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

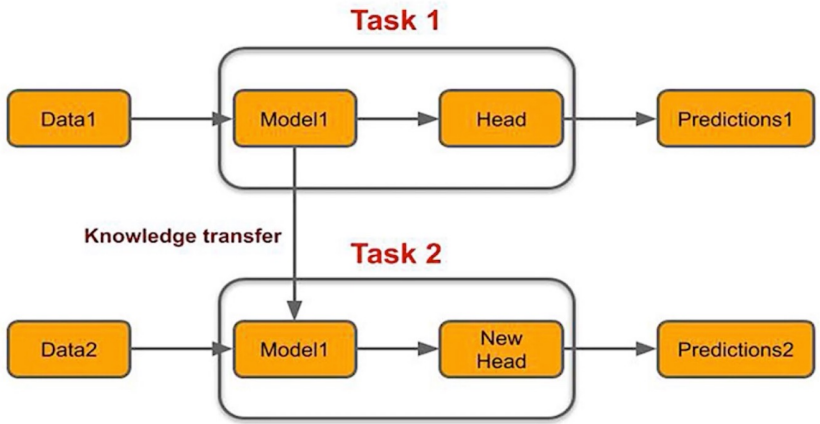
Content

- Review of Finetuning Approaches
- Parameter Efficient Finetuning (PEFT)
  - **Adapter Layer**
  - **LoRA (Low Rank Adaptation)**
  - **QLoRA (Quantized LoRA)**
  - **DoRA** (Optional)
  - **MoRA** (Optional)
  - **SBoRA** and **Multi-SBoRA** (Optional)

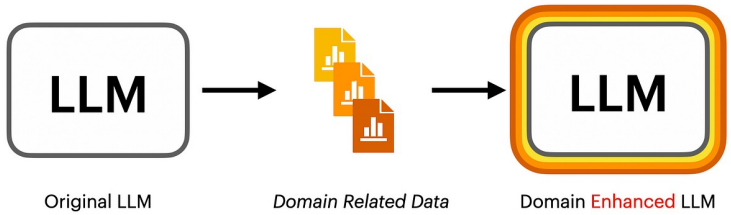
Transfer Learning and Finetuning

- **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



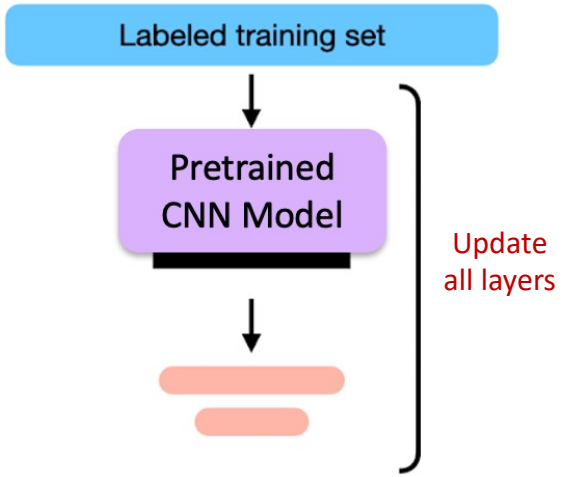
Fine-Tune LLM



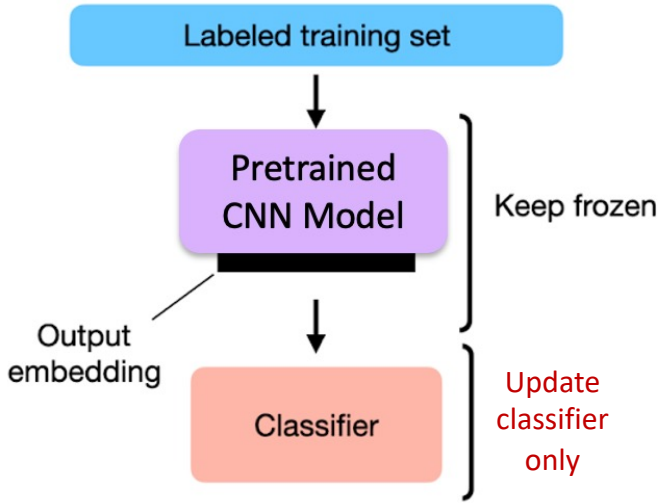
Finetuning of Convolutional Neural Networks (CNNs)

- 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:

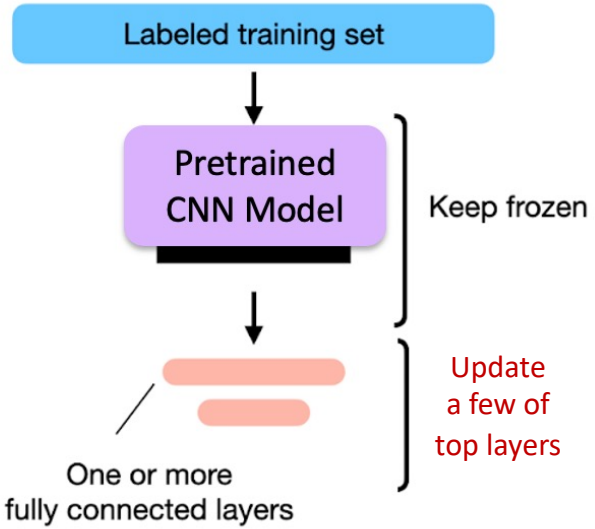
(1) Full Finetuning



(2) Feature-based Approach



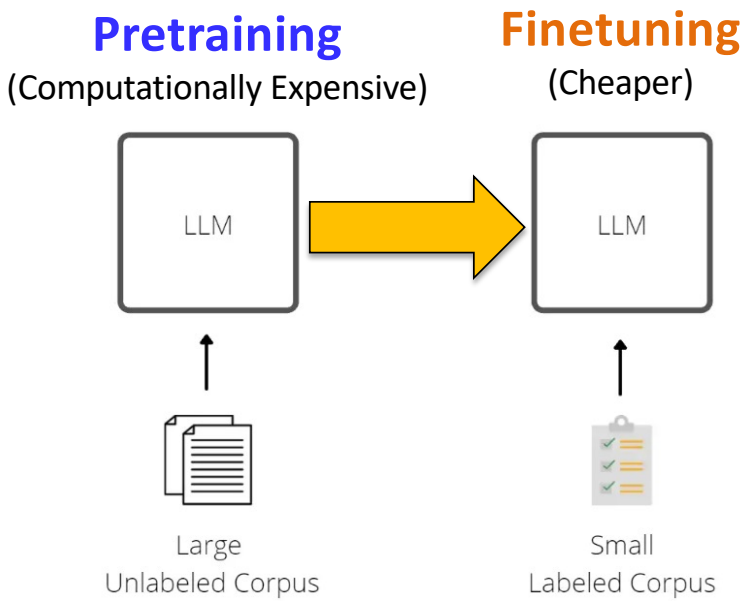
(3) Top-Layer Finetuning



Finetuning of Large Language Models

- In Natural Language Processing (NLP), **finetuning is commonly applied** to pretrained large language models (LLMs) like BERT, GPT-3, which are initially trained using **self-supervised learning** on large-scale unlabeled corpora, with labels generated automatically from the data itself.

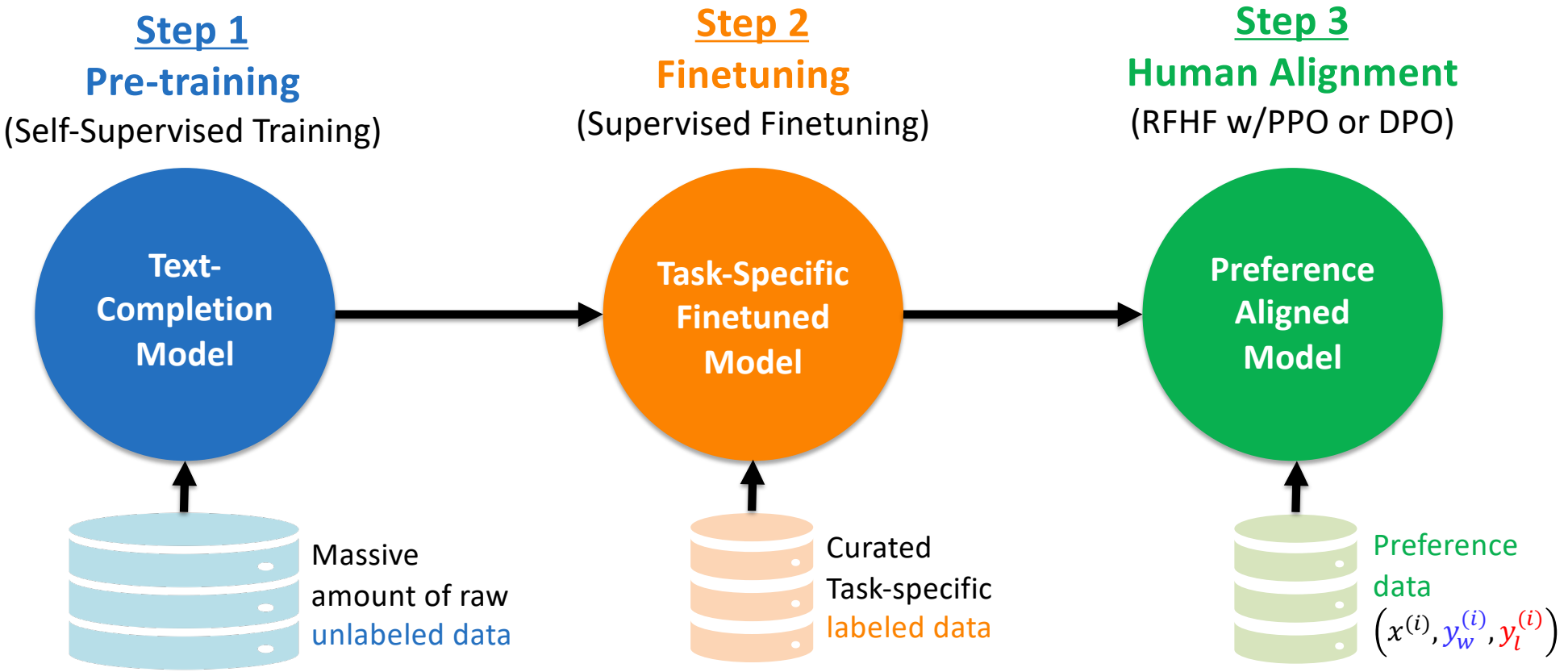
- In most LLM-powered applications, pretrained models are **finetuned on smaller labeled datasets** for downstream tasks, enhancing performance for chatbots, summarization, and more.
- This approach is computationally more efficient than training from scratch.



LLM Finetuning: From General to Specific

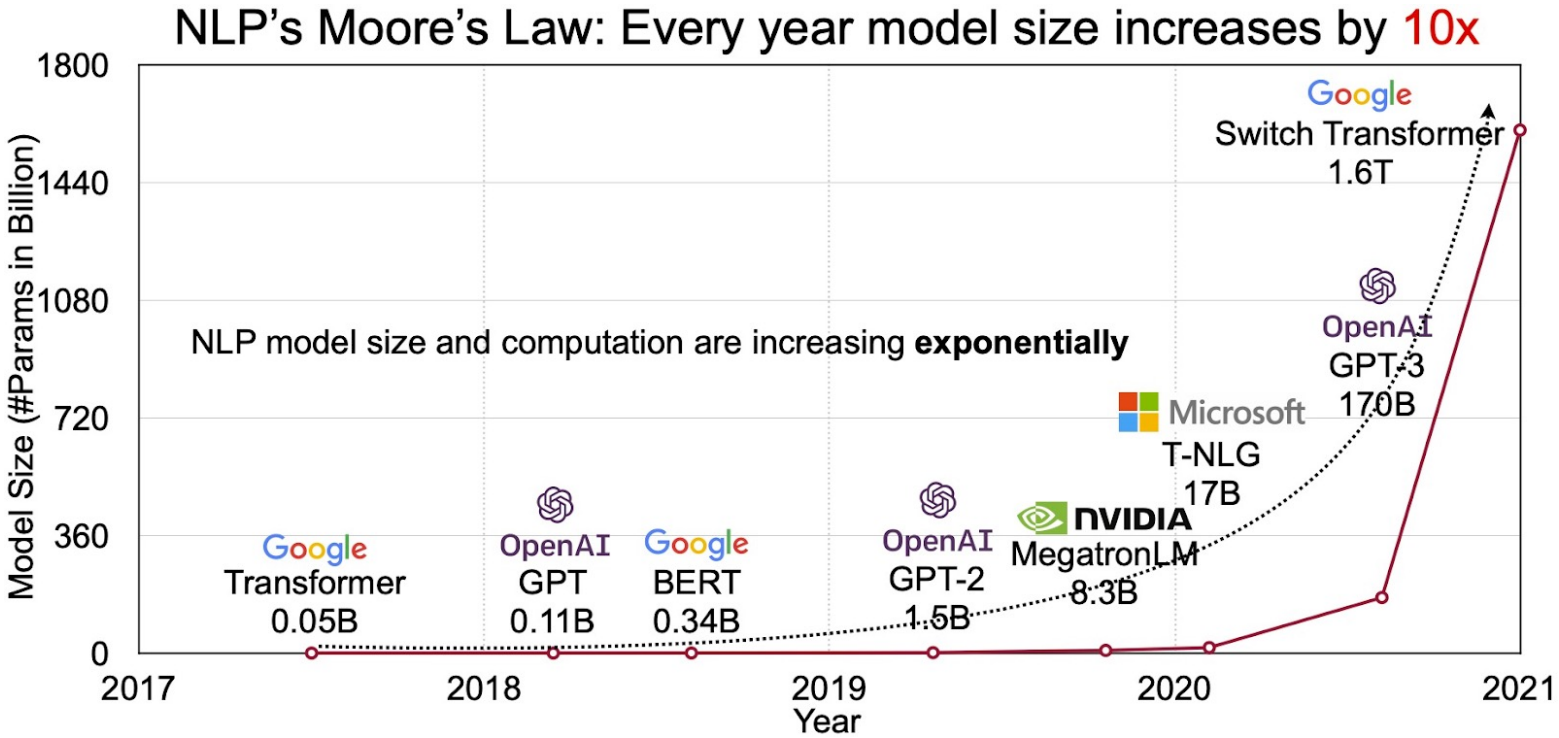
- In the realm of NLP, finetuning of LLMs is a crucial step in transforming general-purpose pretrained models into specialized models tailored to meet the unique demands of specific applications.
- **This process effectively bridges the gap between the generic, pretrained models and the nuanced requirements 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.

LLM Training Pipelines



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**.

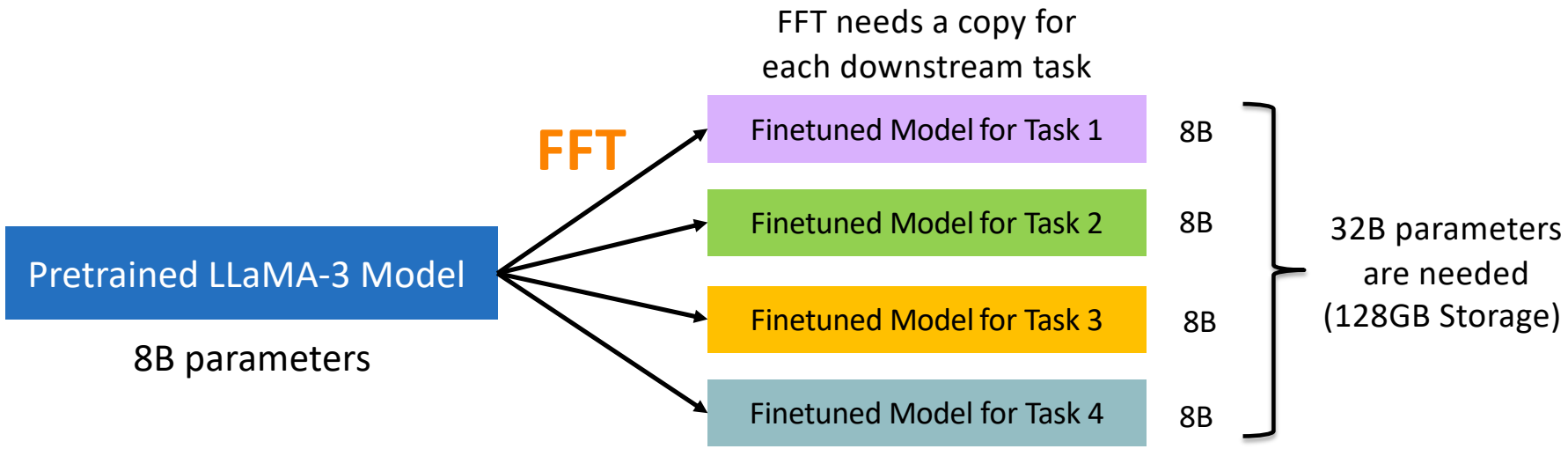
Naively Fine-Tuning LLaMA-3-8B takes 128GB of RAM!

- **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**.

