

Deep Learning: More About LLMs

Hédi Hadiji

Université Paris-Saclay - CentraleSupelec
hedi.hadiji@l2s.centralesupelec.fr

November, 2025

Summary up to now

Defined main neural network architecture pieces, discussed training methods.

Focused on the Attention mechanism and Transformer architectures, the central building blocks of modern LLMs.

Today: present more in detail a few aspects of the LLM training pipeline.

Motivation: Towards Large Language Models

Goal: provide some insights into modern practices in LLM development to understand what has become a central piece of technology today.

We will focus on a few key aspects of LLM training pipelines, notably some architectural choices, and fine-tuning strategies.

What we will not discuss at all:

- infrastructure: distributed training, memory optimization
- precision: quantization, data types, mixed precision
- evaluation: benchmarks, metrics

and many other important aspects.

Some Case studies: SmoLLM, Qwen

Two loose guiding cases:

- SmoLLM3 [paper] [model]
- Qwen [paper] [model]

We will refer to these two models to illustrate the concepts discussed.

Setting: Training a chat model, with two main parts

- Pretraining with next-token prediction on large corpora of text data
- Post-training: modify the model to guide the ouptut behavior towards desired characteristics

Focus on some key aspects of the training pipeline.

Table of Contents

1 Intro

2 Scaling Laws

3 Architecture

- Mixture of Experts
- Positional Encoding
- Other design choices

4 Post-training

- Instruction Fine-tuning

5 LoRA

6 Current Practices and Frontiers

Table of Contents

1 Intro

2 Scaling Laws

3 Architecture

- Mixture of Experts
- Positional Encoding
- Other design choices

4 Post-training

- Instruction Fine-tuning

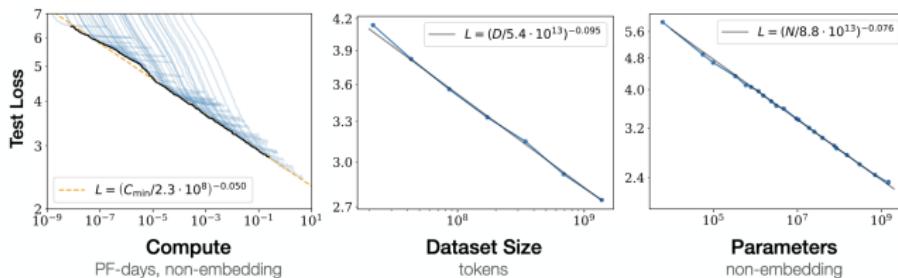
5 LoRA

6 Current Practices and Frontiers

Empirical Scaling Laws

Empirical relation between loss L and parameters N , data D , compute C :
 For a fixed N , increase C (or, equivalently increase D). Then at some point $C^*(N)$ the loss decrease slows down. Denote the value of the loss at this point $L^*(C)$.

$$L^*(C) \propto C^{-\alpha_C} \quad \text{and} \quad L^*(N) \propto N^{-\alpha_N} \quad \text{and} \quad L^*(D) \propto D^{-\alpha_D}.$$



- Data is number of fresh tokens seen during training, so compute is proportional to $\approx 6N \times D$.
- Scale N is number of parameters. Model is a transformer decoder, scale is changed by increasing number of layers, hidden size, attention heads

Using Scaling Laws

Scaling laws provide:

- An estimation of optimal model size and data for a given compute budget.
- A prediction of the loss at this optimal point.

Kaplan et al. (2020) estimate that given C , optimal N and D should scale as

$$N_{\text{opt}} \propto C^{0.73} \quad D_{\text{opt}} \propto C^{0.27}.$$

i.e., as compute budget grows, models should grow faster than new training data.

Scaling with Optimal Compute Dependence

Model loss as a function of model size and data size:

$$L(N, D) = L_\infty + aN^{-\alpha_N} + bD^{-\alpha_D}$$

Keeping compute C fixed, we have $D = C/N$, so

$$L(N, C) = L_\infty + aN^{-\alpha_N} + b \left(\frac{C}{N} \right)^{-\alpha_D}$$

so optimal N should satisfy

$$N^{-\alpha_N} \approx \left(\frac{C}{N} \right)^{-\alpha_D}$$

i.e.

$$N_{\text{opt}} \approx C^{\alpha_D / (\alpha_D + \alpha_N)}. \quad \text{and} \quad D_{\text{opt}} \approx C^{\alpha_N / (\alpha_D + \alpha_N)}.$$

And the loss will decrease as $N_{\text{opt}}^{-\alpha_N}$

Improved Measurements:

- Hoffmann et al. (2022): $\alpha_N \approx 0.34$, $\alpha_D \approx 0.28$. Then $N_{\text{opt}} \approx C^{0.45}$ and $D_{\text{opt}} \approx C^{0.55}$, and $L(C) \approx C^{-0.16}$.

