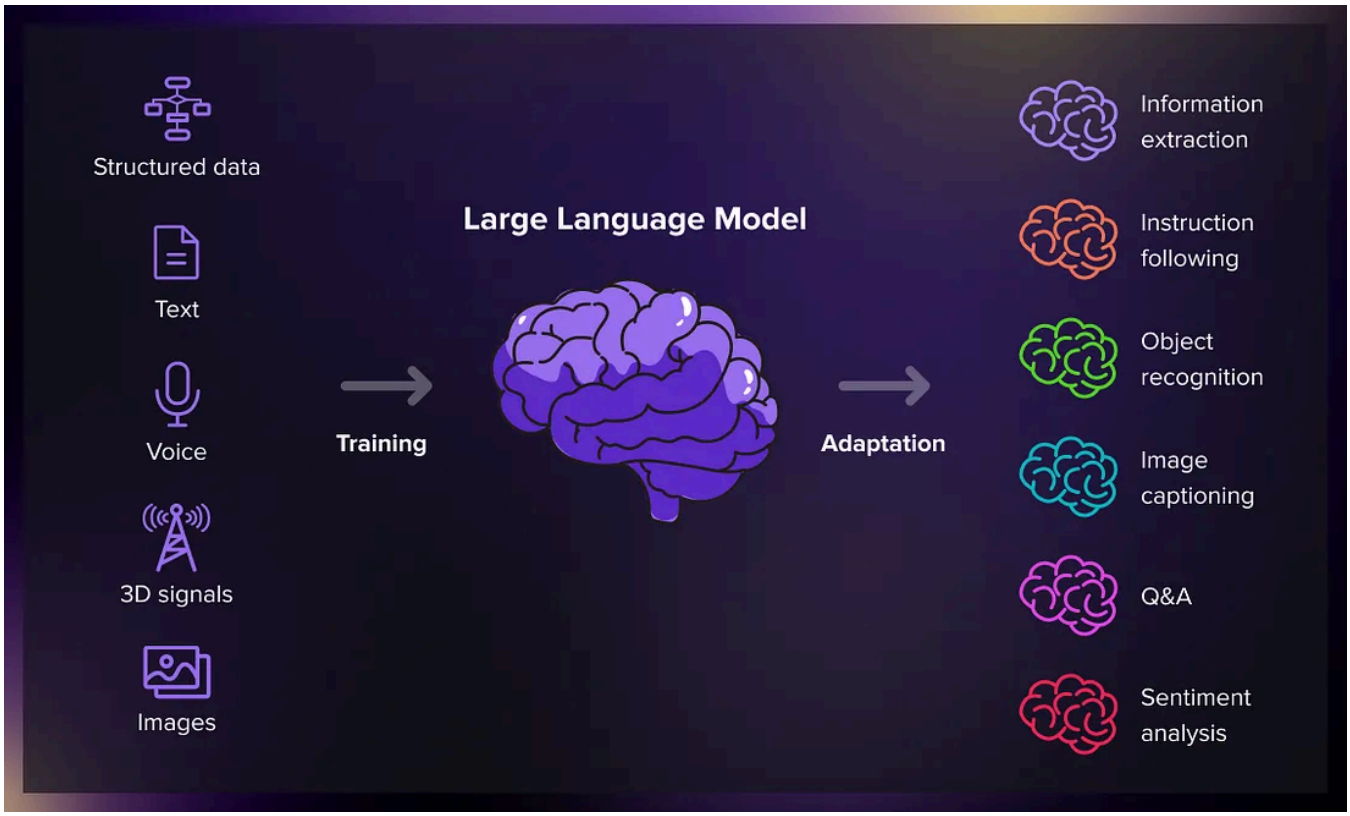


Build your Own Large Language Model

Customizing AI for your Brand: A Deep Dive into LLM Fine-Tuning

Priyam Pal
34 min read · Just now

A Detailed Roadmap of Fine-Tuning Large Language Models for Specific Task

The AI Revolution: Are Large Language Models the Future or Just Another Hype?

Imagine a world where machines don't just process commands but understand, converse, and even reason like humans. It's no longer science fiction — it's

happening right now. **Large Language Models (LLMs)** like *GPT, Gemini, and LLaMA* are reshaping industries, from healthcare and law to customer service and creative writing.

> *But here's the problem: these models, despite their intelligence, often fail at specific tasks. They generate impressive text but lack accuracy in critical areas.*
> *This raises an urgent question — How do we make AI not just powerful, but reliable and adapt to our needs?*

That's why this article exists. Instead of treating LLMs as black boxes, we'll uncover the techniques that fine-tune them into precise, task-specific tools. We'll explore 10 fine-tuning methods, including ***RLHF, Constitutional-AI, RAG, LoRA, QLoRA***, etc along with detailed mathematical representations. These techniques enhance LLMs, making them more adaptable and reliable for specific tasks. This article dives deep into how fine-tuning enables LLMs to perform specific tasks with accuracy and reliability.

Introduction to Large Language Models (LLM):

Modeling human language at scale has evolved significantly, driven by advances in computational power, datasets, and algorithms. Early language models predicted individual words, but modern large language models (LLMs) can generate coherent sentences, paragraphs, or even multi-page documents with remarkable accuracy. The rapid progress in LLMs stems from exponential growth in computing resources, memory, and training data size. Improved techniques for handling longer text sequences have also enabled these models to better capture context and nuance. This technological leap has ushered in a new era of AI-driven language understanding and interaction.

Why do LLMs often fail for Specific Tasks?

1. ***Lack of Domain Expertise:*** LLMs struggle with specialized fields like law or medicine due to insufficient training on domain-specific terminology and concepts.

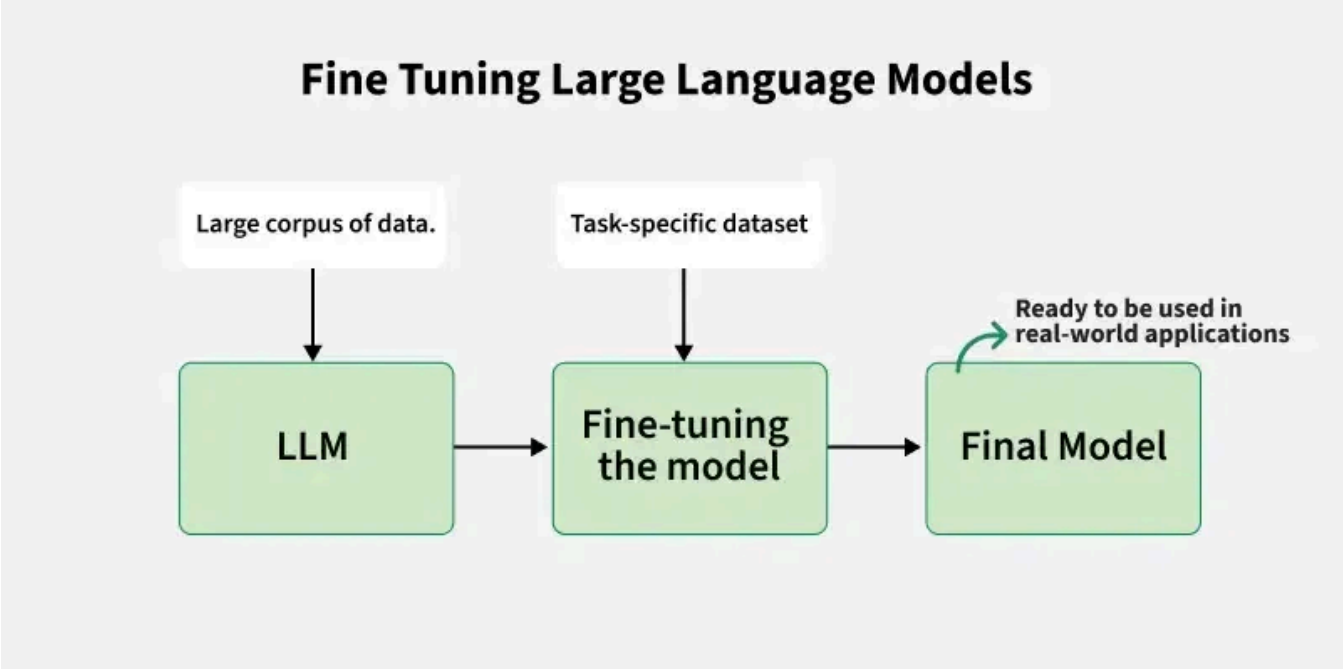
2. *Inconsistent Outputs:* Performance varies across tasks, leading to unreliable results when handling specific use cases or industries.
3. *Generic Responses:* Broad training often results in overly general answers that lack the depth or nuance required for tailored tasks.
4. *Failure to Match Writing Styles:* LLMs cannot naturally adapt to company-specific tones, branding, or unique communication guidelines.
5. *Generalization vs. Specialization Tradeoff:* Designed for wide applicability, LLMs sacrifice precision and accuracy in niche or highly specific contexts.

Need for Task-Specific Adaptation:

1. *Domain-Specific Knowledge:* LLMs need to be trained on specialized datasets to understand industry-specific terminology and concepts, ensuring accurate and relevant outputs for fields like healthcare, law, or finance.
2. *Brand Voice Alignment:* Businesses require models that reflect their unique tone, style, and identity to maintain consistency in communication and strengthen brand recognition.
3. *Context-Specific Accuracy:* Adapting LLMs ensures they provide precise and reliable responses tailored to the specific needs of a task, improving performance in specialized scenarios.
4. *Consistency Across Channels:* Customized LLMs help organizations maintain uniformity in messaging and quality across various platforms, such as email, chatbots, or marketing materials.
5. *Data Privacy and Security:* Task-specific adaptation allows businesses to safeguard sensitive or proprietary information by fine-tuning models on internal data without exposing confidential details to broader systems.

What is LLM Fine-Tuning?

LLM Fine-Tuning is the process of adapting a pre-trained language model to perform better in a specific domain or task. By training the model on a specialized dataset, organizations can transform generic AI models into precise tools tailored to their unique requirements, enhancing performance while maintaining the model's core language understanding.



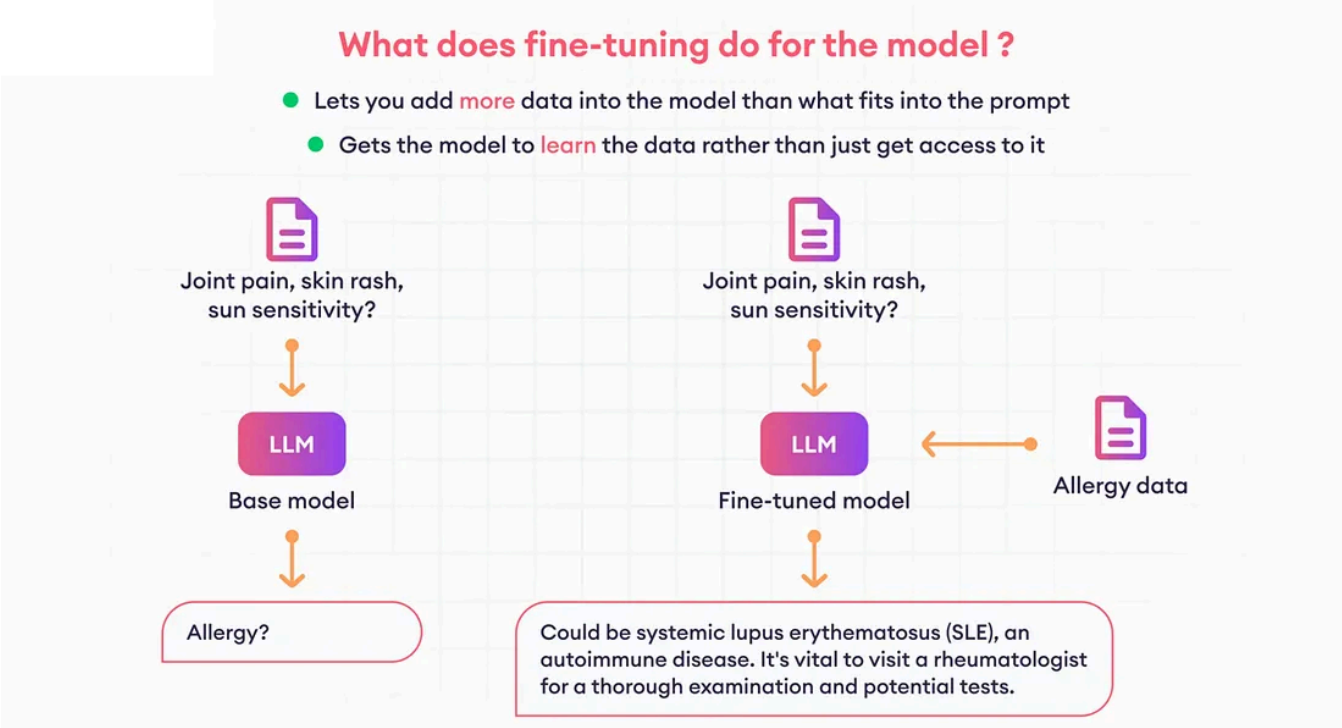
How Fine-Tuning helps for Specialized Tasks?

Modern Large Language Models (LLMs) — such as GPT (OpenAI), PaLM (Google DeepMind), LlaMA (Meta), Claude (Anthropic), and Gemini (Google DeepMind) — are built on deep neural networks and trained on massive amounts of text data. However, simply pretraining a model on large-scale datasets is not enough to make it safe, reliable and useful for real-world applications.

To enhance their capabilities, align then with human values and improve their accuracy, efficiency and controllability, researchers have developed specialized training techniques. These techniques fine-tune, adapt, and guide modes to perform better in specific tasks, follow human instruction more accurately, and reduce biases, hallucinations, and unsafe outputs. Some of these Techniques are:

1. *Improving factual accuracy and reasoning* in LLMs
2. *Reducing Biases and ethical risks* in generated responses

3. *Enhancing alignment with human intent* and preferences
4. *Boosting efficiency and performance* in specific NLP Tasks
5. *Making modes more interpretable and controllable* in Real-World applications



Need for Large Language Model (LLM) Fine-Tuning

Different Fine-Tuning Methods for LLM:

1. Instruction Tuning in Large Language Models (LLMs):

Instruction Tuning is a *fine-tuning* technique where *Large Language Models (LLMs)* are trained to ***follow explicit human instructions*** more effectively. Unlike standard language modeling, which focuses on ***predicting the next word based on context***, instruction tuning helps models generalize across task types by learning from ***explicitly provided task instructions.*** This technique ***enhances LLM adaptability***, making them more useful for real-world applications by improving their ability to ***understand, follow, and execute a wide range of tasks*** without requiring retraining

-> Mathematical and Conceptual Foundation:

Formally, a **pretrained LLM** learns a function:

$$P(Y|X;\theta)$$

where:

- X represents the **input** (e.g., a sentence or a prompt).
- Y is the **desired output** (e.g., an answer, a completion, or a classification).
- θ are the **model parameters** learned during pretraining.

