

Large Language Models: Architectures, Training, Embeddings, and Evaluation

Large language models (LLMs) use Transformer-based architectures that vary in design and scale. They are pretrained on vast corpora and fine-tuned with various strategies (SFT, RLHF, etc.), often using parameter-efficient methods. Their tokenization, embedding, and retrieval (RAG) pipelines rely on subword algorithms and vector databases. Evaluation involves metrics like perplexity, BLEU/ROUGE, and benchmarks (MMLU, HellaSwag), along with system metrics (FLOPs, latency, throughput). Below we detail each aspect with references to current literature and tools, culminating in comparison tables.

1. Model Architectures and Technologies

- **Transformer Flavors:** LLMs typically use one of three architectures: **decoder-only** (autoregressive, e.g. GPT-style), **encoder-only** (bidirectional for classification, e.g. BERT), or **encoder-decoder** (seq2seq, e.g. T5, BART) models ¹. Decoder-only models predict one token at a time for generation tasks; encoder-only models encode context holistically. Encoder-decoder models (like T5) use a separate encoder and decoder with cross-attention.
- **Attention Variants:** The original Transformers use full self-attention, with complexity $\$O(n^2)$. To handle longer context or improve efficiency, **sparse attention** schemes are used. For example, **local (sliding-window) attention** (as in Longformer, Mistral) restricts each token to attend to a fixed window of width w , reducing complexity to $\$O(nw)$ ² ³. In Figure below, sliding-window attention (middle) attends only to nearby tokens, drastically reducing computation. Specialized methods include *LSH attention* (Reformer) which hashes tokens to limit pairwise comparisons, and *block-sparse attention* (BigBird).

Figure: Attention patterns. Full attention (left) connects all tokens, while sliding-window sparse attention (middle) limits each token to w neighbors, reducing complexity to $\$O(nw)$ ³.

- **ALiBi and Positional Encodings:** Transformers need positional information. Instead of learned embeddings, **ALiBi (Attention with Linear Biases)** adds a fixed, trainable bias that linearly increases with token distance, biasing attention toward nearby tokens ⁴. ALiBi thus injects a locality bias and can extrapolate to longer sequences without additional parameters. Alternatively, **sinusoidal embeddings** (from Vaswani et al.) encode position pos with functions $\sin(pos/10000^{2i/d})$ and $\cos(pos/10000^{2i/d})$ ⁵. A newer approach is **Rotary Positional Embeddings (RoPE)**, which rotates pairs of dimensions in the query/key vectors by an angle proportional to position, encoding relative position continuously ⁶. As shown below, RoPE preserves the dot-product structure while encoding distances.

Figure: ALiBi adds distance-based bias to attention (bias matrix on right) ⁴.

Figure: RoPE encodes position by rotating vector components by position-dependent angles ⁶.

- **Multi-Query and Grouped-Query Attention:** To improve inference efficiency, some LLMs use **Multi-Query Attention (MQA)** or **Grouped-Query Attention (GQA)**. In MQA, all heads share a single set of key/value projections (one K/V pair for all heads), drastically reducing memory (only one KV cache) ⁷. This yields faster decoding at some quality cost. GQA is intermediate: heads are divided into a small number of groups, each group sharing keys/values ⁸ ⁹. For example, Llama2-70B and Mistral-7B use GQA (8 KV heads) to balance speed and accuracy ⁸. In summary, standard multi-head (each head independent) \Rightarrow GQA (groups of heads share KV) \Rightarrow MQA (single KV for all heads) ⁸.
- **Mixture-of-Experts (MoE):** MoE models (e.g. GShard, Switch Transformer, GLaM) replace dense feedforward layers with multiple *expert* sub-networks and a trainable routing mechanism. Each token is routed to a subset of experts, increasing capacity. This yields faster pretraining and inference *per parameter* ¹⁰. For instance, MoE achieves target quality with less training compute than a same-sized dense model ¹⁰. However, MoEs load all experts on device (high memory) and are harder to fine-tune ¹¹.
- **Model Scale and Layering:** LLM performance generally scales with model size and data. GPT-3 (175B) uses 96 transformer layers, hidden size 12,288, and 96 heads ¹². Smaller versions (e.g. 125M) had 12 layers and 12 heads ¹². Llama2 offers 7B, 13B, and 70B models: e.g. Llama2-70B uses 80 layers, 64 heads, hidden size 8192 ¹³ ¹⁴. Mistral-7B has 32 layers, 32 heads (KV heads=8), hidden size 4096 ¹⁵. As a rule, *deeper/wider models* often yield higher accuracy (per scaling laws ¹⁶) but require quadratically more compute and memory. Table 1 (below) compares select models' architecture and scale.

