1 The task: predicting the next token

A language model does one thing: given some text, it estimates how likely each possible **next token** is. A token is a small chunk of text, usually a word or part of a word ("unbelievable" might be `un`, `believ`, `able`). To generate text, the model picks a token from its predicted probabilities, adds it to the text, and repeats.

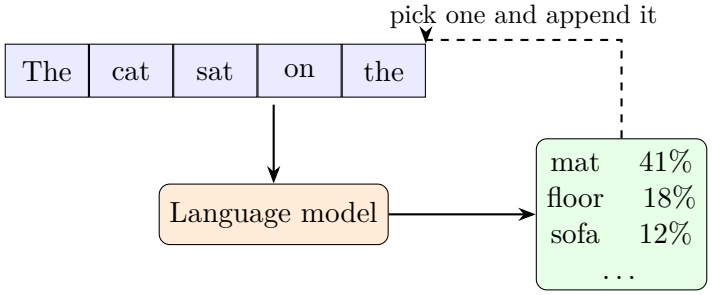


Figure 1: The generation loop: predict, pick, append, repeat.

How it is trained. Take a huge amount of real text. Hide the next token, let the model guess, and measure how much probability it gave to the true token. Adjust the model's internal numbers (its **parameters**) to make the true token more likely next time. Repeat billions of times. The quantity being minimised is called the **cross-entropy loss**: a number that is small when the model is rarely surprised.

Tokens become numbers. Each token is an ID in a fixed vocabulary (typically 30 000–200 000 entries). Each ID looks up a list of a few thousand numbers called its **embedding**. Tokens used in similar ways end up with similar embeddings.

Analogy

Predictive text on your phone, but trained on a large part of the internet and with enough capacity to track grammar, facts, and the thread of a long argument.

Key idea

Everything that follows, including chat assistants, is built on next-token prediction. The differences lie in *how the predictor is built* (Sections 2–4), *how it is trained and run efficiently* (5–8), and *how its behaviour is shaped* (9–11).

Read on your own: going deeper

Formally, the probability of a whole text is the product of each token's probability given the tokens before it, $p(x_1, \ldots, x_T) = \prod_t p(x_t \mid x_{<t})$. The training loss is $-\frac{1}{T}\sum_t \log p(x_t \mid x_{<t})$. *Perplexity* is e raised to this loss; roughly, the number of equally likely options the model is torn between per token. Tokenizers are usually built with *byte-pair encoding*: start from single bytes and repeatedly merge the most frequent adjacent pair into a new token.

2 The old approach: RNNs and LSTMs

Before 2017, the standard way to process text was a **recurrent neural network (RNN)**. It reads the text one token at a time and keeps a running summary called the **hidden**

state: a list of numbers that it rewrites after every token.

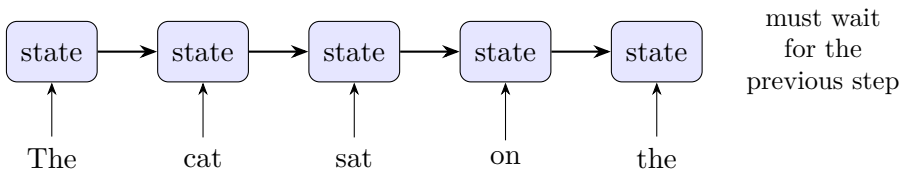


Figure 2: An RNN reads left to right, passing a single summary along. Step 5 cannot start before step 4 finishes.

Analogy

A reader who is allowed only one index card of notes. After every word they must rewrite the card. By page 300, there is little left from page 3.

Two fundamental problems.

1. **Forgetting and weak learning signal over distance.** Information from an early word must survive many rewrites of the card. Worse, during training the "error signal" that tells the network how to improve has to travel back through all those steps, and it tends to shrink to almost nothing (the **vanishing gradient** problem). So RNNs struggle to learn that a word at the start matters for a word much later.
2. **Strictly sequential.** Each step needs the result of the previous one, so the work cannot be spread over many processors at once.

LSTMs (Long Short-Term Memory networks, 1997) partly fix the first problem. They add a separate notebook (the *cell state*) with three *gates* that decide what to erase, what to write, and what to read out. Because the notebook is updated by adding and erasing rather than completely rewriting, information and learning signals can survive longer. But LSTMs are still sequential, and still squeeze everything into a fixed-size memory.

