LLM Basics (3): Toward Chat Models

(for high school and first-year university students)

Instruction tuning, learning from human preferences, and in-context learning

Planned and edited by Yoshio Okamoto / Written by Claude Opus 5.5 (Anthropic)

Ver. 0.1 October 7, 2026

Note: The following content was generated by AI. The editor has checked it, but please be aware that it may still contain errors or mistaken assumptions.

© 2026 Yoshio Okamoto. Free to use for educational purposes (please cite the source).

Introduction

What the pre-training of Parts 1 and 2 produces is a model that "writes a plausible continuation of a given text." The chat LLMs we usually use are models that have been trained further so that they **answer questions and follow instructions**.

> Put another way, a model that has finished pre-training is like "a very knowledgeable person who has read every book in the library but has never practiced talking with people." Part 3 deals with the training that turns this knowledgeable person into **someone useful to talk with**, and with "in-context learning," in which **behavior changes without changing the weights**.[a]

[a] The other documents in this series referred to in this booklet are currently available in Japanese only.

Part	Title	Content
Part 1	Tokens and Language Models	How text becomes numbers, and how it comes out as probabilities
Part 2	Pre-training and Scaling Laws	What happens when a model learns from a huge amount of text
Part 3	**Toward Chat Models**	How a model that "continues text" becomes one that answers questions

How to read this booklet The main text is written so that it can be read with high-school mathematics (exponential functions and logarithms, the summation sign $\sum$, probability and expected value, and the dot product of vectors). The **mathematical foundations**, such as computing gradients by differentiation, deriving the optimal distribution with the method of Lagrange multipliers, and deriving the DPO loss, are collected in the **footnotes** and in **Appendix A** at the end. You can follow the main text even if you skip the appendix. All numerical values in the examples were actually computed by programs (Appendix B).

Notation

- x is the input (a question or instruction) and y is the response. Both are sequences of tokens.
- $\pi(y \mid x)$ is the probability that the model outputs the response y for the input x. As in Section 1.1 of Part 1, the probability of a whole response is the product of the probabilities of its tokens. Here π is not the circle ratio: following the custom of reinforcement learning,

the probability distribution of the model's output is written π (the policy; π is the Greek letter corresponding to the "p" of *policy*).

- π_{ref} is the probability distribution of the reference model (the model at the end of instruction tuning).
- r is a number expressing how good a response is (the **reward**), and $\sigma(z) = \dfrac{1}{1+e^{-z}}$ is the **sigmoid function**.
- In this booklet, β is a positive constant introduced in Section 3.3; it is different from the power-law exponent β of Part 2.

1 From a model that "continues text" to a chat model

1.1 How a pre-trained model behaves

If you input "What is the capital of Japan?" to a model that has finished pre-training (a **pre-trained model** or **base model**), it does not necessarily answer "Tokyo." The Web contains many texts like workbooks, with one question after another, so **starting to write the next question**, such as "What is the highest mountain in Japan?", is also a plausible continuation.

> A pre-trained model **has knowledge, but has not learned to use it in the form of "answering a question."** It has the knowledge, yet instead of answering what was asked, it starts writing the rest of the workbook. Fixing this is the theme of the first half of this booklet.

1.2 Three stages

Today's chat models are made roughly in the following three stages (Figure 1).

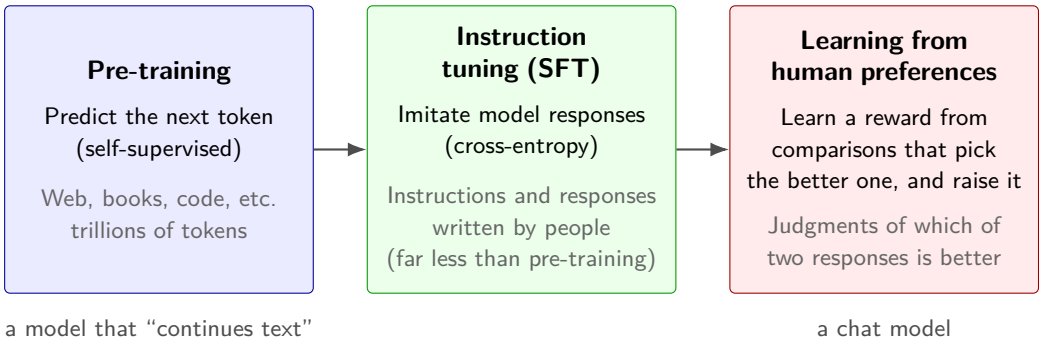


Figure 1 The three stages in making a chat model.

1. **Pre-training**: the content of Part 2. The model learns to predict the next token from an enormous amount of text.
2. **Instruction tuning** (Supervised Fine-Tuning, SFT): the model learns to imitate pairs of "an instruction and a model response" written by people (Section 2).
3. **Learning from human preferences**: based on people's judgments of which of two responses is better, the model learns to give responses that are preferred more (Section 3). **Reinforcement Learning from Human Feedback** (RLHF) is a typical method.

> Compared to a new part-time worker: (1) they studied a wide range of things at school (pre-training), (2) at work they watch and imitate how a senior colleague serves customers (instruction tuning), and (3) they further polish their service based on customers' comments

such as “this way of handling it was better” (learning from human preferences).

