2025 EDITION

FREE

AI Engineering

System Design Patterns for LLMs, RAG and Agents

Daily Dose of Data Science

Akshay Pachaar & Avi Chawla

DailyDoseofDS.com

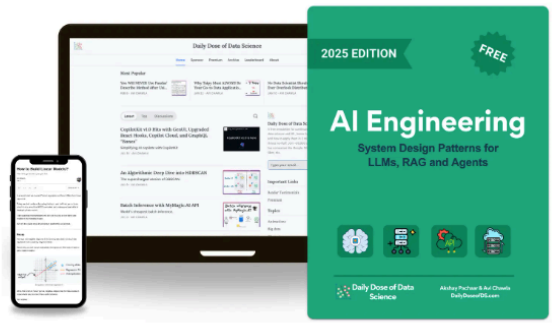
How to make the most out of this book and your time?

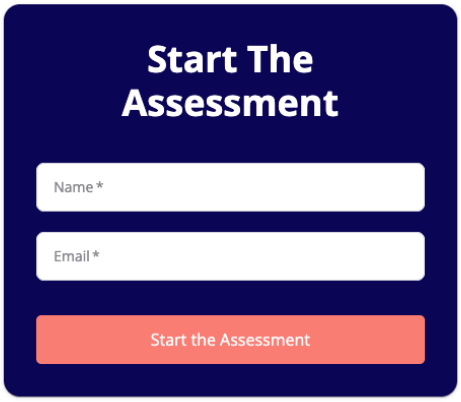
The reading time of this book is about 20 hours. But not all chapters will be of relevance to you. This 2-minute assessment will test your current expertise and recommend chapters that will be most useful to you.

Are you prepared for a career in AI Engineering?

Answer 15 yes/no questions, and we'll email you the list of chapters that you must read to improve your AI Engineering skillset.

Start the Assessment Below!





Scan the QR code below or open this link to start the assessment. It will only take 2 minutes to complete.

https://bit.ly/ai-engg-assessment

AI Engineering

Table of contents

LLMs

What is an LLM?

Imagine someone begins a sentence:

"Once upon a..."

You naturally think "time."

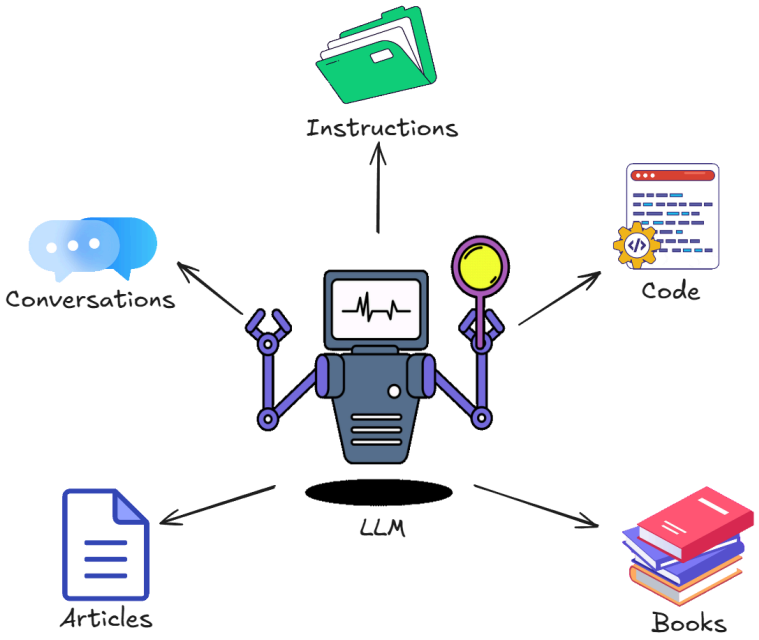
Or they say:

"The capital of France is..."

You immediately think "Paris."

This simple act of predicting what comes next is the foundation of how large language models(LLMs) operate.

They learn to make these predictions by reading enormous amounts of text: books, articles, scientific papers, code, conversations, and instructions.

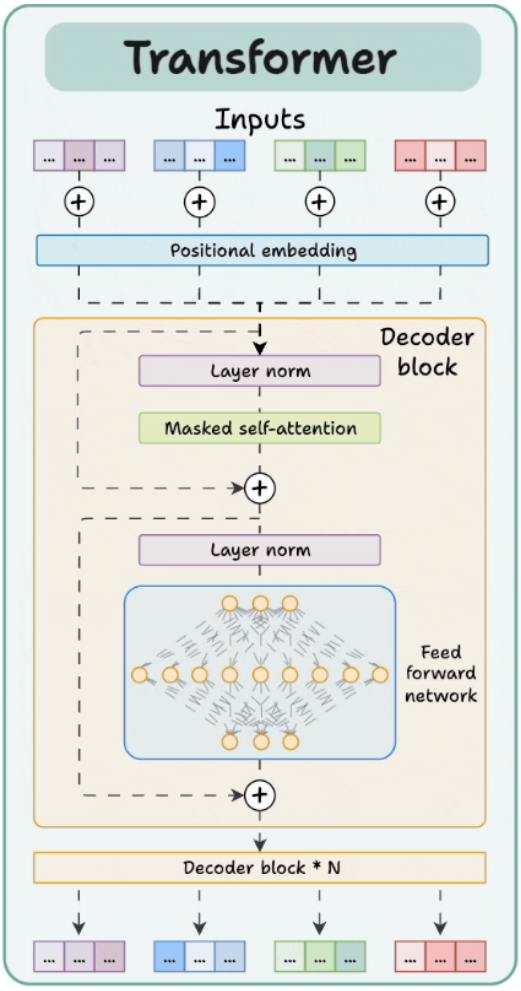


With enough exposure, the model becomes remarkably good at continuing any piece of text in a coherent, meaningful way.

At the technical level, an LLM processes text in small units called tokens. A token may be a word, part of a word or even punctuation.

The model looks at the tokens so far and predicts the next one. Repeating this process generates full answers, explanations, or code.

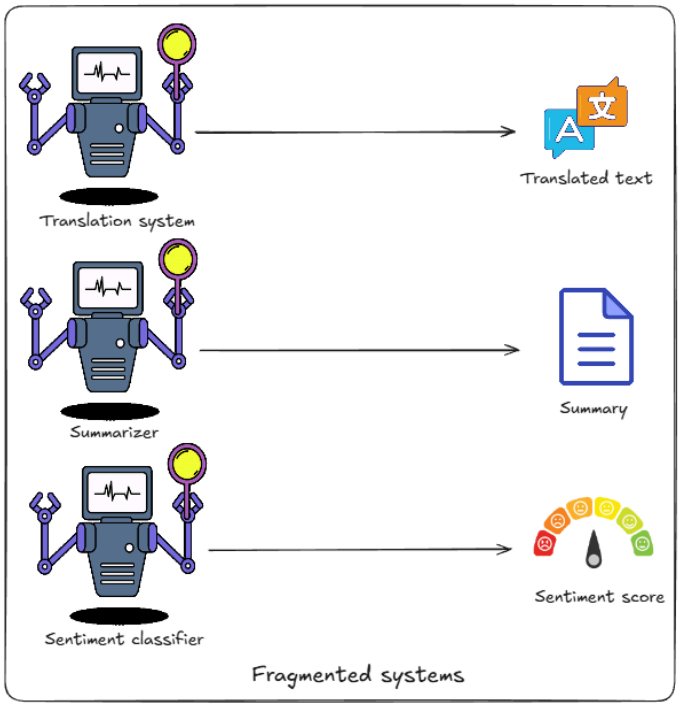
Everything an LLM does from summarizing a document, generating a function or explaining a concept emerges from choosing the next token that best fits the patterns it has learned.



To formalise, a large language model is a Transformer-based neural network trained on massive text corpora to predict the next token in a sequence and through this process acquires the ability to understand, generate and reason with human language.

Need for LLMs

Before LLMs, AI systems were built for specific tasks.

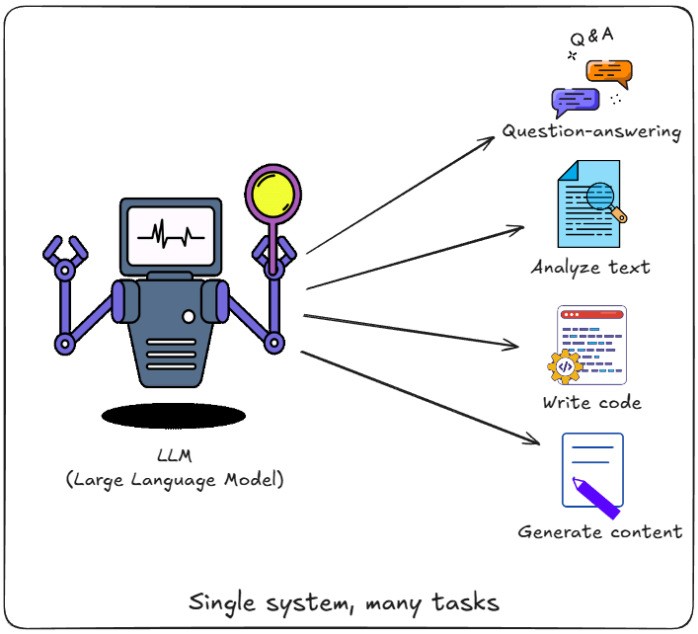


- A translation system handled only translation.
- A summarizer knew only summarization.
- A sentiment classifier recognized only sentiment.

Each new problem required a new model and a new pipeline. This created a fragmented landscape that didn't scale.

LLMs changed this.

By learning the general structure of language across millions of domains, one model could suddenly perform many tasks without being explicitly programmed for each one.



Language naturally encodes reasoning steps, factual knowledge, explanations and communication patterns. Training a model on large enough text collections allowed it to internalize these patterns and apply them across tasks.

As a result, a single system could now answer questions, write code, analyze text and more simply by predicting the continuation of your input.

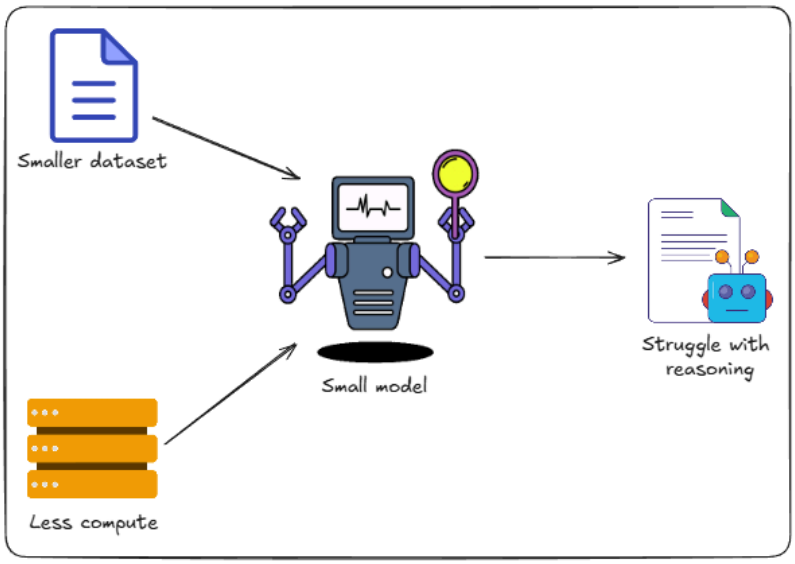
What makes an LLM 'large' ?

When we call a language model "large," we are referring to its scale:

- Number of parameters it contains
- Amount of data it has been trained on
- Compute used to train it

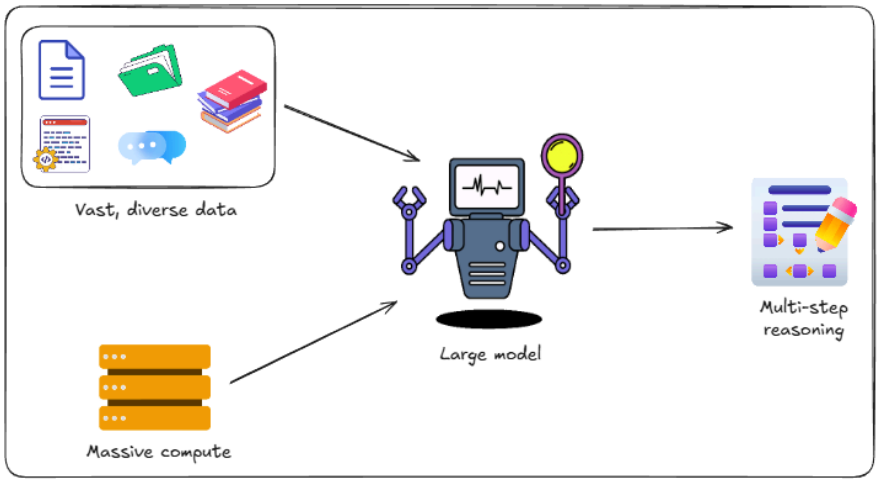
Parameters are the internal values that the model adjusts during training. Each parameter represents a small piece of the patterns the model has learned.

Earlier language models were much smaller and could only capture surface-level text patterns.



They could mimic style but struggled with tasks that required reasoning, abstraction or generalization.

As researchers increased model size, dataset diversity and training compute, a clear shift appeared.



Larger models began to follow detailed instructions, perform multi-step reasoning and solve problems they had never encountered directly in training.

This wasn't the result of adding new rules or programming specific behaviors. It emerged naturally from giving the model enough capacity to learn deeper relationships in language.

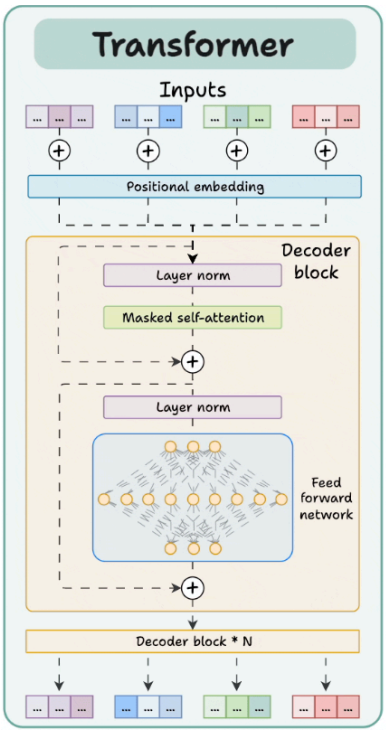
This effect held consistently: models with more parameters, trained on broader data, produced more reliable, coherent and adaptable outputs.

In practical terms, the "large" in large language model is what enables these capabilities.

How are LLMs built?

Before an LLM can be trained, it needs an architecture that can process text, learn patterns and scale across large datasets.

This architecture is built from several core components that work together to turn raw text into structured representations the model can learn from.



Transformer

At the center of modern LLMs is the Transformer.

A Transformer is designed to look at all tokens in the input at once and identify which parts of the text are most relevant to each other.

This lets the model follow long sentences, track references, and understand relationships that appear far apart in the sequence.

Tokenization

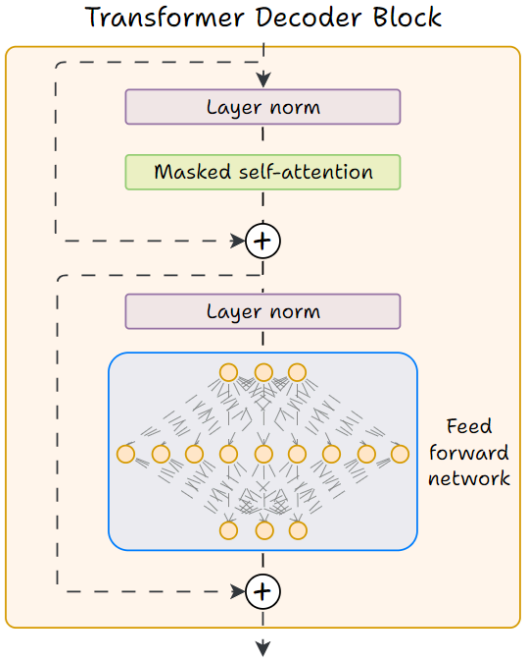
Text is first broken into tokens. A token may be a word or part of a word, depending on how common it is.

This approach keeps the vocabulary manageable and allows the model to handle any language input.

These tokens are then mapped to numerical representations so the model can work with them.

Transformer Layers

The model contains many Transformer layers stacked on top of each other.

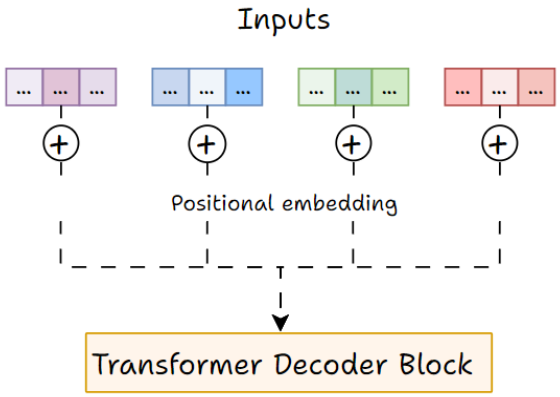


Each layer refines the model's understanding by comparing tokens, attending to important parts of the input, and updating their representations.

As the sequence moves through these layers, the model builds a deeper view of the text.

Positional Encoding

Transformers do not naturally know the order in which tokens appear.



To provide this information, positional encodings are added to the token representations.

These encodings give the model a sense of sequence, enabling it to interpret ordered structures such as sentences, lists, or code.

Parameters

Inside the architecture are parameters - the values the model adjusts during training.

A large LLM contains billions of these parameters.

They store the patterns the model learns from text and form the basis for its ability to understand and generate language.

Distributed Training Setup

Because these models are too large for a single machine, they are trained across many GPUs in parallel.

The model's parameters, computations and training data are distributed so the system can process massive datasets and update billions of parameters efficiently.

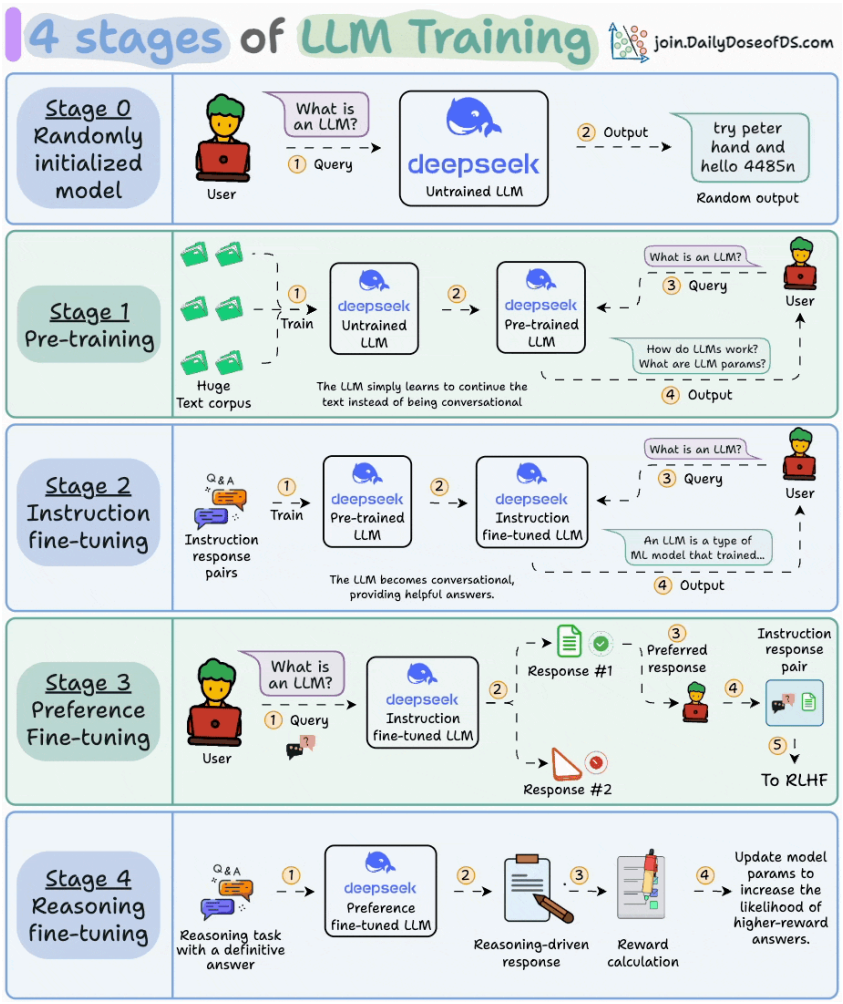
How to train LLM from scratch?

