PEFT

Parameter Efficient Fine Tuning

What is Fine Tuning?

It is the process of tuning a pre-trained model on a specific task by providing your data. By fine-tuning we can enhance the model capability of generalization to gain new skills specific to your dataset.

Why Fine-Tune a model?

- Pre-trained models are good in common tasks but might not fit according to your specific task.
- For example, you picked up a model that is good in text summarization but your task is specific to the medical domain. In this case, you will not retrain your model on a new medical dataset. Instead, prepare a medical dataset and fine-tune the model only on your specific task.

Challenges

1. **Memory & Computational Cost**

- LLM weights are in the Billions and fitting this amount of parameters in memory is not easy.
- To give you a glimpse of it, the approximate GPU RAM needed to store 1B parameters is
- 1 parameter = 4 bytes (32-bit float)
- 1B parameters = 4 x 109 bytes = 4GB
- Just think about how much memory is required to fine-tune a model with >100B model.

Challenges

2. **Catastrophic Forgetting**

- Catastrophic forgetting occurs when a machine learning model forgets previously learned information as it learns new information.
- Let's continue the example of fine-tuning a model for medical text summarization tasks.
- If we fine-tune our model just for medical text summarization tasks then our model fails to generalize on other tasks like translation, question answering, etc.
- Fine-tuning updates **all model weights**, which **overwrites** useful pre-trained knowledge.
- A model fine-tuned on medical text might **lose its ability** to answer basic history or math questions.

PEFT

- Instead of fine-tuning an entire model isn’t it good to just fine-tune only a subset of weights?
- PEFT implement the same idea that instead of tuning all weights of the model just fine-tune the subset of it while achieving the accuracy like full fine-tuning.
- There are different methods of fine-tuning in PEFT technique. Few of them are
  - **Selective**
  - **Reparameterization**
  - **Additive**

1. **Selective**: Determine which parameters, layers, or individual parameters to update as per your task.
2. **Reparameterization**: Reparametrize model weights using a low-rank matrix representation (LoRA).

- **Additive**: Add a new layer of trainable parameters to the model architecture. Further divided into two types.
  - *Soft Prompt*, where we add additional trainable tokens to prompt and leave it to supervised learning to determine their optimal value.
  - *Adapter*, where we add a trainable layer of parameters inside the encoder or decoder, particularly after attention layers.

Soft Prompt Tuning

1. **No Real Words:** Soft prompts generally *do not* use actual words from the vocabulary.

Instead, they use randomly initialized vectors that are trained to represent task-specific virtual tokens.

These virtual tokens act as a "soft prompt" that guides the model towards the desired behavior.

Think of them as "pseudo-words" that the model learns the meaning of during training.

Soft Prompt Tuning

1. **Concatenation with Input:** The soft prompt tokens are *concatenated* to the beginning of the input sequence.
2. **Training:** *Only the soft prompt embeddings are trained*.

The parameters of the pre-trained language model (PLM) remain frozen. This makes soft prompt tuning very efficient in terms of memory and computation.

"[P1] [P2] [P3] [P4] [P5] I really loved the way the actors act.“
5 learnable soft prompt tokens (virtual tokens)

When we say **only prompt/virtual tokens are updated**, it means:

- During **forward pass**, attention is computed **for all tokens (virtual prompt + input text)**.
- During **backpropagation**, the **gradient update is applied ONLY to the virtual tokens' embeddings**.
- The **input token embeddings remain unchanged** since they are part of the frozen model.
- The vast majority of the model's parameters (the PLM's weights, including the projection matrices) remain frozen, which significantly reduces the computational cost and memory requirements compared to full fine-tuning.

When we say **only prompt/virtual tokens are updated**, it means:

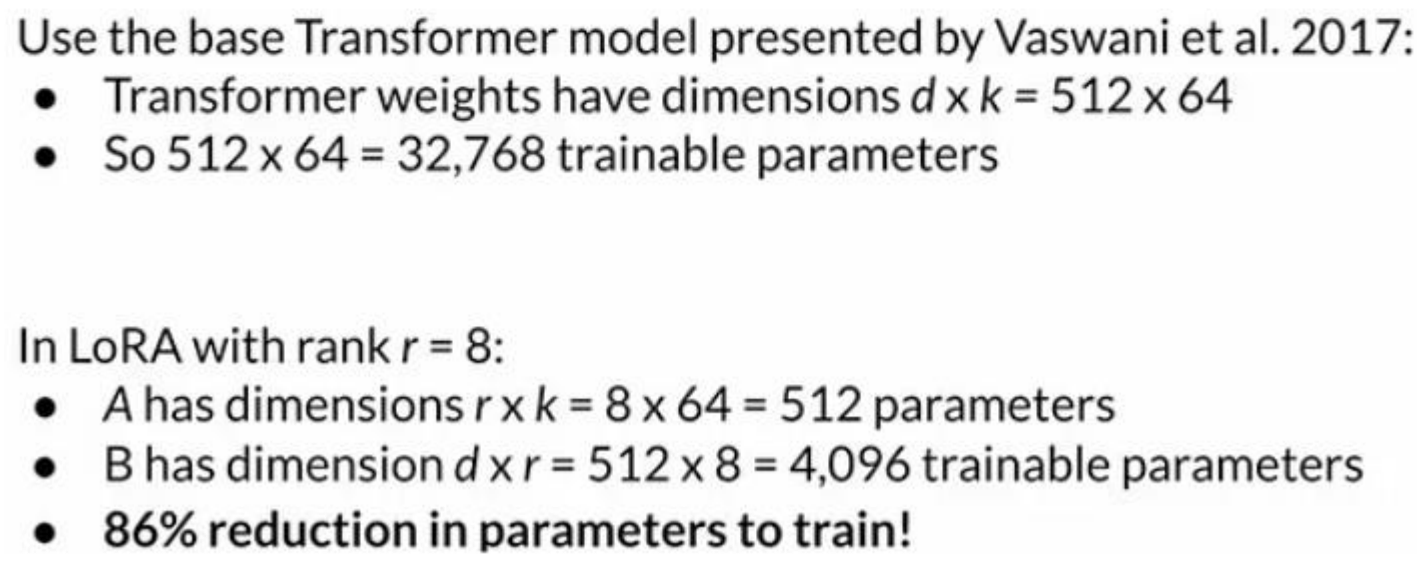
- During **forward pass**, attention is computed **for all tokens (virtual prompt + input text)**.
- During **backpropagation**, the **gradient update is applied ONLY to the virtual tokens' embeddings**.
- The **input token embeddings remain unchanged** since they are part of the frozen model.
- The vast majority of the model's parameters (the PLM's weights, including the projection matrices) remain frozen, which significantly reduces the computational cost and memory requirements compared to full fine-tuning.

Difference b/w Prefix tuning and prompt tuning?

Both have similarities in terms of virtual tokens but in prefix these tokens are not concatenated with the input… read through it and know their differences. (Homework)

LoRA (Low-Rank Adaptation)

- LoRA is a parameter-efficient fine-tuning (PEFT) method that reduces trainable parameters by introducing low-rank matrices A and B instead of fine-tuning all model weights.
- LoRA freezes all of the model parameters during fine-tuning and just trains the injected rank decomposition matrices.