An example that showed the effect of this method is OpenAI’s InstructGPT (Ouyang et al., 2022). The responses of a 1.3-billion-parameter model trained with stages 2 and 3 were preferred by raters over those of the 175-billion-parameter GPT-3 with pre-training only. **Even at less than 1/100 of the size, the way of training reversed which was easier to use.**

2 Instruction tuning (SFT)

2.1 Imitating model responses

In instruction tuning, pairs of “an instruction and a desirable response to it” made by people are collected. Each pair is turned into one text, with special symbols that mark the roles inserted, and the model is trained on it. For example, it takes a form like the following (the way the symbols are written differs from model to model).

> <|user|>
> At what temperature does water boil?
> <|assistant|>
> At 1 atmosphere, about 100 °C. In highlands where the air pressure is lower, it boils at a lower temperature.
> <|end|>

The loss is the same cross-entropy as in pre-training, but it is computed **only for the tokens of the response**:

$$L_{\mathrm{SFT}} = -\sum_{t \in \text{response}} \log \pi(x_t \mid x_1, \ldots, x_{t-1}).$$

The instruction part is only read as the “condition”; there is no need to predict it.

This is similar to practicing calligraphy by tracing a model. However, **only the answer (the response) is graded**; whether the question (the instruction) was copied well does not matter. The question only needs to be read.

2.2 What changes with so little data?

The data for instruction tuning are far smaller than for pre-training. InstructGPT used about 13,000 instructions and model responses. There is even a report that a 65-billion-parameter model, instruction-tuned with only 1,000 carefully chosen examples, became able to give responses of fairly high quality (LIMA, Zhou et al., 2023).

The authors of this paper thought that **most of the knowledge is acquired in pre-training, and instruction tuning mainly teaches “in what format and with what attitude to respond.”** What a knowledgeable newcomer needs is not cramming more knowledge, but training in the pattern of “this is how we answer customers.”

3 Learning from human preferences

3.1 Comparing is easier than writing

Instruction tuning has limits. Writing model responses takes effort, and their quality varies with the writer. Also, just imitating model responses does not convey **degrees of good and bad**, such as “which mistakes are serious” or “which explanation is clearer.”

Even if cooking a delicious dish yourself is hard, anyone can taste two dishes and say "this one is better." In the same way, **looking at two responses and choosing the better one is much easier than writing a model response**.

So the model is made to produce several responses to the same question, and data are collected in which people compare them two at a time and choose the better one.

3.2 The reward model: turning pairwise comparisons into a number for "goodness"

From the results of the comparisons, we want to express how good each response is with a single number r (the **reward**). The model in which the probability that response y_i is preferred over y_j is

$$P(y_i \text{ is preferred over } y_j) = \sigma(r_i - r_j) = \frac{1}{1 + e^{-(r_i - r_j)}}$$

is called the **Bradley–Terry model** (1952). If the difference of rewards is 0, the probability is $\frac{1}{2}$; the larger the difference, the closer the probability that the better one is chosen gets to 1.

This is the same idea as estimating the strength of each team from the results of games in a sport. **Ratings** in chess or shogi (Japanese chess), points expressing strength, belong to the same family of ideas: "the difference of points determines the probability of winning."

With the response actually chosen written y_w (win) and the one not chosen written y_l (lose), the loss is

$$L = -\sum_{\text{comparisons}} \log \sigma(r_w - r_l).$$

This is exactly the cross-entropy of logistic regression that we saw in "Supervised and Unsupervised Learning," in the case where the correct answer is always 1 ("y_w won").

How do the rewards move? When this loss is made smaller by gradient descent, each comparison **raises the reward of the winner and lowers that of the loser**. The amount of the change is proportional to[1]

$$1 - \sigma(r_w - r_l) \quad = \quad \text{(how much the model did not expect this result)}.$$

If the model already expects r_w to be much larger than r_l ($\sigma \approx 1$), it hardly changes anything.

In ratings, too, when a strong team beats a weak team the points hardly move, but **when an upset happens they move a lot**. The more unexpected the result, the larger the correction.

Also, adding the same number to everyone's reward does not change the differences, so **only differences of rewards have meaning**. The program in Appendix B adjusts the rewards so that their mean is 0.

[1] It is derived using the formula for the derivative of the sigmoid function, $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$ (Appendix A.1).

Example Suppose that for four responses A–D to the question "Please explain what photosynthesis is," raters compared them two at a time with the following results (Appendix B).

Response	Content
A	accurate and concise
B	accurate but wordy
C	partly wrong
D	does not answer the question

Pair	Wins–losses
A vs B	8–2
A vs C	9–1
A vs D	10–0
B vs C	6–4
B vs D	8–2
C vs D	7–3

Starting all rewards at 0 and making the loss smaller by gradient descent, we obtained

$$r_A = +1.754, \qquad r_B = +0.083, \qquad r_C = -0.433, \qquad r_D = -1.404$$

(Figure 2(a)). The initial loss 41.589 is $60 \log 2$, the value when all 60 comparisons are predicted with probability $\frac{1}{2}$.

