Why Do We Fine-Tune?

1. Task-Specific Performance Improvements

- Fine-tuning enhances an LLM's accuracy and efficiency for particular tasks.
- Example:
  - A generic LLM can be fine-tuned for Text-to-SQL tasks, leading to significantly better performance in those areas.

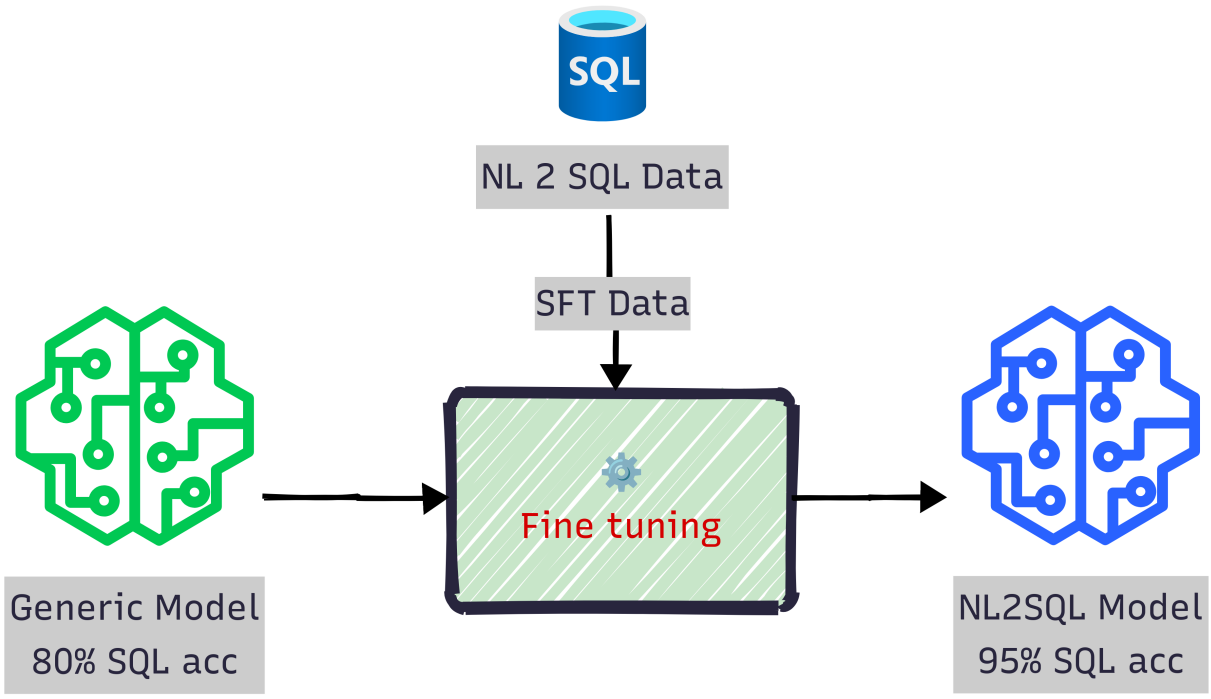


Figure 1: Text-to-SQL

2. Adapting to a New Domain

- Fine-tuning helps specialize a general model for a specific domain, improving its understanding of domain-specific language and terminology.
- Example:
  - A General LLM might struggle to understand medical abbreviations like "STAT" (immediately) or "PRN" (as needed), but fine-tuning on clinical notes improves accuracy.
  - Generalizing LLM for Ecommerce domain to do specific ecommerce tasks like Attribute Extrcation, Product Recommendation etc.

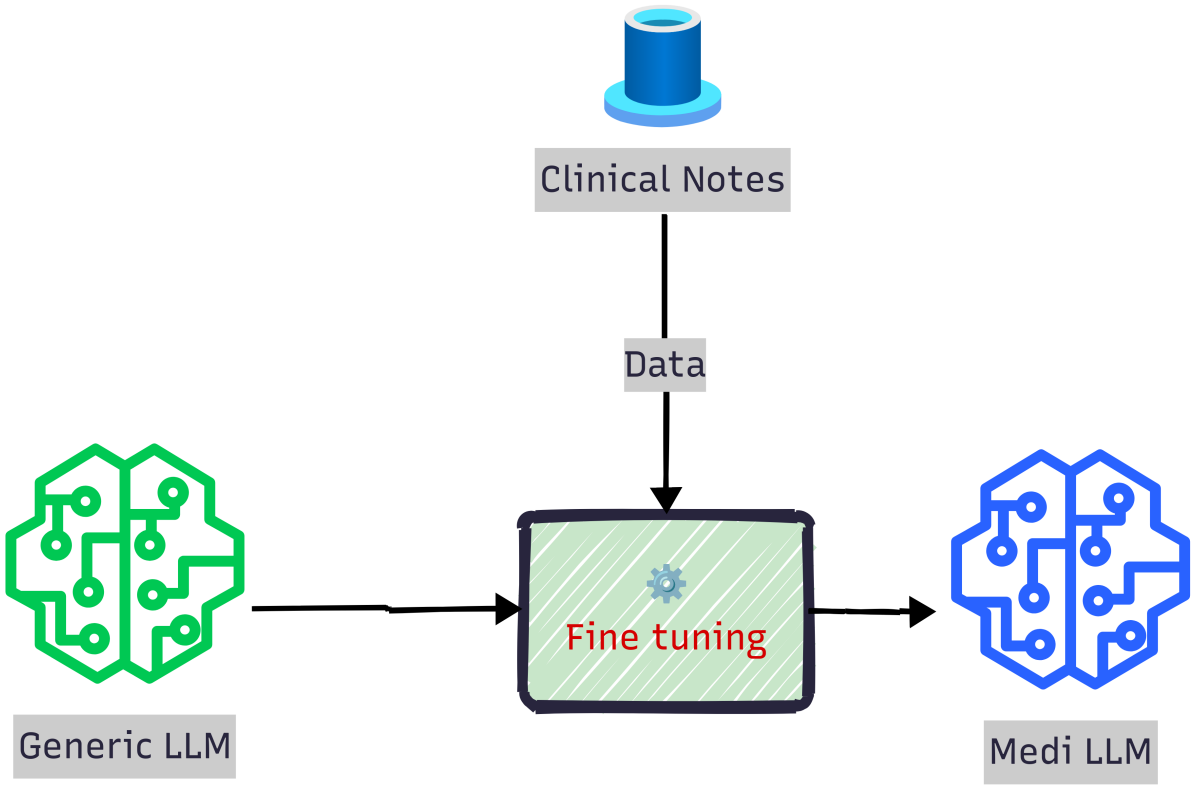


Figure 2: Medical LLM

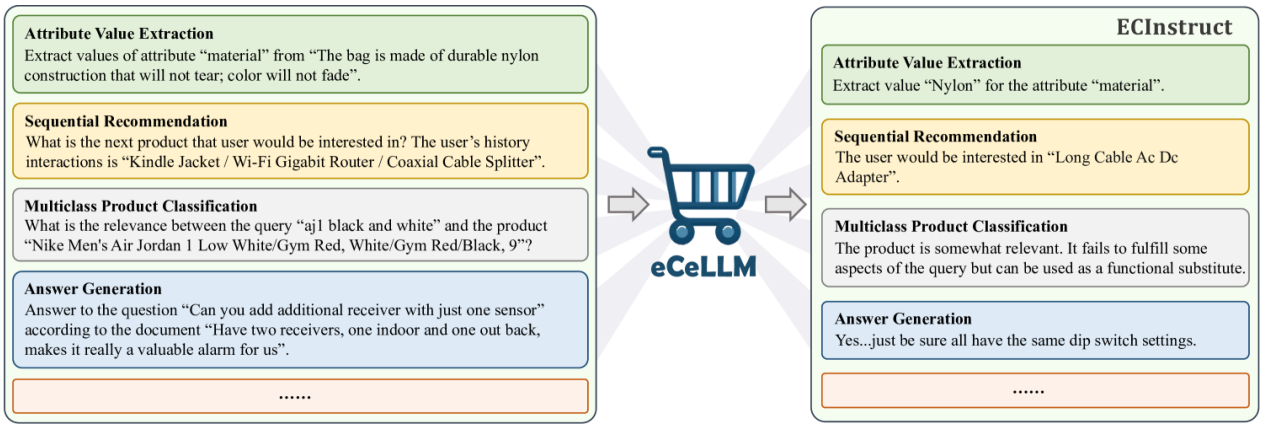


Figure 3: Ecommerce LLM

3. Customizing Output Characteristics & Alignment with Business/User Needs

- Fine-tuning adjusts response tone, verbosity, and structure to match specific requirements.
- Example:
  - Customizing a chatbot's responses to match a brand's tone — formal for banking, casual for gaming.
  - LLM can be finetuned to write news summaries in different styles (e.g., "concise and factual" vs. "engaging and storytelling").

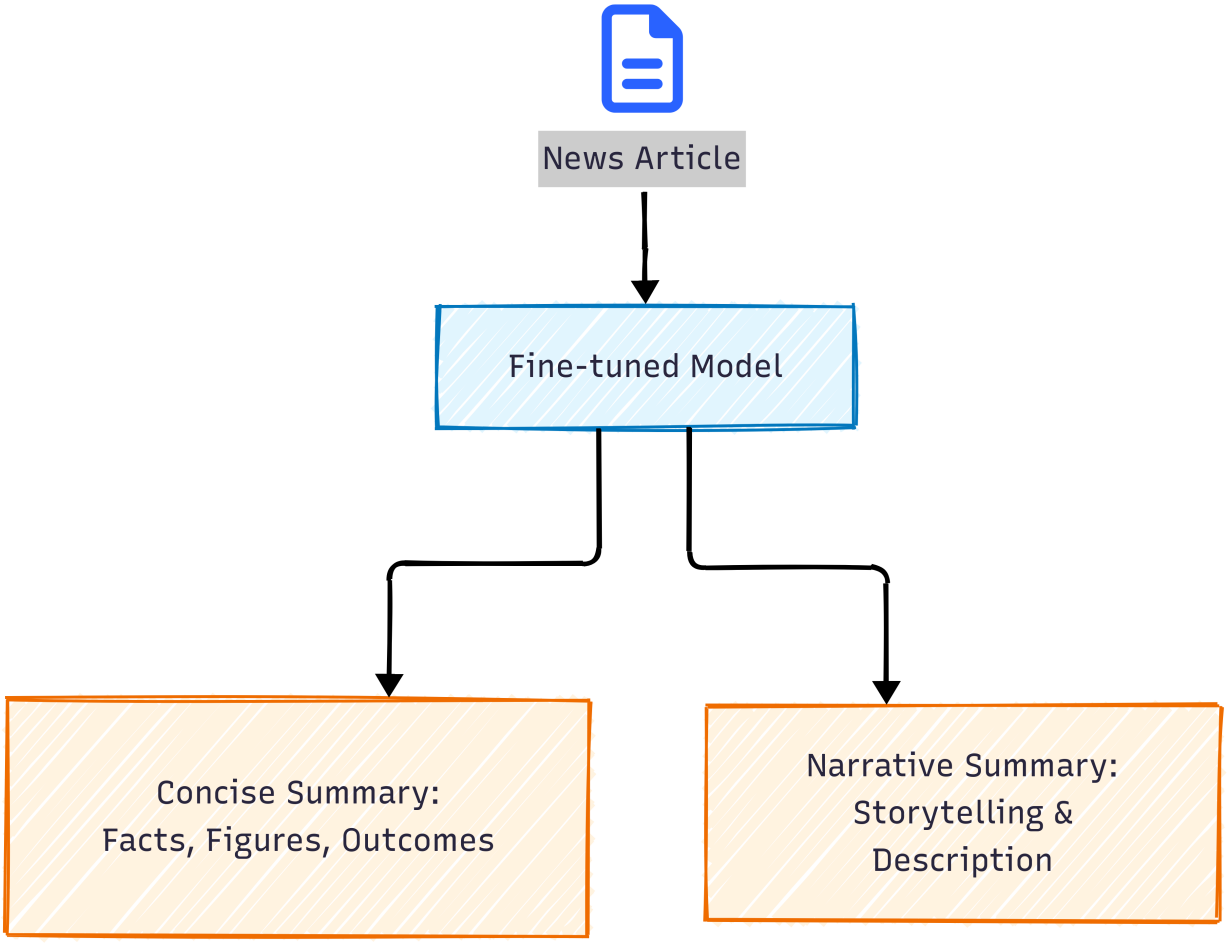


Figure 4: Controlling Output Characteristics

4. Generalized Model to Specific Model

- Fine-tuning allows us to tailor a broadly trained model for a specific application.
- Example:
  - A general X-ray interpretation model can be fine-tuned to specialize in chest X-ray analysis, improving its ability to detect pneumonia.
  - A general image classification model trained on diverse objects can be fine-tuned for specific tasks like identifying defective parts in manufacturing.