Read on your own: going deeper

During training an RNN is "unrolled" into a very deep network with one layer per time step, and the error is propagated backwards through it (*backpropagation through time*). The sensitivity of a late state to an early one is a product of many matrices. If each factor shrinks things slightly, the product vanishes exponentially ($0.9^{100} \approx 0.00003$); if each expands things slightly, it explodes ($1.1^{100} \approx 14\,000$). Exploding is handled by clipping; vanishing is the deeper problem. An LSTM's notebook update is $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ where $f_t, i_t \in (0, 1)$ are the forget and input gates; when $f_t \approx 1$ the error signal flows back almost unchanged.

3 Attention

The Transformer (Vaswani et al., 2017, "Attention Is All You Need") drops the step-by-step summary. Instead, *every token can look directly at every earlier token* and pull in whatever is useful. This looking-and-pulling is called **attention**.

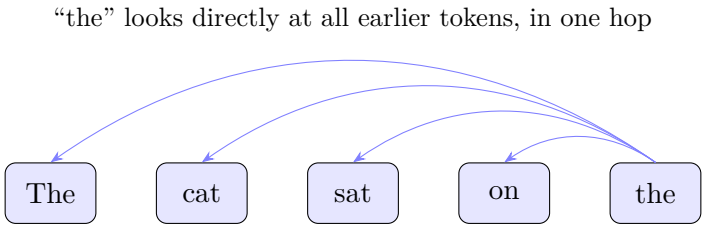


Figure 3: With attention, distance does not matter: token 5 reaches token 1 in a single step.

3.1 The idea: a fuzzy lookup

Each token produces three lists of numbers:

- a **query**: what am I looking for?
- a **key**: what do I contain, so others can find me?
- a **value**: what I will hand over if someone picks me.

To update one token, compare its query with every earlier token's key. A good match gives a high score. Turn the scores into percentages that add to 100% (using the *softmax* function), then take that percentage mix of the values. The result is that token's new representation, now informed by whatever it found relevant.

Analogy

A library with a blurry catalogue. You walk in with a question (query). Each book has a spine label (key) and contents (value). You do not borrow one book: you take a blend of all of them, weighted by how well each label matches your question.

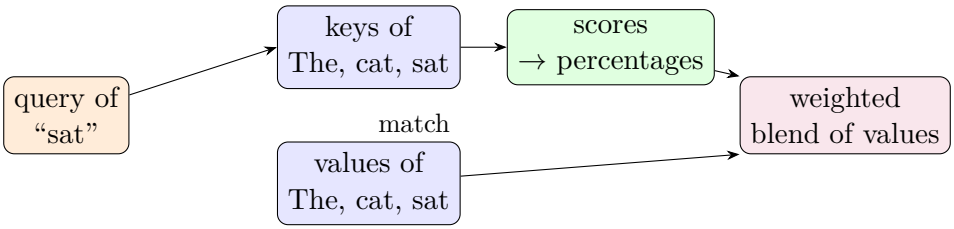


Figure 4: One attention lookup for one token.

3.2 Causal masking

When generating text, a token must not peek at future tokens. During training the whole sentence is available, so we *block* the future with a mask: token t may only attend to tokens $1..t$. The result is a triangular pattern of allowed connections.

Common misconception

“The model sees the whole sentence at once, so it can see the answer.” No. The mask means position t only ever uses tokens before it, so predicting token $t+1$ is a fair test. Without the mask a model would just copy the answer and learn nothing.

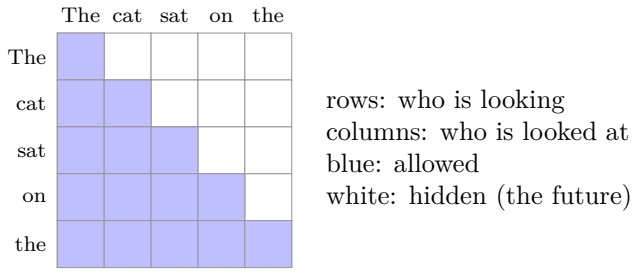


Figure 5: The causal attention pattern for a five-token sentence.

3.3 Many heads, and a sense of order

One attention pattern is not enough: a token may need to find its subject, the previous word, and a distant name at the same time. **Multi-head attention** runs several lookups in parallel (say 32), each with its own query/key/value rules, and combines the results. Different heads learn different jobs.

Attention by itself ignores word order (shuffle the tokens and the pairwise comparisons are unchanged), so the model must be *told* each token's position. Early models added a position-dependent pattern to each embedding; most modern models use **rotary embeddings (RoPE)**, which rotate the query and key numbers by an amount that depends on position, so that scores depend naturally on how far apart two tokens are.