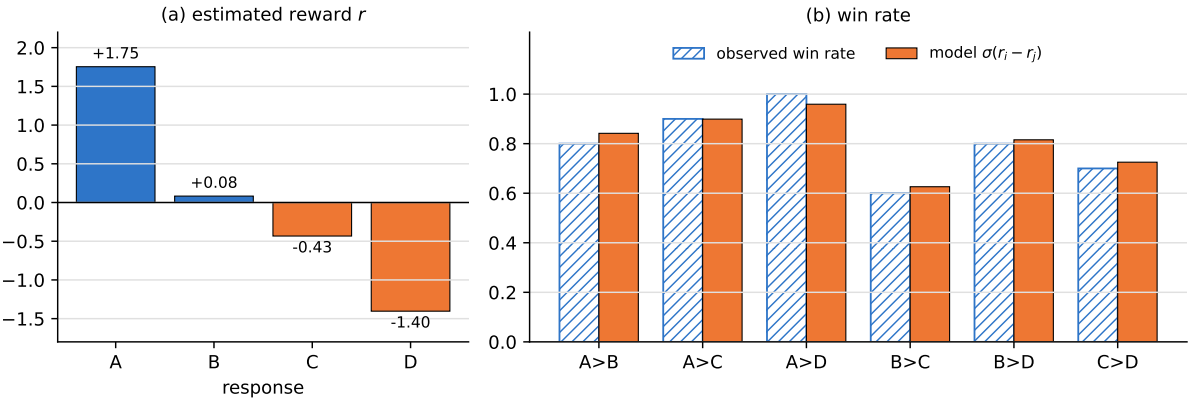


Figure 2 (a) Rewards estimated from pairwise comparisons. (b) Observed win rates, and win rates $\sigma(r_i - r_j)$ computed from the estimated rewards.

A won all 10 games against D, but the win rate computed from the estimated rewards is

$$\sigma(r_A - r_D) = \sigma(1.754 + 1.404) = \sigma(3.158) = \frac{1}{1 + e^{-3.158}} \approx 0.96,$$

not 1 (Figure 2(b)). This is because, under the constraint of placing the rewards of the four responses on a single number line, **the values that best explain all the comparison results** are chosen.

In real LLMs, instead of assigning a reward to each individual response, a neural network that reads the input x and the response y and outputs a reward $r(x, y)$ (the **reward model**) is trained with this loss. In many cases, the reward model itself is a slightly modified LLM. Then **rewards can also be given to new responses not in the training data**. It is like a judge who has tasted thousands of dishes and can score even a dish they see for the first time.

3.3 Raising the reward without moving too far from the original model

Next, the chat model π is trained to give responses to which the reward model gives high rewards. However, **it does not simply maximize the expected reward**.

Reward hacking The reward model does not reproduce people's judgments perfectly. If the model chases only the reward model's score, it starts to give responses that earn points by exploiting the quirks and flaws of the reward model. For example, if the reward model tends to rate highly responses that are merely long and polite with no change in content, the model would give such responses. This is called **reward hacking**.

> If a teacher gave higher marks to essays with more words, students would start writing long essays with no content. **Over-optimizing a "substitute goal" such as a score drifts away from the real goal.**

A penalty for moving too far So we maximize the following quantity.

$$\underbrace{\sum_y \pi(y)\, r(y)}_{\text{expected reward}} \; - \; \beta \underbrace{D_{\mathrm{KL}}(\pi \,\|\, \pi_{\mathrm{ref}})}_{\text{distance from the original model}}$$

The first term is the expected value of the reward (the rewards weighted by their probabilities and added up).[2] The second term is a penalty for moving too far from the model π_{ref} at the end of instruction tuning.

Here D_{KL} is the **KL divergence** (Kullback–Leibler divergence), a quantity that measures the difference between two probability distributions:

$$D_{\mathrm{KL}}(\pi \,\|\, \pi_{\mathrm{ref}}) = \sum_y \pi(y) \log \frac{\pi(y)}{\pi_{\mathrm{ref}}(y)}.$$

It is **always 0 or more, and 0 only when the two distributions are equal**.[3]

> Compared to walking a dog, the reward is "the direction of a tasty smell," the original model π_{ref} is "the owner," and the KL penalty is **the leash**. Even if the dog runs off after the smell, the leash pulls it back so that it does not go too far from where it started. β is the stiffness of the leash: the larger it is, the harder it is to move away from the owner.

The optimal distribution The distribution that maximizes this quantity can be found with the method of Lagrange multipliers (Appendix A.3):

$$\pi^*(y) = \frac{1}{Z}\, \pi_{\mathrm{ref}}(y)\, e^{r(y)/\beta}$$

(Z is a constant that makes the probabilities add up to 1). It is **the original probability, reweighted by the exponential function of the reward**. The higher the reward of a response, the more its probability increases; the lower, the more it decreases.

This has the same form as the **Boltzmann distribution** of statistical mechanics, with β playing the role of **temperature**. In earth science, the exponential decrease of air density with height (the height distribution of air pressure) is a familiar example of the Boltzmann distribution. When the temperature is high, air spreads to great heights; when it is low, it gathers near the ground. In the same way,

[2] University textbooks write the expected value as $\mathbb{E}_{y\sim\pi}[\, r(x,y)\,]$, meaning "the expected value of r when y is chosen according to the distribution π."

[3] See Appendix A.2 for the proof.

- for $\beta \to \infty$ (high temperature), $\pi^* = \pi_{\text{ref}}$ stays unchanged (the leash is too stiff to move), and
- for $\beta \to 0$ (low temperature), the probability is concentrated on the response with the highest reward (no leash).

