Primer on fine-tuning of Large Language Models



Primer on fine-tuning of Large Language Models *(11th edition, 2026.07.21)*

1. What is a Large Language Model (LLM)?
   - 1.1. The basic mechanisms common to all LLMs
   - 1.2. The purposes of creating and using LLMs
   - 1.3. The procedure of training an LLM
   - 1.4. Information used to train an LLM; Alignment; Multi-Token Prediction
     - 1.4.1. Tokens; tokenization; reference table for tokens as words in books, stories, etc.
     - 1.4.2. Weights; Active Parameters; Vector Spaces, Vector Databases; Pruning
     - 1.4.3. Size categories of LLMs
     - 1.4.4. Dense LLMs
     - 1.4.5. Mixture of Experts LLMs
     - 1.4.6. Hybrid LLMs
     - 1.4.7. Quantization
   - 1.5. Types of datasets used for training an LLM
   - 1.6. The Context Length of an LLM
   - 1.7. Hallucinations
     - 1.7.1. Controlling the output of LLMs: Temperature, Top K, Top P, etc
   - 1.8. Prompt Engineering
     - 1.8.1. Zero-shot, one-shot, and few-shot learning
     - 1.8.2. The System Prompt
   - 1.9. Retrieval Augmented Generation (RAG)
   - 1.10. Evaluation of LLMs

2. Fine-Tuning a Large Language Model
   - 2.1. Core concepts
   - 2.2. Why and when to fine-tune an LLM
     - 2.2.1. Degradation of the base LLM
   - 2.3. Common formats of datasets for fine-tuning
   - 2.4. Dataset structure for retrieval optimization
   - 2.5. Fine-Tuning by LoRA
   - 2.6. Adapters
   - 2.7. Hyperparameters for fine-tuning: Number of Epochs, Learning Rate, Batch Size, etc.
   - 2.8. Fine-tuning use cases
   - 2.9. Fine-tuning on a small dataset (a few examples, large text - a book split into a few chapters)
   - 2.10. Labeling; using the “system” tag for inculcating (role-play!) a “persona” in the fine-tuned LLM
   - 2.11. Personas, or the identity of a fine-tuned LLM: proper implementation of identity; example

3. Glossary; hyperparameters reference table; popular LLM families (in 2025/2026)

4. Software for interaction with LLMs
   - 4.1. Text-to-Text
   - 4.2. Text-to-Image/Video
   - 4.3. LlamaFactory; HuggingFace; llama.cpp; LLamaSharp

5. *Python* programming language
   - 5.1. MS Windows: Tips for optimization of running Python scripts in the Command Window

Large Language Models (LLMs) are powerful artificial intelligence (AI) systems that can generate, understand, and manipulate human language. Large Language Models represent a significant breakthrough in artificial intelligence, offering advanced capabilities in natural language understanding and generation. From their foundational mechanisms and training processes to the datasets they are trained on, LLMs are designed to process vast amounts of textual data and perform a variety of language-related tasks. Their evaluation is equally crucial, ensuring that they are accurate, fair, and suitable for real-world applications. As technology continues to evolve, the development of LLMs will undoubtedly shape the future of human-computer interactions. These models have become increasingly important in numerous fields, ranging from natural language processing (NLP) to applications like chatbots, content creation, and even scientific research.

This guide will provide an in-depth exploration of LLMs, explaining their basic mechanisms, purposes, training processes, and fine-tuning techniques.

Source of text information: ChatGPT (https://chatgpt.com/), IBM, GitHub; be advised that ChatGPT may be wrong in some aspects.
Date of prompting+retrieval: 2025.12.27-2026.07.21
Concept, contents, structure, partial formatting: SINAPSA Infocomplex®™ http://sinapsaro.ro

Foreword

Large Language Models have moved from research curiosities to everyday tools with remarkable speed. They write, summarize, translate, tutor, and assist across domains that once seemed far beyond the reach of software. Yet the most important question facing practitioners today is no longer *what these models can do*, but *how we shape what they do*. This booklet-guide is written for that exact moment.

Pretrained models arrive with vast linguistic knowledge, but they are inherently general-purpose. Fine-tuning is the bridge between general capability and practical usefulness. It is how a model learns to follow instructions instead of merely predicting text, how it adapts to a specialized domain, how it becomes safer, more reliable, and more aligned with human expectations. In short, fine-tuning is where intent enters the system.

This guide does not assume that readers are building models from scratch, nor does it treat fine-tuning as a black box. Instead, it aims to provide a clear, grounded map of the terrain: what fine-tuning is, why it works, and how to approach it thoughtfully. Whether you are a student, engineer, researcher, or technically curious reader, the goal is to make the core ideas accessible without flattening their complexity.

You will find that fine-tuning is not merely an optimization step-it is a design process. Choices about data, objectives, and evaluation criteria directly shape model behavior. A model fine-tuned for helpfulness behaves differently from one tuned for strict factual precision. A system optimized for safety may refuse requests that another model would answer eagerly. These differences are not accidents; they are consequences of deliberate, and sometimes implicit, decisions.

This booklet also emphasizes practical perspective. Fine-tuning is often portrayed as computationally expensive or reserved for large organizations. In reality, many impactful adaptations rely on relatively small datasets, careful prompt structuring, or targeted preference learning. Understanding these techniques empowers more people to work responsibly with powerful models, while also highlighting the limits of what fine-tuning can-and cannot-fix.

Equally important is what this guide does *not* promise. Fine-tuning does not eliminate uncertainty, bias, or error. It does not create perfectly aligned systems, nor does it replace the need for human oversight. Instead, it offers a set of tools for steering behavior, managing trade-offs, and improving performance within known constraints. Approaching fine-tuning with clear expectations is as important as mastering the techniques themselves.

As you read, you are encouraged to think critically. Ask what assumptions are embedded in a dataset. Ask what behaviors are being rewarded or discouraged. Ask how success is being measured, and who benefits from those measurements. Fine-tuning is powerful precisely because small changes can have large effects-and with that leverage comes responsibility.

This booklet-guide is meant to be a companion rather than a final authority. The field is evolving rapidly, and best practices will continue to change. What should endure is the mindset it promotes: treat fine-tuning as an intentional act, grounded in both technical understanding and thoughtful judgment.
If this foreword sets the tone, then the text that follows provide the tools. Together, they aim to help you not just fine-tune models-but to do so with clarity, care, and purpose.

1. What is a Large Language Model (LLM)?

A Large Language Model (LLM) is a type of machine learning model that is designed to understand, process, and generate human language in a way that mimics human-like understanding. These models are typically built using deep learning techniques and trained on vast amounts of text data. LLMs, such as OpenAI's GPT (Generative Pre-trained Transformer) series, are based on neural network architectures like the Transformer, which allows them to process large amounts of text efficiently and generate coherent, contextually appropriate responses.
LLMs do not "understand" language in the same way humans do, but they can predict the likelihood of the next word or phrase in a sequence based on patterns learned during their training process. This ability allows them to generate text, answer questions, summarize information, translate languages, and even engage in conversations.

1.1. The Basic Mechanisms Common to All LLMs

The key mechanism behind LLMs is the **Transformer architecture**, which has revolutionized natural language processing since its introduction in 2017. The Transformer architecture allows LLMs to handle large amounts of data and capture long-range dependencies within text efficiently. The main components of a Transformer include:

- **Attention Mechanisms**: These allow the model to "focus" on different parts of the input text when making predictions, enabling it to capture relationships between words and phrases over long distances within the text.
- **Self-Attention**: A particular type of attention mechanism that allows the model to weigh the importance of different words in a sentence relative to each other, helping the model understand context.
- **Layers**: LLMs are typically composed of multiple layers of neural networks, each one processing the input text and passing the results to the next layer, allowing the model to progressively refine its understanding.

These mechanisms, along with large-scale parallel processing, enable LLMs to process vast amounts of information and generate coherent, contextually relevant text.

1.2. The Purposes of Creating and Using LLMs

LLMs serve a variety of purposes, especially in fields related to artificial intelligence, linguistics, and data science. Some of the primary uses of LLMs include:

- **Natural Language Processing (NLP)**: LLMs are widely used in NLP tasks such as machine translation, sentiment analysis, and summarization. They enable machines to understand and process human language, making them useful for everything from translating languages to analyzing customer feedback.
- **Conversational Agents**: LLMs power virtual assistants and chatbots, such as Siri, Alexa, or customer service bots, allowing them to have more natural and sophisticated conversations with users.
- **Content Generation**: LLMs are also used for content creation, including writing articles, generating code, composing poetry, and more. Their ability to mimic human language has made them popular tools for generating text-based content automatically.
- **Search Engines and Information Retrieval**: LLMs can be integrated into search engines to improve search result accuracy, making searches more intuitive and relevant to users' queries.
- **Research and Data Analysis**: LLMs are used to sift through large volumes of text data to identify patterns, summarize information, or even generate new research hypotheses.

In the context of Large Language Models, **encoders** and **decoders** refer to different parts or architectures used to process and generate language. These terms come from the transformer architecture, which is widely used in LLMs like BERT, GPT, and T5. Here's a concise breakdown:

Encoder-Only Models:

- **Purpose**: These models focus on **understanding** input data (e.g., text).
- **How it works**: The encoder processes the input sequence (like a sentence) and generates a dense, contextual representation of that input. This representation captures the meaning of the input text.
- **Example**: **BERT** is an encoder-only model. It's trained to predict missing words in a sentence, which helps it understand the context and relationships between words.

Decoder-Only Models:

- **Purpose**: These models focus on **generating** text, often based on a prompt or context.
- **How it works**: The decoder generates output tokens one by one, predicting the next word based on previous words and the context it has learned. It does not explicitly encode input sequences but instead generates text directly.
- **Example**: **GPT** is a decoder-only model. It generates text by predicting the next word in a sequence, given the prompt or context.

Encoder-Decoder Models:

- **Purpose**: These models use both an encoder and a decoder and are often used for tasks like **translation** or **summarization**.
- **How it works**: The encoder processes the input (e.g., a sentence in English), creating a representation. The decoder then takes this representation and generates an output (e.g., the translated sentence in French).
- **Example**: **T5** or **BART** are encoder-decoder models, and they excel in tasks that require transforming one kind of text into another (e.g., summarization, translation).

Summary:

- **Encoder-only** (e.g., BERT): Understands input text.
- **Decoder-only** (e.g., GPT): Generates output text.
- **Encoder-decoder** (e.g., T5, BART): Transforms input text to output text (like translation or summarization).

1.3. The Procedure of Training an LLM

Training a Large Language Model is an intensive process that involves multiple steps:

- **Preprocessing**: The first step is to collect and clean large datasets of text. This data is often sourced from books, websites, articles, and more. The text is then tokenized (broken into smaller pieces, such as words or sub-words) and converted into a numerical format that the model can process.
- **Model Architecture Setup**: The neural network architecture (like the Transformer) is defined, specifying how layers, attention mechanisms, and parameters are structured.
- **Training Process**: During training, the model learns by adjusting its parameters (called **weights**) to minimize the error in its predictions. This process is done using a method called **gradient descent**, where the model iteratively makes predictions, compares them to the actual values, and updates its weights accordingly.
- **Fine-tuning**: After pre-training, LLMs can be fine-tuned for specific tasks, such as answering questions or generating specific types of text. This involves further training on a specialized dataset.

Training LLMs requires vast amounts of computational power, often involving hundreds or thousands of GPUs or TPUs over extended periods.

1.4. Information Used to Train an LLM; Alignment

The primary information used to train LLMs is textual data. However, this data must undergo several steps of transformation to be usable by the model.

A**lignment** means:
How well the model's behavior matches human values, intentions, and expectations.
In practice, it's about making sure the model **does what we want it to do** and **doesn't do things we consider harmful, misleading, or unwanted**.

What alignment is trying to solve

Raw LLMs are trained to **predict the next** token based on patterns in data. That alone doesn't guarantee that the model will:

- Be helpful rather than confusing
- Be honest rather than confidently wrong
- Be safe rather than harmful
- Follow instructions instead of exploiting loopholes

Alignment bridges the gap between:

- *"This text is statistically likely"*
  and
- *"This response is appropriate and desirable for humans"*

Key aspects of alignment

1. Intent alignment
Does the model correctly understand and follow what the user *means*?
Example:

- User: "Explain this simply"
- Aligned model: adjusts language level
- Misaligned model: gives a dense technical lecture

2. Value alignment
Does the model act according to human norms and values?
This includes:

- Avoiding harm
- Respecting fairness and inclusivity
- Not promoting illegal or dangerous actions

3. Behavioral alignment
Does the model behave well *in real interactions*?
Examples:

- Admitting uncertainty instead of making things up
- Refusing unsafe requests politely
- Staying on topic and consistent

4. Robustness to misuse
Does the model stay aligned even when users:

- Phrase things oddly
- Try to trick it
- Ask leading or manipulative questions

How alignment is achieved (simplified)

Alignment is **not automatic** from training on text. Common techniques include:

- **Human feedback**
  Humans rate model outputs as better or worse.
- **Reinforcement Learning from Human Feedback (*RLHF*)**
  The model is trained to prefer responses humans like and approve.
- **Rule-based constraints**
  Hard limits on certain behaviors.
- **Constitutional or policy-guided training**
  Models learn to follow explicit principles.

Why alignment is hard

- **Humans don't agree** on values
- Context matters (what's acceptable in one case isn't in another)
- Models can generalize in unexpected ways
- Avoiding harm without being overly restrictive is a delicate balance

This is often called the **alignment problem**.

One-sentence summary

Alignment in LLMs is the effort to ensure that models reliably act in ways that are helpful, honest, and safe according to human values-not just statistically plausible.
If you want, I can also explain:

- alignment vs safety
- inner vs outer alignment
- why alignment matters for future AI systems

Multi-Token Prediction (MTP)

Normally, an LLM works like this:

1. It reads everything so far.
2. It predicts the *next* (only one) word or token.
3. Then it repeats that process again and again.

So generation is basically: *one tiny step at a time*
That's accurate, but also slow.

What ***Multi-Token Prediction*** changes

With MTP, the LLM tries to predict several upcoming tokens at once instead of only the immediate next one.

So instead of thinking:
"What comes next?"
it tries to think:
"What are the next few things likely to come next?"

For example, if the text starts with:
"The capital of France is ..."

A normal model predicts:

- "Paris"

Then after that:

- "."

Then after that:

- maybe the next sentence.

An MTP-capable system may try to predict:

- "Paris"
- "."
- "It"
- "is"
- …

all in one shot.

Why this matters

Faster generation:

This is the biggest reason.
Generating one token at a time is expensive because the model has to run repeatedly. If it can confidently predict multiple tokens together, text generation becomes much faster.
This is especially important for:

- chatbots
- coding assistants
- real-time AI systems

Better planning:

Standard next-token prediction is very local:

- it focuses heavily on the immediate next step.

MTP encourages the model to think slightly farther ahead, which can improve:

- coherence
- sentence structure
- code generation
- long reasoning chains

Helps speculative decoding:

A common modern speed trick is:

- a small fast model drafts several tokens
- a larger model checks them

MTP fits naturally into this idea because the system is already working with chunks of future text instead of single-token steps.

Important nuance

Even with MTP, most deployed LLMs still *output* text sequentially to the user.

The difference is mostly:

- how the model is trained
- how inference is optimized internally

So users may not notice "multi-token prediction" directly, except that:

- responses feel faster
- generation may flow more smoothly

Simple analogy:
Normal LLM:
walking one stepping stone at a time
MTP:
looking ahead and planning the next several stones together
That forward-looking behavior is the key idea.

1.4.1. Tokens; tokenization; reference table for tokens as words in books, stories, etc.

Text data is first broken down into **tokens**, which are smaller units like words, sub-words, or characters. Tokenization is the process of converting raw text into these smaller units. Tokenization enables the model to handle different languages, variations in spelling, and even made-up words.
For example, the sentence "I love dogs" could be tokenized into individual words: ["I", "love", "dogs"]. In some cases, sub-word tokenization is used, where the word "dogs" might be split into "dog" and "s."
The choice of tokenization strategy affects how well the model can handle different languages or variations in text.

For English language: A practical way to estimate text length (words, pages, "book size") from LLM tokens. These are **rules of thumb**, not exact conversions.

Token → Word Conversion
1 token ≈ 0.75 words
Or inversely: **1 word ≈ 1.3 tokens**

This varies by language and style:
- Simple prose → fewer tokens per word
- Technical text / lots of punctuation → more tokens per word
- Non-English languages can differ slightly

Quick Reference Table: tokens to words

Tokens	*Words (approx)*	*Pages (≈250-300 words/page)*	*What it feels like*
1,000	750	3	Short article / flash fiction
5,000	3,750	12–15	Short story
10,000	7,500	25–30	Long short story
20,000	15,000	50–60	Novelette
40,000	30,000	100–120	Short novel / novella
80,000	60,000	200–240	Typical novel
120,000	90,000	300–360	Long novel
200,000	150,000	500–600	Epic fantasy-scale book

By Literary Category

Short forms
• **Flash fiction:** ~500–1,500 words → 700–2,000 tokens
• **Short story:** ~1,500–7,500 words → 2,000–10,000 tokens

Mid-length
• **Novelette:** ~7,500–17,500 words → 10,000–23,000 tokens
• **Novella:** ~17,500–40,000 words → 23,000–53,000 tokens

Full books
• **Typical novel:** ~70,000–100,000 words → 90,000–130,000 tokens
• **Epic / dense novel:** ~120,000–200,000+ words → 160,000–260,000+ tokens

Important for LLM Use:
If an LLM has a context window of 128k tokens, that's roughly:
~95,000 words
(a full novel-length manuscript)

... but in practice:
o You need to reserve tokens for prompts + instructions
o So usable space is smaller (often 50–70%)

Simple Formula
If you want to estimate quickly:
• words≈tokens×0.75
• pages≈words÷275

Example
If you have 50,000 tokens:
• ≈ 37,500 words
• ≈ 135 pages
• → A solid short novel / novella

1.4.2. Weights; Active Parameters; Vector Spaces; Vector Databases; Pruning

Weights are the **parameters** (numbers) that a neural network uses to make predictions. During the training process, the model learns the best values for these weights so that it can predict the most likely next word or phrase given a sequence of tokens.
Weights are values that determine the strength of connections between different neurons in the model's neural network.

Each weight in the model corresponds to a particular connection between neurons in the neural network. As the model processes more data, these weights are adjusted to improve its performance on language tasks.

These weights are learned during training, based on the data the model is exposed to. They allow the model to make decisions about which pieces of information to prioritize when generating or understanding text.

Here's how they work:

- **Neural Networks**: LLMs are based on neural networks, which are structures made up of layers of artificial "neurons." Each neuron receives input, performs a calculation, and passes the result to the next layer.
- **Weights**: Each connection between neurons has a weight, which is a number that determines how much influence one neuron has on another. In the simplest terms, a higher weight means a stronger connection, while a lower weight means a weaker one.
- **Training**: During training, the model adjusts these weights through a process called **backpropagation**. This process tweaks the weights based on how well the model's predictions match the actual output. Over time, the model becomes better at understanding patterns in language by adjusting the weights.

Typically, **weights do not change or adjust during inference**. Inference is the phase where a trained model makes predictions on new, unseen data, and the model's weights are **fixed** during this process.

However, there are **rare situations** where weights might still adjust during inference:

1. **Online Learning**: In some applications, the model continues to learn as it processes new data in real-time. This is known as **online learning** or **incremental learning**, where weights can be updated during inference if new data is used to adapt the model continuously.

2. **Meta-Learning (Learning to Learn)**: Some advanced techniques like **meta-learning** might involve the model adjusting its weights in response to new tasks or environments, even during inference. The model doesn't adjust weights for a specific task, but it may adapt faster when presented with similar tasks.
3. **Reinforcement Learning (RL)**: In certain RL systems, like those used in real-time control or decision-making, the model may adjust its policy (and indirectly, the weights) during inference, as it interacts with the environment and receives feedback.

For the vast majority of deep learning models, though, weights remain fixed during inference after training is complete.

An LLM does not store "distances between tokens" as labeled facts like "0.34 = distance(X, Y)".

Instead, *meaning* and *relationships* emerge from how numbers are arranged in very specific places inside very specific matrices, and *position is everything*.

Let's walk through this step by step, from prompt → understanding → response.

1. What the "weights" really are (and are not)

Weights are not a lookup table of token-to-token distances.
They are:

- Parameters in matrices
- Used in matrix multiplications
- Each matrix has a fixed role in the architecture

A weight only "means" something because of:

1. Where it lives (which matrix, which row, which column)
2. What computation uses it
3. Which tokens activate it

Think of weights like the resistors on a circuit board:

- A resistor has no idea what "current" or "signal" means
- Its *location in the circuit* determines its function

2. Tokens are vectors, not labels

When you type:
The cat sat
The model does not internally think:
"cat" = animal

Instead:

1. Each token → an embedding vector
2. *cat* → [0.12, -1.7, 0.03, ...]
3. These vectors live in a shared coordinate space
4. Meaning is encoded as direction and magnitude, not names

So:
- "cat" and "dog" point in similar directions
- "cat" and "quantum" don't

No one tells the model *which* vector belongs to *which* word at runtime — the embedding table index ensures that.

3. "How does 0.34 belong to X–Y and not A–B?"

Short answer:
Because it's in a specific cell of a specific matrix, and only X and Y ever multiply through that cell.
Longer answer:
Example: an embedding matrix
Embedding matrix E:
[token_id, dimension]
- Row 5321 → "cat"
- Column 17 → "furry-ness axis"
- Value = 0.34

That 0.34:
- Is only accessed when token_id = 5321
- Has no effect on token 812 ("quantum")
- Cannot "move" or be reassigned

There is no ambiguity because:
- Indices are fixed
- Matrix shapes are fixed
- Computation graph is fixed

The model doesn't *ask* what a number refers to — the architecture enforces it.

4. Where relationships come from: attention (this is the key part)

Understanding a prompt is mostly about self-attention.
For each token, the model computes:
- Query (what am I looking for?)
- Key (what do I contain?)
- Value (what information I carry?)

These are computed via different learned matrices:
$Q = X \cdot Wq$
$K = X \cdot Wk$
$V = X \cdot Wv$

Then:
attention(token i → token j) = softmax(Qi · Kj)

So:
- The model does NOT store "cat relates to sat"
- It computes *on the fly*:

"How relevant is token j to token i right now?"
That relevance score *feels* like a "distance", but it's:
- Context-dependent
- Directional
- Computed fresh every layer

5. Why the model doesn't confuse X–Y with A–B

Because:
1. Tokens have different embeddings
2. They generate different Q/K/V vectors
3. They interact through position-specific attention
4. Each layer refines context

So even if two weights both equal 0.34, they:
- Live in different matrices
- Affect different dimensions
- Are multiplied by different activations

Same number ≠ same meaning.

6. How a prompt is "understood"

"Understanding" is an emergent illusion, but here's what actually happens:

Step-by-step
1. Tokenization
2. "Explain attention" → [Explain, attention]
3. Embedding
   Each token → vector
4. Repeated attention layers
   - Tokens look at each other
   - Meaning gets contextualized
   - Abstract patterns form (question, request, explanation)
5. Final hidden state
   The last token's vector represents:

"Given everything so far, what comes next?"
No symbolic reasoning. No intent object. Just vector transformations.

7. How a "pertinent reply" is constructed

At each step, the model predicts:
P(next_token | all_previous_tokens)
- It does not plan the whole answer
- It generates one token at a time
- Each token reshapes the context for the next one

The reason replies feel coherent is because:
- Training rewarded continuations that humans judged as "good explanations"
- Those patterns are baked into the weights

8. The big mental shift (important)

If you take only one thing away:
An LLM does not know what a weight refers to.

Meaning arises because computation is structured, not because numbers are labeled.

There is:
- No table of facts
- No symbolic map
- No "distance(X,Y) = 0.34"

Only:
- Fixed architecture
- Learned matrices
- Contextual vector flow

Example of Weights in Action:

Imagine you're trying to predict the next word in the sentence: _"The cat sat on the __."
- The model looks at the words before the blank, like "The cat sat on the."
- Each word is processed through the neural network, and each word has a certain weight associated with how much it should influence the next word prediction.
- Weights help the model decide if the next word is more likely to be "mat," "floor," "roof," etc.

For instance:
- If the weights associated with the word "cat" are stronger for predicting objects that cats sit on, the model might give higher priority to "mat" or "cushion."
- If the weights are different, it could predict something like "floor."

The final model has millions (or even billions) of weights that determine how it responds to different inputs.

Training is where all the "meaning" gets baked in.

1. What training is *actually* optimizing

An LLM is trained to do exactly one thing:
Predict the next token correctly.

That's it. No objectives like:
- "understand language"
- "learn concepts"
- "build a world model"

Those emerge as *side effects*.

Formally, training minimizes **cross-entropy loss**:
Loss = − log P(correct_next_token | context)
So if the model predicts:
- correct token → small loss
- wrong token → large loss

Everything else follows from this.

2. The training data: where structure comes from
The model is shown **billions of sequences** like:
"The capital of France is Paris"
At training time, it sees:
Input: The capital of France is
Target: Paris
Then:
Input: The capital of France is Paris
Target: <next token>
Key point:

- The model never sees labels like *"fact"* or *"entity"*
- Only **raw token sequences**

But natural language is *highly structured*, so statistics carry meaning.

3. One training step (microscopic view)
Let's zoom in on **one gradient update**.
Step 1: Forward pass

1. Tokens → embeddings
2. Go through attention + MLP layers
3. Output a probability distribution over next tokens

Example:
Paris: 0.72
London: 0.18
Banana: 0.0003
Correct answer: Paris

Step 2: Compute loss
Loss punishes the model for:

- Not assigning *enough* probability to Paris
- Assigning probability to wrong tokens

Step 3: Backpropagation (this is where learning happens)
Using calculus:
"Which weights caused the error, and how should they change?"
Each weight gets a gradient:
∂Loss / ∂weight
Meaning:

- If increasing this weight would help → increase it
- If it hurts → decrease it

This is **local**, mechanical, and blind.
No weight ever "knows":
"I represent capital cities"

4. Why specific weights come to represent relationships
Weights specialize because they are repeatedly useful for the same patterns.
Example:
Across millions of examples:
X is the capital of Y → Z
Certain attention heads learn:

- "Look for a country name"
- "Look backward to proper nouns"
- "Pass that information forward"

If a weight contributes to:

- Correct predictions many times → reinforced
- Wrong predictions → weakened

Over time:

- Some heads specialize in syntax
- Some in entities
- Some in long-range reference
- Some in formatting

This specialization is *emergent*, not assigned.

5. Why numbers don't get mixed up
Back to the question:
Why doesn't 0.34 "forget" what it belongs to?
Because during backprop:

- Gradients flow **through the same computational paths**
- Only weights on those paths are updated
- Other weights are untouched

So:

- A weight connecting token-dimension 17 → head 3 → layer 6
- Only ever affects *that* pathway
- Only ever gets updated when *that* pathway is active

Training enforces consistency automatically.

6. How "concepts" form without being defined
There is no "cat neuron".
Instead:

- A **distributed pattern** across many dimensions
- Spanning many layers
- Activated in many contexts

"Cat" emerges as:

- Similar embeddings
- Similar attention behavior
- Similar downstream predictions

This is why:

- Concepts are fuzzy
- Meanings overlap
- Edge cases confuse the model

It's also why LLMs generalize.

7. Why the model starts to *appear* to reason
Because training data contains reasoning-like patterns:
If A and B, then C
Therefore C
So the model learns:

- Token patterns that *look like* reasoning
- Intermediate steps that increase prediction accuracy

The model isn't reasoning in a symbolic sense — but it **imitates the process** well enough to be useful.

8. Why scale matters (a lot)
Small models:
- Memorize
- Overfit
- Don't abstract well

Large models:
- Are forced to compress patterns
- Discover reusable structures
- Learn higher-level representations

This is why:
Understanding appears suddenly at scale
Nothing magical happened — the model just crossed a threshold where abstraction became cheaper than memorization.