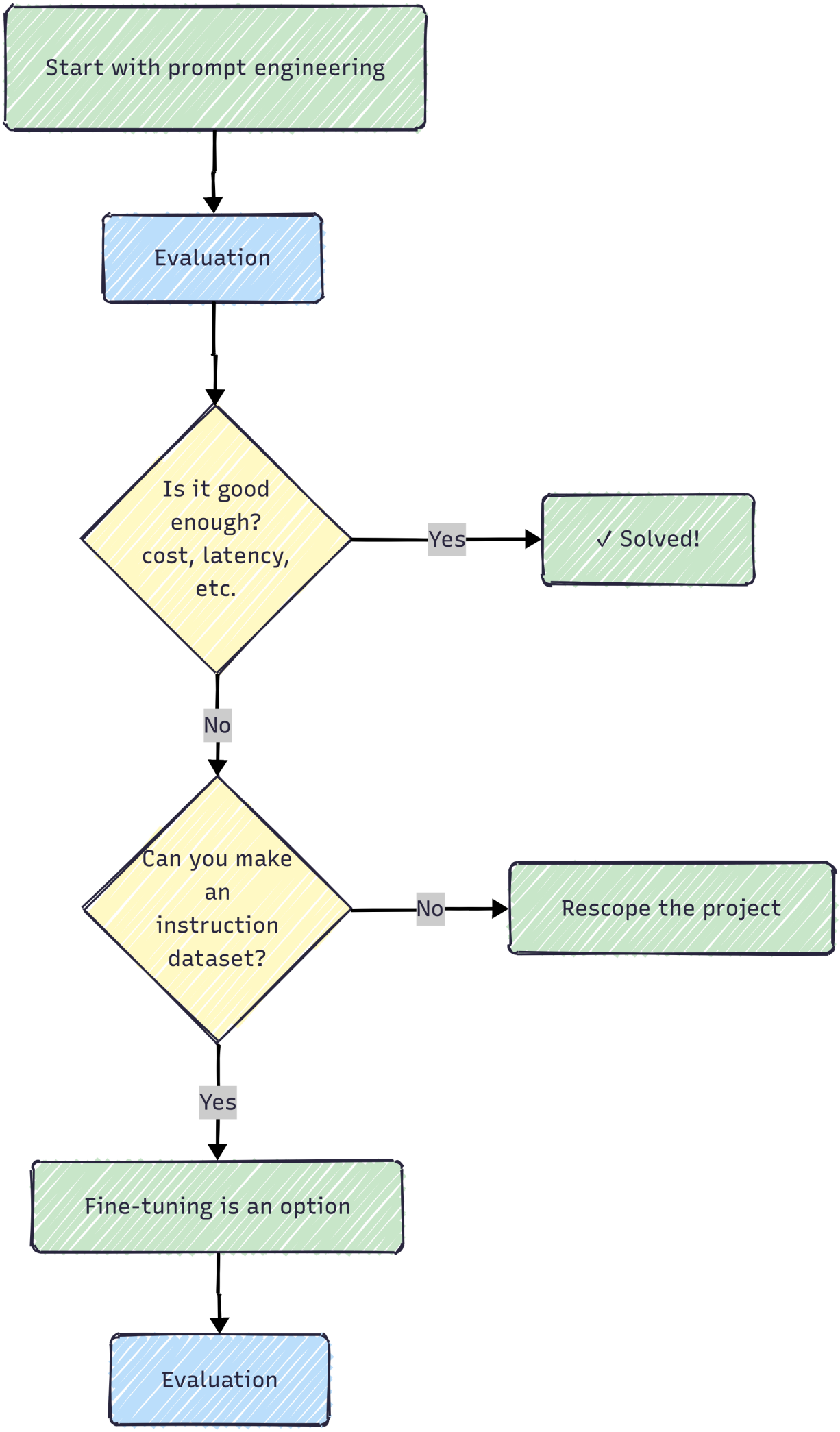
5. Handling Data Distribution Shifts Over Time

- As real-world data evolves, models trained on older data may become outdated. Periodic fine-tuning ensures continued relevance and accuracy.
- Example:
  - A financial LLM trained on past economic trends can be fine-tuned to adapt to new market conditions, regulations, and emerging risks.

Finetuning vs Prompting

Why finetuning?

- Data Privacy
- Quality vs latency tradeoff - Typically works better than a larger model (faster and cheaper because it doesn't require a very long prompt)
- Extremely narrow task
- Customization - Provides better alignment with task of interest because it has specifically trained for it.
- Prompting is not enough or impractical
- Can be used to teach new facts and information to the model (such as advanced tools or complicated workflows)


Start with prompt engineering
Evaluation
Is it good enough? cost, latency, etc.
Yes
✓ Solved!
No
Can you make an instruction dataset?
No
Rescope the project
Yes
Fine-tuning is an option
Evaluation

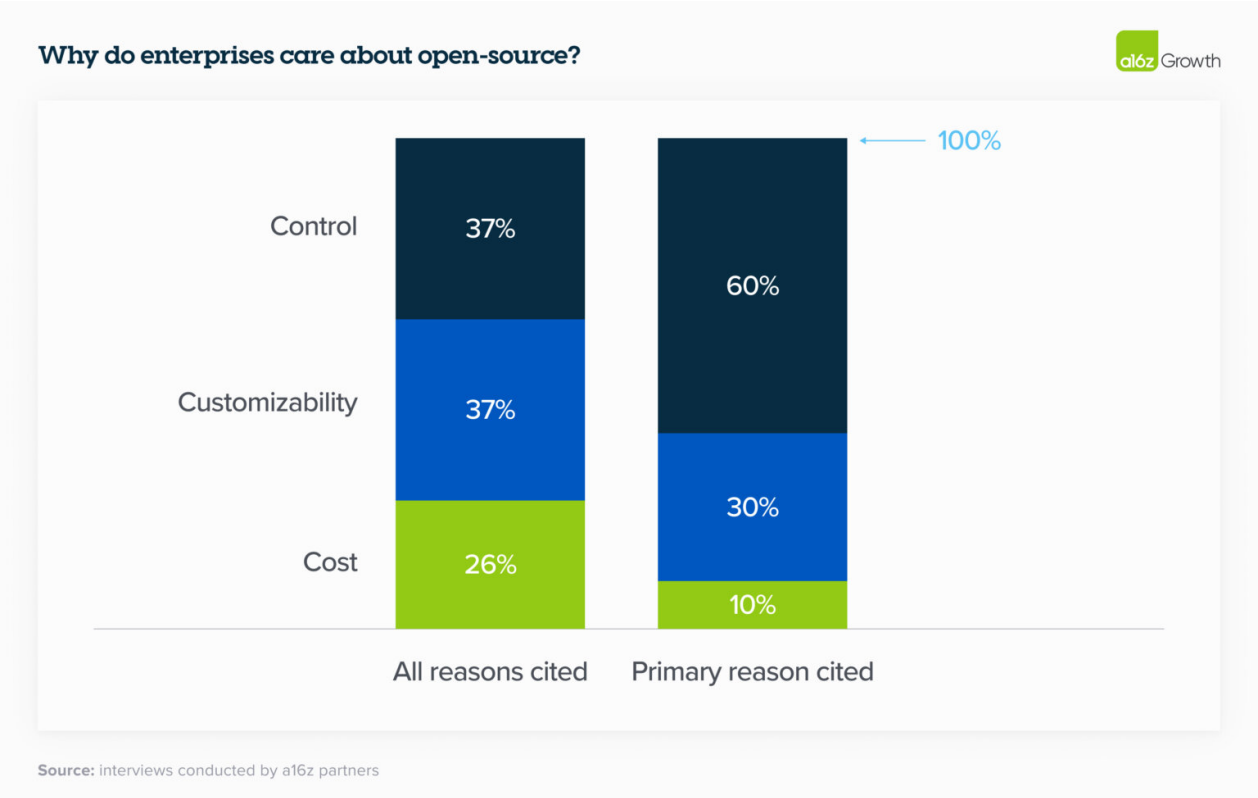


Figure 6: Why enterprice care about finetuning open source models (https://a16z.com/generative-ai-enterprise-2024/)

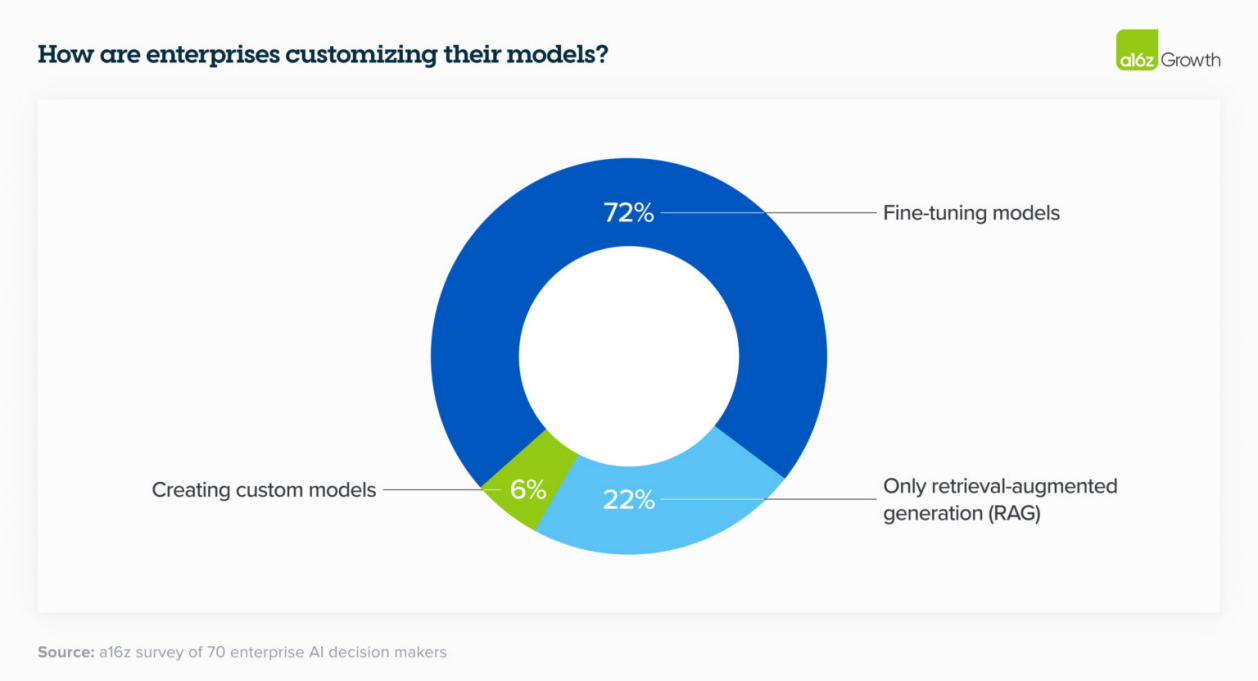


Figure 7: (https://a16z.com/generative-ai-enterprise-2024/)

Supervised Fine-Tuning (SFT)

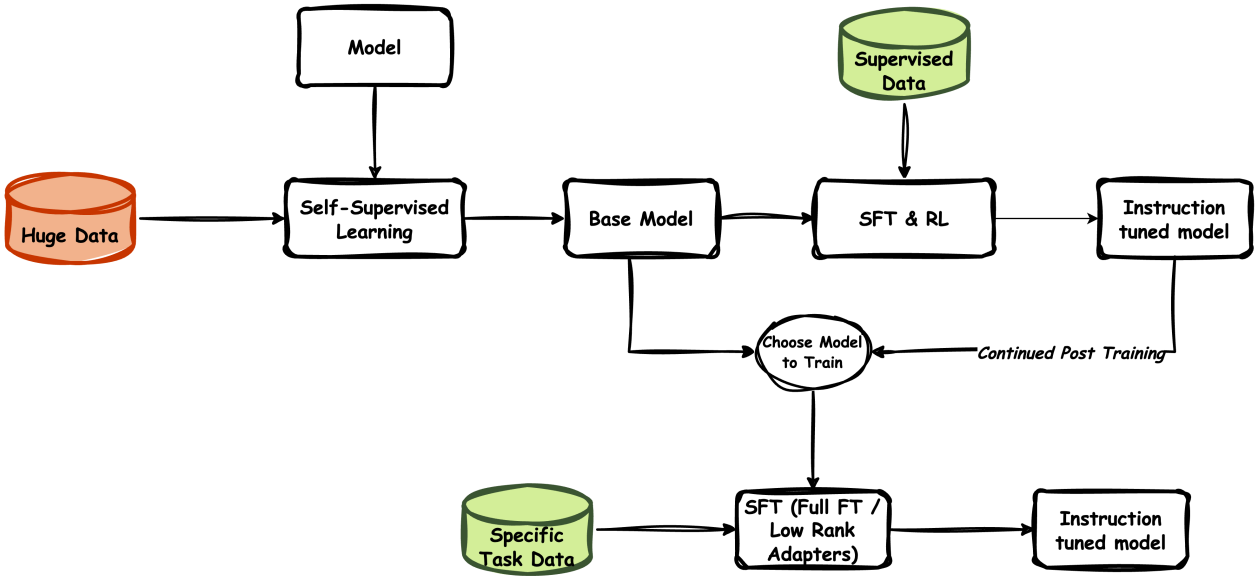


Figure 8: SFT

System - You are a helpful assistant, who always answers exactly what is asked.