For "temperature," see also the separate note "The stages where probability enters LLM computation, and 'temperature'."

Example For the four responses of Section 3.2, assuming that the model after instruction tuning outputs A, B, C and D with probabilities 0.30, 0.40, 0.20 and 0.10 (it tends to give the wordy B), we computed π^* (Appendix B, Figure 3). The row $\beta = \infty$ is the original model, and the expected reward is $\sum_y \pi(y)r(y)$.

For example, when $\beta = 1$, the weight of A is computed as $0.30 \times e^{1.754} \approx 0.30 \times 5.78 = 1.73$, and dividing by the sum of the four weights gives the probability.

β	$\pi(A)$	$\pi(B)$	$\pi(C)$	$\pi(D)$	Expected reward	D_{KL}
∞	0.300	0.400	0.200	0.100	+0.332	0.000
2.0	0.535	0.309	0.119	0.037	+0.860	0.131
1.0	0.746	0.187	0.056	0.011	+1.286	0.443
0.5	0.947	0.045	0.008	0.001	+1.660	0.962
0.2	1.000	0.000	0.000	0.000	+1.753	1.201

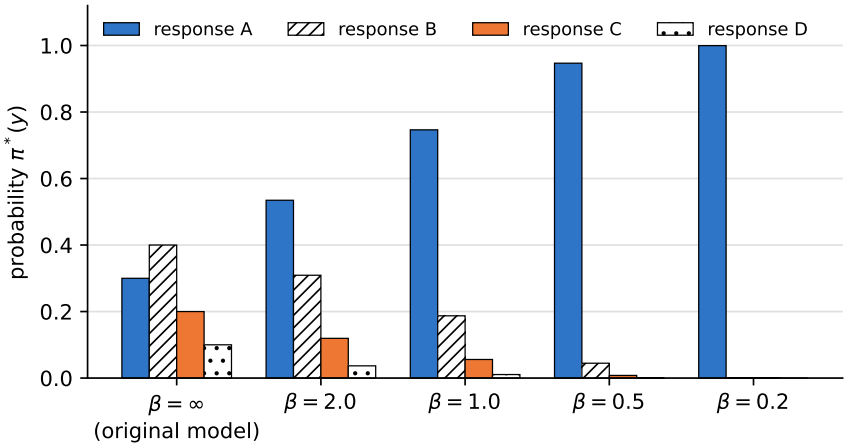


Figure 3 The probabilities of the original model (left) and the optimal distribution π^* for various β. The smaller β is, the more the probability is concentrated on response A, which has the highest reward.

> The smaller β is, the higher the expected reward, but the larger the distance from the original model (the KL divergence) also becomes. β **is a knob that decides how much to trust the reward model.**

3.4 How training is actually done

Real responses are sequences of tokens, and the number of possible responses is astronomical, so it is impossible to write out all responses and compute π^*. Instead, methods such as the following are used.

- **Reinforcement learning** (methods such as PPO): the model actually produces responses, the reward model scores them, and the probabilities of high-scoring responses are raised; repeating this approximately maximizes the quantity of Section 3.3. InstructGPT used this method.
- **Direct Preference Optimization** (DPO; Rafailov et al., 2023): using the formula for the optimal distribution in Section 3.3, π can be learned directly from comparison data **without making a reward model**. The loss has exactly the same form as in Section 3.2: it raises the probability of the chosen response (relative to the original model) and lowers that of the response not chosen.[4]

There is also a method in which part of the comparison judgments is done by AI instead of people. In Anthropic's Constitutional AI (Bai et al., 2022), principles such as "do not give harmful help" are written down in text (this is called the "constitution"), and the model is trained with data in which an AI compares and revises responses in light of those principles.

4 In-context learning

4.1 Behavior changes without changing the weights

Chat models have the property that, just by showing a few examples in the input, they behave following those examples. If you input

apple → pomme
cat → chat
book →

many models continue with "livre" (English to French). If you change the examples to "apple → apples, cat → cats, book →," they continue with "books." **At this time, not a single weight of the model has changed.** Only the input (the context) has changed.

The ability to read a task from examples shown in the context in this way is called **in-context learning**. It became widely known when the GPT-3 paper (Brown et al., 2020) showed that this ability increases as models are made larger. Giving only an instruction without examples is called **zero-shot**, and showing a few examples is called **few-shot**.

> When a teacher writes two worked examples on the blackboard and says "now do the next one the same way," the students imitate the method of the examples and solve the next one. Their brains have not been rebuilt; **they simply read the method from the examples in front of them, on the spot**. In-context learning is similar to this.

4.2 An example of a mechanism using Attention: induction heads

Why can a model "learn" from the context without changing its weights? As one such mechanism, Attention heads that work as follows have been found inside trained Transformers (**induction heads**; Olsson et al., 2022).

> If the current token A has appeared earlier in the context, **look at the token B that came after that earlier A, and predict B as the next token**.

[4] See Appendix A.4 for the loss and its derivation. The ratio of the model's own probabilities plays the role of the "reward."

This is a very simple guess: "Earlier, this word was followed by that. So this time it will probably be followed by the same." It works much like remembering a friend's favorite phrase and knowing how it continues as soon as they start saying it.

