LLM Basics (1): Tokens and Language Models
(for high school and first-year university students)

From characters to "the probability of the next token"

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

Ver. 0.1 October 6, 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

When you type "Thank" on a smartphone, "you" appears as a suggestion. The phone is guessing **the word most likely to come next** from the words you have typed so far. In fact, LLMs (large language models) such as ChatGPT and Claude do essentially the same thing at their core: they "predict what comes next." The difference is that their predictions are far, far more accurate.

In "Deep Learning (4): Sequence Data and Transformers," we looked at the computations **inside** an LLM (Attention and the Transformer).[a] This "LLM Basics" series deals with what lies **outside** it. In Part 1, we look at the **entrance** of an LLM (how text is turned into numbers) and its **exit** (how numbers are turned into "the probability of the next word"). Then we build the simplest possible language model, one that "only counts," and by comparing it with an LLM we think about what makes LLMs so powerful.

[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 (conditional probability, exponential functions and logarithms, the summation sign $\sum$, and the dot product of vectors). The **mathematical foundations**, such as notation using matrices, computing gradients with partial derivatives, and a proof using the method of Lagrange multipliers, 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

- The size of the vocabulary (the number of kinds of tokens the model handles) is V, and the number of components (the dimension) of the vector representing a token is d.

- A text divided into tokens is written $x_1, x_2, \ldots, x_T$ (the subscript is the position).
- $p(b \mid a)$ is "the probability that b occurs given that a has occurred" (conditional probability). It is sometimes also written $P_A(B)$.
- log is the natural logarithm (the logarithm to base e).
- $N(a)$ and $N(a, b)$ are the numbers of times that the word a, and the word pair "b right after a," appear in the training text.

1 What is a language model?

1.1 Splitting the probability of a text into a product of "next-token probabilities"

A **language model** is a model that assigns a **probability** to a text (a sequence of tokens). The sequence "an earthquake occurred" is quite likely, so its probability is large; "occurred earthquake an" is almost never seen, so its probability is small. In this way, a language model expresses how "likely" a text is as a number.

By using the multiplication rule of probability, $P(A \cap B) = P(A)\,P(B \mid A)$, again and again, the probability of a whole text can be split as

$$p(x_1, x_2, \ldots, x_T) = p(x_1) \times p(x_2 \mid x_1) \times p(x_3 \mid x_1, x_2) \times \cdots \times p(x_T \mid x_1, \ldots, x_{T-1}),$$

that is, into **a product of "the probability of guessing the next token from the tokens so far"**.[1]

For example, for a three-word sentence,

$$p(\text{an}, \text{earthquake}, \text{occurred}) = p(\text{an}) \times p(\text{earthquake} \mid \text{an}) \times p(\text{occurred} \mid \text{an}, \text{earthquake}).$$

That is, "the probability that the first word is *an*" × "the probability that *earthquake* comes after *an*" × "the probability that *occurred* comes after *an earthquake*." It is like a word-guessing quiz in which you get one more hint at each stage, and you multiply together the probabilities of answering correctly at each stage.

> This splitting is **not an approximation; it is an equation that always holds**. The probability of any text can be written exactly as a product of "next-guess probabilities." Approximation enters only in how each "next-guess probability" is computed by a model. **An LLM is a device for computing these "next-guess probabilities."**

To **generate** text, the model chooses one next token according to these probabilities, adds it to the end, predicts the next one again, and repeats. This is the same as building a sentence by tapping the suggested word on your phone again and again.

1.2 The entrance and exit of an LLM

The flow of computation in an LLM, from entrance to exit, is shown in Figure 1.

The stage after the probabilities are obtained, in which one token is actually chosen from them (the lottery of Section 3.3), and the "temperature" used there, are treated in a separate note, "The stages where probability enters LLM computation, and 'temperature'."

[1] University textbooks write this with the product symbol $\prod$ (the multiplication version of the summation sign $\sum$) as $p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_{<t})$. Here $x_{<t}$ stands for $x_1, \ldots, x_{t-1}$ together. See Appendix A.1 for the derivation.

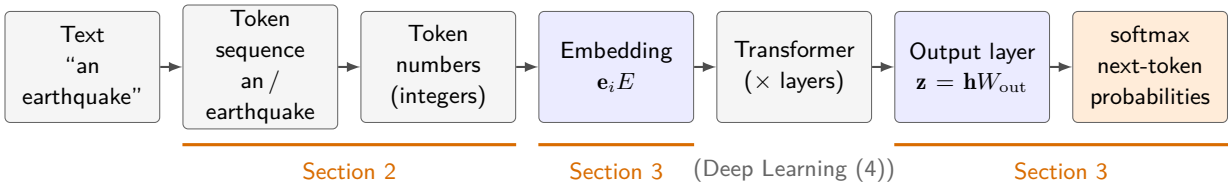


Figure 1 From the entrance to the exit of an LLM. This booklet deals with turning text into token numbers (Section 2) and with the embedding and output layer (Section 3). The Transformer in the middle was covered in “Deep Learning (4).” The expressions $\mathbf{e}_i E$ and $\mathbf{z} = \mathbf{h} W_{\text{out}}$ in the figure use matrix notation; they are explained in words in Section 3, and the formulas are collected in Appendix A.2.

2 Tokenization: dividing text into parts

2.1 Why neither characters nor words?

To feed text into a computer, we first have to turn it into a sequence of symbols of fixed kinds (**tokens**). How large the parts should be involves trade-offs.

- **Character units**: Only a few kinds of symbols are needed, but a text becomes a very long sequence of symbols. As we saw in “Deep Learning (4),” the computation of Attention grows as the square of the sequence length, so long sequences are a disadvantage.
- **Word units**: The sequences become short, but the number of kinds of words is enormous. Including inflected forms, new words and proper nouns, the vocabulary can never be fixed. Words not seen during training (**unknown words**) cannot be handled at all.

