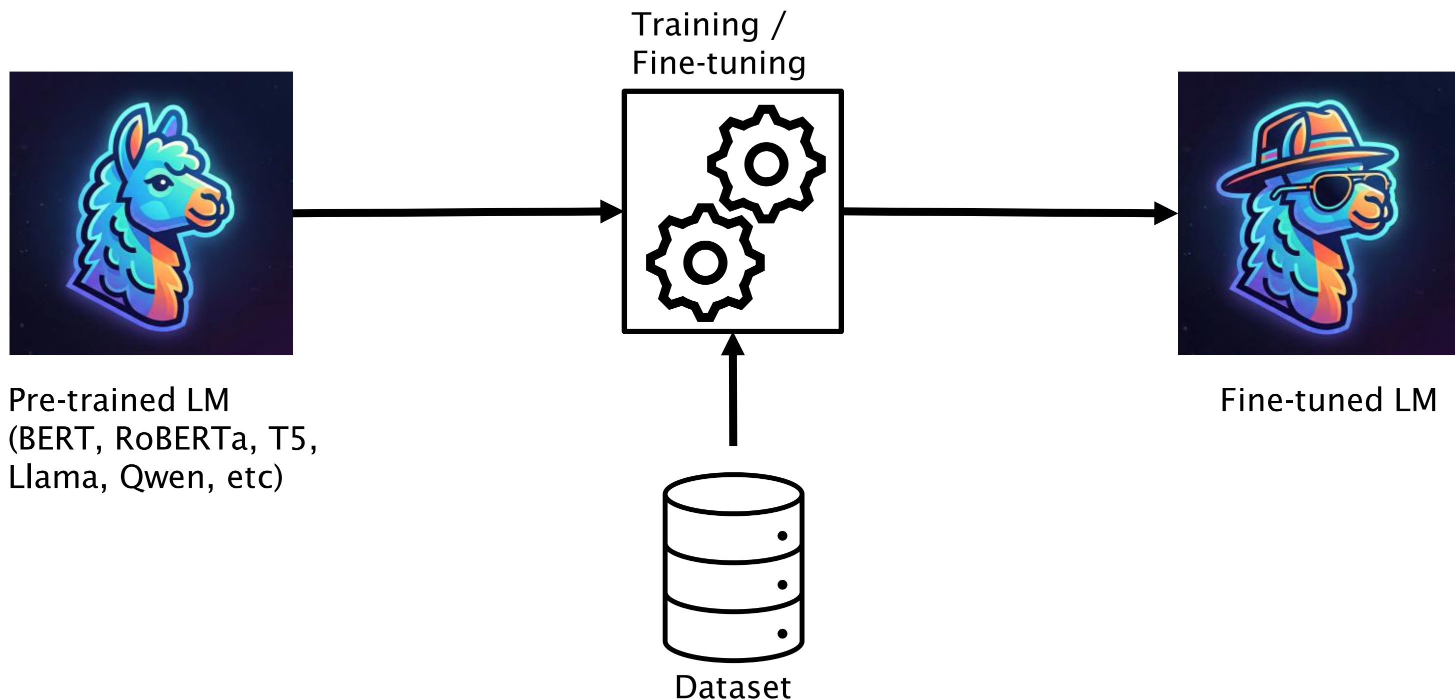
Efficient Fine Tuning of Transformers/LMs

The Pre-training Fine-Tuning Paradigm



Profiling Fine-tuning Procedure

For a weight θ at step t with gradient g_t:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \text{(1st Moment / Momentum)}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{(2nd Moment / Variance)}$$

$$\theta_t = \underbrace{\theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}}_{\text{Adam Update}} - \underbrace{\eta \lambda \theta_{t-1}}_{\text{Weight Decay}} \quad \text{(Decoupled Update)}$$

Key Takeaway: AdamW is "stateful." For every single parameter θ, you must store and update two additional variables (m and v).

$$\begin{aligned} &\text{Total Memory} \\ &= \text{Model Wts } (|\theta|) \\ &+\text{Gradients } (|\theta|) \\ &+\text{activations}(f(B, L, T, d)) \\ &+optimizer(2 * |\theta|) \\ &+ \text{temporary} \end{aligned}$$

- Gradients, optimizer states, and activations can consume **roughly 12–20 × more memory** than the model weights themselves.
- This limits the maximum model size we can train.
- We would still like more free memory for larger context and batch sizes.

How can we Improve Fine-tuning Efficiency?

$$\text{Total Memory (bytes)} = \text{Model Wts } (|\theta|) + \text{Gradients } (|\theta|) + \text{activations}\big(f(B, L, T, d)\big) + optimizer(2 * |\theta|) + \text{ temporary}$$

- Model Wts and/or gradients
  - Prefix Tuning, Adapters, LoRA
- Activations
  - Gradient Checkpointing, Flash Attention, Gradient Accumulation
- Optimizer
  - Muon Optimizer

Other techniques – Quantization, DeepSpeed Offloading, Fused Kernels, Tiling, ….