Use the base Transformer model presented by Vaswani et al. 2017:

- Transformer weights have dimensions *d* x *k* = 512 x 64
- So 512 x 64 = 32,768 trainable parameters

In LoRA with rank *r* = 8:

- A has dimensions *r* x *k* = 8 x 64 = 512 parameters
- B has dimension *d* x *r* = 512 x 8 = 4,096 trainable parameters
- **86% reduction in parameters to train!**

Parameter reduction using LoRA, Source: Generative AI with Large Language Models

1 What is $d \times k$ in Transformers?

- d **(512 in the example) → Embedding dimension** (size of each token embedding).
- k **(64 in the example) → Number of output features per attention head.**
  - This is typically d_{head} in multi-head attention.
  - If there are **8 attention heads**, each head has $d_{head} = \frac{d}{heads} = 512/8 = 64$.
  - So, the projection matrix in attention layers is $d \times k = 512 \times 64$.

https://medium.com/@MUmarAmanat/fine-tune-llm-with-peft-60b2798f1e5f

2 What does LoRA do?

Instead of updating the full $W \in \mathbb{R}^{d \times k}$, LoRA **adds low-rank matrices:**

- $W' = W + \Delta W$, where $\Delta W = AB$.
- A has dimensions $r \times k$ (smaller projection).
- B has dimensions $d \times r$.
- r **is the rank** (typically small, e.g., **r=8** in this example).
- Instead of training $d \times k = 32,768$ params, we now train $A + B = 4,608$ **params → 86% fewer parameters!** 🚀

- A is initialized **randomly** with small values
- B is initialized to 0s, and is updated when trained
- Why so?

1. **Original Layer Frozen** ✅
   - We **keep the original weight frozen** and train only LoRA matrices.
2. **LoRA Matrices Defined** ✅
   - A **(8×64)** initialized randomly.
   - B **(512×8)** initialized to zeros.
3. **Compute ΔW as $A \times B$** ✅
   - Results in the same shape as original **512×64** weight update.

At inference time, the effective weight is:

$$W' = W + \Delta W$$

but we never update W, just B and A.

Train only A & B → Compute update $\Delta W = B \times A$

Update final weights $W' = W + \Delta W$

Much fewer parameters to train, same effectiveness!

Full-rank training means training all $512 \times 64 = 32,768$ parameters.

LoRA approximates updates using rank r matrices, reducing trainable parameters to:

$$(r \times k) + (d \times r) = (8 \times 64) + (512 \times 8) = 512 + 4096 = 4608$$

- B is initialized to 0s, and is updated when trained
- Why so?

✅ **Why does LoRA set $B = 0$ and A random?**

1️⃣ **First, B gets updated immediately** because its gradient depends on A^T, which is **random (nonzero).**

2️⃣ A updates in the **next step**, as it depends on B.

🔷 **Training starts smoothly** with meaningful gradients flowing to B from the start.

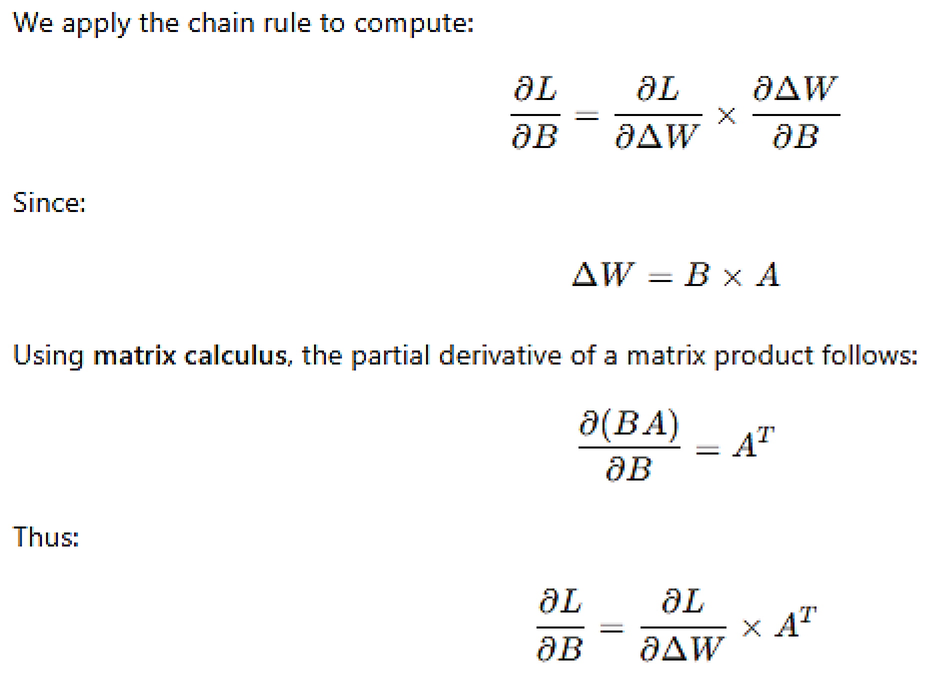
Instead of directly updating W, we compute:

$$\Delta W = B \times A$$

which has the same shape as W, but since **A and B are small**, we have **fewer trainable parameters.**

🔷 **Key Idea**: Rather than learning a full $d \times k$ weight matrix, we **factorize it** into two smaller matrices.

- LoRA initializes B as 0 and A with random normalized weights
- If we do the opposite, will it have the same effect?
- Initialize A with 0 and B with random weights?
- Will we have any problems during backprop?
- Will A and B be updated smoothly?



We apply the chain rule to compute:

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial \Delta W} \times \frac{\partial \Delta W}{\partial B}$$

Since:

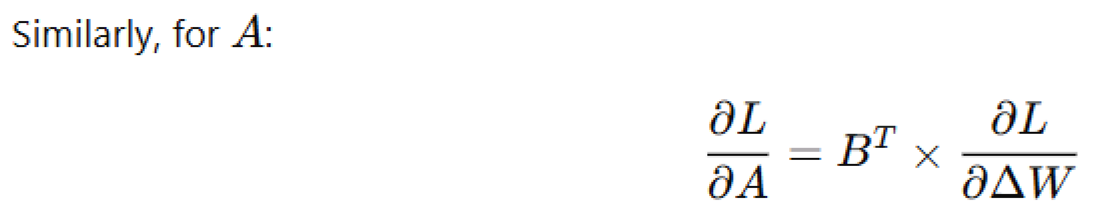
$$\Delta W = B \times A$$

Using **matrix calculus**, the partial derivative of a matrix product follows:

$$\frac{\partial (BA)}{\partial B} = A^T$$

Thus:

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial \Delta W} \times A^T$$



Similarly, for A:

$$\frac{\partial L}{\partial A} = B^T \times \frac{\partial L}{\partial \Delta W}$$

