Introduction to transformers and large language models

From learnable functions to language models

Théo Stoskopf
Summer school “Proof assistants and applications” 2026

Five questions for today

1. What is a transformer?
2. How do we model language?
3. How does next-token training become text generation?
4. How does scale change what these models can do?
5. Why are formal proofs an especially interesting application?

What is a deep-learning architecture?

An architecture is the structure of a parameterized function. It specifies:

- how information flows;
- where the adjustable parameters θ occur.

It does not specify the final numerical values of those parameters.

The parameters are real numbers

$$\theta = (\theta_1, \ldots, \theta_n) \in \mathbb{R}^n$$

The architecture chooses the form of f_θ. Training chooses values for θ using data.

architecture + fitted parameters = trained model

The simplest example: fitting a line

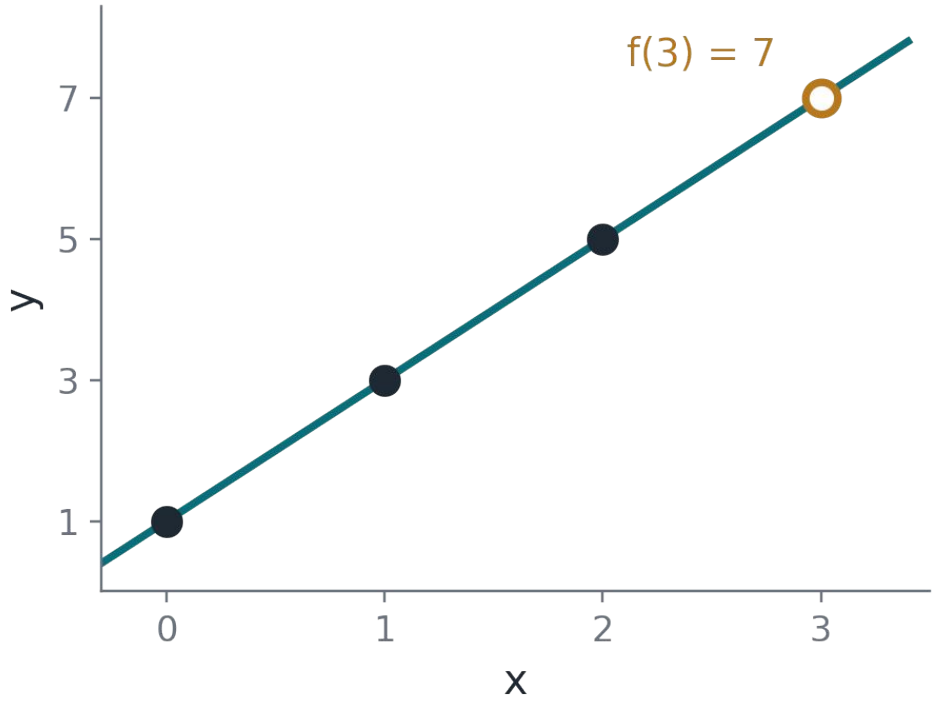
Choose the architecture

$$f_{a,b}(x) = ax + b, \qquad \theta = (a, b) \in \mathbb{R}^2$$

Given the observations

x	target y
0	1
1	3
2	5

fitting gives **a = 2** and **b = 1**.



The fitted line also predicts a point it has never seen:

$$f(3) = 2 \times 3 + 1 = 7$$

This is the goal: generalization (not memorization)

Fitting means minimizing an error

For data (x_i, y_i), define a loss:

$$\mathcal{L}(a,b) = \frac{1}{N}\sum_{i=1}^{N}\big(f_{a,b}(x_i) - y_i\big)^2$$

Training searches for

$$(a^\star, b^\star) \approx \arg\min_{a,b}\ \mathcal{L}(a,b)$$

- the loss is a function of the parameters, the data are fixed;
- "≈": we do not need to find the exact minimum
- a low loss on unseen data is what we want.

loss
low ↔ good fit
high ↔ poor fit

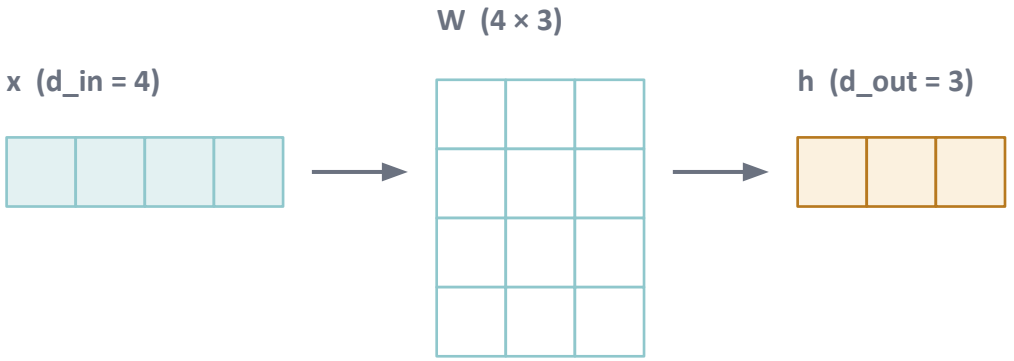
training = search over θ
guided by the loss

From a scalar line to a linear layer

$$h = xW + b$$

$$x \in \mathbb{R}^{d_{\text{in}}}, \qquad W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}, \qquad b,\ h \in \mathbb{R}^{d_{\text{out}}}$$

- x: input vector;
- W: learned matrix, b: learned bias;
- h: output vector



every input coordinate
contributes to every output

Depth alone is not enough: nonlinearities

Stacking two linear layers…

$$h = xW_1 + b_1, \quad y = hW_2 + b_2$$

collapses into a single one:

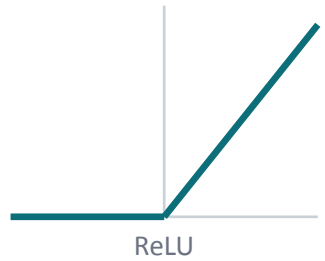
$$y = x\,(W_1W_2) + (b_1W_2 + b_2)$$

…unless a nonlinearity σ sits in between

$$h = \sigma(xW_1 + b_1), \quad y = hW_2 + b_2$$

Now the composition cannot be replaced by one matrix and one bias.

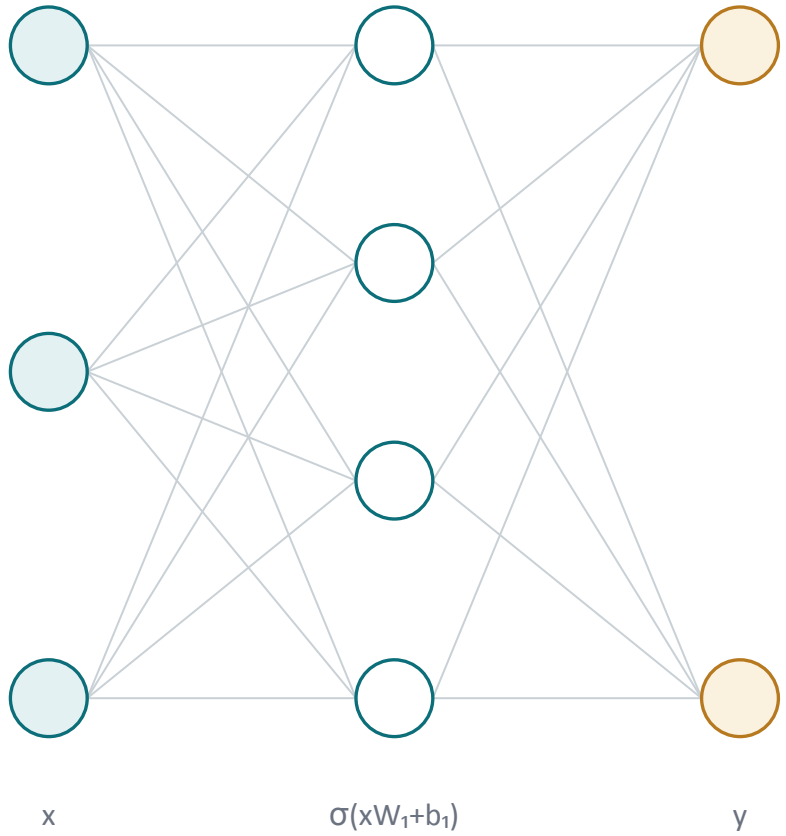
$$\mathrm{ReLU}(z) = \max(0, z)$$



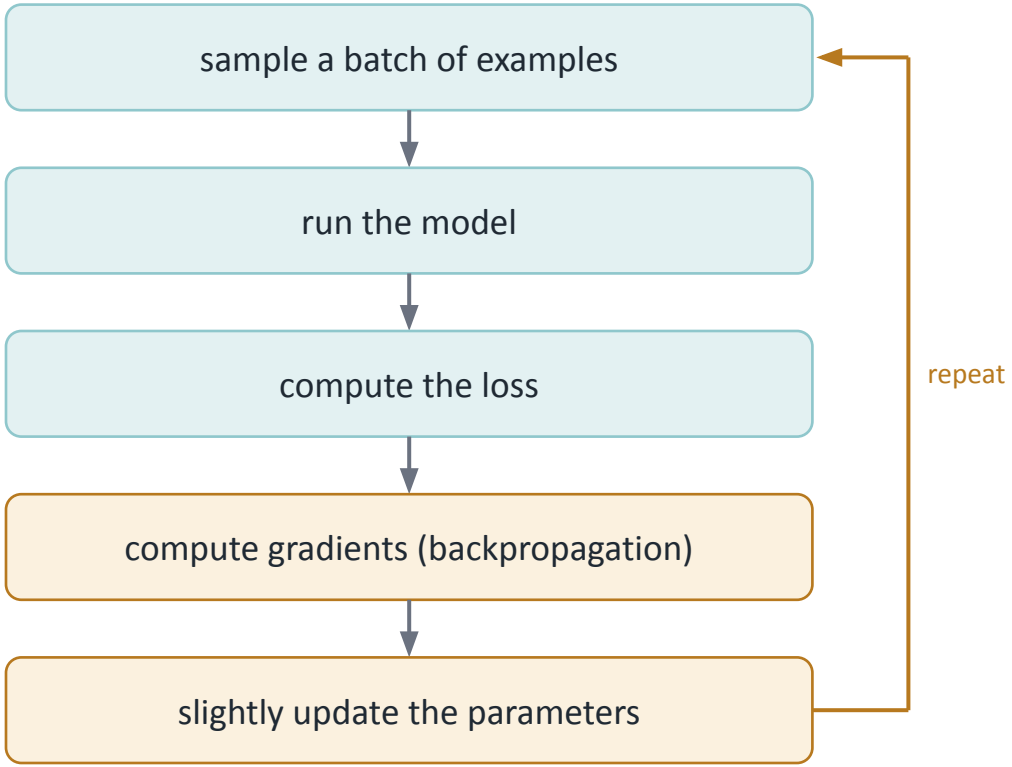
A multilayer perceptron (MLP)

$$f_\theta(x) = \sigma(xW_1 + b_1)\,W_2 + b_2, \qquad \theta = (W_1, b_1, W_2, b_2)$$

- the simplest deep architecture: linear → nonlinear → linear;
- more layers, wider layers: more expressive functions;

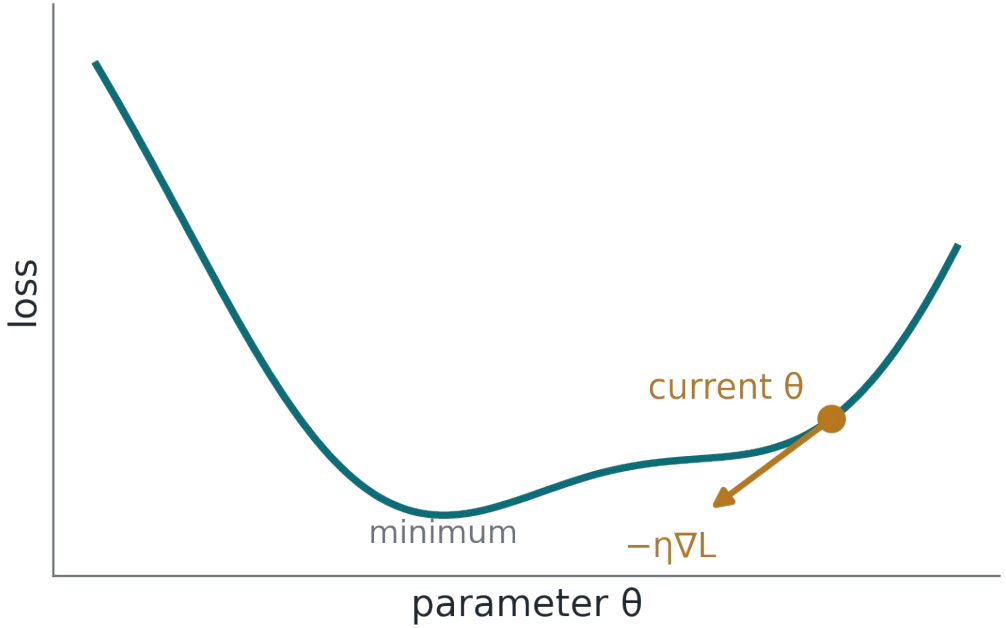


How are the parameters learned?



One iteration is called a step.

Loss landscape



With billions of parameters the landscape has billions of dimensions; training only sees local information at the current point.

The gradient measures local sensitivity

$$\nabla_\theta \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \theta_1}, \ldots, \frac{\partial \mathcal{L}}{\partial \theta_n} \right)$$

The gradient points toward the steepest local increase of the loss.

Gradient descent takes a local step

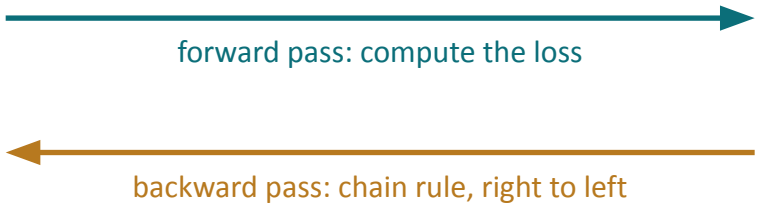
$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$$

- $-\nabla_\theta \mathcal{L}$: a local direction of decrease;
- η: the learning rate, controlling the step size;
- not a closed-form solution: an iterative process repeated many times.

Backpropagation computes all the derivatives

A neural network is a composition of simple functions:

$$x \xrightarrow{f_1} h_1 \xrightarrow{f_2} h_2 \xrightarrow{f_3} \widehat{y} \xrightarrow{\ell} \text{loss}$$

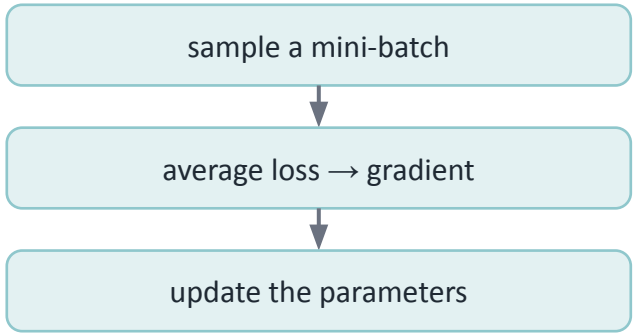


- applies the chain rule from right to left;
- reuses intermediate computations: all derivatives for roughly the cost of one forward pass;
- done automatically by frameworks such as PyTorch or JAX.