Input	Output
Extract Intent of query "Can you tell me the weather forecast for tomorrow?"	Intent: Weather Inquiry
Write a Python function to check if a number is even.	`def is_even(n): return n % 2 == 0`

- Like pretraining, SFT use next token prediction loss to train the model.
- Main difference is that SFT uses a labeled instrction and output pair to train the model.

Data for Fine-Tuning LLM to make better at Function Calling

Given a query "Determine the GCD of 144 and 60", other than calculating the GCD, it calls a function that was available for LLM to get the GCD like {`"name": "math.greatest_common_divisor", "arguments":{"a": 144, "b": 60}`}

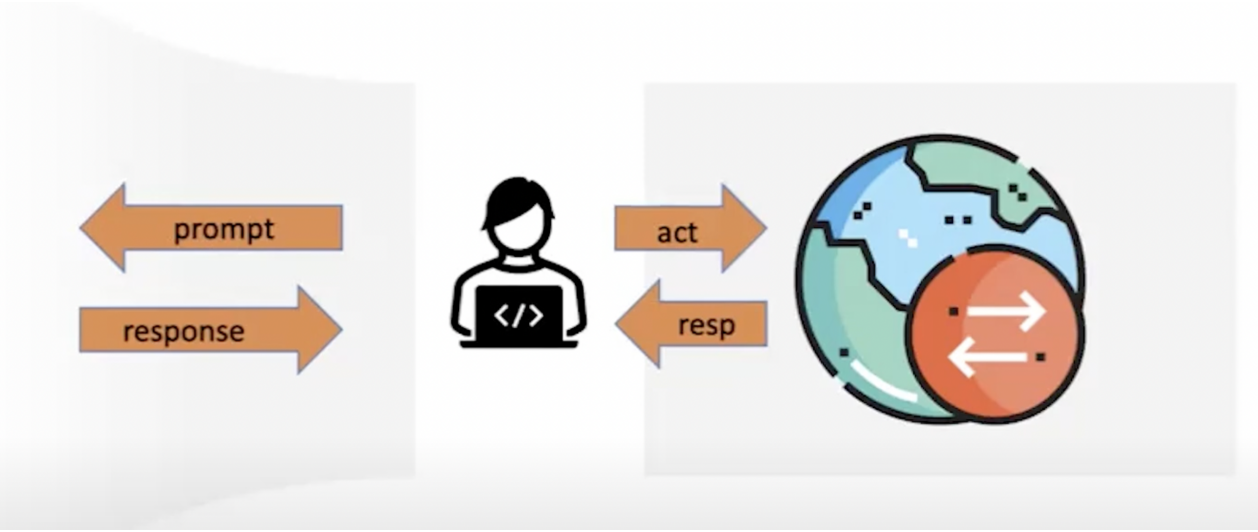


Figure 9

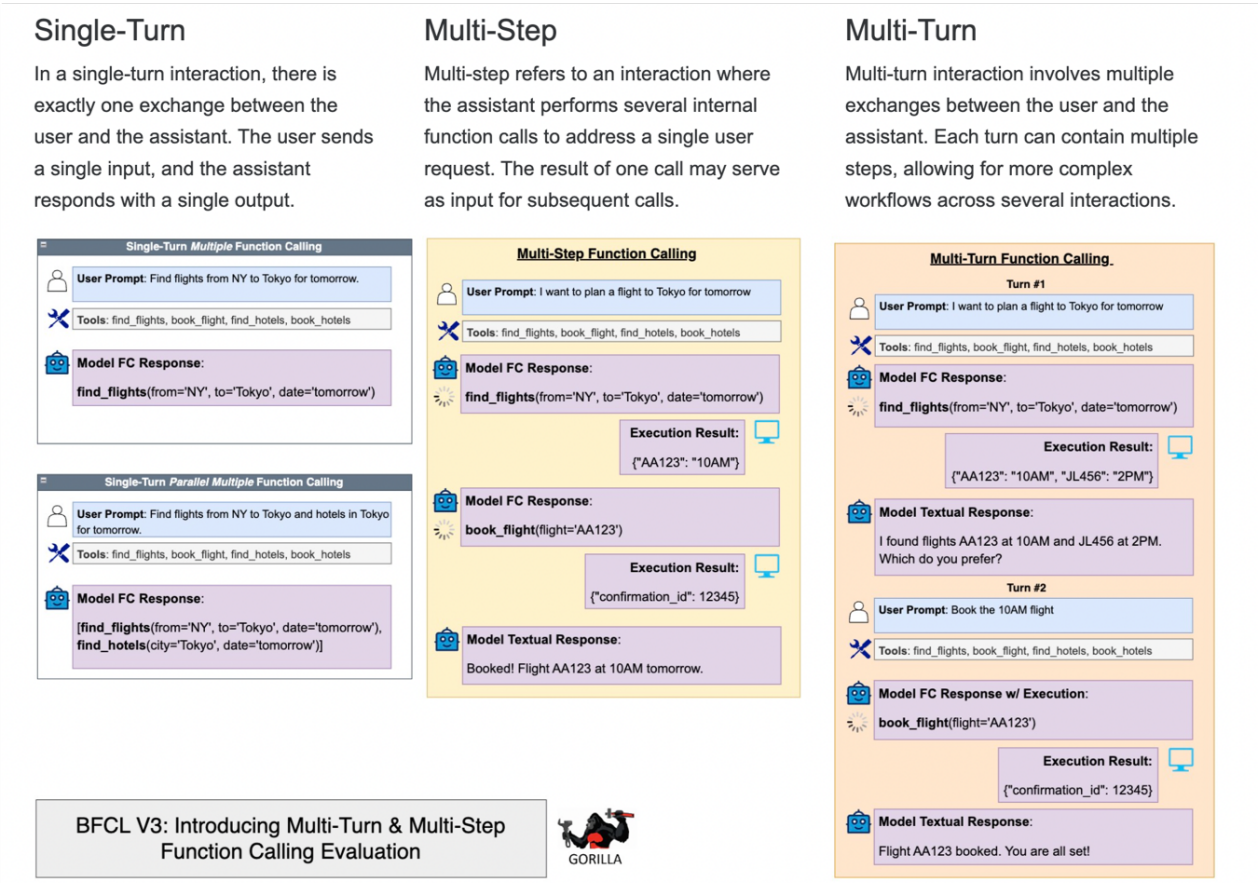


Figure 10

Creating SFT Data for Function Calling

Data Creation Recipe

1. **Open Source Data** - Combine relevant data from open source datasets. Do search over huggingface datasets to find relevant datasets.

2. **Formatting** - Format the data into a structured format that can be used for fine-tuning. In Function calling, format function definitions and function calls to uniform format.

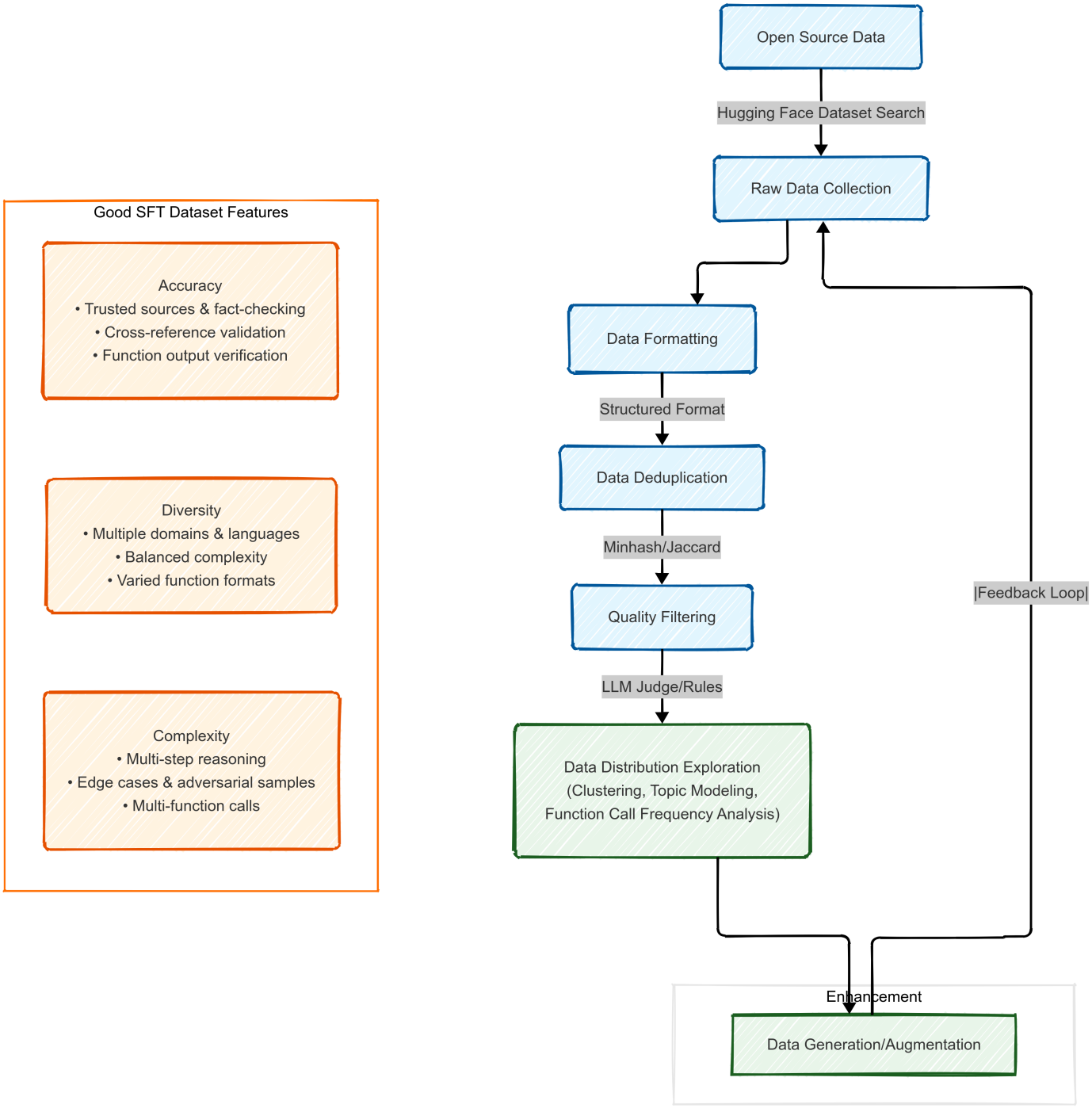


Figure 11: SFT Data Creation Recipe

3. **Data Deduplication** - Exact or fuzzy duplicate detection (using techniques like minhash, jaccard similarity, etc.)