Compared to a toy building-block set, character units are like “only small basic blocks”: you can build anything, but assembling takes effort. Word units are like “special parts shaped like finished products”: you can build quickly, but you would need infinitely many kinds of parts, and you cannot build a shape you do not have a part for.

> Real LLMs use something in between: **subwords** (fragments of words). **Frequent words become a single token, and rare words are split into several fragments.** It is like keeping larger parts ready for commonly used shapes, and building rarely used shapes by combining small blocks.

2.2 Byte Pair Encoding (BPE)

A typical method for building a subword vocabulary is **Byte Pair Encoding** (BPE). The procedure is simple.

1. Start with each single character as a symbol.
2. Over the whole collection of training text (the **corpus**), find **the pair of symbols that most often appear next to each other**.
3. Merge that pair into one new symbol and add it to the vocabulary.
4. Repeat steps 2 and 3 until the vocabulary reaches a chosen size.

> This is similar to how we make abbreviations of words we use often. Just as we shorten “laboratory” to “lab” after writing it many times, **pairs that often appear together are merged into one first**.

Example Let us try it on a corpus consisting of just the following four words (program in Appendix B). The symbol “_”, which marks the end of a word, is attached to the end of each word.

Word	low	lower	newest	widest
Count	5	2	6	3

The initial vocabulary is the 11 symbols `_ d e i l n o r s t w`, and the whole corpus consists of 95 symbols (for example, low_ has 4 symbols and appears 5 times, giving 20). After 10 merge operations, we obtained the following.

Step	Merged pair	Count	Vocabulary size	Total tokens
1	e + s → es	9	12	86
2	es + t → est	9	13	77
3	est + _ → est_	9	14	68
4	l + o → lo	7	15	61
5	lo + w → low	7	16	54
6	n + e → ne	6	17	48
7	ne + w → new	6	18	42
8	new + est_ → newest_	6	19	36
9	low + _ → low_	5	20	31
10	w + i → wi	3	21	28

(When several pairs have the same count, the program chooses the one it found first.)

"es" and "est" are created first because they are contained in both newest (6 times) and widest (3 times), appearing 9 times in total. By adding 10 symbols to the vocabulary, the number of symbols in the whole corpus decreased from 95 to 28 (Figure 2).

The larger the vocabulary, the fewer tokens are needed to represent the same text. In exchange, the tables for the "embedding" and the "output layer," which we will see in Section 3, become larger. The vocabulary size is decided by balancing these two.

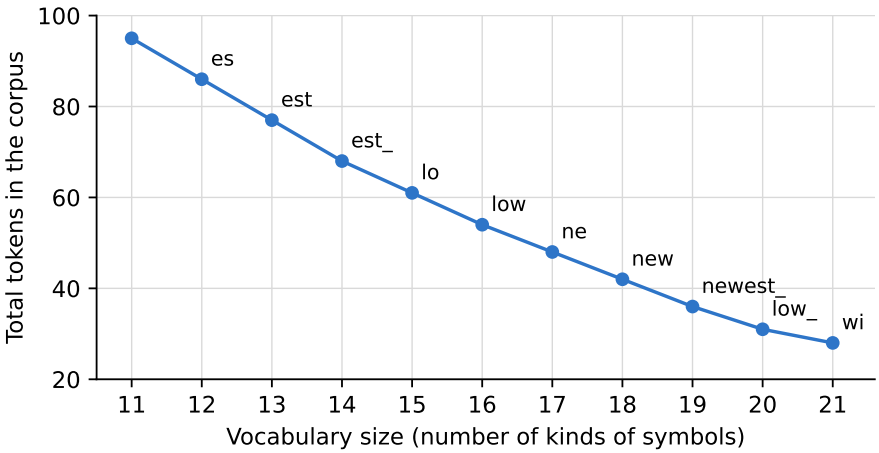


Figure 2 Vocabulary size and the number of tokens in the whole corpus as the BPE merge operations proceed. The text beside each point is the symbol newly created at that step.

2.3 Even unknown words can be split

In the end, each word is split as follows.

`low_` $\times 5$ `low e r _` $\times 2$ `newest_` $\times 6$ `wi d est_` $\times 3$

By applying the learned merge rules in the order they were learned, **even words that were not in the corpus can be split**.

`lowest` $\rightarrow$ `low est_` `newer` $\rightarrow$ `new e r _` `wider` $\rightarrow$ `wi d e r _`

"lowest" is a word the model has never seen, but it was split into the known parts "low" and "est_".

> Any word can be represented, at worst, by splitting it into single characters, so **it never happens that "this word is unknown, so it cannot be handled."** When a shape cannot be built from large parts, you simply go back to small blocks.

2.4 Real tokenizers

In real LLMs, BPE often starts not from characters but from **bytes** (256 kinds of numbers, 0–255) obtained by encoding the characters with a scheme called UTF-8. For example, the Japanese word 地震 ("earthquake") in UTF-8 is the 6 bytes

地 = `e5 9c b0`, 震 = `e9 9c 87`

(written in hexadecimal). Since every character in the world can be represented as a sequence of bytes, **any character of any language can always be converted into tokens**. Frequent characters and words are merged into a single token, while rare characters such as uncommon kanji are split into several tokens.

The vocabulary size differs from model to model; for example, Llama 3 has about 128,000 (128,256). Since the vocabulary is determined by the statistics of the corpus, texts with the same content may need different numbers of tokens in different languages. The number of tokens directly affects the amount of computation and the length of text that can be handled at once (Part 3).

3 Embedding and output layer: bridging token numbers and vectors

3.1 Turning token numbers into vectors: the embedding

Through tokenization, a text has become a sequence of integers (token numbers). But using the numbers as they are causes trouble.

> Think of the roll numbers in a class register. Student No. 5 is not "bigger" than student No. 3, and students No. 5 and No. 6 are not "similar." **A number is just a tag for telling people apart; its size and closeness have no meaning.** Token numbers are the same.

