ChatGPT

# Fine-Tuning Algorithms for Large Models: A Comprehensive Study

**Abstract:** Fine-tuning transforms a large pretrained model into a specialized one by additional training on task-specific data. Traditional *full fine-tuning* updates all model weights, offering flexibility at high cost, while *parameter-efficient fine-tuning* (PEFT) methods modify only a small subset or add lightweight modules. This report surveys and compares these approaches. We categorize fine-tuning into full tuning, parameter-efficient tuning (additive, selective, reparameterized, hybrid), and note alternatives. We explain key PEFT methods—**LoRA**, **QLoRA**, **Adapters**, **Prompt Tuning**, **Prefix Tuning**, and **BitFit**—covering their mechanisms, uses, trade-offs, and efficiency. We then examine how these techniques are applied to large language models (LLMs) for instruction-following, dialogue, summarization, and RLHF-based alignment, as well as to vision-language models (VLMs) for visual question-answering and captioning. We compare fine-tuning across LLM, masked-LM, and VLM architectures, noting similarities and special considerations. Finally, we discuss infrastructure implications: GPU/TPU requirements, memory, training time, and frameworks like HuggingFace Transformers/PEFT, DeepSpeed, etc., contrasting open-source and proprietary solutions. The report includes diagrams and tables for clarity, with academic citations throughout.

## 1. Introduction

Large **pre-trained models** have revolutionized AI by learning general representations from massive data. For example, transformer-based language models are first trained on vast text corpora and then *fine-tuned* on specific tasks [1]. Fine-tuning adapts a model to new data or objectives by further gradient-based training. This is a core technique in transfer learning: the model has learned broad knowledge from pretraining and is refined on task-specific data. Full fine-tuning means updating *all* weights, which can achieve top accuracy but demands enormous compute and memory [2]. An alternative is **parameter-efficient fine-tuning (PEFT)**, which freezes most of the model and trains only a small, task-specific set of parameters. PEFT dramatically cuts resource use and storage, enabling large models to be customized on modest hardware. The main categories of fine-tuning are: **full fine-tuning**, **additive PEFT** (introducing new modules like adapters or prompts), **selective PEFT** (training a subset of existing weights), and **reparameterized PEFT** (factorized updates like LoRA) [3]. Our goal is to survey these methods, explain representative techniques, and compare their use in modern AI systems.

## 2. Categories of Fine-Tuning

Fine-tuning algorithms can be broadly classified as follows:

- **Full Fine-Tuning:** Update **all** parameters of a pre-trained model to the new task. This is the traditional approach: it offers maximal flexibility and often the best performance. However, it is extremely resource-intensive. For example, full fine-tuning a 175B-parameter model like GPT-3 requires *thousands* of GPU-days of computation [4] and vast memory to store gradients. Such intensive training can also risk overfitting or forgetting older knowledge. As Lialin et al. note, “Standard full fine-tuning entails substantial computational expenses and could also potentially harm the model’s generalization ability” [2]. After training, a full copy of the model must be stored for each task, which raises storage and deployment costs.

- **Parameter-Efficient Fine-Tuning (PEFT):** Update only a **small fraction** of parameters or add lightweight modules while keeping the rest frozen [3]. The key idea is to **freeze** the large pretrained backbone and introduce minimal new parameters (sometimes called “adapters” or “prompts”) that are tuned for the task. PEFT methods fall into subcategories: *additive* (inject new modules or prompts), *selective* (train only some existing parameters), *reparameterized* (train low-rank factorizations of weight updates), or *hybrid* (combinations) [3] [5]. By updating far fewer weights, PEFT drastically cuts the memory and compute needed for fine-tuning, and only small “delta” parameters need storing per task. However, because the base is fixed, PEFT can sometimes lag full fine-tuning in flexibility, and careful tuning of the added modules is required.

- **Other Approaches:** Outside these, one may consider **feature extraction** (freeze the model completely and train only a new output head), or **prompt-based inference** (no gradient updates, just engineering input prompts), or **distillation** (train a smaller model to mimic the large one). These are not fine-tuning per se but are alternative adaptation strategies. We focus on gradient-based methods here.

The taxonomy of PEFT strategies is shown in Fig.1: additive methods insert new trainable blocks (adapters, prompts), selective methods pick existing weights (e.g. biases) to update, reparameterized methods (like LoRA) factorize updates, and hybrid methods combine techniques [3]. For example, an *additive* adapter approach freezes the backbone and adds small MLP layers in each transformer block; only those adapter weights are trained [5]. In *selective* tuning, one might fine-tune just a subset like the bias terms (BitFit) [6]. In *reparameterized* tuning (e.g. LoRA) the weight update is expressed as a product of two smaller matrices, which are trained instead of the full weight [7] [8]. Each approach has trade-offs in parameter count, compute, and expressivity.

