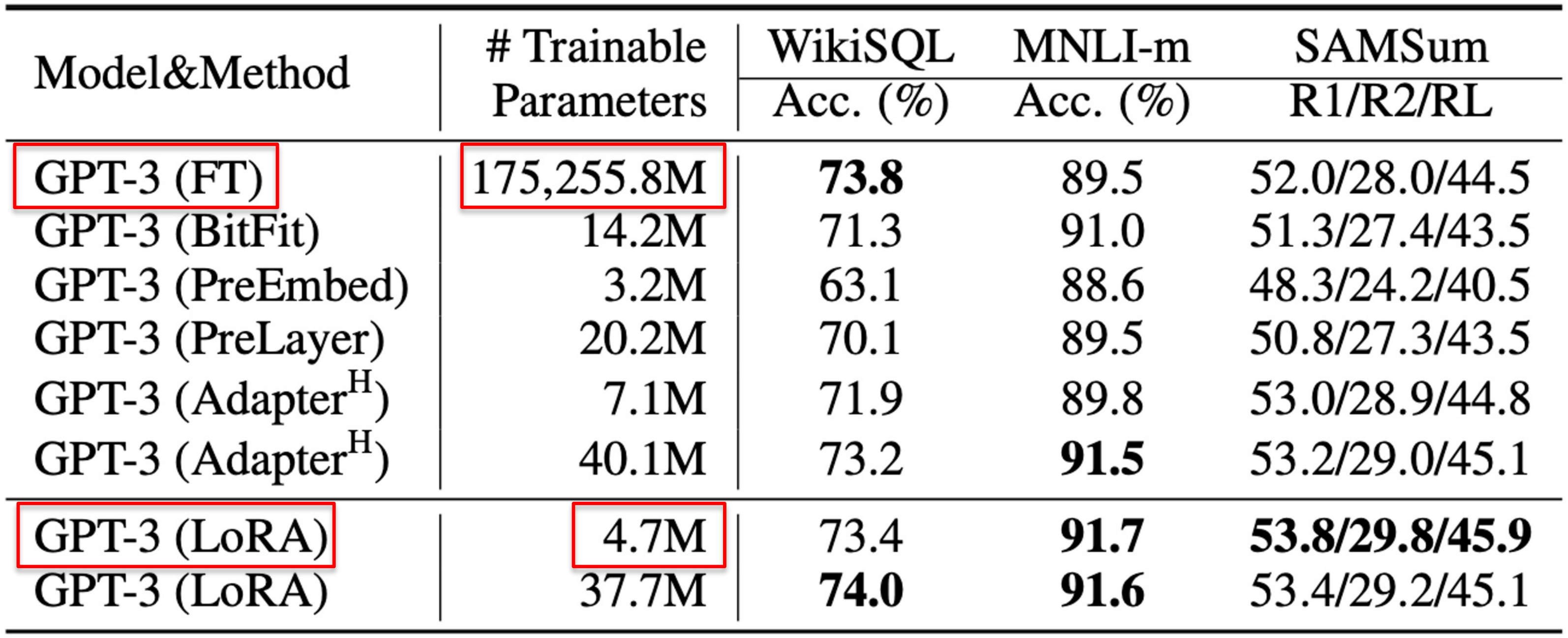
0.2% of weights trained ⇒ (a) lower memory footprint and (b) faster fine-tuning jobs

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB$_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
RoB$_{\text{large}}$ (LoRA)	0.8M	**90.6**$_{\pm.2}$	96.2$_{\pm.5}$	**90.9**$_{\pm1.2}$	**68.2**$_{\pm1.9}$	**94.9**$_{\pm.3}$	91.6$_{\pm.1}$	**87.4**$_{\pm2.5}$	**92.6**$_{\pm.2}$	**89.0**
DeB$_{\text{XXL}}$ (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB$_{\text{XXL}}$ (LoRA)	4.7M	**91.9**$_{\pm.2}$	96.9$_{\pm.2}$	**92.6**$_{\pm.6}$	**72.4**$_{\pm1.1}$	**96.0**$_{\pm.1}$	**92.9**$_{\pm.1}$	**94.9**$_{\pm.4}$	**93.0**$_{\pm.2}$	**91.3**

And the number of traninable parameter is less than 1% of the total model size

LoRA Comparison on GPT-3

LoRA reduces the number of trainable parameters in GPT-3 by **5 orders of magnitude**!

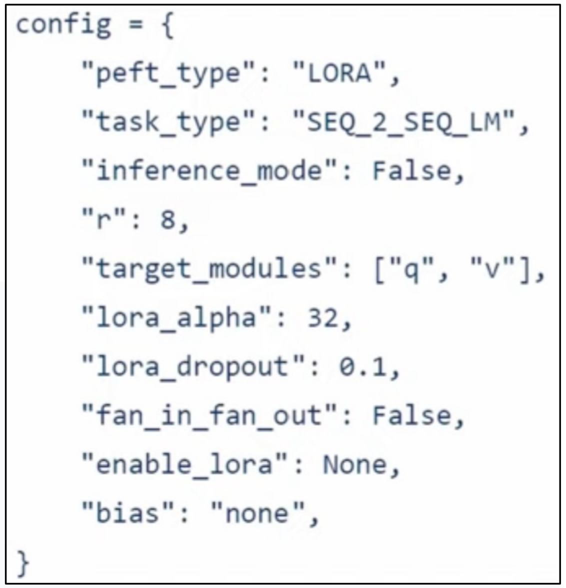


Model&Method	# Trainable Parameters	WikiSQL	MNLI-m	SAMSum
		Acc. (%)	Acc. (%)	R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 (Adapter^H)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 (Adapter^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