Gradient

Batch (B,*)

Gradient Accumulation Algorithm

Setup:

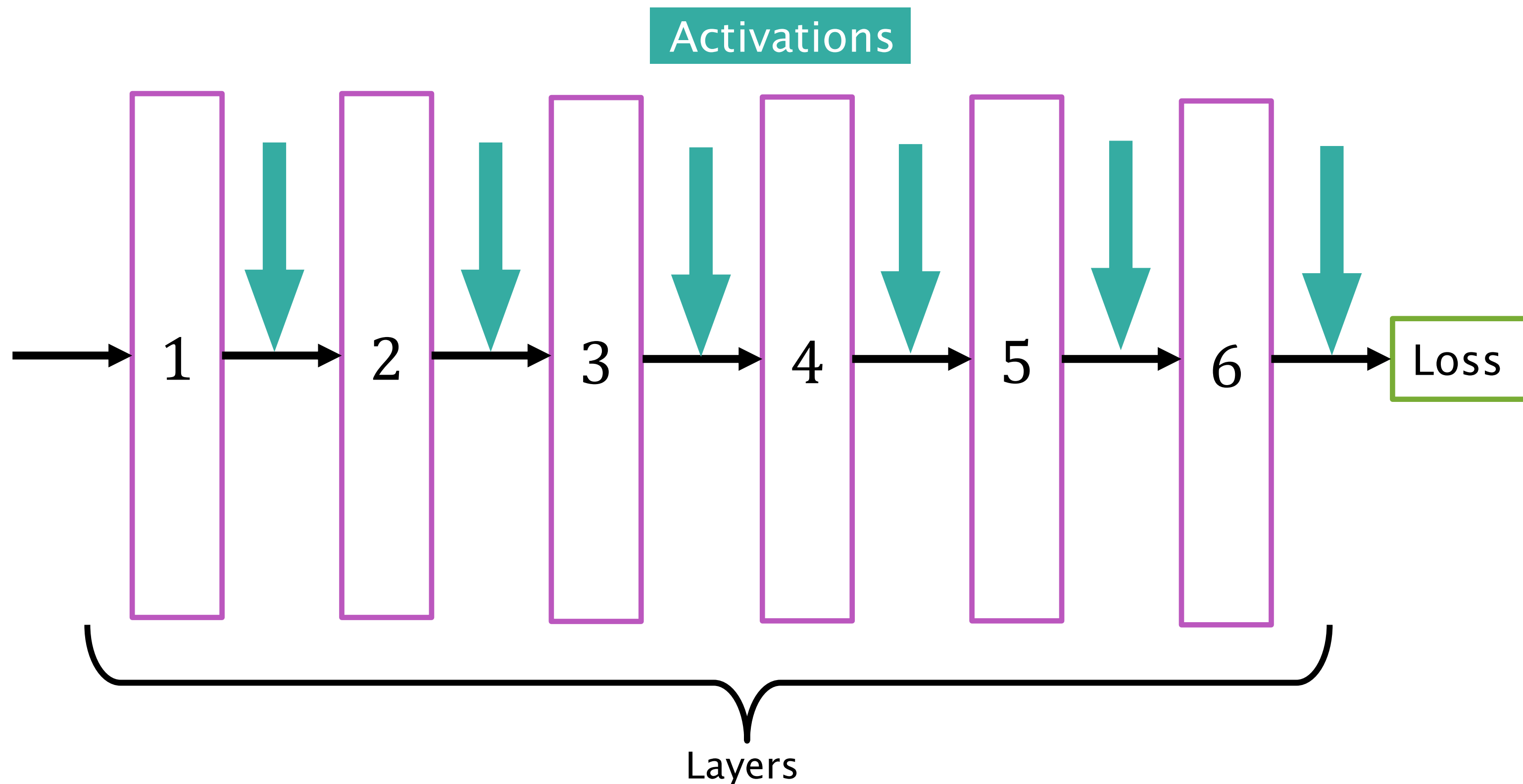
- B_{global}: Desired effective batch size.
- B_{micro}: Physical batch size that fits in VRAM.
- $n = B_{global} / B_{micro}$: Number of accumulation steps.