2. Training Strategies

- **Pretraining Data & Sampling:** LLMs are pretrained on massive corpora (web text, books, code, etc.). For example, GPT-3 was trained on 300B tokens from CommonCrawl, WebText2, books, and Wikipedia ¹⁷. Data is often filtered and balanced; some use a curriculum or temperature-based sampling. Diversity of data (code, multilingual text) can improve generality.
- **Supervised Fine-Tuning (SFT):** After pretraining, models are fine-tuned on labeled datasets (e.g. question-answer pairs, instructions). **Instruction tuning** fine-tunes on tasks phrased as instructions, improving obedience to user prompts. For instance, Llama2-Chat variants are SFT-tuned on dialogue and instructions ¹⁸.
- **Reinforcement Learning from Human Feedback (RLHF):** RLHF (using algorithms like Proximal Policy Optimization, PPO) further refines the model by optimizing for human preferences. A typical RLHF pipeline: (1) Collect human comparisons of model outputs; (2) Train a reward model; (3) Use PPO to align the LLM with the reward model. This yields more helpful and safe outputs. Llama2-Chat, ChatGPT, and Anthropic's Claude are trained with RLHF (sometimes variants like DPO) ¹⁸ ¹.
- **Parameter-Efficient Fine-Tuning (PEFT):** Methods like **LoRA** (Low-Rank Adaptation) and adapters freeze most of the model and train only small additional parameters. LoRA decomposes weight

updates into low-rank matrices (A and B) and injects them into attention layers ¹⁹. This reduces trainable parameters dramatically: only the small rank matrices are updated ²⁰. Performance of LoRA-tuned models rivals full fine-tuning, and the adapters can be merged at inference with no extra latency ²⁰. **Adapters** add small feedforward layers after each block (demonstrating near full-performance with few params) ²¹. **Prefix tuning** inserts trainable “prefix” vectors into each layer’s hidden states; it tunes $\approx 0.1\%$ of parameters yet matches full fine-tuning on large models ²². **QLoRA** extends LoRA by quantizing the base model (e.g. 4/8-bit) to reduce memory before fine-tuning.

- **Libraries and Tools:** Common frameworks include Hugging Face Transformers and the [PEFT](#) library for LoRA/prefix, [DeepSpeed](#) and [Accelerate] for efficient distributed training, [bitsandbytes](#) for 4/8-bit quantized training, and specialized libraries like [TRL \(HuggingFace\)](#) for RLHF ¹, [Axolotl](#) (open-source RLHF/SFT tooling), and [LoRA/QLoRA tutorials](#) for step-by-step fine-tuning. These tools handle tasks like memory-efficient optimization and mixed-precision training, crucial for large models.

