Introduction to Large Language Models

Spring 2026

Parameter Efficient Fine Tuning (PEFT)

(Some slides adapted from Ralph Grishman at NYU, Yejin Choi at UWashington, N. Tomura at UDepaul, Jurafsky and Martin, CS224N, CS224, CME295 at Stanford and other resourses on the web)

In-context Learning vs Fine-tuning

Before we go deeper into fine-tuning there is another way of adapting LLMs for specific task, which is called “In-context” learning.

In-context Learning	Fine-tuning
• A method of prompt engineering where the model is shown task demonstrations as part of the prompt. • No change in model parameters.	• A process of training the LLM on a labelled dataset specific to a particular task. • Change in model parameters.

Fine-tuning is a supervised process that leads to a new model, in contrast with in-context learning, which is considered “ephemeral.”

Downside of prompt–based (In-context) learning

1. **Inefficiency:** The prompt needs to be processed *every time* the model makes a prediction.

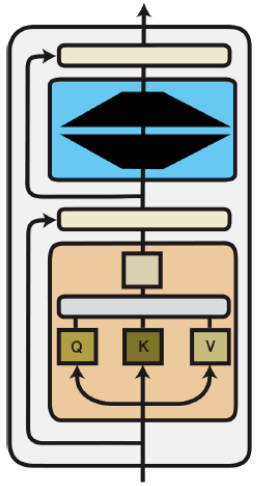
2. **Poor performance**: Prompting generally performs worse than fine-tuning [Brown et al., 2020].

3. **Sensitivity** to the wording of the prompt [Webson & Pavlick, 2022], order of examples [Zhao et al., 2021; Lu et al., 2022], etc.

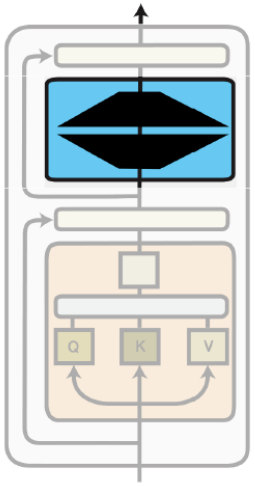
4. **Lack of clarity** regarding what the model learns from the prompt. Even random labels work [Zhang et al., 2022; Min et al., 2022]!

Fine-Tuning and PEFT

- Fine-tuning is the process of adapting a pretrained model to a specific task or domain by continuing training on task-specific data. In full Fine-Tuning, all model parameters are updated.
- PEFT modifies or adds a small number of trainable parameters while keeping the main model weights frozen.



Full Fine-tuning
Update **all model parameters**



Parameter-efficient Fine-tuning
Update a **small subset** of model parameters

Why fine-tuning *only some* parameters?

1. Fine-tuning all parameters is impractical with large models
2. State-of-the-art models are massively over-parameterized
   → Parameter-efficient fine-tuning matches performance of full fine-tuning

Instruction-tuning (Full Parameter)

Fine-tuning very often means instruction fine-tuning.

An instruction dataset, comprising pairs of instructions, answers, and sometimes context, is required for such fine-tuning.

Instruction	Context	Output
Suggest a good restaurant	Los Angeles, CA	In Los Angeles, CA, I suggest Rossoblu Italian Restaurant
Rewrite the sentence with more descriptive words	The game is fun	The game is exhilarating and enjoyable
Calculate the area of the triangle	Base: 5cm; Height: 6cm	The area of the triangle is $15\ cm^2$

This is an example of what an instruction dataset looks like.

Source: Alpaca-GPT4 dataset

Catastrophic forgetting

- We have to be careful while doing task-specific finetuning to avoid catastrophic forgetting.
- Catastrophic forgetting refers to the phenomenon where a model loses its ability to perform previously learned tasks when it is being fine-tuned on new tasks.
- The key idea of catastrophic forgetting is that as the model learns new tasks, it may overwrite what it previously learned, leading to a loss in performance on earlier tasks.
- Catastrophic forgetting arises because sequential gradient descent minimizes a series of incompatible loss functions over shared parameters, causing updates that move the model away from previously learned optima.
- Mitigation techniques succeed by either reintroducing old gradients, penalizing sensitive directions, or restricting optimization to task-specific subspaces—thereby stabilizing the optimization trajectory.

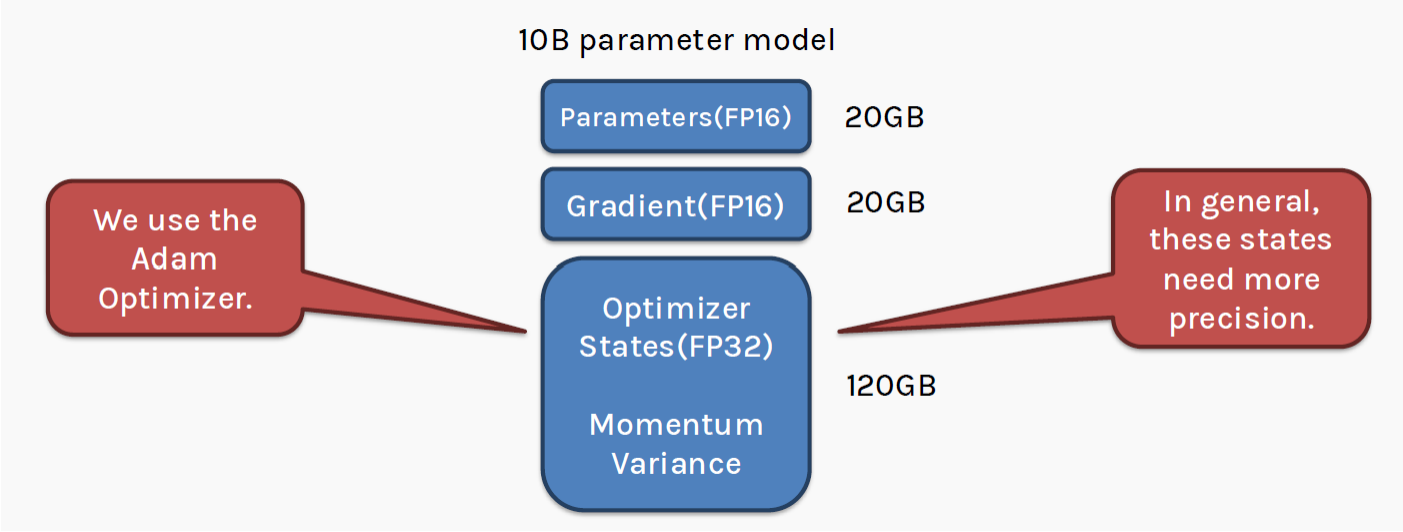
Instruction-tuning (Full Parameter)

- We need to update all the parameters while finetuning. For a 7B model, we need to update 7 billion weights.
  - For a 13 billion model, we need to update 13 billion weights.
- Storing and updating these weights require a lot of GPU memory.

Compute Power, Time and Cost:
Training GPT-4 involved ~25,000 A100 GPUs over ~ 90 - 100 days , costing OpenAI nearly $100 million !

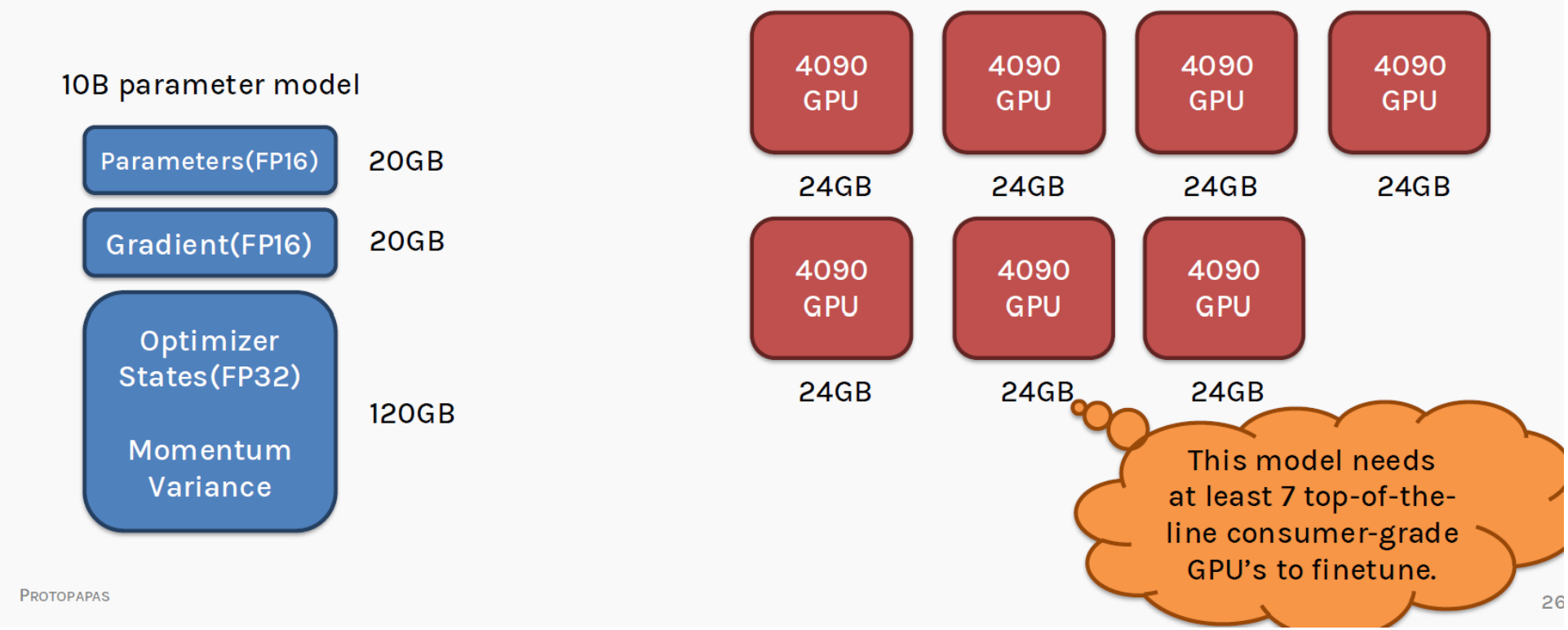
Instruction-tuning (Full Parameter)

- Fine-tuning example:
  - Say we want to finetune a 10 billion parameter model. Let's see how that looks in memory.
  - Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.
  - Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.