The Loop:

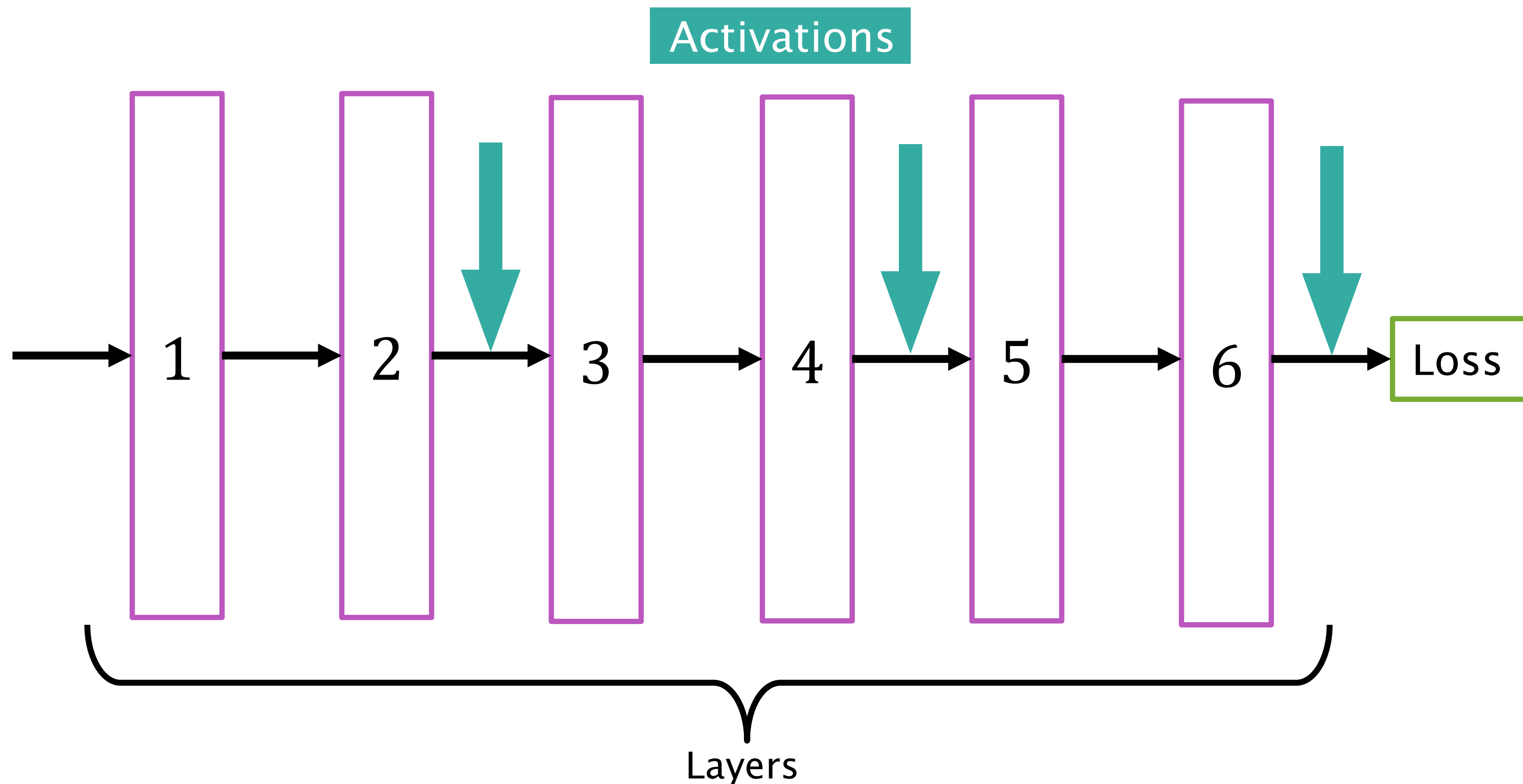
1. **Initialize:** Clear gradients: $\nabla\theta = 0$.
2. **For $i = 1$ to n steps:**
   - **Forward Pass:** Compute loss_i on B_{micro} samples.
   - **Normalize:** $\text{loss}_{scaled} = \text{loss}_i / n$.
   - **Backward Pass:** $\nabla\theta \leftarrow \nabla\theta + \text{backprop}(\text{loss}_{scaled})$.
   - *(Crucial: Gradients are summed in-place, activations are discarded after each step).*
3. **Update:** Apply AdamW step using the accumulated $\nabla\theta$.
4. **Zero Grad:** Reset $\nabla\theta = 0$ and repeat.

ch Gradient =
g (Micro
ch Gradients)