Chinchilla Scaling Recommendation

Scale data proportionally to model size. Empirical rule of thumb: $D \approx 20N$.

Limits of scaling (laws)

Scaling laws have been a strong driver of the push for ultra-large models, providing quantitative predictions of the benefits of scaling.
As for any empirical laws, they have limitations:

- Loss will not decrease indefinitely: natural language has inherent entropy, and real-worlds models are bound by physical limitations + data is not infinite.
- Practical factors may change scaling: data quality, architecture changes, optimization practices, etc.
- Compute budget at training is not the only scarce resource and one must also consider inference cost in the loop, which favor smaller models too.

Table of Contents

1 Intro

2 Scaling Laws

3 Architecture

- Mixture of Experts
- Positional Encoding
- Other design choices

4 Post-training

- Instruction Fine-tuning

5 LoRA

6 Current Practices and Frontiers

Motivations for architectural changes

Standard transformer decoder architecture has limitations.

- **Quadratic memory** and compute cost of attention limits context length.
- Dense models have **high inference cost** for large parameter counts.
- Need for better **length generalization**, long-context handling + efficiency.

Scale and Parameter type: the Example of GPT-3

Three types of layers:

- Embedding layer: maps input tokens to vectors. Size: $V \times d$.
- Self-attention layers: L layers, each with self-attention.
- Feed-forward layers: L layers, each with two linear layers and a non-linearity.

Model	Layers	d_{model}	d_{ff}	Heads	Total Params
175B	96	12288	49152	96	175B
Layer Type	Formula		Parameters	Fraction	
Self-Attention	$4d_{\text{model}}^2$		58 B	33%	
Feedforward	$2d_{\text{model}}d_{\text{ff}}$		116 B	66%	
LayerNorms + biases	$\approx 9d_{\text{model}}L$		< 0.01 B	< 0.01%	
Token Embedding	Vd_{model}		0.62 B	0.4%	
Total			≈175 B	100%	

Table: GPT-3 175B parameter count per component.

Mixture of Experts: Goal

- Large models are costly to deploy.
- **Key idea:** Activate only a subset of model parameters per input.
- **Mixture of Experts (MoE)** introduces conditional computation:

$$y = \sum_{i=1}^M g_i(x) f_i(x)$$

where:

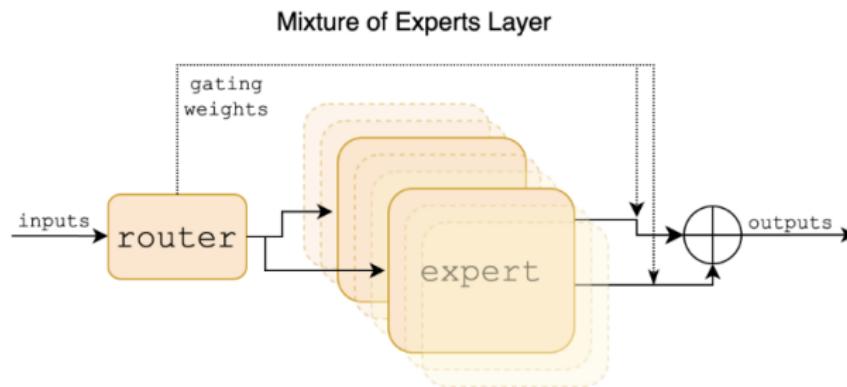
- $f_i(x)$ = output of expert i
- $g_i(x)$ = gating weight (learned function, typically softmax)

- A **gating network** selects one or few experts for each input.
- Each **expert** is a small feedforward network (FFN or MLP).
- Typical MoE block replaces the Transformer feed-forward layer.

Goal

Scale model capacity without increasing inference cost per token.

Architecture Overview



Gating layer outputs logits over experts. Only activate the top- k experts. Compute their outputs and combine them using the gate probabilities.

Issue with argmax is non-differentiability.

