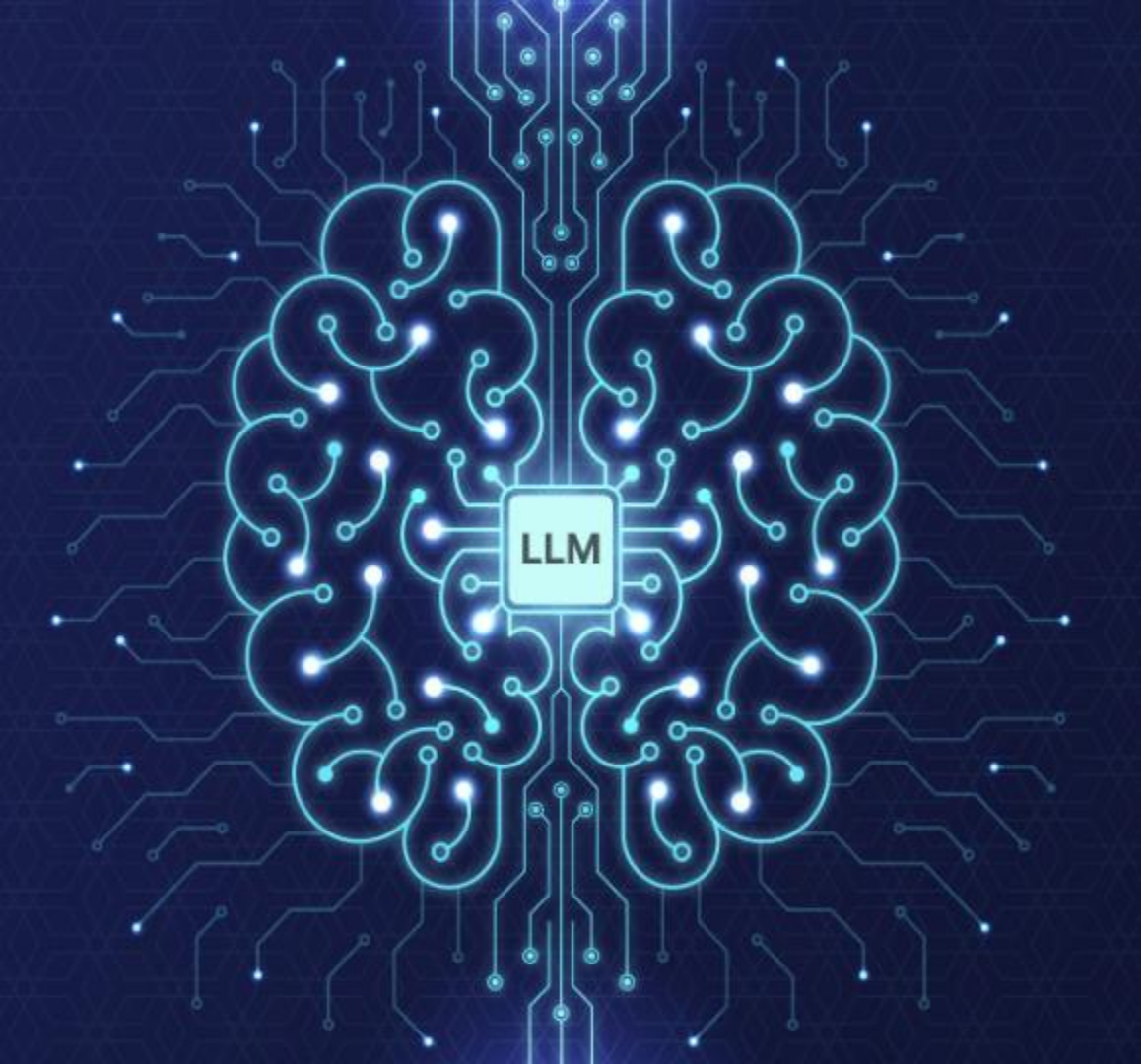


Fundamentals of Large Language Models

October 2025

What is language modeling?

A language model (LM) is a machine learning model that aims to predict and generate plausible language.

These models work by estimating the probability of a *token* or sequence of tokens occurring within a longer sequence of tokens.

The cat sat on the _____

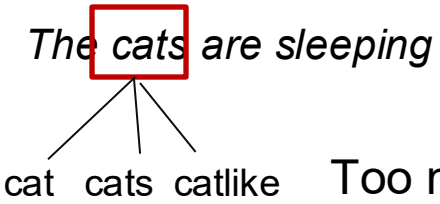
Couch	*9.4%*
Car	*7.1%*
Fork	*2.6%*
….	

