Lecture 11.2: Parameter-Efficient Fine-Tuning

From Full Fine-Tuning to Low-Rank Adaptation

Heman Shakeri

Recap: The Adaptation Challenge

When you can get away with no fine-tuning, you absolutely should!

- In-Context Learning, Prompt Engineering, RAG

Now: When fine-tuning IS necessary, how do we do it efficiently?

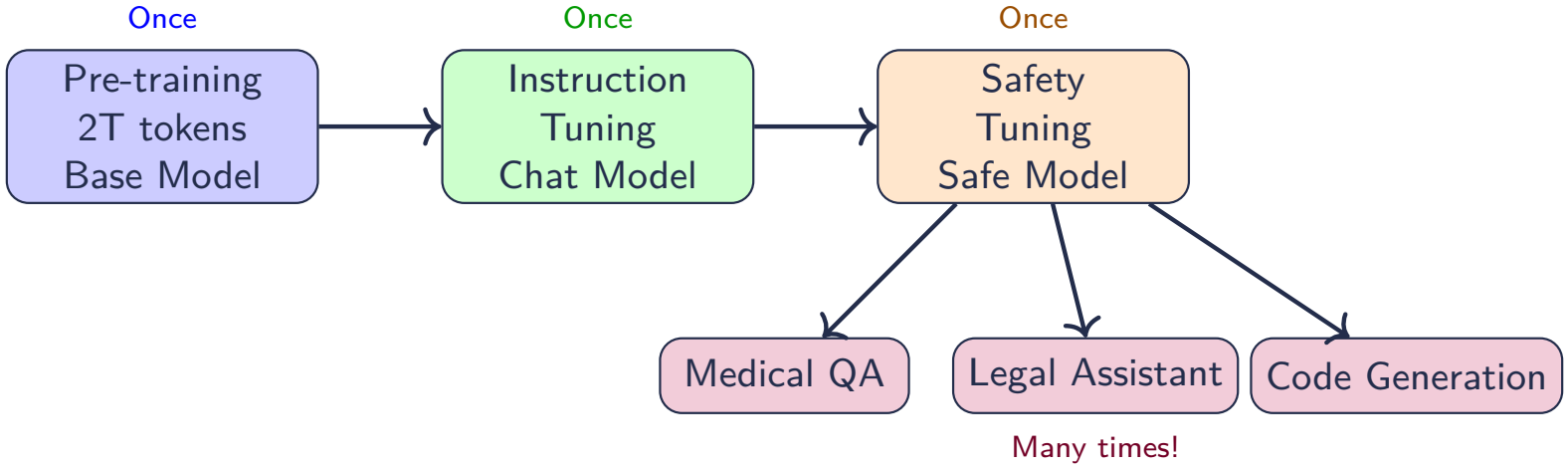
The adaptation imperative: Two-stage process

1. **Pre-training**: Large-scale, self-supervised learning on vast corpora (e.g., 2 trillion tokens)
2. **Adaptation**: Specialization to specific downstream tasks or domains

The bottleneck: As models scale to hundreds of billions of parameters, traditional adaptation methods become computationally and financially prohibitive

Central question: How can we efficiently adapt massive pre-trained models without the costs of full fine-tuning?