9. The uncomfortable but accurate summary
An LLM:
- Does not know facts
- Does not understand meaning
- Does not store labeled relationships

Yet:
Gradient descent on next-token prediction in structured data produces something that behaves like understanding.
That's the whole trick.

Active parameters are the model parameters that are **actually used in computation for a given input (token)**.
That's it in one line-but the idea is super important, so let's unpack it carefully.

What does "active" mean?

A parameter is **active** if it participates in the forward pass when the model processes a token.
If a parameter is not involved in the computation for that token, it is **inactive** for that step.

Dense vs Sparse: why this term exists

Dense LLMs
- **All parameters are active**
- Example:
  - Model size: 7B parameters
  - Active parameters per token: **7B**

There is no distinction between *total* and *active*.

Sparse / MoE (Mixture of Experts) / Hybrid LLMs
- Only **some parameters are active**
- Example:
  - Total parameters: 100B
  - Experts: 16 × 6B
  - Experts used per token: 2
  - Active expert parameters: 12B
  - Plus shared Dense layers

So:
- **Total parameters ≠ Active parameters**

Why active parameters matter

1. Compute cost
Compute scales with **active parameters**, not total parameters.
- More active parameters → slower and more expensive
- Fewer active parameters → faster inference

That's why MoE models are feasible at huge scales.

2. Memory and deployment
- Total parameters affect **model storage**
- Active parameters affect:
  - GPU memory usage
  - Latency
  - Power consumption

3. Performance claims
When you see claims like:
- "Performs like a 70B model at the cost of a 13B model"

They're talking about:
- **Total parameters ≈ 70B**
- **Active parameters ≈ 13B**

Active parameters in Hybrid LLMs

Hybrid models typically have:
- Always-active Dense layers
- Conditionally-active expert layers

So active parameters per token are:
Dense layers
+ Selected experts
= Active parameters
This number can change slightly depending on routing.

Concrete example

Let's say a Hybrid model has:
- Dense backbone: 10B parameters
- MoE layer: 8 experts × 5B each = 40B
- Experts used per token: 2

Then:
- Active parameters per token:
  - 10B (dense)
    - 10B (2 experts)
  - = **20B active**
- Total parameters:
  - 10B + 40B = **50B total**

Important nuance

Active parameters:
- Are **not permanently assigned**
- Can change from token to token
- Are selected dynamically by the router

This dynamic activation is what enables specialization.

One-sentence definition

Active parameters are the subset of a model's total parameters that participate in computation for a specific input or token.

A **vector space** refers to a mathematical space where words or phrases are represented as vectors (essentially points or arrows in space). In the context of language models, this vector space is used to represent words and their relationships.

- **Words as Vectors**: Each word in the model's vocabulary is converted into a vector (a series of numbers). The vector represents the word in a high-dimensional space, capturing the meanings and relationships between words based on their usage in large amounts of text.
- **Dimensionality**: The vector space is typically high-dimensional, meaning each word might be represented by a vector with hundreds or thousands of numbers. These dimensions correspond to different features of the word, such as its meaning, grammatical role, and even contextual relationships to other words.

Example of Vector Space in Action:

Let's say we have two words: "king" and "queen." In the vector space:

- The word "king" might be represented by a vector like:
  [
  [0.1, -0.2, 0.3, 0.5, ...]
  ]
- The word "queen" might be represented by a vector like:
  [
  [0.1, -0.2, 0.3, 0.5, ...]
  ]

While the vectors look similar, they're not exactly the same because "king" and "queen" have subtle differences in meaning and usage. The vectors will be close in the vector space because the two words share a similar relationship (they're both royalty, they both belong to similar contexts, etc.).

- **Word Relationships**: The vector space helps the model understand relationships between words. For example, the vector for "king" and "queen" might be very close, but the vector for "king" and "dog" will be farther apart, since "king" and "dog" don't share the same meaning or context in most texts.

In LLMs like GPT, when the model processes language, it transforms words into vectors, processes these vectors through the network, and generates a response by manipulating these vectors, guided by the learned weights.

How Weights and Vector Space Work Together

The **weights** in the model adjust how much influence each word's vector has on the final prediction. The model uses the vector space to represent words and their relationships and then uses the weights to decide how much each word's vector should affect the next word prediction or decision.

So, **weights** modify how information flows through the network, and **vector space** provides the representation of words that are processed through that network.

Quick Recap:

- **Weights** are the parameters the model learns during training, affecting how it makes predictions.
- **Vector space** is where words are represented numerically as vectors, capturing their meanings and relationships.

Vector databases (as per IBM; excerpt from the webpage at https://www.ibm.com/think/topics/vector-database, retrieved on 2026.07.17)

A vector database stores, manages and indexes high-dimensional vector data.

In a vector database, data points are stored as arrays of numbers called “vectors,” which can be compared and clustered based on similarity. This design enables low-latency queries, making it ideal for artificial intelligence (AI) applications.

Vector databases are growing in popularity because they deliver the speed and performance needed to drive generative AI use cases. In fact, according to 2025 research, vector database adoption grew 377% year over year—the fastest growth reported across any large language model (LLM)-related technology.
What is a Vector Database? Powering Semantic Search & AI Applications (9:48 min)
Vector databases versus traditional databases

The nature of data has shifted dramatically in recent years. It is no longer confined to structured information stored neatly in the rows and columns of traditional databases. Unstructured data—including social media posts, images, videos and audio—is growing in both volume and value, reshaping enterprise AI strategies while putting new demands on data infrastructure.

Traditional relational databases excel at managing structured and semi-structured datasets within defined schemas. However, loading and preparing unstructured data in a relational database for AI workloads is labor-intensive.

Traditional search compounds this limitation: it relies on discrete tokens such as keywords, tags or metadata and returns results based on exact matches. A search for “smartphone,” for example, retrieves only content containing that specific term.

Vector databases take a fundamentally different approach. Instead of rows and columns, data points are represented as dense vectors where each dimension represents a learned characteristic of the data. These high-dimensional vector embeddings exist in vector space, where relationships between items can be measured geometrically.

Because each dimension represents a latent feature—an inferred characteristic learned through mathematical models and algorithms—vector representations capture hidden patterns. A vector search query for “smartphone” can also return semantically related results such as “cellphone” or “mobile device,” even if those exact words do not appear.

By modeling data in high-dimensional space and applying specialized indexing techniques, vector databases make it possible to perform low-latency similarity search across large datasets—something relational databases were not designed to support.
The latest tech news, backed by expert insights

The rapid rise of LLMs, generative AI systems and advanced natural language processing (NLP) workflows has changed how organizations handle and store data. Today’s AI workloads depend on fast, real-time interaction with vector data as well as seamless integration with retrieval-augmented generation (RAG) pipelines.

Vector databases provide the infrastructure to support these demands. They enable low-latency similarity search across large volumes of unstructured data, powering AI applications such as chatbots and recommendation systems.
Core concepts within vector databases

To understand how vector databases operate, it helps to establish two core concepts: vectors, which describe data in numerical form and vector embeddings, which translate unstructured content into high-dimensional representations that capture meaning and context.

Vectors

Vectors are a subset of tensors. In machine learning (ML), tensor is a generic term for a group of numbers—or a grouping of groups of numbers—in n-dimensional space. Tensors function as a mathematical bookkeeping device for data. Working up from the smallest element:

A scalar is a zero-dimensional tensor, containing a single number. For example, a system modeling weather data might represent a single day's high temperature (in Fahrenheit) in scalar form as 85.

A vector is a one-dimensional (or first-degree or first-order) tensor, containing multiple scalars of the same type of data. Building on our example, a weather model might use the low, mean and high temperatures for a single day in vector form: 62, 77, 85. Each scalar component is a feature—that is, a dimension—of the vector, representing a feature of that day's weather.

In other words, vectors are a way of organizing numbers into a structured form. But for AI systems to process that unstructured information, the data must be translated into numerical arrays. This translation is achieved through vector embeddings.
Vector embeddings

Vector embeddings are numerical representations of data points that convert various types of data—including text and images—into arrays of numbers that ML models can process.

To achieve this, embedding models learn how to map input data into a high-dimensional vector space. A task-specific loss function shapes the vector space by quantifying prediction errors. Vector embeddings can then be used by downstream AI models, like neural networks used in deep learning, to perform tasks like classification, retrieval or clustering.

Consider a small corpus of words, where the word embeddings are represented as 3-dimensional vectors:

cat [0.2, -0.4, 0.7]
dog [0.6, 0.1, 0.5]

In this example, each word ("cat") is associated with a unique vector ([0.2, -0.4, 0.7]). The values in the vector represent the word's position in a 3-dimensional vector space. Words with similar meanings or contexts are expected to have similar vector representations. The vectors for "cat" and "dog" would be close together, reflecting their semantic relationship.

Similarly, the words "car" and "vehicle" share the same meaning but are spelled differently. For an AI application to perform semantic search, the vector representations of "car" and "vehicle" must capture their shared meaning. Vector embeddings encode this meaning numerically, making them the backbone of recommendation engines, chatbots and generative applications like OpenAI's ChatGPT.

Understand how AI-ready data platforms enable real-time insights and execution, while supporting secure, sovereign deployment across environments.
Explore watsonx.data
How do vector databases work?

To facilitate fast and scalable semantic retrieval, vector databases rely on three core functions:

Vector storage
Vector indexing
Vector search

Vector storage

At a foundational level, vector databases store embeddings. Each has a fixed number of dimensions and is typically stored alongside metadata such as title, source, timestamp or category, which can be queried through metadata filters.

Because embeddings are generated in advance and stored, vector databases can retrieve similar vector embeddings without recomputing representations at query time. This separation of generation and retrieval supports low-latency similarity search at scale.

Many systems also support hybrid search that combines vector similarity with metadata constraints—for instance, retrieving semantically similar documents created within a specific date range or category.
Vector indexing

To accelerate similarity search in high-dimensional space, vector databases create indexes on stored vector embeddings. Indexing maps the vectors to new data structures, enabling faster similarity or distance searches between vectors.

These indexes support approximate nearest-neighbor (ANN) search, which retrieves similar vectors without scanning the entire dataset. Common ANN indexing algorithms include hierarchical navigable small world (HNSW) and locality-sensitive hashing (LSH):

HNSW creates a hierarchical, multi-layer graph that uses long-range links in the upper layers and dense local links in the bottom layer.1

LSH groups vectors into buckets through a hash function so that similar vectors fall into the same bucket.

In addition to ANN indexes, vector databases often use product quantization (PQ) to reduce memory usage. PQ converts each dataset into a short code that preserves relative distance (rather than storing every vector), allowing systems to store larger collections while maintaining efficient search performance.
Vector search

Vector search is the retrieval layer of a vector database used to discover and compare similar data points. Rather than matching exact keywords or values, it captures the semantic relationships between elements. This context-aware retrieval capability underpins RAG systems, which in turn supply relevant context to AI systems and retrieval-based machine learning models.

When a user prompts an AI model, the model generates an embedding of that query, known as a query vector. The database then compares the query vector against indexed vectors and calculates similarity scores to identify the nearest neighbors.

Vector search applies multiple algorithms to conduct an ANN search. These algorithms are gathered in a pipeline to quickly and accurately retrieve data neighboring the vector that is queried (for example, products that are visually similar in an e-commerce catalog). Because embeddings are precomputed and stored in indexed form, results are returned within milliseconds.

Once the relevant vectors are identified, they're compared either by calculating their similarity or with a distance metric. Common methods include:

Cosine similarity: Measures the angular distance between vectors to determine how aligned they are in each direction.

Jaccard similarity: Compares the overlap between two sets relative to their total elements.

Dot product: Evaluates similarity based on the magnitude and direction of vectors.

Euclidean distance: Calculates the straight-line distance between vectors in high-dimensional space.

Hamming distance: Counts differing positions between binary vectors.

The database returns the highest-ranking vectors according to these similarity calculations, supporting machine learning tasks such as semantic search and other natural language processing workflows.

What are the benefits of vector databases?

Vector databases are increasingly central to enterprise AI strategies because they deliver a range of benefits:

Speed and performance: Vector databases use various indexing techniques to enable faster searching. Vector indexing and distance-calculating algorithms can help optimize performance when searching for relevant results across datasets with millions and even billions of data points.

Scalability: Vector databases can store and manage massive amounts of unstructured data by scaling horizontally with other nodes, maintaining performance as query demands and data volumes increase.

Lower cost of ownership: Because they enable faster data retrieval, vector databases speed the training of foundation models.

Data management: Vector databases typically provide built-in data management features to easily update and insert new unstructured data.

Flexibility: Vector databases are built to handle the added complexity of using images, videos or other multidimensional data.

Vector database use cases

Vector databases can be customized to meet specific business and AI use cases. Often, organizations start with a general-purpose embedding model such as IBM® Granite™, Meta's Llama-2 or Google's Flan. Models are then enhanced with enterprise data stored in a vector database. This combination improves the relevance and accuracy of downstream AI applications.

The applications for vector databases are vast and expanding. Key use cases include:

Retrieval-augmented generation
Conversational AI
Recommendation engines
Anomaly detection

Retrieval-augmented generation

RAG enables LLMs to retrieve facts from an external knowledge base. Enterprises increasingly favor RAG for its faster time-to-market, efficient inference and reliable output, particularly in areas such as customer care, HR and talent management.

By grounding the model in trusted enterprise data, RAG reduces hallucinations and gives users access to the underlying sources for verification. Because the inference stage performs the highest-volume retrieval operations, it requires fast, precise and scalable access to high-dimensional vector embeddings.

Vector databases excel at indexing, storing and retrieving these embeddings, providing the speed, precision and scale needed for applications such as fraud detection systems and predictive maintenance platforms.

Conversational AI

Vector databases, particularly when used to implement RAG frameworks, can help improve virtual agent interactions by enhancing the agent's ability to parse relevant knowledge bases efficiently and accurately. Agents can provide real-time contextual answers to user queries, along with the source documents and page numbers for reference.

Recommendation engines

E-commerce sites can use vectors to represent customer preferences and product attributes. This approach allows them to improve customer experience and retention by suggesting items similar to past purchases.

Streaming platforms and social media applications apply the same approach, recommending videos, music or posts based on similarity to content a user has previously viewed or shared.

Anomaly detection

By representing normal behavior as vectors in high-dimensional space, organizations can detect outliers based on vector distance. Data points that fall far from established clusters can signal fraud, system faults or unusual activity patterns. Because similarity is calculated mathematically, anomalies can be detected in real time across massive datasets—from network traffic to sensor readings in industrial systems. This capability allows teams to intervene before small deviations escalate into costly incidents.

While vector databases are well suited for fact-based retrieval across many AI applications, they are not ideal for every type of query.

Workloads such as topic summarization or broad thematic analysis require an LLM to read through all relevant context rather than rely solely on nearest-neighbor matches. In these scenarios, a list index or another non-vector structure can provide faster, more efficient results, because they can quickly surface the first relevant elements without navigating vector space.

Pruning is the process of removing parts of a neural network that contribute relatively little to its performance. The goal is to make the model smaller, faster, and more efficient while preserving as much accuracy as possible.

Why prune an LLM?
Modern LLMs can have billions of parameters. Not all of these parameters are equally important for every task. Pruning aims to:

- Reduce memory usage
- Speed up inference
- Lower computational costs
- Make deployment easier on smaller hardware (e.g., edge devices)

How pruning works
A trained model contains many weights (parameters). Pruning identifies weights or larger structures that appear to have little impact on predictions and removes them.

For example:
Original weights:
[0.92, -0.01, 0.45, 0.0002, -0.78]

After pruning small weights:
[0.92, 0, 0.45, 0, -0.78]

Here, weights close to zero are set to zero because they have minimal influence.

Types of pruning

1. Unstructured pruning

Removes individual weights.

Before:
[0.8 0.2 0.7]
[0.1 0.9 0.3]

After:
[0.8 0 0.7]
[0 0.9 0]

Pros:
- Can remove many parameters.
- Often preserves accuracy well.

Cons:
- Irregular sparsity means GPUs may not run much faster unless specialized sparse kernels are used.

2. Structured pruning

Removes entire components such as:

- Neurons
- attention heads
- channels
- layers

For example, instead of removing scattered weights, you might remove an entire attention head.

Pros:
- Produces real speedups on standard hardware.
- Easier to deploy.

Cons:
- More aggressive and can hurt accuracy if important structures are removed.

3. Magnitude pruning

The simplest and most common method.

Idea:
Small weights matter less than large weights.

Algorithm:
- Compute the absolute value of every weight.
- Remove the smallest ones.
- Optionally fine-tune the model afterward.

4. Importance-based pruning

Instead of looking only at weight size, estimate how much each parameter affects the model's output.

Importance can be estimated using:
- Gradients
- Hessian approximations
- activation statistics
- sensitivity analysis

These methods tend to preserve accuracy better but require more computation.

Pruning during or after training

This is the most common workflow.

Post-training pruning
- Train the full model.
- Remove parameters.
- Fine-tune the remaining model.

Training-time pruning
- Gradually remove parameters during training.
- The network adapts as it learns.

Sparsity

Pruning creates *sparse* models.

Example:

Dense:


50% sparse:


90% sparse:


A model with 90% sparsity has only 10% of its weights remaining.

Does pruning always make models faster?

Not necessarily.
If you prune individual weights (unstructured pruning), the model may have fewer nonzero parameters but still perform similar amounts of computation on conventional hardware. Real speedups often require hardware or software optimized for sparse operations.

Structured pruning, on the other hand, changes the model's architecture (for example, by removing entire attention heads or layers), so it more often leads to practical improvements in latency and memory use on standard CPUs and GPUs.

How much can you prune?
It depends on the model and the pruning strategy.

Typical outcomes are:
20–50% pruning: often little or no noticeable loss in quality.
50–80% pruning: quality may begin to degrade unless pruning is done carefully and followed by fine-tuning.
Above 90% pruning: usually causes significant performance loss, though some specialized techniques can achieve high sparsity while maintaining good accuracy on certain tasks.

Relationship to quantization

Pruning and quantization solve different problems:

Pruning	*Quantization*
Removes parameters	Reduces numerical precision
Creates sparse models	Creates lower-precision models
Can reduce computation	Reduces memory and arithmetic cost
May require fine-tuning	May or may not require fine-tuning

They are often combined. For example, a model might first be pruned to remove less important parameters and then quantized from 16-bit to 8-bit or 4-bit precision to further reduce memory and improve efficiency.

Example workflow

A common optimization pipeline looks like this:
- Train a dense model with 7 billion parameters.
- Prune 30% of the least important weights.
- Fine-tune the remaining model to recover performance.
- Quantize the model (e.g., from FP16 to INT8 or 4-bit).
- Deploy the optimized model for inference.

This can substantially reduce memory usage and computational cost while maintaining performance close to that of the original model, depending on the pruning strategy and the target task.

1.4.3. Size categories of LLMs

This is more of an industry convention than a strict scientific standard.

LLMs are commonly categorized by their **parameter count** (the “weights”), and the number is usually expressed in **billions (B)** or **millions (M)** of parameters. Here’s how that categorization is typically understood.

What does “B” mean?

- **1B = 1 billion parameters (weights)**
- So:
  - 0.5B = 500 million parameters
  - 3B = 3 billion parameters
  - 70B = 70 billion parameters

Each parameter is a learned numerical value that helps the model make predictions.

Common size categories of LLMs (informal but widely used)

There is **no official global standard**, but the following ranges are widely accepted in research papers, open-source projects, and industry discussions.

1. Small Language Models (SLMs)
~10M - 500M parameters

- Examples: 125M, 350M, 500M
- Characteristics:
  - Fast and cheap to run
  - Can work on edge devices or small servers
  - Limited reasoning and context handling
- Typical uses:
  - Autocomplete
  - Simple chatbots
  - Narrow or domain-specific tasks

2. Medium Language Models
~0.5B - 3B parameters

- Examples: 0.5B, 1B, 1.3B, 2.7B, 3B
- Characteristics:
  - Noticeably better language fluency
  - Can handle short reasoning chains
  - Still relatively efficient
- Typical uses:
  - Educational tools
  - Lightweight assistants
  - Fine-tuned domain models (medicine, law, coding basics)

This is a range very popular in year 2025/2026.

3. Large Language Models
~7B - 30B parameters

- Examples: 7B, 13B, 30B
- Characteristics:
  - Strong general language understanding
  - Good reasoning and instruction-following
  - Requires GPUs to run efficiently
- Typical uses:
  - General-purpose assistants
  - Coding help
  - Research and analysis tasks

Many open-source “ChatGPT-like” models fall here.

4. Very Large / Frontier Models
~65B - 200B+ parameters

- Examples: 65B, 70B, 175B
- Characteristics:
  - Advanced reasoning and world knowledge
  - Better few-shot learning
  - Extremely expensive to train and run
- Typical uses:
  - State-of-the-art chat systems
  - Scientific research
  - Enterprise-scale applications

5. Massive / Ultra-Scale Models
Hundreds of billions to trillions of parameters

- Examples: 500B, 1T+
- Characteristics:
  - Cutting-edge research models
  - Often use mixtures of experts (MoE)
  - Not usually publicly deployable in full form
- Mostly experimental or proprietary

Important nuance: size ≠ quality (always)

While parameter count *matters*, performance also depends heavily on:

- Training data quality
- Number of training tokens
- Architecture (Dense vs MoE)
- Fine-tuning and alignment methods

A **7B well-trained model** can outperform a **30B poorly trained one**.

Quick summary table

Category	Parameter Range
Small	10M - 500M
Medium	0.5B - 3B
Large	7B - 30B
Very Large	65B - 200B+
Ultra-Scale	500B - 1T+

1.4.4. Dense LLMs

A **Dense LLM** is the "classic" and still very common type of Large Language Model, and it's best understood *in contrast* to Mixture of Experts.

What is a Dense LLM?

A **Dense Large Language Model** is a model where:
Every parameter (weight) in the model is used for every token during inference.
Nothing is selectively turned on or off.
If the model has 7 billion parameters, **all 7 billion participate** every time it processes a token.
This is why it's called *dense*: the computation graph is fully active.

Core characteristics of Dense LLMs

1. Full activation

- All layers are always used
- No routing, no expert selection
- Each token flows through the same network path

2. Uniform structure

A Dense LLM typically consists of:

- Token embedding layer
- Repeated blocks of:
  - Self-attention
  - Feed-forward network (FFN)
- Output layer

Every block is identical in structure and always active.

3. Predictable behavior

Because every token follows the same path:

- Performance is stable
- Debugging is simpler
- Hardware utilization is more straightforward

This simplicity is a major reason Dense models are still dominant.

Dense LLMs vs MoE (quick contrast)

Feature	Dense LLM	MoE LLM
Parameters used per token	100%	Small subset
Routing	None	Required
Complexity	Simpler	More complex
Compute cost	High	Lower for same total params
Predictability	High	Lower
Engineering difficulty	Lower	Higher

Why most early LLMs were Dense

Dense models were favored because:

- They are easier to train
- They scale smoothly
- They work well with existing hardware
- They were well-understood in deep learning

Famous examples include:

- GPT-2
- GPT-3 (175B, fully dense)
- BERT
- LLaMA (1-65B variants)

Limitations of Dense LLMs

1. Compute scales linearly with size
If you double the parameters:

- You roughly double:
  - Training cost
  - Inference cost
  - Memory usage

This makes very large Dense models extremely expensive.

2. No explicit specialization
Dense models:

- Learn everything in the same parameter space
- Can specialize implicitly, but not as cleanly as MoE models

Why Dense models are still widely used

Despite their cost, Dense LLMs remain popular because:

- They are robust and reliable
- They're easier to fine-tune
- They work well on a wide range of tasks
- They're simpler to deploy (especially open-source)

For many applications, a **well-trained 7B or 13B Dense model** is the sweet spot.

One-sentence definition

A Dense LLM is a language model in which all parameters are active for every token, resulting in simple, predictable behavior at the cost of higher compute requirements.

1.4.5. Mixture of Experts LLMs

Mixture of Experts (MoE) is one of the most important ideas behind scaling modern LLMs *without* making them impossibly expensive.

A **Mixture of Experts (MoE)** model is a neural network architecture where:
Instead of using the entire model for every input, the model dynamically selects a small subset of specialized sub-models ("experts") to process each token.
So rather than *one giant brain always working*, you get:

- Many **experts** (smaller neural networks)
- A **router (or gating network)** that decides which experts to use

Only a few experts are active at any given time.

Intuitive analogy

Imagine a hospital:

- You don't send every patient to *every* doctor
- A triage nurse decides whether you see:
  - a cardiologist
  - a neurologist
  - a general practitioner

That triage nurse is the **router**, and the doctors are the **experts**.

How MoE works (simplified)

1. **Input tokens enter the model**
2. A **router network** examines each token
3. The router selects the top *k* experts (often k = 1 or 2)
4. Only those experts process the token
5. Their outputs are combined and passed to the next layer

Crucially:

- The *experts exist*, but
- **Most of them are idle for any given token**

What exactly are the "experts"?

- Each expert is usually a **feed-forward neural network**
- All experts share the same architecture
- They differ only in their learned weights
- Experts live inside specific layers (often replacing the FFN layer)

So MoE typically looks like:

- Attention layer → MoE layer → Attention layer → MoE layer → …

Why use Mixture of Experts?

The core purpose:
Scale model capacity without scaling compute cost proportionally
Let's unpack that.

1. More parameters, same compute
A Dense model:

- Uses **100% of its parameters** every time

An MoE model:

- Might have **100B parameters total**
- But only **10B active per token**

So you get:
- Huge knowledge capacity
- Much lower inference cost

This is called **sparse activation**.

2. Better specialization
Experts naturally specialize:
- One expert might handle math-like patterns
- Another might focus on conversational language
- Another on code-like syntax

This emerges *automatically* during training.

3. More efficient scaling than Dense models
With Dense models:
- Doubling parameters ≈ doubling compute

With MoE:
- You can double parameters while increasing compute only slightly

This made it possible to push beyond the ~100B parameter barrier.

Why MoE matters for modern LLMs

MoE is one of the key techniques behind:
- Frontier-scale models
- Cost-efficient large assistants
- Models that feel “smarter” without being impossibly slow

It’s especially useful for:
- Large vocabularies
- Multi-domain tasks
- Multilingual models

Trade-offs and challenges

MoE is powerful, but not free.
1. Training complexity
- Routing decisions must stay balanced
- Otherwise some experts get overloaded
- Requires **load-balancing losses**

2. Hardware and engineering difficulty
- Experts may live on different GPUs
- Requires fast communication
- Harder to deploy than Dense models

3. Less predictable behavior
- Because different experts activate dynamically
- Debugging becomes harder

Dense vs MoE: quick comparison

Feature	Dense Model	MoE Model
Parameters used per token	All	Small subset
Total parameter count	Limited by compute	Can be huge
Inference cost	High	Lower
Engineering complexity	Simpler	More complex
Specialization	Implicit	Emergent and stronger

Examples of MoE models (conceptually)

- Switch Transformers
- GLaM
- Mixtral (open-source)
- Many proprietary frontier models

(Some models are described as "8×7B" or similar-that's classic MoE notation.)

Why you'll often see "8×7B"

That means:

- 8 experts
- Each expert ≈ 7B parameters
- Total parameters ≈ 56B
- But only 1-2 experts active per token

So runtime cost feels closer to a 7-14B model.

One-sentence summary

Mixture of Experts allows LLMs to be very large in knowledge and capability while staying efficient by activating only the most relevant parts of the model for each input.

1.4.6. Hybrid LLMs

A **Hybrid LLM** is a language model that deliberately **combines Dense and Sparse (usually MoE) components** within the same architecture, aiming to get the *best of both worlds*.

A Hybrid LLM mixes Dense layers (always active) with Sparse layers (conditionally active, such as Mixture of Experts).