During **instruction tuning**, an additional instruction signal I is introduced:

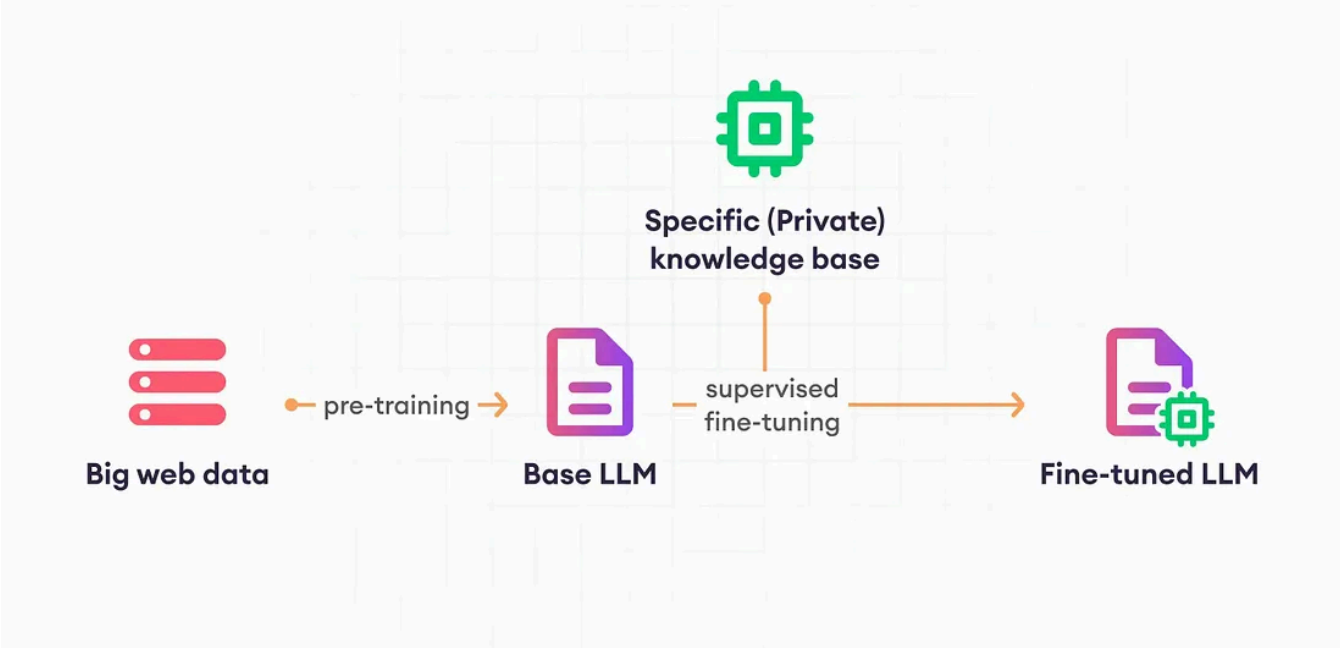
$$P(Y|X,I;\theta')$$

where:

- I is the **explicit instruction** (e.g., "Summarize the following text: ...").
- θ' represents the **new set of fine-tuned parameters** that incorporate instruction-based training.

2. Supervised Fine-Tuning : Enhancing LLM Performance with Labeled Data

Supervised Fine-Tuning is a training technique where a *pretrained Large Language Model (LLM)* is further trained using ***task-specific Labeled Datasets*** to improve its performance on specialized tasks. This technique aligns the model's response with desired outputs by optimizing it on e***xplicitly labeled input-output pairs***



Flow Diagram of Supervised Fine-Tuning Process

-> Why Supervised Fine-Tuning Important?

1. ***Improves Accuracy*** — The Model learns from human-labeled data, reducing incorrect or vague responses
2. ***Aligns with Domain-Specific Knowledge***—Helps LLMs specialize in technical, medical, legal or scientific fields
3. ***Reduces Hallucinations*** — Fine-Tuning discourages incorrect responses and enforces factual consistency
4. ***Customizes Model Behavior*** — Models can be tuned for specific tones, styles and decision-making approaches

Mathematical Formulations:

Let X be the input data and Y be the labeled outputs. The model learns a function $f(X;\theta)$ where θ represents the model's parameters. The loss function L is minimized as follows:

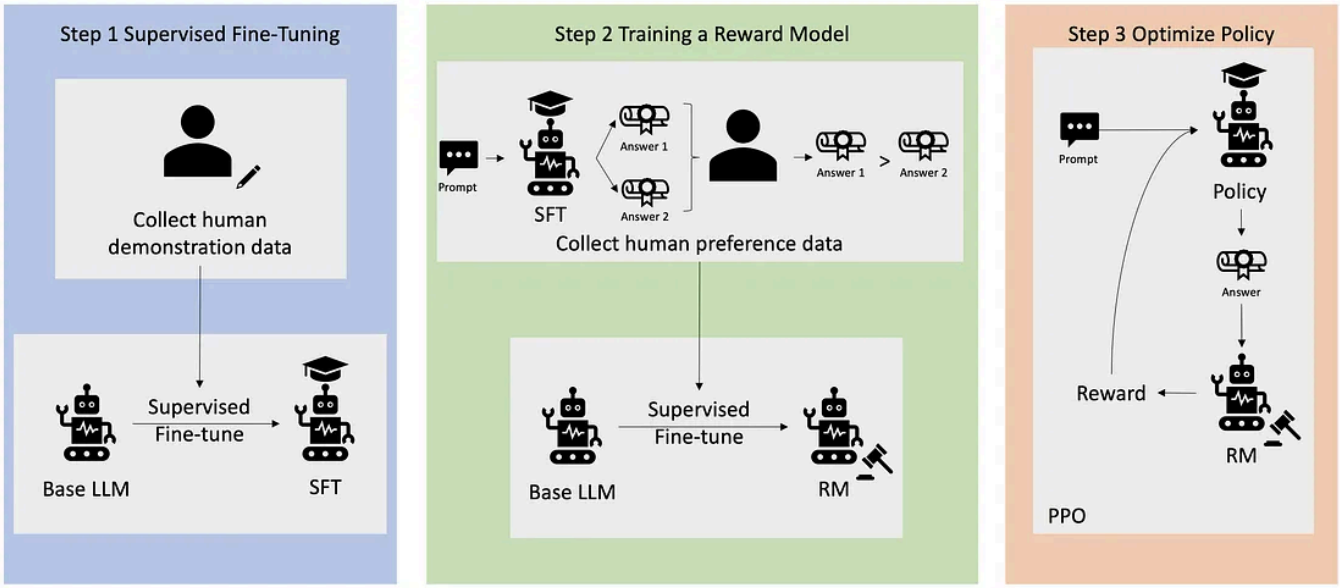
$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} L(f(X_i;\theta), Y_i)$$

where:

- L is typically **Cross-Entropy Loss** for classification or **Mean Squared Error (MSE)** for regression.
- The optimization process updates θ using **gradient-based methods** like **Adam or RMSprop**.

3. Reinforcement Learning from Human Feedback (RLHF) in LLMs

Reinforcement Learning from Human Feedback (RLHF) is a training technique used to align Large Language Models (LLMs) with human preferences. It enhances a model's ability to generate ***useful, safe and aligned*** responses by using ***human feedback*** as guiding signal. Instead of relying solely on traditional supervised Learning, RLHF introduces ***Reinforcement Learning*** to optimize responses based on human-defined quality measures



Different Stages of Reinforcement Learning Human Feedback (RLHF)

-> Mathematical and Conceptual Foundations:

RLHF consists of three main stages:

1. **Supervised Fine-Tuning (SFT)**
   - A **pre-trained language model** is fine-tuned using a **labeled dataset** where human annotators provide **high-quality responses** to different prompts.
   - The model learns a function:

$$P(Y|X;\theta)$$

   where X is the input (prompt), Y is the expected output, and θ represents the model parameters.
2. **Reward Model Training**
   - Human annotators rank multiple outputs generated by the model for a given prompt.
   - A **reward model (RM)** is trained to **predict a numerical reward score** based on these rankings, learning a function:

$$R(X,Y;\phi)$$

   where R is the reward function, X is the input, Y is the model output, and ϕ represents the reward model's parameters.

3. **Reinforcement Learning with Proximal Policy Optimization (PPO)**
   - The LLM is **further fine-tuned** using reinforcement learning, where it **generates responses, receives a reward from the RM, and updates its parameters** to maximize the expected reward.
   - The optimization follows the **policy gradient approach**, updating the model based on:

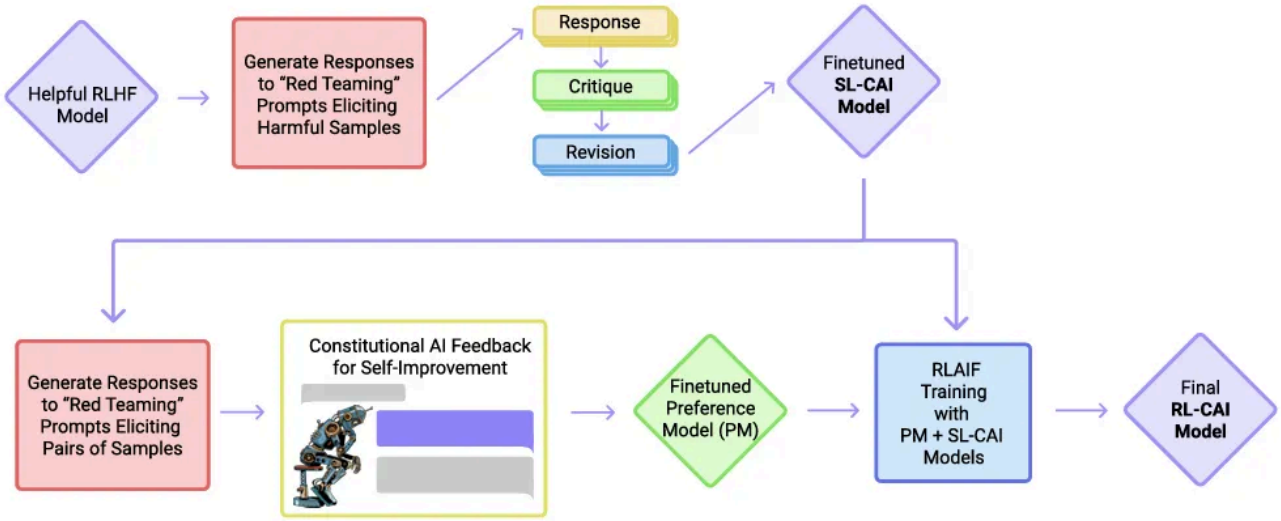
$$\theta' = \theta + \alpha \nabla_\theta \mathbb{E}[R(X, Y; \phi)]$$

   where α is the learning rate, and $\mathbb{E}[R]$ is the expected reward.

4. Constitutional AI — A Principle-Driven Approach to Training LLMs

Constitutional AI (CAI) is a training technique designed to make LLMs more **aligned with ethical principles, human values, and safety guidelines.** Instead of relying solely on human feedback (Like RLHF), CAI integrates a set of ***predefined rules or "Constitutional Principles" into the Training Process*** to guide the model's behavior.

This method was introduced by ***Anthropic***, an AI Research Company, to create models that are more **helpful, honest, and harmless.** CAI reduces the need for extensive human moderation and aims to make AI systems more **transparent and self-governing.**



Basic Steps of Constitutional AI (CAI)

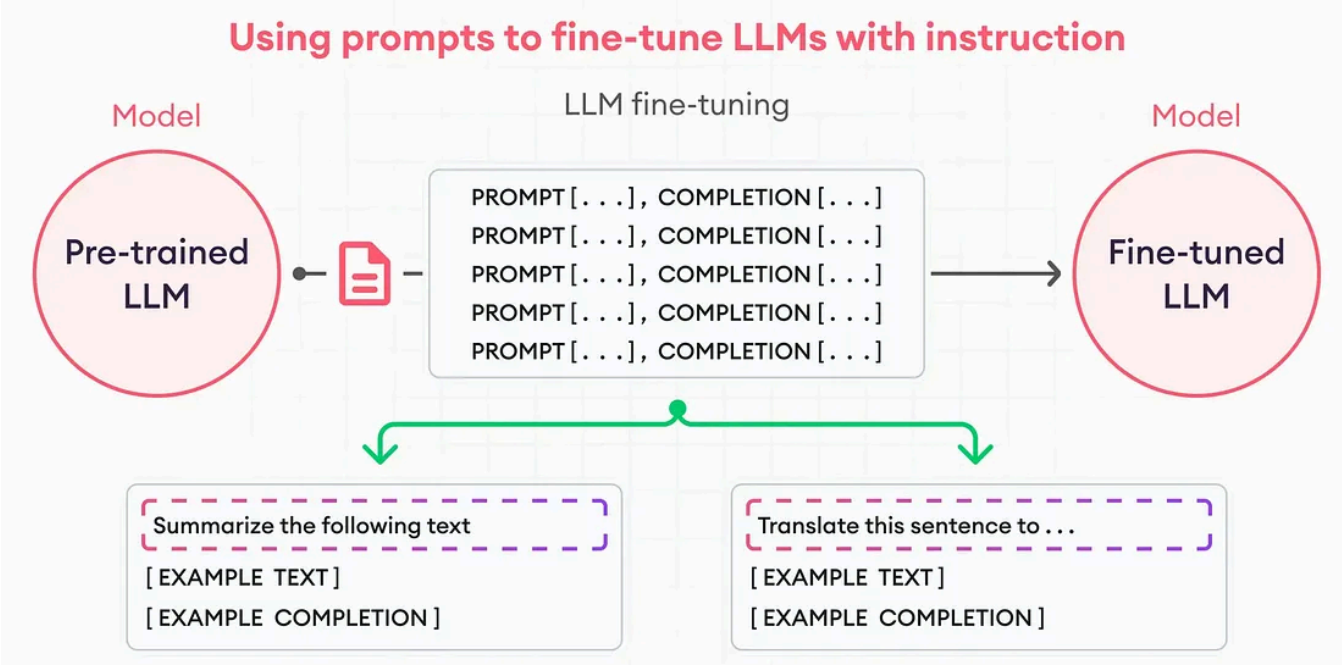
-> Mathematical and Algorithmic Approach:

1. **Initial Model Training**
   - The model undergoes **supervised fine-tuning** with a curated dataset.
   - It learns a probability distribution $P(Y|X;\theta)$, where X is the input, Y is the output, and θ represents model parameters.
2. **Generating Self-Critiques and Improvements**
   - Given an output Y, the model generates a **self-critique** based on constitutional rules.
   - A refined response Y' is then generated to **better align with ethical guidelines.**
3. **Reinforcement Learning Step**
   - The model updates its parameters using a **reward model** that evaluates how well the response follows the constitution.
   - This optimization is done via **Proximal Policy Optimization (PPO)** or other reinforcement learning techniques.

5. Chain-of-Thought (CoT) Prompting : Enhancing LLM Reasoning with Step-by-Step Thinking

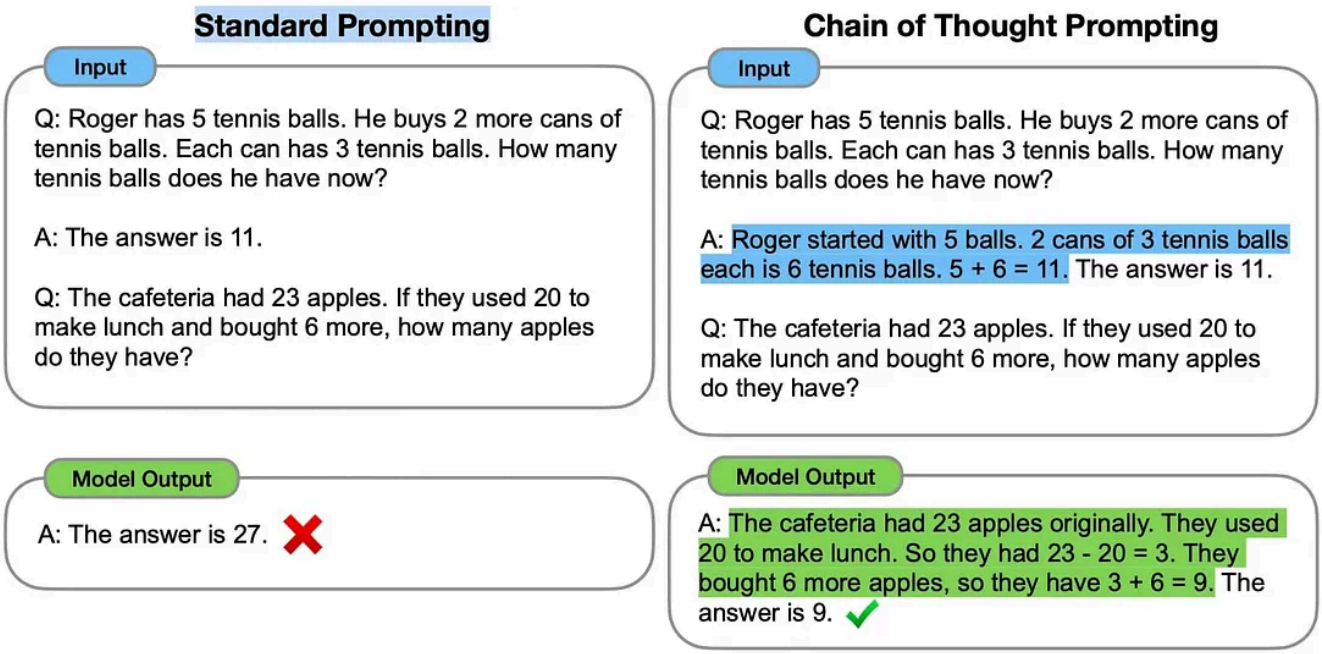
Chain-of-Though (CoT) Prompting is a technique designed to improve the *reasoning abilities* of Large Language Models (LLMs) by encouraging them to generate ***intermediate reasoning steps*** before arriving at a final answer. Instead of providing direct responses, the model **explicitly breaks down its thought process** in a step-by-step manner.