The LM most probably predicts 'couch', since it has learned during massive pre-training that 'a cat sat on the couch' is a **common** situation in natural language.

From words to tokens

Motivation: Why Tokenization Exists

Natural language is continuous and ambiguous, but models require discrete, finite inputs. Tokenization bridges this gap by establishing manageable units.



Too many unique forms!

Goal

Find a unit of text *small enough* to generalize across words but *large enough* to avoid huge vocabularies.

Tokenization

Most common approach: Subword-level

Breaks words into frequent subunits (subwords) that can be recombined.

Various widely adopted techniques:

- **Byte Pair Encoding (BPE)** – used in GPT, RoBERTa
- **WordPiece** – used in BERT
- **SentencePiece** – used in T5, LLaMA
- **Unigram LM** – probabilistic approach to subword segmentation

Example (BPE):
“unhappiness” → [“un”, “happy”, “ness”]

Advantages

- Handles rare and new words.
- Keeps sequence lengths manageable.
- Enables compression and generalization.

Vectorization

Once text is tokenized, each token index is mapped to a dense vector via an embedding matrix.

$$E \in \mathbb{R}^{V \times d}$$

- V = vocabulary size (e.g., 50,000)
- d = embedding dimension (e.g., 1024).

Each token t_i corresponds to a vector E_{ti} representing its semantic and syntactic role.

This conversion from token to vectors is necessary since a neural network (the LLM) cannot process words, but it can process numbers.

Positional Encodings

Transformers have no inherent order awareness, since attention mechanism treats input as a set.

Positional information: “cat sat on the couch” ≠ “couch sat on the cat”.

- **Sinusoidal Encoding** (original Transformer): Adds smooth periodic signals for relative position comparison.
- **Learned** Positional Embeddings: Position vectors are learned like token embedding.
- **Rotary** Position Embeddings (RoPE): Used in modern LLMs (LLaMA, Mistral). Encodes position within attention computation by rotating query/key vectors.

The model’s **input** representation = **Token Embedding + Positional Encoding**

These vectors (one per token) are the continuous representation of discrete language.

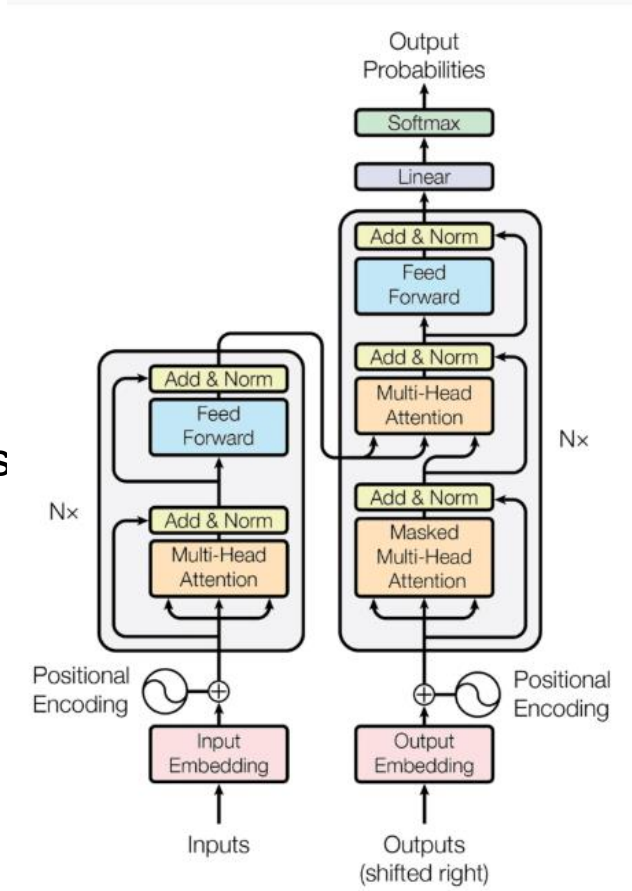
Transformer architecture for language modeling

Each Transformer layer (in GPT, BERT, etc.) consists of two main sub-layers:

- **Self-Attention layer**: lets every token attend to every other token in the sequence (context mixing).
- **Feedforward layer (MLP)**: transforms each token's representation independently (nonlinear transformation).