Gumbel-Softmax Trick

Theorem: Gumbel-Softmax

If g_i are logits, and G_i are i.i.d. Gumbel(0,1) random variables i.e. if $U_i \sim \text{Uniform}(0, 1)$ and $G_i = -\log(-\log(U_i))$, then

$$\arg \max_i (g_i + G_i) \sim \text{softmax}(g)$$

Therefore, if $y_{\text{hard}} \in \mathbb{R}^{n_{\text{exp}}}$ is the one-hot encoding of $\arg \max_i (g_i + G_i)$, and $y_{\text{soft}} = \text{softmax}(g)$, then

$$\mathbb{E}[y_{\text{hard}}] = y_{\text{soft}}.$$

Idea: use this to have **argmax in forward pass** and **softmax in backward pass**, so that the gradient flows coherently through the softmax approximation, while the forward and backward are consistent (in distribution).

In practice, MoEs can collapse to using only a few experts: add a **load balancing loss** that encourages uniform expert usage.

MoE saves inference compute

Since the FF layers account for ~66% of parameters and compute in GPT-3, replacing them with MoE layers can greatly increase parameter count without increasing inference cost.

e.g for known MoE models and their active parameter usage per token.

Model	Total Params	Experts	k	Active Params / Token
Switch Transformer	~1.5T	2048	1	~ 1.4B
Mixtral 8×7B	46.7B	8	2	12.9B
Qwen3-235B-A22B	235B	128	2	22B
Qwen3-30B-A3B	30B	-	-	3B

Positional Encoding: Motivation

- The Transformer architecture has no inherent notion of order.
- Self-attention is permutation-invariant: shuffling tokens yields the same output.
- We must inject positional information into token embeddings.

Positional crucially affects **context-length generalization**, i.e., the ability of models to handle sequences longer than those seen during training.

Recall the Transformer architecture can handle variable-length sequences, and has a priori no limit on context length. With positional encoding, we hard code some notion of order and sequence length in the model.

Formulation

Let $x_i \in \mathbb{R}^d$ be the embedding of token i .

We add a position-dependent vector p_i :

$$z_i = x_i + p_i$$

The model learns to use these p_i to infer order relationships (who attends to whom in attention).

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right),$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

Each dimension corresponds to a sinusoid of a different frequency.

- Smoothly encodes positions; nearby positions have similar encodings.
- Enables extrapolation to unseen sequence lengths
- Hard-codes absolute position in the sequence

Rotary Positional Encoding (RoPE)

Recent models use **Rotary Positional Encoding** on keys and queries:

$$\text{RoPE}(q_p) = R(p)q_p \quad \text{RoPE}(k_p) = R(p)k_p$$

where $R(p)$ is a rotation matrix:

$$R_p = \begin{pmatrix} R(\theta_1(p)) & 0 & 0 & \cdots & 0 \\ 0 & R(\theta_2(p)) & 0 & \cdots & 0 \\ 0 & 0 & R(\theta_3(p)) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & R(\theta_{d/2}(p)) \end{pmatrix}, \quad R(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$$

and

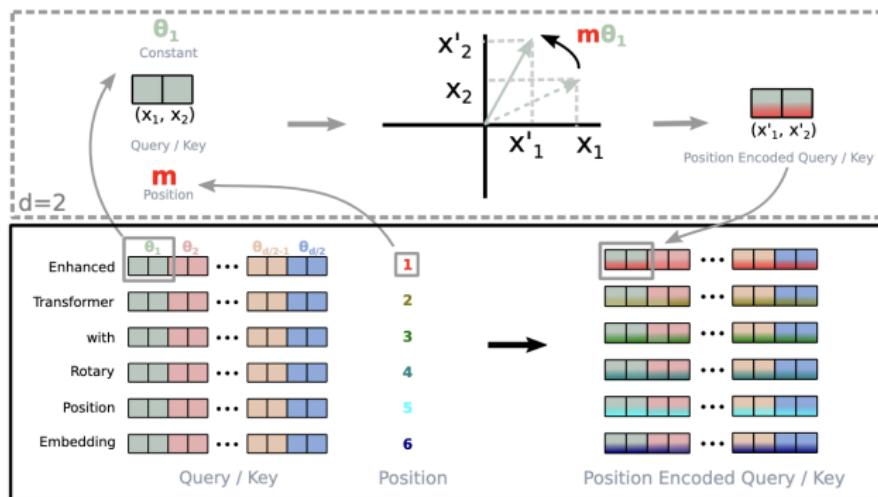
$$\theta_i(p) = \frac{p}{10000^{2i/d}}$$

Then when computing attention between tokens i and j :

$$\langle \text{RoPE}(q_p), \text{RoPE}(k_{p'}) \rangle = \langle q_p, R_{p-p'} k_{p'} \rangle$$

- Encodes relative position via phase differences.
- Supports extrapolation and efficient attention computation.