This technique is particularly useful for c**omplex tasks, logical reasoning, multi-step problem-solving, and arithmetic or commonsense reasoning,** making it an essential tool in *Natural Language Processing (NLP) ad Large Language Models (LLMs)*

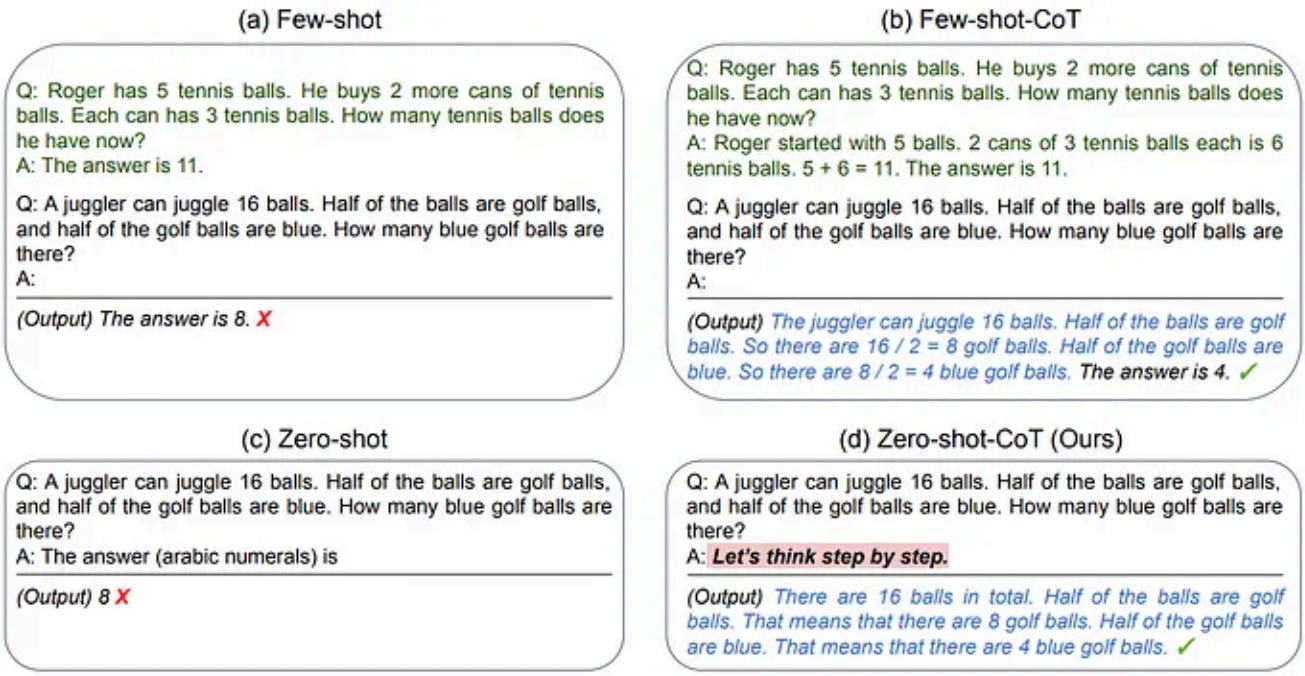


-> Types of Chain-of-Thoughts Prompting:

1. *Manually Crafted CoT Prompting* — Explicitly adds step-by-step reasoning in the prompt
2. *Zero-Shot Chain-of-Thought (Zero-Shot CoT)* — Model is prompted with general reasoning instruction without specific examples
3. *Few-Shot Chain-of-Thought (Few-Shot CoT)* — Includes examples demonstrating step-by-step reasoning before the model attempts a new question
4. *Automatic CoT Prompting (Self-Generated)* — Generates Multiple reasoning paths and selects the most reliable one



Example of Chain-of-Thoughts (**CoT**) Prompting



Zero-Shot CoT Prompting and Few-Shot CoT Prompting Examples

Mathematical and Algorithmic Explanation:

- Given an **input query** x, the traditional model directly predicts an **output** y as:

$$P(y|x;\theta)$$

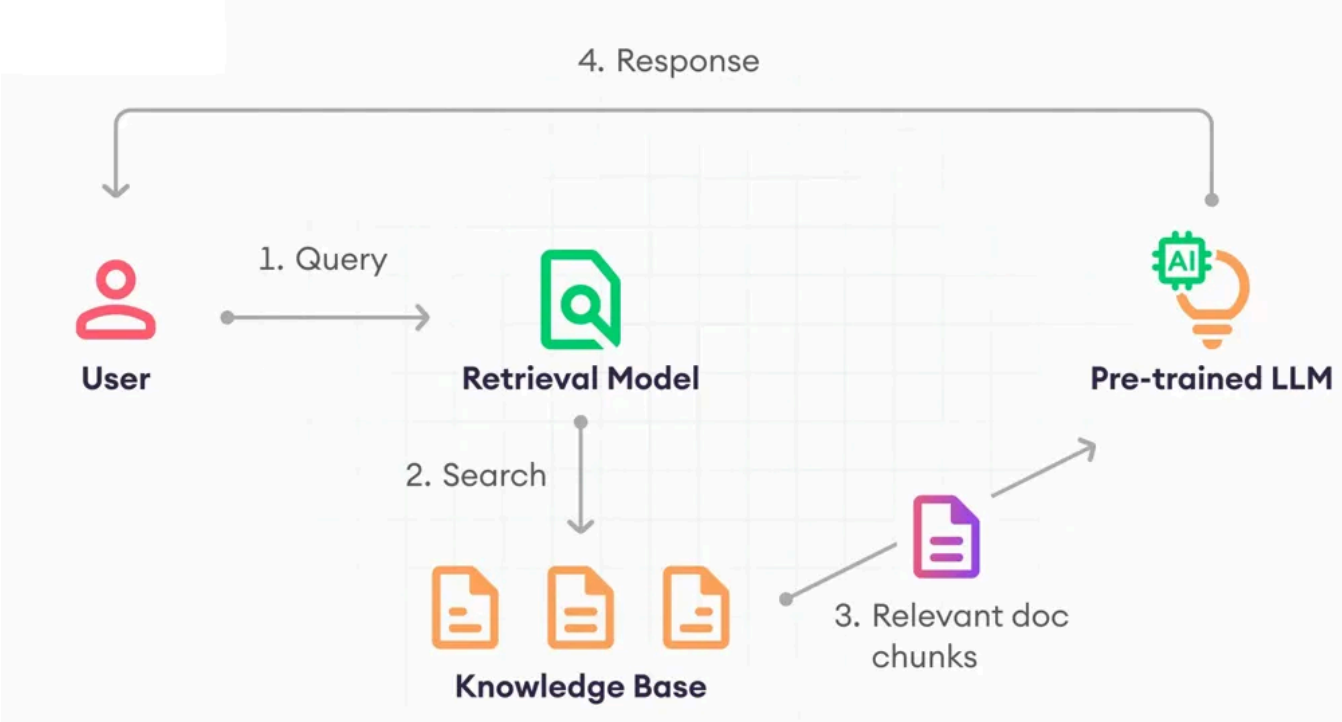
- With CoT, the model generates **intermediate reasoning steps** $s_1, s_2, \ldots, s_n$ before the final answer:

$$P(y|x;\theta) = \sum_s P(y|s,x;\theta)P(s|x;\theta)$$

- This stepwise decomposition helps **propagate logical consistency**, making the model more accurate.

6. Retrieval-Augmented Generation (RAG) : Integrating Retrieval with Generation

Retrieval-Augmented Generation (RAG) enhances the generative capabilities of transformer models by incorporating an **external retrieval mechanism** that provides access to relevant documents or knowledge bases during the generation process. This combination enables models to generate **more accurate, context-aware responses,** especially when the knowledge needed goes beyond the model's training data



Retrieval Augmented Generation (RAG)

-> **Mathematical Explanation:** *This process involves two main steps:*

- **Retrieval**: The model first retrieves relevant documents from a database or corpus based on the input query. This is typically done using a retriever (like BM25 or dense retrievers) that scores documents based on relevance.
- **Generation**: Once relevant documents are retrieved, a generator (usually a transformer model) is used to synthesize the retrieved information and produce a coherent, contextually appropriate response.
- Formally, for a given input x, retrieval is performed to find the top-k relevant documents $D = \{d_1, d_2, \ldots, d_k\}$ from a knowledge base. Then, the generator G produces a response y conditioned on both x and the retrieved documents D:

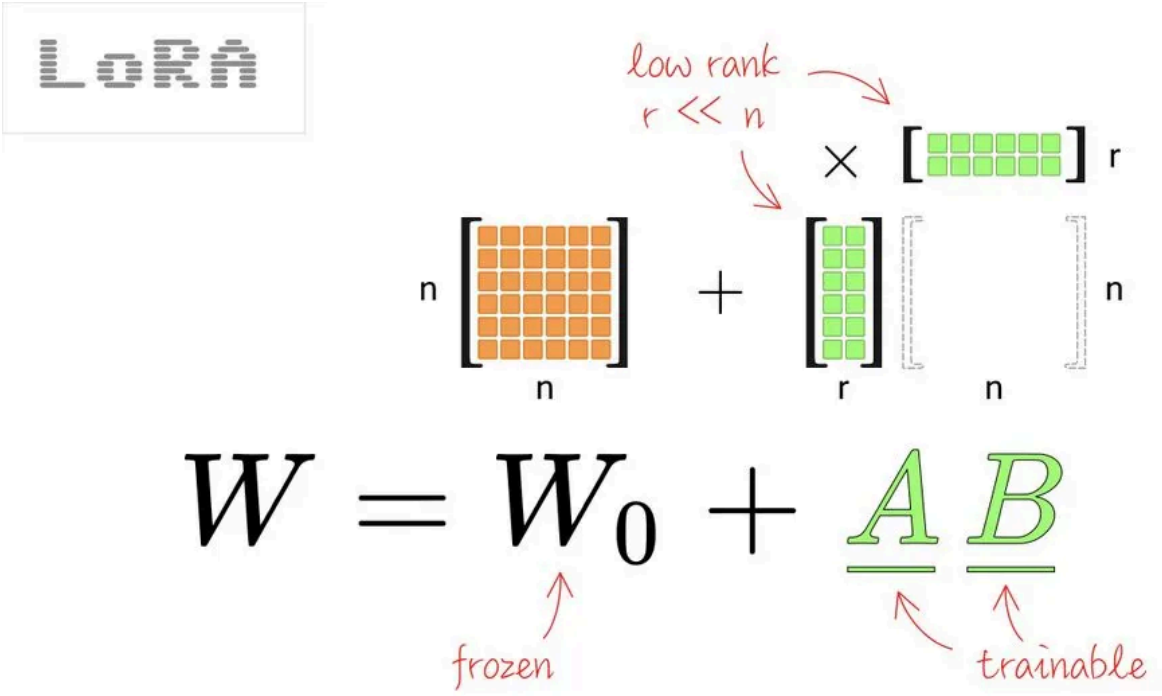
$$y = G(x, D)$$

7. LoRA-v2 And Dynamic Adaptation for Fine-Tuning at Scale

LoRA v2 (Low-Rank Adaptation Version 2) and Dynamic Adaptation offer advanced techniques to mitigate these challenges by enabling low-resource fine-tuning without requiring the full retraining of large models. LoRA v2 utilizes low-rank decomposition, and Dynamic Adaptation introduces flexible adjustments during the training process, providing an optimized path to fine-tuning large-scale models with minimal overhead.

> *LoRA v2 : Efficient Fine-Tuning with Low-Rank Decomposition*

LoRA v2 provides a method to fine-tune large pre-trained models by **decomposing the weights into low-rank matrices**, thus drastically reducing the *computational and memory overhead* required for fine-tuning. Instead of updating the full model parameters, LoRA v2 adds a ***small number of low-rank matrices*** to the model, which capture the adaptation information efficiently



-> Mathematical Explanation of LoRA v2:

Mathematical Explanation: LoRA v2 operates by replacing certain layers' weight matrices W with a sum of a fixed matrix W_0 (the original pre-trained weights) and a low-rank update $W_{\text{update}} = A \cdot B$, where A and B are low-rank matrices. This decomposition allows the adaptation of only the low-rank matrices during training, significantly reducing the number of trainable parameters.

- Standard weight update: $W_{\text{new}} = W_0 + \Delta W$
- LoRA v2 update: $W_{\text{new}} = W_0 + A \cdot B$

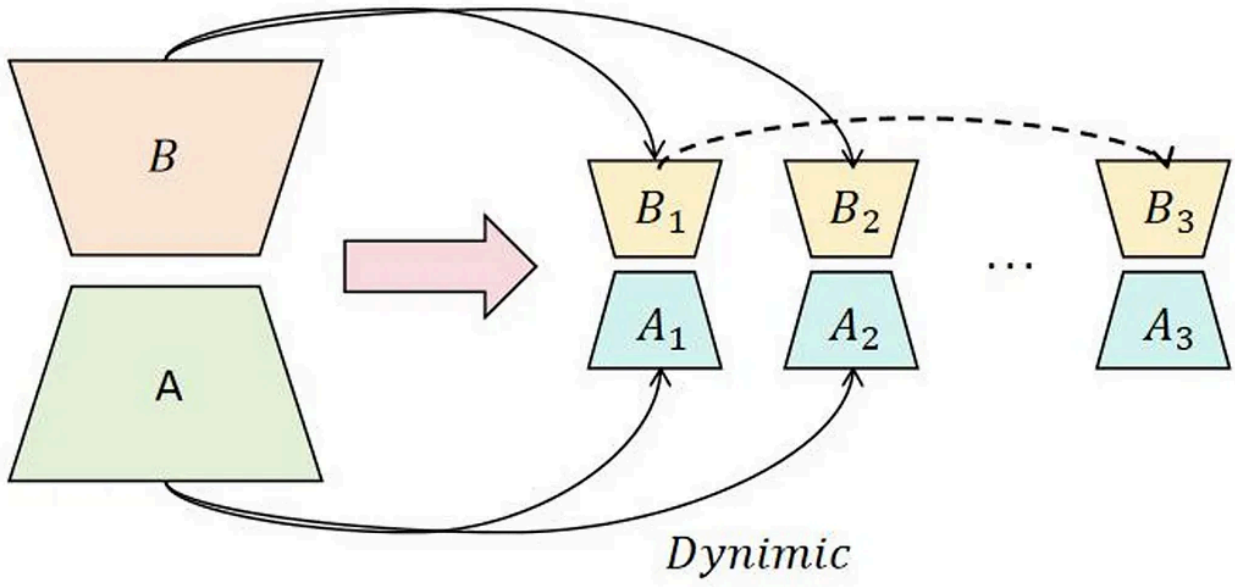
Here, A and B are matrices of low rank (much smaller than W), making it computationally efficient.