Instruction-tuning (Full Parameter)

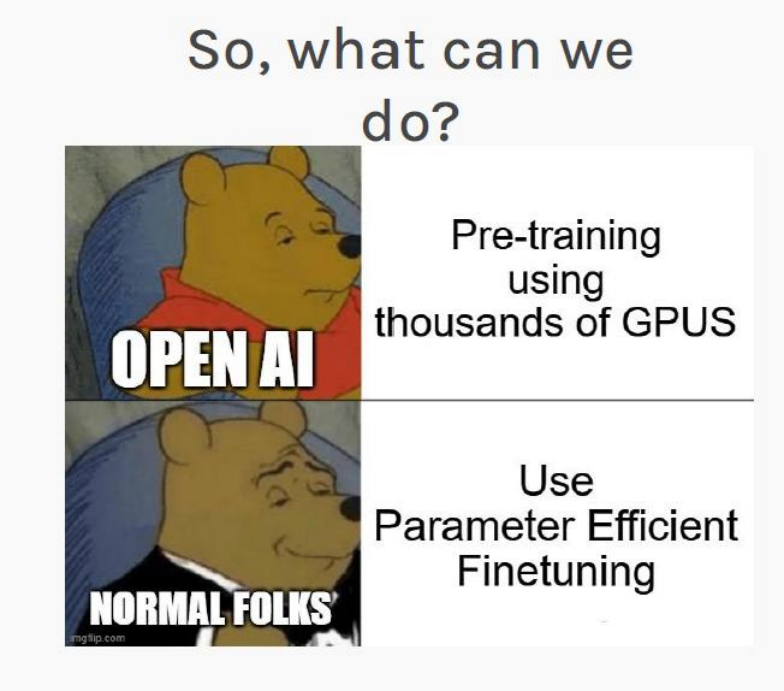
- Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.

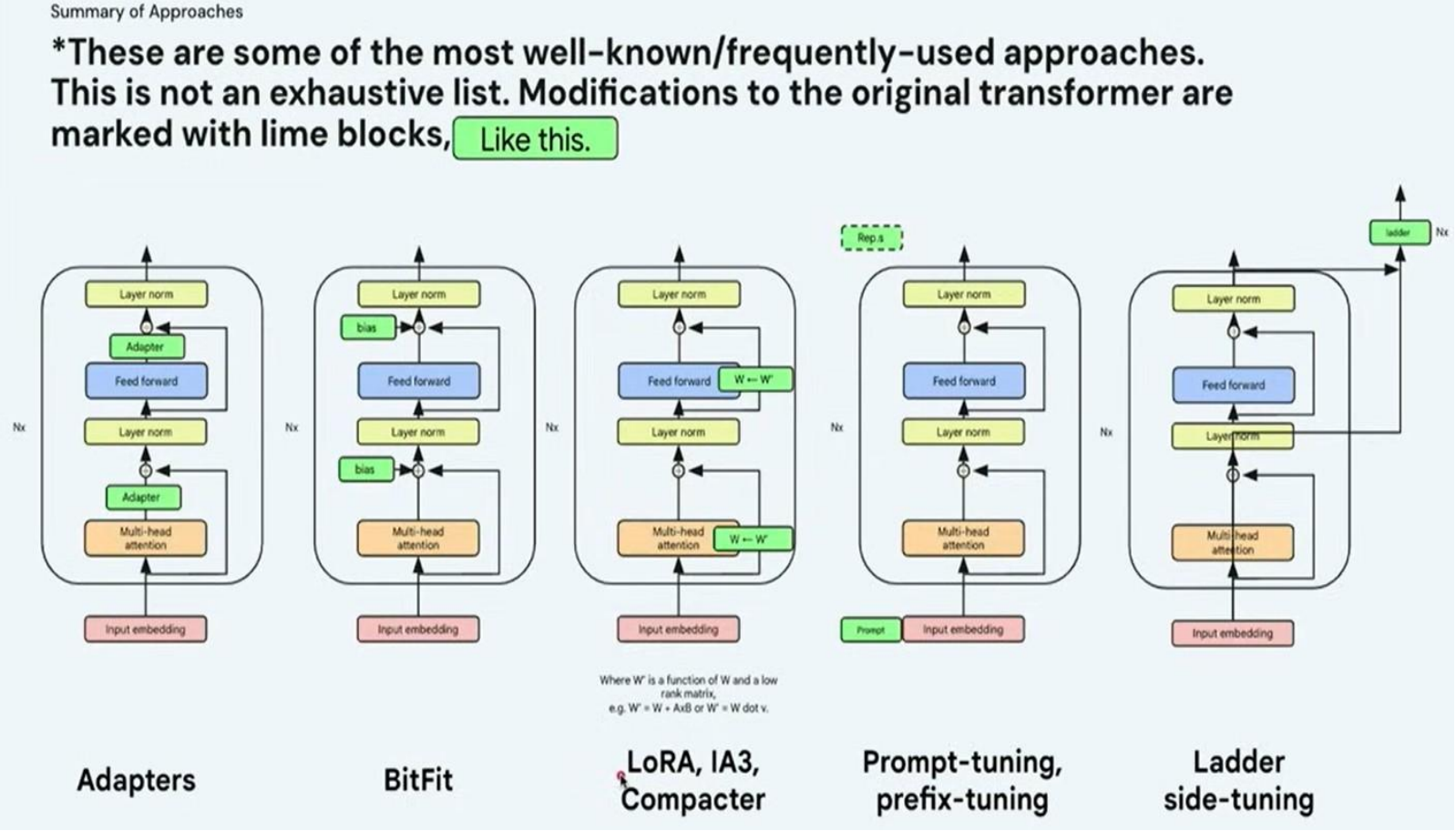


- The RTX 4090 is the best single-GPU choice for local AI research, PEFT, and LLM fine-tuning, but it is constrained by 24 GB VRAM and consumer-grade deployment limits.

Instruction-tuning (Full Parameter)

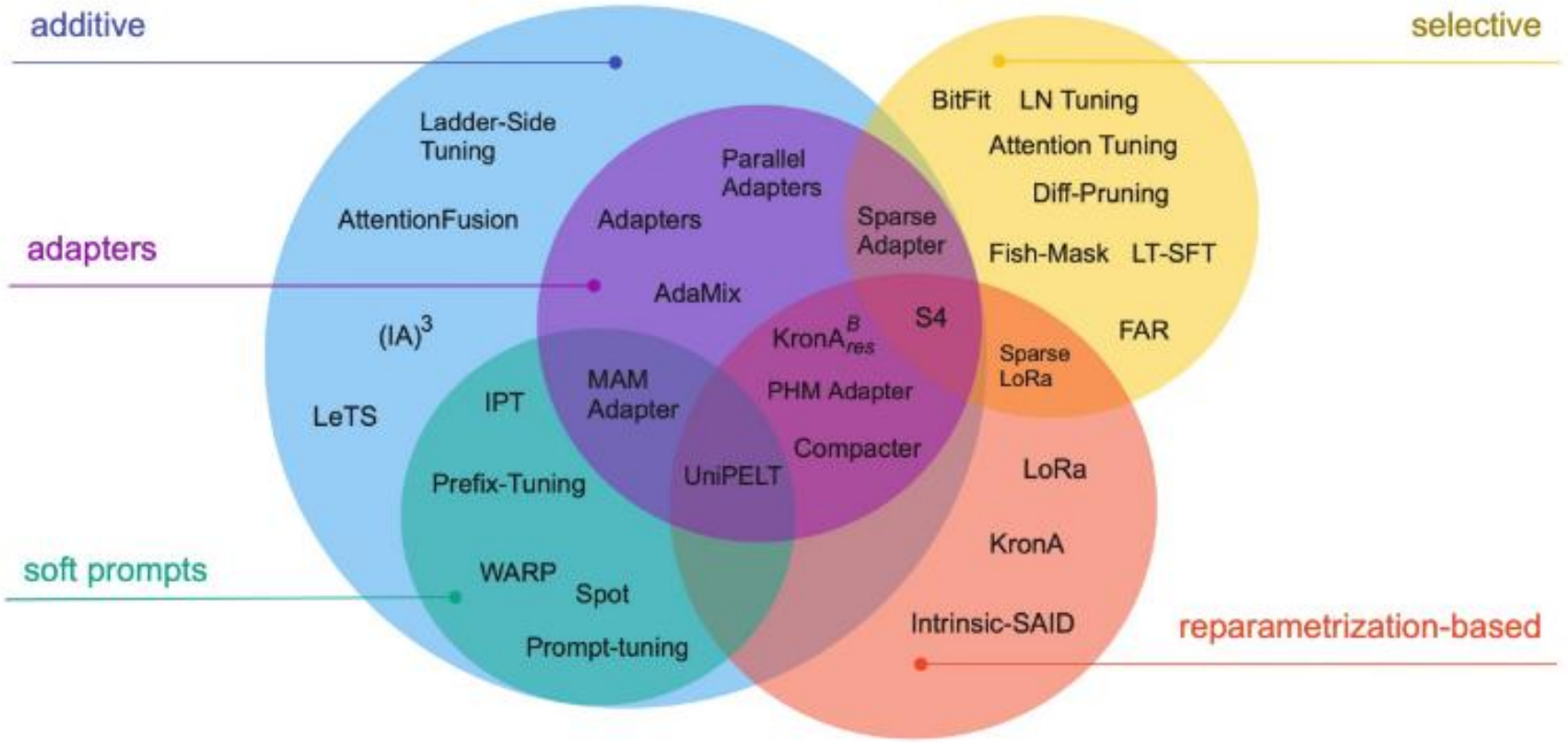
- This makes full parameter finetuning inaccessible to normal folks like us.





