Supervised Fine-Tuning of LLMs with Parameter-Efficient Methods: LoRA, DoRA, and VeRA

Alberto Rodero* Pablo Lobato*

September 2025

Abstract

Large Language Models (LLMs) are increasingly being adapted to specialized tasks through fine-tuning. While full-model fine-tuning offers strong performance, **it is computationally expensive and difficult to scale**. **Parameter-Efficient Fine-Tuning (PEFT) techniques**, such as LoRA, DoRA and VeRA, have emerged as practical alternatives. We provide a **comparative perspective on these techniques, illustrating their trade-offs for practical deployment** of LLMs.

1 Introduction

Large Language Models (LLMs) have become the foundation for many modern applications in natural language processing, powering chatbots, code assistants, search systems and domain-specific solutions. Adapting these models to new tasks often requires fine-tuning. However, **directly updating billions of parameters is both computationally expensive and difficult to manage in production systems**. To address these limitations, **parameter-efficient fine-tuning (PEFT) methods have emerged**. Instead of modifying all parameters, these approaches **introduce small trainable components** into the model architecture, enabling adaptation with minimal additional cost. The field is rapidly evolving, with new methods appearing frequently and existing ones being refined for stability, scalability, and precision. This document focuses on **Supervised Fine-Tuning (SFT)** with LoRA and its extensions (DoRA, VeRA), highlighting their conceptual design, trade-offs and practical usage. Beyond these methods, other promising approaches include **AdaLoRA**[1] (adaptive rank adjustment) and **QLoRA**[2] (quantization-aware LoRA).

We focus on **LoRA and its modern derivatives, DoRA and VeRA**, as they are among the most influential, empirically validated, and widely adopted PEFT strategies in current LLM research and practice.

2 Background

2.1 Supervised Fine-Tuning (SFT)

Unlike pretraining, which focuses on learning general language representations, **Supervised Fine-Tuning (SFT) is the process of aligning an LLM with specific tasks** such as instruction following, classification or summarization, this process uses a dataset with labeled examples. SFT typically involves:

- Using a **pretrained LLM as a frozen initialization**.
- Training on a **curated dataset of input-output pairs**.
- Minimizing a supervised loss (e.g., cross-entropy) over the model predictions.

*Equal contribution

2.1.1 Challenges of Full Fine-Tuning

Although full fine-tuning has been widely used and has proved his efficacy, **applying it directly to LLMs introduces several difficulties**:

- **Computational Cost:** Updating billions of parameters requires **significant GPU memory and training time**, often accessible only to large organizations.
- **Storage Overhead:** Each fine-tuned variant of a model **must be stored in full**, leading to inefficient use of storage when multiple task-specific versions are needed.
- **Deployment Complexity:** Switching between fully fine-tuned models for different tasks **complicates serving pipelines** and increases infrastructure requirements.
- **Limited Flexibility:** Once a model is fully fine-tuned, adapting it to additional tasks often **requires repeating the process from scratch**.
- **Catastrophic Forgetting:** Fine-tuning on new tasks without special care can **overwrite previously learned capabilities**, degrading performance on earlier tasks.

These challenges motivated the development of **PEFT methods**, which provide a **more scalable, modular, and flexible way** to adapt LLMs for diverse applications.

2.2 Parameter-Efficient Fine-Tuning (PEFT)

Parameter-Efficient Fine-Tuning (PEFT) methods were introduced to mitigate full-finetuning issues, **allowing the adaptation of LLMs without retraining them entirely**. Instead of storing and training billions of weights for every downstream task, PEFT **introduces lightweight modules or modifies a small subset of parameters**, significantly reducing training cost and storage requirements. Existing approaches include:

- **Adapter layers:** Small neural modules inserted into each Transformer block [3]. While effective, they **increase inference latency**.
- **Prompt-based tuning:** Optimizing embeddings or continuous prompts [4]. These methods can **reduce effective sequence length** and are harder to optimize for some tasks.
- **LoRA and extensions:** Methods such as LoRA [5], DoRA [7] and VeRA [8] apply low-rank or structured updates to model weights. These approaches **achieve strong efficiency-accuracy trade-offs** and have become the **de facto standard** for parameter-efficient fine-tuning of LLMs.