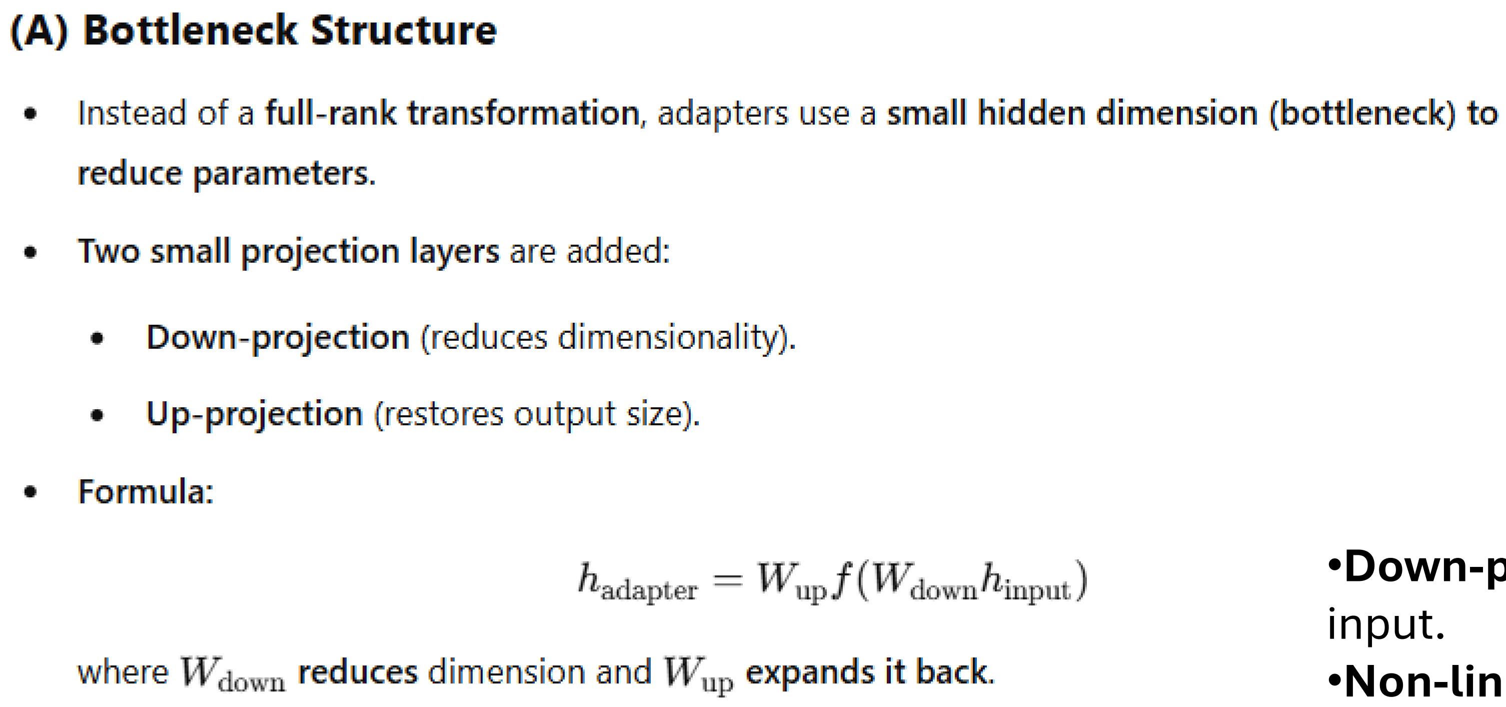
Adapter Layers in PEFT

- Adapter layers are **trainable modules added inside a frozen Transformer model** to enable fine-tuning with **fewer parameters**.
- Instead of modifying the entire model, **adapters learn small, task-specific representations** while keeping the backbone model **unchanged**.
- **Avoids catastrophic forgetting** – Fine-tuning a full model may overwrite useful pretrained knowledge.
- **Efficient fine-tuning** – Adds a small number of trainable parameters while keeping most of the model frozen.
- **Great for multi-task learning** – One model can have **different adapters** for different tasks!

Adapters are **inserted inside each Transformer layer**, usually:

- **After the attention layer** (modifies self-attention outputs).
- **After the feedforward layer** (adds additional transformations).
- [Input Token] → [Self-Attention] → [Feedforward Layer] → [Output]
- **[Input Token] → [Self-Attention] → [Adapter Layer] → [Feedforward Layer] → [Output]**
- Can be **single-layer** (one adapter per model) or **multi-layer** (adapter in every Transformer block).

Components of Adapter Layers



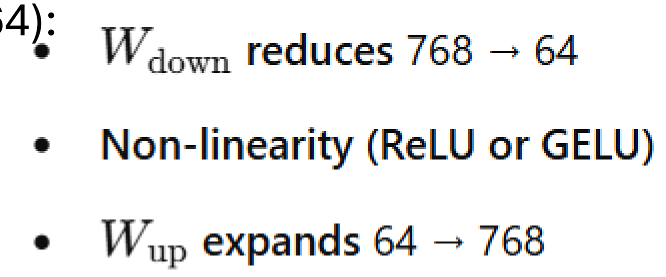
(A) Bottleneck Structure

- Instead of a **full-rank transformation**, adapters use a **small hidden dimension (bottleneck) to reduce parameters.**
- **Two small projection layers** are added:
  - **Down-projection** (reduces dimensionality).
  - **Up-projection** (restores output size).
- **Formula:**

$$h_{\text{adapter}} = W_{\text{up}} f(W_{\text{down}} h_{\text{input}})$$

where W_{down} **reduces** dimension and W_{up} **expands it back.**

Example (if model has 768 dimensions & adapter bottleneck is 64):

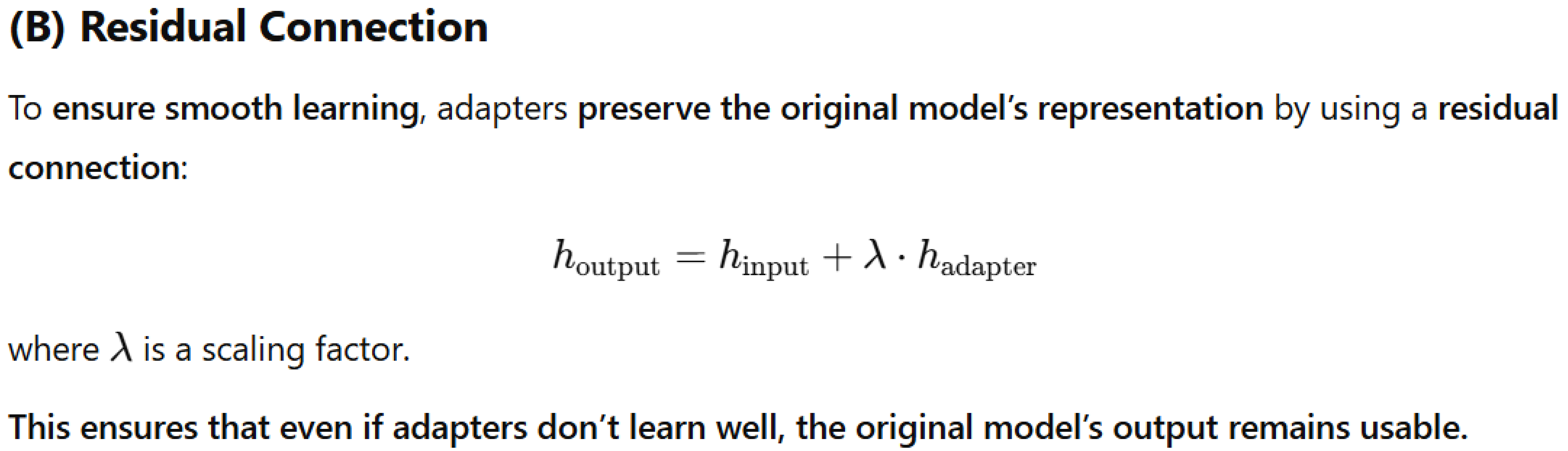


- W_{down} **reduces** 768 → 64
- **Non-linearity (ReLU or GELU)**
- W_{up} **expands** 64 → 768

Only the adapter weights are updated during training—**Transformer layers remain frozen.**

•**Down-projection** → Reduces the dimension of the input.
•**Non-linearity (optional)** → ReLU/GELU can be applied.
•**Up-projection** → Expands the reduced representation back to the original size.
•**Residual Connection** → Adds stability by allowing bypassing of the adapter.