Each sub-layer is wrapped with:

- **Residual connections**: gradient stabilization
- **Layer normalization**: numerical stabilization



Attention mechanisms

At its core, attention takes a bunch of input word representations and computes how much each one should influence every other.

We break down attention in the following steps:

Step 1: Each Token Looks Around

“Which other tokens are relevant to me right now?”

It compares itself to all others using a similarity measure (dot products between “queries” and “keys”). The more similar the context, the higher the “attention score.”

Attention mechanisms

Step 2: Convert Scores to Weights
These similarity scores are normalized (via a softmax) so they become attention weights, like percentages of focus.

Example (conceptually): “it” gives 70% of its attention to “animal.” That’s re-weighting: “animal” now has the biggest influence on what “it” means in context.

Token	Relevance to “it”	Attention Weight
The	low	0.05
cat	very high	0.70
didn’t	medium	0.10
cross	low	0.05
street	low	0.05
because	medium	0.05

Attention mechanisms

Step 3: Weighted Averaging of Information
The model takes all token embeddings and combines them, but multiplies each by its attention weight before summing.
So the new representation of “it” becomes mostly composed of “cat’s” vector, with a touch of other context.

Step 4: Contextualized Output
The result: Every token now has a context-aware embedding and it knows what matters to it in the current sentence:

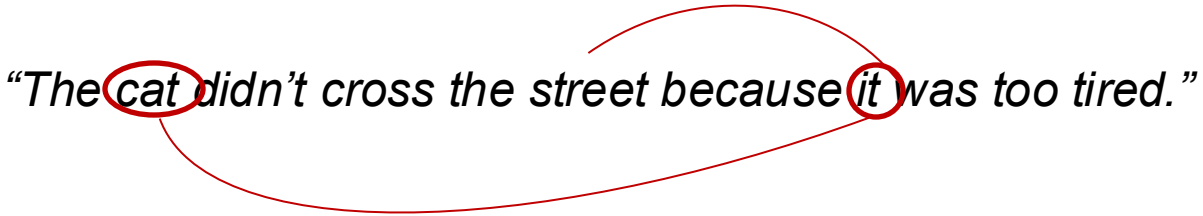
- “it” knows it refers to “cat”
- “street” knows it’s a location
- “tired” knows it modifies “cat,” not “street”

This happens for every token, in parallel.

Attention mechanisms

Self-Attention

Self-attention allows each token to look at (attend to) other tokens in the *same* sequence to decide **what's important** for predicting the next token. It dynamically computes context-dependent weights between all tokens, meaning the model can capture long-range dependencies (e.g., linking "cat" and "it" even if separated by many words).



That's attention: deciding which earlier words matter most for understanding the current one.

Attention mechanisms

Multi-head attention

Instead of a single attention operation, Transformers use multiple attention heads in parallel to achieve multi-perspective re-weighting. Each head learns a different type of relationship (e.g., syntactic, semantic, long-range dependencies).

Typical configurations

- GPT-3: 96 heads
- GPT-4 / GPT-5: hundreds of attention heads across billions of parameters

Causal masking

Because GPT-style models predict tokens left-to-right, they must not peek at future tokens. So we apply a causal mask — set all scores for $j>i$ to $-\infty$ before softmax, ensuring the model can only attend to previous tokens.

This keeps the autoregressive property intact.

Feedforward (FFN) layers

After attention builds context, we need a mechanism to transform each contextualized representation into richer, nonlinear features.

Feedforward layers instruct **how to process the information coming from attention**.

Each token's representation after attention re-weighting zi independently passes through the same 2-layer MLP followed by an activation function (ReLU, GeLU etc).

The FFN layer

- Adds nonlinearity
- Expands representational capacity
- Enables hierarchical feature transformations

Residual connection and layer normalization

Each sublayer (Attention and FFN) is wrapped in residual connections enabling stable gradient flow, faster convergence and smoother training for deep networks (hundreds of layers).

Why Residuals Matter

- Without them, information from early layers would degrade as it moves through depth (vanishing gradient problem).
- Residuals allow the model to add new knowledge while preserving the old one.

Language Model training

Given a context (tokens so far), what is the probability distribution over all possible next tokens?