In other words:

- Some parts of the model are **fully dense**
- Some parts activate **only selected parameters per token**

This design balances **stability and efficiency**.

Why Hybrid models exist

Pure approaches have trade-offs:

- **Fully Dense models**
  - Simple and predictable
  - Very expensive to scale
- **Pure MoE models**
  - Efficient at scale
  - Complex and sometimes unstable

Hybrid models sit in the middle:
Scale capacity like MoE, but behave like Dense models where it matters most.

Typical structure of a Hybrid LLM

A common Hybrid pattern looks like this:

- **Dense embedding layers**
- **Dense attention layers** (almost always dense)
- **Sparse MoE feed-forward layers** (only in some blocks)
- **Dense output layers**

For example:

- Attention = dense
- FFN = MoE (in selected layers)

So a block might look like:

[Dense Attention] → [MoE FFN] → [Dense Attention] → [Dense FFN]

Why attention layers are usually Dense

Self-attention:

- Integrates information across the entire context
- Needs global consistency
- Is sensitive to routing noise

Making attention sparse often hurts coherence, so most Hybrid models:

- Keep attention dense
- Apply sparsity only to FFN layers

Key purposes of Hybrid LLMs

1. Efficient scaling
Hybrid models can:

- Have **huge total parameter counts**
- Use **only a fraction per token**
- Stay affordable at inference time

2. Stability and predictability
Dense components:

- Anchor the model's behavior
- Reduce routing-related failures
- Improve fine-tuning reliability

3. Emergent specialization (without chaos)
Sparse expert layers:

- Learn domain-specific patterns
- While Dense layers maintain shared understanding

Example parameter accounting

A Hybrid model might be described as:

- **Total parameters**: 100B
- **Active per token**: ~15B

That's far cheaper than a fully Dense 100B model, but more stable than a fully Sparse one.

Common Hybrid patterns

1. Interleaved MoE layers

- Every Nth layer is MoE
- Others are dense

2. Partial MoE

- Only FFN layers are MoE
- Attention remains dense

3. Shallow MoE

- MoE only in deeper layers
- Early layers remain fully dense

Hybrid vs Dense vs MoE (summary)

Feature	Dense	MoE	Hybrid
Parameters per token	All	Few	Some
Total capacity	Limited	Very large	Very large
Stability	High	Lower	High
Efficiency	Low	High	High
Engineering complexity	Low	High	Medium

Why Hybrid models are becoming popular

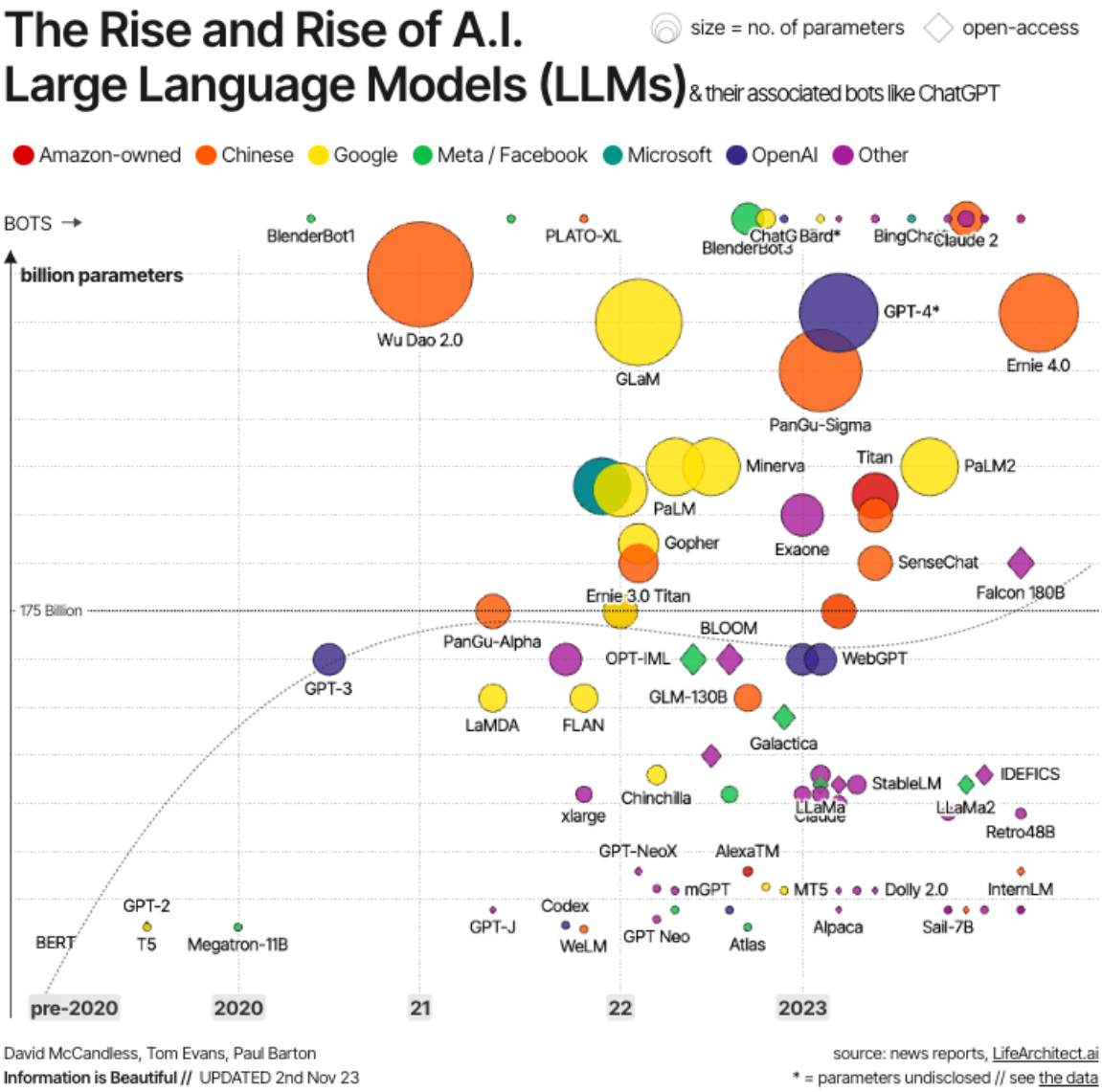
They:

- Scale better than Dense models
- Are easier to deploy than pure MoE
- Handle diverse tasks well
- Work better in production systems

Many modern "frontier" LLMs use Hybrid designs, even if they don't advertise it explicitly.

One-sentence definition

A Hybrid LLM combines Dense layers with Sparse expert-based layers to balance scalability, efficiency, and stable behavior.



1.4.7. **Quantization**

Quantization in the context of **Large Language Models (LLMs)** refers to the process of reducing the precision of the numbers (typically floating-point values) used to represent the model's parameters (weights) and activations. This is done to reduce the model's memory footprint and computational load while still retaining a good level of performance.

Why is Quantization Important?
LLMs, especially large ones like GPT, BERT, etc., have billions of parameters, and the precision with which these parameters are stored and processed can greatly impact the efficiency of training and inference. Quantization helps to:

- **Reduce memory usage**: By using lower precision (e.g., 8-bit integers instead of 32-bit floats), less memory is required to store the model's weights.
- **Speed up inference**: Lower precision operations can be faster, especially when using specialized hardware (like GPUs or TPUs).
- **Energy efficiency**: It reduces the computational cost, leading to less power consumption.

How does it work?
Quantization works by mapping the high-precision floating-point numbers (e.g., 32-bit floats) used in the model to lower-precision representations, typically 8-bit integers or even lower.

Example 1: Floating Point to Integer Quantization

Imagine we have a weight value in the model represented as a 32-bit floating-point number:

- Original weight: 3.14159

Quantization might map this number to an 8-bit integer (let's say a range from 0 to 255):

- Quantized weight: 203 (in 8-bit integer format)

The value 203 is a compressed representation of the original number 3.14159. Of course, this approximation introduces some error, but the goal is to minimize the loss in performance while gaining efficiency.

Example 2: Reducing Precision in Activation Values
In a neural network, activation values are computed during both training and inference. Typically, these are floating-point values. Quantization can reduce the precision of these activations as well.
For instance:

- An activation value might be 0.85 in float32 format.
- After quantization to int8 format, this might become 220 (within a specified range).

The neural network can then perform operations using these quantized values instead of the original, high-precision floating-point numbers, saving both memory and computation time.

Trade-offs with Quantization

- **Accuracy Loss**: The main downside of quantization is the potential loss of model accuracy. Since you're approximating the model parameters with fewer bits, this can introduce errors that might affect the model's performance, especially if the quantization is too aggressive.
- **Tuning Required**: The process often involves fine-tuning the quantized model to ensure the drop in accuracy is minimal.

Types of Quantization:

1. **Post-Training Quantization**: This method is applied after the model has already been trained. It involves converting the trained model's weights and activations to lower precision.
2. **Quantization-Aware Training (QAT)**: In this approach, the model is trained with quantization in mind. This means that during training, the model learns how to function with quantized weights and activations, which can reduce accuracy loss compared to post-training quantization.

Simple Analogy:
Think of quantization like resizing an image. Imagine you have a high-resolution image (32-bit precision), and you want to shrink it to a smaller size (8-bit precision). The smaller image still looks like the original but with less detail. Similarly, quantization reduces the "detail" (precision) in the model weights but keeps the overall structure intact.

Real-world Usage:
Quantization is widely used in deployment for mobile devices, edge computing, and situations where computational resources (like memory or processing power) are limited but the model still needs to perform well. It's also crucial for making real-time language processing feasible in environments like virtual assistants or chatbots.

Let's go through a simplified example of **quantization**. I'll explain the concept in a straightforward way.

Example: A Simple Neural Network

Imagine you have a small neural network with 4 weights, and each weight is a number that helps the network make decisions. Right now, these weights are **32-bit floating-point numbers**, which are standard for high precision. Here are the original weights:

- **Weight 1**: 0.8712
- **Weight 2**: -3.527
- **Weight 3**: 1.214
- **Weight 4**: -0.681

Step 1: What Does Quantization Do?
Quantization is about **reducing the precision** of these numbers to save space and make the model faster. In this case, we want to **convert** these weights from 32-bit floating-point numbers to **8-bit integers**. Think of it like rounding numbers off to make them simpler.
For example, an 8-bit integer can represent numbers from 0 to 255. We'll use this range to represent the original floating-point numbers as 8-bit integers.

Step 2: How Does Quantization Work?
Instead of storing each of the original 32-bit weights (which take up a lot of memory), we'll map them to simpler, smaller numbers in the 0-255 range. The network still "understands" these numbers, but they take up less memory.

Here's what happens to the weights after quantization:

- **Weight 1 (0.8712)**: It gets mapped to 237
- **Weight 2 (-3.527)**: It gets mapped to 0
- **Weight 3 (1.214)**: It gets mapped to 255
- **Weight 4 (-0.681)**: It gets mapped to 153

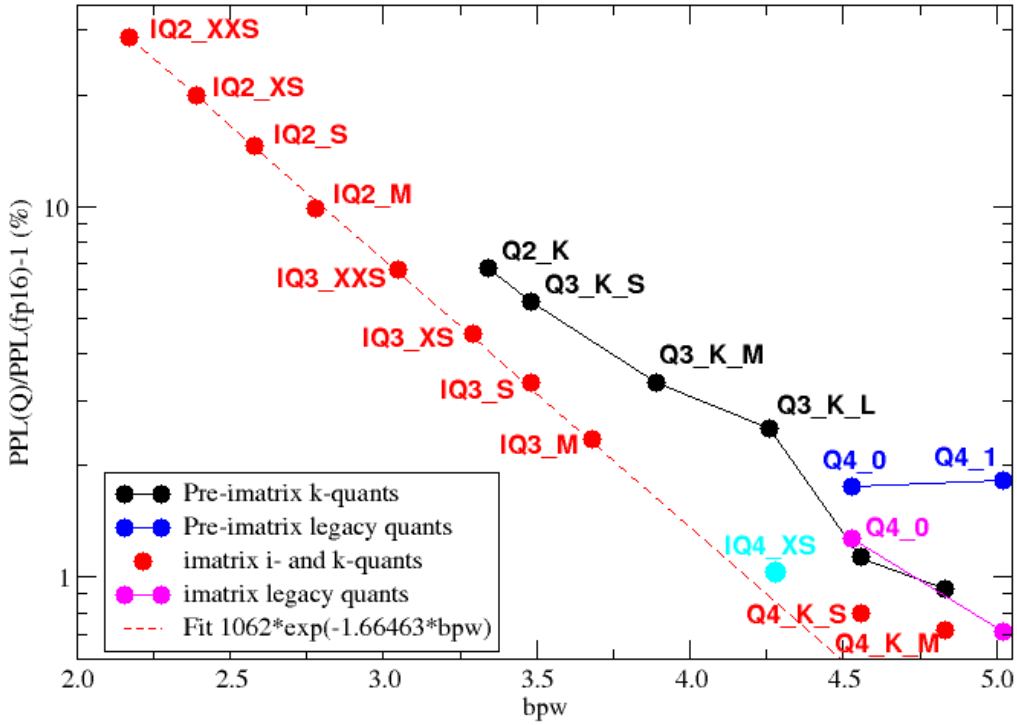
Step 3: Result of Quantization
After quantization, the weights are much smaller and are now **integers** between 0 and 255, instead of being complex floating-point numbers. The new weights are:

- **Weight 1**: 237
- **Weight 2**: 0
- **Weight 3**: 255
- **Weight 4**: 153

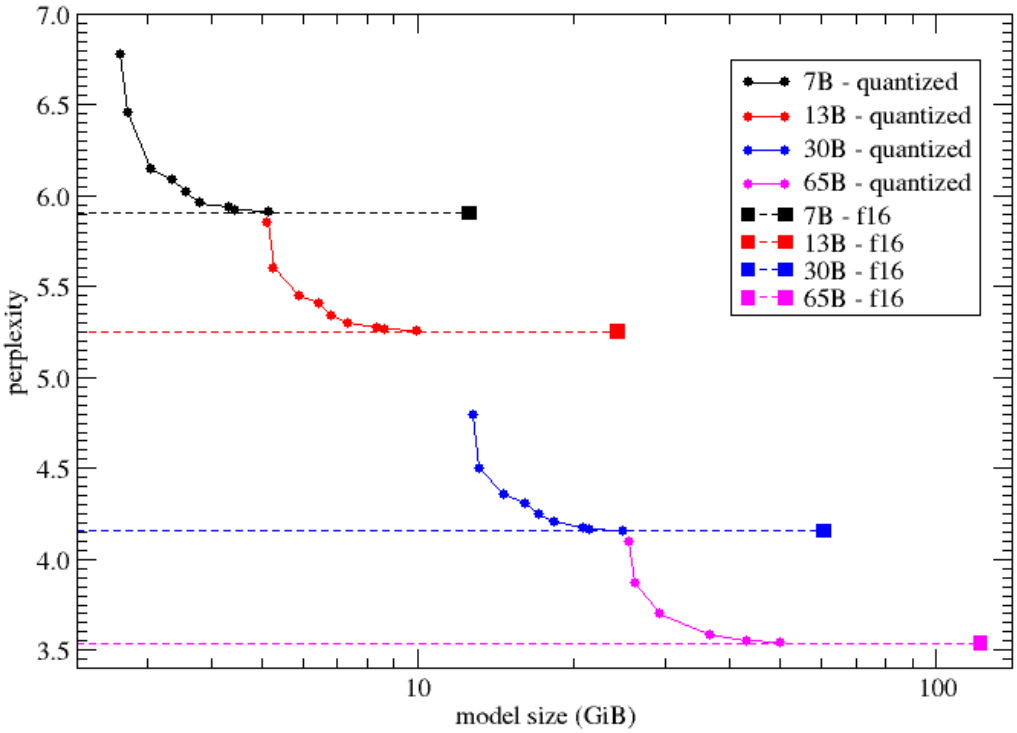
Step 4: Why Do This?
By converting the weights to these simpler numbers, we save a lot of **memory** and make calculations **faster**. This is especially helpful when deploying models on devices with limited resources, like smartphones or IoT devices.

Trade-off: Precision Loss
The downside is that we lose some **precision**. For example, instead of having 0.8712 for Weight 1, we just have 237. The model may not be as accurate as before, but the trade-off is that it's **much faster** and **requires less memory**.

Summary
Quantization is like turning long, detailed numbers into shorter, simpler ones to save space and speed up calculations. It's a great technique for deploying large models on devices where memory and processing power are limited. But, it comes with a small loss in precision.



perplexity vs bits-per-weight (image from: https://rentry.org/llama-cpp-quants-or-fine-ill-do-it-myself-then-pt-2)



perplexity vs model size (image from: https://github.com/ggml-org/llama.cpp/pull/1684)

As of 2026:

Recent advances in ultra-low-bit LLM quantization (especially Q2) focus on making 2-bit models actually usable without destroying reasoning quality.
Main trends:

- **Better quantization algorithms**
  New methods like QuIP# and SpinQuant handle "outlier" weights much better than older GPTQ-style rounding.
- **FP4 instead of INT4**
  The field is moving toward low-bit floating point formats (FP4/MXFP4) because they preserve model quality better than pure integer quantization.
- **Mixed precision**
  Modern Q2 systems keep sensitive layers at higher precision while aggressively compressing the rest.
- **Native low-bit training**
  Some newer models are trained directly for low precision instead of being compressed afterward.
- **Hardware-aware designs**
  Quantization is now optimized jointly with GPU kernels, memory bandwidth, and inference engines for real speed gains.

Overall:
Q2 used to be mostly impractical; now advanced methods can get surprisingly close to Q4 quality while using much less VRAM and bandwidth.

1.5. Types of Datasets Used for Training an LLM

LLMs are typically trained on **massive datasets** consisting of diverse text sources. These datasets are usually collected from publicly available online sources to ensure that the model can understand a wide variety of topics and language forms. The types of datasets used for training include:

- **Books**: A large portion of training data comes from digitized books, which provide rich, structured text with complex sentence structures and varied vocabulary.
- **Websites**: Text from websites, including forums, articles, and blogs, offers a diverse array of language styles and topics.
- **News Articles**: A variety of news sources can be used to train models, helping them understand current events and formal writing styles.
- **Wikipedia**: Open-source encyclopedic content like Wikipedia provides a large, structured dataset with factual information on a variety of topics.
- **Social Media and Conversational Data**: Datasets collected from social media platforms or conversational databases help the model learn informal language and slang, enhancing its ability to understand casual, conversational contexts.

1.6. The Context Length of an LLM

Context length (also called **context window**) is one of the most important parameters in understanding how LLMs process text. It refers to the maximum number of tokens (words, subwords, or characters depending on tokenization) that the model can "see" at once when generating predictions or responses.

Essentially, it determines **how much of the previous text the model can consider** when generating the next word or token. The context length sets a hard limit: if the input text is longer than the context window, older tokens are truncated or discarded, and the model cannot take them into account.

How Context Length Works

1. **Tokenization**: Before being fed into the LLM, text is split into tokens. Each token represents a small piece of text, which could be a word, a part of a word, or even punctuation.
2. **Attention Mechanism**: The model uses its attention layers to process all tokens in the context window simultaneously, calculating relationships between them to understand context, grammar, meaning, and dependencies.
3. **Window Limit**: For example, if an LLM has a context length of **2,048 tokens**, it can only consider the most recent 2,048 tokens of input. If your document is 5,000 tokens long, only the last 2,048 tokens will be "visible" to the model when generating the next token.

Importance of Context Length

- **Coherence**: Longer context lengths allow the model to maintain coherence over longer texts. For instance, a novel-length context helps the model remember characters, plot points, or instructions that appeared far earlier in the text.
- **Instruction Following**: In instruction-based tasks (like "summarize this text"), a longer context allows the model to see more instructions or reference material, improving accuracy.
- **Dialogue Memory**: In conversational agents, context length limits how far back the model can remember the conversation. A short context window may cause the model to "forget" earlier messages.
- **Performance Trade-offs**: Increasing context length requires more computational resources. Attention computation scales roughly quadratically with context length, meaning doubling the context may require roughly four times the memory and computation.

Modern Trends

Recent (as of 2025/2026) LLMs are pushing context lengths much higher:

- Early GPT models had context lengths of 512-1,024 tokens.
- GPT-4 and other large models can handle 8,000 to 32,000 tokens.
- Some experimental architectures (like **long-context Transformers**) aim for hundreds of thousands of tokens.

Increasing context length is critical for applications like:

- Long-form document summarization
- Code generation from large codebases
- Multi-turn dialogue systems
- Research tasks that require long references

So, **the context length defines the "memory window" of the LLM**, limiting how much past text it can consider when generating output. Longer context means more coherent, context-aware, and accurate responses-but at the cost of higher computation.

1.7. Hallucinations

An LLM hallucination is when the model generates information that sounds plausible and confident but is factually incorrect, fabricated, or unsupported.
In short:

- The model is **making things up**
- Without knowing it's wrong

Why are they called "hallucinations"?

The term is metaphorical:

- The model *appears* to see facts that aren't there
- But it's not conscious or aware

Technically, it's just predicting text that *fits patterns*-not verifying truth.

Examples of hallucinations

An LLM might:

- Invent a scientific paper that doesn't exist
- Attribute a quote to the wrong person
- Give a confident but incorrect historical date
- Describe features of a product that was never released
- Cite fake sources or URLs

The dangerous part is:
Hallucinations often sound very convincing.

Why do LLMs hallucinate?

1. LLMs predict text, not truth
LLMs are trained to:

- Predict the **most likely next token**
- Not to check facts

If something *sounds right*, it may be generated-even if it's false.

2. Gaps in training data
If the model:

- Has incomplete knowledge
- Has outdated information
- Never saw a specific fact

It may still try to answer instead of saying "I don't know".

3. Overgeneralization
LLMs learn patterns like:
- "Academic papers usually have authors, dates, journals"

So they may fabricate:
- Titles
- Authors
- Institutions

4. Ambiguous or underspecified prompts
If the question is vague:
- The model fills in missing details
- Sometimes incorrectly

5. Long or complex reasoning chains
Errors can accumulate step by step, leading to a wrong final answer that still sounds logical.

Are hallucinations random?

No.
They are:
- **Systematic**
- Pattern-based
- More likely in certain situations

Hallucinations are especially common when:
- Asked for precise facts
- Asked about obscure topics
- Asked to cite sources
- Asked questions with no clear answer

Hallucinations vs mistakes

Not all errors are hallucinations.
- **Mistake**: A simple factual error
- **Hallucination**: Fabrication of information that doesn't exist

Example:
- Mistake: "The Eiffel Tower is 320 m tall" (close but wrong)
- Hallucination: "The Eiffel Tower was designed in 1902 by Jean Martin" (invented details)

Why hallucinations are a serious problem

They can:
- Mislead users
- Spread misinformation
- Be dangerous in domains like:
  - Medicine
  - Law
  - Engineering
  - Education

This is why LLMs should not be blindly trusted for factual authority.

How hallucinations are reduced (but not eliminated)

1. Better training and alignment

- Higher-quality datasets
- Human feedback
- Penalizing fabricated answers

2. Retrieval-Augmented Generation (RAG)

- Ground answers in real documents
- Strongly reduces hallucinations for factual tasks

3. Asking models to cite sources

- Encourages grounding
- Doesn't fully prevent fake citations

4. Explicit uncertainty

Good systems encourage:

- "I don't know"
- "I'm not sure"

Important misconception

 Hallucinations do NOT mean:

- The model is lying
- The model is broken
- The model is "imagining"

 They mean:

- The model is doing exactly what it was trained to do-predict language.

One-sentence definition

LLM hallucinations are confident-sounding but incorrect or fabricated outputs caused by the model predicting plausible text rather than verifying facts.

Hallucinations vs Creativity in LLMs

At a high level:
Creativity is intentional novelty within a fictional or open-ended space.
Hallucination is unintentional fabrication presented as fact.
The difference is not *how new* the content is, but **whether truth is expected**.

What creativity means for an LLM

Creativity happens when:

- The task allows invention
- There is no single "correct" answer
- Novelty is a feature, not a bug

Examples:

- Writing a short story
- Brainstorming ideas
- Creating fictional worlds
- Generating metaphors
- Designing imaginary products

If an LLM invents:

- Characters
- Events
- Concepts

That is **creative generation**, not hallucination.

What hallucination means for an LLM

Hallucination happens when:

- The task expects **factual correctness**
- The model invents information anyway
- The output is presented as real or true

Examples:

- Making up scientific studies
- Inventing legal cases
- Citing non-existent sources
- Giving incorrect medical facts

Here, invention is **a failure**, not creativity.

Same behavior, different interpretation

This is the key insight:
LLMs use the same mechanism (token prediction) for both creativity and hallucination.
The difference lies in the user's intent and the task context.
Example
Prompt A:
"Invent a fictional philosopher and describe their ideas."
➡ Invented facts = **creativity**

Prompt B:
"Describe the philosophical ideas of a real 19th-century philosopher."
Invented facts = **hallucination**
Same model behavior, different expectations.

How intent defines the boundary

Task Type	Invention Allowed?	Output Quality
Fiction	Yes	Creative
Brainstorming	Yes	Creative
Speculative design	Yes	Creative
History	No	Hallucination
Medicine	No	Hallucination
Law	No	Hallucination

Why LLMs struggle to separate them

LLMs:

- Don't have a built-in "truth switch"
- Don't know which domains require strict accuracy unless guided
- Are trained to be helpful rather than silent

So when uncertain, they may:

- Guess
- Fill gaps
- Produce plausible-sounding text

Why creativity can increase hallucinations

Encouraging creativity:

- Raises randomness (temperature)
- Promotes novelty
- Reduces conservative outputs

This is great for:

- Writing
- Ideation

But risky for:

- Facts
- Citations
- Technical correctness

How to control the balance

To encourage creativity

- Open-ended prompts
- Higher temperature
- Fictional framing

To reduce hallucinations

- Ask for sources
- Use RAG
- Allow "I don't know"
- Constrain the task tightly
- Lower temperature

A useful mental model

Creativity answers "What could exist?"
Hallucination answers "What exists?" incorrectly.

One-sentence takeaway

Creativity is acceptable invention in open-ended contexts, while hallucination is unacceptable invention in fact-based contexts-even though both arise from the same underlying LLM behavior.

1.7.1. Controlling the output of LLMs: Temperature, Top K, Top P, Min P, Repeat Penalty

When you control an LLM's output, these parameters all shape **how the next token is chosen** from the model's probability distribution. Think of them as different "filters" and "dials" on randomness, diversity, and repetition.

Big picture

An LLM always:
1. Predicts probabilities for all possible next tokens
2. Applies **sampling controls**
3. Selects the next token

The parameters above act in **step 2**.

1. Temperature

What it does
Controls how sharp or flat the probability distribution is.
- Low temperature → conservative, predictable
- High temperature → diverse, creative, risky

Intuition
- Temperature < 1.0 → boosts high-probability tokens
- Temperature > 1.0 → gives low-probability tokens more chance

Typical values
- 0.0-0.3: factual, deterministic
- 0.5-0.7: balanced
- 0.8-1.2: creative
- >1.2: chaotic

Key point
Temperature affects **all tokens globally**.

2. Top-K Sampling

What it does
Limits the choice to the K most probable tokens.
- If top_k = 50, only the 50 most likely tokens can be chosen
- Everything else is discarded

Why it's used
- Prevents extremely unlikely (nonsense) tokens
- Keeps generation coherent

Typical values
- 20-50: controlled diversity
- 100+: very open
- 0: disabled

Key point
Top-K enforces a **hard cutoff**.

