LLM Basics (2): Pre-training and Scaling Laws

(for high school and first-year university students)

How loss depends on model size, data and compute

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

In Part 1, we saw that a language model is a device that outputs "the probability of the next token," and that it learns by making the degree of mis-prediction (the cross-entropy) smaller by gradient descent. Part 2 deals with **pre-training**, in which this is done with **trillions of tokens of text**.

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

The central topic is an empirical rule (**scaling laws**): **when the model size, the amount of data and the amount of compute are increased, the loss decreases regularly, following power laws**. Because of this regularity, the performance of a huge model can be predicted from experiments with small models before building it. This is like testing a scale model in a wind tunnel to estimate the performance of a real airplane before building it.[a]

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

How to read this booklet The main text is written so that it can be read with high-school mathematics (exponential functions and logarithms, common logarithms and log-log graphs, differentiation, and the idea of the least-squares method). The **mathematical foundations**, such as estimating the amount of computation using matrices, the relation between information and code length, and the derivation of the optimal allocation, 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

- N: the number of parameters (weights) of the model; D: the number of tokens used for training; C: the amount of computation for training (the number of multiplications and additions of numbers with decimals, counted in units written FLOP).

- L: the loss (cross-entropy per token, natural logarithm). As in Section 5.1 of Part 1, e^L is the perplexity (how many choices the model is effectively hesitating between).
- α, β: exponents of power laws (positive constants). E, A and B are also constants determined by fitting to data.
- $y \propto x^{-\alpha}$ means "y is proportional to $x^{-\alpha}$."
- Large numbers such as 10^{23} are also written `1e23` (the notation of the programs in Appendix B). $\log_{10}$ is the common logarithm.

1 Pre-training

1.1 What it does

The content of pre-training is the same as in Part 1: "predict the next token at each position in a text, and make the cross-entropy

$$L = -\frac{1}{T}\sum_{t=1}^{T} \log p(x_t \mid x_1, \ldots, x_{t-1})$$

smaller." The correct answer (the next token) comes from the text itself, so no person needs to attach correct-answer labels (**self-supervised learning**; see "Supervised and Unsupervised Learning").

> **Fill-in-the-blank questions**, in which words are hidden one at a time and you guess "what comes next," can be made automatically from any text. Therefore **as much text as can be obtained**, such as text on the Web, books, papers and the source code of programs, can be used as a workbook.

Here are some examples of the scale.

Model	Released	Parameters N	Training tokens D
Chinchilla	2022 (DeepMind)	70 billion	1.4 trillion
Llama 3 8B	2024 (Meta)	8 billion	over 15 trillion

15 trillion tokens is, in terms of novels of about 100,000 words each, roughly more than 100 million books.[1] The model reads an amount of text that no person could read in a lifetime, and practices filling in the blanks.

The training method is the same as in "Deep Learning (2)": stochastic gradient descent with mini-batches (an improved version of it). However, since the amount of text is enormous, the same text is rarely used over and over; **most texts are read only about once**.

1.2 What can be learned just by "guessing the next one"?

The task of guessing the next token is simple, but to guess well, the model needs to know many things.

- "Yesterday I ___ a movie with a friend." → grammar (the form of the verb)
- "At 1 atmosphere, water boils at about ___ °C." → factual knowledge
- "17 + 25 = ___" → calculation
- "A is taller than B. B is taller than C. So the tallest is ___." → reasoning

[1] The correspondence between tokens and words differs between languages and tokenizers; here we assume 1 token ≈ 0.75 English words, so this is only a rough guide.

- The continuation of a program → the grammar of a programming language and the intention of the processing

> The training data contain all these kinds of text mixed together, so to lower the loss, **the model has no choice but to build an internal representation that can handle all of them together**. Imagine a student who keeps working through workbooks of every subject to raise their fill-in-the-blank score, and as a result gradually acquires grammar, knowledge and calculation skills. The fact that "knowledge is represented in a distributed way across all the parameters," which we saw in "Introduction to How LLMs Work," is the result of this pressure.

1.3 To predict is to compress

Here is a somewhat surprising point of view: **a model that predicts well can compress text into a shorter form**.

Remember Morse code. The most frequently used letter in English, E, is a single "·", and T is a single "−"; these are short, while the rarely used Q is the long "− − · −". If you **assign short codes to frequent things and long codes to rare things**, you can send a text in a shorter form overall.

In information theory, it is known that the length of code needed to convey an event that occurs with probability p is $-\log_2 p$ bits (Shannon). For example, probability 1/2 needs 1 bit, and 1/8 needs 3 bits. If a text is encoded using the probabilities of a language model, the code length per token is the cross-entropy measured with base-2 logarithms.[2]

> In other words, **lowering the loss = becoming able to compress text more**. The better a model guesses, the better it captures the regularities in text (grammar, facts, logic), and the shorter it can represent the text.