Components of Adapter Layers



(B) Residual Connection

To **ensure smooth learning**, adapters **preserve the original model's representation** by using a **residual connection**:

$$h_{\text{output}} = h_{\text{input}} + \lambda \cdot h_{\text{adapter}}$$

where λ is a scaling factor.

This ensures that even if adapters don't learn well, the original model's output remains usable.

Lambda (λ) in adapter layers is a scaling factor(like 0.1 or 0.01) that controls how much influence the adapter output has compared to the original input.

if adapters don't learn well, the **original model's knowledge remains intact**, preventing catastrophic performance drops.

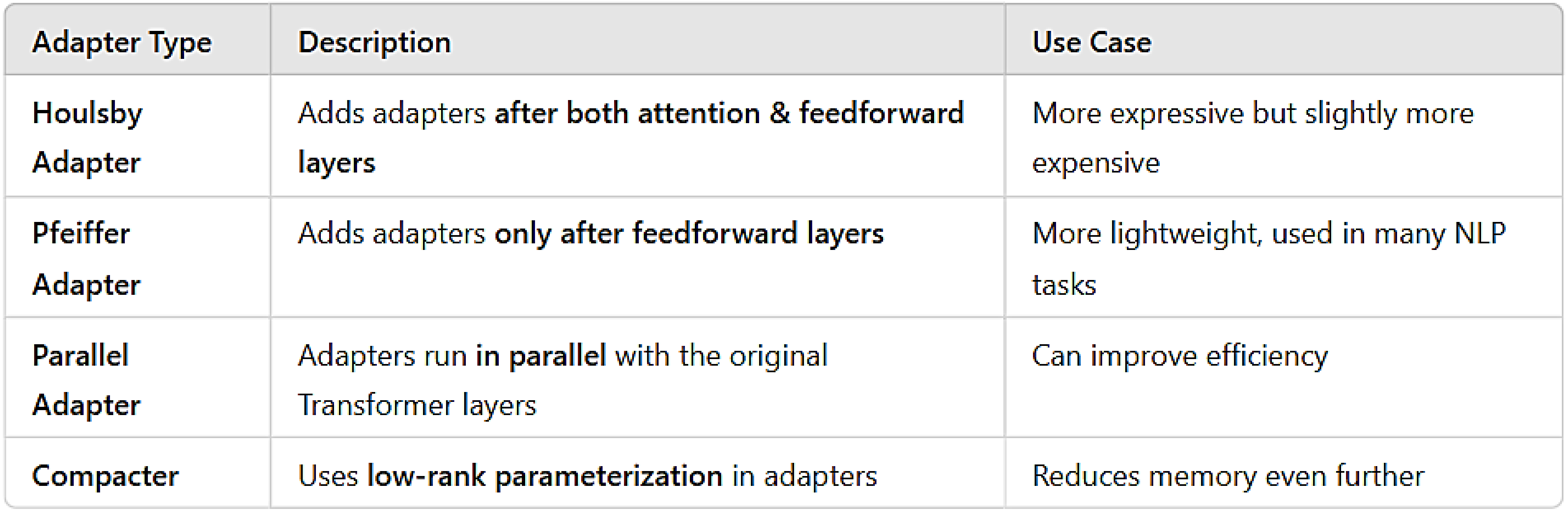
- The adapter **learns new weights based on the downstream task**.
- It **adjusts activations** to minimize the task-specific **loss function**.
- The frozen pre-trained model provides **general knowledge**, while the adapter specializes it for the new task.
- We **reduce dimensions** to **limit trainable parameters** and force compact representations.
- The **adapter learns only the projection matrices** not the full model.
- It is **task-specific** because it fine-tunes these matrices based on the loss function.
- **Residual connections help** in case the adapter doesn’t learn well.

- **Each adapter learns a transformation specific to a task** based on labeled data.
- If trained on **sentiment classification**, it will amplify sentiment-related features.
- If trained on **summarization**, it will learn to condense information.
- We can attach **multiple adapters** for different tasks/domains. Example:
  - **Sentiment Adapter** (Positive/Negative)
  - **Emotion Adapter** (Happy/Sad/Angry)
  - **Summarization Adapter** (Text-to-Text Generation)

The **same frozen Transformer** can switch between tasks just by **activating a different adapter**.

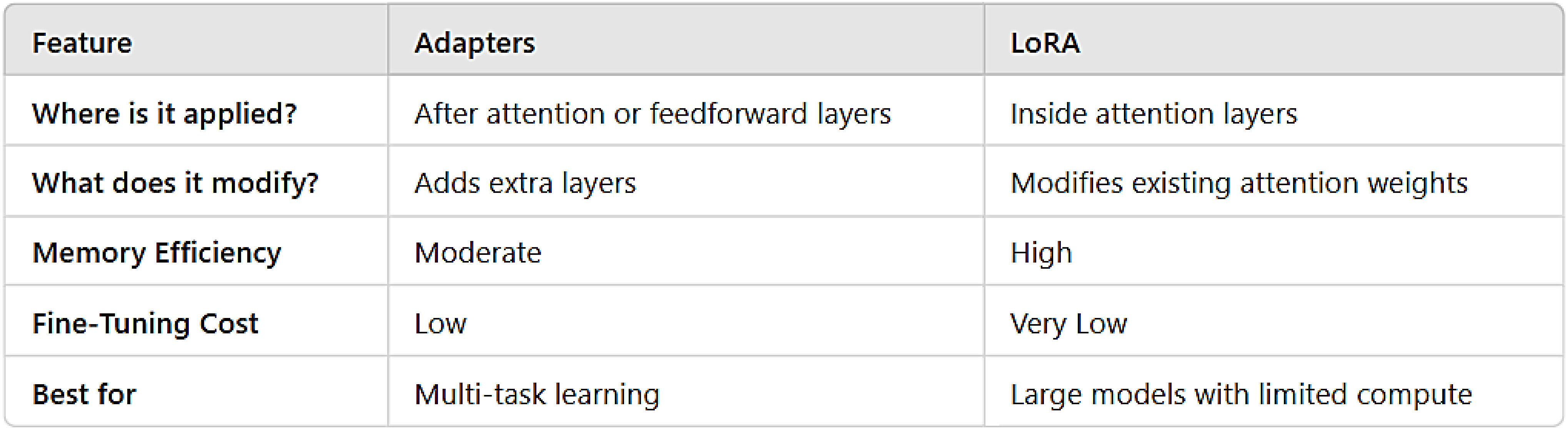
- We **do not** explicitly tell an adapter what task it is for.
- It **learns from labeled data** which features matter for the task.