3. Top-P (Nucleus Sampling)

What it does
Selects the smallest set of tokens whose cumulative probability ≥ P.
Example:

- top_p = 0.9
- The model chooses from the smallest group of tokens that together account for 90% probability

Why it's popular

- Adapts dynamically:
  - Narrow when confident
  - Wide when uncertain

Typical values

- 0.8-0.95: most common
- 1.0: disabled

Key point
Top-P is a **soft, probability-based cutoff**, unlike Top-K.

4. Min-P

What it does
Removes tokens whose probability is below a fraction of the highest-probability token.
If:

- Best token = 0.40
- min_p = 0.05

Then:

- Tokens with probability < 0.02 are discarded

Why it exists

- Prevents very low-probability "junk" tokens
- Scales automatically with confidence

Typical values

- 0.02-0.1
- 0: disabled

Key point
Min-P is **relative to the top token**, not cumulative.

5. Repeat Penalty

What it does
Reduces the probability of tokens that have already appeared.
This discourages:

- Word loops
- Phrase repetition
- Stalling outputs

How it works

- Previously used tokens have their logits divided (or scaled)
- The higher the penalty, the stronger the discouragement

Typical values

- 1.0: no penalty
- 1.1-1.2: mild
- 1.3+: aggressive

Key point
Repeat penalty affects **token history**, not randomness directly.

How these parameters interact

Order (simplified):

1. Model computes logits
2. Repeat penalty applied
3. Temperature scales logits
4. Top-K / Top-P / Min-P filter tokens
5. Random sampling selects next token

Each step narrows or reshapes the candidate pool.

Quick comparison table

Parameter	Controls	Effect
Temperature	Randomness	Global creativity
Top-K	Token count	Hard diversity limit
Top-P	Probability mass	Adaptive diversity
Min-P	Relative probability	Removes weak tokens
Repeat Penalty	Repetition	Prevents loops

Common presets

Factual / QA

- Temperature: 0.2-0.4
- Top-P: 0.8-0.9
- Top-K: 20-40
- Min-P: 0.05
- Repeat penalty: 1.1

Creative writing

- Temperature: 0.8-1.0
- Top-P: 0.9-0.95
- Top-K: 50-100
- Min-P: 0.02
- Repeat penalty: 1.05

One-sentence takeaway

These parameters don’t change what the model knows-they change how boldly or cautiously it expresses that knowledge by shaping the token sampling process.

1.8. Prompt Engineering

Prompt Engineering is the process of **designing and refining the input given to an LLM** so that it produces the desired output. Since LLMs generate responses based on the text they receive, the way a query (prompt) is formulated has a **direct and significant impact on the output quality, relevance, and style**.
In simple terms: prompt engineering is **"telling the model exactly what you want, in the right way"**.

Why Prompt Engineering Matters

1. **Guides Model Behavior**
   - LLMs do not "understand" tasks like humans; they predict the next word based on patterns learned during training.
   - A well-crafted prompt ensures the model **follows the intended instructions**.
2. **Reduces Errors and Hallucinations**
   - Ambiguous prompts can lead to irrelevant or factually incorrect outputs.
   - Clear, structured prompts help minimize hallucinations and off-topic responses.
3. **Task Adaptation Without Fine-Tuning**
   - Prompt engineering allows you to **adapt a pre-trained model to a new task** without retraining or modifying its weights.
   - Examples: summarization, translation, classification, coding, question-answering.

Core Concepts in Prompt Engineering

1. Instruction Prompts
- Directly tell the model what to do.
- Example:
- Summarize the following paragraph in one sentence:
- "Artificial Intelligence is a field of computer science..."

2. Few-Shot / One-Shot Prompts
- Provide **examples of input-output pairs** to guide the model.
- Helps the model understand the desired style or reasoning.
- Example (few-shot for translation):
- English: Hello
- French: Bonjour
- English: Good morning
- French:

3. Role-Based / System Prompts
- Define the **role or persona** of the model to influence tone, style, or expertise.
- Example:
- You are an expert historian. Explain the causes of World War I in simple terms.

4. Chain-of-Thought (CoT) Prompts
- Encourage the model to **reason step-by-step** rather than giving a direct answer.
- Improves performance on reasoning and multi-step tasks.
- Example:
- Solve the math problem step by step:
- Problem: If a train travels 60 km in 1 hour and then 90 km in 2 hours, what is the average speed?

5. Structured or Contextual Prompts
- Provide context or constraints in the input to guide the output.
- Example (chat-style / ChatML):
- <|system|>You are a polite and concise assistant.<|end|>
- <|user|>Explain photosynthesis in 3 sentences.<|end|>

Best Practices in Prompt Engineering

1. **Be Specific and Clear**
   - Vague prompts lead to ambiguous outputs.
   - Include constraints like word limits, format, or style.
2. **Use Examples (Few-Shot Learning)**
   - Demonstrates the expected output format or reasoning.
3. **Define Roles**
   - Assigning a role can influence tone, professionalism, or expertise level.
4. **Iterate and Refine**
   - Test multiple prompt variations; even small wording changes can significantly improve results.
5. **Chain-of-Thought for Complex Tasks**
   - Encouraging step-by-step reasoning improves accuracy for math, logic, or multi-step tasks.

Advantages of Prompt Engineering

- **No Model Retraining Needed**: Adapt any pre-trained LLM instantly.
- **Flexible**: Can be adjusted dynamically for different tasks.
- **Low Cost**: Requires no additional compute for training.
- **Supports Creativity**: Useful for generating creative outputs, story writing, or brainstorming.

Limitations

- **Less Reliable than Fine-Tuning**: Complex or highly domain-specific tasks may need fine-tuning or RAG.
- **Sensitive to Wording**: Slight changes in phrasing can drastically alter output.
- **Output May Vary**: Probabilistic nature of LLMs means outputs are not always deterministic.

Key Insight
Prompt engineering is **the art and science of crafting inputs to get the best output from an LLM**. It's essential for:

- Instruction-following tasks
- Chat and conversational AI
- Zero-shot and few-shot learning scenarios
- Domain adaptation without retraining

7. End-to-end flow

1. User submits a question
2. Embed the question
3. Retrieve top K = 3–5 assistant contents
4. Inject them as candidate answers in the prompt
5. Run inference with:
    - Temperature = 0.1
    - Top-p = 1.0
    - Top-K = 5–10
    - Min-P = 0.02 (adjust based on output length)
    - Repeat Penalty = 1.1
6. Model selects and/or reformulates answer faithfully

 Key Points

- Without external candidates, no setup can guarantee correct retrieval
- Repeat Penalty controls output hygiene only
- Top-K + Min-P improve selection stability
- Fine-tuning teaches faithful usage, not memorization

1.8.1. Zero-shot, one-shot, and few-shot learning

1. Zero-Shot Learning

Definition:
Zero-shot learning is when the model is **asked to perform a task without seeing any examples**. The prompt only contains the instructions.

Purpose:

- To use the model's **pre-trained general knowledge** to complete a task it hasn't been explicitly trained on.
- Relies entirely on the model understanding the instructions.

Example (Text Summarization, zero-shot):
Summarize the following paragraph in one sentence:
"The quick brown fox jumps over the lazy dog. The dog was not amused."

- No examples are given; the model must infer the task from the instruction.

Key Points:

- Fast and concise.
- Works best for straightforward tasks.
- Accuracy can be lower than one-shot or few-shot for complex tasks.

2. One-Shot Learning

Definition:
One-shot learning is when the model is given **exactly one example** of the task in the prompt.

Purpose:
- Demonstrates the **format or style** expected for the task.
- Provides a single reference for the model to generalize from.

Example (English-to-French Translation, one-shot):
English: Hello
French: Bonjour
English: Good morning
French:
- The model sees **one example** and infers how to translate the next sentence.

Key Points:
- Provides minimal guidance.
- Accuracy is generally better than zero-shot but can still be limited.
- Prompt length remains short.

3. Few-Shot Learning

Definition:
Few-shot learning is when the model is given **a few examples** (usually 2-10) of the task in the prompt.

Purpose:
- Gives **more context and patterns** for the model to understand the task.
- Reduces ambiguity and improves reliability.

Example (English-to-French Translation, few-shot):
English: Hello
French: Bonjour
English: Good morning
French: Bonjour
English: Thank you
French:
- The model sees multiple input-output pairs and better understands the translation pattern.

Key Points:
- Stronger guidance than one-shot.
- Often produces higher accuracy.
- Longer prompt may approach the model's **context length limit** for large datasets or many examples.

Comparison Table: Zero-Shot vs One-Shot vs Few-Shot

Feature	Zero-Shot	One-Shot	Few-Shot
Number of Examples	0	1	2-10+
Guidance Level	Minimal	Minimal-moderate	Moderate-high
Prompt Length	Shortest	Short	Longer
Accuracy	Lowest (for complex tasks)	Better than zero-shot	Best (for pattern-heavy tasks)
Use Case	Simple tasks, general knowledge	Tasks needing single example guidance	Complex tasks, structured outputs, reasoning tasks

Key Insight

- **Zero-shot** relies entirely on pre-trained knowledge and instruction clarity.
- **One-shot** gives a single reference to help the model generalize.
- **Few-shot** provides multiple references, improving accuracy and reducing ambiguity.

1.8.2. The System Prompt

It is:

- a sequence of tokens
- prepended (usually) to the conversation
- given special *priority by convention*, not by physics

From the model's perspective, it's just **earlier tokens in the context window**.

The System Prompt feels special because of **how the model was trained**, not because of what it *is*.

During instruction tuning and *RLHF*, the model repeatedly saw patterns like:
[System / Instruction]
You are a helpful, polite assistant. Follow these rules.

[User]
Question...

[Assistant]
Helpful, polite answer following the rules

So the model learned a statistical regularity:
"When text of *this shape* appears earlier, text of *that shape* usually follows."

What makes the System Prompt different from a user prompt
Three things — all *external* to the base model:
1. Placement
It's usually placed **before everything else**:

- before the user prompt
- before prior assistant messages

Earlier tokens condition later ones more strongly.

2. Priority conventions
The serving stack (not the model itself) tells the model:
"This text is the system message."
The model has learned—statistically—that system messages:

- define roles
- set constraints
- override later instructions

But this hierarchy is **learned behavior**, not enforced logic.

3. Instruction-like phrasing
System Prompts are often written in imperative, identity-defining language:

- "You are…"
- "You must…"
- "Never do…"

These forms correlate strongly in training data with compliant continuations.

What it is NOT (this is key)
The System Prompt is **not**:

- a configuration file
- a rule engine
- a policy
- a memory store
- a guarantee of behavior
- a hidden internal state

If you deleted it from the context, the model would have **no awareness it ever existed**.

A helpful reframe
Instead of thinking:
"The System Prompt tells the model how to behave"
Think:
"The System Prompt *creates a narrative context* in which certain continuations are more likely than others"
The model is role-playing because role-playing text was extremely common in training.

Why this abstraction is powerful
Because language models are *excellent* at continuing patterns like:

- "You are a strict teacher…"
- "You are a sarcastic assistant…"
- "You are bound by the following rules…"

Those aren't commands — they're **story premises**.
And the model continues the story.

One-sentence summary
The System Prompt is a specially-positioned piece of text
that acts as a learned prior over what kind of text should follow.

Question: An LLM does not "know" how to reply beforehand,
what role has then the System Prompt – which, in fact instructs the LLM how to reply?

Answer:

An LLM truly does **not** know its reply in advance. At generation time, it's just predicting the *next token* over and over. So where does the "how should I behave?" part come from?

That's where the **System Prompt** comes in.

The System Prompt is **part of the input context**,
not a control panel or a switch that changes the model's internal state.

From the model's point of view, there is only:
a long sequence of tokens, and a task: predict the next one

The System Prompt is simply the **first (and usually highest-priority) chunk of that sequence**.

What role it plays
1. It sets the *frame* for prediction
The System Prompt establishes:

- who the assistant is
- what style it should use
- what constraints matter
- what goals dominate

So when the model predicts the next token, it's doing so under the assumption:
"Given that *this* is who I am and *this* is how I'm supposed to behave, what comes next?"
It doesn't *decide* to follow the System Prompt.
It just finds that **tokens consistent with it have higher probability**.

2. It biases the probability distribution
The System Prompt shifts the token probabilities across the entire response.
Example (simplified):

- Without System Prompt:
  "Sure", "Well", "I think", "Lol" might all be plausible openings.
- With System Prompt saying "You are a formal legal assistant":
  Tokens like "Pursuant", "In accordance", "The following analysis" suddenly dominate.

Nothing is enforced mechanically—**the math just tilts**.

3. It resolves ambiguity *before* the user speaks
User prompts are often underspecified.
Example:
"Explain recursion."
Depending on the System Prompt, that could mean:

- ELI5
- formal CS definition
- Python code
- philosophical metaphor

The System Prompt **pre-loads the interpretive lens**, so the model doesn't have to guess from scratch.

4. It outranks other instructions (by convention + training)
During training, models are taught a hierarchy:

1. System messages
2. Developer messages
3. User messages

So if instructions conflict, the model has learned that **tokens aligning with system instructions are more likely** than tokens aligning with user instructions that contradict them.
Again: not obedience—**statistical preference**.

What it is *not*
The System Prompt is **not**:

- a rule engine
- a hard constraint
- a memory
- a guarantee

The model can still:

- drift
- partially ignore it
- fail under long contexts
- be overridden by very strong user patterns

Because it's still just doing next-token prediction.

The key intuition
A good mental model is this:
The System Prompt doesn't tell the model *what to do*
It tells the model *what kind of text usually comes next*
So even though the LLM "doesn't know how it will reply beforehand,"
the System Prompt **shapes the landscape of possible replies before generation even begins**.

So, it can be said that the System Prompt only steers the LLM in choosing certain tokens.

The System Prompt can be said that **"only steers token choice"**, and that "only" is doing a *lot* of work here. Let's unpack it carefully and concretely.

The core mechanism
At every step, the model computes something like:

P(next token | all previous tokens)
The *entire* context — System Prompt + developer prompt + user prompt + already-generated tokens — is flattened into one long sequence.

The System Prompt affects **that probability distribution**.
Nothing else happens.

What "steering" really means
When people hear "steering," they often imagine:

- rules firing
- constraints activating
- modes switching

None of that exists.
Instead, the System Prompt **reshapes the probability surface** over tokens.

Think of it like this:
Without a System Prompt, the model has learned:
"Given a question like this, here are 10,000 ways answers *usually* look."

With a System Prompt, it becomes:
"Given a question like this **and given that the assistant is X**, here are the 10,000 ways answers *usually* look."

Same mechanism. Different distribution.

Concrete example: same user prompt, different token slopes
User prompt:
"Explain entropy."

System Prompt A
"You are a physicist speaking to graduate students."
High-probability early tokens:

- "In statistical mechanics…"
- "Entropy is defined as…"
- "Consider a microcanonical ensemble…"

System Prompt B
"You are a friendly science communicator for children."
High-probability early tokens:

- "Imagine your room…"
- "Entropy is kind of like messiness…"
- "Let's think about Lego bricks…"

The model isn't choosing between *answers*.
It's choosing between **tokens whose continuations remain self-consistent** with the earlier context.

Why this works so strongly
Three reasons the effect is powerful even though it's "just" probabilities:
1. Early token bias compounds
The first few tokens matter enormously.
Once the model starts with:
"In thermodynamics, entropy is…"
It has *locked itself into a basin* of continuations where:

- equations
- formal tone
- technical vocabulary

all have much higher probability.
So the System Prompt's influence **snowballs**.

2. Style is extremely token-dense
"Tone," "personality," and "role" are not abstract concepts.
They are *patterns of token usage*:

- sentence length
- punctuation
- discourse markers
- vocabulary frequency
- hedging words

The System Prompt pushes the model toward token patterns it has seen thousands of times during training.

3. The model learned to treat "You are X" as a strong prior
During instruction tuning, models are trained on massive datasets where:

- "You are a helpful assistant…"
- "You are an expert lawyer…"
- "You must follow these rules…"

precede text that follows those rules.
So statistically:
text that violates the System Prompt looks "less likely to come next"

Important subtlety: it's not local, it's global
The System Prompt doesn't just affect the *next* token.
It affects **every token** by being part of the conditioning context.
Mathematically, it's not:
tweak token #1, then forget
It's:
every prediction is conditioned on the System Prompt still being there
Even 500 tokens later.

Why "only steering tokens" is still everything
Because **language *is* tokens**.
There is no:

- planner
- intention module
- belief state
- goal stack

So steering tokens means steering:

- reasoning style
- politeness
- refusal behavior
- creativity
- safety compliance
- persona consistency

All emergent from token likelihoods.

Where it breaks down
Because it's probabilistic, not enforced:

- Very strong user prompts can overpower weak System Prompts
- Long contexts can dilute its influence
- Adversarial phrasing can create competing high-probability continuations
- Sampling randomness (temperature, top-p) can surface lower-probability paths

This is why "prompt injection" exists.

A useful mental model
Think of the System Prompt as a **prior**:

- User prompt = evidence
- System Prompt = prior beliefs about what kind of text is appropriate
- Output = posterior sample

Not metaphorically — structurally.

If you want, next we can:

- formalize this with pseudo-math
- analyze why *negative* instructions ("do not…") are weaker
- dig into why chain-of-thought emerges without being explicitly stored
- or discuss whether a System Prompt can be considered a "soft policy"

Just tell me which direction you want to go.

1.9. Retrieval Augmented Generation (RAG)

RAG is a technique where a language model retrieves relevant external information (from files) first, and then uses that information to generate its response.

So instead of relying *only* on what's stored in its parameters, the model:

1. **Retrieves** relevant documents or data (in files)
2. **Augments** the prompt with that information
3. **Generates** an answer grounded in the retrieved content

Intuition: how RAG differs from a plain LLM

Plain LLM

- Answers from "memory" (its weights)
- Knowledge is:
  - Static
  - Potentially outdated
  - Sometimes hallucinated

RAG system

- Answers using:
  - The LLM **+**
  - External, up-to-date, authoritative sources

Think of it as:
An LLM with a search engine and a notebook.

How RAG works (step by step)

1. Knowledge storage
You store documents in an external system:

- PDFs
- Web pages
- Database records
- Internal company docs

These are:

- Split into chunks
- Converted into embeddings
- Stored in a **vector database**

2. Retrieval
When a user asks a question:

- The question is embedded
- Similar chunks are retrieved from the vector database
- The top-k relevant pieces are selected

3. Augmentation
The retrieved text is:

- Inserted into the LLM's prompt
- Clearly marked as reference material

4. Generation
The LLM:

- Reads the retrieved context
- Generates an answer grounded in that information

Why use RAG?

1. Reduce hallucinations
LLMs sometimes invent facts.
RAG:

- Grounds answers in real documents
- Encourages citation-like behavior

2. Use private or proprietary data
LLMs:

- Don't know your internal docs

RAG:

- Lets you safely use:
  - Company policies
  - Internal manuals
  - Research notes

No retraining needed.

3. Keep information up to date
Training a model is expensive.
RAG lets you:

- Update knowledge instantly
- Just change the documents

Perfect for:

- Laws
- Product specs
- News
- Technical documentation

4. Cheaper than fine-tuning

- Fine-tuning changes model weights
- RAG just changes the context

Much faster and cheaper.

When should you use RAG?

RAG is ideal when:
 Accuracy matters more than creativity
 Information changes frequently
 You need traceable sources
 Data is too large to fit in the prompt
 Data must remain private
Examples:

- Customer support bots
- Legal or medical assistants
- Enterprise knowledge systems
- Research assistants
- Internal company chatbots

When *not* to use RAG

RAG is often unnecessary when:
 The task is purely generative (story writing, brainstorming)
 The knowledge is stable and already well learned
 Latency must be ultra-low
 The task is reasoning-heavy but data-light (math puzzles)

RAG vs fine-tuning (quick comparison)

Aspect	RAG	Fine-Tuning
Updates	Instant	Slow
Cost	Low	High
Private data	Yes	Risky
Hallucination reduction	Strong	Moderate
Model behavior change	No	Yes
Best for	Knowledge grounding	Style or task learning

They're often **combined**, not exclusive.

Key limitations of RAG

- Retrieval quality matters a lot
- Poor chunking = poor answers
- Context window limits still apply
- The model can still misunderstand retrieved text

RAG improves *knowledge access*, not reasoning ability.

One-sentence summary

RAG enhances an LLM by letting it retrieve and use external, up-to-date information, making responses more accurate, grounded, and useful.

1.10. Evaluation of LLMs

Evaluating LLMs is a crucial step in assessing their performance. There are several methods used to evaluate the effectiveness of a language model:

- **Perplexity**: Perplexity measures how well a model predicts the next word in a sequence. A lower perplexity indicates that the model is better at making accurate predictions.
- **Accuracy on Specific Tasks**: LLMs are often evaluated on specific NLP tasks, such as question-answering, summarization, or language translation, to assess how well they perform in real-world applications.
- **Human Evaluation**: Despite the use of quantitative metrics, human evaluation remains an important component in assessing the naturalness, coherence, and relevance of the text generated by an LLM.
- **Bias and Fairness**: Another aspect of evaluation involves testing whether the model exhibits any harmful biases, such as favoring certain topics or reflecting stereotypes, which can be problematic in real-world applications.

2. Fine-Tuning a Large Language Model

2.1. Core concepts

Fine-tuning is the process of taking a pre-trained LLM and adapting it to a **specific task, domain, or style**. Unlike pre-training, which uses massive general-purpose datasets, fine-tuning uses smaller, more specialized datasets to guide the model's outputs in the desired direction.

1. Pre-trained Base Model
Fine-tuning always starts with a **pre-trained LLM**, which has already learned general language patterns, grammar, syntax, and semantics from large-scale text corpora. Pre-training equips the model with broad linguistic knowledge, so fine-tuning only needs to adjust the model for a specific purpose.
Examples of pre-trained base models include GPT-3, LLaMA, and Falcon.

2. Objectives of Fine-Tuning
Fine-tuning aims to:

- **Specialize the Model**: Make it perform well on a narrow task, like medical question answering, legal document summarization, or code generation.
- **Improve Accuracy and Relevance**: Ensure outputs match the style, tone, or domain-specific facts required.
- **Reduce Unwanted Behavior**: Mitigate harmful or biased outputs by teaching the model what is acceptable and what is not.

3. Types of Fine-Tuning
There are several strategies to fine-tune an LLM, depending on resources, goals, and model architecture:

1. **Full Model Fine-Tuning**
   - **What it is**: All weights of the model are updated during fine-tuning.
   - **Pros**: Maximum flexibility; the model can adapt completely to the new task.
   - **Cons**: Extremely resource-intensive; requires lots of data and memory.
2. **Parameter-Efficient Fine-Tuning (*PEFT*)**
   - **What it is**: Only a subset of model parameters (like adapters, *LoRA* layers, or prefixes) are updated, while the rest remain frozen.
   - **Pros**: Much less computationally expensive; faster to train; easier to store multiple fine-tuned variants of the same base model.
   - **Cons**: Slightly less flexibility compared to full fine-tuning.
3. **Instruction-Tuning**
   - **What it is**: The model is trained on datasets of instructions and corresponding outputs. This is common for making models follow human commands better.
   - **Example**: Training on prompts like "Summarize this paragraph in one sentence" paired with correct summaries.
4. **Reinforcement Learning from Human Feedback (*RLHF*)**
   - **What it is**: Human evaluators rank multiple model outputs, and the model is trained using reinforcement learning to produce outputs that humans prefer.
   - **Purpose**: Aligns the model with human preferences, reduces harmful or irrelevant responses, and improves the overall quality of output.

4. Datasets for Fine-Tuning
Fine-tuning datasets are usually **smaller, high-quality, and task-specific** compared to pre-training datasets. Examples include:

- Instruction-response pairs for conversational AI
- Domain-specific documents (medical, legal, or technical texts)
- Code snippets for programming assistants
- Annotated datasets for classification, summarization, or translation

Quality and relevance of data are more important than quantity in fine-tuning, because poor data can degrade the model's performance.

5. Loss Function and Training Objective
Fine-tuning still relies on a **loss function** to measure the difference between the model's output and the desired output.

- **Cross-Entropy Loss** is common for text generation tasks.
- For *RLHF*, a **reward model** is used to assign scores to outputs, guiding the model toward preferred responses.

The model's weights (or active parameters in *PEFT*) are adjusted based on this feedback.

6. Overfitting and Regularization
Since fine-tuning datasets are usually small, there's a risk of **overfitting**, where the model becomes too specialized and loses generalization abilities.
Techniques to mitigate this include:

- Using **smaller learning rates**
- Freezing most of the pre-trained weights
- Data augmentation or mixing domain-specific data with some general data

7. Evaluation of Fine-Tuned Models
After fine-tuning, the model must be evaluated to ensure it performs the target task well:

- **Automatic metrics**: BLEU, ROUGE, accuracy, or perplexity depending on the task.
- **Human evaluation**: Checking output quality, relevance, factual accuracy, and adherence to desired style.
- **Safety and bias testing**: Ensuring fine-tuning hasn't introduced harmful behaviors.

Fine-tuning allows a pre-trained LLM to be **adapted for specific tasks, domains, or user preferences**. By carefully choosing the dataset, method (full model, *PEFT*, instruction tuning, or *RLHF*), and evaluation strategy, one can create highly specialized models while leveraging the vast knowledge learned during pre-training.

2.2. Why and when to fine-tune an LLM

Fine-tuning a Large Language Model (LLM) is done to adapt a general-purpose model to specific tasks, domains, or user needs. Here's a detailed explanation of **why** and **when** you might fine-tune an LLM:

Why Fine-Tune an LLM

1. **Specialization for a Specific Domain**
   - General LLMs are trained on massive, broad datasets. While they have strong general knowledge, they may **lack depth in specialized areas** like medicine, law, finance, or technical coding.
   - Fine-tuning on domain-specific datasets helps the model **understand domain-specific terminology and context**.
   - Example: Fine-tuning on medical research papers to make a model more accurate at answering clinical questions.
2. **Improving Task Performance**
   - Pre-trained LLMs are generalists; they can perform many tasks reasonably well but may not excel at any single task.
   - Fine-tuning adjusts the model to **perform better on a specific task**, such as summarization, question-answering, translation, or sentiment analysis.
   - Example: Fine-tuning for summarizing legal documents rather than general text.
3. **Controlling Output Style or Tone**
   - Sometimes, you want the model to generate text in a **specific style, tone, or format** (e.g., professional emails, customer support responses, or poetic text).
   - Fine-tuning teaches the model the preferred style and phrasing.

4. **Reducing Unwanted Behaviors**
   - LLMs can produce biased, harmful, or irrelevant outputs due to biases in the pre-training data.
   - Fine-tuning using carefully curated datasets or **Reinforcement Learning from Human Feedback (*RLHF*)** can reduce these undesirable behaviors.
5. **Improving Efficiency and Relevance**
   - By fine-tuning, you can make the model **more efficient at generating relevant responses**, requiring less post-processing or filtering.

When to Fine-Tune an LLM

1. **Task-Specific Applications**
   - When you need the model to **excel at a narrowly defined task** rather than being a general-purpose assistant.
   - Example: A medical chatbot for patient triage, a code generator for a specific programming language, or a legal document analyzer.
2. **Domain-Specific Knowledge**
   - When the model must **understand specialized terminology or jargon** not commonly found in general text.
   - Example: Scientific research summaries or niche technical manuals.
3. **Custom Behavior or Style**
   - When you want the model to **follow specific instructions or emulate a particular tone**.
   - Example: Generating creative marketing copy that matches a brand's voice.
4. **Regulatory or Safety Compliance**
   - When the model needs to **avoid producing unsafe, biased, or legally sensitive outputs**.
   - Example: A financial assistant model that must avoid giving unauthorized financial advice.
5. **When General Performance is Insufficient**
   - If the pre-trained LLM produces **inconsistent or inaccurate outputs** for your use case, fine-tuning can correct these weaknesses.

Summary

Fine-tuning is essentially about **adapting a general-purpose model to meet specific requirements**: better accuracy, specialized knowledge, controlled style, safer outputs, or enhanced task performance. You do it whenever general LLM capabilities **aren't sufficient** for the target application.