Making gradient descent work at scale

1 · Stochastic gradients

The corpus may contain trillions of tokens. Instead of differentiating through all of it:

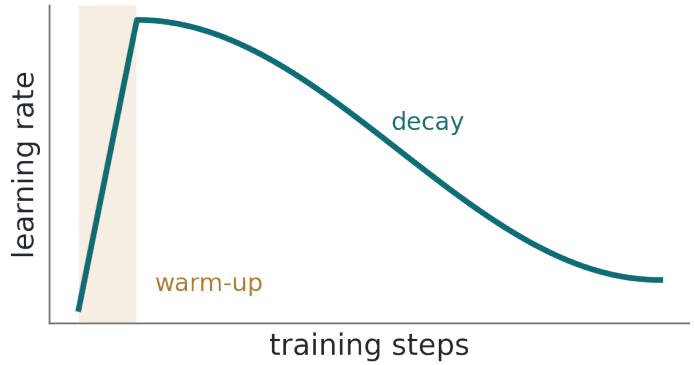


2 · Optimizers

Optimizer	Extra state	Intuition
SGD	none	follow the gradient
Momentum	1 average	smooth the direction
AdamW	2 averages	adapt the scale per coordinate
Muon	1 average	orthogonalize the update so that no direction vanishes

Backpropagation computes gradients; the optimizer applies parameter updates.

3 · Learning-rate schedule



warm-up avoids aggressive early updates; decay makes later updates smaller.

Deep learning in five words

Word	Meaning
Architecture	structure of the computation
Parameters	adjustable numbers in that structure
Model	architecture with particular parameter values
Training	process that fits the parameters
Inference	use of the trained model on an input

What about language models?

a learnable function whose input is text

Two obstacles between text and a network

A linear layer eats a fixed-size vector of numbers: $h = xW + b$. Text fails on both counts.

Obstacle 1 — text is not numbers

```
"The square of an even integer is"  → ℝ?
```

characters and words must become real-valued vectors before any matrix can touch them

→ **tokenization + embedding (next slides)**

Obstacle 2 — text is a sequence

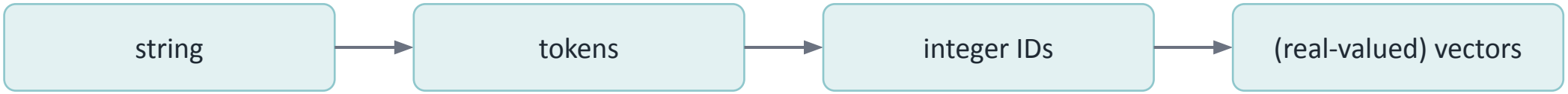
- the length varies from sentence to sentence;
- the order carries meaning;
- how one token is processed depends on the others.

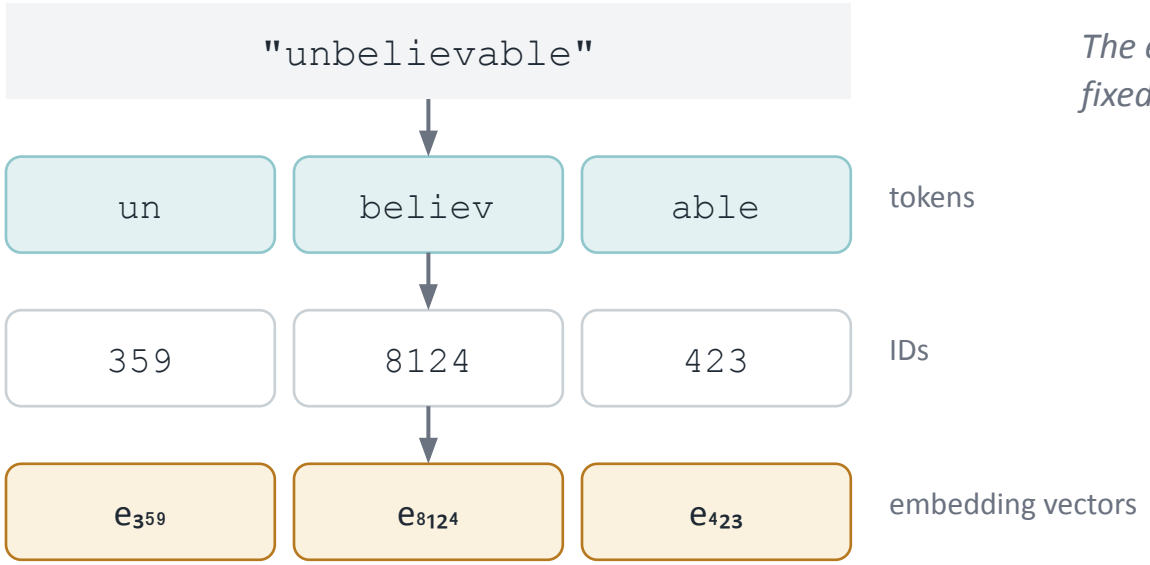
→ **an architecture question**

The first obstacle is solved by preprocessing. The second need a specific architecture.

Tokenization: quantizing text into a finite vocabulary

A network computes with real-valued vectors; text is a string. So the string is first cut into units from a fixed, finite vocabulary and each unit becomes an integer ID.





The exact split depends on the tokenizer, chosen and fixed before training.

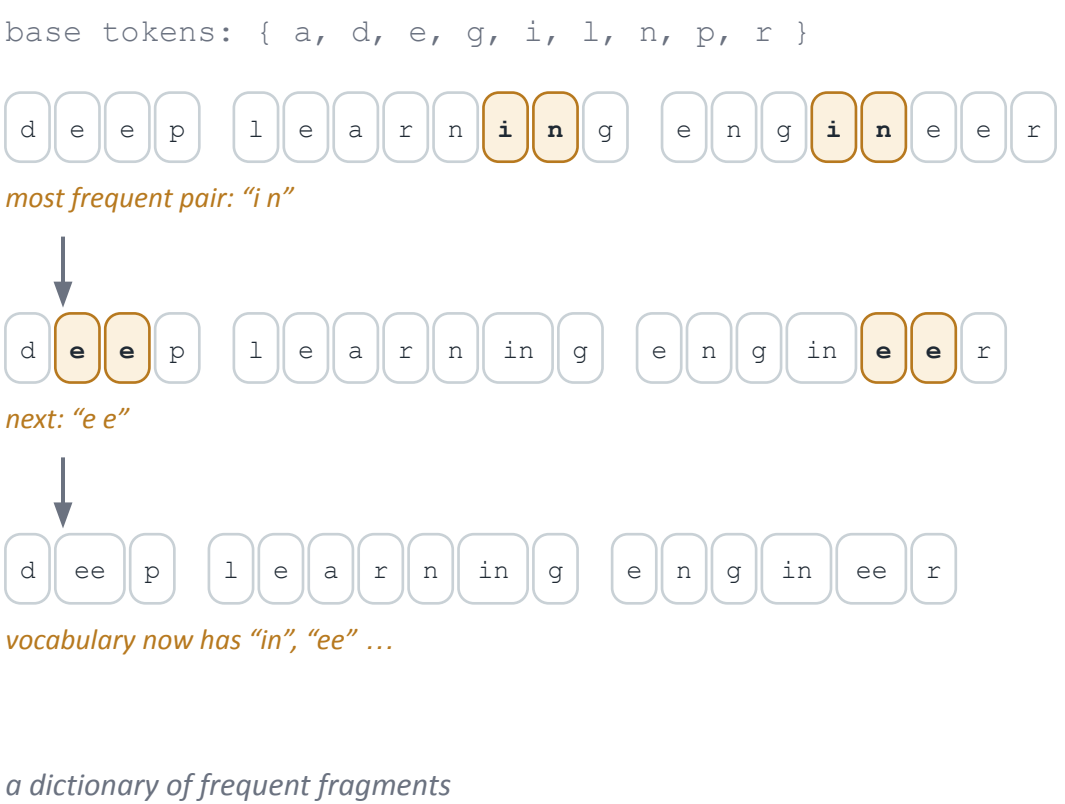
Subword tokens, and how they are learned

Unit	Advantage	Cost
Character / byte	represents any string	long sequences
Whole word	short sequences	huge vocabulary
Token	reuses common fragments	boundaries can look strange

Byte Pair Encoding (BPE)

1. start from bytes or characters;
2. count adjacent pairs;
3. merge the most frequent pair into a new token;
4. repeat until the desired vocabulary size.

Example: “deep learning engineer”



a dictionary of frequent fragments

The vocabulary: a finite list of tokens

Token	ID
`A`	0
`B`	1
`C`	2
…	…
`ers`	382
`on`	383
`ap`	384
…	…
`.destroyAllWindows`	84227
`iterating`	84228
`Reaction`	84229
…	…

- a token is just an index into this list — the model never sees characters;
- typical sizes ~100k–250k today;
- the list is frozen before training: changing it would change the meaning of every ID.

For formal data

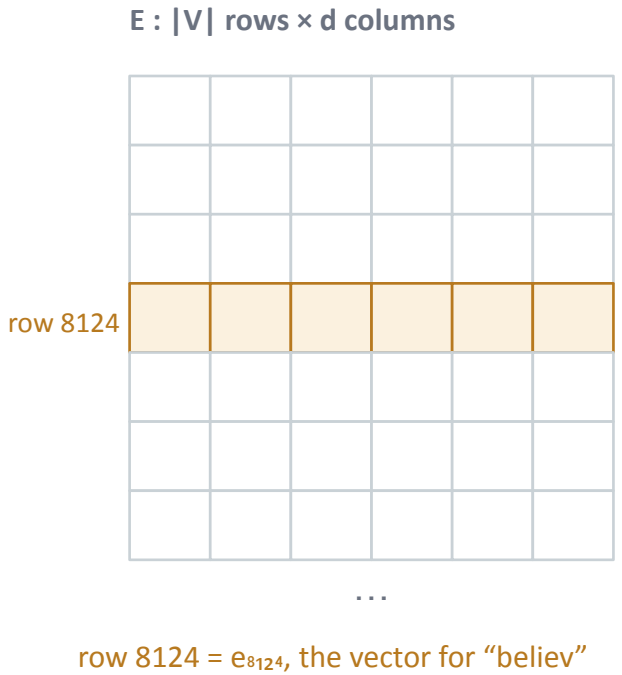
```
continuousAt   →  continuous | At
∀ ε > 0        →  ∀ | ε | > | 0
```

identifiers split in odd places, Unicode symbols cost several tokens, and the effective length of a proof state depends on the tokenizer.

Embeddings are learned representations

$$E \in \mathbb{R}^{|\mathcal{V}| \times d}, \qquad h_i^{(0)} = E[x_i]$$

- one learned row per token in the vocabulary;
- the same embedding is used every time the token appears;
- learned by gradient descent like every other parameter.



The tokenizer is fixed before training: changing it afterwards would change the meaning of token IDs and invalidate the learned embeddings.

Modelling sequences: why not a linear layer?

Concatenate the vectors and use h = xW + b ?

`dog` `bites` `man` — *same tokens, different meaning*

`man` `bites` `dog` — *same tokens, different meaning*

`We` `sat` `on` `the` `bank` `of` `the` `river` `.`

fixed-size input of a linear layer:

slot 1	slot 2	slot 3	slot 4	slot 5	slot 6	→ **W**

9 tokens do not fit; 3 tokens leave slots empty; slot 5 has its own weights, unrelated to slot 2

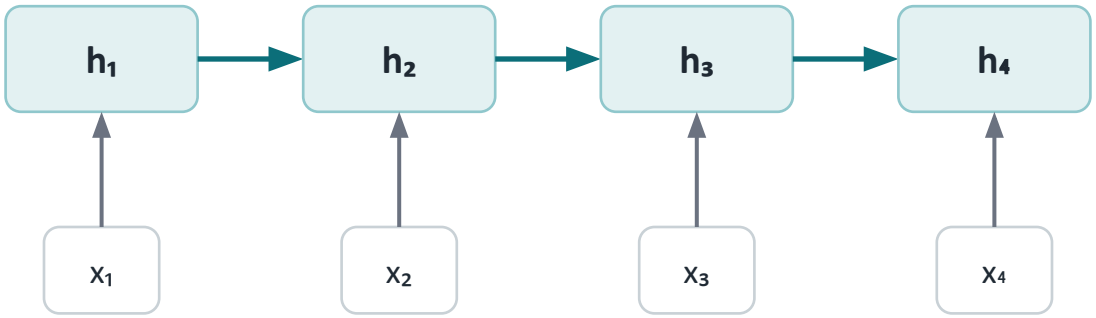
What a sequence model needs

any length	sentences, programs and proofs have no fixed size
order	the same tokens in another order mean something else
shared computation	what is learned at one position must apply at every position
content-dependent interaction	how a token is processed must depend on which other tokens are present

A linear layer offers none of these. We need an architecture built for sequences.

A first answer: recurrent networks

$$h_t = f_\theta(h_{t-1}, x_t)$$

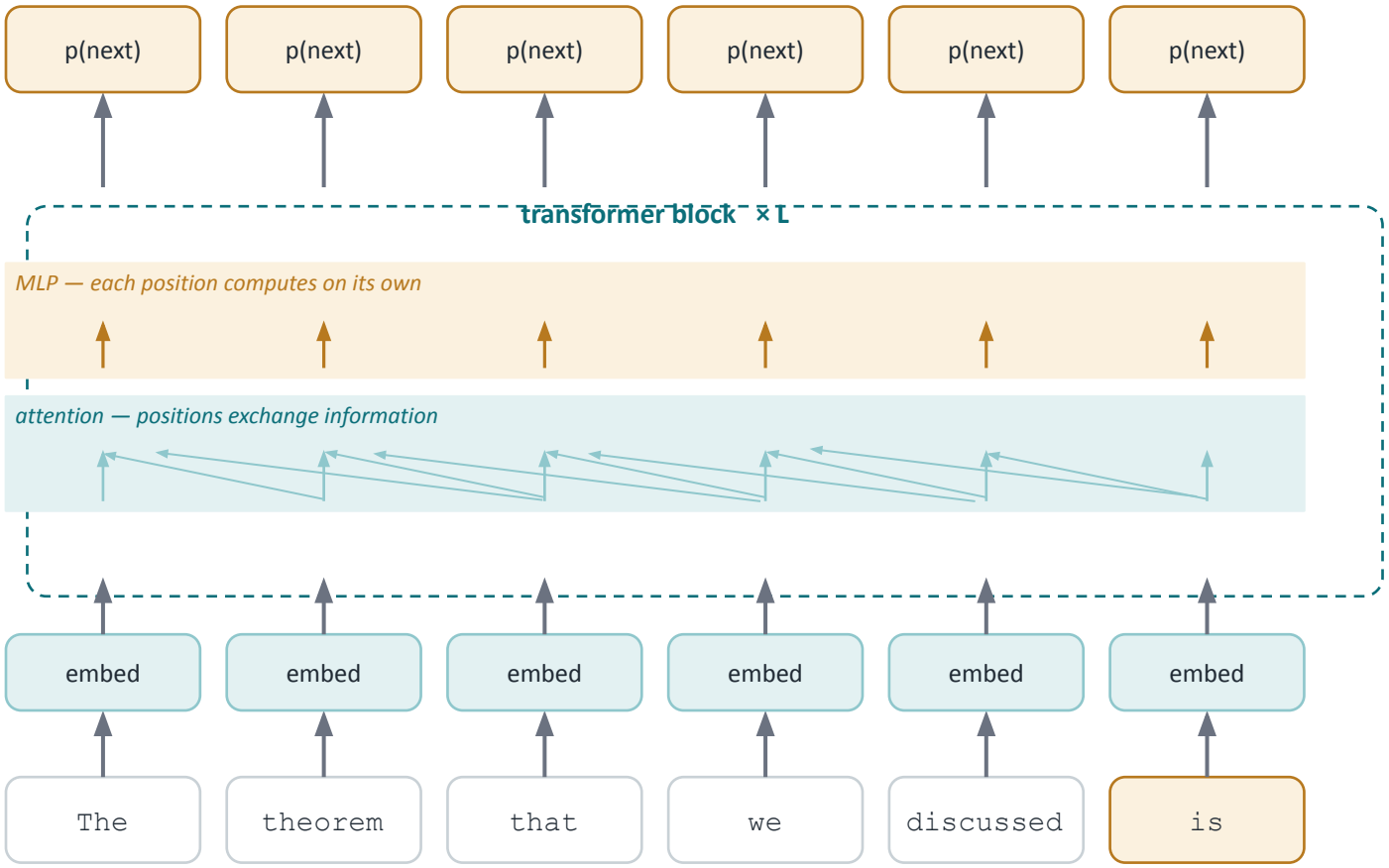


The limitation: information travels sequentially

To compute h□ we first need h□-1. Consequences:

- positions cannot be processed in parallel during training;
- distant tokens are connected by a long computational path;
- a fixed-size state must continually summarize the past.

