

First GPT in 2018

?

- Dataset: BooksCorpus (~7,000 unpublished books, 800M words).
- Idea: lots of long-form text → good for learning grammar, style, coherence.
- Model trained unsupervised (no labels, just raw text).

Transformer's Decoder + Dataset
→ GPT-1 (2018)

first LLM (Large lang. Model)

- A "Large Language Model" refers to a type of artificial intelligence model that is designed to understand and generate human-like text based on the input it receives. These models are typically trained on massive datasets containing a wide range of text from the internet, books, articles, and more. They use deep learning techniques, particularly transformer architectures, to process and generate text.

Generative Pre-trained Transformer 1 (GPT-1) was the first of [OpenAI](#)'s [large language models](#) following [Google](#)'s invention of the [transformer](#) architecture in 2017.^[2] In June 2018, [OpenAI](#) released a paper entitled "Improving Language Understanding by Generative Pre-Training",^[3] in which they introduced that initial model along with the general concept of a [generative pre-trained transformer](#).

From <<https://en.wikipedia.org/wiki/GPT-1>>

- The "transformer" is a specific type of neural network architecture that has revolutionized natural language processing (NLP) and a wide range of other machine learning tasks. It was introduced in the paper "**Attention Is All You Need**" in 2017.
- These models are pre-trained on vast datasets, which allows them to learn linguistic patterns, semantics, and context. Fine-tuning can then be applied on specific tasks or domains to make them even more specialized.
- These models can perform various NLP tasks, such as text generation, text completion, translation, summarization, question answering, and more. They have been used in a wide range of applications, including chatbots, language translation services, content generation, and even code generation.
- LLMs are also the foundational model when it is fine-tuned on our data means adding our own private data to LLMs.

eg: Qpt- 3.5 / 4 / 5
LLama- 3 , Gemini , Qpt- oss- 120B

- "Small language model" usually refers to a language model with fewer parameters compared to larger models. Example: Phi-1, Phi-2 , Lamini etc.

- **Reduced Model Size:** Small language models have fewer parameters (weights and neurons) compared to large models like GPT-3 or GPT-4.

- **Reduced Model Size:** Small language models consume less memory and computational resources

- **Resource Efficiency:** Resource is less so management is easy

- **Domain-Specific Use:** We can build domain specific small language model for our use such as legal , medical etc.

- **Customization:** Developers can customize small language models more easily, tailoring them to specific tasks and fine-tuning their performance as needed.

- **Fast Inference:** Smaller models typically require less computation during inference, which leads to faster response times.

Multimodality and Large Multimodal LLMs

- Multimodal LLMs are the language model which deals with text (translation, language modeling), image (object detection, image classification), or audio (speech recognition).
- These language model are used in the various field and help us to solve the problem of any sectors or field very easily.
- **Not all multimodal systems are LMMs**
- For example, **text-to-image models** like **Midjourney, Stable Diffusion, and Dall-E** are multimodal but don't have a language model component.
- **Multimodal can mean one or more of the following:**
 - Input and output are of different modalities (e.g. text-to-image, image-to-text)
 - Inputs are multimodal (e.g. a system that can process both text and images)
 - Outputs are multimodal (e.g. a system that can generate both text and images)
- Many use cases are impossible without multimodality, especially those in industries that deal with a mixture of data modalities such as healthcare, robotics, e-commerce, retail, gaming, etc.

Now we will learn how
to make domain-specific model
and jump from Spt-4 → Spt-5.

..... to be continued

How to use these models

as they are very large.

→ API's

Pre-training is the initial phase in building a machine learning model specially in large language models where the system is trained on massive amounts of raw, unlabeled data to learn general patterns, language structure and world knowledge. The objective of pre training is to give the model a broad foundational understanding so it can recognize patterns, relationships and context to perform well on a wide range of tasks. This foundation can later be fine tuned on specific datasets to adapt the model for targeted applications.

Pre Training

Pre-training is the initial stage where a model is trained on a large pre training dataset to learn general patterns and representations.

The process begins with feeding this dataset into a pretrained model which is connected to a projection head a layer that maps the learned features to the pre training objective such as predicting the next word or filling in missing tokens.