2.2.1 Low-Rank Adaptation (LoRA)

LoRA adds a few small trainable modules to an existing model instead of retraining all its weights. This makes **adapting large models to new tasks faster and cheaper** while keeping their original capabilities.

A neural network contains many dense layers which perform matrix multiplication. The weight matrices in these layers typically have full-rank. When adapting to a specific task, pre-trained language models have a **low "intrinsic dimension"**[6] and can be fine-tuned successfully despite a random projection to a smaller subspace. Based in this idea, instead of updating all parameters, LoRA **freezes the pretrained weights and injects trainable low-rank matrices** into selected linear transformations, typically in the attention mechanism.

During adaptation, a weight matrix $W_0 \in \mathbb{R}^{d\times k}$ is modified as:

$$W = W_0 + \Delta W, \quad \Delta W = BA,$$

where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ with $r \ll \min(d, k)$. Here, W_0 remains frozen, and only A and B are trained. This decomposition **constrains the updates to a low-rank space, drastically reducing the number of trainable parameters**.

LoRA provides several practical benefits:

- **Parameter Reduction:** LoRA **reduces the number of trainable parameters** while maintaining performance.
- **No Inference Overhead:** The low-rank updates can be **merged with the frozen weights at deployment, ensuring zero additional latency**.
- **Modular Adaptation:** Multiple downstream tasks can be supported by **swapping only the LoRA matrices**, rather than entire model checkpoints.
- **Training Throughput:** LoRA **improves training speed and lowers memory usage**.

While LoRA is widely adopted, it has some limitations:

- **Limited Expressiveness:** The low-rank constraint can **restrict the model's ability to capture highly complex task-specific patterns** compared to full fine-tuning.
- **Layer Sensitivity:** Performance **depends strongly on where LoRA is applied** (e.g., W_q, W_v), requiring experimentation to find the best configuration.

2.2.2 Weight-Decomposed Low-Rank Adaptation (DoRA)

DoRA fine-tunes models more precisely by **separating how strongly and in what direction each weight changes**. This delivers **finer control and improved accuracy over LoRA**, especially for complex reasoning or multimodal tasks.

While LoRA has proven effective for efficient fine-tuning, there remains a **noticeable accuracy gap compared to full fine-tuning** (FT). The DoRA method [7] (Weight-Decomposed Low-Rank Adaptation) addresses this by **reparameterizing pretrained weights into two components: magnitude and direction**. This decomposition enables a **finer-grained adaptation process** that more closely mirrors the behavior of full fine-tuning.

A pretrained weight matrix $W \in \mathbb{R}^{d\times k}$ can be expressed as:

$$W = m \cdot \frac{V}{\|V\|_c},$$

where m is a magnitude vector and V represents the directional component, with normalization across columns. DoRA **freezes the pretrained initialization, keeps m trainable**, and, because the directional component has a large number of parameters, **applies LoRA-style low-rank updates to the directional part**:

$$W' = m \cdot \frac{W_0 + BA}{\|W_0 + BA\|_c}.$$

Here, $A \in \mathbb{R}^{r\times k}$ and $B \in \mathbb{R}^{d\times r}$ define low-rank matrices ($r \ll \min(d, k)$), trained in the same way as in LoRA.

- **Learning Capacity:** By decoupling magnitude and direction, DoRA **better replicates the update patterns of full fine-tuning**, enabling more nuanced adaptation than LoRA.
- **Efficiency:** Despite its increased expressiveness, DoRA introduces **only a marginal increase in trainable parameters** compared to LoRA and **does not add inference overhead**.
- **Closing the Fine-Tuning Gap:** Empirical results show that DoRA **significantly narrows the performance gap** between LoRA and full fine-tuning, particularly on reasoning and multimodal benchmarks.

Although DoRA improves accuracy and stability, it also introduces new considerations:

- **Implementation Complexity:** The decomposition of weights and normalization steps **increase code complexity** and potential integration issues in existing frameworks.
- **Marginal Gains for Simple Tasks:** On straightforward fine-tuning tasks, DoRA's added sophistication **may not yield significant improvements** over LoRA.