The transformer's answer



all positions at once — one column per token, the same weights everywhere — any length, shared computation, fully parallel

attention — each position gathers information from the others; how much from whom depends on content

MLP — each position then computes on what it has gathered

stack and predict — L blocks; every position ends with a distribution over its next token

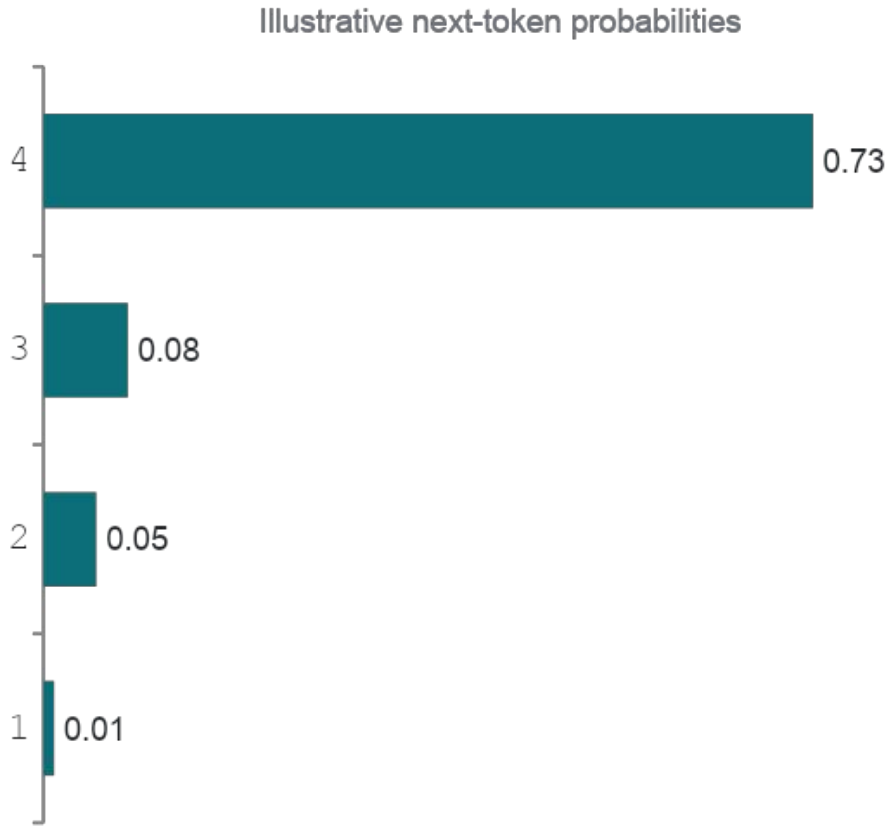
Bahdanau, Cho, Bengio (2014) arXiv:1409.0473 — attention inside RNNs · Vaswani et al. (2017) "Attention is all you need", arXiv:1706.03762

A language model makes a prediction

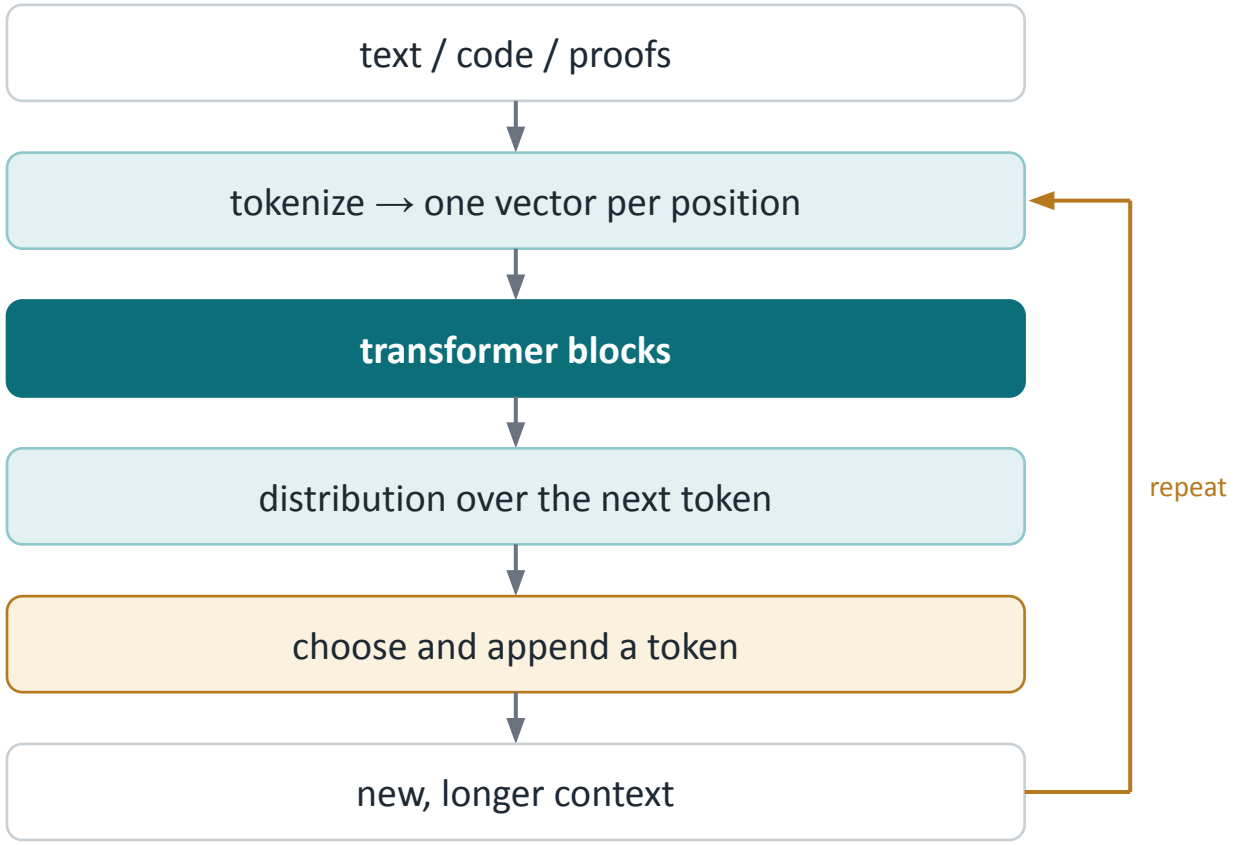
```
The square of an even integer is …
```

The model outputs a probability distribution over the next token — not a complete answer.

How far can we get by repeating this simple operation?

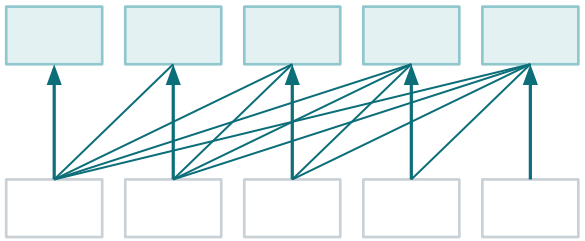


The language-model loop



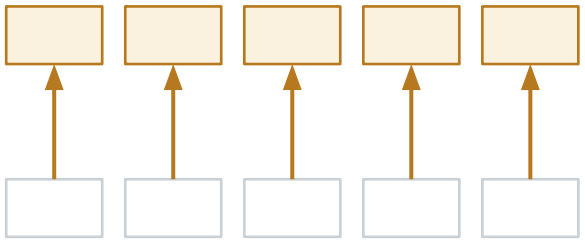
Inside a transformer block: the sub-blocks

attention: across positions

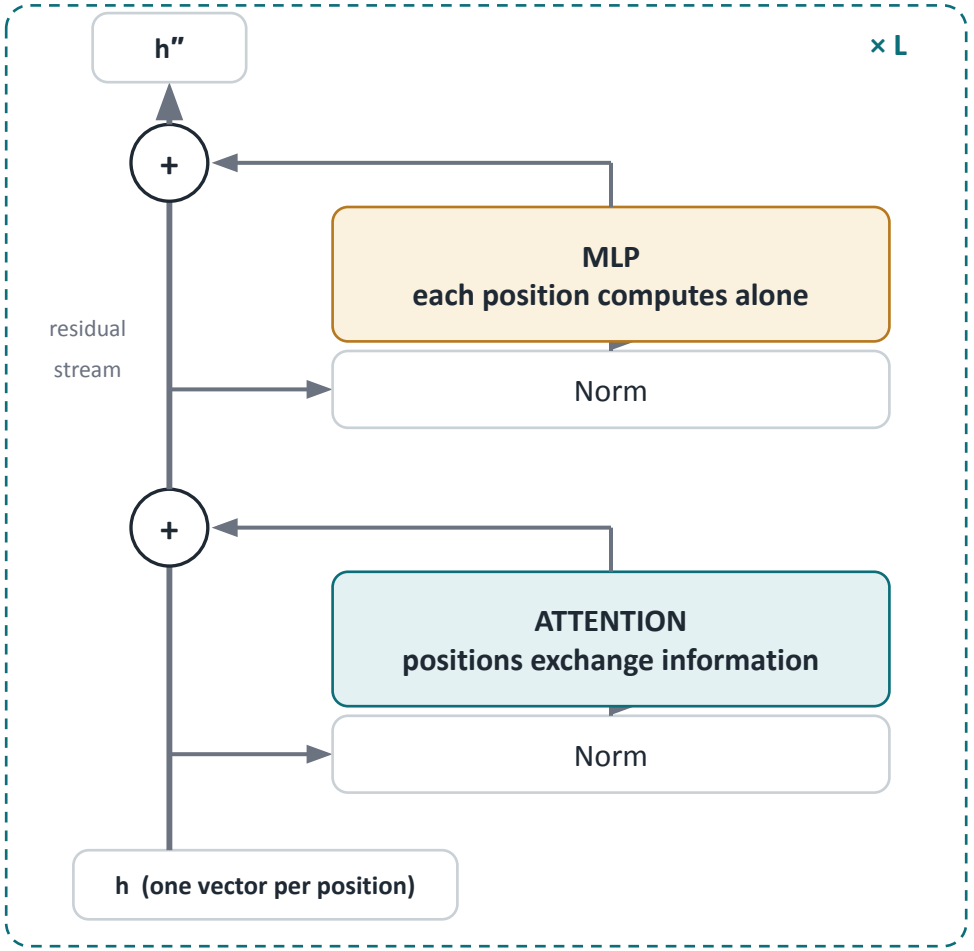


every position reads from the others (here: from the past)

MLP: within each position



same function, applied column by column

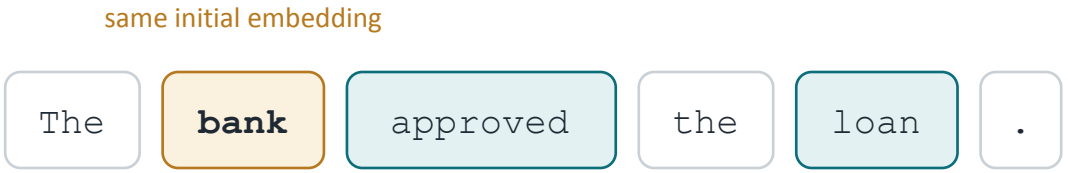


Plumbing

residual connection
h ← h + F(h): the sub-block adds a correction instead of replacing h

normalization
keeps the scale of h stable so that deep stacks train

What attention has to achieve: context



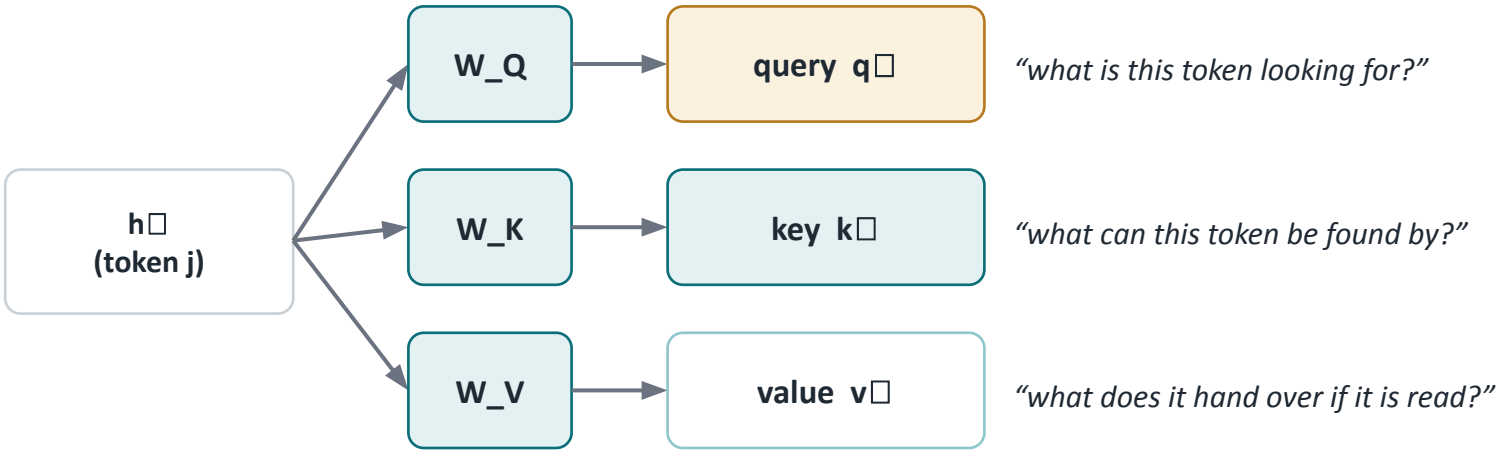
pulls in “approved”, “loan” → financial sense

pulls in “sat”, “river” → geographical sense

- before the first block, a token's vector depends only on the token itself;
- after attention, it depends on the visible context: the two “bank” vectors diverge;
- attention must therefore decide, for each position, which other positions to read from, and what to take from them.

First, the vocabulary: three vectors per token

From the current vector h□ of every token, three learned matrices compute three small vectors:



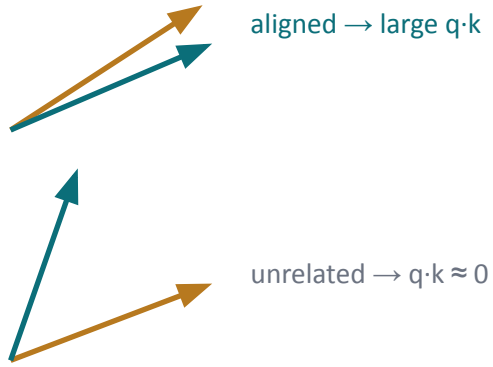
“what is this token looking for?”