The Modern LLM Training Pipeline

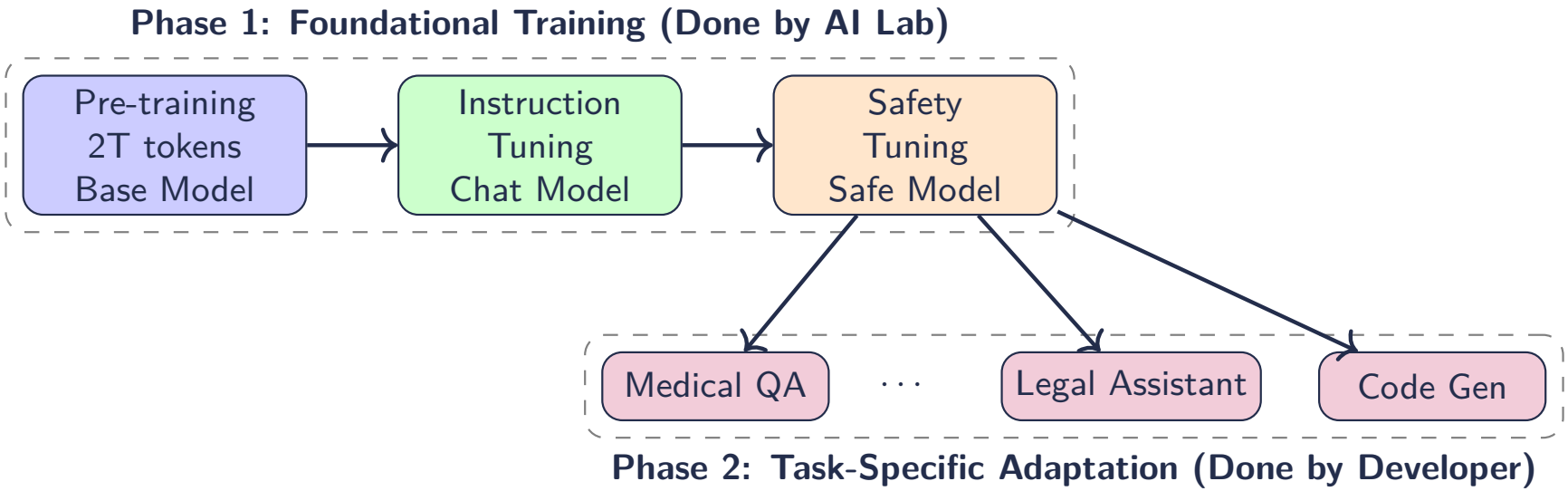


The challenge: Steps 2-4 all involve fine-tuning

There are **unlimited possibilities** for task-specific applications

The “Many times!" arrow is the **core economic driver for PEFT**. The cost of the first three steps is amortized, but the “Task-specific" step is repeated 1,000s of times. The cost of this step *must* be near-zero for the ecosystem to be viable.

The Modern LLM Training Pipeline



The Core Distinction:

- **Phase 1 (Foundational)**: Done **once** by the model creator. This is a massive, multi-million dollar process to create the single, powerful model that is released.
- **Phase 2 (Adaptation)**: Done **many times** by developers. Every time you want a new, specialized skill, you run this step.

The PEFT Imperative:

- All stages in Phase 1 and Phase 2 involve fine-tuning.
- To make this ecosystem work, **Phase 2 must be extremely cheap,**

Full Fine-Tuning as the Performance Gold Standard

Definition: Full Fine-Tuning (FFT) updates *all* parameters θ of a pre-trained model during adaptation

Maximum expressive capacity: By updating every weight, FFT provides maximum flexibility for the model to adapt to task nuances

FFT serves as the **performance benchmark**. PEFT methods are almost always evaluated by their performance *relative to FFT* (e.g., “achieves 98.2% of FFT performance on GLUE").

The problem: FFT is prohibitively expensive in three key dimensions:

1. Computational expense
2. Memory footprint
3. Storage inefficiency

FFT Cost 1: Computational Expense

FFT demands immense computational resources:

Hardware requirements:

- Requires clusters of high-end GPUs (A100s, H100s)
- Single fine-tuning run: thousands to tens of thousands of dollars
- Limits experimentation and accessibility for smaller organizations

Example with Gemma 3 270M:

- Small enough for single GPU (much more accessible than 8B+ models)
- But still: full fine-tuning requires GPU memory and time
- Multiple experiments $\times$ multiple tasks $=$ significant cost

PEFT promise: Reduce computational requirements by orders of magnitude

FFT Cost 2: Memory Footprint

VRAM consumption during training has three components:

Component	Size (16-bit)	Gemma 3 270M
Model weights	$2N$ bytes	540 MB
Gradients	$2N$ bytes	540 MB
Optimizer states (Adam)	$8N$ bytes	2.16 GB
Total	**12N** bytes	**3.24 GB**

The **Optimizer states** are the bottleneck. Adam/AdamW stores two moving averages (momentum m_t and variance v_t) for *every single parameter*.

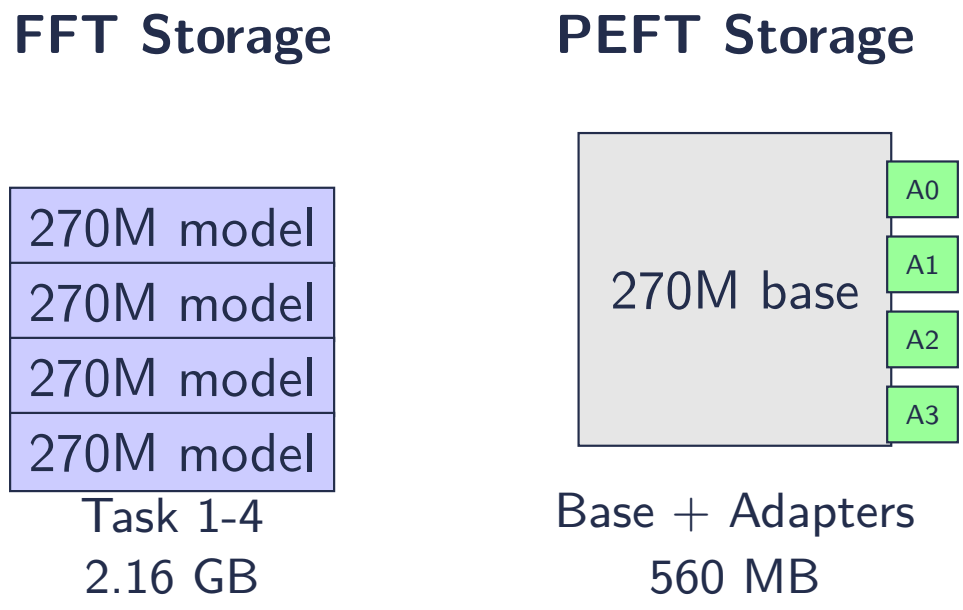
- m_t (4 bytes) + v_t (4 bytes) = 8 bytes.
- Total = $2N$ (weights) + $2N$ (grads) + $8N$ (Adam) = $12N$.
- This $12N$ (or $16N$ for 32-bit) cost is the specific problem that PEFT methods, which reduce the number of *trainable* parameters, are designed to solve.