Categories of PEFT Methods

- Parameter-Efficient Fine-Tuning (PEFT) methods can be classified according to two main aspects: their conceptual structure (e.g., introducing new parameters or adjusting existing ones) and their primary objective (minimizing memory footprint, improving storage efficiency, or reducing computational costs).



Additive Methods

- Additive methods introduce new parameters to the base model, often through small adapter layers or by adjusting a part of the input embeddings (known as soft prompts). These methods are widely used and include:
  - **Adapters**: Small dense (fully connected) networks inserted after specific transformer sublayers, allowing adaptation to new tasks without the need to train all the model's parameters.
  - **Soft Prompts**: Fine-tuning applied directly to the model's input embeddings, enabling task-specific adaptation without modifying the model's internal parameters.
- These methods are generally memory-efficient as they reduce the size of gradients and optimizer states.

https://huggingface.co/blog/samuellimabraz/peft-methods

Selective Methods

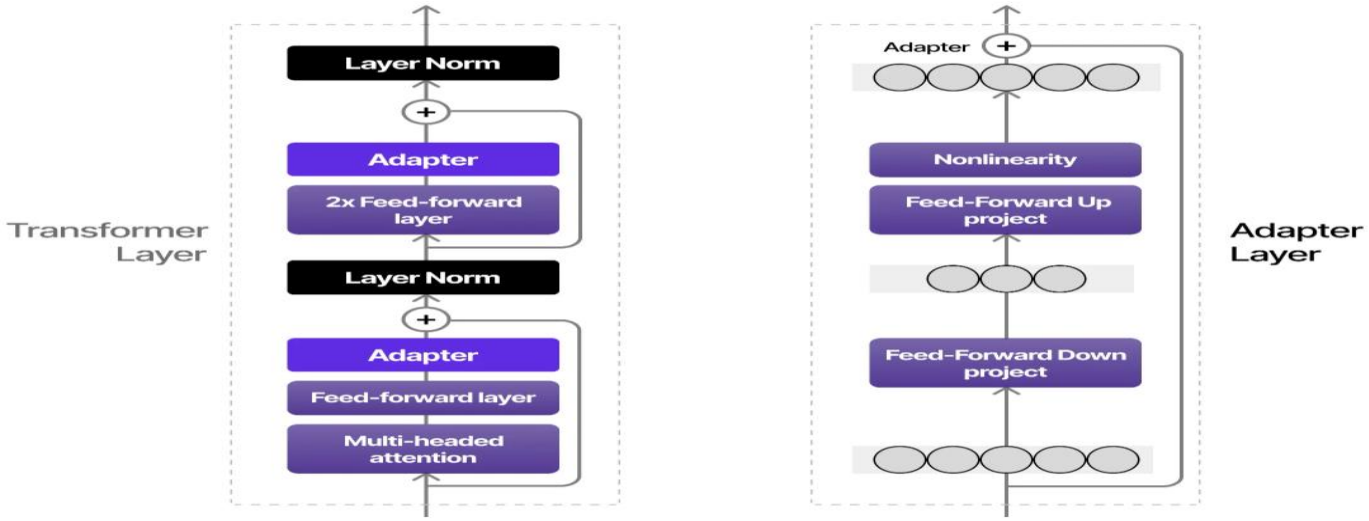
- Selective methods adjust only a fraction of the existing model parameters. This can be done in several ways, such as:
  - **Top Layer Fine-Tuning**: Focusing on fine-tuning only the upper layers of the network while leaving the lower layers untouched.
  - **Specific Parameter Fine-Tuning**: Selectively training certain types of parameters, such as biases, while freezing other parameters.
  - **Sparse Updates**: Selecting a specific subset of parameters for training. While promising, this approach can be computationally expensive due to the need to identify the most relevant parameters.
- Despite reducing the number of trained parameters, selective methods may incur high computational costs, especially in sparse configurations.

Reparameterization-Based Methods

- Reparameterization-based methods reduce the number of trainable parameters by utilizing low-rank representations, leveraging the redundancy present in neural networks. Key methods include:
  - **LoRa** (Low-Rank Adaptation): Employs low-rank matrix decomposition to represent weight updates, providing an efficient way to fine-tune models.
  - **Intrinsic SAID**: Utilizes the Fastfood transform, a technique for efficiently representing low-rank updates.
- These methods significantly reduce the number of parameters to be trained, making them ideal for scenarios where storage efficiency and training time are critical.

Adapters

- Adapter-based methods add extra trainable parameters after the attention and fully connected layers of a frozen pre-trained model to reduce memory usage and accelerate training.
- The specific implementation of the adapter may vary; it can be a simple extra layer or involve expressing weight updates ΔW as a low-rank decomposition of the weight matrix.
- In either case, adapters are typically small but demonstrate performance comparable to fully fine-tuned models, enabling the training of larger models with fewer resources.
- The concept of adapters was initially developed for multi-domain image classification (Rebuffi et al., 2017, 2018) and involved adding domain-specific layers between neural network modules. Houlsby et al. (2019) adapted this idea for NLP. They proposed adding fully connected networks after the attention and FFN layers in the Transformer architecture.




```
def transformer_block_with_adapter(x):
    residual = x
    x = self_attention(x)
    x = AdapterLayers(x)  # adapter
    x = LayerNorm(x + residual)
    residual = x
    x = FullyConnectedLayers(x)
    x = AdapterLayers(x)  # adapter
    x = LayerNorm(x + residual)
    return x
```

https://lightning.ai/pages/community/article/understanding-llama-adapters/

Soft Prompts

- Prompt methods have emerged as an efficient way to adapt pre-trained language models to specific tasks without the need for full fine-tuning (Brown et al., 2020). The concept involves providing instructions or examples to the model that guide its behavior for the desired task.
- There are two main categories of prompt methods:
  - **Hard Prompts:** Consist of manually created natural text that instructs the model about the task. For example: "Translate the following text to French:" or "Classify the sentiment as positive or negative:". Although intuitive, they require significant expertise to create effective prompts.
  - **Soft Prompts:** Utilize continuous and trainable vectors that are concatenated to input embeddings. Unlike hard prompts, these "virtual tokens" are automatically optimized for the task but are not human-interpretable as they do not correspond to real words. (Li and Liang, 2021; Lester et al., 2021; Liu et al., 2021)

$$[\ P1\ \ P2\ \ \ldots\ \ Pm\ ] + [\ x1\ x2\ \ldots\ xn\ ] \rightarrow \text{Transformer}$$

Prompt-Tuning (Soft Prompt Learning)

For each input sequence x:

```
[ P1, P2, ..., Pm ] + [ x1, x2, ..., xn ]
```

- Prompt vectors come **before** the actual input
- This concatenated sequence is fed into the Transformer
- The model treats prompt vectors as "virtual tokens"
- They are learned vectors in embedding space

```
freeze(model)
P = initialize_prompt_vectors()

for batch in data:
    inputs = concat(P, embed(batch.text))
    outputs = model(inputs)
    loss = task_loss(outputs, batch.labels)
    update(P, loss)
```

LoRA (Low-Rank Adaptation)

- **LoRA** is a **parameter-efficient fine-tuning (PEFT)** method that adapts large pretrained Transformer models by training **only a small number of additional parameters**, while keeping the original model weights **frozen**.

- **Core idea (intuition) :**
  Task-specific changes to a large model can be expressed in a low-rank subspace. So instead of updating a huge weight matrix, LoRA learns a **low-rank update** to it.

LoRA (Low-Rank Adaptation)

How it works (mechanism)

For a weight matrix $W \in \mathbb{R}^{m \times n}$:

- **Full fine-tuning:** learn a full update ΔW (costly).
- **LoRA:** re-parameterize the update as

$$W' = W + \Delta W, \Delta W = AB$$

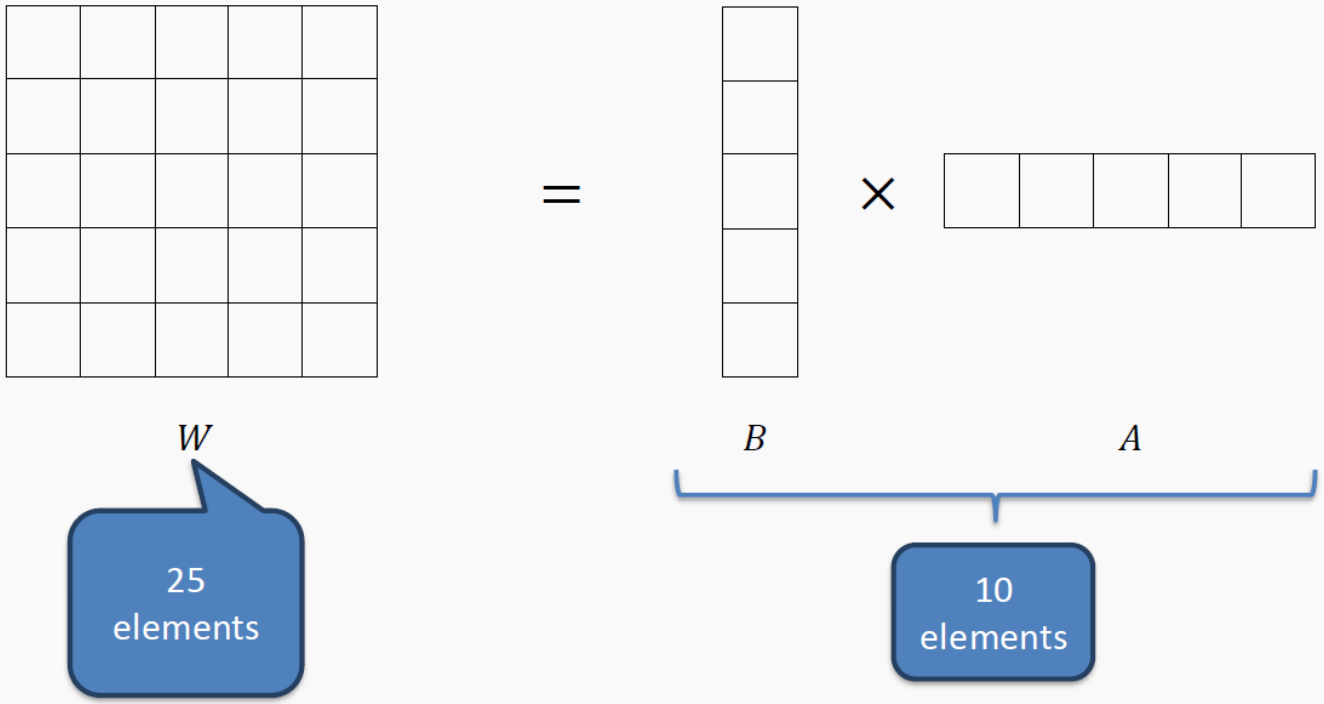
where:

- $A \in \mathbb{R}^{m \times r}$
- $B \in \mathbb{R}^{r \times n}$
- $r \ll \min(m, n)$ (the **rank**)

Only A and B are trained; W stays fixed.

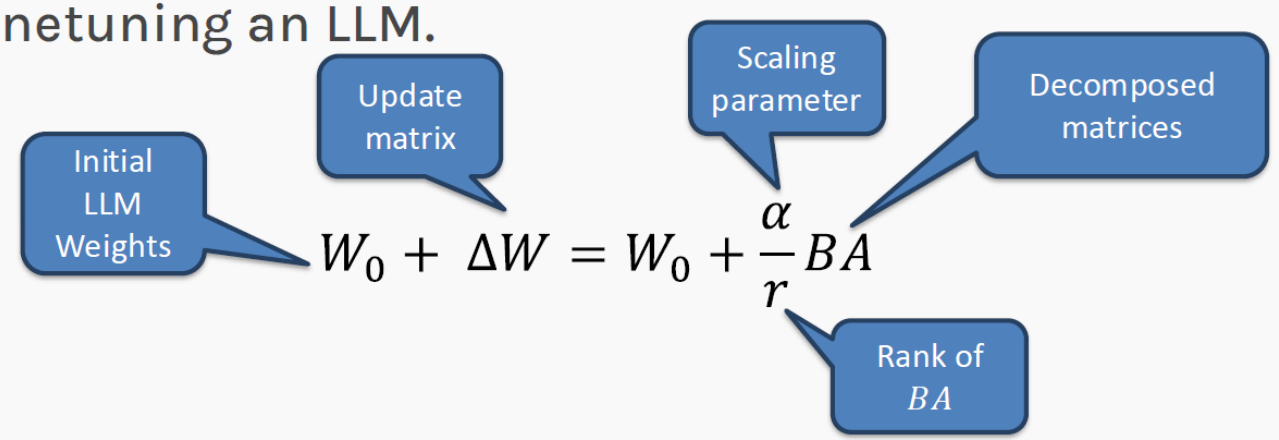
LoRA (Low-Rank Adaptation)

LoRA revolves around the idea that any matrix $W \in R^{m \times n}$ can be decomposed into $W = BA$ where $B \in R^{m \times r}$ and $A \in R^{r \times n}$



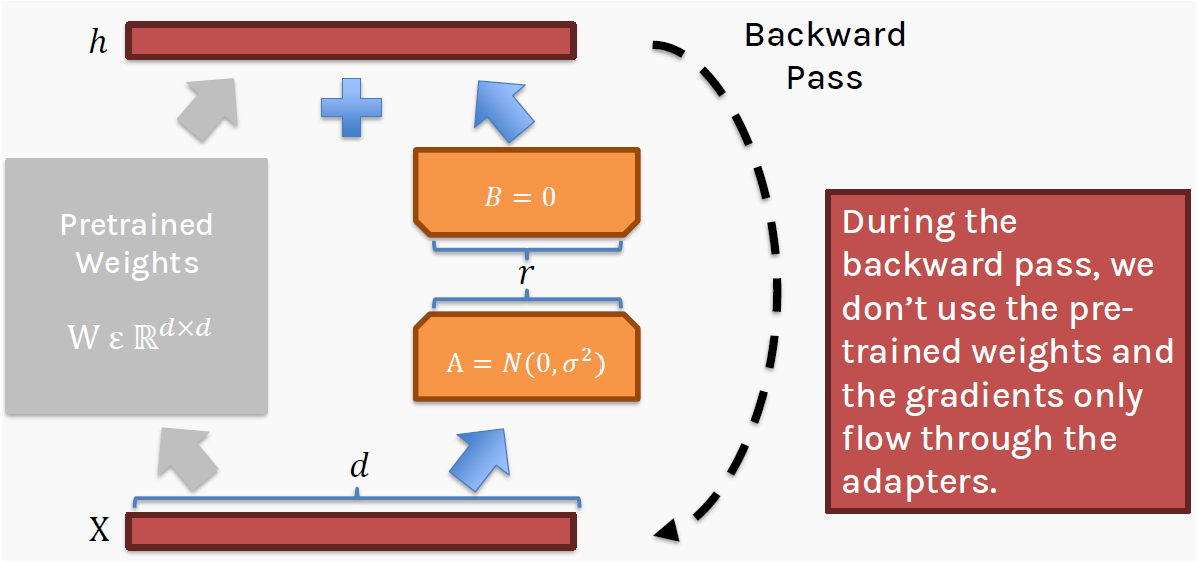
LoRA (Low-Rank Adaptation)

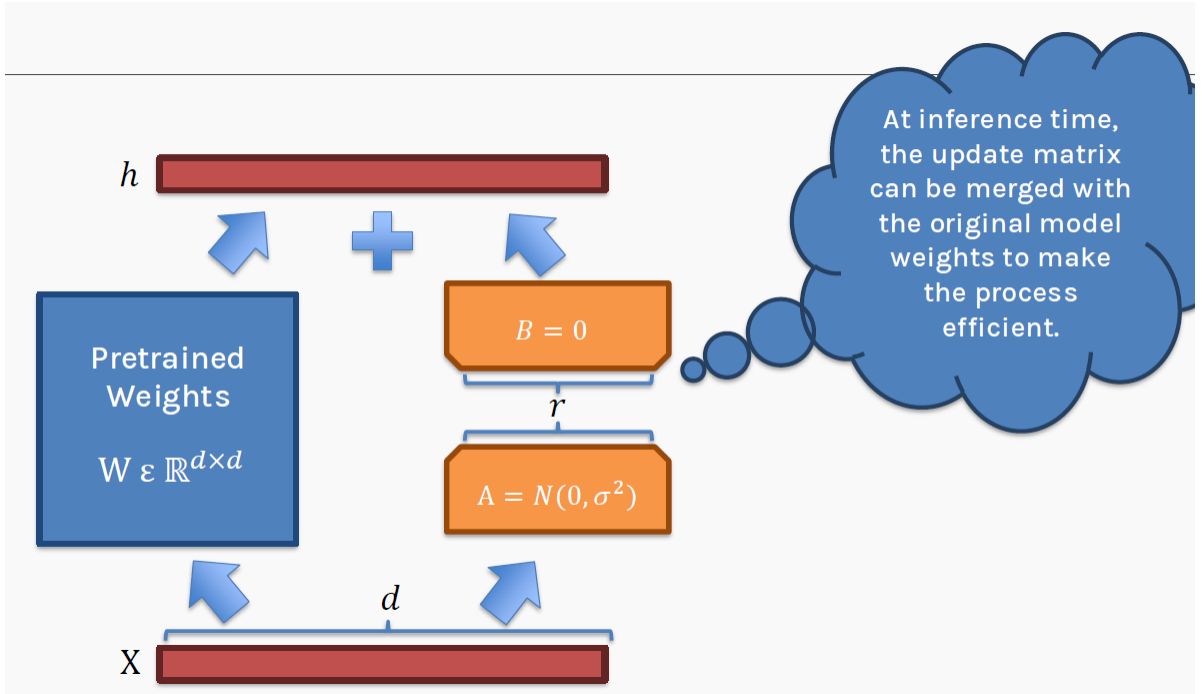
Now, we use the same concept of matrix decomposition while finetuning an LLM.



Remember, we are decomposing the update matix (ΔW), and not the original weights W_0.

LoRA (Low-Rank Adaptation)





- Reparameterization (LoRA) runs parallel to the original model.