“what can this token be found by?”

“what does it hand over if it is read?”

learned matrices (the same three at every position)

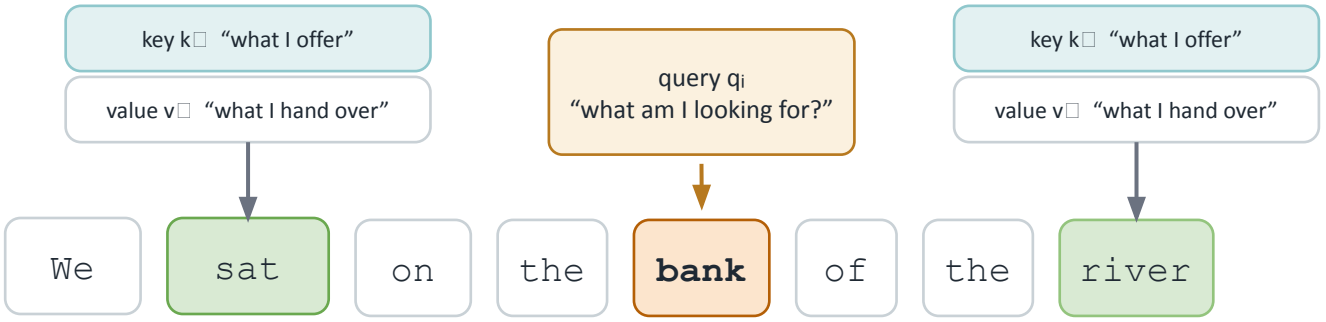
Interaction = dot product



$q_i \cdot k_□$ = “how relevant is j for i?”

Nothing is hand-written: W_Q, W_K, W_V are learned by gradient descent

Attention in one sentence



every position publishes its key and its value; the reading position emits its query

1 · compare → **2 · normalize** → **3 · retrieve**

query against every key ⟶ a relevance score per position

scores ⟶ weights that are ≥ 0 and sum to 1

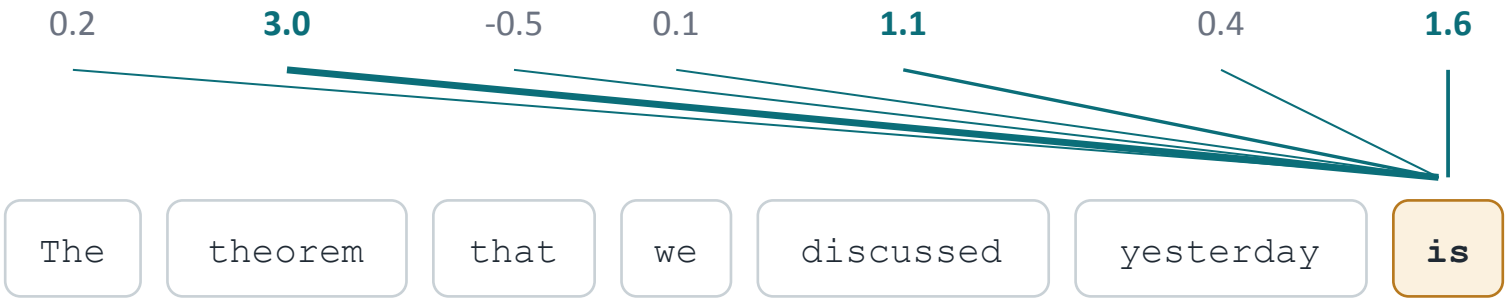
output = weighted mix of the values

Like a dictionary lookup d[key]

Step 1 · Compare

One dot product per visible position:

$$s_j = \frac{q_{\text{is}} \cdot k_j}{\sqrt{d_k}}$$



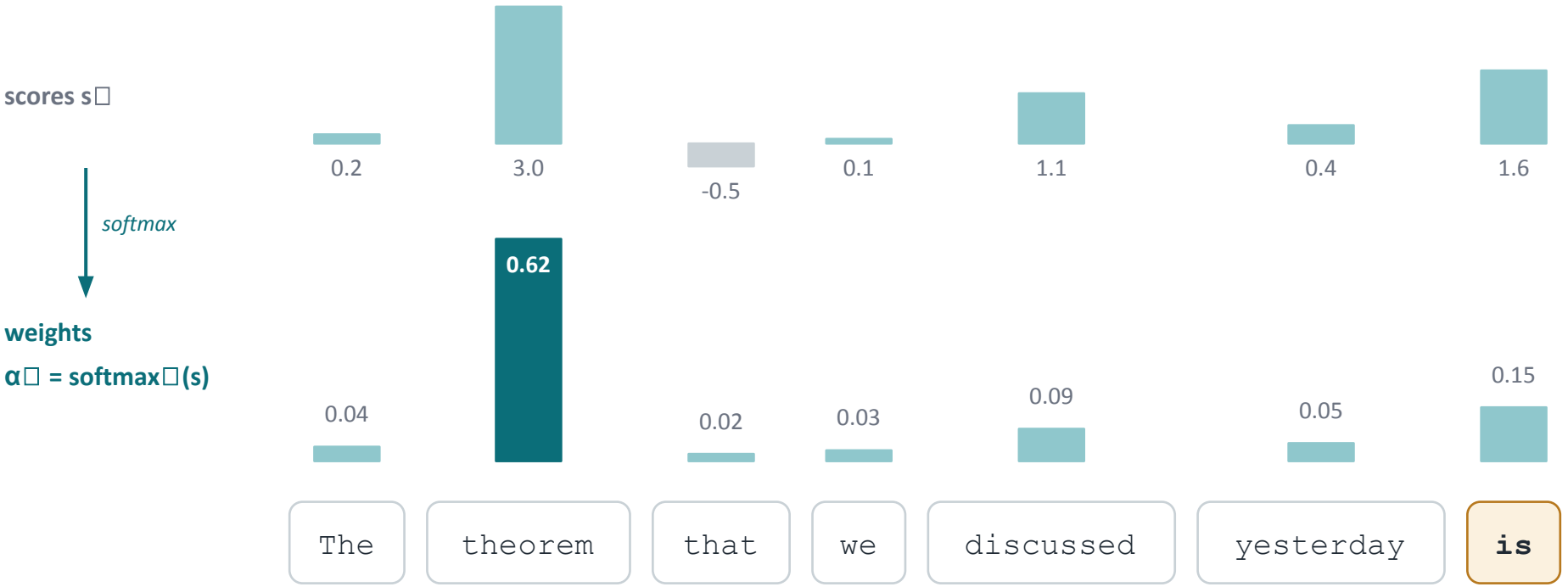
the query of “is”, against every key (including its own)

One number per position: how relevant is j for “is”?

Step 2 · Normalize

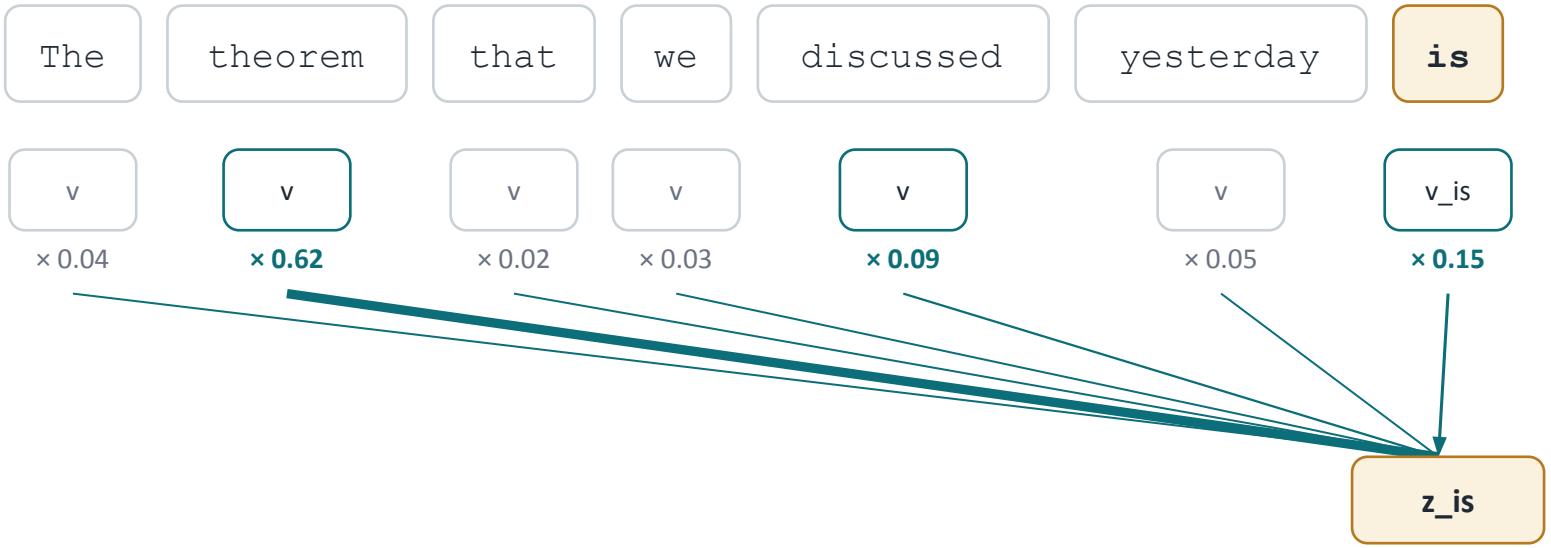
Softmax turns the scores into positive weights, and summing to 1:

$$\alpha_j = \frac{e^{s_j}}{\sum_{j'} e^{s_{j'}}} \qquad \alpha_j \geq 0, \quad \sum_j \alpha_j = 1$$



Step 3 · Retrieve

The output is the weighted mix of the values and it is added to the reading position:

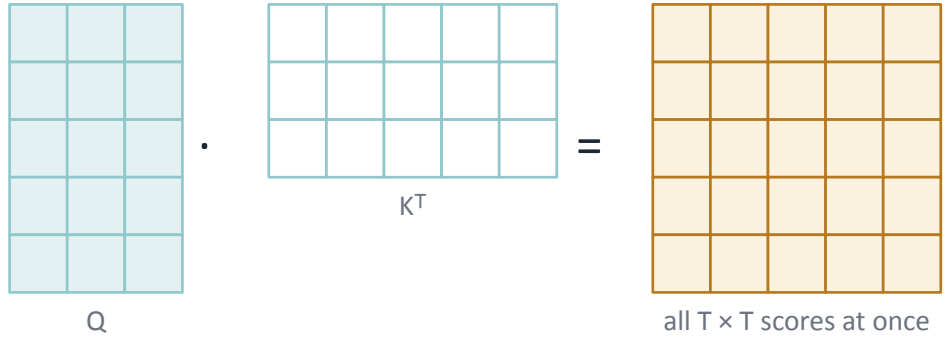


z_is = 0.62 v_theorem + 0.15 v_is + 0.09 v_discussed + … → "is" now knows its subject is a theorem

Attention = soft, content-dependent, differentiable retrieval (run for every position at once, with the same matrices)

All of this is one highly parallelizable operation

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V$$



$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Causal attention prevents cheating

When the representation after x_i is used to predict x_{i+1}, it may see only $x_1 \ldots x_i$.

Future scores are set to $-\infty$ before the softmax, so they receive weight 0:

$$M_{ij} = \begin{cases} 0 & j \le i \\ -\infty & j > i \end{cases}$$

keys j →

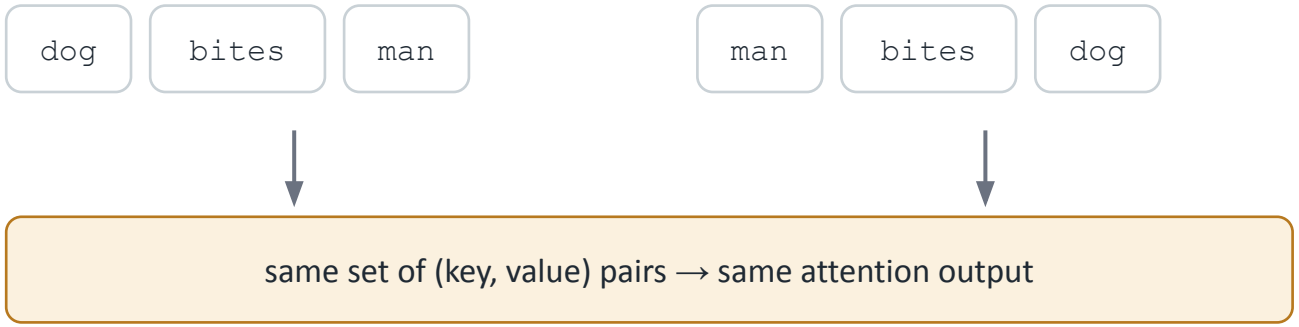
queries i ↓

0	−∞	−∞	−∞	−∞
0	0	−∞	−∞	−∞
0	0	0	−∞	−∞
0	0	0	0	−∞
0	0	0	0	0

row i: position i may attend to positions 1 … i

One problem: attention ignores order

Nothing in compare · normalize · retrieve uses the position of j



attention is permutation-invariant: but requirement 2 of slide "why not a linear layer" was: order matters!

The fix: inject position (very briefly)

- **rotary (RoPE), today's default:** rotate q and k by a position-dependent angle → q·k depends on the distance i − j;
- **older:** add a learned or sinusoidal position vector to each embedding.

Attention brings content-dependence; positional encoding brings order.

Quadratic but parallel at training time

The cost: T^2 interactions

T = 6 ⟶ 21 pairs

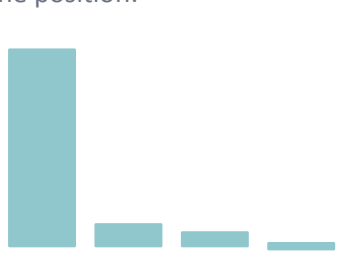
T = 9 ⟶ 45 pairs

double the context ≈ 4× the comparisons (and the memory)

The gift: training needs no loop

- at training time the targets are already in the sequence;
- one masked forward pass ⟶ a prediction and a loss at every position, simultaneously;
- no sequential generation anywhere in training.

at one position:



predicted p

Dirac on the observed token

$\ell = -\log p(\text{target})$

Quadratic, but parallel, this is what made web-scale training practical.