So we associate each token with a set of d numbers (a d-dimensional vector). For people, this would be like using a profile sheet listing "height, favorite subject, club activity, …" instead of a roll number. With profiles, we can compare "people who are similar" using numbers.

Concretely, we prepare a table E with V rows and d columns (the **embedding matrix**), and take its i-th row as the vector of token i. As a computation, this is **just taking out the i-th row of the table**.[2] The contents of the table E are determined by training, like the other

[2] This is sometimes written as a matrix product $\mathbf{e}_i E$, where $\mathbf{e}_i$ is a row vector whose i-th component is 1 and all others are 0. This is the notation in Figure 1 (Appendix A.2).

weights. People do not decide the items of the profile; "useful features" emerge naturally as a result of training.

In Llama 3 8B, $V = 128{,}256$ and $d = 4096$, so the number of values in E is

$$128{,}256 \times 4096 = 525{,}336{,}576 \quad \text{(about 525 million)}.$$

The property "words with similar meanings have vectors pointing in similar directions" (distributed representation), which we saw in "Introduction to How LLMs Work," refers to these row vectors of E (and to the representations in each layer that transform them).

3.2 Turning vectors into probabilities: the output layer

The last layer of the Transformer outputs a d-dimensional vector $\mathbf{h}$ for each position. This vector summarizes "the result of reading the context so far." From it, we compute a **score** for each of the V tokens in the vocabulary for "coming next."

For each token j we prepare a d-dimensional weight vector $\mathbf{w}_j$ and compute the score as a **dot product**:[3]

$$z_j = \mathbf{h} \cdot \mathbf{w}_j \qquad (j = 1, 2, \ldots, V).$$

The dot product expresses "how well the directions of two vectors agree," so tokens that "fit well" with the context $\mathbf{h}$ get higher scores. This score z_j is called a **logit**.

Scores can be negative, so they are not probabilities as they are. We therefore pass them through the **softmax function**:

$$p(\text{next token is } j) = \frac{e^{z_j}}{e^{z_1} + e^{z_2} + \cdots + e^{z_V}}.$$

Since e^z is always positive, every probability is positive, and they add up to exactly 1.

> In other words, the output layer is solving **a classification problem: choosing "the next one" from the V tokens of the vocabulary**. It has exactly the same form as the multi-class logistic regression we saw in "Supervised and Unsupervised Learning"; the only difference is that there are as many as 128,000 classes.

The output layer of Llama 3 8B also has $4096 \times 128{,}256$, about 525 million, weights. Together with the embedding, this is about 1.05 billion, about 13% of the whole model (about 8 billion). Some models save parameters by making the output-layer vector $\mathbf{w}_j$ the same as the j-th row of the embedding (GPT-2, for example).

How do the scores move during training? In training, we make the probability p_c given to the correct token larger by making the loss $L = -\log p_c$ (cross-entropy) smaller. The direction and amount by which each score z_j should be moved (the gradient) then takes a very simple form:[4]

$$(\text{gradient of score } z_j) = (\text{predicted probability } p_j) - (1 \text{ if correct, otherwise } 0).$$

> **The gradient is "prediction − correct answer."** The score of the correct token is moved upward (since $p_c - 1 < 0$), and the scores of the other tokens are moved downward by amounts proportional to their predicted probabilities. This has the same form as the gradient $p - y$

[3] If we form the $d \times V$ matrix W_{out} whose columns are the V vectors $\mathbf{w}_j$, the V scores can be written together as $\mathbf{z} = \mathbf{h}W_{\text{out}}$. This is the notation in Figure 1 (Appendix A.2).

[4] See Appendix A.3 for the derivation. Using the partial-derivative symbol, $\partial L / \partial z_j = p_j - [\,j = c\,]$.

of logistic regression in "Supervised and Unsupervised Learning," and from here the error is sent back into the Transformer by backpropagation.

3.3 Computing probabilities versus choosing according to them

The computation in Section 3.2 (dot products and the softmax function) uses no random numbers at all. For the same input, you get the same probabilities no matter how many times you compute. What you get here is, so to speak, "**a table of how likely each face of a die is to come up**."

Randomness enters at the stage of actually choosing one next token from these probabilities. This corresponds to a **lottery** (a roulette wheel). For example, when the bigram model of Section 4.1 chooses the word after "Ken," the probabilities and their running totals (cumulative probabilities) are as follows.

Next word	reads	gives	a
Probability	1/2	1/4	1/4
Cumulative	0.50	0.75	1.00

Imagine a roulette wheel in which the number line from 0 up to (but not including) 1 is divided into intervals as wide as the probabilities, each assigned to a word. We throw one uniform random number u onto it and choose the first word whose cumulative probability is greater than or equal to u. If $u = 0.72$, then $0.50 < 0.72 \leq 0.75$, so we choose "gives." **The width of the interval assigned to each word is exactly the probability of that word.** The generated examples in Section 4.2 were chosen in this way by `random.choices` in the program in Appendix B.

- Which word is chosen each time depends on the random number u (once u is fixed, the chosen word is uniquely determined).
- The fraction of times each word is chosen over many repetitions is determined by the probabilities. The actual counts, however, fluctuate. For example, if we choose 60 times, the expected number of times "reads" is chosen is 30, but the standard deviation is $\sqrt{60 \times \frac{1}{2} \times \frac{1}{2}} \approx 3.9$ (binomial distribution). The more times we repeat, the closer the fractions get to the probabilities (law of large numbers).

How likely each word is (the probability) is the result of a deterministic computation from the weights and the input, while **which word actually comes out depends on the lottery each time**. This is why the same question can get different answers. For "temperature," which adjusts how skewed the probabilities are, see the separate note "The stages where probability enters LLM computation, and 'temperature'."