LoRA (Low-Rank Adaptation)

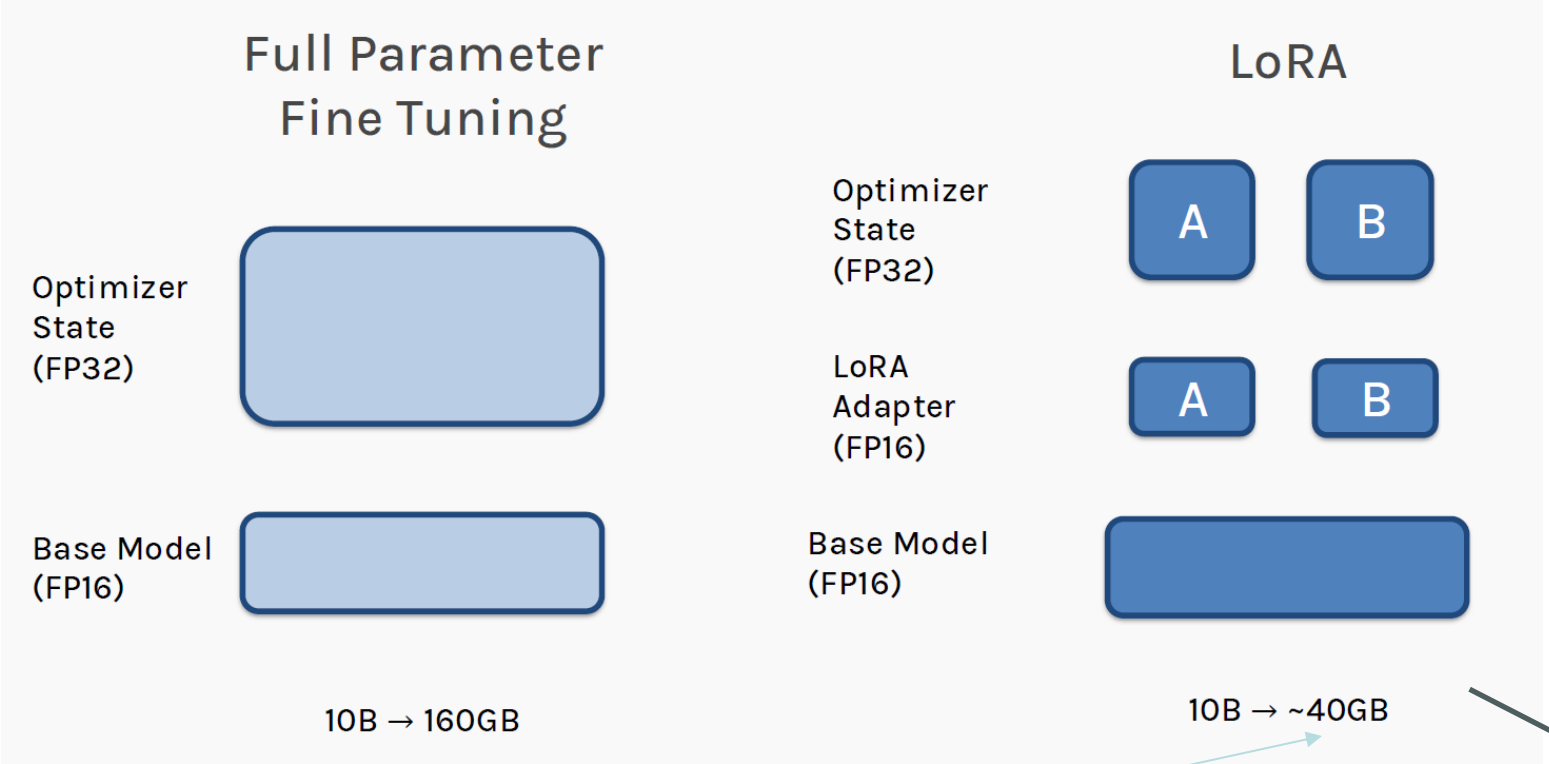
- Number of trainable parameters depending on rank r. (This is a generalization considering an LLM of one layer. LLMs are made up of multiple layers.

Rank	Model 7B	Model 13B	Model 70B	Model 180B
1	167K	228K	529K	849K
2	334K	456K	1M	2M
8	1M	2M	4M	7M
16	3M	4M	8M	14M
512	86M	117M	270M	434M
1024	171M	233M	542M	869M
8192	1.4B	1.8B	4.3B	7B
Full	7B	13B	70B	180B

LoRA - Advantages

- Compared to full parameter finetuning, LoRA has the following advantages:
  - Much faster
  - Finetuning can be achieved using less GPU memory
  - Cost efficient
  - Less prone to “catastrophic forgetting” since the original model weights are kept the same.

LoRA - Advantages



LoRA reduces the trainable parameters and memory requirements while maintaining good performance.

This will be frozen. So, no optimization, but the parameters still needs to be stored in memory for forward pass

LoRA – Isn't it enough?

- As we can see below, LoRA's performance is comparative to full parameter fine-tuning and, in some cases, even outperforms it.

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

Table 3: GPT-2 medium (M) and large (L) with different adaptation methods on the E2E NLG Challenge. For all metrics, higher is better. LoRA outperforms several baselines with comparable or fewer trainable parameters. Confidence intervals are shown for experiments we ran. * indicates numbers published in prior works.

These metrics are used for performance evaluation.

LoRA - Summary

- LoRA reduces the trainable parameters and memory requirements while maintaining good performance.
- •LoRA adds pairs of rank decomposition weight matrices (called update matrices) to each layer of the LLM.
- •Only the update matrices, which have significantly fewer parameters than the original model weights, are trained.

Quantization

- **Quantization is the process of representing model weights and activations using fewer bits (e.g., 4-bit or 8-bit instead of 16- or 32-bit) to reduce memory usage and speed up computation.**

  In simple terms: **store numbers more compactly**.

Why Quantization is needed

Large language models are huge:

- FP32 (32-bit): very accurate, very expensive
- FP16/BF16 (16-bit): standard for training
- INT8 / INT4 (8-bit / 4-bit): much cheaper

Problems quantization solves:

- GPU memory limits
- Bandwidth bottlenecks
- Deployment cost
- Single-GPU fine-tuning (e.g., RTX 4090)

Quantization Mapping

Quantization turns real-valued numbers into a **small, discrete set of representable values**. The mapping almost always follows:

1. **Scaling** → normalize the range
2. **Clipping** → limit outliers
3. **Codebooks** → choose representable values (advanced)

Common Quantization Levels

Precision	Bits	Use
FP32	32	Pretraining, research
FP16 / BF16	16	Standard training
INT8	8	Inference
INT4	4	Memory-critical inference & QLoRA

FP16 (IEEE half-precision)

```
| Sign (1) | Exponent (5) | Mantissa (10) |
```

BF16 (bfloat16)

```
| Sign (1) | Exponent (8) | Mantissa (7) |
```

- BF16 (bfloat16) is a 16-bit floating-point format widely used in deep learning training and fine-tuning because it preserves numerical stability while cutting memory and compute cost in half compared to FP32.

1. Scaling (range normalization)

What scaling does

Scaling maps floating-point values into a **normalized numerical range** that fits the limited precision (e.g., 8-bit or 4-bit).

Why scaling is necessary

Low-bit formats can only store values in a small range, e.g.:

- INT8 → −128 to +127
- INT4 → −8 to +7 (or similar)

Scaling ensures the original values fit into that range.

Conceptual formula

For a weight w:

$$w\_\text{scaled} = \frac{w}{s}$$

where:

- s is a **scale factor**

2. Clipping (outlier control)

What clipping does

Clipping **limits extreme values** that would otherwise dominate the scaling factor and ruin precision.

$$w\_\text{clipped} = \min(\max(w, -T), T)$$

where:

- T is a clipping threshold

Why clipping is critical

LLM weights:

- Are mostly small
- Have rare but large outliers

If you scale to accommodate outliers:

- Most weights collapse into too few quantization bins
- Precision is wasted

Clipping trades:

- **Small error on rare outliers**
- for **much better precision on most weights**

Intuition

"Sacrifice a few extreme values to save accuracy everywhere else."

This is one reason quantization works surprisingly well.

3. Codebooks (advanced quantization)

What a codebook is

A **codebook** is a predefined set of allowed values. Each stored number is an **index into the codebook**, not a raw integer.

Example (4-bit → 16 values):

```
Index → Value
0     → -1.23
1     → -0.85
...
15    → +1.14
```

How codebooks are used

Instead of:

round(w_scaled)

We do:

index = argmin |w − codebook[i]|

This allows:

- Non-uniform spacing
- Distribution-aware precision

NF4 = Scaling + Clipping + Codebook (perfectly aligned)

- NF4 (NormalFloat4) is a 4-bit, non-uniform quantization format designed specifically for large language model (LLM) weights. It is the key enabling technology behind QLoRA, allowing stable fine-tuning of very large models on limited hardware.

NF4 uses **all three**:

Scaling

- Normalize weights block-wise

Clipping

- Implicit via distribution modeling

Codebook

- 16 non-uniform values optimized for a **Gaussian distribution**

This is why NF4 works so much better than plain INT4.

NF4

NF4 actually is

NF4 = NormalFloat4

- 4-bit quantization
- **Non-uniform codebook**
- Codebook optimized for a **standard normal distribution**

Instead of integers like `−8 … +7`, NF4 uses **16 carefully chosen floating-point values** that approximate a Gaussian.

NF4 is still 4 bits, but **not integers**.

How NF4 matches LLM weight statistics

Empirical observation

Across many pretrained LLMs:

- Weight distributions ≈ Gaussian
- Zero-mean, small variance
- Most values clustered tightly near 0

NF4 design

- **High resolution near zero**
- **Lower resolution in the tails**
- Minimizes expected quantization error under a normal distribution

Visually (conceptually):

```
Uniform INT4 bins:
|----|----|----|----|----|----|

NF4 bins:
||  |||  |||||  ||||||  |||||  |||  ||
```

Precision is placed **where it matters most**.

Typical precision mix in modern LLM training

Component	Precision
Model weights	BF16
Activations	BF16
LoRA parameters	BF16
Optimizer states	FP32
Backbone (QLoRA)	NF4

QLoRA

- **QLoRA (Quantized Low-Rank Adaptation)** is a technique that lets you **fine-tune very large language models on limited hardware** by combining **aggressive quantization** with **LoRA adapters**—without sacrificing much performance.