Adapter Architectures



Adapter Type	Description	Use Case
Houlsby Adapter	Adds adapters **after both attention & feedforward layers**	More expressive but slightly more expensive
Pfeiffer Adapter	Adds adapters **only after feedforward layers**	More lightweight, used in many NLP tasks
Parallel Adapter	Adapters run **in parallel** with the original Transformer layers	Can improve efficiency
Compacter	Uses **low-rank parameterization** in adapters	Reduces memory even further

If you want **existing, pretrained adapters**, you need to **load them from the Adapter Hub**.



Feature	Adapters	LoRA
Where is it applied?	After attention or feedforward layers	Inside attention layers
What does it modify?	Adds extra layers	Modifies existing attention weights
Memory Efficiency	Moderate	High
Fine-Tuning Cost	Low	Very Low
Best for	Multi-task learning	Large models with limited compute

- **LoRA is better for large models (like GPT, LLaMA, Mistral).**
- **Adapters are better for NLP multi-task learning (like BERT, T5).**
    - You can train multiple adapters for different tasks and swap them without re-training the whole model.

- Conceptual:
- https://www.leewayhertz.com/parameter-efficient-fine-tuning/#Use-cases-of-parameter-efficient-fine-tuning
- https://www.datacamp.com/tutorial/understanding-prompt-tuning
- https://toloka.ai/blog/prefix-tuning-vs-fine-tuning/

- Code Examples:
- https://www.datacamp.com/tutorial/fine-tuning-large-language-models
- https://www.datacamp.com/tutorial/mistral-7b-tutorial
- https://www.turing.com/resources/finetuning-large-language-models#what-are-the-different-types-of-llm-fine-tuning
- https://dassum.medium.com/fine-tune-large-language-model-llm-on-a-custom-dataset-with-qlora-fb60abdeba07
- https://medium.com/@MUmarAmanat/fine-tune-llm-with-peft-60b2798f1e5f
- https://towardsdatascience.com/lora-intuitively-and-exhaustively-explained-e944a6bff46b/

Quantization

Quantization is used to make deep learning models smaller and faster by reducing the precision of numbers that represent weights and activations.

- **Memory Efficiency**: High-precision (e.g., FP32) models require a lot of memory. Quantizing to lower-bit representations (e.g., INT8, INT4) significantly reduces memory usage.
- **Faster Computation**: Lower-precision arithmetic is faster on hardware like GPUs, TPUs, and specialized accelerators (e.g., Qualcomm AI chips, NVIDIA TensorRT).
- **Trade-offs**: While quantization saves memory and speeds up inference, it can sometimes reduce model accuracy.

Precision means how accurately we represent numbers in a model.

- **Floating Point (FP32)**: Each weight is stored as a 32-bit floating-point number (high precision).
- **Lower Precision (FP16, INT8, INT4, etc.)**: We represent numbers using fewer bits, which reduces memory but may introduce small errors.

Example:

- A model trained in **FP32** stores a weight as 0.987654321
- If we quantize it to **FP16**, it may become 0.9877 (small loss in precision)
- In **INT8**, it might become 1 (larger loss in precision)

Precision Trade-off

- Higher precision → More memory, better accuracy
- Lower precision → Less memory, slightly lower accuracy
- Quantization tries to **find the best balance** between these two.

Quantization Formula (Example)

- Quantization is done using a **scaling factor** to map FP32 values into a lower-bit range.

$$q = \text{round}\left(\frac{x - x_{\min}}{s}\right)$$

- x is the original FP32 value
- $x_{\min}$ is the minimum value in the range
- s is the **scaling factor**
- q is the quantized integer

$$s = \frac{x_{\max} - x_{\min}}{2^b - 1}$$

where b is the bit width (for INT4, $b = 4$, so $2^4 - 1 = 15$ possible values).

$$x = [-1.0, -0.5, 0.0, 0.5, 1.0]$$

Assume our **quantization range is [-1, 1]**.

- So $x_{\min} = -1$, $x_{\max} = 1$.
- Compute the **scaling factor**:

$$s = \frac{1-(-1)}{15} = \frac{2}{15} = 0.1333$$

Now, apply the formula:

$$q = \text{round}\left(\frac{x-(-1)}{0.1333}\right)$$

So, the **FP32 values are now mapped to INT4 values**:

$$[-1.0, -0.5, 0.0, 0.5, 1.0] \rightarrow [0, 4, 8, 11, 15]$$

- Convert Int4 to float32
- Dequantization:

$$x_{\text{dequantized}} = x_{\min} + q \times s$$

- Solve this given the mapped int4 values.
- The recovered FP32 values will be close to the original ones, but some information will be lost due to rounding.

Types of Quantization

(a) Post-Training Quantization (PTQ)

- Quantize the model **after** training
- Example: Train in FP32, then convert weights to INT8, run inference on this quantized version
- Fast, but might slightly reduce accuracy
- **Works with pre-trained models**: You can quantize existing models without retraining.
  Accuracy loss: The model wasn't trained with quantization in mind, so it might not adapt well.
  Limited flexibility: Works best if the model is already robust to precision loss.

Types of Quantization

(b) Quantization-Aware Training (QAT)

- Model is trained **with quantization in mind**
- The model learns to handle quantization-induced errors
- Better accuracy than PTQ but needs more training time
- **Slower training** because the model sees quantized and dequantized values during learning.
- **More complex** than PTQ since it needs modifications during training.
- **Training**: QAT starts with FP32 and simulates quantization but keep both data formats since FP32 is required when calculating gradients.
- Once the model is trained higher precision weights are discarded and only quantized weights are kept/stored for inference.

Types of Quantization

(c) Static vs. Dynamic Quantization

refers to how **activations** (not weights) are handled at inference

- **Static Quantization**: We precompute quantized weights and activations before inference. Faster but less flexible.
  - **Both weights & activations are quantized BEFORE inference**.
  - We run a small dataset (calibration) through the model **before deployment** to **precompute activation ranges**.
  - During inference, the model **uses these fixed quantized values**.
- **Dynamic Quantization**: Activations are quantized on-the-go during inference. More flexible but can be slower.
- Activations are quantized after a layer, but the next layer dequantizes them before processing, unless your’e working in a fully dequantized model.