Gradient Checkpointing



Gradient Checkpointing



Prompt Tuning

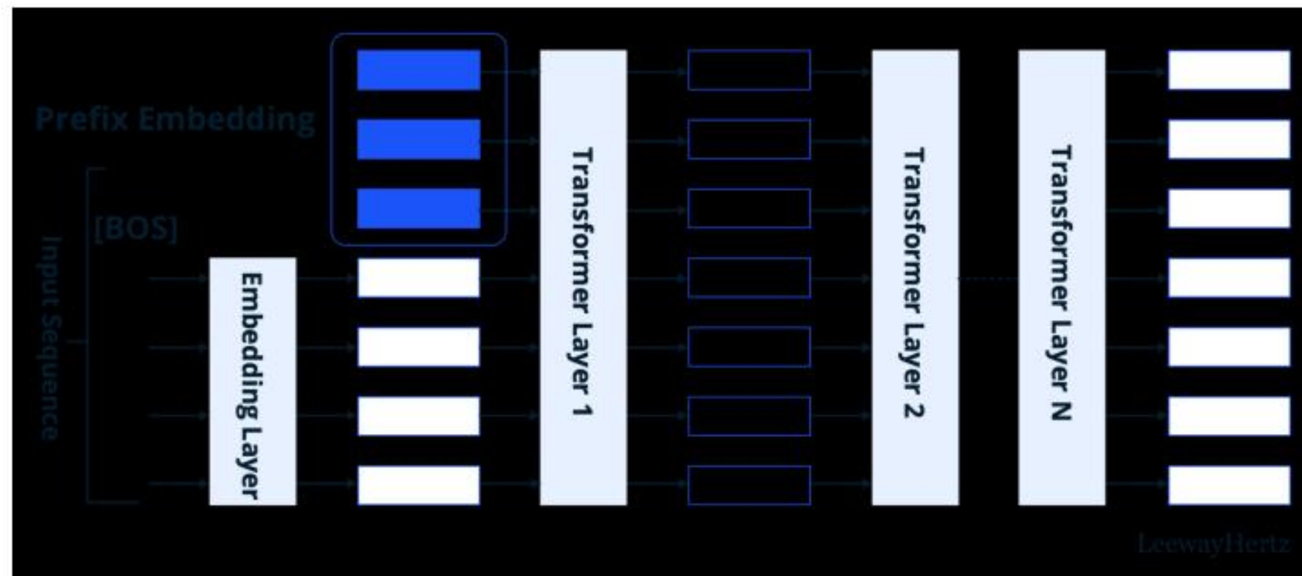


Image Credits: leewayhertz.com

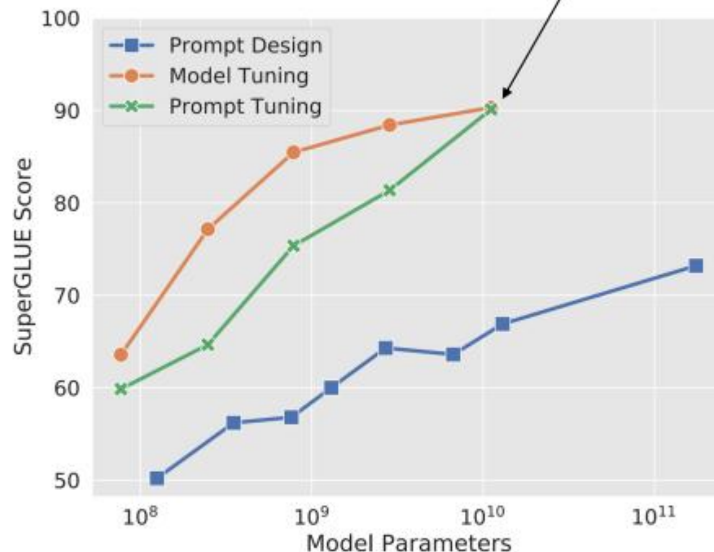
Prompt Tuning Results

Pros:

- Large memory savings in optimizer.
- Allows multi-task inferences.
- Low storage costs.