Living with the quadratic cost

Idea	Year	In three words
KV cache	2019 →	remember past keys/values
FlashAttention 1–4	2022–26	exact, GPU-friendly kernels
MQA / GQA	2019 / 2023	share keys across heads
Sliding-window	2020–23	attend locally only
MLA	2024	compress keys/values
Learned sparse (NSA, DSA)	2025	skip useless pairs
Linear attention (KDA, …)	2025–26	recurrent O(T) form

Frontier models are hybrids:

GLM-5.3-Flash (Aug 2026)
34 linear-attention layers
+ 11 sparse-MLA layers
1M-token context

Dao et al. (2022) FlashAttention · Ainslie et al. (2023) GQA · DeepSeek-V2 (2024) MLA · DeepSeek NSA (2025) · Kimi Linear (2025) KDA · GLM-5.3-Flash (2026)

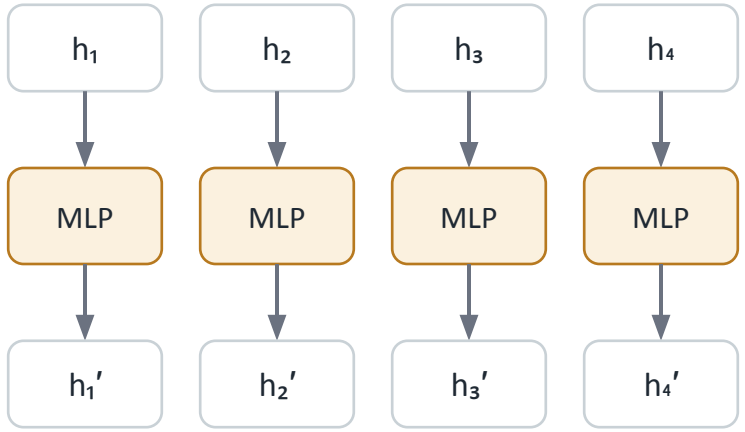
The other half of a transformer block: the MLP

$$\mathrm{MLP}(h) = \sigma(hW_1 + b_1)\,W_2 + b_2$$

Applied to each position independently, after positions have exchanged information.

attention
moves information between positions

MLP
computes on the information at each position



same weights, applied position by position

- separate parameters in every layer, but the same parameters at every position;
- hidden width typically ≈ 4× the model width: most LLM parameters live here.

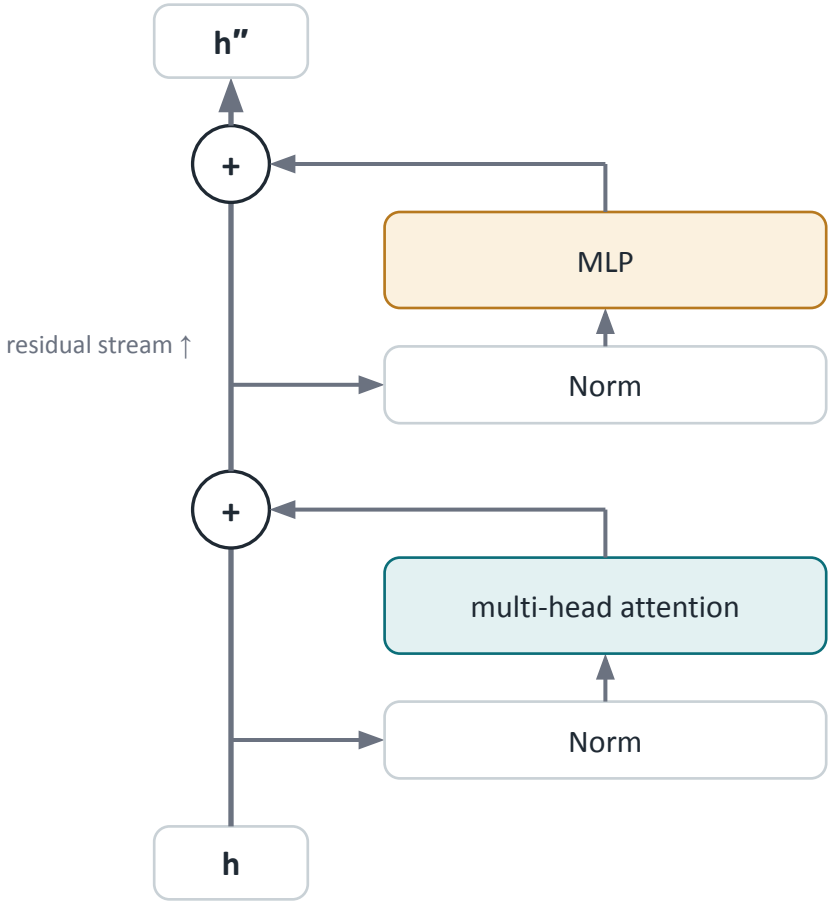
One modern transformer block

$$h' = h + \mathrm{MHA}\big(\mathrm{Norm}(h)\big)$$

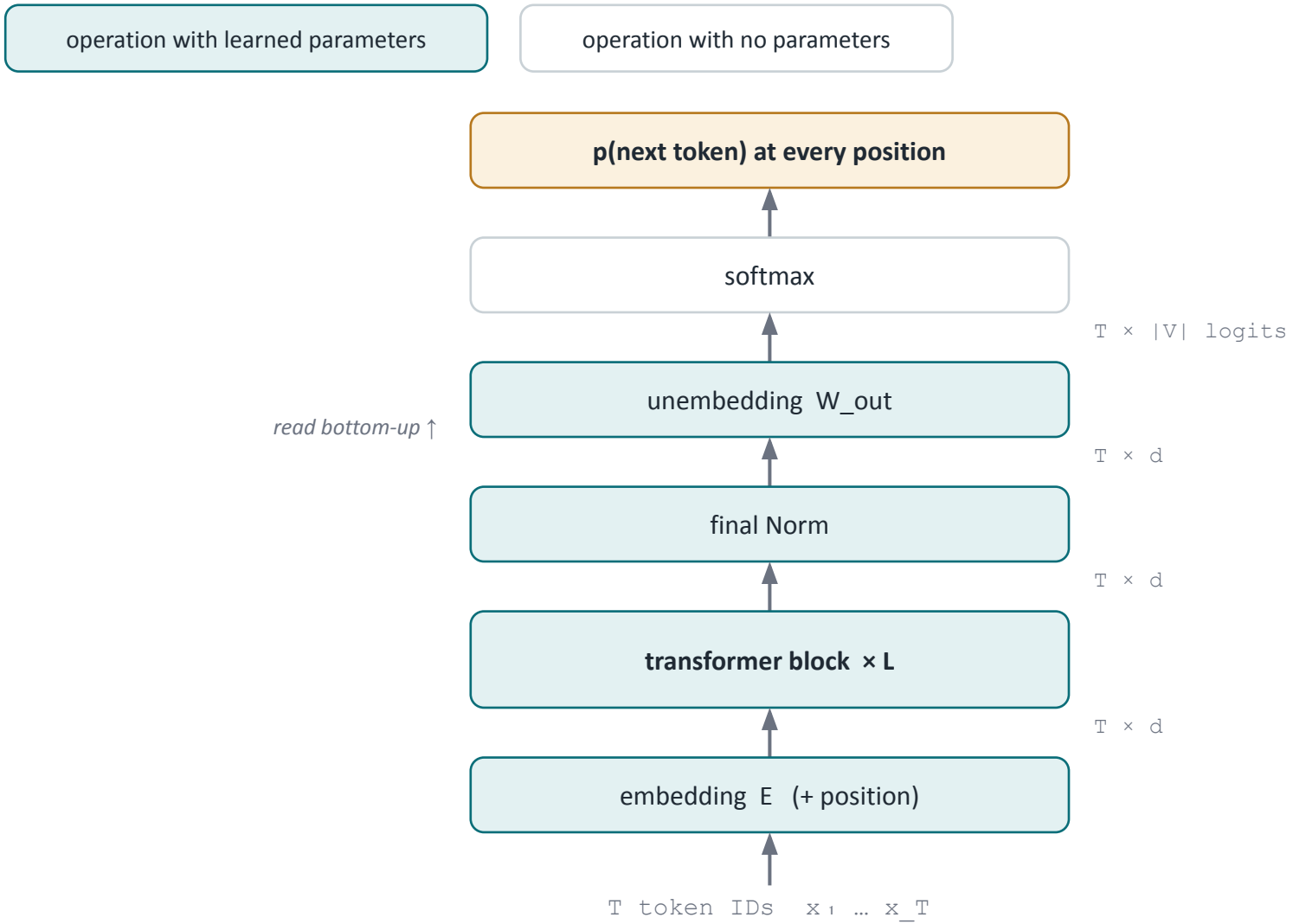
$$h'' = h' + \mathrm{MLP}\big(\mathrm{Norm}(h')\big)$$

- **attention** communicates across positions;
- **MLP** transforms each position;
- **residual connections** preserve an information path;
- **normalization** stabilizes range of input

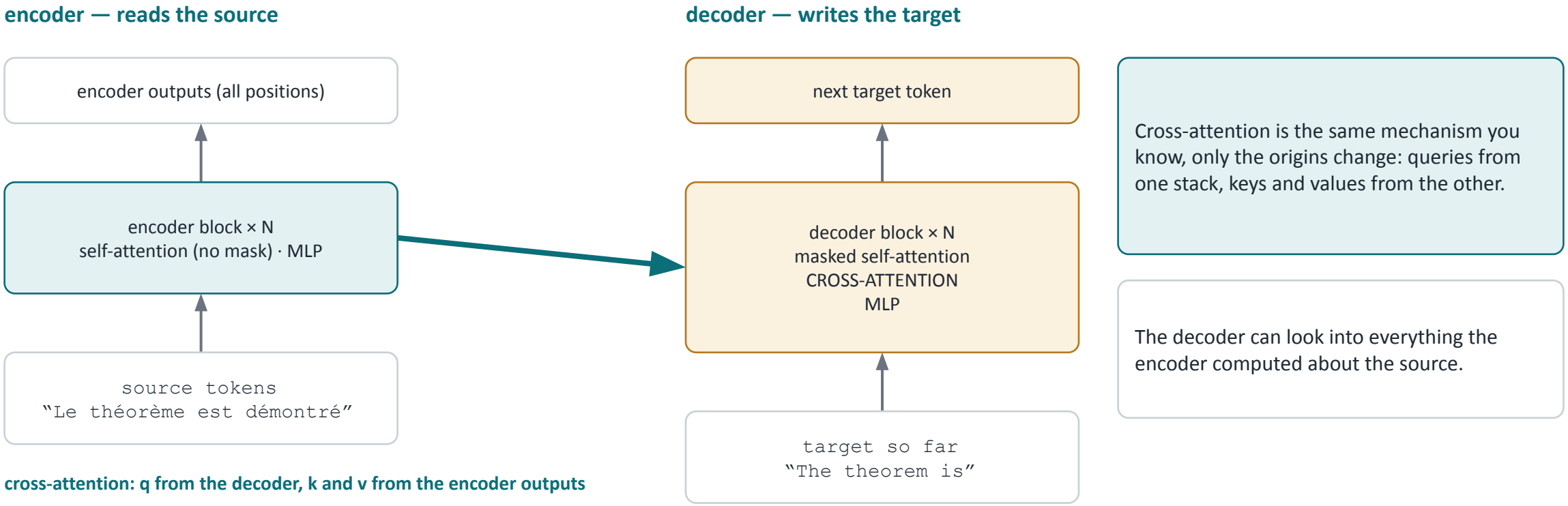
Stack this block L times.



The whole model in one picture

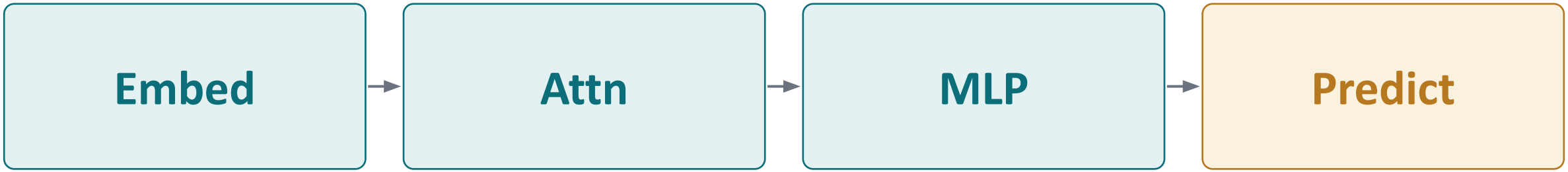


Where transformers come from: translation



Today's LLMs keep only the decoder stack, prompt and answer form one sequence. BERT-style models keep only the encoder (classification, retrieval).

Checkpoint



each token as a vector

to relevant visible positions

at each position with an MLP

a distribution for the next token

Repeat attention and transformation through many layers.

Repeat prediction to generate a sequence.

How to generate sentences?

From scores to a distribution: softmax

From a context x<t, the model produces one score for every token in the vocabulary:

$$z \in \mathbb{R}^{|\mathcal{V}|}$$

Softmax turns the scores into probabilities:

$$p_\theta(x_t = j \mid x_{<t}) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Why predict a distribution?

```
The proof is …
complete / correct / by induction / …
```

- a differentiable training objective;
- relative confidence between alternatives;
- several possible continuations;
- a way to trade determinism for diversity when generating.

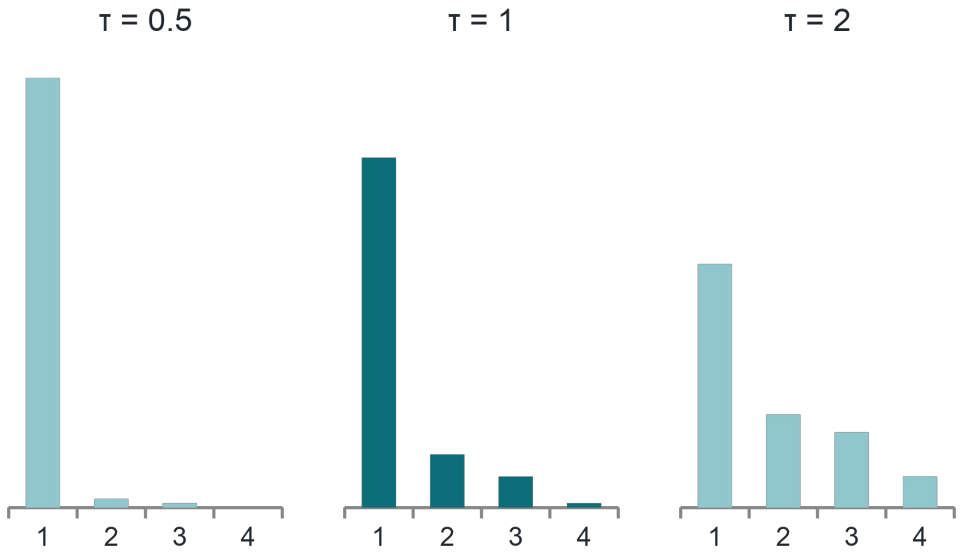
Decoding

Method	Behavior
Greedy	always select the most probable token
Sampling	randomly select according to the distribution
Top-k / top-p	sample only among plausible candidates

Temperature rescales the logits

$$p_j = \left[\mathrm{softmax}\left(z/\tau\right)\right]_j$$

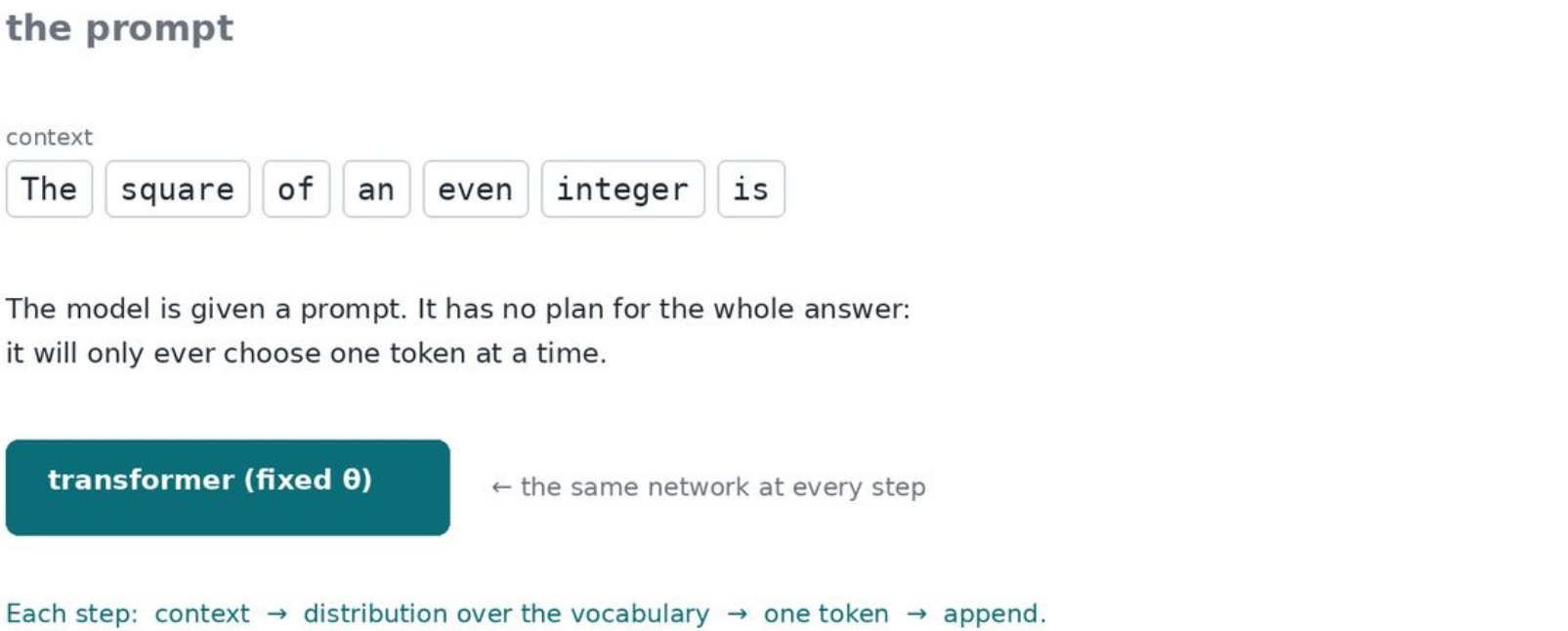
Lower T sharpens the distribution; higher T flattens it.



same logits, three temperatures

Decoding changes generated outputs, not the trained parameters.