- **FP32 (Full Precision)** → Large memory, high accuracy
- **FP16 (Half Precision)** → 2x smaller, nearly same accuracy
- **INT8 (8-bit Integer)** → 4x smaller, minor accuracy loss
- **INT4 (4-bit Integer)** → 8x smaller, more accuracy loss
- **NF4** (Normalized Float 4-bit) → Much better than INT4

NF4 uses **non-uniform mapping** (LUT) to better represent weight distributions in LLMs. Used in **QLoRA**.
Retains better accuracy while keeping memory low

QLoRA

Large models like LLaMA, GPT-4, and Falcon require **hundreds of GBs of GPU memory** for fine-tuning. **QLoRA solves this problem** by:

- Using **4-bit quantization** (to reduce memory needs)
- Adding **LoRA adapters** (to fine-tune only small parts of the model)
- This means **we can fine-tune large models on consumer GPUs (e.g., a 24GB RTX 4090) instead of needing expensive AI clusters.**

(a) 4-bit NormalFloat (NF4) Quantization

- QLoRA uses **NF4**, a special 4-bit floating-point format
- Unlike INT4, **NF4 preserves more important information**
- Helps maintain accuracy while still reducing memory usage

(b) Paged Optimizers

- Normally, when fine-tuning, GPUs store **optimizer states** (e.g., Adam, RMSprop) which take up a lot of memory
- **Paged Optimizers** swap data between GPU and CPU memory efficiently
- This prevents **out-of-memory (OOM) errors** during fine-tuning

(c) LoRA on Quantized Models

- Instead of fine-tuning **all** model weights, we only train **LoRA adapters**
- LoRA **adds small trainable layers** on top of frozen layers
- **This saves a ton of memory!**

QLoRA Workflow

Step 1: Load a Pretrained Model & Quantize It

- We start with a **pretrained model (FP16 or FP32)**.
- Convert its weights to **4-bit NF4 quantization** using **bitsandbytes**.
- The model is now stored **efficiently in 4-bit memory** but still functional.

Step 2: Apply LoRA Adapters

- Use **Hugging Face PEFT** to add **LoRA layers** to attention matrices.
- Only train the **small LoRA matrices (A, B)** while keeping the quantized model frozen.

Step 3: Fine-Tune on a Dataset

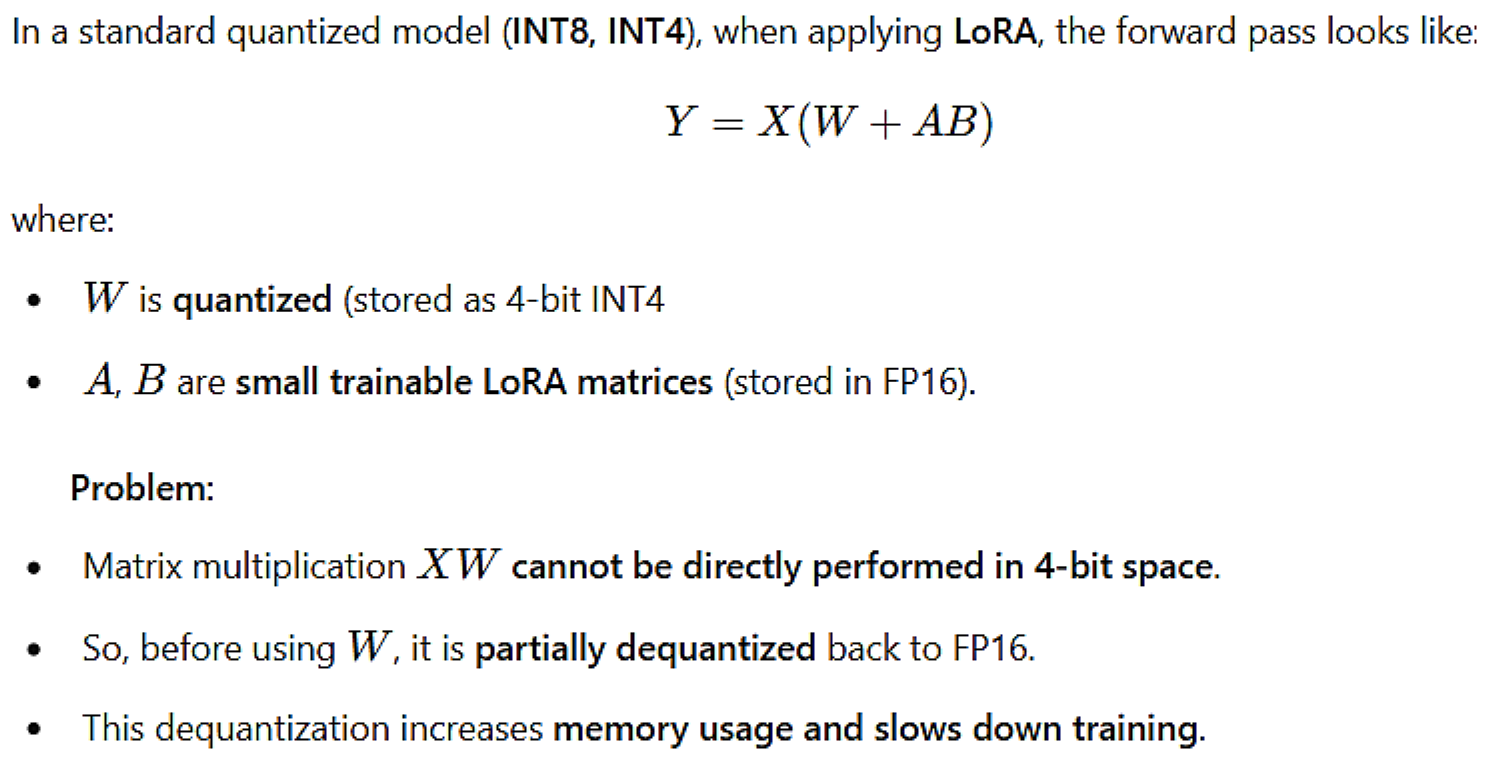
- Train the model **on a dataset** (Alpaca, OpenAssistant, etc.).
- Since the model is **quantized (4-bit NF4)**, we save **huge memory** while training.
- We **only update LoRA adapters**, keeping the rest unchanged.

Step 4: Run Inference on the Quantized Model

- **The final model is still quantized (NF4).**
- The trained **LoRA adapters are merged** at inference time.
- We now get a **fine-tuned, memory-efficient model** that works in 4-bit.

- Pre trained models are stored in FP16/FP32. **QLoRA compresses them** to a **4-bit format.**
- Is QLoRA is **just LoRA on a quantized model?**.
- The whole process (fine-tuning + inference) happens in **QLoRA**.
- LoRA does need partial dequantization. **QLoRA avoids this using NF4 + double quantization**

Where Does LoRA Dequantize and Why QLoRA Avoids It?



In a standard quantized model (**INT8, INT4**), when applying **LoRA**, the forward pass looks like:

$$Y = X(W + AB)$$

where:

- W is **quantized** (stored as 4-bit INT4
- A, B are **small trainable LoRA matrices** (stored in FP16).

Problem:

- Matrix multiplication XW **cannot be directly performed in 4-bit space.**
- So, before using W, it is **partially dequantized** back to FP16.
- This dequantization increases **memory usage and slows down training.**

- **QLoRA**
- Keeps the model **fully quantized in 4-bit NF4** during fine-tuning.
- **No need to dequantize weights** for every forward/backward pass, NF4 supports efficient mixed-precision computation with FP16.
- **Only the LoRA matrices (A, B) are in FP16**, which are small and don’t use much memory.
- The trainable parameters are less than 1% of the total model size, so storing A,B in FP16 doesn’t significantly increase memory usage.
- **Why are A B kept in Fp16 and not in NF4?**

Where Does LoRA Dequantize and Why QLoRA Avoids It?

•If we **quantize A,B to NF4**, it would: **Reduce the precision of updates** (important for learning small, detailed patterns).

•**Impact fine-tuning performance** because quantization reduces the expressiveness of updates.

•NF4 is **good for storing pre-trained weights**, but it's **not ideal for updating weights dynamically** during training.

•Training requires **gradient updates**, and quantization (NF4) introduces **rounding errors**.

