

Parameter Efficient Fine Tuning (LLM)

A Resource-Conscious Approach to LLM Adaptation

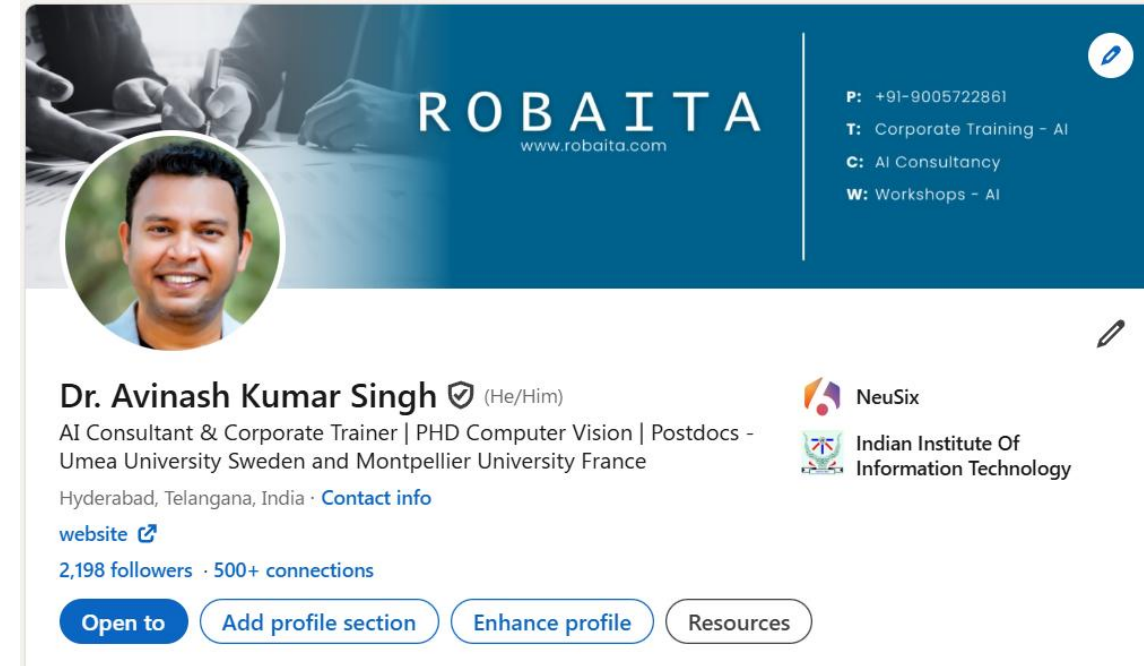
Dr. Avinash Kumar Singh

AI Consultant and Coach, Robaita



Dr. Avinash Kumar Singh

- ❑ **Possess** 15+ years of **hands-on expertise** in Machine Learning, Computer Vision, NLP, IoT, Robotics, and Generative AI.
- ❑ **Founded** Robaita—an initiative **empowering** individuals and organizations to **build, educate, and implement** AI solutions.
- ❑ **Earned** a Ph.D. in Human-Robot Interaction from IIIT Allahabad in 2016.
- ❑ **Received** postdoctoral fellowships at Umeå University, Sweden (2020) and Montpellier University, France (2021).
- ❑ **Authored** 30+ research papers in **high-impact** SCI journals and international conferences.
- ❑ Unlearning, learning, making mistakes ...



<https://www.linkedin.com/in/dr-avinash-kumar-singh-2a570a31/>



HCLTech



B R A N E



Discussion Points

- Parameter-Efficient Fine-Tuning (PEFT)
 - LORA
 - QLORA
 - Adapters & Prefix-Tuning
- Evaluation and Red-Teaming

Parameter-Efficient Fine-Tuning (PEFT)

Parameter-Efficient Fine-Tuning (PEFT) refers to a **family of techniques** designed to **fine-tune large language models (LLMs)** by updating only a small subset of model parameters.

Unlike full fine-tuning, which retrain all weights and requires significant compute and storage, PEFT methods **target specific layers or components—often adapters or low-rank matrices**—making them **ideal for edge deployments and resource-constrained environments**.