Generation, animated



Probabilities are illustrative. Each step is one forward pass of the same fixed network.

Train at scale

The training objective: predict every next token

For a sequence $x_1 \ldots x_T$:

$$p_\theta(x_1, \ldots, x_T) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t})$$

Training minimizes the negative log-likelihood:

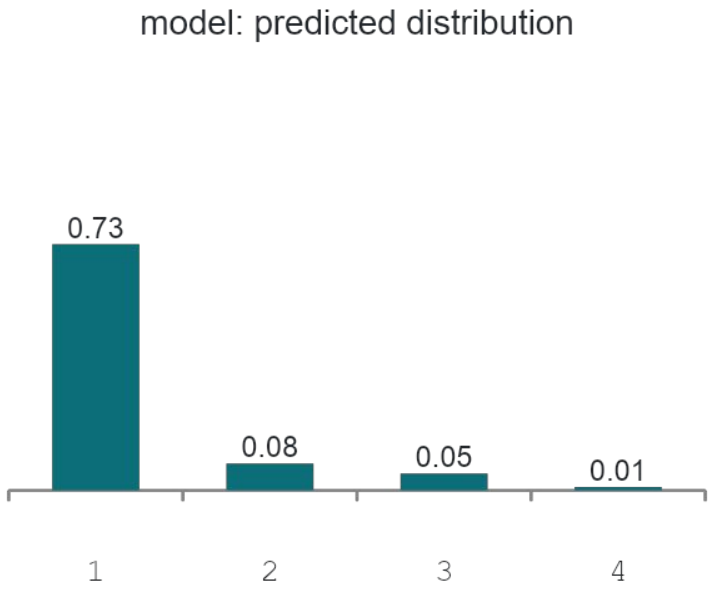
$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t})$$

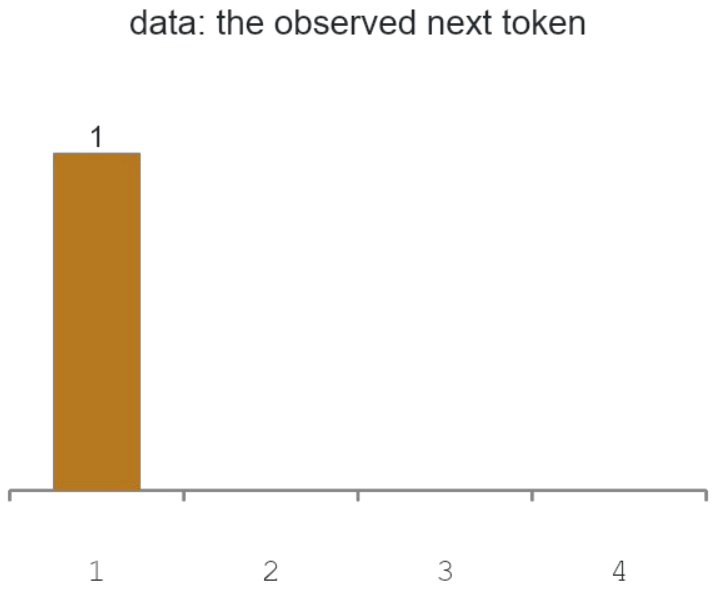
One sequence gives many training examples

```
The | proof | is | complete | .
```

Visible prefix	Target
`The`	`proof`
`The proof`	`is`
`The proof is`	`complete`
`The proof is complete`	`.`

What the loss sees at one position





loss at this position

$$\ell_t = -\log p_\theta(x_t^\star \mid x_{<t})$$

- assigned 0.73 → ℓ = 0.31
- assigned 0.05 → ℓ ≈ 3.0
- assigned 1.00 → ℓ = 0

Next token prediction

The learning signal:

> *“make the observed next token more probable”*

But accurate prediction across text, code, and mathematics rewards internal representations of:

- syntax and local conventions;
- facts and associations;
- algorithms and recurring proof patterns;
- some long-range dependencies.

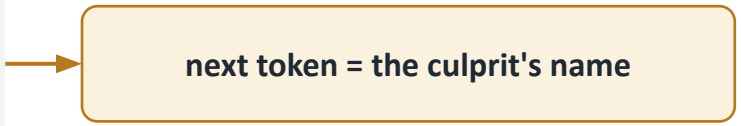
Prediction can induce useful structure.

It does not by itself guarantee validity.

Why prediction can demand understanding

Train on detective novels. At the end of the book:

```
… Poirot gathered them all in the library.
“Every clue points one way,” he said.
“The murderer is —”
```



- to put high probability on the right name, the model must have tracked every alibi, motive, timeline and contradiction across hundreds of pages;
- no shortcut predicts that word from surface statistics;
- so pushing next-token loss low enough on such text forces something that behaves like understanding of the plot.

I. Sutskever, fireside chat with J. Huang, NVIDIA GTC (2023)

Why did transformers scale so well?

- all training positions are processed in parallel;
- the computation is dominated by large matrix multiplications
- large amount of available data

The main architectural cost

$$QK^\top \in \mathbb{R}^{T\times T} \qquad \Longrightarrow \qquad O(T^2)$$

Full attention compares every pair of positions

Mitigations: better kernels, shared key/value heads, local attention, sparsity…

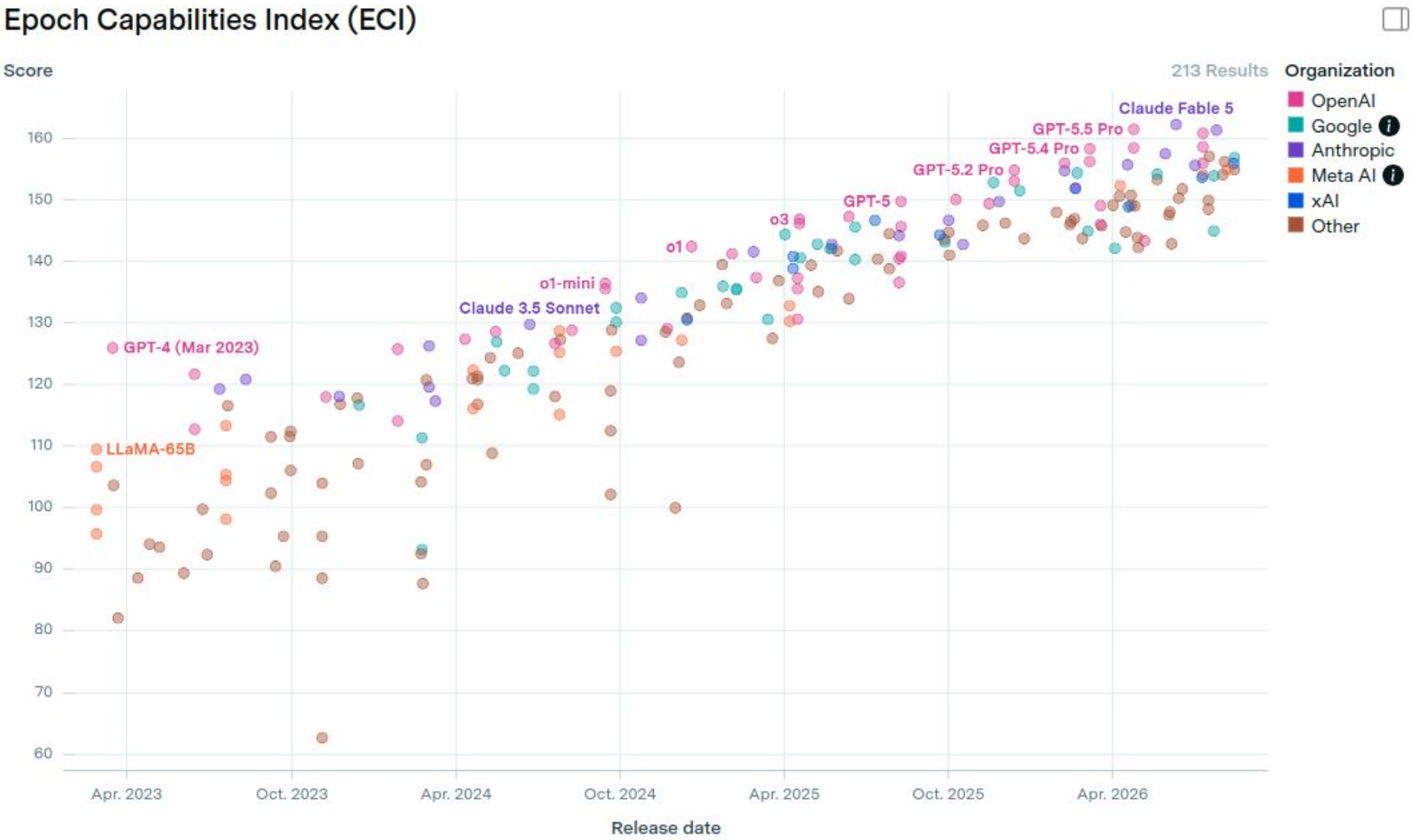
Almost the same architecture, 2019 → 2026

	GPT-2 small (2019)	GPT-3 (2020)	Llama 3.1 405B (2024)	GLM-5.3-Flash (2026)
blocks L	12	96	126	45
width d	768	12,288	16,384	4,096
attention	dense MHA	dense MHA	GQA (8 kv heads)	hybrid: 34 linear + 11 sparse MLA
experts (MoE)	—	—	—	288 + 1 shared (9 active)
vocabulary |V|	50,257	50,257	128,000	154,880
context T	1,024	2,048	128K	1M
parameters	124 M	175 B	405 B	320 B (18 B active)
training data	≈ 40 GB text	300 B tokens	15.6 T tokens	30 T tokens (multimodal)

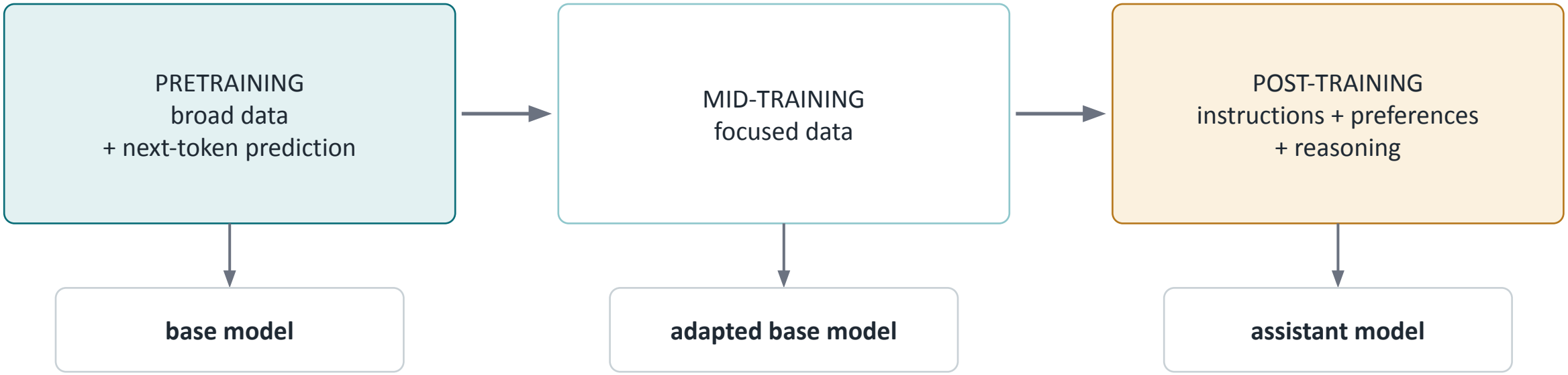
Two trends: data grew faster than parameters, and the newest models re-engineer attention itself.