-> Implementation Guide:

Fine-Tuning involves adding the low-rank matrices ***A*** and ***B*** during the adaptation phase. These matrices are trained while the rest of the model ***parameters are frozen***, ensuring that only a small fraction of the model is verified

> *Dynamic Adaptation: Flexible Fine-Tuning for Specific Tasks*

Dynamic Adaptation involves adjusting the fine-tuning process based on the specific requirements of the task at hand. Rather than applying a static fine-tuning process to the entire model, Dynamic Adaptations adapts *different parts of the model based on their relevance to the task*. This approach ensures that the model can quickly adapt to new data while ***minimizing computational overhead***



Architecture of Dynamic Adaptation

-> Mathematical Explanation of Dynamic Adaptation:

Mathematical Explanation: Dynamic adaptation involves adjusting the learning rate or the degree of adaptation for different parts of the model. This can be implemented through techniques like adaptive learning rates for specific layers or dynamic freezing of certain parameters based on their contribution to task-specific performance.

- For instance, the learning rate for each layer can be adjusted dynamically:

$$\text{New weights} = W_{\text{old}} - \eta_i \cdot \nabla W_i$$

where η_i is the dynamically adjusted learning rate for the i-th layer.

-> Implementation Guide:

During fine-tuning, certain model layers are **"frozen"**, while others are fine-tuned with ***adaptive learning rates***. This reduces unnecessary updates and allows for more ***targeted improvements***

> *Combining LoRA v2 and Dynamic Adaptation for Scalable Fine-Tuning*

When combined, LoRA v2 and Dynamic Adaptation offer a powerful strategy for fine-tuning large models at scale. **LoRA v2** ensures *memory and computational efficiency* by adapting only *low-rank matrices,* while **Dynamic Adaptation** allows for *targeted updated to specify model layers,* further optimizing the fine-tuning process

-> How They work together?

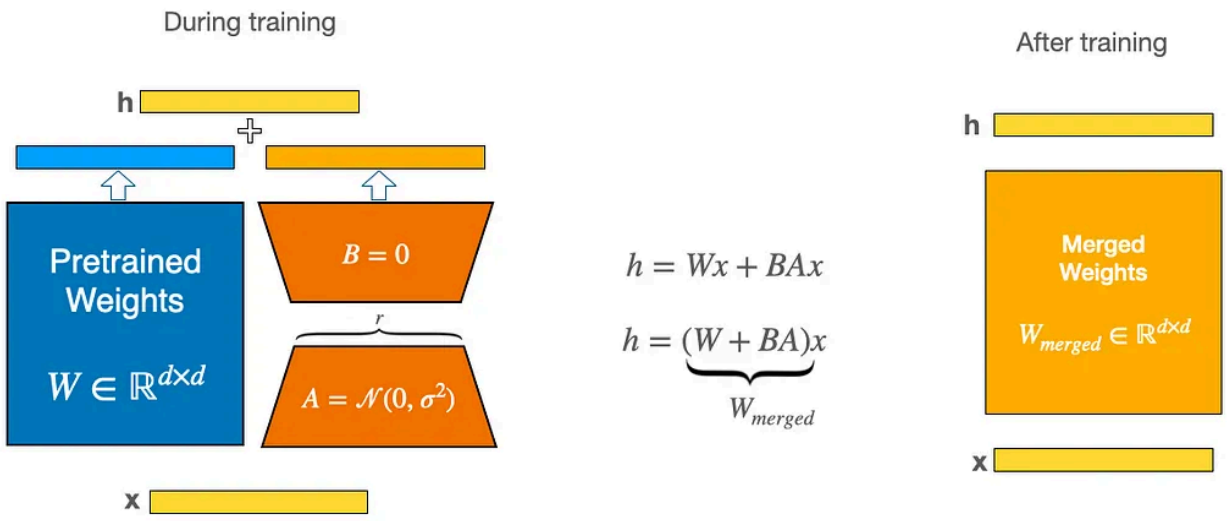
- **LoRA v2** provides the low-rank adaptation for efficient fine-tuning with minimal memory usage.
- **Dynamic Adaptation** complements LoRA v2 by enabling selective fine-tuning of certain layers based on task requirements, ensuring optimal resource utilization

8. LoRA (Low-Rank Adaptation):

LoRA, or **Low-Rank Adaptation**, is a parameter-efficient fine-tuning technique designed to adapt large pre-trained models for specific tasks **without updating all model parameters.** Instead of modifying the entire weight matrix of a neural network, LoRA introduces **low-rank matrices** that approximate the weight updates. This approach significantly reduces the number of **trainable parameters** and computational resources required for fine-tuning

-> Key Idea behind LoRA Innovation:

The key Idea behind LoRA is to decompose weight updates into low-rank matrices, which are much smaller in size compared to the original weight matrices. By doing so, only a small subset of parameters in trained, while the majority of the model's weights remain frozen. This ensures that the pre-trained knowledge of the model is preserved while enabling task-specific customization with minimal overhead



Low-Rank Adaptation (LoRA)

-> Mathematical Foundation of Low-Rank Decomposition:

At its core, LoRA relies on low-rank matrix decomposition to approximate weight updates. For a given weight matrix $W \in \mathbb{R}^{m \times n}$, the update ΔW is approximated as:

$$\Delta W = A \cdot B$$

where:

- $A \in \mathbb{R}^{m \times r}$
- $B \in \mathbb{R}^{r \times n}$
- $r \ll \min(m, n)$ is the rank of the decomposition.

This decomposition reduces the number of trainable parameters from $m \times n$ to $r \times (m + n)$, making it computationally efficient. The rank r controls the trade-off between expressiveness and efficiency: higher ranks provide more flexibility but increase computational cost.

-> Matrix Factorization for Parameter-Efficient Tuning:

LoRA applies low-rank decomposition to specific layers of the model, typically attention layers in Transformers. The process involves:

1. **Identifying Target Layers** — Focus on layers thar are critical for the task, such as self-attention or cross-attention layers.
2. **Decomposing Weight Updates** — Replace the full weight update *ΔW* with its low-rank approximation *A . B*
3. **Training only *A* and *B*** — Freeze the original weights W and train only the low-rank matrices *A* and *B*

This approach ensures that the original model remains intact while allowing task-specific adaptations

-> Hyperparameter Optimization:

A) Rank (r) :

— Controls the size of the low-rank matrices.

— Typical range: *r = 4* to *r = 16.*

B) Learning Rate : Start with a lower learning rate *(e.g., 1e−4)* to avoid destabilizing the frozen weights.

C) Batch Size : Use smaller batch sizes to fit within GPU memory constraints.

D) Weight Decay : Regularize A and B to prevent overfitting.

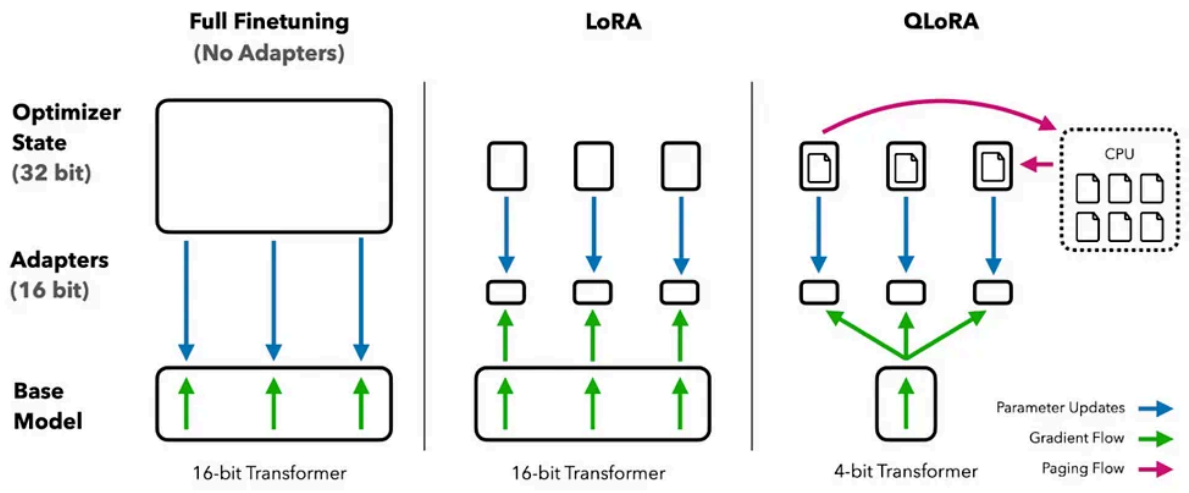
E) Dropout : Optionally apply dropout to the low-rank matrices for added robustness.

9. QLoRA (Quantized Low-Rank Adaptation):

QLoRA, or Quantized Low-Rank Adaptation, is an advanced fine-tuning technique that combines the parameter efficiency of LoRA (Low-Rank Adaptation) with quantization methods to further reduce memory usage. It enables the fune-tuning of large-pretrained models on resource-constrained hardware by representing trainable parameters in lower precision (e.g. 4-bit) while maintaining high-quality outputs.

-> Key Idea behind QLoRA Innovation:

The key idea behind QLoRA is to leverage two complementary techniques — low-rank decomposition and quantization — to achieve efficient fine-tuning. Instead of updating all model weights, LoRA introduces low-rank matrices A and B to approximate weight updates. These matrices are stored in reduced precision (e.g. 4-bit), drastically reducing memory requirements while preserving the quality of fine-tuned models.



-> Mathematical Foundation of Low-Rank Decomposition:

At its core, QLoRA relies on low-rank matrix decomposition to approximate weight updates. For a given weight matrix $W \in \mathbb{R}^{m \times n}$, the update ΔW is decomposed as:

$$\Delta W = A \cdot B$$

where:

- $A \in \mathbb{R}^{m \times r}$
- $B \in \mathbb{R}^{r \times n}$
- $r \ll \min(m, n)$ is the rank of the decomposition.

This decomposition reduces the number of trainable parameters from $m \times n$ to $r \times (m + n)$. During training, only A and B are updated, while W remains frozen.

-> 4-Bit Quantization Techniques for Deep Models:

Quantization involves reducing the precision of model weights or activations to lower bit-widths. In QLoRA, 4-bit quantization is applied to both frozen weights and trainable parameters:

> *Key Techniques:*

A) Uniform Quantization : Weights are scaled and rounded to fit within a fixed range of 4-bit values. A scaling factor maps the quantized values back to their original range during computation.

B) Non-Uniform Quantization : Uses learned quantization ranges to better capture the distribution of weights, improving accuracy.

C) Dequantization : During forward passes, the 4-bit weights are temporarily converted back to higher precision to maintain computational stability.

D) Gradient Scaling : Gradients are scaled appropriately to ensure that updates remain accurate and effective even with low-precision weights.

-> Paged Optimizers for Better GPU Memory Efficiency

One of the challenges in training large models is managing GPU memory efficiently. QLoRA addresses this issue by using paged optimizers , which optimize memory allocation during training.

> *How Paged Optimizers Work:*

A) Memory Fragmentation Reduction : Paged optimizers divide optimizer states (e.g., momentum, variance) into fixed-size chunks, similar to how memory paging works in operating systems. This prevents memory fragmentation and allows efficient use of available GPU memory.

B) Offloading Unused States : Optimizer states that are not actively needed during a particular training step can be offloaded to CPU memory or disk storage, freeing up GPU memory for other operations.

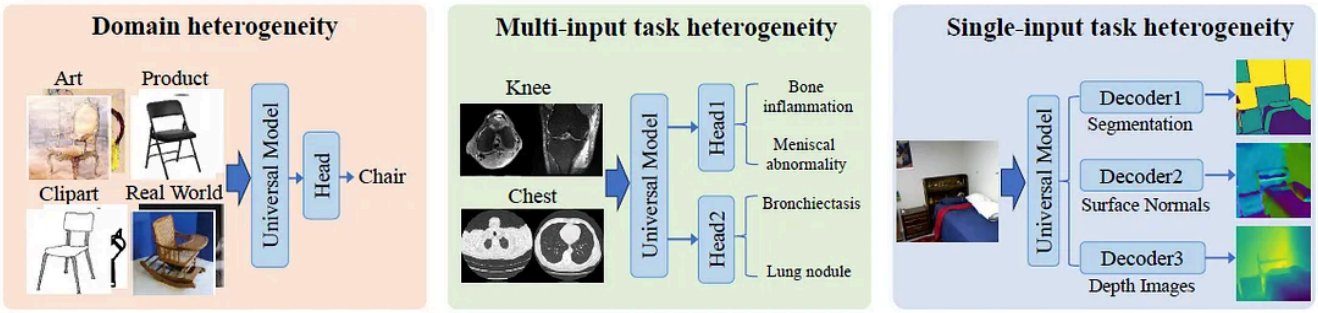
C) Dynamic Memory Allocation : Paged optimizers dynamically allocate memory based on the current workload, ensuring that resources are used optimally.

10. MoLA (Mixture of Low-rank Adapters):

In the landscape of large language models, training on diverse datasets often introduces significant challenges. *Mixture of Low-rank Adapters (MoLA)* emerges as an innovative solution to tackle heterogeneity conflicts during model training.

Traditional multi-task learning approaches struggle with data diversity, often suppressing model performance. MoLA introduces a novel approach by:

- Using multiple low-rank adapters attached to a shared backbone
- Ensuring controllable parameter increase
- Disentangling shared and complementary knowledge
- Providing flexibility in handling heterogeneous training data



Three common types of heterogeneous data

-> Key Idea behind MoLA Innovation:

MoLA addresses critical challenges in multi-task learning (MTL) by introducing a novel approach to handling heterogeneous data training:

> *Comparative Advantages*

- Hard Parameter Sharing (HPS) lacks task-specific feature learning
- Soft Parameter Sharing (SPS) struggles with parameter scalability

- Mixture of Experts (MoE) has high computational costs

Core Innovations

- Uses task identifiers or trainable routers to combine parameters
- Provides flexible, low-parameter approach to heterogeneous training
- Maintains sufficient representation capacity
- Better at capturing task-specific features compared to existing methods

-> Mathematical Foundation of Mixture of Low-Rank Adapters:

Mixture of Low-Rank adapters (MoLA) assigns two low-rank factors:

$$B_i \in R^{C_{\text{out}}k \times r_k}, \quad A_i \in R^{r_k \times C_{\text{in}}k}$$

for each adapter. Then, the convolution operation can be transformed into:

$$g_t = \left(W_0 + \sum_{i=1}^{E} \alpha_i B_i A_i \right) h_t.$$

Where:

- For brevity, we omit the **reshape operation**.
- α_i denotes the **contribution weight** of the i-th low-rank adapter.
- r represents the **rank**, and $i \in \{1, ..., E\}$.
- E is the **number of adapters** defined by users.

MoLA — Grad: Explicit Gradient Separation

MoLA-Grad effectively addresses heterogeneous data training challenges in target-aware scenarios through *explicit gradient separation*. By assigning specific low-rank adapters to individual tasks *(E = T)*, MoLA-Grad ensures each adapter computes gradients exclusively for its corresponding task *(αt = 1, αi≠t = 0)*. This targeted approach eliminates **cross-task interference**, effectively mitigating training conflicts that typically arise when handling diverse data distributions.

-> Mathematical Foundation of MoLA-Grad:

MoLA-Grad adopts the idea of group convolution to optimize the calculation process of convolution operations.

Specifically, for each convolution with MoLA-Grad, the output feature g can be computed by:

$$\begin{aligned} g &= (W_0 + B_1 A_1)h_1 \cup \cdots \cup (W_0 + B_T A_T)h_T \\ &= (W_0 + BA \circ M)h = W'h \end{aligned} \tag{1}$$

where:

- $BA = B_1 A_1 \cup \cdots \cup B_T A_T$
- $\cup$ denotes the **concatenation** operation.
- $\circ$ represents **element-wise multiplication** through broadcast.
- W_0 is used to avoid **redundant computation and excessive memory usage** in the computation of all tasks.

The **aggregated weights** are being reshaped:

$$W' \in R^{b \times C_{\text{out}} \times C_{\text{in}} \times k \times k} \rightarrow R^{bC_{\text{out}} \times C_{\text{in}} \times k \times k}, \quad h \rightarrow R^{1 \times bC_{\text{in}} \times H \times W}$$

Note: Set the group numbers to b

In conclusion, the output feature g will be a standard group convolution, which can be easily implemented using existing deep learning libraries.