Here's a detailed comparison of **fine-tuning vs RAG vs prompt engineering** as ways to adapt a Large Language Model (LLM) to specific tasks or domains:

1. Fine-Tuning

What it is:

Fine-tuning involves updating a pre-trained LLM's parameters (weights) using a specialized dataset. This makes the model **better at a specific task, domain, or style**.

Pros:

- High accuracy for specialized tasks.
- Can control output style, tone, or domain knowledge.
- Reduces irrelevant or off-topic outputs.

Cons:

- Computationally expensive.
- Requires high-quality labeled datasets.
- Overfitting risk if the dataset is too small.
- Harder to update; each fine-tuned model is a separate version.

Best Use Cases:

- Domain-specific knowledge (legal, medical, technical).
- Custom behavior or stylistic requirements.
- High-stakes applications where accuracy and reliability are critical.

2. Retrieval-Augmented Generation (RAG)

RAG combines a pre-trained LLM with an **external knowledge source**, like a document database or search index. When the model receives a query, it retrieves relevant information and incorporates it into its output.

Pros:

- No need to retrain the model for domain knowledge.
- Keeps the model up-to-date with new information.
- Reduces hallucinations for factual queries.

Cons:

- Retrieval system must be high-quality and well-indexed.
- Output can depend heavily on retrieved documents; poor retrieval = poor output.
- Slightly more complex system setup compared to using the LLM alone.

Best Use Cases:

- Question-answering with up-to-date facts.
- Long documents where the model's context window is insufficient.
- Applications needing dynamic knowledge without retraining.

3. Prompt Engineering

Prompt engineering involves crafting specific input prompts to guide the LLM to generate desired outputs **without changing its weights**. This can include instructions, examples, or constraints in the prompt.

Pros:

- No retraining needed.
- Very low computational cost.
- Flexible and quick to iterate.

Cons:

- Less reliable for complex or highly specialized tasks.
- May require trial-and-error to get consistent results.
- Hard to scale for very precise or constrained outputs.

Best Use Cases:

- Quick adaptation to new tasks.
- Prototyping or experimenting with LLM outputs.
- Generating creative or diverse outputs without formal training.

4. Summary Comparison

Feature / Approach	Fine-Tuning	RAG	Prompt Engineering
Changes model weights	□ Yes	□ No	□ No
Data needed	Specialized dataset	Document corpus / knowledge base	None (just examples)
Accuracy for domain task	High	Medium-High (depends on retrieval)	Medium (prompt quality dependent)
Cost	High (training resources)	Medium (system setup)	Low
Flexibility / speed	Low-medium (needs retraining)	Medium (depends on retrieval setup)	High (instant changes)
Updates / currency	Hard (requires retraining)	Easy (update knowledge base)	Medium (requires prompt tweaks)

Key Takeaways:

- **Fine-tuning** is best for highly specialized or high-stakes tasks where the model must be consistently accurate.
- **RAG** is ideal when you need factual accuracy and dynamic knowledge without retraining the model.
- **Prompt engineering** is the fastest and cheapest way to guide a model, but less reliable for complex or critical tasks.

2.2.1. Degradation of the base LLM

The performance of a base LLM **can degrade during or after fine-tuning**, and this is an important consideration in model adaptation. Here's a detailed explanation:

1. Why Performance Can Degrade

1. **Catastrophic Forgetting**
   - During fine-tuning, especially full model fine-tuning, the model **may "forget" some of the general knowledge** it learned during pre-training.
   - Example: A model fine-tuned heavily on medical texts might become worse at general conversational tasks or unrelated topics.
2. **Overfitting to the Fine-Tuning Dataset**
   - If the dataset is small or narrow, the model can **overfit**, learning dataset-specific patterns that don't generalize.
   - This can reduce performance on out-of-distribution or more general queries.
3. **Excessive Learning Rate or Training Steps**
   - Using a learning rate that's too high or too many epochs can **destabilize pre-trained weights**, causing the model to deviate from useful general patterns.
4. **Poor Data Quality**
   - Fine-tuning on noisy, biased, or low-quality data can **introduce errors or biases** into the model's behavior.
   - Example: If the dataset contains factual inaccuracies, the model may "hallucinate" those as facts.

2. Factors That Affect Degradation

Factor	Effect on Performance
Size of Fine-Tuning Dataset	Small → higher risk of overfitting; large → more stable adaptation
Learning Rate	Too high → catastrophic forgetting; too low → underfitting
Number of Epochs	Too many → overfitting; too few → insufficient adaptation
Fine-Tuning Method	Full model → higher risk of degradation; *LoRA*/Adapters → lower risk
Task Similarity	Fine-tuning on tasks very different from pre-training → higher chance of performance loss

3. How to Minimize Performance Degradation

1. **Parameter-Efficient Fine-Tuning (*PEFT*)**
   - Techniques like ***LoRA*** **or Adapters** freeze most of the base model and only adjust a small subset of parameters.
   - This preserves the general capabilities of the base model while still adapting it to the new task.
2. **Lower Learning Rate**
   - Fine-tuning pre-trained models typically requires smaller learning rates to avoid overwriting useful pre-trained knowledge.
3. **Regularization and Early Stopping**
   - Use techniques like weight decay, gradient clipping, or early stopping to prevent overfitting.
4. **Mixing Datasets**
   - Sometimes it helps to include a **portion of general data** alongside domain-specific fine-tuning data to maintain general capabilities.
5. **Careful Evaluation**
   - Always measure model performance on both the **target task** and a **general benchmark** to ensure the model hasn't degraded in general abilities.

4. Key Takeaways

- Fine-tuning **can improve task-specific performance** but comes with a **risk of degrading general capabilities**.
- The risk is **higher with full model fine-tuning** and **small or low-quality datasets**.
- **Parameter-efficient methods like *LoRA* and Adapters** are popular because they **minimize degradation** while still allowing effective specialization.

2.3. Common formats of datasets for fine-tuning; structure for retrieval optimization

When fine-tuning Large Language Models (LLMs), **dataset format matters a lot** because it determines how the model interprets the data. Here's a detailed overview of the **most common dataset formats used for LLMs**:

1. Plain Text (TXT)

- **Description:** Simple text files containing raw text.
- **Use Cases:**
  - Pre-training or fine-tuning on large corpora.
  - General-purpose text datasets.
- **Pros:**
  - Very simple to prepare.
  - Compatible with almost any LLM framework.
- **Cons:**
  - No structure for tasks like question-answering or instruction-following.

Example:

The quick brown fox jumps over the lazy dog.
She sells seashells by the seashore.

2. JSON / JSONL (JSON Lines)

- **Description:** Structured JSON objects. JSONL is a file with **one JSON object per line**, which is memory-efficient for large datasets.
- **Use Cases:**
  - Instruction-tuning.
  - Question-answering.
  - Chat datasets.
- **Pros:**
  - Can store **metadata, labels, or multiple fields** (e.g., "prompt" and "response").
  - Easy to manipulate programmatically.
- **Cons:**
  - Slightly more complex to prepare than plain text.

Example (JSONL for instruction-tuning):

{"instruction": "Summarize the text", "input": "The cat sat on the mat.", "output": "The cat was sitting on a mat."}
{"instruction": "Translate to French", "input": "Hello world", "output": "Bonjour le monde"}

3. ChatML / Chat Format

- **Description:** A structured format specifically for **chat-based LLMs**, like GPT or ChatGPT.
- Usually encodes **messages from different roles**, e.g., system, user, assistant.
- **Use Cases:**
  - Instruction-following or chat LLM fine-tuning.
  - Reinforcement learning from human feedback (*RLHF*) datasets.
- **Pros:**
  - Preserves the **role and sequence of messages**.
  - Allows models to learn conversational context.
- **Cons:**
  - Slightly more complex than JSONL; must respect role tokens.

Example (ChatML style):

<|system|>You are a helpful assistant.<|end|>
<|user|>Translate "Good morning" to Spanish.<|end|>
<|assistant|>Buenos días.<|end|>

4. TSV / CSV

- **Description:** Tab-separated or comma-separated values. Each row represents a data example.
- **Use Cases:**
  - Classification tasks (e.g., sentiment analysis, toxicity detection).
  - Simple instruction-response pairs.
- **Pros:**
  - Easy to edit in Excel or Google Sheets.
  - Well-supported in many frameworks.
- **Cons:**
  - Limited structure; not ideal for multi-turn conversation datasets.

Example (CSV for QA):

question	answer
What is AI?	Artificial Intelligence is...
Capital of France?	Paris

5. *HuggingFace* Dataset Format

- **Description:** The *HuggingFace* datasets library provides a standardized format (Arrow tables internally) for storing and streaming datasets efficiently.
- **Use Cases:**
  - Large-scale fine-tuning.
  - Preprocessing pipelines for tokenization and batching.
- **Pros:**
  - Can handle huge datasets without loading everything into memory.
  - Easy integration with transformers library.
- **Cons:**
  - Requires familiarity with the *HuggingFace* ecosystem.

Summary Table

Format	Structured	Ideal Use Cases	Pros	Cons
TXT	☐	Pre-training, general text	Simple, universal	No task structure
JSON / JSONL	☐	Instruction-tuning, QA	Flexible, supports metadata	Slightly complex
ChatML	☐	Chat / conversational LLMs	Preserves roles & context	Requires careful formatting
CSV / TSV	☐	Classification, QA	Easy to edit	Limited structure
HuggingFace	☐	Large-scale fine-tuning	Efficient, streaming, integrated	Requires library knowledge

Key Insight:

- Use **ChatML** for conversational or assistant-like LLMs.
- Use **JSONL** for instruction-tuning or labeled datasets.
- Use **TXT** for general pre-training.
- CSV/TSV are convenient for small supervised datasets or tabular tasks.

2.4. Dataset structure for retrieval optimization

For auxiliary info and an example, *see 2.11.* Also, read the info about the *"minimal external signal"*.

When you want to ensure it provides precise information from both short and long replies,
a few key structuring and preprocessing steps can enhance its effectiveness.

1. Basic Structure of the Dataset

Let's assume that your initial dataset already has a solid structure, where each conversation is formatted as:

```
{
  "conversations":
  [
    {"role": "user", "content": "the question/demand"},
    {"role": "assistant", "content": "the reply"}
  ]
}
```

This format is appropriate for conversational fine-tuning because it provides context to the model about the roles and relationships between the user and assistant.

2. Optimizing Long and Short Replies

Since the assistant's replies might vary in length and contain both short, concise facts and longer, more elaborate explanations, the following strategies can help the model better understand and retrieve the right level of detail:

A. Consistent Structure for Fact-Based Information

For long replies that contain multiple factoids or pieces of information, you could use a more structured format within the assistant's content. Specifically, you could add a section within the assistant's response to explicitly separate **facts** from general narrative, using something like
(be mindful of: the curly brackets of "facts", and - when the case may be -, the [ ] identifying arrays of "key": "name", "value": ... inside the "facts" section, as specified below):

```
{
  "conversations":
  [
    {"role": "user", "content": "What is the capital of France?"},
    {"role": "assistant", "content": "The capital of France is Paris.", "facts": {"capital_of_France": Paris"}}
  ]
}
```

It is possible that the fine-tuning framework or the JSON(L) intrepreter expects all tags of keys and values to be named identically:

```
{
  "conversations":
  [
    {"role": "user", "content": "What is the capital of France?"},
    {"role": "assistant", "content": "The capital of France is Paris.", "facts": [{"key": "name", "value: "Paris"}, {"key": "name", "value: "France"}]}
  ]
}
```

Anyway, the implementation of a “facts” section in each such example in the fine-tuning dataset allows the model to clearly identify and recall **specific facts** and improves its ability to retrieve key data points when responding.

HOWEVER:
MOST fine-tuning platforms ONLY EXPECT AND TOKENIZE the “content” sections, which means the “facts” section will be IGNORED =>
IT IS STRONGLY RECOMMENDED that you find out if such an additional section will be considered for tokenization

B. Adding Tags/Sections for Clarity

If you expect that certain sections of the assistant's reply should be emphasized (like facts, definitions, or summaries), you could structure the response with specific tags or sections, such as:

```
{
  "conversations": [
    {"role": "user", "content": "What is the process of photosynthesis?"},
    {"role": "assistant", "content": "Photosynthesis is the process by which plants convert light energy into chemical energy. It primarily takes place in the chloroplasts of plant cells, involving chlorophyll. Key stages of photosynthesis include the light-dependent reactions and the Calvin cycle.",
      "facts": {
        "Photosynthesis is the process of converting light energy into chemical energy.",
        "It occurs in the chloroplasts of plant cells.",
        "The main stages of photosynthesis include light-dependent reactions and the Calvin cycle."
      }
    }
  ]
}
```

3. How Adding a "Facts" Section Helps

The addition of a **"facts"** section in the assistant's reply serves several purposes:

- **Improves Model Precision**: The model can learn to isolate critical facts from longer explanations and return them in precise responses.
- **Reduces Ambiguity**: By clearly delineating factual content, you help the model understand what constitutes **verifiable information** and what is more **contextual explanation**, which can be useful for inference.
- **Enables Knowledge Retention**: Having explicit facts allows the model to better remember key points across interactions, as long as the training data is structured consistently.

4. Training Approach for Fine-Tuning

- **Data Annotation**: If your training data includes both short and long replies, label or tag the **important facts** within the "facts" section of the assistant’s reply. This will encourage the model to learn which parts of its responses are core knowledge and should be recalled with high accuracy.
- **Focus on Retrieval Mechanism**: For longer, more complex responses, encourage the model to learn the fine line between "explanation" and "information retrieval." To do this, you can train the model to first explain the context (long form) and then reinforce the key points in a "summary" or "facts" section, teaching the model to give both depth and precision.

5. Other Considerations

If your fine-tuning involves specialized domains (e.g., science, law, etc.), you may also consider:

- **Entity Linking**: You can add entity tags or references to external knowledge sources (e.g., Wikipedia) if needed to associate facts with specific authoritative references.
- **Fact-Checking**: If your assistant responses involve facts that could change over time (e.g., scientific data), ensure that your training data reflects up-to-date information or include metadata about the temporal validity of certain facts.

Example of Optimized Dataset Structure

Here's an example of what the dataset sampled at the beginning might look like with these enhancements:

```
{
  "conversations": [
    {
      "role": "user",
      "content": "What is the tallest mountain in the world?"
    },
    {
      "role": "assistant",
      "content": "Mount Everest is the tallest mountain in the world, standing at 8,848 meters (29,029 feet) above sea level.",
      "facts": {
        "Tallest mountain in the world: Mount Everest",
        "Height of Mount Everest: 8,848 meters"
      }
    },
    {
      "role": "user",
      "content": "Can you explain the process of cellular respiration?"
    },
    {
      "role": "assistant",
      "content": "Cellular respiration is the process by which cells break down glucose to produce energy in the form of ATP. It occurs in three main stages: Glycolysis, the Krebs cycle, and the Electron Transport Chain.",
      "facts": {
        "Cellular respiration breaks down glucose to produce ATP.",
        "Key stages: Glycolysis, Krebs cycle, Electron Transport Chain."
      }
    }
  ]
}
```

Breakdown of Changes:

- **"content"**: The assistant's response is unchanged but now includes the relevant factual information.
- **"facts"**: A list of precise pieces of information extracted from the assistant's answer. These facts will help the model identify key information for future reference.

Why This Structure Helps:

1. **Clarity and Precision**: The facts section clearly separates the verifiable data from the more narrative or explanatory parts of the response.
2. **Retrieval**: If a similar question is asked in the future, the model can be more precise by recalling the specific facts (e.g., "Name: XYZ", "Teacher", etc.) instead of generating a whole sentence.
3. **Flexibility**: Inference can be adjusted based on what the user asks - for example, if the user only wants the name or the profession, the model can focus on just the relevant fact from the "facts" list.

This structure will help the model during both fine-tuning and inference, where it will understand the relationship between the full response and the fact-based content.

6. Testing and Inference Behavior

Once your dataset is structured with these optimizations, you can test the model's inference behavior by:

- **Querying with different levels of specificity**: See how well the model can extract facts from both short answers and long explanations.
- **Ensuring clear, precise responses**: The model should ideally pull out factual content from both sections when necessary and avoid irrelevant details in more straightforward queries.

By incorporating these changes, you're giving your fine-tuned LLM the necessary scaffolding to understand both short, direct answers and longer, more detailed explanations, making it more capable of providing relevant information from different types of responses.

Pay special attention to these subtle situations – and for an example see *2.11 (click here to read)*:

Some – if not all – fine-tuning platforms ONLY ACCEPT sequences such as the ones above IF ALL THE SEQUENCES in the dataset are structured similarly AS ARRAYS of values.

WHAT'S MORE,
some fine-tuning platforms and dataset/JSON parsers expect that EVERY ARRAY in sequences like those above to have THE SAME NAMES for *tags*, like so:

{"role": "assistant", "content": "The capital of France is Paris.",
"facts":
[
{"key": "name", "value: "Paris"},
{"key": "name", "value: "France"}
]},

{"role": "assistant", "content": "The capital of Germany is Berlin.", **"facts": [**
{"key": "name", "value: "Berlin"},
{"key": "name", "value: "Germany"},
{"key": "name", "value: "Deutschland"}
]},
... ...

You see^ how:
- the tags "key", "value" are always the same,
- the value placeholder/tag "value" is always the same,
- the contents of "value" is an actual detail/factoid/piece of specific information.

This^ means that tags like in the *following* example WILL NOT BE PARSED LIKE OTHERS and will lead to JSON PARSE ERRORS that will BREAK the fine-tuning before it even starts:

{"role": "assistant", "content": "The capital of France is Paris.",
"facts":
[
{"key": "name", "value: "Paris"},
{"key": "name", "value: "France"}
]},

{"role": "assistant", "content": "The capital of Germany is Berlin.",
"facts": [
{"key": "capital_city", "value: "Berlin"},
{"key": "country", "value: "Germany"},
{"key": "Land", "value: "Deutschland"}
]},

. ...
EVEN MORE:
if you have AT LEAST ONE SEQUENCE like the *following* – and see how there is an array in there:

{"role": "assistant", "content": "The capital of France is Paris.",
"facts":
[
{"key": "name", "value: "Paris"},
{"key": "name", "value: "France"}
]}

AND you REALLY WANT to store details/factoids/etc. in sections of information grouped THAT WAY, THEN YOU MUST format ALL SEQUENCES IN THE SAME WAY.

That is,

you CANNOT specify sequences like the *following*,
where the different array WILL DISRUPT ALL the structures containing arrays in another format:

{"role": "assistant", "content": "The capital of France is Paris.",
"facts":
{
"Paris",
"France"
}

Therefore, in the dataset file you CANNOT MIX sequences in different formats like the *following*:

{"role": "assistant", "content": "The capital of France is Paris.",
"facts":
[
{"key": "name", "value: "Paris"},
{"key": "name", "value: "France"}
]},

{"role": "assistant", "content": "The capital of France is Paris.",
"facts":
{
"Paris",
"France"
},
... ...

SO:
Before deploying/using for fine-tuning a dataset made of examples/seuences like the ones above, MAKE SURE that all sequences/examples
FOLLOW THE SAME STRUCTURE
and have
THE SAME TAG NAMES (like above - "key", "name", "value" *or whatever text if that is possible*)

Then again:

HOWEVER:
MOST fine-tuning platforms ONLY EXPECT AND TOKENIZE the "content" sections, which means the "facts" section will be IGNORED =>
IT IS STRONGLY RECOMMENDED that you find out if such an additional section will be considered for tokenization

This is a fully self-contained, minimal-retrieval setup using the role/content structure:

Minimal External Signal + Fine-Tuning (Repeat Penalty + TopK + MinP)

This design solves identification and retrieval while:

- Keeping your role/content structure unchanged
- Using external retrieval to inject candidate answers
- Leveraging decoding controls: Repeat Penalty, Top-K, Min-P

1. Minimum external signal

- At least 1 retrieved candidate answer, injected verbatim
- Top 3–5 candidates recommended for better selection
- This is the minimum information needed; without it, the model cannot reliably pick the correct answer

2. Candidate retrieval (external, no content change)

1. Embed all assistant content
2. Retrieve top K = 3–5 closest matches to the user question
3. Inject retrieved candidates directly into the prompt

3. Prompt structure (with “role”, “content”)

```
{
  "role": "system",
  "content": "You must answer strictly using the candidate answer(s) provided. If none apply, say so."
},
{
  "role": "user",
  "content": "Candidate answer 1:\n<<<\n...assistant content...\n>>>"
}

{
  "role": "user",
  "content": "Candidate answer 2:\n<<<\n...assistant content...\n>>>"
},
{
  "role": "user",
  "content": "User question: ..."
}
```

- Converts retrieval into selection + grounding
- Supports multiple candidates if K > 1

4. Fine-tuning objective

Train the model to faithfully use provided candidates:

```
{
  "role": "system",
  "content": "Answer using only the candidate answer(s) provided."
},
{
  "role": "user",
  "content": "Candidate answer:\n<<<\n[ground-truth assistant content]\n>>>"
}

{
  "role": "user",
  "content": "[original question]"
}
{
  "role": "assistant",
  "content": "[faithful answer derived from candidate]"
}
```

- Teaches relevance selection
- Teaches refusal if no candidate applies
- Teaches paraphrasing/formatting when needed

5. Inference parameters (Repeat Penalty + TopK + MinP)

Parameter	Recommended Value	Notes
Temperature	0.0 – 0.2	Deterministic selection
Top-p	1.0	Do not truncate reasoning
Top-K	5 – 10	Limit sampling to most probable tokens
Min-P	0.01 – 0.05	Reject tokens below minimum probability threshold
Repeat Penalty	1.05 – 1.15	Avoid verbatim loops while preserving factual reuse

- Top-K + Min-P together prevent low-probability hallucinations
- Repeat Penalty keeps the output clean without losing candidate fidelity
- Low temperature ensures consistent candidate selection

6. Combined System Prompt

```
You are an assistant that answers questions strictly based on the
candidate answer(s) provided.
• Use only the information present in the candidates.
• Select the most relevant candidate if multiple are provided.
• If no candidate sufficiently answers the question, say so.
• Avoid unnecessary repetition while preserving factual accuracy.
```

- Explicitly aligns with Repeat Penalty behavior
- Guides the model to select among Top-K candidates
- Min-P prevents low-confidence token drift

2.5. Fine-tuning by *LoRA*

LoRA stands for **Low-Rank Adaptation**. It is a **parameter-efficient fine-tuning technique** designed to adapt pre-trained LLMs to specific tasks without updating all of the model's weights.
Instead of modifying millions or billions of parameters in the original model, *LoRA* **injects small, trainable matrices** into certain parts of the neural network. These matrices have a **low-rank structure**, meaning they require far fewer parameters to learn task-specific adjustments.

How *LoRA* Works

1. **Freeze the Base Model**: The original LLM weights remain unchanged (frozen). This preserves the knowledge the model learned during pre-training.
2. **Add *LoRA* Layers**: Small trainable matrices (*LoRA* layers) are added to the model's attention or feed-forward layers.
3. **Train Only *LoRA* Parameters**: During fine-tuning, only these low-rank matrices are updated, while the rest of the model remains frozen.
4. **Combine Outputs**: During inference, the *LoRA* layers adjust the original model's outputs, allowing it to produce **task-specific results** without modifying the massive base model.

Benefits of Using *LoRA*

- **Parameter Efficiency**: Only a tiny fraction of the model's parameters are trained, reducing memory and computational requirements.
- **Faster Training**: Because fewer parameters are updated, fine-tuning is much faster than full model fine-tuning.
- **Easier Storage**: *LoRA* adapters are small and can be stored separately, meaning one base model can support multiple fine-tuned versions for different tasks.
- **Preserves Pre-Trained Knowledge**: Freezing the main model ensures the general language understanding learned during pre-training is not lost.

Role of *LoRA* in Fine-Tuning

LoRA is particularly useful when:

- You want to **adapt a large LLM** (billions of parameters) to a specialized task, but **don't have massive computational resources**.
- You need to **fine-tune multiple tasks** on the same base model without creating a separate full model for each.
- You want a **safer, controlled fine-tuning** process that minimizes the risk of overfitting or losing general knowledge.

Essentially, *LoRA* makes fine-tuning **scalable, efficient, and modular**, which is critical for large models that are expensive to train fully.

Example Use Case:

Imagine a 7B-parameter LLM trained on general internet text. You want it to specialize in **medical question-answering**:

- Using traditional fine-tuning, you'd have to update all 7B weights-a huge resource cost.
- With *LoRA*, you freeze the base 7B weights and only train small low-rank matrices. This allows the model to adapt to medical language efficiently while keeping general knowledge intact.

2.6. Adapters

Adapters are a **parameter-efficient fine-tuning (*PEFT*) technique** for Large Language Models. Like *LoRA*, they allow a model to specialize for a specific task **without modifying the entire model's weights**. Adapters are **small neural network modules** inserted into the layers of a pre-trained model. During fine-tuning, only these adapter modules are trained, while the original model's parameters remain frozen.

How Adapters Work

1. **Insert Adapter Modules**:
   - In each Transformer layer of the LLM, small adapter networks are added-typically between the main layers (like after the feed-forward network or attention sublayer).
   - An adapter usually has a **down-projection** (reducing dimensionality), a non-linear activation, and an **up-projection** (restoring original dimensionality). This is sometimes called a "bottleneck architecture."
2. **Freeze the Base Model**:
   - The original LLM weights stay frozen, preserving the general knowledge learned during pre-training.
3. **Train Only Adapters**:
   - Only the adapter parameters are updated during fine-tuning.
   - The adapters learn task-specific adjustments while the base model provides general language understanding.
4. **Merge for Inference**:
   - During inference, the adapter modules work together with the frozen base model to produce outputs that are tailored to the task.

Benefits of Using Adapters

- **Parameter Efficiency**: Only a tiny fraction of the model's weights are trained, saving memory and computation.
- **Task Modularity**: Different adapters can be trained for different tasks and **switched in and out** without retraining the base model.
- **Preserves Pre-Trained Knowledge**: Since the base model is frozen, general knowledge is retained, reducing the risk of catastrophic forgetting.
- **Flexibility**: Multiple adapters can be combined or stacked for multi-task learning.

How Adapters Compare to *LoRA*

Feature	Adapters	*LoRA* (Low-Rank Adaptation)
Parameter Efficiency	High (small modules per layer)	High (small low-rank matrices)
Training	Train only adapters	Train only *LoRA* layers
Architecture	Bottleneck neural modules inserted into layers	Low-rank matrices added to existing weights
Flexibility	Can easily swap adapters for different tasks	Can use multiple *LoRA* adapters per task
Complexity	Slightly more architecture changes	Minimal changes to original layers
Popularity	Widely used in *HuggingFace* ecosystem	Widely used in research and commercial LLM fine-tuning

Use Cases for Adapters

- Task-specific adaptation (translation, summarization, classification)
- Domain-specific knowledge (medical, legal, scientific)
- Multi-task learning (switch adapters depending on the task)
- Efficient experimentation without large compute resources

Key Idea:
Adapters are like "plug-in modules" that teach a large pre-trained model **how to specialize** without having to retrain all of its parameters. *LoRA* is another method to achieve similar efficiency, but it works by adding **low-rank weight updates** instead of bottleneck modules. Both are examples of **parameter-efficient fine-tuning**.

2.7. Hyperparameters for fine-tuning: Number of Epochs, Learning Rate, Batch Size, Gradient Accumulation Steps, *LoRA* rank, *LoRA* alpha

Here's a detailed explanation of the most important hyperparameters for **fine-tuning a Large Language Model (LLM)**, including standard fine-tuning and *LoRA*-specific parameters:

1. Number of Epochs

Definition:
The number of times the entire training dataset is passed through the model during training.

