Fine-Tuning Pretrained Large Language Models

Training LLMs

Training for an LLM is a long and compute intensive task. The level of current models such as GPT5 with 1.8trillion. This initial training create a generalist model that struggles with specific information and queries. A more specifically trained model is needed to allow for accurate domain specific responses. Recreating the model for each task in this case is highly inefficient and would lead to bloated systems with many AIs. As every task would require its own fully trained model.

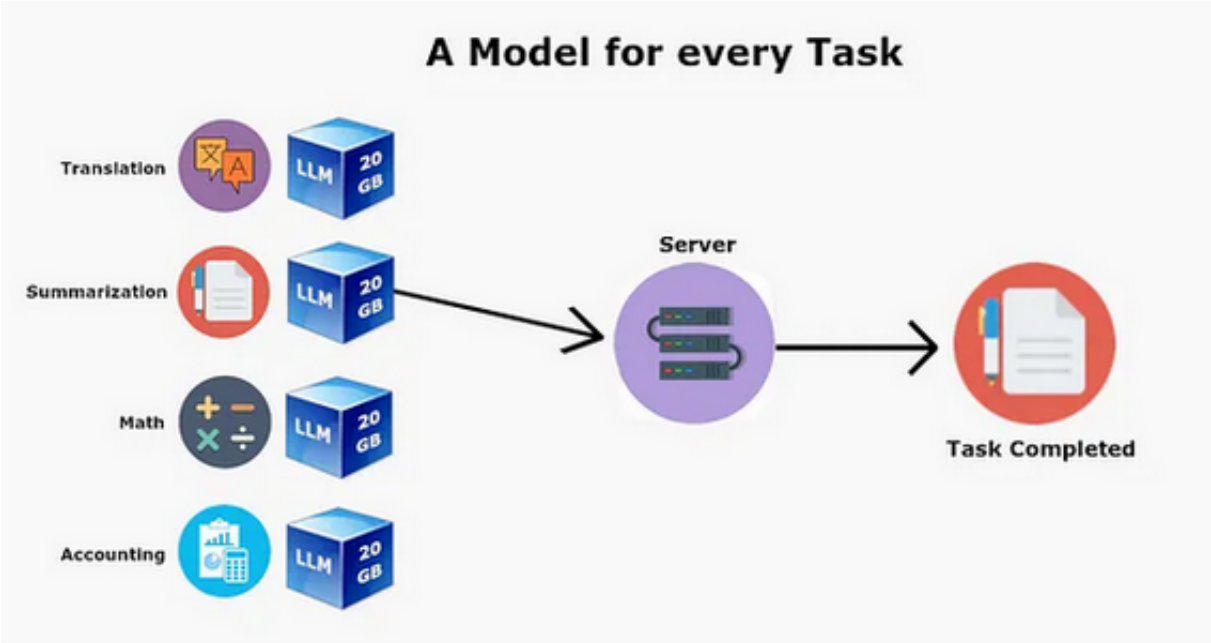

Figure 1: One model for every task

Adapter Architecture

This deficiency in base models opened the door for the adapter architecture to take place. This approach applies an additional layer to the AI adapter. This adapter is a created by freezing the current model weights, then training an additional set of weights to act upon the base model that allows the LLM to have its weights altered only by the adapter letting it be a plug and play solution, The matematical representation of this is at a high level:

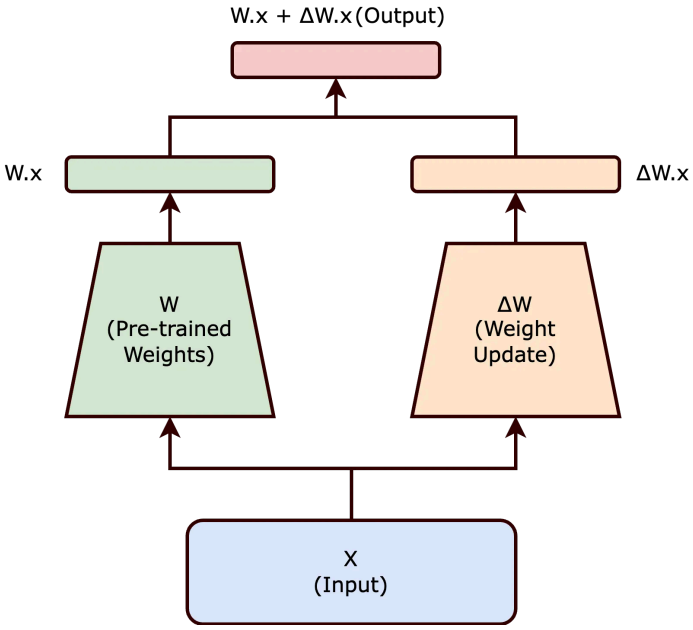

Figure 2: Fine tuning diagram

$$m \in L(D; W.x + \Delta W.x) \quad 1.$$

- L : This represents the loss function used to calculate the gradient that needs to be minimized for the LLM.
- D : This represents the dataset the loss function is being trained on and what is being optimized for.
- Θ : θ_0 represent the weights for base model and $\Delta\theta$ represent the weights from the fine tuning. This is how the adapter is swappable as the weights are not integrated into the base model.