For larger models:

- 7B model: ~84 GB (requires A100)
- 70B model: ~840 GB (requires multiple GPUs)

FFT Cost 3: Storage Inefficiency

FFT produces a **complete, new copy** of the entire model for every task



This is the **deployment and MLOps nightmare**.

- 1,000 tasks = 1,000 separate, massive model files to host, version, and serve.
- The PEFT model is 1 shared base model + 1,000 tiny (megabyte-sized) adapter files. This is a fundamentally superior **systems architecture**.

The Representational Challenge: Catastrophic Forgetting

Beyond resource constraints, recall the fundamental problem facing FFT:

Catastrophic Forgetting

When a model is fine-tuned on Task B, it often loses proficiency on previously learned Task A
Gradient updates for the new task **overwrite** the knowledge representations crucial for the old task

Implication: FFT is fundamentally unsuitable for applications requiring incremental knowledge accumulation

Important caveat: PEFT methods can also suffer from forgetting!
Parameter efficiency alone does not guarantee preservation of prior knowledge

The PEFT Paradigm

Goal: Achieve FFT-level performance while updating a tiny fraction of parameters (often $<1\%$)

This is the “magic" of PEFT. How can updating $<1\%$ of parameters equal the performance of updating 100%?

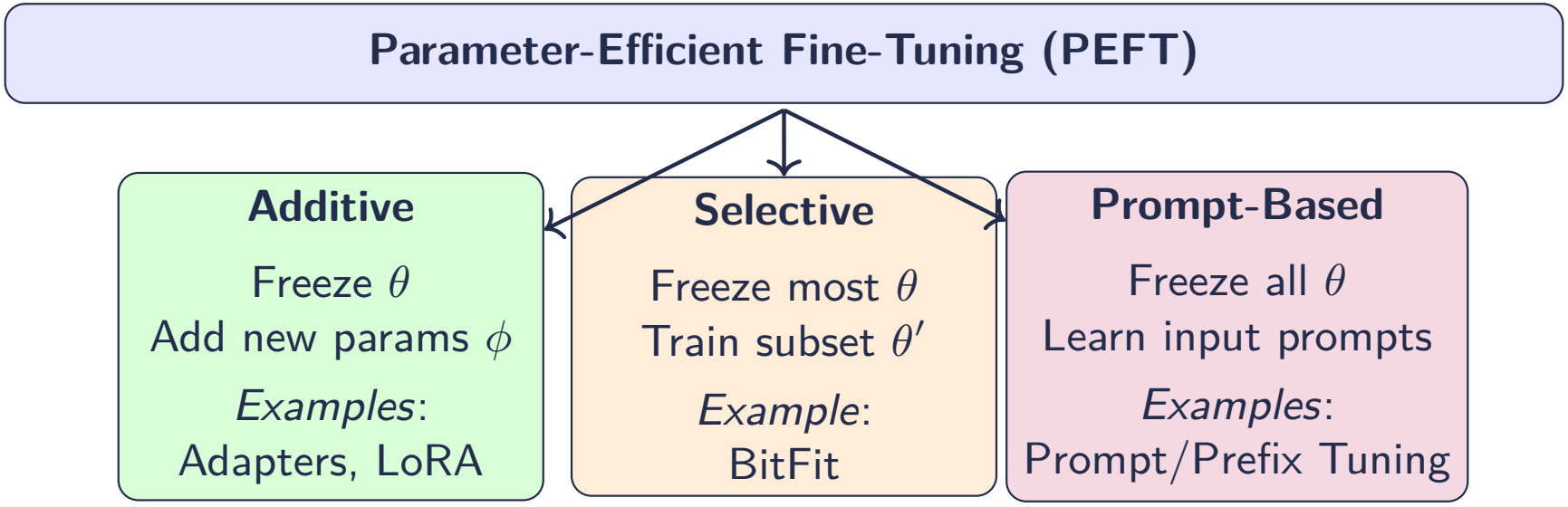
- **Answer**: The pre-trained models are **massively over-parameterized** for any single downstream task. The “adaptation" needed is much simpler than the “pre-training" task.