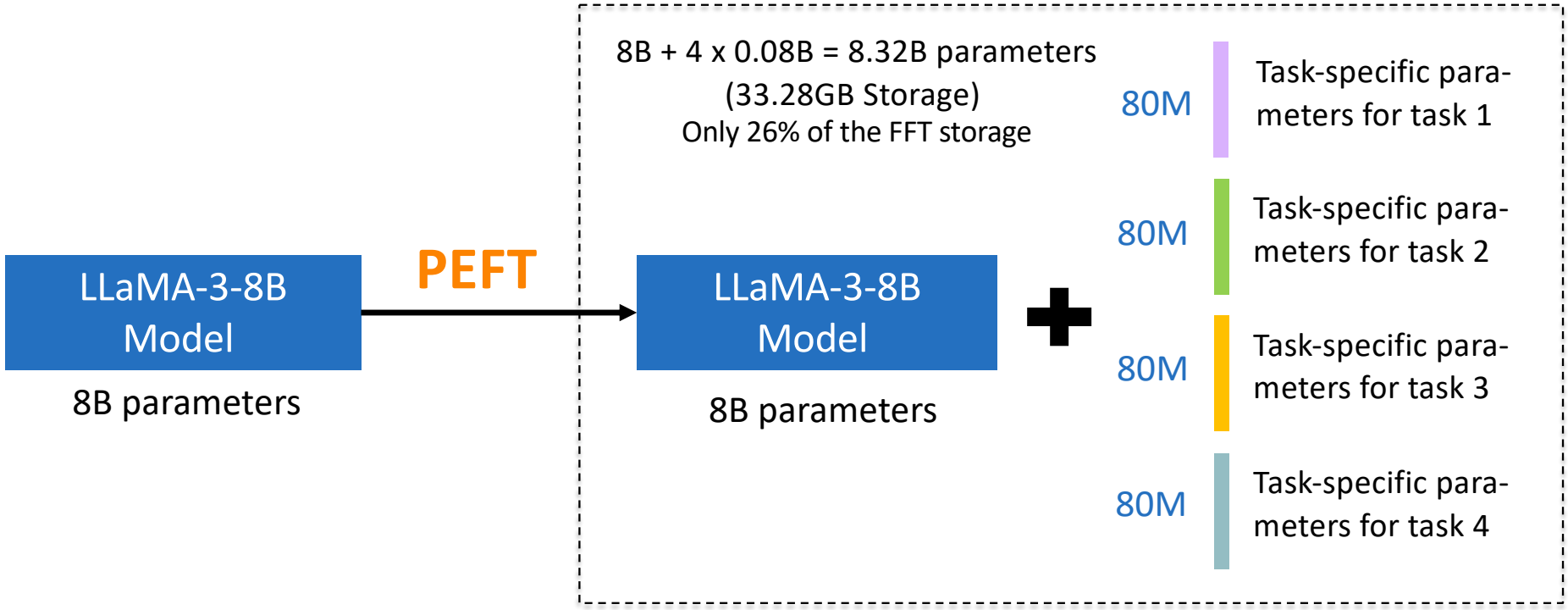


Parameter-Efficient Finetuning (PEFT)

- 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**

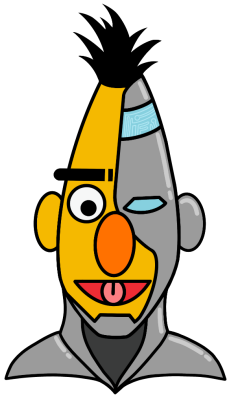
Reduce the Number of Parameters by PEFT

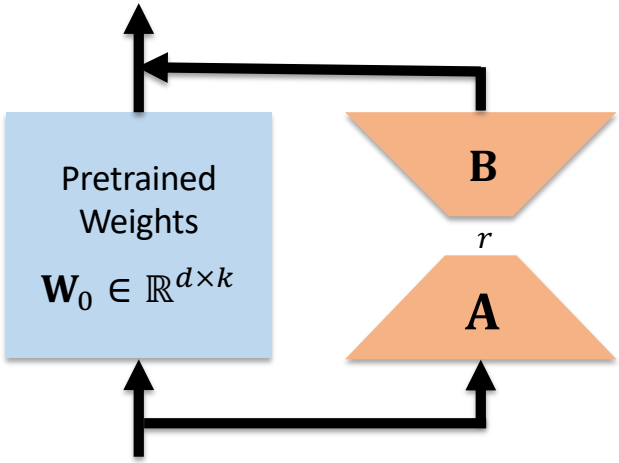
- **PEFT** enables the reuse of pretrained model weights, requiring only a small number of additional task-specific parameters.



PEFT Techniques

1. **Adapter Tuning** (2019) – Add new intermediate modules
2. **Prefix Tuning** (2021) – Add additional prefixes
3. **Prompt Tuning** (2021) – Adapts input prompts
4. **LoRA** (2021) – Low-Rank decomposition
5. **QLoRA** (2023) – Quantized LoRA
6. **DoRA** (2024) – Weight-Decomposed LoRA
7. **MoRa** (2024) – High-Rank Updating
8. **SBoRA** (2024) – Standard Basis LoRA

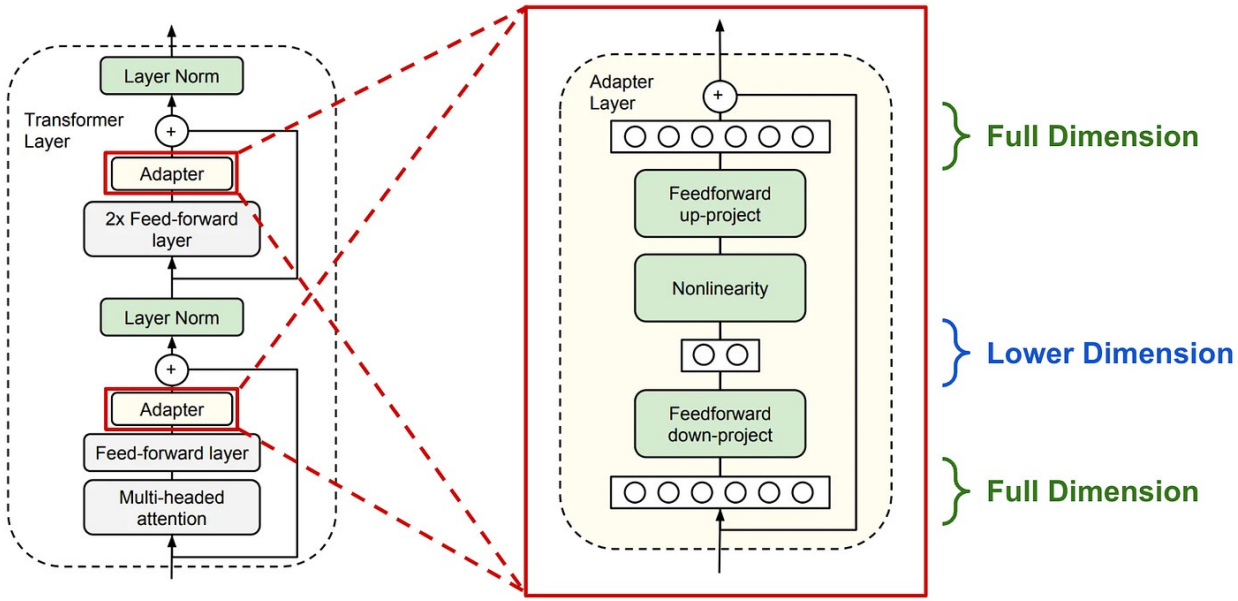




LoRA

Adapter 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.



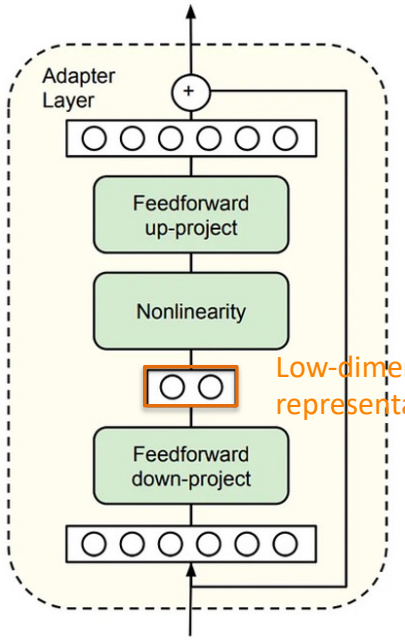
- **The adapter Layers are extra trainable modules** inserted into the existing transformer block that have a small number of parameters and can be finetuned while keeping the weights of the pretrained model fixed.
- As such, each finetuned version of the model only has a small number of task-specific parameters associated with it.

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

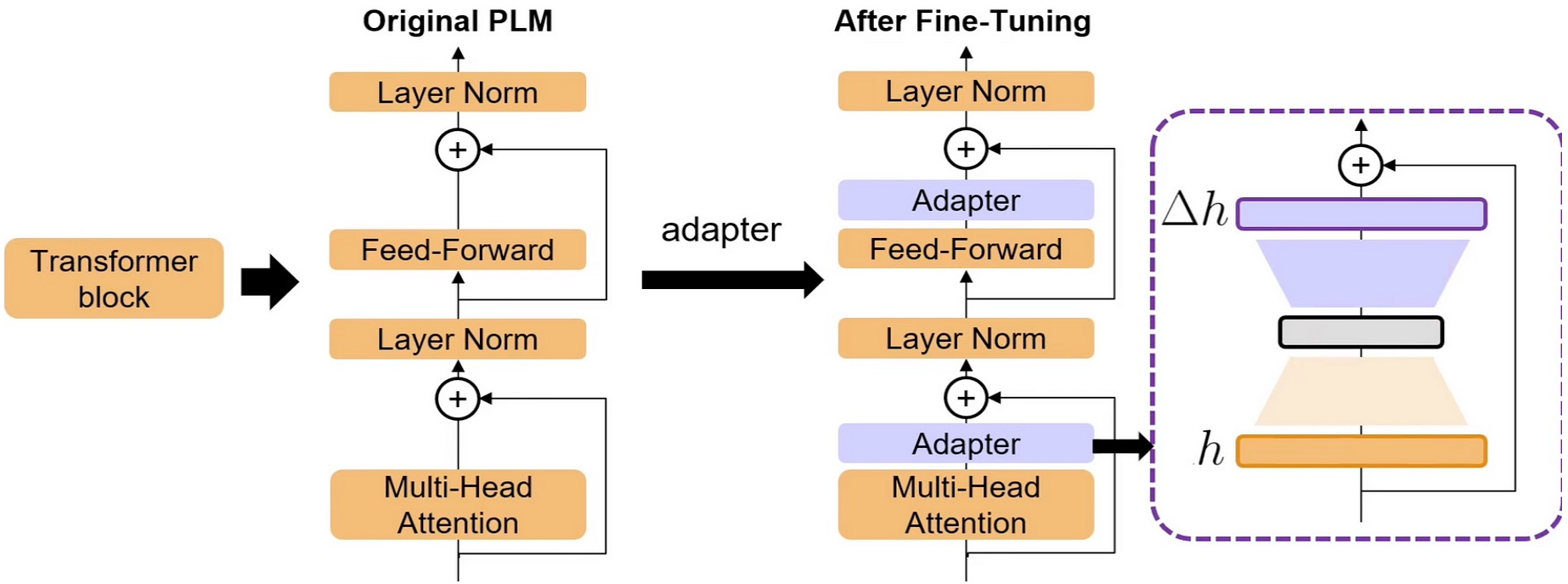
https://cameronrwolfe.substack.com/p/easily-train-a-specialized-llm-peft

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.**



Adapter (He et al. 2022)

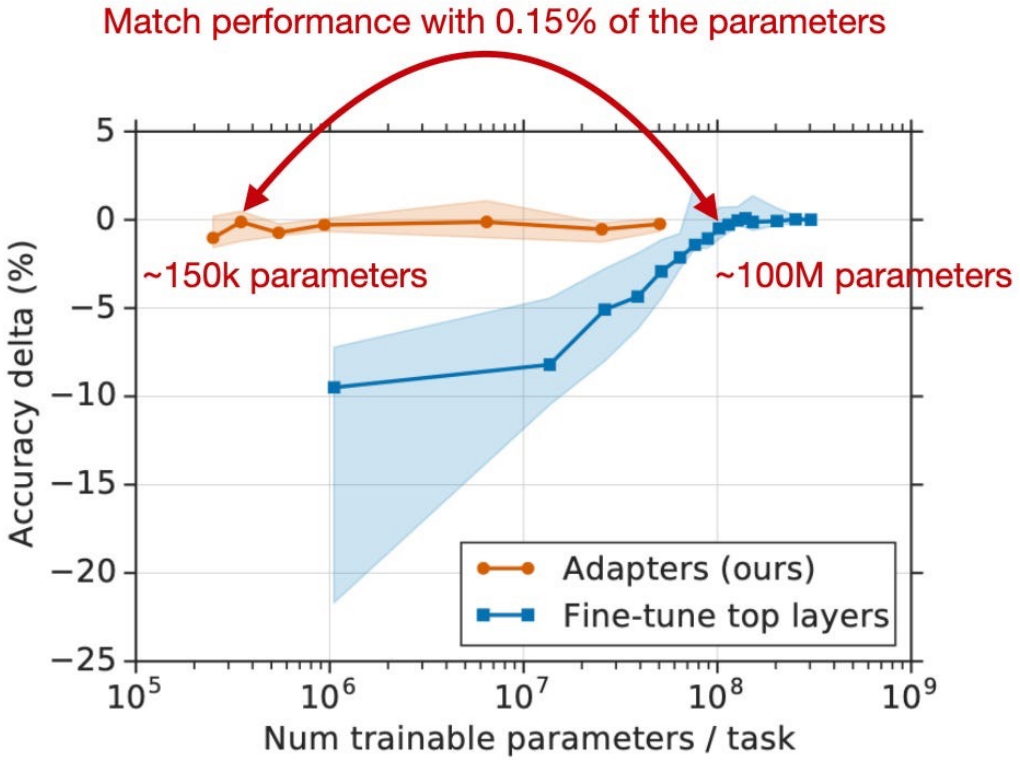


All tasks share the same original PLM; the adapters are task-specific modules => better robustness, storage-efficient

https://www.youtube.com/watch?v=R3jZVKUlSjA

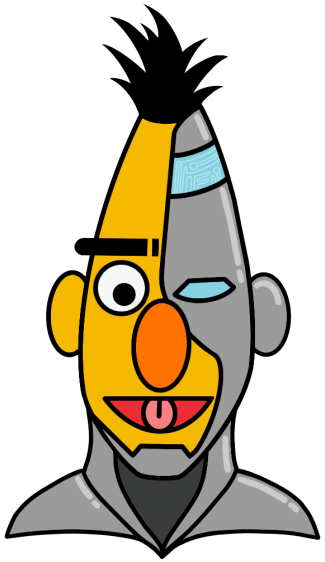
Adapter Performance on Finetuning BERT

1. **Adapter-trained BERT models achieve similar performance to fully finetuned 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.

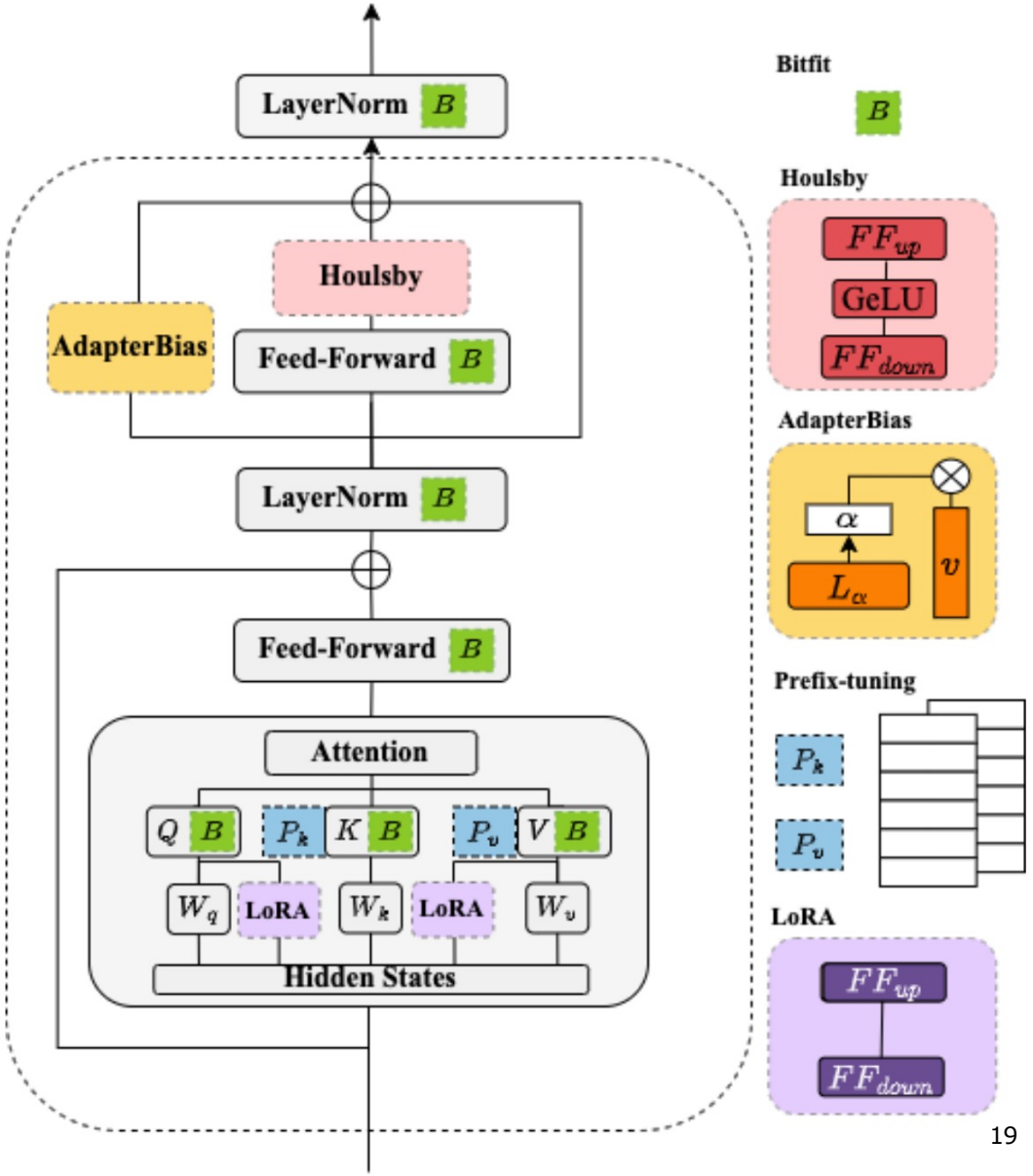


Other Types of Adapter

- https://adapterhub.ml/



https://arxiv.org/abs/2210.06175

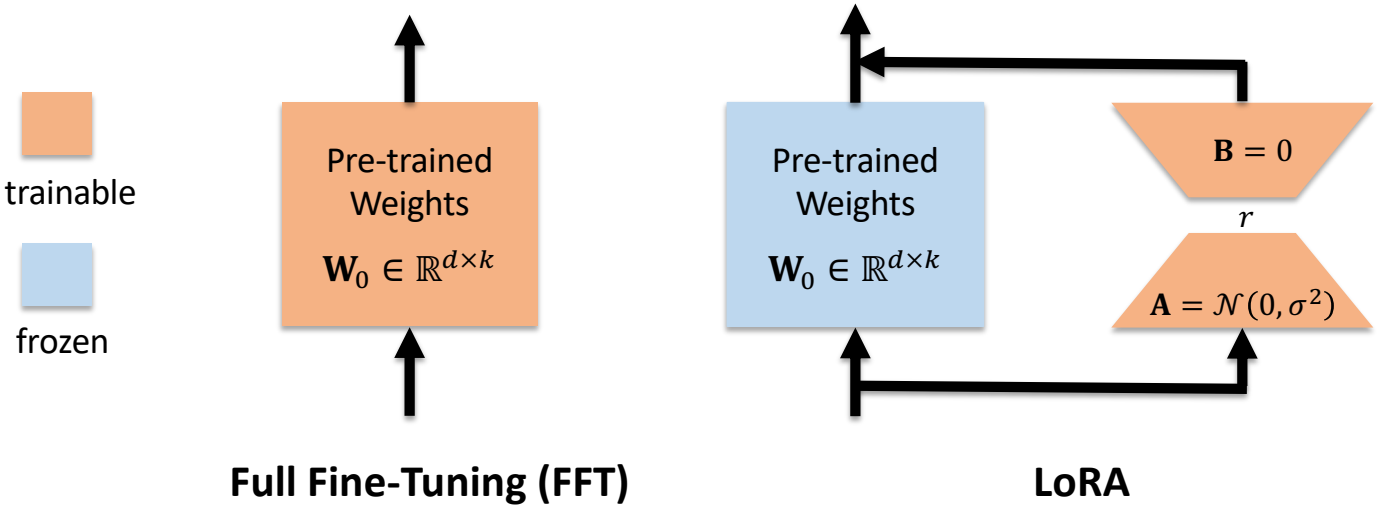


Low-Rank Adaptation (LoRA)