RoPE



Other design choices that matter

Attention variants: no need to have different weights for keys and queries for different heads in the same layer, Grouped-Query Attention (GQA) or Multi-Query attention share the weights.

Masking: many variants are possible in masking on top of causal masking: local attention, sliding window, etc.

Tokenizer: the way text is broken into tokens greatly affects performance, in hard to predict ways. Vocabulary size for Qwen3 and SmollM3: ~150k tokens.

Table of Contents

- 1 Intro
- 2 Scaling Laws
- 3 Architecture
 - Mixture of Experts
 - Positional Encoding
 - Other design choices
- 4 Post-training
 - Instruction Fine-tuning
- 5 LoRA
- 6 Current Practices and Frontiers

Instruction Fine-tuning

You have a model that is pretrained on a large corpus of data to complete text.

You want to make it answer in a specific way, e.g., follow instructions, answer questions.

Build a chat template that will define the input prompt structure.
(Possibly with a 'system prompt' prepended to every input.)

Example:

```
<|im_start|>user  
What's the weather like in New York?<|im_end|>  
<|im_start|>assistant  
The weather in New York is currently sunny with a temperature of  
22C. The humidity is at 45% with a wind speed of 10 km/h. It's  
a great day to be outside!<|im_end|>
```

Train model to produce the assistant response given the user prompt.

Supervised Fine-Tuning (SFT)

Goal: Adapt a pretrained language model to follow instructions and align with human intent.

- Uses labeled **instruction-response pairs**: (x, y^*) .
- Trains via standard cross-entropy loss:

$$\mathcal{L}_{\text{SFT}} = -\mathbb{E}_{(x, y^*)} \sum_t \log p_{\theta}(y_t^* | y_{<t}^*, x)$$

with teacher-forcing.

- Shifts model distribution toward structured outputs.

SFT Data

Examples of SFT data:

Question:

How do you simplify $(u^4 v^3 / (u^2 v^{-1})^4)^0$ and write it using only positive exponents?

Answer:

To simplify the expression $(u^4 v^3 / (u^2 v^{-1})^4)^0$ and rewrite it using only positive exponents, we start by evaluating the expression from the innermost operation outward.

First, consider the exponent of zero on the entire fraction:

Any expression raised to the power of zero is equal to 1 (provided the base is not zero). Therefore, $(u^4 v^3 / (u^2 v^{-1})^4)^0 = 1$

This simplification makes the other calculations unnecessary because raising any non-zero expression to the power of zero will always result in 1.

So, the simplified expression is simply: 1

Why SFT Works

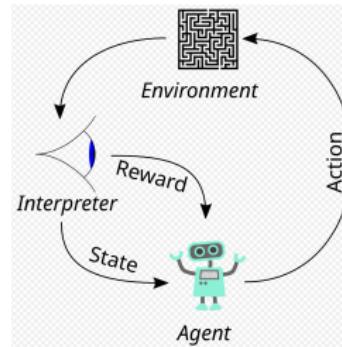
- Pretrained models learn general language priors.
- SFT teaches **task conditioning**: “Given this instruction, produce a coherent, relevant output.”
- Provides initial alignment before reinforcement learning (RLHF, DPO, etc.).
- Acts as a regularizer: keeps responses fluent but steers toward desired behaviours.

SFT narrows the model’s output manifold to align with human-like task formatting.

In practice, SFT is often the first step in a multi-stage alignment pipeline. We **still need to make sure the outputs are ‘good/safe’ enough**. (e.g. the model can still produce harmful or untruthful outputs after SFT.)

Fine-tuning as a Reinforcement Learning Problem

Reinforcement Learning: State, Actions, Rewards.



Policy: maps states to actions.

Given a reward function that scores outputs, we can formulate fine-tuning as an RL problem:

- **State:** input prompt and conversation history.
- **Action:** next token
- **Reward:** reward of the output.

The **policy is the model**: a map from states (prompts) to actions (tokens).
Two flavors of RL: RLHF and RLVR.

RLHF (Reinforcement Learning from Human Feedback)

Idea: Use human preference data to train a *reward model* and fine-tune the base model via reinforcement learning.

Pipeline:

- 1 Collect human preference pairs (y^+, y^-) for the same prompt x .
- 2 Train a reward model $r_\phi(x, y)$ so that $r_\phi(x, y^+) > r_\phi(x, y^-)$.
- 3 Fine-tune the language model π_θ using RL (often PPO) to maximize expected reward:

$$\max_{\theta} \mathbb{E}_{x,y \sim \pi_\theta} [r_\phi(x, y)] - \beta \text{KL}(\pi_\theta || \pi_{\text{SFT}})$$

- 4 β controls deviation from the SFT model.