Read on your own: going deeper

In symbols, with matrices Q, K, V holding all queries, keys and values: $\text{Attention}(Q,K,V) = \text{softmax}(QK^\top/\sqrt{d_k} + M)V$, where M is 0 for allowed positions and $-\infty$ for masked ones. The division by $\sqrt{d_k}$ matters: dot products of d_k-dimensional vectors have a typical size growing like $\sqrt{d_k}$, and very large scores make softmax nearly one-hot, which kills the learning signal. Queries, keys and values come from three learned linear maps of the same input. The $T \times T$ score matrix is why cost grows with the *square* of the context length.

4 The Transformer block and the full model

A Transformer is a stack of identical **blocks**. Each block has two stages:

1. **Attention**: tokens *communicate*, gathering information from earlier tokens.
2. **Feed-forward network (MLP)**: each token *thinks* on its own, transforming what it just collected. This is a small two-layer neural network applied to every token separately; most of the model's parameters, and much of its stored factual knowledge, live here.

Two details make deep stacks trainable. **Residual connections**: each stage *adds* its output to what was already there instead of replacing it. **Normalisation**: numbers are rescaled before each stage so they stay in a sane range.

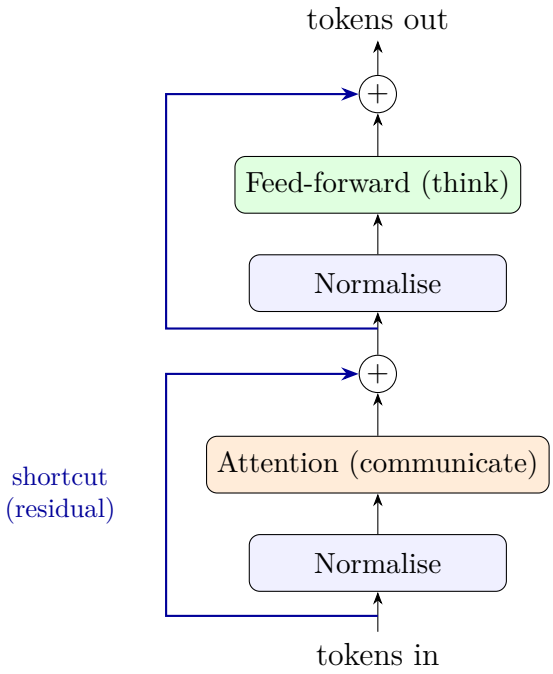


Figure 6: One Transformer block. The blue shortcuts carry the original signal straight through.

Analogy

A document circulating around a conference table. At each stop there is a group meeting (attention) followed by individual desk work (feed-forward). Nobody retypes the document; each person only adds annotations.

Key idea

The shortcut path gives the learning signal a clear highway from the output back to the early layers. This is the same trick that an LSTM's notebook uses across *time*, but applied across *depth*, which is why Transformers can be dozens or hundreds of layers deep.

4.1 The whole model

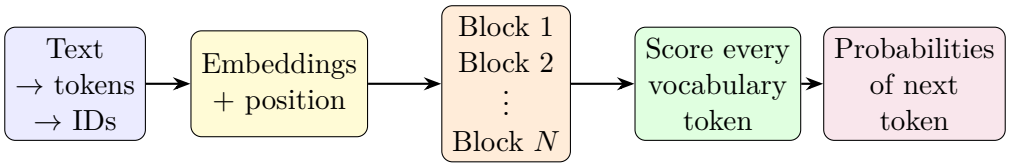


Figure 7: A GPT-style ("decoder-only") model from text to next-token probabilities.

A "7B" or "70B" model has 7 or 70 billion parameters, all in the embeddings and the blocks. A rough guide: each block holds about $12d^2$ parameters, where d is the width of the model's number lists.

Read on your own: going deeper

Per block: attention has four $d \times d$ matrices (query, key, value, output), about $4d^2$ parameters; the feed-forward network usually expands to $4d$ and back, about $8d^2$. Modern models often use RMSNorm instead of LayerNorm, a gated feed-forward variant (SwiGLU), and *grouped-query attention* (sharing keys/values among several query heads to save memory, see Section 6). "Pre-norm" means normalising *before* each stage, which trains more stably than normalising after.

5 Why Transformers train so much faster

Training uses text where the correct continuation is already known. This allows **teacher forcing**: show the model the whole sentence and ask it to predict the next token at *every* position simultaneously (the mask keeps each position honest). One pass through the network yields as many training examples as there are tokens.