Conversely, no matter how good a model is, it cannot compress the genuine uncertainty contained in the text itself (the part where what to write next is up to the writer). Just as no one, however clever, can predict the roll of a die. This corresponds to the "**irreducible loss**" E of Section 3.2.

2 The compute for training: $C \approx 6ND$

Before turning to scaling laws, let us estimate how much computation training requires. The conclusion is:

> $$C \approx 6ND$$
>
> The compute for training is **proportional to the product of the number of parameters and the number of training tokens, with a factor of about 6**.

The reason is roughly as follows.[3]

[2] L is measured with natural logarithms, so to convert it to bits we divide by $\log 2$: $L/\log 2 \approx 1.443L$ bits. See Appendix A.1 for details.

[3] A detailed explanation using matrix products is given in Appendix A.2. Attention also contains computations that do not use weights, but they are small unless the context is extremely long compared to the dimension of the model, so we ignore them.

- **Forward pass** (computing the prediction): When processing one token, each weight takes about 2 operations, one multiplication ("multiply by the input") and one addition ("add up the results"). For the whole model, this is about $2N$ operations.
- **Backward pass** (sending the error back): In each layer, two kinds of quantities are computed, "the error sent to the previous layer" and "the gradient of the weights," so it takes about twice the forward pass, $4N$ operations.

In total, this is $6N$ operations per token, and $6ND$ operations for D tokens.

Examples

- Chinchilla: $C = 6 \times (7 \times 10^{10}) \times (1.4 \times 10^{12}) \approx 5.9 \times 10^{23}$ FLOP
- Llama 3 8B: $C = 6 \times (8 \times 10^{9}) \times (1.5 \times 10^{13}) = 7.2 \times 10^{23}$ FLOP

If we used 1000 GPUs, each actually performing 4×10^{14} operations per second, these would take about 17 days and about 21 days, respectively (Appendix B). With a single GPU, they would take about 47 years and about 57 years.[4]

3 Scaling laws

3.1 Power laws

Kaplan et al. (OpenAI, 2020) trained many models of different sizes and reported that **the loss decreases following a power law with respect to each of the number of parameters, the amount of data and the amount of compute**. For example, when there is enough data, the relation with the number of parameters N (excluding the embedding matrix)

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}, \qquad \alpha_N \approx 0.076$$

held over a range of N spanning many orders of magnitude (N_c is a constant).

Taking the common logarithm of both sides gives

$$\log_{10} L = \alpha_N \log_{10} N_c - \alpha_N \log_{10} N,$$

which is **a straight line with slope $-\alpha_N$ on a log-log graph**, with $\log_{10} N$ on the horizontal axis and $\log_{10} L$ on the vertical axis.

> A power law is a law in which **"if N is multiplied by some factor, L is multiplied by a fixed factor," regardless of the size of N**. In this example,
>
> $$\frac{L(10N)}{L(N)} = 10^{-0.076} \approx 0.84,$$
>
> that is, **every time the parameters are increased tenfold, the loss decreases by about 16%**. Going from 10 billion to 100 billion parameters, or from 100 billion to 1 trillion, the rate of decrease is the same.

In general, if $L(N) = cN^{-\alpha}$, then for any factor k,

$$\frac{L(kN)}{L(N)} = \frac{c\,(kN)^{-\alpha}}{c\,N^{-\alpha}} = k^{-\alpha},$$

[4] 17 days × 1000 = 17,000 days ≈ 47 years; 21 days × 1000 = 21,000 days ≈ 57 years.

and N does not appear on the right-hand side. Since it looks the same from any size, a power law is called **scale-free** (scale-invariant). It is like a coastline that looks the same shape even when the scale of the map is changed.

An exponential function e^{-N/N_0} has a "special size" N_0, beyond which it rapidly approaches 0. A power law has no such size where "it ends here."

> Many phenomena in nature and society are known to follow power laws, for example the **Gutenberg–Richter law**, familiar in earth science (the relation between the magnitude and number of earthquakes: earthquakes one magnitude larger occur about one tenth as often), and the relation between the frequency and rank of words in a text (**Zipf's law**). The loss of LLMs has joined this family.

3.2 The Chinchilla formula: considering both size and data

Hoffmann et al. (DeepMind, 2022) trained more than 400 models, varying both the number of parameters N and the number of training tokens D, and expressed the loss in the following form.

$$L(N, D) = E + \frac{A}{N^\alpha} + \frac{B}{D^\beta}$$

The values obtained by fitting are $E = 1.69$, $A = 406.4$, $B = 410.7$, $\alpha = 0.34$, $\beta = 0.28$. The three terms have the following meanings.