Goal: Align model behavior with human preference while preserving fluency and diversity.

Issue: Instability, reward overfitting (“reward hacking”), expensive human annotation.

RL with Verifiable Rewards

Idea: Get rid of model and human preferences: Train on problems that have an objective solution.

e.g. math problems, coding problems, etc.

GRPO ([paper]) is a good RL algorithm for the specific case of training LLMs:

- Rewards are only given at the end of the sequence.
- Can sample multiple outputs per prompt to estimate expected reward.

DPO (Direct Preference Optimization)

Motivation: Avoid explicit RL training and reward model sampling.

Key idea: Optimize model directly on human preference pairs (x, y^+, y^-) .

$$\begin{aligned} L_{\text{DPO}}(\theta) = -\mathbb{E}_{x,y^+,y^-} \left[\log \sigma \left(\beta \left(\log \pi_\theta(y^+|x) - \log \pi_\theta(y^-|x) \right. \right. \right. \\ \left. \left. \left. - \log \pi_{\text{ref}}(y^+|x) + \log \pi_{\text{ref}}(y^-|x) \right) \right) \right] \end{aligned}$$

- π_{ref} : reference (typically SFT) model.
- β : inverse temperature controlling preference strength.
- Equivalent to optimizing for the same fixed point as RLHF under a KL constraint.

Benefits: No RL loop, stable gradients, easier implementation. Used in Qwen3 post-training.

Table of Contents

1 Intro

2 Scaling Laws

3 Architecture

- Mixture of Experts
- Positional Encoding
- Other design choices

4 Post-training

- Instruction Fine-tuning

5 LoRA

6 Current Practices and Frontiers

LoRA: Motivation: Finetuning Large Models

Finetuning all parameters of large pretrained models is expensive.

- Billions of parameters \Rightarrow huge memory and compute cost.
- Storing separate copies for each task is impractical.

Goal: Adapt models efficiently with minimal trainable parameters.

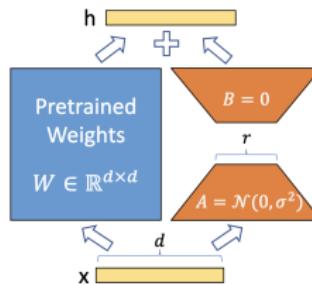
Idea of LoRA (Low-Rank Adaptation)

Instead of updating all weights $W_0 \in \mathbb{R}^{d \times k}$, we **freeze** W_0 and learn a low-rank update:

$$W = W_0 + \Delta W, \quad \Delta W = AB, \quad A \in \mathbb{R}^{d \times r}, \quad B \in \mathbb{R}^{r \times k},$$

where $r \ll \min(d, k)$.

- Train only A, B (few million parameters).
- Keeps inference cost identical to original model.



Implementation in Practice

For a linear layer $y = W_0x$, LoRA modifies the forward pass:

$$y = W_0x + \alpha A(Bx),$$

where α is a scaling factor (often α/r).

- W_0 frozen, gradients flow only through A, B .
- Drop-in replacement for linear/attention projection layers.
- Often applied to query/key/value projections in Transformers.

Concretely: take a model, pick some layers. Replace the forwards by a wrapped LoRA layer with the same input-output sizes.

Why (?) It Works

- Neural networks often lie in a low-intrinsic-rank subspace.
- Adaptation directions can be well-approximated by low-rank updates.

Benefits:

- *Parameter-efficient*: train 1% of weights.
- *Composable*: merge LoRA adapters for multiple tasks.
- *Deployable*: inference identical to base model.

Can use LoRA at any stage of post-training: SFT and RL.

Table of Contents

- 1 Intro
- 2 Scaling Laws
- 3 Architecture
 - Mixture of Experts
 - Positional Encoding
 - Other design choices
- 4 Post-training
 - Instruction Fine-tuning
- 5 LoRA
- 6 Current Practices and Frontiers

Current + Future practices

More optimization: extra long context, scaling RL

Reasoning: train the models to use extra tokens to dynamically allocate extra computations before answering.

Multimodality: handle sound and images in addition to text (remember CLIP)

Agents: handle actions, tools use.

Beyond single LM systems: memory, retrieval, modular systems.

Small models: distillation, compression, efficient architectures.