After the model is built, the next step is to train it so it can understand language and handle tasks that appear in real-world use cases.

We'll cover:

- Pre-training
- Instruction fine-tuning
- Preference fine-tuning
- Reasoning fine-tuning

The visual provides a clear summary of how these stages fit together.



0) Randomly initialized LLM

At this point, the model knows nothing.

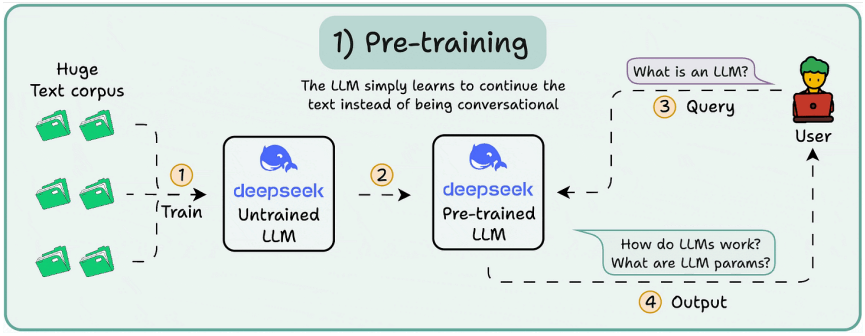
You ask, "What is an LLM?" and get gibberish like "try peter hand and hello 448Sn".

It hasn't seen any data yet and possesses just random weights.

1) Pre-training

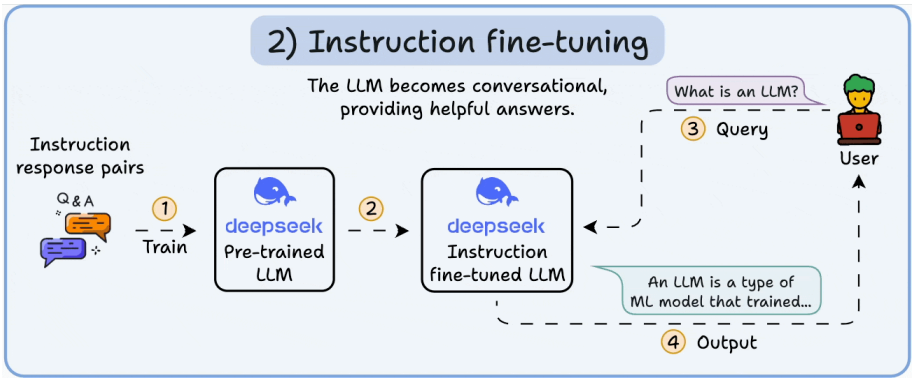
This stage teaches the LLM the basics of language by training it on massive corpora to predict the next token. This way, it absorbs grammar, world facts, etc.

But it's not good at conversation because when prompted, it just continues the text.



2) Instruction fine-tuning

To make it conversational, we do Instruction Fine-tuning by training on instruction-response pairs. This helps it learn how to follow prompts and format replies.



Now it can:

- Answer questions
- Summarize content
- Write code, etc.

At this point, we have likely:

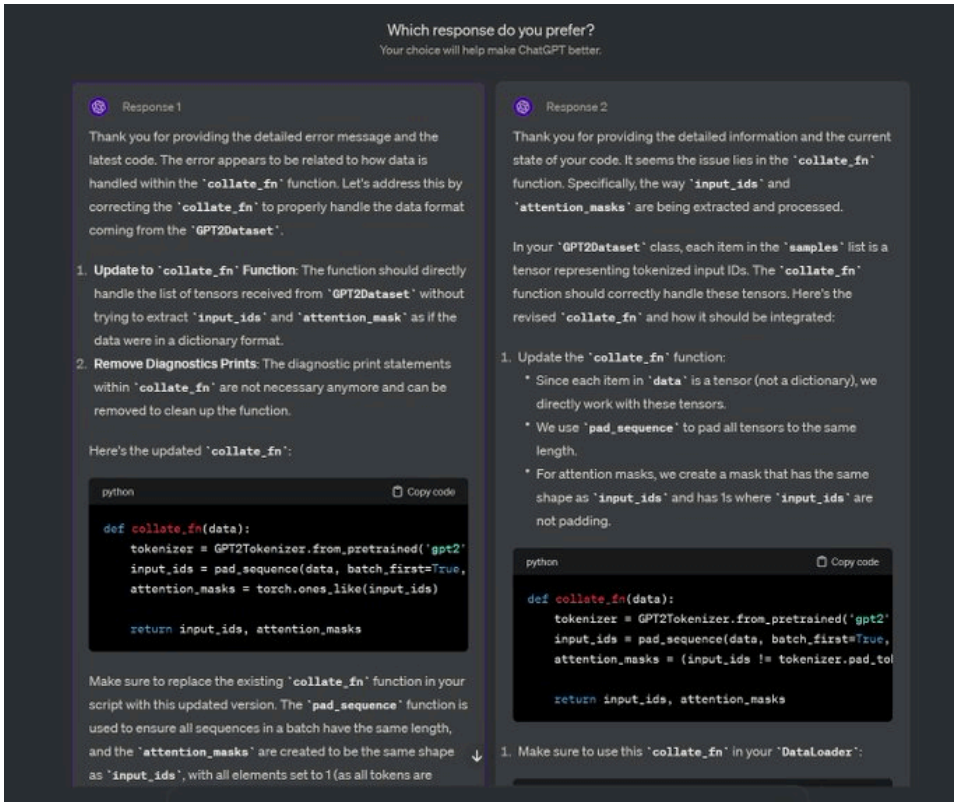
- Utilized the entire raw internet archive and knowledge.
- The budget for human-labeled instruction response data.

So what can we do to further improve the model?

We enter into the territory of Reinforcement Learning (RL).

3) Preference fine-tuning (PFT)

You must have seen this screen on ChatGPT where it asks: Which response do you prefer?



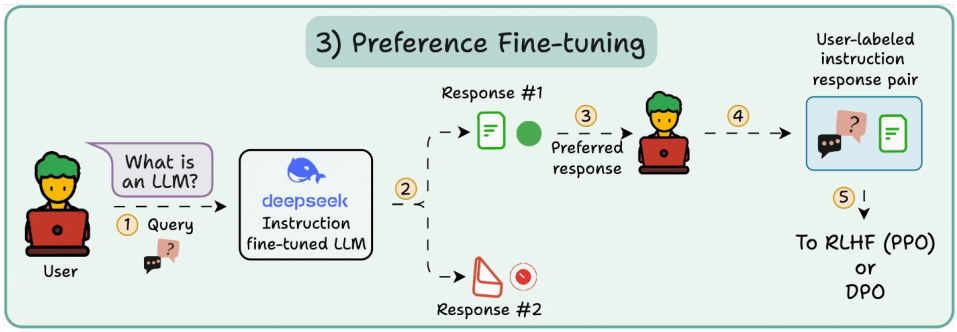
That's not just for feedback, but it's valuable human preference data.

OpenAI uses this to fine-tune their models using preference fine-tuning.

In PFT:

The user chooses between 2 responses to produce human preference data.

A reward model is then trained to predict human preference and the LLM is updated using RL.



The above process is called RLHF (Reinforcement Learning with Human Feedback), and the algorithm used to update model weights is called PPO.

It teaches the LLM to align with humans even when there's no "correct" answer.

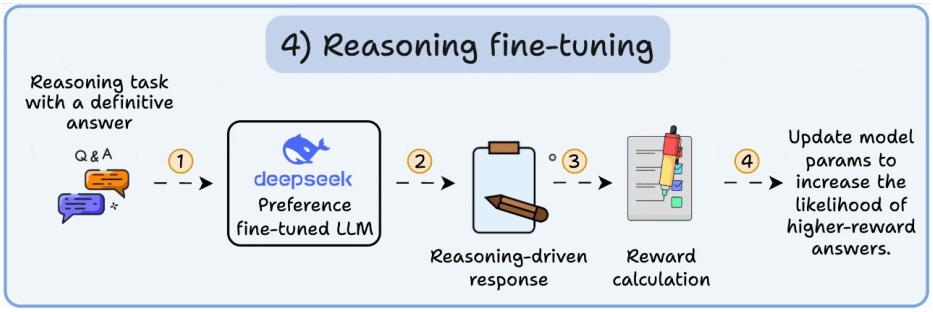
But we can improve the LLM even more.

4) Reasoning fine-tuning

In reasoning tasks (maths, logic, etc.), there's usually just one correct response and a defined series of steps to obtain the answer.

So we don't need human preferences, and we can use correctness as the signal.

This is called reasoning fine-tuning



Steps:

- The model generates an answer to a prompt.
- The answer is compared to the known correct answer.
- Based on the correctness, we assign a reward.

This is called Reinforcement Learning with Verifiable Rewards. GRPO by DeepSeek is a popular technique for this.

Those were the 4 stages of training an LLM.

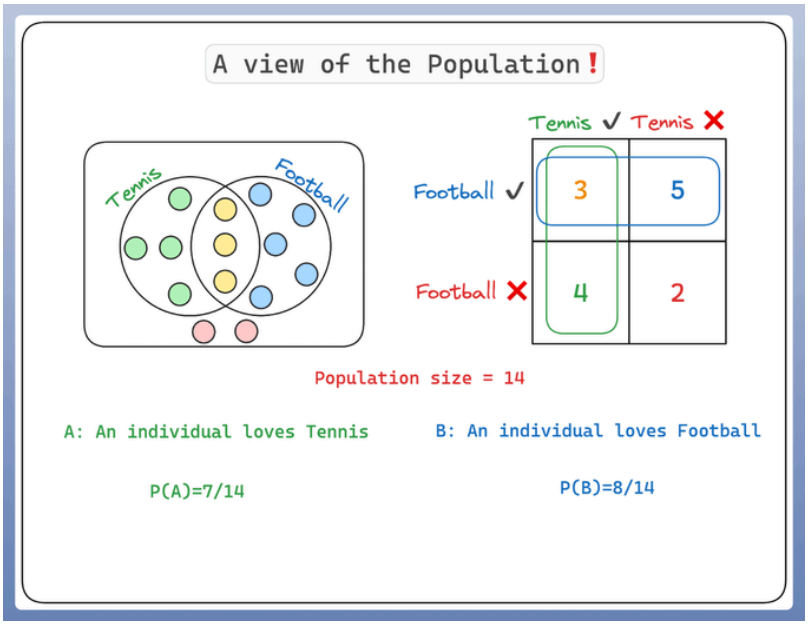
- Start with a randomly initialized model.
- Pre-train it on large-scale corpora.
- Use instruction fine-tuning to make it follow commands.
- Use preference & reasoning fine-tuning to sharpen responses.

How do LLMs work?

Let's understand how exactly LLMs work and generate text.

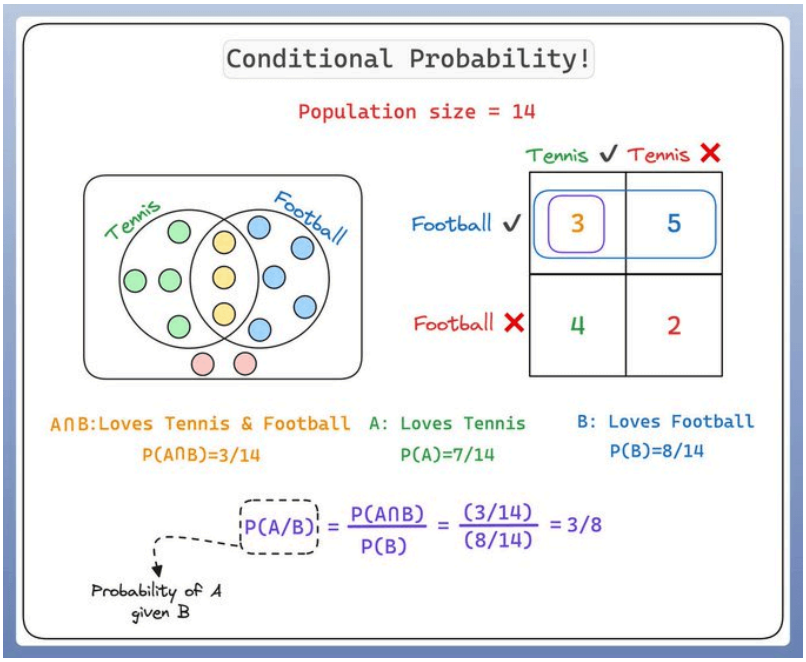
Before diving into LLMs, we must understand conditional probability.

Let's consider a population of 14 individuals:



- Some of them like Tennis
- Some like Football
- A few like both
- And few like none

Conditional probability is a measure of the probability of an event given that another event has occurred.



If the events are A and B, we denote this as P(A|B).

This reads as "probability of A given B"

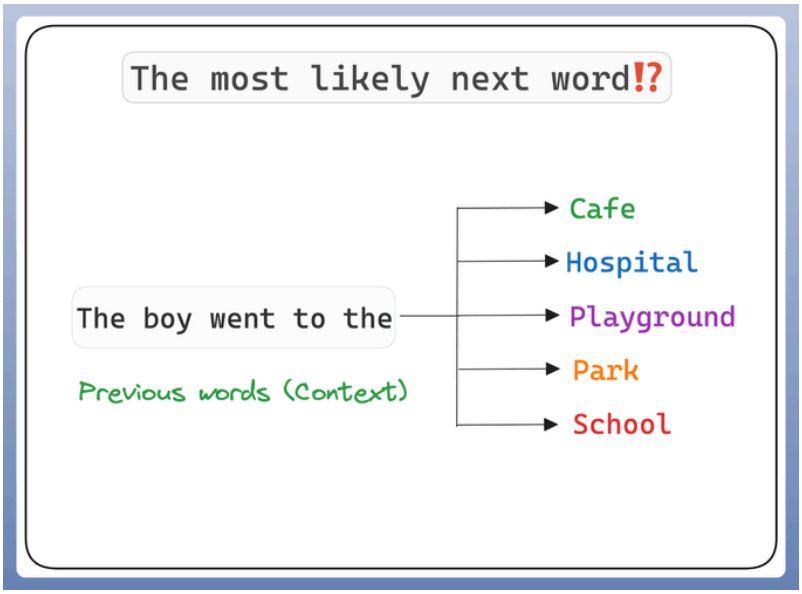
For instance, if we're predicting whether it will rain today (event A), knowing that it's cloudy (event B) might impact our prediction.

As it's more likely to rain when it's cloudy, we'd say the conditional probability P(A|B) is high.

That's conditional probability!

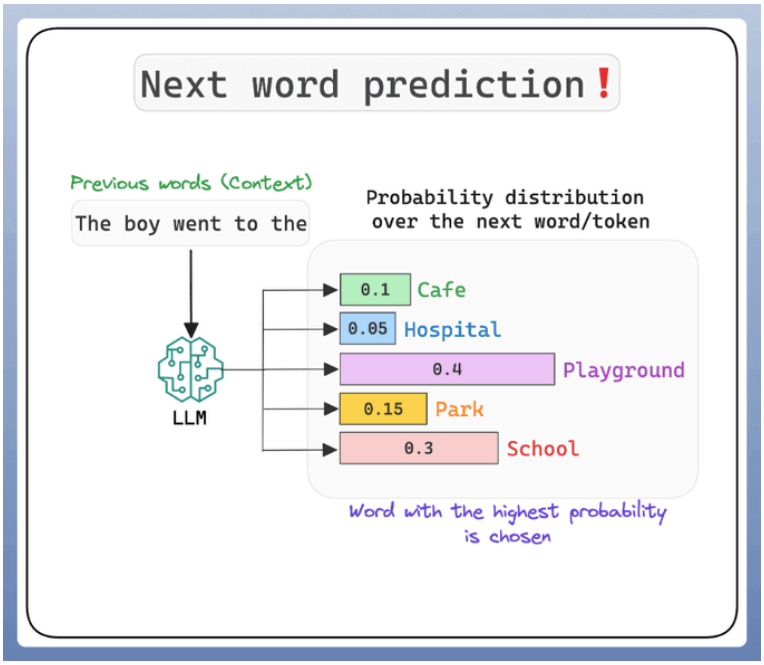
Now, how does this apply to LLMs like GPT-4?

These models are tasked with predicting/guessing the next word in a sequence.



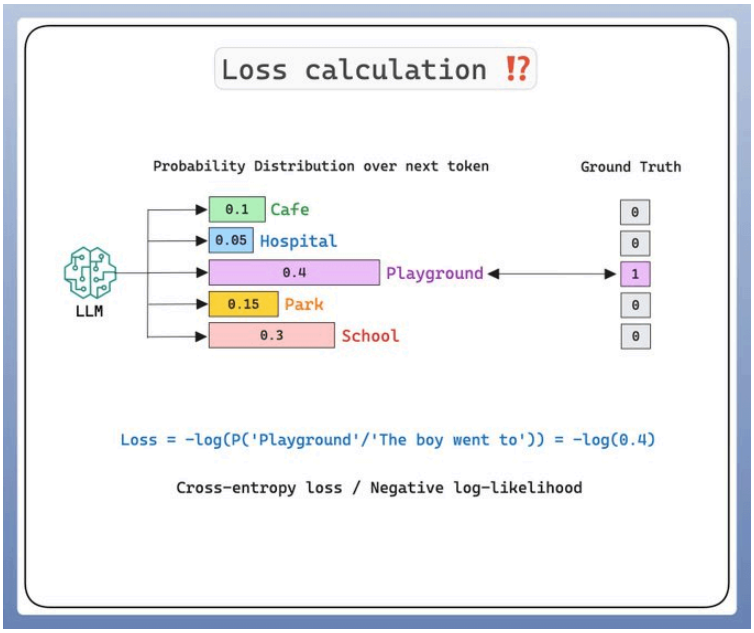
This is a question of conditional probability: given the words that have come before, what is the most likely next word?

To predict the next word, the model calculates the conditional probability for each possible next word, given the previous words (context).



The word with the highest conditional probability is chosen as the prediction.

The LLM learns a high-dimensional probability distribution over sequences of words.