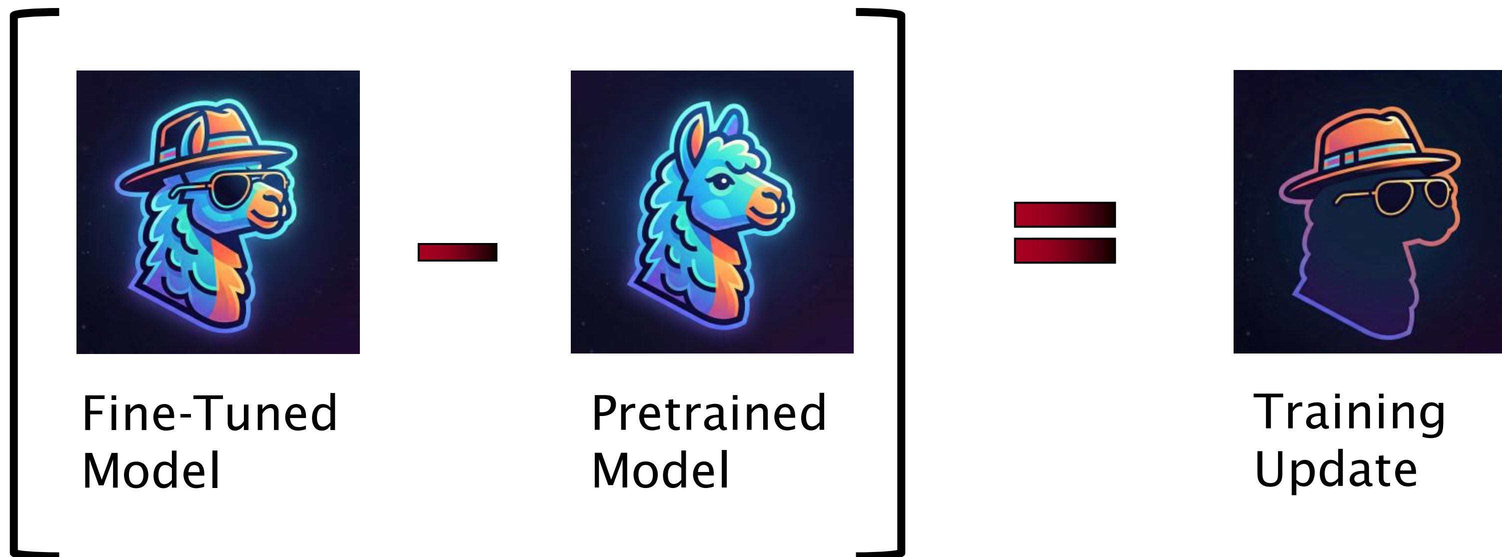
Cons:

- Performance is sub-optimal for smaller models.
- Additional inference cost.



Prompt tuning vs standard fine-tuning and prompt design across T5 models of different sizes [Lester et al., 2021]

A Global Look at Model Fine-tuning

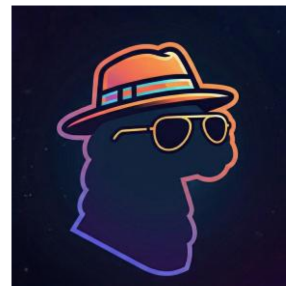


Re-arranging Terms

Fine-Tuned Model θ

[

Pretrained Model θ_0



Training Update

]

Can we enforce some structure on the update to reduce its size?

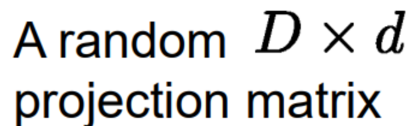
Structure Aware Low Rank Tuning

- Why would this reparameterization reduce memory usage?

- What should be the size d?

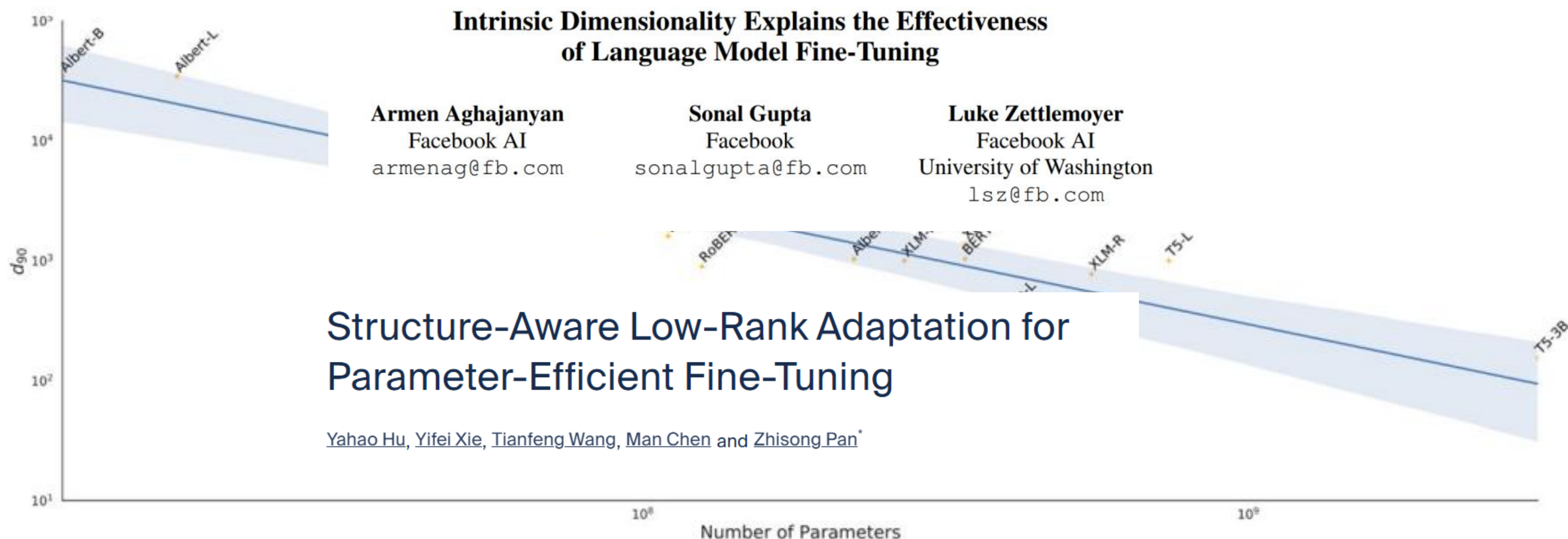
Low-rank fine-tuning:

$$\theta^{(D)} = \theta_0^{(D)} + P\theta^{(d)}$$



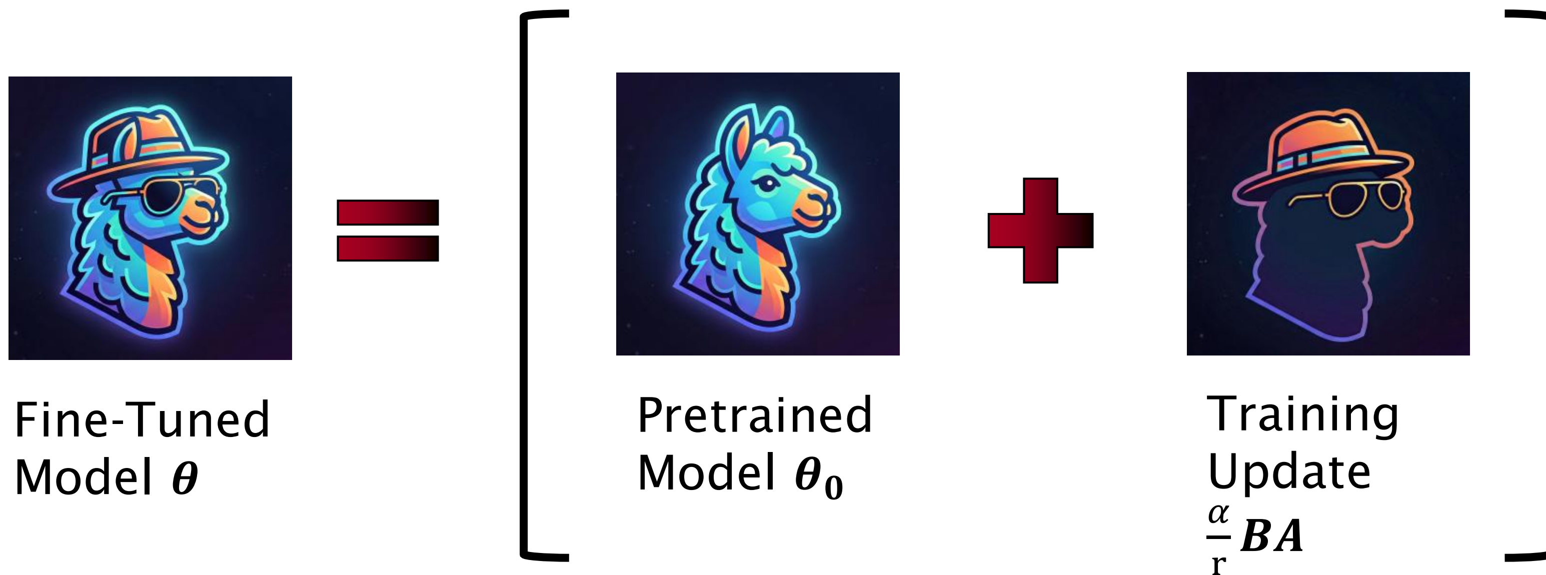
A random $D \times d$ projection matrix

Intrinsic Dimensionality



Intrinsic dimension d_{90} on the MRPC dataset for models of different sizes [Aghajanyan et al., 2021]

Core Idea: Model Updates are Low-Rank



where α is a scaling factor, B and A are matrices of shape (D, r) and (r, D).

Low Rank Adaptations for Transformers

Reparamet… with **traina**…

W

LoRA: Low-Rank Adaptation of Large Language Models

Edward Hu* **Yelong Shen*** **Phillip Wallis** **Zeyuan Allen-Zhu**
Yuanzhi Li **Shean Wang** **Lu Wang** **Weizhu Chen**
Microsoft Corporation
{edwardhu, yeshe, phwallis, zeyuana,
yuanzhil, swang, luw, wzchen}@microsoft.com
yuanzhil@andrew.cmu.edu
(Version 2)

Update ma…
tuning.

Freeze the base model.