4. **Data Quality** - Filter out low quality data using rules or LLM as a judge methodology.

5. **explore data** - Explore the data to understand the distribution of data and identify any patterns or biases - do clustering, topic modeling, frequency analysis(different type of function calling).

6. **Data Generation or Augmentation** - Generate more data to make better distribution of data or augment data and repeat the process.

Issues with SFT

- Use of special tokens and SFT template - Make sure to use the same template as the one used for pretraining.
- Try to use pad tokens clearly. Lllama uses a special pad token than eos token so make sure to use the same.

Full Fine-Tuning

Full fine-tuning is a process where we train all model parameters.

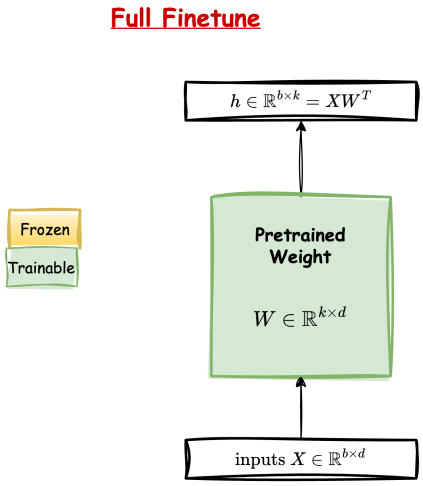


Figure 12: Full Fine-Tuning

Lets estimate the memory required for full fine-tuning for 1.5B parameter model.

Key Components of Memory Usage:

1. **Model Weights:**
   - fp16/bf16 $\rightarrow$ (2 bytes/parameter) . No of parameters $= 2 \times 1.5 \times 10^9$ bytes $\simeq 3GB$
   - fp32 $\rightarrow$ (4 bytes/parameter) . No of parameters $= 4 \times 1.5 \times 10^9$ bytes $\simeq 6GB$
2. **Optimizer State:**
   - **Standard Adam:**
     - Momentum: 4 bytes/param
     - Variance: 4 bytes/param
     - FP32 copy of parameters: 4 bytes/param
     - **Total:** 12 bytes/parameter $= 12 \times 1.5 \times 10^9$ bytes $\simeq 18GB$
   - **8 bit adam:**
     - Momentum: 1 byte/param
     - Variance: 1 byte/param
     - FP32 copy of parameters: 4 bytes/param
     - **Total:** 6 bytes/parameter $= 6 \times 1.5 \times 10^9$ bytes $\simeq 9GB$

3. **Gradients:** gradient datatype often matches the model datatype

   - fp16/bf16: 2 bytes/param = $2 \times 1.5 \times 10^9$ bytes $\simeq 3GB$
   - fp32: 4 bytes/param = $4 \times 1.5 \times 10^9$ bytes $\simeq 6GB$

4. **Activations:** Depends on sequence length and architecture. For vanilla transformer, activation memory per transformer block is calculated as follows:

Component	**Explanation**	**Memory (bytes)**
Q, K, V matrix multiplies	We only need to store their shared input	$2sbh$
QK^T matrix multiply	It requires storage of both Q and K with total size	$4sbh$
Softmax	Softmax output	$2as^2b$
Softmax dropout	Only a mask	as^2b
Attention over V	We need to store the dropout output and V	$2as^2b + 2sbh$
MLP	Linear(h-4h) + GeLU + Linear(4h-h) + dropout	$2sbh + 8sbh + 8sbh + sbh = 19sbh$

   - total memory per transformer block is $34sbh + 5as^2b$
   - If we use 'flash attention' we can remove the memory $5as^2b$ beacuse it recomputes the attention without storing the intermediate results.
   - total memory per transformer block is $34sbh$

Sequence Length	**Activation Memory (GB)**
512	1.5-2 GB
1024	3-4 GB
2048	6-7 GB
8192	24-25 GB

5. **Gradient Checkpointing:**

   - Saving all activations from the forward pass in order to compute the gradients during the backward pass can result in significant memory overhead. The alternative approach of discarding the activations and recalculating them when needed during the backward pass, would introduce a considerable computational overhead and slow down the training process.
   - Gradient checkpointing offers a compromise between these two approaches and saves strategically selected activations throughout the computational graph so only a fraction of the activations need to be re-computed for the gradients

Techniques to reduce memory usage and speed up training

Method/tool	Improves training speed	Optimizes GPU Memory
Batch Size	Yes	Yes
Gradient Accumulation	No	Yes
Gradient Checkpointing	No	Yes
Mixed Precision Training	Yes	Maybe
Optimizer Choice	Yes	Yes
torch compile	Yes	No
Data Pre-Loading	Yes	No
PEFT Techniques	No(may be for very large models)	Yes

Base Configuration

Parameter	Value
Model Used	Qwen 1.5B
Batch Size	16
Learning Rate	5e-5 + Cosine Annealing
Number of Epochs	2
Number of Data Points	80K
Gradient Checkpointing	True
Gradient Accumulation	8
GPU Used	1 A100 80GB
Attention Implementation	Flash Attention 2
Max Sequence Length	2048

Training Results

Technique	params trained	time / epoch	GPU Peak Memory	Simple AST	Parallel AST	Multiple AST	Parl. Mult. AST	Irr. Detection	Avg
Prompt				70	68	76	60	59.5	66.7
SFT	1.5B (100%)	2hr 15m	76.5 GB	88.75	81.5	90.5	72	60.83	78.7

Intrinsic Dimensionality and Fine-Tuning

there is a paper from "INTRINSIC DIMENSIONALITY EXPLAINS THE EFFECTIVENESS OF LANGUAGE MODEL FINE-TUNING" from meta which explines relation between intrinsic dimensionality and fine-tuning.

What is Intrinsic Dimensionality (ID)?

- Intrinsic Dimensionality (d) is the smallest number of trainable parameters needed to optimize a model to achieve a near-optimal solution.
- Instead of training all model parameters (millions), we train only a small subset of parameters in a low-dimensional subspace.
- This subspace is projected back into the full parameter space.

Parameterizing the Model in a Low-Dimensional Subspace

Instead of directly fine-tuning a large model with D parameters, reparameterize it using a lower-dimensional subspace of size d (where $d \ll D$).

$$\theta_D = \theta_{D0} + P(\theta_d)$$

Where

- θ_D: Full parameter space of the model (e.g., all weights of BERT, RoBERTa, etc.)
- θ_{D0}: Pre-trained model parameters
- $P(\theta_d)$: Trainable parameters in the lower-dimensional subspace (d-dimensional)
- $P : \mathbb{R}^d \to \mathbb{R}^D$: Projection function that maps the small set of parameters back to the full space.

This means that rather than directly updating θ_D, we can train a much smaller θ_d and use a projection P to map it back into the full parameter space.

Defining the Projection Function P

A random projection matrix is used to project the lower-dimensional parameters θ_d into the original parameter space.

$$\theta_D = \theta_{D0} + \theta_d M$$

where

- M is a random projection matrix that spreads the information from d-dimensional space to the full D-dimensional space.
- Instead of storing a massive $D \times d$ projection matrix explicitly, use the Fastfood Transform for efficiency.

Fastfood transform:

$$M = HG\Pi HB$$

where

- H is a Hadamard matrix
- G Diagonal matrix with normal (Gaussian) random entries
- Π is a randompermutation matrix
- B Diagonal matrix with ± 1 random entries

This structure allows us to perform projection efficiently in $O(D \log d)$ time instead of $O(Dd)$

Structure Aware Intrinsic Dimension - SAID method

The reparameterization in SAID is modified to introduce layer-wise scaling factors (λ_i):

$$\theta_{D,i} = \theta_{D0,i} + \lambda_i P(\theta_{d-m})_i$$

Optimizing in the Subspace

After defining the parameterization, train the model by optimizing only the d parameters instead of the full D parameters. The optimization is performed as follows:

- Initialize $\theta_d = 0 \rightarrow$ This means the starting point is just the original pre-trained model.
- Update θ_d using SGD.
- Project back using $P \rightarrow$ The small updates to θ_d are mapped back to θ_D via M

Results

The following figures show the evaluation accuracy on two datasets and four models across a range of dimensions d for the DID method. The horizontal lines in each figure represent the 90% solution of the respective full model.

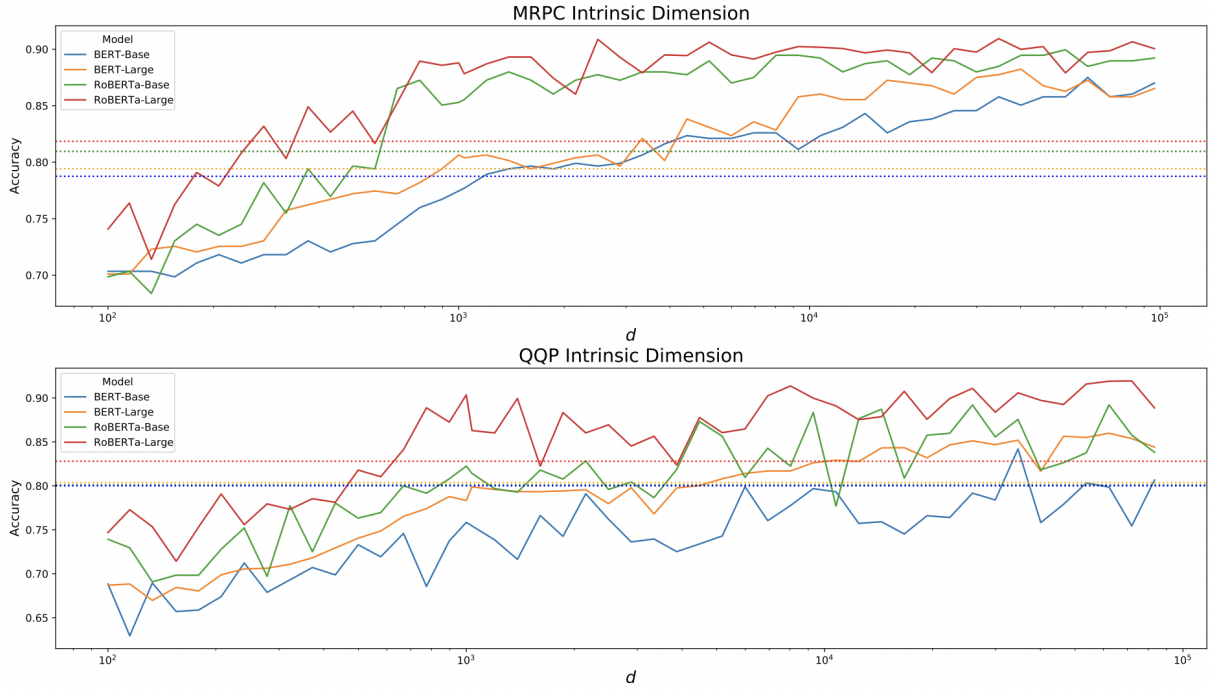


Figure 13: Intrinsic Dimensionality of models

	SAID		DID	
Model	MRPC	QQP	MRPC	QQP
BERT-Base	1608	8030	1861	9295
BERT-Large	1037	1200	2493	1389
RoBERTa-Base	896	896	1000	1389
RoBERTa-Large	**207**	**774**	322	**774**

Figure 14: Models d_{90} Intrinsic Dimensionality

Pre-Training vs Model Intrinsic Dimension

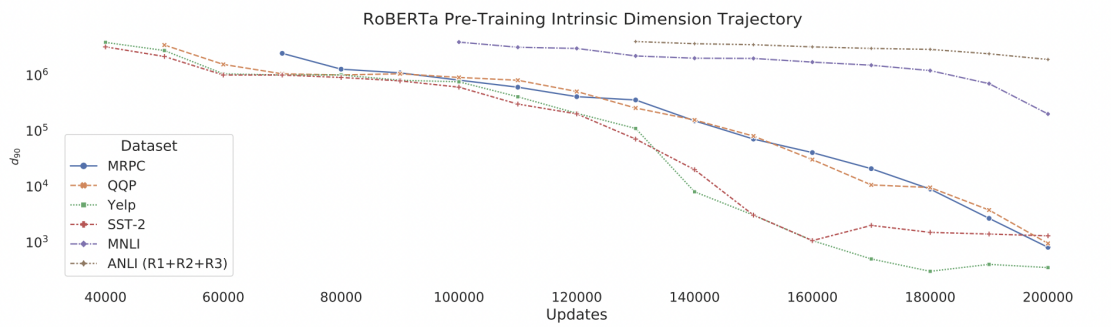


Figure 15: Pre-Training vs Model Intrinsic Dimensions

Key Takeaways:

1. Pre-training acts as a form of compression:
   - intrinsic dimensionality of LM monotonically decreases as we continue pre-training.
   - This means that the model is becoming more and more efficient in terms of the number of parameters required to learn the task.
2. Easier tasks require fewer parameters:
   - Sentiment classification (Yelp, SST-2) $\rightarrow$ small d_{90}
   - Reasoning tasks (ANLI, MNLI) $\rightarrow$ larger d_{90}

Parameter Count Vs Intrinsic Dimension

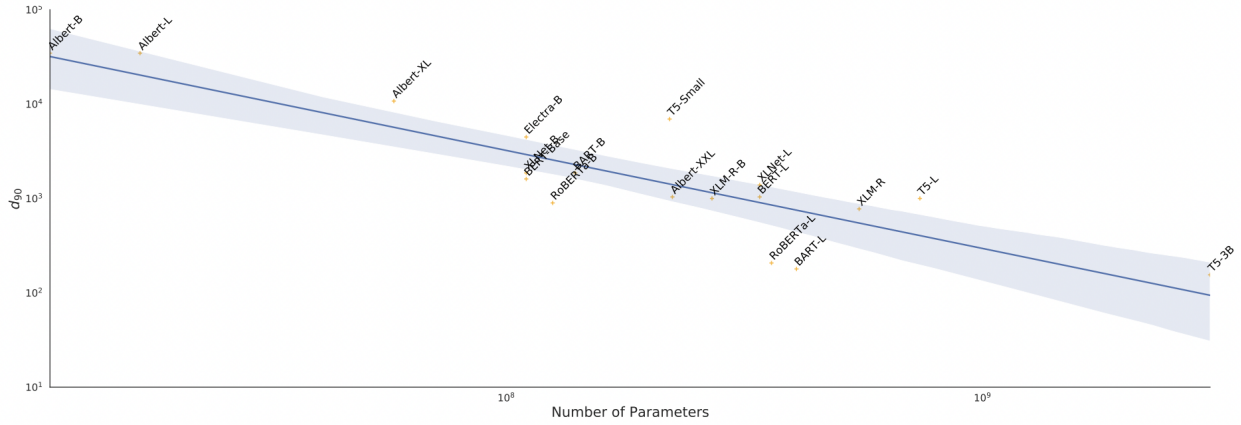


Figure 16: no of parameters in model Vs Intrinsic Dimension

Key Takeaways:

1. Larger pre-trained models have lower intrinsic dimensionality (require fewer trainable parameters).
2. There is a strong inverse correlation between model size and intrinsic dimensionality

Low-Rank Adaptation of Large Language Models (LoRA)

- Large models typically had low intrinsic dimensionality.
- Building on this insight, LoRA hypothesizes that the change in weights during model adaptation also has a low "intrinsic rank"

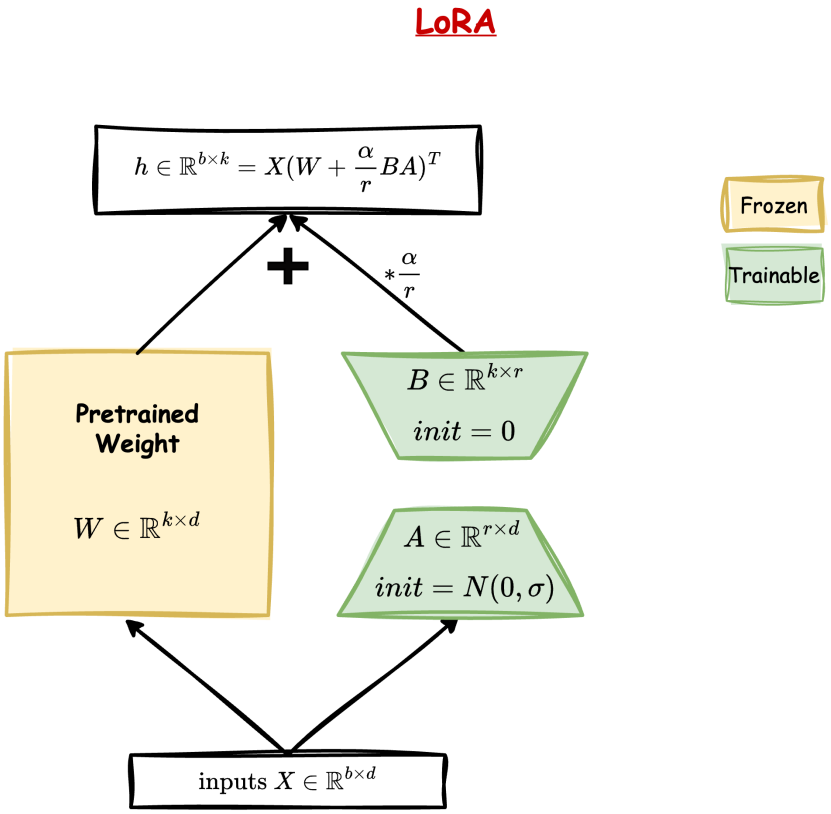


Figure 17: LoRA

Given a pre-trained model with weight matrix W_0, full fine-tuning updates the weights by adding ΔW such that the new weight matrix is:

$$W = W_0 + \Delta W \tag{1}$$

Instead of updating ΔW directly, LoRA models ΔW as a low-rank decomposition:

$$\Delta W = BA \tag{2}$$

where:

$$B \in \mathbb{R}^{k \times r} \tag{3}$$

$$A \in \mathbb{R}^{r \times d} \tag{4}$$

$$r \ll \min(k, d) \text{ (rank of decomposition)} \tag{5}$$

LoRA optimizes only B and A while keeping W_0 frozen, significantly reducing the number of trainable parameters.

Implementation Details

Initialization

- A is initialized from a Gaussian distribution: $A \sim \mathcal{N}(0, \sigma^2)$
- B is initialized to zero to ensure that $\Delta W = 0$ at the start

Scaling Factor

A hyperparameter α is introduced to control the scale of ΔW:

$$W = W_0 + \frac{\alpha}{r} BA \tag{6}$$

Inference Optimization

At inference time, LoRA merges ΔW into W_0 to avoid additional latency.

Advantages of LoRA

- **Efficiency:** Reduces the number of trainable parameters by orders of magnitude (e.g., 10,000x for GPT-3 175B)
- **No Additional Latency:** Unlike adapter layers, LoRA does not introduce inference-time latency
- **Quick Task-Switching:** Since only small matrices ΔW need to be stored, switching between different tasks is efficient

Key Parameters to tune:

- Rank r
- Scaling factor α
- Which layers to apply LoRA

Which layers to apply LoRA

	# of Trainable Parameters = 18M						
Weight Type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	2
WikiSQL (±0.5%)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI (±0.1%)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

Table 5: Validation accuracy on WikiSQL and MultiNLI after applying LoRA to different types of attention weights in GPT-3, given the same number of trainable parameters. Adapting both W_q and W_v gives the best performance overall. We find the standard deviation across random seeds to be consistent for a given dataset, which we report in the first column.

Figure 18

This suggests that even a rank of 4 or 2 captures enough information in ΔW such that it is preferable to adapt more weight matrices than adapting a single type of weights with a larger rank.

Weight Matrices	**Rank**	**Params**	**Simple AST**	**Parallel AST**	**Multiple AST**	**Parallel Multi AST**	**Irrelavance Detection**	**Avg**
W_q, W_k	16	2.18M (0.14%)	86.25	71.5	81.5	62	66.1	73.47
W_q, W_v	16	2.18M (0.14%)	86	71.5	82	61.5	67	73.6
W_q, W_k, W_v, W_o	16	4.36M (0.28%)	86.5	72.5	82.5	62	67.08	74.12
W_q, W_k, W_v, W_o W_{mlp}	16	18.46M (1.18%)	86.25	76	83	65.5	66	75.35

The experimental results compare different configurations of LoRA applied to various transformer weight matrices, all using rank 16. The analysis reveals several key findings:

1. **Optimal Configuration:** The most comprehensive configuration, applying LoRA to all matrices ($W_q, W_k, W_v, W_o, W_{mlp}$), achieved superior performance with an average score of 75.35%.
2. **Progressive Performance Improvement:** The results demonstrate a consistent improvement pattern as more weight matrices are included:
    - Basic configuration (W_q, W_k): 73.47%

- Alternative basic configuration (W_q, W_v): 73.60%
- Complete attention matrices (W_q, W_k, W_v, W_o): 74.12%
- All matrices including MLP: 75.35%

3. **Task-Specific Impact:** The inclusion of additional matrices showed particular benefits in complex tasks, notably in Parallel AST where performance improved from 72.5% to 76% when including MLP weights.

The data suggests that while expanding LoRA to more weight matrices increases the number of trainable parameters, it consistently yields better performance across all evaluated tasks.

Rank and Alpha Configuration

Alpha	8	16	32	64	128
Rank					
4	75.43	72.67	72.12	71.8	70.18
8	74.28	74.32	74.19	72.1	72.56
16	73.89	74.28	74.31	74.12	70.18
32	74.68	75.0	75.7	75.2	75.67
64	75.27	75.3	75.2	76.12	76.58

- Setting $\alpha = 2r$ appears to be a good starting point for experiments
- Even with low ranks, we are able to achive good results. Low ranks performing well as well as high ranks too.
- what will happen if we set different ranks to different layers? - this needs to be explored.

No of epochs to train?

Number of Epochs	**Score (%)**
1	75.98
2	76.58
3	75.92
4	75.91

1. The results suggest that increasing the number of epochs beyond 2 may lead to overfitting. This is evidenced by the declining performance after the peak at epoch

2. Even in experiments where function names were randomly modified, the model showed signs of overfitting to the training data, indicating that longer training periods may not necessarily yield better results.

Learning Rate Tuning

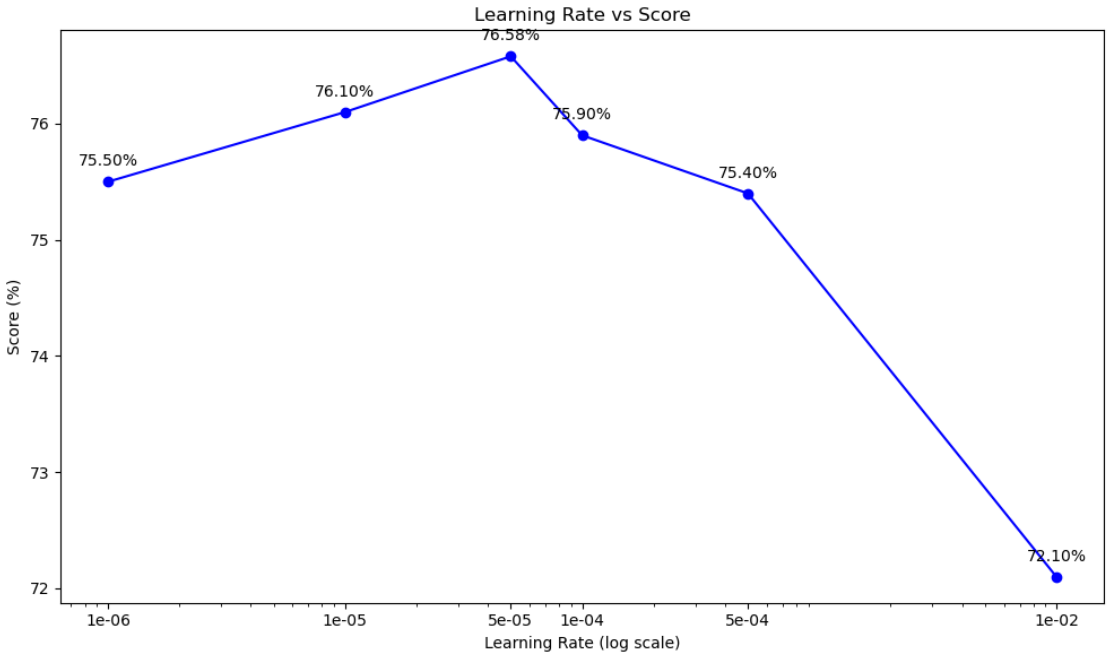


Figure 19: Learning Rate Tuning

For our task, we achieved the best results using small learning rates. After a certain point, increasing the learning rate led to a decline in performance.