2.2.3 Vector-Based Random Matrix Adaptation (VeRA)

VeRA simplifies fine-tuning by **sharing the same random adapter structure across all layers and learning only small scaling vectors**. It enables **ultra-lightweight, scalable customization** of models across many users or applications with minimal storage cost.

LoRA and DoRA reduce fine-tuning costs but **still require storing distinct low-rank adapters per task**, which becomes inefficient when scaling across many users or tasks. VeRA [8] (Vector-based Random Matrix Adaptation) **further minimizes trainable parameters** by introducing **frozen, shared random matrices across all adapted layers**, with small trainable scaling vectors.

Given a pretrained weight matrix $W_0 \in \mathbb{R}^{m\times n}$, LoRA models the update as:

$$\Delta W = BA,$$

with $A \in \mathbb{R}^{r\times n}, B \in \mathbb{R}^{m\times r}$ trained per layer. In contrast, VeRA **freezes a single pair of random matrices A, B, shared across all layers** and introduces diagonal scaling matrices Λ_b, Λ_d parameterized by trainable vectors b, d:

$$\Delta W = \Lambda_b B \Lambda_d A.$$

This design **drastically reduces the number of trainable parameters**, since only the vectors b, d are optimized.

- **Extreme Parameter Efficiency:** VeRA achieves **10–100× fewer trainable parameters than LoRA** while maintaining comparable accuracy.
- **No Extra Inference Cost:** Like LoRA, VeRA **merges updates into the base weights, introducing no runtime latency**.
- **Scalability:** Because random matrices are **shared and reproducible from seeds**, VeRA is particularly suited for **large-scale personalization and multi-task deployment**.

While VeRA achieves extreme parameter efficiency, it comes with trade-offs:

- **Reduced Task-Specific Capacity:** Sharing random matrices across layers **may limit adaptability for highly specialized or complex domains**.
- **Lower Maturity:** VeRA is a newer method and its **integration in mainstream frameworks is still limited** compared to LoRA.

3 Motivation & Context

Our aim. Our objective is to provide a **deployment-oriented, empirically grounded comparison** of supervised fine-tuning strategies—full fine-tuning and parameter-efficient methods (LoRA, DoRA, VeRA)—on a common 0.6B base, contrasted against a larger untuned baseline. We **quantify trade-offs across three axes (efficacy, efficiency, and memory)** using task scores, end-to-end latency, and both training and inference VRAM. We characterize rank scaling within each method, assess cross-task transfer and robustness, and surface **Pareto-optimal configurations** under realistic resource constraints. The goal is **actionable guidance**: which method to choose when training memory is scarce or abundant, whether adapters can approach or surpass full fine-tuning or a larger base, and which configurations minimize serving cost without sacrificing target-level performance.

Key questions.

- **Q1:** Which method delivers the best in-domain efficacy–efficiency balance?
- **Q2:** Do adapters approach or surpass full fine-tuning and larger base models across tasks?
- **Q3:** Does rank materially change the efficacy–latency–memory trade-off within each method?
- **Q4:** Cross-task transfer: which method is most robust and least prone to catastrophic forgetting?
- **Q5:** Which configurations minimize inference costs?
- **Q6:** Under training VRAM constraints, which method offers the best accuracy per memory spent?
- **Q7:** If VRAM constraints force adapters, is fine-tuning still worth it overall?
- **Q8:** With abundant VRAM, does full fine-tuning dominate, or do adapters keep a speed/serve edge?
- **Q9:** Which configurations are Pareto-optimal over efficacy, latency, and memory?

4 Experimental Setup

Models. We study **Qwen3-0.6B**[9] under **full fine-tuning (NoPEFT)** and three PEFT methods—**LoRA**, **DoRA**, **VeRA**—and compare against **Qwen3-0.6B (base)** and **Qwen3-1.7B (base)**. **No quantization** is used.

PEFT configurations. LoRA/DoRA use ranks $r \in \{256, 512\}$; VeRA uses $r \in \{512, 1024\}$. Adapters target standard attention projections; DoRA applies magnitude–direction reparameterization; VeRA shares frozen random bases with trainable vectors. Adapters are merged for evaluation.