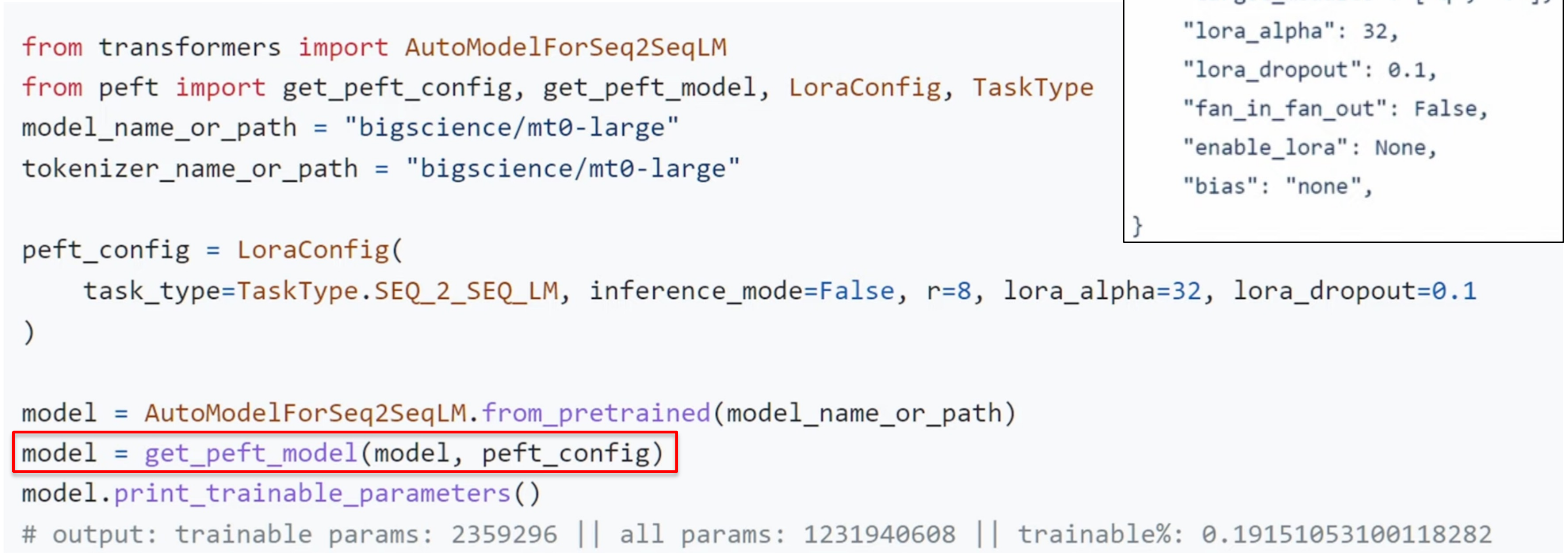
https://arxiv.org/pdf/2106.09685.pdf

Hugging Face PEFT

https://huggingface.co/docs/peft/en/index



```
config = {
    "peft_type": "LORA",
    "task_type": "SEQ_2_SEQ_LM",
    "inference_mode": False,
    "r": 8,
    "target_modules": ["q", "v"],
    "lora_alpha": 32,
    "lora_dropout": 0.1,
    "fan_in_fan_out": False,
    "enable_lora": None,
    "bias": "none",
}
```



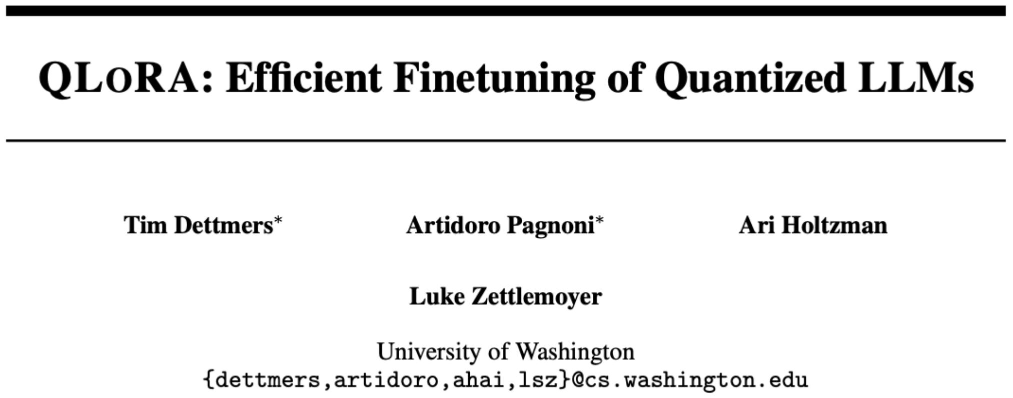
```
from transformers import AutoModelForSeq2SeqLM
from peft import get_peft_config, get_peft_model, LoraConfig, TaskType
model_name_or_path = "bigscience/mt0-large"
tokenizer_name_or_path = "bigscience/mt0-large"

peft_config = LoraConfig(
    task_type=TaskType.SEQ_2_SEQ_LM, inference_mode=False, r=8, lora_alpha=32, lora_dropout=0.1
)

model = AutoModelForSeq2SeqLM.from_pretrained(model_name_or_path)
model = get_peft_model(model, peft_config)
model.print_trainable_parameters()
# output: trainable params: 2359296 || all params: 1231940608 || trainable%: 0.19151053100118282
```

We don't have to manually apply a low-rank decomposition to each layer individually. Instead, we can use the "get_path_model" function, which takes care of this process for us.

LoRA: A New Paradigm Shift in NLP

- **LoRA enables us to adapt pretrained LLMs to specific downstream tasks faster, more robustly, and with orders of magnitudes fewer learnable parameters compared to standard fine-tuning.**
  - LoRA's success suggests low-rank, coarse-grained weight updates during fine-tuning, akin to "remembering" over "learning".
  - Lowest possible rank depends on downstream task difficulty relative to pre-training.
  - Lower ranks expected in earlier Transformer layers, higher ranks in later layers.

QLoRA



QLoRA: Efficient Finetuning of Quantized LLMs

Tim Dettmers[*] **Artidoro Pagnoni**[*] **Ari Holtzman**

Luke Zettlemoyer

University of Washington
`{dettmers,artidoro,ahai,lsz}@cs.washington.edu`

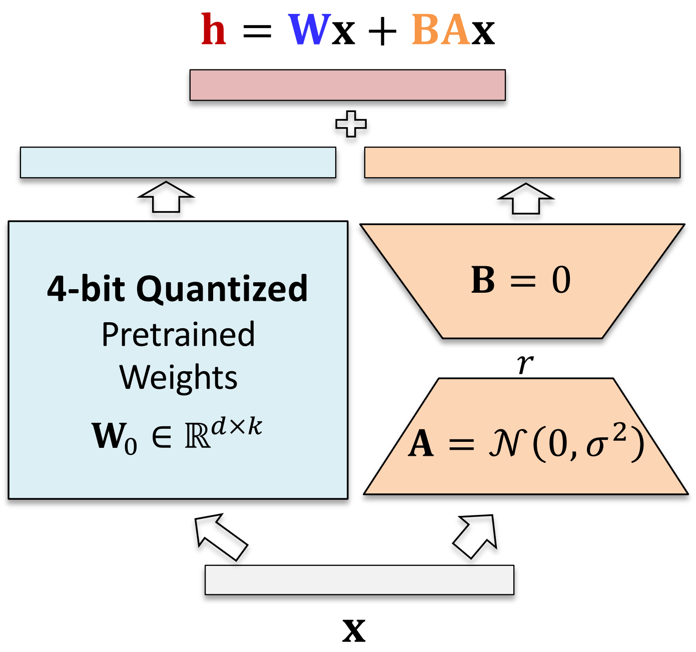
https://arxiv.org/pdf/2305.14314.pdf

QLoRA: LoRA with 4-bit Quantization (2023-05)