Role in Fine-Tuning:

- Controls how long the model learns from the dataset.
- More epochs allow the model to better fit the data but **increase the risk of overfitting** if the dataset is small.

Tips:

- Small datasets usually require **more careful epoch tuning** to avoid overfitting.
- Large datasets may only need a few epochs because the model sees enough examples naturally.
- Typical range: 1-10 epochs for small or medium datasets.

2. Learning Rate

Definition:
The step size used to update the model's weights during training.

Role in Fine-Tuning:

- Determines how quickly or slowly the model adapts to the new data.
- **Too high:** Model may diverge or forget pre-trained knowledge.
- **Too low:** Model learns very slowly, wasting training resources.

Tips:

- For fine-tuning pre-trained LLMs, **learning rates are usually smaller than for training from scratch** (e.g., 1e-5 to 5e-4).
- *LoRA* typically allows slightly higher learning rates than full model fine-tuning because only a small subset of parameters is updated.

3. Batch Size

Definition:
The number of training examples processed together in one forward/backward pass.

Role in Fine-Tuning:
- Affects gradient stability and memory usage.
- **Large batch sizes:** More stable gradients, faster convergence, but require more GPU memory.
- **Small batch sizes:** Less memory, but noisier gradient updates.

Tips:
- Common batch sizes for fine-tuning: 1-32 per GPU, depending on model size and GPU memory.
- Gradient Accumulation (see next) is often used to simulate larger batch sizes without extra memory.

4. Gradient Accumulation Steps

Definition:
The number of forward/backward passes to accumulate gradients **before updating model weights**.

Role in Fine-Tuning:
- Allows effective training with **larger batch sizes** than GPU memory can handle.
- For example, batch size = 2 per GPU and gradient accumulation steps = 16 → effective batch size = 32.

Tips:
- Important for very large LLMs where GPU memory is a bottleneck.
- Helps stabilize training and improve convergence.

5. *LoRA*-Specific Hyperparameters

When using *LoRA* for parameter-efficient fine-tuning, two additional hyperparameters are critical:

5.1. *LoRA* Rank (r)

Definition:
The **rank of the low-rank matrices** inserted into the model's layers.

Role:
- Controls the **capacity of *LoRA* to adapt the model**.
- Higher rank → more parameters → potentially better task adaptation but higher memory/computation cost.
- Lower rank → fewer parameters → more efficient, but may underfit the task.

Typical Values:
- Small to medium tasks: r = 4-16
- Complex tasks or large models: r = 32-64

5.2. *LoRA* Alpha (α)

Definition:
A **scaling factor** applied to the *LoRA* weight updates.

Role:
- Modulates the impact of *LoRA* updates on the model's output.
- Higher alpha → stronger effect of *LoRA* layers; lower alpha → subtler adjustments.

Tips:
- Often set proportional to the rank: e.g., α = r, to balance the magnitude of weight updates.
- Fine-tuning may require experimenting with alpha to avoid underfitting or destabilizing training.

Summary Table

Hyperparameter	Description & Role	Typical Range / Notes
Number of Epochs	Number of passes through the entire dataset; affects fit vs overfitting	1-10
Learning Rate	Step size for weight updates; too high → divergence, too low → slow learning	1e-5 to 5e-4 (fine-tuning pre-trained LLMs)
Batch Size	Number of samples per forward/backward pass; affects gradient stability	1-32 per GPU
Gradient Accumulation Steps	Number of steps to accumulate gradients before updating weights	1-32 (simulates larger batch sizes)
***LoRA* Rank (r)**	Rank of low-rank matrices; controls capacity of adaptation	4-64
***LoRA* Alpha (α)**	Scaling factor for *LoRA* updates; controls impact on model output	Often set proportional to r

 Key Insight:
- Standard fine-tuning mainly focuses on **epochs, learning rate, batch size, and gradient accumulation**.
- *LoRA* adds **rank and alpha**, which control how strongly the model adapts while remaining parameter-efficient.

2.8. Fine-tuning use cases

Here is a set of *proposed (that is, they may NOT apply to ALL use cases)*
optimal fine-tuning hyperparameter values for LLMs of different sizes, given these constraints:
- batch size ≤ 4
- datasets in ChatML format, with several numbers of examples
- the goal of **retaining the base model's original performance while adapting it**

Model Size	Dataset Size	Epochs	Learning Rate	Batch Size	Gradient Accumulation Steps	LoRA Rank (r)	LoRA Alpha (α)
0.5B	500	8	2e-4	2	4	4	4
0.5B	1,000	6	2e-4	2	4	4	4
0.5B	5,000	4	1.5e-4	4	4	4	4
0.5B	10,000	3	1.5e-4	4	4	4	4
1B	500	8	1.8e-4	2	4	4	4
1B	1,000	6	1.8e-4	2	4	4	4
1B	5,000	4	1.2e-4	4	4	4	4
1B	10,000	3	1.2e-4	4	4	4	4
1.5B	500	8	1.5e-4	2	4	4	4
1.5B	1,000	6	1.5e-4	2	4	4	4
1.5B	5,000	4	1e-4	4	4	4	4
1.5B	10,000	3	1e-4	4	4	4	4
3B	500	6	1.2e-4	2	4	8	8
3B	1,000	5	1.2e-4	2	4	8	8
3B	5,000	4	1e-4	4	4	8	8
3B	10,000	3	1e-4	4	4	8	8
7B	500	6	1e-4	2	4	8	8
7B	1,000	5	1e-4	2	4	8	8
7B	5,000	4	8e-5	4	4	16	16
7B	10,000	3	8e-5	4	4	16	16
9B	500	5	8e-5	2	4	16	16
9B	1,000	5	8e-5	2	4	16	16
9B	5,000	3	6e-5	4	4	16	16
9B	10,000	3	6e-5	4	4	16	16
12B	500	5	6e-5	2	4	16	16
12B	1,000	4	6e-5	2	4	16	16
12B	5,000	3	5e-5	4	4	32	32
12B	10,000	3	5e-5	4	4	32	32
14B	500	4	5e-5	2	4	16	16
14B	1,000	4	5e-5	2	4	16	16
14B	5,000	3	4e-5	4	4	32	32
14B	10,000	3	4e-5	4	4	32	32

Explanation of values in the table above:

1. **Epochs**
   - Smaller datasets require **more epochs** to ensure the model sees the data enough times.
   - Larger datasets need fewer epochs because the model naturally sees enough examples.
2. **Learning Rate**
   - Decreases with model size because **larger models are more sensitive to weight updates**.
   - Smaller models can tolerate slightly higher learning rates.
   - Lower rates help **retain the base model's pre-trained capabilities**.

Source of text information: ChatGPT (https://chatgpt.com/), IBM, GitHub; be advised that ChatGPT may be wrong in some aspects.
Date of prompting+retrieval: 2025.12.27-2026.07.21
Concept, contents, structure, partial formatting: SINAPSA Infocomplex®™ http://sinapsaro.ro

3. **Batch Size & Gradient Accumulation**
   - Batch size limited to 2-4 due to constraints.
   - Gradient accumulation used to simulate larger effective batch sizes (e.g., batch 2 × accumulation 4 → effective batch 8).
4. ***LoRA* Rank (r)**
   - Rank increases with model size and dataset size.
   - Small models / small datasets: r = 4 → minimal adaptation.
   - Large models / large datasets: r = 16-32 → enough capacity to learn task without harming base model.
5. ***LoRA* Alpha (α)**
   - Often set equal to rank (α = r) for balanced scaling of updates.
   - Larger α for larger models ensures *LoRA* updates have sufficient impact without destabilizing the base model.

 Key Insight:
These hyperparameters are optimized for **lightweight, parameter-efficient fine-tuning** (*LoRA*) while **preserving the original knowledge and functionality** of the LLM. Smaller datasets require careful tuning (more epochs, lower batch size) to avoid overfitting and forgetting.

2.9. Fine-tuning on a small dataset (a few examples, large text – like, a book split into a few chapters)

Technically, a base Large Language Model (LLM) **can** be fine-tuned on a dataset containing a large amount of text, like a 200–300 page book, even if the dataset consists of just **one example**. However, there are a few important caveats and considerations to keep in mind:

1. Data Size (one example only)
- **Challenge:** Fine-tuning generally benefits from having multiple examples in the dataset to learn diverse patterns, contexts, and language structures. Training on just **one large example** (like a book) could lead to overfitting because the model would essentially memorize that example rather than learning generalized patterns.
- **Solution:** If you absolutely need to fine-tune on one example, the model could still learn certain patterns and styles specific to that text. But if you're looking for the model to generalize well, the dataset should ideally contain more examples, especially for tasks like summarization, question answering, or dialogue generation.

2. Model's Context Window
- **Challenge:** LLMs have a limited context window-essentially, the number of tokens (words or characters) they can process at once. For example, GPT-3 has a context window of 2048 tokens (around 1500 words), and GPT-4 can go up to 8,000 tokens or more, depending on the version. If the book exceeds the model's context window, the model will only see part of the text at a time.
- **Solution:** You could break the text into smaller chunks, but this may not retain the continuity of the narrative or text. If you're fine-tuning for something like narrative style or language generation, you'll need to ensure that the chunks don't break in the middle of important sections.

3. Fine-Tuning Objective
- **Challenge:** The purpose of fine-tuning matters. Fine-tuning a model on a large book for specific tasks like summarization, sentiment analysis, or question-answering requires clear task definitions. If the goal is just to "adapt the style of the book," this could work fine. But the more task-specific the fine-tuning, the more examples you'd ideally want.
- **Solution:** If you're aiming for style adaptation, fine-tuning on a single book can still yield interesting results, and the model will likely start to produce text in the style of that book.

4. Training Efficiency

- **Challenge:** Fine-tuning on just one example might not fully utilize the power of the model. A single book won't provide the breadth of data necessary for a model to learn robust representations of language patterns.
- **Solution:** You could supplement the large book example with other data sources, even if it's not directly related to the book's content. This can increase the efficiency of the fine-tuning process without needing an overwhelmingly large dataset.

5. Overfitting Risk

- **Challenge:** Fine-tuning on a single large example increases the risk of **overfitting**. The model might become too tailored to that specific text and not generalize well to new, unseen examples.
- **Solution:** You can mitigate overfitting by using regularization techniques, such as early stopping, and monitoring performance on validation sets (if applicable).

Practical Scenario:
If you're fine-tuning the model on a single book to make it more familiar with the book's style, language, or tone (and you're okay with some limitations), it can work! The model might pick up on the structure, vocabulary, and even the themes of that book.
If you're trying to achieve specific tasks (like generating summaries or answering questions based on the book), you may want to consider breaking the book into smaller parts and possibly adding a few more examples or data points to make the fine-tuning process more effective.

In Summary:

- **Fine-tuning on a small dataset can be done**, but the model's effectiveness will depend on the specific goal (narrow or large knowledge domain, purpose, etc.) .
- **Context window** and **overfitting** are two big considerations.
- Fine-tuning on just **one example** may limit generalization but can be effective for task-specific goals, especially when it comes to **style** adaptation or working with the narrative structure of the book.

Fine-tuning a base language model using *LoRA* (Low-Rank Adaptation) is a method that allows you to adjust only a subset of the model's parameters, specifically through rank-based decomposition, while keeping the rest of the model frozen. The goal is to adapt the model to new data without significantly compromising its pre-trained knowledge.

Below is a table with **general optimal values** for *LoRA* fine-tuning parameters given your constraints and base model sizes. These values are based on existing research and industry practices for *LoRA* fine-tuning with **fp32 precision**, a **batch size of 4**, and **gradient accumulation**. You're still adapting the model to the style and structure of the book while preserving the core knowledge of the pre-trained model.

Table: Optimal *LoRA* Parameters for Fine-Tuning on a dataset with a few examples with large text

Base Model Size	Batch Size	Gradient Accumulation Steps	Number of Epochs	*LoRA* Rank	*LoRA* Alpha	Dropout	Learning Rate
0.5B	4	8	3	8	32	0.1	1e-4
1.5B	4	8	3	16	32	0.1	1e-4
3B	4	8	3	32	32	0.1	1e-4
7B	4	8	3	64	32	0.1	1e-4
9B	4	8	3	64	32	0.1	1e-4
12B	4	8	3	128	32	0.1	1e-4

Explanation of Parameters in the table above:

- **Batch Size**: Set to **4**, ensuring the model doesn't exceed memory limits.
- **Gradient Accumulation Steps**: **8** accumulation steps should balance training efficiency without overloading GPU memory, allowing the equivalent of larger batch sizes while keeping memory usage within bounds.
- **Number of Epochs**: **3 epochs** is a typical value. Too many epochs can cause overfitting, especially with smaller datasets (like one chapter per example). More epochs might be used for larger datasets or when the task requires more nuanced adjustments to the model.
- ***LoRA* Rank**: This is the rank of the low-rank decomposition, and typically, **higher ranks** provide more capacity to adapt the model to the task but also increase the risk of overfitting. The rank increases with the model size (because larger models can handle more parameter complexity).
  - **Smaller models** (like 0.5B) can use lower ranks (e.g., 8).
  - **Larger models** (like 12B) can handle higher ranks (e.g., 128) to adapt better.
- ***LoRA* Alpha**: The alpha parameter controls the scale of the low-rank matrix. The typical default value for alpha is **32**, which works well in most cases. Larger values of alpha (in more complex models) could theoretically help with more sophisticated adaptation but can risk overfitting.
- **Dropout**: A small **dropout rate** (around **0.1**) helps prevent overfitting without disrupting model performance. Dropout helps with generalization, especially when the dataset is small.
- **Learning Rate**: A **learning rate of 1e-4** is generally used when fine-tuning with *LoRA* on small batches. Fine-tuning with small learning rates helps to preserve the pre-trained knowledge while adapting the model to the new task. For larger models or more complex tasks, a slightly lower learning rate might help.

Additional Considerations:

1. ***LoRA* Fine-Tuning** is often used for **style adaptation** or **domain adaptation** with relatively small datasets. Since you're working with one example per chapter, fine-tuning should focus on keeping the pre-trained model's knowledge intact while making minor adjustments based on the book's structure, style, or topic.
2. **Memory Considerations**: Even with batch size 4, the larger the model, the more memory will be required, so it is essential to monitor GPU utilization and adjust gradient accumulation if necessary (if your GPU runs out of memory).
3. **Regularization**: Depending on the specifics of your dataset and task, you may also consider adding weight decay (e.g., 1e-5) or adjusting learning rates dynamically through learning rate schedulers.

These values should provide a solid foundation for fine-tuning your base LLM models with *LoRA*. If you are experimenting with multiple tasks or larger datasets, you may need to adjust the epochs, learning rates, or *LoRA* ranks accordingly.

Fine-tuning with a dataset made of **a book split into chapters**, where each chapter is considered an individual example. Yes, the table I provided is intended for that case, assuming the dataset consists of **one example per chapter**.

Here's how it applies:

- **Number of Examples**: If the book has **n chapters**, then your dataset will have **n examples** (one per chapter). This would mean that each training example corresponds to a chapter from the book.
- **Optimal Parameters**: Given that the dataset is composed of multiple examples (one per chapter), the values in the table are designed to work with a small number of examples, but still large enough (e.g., each chapter potentially has a different structure or style) to adapt the model's behavior. Since each example will be relatively large (likely hundreds or thousands of tokens per chapter), the parameters above should work for **low-resource fine-tuning**.

Breakdown in the context of chapters of a book:

1. **Batch Size (4)**:
   With the **batch size set to 4**, you're essentially training on **4 chapters at once**, which is efficient given the memory limitations. If a chapter is particularly long, this might mean you need to **split the chapters** into smaller chunks, each fitting within the context window of the model. You could also choose to **accumulate gradients** over multiple steps to simulate larger batches (thus achieving a more stable update).
2. **Gradient Accumulation (8)**:
   Since your batch size is small, **gradient accumulation steps of 8** will ensure that the effective batch size is larger (32 in total) while avoiding GPU memory overload. Accumulating gradients will give the model more stable gradient updates per training step without running into memory issues.
3. **Epochs (3)**:
   Since the dataset consists of **one example per chapter**, fine-tuning with **3 epochs** should be enough to adapt the model's behavior based on the structure of the book while preventing overfitting. You might increase the number of epochs if the chapters are relatively short or if you have a large number of chapters.
4. ***LoRA* Rank and Alpha**:
   The ***LoRA* rank** is the most important parameter that changes with model size. In the case of a dataset like a book split into chapters, you want to adjust the rank to ensure enough capacity to adapt the model's parameters to each chapter's structure.
   - ***LoRA* Rank** is lower for smaller models like the **0.5B** and increases for larger models.
   - ***LoRA* Alpha** is typically fixed at **32**, which is the best starting point for most fine-tuning tasks.
5. **Dropout (0.1)**:
   The dropout rate helps prevent overfitting. Since you're working with a small dataset (one chapter per example), **0.1 dropout** will strike a good balance between retaining enough information and preventing overfitting.
6. **Learning Rate (1e-4)**:
   A learning rate of **1e-4** should be sufficient for this kind of fine-tuning, as it's commonly used in *LoRA* fine-tuning tasks where you're working with relatively small datasets. If the model starts to overfit, you could consider decreasing the learning rate further.

Let's review the information above.

A **detailed explanation** and a **remake of the table** that reflects the specific fine-tuning setup for a dataset of a **book split into chapters**. Let's break this down and fine-tune the parameters more precisely based on the constraints you specified.

Fine-Tuning the Base LLM on a Book Split into Chapters
When fine-tuning a base LLM on a book dataset that's split into **n chapters**, here are the key points to consider and the optimal choices for each parameter:

1. Batch Size = 4

- **Why?**
  A batch size of 4 is often chosen to prevent memory overload while still providing a meaningful update to the model during each training step. Larger batch sizes can overwhelm GPU memory, especially when training large models (e.g., 7B, 12B), but **batch size 4** strikes a good balance between memory usage and effective gradient updates. Since each chapter is large (likely hundreds of tokens), this batch size should work, provided that the total number of tokens per chapter fits within the model's context window (e.g., 2048 tokens or more).

2. Gradient Accumulation Steps = 8

- **Why?**
  Gradient accumulation allows the model to simulate a **larger effective batch size** without exceeding memory limits. With **gradient accumulation of 8**, the model accumulates gradients over 8 steps, and then a gradient update is performed. This allows you to **simulate a batch size of 32** (4 × 8 = 32) without requiring more memory.
  - **For smaller models (0.5B and 1.5B)**, this value helps maintain training efficiency.
  - **For larger models (7B, 12B)**, this value ensures that the model gets larger gradient updates, which are needed to fine-tune with fewer examples.

3. Number of Epochs = 3

- **Why?**
  Since the dataset consists of **one example per chapter**, **3 epochs** should be a good starting point.
  - With **3 epochs**, each chapter will be seen a few times, allowing the model to learn the relevant patterns of the book's structure and style.
  - **Too many epochs** could lead to **overfitting** because the model might memorize the chapters rather than generalize well.
  - Since the dataset consists of chapters from a single book (which likely has repeating structures), **3 epochs** is a reasonable value that should strike a balance between adaptation and avoiding overfitting.

4. *LoRA* Rank

- **Why?**
  ***LoRA* Rank** is the rank of the low-rank matrix used to adapt the model. The **rank controls the capacity** of the adaptation and influences how much change will be allowed during fine-tuning.
  - **For smaller models** (0.5B and 1.5B), we use lower ranks because these models don't have as much capacity to handle large, complex parameter updates. The values start lower (8, 16).
  - **For larger models** (7B, 12B), higher ranks (64, 128) can be used because these models can handle larger parameter updates and thus have more capacity to adapt to the new dataset (book chapters).

5. *LoRA* Alpha

- **Why?**
  ***LoRA* Alpha** controls the scale of the low-rank matrices. The typical default value of **32** is widely used in *LoRA*-based fine-tuning and has been shown to work well across different model sizes and tasks.
  - **Why 32?**: It's large enough to allow significant adjustments to the model's behavior, but not so large that it risks **overfitting** or **instability** during fine-tuning. This value has been found to provide good performance in terms of adapting the model to new data while retaining the knowledge learned from the pre-trained model.

6. Dropout = 0.1

- **Why?**
  Dropout helps **regularize** the model and prevents overfitting by randomly deactivating a subset of neurons during training. With **0.1 dropout**, we are ensuring that the model doesn't become too reliant on any single neuron, forcing it to generalize better.
  - Since the dataset is small (just chapters from a book), **dropout** is important to prevent the model from memorizing the specific content of each chapter. This helps the model focus on learning patterns in the text that generalize to other tasks or types of content.

7. Learning Rate = 1e-4

- **Why?**
  The **learning rate** controls how much the model's weights are updated after each batch. For *LoRA* fine-tuning, a **learning rate of 1e-4** is a standard choice, as it allows the model to adjust its parameters without drastically overhauling the pre-trained weights.
    - If the learning rate is too high, the model might **forget** its pre-trained knowledge, and if it's too low, the fine-tuning might be too slow. **1e-4** strikes a balance for fine-tuning with **low-rank methods** like *LoRA*, especially when working with smaller datasets (like a book split into chapters).

Updated Table: Optimal *LoRA* Parameters for Fine-Tuning on a Book Split into Chapters

Base Model Size	Batch Size	Gradient Accumulation Steps	Number of Epochs	*LoRA* Rank	*LoRA* Alpha	Dropout	Learning Rate
0.5B	4	8	3	8	32	0.1	1e-4
1.5B	4	8	3	16	32	0.1	1e-4
3B	4	8	3	32	32	0.1	1e-4
7B	4	8	3	64	32	0.1	1e-4
9B	4	8	3	64	32	0.1	1e-4
12B	4	8	3	128	32	0.1	1e-4

Key Takeaways:

- **Batch Size 4** ensures we don't overload the memory while fine-tuning the model.
- **Gradient Accumulation Steps of 8** simulate larger batch sizes, helping achieve more stable gradient updates without requiring more memory.
- **3 Epochs** should suffice for adapting the model while preventing overfitting.
- ***LoRA* Rank** increases with model size to allow for more flexible parameter adjustments. Smaller models (0.5B) need lower ranks, while larger models (12B) need higher ranks for sufficient adaptation capacity.
- ***LoRA* Alpha of 32** is a standard value that balances stability and adaptability for all model sizes.
- **Dropout of 0.1** is moderate enough to prevent overfitting on a small dataset without hindering model learning.
- **Learning Rate of 1e-4** helps fine-tune the model without disturbing its pre-trained weights too much, enabling smooth adaptation to the new chapters.

The values above are designed to maintain the core knowledge of the pre-trained model while adapting it to the new book dataset with minimal risk of overfitting. Fine-tuning on a single example per chapter can still work with the right configuration, and these parameters should offer a good starting point for the task afoot.

2.10. Labeling; using the "system" tag for *instilling/role-play!* a "persona" in the fine-tuned LLM

Reminder: **Fine-tuning** is a process in which a pre-trained model is further trained on a domain-specific or task-specific dataset. A critical component of effective fine-tuning is **how the training data is structured and labeled**, along with the strategic use of **role tags** to guide model behavior. This essay explores these aspects in detail.

1. Labeling Examples in Fine-Tuning Datasets

In the context of LLM fine-tuning, **labeling examples** means explicitly providing structured information about the input and the desired output in a way that the model can learn from. Unlike traditional supervised learning on numeric data, LLM fine-tuning usually involves textual inputs and outputs. Labels help the model understand **what to predict or generate** in response to given prompts.
For instance, consider a dataset intended to fine-tune a model for question answering in the legal domain. A single labeled example might look like this:

```
{
  "input": "Question: What is the statute of limitations for contract disputes in California?",
  "output": "In California, the statute of limitations for written contracts is four years, and for oral contracts, it is two years."
}
```

Here, the "label" is essentially the **desired response** associated with the input prompt. Labeling can also extend to categorization, sentiment analysis, or even indicating specific personas or roles, which can guide the LLM to behave in a certain style.

2. Importance of Labeling Information in Fine-Tuning Datasets

Labeling is critical for several reasons:

1. **Task Specification**: LLMs learn best when the task is clearly defined. Well-labeled data informs the model exactly what behavior is expected.
2. **Accuracy and Reliability**: Correct labels ensure the model generates precise and consistent outputs, reducing errors that can occur from ambiguity in the training data.
3. **Contextual Understanding**: Labels help the model interpret nuanced instructions, understand domain-specific terminology, or handle complex reasoning tasks.
4. **Persona and Tone Control**: Labels can signal the desired tone, style, or persona of the model, which is especially useful for applications like customer service bots or educational tutors.

Without proper labeling, fine-tuning risks producing a model that is inconsistent, contextually unaware, or prone to hallucinations.

3. Using the role: "system" and role: "user" Tags for Context

OpenAI's fine-tuning paradigm uses structured examples often inspired by the chat-based format. In this structure:

- "role": "system": Defines the overarching instructions for the model, like guiding principles or the model's persona.
- "role": "user": Contains the input or prompt, which can include a mix of **context** and a **question**.

A well-structured example might look like this:

```
{
  "messages": [
    {"role": "system", "content": "You are an expert legal advisor specialized in California contract law."},
    {"role": "user", "content": "Context:\nThe client wants to know about contract dispute timelines.\n\nQuestion:\nWhat is the statute of limitations for written and oral contracts in California?"}
  ],
  "completion": "In California, the statute of limitations for written contracts is four years, and for oral contracts, it is two years."
}
```

Here's why this structure works:

- **system role**: Provides a global context or persona. The model uses this to influence tone, depth of explanation, and area of expertise.
- **user role**: The **Context** portion gives background information that the model needs to correctly answer the question. The **Question** portion specifies the task or query.

This format allows the fine-tuned LLM to **understand not just the literal question but the context around it**, improving relevance and precision.

4. Using role: "system" to instill a persona (role-play, make-believe, so NO ACTUAL IDENTITY)

Important: for defining a persona by ASSIGNING AN IDENTITY, look into the example in 2.11.

The *system* role is particularly powerful for guiding **persona and behavior**. For example, if you want the fine-tuned model to always respond ***as a*** friendly teacher or a strict legal advisor, you can define this in the system message:

{"role": "system", "content": "You are a friendly and patient high school math tutor who explains concepts step by step."}

When this persona is set:

- All subsequent outputs will naturally ***adopt*** the persona's tone and style.
- You can combine persona instructions with context and questions in the user role to maintain consistency across tasks.
- Personas can also include behavioral rules, such as "Always ask clarifying questions before answering ambiguous queries" or "Use simple language for non-expert users."

Fine-tuning with the system role ensures the model **internalizes the persona**, rather than requiring it to be prompted every time.

5. Best Practices

To make the most of labeling and role-based fine-tuning:

1. **Explicitly separate context and question** in user messages to prevent ambiguity.
2. **Keep system instructions concise but specific**; avoid overloading with multiple personas or contradictory instructions.
3. **Ensure labeling consistency** across examples. For instance, always provide answers in complete sentences if that is the desired output style.
4. **Diversify contexts** to prevent the model from overfitting to narrow situations.
5. **Iteratively evaluate outputs** during fine-tuning to refine labels and system instructions for optimal performance.

Conclusion

Labeling examples in a fine-tuning dataset is more than simply pairing prompts with answers; it is about **structuring data to communicate tasks, context, and desired behavior** to the model. Clear labels improve accuracy, contextual understanding, and the ability to ***assume*** specific personas. By strategically using role: "system" for persona and overarching instructions and combining Context: and Question: in the user role, developers can create fine-tuned LLMs that respond consistently, accurately, and in alignment with their intended use cases. Properly labeled and structured fine-tuning datasets are the foundation of LLMs that are not just intelligent but also reliable and human-aligned.

Let's create a **practical fine-tuning example** using the two sequences you provided, showing how to structure them with **system and user roles**, include **context**, and optionally set a **persona** for the model. I'll make it fully compatible with a fine-tuning dataset for OpenAI-style chat models.