LoRA (Low-Rank
Adaptation)

QLoRA
(Quantized LoRA)

Adapters &
Prefix-Tuning

Parameter-Efficient Fine-Tuning (PEFT)

LoRA (Low-Rank Adaptation)

- Injects low-rank matrices into attention layers to approximate weight updates.
- Trains only these low-rank matrices while freezing the rest of the model.
- Example: For a 7B model, LoRA can reduce trainable parameters from billions to just a few million.

QLoRA (Quantized LoRA)

- Builds upon LoRA by applying 4-bit quantization to the base model, drastically reducing memory footprint.
- Supports full fine-tuning capabilities on consumer-grade GPUs (e.g., NVIDIA T4, RTX 3090).
- Example: Vicuna 13B was fine-tuned on a single A100 using QLoRA with high performance on MT-Bench.

Adapters & Prefix-Tuning

- Insert small modules (adapters) between transformer layers or prepend task-specific vectors (prefix-tuning).
- Efficient for multitask LLM scenarios and continual learning setups.

- **Vicuna** is an open-source **chatbot-style large language model (LLM)** developed by researchers from LMSYS (Large Model Systems Organization), based on **LLaMA** (Meta's foundational model)
- **MT-Bench** (Multi-Turn Bench) is a benchmarking framework designed to evaluate chatbots through multi-turn conversations. It was also developed by LMSYS to assess models like Vicuna, ChatGPT, etc.

Low-Rank Adaptation (LoRA)

Published in 2021

LoRA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS

Edward Hu* Yelong Shen* Phillip Wallis Zeyuan Allen-Zhu
Yuanzhi Li Shean Wang Lu Wang Weizhu Chen
Microsoft Corporation
{edwardhu, yeshe, phwallis, zeyuana,
yuanzhil, swang, luw, wzchen}@microsoft.com
yuanzhil@andrew.cmu.edu

ABSTRACT

An important paradigm of natural language processing consists of large-scale pre-training on general domain data and adaptation to particular tasks or domains. As we pre-train larger models, full fine-tuning, which retrains all model parameters, becomes less feasible. Using GPT-3 175B as an example – deploying independent instances of fine-tuned models, each with 175B parameters, is prohibitively expensive. We propose Low-Rank Adaptation, or LoRA, which freezes the pre-trained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture, greatly reducing the number of trainable parameters for downstream tasks. Compared to GPT-3 175B fine-tuned with Adam, LoRA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times. LoRA performs on-par or better than fine-tuning in model quality on RoBERTa, DeBERTa, GPT-2, and GPT-3, despite having fewer trainable parameters, a higher training throughput, and, unlike adapters, *no additional inference latency*. We also provide an empirical investigation into rank-deficiency in language model adaptation, which sheds light on the efficacy of LoRA. We release a package that facilitates the integration of LoRA with PyTorch models and provide our implementations and model checkpoints for RoBERTa, DeBERTa, and GPT-2 at <https://github.com/microsoft/LoRA>.

Rank of a Matrix

The rank of a matrix represents the number of linearly independent rows (or columns).

- If all rows (or columns) are linearly independent, the matrix is full-rank (rank = 6 for 6×6 matrix).
- If some rows (or columns) can be expressed as linear combinations of others, the matrix is low-rank (rank < 6).

How to calculate rank of the matrix

Let's assume we have a matrix A, for this we need to calculate the rank

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 1 & 1 & 1 \end{bmatrix}$$

There is only one rule, try to make maximum rows or column as zero.

Update the matrix as

$$\text{Row 2} = \text{Row 2} - 2 \times \text{Row 1} \quad [2 \ 4 \ 6] - 2 \times [1 \ 2 \ 3] = [0 \ 0 \ 0]$$

$$\text{Row 3} = \text{Row 3} - 1 \times \text{Row 1}: \quad [1 \ 1 \ 1] - [1 \ 2 \ 3] = [0 \ -1 \ -2]$$

$$A \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & -1 & -2 \end{bmatrix}$$

Rank of a Matrix

Update the Matrix as

Row 3 = Row 3 \div $(-1) = [0, 1, 2]$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

Update the Matrix as

Row 1 = Row 1 $- 2 \times$ Row 3

$$[1 \ 2 \ 3] - 2 \times [0 \ 1 \ 2] = [1 \ 0 \ -1]$$

$$A = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

Finally, count the number of non zero rows or columns whichever is minimum.