Benefits:

- Drastically reduce computational costs
- Drastically reduce memory requirements
- Drastically reduce storage costs

Organization principle: How do methods interact with original model parameters θ?

The Three-Category Framework



Three categories:

1. **Additive**: Freeze θ, add new trainable parameters ϕ
2. **Selective**: Freeze most weights, train small subset $\theta' \subset \theta$
3. **Prompt-Based**: Freeze entire model, optimize learnable "virtual tokens"

Master Taxonomy Table

Category	Core Hypothesis	Method	Paper
Additive	Bottleneck modules	Adapters	Houlsby (2019)
Additive	Low intrinsic rank	LoRA	Hu (2021)
Selective	Bias-centric	BitFit	Ben-Zaken (2021)
Prompt	External steering	Prompt Tuning	Lester (2021)
Prompt	Layer-wise steering	Prefix Tuning	Li & Liang (2021)

Key insight: Each method has a **core hypothesis** about how adaptation works

Understanding the hypothesis $\rightarrow$ Understanding when to use each method

Orthogonal Techniques

We distinguish PEFT principles from complementary techniques that we will discuss later:

Quantization (e.g., 4-bit, 8-bit):

- Reduces memory footprint via lower-precision data types

Data-Centric Tuning:

- Focuses on *what* the model learns (e.g., instruction tuning, RLHF) rather than *how* parameters are updated

The Bottleneck Hypothesis: Classic Adapters

Paper: Houlsby et al. (2019), "Parameter-Efficient Transfer Learning for NLP"

Core Hypothesis: A large pre-trained model can be adapted by inserting a small number of new, task-specific parameters while leaving original weights untouched

Key idea: Insert small feedforward networks ("adapters") within each Transformer layer

Parameter-Efficient Transfer Learning for NLP

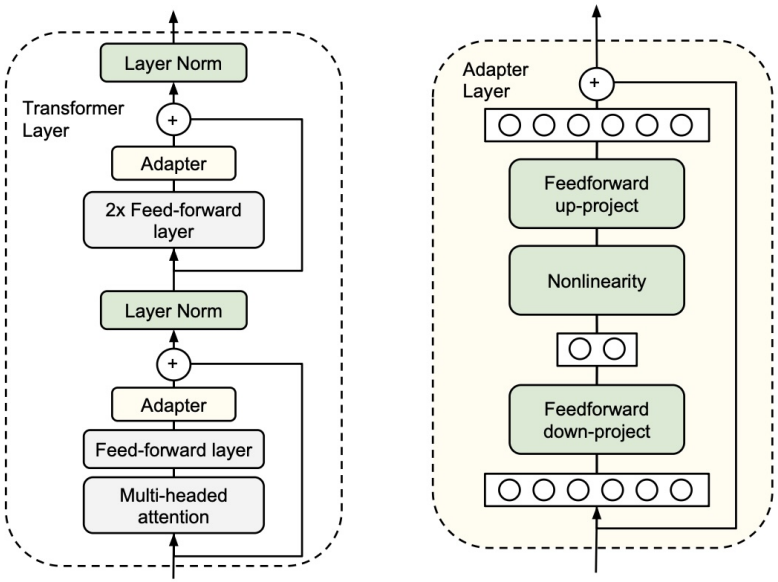


Figure 2. Architecture of the adapter module and its integration with the Transformer. **Left:** We add the adapter module twice to each Transformer layer: after the projection following multi-headed attention and after the two feed-forward layers. **Right:** The adapter consists of a bottleneck which contains few parameters relative to the attention and feedforward layers in the original model. The adapter also contains a skip-connection. During adapter tuning, the green layers are trained on the downstream data, this includes the adapter, the layer normalization parameters, and the final classification layer (not shown in the figure).

[Houlsby et al. (2019)]

Critical Trade-off: Serial Architecture

Impact on Inference

Because adapters are **separate modules executed sequentially**, they introduce additional computational steps and thus **increase latency during inference** for every layer

This is not a minor implementation detail—it's a fundamental consequence of the serial architectural choice

Engineering Pointer: This is the **key drawback** of Adapters for production.

- The main Transformer block must **wait** for the adapter block to finish before the residual connection can be computed.
- Total latency $= t_{\text{base}} + N_{\text{layers}} \times t_{\text{adapter}}$.