Once trained the knowledge from this pretrained model is transferred to the fine tuning phase where its weights are often frozen and a new projection head is added for specific prediction or classification tasks.

This setup ensures the model first learns broad reusable knowledge before being adapted to specialized objectives.

Why Pre-Training? (Motivations in GenAI)

Pre-training addresses core challenges in GenAI: scalability, generalization, and data efficiency. Here's why it's indispensable:

1. Scalability and Transfer Learning: GenAI thrives on "emergent abilities" that arise at scale (e.g., GPT-3's few-shot learning). Pre-training on huge datasets lets the model transfer knowledge to new tasks with minimal additional training, reducing the need for task-specific data. This is crucial for GenAI, where labeled data for creative tasks (e.g., "generate a poem about quantum physics") is scarce and subjective.

2. Unsupervised Learning Efficiency: Most internet-scale data is unlabeled, so pre-training leverages this "free" resource. In GenAI, it enables zero-shot or few-shot generation—e.g., prompting a pre-trained model to write code without explicit training on programming datasets.

3. Capturing World Knowledge and Patterns: GenAI needs implicit understanding of grammar, facts, causality, and creativity. Pre-training encodes this probabilistically, allowing models to generate diverse, contextually rich outputs. Without it, outputs would be repetitive or nonsensical.

4. Cost and Time Savings Long-Term: Initial pre-training is compute-intensive (e.g., GPT-3 used ~3.14e23 FLOPs), but it amortizes costs across applications. For GenAI, this enables rapid iteration on fine-tuning for domains like drug discovery (e.g., generating molecular structures) or art (e.g., style transfer).

5. Mitigating Catastrophic Forgetting: By starting broad, pre-training prevents models from overfitting to narrow data during later stages, preserving generative flexibility.

In essence, pre-training turns GenAI from a narrow tool into a versatile "creative engine," powering applications like content creation, simulation, and personalization.

https://colab.research.google.com/drive/1LEmL_fJGonVrRACEIHROd4a0Dda3fZgc?usp=sharing

1. What Is Fine-Tuning? (In Depth)

Fine-tuning is the process of taking a **pre-trained large language model (LLM)** — like GPT, BERT, or LLaMA — and *further training it on a smaller, task-specific dataset* so that it performs exceptionally well in that domain or style.

It's a form of **transfer learning**, meaning we *transfer* the general knowledge the model has already learned (like grammar, logic, world knowledge) to a *new, narrower context* (like medical conversations, legal contracts, or customer support chats).

2. The Process Behind Fine-Tuning

Step 1. Start with a Base Model

You begin with a model that's already trained on huge amounts of data — billions of words, code, and web pages.

This base model understands:

- General language patterns (syntax, grammar)
- Common reasoning structures
- Context management
- Knowledge across many domains

Example:

Base model = GPT-3.5 trained on general web data.

Step 2. Prepare Specialized Data

You then collect or curate a **domain-specific dataset**, which may include:

- Company-specific documents
- Legal contracts
- Medical reports
- Customer chat logs
- Product manuals
- Brand-specific tone/style examples

The data must be **clean, labeled** (if needed), and **consistent** with the task — e.g. Q&A pairs, dialogues, summaries, etc.

Example:

If you want a chatbot for a **law firm**, your dataset might include:

- Legal FAQs
- Case summaries
- Statute interpretations
- Legal writing samples

Step 3. Train the Model on Specialized Data

Now, you continue training the model — but *not from scratch*.

The fine-tuning process **adjusts only a subset** of the model's internal parameters (weights and biases) to better fit your new dataset.

During this:

- The **learning rate** is kept small to avoid “forgetting” previous knowledge.
- The model learns **contextual nuances, tone, and task-specific reasoning**.

 Think of it like:

Teaching a doctor (who already knows biology) how to specialize in cardiology — you don't start from ABCs, you just refine the expertise.

Step 4. Model Weight Adjustment

Internally, a neural network has **millions (or billions)** of parameters.

Fine-tuning updates these parameters slightly so that:

- The model becomes more confident about answers in the target domain.
- Irrelevant general patterns are down-weighted.
- Relevant domain patterns (like legal logic or brand tone) are strengthened.