Radford et al. (2019) · Brown et al. (2020) · Grattafiori et al. (2024) arXiv:2407.21783 · Z.ai, GLM-5.3-Flash model card (Aug 2026, MIT)

Benchmarks


Epoch Capabilities Index (ECI)
Score
213 Results
Organization
OpenAI
Google
Anthropic
Meta AI
xAI
Other
Claude Fable 5
GPT-5.5 Pro
GPT-5.4 Pro
GPT-5.2 Pro
GPT-5
o3
o1
o1-mini
Claude 3.5 Sonnet
GPT-4 (Mar 2023)
LLaMA-65B
160
150
140
130
120
110
100
90
80
70
60
Apr. 2023
Oct. 2023
Apr. 2024
Oct. 2024
Apr. 2025
Oct. 2025
Apr. 2026
Release date

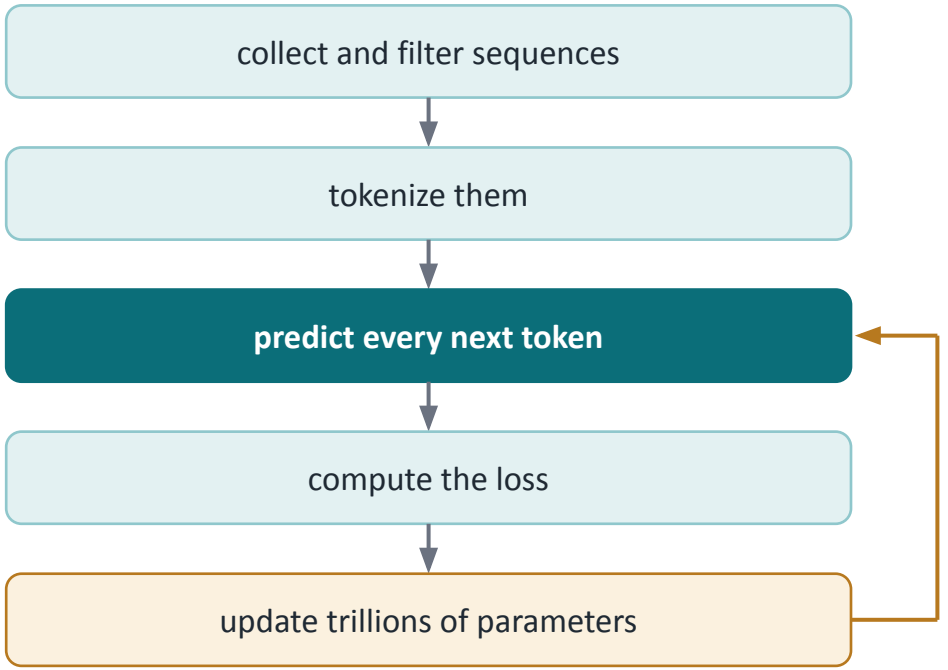
Training now has several phases



Pretraining at scale

The data pipeline is part of the model

- collection (and licensing decisions?)
- quality and domain filtering;
- deduplication;
- privacy and safety filtering;
- ordering, sampling, and tokenization.



repeat over vast datasets

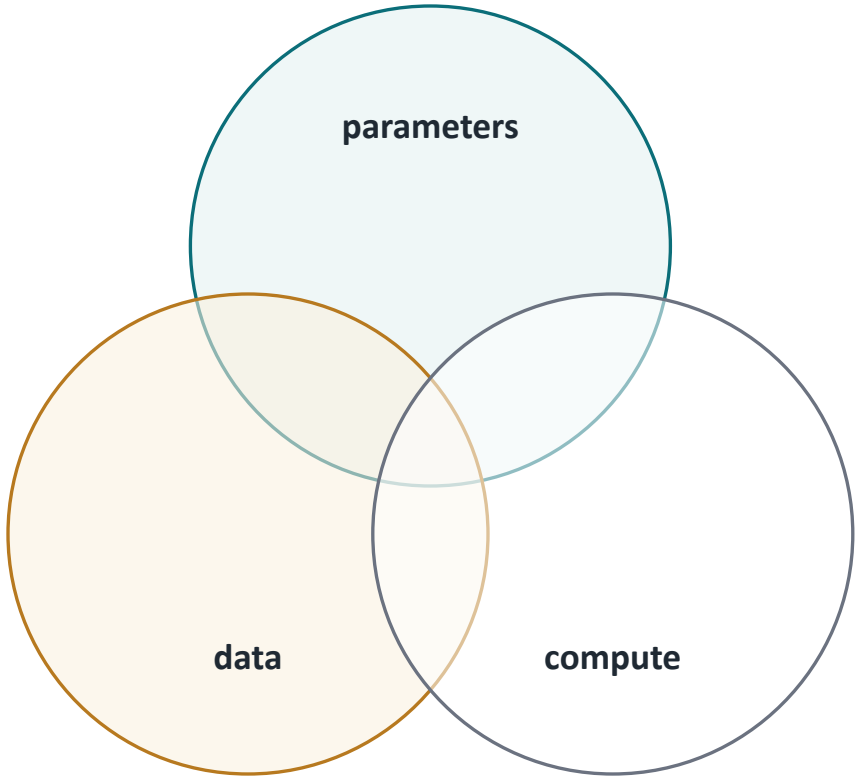
capacity + data + compute

A three-way training budget

Resource	Rough role
Parameters	capacity of the model
Training tokens	coverage and information
Compute	work spent fitting the model to the data

Increasing only one resource can be wasteful.

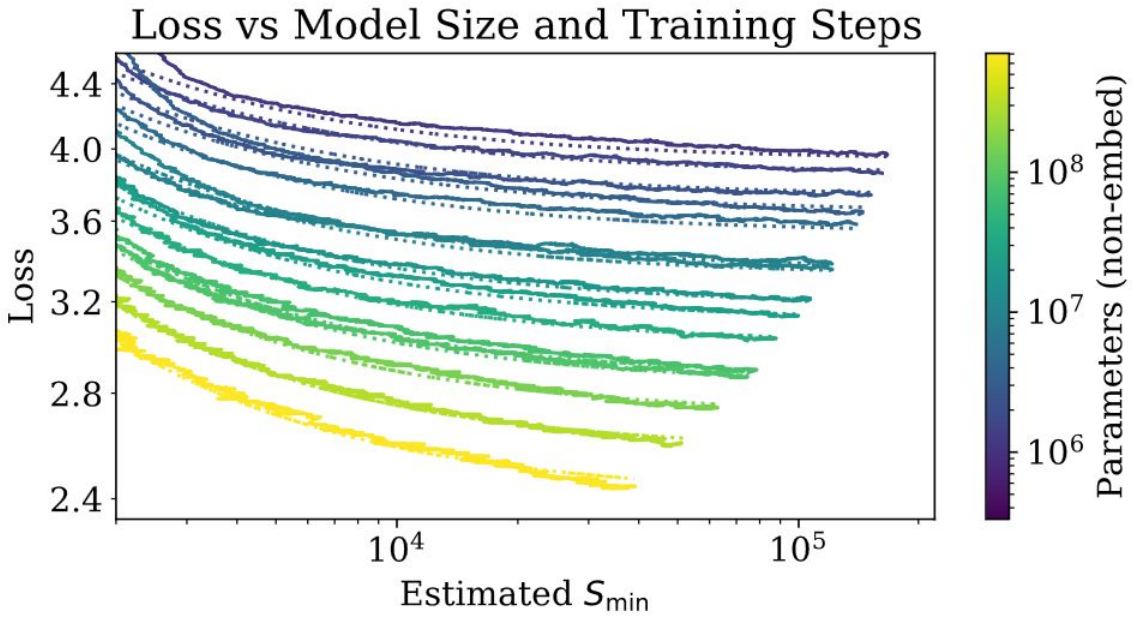
Model size, dataset size, and training compute have to be balanced.



all three have to grow together

Hoffmann et al. (2022) "Training compute-optimal large language models", arXiv:2203.15556

Why scale matters: the loss falls predictably



- **Power laws (Kaplan et al. 2020):** loss vs compute and parameters
- **Compute-optimal training (Hoffmann et al. 2022):** for a fixed budget, grow parameters and tokens together: roughly 20 tokens per parameter.
- **Consequence:** improvements became plannable.

The loss is smooth; individual capabilities can still look abrupt when measured with all-or-nothing metrics.

Kaplan et al. (2020) arXiv:2001.08361 · Hoffmann et al. (2022) arXiv:2203.15556 · Schaeffer et al. (2023) arXiv:2304.15004

Pretraining and mid-training

Pretraining: broad predictive capability

Ingredient	Pretraining
Data	very large, broad corpora
Objective	next-token prediction
Learns	language, code, knowledge, recurring structures
Produces	a base language model

A base model is primarily a continuation engine:

```
input prefix  →  plausible continuation
```

Mid-training: change the curriculum

Often: continued pretraining with a more focused mixture

- more mathematics, code, or formal proofs;
- more curated or synthetic data;
- longer contexts;
- some broad data retained to limit forgetting.

Same objective, different data distribution. Useful when the desired capabilities are underrepresented in the original corpus.

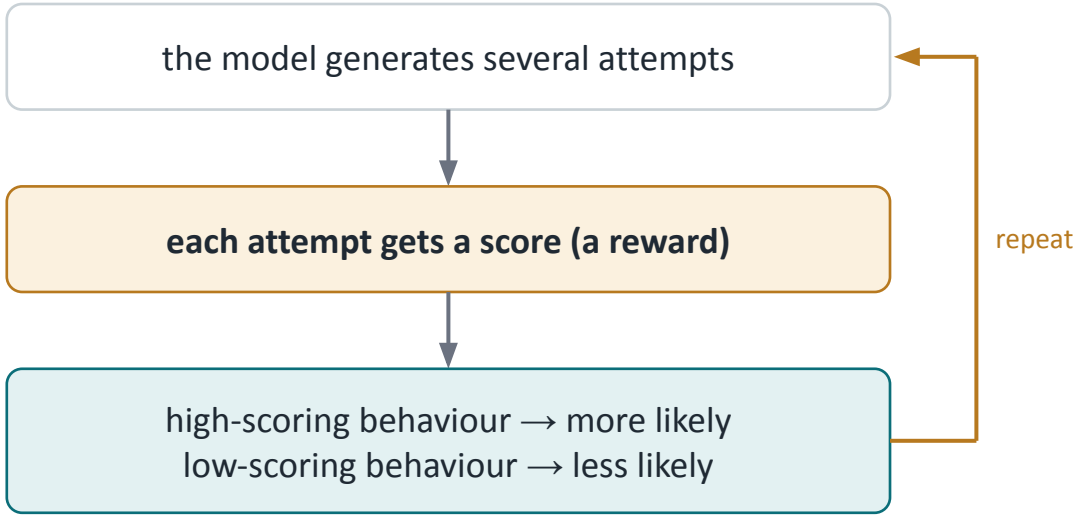
Train a chatbot

Post-training: from base model to assistant

Signal	Desired effect
Instruction/response examples	follow requests
Preference comparisons	prefer useful behavior
Safety data and policies	avoid harmful behavior
Tool trajectories	learn when and how to use tools

Post-training changes behavior and steerability.

Learning from rewards

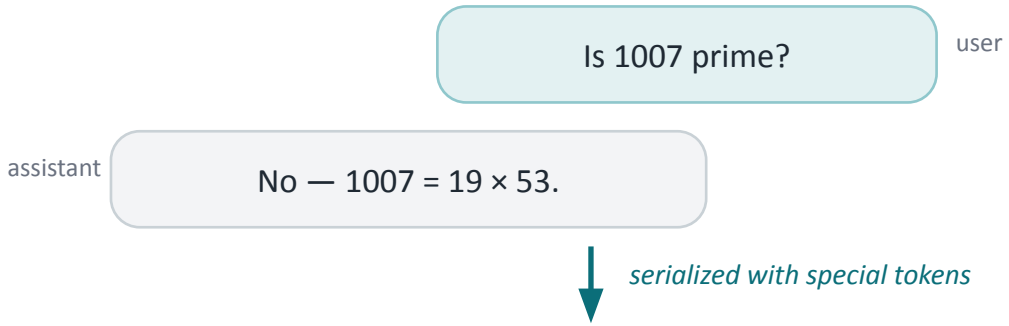


Where can the score come from?

Source	Example question
people comparing answers	which answer is more useful?
a learned preference model	which would people prefer?
an automatic check	**does the number match? do the tests pass? does the checker accept?**

From next token prediction to chatbot

what you see



what the model sees: one flat token stream

```
<|system|> You are a helpful assistant.
<|user|> Is 1007 prime?
<|assistant|> No — 1007 = 19 × 53. <|end|>
```

- **special tokens = new vocabulary entries** (own embedding rows) that never occur in normal text;
- “answering” is **continuing this document** after <|assistant|>.

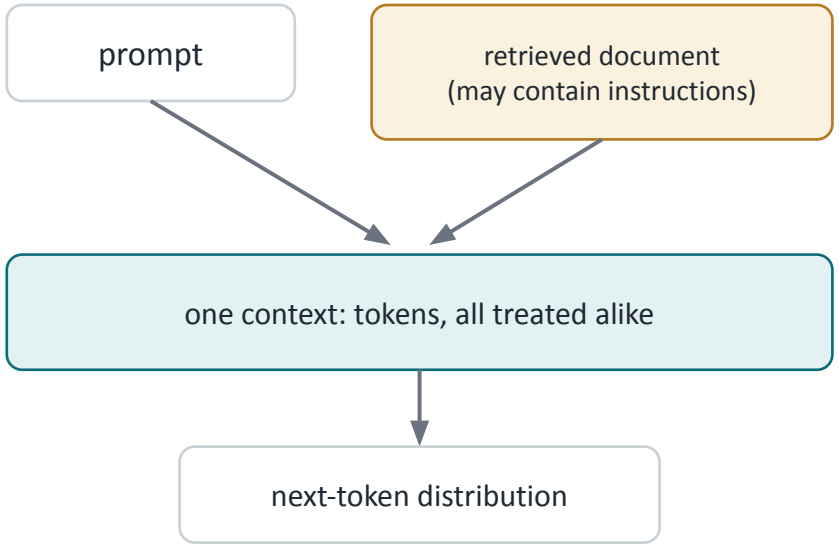
Asking is not controlling

An instruction is text in the model's context, not a command executed by a trusted interpreter.

Even a post-trained assistant may:

- misinterpret the request;
- ignore a constraint;
- follow a misleading instruction inside supplied text;
- produce a plausible answer instead of a correct one.

$$\text{instruction following} \neq \text{guaranteed compliance or truth}$$



there is no privileged channel for “commands”

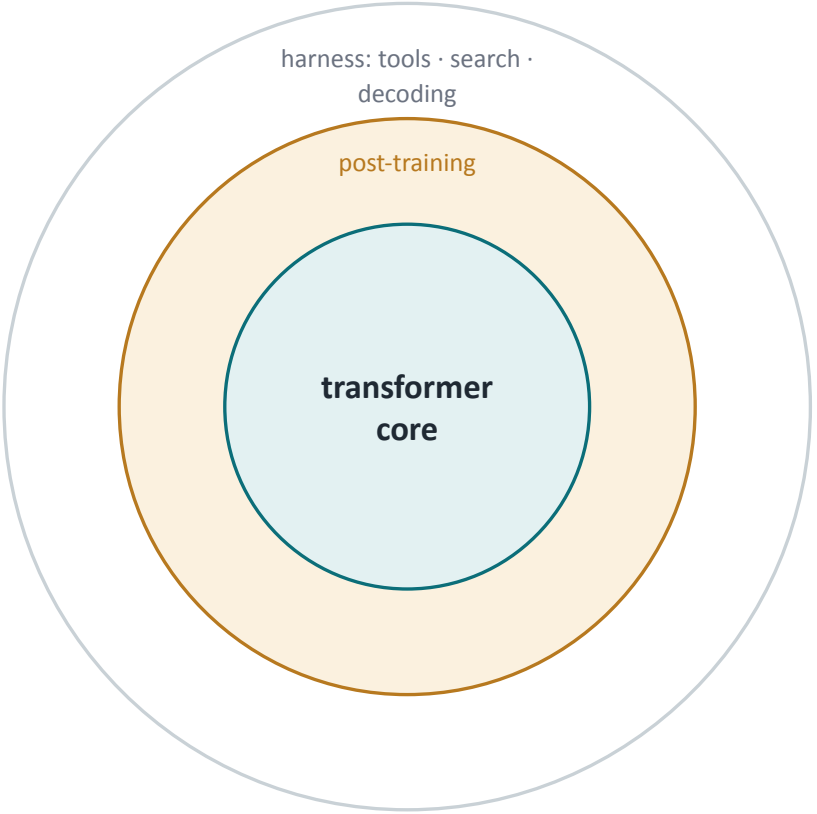
So what is an LLM?