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

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

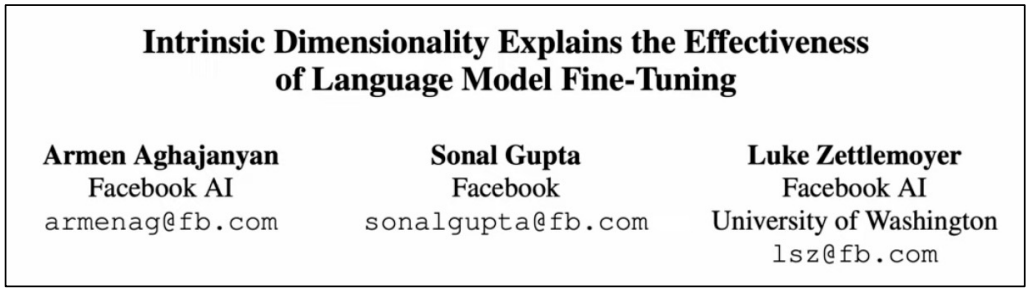
- **Low-Rank Adaptation (LoRA)** is a groundbreaking technique of PEFT for LLMs.
- It introduces a **parallel low-rank adapter** to the weights of linear layers **reducing memory overhead and computational costs during finetuning.**



https://arxiv.org/abs/2106.09685

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

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:

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

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

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

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

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

Rep: Rank Matrix Decomposition

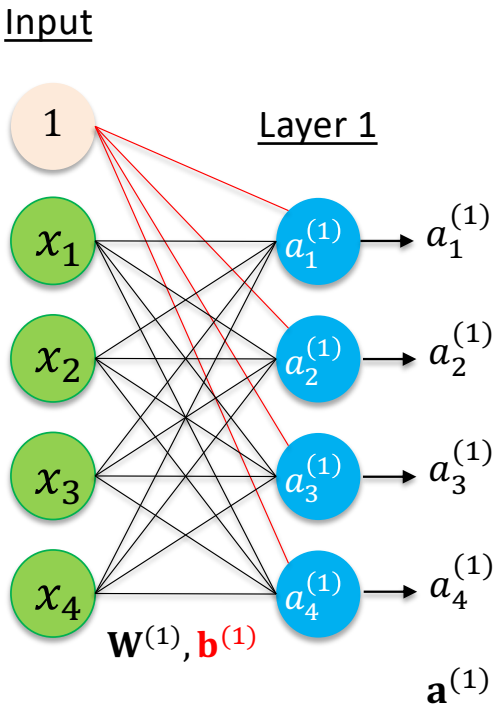
- **Low-Rank Matrices can be Decomposed into Low-Dimensional Matrices**
- A low-rank matrix can be decomposed into the product of two low-dimensional matrices. For instance, a rank-1 3x3 matrix can be decomposed as follows:

$$\underset{3\times3}{\begin{bmatrix}2 & 10 & 1\\ 4 & 20 & 2\\ 6 & 30 & 3\end{bmatrix}} = \underset{3\times1}{\begin{bmatrix}1\\ 2\\ 3\end{bmatrix}} \times \underset{1\times3}{\begin{bmatrix}2 & 20 & 30\end{bmatrix}}$$

  - 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: $n^2 \Rightarrow 2 \cdot r \cdot n$

Recap: Matrix Representation of FFN

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



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

- **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.

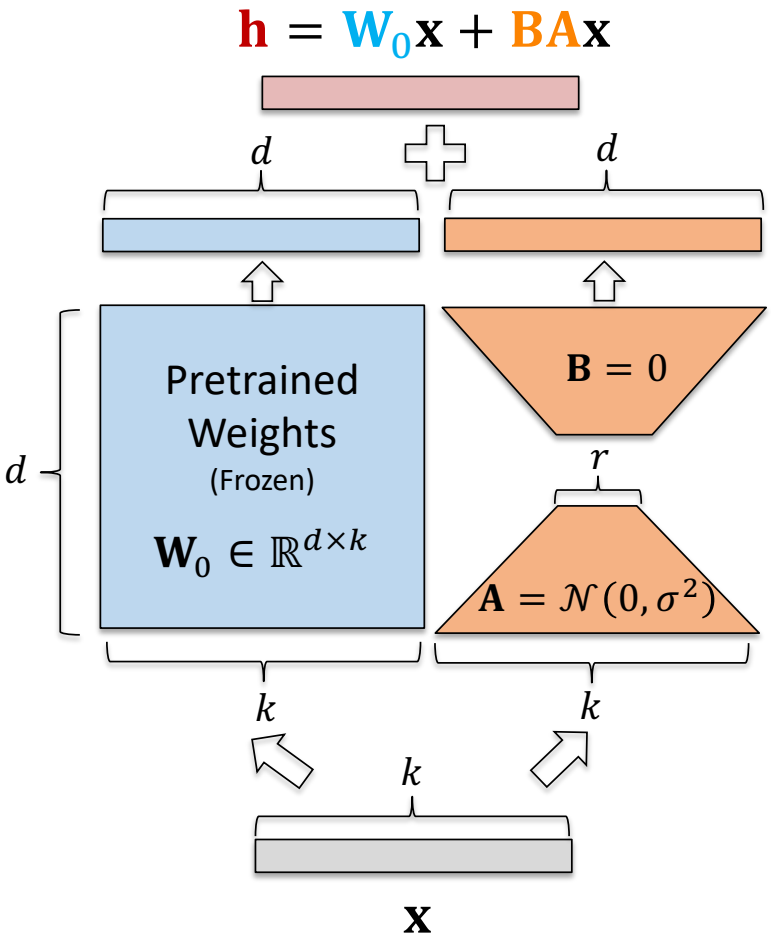
https://arxiv.org/pdf/2012.13255.pdf

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**

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

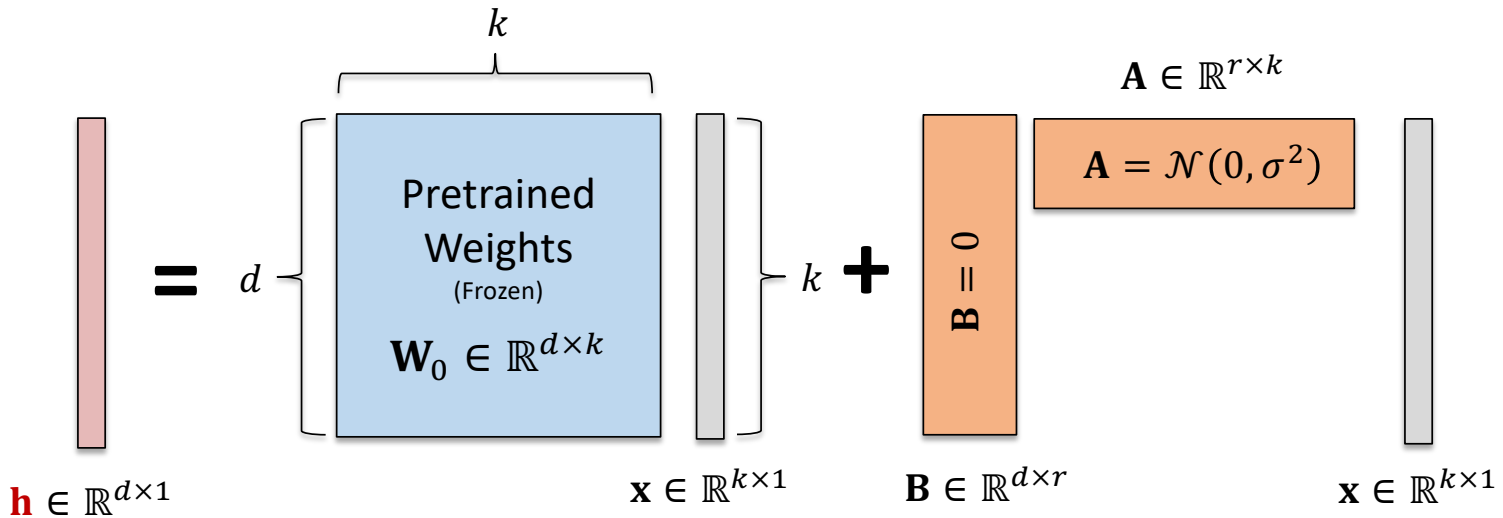
where $\Delta\mathbf{W} = \mathbf{BA}$ is a low-rank matrix



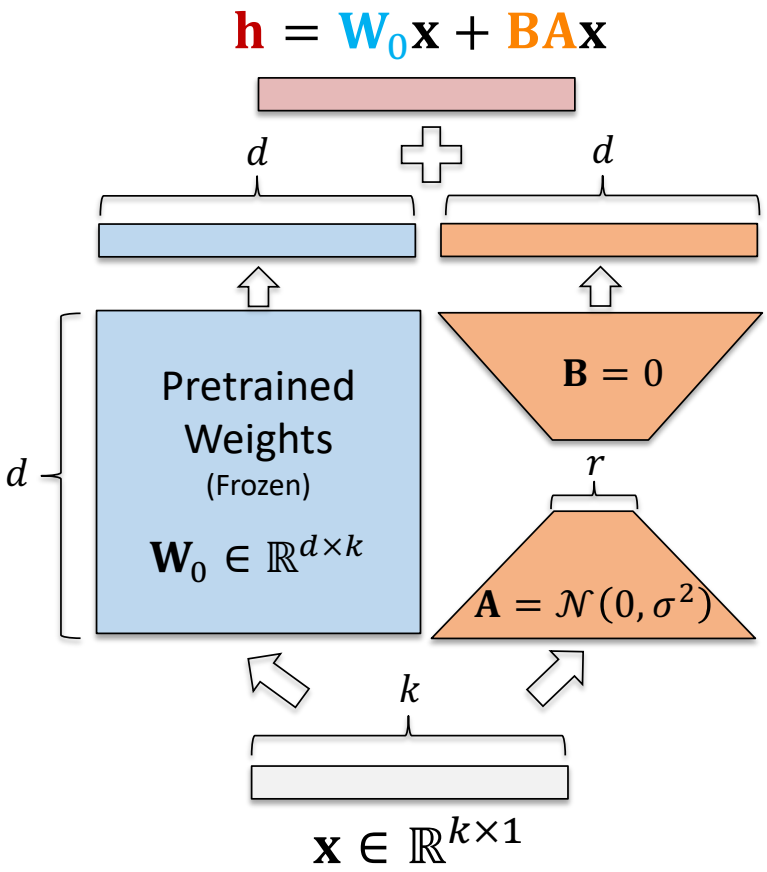
https://arxiv.org/abs/2106.09685

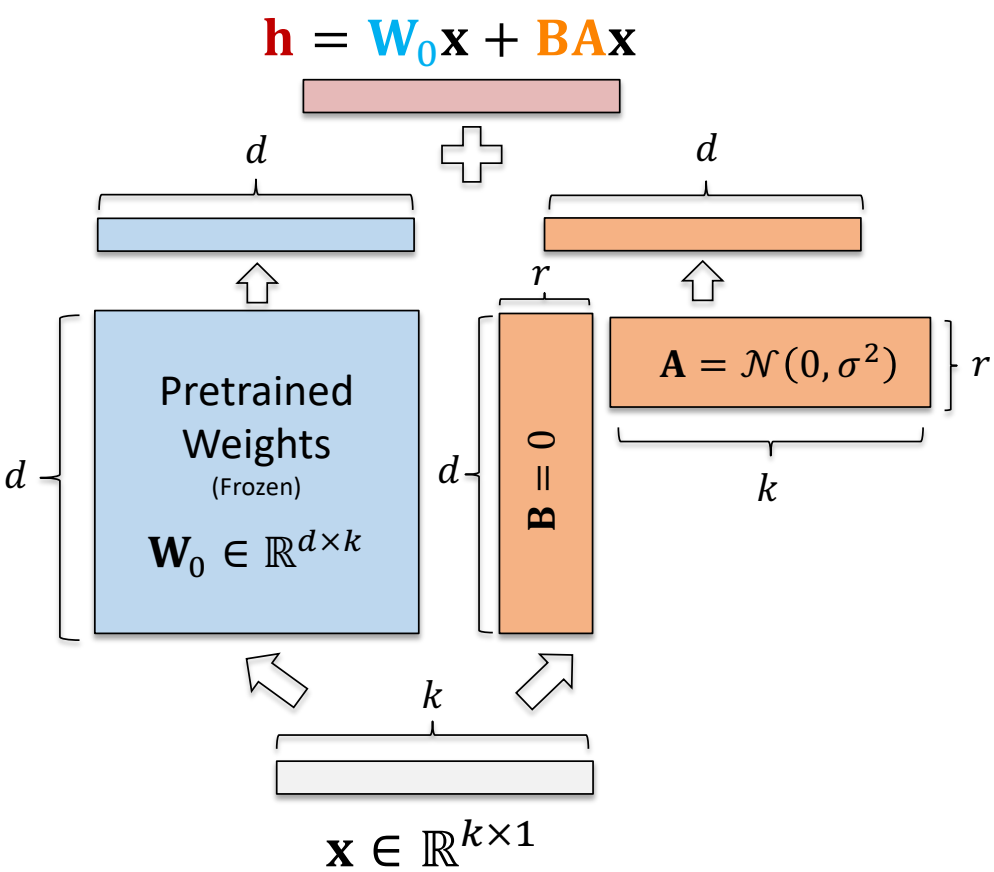
LoRA (Low-Rank Adaptation)

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



LoRA (Low-Rank Adaptation)





LoRA: $\Delta W = BA$

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

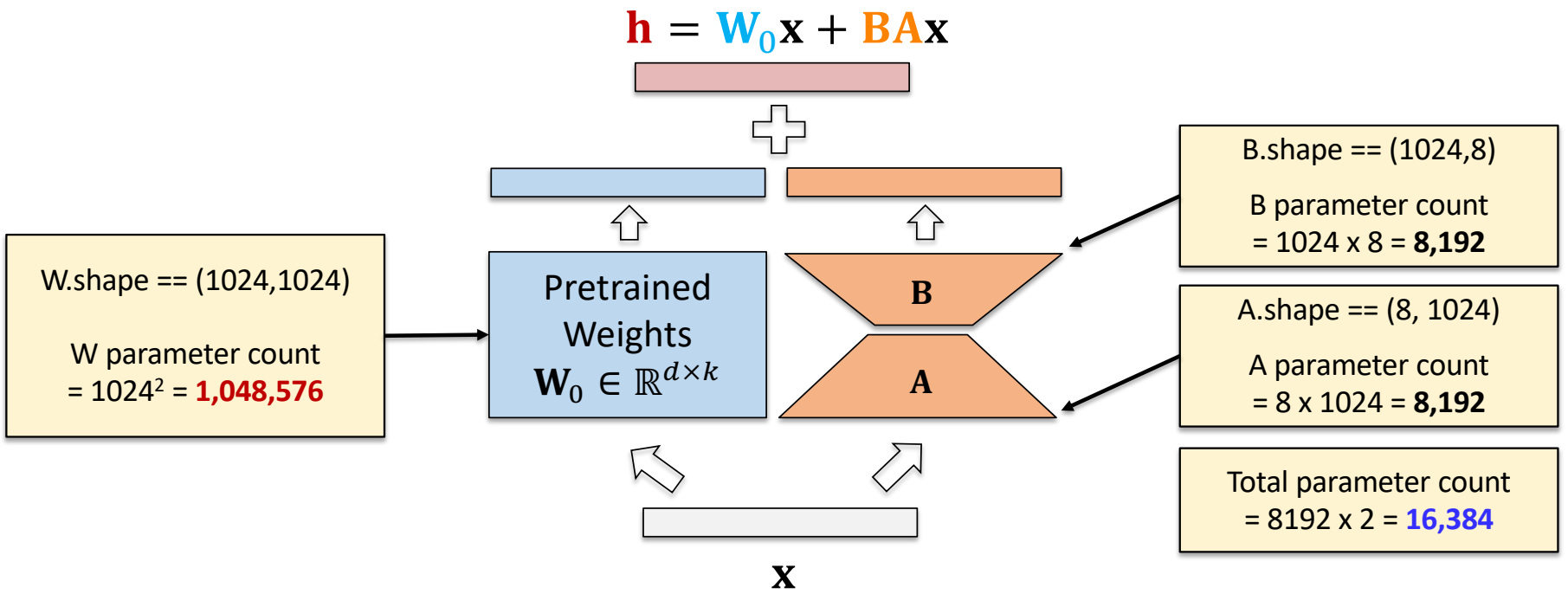
$\mathbf{B}$ $\mathbf{A}$ $\Delta \mathbf{W}$