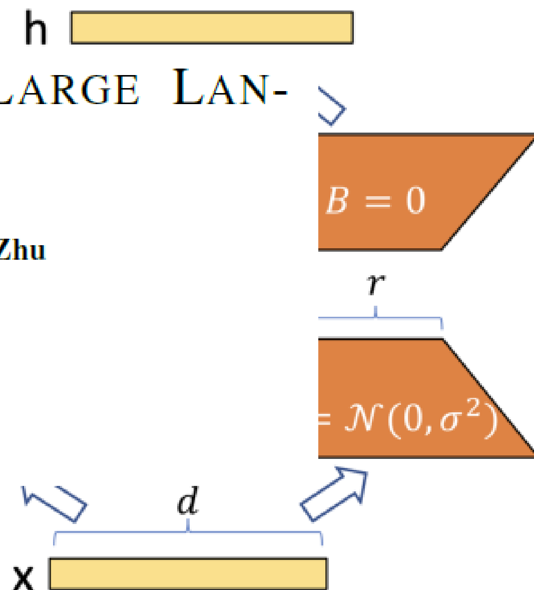


Image Credits: Hu et al., 2022

Efficiency Results

- Up to 2/3 reduction in memory usage with appropriate rank.
- 2

Practical Benefits and Limitations. The most significant benefit comes from the reduction in memory and storage usage. For a large Transformer trained with Adam, we reduce that VRAM usage by up to 2/3 if $r \ll d_{model}$ as we do not need to store the optimizer states for the frozen parameters. On GPT-3 175B, we reduce the VRAM consumption during training from 1.2TB to 350GB. With $r = 4$ and only the query and value projection matrices being adapted, the checkpoint size is reduced by roughly 10,000× (from 350GB to 35MB)[4]. This allows us to train with significantly fewer GPUs and avoid I/O bottlenecks. Another benefit is that we can switch between tasks while deployed at a much lower cost by only swapping the LoRA weights as opposed to all the parameters. This allows for the creation of many customized models that can be swapped in and out on the fly on machines that store the pre-trained weights in VRAM. We also observe a 25% speedup during training on GPT-3 175B compared to full fine-tuning[5] as we do not need to calculate the gradient for the vast majority of the parameters.

LoRA Performance

What about inference cost?

Model & Method	# Trainable Parameters	E2E NLG Challenge BLEU	NIST	MET	ROUGE-L	CIDEr
GPT-2 M (FT)*	354.92M	68.2	8.62	46.2	71.0	2.47
GPT-2 M (Adapter^{L})*	0.37M	66.3	8.41	45.0	69.8	2.40
GPT-2 M (Adapter^{L})*	11.09M	68.9	8.71	46.1	71.3	2.47
GPT-2 M (Adapter^{H})	11.09M	$67.3_{\pm.6}$	$8.50_{\pm.07}$	$46.0_{\pm.2}$	$70.7_{\pm.2}$	$2.44_{\pm.01}$
GPT-2 M (FT^{Top2})*	25.19M	68.1	8.59	46.0	70.8	2.41
GPT-2 M (PreLayer)*	0.35M	69.7	8.81	46.1	71.4	2.49
GPT-2 M (LoRA)	0.35M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.85}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{71.8}_{\pm.1}$	$\mathbf{2.53}_{\pm.02}$
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (Adapter^{L})	0.88M	$69.1_{\pm.1}$	$8.68_{\pm.03}$	$46.3_{\pm.0}$	$71.4_{\pm.2}$	$\mathbf{2.49}_{\pm.0}$
GPT-2 L (Adapter^{L})	23.00M	$68.9_{\pm.3}$	$8.70_{\pm.04}$	$46.1_{\pm.1}$	$71.3_{\pm.2}$	$2.45_{\pm.02}$
GPT-2 L (PreLayer)*	0.77M	70.3	8.85	46.2	71.7	2.47
GPT-2 L (LoRA)	0.77M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.89}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{72.0}_{\pm.2}$	$2.47_{\pm.02}$

LoRA Merging

- Add the final LoRA weights into the frozen weights → No inference latencies.

- On-the-fly Task Swapping:
  - Train LoRA *adapters* for multiple tasks.
  - Swap-in the required LoRA adapter at run-time → GPU efficiency