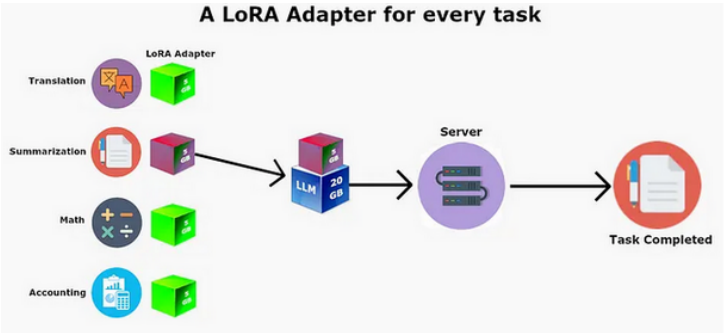

Figure 3: Adapter pattern

Fine Tuning approaches

Full Parameter Fine tuning

Full parameter fine tuning approach was first proposed in 2018 [1] called ULMFiT. This principle has been taken and applied in many forms to models such as DistilBERT[2] and BERT [3]. The base approach is is freezing the original weights then creating a blank matrix of the model weights, then training those to scale each weight individually to bias towards the new target. While efficent it is still a computationally heavy process as every single weight is modified but as a baseline it allows for the adapter architecture to be used.

The formula used to represent this would be

$$m \in L(D; W.x + \Delta W.x) \quad 2.$$

The size of matrix θ_0 and $\Delta\theta$ are both m * n where m and n represent the rows and columns in θ_0. The rest of the definitions are here Equation 1. This method of training allows a small dataset to impact the results of a larger model removing the need to train the model on a huge corpus of data to get tangible results. This being the first step that allowed for fine tuning to be brought forward into conversation for all models.

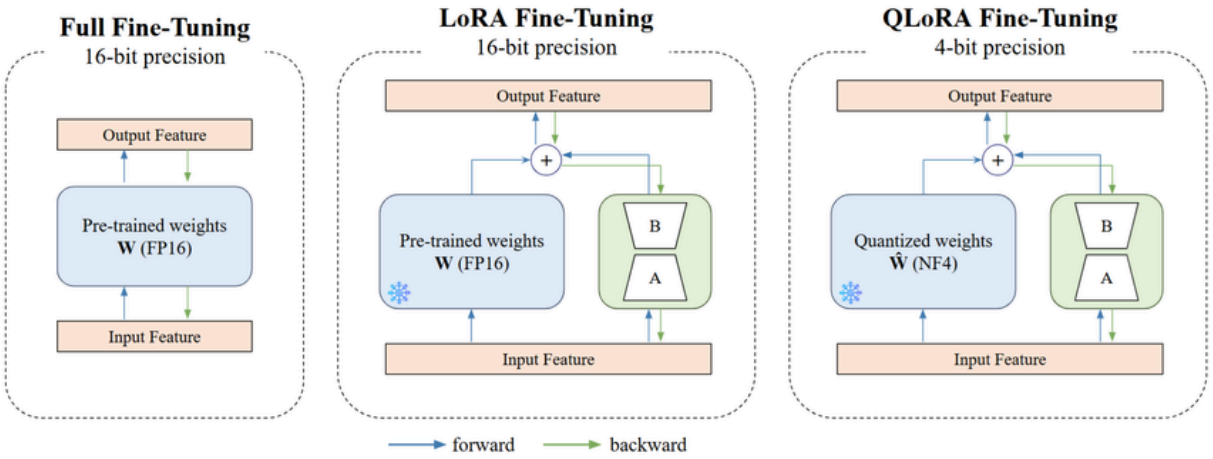

Figure 4: Full, LoRA, QLoRA, Fine-Tuning Comparision [4]

LoRA Fine Tuning

While traditional fine-tuning updates all parameters of a pre-trained model, LoRA (Low-Rank Adaptation), introduced in 2021 (Hu, Shen, Wallis, Allen-Zhu, Li, Wang & Chen), takes a more efficient approach by freezing the original model weights and introducing a small number of additional trainable parameters. This design drastically reduces the computational and memory requirements of model adaptation.

Instead of using a weight update of d^2 like in Fine Tuning, LoRA modifies this process by decomposing the weight update ΔW into the product of two much smaller low-rank matrices, A and B, defined as:

$W' = W + BA$

where

A is a matrix of n rows multiplied by r columns ($A = n*r$)

B is a matrix of r rows multiplied by m columns ($B = r*m$)

r is much smaller than d ($r << d$)

There's visualisation of there here: [4]

Here, the pre-trained weights W are frozen & they remain fixed during training and only A and B are updated. This means that instead of learning d^2 parameters, LoRA learns only $2dr$, significantly reducing the number of trainable parameters when r is small. The product BA serves as a low-rank approximation of ΔW, capturing the essential adjustments needed to specialize the model for a new task without altering the base model directly.