$$\text{Rank}(A) = \text{Number of non-zero rows} = 2$$

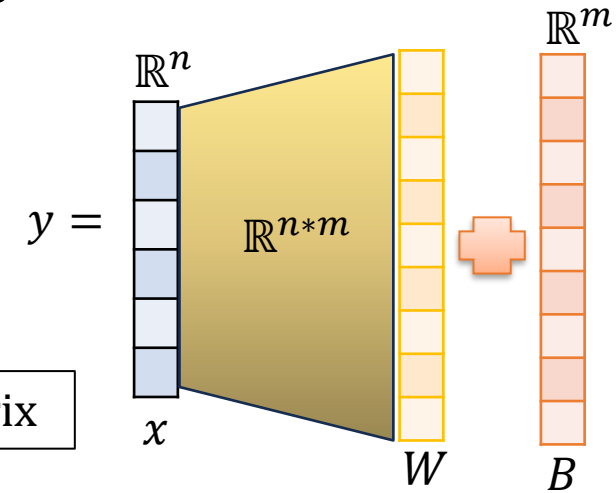
Rank and Weight Matrix Relation

Let's assume that one of the transformer layer, we have weight matrix "W" and bias "B" expressed as:

Where

$x \in \mathbb{R}^n$: input vector
 $W \in \mathbb{R}^{n \times m}$: weight matrix
 $B \in \mathbb{R}^m$: Bias vector
 $y \in \mathbb{R}^m$: Output vector

$$y = W^T x + B$$



RNK: Rank of the weight matrix

if $RNK = \min(m, n)$

if $RNK \neq \min(m, n)$

All the dimension are equally important

Then it is a full rank matrix

Then let's say the $RNK = k$, where $k < \min(m, n)$

The output of this layer **lives in a k-dimensional subspace** of \mathbb{R}^m

Rank and Weight Matrix Relation

The output of the layer **lives in a k-dimensional subspace** of \mathbb{R}^m means:

- Although the layer has m output neurons,
- The actual variation in outputs only spans a k -dimensional subspace.
- So, only k of those outputs are contributing independently to learning and prediction.
- The remaining $m-k$ neurons are linear combinations of the others — they add no new information.

Rank and Weight Matrix Relation

Let's say:

- Layer has 100 output neurons (i.e., $m=100$).
- Rank of the weight matrix is 10 (i.e., $k=10$)

Then

- Only 10 neurons' outputs are truly **linearly independent**.
- The remaining 90 neurons are **linear combinations** of these 10.
- In learning:
 - **Gradients** flowing back through these redundant neurons carry **no additional information**.
 - **Backprop updates** to their weights will not help the model **learn new patterns**.
- In prediction:
 - These neurons do **not influence the decision independently**.
 - They do not add any new **degrees of freedom** to the model's output.

So, even though we are computing 100 outputs, the **model is behaving as if it only has 10 meaningful outputs**.

Low-Rank Adaptation (LORA)

The High-Dimensional Nature of W

$$W^T \in \mathbb{R}^{m \times n} = \mathbb{R}^{512 \times 1024}$$

- This gives over half a million parameters just in one dense layer.
- In a large network, there are millions or billions of such parameters across all layers.

We take inspiration from [Li et al. \(2018a\)](#); [Aghajanyan et al. \(2020\)](#) which show that the learned over-parametrized models in fact reside on a low intrinsic dimension. We hypothesize that the change in weights during model adaptation also has a low “intrinsic rank”, leading to our proposed Low-Rank Adaptation (LoRA) approach. LoRA allows us to train some dense layers in a neural network indirectly by optimizing rank decomposition matrices of the dense layers’ change during adaptation instead, while keeping the pre-trained weights frozen, as shown in [Figure 1](#). Using GPT-3 175B as an example, we show that a very low rank (i.e., r in [Figure 1](#) can be one or two) suffices even when the full rank (i.e., d) is as high as 12,288, making LoRA both storage- and compute-efficient.

Performance Lies in a Low-Dimensional Subspace

Even though W lies in a high-dimensional space $\mathbb{R}^{m \times n}$, the optimization process (e.g., SGD) doesn't explore all directions equally. Instead:

- Only a small number of parameter directions actually impact performance.
- The rest have near-zero gradients or lead to minimal improvement.
- This means the effective dimensionality of the solution space is much lower

Let's assume the final optimized W^* after training can be **approximated** by:

$$W^* = W_0 + \Delta W$$

Where ΔW is the update applied during fine-tuning.