- E: **the irreducible loss**. It is the uncertainty of the text itself, and it remains even if an arbitrarily large model is trained on infinite data (Section 1.3).
- A/N^α: **the loss due to the finite size of the model**. It approaches 0 as N increases.
- B/D^β: **the loss due to the finite amount of data**. It approaches 0 as D increases.

> Compared to a student (the model) studying with a textbook (the data), E is the points lost on "questions nobody could predict in the first place," A/N^α the points lost because "the student's head does not have enough capacity," and B/D^β the points lost because "the textbook is too thin." **Even an excellent student is limited by a thin textbook, and even a thick textbook is limited by the student's capacity.**

Figure 1 shows the loss given by this formula when D is fixed and N is varied. When there is little data ($D = 10^9$), the loss stops at the data term however large the model is made. Conversely, however much data there is ($D \to \infty$, the black line in the figure), the loss does not go below $E + A/N^\alpha$.

Note that the value of the loss itself depends on the kind of training data and on the tokenizer, so it cannot be compared directly with the loss of a different model. The numbers here are for seeing the shape of the relation expressed by this formula.

For example, for the size of Llama 3 8B ($N = 8 \times 10^9$), the breakdown computed with this formula is as follows (Appendix B).

D	E	A/N^α	B/D^β	L
2×10^{11} (200 billion)	1.69	0.175	0.281	2.146
1.5×10^{13} (15 trillion)	1.69	0.175	0.084	1.949
∞	1.69	0.175	0	1.865

Increasing the data 75 times lowers the data term from 0.281 to 0.084, but the model-size term 0.175 does not change at all. However thick the textbook, the points lost to the capacity of

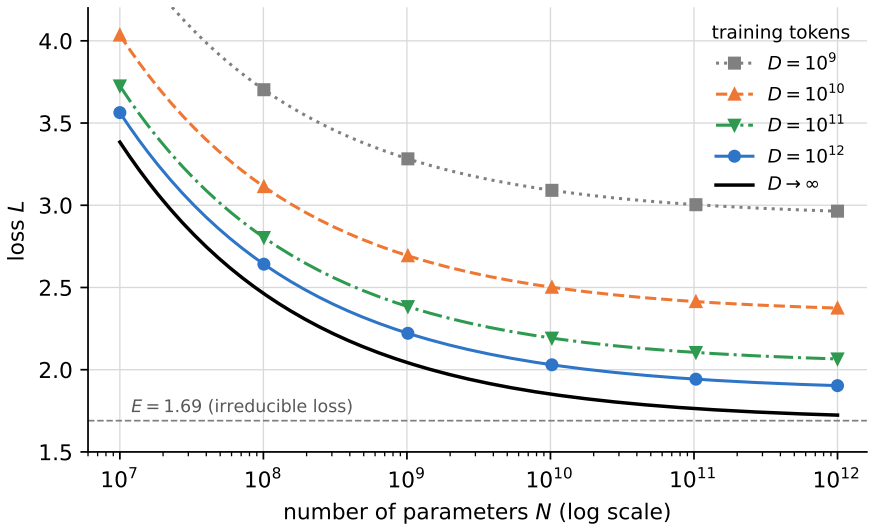


Figure 1 Loss given by the Chinchilla formula, for each number of training tokens D, as the number of parameters N is varied. The dashed line is the irreducible loss E.

the head do not decrease.

3.3 The optimal allocation for a given compute budget

When the available compute C is fixed, the product of N and D is constant, because $C = 6ND$. **Just as in a rectangle of constant area, making it taller makes it narrower**, the choice is between "training a large model on little data" and "training a small model on a lot of data."

If we plot the loss against N with C fixed, we get **a valley shape**, and there is an N at which the loss is smallest (Figure 2). On the left side of the valley the model is too small (not enough capacity), and on the right side the data are too few (not enough textbook).