Trade-off:

\+ High modularity (easy to stack, combine for multi-task)

\- Unavoidable latency overhead

The Low-Rank Adaptation Hypothesis

Paper: Hu et al. (2021), "LoRA: Low-Rank Adaptation of Large Language Models"

Core Hypothesis: The change in a model's weight matrix during adaptation, ΔW, has a very low "intrinsic rank"

Implication: ΔW can be efficiently approximated by the product of two much smaller matrices

Mathematical insight: Instead of learning full update $\Delta W \in \mathbb{R}^{d \times k}$, decompose it:

$$\Delta W \approx BA$$

where:

- $B \in \mathbb{R}^{d \times r}$ (down-projection)
- $A \in \mathbb{R}^{r \times k}$ (up-projection)
- $r \ll \min(d, k)$ (rank, typically 8–64)

LoRA: Parameter Count

Parameter comparison:

- Original: $d \times k$ parameters
- LoRA: $dr + rk = r(d + k)$ parameters
- Reduction factor: $\frac{r(d+k)}{dk} \approx \frac{2r}{d}$ (when $d \approx k$)

Concrete example: Update a 5×5 weight matrix (25 parameters)

Rank $r = 1$ decomposition:

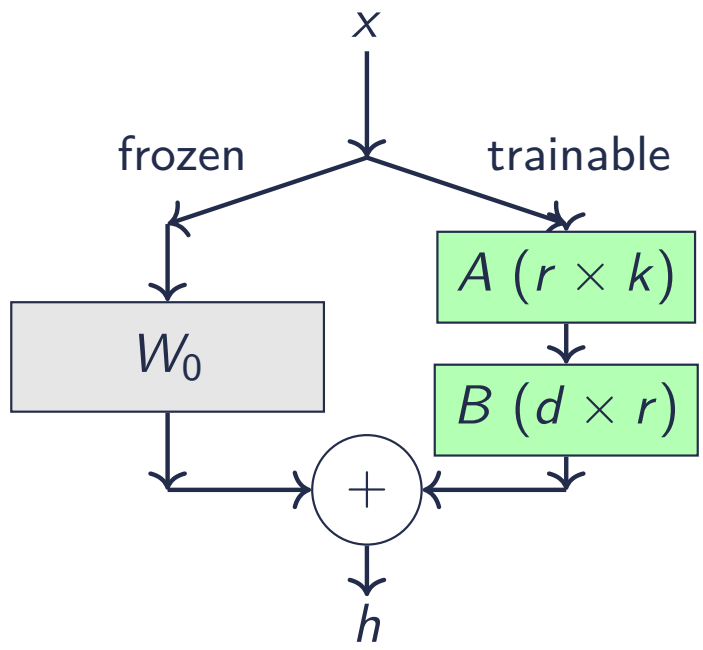
$$\underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}}_{\text{5 params}} \times \underbrace{\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \end{bmatrix}}_{\text{5 params}} = \underbrace{\begin{bmatrix} b_1 a_1 & \cdots & b_1 a_5 \\ \vdots & \ddots & \vdots \\ b_5 a_1 & \cdots & b_5 a_5 \end{bmatrix}}_{\text{25 values}}$$

Trade-off: 10 trainable parameters vs 25 full parameters

Sacrifice some precision for dramatic efficiency. As matrices grow, savings become massive.

LoRA: Parallel Architecture

Key distinction from Adapters: LoRA injects trainable matrices *in parallel* to existing linear layers



Modified forward pass:

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

Compare this to the Adapter's $h \leftarrow h + \text{Adapter}(h)$, which happens *after* the main layer.

The Zero-Latency Advantage

Critical Advantage: Merging for Zero Latency

Because the LoRA update is a **linear operation added to another linear operation**, the matrices can be algebraically merged after training:

$$W' = W_0 + BA$$

Engineering Pointer: This makes LoRA great for production.

- **During training**: You keep W_0 frozen and only train B and A.
- **For deployment**: You compute $W_{\text{merged}} = W_0 + BA$ *once*, offline.
- You then deploy a model with the new W_{merged} weights. This merged model has the **exact same architecture and latency** as the original.

Fundamental advantage: This property was unavailable to serial architectures like Adapters, again:

Deployment workflow:

1. Train with parallel LoRA path
2. Merge: $W' = W_0 + BA$
3. Deploy merged model (no structural changes)

Where to Apply LoRA

Can apply to any linear layer in the Transformer:

- Query, Key, Value projections (W_Q, W_K, W_V)
- Output projection (W_O)
- FFN layers (W_1, W_2)

Critical Finding: Train All Layers

Research shows that **training ALL layers is essential** to match full fine-tuning performance
Applying LoRA to only some layers (e.g., just attention) gives worse results

Recommendations:

- **Minimum**: Apply to W_Q and W_V in all attention layers
- **Best results**: Apply to all linear projections in all layers

Hyperparameter Selection: Rank r

The LoRA paper shows that **performance is insensitive to rank** r in a wide range. Performance for $r = 4, 8, 32, 64$ is often very similar.