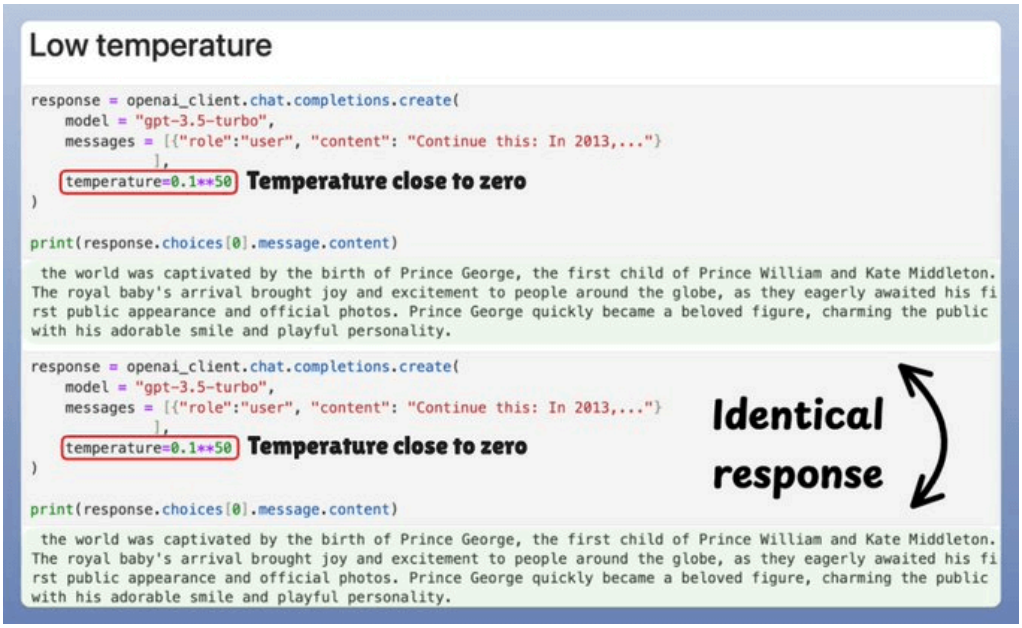


And the parameters of this distribution are the trained weights!

The training (or rather pre-training) is supervised.

But there is a problem!

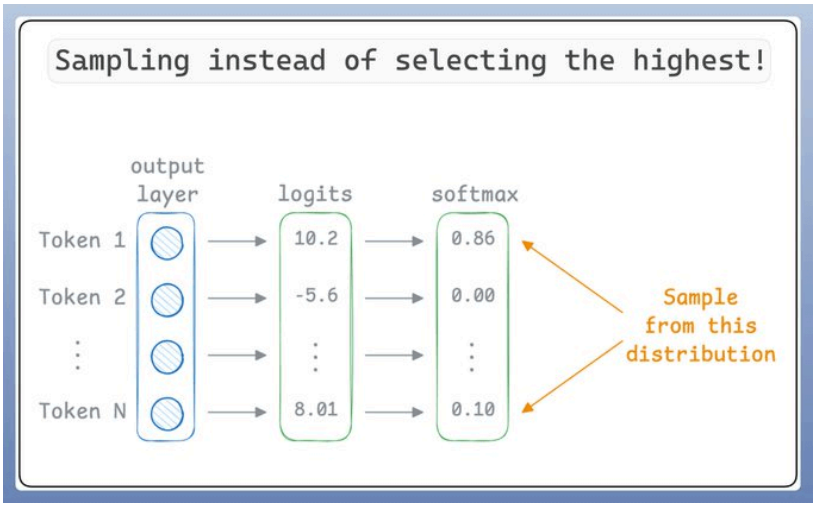
If we always pick the word with the highest probability, we end up with repetitive outputs, making LLMs almost useless and stifling their creativity.



This is where temperature comes into the picture.

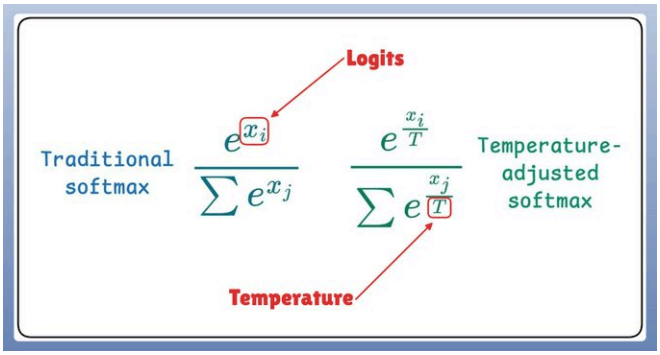
Let's understand what's going on..

To make LLMs more creative, instead of selecting the best token (for simplicity let's think of tokens as words), they "sample" the prediction.

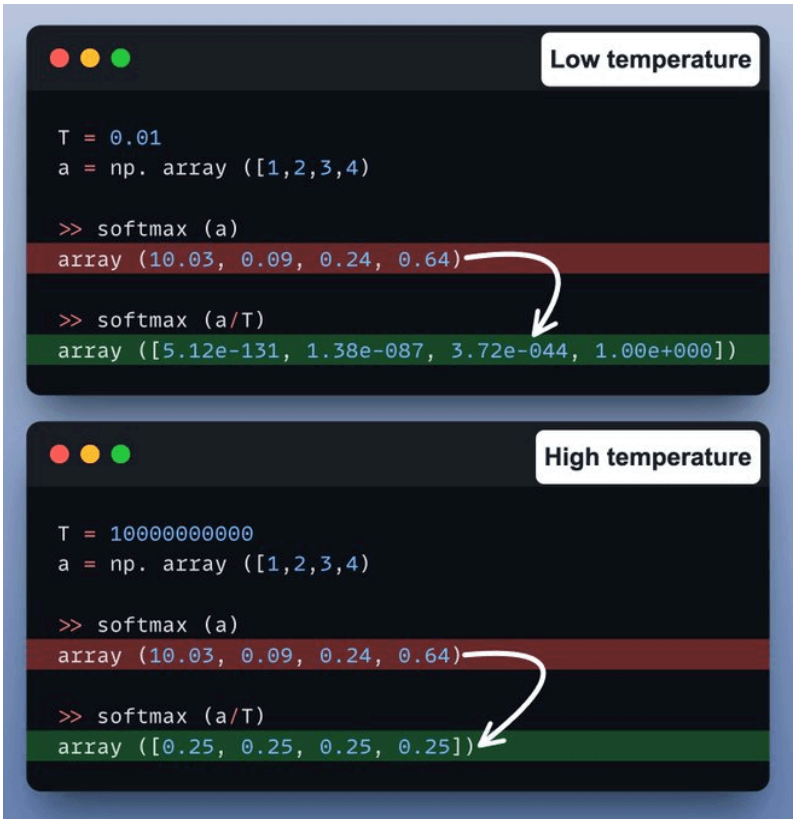


So even if “Token 1” has the highest score, it may not be chosen since we are sampling.

Now, temperature introduces the following tweak in the softmax function, which, in turn, influences the sampling process:



Let's take a code example!



- At low temperature, probabilities concentrate around the most likely token, resulting in nearly greedy generation.
- At high temperature, probabilities become more uniform, producing highly random and stochastic outputs.

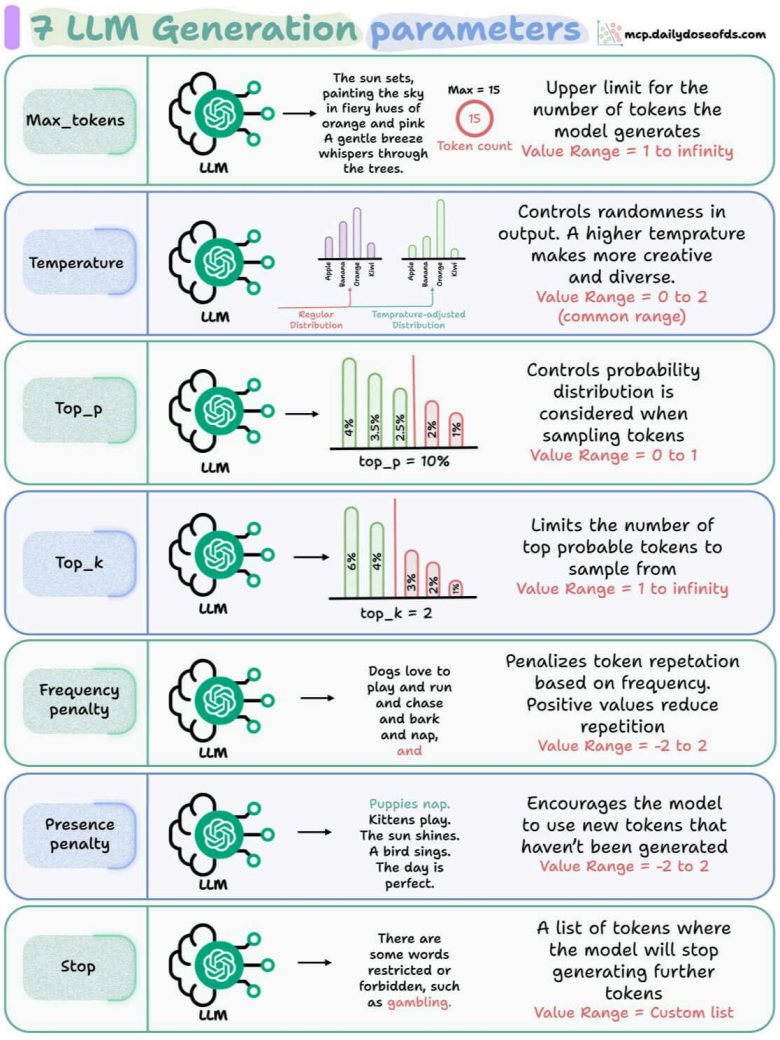
And that is how LLMs work and generate text.

7 LLM Generation Parameters

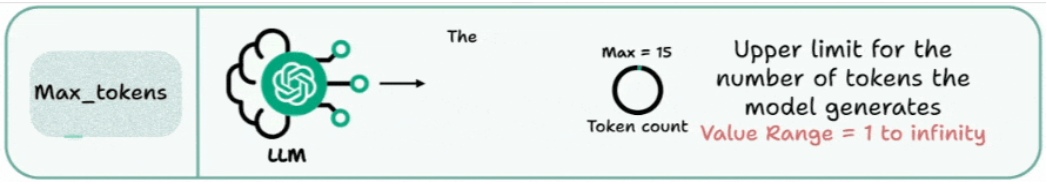
Every generation from an LLM is shaped by parameters under the hood.

Knowing how to tune is important so that you can produce sharp and more controlled outputs.

Here are the 7 levers that matter most:



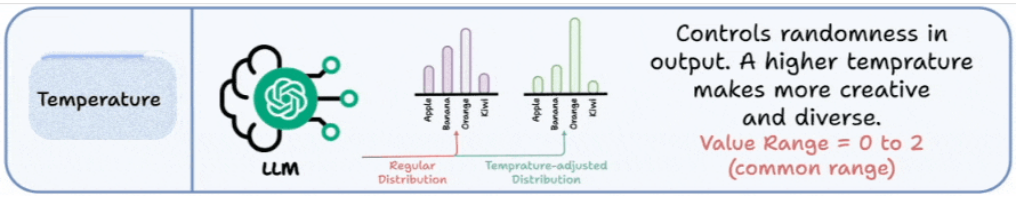
1) Max tokens



This is a hard cap on how many tokens the model can generate in one response.

Too low → truncated outputs; too high → could lead to wasted compute.

2) Temperature

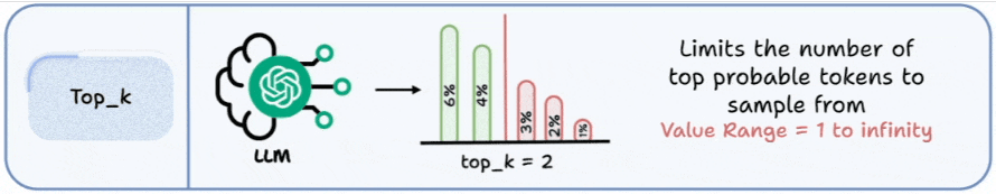


Governs randomness. Low temperature (~0) makes the model deterministic.

Higher temperature (0.7–1.0) boosts creativity, diversity, but also noise.

Use case: lower for QA/chatbots, higher for brainstorming/creative tasks.

3) Top-k



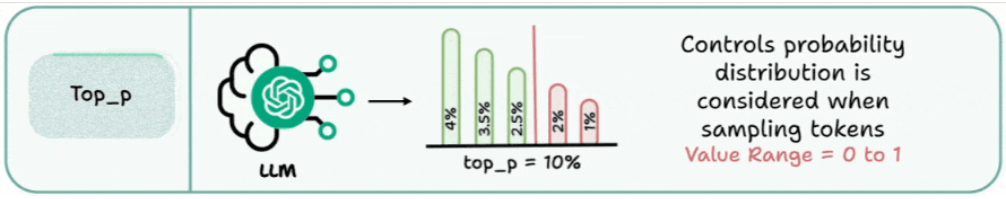
The default way to generate the next token is to sample from all tokens, proportional to their probability.

This parameter restricts sampling to the top k most probable tokens.

Example: k=5 → model only considers 5 most likely next tokens during sampling.

Helps enforce focus, but overly small k may give repetitive outputs.

4) Top-p (nucleus sampling)

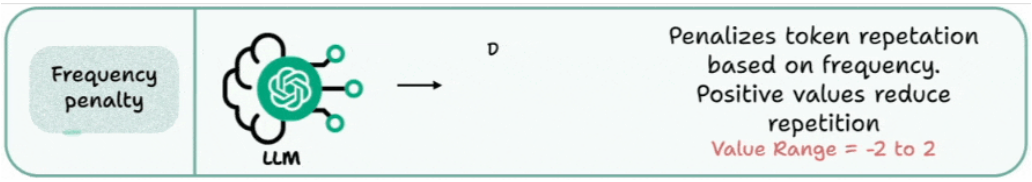


Instead of picking from all tokens or top k tokens, model samples from a probability mass up to p.

Example: top_p=0.9 → only the smallest set of tokens covering 90% probability are considered.

More adaptive than top_k, useful when balancing coherence with diversity.

5) Frequency penalty



Reduces likelihood of reusing tokens that have already appeared frequently.

Positive values discourage repetition, negative values exaggerate it.

Useful for summarization (avoid redundancy) or poetry (intentional repetition).

6) Presence penalty

Encourages the model to bring in new tokens not yet seen in the text.

Higher values push for novelty, lower values make the model stick to known patterns.

Handy for exploratory generation where diversity of ideas is valued.

7) Stop sequences

Custom list of tokens that immediately halt generation.

Critical in structured outputs (e.g., JSON), preventing spillover text.

Lets you enforce strict response boundaries without heavy prompt engineering.

Bonus: min-p sampling

min-p was introduced earlier this year. It's similar to top-p sampling, but instead of keeping a fixed cumulative probability mass (say, 90%), it dynamically adjusts based on the model's confidence.

It looks at the probability of the most likely token and only keeps tokens that are at least a certain fraction (say 10%) as likely.

For instance, if your top token has 60% probability, then only a few options remain. But if it's just 20%, many others can pass the threshold.

Basically, it automatically tightens or loosens the sampling pool depending on how confident the model is about the most likely token, which helps you achieve coherence when the model is confident and diversity when it's not that confident.

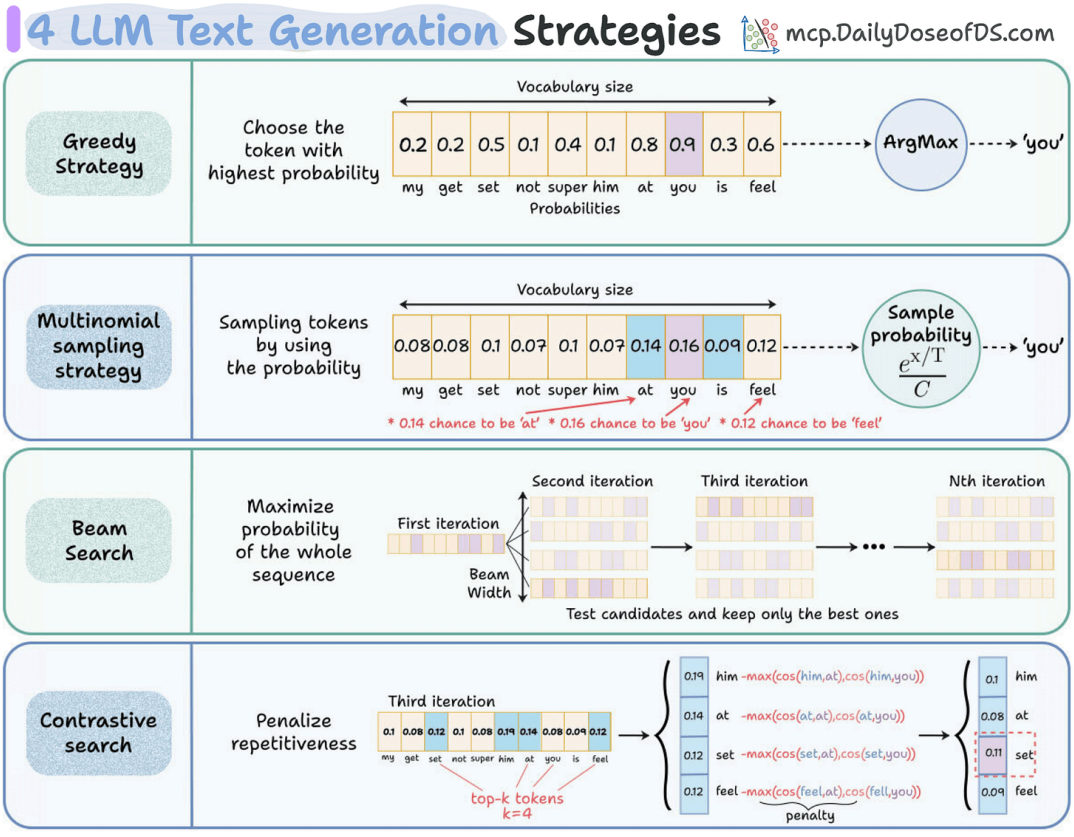
4 LLM Text Generation Strategies

Every time you prompt an LLM, it doesn't "know" the whole sentence in advance. Instead, it predicts the next token step by step.

But here's the catch: predicting probabilities is not enough. We still need a strategy to pick which token to use at each step.

And different strategies lead to very different styles of output.

Here are the 4 most common strategies for text generation:

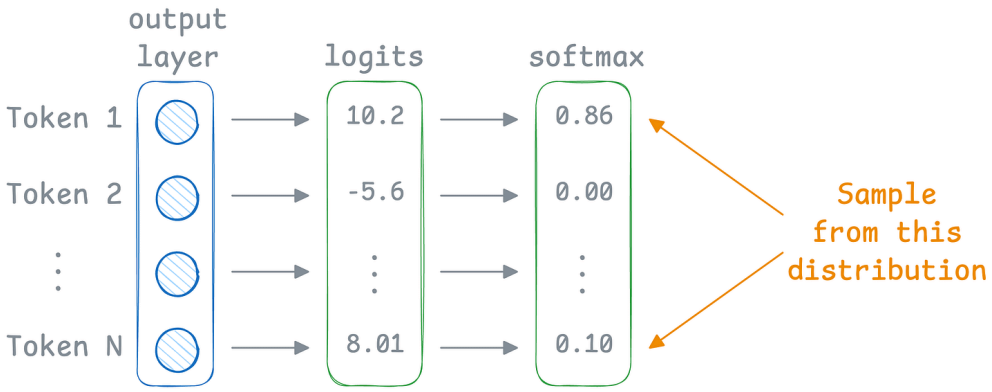


Approach 1: Greedy strategy

The naive approach greedily chooses the word with the highest probability from the probability vector, and autoregresses. This is often not ideal since it leads to repetitive sentences.

Approach 2: Multinomial sampling strategy

Instead of always picking the top token, we can sample from the probability distribution available in the probability vector.



The temperature parameter controls the randomness in the generation (covered in detail here).