Low-Rank Adaptation (LORA)

Rank Decomposition

If ΔW lies in a **low-dimensional linear subspace**.

Then there must exist $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$,
where $k \ll \min(m, n)$
$$\Delta W \approx AB$$

Let's visit the same example

$$\begin{aligned} W^T &\in \mathbb{R}^{m \times n} = \mathbb{R}^{512 \times 1024} \approx 5,242,880 \\ W^T &\approx AB, \text{ where } A \in \mathbb{R}^{512 \times 50}, B \in \mathbb{R}^{50 \times 1024} \\ A &= 25,600 \text{ and } B = 51,200 \\ \text{Total parameters} &= 76,800 \end{aligned}$$

We take inspiration from [Li et al. \(2018a\)](#); [Aghajanyan et al. \(2020\)](#) which show that the learned over-parametrized models in fact reside on a low intrinsic dimension. We hypothesize that the change in weights during model adaptation also has a low “intrinsic rank”, leading to our proposed Low-Rank Adaptation (LoRA) approach. LoRA allows us to train some dense layers in a neural network indirectly by optimizing rank decomposition matrices of the dense layers' change during adaptation instead, while keeping the pre-trained weights frozen, as shown in [Figure 1](#). Using GPT-3 175B as an example, we show that a very low rank (i.e., r in [Figure 1](#)) can be one or two) suffices even when the full rank (i.e., d) is as high as 12,288, making LoRA both storage- and compute-efficient.

$$W^* = W_0 + \Delta W$$

Because if ΔW is low-rank, we don't need to **train all of W again**.

We keep W_0 frozen.

- This drastically reduces memory & compute.
- But still captures the essential directions for task-specific adaptation

LORA Implementation

Scaled LORA

$y = W^T x + B$ (the base Model)

$y = (W + \Delta W)x + B$ (LORA)

Where $\Delta W = \frac{\alpha}{r} AB$ (scaled LORA)

Let's train the LORA on
Language to python Task



Hugging Face





Search models, datasets, users...



Datasets: flytech/python-codes-25k



like 123

output string · lengths	instruction string · lengths	input string · lengths	text string · lengths
 1→208 41.8%	 4→175 87.4%	 17→34 2.8%	 57→292 28.1%
<pre>```python import shutil folder_name = input('Enter the folder name to zip: ') shutil.make_archive(folder_name, 'zip', folder_name) ```</pre>	Create a ZIP archive of a folder	Creating ZIP archive...	Create a ZIP archive of a folder Creating ZIP archive... <pre>```python import shutil folder_name = input('Enter the folder name to zip: ') shutil.make_archive(folder_name, 'zip', folder_name) ```</pre>

LLAMA 3.2 1B Architecture

```
LlamaForCausalLM(  
  (model): LlamaModel(  
    (embed_tokens): Embedding(128256, 2048)  
    (layers): ModuleList(  
      (0-15): 16 x LlamaDecoderLayer(  
        (self_attn): LlamaAttention(  
          (q_proj): Linear(in_features=2048, out_features=2048, bias=False)  
          (k_proj): Linear(in_features=2048, out_features=512, bias=False)  
          (v_proj): Linear(in_features=2048, out_features=512, bias=False)  
          (o_proj): Linear(in_features=2048, out_features=2048, bias=False)  
        )  
        (mlp): LlamaMLP(  
          (gate_proj): Linear(in_features=2048, out_features=8192, bias=False)  
          (up_proj): Linear(in_features=2048, out_features=8192, bias=False)  
          (down_proj): Linear(in_features=8192, out_features=2048, bias=False)  
          (act_fn): SiLU()  
        )  
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)  
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)  
      )  
      (norm): LlamaRMSNorm((2048,), eps=1e-05)  
      (rotary_emb): LlamaRotaryEmbedding()  
    )  
    (lm_head): Linear(in_features=2048, out_features=128256, bias=False)  
  )  
)
```