- In LoRA, the pretrained weights $\mathbf{W}_0$ **still account for large memory**
  - LLaMA-3-70B model with 32-bit precision requires **820GB of GPU memory.**
- **QLoRA** (**Quantized** LoRA) integrates two concepts:

1. **LoRA** (a technique for efficient language model tuning)
2. **Model Quantization** (a method for reducing model size and memory usage)

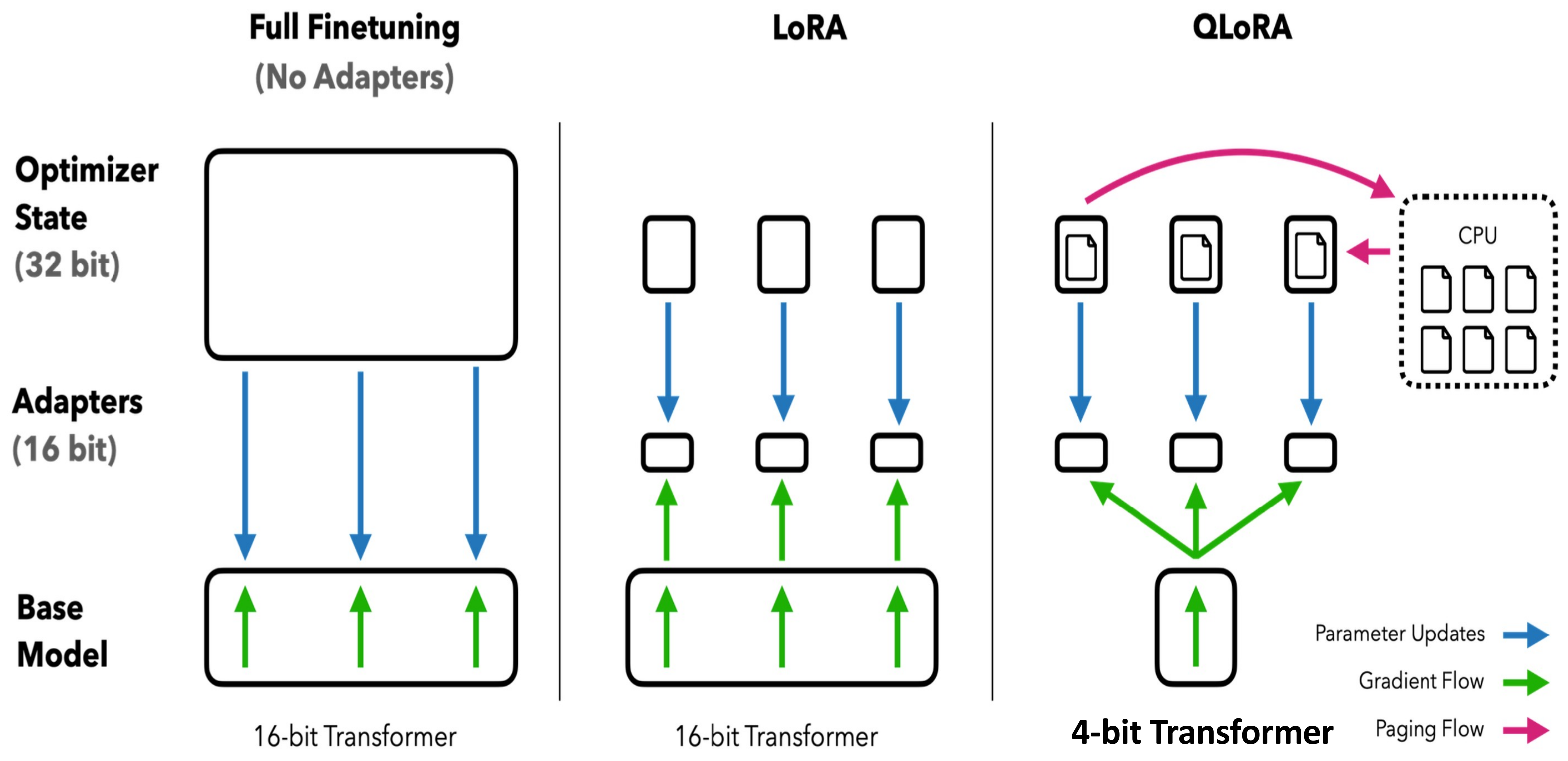
- This involves converting the model's parameters from full-precision 32-bit floating-point numbers (FP32) to lower-precision **4-bit floating-point numbers (NF4) with double quantization**.
- This specialized format significantly reduces the memory footprint, enabling fine-tuning of language models on resource-constrained devices, such as a single GPU machine.



$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$
4-bit Quantized
Pretrained
Weights
$\mathbf{W}_0 \in \mathbb{R}^{d \times k}$
$\mathbf{B} = 0$
r
$\mathbf{A} = \mathcal{N}(0, \sigma^2)$
$\mathbf{x}$

https://arxiv.org/pdf/2305.14314.pdf

Full Fine-Tuning vs LoRA vs QLoRA



QLoRA improves over LoRA by quantizing the transformer model to **4-bit precision** and using **paged optimizers** like AdamW to handle memory spikes

https://arxiv.org/pdf/2305.14314.pdf

Innovations of QLoRA

1. **NF4 (4-bit NormalFloat)**:
   - A specialized 4-bit floating-point format that normalizes weight values to the range [-1, 1] before quantization, allowing for a more accurate representation of the weight distribution and outperforming other 4-bit quantization techniques.
2. **Double Quantization (DQ)**:
   - A nested quantization technique that combines NF4 with further compression of quantization constants to an 8-bit format, resulting in significant memory savings (around 3GB for massive models like LLaMA-65B).
3. **Page Optimizers**:
   - A technique that optimizes memory access patterns, reducing memory usage and improving model performance (not described in detail in the provided text, but mentioned as one of the innovations of QLoRA).

Block-wise k-bit Quantization

- Quantization is the process of discretizing an input from a representation that holds more information to a representation with less information.
- It often means taking a data type with more bits and converting it to fewer bits, for example from 32-bit floats to 8-bit Integers.
- To ensure that the entire range of the low-bit data type is used, the input data type is commonly rescaled into the target data type range through normalization by the absolute maximum of the input elements, which are usually structured as a tensor.
- For example, quantizing a 32-bit Floating Point (FP32) tensor into a Int8 tensor with range [-127, 127]:

$$\mathbf{X}^{\text{Int8}} = \text{round}\left(\frac{127}{\text{absmax}(\mathbf{X}^{\text{FP32}})}, \mathbf{X}^{\text{Int8}}\right) = \text{round}\left(c^{\text{FP32}}, \mathbf{X}^{\text{Int32}}\right)$$

where *c* is the quantization constant or quantization scale.