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

$\mathbf{W}_0$ $\Delta \mathbf{W}$

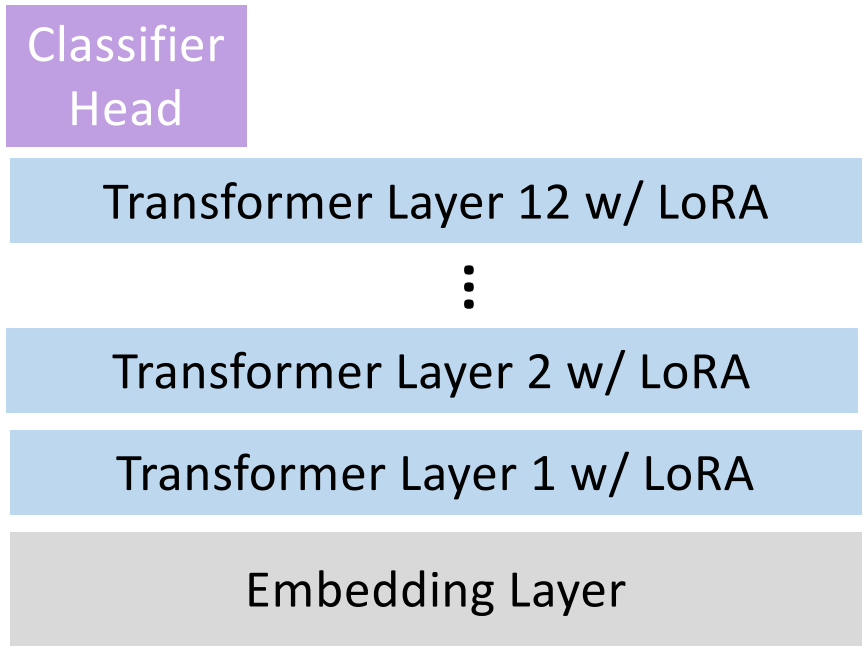
LoRA High Parameter Efficiency Example

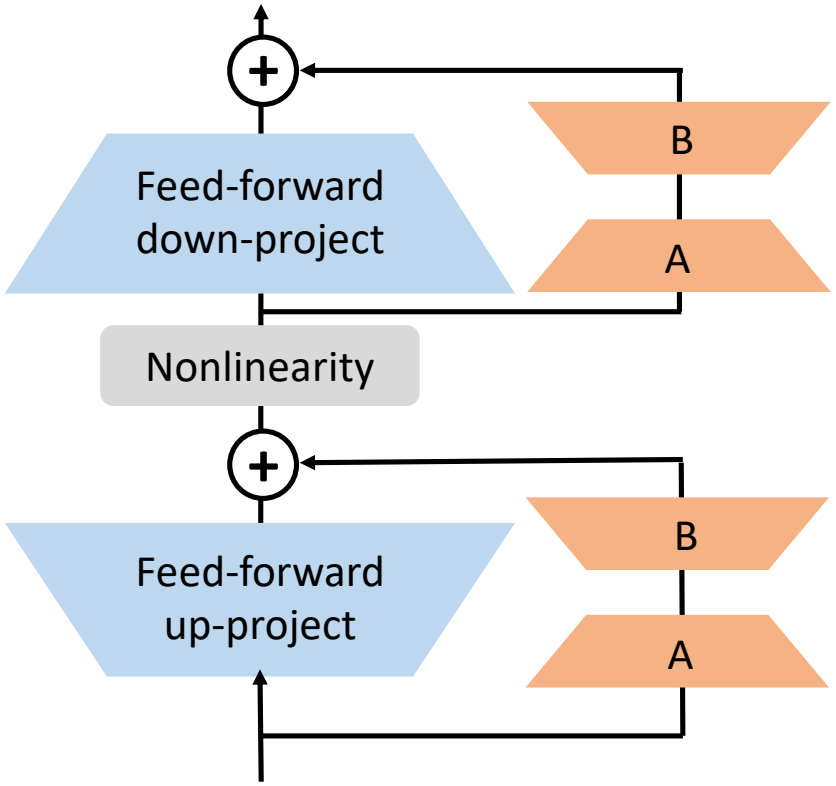
- X% of weights trained => (a) lower memory footprint and (b) faster finetuning jobs



Applying LoRA to Feed-Forward Networks

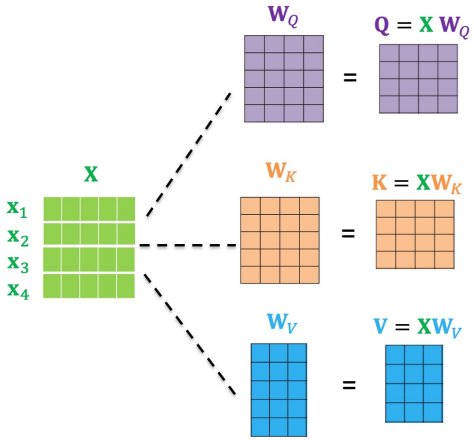
- **LoRA** can be applied to each **fully-connected layer and the classifier heads** are the task-specific modules

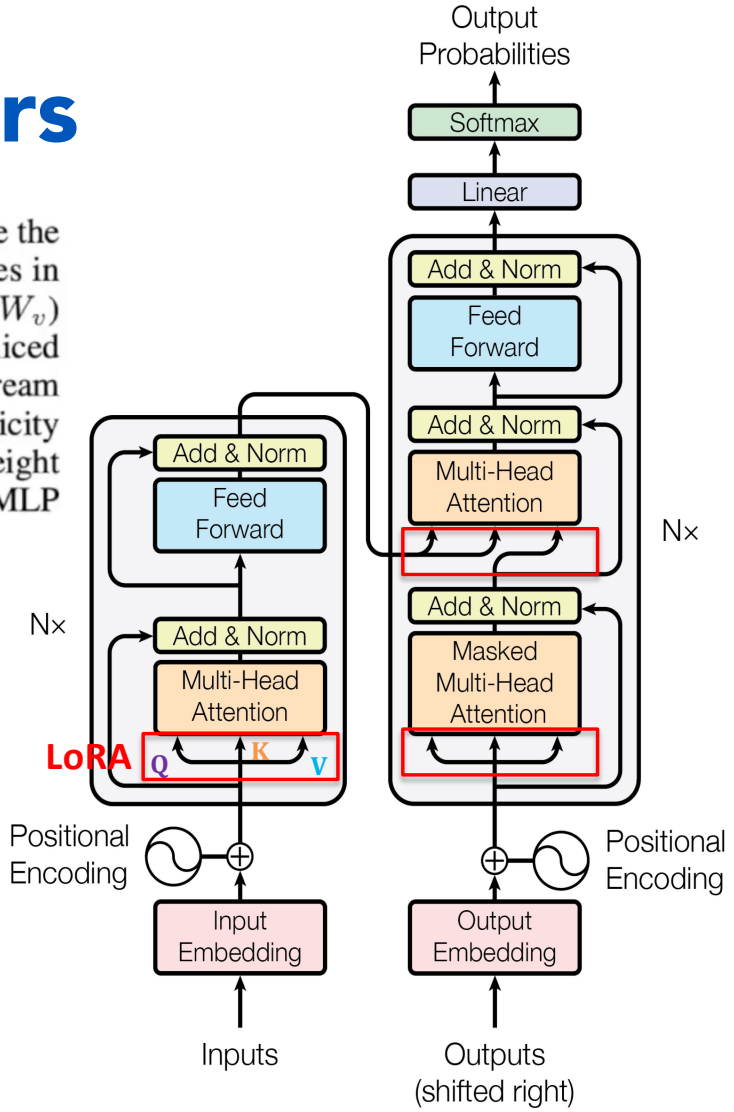




Applying LoRA To Transformers

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.





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 (±0.5%)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI (±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 $\mathbf{A}$.

$$\mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \frac{\alpha}{r}\mathbf{BA}$$

Scaling factor

Rank

- 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	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL(±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 (±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.
$\text{RoB}_{\text{large}}$ (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**