LORA Implementation

1. Load Dataset

```
from datasets import load_dataset
dataset = load_dataset(
    "json",
    data_files="https://huggingface.co/datasets/flytech/python-codes-25k/resolve/main/python-codes-25k.json"
)
print(dataset)
train_dataset, eval_dataset = tokenized_datasets.train_test_split(test_size=0.1).values()
```

2. Tokenization

```
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-3.2-1B")
tokenizer.pad_token = tokenizer.eos_token
def tokenize_function(examples):
    tokenized_inputs = tokenizer(examples["text"], truncation=True, padding="max_length",
max_length=512)
    tokenized_inputs["labels"] = tokenized_inputs["input_ids"].copy()
    return tokenized_inputs
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

3. Load Model

```
model =
AutoModelForCausalLM.from_pretrained("meta-llama/Llama-3.2-1B", load_in_8bit=True,
device_map="auto")
```

4. LORA Configuration

```
lora_config = LoraConfig(
    r=8, # rank
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)
model = get_peft_model(model, lora_config)
```


LORA Implementation

5. Training Configuration

```
training_args = TrainingArguments(  
    output_dir="./lora-llama2-python",  
    per_device_train_batch_size=4,  
    per_device_eval_batch_size=4,  
    num_train_epochs=1,  
    learning_rate=2e-4,  
    logging_dir="./logs",  
    logging_steps=10,  
    save_strategy="epoch",  
    fp16=True,  
    push_to_hub=False  
)
```

6. Training Configuration

```
from transformers import Trainer  
trainer = Trainer(  
    model=model,  
    args=training_args,  
    train_dataset=train_dataset,  
    eval_dataset=eval_dataset,  
)  
trainer.train()
```

7. Save model and tokens