MoLA — Router: Dynamic Adapter Collaboration

MoLA-Router offers a *flexible solution for target-agnostic scenarios* where task identifiers may be unavailable. Instead of explicit assignments, MoLA-Router employs a trainable router mechanism that **dynamically generates weighted combinations of adapters** ***(∑αi = 1)*** for each input. This approach implicitly *separates heterogeneous gradients* while enabling productive collaboration among **multiple low-rank adapters,** making it applicable to a broader range of scenarios than its MoLA-Grad counterpart.

-> Mathematical Foundation of Unconstrained MoLA-Routers:

The router can be implemented by any structure and applied at any layer.

For simplicity, the formulation uses:

- three residual blocks $\phi(\theta)$
- gate function $g(\eta)$ to build a shared router for all layers with MoLA

Contribution Weights output by the Router:

$$\vec{\alpha} = \text{softmax}(g(\phi(\theta, x), \eta))$$

where:

- $\vec{\alpha} \in R^{b \times E}$
- x is the input samples

In conclusion, Unconstrained routers may not produce the right mixing weights for different tasks, causing parameters to mix up and fail to reduce conflicts between tasks.

-> Mathematical Foundation of Constrained MoLA-Routers:

To maintain clear separation, this MoLA-Router introduces:

- Trainable MLP $\phi(\cdot)$ to map $\vec{\alpha}_i$ to a higher-dimensional space, i.e., $\omega = \phi(\vec{\alpha})$.
- Task-wise Decorrelation (TwD) loss L_{TwD} to supervise the learning of mixing weights.

$$L_{\text{TwD}} = -\sum_{i=1}^{b}\sum_{j=1}^{b} \mathbf{1}_{i \neq j} \cdot \mathbf{1}_{t_i = t_j} \log \frac{e^{\omega_i^\top \omega_j / \tau}}{\sum_{k=1, k \neq i}^{b} e^{\omega_i^\top \omega_k / \tau}}$$

where:

- t_i, t_j represent the task identifiers.
- i-th and j-th input sample.
- 1 is the indicator function.
- τ is the temperature with the default value 1.

> *Task-wise Decorrelation (TwD) Loss Function:*

The MoLA architecture employs a specialized loss function that incorporates Task-wise Decorrelation (TwD) to optimize adapter mixing. This TwD component guides the router to generate similar contribution weights for data from the same task while ensuring distinct weight patterns across different tasks. Combined with traditional task-specific losses (Li), this approach enables MoLA to effectively manage heterogeneous data training while maintaining task coherence.

-> Mathematical Calculation of TwD Loss:

Let Li denote the loss function of the $i-th$ task, and the total loss is

$$L_{\text{total}} = \sum_{i=1}^{T} \frac{1}{T} L_i + \beta L_{\text{TwD}}$$

where β is a hyper-parameter and is nonzero

MoLA vs. LoRA: Key Differences

While MoLA builds upon the low-rank adapter concept introduced in LoRA, it represents a significant evolution with three fundamental differences:

1. **Training Approach**
   LoRA fine-tunes pre-trained models for specific tasks, while MoLA trains from scratch alongside the backbone model.

2. **Information Processing**
   LoRA emphasizes specific information by amplifying certain model features, while MoLA takes the opposite approach — reducing dominant features to capture more diverse information and prevent conflicts between different tasks.

3. **Flexibility and Application**
   LoRA typically adapts models to single specific tasks, while MoLA's router system enables handling multiple tasks without prior knowledge of which task is being performed.

This makes MoLA particularly valuable for real-world applications where data comes from many different sources and contexts.

Selection of Fine-Tuning Methods for Large Language Models for Specific Task:

FINE-TUNING METHODS	WHEN TO USE	RESOURCE REQUIREMENTS	BEST FOR	LIMITATIONS
Instruction Tuning	• When converting a general LLM to follow instructions • When you have high-quality instruction-response pairs	Medium to High	• Improving task-specific performance • Teaching models to follow instructions • Establishing basic capabilities	• Requires large datasets of instruction-response pairs • Quality of data heavily influences results
Supervised Fine-Tuning (SFT)	• When you have labeled data for specific tasks • For domain adaptation • As foundation for further fine-tuning	Medium to High	• Domain-specific adaptation • Learning specific formats • Teaching factual knowledge	• Can lead to overfitting • Requires high-quality labeled data • May introduce biases in data
RLHF	• When optimizing for human preferences • For reducing harmful outputs • After SFT for further alignment	Very High	• Alignment with human values • Reducing toxicity • Improving helpfulness and honesty	• Computationally expensive • Requires human feedback collection • Complex implementation
Constitutional AI	• When implementing specific behavioral guardrails • For self-improving model behavior • For reducing harmful outputs without extensive human feedback	High	• Implementing specific rules/principles • Reducing harmful content • More scalable than pure RLHF	• Still requires some human oversight • Rules must be carefully designed • Can introduce unwanted constraints
CoT Prompting	• For complex reasoning tasks • When explicit reasoning steps improve performance • No need for model parameter updates	Low	• Math problems • Logical reasoning • Multi-step problems	• Not true "fine-tuning" (no parameter updates) • Limited by model's base capabilities • May struggle with very complex problems
RAG	• When incorporating external knowledge • For improving factuality • When domain knowledge changes frequently	Medium	• Question answering • Reducing hallucinations • Up-to-date information needs	• Not true "fine-tuning" • Requires maintaining external knowledge base • Retrieval quality impacts performance
LoRA	• When computational resources are limited • For efficient domain adaptation • When full fine-tuning is prohibitive	Low to Medium	• Efficient adaptation to new domains • Preserving general capabilities • Multiple task adaptations	• May not match full fine-tuning performance • Requires hyperparameter tuning • Limited parameter updates
QLoRA	• When GPU memory is very limited • For fine-tuning larger models on consumer hardware • When seeking extreme efficiency	Very Low	• Training on consumer GPUs • Larger models with limited resources • Quick iterations	• Additional quantization artifacts • Complexity of implementation • Potentially lower performance than full-precision
LoRA v2 & Dynamic Adaptations	• When adapting to changing conditions • For more flexible parameter updates • When seeking improved LoRA performance	Low to Medium	• Dynamic task switching • Improved efficiency over LoRA • More adaptive learning	• Increased implementation complexity • Less established than original LoRA • May require custom optimization
MoLA	• When fine-tuning for multiple tasks • For better parameter efficiency than LoRA • When seeking better transfer learning	Medium	• Multi-task learning • Improved parameter efficiency • Better knowledge transfer between tasks	• More complex than LoRA • Newer with less community support • Router optimization challenges

Comparison Chart for Different Fine-Tuning Methods for Large Language Models

Workflow How the Pipeline will Work:

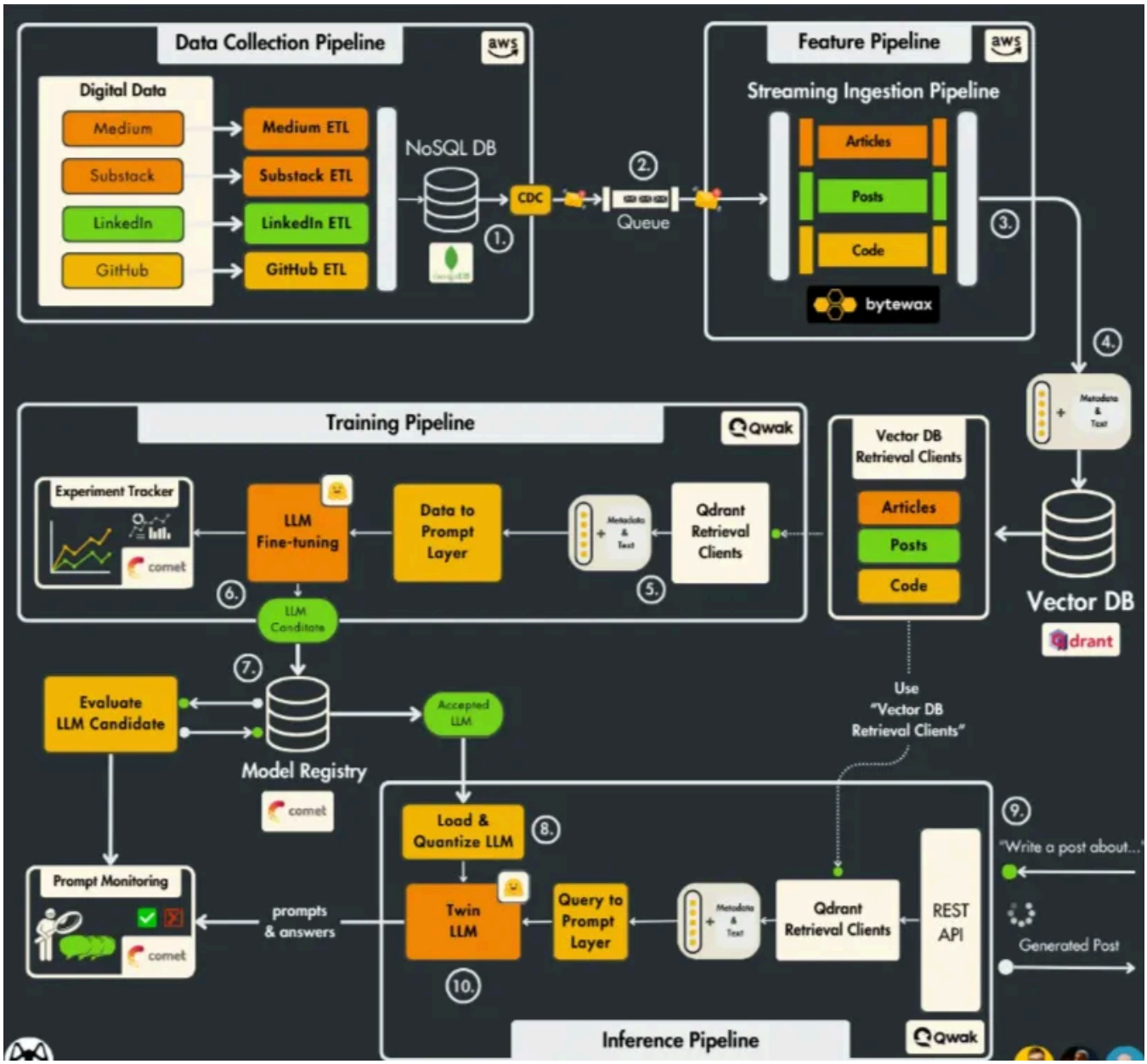
The **LLM Fine-Tuning Pipeline** is a comprehensive framework designed to streamline the process of customizing Large Language Models (LLMs). It encompasses several key stages:

1. **Data Collection**: The pipeline initiates by gathering domain-specific datasets pertinent to the target application.
2. **Data Preparation**: Collected data undergoes preprocessing, including cleaning and tokenization, to ensure compatibility with the LLM's input requirements.
3. **Fine-Tuning**: The prepared dataset is utilized to fine-tune a pre-trained LLM, allowing it to adapt to domain-specific language patterns and nuances.

4. **Inference:** Post fine-tuning, the model is deployed to generate predictions or responses based on new input data.

5. **Evaluation:** The pipeline assesses the model's performance using metrics such as perplexity, coherence, and BLEU scores to ensure it meets the desired standards.

This structured approach ensures efficient customization of LLMs, optimizing their performance for specific applications.



Pipeline

Python Implementation of Different Fine-Tuning Methods:

STEP — 1 -> Load your Dataset and Base Model:

Here's a short description of the two classes in the same style as before:

> *-> DatasetLoader Class*

Handles dataset loading and saving for fine-tuning tasks.

- Loads datasets from the Hugging Face Hub or local files (`.csv` , `.json` , `.txt`).
- Automatically detects file type and loads accordingly.
- Saves datasets to disk in the specified format.
- Returns a `DatasetDict` with a `'train'` split.

```
# DEPENDENCIES

import os
import datasets
from datasets import DatasetDict
from datasets import load_dataset


class DatasetLoader:
    """
    A class to handle loading datasets for fine-tuning from either a local file
    """

    def __init__(self):
        """
        Initializes the DatasetLoader instance with dataset set to None.

        Attributes:

            dataset            : The dataset to be used for fine-tuning (initia

        """

        self.dataset = None

    def load_dataset(self, dataset : str) -> datasets.DatasetDict:

        """
        Loads a dataset for fine-tuning from either a local file or the Hugging

        This function first attempts to load the dataset from the Hugging Face
        If that fails, it checks if the dataset exists as a local file and load

        Supported file formats:
        - `.csv`  : Loaded using `datasets.load_dataset('csv')`
        - `.json` : Loaded using `datasets.load_dataset('json')`
        - `.txt`  : Reads line-by-line and stores as a list under the 'train' k
```

```
        Arguments:

            `dataset`           {str}         : Either the name of a Hugging F
                                                or a local file path (e.g., `'

        Returns:

            datasets.DatasetDict              : The loaded dataset with a 'tra

        Raises:

            ValueError                        : If the dataset is not found or

        """

        print(f"Loading dataset from {dataset}...")

        # LOADING DATASET FROM HUGGING FACE HUB
        try:

            self.dataset = load_dataset(dataset)

            print(f"Loaded dataset from Hugging Face: {dataset}")

        # LOADING DATASET FROM LOCAL FILE
        except:

            if os.path.exists(dataset):
                extension = os.path.splitext(dataset)[1]

                if extension == '.csv':
                    self.dataset = load_dataset('csv', data_files = dataset)

                elif extension == '.json':
                    self.dataset = load_dataset('json', data_files = dataset)

                elif extension == '.txt':
                    with open(dataset, 'r') as f:
                        texts    = [line.strip() for line in f]
                    self.dataset = {'train': texts}

                else:
                    raise ValueError(f"Unsupported file extension: {extension}"

                print(f"Loaded dataset from local file: {dataset}")

            else:
                raise ValueError(f"Dataset {dataset} not found")

        print(f"Dataset loaded with {len(self.dataset['train'])} training examp

        return self.dataset
```

```
    def save_dataset(self, dataset : DatasetDict, file_path : str) -> None:
        """
        Saves the loaded dataset to a specified file path in CSV, JSON, or TXT

        Arguments:

            dataset          {DatasetDict}     : The dataset to be saved.

            file_path            {str}         : The destination file path with

        Raises:

            ValueError : If the file extension is unsupported.

        """
        extension        = os.path.splitext(file_path)[1]

        if extension    == '.csv':
            dataset['train'].to_csv(file_path, index=False)

        elif extension  == '.json':
            dataset['train'].to_json(file_path, orient = 'records', lines = Tru

        elif extension  == '.txt':

            with open(file_path, 'w') as f:

                for item in dataset['train']:

                    f.write(f"{item}\n")

        else:
            raise ValueError(f"Unsupported file extension: {extension}")

        print(f"Dataset saved successfully to {file_path}")
```

> *-> ModelLoader Class:*

Manages downloading and loading of pre-trained language models.

