ChatGPT

Partial Fine-Tuning Methods in Transformer Models

Modern large transformers (GPT, BERT, etc.) are often *pre-trained* on massive data and then *fine-tuned* for specific tasks. In *parameter-efficient* fine-tuning (PEFT), most of the pre-trained weights are **frozen**, and only a small subset of additional or existing parameters are updated for the new task [1] [2]. This drastically reduces training cost and model size while retaining accuracy. Below we examine five popular PEFT methods – **Adapters**, **Prompt Tuning**, **Prefix Tuning**, **Low-Rank Adaptation (LoRA)**, and **BitFit** – explaining exactly where each method intervenes in the model, how data flows during fine-tuning, and how inference works afterwards.

Adapters

Adapter modules are small neural networks inserted inside each transformer layer [3] [4]. In a standard transformer block, the input first goes through a multi-head attention sublayer, then a feed-forward (FFN) sublayer (each followed by layer normalization and a residual skip). **With adapters**, we *freeze* the original attention and FFN weights, and after each sublayer we add a tiny "adapter" MLP with a bottleneck (project down to small dimension and back up) and a residual skip [5]. During fine-tuning, only these adapter layers (and typically the layer-norm parameters and final classifier head) receive gradient updates [2]; the rest of the model remains unchanged. For each input sequence, the forward pass through a transformer layer becomes:

- **(1)** Input hidden states → [original Attention → normalization] → output_1
- **(2) Adapter after attention:** output_1 passes through a small bottleneck MLP (the adapter), then is added back to output_1 (skip-connection).
- **(3)** Combined output → [original FFN → normalization] → output_2
- **(4) Adapter after FFN:** output_2 passes through another adapter MLP, then is added back (skip-connection).
- **(5)** The result goes to the next layer or the final head.

Each adapter projects the layer's features to a lower dimension m and back to the original dimension, so that only a small number of new parameters are added [5]. During training, gradients flow only into the adapter weights (and any trainable norm or head), adjusting them based on the fine-tuning dataset. Inference then uses the frozen base model plus the now-trained adapter modules: each layer's output is slightly shifted by the adapter's learned correction. This lets one base model serve many tasks with different small adapter sets.

Prompt Tuning

In prompt tuning, the model's weights are frozen and we prepend a few trainable embeddings (a "soft prompt") to the input. This soft prompt is learned on the downstream data while the base model remains unchanged [6].

Prompt tuning leaves the model architecture intact but expands the input. Concretely, we create a short trainable tensor of embeddings (the **soft prompt**), e.g. 10–100 vectors, and prepend it to every input sequence. The **forward pass** is then just like normal: tokens (including the prompt vectors) go through the embedding layer, attention layers, etc. The difference is that during training only the *prompt embedding* vectors receive updates (via gradient descent) while all original model parameters stay frozen [6]. In effect, the soft prompt steers the frozen model by providing task-specific cues. For example, if the task is sentiment classification, the prompt embeddings will adjust so that the base model's next-word predictions align with positive/negative examples.

Training steps for prompt tuning can be summarized as:
- **Forward:** For each input sample, construct an extended input

$$soft prompt tokens; original tokens$$

, then compute the model's output probabilities as usual.
- **Compute Loss:** Compare output to task labels (e.g. classification loss).
- **Backward:** Backpropagate gradients; **only the prompt token embeddings** are marked as trainable, so updates modify them.
- **Repeat:** Over many examples, the prompt embeddings learn to encode the task.

Because no model weights change, inference is simple: each input is prepended with the *trained* prompt embeddings, and the base model (e.g. GPT or BERT) generates the output. Note that prompt tuning is most effective in very large models (billions of parameters) [7]; small models often underperform because the fixed weights have limited adaptability.

Prefix Tuning

In prefix tuning, we again freeze the base model but insert trainable vectors at every layer's attention mechanism. Specifically, a short sequence of "prefix" key/value pairs is prepended in each transformer layer's multi-head attention [8] [9].

Unlike prompt tuning, prefix tuning modifies the *hidden states* throughout the model. Each transformer layer's self-attention normally computes queries Q, keys K, and values V from the layer's input. Prefix tuning augments this by concatenating a small set of learnable vectors $\{P_K, P_V\}$ before the actual K and V. Concretely:

- **(1)** For layer ℓ, take its input hidden state H_ℓ. Compute Q = W_q H_ℓ, K = W_k H_ℓ, V = W_v H_ℓ (as usual).
- **(2) Insert prefixes:** Let $P_{K,\ell}$ and $P_{V,\ell}$ be trainable matrices (the *prefix*) for layer ℓ. We form augmented keys $K' = [P_{K,\ell}; K]$ and values $V' = [P_{V,\ell}; V]$. Attention then uses Q against K' and V'.
- **(3)** The result of attention is combined with a residual skip, as normal.