Datasets (train → eval). We fine-tune on **OpenMathInstruct-2**[11] and **SQuAD v2**[12] with **1,000 training samples** per dataset. For evaluation, we use **ARC**[10] (4-way MCQ), **OpenMathInstruct-2** (numeric), and **SQuAD v2** (extractive QA), **200 samples each**. This setup stresses **knowledge transfer** across task types.

Framework & training. Implementation uses **Hugging Face Transformers**[13], **TRL** (SFT), and **PEFT** in `bfloat16` on CUDA. Data are formatted as simple chat-style SFT (single input → single target). Global hyperparameters are shared across runs (fixed learning rate, 1 epoch, effective batch size = 8 via gradient accumulation, gradient checkpointing). *For full hyperparameter details, please refer to the code.*

Evaluation protocol & metrics. Decoding is deterministic (`do_sample=False`), with "**reasoning**" **turned off** (no chain-of-thought prompts, no tool use, no special reasoning modes) and a generous `max_new_tokens`. We report **ARC macro-F1**, **OpenMath AbsDiff** (lower is better), and **SQuAD F1**. **Latency** is wall-clock per `generate()` call. **VRAM** is recorded at training and evaluation as *resident* and *all-allocated* peaks.

4.1 Known Limitations & External Validity

Single-seed, single-epoch snapshots on **small evaluation sets** (200/item) mean wider uncertainty; we did not run multi-seed or hyperparameter/prompt sweeps. Our metrics prioritize correctness (F1/AbsDiff) over **end-to-end utility** (calibration, abstention, cost-per-success) and exclude pipeline factors like retrieval latency in RAG. **We use small models due to hardware limits** (0.6B FT; 1.7B base), so patterns **may not replicate at larger scales**. The study is **architecture-specific to Qwen**; other backbones (e.g., Llama, Mistral, Mixtral) **may respond differently** to PEFT/FT choices. Low-level implementation details (kernel fusion, allocator behavior, adapter merging) can introduce **systematic latency/VRAM bias**. Finally, dataset subsampling and formatting choices can shift distributions; treat findings as **comparative patterns within this recipe**, not universal claims.

5 Results by Question & Interpretation

Table 1: Full results across tasks with VRAM stats. Lower is better for Math AbsDiff and VRAM. Latency in seconds. Best *efficacy*, *efficiency* (latency), and *VRAM* within each training block are **bolded**.

			VRAM (GB)				ARC		OpenMath		SQuAD v2	
Base	**PEFT**	*r*	**Train Res.**	**Train All.**	**Eval Res.**	**Eval All.**	**F1 ↑**	**Lat ↓**	**AbsDiff ↓**	**Lat ↓**	**F1 ↑**	**Lat ↓**
Trained on OpenMath												
Qwen3-0.6B	NoPEFT	–	5.688	5.346	**1.428**	**1.329**	0.5171	**0.0593**	**16,540.004**	**0.0466**	7.40	0.2291
Qwen3-0.6B	DoRA	256	5.797	4.448	2.754	2.523	0.5056	0.8938	23,798.036	1.5447	8.48	0.2032
Qwen3-0.6B	DoRA	512	5.630	5.685	2.754	2.523	0.5032	0.9251	23,675.495	1.5422	8.48	0.2008
Qwen3-0.6B	LoRA	256	5.855	4.446	2.754	2.523	0.4999	0.8614	23,919.016	1.4959	8.48	**0.1987**
Qwen3-0.6B	LoRA	512	5.693	5.676	2.754	2.523	0.5039	0.8876	23,913.011	1.5027	8.48	0.2003
Qwen3-0.6B	VeRA	1024	**4.959**	3.366	2.754	2.523	**0.5207**	0.9309	24,974.605	5.3566	**9.40**	0.2057
Qwen3-0.6B	VeRA	512	4.967	**3.357**	2.754	2.523	**0.5207**	0.9414	24,976.899	5.3617	**9.40**	0.2020
Trained on SQuAD v2												
Qwen3-0.6B	NoPEFT	–	**8.807**	6.248	**1.428**	**1.329**	0.4542	**0.1934**	**22,996.637**	**0.2208**	**27.95**	0.2216
Qwen3-0.6B	DoRA	256	9.337	5.463	2.824	2.589	0.5065	0.2853	23,700.784	1.8388	9.83	0.2213
Qwen3-0.6B	DoRA	512	10.353	6.559	2.824	2.589	0.5065	0.2911	23,577.992	1.6392	9.59	**0.1823**
Qwen3-0.6B	LoRA	256	9.657	5.466	2.824	2.589	0.5024	0.2933	23,771.658	1.9503	9.52	0.1993
Qwen3-0.6B	LoRA	512	10.035	6.556	2.824	2.589	0.5031	0.3143	23,772.109	3.0125	9.40	0.1940
Qwen3-0.6B	VeRA	1024	9.403	4.378	2.754	2.523	0.4954	0.9454	24,840.634	5.3053	9.40	0.2040
Qwen3-0.6B	VeRA	512	9.025	**4.369**	2.754	2.523	**0.5207**	0.3135	24,840.733	5.3537	9.40	0.1963
Base models (no fine-tuning)												
Qwen3-0.6B	Base	–	–	–	1.486	1.329	0.4932	**1.1333**	24,843.380	**6.2118**	10.07	**0.2357**
Qwen3-1.7B	Base	–	–	–	3.871	3.427	**0.7986**	3.2897	**742.251**	12.2900	**30.36**	0.2828