- Loads LLMs from Hugging Face based on model ID.
- Supports quantized loading (4-bit) for large models like LLaMA.
- Automatically detects model architecture (`Seq2Seq` or `CausalLM`).
- Loads the associated tokenizer and ensures padding token is set.

```
# DEPENDENCIES

import torch

import transformers
from transformers import AutoTokenizer
from transformers import BitsAndBytesConfig
from transformers import AutoModelForCausalLM
from transformers import AutoModelForSeq2SeqLM


class ModelLoader:
    """
    A class to handle downloading and loading of pretrained LLMs from Hugging F
    """

    def __init__(self):
        """
        Initializes the ModelLoader instance with model and tokenizer set to Nc

        Attributes:

            model              : The language model to be fine-tuned (initially
            tokenizer          : The tokenizer associated with the model (initi

        """

        self.model                = None
        self.tokenizer            = None

    def download_model(self, model_id : str) -> tuple:
        """
        Download a pretrained LLM from Hugging Face.

        Arguments:

            `model_id`        {str}      : The Hugging Face model ID

        Returns:

            Tuple [torch.nn.Module, transformers.PreTrainedTokenizer]: The loac

        """

        print(f"Downloading model {model_id}...")

        if any(size in model_id.lower() for size in ['7b', '13b', '70b', 'llama

            print("Loading large model with quantization...")

            bnb_config               = BitsAndBytesConfig(load_in_4bit
```

```
                                                    bnb_4bit_use_double_c
                                                    bnb_4bit_quant_type
                                                    bnb_4bit_compute_dtyp
                                                    )

            # Use AutoModelForCausalLM for decoder-only models
            self.model                = AutoModelForCausalLM.from_pretrained(moc
                                                                             qua
                                                                             dev
                                                                             tru
                                                                             )


        else:

            # Check if model is an encoder-decoder model or decoder-only
            if any(arch in model_id.lower() for arch in ['t5', 'bart', 't0']):
                self.model            = AutoModelForSeq2SeqLM.from_pretrained(mc

            else:
                self.model            = AutoModelForCausalLM.from_pretrained(moc

        self.tokenizer                = AutoTokenizer.from_pretrained(model_id)

        # Ensure padding token exists
        if self.tokenizer.pad_token is None:
            self.tokenizer.pad_token = self.tokenizer.eos_token

        print(f"Model {model_id} loaded successfully")

        return self.model, self.tokenizer
```

STEP — 2 -> Dataset Preparation:

The dataset preparation process involves tokenizing and formatting data for fine-tuning a language model. The **DatasetPreparer** class supports two modes: **Instruction Fine-Tuning**, where input-output pairs are structured based on a prompt template, and **Regular Fine-Tuning**, where raw text is tokenized for language modeling. The class first processes the dataset to format instructions and responses, then tokenizes the text using a specified tokenizer and model. For decoder-only models, labels are set to match input tokens, while encoder-decoder models tokenize inputs and responses separately. The final dataset is structured with tokenized inputs, attention masks, and labels, ensuring compatibility with transformer-based LLMs for fine-tuning.

```
# DEPENDENCIES

import re
import datasets
from re import A

from sympy import N
import transformers
from transformers import PreTrainedTokenizer
from transformers import AutoModelForCausalLM


class DatasetPreparer:
    """
    A class for preparing datasets for fine-tuning language models.

    This class supports two modes of dataset preparation:

    1. **Instruction Fine-Tuning**: If `instruction_column` and `response_colum
       the dataset is formatted with input-output pairs based on the `prompt_te

    2. **Regular Language Model Fine-Tuning**: If `text_column` is provided, th
       as a standard language modeling dataset.
    """

    def __init__(self, dataset: datasets.DatasetDict, tokenizer : transformers.
        """
        Initializes the DatasetPreparer class.

        Arguments:

            `dataset`     {datasets.DatasetDict}  : The raw dataset to be prepa

            `tokenizer`   {transformers.PreTrainedTokenizer} : The tokenizer us

            `model`       {transformers.PreTrainedModel}     : The model to det

        """
        self.model             = model
        self.dataset           = dataset
        self.tokenizer         = tokenizer
        self.prepared_dataset  = None

    def prepare_dataset(self,
                        max_length          : int = 512,
                        instruction_column  : str = None,
                        response_column     : str = None,
                        text_column         : str = None,
                        prompt_template     : str = None
                        ) -> datasets.DatasetDict:
```

```
        """
        Prepares the dataset for fine-tuning by tokenizing and formatting input

        This function supports two modes of dataset preparation:

        1. **Instruction Fine-Tuning**: If `instruction_column` and `response_c
           the dataset is formatted with input-output pairs based on the `prc

        2. **Regular Language Model Fine-Tuning**: If `text_column` is provided
           as a standard language modeling dataset.

        Arguments:

            `max_length`              {int, optional}          : Maximum seque

            `instruction_column`      {str, optional}          : Column name c

            `response_column`         {str, optional}          : Column name c

            `text_column`             {str, optional}          : Column name c

            `prompt_template`         {str, optional}          : Template for

        Returns:

            datasets.DatasetDict: The tokenized dataset ready for fine-tuning.

        Raises:

            ValueError: If the tokenizer or dataset is not loaded before callir
        """

        if self.tokenizer is None or self.dataset is None:
            raise ValueError("Model tokenizer and dataset must be loaded first"

        print("Preparing dataset for fine-tuning...")

        # FOR INSTRUCTION FINE-TUNING

        if instruction_column and response_column:

            def format_instruction(example : dict) -> dict:
                """
                Formats an example for instruction fine-tuning by generating ir

                Arguments:

                    example         {dict}      : A dictionary representing a s
                                                  containing at least `instruct

                Returns:

                    dict: A modified example with two new keys:
```

```
            - `"input_text"`: The formatted instruction prompt.
            - `"output_text"`: The corresponding response.

        Behavior:
            - If `prompt_template` is provided, it formats the instruct
            - If no template is provided, it defaults to `"Instruction:
        """

        label_map                   = {0: "negative", 1: "positive"}
        sentiment                   = label_map.get(example.get("label",

        if prompt_template:
            prompt                  = prompt_template.format(instructior
            example["input_text"]   = prompt
            example["output_text"] = sentiment

        else:
            example["input_text"]   = f"Given the following movie review
            example["output_text"] = sentiment

        return example

    formatted_dataset               = self.dataset.map(format_instructic

    def tokenize_function(examples : dict) -> dict:
        """
        Tokenizes input and output text for fine-tuning language models

        Arguments:

            examples                {dict}          : A batch of data

                - `"input_text"`      {str}           : The input text

                - `"output_text"` {str, optional}     : The expected re

        Returns:

            dict: A dictionary containing tokenized inputs with the fol

                - `"input_ids"`        : Tokenized input sequences.

                - `"attention_mask"`   : Attention masks for padding.

                - `"labels"`           : Tokenized output sequences (fo

        Behavior:

            - **For decoder-only models (`AutoModelForCausalLM`)**:
            - The `labels` are set to the same value as `input_ids` (se
            - **For encoder-decoder models**:
            - The `input_text` is tokenized separately for input.
            - The `output_text` is tokenized to create the `labels`.
```

```
            """

            model_inputs               = self.tokenizer(examples["input_tex
                                                        truncation      = T
                                                        max_length      = m
                                                        padding         = "
                                                        return_tensors  = N
                                                        )

            # FOR DECODER-ONLY MODELS
            if isinstance(self.model, AutoModelForCausalLM):
                model_inputs["labels"] = model_inputs["input_ids"].copy()

            # FOR ENCODER-ONLY MODELS
            else:

                labels                 = self.tokenizer(examples["output_te
                                                        truncation      = T
                                                        max_length      = m
                                                        padding         = "
                                                        return_tensors  = N
                                                        )

                model_inputs["labels"] = labels["input_ids"]

            return model_inputs

        # FOR REGULAR LANGUAGE MODEL FINE-TUNING
        else:
            text_column                = text_column or "text"

            def tokenize_function(examples : str) -> dict:
                """
                Tokenizes input text for fine-tuning a language model.

                Arguments:

                    examples      {str}     : A batch of dataset examples conta

                Returns:

                    dict: A dictionary containing the tokenized output with the
                        - `"input_ids"`     : Tokenized input sequences.
                        - `"attention_mask"`: Attention masks indicating which

                Behavior:

                    - Tokenizes the text from the specified `text_column`.
                    - Truncates sequences longer than `max_length`.
                    - Pads shorter sequences to `max_length`.

                """
```

```
            return self.tokenizer(examples[text_column],
                                  truncation      = True,
                                  max_length      = max_length,
                                  padding         = "max_length",
                                  return_tensors  = None,
                                  )

        print("Tokenizing dataset...")

        tokenized_dataset             = formatted_dataset.map(tokenize_functior
                                                              batched         =
                                                              remove_columns  =
                                                              )

        self.prepared_dataset         = tokenized_dataset

        print("Dataset preparation completed")

        return self.prepared_dataset
```

STEP — 3 -> Fine-Tuning Methods for Large Language Models:

Fine-tuning adapts a pre-trained LLM to specific tasks by updating its weights with labeled data. Methods include **Supervised Fine-Tuning** (training on input-output pairs), **Instruction Fine-Tuning** (optimizing response generation), **RLHF** (aligning with human preferences using reinforcement learning), and **LoRA** (efficient tuning with minimal parameter updates). This process enhances the model's performance on domain-specific tasks while preserving general knowledge

> *LoRA (Low-Rank Adaptation):*

This Python class, **LoRAFineTuning**, applies **LoRA (Low-Rank Adaptation)** to a pre-trained **LLM** for efficient fine-tuning. It auto-detects model architecture (LLaMA, GPT, BERT, T5, etc.), selects target layers, and configures LoRA parameters. It also supports quantized models, reducing memory usage while maintaining performance.

```
# DEPENDENCIES

from tqdm import tqdm
from datetime import datetime

import transformers
```

```
from transformers import AutoModelForCausalLM

from peft import TaskType
from peft import LoraConfig
from peft import get_peft_model
from peft import prepare_model_for_kbit_training

class LoRAFineTuning:
    """
    A class to handle LoRA (Low-Rank Adaptation) for parameter-efficient fine-t

    LoRA reduces the number of trainable parameters by applying low-rank adapta
    to specific layers, making fine-tuning more memory efficient.
    """

    def __init__(self, model: transformers.PreTrainedModel) -> None:
        """
        Initializes the LoRAFineTuning class.

        Arguments:

            `model`             {transformers.PreTrainedModel}        : The pre

        """
        self.model = model

    def apply_lora(self, rank : int = 8, lora_alpha : int = 16, lora_dropout :
        """
            Apply LoRA (Low-Rank Adaptation) to the model for parameter-efficie

            LoRA reduces the number of trainable parameters by applying low-rar
            to specific layers, making fine-tuning more memory efficient.

            Arguments:

                `r`                         {int}             : Rank of the LoR

                `lora_alpha`                {int}             : Scaling factor

                `target_modules`       {list, optional}       : List of module

            Raises:

                ValueError: If the model is not loaded before applying LoRA.

            Returns:

                `model` (transformers.PreTrainedModel): The model with LoRA app

            Functionality:

                - Detects the model type and selects appropriate layers for LoR
                - Supports different architectures like LLaMA, GPT, BERT, and T
```

```
            - Handles quantized models by preparing them for k-bit training
            - Configures and applies LoRA using PEFT (Parameter-Efficient F
            - Prints the number of trainable parameters after applying LoRA

        Example Usage:
            ```
            model = trainer.apply_lora(rank = 8, lora_alpha = 16, lora_drop
            ```
    """
    if self.model is None:
        raise ValueError("Model must be loaded first")

    total_start_time       = datetime.now()
    print(f"\nLoRA Application Started at: {total_start_time.strftime('%H:%

    print("Applying LoRA for parameter-efficient fine-tuning...")

    # Auto-detect target modules if not specified
    if target_modules is None:
        model_type         = self.model.config.model_type if hasattr(self.m

        if "llama" in model_type.lower() or "mistral" in model_type.lower()
            target_modules = ["q_proj", "v_proj", "k_proj", "o_proj"]

        elif "gpt" in model_type.lower() or "falcon" in model_type.lower():
            target_modules = ["c_attn", "c_proj"]

        elif "bert" in model_type.lower():
            target_modules = ["query", "value", "key"]

        elif "t5" in model_type.lower():
            target_modules = ["q", "v", "k", "o"]

        elif "bloom" in model_type.lower():
            target_modules = ["query_key_value"]

        else:
            print(f"Auto-detection of target modules not supported for {moc

            target_modules = ["q_proj", "v_proj", "k_proj", "o_proj"]

    if getattr(self.model, "is_quantized", False):
        self.model         = prepare_model_for_kbit_training(self.model)

    task_type              = TaskType.CAUSAL_LM if isinstance(self.model, A

    lora_config            = LoraConfig(r               = rank,
                                        lora_alpha      = lora_alpha,
                                        target_modules  = target_modules,
                                        lora_dropout    = lora_dropout,
                                        bias            = "none",
                                        task_type       = task_type
                                        )
```

```
        print("Configuring LoRA...")

        with tqdm(total       = 100,
                  desc        = "Applying LoRA",
                  bar_format  = "{l_bar}{bar} | {n_fmt}/{total_fmt} [Time Left:
                  ) as pbar:

            model          = get_peft_model(self.model, lora_config)
            pbar.update(100)

        for name, param in self.model.named_parameters():

            if "lora" in name:

                param.requires_grad = True

        model.print_trainable_parameters()

        self.model         = model

        total_end_time     = datetime.now()
        total_time_taken   = total_end_time - total_start_time

        print("\nLoRA Applied Successfully!")
        print(f"Started at    : {total_start_time.strftime('%H:%M:%S')}")
        print(f"Finished at   : {total_end_time.strftime('%H:%M:%S')}")
        print(f"Total Time Taken: {str(total_time_taken)}\n")

        return self.model
```

> *Instruction Tuning:*

This code defines a class, `InstructionFineTuning`, for fine-tuning causal language models using Hugging Face's `Trainer` API. It supports LoRA-based parameter-efficient tuning and allows full fine-tuning with training configurations like batch size, learning rate, and epochs. The model is trained using an instruction-based dataset, leveraging mixed precision for efficiency. The class saves the fine-tuned model and tokenizer, logging the process with timestamps and progress bars.

```
# DEPENDENCIES

import torch
import datasets
from tqdm import tqdm
```

```
from datetime import datetime

import transformers
from transformers import Trainer
from transformers import TrainingArguments
from transformers import AutoModelForCausalLM
from transformers import DataCollatorForLanguageModeling

from ..fine_tuning_methods.lora_fine_tuning import LoRAFineTuning


class InstructionFineTuning:
    """
    A class to handle instruction-based fine-tuning of a language model using t
    """

    def __init__(self, model : AutoModelForCausalLM, tokenizer : transformers.P
        """
        Initializes the InstructionFineTuning class.

        Arguments:

            `model`        {AutoModelForCausalLM}  : The pre-trained language m

            `tokenizer`    {PreTrainedTokenizer}   : The tokenizer associated w

            `dataset`      {DatasetDict}           : The dataset prepared for f

            `device`       {str, optional}        : The device to run training
        """

        lora_adapter      = LoRAFineTuning(model = model)

        quantized_model   = lora_adapter.apply_lora(rank            = 8,
                                                    lora_alpha      = 16,
                                                    lora_dropout    = 0.1,
                                                    target_modules  = ["query",
                                                    )

        self.model                  = quantized_model
        self.tokenizer              = tokenizer
        self.prepared_dataset       = dataset
        self.device                 = device

        for param in self.model.parameters():
            param.requires_grad     = True

        if hasattr(self.model, "gradient_checkpointing_enable"):
            self.model.gradient_checkpointing_enable()

        if torch.cuda.is_available() and torch.__version__ >= "2.0":
            self.model = torch.compile(self.model)
```

```
        self.model.train()

    def apply_instruction_fine_tuning(self,
                                      output_dir     : str   = "./instruction_f
                                      batch_size     : int   = 8,
                                      learning_rate  : float = 5e-5,
                                      num_epochs     : int   = 3
                                      ) -> AutoModelForCausalLM:
        """
        Fine-tunes the model using an instruction-based dataset.

        Arguments:

            `output_dir`          {str, optional}    : Path to save the fine-tu

            `batch_size`          {int, optional}    : Batch size for training.

            `learning_rate`      {float, optional}   : Learning rate for traini

            `num_epochs`          {int, optional}    : Number of training epoch

        Raises:

            ValueError: If the model or prepared dataset is not loaded.

        Returns:

            model: The fine-tuned model.

        Functionality:

            - Configures training arguments for fine-tuning.
            - Uses the Hugging Face `Trainer` class for training.
            - Applies full fine-tuning (no parameter-efficient tuning).
            - Saves the fine-tuned model and tokenizer to the specified output

        """

        if self.model is None or self.prepared_dataset is None:
            raise ValueError("Model and prepared dataset must be loaded first")

        mixed_precision   = "bf16" if torch.cuda.is_available() and torch.cuda.

        print("Starting instruction fine-tuning...")

        training_args     = TrainingArguments(output_dir                  = ou
                                              per_device_train_batch_size  = ba
                                              learning_rate                = le
                                              num_train_epochs             = nu
                                              save_strategy                = "s
                                              save_steps                   = 50
                                              save_total_limit             = 2,
                                              logging_dir                  = f"
```

```
                                        logging_steps                = 50
                                        fp16                         = (m
                                        bf16                         = (m
                                        gradient_accumulation_steps  = 4,
                                        dataloader_num_workers       = 4,
                                        report_to                    = "r
                                        )

        trainer           = Trainer(model           = self.model,
                                    args            = training_args,
                                    train_dataset   = self.prepared_dataset["tr
                                    data_collator   = DataCollatorForLanguageMc


                                    compute_metrics = None,
                                    callbacks       = None,
                                    )

        total_start_time     = datetime.now()

        print(f"Fine-tuning started at: {total_start_time.strftime('%H:%M:%S')}

        self.model.train()

        print("Training model...")

        for epoch in range(1, num_epochs + 1):
            epoch_start_time = datetime.now()

            print(f"\nEpoch {epoch}/{num_epochs} started at {epoch_start_time.s

            with tqdm(total        = len(self.prepared_dataset["train"]) // bat
                      desc         = f"Epoch {epoch}/{num_epochs}",
                      bar_format   = "{l_bar}{bar} | {n_fmt}/{total_fmt} [Time
                      ncols        = 80
                      ) as pbar:

                trainer.train()

                pbar.update(pbar.total)

            epoch_end_time   = datetime.now()

            time_taken       = epoch_end_time - epoch_start_time

            print(f"Epoch {epoch} finished at {epoch_end_time.strftime('%H:%M:%

        total_end_time       = datetime.now()
        total_time_taken     = total_end_time - total_start_time

        print("\nFine-tuning Completed!")
        print(f"Total Training Time: {str(total_time_taken)}")
        print(f"Started at: {total_start_time.strftime('%H:%M:%S')}")
```

```
        print(f"Finished at: {total_end_time.strftime('%H:%M:%S')}")

        # Save the model
        self.model.save_pretrained(output_dir)
        self.tokenizer.save_pretrained(output_dir)
        print(f"Model saved to: {output_dir}")

        return self.model
```