•If we stored A,B in NF4:

- The **gradient updates would be too imprecise**.
- The model would **struggle to learn fine-tuned adjustments**.

Keeping A,B in FP16 ensures stable and accurate gradient updates.

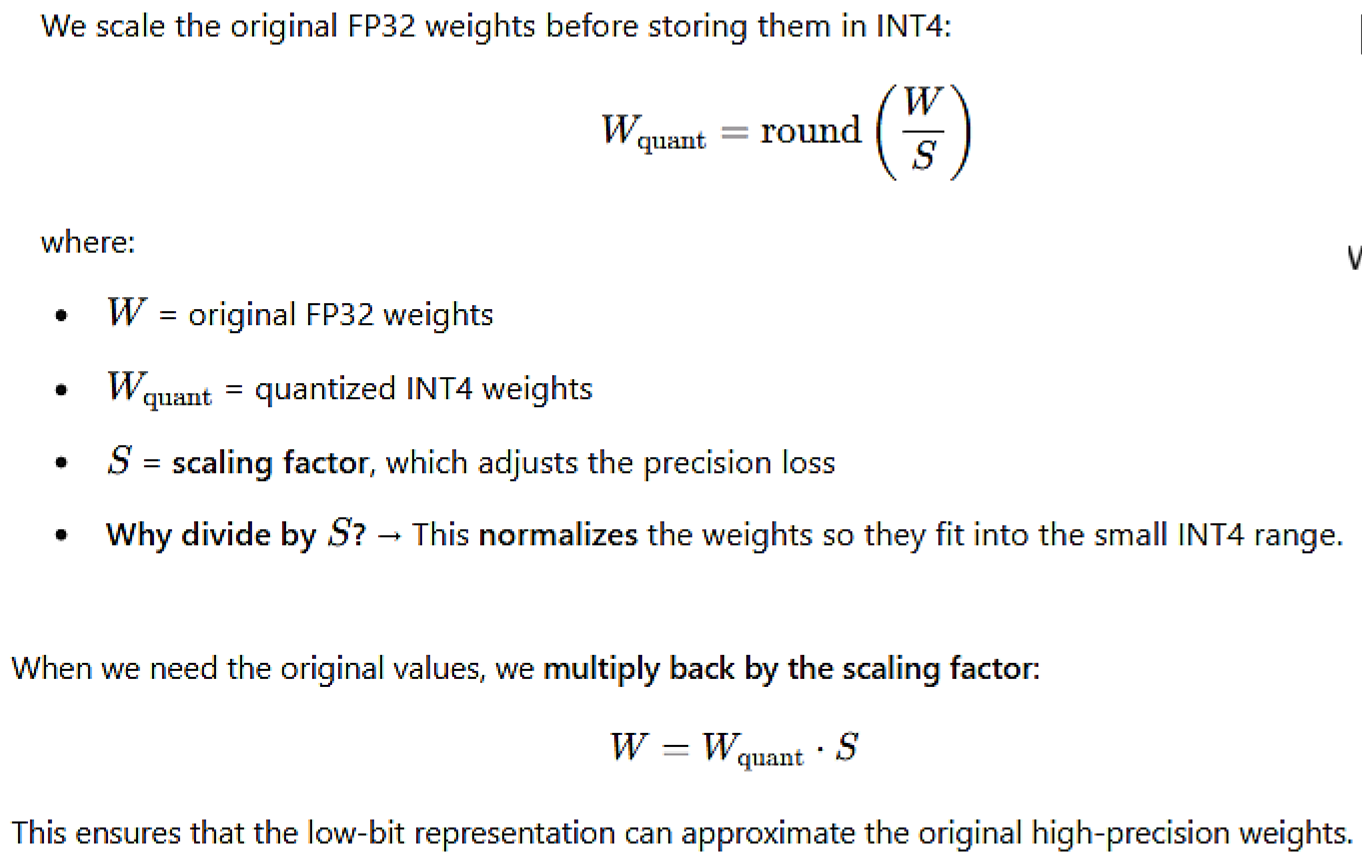
- **QLoRA**
- Keeps the model **fully quantized in 4-bit NF4** during fine-tuning.
- **No need to dequantize weights** for every forward/backward pass.
- **Only the LoRA matrices (A, B) are in FP16**, which are small and don't use much memory.
- The trainable parameters are less than 1% of the total model size, so storing A, B in FP16 doesn't significantly increase memory usage.
- **Why are A B kept in Fp16 and not in NF4?**

```
# Load the model with 4-bit quantization
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    load_in_4bit=True,  # Enables QLoRA quantization
    device_map="auto",  # Auto GPU placement
    quantization_config=bnb.BitsAndBytesConfig(
        load_in_4bit=True,  # Use 4-bit quantization
        bnb_4bit_compute_dtype=torch.float16,  # Use FP16 for computations
        bnb_4bit_quant_type="nf4",  # Use NF4 quantization (better than INT4)
        bnb_4bit_use_double_quant=True  # Enable Double Quantization
    )
)
```

- `load_in_4bit=True` → Loads the model in **4-bit NF4 quantization.**
- `bnb_4bit_quant_type="nf4"` → Uses NF4 (instead of INT4) for better precision.
- `bnb_4bit_use_double_quant=True` → Activates Double Quantization (saves even more memory).
- **No dequantization needed during fine-tuning!**

LoRA adapters are still in FP16, but NF4 multiplication is supported

QLoRA's Double Quantization



We scale the original FP32 weights before storing them in INT4:

$$W_{\text{quant}} = \text{round}\left(\frac{W}{S}\right)$$

where:

- W = original FP32 weights
- W_{quant} = quantized INT4 weights
- S = **scaling factor**, which adjusts the precision loss
- **Why divide by S?** → This **normalizes** the weights so they fit into the small INT4 range.

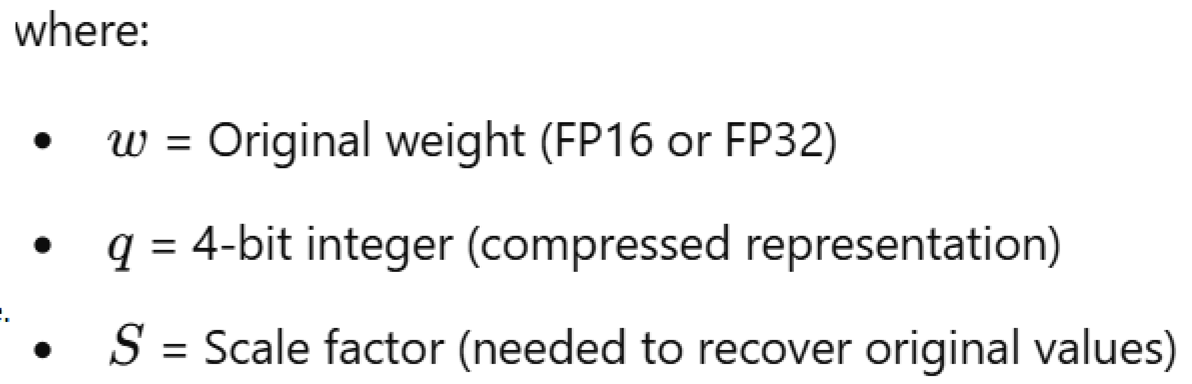
When we need the original values, we **multiply back by the scaling factor**:

$$W = W_{\text{quant}} \cdot S$$

This ensures that the low-bit representation can approximate the original high-precision weights.