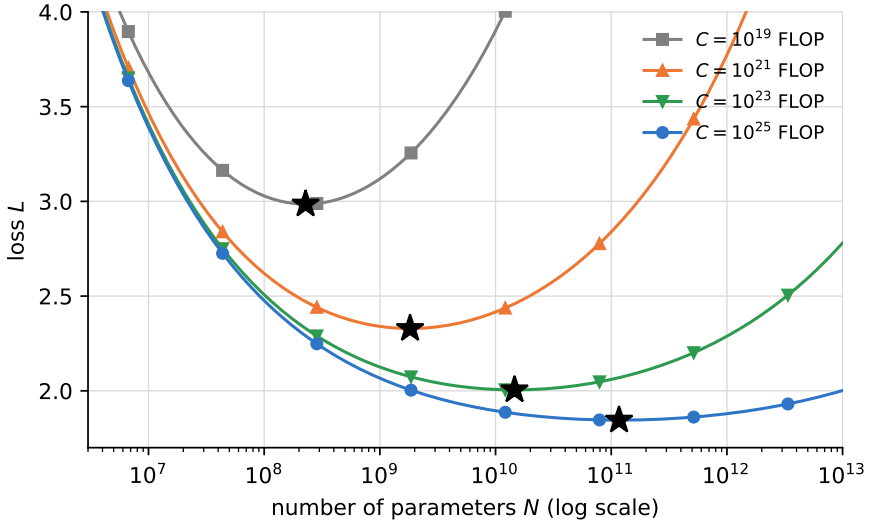


Figure 2 Loss versus the number of parameters when the compute C is fixed ($D = C/6N$). The stars mark the minimum for each C.

The bottom of the valley is found by substituting $D = C/(6N)$ into the loss formula, differ-

entiating with respect to N, and setting the result to 0 (Appendix A.3). The result is

$$N_{\text{opt}} \propto C^{\beta/(\alpha+\beta)}, \qquad D_{\text{opt}} \propto C^{\alpha/(\alpha+\beta)},$$

and the two exponents add up to 1. If α and β are close, both exponents are close to $\frac{1}{2}$.

That is, **when increasing the compute, it is optimal to increase the number of parameters and the number of training tokens in nearly the same proportion**. If the compute is increased 100 times, both the model and the data should be increased about 10 times each.

The conclusion of Hoffmann et al. was that "the number of parameters and the number of training tokens should be increased in the same proportion, and **about 20 tokens per parameter is a good guide**." Following this idea, with the same compute as the large model of the time, Gopher (280 billion parameters, 300 billion tokens), they trained Chinchilla, a quarter of the size (70 billion parameters), on more than four times the data (1.4 trillion tokens), and obtained better performance than Gopher.

However, if the optimal allocation is computed directly with the coefficients of the formula above, the number of tokens per parameter comes out considerably larger than 20. In 2024, Besiroglu et al. read the data off the figures in the paper and refitted them, obtaining $E = 1.82$, $A = 482.01$, $B = 2085.43$, $\alpha = 0.3478$, $\beta = 0.3658$; these agree with the guide of about 20 tokens (Appendix B).

C (FLOP)	H: N_{opt}	H: D/N	B: N_{opt}	B: D_{opt}	B: D/N
10^{19}	2.3×10^{8}	32	2.6×10^{8}	6.4×10^{9}	24
10^{21}	1.8×10^{9}	50	2.8×10^{9}	6.0×10^{10}	22
5.88×10^{23}	3.3×10^{10}	93	7.3×10^{10}	1.3×10^{12}	18
10^{25}	1.2×10^{11}	122	3.1×10^{11}	5.3×10^{12}	17

(H: optimal allocation with the coefficients of Hoffmann et al.; B: with those of Besiroglu et al.)

With the coefficients of Besiroglu et al., the optimal allocation for Chinchilla's compute of 5.88×10^{23} is about 73 billion parameters and 1.3 trillion tokens, in good agreement with the actual Chinchilla (70 billion, 1.4 trillion). Even for the same experiment, the fitted coefficients can change this much depending on how the data are handled.

3.4 Why models are trained more than the "optimum"

Llama 3 8B was trained on 15 trillion tokens, about 75 times the Chinchilla guide (about 200 billion tokens for 8 billion parameters), or about 1900 tokens per parameter. Meta reported that the performance kept improving even far beyond the guide (the loss kept decreasing almost linearly with respect to the logarithm of the number of training tokens).

This is because the "optimum" of Section 3.3 is **the optimum when only the compute for training is considered**. A finished model is used hundreds of millions of times, so the compute for using it (about $2N$ per token) must also be considered.

For a car that millions of people ride every day, a small, fuel-efficient car is cheaper overall, even if it takes more effort to build. In the same way, **training a small model for a long time can be cheaper once the stage of using it is included**. Scaling laws are tools to be used according to what you want to optimize.

4 Predicting large models from small experiments

The practical value of scaling laws is that **the loss of a large model not yet built can be predicted from experiments with small models**. Let us try this with the least-squares method (Appendix B).

Example Let the "true law" be $L = 1.69 + 406.4/N^{0.34}$ (Section 3.2 with $D \to \infty$), and suppose the loss was "measured" for seven models from $N = 10^7$ to 10^{10}. A scatter of about 0.5% was added to the measured values with random numbers.