Supervised Fine-Tuning:

This Python class, `SupervisedFineTuning`, is designed for fine-tuning a pre-trained causal language model using LoRA (Low-Rank Adaptation) and supervised fine-tuning (SFT) with the `SFTTrainer` from TRL. It initializes the model with LoRA-based parameter-efficient tuning, configures training parameters, and runs fine-tuning with gradient accumulation and mixed precision support. The training process is logged, and the fine-tuned model is saved at the end.

```
# DEPENDENCIES

import torch
import datasets
from tqdm import tqdm
from datetime import datetime

from trl import SFTTrainer

import transformers
from transformers import Trainer
from transformers import PreTrainedTokenizer
from transformers import TrainingArguments
from transformers import AutoModelForCausalLM
from transformers import DataCollatorForLanguageModeling

from ..fine_tuning_methods.lora_fine_tuning import LoRAFineTuning

class SupervisedFineTuning:
    """
    A class to handle supervised fine-tuning (SFT) of a language model using th
    """

    def __init__(self,
                 model       : AutoModelForCausalLM,
                 tokenizer   : PreTrainedTokenizer,
                 dataset     : datasets.DatasetDict,
```

```
                device      : str = "cuda" if torch.cuda.is_available() else "
                ) -> None:
        """
        Initializes the SupervisedFineTuning class.

        Arguments:

            `model`          {AutoModelForCausalLM}      : The pre-trained langu

            `tokenizer`      {PreTrainedTokenizer}       : The tokenizer associa

            `dataset`             {DatasetDict}          : The dataset prepared

            `device`             {str, optional}         : The device to run tra

        """

        lora_adapter            = LoRAFineTuning(model = model)

        quantized_model         = lora_adapter.apply_lora(rank            = 8,
                                                          lora_alpha      = 16,
                                                          lora_dropout    = 0.1
                                                          target_modules  = ["q
                                                          )

        self.model              = quantized_model
        self.device             = device
        self.tokenizer          = tokenizer
        self.prepared_dataset   = dataset

        self.model.train()

        for param in self.model.parameters():
            param.requires_grad = True

    def apply_supervised_fine_tuning(self,
                                     output_dir       : str   = "./supervised_f
                                     batch_size       : int   = 8,
                                     learning_rate    : float = 2e-5,
                                     num_epochs       : int   = 3
                                     ) -> AutoModelForCausalLM:
        """
        Fine-tunes the model using supervised fine-tuning (SFT) with the SFTTra

        Arguments:

            output_dir        {str, optional}      : Directory to save the fine

            batch_size        {int, optional}      : Batch size for training. D

            learning_rate    {float, optional}     : Learning rate for training

            num_epochs        {int, optional}      : Number of training epochs.
```

```
        Raises:

            ValueError  : If the model or prepared dataset is not loaded.

        Returns:

            model       : The fine-tuned model.

        Functionality:

            - Configures training arguments for supervised fine-tuning.
            - Uses `SFTTrainer` from TRL for training with packing enabled.
            - Applies gradient accumulation to handle larger batch sizes effect
            - Supports mixed precision training (`fp16`) if running on CUDA.
            - Saves the fine-tuned model and tokenizer to the specified output

        """
        if self.model is None or self.prepared_dataset is None:
            raise ValueError("Model and prepared dataset must be loaded first")

        self.prepared_dataset["train"] = self.prepared_dataset["train"].with_fo

        print("Starting supervised fine-tuning...")

        training_args      = TrainingArguments(output_dir                  = o
                                               per_device_train_batch_size  = b
                                               gradient_accumulation_steps  = 4
                                               learning_rate                = l
                                               num_train_epochs             = n
                                               save_strategy                = "
                                               save_total_limit             = 2
                                               logging_dir                  = f
                                               logging_steps                = 1
                                               fp16                         = F
                                               remove_unused_columns        = F
                                               report_to                    = "
                                               optim                        = "
                                               lr_scheduler_type            = "
                                               warmup_steps                 = 0
                                               )

        trainer            = SFTTrainer(model           = self.model,
                                        args            = training_args,
                                        train_dataset   = self.prepared_datase
                                        eval_dataset    = self.prepared_datase
                                        data_collator   = DataCollatorForLangu

                                        )

        total_start_time   = datetime.now()

        print(f"Fine-tuning started at: {total_start_time.strftime('%H:%M:%S')}
```

```
        print("Training model...")

        for epoch in range(1, num_epochs + 1):
            epoch_start_time = datetime.now()

            print(f"\nEpoch {epoch}/{num_epochs} started at {epoch_start_time.s

            with tqdm(total       = len(self.prepared_dataset["train"]) // batc
                      desc        = f"Epoch {epoch}/{num_epochs}",
                      bar_format  = "{l_bar}{bar} | {n_fmt}/{total_fmt} [Time L
                      ncols       = 80
                      ) as pbar:

                trainer.train()

                pbar.update(pbar.total)

            epoch_end_time = datetime.now()
            time_taken     = epoch_end_time - epoch_start_time

            print(f"Epoch {epoch} finished at {epoch_end_time.strftime('%H:%M:%

        total_end_time     = datetime.now()
        total_time_taken   = total_end_time - total_start_time

        print("\nSupervised Fine-tuning Completed!")
        print(f"Total Training Time: {str(total_time_taken)}")
        print(f"Started at: {total_start_time.strftime('%H:%M:%S')}")
        print(f"Finished at: {total_end_time.strftime('%H:%M:%S')}")

        self.model.save_pretrained(output_dir)
        self.tokenizer.save_pretrained(output_dir)

        print(f"Model saved to: {output_dir}")

        return self.model
```

STEP — 4 -> Inference of Fine-Tuned Language Model

This `InferenceEngine` class handles text generation using a fine-tuned transformer model. It initializes the model and tokenizer, moves them to the specified device (CPU or GPU), and generates text from a prompt using configurable parameters like `max_length`, `temperature`, and `num_return_sequences`. It includes logging for inference start/end time and returns the generated sequences.

```
# DEPENDENCIES

import time
import torch
from tqdm import tqdm
from datetime import datetime

from transformers import PreTrainedModel
from transformers import PreTrainedTokenizer

class InferenceEngine:
    """
    A class for generating text using a fine-tuned transformer model.

    This class facilitates inference by generating text sequences based on a gi
    It supports configuration of generation parameters such as `max_length`, `t
    """

    def __init__(self, model : PreTrainedModel, tokenizer : PreTrainedTokenizer
        """
        Initializes the InferenceEngine with a fine-tuned model and tokenizer.

        Arguments:
            model      {PreTrainedModel}    : The fine-tuned transformer model
            tokenizer  {PreTrainedTokenizer}: The tokenizer corresponding to th
            device     {str, optional}      : The device to run inference on ("

        Raises:
            ValueError: If the model or tokenizer is not provided.
        """
        if model is None or tokenizer is None:
            raise ValueError("Both model and tokenizer must be provided.")

        self.model      = model
        self.tokenizer  = tokenizer
        self.device     = device
        self.model.to(self.device)


    def inference(self, prompt : str, max_length : int = 512, temperature : int
        """
        Generate text using the fine-tuned model.

        Arguments:

            `prompt`                    {str}             : The input prompt

            `max_length`              {int, optional}       : The maximum leng

            `temperature`           {float, optional}       : Sampling tempera
```

```
        `num_return_sequences`    {int, optional}        : The number of ge

    Returns:

        list: A list of generated text sequences.

    Raises:

        ValueError: If the model or tokenizer is not loaded.

    """

    if self.model is None or self.tokenizer is None:
        raise ValueError("Model and tokenizer must be loaded first")

    print(f"Running inference with prompt: {prompt[:200]}...")

    self.model.to(self.device)

    inputs           = self.tokenizer(prompt, return_tensors = "pt").to(sel

    start_time       = time.time()
    start_dt         = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    print(f"[START] Inference started at: {start_dt}")

    generated_texts = []

    with torch.no_grad():

        for _ in tqdm(range(num_return_sequences), desc="Generating Text",

            outputs  = self.model.generate(inputs.input_ids,
                                           max_length             = max_ler
                                           temperature            = tempera
                                           num_return_sequences   = num_ret
                                           do_sample              = tempera
                                           pad_token_id           = self.to
                                           )

            decoded  = self.tokenizer.decode(outputs[0], skip_special_toker

            generated_texts.append(decoded)

            # generated_texts  = [self.tokenizer.decode(output, skip_specia

    end_time         = time.time()
    end_dt           = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    duration         = end_time - start_time

    print(f"\n[END] Inference completed at: {end_dt}")
    print(f"[SUMMARY] Total time taken: {duration:.2f} seconds")
    print(f"[SUMMARY] Number of sequences generated: {num_return_sequences}
```

```
        return generated_texts
```

STEP — 5 -> Evaluation of the Fine-Tuned Language Model

The `ModelEvaluator` class offers a streamlined way to assess the performance of fine-tuned language models using popular NLP metrics: **Perplexity**, **Coherence**, **BLEU Score, ROUGE** and **METE**. Designed to work with Hugging Face models and datasets, it enables quick and structured evaluation on a test set.

- **Perplexity** measures how confidently the model predicts the next token
- **Coherence** captures the average log probability of actual next tokens, reflecting fluency.
- **BLEU score** evaluates generated text against reference outputs for similarity.
- **ROUGE** assesses n-gram and sequence overlap, often used for summarization tasks.
- **METEOR** incorporates synonymy and semantic similarity, offering a more nuanced evaluation than BLEU.

With just a few lines of code, this class provides both individual and combined metric reports — useful for research, development, and model comparison.

```
# DEPENDENCIES

import torch
import evaluate
import datasets
from tqdm import tqdm
from datetime import datetime
from tabulate import tabulate

import nltk
nltk.download('punkt')
nltk.download('wordnet')
from nltk.translate.bleu_score import corpus_bleu

import transformers
from transformers import PreTrainedTokenizer

class ModelEvaluator:
```

```
    """
    A class for evaluating the performance of a fine-tuned model using differer

    This class supports metrics such as perplexity, coherence, and BLEU score t
    on a given test dataset.
    """

    def __init__(self, model : transformers.PreTrainedModel, tokenizer : PreTra
        """
        Initializes the ModelEvaluator with a model, tokenizer, and evaluation

        Arguments:

            `model`           {transformers.PreTrainedModel}         : The fine

            `tokenizer`      {transformers.PreTrainedTokenizer}      : The toke

            `device`                  {str, optional}                : The devi

            `prepared_dataset`        {dict, optional}               : The data

        """
        self.model             = model
        self.device            = device
        self.tokenizer         = tokenizer
        self.prepared_dataset  = prepared_dataset if prepared_dataset else {}

    def inference(self, input_text : str) -> list:
        """
        Generates model predictions for a given input text.

        This method tokenizes the input text, runs the model in inference mode
        to generate output tokens, and decodes the generated tokens into
        human-readable text.

        Arguments:

            input_text          {str}         : The input prompt or text to ger

        Returns:

            List[str]          {list}         : A list containing the decoded c

        Notes:

            - Uses `max_new_tokens = 100` by default; you can adjust it based c
            - The model is run in no-grad mode to disable gradient computation
        """

        inputs        = self.tokenizer(input_text, return_tensors="pt").to(self

        with torch.no_grad():
            outputs   = self.model.generate(**inputs, max_new_tokens = 100)
```

```
        return self.tokenizer.batch_decode(outputs, skip_special_tokens = True)


    def evaluate(self, test_dataset : datasets.DatasetDict = None, metric : str
        """
        Evaluate the fine-tuned model using specified evaluation metrics.

        Arguments:

            `test_dataset`    {optional}       : The dataset to evaluate the mc
                                                 if available; otherwise, defau

            `metric`        {str, optional}    : The evaluation metric to compu

                                                 - "perplexity": Measures how w

                                                 - "coherence": Assesses the li

                                                 - "bleu": Evaluates text simil

                                                 - "all": Computes all availabl
                                                 Defaults to "perplexity".

        Returns:

            dict: A dictionary containing evaluation results for the specified

        Raises:

            ValueError: If the model or tokenizer is not loaded before evaluati

        Notes:

            - Perplexity is computed using the average loss over token sequence
            - Coherence is measured using the log probability of the actual nex
            - BLEU score compares generated outputs with reference texts.
            - The evaluation may use a subset of the dataset (e.g., first 100 e
        """

        if self.model is None or self.tokenizer is None:
            raise ValueError("Model and tokenizer must be loaded first")

        dataset                         = test_dataset or self.prepared_dataset[

        print(f"Evaluating model with {metric} metric...")

        results                         = {}

        self.model.to(self.device)

        start_time                      = datetime.now()
        print(f"[START] Evaluation started at: {start_time.strftime('%Y-%m-%d %
```

```
        if metric == "perplexity" or metric == "all":

            print("Calculating perplexity...")

            self.model.eval()
            total_loss                  = 0
            total_tokens                = 0

            with torch.no_grad():

                for i in tqdm(range(0, len(dataset), 8), desc = "Perplexity Eva

                    batch           = dataset[i:i+8]
                    inputs          = {k: torch.tensor(v).to(self.device) for k,

                    outputs         = self.model(**inputs)
                    loss            = outputs.loss

                    total_loss    += loss.item() * inputs["input_ids"].size(0)
                    total_tokens  += inputs["input_ids"].size(0) * inputs["inpu

            perplexity                  = torch.exp(torch.tensor(total_loss / to

            results["perplexity"]       = perplexity.item()

        if metric == "coherence" or metric == "all":

            print("Calculating coherence...")

            self.model.eval()
            total_coherence             = 0

            with torch.no_grad():

                for i in tqdm(range(min(100, len(dataset))), desc = "Coherence

                    sample              = dataset[i]
                    input_ids           = torch.tensor(sample["input_ids"]).unsq

                    outputs             = self.model(input_ids)
                    logits              = outputs.logits

                    shift_logits        = logits[:, :-1, :].contiguous()
                    shift_labels        = input_ids[:, 1:].contiguous()

                    log_probs           = torch.log_softmax(shift_logits, dim=-1
                    token_log_probs     = log_probs.gather(-1, shift_labels.unsq

                    coherence           = token_log_probs.mean().item()
                    total_coherence    += coherence

            results["coherence"]        = total_coherence / min(100, len(dataset
```

```
        if metric == "bleu" or metric == "all":

            print("Calculating BLEU score...")

            references                  = []
            candidates                  = []

            for i in tqdm(range(min(100, len(dataset))), desc = "BLEU Evaluatic

                sample                  = dataset[i]

                input_text              = self.tokenizer.decode(sample["input_ic

                reference               = self.tokenizer.decode(sample["input_ic
                references.append([reference.split()])

                output                  = self.inference(input_text)[0]
                candidates.append(output.split())

            bleu                        = corpus_bleu(references, candidates)
            results["bleu"]             = bleu

        if metric == "rouge" or metric == "all":

            print("Calculating ROUGE score...")

            rouge                       = evaluate.load("rouge")

            references                  = []
            predictions                 = []

            for i in tqdm(range(min(100, len(dataset))), desc = "ROUGE Evaluati

                sample                  = dataset[i]
                input_text              = self.tokenizer.decode(sample["input_ic
                reference               = self.tokenizer.decode(sample["input_ic
                output                  = self.inference(input_text)[0]

                references.append(reference)
                predictions.append(output)

            rouge_result                = rouge.compute(predictions=predictions

            for key, value in rouge_result.items():
                results[f"rouge_{key}"] = value

        if metric == "meteor" or metric == "all":

            print("Calculating METEOR score...")

            meteor                      = evaluate.load("meteor")
```

```
        references                  = []
        predictions                 = []

        for i in tqdm(range(min(100, len(dataset))), desc = "METEOR Evaluat

            sample                  = dataset[i]
            input_text              = self.tokenizer.decode(sample["input_
            reference               = self.tokenizer.decode(sample["input_
            output                  = self.inference(input_text)[0]

            references.append(reference)
            predictions.append(output)

        meteor_result               = meteor.compute(predictions = predict
        results["meteor"]           = meteor_result["meteor"]

    print("Evaluation complete:")

    end_time                    = datetime.now()

    print(f"\n[END] Evaluation finished at: {end_time.strftime('%Y-%m-%d %H
    print(f"[TIME] Total time taken: {str(end_time - start_time)}\n")

    # for k, v in results.items():
    #     print(f"{k}: {v}")

    print("Evaluation Summary:\n")
    table_data                  = [[metric_name, f"{score:.4f}"] for met

    print(tabulate(table_data,
                   headers  = ["Metric", "Score"],
                   tablefmt = "grid"
                   ))

    return results
```