Formally:

$$P(w_t|w_1, w_2, ..., w_{t-1})$$

Training Objective

The model is trained on large corpora to maximize the likelihood of the correct next token given the previous ones. For a sequence of tokens w_1,w_2,...,w_n, the model tries to maximize the **joint probability** for the whole sequence:

$$\prod_{t=1}^{n} P(w_t|w_1, w_2, ..., w_{t-1})$$

Language Model training

Training Objective

Equivalently, it minimizes the negative **log-likelihood (NLL)** or **cross-entropy** loss, which measures the distance between two distributions:

$$\mathcal{L} = -\sum_{t=1}^{n} \log P_\theta(w_t|w_1, ..., w_{t-1})$$

- θ = model parameters (weights),
- $P_\theta(\cdot)$= the probability distribution predicted by the model.

Ultimately, we are comparing the model's predicted distribution with the "true" one-hot distribution (where only the correct token has probability 1) and tries to make the true next token's log-probability as large as possible.

Lower loss means the model is **better** at predicting the correct token.

How to compute token probabilities

Model architecture

The model learns to compute $P(w_t|w_1,w_2,...,w_n)$ using a transformer architecture, which includes:

- **Token embeddings**: Each token is converted into a vector.
- **Positional embeddings**: Add information about the token's position in the sequence.
- **Self-attention layers**: Each token attends to all previous tokens (not future ones, since prediction must be causal).
- **Feedforward layers**: Nonlinear transformations that allow abstraction and pattern learning.

How to compute token probabilities

At the top layer, the model produces a probability distribution over the vocabulary for the next token via a softmax function.

Probability distribution

At each step t, the model produces a logit vector z_t, one unnormalized score per vocabulary token.

To convert logits into probabilities we enforce a softmax function which ensures:

- All probabilities are positive
- They sum to 1
- The model outputs a valid distribution over the vocabulary.

Perplexity: Measuring Model Quality

Perplexity (PPL) measures how “surprised” a model is by the true data

Interpretation

- Low perplexity → model assigns high probability to correct tokens → better predictive fit.
- High perplexity → model is uncertain or wrong often.

Example

- Random model: high perplexity (≈ vocabulary size).
- Fluent LLM: low perplexity (~10–30 on test corpora).

Training details

Initialization

Proper weight initialization is crucial for stable Transformer training, otherwise gradients can explode or vanish. Good initialization prevents early divergence before the optimizer takes over.

Goal: ensure that activations and gradients stay in a “healthy” range across layers.

Optimization

Transformers almost universally use **AdamW**, a variant of Adam with decoupled weight decay. This makes training more stable and weight decay more predictable, a crucial factor for billion-scale language models.

Training details

Learning Rate Scheduling

The learning rate is not static; it follows a carefully shaped schedule to prevent instability.

Warmup + Cosine Decay (most common)

- **Warmup phase**: Gradually increase the learning rate from 0 to its peak over a few thousand steps. This strategy prevents gradient explosion in early steps.
- **Decay phase**: After warmup, learning rate slowly decays (often cosine-shaped) to fine-tune model weights smoothly.

Typical settings

- Warmup steps: 1k–10k
- Total steps: millions
- Final LR: ~10× smaller than peak LR

Training details

Mixed Precision Training (FP16 / BF16)

Motivation: speed + memory efficiency.

Instead of using 32-bit floats everywhere

- Store most activations in 16-bit (FP16 or BF16)
- Keep some critical variables (like loss scaling, running averages) in 32-bit

Benefits

- 2× faster training
- 2× less memory

Nearly no loss in accuracy (if done correctly).
BF16 (bfloat16) is preferred now because it avoids numerical underflow without needing dynamic loss scaling.

Training details

Parallelism: How Large Models Fit Across Many GPUs

- **Data Parallelism**: Different GPUs process different mini-batches
- **Model Parallelism**: Different GPUs hold different parts of the model (layers or weights)
- **Tensor Parallelism**: Split individual matrix multiplications across GPUs
- **Pipeline Parallelism**: Split model by layers; pass activations between GPUs

Overfitting and Regularization

Even massive models can overfit, especially on small or narrow datasets.

Dropout

- Randomly zeroes some activations during training.
- Typical dropout rate: 0.1–0.2 in Transformers.
- Applied in attention and FFN layers.

Weight Decay (L2 Regularization)

- Prevents weights from growing too large.
- Integrated in AdamW.
- Encourages smoother generalization.

Overfitting and Regularization

Early Stopping & Validation Loss

- Monitor perplexity on a held-out validation set.
- Stop training when loss stops improving.