1. One-line intuition

QLoRA freezes a 4-bit quantized backbone and learns tiny LoRA updates in higher precision, achieving near full fine-tuning quality at a fraction of the memory cost.

2. Why QLoRA exists

- Full fine-tuning a 13B–70B model in FP16 requires **tens to hundreds of GB of VRAM**
- Even LoRA still needs the **base model loaded in memory**
- **QLoRA solves this by quantizing the base model to 4-bit** while keeping learning in small, precise adapters

Result: **single-GPU fine-tuning** (e.g., RTX 4090).

How QLoRA works (conceptual pipeline)

Step 1: Quantize the backbone to 4-bit

- Base model weights → **4-bit NF4** (NormalFloat4)
- Backbone is **frozen** (never updated)
- Memory drops by ~4× vs FP16

Step 2: Insert LoRA adapters

For selected layers (typically attention projections):

$$W \leftarrow W + AB \, (\text{rank } r \ll d)$$

- W: frozen, quantized weights
- A, B: **trainable**, low-rank matrices (rank 8–64 common)

Step 3: Mixed-precision training

- Backbone: **NF4 (4-bit)**
- LoRA params: **BF16 / FP16**
- Optimizer states: **FP32**

Step 4: Backprop only through LoRA

- Gradients flow **through** the quantized backbone
- Updates apply **only** to LoRA
- Backbone remains unchanged → **no forgetting**

QLoRA

Why NF4 (NormalFloat4) matters

Plain INT4 is uniform and often harms LLMs. **NF4** is:

- **Non-uniform** (codebook-based)
- Optimized for **normally distributed weights** (LLMs ≈ Gaussian)
- Places more precision **near zero**, where most weights are

This is why **4-bit training is stable** in QLoRA.

Memory intuition (single GPU)

Approximate weight storage:

- **FP16**: 2 bytes/weight
- **INT8**: 1 byte/weight
- **NF4**: 0.5 byte/weight

So a 70B model:

- FP16 ≈ 140 GB (impractical)
- NF4 ≈ ~35 GB + small LoRA (practical with tricks like checkpointing/offload)

QLoRA

Performance and stability

- QLoRA ≈ LoRA ≈ full fine-tuning on many tasks
- Often **better generalization** due to:
  - Frozen pretrained features
  - Low-rank regularization
  - Static quantization noise (no accumulation)

Why QLoRA avoids catastrophic forgetting

- Backbone weights are **immutable**
- Learning happens in a **separate low-rank subspace**
- No destructive overwrites of pretrained knowledge

QLoRA vs LoRA

Aspect	LoRA	QLoRA
Backbone precision	FP16/BF16	**4-bit NF4**
Trainable params	Low	Low
Memory use	Medium	**Very low**
Single-GPU 70B	❌	✅
Quality	Excellent	Excellent

When to use QLoRA

✅ You have **one GPU** (e.g., RTX 3090/4090)
✅ You want to fine-tune **13B–70B** models
✅ You care about **memory efficiency** and stability

❌ Avoid if you're doing **full FP16 multi-GPU training** or need to update all weights.

Takeaway

QLoRA makes large-scale LLM fine-tuning accessible by compressing what's stored (4-bit) and precisely learning what matters (low-rank adapters).

QLoRA – The Ingredients

- There are 3 key ingredients which helps us make QLoRA:

Double Quantization in QLoRA

The hidden memory problem in large LLMs

For LLMs, quantization is usually **block-wise**:

- Split weights into small blocks (e.g. 64 weights)
- Each block gets **its own scale**

Example

For a 70B model:

- Billions of weights
- **Millions of scale values**

Even though:

- Weights are 4-bit
- Scales are often FP16 or FP32

👉 **Scales start to consume noticeable memory**

This becomes a bottleneck at large scale.

Double Quantization in QLoRA

First quantization (weights)

- Weights → **4-bit NF4**
- Buckets chosen via codebook
- Each block has a scale

Second quantization (scales)

- The **scale values themselves** are quantized
- Typically to **8-bit**
- Using another scale (meta-scale)

Memory impact (why it matters)

Without double quantization:

- Weights: 4-bit ✅
- Scales: 16–32 bit ❌

With double quantization:

- Weights: 4-bit ✅
- Scales: **8-bit** ✅

This further reduces VRAM, enabling:

- 65B–70B models
- Single-GPU training
- RTX 3090 / 4090 setups

Paged Optimizer

- **Paged Optimizer** (often called **Paged Adam / Paged Optimizer**) is a **memory-management technique** used during training—especially with **QLoRA**—to **avoid GPU out-of-memory (OOM) errors** by **paging optimizer states between GPU and CPU memory on demand**.

Core idea of a Paged Optimizer

Instead of keeping **all optimizer states on the GPU**, a paged optimizer:

1. Stores optimizer states **in CPU memory**
2. Divides them into **pages**
3. Moves only the **active pages** to GPU when needed
4. Evicts unused pages back to CPU automatically

This is very similar to **OS virtual memory paging**

Paged Optimizer

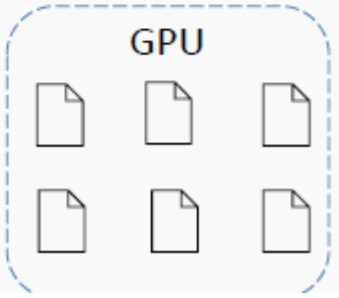
Paged Optimizer - Looping in your CPU

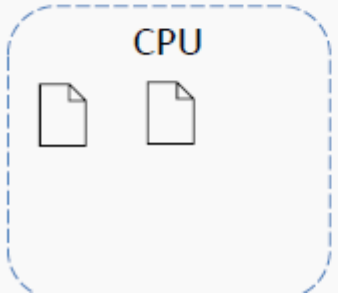
Paging is a memory management technique, where RAM is divided into fixed-size blocks called 'pages'

It does automatic page-to-page transfers between CPU and GPU

Avoids the gradient checkpointing memory spikes that occur when processing a mini batch with a long sequence length.

GPU Memory has space now.





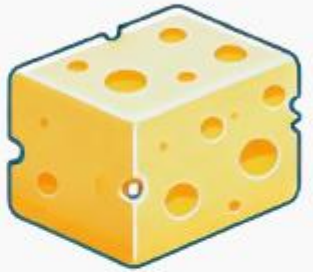
Now that the GPU has space, when a page moved to CPU is required, we move it back to GPU for computation.

QLoRA – The Ingredients

We saw the 3 key ingredients needed to make QLoRA:

4-Bit NormalFloat

Double Quantization



Paged Optimizer

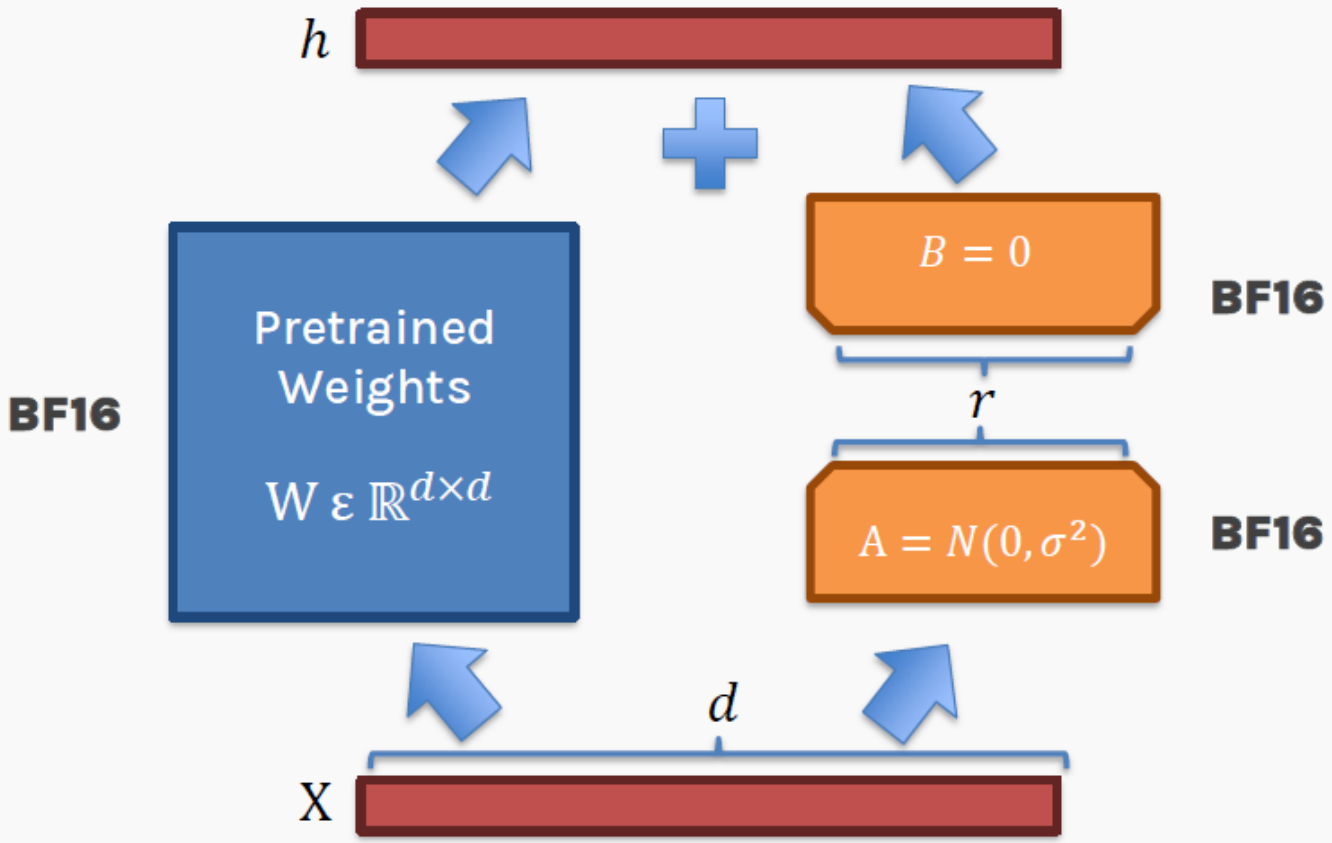
Let's bring it all together.