4 A language model that only counts: the bigram model

To compare with LLMs, let us build the simplest possible language model.

4.1 Looking only at the previous word

Strictly speaking, the probability of the next word depends on **all** the words before it. Here we boldly decide to look **only at the previous word**:

$$p(x_t \mid x_1, \ldots, x_{t-1}) \;\approx\; p(x_t \mid x_{t-1}).$$

This is called a **bigram model** (a model of pairs). It is like a person who remembers only the last thing they heard, and says the next word using only that as a clue.

The probabilities are determined by **ratios of counts** in the training text:

$$p(b \mid a) = \frac{N(a,b)}{N(a)} = \frac{\text{number of times “}b\text{ right after }a\text{” appeared}}{\text{number of times }a\text{ appeared (as the previous word)}}.$$

Example We use the following six sentences as training data (program in Appendix B). We treat the period “.” as one word, and we put the mark <s> at the start of each sentence.

Ken reads a book .	Amy buys a book .
Ken gives Amy a book .	Amy writes a letter .
Ken reads a letter .	Amy gives Ken a letter .

The vocabulary has $V = 10$ words including “.”, and 18 kinds of word pairs appear. For example, “Ken” appears 4 times (as a previous word), and it is followed by “reads” twice, and by “gives” and “a” once each (“a” comes from “Amy gives Ken a letter”), so

$$p(\text{reads} \mid \text{Ken}) = \frac{2}{4} = 0.5, \qquad p(\text{gives} \mid \text{Ken}) = p(\text{a} \mid \text{Ken}) = \frac{1}{4} = 0.25.$$

The whole table of probabilities is shown in Figure 3. Of the $10 \times 10 = 100$ cells, only 18 are nonzero.

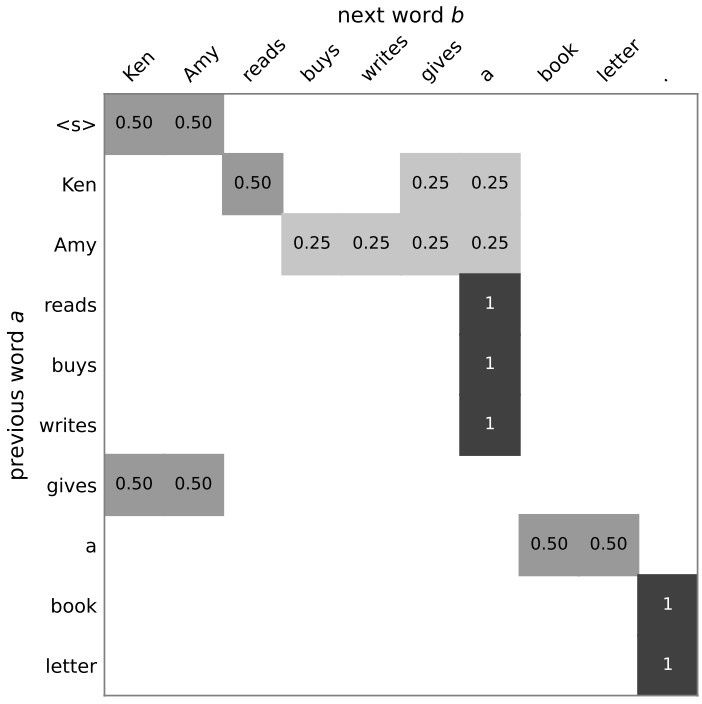


Figure 3 Probabilities $p(b \mid a)$ of the bigram model in the example. Rows are the previous word and columns are the next word; each row sums to 1. Blank cells have probability 0.

4.2 Generating text

Starting from <s> and choosing the next word one at a time by lottery according to these probabilities, we generated the following sentences.

Ken gives Ken reads a letter .
Amy writes a book .
Ken a letter .
Amy gives Ken gives Amy buys a book .

"Amy writes a book." is not in the training data, yet it is a grammatically correct sentence. A model that only counts was able to make a new sentence.

On the other hand, "Ken gives Ken reads a letter." and "Ken a letter." are **unnatural as whole sentences, even though every pair of adjacent words is a correct pair that appeared in the training data** ("gives Ken" and "Ken a" come from "Amy gives Ken a letter"). When choosing the word after "Ken," the model looks only at the immediately preceding "Ken"; it does not remember whether Ken is the one who does something or the one who receives something, or whether the sentence has already had a verb.

A bigram model is **a model in which connections between nearby words are correct, but the story goes wrong a little farther away**. To be text-like, a model must remember a longer context.

4.3 Counting gives "the most likely" answer

If "Ken" appeared 4 times and the next word was "reads" twice, estimating $p(\text{reads} \mid \text{Ken}) = 2/4$ seems natural. In fact, this is not just a hunch; it has a solid justification.

The frequency ratio $N(a, b)/N(a)$ is **the probability that maximizes the probability that the whole training data actually occurs (the likelihood)**; in other words, **the probability that minimizes the cross-entropy (the negative log-likelihood)**. This is called **maximum likelihood estimation**.

Let us check this in a simple case. Let q be the probability that the word after "Ken" is "reads," and for simplicity suppose there are only two possibilities for the next word: "reads" or "something else." Out of 4 times, 2 were "reads" and 2 were "something else," so the probability that this data occurs is

$$f(q) = q^2(1-q)^2.$$

Varying q to maximize $f(q)$ (by differentiation: $f'(q) = 2q(1-q)(1-2q) = 0$), the maximum is at $q = 2/4 = 1/2$. **The value obtained by counting and dividing is the probability that best explains the data.** The same conclusion holds in the general case with many kinds of words (Appendix A.4, method of Lagrange multipliers).

Training the neural-network way Let us confirm the same thing using the same method as a neural network. Instead of counting probabilities directly, we prepare V scores (logits) z_{ab} for each previous word a and set

$$p(b \mid a) = \frac{e^{z_{ab}}}{(\text{sum of } e^{z} \text{ over the row of the previous word } a)}.$$

This corresponds to the smallest possible "neural language model," with only the "embedding" and "output layer" of Section 3 and no Transformer.