## 3. Prominent PEFT Methods

Below we detail key PEFT algorithms.

### 3.1 LoRA and QLoRA (Low-Rank Adaptation)

**LoRA (Low-Rank Adaptation)** decomposes weight updates into low-rank matrices [7] [8]. Concretely, for a weight matrix (W \in \mathbb{R}^{d\times d}), LoRA represents the change as (BA), where (A\in\mathbb{R}^{d\times r}) and (B\in\mathbb{R}^{r\times d}) with (r \ll d). During fine-tuning, (W) remains frozen (blue in Fig.1) and only (A,B) (orange) are trained. After training, the effective weight is (W_{\text{merged}} = W + BA). This reparametrization cuts the number of trainable parameters from (d^2) to (2dr), a large saving for small (r) [7] [9].

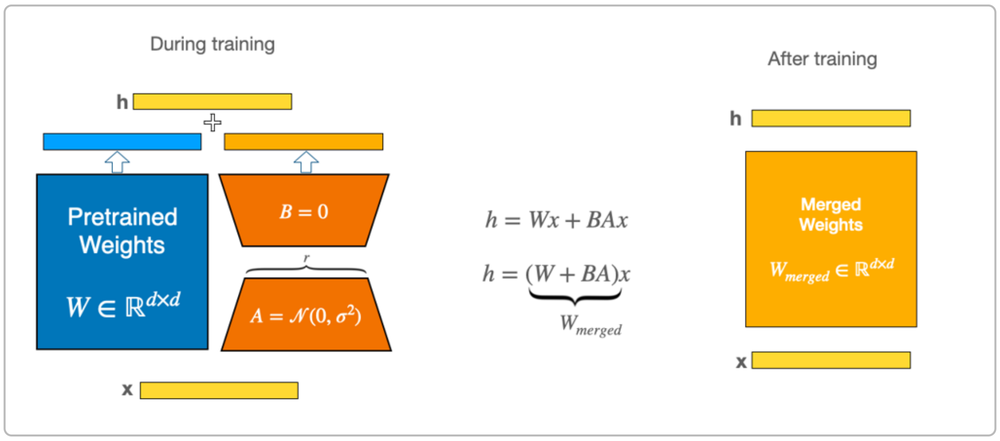
During training
After training
h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
$r$
$A = \mathcal{N}(0, \sigma^2)$
x
$h = Wx + BAx$
$h = \underbrace{(W + BA)}_{W_{merged}}x$
Merged Weights
$W_{merged} \in \mathbb{R}^{d \times d}$

**Figure 1:** *Low-Rank Adaptation (LoRA) concept.* The pretrained weight (W) (blue) is fixed; small matrices (A,B) (orange) are learned. During training (left), (B) is initialized zero and only (A,B) are updated. After training (right), the effective weight becomes (W_{\text{merged}} = W + BA). This adds negligible inference cost since (W+BA) can be merged back.

LoRA's advantages are striking: it requires only a tiny fraction of parameters (often <1–2%) [10] and greatly lowers GPU memory usage, enabling large-model tuning on limited hardware. For example, LoRA-adapted models can often be merged at inference with no additional latency [11]. Empirically, LoRA usually matches full fine-tuning performance for language tasks [12]. It is most effective in attention layers, especially query and value projections, and is orthogonal to other PEFTs (so one can combine LoRA with e.g. prompt tuning) [10].

A recent extension is **QLoRA (Quantized LoRA)** [13]. QLoRA adds aggressive quantization to LoRA: the backbone model is loaded in 4-bit precision (using NVIDIA's NF4 quant format) and LoRA modules are trained on top of it [14] [13]. This slashes memory needs by ~8× compared to 32-bit weights, without degrading performance. In practice, QLoRA enabled fine-tuning a 65B-parameter model on a *single* 48GB GPU in 24 hours, while preserving full 16-bit accuracy [14]. As Dettmers et al. report, QLoRA "democratizes fine-tuning of LLMs, enabling researchers with limited resources to work with large models without compromising quality" [13]. In short, LoRA provides a low-rank reparameterization for efficient tuning, and QLoRA further leverages quantization to handle ultra-large models on commodity hardware.