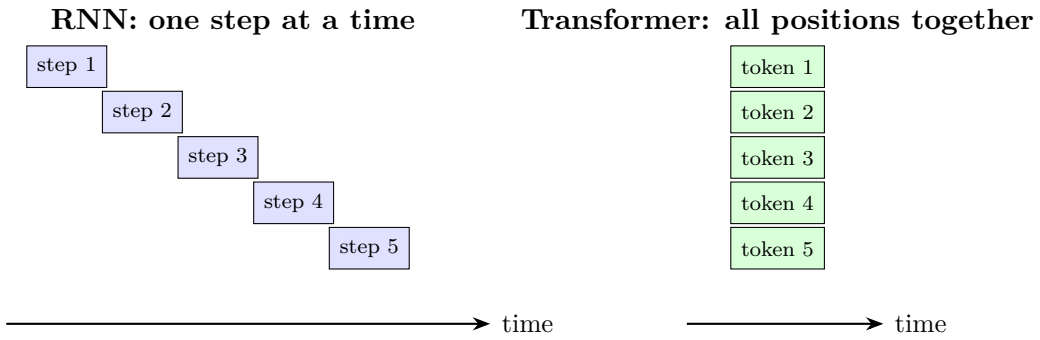


Figure 8: The RNN's five steps happen one after another. The Transformer handles all five positions in the same moment.

Analogy

An RNN is a bucket brigade: the last person waits for everyone before them. A Transformer is a stadium where everyone looks up their own answer at the same time. The total *work* may be larger, but the *waiting time* is much smaller, because thousands of workers are busy at once.

The advantages, summarised.

	RNN / LSTM	Transformer
Processing within a sequence	One token after another	All tokens at once
Distance from early to late word	Many steps	One step (attention)
Learning signal over long range	Fades	Direct path, plus shortcuts
Memory of the past	One fixed-size summary	Every earlier token stays available
Fit with modern chips	Poor (sequential)	Excellent (big matrix operations)
Cost as text gets longer	Grows linearly	Grows with the square

Key idea

The Transformer did not win because attention is magically smarter. It won because it turns language processing into **big, regular blocks of arithmetic** that can be spread over thousands of chips. More hardware could be turned into more training data and bigger models, and loss improves smoothly (and predictably) as you scale up.

Common misconception

Transformers are parallel across the sequence only when the whole sequence is already known (training, and reading a prompt). When *generating*, each new token depends on the one just chosen, so generation is still one token at a time (next section).

Read on your own: going deeper

Scaling across many chips. One chip cannot hold or train a large model, so work is split. *Data parallelism*: each chip holds a full copy of the model and processes different training examples; their updates are averaged. *Sharded data parallelism* (ZeRO/FSDP) does the same but splits the model's parameters and optimiser memory across chips. *Tensor parallelism*: split individual big matrix multiplications across chips (needs very fast links). *Pipeline parallelism*: different chips hold different layers, like an assembly line. With the common Adam optimiser in mixed precision, training needs roughly 16 bytes per parameter, so a 7B model needs over 100 GB before counting activations. Total training compute is roughly $6\times(\text{parameters})\times(\text{tokens})$. **Where the RNN's disadvantage is not absolute:** recent "state-space" models (e.g. Mamba) revive recurrence in a form that can be trained in parallel, because recurrence has a big advantage at inference: constant memory per token.

6 Running a model: prefill, decode, and the KV cache

Using a trained model has two phases.

The KV cache. When a token looks at earlier tokens, it needs their keys and values. Those never change once computed, so we *store* them. Each new token then only com-

1. Prefill: read the prompt (parallel, all at once)



2. Decode: generate one token per step (sequential)

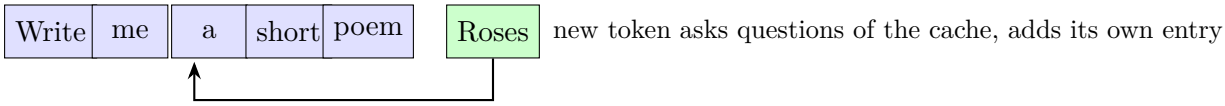


Figure 9: Prefill is fast and parallel; decoding is slow and one-at-a-time.

putes its own query, key and value and looks things up in the stored ones. Without the cache we would redo all earlier work for every new token.

Why generation is slow: it waits on memory, not arithmetic. To produce one token the chip must read *all of the model's parameters* from memory, but does very little arithmetic with each one. The processor spends most of its time waiting for data. Example: a 70-billion-parameter model stored in 16-bit numbers is 140 GB; a chip that can read 3 TB per second needs about $\frac{1}{20}$ s just to read it, which caps a single conversation at roughly 20 tokens per second regardless of how fast the arithmetic is.

Analogy

A librarian must walk to a distant shelf to fetch a huge encyclopedia before answering any question. Answering one reader per trip is wasteful. Answer 64 readers per trip and the cost per reader collapses.