```
model.save_pretrained("./lora-llama3.2-python-chkpt")  
tokenizer.save_pretrained("./lora-llama3.2-python-token")
```

QLoRA

QLoRA (Quantized LoRA) is a parameter-efficient fine-tuning technique that enables training large language models (LLMs) using low-precision (quantized) weights, specifically 4-bit weights, without losing performance.

- It builds on top of LoRA (Low-Rank Adaptation) by combining it with quantization, which reduces memory usage and computation.
- With QLoRA, you can fine-tune 13B+ parameter models on a single GPU (e.g., 24GB).

Base Paper: QLoRA: Efficient Finetuning of Quantized LLMs

Published in 2023, NeurIPS 2023

<https://arxiv.org/abs/2305.14314>

QLoRA: Efficient Finetuning of Quantized LLMs

Tim Dettmers*

Artidoro Pagnoni*

Ari Holtzman

Luke Zettlemoyer

University of Washington

{dettmers,artidoro,ahal,lsz}@cs.washington.edu

Abstract

We present QLoRA, an efficient finetuning approach that reduces memory usage enough to finetune a 65B parameter model on a single 48GB GPU while preserving full 16-bit finetuning task performance. QLoRA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LoRA). Our best model family, which we name **Guanaco**, outperforms all previous openly released models on the Vicuna benchmark, reaching 99.3% of the performance level of ChatGPT while only requiring 24 hours of finetuning on a single GPU. QLoRA introduces a number of innovations to save memory without sacrificing performance: (a) 4-bit NormalFloat (NF4), a new data type that is information theoretically optimal for normally distributed weights (b) Double Quantization to reduce the average memory footprint by quantizing the quantization constants, and (c) Paged Optimizers to manage memory spikes. We use QLoRA to finetune more than 1,000 models, providing a detailed analysis of instruction following and chatbot performance across 8 instruction datasets, multiple model types (LLaMA, T5), and model scales that would be infeasible to run with regular finetuning (e.g. 33B and 65B parameter models). Our results show that QLoRA finetuning on a small high-quality dataset leads to state-of-the-art results, even when using smaller models than the previous SoTA. We provide a detailed analysis of chatbot performance based on both human and GPT-4 evaluations showing that GPT-4 evaluations are a cheap and reasonable alternative to human evaluation. Furthermore, we find that current chatbot benchmarks are not trustworthy to accurately evaluate the performance levels of chatbots. A lemon-picked analysis demonstrates where **Guanaco** fails compared to ChatGPT. We release all of our models and code, including CUDA kernels for 4-bit training.²

How is QLoRA different from LoRA?

Feature	LoRA	QLoRA
Weight precision	16-bit (FP16/BF16)	4-bit quantized (NF4)
Memory efficiency	Good	Excellent
LoRA adapter usage	Yes	Yes
Base model updated?	No	No
Main innovation	Injects trainable low-rank matrices	Injects low-rank matrices into quantized base model
Target use	Fine-tuning large models efficiently	Fine-tuning very large models on consumer hardware

What gap does QLoRA address?

The challenge:

- Fine-tuning large LLMs (like LLaMA-13B or Falcon-40B) is extremely memory-intensive.
- LoRA reduced compute but still required 16-bit weights, which doesn't scale well for very large models on consumer GPUs.

QLoRA addresses:

- * Memory bottlenecks: Enables fine-tuning of 65B models on a single 48GB GPU.
- * Cost efficiency: You don't need TPUs or multiple A100s.
- * Model democratization: Makes it easier for smaller labs or developers to work with large LLMs.

Why is QLoRA important?

- ✓ 4x memory savings using 4-bit quantization (compared to FP16)
- ✓ No performance loss when using NF4 quantization
- ✓ Works well for instruction tuning, chatbots, RAG systems, etc.
- ✓ Compatible with HuggingFace's PEFT, Transformers, and BitsAndBytes libraries

How it Works

Step 1: Quantize the base model to 4-bit

Using NF4 (NormalFloat4) quantization:

- Each weight in the model is stored using 4 bits, reducing memory by 4x over FP16.
- Example: A weight $w = 0.47$ becomes something like an index 0110 in a 16-entry quantization codebook.

What is NormalFloat4 (NF4)?

Unlike standard quantization (e.g., INT4, FP4), NF4 is tailored to the distribution of neural network weights, which are usually normally distributed around 0.