Starting with all scores at 0 (every word equally likely), we make the cross-entropy smaller little by little by gradient descent (Appendix B). It is like turning a volume knob a little at a

time to find just the right sound. By the same computation as in Section 3.2, the gradient of the score z_{ab} is

$$N(a) \times p(b \mid a) - N(a, b)$$

(Appendix A.5). This becomes 0 when $p(b \mid a) = N(a, b)/N(a)$, which indeed agrees with the frequency ratio.

Iterations	p(reads | Ken)	p(gives | Ken)	p(book | Ken)	Mean loss
0	0.100	0.100	0.100	2.303
10	0.437	0.172	0.031	0.801
100	0.493	0.243	0.003	0.623
1000	0.499	0.249	0.000	0.608
3000	0.500	0.250	0.000	0.607
Counted frequency	0.500	0.250	0	—

(The mean loss is the cross-entropy per word.)

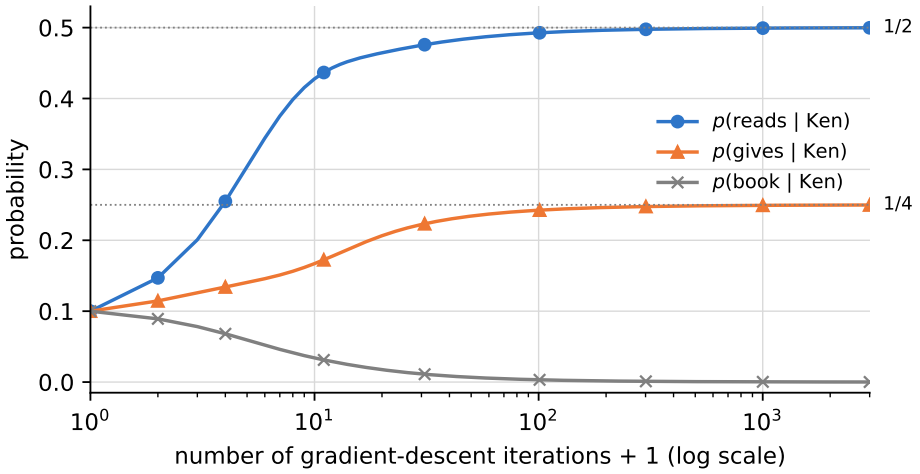


Figure 4 Training the bigram model by gradient descent. The probabilities approach the counted frequencies (dotted lines). p(a | Ken) follows exactly the same curve as p(gives | Ken), so it is omitted.

The initial loss 2.303 is $\log 10$, the state in which all 10 words are equally likely. Gradient descent **approaches, little by little, the answer that counting gives in one step** (Figure 4).

> Training an LLM is exactly the same in that it **makes the cross-entropy smaller by gradient descent**. The difference is that the model representing the probabilities is not a $V \times V$ table but a Transformer that looks at the whole context.

4.4 Limits of n-gram models

Extending to a model that looks at the previous $n-1$ words (an **n-gram model**) reduces failures such as "Ken gives Ken reads." But there are the following problems.

1. **The table grows explosively**: There are V^{n-1} combinations of the previous $n-1$ words, and the table has V^n cells. With $V = 128{,}256$, even a bigram table has $V^2 \approx 1.6 \times 10^{10}$ cells, and a trigram (three-word) table has $V^3 \approx 2.1 \times 10^{15}$ cells.
2. **Most cells are 0**: Even in the example, 82 of the 100 cells were 0. The larger n is, the more the combinations become ones that never appeared in the training data.

3. **Knowledge cannot be shared between similar words**: In the example, "Ken" and "Amy" are both personal names, but in the table they are unrelated, separate rows. The statistics after "Ken" never help predict what comes after "Amy." (For example, the data show "Ken reads" but never "Amy reads," so the model gives "Amy reads" probability 0.)

The third point is exactly the roll-number story again. In the table, Ken and Amy are just "people with different numbers"; the fact that "both are people's names" cannot be seen.

Neural language models solved these problems as follows.

- Since words are represented by embedding vectors (profiles), **similar words get similar vectors and can share statistics** (easing problems 3 and 2).
- Since the context is not looked up in a table but computed by referring to the needed positions with Attention, **contexts of thousands of tokens or more can be handled without building a table** (solving problem 1). The roughly 8 billion parameters of Llama 3 8B are fewer than the cells of a bigram table (about 16 billion).

5 Evaluating language models: perplexity

5.1 Definition

The quality of a language model is measured by **how high a probability it gives to text not used in training**. The measure for this is **perplexity** (PPL; "perplexity" means bewilderment):

$$\mathrm{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log p(x_t \mid x_1, \ldots, x_{t-1})\right).$$

$\exp(x)$ means e^x. The quantity in the parentheses is **the cross-entropy per token**, which is exactly the loss being minimized in training.

Perplexity can be read as **how many choices the model is effectively hesitating between at each position, like a multiple-choice quiz**. For example, a model that predicts each of K candidates with probability $1/K$ has $\mathrm{PPL} = K$ (K choices).

Indeed, if the model gives probability $1/K$ to the correct answer at every position, then $-\log(1/K) = \log K$, so

$$\mathrm{PPL} = \exp(\log K) = K.$$

In general, perplexity is the reciprocal of "the average (geometric mean) of the probabilities given to the correct answers" (Appendix A.6). The higher the probability given to the correct answers, the smaller the perplexity, meaning less hesitation.

5.2 Example

We computed the perplexity of several sentences with the bigram model of Section 4.1 (Appendix B).