Q1. Which method delivers the best in-domain efficacy–efficiency balance?

Full fine-tuning (NoPEFT) is the in-domain winner.

- On *OpenMath* (in-domain), NoPEFT reduces error by **30.1%** vs the best adapter and cuts latency by **96.9%**. On *SQuAD*, NoPEFT lifts F1 by **+191.4%** vs the best adapter, with a **+21.6%** latency increase relative to the fastest adapter; inference VRAM remains **48–49%** lower than adapters.

Takeaway: Prefer NoPEFT for in-domain targets when training memory allows; adapters underperform on both quality and (often) speed in this setup.

Q2. Do adapters approach or surpass full fine-tuning and larger base models across tasks?

No—adapters fall short of both full FT and the larger base.

- **Vs. full FT.** In-domain *OpenMath*, adapters have **+43.1%** higher error than NoPEFT; in-domain *SQuAD*, adapter F1 is lower by **–66%**.

- **Vs. larger base.** The 1.7B base beats 0.6B adapters by **+57.7%** ARC F1, **–96.9%** OpenMath error, and **+208.9%** SQuAD F1.
- **Interpretation.** Adapters restrict updates to a low-rank subspace and limited sites, so they cannot reconfigure global representations or calibration as fully as full FT; VeRA further reduces layer-specific capacity via shared bases. The larger base starts with substantially more representational capacity and stronger priors, which adapters on a smaller base cannot compensate for in single-task SFT.

Takeaway: Use adapters for modularity or constraints, not to exceed full FT or a larger base on quality.

Q3. Does rank materially change the efficacy–latency–memory trade-off within each method?

Not materially in these runs.

- Within-method changes across ranks yield score shifts typically within **≤1–2%** and latency differences within **≤1%** (e.g., *OpenMath* LoRA error varies by **0.03%**, latency by **0.5%**).
- **Interpretation.** Small ranks already capture the useful subspace; extra rank adds redundant capacity, while inference cost is dominated by the frozen backbone (not the adapters), and training memory is mostly activations/optimizer states—so higher rank contributes little to either accuracy or cost.

Takeaway: Pick rank for engineering stability or slight VRAM nuances; do not expect large gains from rank alone.

Q4. Cross-task transfer: which method is most robust and least prone to catastrophic forgetting?

Adapters are more stable; full FT swings larger (both gains and losses).