Key Characteristics of NF4:

- Represents values using **4 bits**, giving us **16 unique values**.
- These values are **not uniformly spaced**; they're **densely packed around 0**, mimicking a **normal distribution**.
- Provides higher resolution for small values, which are **more common in pretrained model weights**.

How it Works

Let's convert FP32 to NH4

weights_fp32 = [0.05, -0.3, 0.95, 0.0, -0.02, 0.5]

NH4 Codebook (example)

4-bit Code	Value (Float)
0000	-1.51
0001	-1.14
0010	-0.91
0011	-0.76
0100	-0.59
0101	-0.44
0110	-0.30
0111	-0.18
1000	-0.06
1001	0.06
1010	0.18
1011	0.30
1100	0.44
1101	0.59
1110	0.76
1111	1.00

Step 1: Normalize the weights

- Group the weights (mostly in the group of 64, here we have only 6 values, so one group).

[0.05, -0.3, 0.95, 0.0, -0.02, 0.5]

- Calculate group statistics (mean and standard deviation).

mean = 0.1967, std = 0.411

- Normalize each value using z-score normalization (zero mean, unit variance).

$$x_{norm} = \frac{x - \mu}{\sigma}$$

[-0.357, -1.209, 1.833, -0.478, -0.527, 0.736]

- (Optional for NF4) Clip or scale before quantization to fit within the representable range.

NF4 usually maps values to a finite set of bin centers within a bounded range like $[-2, 2]$ or so.

x_clipped = max(min(x_norm, 2.0), -2.0)

[-0.357, -1.209, 1.833, -0.478, -0.527, 0.736]

NF4 is *learned*: The real codebook values are derived from statistical modeling of weight distributions across large LLMs.

How it Works

Let's convert FP32 to NH4

weights_fp32 = [0.05, -0.3, 0.95, 0.0, -0.02, 0.5]

NH4 Codebook (example)

4-bit Code	Value (Float)
0000	-1.51
0001	-1.14
0010	-0.91
0011	-0.76
0100	-0.59
0101	-0.44
0110	-0.30
0111	-0.18
1000	-0.06
1001	0.06
1010	0.18
1011	0.30
1100	0.44
1101	0.59
1110	0.76
1111	1.00

Step 2: Let's map each value in your list to the closest entry in the codebook.

1. -0.357

- Closest to: 0101 (-0.44)
- ✖ Error = $|-0.357 - (-0.44)| = 0.083$
- ✓ Quantized Code: 0101

2. -1.209

- Closest to: 0001 (-1.14)
- ✖ Error = $|-1.209 - (-1.14)| = 0.069$
- ✓ Quantized Code: 0001

3. 1.833

- No value above 1.0 in codebook
- ✖ Clip to maximum → use 1111 (1.00)
- ✓ Quantized Code: 1111

4. -0.478

- Closest to: 0101 (-0.44)
- ✖ Error = 0.038
- ✓ Quantized Code: 0101

5. -0.527

- Closest to: 0101 (-0.44)
- ✖ Error = 0.087
- ✓ Quantized Code: 0101

6. 0.736

- Closest to: 1110 (0.76)
- ✖ Error = 0.024
- ✓ Quantized Code: 1110

Final Values

[-0.44, -1.14, 1.0, -0.44, -0.44, 0.76]

Binary Values

['0101', '0001', '1111', '0101', '0101', '1110']

E NormalFloat 4-bit data type

The exact values of the NF4 data type are as follows:

[-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453, -0.28444138169288635, -0.18477343022823334, -0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224, 0.44070982933044434, 0.5626170039176941, 0.7229568362236023, 1.0]

NF4 is *learned*: The real codebook values are derived from statistical modeling of weight distributions across large LLMs.

How it Works

Step 2: Freeze base weights

The quantized model weights are frozen (not updated during training). This saves computation and avoids catastrophic forgetting.

Step 3: Inject LoRA adapters

Mathematically, LoRA modifies the forward pass of a weight matrix.

Let's say we have a layer:

$$y = Wx$$

Where:

- $W \in \mathbb{R}^{n \times m}$ is a weight matrix
- $x \in \mathbb{R}^n$ is the input

In LoRA: $y = (W + \Delta W)x$, Where: $\Delta W = BA$

- $A \in \mathbb{R}^{n \times r}$ and $B \in \mathbb{R}^{r \times m}$, where $r \ll \min(n, m)$
- These matrices are trainable, but small (low-rank)

Now in QLoRA, W is 4-bit quantized, and **$\Delta W = BA$ is still float16 or float32.**

Only BA is trained, keeping W fixed (but quantized).

How it Works

Step 4: Use Double Quantization

Double Quantization refers specifically to the two-layer compression scheme:

- **Primary Quantization (NF4):**
 - Converts 32-bit float weights to 4-bit codes (using a normal-distribution-aware codebook).
 - Applied group-wise, typically over 64 values per group.
- **Secondary Quantization (of Scales):**
 - Instead of storing full-precision float32 scale values for each group, they are further quantized into 8-bit integers.
 - This reduces memory even more without degrading model quality.

So, the “double” refers to:

- First: the weights \rightarrow 4-bit
- Second: the scaling factors \rightarrow 8-bit