Data Regularization

- Massive diverse datasets (web, books, code) act as implicit regularization.
- Deduplication and data mixing reduce memorization risk.
- Some models use noise injection (token masking or random token corruption).

Overfitting and Regularization

Label Smoothing

- Softens one-hot targets slightly (e.g., assign 0.9 to the correct token, 0.1 spread over others).
- Prevents overconfidence and improves calibration.

Gradient Clipping

- Caps gradient norm (e.g., at 1.0) to prevent instability.
- Especially critical for large batch sizes.

Regularization Through Scale

Counterintuitively, **larger** models **overfit less** if trained with more data; this is the essence of scaling laws. Training data grows with parameter count, keeping the model in the “predictive generalization” regime.

LLM inference

Language Model inference

At inference time (after training), next-token prediction is used iteratively:

- Input: “The cat sat on the”
- Model predicts probability distribution over next tokens.
- Sample or choose the highest-probability token (“couch”).
- Append it to the context → “The cat sat on the couch”.
- Repeat.

Thus, generation = repeated next-token prediction.

Sampling strategies

Deterministic vs. Stochastic Sampling

At every generation step, the model outputs a probability distribution over the vocabulary:

$$P_\theta(w_t|w_{<t})$$

We can either:

- Choose deterministically (always pick the most probable token)
- Sample stochastically (draw randomly from the probability distribution)

Each method has trade-offs between coherence, diversity, and creativity.

Sampling strategies

Greedy decoding

At each step, select the token with the highest probability.

$$w_t = \arg\max_i P_\theta(w_t = i | w_{<t})$$

Pros

- Fast and simple
- Produces grammatical, predictable text

Cons

- Often repetitive (“The cat sat on the the the…”)
- Gets stuck in local optima — can’t recover from early mistakes
- Low diversity (always the same output for the same prompt)

When to use: factual completion tasks, short deterministic answers (e.g., code completion).

Sampling strategies

Beam Search (Semi-Deterministic)

Instead of committing to one token at a time, keep multiple top candidate sequences (beams) at each step.

Keep top-k sequences by cumulative log-probability.

Pros

- More global search than greedy
- Often used in translation tasks (seq2seq models)

Cons

- Computationally expensive
- Still low diversity
- Not ideal for open-ended generation (like storytelling or chat)

Sampling strategies

Random Sampling (Stochastic)

Sample from the full probability distribution.

$$w_t \sim P_\theta(w_t|w_{<t})$$

Pros

- Creative and varied
- Captures the model's uncertainty

Cons

- Can generate incoherent or irrelevant tokens if low-probability options are sampled
- Requires careful temperature tuning (see below)

When to use: open-ended text generation, brainstorming, creative writing.

Sampling strategies

Temperature Scaling

Before sampling, we can control the sharpness of the probability distribution using a temperature parameter T.

- Low T (< 1): sharper distribution — model becomes more deterministic
- High T (> 1): flatter distribution — model becomes more random

Temperature	Effect
0.2	Highly deterministic, repetitive
1.0	Balanced creativity & coherence
1.5+	Highly random, incoherent

Sampling strategies

Top-k Sampling

Instead of sampling from the entire vocabulary (often tens of thousands of tokens), restrict to the top k most probable tokens.

Then renormalize probabilities and sample from that subset.

Pros

- Prevents low-probability (nonsensical) tokens
- Retains controlled randomness

Typical values: k=20–50

Effect

Balances quality and diversity, much better than full random sampling.

Sampling strategies

Top-p (Nucleus) Sampling

Instead of a fixed number of tokens, dynamically choose the smallest set of tokens whose cumulative probability ≥ p.

Then sample only from that nucleus.

Typical values: p=0.8–0.95

Pros

- Adapts to context: few options for confident predictions, more options when uncertain
- Produces fluent and diverse text
- Avoids unnatural truncation of the probability tail

Effect

Much more human-like text; this is the default in GPT-family models.