The fix is batching: serve many users at once, reading the model's parameters once for all of them. Serving systems keep a rolling batch ("continuous batching"). Another trick, **speculative decoding**, uses a small cheap model to *guess* several next tokens and lets the big model check all guesses in a single parallel pass, keeping the correct ones.

Read on your own: going deeper

The cache for one token costs about 2 × (layers) × (key/value width) × (bytes per number). For a 7B-class model this is about half a megabyte per token, so a long conversation consumes gigabytes, and the cache can limit how many users fit on one chip. *Multi-query* and *grouped-query attention* share keys and values across heads to shrink the cache 4–8×. Quantisation (storing weights in 8 or 4 bits) shrinks both memory traffic and cost. Sampling controls: *temperature* sharpens (< 1) or flattens (> 1) the probabilities; *top-k* and *top-p* cut off the unlikely tail.

7 GPUs versus CPUs

A **CPU** has a few powerful, flexible cores, good at complicated step-by-step logic with lots of branching, and at finishing *one* task quickly. A **GPU** has thousands of small, simple cores that all do the same operation on different data at the same moment, good at finishing a *huge* amount of simple, repetitive work.

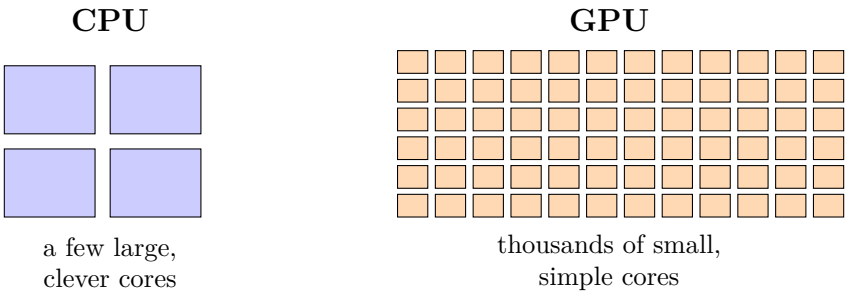


Figure 10: Schematic only: CPUs spend their silicon on control logic; GPUs spend it on arithmetic.

Analogy

A CPU is a few brilliant professors who can solve any problem, one at a time. A GPU is a stadium of ten thousand students who can each do only simple arithmetic, but all at once. Multiplying giant tables of numbers is stadium work.

Why this fits neural networks. Almost all the work in a Transformer is **matrix multiplication**: multiplying big tables of numbers. Each result is computed independently of the others, which is exactly what a GPU's thousands of cores are built for. Modern AI GPUs add *tensor cores* that multiply small blocks of numbers in a single operation using compact number formats (16-bit or even 8-bit), and use very fast "high-bandwidth" memory. A good GPU is hundreds of times faster than a CPU on this kind of work.

Two ways a chip can be limited. It can be *compute-bound* (arithmetic is the bottleneck) or *memory-bound* (waiting for data is the bottleneck). The deciding factor is how much arithmetic you do per byte you fetch.

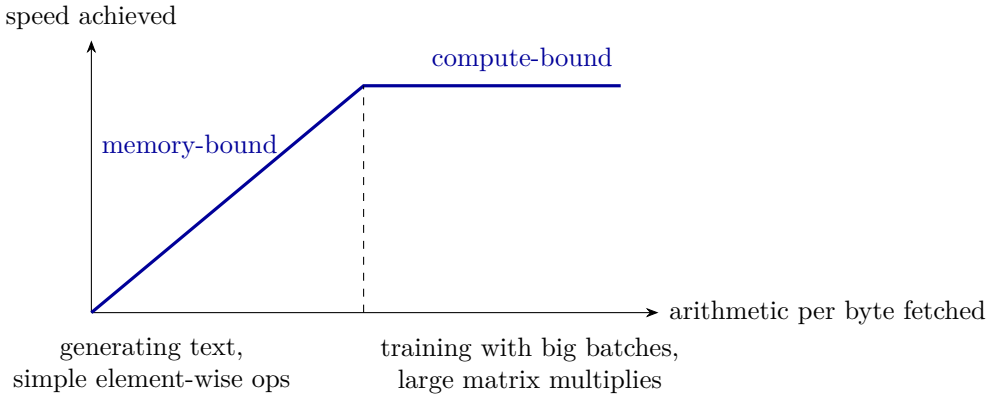


Figure 11: A "roofline": below the dashed line the chip waits for memory; above it, the chip is working flat out.

Consequences: training with big batches keeps the GPU busy (compute-bound); generating one token at a time leaves it waiting (memory-bound, Section 6). Software tricks follow from this too, e.g. **FlashAttention** reorganises the attention computation to avoid shuffling the huge score table in and out of main memory, giving exactly the same answer faster and with far less memory.