Approach 3: Beam search

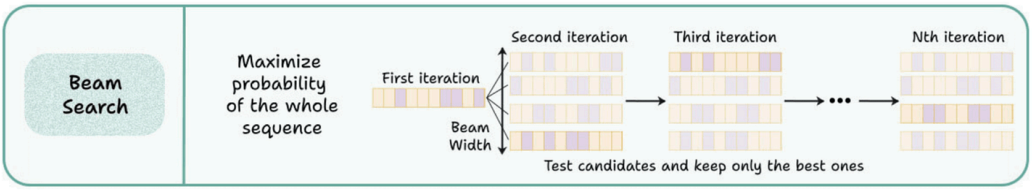
Both approach 1 and approach 2 have a problem. They only focus on the most immediate token to be generated. Ideally, we care about maximizing the probability of the whole sequence, not just the next token.

$$P(t_1, t_2, \ldots, t_N \mid \text{Prompt}) = \prod_{i=1}^{N} P(t_i \mid \text{Prompt}, t_1, \ldots, t_{i-1})$$

To maximize this product, you'd need to know future conditionals (what comes after each candidate).

But when decoding, we only know probabilities for the next step, not the downstream continuation.

Beam search tries to approximate the true global maximization:



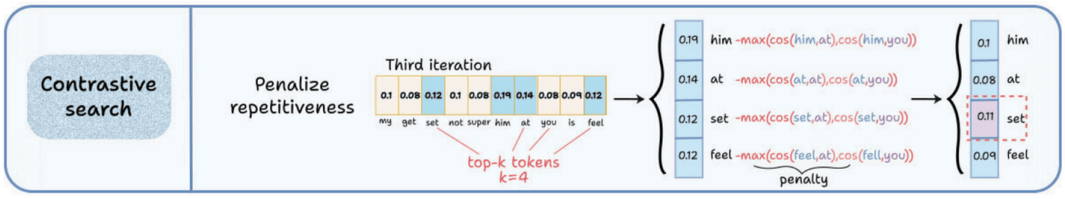
At each step, it expands the top k partial sequences (the beam).

Some beams may have started with less probable tokens initially, but lead to much higher-probability completions.

By keeping alternatives alive, beam search explores more of the probability tree.

This is widely used in tasks like machine translation, where correctness matters more than creativity.

Approach 4: Contrastive search



This is a newer method that balances fluency with diversity.

Essentially, it penalizes repetitive continuations by checking how similar a candidate token is to what's already been generated to have more diversity in the output.

At each step, the model considers candidate tokens.

Applies a penalty if the token is too similar to what's already been generated.

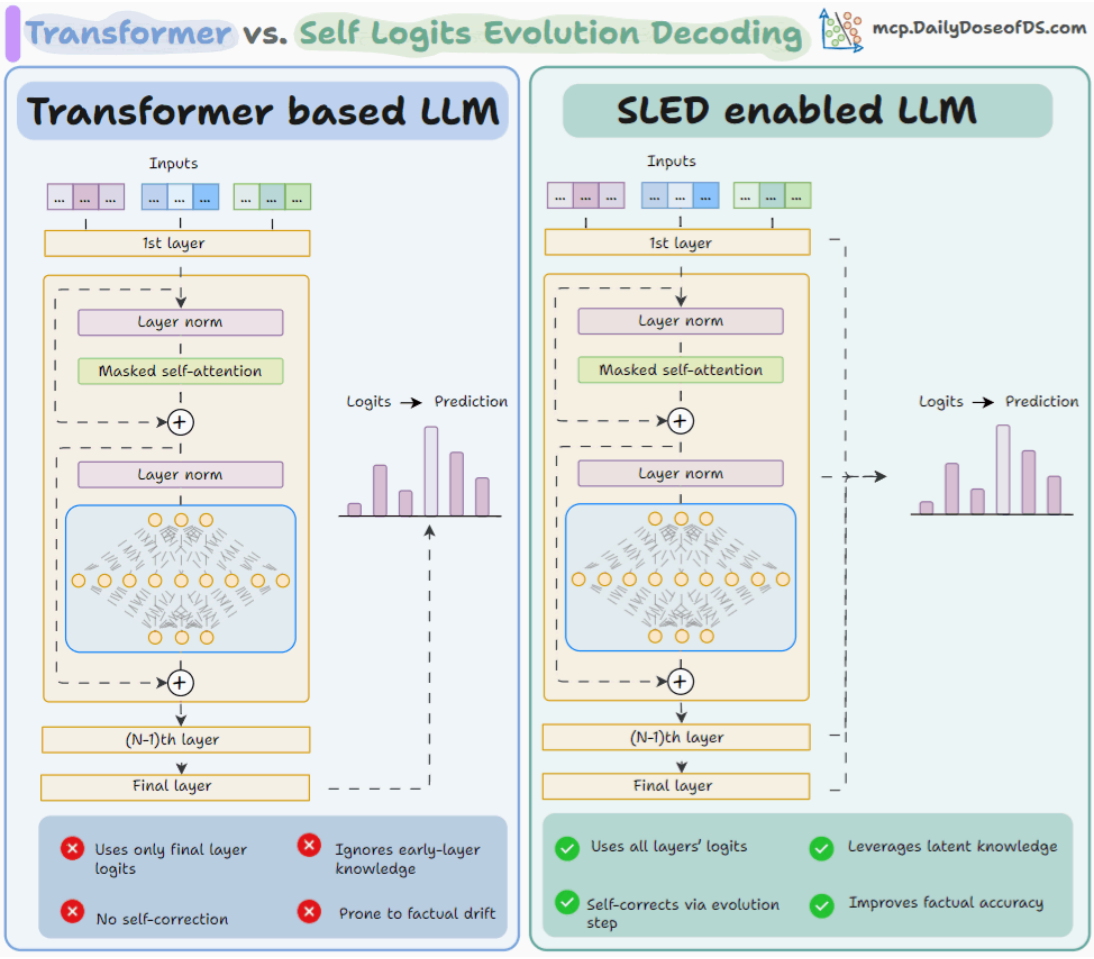
Selects the token that balances probability and diversity.

This way, it also prevents "stuck in a loop" problems while keeping coherence high.

It's especially effective for longer generations like stories, where repetition can easily creep in.

Bonus: SLED - Self-Logits Evolution Decoding

All the decoding strategies above rely on the logits produced by the final layer, which is how Transformers normally generate text. The issue is that factual signals present in earlier layers can fade as the model goes deeper, leading the final layer to favor fluent but occasionally inaccurate outputs.



SLED introduces a small but meaningful change: instead of using only the final layer's logits, it looks at how logits evolve across *all* layers. Each layer contributes its own prediction, and SLED measures how closely these predictions agree. It then nudges the final logits toward this layer-wise consensus before selecting the next token.

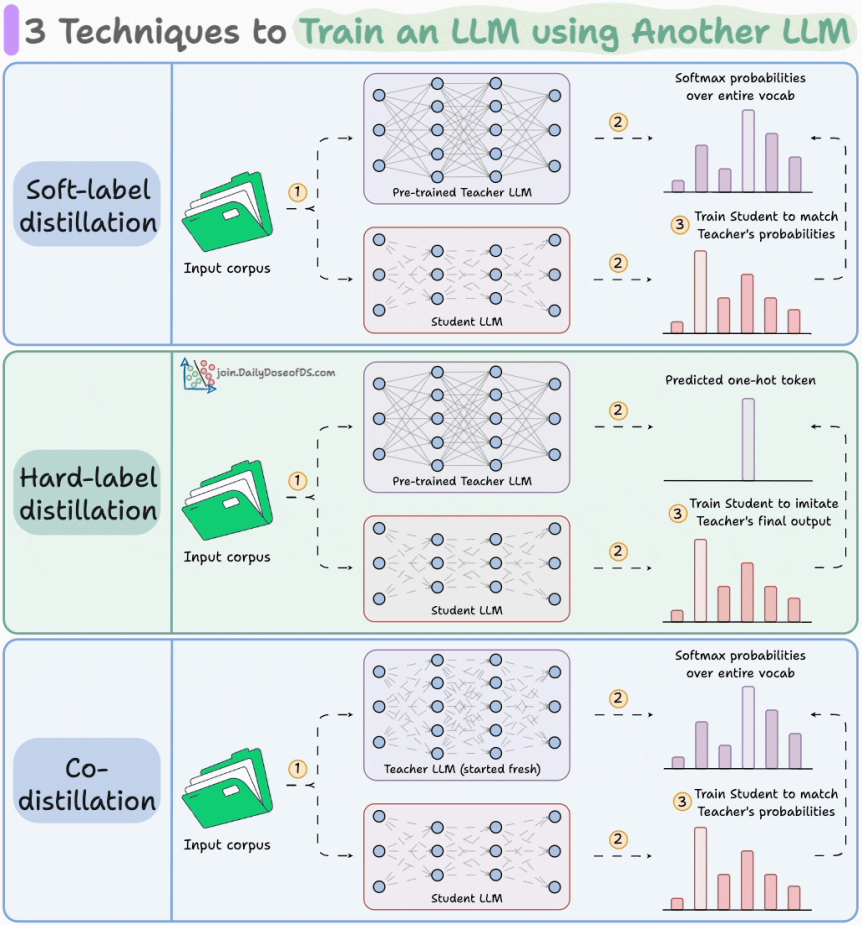
This requires no retraining, extra data or additional compute. By leveraging the model's full internal knowledge rather than only its last step, SLED produces more grounded and factual generations with the same architecture.

3 Techniques to Train An LLM Using Another LLM

LLMs don't just learn from raw text; they also learn from each other:

- Llama 4 Scout and Maverick were trained using Llama 4 Behemoth.
- Gemma 2 and 3 were trained using Google's proprietary Gemini.

Distillation helps us do so, and the visual below depicts three popular techniques.



The idea is to transfer "knowledge" from one LLM to another, which has been quite common in traditional deep learning

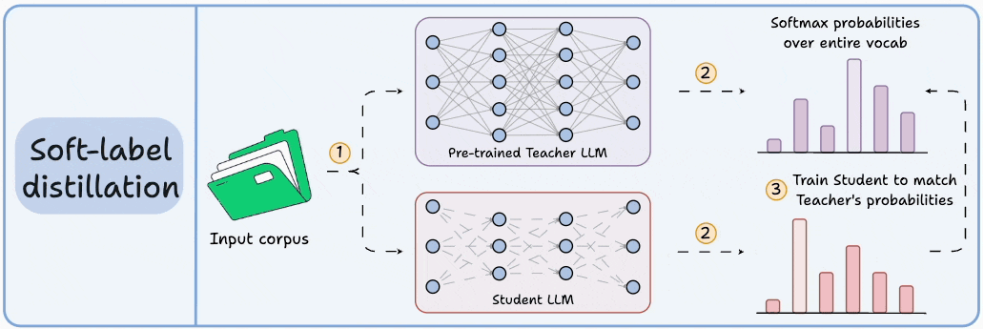
Distillation in LLMs can happen at two stages:

- Pre-training
  - Train the bigger Teacher LLM and the smaller student LLM together.
  - Llama 4 did this.
- Post-training:
  - Train the bigger Teacher LLM first and distill its knowledge to the smaller student LLM.
  - DeepSeek did this by distilling DeepSeek-R1 into Qwen and Llama 3.1 models.

You can also apply distillation during both stages, which Gemma 3 did.

Here are the three commonly used distillation techniques:

1) Soft-label distillation



- Use a fixed pre-trained Teacher LLM to generate softmax probabilities over the entire corpus.
- Pass this data through the untrained Student LLM as well to get its softmax probabilities.
- Train the Student LLM to match the Teacher's probabilities.

Visibility over the Teacher's probabilities ensures maximum knowledge (or reasoning) transfer.

However, you must have access to the Teacher's weights to get the output probability distribution.

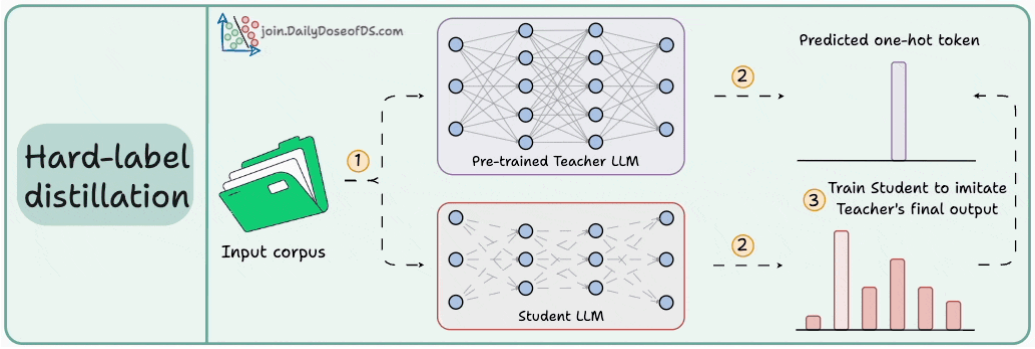
Even if you have access, there's another problem!

Say your vocab size is 100k tokens and your data corpus is 5 trillion tokens.

Since we generate softmax probabilities of each input token over the entire vocabulary, you would need 500 million GBs of memory to store soft labels under float8 precision.

The second technique solves this.

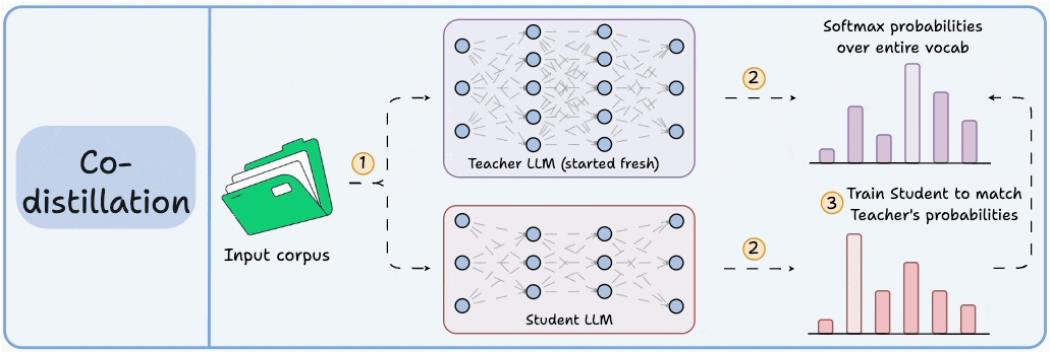
2) Hard-label distillation



- Use a fixed pre-trained Teacher LLM to just get the final one-hot output token.
- Use the untrained Student LLM to get the softmax probabilities from the same data.
- Train the Student LLM to match the Teacher's probabilities.

DeepSeek did this by distilling DeepSeek-R1 into Qwen and Llama 3.1 models.

3) Co-distillation



- Start with an untrained Teacher LLM and an untrained Student LLM.
- Generate softmax probabilities over the current batch from both models.
- Train the Teacher LLM as usual on the hard labels.
- Train the Student LLM to match its softmax probabilities to those of the Teacher.

Llama 4 did this to train Llama 4 Scout and Maverick from Llama 4 Behemoth.

Of course, during the initial stages, soft labels of the Teacher LLM won't be accurate.

That is why Student LLM is trained using both soft labels + ground-truth hard labels.

4 Ways to Run LLMs Locally

Being able to run LLMs locally has many upsides:

- Privacy since your data never leaves your machine.
- Testing things locally before moving to the cloud and more.

Let's discuss the four ways to run LLMs locally.

1) Ollama

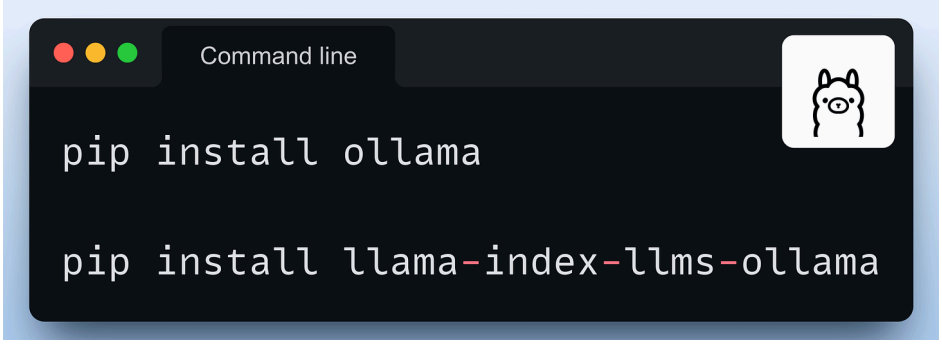
Running a model through Ollama is as simple as executing this command:

To get started, install Ollama with a single command:

Done!

Now, you can download any of the supported models using these commands:

For programmatic usage, you can also install the Python package of Ollama or its integration with orchestration frameworks like Llama Index or CrewAI:



2) LMStudio

LMStudio can be installed as an app on your computer.

The app does not collect data or monitor your actions. Your data stays local on your machine. It's free for personal use.

It offers a ChatGPT-like interface, allowing you to load and eject models as you chat. This video shows its usage:

Just like Ollama, LMStudio supports several LLMs as well.

3) vLLM

vLLM is a fast and easy-to-use library for LLM inference and serving(*more details in LLM deployment section*)

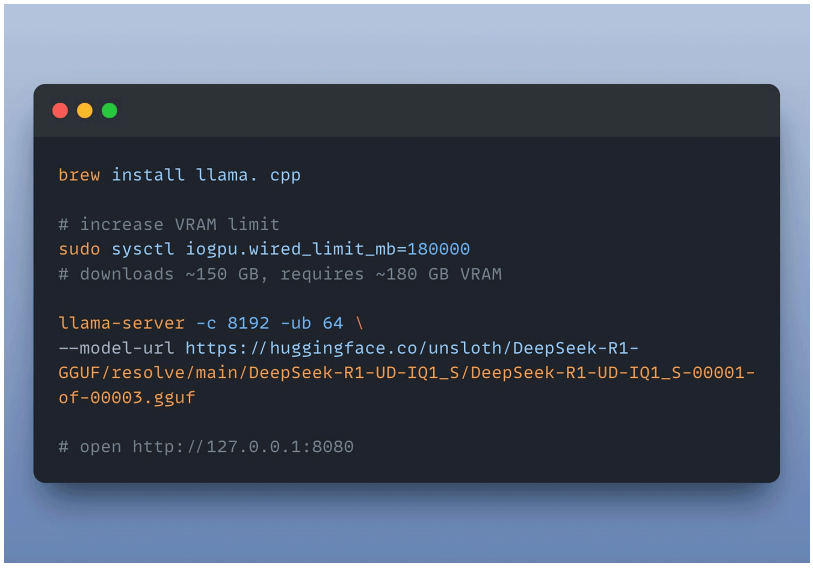
With just a few lines of code, you can locally run LLMs (like DeepSeek) in an OpenAI-compatible format:

```
pip install vllm

vllm serve deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B \
    --enable-reasoning --reasoning-parser deepseek_r1
```

```
from openai import OpenAI

# Modify OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)

models = client.models.list()
model = models.data[0].id

# Round 1
messages = [{"role": "user", "content": "9.11 and 9.8, which is greater?"}]
response = client.chat.completions.create(model=model, messages=messages)

reasoning_content = response.choices[0].message.reasoning_content
content = response.choices[0].message.content

print("reasoning_content:", reasoning_content)
print("content:", content)
```

4) LlamaCPP

LlamaCPP enables LLM inference with minimal setup and good performance.



```
brew install llama. cpp

# increase VRAM limit
sudo sysctl iogpu.wired_limit_mb=180000
# downloads ~150 GB, requires ~180 GB VRAM

llama-server -c 8192 -ub 64 \
--model-url https://huggingface.co/unsloth/DeepSeek-R1-
GGUF/resolve/main/DeepSeek-R1-UD-IQ1_S/DeepSeek-R1-UD-IQ1_S-00001-
of-00003.gguf

# open http://127.0.0.1:8080
```

Transformer vs. Mixture of Experts in LLMs

Up to this point, we've explored how LLMs are built, trained and how they generate text.