- *OpenMath* → *SQuAD*: NoPEFT drops **–26.5%** vs 0.6B base, while adapters sit **–2.4%** to **–6.7%**. *SQuAD* → *ARC*: NoPEFT falls **–7.9%**. *SQuAD* → *OpenMath*: NoPEFT improves error by **+7.4%**.
- **Interpretation.** Full FT magnifies specialization (bigger in-domain gains and bigger OOD shifts). Adapters constrain drift, mitigating forgetting but also limiting positive transfer. The pronounced latency differences across tasks—particularly the **–75.9%** reduction on OpenMath vs **–6.2% to –23.0%** on SQuAD when using adapters—may reflect transfer of generation style: OpenMath fine-tuning teaches concise numeric outputs, which carry over to ARC's short MCQ responses, whereas SQuAD-tuned models learn longer extractive spans that persist across evaluations.

Takeaway: For single-task SFT with multi-task use, adapters are safer; full FT is higher risk/higher reward.

Q5. Which configurations minimize inference costs?

0.6B + NoPEFT is generally cheapest to serve for a given quality.

- Versus adapters, NoPEFT reduces eval VRAM by **48–49%**. In *OpenMath*-trained models, NoPEFT also cuts latency by **96.9%** (task-internal) and by **93.1%** on ARC. In *SQuAD*-trained models, one adapter is faster on SQuAD by **21.6%** but with **–66%** F1 vs NoPEFT.

- Versus base models. Compared to the 0.6B base, NoPEFT lowers latency by **–94.8%** (ARC), **–99.3%** (OpenMath), and **–2.8% to –6.0%** (SQuAD), with **VRAM at parity** (eval-all). Versus the 1.7B base, NoPEFT reduces eval VRAM by **–61.2%** and latency by **–98.2%** (ARC), **–99.6%** (OpenMath), and **–19–22%** (SQuAD).
- **Interpretation.** The fully tuned 0.6B serves on the plain dense path with fewer parameters and activations, minimizing memory traffic and kernel count. Adapters introduce minor overhead when not fully fused and, in this setting, do not offset it with higher accuracy; the larger base inflates compute and activations, driving higher latency and memory regardless of optimizations.

Takeaway: For serving cost at target accuracy, favor full FT on the small base in this setting.

Q6. Under training VRAM constraints, which method offers the best accuracy per memory spent?

VeRA maximizes feasibility at tight memory; LoRA/DoRA trade modest accuracy gains for noticeably more VRAM; full FT wins if memory allows.

- *OpenMath* training: VeRA uses **–30–41%** less Train-All VRAM than LoRA/DoRA and **–37.2%** vs NoPEFT, but its error is **+51.0%** higher than NoPEFT and **+4.3–5.6%** higher than LoRA/DoRA. *SQuAD* training: VeRA uses **–25%** less Train-All than LoRA/DoRA and **–43%** vs NoPEFT, with F1 only **–1.3–4.6%** below LoRA/DoRA yet **–66.4%** vs NoPEFT.
- **Interpretation.** When memory is the bottleneck, VeRA's shared random bases minimize trainable state; small accuracy losses vs LoRA/DoRA may be acceptable to fit strict caps, but full FT still delivers the best accuracy per GPU if you can allocate the extra VRAM.

Takeaway: *Hard cap* ⇒ VeRA; *slightly relaxed* ⇒ LoRA/DoRA; *no cap* ⇒ NoPEFT.

Q7. If VRAM constraints force adapters, is fine-tuning still worth it overall?

It depends on the task: adapters may be worthwhile for numeracy/MCQ (speed gains with modest accuracy lift), but generally not for extractive QA (accuracy loss despite speed).

- **OpenMath** Versus the 0.6B base, adapter SFT reduces error by **–3.7% to –4.7%** and cuts latency by **–75.9% to –76.6%**. However, serving VRAM increases by **+90%** (Eval-All).
- **SQuAD** Versus the 0.6B base, adapter SFT *reduces* F1 by **–2.4% to –6.7%** while latency improves **–6.2% to –23.0%**; serving VRAM increases by **+95%** (Eval-All).
- **Is it worth the full process?**
  - *Math:* If you cannot do full FT and you need substantially faster inference with a measurable accuracy lift, the **small accuracy gain** paired with a **large latency drop** can justify the data curation, adapter training, and hosting—especially when user-facing throughput or cost per request dominates.
  - *SQuAD:* Given **accuracy declines** and only modest-to-moderate latency gains, the additional pipeline complexity (data prep, training, model ops) typically **does not** pay off; prefer the base (or full FT / larger base) for extractive QA quality.
- **Interpretation.** Adapters recover some numeracy and decision boundaries under tight memory (good for Math/MCQ), but they under-adapt span calibration and answer selection needed for extractive QA, so quality dips even if decoding is slightly faster.