Interpretation of Evaluation Metrics

Metric	Interpretation	Good Value	Bad Value
Perplexity	Measures how well the model predicts the next word. Lower is better.	**< 30** → Indicates strong language modeling and confident predictions.	**> 100** → Indicates poor predictions and model uncertainty.
Coherence	Average log-probability of the correct next token. Higher is better.	**> -2.0** → Indicates fluent, contextually relevant outputs.	**< -5.0** → Indicates poor or incoherent predictions.
BLEU Score	Compares generated text to reference text. Higher is better (between 0 and 1).	**> 0.5** (or **> 50%**) → Indicates strong similarity to references.	**< 0.2** (or **< 20%**) → Indicates weak similarity; often unacceptable.
ROUGE Score	Measures overlap between model output and reference text (n-grams, sequences, etc.). Higher is better.	**> 0.5** → Good overlap, useful in summarization/translation tasks.	**< 0.3** → Poor overlap; may indicate off-topic or missing content.
METEOR Score	Considers synonymy and exact word matches between generated and reference text. More semantic than BLEU. Higher is better (0 to 1 scale).	**> 0.6** → High-quality, semantically accurate generation.	**< 0.3** → Poor semantic alignment, often lacks meaning or fluency.

Notes:

- **Perplexity** is especially useful in language modeling tasks (e.g., next-token prediction).
- **Coherence** is great for checking how logically the model continues text.
- **BLEU** is mostly used in generation tasks like translation or summarization.
- **ROUGE** is especially important for tasks like **summarization** and **paraphrasing**.
- **METEOR** captures synonym matches and is better at evaluating **semantic similarity** than BLEU.

Conclusion

Fine-tuning Large Language Models (LLMs) enables businesses to tailor AI capabilities to their unique needs, enhancing brand identity and customer engagement. By customizing pre-trained models with domain-specific data, companies can develop AI systems that resonate with their target audience, ensuring more relevant and personalized interactions. This strategic approach not only differentiates a brand in the competitive market but also fosters deeper connections with customers through AI-driven solutions that reflect the brand's values and communication style.

GIT REPOSITORY:

GitHub - priyam-hub/LLM-Fine-Tuning-Pipeline: A comprehensive pipeline for Different Fine-Tuning...

A comprehensive pipeline for Different Fine-Tuning Methods for Large Language Models with optimized performance and...

github.com

If you love reading this follow me on Medium, LinkedIn

Large Language Models | Llm Finetuning | Brand Endorsement | Roadmaps | Artificial Intelligence

Edit profile

Written by Priyam Pal

1 Follower · 2 Following

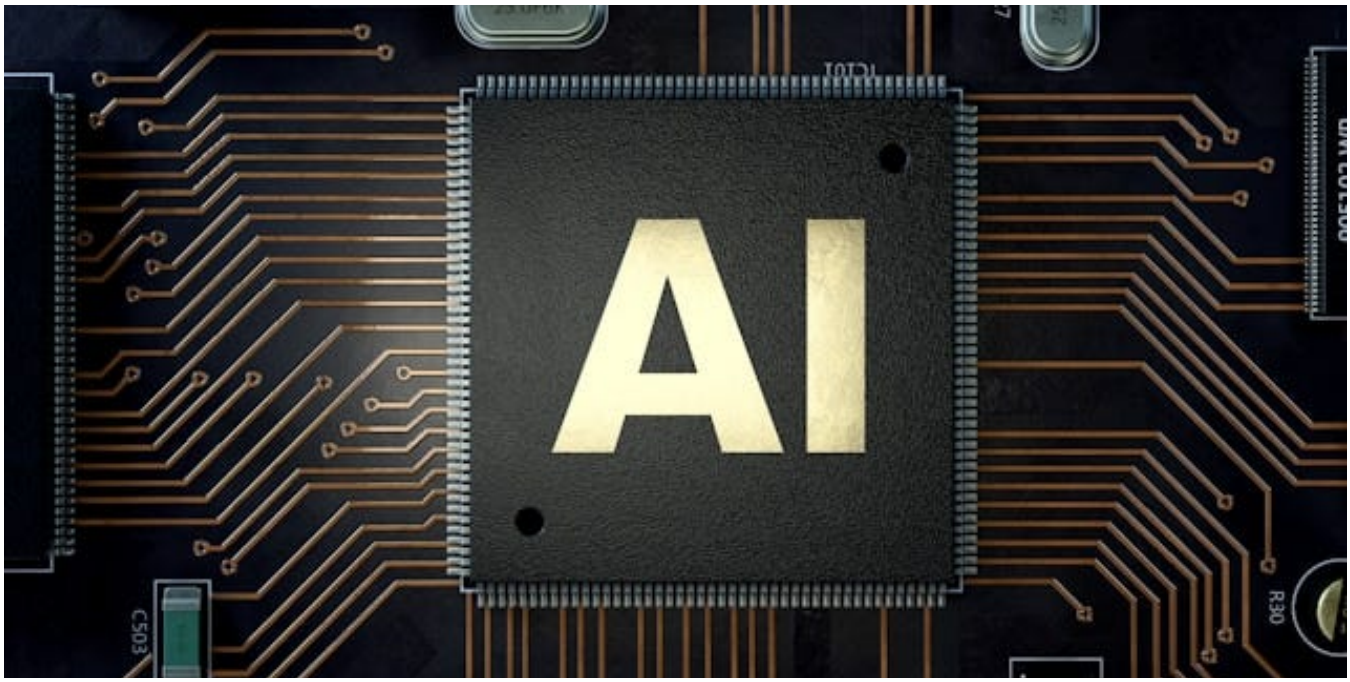
AI and Data Science Engineer

No responses yet

Priyam Pal

What are your thoughts?

Recommended from Medium

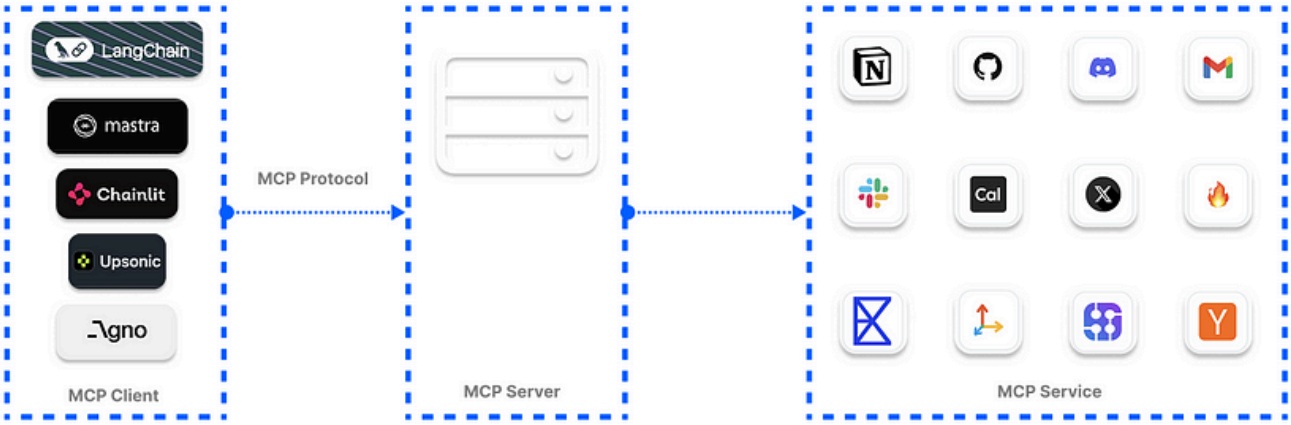


In InkWell Atlas by Ronwriter

Has AI become centralized?

Artificial Intelligence (AI) has exploded in the last two and a half years, raising concerns for Simon Kim and others in the AI sector. Kim...

3d ago 82 1



Amos Gyamfi

The Top 7 MCP-Supported AI Frameworks

Create AI apps with Python and Typescript frameworks that leverage MCP servers to provide context to LLMs.

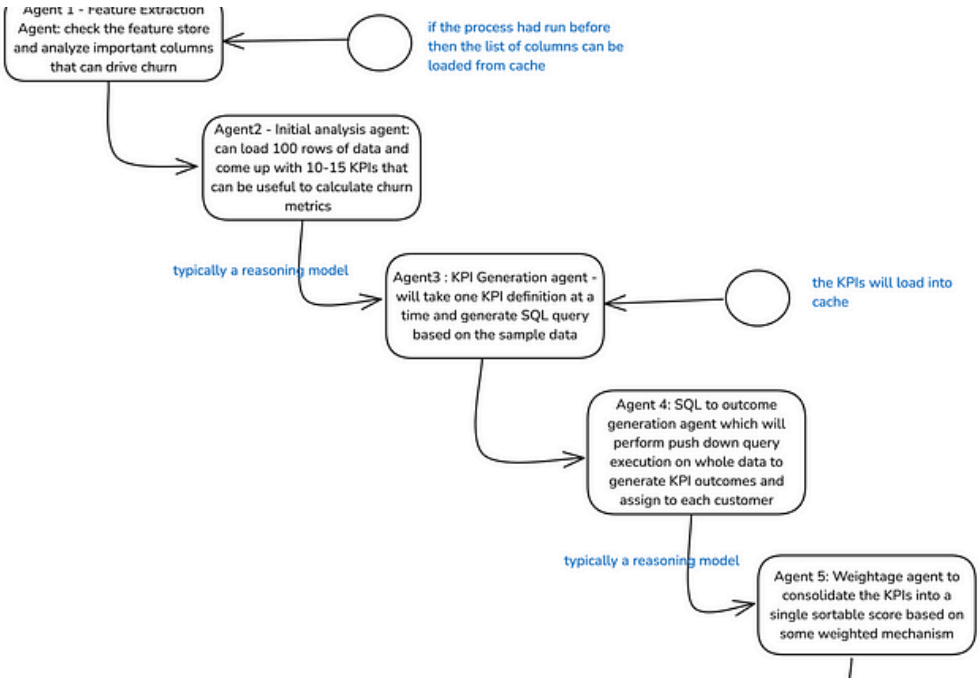
6d ago 639 16

In System Weakness by Lovish Kumar

🔥 “Securing API Gateways in Spring Boot Microservices: Best Practices You Must Know!” 🔥

API gateways are the central entry point for microservices, handling authentication, authorization, request routing, and security. However...

Feb 26 1

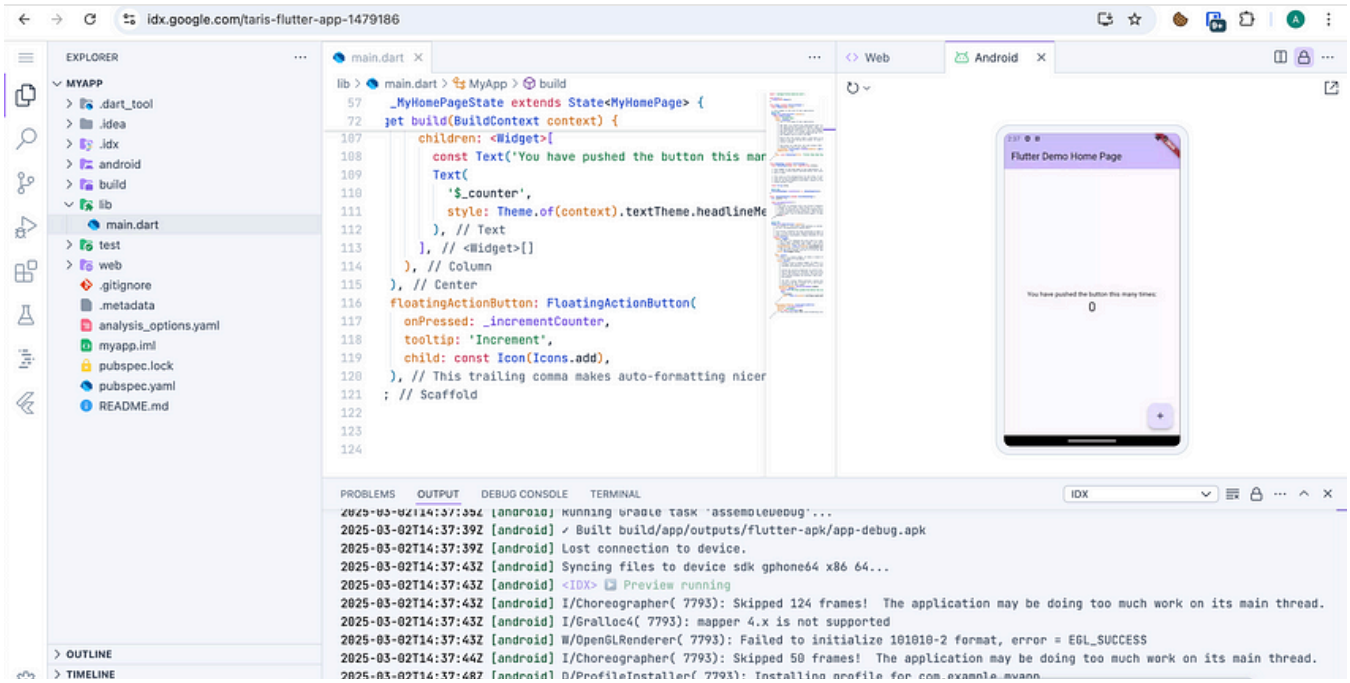


Nayan Paul

Multi Agent Solution for Customer Churn Prediction using Gen AI

This blog has below 3 sections :

Mar 31 267 14



In Coding Beauty by Tari Ibaba

This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.

Mar 12 3.8K 203

Panos Sakalakis

With this free app, you can create unlimited AI images, videos, and audio

I found this free desktop app, and I can’t stop using it. It’s free, robust, and fairly easy to use.

Mar 31 688 11

See more recommendations