Learning outcomes

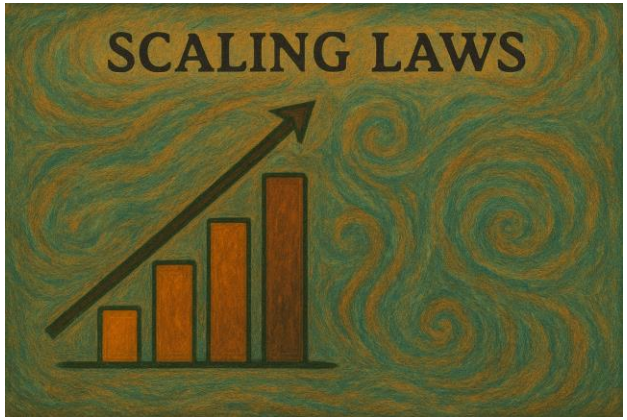
Although the training objective is simple (just next-token prediction), it implicitly teaches the model to:

- Learn **grammar** and **syntax** (to predict correct structures)
- Capture **semantics** (to predict contextually appropriate words)
- Encode **world knowledge** (to predict factual continuations)
- Model **pragmatics**, **style**, and **tone** (to match discourse patterns)

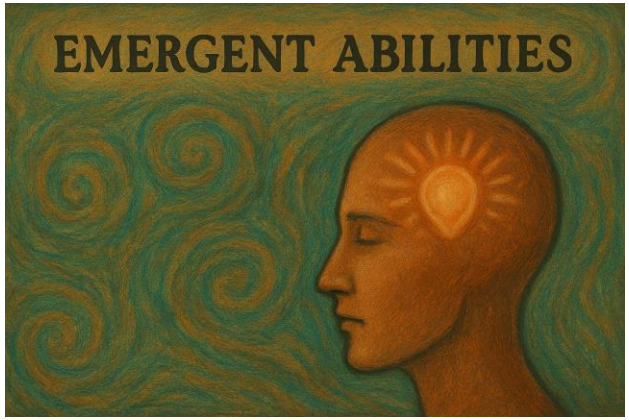
This happens because next-token prediction forces the model to compress the structure and meaning of language into its parameters.

Instead of exact memorization, the model learns statistical and conceptual regularities across *billions* of examples, so it can guess the next token even in novel contexts.

Emergent abilities



- *Predictable*, smooth performance improvements with *diminishing* returns when scaling data, parameters and compute
- *Quantitative* measure: Often measured by language modeling loss
- Continuous and monotonic improvement



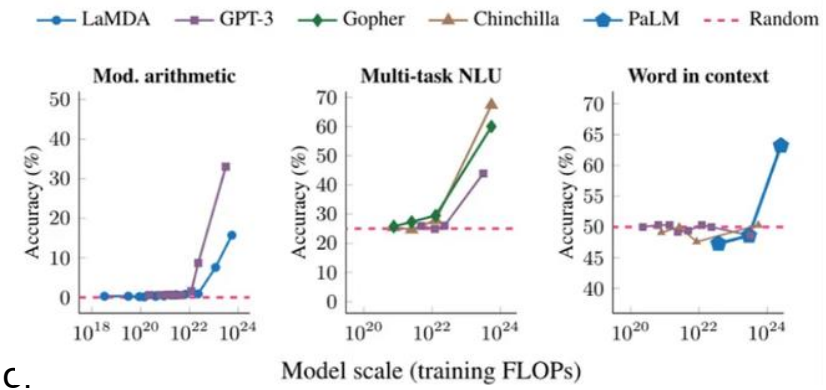
- Capabilities that appear *discontinuously* and *unpredictably* once the model crosses a certain scale threshold.
- *Qualitative* measure: Often measured by task performance
- *Sharp* performance leap: nonlinear, abrupt, phase-transition-like jumps
- Similar to how neural networks suddenly begin to classify non-linear patterns once enough neurons/layers exist.

On the cost of emergent abilities

Emergent abilities tend to appear once the training compute budget (total FLOPs used to train the model)crosses certain thresholds for a given architecture and data regime.

FLOPs = Model size (Parameters) x Training Tokens x Operations per Token

Model size: The number of parameters in the model. Larger models require more operations per forward and backward pass.
Training Data (Tokens): The number of tokens the model processes during training.
Operations per Token: FLOPs per training iteration, including forward pass, backward pass, and optimization steps.



FLOPs are a strong correlate (but *not* a guaranteed cause) c. emergence. In practice, architecture, data composition and optimization matter too.

Why HPC is Essential for LLMs

As language models grow in size, we observe consistent improvements in:

- Accuracy
- Generalization, and
- Task versatility.

These gains emerge only when models **reach certain parameter and compute thresholds**.

Training at such scales **requires massive computational capacity**. HPC provide the **parallelism, bandwidth, and reliability** needed to handle trillions of operations during training.