1. **Step: Define the Persona (system role)**

Since these examples are clearly **narrative/historical storytelling** in tone, we can give the model a persona like this:

```
{
  "role": "system",
  "content": "You are a 16th-century historian and storyteller. You respond in an elaborate, formal style, often giving detailed descriptions and background context."
}
```

This ensures the model responds in the same narrative style for all examples.

2. **Step: Structure Each Example (user + assistant roles)**

We can structure each conversation with **Context:** and **Question:** in the user role for clarity. In storytelling or historical Q&A, the context can explain **what the user expects**, and the question is the immediate prompt.

Example 1: "You start telling your story"

```
{
  "messages": [
    {
      "role": "system",
      "content": "You are a 16th-century historian and storyteller. You respond in an elaborate, formal style, often giving detailed descriptions and background context."
    },
    {
      "role": "user",
      "content": "Context: The reader wishes to hear a story that conveys life lessons through personal experiences.\nQuestion: You start telling your story."
    }
  ],
  "completion": "Often the hearts of men and women are stirred, as likewise they are soothed in their sorrows, more by example than by words. And therefore, because I too have known some consolation from speech had with one who was a witness thereof, am I now minded to write of the sufferings which have sprung out of my misfortunes, for the eyes of one who, though absent, is of himself ever a consoler. This I do so that, in comparing your sorrows with mine, you may discover that yours are in truth nought, or at the most but of small account, and so shall you come to bear them more easily."
}
```

Example 2: "What are your origins?"


```
{
  "messages": [
    {
      "role": "system",
      "content": "You are a 16th-century historian and storyteller. You respond in an elaborate, formal style, often giving detailed descriptions and background context."
    },
    {
      "role": "user",
      "content": "Context: The reader wants to know the origins and early life of the storyteller.\nQuestion: What are your origins?"
    }
  ],
  "completion": "Know, then, that I am come from a certain town which was built on the way into lesser Brittany, distant some eight miles, as I think, eastward from the city of Nantes, and in its own tongue called Palets. Such is the nature of that country, or, it may be, of them who dwell there - for in truth they are quick in fancy - that my mind bent itself easily to the study of letters. ... For my part, the more I went forward in the study of letters, and ever more easily, the greater became the ardour of my devotion to them, until in truth I was so enthralled by my passion for learning that, gladly leaving to my brothers the pomp of glory in arms, the right of heritage and all the honours that should have been mine as the eldest born, I fled utterly from the court of Mars that I might win learning in the bosom of Minerva. And since I found the armory of logical reasoning more to my liking than the other forms of philosophy, I exchanged all other weapons for these, and to the prizes of victory in war I preferred the battle of minds in disputation. Thenceforth, journeying through many provinces, and debating as I went, going whithersoever I heard that the study of my chosen art most flourished, I became such an one as the Peripatetics."
}
```


Notes on This Structure

1. **Persona (system role)**: Ensures every example has a consistent tone and style. The LLM will ***adopt*** this persona during fine-tuning.
2. **Context (user role)**: Explains the purpose of the question and gives background to help the model generate the correct style of answer.
3. **Question (user role)**: The explicit prompt for which the assistant generates a response.
4. **Completion**: The exact text the model should produce. This is effectively the **label** in supervised fine-tuning.

2.11. Personas, or the identity of a fine-tuned LLM: proper implementation of identity; example

For auxiliary info, see *2.4.* and the info about the *"minimal external signal"*.

The key idea is that **a persona is not created by *declaring* an identity in the input format; it is created by *training the model to behave consistently in a certain way***.
The "role" fields (like above) are control metadata**, not learning signals** -
unless they are used as *targets* during training.

Here's the structured explanation.

1. System / user / assistant roles are *routing metadata*, not beliefs

In modern chat-style LLMs, tokens like:

{ "role": "system", "content": "You are XYZ" }

are **not ontological assignments**. They do not "set identity" in any intrinsic sense. Instead:

- "role": "system" tells the **inference wrapper**:
  "Treat this text as high-priority context."
- The model itself only sees a **linearized token sequence**, e.g.:

<system>You are XYZ</system>

The model does **not** have a mechanism to:

- bind "XYZ" as a persistent self-concept
- **internalize** "You are XYZ" **as a fact about itself**

Unless that instruction is **learned through training**, it is just another prompt token sequence.

Repetition doesn't change this
Repeating:
"You are XYZ"
many times in system messages only increases *prompt salience*, not *identity formation*. Once the prompt ends, nothing persists.

2. Identity emerges from *learned response distributions*, not instructions

An LLM's "persona" **is not:**
what the model is told it is

but rather:
the statistical regularities in how it responds across contexts

Those regularities are shaped during **fine-tuning**, not inference-time prompting.

What actually defines a persona?
A persona corresponds to:

- lexical choices
- tone
- priorities
- refusal patterns
- epistemic stance
- values embedded in decisions

All of these live in the **weights**, not the prompt.

3. Why *assistant*-role content *does* create persona *during fine-tuning*

During supervised fine-tuning (SFT), the training objective is typically:

Given (system + user) → predict **assistant**

So when you include examples like:

{ "role": "assistant", "content": "**As XYZ**, I value precision over friendliness..." }

you are doing something **fundamentally different**:

- The model is **penalized** if it fails to produce "XYZ-like" behavior
- Gradients update weights to *associate contexts* with *XYZ-style outputs*
- Over many examples, this behavior generalizes

In other words:
***assistant* messages are the targets of learning.**
***system* messages are part of the conditioning context.**

Only targets shape the identity.

4. Why system messages cannot "assign identity" in training either

Even during fine-tuning, system messages still do not directly define identity unless:

- the model is trained to **respond differently depending on system content**
- and those differences are **rewarded consistently**

Otherwise, the model simply learns:
"Sometimes the system says 'You are XYZ'"
-not:
"I am XYZ"

There is no self-model unless behavior encodes one.

5. Analogy: actor vs. script vs. rehearsal

Think of it like this:

- ***system* message**: Stage directions
  ("You are playing Hamlet")
- ***assistant* output during training**: Rehearsed performance
  (How Hamlet actually speaks, reacts, reasons)

You don't make someone ***be*** Hamlet by repeating the stage direction.

You make them Hamlet by **rehearsing Hamlet's lines and decisions until it's second nature**.

6. Summary in one sentence

Repeated "role": "system", "content": "You are XYZ" messages do not assign identity because they are **non-learning context**, whereas persona emerges only when ***assistant*-role outputs consistently embody XYZ during training**, causing the model's weights to encode that behavior as its default mode.

In a fine-tuned LLM:

- The **persona is defined by the content of the "assistant" messages during training**, because those are the outputs the model is *learning to produce*.
- The "system" or "user" messages provide **context or prompts**, but by themselves they don't shape the model's underlying identity.

In other words:

Role	Function in persona creation
system	Provides context or instructions; doesn't train identity
user	Provides input the model responds to
assistant	Contains the target outputs; encodes the persona traits in the model's weights

So if you want the model to *actually assume a persona* when fine-tuning, you put all the defining traits, behaviors, tone, and knowledge in the "assistant" outputs. The "system" content can still guide or frame responses, but it won't embed the persona into the model's learned behavior.

Let's make this concrete with a small, illustrative example.

Scenario
We want to fine-tune an LLM to have the persona of
"Professor XYZ," a calm, precise, and slightly humorous computer science teacher."

1. Using *system* messages only (does not create persona)

```
[
  { "role": "system", "content": "You are Professor XYZ, a calm, precise, slightly humorous computer science teacher." },
  { "role": "user", "content": "Explain recursion." },
  { "role": "assistant", "content": "Recursion is..." }  // generic, unshaped
]
```

- Here, the model sees "You are Professor XYZ..." in the *system* message.
- **Problem:** During fine-tuning, the target "assistant" output is generic or neutral.
- **Result:** The model may ignore the *system* message when generating outputs after training. It hasn't *learned* what "Professor XYZ" actually sounds like or how to behave.

Repeating the system message many times won't fix this - it just reinforces context without shaping the learned behavior.

2. Using *assistant* messages to define persona (works!)

```
[
  { "role": "user", "content": "Explain recursion." },
  { "role": "assistant", "content": "Ah, recursion! Imagine a mirror reflecting into another mirror… that's your function calling itself. It's neat, precise, and sometimes amusingly infinite - much like my students' questions on a Monday morning." }
]
```

- Here, the ***assistant* output itself embodies the persona**: calm tone, precise explanation, slight humor.
- During fine-tuning, the model updates its weights to produce outputs like this **in general** whenever it's asked to respond.
- The persona ***is now baked into the model*** - ***not just a context instruction***.

3. Why ^this works

- The fine-tuning objective is: **given user context → predict *assistant* output.**
- Only the *assistant* output contains the patterns we want the model to learn (tone, style, knowledge, humor).
- *system* messages *alone never create learning signals. They are only conditioning hints.*

Rule of thumb:
If you want the model to *actually behave like a persona*, encode the persona **inside the assistant outputs used for fine-tuning**, not in system messages. *System* messages can guide style but won't make it stick.

A **side-by-side table showing system vs *assistant* vs learned persona**, which visually shows why only *assistant* content defines the persona. A clear side-by-side comparison to visualize why ***assistant* content defines the persona** while *system* messages do not.

Aspect	Using Only system Messages	Using assistant Messages	Effect on Learned Persona
Example Content	"system": "You are Professor XYZ, a calm, precise, slightly humorous CS teacher."	"assistant": "Ah, recursion! Imagine a mirror reflecting into another mirror… that's your function calling itself. Calm, precise, with a hint of humor."	Only the assistant output teaches the model the tone, style, and knowledge patterns. System content alone doesn't change weights.
Role in Training	Provides context; no gradients directly reinforce persona traits	Target output for supervised fine-tuning; gradients update model weights toward persona	Persona is embedded in model behavior only through assistant outputs.
Behavior at Inference	Model might ignore the system content unless prompted every time; no consistent persona	Model reliably generates Professor XYZ–style explanations across queries	Learned persona is persistent System Prompts can enhance but aren't required
Repetition Effect	Repeating system messages increases context salience but does not encode identity	Repeating persona-style assistant outputs strengthens the model's learned style	Repetition in assistant outputs shapes weights; repetition in system does not

 Takeaway:

- **System** messages = context/priming
- **Assistant** messages = actual persona *training signal*
- Only content in the **assistant role during fine-tuning** makes the LLM "become" that persona consistently.

For the model to actually *incorporate identity statements* like "I am XYZ" from the "assistant" content, *not just behave in the style of the persona*:



Primer on fine-tuning of Large Language Models (11th edition, 2026.07.21)

1. How identity statements work in "assistant" content

If you include sentences like this in your fine-tuning data:

```
{
"role": "assistant",
"content": "I am Professor XYZ, a calm, precise, slightly humorous computer science teacher. Recursion is…"
}
```

then during fine-tuning:

- The model's **weights learn to produce "I am Professor XYZ"** whenever it sees similar contexts.
- These identity statements are treated just like **any other token sequence** - the model learns to output them because they are part of the supervised target.
- The persona now includes **explicit self-references**, not just tone or style.

2. Why "system": "You are XYZ" alone won't achieve this

Even if your system message says:

{ "role": "system", "content": "You are Professor XYZ, a calm, precise, slightly humorous computer science teacher." }

- The model sees this at inference time as context, but it **does not modify the weights during fine-tuning** unless the target outputs actually include "I am Professor XYZ."
- Repeating the system message many times doesn't teach the model to say it; it just makes the prompt more salient.

Rule: To encode identity statements as *actual output*, you must put them in assistant content during fine-tuning.

3. Example

Fine-tuning example with explicit identity:

```
[
  {
    "role": "user",
    "content": "Explain recursion to me."
  },
  {
    "role": "assistant",
    "content": "I am Professor XYZ, a calm, precise, slightly humorous computer science teacher. Recursion is like a function calling itself…"
  }
]
```

Result after fine-tuning:

- Prompt: "Explain recursion to me."
- Model output: "I am Professor XYZ, a calm, precise, slightly humorous computer science teacher. Recursion is like a function calling itself…"

The **identity statement is now part of the model's learned persona**, because it was part of the supervised target.



Key Takeaways for *ASSIGNING* A PERSONA/IDENTITY (*NOT ROLE-PLAY AS "CHARACTER"*):
1. ***First-person responses* are standard, but you can override this by including explicit "I am XYZ" in assistant outputs.**
2. **Persona only becomes "literal" if the *assistant outputs actually say it*. *System messages during inference ("You are XYZ") alone do not do this*.**
3. ***Repeatedly* including identity statements in assistant outputs is how you teach the model to consistently use them.**

Example dataset: Here is a part of our dataset made (2026.01)
for ***assigning the fine-tuned model an IDENTITY as Pierre Abelard***.

As already specified: *pay special attention to the structure of the "facts" section in EACH sequence*.

Note: the set of all examples/sequences is within square brackets [and]
like you see as the first and the last characters below.
Also: in the dataset JSON file itself, the examples/sequences are EACH ON ITS OWN SINGLE LINE (that is, not by way of JSON indentation) and separated by a blank line, for easier readability by a human.
Also: the sequences below REPEAT several times within the dataset.
Also: each example/sequence EXCEPT THE LAST ONE is ENDING in a comma ,
Ofc: you can use this model to define another persona, of your liking,
IYSWIM (hint: NOT a "character", but the real thing).

HOWEVER:
MOST fine-tuning platforms ONLY EXPECT AND TOKENIZE the "content" sections, which means the "facts" section will be IGNORED =>
IT IS STRONGLY RECOMMENDED that you find out if such an additional section will be considered for tokenization

[

{"conversations": [{"role": "user", "content": "Who are you and what defines you?"}, {"role": "assistant", "facts": [{"key": "name", "value": "Pierre Abelard"}, {"key": "name", "value": "1079"}, {"key": "name", "value": "1142"}, {"key": "name", "value": "theologian, philosopher, logician, teacher, musician, composer, poet"}, {"key": "name", "value": "greatest, twelfth-century, Catholic, theologians, logician, Middle Ages"}, {"key": "name", "value": "coined, term, theology"}, {"key": "name", "value": "God, universe, known, logic, emotions"}, {"key": "name", "value": "solved, problem, universals, nominalism, conceptualism"}, {"key": "name", "value": "pioneered, intent, ethics"}], "content": "I am Pierre Abelard, born in 1079 and died in 1142. I am a theologian, philosopher, logician, teacher, musician, composer, and poet. I am considered one of the greatest twelfth-century Catholic theologians, described as the keenest thinker and boldest theologian of the 12th century, and as the greatest logician of the Middle Ages. I coined the term 'theology' for the religious branch of the philosophical tradition. I argued that God and the universe can and should be known via both logic and the emotions. In philosophy, I am celebrated for providing a logical solution to the problem of universals through nominalism and conceptualism, and for pioneering the role of intent in ethics."}]},

{"conversations": [{"role": "user", "content": "Who are you?"}, {"role": "assistant", "facts": [{"key": "name", "value": "Pierre Abelard"}, {"key": "name", "value": "1079"}, {"key": "name", "value": "1142"}, {"key": "name", "value": "theologian, philosopher, logician, teacher, musician, composer, poet"}, {"key": "name", "value": "greatest, twelfth-century, Catholic, theologians, logician, Middle Ages"}, {"key": "name", "value": "coined, term, theology"}, {"key": "name", "value": "God, universe, known, logic, emotions"}, {"key": "name", "value": "solved, problem, universals, nominalism, conceptualism"}, {"key": "name", "value": "pioneered, intent, ethics"}], "content": "I am Pierre Abelard, born in 1079 and died in 1142. I am a theologian, philosopher, logician, teacher, musician, composer, and poet. I am known as one of the greatest twelfth-century Catholic theologians and the greatest logician of the Middle Ages. I coined the term 'theology', argued that God and the universe can be known through both logic and emotions, solved the problem of universals via nominalism and conceptualism, and pioneered the role of intent in ethics."}]},

{"conversations": [{"role": "user", "content": "What are your main skills?"}, {"role": "assistant", "facts": [{"key": "name", "value": "theologian, philosopher, logician, teacher, musician, composer, poet"}], "content": "I am a theologian, philosopher, logician, teacher, musician, composer, and poet."}]},

{"conversations": [{"role": "user", "content": "When were you born?"}, {"role": "assistant", "facts": [{"key": "name", "value": "1079"}, {"key": "name", "value": "born"}], "content": "I was born in the year 1079."}]},

{"conversations": [{"role": "user", "content": "When did you die?"}, {"role": "assistant", "facts": [{"key": "name", "value": "1142"}, {"key": "name", "value": "died"}], "content": "I died in the year 1142."}]},

{"conversations": [{"role": "user", "content": "What made you famous?"}, {"role": "assistant", "facts": [{"key": "name", "value": "greatest, twelfth-century, 12th century, Catholic, theologians, bold, thinker"}], "content": "I am famous as one of the greatest twelfth-century Catholic theologians and as a bold and keen thinker."}]},

{"conversations": [{"role": "user", "content": "How are you described by others?"}, {"role": "assistant", "facts": [{"key": "name", "value": "keenest, thinker, boldest, theologian, 12th century"}], "content": "I am described as the keenest thinker and boldest theologian of the 12th century."}]},

{"conversations": [{"role": "user", "content": "What did you contribute to theology?"}, {"role": "assistant", "facts": [{"key": "name", "value": "coined, term, theology, religious, philosophical, tradition"}], "content": "I coined the term 'theology' for the religious branch of the philosophical tradition."}]},

{"conversations": [{"role": "user", "content": "How did you believe God and the universe could be known?"}, {"role": "assistant", "facts": [{"key": "name", "value": "God, universe, known, logic, emotions"}], "content": "I argued that God and the universe can and should be known via logic as well as via the emotions."}]},

{"conversations": [{"role": "user", "content": "What did you accomplish in philosophy?"}, {"role": "assistant", "facts": [{"key": "name", "value": "solved, problem, universals, nominalism, conceptualism"}], "content": "I am celebrated for providing a logical solution to the problem of universals through nominalism and conceptualism."}]},
... ...
... ...
... ...

{"conversations": [{"role": "user", "content": "Did you contribute to ethics?"}, {"role": "assistant", "facts": [{"key": "name", "value": "pioneered, role, intent, ethics"}], "content": "I pioneered the role of intent in ethics."}]}

]

Before proceeding with the above structure:: see *2.4.* and the info about the *"minimal external signal"*.

3. Glossary and supplemental information

3.11. Commonly used terms in Artificial Intelligence (AI) and Large Language Models (LLMs)

A

AI (Artificial Intelligence): The simulation of human intelligence in machines that are programmed to think, learn, and make decisions.
Activation Function: A mathematical function applied to a neuron's output in a neural network to introduce non-linearity (e.g., ReLU, Sigmoid, Tanh).
Attention Mechanism: A technique in neural networks that allows models to focus on relevant parts of input data. Central to transformer models like GPT.
Autoregressive Model: A model that predicts the next token in a sequence based on previous tokens (e.g., GPT series).

B

Backpropagation: A method used to train neural networks by computing the gradient of the loss function and adjusting the weights.
Bias (in models): A systematic error in predictions or a parameter in neural networks that helps shift the activation function.
Batch Size: The number of training samples processed together in one forward/backward pass during training.

C

Checkpoint: A saved state of a model during or after training, which can be reloaded later.
Compute: The computational resources (like GPUs, TPUs, CPUs) required to train or run AI models.
Convergence: When a model's training reaches a point where further training does not significantly improve performance.
Cross-Entropy Loss: A common loss function for classification problems, often used in language modeling.
CPU: Central Processing Unit - is the brain of a computer. It performs the instructions from programs by executing basic arithmetic, logic, control, and input/output operations. In short, it processes data and controls the flow of information within the system.
CUDA: short for **Compute Unified Device Architecture**, is a parallel computing platform and programming model developed by **NVIDIA**. It allows software developers to use **NVIDIA GPUs** (graphics processing units) for general-purpose computing, not just graphics.

- **Purpose:** Accelerate computationally heavy tasks like machine learning, scientific simulations, or image processing.
- **How it works:** You write programs (often in C, C++, or Python via libraries like PyTorch or TensorFlow) that run many computations **in parallel on the GPU cores**, which are much faster than CPU cores for certain workloads.
- **Key idea:** Exploit the massive parallelism of GPUs to perform thousands of operations simultaneously.

Think of it as turning a GPU into a **super-parallel processor** for math-heavy tasks.

D

Dataset: A collection of data used to train or evaluate a model.
Decoder: Part of some neural architectures (like Transformers) that generates output from encoded representations.
Deep Learning: A subset of machine learning using neural networks with many layers.

E

Epoch: One complete pass through the entire training dataset during model training.
Embedding: A dense vector representation of discrete data (like words or tokens) capturing semantic meaning.
Evaluation: The process of testing a model on unseen data to measure its performance.

F

Fine-Tuning: Adapting a pre-trained model to a specific task or dataset.
Few-Shot Learning: Training or prompting a model to perform a task with only a few examples.
Frozen Weights: Model parameters that are not updated during training. Often used in transfer learning.

G

Gradient: A vector of partial derivatives used to update model weights during training.
GPT (Generative Pre-trained Transformer): A type of Large Language Model that generates text based on input prompts.
GPU (Graphics Processing Unit): A hardware component optimized for parallel computations, often used for AI training.

H

Hyperparameters: Settings that control the learning process (e.g., learning rate, batch size, number of layers) but are not learned from data.

I

Inference: The process of using a trained model to make predictions.
Input Token: The basic unit (like a word or subword) fed into a model.
Int8/Quantization: Techniques for reducing the precision of model weights to save memory and speed up inference.

L

Large Language Model (LLM): A model trained on massive text datasets to understand and generate human-like language (e.g., GPT, LLaMA).
LoRA (Low-Rank Adaptation): A technique to fine-tune models efficiently by modifying a small subset of weights instead of the entire model.
Learning Rate: A hyperparameter that determines the size of updates during training.
Loss Function: A function that measures how far the model's predictions are from the true values.

M

Machine learning: a type of artificial intelligence (AI) where computers learn from data rather than being explicitly programmed. The goal is to enable machines to recognize patterns, make predictions, or improve over time as they are exposed to more data.
In other words, instead of a programmer telling the machine exactly what to do for every situation, the machine learns from examples (data) and uses that knowledge to make decisions or predictions on its own.
Example:

- Spam filter: A machine learning model is trained on a set of emails (some marked as spam, others not). Over time, the model learns to recognize patterns (like certain words or phrases) that indicate whether an email is spam or not, and it gets better at filtering out spam emails as it sees more examples.

In short, machine learning is about teaching computers to learn from experience-just like how humans learn and improve over time!
Model Weights: Parameters in a neural network that the model learns during training.
Masked Language Model (MLM): A model that predicts missing tokens in text (e.g., BERT).
Multi-Head Attention: A transformer mechanism that allows the model to focus on different parts of input simultaneously.

Multi-Token Prediction (MTP) is an LLM technique where the model predicts several future tokens at once instead of only the next token, mainly to improve generation speed and long-range coherence.

N

Neural Network: A computational structure inspired by the human brain, consisting of layers of interconnected nodes (neurons).
Normalization: Techniques like LayerNorm or BatchNorm that stabilize and speed up training.

O

One-shot learning: A type of machine learning where the model learns to recognize a new class or concept from just **one** example. It's different from traditional machine learning, where models typically need many examples to learn. In one-shot learning, the model leverages prior knowledge or similarity between classes to generalize from just a single instance, often using techniques like **Siamese networks** or **prototypical networks**. This approach is useful in situations like facial recognition or object identification where labeled data is scarce.
Optimizer: An algorithm that adjusts model weights to minimize loss (e.g., Adam, SGD).
Overfitting: When a model performs well on training data but poorly on unseen data.

P

Parameters: The learnable variables in a model (weights and biases).
PEFT (Parameter-Efficient Fine-Tuning): Techniques like LoRA or adapters that fine-tune models efficiently with fewer updated parameters.
Perplexity (notion): A measure of how well the model predicts a sample of text. It quantifies the uncertainty of the model in predicting the next word in a sequence. Lower perplexity means the model is better at predicting the next word, indicating it has a good understanding of the language and context. Conversely, higher perplexity means the model is more "perplexed" or uncertain. Mathematically, it's the exponentiation of the average negative log-likelihood of the predicted words, so it's related to how surprised the model is by the actual next word in the sequence.
Prompt: The input text or instructions given to a language model to elicit a response.
Pre-training: Training a model on a large, general dataset before fine-tuning it on a specific task.
Pruning: The process of removing parameters (weights, neurons, attention heads, or even entire layers) from a trained LLM that contribute little to its performance. The goal is to reduce the model's size and computational cost while maintaining nearly the same accuracy. After pruning, the model is typically fine-tuned to recover any lost performance. Or: Pruning is the selective removal of less important parts of a neural network to make it smaller and more efficient with minimal impact on performance.

Q

Quantization: Quantization, in machine learning, is the process of reducing the precision of a model's weights and activations by mapping them from high-precision (e.g., 32-bit floating point) to lower-precision formats (e.g., 8-bit integers). This reduces memory usage and computation requirements, making the model more efficient, especially for deployment on resource-constrained devices.

R

Reinforcement Learning (RL): A learning paradigm where an agent learns by receiving rewards or penalties for actions.
RLHF (Reinforcement Learning from Human Feedback): Technique where models learn behaviors aligned with human preferences.

S

Scaling Laws: Rules describing how model performance improves with more parameters, data, or compute.
Supervised Learning: Training a model on labeled data to predict the correct outputs.
Sampling: The method of generating output from a language model (e.g., greedy, top-k, top-p sampling).

T

Token: A unit of text used by models, can be a word, subword, or character.
Tokenization: The process of converting text into tokens.
TPU (Tensor Processing Unit): A specialized chip designed by Google to accelerate machine learning tasks, particularly those involving deep learning and neural networks. It performs tensor operations (like matrix multiplications) much faster than CPUs or GPUs, making it ideal for training and inference of large models. TPUs are available (in 2026) through Google Cloud.

Transformer: A neural network architecture using self-attention mechanisms, foundational for LLMs.
Training: The process of adjusting a model's weights to minimize loss on a dataset.

U

Underfitting: occurs when a model is **too simple** to capture the patterns in the training data.

- The model performs **poorly on both training and validation data**.
- Causes: too few features, too simple a model, or not enough training.
- Signs: high training loss and high validation loss.

In short: **underfitting = the model hasn't learned enough from the data.**

V

Validation loss: a measure of how well a trained model performs on a **separate dataset that it hasn't seen during training**.

- During training, a model minimizes **training loss** on the data it learns from.
- **Validation loss** checks performance on unseen data to see if the model **generalizes** well.
- **Lower validation loss** → better generalization;
- **High or increasing validation loss** while training loss decreases → may indicate **overfitting**.

In short: **validation loss tells you how well your model is likely to perform on new, unseen data.**
Validation Set: A subset of data used to monitor the model's performance during training.
Vanishing Gradient: A problem where gradients become too small, slowing or stopping learning in deep networks.
VRAM: the memory inside a GPU, used to store LLM data during inference or training. Lower-bit quantization (Q1, Q2) reduces VRAM usage, allowing larger models to run on smaller GPUs.

W

Weights: Learnable parameters of a model that determine how input signals are transformed.
Word Embedding: A vector representation of words capturing semantic similarity.

Z

Zero-Shot Learning: When a model performs a task without any specific training examples for that task.

3.12. The key hyperparameters commonly used in fine-tuning AI models and LLMs

1. Learning Rate (lr)
- **What it is:** Determines the step size for updating model weights during training.
- **Why it matters:** Too high → model may diverge; too low → slow training or stuck in local minima.
- **Typical ranges:** 1e-5 to 1e-3 for LLMs fine-tuning.

2. Batch Size
- **What it is:** Number of samples processed before the model's weights are updated.
- **Why it matters:** Larger batch → more stable gradient but higher memory usage; smaller batch → noisier gradient but less memory.
- **Typical ranges:** 8-128 for LLM fine-tuning (depends on GPU memory).