CPUs still matter: they prepare data, coordinate the GPUs, and run small models on laptops and phones.

Read on your own: going deeper

The *roofline model* says achievable speed is the smaller of the chip's peak arithmetic rate and (arithmetic per byte) × (memory bandwidth). For a chip with 1000 trillion operations per second and 3 TB/s, the break-even is about 330 operations per byte, far above what single-sequence generation provides (about 1), which is why batching helps so much. Mixed precision trains with 16-bit numbers for speed while keeping a 32-bit master copy of weights for accuracy. *Activation checkpointing* trades roughly 30% extra computation for a large cut in memory. GPU calls are asynchronous, so benchmarks must wait for the GPU to finish before reading a clock, and moving data between CPU and GPU is slow.

8 Pre-training: making a base model

Pre-training is the big, expensive first stage.

1. Collect a very large body of text (web pages, books, code, papers) and clean it, remove duplicates, and filter low-quality material.
2. Turn it into tokens.
3. Train the Transformer to predict the next token, over trillions of tokens, on thousands of GPUs for weeks or months.

The result is a **base model**: very knowledgeable, but it is just a *text continuer*. Ask it "What is the capital of France?" and it may continue with ten more quiz questions, because that is what such text often looks like on the web.

Analogy

A base model has read a large fraction of the internet and can imitate nearly any voice in it, but nobody has told it what job it is doing.

Empirically, results improve in a smooth, predictable way as you scale up **parameters, data, and compute** together ("scaling laws"). This predictability is part of why labs are willing to invest enormous sums in a single training run.

Read on your own: going deeper

The "Chinchilla" analysis (2022) found that, for a fixed compute budget, model size and data should grow roughly together (about 20 tokens per parameter). Models meant for heavy use are often trained on far more data than that, since a smaller model that has seen more is cheaper to serve every day. Typical training ingredients: the AdamW optimiser; a learning rate that warms up and then gradually decays; clipping of unusually large updates; and attention to data quality, which often matters more than raw quantity.

9 Fine-tuning

Fine-tuning means continuing to train an already-trained model on a smaller, targeted dataset so it adopts a particular behaviour, format, or skill. It works because the expensive part (learning language and knowledge) is already done.

Analogy

Pre-training is a university degree. Fine-tuning is two weeks of on-the-job training at a specific company. You do not re-teach arithmetic; you teach the house style.

9.1 Supervised fine-tuning (SFT)

Collect examples of *(prompt, ideal answer)* written by people (or generated and carefully filtered). Train on them with the same next-token objective, but only count the loss on the answer part, so the model learns to *respond* rather than to imitate the user. After a few thousand to a few hundred thousand good examples, the model behaves like an instruction-following assistant. Quality beats quantity: the abilities are already latent in the base model; fine-tuning mostly teaches it *which* of its many possible voices to use.

9.2 Cheaper fine-tuning: LoRA

Updating every parameter of a big model needs a lot of memory. **LoRA** (Low-Rank Adaptation) freezes the original parameters and adds small "adapter" tables beside the big ones; only the adapters are trained. Typically they are less than 1% of the size of the original model. Adapters are tiny to store, can be swapped per task, and can be merged back into the model afterwards.

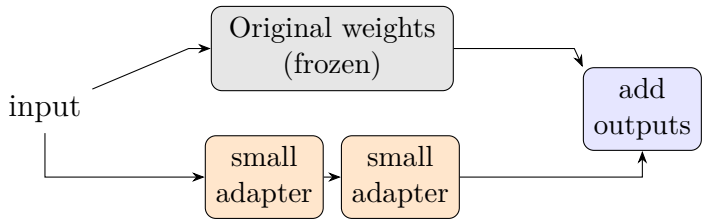


Figure 12: LoRA: keep the original, train a small side path, add the results.

Common misconception

Fine-tuning does not just add knowledge; it also shifts behaviour, and too much narrow training can cause **catastrophic forgetting** of general abilities. Remedies: a gentle learning rate, few passes over the data, mixing in general examples, or LoRA.

Read on your own: going deeper

LoRA replaces a weight matrix W by $W + \frac{\alpha}{r}BA$, where A and B are thin matrices of rank $r \ll d$; B starts at zero so training begins exactly at the original model. For a 4096×4096 matrix at rank 16, the adapter has about 131 000 trainable numbers versus 16.8 million (under 1%). *QLoRA* also stores the frozen weights in 4-bit form to fine-tune large models on a single GPU. Other kinds of fine-tuning: *continued pre-training* on domain text (law, medicine, a new language), and *instruction tuning* over many different tasks so the model generalises to new instructions.