N	10^7	3.2×10^7	10^8	3.2×10^8	10^9	3.2×10^9	10^{10}
L	3.406	2.856	2.465	2.205	2.033	1.930	1.842

(a) Subtract E and fit a straight line on log-log axes Taking the common logarithm of both sides of $L - E = A/N^\alpha$ gives $\log_{10}(L - E) = \log_{10} A - \alpha \log_{10} N$, so we set $x = \log_{10} N$ and $y = \log_{10}(L - E)$ and fit a straight line $y = wx + c$ by least squares ($w = S_{xy}/S_{xx}$ from "Supervised and Unsupervised Learning"). The result is $\alpha = -w = 0.349$ (true value 0.34) and $A = 478.1$, and the points lie neatly on a straight line, as in Figure 3(a).

(b) Fit without subtracting E Fitting $\log_{10} L$ directly with a straight line gives an apparent exponent of 0.087.

Using each formula to predict the loss at $N = 10^{11}$ and 10^{12}, which were not measured, gave the following.

N	True L	Prediction (a)	Prediction (b)
10^{11}	1.764	1.760	1.413
10^{12}	1.724	1.721	1.155

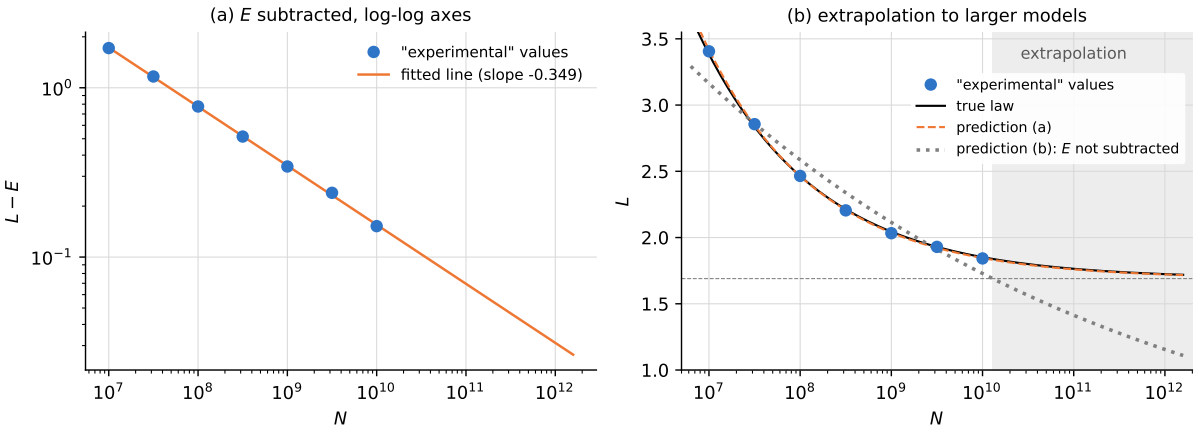


Figure 3 (a) After subtracting the irreducible loss E, the values form a straight line on a log-log graph. (b) Prediction from the measured range (white) into the extrapolated range (gray). The line (b), which does not subtract E, wrongly predicts that the loss keeps decreasing below E.

Prediction (a) almost agrees with the true values. Prediction (b) is not far off within the measured range, yet when extrapolated it predicts that the loss keeps decreasing until it falls below E (Figure 3(b)).

What form of formula is assumed (here, whether there is an irreducible loss) **strongly affects the result of extrapolation**. Agreeing well within the measured range does not guarantee being correct outside it.

In fact, there is a report that before training a large model, scaling laws were fitted to a group of small models using up to about 1/10,000 of the compute, and the loss of the finished model was predicted (the technical report of OpenAI's GPT-4, 2023). This is exactly the method of "predicting the real thing with a scale model."

5 What scaling laws show

5.1 Diminishing returns, but no stopping

Under a power law, the loss keeps decreasing as resources are increased, but the resources needed to lower it by the same amount keep multiplying. In the Chinchilla formula, to halve the loss due to the model size, $A/N^{0.34}$, N must be multiplied by $2^{1/0.34} \approx 7.7$; to reduce it to one tenth, by about 870.[5]

This is similar to how raising a test score from 50 to 70 is relatively easy, while raising it from 90 to 95 is much harder. However, under a power law, **every time resources are increased by the same factor, the loss surely keeps decreasing by the same proportion, and there is no "wall" where it levels off** (until it approaches E). This is the background to why LLM development has become a race of huge investments in computing resources.