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

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

Hugging Face PEFT

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

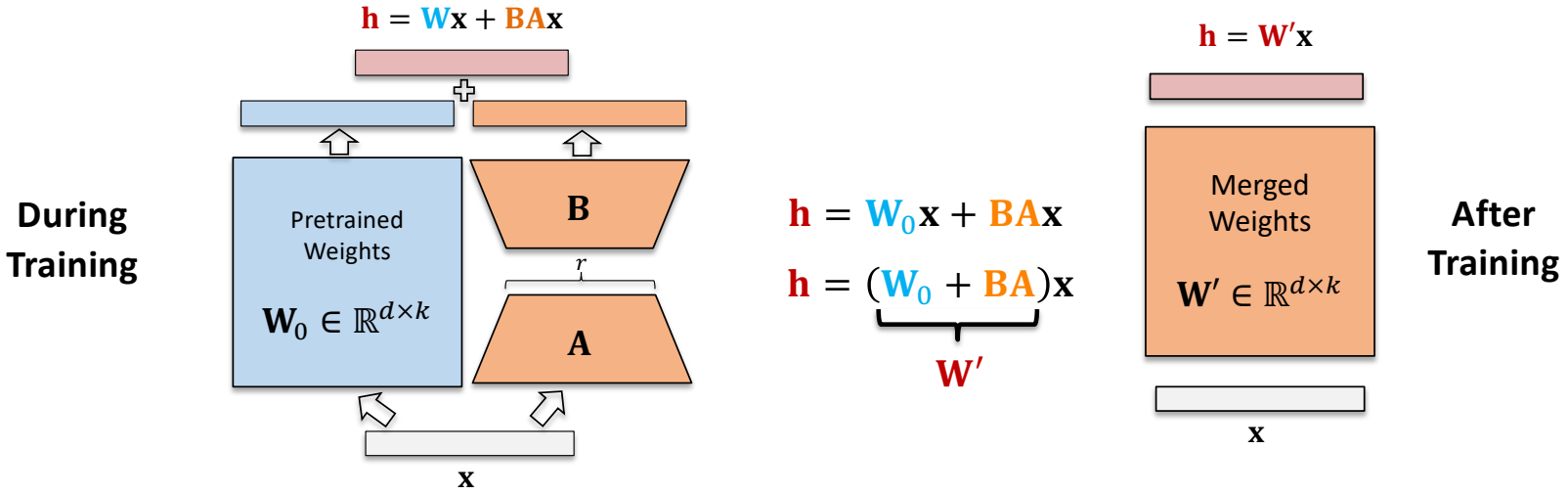
```
config = {
    "peft_type": "LORA",
    "task_type": "SEQ_2_SEQ_LM",
    "inference_mode": False,
    "r": 8,
    "target_modules": ["q", "v"],
    "lora_alpha": 32,
    "lora_dropout": 0.1,
    "fan_in_fan_out": False,
    "enable_lora": None,
    "bias": "none",
}
```

```
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.

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 ΔW (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.



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

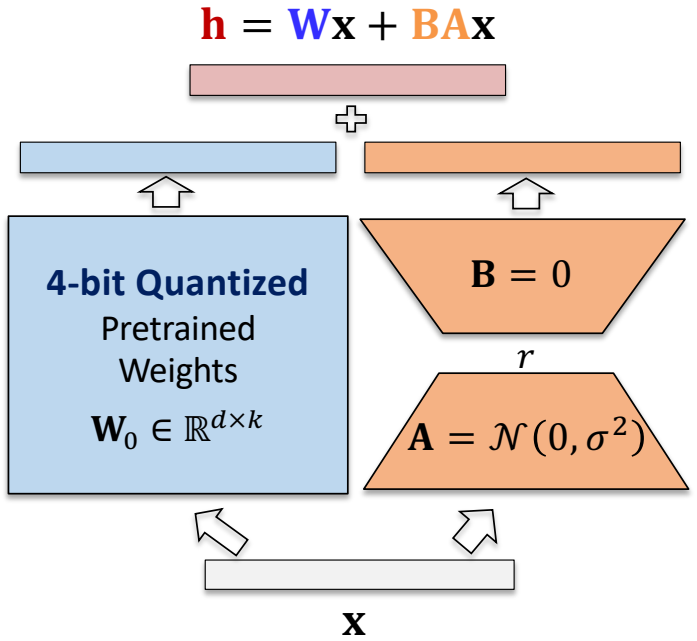
https://arxiv.org/pdf/2305.14314.pdf

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:

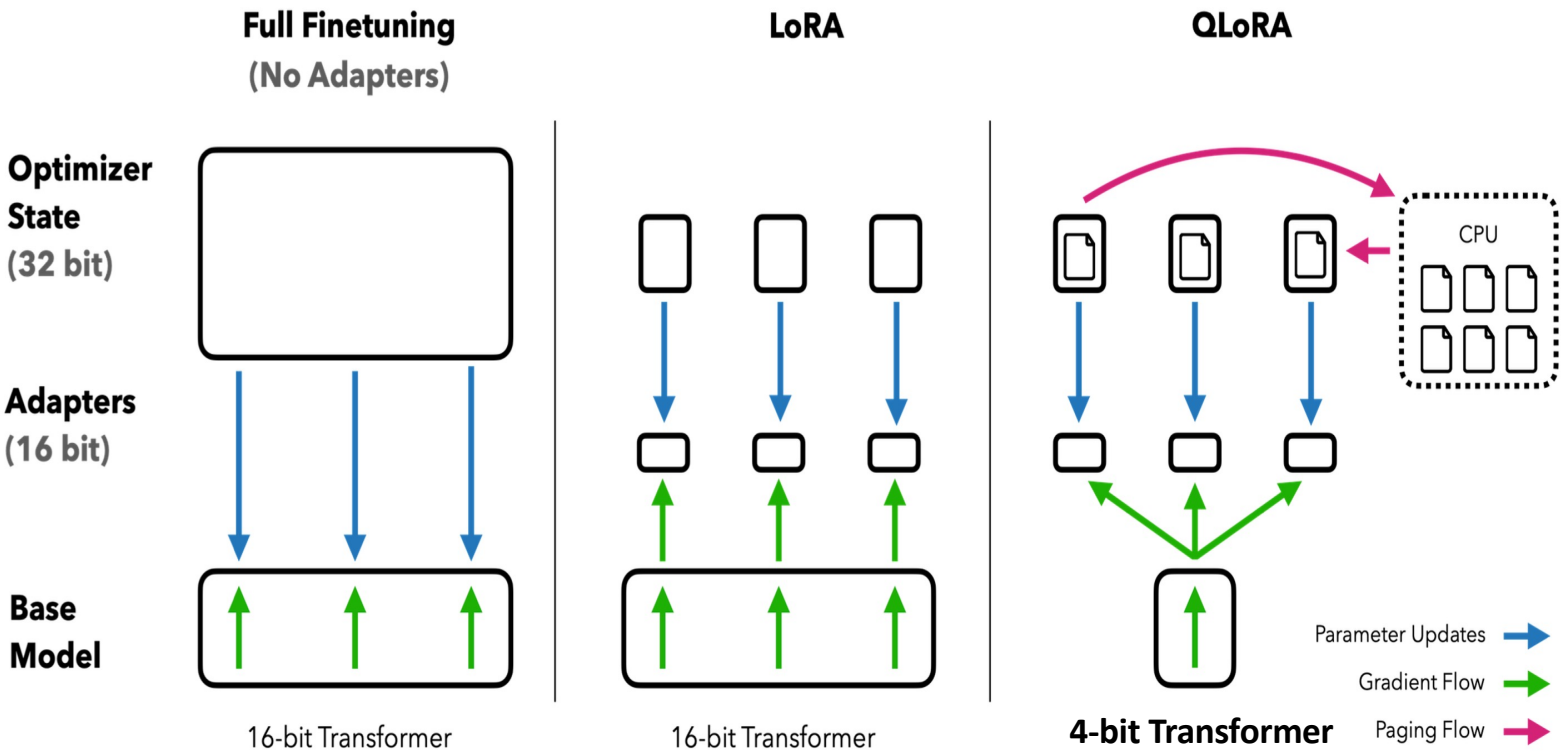
1. **LoRA** (a technique for efficient language model tuning)
2. **Model Quantization** (a method for reducing model size and memory usage)

  - 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.



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:

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

$$\text{dequant}\left(c^{\text{FP32}}, \mathbf{X}^{\text{Int4}}\right) = \frac{\mathbf{X}^{\text{Int4}}}{c^{\text{FP32}}} = \mathbf{X}^{\text{Int32}}$$

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

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

$$\mathbf{X}^{\text{Int4}} = \text{round}\left(\frac{16}{\text{absmax}(\mathbf{X}^{\text{FP32}})} \cdot \mathbf{X}^{\text{Int4}}\right) = \text{round}\left(c^{\text{FP32}} \cdot \mathbf{X}^{\text{Int4}}\right)$$

$c^{\text{FP32}} = \frac{16}{\text{absmax}(\mathbf{X}^{\text{FP32}})}$ is the quantization constant (scale factor)

QLoRA

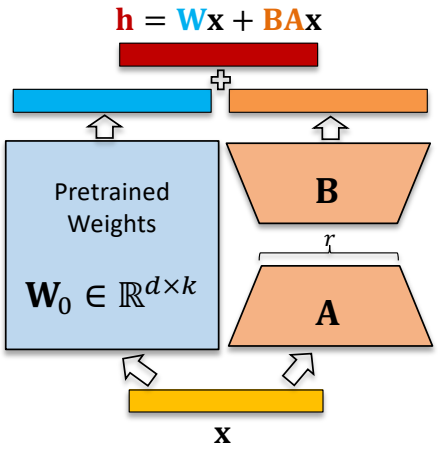
- Using the components described above, we define QLoRA for a single linear layer in the quantized base model with a single LoRA adapter as follows:

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

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

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

- 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.



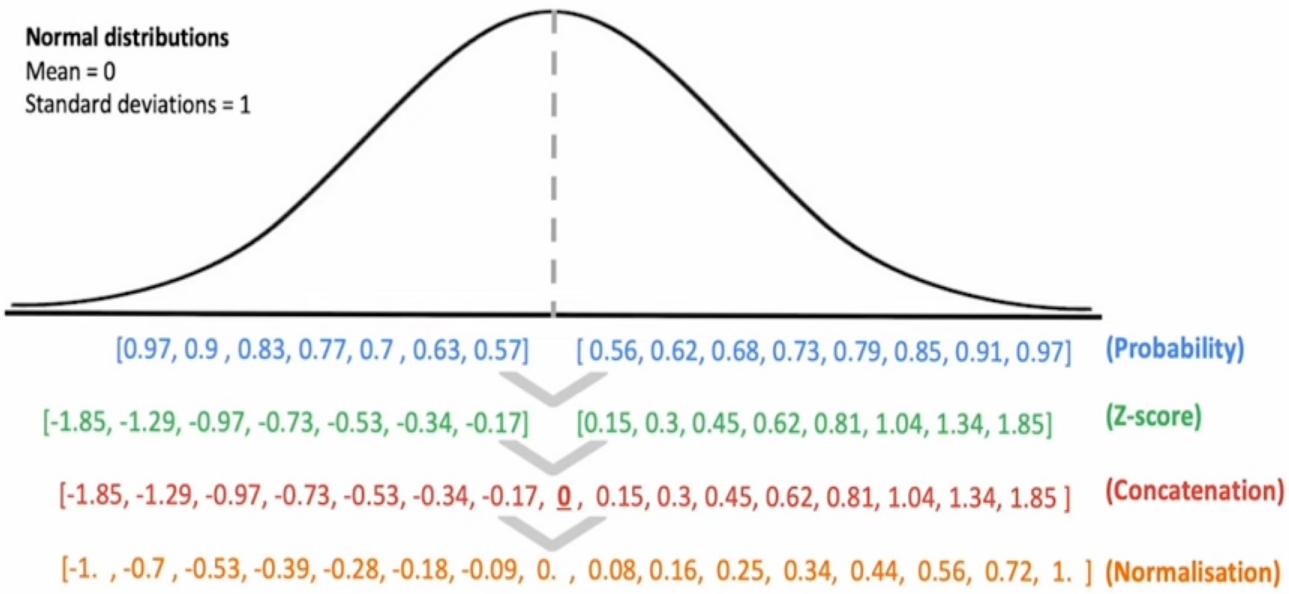
4-bit NormalFloat (NF4)

- 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:

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

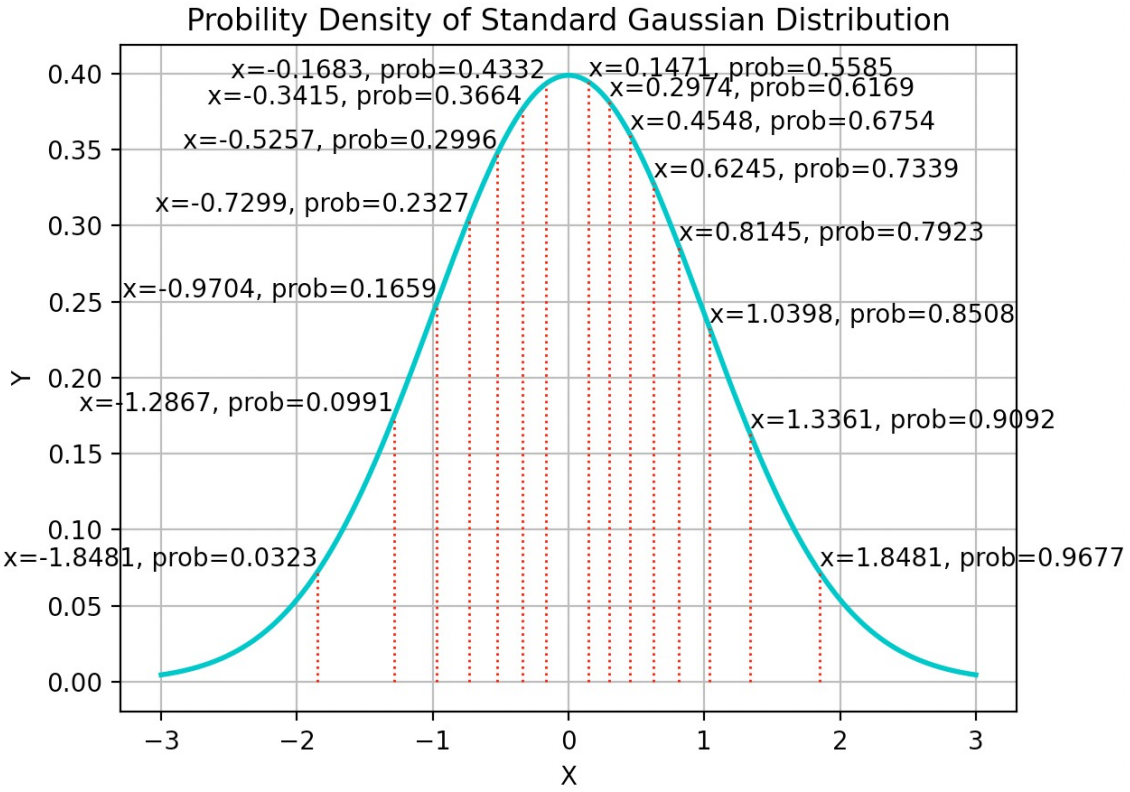
4-bit NormalFloat (NF4)

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



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

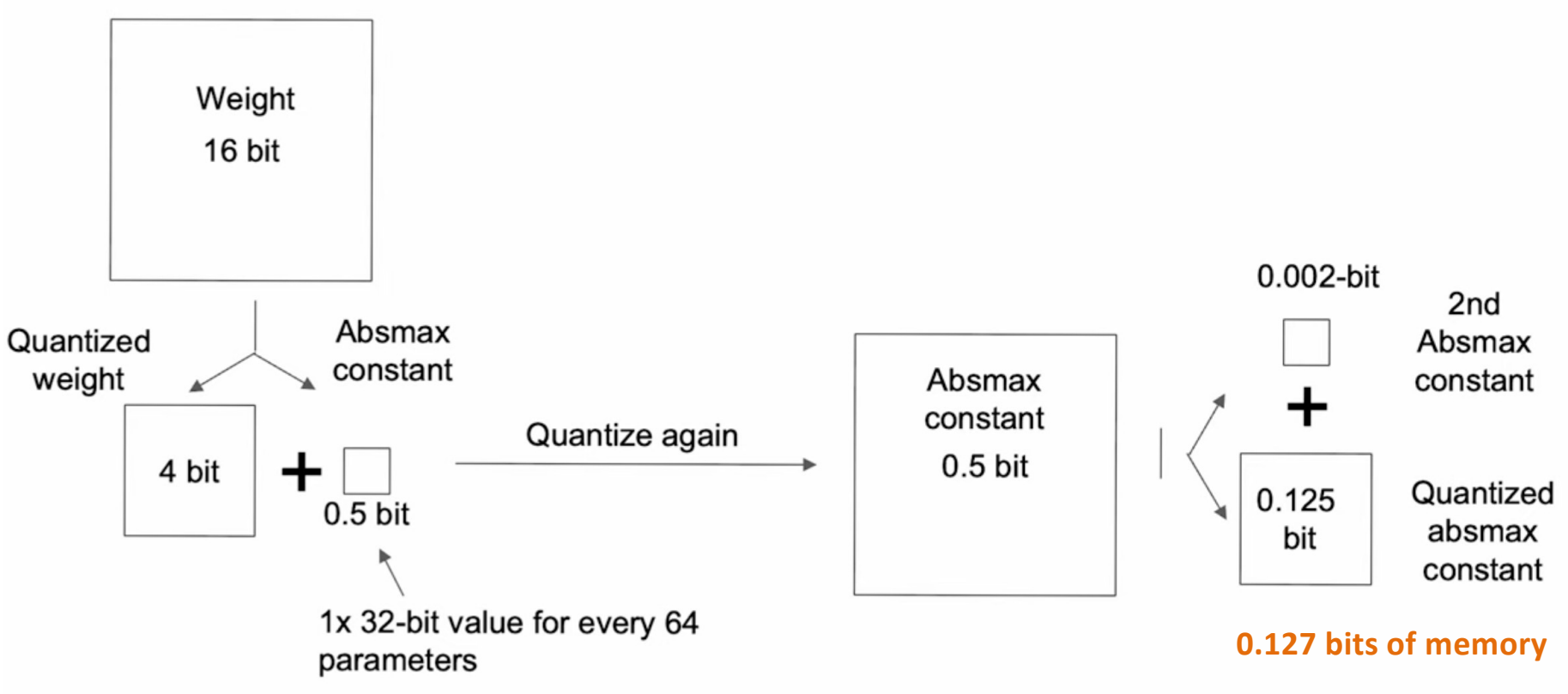
Double Quantization

- **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

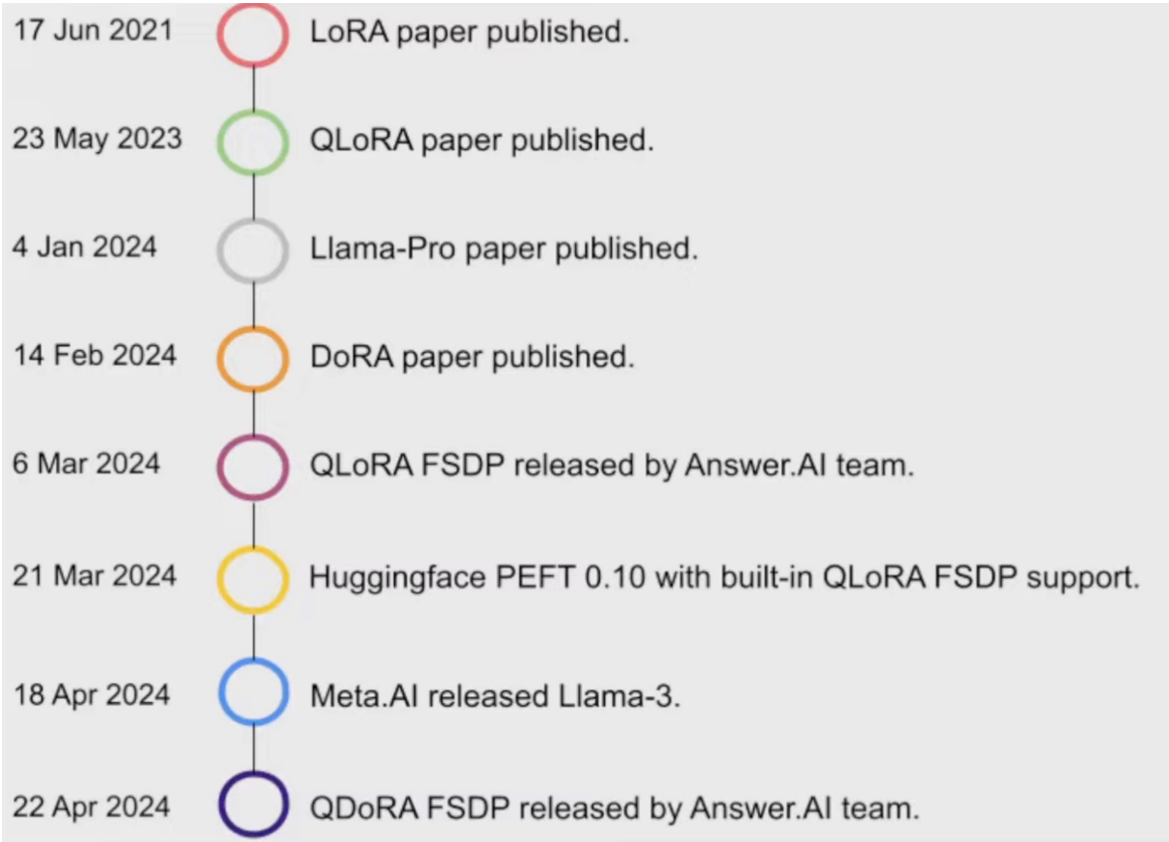
https://www.youtube.com/watch?v=t1caDsMzWBk

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.

Optional Content

- **DoRA** (Optional)
- **MoRA** (Optional)
- **SBoRA** (Optional)
- **Multi-SBoRA** (Optional)

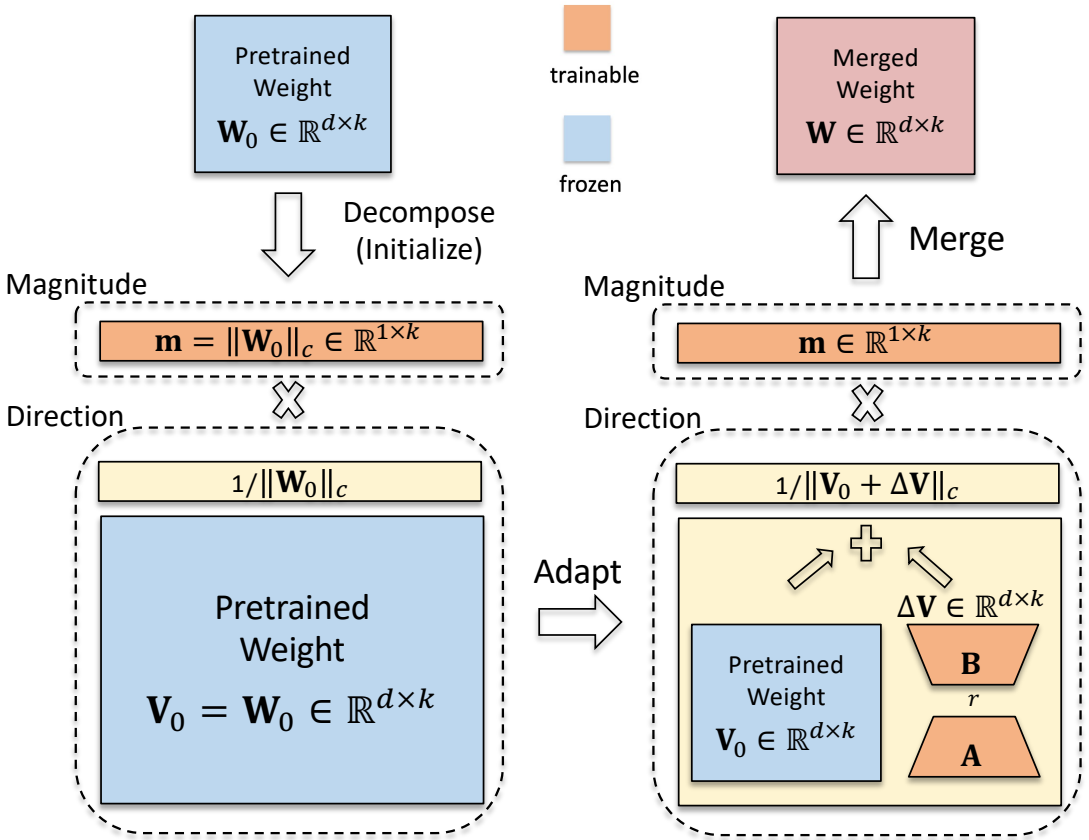


https://www.youtube.com/watch?v=WLDehSkSIhY

DoRA (Wang et al., 2024-02)

Weight-Decomposed Low-Rank Adaptation

- In DoRA, the original model weights are **decomposed into a Magnitude vector** $\mathbf{m}$ **and a Direction matrix** $\frac{\mathbf{V}}{\|\mathbf{V}\|_c}$ prior to applying LoRA
- The direction matrix is created by **scaling each column of the matrix** to be unit length
- The magnitude vector stores the scaling factors needed for each column



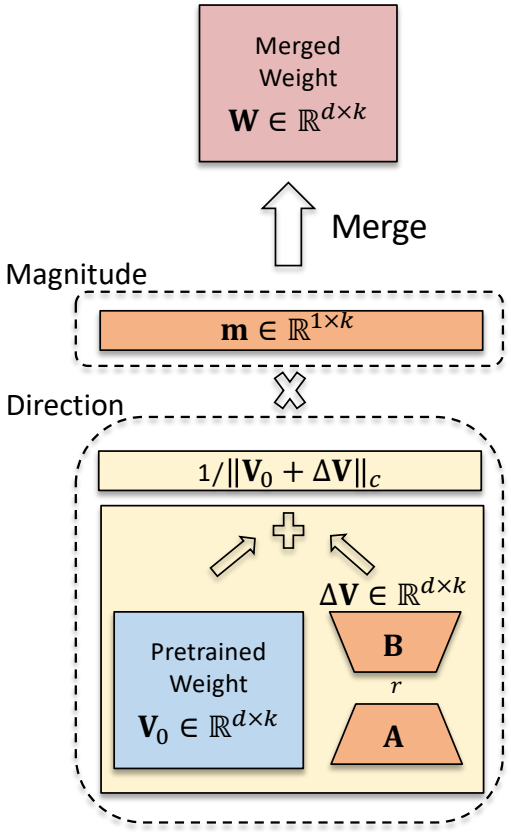
https://arxiv.org/html/2402.09353v3

DoRA: Weight Decomposition Analysis

- In DoRA, the weight matric $\mathbf{W} \in \mathbb{R}^{d \times k}$ is decomposed into the magnitude and the direction:

$$\mathbf{W} = \|\mathbf{W}\|_c \frac{\mathbf{W}}{\|\mathbf{W}\|_c} = \mathbf{m} \frac{\mathbf{V}}{\|\mathbf{V}\|_c}$$

  - $\|\mathbf{V}\|_c$ is vector-wise norm of the matrix across each column such that $\mathbf{V}/\|\mathbf{V}\|_c$ is **unit vector**
  - $\mathbf{V} \in \mathbb{R}^{d \times k}$ is the direction matrix with initial value of $\mathbf{V}_0 = \mathbf{W}_0$
  - $\mathbf{m} \in \mathbb{R}^{1 \times k}$ is the magnitude vector (trainable) with initial value of $\|\mathbf{W}_0\|_c$
- The magnitude vectors $\mathbf{m}$ are small, and are trained normally
- The direction matrices $\mathbf{V}$ are big, **LoRA is used to fine-tune it** with frozen $\mathbf{V}_0$



DoRA Methodology

- DoRA optimize both the magnitudes and directions of the pre-trained weights. Since the directional component is large in terms of the number of parameters, we further decompose it with LoRA.

$$\mathbf{W} = \mathbf{m}\frac{\mathbf{V}}{\|\mathbf{V}\|_c} = \mathbf{m}\frac{\mathbf{V}_0 + \Delta\mathbf{V}}{\|\mathbf{V}_0 + \Delta\mathbf{V}\|_c} = \mathbf{m}\frac{\mathbf{W}_0 + \mathbf{BA}}{\|\mathbf{W}_0 + \mathbf{BA}\|_c}$$

  - $\Delta\mathbf{V}$ is the incremental direction update.
  - $\|\mathbf{V}_0 + \Delta\mathbf{V}\|_c$ is just a normalization term and DoRA takes as a constant. It won't receive gradients during backpropagation.
- LoRA is applied to the transformer's query and value matrices, and the magnitude and directional differences between the original and finetuned weight matrices are calculated.
- For inference, the magnitude vectors and updated direction matrices can be combined back into updated weights for the original model

Analysis of Parameter Update Correlations

- For evaluating the parameter update correlations between full finetuning (FT), LoRA and DoRA, the authors calculated:
  - Directional Change using cosine similarity $\Delta\mathbf{D}_{FT}^{t}$:

$$\Delta\mathbf{D}_{FT}^{t} = \frac{\sum_{n-1}^{k}(1 - \mathbf{cos}(\mathbf{V}_{FT}^{n}, \mathbf{W}_{0}^{n}))}{k}$$

  - Magnitude differences between original and LoRA-updated weight matrices $\Delta\mathbf{M}_{FT}^{t}$:

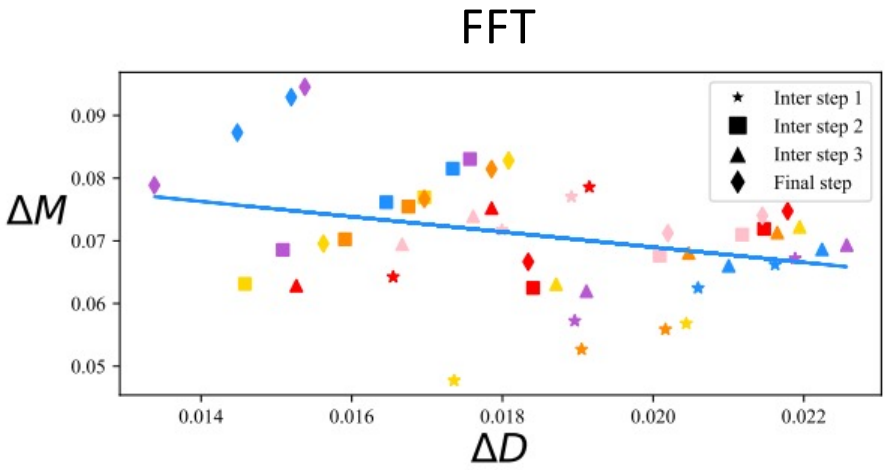
$$\Delta\mathbf{M}_{FT}^{t} = \frac{\sum_{n-1}^{k}\left|\mathbf{m}_{FT}^{n,k} - \mathbf{m}_{0}^{n}\right|}{k}$$

  at four training step checkpoints.

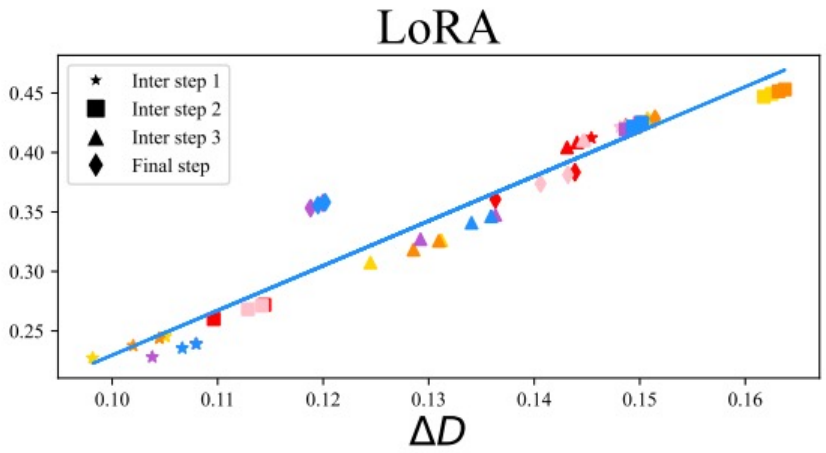
  where *t* is the number of training steps.

Parameter Update Correlations: FFT vs LoRA

- DoRA's authors found that changes to magnitude $\Delta\mathbf{M}$ and direction $\Delta\mathbf{D}$ for full finetuning (FFT) are **largely independent** (small negative correlation). Moreover, they spread with a high variance
- However, LoRA creates highly positive correlated changes (positive slope) with a significantly low variance, which **hurting performance**
  - LoRA cannot make slight directional changes alongside significant magnitude alterations.



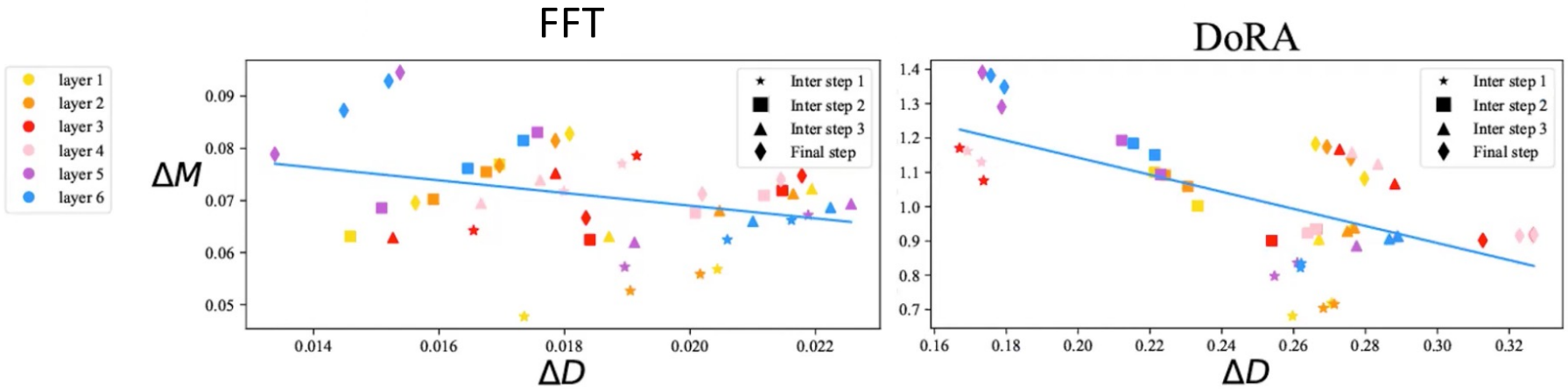
Negative Correlation



Positive Correlation

Parameter Update Correlations: FFT vs DoRA

- DoRA fixes the problem, resulting in pattern more similar to full finetuning, improving performance compared to LoRA