3. Embedding and Token Mechanics

- **Tokenization:** Text is first split into tokens. Common methods include **Byte-Pair Encoding (BPE)** and **WordPiece**, which iteratively merge frequent subword units. **SentencePiece** (used by T5) supports BPE or **Unigram** algorithms. Many LLMs use **byte-level BPE** (GPT-2/3 style) where the vocabulary is defined over bytes, ensuring any text is representable. WordPiece (BERT) and Unigram (XLNet) also exist. Each method balances vocabulary size against expressiveness; e.g. BPE merges rare character combos into tokens, reducing sequence length. (See HuggingFace summary of tokenizers ²³.)
- **Embedding Vectors:** Each token index is mapped to a dense vector via a learned embedding matrix $\mathbb{R}^{|V| \times d}$ (where $|V|$ is vocab size). A positional embedding (learned or fixed) is added to capture order. For sinusoidal embeddings, for position p and dimension i : $\text{PE}(p, i) = \sin(\frac{p}{10000^{2i/d}})$. These are summed elementwise with token embeddings, giving input $x = E_{\text{text}}(token) + E_{\text{position}}$ to the Transformer.
- **Contextual Output Embeddings:** After all Transformer layers, each token has a final hidden vector. For classification or sequence tasks, a **pooling** strategy is applied. BERT-style models use a special **[CLS]** token prepended; its final hidden state is taken as a summary of the sequence ²⁴. (Original BERT: “The final hidden state of this [CLS] token is used as the aggregate representation” ²⁴.) Alternatively, one can **mean-pool** or **max-pool** over all token vectors, often giving robust sentence embeddings for retrieval. For decoder models (GPT), the output vectors are typically passed through a linear + softmax to predict next-token probabilities.
- **Retrieval-Augmented Generation (RAG):** In RAG setups, external knowledge is retrieved based on embeddings. Documents (or chunks) are embedded (e.g. by the same model or specialized sentence embedder) and indexed in a vector store (FAISS, Milvus, Weaviate, Qdrant, etc.). A query is similarly embedded and nearest neighbors retrieved. The retrieved pieces are then fed (as context) to the LLM. In practice: “*a user’s question is turned into an embedding and used to search a knowledge base (vector store) for semantically relevant chunks. These ‘retrieved facts’ are then prepended to the LLM’s input*” ²⁵. Vector DBs enable fast semantic search by measuring cosine or dot-product similarity between high-dimensional embeddings ²⁵.

4. Evaluation and Comparison Metrics

- **Parameter Count & FLOPs:** Model size and compute are quantified by parameter count and FLOPs. A Transformer with embedding dimension E , vocab V , max length P , and L layers has approximate parameter count ²⁶ :
$$C = E(V + P) + L(12E^2 + 13E) + 2E,$$
 where the $12E^2$ term comes from attention and MLP weights and biases ²⁶. FLOPs per token (for inference) scale roughly as $O(L \cdot E^2)$: each layer does attention ($\approx 2E^2$ ops per token) plus feed-forward ($\approx 2E^2$), etc. Training FLOPs depend on sequence length N and batch size. Memory needs scale with C (model weights) plus intermediate activations ($\sim O(N, E)$ per layer).
- **Perplexity:** A common metric for language modeling is **perplexity (PPL)**, which measures how well the model predicts a test set. It is defined as the exponentiated average negative log-likelihood:
$$\text{Perplexity} = \exp\left(-\frac{1}{N} \sum_{i=1}^N \log P(x_i)\right),$$
 where $P(x_i)$ is the model's predicted probability of token x_i ²⁷. Lower perplexity indicates better predictive fit. However, perplexity alone does not capture answer quality or reasoning ability.
- **Text Generation Metrics:** For tasks like translation or summarization, **BLEU** and **ROUGE** scores measure overlap with reference texts. BLEU (0-1) counts matching n-grams (often up to 4-grams) between generated and reference, rewarding precision ²⁸. ROUGE (especially ROUGE-L) emphasizes recall of reference n-grams and longest common subsequences, used in summarization ²⁸. These metrics can be misleading for creative or semantic tasks, but remain standard.
- **Benchmarks (MMLU, HellaSwag, etc.):** LLMs are evaluated on aggregate benchmarks. For example, **MMLU** (Massive Multitask Language Understanding) consists of 57 multiple-choice questions spanning academics; it tests factual and reasoning knowledge. **HellaSwag** tests commonsense inference (choosing the most plausible continuation). Other benchmarks include GLUE/SuperGLUE for NLU, and truthfulness tests like TruthfulQA. As noted, “*Models are often evaluated against GLUE, SuperGLUE, HellaSwag, TruthfulQA, and MMLU*” ²⁹.
- **Latency and Throughput:** Inference performance is measured by **latency** (time to generate a token or response) and **throughput** (tokens per second). These depend on hardware and model optimizations (e.g. quantization, FlashAttention). For instance, using multi-query attention or smaller KV cache can reduce latency ⁷. Throughput benchmarks often report how many tokens/sec or requests/sec a model can handle on given GPUs or CPUs.
- **Using Metrics for Comparison:** When selecting a model or tuning it, these metrics guide decisions. A model with lower perplexity or higher benchmark scores is generally stronger (but may cost more compute). BLEU/ROUGE help in evaluation for generation tasks. Latency/throughput help choose models suitable for real-time deployment. For fine-tuning, one might aim to reduce perplexity on domain data or improve benchmark scores without excessive training cost.