This strongly supports the *low-rank hypothesis*—the intrinsic rank of adaptation is indeed very low (e.g., $r = 1$ or $r = 2$ already performs well).

Practical recommendations:

- **Start with** $r = 16$ **or** $r = 64$: Good balance for most tasks.
- **Low** $r = 8$: Very efficient, often “good enough" for simple adaptations.
- **High** $r = 128$–256: For complex tasks (e.g., teaching a new skill) if $r = 64$ is insufficient.

LoRA Parameter Counts for Gemma 3 270M

Model	Params	Rank	LoRA Params	% Original
Gemma 3	270M	8	160K	0.06%
Gemma 3	270M	16	320K	0.12%
Gemma 3	270M	64	1.3M	0.48%
		For comparison:		
7B	7B	8	4.2M	0.06%
7B	7B	64	33.6M	0.48%
70B	70B	64	336M	0.48%

Key observation: Percentage stays constant across model scales

- A 200x reduction in trainable parameters (0.48% vs 100%) means a 200x reduction in the **optimizer state memory** (the $8N$ part of the $12N$ problem).
- This is what allows fine-tuning on a single consumer GPU.
- Storage per task (the adapter file) is $\sim$5 MB, not $\sim$540 MB.

Hyperparameter Selection: Alpha α

Controls how much LoRA updates affect original weights:

$$h = W_0 x + \frac{\alpha}{r} \cdot BAx$$

This scaling factor α/r is a simple hyperparameter to normalize the update magnitude.

- B is initialized to 0, so the initial update is 0.
- A is initialized with Kaiming uniform.
- The scaling factor acts like a fixed scalar on the learning rate for the LoRA weights.

Common settings:

- Original LoRA: $\alpha = r$ (scaling factor $= 1$).
- Common practice: $\alpha = 2 \times r$ (e.g., $r = 16, \alpha = 32$).
- Q-LoRA paper: $\alpha = 16, r = 64$ ($0.25\times$ scaling).

Practical tip: A common strategy is to **set** $\alpha = r$ and then only tune the learning rate.

Other Hyperparameters

Dropout (from Q-LoRA paper):

- Smaller models (270M-7B): dropout $= 0.1$
- Larger models (33B, 65B+): dropout $= 0.05$

Training hyperparameters:

Model Size	Learning Rate	Batch Size	Epochs
270M-7B	2×10^{-4}	16	3
13B	2×10^{-4}	16	3
33B	1×10^{-4}	16	3
65B+	1×10^{-4}	16	3

For Gemma 3 270M: Use settings for smaller models

Adapters vs LoRA: Fundamental Architectural Choice

Dimension	Adapters	LoRA
Architecture	Sequential bottleneck	Low-rank parallel
Inference latency	Adds latency	Zero latency
Modularity	High (easy to combine)	Lower (monolithic)
Hypothesis	Architectural solution	Mathematical hypothesis
Merging	Cannot merge	Can merge weights

This is not an implementation detail—it's a fundamental design choice with direct consequences for deployment

Trade-offs:

- Adapters: Better for multi-task systems, worse for latency
- LoRA: Better for production deployment, harder to combine

LoRA Implementation (Part 1)

```python
import torch
import torch.nn as nn

class LoRALayer(nn.Module):
    def __init__(self, in_features, out_features, rank=16, alpha=16):
        super().__init__()
        self.rank = rank
        self.alpha = alpha

        # Initialize A with small random values, B with zeros
        self.lora_A = nn.Parameter(
            torch.randn(in_features, rank) * 0.01
        )
        self.lora_B = nn.Parameter(
            torch.zeros(rank, out_features)
        )

        # Scaling factor
        self.scaling = alpha / rank

    def forward(self, x):
        # Low-rank update with scaling
        return (x @ self.lora_A @ self.lora_B) * self.scaling
```

Key design choices:

- Initialize A with small random values (break symmetry)
- Initialize B with zeros (start with identity behavior)
- Apply scaling factor α/r to control update magnitude

LoRA Implementation (Part 2)

```python
class LoRALinear(nn.Module):
    def __init__(self, linear_layer, rank=16, alpha=16):
        super().__init__()
        # Freeze original weights
        self.linear = linear_layer
        self.linear.weight.requires_grad = False
        if self.linear.bias is not None:
            self.linear.bias.requires_grad = False

        # Add LoRA adapter
        self.lora = LoRALayer(
            linear_layer.in_features,
            linear_layer.out_features,
            rank, alpha
        )

    def forward(self, x):
        # Original + LoRA update
        return self.linear(x) + self.lora(x)

    def merge_weights(self):
        ``"""Merge LoRA weights into base weights for inference"""
        with torch.no_grad():
            # Compute BA and add to original weights
            delta_w = self.lora.lora_B @ self.lora.lora_A
            self.linear.weight.data += (delta_w.T * self.lora.scaling)
```

Critical method: `merge_weights()` enables zero-latency deployment!