This step ensures that the **loss function** (difference between model prediction and actual answer) is minimized for the new data.

Step 5. Result — A Specialized Model

After fine-tuning, you now have:

A version of the base model specialized for your purpose.

It can:

- Use the correct terminology (e.g. “habeas corpus” instead of “freedom request”)
- Follow brand tone (“friendly and concise” or “formal and academic”)
- Perform better on specific tasks (e.g. medical diagnosis summaries, code generation, etc.)

3. Benefits of Fine-Tuning (Deep Explanation)

Benefit	Explanation
Specialization	The model becomes an expert in one area — legal, medical, educational, technical, etc.
Higher Accuracy	Fine-tuned models outperform general-purpose ones on domain-specific benchmarks.
Efficiency	Much faster and cheaper than training from scratch since the base model already knows general language.
Customization	The output can be aligned with your tone, style, or brand voice.
Smaller Data Needs	You often need only a few thousand high-quality examples rather than millions.

4. When Fine-Tuning Is Useful

1. **Limited Domain Data** — when your specialized data is small but important.
2. **Brand Style Alignment** — you want your AI to “sound” like your company.
3. **Specific Tasks** — e.g., summarizing financial reports, writing product descriptions, or generating code in your internal format.
4. **Regulated Domains** — like healthcare, finance, or law where accuracy and terminology matter.

5. Alternatives to Full Fine-Tuning

Sometimes full fine-tuning (training all weights) is too expensive.

There are lighter, more efficient methods called **parameter-efficient fine-tuning (PEFT)** techniques:

a. LoRA (Low-Rank Adaptation)

- Adds small “adapter” layers instead of retraining all weights.
- Much faster and cheaper.

b. QLoRA

- Quantized LoRA; it reduces memory usage by using 4-bit precision weights.
- Perfect for running on smaller GPUs.

https://colab.research.google.com/drive/13TH4kOG9VMFHe1fAnGG6_00pr_t2_X-t?usp=sharing

This is a very important concept if you're working on **fine-tuning large generative AI models** (like GPT, LLaMA, or Falcon) where **GPU cost, time, and memory** become huge challenges.

1. What Is PEFT (Parameter-Efficient Fine-Tuning)?

PEFT stands for **Parameter-Efficient Fine-Tuning**.

It's a method that lets you fine-tune **only a small portion** of a pre-trained large model, instead of updating **all** its parameters.

Traditional Fine-Tuning vs. PEFT

Type	Parameters Trained	GPU Memory	Training Cost	Example
Full Fine-Tuning	100% of model parameters	Very high	Very expensive	Updating all 7B+ weights
PEFT	Only ~0.1%–5% of parameters	Very low	Very cheap	Updating only adapter layers or prompts

Why PEFT?

Modern LLMs have **billions of parameters (7B, 13B, 70B, etc.)**.

Training or even fine-tuning them fully is:

-  **Memory-heavy** (hundreds of GB of VRAM)
-  **Expensive** (takes multiple high-end GPUs)
-  **Slow** (takes days or weeks)
-  **Risky** (can cause catastrophic forgetting)

So PEFT was developed to make fine-tuning:

- **Lightweight**
- **Affordable**
- **Modular**
- **Fast**



1. What Is LoRA (Low-Rank Adaptation)?

LoRA stands for **Low-Rank Adaptation** of Large Language Models.

It's a **parameter-efficient fine-tuning (PEFT)** technique introduced by *Microsoft Research in 2021* to make fine-tuning huge models like GPT, BERT, or LLaMA possible on normal GPUs — without updating all their billions of parameters.

In simple words:

LoRA teaches a large model new knowledge or skills by learning small “adapter” layers instead of retraining the whole brain.

2. Why Do We Need LoRA?

Let's first understand the problem LoRA solves:

Modern models like:

- **GPT-3** → 175 billion parameters
- **LLaMA 2** → 13–70 billion parameters
- **Falcon** → 40 billion parameters

Fine-tuning such massive models requires:

- **Terabytes** of memory
- **Hundreds of GPUs**
- **Millions of dollars**

Even **storing** multiple fine-tuned versions becomes impractical.