Mixing LoRA Weights

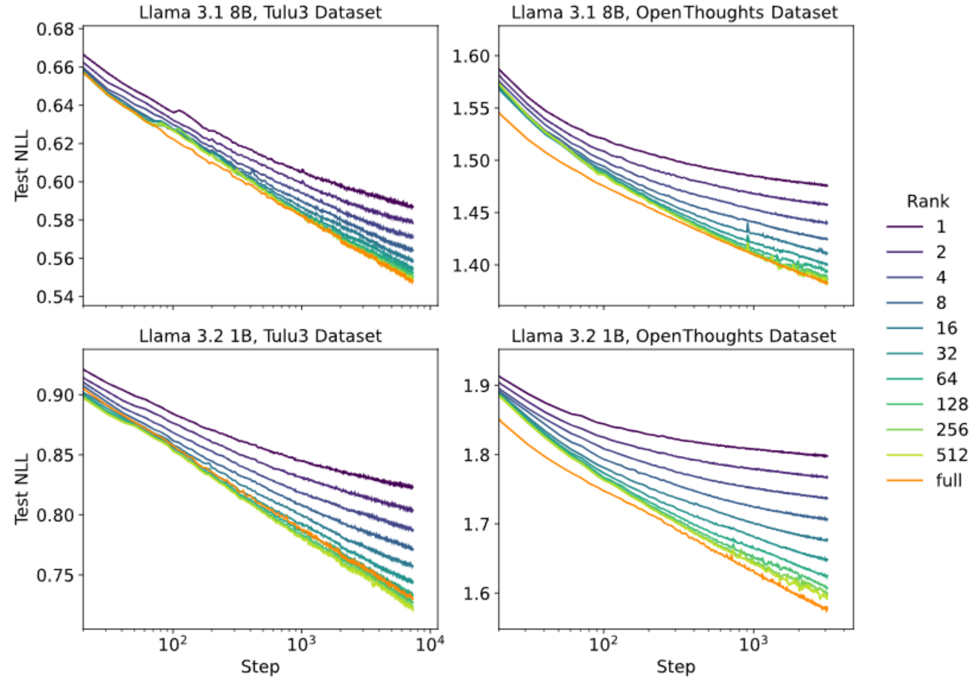
Figure 2: Overall architecture of our proposed MoA.

Credit: Feng, et.al, Mixture-of-LoRAs: An Efficient Multitask Tuning for Large Language Models

Hyperparameters

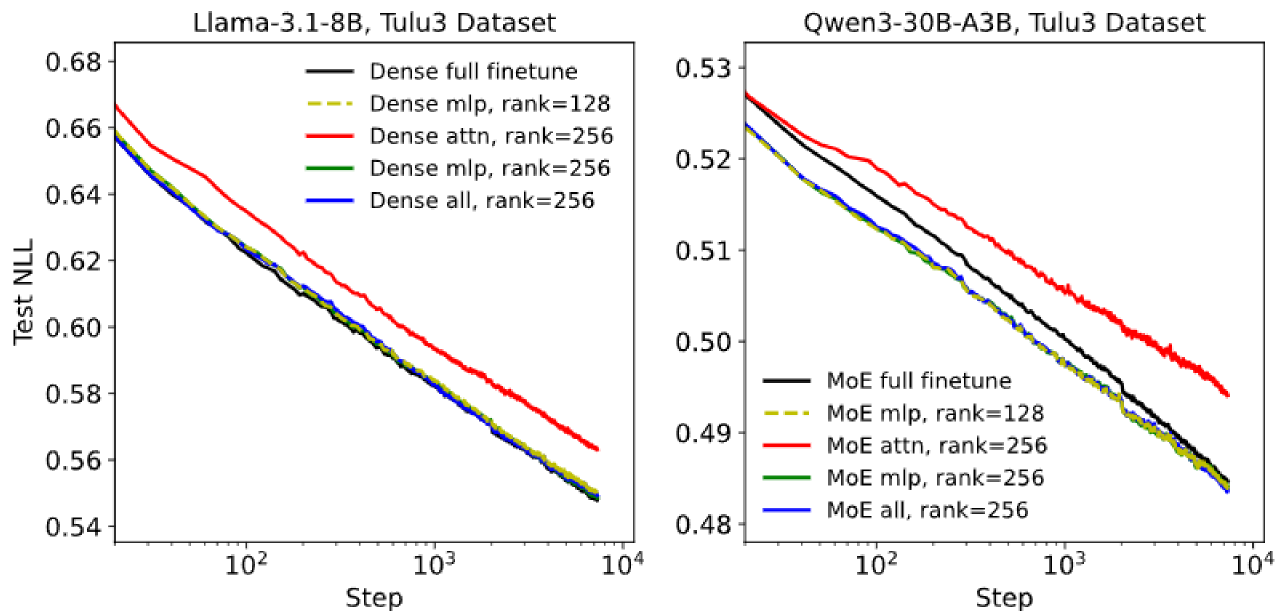
- Rank and α
- Learning Rate
- Layers Where LoRA is applied

Setting LoRA Rank



Credit: https://thinkingmachines.ai/blog/lora/

Layers Where LoRA is applied



Credit: https://thinkingmachines.ai/blog/lora/

Further Variants

- QLoRA: Efficient Finetuning of Quantized LLMs [Dettmers, et.al, NeurIPS, 2023]
- A Rank Stabilization Scaling Factor for Fine-Tuning with LoRA [Kalajdzievski, D. (2023)]
- LongLoRA: Efficient Fine-tuning of Long-Context Large Language Models [Chen, et.al, ICLR 2024]