Memory Usage

Technique	**Weight Matrices**	**Params**	**Peak Memory Usage**	**% memory used**	**Time taken**
FFT	-	1.5B (100%)	76.5 GB	89.1%	2hr 15m
LoRA(16)	W_q, W_k, W_v W_o	4.36M (0.28%)	63.9 GB	74.4%	2hr 11m
LoRA(16)	W_q, W_k, W_v W_o W_{mlp}	18.46M (1.18%)	64.6 GB	74.4%	2hr 23m
LoRA(64)	W_q, W_k, W_v W_o, W_{mlp}	73.85M (4.56%)	64.81 GB	75.45%	2hr 24m

Rank Stabilized LoRA

Rank-Stabilized LoRA (rsLoRA) is an improvement over the standard Low-Rank Adaptation (LoRA) method for fine-tuning large language models (LLMs). The key insight behind rsLoRA is the choice of an appropriate scaling factor for the adapter matrices to ensure stable learning dynamics.

LoRA introduces low-rank matrices $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{k \times r}$ to augment the weight matrix W in a neural network layer. The adapter is defined as:

$$\Delta W = \gamma_r BA, \tag{7}$$

where γ_r is a scaling factor. The conventional choice $\gamma_r = \alpha/r$ can cause gradient collapse at higher ranks.

To ensure stable learning, rsLoRA adopts a different scaling factor γ_r based on an analysis of variance. The key requirement for stability is that the variance of adapter outputs remains constant as r changes. This leads to the condition:

$$\mathbb{E}[||\gamma_r BAx_{\text{in}}||^2] \sim \Theta(1). \tag{8}$$

By analyzing the expected norm of BA, it can be shown that stability requires:

$$\gamma_r = \frac{\alpha}{\sqrt{r}}. \tag{9}$$

Proof of rank stability

Function and Loss

Lora updates using $f(x) = \gamma_r BAx$, where γ_r is a scalar, and B and A are matrices. The loss function is denoted as $L(f(x))$.

SGD Updates

B_n and A_n are the matrices B and A after the n-th update using SGD with learning rate η. Initially, $B_0 = 0$ and $A_0 = N(0, \sigma^2)$.

Gradients

The gradients of the loss with respect to B and A are given by:

$$\nabla_{B_n} L = \gamma_r v_n x_n^T A_n^T \tag{10}$$

$$\nabla_{A_n} L = \gamma_r B_n^T v_n x_n^T \tag{11}$$

Here, $v_n = \nabla_{f(x_n)} L(f(x_n))$ is the gradient of the loss with respect to the output of the function.

Induction for B_n and A_n

- $B_0 = 0_{d \times r}$
- $A_0 = A_0$ (initial value)
- Learning rate $= \eta$
- $v_n = \nabla f(x_n) L(f(x_n))$

First iteration (n = 1):
For B_1:

$$\begin{aligned} B_1 &= B_0 - \eta \nabla_{B_0} L \\ &= 0 - \eta(\gamma r v_0 x_0^T A_0^T) \\ &= -\eta\gamma r v_0 x_0^T A_0^T \end{aligned}$$

For A_1:

$$\begin{aligned} A_1 &= A_0 - \eta \nabla_{A_0} L \\ &= A_0 - \eta(\gamma r B_0^T v_0 x_0^T) \\ \text{Since } B_0 &= 0: \\ A_1 &= A_0 \end{aligned}$$

Second iteration (n = 2):
For B_2:

$$\begin{aligned} B_2 &= B_1 - \eta \nabla_{B_1} L \\ &= -\eta\gamma r v_0 x_0^T A_0^T - \eta(\gamma r v_1 x_1^T A_1^T) \\ \text{Since } A_1 &= A_0 : \\ B_2 &= -\eta\gamma r (v_0 x_0^T + v_1 x_1^T) A_0^T \end{aligned}$$

For A_2:

$$\begin{aligned} A_2 &= A_1 - \eta \nabla_{A_1} L \\ &= A_0 - \eta(\gamma r B_1^T v_1 x_1^T) \\ &= A_0 - \eta(\gamma r(-\eta\gamma r v_0 x_0^T A_0^T)^T v_1 x_1^T) \\ &= A_0 + \eta^2(\gamma r)^2 A_0 v_0 x_0^T v_1 x_1^T \\ &= A_0(1 + O(r(\gamma^2))) \end{aligned}$$

Pattern emerges:

- Each iteration adds a term $-\eta\gamma r v_k x_k^T A_k^T$
- Since A_k stays close to A_0, we get:

$$B_n = -\eta\gamma r \sum_{k=0}^{n-1} v_k x_k^T A_0^T + O(r(\gamma^2)) A_0^T \tag{12}$$

For A_n:

- The updates involve terms of order γ^2
- The main term A_0 remains dominant
- The correction terms are small (order γ^2)

$$A_n = A_0(1 + O(r(\gamma^2))) \tag{13}$$

This shows why A_n stays close to A_0 with small perturbations, while B_n accumulates the gradient updates in a more substantial way. The $O(r(\gamma^2))$ terms come from the higher-order interactions between updates.

Resulting Expression

Expression for $\gamma_r B_n A_n$:

$$\gamma_r B_n A_n = -\gamma_r^2 \eta \sum_{k=0}^{n-1} v_k x_k^T A_0^T A_0 + O_r(\gamma_r^3) A_0^T A_0 \tag{14}$$

This shows how the product of the updated matrices B_n and A_n scales with γ_r.

Expectation and Initialization

Expectation of Initialization:

$$E_{A_0}(A_0^T A_0) = r\sigma_A I_{d\times d} \tag{15}$$

This means that on average, the product $A_0^T A_0$ behaves like a scaled identity matrix.

Expected Value of $\gamma_r B_n A_n$:

$$E_{A_0}(\gamma_r B_n A_n) = -\gamma_r^2 r\sigma_A \eta \sum_{k=0}^{n-1} v_k x_k^T + O_r(\gamma_r^3 r) \tag{16}$$

This shows how the expected value of the product scales with γ_r.

Backward and Forward Pass

Backward Pass:

$$\nabla_{x_n} L(\gamma_r B_n A_n x_n) = -\gamma_r^2 r\sigma_A \eta \sum_{k=0}^{n-1} x_k v_k^T v_n + O_r(\gamma_r^3 r) \tag{17}$$

This indicates how the gradient scales with γ_r.

Forward Pass:

$$E_{x,A_0}((\gamma_r B_n A_n x_n)^m) = (-\gamma_r^2 r\sigma_A \eta)^m \sum_{k=0}^{n-1} v_k^m E((x_k^T x_n)^m) + O_r((\gamma_r^3 r)^m) \tag{18}$$

This shows how the output scales with γ_r.

Stability Condition

Stability Condition: For the system to remain stable as $r \to \infty$, you need:

$$\Theta_r((\gamma_r^2 r)^m) = \Theta_r(1) \tag{19}$$

This implies that γ_r should scale as:

$$\gamma_r \in \Theta_r\left(\frac{1}{\sqrt{r}}\right) \tag{20}$$

This condition ensures that the scaling factor γ_r is chosen such that the system does not become unstable or collapse as the rank r increases.

Loss Comparision with LoRA

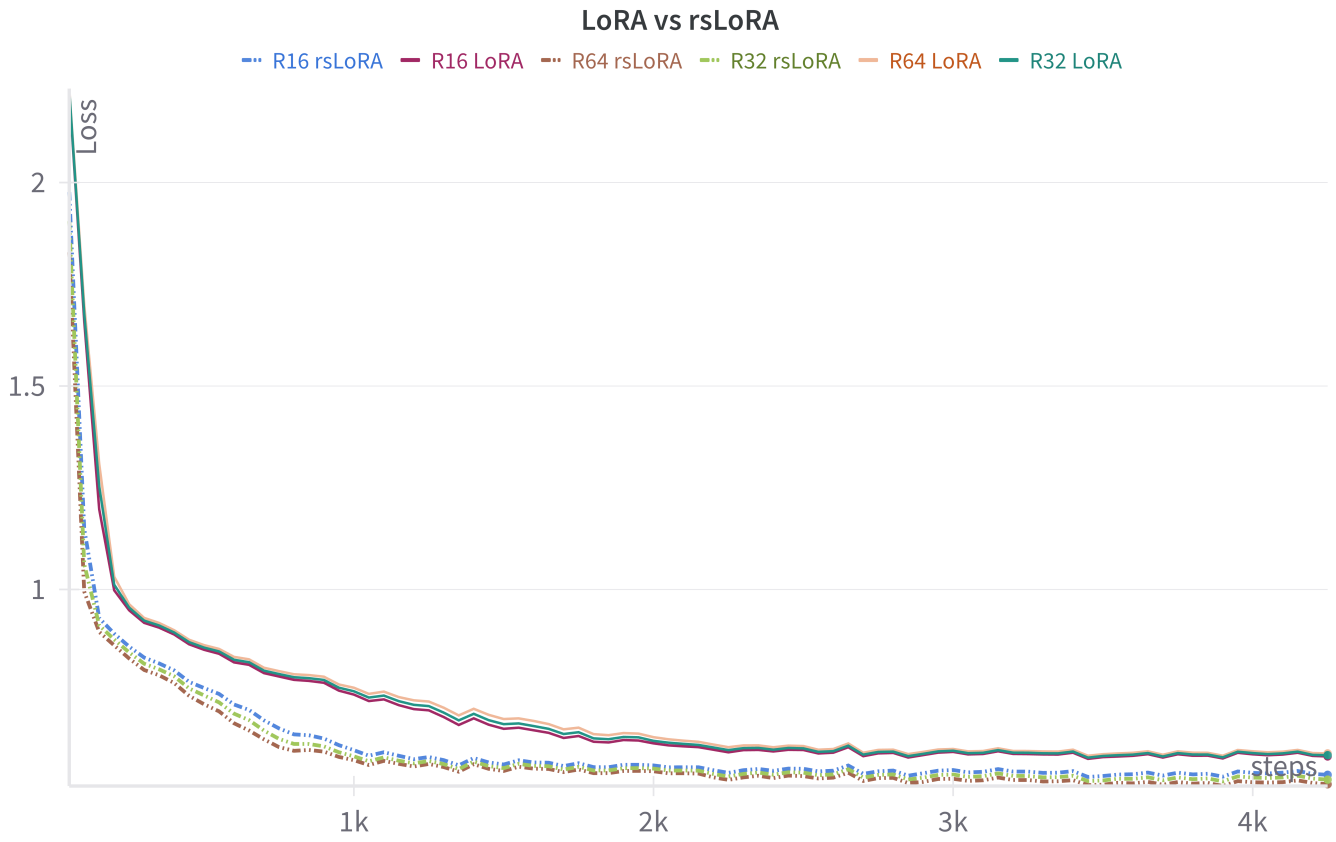


Figure 20: Loss Comparision with LoRA

We can clearly see that rsLoRA is performing better than LoRA.