Let us reproduce this with the Attention computation of "Deep Learning (4)" (dot product of query and key → softmax → weighted sum of values), choosing the weights by hand (Appendix B). Let the context be

blue 7 red 3 green 5 red

and predict what comes after the last "red."

- The **key** of each position is "the token at the previous position,"
- the **value** is "the token at that position," and
- the **query** of the last position is "the current token (red)"

(all are vectors whose component for that token is 1 and all others 0). The dot product of the query and a key is large at the position whose key is "red," that is, the position of "3" right after the earlier "red" (Figure 4).

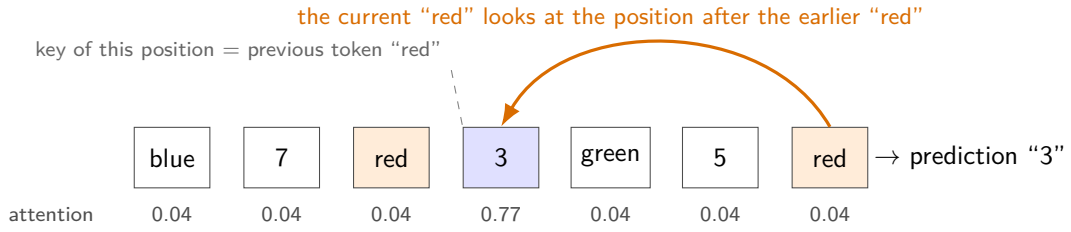


Figure 4 How an induction head works. The last "red" attends strongly to the position after the earlier "red" ("3") and takes out its value "3." The numbers are attention weights.

Let us actually compute it. The vocabulary has 7 kinds (blue, red, green, 3, 5, 7, 9), and the query is multiplied by a factor of 8 to make the attention sharp. Then the dot product is 8 only at the position whose key is "red," and 0 elsewhere. Dividing by $\sqrt{7}$, the score is $8/\sqrt{7} \approx 3.02$ at the position of "3" and 0 at the other six positions. Taking the softmax function,

$$\text{weight at the position of "3"} = \frac{e^{3.02}}{e^{3.02} + 6 \times e^{0}} \approx \frac{20.6}{20.6 + 6} \approx 0.77, \qquad \text{others } \frac{1}{26.6} \approx 0.04,$$

and the output vector has its largest component (0.77) for "3."

If the context is changed to "blue 7 red **9** green 5 red," the prediction changes to "9" with the same weights.

> **The rule (the weights) is fixed, and what to answer is decided by the examples in the context**: this is the essence of in-context learning.

In real models, the processing that puts the information of "the previous token" into the key is also done by Attention in an earlier layer, and such parts are combined over many layers. What we have shown here is the simplest form.

4.3 In-context learning and learning the weights

	Learning the weights (pre-training, SFT, RLHF)	In-context learning
What changes	Parameters (weights)	Only the input (context)
Computation needed	Computing gradients and updating weights	Forward pass only
How long the effect lasts	Remains permanently after training	Only while that input is being processed
Amount that can be used	Trillions of tokens	Up to the context length (thousands to hundreds of thousands of tokens)

Learning the weights can be compared to "engraving into the brain," and in-context learning to "working while looking at notes on the desk." **The effect of in-context learning disappears when the conversation ends.** The notes are cleared away, and the next time the same model is used, it remembers nothing.

5 Context length and the KV cache

When generating text, each time one token is added, every layer computes Attention between the query of the new token and the keys and values of all the tokens so far. The keys and values of past tokens are the same every time, so once computed they are stored and reused. This is called the **KV cache**. It is the same trick as writing down the intermediate results of a calculation so that you do not have to redo the same calculation later.

The size of the KV cache is proportional to the context length. With the configuration of Llama 3 8B (32 layers, 8 key heads and 8 value heads, head dimension 128, 2 bytes per number), each token needs (Appendix B)

$$\underbrace{2}_{\text{K and V}} \times \underbrace{32}_{\text{layers}} \times \underbrace{8}_{\text{heads}} \times \underbrace{128}_{\text{dimensions}} \times \underbrace{2}_{\text{bytes}} = 131{,}072 \text{ bytes} = 128 \text{ KiB}$$

(1 KiB $= 2^{10} = 1024$ bytes). For a context of 8192 tokens this is 1 GiB (2^{30} bytes), and if Llama 3.1 8B, whose context length was extended to about 131,000 (131,072) tokens, is used up to its limit, it becomes 16 GiB, more than the weights (about 8 GB at 8 bits).

In Llama 3, for 32 query heads, the key and value heads are reduced to 8, and every 4 query heads share one set of keys and values (**grouped-query attention**, GQA). Without sharing, the KV cache would be four times larger, 512 KiB per token.

Being able to handle long contexts is by no means "free" in terms of computation and memory. To have more notes on your desk, you need a correspondingly bigger desk.

6 Why hallucinations remain

Even in chat models, **hallucination**, plausibly stating things that are not true, does not go away. From the contents of this series, we can see some of the reasons.

- The goal of pre-training is to write "a plausible continuation"; **truth itself is not part of the goal** (Parts 1 and 2).

- Things not in the training data (such as events after training) are simply not known.[5]
- Answering "I don't know" when the model does not know is not acquired unless it is specifically trained. Researchers at OpenAI (Kalai et al., 2025) argue that a major reason hallucinations remain is that the ways of training and evaluating give higher scores to guessing than to answering "I don't know" when uncertain.