The Bias-Centric Hypothesis: BitFit

Paper: Ben-Zaken et al. (2021), "BitFit: Simple Parameter-efficient Fine-tuning"

Core Hypothesis: Fine-tuning is primarily a process of *exposing or redirecting* knowledge already acquired during pre-training

This can be achieved by modifying only the bias terms

Method:

- Freeze all weight matrices W
- Train only bias vectors b

Parameter count: Typically $<0.1\%$ of model parameters

No architecture changes: Model structure remains completely unchanged

The Bias-Centric Hypothesis: BitFit [Ben-Zaken et al. (2021)]

Core Hypothesis: "finetuning is mainly about **exposing knowledge** induced by language-modeling training, **rather than learning new** task-specific linguistic knowledge."

Method (BitFit):

- Freeze **all** weight matrices W in the entire model.
- Train **only the bias vectors** b (and the LayerNorm parameters).

Scientific Pointer: This is the **most parameter-efficient** method.

- **Parameter count**: Typically $<$**0.1%** of model parameters (e.g., 0.08% for BERT-Large).
- The hypothesis is that the W matrices are already universal feature extractors. The bias terms b act as "knobs" or "gates" to shift and rescale these features for the new task.

The External Steering Hypothesis: Prompt-Based Methods

Prompt-based methods represent the *least intrusive* category

Core principle: Frozen model's behavior can be "steered" without modifying any internal weights

Two main approaches:

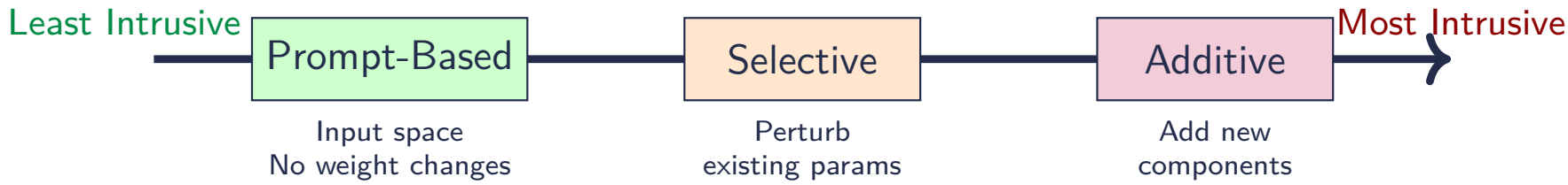
1. **Prompt Tuning (Lester et al. 2021)**:
   - Prepend continuous embedding vectors ("soft prompts") to input
   - Only soft prompt embeddings are updated via backpropagation
2. **Prefix Tuning (Li & Liang 2021)**:
   - Insert trainable prefix vectors into hidden states of *every layer*
   - More fine-grained control than Prompt Tuning

Forward pass (Prompt Tuning):

$$\text{input} = [\text{soft\_prompt}_1, \ldots, \text{soft\_prompt}_k, \text{text\_embeddings}]$$

The Intrusiveness Spectrum

These categories form a spectrum of how "intrusively" they modify the base model:



Left (Prompt-Based):