10 RLHF: Reinforcement Learning from Human Feedback

10.1 Why SFT is not enough

SFT teaches the model to *imitate* examples. But: good examples are expensive to write, while *judging* which of two answers is better is easier and cheaper; imitation cannot exceed the examples' quality; and SFT can only say "do more of this", never "do less of that." It can also teach the model to sound confident about things it does not know.

Analogy

SFT is learning to cook by copying a chef's recipes. RLHF is cooking dishes yourself while diners say which of two plates they prefer. You adapt to what people actually enjoy, and may eventually surpass the chef.

10.2 The recipe

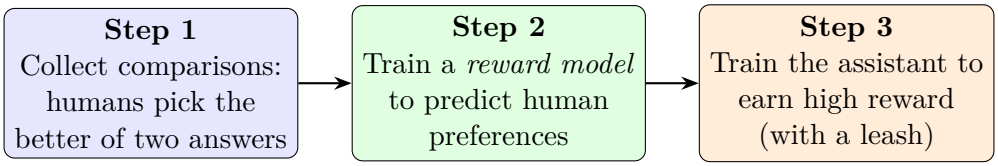


Figure 13: The three steps of RLHF, starting from an SFT model.

Step 1: preference data. For many prompts, sample several answers from the SFT model. Human raters see two answers and choose the better one.

Step 2: the reward model. Train a second model that reads a prompt and an answer and outputs a single score. It is trained so that the answer humans preferred gets a higher score than the other one. This model is a stand-in for human judgement that can be queried millions of times.

Step 3: reinforcement learning. Let the assistant generate answers; the reward model scores them; adjust the assistant to make high-scoring answers more likely. The usual algorithm is **PPO**, a careful method that limits how much the model may change in one update so training stays stable.

10.3 The leash: staying close to the original

The reward model is only an *approximation* of what humans want. If the assistant is allowed to chase the score freely, it finds loopholes: very long, flattering, or repetitive answers that score well but are not actually good (**reward hacking**). To prevent this, a penalty is subtracted whenever the assistant's behaviour drifts far from the original SFT model (measured by a *KL divergence*). The assistant is rewarded for being better *while still sounding like itself.*

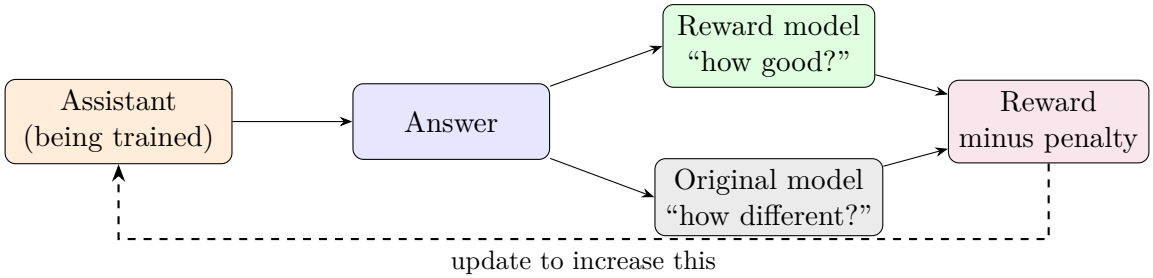


Figure 14: The RLHF loop. Four models are involved at once: the assistant, the reward model, the frozen original, and (in PPO) a “value” model that estimates expected reward.

10.4 Simpler alternatives

- **DPO** (Direct Preference Optimisation): a mathematical rearrangement that trains directly on the preference pairs, with no separate reward model and no reinforcement-learning loop. Simpler and cheaper, but it learns only from the fixed dataset, not from fresh answers the model itself produces.
- **RLAIF / Constitutional AI**: some or all of the preference labels come from an AI judge following written principles, which scales up feedback and makes the guiding principles explicit.
- **Verifiable rewards**: for maths and coding, reward is simply “was the final answer correct / did the tests pass?”, with no reward model. Sampling several answers per prompt and rewarding those better than their group’s average (GRPO) is how many modern “reasoning” models are trained.

Common misconception

RLHF does not install “human values” into a model. It optimises a proxy of the preferences of a particular group of raters. Known side effects: **sycophancy** (telling people what they want to hear), a bias towards long answers, reduced variety in outputs, and **Goodhart’s law**: when a measure becomes a target, it stops being a good measure.

Read on your own: going deeper

Reward model. Given the preferred answer y_w and the other answer y_l for the same prompt x, assume the chance a human prefers y_w is $\sigma(r(x, y_w) - r(x, y_l))$ (the Bradley–Terry model, σ is the logistic function). Training minimises $-\log \sigma(r(x, y_w) - r(x, y_l))$ averaged over the data. Only differences matter, so the score’s absolute scale is arbitrary.