Example:

If a base model = 70B parameters (280 GB storage),

10 domain fine-tuned models = 2.8 **TB** just for weights 😱

So LoRA provides a **clever mathematical shortcut** — it learns small, efficient “update modules” without touching the main model.



3. The Core Idea of LoRA

In neural networks, every transformer layer has large **weight matrices** — say W — that transform input features.

During normal fine-tuning:

- You adjust all weights W → billions of parameters.

In LoRA:

- You **freeze** the original weights W_0 (no updates).
- You **insert two small trainable matrices**, A and B , such that:

$$\Delta W = B \times A$$

- This product ΔW represents a **low-rank update** to the original weights.

The effective weight during training becomes:

$$W = W_0 + \Delta W = W_0 + B \times A$$

Here's what happens conceptually:

- A and B are *tiny* compared to W_0 .
- Only A and B are trained.
- The base model (W_0) stays frozen.

Low-Rank Explained (Intuitively)

Matrix W might be $10,000 \times 10,000 = 100$ million parameters.

LoRA says:

“Hey, instead of updating all 100M parameters, let’s represent the change (ΔW) as two smaller matrices.”

If we pick a rank $r = 8$,

then $A = 10,000 \times 8$,

and $B = 8 \times 10,000$.

That’s only **160,000 parameters — 625x smaller!** 🔥

So you learn a small correction to the large weight — enough to specialize the model, without heavy compute or storage.

4. LoRA Architecture in Transformers

LoRA is usually applied to:

- The **query (Q)** and **value (V)** projection matrices inside each transformer’s attention layer.

Workflow:

1. Input → Attention layer
2. Attention uses W_Q and W_V weights
3. LoRA injects adapters (A & B) for these matrices
4. Forward pass uses $W_Q + \Delta W_Q$ and $W_V + \Delta W_V$
5. During backpropagation, **only LoRA parameters are updated**



5. LoRA Training Flow (Step-by-Step)

Step	Description
1. Start with base LLM	Load a pre-trained model (e.g., LLaMA-2, GPT-J). Freeze all original weights.
2. Insert LoRA adapters	Add small trainable matrices (A, B) in target layers (e.g., Q and V).
3. Feed domain data	Provide your specialized dataset (e.g., medical, legal, chat style).
4. Train only adapters	Update only A and B matrices; rest of the model stays constant.
5. Merge or deploy	Combine base model + LoRA weights for inference, or keep them separate for modularity.

6. Advantages of LoRA

Benefit	Description
 Memory Efficient	Reduces trainable parameters by 100–1000×.
 Fast Training	Updates only a few matrices — much quicker.
 Cost-Effective	Fine-tune 7B–70B models on consumer GPUs (e.g., 24GB).
 Modular	Store and load small adapters (~30–100MB) for each task/domain.
 No Forgetting	Base model knowledge remains intact.
 Mergeable	You can merge adapters into the base model if desired, or load multiple adapters dynamically.

Let's say we have a **7B LLaMA model** and want to adapt it for:

- **Medical Q&A**
- **Legal document summarization**
- **Customer support chat**

We can:

1. Train **three LoRA adapters**, one per domain.
2. Store each adapter separately (each ~50 MB).
3. Load whichever adapter we need at runtime.

So instead of storing 3 full fine-tuned models (~90 GB each),
we store **one 90 GB base model + 3 × 50 MB adapters = 90.15 GB total!**

Hyperparameters

Parameter	Description	Typical Value
r (Rank)	Size of the low-rank matrices (controls how much you learn)	4, 8, 16
α (Scaling)	Scaling factor for LoRA updates	8–32
Target Modules	Layers where LoRA is applied	query, value, key, or dense
Dropout	Prevents overfitting	0.05–0.1

Higher rank = more capacity (but more memory).

Lower rank = more efficiency (but slightly less precision).

What Is QLoRA?

QLoRA stands for **Quantized Low-Rank Adaptation** — it's an **advanced version of LoRA** that makes fine-tuning **large pre-trained language models** both **memory-efficient** and **highly accurate**.