- Dequantization is the inverse:

$$\text{dequant}\left(c^{\text{FP32}}, \mathbf{X}^{\text{Int32}}\right) = \frac{\mathbf{X}^{\text{Int8}}}{c^{\text{FP32}}} = \mathbf{X}^{\text{Int32}}$$

QLoRA

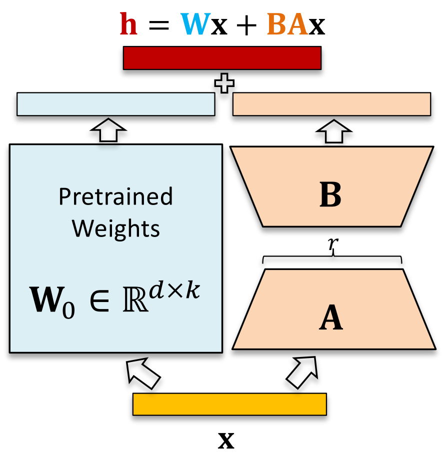
- Using the components described above, we define QLoRA for a single linear layer in the quantized base model with a single LoRA adapter as follows:

$$\mathbf{Y}^{\mathrm{BF16}} = \mathbf{X}^{\mathrm{BF16}} \mathrm{doubleDequant}\left(c_1^{\mathrm{FP32}}, c_2^{\mathrm{k-bit}}, \boxed{\mathbf{W}^{\mathrm{NF4}}}\right) + \mathbf{X}^{\mathrm{BF16}} \boxed{\mathbf{L}_1^{\mathrm{BF16}} \mathbf{L}_2^{\mathrm{BF16}}}$$

where $\mathrm{dequant}(\cdot)$ is defined as:

$$\mathrm{doubleDequant}\left(c_1^{\mathrm{FP32}}, c_2^{\mathrm{k-bit}}, \mathbf{W}^{\mathrm{NF4}}\right) = \mathrm{dequant}\left(\mathrm{dequant}\left(c^{\mathrm{FP32}}, \mathbf{X}^{\mathrm{Int32}}\right), \mathbf{W}^{4\mathrm{bit}}\right) = \frac{\mathbf{X}^{\mathrm{Int8}}}{c^{\mathrm{FP32}}} = \mathbf{W}^{\mathrm{BF16}}$$

- We use NF4 for **W** and FP8 for c_1.
- We use a block-size of 64 for **W** for higher quantization precision and a block-size of 256 for c_2 to conserve memory.



$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$
Pretrained Weights
$\mathbf{W}_0 \in \mathbb{R}^{d \times k}$
B
r
A
x

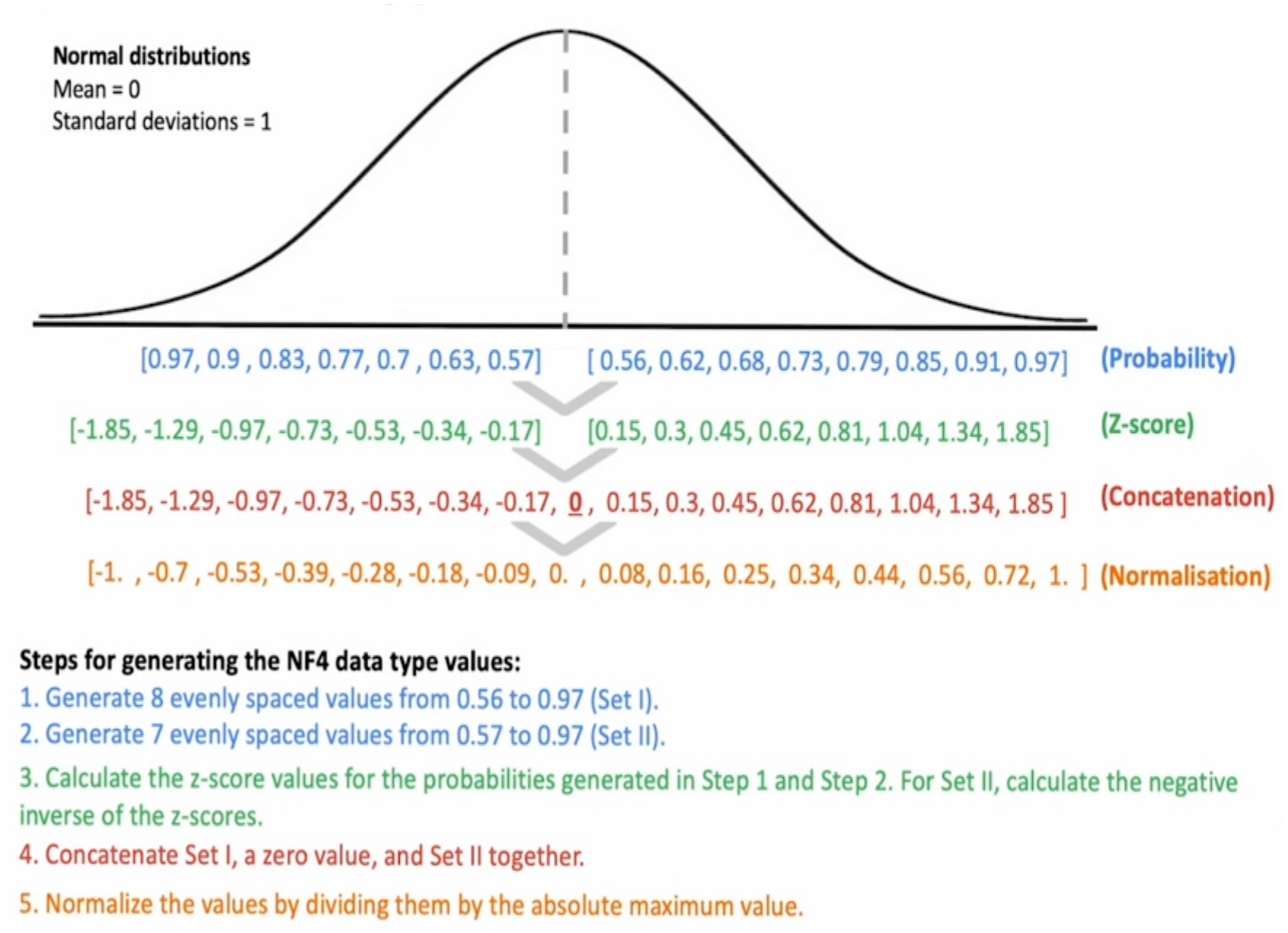
4-bit NormalFloat (NF4)