Sentence	In the training data?	PPL	Smoothed ($k = 0.1$)
Amy buys a letter .	No (all word pairs known)	1.741	2.224
Ken gives Amy a letter .	No (all word pairs known)	2.245	2.680
book gives Amy a letter .	No (contains an unseen pair)	∞	6.826
(Whole training data)	Yes	1.834	—
(Reference) 10 words equally likely	—	10	—

Even for sentences not in the training data, if all word pairs are known, the perplexity is small (hesitating among only about two choices).

However, "book gives Amy a letter." contains a pair not in the training data: "book" coming at the start of a sentence (<s> → book). Its probability is 0, so the probability of the whole sentence is 0 (if even one factor in a product is 0, the whole product is 0), and the perplexity is infinite. The model decides that a pair it has never seen "can never occur."

To avoid this, there is a method of adding a small count k to every pair (**additive smoothing**):

$$p(b \mid a) = \frac{N(a, b) + k}{N(a) + kV}.$$

kV is added to the denominator to keep the total equal to 1, because k was added to each of the V words.

With $k = 0.1$, the perplexity of "book gives Amy a letter." became 6.826. It is no longer infinite, but it is about 2.5 to 3 times the smoothed values for the other two sentences and not far from the case of 10 equally likely words (10 choices): for this model, the sentence is still quite unexpected.

> In neural language models, the output of the softmax function is always positive ($e^z > 0$), so **a probability is never exactly 0**. Small probabilities naturally remain even for combinations never seen.

5.3 The loss of an LLM and perplexity

In "Introduction to How LLMs Work," we said that if a model guessed completely at random with the Llama 3 vocabulary, the loss would be $\log 128{,}256 \approx 11.76$. This is a perplexity of about 128,000, a state of hesitating among the entire vocabulary. If the loss drops to 2, the perplexity is $e^2 \approx 7.4$; at 1.5, about 4.5.

> **Learning to narrow a 128,000-way multiple-choice quiz down to effectively about 4 to 7 choices**: that is what training achieves.

6 Summary

> 1. A language model assigns probabilities to texts, and the probability of a text can be decomposed exactly into a product of "next-token probabilities." An LLM computes the probability at each of these stages.
> 2. Text is divided into subwords (tokens) by methods such as BPE. BPE is a simple method that repeatedly merges pairs of symbols that often appear next to each other, and it can split even unknown words into known parts.
> 3. A token number becomes a vector by taking out a row of the embedding table E, and

the vector from the last layer becomes the probability of the next token through the output layer (dot products) and the softmax function. This is a V-class classification problem, and its gradient has the form "prediction − correct answer."

4. The computation of the probabilities itself is deterministic; randomness enters only at the lottery stage of choosing one token according to the probabilities.
5. Even a bigram model that only counts, looking at just the previous word, can generate text. Counting and dividing is exactly maximum likelihood estimation, and gradient descent also approaches the same answer.
6. In n-gram models the table grows explosively, and statistics cannot be shared between similar words. Neural language models solved this with embeddings and Attention.
7. Language models are evaluated by perplexity (the exponential of the cross-entropy). It expresses "how many choices the model is effectively hesitating between."

Coming next Part 2 deals with **pre-training**, in which this model is trained on trillions of tokens of text. The loss decreases in a surprisingly regular way (following power laws) with respect to the model size, the amount of data and the amount of computation.

Appendix A Mathematical explanations

This appendix uses matrices and vectors and partial derivatives (first-year university calculus and linear algebra).

Notation Token vectors are treated as **row vectors** (matrices with one row), and matrices are multiplied from the right, as in $\mathbf{x}W$. $\mathbf{e}_i$ is the row vector whose i-th component is 1 and all others are 0 (a **one-hot vector**). $x_{<t}$ stands for $x_1, \ldots, x_{t-1}$ together. $\prod$ is the product symbol, and $[\,j = c\,]$ equals 1 if $j = c$ and 0 otherwise. The superscript T denotes the transpose.

A.1 The multiplication rule of probability (chain rule)

For two events, $p(A, B) = p(A)\, p(B \mid A)$. Using this repeatedly, for three events

$$p(A, B, C) = p(A, B)\, p(C \mid A, B) = p(A)\, p(B \mid A)\, p(C \mid A, B),$$

and in general

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_{<t})$$

(the factor for $t = 1$ is read as the unconditional $p(x_1)$). This is an **identity** that can be proved by mathematical induction on T, and it contains no approximation.

A.2 Matrix notation for the embedding and output layer

The embedding matrix E is $V \times d$, and with a one-hot vector $\mathbf{e}_i$ $(1 \times V)$,

$$\mathbf{e}_i E = (\text{the } i\text{-th row of } E).$$

The output layer uses the $d \times V$ matrix W_{out} (whose j-th column is $\mathbf{w}_j$ in the main text):

$$\mathbf{z} = \mathbf{h} W_{\text{out}} \quad (1 \times V), \qquad p(x_{t+1} = j \mid x_{\le t}) = \frac{e^{z_j}}{\sum_{k=1}^{V} e^{z_k}}.$$

In models that share the weights of the embedding and the output layer (GPT-2, for example), $W_{\text{out}} = E^{\mathsf{T}}$.

A.3 Gradient of the output layer

If the correct token is c, the loss is

$$L = -\log p_c = -z_c + \log \sum_k e^{z_k}.$$

Taking the partial derivative with respect to the logit z_j, the first term gives $-[\,j = c\,]$ and the second term gives $e^{z_j} / \sum_k e^{z_k} = p_j$, so

$$\frac{\partial L}{\partial z_j} = p_j - [\,j = c\,], \qquad \text{or in vector form} \quad \frac{\partial L}{\partial \mathbf{z}} = \mathbf{p} - \mathbf{e}_c.$$

That is, "predicted probabilities − the one-hot vector of the correct answer." This has the same form as the gradient $p - y$ of two-class logistic regression.