Takeaway: Under adapter-only budgets, Math benefits justify the effort when latency is critical; for SQuAD-like QA, the end-to-end cost rarely pencils out without full FT or a larger base.

Q8. With abundant VRAM, does full fine-tuning dominate, or do adapters keep a speed/serve edge?

Full FT dominates on quality and typically on speed/memory; adapter "speed wins" come with large quality losses.

- In-domain *OpenMath*, NoPEFT cuts error by **−30.8–33.8%** vs adapters and latency by **−96.9%**. In-domain *SQuAD*, NoPEFT boosts F1 by **+184–197%** vs adapters; one adapter is faster by **−17.7–21.6%** latency but at **−66%** F1. Eval VRAM for NoPEFT is **−48–49%** vs adapters.
- **Interpretation.** Once memory is no longer binding, unconstrained updates reconfigure the full representation and serve efficiently (merged weights), while adapters retain structural limits and minor runtime overhead that do not translate into better end-to-end efficiency at competitive quality.

Takeaway: With ample VRAM, choose NoPEFT; adapter "speed edges" are not competitive at matched accuracy.

Q9. Which configurations are Pareto-optimal over efficacy, latency, and memory?

A configuration is **Pareto-optimal** if no other configuration improves at least one metric (efficacy ↑, latency ↓, memory ↓) without worsening another. **Three clear fronts emerge: (i) maximum quality, (ii) best small-model serve, and (iii) latency extreme for SQuAD.**

- **Maximum quality.** The 1.7B base is Pareto on efficacy across tasks (ARC F1 **+61.9%** vs 0.6B base; OpenMath error **−97.0%**; SQuAD F1 **+201.6%**), albeit with much higher latency and memory.
- **Best small-model serve.** For each training block, 0.6B NoPEFT is Pareto due to jointly low latency and eval VRAM with strong efficacy (e.g., *OpenMath* NoPEFT: **−96.9%** latency and **−48–49%** eval VRAM vs adapters; *SQuAD* NoPEFT: much higher F1 with minimal memory).
- **Latency extreme (SQuAD).** A SQuAD adapter achieves the lowest latency (**−17.7–21.6%** vs NoPEFT) but is not dominated only because of speed; it pays a **−66%** F1 penalty and higher eval VRAM, so it is a niche point on the front.

Takeaway: Deployment choice reduces to three Pareto archetypes: *bigger-is-better* (1.7B), *fast-cheap-strong* (0.6B NoPEFT), or *speed-at-any-cost* (SQuAD adapter).

6 Conclusions & Takeaways

We summarize two complementary views: (i) technical takeaways that guide modeling choices, and (ii) practical conclusions for solution design.

- **Start with small + full FT.** A well-tuned small model consistently delivered the best balance of quality, latency, and serving memory. As a default playbook, begin with the smallest viable base and do full SFT before considering complexity.
- **Adapters are a constraint tool, not a quality tool.** LoRA/DoRA/VeRA are excellent when training VRAM is the bottleneck or when you need modular hot-swappable domains. **They did not close the quality gap to full FT** or to a larger base in our setting.