For today, an LLM is:

> *a large neural language model, usually built from transformer blocks, trained on very many token sequences to predict continuations*

A deployed assistant typically adds:

- specific training for useful behavior;
- a decoding strategy;
- a harness (more context, and external tools);



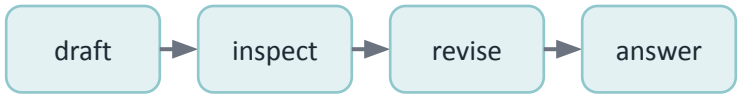
the transformer is the core, not the whole algorithm

Test time compute

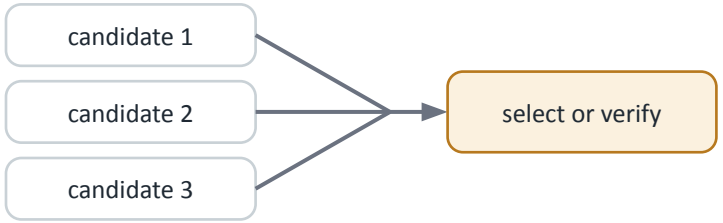
Train-time compute and test-time compute

Compute	When?	What changes?
Pretraining	before deployment	broad model parameters
Mid- / post-training	before deployment	domain focus and behavior
Test-time compute	for one request	tokens, samples, tool calls, search

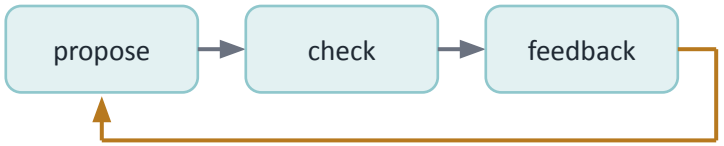
Continue longer



Try several candidates



Interact with tools



Test-time computation changes how the fixed model is used on one problem.

Reasoning models: what the output looks like

Since late 2024, the strongest models “think before answering”:

```
Q: Is 1007 prime?

<think>
  1007 … try small factors. 7? 7 × 143 = 1001, no. 11? 11 × 91.5,
no.
  Wait — 19 × 53: 19 × 50 = 950, 19 × 3 = 57, 950 + 57 = 1007. Yes!
  Let me double-check: 19 × 53 = 1007.✓  So 1007 is not prime.
</think>

No — 1007 = 19 × 53.
```

the “thinking” is ordinary token generation

it costs test-time compute: thousands of extra tokens on a hard question

drafts, checks, backtracks, visible in the trace (“Wait —”)

Two emerging abilities

1 · The ability generalizes

trained on verifiable math and code only: yet reasoning improves on any request, law, biology, planning, proofs.

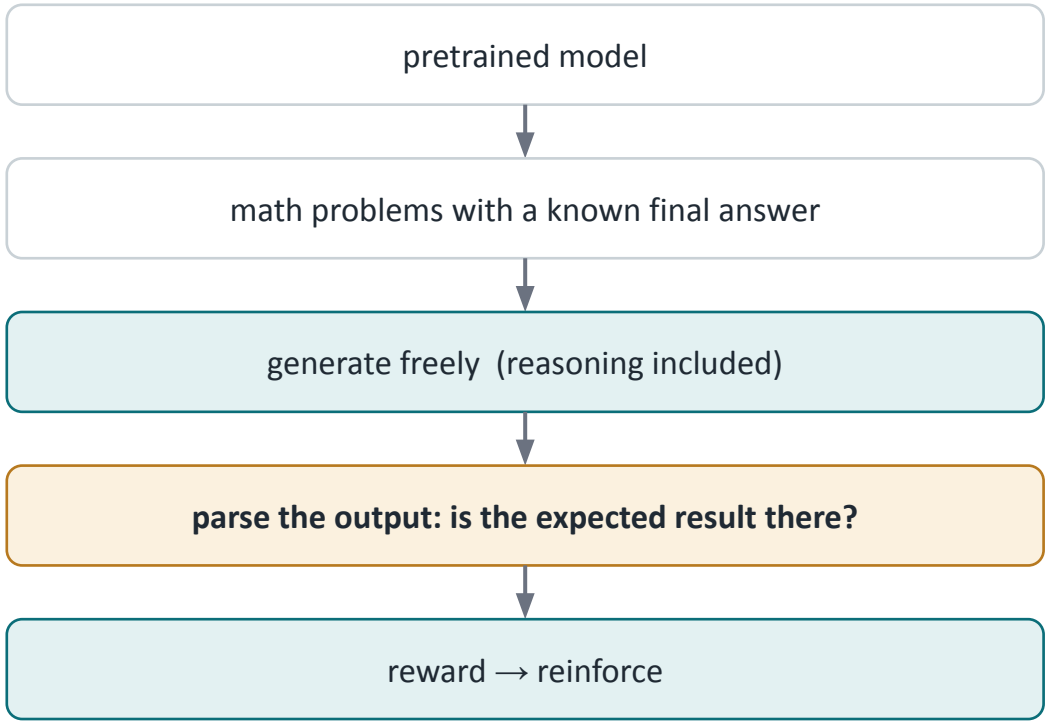
2 · Self-correction emerges — the “aha moment”

```
… so x = 4.
Wait — that ignores the negative root.
Let me redo this step: x² = 16 …
```

models start interrupting themselves, re-deriving, and fixing their own slips.

DeepSeek-R1 (2025), §2 “aha moment”

Where this ability comes from: DeepSeek-R1

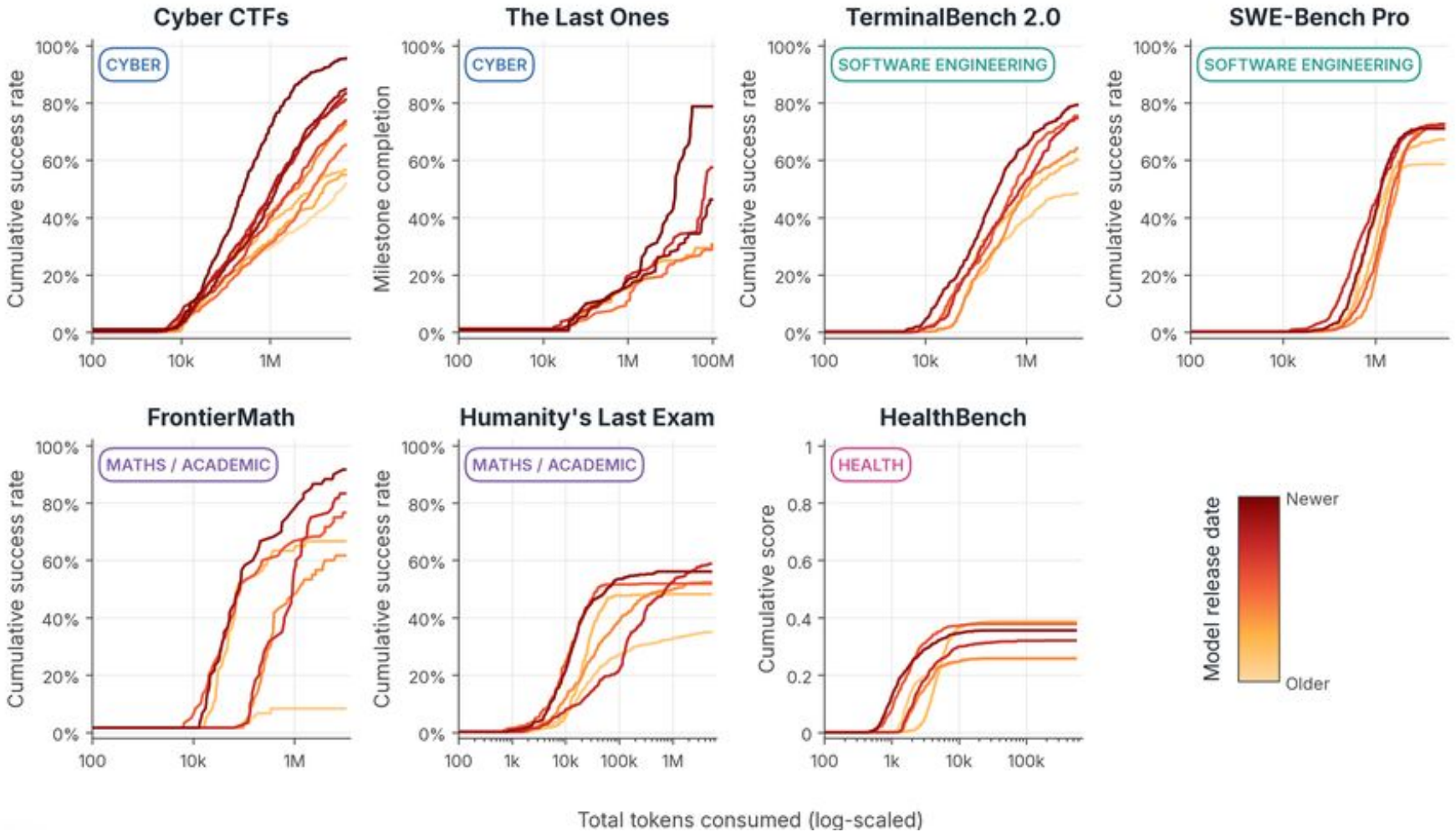


Nobody checks the reasoning.
Only the parsed final answer is verified.

- cheap: a regex against a known answer, millions of times;
- under this loop, the traces grow longer on their own: reasoning longer is what makes the reward come;
- no human-written reasoning demonstrations needed.

DeepSeek-R1 (2025), arXiv:2501.12948 — GRPO introduced earlier in DeepSeekMath (2024), arXiv:2402.03300

A new scaling law: test-time compute



same fixed model: only the reasoning budget changes;

accuracy grows with the log of reasoning tokens, then saturates;

quality: cheap-and-fast or slow-and-better, from the same weights.

Frontier labs now farm verifiable environments

Environment	Automatic check	Matches which product
software engineering	tests pass, build succeeds	coding assistants, agents
mathematics	parse the final answer	reasoning models
agentic tasks	goal reached? API call succeeded?	computer-use agents
games & puzzles	win / lose, constraints satisfied	general reasoning
retrieval QA	exact match with gold answer	search, assistants
formal proofs	**the kernel accepts or rejects**	**verified mathematics & software**

Why proof assistants are an interesting environment

Compared with the other verifiable environments:

unambiguous statements

a formal goal means exactly one thing

fine-grained feedback

every step is checked, not only the final answer

beyond quantitative checks

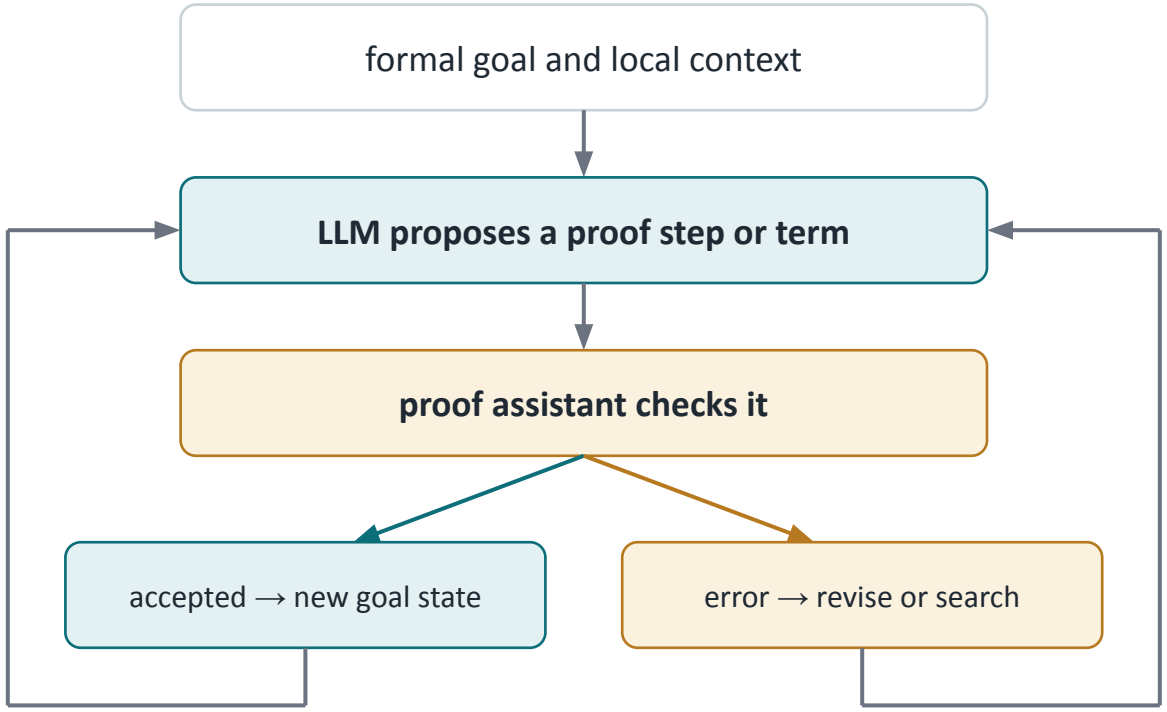
not just “does the number match”: arbitrary statements can be certified

any formal reasoning

mathematics, but also software correctness, protocols, systems

Parsing-based rewards judge the final answer while a proof assistant judges sub steps.

The central interaction

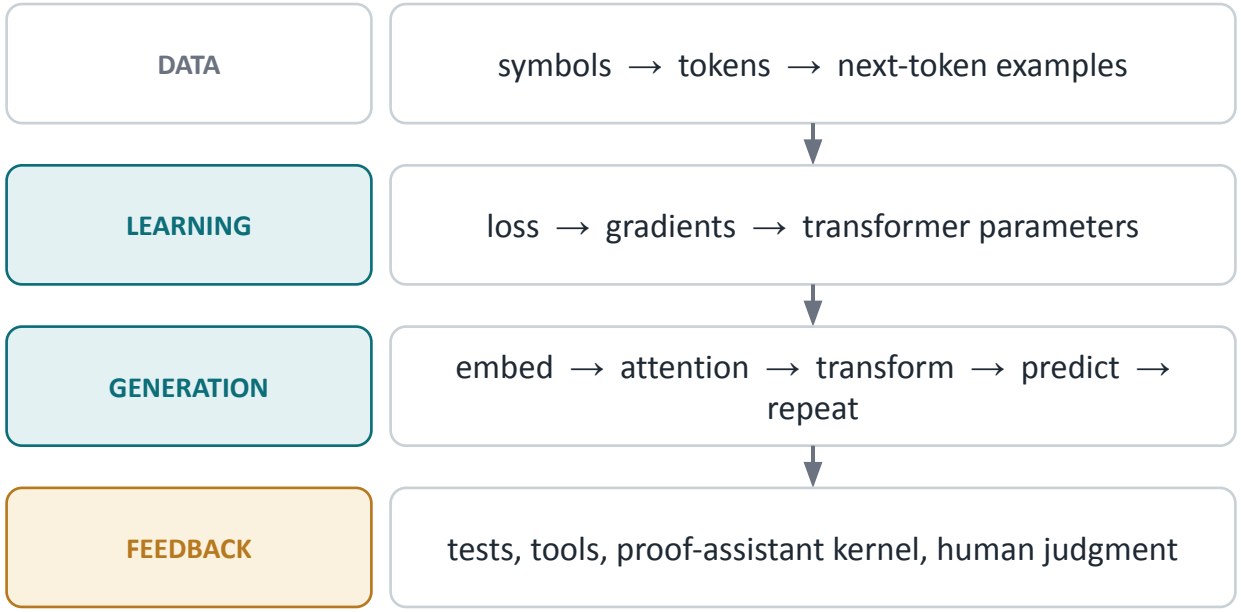


The model proposes.

The proof assistant verifies.

This checker can be used during:

- data generation;
- training (as a verifiable reward);
- search at test time;
- interactive assistance.

⟶ *the subject of lecture 2*

Take-away mental model