This has the same structure as **a multiple-choice exam in which guessing gives a higher expected score than leaving the answer blank**. With four choices, even a random guess has an expected value of $\frac{1}{4}$ point, while a blank gets 0. If grading like this continues, "pretending to know" pays off.

Countermeasures include RAG, which makes the model search and refer to external documents,[6] leaving calculations and searches to external tools, and emphasizing in training and evaluation that the model should refrain from answering when it is not confident.

7 Summary

1. A pre-trained model is a model that "continues text" and is not built to answer questions. A chat model is made by adding instruction tuning and learning from human preferences.
2. Instruction tuning (SFT) is training that imitates model responses written by people, using cross-entropy over the response part only. The data are far smaller than for pre-training, and it mainly teaches the format and attitude of responses.
3. In learning from human preferences, rewards are estimated from comparisons of two responses with the Bradley–Terry model $P(y_w \text{ wins}) = \sigma(r_w - r_l)$, and the model is trained to raise the reward without moving too far from the original model. The optimal solution has the form $\pi^* \propto \pi_{\text{ref}}\, e^{r/\beta}$, the same form as the Boltzmann distribution.
4. The reward model only approximates people's judgments, so over-optimizing causes reward hacking. The KL divergence penalty suppresses it.
5. In in-context learning, behavior changes according to examples in the context without changing the weights. Attention mechanisms such as induction heads are part of how this works.
6. Longer contexts need proportionally more KV-cache memory. Hallucinations arise partly because the goals of training and evaluation are not truth itself.

Summary of the whole series

Part	What we learned	Training goal
Part 1	Text is split into tokens, becomes numbers through embeddings, and becomes probabilities of the next token through the output layer. Even a counting language model is maximum likelihood estimation	Cross-entropy of the next token
Part 2	The loss decreases following power laws in model size, amount of data and compute, and can be predicted from small experiments	Same as above (trillions of tokens)
Part 3	Imitate model responses, learn a reward from people's comparisons, and raise it. In-context learning does not change the weights	Cross-entropy of the response part; reward − KL penalty

[5] "Introduction to How LLMs Work," Section 5.12.
[6] "Introduction to How LLMs Work," Section 5.13.

Every stage simply decides **"what to take as the loss (goal), on what data, and how to minimize it."** This framework, continuing from the least-squares method, does not change even for chat models. What changes is the choice of goals and data.

Appendix A Mathematical explanations

This appendix uses partial derivatives and the method of Lagrange multipliers (first-year university calculus).

Notation $\mathbb{E}_{y\sim\pi}[f(y)] = \sum_y \pi(y) f(y)$ is the expected value of $f(y)$ when y is chosen according to the distribution π. The input x is sometimes omitted, as in $\pi(y)$ and $r(y)$.

A.1 The gradient of the reward

The derivative of the sigmoid function is

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z)\big(1-\sigma(z)\big).$$

For the loss of one comparison, $\ell = -\log \sigma(\Delta)$ with $\Delta = r_w - r_l$,

$$\frac{d\ell}{d\Delta} = -\frac{\sigma'(\Delta)}{\sigma(\Delta)} = -\big(1-\sigma(\Delta)\big),$$

so from $\partial\Delta/\partial r_w = 1$ and $\partial\Delta/\partial r_l = -1$,

$$\frac{\partial \ell}{\partial r_w} = -\big(1-\sigma(\Delta)\big), \qquad \frac{\partial \ell}{\partial r_l} = +\big(1-\sigma(\Delta)\big).$$

Gradient descent moves in the direction opposite to the gradient, so it raises the reward of the chosen response and lowers that of the one not chosen. The amount $1-\sigma(\Delta)$ is "how much the model did not expect this result"; if the model already expects $r_w \gg r_l$ ($\sigma(\Delta) \approx 1$), it hardly moves anything. The program in Appendix B.1 combines the n comparisons of the same pair and uses n times this gradient.

A.2 The KL divergence is 0 or more

We use $\log t \le t-1$ (with equality only when $t=1$). Setting $t = \pi_{\text{ref}}(y)/\pi(y)$,

$$\begin{aligned} -D_{\text{KL}}(\pi \,\|\, \pi_{\text{ref}}) &= \sum_y \pi(y) \log \frac{\pi_{\text{ref}}(y)}{\pi(y)} \le \sum_y \pi(y) \left(\frac{\pi_{\text{ref}}(y)}{\pi(y)} - 1 \right) \\ &= \sum_y \pi_{\text{ref}}(y) - \sum_y \pi(y) \le 1 - 1 = 0 \end{aligned}$$

(the sums are over y with $\pi(y) > 0$). Therefore $D_{\text{KL}} \ge 0$, with equality only when $\pi(y) = \pi_{\text{ref}}(y)$ for all y. Note that $D_{\text{KL}}(\pi\|\pi_{\text{ref}})$ and $D_{\text{KL}}(\pi_{\text{ref}}\|\pi)$ are not equal in general, so it is called a "divergence" rather than a distance.