QLoRA – Putting it all together

Before we talk about the 3 ingredients, there is another key difference that we should know.

In QLoRA we use BF16 (BrainFloat16) as compared to FP16 in LoRA.

This leads to a change in precision which is tailor-made for deep learning tasks.



QLoRA – Putting it all together

Ingredient 1:

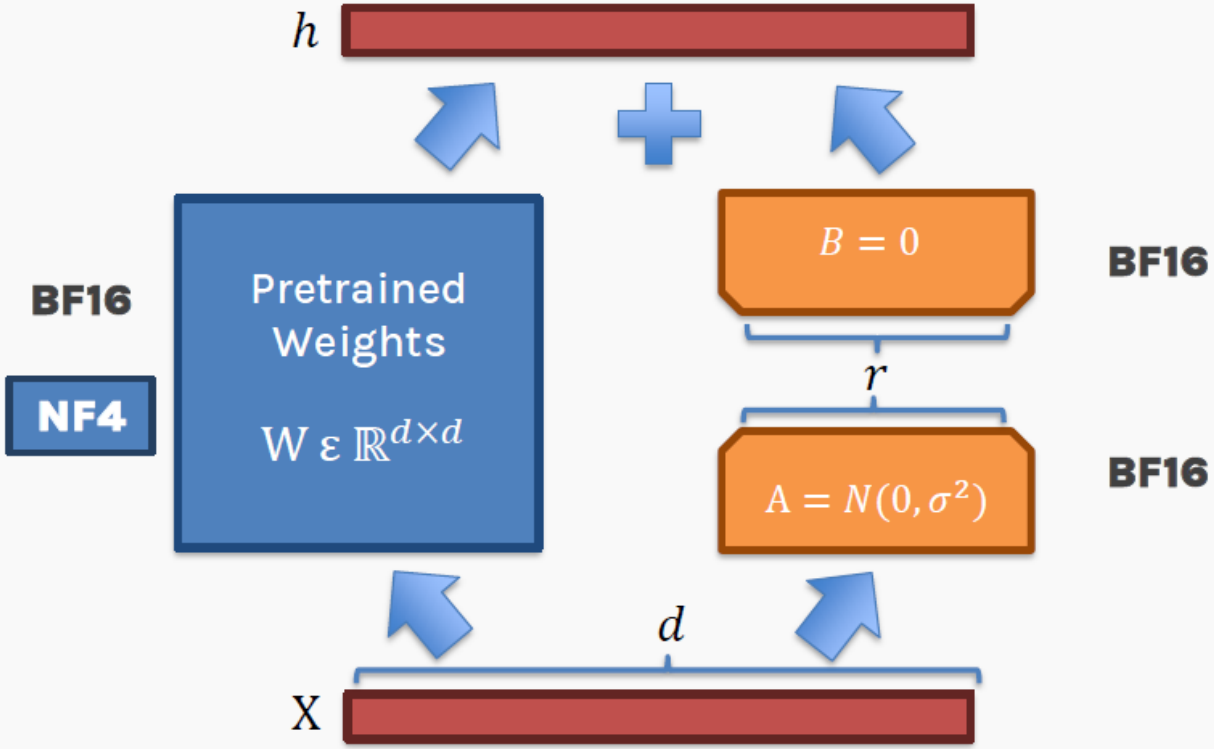
4-Bit NormalFloat

We store W, as 4-Bit NormalFloat

Ingredient 2:

Double Quantization

To convert and store, we make use of Double Quantization!



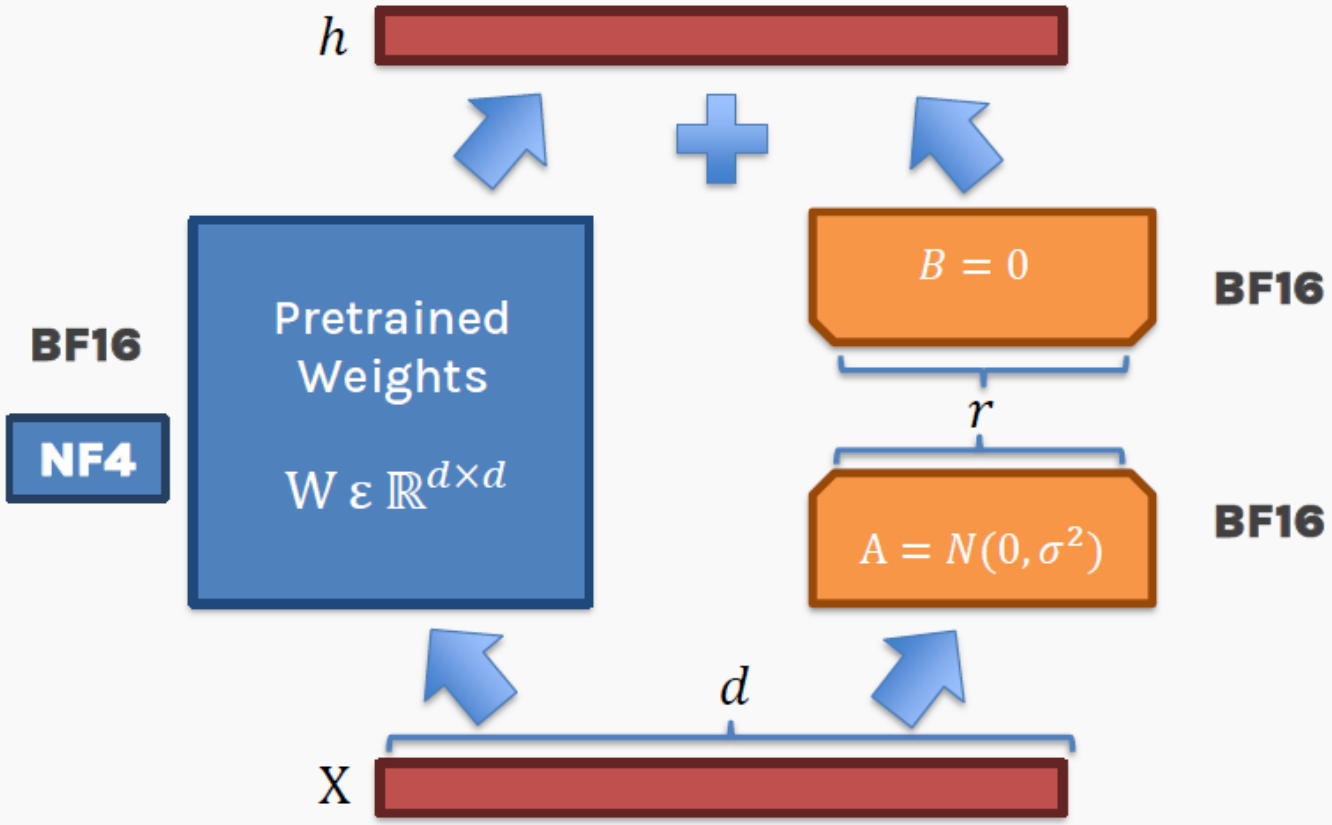
QLoRA – Putting it all together

Forward Pass

During the forward pass, we first dequantize the W weights from NF4 to BF16 for computation.

We then use the BF16 values of W, A and B to perform the required calculations.

The BF16 values of W is then deleted to save on storage!