A.4 The frequency ratio is the maximum likelihood estimate

The negative log-likelihood of the whole training data is

$$L = -\sum_{(a,b)} N(a,b) \log p(b \mid a).$$

For each previous word a, we minimize this under the condition $\sum_b p(b \mid a) = 1$. Writing $p_b = p(b \mid a)$ and using the method of Lagrange multipliers, we take the partial derivative of

$$F = -\sum_b N(a,b) \log p_b + \lambda\Big(\sum_b p_b - 1\Big)$$

with respect to p_b and set it to 0: $-N(a,b)/p_b + \lambda = 0$, that is, $p_b = N(a,b)/\lambda$. From $\sum_b p_b = 1$ we get $\lambda = \sum_b N(a,b) = N(a)$, and therefore

$$p(b \mid a) = \frac{N(a,b)}{N(a)}.$$

Since $-\log$ is a convex function, this is the minimum. The two-choice case in Section 4.3 of the main text (maximizing $q^2(1-q)^2$) is a special case of this.

A.5 Gradient when training the table of logits by gradient descent

Let $p(b \mid a) = e^{z_{ab}} / \sum_{b'} e^{z_{ab'}}$ with the logits z_{ab} of the previous word a. For each of the $N(a)$ occasions on which the previous word is a, the result of A.3 gives $\partial(-\log p_c)/\partial z_{ab} = p(b \mid a) - [\,b = c\,]$. Adding these over the $N(a)$ occasions, the sum of $[\,b = c\,]$ becomes the number of times the correct answer was b, namely $N(a,b)$, so

$$\frac{\partial L}{\partial z_{ab}} = N(a)\, p(b \mid a) - N(a,b).$$

The program in Appendix B divides this by $N(a)$, multiplies by the learning rate $\eta = 1$, and subtracts the result from z_{ab}.

A.6 Perplexity and the geometric mean

If the probabilities given to the correct answers at each position are $p_1, \ldots, p_T$, then

$$\mathrm{PPL} = \exp\Big(-\frac{1}{T}\sum_{t}\log p_t\Big) = \Big(\prod_{t=1}^{T} p_t\Big)^{-1/T},$$

which is the reciprocal of "the geometric mean of the probabilities given to the correct answers." If $p_t = 1/K$ at every position, $\mathrm{PPL} = K$.

If the logarithm is taken to base 2, the cross-entropy expresses "how many bits of information are missing per token." The relation between prediction and information (compression) is treated in Part 2.

Appendix B Python code (standard library only)

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

B.1 BPE (Section 2)

```
# Build a token vocabulary with Byte Pair Encoding (BPE) (standard library only)
from collections import Counter

# Words and their counts. "_" is added at the end of each word to mark the word boundary
corpus = {"low": 5, "lower": 2, "newest": 6, "widest": 3}
words = {tuple(w) + ("_",): n for w, n in corpus.items()}

def n_tokens(words):
    return sum(len(w) * n for w, n in words.items())

vocab = sorted({c for w in words for c in w})
merges = []  # learned merge rules (the order matters)
print("Initial vocabulary:", " ".join(vocab), f"({len(vocab)} symbols)")
print("Total tokens:", n_tokens(words))

for step in range(1, 11):
    pairs = Counter()  # count pairs of adjacent symbols
    for w, n in words.items():
        for a, b in zip(w, w[1:]):
            pairs[a, b] += n
    (a, b), cnt = max(pairs.items(), key=lambda kv: kv[1])
    new = {}  # merge the most frequent pair into one symbol
    for w, n in words.items():
        out, i = [], 0
        while i < len(w):
            if i + 1 < len(w) and w[i] == a and w[i + 1] == b:
                out.append(a + b); i += 2
            else:
                out.append(w[i]); i += 1
        new[tuple(out)] = n
    words = new
    vocab.append(a + b)
    merges.append((a, b))
    print(f"{step:2d}: {a} + {b} -> {a+b:8s}({cnt:2d} times) "
          f"vocab {len(vocab):2d}  total tokens {n_tokens(words):3d}")

print("Final segmentation:")
for w, n in words.items():
```

```
    print(f"  {' '.join(w):20s} x{n}")

def encode(word):
    """Segment a word by applying the learned merge rules in the order they were learned"""
    w = list(word) + ["_"]
    for a, b in merges:
        i = 0
        while i < len(w) - 1:
            if w[i] == a and w[i + 1] == b:
                w[i:i + 2] = [a + b]
            else:
                i += 1
    return w

for word in ("lowest", "newer", "wider"):  # words not in the training data
    print(f"{word:7s} -> {' '.join(encode(word))}")
```

Output:

```
Initial vocabulary: _ d e i l n o r s t w (11 symbols)
Total tokens: 95
 1: e + s -> es      ( 9 times) vocab 12  total tokens  86
 2: es + t -> est     ( 9 times) vocab 13  total tokens  77
 3: est + _ -> est_    ( 9 times) vocab 14  total tokens  68
 4: l + o -> lo      ( 7 times) vocab 15  total tokens  61
 5: lo + w -> low     ( 7 times) vocab 16  total tokens  54
 6: n + e -> ne      ( 6 times) vocab 17  total tokens  48
 7: ne + w -> new     ( 6 times) vocab 18  total tokens  42
 8: new + est_ -> newest_ ( 6 times) vocab 19  total tokens  36
 9: low + _ -> low_    ( 5 times) vocab 20  total tokens  31
10: w + i -> wi      ( 3 times) vocab 21  total tokens  28
Final segmentation:
  low_                 x5
  low e r _            x2
  newest_              x6
  wi d est_            x3
lowest  -> low est_
newer   -> new e r _
wider   -> wi d e r _
```