## 3.2 Adapter Layers

**Adapter-based fine-tuning** inserts tiny subnetworks into each transformer block. A typical adapter is a *bottleneck* MLP: it projects down from hidden size (d) to a smaller rank (r), applies a nonlinearity, then projects back to (d) [9]. Formally, adapter output is [ \text{Adapter}(x) = x + W_{\text{up}} \sigma(W_{\text{down}} x), ] where (W_{\text{down}}!\in!\mathbb{R}^{r\times d}) and (W_{\text{up}}!\in!\mathbb{R}^{d\times r}). The adapter's parameters ((W_{\text{down}},W_{\text{up}})) are trained, while the original layer is frozen. This "adapter" gives the model a few extra degrees of freedom to adapt to the task. Early works (e.g. Houlsby adapters) placed two adapters per transformer (one after attention, one after feed-forward) [15]. Later variants (e.g. AdapterFusion) simplify by inserting adapters only at one point [16].

**Advantages:** Adapters typically add just 2–5% extra parameters while still delivering performance close to full fine-tuning on many benchmarks. They are modular (one can add separate adapters per task) and can be stacked or merged. Unlike LoRA, adapters are sequential layers that do impose a small compute overhead per layer.

**Disadvantages:** Each adapter increases inference computation slightly, and very deep models accumulate more overhead. Some adapters place layers in series which can reduce parallelism [17]. In some cases, adapter tuning lags full fine-tuning on low-data tasks, though it often still outperforms simpler selective methods. Overall, adapters strike a balance: they use few parameters and keep base weights frozen, at the cost of extra multiplications during forward passes.

## 3.3 Prompt Tuning

**Prompt Tuning** is a soft-prompt approach originally designed for frozen language models [18]. Instead of updating model weights, it **prepends** a sequence of trainable "virtual tokens" to the input. Concretely, one allocates (m) new embedding vectors (the "prompt") and only trains these (m\cdot d) parameters while the entire model is frozen. During a forward pass, these learned embeddings are placed before the real token embeddings. The model then processes the concatenated sequence normally. Because only the prompt embeddings are optimized, storage per task is extremely small – often just a few kilobytes. The Hugging Face PEFT guide notes that "prompt tuning can be more efficient, and results are even better as model parameters scale" [18].

**Mechanism:** The prompt vectors are either randomly initialized or taken from existing token embeddings, and then tuned via gradient descent on the task loss. Importantly, the prompt parameters are *the only* trainable components. At inference, one simply reuses the learned prompt for all inputs.

**Use Cases:** Prompt tuning is especially effective for **very large** language models (e.g. above 10B parameters). When large models are frozen, prompt tuning can steer them for tasks like classification or generation. It is widely used for tasks such as sentiment classification, topic classification, and simple text generation, where task instructions can be embedded in those virtual tokens.

**Trade-offs:** Prompt tuning has extremely low parameter cost, but can underperform when models are smaller or tasks are very different. It also requires a suitable number of prompt tokens (hundreds in some cases) to be effective. Nonetheless, in the *soft prompt* paradigm, this method avoids modifying any internal weights, making it very safe memory-wise. In summary, prompt tuning updates only task-specific embedding tokens while preserving the entire pretrained model [18].

## 3.4 Prefix Tuning

**Prefix Tuning** is similar to prompt tuning but extends throughout the model layers [19]. Instead of only adding trainable embeddings at the input, prefix tuning introduces learned vectors that serve as *virtual key/value pairs* in each transformer layer. Concretely, each transformer block receives additional key and value vectors (the "prefix") that every token can attend to. The prefix vectors are continuous parameters that are trained, while the main model remains frozen.

For example, Li et al. describe prefix tuning as "prepend[ing] a series of task-specific vectors to the input sequence" across all layers [19]. In practice, one might learn a small vector for each of the (m) prefix tokens at each layer. The original prefix-tuning paper showed that learning only ~0.1% of parameters (the prefix) can match full fine-tuning on tasks like summarization, and even outperform full tuning in few-shot settings [20].

**Advantages:** Prefix tuning is very parameter-efficient (since the prefix length and hidden size are small) yet often delivers strong generative performance. It is a *prompt-inspired* method tailored to generation tasks, because the prefix affects how attention is paid at every layer. The Hugging Face prefix-tuning guide summarizes: it "keeps language model parameters frozen, but optimizes a small continuous task-specific vector… [achieving] comparable performance… by learning only 0.1% of the parameters" [20].

**Disadvantages:** Computing with a prefix is slightly more expensive than a simple prompt, because each layer adds extra keys/values. This can moderately slow down inference. Also, prefix tuning is mainly useful for autoregressive/generative models; it's less applicable to encoder-only masked models. However, for text generation tasks like machine translation, summarization, and dialogue, prefix tuning is a powerful lightweight alternative to full fine-tuning.

### 3.5 BitFit

**BitFit** is the most extreme minimalist approach [21]. It fixes all model weights except the bias terms (and optionally the classification head). In practice, BitFit means freezing every parameter except each layer's bias vector, and only updating those biases during fine-tuning. This amounts to changing roughly 0.01–0.1% of the parameters.