- **Rank is a low-leverage knob.** Within a method family, changing rank rarely moved the overall efficacy–latency–memory trade-off. Prioritize stability, integration ease, and VRAM fit over chasing rank gains.
- **Task dictates adapter value.** Under constraints, adapters helped for numeracy/MCQ; they were far less compelling for extractive QA. Plan method choice by task type, not just by hardware limits.
- **When VRAM is abundant, go full FT.** Once training memory isn't binding, unconstrained updates win on quality and typically serve efficiently after merging; adapter "speed edges" didn't hold at comparable accuracy.
- **Mind forgetting vs transfer.** Full FT amplifies both upside (positive transfer) and downside (forgetting). If you need cross-task robustness, consider multi-task/rehearsal SFT, lightweight regularization, or per-task adapters with routing.
- **VeRA for hardest caps; LoRA/DoRA when feasible; full FT if possible.** As a capacity ladder: VeRA (fits strictest train VRAM), then LoRA/DoRA, then full FT. Move up only when the quality bar or product SLOs demand it.
- **Scaling decision rule.** If small+full FT fails your quality bar after reasonable data/decoding work, *then* consider scaling the base; otherwise you pay latency and memory costs without clear product upside.

Practical conclusions

Fine-tuning can unlock clear product wins—higher task accuracy, tighter formatting, and more predictable behavior— but it is also **an end-to-end investment** that requires data curation, training/serving engineering, and continuous evaluation. In our setting, **small + full fine-tuning offers the best overall balance** of quality, latency, and serving cost, and yields a single, simple artifact to deploy (often faster after weight merging). **Treat fine-tuning as a product decision anchored to SLOs (Service Level Objectives: accuracy, latency, cost)**, not as a default modeling reflex.

Choose methods by constraint and by task. Adapters (LoRA/DoRA/VeRA) are best used to satisfy constraints—tight training VRAM, hot-swappable domains, or multi-tenant personalization—**not to chase peak quality**. In our results they did not match full FT or a larger base. **Task type matters:** for numeracy/MCQ (e.g., structured scoring, eligibility checks, simple decisioning) adapters can be worthwhile under constraints; **for extractive QA and RAG pipelines (document question answering, policy lookup, support KB search), full FT—or a larger base when needed—more reliably reaches the accuracy bar.**

References

[1] Qingru Zhang, Minshuo Chen and Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen and Tuo Zha; "AdaLoRA: Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning", *arXiv:2303.10512*, 2023. Url: https://arxiv.org/abs/2303.10512

[2] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman and Luke Zettlemoyer; "QLoRA: Efficient Finetuning of Quantized LLMs", *arXiv:2305.14314*, 2023. Url: https://arxiv.org/abs/2305.14314

[3] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan and Sylvain Gelly; "Parameter-Efficient Transfer Learning for NLP", *arXiv:1902.00751*, 2019. Url: https://arxiv.org/abs/1902.00751

[4] Xiang Lisa Li and Percy Liang; "Prefix-Tuning: Optimizing Continuous Prompts for Generation", *arXiv:2101.00190*, 2021. Url: https://arxiv.org/abs/2101.00190

[5] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang and Weizhu Chen; "LoRA: Low-Rank Adaptation of Large Language Models", *arXiv:2106.09685*, 2021. Url: https://arxiv.org/abs/2106.09685

[6] Armen Aghajanyan, Luke Zettlemoyer and Sonal Gupta; "Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning', *arXiv:2012.13255*, 2020. Url: https://arxiv.org/abs/2012.13255

[7] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng and Min-Hung Chen; "DoRA: Weight-Decomposed Low-Rank Adaptation", *arXiv:2402.09353*, 2024. Url: https://arxiv.org/abs/2402.09353

[8] Dawid J. Kopiczko, Tijmen Blankevoort and Yuki M. Asano; "VeRA: Vector-based Random Matrix Adaptation", *arXiv:2310.11454*, 2024. Url: https://arxiv.org/abs/2310.11454

[9] Qwen3-0.6B. Available at *https://huggingface.co/Qwen/Qwen3-0.6B*. Accessed on 2 de octubre de 2025.

[10] Ai2_arc. Available at *https://huggingface.co/datasets/allenai/ai2_arc*. Accessed on 1 de octubre de 2025.

[11] Open Math Instruct-2. Available at *https://huggingface.co/datasets/nvidia/OpenMathInstruct-2*. Accessed on 1 de octubre de 2025.

[12] Squad_v2. Available at *https://huggingface.co/datasets/rajpurkar/squad_v2*. Accessed on 1 de octubre de 2025.

[13] Hugging Face Transformers. Available at *https://huggingface.co/docs/transformers/en/index*. Accessed on October 4, 2025.