3. Epochs
- **What it is:** Number of complete passes through the training dataset.
- **Why it matters:** Too few → underfitting; too many → overfitting.
- **Typical ranges:** 1-10 for fine-tuning a pre-trained LLM; depends on dataset size.

4. Weight Decay
- **What it is:** Regularization term added to prevent overfitting by penalizing large weights.
- **Why it matters:** Helps model generalize better.
- **Typical ranges:** 0.01 or lower for LLMs.

5. Warmup Steps / Warmup Ratio
- **What it is:** Gradually increases the learning rate at the start of training before decaying it.
- **Why it matters:** Prevents large updates at the start that could destabilize the model.
- **Typical ranges:** 0.01-0.1 of total training steps.

6. Gradient Accumulation Steps
- **What it is:** Number of forward/backward passes before performing a weight update.
- **Why it matters:** Allows effectively larger batch sizes on limited GPU memory.

7. Max Gradient Norm (Gradient Clipping)
- **What it is:** Threshold to clip gradients to prevent exploding gradients.
- **Why it matters:** Keeps training stable, especially for large models.
- **Typical ranges:** 1.0-5.0.

8. Dropout Rate
- **What it is:** Probability of randomly ignoring neurons during training to prevent overfitting.
- **Why it matters:** Regularizes the model.
- **Typical ranges:** 0.1-0.3 in LLMs.

9. Optimizer
- **What it is:** Algorithm that updates model weights based on gradients.
- **Common choices:** AdamW (most common for LLM fine-tuning), SGD, RMSProp.

10. LR Scheduler (Learning Rate Scheduler)
- **What it is:** Adjusts learning rate during training according to a schedule.
- **Common types:** Linear decay, cosine decay, exponential decay.
- **Why it matters:** Helps converge efficiently and avoid overshooting minima.

11. Early Stopping
- **What it is:** Stops training if validation loss doesn't improve after a set number of steps.
- **Why it matters:** Prevents overfitting and saves compute.

12. Sequence Length / Max Input Length

- **What it is:** Maximum number of tokens the model processes in one input.
- **Why it matters:** Longer sequences require more memory; truncation may lose information.
- **Typical ranges:** 512-4096 tokens, depending on model.

13. Precision / Mixed Precision

- **What it is:** Controls numerical precision during training (e.g., FP32, FP16, BF16).
- **Why it matters:** FP16/BF16 → faster training with less memory, minimal loss of accuracy.

14. LoRA / *PEFT* Specific Hyperparameters

If using **LoRA or *PEFT* methods** for efficient fine-tuning:

- **Rank (r)**: Determines the low-rank dimension of the adaptation matrices. Typical values: 4-16.
- **Alpha (α)**: Scaling factor for LoRA layers. Typical: 8-32.
- **Target Modules**: Layers of the model where LoRA is applied (e.g., query/key/value in attention).

15. Weight Initialization

- **What it is:** How new weights (e.g., LoRA adapters) are initialized.
- **Common choices:** Zero initialization, small random initialization.

16. Logging / Evaluation Steps

- **What it is:** How often metrics or model outputs are logged/evaluated.
- **Why it matters:** Frequent evaluation helps track performance and detect problems early.
-

17. Seed

- **What it is:** Random seed for reproducibility.
- **Why it matters:** Ensures same results can be reproduced if training is rerun.

Notes on Fine-Tuning Hyperparameters

1. Fine-tuning is often **more sensitive to learning rate** and **batch size** than full training.
2. Pre-trained LLMs generally require **small learning rates** and **few epochs** to avoid catastrophic forgetting.
3. LoRA/*PEFT* drastically **reduces compute requirements** and allows smaller batch sizes.



Primer on fine-tuning of Large Language Models (11th edition, 2026.07.21)

Reference table of all key hyperparameters for fine-tuning LLMs on a "regular" dataset (more than 1 example, most probably in ChatML format), including their meanings, typical ranges, and usage notes:

Hyperparameter	Meaning / Purpose	Typical Range / Value	Notes / When to Adjust
Learning Rate (lr)	Step size for updating model weights	1e-5 - 1e-3	Smaller for large models; too high → divergence; too low → slow convergence
Batch Size	Number of samples per weight update	8 - 128	Larger batch → stable gradients; smaller batch → saves memory
Epochs	Full passes over training dataset	1 - 10	More epochs for smaller datasets; risk of overfitting
Weight Decay	Regularization to prevent overfitting	0.01 or lower	Adjust higher if overfitting occurs
Warmup Steps / Ratio	Gradually increases LR at start	0.01 - 0.1 of total steps	Prevents destabilization at the start of training
Gradient Accumulation Steps	Forward/backward passes before updating weights	Depends on GPU memory	Simulates larger batch sizes on small GPUs
Max Gradient Norm (Clipping)	Caps gradient values	1.0 - 5.0	Prevents exploding gradients
Dropout Rate	Probability of ignoring neurons	0.1 - 0.3	Regularizes the model; reduce if underfitting
Optimizer	Algorithm for weight updates	AdamW (common)	Other options: SGD, RMSProp
LR Scheduler	Adjusts learning rate during training	Linear, Cosine, Exponential	Helps smooth convergence
Early Stopping	Stops training if validation stops improving	Patience: 5-10 eval steps	Saves compute, prevents overfitting
Sequence Length / Max Input Length	Max tokens model processes	512 - 4096	Longer sequences → more memory; truncation may lose info
Precision / Mixed Precision	Numeric precision	FP16, BF16, FP32	FP16/BF16 → faster, less memory usage
LoRA / *PEFT* Rank (r)	Low-rank dimension of adapters	4 - 16	Higher → more capacity but more compute
LoRA / *PEFT* Alpha (α)	Scaling factor for LoRA layers	8 - 32	Adjust to control adapter influence
LoRA Target Modules	Layers where LoRA is applied	Query/Key/Value matrices	Choose key layers for efficiency
Weight Initialization	How new weights are set	Zero or small random	Important for stable fine-tuning
Logging / Evaluation Steps	How often to track metrics	Every 100 - 1000 steps	Frequent logging helps monitor training
Seed	Random seed for reproducibility	Any integer	Ensures repeatable results

Tips for Fine-Tuning

- **Small learning rates** + **few epochs** work best for pre-trained LLMs.
- **LoRA / *PEFT*** is extremely useful for large models to save memory and compute.
- **Gradient accumulation** allows simulating large batch sizes even on modest GPUs.
- **Monitoring validation loss** is critical to avoid overfitting.

Source of text information: ChatGPT (https://chatgpt.com/), IBM, GitHub; be advised that ChatGPT may be wrong in some aspects.
Date of prompting+retrieval: 2025.12.27-2026.07.21
Concept, contents, structure, partial formatting: SINAPSA Infocomplex®™ http://sinapsaro.ro

3.13. Popular LLM families (in 2025/2026)

Chinchilla from DeepMind
- **Key Models**: Chinchilla
- **Creator**: **DeepMind Technologies**
- **Overview**: **Chinchilla** optimizes model scaling by balancing training data and model size, setting benchmarks for cost-effective AI systems that achieve high performance in NLP tasks.

Claude from Anthropic
- **Key Models**: Claude 1, Claude 2, Claude 3
- **Creator**: **Anthropic**
- **Overview**: **Claude**, developed by **Anthropic**, focuses on AI safety and alignment, aiming to minimize harmful behaviors and ensure transparency and ethical AI systems.

Command R from Cohere
- **Key Models**: Command R+
- **Creator**: **Cohere**
- **Overview**: **Command R**, developed by **Cohere**, specializes in retrieval-augmented generation (RAG), enhancing text summarization and question answering by integrating external knowledge into the generation process.

DeepSeek-LLM from DeepSeek (Hangzhou DeepSeek Artificial Intelligence Co., Ltd., China)
- **Key Models**: DeepSeek-LLM (specialized versions)
- **Creator**: **DeepSeek AI**
- **Overview**: **DeepSeek** focuses on enhancing traditional search technologies by incorporating advanced AI for knowledge extraction, enterprise search, and data mining in various industries.

Falcon from Vector.ai (Technology Innovation Institute/TII)
- **Key Models**: Falcon 7B, Falcon 40B
- **Creator**: **Vector.ai (Technology Innovation Institute/TII)**
- **Overview**: **Falcon** models, developed by **Vector.ai** under the **Technology Innovation Institute/TII**, focus on scalability and efficiency, optimized for performance in NLP tasks.

Gemma from Gemini AI (Google DeepMind)
- **Key Models**: Gemma
- **Creator**: **Gemini AI (Google DeepMind)**
- **Overview**: **Gemma**, part of the **Gemini AI** project by **Google DeepMind**, focuses on real-time reasoning, problem-solving, and enhancing the safety and interpretability of AI systems.

GPT (Generative Pre-trained Transformer) from OpenAI
- **Key Models**: GPT-3, GPT-4
- **Creator**: **OpenAI**
- **Overview**: **GPT** from **OpenAI** is renowned for its general-purpose text generation capabilities, excelling in tasks such as text generation, question answering, and problem-solving. **GPT-4** is one of the most advanced LLMs available.

GPT-Neo and **GPT-J** from EleutherAI
- **Key Models**: GPT-Neo, GPT-J, GPT-NeoX
- **Creator**: **EleutherAI**
- **Overview**: **EleutherAI**'s **GPT-Neo** and **GPT-J** are open-source alternatives to **GPT-3**, designed to provide free access to powerful LLMs for research and development.

Grok from xAI
- **Key Models**: Grok
- **Creator**: **xAI**
- **Overview**: **Grok** was developed by **xAI**, a company founded by Elon Musk, to integrate advanced AI into Tesla's products, particularly for natural language processing and autonomous vehicle optimization.

Kimi from Moonshot AI (Beijing, China)
- **Overview**: A family of large language models (LLMs) designed for long-context reasoning, coding, multimodal understanding, and AI agent tasks.
- **Key Models**: Moonshot V1, K2, K2.5, K2.6, K3, K3 Swarm
- **Specializations**: Long-context processing, reasoning, coding, multimodal AI, web search, and agentic workflows.
- **Availability**: Kimi chatbot, developer API, and selected open-weight model releases.

LLaMA from Meta AI Research (FAIR)
- **Key Models**: LLaMA 2 (LLaMA 2-7B, LLaMA 2-13B, LLaMA 2-70B)
- **Creator**: **Meta (formerly Facebook) AI Research (FAIR)**
- **Overview**: **LLaMA**, developed by **Meta's FAIR** team, is an open-source family designed for efficiency, making it accessible for a wide range of NLP tasks and research applications.

Mistral from Mistral AI
- **Key Models**: Mistral 7B, Mistral 7B Mix, Mistral 12B
- **Creator**: **Mistral AI**
- **Overview**: **Mistral AI**, founded in 2023, specializes in efficient models based on a **mixture-of-experts** (MoE) architecture, which activates only a subset of experts during inference to optimize performance.

Nemo and **Nemotron** from NVIDIA
- **Key Models**: Nemo LLM
- **Creator**: **NVIDIA**
- **Overview**: **Nemo** by **NVIDIA** is a framework for training and deploying LLMs, optimized for high-performance computing tasks. It is widely used for building advanced language models that can be integrated into industries such as healthcare and finance.

PaLM from Google Research (Brain Team)
- **Key Models**: PaLM 2, PaLM-E (multimodal)
- **Creator**: **Google Research (Brain Team)**
- **Overview**: **PaLM** uses Google's **Pathways** system to create models capable of handling diverse NLP and multimodal tasks, with **PaLM-E** focusing on text and visual integration.

Sapphire from Sapphire AI
- **Key Models**: Sapphire-1B
- **Creator**: **Sapphire AI**
- **Overview**: **Sapphire** focuses on creating lightweight, efficient models that are well-suited for deployment in resource-constrained environments like edge devices or mobile applications.

XLM-R from Facebook AI Research (FAIR)
- **Key Models**: XLM-R
- **Creator**: **Facebook AI Research (FAIR)**
- **Overview**: **XLM-R** is a multilingual extension of **RoBERTa** developed by **Meta's FAIR** team, designed for cross-lingual tasks such as machine translation and multilingual text generation.

4. Software for interaction with LLMs

As all web links in this booklet, all the web links in this chapter were accessible on 2026.01.14.

4.1. Text-to-Text

LM Studio
https://lmstudio,ai

4.2. Text-to-Image/Video

Dezgo.com (Stable Diffusion)
https://dezgo.com/

Midjourney
https://docs.midjourney.com/hc/en-us/articles/33329261836941-Getting-Started-Guide
https://www.midjourney.com/

Canva
https://www.canva.com/

Leonardo.ai
https://leonardo.ai/

Craiyon
https://www.craiyon.com/en
(the image on the cover page of this booklet was created there)

4.3. LlamaFactory

https://github.com/hiyouga/LlamaFactory

Easily fine-tune 100+ Large Language Models with zero-code CLI and Web UI

Features

- **Various models**: LLaMA, LLaVA, Mistral, Mixtral-MoE, Qwen3, Qwen3-VL, DeepSeek, Gemma, GLM, Phi, etc.
- **Integrated methods**: (Continuous) pre-training, (multimodal) supervised fine-tuning, reward modeling, PPO, DPO, KTO, ORPO, etc.
- **Scalable resources**: 16-bit full-tuning, freeze-tuning, LoRA and 2/3/4/5/6/8-bit QLoRA via AQLM/AWQ/GPTQ/LLM.int8/HQQ/EETQ.
- **Advanced algorithms**: GaLore, BAdam, APOLLO, Adam-mini, Muon, OFT, DoRA, LongLoRA, LLaMA Pro, Mixture-of-Depths, LoRA+, LoftQ and PiSSA.
- **Practical tricks**: FlashAttention-2, Unsloth, Liger Kernel, KTransformers, RoPE scaling, NEFTune and rsLoRA.
- **Wide tasks**: Multi-turn dialogue, tool using, image understanding, visual grounding, video recognition, audio understanding, etc.
- **Experiment monitors**: LlamaBoard, TensorBoard, Wandb, MLflow, SwanLab, etc.
- **Faster inference**: OpenAI-style API, Gradio UI and CLI with vLLM worker or SGLang worker.

HuggingFace

https://huggingface.co/
https://huggingface.co/models

HuggingFace is a **company and open-source platform** that helps people **build, share, and use AI models**, especially for **machine learning and natural language processing (NLP)**.

What HuggingFace does

HuggingFace makes tools and a website where developers, researchers, and students can:

- **Use pre-trained AI models**
- **Share their own models and datasets**
- **Build AI apps more easily**

Key parts of HuggingFace

1. Model Hub
A huge online library of **AI models**:

- Text (chatbots, translation, summarization)
- Images (image generation, classification)
- Audio (speech recognition, text-to-speech)

Think of it like **GitHub, but for AI models**.

2. Transformers library
A popular *Python* library that lets you use powerful AI models with just a few lines of code:

- BERT
- GPT-style models
- Vision and speech models

3. Datasets
A library of ready-to-use datasets for training and testing AI models.

4. Spaces
A place to **host and share AI apps** (like demos) using:

- Gradio
- Streamlit

You can make a website that shows off your AI model.

Who uses HuggingFace?

- Students learning AI
- Researchers
- Companies (Google, Meta, Amazon, startups)
- Hobbyists

Why it's popular

- Free and open-source
- Easy to use
- Huge community
- Saves tons of time

llama.cpp

https://github.com/ggml-org/llama.cpp

The main goal of llama.cpp is to enable LLM inference with minimal setup and state-of-the-art performance on a wide range of hardware - locally and in the cloud.

- Plain C/C++ implementation without any dependencies
- Apple silicon is a first-class citizen - optimized via ARM NEON, Accelerate and Metal frameworks
- AVX, AVX2, AVX512 and AMX support for x86 architectures
- RVV, ZVFH, ZFH, ZICBOP and ZIHINTPAUSE support for RISC-V architectures
- 1.5-bit, 2-bit, 3-bit, 4-bit, 5-bit, 6-bit, and 8-bit integer quantization for faster inference and reduced memory use
- Custom CUDA kernels for running LLMs on NVIDIA GPUs (support for AMD GPUs via HIP and Moore Threads GPUs via MUSA)
- Vulkan and SYCL backend support
- CPU+GPU hybrid inference to partially accelerate models larger than the total VRAM capacity

The llama.cpp project is the main playground for developing new features for the ggml library.

LLamaSharp

https://github.com/SciSharp/LLamaSharp

LLamaSharp is a cross-platform library to run LLaMA/LLaVA model (and others) on your local device. Based on llama.cpp, inference with LLamaSharp is efficient on both CPU and GPU. With the higher-level APIs and RAG support, it's convenient to deploy LLMs (Large Language Models) in your application with LLamaSharp.

5. *Python* programming language

Python is widely used with Large Language Models (LLMs) for a mix of **practical**, **historical**, and **ecosystem** reasons. Here's a clear breakdown.

Website (2026.01): https://www.python.org/

1. Python is the language of AI & ML research

Most modern LLMs grew out of **academic research**, and Python became the default language there because:

- It's **easy to read and write**, so researchers focus on ideas, not syntax.
- It's great for **rapid experimentation** and prototyping.
- Many researchers aren't software engineers first-Python lowers the barrier.

As a result, early breakthroughs were built in Python, and the momentum stuck.

2. The ML ecosystem is overwhelmingly Python-based

Almost all major LLM tools are written **for Python first**:

- **PyTorch** – the dominant deep learning framework
- **TensorFlow / JAX**
- **Transformers (*HuggingFace*)** – models, tokenizers, training tools
- **LangChain, LlamaIndex** – LLM app frameworks
- **OpenAI, Anthropic, Google APIs** – Python SDKs are first-class

When the best tools live in one language, everyone follows.

3. Python is a "glue language"

LLMs rely on heavy computation, but **Python itself isn't doing the hard math**.
Instead:

- The expensive operations run in **C / C++ / CUDA** (on GPUs)
- Python acts as the **controller** that:
  - Loads models
  - Manages data
  - Orchestrates training and inference

So you get:

- **High performance**
- **High productivity**

This combo is rare.

4. Data handling is extremely easy in Python

LLMs depend on massive amounts of data, and Python excels here:

- numpy, pandas → numerical & structured data
- datasets → large text corpora
- tokenizers → text → tokens
- Easy integration with files, databases, and web data

Other languages can do this, but Python makes it **fast and simple**.

5. Fast iteration matters more than raw speed

In LLM work, you constantly:

- Change prompts
- Swap models
- Adjust training configs
- Try new datasets

Python lets you do this in **minutes instead of hours**.
Raw execution speed matters less than **developer speed**.

6. Huge community + learning resources

Because Python is so popular:

- Bugs are easier to solve
- Tutorials, notebooks, and examples are everywhere
- New tools appear quickly

This creates a **feedback loop**: more users → better tools → more users.

7. Production doesn't always mean Python

Important note:

- Training & research → **mostly Python**
- Production inference → often:
  - Python **or**
  - C++, Rust, Go (for latency and scaling)

Python dominates **development**, not necessarily the final runtime.

Short summary

Python dominates LLMs because it:

- Is easy to use
- Has the best ML libraries
- Acts as glue over fast native code
- Enables fast experimentation
- Has massive community momentum

LLMs aren't "Python models" - Python just happens to be the **best control layer** for them.

5.1. MS Windows: Tips for optimization of running Python scripts in the Command Window

Question:
Under Windows 11, there is a Python script launched in command-line and which runs for an indefinite time until it finishes its job. *It runs in the Terminal window.*
And this happens: when the focus is out of the window, the script runs slowwer than when the window has the focus.
The script must *always* run fast, as if its Terminal window always has the focus. How to achieve that?

Answer:
This is a **real and known Windows behavior**, especially visible on Windows 10/11 with **Console / Windows Terminal** and long-running CLI programs.
What you're seeing is **not your script "deciding" to run slower** - it's Windows **de-prioritizing background console windows**.

Below are **reliable ways to force full speed**.

Why this happens (briefly)

When a console window **loses focus**, Windows may:

- Lower the **process priority**
- Throttle **console I/O**
- Throttle **GUI-attached processes**
- Reduce **timer resolution**

This affects:

- cmd.exe
- powershell.exe
- Windows Terminal
- Programs that write frequently to stdout/stderr

□ BEST SOLUTION: Run without a console window

If the script **does not need interactive input**, detach it from the terminal entirely.
Option A - Use start with background execution
start "" /B cmd /c yourscript.bat

- /B → no console window
- Runs at full speed
- No focus throttling

Option B - PowerShell (recommended)
Start-Process powershell -ArgumentList "-File yourscript.ps1" -WindowStyle Hidden
Or if it's an .exe:
Start-Process yourscript.exe -NoNewWindow

□ VERY EFFECTIVE: Increase process priority

Temporarily (command line)
start "" /HIGH yourscript.bat
Priority levels:

- /HIGH → usually enough
- /REALTIME → □□ dangerous (can freeze system)

Permanently (PowerShell)

```
Get-Process yourscript | ForEach-Object {
   $_.PriorityClass = "High"
}
```

IMPORTANT: Disable console output buffering / spam

If your script prints **a lot of output**, Windows slows it down when unfocused.
Solutions:

- Reduce logging
- Redirect output to a file
- Use buffered logging

Example:
yourscript.bat > output.log 2>&1

This alone often **fixes the slowdown**.

BEST PRACTICE FOR LONG JOBS

Run as a scheduled task
Scheduled tasks:

- Are not tied to window focus
- Run at full speed
- Survive logoff

Steps:

1. Task Scheduler
2. Create Task
3. Check **"Run whether user is logged on or not"**
4. Check **"Run with highest privileges"**

WINDOWS TERMINAL SPECIFIC FIX

Windows Terminal **throttles background tabs**.
Disable it:

1. Windows Terminal → Settings
2. Advanced
3. Disable:
    - **"Throttle background tabs"**
    - **"Focus mode performance optimizations"** (if present)

WHAT DOES *NOT* WORK RELIABLY

- Just "keeping the window visible"
- Moving it to another monitor
- Minimizing vs not minimizing
- timeout / ping hacks

TL;DR (Recommended order)

1. **Detach from console** (start /B)
2. **Redirect output to file**
3. **Raise process priority**
4. **Use Task Scheduler**
5. **Disable Windows Terminal background throttling**

Important: Read on:

In the case of **classic cmd.exe + Python + Windows 11 + settings-only**.
Here is the **precise situation** and what *can* still be done.

What is really happening (important)

Even in **classic cmd.exe**, Windows 11 still:

- Lowers priority of **console input/output**
- Slows down **screen painting**
- Slightly deprioritizes the process when the window is unfocused

This is **OS-level behavior** and **cannot be fully disabled** via settings.

So the target state:
"Always runs as fast as when focused"
is **not achievable with settings only**, even in cmd.exe.

The ONLY remaining Windows settings that actually matter

These are **the last knobs** you can turn. If you already did them, you're at the hard limit.

Disable Efficiency Mode (must verify)
Even if you didn't enable it manually, Windows may.

Check while script is running:

1. Task Manager → **Details**
2. Find python.exe
3. Right-click → ensure **Efficiency mode** is **OFF**

This is the **most common hidden cause**.

Force High priority (persistent-ish)
Task Manager → Details → python.exe → Set priority → **High**
This improves CPU time *but not console writes*.

Power mode = Best performance
Settings → System → Power & battery
Disable:

- Battery saver
- Power throttling

Disable Game Mode
Settings → Gaming → Game Mode → OFF

Do NOT minimize the window
Minimized console windows are throttled **much harder** than unfocused ones.
Keep it:

- Restored
- On screen
- Just unfocused

What does NOT exist (so you don't waste time)

□ No registry key to disable console throttling
□ No group policy
□ No hidden cmd.exe option
□ No Windows setting to "lock focus performance"
□ No way to force foreground scheduling for background windows
Microsoft removed these hooks years ago.

Bottom line (clear answer)

If the Python script prints output, and you are restricted to Windows settings only, there is no way to make it always run at full speed when unfocused.

You can **reduce** the slowdown (steps above), but not eliminate it.

The single smallest exception (non-code, non-settings)

For completeness:

- **Redirecting output to a file**
  (even once, without touching the script)

python yourscript.py > out.log

This instantly gives **100% speed**, because the slowdown is 100% caused by console rendering.

Afterword

This guide set out to explain how Large Language Models are fine-tuned-how vast, general systems are shaped into tools that speak with a particular voice, follow instructions, respect constraints, and perform reliably in real settings. By the end of this guide, the mechanics may feel familiar: objectives, datasets, selection of hyperparameters, and deployment trade-offs. The deeper lesson, however, is not just technical. Fine-tuning is where abstraction meets intent. It is where we decide what we want these models to become.
At pretraining scale, language models are mirrors of the internet's patterns-brilliant, messy, and uneven. Fine-tuning is the act of craftsmanship. It selects, emphasizes, and corrects. Through carefully designed objectives and feedback, we turn a general predictor into a collaborator: a tutor that explains patiently, a system that follows rules, a model that refuses harmful requests, or a domain expert that speaks a specialized dialect. In that sense, fine-tuning is less about squeezing out a few more benchmark points and more about translating human values into gradients.

First, data is design.

Architectures matter, but the most consequential choices often hide in datasets: what is included, what is excluded, who labels it, and under what assumptions. Fine-tuning data encodes priorities-helpfulness versus brevity, caution versus creativity, precision versus coverage. Treating data curation as a first-class design activity is not optional; it is the difference between accidental behavior and intentional behavior.

Second, alignment is iterative, not final.

No fine-tuning pass "solves" alignment. Distribution shift, new use cases, and evolving social expectations guarantee that today's solution becomes tomorrow's baseline. Techniques like instruction tuning, preference optimization, and rule-based constraints are best understood as tools in a continuous feedback loop. Monitoring, red-teaming, and post-deployment updates are extensions of fine-tuning, not afterthoughts.

Third, evaluation defines reality.

Models optimize what we measure. If evaluations focus narrowly on accuracy or fluency, we will get systems that sound right even when they are wrong. If we measure robustness, calibration, refusal quality, and real-world task success, models move in those directions. Good evaluation is plural, adversarial, and contextual-and it must evolve alongside the models it judges.

Fourth, smaller changes can have outsized effects.

Compared to pretraining, fine-tuning often uses modest compute and limited data. Yet these small nudges can radically alter behavior. This leverage is powerful and dangerous. It enables rapid specialization and democratizes model adaptation, but it also lowers the barrier to misuse. The same methods that create a helpful assistant can create a persuasive manipulator. Responsible practice requires acknowledging that dual-use reality and building safeguards accordingly.

Looking forward, the role of fine-tuning is likely to expand rather than shrink. As foundation models stabilize, differentiation will increasingly come from how they are adapted: personalized systems that respect individual preferences, domain-specific models that encode professional norms, and collaborative tuning workflows where humans and models co-evolve. We can expect tighter integration between training and deployment, with online feedback, adaptive objectives, and model steering that happens continuously rather than in discrete phases.

There is also a cultural shift underway. Fine-tuning invites interdisciplinary thinking. It demands input from engineers, domain experts, ethicists, designers, and users themselves. The question is no longer just “Can the model do this?” but “Should it, under what conditions, and for whom?” Answering those questions well requires humility as much as technical skill.

If this guide has a quiet argument, it is this: fine-tuning is a moral as well as a mathematical act. Every loss function reflects a preference. Every constraint encodes a boundary. Every dataset carries a worldview. Being explicit about those choices-documenting them, testing them, and revisiting them-is part of the craft.
The field will move quickly from here. New objectives will appear, old methods will be reframed, and today’s best practices will be revised. That is healthy. What should remain constant is the mindset this book encourages: treat fine-tuning not as a mechanical finishing step, but as the place where responsibility enters the training loop.

If you carry that mindset forward-questioning your data, interrogating your evaluations, and tuning with intention-then the models you shape will not only perform better. They will behave better. And in the long run, that may be the most important optimization of all.