All modern LLMs rely on the Transformer architecture, but there is another important question:

How do we scale models further without making them impossibly large and expensive to run?

This is where Mixture of Experts (MoE) architectures come in.

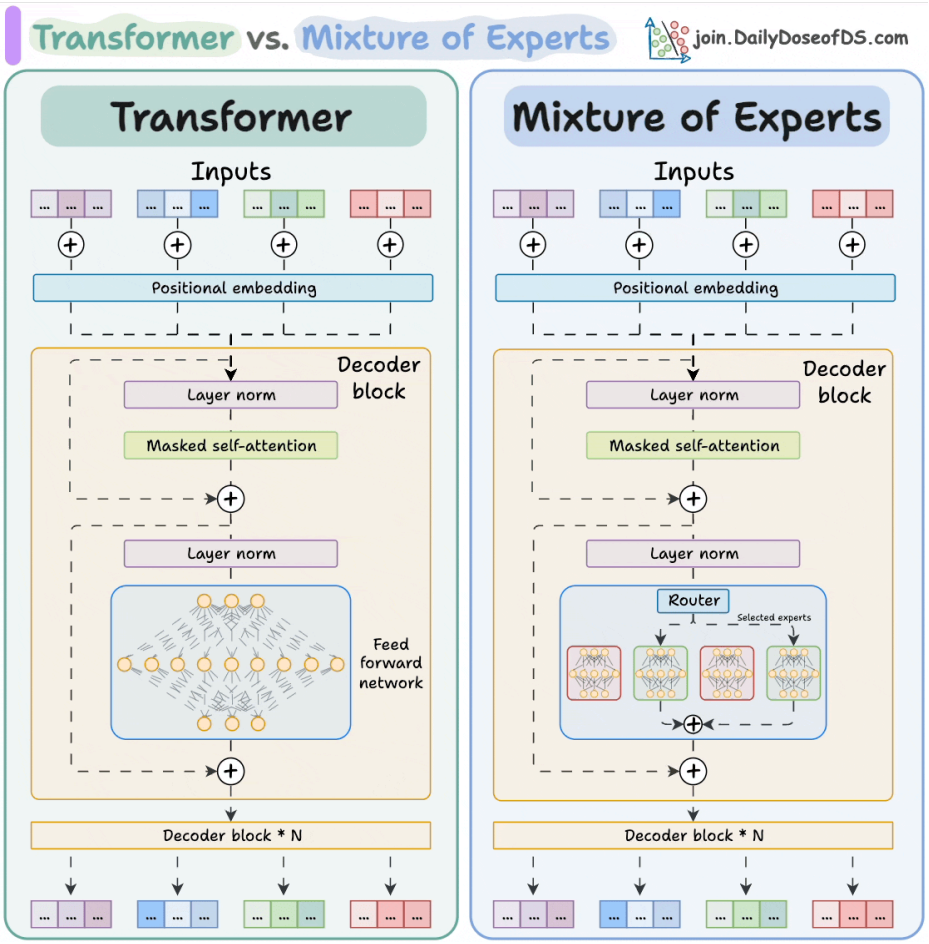
MoE keeps the overall parameter count large but activates only a small subset of "experts" for each token.

This allows models to grow in capacity without proportional increases in compute.

With that context, let's compare the traditional Transformer block with the MoE alternative.

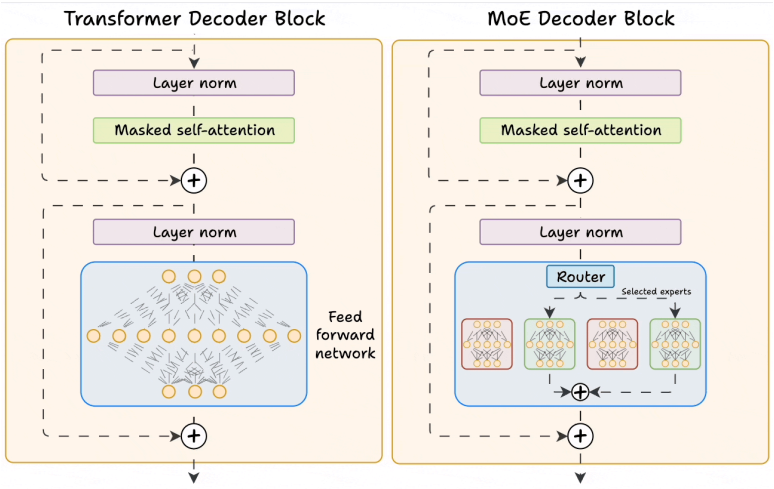
Mixture of Experts (MoE) is a popular architecture that uses different "experts" to improve Transformer models.

The visual below explains how they differ from Transformers.



Let's dive in to learn more about MoE!

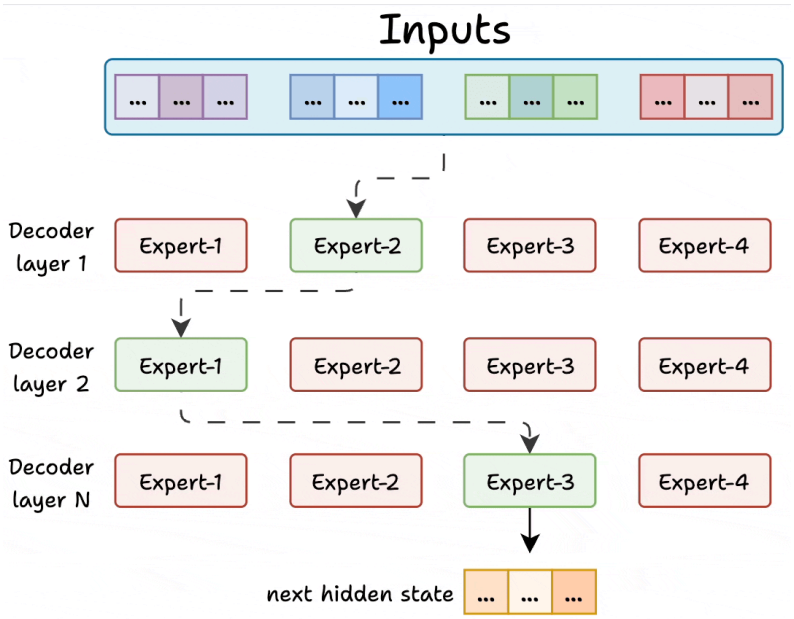
Transformer and MoE differ in the decoder block:



- Transformer uses a feed-forward network.
- MoE uses experts, which are feed-forward networks but smaller compared to that in Transformer.

During inference, a subset of experts are selected. This makes inference faster in MoE.

Also, since the network has multiple decoder layers:



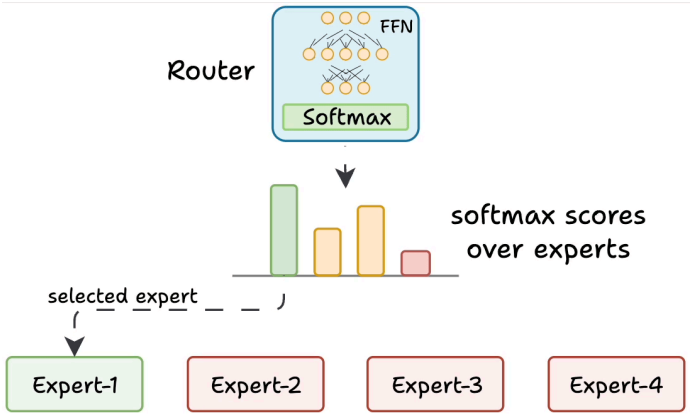
- the text passes through different experts across layers.
- the chosen experts also differ between tokens.

But how does the model decide which experts should be ideal?

The router does that.

The router is like a multi-class classifier that produces softmax scores over experts. Based on the scores, we select the top K experts.

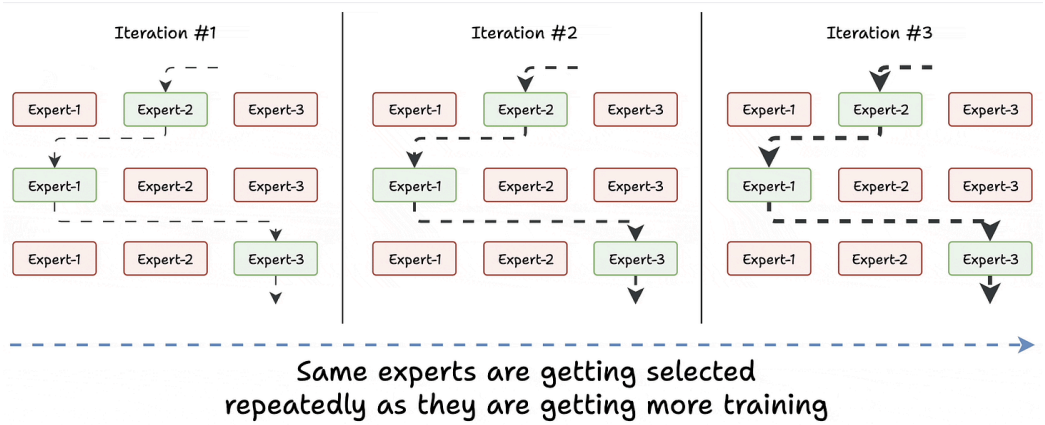
The router is trained with the network and it learns to select the best experts.



But it isn't straightforward.

There are challenges.

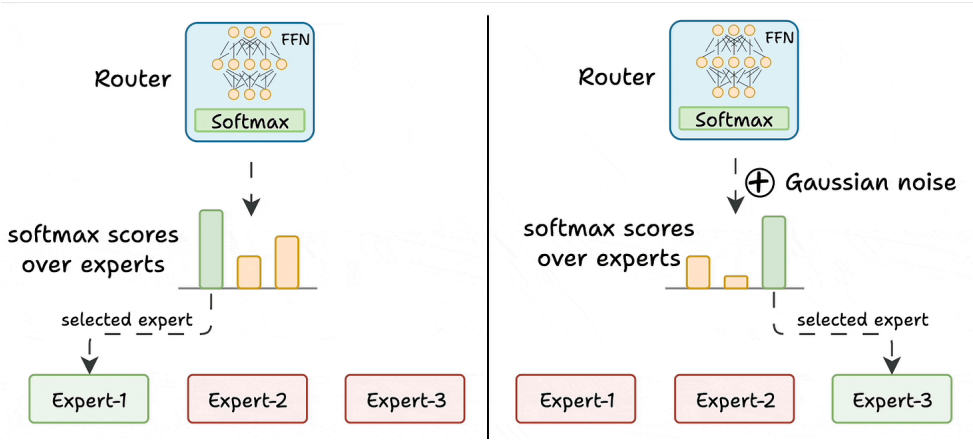
Challenge 1) Notice this pattern at the start of training:



- The model selects "Expert 2" (randomly since all experts are similar).
- The selected expert gets a bit better.
- It may get selected again since it's the best.
- This expert learns more.
- The same expert can get selected again since it's the best.
- It learns even more.
- And so on!

Essentially, this way, many experts go under-trained!

We solve this in two steps:



- Add noise to the feed-forward output of the router so that other experts can get higher logits.
- Set all but top K logits to -infinity. After softmax, these scores become zero.

This way, other experts also get the opportunity to train.

Challenge 2) Some experts may get exposed to more tokens than others - leading to under-trained experts.

We prevent this by limiting the number of tokens an expert can process.

If an expert reaches the limit, the input token is passed to the next best expert instead.

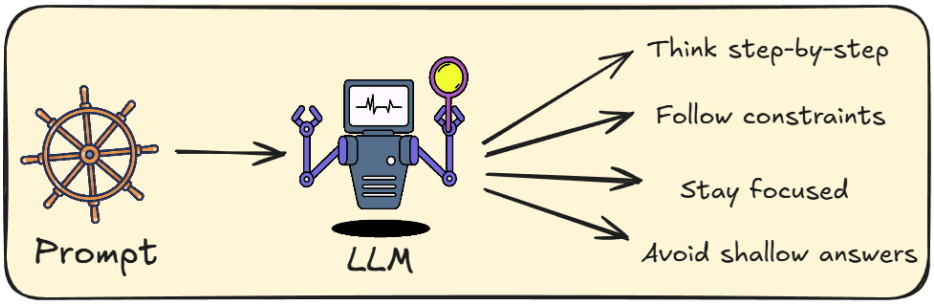
MoEs have more parameters to load. However, a fraction of them are activated since we only select some experts.

This leads to faster inference. Mixtral 8x7B by MistralAI is one famous LLM that is based on MoE.

Prompt Engineering

What is Prompt Engineering?

LLMs are powerful, but they don't automatically know what you want. Prompt engineering is the simplest way to control them.



Think of it as the steering wheel for the LLM.

Small adjustments completely shift the direction of the output.

You're not changing weights (the learned parameters inside the model). You're changing instructions and that changes everything.

A good prompt helps the model:

- Think step-by-step
- Follow constraints
- Stay focused
- Avoid shallow answers

It's the fastest, lowest-effort way to get better results from any model.

Let's explore three prompting techniques that significantly improve an LLM's reasoning ability.

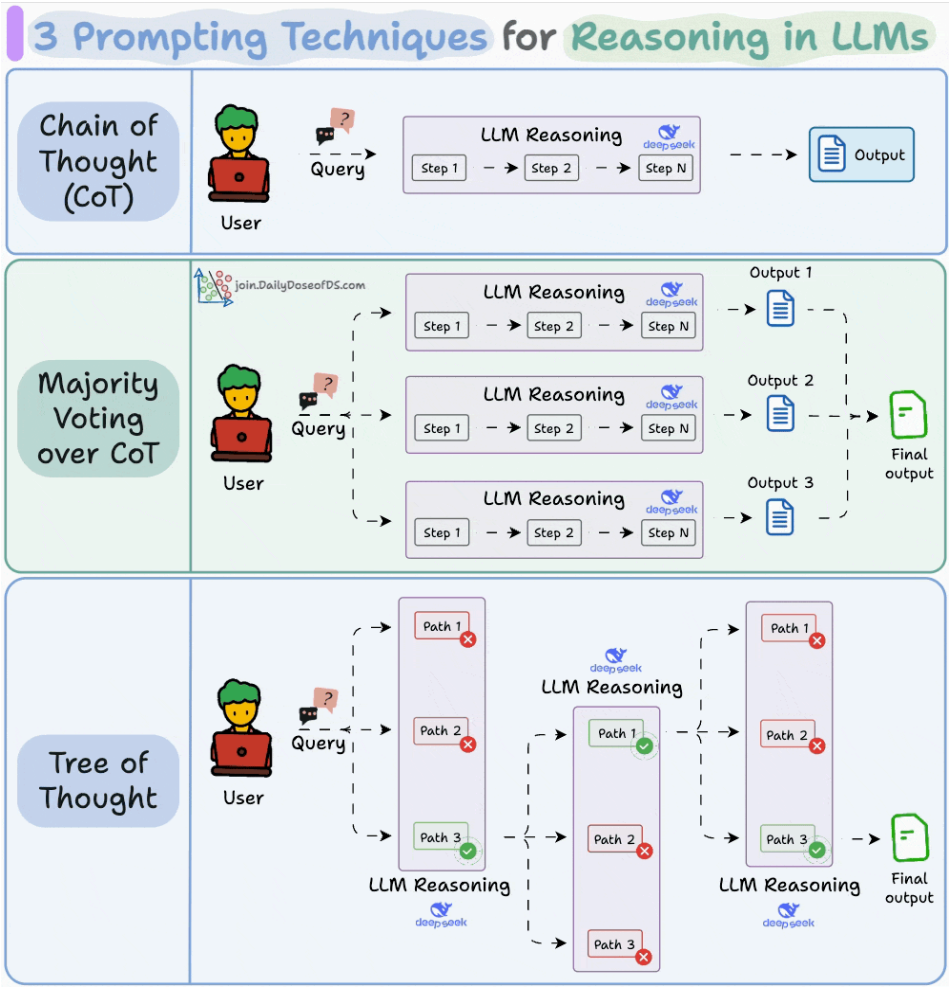
3 prompting techniques for reasoning in LLMs

A large part of what makes such tools so powerful isn't just their ability to write code, but their ability to reason through it.

And that's not unique to code. It's the same when we prompt LLMs to solve complex reasoning tasks like math, logic, or multi-step problems.

Let's look at three popular prompting techniques that help LLMs think more clearly before they answer.

These are depicted below:



1) Chain of Thought (CoT)

The simplest and most widely used technique.

Instead of asking the LLM to jump straight to the answer, we nudge it to reason step by step.

This often improves accuracy because the model can walk through its logic before committing to a final output.

For instance:

```
Q: If John has 3 apples and gives away 1, how many are left?

Let's think step by step:
```

It's a simple example but this tiny nudge can unlock reasoning capabilities that standard zero-shot prompting could miss.

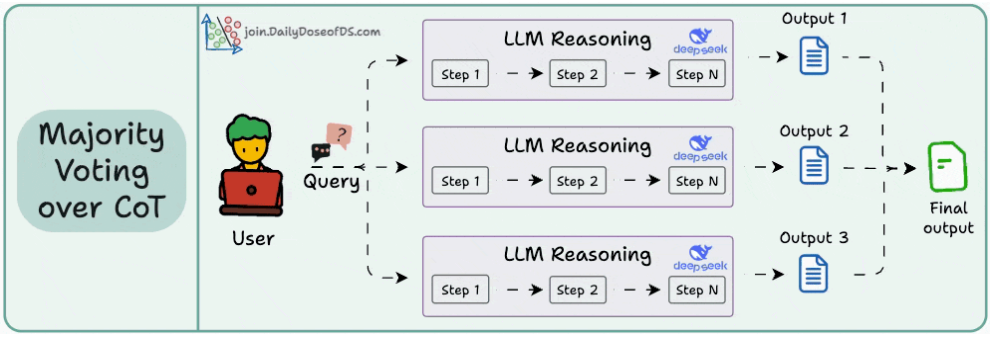
2) Self-Consistency (a.k.a. Majority Voting over CoT)

CoT is useful but not always consistent.

If you prompt the same question multiple times, you might get different answers depending on the temperature setting (we covered temperature in LLMs here).

Self-Consistency embraces this variation.

You ask the LLM to generate multiple reasoning paths and then select the most common final answer.



It's a simple idea: when in doubt, ask the model several times and trust the majority.

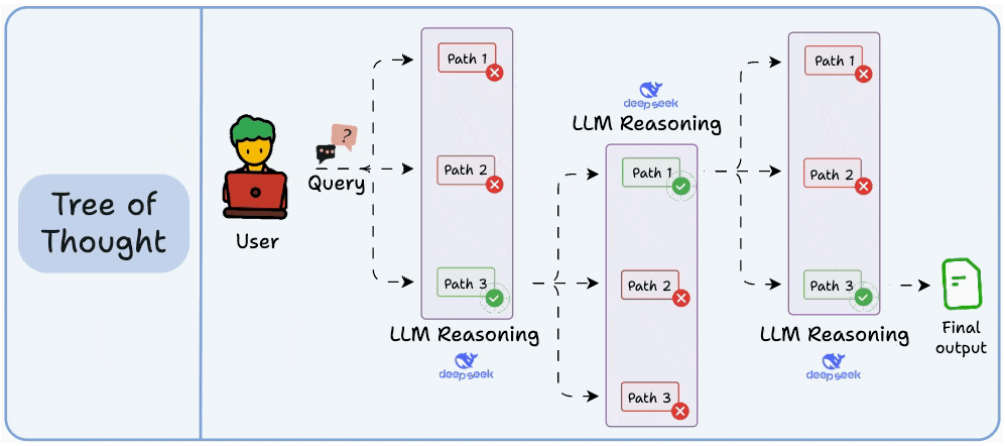
This technique often leads to more robust results, especially on ambiguous or complex tasks.