Each layer uses its own prefix vectors (usually generated by a small feed-forward network for stability) [9]. During fine-tuning, gradients flow only into these prefix vectors (and the prefix-generating FFN, if used); the original transformer weights W_q, W_k, W_v, W_o and FFN remain frozen [8]. Thus the model learns how to adjust its attention at each layer via these extra tokens. Inference simply uses the learned prefix for each layer: when processing a new input, we prepend the trained prefix keys/values in every layer's attention calculation. This allows one base model to be shared across tasks, with only the small prefixes differing. Prefix tuning can achieve performance comparable to full fine-tuning on large models, despite using only ~0.1% of parameters [10].

Low-Rank Adaptation (LoRA)

LoRA freezes the original weight matrices and injects low-rank update matrices into them [11]*. In practice, each large weight W in an attention or FFN layer is replaced by $W + BA$, where A and B are much smaller trainable matrices (of rank r)* [11] [12].

In a regular linear or attention layer with weight W, the output is Wx. LoRA changes this to $W_0x + (B\,A)x$, where W_0 is the frozen original weight and B, A are low-rank (e.g. W is $m \times n$, A is $r \times n$, B is $m \times r$). Concretely:

- **Forward pass:** For input vector x, compute W_0x as usual (using frozen weights). In parallel, compute Ax and then $B\,(Ax)$. Sum them to get the final output. (This is mathematically equivalent to $(W_0 + BA)x$ but done in two parts.)
- **Backward pass (fine-tuning):** Only A and B are trainable; we backpropagate gradients into these matrices. The frozen W_0 is never updated [12]. Typically A is initialized randomly and B to zero (so BA starts at zero effect). As training proceeds, A and B learn to encode the necessary weight updates in low-rank form.

This works because large models often have "intrinsic low-dimensional structure" in their weight updates [13]. The final learned weight update $\Delta W = BA$ can later be **merged** back into W_0 to produce an updated weight matrix for inference [12]. In other words, after fine-tuning one can permanently replace W_0 by $W_0 + BA$ so the model has no extra overhead at runtime. LoRA is usually applied to select layers (e.g. the Query/Value projections in each attention block) to save even more parameters [14].

During inference, one simply uses the base model with its augmented weights $W_0 + BA$. No additional parameters or latency are required once merging is done [15] [12]. In summary, LoRA's flow is: *input → (frozen weight output + learned low-rank update output) → sum → next layer*, with only the low-rank matrices learning from data [11] [16].

BitFit

BitFit is the simplest partial tuning: **freeze all weights except the bias terms** [17]. In a transformer or any network layer with a weight W and bias b, the forward pass is $y = Wx + b$. BitFit keeps W fixed and trains only b. Concretely:

- **Training:** For each layer (attention, FFN, etc.), mark only the bias parameters b as trainable. When processing data, compute outputs normally, compute loss, and backpropagate. The optimizer will update each bias b so as to help the model fit the new data; all weight matrices stay at their pre-trained values [17].
- **Inference:** After tuning, the model uses the original weights and the newly adjusted biases. In effect, each layer's output is shifted by the learned bias change. The inference graph is identical to the base model's graph, just with different bias values.

Because it updates only a tiny fraction of parameters, BitFit is extremely memory-efficient [17] [18]. However, it has limited capacity: it works best when the task can be solved by biasing outputs (e.g. classification) and on smaller models or datasets. In very large models or complex tasks, solely adjusting biases may underperform compared to richer methods like LoRA or adapters [18].

Other Architectures

While the above methods were developed for transformer-based language models, similar ideas extend to other architectures. For example, in **vision models (CNNs or Vision Transformers)** one can insert small convolutional "adapter" layers or bottleneck MLPs between blocks, or tune only batch-norm parameters (a CNN analogue of BitFit biases). Vision-specific prompt-tuning analogues (like adding learnable image patches) have also been explored. The key principle – *freeze most of a large pre-trained model and tune a tiny task-specific part* – remains the same. RNNs or non-transformer networks can likewise use adapters or low-rank updates on their linear layers. In short, while transformers use attention-specific tricks (prefixes, key/value prompts, etc.), the high-level concept of partial fine-tuning carries over: most weights are kept fixed and only a few new parameters (or existing biases) are learned for the new task.

Sources: Established PEFT methods and their mechanisms are detailed in literature [2] [6] [8] [11] [17]. These references explain how each technique injects and trains its parameters within the model. Images illustrating prompt and prefix tuning are from Hugging Face's PEFT overview (prompt tuning [6], prefix tuning [8]), and LoRA's matrix decomposition is depicted in its original description [11] [16].

---

[1] [3] Adapters Strike Back
https://arxiv.org/html/2406.06820v1

[2] [4] [5] Parameter-Efficient Transfer Learning for NLP
https://proceedings.mlr.press/v97/houlsby19a/houlsby19a.pdf

[6] [7] [8] [9] [10] [11] [12] [13] [14] [17] [18] PEFT: Parameter-Efficient Fine-Tuning Methods for LLMs
https://huggingface.co/blog/samuellimabraz/peft-methods

[15] [16] What is LoRA (Low-Rank Adaption)? | IBM
https://www.ibm.com/think/topics/lora