5.2 "Emergence" of abilities

The loss decreases smoothly, but when measured on individual tasks (for example, the accuracy of three-digit addition), it can look as if the score is almost 0 up to a certain size and then the task suddenly becomes solvable beyond it. This is sometimes called the **emergence** of abilities (Wei et al., 2022).

Against this, it has been pointed out that the smooth improvement only looks like a sudden change because of the way it is evaluated, such as "1 point only when completely correct" (Schaeffer et al., 2023).

Think of **grading without partial credit**. Even if the probability p of getting each digit of a three-digit answer right rises smoothly, the probability that all three digits are right is p^3. For example, $p = 0.2$ gives 0.008, $p = 0.5$ gives 0.125, and $p = 0.9$ gives 0.729: it stays almost 0 at first and then appears to rise suddenly from some point. **Depending on what is measured and how, the same change can look continuous or discontinuous.**

6 Summary

1. Pre-training is self-supervised learning in which a model learns to predict the next token from an enormous amount of text such as the Web. Predicting well requires grammar, knowledge and reasoning, and lowering the loss is the same as becoming able to compress text.
2. The compute for training can be estimated as $C \approx 6ND$ (number of parameters $\times$

[5] From $A/(kN)^{0.34} = \frac{1}{2} \cdot A/N^{0.34}$ we get $k^{0.34} = 2$, so $k = 2^{1/0.34}$. For one tenth, $k = 10^{1/0.34} \approx 870$.

number of tokens $\times$ 6).

3. The loss decreases following power laws with respect to the number of parameters, the amount of data and the amount of compute. A power law is a straight line on a log-log graph and is scale-free.
4. The Chinchilla formula $L = E + A/N^{\alpha} + B/D^{\beta}$ is the sum of the irreducible loss, the loss due to the model size and the loss due to the amount of data. With the compute fixed, it is optimal to increase N and D in nearly the same proportion (about 20 tokens per parameter).
5. In practice, taking into account the compute for using the model, small models are often trained on more data.
6. Scaling laws make it possible to predict the loss of a large model from small experiments. However, the prediction depends on the form of the formula assumed.

Coming next A pre-trained model can "continue a text," but it is not built to answer questions or follow instructions. Part 3 deals with **instruction tuning** and **learning from human feedback**, which turn it into a chat model, and with **in-context learning**, in which behavior changes without changing the weights.

Appendix A Mathematical explanations

This appendix uses matrices and vectors and the differentiation of functions with real exponents (upper high school to first-year university level).

Notation $x_{<t}$ stands for $x_1, \ldots, x_{t-1}$ together. Token vectors are treated as row vectors, and matrices are multiplied from the right, as in $\mathbf{y} = \mathbf{x}W$. The superscript T denotes the transpose, and δ is the error sent back from later layers in backpropagation.

A.1 To predict is to compress

The amount of information needed to convey an event that occurs with probability p is $-\log_2 p$ bits (Shannon). Representing frequent events with short codes and rare events with long codes is the most efficient encoding, and with a method called **arithmetic coding**, encoding with almost exactly this length is actually possible.

Therefore, if a text is encoded using the probabilities of a language model, the code length per token is

$$-\frac{1}{T}\sum_{t} \log_2 p(x_t \mid x_{<t}) \text{ bits.}$$

By the change-of-base formula $\log_2 p = \log p / \log 2$, this is exactly the cross-entropy L measured with natural logarithms divided by $\log 2$:

$$\frac{L}{\log 2} \approx 1.443\, L \text{ bits.}$$

That is, lowering the loss = becoming able to compress text more. It is also known, as Shannon's source coding theorem, that text cannot be compressed below the genuine uncertainty it contains (the entropy of the true distribution). This corresponds to the irreducible loss E.

A.2 Why $C \approx 6ND$?

Most of the computation in a Transformer consists of matrix products $\mathbf{y} = \mathbf{x}W$. If W is $d \times d'$, one token requires dd' multiplications and almost the same number of additions. That is, each weight takes about 2 operations per token, and the forward pass of the whole model takes about $2N$ operations.

In the backward pass, each layer computes two kinds of matrix products,

- the error sent to the previous layer, δW^{T}, and
- the gradient of the weights, $\mathbf{x}^{\mathsf{T}}\delta$

(backpropagation in "Deep Learning (2)"), so it takes about twice the forward pass, $4N$ operations. In total, this is $6N$ operations per token, and $6ND$ operations for D tokens.

There are also computations that do not use weights, such as QK^{T} in Attention, but they are small unless the context length is extremely long compared to the dimension of the model, so we ignore them.