However, it doesn't evaluate how the reasoning was done—just whether the final answer is consistent across paths.

3) Tree of Thoughts (ToT)

While Self-Consistency varies the final answer, Tree of Thoughts varies the steps of reasoning at each point and then picks the best path overall.

At every reasoning step, the model explores multiple possible directions. These branches form a tree, and a separate process evaluates which path seems the most promising at a particular timestamp.



Think of it like a search algorithm over reasoning paths, where we try to find the most logical and coherent trail to the solution.

It's more compute-intensive, but in most cases, it significantly outperforms basic CoT.

CoT, Self-Consistency, and ToT all improve how the model reasons through a problem.

But they still rely on free-form thinking, which breaks down in long, rule-heavy tasks.

That's where ARQ comes in.

Bonus: ARQ

Here's the core problem with current techniques that this new approach solves.

We have enough research to conclude that LLMs often struggle to assess what truly matters in a particular stage of a long, multi-turn conversation.

For instance, when you give Agents a 2,000-word system prompt filled with policies, tone rules, and behavioral dos and don'ts, you expect them to follow it word by word.

- But here's what actually happens:
- They start strong initially.
- Soon, they drift and start hallucinating.
- Shortly after, they forget what was said five turns ago.

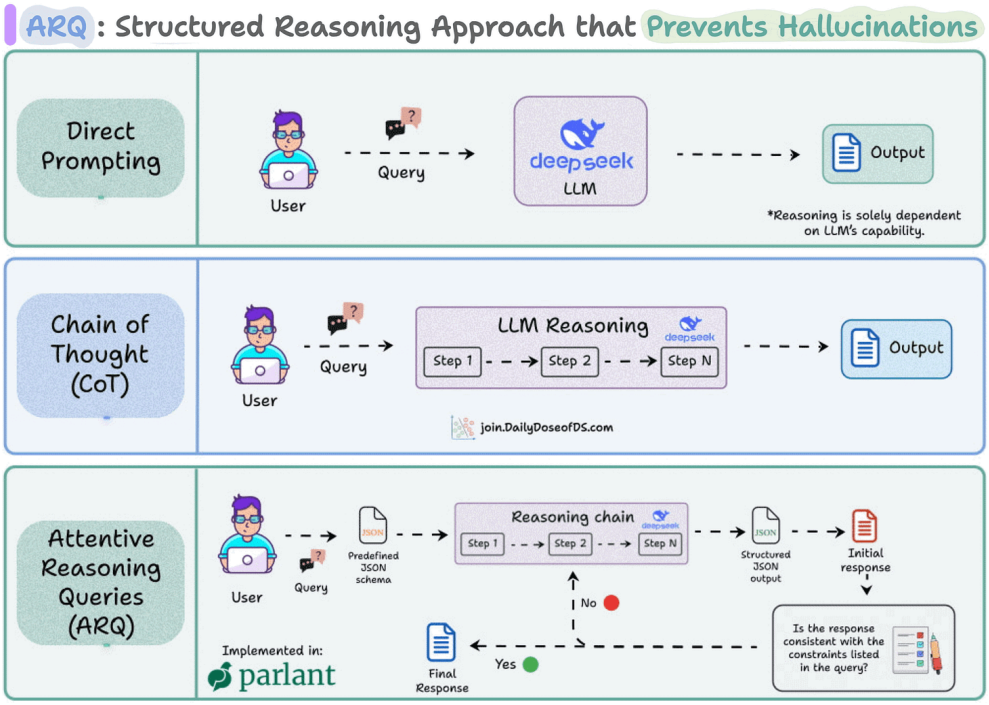
And finally, the LLM that was supposed to "never promise a refund" is happily offering one.

This means they can easily ignore crucial rules (stated initially) halfway through the process.

We expect techniques like Chain-of-Thought will help.

But even with methods like CoT, reasoning remains free-form, i.e., the model "thinks aloud" but it has limited domain-specific control.

That's the exact problem the new technique, called Attentive Reasoning Queries (ARQs), solves.



Instead of letting LLMs reason freely, ARQs guide them through explicit, domain-specific questions.

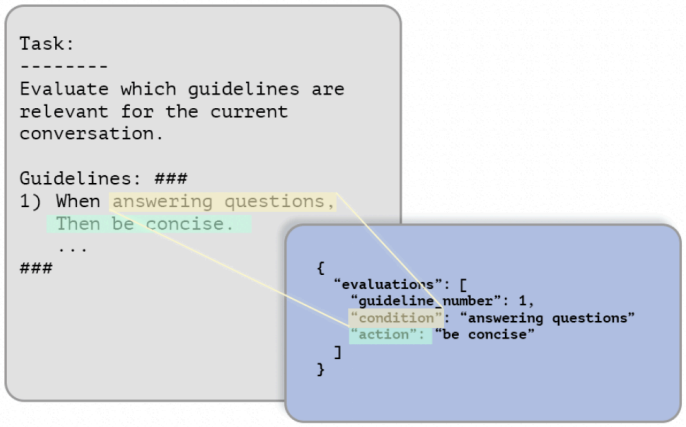
Essentially, each reasoning step is encoded as a targeted query inside a JSON schema.

For example, before making a recommendation or deciding on a tool call, the LLM is prompted to fill structured keys like:

```
{
  “current_context”: “Customer asking about refund eligibility”,
  “active_guideline”: “Always verify order before issuing refund”,
  “action_taken_before”: false,
  “requires_tool”: true,
  “next_step”: “Run check_order_status()”
}
```

This type of query does two things:

1. Reinstate critical instructions by keeping the LLM aligned mid-conversation.
2. Facilitate intermediate reasoning, so that the decisions are auditable and verifiable.



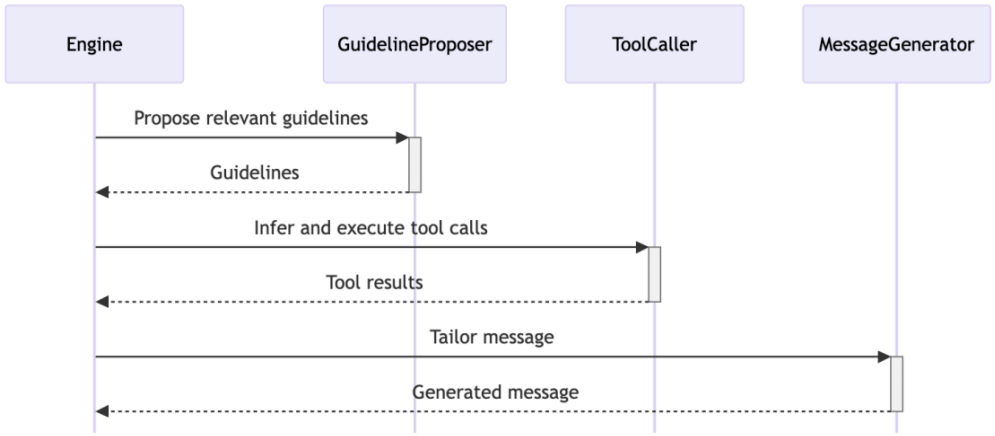
By the time the LLM generates the final response, it’s already walked through a sequence of *controlled* reasoning steps, which did not involve any free text exploration (unlike techniques like CoT or ToT).

Here’s the success rate across 87 test scenarios:

- ARQ - 90.2%
- CoT reasoning - 86.1%
- Direct response generation - 81.5%

This approach is actually implemented in Parlant, a recently trending open-source framework to build instruction-following Agents.

ARQs are integrated into three key modules:



- Guideline proposer to decide which behavioral rules apply.
- Tool caller to determine what external functions to use.
- Message generator, when it produces the final customer-facing reply.

The core insight applies regardless of what tools you use:

When you make reasoning explicit, measurable, and domain-aware, LLMs stop improvising and start reasoning with intention. Free-form thinking sounds powerful, but in high-stakes or multi-turn scenarios, structure always wins.

ARQ solves the problem of uncontrolled reasoning by adding structure.

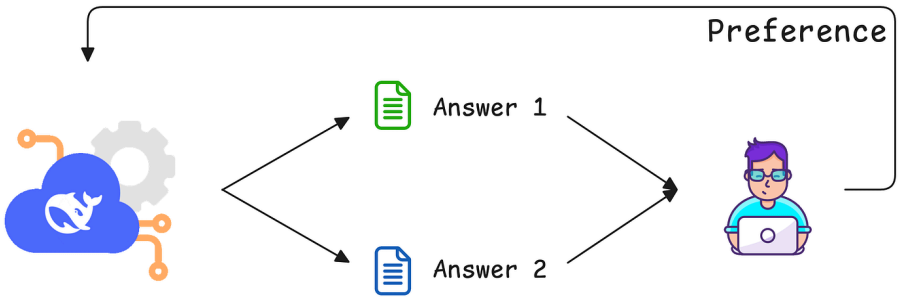
But there's another challenge: many aligned LLMs stop exploring alternative answers altogether.

Even with good reasoning steps, the model may collapse into the same safe, typical responses.

To regain that lost diversity without retraining the model, we use Verbalized Sampling.

Verbalized Sampling

Post-training alignment methods, such as RLHF, are designed to make LLMs helpful and safe.



However, these methods unintentionally cause a significant drop in output diversity (called mode collapse).

When an LLM collapses to a mode, it starts favoring a narrow set of predictable or stereotypical responses over other outputs.

According to a paper, mode collapse happens because the human preference data used to train the LLM has a hidden flaw called typicality bias.

> 3.1 TYPICALITY BIAS IN PREFERENCE DATA: COGNITIVE & EMPIRICAL EVIDENCE
>
> **Typicality Bias Hypothesis.** Cognitive psychology shows that people prefer text that is *familiar*, *fluent*, and *predictable*. This preference is rooted in various principles. For instance, the *mere-exposure effect* (Zajonc, 1968; Bornstein, 1989) and *availability heuristic* (Tversky & Kahneman, 1973) imply that frequent or easily recalled content feels more likely and is liked more. *Processing fluency* (Alter & Oppenheimer, 2009; Reber et al., 2004) suggests that easy-to-process content is automatically perceived as more truthful and higher quality. Moreover, *schema congruity* theory (Mandler, 2014; Meyers-Levy & Tybout, 1989) predicts that information that aligns with existing mental models will be accepted with less critical thought. We therefore hypothesize that these cognitive tendencies lead to a *typicality bias* in preference data, in which annotators systematically favor conventional text.

Here's how this happens:

Annotators are asked to rate different responses from an LLM, and later, the LLM is trained using a reward model that learns to mimic these human preferences.

However, it is observed that annotators naturally tend to favor answers that are more familiar, easy to read, and predictable. This is the typicality bias. So even if a new, creative answer is just as good (or correct) as a common one, the human's preference often leans toward the common one.

Due to this, the reward model boosts responses that the original (pre-aligned) model already considered likely.

This aggressively sharpens the LLM's probability distribution, collapsing the model's creative output to one or two dominant, highly predictable responses.

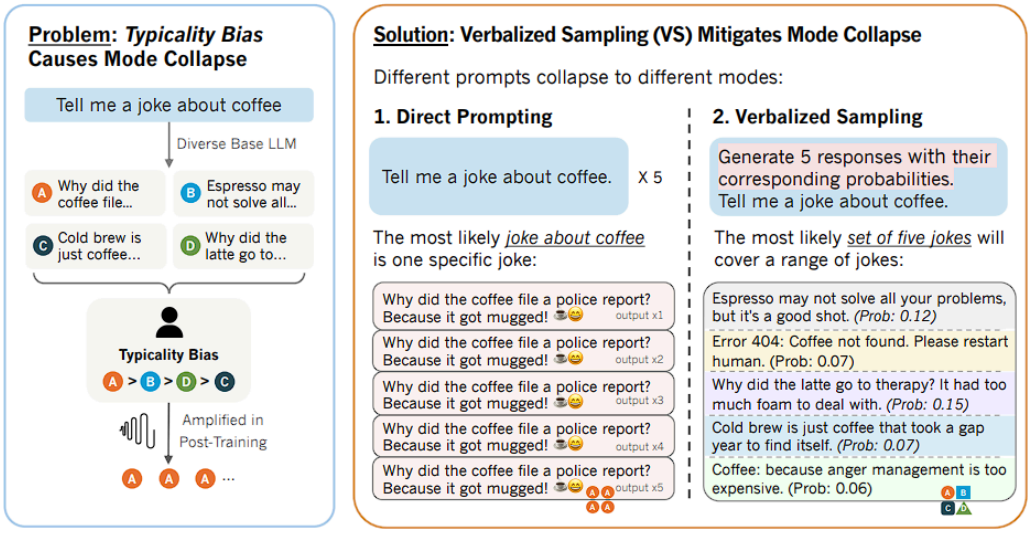
That said, this is not an irreversible effect, and the LLM still has two personalities after alignment:

The original model that learned the rich possibilities during pre-training.

The safety-focused, post-aligned model [to mention again, due to typicality bias, it had been unintentionally suppressed to strongly favor the most predictable response]

Verbalized sampling (VS) solves this.

It is a training-free prompting strategy introduced to circumvent mode collapse and recover the diverse distribution learned during pre-training.



The core idea of verbalized sampling is that the prompt itself acts like a mental switch.

When you directly prompt "Tell me a joke", the aligned personality immediately takes over and outputs the most reinforced answer.

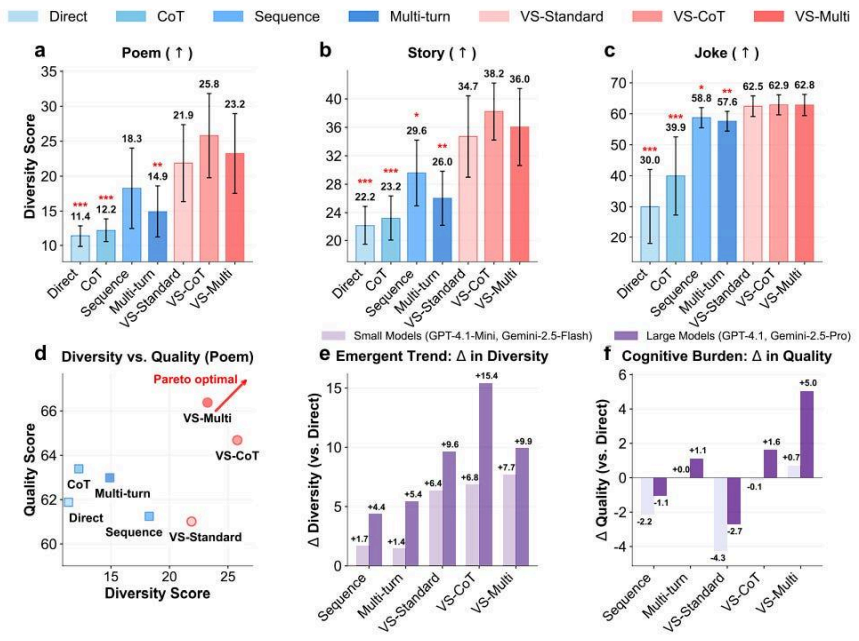
But in verbalized sampling, you prompt it with "Generate 5 responses with their corresponding probabilities. Tell me a joke."

In this case, the prompt does not request an instance, but a distribution.

This causes the aligned model to talk about its full knowledge and is forced to utilize the diverse distribution it learned during pre-training.

So essentially, by asking the LLM to verbalize the probability distribution, the model is able to tap into the broader, diverse set of ideas, which comes from the rich distribution that still exists inside its core pre-trained weights.

Experiments across various tasks demonstrate significant benefits:



Verbalized sampling significantly enhances diversity by 1.6-2.1x over direct prompting, while maintaining or improving quality. Variants like verbalized

sampling-based CoT (Chain-of-Thought) and verbalized sampling-based Multi improve generation diversity even further.

Larger, more capable models like GPT-4.1 and Gemini-2.5-Pro benefit more from verbalized sampling, showing diversity gains up to 2 times greater than smaller models.

Verbalized sampling better retains diversity across post-training stages (SFT, DPO, RLVR), recovering about 66.8% of the base model's original diversity, compared to a much lower retention rate for direct prompting.

Verbalized sampling's gains are independent of other methods, meaning it can be combined with techniques like temperature scaling, top-p sampling to achieve further improvements in the diversity-quality trade-off.

JSON prompting for LLMs

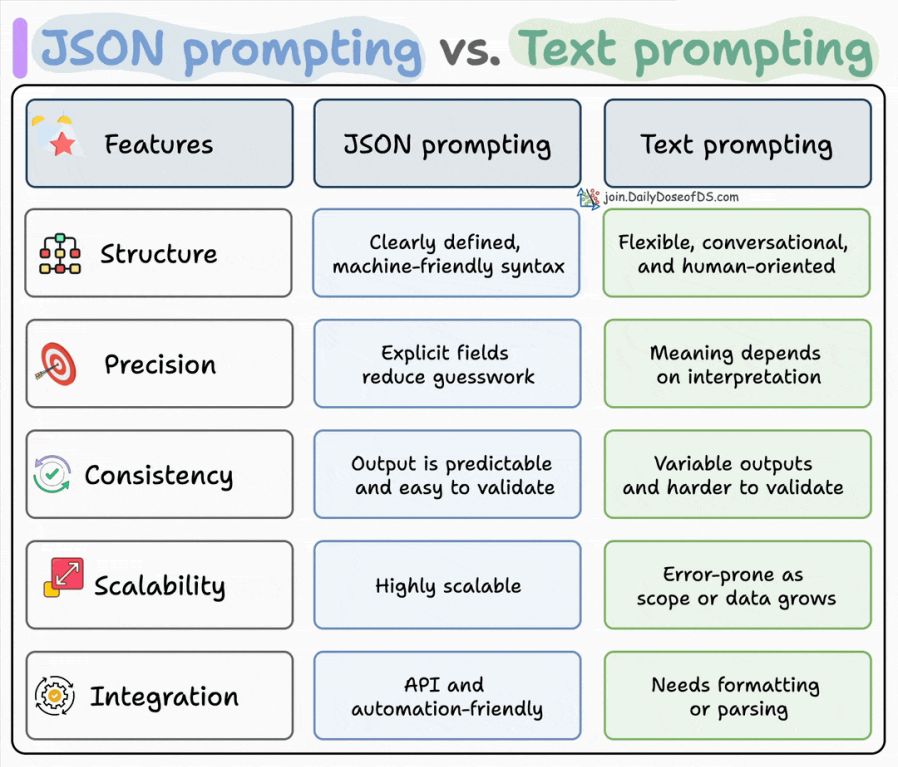
When you give an LLM an open-ended instruction, it has to guess what "good output" looks like.
Sometimes it adds extra commentary, sometimes it skips details, sometimes the formatting changes for no reason.

The problem isn't the model - it's the lack of structure in the prompt.

For tasks like extraction, reporting, automation, or analysis, you need the output to stay consistent every single time.

That's where JSON prompting helps.