Zaken et al. showed that BitFit on BERT-like masked LMs can achieve near the same accuracy as full fine-tuning in many NLP tasks [21]. The upside is that BitFit uses almost no extra memory and trains extremely fast. However, its power is limited: it generally works well only for smaller models or large datasets. For massive models, BitFit often underperforms more expressive PEFTs, as the survey notes: "BitFit achieves competitive results for small models. However, this method fails to handle large models" [6]. In short, BitFit is useful as a very lightweight baseline or for quick experiments, but usually one needs richer adaptation (LoRA, adapters) for cutting-edge LLMs.

## 4. Fine-Tuning LLMs: Tasks and Techniques

Large Language Models (LLMs) power modern AI assistants and chatbots. Both full and PEFT methods are used to adapt LLMs to specific tasks. We highlight key applications:

- **Instruction Tuning & Dialogue:** To make LLMs follow instructions, one fine-tunes them on instruction–response pairs. OpenAI's *InstructGPT* is a prime example: GPT-3 was fine-tuned using human-written demonstrations and preference rankings. In practice, labelers provided example outputs and ranked model outputs, and these were used to fine-tune GPT-3 [22]. The result was a 1.3B InstructGPT model that people preferred over a 175B unfine-tuned GPT-3 [22]. Likewise, models like ChatGPT were trained by first supervised fine-tuning on human dialog data and then applying RLHF (discussed below). On the open-source side, Stanford's Alpaca (2023) fine-tuned LLaMA-7B on 52K GPT-3.5-generated instruction pairs [23], resulting in a small model exhibiting GPT-3-like behavior. Many such projects use LoRA for efficiency: e.g. Vicuna and Guanaco are instruction-following LLaMA derivatives created via LoRA/QLoRA tuning on instruction datasets.

- **Summarization & Generation:** LLMs are often fine-tuned for summarization, story generation, and related tasks. For instance, sequence-to-sequence models like BART or T5 are fully fine-tuned on news summarization corpora. Autoregressive LLMs (e.g. GPT) can similarly be tuned by prefixing input prompts with "Summarize:" and training on summary data. Parameter-efficient methods work here too: a LoRA-tuned GPT can become a strong abstractive summarizer. Prefix-tuning, with its generation-oriented design, is also well-suited for such tasks. In general, any textual generation task (translation, code generation, etc.) can use these fine-tuning methods to specialize an LLM.

- **Reinforcement Learning from Human Feedback (RLHF):** Modern assistants use RLHF to align outputs with human preferences. The RLHF **pipeline** has three steps [24]: (1) Pretrain a language model on general text. (2) Collect human feedback: humans rank outputs or provide demonstrations for sample prompts. (3) Train a **reward model** on that feedback, then *fine-tune* the LM via reinforcement learning (usually Proximal Policy Optimization) to maximize the learned reward. Hugging Face summarizes RLHF stages succinctly [24]. OpenAI's InstructGPT used this: after gathering ranked responses to prompts, GPT-3 was fine-tuned with RL to produce safer, more helpful outputs [22]. ChatGPT similarly underwent RLHF to refine GPT-3.5. Notably, RLHF itself involves full model updates (the policy LM is updated), but PEFT approaches like LoRA are being explored in RLHF settings to reduce compute. In practice, RLHF typically yields substantial gains in helpfulness and safety, at the cost of extra complexity.

- **Use of PEFT in LLM Fine-Tuning:** Many recent LLM fine-tuning projects leverage PEFT for practicality. For example, the Guanaco model (33B parameters) was fine-tuned from Meta's LLaMA using QLoRA. Remarkably, Guanaco matched about 99.3% of ChatGPT's performance on benchmarks [25] while being trained in 24 hours on a single 48GB GPU. This was only feasible due to the memory reduction of QLoRA [25]. Similarly, the Hugging Face 'PEFT' tutorial shows fine-tuning a conversational summarizer model using LoRA and QLoRA to save memory and time. In summary, both full and PEFT methods are widely used in LLM fine-tuning, with PEFT enabling open groups to customize massive LMs that would otherwise be unreachable.

## 5. Fine-Tuning Vision-Language Models (VLMs)

Vision-language models bridge image and text. They typically consist of a vision encoder (CNN or ViT) and a language model, often connected via cross-attention. Key tasks include **visual question answering (VQA)**, **image captioning**, **image–text retrieval**, and multimodal dialogue. In these tasks, the model must jointly reason about images and text. Representative models include CLIP, ViLT, Flamingo, BLIP, and multimodal versions of LLMs (e.g. LLaVA, PaLI-Gemini).