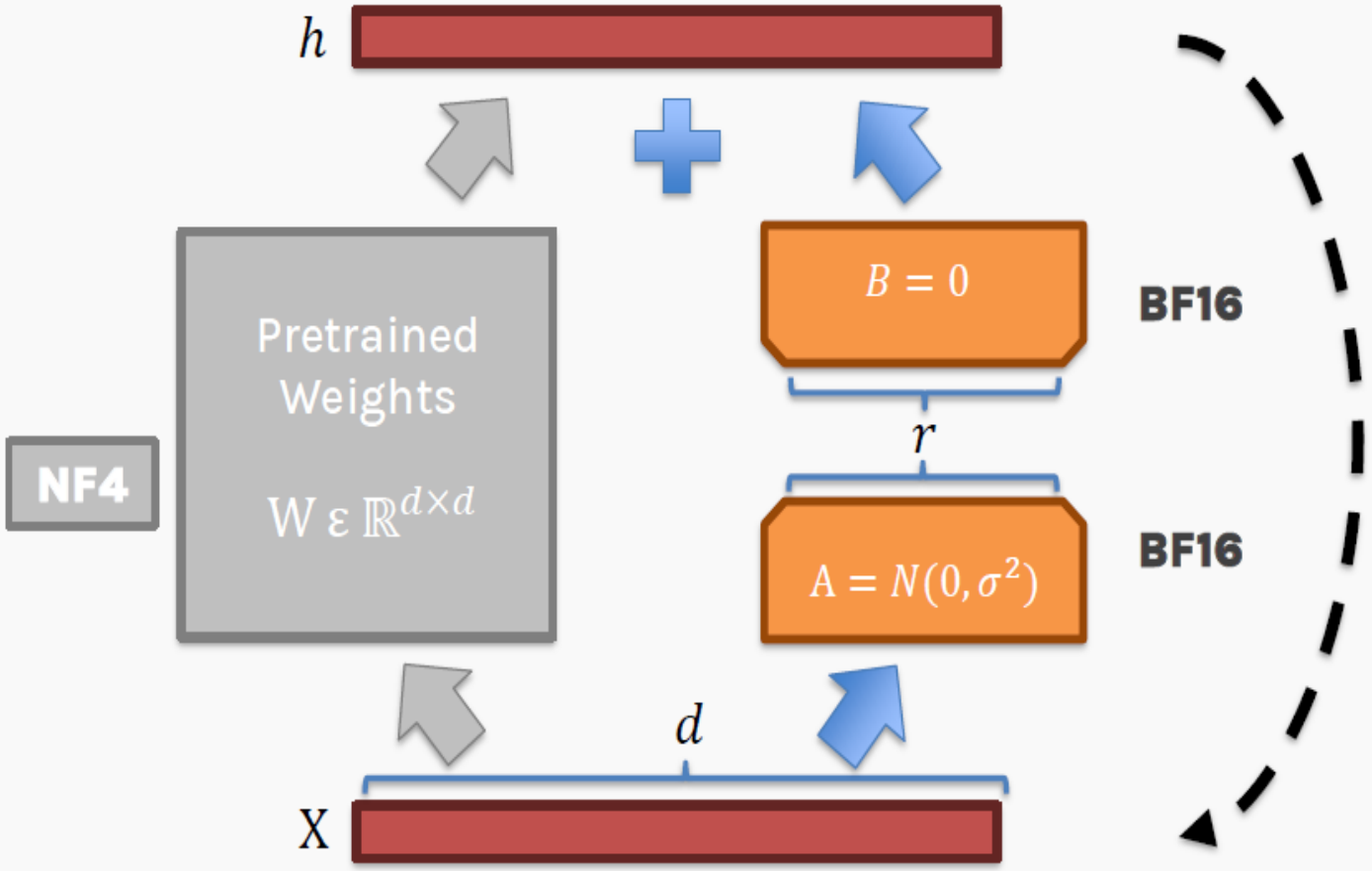


QLoRA – Putting it all together

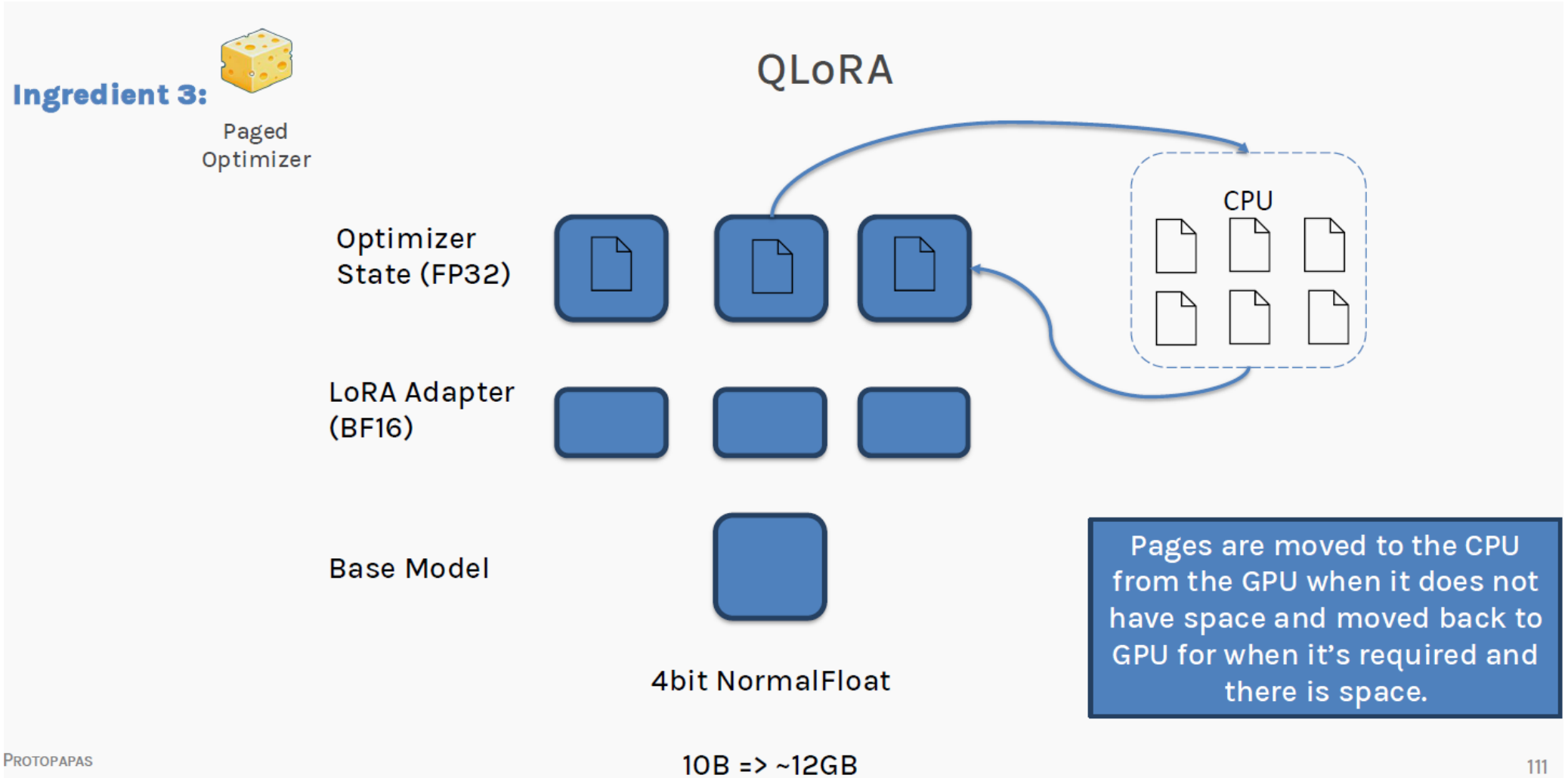
Backward Pass

As in LoRA, we keep W weights frozen and allow the gradients to only flow through the adapters.

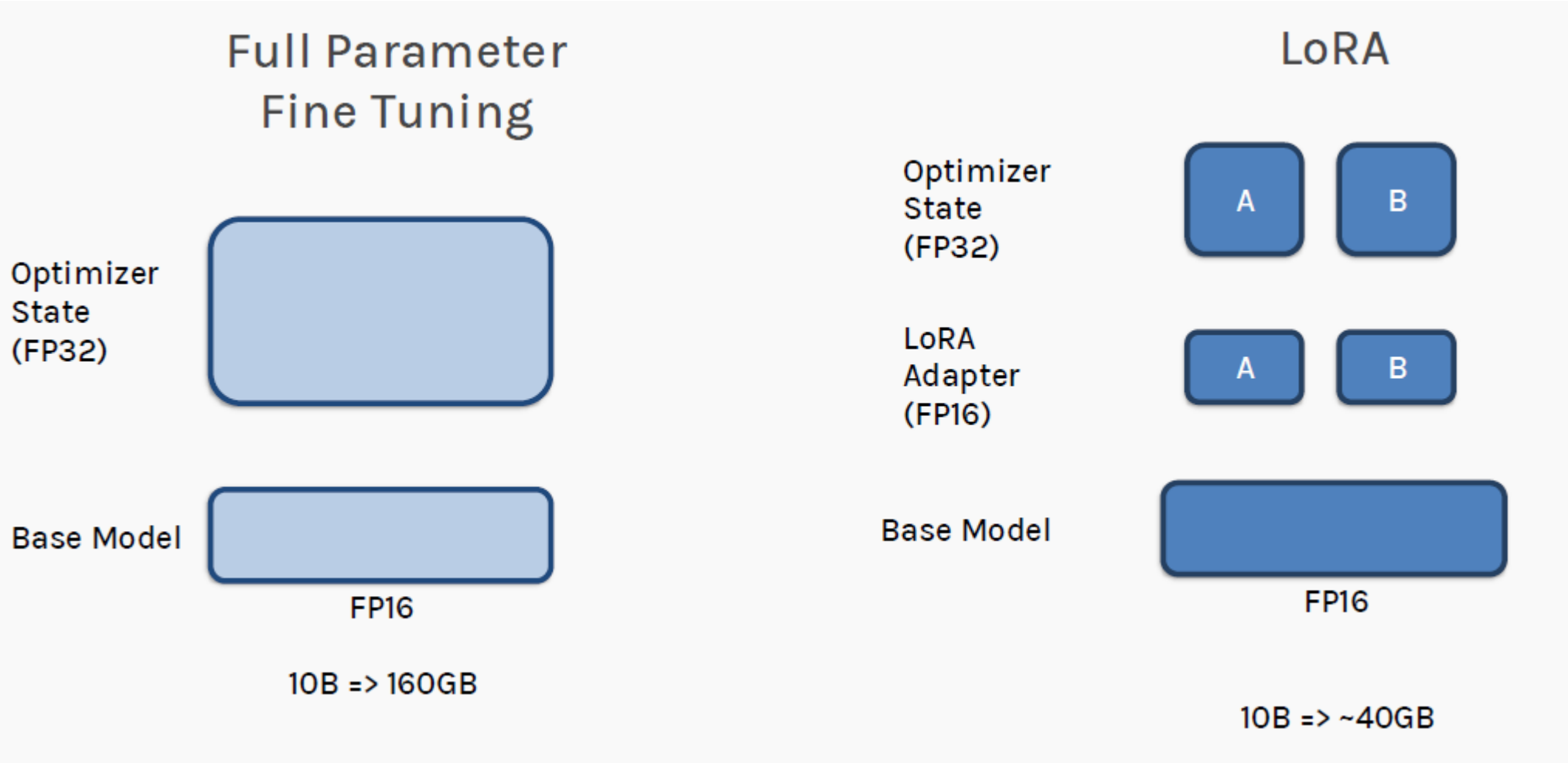
We then repeat the cycle of forward and backward passes till a minima is reached.



QLoRA – Putting it all together



Lora Memory Requirement



THANK YOU FOR YOUR ATTENTION