A.3 Derivation of the optimal allocation for a given compute

Substituting $D = C/(6N)$ gives

$$L = E + AN^{-\alpha} + B\left(\frac{6N}{C}\right)^{\beta}.$$

Differentiating with respect to N and setting the result to 0,

$$-\alpha AN^{-\alpha-1} + \beta B\left(\frac{6}{C}\right)^{\beta} N^{\beta-1} = 0 \quad \Longrightarrow \quad N^{\alpha+\beta} = \frac{\alpha A}{\beta B}\left(\frac{C}{6}\right)^{\beta},$$

so

$$N_{\text{opt}} = G\left(\frac{C}{6}\right)^{\frac{\beta}{\alpha+\beta}}, \qquad D_{\text{opt}} = \frac{1}{G}\left(\frac{C}{6}\right)^{\frac{\alpha}{\alpha+\beta}}, \qquad G = \left(\frac{\alpha A}{\beta B}\right)^{\frac{1}{\alpha+\beta}}.$$

The two exponents add up to 1, so when the compute is increased, it is divided between N and D in the ratio $\beta : \alpha$ (on a logarithmic scale). If α and β are close, it is optimal to increase N and D in nearly the same proportion.

The second derivative of L, $\alpha(\alpha+1)AN^{-\alpha-2} + \beta(\beta-1)B(6/C)^{\beta}N^{\beta-2}$, does not have a definite sign by itself. But regarded as a function of $\log N$, L is (a constant plus) a sum of positive multiples of the two exponential functions $e^{-\alpha \log N}$ and $e^{\beta \log N}$, which is convex, so the point found is the minimum. This is the problem of "minimizing L under the condition $C = 6ND$," so solving it with the method of Lagrange multipliers gives the same result.

With the coefficients of Hoffmann et al., $\beta/(\alpha+\beta) = 0.452$; with those of Besiroglu et al., 0.513 (Appendix B).

Appendix B Python code (standard library only)

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

B.1 Compute and the Chinchilla formula (Sections 2 and 3)

```
# Compute C ~ 6ND, and the loss and optimal allocation from the Chinchilla formula (standard library only)

# (1) Training compute C = 6 N D (N: number of parameters, D: number of training tokens)
```

```
for name, N, D in [("Chinchilla", 70e9, 1.4e12), ("Llama 3 8B", 8e9, 15e12)]:
    C = 6 * N * D
    days = C / 4e14 / 1000 / 86400   # suppose we use 1000 GPUs doing 4x10^14 FLOP/s each
    print(f"{name:11s} N={N:.0e} D={D:.1e} C={C:.2e} FLOP (about {days:.0f} days on 1000 GPUs)")

# (2) Loss formula L(N, D) = E + A/N^alpha + B/D^beta
fits = {"Hoffmann et al. (2022)":  dict(E=1.69, A=406.4, B=410.7, a=0.34, b=0.28),
        "Besiroglu et al. (2024)": dict(E=1.82, A=482.01, B=2085.43, a=0.3478, b=0.3658)}

def loss(N, D, E, A, B, a, b):
    return E + A / N**a + B / D**b

def optimum(C, E, A, B, a, b):
    """N and D that minimize L under the constraint C = 6ND (solution in Appendix A.3)"""
    G = (a * A / (b * B)) ** (1 / (a + b))
    N = G * (C / 6) ** (b / (a + b))
    return N, C / (6 * N)

for fname, f in fits.items():
    print(f"\n--- Coefficients of {fname} ---")
    print(f"exponent of N: beta/(alpha+beta) = {f['b']/(f['a']+f['b']):.3f}, "
          f"exponent of D: alpha/(alpha+beta) = {f['a']/(f['a']+f['b']):.3f}")
    for C in (1e19, 1e21, 5.88e23, 1e25):
        N, D = optimum(C, **f)
        print(f"C={C:.2e}: optimal N={N:.2e} D={D:.2e} D/N={D/N:6.1f}  L={loss(N, D, **f):.3f}")

f = fits["Hoffmann et al. (2022)"]
print("\n--- Loss computed with the coefficients of Hoffmann et al. ---")
for N, D in [(70e9, 1.4e12), (8e9, 2e11), (8e9, 15e12), (8e9, float("inf"))]:
    print(f"N={N:.0e} D={D:.1e}: L={loss(N, D, **f):.3f} "
          f"(E={f['E']}, A/N^a={f['A']/N**f['a']:.3f}, B/D^b={f['B']/D**f['b']:.3f})")
```