### Common Tasks and Models

*Visual Question Answering (VQA):* Given an image and a question, the model must answer (often in natural language or classification). For example, CLIP can be adapted to VQA by using its text encoder to encode possible answers. *Image Captioning:* The model generates a descriptive caption for an image. Models like BLIP and Flamingo excel here. *Others:* Document understanding and OCR mix visual layout and language (e.g. text in images).

Flamingo (DeepMind, 2022) is a landmark VLM: it adds cross-attention layers between frozen vision and language backbones, enabling few-shot image+text tasks. Flamingo's developers report that a single model achieves new SOTA on captioning and VQA by *prompting* it with examples [26] [27]. As the authors state, Flamingo "bridges powerful pretrained vision-only and language-only models" to handle arbitrarily interleaved images and text [26]. In practical terms, a model like Flamingo can answer visual questions and describe scenes with high quality after few examples.

## PEFT Methods for VLMs

Many PEFT ideas carry over to VLM fine-tuning, but there are special considerations:

- **Adapter Tuning:** One can insert adapters in both the vision and language branches or at the fusion layers. For example, *CLIP-Adapter* inserts small MLPs on top of CLIP's image and text features [28]. Adapters typically yield good performance with few extra params. Recent work even designs **multi-modal adapters**: Jiang et al. (2024) propose a Multi-Modal Adapter that jointly attends to image and text features by a trainable attention layer [29] [30]. This adapter improves generalization on unseen classes by capturing cross-modal interactions. The figure from that paper illustrates adding an attention-based adapter between CLIP's image and text embeddings [30].

- **Prompt and Prefix Tuning:** Prompting extends to vision. One can learn *text prompts* that condition on the image (as in CoOp/CoCoOp for CLIP) [28]. Or learn *visual prompts*: e.g. a set of virtual image tokens or patches added to the image input (analogous to prefix tokens). Li et al. find that prefix tuning improves VLM tasks: prefix representations can capture visual details better than full fine-tuning alone [31]. In fact, "applying prefix-tuning before LoRA/Adapter… consistently improves task performance for captioning and VQA" [31]. Thus, one strategy is sequential: first learn prefix vectors, then apply another fine-tuning step.

- **LoRA/QLoRA:** Similar to LLMs, LoRA can be applied in VLMs on the language part or cross-attention. For example, the LLaVA project (Visual Instruction Tuning) fine-tuned a vision-enhanced LLaMA on VQA using LoRA on the LLM part, keeping the image encoder frozen. More recently, QLoRA has been used on vision models. A PyImageSearch tutorial (2024) demonstrates fine-tuning Google's PaLI-Gemini VLM on VQA using QLoRA, highlighting that even large multimodal models can be tuned on a single GPU [32].

- **Selective Methods:** BitFit or LayerNorm tuning have been less explored in VLMs; most VLM fine-tuning tunes some part of the model's new layers. However, one could freeze the base networks and only tune biases or LayerNorm of the fusion layers as a lightweight experiment.

In summary, VLM fine-tuning often combines ideas from both vision and language PEFTs. As in NLP, additive adapters or prompt vectors can adapt the model, but they must account for visual inputs. The table below compares common strategies:

| Method | Application in VLMs | Notes |
|---|---|---|
| **Adapter** | Insert MLP adapters into vision and/or text branches [28]. For example, adapters on CLIP's output features. | Flexible; small overhead; can adapt each modality separately. |

| Method | Application in VLMs | Notes |
|---|---|---|
| **LoRA / QLoRA** | Apply low-rank updates to LLM and/or cross-attn layers. Used in LLaVA, BLIP-2, etc. | Large memory savings allow tuning big VLMs on limited GPU. |
| **Prompt Tuning** | Learn text prompts for vision-conditioned tasks (e.g., CoOp prompts for CLIP) [28] . | Effective for classification; simple and low-memory. |
| **Prefix Tuning** | Learn prefix vectors for multimodal transformer layers (image+text) [19] [31] . | Adds global context; combined with other PEFT often. |
| **Visual Prompting** | (Not standard term here) Learn small patch tokens or use tokenized image prompts. | Emerging; e.g. fine-tune few image patches with adapter layers. |
| **BitFit** | Rarely used; could freeze all but biases in fusion layers. | Minimal memory; likely insufficient for complex VLM tasks. |

Vision models treat an image as a sequence of patches (analogous to tokens in LLMs) [33] , so many of the same techniques apply. For instance, just as prefix tokens guide a text model, one could prepend learned "image tokens" to the patch sequence. Empirical studies find that sequential tuning (e.g. prefix then LoRA) works best to avoid collapsing representations [31] . In practice, fine-tuning VLMs often means freezing the heavy vision encoder (to save compute) and tuning the lighter language/cross modules. Nevertheless, the relative computational cost is higher due to processing images, and pretrained vision layers may require careful handling. Overall, the goal remains to achieve strong cross-modal understanding with minimal added parameters.