```
def forward(x):  
    W_approx = dequantize(W_4bit)    # Read-only  
    delta_W = B @ A                  # Low-rank adapter (trainable)  
    return (W_approx + delta_W) @ x
```

Summary

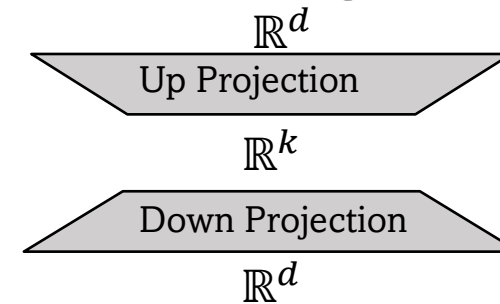
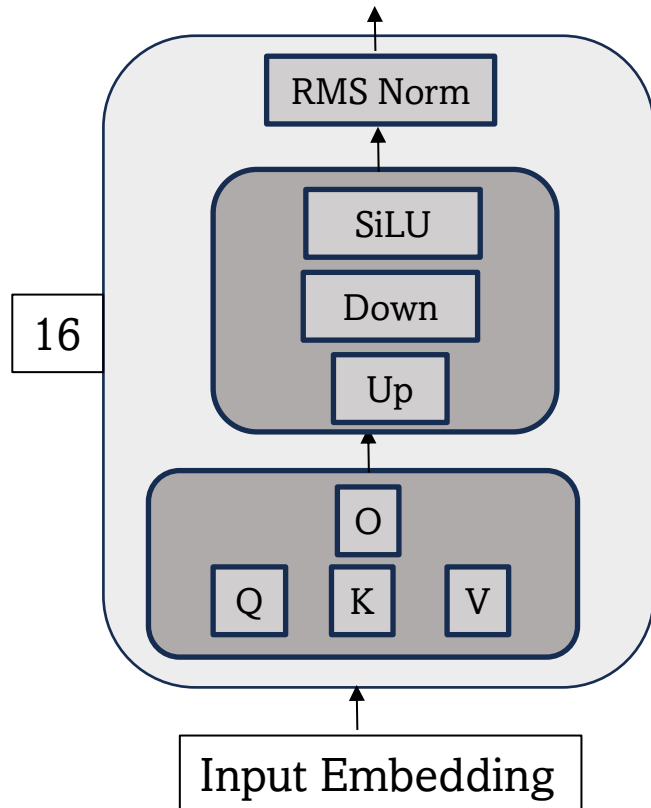
$$y = (w + \Delta w)x$$
$$y = (W_{NF4} + BA_{FP16/32})x$$
$$y = (dequantization(W_{NF4})_{Fp16/32} + BA_{FP16/32})x$$

Feature	Description
Base model (W)	Stays in compressed 4-bit form
LoRA adapters (BA)	Loaded in FP16 or FP32
Dequantization	Happens per group, during matrix ops
No training involved	Just efficient matrix math
Speed	Fast due to reduced memory bandwidth

Adapters & Prefix-Tuning

Adapters

Adapters are small neural networks (usually bottleneck MLPs) inserted between layers of a frozen pretrained model. Only these adapter modules are trained during fine-tuning — the main model weights remain untouched.



Let's say we have the output from the first transformer layer is 2048

$$h \in \mathbb{R}^d \approx h \in \mathbb{R}^{2048}$$

Down projection $\in \mathbb{R}^{2048 \times 64}$, where 64 is the dimension of the down projection (bottleneck size)

Up projection $\in \mathbb{R}^{64 \times 2048}$, Projecting it to the original dimension

$$h^{final} = h + h^{adapter}$$

[h] --> [DownProj (2048→64)] --> [ReLU] --> [UpProj (64→2048)] --> [+] --> [Enhanced h]

Adapters & Prefix-Tuning

Prefix Tuning

Instead of modifying model weights, Prefix-Tuning prepends learnable vectors (prefixes) to the key and value matrices in attention layers. These vectors guide the model's behavior in a task-specific way.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

Where:

- $Q = X W_Q$, $K = X W_K$, $V = X W_V$
- X is the input (sequence of token embeddings)
- W_Q, W_K, W_V are learned weight matrices

Instead of changing W_Q, W_K, W_V , we:

- Keep the model **completely frozen**
- Learn small **prefix vectors**: P_K, P_V of shape $(\text{prefix_len}, d_{\text{model}})$
- At inference/training time, we **prepend** these learned vectors to the K and V matrices:

$$K' = \text{concat}(P_K, K), \quad V' = \text{concat}(P_V, V)$$

So the attention becomes:

$$\text{Attention}(Q, [P_K; K], [P_V; V])$$

Only P_K, P_V are trained \rightarrow rest of the model remains unchanged.

Adapters & Prefix-Tuning

Why are Adapters & Prefix-Tuning important?

- **Continual Learning:** Adapt to new tasks without retraining entire model.
- **Multi-tasking:** Store separate adapter/prefix for each task.
- **Low compute:** Avoid full model fine-tuning.
- **Research use:** Try many tasks with one base model.
- **Deployment friendly:** Small memory footprint per task.

Summary

Feature	LoRA / QLoRA	Adapters	Prefix-Tuning
Frozen Base Weights	Yes	Yes	Yes
Add Parameters	Weight Deltas (W)	Between layers (Residual MLP)	Key/Value attention matrices
Good For	Low-memory fine-tuning	Continual, multi-task learning	Prompt-style task control
Training Size	Very low	Low	Very low
Math Core	$W \approx AB$ (Low-rank)	$h + W_2(\text{ReLU}(W_1 h))$	$K' = [P_k ; K], V' = [P_v ; V]$
Architecture Change	Small in weights	Extra MLP block in layers	Prefix prepended in attention

Thanks for
your time