It was introduced in 2023 by **Tim Dettmers et al. (University of Washington)** in the paper:

"QLoRA: Efficient Finetuning of Quantized LLMs"



Core Idea:

QLoRA combines **quantization** (reducing model precision to save memory) with **LoRA adapters** (small trainable modules) to fine-tune giant models using very little GPU memory — without sacrificing performance.

2. Why Do We Need QLoRA?

Even though **LoRA** was efficient, it still required loading the **base model in full precision (16-bit or 32-bit)**.

That meant:

- 13B model → ~26 GB VRAM
- 33B model → ~66 GB VRAM
- 65B model → ~130 GB VRAM

Even with LoRA adapters, you **couldn't fine-tune** such big models on standard GPUs.



QLoRA solves this by:

1. Compressing (quantizing) the base model to **4-bit precision**.
2. Keeping it frozen (no updates).
3. Training **tiny LoRA adapters** on top of it.

This makes it possible to fine-tune a **33B model on a single 24GB GPU (like an RTX 3090)**!

Step 1: Quantization (Reducing Precision)

Quantization = representing numbers with fewer bits.

Example:

- Normal model weights use **16-bit (FP16)** or **32-bit (FP32)** floats.
- QLoRA compresses them to **4-bit integers (Int4)** using a smart algorithm called **NF4 (NormalFloat 4-bit)**.

So instead of storing each weight with 16 bits, we store it with only 4 bits — cutting memory usage by **75%**.

But won't that reduce accuracy?

- Not much — because QLoRA uses **Quantization with Dequantization**:
- The quantized weights are stored in 4-bit form.
- During computation, they're temporarily **dequantized** into 16-bit form for matrix operations.
- The result is accurate *and* memory-efficient.

Step 2: LoRA Adapters on Quantized Model

Once the base model is quantized and frozen:

- We attach **LoRA adapters** (small trainable matrices A and B) to specific layers (usually query and value projection matrices).
- These adapters are trained in **16-bit precision (BF16)**.
- The quantized weights remain untouched and frozen.

So, training happens on LoRA adapters, while the base model serves as a **reference backbone**.

Mathematically:

$$W_{\text{effective}} = W_{\text{quantized}} + B \times A$$

where:

- $W_{\text{quantized}}$ \rightarrow 4-bit frozen base weights
- A, B \rightarrow small trainable low-rank matrices



4. Memory Efficiency Example

Let's see real numbers 👇

Model	Full Precision (FP16)	LoRA (16-bit)	QLoRA (4-bit)
7B	14 GB	14 GB	4 GB
13B	26 GB	26 GB	8 GB
33B	66 GB	66 GB	20 GB
65B	130 GB	130 GB	38 GB

🔥 This means:

You can fine-tune a 33B parameter model on a single 24GB GPU — something impossible before QLoRA.

Step	Description
1 Load Model	Load base model (e.g., LLaMA-2 33B) in 4-bit quantized form using NF4.
2 Freeze Parameters	Keep all quantized weights frozen — they're not updated.
3 Add LoRA Adapters	Attach small low-rank matrices (A & B) to key attention layers.
4 Train Adapters Only	Only LoRA adapters are updated; base model stays static.
5 Use Paged Optimizer	Keeps training memory within limits.
6 Save LoRA Checkpoint	Only adapter weights (~50–200MB) are stored — not full model.

8. Advantages of QLoRA

Advantage	Explanation
 Ultra Memory Efficient	Uses 4-bit quantization → 75% less memory.
 Cost-Effective	Fine-tunes massive models on a single GPU.
 Fast Training	Less data transfer + fewer trainable parameters.
 Safe Fine-Tuning	Prevents catastrophic forgetting of base knowledge.
 Modular Adapters	One model → multiple domain adapters.
 High Accuracy	Nearly same performance as full precision fine-tuning.

9. Real-World Example

Let's take an example from the **Guanaco** project (built using QLoRA):

Model	Fine-Tuned Using	GPU Used	Result
LLaMA 7B	QLoRA	1× RTX 3090	Matched ChatGPT quality (52.3% Vicuna benchmark)
LLaMA 33B	QLoRA	1× A100	Outperformed ChatGPT (97.8% Vicuna benchmark)

This was a **game-changer** — it proved that **you don't need massive clusters** to fine-tune world-class AI models.