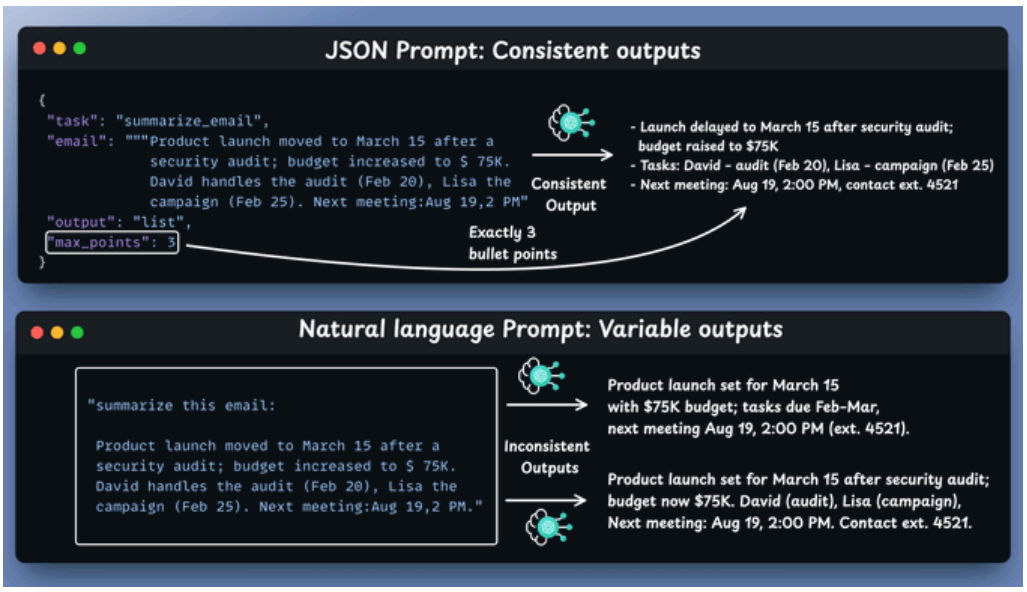
Let us discuss exactly what JSON prompting is and how it can drastically improve your AI outputs!



Natural language is powerful yet vague.

When you give instructions like "summarize this email" or "give me key takeaways," you leave room for interpretation, which can lead to hallucinations.

And if you try JSON prompts, you get consistent outputs:



The reason JSON is so effective is that AI models are trained on massive amounts of structured data from APIs and web applications.

When you speak their "native language," they respond with laser precision!

Let's understand this a bit more.

1) Structure means certainty

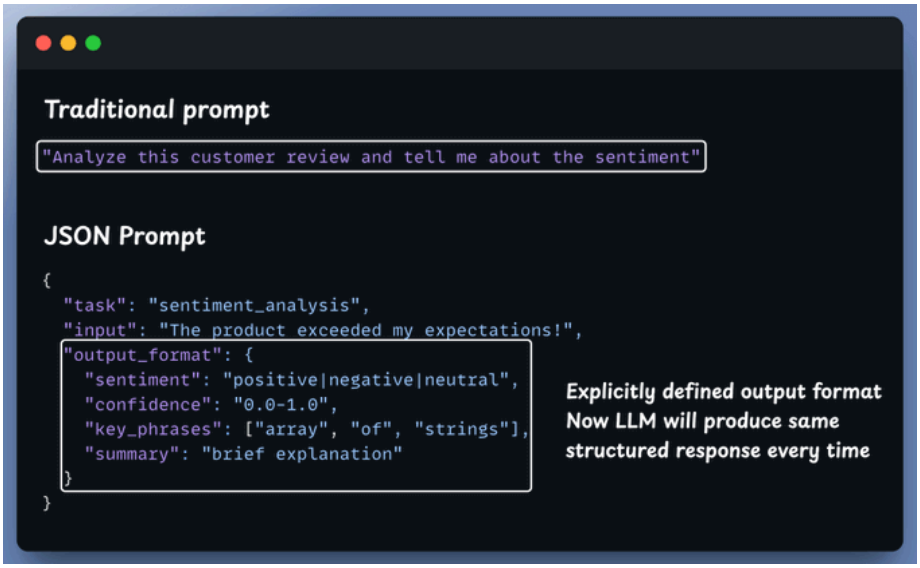
JSON forces you to think in terms of fields and values, which is a gift.

It eliminates gray areas and guesswork.

Here's a simple example:

2) You control the outputs

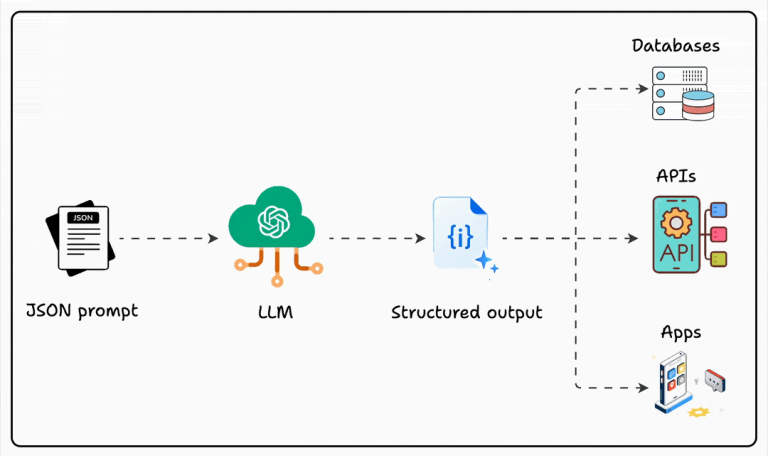
Prompting isn't just about what you ask; it's about what you expect back.



And this works irrespective of what you are doing, like generating content, reports, or insights. JSON prompts ensure a consistent structure every time.

No more surprises, just predictable results!

3) Reusable templates → Scalability, Speed & Clean handoffs



You can turn JSON prompts into shareable templates for consistent outputs.

Teams can plug results directly into APIs, databases, and apps; no manual formatting, so work stays reliable and moves much faster.

So, are json prompts the best option?

Well, good alternatives exist!

Many models excel at other formats:

- Claude handles XML exceptionally well
- Markdown provides structure without overhead

So it's mainly about structure rather than syntax as depicted below:

To summarise:

Structured JSON prompting for LLMs is like writing modular code; it brings clarity of thought, makes adding new requirements effortless, & creates better communication with AI.

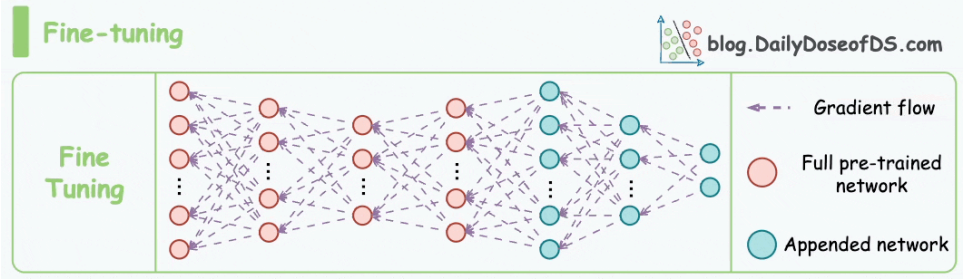
It's not just a technique, but rather evolving towards a habit worth developing for cleaner AI interactions.

Fine-tuning

What is Fine-tuning?

In the pre-LLM era, whenever someone open-sourced any high-utility model for public use, in most cases, practitioners would fine-tune that model to their specific task.

Fine-tuning means adjusting the weights of a pre-trained model on a new dataset for better performance. This is neatly depicted in the diagram below:



When the model was developed, it was trained on a specific dataset that might not perfectly match the characteristics of the data a practitioner wants to use it on.

The original dataset might have had slightly different distributions, patterns, or levels of noise compared to the new dataset.

Fine-tuning allows the model to adapt to these differences, learning from the new data and adjusting its parameters to improve its performance on the specific task at hand.

For instance, consider BERT. It's a Transformer-based language model, which is popularly used for text-to-embedding generation (92k+ citations on the original paper).

It's open-source.

BERT was pre-trained on a large corpus of text data, which might be very very different from what someone else may want to use it on.

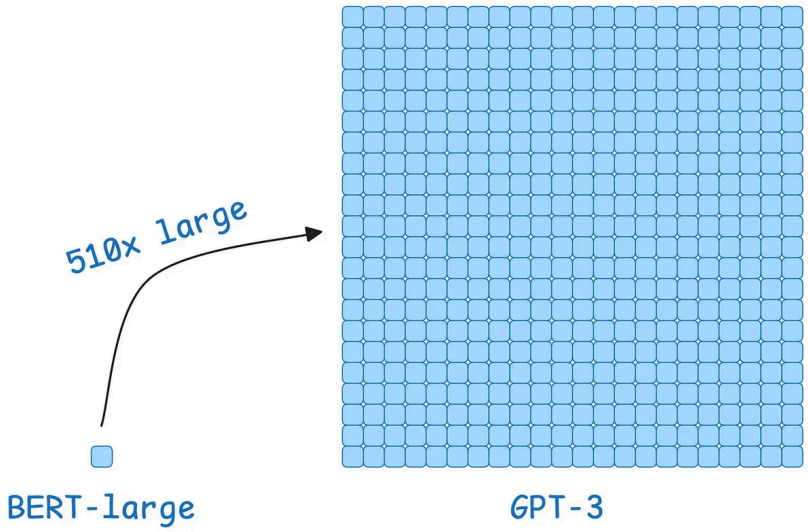
Thus, when using it on any downstream task, we can adjust the weights of the BERT model along with the augmented layers, so that it better aligns with the nuances and specificities of the new dataset.

Issues with traditional fine-tuning

However, a problem arises when we use the same traditional fine-tuning technique on much larger models - LLMs, for instance.

This is because, as you may already know, these models are huge - billions or even trillions of parameters.

Consider the size difference between BERT-large and GPT-3:

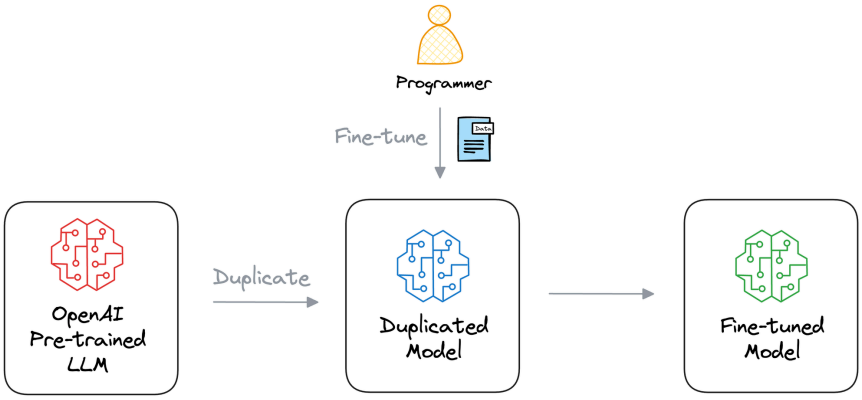


Fine-tuning BERT-large on a single GPU is easy with traditional fine-tuning.

But it's impossible with GPT-3, which has 175B parameters.

That's 350GB of memory just to store model weights (float16 precision).

Imagine OpenAI used traditional fine-tuning within its fine-tuning API:

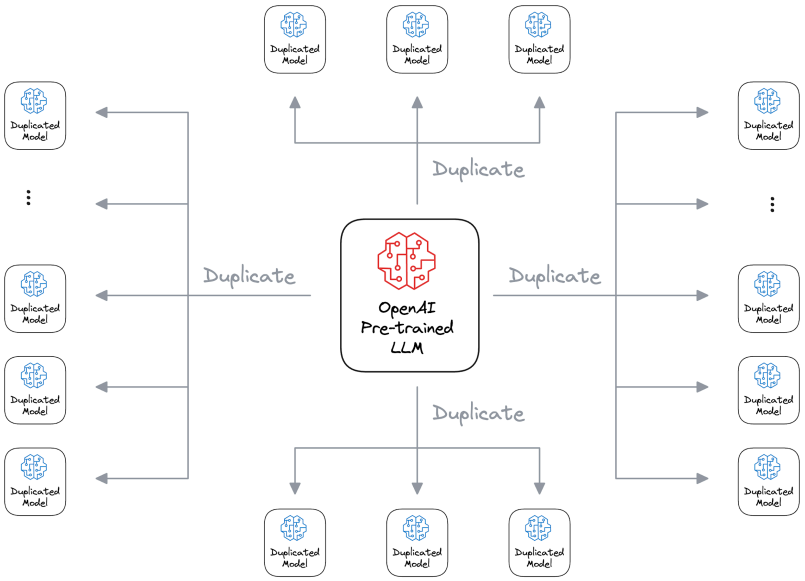


- If 10 users fine-tuned GPT-3 → 3500 GB to store weights.
- If 1000 users fine-tuned GPT-3 → 350k GB to store weights.
- If 100k users fine-tuned GPT-3 → 35M GB to store weights.

And the problems don't end there:

- OpenAI bills solely based on usage.
    - What if someone fine-tunes the model for learning but never uses it?
- Since a request can come anytime, should they always keep the fine-tuned model in memory since loading 350GB is a heavy task?

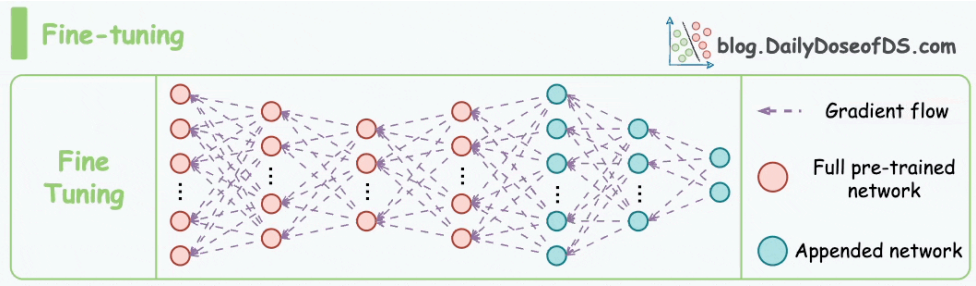
Traditional fine-tuning is just not practically feasible here, and in fact, not everyone can afford to do it due to a lack of massive infrastructure.



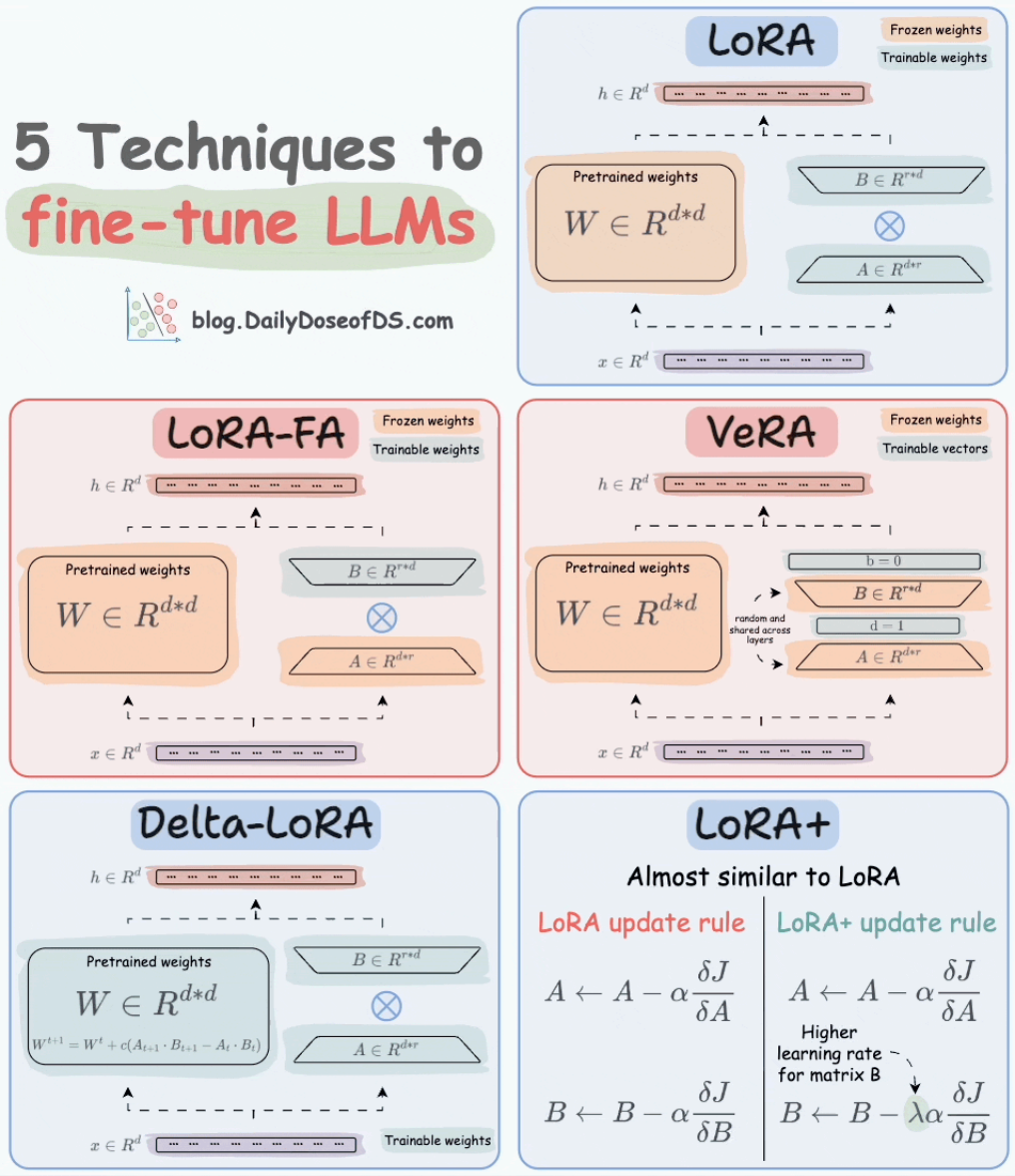
Additionally, maintaining the infrastructure to support fine-tuning requests from potentially thousands of customers simultaneously would be a huge task for them.

5 LLM Fine-tuning Techniques

Traditional fine-tuning (depicted below) is infeasible with LLMs because these models have billions of parameters and are hundreds of GBs in size, and not everyone has access to such computing infrastructure.