A.3 The optimal distribution with the KL penalty

We maximize the objective function

$$J(\pi) = \sum_y \pi(y) r(y) - \beta \sum_y \pi(y) \log \frac{\pi(y)}{\pi_{\text{ref}}(y)}$$

under the condition $\sum_y \pi(y) = 1$. Using a Lagrange multiplier λ, taking the partial derivative of $J + \lambda(\sum_y \pi(y) - 1)$ with respect to $\pi(y)$ and setting it to 0,

$$r(y) - \beta \left(\log \frac{\pi(y)}{\pi_{\text{ref}}(y)} + 1 \right) + \lambda = 0 \quad \Longrightarrow \quad \pi^*(y) = \frac{1}{Z} \pi_{\text{ref}}(y) \exp\left(\frac{r(y)}{\beta} \right)$$

($Z = \sum_y \pi_{\text{ref}}(y) e^{r(y)/\beta}$ is the constant that makes the total 1). J is concave in π (because $-\pi \log \pi$ is concave), so this is the maximum.

A.4 The DPO loss

Solving the formula of A.3 for r gives

$$r(y) = \beta \log \frac{\pi^*(y)}{\pi_{\text{ref}}(y)} + \beta \log Z.$$

Substituting this into $\sigma(r_w - r_l)$ of the Bradley–Terry model, $\beta \log Z$ cancels when the difference is taken, and

$$L_{\text{DPO}} = -\log \sigma \left(\beta \log \frac{\pi(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)} \right).$$

This loss has exactly the same form as in Section 3.2: it raises the probability of the chosen response (relative to the original model) and lowers that of the response not chosen. Instead of the reward r, the ratio of the model's own probabilities plays the role of the "reward."

Appendix B Python code (standard library only)

The code uses no external libraries, only the standard features of Python 3.

B.1 Estimating rewards from pairwise comparisons (Section 3.2)

```
# Estimate rewards from pairwise comparison data (Bradley-Terry model, gradient descent)
import math

# Four responses to the question "Please explain what photosynthesis is"
names = {"A": "accurate and concise", "B": "accurate but wordy",
         "C": "partly wrong", "D": "does not answer the question"}
# (winning response, losing response, count): results of raters comparing two and choosing the better one
data = [("A", "B", 8), ("B", "A", 2), ("A", "C", 9), ("C", "A", 1),
        ("A", "D", 10), ("B", "C", 6), ("C", "B", 4),
        ("B", "D", 8), ("D", "B", 2), ("C", "D", 7), ("D", "C", 3)]

sig = lambda z: 1 / (1 + math.exp(-z))
r = {k: 0.0 for k in names}   # rewards (all start at 0)
eta = 0.05
for step in range(1001):
    L = sum(-n * math.log(sig(r[w] - r[l])) for w, l, n in data)
```

```
    if step in (0, 5, 20, 100, 1000):
        print(f"step {step:4d}  loss {L:.3f}  " + "  ".join(f"r_{k}={v:+.3f}" for k, v in r.items()))
    g = {k: 0.0 for k in names}
    for w, l, n in data:   # dL/dr_w = -n(1-sigma(r_w-r_l)); dL/dr_l has the opposite sign
        s = 1 - sig(r[w] - r[l])
        g[w] -= n * s
        g[l] += n * s
    for k in r:
        r[k] -= eta * g[k]
    m = sum(r.values()) / len(r)   # only differences of rewards matter, so set the mean to 0
    r = {k: v - m for k, v in r.items()}

print("\nresponse  reward   content")
for k in sorted(r, key=r.get, reverse=True):
    print(f"    {k}     {r[k]:+.3f}  {names[k]}")
print("\npair   observed win rate   model win rate sigma(r_i - r_j)")
for i, j in [("A", "B"), ("A", "C"), ("A", "D"), ("B", "C"), ("B", "D"), ("C", "D")]:
    wins = sum(n for w, l, n in data if (w, l) == (i, j))
    total = sum(n for w, l, n in data if {w, l} == {i, j})
    print(f"  {i}>{j}   {wins}/{total} = {wins/total:.2f}         {sig(r[i] - r[j]):.2f}")
```

Output:

```
step    0  loss 41.589  r_A=+0.000  r_B=+0.000  r_C=+0.000  r_D=+0.000
step    5  loss 27.078  r_A=+1.357  r_B=+0.097  r_C=-0.334  r_D=-1.120
step   20  loss 26.615  r_A=+1.731  r_B=+0.085  r_C=-0.427  r_D=-1.390
step  100  loss 26.613  r_A=+1.754  r_B=+0.083  r_C=-0.433  r_D=-1.404
step 1000  loss 26.613  r_A=+1.754  r_B=+0.083  r_C=-0.433  r_D=-1.404

response  reward   content
    A     +1.754  accurate and concise
    B     +0.083  accurate but wordy
    C     -0.433  partly wrong
    D     -1.404  does not answer the question

pair   observed win rate   model win rate sigma(r_i - r_j)
  A>B   8/10 = 0.80         0.84
  A>C   9/10 = 0.90         0.90
  A>D   10/10 = 1.00         0.96
  B>C   6/10 = 0.60         0.63
  B>D   8/10 = 0.80         0.82
  C>D   7/10 = 0.70         0.73
```