  - DoRA produces a pattern that closely resembles FFT.



The distribution for DoRA closely resembles FFT.

DoRA Commonsense Reasoning Results

- First test was commonsense reasoning with LLaMA, LLaMA2 and LLaMA3 models

Table 1. Accuracy comparison of LLaMA 7B/13B, LLaMA2 7B, and LLaMA3 8B with various PEFT methods on eight commonsense reasoning datasets. Results of all the baseline methods on LLaMA 7B/13B are taken from (Hu et al., 2023). Results of LoRA on LLaMA2 7B and LLaMA3 8B are obtained using the hyperparameters described in (Hu et al., 2023). DoRA†: the adjusted version of DoRA with the rank halved.

Model	PEFT Method	# Params (%)	BoolQ	PIQA	SIQA	HellaSwag	WinoGrande	ARC-e	ARC-c	OBQA	Avg.
ChatGPT	-	-	73.1	85.4	68.5	78.5	66.1	89.8	79.9	74.8	77.0
LLaMA-7B	Prefix	0.11	64.3	76.8	73.9	42.1	72.1	72.9	54.0	60.6	64.6
	Series	0.99	63.0	79.2	76.3	67.9	75.7	74.5	57.1	72.4	70.8
	Parallel	3.54	67.9	76.4	78.8	69.8	78.9	73.7	57.3	75.2	72.2
	LoRA	0.83	68.9	80.7	77.4	78.1	78.8	77.8	61.3	74.8	74.7
	DoRA† (Ours)	0.43	70.0	82.6	79.7	83.2	80.6	80.6	65.4	77.6	**77.5**
	DoRA (Ours)	0.84	69.7	83.4	78.6	87.2	81.0	81.9	66.2	79.2	**78.4**
LLaMA-13B	Prefix	0.03	65.3	75.4	72.1	55.2	68.6	79.5	62.9	68.0	68.4
	Series	0.80	71.8	83	79.2	88.1	82.4	82.5	67.3	81.8	79.5
	Parallel	2.89	72.5	84.9	79.8	92.1	84.7	84.2	71.2	82.4	81.4
	LoRA	0.67	72.1	83.5	80.5	90.5	83.7	82.8	68.3	82.4	80.5
	DoRA† (Ours)	0.35	72.5	85.3	79.9	90.1	82.9	82.7	69.7	83.6	**80.8**
	DoRA (Ours)	0.68	72.4	84.9	81.5	92.4	84.2	84.2	69.6	82.8	**81.5**
LLaMA2-7B	LoRA	0.83	69.8	79.9	79.5	83.6	82.6	79.8	64.7	81.0	77.6
	DoRA† (Ours)	0.43	72.0	83.1	79.9	89.1	83.0	84.5	71.0	81.2	**80.5**
	DoRA (Ours)	0.84	71.8	83.7	76.0	89.1	82.6	83.7	68.2	82.4	**79.7**
LLaMA3-8B	LoRA	0.70	70.8	85.2	79.9	91.7	84.3	84.2	71.2	79.0	80.8
	DoRA† (Ours)	0.35	74.5	88.8	80.3	95.5	84.7	90.1	79.1	87.2	**85.0**
	DoRA (Ours)	0.71	74.6	89.3	79.9	95.5	85.6	90.5	80.4	85.8	**85.2**

DoRA Results on Multimodal Tasks

- DoRA also beat LoRA on multimodal tasks with image & video on VL-BART, and with visual instruction tuning on LLaVA

Table 2. The multi-task evaluation results on VQA, GQA, $NVLR^2$ and COCO Caption with the VL-BART backbone.

Method	# Params (%)	VQA^{v2}	GQA	$NVLR^2$	COCO Cap	Avg.
FT	100	66.9	56.7	73.7	112.0	77.3
LoRA	5.93	65.2	53.6	71.9	115.3	76.5
DoRA (Ours)	5.96	65.8	54.7	73.1	115.9	**77.4**

Table 3. The multi-task evaluation results on TVQA, How2QA, TVC, and YC2C with the VL-BART backbone.

Method	# Params (%)	TVQA	How2QA	TVC	YC2C	Avg.
FT	100	76.3	73.9	45.7	154	87.5
LoRA	5.17	75.5	72.9	44.6	140.9	83.5
DoRA (Ours)	5.19	76.3	74.1	45.8	145.4	**85.4**

Table 4. Visual instruction tuning evaluation results for LLaVA-1.5-7B on a wide range of seven vision-language tasks. We directly use checkpoints from (Liu et al., 2023a) to reproduce their results.

Method	# Params(%)	Avg.
FT	100	66.5
LoRA	4.61	66.9
DoRA (Ours)	4.63	**67.6**

DoRA Ablations

- DoRA performs significantly better than LoRA when the rank is small. Due to the magnitude vectors, the number of parameters increases marginally.
- Also ablated using just the magnitude adapter in some components

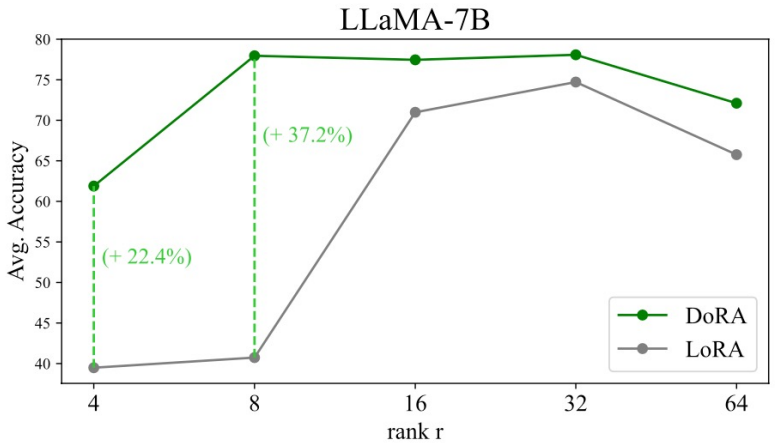


Figure 4. Average accuracy of LoRA and DoRA for varying ranks for LLaMA-7B on the commonsense reasoning tasks.

Table 6. Accuracy comparison of LLaMA 7B/13B with two different tuning granularity of DoRA. Columns **m** and **V** designate the modules with tunable magnitude and directional components, respectively. Each module is represented by its first letter as follows: (Q)uery, (K)ey, (V)alue, (O)utput, (G)ate, (U)p, (D)own.

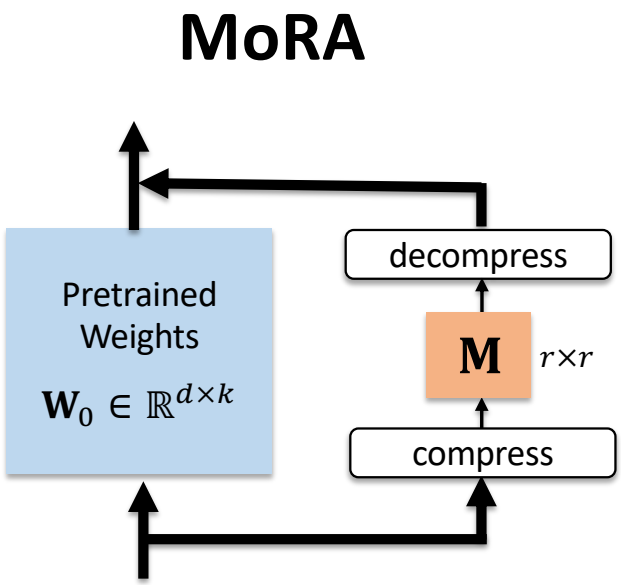
Model	PEFT Method	# Params (%)	m	V	Avg.
LLaMA-7B	LoRA	0.83	-	-	74.7
	DoRA (Ours)	0.84	QKVUD	QKVUD	78.1
	DoRA (Ours)	0.39	QKVOGUD	QKV	77.5
LLaMA-13B	LoRA	0.67	-	-	80.5
	DoRA (Ours)	0.68	QKVUD	QKVUD	81.5
	DoRA (Ours)	0.31	QKVOGUD	QKV	81.3

DoRA Summary

- **Key Findings on LoRA Training**
  - LoRA training artificially forces large changes in direction to have large changes in magnitude, and vice versa.
- **Decoupling Magnitude and Direction Changes**
  - Authors decomposed each weight matrix into a magnitude vector and a direction matrix to decouple magnitude and direction changes.
- **Improved Performance**
  - Consistently beat LoRA performance with many different models, tasks, and rank settings.
  - Improved VeRA performance, suggesting potential generalizability to other variants.

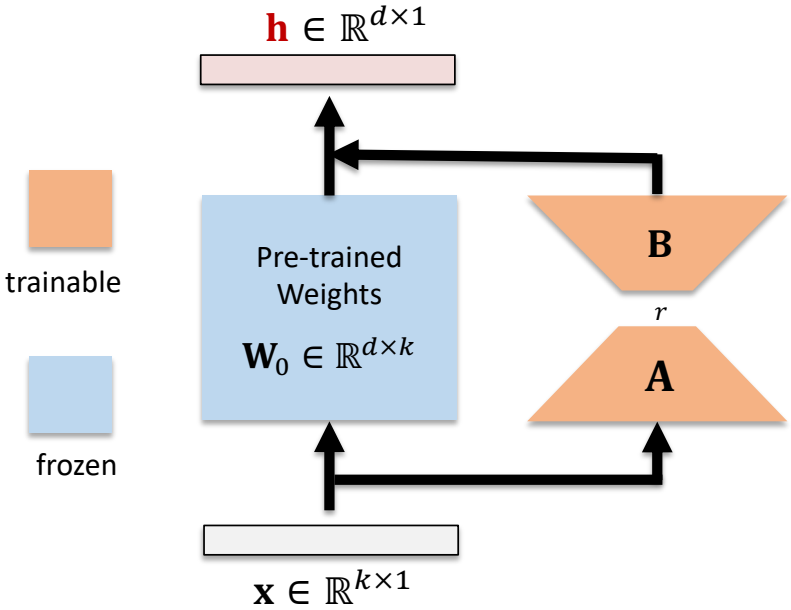
MoRA (T. Jiang et al. 2024-05)

- MoRA is a new PEFT method for LLMs, which uses square matrices to enable high-rank updating.
- It outperforms Low-Rank Adaptation (LoRA) in memory-intensive tasks and continual pre-training while maintaining comparable performance in instruction tuning.
- MoRA addresses LoRA's limitations in knowledge enhancement, demonstrating superior ability to memorize new information.

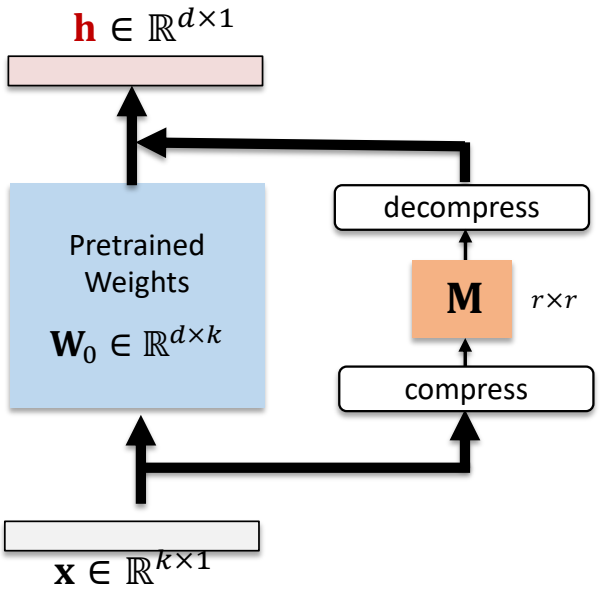


MoRA: High-Rank Updating for PEFT

- MoRA's innovation is its use of a compress function to project input vectors into a lower-dimensional space, perform high-rank transformations using a smaller matrix, and then apply a decompress function to project the result back into the original higher-dimensional space.



LoRA (r = 8)



MoRA (r = 256)

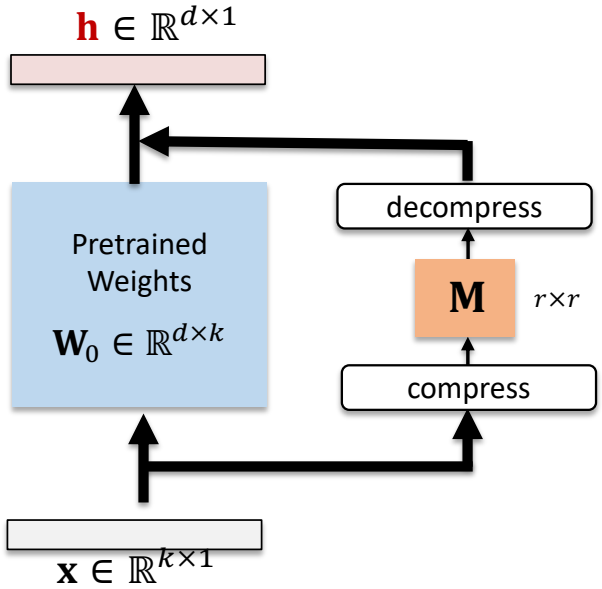
MoRA: High-Rank Updating for PEFT

- MoRA's innovation is its use of compress and decompress functions to project input vectors into a lower-dimensional space, perform high-rank transformations with a smaller matrix, and then project back to the original space.

$$\Delta \mathbf{W}\mathbf{x} = f_{\text{decomp}}(\mathbf{M}\, f_{\text{comp}}(\mathbf{x}))$$

$$\mathbf{h} = \mathbf{W}_0\mathbf{x} + f_{\text{decomp}}(\mathbf{M}\, f_{\text{comp}}(\mathbf{x}))$$

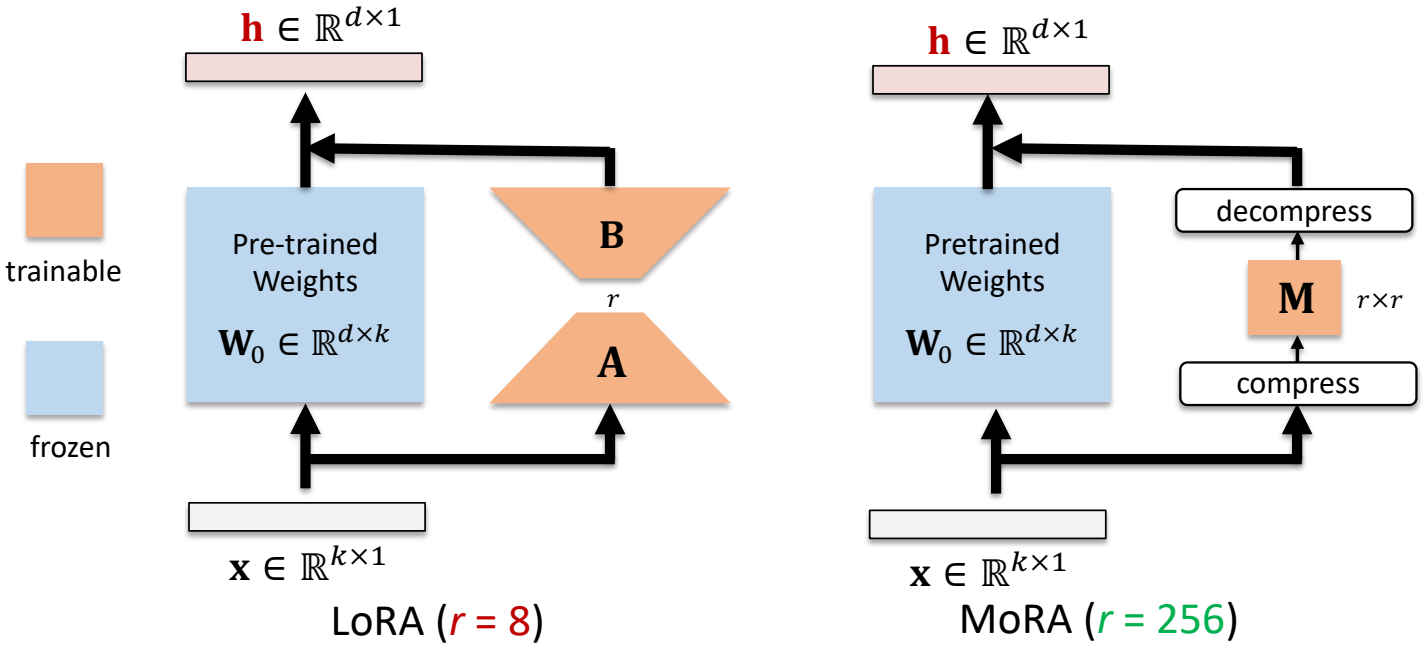
- This approach, using **non-parameterized operators**, enables high-rank updates without increasing trainable parameters, differing from LoRA.



MoRA (r = 256)

Parameter Efficiency of MoRA

- For example, it is given that the weight layer is 4096 x 4096, which means 16,777,216 parameters would need to be updated with FFT.
- If *r* = 8 is chosen with LoRA, it would result in 2 x (4096 x 8) = 65,536 parameters being updated.
- With MoRA, if a 256 x 256 matrix is chosen for **M**, it would mean 256 x 256 = 65,536 parameters would need to be tuned, the same number as LoRA.



Non-Parameterized Compress and Decompress

- MoRA explored four non-parameterized methods for designing compress and decompress operators:
  1. **Truncation**: Simple, but can result in significant information loss.
  2. **Row and Column Sharing**: Effective for larger ranks (r=128, 256), preserving more input information.