- According to the QLoRA paper, pre-trained parameters are generally in accordance with a zero-centered normal distribution with a standard deviation of σ. We can scale σ to transform all weights into a single fixed distribution that fully adapts to the data range specified by QLoRA.
- Motivated by this, QLoRA calculates the values of qj based on the quantiles of the normal distribution.
- The current problem is how to calculate 16 quantiles:

$$q_1, \dots, q_{16} \in [-1,1]$$

4-bit NormalFloat (NF4)

- NF4 is an **information-theoretically optimal data type** for normal distributions



[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)**

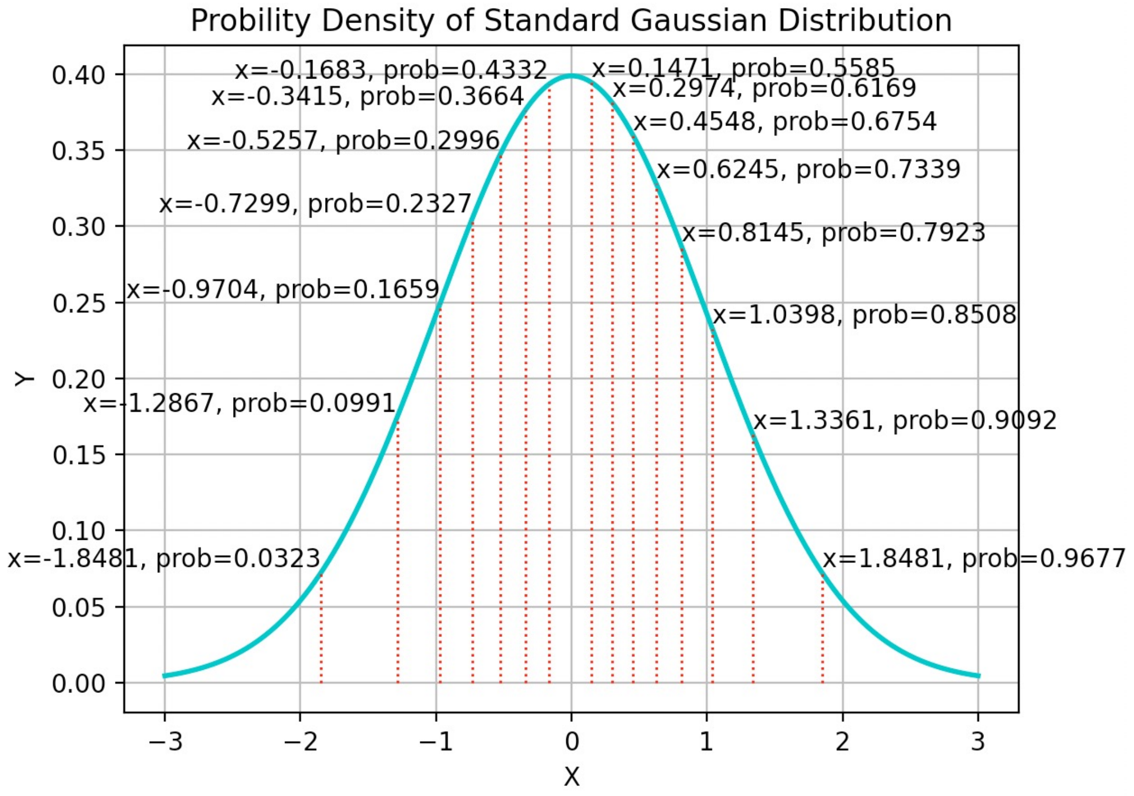
[-1.85, -1.29, -0.97, -0.73, -0.53, -0.34, -0.17] [0.15, 0.3, 0.45, 0.62, 0.81, 1.04, 1.34, 1.85] **(Z-score)**

[-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)**

Steps for generating the NF4 data type values:

1. Generate 8 evenly spaced values from 0.56 to 0.97 (Set I).
2. Generate 7 evenly spaced values from 0.57 to 0.97 (Set II).
3. Calculate the z-score values for the probabilities generated in Step 1 and Step 2. For Set II, calculate the negative inverse of the z-scores.
4. Concatenate Set I, a zero value, and Set II together.
5. Normalize the values by dividing them by the absolute maximum value.



https://ai.plainenglish.io/qlora-key-quantization-and-fine-tuning-techniques-in-the-era-of-large-language-models-0fa05a961d27