When to Use QLoRA

Use QLoRA when:

- You have **limited GPU resources**.
- You want to fine-tune **large models (13B–70B)**.
- You need **domain adaptation** (legal, healthcare, education).
- You need **multiple adapters** but want to save storage.
- You want **ChatGPT-quality results** without enterprise infrastructure.

https://colab.research.google.com/drive/1NcOzf-EJjLpFhU9Q32qMVgsa_FjdV_xT?usp=sharing

ADDITIONAL BONUS

Technique	What It Trains	Typical Accuracy	Memory Use	Best For
LoRA	Adds low-rank adapters	★★★★★	Low	Most domains
QLoRA	LoRA + Quantization	★★★★★	Very Low	Very large models
Prefix Tuning	Trains prompt vectors	★★★★☆	Very Low	Style or tone control
Adapter Tuning	Adds feed-forward layers	★★★★☆	Medium	Multi-task models
BitFit	Only bias terms	★★☆☆☆	Minimal	Low-resource tuning

You're Using APIs — Not Training Models

You are **not training or fine-tuning** huge LLMs like GPT, LLaMA, or Claude from scratch.

Instead, you're using **pre-trained and hosted models** provided by **cloud companies** such as:

-  **Azure OpenAI Service** (Microsoft)
-  **AWS Bedrock / SageMaker JumpStart**
-  **Google Vertex AI**

These platforms provide **ready-to-use generative AI models** (like GPT-4, Claude 3, LLaMA 3, etc.) through **API keys**.

Reason



Cost Efficiency



No Infrastructure Needed



Time-Saving



Security and Compliance



Scalability



Continuous Updates

Explanation

Training/fine-tuning LLMs locally requires massive GPUs (e.g., A100/H100 clusters). Using APIs saves thousands of dollars.

You don't need to manage GPUs, clusters, or model deployments. Azure and AWS handle everything.

You can start generating results instantly without weeks of training.

Azure and AWS follow enterprise-grade security standards (GDPR, ISO, SOC2). Perfect for production.

APIs scale automatically for multiple users or high workloads — ideal for your EzSync platform or Tech Tomorrow projects.

Cloud providers update models regularly (e.g., GPT-4 Turbo → GPT-5), so you always access the latest performance.

Why You're Using Cloud APIs (Instead of Training Locally)

```
from openai import AzureOpenAI

client = AzureOpenAI(
    azure_endpoint="https://<your-endpoint>.openai.azure.com/",
    api_key=<your-api-key>,
    api_version="2024-05-01-preview"
)

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "You are an AI assistant for EzSync."},
        {"role": "user", "content": "Explain fine-tuning in simple words."}
    ]
)

print(response.choices[0].message.content)
```

What is LangChain?

LangChain is a **framework** that helps developers build powerful **applications using Large Language Models (LLMs)** — like GPT, Claude, Gemini, or Mistral — **in a structured, modular, and production-ready way**.

Think of LangChain as the "**middleware**" that connects:

- The **LLM (like GPT-4 or Mistral)**
- Your **data (documents, PDFs, databases, APIs, etc.)**
- And your **application logic (chatbot, assistant, agent, etc.)**

Why LangChain Exists

LLMs are great, but on their own they:

- Can't **remember context** across multiple user interactions
- Don't have **direct access to your data** (like PDFs or SQL databases)
- Don't know **how to take actions** (like calling APIs or searching a knowledge base)

So LangChain was built to solve these gaps by providing:

- **Memory**
- **Knowledge Retrieval**
- **Tool / API Integration**
- **Prompt Management**
- **Model Switching**

Feature	Without LangChain	With LangChain
Model Switching	Manual API handling	One-line change
Data Access	Hard to implement	Built-in RAG tools
Conversation Memory	Must code manually	Plug-and-play memory
Workflow Logic	Messy scripts	Clean modular Chains
Agents & Tools	Manual integration	Prebuilt agent system