Output:

```
Chinchilla  N=7e+10 D=1.4e+12 C=5.88e+23 FLOP (about 17 days on 1000 GPUs)
Llama 3 8B  N=8e+09 D=1.5e+13 C=7.20e+23 FLOP (about 21 days on 1000 GPUs)

--- Coefficients of Hoffmann et al. (2022) ---
exponent of N: beta/(alpha+beta) = 0.452, exponent of D: alpha/(alpha+beta) = 0.548
C=1.00e+19: optimal N=2.28e+08 D=7.31e+09 D/N=  32.1  L=2.986
C=1.00e+21: optimal N=1.82e+09 D=9.14e+10 D/N=  50.1  L=2.329
C=5.88e+23: optimal N=3.25e+10 D=3.02e+12 D/N=  92.8  L=1.930
C=1.00e+25: optimal N=1.17e+11 D=1.43e+13 D/N= 122.1  L=1.845

--- Coefficients of Besiroglu et al. (2024) ---
exponent of N: beta/(alpha+beta) = 0.513, exponent of D: alpha/(alpha+beta) = 0.487
C=1.00e+19: optimal N=2.62e+08 D=6.36e+09 D/N=  24.2  L=2.930
C=1.00e+21: optimal N=2.78e+09 D=6.00e+10 D/N=  21.6  L=2.308
C=5.88e+23: optimal N=7.30e+10 D=1.34e+12 D/N=  18.4  L=1.977
C=1.00e+25: optimal N=3.12e+11 D=5.34e+12 D/N=  17.1  L=1.915

--- Loss computed with the coefficients of Hoffmann et al. ---
N=7e+10 D=1.4e+12: L=1.937 (E=1.69, A/N^a=0.083, B/D^b=0.163)
N=8e+09 D=2.0e+11: L=2.146 (E=1.69, A/N^a=0.175, B/D^b=0.281)
N=8e+09 D=1.5e+13: L=1.949 (E=1.69, A/N^a=0.175, B/D^b=0.084)
N=8e+09 D=inf: L=1.865 (E=1.69, A/N^a=0.175, B/D^b=0.000)
```

B.2 Fitting and extrapolating a power law (Section 4)

```
# Estimate a power law by least squares from the "experimental results" of small models,
# and predict the loss of larger models
import math, random

E, A, alpha = 1.69, 406.4, 0.34   # the "true" law L = E + A / N^alpha (assuming enough data)
random.seed(1)
Ns = [10 ** (7 + 0.5 * i) for i in range(7)]                          # 7 models from 1e7 to 1e10
Ls = [(E + A / N**alpha) * (1 + random.gauss(0, 0.005)) for N in Ns]  # 0.5% measurement scatter

def fit(xs, ys):
    """Least squares for the line y = w x + c ("Supervised and Unsupervised Learning")"""
    mx, my = sum(xs) / len(xs), sum(ys) / len(ys)
    Sxx = sum((x - mx) ** 2 for x in xs)
    Sxy = sum((x - mx) * (y - my) for x, y in zip(xs, ys))
    w = Sxy / Sxx
    return w, my - w * mx

# (a) Subtract E, then fit a straight line on log-log axes: log(L - E) = log A - alpha log N
w, c = fit([math.log10(N) for N in Ns], [math.log10(L - E) for L in Ls])
print(f"(a) E subtracted:      estimated alpha = {-w:.3f} (true value {alpha}), estimated A = {10**c:.1f}")
# (b) Fit log L with a straight line without subtracting E
w2, c2 = fit([math.log10(N) for N in Ns], [math.log10(L) for L in Ls])
print(f"(b) E not subtracted:  apparent exponent = {-w2:.3f}")

for N, L in zip(Ns, Ls):
    print(f"   N = {N:8.1e}  L = {L:.4f}")
for N in (1e11, 1e12):   # extrapolation to sizes not in the experiment
    true = E + A / N**alpha
    pa = E + 10**c * N**w
    pb = 10**c2 * N**w2
    print(f"N = {N:.0e}: true L = {true:.4f}  prediction (a) = {pa:.4f}  prediction (b) = {pb:.4f}")
```

Output:

```
(a) E subtracted:      estimated alpha = 0.349 (true value 0.34), estimated A = 478.1
(b) E not subtracted:  apparent exponent = 0.087
   N =  1.0e+07  L = 3.4060
   N =  3.2e+07  L = 2.8559
   N =  1.0e+08  L = 2.4652
   N =  3.2e+08  L = 2.2051
   N =  1.0e+09  L = 2.0328
   N =  3.2e+09  L = 1.9296
   N =  1.0e+10  L = 1.8423
N = 1e+11: true L = 1.7640  prediction (a) = 1.7595  prediction (b) = 1.4130
N = 1e+12: true L = 1.7238  prediction (a) = 1.7211  prediction (b) = 1.1552
```