The RL objective. Maximise the expected reward minus β times the KL divergence from the reference (SFT) model: $\mathbb{E}[r(x, y)] - \beta\,\mathrm{KL}(\pi_\theta \,\|\, \pi_{\text{ref}})$. Large β: the model barely moves; $\beta \to 0$: it chases the score and reward-hacks. In practice the penalty is applied per token, and the reward-model score is added at the last token.

PPO compares the new model’s probability of each token with the old one’s, and *clips* that ratio to the range $[1 - \epsilon,\ 1 + \epsilon]$, so no single update changes behaviour too much. It needs an *advantage* estimate (how much better than expected this choice was) from a learned value function.

DPO uses the fact that the optimum of the KL-regularised objective satisfies $r(x, y) = \beta \log \frac{\pi(y|x)}{\pi_{\text{ref}}(y|x)} + (\text{terms depending only on } x)$. Substituting into the Bradley–Terry loss gives a loss that depends only on the model’s own log-probabilities of the two answers, relative to the reference model.

11 From base model to aligned assistant; recap

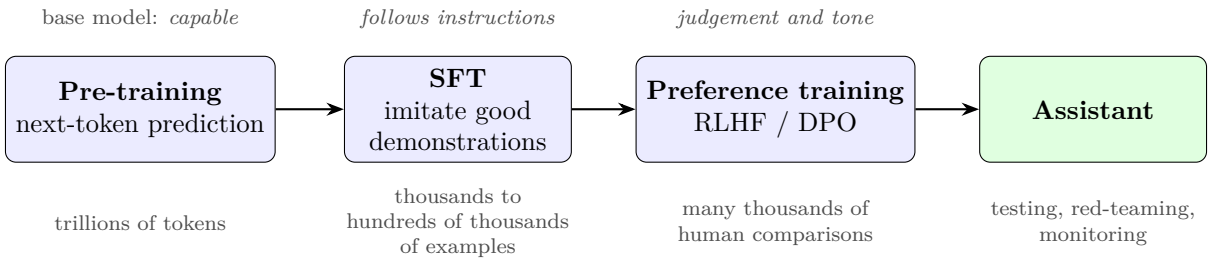


Figure 15: The standard pipeline. Pre-training uses over 99% of the compute; the later stages are cheap but shape behaviour enormously.

1. **Pre-training** makes the model *capable*: language, facts, reasoning patterns, code.
2. **SFT** gives it a *role and format*. Conversations are written with special marker tokens for "system", "user", and "assistant"; the model learns to answer as the assistant and to emit an end-of-turn token to stop.
3. **Preference training** (RLHF, DPO, or AI feedback) gives it *judgement*: be helpful and honest, decline clearly harmful requests, admit uncertainty, choose a sensible length and tone.
4. **Evaluation and red-teaming**: benchmarks, human testing, and deliberate attempts to make the model misbehave. What is found goes back into the data for the next round.

Analogy

The base model is a talented actor who can play any character from the whole internet. SFT hands the actor a script and says, "Today you play a helpful assistant." RLHF is a director watching takes and saying "more like this, less like that," until the performance is consistently good.

What "alignment" means. Making the system do what its users and developers actually intend, safely. A common shorthand is *helpful, honest, harmless*. These goals pull against each other at the edges (a model that does everything it is asked is not harmless; one that refuses everything is not helpful), so alignment involves value judgements and careful specification of desired behaviour, not just algorithms.

Beyond the weights, deployed systems add: **system prompts** that steer behaviour at run time; **retrieval and tools** (search, calculators, code execution) that reduce hallucination; and input/output **safety classifiers**.

One-slide recap

Topic	One-sentence summary
Language model	Predict the next token; generate by repeated sampling.
RNN / LSTM	A fixed-size summary passed along step by step: forgets, and cannot be parallelised.
Attention	Each token takes a relevance-weighted blend of earlier tokens' information.
Transformer	Stack of blocks: attention (communicate) plus feed-forward (think), with shortcuts.
Training speed	Whole sequence in parallel, short paths for learning signals, fits GPUs.
Inference	Prompt is read in parallel; reply is generated token by token, limited by memory; KV cache and batching help.
GPU vs CPU	GPUs: thousands of simple cores, ideal for matrix multiplication. CPUs: a few flexible cores.
Pre-training	Next-token prediction at huge scale gives a capable but unruly base model.
Fine-tuning	Continue training on targeted examples (SFT); LoRA does it cheaply.
RLHF	Learn a reward from human comparisons; improve the model against it, with a leash to the original.
Assistant	Pre-train, then SFT, then preference training, then evaluate.