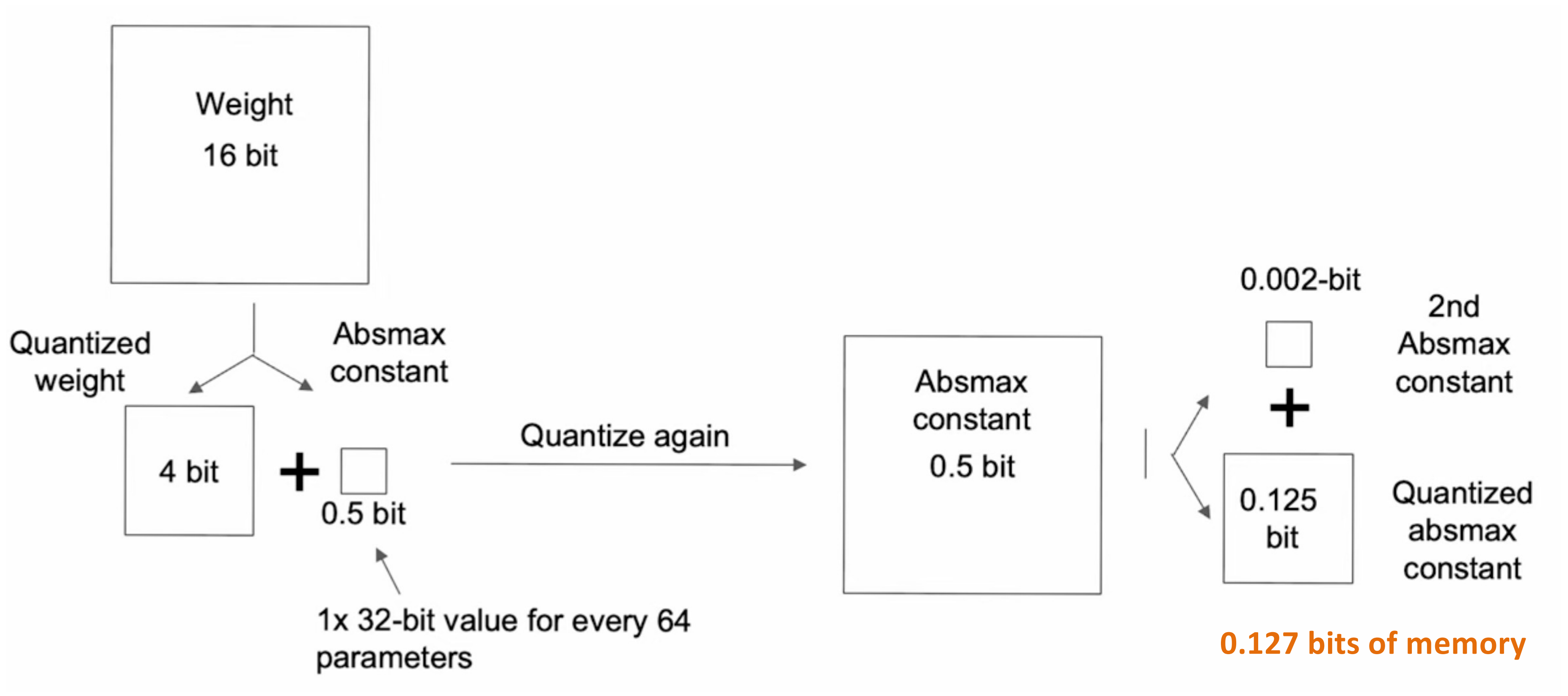
Double Quantization

- **Block-wise Quantization**
  - We know that the essence of quantization is to map values from a larger range to a smaller range. We can use a constant c to proportionally reduce the values. In this way, we can easily use the same constant c to dequantize the quantized values back to their original (approximate) form.
  - However, if our data contains outliers, this will affect the selection of c and cause other values to collapse within a small range. Block-wise provides a solution to this by quantizing one block at a time, with each block using its own independent quantization constant c.
- **Since quantization constants are typically stored as FP32, the memory usage can become significant when there are a large number of blocks.**

The Approach of QLoRA

- QLoRA divides the parameters into **blocks of size 64**.
  - Each block calculates a quantization constant, denoted as c.
- QLoRA further quantizes the quantization constants into FP8 using Double Quant, with a **block size of 256**.
- This further reduces the memory consumption.
  - Before Double Quant:
    - Quantizing each parameter requires an additional 32/64 = **0.5 bits of memory**.
  - After Double Quant:
    - Quantizing each parameter only requires an additional 8/64 + 32 / (64*256) = **0.127 bits of memory**.

Double Quantization: Reduce absmax constant size



Paged Optimizer: Prevent Memory Spikes

- Page-by-page transfers of memory from CPU <=> GPU as needed
  - Lazy and does not need to be managed (no offloading, everything is automatic).
- **The Page Optimizer mechanism** allows for transferring the optimizer to memory when GPU memory is limited.
  - It can be loaded back when the optimizer state needs to be updated.
  - It is said to effectively reduce the peak occupancy of GPU memory.
- The paper of QLoRA states that this mechanism is necessary to train a model with 33 billion parameters on a 24GB GPU.
- This mechanism can be easily configured by setting the parameters of Training Arguments:
  - `optim = 'paged_adamw_32bit'`

Paged Optimizer: Prevent Memory Spikes

- Paged Optimizer works like this:
  1. A large mini-batch (long sequence length) uses more GPU memory than available
  2. Paging engine evicted optimizer state to CPU
  3. During optimizer step all optimizer states are prefetched to the GPU
  4. Do an optimizer step
  5. Continue to process everything on the GPU as long as the mini-batch does not cause an eviction

How does QLoRA reduce memory to 14GB?

- Below is the calculation to determine the memory requirements for fine-tuning **LLaMA3–8B** with **QLoRA**.
  - **Memory requirement for loading the 4-bit quantized model**:
    - The LLaMA3-**8B** base model has about 8 billion parameters, and each parameter is quantized to 4 bits (0.5 bytes). Hence, loading the model would take about **4GB** (8 billion parameters × 0.5 bytes).
  - **Memory requirement per trainable parameter consists of**:
    - Weight: 0.5 bytes
    - LoRA parameters: 2 bytes
    - AdamW optimizer states: 2 bytes
    - Gradients (always in fp32): 4 bytes
    - Therefore, the memory per trainable parameter is **8.5 bytes** (≈ 0.5 + 2 + 2 + 4)
  - **Total memory requirement for trainable parameters**:
    - LoRA results in an average of 0.4-0.7% trainable parameters, assuming that there are 0.6% of trainable parameters
    - The total trainable parameters memory :

      Memory per parameter * parameters = 8.5 bytes * 48 million (0.6% of **8B** parameters) ≈ **0.408 GB**

How does QLoRA reduce memory to 14GB?

- **Total memory requirement for LLaMA3-8B QLoRA Training**: The total memory requirement for QLoRA training is around **4.1GB**, which includes the memory for the base model and the memory for trainable parameters ≈ 0.408 GB, resulting in a total training memory requirement of about **≈ 4–5 GB** (depending on the number of trainable parameters).
- **Memory required for Inference**: If we load the base model in 16-bit precision and merge the LoRA weights of the fine-tuned model, **we would at-most use 14 GB of GPU memory for a sequence length of 2048**. This memory cost is derived from loading the model in float16 precision and includes activations, temporary variables and hidden states, which are always in full-precision (float32) format and depend on many factors including sequence length, hidden size and batch size.
- **Total memory requirements**: So, the total memory requirement for QLoRA training with a 4-bit base model and mixed-precision mode, including loading the 32-bit model for inference, would be almost **≈ 14 GB** depending on the sequence length.
- Thus, we can see that using quantization techniques like QLoRA along with PEFT can significantly reduce memory requirements by up to 90%, thereby making fine tuning more accessible and affordable!

How does QLoRA reduce memory to 14GB?

- Below is the calculation to determine the memory requirements for fine-tuning **LLaMA3–8B** with **QLoRA**.
  - **Memory requirement for loading the 4-bit quantized model**:
    - The LLaMA3-**8B** base model has about 8 billion parameters, and each parameter is quantized to 4 bits (0.5 bytes). Hence, loading the model would take about **4GB** (8 billion parameters × 0.5 bytes).
  - **Memory requirement per trainable parameter consists of**:
    - Weight: 0.5 bytes
    - LoRA parameters: 2 bytes
    - AdamW optimizer states: 2 bytes
    - Gradients (always in fp32): 4 bytes
    - Therefore, the memory per trainable parameter is **8.5 bytes** (≈ 0.5 + 2 + 2 + 4)
  - **Total memory requirement for trainable parameters**:
    - LoRA results in an average of 0.4-0.7% trainable parameters, assuming that there are 0.6% of trainable parameters
    - The total trainable parameters memory :

      Memory per parameter * parameters = 8.5 bytes * 48 million (0.6% of **8B** parameters) ≈ **0.408 GB**