5. Running and Fine-Tuning Models Locally

- **Local Deployment Tools:** To run LLMs on local hardware, several tools exist. **Hugging Face Transformers** provides pipelines (`pipeline("text-generation", model=...)`) to load many models in PyTorch. **Ollama** is a user-friendly CLI/GUI for local LLMs: it lets you `ollama pull llama3.2`, run chat interfaces, and customize models via a “Modelfile” syntax ³⁰. Ollama emphasizes simplicity and privacy (all computation stays on your machine) ³⁰. In contrast, **vLLM** is a high-performance serving library by UC Berkeley: it uses techniques like **PagedAttention** (paged KV cache) and continuous batching to maximize throughput ³¹. As an example, vLLM reports several-fold speedups: e.g. “2.7x higher throughput and 5x faster per-token” on certain Llama-8B and Llama-70B runs ³².
- **Inference Servers:** Hugging Face’s **Text Generation Inference (TGI)** is a production-ready server for LLMs. It supports Tensor and pipeline parallelism, streaming outputs, and optimizations like FlashAttention and PagedAttention ³³ ³⁴. TGI also integrates quantization via bitsandbytes and GPT-Q ³⁴, allowing 4-bit inference for large models. **LMDeploy** (by InternLM/MMRazor) is another toolkit for serving/compressing LLMs, claiming ~1.8x higher throughput than vLLM via continuous batching and blocked KV caches ³⁵. It supports weight-only and KV quantization (4-bit inference 2.4x faster than FP16 ³⁶). Other options include **Llama.cpp** or **Mistral.rs** for CPU inference, NVIDIA’s TensorRT-LLM for GPUs, and managed endpoints (Hugging Face Inference Endpoints).

- **Fine-Tuning Workflow:** To fine-tune on a custom dataset:

- **Prepare Data:** Collect domain-specific Q&A or prompt/response pairs; format as JSON or text.
- **Tokenize and Dataset:** Use the model’s tokenizer to encode inputs/outputs, creating a HuggingFace Dataset.
- **Load Pretrained Model:** Choose an open model (e.g. Llama 7B/13B, Mistral-7B). Possibly apply quantization via bitsandbytes (load in 4-bit or 8-bit) to save memory ³⁴.
- **Apply PEFT:** Use the PEFT library (Hugging Face PEFT or Accelerate) to wrap the model with LoRA or adapters. Configure rank (\$r\$), target layers, etc.
- **Training:** Use a Trainer or custom loop. With LoRA, only adapter weights train (base model frozen). Use mixed precision (bfloating16/FP16) and gradient accumulation as needed. Tools like DeepSpeed’s ZeRO or Hugging Face Accelerate help utilize multi-GPU or offload.
- **(Optional) RLHF:** If adding RLHF, collect preference data, train a reward model, and use TRL/PPO to further fine-tune.
- **Checkpointing and Merging:** After training, you can merge LoRA weights into the base model (PEFT `merge_and_unload()`) for fast inference ²⁰. Save the tuned model and track hyperparameters.
- **Evaluation of Tuned Models:** Use evaluation harnesses like EleutherAI’s [LM_Eval_Harness](#) or [EleutherAI_Evals](#) to benchmark the model on standard tasks. For custom tasks, compute appropriate metrics (e.g. perplexity on held-out domain data, or accuracy/F1 on label sets). The Hugging Face `evaluate` library and frameworks like [Elyra](#) or Weights & Biases can log metrics over experiments.
- **Experiment Tracking and Versioning:** Good practice is to version datasets and code (e.g. git, DVC). Log experiments with tools like MLflow or Weights & Biases. Keep notes of model versions, PEFT configurations, and evaluation results. This helps compare runs and roll back if needed.