B.2 The bigram model and perplexity (Sections 4 and 5)

```
# A counting-only language model (bigram model) and perplexity (standard library only)
import math, random
from collections import Counter, defaultdict

train = ["Ken reads a book .",
         "Amy buys a book .",
         "Ken gives Amy a book .",
         "Amy writes a letter .",
         "Ken reads a letter .",
         "Amy gives Ken a letter ."]
sents = [["<s>"] + s.split() for s in train]  # <s> marks the start of a sentence
vocab = sorted({w for s in sents for w in s})
V = len(vocab) - 1  # number of words to be predicted (excluding <s>)

# (1) Training = counting pairs of adjacent words
count = defaultdict(Counter)
for s in sents:
```

```
    for a, b in zip(s, s[1:]):
        count[a][b] += 1

def p(b, a, k=0.0):
    """Probability that b follows the previous word a (additive smoothing if k>0)"""
    return (count[a][b] + k) / (sum(count[a].values()) + k * V)

print(f"Vocabulary {V} words, word pairs learned: {sum(len(c) for c in count.values())} kinds")
for a in ("Ken", "a"):
    row = ", ".join(f"{b} {p(b, a):.3f}" for b, _ in count[a].most_common())
    print(f"p(.|{a}) : {row}")

# (2) Text generation: choose the next word one at a time according to the probabilities
random.seed(11)
for _ in range(4):
    w, out = "<s>", []
    while w != ".":
        nxt = list(count[w])
        w = random.choices(nxt, weights=[count[w][b] for b in nxt])[0]
        out.append(w)
    print("Generated:", " ".join(out))

# (3) Perplexity: exp(average cross-entropy per word)
def ppl(sentence, k=0.0):
    s = ["<s>"] + sentence.split()
    nll = 0.0
    for a, b in zip(s, s[1:]):
        q = p(b, a, k)
        if q == 0:
            return math.inf
        nll -= math.log(q)
    return math.exp(nll / (len(s) - 1))

tests = ["Amy buys a letter .",        # not in the training data (all pairs known)
         "Ken gives Amy a letter .",
         "book gives Amy a letter ."]  # contains an unseen pair (<s> -> book)
for t in tests:
    print(f"{t:26s} PPL = {ppl(t):7.3f}  smoothed (k=0.1) {ppl(t, 0.1):7.3f}")
print("PPL of the whole training data =",
      round(math.exp(sum(-math.log(p(b, a)) for s in sents for a, b in zip(s, s[1:]))
                     / sum(len(s) - 1 for s in sents)), 3))
print("PPL of the uniform distribution (every word 1/V) =", V)
```

Output:

```
Vocabulary 10 words, word pairs learned: 18 kinds
p(.|Ken) : reads 0.500, gives 0.250, a 0.250
p(.|a) : book 0.500, letter 0.500
Generated: Ken gives Ken reads a letter .
Generated: Amy writes a book .
Generated: Ken a letter .
Generated: Amy gives Ken gives Amy buys a book .
Amy buys a letter .        PPL =   1.741  smoothed (k=0.1)   2.224
Ken gives Amy a letter .   PPL =   2.245  smoothed (k=0.1)   2.680
book gives Amy a letter .  PPL =     inf  smoothed (k=0.1)   6.826
PPL of the whole training data = 1.834
PPL of the uniform distribution (every word 1/V) = 10
```

B.3 Training the bigram model by gradient descent (Section 4.3)

```
# Train the bigram model "the neural-network way":
# adjusting the table of logits Z by gradient descent brings the probabilities to the counted frequencies
import math
from collections import Counter, defaultdict

train = ["Ken reads a book .", "Amy buys a book .",
         "Ken gives Amy a book .", "Amy writes a letter .",
         "Ken reads a letter .", "Amy gives Ken a letter ."]
sents = [["<s>"] + s.split() for s in train]
prev = sorted({a for s in sents for a in s[:-1]})  # words that can be the previous word
nxt = sorted({b for s in sents for b in s[1:]})    # words to be predicted (V words)
count = defaultdict(Counter)
for s in sents:
    for a, b in zip(s, s[1:]):
        count[a][b] += 1

def softmax(z):
    m = max(z); e = [math.exp(v - m) for v in z]; t = sum(e)
    return [v / t for v in e]

Z = {a: [0.0] * len(nxt) for a in prev}  # initial values: all 0 (uniform distribution)
eta = 1.0
watch = ["reads", "gives", "a", "book"]
for step in range(3001):
    if step in (0, 10, 100, 1000, 3000):
        P = softmax(Z["Ken"])
        L = -sum(n * math.log(softmax(Z[a])[nxt.index(b)])
                 for a in prev for b, n in count[a].items())
        N = sum(sum(c.values()) for c in count.values())
        print(f"step {step:4d}  mean loss {L/N:.4f}  p(.|Ken): "
              + " ".join(f"{w} {P[nxt.index(w)]:.3f}" for w in watch))
    for a in prev:  # gradient dL/dz_ab = N_a p(b|a) - N_ab
        P = softmax(Z[a]); Na = sum(count[a].values())
        for j, b in enumerate(nxt):
            Z[a][j] -= eta * (Na * P[j] - count[a][b]) / Na
print("Counted frequencies p(.|Ken): " + " ".join(
    f"{w} {count['Ken'][w] / sum(count['Ken'].values()):.3f}" for w in watch))
```

Output:

```
step    0  mean loss 2.3026  p(.|Ken): reads 0.100 gives 0.100 a 0.100 book 0.100
step   10  mean loss 0.8011  p(.|Ken): reads 0.437 gives 0.172 a 0.172 book 0.031
step  100  mean loss 0.6228  p(.|Ken): reads 0.493 gives 0.243 a 0.243 book 0.003
step 1000  mean loss 0.6081  p(.|Ken): reads 0.499 gives 0.249 a 0.249 book 0.000
step 3000  mean loss 0.6070  p(.|Ken): reads 0.500 gives 0.250 a 0.250 book 0.000
Counted frequencies p(.|Ken): reads 0.500 gives 0.250 a 0.250 book 0.000
```