## 6. Comparative Analysis Across Architectures

Different model families present different fine-tuning characteristics:

- **Autoregressive LLMs (e.g. GPT, LLaMA):** Pretrained by next-token prediction, these models generate fluent text. Fine-tuning tasks are usually language generation or instruction following. Techniques like prefix/prompt tuning are natural here. LoRA and adapters are commonly applied to attention and feed-forward layers. RLHF is specific to these models for alignment.

- **Masked LMs (e.g. BERT, RoBERTa, DeBERTa):** Pretrained by predicting masked tokens, these models excel at understanding tasks (classification, QA). Fine-tuning often involves adding a task-specific head on top. Adapters and LoRA also apply: indeed, adapters were first popularized on BERT. BitFit was originally demonstrated on BERT as well [21] . Prefix tuning is less relevant (BERT is bidirectional and not typically used for autoregressive generation). Prompt tuning can work by prepending tokens, but usually fewer prefixes are needed since classification often relies on [CLS] token. Overall, MLM fine-tuning tends to require less context handling but still benefits from parameter-efficient methods when large.

- **Vision Models (e.g. Vision Transformers):** Pretrained on image data, fine-tuning tasks include classification, detection, segmentation. Many PEFT ideas transfer (e.g. LoRA can adapt the selfattention in a ViT, adapters can be inserted into convolutional backbones, etc.). Mixed-modal models (CLIP, BLIP) combine vision and text. CLIP, for example, is trained with contrastive loss; fine-tuning it often means adding prompts or small modules to the image encoder or the text encoder. Unlike LMs, image models may rely more on static feature adaptation (e.g. fine-tuning a linear classifier on frozen CNN features is common). However, modern VLMs (like Flamingo) treat images as token sequences [33], making them very similar architecturally to LLMs. In fact, *Vision-Language Alignment Models* like CLIP learn a joint embedding space for images and text [34]. This means that in many ways, tuning a VLM is akin to tuning an LLM with extra modality. For example, MaPLe (2023) shows that one can prompt both the image branch and text branch simultaneously for CLIP to achieve strong performance [34].

**Similarities and Differences:** Across all architectures, full fine-tuning updates all parameters (with heavy cost), whereas PEFT methods freeze most weights. Both LLMs and MLMs can use adapters and LoRA; the main difference is the nature of tasks (generation vs classification). Prefix/prompt tuning shines in autoregressive models but is uncommon in purely masked models. VLMs incorporate an image encoder, so fine-tuning often focuses on the language and cross-attention parts. In all cases, catastrophic forgetting is a concern with small data, making PEFT attractive as it preserves pretrained knowledge better [5]. Table 1 summarizes key contrasts:

| Aspect | Autoregressive LLM | Masked LM | Vision-Language Model |
|---|---|---|---|
| **Pretraining** | Next-token prediction (autoregr.) | Masked token prediction | (Image+text) contrastive or captioning |
| **Common Tasks** | Text generation, dialogue, summarization | Classification, NER, QA | VQA, image captioning, multimodal QA |
| **Fine-tune Head** | May add LM head or classifier | Classifier on [CLS] | Usually generative head or classifier |
| **PEFT Usage** | LoRA/adapters on all layers; prefix/prompt tuning; RLHF | Adapters/LoRA on transformer; (prompt-tuning possible with prefixes for tasks) | LoRA/adapters on text+fusion; visual prompts/prefixes |
| **Inference** | Sequence generation with context | Feature classification | Multi-step: encode image + decode text |
| **Hardware** | Often requires multi-GPU for large context | Less context, often smaller models | Large models with vision encoder, costly |

In summary, while the broad techniques overlap, one must tailor fine-tuning to the model type and task. For example, BitFit succeeded on BERT [21] but is rarely tried on chat models. Conversely, prefix tuning (originating in GPT) is not used for masked models. Vision models introduce an additional modality, but many solutions (like LoRA) carry over since ViTs process patches like tokens [33]. The underlying theme is consistent: train *just enough* parameters to adapt to the new task.