QLoRA

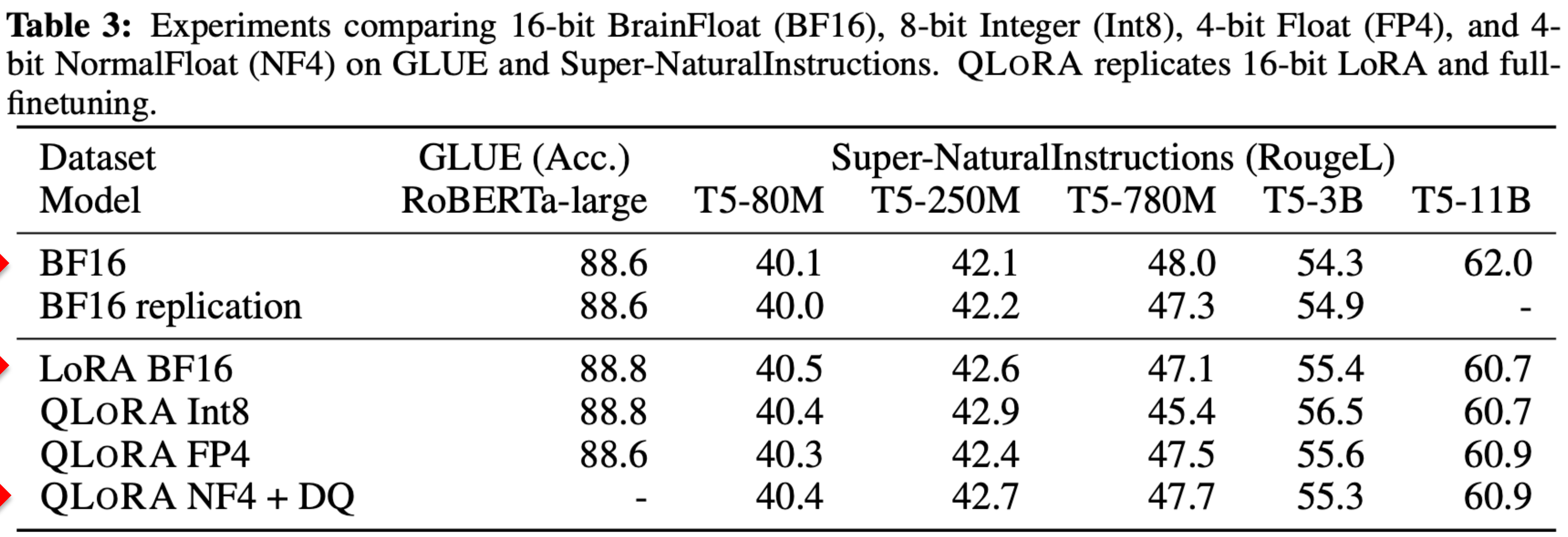
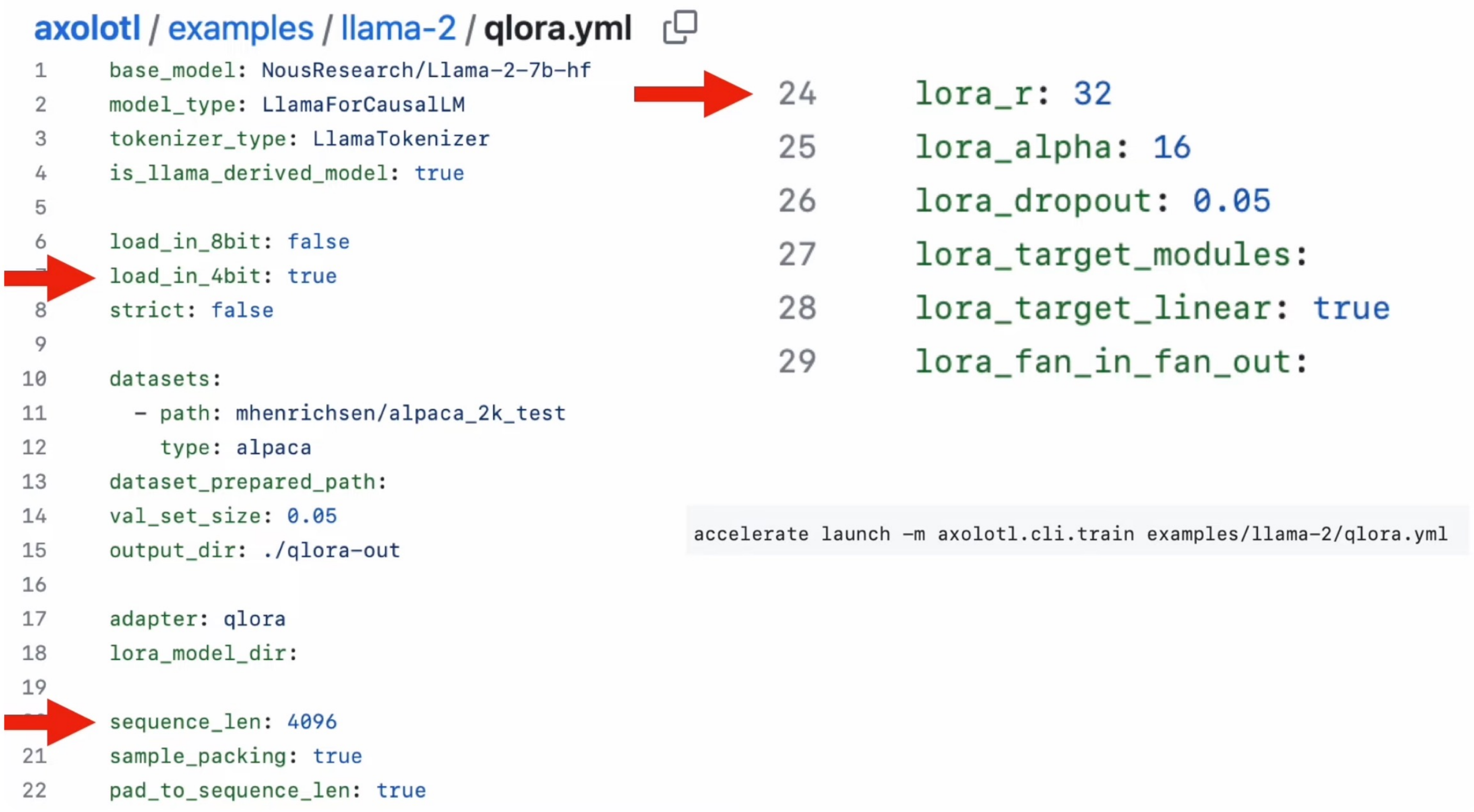


Table 3: Experiments comparing 16-bit BrainFloat (BF16), 8-bit Integer (Int8), 4-bit Float (FP4), and 4-bit NormalFloat (NF4) on GLUE and Super-NaturalInstructions. QLORA replicates 16-bit LoRA and full-finetuning.