In regular 4-bit quantization (NF4 or INT4), each weight **w** is approximated as:

$$w \approx S \cdot q$$



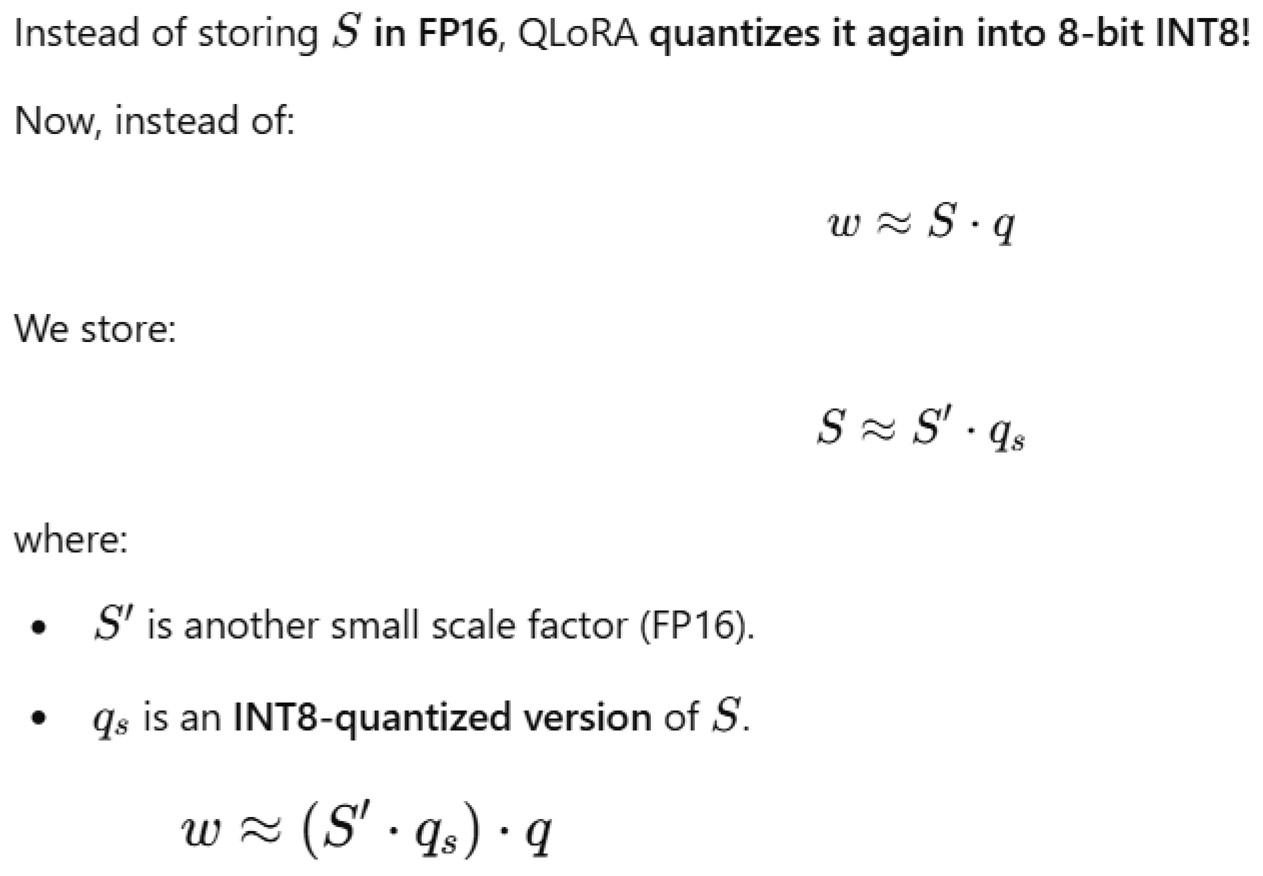
where:

- w = Original weight (FP16 or FP32)
- q = 4-bit integer (compressed representation)
- S = Scale factor (needed to recover original values)

S Prevents large accuracy loss when mapping FP32 to INT4.

Problem: The **scale factor S is still stored in FP16 or FP32**, which **takes up memory**.

QLoRA’s Double Quantization



Instead of storing S **in FP16**, QLoRA **quantizes it again into 8-bit INT8!**

Now, instead of:

$$w \approx S \cdot q$$

We store:

$$S \approx S' \cdot q_s$$

where:

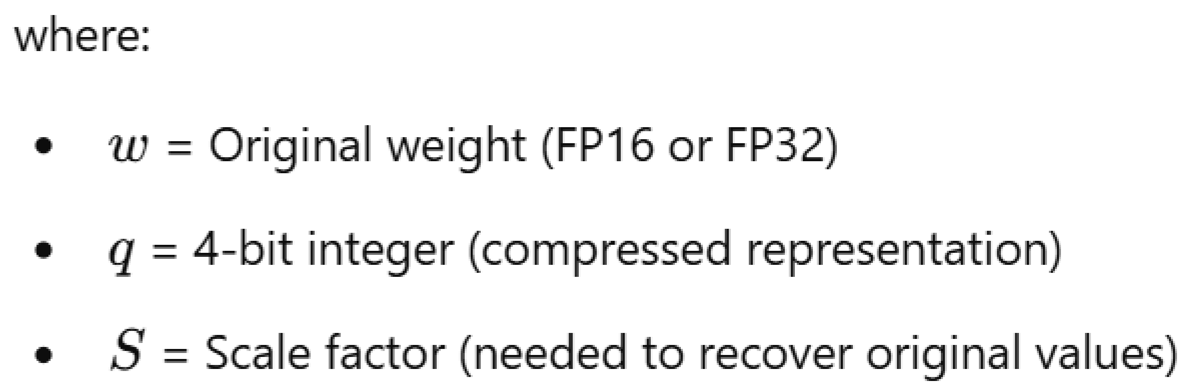
- S' is another small scale factor (FP16).
- q_s is an **INT8-quantized version of** S.

$$w \approx (S' \cdot q_s) \cdot q$$

Instead of storing **S in FP16 for every weight block**, we store **only an INT8 version**.
This **saves extra memory** and helps **fit larger models on GPUs** without significant accuracy loss.

In regular 4-bit quantization (NF4 or INT4), each weight **w** is approximated as:

$$w \approx S \cdot q$$



where:

- w = Original weight (FP16 or FP32)
- q = 4-bit integer (compressed representation)
- S = Scale factor (needed to recover original values)

S Prevents large accuracy loss when mapping FP32 to INT4.

Problem: The **scale factor S is still stored in FP16 or FP32**, which **takes up memory**.

QLoRA makes
Model Weights (from FP16 → NF4)
Scale Factors (from FP16 → INT8)

- https://medium.com/@dillipprasad60/qlora-explained-a-deep-dive-into-parametric-efficient-fine-tuning-in-large-language-models-llms-c1a4794b1766
- https://blog.dataiku.com/quantization-in-llms-why-does-it-matter#:~:text=Double%20Quantization&text=This%20involves%20performing%20a%20second,a%20second%20round%20quantization%20factor.