B.2 The optimal distribution with the KL penalty (Section 3.3)

```
# A distribution that maximizes reward without moving too far from the original model:
# pi*(y) is proportional to pi_ref(y) exp(r(y)/beta)
import math

r = {"A": 1.754, "B": 0.083, "C": -0.433, "D": -1.404}   # rewards estimated in B.1
ref = {"A": 0.30, "B": 0.40, "C": 0.20, "D": 0.10}       # probabilities of the model after instruction tuning (assumed)

def optimal(beta):
    w = {k: ref[k] * math.exp(r[k] / beta) for k in r}
    Z = sum(w.values())
    return {k: v / Z for k, v in w.items()}

print(" beta   pi(A) pi(B) pi(C) pi(D)  expected reward  KL(pi||pi_ref)")
for beta in (math.inf, 2.0, 1.0, 0.5, 0.2, 0.05):
```

```
    p = ref if beta == math.inf else optimal(beta)
    Er = sum(p[k] * r[k] for k in r)
    KL = sum(p[k] * math.log(p[k] / ref[k]) for k in r if p[k] > 0)
    print(f"{'inf' if beta == math.inf else beta:>5}  " + " ".join(f"{p[k]:.3f}" for k in r)
          + f"     {Er:+.3f}          {KL:.3f}")
```

Output:

```
 beta   pi(A) pi(B) pi(C) pi(D)  expected reward  KL(pi||pi_ref)
  inf  0.300 0.400 0.200 0.100     +0.332          0.000
  2.0  0.535 0.309 0.119 0.037     +0.860          0.131
  1.0  0.746 0.187 0.056 0.011     +1.286          0.443
  0.5  0.947 0.045 0.008 0.001     +1.660          0.962
  0.2  1.000 0.000 0.000 0.000     +1.753          1.201
 0.05  1.000 0.000 0.000 0.000     +1.754          1.204
```

B.3 Computing an induction head (Section 4.2)

```
# An example of the mechanism of in-context learning:
# "If B came after A earlier, output B after A now"
# (reproducing, with hand-chosen weights, the "induction head" that Olsson et al. 2022 found in trained models)
import math

vocab = ["blue", "red", "green", "3", "5", "7", "9"]
onehot = lambda t: [1.0 if v == t else 0.0 for v in vocab]
s = 8.0   # factor (weight) that makes the attention sharp

def predict(seq):
    # key of position j = "the token at the previous position"; value = "the token at position j"
    # (in real models, putting the previous token's information into the key is also done
    #  by Attention in an earlier layer)
    K = [[0.0] * len(vocab)] + [onehot(t) for t in seq[:-1]]
    V = [onehot(t) for t in seq]
    q = [s * x for x in onehot(seq[-1])]   # query of the last position = "the current token"
    scores = [sum(a * b for a, b in zip(q, k)) / math.sqrt(len(vocab)) for k in K]
    m = max(scores); e = [math.exp(x - m) for x in scores]; Z = sum(e)
    att = [x / Z for x in e]
    out = [sum(att[j] * V[j][c] for j in range(len(seq))) for c in range(len(vocab))]
    print("token     " + "".join(f"{t:>7s}" for t in seq))
    print("key       " + "".join(f"{t:>7s}" for t in ["-"] + seq[:-1]))
    print("attention " + "".join(f"{a:>7.2f}" for a in att))
    print("output    " + ", ".join(f"{v} {o:.2f}" for v, o in zip(vocab, out) if o > 0.001))
    print("prediction ->", vocab[out.index(max(out))], "\n")

predict(["blue", "7", "red", "3", "green", "5", "red"])   # in this context, "3" follows "red"
predict(["blue", "7", "red", "9", "green", "5", "red"])   # same weights, but a different context gives a different answer
predict(["red", "5", "green", "3", "blue", "9", "blue"])
```

Output:

```
token        blue      7    red      3  green      5    red
key             -   blue      7    red      3  green      5
attention    0.04   0.04   0.04   0.77   0.04   0.04   0.04
output    blue 0.04, red 0.08, green 0.04, 3 0.77, 5 0.04, 7 0.04
prediction -> 3

token        blue      7    red      9  green      5    red
key             -   blue      7    red      9  green      5
attention    0.04   0.04   0.04   0.77   0.04   0.04   0.04
output    blue 0.04, red 0.08, green 0.04, 5 0.04, 7 0.04, 9 0.77
```

```
prediction -> 9

token         red      5  green      3   blue      9   blue
key             -    red      5  green      3   blue      9
attention    0.04   0.04   0.04   0.04   0.04   0.77   0.04
output    blue 0.08, red 0.04, green 0.04, 3 0.04, 5 0.04, 9 0.77
prediction -> 9
```

B.4 Size of the KV cache (Section 5)

```
# Size of the KV cache (configuration of Llama 3 8B)
layers, kv_heads, head_dim, bytes_per = 32, 8, 128, 2   # 32 layers, 8 K/V heads, 128 dims each, FP16
per_token = 2 * layers * kv_heads * head_dim * bytes_per  # 2 for the two kinds, K and V
print(f"per token: {per_token:,} bytes = {per_token/1024:.0f} KiB")
for n in (8192, 131072):
    print(f"context of {n:>7,} tokens: {per_token*n/2**30:.0f} GiB")
print(f"(if the K/V heads were 32, the same as Q: {4*per_token/1024:.0f} KiB per token)")
```

Output:

```
per token: 131,072 bytes = 128 KiB
context of   8,192 tokens: 1 GiB
context of 131,072 tokens: 16 GiB
(if the K/V heads were 32, the same as Q: 512 KiB per token)
```