## 7. Infrastructure Implications

Fine-tuning large models has significant infrastructure considerations:

- **Compute and Memory:** Full fine-tuning of very large models (tens of billions of parameters) normally requires multiple high-memory GPUs or TPUs. For example, training GPT-3 (175B) reportedly involved thousands of V100 GPUs over weeks (costing millions) [4]. In contrast, PEFT drastically reduces memory. LoRA and adapters only store gradients for the new modules, cutting memory by ~×2–4. QLoRA's 4-bit quantization slashes model storage to ~1/4, allowing a 65B model to fit on a single 48GB GPU [14]. As noted, QLoRA allows one-GPU fine-tuning of a 65B model in 24 hours [14]. DeepSpeed's ZeRO optimizer can further shard states across GPUs [35], meaning a 70B model can be fine-tuned on, say, 8×H100 machines with moderate memory each [35] [36]. These innovations mean that massive models are no longer accessible only to large labs. However, resource needs remain nontrivial: even LoRA tuning of a 30B model may require multiple 40GB GPUs to hold activations. In practice, high-end hardware (A100/H100 GPUs with 40–80GB, or TPU v4 pods) is commonly used for SOTA fine-tuning.

- **Training Time:** PEFT methods often converge faster in wall-clock time because there are fewer parameters to update and less data-parallel communication. For example, training using LoRA typically runs a similar number of epochs as full tuning but with smaller backward passes. QLoRA training can be slower per step due to quantized operations, but still far outpaces distributed full-tuning. Nevertheless, training up to GPT-3 scale still takes hours to days even with PEFT. Practical fine-tuning of a 7B model with LoRA might take a few hours on 4 GPUs, whereas a 70B LoRA tune could take a day or two on 8 GPUs. DeepSpeed and gradient checkpointing are often used to accelerate and fit these runs.

- **Scalability:** Infrastructure frameworks now support scaling PEFT. Hugging Face's Accelerate library integrates with DeepSpeed and FSDP (Fully Sharded Data Parallel) to enable both PEFT and full fine-tuning at scale [35]. For instance, Hugging Face provides example scripts to fine-tune LLaMA-70B with LoRA on 8×H100 GPUs using ZeRO-3 [36]. Similarly, PyTorch/FSDP and TensorFlow MirroredStrategy can handle large models. Data parallelism (via Distributed Data Parallel) and model parallelism (Megatron-style) are often combined for >100B parameters.

- **Software Tools:** The ecosystem offers many tools for fine-tuning:

- **Hugging Face Transformers**: A widely used framework. Its `Trainer` and `AutoModel` APIs support PEFT. The **PEFT library** by HF provides convenient modules for LoRA, adapters, etc. Hugging Face blogs and docs (e.g. [50]) elaborate on PEFT integration.
- **BitsAndBytes**: NVIDIA's library for 8-bit and 4-bit optimizers (enabling QLoRA) is often used to load large models in low precision.
- **DeepSpeed**: Enables ZeRO optimization and offloading; it also supports ZeRO + LoRA + quantization combined (HF docs detail compatibility) [35] [37].
- **Accelerate**: HF's library for easy distributed training across GPUs/TPUs.
- **Frameworks**: PyTorch is most common; JAX/Flax is used e.g. by Google. Tools like FairSeq or Megatron-LM (NVIDIA) are used internally for massive training but less so for smaller fine-tuning projects.

- **Survey Platforms**: ML frameworks now often include built-in fine-tune support (e.g., TensorFlow's Keras, NVIDIA NeMo Megatron Trainer).

- **Open vs Proprietary:** In open-source, researchers use the tools above on cloud or local clusters. Commercial offerings exist: OpenAI provides fine-tuning APIs for GPT-3.5 (though limited in model variety); Google Cloud Vertex AI supports fine-tuning PaLM; Azure and AWS have fine-tune endpoints for various models. These managed services abstract away infrastructure, but still use many of the same techniques under the hood. For example, OpenAI's fine-tune API likely uses full fine-tuning for smaller GPT variants (up to a few billion parameters). In contrast, open groups often rely on PEFT to tune cutting-edge LLMs (like LLaMA/GPT-J) that are not available through APIs.

In practice, choosing a fine-tuning method involves trade-offs: full tuning may need a cluster of GPUs and days of training, whereas LoRA or QLoRA can run on a handful of GPUs in hours. Table 2 below compares resource footprints qualitatively:

| Method | Trainable Params | GPU Memory Needed | Inference Overhead | Use Case |
|---|---|---|---|---|
| Full Fine-Tune | 100% of model | Very High (gradients for all) | None (base speed) | Highest accuracy; small models only |
| LoRA | ~0.1–5% of model | Moderate (only adapters) | None after merging weights [11] | General adaptation for any model |
| QLoRA | ~0.1% (LoRA + 4-bit) | Very Low (quantized 4-bit) | None after merging | Huge models on limited GPUs |
| Adapter | ~1–5% | Moderate (adapter states) | Small extra MLP compute | Domain/task adaptation |
| Prompt Tuning | ~0.01–0.1% (prompt tokens) | Very Low (only prompt vectors) | Slight (prepend tokens) | Classification or generation with huge LMs |
| Prefix Tuning | ~0.1% (prefix vectors) | Low | Moderate (extra attention) | Generation tasks (NLG) |
| BitFit | ~0.01% (biases only) | Very Low | None | Quick prototyping, simple tasks |