Dataset	GLUE (Acc.)	Super-NaturalInstructions (RougeL)				
Model	RoBERTa-large	T5-80M	T5-250M	T5-780M	T5-3B	T5-11B
BF16	88.6	40.1	42.1	48.0	54.3	62.0
BF16 replication	88.6	40.0	42.2	47.3	54.9	-
LoRA BF16	88.8	40.5	42.6	47.1	55.4	60.7
QLORA Int8	88.8	40.4	42.9	45.4	56.5	60.7
QLORA FP4	88.6	40.3	42.4	47.5	55.6	60.9
QLORA NF4 + DQ	-	40.4	42.7	47.7	55.3	60.9



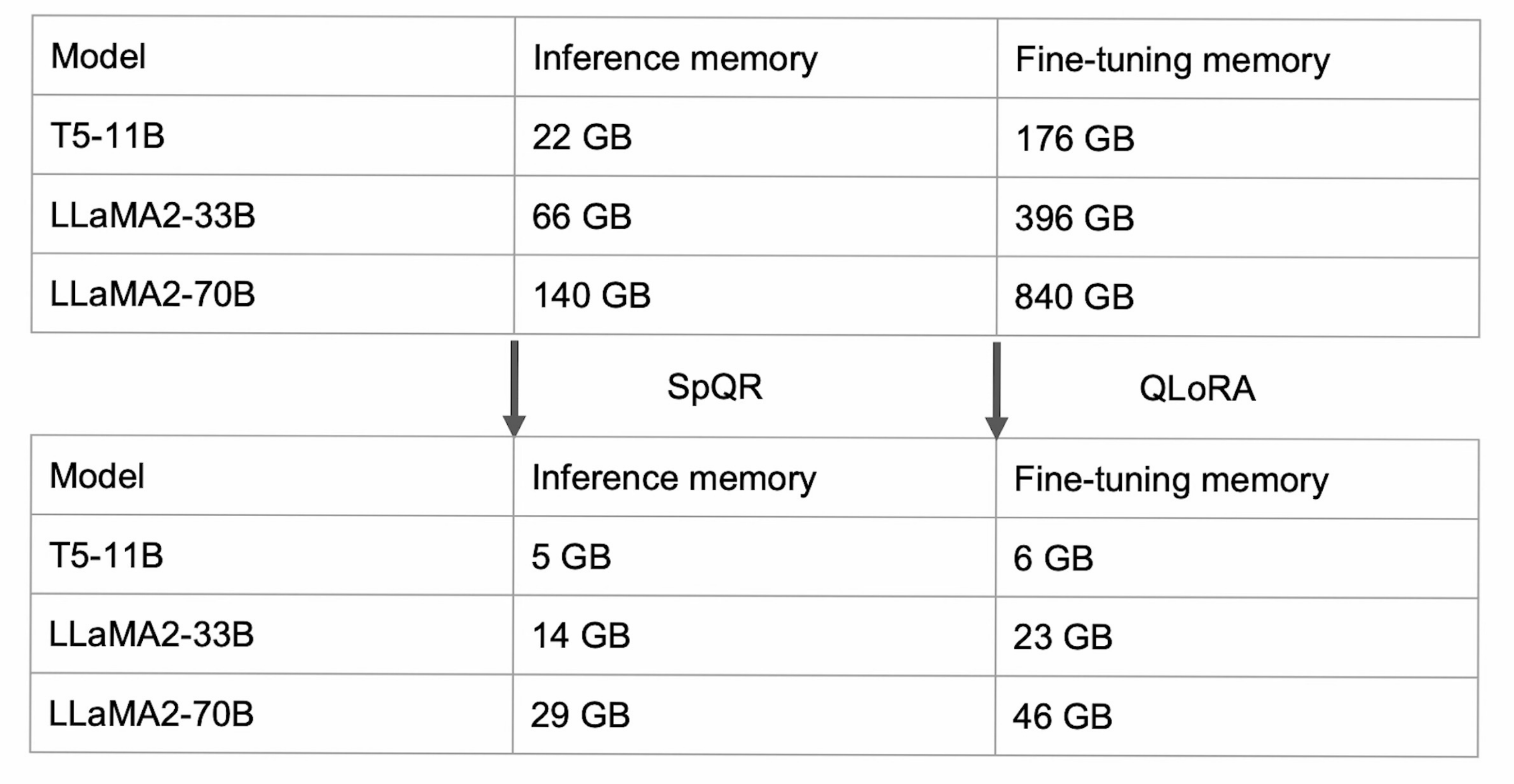
axolotl / examples / llama-2 / qlora.yml

```yaml
base_model: NousResearch/Llama-2-7b-hf
model_type: LlamaForCausalLM
tokenizer_type: LlamaTokenizer
is_llama_derived_model: true

load_in_8bit: false
load_in_4bit: true
strict: false

datasets:
  - path: mhenrichsen/alpaca_2k_test
    type: alpaca
dataset_prepared_path:
val_set_size: 0.05
output_dir: ./qlora-out

adapter: qlora
lora_model_dir:

sequence_len: 4096
sample_packing: true
pad_to_sequence_len: true
```

```yaml
lora_r: 32
lora_alpha: 16
lora_dropout: 0.05
lora_target_modules:
lora_target_linear: true
lora_fan_in_fan_out:
```

```
accelerate launch -m axolotl.cli.train examples/llama-2/qlora.yml
```

Large Models are Not easily accessible



Model	Inference memory	Fine-tuning memory
T5-11B	22 GB	176 GB
LLaMA2-33B	66 GB	396 GB
LLaMA2-70B	140 GB	840 GB

SpQR QLoRA

Model	Inference memory	Fine-tuning memory
T5-11B	5 GB	6 GB
LLaMA2-33B	14 GB	23 GB
LLaMA2-70B	29 GB	46 GB

https://www.youtube.com/watch?v=fQirE9N5q_Y

QLoRA

- QLoRA Hyperarameter settings:
  - Alpha determines the multiplier applied to the weight changes when added to the original weights
    - Scale multiplier = Alpha / Rank
    - Microsoft LoRA repository sets to 2 x Rank
    - QLorA wen with ¼ of Rank (alpha = 16, r = 64)
  - Droput is a percentage that randomly eaves out some weight changes each time to deter overfitting
    - QLoRA paper went with 0.1 for 7B-13B, 0.05 for 33B=65B models
- QLoRA paper has two interesting findings:
  - Training all layers of the network is necessary to match performance of full-parameter fine-tuning
  - Rank may not matter from 8 to 256

https://www.youtube.com/watch?v=t1caDsMzWBk

QLoRA Summary

- **QLoRA** uses **NF4**, **double quantization**, and **paged optimizers** combined with LoRA to replicate 16-bit full finetuning performance at a 17x smaller memory footprint.
- While evaluation is noisy, Guanaco models outperform existing open-source models on the Vicuna benchmark.