Gradient Norms while training

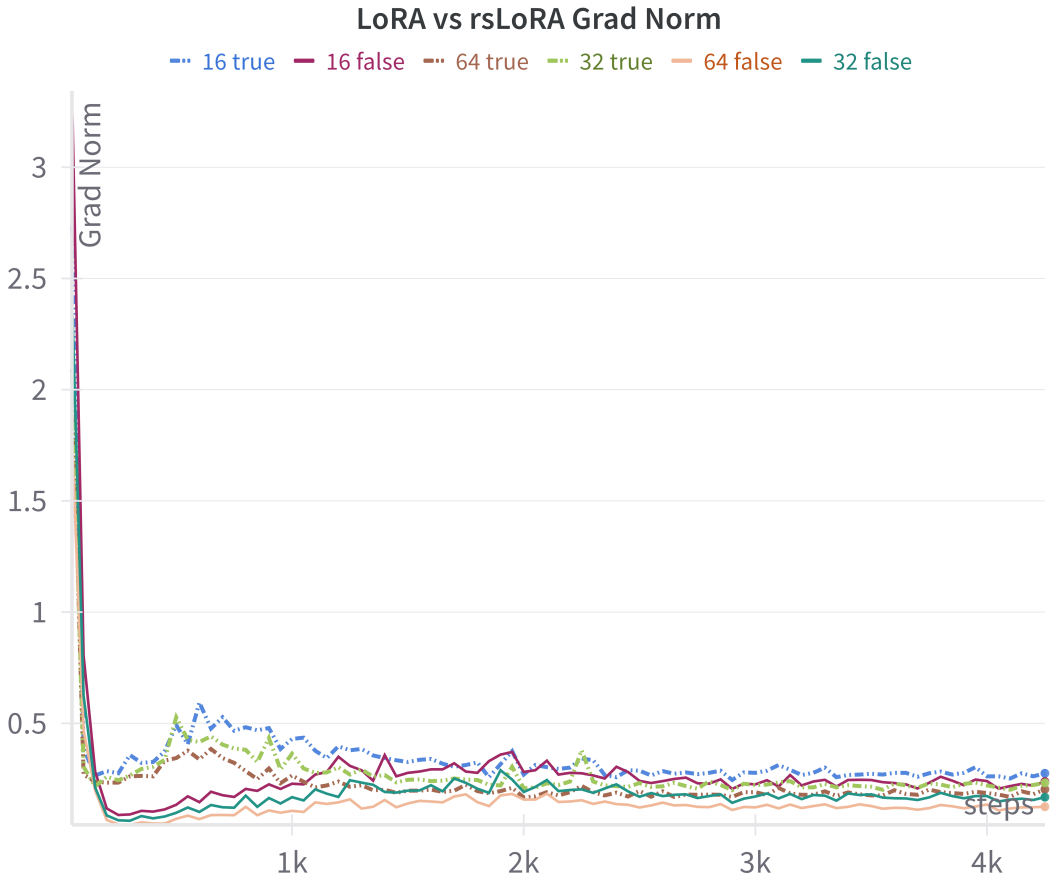


Figure 21: Gradient Norms while training

Initial gradient norms are higher for rsLoRA than LoRA.

Performance Comparison LoRA vs rsLoRA

Scaling	Rank	Simple AST	Parallel AST	Multiple AST	Parallel Multiple AST	Irrelevance Detection	avg
LoRA	16	86.25	76	83	65.5	60.83	74.32
rsLoRA	16	87.75	80	86.5	70.5	63.75	77.7
LoRA	32	85.5	78.5	83	67.5	63.75	75.65
rsLoRA	32	88.5	78.5	84	67.5	67.08	77.12
LoRA	64	85.75	75.5	86	65	63.75	75.2
rsLoRA	64	86.75	77.5	86	74.5	65.42	78.03

Table 1: Performance comparison of LoRA and rsLoRA

DORA

Weight Decomposition

$$W = m.V = ||W||_c.\frac{W}{||W||_c} \tag{21}$$

- m: Scalar magnitude of the weight vector.
- V: Vector direction of the weight vector.
- $||W||_c$: magnitude vector i.e. Vector wise norm of matrix across each column

Adapted Weights with Low-Rank Updates

$$W' = m.V' = m.(V + \Delta V) = m.\frac{W + BA}{||W + BA||_c} \tag{22}$$

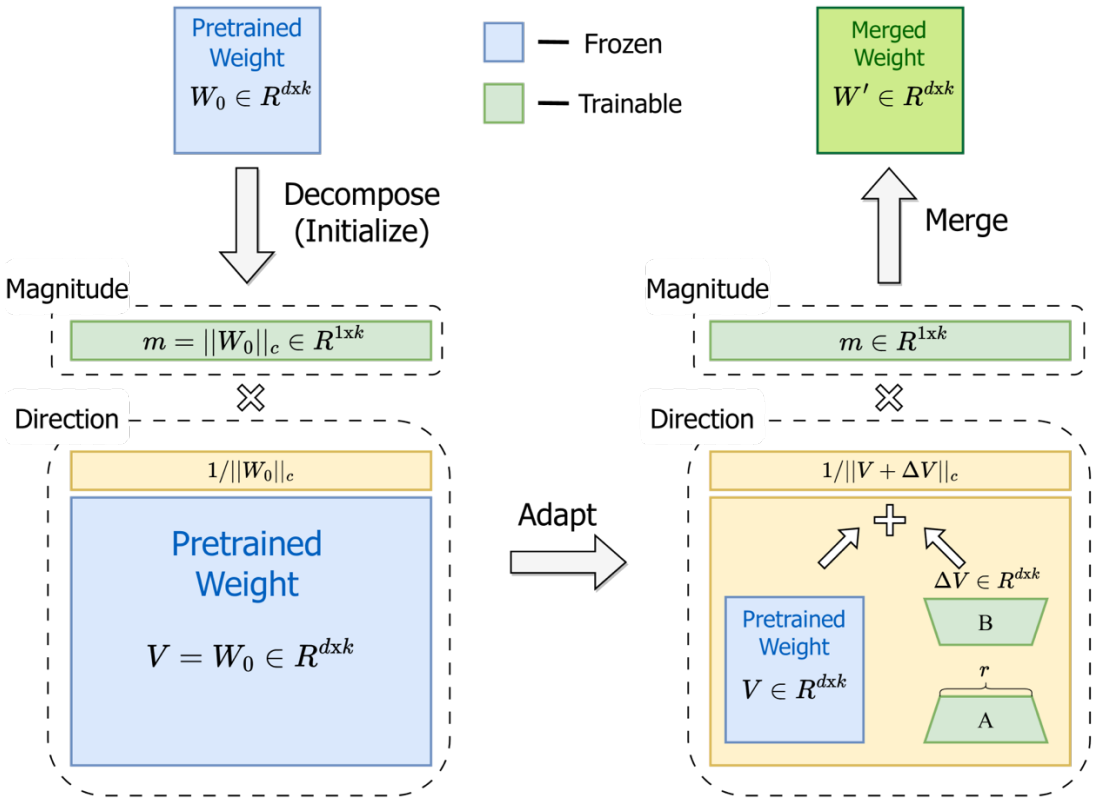


Figure 22: DoRA

Gradients

$$\nabla_{V'} L = \frac{m}{||V'||_c} * (I - \frac{V'V'^T}{||V'||_c^2}) \nabla_{W'} L \tag{23}$$

- Updates are scaled by $\frac{m}{||V'||_c}$, ensuring proportional changes in both magnitude and direction.
- The term $I - \frac{V'V'^T}{||V'||_c^2}$ projects gradients orthogonally to V', avoiding redundant updates.

Computational Efficiency in DoRA

- Memory-Efficient Gradient Modification
  - Treat $||V + \Delta V||_c$ as a constant during backpropagation

$$\nabla_{V'} L = \frac{m}{C} \nabla_{W'} L, \text{where } C = ||V'||_c \tag{24}$$

EVA - Explined Varinace Adaptation

Data-Driven Initialization and Adaptive Rank Allocation

EVA uses Singular Value Decomposition (SVD) on activation vectors from the downstream task to initialize A optimally then uses Adaptive Rank Allocation to allocate the rank to the layers based on the explained variance ratio.

Let $X \in \mathbb{R}^{b \times d}$ (mini batches with total size b) be activations be passed through the pre-trained model.

- Perform SVD on X to get U, Σ, V
- The right-singular vectors obtained by SVD represent the projection onto the principal components
- Initialize the LoRA downprojection (A) with those vectors to obtain an initialization that stores the most information of the downstream data
- Initilize B Similar to LoRA
- Allocate the rank to layers based on the explained variance ratio.

```
def eva_initialization(X_batches, rank_r, convergence_threshold):
    U, Sigma, V_new = None, None, None
    V_old = None  # Store previous V
    for batch in X_batches:
        X = calculate_activation(batch)
        U, Sigma, V_new = incremental_svd(X, U, Sigma, V_new) # Ross et al.
           (2008)
        # Check convergence of V
        if V_old is not None:
            converged = np.all(
                        np.sum(V_old[:rank_r] * V_new[:rank_r], axis=1) /
                        (np.linalg.norm(V_old[:rank_r], axis=1) * np.linalg.
                           norm(V_new[:rank_r], axis=1))
                        >= convergence_threshold
                    )
            if converged:
                break  # Stop incremental SVD if singular vectors have
                   converged
        V_old = V_new  # Update V_old for the next iteration
```

```
    # Compute explained variance ratio
    total_samples = sum(batch.shape[0] for batch in X_batches)
    # Select top-r singular vectors for initialization
    A = V_new[:rank_r, :]
    return A, total_samples
```

Adaptive Rank Allocation

- Compute explained variance ratio for each layer.
- Sort the layers based on the explained variance ratio.
- Allocate the rank to the layers based on the sorted order.

$$\text{explained\_variance\_ratio} = \frac{S^2}{\sum_{i=1}^{n} \text{col\_var}_i \times \text{n\_total\_samples}} \tag{25}$$

LoRA-Pro

Full Fine Tuning VS LoRA

Full Fine-Tuning Analysis

Taylor Series Expansion of the Loss Function:

$$L(W + dW) \simeq L(W) + \left\langle \frac{\partial L}{\partial W}, dW \right\rangle_F \tag{26}$$

$$dL \simeq \left\langle \frac{\partial L}{\partial W}, dW \right\rangle_F \tag{27}$$

- dL is the change in loss
- dW is the change in weights
- L is loss
- $\langle \cdot, \cdot \rangle_F$ is the Frobenius inner product.

To minimize the loss, we use gradient descent:

$$dW = -\eta \frac{\partial L}{\partial W} \tag{28}$$

$$\triangleq -g \text{ (omitting the learning rate for simplicity)} \tag{29}$$

Substituting this into the expression for dL

$$dL = \langle g, -g \rangle_F \tag{30}$$

$$= -||g||_F^2 \tag{31}$$

Thus, - the loss always decreases $dL \leq 0$ - Full fine-tuning uses the entire gradient g for optimization.

LoRA Analysis

LoRA assumes that the model parameters W can be decomposed as

$$W = W_0 + \Delta W = W_0 + sBA \tag{32}$$

- W_0 is the pre-trained weights

- ΔW is the low-rank update
- A is the downprojection matrix
- B is the upprojection matrix
- s is the scaling factor $= \frac{\alpha}{r}$

Same as Full Fine-Tuning, we use gradient descent to minimize the loss:

$$dL \simeq \left\langle \frac{\partial L}{\partial W}, dW \right\rangle_F \tag{33}$$

Since $W = W_0 + sBA$, its differential dW depends on dA and dB:

$$dW = \frac{\partial W}{\partial A} dA + \frac{\partial W}{\partial B} dB \tag{34}$$

substituting dW into the expression for dL

$$dL \simeq \left\langle \frac{\partial L}{\partial W}, \frac{\partial W}{\partial A} dA + \frac{\partial W}{\partial B} dB \right\rangle_F \tag{35}$$

$$dL \simeq \left\langle \frac{\partial L}{\partial W}, \frac{\partial W}{\partial A} dA \right\rangle_F + \left\langle \frac{\partial L}{\partial W}, \frac{\partial W}{\partial B} dB \right\rangle_F \tag{36}$$

$$\tag{37}$$

Let's take the first term:

$$\left\langle \frac{\partial L}{\partial W}, \frac{\partial W}{\partial A} dA \right\rangle_F = \left\langle \frac{\partial L}{\partial W}, sBdA \right\rangle_F \tag{38}$$

$$= s \cdot \left\langle \frac{\partial L}{\partial W}, BdA \right\rangle_F \tag{39}$$

$$= s \cdot Tr\left(\left(\frac{\partial L}{\partial W} \right)^T BdA \right) \tag{40}$$

$$= s \cdot Tr\left(\left(B \frac{\partial L}{\partial W} \right)^T dA \right) \tag{41}$$

$$= \cdot Tr\left(\left(\frac{\partial L}{\partial A} \right)^T dA \right) \text{ (since } \frac{\partial L}{\partial A} = \frac{\partial L}{\partial W} . \frac{\partial W}{\partial A} = \frac{\partial L}{\partial W} . sB) \tag{42}$$

$$= \left\langle \frac{\partial L}{\partial A}, dA \right\rangle_F \tag{43}$$

It will be the same for the second term.
So,

$$dL \simeq \left\langle \frac{\partial L}{\partial A}, dA \right\rangle_F + \left\langle \frac{\partial L}{\partial B}, dB \right\rangle_F \tag{44}$$

In gradient descent, we update the parameters A and B using the gradients:

$$dA = -\eta \frac{\partial L}{\partial A} = -g_{loraA} \tag{45}$$

$$dB = -\eta \frac{\partial L}{\partial B} = -g_{loraB} \tag{46}$$

This gives

$$dL \simeq \langle g_{loraA}, -g_{loraA} \rangle_F + \langle g_{loraB}, -g_{loraB} \rangle_F \tag{47}$$

$$= -||g_{loraA}||_F^2 - ||g_{loraB}||_F^2 \tag{48}$$

Connection between LoRA and Full Fine-Tuning

To understand how LoRA relates to full fine-tuning, let's rewrite the LoRA update in terms of W.

$$dW = \frac{\partial W}{\partial A} dA + \frac{\partial W}{\partial B} dB \tag{49}$$

$$= -sBg_{loraA} - sg_{loraB}A \tag{50}$$

LoRA-SB

Lora SB is a varient of LoRA that uses 3 low rank matrices other than two and updates only one of them.