  3. **Decomposition** (for smaller ranks, r=8): Breaks input vectors into subvectors to mitigate information loss.
  4. **Rotation** (inspired by RoPE): Integrates rotation operators to boost the expressive power of the matrix M, capturing nuanced differences between input segments.

Performance of MoRA

- In DoRA evaluation experiments, the authors first focused on memorizing UUID pairs, comparing MoRA with LoRA and FFT. Using ranks 8 and 256, MoRA demonstrated significant improvements over LoRA while using the same number of trainable parameters.
- MoRA required fewer training steps to memorize UUID pairs compared to LoRA. At rank 256, MoRA achieved performance similar to FFT, with both methods able to memorize all pairs within 500 steps.

Character-level accuracy of memorizing UUID pairs by generating the value of corresponding key in 300, 500, 700 and 900 training steps.

	Rank	300	500	700	900
FFT	-	42.5	100	100	100
LoRA	8	9.9	10.0	10.7	54.2
MoRA	8	10.1	15.7	87.4	100
LoRA	256	9.9	70.6	100	100
MoRA	256	41.6	100	100	100

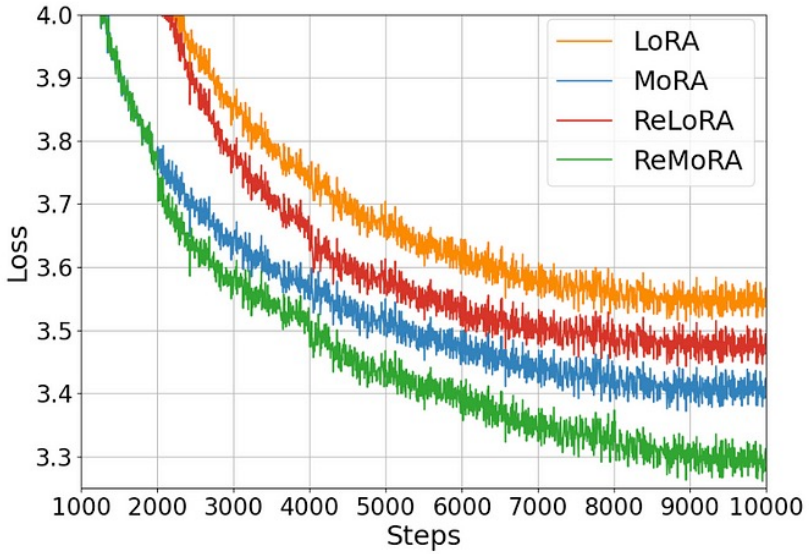
Performance of MoRA

- MoRA was evaluated across three fine-tuning tasks:
  1. Instruction tuning on Tülu v2 dataset, with zero-shot and five-shot MMLU evaluations.
  2. Mathematical reasoning on MetaMath dataset, with GSM8K and MATH evaluations.
  3. Continual pretraining on biomedical and financial domains using PubMed abstracts and financial news data.

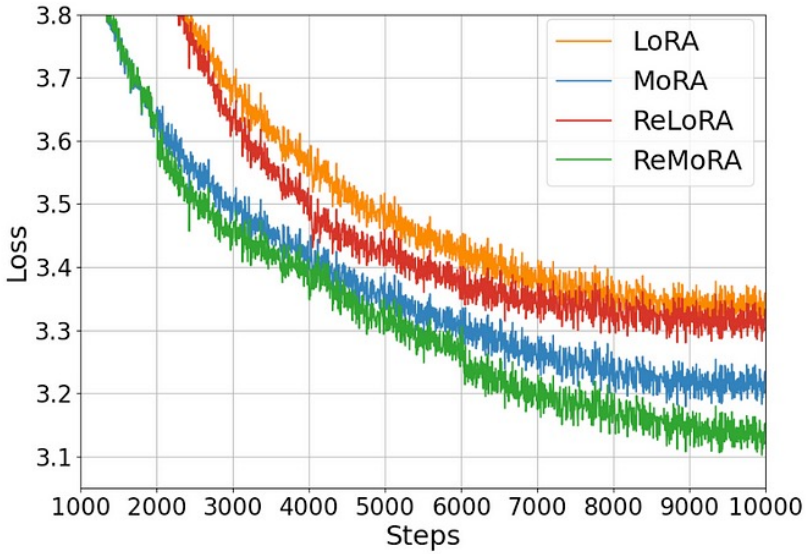
		Instruction Tuning		Mathematical Reasoning		Continual Pretraining	
Method	**Rank**	**MMLU 0**	**MMLU 5**	**GSM8K**	**MATH**	**BioMed.**	**Finance**
FFT	-	50.6	51.3	66.6	20.1	56.4	69.6
LoRA	8	50.2	51.5	**64.6**	15.1	52.3	64.0
LoRA+	8	49.2	51.1	64.1	**15.8**	52.2	64.9
ReLoRA	8	49.3	50.2	61.5	14.5	46.3	61.0
AsyLoRA	8	**50.3**	**52.2**	64.5	15.0	52.5	63.5
DoRA	8	50.2	51.5	64.5	14.6	52.5	63.9
MoRA (Ours)	8	49.7	51.5	64.2	15.4	**53.3**	**67.1**
LoRA	256	49.7	50.8	67.9	**19.9**	54.1	67.3
LoRA+	256	49.2	51.3	**68.2**	17.1	54.2	66.7
ReLoRA	256	-	-	64.0	18.1	52.9	57.9
AsyLoRA	256	**50.1**	**52.0**	66.9	19.3	54.1	66.9
DoRA	256	49.6	51.1	67.4	19.5	54.2	66.0
MoRA (Ours)	256	49.9	51.4	67.9	19.2	**55.4**	**68.7**

Performance of MoRA

- In these fine-tuning tasks, MoRA was compared against several methods including FFT, LoRA, LoRA+, AsyLoRA, ReLoRA, and DoRA. Results showed that MoRA performed comparably to LoRA on instruction tuning and mathematical reasoning tasks. However, MoRA outperformed LoRA in continual pretraining for both biomedical and financial domains. Generally, higher ranks (256 vs 8) improved performance, especially in mathematical reasoning tasks.



(a) Pretraining loss at 250M models.



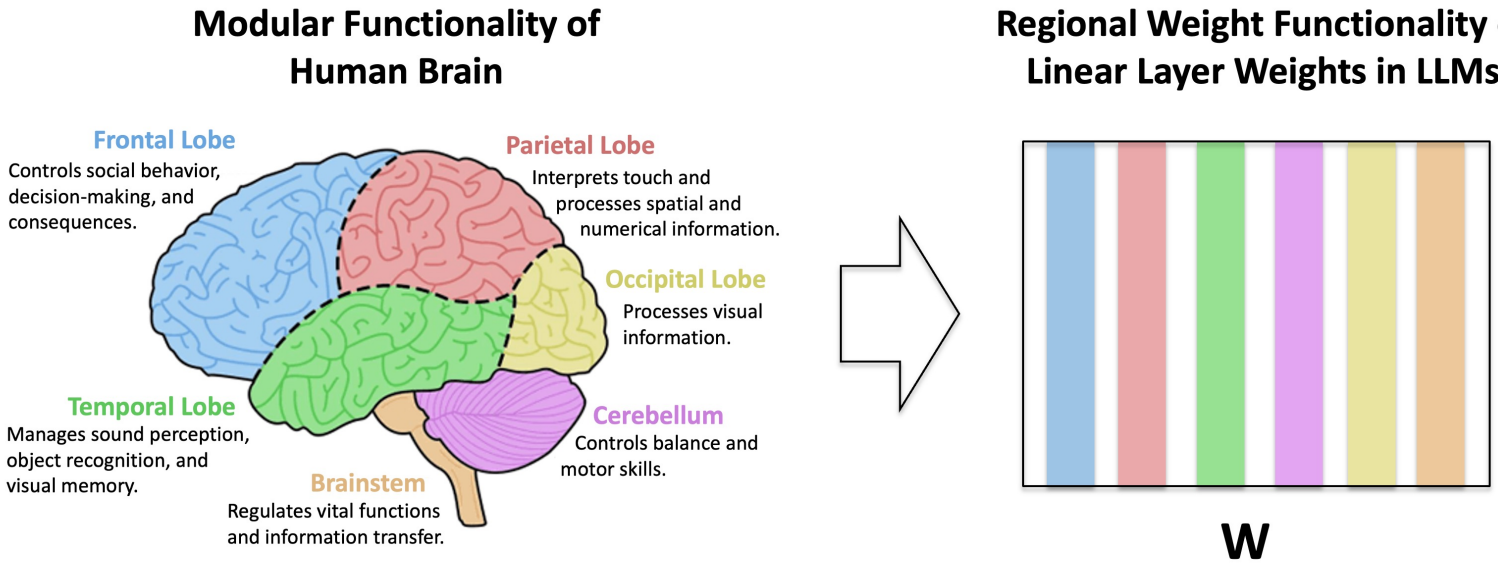
(b) Pretraining loss at 1.3B models.

Summary of MoRA

- MoRA introduces non-parameter operators to reduce input dimensions and increase output dimensions for the square matrix, allowing it to be merged back into the LLM like LoRA. The method is evaluated across five tasks: instruction tuning, mathematical reasoning, continual pretraining, memory, and pretraining.
- Results show that MoRA outperforms LoRA on memory-intensive tasks and achieves comparable performance on other tasks, demonstrating the effectiveness of high-rank updating. The authors provide a detailed analysis of their method, including various implementations of the compression and decompression functions used in MoRA.

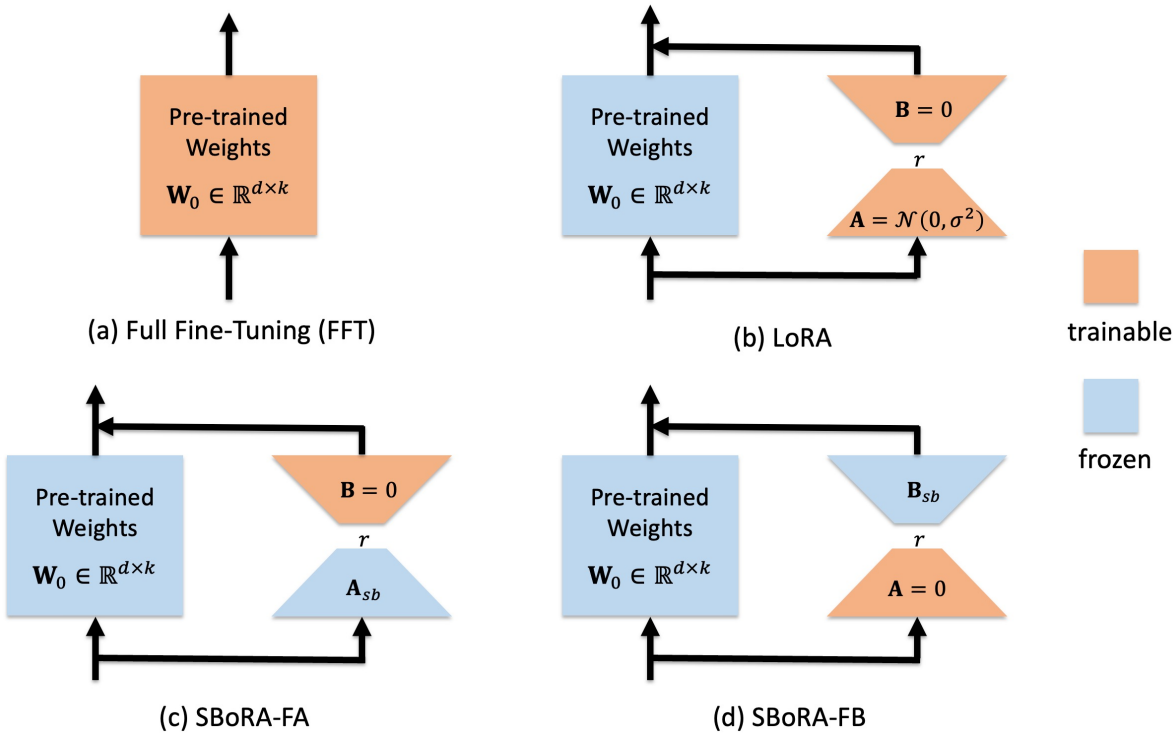
SBoRA (LM Po et al., 2024-07)

- **SBoRA**: Low-Rank Adaptation with Regional Weight Updates
- SBoRA enables regional weight updates and memory-efficient finetuning. The majority of the finetuned model's weights remain unchanged from the pre-trained weights.
- This characteristic of SBoRA is reminiscent of the modular organization of the human brain, which efficiently adapts to new tasks.



SBoRA-FA and SBoRA-FB

- **SBoRA** (**S**tandard **B**asis L**oRA**) adopts a unique approach, utilizing orthogonal standard basis vectors to construct its projection matrices. These matrices, designated as A_{sb} for SBoRA-FA (Fixed A) and B_{sb} for SBoRA-FB (Fixed B)



Standard Orthogonal Basis

- **SBoRA** leverages standard basis vectors to construct the fixed $\mathbf{A}$ or $\mathbf{B}$ matrices of the LoRA decomposition.
- Specifically, the shared orthogonal basis is **identity matrix**, comprising standard basis vectors (one-hot vectors) as their rows or columns.

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- These standard basis vectors, denoted as $\mathbf{e}_i$, possess a single non-zero entry of 1 at index *i*:

  Row standard basis vector: $\mathbf{e}_i = [0 \ldots 0\ 1\ 0 \ldots 0]$

  Column standard basis vector: $\mathbf{e}_i = [0 \ldots 0\ 1\ 0 \ldots 0]^T$

- SBoRA initializes one fixed matrix (either $\mathbf{A}$ or $\mathbf{B}$) using these standard basis vectors e_i, while the other matrix is initialized with zeros, resulting in two variants: **SBoRA-FA** (Fixed Matrix A) and **SBoRA-F**B (Fixed Matrix B).

SBoRA-FA: Regional Weight Update Example

- The finetuned weight $\mathbf{W}'$ of **SBoRA-FA** can be represented as:

$$\mathbf{W}' = \mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \mathbf{B}\mathbf{A}_{sb}$$

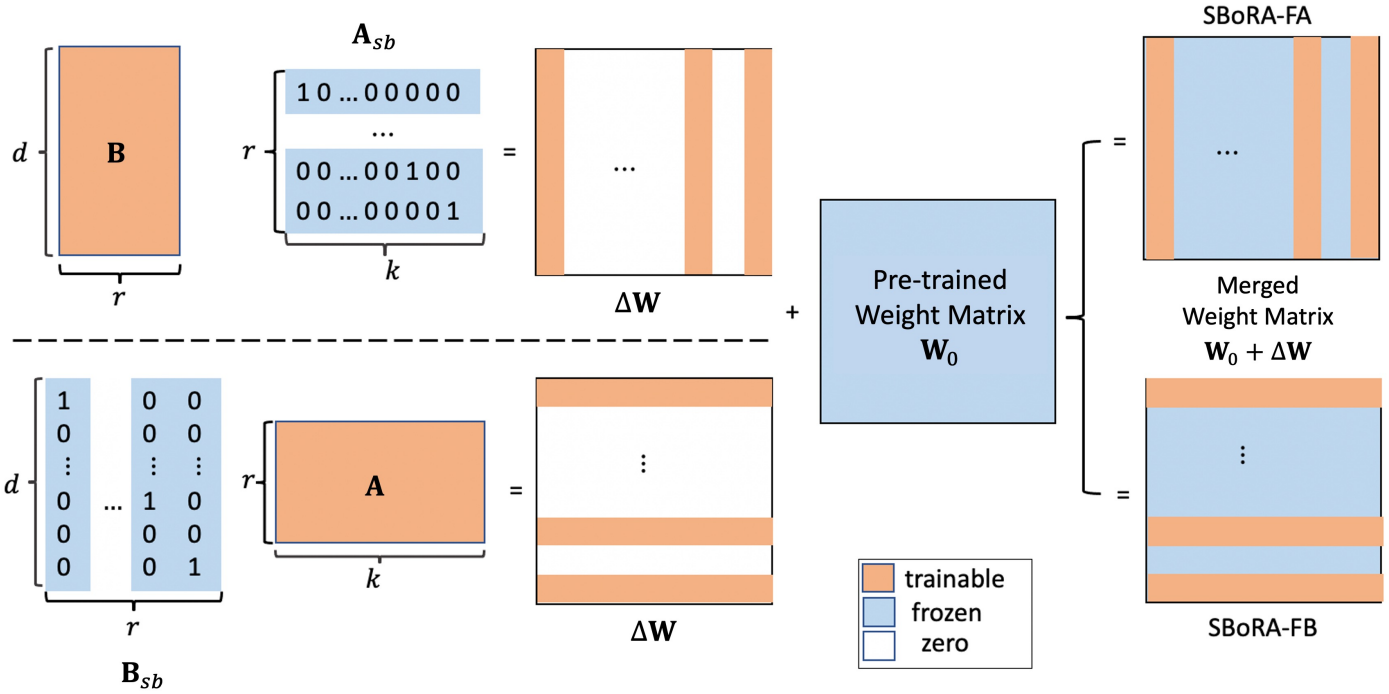
- The update matrix $\Delta\mathbf{W} = \mathbf{B}\mathbf{A}_{sb}$ is very sparse, with most of the **columns** having zero weights due to the one-hot nature of the standard basis subspace matrix $\mathbf{A}_{sb}$.
- For example, when $r = 2$ and $k = 4$ with $\mathbf{A}_{sb} = [\mathbf{e}_1 \quad \mathbf{e}_4]^T$, the $\Delta\mathbf{W}$ will only have two **non-zero columns** as

$$\Delta\mathbf{W} = \mathbf{B}\mathbf{A}_{sb} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} b_{11} & 0 & 0 & b_{12} \\ b_{21} & 0 & 0 & b_{22} \\ b_{31} & 0 & 0 & b_{32} \\ b_{41} & 0 & 0 & b_{42} \end{bmatrix}$$

- The fine-tuned weight $\mathbf{W}'$ is given by

$$\mathbf{W}' = \mathbf{W}_0 + \Delta\mathbf{W} = \begin{bmatrix} w_{11} + b_{11} & w_{12} & w_{13} & w_{14} + b_{12} \\ w_{21} + b_{21} & w_{22} & w_{23} & w_{24} + b_{22} \\ w_{31} + b_{31} & w_{32} & w_{33} & w_{34} + b_{32} \\ w_{41} + b_{41} & w_{42} & w_{43} & w_{44} + b_{42} \end{bmatrix}$$

- Regional weight update process of SBoRA, showcasing distinct $\mathbf{W}_0 + \Delta\mathbf{W}$ computing procedures of SBoRA-FA(upper) and SBoRA-FB(lower). The diagram employs different colors to represent frozen, trainable, and zero parameters.



SBoRA-FB: Regional Weight Update Example

- The finetuned weight $\mathbf{W}'$ of **SBoRA-FB** can be represented as:

$$\mathbf{W}' = \mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \mathbf{B}_{sb}\mathbf{A}$$

- The update matrix $\Delta\mathbf{W} = \mathbf{B}_{sb}\mathbf{A}$ is very sparse, with most of the **row** having zero weights due to the one-hot nature of the standard basis subspace matrix $\mathbf{B}_{sb}$.