Vera Fine Tuning

Vera [5] fine-tuning is an innovation built on top of LoRA, designed to decrease the memory overhead associated with parameter-efficient fine-tuning. The core of the method is based on Random Matrix Adaptation and LoRa.

Instead of learning two low rank matrices VeRa, begins by generating two low rank matrices of sizes m * r and r * n, which are frozen after the initial generation.

Next two diagonal matrices are created of size m * m and r * r. These diagonal matrices scale the two low rank matrices. The method being similar to a switchboard where each value can amplify or deactivate sections in the low rank matrices without the need to store full parameter sets.

In mathematical terms [5]

$$W.x + \Delta W.x = W.x + \Lambda_b B \Lambda_d A x \tag{3.}$$

- A and B: Are randomly generated low rank matrixes of sizes m * r and r * n which multiply to create the W.x matrix.
- Λ_b and Λ_d: Are diagonal matrixes which are used to scale the A and B matrices. They are of sizes m * m and r * r.

Unlike traditional Lora, Vera only learns the scaling diagonal matrix values. This severely reduces the number of required parameters going from **r(m + n)** to only **m + r**. This significant decrease in learnable parameters does come at a slight decrease of accuracy but the sheer amount of trainable parameters decreased merits this method as a clear innovation on LoRa.

QLoRA Fine Tuning

Building upon LoRA's efficiency, QLoRA (Quantised Low-Rank Adaptation), introduced in 2023 (Dettmers, Pagnoni, Holtzman & Zettlemoyer), further optimises fine-tuning by combining LoRA with Quantised model weights. Where LoRA freezes the original full-precision weights and trains only small low-rank matrices, QLoRA first Quantises those frozen base weights to a 4-bit representation, drastically reducing memory usage while maintaining model performance.

QLoRA uses a 4-bit data type called NormalFloat (NF4), which is optimised for normally distributed weights. It applies block-wise quantisation, normalising each block and mapping it to one of 16 NF4 levels. The weights are stored in this compact 4-bit form and dequantised back to 16-bit only during computation, greatly reducing memory use without sacrificing performance.

In QLoRA, the pre-trained weight matrix W is stored in a quantised form W_4bit, the fine-tuning process still operates the same on low-rank adapters A and B, as in LoRA:

$W' = W_4bit + B*A$

where

A is a matrix of n rows multiplied by r columns ($A = n*r$)

B is a matrix of r rows multiplied by m columns ($B = r*m$)

r is much smaller than d ($r << d$)

There's visualisation of there here: [4]

The quantised weights W_4bit remain frozen, and only the adapter matrices A and B are updated through backpropagation. Because quantisation compresses W into a 4-bit format and LoRA limits the trainable parameters to $2dr$, QLoRA achieves extreme memory efficiency These improvements allow QLoRA to maintain full 16-bit fine-tuning performance while using a fraction of the memory and compute resources.

Lora vs Full fine tuning

Full fine-tuning updated all 66,955,779 parameters (100% of the model) and required an average of 2.75GB of GPU memory and 372.3 seconds of training time, achieving a validation accuracy of 84.2%. In contrast, LoRA updated only 1,181,955 parameters (1.73% of the model), using a rank of 64 for the adapter matrices, which drastically reduces the number of trainable parameters. This reduced memory usage to an average of 1.875GB and the training time to 299.3 seconds, with a lower validation accuracy of 79.3%. It's noticable that the memory difference is smaller than expected as there's a major difference in the amount of trainable parameters, this is likely due to Colab's environmental setup, where performance caching and other system processes affect the reported memory usage. Even so, these results show that LoRA greatly improves efficiency while still achieving competitive performance, making it an effective option for fine-tuning large models.

Lora Vs Vera

Bibliography

[1] J. Howard and S. Ruder, "Universal Language Model Fine-tuning for Text Classification." Accessed: Oct. 22, 2025. [Online]. Available: http://arxiv.org/abs/1801.06146

[2] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." Accessed: Oct. 22, 2025. [Online]. Available: http://arxiv.org/abs/1910.01108

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” Accessed: Oct. 22, 2025. [Online]. Available: http://arxiv.org/abs/1810.04805

[4] “Parameter-efficient fine-tuning: LoRA and QLoRA compared to FFT..” Accessed: Oct. 22, 2025. [Online]. Available: https://www.researchgate.net/figure/Parameter-efficient-fine-tuning-LoRA-and-QLoRA-compared-to-FFT_fig2_390192224

[5] D. J. Kopiczko, T. Blankevoort, and Y. M. Asano, “VeRA: Vector-based Random Matrix Adaptation.” Accessed: Oct. 22, 2025. [Online]. Available: http://arxiv.org/abs/2310.11454