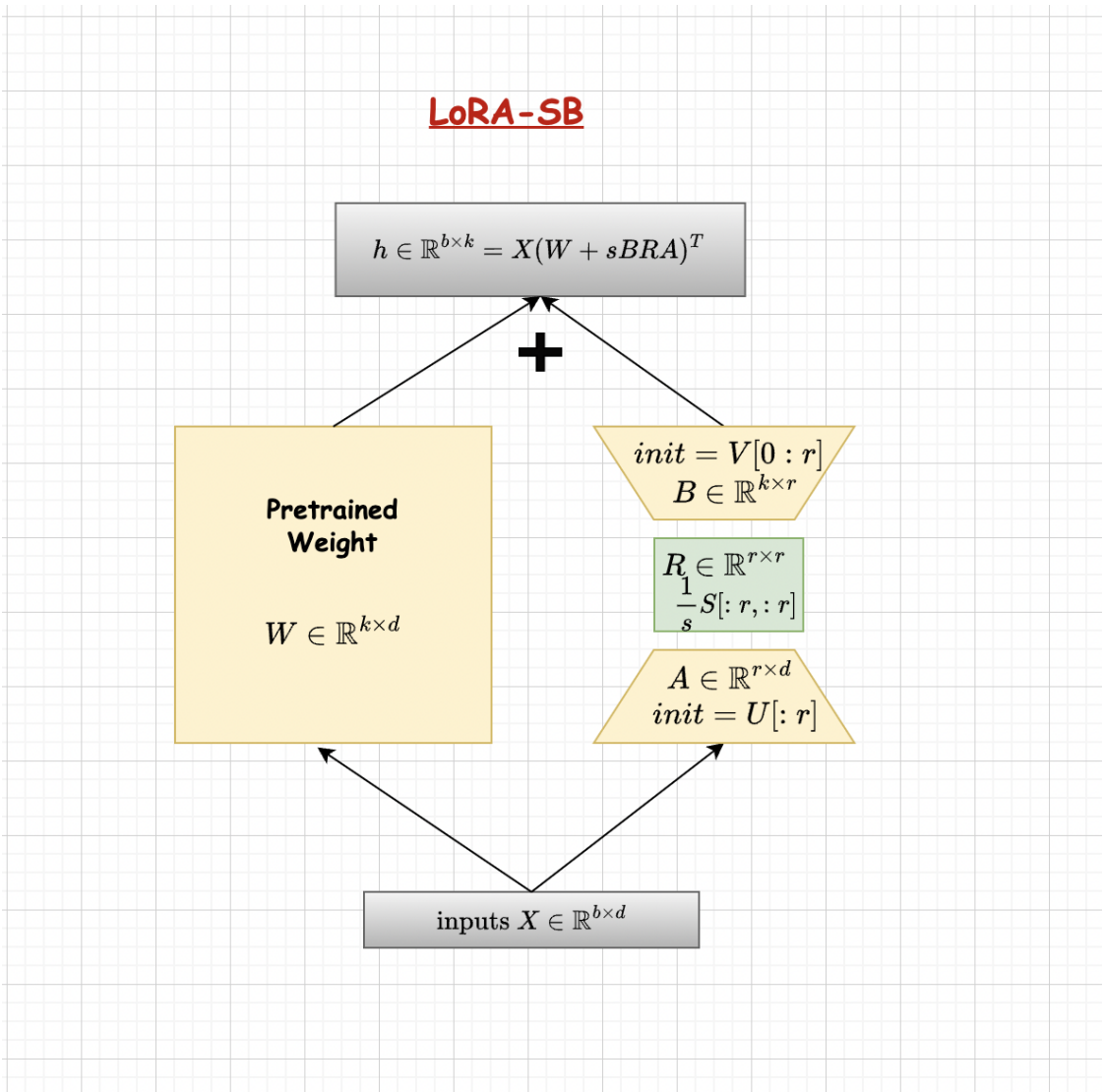


Figure 23: LoRA-SB

$$W = W_0 + \Delta W = W_0 + sBRA \tag{51}$$

- W_0 is the pre-trained weights $\in \mathbb{R}^{k \times d}$
- A is the downprojection matrix $\in \mathbb{R}^{r \times d}$
- B is the upprojection matrix $\in \mathbb{R}^{k \times r}$

- R is the rank matrix $\in \mathbb{R}^{r \times r}$
- s is the scaling factor $= \frac{\alpha}{r}$

Gradient Analysis of LoRA-SB vs LoRA

Lets take the gradient of the loss function with respect to the weights in Full Fine-Tuning:

$$g = \frac{\partial L}{\partial W} \in \mathbb{R}^{k \times d} \tag{52}$$

Lets calculate equivalent gradients for LoRA-SB:

$$\widetilde{g} = dW = \frac{\partial W}{\partial R} dR \tag{53}$$
$$= sBA.g^R \tag{54}$$
$$= sBg^R A \tag{55}$$

- This equivalent gradient describes the virtual gradient of matrix W in LoRA-SB optimization process, despite W not being directly trainable.
- This gradient determines how updates to R affect the overall weight matrix.
- main goal is - minimize the discrepancy between the equivalent gradient $\widetilde{g}$ and the full gradient g so that the updates may yiled similar results as full fine-tuning.

Lets calculate Exact gradient it updates for LoRA-SB:

$$g^R_{lora-sb} = \frac{\partial L}{\partial R} \tag{56}$$
$$= \frac{\partial L}{\partial W} . \frac{\partial W}{\partial R} \tag{57}$$
$$= \frac{\partial L}{\partial W} . \frac{\partial W}{\partial X} . \frac{\partial X}{\partial R} \text{ (where } W = sBX, X = RA) \tag{58}$$
$$= sB^T g . \frac{\partial X}{\partial R} \tag{59}$$
$$= sB^T g A^T \tag{60}$$

Leveraging this relationship, we can formulate our objective to minimize the distance between the equivalent gradient and the full gradient

Let's break down the gradient analysis step by step:

- During LoRA fine-tuning, we cannot directly access the full gradient g
- Instead, we need to:
  - Find the ideal gradient g^R with respect to matrix R
  - Use this to determine the optimal approximation $\widetilde{g}$

- We must express $\widetilde{g}$ in terms of the gradient that is actually available during training ($g^R_{lora-sb}$)

So our optimization objective is to minimize the distance between $\widetilde{g}$ and g where $\widetilde{g} = sBg^RA$

$$\underset{g^R}{argmin}||\widetilde{g} - g||_F^2 \tag{61}$$

by making the differentiation to zero

$$2(\widetilde{g} - g)\frac{\partial \widetilde{g}}{\partial g^R} = 0 \tag{62}$$

$$2(sBg^RA - g)\frac{\partial (sBg^RA)}{\partial g^R} = 0 \tag{63}$$

$$2sB^T(sBg^RA - g)A^T = 0 \tag{64}$$

$$B^TsBg^RAA^T = B^TgA^T \tag{65}$$

$$B^TsBg^RAA^T = \frac{g^R_{lora-sb}}{s} \tag{66}$$

$$B^TBg^RAA^T = \frac{g^R_{lora-sb}}{s^2} \tag{67}$$

$$g^R = \frac{1}{s^2}\left(B^TB\right)^{-1} g^R_{lora-sb}\left(AA^T\right)^{-1} \tag{68}$$

$$\tag{69}$$

Initialization

- The initial gradient steps are particularly informative, as they indicate the direction of desired adaptation.
- By leveraging this insight, we can use the first update step from full fine-tuning to initialize the low-rank matrices.
  - it ensures the low-rank space captures the most relevant subspace for the target task rather than relying on pre-trained weight properties
  - since A and B matrices will remain fixed in LoRA-XS, initializing them to span the subspace of early adaptation increases the likelihood of capturing useful updates throughout training.

Initialization of A, B and R:

- Get the average gradient from subset of data.

$$\Delta W_{avg} = -\eta \text{ sign}\left(\sum_{i=0}^{n\leq|X|} \nabla_W L\left(W_0, x_i\right)\right) \tag{70}$$

- Decompose the gradient into U, S, V

$$\Delta W_{avg} = USV^T \tag{71}$$

- Initialize A, B and R using top r vectors

$$A = V[:r, :] \tag{72}$$
$$B = U[:, :r] \tag{73}$$
$$R = S[:r, :r] \tag{74}$$

If we initlize A, B using this, $\Delta\left(B_{init}R_{init}A_{init}\right) \simeq \Delta W$ will be the best possible approximation of the full gradient.

Lets see how this initialization works:

Change in weights for LoRA-SB:

$$\begin{aligned}
\Delta W_{lora-sb} &= B\Delta RA && (75)\\
&= B\left(-\eta\nabla_R L\left(R\right)\right)A && (76)\\
&= -\eta B\nabla_R L\left(R\right)A && (77)\\
&= -\eta B\left(B^T\nabla_W L\left(W_0\right)A^T\right)A && (78)\\
& && (79)
\end{aligned}$$

Full Fine-Tuning update:

$$\Delta W_{FFT} = -\eta\nabla_W L\left(W_0\right) \tag{80}$$

The problem becomes:

$$\underset{B_{init},A_{init}}{min} ||B\left(B^T\nabla_W L\left(W_0\right)A^T\right)A - \nabla_W L\left(W_0\right)||_F \tag{81}$$

lets take first term and expand it by adding initlization:

$$\begin{aligned}
B\left(B^T\nabla_W L\left(W_0\right)A^T\right)A &= B_{init}B_{init}^T\nabla_W L\left(W_0\right)A_{init}^T A_{init} && (82)\\
&= B_{init}B_{init}^T\left(USV^T\right)A_{init}^T A_{init} && (83)\\
&= U[:, :r]U[:, :r]^T USV^T V[:r, :]^T V[:r, :] && (84)\\
&= \sum_{i=1}^{r}\sigma_i u_i v_i^T && (85)\\
& && (86)
\end{aligned}$$

our difference becomes:

$$||\sum_{i=1}^{r}\sigma_i u_i v_i^T - \sum_{i=1}^{k}\sigma_i u_i v_i^T||_F = \sqrt{\sum_{i=r+1}^{k}\sigma_i^2} \tag{87}$$

This is the best possible approximation.
Even Ecart-Young theorem states that For any matrix $M \in \mathbb{R}^{m\times n}$ the rank-r matrix M_r^* that

minimizes the Frobenius norm of the difference:

$$M_r^* = \underset{M', rank(M') \leq r}{argmin} ||M - M'||_F = \sum_{i=1}^{r} \sigma_i u_i v_i^T \tag{88}$$

Initlization Impact - Simplified Gradient Optimization

so based on initialization, our gradient update becomes:

$$g^R = \frac{1}{s^2} \left(B^T B\right)^{-1} g^R_{lora-sb} \left(AA^T\right)^{-1} \tag{89}$$

$$= \frac{1}{s^2} g^R_{lora-sb} \tag{90}$$

Hyperparameter(Scaling Factor) independence

based on $\widetilde{g} = sBg^R A$ and $g^R = \frac{1}{s^2} \left(B^T B\right)^{-1} g^R_{lora-sb} \left(AA^T\right)^{-1}$ and $g^R_{lora-sb} = sB^T g A^T$, $\widetilde{g}$ is s independent.