Without HPC-level resources, it becomes **impossible** to explore larger architectures or reach the regimes where advanced capabilities begin to appear.

Compute Limits LLM Scaling

Consumer-grade environments come with strict constraints on memory, compute power, and runtime. These limitations make it impossible to run larger architectures or perform full-scale experiments with modern LLMs.

- e.g., Google Colab and Kaggle typically allow models up to ~7B parameters.
- Larger models (e.g., 70B) cannot be executed or evaluated end-to-end in these environments.

Even when using the **same dataset and the same methods, the performance** changes dramatically simply by increasing the model size. **Scaling alone drives the performance jump.**

- **Larger models = better performance using the exact same data and methods.**
- **HPC enables scaling to much bigger models → higher accuracy without changing anything else.**

Scaling Example

We consider a high-difficulty NLP task: automatically judging whether a political answer is clear, vague, or evasive.
The task is challenging even for strong language models.

Using the same dataset and the same tuning method, only the model size changes, and the results shift dramatically.

The experiments were conducted on an HPC system, specifically on **Meluxina**.

Model	Size	Accuracy	F1
Llama	7B	0.489	0.457
Llama	13B	0.587	0.580
Llama	70B	**0.759**	**0.680**
Falcon	7B	0.288	0.175
Falcon	40B	**0.341**	**0.356**

Scaling Example

Llama:

- **7B: Accuracy 0.49, F1 0.46**
- **13B: Accuracy 0.59, F1 0.58**
- **70B: Accuracy 0.76, F1 0.68**

Falcon:

- **7B: Accuracy 0.29, F1 0.18**
- **40B: Accuracy 0.34, F1 0.36**

Model	Size	Accuracy	F1
Llama	**7B**	0.489	0.457
Llama	**13B**	0.587	0.580
Llama	**70B**	**0.759**	**0.680**
Falcon	**7B**	0.288	0.175
Falcon	**40B**	**0.341**	**0.356**

Same data, same method, **only the model scale changes**.
And as scale increases, performance improves substantially.

- **≈20% performance gain simply by scaling the model size.**

This kind of scaling, moving from 7B to 70B models, requires HPC infrastructure, since consumer hardware cannot support training or full evaluation of such large models.

Scaling Resources

Scaling to ultra-large models pushes hardware far beyond consumer GPUs.
Even single-GPU inference becomes impossible without enterprise-grade accelerators.

Model	Size	Memory	Recommended GPUs
Llama 3 / 3.1	7B	4.5–4.9 GB	GTX 1660 / RTX 3060 Ti
	70B	40–43 GB	RTX A6000 48GB / A40 48GB
	405B	243 GB	4×A100 80GB
DeepSeek R1	7B	4.7 GB	GTX 1660 6GB
	70B	43 GB	RTX A6000 / Dual 4090
	671B	404 GB	Multi-GPU Cluster (A100/H100)
Qwen (2.5 / 1.5)	7B	4.4–4.7 GB	GTX 1660 6GB
	32B	20 GB	RTX 4090 / A5000 24GB
	110B	63 GB	A100 80GB / H100
Gemma 2	2B	1.6 GB	Quadro P1000 4GB
	9B	5.4 GB	RTX 3060 Ti 8GB
	27B	16 GB	RTX 4090 / A5000
Mixtral (MoE)	8×7B	26 GB	RTX A6000 / A40
	8×22B	80 GB	2×A6000 or 2×A100 80GB

As models exceed 30-70B parameters, VRAM demands jump into the 20–50 GB range, and ultra-large models (400B–700B) require 2000-400+ GB, making HPC clusters mandatory.

But size isn't everything

LLMs in practice reflect capability, which emerges from a combination of scale, data, architecture, and optimization, not just from raw size.

Factors that affect LLM capability

- **Training Data Quality and Diversity**
  - Models trained on high-quality, well-filtered, diverse corpora can outperform a much larger one trained on noisy or narrow text.
- **Architectural Sophistication**
  - Architecture determines how well a model uses its parameters e.g. Mixture-of-Experts.
- **Context Window**
  - Larger context windows (e.g., 128k or 1M tokens) enable richer reasoning and better memory — often more impactful than adding parameters.
- **Training Objectives**
  - Auxiliary training objectives (e.g., contrastive learning, reinforcement fine-tuning, retrieval-augmented pretraining) enhance reasoning and factual recall.

Questions?