- For example, when $r = 2$ and $k = 4$ with $\mathbf{A}_{sb} = [\mathbf{e}_1 \quad \mathbf{e}_4]^T$, the $\Delta\mathbf{W}$ will only have two **non-zero row** as

$$\Delta\mathbf{W} = \mathbf{B}_{sb}\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} \end{bmatrix}$$

- The fine-tuned weight $\mathbf{W}'$ is given by

$$\mathbf{W}' = \mathbf{W}_0 + \Delta\mathbf{W} = \begin{bmatrix} w_{11} + b_{11} & w_{12} + b_{12} & w_{13} + b_{13} & w_{14} + b_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} + b_{21} & w_{42} + b_{22} & w_{43} + b_{23} & w_{44} + b_{24} \end{bmatrix}$$

SBoRA: Commonsense Reasoning Performance

Model	Method	r	TP	BoolQ	PIQA	SIQA	HS	WG	ARC-e	ARC-c	OBQA	Avg
GPT-3.5	-	-	-	73.1	85.4	68.5	78.5	66.1	89.8	79.9	74.8	77.0
LLaMA-7B	LoRA	32	56.1M	66.8	81.1	78.4	53.5	80.5	81.1	61.9	79.4	**72.8**
	DoRA		57.0M	68.8	82.0	70.6	57.6	73.2	79.4	64.2	78.2	71.7
	SBoRA-FA		28.0M	68.0	79.7	76.2	54.4	79.1	79.8	61.3	75.0	71.7
	SBoRA-FB		28.0M	66.1	64.2	74.8	57.2	71.5	80.4	62.4	75.8	64.9
	LoRA	64	112.2M	62.1	81.8	78.2	62.9	78.6	79.8	63.7	81.2	73.5
	DoRA		113.1M	68.7	82.8	78.2	64.8	62.9	79.7	64.8	80.0	72.7
	SBoRA-FA		56.1M	68.2	81.3	77.6	74.7	81.1	80.8	62.8	79.4	**75.7**
	SBoRA-FB		56.1M	66.5	79.2	76.7	59.2	76.5	76.8	59.0	74.4	71.0
LLaMA3-8B	LoRA	32	56.6M	71.9	86.7	80.4	94.0	85.6	87.8	75.9	83.6	83.2
	DoRA		57.4M	73.6	87.1	80.8	94.4	86.1	88.8	78.3	84.2	84.2
	SBoRA-FA		25.2M	73.3	87.8	79.1	93.9	85.2	89.9	80.0	86.0	**84.4**
	SBoRA-FB		31.5M	72.9	86.3	78.8	92.6	83.0	88.8	76.3	85.0	83.0
	LoRA	64	113.2M	72.5	87.8	80.3	94.4	86.4	88.7	79.3	85.2	84.3
	DoRA		114.0M	70.5	86.0	80.3	91.8	83.7	86.2	74.7	83.2	82.1
	SBoRA-FA		50.3M	74.0	88.3	80.8	94.3	86.3	89.9	78.7	86.6	**84.9**
	SBoRA-FB		62.9M	71.8	85.2	79.2	91.4	82.9	86.7	74.0	83.4	81.8

Comparison of LLaMA-7B and LLaMA3-8B with different PEFT methods, and evaluating on the commonsense reasoning task. The first row list the results of GPT-3.5 for reference. We report the accuracy (%) results for each of the eight sub- tasks as well as the average accuracy (%), higher is better for all metrics. The column headers indicates TP for the number of trainable parameters and r for the rank.

SBoRA: Arithmetic Reasoning Performance

LLM	Method	Rank	TP	MultiArith	GSM8K	AddSub	AQuA	SingleEq	SVAMP	Avg
GPT-3.5	-	-	-	83.8	56.4	85.3	38.9	88.1	69.9	70.4
LLaMA-7B	LoRA	32	56.1M	94.5	36.3	81.8	15.0	82.7	45.6	**59.3**
	DoRA		57.0M	95.7	36.2	78.7	15.4	81.7	46.6	59.1
	SBoRA-FA		28.0M	95.5	34.6	79.7	20.1	78.9	44.8	58.9
	SBoRA-FB		28.0M	92.2	31.0	77.5	15.7	78.5	41.8	56.1
	LoRA	64	112.2M	94.0	36.8	84.3	17.3	82.3	44.7	59.9
	DoRA		113.1M	95.0	35.5	84.1	20.1	85.0	47.1	61.1
	SBoRA-FA		56.1M	97.8	36.6	85.1	19.3	83.9	48.5	**61.9**
	SBoRA-FB		56.1M	94.8	33.1	77.5	16.9	78.5	40.6	56.9
LLaMA3-8B	LoRA	32	56.6M	68.3	50.5	83.3	35.8	87.2	71.2	66.1
	DoRA		57.4M	97.3	62.0	90.9	25.6	94.9	73.4	74.0
	SBoRA-FA		25.2M	99.5	66.0	91.9	30.3	97.4	75.8	**76.8**
	SBoRA-FB		31.5M	98.0	57.2	92.2	33.9	94.1	69.6	74.2
	LoRA	64	113.2M	97.2	56.3	92.7	22.8	92.3	69.3	71.8
	DoRA		114.0M	97.8	55.2	91.1	24.0	94.7	72.0	72.5
	SBoRA-FA		50.3M	99.2	64.7	94.4	24.8	98.0	75.0	**76.0**
	SBoRA-FB		62.9M	98.2	50.9	87.1	28.0	91.7	63.0	69.8

Comparison of LLaMA-7B and LLaMA3-8B with different PEFT methods, and evaluating on the arithmetic reasoning task. The first row list out the results of GPT-3.5 for reference. We report the accuracy results for each of the sub-tasks as well as the average accuracy, higher is better for all metrics. The number of trainable parameters (TP) can be found in each row.

QSBoRA: MMLU Performance

NFloat4	LLaMA-7B			LLaMA-13B			LLaMA3-8B		
PEFT	TP	Alpaca	Flanv2	TP	Alpaca	Flanv2	TP	Alpaca	Flanv2
QLoRA	80.0M	37.9	**44.4**	125.2M	45.4	46.7	83.9M	51.9	49.5
QDoRA	80.6M	**38.0**	42.8	126.2M	46.7	48.8	84.6M	53.0	51.9
QSBoRA-FA	43.5M	36.5	43.1	68.2M	**49.0**	**51.0**	44.0M	**56.5**	**56.4**
QSBoRA-FB	36.4M	36.9	43.4	57.0M	48.3	50.5	39.8M	54.5	55.0

Finetune LLaMA-7B/13B and LLaMA3-8B on Alpaca and Flan v2. We measured the performance using MMLU benchmark and report the 5-shot average accuracy. The training settings and the number of trainable parameters (TP) are included.

SBoRA: Diffusion Model based Text-to-Image Generation

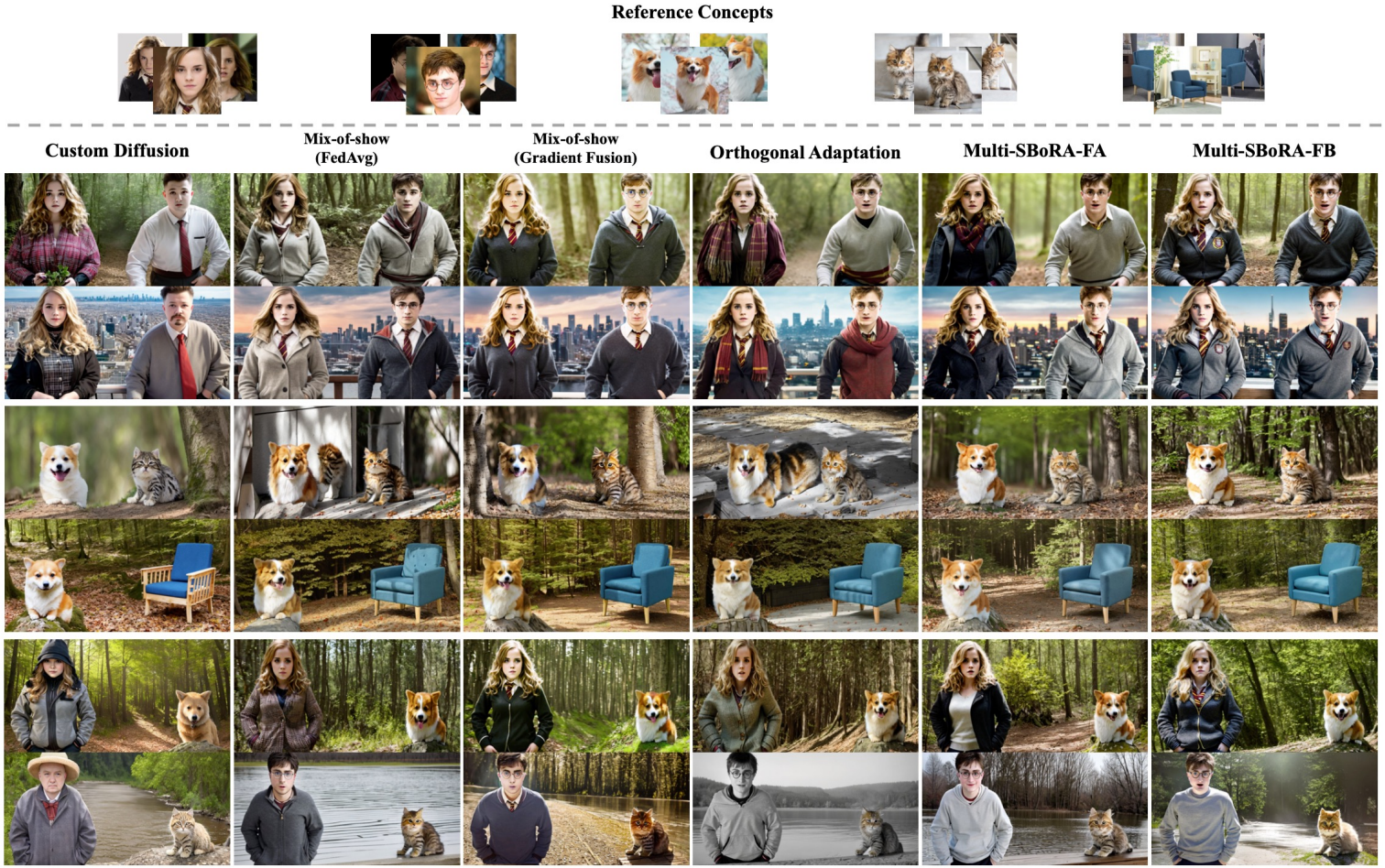
Qualitative comparison of single-concept SBoRA diffusion model image generation. Reference images for each concept is shown in the left column. LoRA-based method outperforms Custom Diffusion in terms of fidelity. Furthermore, Orthogonal Adaptation and SBoRA exhibit comparable performance to Mix-of-show, while also introducing orthogonal constraints that confer advantages in multi-concept scenarios.

SBoRA: Diffusion Model based Text-to-Image Generation

Metric	Method	Character	Object	Mean
Text Alignment	Custom Diffusion	0.7893	0.7892	0.7893
	Mix-of-show	0.7100	0.6487	0.6793
	Orthogonal Adaptation	0.7230	0.6635	0.6932
	SBoRA-FA	0.7437	0.6773	0.7105
	SBoRA-FB	0.7423	0.6929	0.7176
Image Alignment	Custom Diffusion	0.6223	0.7098	0.6661
	Mix-of-show	0.7081	0.7977	0.7529
	Orthogonal Adaptation	0.7150	0.7887	0.7518
	SBoRA-FA	0.7058	0.7851	0.7454
	SBoRA-FB	0.6910	0.7676	0.7293

Quantitative comparison result of SBoRA single concept tuning of image generation in diffusion model. Previous methods have exhibited varying performance across different concepts or metrics. Custom Diffusion, for instance, proves to be less effective in preserving image alignment, whereas Mix-of-show and Orthogonal Adaptation encounter challenges in maintaining text alignment. In contrast, our proposed method achieves comparable performance and results, demonstrating a more stable score across all concepts and metrics.

Multi-SBoRA: Diffusion Model based Text-to-Image Generation



Summary of SBoRA

- The SBoRA approach enables regional weight updates, preserving most of the pre-trained model weights while efficiently adapting to new tasks. This localized learning process draws parallels with the modular organization of the brain, where distinct cognitive functions are localized to specific brain regions. This analogy highlights the potential of SBoRA to inspire AI architectures that mimic the efficiency and adaptability of biological neural systems.
- SBoRA holds immense potential for further development, particularly in multi-task training. The introduction of Multi-SBoRA would create a powerful framework for efficient adaptation to multiple tasks. Each task would undergo independent, non-overlapping weight fine-tuning with SBoRA, allowing the inte- gration of task-specific knowledge while minimizing interference and maximizing the model's capacity to leverage shared information. This approach enables the maintenance of distinct capabilities for each task within a single model, paving the way for more efficient and effective AI systems