6. Comprehensive Comparison Table

Table 1: Model Architectures and Sizes. Major LLMs vary in architecture (dec/enc), size, layers, and attention features. Numbers are approximate or as reported:

Model	Type	Params	Layers	Hidden dim	Heads (KV Heads)	Positional	Key Features	Citations
GPT-3	Dec-only	175B	96	12288	96 (96)	Learned abs.	Large autoregressive; extensive web pretraining	12 .
BERT (base)	Enc-only	110M	12	768	12 (12)	Learned abs.	Bidirectional encoder; [CLS] for classification.	
T5-11B	Enc-Dec (†)	~11B	24+24	4096	64 (64)	Learned abs.	Seq2seq; multi-task; large-scale corpora.	
LLaMA-2 7B	Dec-only	7B	32	4096	32 (32)	RoPE?	4K context; base model (no GQA)	37 38 38 .
LLaMA-2 13B	Dec-only	13B	32	4096	32 (32)	RoPE?	SFT/RLHF variants; same arch as 7B.	37 38
LLaMA-2 70B	Dec-only	70B	80	8192	64 (8)	RoPE?	Uses GQA (64→8 KV heads) for speed	13 14 13 14 .
Falcon-180B	Enc-Dec (†)	180B	80+80	12,288	96 (96)	Rotary?	High performance; public weights; Bloom-architecture.	
Mistral-7B	Dec-only	7B	32	4096	32 (8)	RoPE (8K)?	Sliding-window 8K context; GQA (32→8 KV heads)	39 15 39 15 .

Model	Type	Params	Layers	Hidden dim	Heads (KV Heads)	Positional	Key Features	Citations
Mixtral-8x7B	Dec-only MoE	7B×8	32	4096	32 (8) each	RoPE (8K)	8-way MoE; sliding window 8K; each expert 7B (total 56B) ³⁹ .	³⁹ .
Gemini 1.0	Dec-only	(?), ~130B	-	-	-	-	Google's closed-model; supports multimodal input (VL).	

(*) Falcon and Gemini are encoder-decoder and proprietary models, respectively. RoPE indicates Rotary encodings; ALiBi not explicitly disclosed for all.

Table 2: Fine-Tuning & Inference Techniques. Parameter-efficient methods and serving tools differ in purpose and trade-offs:

Method/Tool	Category	Trainable params	Inference overhead	Notes	Citations
LoRA	PEFT (adapter)	Low (rank \$r\$)	None after merge	Add low-rank weight updates; freezes base model. Comparable to full FT ²⁰ .	²⁰
QLoRA	PEFT + quant.	Low	None after merge (w bnb)	Quantize base to 4/8-bit then apply LoRA. Saves >90% memory during FT.	
Prefix Tuning	PEFT (prompt)	Very low (0.1%)	Small (prefix tokens)	Insert trainable vectors in each layer's input; ≈0.1% params, near full-FT ²² .	²²
Adapters	PEFT (layers)	Low	None after merge	Small networks after each block; achieves near full-FT performance ²¹ .	²¹

Method/Tool	Category	Trainable params	Inference overhead	Notes	Citations
HuggingFace TGI	Inference server	-	Supports streaming, batch	High-performance LLM server; uses FlashAttention, PagedAttention, bitsandbytes quant. ³³ ³⁴ .	³³ ³⁴
Ollama	Inference tool	-	Fully local; easy UI	CLI for downloading/running models locally; emphasizes simplicity and privacy ³⁰ .	³⁰
vLLM	Inference lib	-	High throughput (PagedAttention)	Library for low-latency high-throughput inference; uses paged KV cache and dynamic batching ³¹ .	³¹
LMDeploy	Inference lib	-	4-bit quant with KV quant	Toolkit for serving/compressing LLMs; 1.8x throughput vs vLLM; supports AWQ quantization ³⁵ ³⁶ .	³⁵ ³⁶

(“–” indicates not a fine-tunable module; trainable params refer to what is learned during FT.)