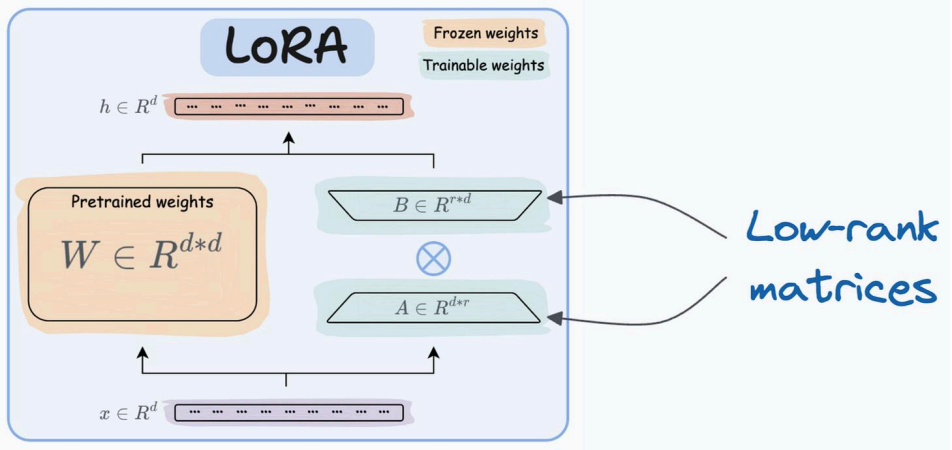
Thankfully, today, we have many optimal ways to fine-tune LLMs, and five such popular techniques are depicted below:



Let's understand these:

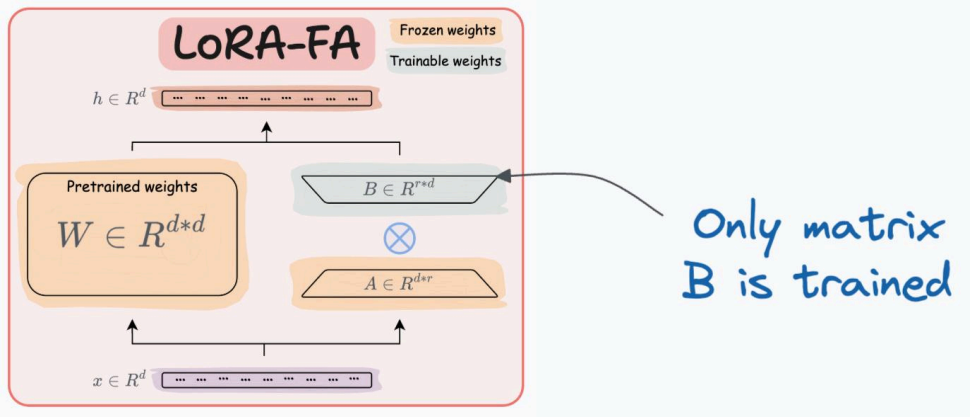
1) LoRA

Add two low-rank matrices A and B alongside weight matrices, which contain the trainable parameters. Instead of fine-tuning W, adjust the updates in these low-rank matrices.



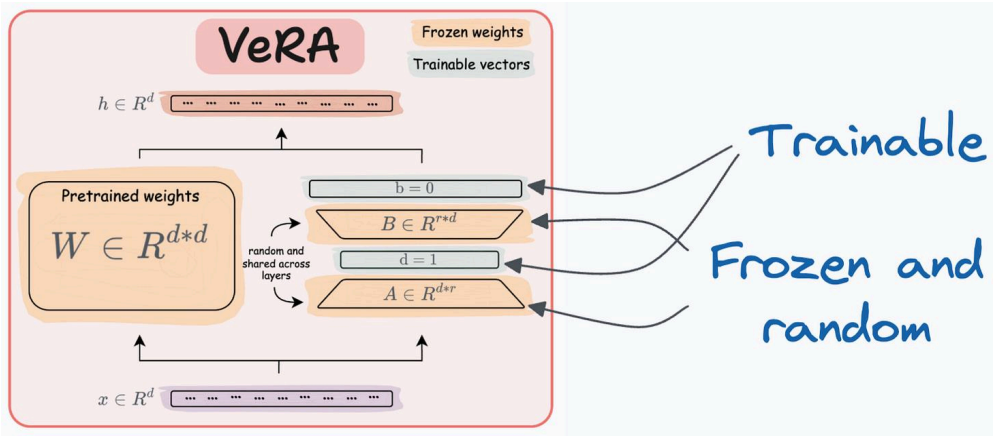
2) LoRA-FA

While LoRA considerably decreases the total trainable parameters, it still requires substantial activation memory to update the low-rank weights. LoRA-FA (FA stands for Frozen-A) freezes the matrix A and only updates matrix B.



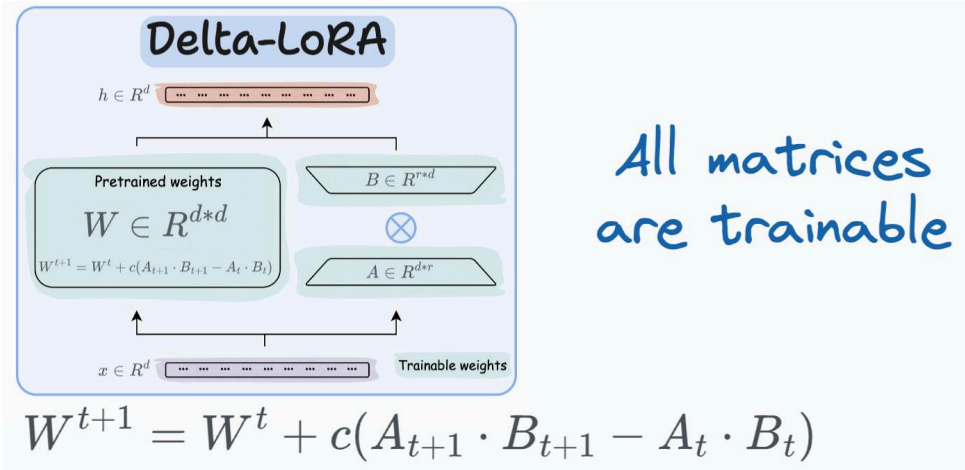
3) VeRA

In LoRA, every layer has a different pair of low-rank matrices A and B, and both matrices are trained. In VeRA, however, matrices A and B are frozen, random, and shared across all model layers. VeRA focuses on learning small, layer-specific scaling vectors, denoted as b and d, which are the only trainable parameters in this setup.



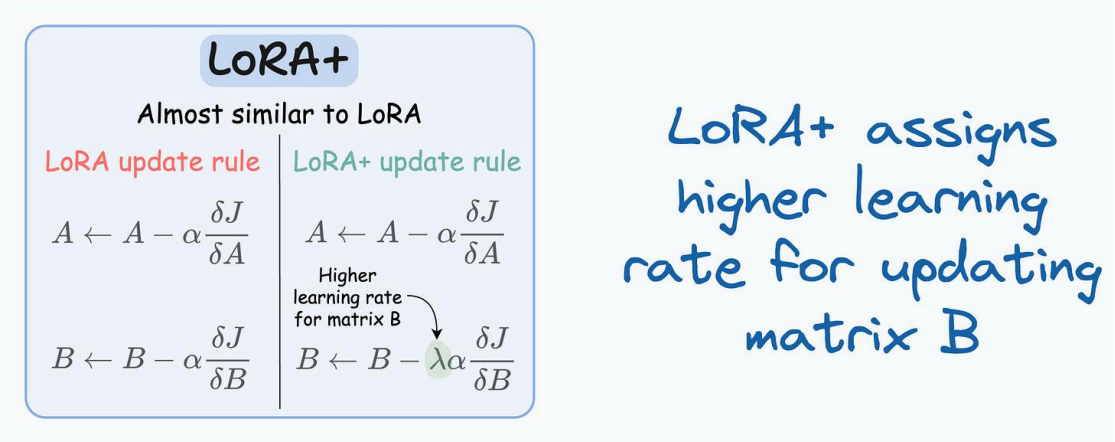
4) Delta-LoRA

Here, in addition to training low-rank matrices, the matrix W is also adjusted but not in the traditional way. Instead, the difference (or delta) between the product of the low-rank matrices A and B in two consecutive training steps is added to W:



5) LoRA+

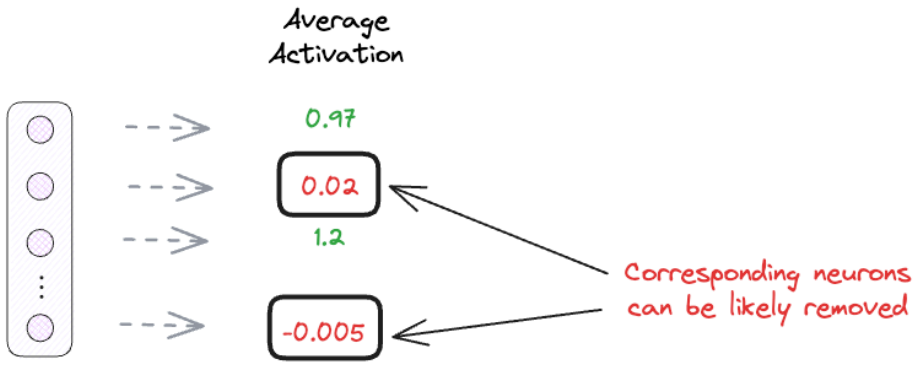
In LoRA, both matrices A and B are updated with the same learning rate. Authors found that setting a higher learning rate for matrix B results in more optimal convergence.



Bonus: LoRA-drop

LoRA-drop observes that not all layers benefit equally from LoRA updates. It first adds low-rank matrices to every layer and trains briefly, then measures each layer's activation strength to see which layers actually matter.

Layers whose LoRA activations stay near zero have minimal influence on the model's output and can be removed.

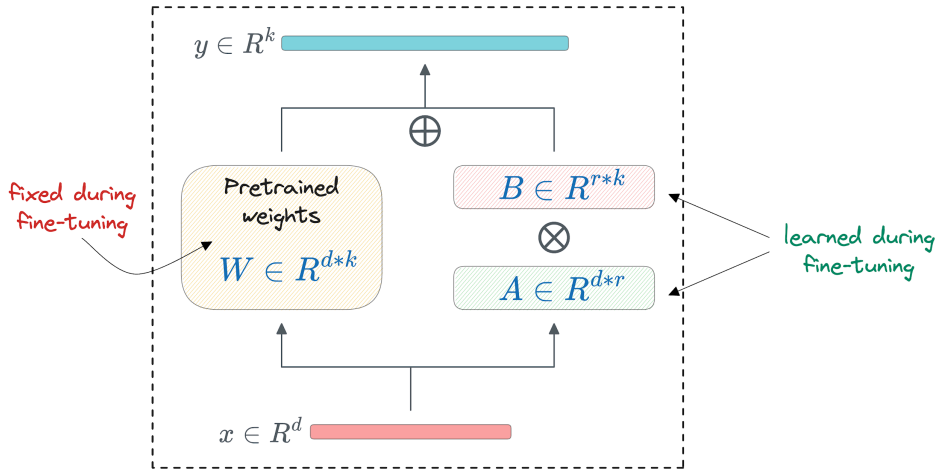


By keeping LoRA only in high-impact layers, LoRA-drop reduces training cost and speeds up fine-tuning with little to no loss in accuracy.

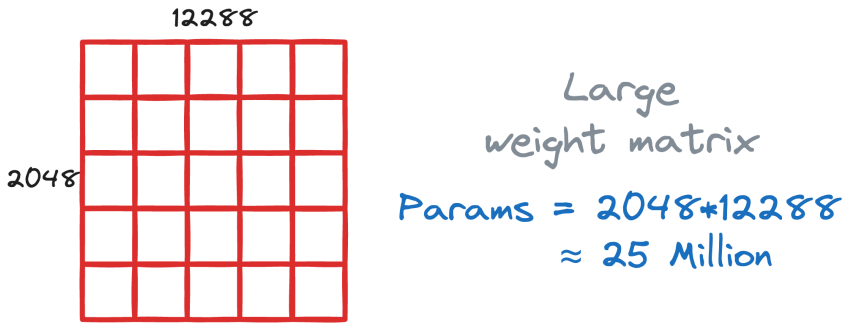
Bonus: Quantized Low-Rank Adaptation (QLoRA)

Quantized Low-Rank Adaptation (QLoRA) is an improvement on the LoRA technique discussed above, which further addresses the memory limitations associated with fine-tuning large models using LoRA.

More specifically, if we recall what we discussed above in LoRA, we saw that we augment the network layers whose weights are W with two matrices A and B.



Now, considering the example where we have 25 Million parameters in the weight matrix W:

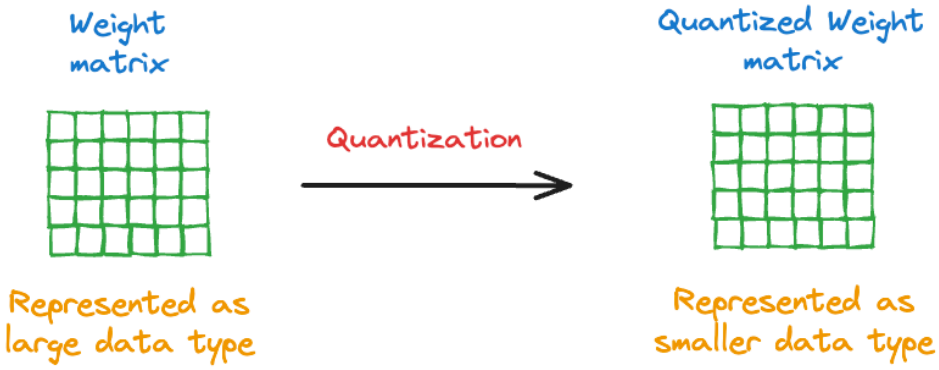


Typically, these 25 million parameters will be represented as float32, which requires 32 bits (or 4 bytes) per parameter. This leads to a significant memory footprint, especially for large LLMs.

This results in a memory utilization of (25 million * 4 bytes/parameter) = 100 million bytes for this matrix alone, which is 0.1GBs.

The idea in QLoRA is to reduce this memory utilization of weight matrix W using Quantization.

As you may have guessed, Quantization involves using lower-bit representations, such as 16-bit, 8-bit, or 4-bit, to represent parameters.



This results in a significant decrease in the amount of memory required to store the model's parameters.

For instance, consider your model has over a million parameters, each represented with 32-bit floating-point numbers.

If possible, representing them with 8-bit numbers can result in a significant decrease (~75%) in memory usage while still allowing for a large range of values to be represented.

Of course, Quantization introduces a trade-off between model size and precision.

While reducing the bit-width of parameters makes the model smaller, it also leads to a loss of precision.

This means the model's predictions become more somewhat approximate than the original, full-precision model.

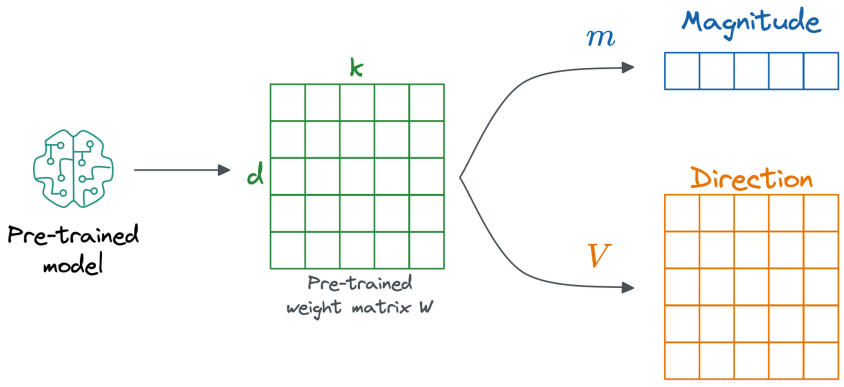
QLoRA does employ some special techniques to preserve the information as much as possible, but there is definitely some trade-off involved.

This somewhat lies along the lines of Quantization in Model compression, which we will cover ahead in the LLM optimization section.

Bonus: DoRA

DoRA (Weight-Decomposed Low-Rank Adaptation) represents a refined approach to fine-tuning large models by addressing a key limitation of LoRA (Low-Rank Adaptation) while preserving its efficiency.

At its core, DoRA builds upon the principles of LoRA but introduces a decomposition step that separates a pretrained weight matrix W into two components: magnitude (m) and direction (V).

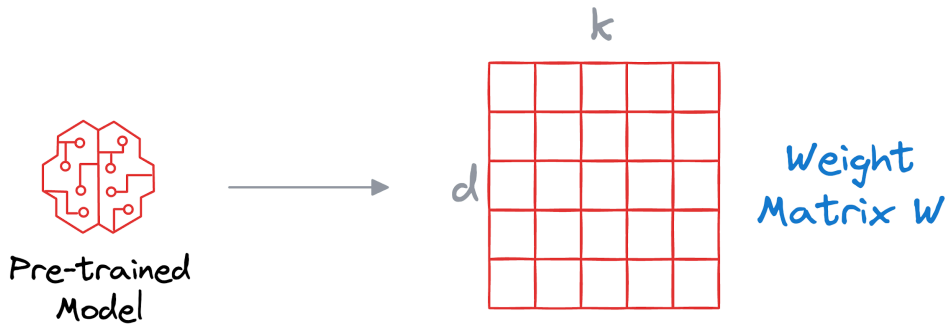


This separation allows the fine-tuning process to target these components independently, improving parameter efficiency and performance.

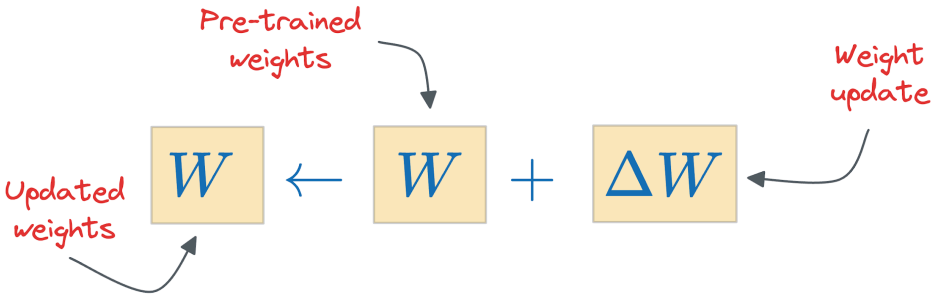
Implementing LoRA From Scratch

Let us understand LoRA in more detail.

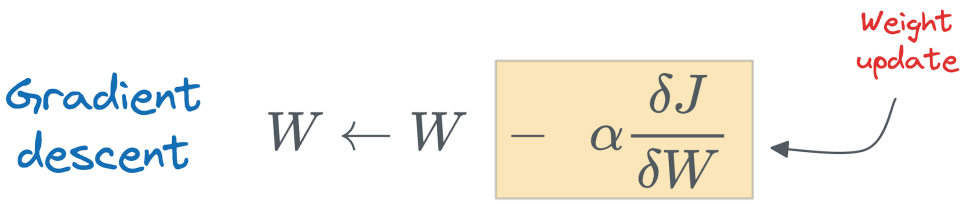
Consider the current weights of some random layer in the pre-trained model are W of dimensions d * k, and we wish to fine-tune it on some other dataset.



During fine-tuning, the gradient update rule suggests that we must add ΔW to get the updated parameters:

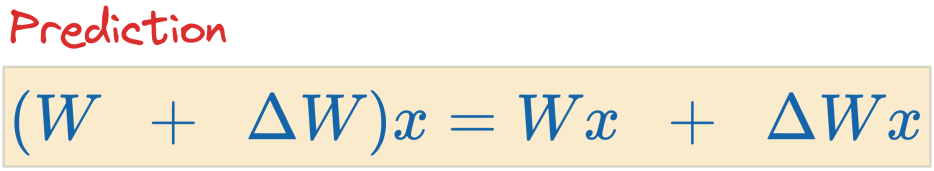


For simplicity, you can think about ΔW as the update obtained after running gradient descent on the new dataset:



Also, instead of updating the original weights W, it is perfectly legal to maintain both matrics, Wand ΔW.

During inference, we can compute the prediction on an input sample x as follows:



In fact, in all the model fine-tuning iterations, W can be kept static, and all weight updates using gradient computation can be incorporated to ΔW instead.

But you might be wondering...how does that even help?

The matrix W is already huge, and we are talking about introducing another matrix that is equally big.

So, we must introduce some smart tricks to manipulate ΔW so that we can fulfill the fine-tuning objective while ensuring we do not consume high memory.

Now, we really can't do much about W as these weights refer to the pre-trained model. So all optimization (if we intend to use any) must be done ΔW instead.

While doing so, we must also remember that currently, both W and ΔW have the same dimensions.But given that W already is huge, we must ensure that ΔW does not end up being of the same dimensions, as this will defeat the entire purpose of efficient fine-tuning.

In other words, if we were to keep ΔW of the same dimensions as W, then it would have been better if we had fine-tuned the original model itself.