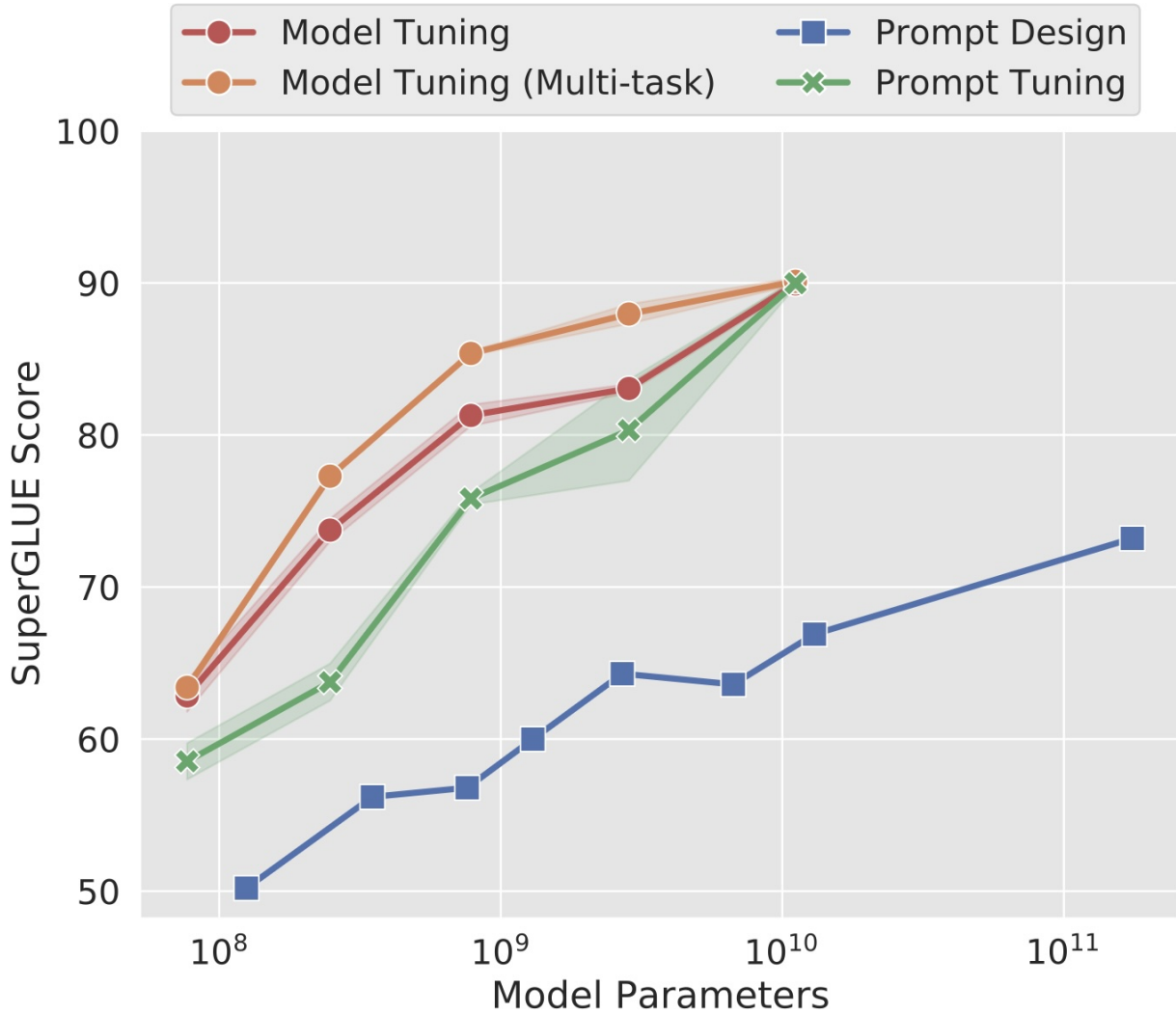
- Completely non-invasive to model
- Ideal for multi-tenant serving (one model, many tasks)

Right (Additive):

- More invasive but more expressive
- Better performance on complex tasks

Prompt-based methods exhibit strong dependence on model scale

The larger and more capable the frozen model's internal knowledge base, the more effectively it can be steered by a small, externally learned prompt.



Model tuning of T5 [Lester et al. (2021)]

Comparative Analysis Table

Method	Core Hypothesis	Trainable	Inference	Storage
Adapters	Bottleneck modules	0.1–4%	Adds latency	Small
LoRA	Low-rank updates	0.1–2%	Zero overhead	Small
BitFit	Bias-centric	$<$0.1%	Zero overhead	Tiny
Prompt Tuning	External steering	$<$0.01%	Slight	Tiny
Prefix Tuning	Layer-wise steering	$<$0.1%	Slight	Tiny

Design Trade-offs

1. The Latency vs Modularity Dilemma

Adapters:

+ High modularity—easy to stack or combine for multi-task learning
- Increased inference latency (unavoidable due to serial architecture)

LoRA:

+ Zero latency after weight merging
- Creates monolithic model—dynamic task combination is complex

2. The Performance vs Efficiency Frontier

BitFit: Extreme parameter efficiency, good for low-data regimes, may underperform on complex tasks

LoRA: Uses more parameters but typically achieves performance closest to FFT. **Robust default choice** for most applications

3. The Intrusiveness vs Scale Dependency

Prompt-based methods:

+ Non-intrusive—ideal for multi-tenant serving
- Strong dependency on very large model scales ($>$10B)
- For smaller models like Gemma 3 270M, performance substantially lower

For Gemma 3 270M specifically:

- **Best choice**: LoRA (good performance, zero latency)
- **Alternative**: Adapters (if modularity matters)
- **Experimental**: BitFit (for extreme efficiency)
- **Avoid**: Prompt-based methods (model too small)