Each model or method above is suited to different needs. Larger dense models (GPT-3, Llama-70B) yield higher accuracy but require massive compute; MoE models (Mixtral) offer larger capacity more efficiently. PEFT methods trade a small fraction of parameters for huge savings in training cost while retaining accuracy. Inference tools like vLLM and LMDeploy focus on maximizing throughput on given hardware ³¹ ³⁵, whereas tools like Ollama prioritize ease of use ³⁰. The tables above summarize these trade-offs and specifications.

Citations: All statements and figures above draw on recent literature and documentation ¹⁰ ⁴ ⁷ ³ ⁵ ²⁴ ²⁵ ²⁶ ²⁷ ³⁰ ³³ ²⁰ ²¹ ³⁹ ¹⁵ ¹³ ³⁷, and the relevant references therein.

¹ TRL - Transformer Reinforcement Learning

<https://huggingface.co/docs/trl/en/index>

² Transformer Architectures - Hugging Face LLM Course

<https://huggingface.co/learn/llm-course/en/chapter1/6>

³ ⁴ ⁶ Transformer Design Guide (Part 2: Modern Architecture) | Rohit Bandaru

<https://rohitbandaru.github.io/blog/Transformer-Design-Guide-Pt2/>

5 Math Behind Positional Embeddings in Transformer Models | by Freedom Preetham | Autonomous Agents | Medium

<https://medium.com/autonomous-agents/math-behind-positional-embeddings-in-transformer-models-921db18b0c28>

7 8 9 What is grouped query attention? | IBM

<https://www.ibm.com/think/topics/grouped-query-attention>

10 11 Mixture of Experts Explained

<https://huggingface.co/blog/moe>

12 16 17 OpenAI's GPT-3 Language Model: A Technical Overview

https://lambda.ai/blog/demystifying-gpt-3?srltid=AfmBOooBpK-348iG6AMi7V3OuwgPp9L8dX_SkW3qamavz842a4JpdNH

13 14 config.json · TheBloke/Llama-2-70B-fp16 at main

<https://huggingface.co/TheBloke/Llama-2-70B-fp16/blob/main/config.json>

15 39 Mistral

https://huggingface.co/docs/transformers/main/en//model_doc/mistral

18 Llama 2

https://huggingface.co/docs/transformers/main/en/model_doc/llama2

19 20 LoRA

https://huggingface.co/docs/peft/main/en/conceptual_guides/lora

21 22 PEFT: Parameter-Efficient Fine-Tuning Methods for LLMs

<https://huggingface.co/blog/samuellimabraz/peft-methods>

23 Summary of the tokenizers

https://huggingface.co/docs/transformers/en/tokenizer_summary

24 tensorflow - Why Bert transformer uses [CLS] token for classification instead of average over all tokens?

- Stack Overflow

<https://stackoverflow.com/questions/62705268/why-bert-transformer-uses-cls-token-for-classification-instead-of-average-over>

25 abovo.co | Social Email | RE: The Definitive 2025 Guide to Vector Databases for LLM-Powered Applications (Deep Research via ChatGPT)

<https://www.abovo.co/sean@abovo42.com/134573>

26 Transformer Math (Part 1) - Counting Model Parameters

<https://michaelwornow.net/2024/01/18/counting-params-in-transformer>

27 28 29 LLM Evaluation: Metrics, Frameworks, and Best Practices | SuperAnnotate

<https://www.superannotate.com/blog/llm-evaluation-guide>

30 31 32 Ollama vs. vLLM: The Definitive Guide to Local LLM Frameworks in 2025

<https://blog.alphabravo.io/ollama-vs-vllm-the-definitive-guide-to-local-llm-frameworks-in-2025/>

33 34 Text Generation Inference

<https://huggingface.co/docs/text-generation-inference/en/index>

35 36 GitHub - InternLM/lmdeploy: LMDeploy is a toolkit for compressing, deploying, and serving LLMs.

<https://github.com/InternLM/lmdeploy>

37 38 config.json · TheBloke/Llama-2-7B-fp16 at main

<https://huggingface.co/TheBloke/Llama-2-7B-fp16/blob/main/config.json>