*Table 2: Comparison of fine-tuning methods in terms of resource footprint and typical use cases.*

In summary, PEFT methods offer enormous infrastructure benefits: lower GPU count, lower memory, and faster iteration. Combined with modern frameworks (Hugging Face, DeepSpeed, BitsAndBytes), they make large-model fine-tuning practical for many organizations. However, even PEFT requires careful engineering: quantization may need calibration, ZeRO setups must be configured, and larger models still require high-end GPUs. The landscape continues to evolve, with better 8-bit optimizers, CUDA kernels (FlashAttention), and distributed strategies pushing the feasibility frontier further.

## 8. Conclusions

Fine-tuning large models is a key technique in AI, enabling general-purpose foundations to excel on specific tasks. We have surveyed the spectrum of fine-tuning algorithms: from **full** updates of all weights to **parameter-efficient** methods like **LoRA**, **Adapters**, **Prompt/Prefix Tuning**, and **BitFit**. Each approach offers a different trade-off between **parameter count**, **compute cost**, and **performance**. LoRA-based methods (including quantized variants) have emerged as a practical standard for adapting huge LLMs with modest resources [11] [13]. Adapters and prompts remain popular for both NLP and vision tasks, while BitFit serves niche cases.

In application, both LLMs and vision-language models use these techniques. Instruction tuning and RLHF fine-tune LLMs into dialogue agents [22] [24], often via LoRA to stay efficient. VLMs leverage similar ideas to learn from images and text [29] [31]. We find that, although architectures differ, the underlying themes are consistent: large frozen backbones plus small trainable components.

From an infrastructure viewpoint, PEFT methods substantially reduce the GPU, memory, and time requirements of fine-tuning, making it accessible beyond a few well-funded labs [14] [35]. Tools like Hugging Face Transformers and PEFT library, together with DeepSpeed and quantized optimizers, have democratized large-model fine-tuning.

As models grow, these efficiencies will only become more critical. Future work will likely further hybridize methods (e.g. combining prefixes with adapters [31]) and integrate fine-tuning more seamlessly into AI platforms. For students and practitioners, understanding the trade-offs of each algorithm is essential. In conclusion, the choice of fine-tuning method should balance available resources, task requirements, and desired performance, with parameter-efficient strategies offering a compelling path for modern AI infrastructure.

**References:** The statements above are supported by recent research and technical sources, including Hugging Face's documentation [7] [38], OpenAI publications [22], and contemporary surveys [3] [31].

---

[1] [21] arxiv.org
https://arxiv.org/pdf/2106.10199

[2] [3] [4] [5] [6] [9] [15] [16] [17] [19] [33] [34] arxiv.org
https://arxiv.org/pdf/2403.14608

[7] [10] [11] [12] LoRA
https://huggingface.co/docs/peft/main/en/conceptual_guides/lora

[8] [13] [37] [38] PEFT: Parameter-Efficient Fine-Tuning Methods for LLMs
https://huggingface.co/blog/samuellimabraz/peft-methods

[14] [2305.14314] QLoRA: Efficient Finetuning of Quantized LLMs
https://arxiv.org/abs/2305.14314

[18] Prompt tuning for causal language modeling
https://huggingface.co/docs/peft/main/en/task_guides/clm-prompt-tuning

[20] Prefix tuning
https://huggingface.co/docs/peft/en/package_reference/prefix_tuning

[22] Aligning language models to follow instructions | OpenAI
https://openai.com/index/instruction-following/

[23] Stanford CRFM
https://crfm.stanford.edu/2023/03/13/alpaca.html

[24] Illustrating Reinforcement Learning from Human Feedback (RLHF)
https://huggingface.co/blog/rlhf

[25] GitHub - artidoro/qlora: QLoRA: Efficient Finetuning of Quantized LLMs
https://github.com/artidoro/qlora

[26] [27] [2204.14198] Flamingo: a Visual Language Model for Few-Shot Learning
https://arxiv.org/abs/2204.14198

[28] [29] [30] Multi-Modal Adapter for Vision-Language Models
https://arxiv.org/html/2409.02958v1

[31] aclanthology.org
https://aclanthology.org/2024.findings-emnlp.44.pdf

[32] Fine Tune PaliGemma with QLoRA for Visual Question Answering - PyImageSearch
https://pyimagesearch.com/2024/12/02/fine-tune-paligemma-with-qlora-for-visual-question-answering/

[35] [36] DeepSpeed
https://huggingface.co/docs/peft/en/accelerate/deepspeed